Performance of Parallel Iterative Solvers: 
a Library, a Prediction Model, and a Visualization Tool

*Department of Statistics and Computer Science
La Laguna University
38071 Tenerife, Spain
+Department of Electronics and Systems
A Coruña University
15071 A Coruña, Spain
Department of Electronics and Computer Science
Santiago de Compostela University
15782 Santiago, Spain

The objective of this work is the analysis and prediction of the performance of irregular codes, mainly in their parallel implementations. In particular, this paper focuses on parallel iterative solvers for sparse matrices as a relevant case of study of this kind of codes. An efficient library of solvers and preconditioners was developed using HPF and MPI as parallel platforms. For this library, models to characterize and predict the behavior of the execution of the methods, preconditioners and kernels were introduced. To show the results of these models, a visualization tool with an easy to use GUI interface was implemented. Finally, results of the prediction models for the codes of the parallel library are presented using the visualization tool.

Keywords: parallel iterative solvers, sparse algebra, performance prediction, performance visualization, HPF, MPI

1. INTRODUCTION

Performance evaluation, instrumentation and visualization of parallel codes has been found to be a complex multidimensional problem [1] in parallel systems. Therefore, a priori performance prediction of these codes becomes a hard problem.

Tuning the performance of codes on distributed memory systems has been a high time-consuming task for users. They need to be able to understand and correct the performance problems in order to achieve good results. This is especially relevant when high level libraries and programming languages are used to implement parallel codes, as in the case of High Performance Fortran (HPF) [2]. A performance data collection, analysis and visualization environment is needed to detect the effects of architectural and system software variations.

Most performance tools, both research and commercial, focus on low level message-passing platforms, such as MPI and PVM [3-5], and the approach most often used by these tools is to collect performance data during program execution, and then provide

Received September 3, 2001; accepted April 15, 2002.
Communicated by Jang-Ping Sheu, Makoto Takizawa and Myongsoon Park.
post-mortem display and analysis of performance information [6, 7]. Our proposal is
different, we present a tool that predicts the performance of irregular HPF and MPI codes
without execute them in a parallel system.

Efficient implementations of regular codes in data-parallel programming model in
HPF are straightforward, however, the treatment of irregular codes is much more com-
plex. In particular, we focus on the particular case of iterative solvers on sparse matric-
es. Several techniques for handling this problem using intrinsic and library procedures
as well as data distribution directives can be applied [8-10]. We developed an exhaustive
HPF library of iterative methods and preconditioners [11], and a second version of
the library was implemented using the message-passing programming model for certain
kernels of the library to achieve better performance [12]. We built a performance pre-
diction model to analyze and predict the performance of these codes. And finally, we
implemented a performance visualization tool to show all the performance data provided
by the prediction models in order to help users to check inefficiencies and select the
appropriate solver.

In the literature, many iterative methods have been presented, and it is impossible to
cover them all. We chose the methods that represent the current state of the art for
solving large sparse linear systems [13], or that present special programming features.

The parallel codes were implemented on the Fujitsu AP3000 [14], a distributed
memory multiprocessor, which consists of 12 Ultra-SPARC processors connected by a
2D torus network. However, both the parallel codes and the performance tool can be
used directly on other parallel and distributed platforms.

2. A LIBRARY OF ITERATIVE METHODS FOR SPARSE SYSTEMS

Let us consider applications that can be formulated in terms of the matrix equation
\[ A \cdot x = b, \] called a 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. Some
applications 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 are preferred, especially when \( A \) is very
large and sparse, due to their efficiency in both memory and work requirements.

The term “iterative method” refers to a wide range of techniques that use successive
approximations to obtain more accurate solutions to a linear system at each step. From
the algorithmic point of view, non-stationary iterative methods [13] consist of four major
kernels: sparse matrix-vector products, preconditioners, inner products and vector up-
dates (\texttt{daxpy}). The choice of a preconditioner is of prime importance for efficiently
obtaining a solution of the linear system; nevertheless preconditioning will not be dis-
cussed here as it is often problem-dependent. The sparse matrix-vector product is the
kernel that uses up most of the execution time on each iteration of any iterative method.
Thus, in this section, this kernel is presented in detail. The other kernels are described
in [12, 15, 16].

\textsc{ParaIsO} (PARAllel Iterative SOlver), the parallel library developed in this work,
includes several iterative methods, such as the Conjugate Gradient (CG), Biconjugate
Gradient (BiCG), Biconjugate Gradient Stabilized (BiCGSTAB), Conjugate Gradient
Squared (CGS), Generalized Minimal Residual (GMRES), Jacobi method, Quasi-Minimal Residual (QMR) and Gauss-Seidel Successive Over-Relaxation (SOR) methods. Some preconditioners are also implemented, and can be applied to the target sparse matrix to transform it into one with a more favorable spectrum. These preconditioners are: the Jacobi preconditioner, Symmetric Successive Over-Relaxation (SSOR), Incomplete LU factorization (ILU(0)), Incomplete LU factorization with threshold (ILUT), the Neumann Polynomial preconditioner and the Least Squares Polynomial preconditioner.

We have implemented three versions of these codes on the AP3000: a Fortran 90 version, an HPF version and an enhanced HPF version with these kernels coded in MPI (which we refer to as HPI).

In order to save storage and computational resources, a sparse matrix must be represented in a compressed format [13]. We used a common sparse matrix storage scheme: CSC (Compressed Sparse Column). This scheme uses the following three arrays to store an $N \times N$ sparse matrix with $NNZ$ non-zero entries:

- $d(\text{NNZ})$, containing the non-zero elements stored column-wise;
- $\text{rowind}(\text{NNZ})$, which stores the row numbers of each non-zero element;
- $\text{colptr}(N+1)$, whose $j$th entry points to the first entry of the $j$th column in $d$ and $\text{rowind}$.

### 2.1 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. Henceforth, we assume that vectors are represented in CSC format as $N$-element arrays, and that the sparse matrix is represented as three one-dimensional arrays.

HPF readily supports inner product operations by means of an intrinsic function (routine $\text{DOT_PRODUCT}$), and the $\text{daxpy}$ operation is easily performed using HPF’s parallel array assignments. In any parallel implementation that distributes the vectors and the matrix across local memories, the inner products and sparse matrix-vector multiplications require data communications. The element-wise multiplications in the inner product operations can be performed locally without any communication overhead, while the merge phase for adding up the partial results from processors involves some communication overheads. However, the data distributions can be arranged so that all the other computations will be performed locally. For each operation, we got a data distribution pattern that can be used to obtain optimal performance.

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

#### 2.1.1 HPF implementation of matrix–vector product code

We consider the multiplication of an $N \times N$ sparse matrix $A$, with $NNZ$ non-zero entries, by a vector $\mathbf{x}$ that gives a vector $\mathbf{y}$, both of dimension $N$. Different solutions have
been given for this problem. One implies the modification of the matrix by adding padding elements in order to obtain a regular sparse matrix [8]. Other solutions involve the use of HPF extensions to include specific data distributions for sparse matrices [9, 10].

We propose to avoid the use of padding or HPF extensions by using HPF intrinsic procedures [11]. Loops are replaced by calls to intrinsic and library procedures, which are inherently parallel. The HPF code for the sparse matrix-vector product ($spmatvec$) with CSC storage using HPF library procedures is shown in Listing 1.

Listing 1. Code for the sparse matrix-vector product in HPF.

```
1 INTEGER, DIMENSION(N + 1) :: colptr
2 INTEGER, DIMENSION(NNZ) :: rowind
3 REAL, DIMENSION(NNZ) :: d
4 REAL, DIMENSION(N) :: x
5 REAL, DIMENSION(NNZ) :: y
6 REAL, DIMENSION(NNZ) :: aux
7 LOGICAL, DIMENSION (NNZ) :: segment
8  !HPF$ ALIGN (:) WITH x(:) :: y
9  !HPF$ ALIGN (:) WITH d(:) :: rowind, aux, segment
10 !HPF$ DISTRIBUTE (BLOCK) :: d, x
11 !HPF$ DISTRIBUTE (*) :: colptr
12
13 y = zero
14 aux(colptr(:N)) = x
15 aux = COPY_PREFIX (aux, SEGMENT = segment)
16 aux = d * aux
17 y = SUM_SCATTER(aux, y, rowind)
```

Fig. 1. Lines 15 and 16 of the code for sparse matrix-vector product.
The code starts extending vector $\mathbf{x}$ to an auxiliary vector $\mathbf{aux}$ (lines 15 and 16), so that $\mathbf{x}$ is aligned with the columns of $\mathbf{A}$. This procedure is shown in Fig. 1. First, in line 15, a copy of the values of $\mathbf{x}$ are stored in $\mathbf{aux}$ guided by $\mathbf{colptr}$. It should be noted that, in order to reduce the communication cost of this step, vector $\mathbf{colptr}$ is replicated among all the processors (line 12). Therefore, each entry of $\mathbf{x}$ that is needed for the product in any column is copied to the corresponding positions of $\mathbf{aux}$ (line 16). The manner of performing data expansion depends on the matrix pattern, and is guided by a logical vector ($\mathbf{segment}$) computed prior to the sparse matrix-vector routine. Here, $\mathbf{segment}$ marks the changes of column in vector $\mathbf{d}$ and $\mathbf{rowind}$. So, a switch $\mathbf{T} \rightarrow \mathbf{F}$, or $\mathbf{F} \rightarrow \mathbf{T}$, marks a column change.

In Fig. 2 we show the execution of lines 17 and 18 of the code. Since vector $\mathbf{aux}$ is aligned with $\mathbf{d}$, the product of line 17 is executed without communications. Finally, in line 18, a gathering operation is performed to obtain the final result in the distributed vector $\mathbf{y}$. In this code, communications and computations are performed separately. Lines 15, 16 and 18 perform communications in such a way that the floating point operations in line 17 are carried out locally.

**Fig. 2. Lines 17 and 18 of Listing 1.**

### 2.2 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 to as HPI. We assume that the matrix is distributed according to a Block Column Scatter scheme (BCS) [17]. 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 Fig. 3.
To implement these kernels, the execution of some functions in order to redistribute the vectors used in the HPF part of the code (block distribution) to distributions used in the MPI kernels (cyclic) and vice versa is mandatory.

The structure of the HPI library is shown in Fig. 4. In this figure, the stages of the HPI execution are depicted. Some of the main kernels of these stages are displayed.

2.2.1 BCS matrix redistribution

The three vectors that represent the sparse matrix are distributed in HPF; it is necessary to transform it into a BCS matrix in order to implement the basic kernels in MPI. Therefore, this starting distribution is the same as in the HPF version [18] as shown in Listing 2.

Listing 2. HPF vector distribution.

```hpfortran
1  !HPF$ DISTRIBUTE (BLOCK) :: d, rowind
2  !HPF$ DISTRIBUTE (*) :: colptr
```

2.2.2 Sparse matrix-vector product in HPI

The sparse matrix-vector product is the most time consuming operation in each iteration. For this reason, its implementation should be carried out as efficiently as pos-
possible since it will determine the global efficiency of the method. Let us assume a typical situation in an iterative method:

Listing 3. Excerpt of the code for a typical iterative method.

```fortran
1   DO iter = 1, MAX_ITER
2      HPF operations with vectors
3      ...
```
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 Fig. 5 and summarized as follows:

![Fig. 5. HPI sparse matrix-vector product on a 2 × 2 processor mesh.](image)

In order to implement redistribution of the vector from block to cyclic by columns, a preprocessing stage is needed to determine the data that will be sent to each processor, their size and the stride. This stage is only executed once in the HPI_init function (see Fig. 4) and, given the regularity of distributions, it lacks communications. In a 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, then the remaining ones are equidistant. In each call to this function, a communication by rows is carried out, followed by another 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. A detailed description of the redistribution routines can be found in [12].

**Listing 4. Excerpt of the code for the HPI spmatvec routine.**

```c
1  HPI_spmatvec(A, x, y) /* input vector x, output vector y */
2      Block2CyclicCols(x, x_cyc_cols);
```
3. PERFORMANCE PREDICTION

To predict the performance of the codes of PARAISO, we focused on events related to computations, communications, and memory accesses. Next, we describe the prediction model for all of them.

3.1 Analysis of Computations

Execution time is the common measure of computer performance. However, another popular alternative [19] in numerical codes is million floating point operations per second (MFLOPS), especially in numerical codes like ours. Therefore, the proposed prediction model counts the number of floating point operations (FLOPs) required for every kernel in the library. Based on these kernels, it counts the number of FLOPs for one iteration of every method. We cannot 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.

The number of FLOPs for every preconditioner is also counted. In this case, both the number of FLOPs for the construction stage of the preconditioner and the number of FLOPs for its application are given. It must be taken into account that, in most cases, this is only an estimation of the number of operations, as some run-time parameters necessary for counting (such as dropping in the ILUT preconditioner).

In some preconditioners, as in the case of kernels, counting consists on the evaluation of simple algebraic expressions involving certain parameters of the matrix at hand, such as the number of non-zero entries or the dimension of the matrix, and certain information about the operations, such as the grade of the polynomial preconditioners. In

Table 1. Number of FLOPs for kernels and methods, where \( n \) is matrix dimension (N), \( \alpha \) is \( \text{NNZ} \), and \( r \) is the restart parameter for the GMRES method.

<table>
<thead>
<tr>
<th>KERNELS</th>
<th>METHOIDS</th>
<th>FLOPs 1st iter</th>
<th>FLOPs next iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>spmatvec</td>
<td>CG</td>
<td>( 19n + 7\alpha - 3 )</td>
<td>( 12n + 2\alpha )</td>
</tr>
<tr>
<td>spmatectrans</td>
<td>BiCG</td>
<td>( 23n + 9\alpha - 4 )</td>
<td>( 16n + 4\alpha - 1 )</td>
</tr>
<tr>
<td>dotproduct</td>
<td>BiCGstab</td>
<td>( 29n + 9\alpha )</td>
<td>( 22n + 4\alpha + 3 )</td>
</tr>
<tr>
<td>jacobi_split</td>
<td>CGS</td>
<td>( 25n + 9\alpha - 4 )</td>
<td>( 18n + 5\alpha - 1 )</td>
</tr>
<tr>
<td>sor_split</td>
<td>GMRES</td>
<td>( (r^2 + 6r + 13)n )</td>
<td>( (r^2 + 6r + 8)n + (2r + 3)\alpha )</td>
</tr>
<tr>
<td>stoptest</td>
<td></td>
<td>( + (2r + 7)\alpha - ((r^3/3) )</td>
<td>( + (2r + 3) - ((r^3/3) )</td>
</tr>
<tr>
<td>norm_inf</td>
<td></td>
<td>( + (9r^2/2) - (5r/6) - 2 )</td>
<td>( + (9r^2/2) - (5r/6) - 1 )</td>
</tr>
<tr>
<td>triang</td>
<td>QMR</td>
<td>( 35n + 9\alpha + 16 )</td>
<td>( 24n + 4\alpha + 18 )</td>
</tr>
<tr>
<td></td>
<td>Jacobi</td>
<td>( 11n + 4\alpha - 1 )</td>
<td>( 5n + 2\alpha )</td>
</tr>
<tr>
<td></td>
<td>SOR</td>
<td>( 10n + 5\alpha - 1 )</td>
<td>( 4n + 2\alpha )</td>
</tr>
</tbody>
</table>
Table 2. Number of FLOPs for the preconditioners, where \( n \) is matrix dimension \((N)\), \( \alpha \) is \( \text{NNZ} \), and \( g \) is the grade of the polynomial in Newmann and Isquares preconditioners.

<table>
<thead>
<tr>
<th>Preconditioner</th>
<th>FLOPs for the construction</th>
<th>FLOPs for the application</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Jacobi</td>
<td>( n )</td>
<td>( n )</td>
</tr>
<tr>
<td>Newmann</td>
<td>( \alpha )</td>
<td>( 2\alpha + gn )</td>
</tr>
<tr>
<td>SSOR</td>
<td>( (3\alpha - n)2 )</td>
<td>( 2\alpha - n )</td>
</tr>
<tr>
<td>( \text{ilu0} )</td>
<td>( (\alpha - n)2 + (\alpha/n)(\alpha - n)/2 )</td>
<td>( 2\alpha - n )</td>
</tr>
<tr>
<td>( \text{iluT} )</td>
<td>( 2 + (2n^3 - n^2 + n)/2 + \log(n\text{entries} + \alpha) )</td>
<td>( + (2\text{entries} - 1)n )</td>
</tr>
<tr>
<td>Isquares</td>
<td>( \alpha + (11(g + 1) + 19)(g + 1)/2 )</td>
<td>( 2g(\alpha + n) )</td>
</tr>
</tbody>
</table>

In other cases, counting becomes a much more expensive task because it depends on dynamic parameters, such as the matrix pattern. In these cases, we assume some parameters by default and estimate the number of FLOPs for them (i.e. a representative particular case).

In Tables 1 and 2, the numbers of FLOPs for different kernels, methods and preconditioners of 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 computation of the required norms for the stopping criterion are considered. The counting of the number of FLOPs for the HPI version of PARAISO is the same as in the HPF version.

### 3.2 Analysis of Communications

The study of the communication patterns generated by an HPF program 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 numbers of processors or different problems).

PARAISO was developed using the Portland Group HPF compiler: \texttt{pghpf} [20]. A profiling tool called \texttt{SvPablo} [21] from the Pablo Project [22, 23] is freely available for this compiler. This library offers a brief summary about communications on a per-line basis. Taking advantage of the fact that the source code of \texttt{SvPablo} is available, we developed a number of new routines that included in this tool to provide more detailed information from communications for the execution of the HPF program [11, 18, 24]. With this extension, two types of trace information can be extracted: a summary of the computations and communications, provided by the original \texttt{pghpf2SDDF} library, and a detailed view of the communication events, provided by our extension.

Once the trace information for a given routine is obtained, the communication patterns for the kernels in the PARAISO can be extracted. By means of an example, the behavior of the sparse matrix-vector product is described.
3.2.1 Communications of the sparse matrix-vector product in HPF

The matrix-vector product was implemented in HPF using intrinsic functions in order to achieve high performance as explained in section 2.1.1. From the point of view of communications, only lines 13, 14 and 16 in Listing 1 produce messages.

Sentence \( \text{aux} \{ \text{colptr}(:N) \} = \text{x} \) implements the first stage of filling \( \text{aux} \) vector as shown in Fig. 1. This HPF line presents an indirection on the left-hand side of the instruction. Thus, the compiler cannot detect which elements of vector \( \text{x} \) will be assigned to the corresponding \( \text{aux} \) entry, as they depend on the values of \( \text{colptr} \), which are unknown at compilation time. The compiler solves this situation in two stages: first, vector \( \text{x} \) is sent to processor 0, and then this processor calculates the corresponding \( \text{aux}(i) \) (since \( \text{colptr} \) is replicated); second, it sends the result to the processor that owns \( \text{aux}(i) \). This situation is shown in Fig. 6 for four processors. Note the low efficiency of this approach, as in fact, the HPF line is executed in a sequential manner.

![Fig. 6. aux (colptr(:N)) = x using four processors.](image)

Line \( \text{aux} = \text{COPY\_PREFIX} \{ \text{aux}, \text{SEGMENT} = \text{segment} \} \) implements expansion of \( \text{x} \) guided by \( \text{segment} \), a logical vector defined previously of the matrix-vector routine. \( \text{COPY\_PREFIX} \) accepts any logical vector as \( \text{SEGMENT} \) [25]. In our case, \( \text{segment} \) and \( \text{aux} \) are distributed in a block scheme, and they are aligned. Therefore, communications will occur only for the first and last elements assigned to each processor. This situation is solved through the sharing of two data elements with all neighbors for each PE as in a generalized hypercube topology. The communications performed by this code consist of the exchange of 16 bytes of data between neighbors in a hypercube topology. This behavior does not depend on the matrix size \( N \), nor on the pattern of the matrix, nor on the hypercube dimension.

Sentence \( \text{y} = \text{SUM\_SCATTER} \{ \text{aux}, \text{y}, \text{rowind} \} \) is performed using an HPF intrinsic routine. It is also optimized for communications, and the behavior in this case can be described in two stages. Firstly, the data from distributed arrays \( \text{aux} \) and \( \text{rowind} \) are sent to the owner of \( \text{y} \), and the sum is performed locally. There is a possibility of packed communications in this case (two or more elements of \( \text{aux} \) and \( \text{rowind} \) are sent in one message). The left-hand side of Fig. 7 illustrates this operation. Vector \( \text{y} \) guides the operations according to the owner’s compute rule in such a way that the corresponding values of \( \text{aux} \) and \( \text{rowind} \) are sent to each processor.

The gathering operation is completed in our particular case. However, the intrinsic function \( \text{SUM\_SCATTER} \) allows for an extra argument [25] a logical vector which defines a
mask for the gathering operation. The communication induced by this situation explains the second part of messages described below; they are performed even though they are not needed in our particular use of this function. This communication is similar to that generated by the \texttt{COPY\_PREFIX} function. In this case, the size of the messages does not depend on the matrix size or pattern, but rather on the number of processors. The right-hand side of Fig. 7 illustrates this operation.

### 3.2.2 Communications of the sparse matrix-vector product in HPI

The HPI version of \texttt{PARAISO} integrates MPI coded kernels using 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 as the inner product, are performed in HPF, and we can use the HPF communication prediction routines in this case.

As explained in section 2.2, the MPI implementation of the kernels is based on a distributed BCS matrix and a number of redistribution routines used to exchange data between the HPF and MPI worlds. The performance of these kernels is predicted by simulating them using an array of BCS matrices (one for each processor). All the communications in these kernels are based on global MPI communication routines (\texttt{MPI\_Alltoallv}, \texttt{MPI\_Allreduce}, etc.) which direct the communication patterns defined in \texttt{HPI\_init}. The prediction routines for HPI build a simulation of these commu-
communication patterns and gather the corresponding communication counts for every MPI communication routine.

As an example, we show here the spmatvec kernel. From the point of view of communications, this kernel has three main stages:

1. A redistribution of vector \( x \) from block to cyclic by columns (b2cc). This routine has two global communication operations: an MPI_Alltoallv by rows operation (we have a 2D mesh of processors in HPI) and an MPI_Allgatherv by columns operation.
2. A reduction operation to obtain \( y \) in all processors (distributed cyclic by rows). In this case, the reduction operation is performed by an MPI_Allreduce by rows, which consists of a normal reduction operation followed by a broadcast operation.
3. A redistribution of vector \( y \) from cyclic by rows to block (cr2b). This new redistribution routine performs an MPI_Alltoallv by columns operation.

The HPI spmatvec kernel prediction routine simulates all these operations based on the number of processors and the configuration of the mesh. Every message count is gathered in a structure that is passed to the visualization tool in order to be display.

### 3.3 A Model to Predict the Execution Times of Computations and Communications

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

We have information about the number of FLOPs that each processor has to execute as well as the number of cache misses [26] and the number and volume of messages sent to and received by each processor. With this information we develop a model that provides an estimation of the computation time due to them. Fig. 8 shows the relationship between the number of FLOPs and the actual execution time for every method. Note that it can be accurately modeled by a linear expression:

\[
t_{\text{comp}} = \gamma f + \beta,
\]

where \( f \) is the number of FLOPs and \( t_{\text{comp}} \) is the execution time in seconds. The values of \( \gamma \) and \( \beta \) depend on the iterative method. They are shown in Table 3. \( R^2 \) is the fitting standard deviation.

<table>
<thead>
<tr>
<th>Method</th>
<th>( \gamma )</th>
<th>( \beta )</th>
<th>( R^2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG, BiCG, BiCGStab, GMRES, CGS</td>
<td>0.419</td>
<td>0.0013</td>
<td>1.00</td>
</tr>
<tr>
<td>SOR</td>
<td>0.637</td>
<td>0.0014</td>
<td>0.99</td>
</tr>
<tr>
<td>Jacobi</td>
<td>0.726</td>
<td>0.0006</td>
<td>0.99</td>
</tr>
<tr>
<td>QMR</td>
<td>0.375</td>
<td>0.0012</td>
<td>0.99</td>
</tr>
</tbody>
</table>
We have carried out a similar study on the cost of the communications, by measuring the time needed for sending and receiving messages of different sizes. Once again, using a linear model, we have

\[ t_{\text{comm}} = \gamma m + \beta, \]  

where \( m \) is the message size in doubles (8 bytes) and \( t_{\text{comm}} \) is the execution time in seconds. We obtain three different intervals for characterizing this behavior linearly (shown in Table 4). We have used a ping-pong benchmark to establish these measures. As an example, Fig. 9 shows the curve fitting for large messages.

Fig. 9. Linear model of communications for large messages.
Table 4. Linear model for communication time: fitting parameters.

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

Note that, for small messages, the correlation indexes are not high due to the great variance when measuring small runtimes. For larger messages, we obtain better results for the fittings.

We used a model to predict the number of cache misses found in accessing data for the HPI version of PARAISO. The model is based on a program that simulates the secondary cache of each processor of the AP3000. This model is explained in [26].

To validate the prediction model, a number of experiments have been carried out on the AP3000 multiprocessor system. As an example we used the Conjugate Gradient method with a set of matrices of the Harwell-Boeing suite [27]. In Fig. 10, the predicted and real execution times are shown for mat2873, bcspwr10, skirt and example matrices. Fig. 11 shows the prediction error for these matrices with the CG method. Note that most of the predictions produce errors lower than 15%.

Fig. 10. Validation of the predictions for method CG with different matrices.

4. VISUALIZATION TOOL

The results of performance predictions, as described in section 3, were integrated into an user-friendly interactive visual tool designed for users and developers. In addition, this tool includes detailed information about the matrices themselves, the that we called methods, preconditioners and kernels, as well as other information about the execution of the codes.
AVISPA (A VISualization Tool for PARAISO) was developed under TCL/TK, and uses a freely-available plotting library, PiPlot [28], to show the information mentioned before in a visual manner. In fact, it is a GUI interface that allows the user to interact with the analysis of the behavior of the codes of PARAISO under different conditions (number of processors, HPF or HPI version, different distributions, different matrices, ...). A detailed description and a user’s guide can be found in [29, 30]. Next, some examples of the capabilities of AVISPA are shown.

Fig. 12 shows how AVISPA handles the selection of a sparse matrix. In particular, it shows the menu to load the matrix, that launches the window to select the matrix file from disk. The user can see the pattern of the matrix and some additional features in two different windows as shown in this figure.

As soon as the matrix and the right-hand side vector of the system were selected, the user can choose the number of processors, the method and preconditioner, and the parallel version (HPF or HPI) of the codes. Fig. 13 displays examples of how some of these selections can be done in AVISPA. In addition, a window displaying statistics about the number of FLOPs is included in this figure. Finally another window provides theoretical information about the selected method.
Among the big amount of performance prediction data, in Fig. 14 some examples are shown. In particular, this figure displays information about per-processor cache misses, and the predicted number and size of communications, both sent and received messages. It also shows the predicted load balance using a Kiviat diagram, the ratio between computation costs and communication overheads in a sector diagram, and a window summarizing the total execution time per iteration.

As an additional feature of AVISPA, the user can, in fact, execute the methods and preconditioners. In particular, a small number of iterations can be executed in order to analyze the residual evolution, and so, get a first approach about the convergence of the methods. Fig. 15 displays the window that shows the residuals for the first 10 iterations.
of a method that does not converge in this case.

5. CONCLUSIONS

In this work, a parallel iterative solver library is presented, and a performance prediction tool for this library is described. The library has been implemented using HPF, and a number of kernels of this library are coded in MPI in order to achieve better performance.

The execution time of these codes has been characterized in terms of the number of FLOPs, and the communication patterns of the main kernels have been established. With this information, the performance prediction model gives a detailed view of the performance of these codes.

By using AVISPA, the visualization tool, it is easy to understand the behavior of the application, 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. A variety of graphical displays presents important aspects about the behavior
Fig. 15. Example of the Residual evolution of 10 iterations of the C6 method.

of the execution of the codes: detailed views of computations and communications, pro-
gram execution statistics, communication statistics, an optional source-code displays, etc.

Future work on this environment will be required in order to obtain more accurate
prediction times for communication in the HPI version, and it will be especially neces-
sary to model the collective communication routines used in HPI kernels.

ACKNOWLEDGMENT

This work was supported by the Spanish MCYT and European FEDER under pro-
ject TIC2001-3694-C02 and by the European Commission under project IST-2001-
32243.

REFERENCES

2. High Performance Fortran Forum, High Performance Fortran Language Specifica-
tion, 1997.
19. D. Patterson and J. Hennessy, *Computer Architecture: A Hardware Software Inter...
Performance of Parallel Iterative Solvers

face, Morgan Kaufmann, 1996.


Vicente Blanco received the B.A. in Physics and the M.S. degree in Physics from the University of Santiago de Compostela in 1992 and 1993 respectively. Currently he is an assistant professor in the department of Statistics and Computer Science in the University of La Laguna whereas he is completing his Ph.D. in Computer Science. His research interests include performance evaluation, prediction and visualization of parallel codes, parallel algorithms for dense and sparse algebra, and Grid technology.
Patricia González received the M.S. and Ph.D. degree in Physics from the University of Santiago de Compostela in 1996 and 2001 respectively. Currently she is working in the department of Electronics and Systems in the University of A Coruña. Her research interests include parallel algorithms and numerical methods for solving dense and sparse systems of equations, data parallel programming for irregular algorithms, and Grid technology.

José C. Cabaleiro received the M.S. degree in Physics from the University of Santiago de Compostela in 1989 and the Ph.D. degree from the University of Santiago de Compostela in 1994. Currently he is an associate professor in the department of Electronics and Computer Science at the University of Santiago de Compostela. His research interests are parallel numerical algorithms for dense and sparse matrices, data parallel programming for irregular algorithms, prediction and improvement of performance for irregular problems on cache memories and computer graphics.

Dora B. Heras received a M.S. degree in Physics in 1994 and a Ph.D. in 2000 in the department of Electronics and Computer Science at the University of Santiago de Compostela (Spain). She is currently an Associate Professor at the same University. Her research interests include parallel programming for irregular algorithms and the study and improvement of the behavior of irregular codes on the memory hierarchy.

Tomás F. Pena received the M.S. degree in Physics from the University of Santiago de Compostela in 1989 and the Ph.D. degree from the University of Santiago de Compostela in 1994. Currently he is an associate professor in the department of Electronics and Computer Science at the University of Santiago de Compostela. His research interests involve parallel numerical algorithms for dense and sparse matrices, data parallel programming for irregular algorithms, FEM for modeling and simulation of homotransistors and heterojunction bipolar transistors, parallel BEM methods, and parallel volume rendering for computer graphics.
Juan J. Pombo received his B.A. in physics at the University of Santiago de Compostela in 1996. Since 1997, when he received his M.S. degree, he has been a member of the Department of Electronics and Computer Science at the same university, where nowadays he is completing his Ph.D. in Computational Science. His research fields during these years cover several aspects of parallel computing. Mainly, those arising from irregular, unstructured computing, such as parallel sparse matrices methods, and parallel codes for mesh generation, finite element analysis and adaptive schemes for FEM.

Francisco Fernández Rivera is an associate professor of computer architecture at the University of Santiago de Compostela (Spain). He attained his M.S. degree in 1986, and his Ph.D. in 1990 at the University of Santiago de Compostela. Throughout his career he has conducted research and published extensively in the areas of computer-based applications, parallel processing and computer architecture. His current research interests include compilation of irregular codes for parallel and distributed systems, and its applications, the analysis and prediction of performance on parallel systems and the design of parallel algorithms of sparse matrix algebra.