ParLSS: a Parallel Self-Verified Solver for Dense Systems of Linear Equations

Alexandre Almeida
Universidade Federal do Rio Grande do Sul
Instituto de Informática
Campus do Vale – Av. Bento Gonçalves, 9500
Porto Alegre – RS – Brasil
avalmeida@inf.ufrgs.br

Carlos Amaral Hölbig
Universidade de Passo Fundo
Curso de Ciência da Computação
Campus 1 – Bairro São José – CEP 91501-970
Passo Fundo – RS – Brasil
holbig@upf.br

Abstract

The LSS solver is a self-verified numerical application that computes a solution for dense systems of linear equations of the form $Ax = b$. This work presents ParLSS: a new parallel numerical stable version of the LSS solver that keeps the same accuracy delivered by the original sequential LSS. Also, this work introduces a new set of MPI collective communication functions for some data types of the C-XSC library. We show that these functions played a key role in optimizing communication overhead among the parallel application processes.

1. Introduction

According to [2], the solution of systems of linear equations $Ax = b$, where A is a dense coefficient matrix, has a vast application in many areas like physics and engineering. However, when the task of solving this kind of system is attributed to computers, both the numerical accuracy of results and the application processing time may become issues that, if not addressed appropriately, tend to make the application unreliable and unfeasible. Such a conclusion can be inferred, for example, by the experimental results presented in [9].

The numerical instability is a problem inherent to all digital machines, mainly because values in the set of real numbers have to be stored in registers limited to a mantissa length. This limitation causes round-off errors on most of real numbers, besides making arithmetic operation results susceptible to cancellation of leading digits [6]. In this scenario, such issues may lead a numerical application to return incorrect results.

The Verified Computing is a paradigm that can be adopted to treat and avoid the machine numerical instability, making possible the computation of accurate results. Using this programming paradigm, the application itself becomes capable of deciding whether to execute an alternative algorithm, or even the same procedure again but using a higher precision. Both of these situations may occur if the application detects that the numerical results are not sufficiently accurate for the problem being solved, that is, within an acceptable error range [6]. However, the adoption of this programming model may bring a considerable performance penalty to the application, as can be seen in [9].

The LSS (Linear System Solver) is a numerical application written in C++ language that solves dense systems of linear equations. Through the C-XSC library [4], which enables the development of algorithms that deliver highly accurate results, the LSS solver implements the Verified Computing so that it is able to assure the correctness of its results. This work presents ParLSS: a parallel version of the LSS solver for distributed memory machines, implemented using the MPI library. What motivates our work is the low performance of the sequential version, due to both the adoption of Verified Computing and the C-XSC library. Also, this work presents the running time of an optimized sequential version for the LSS solver. This optimized version was used as a base to develop the parallel implementation.

Besides the parallel version, MPI collective communication functions for some of the C-XSC data types are presented. These functions were developed to take advantage of the data distribution pattern adopted in the parallel version of the solver, and also to avoid the use of point-to-point communication.

This paper is organized as follows. Section 2 presents an overview of the LSS solver, describing the optimized sequential solver and its processing time results; section 3 introduces the ParLSS solver, and presents an evaluation of performance; our conclusions and future work are presented in section 4.

2. LSS Solver Overview

The LSS solver is made of two main algorithms: a single precision and a double precision algorithm. The double precision procedure is executed only when the single precision cannot determine a sufficiently accurate approximation for the linear system being solved. This behavior happens mostly for ill-conditioned systems [5].

When a linear system $Ax = b$ is given as input to the solver, the single precision algorithm determines an approximate inverse A^{-1} of the dense coefficient matrix A, using the Gauss-Jordan method. After the inverse matrix is obtained, the solver calculates an approximate result x' for the linear system through the equation $x' = A^{-1} \times b$.

Then, the result x' obtained is refined by a series of steps that improve the approximate result by correcting it with residual error values. These residuals are calculated through the high precision accumulators provided by C-XSC library (`dotprecision` data types). When the single precision procedure finishes improving x', the solver verifies if the solution was obtained. If this verification step fails, then the double precision procedure is executed. Otherwise, the execution ends.

The double precision algorithm recomputes the approximate inverse A^{-1} in higher precision using a method called *Rump's device* [6], obtaining an improved inverse Ai^{-1}. In addition, a residual error matrix E of Ai^{-1} is computed, and the steps of the single precision procedure are re-executed, but this time considering both Ai^{-1} and E.

More details about the LSS solver algorithm, as well as its theoretical background can be found in [5] and [6].

2.1. Optimized LSS solver

Before the development of the parallel version, the original LSS solver was optimized by the elimination of redundant steps inside the single and double precision algorithms. Furthermore, the Gauss-Jordan method, implemented in the original solver using the C-XSC data types, was re-written in pure C++ language in order to improve its performance by removing the C-XSC library overhead. However, to keep the numerical stability of the Gauss-Jordan method, only those points of the code that performed calculations (e.g., divisions) where kept under the C-XSC control, since these are the only "hot spots" in the code subject to introduce numerical instability on the method. In addition, the routine `cblas_dswap` of ATLAS library [8] was used to perform the row exchange operation of the method.

The execution times of both the original and optimized sequential versions, as well as the performance gain (speed-up), are shown in Table 1 as a function of the linear system size. Also, all the time values are averages on 10 executions. The test platform used was a desktop computer with a 2.2 GHz Intel Core 2 Duo E4500, 2 GB of main memory, 2 MB of cache and running Ubuntu Linux 7.04 (kernel 2.6.21).

System size	Original (s)	Optimized (s)	Speed-up
512	388.260	224.565	1.729
1024	3353.750	1873.023	1.791
2048	27342.00	15084.160	1.813

Table 1. Processing time of original and optimized LSS

These time results were obtained by processing systems with coefficient matrices initialized as an ill-conditioned Hilbert [5] square matrix with entries $H_{i,j} = \frac{1}{i+j-1}$. Using this kind of matrix, we were able to "force" the solver to execute both the single and double precision algorithms, so that the full extent of the LSS code were reached.

Even though the optimized version showed a lower execution time, it still can be considered prohibitive when large scale instances are processed (e.g., about 4 hours for 2048 system size). In the next section, it is presented ParLSS, which is our parallel approach based on the optimized sequential LSS.

Still, as a related work on parallel self-verified solvers, the FastILSS solver [9] uses an alternative method based on a multiple precision algorithm to achieve quality of results, avoiding the high precision accumulators of C-XSC. In such parallel approach, the user is able to set a desired precision for the numerical results in order to improve the processing time, causing a possible loss of accuracy. The main difference of our parallel solver in relation to FastILSS, is that our approach keeps the original numerical quality delivered by the sequential LSS solver.

3. ParLSS

This section presents some details and a performance evaluation of ParLSS solver.

The low performance of the sequential solver is caused by a total of four steps inside the single and double precision algorithms, as we show by experimental results in [1]. Therefore, such steps are chosen to be processed in parallel, in order to decrease the general response time of the solver.

In the single precision procedure, there are two major steps that slowdown the execution of the solver. The Gauss-Jordan method, which determines the inverse of the coefficient matrix, and the last step of the single precision, which performs a matrix–matrix product and a matrix–matrix sub-

traction. Similarly, in the double precision step three major steps could be identified. The Gauss-Jordan method once more, which is used to determine the result of the *Rump's device* procedure, and two other steps that perform each a matrix-matrix product.

Even though the Gauss-Jordan method represents one of the bottlenecks of time, its parallelization was postponed to a future version of ParLSS, since it is necessary to investigate other methods that calculate the inverse matrix more efficiently. Also, the algorithm of such a method must provide conditions to exploit data parallelism. Furthermore, parallel libraries like ScaLAPACK could be considered. However, these libraries do not assure numerical quality as the C-XSC library does, i.e., the adoption of ScaLAPACK could make the solver numerically unstable.

In relation to the ParLSS data partitioning scheme, the block-striped partitioning [7] was adopted, where each process is assigned one block of complete rows of a given matrix. The next subsection presents MPI functions for collective data communication that support C-XSC data types.

3.1. MPI functions for C-XSC

In order to perform both the data partitioning scheme and data communication among processes, a collective communication model is appropriate in terms of efficiency. Therefore, the functions MPI_Bcast, MPI_Scatter, MPI_Gather and MPI_Allgather were re-implemented to support the C-XSC library data types *rmatrix* and *imatrix* used in the solver.

This set of collective operation functions can be seen as an extension to the point-to-point communication functions provided by [3]. Next, it is presented the performance evaluation of ParLSS, and also the communication time growth of MPI_Bcast in comparison with a point-to-point communication model.

3.2. Parallel test platform

The ParLSS was evaluated on a 32-node Network of Workstations. Each node is a dual-core 2.2 GHz Intel Core 2 Duo E4500, 2 MB of cache, 2 GB of main memory (two dual-channel modules of 1 GB each) and running Ubuntu Linux 7.04 (kernel 2.6.21). The interconnection network is a conventional fast-ethernet (100 Mbps). The MPI distribution used was LAM/MPI 7.1.2 with gcc 4.1.

3.3. Performance results

The performance of ParLSS was measured considering linear systems of order 64, 256, 1024 and 2048, using 2, 4, 8, 16, 32 and 64 MPI processes, where each node of the cluster was assigned two processes in order to take advantage of the dual-core processors. Also, like for the sequential version performance tests, the coefficient matrix of each test case was initialized as a Hilbert matrix.

The processing time was obtained for the ParLSS as a whole, and for the three parallel steps developed. All the results were determined by the average of time of 10 executions, since the standard deviation of the time samples were within a small range (less than 2%). Figure 1 shows the speed-up of the most processing intensive step on the solver, where a super linear speed-up could be observed for some cases. Because this parallel step is the last of the solver (consequently the last of the double precision algorithm), most of data was already distributed by previous parallel steps so that this step does not need to spend time waiting for data input, what makes it even more cpu-bound.

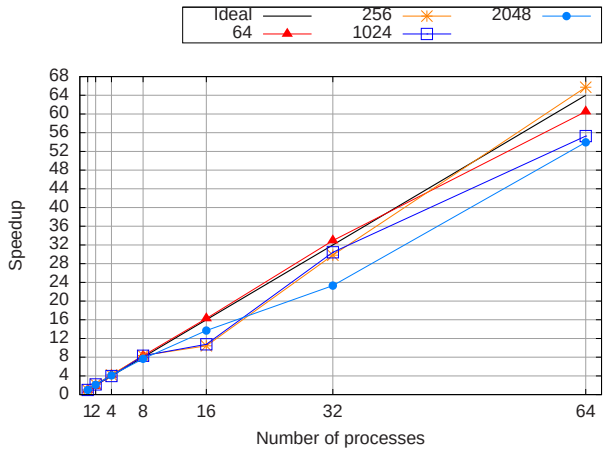

Figure 1. Speed-up for the last parallel step (double precision)

Even though the speed-up of this parallel step was almost ideal, and the speed-up obtained by the other parallel steps were also reasonable, the ParLSS solver speed-up as a general, presented in Figure 2, shows a low efficiency due to the sequential Gauss-Jordan method. This kind of speed-up was expected because, as previously stated, the inverse matrix computation is one of the bottlenecks of time in the solver algorithm.

For the evaluation of the collective communication functions, it was performed a replication of a square matrix (C-XSC *rmatrix* data type) of size 2048 among 2, 4, 8, 16, 32 and 64 MPI processes. The objective of this experiment was to compare the communication overhead of a point-to-point data transfer model, using the functions MPI_Send/Recv provided by [3], and a collective commu-

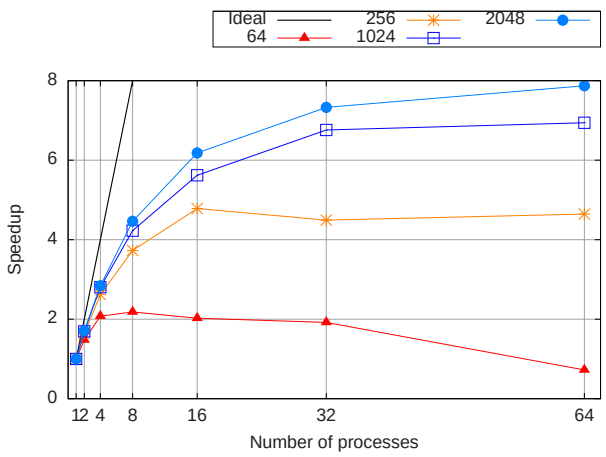

Figure 2. ParLSS solver speed-up

nication model, using the MPI_Bcast function developed in this work.

Figure 3 presents the growth of the communication overhead as the number of processes increase. It can be seen that the point-to-point MPI_Send/Recv tends to prohibitive values of time in a faster rate than MPI_Bcast.

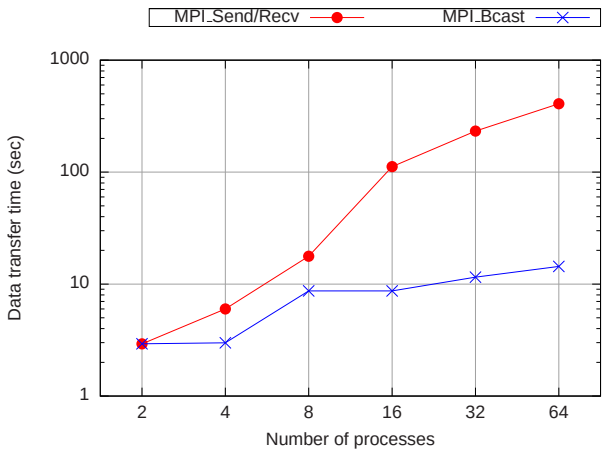

Figure 3. Growth of the communication overhead

4. Conclusions and Future Work

In this paper, we presented a new optimized sequential version for the Linear System Solver as well as ParLSS, which is a parallel approach for distributed memory machines.

The time results for the optimized sequential version showed a considerable improvement over the original version, only by removing redundant calculations and by "reducing" the use of C-XSC data types in the Gauss-Jordan method.

The parallel version showed a low speed-up, mainly because the computation of the inverse matrix is still executed sequentially. However, the parallel steps showed a reasonable speed-up, so adapting a parallel numerical stable procedure to determine the inverse matrix would certainly improve ParLSS efficiency.

In relation to the use of C-XSC library in parallel environments, as can be seen in the experimental results presented, the communication cost that the MPI_Send/Recv functions would bring to ParLSS would be high. So, we were able to avoid such overhead by using the collective communication functions developed, yet extending the set of peer-to-peer MPI functions of [3].

As a future work, we will investigate alternative methods to compute the inverse matrix in parallel, possibly using the C-XSC library to guarantee the correctness of numerical results.

References

[1] A. Almeida and C. Hölbig. Análise de Desempenho do Solver Verificado LSS. In *8a. Escola Regional de Alto Desempenho*, 2008. Anais.

[2] D. M. Cláudio and J. M. Marins. *Cálculo Numérico Computacional: Teoria e Prática*. Editora Atlas, 2000.

[3] M. Grimmer. Extending the range of c-xsc: Some tools and applications for the use in parallel and other environments. In *Numerical Validation in Current Hardware Architectures*, Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. IBFI, Schloss Dagstuhl, Germany.

[4] W. Hofschuster and W. Krämer. C-xsc 2.0 – a c++ library for extended scientific computing. *Lecture Notes in Computer Science*, 2004.

[5] C. Hölbig and W. Krämer. Selfverifying solvers for dense systems of linear equations realized in c-xsc. Technical report, Bergische Universität Wuppertal, 2003.

[6] C. A. Hölbig. *Ambiente de Alto Desempenho com Alta Exatidão para a Resolução de Problemas*. PhD thesis, Universidade Federal do Rio Grande do Sul, 2005.

[7] V. Kumar et al. *Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms*. Benjamin-Cummings Pub Co, 1994.

[8] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999. CD-ROM Proceedings.

[9] M. Zimmer and W. Krämer. Fast (parallel) dense linear interval systems solvers in c-xsc using error free transformations and blas. In *Numerical Validation in Current Hardware Architectures*, Dagstuhl, Germany, 2008. IBFI, Schloss Dagstuhl, Germany.