Asynchronous Preconditioners for Efficient Solving of Non-linear Deformations

Hadrien Courtecuisse, Jérémie Allard, Christian Duriez, Stéphane Cotin

▶ To cite this version:

HAL Id: hal-00688865

https://hal.inria.fr/hal-00688865

Submitted on 18 Apr 2012

Asynchronous Preconditioners for Efficient Solving of Non-linear Deformations

Hadrien Courtecuisse   Jérémie Allard   Christian Duriez   Stéphane Cotin

Shaman Team, INRIA Lille, France

Abstract

In this paper, we present a set of methods to improve numerical solvers, as used in real-time non-linear deformable models based on implicit integration schemes. The proposed approach is particularly beneficial to simulate non-homogeneous objects or ill-conditioned problem at high frequency. The first contribution is to desynchronize the computation of a preconditioner from the simulation loop. We also exploit today's heterogeneous parallel architectures: the graphic processor performs the mechanical computations whereas the CPU produces efficient preconditioners for the simulation. Moreover, we propose to take advantage of a warping method to limit the divergence of the preconditioner over time. Finally, we validate our work with several preconditioners on different deformable models. In typical scenarios, our method improves significantly the performances of the perconditioned version of the conjugate gradient.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Physically based modeling; G.1.3 [Numerical Analysis]: Numerical Linear Algebra—Conditioning, Linear systems (direct and iterative methods)

1. Introduction

In the context of real-time or interactive simulations of deformable objects, only a few milliseconds are available to compute the next positions. A high update frequency is required to produce the sensation of interactivity. In applications involving a real-time constraint, it is even necessary that the computation time be at most the same as the time step of the integration scheme. Moreover, when the user is allowed to interact with the simulation, it is necessary to use robust algorithms. For these reasons, we rely in this paper on an implicit integration scheme, providing both guarantees of stability and support of large time steps. The main drawback of implicit schemes however is the requirement to solve large systems of equations. Additionally, the modeling of deformable structures introduces non-linear behaviors both in the strain tensor (i.e. rotations in geometrical displacements), and the stress-strain relation (i.e. the material constitutive law). Consequently, a linearization of the deformation model must be performed at least once per time step, producing a new large linear system that need to be efficiently constructed and solved. Despite the large amount of previous works aimed at reducing the computational cost of this step, it often remains a major performance bottleneck.

Looking at this problem from another angle, the computational power available within the hardware of computers is ever increasing. However, the current trend is focused on increasing the number of computation units rather than their individual speed (in part due to frequency and heat-dissipation limits). Nowadays, one or two quad-core processors can be found in standard desktop computers. Moreover, the graphics processing unit (GPU) can now be exploited using general-purpose programming languages [Mun08, NVI07]. Using them, it is possible to exploit hundreds or even thousands of computation units to solve the numerical problems required for simulation. Doing so efficiently is however a very difficult undertaking. Indeed, it is necessary to separate a large number of parallel tasks, while optimizing the data access patterns to account for hardware-specific memory and cache hierarchical layouts. Compared to previous sequential systems, one important consequence is that to obtain the best results it is sometimes necessary to modify, or even completely change, the chosen algorithms.

In this work, we propose a new approach for a critical part of the simulation of deformable objects, namely the preconditioning of the iterative linear solver. We aim to achieve significant performance gains both by taking full advantage of the heterogeneous architecture of current hardware (and next generations as announced by several vendors), and exploiting the specificities of deformable object simulations, such as temporal coherence and geometrical non-linearities.

The rest of this paper is organized as follows. After reviewing related works in section 2, we present in section 3 the relevant deformable models and preconditioning techniques used in our work. Then, we propose in section 4 to desynchronize the computation of preconditioners from the simulation loop. As detailed in section 5, we factorize the system matrix in another thread, and re-use the last computed factorization in the main simulation loop. In section 6, we increase the efficiency of this new method by computing local rotations in a block diagonal matrix. This matrix is used for warping the matrix of the preconditioner and improves its duration of usefulness when simulating deformations with large rotations. In section 7, we also show that the preconditioning technique can be coupled with a GPU-based conjugate gradient implementation and provide significant accelerations of the computation time. Finally, in section 8, we evaluate and compare our method on several deformation models, using different preconditioners.

2. Related works

There is a considerable volume of works in the area of deformable objects simulation. In this section, we will concentrate on the numerical solver aspect, particularly in the context of interactive applications. We refer the reader to [NMK*06] for a broad survey of the other aspects.

In the literature, two families of algorithms are proposed: direct and iterative methods. First, direct solvers provide the solution in a fixed number of steps, initially by computing the actual inverse of the system matrix [BNC96], or more recently by creating a factorization that can then be used to compute the solution [BJ05]. These methods are often too costly to be applied at each time-step except for small models, which is why they often used as a pre-processing step for linear models. They are also often used in combination with an approach to reduce the number of degrees of freedom of the model, either using condensation on surface nodes [BNC96], or reduced-coordinate models [BJ05].

The second class of methods are iterative algorithms [BBC*94], which start from an initial estimate and iteratively refine it to approach the exact solution. The most popular algorithm is the Conjugate Gradient (CG) algorithm [She94], introduced in the context of cloth simulations by Baraff and Witkin [BW98]. Although in theory up to n iterations are necessary to achieve convergence (where n is the number of equations), in practice it is possible to stop the algorithm much sooner. However, while visually the result can

appear sufficiently realistic, a premature end of the iterative process can introduce artificial damping. Also, the convergence rate can be slow for ill-conditioned problem (such as non-homogeneous objects).

Despite the above limitations, the CG algorithm proved very efficient in most applications. It is also rather easy to implement, as it relies on only three basic operations: matrix–vector products, vector–vector inner products, and linear combination of vectors. A very interesting property of these operations is their inherent support for parallelism. Several works presented parallel implementations on CPU [PO09, HRF09] and GPU [BFGS03, BCL07]. The matrix–vector products are the most critical for the achieved performances. The main bottleneck is memory bandwidth, as less than two floating-point operations are executed for each scalar value read from memory. As such, the main difficulty is to find an efficient data storage which minimizes transfers between processors, and maximizes locality to efficiently exploit CPU cache hierarchies, or GPU coalesced accesses. An alternative approach is to never actually store the system matrix, but instead directly implement the product operations based on the elements contributing to the system of equations [ACF*07]. This can significantly reduce the required memory bandwidth, as well as improve code modularity.

The multigrid method is a fast and popular approach for solving large boundary value problems. It is based on estimating the solution using multiple levels of coarse approximations of the system [WT04]. Then, the solution is obtained by solving the coarse problem, which is fast to compute, and recursively using it as an approximation to the finer level. Recently, Zun et al. [ZSTB10] introduced an efficient multigrid method to solve high-resolution elastic models. However, building such hierarchy is not trivial, thus it is mostly used with regular meshes and homogeneous materials.

Another large area of research, and the most closely related to this work, aims to improve the performance of the CG algorithm with the use of preconditioners to speed-up its numerical convergence. In a way, the underlying idea is to combine the iterative solver with the use of a direct solver, applied however to an approximation of the system matrix. The proposed approximations range from generic methods to problem-specific analysis.

Regarding generic methods, Baraff and Witkin [BW98] proposed to use a diagonal inverse, often called a Jacobi preconditioner. [CK02] extends the method using a 3×3 block diagonal preconditioner, also used in [BA04] with the addition of support for projective constraints. More complicated preconditioner such as symmetric successive over-relaxation (SSOR) or Incomplete Cholesky (IC) factorizations have also been studied [HES03]. However, their results show a performance improvement of only up to 20% in typical simulations. Indeed, even if preconditioners are very

close to the real system matrix, their applications produce two overheads. The first one is the time to inverse the preconditioner, the second one is the time to apply it. Namely, some very good preconditioners (i.e. that provides a low condition number) are based on a factorization (complete or not) that needs to be computed at the beginning of the CG algorithm. The second part is the application of the preconditioner at each iteration of the CG. So, applying a preconditioner requires to find a ratio between the overhead to use it and the time we gain in the main loop of the simulation.

Finally, several techniques specific to the deformation models have been proposed. Boxerman et al. [BA04] propose a very efficient solver, specifically designed for cloth simulations, where only stiff forces are integrated implicitly. This, combined with projective constraints related to fixed points due to contacts, allows for a much sparser system matrix, which in turn can be re-ordered and partitioned to separate different regions that can be solved in parallel. Müller et al. [MDM*02] introduce a method called stiffness warping which consists in evaluating a global rotation matrix for a deformable body. This rotation is applied around the stiffness matrix, which remains constant during all the simulation. The rotation matrix can be computed using shape matching method [MHTG05] on a cloud of points without any constraint on the model. This global rotation matrix does not account for the local rotations on different part of the object when it deforms. However, Garcia et al. [GMPR06] exploit this method to build a very good preconditioner. They compute off-line an exact factorization of the system, and update their preconditioner using the global rotation matrix of the deformable object. They justify the use of the precomputation by the property that the condition number does never vary more than 25% during their simulations. A similar approximation, but using local rotations, is used in another context by [SDC08] to handle the contacts constraints, but they do not address the problem of preconditioning. An important limitation of this method, which is common to most approach relying on pre-computations, is that it does not support non-linear material constitutive laws, as well as online changes such as cutting or local mesh refinements.

3. Deformable objects simulation

This section presents an high-level view of the steps involved in the simulation of deformable models. We also briefly present the important features of the models and preconditioners used to illustrate our method.

3.1. Time integration equations

A very generic way of describing the physical behavior of deformable bodies is to model the internal and the external forces as a function $\mathbf{f}(\mathbf{x}, \mathbf{v})$ of the current state given by a set of positions $\mathbf{x}$ and velocities $\mathbf{v}$. The acceleration $\mathbf{a}$ is then defined using the *mass* matrix $\mathbf{M}$ and Newton's second law:

$$\mathbf{Ma} = \mathbf{f}(\mathbf{x}, \mathbf{v}). \tag{1}$$

This introduces a non-linear ordinary differential equation system which is integrated using an implicit scheme. It allows us to obtain a stable simulation with relatively large time steps but requires us however to compute the solution of a non-linear system of equations at each time step. In the remainder of the paper, we present our method using a backward Euler scheme and a unique linearization by time step. Thus, the velocities and positions are based on accelerations at the end of the time step:

$$\mathbf{v}_{t+h} = \mathbf{v}_t + h\mathbf{a} \qquad \mathbf{x}_{t+h} = \mathbf{x}_t + h\mathbf{v}_{t+h} \tag{2}$$

$$\mathbf{Ma} = \mathbf{f}(\mathbf{x}_{t+h}, \mathbf{v}_{t+h}) \tag{3}$$

To solve it we linearize $\mathbf{f}$ around the initial position:

$$\mathbf{f}(\mathbf{x}_{t+h}, \mathbf{v}_{t+h}) \approx \mathbf{f}(\mathbf{x}_t, \mathbf{v}_t) + \mathbf{K}(\mathbf{x}_{t+h} - \mathbf{x}_t) + \mathbf{B}(\mathbf{v}_{t+h} - \mathbf{v}_t) \tag{4}$$

where $\mathbf{K}$ is the *stiffness* matrix and $\mathbf{B}$ the *damping* matrix. Substituting (2) and (4) into (3) provides the final linearized system :

$$\underbrace{\left(\mathbf{M} - h\mathbf{B} - h^2\mathbf{K}\right)}_{\mathbf{A}} \mathbf{dv} = \underbrace{h\mathbf{f}(\mathbf{x}_t, \mathbf{v}_t) + h^2\mathbf{K}\mathbf{v}_t}_{\mathbf{b}} \tag{5}$$

where $\mathbf{dv} = h\mathbf{a} = \mathbf{v}_{t+h} - \mathbf{v}_t$.

We obtain a symmetric positive definite matrix $\mathbf{A}$ which allow us to use the CG algorithm to solve the system. While we develop here the equations corresponding to the Backward Euler integration scheme with a single linearization, we emphasis that the method proposed in this paper is not limited to this case, and could also be used with a Newmark scheme, or with the Newton-Raphson method and several linearizations per time-step.

3.2. Deformation models

To validate our method we use three different volume deformation models which compute the internal forces in different way. The simplest one is the mass spring model [Mil88] which consist in a set of point masses linked by 1D springs. A linear spring creates a force along the direction defined by the vector between two particles. The norm of the force is proportional to the difference between the current length of the spring and its rest length. Similar models are often used in cloth simulations and simple soft bodies.

We also apply our method to a co-rotational Finite Element Model (FEM) of Nesme *et al.* [NPF05]. The volume of each object is discretized into a set of tetrahedral elements. The deformation (the strain) inside each element is deduced from the displacements of its vertices using the *shape functions*, which are simply a linear interpolation based on barycentric coordinates in our case. The local rotation of each element is estimated in order to eliminate the influence of rotations on the computation of the strain (see Fig. 1). The material properties are given by the constitutive Hooke's law that provides a linear relation between the stress and the strain. The resulting internal forces are computed by integrating this relation onto the volume of each
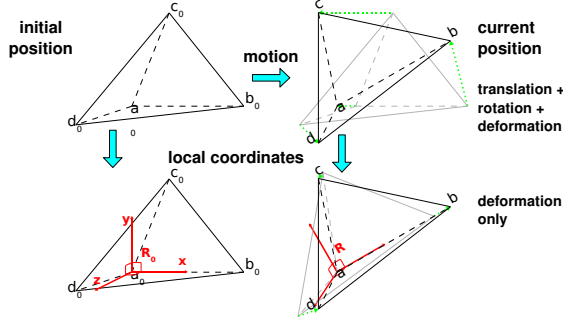
Figure 1: *Co-rotational FEM: a local frame is computed on each element to handle large rotations.*

element [MG04, HS04]. Note that computing a local rotation in order to eliminate the non-linearities they introduce is an idea that is reused in the method proposed in this paper.

Finally we used the Multiplicative Jacobian Energy Decomposition (MJED) method by Marchesseau *et al.* [MHC*10]. Its main feature is a fast stiffness matrix assembly for a large variety of isotropic and anisotropic materials. This model is a generic formulation of the energy inside hyperelastic materials on linear tetrahedral meshes. It permits us to simulate non-linear constitutive models such as St Venant Kirchoff in real-time.

After integration and linearization, all of these models and can be written as equation (5).

3.3. Preconditioning techniques

To handle non-homogeneous simulations or high material stiffnesses, it is necessary to use preconditioning to obtain a reasonable convergence rate. An important criteria for selecting the appropriate preconditioner is the *condition number* of the system matrix, which is a measure related to the difficulty of the problem. This number $\kappa(\mathbf{A})$ gives a bound on how inaccurate the solution $\mathbf{dv}$ will be if we apply a small perturbation on the force vector $\mathbf{b}$. It can be evaluated with :

$$\kappa(\mathbf{A}) = \|\mathbf{A}\| \cdot \|\mathbf{A^{-1}}\| \qquad (6)$$

A preconditioner is used to reduce the condition number of the system, by defining an approximation of the system matrix $\mathbf{A}$ which is easier to invert. Solving equation (5) with a preconditioner $\mathbf{P}$ can be written as :

$$\mathbf{P^{-1}Ax} = \mathbf{P^{-1}b} \qquad (7)$$

The condition number of this modified problem is associated with $\kappa(\mathbf{P^{-1}A})$. If we choose $\mathbf{P}$ sufficiently close to $\mathbf{A}$, then $(\mathbf{P^{-1}A})$ will be close to the identity matrix, its condition number close to 1, ensuring a fast convergence.

For this paper, we used some well known preconditioners which present different inversion and application costs. Some of them can be easily obtain from the system matrix,

such as Jacobi which is simply the diagonal matrix, or SSOR which consist in approximating the matrix with a factorization based on its lower and upper triangular sub-matrices:

$$\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{L^T} \quad \Rightarrow \quad \mathbf{P} = (\mathbf{D} + \mathbf{L})\mathbf{D^{-1}}(\mathbf{D} + \mathbf{L})^T \quad (8)$$

where $\mathbf{L}$ is the lower triangular part of $\mathbf{A}$. We also used more computational intensive preconditioner such as Cholesky factorization, which consist in computing :

$$\mathbf{A} = \mathbf{LL^T} \qquad (9)$$

where $\mathbf{L}$ is a triangular sparse factorization of $\mathbf{A}$. Finally, we also used an Incomplete Cholesky Factorization which consist in dropping values in the factorization which are under an arbitrary tolerance. All of these preconditioners produces sparse factorization which are more efficient to apply than a dense inverse matrix.

The most efficient preconditioners, based on factorizations, are difficult to parallelize on a GPU. Then, even with a GPU version of the CG, such computations remains on the CPU. However, this requires the transfer of multiple vectors between CPU and GPU at each iteration and penalizes the use of preconditioners on GPU-based CG implementations.

4. Amortized preconditioning

Computation time is a major constraint for interactive simulations. Indeed, when we simulate the deformation of an object in real time, only few milliseconds should be spent for the computation of each step. Fortunately, we can exploit time coherency to reuse some of the results between time-steps. In most cases, there is not much variation of the simulation state $(\mathbf{x}, \mathbf{v})$ and the system matrix $\mathbf{A}$ from one step to another. Thus, the first contribution of this paper is to exploit this property by reusing preconditioners. As discussed in section 2, previous works showed that only the simplest preconditioners are beneficial for interactive simulations, as the computation time needed to obtain an accurate preconditioner can be larger than the frame rate of the simulation.

In typical cases, this temporal coherency only holds for short period of time. The system can significantly and abruptly change if the objects undergo large deformations such as introduced by collisions and user interactions. Indeed, if we never update the preconditioner, the simulation will increasingly diverge from its initial state, and the preconditioner will actually end-up slowing down the simulation. To avoid this problem, we propose to update the preconditioner periodically. These updates could be used after a user-specified number of time-steps, or adjusted more intelligently based on an evaluation of the quality of the preconditioner given the changes since its computation.

We can express the current system matrix as a perturbation from previous matrices of the simulation. If we call $\Delta\mathbf{A}$ a perturbation to the current system matrix, we obtain :

$$\mathbf{A}(x_{t+\Delta t}) = \mathbf{A}(x_t) + \Delta\mathbf{A} \qquad (10)$$

When Δt is small, $\|\Delta \mathbf{A}\|$ is close to zero. Moreover, if we compute an exact preconditioner such that $\mathbf{P}^{-1}(x_t) = \mathbf{A}^{-1}(xt)$, the generated error introduced by the delay Δt of the preconditioner application, can be evaluated with :

$$\mathbf{P}^{-1}(x_t) \cdot \mathbf{A}(x_{t+\Delta t}) = \mathbf{I} + \mathbf{P}^{-1}(x_t) \cdot \Delta \mathbf{A} \qquad (11)$$

For sufficiently small Δt, the condition number $\kappa(\mathbf{P}^{-1}(x_t) \cdot \mathbf{A}(x_{t+\Delta t}))$ should not degrade too far from the condition number of the original preconditioner. This approach should be most beneficial for ill-conditioned simulations involving few or very localized sudden changes.

In order to amortize the cost of computing a preconditioner, the computation steps for a given time-step will differ depending on whether we decide to update the preconditioner. If it is updated, then we first need to build the system matrix $A(x_t)$ (what we will call the *Build* task). Then we construct a new preconditioner P_t from this matrix, which can involve computing its inverse or a symmetric factorization (the *Invert* task). Finally, we use this preconditioner in the CG algorithm to find the solution and update the simulation (the *Step* task). If the preconditioner is not updated in a given time-step, then we simply keep the previously built preconditioner, so the *Invert* task is skipped. However, the *Build* task can still be necessary, depending on whether implementation of the CG uses the explicitly built matrix A.

To decide when to update the preconditioners, several heuristics can be used, based on : the number of simulation steps during which the same preconditioners have been used; the number of iterations currently necessary for the CG to converge. In simple simulations with only one deformable object, it is often enough to use a constant number of steps between updates. For more demanding scenarios, such as multiple simulated objects that are all candidates for updates, a more advanced heuristic based on both criteria can be used to trigger the update only when it is necessary. For instance, it would be possible to set a minimal number of simulation steps as well as CG iterations below which a preconditioner is never updated (avoiding ceaseless recomputations). Additionally, we can maintain for each object a sum of all CG iterations above the threshold, as a priority criterion for the update of the associated preconditioner relative to the others.

The scheme proposed in the section allows the use of much more powerful preconditioners which are very costly to build, as this overhead will be amortized over all the subsequent time-steps until it is updated again.

5. Multi-threaded asynchronous preconditioning

The previous technique provides a scheme that improves the *overall* simulation time. It is achieved by only updating the preconditioner in a subset of the time-steps. However, an important drawback is that these few time-steps can be significantly longer to compute than the others. As a consequence, this first technique is unsuitable for interactive applications, where a smooth refresh rate is desired.

To overcome this limitation, we propose to use a second thread that will execute the factorization of the preconditioner (see Fig. 2). Indeed, nowadays most computers have several computation units, and it is thus beneficial to share the computation cost between them. However, multi-threaded programming can be difficult, as all dependencies and synchronizations between threads must be carefully handled. Fortunately, the *Invert* task is well segregated from the rest of the simulation: its only input is the current system matrix A, while its only output is the data structures necessary to apply the preconditioner (i.e. the inverse matrix or factorization of P).
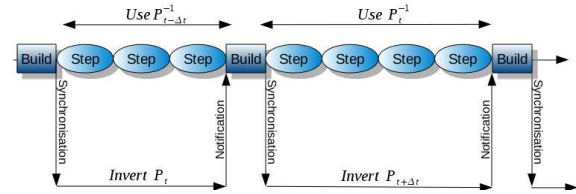
Figure 2: *Multi-threaded preconditioning. The simulation thread is never blocked to compute a preconditioner, while the second thread can exploit all its computational power to produces high-quality preconditioner for the simulation.*

Optionally, the *Build* task, constructing matrix A, can be migrated from the simulation thread to the factorization thread. This is only useful if the simulation thread does not otherwise need to build this matrix. Migrating this operation implies a more invasive change in the implementation of the simulation, as all internal data structures used in the construction of this matrix now need to be shared between threads. However, if both the *Build* and *Invert* tasks are executed in the factorization thread, then simulation steps where the preconditioner is updated take nearly the same time as other steps, as only the inter-thread communication overheads remain. This leads to the fastest and smoothest refresh rate for the simulation. Both approach were implemented for this paper. In our experience, the use of the first option only introduces a maximal computation overhead of 10% for the simulation steps that are chosen to building the matrix.

One preconditioner is factorized in a separate thread while another is used, at the same time, by the CG in the simulation loop. Similarly, the system matrix A is used in the factorization thread while a new one could be built and used within the simulation. Consequently, we need a double buffer that avoids writing conflicts, allowing one thread to use one version of a matrix while the other is computed a new version. This double buffering has a limited memory overhead as we use a compact sparse representation such as Compressed Row Storage (CRS) [BBC*94].

The launch of a new factorization can simply be triggered by the end of the preceding computation. This provides to the CG the latest available preconditioner, while the thread

associated to the preconditioner never stops computing. We could also use a more advanced heuristic, as presented in the previous section, to update the factorization only when it is necessary. It is particularly useful if several objects are simulated and several preconditioners need to be updated at the same time. The only additional criteria, but an important one nonetheless, is that the simulation thread should never wait on the factorization results, and thus the computation of a new preconditioner should only be started once the result of the preceding one was received.

Overall, the proposed multi-threaded approach allows us to desynchronize the preconditioner from the simulation to produce good inverse approximations of the system matrix continuously. Compared to the scheme proposed in the previous section, the main drawback of this new approach is that when the simulation thread receives a new preconditioner, it is already old. Indeed, it was computed using a matrix built several steps previously. Defining Δt as the average update period in simulation time, each preconditioner will be used from time $t + \Delta t$ to time $t + 2\Delta t$. Using the previous approach, it would have been used from time t to $t + \Delta t$, so the preconditioners used in the asynchronous approach are three times as old on average as in the single-threaded version. On the other hand, the major gain is that the factorization computation doesn't slow down the simulation.

6. Local rotations warping

During the time we update the next preconditioner, simulated objects keep moving. This motion can include deformations, but also translations and rotations. Most deformable models rely on a Lagrangian formulation, where the degrees of freedom are the spatial coordinates of a set of points (often called particles or nodes) embedded withing the objects. The physical model contribute mass and stiffness acting between their particles, from which the values of the system matrix A are derived. In most cases, small translations and deformations do not significantly alter these values, however even a relatively small rotation will have an important effect. Intuitively, it is similar to a change of coordinate system for the definition of the degrees of freedom. Such change can quickly decrease the effectiveness of a preconditioner based on an earlier simulation state.

To overcome this problem, we propose to use a method inspired by the co-rotational formulation in FEM [MG04, NPF05], where a rotation matrix is estimated from the current motion at the level of each element and used to compute the stiffness matrix in a local frame (see Fig. 1). To apply the same idea to the preconditioner, we first estimate the rotation for each node of the model, relative to their position at the time the preconditioner was computed.The rotation of all the nodes are stored in a block diagonal rotation matrix $\mathbf{R}$. This rotation matrix is then used to warp the factorized matrix of the preconditioner, i.e. equation (7) is replaced by:

$$\mathbf{R}\mathbf{P}^{-1}\mathbf{R}^{\mathbf{T}}\,\mathbf{A}\mathbf{x} = \mathbf{R}\mathbf{P}^{-1}\mathbf{R}^{\mathbf{T}}\,\mathbf{b} \qquad (12)$$

However, unlike previous works, the rotations are not defined from the rest position but from the position $\mathbf{x}_t$ at time t that was picked to compute the preconditioner. This warping of the preconditioner matrix allows to account for a part of the geometrical non-linearities that comes from the rotations of the elements between the preconditioner invert position and the current position (see Fig. 3).
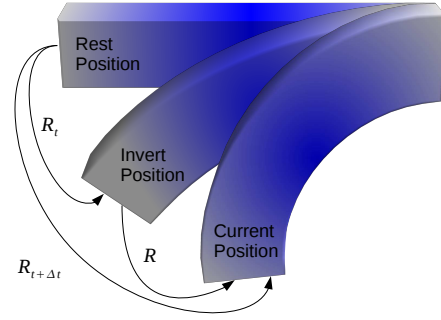
Figure 3: *We estimate, for each node of a deformable object, a rotation at different times, using the rest position as a reference. Combining them provides the rotation matrix* $\mathbf{R}$ *from the time t that a preconditioner was computed, to the current simulation time $t + \Delta t$.*

When the deformation model uses a corotational formulation [MG04], we exploit the computed rotations on each element, using for each node the mean rotation from neighboring elements. For other models, we rely on shape matching [MHTG05], computing an estimate of the rotations of each node using the motions of the nodes that are its topological neighbors up to a given level (typically 1 or 2).

Often the rotations of the elements are evaluated from a given undeformed position (i.e. the rest shape in FEM), which is not the invert position used to perform the factorization. To solve this problem, we evaluate the current rotations $\mathbf{R}_{\mathbf{t}+\Delta\mathbf{t}}$ from the rest position and we apply the inverse of the rotations $\mathbf{R}_{\mathbf{t}}$ of the invert position (see Fig. 3) :

$$\mathbf{R} = \mathbf{R}_{\mathbf{t}}^{-1}\,\mathbf{R}_{\mathbf{t}+\Delta\mathbf{t}} \qquad (13)$$

where $\mathbf{R}$ is the final matrix used in equation (12). In practice, the product $\mathbf{R}\mathbf{P}^{-1}\mathbf{R}^{\mathbf{T}}$ is never explicitly computed, instead we implement matrix–vector products by sequentially multiplying by each matrix.

Warping a preconditioner with a rotation matrix is useful if parts of the deformations derive from rotating element, which is a common occurrence. The overhead is relatively small, as the computation time is linear in the number of nodes, and the resulting matrix is block diagonal. In summary, it can be seen as a way to limit the deterioration of preconditioners, by removing some geometrical non-linearities, which should allow us to update the matrix less frequently.
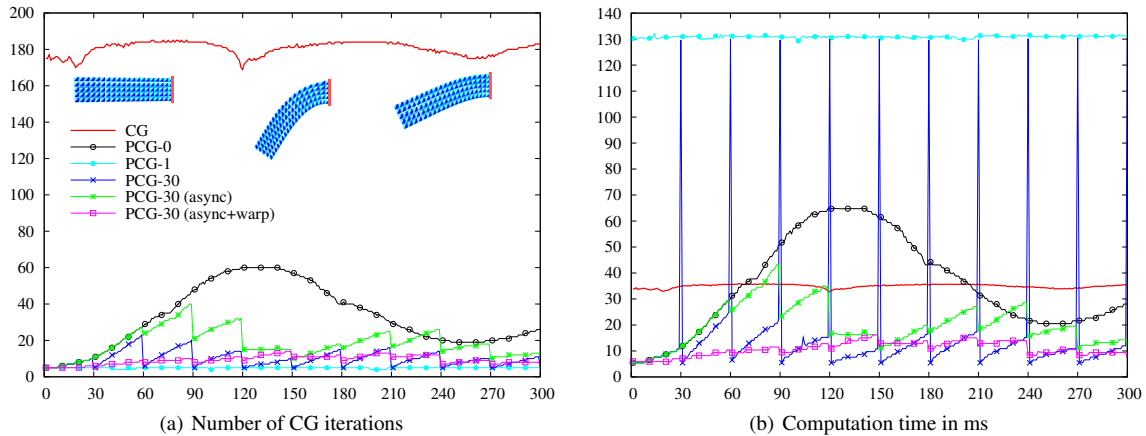
Figure 4: *Performances of different preconditioning schemes during the simulation of a co-rotational FEM beam bending under gravity, measured as (a) the number of iterations required by the CG to converge, and (b) the computation time per simulation time-step. All preconditioners uses an Incomplete Cholesky factorization, but with different update strategies.*

7. Implementation

Many components are combined to form a complete physical-based simulator. In this section, we describe the features of our implementation that are important to understand and reproduce the results of the next section.

For the mechanics, we used a Backward Euler time integrator, relying on CG [She94] to solve the linear system. The deformation models presented in section 3.2 were all implemented on the CPU, with an additional GPU version for the mass-spring and co-rotational FEM [CJA*10].

Simple preconditioners (Jacobi, SSOR) are implemented directly, while those requiring sparse factorizations are delegated to external highly optimized solver libraries : Taucs [TCR03] or Pardiso [SBR06]. Both libraries provides parallel CPU implementations of complete or incomplete factorizations, while the application of this factorization as a preconditioner is implemented in parallel only by Pardiso. This additional level of parallelism permit us to use all CPU cores for the preconditioners, while the GPU is exploited to compute the mechanical computations.

8. Results and discussion

In this section, we validate and present the results of our approach, by measuring convergence rates and computation times in different scenarios. For the experiments, we used a quad-core Intel® Core™ i7 3.07 GHz CPU, and a Nvidia® GeForce® GTX 480 GPU programmed with Cuda 3.0.

8.1. Amortized and asynchronous preconditioning

In this section we evaluate the influence of the asynchronous inversion of the preconditioner. We chose the simple example of an homogeneous beam (similar to Fig. 3) which is fixed at one extremity and simply deforms under gravity.

We used an incomplete Cholesky preconditioner, and the deformable object is modeled as 3000 tetrahedral elements using co-rotational FEM.

Fig. 4 presents the obtained performances, the number of iterations to converge and the required time to compute each time-step, for six different preconditioning schemes :

- *CG* : no preconditioner.
- *PCG-0* : pre-computed preconditioner.
- *PCG-1* : preconditioner updated at each time-step.
- *PCG-30* : preconditioner only updated every 30 time-steps (section 4).
- *PCG-30 (async)* : asynchronous update of the preconditioner using a separate thread (section 5).
- *PCG-30 (async+warp)* : asynchronous update with local rotations warping (section 6).

CG and *PCG-1* correspond to the classical CG algorithm with or without preconditioning. They provide a basis for comparing all the other schemes. All preconditioners should improve on the convergence rate of *CG*, as well as its computational cost (which is harder, due to their introduced overheads). *PCG-1*, which always rely on a fresh preconditioner, provides the best convergence rate, but also in this scenario the worst performances, as it requires computing an incomplete factorization of the matrix at each time-step. This factorization takes more time than needed by the non-preconditioned CG to converge. Note that is this is partly due to the numerical well-conditioning of this scenario (the object is homogeneous), as well as the use of GPU to implement CG iterations. *PCG-0* illustrates the performances that we obtain if we use a pre-computed preconditioner that is never updated. While it provides a reasonable convergence rate in this example, it quickly degrades when deformations become increasingly far from the rest shape.

PCG-30 illustrates the first and simplest scheme proposed in this paper : we simply update the preconditioner only every 30 simulation steps. Indeed, we can see that on time-steps where this update occurs, the convergence rate is as good as *PCG-1*, however the computational cost also spikes to the same level. In-between these steps, the quality of the preconditioner decreases. While the total cost of this scheme is better than *CG*, the stalls introduced by the periodical updates forbid its use in interactive applications. When the inversion is computed in a second thread (*PCG-30 async*), the computation of the factorization do not affect the computation time in the simulation anymore. This second scheme removes the periodic stalls, at the cost however of using older preconditioners, which negatively affect the convergence rate (we can see that the number of iterations of *PCG-30 async* starts close to the last value of *PCG-30* and continue to degrade from it until the next update). This drawback is alleviated in the final scheme (*PCG-30 async+warp*), which uses estimated local rotations to reduce the rate at which the quality of a preconditioner decrease as the simulation advances. This scheme obtains a nearly optimal preconditioner at each step of the simulation, with both few iterations and fast computations without stalls. Note that a more detailed evaluation of the influence of warping will be presented in section 8.2.

We also measured the influence of the size of the simulation mesh on the performance (see Table 1). We use a Cholesky preconditioner computed using Pardiso [SBR06] with 4 threads for the simulation and 4 threads for the factorization.As expected, the computation time increases with the number of elements. Moreover, the time taken to factorize the matrix increases but always represent a small number of simulation steps (less than 4 steps), as each iteration of the preconditioned CG is also more costly. This result hints that our method should be beneficial for both the simulation of small and large objects, as a similar update frequency for the preconditioner can be maintained. However, we did not study the scalability of our method beyond size limit where the simulation is no more interactive.

Elements	Time (ms) to factorize	CG iterations	Time (ms) per steps	Steps to factorize
540	5.84	4.40	3.32	1.76
3000	38.47	5.01	12.78	3.01
7350	125.9	5.56	41.68	3.02
24000	730.5	6.09	208.72	3.50

Table 1: *Performances with different mesh sizes.*

8.2. Warping and non-linearities

This section presents the influence of the preconditioner warping method introduced in this paper (see section 6), on different deformation models. Different deformations are simulated to introduce respectively material non-linearities, geometrical non-linearities, and both. In Fig. 2, we evaluate the performance obtained with or without warping, on different deformation models : mass-spring, co-rotational FEM [NPF05], and MJED [MHC*10]. While rotation warping is not a new idea [MG04, GMPR06, SDC08], it has not been used in the same manner before, and more importantly it was not validated on deformation models other than co-rotational FEM, such as MJED which include material non-linearities.

We tested three preconditioning schemes: none, precomputed, and asynchronous preconditioners. The precomputed preconditioner with warping illustrates the performances that is achieved if we combine the works of García *et al.* [GMPR06] and Saupin *et al.* [SDC08], i.e. never updating the preconditioner but correcting geometrical non-linearities by local rotations warping. From its results, we can see than warping alone is useful, however it does not entirely remove the need to update the preconditioner at least periodically.
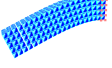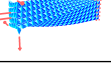
Using our asynchronous preconditioner method with warping, significant improvements of the quality of the preconditioner are obtained when geometrical non-linearities occur (simulations 2 and 3). Indeed, the warping method divides the required number of iterations up to a factor of $5\times$ on co-rotational FEM, $2\times$ for mass springs and $1.5\times$ for MJED.

When the simulation doesn't introduce geometrical non-linearities, the warping technique slightly decreases the performances. However, in the worst case, when the beam is being stretched using springs, the overhead of using warping method is at max. 20% of the computation time and represents only one or two additional iterations. In contrast, using a corotational model, the warping method is always beneficial. To address this slight inefficiency, it could be beneficial to enable or disable the warping according to the amount of local rotations computed during the deformation.

8.3. Preconditioners evaluation

We propose to evaluate the convergence using different preconditioners on non homogeneous problems. These problems are known to be difficult for the CG. We present performances evaluation in Table 5. The first (standard) preconditioners are updated at each step of the simulation whereas (async + warp) versions are factorized in another thread.

One can first notice that all of the preconditioners produce an acceleration compared to the standard CG which requires many iterations to converge on heterogeneous examples. On this particular example, the IC updated at each steps provides an acceleration of about 10% (whereas it was $4\times$ slower than the CG on the homogeneous simulation of Fig. 4). For the exact Cholesky factorization, we use an efficient parallel super-nodal algorithm provided by *Taucs* [TCR03], while only a less-optimized algorithm is available for the incomplete factorization (which is why it is actually slower to compute). The exact solver requires only to be applied once as preconditioner, producing an acceleration about $3\times$ faster

Load scenario	Model	No preconditioner		Precomputed preconditioner				Asynchronous preconditioner			
		No warping		No warping		Warping		No warping		Warping	
		Iter.	Time	Iter.	Time	Iter.	Time	Iter.	Time	Iter.	Time
	Corot.	257.44	110.97	14.23	23.45	10.94	19.70	6.91	12.72	5.61	11.27
	MJED	117.36	13.74	12.24	16.33	13.99	20.41	4.51	6.97	4.73	8.10
	Spring	56.28	8.34	13.46	18.82	16.03	23.94	7.72	14.24	8.14	15.90
	Corot.	252.63	107.40	20.99	34.30	6.90	12.57	13.58	23.02	6.51	12.83
	MJED	179.28	20.03	13.52	17.86	8.77	13.19	9.52	12.98	6.62	11.24
	Spring	82.55	11.72	16.81	23.31	9.51	14.56	11.82	19.89	8.65	17.00
	Corot.	264.20	119.39	93.20	154.63	15.54	27.83	20.70	35.19	6.87	13.76
	MJED	119.61	16.32	35.87	47.94	28.23	40.44	10.33	14.35	8.78	13.82
	Spring	51.28	8.54	44.29	62.85	19.61	29.68	11.72	18.59	7.71	14.30

Table 2: *Influence of rotation warping for different solicitations applied on different deformation models. The number of iterations represents the average values over a time sequence. The beams are composed of 3000 homogeneous tetrahedral elements, the condition number is approximately 1246.*

than the CG. Note also that the achieved performance corresponds to what would be obtained if we used this factorization as a direct solver, as it would also have been computed and applied once per time-step.

The Jacobi and SSOR preconditioners generate an acceleration of about 40%. However, SSOR doesn't require any precomputation to invert the matrix. Then, their *Invert* time correspond only to the time to build the matrix. This operation represent 30% of the application of the SSOR preconditioner. Our method permit us to avoid the cost of this build

	Preconditioner	CG iterations	Computation time (ms)		
			Invert	Solve	Total
Standard	– (CG)	654.03	–	126.44	126.44
	Jacobi	356.96	.15	86.42	86.58
	SSOR	116.72	30.60	55.61	86.21
	Cholesky	1.00	39.27	3.13	42.40
	IC	6.07	107.24	7.29	114.54
async + warp	Jacobi	357.99	.16	93.18	93.34
	SSOR	116.76	.26	62.70	62.97
	Cholesky	6.02	.06	11.92	11.98
	IC	8.71	.03	10.76	10.79

(a) Performance measurements on non-homogeneous beam

(b) Non homogeneous beam   (c) Interactive simulation

Figure 5: *(a) Performances evaluation of a non homogeneous beam composed of 3000 elements, falling under gravity. (b) Different colors shows the different young modulus (1000, 50 and 10 for resp. the blue, green and red elements), the condition number of the matrix is 10317. (c) Interactive example composed of 4406 non homogeneous elements.*

within the simulation loop, without significantly decreasing the quality of the preconditioner, as the matrix from the previous iteration will be used instead. However, the Jacobi preconditioner is the only one which does not benefit from our method, as its matrix is too simple to build and invert.

The Cholesky and Incomplete Cholesky factorizations are the fastest preconditioners we used in this paper. Indeed, their lack of efficiency found in previous works come from the factorization cost. With our method this factorization is hidden away in another thread. In addition, the warping technique permits to limit the divergence of the approximation even on a non homogeneous object. This allows us to simulate homogeneous and heterogeneous objects with about the same level of accuracy, as even highly non-homogeneous simulations such as in the illustrated example converges within 10 iterations once a precise preconditioner is used.
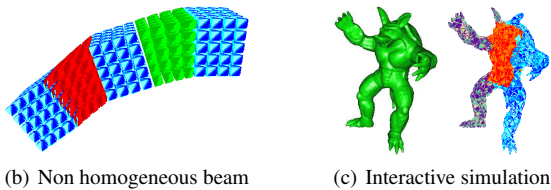
8.4. Interactive simulation

While we concentrated on non-interactive test-cases to evaluate our method, we also tested it in an interactive application visible in Fig. 5(c) and better illustrated in the accompanying video. We use an example of a deformable armadillo with a non homogeneous stiffness, that the user can interactively move and rotate to experience the responsiveness and smoothness of the simulation. In this example, while a non-preconditioned version would not converge in a reasonable time for interactive rates, our method was able to maintain a framerate between 50 and 80 FPS.

9. Conclusion

This paper presents a new method to improve the computation time of physics-based deformation models in the context of real-time and accurate simulations. The simulation uses a conjugate gradient algorithm that is very efficiently preconditioned using an asynchronous factorization and a warping technique. Results are illustrated using various deformation models and multiple sequences of simulation cases, and show performance improvements up to $5\times$.

A limitation of our method can appear in case of collisions producing sudden modifications of the system. In this case the preconditioner may not be as useful as in our examples. Therefore, the preconditioned conjugate gradient could require more iterations to converge and the user may experience a slowdown in the simulation. However, if the frame rate has to be maintained, it's also possible to limit the number of iterations. In this case, the conjugate gradient won't converge and virtual objects might look softer, but this should be corrected as soon as objects are stabilized and the preconditioner is updated.

Another important limitation of our method is the lack of support for topological changes such as cutting or fracturing simulations. Indeed, when such modifications occur, the dimension and the structure of the matrix change. In these cases, one could simply disable the preconditioner until an updated version is available on the new topology. However, more efficient schemes could be studied as a possible direction of future works. Another interesting extension to investigate could be the handling of multiple objects within the simulations, as we suggested some heuristics on how to decide which preconditioner might be the best to update, but this topic requires further studies.

References

[ACF*07] ALLARD J., COTIN S., FAURE F., BENSOUSSAN P.-J., POYER F., DURIEZ C., DELINGETTE H., GRISONI L.: SOFA - an open source framework for medical simulation. In *Medicine Meets Virtual Reality (MMVR'15)* (2007), pp. 1–6.

[BA04] BOXERMAN E., ASCHER U.: Decomposing cloth. In *Proceedings of SCA '04* (2004), ACM SIGGRAPH/Eurographics, pp. 153–161.

[BBC*94] BARRETT R., BERRY M., CHAN T. F., DEMMEL J., DONATO J., DONGARRA J., EIJKHOUT V., POZO R., ROMINE C., DER VORST H. V.: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, 1994.

[BCL07] BUATOIS L., CAUMON G., LÉVY B.: Concurrent number cruncher: An efficient sparse linear solver on the GPU. In *High Performance Computation Conference (HPCC)* (2007).

[BFGS03] BOLZ J., FARMER I., GRINSPUN E., SCHÖDER P.: Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Trans. Graph. 22* (2003), 917–924.

[BJ05] BARBIČ J., JAMES D. L.: Real-time subspace integration for st. venant-kirchhoff deformable models. *ACM Trans. Graph. 24*, 3 (2005), 982–990.

[BNC96] BRO-NIELSEN M., COTIN S.: Real-time volumetric deformable models for surgery simulation using finite elements and condensation. *Comput. Graph. Forum 15*, 3 (1996), 57–66.

[BW98] BARAFF D., WITKIN A.: Large steps in cloth simulation. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (1998), ACM, pp. 43–54.

[CJA*10] COURTECUISSE H., JUNG H., ALLARD J., DURIEZ C., LEE D. Y., COTIN S.: GPU-based real-time soft tissue deformation with cutting and haptic feedback. *Progress in Biophysics and Molecular Biology* (2010). Special Issue on Soft Tissue Modelling, to appear.

[CK02] CHOI K.-J., KO H.-S.: Stable but responsive cloth. *ACM Trans. Graph. 21*, 3 (2002), 604–611.

[GMPR06] GARCÍA M., MENDOZA C., PASTOR L., RODRÍGUEZ A.: Optimized linear fem for modeling deformable objects. *Comput. Animat. Virtual Worlds 17* (2006), 393–402.

[HES03] HAUTH M., ETZMUSS O., STRASSER W.: Analysis of numerical methods for the simulation of deformable models. *The Visual Computer 19*, 7-8 (2003), 581–600.

[HRF09] HERMANN E., RAFFIN B., FAURE F.: Interative physics simulation on multicore architectures. In *Proceedings of the 9th Eurographics Symposium on Parallel Graphics and Visualization (EGPGV'09)* (Mar. 2009).

[HS04] HAUTH M., STRASSER W.: Corotational simulation of deformable solids. In *Proceedings of WSCG* (2004), pp. 137–145.

[MDM*02] MÜLLER M., DORSEY J., MCMILLAN L., JAGNOW R., CUTLER B.: Stable real-time deformations. In *Proceedings of SCA '02* (2002), ACM SIGGRAPH/Eurographics, pp. 49–54.

[MG04] MÜLLER M., GROSS M.: Interactive virtual materials. In *GI '04: Proceedings of Graphics Interface 2004* (2004), Canadian Human-Computer Communications Society, pp. 239–246.

[MHC*10] MARCHESSEAU S., HEIMANN T., CHATELIN S., WILLINGER R., DELINGETTE H.: Multiplicative jacobian energy decomposition method for fast porous visco-hyperelastic soft tissue model. *MICCAI'10* (Sept. 2010).

[MHTG05] MÜLLER M., HEIDELBERGER B., TESCHNER M., GROSS M.: Meshless deformations based on shape matching. *ACM Trans. Graph. 24*, 3 (2005), 471–478.

[Mil88] MILLER G. S. P.: The motion dynamics of snakes and worms. In *Proceedings of SIGGRAPH '88* (1988), ACM, pp. 169–173.

[Mun08] MUNSHI A. (Ed.): *The OpenCL Specification Version: 1.0.* The Khronos Group, 2008.

[NMK*06] NEALEN A., MUELLER M., KEISER R., BOXERMAN E., CARLSON M.: Physically based deformable models in computer graphics. *Comput. Graph. Forum 25* (2006).

[NPF05] NESME M., PAYAN Y., FAURE F.: Efficient, physically plausible finite elements. In *Eurographics 2005* (2005).

[NVI07] NVIDIA CORPORATION: NVIDIA CUDA compute unified device architecture programming guide, 2007.

[PO09] PARKER E. G., O'BRIEN J. F.: Real-time deformaton and fracture in a game environment. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animaton* (Aug. 2009), pp. 156–166.

[SBR06] SCHENK O., BOLLHÖFER M., RÖMER R. A.: On large-scale diagonalization techniques for the anderson model of localization. *SIAM J. Sci. Comput. 28*, 3 (2006), 963–983.

[SDC08] SAUPIN G., DURIEZ C., COTIN S.: Contact model for haptic medical simulations. In *ISBMS '08: Proceedings of the 4th international symposium on Biomedical Simulation* (2008), pp. 157–165.

[She94] SHEWCHUK J. R.: *An introduction to the conjugate gradient method without the agonizing pain.* Tech. Rep. CMU-CS-TR-94-125, Carnegie Mellon University, 1994.

[TCR03] TOLEDO S., CHEN D., ROTKIN V.: Taucs: A library of sparse linear solvers version 2.2, Feb. 2003.

[WT04] WU X., TENDICK F.: Multigrid integration for interactive deformable body simulation. In *Proceedings of International Symposium on Medical Simulation* (2004), pp. 92–104.

[ZSTB10] ZHU Y., SIFAKIS E., TERAN J., BRANDT A.: An efficient multigrid method for the simulation of high-resolution elastic solids. *ACM Trans. Graph. 29*, 2 (2010), 1–18.