2009 11th IEEE International Conference on High Performance Computing and Communications

Parallel Dense Gauss-Seidel Algorithm on Many-Core Processors

Hadrien Courtecuisse and Jérémie Allard
INRIA Lille Nord Europe, France
Email: hadrien.courtecuisse@inria.fr, jeremie.allard@inria.fr

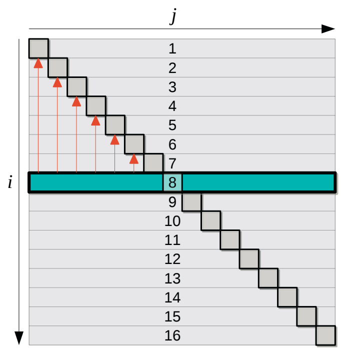
***Abstract*—The Gauss-Seidel method is very efficient for solving problems such as tightly-coupled constraints with possible redundancies. However, the underlying algorithm is inherently sequential. Previous works have exploited sparsity in the system matrix to extract parallelism. In this paper, we propose to study several parallelization schemes for fully-coupled systems, unable to be parallelized by existing methods, taking advantage of recent many-cores architectures offering fast synchronization primitives. Experimental results on both multi-core CPUs and recent GPUs show that our proposed method is able to fully exploit the available units, whereas trivial parallel algorithms often fail.**

This method is illustrated by an application in medical intervention planning, where it is used to solve a linear complementary problem (LCP) expressing the contacts applied to a deformable body.

***Keywords*-parallel algorithms, linear complementary problem, GPGPU, physically based modeling**

```
1  repeat
2      $\epsilon = 0$
3      for $i = 1$ to $n$ do
4          $t_i = 0$
5          for $j = 1$ to $n$ do
6              if $i \neq j$ then
                   $t_i \mathrel{+}= a_{ij} * x_j$
7          $\tilde{x}_i = x_i$
8          $x_i = -(b_i + t_i)/a_{ii}$
9          if $x_i < 0$ then
               $x_i = 0$
10         $\epsilon = \epsilon + |\tilde{x}_i - x_i|$
11 until $\epsilon < \epsilon_{converge}$
```

(a)



(b)

Figure 1. (a) Gauss-Seidel algorithm applied to a LCP. (b) Processing a given line i requires results of all the previous lines (these dependencies are represented here by red arrows).

I. Introduction

The Gauss-Seidel method is used in many applications, to solve problems such as a set of inter-dependent constraints or as a relaxation step in multigrid methods [1]. The heart of the algorithm is a loop that sequentially process each unknown quantity. Many parallelization schemes have been studied on high performance computing platforms, exploiting the sparsity of the system matrix to process in parallel independent quantities (i.e. variable i and j can be computed independently as long as the element (i, j) in the matrix is zero). A tightly coupled system however, leading to a dense matrix, cannot be thus divided. This property, combined with the cost of synchronization primitives in traditional distributed architectures, prevented parallelizing the Gauss-Seidel method in this case, and alternate algorithms such as the fully parallel but slow to converge Jacobi method were instead used.

In recent years, computing architectures have become increasingly parallel, with the ubiquitous use of multi-core CPUs, and the massive parallelism available in modern GPU, like the NVIDIA GeForce GTX 280 with 240 cores, or the AMD Radeon HD 4850 with 800 cores. These new architectures do not offer the same trade-off in term of computation power versus communication and synchronization overheads, as traditional high-performance platforms. This change, combined with the need to rely on parallelism rather than continuous improvements of a single unit's speed to increase the speed of a computation, justifies revisiting the possible parallelization strategies of broadly used algorithms.

This paper aims at presenting several parallelization strategies for the dense Gauss Seidel method, from simple internal loop vectorization to more complex schemes such as multi-level synchronizations and fine-level dependencies analysis. These strategies are compared and evaluated through performance measurements on a large range of hardware architectures.

II. Related Works

A. The Gauss-Seidel Method

The Gauss-Seidel method [2] is an iterative technique used to solve problems such as a linear system of equations:

$$Ax + b = 0 \tag{1}$$

or a linear complementary problem (LCP) [3]:

$$w - Ax = b \tag{2}$$

$$w_i \geq 0,\ x_i \geq 0 \text{ and } w_i x_i = 0 \text{ for all } i \tag{3}$$

Note that (3) simply means that all values in x and w are positive and for each i at least one of (x_i, w_i) must be 0. To resolve such a system of equations with the Gauss-Seidel method, each unknown variable x_i is sequentially updated according to the i^{th} equation. The resulting algorithm applied to a LCP is presented in Fig. 1.

To resolve a linear system such as (1), line 9 must be removed. Other types of problems (bilateral constraints, friction, ...) can be handled by changing computations in lines 8 and 9. However, the overall structure and data dependencies remain the same. In the remainder of this paper, we will only consider the resolution of unilateral constraints, however the discussed methods can be similarly applied to other problems.

B. Parallel Constraints Solvers

A parallel Gauss-Seidel algorithm for sparse matrices is proposed in [4] and [1]. The main idea is to gather independent groups of constraints, which could then be processed in parallel. Constraints which could not be put in a group without adding dependencies are put in a last group, internally parallelized based on a graph-coloring approach. While this method produces nice speedups with few processors, it relies on the sparsity of the matrix to extract parallelism, which on massively parallel architectures might not be enough. Moreover, expensive pre-computations are required when the sparsity pattern changes. A similar idea can be used effectively for rigid body simulations with a very large number of objects [5]. In this case most constraints are independent of each other, allowing to easily extract group of independent constraints. Parallel computations can then be achieved within each group.

In some applications multiple LCP have to be resolved in parallel, such as when detecting collision between convex objects. [6] parallelize the resolution of a small LCP and relies on solving many instances of the problem in parallel. This removes the need for global synchronizations, and thus allows to efficiently exploit all processors in the GPU.

If we relax the strict precedence relationships in the Gauss-Seidel method [7], more parallelism can be extracted even for highly coupled problems. As we add more processing units, constraints are processed in parallelrather than sequentially, increasingly tending toward a Jacobi algorithm. The main drawback is then that a larger number of iterations is required to obtain an equivalent precision.

C. GPU Architecture and Programming

Initially programming GPU for general purpose computations required the use of graphics-oriented libraries and concepts such as pixels and textures. However, the two major GPU vendors released new general programming models, CUDA [8] and CTM [9]. Both provides direct access to the underlying parallel processors in the GPU, as well as less limited instructions, such as write operations at arbitrary locations. Recently, a multi-vendor standard, OpenCL [10], was released. While no implementation are available yet, its programming model is very similar to CUDA. We will thus base our GPU implementation on CUDA, but the presented algorithms should be applicable to other vendors once OpenCL support is available.

The latest generation of GPUs contains many processors (on the order of hundreds), and exploiting them fully requires creating many parallel tasks (or *threads* in the CUDA terminology). On NVIDIA G80 and GT200 architecture, the computation units are grouped into *multi-processors*, each containing 8 processors. Threads are correspondingly grouped into *thread blocks*, each block being executed on a single multi-processor (which can execute multiple blocks in parallel). A single batch of thread blocks is active at a given time, all executing the same program (*kernel*).

Little control is provided over tasks scheduling within a GPU, and synchronization primitives between thread blocks are very limited. In CUDA, threads can only be synchronized within each thread block, however such synchronization is very efficient. Many blocks are required to use all available processors. Global synchronization can thus only be achieved between kernel invocations, which currently can only be controlled by the CPU. As a consequence, the parallelization strategy used must require as few global synchronization as possible, and must launch a very high number of tasks in-between. More details on the programming issues of this architecture are discussed in [11].

D. GPU-based Linear Solvers

While to our knowledge no attempt has been made yet to parallelize a large dense Gauss Seidel algorithm on a GPU, quite a few other algorithms have been studied, in particular to solve sparse and dense linear systems.

Dense linear algebra routines [12] are provided by NVIDIA, and their careful optimization are well studied [13]. It is well known that due to their regular access patterns, dense linear algebra algorithms are well suited to GPU's architecture. However, due to the amount of data contained in a dense matrix, in most cases the computations are bandwidth-limited. Direct factorization-based solvers have been ported to GPUs [14], [15]. Most of these works rely on blocking strategies to parallelize operations.

Sparse linear algebra is somewhat more difficult to adapt to GPUs, at least for unstructured problems. Several techniques have been proposed [16], [17], [18]. The major issues involve how the matrix is stored (compressed storage formats), and whether blocking is used.

III. Parallel Dense Gauss-Seidel

In order to exploit massively parallel architectures, a large number of parallel tasks must be extracted, which can be challenging for Gauss-Seidel as each constraint must be treated sequentially. In the following, we will describe how to parallelize one iteration of the outer-loop in a Gauss-Seidel algorithm with n constraints. The first two strategies are trivial parallelizations of the original algorithm, while the remaining two are more involved but are able to extract a much higher degree of parallelism.

A. Row Parallel Algorithm

The internal loop of the algorithm computes n products (each corresponding to a row of the system matrix), which

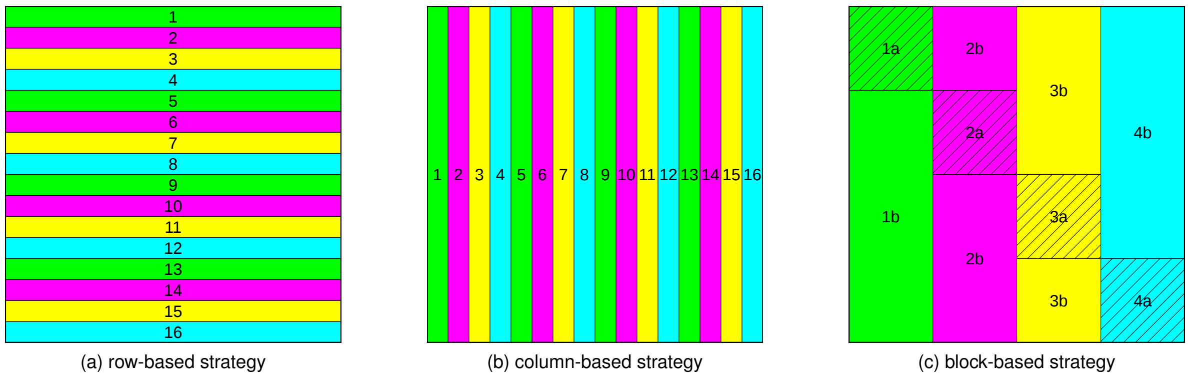


Figure 2. GPU parallelization schemes: each rectangle represents a group of tasks processing a subset of the system matrix (16×16 in this example).

can be evaluated in parallel. Their results must then be combined. This computation is similar to a dot-product and can be parallelized using classical approaches such as a recursive reduction. We need to wait for the computation of a full row in order to update the corresponding constraint and start computing the next row. This requires n global synchronizations for each iteration, plus additional synchronizations that might be required for the reductions.

Fig. 2a presents the task groups created by this strategy. In-between global synchronizations, only n parallel tasks can be launched.

B. Column Parallel Algorithm

We can eliminate the recursive reduction step in the previous strategy by creating groups of tasks corresponding to the columns of the matrix, as shown in Fig. 2b. This requires an additional temporary vector t of size n to store partial accumulations on each row. Within a column, each task can accumulate its contribution independently inside t, eliminating the synchronizations previously required by the reduction. However, this strategy still requires n global synchronizations, with only n parallel tasks in-between.

C. Block-Column Parallel Algorithm

In CUDA, global synchronizations require expensive overheads. In order to reduce their number, a much faster synchronization primitive is provided, but it can only be used within thread blocks. If we create only one such a block, we would be able to synchronize all computations without CPU intervention, but we would only be able to exploit a small fraction of the processors in the GPU. As the system matrix is dense in our application, we cannot rely on finding subsets of the constraints that are independent from each other and thus can be treated in parallel.

We can organize our n constraints in groups of m constraints, producing $g = \lceil n/m \rceil$ groups. If we try to use global synchronizations only between groups, we still need to wait for the evaluation of the first constraint before computing its contribution on the second constraint within each group. However, we don't need to compute the contributions to the constraints on the other groups in order to update all constraints inside the group. This allows us to propose the algorithm presented in Fig. 3.

This strategy requires two tasks groups per group of constraints: first to compute the block on the diagonal of the matrix, then to process the rest of the columns. This corresponds to the a and b blocks in Fig. 2c.

Thanks to the groups, only $2g$ global synchronizations are required by this strategy. Moreover, while the first step only creates m parallel tasks, the second step launches $m(n-m)$ tasks, nearly m times as many as the previous strategies.

D. Atomic Update Counter Parallel Algorithm

The previous strategies rely only on a combination of global synchronization and fast barriers within thread groups. While using blocks allowed to reduce the reliance on global synchronizations, they are still used and thus limit the scalability of the algorithm. In order to completely remove the need for global synchronization, another primitive must be used to insure correct ordering of computations.

We can see that in order to start computing a given block on the diagonal of the system matrix, the computations of all the other blocks on the same line must be completed, which for the block to the left of the diagonal requires that the computation of the previous diagonal block must be finished. Thus, for any parallelization schemes respecting all data dependencies, at any given time only one diagonal block can be processed. By storing in a shared memory location an integer $counter$ storing how many diagonal blocks have been processed, we can implicitly deduce whether a given value x_j is up-to-date in order to process a given element a_{ij} for iteration $iter$ of the algorithm :

$$\begin{cases} counter > \lceil n/m \rceil (iter - 1) + \lfloor j/m \rfloor, & \text{for } j > i \quad (4) \\ counter > \lceil n/m \rceil iter + \lfloor j/m \rfloor, & \text{for } j < i \quad (5) \end{cases}$$

(4) expresses dependencies of blocks in the upper triangular part of the matrix, where values from the previous iteration are used, whereas (5) relates to blocks in the lower triangular

```
Data: $\epsilon$            // shared real value storing the current residual
repeat
    $\epsilon = 0$
    for $j_g = 0$ to $g - 1$ do
        $i_g = j_g$                             // (a) block on the diagonal
        parfor $i_t = 1$ to $m$ do
            $i = i_g m + i_t$
            for $j_t = 1$ to $m$ do
                $j = j_g m + j_t$
                if $i = j$ then
                    $\tilde{x}_i = x_i$
                    $x_i = -(b_i + t_i)/a_{ii}$
                    if $x_i < 0$ then $x_i = 0$
                    $t_i = 0$
                    $\epsilon = \epsilon + |\tilde{x}_i - x_i|$
                else
                    $t_i = t_i + a_{ij} * x_j$
                barrier
        parfor $(i_t, j_t) = (1, 1)$ to $(n - m, m)$ do
            $i = i_t$                           // (b) non-diagonal blocks
            // skip block on the diagonal
            if $i_t \geq j_g m$ then $i = i_t + m$
            $j = j_g m + j_t$
            $t_i = t_i + a_{ij} * x_j$
until $\epsilon < \epsilon_{converge}$
```

Figure 3. Block-based parallelization of one iteration of the outer-loop in the Gauss-Seidel method. Depending on the hardware architecture, each parfor will be translated into parallel groups of threads, or executed as small loops with a given processor. The barrier primitive applies to the parent inner-most parfor loop. There is an implicit global synchronization in-between out-most parfor loops.

```
Data: $\epsilon$            // shared real value storing the current residual
Data: $counter = 0$         // shared atomic counter storing how
                            // many diagonal blocks have been processed
parfor $(iter, i_g) = (0, 0)$ to $(g - 1, iter_{max} - 1)$ do
    parfor $i_t = 1$ to $m$ do
        $i = i_g m + i_t$
        $t_i = 0$
    for $j_g = i_g + 1$ to $g - 1$ do // (a) blocks after the diagonal
        wait($counter > (iter - 1) * g + j_g$)
        parfor $(i, j) = (1, 1)$ to $(m, m)$ do
            $(i, j) = (i_g m + i_t, j_g m + j_t)$
            $t_i = t_i + a_{ij} * x_j$
    for $j_g = 0$ to $i_g$ do           // (b) blocks before the diagonal
        wait($counter > iter * g + j_g$)
        parfor $(i_t, j_t) = (1, 1)$ to $(m, m)$ do
            $(i, j) = (i_g m + i_t, j_g m + j_t)$
            $t_i = t_i + a_{ij} * x_j$
    if $counter = iter_{max} * g$ then return $\epsilon$
    $j_g = i_g$                               // (c) block on the diagonal
    if $i_g = 0$ then $\epsilon = 0$
    for $i_t = 1$ to $m$ do
        $i = i_g m + i_t$
        parfor $j_t = 1$ to $m$ do
            $j = j_g m + j_t$
            if $i \neq j$ then $t_i = t_i + a_{ij} * x_j$
        $\tilde{x}_i = x_i$
        $x_i = -(b_i + t_i)/a_{ii}$
        if $x_i < 0$ then $x_i = 0$
        $\epsilon = \epsilon + |\tilde{x}_i - x_i|$
    // notify blocks waiting on this value
    if $(i_g = g - 1)$ and $(\epsilon < \epsilon_{converge})$ then
        $counter = iter_{max} * g$
    else
        $counter = iter * g + i_g$
```

Figure 4. Parallelization of the Gauss-Seidel method using an atomic counter to insure that the computations ordering respects the dependencies.

part of the matrix, requiring values updated during the current iteration. If the underlying hardware can provide an atomically-updated counter while offering sufficient memory ordering guarantees (i.e. all writes prior to an update of the counter will be visible to other threads before the counter update), then it can be used to remove all other synchronizations. The resulting parallel Gauss-Seidel algorithm is detailed in Fig. 4 and illustrated in Fig. 5. Note that initially the dependencies introduce large delays. However if all threads are then executed at the same speed, they will keep their respective offset and should rarely have to wait again.

IV. Implementation and Results

Two implementations of the proposed algorithms were developed and evaluated on multi-core CPUs as well as NVIDIA GPUs.

A. GPU Implementation using CUDA

For each group of tasks as represented in Fig. 2, a batch of thread blocks is executed. Each thread block contains 64 threads for the first 3 strategies, and 16×8 for the graph-based approach. Each thread handles one element of the system matrix (two for the graph approach).

For the row-based strategy, the reduction is done locally in each thread block, and then the $n/64$ values are downloaded back to the CPU for the final accumulation. This proved faster than using an additional kernel invocation for the matrix sizes we used (up to 10000 constraints).

To implement the block-based strategy, we use groups of $m = 64$ columns. The first kernel, corresponding to step *(a)* of algorithm 3, is executed with a single thread block, containing 64 threads. It contains a loop over the 64 variables of the current group, synchronized using *syncthread* CUDA operations. The second kernel, corresponding to step *(b)*, is executed with $n - 64$ blocks, each of 64 threads. Each block computes the contribution of the constraints of the current group to a specific constraint in another group. While the first kernel is only able to exploit a single multi-processor, the computations executed by the second kernel dominate the run-time more and more as the size of the system increases. This allows to exploit all multiprocessors of the GPU, using for instance $983,040$ threads for a 1024×1024 matrix. However, in this case we are still requiring 32 kernel invocations per Gauss-Seidel iteration.

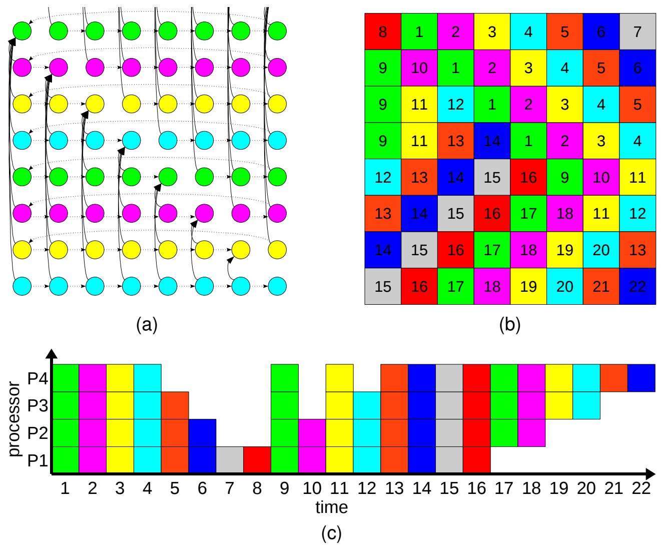


Figure 5. Graph based parallel Gauss-Seidel applied to a 8×8 block matrix. (a) Execution graph with 4 threads (dashed arrows). Vertical solid arrows represent dependencies between computations. (b) Order of execution on 4 processors, neglecting latencies issues. (c) Scheduling of computations on each processor over time for the first iteration.

The final graph-based strategy, removing all global synchronizations, is actually implemented as a single kernel invocation. Each iteration of the out-most parfor loop (line 1 of algorithm 4) is executed within a 16×8 thread group (each thread handling two values in a 16×16 bloc), and scheduled in a round-robin manner on the available multiprocessors. As the underlying programming model does not offer any guarantees as to the order of invocation of thread blocks, this scheduling is done manually. The achieved performances will be highly affected by the implementation of the wait operations, as all synchronizations are based on it. We implemented and tested 3 variants:

1) **graph** : The first thread of each thread block executes a simple spin-loop constantly reading the shared counter, while the other threads are stopped within a *syncthread* CUDA operation.
2) **graph-2** : During the spin-loop, if after reading the shared counter the dependency is not satisfied, then all threads process one block from the next block-line to be handled, provided its own dependency is satisfied.
3) **relaxed** : wait are simply ignored. This actually changes the semantic of the algorithm [7], but guarantees that all processors are computing at full-speed.

The second implementation is equivalent to processing two block-lines at the same time, but giving a priority to the first one. This can remove some of the synchronization overheads. For example, let's suppose we have t thread blocks processing a matrix with $4t$ block-lines. Each thread block will compute one every t block-line. When a thread starts a new line, the first $3t$ blocks will not require any waits, as the required dependencies where already required to compute the previous line completed by this thread. However, the last quarter of the blocks will possibly introduce delays, as they are waiting for values computed quite recently. However, once we reach this part, we know that half of the next line that we will have to compute is already ready, as it uses the same dependencies as this line. This means that instead of wasting time and bandwidth spin-looping on the shared counter, we can compute blocks from the next line. We need to be careful however to restart computations on the first line as soon as it is ready, otherwise we could delay the computation of the final diagonal block, which will introduce even more delays for the threads waiting for it.

B. Multi-threaded CPU Implementation

Until the number of cores of CPUs increase by an order of magnitude, parallelizing an algorithm on a single computer only requires extracting a few parallel tasks, and not tens of thousands as for GPUs. As a result, our CPU implementation only parallelizes the out-most parallel loop of each algorithm, corresponding approximately to all the threads executed in a single GPU multiprocessor. Local barriers are thus not necessary. Global synchronizations are handled by an atomic counter on which each thread spin-loops until it reaches the number of expected threads. The atomic counter for the graph-based strategy is implemented similarly.

In order to support larger-scale platforms, consisting of more than a couple of processors, non uniform memory architectures (NUMA) must be considered. To that end, at initialization each thread allocates and copies the part of the matrix it will work on, to insure optimal accesses locality.

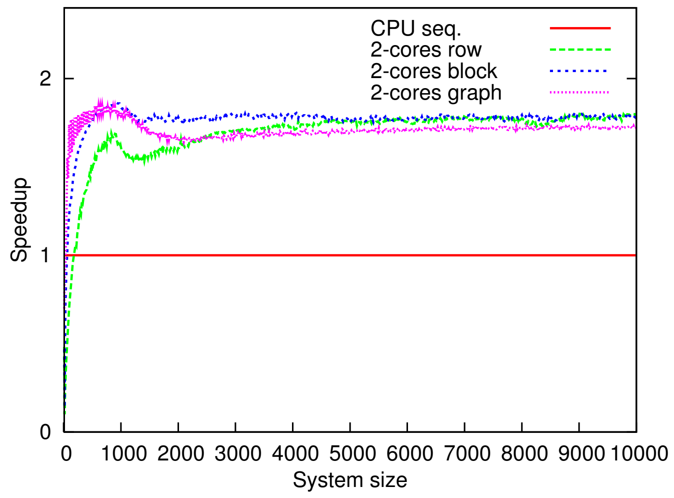
C. Performance Measurements

We measured the time required to solve LCP of different sizes, compared to a reference CPU-based implementation. The following architectures were used:

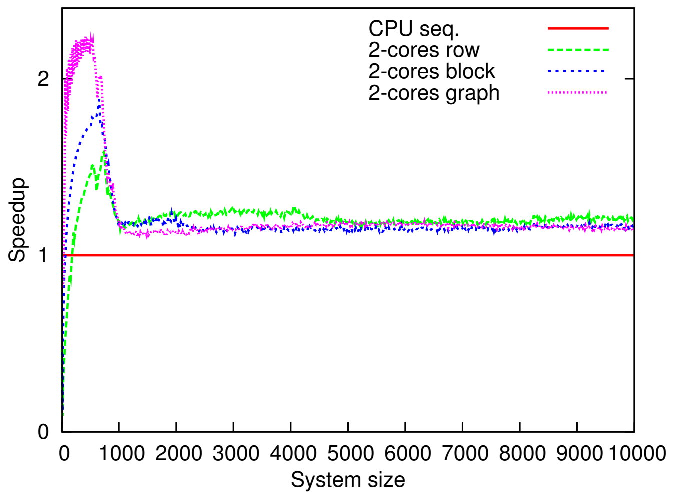
- A single *dual*-core Intel®Core™2 Duo CPU E6850 at 3.00 GHz (2 cores total)
- A single *quad*-core Intel®Core™2 Extreme CPU X9650 at 3.00 GHz (4 cores total)
- A *bi quad*-core Intel®Core™2 Extreme CPU X9650 at 3.00 GHz (8 cores total)
- An *octo* dual-core AMD®Opteron™ Processor 875 at 2.2 GHz (16 cores total)
- A NVIDIA®GeForce™ GTX 280 GPU at 1.3 GHz (30 8-way multi-processors, 240 cores total)

The Intel-based architectures all use an uniform memory layout, whereas the *octo* is a NUMA with 4 nodes, each linking a pair of CPU to 8 GB of RAM.

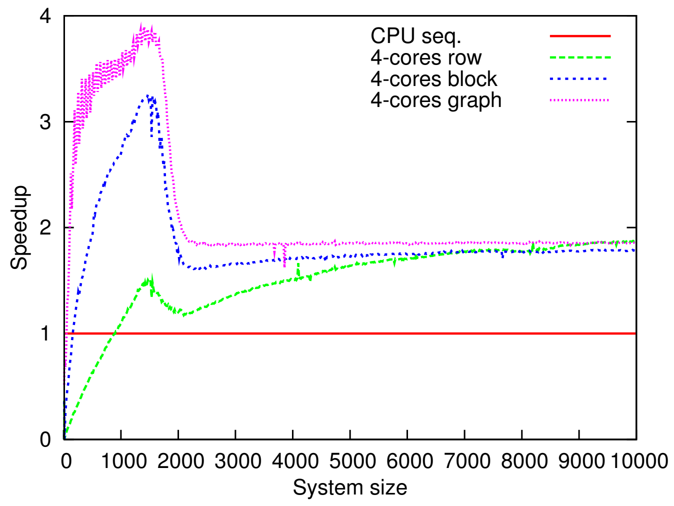
To remove dependency on the exact matrices solved, the number of iterations in the Gauss-Seidel algorithm is fixed at 100. However the achieved accuracy is still computed and transmitted back at each iteration, as it would otherwise remove synchronization overheads associated with it.



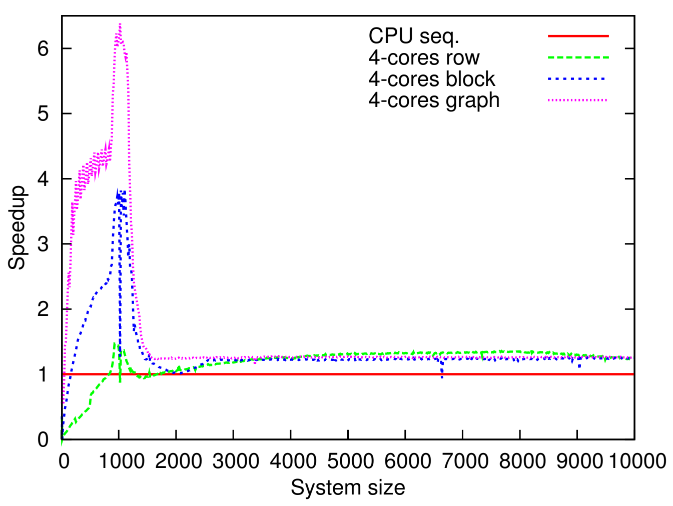
(a) dual-core, single-precision



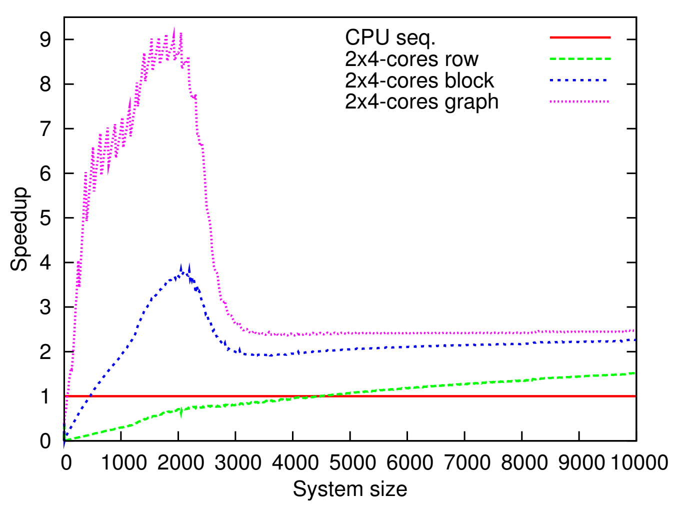
(b) dual-core, double-precision



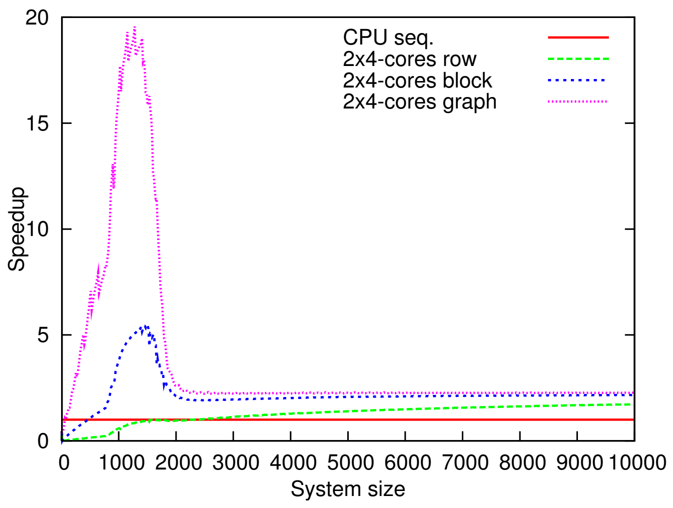
(c) quad-core, single-precision



(d) quad-core, double-precision

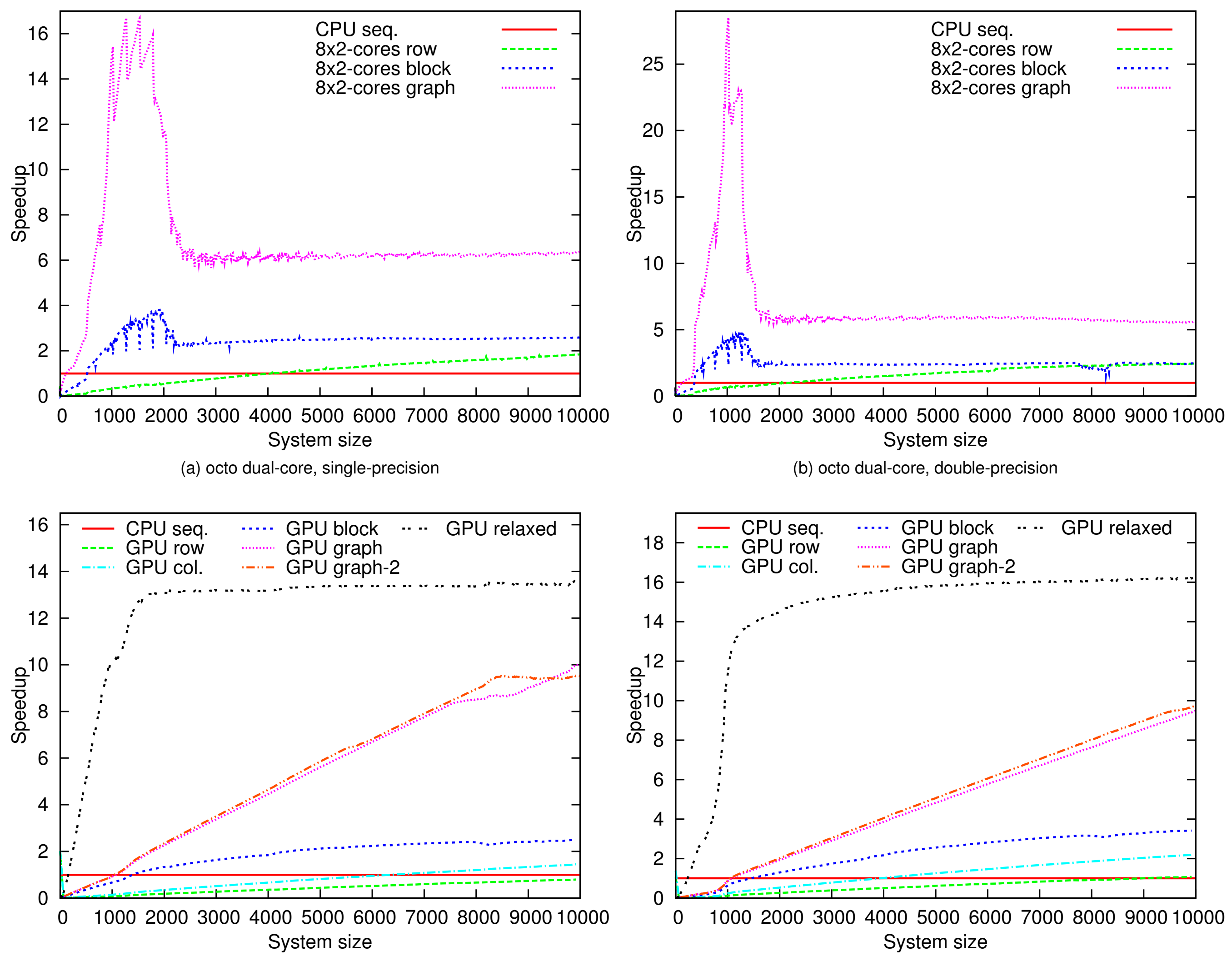


(e) bi quad-core, single-precision



(f) bi quad-core, double-precision

Figure 6. Computation time measurements on ubiquitous multi-core architectures, compared to optimized sequential algorithm on a single CPU core. The *row* algorithm is presented in section III-A, *block* in section III-C, and *graph* in section III-D.



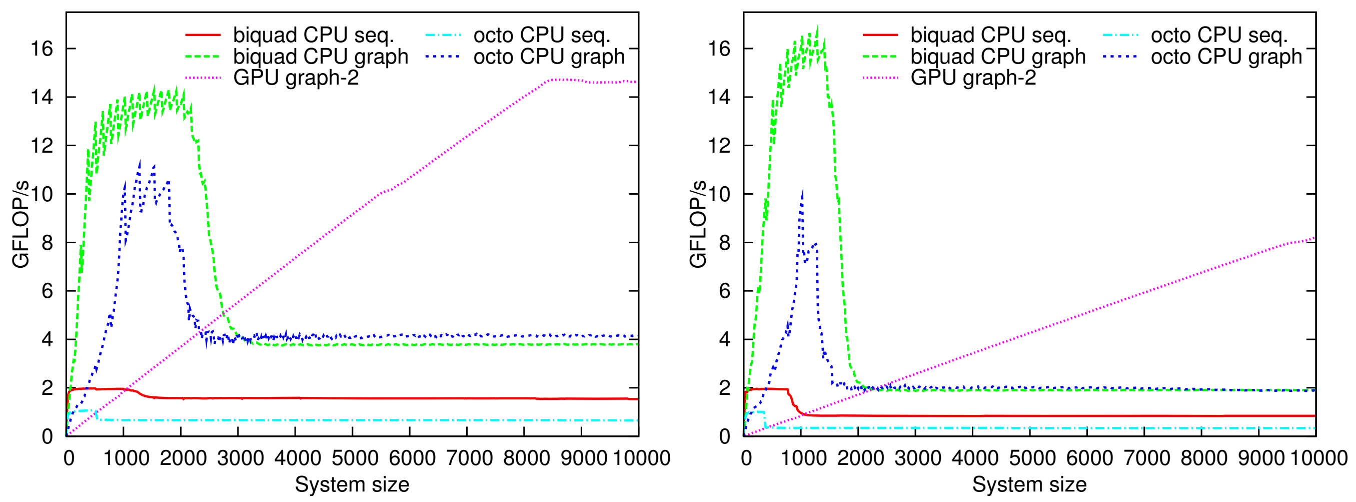
(a) octo dual-core, single-precision (b) octo dual-core, double-precision (c) GTX 280 GPU, single-precision (d) GTX 280 GPU, double-precision

Figure 7. Computation time measurements on many-core architectures.

Figs. 6 and 7 present the speedups measured on each architecture. On all CPU-based architectures, we observe super-linear speedups (up to 6× on *quad*, 18× on *biquad*, and 28 × on *octo*) for matrices up to about 2000 × 2000. It can be explained by the fact that matrices of such size do not fit anymore in the cache of a single processor, but still do once split among the available processors. In this range, only the graph-based strategy is able to achieve best performances. Once the matrix size becomes bigger than the caches, the memory bus becomes the bottleneck. In fact, even on the 8-core *bi-quad* computer, parallelizing over 8 threads is hardly faster than 2 threads. Only NUMA systems such as *octo* are able to achieve better scalability, with speed-ups stabilizing around 6× for the graph-based strategy, which is close to peak considering the hardware is made of 8 processors.

On GPU, the simple *row* and *col* parallelizations are not able to really exploit the available computation units. The *block* algorithm surpasses the CPU sequential implementation for matrices of more than 1400 × 1400, but it does not improve it much, up to 2.5× for single-precision and 3.4× for double-precision. The first *graph* implementation achieves similar performances for small matrices. It becomes faster for large problem size, reaching a speedup of 10× for single-precision computation on a matrix of size 10000×10000, and 9.5× for double-precision. The *graph-2* algorithm, overlapping waits with computations for the next line, is able to improve performances by up to 10 percents for single-precision matrices of size 8500×8500, but it then becomes less efficient for larger matrices.

Compared to the *relaxed* algorithm, we can see that the overhead related to dependencies checks is still significant.



(a) single-precision

(b) double-precision

Figure 8. Giga floating-point operations per seconds (GFLOP/s) achieved on each architecture.

The computation speed achieved by the fastest algorithms on the many-cores and GPU architectures is presented in Fig. 8. The GPU is able to surpass the sequential CPU performances except for very small matrices. However parallel CPU implementations are the fastest for single-precision matrices up to 2800×2800 and double-precision matrices up to 2300×2300. For larger matrices, the GPU becomes faster, exploiting more and more of its 240 computation units and benefiting from its massive internal bandwidth.

V. Application

For medical training and intervention planning, we developed using the SOFA framework [19] an endovascular simulator, modeling catheters and coils manipulated inside arteries to treat conditions such as aneurysms (Fig. 9). As these deformable objects slide along the vessel borders, a high number of contacts must be accurately modeled and resolved. This proved a challenge both in terms of accuracy and performance.

Frictionless contacts can be modeled based on Signorini's law, indicating that there is complementarity between the interpenetration distance ${}^{n}\delta$ and the contact force ${}^{n}\mathrm{f}$:

$$0 \leq {}^{n}\delta \perp {}^{n}\mathrm{f} \geq 0 \tag{6}$$

To compute the forces that need to be applied to reach a contact-free configuration, we solve the following LCP:

$$\begin{cases} \boldsymbol{\delta} = \mathbf{HCH}^{T}\mathbf{f} + \boldsymbol{\delta}^{free} \\ 0 < \boldsymbol{\delta} \perp \mathbf{f} > 0 \end{cases} \tag{7}$$

Where $\mathbf{C}$ is the mechanical compliance matrix, and H the transformation matrix relating contacts to mechanical degrees of freedom.

We use an extended formulation of this LCP [20] in order to include friction computation, creating three constraints per contact (the normal penetration (6) and the frictions in two tangent directions). This LCP is then solved using the Gauss-Seidel algorithm.

This application requires solving at each frame a LCP containing between 600 and 3000 constraints (depending on the number of contacts and whether friction is used). In this range, the CPU parallelization is currently the most efficient, achieving a speedup between $5\times$ and $20\times$ for this step, leading to an significant decrease in computation times. This GPU is able to achieve accelerations up to $3\times$, but most importantly it should allow us to increase the complexity of our simulations in the future, while limiting the time taken by the constraints solving step.

VI. Conclusion

We presented a new parallel Gauss-Seidel algorithm, allowing to efficiently exploit all the processors available in the latest generation of CPUs and GPUs. Contrary to previous works, this approach does not require a spare system matrix in order to extract enough parallelism. While a dense Gauss-Seidel algorithm introduces many constraining dependencies between computation, we showed through our experimental studies that by carefully handling them and overlapping computations we are able to fully exploit up to 240 processing units for large problems, achieving speedups on the order of $10\times$ compared to a CPU-based sequential optimized implementation.

This algorithm was used in an interventional radiology medical simulator. It allowed to significantly reduce the time used for the constraint solving step. However, the other steps of the simulation are now prevalent in the achieved speed, particularly collision detection and the construction of the mechanical compliance matrix. Thus the overall speedup is currently only up to $2\times$. Parallelizing the remaining steps

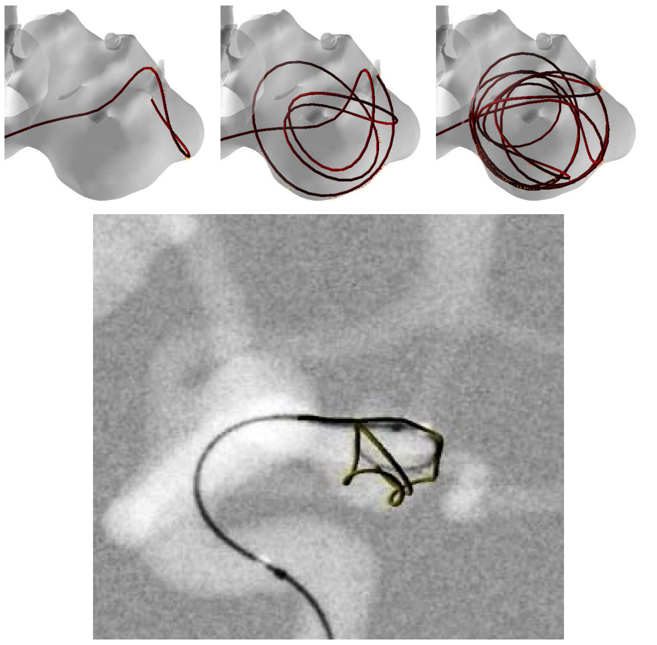


Figure 9. Coil embolization simulation. Sequence of 3 steps while deploying a coil inside an aneurysm, and augmented reality control view.

should allow us to benefit more in the future, to be able to increase the achieved realism and advance toward patient-specific pre-operative planning simulations.

As a future work, it would be interesting to investigate hybrid algorithms for block-sparse matrices, such as when multiple deformable bodies are in contact. For such cases, we could imagine a strategy where dense blocks containing constraints attached to the same objects are handled by our dense parallel algorithm, with an extra layer of higher-level parallelism (multi-GPUs or in a cluster) handling multiple blocks involving constraints linked to disjoint objects.

Acknowledgments

We would like to thank NVIDIA for providing an engineering sample of a GT200 GPU.

References

[1] M. F. Adams, "A distributed memory unstructured Gauss-Seidel algorithm for multigrid smoothers," in *Proc. of the 2001 ACM/IEEE conference on Supercomputing*, 2001.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, 1994.

[3] K. Murty, *Linear Complementarity, Linear and Nonlinear Programming*. Internet Edition, 1997.

[4] D. P. Koester, S. Ranka, and G. C. Fox, "A parallel Gauss-Seidel algorithm for sparse power system matrices," in *Proc. of the 1994 ACM/IEEE conference on Supercomputing*, 1994, pp. 184–193.

[5] A. Bond, "Havok FX: GPU-accelerated physics for PC games," in *Proc. of Game Developers Conference 2006*, 2006.

[6] P. Kipfer, "LCP algorithms for collision detection using CUDA," in *GPU Gems 3*, 2007, ch. 33, pp. 723–740.

[7] M. Renouf, F. Dubois, and P. Alart, "A parallel version of the non smooth contact dynamics algorithm applied to the simulation of granular media," *J. Comput. Appl. Math.*, vol. 168, no. 1-2, pp. 375–382, 2004.

[8] NVIDIA Corporation, "NVIDIA CUDA compute unified device architecture programming guide," 2007.

[9] M. Peercy, M. Segal, and D. Gerstmann, "A performance-oriented data parallel virtual machine for GPUs," in *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, 2006, p. 184.

[10] Khronos OpenCL Working Group, *The OpenCL Specification Version: 1.0*, A. Munshi, Ed. The Khronos Group, 2008.

[11] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008, pp. 73–82.

[12] NVIDIA Corporation, "NVIDIA CUBLAS library," 2007.

[13] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proc. of the 2008 ACM/IEEE conference on Supercomputing*, 2008, pp. 1–11.

[14] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha, "LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware," in *Proc. of the 2005 ACM/IEEE conference on Supercomputing*, 2005.

[15] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. van de Geijn, "Solving dense linear algebra problems on platforms with multiple hardware accelerators," in *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2009.

[16] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse matrix solvers on the GPU: conjugate gradients and multigrid," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 917–924, 2003.

[17] L. Buatois, G. Caumon, and B. Lévy, "Concurrent number cruncher - a GPU implementation of a general sparse linear solver," *International Journal of Parallel, Emergent and Distributed Systems*, to appear.

[18] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA, Technical Report NVR-2008-004, Dec. 2008.

[19] J. Allard, S. Cotin, F. Faure, P.-J. Bensoussan, F. Poyer, C. Duriez, H. Delingette, and L. Grisoni, "SOFA – an open source framework for medical simulation," in *Medicine Meets Virtual Reality (MMVR'15)*, Long Beach, USA, 2007, http://www.sofa-framework.org/.

[20] C. Duriez, F. Dubois, A. Kheddar, and C. Andriot, "Realistic haptic rendering of interacting deformable objects in virtual environments," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 1, pp. 36–47, 2006.