Parallelization of GSL on Clusters of Symmetric Multiprocessors[*]

J. Aliaga[a], F. Almeida[b], J.M. Badía[a] , S. Barrachina[a], V. Blanco[b], M. Castillo[a], R. Mayo[a], E.S. Quintana[a], G. Quintana[a], C. Rodríguez[b], F. de Sande[b], A. Santos[b]

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

[b]Depto. de Estadística, Investigación Operativa y Computación, Univ. de La Laguna, 38.271–La Laguna, Spain; {falmeida,vblanco,casiano,fsande}@ull.es.

In this paper we discuss the application of an hybrid programming paradigm that combines message-passing (MPI) with shared memory programming (OpenMP). We apply this model to the parallel solution of two basic problems: the sparse matrix-vector product and the dynamic programming problem. We compare the results of the hybrid model with the application of a pure MPI model on a cluster of dual Intel Xeon processors. The experimental results show that the behavior of both models depend, among other factors, on the application and on the size of the problems. While with the dynamic programming problem we obtain very good speedups, in the case of the matrix-vector product the algorithms do not take very good profit of the dual processors.

1. Introduction

The GNU Scientific Library (GSL) [6] is a collection of hundreds of routines for numerical scientific computations written in ANSI C, which includes codes for complex arithmetic, matrices and vectors, linear algebra, integration, statistics, and optimization, among others. The reason GSL was never parallelized seems to be the lack of a globally accepted standard for developing parallel applications. We believe that with the introduction of OpenMP and MPI the situation has changed substantially. OpenMP [8] has become a standard *de facto* for exploiting parallelism using *threads*, while MPI [7] is nowadays accepted as the standard interface for developing parallel applications following the message-passing programming model.

On the other hand, the application of parallelism has increased in recent years partly due to the arising of the clusters of personal computers. This kind of parallel architecture can be easily upgraded and offers a very good performance/price relation. Clusters are naturally programmed using the message-passing model. However, most recent clusters are composed of nodes that are small shared-memory multiprocessors containing from 2 to 8 standard personal computers. A trend of these hybrid parallel computers could be the inclusion of multi-core processors, while SMPs are being clustered to increase the number of CPUs of the architecture.

It seems that a natural, efficient and portable way to take profit of hybrid architectures is to combine the MPI library to communicate the different nodes, with OpenMP to run several threads on the processors included in each node of the cluster [11], [3], [1].

This hybrid paradigm allows us to exploit efficiently the shared memory without having to add MPI communications within each node. Besides, this paradigm could combine coarse grain parallelism among the nodes with fine grain parallelism within them, exploiting besides the dynamic load balancing possibilities of the OpenMP model. However, there is not guarantee that the hy-

[*]This work has been supported by the EC (FEDER) and the Spanish MCyT (Plan Nacional de I+D+I, TIC2002-04400-C03-03 and TIN2005-09037-C02).

brid model obtains better performance that a pure MPI implementation, [2]. Besides, other parallel programming models can also be used on clusters of SMPs [9].

In this paper we investigate the parallelization of a specific part of GSL using the hybrid parallel model, and we compare it with the use of a pure message-passing model based on the MPI library. In Section 2 we elaborate our first case study, describing a parallelization of the sparse matrix-vector product addressed to SMPs, and the experimental results. In Section 3 we perform the same kind of description and analysis in the case of the dynamic programming problem. Finally, concluding remarks follow in Section 4.

2. Case study I: sparse matrix-vector product

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 [5] for the interface design of the *Basic Linear Algebra Subprograms* (BLAS) for unstructured 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 [4]. As there seems to be no definitive advantage of any of the above-mentioned schemes, in our codes we employ a variant of 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. Each process operates then with the elements of its corresponding block of rows.

The following (simplified) data structure is used to manage distributed sparse matrices on each processor:

```
typedef struct {
  size_t local_size1;   // local row size
  size_t local_size2;   // Global column size
  size_t local_nz;      // Global non-zeros
  int **rowptr; // pointers to rows
  void *local_data;
} internal_dmdd_sparse_matrix;
```

In this structure, `rowptr` stores pointers to the init of each row in the vector `local_data`. This vector stores the matrix elements in the processor by rows. Each element is stored as three adjacent values: `(row_index, column_index, element_value)`. We have chosen this storage scheme instead of the three vectors of the classical coordinate format in order to improve the locality in the access to the elements of the matrix.

2.1. Parallelizing the Sparse Matrix-Vector Product

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

$y \leftarrow y + \alpha \cdot A \cdot x,$

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 [10], 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 (2.1) by its transpose or its conjugate transpose.

First we will describe the parallelization of the product using a message-passing model, and then we will describe how we can modify it so that each process executes several threads on a shared-memory environment.

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 the vector x involved in the product is 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 parallelism of the rowwise version of the product arises from the fact that every dot product among a row of the distributed matrix A and the replied vector x can be performed fully in parallel to obtain different elements of the solution vector y. Therefore, the code executed on each process in the pure message-passing version of the product is the following:

```
for (i = 0; i < local_size1; i++) {
  pos = rowptr[i];  // first element in row i
  k = row_index(pos);
  temp = 0.0;
  for (j =  0; j < nz_in_row(i); j++) {
    temp += value_in(pos) * x[k];
    inc_coordinate(pos);
  }
  y[k] += alpha * temp;
}
```

In the previous code, `row_index(pos)` is a macro that obtains the row index of the matrix element from the position `pos` in `local_data`. Another C macro, `nz_in_row(i)`, returns the number of non-zero elements in the `i-th` row. Finally, `inc_coordinate(pos)` shifts the position `pos` to the next element of the matrix.

Once each process has computed a block of the solution vector y, a collective communication (of type `MPI_Allgather`) is required to replicate the results. This operation is (almost) perfectly balanced as all processes have a close number of elements of y.

Let us now describe the hybrid version of the algorithm. Suppose that each MPI process is executed on a node that is a shared memory multiprocessor. An easy and portable way to take profit of this kind of architecture is to use OpenMP in order to distribute the computations of each process among the different processors of each node. Exploiting the same idea about the parallelism of the matrix-vector product than before, we can compute in parallel the iterations of the outer loop of the previous algorithm, corresponding to independent dot products. This idea can be implemented by adding the following directive:

```
#pragma omp parallel for private (pos, k, temp, j)
```

just before the outer loop. A certain amount of dot products are thus computed by each thread and the threads only need to be synchronized once, on termination of the outer loop. If the non-zero elements of the matrix are not evenly distributed among the different rows, a dynamic scheduling of the threads provides a better load balancing. This could be one of the main advantages of using OpenMP threads on each node instead of MPI processes, where load balancing should be implemented by the programmer by means of an appropriate data and task distribution.

It is worth noticing that the global communication among the MPI processes is performed once finished the OpenMP parallel region. The different threads have partially contributed to the computation of the solution vector corresponding to the each MPI process. Then, the master thread on each node can access that information and perform the global communication. Since the communication is out of the OpenMP parallel region, theoretically a non-thread safe version of MPI could be used.

2.2. Experimental analysis

Our experimental platform is a dual-SMP cluster of 34 nodes. Each node consists of two Intel Xeon Processors at 2.4 Ghz., 1GB of RAM and 512 KB of L2 cache. The nodes are connected through a Myrinet network with a bandwidth of 2Gb/s. Each node runs a Linux kernel version 2.4.24bi and we used Intel icc compiler version 7.1 with the option -O3. The algorithms have been implemented with the MPI v.1.2.5 compiled with the `-ch_gm` option in order to exploit the GM communication library of the Myrinet network.

We have performed experiments with three versions of the parallel matrix-vector product:

- An *uniprocessor* version in which we run the pure MPI algorithm using only one of the processors of each node.

- A *pure MPI* version in which we have run the same pure MPI algorithm spawning two MPI processes on each node. In order to exploit the shared memory for the intra-node communications we used the option `--gm-numa-shmem` of the `mpirun.ch_gm` command.

- A *hybrid MPI+OpenMP* version in which we run a MPI process per node and this process spawns two OpenMP threads to perform the computation.

We have tested the performance of the three versions with different problem sizes (sparsity and matrix size) and using different number of processors. The left-hand side of Fig. 1 shows the speed-ups of the parallel algorithms with respect to the sequential algorithm. The figure shows the results for matrices of two sizes: 30.000 (30k) and 40.000 (40k), both cases with an sparsity of 2%.

The speed-ups are far from the optimal values. This is mainly due to the effect of the communication cost in an application with a computation cost depending on a small number of non-zero elements. On the other hand, as we use random generated matrices, the number of non-zero elements on each processor is balanced, and so the number of flops. However the distribution of the elements in the columns is quite different on the different processors, producing a different access pattern to the memory and unbalancing the total computational cost.

The experiments show also that the speed-up of the hybrid version overcomes the speed-up of the uniprocessor version when we take profit of the dual nodes of the machine. However the pure MPI version of this application in our experimental environment is only better than the uniprocessor version for some problem sizes, for example, matrices with 40k rows. We can also see that, as we increase the size of the matrix the speed-up of the pure MPI version approaches the speed-up of

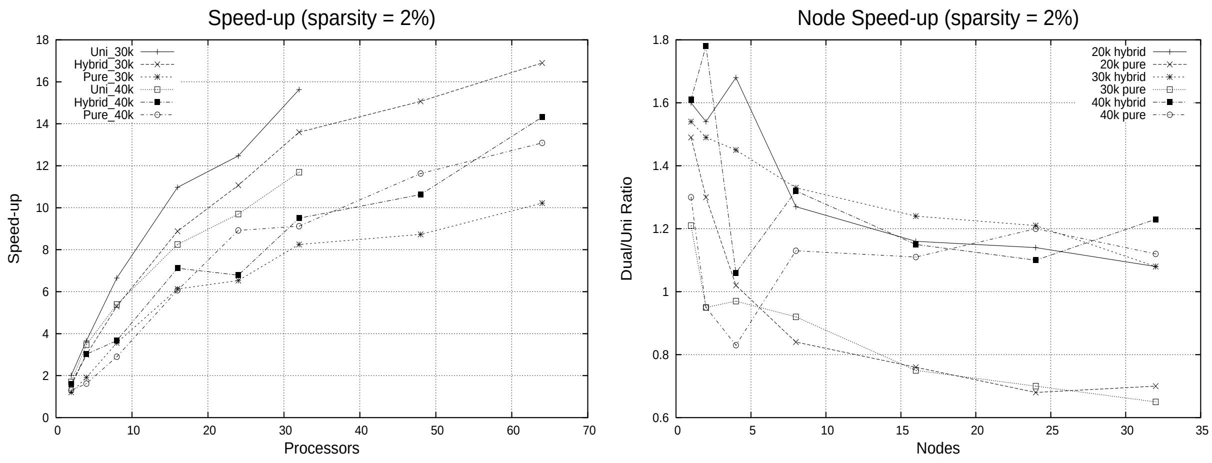


Figure 1. Performance of the parallel versions of the sparse matrix-vector product.

the hybrid version, and in some cases even surpasses it. In order to analyze the use of the hybrid architecture we plot in the right-hand side of Fig. 1 the *Node speed-up* [2]. Given a fixed number of nodes, this parameter is the ratio between the time spent using the two processors on each node and using only one processor per node. The figure shows that the results obtained are always suboptimal (< 2). In some cases, the pure MPI versions offers node speed-ups less than 1, which means that it would have been better to run only one process per node than to run two processes trying to use both processors.

We could expect that, for a fixed number of processors, the uniprocessor version offers better results than the pure and hybrid versions, because in the last two cases the two processes or threads running on each node have to share its resources (memory, network, etc.). However, the results shown in Fig. 1 are worse than expected even having into account the previous kind of collision. The detailed justification of the experimental results requires a deeper analysis of the influence of several factors, including the communication and computation costs, the use and access to the memory, etc. We intend to develop this analysis of the application and the experimental environment in a near future.

3. Case study II: dynamic programming

Dynamic programming (DP) is an important problem-solving technique that has been widely used in various fields such as control theory, operations research, biology and computer science. Dynamic programming is a useful method when the solution of a problem can be expressed as the result of a sequence of decisions. The method enumerates all decision sequences and selects the best among them. Dynamic Programming often reduces the amount of enumeration drastically by avoiding the consideration of decision sequences that will not deliver optimal solutions. An optimal sequence of decisions is obtained by making explicit use of the principle of optimality. This approach derivates into a general recurrence formula where, on a multistage-problem, the optimal values for subproblems involving i decisions are computed in terms of subproblems involving $i - 1$ decisions.

For instance, in the Single Resource Allocation Problem, we need to allocate M units of an indivisible resource to N tasks so that the sum of the effectiveness is maximized. The problem can be formally stated as:

$$\max\ z\ = \sum_{j=1}^{N} f_j(x_j) \quad \text{subject to } \sum_{j=1}^{N} x_j = M,$$

where $f_j(x_j)$ gives the benefit of allocating x_j units of resource to task j.

Now denote by $G[i][x]$ the optimal benefit of the subproblem and consider the first i tasks and x units of resource. The dynamic programming recurrence equations are then formulated as follows:

$$\begin{array}{rcl} G[i][x] & = & \max\{G[i-1][x-j] + f_i(j) : 0 < j \leq x\}, \quad i = 2, ..., N, \\ G[1][x] & = & f_1(x), \quad 0 < x \leq M, \quad \text{and} \\ G[i][x] & = & 0, \quad i = 1, ..., N; \quad x = 0. \end{array}$$

The value $G[N][M]$ gives the total income for the optimization problem and, in order to be computed, the dynamic programming table G is needed in advance.

3.1. Parallelizing the Dynamic Programming Problem

The Dynamic Programming table is necessary to obtain the optimal solution, and the values of the table must be computed following the order imposed by the dependences of the recurrence equations. A simple parallelization of this operation on a distributed-memory platform using MPI replicates the table on all nodes. With G distributed by columns, the processes compute in parallel the entries belonging to the same row of the table, with the rows being computed sequentially from top to bottom. We next show the code executed on the $myid$ MPI process:

```
for (i = 0; i <= N_TASKS; i++) {
   for (x = displs[myid]; x < displs[myid + 1]; x++) {
      G[i][x] = (*f)(i, 0);
      for (j = 0; j <= x; j++) {
         fix = G[i - 1][x - j] + (*f)(i, j);
         if (G[i][x] > fix)
            G[i][x] = fix;
      }
   }
   MPI_Allgatherv(&G[i][displs[myid]], ... );
}
```

An *all-to-all* communication operation is performed to replicate every row. Since the inner loop ($0 < j \leq x$) is not constant, we follow a block distribution with variable block sizes. The sizes of the blocks are computed so that they minimize the load imbalance. The number of columns assigned to each processor are computed as an arithmetic progression, and the starting indexes are stored in vector `displs`. This evaluation also forces the use of different parallel regions in the OpenMP version.

The following code summarizes the hybrid version of the algorithm.

```
#pragma omp parallel private(...)
for (i = 0; i <= N_TASKS; i++)
   for (x = th_displs[th_id]; x < th_displs[th_id + 1]; x++)
        // idem than in the pure MPI version
   #pragma omp barrier
   #pragma omp master
     MPI_Allgatherv(&G[i][th_displs[th_id]], ... );
```

The hybrid algorithm exploits the same kind of parallelism than the pure MPI version on each node. The columns assigned to each MPI process, and so the iterations of the loop on line 3, are distributed among several OpenMP threads. In order to balance the load we can distribute a different number of columns to each thread following the same kind of distribution than in the pure MPI version. The starting column assigned to each thread is stored in vector `th_displs`.

On the other hand, each time the algorithm finishes the computations corresponding to one of the rows of the matrix, the MPI processes have to perform an *all-to-all* communication operation. This communication can only be performed when all the threads have finished their iterations of the loop on line 3, and so we added an OpenMP barrier on line 5. Besides, we used a `omp master` directive to guarantee that only the master thread participates on the MPI communication.

3.2. Experimental analysis

In order to perform the experimental analysis of this second case of study we used the same experimental environment that with the sparse matrix-vector product. We tested the same three versions of the algorithm (uniprocessor, pure MPI and hybrid MPI + OpenMP) with different problem sizes: 1000 (1k) and 3000 (3k).

The left-hand side of Fig. 2 reports the speed-ups of the three versions of the algorithm. The speed-ups are very good for all the versions of the algorithm and they increase with the size of the problem. We can even observe super-linear speed-ups with the largest problem, that are probably a consequence of a better use of the cache memory in the parallel algorithms.

Regarding the relative behavior of the three versions of the algorithm, we can see that both, the pure MPI and the hybrid version, continue the almost linear increasing of the speed-ups of the uniprocessor version when we exploit both processors of the nodes. The right-hand side of the figure 2 shows that the node speed-ups are always larger than 1 and in some cases even larger than 2. Moreover, the node speed-ups increase with the problem size, and they are almost independent of the number of nodes in the case of the largest problem size.

Besides, for the largest problem size, the hybrid parallelization offers better speed-ups than the pure MPI one. The performance behavior, is reversed for the smallest problem size where speed-ups of the pure MPI code surpass the speed-ups of the hybrid version.

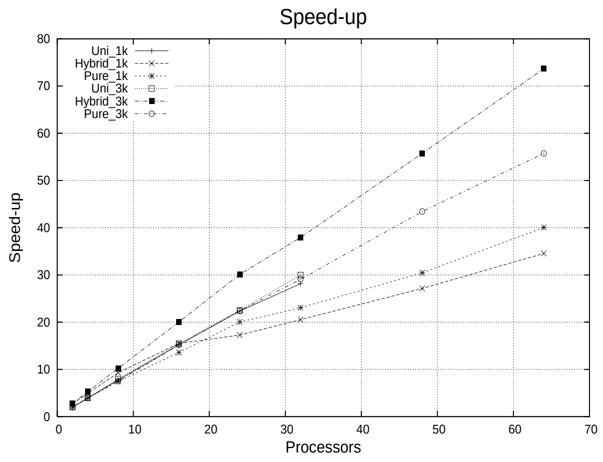
Although even better performances could be achieved using tuned tiled block-cyclic parallelizations, this requires information on the optimal tile sizes a priori. Since these sizes are architectural and problem dependent, the approach seems to be hardly suitable for a general purpose library.

4. Conclusions

In this paper we study different models to parallelize algorithms on clusters of SMPs. Specifically we study two basic models: a pure MPI model and a hybrid MPI + OpenMP model. In the first case a MPI process is executed on each processor. In this second model an MPI process is executed on each node and OpenMP threads cooperate to perform the task assigned to each process.

In order to compare both models we used two applications: the sparse matrix-vector product and the dynamic programming problem. Both applications use different schemes of parallelization. In the matrix-vector product the algorithm starts by computing the product without communications, and then an all-to-all communication is performed to gather the result in all the processes. The dynamic programming problem is solved by means of succession of steps. Each one starts with a computation phase and finishes with a communication phase that synchronizes all the processes.

The experimental results on a cluster of dual processors show that the performance mainly depends on the application and on the problem size. While in the case of the dynamic programming problem



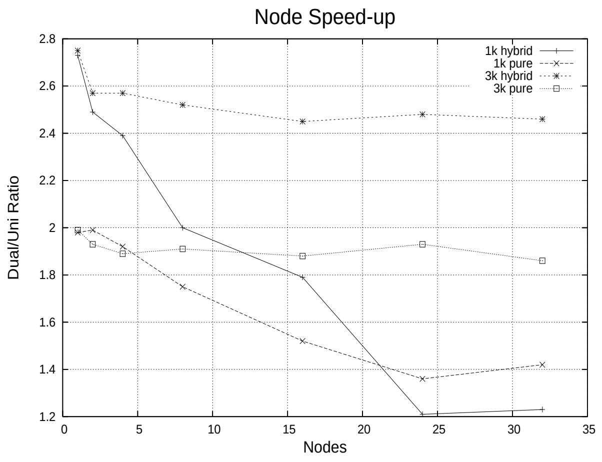


Figure 2. Performance of the parallel versions of the dynamic programming problem.

we obtain very good speed-ups, the results are not so good in the case of the matrix-vector product. In the first case both parallel models take profit of the dual node clearly increasing the speed-ups of the algorithm executed on one processor of each node. However, in the case of the matrix-vector product the results using both processors per node are better than the results using one processor per node only in some cases. Finally, the programming model that obtains the best results depend in both applications on the size of the problem.

References

[1] Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks. In *Supercomputing'00*, 2000.

[2] Franck Cappello, Olivier Richard, and Daniel Etiemble. Investigating the performance of two programming models for clusters of SMP PCs. In *6th IEEE Symp. on High-Performance Computer Architecture*, pages 349–359, 2000.

[3] N. Drosinos and N. Koziris. Performance comparison of pure MPI vs hybrid MPI-OpenMP parallelization models on SMP clusters. In *18th Int. Parallel & Distributed Symposium*, pages 15 (CD–ROM), 2004.

[4] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, UK, 1986.

[5] 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.

[6] 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.

[7] MPI Forum web page. Available at *http://www.mpi-forum.org*.

[8] OpenMP web page. Available at *http://www.openmp.org*.

[9] R. Rabenseifner. Hybrid parallel programming: Performance problems and chances. In *45th CUG Conference*, pages 12–16, 2003.

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

[11] Lorna Smith and Mark Bull. Development of mixed mode MPI / OpenMP applications. *Scientific Programming*, 9(2-3/2001):83–98, 2001. Presented at Workshop on OpenMP Applications and Tools (WOMPAT 2000), San Diego, Calif., July 6-7, 2000.