Parallelization of the GNU Scientific Library on Heterogeneous Systems

J. Aliaga*, F. Almeida†, J.M. Badía*, S. Barrachina*, V. Blanco†, M. Castillo*, U. Dorta†, R. Mayo*, E.S. Quintana*, G. Quintana*, C. Rodríguez†, F. de Sande†

*Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain; {*aliaga,badia,barrachi,castillo,mayo,quintana,gquintan*}@*icc.uji.es*.

†Depto. de Estadística, Investigación Operativa y Computación, Universidad de La Laguna, 38.271–La Laguna, Spain; {*falmeida,vblanco,casiano,fsande*}@*ull.es*.

Abstract— **In this paper we present our joint efforts towards the development of a parallel version of the GNU Scientific Library for heterogeneous systems. Two well-known operations arising in discrete mathematics and sparse linear algebra allow us to describe the design and the implementation of the library, and to report experimental results on heterogeneous clusters of personal computers.**

Index Terms— **GNU Scientific Library, scientific computing, parallel algorithms and architectures, heterogeneous parallel systems.**

I. Introduction

The GNU Scientific Library (GSL) [1] is a collection of hundreds of routines for numerical scientific computations developed from scratch. Although coded in ANSI C, these routines present a modern application programming interface for C programmers, and the library employs an object oriented methodology allowing wrappers to be written for high-level programming languages. The library includes numerical routines for complex arithmetic, matrices and vectors, linear algebra, integration, statistics, and optimization, among others.

There is currently no parallel version of GSL, probably due to the lack of a globally accepted standard for developing parallel applications when the project started. However, we believe that with the introduction of MPI the situation has changed substantially. MPI is nowadays accepted as the standard interface for developing parallel applications using the distributed-memory (or message-passing) programming model.

As a natural challenge and evolution of our research in the area of parallel and distributed computing, we intend to develop a parallel integrated version of GSL that can be used as a problem solving environment for numerical scientific computations. In particular, we plan our library to be portable to several parallel architectures, including shared and distributed-memory multiprocessors, hybrid systems (consisting of a combination of both types of architectures), and clusters of heterogeneous processors. We believe our approach to be different to some other existing parallel scientific libraries (see, e.g., *http://www.netlib.org*) in that our library targets multiple classes of architectures and, in particular, heterogeneous systems.

In this paper we describe the design and implementation of the part of our parallel integrated library which deals with distributed-memory heterogeneous systems. For simplicity, in the current version of the library we only consider heterogeneity in the processor computational performance, ignoring usual differences in the memory access speed or the interconnection network bandwidth. An additional, and even more important simplification, is that we ignore the different nature of the architectures in the system that might produce erroneous results or provoke deadlocks. While most modern computer architectures agree in general aspects regarding the representation of data (like, e.g., the use of IEEE 754 for real values), there are still a few of these details that can result in potential hazards, specially in iterative algorithms involving numerical computations [2]. We are aware this simplification is by no means a trivial one; although there exist reasonably straightforward solutions to some of these problems, implementing them comes at the expense of an important overhead in communication [2].

We employ two classical, simple operations arising in discrete mathematics and sparse linear algebra to illustrate our approach. Nevertheless, the results in this paper extend to a wide range of the routines in GSL. Our current efforts are focused on the definition of the architecture, the specification of the interfaces, and the design and parallel implementation of a certain part of GSL.

The rest of the paper is structured as follows. In Section II we describe the software architecture of our parallel integrated library for numerical scientific computing. In Section III we present our view of the parallel system and the programming model interface. Then, in Sections IV and V, we employ two examples to illustrate the solutions we adopted in order to accommodate data structures such as vectors and (sparse) matrices for heterogeneous computing. Finally, experimental results are given in VI, and some concluding remarks follow in Section VII.

II. Software Architecture

Our library has been designed as a multilevel software architecture; see Fig. 1. Following a common strategy in computer networks, each layer offers certain services to the higher layers and hides those layers from the details on how these services are implemented.

The *User Level* (the top level) provides a sequential interface that hides the intricacies of parallel programming to those

0-7695-2210-6/04 $20.00 © 2004

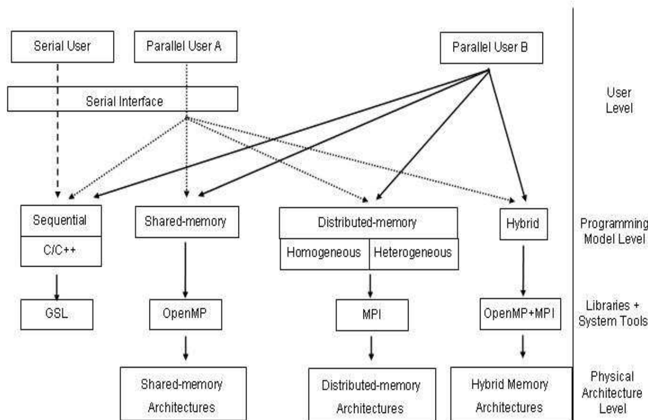

Fig. 1. Software architecture of the parallel integrated library for numerical scientifi c computing.

users with no such experience by supplying services through C/C++ functions according to the prototypes specified by the sequential GSL interface.

The *Programming Model Level* provides instantiations of the GSL library for several computational models, including the distributed-memory model. This level implements the services for the upper level using standard libraries and parallelizing tools like (the sequential) GSL, MPI, and OpenMP.

In the *Physical Architecture Level* the design includes shared-memory platforms (e.g., the SGI Origin 3000), distributed-memory architectures (like clusters of heterogeneous PCs), and hybrid systems (clusters of processors with shared-memory). Clearly the performance of the parallel routines will depend on the adequacy between the programming paradigm and the target architecture [3], [4].

III. Distributed-Memory Programming Model

A. View of the Parallel System

In the distributed-memory programming model we view the parallel application as being executed by a set of p *peer* processes, P_0, P_1,...,P_{p-1}, with all processes executing the same parallel code, possibly on different data (in general, this is know as Single Program Multiple Data, or SPMD [5], [6]). We prefer this organization over a master-slave organization due to its simplicity in certain applications [7], [3], [4].

We map one process per processor of the target parallel system where, in order to balance the computational load, a process will carry out an amount of work that is proportional to the performance of the corresponding processor. Performance of the processors is assumed to be measured off-line by the user, and provided in a configuration file with a specific format. As an example, we show next a simplified example of a configuration file for an heterogeneous system consisting of three nodes, (`ucmp0`, `ucmp12`, `sun`, with one processor each, and one node, `linex`), with 2 processors:

```
1: ucmp0:  1.0
2: ucmp12: 2.0
3: sun:    3.5
4: linex:  3.0
5: linex:  3.0
```

The performance figures in the example are normalized with respect to the slowest processor, `ucmp0`, which is always assigned a performance of 1.0. More elaborated versions of our library will eventually include the possibility of automatically generating the performance numbers by running a collection of benchmarks [8], [9]. A third possibility is that of running a sequential version of the parallel codes on a much smaller data [7].

Our parallel codes are also appropriate for homogeneous parallel systems, which are considered as a particular case of heterogeneous systems. However, in that case the codes incur in a small performance overhead due to the processing of the special data structures necessary in order to handle the heterogeneity.

B. Programming Model Interface

All programming models present a similar interface at this level. As an example, Table I relates the names of several sequential User Level routines with those corresponding to the instantiation of the parallel library for the distributed-memory model. The letters "`dm`" in the routine names after the GSL prefix ("`gsl_`") specify the distributed-memory model, and the next two letters, "`dd`", indicate that data are distributed. Other instantiations of the library include routines for shared-memory and hybrid systems.

User Level	Programming Model Level
Sequential	Distributed-memory
`fscanf()`	`gsl_dmdd_fscanf()`
`gsl_sort_vector()`	`gsl_dmdd_sort_vector()`
`gsl_usmv()`	`gsl_dmdd_usmv()`

TABLE I
Mapping of User Level routines to the corresponding parallel routines.

A program that sorts a vector is used next to expose the interface of the distributed-memory programming model:

```
1: #include <mpi.h>
2: #include <gsl_dmdd_sort_vector.h>
3: void main (int argc, char * argv []) {
4:    int n = 100, status;
5:    gsl_dmdd_vector * v;
      ...
6:    MPI_Init (& argc, & argv);
7:    gsl_dmdd_set_context (MPI_COMM_WORLD);
8:    v = gsl_dmdd_vector_alloc (n, n);      // Allocate
9:    gsl_dmdd_vector_scanf (v, n);          // Input
10:   status = gsl_dmdd_sort_vector (v);     // Sort
11:   printf ("Test sorting: %d\n", status); // Output
12:   gsl_dmdd_vector_free (v);              // Deallocate
13:   MPI_Finalize ();
14:   exit(0);
15: }
```

Here the user is in charge of initializing and terminating the parallel machine, with the respective invocations of routines `MPI_Init()` and `MPI_Finalize()`. Besides, as the information about the parallel context is needed by the GSL kernel, the user must invoke routine `gsl_dmdd_set_context` to transfer this information from the MPI program to the kernel and create the proper GSL context. Therefore, such routine

must be called before any other parallel GSL routine. It is also inside this routine that a process becomes aware of the performance of the corresponding processor. The MPI program above assumes the vector to be distributed among all processes so that, when routine `gsl_dmdd_vector_alloc` is invoked, the allocation of the elements follows a certain block distribution policy. If necessary, the user can access individual elements through the *Setters* and *Getters* methods of the structure. The call to `gsl_dmdd_sort_vector` sorts the distributed vector.

IV. A Case Study for Vectors: Sorting

Vectors (or one-dimensional arrays) are among the most common data structures appearing in scientific codes. In order to parallelize such codes, one of the first decisions to make is how to distribute the computational work among the processes in order to maximize the parallelism while minimizing overheads due, e.g., to communication or idle times.

In this section we first illustrate how we accommodate vectors in our parallel codes for heterogeneous systems and then we present the parallelization of a common operation arising in discrete-mathematics: sorting. The section is concluded with a few remarks on the possibilities of generalizing these ideas to deal with matrices.

A. Dealing with Vectors

Our approach consists in dividing the vector into b blocks of variable size, with the block sizes depending on the performance of the target processors for a given distribution policy. This corresponds to a block-cyclic nonuniform data distribution, with variable block sizes. Each process then performs the necessary operations on the data chunk assigned to it.

The following data structure is used to manage non-uniformly distributed vectors:

```
1: typedef struct {
2:   size_t global_size;         // Global size
3:   gsl_vector * local_vector;  // Local data
4:   size_t cycle_size;          // Cycle size
5:   size_t global_stride;       // Global stride
6:   size_t * local_sizes;       // Elements per process
7:   size_t * block_sizes;       // Block sizes per process
8:   size_t * offset;            // Offsets per process
9: } gsl_dmdd_vector;
```

Here, `local_vector`, of GSL type `gsl_vector`, is used to store the local data. Additional arrays replicate on each process the information needed by the *Setters* and *Getters* methods to transform global addresses to local addresses.

In order to create a distributed vector, routine `gsl_dmdd_vector_alloc` receives in a parameter the `cycle_size` which defines the distribution policy. As an example, figure 2 depicts the contents of this structure when used to store a vector of 16 elements that is distributed among 4 processes. Here, processes P_0 and P_1 are assumed to be run on processors that are three times faster than those the other two processes are mapped to. Also, the block sizes are 3 and 1 for the fast and slow processors, respectively.

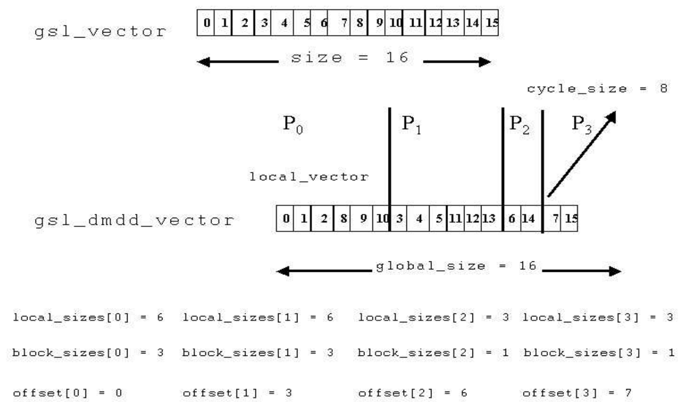

Fig. 2. A block-cyclic data layout of a distributed vector with 16 elements and 2 fast and 2 slow processors.

B. Sorting Vectors in Heterogeneous Systems

A sorting routine is used next to illustrate the parallelization in the distributed-memory programming model. For generality, we have chosen the well-known Parallel Sort by Regular Sampling (PSRS) algorithm, introduced in [10]. This algorithm was conceived for distributed-memory architectures with homogeneous processors and has good load balancing properties, modest communication requirements, and a reasonable locality of reference in memory accesses. We have adapted the algorithm for heterogenous environments using the `gsl_dmdd_vector` structure and a pure block nonuniform distribution policy (no cycles).

The proposed nonuniform PSRS algorithm is composed of the following five stages:

1) Each process sorts its local data, chooses $p-1$ (regularly spaced) samples, and sends them to P_0. The stride used to select the samples is, in this heterogeneous context, different in each process and is calculated in terms of the size of the local array to sort.
2) Process P_0 sorts the collected samples, finds $p-1$ "pivots", and broadcasts them to the remaining processes. The pivots are selected such that the merge process in step 4 generates the appropriate sizes of local data vectors according to the computational performance of the processors.
3) Each process partitions its data and sends its i-th partition to process P_i.
4) Each process merges the incoming partitions.
5) All processes participate in redistributing the results according to the data layout specified for the output vector.

A redistribution of data is required even for a vector that is distributed by blocks, since the resulting sizes of the chunks after stage 4 in general do not fit into a proper block distribution. The time spent in this last redistribution is proportional to the final global imbalance and depends on the problem data.

C. Matrices: Just a Generalization of Vectors?

Although matrices (or two-dimensional arrays) can be initially considered as a generalization of vectors, their use in parallel codes is quite more complicated [11], [12], [13], [14], [15], [16]. In particular, if the processes are organized following a two-dimensional topology (grid) in an heterogeneous system, the problem of mapping the processes into the grid and the subsequent distribution of the data over the process grid has been demonstrated to be an NP-complete problem [11]. Notice also that, as elements of matrices and vectors are often combined in certain operations like the matrix-vector product or the solution of a linear system, the parallel performance of these operations could largely benefit from using "compatible" or "conformal" distributions for matrices and vectors.

Two-dimensional block-cyclic data distributions for (dense) matrices have been proposed and utilized in libraries such as ScaLAPACK [17] and PLAPACK [18] for homogeneous parallel systems. More recently, ScaLAPACK data layout has been extended to accommodate heterogeneous systems in [11].

At this point we have not yet decided on how to deal with dense matrices in our parallel library for heterogeneous systems. However, we do have incorporated a special case of two-dimensional data structures, sparse matrices, to be described next.

V. A Special Problem: Sparse Matrices

Sparse matrices arise in a vast amount of areas, some as different as structural analysis, pattern matching, control of processes, tomography, or chemistry applications, to name a few. Surprisingly enough, GSL does not include routines for sparse linear algebra computations. This can only be explained by the painful lack of standards in this area: Only very recently the BLAS Technical Forum came with a standard [19] for the interface design of the *Basic Linear Algebra Subprograms* (BLAS) for unstructured sparse matrices.

Although several parallel packages for sparse linear algebra have been developed during recent years (see the survey at `http://www.netlib.org/utk/people/JackDongarra/la-sw.html`), these are all addressed for homogeneous parallel systems. Therefore, we believe our work to be unique in that it both follows the standard specification for the sparse BLAS interface and targets parallel heterogeneous systems.

A. Dealing with Sparse Matrices

The implementation, parallelization, and performance of sparse computations strongly depend on the storage scheme employed for the sparse matrix which, in many cases, is dictated by the application the data arises in.

Two of the most widely-used storage schemes for sparse matrices are the coordinate and the Harwell-Boeing (or compressed sparse array) formats [20]. In the coordinate format, two integer arrays of length nz hold the row and column indices of the nonzero elements of the matrix, while a third array, of the same dimension, holds the values. In general, these values are listed in the arrays by rows or by columns, resulting in the rowwise or columnwise variants of this format.

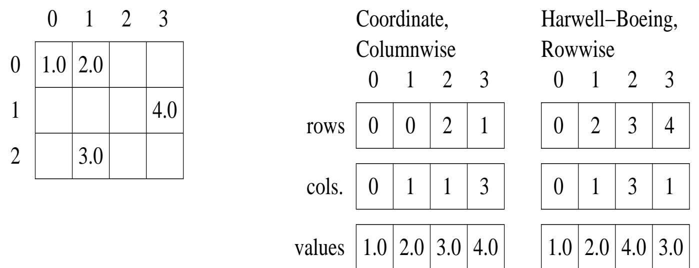

Fig. 3. Storage schemes for a 3×4 sparse matrix with $nz = 4$ nonzero elements.

The rowwise Harwell-Boeing scheme also employs a pair of arrays of length nz for the column indices and the values. A third array, of length equal the number of rows of the matrix plus 1, determines then the starting/ending indices for each one of the rows. The roles of the row and column arrays are swapped for the columnwise Harwell-Boeing variant. Figure 3 illustrates the use of these two storage schemes by means of a simple example.

As there seems to be no definitive advantage of any of the above-mentioned variants, in our codes we employ the rowwise coordinate format.

Our approach to deal with parallel platforms consists in dividing the matrix into p blocks of rows with, approximately, the same size. (Notice here that the number of blocks equals the number of processes; the reason for this will become clear when we describe the parallelization of the sparse matrix-vector product.) Each process operates then with the elements of its corresponding block of rows.

The following (simplified) data structure is used to manage non-uniformly distributed sparse matrices:

```
1: typedef struct {
4:   size_t global_size1;   // Global row size
5:   size_t global_size2;   // Global column size
6:   size_t global_nz;      // Global non-zeros
7:   void   * local_matrix; // Local data
5:   size_t * owns;         // Who owns a row?
6:   size_t * permutation;  // Permuted row indices
8: } internal_dmdd_sparse_matrix;
```

In this structure, `local_matrix` stores the local data of a process using the three vectors of the rowwise coordinate format. Also, "array" `permutation` allows to define a permutation of the matrix rows (for performance purposes). Array `owns` specifies the starting/ending indices of the block of rows that each process owns.

B. Parallelizing the Sparse Matrix-Vector Product

We describe next the parallel implementation of the sparse matrix-vector product

$$y \leftarrow y + \alpha \cdot A \cdot x, \quad (1)$$

where a (dense) vector y, of length m, is updated with the product of an $m \times n$ sparse matrix A times a (dense) vector x, with n elements, scaled by a value α. Notice that this is by far the most common operation arising in sparse linear algebra [21] as it preserves sparsity of A while allowing to employ codes that exploit the zeroes in the matrix to reduce the

computational cost of the operation. For simplicity, we ignore hereafter the more general case, where A can be replaced in (1) by its transpose or its conjugate transpose.

The (sparse) matrix-vector product is usually implemented as a sequence of *saxpy* operations or dot products, with one of them being preferred over the other depending on the target architecture and, in sparse algebra, the specifics of the data storage.

In our approach, taking into account the rowwise storage of the data, we decided to implement the parallel sparse matrix-vector product as a sequence of dot products. In order to describe the code we assume that both vectors involved in the product, x and y, are initially replicated by all processes. We also consider a block partitioning of vector $y = (y_0, y_1, \ldots, y_{p-1})$, with approximately an equal number of elements per block, and a partition of the sparse matrix A by blocks of roughly m/p rows as $A = \left(A_0^T, A_1^T, \ldots, A_{p-1}^T\right)^T$. The proposed nonuniform parallel sparse-matrix vector algorithm is composed of the following two stages:

1) Each process P_i computes its corresponding part of the product, $z_i = A_i \cdot x$; it then scales and accumulates the result as $z_i = y_i + \alpha \cdot z_i$.
2) Each process gathers the local results computed by the remaining processes and forms a replicated copy of y.

Notice that, because of the replication of x, the first stage does not require any communication. The second stage requires a collective communication (of type `MPI_Allgather`) to replicate the results. This operation is (almost) perfectly balanced as all processes have a close number of elements of y.

Notice, however, that the computational load may not be balanced at all, depending on the sparsity pattern of the data. In the next subsection we describe our efforts to balance the computational load while maintaining an equal amount of rows per process that also balances the communications in Stage 2.

C. Balancing the Computational Load in the Sparse Matrix-Vector Product

We propose to store the matrix with the rows permuted so that the number of nonzero entries in all row blocks A_0, A_1,…,A_{p-1}, is roughly proportional to the performance of the processor the process which owns this block is mapped to. Notice that this row permutation is only computed once, when the matrix is allocated, and can be used in all subsequent matrix-vector products. Thus, the cost of computing the permutation is amortized by using the same matrix in several computations. Repeated matrix-vector products with the same sparse matrix often occur, e.g., in the solution of linear systems by iterative methods [21] or in an implementation of the product of a sparse matrix times a dense matrix.

The problem of computing this permutation can be assimilated to a classical scheduling problem from operating systems (assign a fixed number of tasks to the processors of a system in order to reduce the response time). We propose to obtain an approximate solution by employing a heuristic procedure as follows. The strategy proceeds by first assigning the row (not assigned) with the largest number of nonzero elements to the fastest process. Then, for each one of the remaining processes, it searches for a row (not assigned) with a number of nonzero elements that makes the total number of nonzero elements assigned to this process proportional to its performance. The procedure is repeated, starting with the fastest process, until there are no rows left to assign.

The results of this heuristic can be easily refined by an iterative procedure that proceeds by comparing the rows assigned to a pair of processes and swapping them in case this leads to a better load balance.

The theoretical cost of this heuristic algorithm can be estimated as follows. A vector containing the number of nonzero entries in each matrix row needs first to be sorted, with a cost of $o(m \log_2 m)$. Then, the assignment procedure is performed on this sorted vector, which results in m binary searches for a cost of $o(m \log_2 m)$. Finally, each refinement iteration requires two linear searches on vectors of size m/p, for an average cost of m/p.

VI. Experimental Results

In this section we report experimental results for the sorting and sparse matrix-vector product on two different heterogeneous clusters consisting of 11 and 9 nodes (see Table II), connected in both cases via a *Fast Ethernet* (100 Mbit/sec.) switch. Cluster 1 has been used for the sorting operation while Cluster 2 is the testbed for the sparse matrix-vector product operation. Parallel codes were compiled and generated with the `mpich` version of the MPI compiler and optimizing flag `-O3`. Processors are added in the experiments starting with those classified as node type 0 (until there are no processors left of this type), then processors of node type 1, and so on. Our experiments are designed to compare the performances of the nonuniform data distribution vs. the uniform one (that is, one that does not take into account the different performances of the processors). As a measure of the efficacy we use the "speed-up" defined as

$$Sp = T_1/T_p,$$

where T_1 stands for the execution time of the serial algorithm in a single processor of node type 0, and T_p is the execution time of the parallel algorithm using p processors. Notice that node type 0 is chosen to be the fastest processor in the cluster 1, and the slowest one for cluster 2.

Name	Node type	Architecture	Processor Frequency	#Nodes : #Processors/node
Cluster 1	0	Intel Pentium Xeon	1.4 GHz	1 : 4
	1	AMD (Duron)	800 MHz	4 : 1
	2	AMD-K6	500 MHz	6 : 1
Cluster 2	0	Intel Pentium II	333 MHz	1 : 1
	1	Intel Pentium II	400 MHz	1 : 1
	2	Intel Pentium III	550 MHz	1 : 1
	3	AMD (Athlon)	900 MHz	5 : 1
	4	Intel Pentium Xeon	700 MHz	1 : 4

TABLE II

Heterogeneous platforms utilized in the experimental evaluation.

For the sorting operation we generate double data vectors with entries randomly distributed. Problem sizes vary from 1×10^6 until 26×10^6. Figures 4 and 5 contain the speed-up obtained with the nonuniform and uniform data distribution, respectively. The running times do not measure the initial data distribution. The uniform data layout produces the usual decrease in performance and peaks in the speed-up as a consequence of the introduction of slow processors. On the other hand, the use of a nonuniform data distribution results in a softer and improved speed-up curve (see Fig. 4). Comparison between the execution times of these two data layouts is made explicit in Fig. 6.

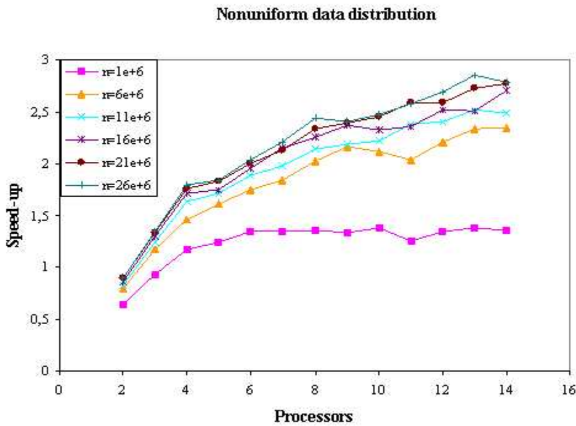

Fig. 4. Speed-up of the sorting operation with nonuniform data distribution in cluster 1.

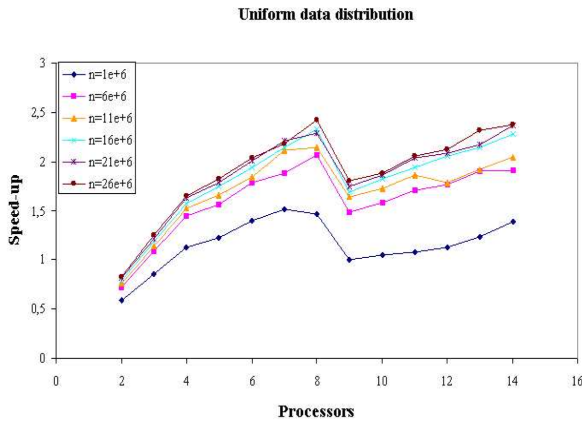

Fig. 5. Speed-up of the sorting operation with uniform data distribution in cluster 1.

We next carry out similar experiments for the sparse matrix-vector product. In the evaluation we utilize a moderately large sparse matrix from the *Matrix Market* collection (`http://math.nist.gov/MatrixMarket`), known as PSIMGR1, and second sparse matrix with randomly generated entries. The first matrix presents no regular structure, 543160 nonzero real entries, and 3140 rows/columns. Square unsymmetric random matrices are generated of order m=3000 and degrees of sparsity of 5% and 10%.

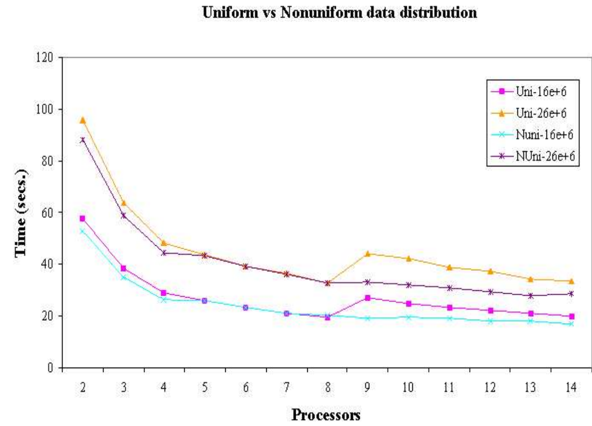

Fig. 6. Execution times of the sorting operation in cluster 1.

Speed-ups of the sparse matrix-vector product with data distributed following nonuniform and uniform layouts are reported in Figs. 7 and 8, respectively. The results show better speed-ups for the nonuniform distribution and also a better "scalability" when the number of processors is increased up to 9. Execution times for both data distributions are reported in Fig. 9. Here, we carry out 10 runs of the parallel codes in order to smooth possible spurious results. The nonuniform layout presents execution times that are more reduced than those of the uniform one, with the gap between the two data distributions being larger when there is more heterogeneity among the processors included in the experiment.

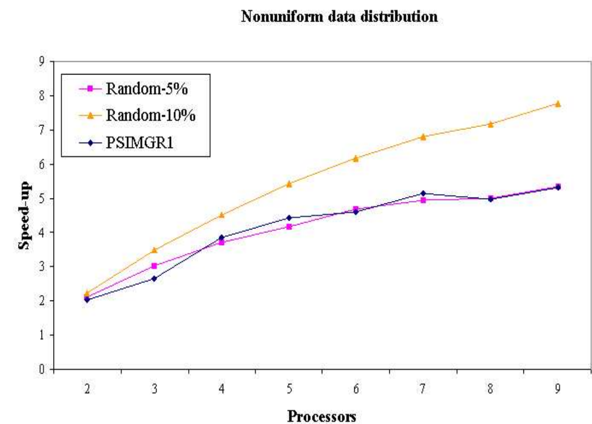

Fig. 7. Speed-up of the sparse matrix-vector product with nonuniform data distribution in cluster 2.

Table III reports the standard deviation of processor finishing times for computing the sparse matrix-vector product with and without the load balancing heuristic (that is, uniform and nonuniform distribution, respectively). Notice that, in the parallel matrix-vector product routine all communication is performed at the end of the computation, in a global reduce

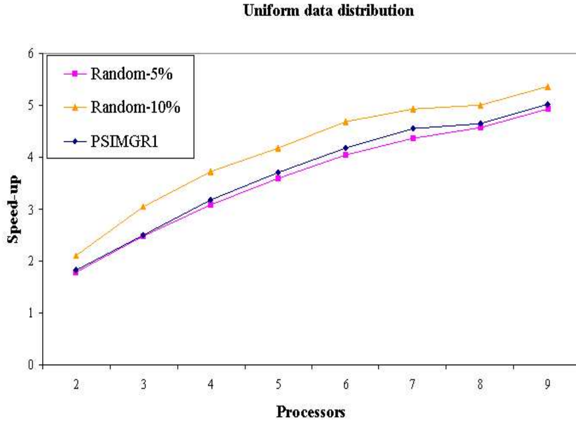

Fig. 8. Speed-up of the sparse matrix-vector product with uniform data distribution in cluster 2.

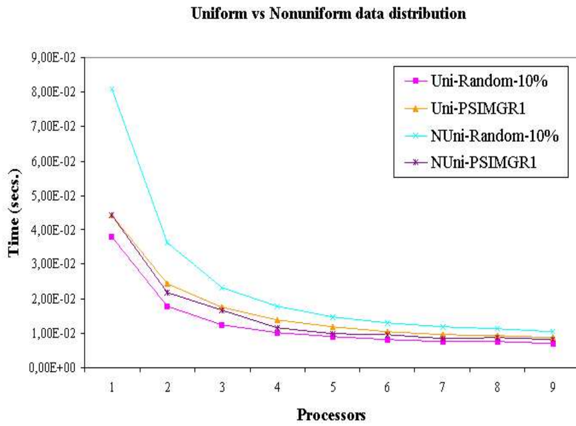

Fig. 9. Execution times of the sparse matrix-vector product in cluster 2.

operation. Therefore, this communication operation acts as a synchronization barrier for the processors, balancing the global execution times. Therefore, in order to evaluate the benefits from the proposed balancing heuristic, the results in the table only report the variance in the execution times required for the arithmetic operations. The table shows that the heuristic reduces the standard deviation of the finishing times roughly by an order of magnitude.

We also measured the time required to balance the matrix using our heuristic on a single processor resulting in 1.5e-2 sec. for the PSIMGR1 matrix. In this same processor, the sparse matrix-vector product required 4.4e-2 sec. Therefore, the cost of the balancing heuristic is about 33% of that of the matrix-vector product routine. As repeated matrix-vector products involving the same sparse matrix are usual in iterative procedures in sparse linear algebra, but matrix balancing only needs to be performed once, we expect this cost to be largely amortized over the computations.

	Random-5%		Random-10%		PSIMGR1	
#Proc.	Uni	NUni	Uni	NUni	Uni	NUni
2	4.0e-3	1.5e-4	8.1e-3	2.3e-4	4.5e-3	9.2e-5
3	2.6e-3	1.4e-4	5.2e-3	2.7e-4	3.1e-3	3.3e-3
4	1.8e-3	1.3e-4	3.6e-3	1.7e-4	1.9e-3	5.3e-5
5	1.3e-3	1.4e-4	2.6e-3	1.4e-4	1.4e-3	8.9e-5
6	1.0e-3	7.4e-5	2.0e-3	1.6e-4	1.1e-3	6.2e-5
7	8.3e-4	6.6e-5	1.7e-3	1.2e-4	9.4e-4	5.2e-5
8	6.8e-4	4.5e-5	1.4e-3	1.6e-4	7.2e-4	3.0e-4
9	5.7e-4	5.9e-5	1.2e-4	1.6e-4	6.0e-4	3.2e-4

TABLE III

STANDARD DEVIATION OF PROCESSOR FINISHING TIMES FOR THE SPARSE MATRIX-VECTOR PRODUCT.

VII. CONCLUSIONS AND FUTURE WORK

We have described the design and development of a parallel version of GNU Scientific Library for distributed-memory parallel architectures consisting of heterogeneous processors. Two simple operations coming from sparse linear algebra and discrete mathematics have been used to expose the architecture and interface of the system, and to report experimental results on the performance of the parallel library. These examples also serve to show that our approach can be extended to a wide range of GSL routines.

Our current efforts are focused on the definition of the architecture of the library, the specification of the interfaces, and the design and parallel implementation on heterogeneous systems of the complete parallel sparse BLAS operations, as well as certain numerical operations of special interest, such as FFTs, and discrete algorithms for optimization.

ACKNOWLEDGMENTS

This work has been partially supported by the EC (FEDER) and the Spanish MCyT (Plan Nacional de I+D+I, TIC2002-04498-C05-05 and TIC2002-04400-C03).

REFERENCES

[1] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi, *GNU scientific library reference manual*, July 2002, ed. 1.2, for GSL Version 1.2.

[2] L. Blackford, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, A. Petitet, H. Ren, K. Stanley, and R. Whaley, "Practical experience in the dangers of heterogeneous computing," University of Tennessee, LAPACK Working Note 112 CS-96-330, Apr. 1996.

[3] A. Grama, V. Kumar, A. Gupta, and G. Karypis, *An Introduction to Parallel Computing: Design and Analysis of Algorithms*, 2nd ed. Addison-Wesley, 2003.

[4] D. Skillicorn and D. Talia, "Models and languages for parallel computation," *ACM Computing Surveys*, vol. 30, no. 2, pp. 123–169, 1998.

[5] R. Buyya, *High Performance Cluster Computing*. Upper Saddle River, NJ: Prentice-Hall, 1999.

[6] B. Wilkinson and M. Allen, *Parallel Programming. Techniques and Applications using Networked Workstations and Parallel Computers*. Upper Saddle River, NJ: Prentice–Hall, 1995.

[7] A. Lastovetsky, *Parallel Computing on Heterogeneous Networks*. John Wiley & Sons, 2003.

[8] P. Fortier and H. Michel, *Computer systems performance evaluation and prediction*. Burlington, MA: Digital Press, 2003.

[9] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. New York, NY: Wiley-Interscience, 1991.

[10] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. Wong, and H. Shi, "On the versatility of parallel sorting by regular sampling," *Parallel Computing*, vol. 19, no. 10, pp. 1079–1103, 1993.

[11] O. Beaumont, V. Boudet, and A. Petitet, "A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers)," *IEEE Trans. Computers*, vol. 50, no. 10, pp. 1052–1070, 2001.

[12] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Load balancing strategies for dense linear algebra kernels on heterogeneous two-dimensional grids," in *14th Int. Parallel and Distributed Processing Symposium (IPDPS'2000)*, 2000, pp. 783–792.

[13] O. Beaumont, A. Legrand, F. Rastello, and Y. Robert, "Dense linear algebra kernels on heterogeneous platforms: Redistribution issues," *Parallel Computing*, vol. 28, pp. 155–185, 2002.

[14] O. Beaumont, A. Legrand, and Y. Robert, "Static scheduling strategies for dense linear algebra kernels on heterogeneous clusters," in *Parallel Matrix Algorithms and Applications*. Universit´e de Neuchâtel, 2002.

[15] E. Dovolnov, A. Kalinov, and S. Klimov, "Natural block data decomposition of heterogeneous computers," in *17th Int. Parallel and Distributed Processing Symposium (IPDPS'2003)*, 2003.

[16] A. Kalinov and A. Lastovetsky, "Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers," *Journal of Parallel and Distributed Computing*, vol. 61, no. 4, pp. 520–535, 2001.

[17] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley, *ScaLAPACK Users' Guide*, SIAM, Philadelphia, PA, 1997.

[18] R. A. van de Geijn, *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.

[19] I. Duff, M. Heroux, and R. Pozo, "An overview of the sparse basic linear algebra subprograms," *ACM Trans. Math. Software*, vol. 28, no. 2, pp. 239–267, 2002.

[20] I. Duff, A. Erisman, and J. Reid, *Direct Methods for Sparse Matrices*. Oxford, UK: Clarendon Press, 1986.

[21] Y. Saad, "Iterative methods for sparse linear systems," January 2000, 2nd edition with corrections; available at *http://www-users-cs.umn.edu/~saad/books.html*.