MULTIBODY DYNAMICS 2005, ECCOMAS Thematic Conference
J.M. Goicolea, J. Cuadrado, J.C. García Orden (eds.)
Madrid, Spain, 21–24 June 2005

COMPUTATIONAL ASPECTS AND RECENT IMPROVEMENTS IN THE OPEN-SOURCE MULTIBODY ANALYSIS SOFTWARE “MBDYN”

Pierangelo Masarati[⋆], **Marco Morandini**[⋆], **Giuseppe Quaranta**[⋆], **and Paolo Mantegazza**[⋆]

[⋆]Dipartimento di Ingegneria Aerospaziale
Politecnico di Milano
Campus Bovisa, via La Masa 34, 20156 Milano, Italy
e-mail: pierangelo.masarati@polimi.it, marco.morandini@polimi.it, giuseppe.quaranta@polimi.it, paolo.mantegazza@polimi.it,
web page: http://www.aero.polimi.it/

Keywords: Computational Mechanics, Sparse Matrix Handling, Parallelization, Free Software.

Abstract. *This paper discusses some development directions recently followed with the aim of further improving the performances of the general-purpose multibody software MBDyn in terms of computational time. MBDyn is a free software that allows to simulate the dynamics of a broad class of mechanical and multidisciplinary problems, ranging from the real-time simulation of very small models, to the investigation of the dynamics of complex and large deformable aeroservoelastic systems. Very specific performance improvements may not be exploitable overall the entire range of applicability of the code; as a consequence, different strategies have been investigated, with the common aim of being of the broadest possible usability. The directions that gave the most interesting results were: sparse matrix handling, assembly and linear solution; incomplete and iterative nonlinear problem solution; parallelization of matrix assembly and factorization. The paper illustrates the rationale behind the investigation of these implementation issues, the results obtained so far with reference to the typical problems MBDyn is used for, and the directions of further investigation.*

1 INTRODUCTION

This paper discusses significant computational aspects of general-purpose multibody simulation and recent computational improvements of the free multibody analysis software MBDyn [7]. General-purpose multibody analysis software means that a single software tool is able to perform rather different types of analysis and to address rather different problems, as opposed to task specialization. Examples are: real-time hardware-in-the-loop simulation for robotics [1], large multidisciplinary interactional problems involving CFD aerodynamics [11], trajectory optimization [2], and more. As a consequence, it is worthless to address efficiency by specializing the software only for a given application; on the contrary, given the importance of software reuse and commonality of tools, efficiency must be pursued despite the inevitable trade-offs dictated by the flexibility of use.

Key aspects recently addressed in the enhancement of the MBDyn software are described in this paper. Among these, the implementation of different:

a. linear solution strategies

b. assembly strategies

c. nonlinear solution strategies

d. parallelization strategies

under a common object-oriented framework, allows to use the optimal combination of the above based on the properties of the problem under analysis, resulting in very efficient solution of highly demanding problems, like:

- real-time hardware-in-the-loop simulation, with scheduling and interprocess communication delegated to the real-time application interface (RTAI) fro the Linux OS; key issue is to minimize the worst-case solution for the single time step;
- path-planning and optimization in general; key issue is to minimize the overall solution time, and the need to interact with external modules for control (e.g. Scicos, Simulink and so on);
- "rapid prototyping", i.e. the capability to quickly implement and efficiently analyze incomplete problems, where configuration dependent forces are only modeled as residuals, thus losing the second-order convergence properties of direct solvers; the presented success story shows the effectiveness of matrix-free nonlinear solution algorithms when solving the tire-ground interaction problem of a landing aircraft [4].

The discussed features have been implemented with minimal impact on the structure of the code, resulting in outstanding solution efficiency improvements for all the different target problems.

The problems of interest can be split in three categories: dense (1 to 60 equations; typically best dealt with by dense matrices and solvers), very small (60 to 2000; fill-in $\leq$ 10%) and small (2000 to 20000; fill-in $\leq$ 5%). Much larger problems greatly benefit from classical sparse problem handling, but are definitely out of the scope of typical multibody analysis. The paper shows that while small problems ($\div$2000 equations) may benefit from dedicated assembly and matrix handling techniques, very small problems (200$\div$2000 equations) can lead to very efficient solution algorithms without losing the generality of the approach to the multibody formalism [9, 10].

```
NLS::Solve() {
    // Newton-Raphson
    while (true) {
        if (DM->Residual()->Test()) {
            return;
        }
        if (new_jacobian) {
            DM->Jacobian();
        }
        LS->Solve();
        DM->Update();
    }
}

SS::Advance() {
    SS->Predict();
    NLS->Solve();
}
```

Figure 1: Newton-Raphson nonlinear solver inner loop.

2 COMPUTATIONAL ISSUES AND SELECTED IMPROVEMENTS

The software MBDyn has been designed from the beginning in a modular manner, because one of its purposes is to allow independent researchers to investigate new solution approaches by designing and interchanging software components: the problem, by designing new elements; linear and nonlinear solvers, integration schemes, and more. The time integration of nonlinear problems is delegated to a sequence of layers consisting in:

- a “nonlinear problem data manager” (DM);
- a “nonlinear solver” (NLS), initially based on Newton-Raphson’s iterative procedure, but later extended to include iterative, matrix-free solution approaches;
- a “step solver” (SS), which takes care of advancing the solution step by step, predicting the new guess and controlling the convergence of the nonlinear problem solution;
- a “linear solver” (LS), which takes care of the linear algebra involved in the iterative solution of the nonlinear problem.

Figure 1 illustrates the main inner loop in pseudo-code when the Newton-Raphson nonlinear solver is used. During its evolution, the need to improve the performances in terms of problem size and complexity, and of solution time led to the development of different components for each layer; the different step solvers are not addressed in this work.

2.1 Linear Solution Strategies

From the beginning, an abstract layer for the solution of linear algebra problems has been designed. The reference problems required sparse linear solvers to efficiently handle the very sparse problems from a few hundreds to a few thousands of equations that typically result from models of deformable mechanical and multidisciplinary systems formulated with a *Redundant Coordinate Set* approach.

Different sparse solvers are currently supported. The “workhorse” is Umfpack, from the University of Florida (`http://www.cise.ufl.edu/research/sparse/umfpack/`), which is the standard sparse solver used by Matlab and other numerical packages.

```
typedef std::map<int, double> row_cont_type;
std::vector<row_cont_type> col_indices;

double& operator()(int i_row, int i_col)
{
    row_cont_type::iterator i;
    row_cont_type& row = col_indices[i_col];

    i = row.find(i_row);
    if (i == row.end()) {
        return row[i_row] = 0.;
    }
    return i->second;
}
```

Figure 2: Sparse map data structure for sparse matrices.

The need to address very specific problems, significantly very small ones (100÷200 equations) within the very tight scheduling required by real-time simulation forced into looking for as much efficiency as possible. For this purpose, support for the LAPACK dense LU solver has been added to the suite of linear solvers supported by MBDyn. In most of the applications, this outperforms Umfpack when addressing problems up to 100 equations with a fill-in of 25% and above.

Nonetheless, since the typical problem fill-in was limited enough to make sparse matrix handling appealing, and the need to address slightly larger problems (up to 200 equations) with the same stringent requirements, suggested the possibility to design a very specific solver, which combines the minimal access cost of dense matrices with the minimal computational cost of sparse matrix factorization and substitution [9]. This specialized solver outperforms all the linear solvers that were considered in most cases, based on the sparsity of the matrix and on the sparsity pattern, up to 1500÷2000 equations, as illustrated in [9, 10] and discussed later in the applications.

2.2 Assembly Strategies

The use of sparse matrices may dramatically reduce the factorization and substitution time when efficient linear solvers are used. However, they introduce a cost when the coefficients of the matrix are accessed to add or modify a coefficient during the problem assembly.

Sparse matrix handling in MBDyn, for use with Umfpack and other sparse solvers, is based on C++ STL containers. The matrix, called "sparse map", is designed as a vector of associative containers (maps) that contain the real-typed value with the related row index, as illustrated[1] in Figure 2. The sparse map matrix provides a method that packs the matrix in column-compressed form when it is passed to the sparse linear solver. The packing must be repeated before each matrix factorization.

An interesting evolution is represented by the "column compressed" sparse matrix handling. It consists in directly accessing the matrix in the column compressed form obtained prior to factorization, under the assumption that the shape of the matrix non-zeroes does not change between assemblies. The column-compressed form is made of a vector of reals containing the non-zeroes, a vector of integers containing the row indices of the non-zeroes, sorted throughout each column, and another vector of integers pointing to the beginning of the columns. A generic

[1]Note that the (pseudo-)code of Figure 2 actually fills the matrix of zeroes if, for example, the matrix is accessed by means of the operator () for read; a different operator must be designed for read operations.

```
    std::vector<double>& values;
    const std::vector<int>& row_indices;
    const std::vector<int>& column_start;

    double& operator()(int i_row, int i_col)
    {
        int row_begin = column_start[i_col - 1];
        int row_end = column_start[i_col] - 1;
        int idx;
        int row;

        if (OutOfRange(i_row)) {
            // out of range: rebuild
            throw ErrRebuildMatrix();
        }

        // binary search
        while (row_end >= row_begin) {
            idx = (row_begin + row_end)/2;
            row = row_indices[idx];
            if (i_row < row) {
                row_end = idx - 1;
            } else if (i_row > row) {
                row_begin = idx + 1;
            } else {
                return values[idx];
            }
        }

        // not found: rebuild
        throw ErrRebuildMatrix();
    }
```

Figure 3: Compressed-column data structure for sparse matrices.

non-zero can be accessed by directly accessing a column with the column index, and performing a binary search across it for the row index. If it is not present, the symbolic structure of the sparse matrix changed; this is handled by throwing an exception that causes the matrix to return into the sparse map form, and the assembly to restart.

In typical problems, the sparsity pattern of the matrix is usually preserved for most of the iterations; as a consequence, the use of the column-compressed sparse matrix allows to entirely save the packing phase, while the cost of the binary search required to access the generic coefficient is comparable to that of accessing the sparse map (it is exactly the same if the map is implemented with a binary tree).

The specialized structure of the “naïve” solver [9] represents yet another radical change in matrix structure. In this latter case, a dense matrix is used to store the values, while the row and column indices to non-zeroes are stacked in other two dense matrices. This approach is quite memory intensive, and quickly loses efficiency when the problem size grows not only because of physical memory constraints, but also, and essentially, because large memory buffers quickly destroy CPU cache locality.

2.3 Nonlinear Solution Strategies

The original solution method used by MBDyn is based on Newton-Raphson iterations. This method requires the factorization of the Jacobian matrix and, for large problems, it can be computationally intensive. In the current implementation, the modified Newton iteration reuses the same Jacobian factorization for a fixed number of iterations, and at least one new Jacobian is computed for each time step. This approach is recommended, for instance, when performing simulations in real-time, because a fresh Jacobian factorization is assumed to yield a faster convergence, and the cost of each time step is almost constant without the dramatic increase resulting when the worst case of a Jacobian re-computation and re-factorization occurs.

It is well known that the linear equation resulting from the Newton method can be solved in an approximate manner. The resulting methods are usually called *Inexact Newton* and see broad applications in computational mechanics and other fields [5]. Among them, iterative Newton methods represent a very interesting class, where basically the linear equation is solved for each step using an approximate iterative solution. Typically, the nonlinear iteration generates a sequence, called *outer iteration*, and the linear iteration that generates the approximation for each step, called *inner iteration*. Krylov methods are usually adopted as iterative inner solvers, like the Generalized Minimum RESidual (GMRES) or the BiCGStab [15]. They basically require, during each inner iteration, a simple multiplication of the Jacobian tangent matrix of the problem with a vector, plus the adoption of a preconditioner. The product of the Jacobian matrix $\boldsymbol{J}$ times a generic vector $\boldsymbol{w}$ can be approximated by a finite difference formula using the residual vector $\boldsymbol{r}$ as

$$\boldsymbol{J}(\boldsymbol{x})\boldsymbol{w} = \|\boldsymbol{w}\| \frac{\boldsymbol{r}\left(\boldsymbol{x} + h\boldsymbol{w}\dfrac{\|\boldsymbol{x}\|}{\|\boldsymbol{w}\|}\right) - \boldsymbol{r}(\boldsymbol{x})}{h\|\boldsymbol{x}\|}. \tag{1}$$

In this expression h is the amplitude of the finite difference step, which must be always as small as possible, although compatible with the sensitivity of the problem and the round-off errors. For this reason the selected step is usually scaled keeping into account the differences between the scale of the solution vector $\boldsymbol{x}$ and the test vector $\boldsymbol{w}$. This attenuates the need to have a precise approximation of the Jacobian matrix for the equations that are solved; it is only required to be able to assemble the residual vector $\boldsymbol{r}$ for perturbations of the state $\boldsymbol{x}$ in different testing directions. The resulting methods are denominated *matrix free* since no tangent matrix is required. The preconditioner is used to accelerate the convergence of the inner iterative solution. Ideally, if the preconditioner is the exact approximation of the inverse of the Jacobian matrix, the convergence will be reached in just one step; so it is necessary to build an approximation of this matrix, for example adopting the inverse of the matrix assembled using the elements for which the tangent matrix can be easily derived. To make the problem less computationally intensive, the preconditioner may be retained, until the number of inner iteration to converge grows past a specific threshold. Finally, the inner iteration does not need to converge with high accuracy, so the solution can be approached in a faster manner by relaxing the convergence criteria. The application of this method can be crucial crucial for all those complex elements for which the generation of the analytical tangent matrix is either extremely complex or impossible (see for example [4]).

2.4 Parallelization Strategies

A coarse-grained parallelization of the analysis can be obtained by partitioning the problem into subdomains that are solved separately on different CPUs, while the interface problem is

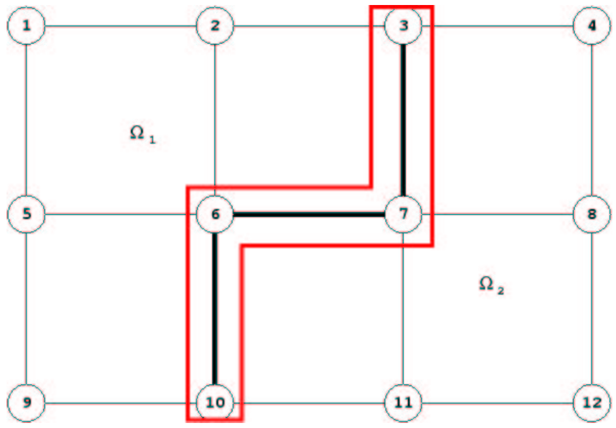

Figure 4: Element-based partition.

solved at the end, based on the contributions from each subproblem. It is apparent that a small interface is key to the effectiveness of this "divide et impera" approach, because the size of the interface problem governs the amount of data that needs be transmitted and, since it is performed after the solution of the subproblems, it represents a bottleneck and inevitably reduces to a dense problem, with cubic solution cost.

The computational domain Ω is first split into the s subdomains Ω_i by means of an element-based partition (Fig. 4). This means that no element must be split between two subdomains, i.e. all the information related to a given element is mapped to the same processor. As a result, there is no need for information exchange while the assembly phases are performed. Anyway, it is worth noting that while the elements can belong to one subdomain only, nodes may belong to multiple subdomains. By reordering the unknowns to label the interface nodes at the end, the linear system associated with the problem assumes the structure:

$$\begin{pmatrix} \boldsymbol{B}_1 & 0 & \cdots & 0 & \boldsymbol{E}_1 \\ 0 & \boldsymbol{B}_2 & & \vdots & \boldsymbol{E}_2 \\ \vdots & & \ddots & & \vdots \\ 0 & \cdots & & \boldsymbol{B}_s & \boldsymbol{E}_s \\ \boldsymbol{F}_1 & \boldsymbol{F}_2 & \cdots & \boldsymbol{F}_s & \boldsymbol{C} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \vdots \\ \boldsymbol{x}_s \\ \boldsymbol{y} \end{pmatrix} = \begin{pmatrix} \boldsymbol{f}_1 \\ \boldsymbol{f}_2 \\ \vdots \\ \boldsymbol{f}_s \\ \boldsymbol{g} \end{pmatrix}, \tag{2}$$

where the $\boldsymbol{B}_i$ are the local subdomain matrices, $\boldsymbol{C}$ is the interface matrix and $\boldsymbol{E}_i$, $\boldsymbol{F}_i$ are the coupling matrices. Each $\boldsymbol{x}_i$ is a vector containing the internal unknowns of the subdomain Ω_i, and $\boldsymbol{y}$ is the vector of the interface unknowns; $\boldsymbol{f}_i$ and $\boldsymbol{g}$ are the corresponding right-hand terms. In a more compact form the system can be expressed as

$$\begin{pmatrix} \boldsymbol{B} & \boldsymbol{E} \\ \boldsymbol{F} & \boldsymbol{C} \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{pmatrix} = \begin{pmatrix} \boldsymbol{f} \\ \boldsymbol{g} \end{pmatrix}. \tag{3}$$

Provided $\boldsymbol{B}$ is non-singular, the unknown $\boldsymbol{x}$ can be expressed as

$$\boldsymbol{x} = \boldsymbol{B}^{-1}(\boldsymbol{f} - \boldsymbol{E}\boldsymbol{y}). \tag{4}$$

By substituting Equation (4) into the second block-row of equation (3), the following reduced system is obtained:

$$\boldsymbol{S}\boldsymbol{y} = \boldsymbol{g} - \boldsymbol{F}\boldsymbol{B}^{-1}\boldsymbol{f}. \tag{5}$$

where the matrix $\boldsymbol{S}$ is called the *Schur complement matrix*; it assumes the form:

$$\boldsymbol{S} = \boldsymbol{C} - \boldsymbol{F}\boldsymbol{B}^{-1}\boldsymbol{E}. \tag{6}$$

After assembling all the elements that belong to subdomain Ω_i, a local sub-matrix results with the structure:

$$\begin{pmatrix} \boldsymbol{B}_i & \boldsymbol{E}_i \\ \boldsymbol{F}_i & \boldsymbol{C}_i \end{pmatrix}. \tag{7}$$

By calling $\boldsymbol{R}_i$ the restriction operator to the local interface values, so that $\boldsymbol{R}_i\boldsymbol{y} = \boldsymbol{y}_i$, the Schur matrix is obtained as

$$\boldsymbol{S} = \sum_{i=1}^{s} [\boldsymbol{R}_i\boldsymbol{C}_i\boldsymbol{R}_i^T - \boldsymbol{R}_i\boldsymbol{F}_i\boldsymbol{B}_i^{-1}\boldsymbol{E}_i\boldsymbol{R}_i^T] = \sum_{i=1}^{s} \boldsymbol{R}_i\boldsymbol{S}_i\boldsymbol{R}_i^T. \tag{8}$$

A solution method based on this approach involves five steps:

1. The local matrices $\boldsymbol{B}_i$ are factored, fully exploiting any natural sparsity.
2. The local parts of the right-hand side of the reduced system Equation (5) are assembled and transmitted to a “master” processor that will deal with the interface problem.
3. The local parts of the Schur complement matrix are assembled and transmitted as well.
4. The reduced system is solved.
5. The other unknowns are computed by the back-substitution shown in Equation (4).

Only the 4^{th} step cannot be performed concurrently in a parallel environment, so it must be considered the bottleneck phase. Furthermore, when a modified Newton-Raphson method is used, since the Jacobian matrix is not updated at each iteration, steps (1) and (3) must be performed only at the beginning of the iterative solution, and many of the operations required during the assembly phase (2) need to be performed only once as well.

Usually, the direct substructuring method described here is not considered feasible for large structural problems, because the size of the interface problem grows rapidly; moreover, the Schur matrix presents a lower grade of sparsity than the original system, so an iterative inner solver should be considered, which does not require the explicit assembly of matrix S. On the contrary, this strategy can be very effective when some special conditions are met. This is the case of complex systems with a peculiar topology of the computational domain that allows the generation of a partition with very small interfaces [3]. Many common mechanical problems show a topology that meets this requirement; a clear example is represented by the rotorcraft analysis models described later in the results section.

It is not easy to define an appropriate dimension for the computational domain that is analyzed with a multibody multidisciplinary simulator. The multidisciplinarity requirement obviously does not allow any structure in the domain, because structurable, i.e. physical space related, and non-structurable, i.e. abstract, unknowns coexist in the solution space. Anyway a typical problem can be usually thought as quasi-monodimensional, with some multiple paths or closed-circuits. This is basically true because the underlying structure is usually made of rigid bodies connected by algebraic constraints, which are the irreducible parts of the computational domain. As a consequence, the computational grid can be subdivided into parts with an optimal ratio between internal and interface nodes. Clearly, the search for a minimal-interface partition

```
for (int row = thr; row < N; row += Nthr) {
        for (int t = 1; t < Nthr; t++ ) {
                A[0][row] += A[t][row];
        }
}
```

Figure 5: Contribution of thread `thr` to parallel assembly of buffers.

is crucial, so this task cannot be performed manually; it is delegated to standard partitioning tools (for example, METIS).

Another parallelization approach has been explored in view of its application to real-time simulation. In essence, the availability of sparse matrix storage schemes that result in very compact mapping of the non-zeroes, like the previously discussed column-compressed form, suggested to address the parallelization of the assembly on SMP architectures by allowing multiple concurrent assembly threads to access the elements in a competitive manner, without any prior model partitioning, by way of a specifically designed concurrency-aware element iterator. Each thread assembles its portion of problem on its local compressed buffer, accessing in read-only mode the common structure of the indices as described in Figure 3. At the end, the threads sum their buffers in a single storage. This operation can be performed concurrently as well, because the buffers exactly map on the same coefficients, so, for instance, each thread may safely add a specific portion of the coefficients of all buffers, as illustrated in Figure 5.

This approach relies on the fact that assembly may account for up to 30% of the execution time, especially when multiple connections exist between a relatively limited set of nodes. This is the case, for instance, of aeroelastic problems, where elastic, inertial and aerodynamic elements contribute to the same equations and some of these elements may require intensive operations (e.g. table lookups for aerodynamic forces). Unfortunately the parallel assembly did not bring all the expected advantages, because the impact of assembly on the overall simulation cost reduced dramatically when the column-compressed form, which is essential for the multi-thread assembly, was introduced. Only very limited additional speedups were obtained, and the overhead related to task scheduling makes the parallel assembly ineffective or even adverse for very small models, so it is not applicable to real-time simulation.

Among the directions for future development, there remain few combinations of the above described improvements that are currently unsupported. Some of them may be of interest; for instance, the possibility to exploit multithreaded parallelization of assembly on SMP machines in conjunction with Schur parallelization on remote machines in a cluster may allow to exploit both gains simultaneously when addressing very large problems.

3 APPLICATIONS

3.1 Landing Gear Simulation

The problem addresses the simulation of the gear-walk phenomenon, as described in [4]: an instability related to the interaction of the Automatic Braking System (ABS) with the deformability of the landing gear.

The modeling of the forces that the tire exchanges with the ground represents a crucial point in this type of problems. The tire and shock absorber models implemented in MBDyn were inherited from an already validated software program, but they did not provide any Jacobin, and some of the empirical functions they was based on could not be easily differentiated analytically. Furthermore, during the landing phase, the wheel, the shock absorber and the gear deformable structure are subject to very high, impulsive loads, and the wheel and the shock absorber showed

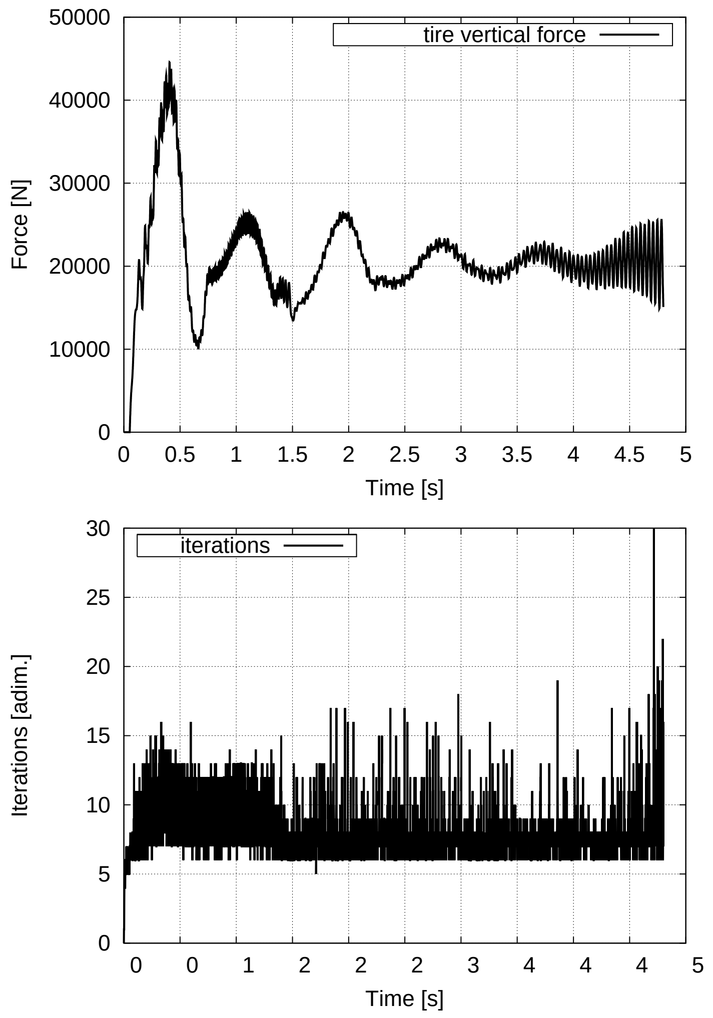

Figure 6: Tire normal force and iterations during landing and braking maneuver.

a very high stiffness that increased with gear compression. As a result, the default Newton-Raphson solver is not able to successfully integrate the equations of motion, even with the smallest time step that would otherwise be reasonable for the problem (1e-3÷1e-4 s).

The iterative matrix-free solver proved to be a valid alternative to the numerical differentiation of that specific user-defined elements, and allowed to efficiently simulate the whole landing and braking phases with fairly reasonable time steps. Figure 6 shows the iterations count as opposed to the tire normal force during a landing impact followed by braking in a manner that initiates the gear walk instability phenomenon. Note that the number of iterations remains limited and fairly independent from the changes in ground forces. The time step is 0.5e-3 s.

3.2 Real-Time Simulation

The real-time simulation of robots and rotorcraft wind-tunnel models has been addressed by means of MBDyn [1, 6, 10]. In all cases, the modeling needs could not be satisfied by models with less than 120–150 equations; thus the performances of the analysis were essentially dictated by the need to find an acceptable trade-off between the highest possible sample rate, requested by accuracy needs (e.g. a minimal number of steps per rotor revolution) or by interaction requirements with other tasks participating in the simulation (e.g. the controller process).

Table 1: Rotorcraft dynamics analysis: 890 equations (coarse model, realistic)

Feature	Time
Baseline	1.00
Column-compressed (CC)	0.81
Assembly parallelization + CC (2 CPU)	0.78
Naïve solver	0.71

Actually, the needs of the real-time simulation were the real drivers of many of the performance-oriented speculations described in this paper.

One of the main points of speculation was the capability to exploit Symmetric Multi Processor (SMP) architectures to split the easily parallelizable portions of the solution into multiple threads of execution. Unfortunately, no appreciable success was obtained by spreading the Jacobian and residual assembly, or by parallelizing the factorization and the back substitution with the "naïve" solver described in [9]. The most significant improvements, apart from an overall "squeezing" of the code, came from the "naïve" solver in scalar form.

3.3 Rotorcraft Simulation

The parallelization of Jacobian and residual assembly partially gave the expected improvements when addressing much larger models. For rotorcraft dynamics analysis, typical models [8, 14, 13] consisting in deformable rotor blades, detailed rotor hub and control system kinematics and, for tiltrotors, deformable wing dynamics, result in 600-2000 equations, with hundreds of beam, inertia and aerodynamics elements, and dozens of joints. The problem is very sparse, so the solution by means of efficient sparse solvers, ranging from Umfpack to the built-in "naïve" solver, result in very fast analysis. Nonetheless, the factorization and back substitution, namely the linear solution, still accounts for a vast majority of the analysis time, while assembly reduces to at most 30% of the overall solution time, despite the high elements versus equations ratio.

The use of the column-compressed form of the sparse matrix storage showed dramatic reductions in computational time; this indicates that the cost of creating the sparse maps and then packing them into the form required by the solver accounts for a significant amount of the simulation time.

The parallelization of the assembly phase, on dual Athlon CPU systems, showed further reductions in the solution time of about 3÷4%, thus indicating that the actual cost of the elements assembly lies between 6 and 10% of the overall computational time; this is consistent with results from standard profiling software.

Tables 1–3 show some figures about computational time reductions for the tiltrotor model described in [13]. The model of the rotor is illustrated in Figure 7; in the two cases, it consists in respectively about 900 and 1500 equations that describe the dynamics of the deformable rotor of a tiltrotor aircraft, supported by a deformable wing. The integration occurs with a fixed time step of 2.5e-4 that is required to capture the dynamics of the blades with the desired accuracy.

As a comparison, the performances related to two benchmarks, respectively made of about 1450 and 2900 equations, representing a string of beams clamped at one end and excited by an impulsive force at the other end, are illustrated in Tables 4 and 5. The assembly is required only every 5 residuals; as each step converges in 2–3 iterations, only one Jacobian per time step is assembled. The figures above change dramatically if the Jacobian is re-factored during each iteration, as shown in Table 6; note that in that case, the reference time is slightly more

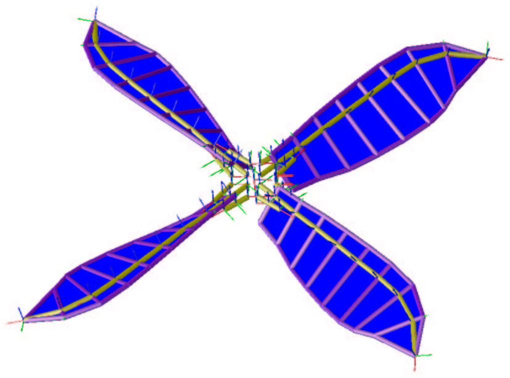

Figure 7: Deformable rotor model.

Table 2: Rotorcraft dynamics analysis: 1455 equations (refined model, realistic)

Feature	Time
Baseline	1.00
Column-compressed (CC)	0.83
Assembly parallelization + CC (2 CPU)	0.80
Naïve solver	0.78

Table 3: Rotorcraft dynamics analysis: 2415 equations (over-refined model, unrealistic)

Feature	Time
Baseline	1.00
Column-compressed (CC)	0.83
Assembly parallelization + CC (2 CPU)	0.79
Naïve solver	0.75

Table 4: Structural dynamics of beam benchmark: 1452 equations; modified Newton

Feature	Time
Baseline	1.00
Column-compressed (CC)	0.80
Assembly parallelization + CC (2 CPU)	0.77
Naïve solver	0.66
Solution parallelization (Schur; 2 CPU)	0.58

Table 5: Structural dynamics of beam benchmark: 2892 equations; modified Newton

Feature	Time
Baseline	1.00
Column-compressed (CC)	0.82
Assembly parallelization + CC (2 CPU)	0.78
Naïve solver	0.69
Solution parallelization (Schur; 2 CPU)	0.61

Table 6: Structural dynamics of beam benchmark: 1452 equations; full Newton

Feature	Time
Baseline	1.00
Column-compressed (CC)	0.77
Assembly parallelization + CC (2 CPU)	0.77
Naïve solver	0.50
Solution parallelization (Schur; 2 CPU)	0.53

than 2.5 times that of Table 4. The advantages of the parallel assembly are essentially eaten up by the time spent in factorization; on the contrary, the enormous savings in creating and packing the sparse matrix are preserved almost untouched. The “naïve” solver magnifies its performances, thanks to its great factoring efficiency, while the Schur solver does not improve as well, because the transmissions related to Jacobian assembly and partial factorization impact on the performance reductions.

This benchmark may not be considered representative of a realistic analysis for many reasons. First of all, it appears that the “naïve” solver performs better than Umfpack with an even larger problem; however, realistic applications show the opposite. Also, the Schur parallelization appears to behave quite well, since it gives a quasi-linear scaling; this behavior is not confirmed by realistic applications, where the interface is larger (it is exactly 1 node in the above example) and the partitioning is not exactly symmetric. Note that the above results have been obtained using Umfpack as local subproblem linear solver and LAPACK as interface linear solver. Better performances are expected when using the “naïve” solver for the subproblems, but there appear to be interaction issues that prevent the two from working together. Unfortunately, no significant scalings appear for the rotor essentially because no optimal partitioning on two CPUs can be determined. However, earlier analyses on a 8 CPU SMP architecture showed interesting performances also on realistic cases, as discussed in [12]; note however that the optimistic results obtained in that work were partially related to less than ideal scalar performances. However, the above figures should give an idea of the capabilities brought in by all the approaches considered in order to speed-up the analysis of models that, although “small” from the point of view of linear solution, can be considered “large” from the point of view of multibody analysis.

4 CONCLUSIONS

The paper discussed some of the strategies recently applied to the multibody software MBDyn to further reduce its execution time. Generally beneficial improvements have been sought, although mainly focusing on the most promising for the types of problems commonly addressed by this software.

Apart from the dramatic improvements obtained by designing a specialized linear solver for small and very small sparse systems, described in a companion paper, significant improvements have been obtained by addressing the assembly of sparse matrices, which is now performed in parallel on SMP architectures, and reusing as much as possible the sparsity patterns across different assembly iterations and time steps.

The introduction of inexact, iterative matrix-free solvers allowed to efficiently solve very stiff problems with incomplete Jacobian matrix.

A coarse-grained topological parallelization scheme for the solution of the nonlinear problem has been revitalized for the very specific class of almost tree-like problems that are typical of rotorcraft dynamics simulation.

REFERENCES

[1] M. Attolico and P. Masarati. A multibody user-space hard real-time environment for the simulation of space robots. In *Fifth Real-Time Linux Workshop*, Valencia, Spain, November 9–11 2003.

[2] M. Attolico, P. Masarati, and P. Mantegazza. Trajectory optimization and real-time simulation for robotics applications. In *Multibody Dynamics 2005, ECCOMAS Thematic Conference*, Madrid, Spain, June 21–24 2005.

[3] P. E. Bjørstad and A. Hvidsten. Iterative methods for substructured elasticity problems in structural analysis. In G. M. R. Glowinski, G. Golub and S. J. Périaux eds., editors, *Proceedings of the first international symposium on Domain Decomposition Methods for Partial Differential Equations*, Philadelphia, PA, 1988.

[4] S. Gualdi, M. Morandini, P. Masarati, and G. L. Ghiringhelli. Numerical simulation of gear walk instability in an aircraft landing gear. In *CEAS Intl. Forum on Aeroelasticity and Structural Dynamics 2005*, Muenchen, Germany, June 28–July 1 2005.

[5] C. Kelley. *Iterative methods for linear and nonlinear equations*. SIAM, Philadelphia, PA, 1995.

[6] P. Masarati, M. Attolico, M. W. Nixon, and P. Mantegazza. Real-time multibody analysis of wind-tunnel rotorcraft models for virtual experiment purposes. In *AHS* 4th *Decennial Specialists' Conference on Aeromechanics*, Fisherman's Wharf, San Francisco, CA, January 21–23 2004.

[7] P. Masarati, M. Morandini, G. Quaranta, and P. Mantegazza. Open-source multibody analysis software. In *Multibody Dynamics 2003, International Conference on Advances in Computational Multibody Dynamics*, Lisboa, Portugal, July 1–4 2003.

[8] P. Masarati, D. J. Piatak, G. Quaranta, and J. D. Singleton. Further results of soft-inplane tiltrotor aeromechanics investigation using two multibody analyses. In *American Helicopter Society* 60th *Annual Forum*, Baltimore, MD, June 7–10 2004.

[9] M. Morandini and P. Mantegazza. Using dense storage to solve small sparse linear systems. Submitted to *ACM Transactions on Mathematical Software* (ACM TOMS).

[10] M. Morandini, P. Masarati, and P. Mantegazza. A real-time hardware-in-the-loop simulator for robotics applications. In *Multibody Dynamics 2005, ECCOMAS Thematic Conference*, Madrid, Spain, June 21–24 2005.

[11] G. Quaranta, G. Bindolino, P. Masarati, and P. Mantegazza. Toward a computational framework for rotorcraft multi-physics analysis: Adding computational aerodynamics to multibody rotor models. In 30th *European Rotorcraft Forum*, pages 18.1–14, Marseille, France, 14–16 September 2004.

[12] G. Quaranta, P. Masarati, and P. Mantegazza. Multibody analysis of controlled aeroelastic systems on parallel computers. *Multibody System Dynamics*, 8(1):71–102, 2002.

[13] G. Quaranta, P. Masarati, and P. Mantegazza. Dynamic characterization and stability of a large size multibody tiltrotor model by pod analysis. In *ASME 19*th *Biennial Conference on Mechanical Vibration and Noise (VIB)*, Chicago Il, September 2–6 2003.

[14] G. Quaranta, P. Masarati, and P. Mantegazza. Assessing the local stability of periodic motions for large multibody nonlinear systems using POD. *Journal of Sound and Vibration*, 271(3–5):1015–1038, 2004.

[15] Y. Saad. *Iterative methods for sparse linear systems*. PWS Publishing Company, Boston, MA, 1996.