Parallelization of GSL: Architecture, Interfaces, and Programming Models*

J. Aliaga[1], F. Almeida[2], J.M. Badía[1], S. Barrachina[1], V. Blanco[2], M. Castillo[1], U. Dorta[2], R. Mayo[1], E.S. Quintana[1], G. Quintana[1], C. Rodríguez[2], and F. de Sande[2]

[1] Depto. de Ingeniería y Ciencia de Computadores
Univ. Jaume I, 12.071–Castellón, Spain
{aliaga,badia,castillo,mayo,quintana,gquintan}@icc.uji.es
[2] Depto. de Estadística, Investigación Operativa y Computación
Univ. de La Laguna, 38.271–La Laguna, Spain
{falmeida,vblanco,casiano,fsande}@ull.es

Abstract. In this paper we present our efforts towards the design and development of a parallel version of the Scientific Library from GNU using MPI and OpenMP. Two well-known operations arising in discrete mathematics and sparse linear algebra illustrate the architecture and interfaces of the system. Our approach, though being a general high-level proposal, achieves for these two particular examples a performance close to that obtained by an *ad hoc* parallel programming implementation.

1 Introduction

The GNU Scientific Library (GSL) [2] is a collection of hundreds of routines for numerical scientific computations coded.Although there is currently no parallel version of GSL, probably due to the lack of an accepted standard for developing parallel applications when the project started, we believe that with the introduction of MPI and OpenMP the situation has changed substantially.

We present here our joint efforts towards the parallelization of of GSL using MPI and OpenMP. In particular, we plan our library be portable to several parallel architectures, including distributed and shared-memory multiprocessors, hybrid systems -consisting of a combination of both types of architectures-, and clusters of heterogeneous nodes. Besides, we want to reach two different classes of users: a programmer with an average knowledge of the C programming language but with no experience in parallel programming, that will be denoted as user A, and a second programmer, or user B, that regularly utilizes MPI or OpenMP. As a general goal, the routines included in our library should execute efficiently on the target parallel architecture and, equally important, the library should appear to user A as a collection of traditional serial routines. 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

* Supported by MCyT projects TIC2002-04400-C03, TIC2002-04498-C05-05.

D. Kranzlmüller et al. (Eds.): EuroPVM/MPI 2004, LNCS 3241, pp. 199–206, 2004.

architectures. Moreover, we offer the user a sequential interface while trying to avoid the usual loss of performance of high-level parallel programming tools.

In this paper we describe the software architecture of our parallel integrated library. We also explore the interface of the different levels of the system and the challenges in meeting the most widely-used parallel programming models using two classical numerical computations. Specifically, we employ the operation of sorting a vector and the USAXPY (unstructured sparse α times x plus y) operation [1]. Although we illustrate the approach with two simple operations, 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 parallelization of a certain part of GSL. Parallelizing the complete GSL can then be considered as a labor of software reusability. Many existing parallel routines can be adapted without too much effort to use our interfaces while, in some other cases, the parallelization will require a larger code re-elaboration.

The rest of the paper is structured as follows. In Section 2 we describe the software architecture of our parallel integrated library for numerical scientific computing. Then, in Sections 3–5, we describe the functionality and details of the different levels of the architecture from top to bottom. Finally, some concluding remarks follow in Section 6.

2 Software Architecture of the Parallel Integrated Library

Our library has been designed as a multilevel software architecture; see Fig. 1. Thus, 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 parallelism to user A and supplies the services through C/C++ functions according to the prototypes specified by the sequential GSL interface (for example, a `gsl_sort_vector()` routine is provided to sort a `gsl_vector` data array).

The *Programming Model Level* provides a different instantiation of the GSL library for each one of the computational models: sequential, distributed-memory, shared-memory, and hybrid. The semantics of the functions in the Programming Model Level are those of the parallel case so that user B can invoke them directly from her own parallel programs. The function prototypes and data types in user A codes are mapped into the appropriate ones of this level by just a renaming procedure at compilation time. The Programming Model Level implements the services for the upper level using standard libraries and parallelizing tools like (the sequential) GSL, MPI, and OpenMP. In the distributed-memory (or message-passing) programming model we view the parallel application as being executed by p *peer* processes, P_0, P_1,...,P_{p-1}, where the same parallel code is executed by all processes on different data.

In the *Physical Architecture Level* the design includes shared-memory platforms, distributed-memory architectures, and hybrid and heterogeneous systems

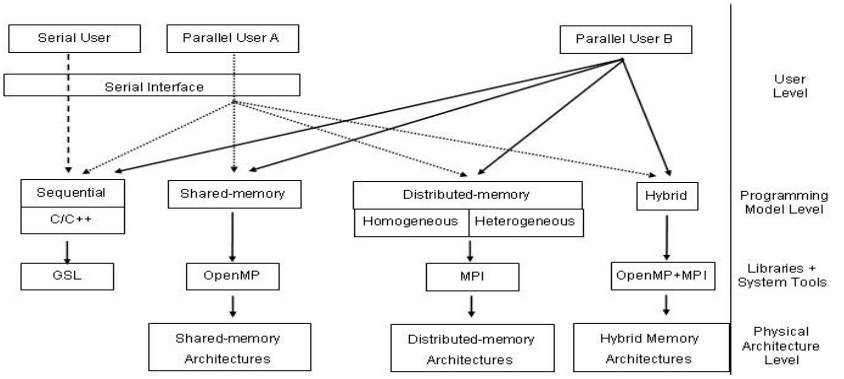

Fig. 1. Software architecture of the parallel integrated library for numerical scientific computing.

(clusters of nodes with shared-memory and processors with different capabilities). 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. The performance of the parallel routines will depend on the adequacy between the programming paradigm chosen by the user and the target architecture.

In the following sections we review the functionality, interfaces, and further details of the different levels of the software architecture, from top to bottom.

3 User Level

The basic purpose of this level is to present user A with the classical interface and interaction of the sequential GSL routines. To illustrate the challenges involved in this task, consider the following program which reads two vectors, computes their USAXPY, and outputs the result:

```
1:  #include <gsl_sparse_vector.h>
2:  void main (int argc, char * argv[]) {
      ...
3:    scanf ("Value of nz %u", &nz);
4:    y = gsl_vector_alloc (n);
5:    x = gsl_sparse_vector_alloc (n, nz); // Allocate
6:    gsl_vector_scanf (y, n);
7:    gsl_sparse_vector_scanf (x, n, nz);
8:    gsl_usaxpy (alpha, x, y);            // USAXPY operation
9:    printf ("Result y = alpha x + y\n"); // Output
10:   gsl_vector_printf (y, n);
11:   gsl_vector_free (y);                 // Deallocate
12:   gsl_sparse_vector_free (x); }
```

What a serial user in general expects is to compile this program using, e.g., the `make` utility, and execute the resulting runnable code from the command line. We offer the user the proper `Makefile` that at compilation time, depending on the programming model selected, maps (renames) the function prototypes into

the appropriate ones of the Programming Model Level. This mapping includes user A's data I/O routines, the `main()` function in the code, and the data types. We also provide several `gslrun` scripts (one per programming model) to launch the execution of the program.

The code does not need to be transformed further in case the parallelization is targeted to a shared-memory programming model. Parallelism is exploited in this case inside the corresponding parallel GSL routines.

Nevertheless, when the code is targeted to the distributed-memory programming model, we still need to deal with a few problems. First, notice that following our *peer* processes approach, the execution of the user's program becomes the execution of p processes running in parallel, where user's data I/O is performed from a single process. Also, a different question to be considered is that of error and exception handling. Finally, some execution errors due to the parallel nature of the routines cannot be masked and must be reported to the end-user as such.

4 Programming Model Level

In this section we first review the interface of the routines in the parallel instantiations of the GSL library, and we then describe some details of the major parallelization approaches utilized in the library.

4.1 Programming Model Interface

All programming models present similar interfaces at this level. As an example, Table 1 relates the names of several sequential User Level routines with those of the different instantiations of the parallel library. The letters "`sm`", "`dm`", and "`hs`" after the GSL prefix ("`gsl_`") denote the programming model: shared-memory, distributed-memory, and hybrid systems, respectively. In the distributed-memory model, the following two letters, "`rd`" or "`dd`", specify whether the data are replicated or distributed.

Table 1. Mapping of User Level routines to the corresponding parallel routines.

User Level	Programming Model Level		
Sequential	Shared-memory	Distributed-memory	Hybrid
`fscanf()`	`fscanf()`	`gsl_dmrd_fscanf()` `gsl_dmdd_fscanf()`	`gsl_hs_fscanf()`
`gsl_sort_vector()`	`gsl_sm_sort_vector()`	`gsl_dmrd_sort_vector()` `gsl_dmdd_sort_vector()`	`gsl_hs_sort_vector()`

At the Programming Model Level the interface supplied to the User Level is also available as a parallel user-friendly interface to user B. The parallel routines can thus be employed as building blocks for more complex parallel programs.

A sorting routine 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:     MPI_Init (& argc, & argv);
5:     gsl_dmdd_set_context (MPI_COMM_WORLD); // Allocate
6:     gsl_dmdd_scanf ("Value of n %u", &n);  // Read
7:     v = gsl_dmdd_vector_alloc (n, n);      // Block-Cyclic Allocation
8:     gsl_dmdd_vector_scanf (v, n);
9:     status = gsl_dmdd_sort_vector (v);     // Sorting operation
10:    printf ("Test sorting: %d\n", status); // Output
11:    gsl_dmdd_vector_free (v);              // Deallocate
12:    MPI_Finalize (); }
```

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. The MPI program above assumes the vector to sort to be distributed among all processes so that, when routine `gsl_dmdd_vector_alloc(n, cs)` is invoked, the allocation for the `n` elements of a `gsl_vector` is distributed among the whole set of processors following a block-cyclic distribution policy with cycle size `cs`. In the case of heterogeneous systems, the block sizes assigned depend on the performance of the target processors. The call to `gsl_dmdd_sort_vector` sorts the distributed `vector` following the PSRS algorithm [3] described later.

4.2 Implementation in the Distributed-Memory Programming Model

Our library currently supports two data distributions: In the replicated layout a copy of the data is stored by all processes. In the distributed layout the data are partitioned into a certain number of blocks and each process owns a part of these blocks; in heterogeneous systems the partitioning takes into consideration the different computational capabilities of the processors where the processes will be mapped.

All I/O routines in the distributed-memory programming model (e.g., routine `gsl_dmdd_fscanf`) perform the actual input from P_0 and any value read from the input is then broadcasted to the rest of processes; analogously, any data to be sent to the output is first collected to P_0 from the appropriate process. While scalar data can be easily replicated using the policy just described, replication of GSL arrays has to be avoided in order to minimize memory requirements. This implies parallelizing the routines of GSL which deal with these derived data types. Notice that data I/O performed by user B in her program directly refers to the routines in the `stdio` library and therefore is not mapped to the I/O routines in the distributed-memory programming model.

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 [3]. This algorithm was conceived for distributed-memory architectures with

homogeneous nodes and has good load balancing properties, modest communication requirements, and a reasonable locality of reference in memory accesses.

The PSRS algorithm is composed of the following five stages:

1. Each process sorts its local data, chooses $p-1$ "pivots", and sends them to P_0. The stride used to select the samples is, in the case of heterogeneous contexts, different on each processor and is calculated in terms of the size of the local array to sort.
2. Process P_0 sorts the collected elements, finds $p-1$ pivots, and broadcasts them to the remaining processes. Again, for the heterogeneous systems, 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.

The case where the vector to sort is replicated poses no special difficulties: a simple adjustment of pointers allows the processes to limit themselves to work with their corresponding portions of the vector. Only stage 5 implies a redistribution. When the output vector is replicated, each process has to broadcast its chunk. Redistribution is also required even for a vector that is distributed by blocks, since the resulting chunk sizes after stage 4 in general do not fit into a proper block distribution. The average time spent in this last redistribution is proportional to the final global imbalance.

4.3 Implementation in the Shared-Memory Programming Model

In this subsection we explore the parallelization on a shared-memory parallel architecture of the USAXPY operation using OpenMP. A simplified serial implementation of this operation is given by the following loop:

```
1:    for (i = 0; i < nz; i++) {
2:      iy = indx [i];
3:      y [iy] += alpha * valx [i];  }
```

As specified by the BLAS-TF standard [1], sparse vectors are stored using two arrays: one with the nonzero values and the other an integer array holding their respective indices (`valx` and `indx` in our example).

The USAXPY operation, as many others arising in linear algebra, is a typical example of a routine that spends much of its time executing a loop. Parallel shared-memory architectures usually reduce the execution time of these routines by the executing iterations of the loops in parallel across multiple processors. In particular, the OpenMP compiler generates code that makes use of *threads* to execute the iterations concurrently.

The parallelization of such a loop using OpenMP is quite straight-forward. We only need to add a `parallel for` compiler directive, or *pragma*, before the loop, and declare the scope of variable `iy` as being `private`. As all iterations perform the same amount of work, a *static schedule* will produce an almost perfect load balancing with a minimum overhead.

Table 2. Characteristics of the platforms handled in the experimental evaluation.

Type	Manufacturer/ Architecture	Processor Frequency	Memory size (RAM/Cache)	#Proc.	Commun. Network
PC cluster distributed-memory	Intel Pentium Xeon	2.4GHz	1GB/512KB L2	34	Gigabit Ethernet Myrinet
Heterogeneous Cluster	Intel Pentium Xeon AMD (Duron) AMD-K6	1.4 GHz 800 MHz 500 MHz	2GB/512 L2 256MB/64 L2 256MB/64 L2	4 4 6	Fast Ethernet
CC-NUMA SGI Origin 3800	SGI MIPS R14000	500MHz	1GB/16MB L2	160	Crossbar Hypercube
Shared-memory	Intel Pentium Xeon	700MHz	1GB/1MB L2	4	System bus

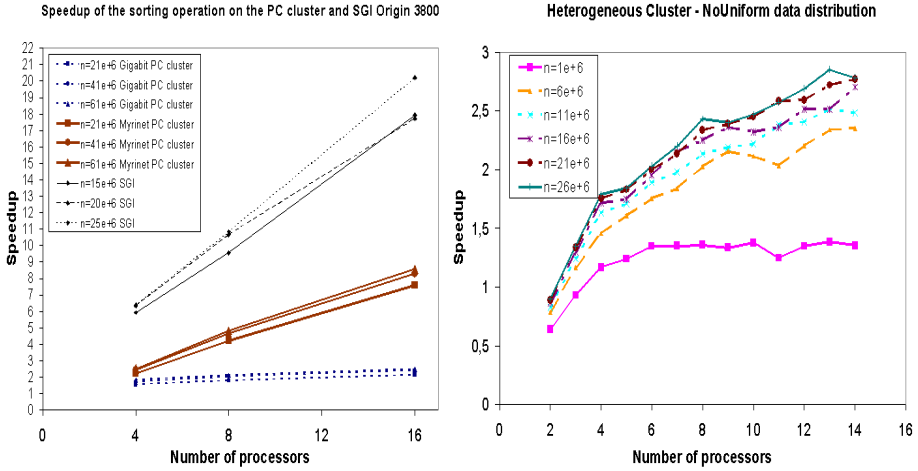

Fig. 2. Parallel performance of the Sorting case studies.

5 Physical Architecture Level

In this section we report experimental results for the parallel versions of the sorting and USAXPY operations on several platforms; see Table 2.

The parallel PSRS algorithm is evaluated on three different platforms (see Fig. 2): a PC cluster (using a distributed data layout), a SGI Origin 3800 (with a replicated data layout), and a heterogeneous cluster (with a nonuniform data distribution). In the PC cluster we used MPICH 1.2.5 for the Gigabit Ethernet and GMMPI 1.6.3 for the Myrinet network. The native compiler was used for the SGI Origin 3800. The MPICH 1.2.5 implementation was also employed for the heterogeneous cluster. The use of the Myrinet network instead of the Gigabit Ethernet in the PC cluster achieves a considerable reduction in the execution time. The parallel algorithm presents acceptable parallel performances. As could be expected, the execution time is reduced in both architectures when the number of processors is increased. We observed super-linear speed-ups in the SGI platform that are a consequence of a better use of the cache memory in the parallel algorithms. Due to the heterogeneous data distribution, better speed-ups are also achieved in the heterogeneous cluster.

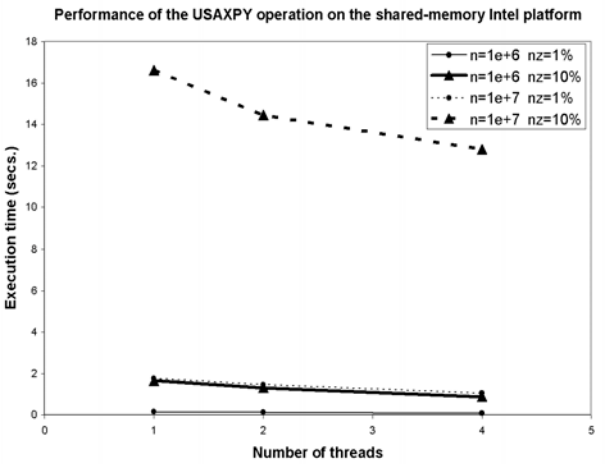

Fig. 3. Parallel performance of the USAXPY case studie.

The parallel performances reported for USAXPY operation on the shared-memory Intel platform were obtained using the Omni 1.6 OpenMP compiler (http://phase.hpcc.jp/Omni). The results in Fig. 3 show a moderate reduction in the execution time of the operation when the problem size is large enough. We believe the poor results for the smaller problems to be due to a failure of the OpenMP compiler to recognize the pure parallel nature of the operations, but further experimentation is needed here.

6 Conclusions

We have described the design and development of an integrated problem solving environment for scientific applications based on the GNU Scientific Library. Our library is portable to multiple classes of architectures and targets also a class of users with no previous experience in parallel programming.

Two simple operations coming from sparse linear algebra and discrete mathematics have been used here to expose the architecture and interface of the system, and to report preliminary results on the performance of the parallel library.

References

1. I.S. Duff, M.A. Heroux, and R. Pozo. An overview of the sparse basic linear algebra subprograms. *ACM Trans. Math. Software*, 28(2):239–267, 2002.
2. 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.
3. X. Li, P. Lu, J. Schaeffer, J. Shillington, P.S. Wong, and H. Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19(10):1079–1103, 1993.