Performance Prediction for Parallel Iterative Solvers

V. Blanco[1], P. González[2], J.C. Cabaleiro[3], D.B. Heras[3],
T.F. Pena[3], J.J. Pombo[3], and F.F Rivera[3]

[1] Dept. of Statistics and Computer Science,
La Laguna University, 38071 Tenerife. Spain
`Vicente.Blanco@ull.es`
[2] Dept. of Electronics and Systems,
A Coruña University, A Coruña. Spain
`patricia@dec.usc.es`
[3] Dept. of Electronics and Computer Science,
Santiago de Compostela University, 15706 Santiago. Spain
`{caba,dora,tomas,juanjo,fran}@dec.usc.es`

Abstract. In this paper, an exhaustive parallel library of sparse iterative methods and preconditioners in HPF and MPI was developed, and a model for predicting the performance of these codes is presented. This model can be used both by users and by library developers to optimize the efficiency of the codes, as well as to simplify their use. The information offered by this model combines theoretical features of the methods and preconditioners in addition to certain practical considerations and predictions about aspects of the performance of their execution in distributed memory multiprocessors.

1 Introduction

The complexity of parallel systems makes *a priori* performance prediction difficult. The reasons for the poor performance of codes on distributed memory systems can be varied, and users need to be able to understand and correct performance problems. This fact is especially relevant when high level libraries and programming languages are used to implement parallel codes, as in the case of HPF. A performance data collection, analysis and visualization environment is needed to detect the effects of architectural and system software variations.

Most of the performance tools, both research and commercial, focus on low level message–passing platforms such as MPI or PVM[4], and the most prevalent approach taken by these tools is to collect performance data during program execution and then provide *post–mortem* display and analysis of performance information[12]. Our proposal is different; we present a model that predicts the performance of irregular HPF and MPI codes.

The efficient implementation of irregular codes in HPF is difficult. However, several techniques for handling this problem using intrinsic and library procedures as well as data distribution directives can be applied. An exhaustive HPF

P.M.A. Sloot et al. (Eds.): ICCS 2002, LNCS 2330, pp. 923–932, 2002.

library of iterative methods and preconditioners was developed[2]. A second version of the library was developed using the message–passing programming model for certain kernels of the library to obtain better performance[3]. The model presented in this paper analyses the performance of these codes, and can be used both by users of this library to optimize the efficiency, and by library developers to check inefficiencies.

In the literature, many iterative methods have been presented and it is impossible to cover them all. We chose the methods given below, either because they represent the current state of the art for solving large sparse linear systems [1] or because they present special programming features.

2 A library of iterative methods

2.1 Sparse linear systems

Let us consider applications that can be formulated in terms of the matrix equation $A \cdot x = b$, called linear system, where matrix A and vector b are given, and x must be calculated. The structure of A is highly dependent on the particular application, and some of them give rise to a matrix that is effectively dense and can be efficiently solved using direct factorization–based methods, whereas others generate a matrix that is sparse. For these types of matrices, iterative methods[1] are preferred, especially when A is very large and sparse, due to their efficiency in both memory and work requirements.

We developed $\mathcal{PARAISO}$ (PARAllel Iterative SOlver), that is a lib that includes several iterative methods, such us the Conjugate Gradient (CG), the Biconjugate Gradient (BiCG), the Biconjugate Gradient Stabilized (BiCGSTAB), the Conjugate Gradient Squared (CGS), the Generalized Minimal Residual (GMRES), the Jacobi method, the Quasi–Minimal Residual (QMR) and the Gauss–Seidel Successive Over–Relaxation (SOR). Some preconditioners are also implemented, and can be applied to the target sparse matrix to transform it into one with a more favourable spectrum. These preconditioners are: the Jacobi preconditioner, the Symmetric Successive Over–Relaxation (SSOR), the Incomplete LU factorization (ILU(0)), the Incomplete LU factorization with threshold (ILUT), the Neumann Polynomial preconditioner and the Least Squares Polynomial preconditioner.

We implemented three version of these codes on the AP3000[7]. A F90 version, a HPF version and an enhanced HPF version with those kernels coded in MPI (which we refer to as HPI).

2.2 HPF Implementation

The data–parallel programming model upon which HPF is based requires a well–defined mapping of the data onto local memories in order to achieve an efficient parallel code architecture. Henceforth, we assume that vectors are represented as N–element arrays and the sparse matrix is represented as three one–dimensional arrays, either in CSC or CSR format[9].

The most used operations in paraiso are the dotproduct and daxpy operations. HPF readily supports the inner product operations by an intrinsic function (`DOT_PRODUCT`), and in addition, the `daxpy` operation is easily performed using HPF's parallel array assignments. In any parallel implementation that distributes the vectors and the matrix among processor memories, the inner products and sparse matrix–vector multiplications require data communications. The element–wise multiplications in the inner products can be performed locally without any communication overhead, while the merge phase for adding up the partial results from processors involves some communication overhead. However, the data distributions can be arranged so that all of the other computations will be performed on local data only. For each operation we will show a data distribution pattern in order to obtain optimal performance, and how the operation is coded in HPF.

Using N_p processors, `daxpy` operations can be performed in $\mathcal{O}(N/N_p)$ time on any architecture. On the other hand, the inner products take $\mathcal{O}(N/N_p)$ time for the local phase, but the merge overheads change according to the network architecture.

As an example, we show here the code for a HPF implementation of matrix–vector product. Let's consider the multiplication of an $N \times N$ arbitrarily sparse matrix A, with NNZ non–zero entries, by an $N \times 1$ vector `x` that gives a $N \times 1$ vector `y`. Different solutions have been given to solve this problem. One implies the modification of the matrix adding *padding* elements in order to obtain a regular sparse matrix[10]. Other solutions involve the use of HPF extensions to include specific data distributions for sparse matrices[11]. We propose to use HPF intrinsic procedures[2]. Loops are replaced by calls to intrinsic and library procedures, which are inherently parallel. The HPF code for the spmatvec with CSC storage using HPF library procedures is shown below. A detailed description of the code can be found in [3]

```
INTEGER, DIMENSION(N+1)    :: colptr
INTEGER, DIMENSION(NNZ)    :: rowind
REAL, DIMENSION(NNZ)       :: d
REAL, DIMENSION(N)         :: x
REAL, DIMENSION(N)         :: y
REAL, DIMENSION(NNZ)       :: aux
LOGICAL, DIMENSION(NNZ)    :: segment

!HPF$ ALIGN (:) WITH x(:) :: y
!HPF$ ALIGN (:) WITH d(:) :: rowind, aux, segment
!HPF$ DISTRIBUTE (BLOCK)  :: d, x
!HPF$ DISTRIBUTE (*)      :: colptr

y = ZERO
aux(colptr(:N)) = x
aux = COPY_PREFIX(aux, SEGMENT = segment)
aux = d * aux
y = SUM_SCATTER(aux, y, rowind)
```

2.3 Hybrid Implementation: HPF+MPI

In this section we present the implementation of the main kernels of the iterative methods using MPI (Message Passing Interface). The objective is to take advantage of the flexibility of the message–passing paradigm to optimize irregular computations in these kernels. In this way, the computations that involve vectors (dot products, update of vectors (`daxpy`), etc.) can be efficiently coded in HPF, whereas the irregular kernels are coded in MPI. Henceforth, the version of Paraiso based on this approach will be referred as HPI.

To be able to carry out this approach, the matrix is distributed according to a Block Column Scatter scheme (BCS)[8]. The execution of some functions to redistribute the vectors used in the HPF part of the code (block distribution) to the distributions used in the MPI kernels (cyclic distribution) and vice versa is mandatory. These redistributions require high communication overheads.

The three vectors that represent the sparse matrix are distributed by HPF; it is necessary to transform it into a BCS matrix in order to implement the basic kernels in MPI.

Whereas, the BCS distribution uses a cyclic projection of the matrix onto $P \times Q$ processors. The matrix is partitioned according to a $P \times Q$ template, and each processor takes the non–null entries that fills with its position in the template, as shown in the example in figure 1.

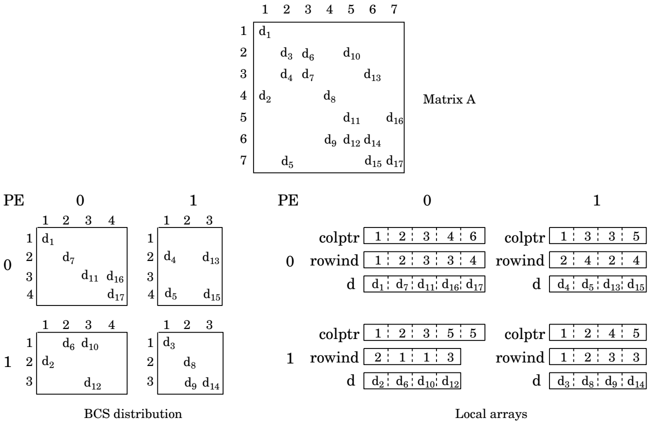

Fig. 1. BCS partition of a sparse matrix in a 2 × 2 processor mesh.

As the sparse matrix–vector product is the most time consuming operation in each iteration, its implementation should be as efficient as possible. Let us suppose a typical situation in an iterative method:

```
DO iter = 1, MAX_ITER
    HPF operations with vectors
    ...
    CALL HPI_spmatvec(A, x, y)
    ...
    HPF operations with vectors
END DO
```

For an efficient MPI implementation of the sparse matrix–vector product and taking into account the BCS distribution of the matrix, it is necessary to redistribute the input vector, x, from a block to a cyclic by columns distribution. The sparse matrix–vector product is then computed given a cyclic by rows distributed vector. Finally, this vector is redistributed to block in order to obtain the output vector, y. This process is shown in figure 2 and it is summarized as follows:

```
HPI_spmatvec(A, x, y)    /* input vector x, output vector y */
Block2CyclicCols(x, x_cyc_cols);
spmatvec(A, x_cyl_cols, y_cyc_rows);
CyclicRows2Block(y_cyc_rows, y);
```

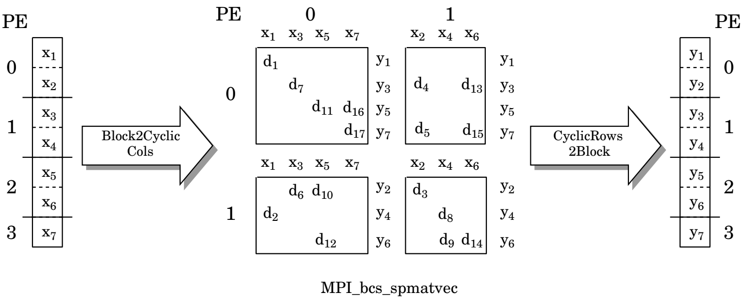

Fig. 2. HPI sparse matrix-vector product on a 2 × 2 processor mesh.

In order to implement the redistribution of the vector from block to cyclic by columns; it is needed a preprocessing stage to obtain the data that will be sent to each processor, their size and their stride. This stage is only executed once in the `HPI_init` function[3], and given the regularity of the distributions, lacks communications. With this information, the necessary redistributions for each sparse matrix–vector product can be carried out in two stages. In the preprocessing step, the elements that will be sent to the processors in the same row are determined. Given the regularity of the block and cyclic by columns distributions, it can easily be seen that if the first element to be sent is known, the remaining ones are equidistant. In each call to this function a communication step, by rows, is carried out, followed by another one, by columns, to complete the data needed by each processor. Each one of these communication steps is carried out by means of a collective communication. In the case of redistribution from cyclic by rows to block, since there is redundancy of data, only a communication step by columns is needed.

3 Performance Prediction

The complexity of parallel computers makes *a priori* performance prediction difficult. For this reason, performance data collection, analysis and visualization environments are needed to detect the effects of architectural and system software variations.

When programming these systems, the reasons for poor performance of parallel message–passing and data parallel codes can be varied and complex, and users need to be able to understand and correct performance problems. Performance tools can help by monitoring the execution of a program and producing performance data that can be analyzed to locate and understand areas of poor performance. This situation is particularly relevant when high–level libraries and programming languages are used to implement parallel codes, as in the case of HPF. This is true in regular problems, but it is especially important and difficult on irregular codes, such as those included in $\mathcal{PARAISO}$.

Most of the performance tools, both research and commercial, focus on low level message–passing platforms[4], such as MPI or PVM, and the most prevalent approach taken by these tools is to collect performance data during program execution, and then provide *post-mortem* display and analysis of performance information[12]. Our proposal is different, we present a model that predicts performance of the irregular codes of $\mathcal{PARAISO}$ before executing them, giving valuable information about theoretical and practical considerations that can help the user to understand the execution of several iterative methods on their particular sparse linear system.

3.1 Analisis of computations

Execution time is the common measure of computer performance. However, another popular alternative in numerical codes is *million floating point operations per second* (MFLOPS). MFLOPS gauges the capability of a system to deal with floating point math instead of raw instructions. The estimation of the number of FLOPs depends on the target machine. Hence, MFLOPS are not reliable, as the group of floating point operations is not consistent in different systems.

The proposed prediction model counts the number of FLOPs required for every kernel in the library. Based on these kernels it counts the number of FLOPs for one iteration of every method and preconditioner. We can not predict the number of iterations required to achieve convergence, however, the number of FLOPs per iteration gives an idea of the computational cost of any method.

In table 1, the number of FLOPs for different kernels and methods of the library $\mathcal{PARAISO}$ are shown. The number of FLOPs required for the first iteration of each method, which is the most expensive one is also included. In this way, the initial set of residuals and the computation of the required norms for the stopping criterium are considered.

3.2 Analisis of communications

The study of the communication pattern generated by HPF programs is essential for predicting its overhead. The straightforward way to check this pattern is to execute the program with profiling capabilities, taking data from different executions (i.e., different number of processors or different problems).

Table 1. #FLOPs for kernel and methods. n is matrix dimension (N). α is NNZ. r is the restart for the GMRES method

KERNELS	
kernel	**FLOPs**
spmatvec	2α
spmatvectrans	$2\alpha - 1$
dotproduct	$2n - 1$
jacobi_split	n
sor_split	$\alpha + n$
stoptest	$2n$
norm_inf	α
triang	$2\alpha - n$

METHODS		
method	**FLOPs 1st iter**	**FLOPs next iter**
CG	$19n + 7\alpha - 3$	$12n + 2\alpha$
BiCG	$23n + 9\alpha - 4$	$16n + 4\alpha - 1$
BiCGstab	$29n + 9\alpha$	$22n + 4\alpha + 3$
CGS	$25n + 9\alpha - 4$	$18n + 5\alpha - 1$
GMRES	$(r^2 + 6r + 13)n$ $+(2r + 7)\alpha - ((r^3/3)$ $+(9r^2/2) - (5r/6) - 2)$	$(r^2 + 6r + 8)n + (2r + 3)\alpha$ $+(2r + 3)\alpha - ((r^3/3)$ $+(9r^2/2) - (5r/6) - 1)$
QMR	$35n + 9\alpha + 16$	$24n + 4\alpha + 18$
Jacobi	$11n + 4\alpha - 1$	$5n + 2\alpha$
SOR	$10n + 5\alpha - 1$	$4n + 2\alpha$

We used a profiling tool to trace $\mathcal{PARAISO}$ library. Once the trace information for a given routine is obtained, the communication patterns for the different kernels in $\mathcal{PARAISO}$ library can be extracted. As an of an example, the behavior of the matrix–vector kernel is described. next.

The matrix–vector product was implemented in HPF using intrinsic functions in order to achieve high performance as was explained in section 2.2. From the point of view of communications, only lines 15, 16 and 18 produce messages[2].

`aux(colptr(:N)) = x` implements the first stage of filling `aux` vector. This HPF line presents an indirection in the left–hand side of the statement, Thus, the HPF compiler cannot detect which elements of vector `x` will be assigned to the correponding `aux` entry, as they depend on the values of `colptr` which are unknown at compilation time. The compiler that we used solves the situation in two stages: first, vector `x` is sent to processor 0 and then this processor calculates the corresponding `aux(i)` (since `colptr` is replicated); and then, it sends the result to the processor that owns `aux(i)`. The situation is described in figure 3 for four processors. Note the low efficiency of this approach, as in fact, the HPF line is executed in a sequential way. Similar study was made with routines at lines 16 and 18[3].

The HPI version of $\mathcal{PARAISO}$ integrates MPI coded kernels with HPF coded methods and preconditioners. From the point of view of communications, we only need to predict the MPI kernels, since dense operations with communications, such us the inner product, are performed in HPF and we can use the HPF communication prediction routines in this case.

As is explained in section 2.3, the MPI implementation of the kernels is based on a distributed BCS matrix and a number of redistribution routines to exchange data between the HPF and MPI worlds. The prediction of the performance of these kernels consists of simulating them by using an array of BCS matrices (one for each processor). All the communications in these kernels are based on global MPI communication routines (MPI_Alltoallv, MPI_Allreduce, etc.) which

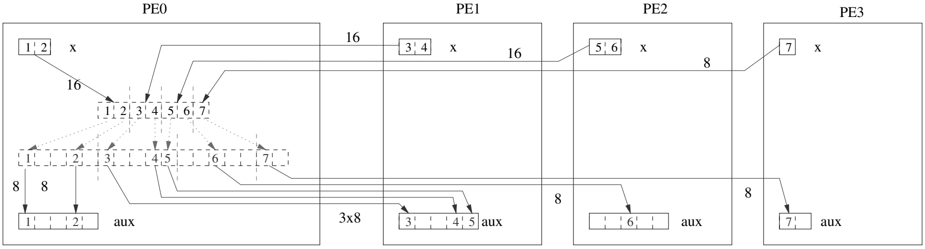

Fig. 3. `aux(colptr(:N)) = x` using four processors

direct the communication patterns defined in HPLinit. The prediction routines for HPI builds a simulation of these communications patterns and gathers the corresponding communication accounting for every MPI communication routine.

3.3 A model to predict execution times of computations and communications

In order to predict the execution time of parallel irregular codes such as those of $\mathcal{PARAISO}$, it is necessary to consider a great number of events: computations, communications, memory access costs, waiting times, etc.

We construct a model that provides an estimation of the computation time due to FLOPS, the number of cache misses, and the communication times for each message according to its size. The relationship between the number of FLOPs and the actual execution time for every method can be modeled by the following linear expression:

$$t_{comp} = \gamma f + \beta \qquad (1)$$

where f is the number of FLOPs and t_{comp} is the execution time in seconds. The values for the parameters γ and β depends on the iterative method. They are shown in table 2. R^2 is the fitting standard deviation.

Table 2. Linear model for computation and communication time. Fitting parameters

Computations			
Method	γ	β	R^2
CG, BiCG, BiCGStab, GMRES, CGS	0.419	0.0013	0.99
SOR	0.637	0.0014	0.99
Jacobi	0.726	0.0006	0.99
QMR	0.375	0.0012	0.99

Communications				
Message	Size (doubles)	γ	β	R^2
Send	0–120	$1.59 \cdot 10^{-7}$	$7.63 \cdot 10^{-5}$	0.39
Receive	0–120	$3.21 \cdot 10^{-7}$	$4.59 \cdot 10^{-5}$	0.66
Send	121–1000	$6.25 \cdot 10^{-8}$	$1.15 \cdot 10^{-4}$	0.92
Receive	121–1000	$7.44 \cdot 10^{-8}$	$7.41 \cdot 10^{-5}$	0.93
Send	1001–	$8.02 \cdot 10^{-8}$	$8.91 \cdot 10^{-5}$	1.00
Receive	1001–	$1.40 \cdot 10^{-7}$	$9.75 \cdot 10^{-5}$	1.00

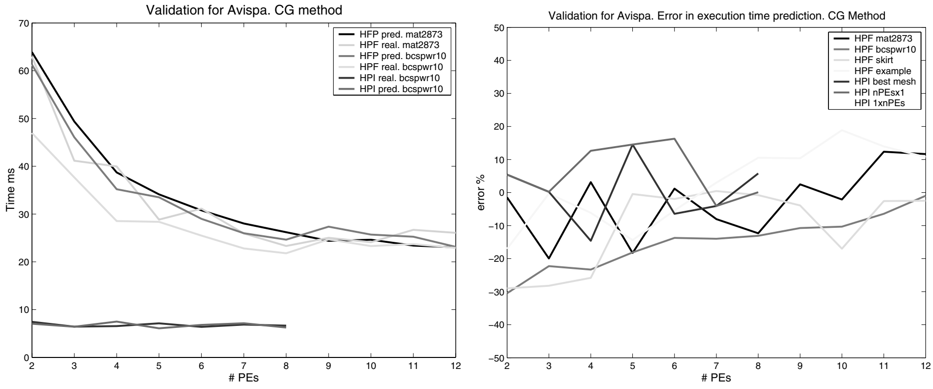

Fig. 4. Validation of the prediction model for the CG method with different matrices

We have carried out a similar study for the cost of the communications, measuring the time for sending and receiving a message of different sizes. Once again, according to a linear model we have:

$$t_{comm} = \gamma m + \beta \tag{2}$$

where m is the message size in doubles (8 bytes) and t_{comm} is the execution time in seconds. We obtain three different intervals for characterizing this behaviour (shown in table 2). We have used a ping-pong benchmark to establish these measures. Note that, for small messages, the correlation indexes are not high due to the great variance of measuring small runtimes. For larger messages we obtain better results for the fittings.

Finally, we have used a model to predict the number of cache misses found in accessing data for the HPI version of $\mathcal{PARAISO}$. The model[6] is based on a program that simulates the secondary cache of each processor of the AP3000.

4 Results

To validate the prediction model, a number of experiments have been carried out on the AP3000 multiprocessor system. As an example we show the Conjugate Gradient method with a set of matrices of the Harwell–Boeing suite[5]. In figure 4 the predicted and real execution times, and the prediction error for different matrices with the CG method are shown. Note that most of the predictions show errors lower than 15%.

5 Conclusion

A parallel iterative solver library is presented and a performance prediction model for this library is developed. The library has been implement using HPF and a number of kernels of the library were codes in MPI in order to achive better performance.

The execution time of these codes has been characterized in terms of MFLOPs, and the communication patterns of the principal kernels were established. By using the prediction model, it is easy to understand the application behaviour, to evaluate the load balance, to analyze the performance of the kernels, to investigate the communication patterns and performance, and to identify communication hot spots.

Future work for this prediction model will be required in order to obtain better prediction times for communication in HPI version, and it will be especially necessary to model especially the collective communications routines used in HPI kernels.

6 Acknowledgements

The work was supported by CESGA and FECIT under project 1998CP199.

References

1. R. Barret, M. Berry, et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1994.
2. V. Blanco, J. C. Cabaleiro, P. González, D. B. Heras, T. F. Pena, J. J. Pombo, and F. F. Rivera. A performance analysis tool for irregular codes in HPF. In *Fifth European SGI/Cray MPP Workshop*, Bologna, 1999.
3. V. Blanco, J. C. Cabaleiro, P. González, D. B. Heras, T. F. Pena, J. J. Pombo, and F. F. Rivera. Paraiso project. www.ac.usc.es/~paraiso, jun 2000.
4. S. Browne, J. Dongarra, and K. London. Review of performance analysis tools for mpi parallel programs. www.cs.utk.edu/~browne/perftools-review.
5. I. Duff, R. Grimes, and J. Lewis. Users guide for the harwell-boeing sparse matrix collection. Technical report, CERFACS, 1992.
6. D. Heras, V. Blanco, J. Cabaleiro, and F. Rivera. Modeling and improving locality for the sparse matrix–vector product on cache memories. *High Performance Numerical Methods and Applications*, 2000. special issue in Future Generation Computer Systems.
7. H. Ishihata, M. Takahashi, and H. Sato. Hardware of ap3000 scalar parallel server. *Fujitsu Sci. Tech.*, pages 24–30, 1997.
8. L. F. Romero and E. L. Zapata. Data distributions for sparse matrix vector multiplication. *Parallel Computing*, 21(4):583–605, April 1995.
9. Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Co., 1996.
10. E. Sturler and D. Loher. Parallel solution of irregular, sparse matrix problems using High Performance Fortran. Technical Report TR-96-39, Swiss Center for Scientific Computing, 1996.
11. M. Ujaldon, E. Zapata, B. Chapman, and H. Zima. Vienna Fortran/HPF extensions for sparse and irregular problems and their compilation. *IEEE Transactions on Parallel and Distributed Systems*, 8(10):1068–1083, Oct. 1997.
12. Vampir. Visualization and analysis of mpi programs. www.pallas.de.