Towards best achievable floating point performance for linear algebra computations on COTS Beowulf clusters

S. Fourmanoit[1], R. Roy[1] & F. Bertrand[2]
[1] *Department of Computer Engineering*
[2] *Department of Chemical Engineering*
École Polytechnique de Montréal, Canada

Abstract

The solution of linear systems is required for many modern engineering applications from computational fluid dynamics to biomedical imaging. This paper shows how a low cost Beowulf cluster can be optimized to handle various CPU-intensive linear algebra computations using the scalable linear algebra package (ScaLAPACK). The paper focuses on the techniques and tools that we have developed to enhance the performance of a basic reference implementation. The tuning of system performance includes both partial recoding of some modules and careful engineering of clusters design, doubling speedup as compared to the reference implementation.

1 Introduction

In a lot of engineering applications such as CFD or biomedical imaging, the solution of large linear systems is required. Linear algebra automated handling has always been a computer-intensive activity. The improvements in coding techniques and processor speed have led many engineers to ask for increasingly larger system manipulations, and this demand currently exceeds each increase in computational power. This behaviour has been amplified by a recent trend in many engineering schools to teach computer-aided numerical differential equations and to use generic code solvers, thus broadening the users base for such methods.

Not so long ago, the solution of huge CPU-intensive linear systems were possible only at meteorological centers or similar facilities, where the computing power was often needed for critical analysis. Unfortunately, most organizations outside of government and academia could not access these resources. Alternatively, large corporations and universities could purchase shared-memory supercomputers such as those built by IBM or SGI. Those machines were however expensive, with unclear upgrade/recycling path and relatively high maintenance costs. Smaller companies and research institutions simply could not afford such proprietary systems. Thus, the solution of large linear systems (with typically $> 10^6$ unknowns) were often beyond the reach of many of the engineers and researchers whose applications could depend on it.

This paper discusses the handling of large linear algebra system using inexpensive *Beowulf* clusters [1]. Over the past few years, very positive claims have been made in favor of PC clusters, and great pieces of software such as ScaLAPACK were built to specifically drive such computations on virtually all distributed memory systems, including *Beowulfs*. However, on such very low cost clusters, out-of-the-box installation of ScaLAPACK will unlikely lead to satisfying through-puts, unless some significant tuning is made on both the software and the hardware sides. From a standard reference setting of ScaLAPACK over Fast-Ethernet, this paper will show how using specialized floating point instructions and a simple yet carefully layout network topology can lead to better, scalable overall performance for virtually all linear algebra manipulation routines enclosed in ScaLAPACK.

The paper is organized as follows. In section 2, we will describe the structure of ScaLAPACK and give relevant details for superimposing communications in the linear algebra solvers. Section 3 is devoted to a description of techniques and tools available for fine-tuning this specific library on clusters. Measures of performance are given for a test-bed facility composed of 16 Athlon nodes with a main server at the Centre de recherche en calcul appliqué (CERCA); communications are optimized by adding 4-node switches over a Cartesian topology. Finally, we draw some conclusions in section 4.

2 ScaLAPACK architecture

2.1 Overview

Rephrasing its authors [2], the Scalable Linear Algebra Package (ScaLAPACK) was written to provide various efficient linear algebra computations on a distributed memory computer: linear systems, least squares and eigenvalues routines operating on dense, narrow-band and tridiagonal matrices were proposed, using aggressive block segmenting algorithms to minimize communication whenever possible. As shown on Figure 1, the ScaLAPACK design is also very well modularized. In fact, one of the great strengths of ScaLAPACK is its ability to use, on local computational nodes, Level 2 and 3 BLAS (Basic Linear Algebra Subprograms for matrix-to-vector and matrix-to-matrix operations) for all the computations. On the other hand, a communication package, called BLASC (BLAS Communication Subprograms) is responsible for all the required inter-process communications, coordinating the overall operations with PBLAS (Parallel BLAS), an interface enabling ScaLAPACK routines to look roughly the same as their serial LAPACK counterparts.

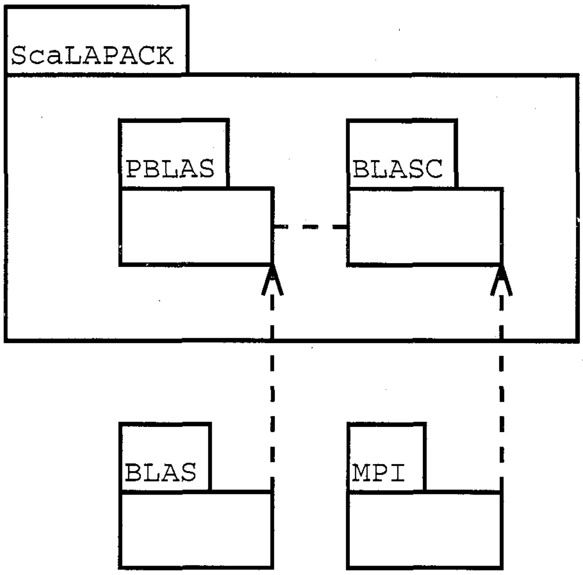

Figure 1: Overall ScaLAPACK packaging using MPI

2.2 The heart of ScaLAPACK: the two-dimensional block-cyclic distribution

With ScaLAPACK, machine-specific code for computation is entirely kept into BLAS, and so is machine-specific code for communications in BLASC. This nice component-based layout has been made possible by the data distribution scheme adopted in ScaLAPACK, called the two-dimensional block-cyclic distribution [2]. In a nutshell, this data mapping between computational processes looks a lot like shuffling a card deck. From a matrix $A = [a_{i,j}]_{m \times n}$, we create $P_r \times P_c$ process matrices noted $\tilde{P}_k = [p_{k_{i,j}}]_{q \times r}$, $k \in \{0, 1, \ldots, P_r P_c - 1\}$ such as:

$$q = \begin{cases} \lfloor m/P_r \rfloor B_r & , \quad k \not\equiv 0 \bmod \lceil m/B_r \rceil \\ \lfloor m/P_r \rfloor B_r + m \bmod B_r & , \quad k \equiv 0 \bmod \lceil m/B_r \rceil \end{cases} \tag{1}$$

$$r = \begin{cases} \lfloor n/P_c \rfloor B_c & , \quad \lceil k/P_c \rceil \neq P_r \\ \lfloor n/P_c \rfloor B_c + n \bmod B_c & , \quad \lceil k/P_c \rceil = P_r \end{cases} \tag{2}$$

and

$$p_{k_{i,j}} = a_{(k+P_r\lfloor i/B_r \rfloor)B_r + i \bmod B_r,\ (k+P_c\lfloor j/B_c \rfloor)B_c + j \bmod B_c}. \tag{3}$$

where B_r and B_c are respectively the block row and block column sizes, and all i and j indices start from zero. This scheme has great overall properties of load balancing, intra/extra process data alignment, allowing both Level 3 BLAS operations [3] and simplified communications [4] because - and this will be its essential property - the process matrices $\tilde{P}_k$ are in fact part of a larger 2D process "grid" (hence the P_r and P_c notation), mapping one-to-one content of matrix A on a matrix G defined as:

$$G = \left[\tilde{P}_k\right]_{P_r \times P_c} = \left[[p_{k_{i,j}}]_{k \bmod P_r, \lfloor k/P_r \rfloor}\right]_{m \times n} \tag{4}$$

where all data in a column (resp. row) of A are still present in the corresponding column (resp. row) of G, making the processes belonging to the same column or row of the $P_c \times P_r$ process grid the only neighbors potentially needing to communicate.

This distribution is the general mapping used with dense matrix. With sparse matrices (either tridiagonal or narrow-band) also handled by ScaLAPACK, other mappings (1D row or column block-cycling distributions) are used: they are special cases of this one when P_c or P_r equals one.

As an illustration, if we decide to distribute a 4×4 matrix on a 2×2 process grid using 1×1 block size, we would obtain the distribution shown on Figure 2.

$$\begin{array}{|c|c|c|c|} \hline a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ \hline a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ \hline a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ \hline a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \\ \hline \end{array} \longrightarrow \begin{array}{cc} \begin{array}{|c|c|} \hline a_{0,0} & a_{0,2} \\ \hline a_{2,0} & a_{2,2} \\ \hline \end{array} & \begin{array}{|c|c|} \hline a_{0,1} & a_{0,3} \\ \hline a_{2,1} & a_{2,3} \\ \hline \end{array} \\ \\ \begin{array}{|c|c|} \hline a_{1,0} & a_{1,2} \\ \hline a_{3,0} & a_{3,2} \\ \hline \end{array} & \begin{array}{|c|c|} \hline a_{1,1} & a_{1,3} \\ \hline a_{3,1} & a_{3,3} \\ \hline \end{array} \end{array}$$

Figure 2: Block-cyclic distribution on a 2×2 grid

3 Techniques and tools for fine-tuning ScaLAPACK on Beowulf clusters

From the previous section, we can assume that there are two main strategies for optimizing ScaLAPACK on COTS Beowulf clusters: replacing the floating point intensive BLAS or the node-local communication layer BLASC with new implementations.

In the next two subsections, we will look at both approaches separately. A short analysis of today's typical COTS hardware - over 1 GHz single or dual x86 computers (Intel or AMD) with a fair amount of RAM on Fast-Ethernet - quickly shows where optimization will be the most significant. The 1.4 GHz AMD Athlon Thunderbird core, for instance, can easily sustain almost one floating point operation per clock cycle on any reasonably dense computation code in both 32 bits and 80/64 bits mode without any specific optimization (leading to a 1.4 GFlop sustained performance). On the other hand, Fast-Ethernet, with a peak at 12.5 MB/s throughput, can only transmit a maximum of 3 millions of 32-bit results in duplex mode (neglecting any communication overhead). Communications are therefore absolutely assured to be the bottleneck. Performance tests done on our 16-node AMD Athlon cluster with Fast-Ethernet (Intel EtherExpressPro 100, using MPICH 1.1.0 as the BLASC underlying message-passing library) show how well the BLASC and Level 3 BLAS perform on this architecture. Table 1 shows the flop rate achieved by matrix multiplication BLAS routine: routine SGEMM/DGEMM (F_{MM}) on a node versus the theoretical peak performance of that node, and the approximated values of the latency (t_m) and bandwidth ($1/t_\mu$) achieved by the BLASC versus the underlying message-passing software for this machine. The values for latency were obtained by timing the cost of a 0-length message. The saturation was obtained by increasing message length until bandwidth is saturated. We used the same timing mechanism for both the BLASC and the underlying message-passing library.

Table 1: Reference single node performance of BLAS and BLASC implementation on test-bed cluster

MFlops		$t_m (\mu s)$		$1/t_\mu$ (MB/s)	
F_{MM}	Peak	BLASC	Native	BLASC	Native
1120	1400	170	130	10.8	11.0

3.1 Optimizing Level 2 and Level 3 BLAS

Optimizing BLAS is a standard job for "sequential" programmers. Nowadays, thanks to both manufacturers and open source developers, architecture-optimized implementation of BLAS were written for almost all processors commonly used in COTS Beowulf clusters, such as Intel[5] and AMD[6]. Automated optimizers, such as ATLAS[7] can also be used to minimize memory penalty involved in cache hierarchy data transfer.

3.1.1 Performance

Generally speaking, optimization is based on a better use of new floating point instruction sets built into the processors (MMX, 3DNOW!, SSE, Altivec, etc.) to implement some level of simple vectorization. In addition to the reference Fortran 77 implementation, we have also tested Intel's math kernel[5] and we reproduced the work of the aggregate consortium[6] concerning the AMD Athlon processor. Table 2 shows the flop rate achieved by the matrix-matrix multiply Level 3 BLAS routine SGEMM/DGEMM (F_{MM}) on a node versus its theoretical peak performance.

Table 2: Performance of various BLAS implementations on the 16-node test-bed cluster

	MFlops	
	F_{MM}	Peak
Reference Implementation	1120	1400
Intel for Pentium	1283	1400
Intel for Pentium Pro	1612	1400
Aggregate SWAR Implementation	2694	1400

Since Athlon peak performance was only an estimation, it is not surprising to see that it can be exceeded. On the standard i386 instruction set, there is no way to perform more than a floating point operation per clock cycle: this is no longer the case with Athlon extended instruction set, which potentially performs up to three floating point operations per cycle. Once again, it must be pointed out that this speed-up cannot be achieved on the Beowulf cluster in parallel mode because of the relatively slow bandwidth, at least for affordable port-to-port interconnections.

3.2 Optimizing BLASC Communication Subprograms

The second alternative for optimization is communications, on both the hardware and the software sides. Our previous analysis of the two-dimensional block-cyclic distribution used extensively through ScaLAPACK has lead us to physically build the versatile and inexpensive 2D Cartesian topology shown in Figure 3.

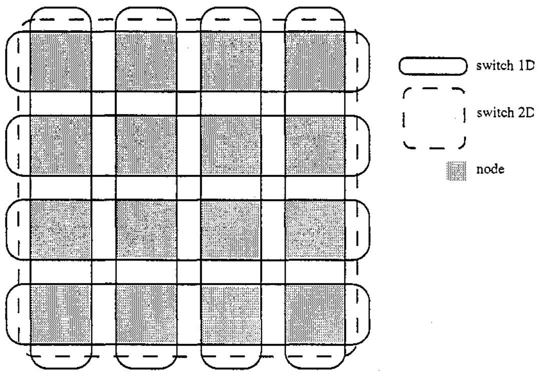

Figure 3: Schematic interconnect for a 16-node Cartesian topology

On this figure, each computational node (here, we assumed one process per node to simplify) is connected to the others nodes through three interfaces: a global one giving access to every other node (via a switch labeled 2D) and two "local" ones giving only access to row and column neighbors (switch 1D). There is an order of magnitude between these local 1D switches and the global 2D one. We must insist that this setting is quite affordable, requiring only the interconnection supplementary material described in Table 3.

Table 3: Networking material cost for a 16-node COST Beowulf cluster optimized with a Cartesian grid topology

Quantity	Description	Unitary Cost (USD)	Total Cost
48	Fast Ethernet Card (DEC Tulip)	40	1920
8	Full Duplex 4-port switch	50	400
1	Full Duplex 16-port switch	1500	1500
		Total:	< 4000 $

Once installed, this topology is nothing but a physical realization of BLASC conceptual connection patterns for all of the standard process grids it could use for 16 processes or less: 1×2, 1×4, 2×2, 2×4, 4×4, 1×8, 1×16 and of course their transposes. Moreover, every pair of neighbor processes has redundant interconnecting paths, thus providing quicker and more reliable connections. Local pairing using 4-port interconnects for an entire row or column (what is called a *scope* in BLASC terminology) and direct message broadcasting are possible since every message is encapsulated into its own BLASC context (closely related to MPI communicators), preventing overlapping message universes from colliding.

Regarding software adaptation, two different approaches were used, giving similar results in terms of performance speed-up: building a new abstract device into MPI or using Linux Kernel 2.4 routing and QoS policies.

3.2.1 `newp4` : a routing ADI interface

This approach, totally internal to MPICH 1.1, is based upon the Abstract Device (ADI-2) to implement a small routing policy system on top of a `p4` device. In addition to the usual `MPI_COMM_WORLD` communicator, two other communicators are created during initialization virtually connecting row and column neighbors. Since real broadcast calls used by BLASC (`MPI_Bcast`) were not implemented, a bufferization was internally made via the ADI and data were flushed only when buffer was full or when IP addresses belonging to different switches were asked for. With this approach, each process only sees one device that transparently sends information where it belongs, improving network usage.

3.2.2 Linux Kernel 2.4 routine and QoS policies

This other approach is totally external to ScaLAPACK or its dependencies, and is also a lot quicker to install. The whole idea is that you do not need to tell MPI/BLASC anything about interfaces as long as packets getting out of a node can be correctly re-routed. Basically, we use IP-tables for intercepting all out-bound packets to the cluster-wide interface and re-sending them, whenever possible, to another interface compatible with the source and destination addresses. A similar procedure is used for incoming packets.

3.2.3 Performance

Setting efficient data distribution for specific algorithms (by reshaping the $P_c \times P_r$ dimensions of the process grid) is the key to real performance enhancements in ScaLAPACK. In fact, the algorithms currently implemented in ScaLAPACK fall into two main categories.

In the first category, a block of rows or columns is replicated in all process rows or columns, in such a way that the sources of successive broadcasts are themselves owner of orderly fashioned blocks of data contained in original matrices. The QR factorization and the right looking variant of the LU factorization are typical examples of such algorithms.Therefore, the LU, QR, and QL factorizations perform better for "flat" process grids ($P_c \gg P_r$). These factorizations perform a reduction operation for each matrix column (pivoting in the LU factorization for instance). After this reduction has been performed, it is important to update the next block of columns as fast as possible. This update is done by broadcasting this block, making our Cartesian topology especially useful, since columns and rows processes are always physically connected to direct links suitable for broadcast.

The second group of algorithms is characterized by the physical transposition of a group of rows and/or columns at each step, since it minimizes average data communication. Square or near square grids are more adequate from a performance point of view for these transposition operations. Examples of such algorithms implemented in ScaLAPACK include the right-looking variant of the Cholesky factorization or the matrix inversion algorithm. Our topology will also handle it.

As in [2], table 4 illustrates the speed of the ScaLAPACK driver routine PSGESV for solving a square linear system of order N by LU factorization with partial row pivoting of a real matrix. For all timings, native 32-bit floating-point arithmetics was used, on Aggregate SWAR BLAS Implementation. Similarly, table 5 shows results for a matrix-vector product $y \leftarrow y + Ax$, where A is a square matrix of order N and x and y are vectors that are both distributed over a process column.

Table 4: Speed in MFlops of PSGESV for $N \times N$ matrices

Implementation	Process Grid	Block Size	Values of N			
			2000	5000	7500	10000
Reference	1×4	64	213	350	380	
	2×4	64	191	500	625	635
Cartesian 2D	1×4	64	404	611	627	
	2×4	64	415	716	976	1295

Table 5: Speed in MFlops of PSGEMV for $N \times N$ matrices

Implementation	Process Grid	Block Size	Values of N			
			2000	5000	7500	10000
Reference	4×4	64	275	332	345	352
Cartesian 2D	4×4	64	384	600	851	1021

4 Conclusion

The performance of the scalable linear algebra package ScaLAPACK was shown to be substantially enhanced after superimposing a low-cost Cartesian network topology on a COTS cluster. This is done by implementing on the hardware level what ScaLAPACK implicitly suggests from its software design. Moreover, the hardware needed to substantially speedup this package is affordable compared to the overall cluster cost. Communications were managed either by defining a new abstract device, or by using operating system's routing interfaces. These software tuning can be devised to be relatively transparent to the end user, and also quite straightforward to make.

The benefit of this approach is that the scientific community can significantly increase their ability to solve large linear systems on COTS Beowulf clusters for only a small added fee as much in term of money than labor.

Acknowledgments – This work has been carried out partly with the help of grants from the Natural Science and Engineering Research Council of Canada. Funding was also provided by the HPC program at Centre de recherche en calcul appliqué (CERCA).

References

[1] Becker, D.J, Sterling, T., Savarese, D., Dorband, J.E., Ranawak, U.A., & Parker, C.V., "Beowulf: a parallel workstation for scientific computation," *Proc. International Conference on parallel processing,* 1995.

[2] Blackford, L.S., Choi, J., Cleary, A., D'Azevedo, E. Demmel, J., Dhillon, I, Dongarra, J, Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., & Whaley, R.C, *Scalapack Users' Guide,* Society for Industrial and Applied Mathematics: Philadelphia, 1997.

[3] Dongarra, J., Van de Geijn, R., & Walker, D. "Scalability issues in the design of a library for dense linear algebra," *Journal of Parallel and Distributed Computing,* **22**, pp. 523-537, 1994.

[4] Hendrickson, B., & Womble,D., "The torus-wrap mapping for dense matrix calculations on massively parallel computers," *Society for Industrial and Applied Mathematics Journal or Scientific and Statistic Computing*, **15**, pp. 1201-1226, 1994.

[5] Intel Corporation, *Intel Math Kernel Library, Reference Manual*, United States of America, 2001.

[6] Fisher, R.J., & Dietz, H.G., Compiling for SIMD Within a Register, *The Workshop on Programming Languages and Compilers for Parallel Computing (LCPC)*, 1998.

[7] Whaley, R.C, Dongarra, J.,Petitet, A., "Automated Empirical Optimization of Software and the ATLAS Project," *Parallel Computing,* **27**, pp 3-25, 2001.