Proceedings of the ASME 2009 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference
IDETC/CIE 2009
August 30 - September 2, 2009, San Diego, California, USA

DETC2009-86274

PARALLEL LINEAR EQUATION SOLVERS AND OPENMP IN THE CONTEXT OF MULTIBODY SYSTEM DYNAMICS

Francisco González[*] **Alberto Luaces** **Daniel Dopico** **Manuel González**
Escuela Politécnica Superior, Universidad de A Coruña
Mendizábal s/n, Ferrol, A Coruña, Spain

ABSTRACT

Computational efficiency of numerical simulations is a key issue in multibody system (MBS) dynamics, and parallel computing is one of the most promising approaches to increase the computational efficiency of MBS dynamic simulations.

The present work evaluates two non-intrusive parallelization techniques for multibody system dynamics: parallel sparse linear equation solvers and OpenMP. Both techniques can be applied to existing simulation software with minimal changes in the code structure; this is a major advantage over MPI (Message Passing Interface), the de facto standard parallelization method in multibody dynamics. Both techniques have been applied to parallelize a starting sequential implementation of a global index-3 augmented Lagrangian formulation combined with the trapezoidal rule as numerical integrator, in order to solve the forward dynamics of a variable number of loops four-bar mechanism. This starting implementation represented a highly optimized code, where the overhead of parallelization would represent a considerable part of the total amount of elapsed time in calculations.

Several multi-threaded solvers have been added to the original software. In addition, parallelizable regions of the code have been detected and multi-threaded via OpenMP directives. Numerical experiments have been performed to measure the efficiency of the parallelized code as a function of problem size and matrix filling ratio. Results show that the best parallel solver (Pardiso) performs better than the best sequential solver (CHOLMOD) for multibody problems of large and medium sizes leading to matrix fillings above 10 non-zeros per variable. OpenMP also proved to be advantageous even for problems of small sizes, in despite of the small percentage of parallelizable workload with respect to the total burden of the execution of the code. Both techniques delivered speedups above 70% of the maximum theoretical values for a wide range of multibody problems.

1 INTRODUCTION

Computational efficiency of numerical simulations is a key issue in multibody system (MBS) dynamics. When MBS dynamics is used in Computer Aided Design and Engineering, faster simulations allow the design engineer to perform what-if-analyses and optimizations in shorter times, increasing productivity and interaction with the model. Moreover, some applications like hardware-in-the-loop settings or human-in-the-loop devices cannot be developed unless MBS forward dynamic simulations are performed in real-time. Hence, computational efficiency is a very active area of research in multibody systems dynamics.

Parallel computing is one of the approaches to increase the computational efficiency of MBS dynamic simulations. The first parallel MBS algorithm was proposed by Kasahara in 1987 [1]; since then, a variety of formulations and simulation algorithms have been developed to exploit parallel computing architectures in MBS dynamics ([2-8], among others). Some of these algorithms apply parallelization directly at the level of equations of motion, which are formulated in a form that facilitates the concurrent evaluation of their different terms, see e.g. Bae [9], Hwang [10] and Avello [11]; most of these algorithms are based on recursive or semi-recursive formulations. Other algorithms apply substructuring techniques to partition the multibody system in disjoint subdomains, which are solved concurrently taking into account the interconnection constrains, see e.g. Mukherjee [12] and Quaranta [13]. With regard to the implementation, the Message Passing Interface (MPI) [14] has become the de facto standard for the parallelization of multibody dynamic simulation codes [8,13,15]. MPI is a message-passing application programmer interface that provides functionality to enable communication and synchronization between a set of processes which run concurrently. Due to its language independence, high performance, scalability and good portability through

[*] PhD student and corresponding author, Phone: (+34) 981337400 ext. 3870, Fax: (+34) 981337410, Email: fgonzalez@udc.es

completely different parallel architectures (from shared-memory processors to computer clusters), it has been broadly accepted in the field of parallel MBS dynamics.

The aforementioned parallel methods for MBS dynamics could be described as *intrusive* parallelization, since they introduce major modifications both in formulations and implementations. Formulations are specifically designed to obtain highly parallelizable numerical computations, and most importantly, parallel MPI-based implementations enforce a particular MPI-oriented code design: the programmer must explicitly divide tasks in processes and insert message-passing operations for data transfer and synchronization. As a result, the structure of a MPI-based parallel code is usually quite different from its sequential counterpart. These parallelization methods have proved to attain very good results in terms of efficiency and scalability in the context of MBS dynamics, as demonstrated e.g. by Anderson [15] and Quaranta [13]. However, their intrusive character makes them quite difficult to apply to existing sequential MBS dynamic simulation codes. Many of these sequential packages, developed by academia, still have a great value as research tools and they are successfully used in ongoing industrial applications. Due to their internal complexity and design dependencies with third-party software, parallelization of these MBS packages by intrusive methods like MPI would be very time-consuming and error-prone. For that reason, most of them remain as sequential codes which cannot take advantage of today's almost ubiquitous availability of parallel computing architectures, present even in low-cost laptop computers. This limitation will be accentuated in the future, since trends indicate that performance of single processors is close to reach its limit and that multi-core processors are the preferred technology to increase computing power in the next decade [16].

The goal of this work is to investigate alternative *non-intrusive* parallelization methods for MBS dynamics, which do not require major modifications in existing formulations and implementations. Although their scalability may be inferior when compared to intrusive methods, such non-intrusive methods could be easily applied to parallelize the abovementioned legacy sequential MBS simulation packages, and they may also reduce the effort required to develop some kinds of new parallel formulations and implementations. This work investigates two *non-intrusive* parallelization methods for MBS dynamics: (1) the use of parallel sparse linear equation solvers, and (2) the OpenMP parallel programming model.

Linear equation solvers represent an opportunity for non-intrusive parallelization since the solution of linear equation systems is a key process in many MBS dynamic simulation codes. This linear algebra operation is present in almost all simulation methods except some types of fully recursive formulations [17], although its weight in the total computation time of the simulation depends on the type of problem and formulation. Global formulations, which use a high number of coordinates and constraint equations to define the position of the multibody system, lead to comparatively big sparse linear equation systems whose solution usually consumes around 30-60% of the total CPU time in a dynamic simulation. Topological formulations lead to smaller and more compact linear equation systems, and therefore their weight is reduced to less than 30% of the total CPU time; however, if flexible bodies are considered, matrix sizes increase and the solution of linear equation systems also takes a significant percentage of the CPU time even for topological formulations. As a result, the performance of the linear equation solver is critical to the efficiency of most MBS dynamic simulations. The replacement of a sequential solver by a parallel solver is considered a non-intrusive parallelization technique because it only requires minor changes in the code, provided both solvers use similar sparse matrix storage formats. Many parallel linear equation solvers have been developed in the last years, but they are not considered to be appropriate for MBS dynamics due to the small matrix sizes involved in this field of computational mechanics. Comparative studies about their performance have been published by Gupta [18,19], Tracy [20] and Davis [21]; however, the test problems used in these studies do not fit the particular features of MBS dynamics, specially in regard to matrix sizes (in MBS dynamics, typical sizes are at least two orders of magnitude smaller than in Finite Element Analysis or Computational Fluid Dynamics), and therefore their conclusions cannot be extrapolated since parallel solvers will perform very differently under these circumstances. The first contribution of this work is the evaluation of the efficiency and suitability of parallel sparse linear equation solvers in the context of multibody system dynamics, a subject that has not been investigated yet.

The second non-intrusive parallelization method explored in this work is the OpenMP parallel programming model [22]. OpenMP is a standard application programming interface to support multi-threaded parallel programming. It is scalable and portable like MPI, but it has two important differences. First, OpenMP is only targeted at shared-memory multiprocessor architectures, while MPI supports both shared- and distributed-memory architectures. However, this OpenMP limitation is not a severe disadvantage in the field of MBS forward dynamics: due to the characteristics of the problem, concurrent tasks running a parallelized simulation must exchange data several times per integration step (usually in the order of milliseconds), causing a high communication overhead compared with other applications. As a consequence, gains obtained from concurrent computation can be easily outweighed by the high communication overhead in distributed-memory architectures like PC clusters [13]. Conversely, the low communication overhead of shared-memory architectures, supported by OpenMP, makes them more appropriate to run parallel MBS simulations. Another advantage of shared-memory architectures is the availability of low-cost commodity hardware with 2 or 4 CPU cores, like Intel® Core™ 2 Quad and AMD Phenom™ X4. The second core difference between OpenMP and MPI concerns with the programming model: OpenMP is based on a multi-threaded model simpler to use than the MPI's multi-process model. This key difference delivers important advantages when OpenMP is applied to parallelize a sequential code [23]: (a) the initial design can be maintained and only minor changes in the code are required, (b) data transfer and task synchronization are handled transparently by OpenMP, (c) parallelization can be applied incrementally. These three advantages make OpenMP a non-intrusive parallelization method when compared to MPI. On the other hand, Krawezik [24] demonstrated that OpenMP cannot achieve the same performance as MPI for some types of numerical problems and code designs, hence its pros and cons in a particular domain shall be evaluated before claiming it as a better technique than MPI. Despite of its potential advantages, studies about the efficiency of OpenMP in the context of MBS dynamics have not been published yet, and this subject will be the second contribution of this work.

The rest of the paper is organized as follows: Section 2 describes the numerical experiments used to evaluate the efficiency and applicability of the two proposed non-intrusive parallelization methods: test problem, dynamic formulation, and parallelization procedures applied to a starting sequential implementation. Section 3 presents and analyzes the results of numerical experiments. Finally, Section 4 extracts conclusions and suggests future work.

2 METHODS

In order to study the efficiency and applicability of the two proposed non-intrusive parallelization methods, a test problem will be solved with a given dynamic formulation. This formulation is initially implemented in a sequential simulation code, which will be parallelized by means of parallel linear equation solvers and OpenMP.

This test setup represents a worst-case scenario for parallelization in terms of problem, dynamic formulation and implementation, as it will be explained in the following subsections. With this approach, the obtained performance results will represent a lower limit when the non-intrusive parallelization methods investigated in this work are applied to legacy MBS simulation codes.

2.1 Test problem:

The selected test problem is a 2D one degree-of-freedom assembly of four-bar linkages with *L* 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 by Anderson [25] and González [26,27].

2.2 Dynamic formulation:

The L-four-bar mechanism has been modelled using planar natural coordinates (global and dependent) [17], leading to $2L+2$ variables (the *x* and *y* coordinates of the B points), and $2L+1$ 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:

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

where $\mathbf{M}$ is the mass matrix (constant for the proposed test problem), $\ddot{\mathbf{q}}$ are the accelerations, $\mathbf{\Phi}_{\mathbf{q}}$ the Jacobian matrix of the constraint equations, α the penalty factor, $\mathbf{\Phi}$ 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.

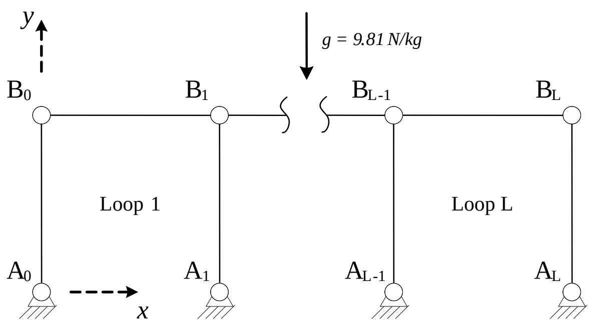

Figure 1: *L*-loop four-bar mechanism.

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:

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

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)

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

where $\mathbf{C}$ and $\mathbf{K}$ 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 $\mathbf{q}_{n+1}$ satisfy the equations of motion (1) and the constraint conditions $\mathbf{\Phi} = \mathbf{0}$, but the corresponding sets of velocities $\dot{\mathbf{q}}^*$ and accelerations $\ddot{\mathbf{q}}^*$ may not satisfy $\dot{\mathbf{\Phi}} = \mathbf{0}$ and $\ddot{\mathbf{\Phi}} = \mathbf{0}$. 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:

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

All numerical experiments will be performed using a time-step Δt of 10^{-3} seconds and a penalty factor α of 10^8. This global method, described in detail in Bayo [28], has proved to be a very robust and efficient global formulation, see e.g. Cuadrado [29,30], but it has been designed for sequential computation and it is not as suitable for parallelization as topological formulations. For that reason, it nearly represents a worst-case scenario for parallelization with regard to dynamic formulations.

The number of loops in the test mechanism can be adjusted to generate problems of different sizes. In MBS dynamics, global formulations generate several hundreds of variables when applied to automotive or railway vehicles made up of rigid bodies. Topological formulations lead to problems of smaller size, but when body flexibility needs to be considered, the number of variables increases even with this kind of formulations. If flexible bodies are described by component mode synthesis, as explained by Ambrosio [31] and Cuadrado [29], multibody models of automobile or railway vehicles can easily exceed 1000 variables. If non-linear elastic or plastic behaviour is considered, the number of variables in the problem is augmented by the degrees of freedom of the finite element discretization of the flexible bodies, see e.g. García Orden [32] or Sugiyama [33]; under these circumstances, multibody models in industrial applications may reach 10^4 variables. This number can be considered the upper-level limit in the field of multibody dynamics, at least during the next decade. For that reason, the numerical experiments will be performed using a number of variables *N* that ranges from 100 to 8000 (generated by a number of loops *L* from 49 to 3999).

2.3 Initial sequential implementation:

The dynamic formulation described above has been implemented in an in-house developed C++ MBS simulation software. This initial implementation has been heavily optimized for sequential execution by using efficient BLAS implementations [34] for dense linear algebra, symbolic pre-processing of sparse matrix computations, fast access to sparse storage formats and state-of-the-art sequential lineal equation solvers, as described by González [27]. These optimizations reduced CPU times by a factor of 3 compared with more traditional implementations of the same dynamic formulation. On the other side, such a highly optimized sequential code makes it difficult to gain advantage from parallelization: since computations are performed at higher FLOPS rates (floating point operations per second) and in shorter times, the relative weight of the communication overhead associated with parallelization becomes higher; in addition, some optimization techniques make fine-grained parallelization unable to be applied to certain code sections, as it will be explained later. Again, the described initial implementation represents a nearly worst-case scenario for parallelization. Indeed, the parallelization of this code by means of MPI would be very cumbersome.

Table 1 summarizes the results of a performance analysis of the initial sequential formulation for tests problems of 1000 and 8000 variables. Both cases show very similar profiling results, since the use of symbolic pre-processing of sparse computations through all the code leads to nearly *O(N)* tasks in spite of using a dynamic formulation usually classified as *O(N^3)*. This performance analysis will be used to guide the parallelization described in the next subsections.

Table 1: Performance analysis of the initial sequential implementation for problems of N variables.

Task	Description	Eq.	% of elapsed time	
			N=1000	N=8000
1	Predictor	(2)	4.1	4.0
2	Evaluate dynamic terms	(3,5)	9.3	9.8
3	Evaluate tangent matrix	(5)	11.8	11.8
4	Evaluate residual vector	(3)	7.6	7.6
5	Factorize tangent matrix	(4)	36.8	36.7
6	Back-substitution	(4)	5.9	5.8
7	Project velocities	(6)	9.4	9.3
8	Project accelerations	(6)	12.3	12.2
9	Other	-	2.8	2.8
Total elapsed time (s)			10.0	102.4

2.4 Parallelization with multi-threaded solvers

Table 1 shows that around 54% of the CPU time is consumed by the solution of linear equation systems: matrix factorization (task 5, close to 37%) and back-substitutions (task 6 and part of tasks 7 and 8). This high contribution is caused by the simplicity of the dynamic terms in the proposed test problem (task 2); in problems with time-consuming force, constraint and Jacobian evaluations, task 2 can achieve higher percentages of runtime and reduce the contribution of linear equation systems. Nevertheless, this operation is a significant bottleneck in most MBS dynamic simulations and represents an important opportunity for non-intrusive parallelization.

In a previous work [27], the authors measured the efficiency of different dense and sparse *sequential* linear equation solvers in the real-time simulation of MBS dynamics; the number of variables *N* in that study ranged from 10 to 100. Results demonstrated that current state-of-the-art sparse implementations outperform dense implementations even for very small problems (e.g., 20 variables), contradicting a widespread conviction in MBS dynamics. Three sequential solvers were found to be the most efficient ones, as a function of the type of multibody problem and dynamic formulation:

- CHOLMOD, a left-looking supernodal symmetric positive-definite solver [35].
- KLU, an unsymmetric solver specially designed for circuit simulation [36].
- WSMP (sequential version), a symmetric indefinite or unsymmetric solver [37].

In this work, these three top-performing sequential solvers will be compared against parallel solvers. Given the large number of existing parallel sparse solvers, a selection process has been applied to narrow the scope: iterative solvers have been discarded, since they have a high communication overhead during each iteration, so they work efficiently only for very large problems out of the scope of MBS dynamics [38]; the same argument applies to out-of-core solvers. From the remaining parallel linear equation solvers, three of them which have demonstrated good performance for matrix sizes close to those found in MBS dynamics [18,19] have been selected to evaluate their performance in this field:

- SuperLU (multi-threaded version), a nonsymmetric solver [39].
- Pardiso, a symmetric or unsymmetric, positive definite or indefinite solver [40].
- WSMP (multi-threaded version) [37].

The efficiency of a linear equation solver depends on three factors: matrix size, sparsity pattern and matrix filling. In this study, the effect of matrix size has been analyzed by solving the test problem with a number of variables N ranging from 100 to 8000. The effect of the sparsity pattern has been greatly diminished by reordering the tangent matrix: each of the six benchmarked solvers supports different fill reducing reordering strategies, usually computed by third-party numerical libraries like METIS [41] or AMD [42] and its variants, among others; all of them have been tested in the symbolic pre-processing stage of the simulation code, to select the best one for each simulation case. For that reason, the results obtained for the proposed test problem will be still valid for other multibody problems leading to different sparsity patterns.

With regard to matrix filling, the described formulation applied to the test problem of L loops leads to a tangent matrix in Eq. (5) of size $N = 2L+2$ with $12L+4$ structural non-zeros. A meaningful matrix filling indicator can be computed as the ratio between the number of non-zeros (NNZ) and N. In this case $NNZ/N \approx 6$, a typical value for global formulations applied to problems involving rigid bodies. Nevertheless, other dynamic formulations and multibody problems may lead to higher filling ratios, as depicted in Table 2.

Table 2: Typical matrix filling ratios in multibody dynamics (N = number of variables, NNZ = number of non-zeros).

Type of problem and dynamic formulation	*NNZ/N*
Rigid bodies - Global formulations	< 10
Rigid bodies – Topological formulations Flexible bodies - Component mode synthesis	10 – 30
Flexible bodies – Finite element mesh	30 – 100

Problems involving rigid bodies lead to higher fillings if topological or hybrid formulations are used [30]; the same filling range applies if the problem involves flexible bodies and they are described by component mode synthesis [29,31]. Finally, if flexibility is described by introducing the degrees of freedom of the finite element discretization in the multibody problem [32,33], the filling of the finite element stiffness matrix dominates the tangent matrix; in these cases, matrix filling ranges from 30 to 100, depending on the type of finite element (beam, shell, brick, etc.). In this study, the effect of matrix filling has been analyzed by introducing a variable number of artificial non-zeros in the sparsity pattern of the original tangent matrix.

Only minor changes were required in the initial sequential implementation to incorporate the three proposed parallel solvers, because they use the same storage format used by the three above mentioned sequential solvers already supported by the simulation code (Compressed Column Storage format or CCS). This format, also known as the Harwell-Boeing sparse matrix format, is quite common in direct sparse linear equation solvers. For solvers used in symmetric mode (CHOLMOD, WSMP, Pardiso), only the upper or lower triangular part of the tangent matrix is computed in Eq. (5), depending on the requirements of each solver; for non-symmetric solvers (KLU, SuperLU), the whole matrix is evaluated and factorized.

Some benchmarks for linear equation solvers [18,19] only measure factorization and solve (forward and back substitution) times. In this work, the total elapsed time for a multibody dynamic simulation was measured, since this procedure takes into account other important attributes like precision (more precise solvers will need less iterations in Eq. (4)) and memory footprint (its effect on the behavior of CPU-cache can affect the overall performance of the simulation).

2.5 Parallelization with OpenMP:

OpenMP [22] is a standard application programming interface to support multi-threaded parallel programming in shared-memory architectures. It provides a set of *directives* that can be added to a sequential program in Fortran, C, or C++ to describe, with minimal modifications in the code, how the work is to be distributed among multiple threads that run in parallel. A good description of OpenMP is provided by Chapman [23].

```
// Calls 2 functions in parallel
void example1()
{
     #pragma omp parallel sections

     #pragma omp section
     function1();
     #pragma omp section
     function2();
}

// '1'-norm of a vector in parallel
double example2(double v[], int n)
{
     double sum = 0;
     #pragma omp for reduction(+:sum)
     for (int i=0; i<n; i++) {
          sum = sum + v[i];
     }
     return sum;
}
```

Figure 2: Example of OpenMP directives for parallelization.

Fig. 2 shows a couple of examples of parallelization with OpenMP: the first one calls two code sections in parallel, while the second one splits a *for* loop into several non-overlapping fragments to traverse them in parallel and accumulate the results. These directives are understood by OpenMP compilers, which deal with the burden of working out the communication and synchronization details of the parallel program. The directives look like comments to regular, non OpenMP-aware compilers, which will generate sequential code. In this way, the same source code can be used in both sequential and parallel versions; this feature can simplify the maintenance of MBS simulation codes that are used to run simulations in both desktop PCs (suitable for parallel execution) and embedded microprocessors (which only support sequential execution) like automotive Electronic Control Units (ECUs).

Coarse-grained parallelization, in which large program regions are executed concurrently, can be easily achieved with OpenMP. An analysis of the profiling results in Table 1 and the sequence of calculations in Eqs. (2) to (6) evidences that two pairs of tasks (3-4 and 7-8) can be executed concurrently, as shown in Fig. 3. On the other hand, tasks 1 and 2 cannot be scheduled in parallel because the second one is inside the Newton-Raphson iteration of Eq. (4).

In addition, some of the optimizations implemented in the initial sequential version make not possible to apply fine-grained parallelization. For example, the Jacobian evaluation, which represents around 80% of Task 2, has been optimized for fast write operations to matrix data stored in CCS format [27]. This optimization reduced the evaluation time by a factor of 3 but it requires a sequential traversing of the involved *for* loop, which cannot be split like in Fig. 2.

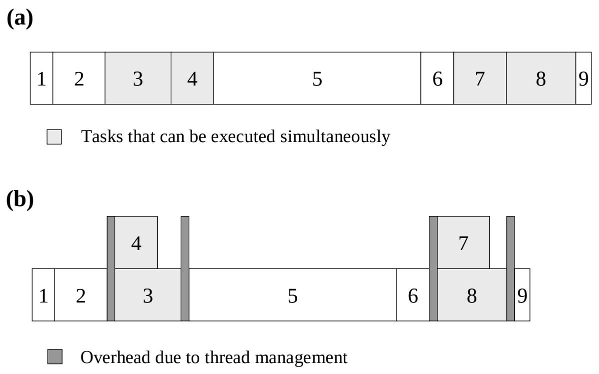
(a)
1
2
3
4
5
6
7
8
9
Tasks that can be executed simultaneously
(b)
4
7
1
2
3
5
6
8
9
Overhead due to thread management

Figure 3: Distribution of computational load in (a) the initial sequential version and (b) the proposed parallel version.

2.6 Test environment and implementation details:

Simulations have been run in a desktop PC with a dual-core Intel Core Duo E6300 CPU (1.86 GHz clock speed in each core, 64 Kb L1 cache, 2 Mb L2 cache) and 1 Gb of RAM, running Linux OS kernel 2.6.24 in 64 bit mode. Two compiler toolchains have been used: the GNU Compiler Collection (gcc version 4.3) and the Intel C/C++ Compiler (icc version 10.1); both of them support OpenMP.

A parallel computer with only two CPUs has been used because the tested dynamic formulation, heavily oriented to sequential execution, will deliver poor scalability since the fraction of parallelizable code will be relatively small. The goal of this work is to test whether the proposed non-intrusive parallelization methods can increase the efficiency of MBS dynamic simulations; if they can, the scalability of the speedups will greatly depend on the multibody problem and dynamic formulation.

3 RESULTS AND DISCUSSION

The following subsections present numerical results for the two abovementioned non-intrusive parallelization methods.

3.1 Multi-threaded linear equation solvers:

Fig. 4 shows the elapsed times for dynamic simulations with a number of variables N ranging from 100 to 4000 and three representative values of the matrix filling ratio according to Table 2 (NNZ/N = 6, 20, 50). Sequential single-threaded (st) solvers are represented in dashed lines, while parallel multi-threaded (mt) solvers are represented in solid lines.

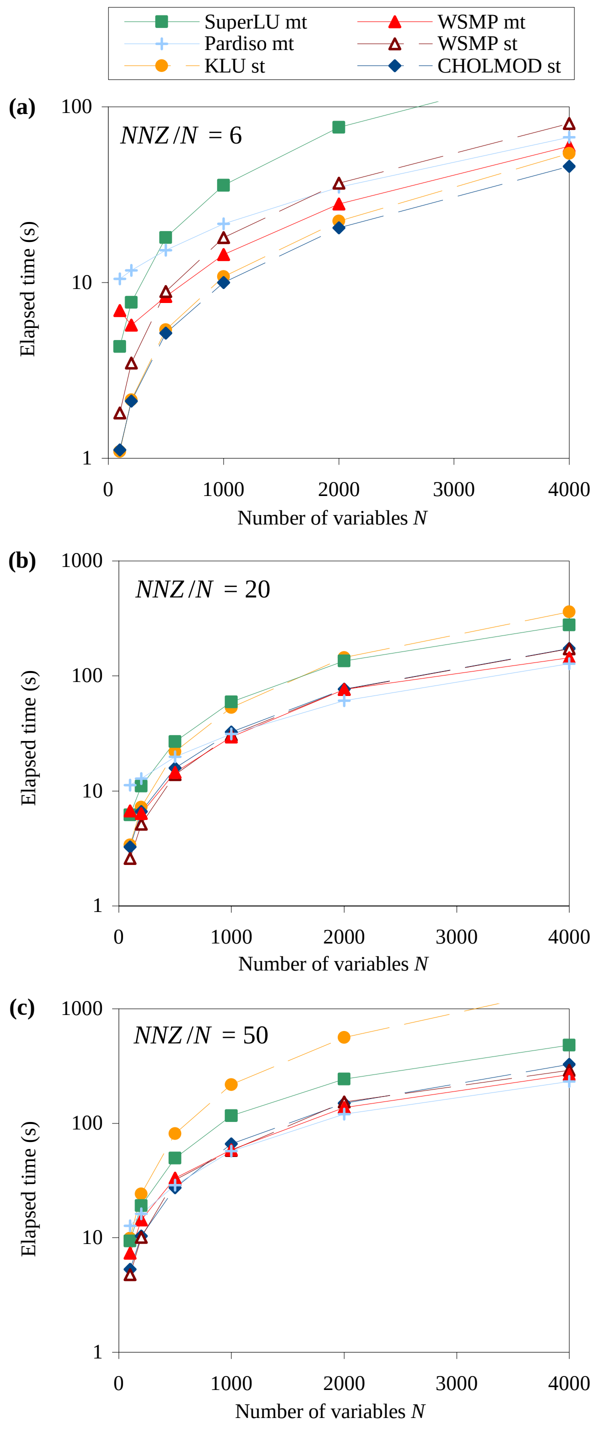

Figure 4: Performance of linear equation solvers as a function of problem size and matrix filling.

Elapsed times for N in the range 4000 – 8000 follow the tendencies indicated on the right side of the figures, so they have not been represented. Fig. 4a evidences that parallel solvers are not competitive for problems with low filling ratios: in these circumstances, KLU (unsymmetric solver) and CHOLMOD (symmetric positive definite solver) perform better than any other. The efficiency of KLU is outstanding in this case, taking into account that, due to its unsymmetric nature, the whole tangent matrix is evaluated and factorized during the simulation. The explanation for this excellent behaviour is that KLU is a sparse LU factorization algorithm specially designed for use in circuit simulation problems, which have a typical filling ratio NNZ/N of 7-8; however, this feature is also an important penalty for filling ratios above 10. For medium (Fig. 4b) and high (Fig. 4c) filling ratios, Pardiso emerges as the best solver for problems of large size. For medium size problems, CHOLMOD continues to be the most efficient solver under these circumstances.

In order to gain insight into the most favourable conditions for each solver, numerical experiments similar to those represented in Fig. 4 have been run with a matrix filling ratio within a range from 6 to 100.

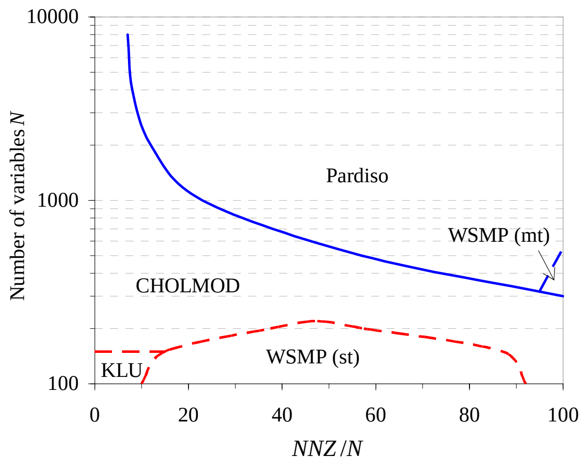

Figure 5: Best solver, as a function of problem size and matrix filling.

Results are synthesized in Fig. 5, which represents the regions where each solver delivers the best performance, as a function of the number of variables N and the filling ratio NNZ/N. The blue solid line draws up the boundary between the parallel and the sequential solvers, and the red dashed lines draw up the boundary between different sequential solvers. This figure serves as a decision tool to identify which solver is best suited for a particular multibody simulation. Fig. 5 shows that, contrary to general beliefs, parallel solvers can increase simulation efficiency for a wide range of problems in MBS dynamics. Pardiso dominates the region of parallel solvers, since the multi-threaded version of WSMP is only better in a small, non-representative region. On the other hand, CHOLMOD dominates the region of sequential solvers, while KLU and single-threaded WSMP are only competitive for small problems under 200 variables; these last results fully agree with the recommendations given in [27] for problems under 100 variables.

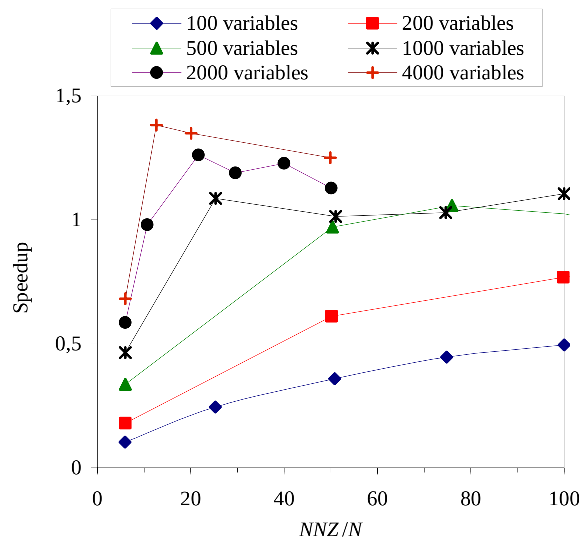

Figure 6: Speedup of Pardiso compared with the best sequential solver.

Since Pardiso has been demonstrated to perform better than sequential solvers for many multibody problems, it is important to quantify the speedups that it can deliver. Fig. 6 represents the speedups achieved by Pardiso, as a function of the filling ratio *NNZ*/*N* and the number of variables *N*; the speedup *S* is relative to the best sequential solver in each point of the figure:

$$S = \frac{elapsed\ time_{sequential}}{elapsed\ time_{parallel}} \qquad (7)$$

Table 3 shows the maximum speedup that can be achieved by a parallel solver in the tested implementation, for three typical values of the filling ratio; the values have been obtained from profiling results and Amdahl's law: for a program with a parallel fraction *f* running on *P* processors, the maximum speedup is:

$$S(P)_{\max} = \frac{1}{f/P + 1 - f} \qquad (8)$$

Table 3: Maximum speedup for 2 processors due to parallelization of the linear equation solver in the tested implementation, as a function of the matrix filling ratio NNZ/N.

NNZ/*N*	Factorizations and backsubstitutions	Max. speedup
6	52%	1.35
20	69%	1.53
50	68%	1.52

The information given in Fig. 6 and Table 3 is important in order to correctly interpret the results in Fig. 5. While Pardiso performs better for *N* < 1000 in a significant region of Fig. 5, the delivered speedups are very small compared with the best sequential solver (CHOLMOD), specially for *NNZ*/*N* > 50. Pardiso only delivers significant speedups for *N* > 1000, and it achieves the maximum performance for *NNZ*/*N* in the range from 10 to 30. In some cases, the speedups exceed 70% of the maximum values predicted by Amdahl's law in Table 3.

With regard to the effect of the compiler toolchain on the simulation efficiency, it has been observed that the two tested toolchains (GNU and Intel) can increase or decrease the elapsed times for the tested solvers in a factor up to 34%, depending on matrix size and filling ratio. However, in the conditions where each solver performs better (according to Fig. 5) the effect of the compiler is diminished, as shown in Table 4. In general, icc gives slightly better results than gcc, specially for Pardiso.

Table 4: Effect of compiler toolchain on the efficiency of linear equation solvers in the region where each solver performs best.

Linear equation solver	Best compiler	Min. gain	Max. gain
Pardiso	icc	7%	18%
Cholmod	icc	1%	8%
WSMP (st)	icc/gcc	-2%	2%
KLU	icc	-1%	7%

3.2 OpenMP:

Fig. 7 shows the elapsed times for dynamic simulations with the OpenMP parallel version of the code, for a number of variables ranging from 100 to 8000 and a filling ratio *NNZ*/*N* ≈ 6 (no artificial non-zeros were added to the tangent matrix).

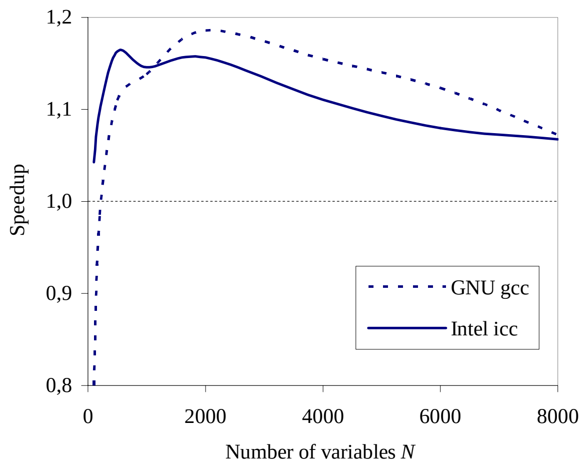

Figure 7: Speedup of the OpenMP parallel implementation.

The simulations have been run using CHOLMOD as linear equation solver. Since most of the burden of OpenMP parallelization is carried out by the compiler, results for both compiler toolchains (GNU and Intel) have been represented.

Taking into account the profiling data in Table 1, the task schedule shown in Fig. 3b can deliver a maximum speedup of 1.20. Results indicate that the compiler has a significant effect on the performance of the OpenMP parallel version. Intel OpenMP implementation, with a lower communication overhead, delivers speedups greater than one even for small problems of 100 variables, and it achieves the optimum conditions for around 500 variables. The GNU implementation needs more than 200 variables to become advantageous, and delivers the maximum values for 2000 variables; however, the speedups of GNU are higher, reaching the 95% of the maximum theoretical value (1.20).

Fig. 7 also shows that OpenMP speedups start to fall for *N* > 2000. This fact does not agree with the normal behaviour of parallel programs: both the communication overhead due to parallelization and the maximum speedup do not depend on *N* and should be constant (the overhead of thread creation and destruction depends only on the number of threads, and Table 1 demonstrates that the relative elapsed times of the parallelized tasks do not depend on *N*). Therefore, the maximum speedup $S_{max} = 1.20$ should be a horizontal asymptote for the curve $S(N)$, as it happens in MPI parallel codes [15]. This weird behaviour may be produced by adverse effects in the cache memory, because tasks scheduled in parallel in Fig 3b share part of the data: both tasks 3 and 4 operate with the mass matrix, Jacobian matrix and constraint vector, and both tasks 7 and 8 operate with the tangent matrix factorization and other common terms. Since each CPU has its own private cache, common data terms must be transferred twice from memory to cache, and for large problem sizes the memory bandwidth becomes a bottleneck [23]. This phenomenon is not frequent in MPI parallel implementations, since MPI processes operate in private, unshared data. Adverse effects of cache can be diminished with a proper allocation and distribution of data, as explained in [23]. However, these techniques can enforce major changes in existing sequential MBS simulation codes, and their effect highly depends on the computer architecture and the compiler toolchain; therefore, they cannot be considered as non-intrusive or easy to implement.

Nevertheless, results demonstrate that OpenMP is advantageous even for small problems and that it can deliver speedups above 80% of the maximum theoretical value for a wide range of problem sizes (from 500 to 3500 variables), provided the appropriate compiler toolchain is selected. Given the simplicity of its application to sequential codes, it is a valuable tool for non-intrusive parallelization of existing MBS simulation packages. The attainable speedups depend on the simulation characteristics; for example, Lugrís [43] describes two formulations for flexible multibody dynamics that spend up to 82% of the elapsed time in computing matrix terms associated with flexible bodies; the parallelization of this task with OpenMP would be straightforward, and speedups above 2 could be easily achieved on a quad-core processor. Problems with very time-consuming force evaluations (e.g. collision forces between complex geometries) can also achieve high improvements due to OpenMP parallelization.

4 CONCLUSIONS

Two non-intrusive parallelization techniques, parallel linear equation solvers and OpenMP, have been used to parallelize a starting sequential implementation of an MBS dynamic simulation software, in order to investigate their efficiency and suitability in the field of multibody dynamics. Both techniques are usually considered not appropriate for MBS dynamics due to the small sizes of matrix computations involved in this field.

Regarding the efficiency and suitability of parallel sparse linear equation solvers, the following conclusions can be established:

- Parallel solvers are advantageous for two types of multibody problems: (a) problems with more than 2000 variables leading to matrix filling ratios *NNZ/N* from 10 to 30 (the case for rigid multibody problems with topological formulations or flexible multibody body problems solved by component mode synthesis), and (b) problems with more than 2000 variables leading to matrix filling ratios *NNZ/N* above 30 (the case for flexible multibody body problems solved by introducing the finite element discretization in the formulation). Out of these two regions, sequential solvers (especially CHOLMOD) are more efficient.
- Pardiso is the most efficient parallel solver in the abovementioned conditions among the three tested parallel linear equation solvers (SuperLU, Pardiso and WSMP).
- The speedups delivered by Pardiso in the abovementioned conditions exceed 70% of the maximum theoretical value predicted by Amdahl’s law for matrix filling ratios in the range from 10 to 30. Beyond that point, speedups fall gradually. In addition, the speedups get higher as the problems increase their size.

Regarding the efficiency and suitability of the non-intrusive OpenMP parallel programming model, the following conclusions can be established:

- The parallelization of several tasks of an existing sequential dynamic simulation software was very easy to implement with OpenMP.
- The OpenMP parallel version proved to be advantageous even for small problems of 100 variables, and the speedups exceeded 80% of the maximum theoretical value predicted by Amdahl’s law for problem sizes in the range from 500 to 3500 variables.
- Beyond a certain problem size (2000 variables), the speedups fall gradually. This abnormal behaviour could be caused by adverse effects in the CPU’s cache memories.
- The compiler toolchain has a significant effect on the efficiency of OpenMP: Intel icc performs better for problems of less than 1000 variables, while GNU gcc performs better for larger problems.

Although both parallelization techniques cannot deliver high absolute speedups due to their non-intrusive character, their application is straightforward and therefore they are very appropriate to achieve partial parallelization of existing sequential multibody simulation codes with minimal effort. In addition, the good performance and ease-of-use of OpenMP suggests that it could be a substitute of MPI in the development and implementation of new formulations specially targeted to parallel execution; this topic represents an open line for future work.

ACKNOWLEDGMENTS

This research has been sponsored by the Spanish MEC (grant No. DPI2006-15613-C03-01 and the F.P.U. Ph.D. fellowship No. AP2005-4448).

REFERENCES

[1] Kasahara, H., Fujii, H., and Iwata, M., 1987, "Parallel Processing of Robot Motion Simulation," Munich, Germany.

[2] Eichberger, A., Fuhrer, C., and Schwertassek, R., 1993, "The Benefits of Parallel Multibody Simulation and Its Application to Vehicle Dynamics," Advanced multibody system dynamics: simulation and software tools, pp. 107-126.

[3] Fisette, P., and Peterkenne, J. M., 1998, "Contribution to Parallel and Vector Computation in Multibody Dynamics," Parallel Computing, 24(5-6), pp. 717-728.

[4] Cuadrado, J., Cardenal, J., Morer, P., and Bayo, E., 2000, "Intelligent Simulation of Multibody Dynamics: Space-State and Descriptor Methods in Sequential and Parallel Computing Environments," Multibody System Dynamics, 4(1), pp. 55-73.

[5] Critchley, J., and Anderson, K. S., 2003, "On Parallel Methods of Multibody Dynamics," ASME 2003 Design Engineering Technical Conferences (DECT '03), Chicago, Illinois (USA).

[6] Critchley, J. H., and Anderson, K. S., 2004, "A Parallel Logarithmic Order Algorithm for General Multibody System Dynamics," Multibody System Dynamics, 12(1), pp. 75-93.

[7] Anderson, K. S., and Oghbaei, M., 2005, "A State-Time Formulation for Dynamic Systems Simulation Using Massively Parallel Computing Resources," Nonlinear Dynamics, 39(3), pp. 305-318.

[8] Anderson, K., Mukherjee, R., Critchley, J., Ziegler, J., and Lipton, S., 2007, "POEMS: Parallelizable Open-Source Efficient Multibody Software," Engineering with Computers, 23(1), pp. 11-23.

[9] Bae, D. S., Kuhl, J. G., and Haug, E. J., 1988, "A Recursive Formulation for Constrained Mechanical System Dynamics .3. Parallel Processor Implementation," Mechanics of Structures and Machines, 16(2), pp. 249-269.

[10] Hwang, R. S., Bae, D. S., Kuhl, J. G., and Haug, E. J., 1990, "Parallel Processing for Real-Time Dynamic System Simulation," Journal of Mechanical Design, 112(4), pp. 520-528.

[11] Avello, A., Jimenez, J. M., Bayo, E., and Dejalon, J. G., 1993, "A Simple and Highly Parallelizable Method for Real-Time Dynamic Simulation-Based on Velocity Transformations," Computer Methods in Applied Mechanics and Engineering, 107(3), pp. 313-339.

[12] Mukherjee, R. M., Anderson, K. S., and Ziegler, J., 2005, "Multigranular Molecular Dynamics Simulations of Polymer Melts Using Multibody Algorithms," Long Beach, California, USA.

[13] Quaranta, G., Masarati, P., and Mantegazza, P., 2002, "Multibody Analysis of Controlled Aeroelastic Systems on Parallel Computers," Multibody System Dynamics, 8(1), pp. 71-102.

[14] Argonne National Laboratory, 2008, "MPI," http://www-unix.mcs.anl.gov/mpi/.

[15] Anderson, K. S., and Duan, S., 1999, "A Hybrid Parallelizable Low-Order Algorithm for Dynamics of Multi-Rigid-Body Systems: Part I, Chain Systems," Mathematical and Computer Modelling, 30(9-10), pp. 193-215.

[16] Gorder, P. F., 2007, "Multicore Processors for Science and Engineering," Computing in Science & Engineering, 9(2), pp. 3-7.

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

[18] Gupta, A., 2002, "Recent Advances in Direct Methods for Solving Unsymmetric Sparse Systems of Linear Equations," ACM Transactions on Mathematical Software, 28(3), pp. 301-324.

[19] Gupta, A., 2007, "A Shared- and Distributed-Memory Parallel General Sparse Direct Solver," Applicable Algebra in Engineering Communication and Computing, 18(3), pp. 263-277.

[20] Tracy, F. T., Oppe, T. C., and Gavali, S., 2007, "Testing Parallel Linear Iterative Solvers for Finite Element Groundwater Flow Problems," Pittsburgh, PA, USA.

[21] Davis, R. L., Henz, B. J., and Shires, D. R., 2003, "Performance Evaluation of Parallel Sparse Linear Equation Solvers for Positive Definite Systems," Las Vegas, Nevada, USA.

[22] OpenMP Architecture Review Board, 2008, "OpenMP," http://openmp.org.

[23] Chapman, B., Jost, G., van der Pas, R., and Kuck, D. J., 2007, Using OpenMP: Portable Shared Memory Parallel Programming, The MIT Press.

[24] Krawezik, G., and Cappello, F., 2006, "Performance Comparison of MPI and OpenMP on Shared Memory Multiprocessors," Concurrency and Computation-Practice & Experience, 18(1), pp. 29-61.

[25] Anderson, K. S., and Critchley, J. H., 2003, "Improved 'Order-N' Performance Algorithm for the Simulation of Constrained Multi-Rigid-Body Dynamic Systems," Multibody System Dynamics, 9(2), pp. 185-212.
[26] González, M., Dopico, D., Lugrís, U., and Cuadrado, J., 2006, "A Benchmarking System for MBS Simulation Software: Problem Standardization and Performance Measurement," Multibody System Dynamics, 16(2), pp. 179-190.
[27] González, M., González, F., Dopico, D., and Luaces, A., 2008, "On the Effect of Linear Algebra Implementations in Real-Time Multibody System Dynamics," Computational Mechanics, 41(4), pp. 607-615.
[28] Bayo, E., and Ledesma, R., 1996, "Augmented Lagrangian and Mass-Orthogonal Projection Methods for Constrained Multibody Dynamics," Nonlinear Dynamics, 9(1-2), pp. 113-130.
[29] Cuadrado, J., Gutierrez, R., Naya, M. A., and 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(9), pp. 1033-1052.
[30] Cuadrado, J., Dopico, D., González, M., and Naya, M., 2004, "A Combined Penalty and Recursive Real-Time Formulation for Multibody Dynamics," Journal of Mechanical Design, 126(4), pp. 602-608.
[31] Ambrosio, J. A. C., and Goncalves, J. P. C., 2001, "Complex Flexible Multibody Systems With Application to Vehicle Dynamics," Multibody System Dynamics, 6(2), pp. 163-182.
[32] García Orden, J. C., and Goicolea, J. M., 2000, "Conserving Properties in Constrained Dynamics of Flexible Multibody Systems," Multibody System Dynamics, 4(3), pp. 225-244.
[33] Sugiyama, H., and Shabana, A. A., 2004, "Application of Plasticity Theory and Absolute Nodal Coordinate Formulation to Flexible Multibody System Dynamics," Journal of Mechanical Design, 126(3), pp. 478-487.
[34] NIST, 2006, "Basic Linear Algebra Subprograms," http://www.netlib.org/blas/.
[35] Chen, Y., Davis, T. A., Hager, W. W., and Rajamanickam, S., 2006, "Algorithm 8xx: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate," http://www.cise.ufl.edu/~davis/techreports/cholmod/tr06-005.pdf.
[36] Davis, T. A., and Stanley, K., 2004, "KLU: a Clark Kent Sparse LU Factorization Algorithm for Circuit Matrices," http://www.cise.ufl.edu/~davis/techreports/KLU/pp04.pdf
[37] Gupta, A., Joshi, M., and Kumar, V., 1998, "WSSMP: A High-Performance Serial and Parallel Symmetric Sparse Linear Solver," Applied Parallel Computing, 1541, pp. 182-194.
[38] Saad, Y., 2003, Iterative Methods for Sparse Linear Systems, SIAM, Philadelphia.
[39] Demmel, J. W., Gilbert, J. R., and Li, X. Y. S., 1999, "An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination," Siam Journal on Matrix Analysis and Applications, 20(4), pp. 915-952.
[40] Schenk, O., Gartner, K., Fichtner, W., and Stricker, A., 2001, "PARDISO: a High-Performance Serial and Parallel Sparse Linear Solver in Semiconductor Device Simulation," Future Generation Computer Systems, 18(1), pp. 69-78.
[41] Karypis, G., and Kumar, V., 1999, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," Siam Journal on Scientific Computing, 20(1), pp. 359-392.
[42] Amestoy, P. R., Davis, T. A., and Duff, I. S., 1996, "An Approximate Minimum Degree Ordering Algorithm," Siam Journal on Matrix Analysis and Applications, 17(4), pp. 886-905.
[43] Lugris, U., Naya, M. A., González, F., and Cuadrado, J., 2007, "Performance and Application Criteria of Two Fast Formulations for Flexible Multibody Dynamics," Mechanics Based Design of Structures and Machines, 35, pp. 381-404.