MULTIBODY DYNAMICS 2005, ECCOMAS Thematic Conference
C.L. Bottasso, P. Masarati, L. Trainelli (eds.)
Milano, Italy, 25–28 June 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

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

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 sparse matrix computation strategies. Numerical experiments have been performed using a variable number of loops in the mechanism (i.e. number of variables in the problem). Results show that optimal implementations increase the simulation efficiency in a factor of 2 -3, compared with our starting, non-optimized implementations. Finally, we provide guidelines to increase the efficiency of MBS dynamic simulations, by recommending some optimization techniques and the linear equation solvers which provide the best compromise between performance and usability.*

1 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. 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 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. But 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 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: dense or sparse. Many efficient implementations for all of these routines have been made freely available in the last decade. Their performance has been compared in previous works, whether in an application-independent context [5-7], or 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, taking into account their efficiency and usability (easy-of-use, portability, etc.). This information will be very helpful to researchers developing real-time multibody simulation codes.

The remaining 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; Section 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, guidelines for efficient implementations and areas of future work.

2 BENCHMARK SETUP

In order to study the effect of the 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 by other authors as a benchmark problem for multibody system dynamics [3,10].

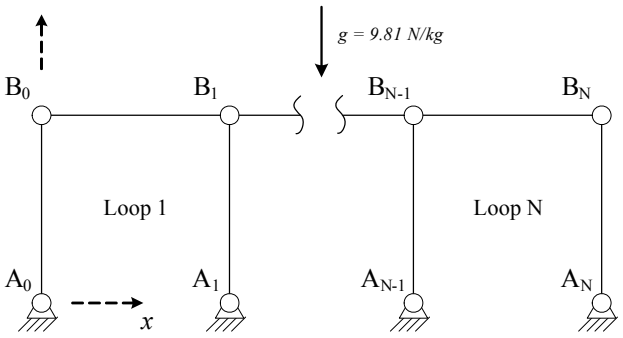

Figure 1: N-four-bar mechanism.

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 multi-body 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 taken 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:

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

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 an 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:

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

This method, described in detail in [12], has proved to be a very 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 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 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 proved to be quite faster than commercial codes [14]. Its efficiency will serve as a reference to measure the performance improvements achieved with the new implementations proposed in this paper.

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: Percentage of the total CPU time required by each algorithm phase in the starting implementation: dense version (for N=10 loops, 22 variables) and sparse version (for N=40 loops, 82 variables).

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

In order to test alternative implementations, we 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. All numerical experiments will be performed using double precision, the GNU gcc compiler with O3 optimization, and a computer with an AMD Athlon64 CPU running the Linux O.S.

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 which 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 higher performance than sparse implementations.

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 which 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 researchers and hardware vendors. These optimized BLAS versions exploit hardware features of modern computer architectures to get the best computational efficiency. In addition to the reference BLAS implementation, we have tested three optimized implementations:

- ATLAS (Automatically Tuned Linear Algebra Software), which applies empirical techniques to generate an optimal implementation for any hardware architecture [7].
- GotoBLAS, based on optimized, hand-written assembler kernels for the most popular hardware architectures [16].
- ACML, developed by the microprocessor manufacturer AMD for its CPU models [17]. Other hardware vendors also provide their own implementations (MKL from Intel, SCSL from SGI, etc.).

Dynamic simulations can also take 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 libraries like ATLAS and ACML supply their own optimized versions of the LAPACK linear solvers. Since the tangent matrix of the proposed dynamic formulation is symmetric and positive-definite, the LAPACK routines DPOTRF and DPOTRS have been used in the simulations.

The proposed test problem, with 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, and performance results are shown in Figure 2. The legend text is encoded in the form "BLAS implementation + LAPACK implementation" (except for the starting implementation), and the combinations are ordered by efficiency.

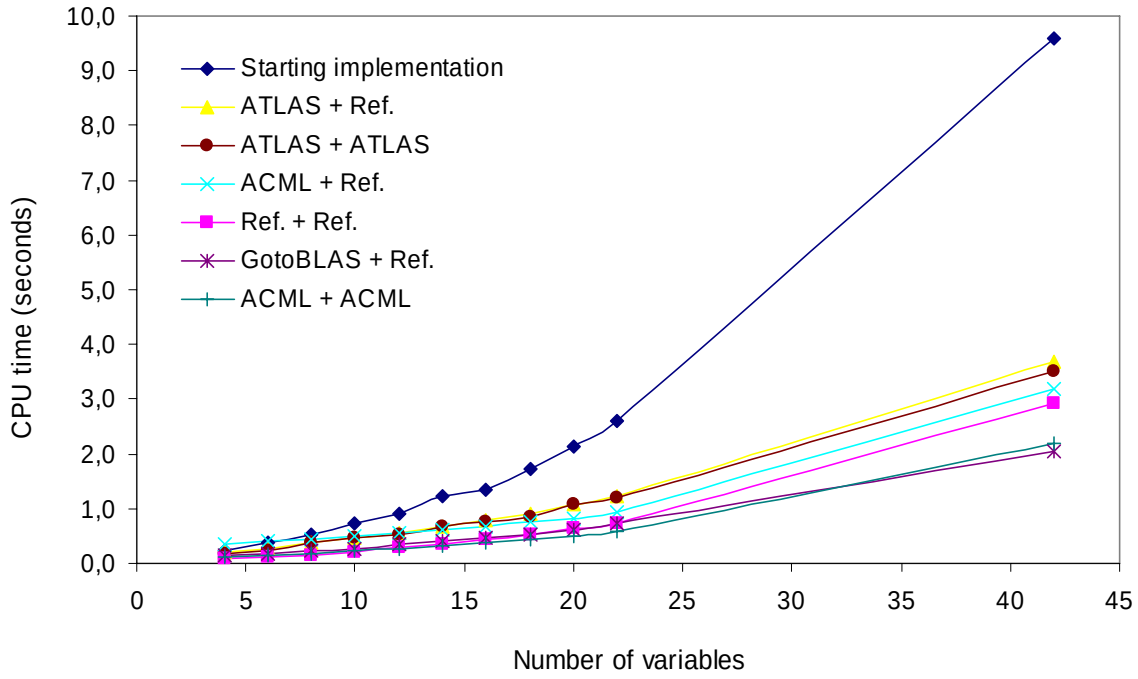

Figure 2: Performance of different dense BLAS and LAPACK implementations.

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 problem size, compared with our previous starting implementation. The ATLAS library is very sensitive to the development environment (e.g. compiler version) and it is under strong development; this can explain its low performance, compared to the reference implementation. ACML and GotoBLAS libraries deliver the best results except for very small problems (less than 10 variables), but they are available only for certain CPU models due to their CPU-bound design. The implementation named "Ref.+Ref." delivers the best performance for very small problems, and 70% of the performance of the best implementation for medium-size problems; in addition, it has a very good portability (it is written in plain Fortran 77) and usability (the installation process is straightforward).

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 a road vehicle leads to matrices of dimension 200x200 approximately [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 road vehicle, with some of its bodies characterized as flexible elements (described by component mode synthesis), leads to matrices of dimension 100x100 approximately. 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 Modeling (FEM) or Computer Fluid Dynamics (CFD).

Regarding the sparsity, the proposed test problem leads, with the presented MBS dynamic formulation, 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 6%, 3% and 0.6%. These are typical values in MBS simulations, and, again, they are quite different (lower in this case) from the values in other applications requiring sparse matrix technology.

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

a) The involved sparse matrices are relatively small and dense. Algorithms that speed-up computations at the cost of a higher memory consumption are not a problem under these conditions.
b) Matrix computations are very repetitive, and the sparse patterns remain constant during the simulation. 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.

4.1 Optimized sparse matrix computations

Several sparse matrix libraries are available 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 [19]. Among the several supported storage formats for sparse matrix, we selected the compressed sparse column format, since it is required by all the sparse linear equation solvers tested in Section 4.2. Its design and implementation unify mathematical notation via operator overloading and efficient code generation via expression templates. Even though, the performance of some matrix computations can be further improved if some special algorithms are applied. 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.

The second optimized operation is the matrix addition computed in Eq. (5). Our starting sparse implementation, implemented in Fortran 90, used the Harwell MA27 routine as linear equation solver, which requires the sparse matrix to be stored in coordinate format, and allows duplicated entries in the matrix structure (Figure 3b). With this data format, the matrix addition is not actually computed, since the different terms are appended as duplicated entries in the tangent matrix. In our new implementation, using the compressed sparse column format

(Figure 3c), matrix additions require complex data traversing that slows-down the performance.

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 3 & 4 \\ 0 & 5 & 6 \end{pmatrix} \quad \text{(a)}$$

$$val = (1,2,3,4,5,6) \qquad indx = (1,2,2,2,3,3) \qquad jndx = (1,1,2,3,2,3) \quad \text{(b)}$$

$$val = (1,2,3,4,5,6) \qquad indx = (1,2,2,3,2,3) \qquad pntr = (1,3,5,7) \quad \text{(c)}$$

Figure 3: Sparse storage formats used in out implementations: (a) example matrix; (b) coordinate format, used in the starting implementation; (c) compressed column format, used in the new implementation.

The following approach was taken in order to compute matrix additions in compressed sparse column format:

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

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** have the same sparse pattern (same *indx* and *pntr* arrays in the compressed sparse column format), and therefore the matrix addition can be computed as a vector addition of the *val* arrays, see Figure (7):

$$\mathbf{val_B} = t_1\mathbf{val}_{\mathbf{A_1}} + t_2\mathbf{val}_{\mathbf{A_2}} \tag{7}$$

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 slow-down is negligible compared with the gains derived from the fast matrix addition.

Finally, the third optimized operation is the sparse matrix access. The write operation $\mathbf{A}(i,j) = a_{ij}$, straightforward in dense storage, needs additional position lookup in the compressed sparse column format. In the proposed formulation, the update of the Jacobian matrix $\mathbf{\Phi_q}$ in each iteration takes 10% 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, creating a vector which holds indices to positions in the *val* array of the compressed column format, in the same order of evaluation. Latter, in the simulation stage, the Jacobian evaluation is performed using this index vector, without the need to map *(i,i)* indices to memory addresses.

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.

Sparse operation	CPU time (microseconds)		(%)
	Not optimized	Optimized	
1) Rank-k update of symmetric matrix	2528.2	9.4	0.4
2) Matrix addition	140.9	1.9	1.3
3) Jacobian matrix evaluation	11.6	3.8	32.5

Table 2: Optimized sparse matrix computations.

4.2 Evaluation of sparse linear equation solvers

Data in Table 1 shows that, in our starting sparse implementation, 50% of the total CPU time is spent in tangent matrix factorizations and back-substitutions, Eq. (4) and (6). Thus, the main performance improvements 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 [20].

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 best 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) 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 fits MBS dynamic simulations best.

We have selected a set of free sparse solvers that appear to suit MBS dynamics. 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 [21].
- KLU, a solver specially designed for circuit simulation matrices [22].
- SuperLU (in its serial version), an unsymmetric general purpose solver [23].
- Umfpack, an unsymmetric multifrontal solver [24].
- WSMP, a symmetric indefinite solver [25].

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. 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 a set of reordering strategies; all of them were tested to select the best one. In addition, all the optimizations described in the previous Section were applied to our new sparse implementation.

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, but the linear trends are also preserved for higher number of variables. The legend text shows the name of the sparse solvers, ordered by increasing efficiency.

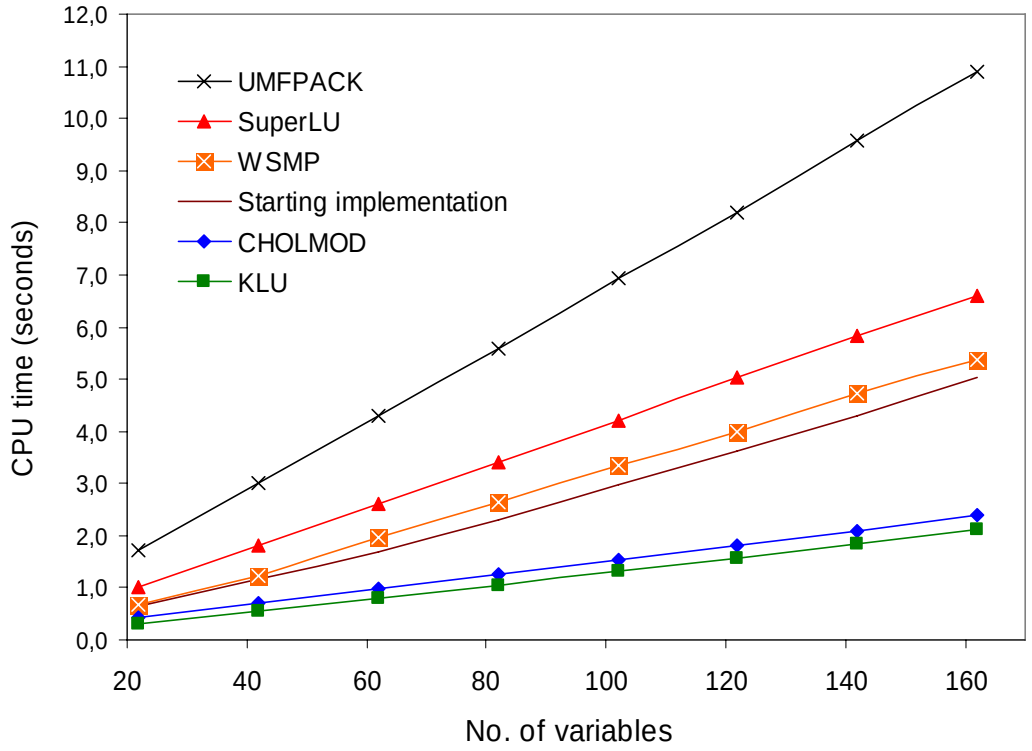

Figure 4: Performance of different sparse linear equation solvers.

Surprisingly, KLU is the fastest solver, despite it is 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 a very sparse matrix, 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. Therefore, 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).

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, as we recommended in Section 3. We have executed the same numerical experiment using the faster GotoBLAS and ACML implementations, and CPU times have only decreased in a 2% - 3%. Hence, we keep the recommendation about using the reference BLAS implementation for MBS dynamics, even in sparse implementations, since it provides the best compromise between performance and usability.

5 SPARSE IMPLEMENTATIONS 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. 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 implementations.

Number of variables	4	20	42
CPU time ratio (dense/sparse)	1.3	2.4	5.0

Table 3: CPU time ration between best dense and sparse implementations, for small problem sizes.

Table 3 shows the CPU time ratios between the best dense and sparse implementations, confirming that modern, optimized sparse implementations are faster than dense, even for small problems. 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 problems or formulations that lead to a coefficient matrix with a higher number of non-zeros, as in the case of topological formulations. In order to get insight about this fact, an additional numerical experiment has been performed: a linear equation system with coefficient matrix of variable size and number of non-zeros has been solved using the best dense and sparse solvers. The curves shown in Figure 5 represent the points (size, number of non-zeros) where both methods take the same CPU time, depending on the sparse solver (KLU or Cholmod). In the region below the curves, the sparse method is faster. As a result, sparse implementations solve linear equations faster even if the number of non-zeros is close to 60%. Since the coefficient matrix resulting from MBS dynamic formulations rarely exceeds a 50% of non-zeros in the worst cases (topologic formulations applied to mechanisms with many closed loops), and the overhead caused by the sparse implementation in simulation stages different from linear equation solving is low, it can be affirmed that sparse implementations provide better performance for most of the MBS dynamic simulations, even in small-sized problems.

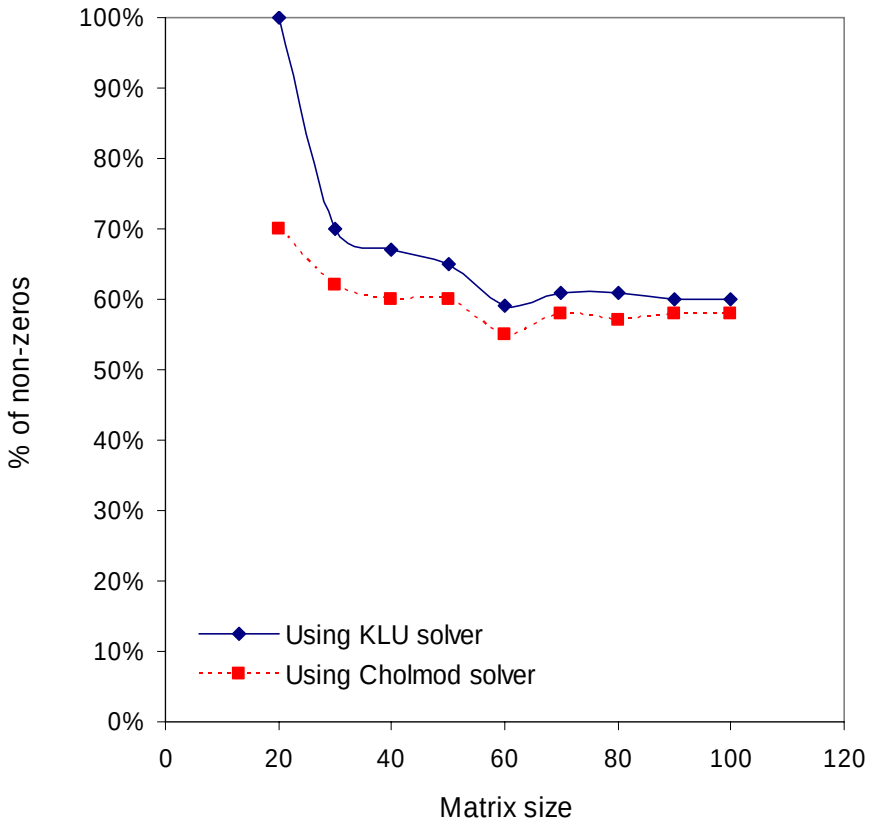

Figure 5: Equilibrium point between best dense and sparse implementations, as a function of matrix size and % of non-zeros.

6 CONCLUSIONS

Based on the results from the different numerical experiments, the following conclusions can be established:

- Efficient linear algebra implementations can speed-up the efficiency of MBS dynamic simulations in a factor of 2-3, compared with traditional implementations.
- Sparse implementations perform best than dense implementations, even for small-sized problems and a relatively dense coefficient matrix. Efficiency gains range approximately from 2 to 5 for less than 40 variables.
- Recently developed sparse matrix libraries greatly simplify the task of developing efficient sparse implementations for MBS dynamics. However, the performance of their default algorithms is poor, since they do not take profit of the constant sparse pattern of the involved matrices in MBS dynamic simulations. Optimizations based on symbolic preprocessing of the sparse matrix computations can deliver huge speedups in MBS dynamics.

- The Cholmod sparse linear equation solver is recommended for dynamic formulations leading to a symmetric matrix, when they are applied to non-smooth problems (with high variations in the magnitude of the elements of the coefficient matrix).
- The KLU sparse linear equation solver is recommended in two cases: (a) dynamic formulations leading to a unsymmetric matrix; (b) dynamic formulations leading to a symmetric matrix and applied to smooth problems (without high variations in the magnitude of the elements of the coefficient matrix); in this last case, the *refactor* routine can be used to speed-up the simulation.
- BLAS is recommended to compute low-level matrix and vector operations, instead of other alternatives (Fortran 90 matrix operators, or hand-written loops). In particular, the reference BLAS implementation provides the best compromise between performance and usability.

In summary, developing an efficient implementation for a MBS dynamic formulation can be easily achieved provided the above recommendations are followed. All the recommended software libraries are free and open-source (they can be downloaded from the Internet), and the proposed optimization techniques are not bounded to the C++ language, since they can be easily implemented in other programming languages.

ACKNOWLEDGEMENTS

This research has been sponsored by the Spanish MEC (grant No. DPI2003-05547-C02-01) and the Galician DGID (Grant No. PGIDT04PXIC16601PN).

REFERENCES

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

[2] D. S. Bae, J. K. Lee, H. J. Cho, H. Yae. An Explicit Integration Method for Realtime Simulation of Multibody Vehicle Models. *Computer Methods in Applied Mechanics and Engineering*, **187**, 337-350, 2000.

[3] K. S. Anderson, J. H. Critchley. Improved 'Order-N' Performance Algorithm for the Simulation of Constrained Multi-Rigid-Body Dynamic Systems. *Multibody System Dynamics*, **9**, 185-212, 2003.

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

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

[6] J. A. Scott, Y. F. Hu, N. I. M. Gould. 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, 2006.

[7] R. C. Whaley, A. Petitet, J. J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, **27**, 3-35, 2001.

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

[9] J. S. K. Yu, C. H. Yu. 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, 2002.

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

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

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

[13] J. Cuadrado, R. Gutierrez, M. A. Naya, P. Morer. 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, 2001.

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

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

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

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

[18] NETLIB. LAPACK. http://www.netlib.org/lapack/, 2007.

[19] J. Walter, M. Kock. UBLAS. http://www.boost.org/libs/numeric/, 2006.

[20] J. J. Dongarra. Freely Available Software For Linear Algebra On The Web. http://www.netlib.org/utk/people/JackDongarra/la-sw.html, 2004.

[21] Y. Chen, T. A. Davis, W. W. Hager, S. Rajamanickam. Algorithm 8xx: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. http://www.cise.ufl.edu/~davis/techreports/cholmod/tr06-005.pdf, 2006.

[22] T. A. Davis, K. Stanley. KLU: a Clark Kent Sparse LU Factorization Algorithm for Circuit Matrices. http://www.cise.ufl.edu/~davis/techreports/KLU/pp04.pdf, 2004.

[23] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. Y. S. Li, J. W. H. Liu. A Supernodal Approach to Sparse Partial Pivoting. *Siam Journal on Matrix Analysis and Applications*, **20**, 720-755, 1999.

[24] T. A. Davis. Algorithm 832: UMFPACK V4.3 - An Unsymmetric-Pattern Multifrontal Method. *ACM Transactions on Mathematical Software*, **30**, 196-199, 2004.

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