Array Redistribution in ScaLAPACK using PVM

Jack Dongarra *
The University of Tennessee and Oak Ridge National Laboratory

Loic Prylli, Cyril Randriamaro and Bernard Tourancheau†
LIP, Ecole Normale Suprieure de Lyon, France

Abstract

Linear algebra on distributed-memory parallel computers raises the problem of data distribution of matrices and vectors among the processes. Block-cyclic distribution works well for most algorithms. The block size must be chosen carefully, however, in order to achieve good efficiency and good load balancing. This choice depends heavily on each operation; hence, it is essential to be able to go from one distribution to another very quickly. We present here the algorithms implemented in the ScaLAPACK library, and we discuss timing results on a network of workstations and on a Cray T3D using PVM.

1 Introduction

The problem of data redistribution occurs when one deals with arrays (from vectors to multidimensional arrays) on parallel distributed-memory computers. Data redistribution applies both to data-parallel languages such as High Performance Fortran (HPF) and to single-process multiple-data (SPMD) programs with message passing. In the first case the redistribution is implicit in array statements such as $A = B$, where A and B are two matrices with different distributions. In the second case two approaches are possible. A library function may be called to do the operation, or the redistribution may be hidden (for instance, when a routine is called to do an LU decomposition, the routine can begin by checking whether the matrix passed has an optimal distribution; if not, the routine can do a redistribution and then go back to the initial distribution).

For a long time, redistribution was considered very difficult. Most implementations restricted the possible distributions to block or cyclic distributions [**?**, **?**, **?**, **?**, **?**]. In some implementations, all block sizes had to be multiple of each other, in order to reduce the memory access operations. Recent studies show that data redistribution can be done at compile time [**?**, **?**]. However, all of these studies address the compilation techniques for redistribution of arrays with a fixed number of processes.

*Computer Science Department, Knoxville, TN 37996-1301, USA and also Oak Ridge National Laboratory, Mathematical Sciences Section, P.O. Box 2008, Oak Ridge, TN 37831-6367, USA.

†46, alle d'Italie, 69364 LYON cedex 07, France (e-mail lprylli@lip.ens-lyon.fr)

We present here an algorithm and implementation of the redistribution routine that is used in ScaLAPACK [**?**, **?**]. We use a dynamic approach in order to construct the communication sets and then efficiently communicate them. Our algorithm uses several strategies depending of the amount of data to be communicated and the target architecture capabilities. The result is a very fast, scalable algorithm capable of loading and downloading from/to one process to/from many others.

The structure of the paper is as follow. Section 2 introduces the ScaLAPACK data distribution models and notation used in this paper. Section 3 presents the algorithms that were used for the redistribution of data. Section 4 discusses timing results obtained on different machines running PVM.

2 SCALAPACK data distribution and redistribution

The SCALAPACK library uses the block-cyclic data distribution on a virtual grid of processors in order to reach good load-balance, good computation efficiency on arrays and an equal memory usage between processors. Arrays are wrapped by block in all dimensions corresponding to the processor grid. The figure 1 illustrate the organization of the block-cyclic distribution of a 2D arrays on a 2D grid of P processors.

The distribution of a matrix is defined by four main parameters: a block width size, r; a block height size, s; the number of processor in a row, P_{row}; the number of processors in a column, P_{col} and few others to determine, when a sub-matrix is used, which element of the global matrix is the the starting point and which processor it belongs to.

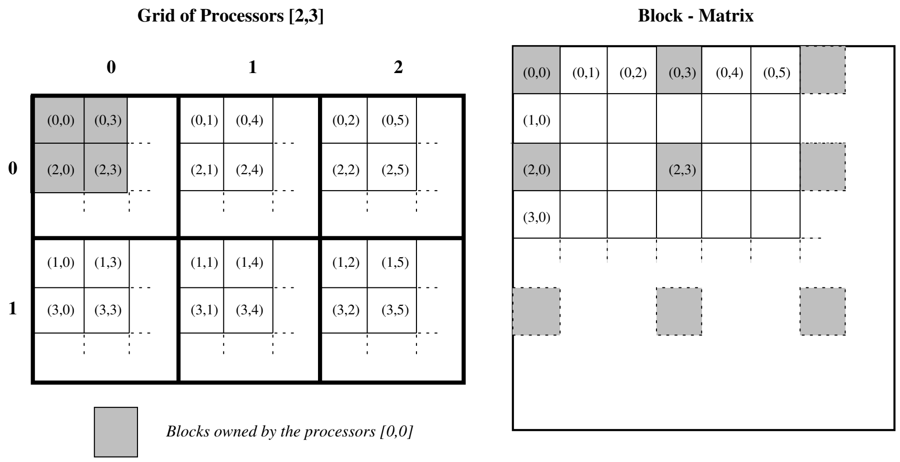

Figure 1: The block cyclic data distribution of a 2D array on a 2×3 grid of processors.

In SCALAPACK, the efficiency of redistribution is crucial as in any data parallel approach because it should be negligible or at least small compared to the computation it was done for. This is especially difficult since the redistribution operation has to be done dynamically, with no compile-time or static information. This dynamic approach implies that we deal from the beginning with the most general case of redistribution allowed by our constraints, namely cyclic with blocks of size (r, s) on a $P_{row} \times P_{col}$ virtual grid to cyclic with blocks of size (r', s') on a $P^{'}_{row} \times P^{'}_{col}$ virtual grid[1].

Moreover, no latency hiding techniques or overlapping can be used between the redistribution and the previous computation because these routines are independent (remark that it does not prevent the use of these techniques inside the redistribution routine itself, as it is explained in section 3.3).

3 Redistribution algorithm

The whole problem of data redistribution is for each processor to find which data stored locally has to be send to the others and respectively how much data it will receive from the others and where it will store it locally. Then the communication problem itself occurs on the target computer.

3.1 Computation of data sets in one dimension

If we assume that the data are stored contiguously in a block cyclic fashion on the processors, the problem is then to find which data items stored on processor p_i will be send to processor p_j. These data items have to be packed in one message before being sent to p_j in order to avoid start-up delays.

Our algorithm scan at the same time the matrix indices of the data blocks stored on p_i and those that will be stored on p_j. More precisely, we keep two counters, one corresponding to p_i's data location in the global matrix and the other to p_j's one. We increment them progressively by block as in a merge sort in order to determine the overlap areas (the comparison number is linear in the number of blocks). Then we pack the data items corresponding to the overlap areas in one message to be send to p_j.

3.2 General algorithm for the computation of data sets

The block scanning is done dimension by dimension and the overlapping indexes are the Cartesian product of the intervals computed in each dimension. (There is no limitation in the number of dimension scanned and the complexity is linear in the sum of the dimension sizes while the packing is obviously linear in the size of the data).

This work is done in each processor in order to send the data and respectively to receive the data, and stored them at the right place in local memory.

3.3 Optimizations

The obvious scanning strategy involves testing every index, but the intersection of intervals in a block cyclic distribution is in fact periodic. Hence, instead of a full block scanning, the algorithm stops as soon as it reaches the cyclic bound. Moreover, it reduces the bound on the storage necessary for the intersection patterns. In practice this optimization is interesting only for very small block sizes, and there is a point where it is more interesting to do an analytic determination of the cyclic distribution patterns (see, e.g., [**?**]).

[1]Remark that this general case includes the loading and down-loading of data from a processor to a multicomputer and also calls to parallel routines from a sequential code.

Synchronous communication : The algorithm uses the nonblocking send protocol. Therefore, in order to minimize process idle time and to avoid deadlocks resulting from buffer limitations, every send function call must have the corresponding receive function call *at the same time*. In other words, if the receiving process performs its n-th communication, the sending process performs its n-th communication, too.

The strategy implemented here can be compared with a rolling caterpillar composed of processes: at step d, each process p_i, $(0 \leq i < P)$ exchanges its data with process $p_{((P-i-d) mod P)}$. Figure 2 illustrates this method.

```
                    Step 1      step 2      step 3

                    0 1 2 3      0 1 2      7 0 1 2
8 processors                   7       3                 ....
                    7 6 5 4      6 5 4      6 5 4 3


                    0 1 2         0 1 2     6 0 1
7 processors              3     6                 2      ....
                    6 5 4         5 4 3     5 4 3
```

Figure 2: The caterpillar communication method is illustrated with an even (8) and an odd (7) number of processes. The communication occurs between the vertical pairs of processes (a process alone communicates with itself). .

Asynchronous communication : The asynchronous communication method is simpler: there is no supposition about the target computer's ability to receive from any process. Hence, the sizes of the messages to be received are computed first. Then the asynchronous receives are posted, followed by the sends.

Communication pipelining : In the pipeline method, we take advantage of the possibility that work can be divided in small units. Each process p_i can receive elements in a small packet. It can also at the same time pack elements to be sent and unpack elements it just received, instead of waiting for all the information from another process.

The pipelined method is an overlapping strategy similar to the work described in [**?**]. In the overlapped algorithm the information is scheduled with the caterpillar method, and we divide this protocol in several steps (each rotation of the caterpillar). For each step there are two asynchronous sends (from a node p_i to its corresponding p_j and from p_j to p_i). With this overlap between communication and calculation the program timings are greatly improved on machines with rather slow communications (i.e., local area networks of workstations and "old" parallel computers).

4 Timing results

The experiments corroborate very well our expectation, the computing of the data sets is negligible compare to the communication and packing and the global routine execution time is very efficient.

4.1 On a LAN of workstation using PVM

The PVM machine was composed of 4 workstations and the tests were ran during a Saturday night (no fever). We generate random tests in a range that is reasonable for the target algorithm using a $N \times N$ matrix ($1 \leq r < \sqrt{\frac{N}{P_{row}}}$ and $1 \leq s < \sqrt{\frac{N}{P_{col}}}$), and compare them to the LU decomposition on the same matrix size in Figure 3.

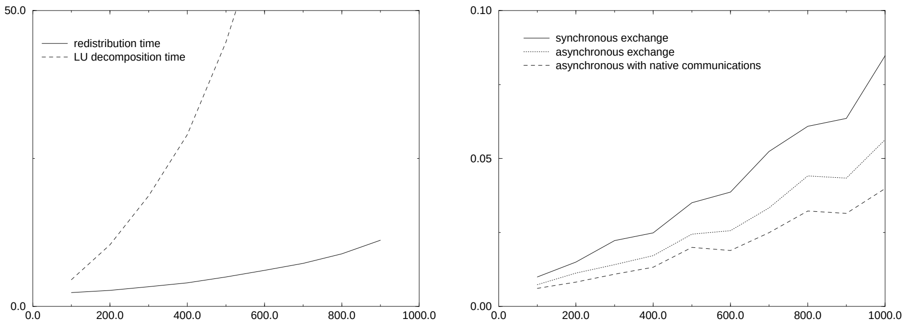

Figure 3: Timings of the redistribution tests in seconds as a function of the matrix size. On the left average of 20 random data-distributions in seconds compared to (the best) LU decomposition on a 4 nodes LAN of SUN Sparc-ELC. On the right on a 32 nodes Cray T3D.

The LU decomposition is far more costly than the redistributions, moreover, the worst timing of LU decomposition is twice the one plotted while there is no big differences between redistribution times.

4.2 On a Cray T3D parallel machine

The Cray T3D proposed a home-made version of PVM based on the Cray native primitives. We show on Figure 3 the two algorithms described before implemented using this Cray PVM version and the asynchronous algorithm (best one) implemented directly with the Cray native shared memory primitives.

The results show the very good communication performances achieved by this machine, especially with the shared memory communications.

The timings are very good indeed regarding to classical computation duration, for instance the benchmark of LU decomposition of a 1600×1600 matrix is 1.7 second on this machine[2].

5 Conclusion

In general, the redistribution of data is useful for improving the performance of parallel linear algebra routines. However, the redistribution must be very efficient. Our work demonstrates that the redistribution of data can be done efficiently using our algorithm and optimization. Our algorithm, implemented for the ScaLAPACK library, computes all redistributions and is not limited to a set of block-cyclic redistributions. It is also useful when dealing with submatrices (but cannot take into account strides that are not 1 or the array leading dimension).

Our results are encouraging for the frequent use of data redistribution both in explicit parallel programming and in redistribution library routines with calls in codes generated by HPF compilers.

[2](from the LINPACK benchmark database)

References