On the Effect of Linear Algebra Implementations in Real-Time Multibody System Dynamics

Manuel González, Francisco González, Daniel Dopico, Alberto Luaces

This is a post-peer-review, pre-copyedit version of an article published in Computational Mechanics. This version of the article has been accepted for publication, after peer review and is subject to Springer Nature's AM terms of use, but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: https://doi.org/10.1007/s00466-007-0218-2.

Submitted to Computational Mechanics on July 11, 2007

On the effect of linear algebra implementations in real-time multibody system dynamics

Manuel González*, Francisco González*, Daniel Dopico*, Alberto Luaces*

**Escuela Politécnica Superior*
Universidad de A Coruña, Mendizábal s/n, 15403 Ferrol, Spain
e-mail: lolo@cdf.udc.es, fgonzalez@udc.es, ddopico@udc.es, aluaces@udc.es
web page: http://lim.ii.udc.es

Abstract. This paper compares the efficiency of multibody system (MBS) dynamic simulation codes that rely on different implementations of linear algebra operations. The dynamics of an N-loop four-bar mechanism has been solved with an index-3 augmented Lagrangian formulation combined with the trapezoidal rule as numerical integrator. Different implementations for this method, both dense and sparse, have been developed, using a number of linear algebra software libraries (including sparse linear equation solvers) and optimized sparse matrix computation strategies. Numerical experiments have been performed in order to measure their performance, as a function of problem size and matrix filling. Results show that optimal implementations can increase the simulation efficiency in a factor of 2-3, compared with our starting classical implementations, and in some topics they disagree with widespread beliefs in MBS dynamics. Finally, advices are provided to select the implementation which delivers the best performance for a certain MBS dynamic simulation.

Keywords: *multibody dynamics, real-time, performance, lineal algebra, implementation.*

INTRODUCTION

Dynamic simulation of multibody systems (MBS) is of great interest for dynamics of machinery, road and rail vehicle design, robotics and biomechanics. Computer simulations performed by MBS simulation tools lead to more reliable, optimized designs and significant reductions in cost and time of the product development cycle. The computational efficiency of these tools is a key issue for two reasons. First, there are some applications, like hardware-in-the-loop settings or human-in-the-loop devices, which cannot be developed unless MBS simulation is performed in real-time. And second, when MBS simulation is used in virtual prototyping, faster simulations allow the design engineer to perform what-if-analyses and optimizations in shorter times, increasing productivity and interaction with the model. Therefore, computational efficiency is an active area of research in MBS, and it holds a relevant position in MBS-related scientific conferences and journals.

A great variety of methods to improve simulation speed have been proposed during the last years [1-3]. Most of these methods base their efficiency improvements on the development of new dynamic formulations. However, although implementation aspects can also play a key factor in the performance of numeric simulations, their effect on real-time multibody system dynamics has not been studied in detail. Some recent contributions have investigated the possibilities of parallel implementations [4], but comprehensive comparisons about serial implementations in MBS dynamics have not been published yet.

Multibody dynamics codes make an intensive use of linear algebra operations. This is especially true in global methods, which use a relative big number of coordinates and constraint equations to define the position of the system; these methods usually lead to $O(N^3)$ algorithms, where N is the number of bodies, and spend around 80% of the CPU time in matrix computations. Topological methods lead to O(N) algorithms due to the reduced size of the involved matrices, and therefore the weight of matrix computations is also reduced. However, if flexible bodies are considered, matrix computations take a significant percentage of simulation time even for topological methods.

As a result, the implementation of linear algebra operations is critical to the efficiency of MBS dynamic simulations. These operations can be grouped into two categories: (a) operations between scalars, vectors and matrices, and (b)

solution of linear systems of equations; two additional orthogonal categories can be established based on the data storage format: dense storage or sparse storage. Many efficient implementations for these routines have been made freely available in the last decade. Their performance has been compared in previous works, both in an application-independent context [5-7] and under the perspective of a particular application like Finite Element Analysis [8] or Computational Chemistry [9]. But, as it will be explained in this paper, these studies do not fit the particular features of MBS dynamics, and therefore their conclusions cannot be extrapolated to this field.

The goal of this paper is to compare the efficiency of different implementations of linear algebra operations, and study their effect in the context of MBS dynamic simulation. Results will provide guidelines about which numerical libraries and implementation techniques are more convenient in each case. This information will be very helpful to researchers developing high-performance or real-time multibody simulation codes.

The remainder of the paper is organized as follows: Section 2 describes the test problem and the dynamic formulation used in the numerical experiments to compare the efficiency of different implementations; Sections 3 and 4 present efficient implementations for dense and sparse linear algebra, respectively; Section 5 compares the results obtained in Sections 3 and 4 and extrapolates them to other dynamic formulations; finally, Section 6 provides conclusions, advices for efficient implementations and areas of future work.

2 BENCHMARK SETUP

In order to study the effect of linear algebra implementations in MBS dynamic simulations, a test problem will be solved with a particular dynamic formulation using different software implementations. A starting implementation will also be described, since its efficiency will serve as a reference to measure performance improvements.

2.1 Test Problem

The selected test problem (Fig. 1) is a 2D one degree-of-freedom assembly of four-bar linkages with N loops, composed by thin rods of 1 m length with a uniformly distributed mass of 1 kg, moving under gravity effects. Initially, the system is in the position shown in Figure 1, and the velocity of the x-coordinate of

point B_0 is +1 m/s. The simulation time is 20 s. This mechanism has been previously used as a benchmark problem for multibody system dynamics [3,10].

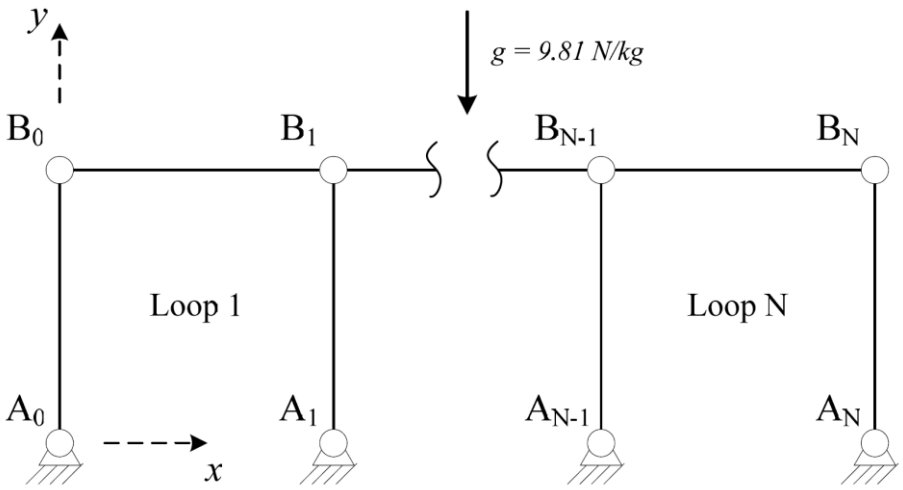

Figure 1: N-four-bar mechanism.

2.2 Dynamic Formulation

The N-four-bar mechanism has been modeled using planar natural coordinates (global and dependent) [11], leading to 2N+2 variables (the x and y coordinates of the B points), and 2N+1 constraints, associated with the constant length condition of the rods. The equations of motion of the whole multibody system are given by the well-known index-3 augmented Lagrangian formulation in the form:

$$\begin{aligned} &\mathbf{M}\ddot{\mathbf{q}} + \boldsymbol{\Phi}_{\mathbf{q}}^{T}\alpha\boldsymbol{\Phi} + \boldsymbol{\Phi}_{\mathbf{q}}^{T}\boldsymbol{\lambda}^{*} = \mathbf{Q} \\ &\boldsymbol{\lambda}_{i+1}^{*} = \boldsymbol{\lambda}_{i}^{*} + \alpha\boldsymbol{\Phi}_{i+1}\ , \qquad i = 0,\ 1,\ 2,\ \ldots \end{aligned} \tag{1}$$

where $\mathbf{M}$ is the mass matrix (constant for the proposed test problem), $\ddot{\mathbf{q}}$ are the accelerations, $\boldsymbol{\Phi}_{\mathbf{q}}$ the Jacobian matrix of the constraint equations, α the penalty factor, $\boldsymbol{\Phi}$ the constraints vector, $\boldsymbol{\lambda}^{*}$ the Lagrange multipliers and $\mathbf{Q}$ the vector of applied and velocity dependent inertia forces. The Lagrange multipliers for each time-step are obtained from an iteration process, where the value of $\boldsymbol{\lambda}_{0}^{*}$ is equal to the $\boldsymbol{\lambda}^{*}$ obtained in the previous time-step.

As integration scheme, the implicit single-step trapezoidal rule has been adopted. The corresponding difference equations in velocities and accelerations are:

$$\begin{aligned} \dot{\mathbf{q}}_{n+1} &= \frac{2}{\Delta t}\mathbf{q}_{n+1} + \hat{\dot{\mathbf{q}}}_n; \qquad \hat{\dot{\mathbf{q}}}_n = -\left(\frac{2}{\Delta t}\mathbf{q}_n + \dot{\mathbf{q}}_n\right) \\ \ddot{\mathbf{q}}_{n+1} &= \frac{4}{\Delta t^2}\mathbf{q}_{n+1} + \hat{\ddot{\mathbf{q}}}_n; \quad \hat{\ddot{\mathbf{q}}}_n = -\left(\frac{4}{\Delta t^2}\mathbf{q}_n + \frac{4}{\Delta t}\dot{\mathbf{q}}_n + \ddot{\mathbf{q}}_n\right) \end{aligned} \tag{2}$$

Dynamic equilibrium can be established at time-step n + 1 by introducing the difference Eq. (2) into the equations of motion (1), leading to a nonlinear algebraic system of equations with the dependent positions as unknowns:

$$\mathbf{f}(\mathbf{q}) = \mathbf{M}\mathbf{q}_{\mathbf{n+1}} + \frac{\Delta t^2}{4}\boldsymbol{\Phi}_{\mathbf{q}_{n+1}}^T\left(\alpha\boldsymbol{\Phi}_{n+1} + \boldsymbol{\lambda}_{n+1}\right) - \frac{\Delta t^2}{4}\mathbf{Q}_{n+1} + \frac{\Delta t^2}{4}\mathbf{M}\hat{\ddot{\mathbf{q}}}_{\mathbf{n}} = 0 \tag{3}$$

Such system, whose size is the number of variables in the model, is solved through the Newton-Raphson iteration

$$\left[\frac{\partial \mathbf{f}(\mathbf{q})}{\partial \mathbf{q}}\right]_i \Delta\mathbf{q}_{\mathbf{i+1}} = -\left[\mathbf{f}(\mathbf{q})\right]_i \tag{4}$$

using the approximate tangent matrix (symmetric and positive-definite)

$$\left[\frac{\partial \mathbf{f}(\mathbf{q})}{\partial \mathbf{q}}\right] \cong \mathbf{M} + \frac{\Delta t}{2}\mathbf{C} + \frac{\Delta t^2}{4}\left(\boldsymbol{\Phi}_{\mathbf{q}}^T\alpha\boldsymbol{\Phi}_{\mathbf{q}} + \mathbf{K}\right) \tag{5}$$

where **C** and **K** represent the contribution of damping and elastic forces of the system (which are zero for the test problem). Once convergence is attained into the time-step, the obtained positions $\mathbf{q}_{\mathbf{n+1}}$ satisfy the equations of motion (1) and the constraint conditions $\boldsymbol{\Phi} = 0$, but the corresponding sets of velocities $\dot{\mathbf{q}}^*$ and accelerations $\ddot{\mathbf{q}}^*$ may not satisfy $\dot{\boldsymbol{\Phi}} = 0$ and $\ddot{\boldsymbol{\Phi}} = 0$. To achieve this, cleaned velocities $\dot{\mathbf{q}}$ and accelerations $\ddot{\mathbf{q}}$ are obtained by means of mass-damping-stiffness orthogonal projections, reusing the factorization of the tangent matrix:

$$\left[\frac{\partial \mathbf{f}(\mathbf{q})}{\partial \mathbf{q}}\right]\dot{\mathbf{q}} = \left[\mathbf{M} + \frac{\Delta t}{2}\mathbf{C} + \frac{\Delta t^2}{4}\mathbf{K}\right]\dot{\mathbf{q}}^* - \frac{\Delta t^2}{4}\mathbf{\Phi}_\mathbf{q}^T \alpha \mathbf{\Phi}_\mathbf{q}$$
$$\left[\frac{\partial \mathbf{f}(\mathbf{q})}{\partial \mathbf{q}}\right]\ddot{\mathbf{q}} = \left[\mathbf{M} + \frac{\Delta t}{2}\mathbf{C} + \frac{\Delta t^2}{4}\mathbf{K}\right]\ddot{\mathbf{q}}^* - \frac{\Delta t^2}{4}\mathbf{\Phi}_\mathbf{q}^T \alpha \left(\dot{\mathbf{\Phi}}_\mathbf{q}\dot{\mathbf{q}} + \dot{\mathbf{\Phi}}_t\right) \quad (6)$$

This method, described in detail in [12], has proved to be a robust and efficient global formulation [13,14]. All numerical experiments will be performed using a time-step Δt of $1.25 \cdot 10^{-3}$ seconds and a penalty factor α of 10^8.

2.3 Starting Implementation

In our starting implementation, the simulation algorithm was implemented using Fortran 90 and the Compaq Visual Fortran compiler. Two versions were developed: (a) a dense matrix storage version, using Fortran 90 matrix manipulation capabilities and the linear equation solver included with this compiler (IMSL Fortran Library, from Visual Numerics), and (b) a sparse matrix storage version, using the MA27 sparse linear equation solver from the Harwell Subroutine Library. These two implementations, typical in the multibody community, have been tuned and improved by our group during the last years, and they have proved to be faster than commercial codes [13,14]. Its efficiency will serve as a reference to measure the performance improvements achieved with the new implementations proposed in this paper.

Table 1: Percentage of the total CPU time required by each algorithm phase in the starting implementation for typical problem sizes: dense version in small problems (10 loops, 22 variables) and sparse version in medium-size problems (40 loops, 82 variables).

Stage	Dense	Sparse
Evaluation of residual and tangent matrix, Eq. (3) and (5)	48%	15%
Evaluation of right-term in orthogonal projections, Eq. (6)	4%	13%
Tangent matrix factorizations and back-substitutions, Eq. (4) and (6)	44%	51%
Other	4%	21%

Table 1 shows the results of a CPU usage profiling in our starting implementations, for both dense and sparse versions, applied to representative problem sizes. As stated in the introduction, matrix computations consume most of the CPU time.

In order to test alternative implementations, the authors have developed a new MBS simulation software, implemented in C++, which can be easily configured to use different matrix storage formats and linear algebra algorithms and implementations. Numerical experiments have been performed on an AMD Athlon64 CPU. After testing different operating systems and compilers, results show that their effect on the performance is an order of magnitude lower than the effect of linear algebra implementations. Final CPU times have been measured using the GNU gcc compiler and the Linux O.S., without loss of generality.

3 EFFICIENT DENSE MATRIX IMPLEMENTATIONS

Global formulations applied to reduced rigid models (e.g. an industrial robot), or topological formulations applied to medium-sized rigid models (e.g. a complete road vehicle), lead to algorithms that operate with small-sized matrices of dimensions less than 50x50. In these cases, dense linear algebra is frequently used in MBS dynamics, since it is *supposed* to provide equal or higher performance than sparse implementations. Achieving real-time in the simulation of these small problems can be a challenge in hardware-in-the loop settings (e.g. advanced Electronic Stability Control systems for automobiles), due to the low computing power of embedded microprocessors, the small time-steps required for hardware synchronization and the added control logic.

A straightforward way to increase the performance of dense matrix computations is by using an efficient implementation of BLAS (Basic Linear Algebra Subprograms). BLAS [15] is a standardized interface that defines routines to perform low level operations between scalars, dense vectors and dense matrices. A Fortran 77 reference implementation is available, and more efficient implementations have been developed by hardware vendors and researchers. These optimized BLAS versions exploit hardware features of modern computer architectures to get the best computational efficiency. In addition to the reference Fortran 77 implementation, three optimized BLAS implementations have been tested:

- ATLAS (Automatically Tuned Linear Algebra Software), which employs empirical techniques to generate an optimal implementation for any hardware architecture [7].

- GotoBLAS, based on optimized assembler kernels, hand-written for the most popular hardware architectures [16].
- ACML, developed by the microprocessor manufacturer AMD for its CPUs [17]. Other hardware vendors also provide their own implementations (MKL from Intel, SCSL from SGI, etc.).

Dynamic simulations can also make a profit of these optimized BLAS implementations in the solution of dense linear equation systems, provided the LAPACK library is used [18], since its linear equation solvers are based on low-level BLAS operations. In addition to the reference LAPACK implementation, written in Fortran 77, some optimized BLAS implementations like ATLAS and ACML supply their own optimized versions of the LAPACK linear solvers.

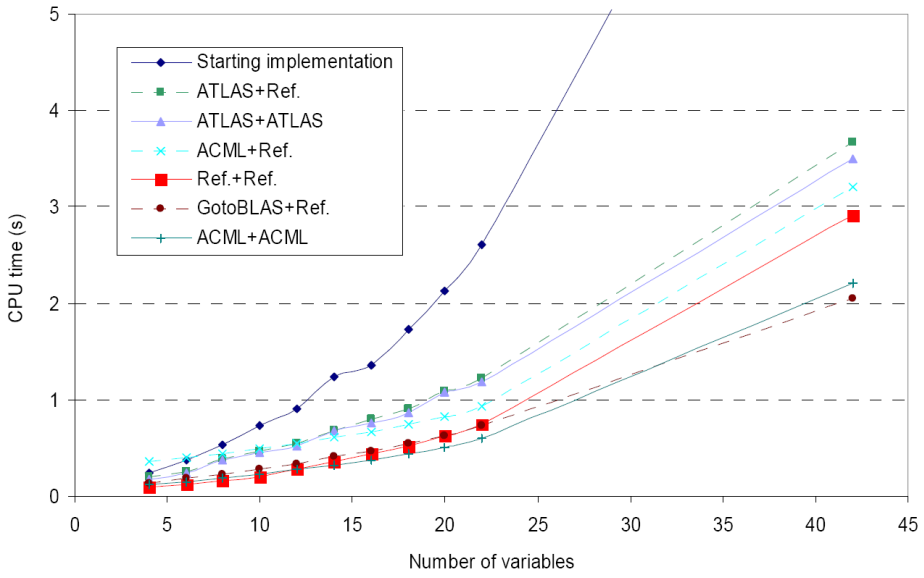

Figure 2: Performance of different dense BLAS and LAPACK implementations.

The proposed test problem, with a number of loops ranging from 1 to 20 (i.e. number of variables ranging from 4 to 42), was solved using different BLAS and LAPACK implementations to perform all matrix computations. Since the tangent matrix in the proposed dynamic formulation is symmetric and positive-definite (SPD), only the lower triangular part of the matrix is computed, and the LAPACK routines DPOTRF and DPOTRS have been used as linear equation solver. Performance results are shown in Figure 2, where the legend text is encoded in the

form “BLAS implementation + LAPACK implementation” (except for the starting implementation), and the combinations are ordered by increasing efficiency.

Results in Figure 2 clearly show the advantage of using BLAS and LAPACK, which speed-up the simulation in a factor between 2 and 5, depending on the problem size, compared with our previous starting implementation. The low performance of the ATLAS implementation, compared to the BLAS reference implementation, can be explained by its high sensitiveness to the development environment (e.g. compiler version) and its current unstable state (it is under strong development). The vendor implementation (ACML) and GotoBLAS deliver the best results except for very small problems (up to 10 variables). The implementation named “Ref.+Ref.” delivers the best performance for very small problems, and 70-80% of the performance of the best implementations for medium-size problems (3 times more efficient than our starting implementation); in addition, it has a very good portability (it is written in plain Fortran 77) and usability: the installation process is straightforward, which is not always true for other implementations.

Since some MBS dynamic formulations lead to a non-symmetric tangent matrix [19], the same numerical experiment has been executed using general algorithms (not SPD-specific) to compute all matrix operations; CPU times are about 15% higher, but the efficiency ranking of Figure 2 is maintained.

4 EFFICIENT SPARSE MATRIX IMPLEMENTATIONS

In MBS dynamics, sparse matrix techniques are used in global formulations applied to medium- or big-sized rigid models; as an example, a global model of an automobile leads to matrices of dimension about 200x200 [14]. If flexible bodies are considered, the matrix size increases, making sparse techniques profitable even if topological formulations are used: a topological model of the same automobile, with some of its bodies characterized as flexible elements (described by component mode synthesis), leads to matrices of dimension about 100x100. In any case, MBS models developed with real-time formulations hardly ever lead to matrix sizes bigger than 1000x1000, significantly smaller than the typical sizes in other applications like Finite Element Analysis (FEA) or Computer Fluid Dynamics (CFD).

Regarding the sparsity, the proposed test problem and MBS dynamic formulation lead to a tangent matrix of size 2N+2 and 12N+4 structural non-zeros. For matrices of size 50x50, 100x100 and 500x500, the corresponding number of non-zeros is 12%, 6% and 1%. These are representative values for MBS simulations, and they are quite higher than typical values in other applications that require sparse matrix technology (FEA, CFD).

Hence, MBS dynamics has two characteristics which make its sparse matrix computations different from other applications:

a) Matrix computations are very repetitive, and the sparse patterns remain constant during the simulation. Therefore, symbolical preprocessing can be applied to almost all matrix expressions at the beginning of the simulation, in order to accelerate the numerical evaluations during the simulation. Section 4.1 presents some tips to exploit this feature.
b) The involved sparse matrices are relatively small and dense, compared with the typical values in sparse matrix technology. Section 4.2 evaluates how sparse linear equation solvers perform in these circumstances.

4.1 Optimized sparse matrix computations

Several numerical libraries are available nowadays to support sparse matrix computations: MTL, MV++, Blitz++, SparseKIT, etc. For our new implementations, we have chosen uBLAS, a C++ template class library that provides BLAS functionality for sparse matrices [20]. Its design and implementation unify mathematical notation via operator overloading and efficient code generation via expression templates. Even though, the performance of some matrix operations can be further improved if some special algorithms are used. Results of CPU usage profiling (similar to Table 1) guided us to optimize three operations:

The first optimized operation is the rank-k update of symmetric matrix, $\mathbf{\Phi}_\mathbf{q}^T \alpha \mathbf{\Phi}_\mathbf{q}$, computed in Eq. (5). Since the sparse structure of the Jacobian matrix $\mathbf{\Phi}_\mathbf{q}$ is constant, a symbolic analysis is performed in order to pre-calculate the sparse pattern of the result matrix and to create a data structure that holds the operations needed to evaluate it during the simulation. In our starting sparse implementation, a similar approach was taken, but the Jacobian matrix was stored as dense, to simplify the operations at the cost of a higher memory usage.

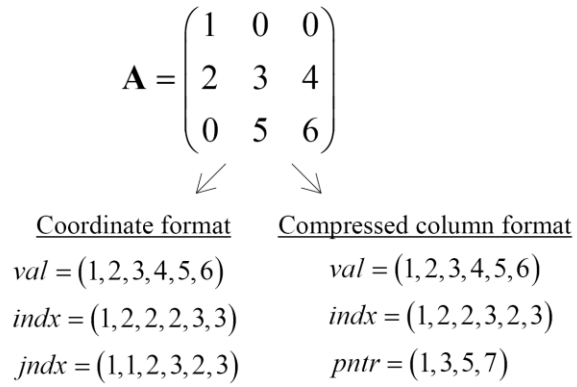

Figure 3: Sparse storage formats used in out implementations.

The second optimized operation is the matrix addition computed in Eq. (5). Our starting sparse implementation used the Harwell MA27 routine as linear equation solver, which requires the sparse matrix to be stored in coordinate format (Figure 3), and allows duplicated entries in the matrix structure. Therefore, the matrix addition is not actually computed, since the different terms are appended as duplicated entries in the tangent matrix. Our new implementation uses the compressed column storage format (Figure 3), since it is required by the sparse linear equation solvers tested in Section 4.2. With this storage format, matrix additions require complex data traversing that slows down the performance. The following approach was taken in order to optimize the operation:

$$\mathbf{B} = t_1\mathbf{A_1} + t_2\mathbf{A_2} \tag{7}$$

In the preprocessing stage, the sparse pattern of **B** is calculated as the union of $\mathbf{A_1}$ and $\mathbf{A_2}$ sparse patterns, and the resulting pattern is added to $\mathbf{A_1}$ and $\mathbf{A_2}$. In this way, $\mathbf{A_1}$, $\mathbf{A_2}$ and **B** share the same sparse pattern (same *indx* and *pntr* arrays in the compressed column storage format shown in Figure 3), and therefore, the matrix addition can be computed as a vector addition of the *val* arrays:

$$\mathbf{val_B} = t_1\mathbf{val}_{\mathbf{A_1}} + t_2\mathbf{val}_{\mathbf{A_2}} \tag{8}$$

This technique increases the number of non-zeros (NNZ) of the addend matrices. In the proposed MBS dynamic formulation, the NNZ of the mass matrix **M** is increased in a 10% approximately, which slows down the matrix-vector

multiplications needed in the right terms of Eq. (3) and (6). However, the simulation timings show that this slowdown is negligible compared with the gains derived from the fast matrix addition.

Finally, the third optimized operation concerns sparse matrix access. The write operation $\mathbf{A}(i,j) = a_{ij}$, straightforward in dense storage, needs additional position lookup when the compressed column storage is used. In the proposed formulation, the update of the Jacobian matrix $\mathbf{\Phi_q}$ in each iteration takes 10-15% of the CPU time. The involved operations are rather simple, and most of this time is spent in matrix access. In order to optimize this procedure, a preprocessing stage evaluates the Jacobian matrix and registers the order in which entries $\mathbf{\Phi_q}(i,j)$ are written in the *val* array of the compressed column format, creating a vector that holds indices to these positions, in the same order of evaluation. Latter, in the simulation stage, access to the Jacobian matrix is performed using this index vector, without the need to map *(i,j)* indices to memory addresses for each write operation.

Table 2: Efficiency of the optimized sparse matrix operations.

Sparse operation	CPU time (microseconds)		Ratio
	Not optimized	Optimized	
1) Rank-k update of symmetric matrix	2528.2	9.4	269
2) Matrix addition	140.9	1.9	74
3) Jacobian matrix evaluation	11.6	3.8	3

Table 2 summarizes the performance gains delivered by the proposed optimizations, compared with the performance delivered by the uBLAS default algorithms (which are similar to other generic sparse matrix libraries). The numerical experiment used the matrix terms derived from an N-four-bar mechanism with N=40 loops, which leads to a tangent matrix of size 82x82. Results show the importance of optimizing rank-k updates and matrix additions, since the performance delivered by off-the-shelf sparse matrix libraries is not satisfactory for these repetitive operations.

4.2 Evaluation of sparse linear equation solvers

Data in Table 1 shows that, in our starting sparse implementation, about 50% of the total CPU time is spent in tangent matrix factorizations and back-substitutions, Eq. (4) and (6). Thus, the main performance improvements in MBS dynamic simulation can be achieved by using a more efficient sparse linear solver. During the last decade, sparse solvers have significantly improved the state of the art of the solution of general sparse linear equation systems, and more than 30 sparse solver libraries are freely available in the World Wide Web [21].

The efficiency of sparse solvers varies greatly depending on the matrix size, structure, number of non-zeros, etc. In addition, solving a sparse linear equation system usually involves three stages: preprocessing (ordering, symbolic factorization), numerical factorization and back substitution; some solvers are very fast in the first stage, while others perform better in the second or third stage. The performance of sparse solvers has been compared in previous works [5,6], but the conditions of these studies (in particular, matrix sizes and percentage of non-zeros) do not fit the above-mentioned particular features of MBS dynamics, and therefore their conclusions cannot be extrapolated to this field. As a result, it is almost impossible to determine, without numerical experiments, which sparse solver will deliver the best performance in an MBS dynamic simulation.

Given the large number of existing sparse solvers, a selection process is required in order to narrow the scope. Solvers for shared memory or distributed memory parallel machines have been discarded, since the small matrix sizes in MBS real-time dynamics (almost fit in the CPU cache memory) makes them unprofitable. The same argument applies to iterative solvers and out-of-core solvers, designed for very big linear equation systems. From the remaining solvers, those that performed best in previous comparative studies have been selected:

- Cholmod, a left-looking supernodal symmetric positive definite solver [22].
- KLU, a solver specially designed for circuit simulation matrices [23].
- SuperLU (serial version), an unsymmetric general purpose solver [24].
- Umfpack, an unsymmetric multifrontal solver [25].
- WSMP, a symmetric indefinite solver [26].

Despite the coefficient matrix is symmetric positive-definite in the proposed dynamic formulation, we have included in the numerical experiments some general, non-symmetric solvers (KLU, SuperLU, Umfpack), since other dynamic formulations lead to a non-symmetric coefficient matrix [19]. In these cases, the whole coefficient matrix (upper and lower parts) is computed, while with symmetric solvers only half matrix is used in the formulation equations. Each solver supports its own set of reordering strategies; all of them have been tested to select the best one in each simulation. In addition, all the optimizations described in the previous Section were applied to our new sparse implementation.

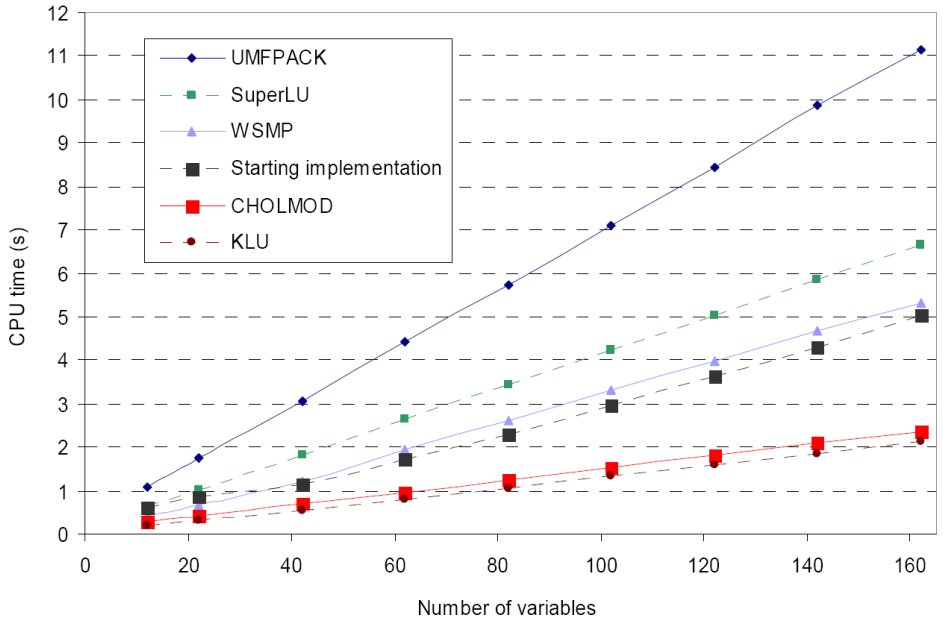

Figure 4: Performance of different sparse linear equation solvers as a function of problem size.

The proposed test problem, with a number of loops ranging from 10 to 500 (i.e. number of variables ranging from 22 to 1002), was solved using different sparse solvers. Performance results are shown in Figure 4 for a number of variables up to 160, since the trends are preserved for higher number of variables. The legend text shows the name of the sparse solvers, ordered by increasing efficiency.

Surprisingly, KLU is the fastest solver, despite being a general solver that does not exploit the symmetric positive definite condition of the coefficient matrix; in addition, it has been designed for circuit simulation problems, which lead to very sparse matrices, the opposite case of MBS dynamics. However, these results have been obtained by using the KLU *refactor* routine for numerical factorizations,

which reuses the pivoting strategy generated in the preprocessing stage. In multibody problems where the elements of the tangent matrix of Eq. (5) may significantly change their relative values during the simulation (e.g. due to violent impacts), the initial pivoting strategy may become invalid and the *refactor* routine would probably accumulate high numerical errors. To avoid this, the KLU solver can recalculate the pivoting strategy in each numerical factorization, but this method increases the CPU times in a 50%. On the other hand, Cholmod, a symmetric positive definite solver, performs at 85% of KLU, despite recalculating the pivoting strategy in each numerical factorization. Our best new sparse implementations (using KLU or Cholmod) perform faster that our starting implementation, in a factor from 2 (small problems) to 3 (big problems of 1000 variables).

4.3 Effect of dense BLAS implementation

Some sparse solvers, like Cholmod, rely on dense BLAS to increase their performance. In addition, some sparse matrix operations (e.g. the optimized matrix addition described in Section 4.1) are actually computed as dense vector operations using BLAS routines. Results shown in Figure 4 have been generated using the reference BLAS implementation. The same numerical experiment has been executed using the faster, optimized GotoBLAS and ACML implementations, and CPU times have decreased only in a 2% - 3%. Hence, the reference BLAS implementation is recommended for MBS dynamics in sparse implementations, since it provides the best compromise between performance and usability.

5 SPARSE VS. DENSE IMPLEMENTATIONS

As stated previously, dense linear algebra is frequently used in MBS dynamics for small problems (dimension of the coefficient matrix lower than 50), since it is supposed to provide higher performance than sparse implementations [27]. Our starting sparse implementation, which already employs some of the optimizations described in Section 4.1, disagrees with this assumption, and this fact is reinforced with the performance of our new optimized implementations: sparse versions perform always faster than dense versions even for small problems, in a factor which ranges from 1.5 (problems of 10 variables) to 5 (problems of 50 variables).

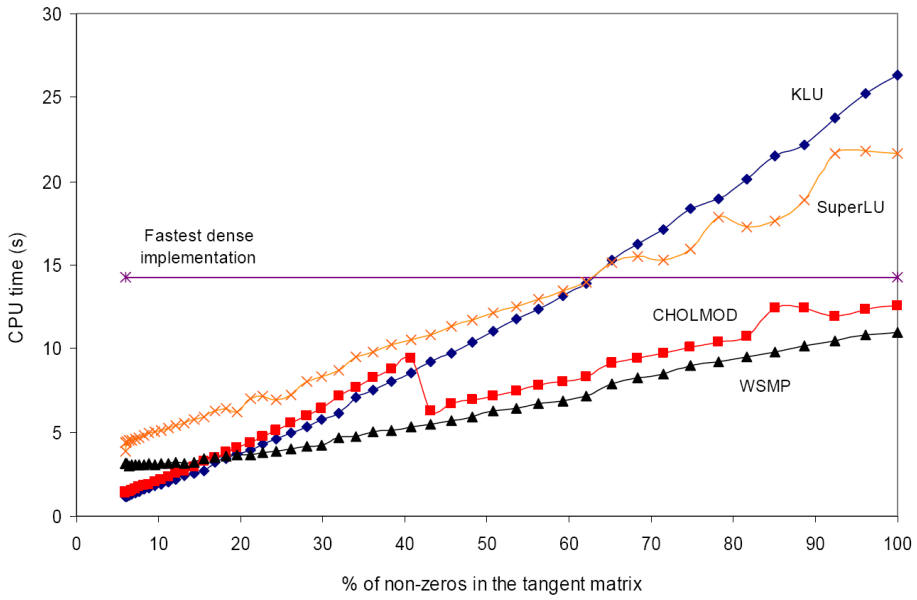

Figure 5: Performance of different sparse linear equation solvers as a function of tangent matrix filling, for a problem size of 100 variables.

However, this conclusion has been obtained for the proposed test problem and dynamic formulation, and it could be argued that it cannot be generalized to other situations that lead to a coefficient matrix with a higher percentage of non-zeros, as in the case of highly constrained mechanism or topological formulations. The objection could be made to the efficiency ranking shown in Figure 4. In order to get insight about this subject, the numerical experiments used to generate Figure 4 were repeated, but in this case artificial non-zeros were introduced in the mass matrix **M**, in order to generate a tangent matrix with a variable percentage of non-zeros. Figure 5 shows the CPU times for a mechanism of 48 loops (100 variables), as a function of matrix filling. Results show that two sparse implementations, based on the Cholmod and WSMP sparse solvers, are always faster than the best dense implementation, even with 100% of non-zeros in the tangent matrix. This surprising fact can be explained by two factors: (a) Cholmod and WSMP rely on dense BLAS routines to perform the factorization, and therefore they start to operate as dense solvers as the matrix filling increases; (b) the percentage of non-zeros is always lower in the Jacobian matrix than in the tangent matrix, hence optimized sparse implementations achieve significant time savings in Jacobian operations, in comparison with dense implementations.

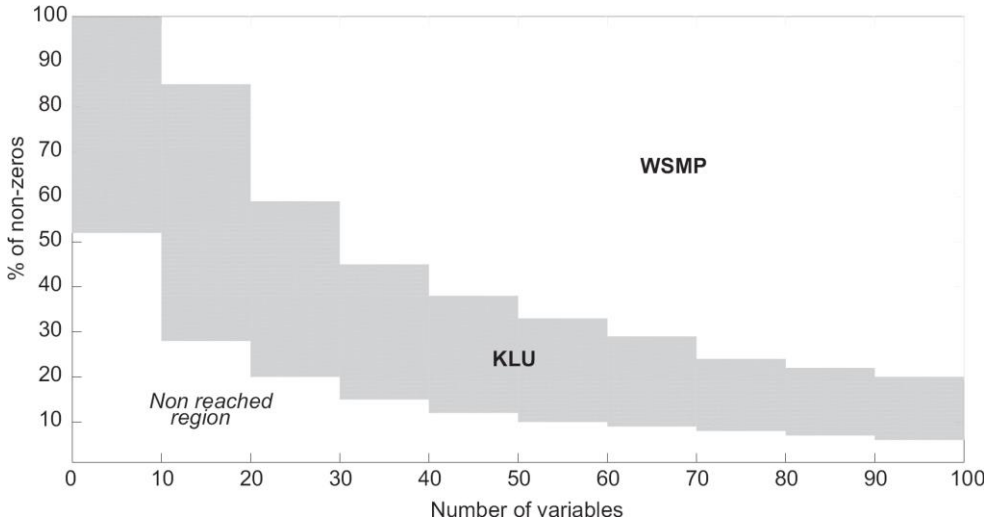

Figure 6: Best implementation, as a function of problem size and percentage of non-zeros in the tangent matrix.

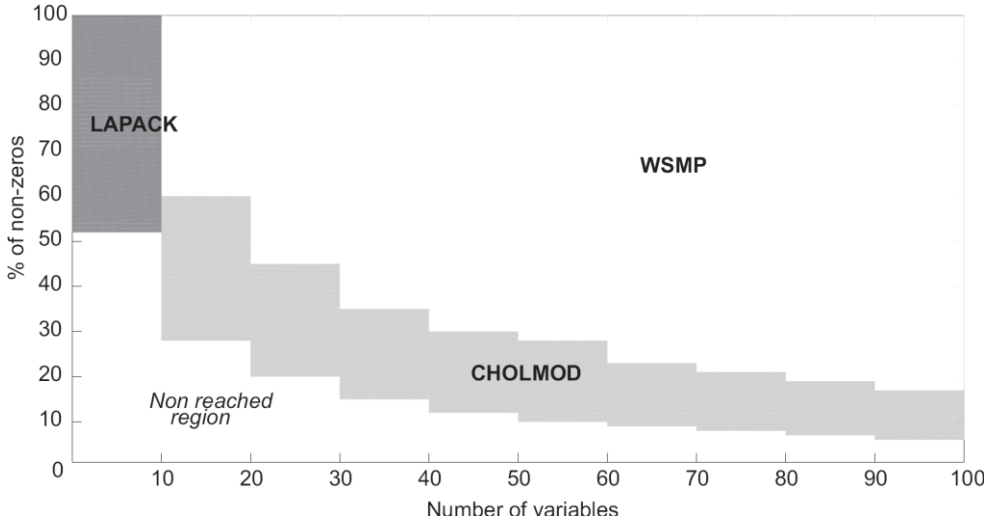

Figure 7: Best implementation, as a function of problem size and percentage of non-zeros in the tangent matrix (*refactor* routine of KLU is not used).

Results for other problem sizes are synthesized in Figure 6: the different regions represent the points *(problem size, matrix filling)* where each implementation delivers the best performance. For most MBS problems and dynamic formulations, a sparse implementation based on the KLU solver will be the frontrunner. However, topological formulations with a symmetric tangent matrix will benefit from a sparse implementation based on the WSMP solver, specially when they are applied to rigid models, which result in a higher matrix filling.

Figure 6 has been obtained by using the KLU *refactor* routine for numerical factorizations. As explained in Section 4.2, this may cause trouble in problems where the entries of the tangent matrix change their relative values significantly during the simulation. If the refactor routine is not used, Figure 7 is obtained. In this case KLU is replaced by Cholmod, WSMP increase its influence area, and the dense implementation based on LAPACK emerges for very small problems (less than 10 variables), but with a very small advantage. Conversely, two exceptions can be mentioned: (a) for dynamic formulations with symmetric indefinite tangent matrices, WSMP would be the frontrunner for almost all the situations, since Cholmod does not support them; (b) for dynamic formulations with unsymmetric tangent matrices, KLU would be the again frontrunner for almost all the situations (even if the *refactor* routine is avoided), since WSMP does not support them.

6 CONCLUSIONS

Regarding the implementation aspects of MBS dynamic simulations, the following conclusions can be established:

- Efficient linear algebra implementations can speed up the efficiency in a factor of 2-3, compared with traditional implementations. In other words, problems of double or triple size can be solved with the same resources.
- The proposed optimizations based on symbolic preprocessing of the sparse matrix computations can deliver huge speedups, since off-the-shelf sparse matrix libraries do not take advantage of the constant sparse pattern of operations during the dynamic simulation.
- Optimized sparse implementations are recommended since they perform better than optimized dense implementations, even for small-sized problems or relatively dense matrices. This disagrees with the widespread belief in MBS dynamics.
- Concerning sparse linear equation solvers, it has been found that KLU, an unfamiliar solver designed for circuit simulation, performs very well with many of the linear equation systems resulting from MBS dynamics. In addition, it was found that the reference BLAS implementation provides the best compromise between performance and usability for sparse implementations.

The results from numerical experiments are summarized in Table 3, which provides a simple decision rule to select the best linear equation solver for MBS dynamics, based on matrix type, size and percentage of non-zeros. Efficient implementations of global MBS dynamic formulations can be easily achieved, provided the above recommendations are followed. All the recommended software libraries are freely available, and the proposed optimization techniques are not bounded to any programming language.

Table 3: Decision rules for selecting the best sparse solver for MBS dynamics, based on matrix type, size and percentage of non-zeros.

Type of tangent matrix	(No. of variables) x (% of non-zeros - 10)	
	< 900	> 900
Symmetric positive definite	KLU (smooth problems) Cholmod (rough problems)	WSMP
Symmetric	KLU	WSMP
Unsymmetric	KLU	KLU

As a consequence of the abovementioned conclusions, the limit for problem size where global formulations perform better than topological formulations, established in the order of 40 variables [14], should be revised. This limit was obtained using dense implementations, and it might get higher if the proposed optimized sparse implementations were used, since their effects on the efficiency are higher in global formulations than in topological formulations. In addition, further work must be carried out in order to determine if the proposed recommendations are still valid for other formulations, since all the numerical experiments have been performed using a particular global formulation.

ACKNOWLEDGEMENTS

This research has been sponsored by the Spanish MEC (grant No. DPI2003-05547-C02-01 and the F.P.U. Ph.D. fellowship No. AP2005-4448) and the Galician DGID (Grant No. PGIDT04PXIC16601PN).

REFERENCES

[1] **Cuadrado J, Cardenal J, Morer P** (1997) Modeling and Solution Methods for Efficient Real-Time Simulation of Multibody Dynamics, Multibody System Dynamics 1: 259-280

[2] **Bae DS, Lee JK, Cho HJ, Yae H** (2000) An Explicit Integration Method for Realtime Simulation of Multibody Vehicle Models, Computer Methods in Applied Mechanics and Engineering 187: 337-350

[3] **Anderson KS, Critchley JH** (2003) Improved 'Order-N' Performance Algorithm for the Simulation of Constrained Multi-Rigid-Body Dynamic Systems, Multibody System Dynamics 9: 185-212

[4] **Anderson K, Mukherjee R, Critchley J, Ziegler J, Lipton S** (2007) POEMS: Parallelizable Open-Source Efficient Multibody Software, Engineering with Computers 23: 11-23

[5] **Gupta A** (2002) Recent Advances in Direct Methods for Solving Unsymmetric Sparse Systems of Linear Equations, ACM Transactions on Mathematical Software 28: 301-324

[6] **Scott JA, Hu YF, Gould NIM** (2006) An Evaluation of Sparse Direct Symmetric Solvers: An Introduction and Preliminary Findings, Applied Parallel Computing: State of the Art in Scientific Computing 3732: 818-827

[7] **Whaley RC, Petitet A, Dongarra JJ** (2001) Automated Empirical Optimizations of Software and the ATLAS Project, Parallel Computing 27: 3-35

[8] **Turek S, Becker C, Runge A** (2001) The FEAST Indices. Realistic Evaluation of Modern Software Components and Processor Technologies, Computers & Mathematics with Applications vol.41, no.10-11: 1431-1464

[9] **Yu JSK, Yu CH** (2002) Recent Advances in PC-Linux Systems for Electronic Structure Computations by Optimized Compilers and Numerical Libraries, Journal of Chemical Information and Computer Sciences 42: 673-681

[10] **Gonzalez M, Dopico D, Lugrís U, Cuadrado J** (2006) A Benchmarking System for MBS Simulation Software: Problem Standardization and Performance Measurement, Multibody System Dynamics 16: 179-190

[11] **García de Jalón J, Bayo E** (1994) Kinematic and Dynamic Simulation of Multibody Systems - The Real-Time Challenge, Springer-Verlag, New York

[12] **Bayo E, Ledesma R** (1996) Augmented Lagrangian and Mass-Orthogonal Projection Methods for Constrained Multibody Dynamics, Nonlinear Dynamics 9: 113-130

[13] **Cuadrado J, Gutierrez R, Naya MA, Morer P** (2001) A Comparison in Terms of Accuracy and Efficiency Between a MBS Dynamic Formulation With Stress Analysis and a Non-Linear FEA Code, International Journal for Numerical Methods in Engineering 51: 1033-1052

[14] **Cuadrado J, Dopico D, Gonzalez M, Naya M** (2004) A Combined Penalty and Recursive Real-Time Formulation for Multibody Dynamics, Journal of Mechanical Design 126: 602-608

[15] **NIST** (2006) Basic Linear Algebra Subprograms. http://www.netlib.org/blas/

[16] **Goto K** (2006) GotoBLAS. http://www.tacc.utexas.edu/resources/software/

[17] **AMD** (2007) AMD Core Math Library. http://developer.amd.com/acml.jsp

[18] **NETLIB** (2007) LAPACK. http://www.netlib.org/lapack/

[19] **Dopico D, Lugris U, Gonzalez M, Cuadrado J** (2006) Two Implementations of IRK Integrators for Real-Time Multibody Dynamics, International Journal for Numerical Methods in Engineering 65: 2091-2111

[20] **Walter J, Kock M** (2006) UBLAS. http://www.boost.org/libs/numeric/

[21] **Dongarra JJ** (2004) Freely Available Software For Linear Algebra On The Web. http://www.netlib.org/utk/people/JackDongarra/la-sw.html

[22] **Chen Y, Davis TA, Hager WW, Rajamanickam S** (2006) Algorithm 8xx: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. http://www.cise.ufl.edu/~davis/techreports/cholmod/tr06-005.pdf

[23] **Davis TA, Stanley K** (2004) KLU: a Clark Kent Sparse LU Factorization Algorithm for Circuit Matrices. http://www.cise.ufl.edu/~davis/techreports/KLU/pp04.pdf

[24] **Demmel JW, Eisenstat SC, Gilbert JR, Li XYS, Liu JWH** (1999) A Supernodal Approach to Sparse Partial Pivoting, Siam Journal on Matrix Analysis and Applications 20: 720-755

[25] **Davis TA** (2004) Algorithm 832: UMFPACK V4.3 - An Unsymmetric-Pattern Multifrontal Method, ACM Transactions on Mathematical Software 30: 196-199

[26] **Gupta A, Joshi M, Kumar V** (1998) WSSMP: A High-Performance Serial and Parallel Symmetric Sparse Linear Solver, Applied Parallel Computing 1541: 182-194

[27] **Cuadrado J, Dopico D** (2004) A Combined Penalty and Semi-Recursive Formulation for Closed-Loops in MBS, Eleventh World Congress in Mechanism and Machine Science, Vols 1-5, Proceedings 637-641