

Experiments on Parallel Matrix-Vector Product

S. Riyavong[†]

CERFACS Working Note WN/PA/03/127

Abstract

The matrix-vector product is one of the most important computational components of Krylov methods. We design a data structure for a distributed matrix to compute the matrix-vector product efficiently on parallel machines using MPI. Experiments with GMRES without preconditioner have been performed with sparse matrices selected from the Rutherford-Boeing Collection. Numerical results on a COMPAQ and a cluster of PC's are analysed and discussed.

Key words. parallel matrix-vector product, parallel GMRES method, data structure.

1 Introduction

Krylov subspace methods are currently ubiquitous as a tool for solving linear systems, especially for very large sparse matrices where direct methods are prohibitive. One of the principal computational components of the methods is the matrix-vector multiplication. This operation can be parallelized in the Single Program Multiple Data model (SPMD) [2], by partitioning the rows of a sparse matrix evenly into P blocks, where P is the number of processors. Each block should have a minimum number of columns in common with other blocks in order to reduce the communication cost. To achieve this we use the partitioner PaToH [9] to partition a matrix. Then each block is mapped onto a processor, on which the corresponding local matrix-vector multiplication takes place. Data exchange, if any, among the processors is performed using the MPI library. In this working note, which is a sequel to [6], we describe a data structure for storing both local matrix entries, including the associated local vectors, and communication information. The data structure is described in Section 2. The implementation and numerical results are given in Section 3. Finally, some conclusions are discussed in Section 4.

2 Local Data Structure

We compute

$$y = Ax$$

where A is an $n \times n$ matrix, x and y are vectors of size n , in a parallel distributed environment. To evenly distribute the matrix over the processors, we partition it with PaToH [9] and we permute its rows and columns, using the results from the partitioner, such that the partitioned matrix forms a block structure, see [6]. The structure is such that as many nonzero entries of the sparse matrix as possible are contained within diagonal blocks.

The data structure we describe here is designed for this kind of partitioned matrix. In distributing the matrix and vectors among processors for computing with Krylov methods, it is advantageous to distribute them row-wise, i.e. the same components of vectors x and y and rows of the matrix A are assigned to the same processor as illustrated in Figure 1. To be more specific, we denote by $I = \{1, 2, \dots, n\}$ the set of row indices which is divided into subsets $\{I_1, I_2, \dots, I_P\}$. The entries in I_K are the components of vectors and indices of rows of the matrix that are assigned to processor K . We will call the corresponding variables *internal*. In order to perform a local matrix-vector product, the local vector x is composed of the internal variables and the *external* variables that correspond to columns of the local matrix that are non-internal. Communication is needed for the external variables. This will be illustrated later.

The local data structure of a distributed matrix includes the indices and values of the nonzero entries of the local matrix. The entries of the local matrix can be stored in any of the formats described in [7]. We use the Compressed Sparse Row (CSR) format for general local matrices in this work. In addition,

[†]CERFACS, 42 Avenue G. Coriolis, 31057 Toulouse Cedex, France

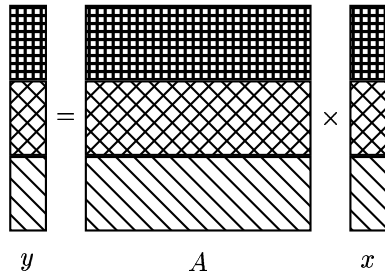


Figure 1: Distributing of matrix and vectors to processors as defined by shading

we have to store communication information. The communication information specifies which data has to be exchanged (sent or received) with which processor. To illustrate this, consider Figure 2, where the block-structured matrix and its associated vectors are distributed among four processors. This matrix has been partitioned and its rows and columns have been permuted to form blocks. The rows and columns are indexed such that their indices are in natural order, i.e. $1, 2, \dots, n$. Thus we have $I_1 = \{1, 2, 3\}$, $I_2 = \{4, 5, 6, 7\}$, $I_3 = \{8, 9, 10\}$, and $I_4 = \{11, 12\}$, holding the indices of x and y and indices of rows of A that are assigned to processors 1, 2, 3, and 4, respectively. Denoting capital letters X and Y for local vectors, we get, for example, the local matrix and the local vectors assigned to processor 2 as shown in Figure 3. The mapping between the global indices and the local indices is as below.

global x	2	3	4	5	6	7	8	10
local X	1	2	3	4	5	6	7	8
global y	4	5	6	7				
local Y	1	2	3	4				
x_{ind}	3	4	5	6				

where x_{ind} stores the local indices of X corresponding to internal variables. This means that processor 2 has to receive from the neighbours the data, corresponding to the external variables, for X_1 , X_2 , X_7 , and X_8 to compute the local matrix-vector product. The local rectangular matrix is of size 4×8 and its entries are stored in CSR format as

	1	2	3	4	5	6	7	8	9	10	11	12
rowptr	1	4	7	10	13							
colidx	3	4	6	2	4	7	3	5	8	1	4	6
val	6	13	-9	4	-8	37	49	-5	15	27	-4	51

The information relating to the communication is stored in six variables. This storage format is similar to the compressed sparse storage for matrices.

- **sndproc**: The list of the neighbouring processor IDs to which processor 2 sends X_3 , X_4 , X_5 , X_6 .
- **sndptr**: The pointer to **sndidx** such that $X(\text{sndidx}(\text{sndptr}(i):\text{sndptr}(i+1)-1))$ are sent to neighbouring processor **sndproc**(i).
- **sndidx**: The list of **colidx** that are shared with neighbouring processors.

	1	2	3	4	5	6
sndproc	1	3	4			
sndptr	1	4	6	7		
sndidx	3	4	6	5	6	3

- **rcvproc**: The list of IDs of neighbouring processors from which processor 2 receives the data for X_1 , X_2 , X_7 , X_8 .
- **rcvptr**: The pointer to **rcvidx** such that $X(\text{rcvidx}(\text{rcvptr}(i):\text{rcvptr}(i+1)-1))$ are received from neighbouring processor **rcvproc**(i).
- **rcvidx**: The list of **colidx** where processor 2 has to store data from the neighbouring processors.

	1	2	3	4
rcvproc	1	3		
rcvptr	1	3	5	
rcvidx	1	2	7	8

This data structure is convenient for computing a local matrix-vector product on a parallel machine with MPI as the message passing interface. The steps in the calculation are shown in Algorithm 1.

Algorithm 1 Parallel Matrix-Vector Product

1. **for** $i=1$ to $\#sndproc$ **do**
 2. $buffer \leftarrow X(sndidx(sndptr(i):sndptr(i+1)-1))$
 3. MPI_SEND: $buffer \rightarrow sndproc(i)$
 4. **end for**
 5. **for** $i=1$ to $\#rcvproc$ **do**
 6. MPI_RECV: $buffer \leftarrow rcvproc(i)$
 7. $X(rcvidx(rcvptr(i):rcvptr(i+1)-1)) \leftarrow buffer$
 8. **end for**
 9. $Y \leftarrow A_{loc}X$
-

To send the data to the neighbouring processor i we store the data in a buffer and perform one communication with this neighbour. In the next section, we describe how this data structure can be used to implement the parallel matrix-vector product in the GMRES library code [3].

3 Implementation and Experimental Results

When solving a sparse system of linear equations with the GMRES [8] method in a distributed computing environment, we have to compute matrix-vector products as well as scalar products of two vectors. In the matrix-vector product, the current processor exchanges data with its neighbours and then computes the product locally. The scalar product can be formed by computing the local inner product, i.e. corresponding to the internal variables, then summing contributions from all processors using MPI_ALLREDUCE.

Due to the reverse communication design of the GMRES library [3], we can develop a matrix-vector product routine independent from the library. The purpose of the experiments is to determine the parallel performance of the matrix-vector products and of the dot products. We use GMRES without a preconditioner and perform a fixed number of 500 iterations. For orthogonalization of the basis vectors of the Krylov subspace, as investigated in [4], ICGS is the method of choice for GMRES in a parallel distributed environment. Thus we use it throughout the experiments. With ICGS we combine the communication of many dot products. The local dot products can be gathered into one matrix-vector operation using a BLAS 2 routine for efficiency. At this stage, all computations are synchronized with MPI_ALLREDUCE. The matrices in the experiment are of the order of a few tens of thousands and are from the Rutherford-Boeing Collection [1].

We have tested our implementation on two platforms: a COMPAQ and a cluster of PC's. The results are shown in Tables 1 and 2, respectively. Some parameters in the tables are described below.

- The columns of $\max\#nloc$ and $\max\#nnz$ are the maximum number of rows and the maximum number of nonzero entries, respectively, for a local matrix. In this work, we partition the matrices for 2, 4, and 8 processors. For a balanced partitioning, both $\#nloc$'s and $\#nnz$'s of all local matrices should be approximately the same and when we double the number of processors, they should reduce in half. When these properties are found in the local matrices, we say that the load balance among the processors is good. The tables show that for all matrices, except `poli_large.rb`, both $\#nloc$ and $\#nnz$ are approximately halved when the number of processors are doubled, which indicates a good load balance. The size of $\#nloc$ has a direct effect on the computing time of a local dot product. Since the number of computations for a local dot product is proportional to $\#nloc$, we approximate linearly the relation between the elapsed time for computing it and $\#nloc$. In the case of $\#nnz$, we find that five matrices `ex19.rb`, `garon02.rb`, `lhr10c.rb`, `memplus.rb`, and `olafu.rb` are partitioned with noticeable load imbalance while the others show quite a balanced partitioning of nonzero entries. The number of computation for matrix-vector product is linear with respect to $\#nnz$.

		1	2	3	4	5	6	7	8	9	10	11	12	
proc 1	y_1	1		38		14								x_1
	y_2	2	5		3									x_2
	y_3			-7				86						x_3
proc 2	y_4				6	13		-9						x_4
	y_5			4		-8			37					x_5
	y_6				49		-5				15			x_6
	y_7		27			-4		51						x_7
proc 3	y_8					21			14		87		73	x_8
	y_9							32		16				x_9
	y_{10}										11			x_{10}
proc 4	y_{11}									23		21		x_{11}
	y_{12}				67						47		19	x_{12}

Figure 2: Partitioning a matrix for four processors.

$$\begin{array}{|c|} \hline Y_1 \\ \hline Y_2 \\ \hline Y_3 \\ \hline Y_4 \\ \hline \end{array} = \begin{array}{|c|c|c|c|c|c|c|c|} \hline & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline & & & 6 & 13 & & -9 & & \\ \hline & & 4 & & -8 & & & 37 & \\ \hline & & & 49 & & -5 & & & 15 \\ \hline & 27 & & & -4 & & 51 & & \\ \hline \end{array} \times \begin{array}{|c|} \hline X_1 \\ \hline X_2 \\ \hline X_3 \\ \hline X_4 \\ \hline X_5 \\ \hline X_6 \\ \hline X_7 \\ \hline X_8 \\ \hline \end{array}$$

Figure 3: The local matrix and local vectors for processor 2.

matrix info				#proc	elapsed time (sec)			
matrix	n	max#nloc	max#nnz		matvec	msg exch	dotprod	MPI_ALLREDUCE
bayer02.rb	13935	7020	32160	2	0.448	0.085	15.531	1.366
		3519	16184	4	0.171	0.435	5.262	0.924
		1775	8216	8	0.065	0.085	1.611	1.290
epb1.rb	14734	7367	47540	2	0.668	0.094	8.490	1.773
		3684	23787	4	0.246	0.751	2.970	0.921
		1842	11944	8	0.102	0.139	0.961	1.062
ex19.rb	12005	6003	174775	2	1.687	0.007	12.579	2.485
		3002	87963	4	0.639	0.397	4.177	2.168
		1564	45271	8	0.246	0.081	1.246	1.565
garon02.rb	13535	6768	195803	2	2.597	0.087	7.555	0.839
		3384	98794	4	1.057	2.612	2.708	0.727
		1692	50748	8	0.400	0.198	0.871	0.815
lhr10c.rb	10672	5396	124253	2	1.434	0.088	6.436	0.649
		2734	62327	4	0.534	0.107	1.813	1.045
		1368	32099	8	0.223	0.104	0.539	0.704
memplus.rb	17758	8879	71619	2	0.905	0.158	18.384	0.476
		4444	36131	4	0.375	0.164	7.134	1.763
		2232	22875	8	0.143	0.280	2.421	1.690
olafu.rb	16146	8023	549977	2	7.480	0.101	17.876	1.704
		4038	288521	4	2.938	0.107	7.304	2.214
		2022	144842	8	1.155	0.147	2.593	2.299
perrel.rb	20700	10350	255575	2	3.668	0.103	24.712	2.022
		5175	128150	4	1.517	0.092	5.905	0.902
		2590	64425	8	0.608	0.139	2.517	1.838
poli_large.rb	15575	7833	16895	2	0.274	0.095	8.375	0.918
		4073	8696	4	0.104	0.085	3.090	0.809
		1963	4316	8	0.041	0.150	1.013	0.785
raefsky03.rb	21200	10600	748096	2	10.509	0.305	24.095	3.803
		5304	374176	4	4.713	0.119	10.527	1.077
		2664	188088	8	1.941	0.176	4.254	3.784
wang3.rb	26064	13032	88606	2	1.255	3.548	15.202	3.140
		6516	44344	4	0.554	0.122	6.831	0.936
		3258	22242	8	0.230	0.216	2.636	2.030

Table 1: Elapsed time of computation on COMPAQ

matrix info				#proc	elapsed time (sec)			
matrix	n	max#nloc	max#nnz		matvec	msg exch	dotprod	MPI_ALLREDUCE
bayer02.rb	13935	7020	32160	2	1.274	0.048	38.452	1.028
		3519	16184	4	0.756	0.057	19.018	4.034
		1775	8216	8	0.329	0.074	10.985	4.517
epb1.rb	14734	7367	47540	2	1.877	0.056	20.165	0.180
		3684	23787	4	0.906	0.067	10.612	1.631
		1842	11944	8	0.509	0.080	6.348	3.798
ex19.rb	12005	6003	174775	2	5.607	0.006	31.414	3.382
		3002	87963	4	2.704	0.040	16.294	3.470
		1564	45271	8	1.440	0.067	8.895	4.490
garon02.rb	13535	6768	195803	2	6.131	0.076	18.186	0.771
		3384	98794	4	3.061	0.103	9.899	1.061
		1692	50748	8	1.564	0.115	5.910	3.171
lhr10c.rb	10672	5396	124253	2	3.824	0.052	16.540	0.985
		2734	62327	4	1.985	0.071	7.533	2.237
		1368	32099	8	1.019	0.079	4.569	3.017
memplus.rb	17758	8879	71619	2	2.943	0.549	41.889	0.872
		4444	36131	4	1.360	0.339	22.167	2.906
		2232	22875	8	0.913	0.320	11.749	3.257
olafu.rb	16146	8023	549977	2	15.831	0.099	42.175	2.409
		4038	288521	4	8.416	0.095	22.907	3.280
		2022	144842	8	4.460	0.102	12.854	6.768
perrel.rb	20700	10350	255575	2	7.527	0.132	54.518	0.355
		5175	128150	4	4.040	0.116	17.475	2.093
		2590	64425	8	2.077	0.102	10.102	1.919
poli_large.rb	15575	7833	16895	2	0.975	0.073	25.110	0.983
		4073	8696	4	0.468	0.083	11.653	1.367
		1963	4316	8	0.202	0.105	5.895	1.155
raefsky03.rb	21200	10600	748096	2	21.791	0.218	60.650	0.617
		5304	374176	4	11.126	0.150	35.140	8.409
		2664	188088	8	5.566	0.118	16.307	4.388
wang3.rb	26064	13032	88606	2	3.567	0.250	39.923	0.226
		6516	44344	4	1.868	0.156	18.583	0.647
		3258	22242	8	0.904	0.139	9.947	2.544

Table 2: Elapsed time of computation on cluster of PC

- The column of `#proc` is the number of processors used in the calculation.
- The column of `matvec` is the maximum total elapsed time of the 500 iterations for the slowest processor to compute the local matrix-vector product. Because the computing time for a local matrix-vector product is proportional to `#nnz`, the slowest processor should be the one whose number of nonzero entries of the local matrix is maximum. The elapsed time for this processor represents the computing time. For matrices partitioned with a good load balance, the computing time for all processors should be approximately the same. Once a local vectors can be stored in the L1-cache it can be reused without loading it from the main memory. Due to this effect a superlinear speedup could be seen when we increase the number of processors.
- In the column of `msg_exch`, before computing a local matrix-vector product, all processors exchange data with their neighbours. `msg_exch` is the maximum total elapsed time of 500 iterations of the processor that does the most communication with its neighbours. The communication cost is another factor to be taken into account when partitioning a matrix for parallel computing. For nontrivial matrices, the partitioning has to give a good trade-off between communication cost and load balance. The communication time of the current processor is proportional to both the number of neighbouring processors and the amount of exchanged data. With many neighbours, there are many times to initialize latency of the network. The data is transferred through the network, so the larger the amount of data, the more time is required to transfer them. Furthermore, when the network is busy, we cannot estimate the communication cost correctly.
- The column of `dotprod` is the maximum total elapsed time of all computations for the slowest processor to compute the local dot product. The number of operations for a local dot product is proportional to `#nloc`. When the local vectors fit in the cache, they can be reused which results in a superlinear speedup of the computing time when increasing the number of processors.
- The column of `MPI_ALLREDUCE` is the maximum total elapsed time of all computations for the slowest processor to sum the contributions from all processors using `MPI_ALLREDUCE`. Note that `MPI_ALLREDUCE` is a global synchronisation point where load imbalance influences the computing times.

Before analysing the experimental results, the characteristics of the computing machines are specified. We experimented on two platforms:

1. A COMPAQ, a cluster of 40 nodes and each node has the following characteristics

processor: 4 Alpha SC 45, 64 KB L1-cache, 8 MB L2-cache
memory: 4 GB of RAM shared by four processors
network: Quadrics with latency 3 μ sec and bandwidth 200 MB/s

2. A cluster of 8-node PC and each node has the following characteristics [5]

processor: 2 Pentium III, 16 KB L1-cache, 256 KB L2-cache
memory: 1 GB of RAM shared by two processors
network: Myrinet with latency 9 μ sec and bandwidth 250 MB/s

The experimental results on the COMPAQ are shown in Table 1. We consider the four different time measurements separately.

- The speedup of computing a local matrix-vector product is more than two when the number of processors is doubled. This is attributed to the efficient use of the cache of the machine. The L1-cache (64 KB) and L2-cache (8 MB) can store in double precision 8192 numbers and 1048576 numbers, respectively. The L2-cache is large enough to store a local matrix and two local vectors for all the examples. We observe many instances of superlinear speedup which we attribute to better use of the L1-cache.
- The performance of the local dot product of vectors can be much improved when the vectors can be stored in L1-cache. We notice that this is the case when $2*\#nloc < 8192$, where the speedup is more than three. When the size of the local vectors gets larger but they still fit into the L2-cache the speedup decreases but is still more than two when doubling the number of processors.

elapsed time for	1	2	3	4	5	6	7	8	9	10
matvec	1.601	1.305	1.365	1.373	1.361	1.205	1.210	1.213	1.209	1.275
msg exch	0.125	0.107	0.129	0.117	0.115	0.087	0.087	0.085	0.087	0.084
dotprod	14.506	16.092	15.978	16.039	16.357	14.023	14.116	14.735	13.999	14.144
MPI_ALLREDUCE	1.364	2.987	0.862	0.461	0.567	1.811	1.302	2.049	1.892	1.964

Table 3: The variation of the results on the COMPAQ

- The elapsed time of the data exchange depends on the number of neighbouring processors and the amount of data. Because on this computer there are four processors per node, the communication cost is different when the data exchange among processors takes place in the same node or between nodes. Moreover, the batch scheduler does not necessarily allocate processors within the same node for a parallel job. We run the code in multi-user mode so the traffics in the network and memory influence significantly in a way that is hard to predict.
- Similarly to the previous case, the elapsed time for `MPI_ALLREDUCE` is difficult to estimate. The difference is that the data exchange time depends on the number of neighbours but the time for `MPI_ALLREDUCE` depends on all processors. So we expect that the elapsed times in this case vary with the number of processors but the measurements sometimes give different results due to the causes mentioned above. The big variation in particularly the communication times are illustrated in Table 3, where the results of ten runs of `wang3.rb` for 2 processors are shown.

Table 2 shows the results of experiments on the cluster of PCs.

- Because the L1-cache (16 KB) of the processor is too small for storing even one local vector in almost all cases, we do not see a clear cache effect for computing a local matrix-vector product.
- The elapsed time for local dot products decreases approximately by a factor of two when the number of processors is doubled, i.e. linear speedup. Because the L1-cache is too small to store a complete vector, some entries of the vectors are stored in the L2-cache. This deteriorates the performance when compared to the case where all entries of the vectors can be stored in the L1-cache.
- As mentioned before, the elapsed time for data exchange depends on the number of neighbours and the amount of data. So, using results from the previous work [6], we expect that the data exchange time should increase when we increase the number of processors. However, we notice the results for matrix `memplus.rb`, `raefsky03.rb`, `wang3.rb`, and `perrel.rb` do not follow the expectation. Repeated tests show the same patterns (see Table 4). For these results we do not have a satisfactory explanation.
- The elapsed time for `MPI_ALLREDUCE` should increase with the number of processors. This expectation is followed by the results of almost all matrices. The most important exception is `raefsky03.rb`. This matrix is partitioned quite evenly so this effect can a priori not be explained by a bad load balancing. To determine the consistency of this result we have rerun this example five times. The result is shown in Table 4. This shows a significant variation of the results but all measurements consistently show significantly longer maximum elapsed times for `MPI_ALLREDUCE` on four processors than on eight. To explain this we tabulate in Table 5 the time measurements for a calculation using four processors. This table gives for all the processors (and not only for the slowest) `nloc`, `nnz`, `dotprod`, `msg exch`, `matvec`, `MPI_ALLREDUCE` and the total time of the calculation (which includes the vector updates). As we can see, the matrix and vector are evenly partitioned. However, the calculation (in particular the dot products) are performed more efficiently on processors three and four. Since `MPI_ALLREDUCE` synchronizes all processors, processors three and four have to wait for processors one and two to reach this point. For this reason the elapsed time for `MPI_ALLREDUCE` is much higher on processors three and four, and consists mainly of waiting time. We do not have an explanation why the calculations are performed with a difference performance on different processors for this example. This requires further investigation but that is outside the scope of this work.

elapsed time (sec)	1	2	3	4	5
2 processors					
dotprod	67.062	62.238	61.498	61.279	61.297
msg exch	0.246	0.239	0.218	0.213	0.223
matvec	22.501	22.407	22.274	22.098	22.276
MPI_ALLREDUCE	0.404	0.369	0.485	0.492	0.518
4 processors					
dotprod	37.852	34.470	34.034	33.309	32.662
msg exch	0.171	0.144	0.163	0.144	0.160
matvec	11.606	11.192	11.442	11.190	11.449
MPI_ALLREDUCE	18.913	13.387	12.645	9.322	8.421
8 processors					
dotprod	17.037	16.702	16.790	16.799	16.826
msg exch	0.113	0.109	0.113	0.108	0.341
matvec	5.951	5.852	5.841	5.855	5.891
MPI_ALLREDUCE	6.938	5.815	5.621	5.490	6.061

Table 4: The variation of the results for computing `raefsky03.rb` on the cluster of PC's

elapsed time(sec)	proc 1	proc 2	proc 3	proc 4
dotprod	37.852	37.641	28.818	28.946
msg exch	0.122	0.118	0.171	0.157
matvec	11.541	11.606	10.192	10.270
MPI_ALLREDUCE	0.158	0.456	18.908	18.913
total time	86.947	86.946	86.940	86.939
#nloc	5300	5300	5304	5296
#nnz	373920	374176	370624	370048

Table 5: Time measurement of four processors for computing `raefsky03.rb` on the cluster of PC's

For our experiments, the elapsed time for the dot product dominates all calculations. This is due to using GMRES without restarting. The dimension of Krylov subspace is 500, the number of iterations. In practice, GMRES is used with a restart parameter, say 20, and so the dimension of the Krylov subspace is just 20, which is the number of vectors to be orthogonalized. When this is the case, the matrix-vector products will take computing time over the others or at least the balance between a local dot product and local matrix-vector products will be more even.

4 Conclusion

We designed, implemented, and tested a data structure appropriate for computing the multiplication between a sparse matrix and a vector on distributed computing environments. Numerical results using a GMRES solver with a test set of large matrices show a satisfactory performance and scalability.

Acknowledgments. The author thanks Dr. Luc Giraud for the helpful discussion and some ideas on the data structure and parallel GMRES method. I thank Dr. Martin Van Gijzen for the suggestion on the input files and helpful discussion and comments on analysing the experimental results which shape this working note. Thanks to Prof. Iain Duff for his comments on technical writing.

References

- [1] I. S. Duff, R. G. Grimes, and J. G. Lewis. The Rutherford-Boeing Sparse Matrix Collection. Technical Report TR/PA/97/36, CERFACS, Toulouse, France, 1997. Also Technical Report RAL-TR-97-031 from Rutherford Appleton Laboratory and Technical Report ISSTECH-97-017 from Boeing Information & Support Services.
- [2] Ian Foster. *Designing and building parallel programs*. Addison-Wesley, 1995.
- [3] V. Frayssé, L. Giraud, S. Gratton, and J. Langou. A set of GMRES routines for real and complex arithmetics on high performance computers. Technical Report TR/PA/03/03, CERFACS, Toulouse, France, 2003.
- [4] V. Frayssé, L. Giraud, and H. Kharraz-Aroussi. On the influence of the orthogonalization scheme on the parallel performance of GMRES. Technical Report TR/PA/98/07, CERFACS, Toulouse, France, 1998. Preliminary version of proceeding of EUROPAR'98 Parallel Processing.
- [5] L. Giraud. Combining shared and distributed memory programming models on clusters of symmetric multiprocessors: Some basic promising experiments. Working Note WN/PA/01/19, CERFACS, Toulouse, France, 2001. Preliminary version of the paper published in the International Journal of High Performance Computing Applications, vol. 16, nber 4, pp 425-430, 2002.
- [6] S. Riyavong. Experiments on sparse matrix partitioning. Working Notes WN/PA/03/32, CERFACS, Toulouse, France, 2003.
- [7] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS publishing, New York, 1996.
- [8] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Scientific and Statistical Computing*, 7:856–869, 1986.
- [9] Ümit V. Çatalyürek and Cevdet Aykanat. PaToH: Partitioning Tools for Hypergraphs. Technical report, 2002.