Eurographics Symposium on Parallel Graphics and Visualization (2007)
Jean M. Favre, Luis Paulo dos Santos, and Dirk Reiners (Editors)

Exploiting Parallelism in Physically-Based Simulations on Multi-Core Processor Architectures

Bernhard Thomaszewski[1] Simon Pabst[1] Wolfgang Blochinger[2]

[1]WSI/GRIS, Universität Tübingen, Germany
[2] Symbolic Computation Group, Universität Tübingen, Germany

Abstract

As multi-core processor systems become more and more widespread, the demand for designing efficient parallel algorithms propagates also into the field of computer graphics. This is especially true for the physically-based simulation, which is notorious for expensive numerical methods. In this paper we explore possibilities for accelerating these algorithms on modern multi-core architectures. As an application we focus on physically-based cloth simulation. In this context, two distinct problems can be identified: the physical model and the collision handling stage – both bearing potential bottlenecks for the simulation. From the parallelization point of view these two components are substantially different. The physical model can be treated efficiently using static problem decomposition. The collision handling problem, however, requires a different approach, due to its dynamically changing structure. We address this problem using multi-threaded programming with fully dynamic task decomposition. Furthermore, we propose a new task splitting approach based on a robust work estimate. The associated data is derived from temporal coherence. Altogether, the combination of different parallelization techniques leads to a concise and yet versatile framework for highly efficient physical simulation.

Categories and Subject Descriptors (according to ACM CCS): C.1.4 [Processor Architectures]: Parallel Architectures, G.1.3 [Numerical Analysis]: Numerical Linear Algebra, G.4.5 [Mathematical Software]: Parallel and Vector Implementations, I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

1. Introduction

Physically-based simulation is an important research area in computer graphics and has a broad range of applications. The most prominent examples are fluid, soft body, and cloth simulation. All of these applications utilize computationally intensive methods, and runtimes for realistic scenarios are often excessive. Of course, for the physically more accurate variants this situations is further aggravated. Most of the computation time is spent on two stages, time integration and collision handling. In the following, we will therefore consider these two major bottlenecks, which are present in almost every physically-based simulation. Although we will focus on cloth simulation in this work, the techniques proposed herein transfer to many other applications.

1.1. Implicit time integration

Oftentimes, the physical model at the centre of a specific simulator gives rise to stiff differential equations with respect to time. Due to stability reasons implicit schemes are widely accepted as the method of choice for numerical time integration (cf. [BW98]). Implicit schemes require the solution of a (non-)linear system at each time step. As a result of the spatial discretization, the matrix of this system is usually very sparse. There are essentially two alternatives for the solution of the system. One is to use an iterative method like the popular conjugate gradients (cg) algorithm [She94]. Another is to use a direct solver based on some sort of factorization. The cg-method is more popular in computer graphics as it offers much simpler user interaction, alleviates the integration of arbitrary boundary conditions and allows balancing accuracy against speed. We will therefore focus on the cg-method in this work.

1.2. Collision Handling

For realistic scenes it is necessary to model the interaction of deformable objects with their virtual environment. This involves the detection of proximities (collision detection) and the reaction necessary to keep an intersection free state (collision response). In the remainder, we refer to these two components collectively as *collision handling*. We usually distinguish between external collisions (with other objects in the scene) and self-collisions. For each of these types different variants of algorithms are usually used. Even with common acceleration structures (see Sec. 3.2) these algorithms are still computationally expensive. For complex scenarios with complicated self-collisions the collision handling can easily make up more than half of the overall computation time. It is therefore a second bottleneck for the physical simulation and hence deserves special attention.

2. Related Work

Parallel Numerics. The parallel solution of large sparse linear systems is a well explored and yet active field in high performance computing. Most of the work from this field is focused on problem sizes which are considerably larger than the ones dealt with in computer graphics. Therefore, usual techniques do not necessarily translate directly to our application area. Generally, good overviews on parallel numerical algebra can be found in the textbook by Saad [Saa03] and the report compiled by Demmel et al. [DHv93]. The parallel implementation of sparse numerical kernels like the ones used in this work were investigated e.g. by Hallaron [O'H97]. Special ordering strategies explored node ordering strategies and programming paradigms for sparse matrix computations. Parallel preconditioning was, however, not considered in these two works.

Parallel Cloth Simulation. Previous research on parallel cloth simulation addressed shared address space [RRZ00, LGPT01, GRR*05] as well as message passing-based architectures [ZFV02, ZFV04, KB04, TB06]. Since this paper specifically deals with multi-core CPUs, we will restrict our discussion of related work to approaches designed for shared address space machines.

Lario et al. [LGPT01] describe the parallelization of a cloth simulator which employs multilevel techniques. The authors focus on the cloth motion modeling stage and do not address parallel collision detection. Particularly, they present a comparison between message passing-based and thread-based parallelization of multilevel methods on different shared address space architectures.

Romero et al. [RRZ00] present a parallel cloth simulator designed for non-uniform memory access (NUMA) architectures. Their work addresses parallelization of the physical modeling phase and of the collision handling phase. While the approach taken for the modeling phase is similar to our work, the way collision handling is carried out differs significantly. In their work, parallel collision handling is implemented by a data-parallel strategy which partitions lists of potentially colliding primitives. These lists are maintained by heuristics. Bounding volume hierarchy (BVH) tests are only carried out to initialize the lists and in case the size of the lists exceeds a given threshold. By contrast, our approach to collision handling aims at robustness and accuracy. As a consequence, we perform a complete series of BVH tests in every collision handling phase. However, this strategy requires parallelizing the BVH testing procedure. Due to the hierarchical and irregular nature of these tests, we apply a task-parallel method which is based on fully dynamic problem decomposition.

The work of Gutierréz *et al.* [GRR*05] draws special attention to histogram reduction computations which can be found at the core of numerical simulation codes, like cloth simulation. The authors present a framework for partitioning-based methods on NUMA machines which exploits data affinity. In the context of this framework, several methods for parallel reduction are applied to the force computation loop of a cloth simulator and compared with each other. While their work concentrates on optimizing a specific aspect, our approach encompasses all of the computation intensive components of physically-based simulation.

3. Physically-based Cloth Simulation

As already stated in the introduction, the methods described in this article apply to any specific approach provided it uses implicit time integration for the physical model and bounding volume hierarchies for the collision handling stage. Details on the modules employed in this work are given below.

3.1. Physical Model

The basis for the physical model is a continuum mechanics formulation of linear elasticity theory [Cia92]. The central quantities in this case are *strain*, which is a dimensionless deformation measure, and *stress*, which is a resulting force per area. These two variables are related to each other through a so called constitutive law, which in our case is simply linear. The resulting partial differential equation (PDE) is discretized using a linear finite element approach as described in [EKS03]. For dynamic simulation, inertia effects have to be included as well as viscosity and possibly external forces. Finally, the problem of implicit time integration can be defined as follows. Let Y be a vector of size $6n$ where n is the number of nodes in the system. Y is the concatenation of positions and velocities of the nodes. The first order implicit Euler integrator now seeks to find $Y(t+h)$ such that

$$Y(t+h) = Y(t) + hf(t, x(t+h), v(t+h)) + O(h^2) \qquad (1)$$

where f is the derivative of the state vector Y. Generally, f is a nonlinear function in terms of Y and the system has to be solved using Newton's method. In any case this breaks down to (repeatedly) solving linear systems. In our particular case, Eq. (1) is linear and we therefore only need to solve one linear system per time step. Our approach to the parallel solution of this system using the method of conjugate gradients is described in Sec. 4.

3.2. Collision Handling

Collision Detection. As a first step, possible interferences have to be detected for the deformable objects in the scene. Since all objects are represented as polygonal meshes, this could be accomplished by testing every pair of primitives (i.e., polygons) geometrically for intersection. Because the average runtime of this naive approach is unacceptably high, BVHs are usually used for acceleration [THM*05]. In this way, non-intersecting parts are quickly ruled out for a given object pair. Such a hierarchy consists of two components: a tree representing the topological subdivision of the object into increasingly finer regions and bounding volumes enclosing the geometry associated with every node in the tree. In our implementation we use discrete oriented polytopes (k-DOPs) as bounding volumes (see [KHM*98, MKE03]).

Testing two objects for interference using BVHs is a recursive process. First, the bounding volumes associated with the roots of the two hierarchies are tested for overlap. Only if they overlap, the respective children are tested recursively against each other. Finally, the leaves of the tree need to be checked for intersection using exact geometric tests. If a test signals close proximity or intersection, an appropriate collision response has to be generated next.

Collision Response. Generally speaking, the task of the collision response stage is to prevent intersections. There are various methods to achieve this, ranging from motion constraints over repulsion forces to stopping impulses. Constraints are simple to enforce and do a good job when it comes to preventing intersections with external objects in rather simple scenes. However, releasing constraints is usually cumbersome and often leads to nodes being arbitrarily fixed at some point in space. This is particularly disturbing for self-collisions and literally breaks the simulation. In our implementation we therefore use a combination of repelling forces and stopping impulses (see [BFA02]). If the distance between two approaching objects falls below a certain threshold, we apply a repulsion force. If the objects cannot be stopped in this way during the next few time steps, we apply stopping impulses which reliably prevent imminent intersections. While this is a straightforward concept in the sequential case, there are some important implications for parallel implementations. We will discuss these issues in Sec. 5.

4. Parallel Solution of Sparse Linear Systems

We assume that a sparse linear system of the form $\mathbf{Ax} = \mathbf{b}$ is to be solved up to some residual tolerance using the cg-method. The number of necessary iterations and therefore the speed of convergence depends on the condition number of the matrix $\mathbf{A}$. Usually, this condition number is improved using a preconditioning matrix $\mathbf{M}$ leading to a modified system

$$\mathbf{M}^{-1}\mathbf{Ax} = \mathbf{M}^{-1}\mathbf{b},$$

where $\mathbf{M}^{-1}\mathbf{A}$ is supposed to have a better condition number and $\mathbf{M}^{-1}$ is fairly easy to compute. The choice of an appropriate preconditioner is crucial because it can reduce the iteration count substantially. The setup and solution of the linear system now breaks down to a sequence of operations in which (due to their computational complexity) the sparse matrix vector multiplication (SpMV) and the application of the preconditioner are most important. As a basis for the actual parallelization we will consider problem decomposition approaches subsequently.

4.1. Problem Decomposition

In the following, we assume the compressed row storage (CRS) format for sparse matrices in which nonzero entries are stored in an array along with a row pointer and a column index array (see [Saa03]). The most intuitive (and abstract) way to decompose the SpMV operation into a number of smaller sub-problems is to simply partition the matrix into sets of contiguous rows. The multiplication can then be carried out in parallel among the sets. This simple approach has several disadvantages. First, the matrices we deal with are always symmetric (due to the underlying PDE). Hence, only the upper triangular part, including the diagonal, has to be stored. This leads to smaller memory requirements for the data as well as for the index structure. In its sequential version, the resulting numerical kernel is more efficient (cf. [LVDY04]): it visits every matrix entry only once, performing both dot products and vector scalar products. However, the access pattern to the solution vector is not as local as for the non-symmetric case, i.e., entries from different sets need to be written by a single processor. The required synchronization would make a direct parallel implementation of the symmetric SpMV kernel inefficient. Another reason why the above decomposition is inadequate is that it does not take into account two other important components of linear system solution, matrix assembly and preconditioning.

Methods based on domain decomposition are better suited for this case. They divide the input data geometrically into disjoint regions. Here, we will only consider non-overlapping vertex decompositions, which result in a partitioning P of the domain Ω into subdomains Ω_i such that $\Omega = \cup_i \Omega_i$ and $\Omega_i \cap \Omega_j = \emptyset$, for $i \neq j$. Decompositions can be obtained using graph partitioning methods such as Metis [KK96] in our case. An example of this can be seen in Fig. 1, which also shows a special vertex classification. This will be explained in the next section.

4.2. Parallel Sparse Matrix Vector Multiplication

Let $n_{i,loc}$ be the number of local vertices belonging to partition i and let V_i be the set of corresponding indices. These vertices can be decomposed into n_{int} internal vertices and n_{bnd} interface or boundary vertices, which are adjacent to n_{ext} vertices from other partitions (see Fig. 1). If we reorder the vertices globally such that vertices in one partition are enumerated sequentially we obtain again a partitioning of the matrix into a set of contiguous rows. The rows $a_{i,0}$ to $a_{i,n}$

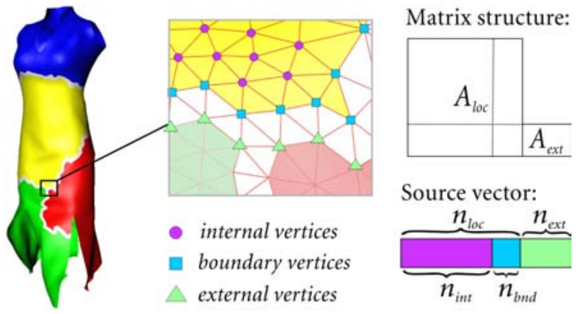

Figure 1: *Decomposition of a mesh into four disjoint partitions indicated by different colours. The associated vertex ordering leads to a special structure of the matrices and the source vector.*

of matrix A where $i \in V_i$ have the following special structure: the block $A_{i,loc}$ defined by $\{a_{lm} | l \in V_i, m \in V_i\}$ and lying on the diagonal of A is symmetric. The nonzero entries in this block describe the interaction between the local nodes of partition i. More specifically, this means that when nodes l and m are connected by an edge in the mesh, there is a nonzero entry a_{lm} in the corresponding submatrix of A. Besides this symmetric block on the diagonal there are further nonzero entries a_{le} where $l \in V_i$ is an interface node and $e \notin V_i$. These entries describe the coupling between the local interface nodes and neighbouring external nodes. The multiplication can be carried out efficiently in parallel if we adopt the following local vertex numbering scheme (cf. [Saa03]). The local vertices are reordered such that all internal nodes come before the interface nodes. For further performance enhancement, a numbering scheme that exploits locality (such as a self avoiding walk [OBHL02]) can be used to sort the local vertices. Then, external interface nodes from neighbouring partitions are locally renumbered as well. Let A_{ext} be the matrix which describes the coupling between internal and external interface nodes for a given partition. Notice that A_{ext} is a sparse rectangular matrix with n_{bnd} rows. With this setup the multiplication proceeds as follows

1. $y(0, n_{loc}) = A_{loc} \cdot x(0, n_{loc})$
2. $y(n_{int}, n_{loc}) = y(n_{int}, n_{loc}) + A_{ext} \cdot x_{ext}(0, n_{ext})$

The first operation is a symmetric SpMV, the second one is a non-symmetric SpMV followed by an addition. Both these operations can be carried out in parallel among all partitions. This decomposition is not only used for the SpMV kernel but also as a basis for the parallel matrix assembly, preconditioner setup and preconditioner application.

4.3. Parallel Preconditioning

In order to make the cg-method fast, it is indispensable to use an efficient preconditioner. There is a broad variety of different preconditioners ranging from simple diagonal scaling (Jacobi preconditioning) to sophisticated multilevel variants. For the actual choice one has to weigh the time saved from the reduced iteration count against the cost for setup and repeated application of the preconditioner. Additionally, one has to take into account how well a specific preconditioner can be parallelized. Unfortunately, designing efficient preconditioners is usually the most difficult part in the parallel cg-method [DHv93]. As an example, the Jacobi preconditioner is very simple to set up and apply even in parallel but the reduction of necessary iterations is rather limited. Preconditioners based on (usually incomplete) factorization of the matrix itself or an approximation of it are more promising. One example from this class is the SSOR preconditioner. It is fairly cheap to set up and leads to the solution of two triangular systems. For the sequential case, this preconditioner has proven to be a good choice in terms of efficiency [HE01]. However, parallelizing the solution of the triangular systems is very hard. Even if it is not possible to decouple the solution of the original triangular systems into independent problems we can devise an approximation with the desired properties. Let $\bar{A}$ be the block diagonal matrix with block entries $A_{ii} = A_{i,loc}$, i.e. the external matrices A_{ext} are dropped from A to give $\bar{A}$. Setting up the SSOR-preconditioner on this modified matrix leads again to the solution of two triangular systems. However, solving these systems breaks down to the solution of decoupled triangular systems corresponding to the $A_{i,loc}$ blocks on the diagonal. This means that they can be carried out in parallel for every partition. For reasons of data locality we use n smaller SSOR preconditioners constructed directly from the $A_{i,loc}$-blocks. Approximating A with $\bar{A}$ means a comparably small loss of information which in turn leads to a slightly increased iteration count. However, in the test cases we performed this increase was small compared to the speedup obtained through parallelization. As a result, the preconditioner scales very well both in terms of setup and application (see Sec. 6.2).

4.4. Optimizations

Besides the points that were treated above a further aspect restricts the efficiency of a parallel implementation of the cg-method. Dense matrix multiplications usually scale very well since they have regular access patterns to memory and a high computational intensity. For the SpMV kernel, however, the computational intensity per data element is rather modest and the locality of data accesses to the source vector is low. The performance of the SpMV algorithm is therefore mostly limited by memory bandwidth and cache performance. This fact is also reflected by our experiments (see Sec. 6.2), which indicate that the optimal speedup using four cores is not reached. This can be attributed to the fact that two cores per processor share the same memory interface. Because of this, it is important to improve data locality and thus cache performance. A good way to achieve this is to exploit the natural block layout of the matrix as determined by the underlying PDE: the coupling between two vertices is described by a 3x3 block – therefore nonzero entries in the matrix occur always in blocks. This blocked data layout already compensates for a lot of the inefficiency. An additional benefit can be achieved using single precision floating point data instead of double precision. This reduces the necessary matrix data (not including index structure) transferred from memory by a factor of two. We found that with only minor modifications even the largest examples did not show stability problems using single precision arithmetic. From our measurements we draw the conclusion that these modifications together are sufficient to yield satisfying speedups (see Sec. 6.2).

5. Parallel Collision Handling

From the parallelization point of view, the problem of collision handling differs substantially from the physical model. Collisions can be distributed very unevenly in the scene and their (typically changing) locations cannot be determined statically. This is why the naive approach of letting each processor care for the collisions of its own partition can lead to considerable processor idling, which seriously affects the overall parallel efficiency. Therefore, a dynamic problem decomposition is mandatory. Compared to previous work aimed at distributed memory architectures [TB06], our basic parallelization strategy is similar. However, the shared-memory setting enables us to set up heuristics exploiting temporal and spatial coherence. In this way, we can effectively control thread creation overhead.

5.1. Basic Problem Decomposition

The recursive collision test of two BVHs can be considered as a depth-first tree traversal. For inducing parallelism, we implemented this procedure using a stack which holds individual tests of two BVs. During the traversal, the expansion of a node yields n additional child nodes. We process one node immediately while the others are pushed onto the stack. The traversal goes on downwards until a leaf is reached. Upward traversal begins by processing elements from the stack. In this way, all of the nodes in the tree are visited. The basic idea to dynamically generate parallelism is now to remove nodes from the stack in an asynchronous way and to create tasks from them. One or more tasks can then be assigned to a thread and executed on an idle core.

Unlike in the distributed memory setting we do not have to care for load balancing explicitly. As long as there are enough threads ready for execution the scheduler will keep all cores busy. However, for problems with high irregularity, like parallel collision handling, it is generally impossible to precisely adjust the amount of logical parallelism to be exploited to the amount of available parallelism (i.e., idle processors). Especially on shared memory architectures, thread creation overhead can considerably contribute to the overall parallel overhead. Therefore, an over-saturation with threads has to be avoided as well.

In our approach, we minimize thread creation overhead on two levels. On the algorithmic level, we employ a heuristics-based approach which prevents threads with too fine a granularity from being generated. On the implementation level, we decouple the process of thread creation and thread execution. The next two paragraphs explain these optimizations in more detail.

5.2. Controlling Task Granularity

For effectively controlling the granularity of a task, we need a good estimate of how much work corresponds to a certain task. The computational cost for carrying out a test in the collision tree is determined by the number of nodes in its subtree. Generally, this number is not known in advance. Because of the inherent temporal locality due to the dynamic simulation we can, however, exploit coherence between two successive time steps. After each collision detection pass we compute the number of tests in the respective subtree for every node in the collision tree using back propagation. This information is then used as a work estimate for tasks in the subsequent collision handling phase.

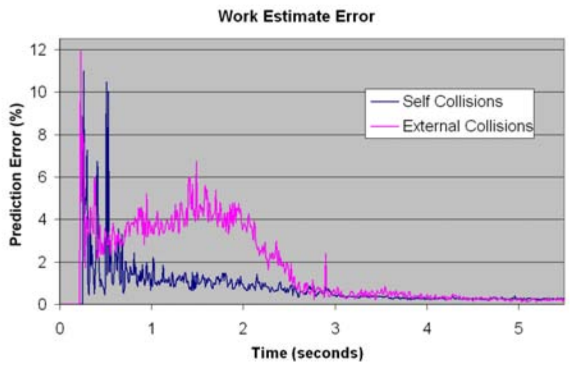

Figure 2: *Work estimate error for scene 2. The diagram shows the deviation from the actual amount of work over time in percent. Even in this very dynamic scene, the temporal coherence is high.*

In this way, we can avoid creating tasks with too small an amount of work. Additionally, we can use this information to determine which tests should be carried out immediately. The error involved in the work estimation is usually very small. This can also be seen in Fig. 2 which shows a comparison of the estimated and the actual work load for 5.5 seconds of simulation (second test scene (see 6.1). For evaluating the benefit arising from this new task splitting scheme, we performed comparisons with two alternative approaches. The first one, being the simplest variant, carries out the test corresponding to leftmost subnode immediately and assigns the remaining subnodes to tasks. The second one is based on randomization, which is a widely adopted paradigm for achieving well-balanced load distribution in parallel applications. In this case, we randomly select the subnode to be treated immediately. The results of these comparisons show that our new scheme is very competitive. While the randomized variant already beats the simple approach, our robust work estimation scheme can even improve on this. It clearly outperforms both of the methods in common scenarios (as e.g. scene 1) and keeps track with the randomized variant even in most demanding applications as e.g. our second test scene (see Fig. 6). This attests to the fact that temporal coherence in dynamic collision detection is a valuable source for performance improvements.

5.3. Implementation

As in our previous work (which addressed distributed memory architectures) we employed the DOTS system platform [BKLW99] for parallelizing collision handling. The interested reader will find a detailed description of the fully dynamic problem decomposition process with the (strict) multithreading parallel programming model provided by DOTS in [TB06].

In order to ensure high performance on shared memory architectures, DOTS employs lightweight mechanisms for manipulating threads. Forking a thread results in the creation of a passive object, which can later be instantiated for execution. Thread objects can either be executed by a pre-forked OS native worker thread or can be executed as continuation of a thread which would otherwise be blocked, e.g., a thread reaching a synchronization primitive.

6. Results

Because the aim of this work is to accelerate computations for physically-based simulations on commodity platforms, we decided to use a system which is easily available at the current time. This system is based on a dual AMD Opteron 270 machine with 2GB of main memory. Each of the Opterons is a dual core processor running at 2GHz. The memory architecture is shared address space, more specifically cc-NUMA (cache-coherent-NUMA).

6.1. Test Scenes

We tested our approach with two scenes, each of them highlighting different aspects. Since the focus is on accelerating commonly used scenarios, we decided to use only moderately large input data. This is an important difference to the distributed memory setting, which traditionally aims at problem sizes exceeding the capacity of a single workstation. The first example (see Fig. 3) is a simulation of a dress worn by a female avatar with a fairly complex geometry (over 27000 vertices). The dress, consisting of roughly 4500 vertices, is pre-positioned around the body and drapes under gravity during one second of simulation. This test scene focuses primarily on the parallel performance of the physical model and on the case of evenly distributed collisions.

The second scene puts special emphasis on self-collisions.

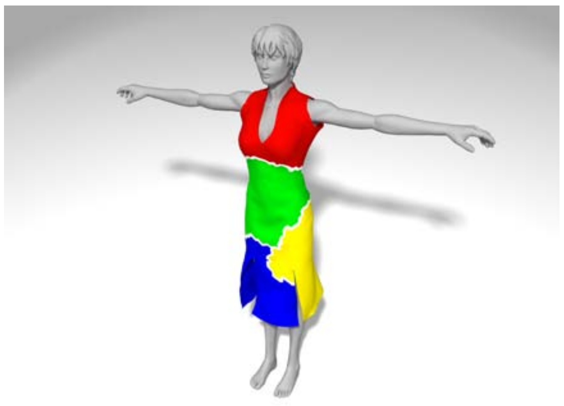
Figure 3: *Test scene 1 consists of a woman wearing a dress which is comprised of roughly 4500 vertices. The avatar consists of more than 27000 vertices.*

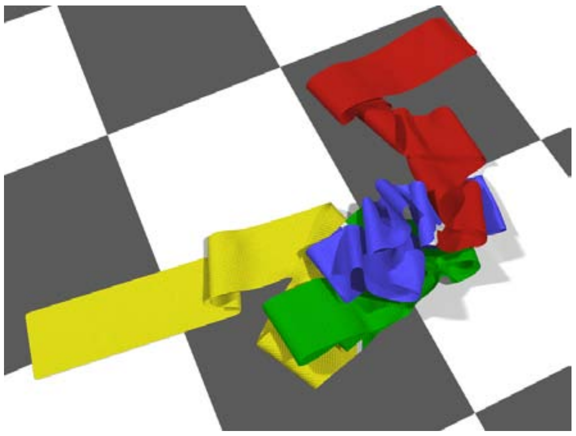
Figure 4: *In the second test scene, the deformable object is a long vertically oriented ribbon (0.05m x 2.00m) comprised of 4141 vertices.*

In this more dynamic example, the deformable object is a long vertically oriented ribbon, comprised of 4141 vertices (see Fig. 4). It first falls onto two differently inclined planes, from which it rebounds towards the floor where it finally comes to rest. In the course of the simulation, external collisions as well as complicated self-collisions occur. The collisions are, however, not as evenly distributed as in the first example and change dynamically over time. Hence, the temporal and spatial coherence is considerably lower than in the first scene.

6.2. Measurements

This section presents runtime measurements for the test scenes described above. In both cases, separate timings are given for the three important phases, i.e. application of

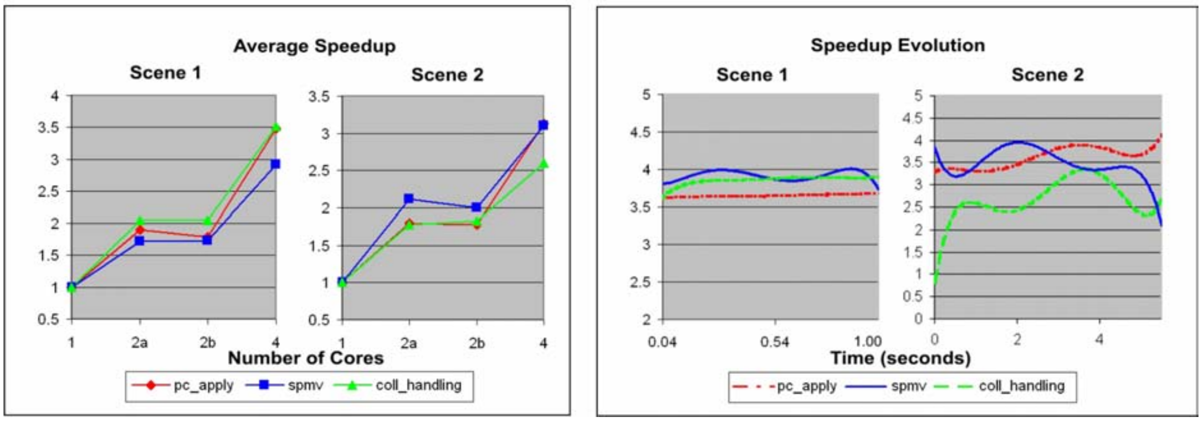

Figure 5: *Integral speedups obtained for the test scenes. The two leftmost tables show average speedups of the different stages for both of the scenes. The two rightmost diagrams show the evolution of speedup over time.*

the preconditioner (*pc_apply*), sparse matrix vector product (*spmv*) and the collision handling stage (*coll_handling*). The leftmost table in Fig. 5 shows the results obtained for the first scene, indicating a high parallel efficiency for each of the stages. As can be seen in the third table from the left, the speedup stays nearly constant over time for this rather static scene.

The second table from the left in Fig. 5 shows the results for scene 2. Excellent performance for the numerics is obtained while the speedup for the collision handling stage is lower as for the first scene. One reason for this behaviour is that collisions occur only marginally in the beginning of the simulation (see Fig. 5, rightmost). Hence, there is not enough work to yield good parallel efficiency. Furthermore, it has to be noted that this test scenario is very challenging as it exhibits only low temporal coherence and collisions are distributed very unevenly. We chose this case to evaluate the robustness of our method and we consider the results satisfying, although there is still room for improvements. It is interesting to see the evolution of the speedup for this scene over time (see 5, rightmost). The computational intensity for the numerics slightly varies but more noticeable is the performance for the collision handling stage. The curve reflects the temporal progression of the scene: it first shows a steep slope as more and more collisions occur. The curve attains its peak after the third second when multiple fabric layers come to lie on each other and finally slightly decreases as the ribbon untangles.

The last aspect to notice is the performance of the different task creation strategies. Fig. 6 shows their influence on the performance of the parallel collision handling algorithm for the two scenes. It can be seen that the naive stationary approach does not scale well when compared to the randomized version, which already shows quite a good performance. The work estimate approach performs very good for the first

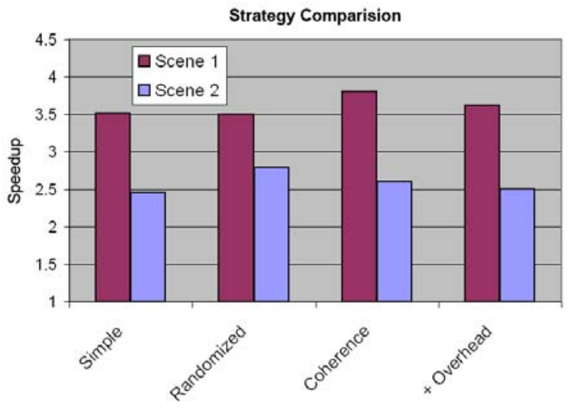

Figure 6: *Comparison of different strategies used for task generation. Our coherence-based method keeps track with the randomized variant for scene 2 and clearly outperforms it in scene 1. Results without the overhead for updating coherency data are also shown.*

scene and attains a similar speedup as the randomized version for the second scene. The diagram also shows speedups for the work estimate approach with and without the overhead for updating the coherence data. At the moment, this is done sequentially as a post-processing step after each collision handling pass. It could, however, be integrated directly into the collision detection and in this case, the overhead could almost completely be avoided.

7. Conclusions and Future Work

In this work we presented key techniques for exploiting parallelism in physically-based simulations on multi-core architectures. We focused on the two major bottlenecks of the simulation, namely the solution of the linear system and the collision handling stage, and proposed efficient parallel algorithms to accelerate these problems. Our initial performance measurements confirm the parallel efficiency of these methods and indicate that physically-based simulations on modern commodity platforms can be greatly accelerated if parallelism is exploited. Because the scalability is encouraging, we would like to further explore the presented methods using more processors. It will be particularly interesting to see how well bandwidth limited components (e.g., SpMV) scale on quad- and eventually *n*-core-based systems.

References

[BFA02] BRIDSON R., FEDKIW R. P., ANDERSON J.: Robust Treatment of Collisions, Contact, and Friction for Cloth Animation. In *Computer Graphics (Proc. SIGGRAPH)* (2002), pp. 594–603.

[BKLW99] BLOCHINGER W., KÜCHLIN W., LUDWIG C., WEBER A.: An object-oriented platform for distributed high-performance Symbolic Computation. *Mathematics and Computers in Simulation 49* (1999), 161–178.

[BW98] BARAFF D., WITKIN A.: Large Steps in Cloth Simulation. In *Computer Graphics (Proc. SIGGRAPH)* (1998), pp. 43–54.

[Cia92] CIARLET P. G.: *Mathematical Elasticity. Vol. I.* North-Holland Publishing Co., Amsterdam, 1992.

[DHv93] DEMMEL J., HEATH M., VAN DER VORST H.: Parallel numerical linear algebra. In *Acta Numerica 1993*. Cambridge University Press, Cambridge, UK, 1993, pp. 111–198.

[EKS03] ETZMUSS O., KECKEISEN M., STRASSER W.: A Fast Finite Element Solution for Cloth Modelling. *Proc. Pacific Graphics* (2003).

[GRR*05] GUTIERRÉZ E., ROMERO S., ROMERO L. F., PLATA O., ZAPATA E. L.: Parallel techniques in irregular codes: cloth simulation as case of study. *Journal of Parallel and Distributed Computing 65*, 4 (April 2005), 424–436.

[HE01] HAUTH M., ETZMUSS O.: A High Performance Solver for the Animation of Deformable Objects using Advanced Numerical Methods. In *Computer Graphics Forum* (2001), pp. 319–328.

[KB04] KECKEISEN M., BLOCHINGER W.: Parallel implicit integration for cloth animations on distributed memory architectures. In *Proc. of Eurographics Symposium on Parallel Graphics and Visualization 2004* (Grenoble, France, June 2004).

[KHM*98] KLOSOWSKI J. T., HELD M., MITCHELL J. S. B., SOWIZRAL H., ZIKAN K.: Efficient collision detection using bounding volume hierarchies of *k*-DOPs. *IEEE Transactions on Visualization and Computer Graphics 4*, 1 (1998), 21–36.

[KK96] KARYPIS G., KUMAR V.: *Parallel Multilevel k-way Partitioning Schemes for Irregular Graphs*. Tech. Rep. 036, Minneapolis, MN 55454, May 1996.

[LGPT01] LARIO R., GARCIA C., PRIETO M., TIRADO F.: Rapid Parallelization of a Multilevel Cloth Simulator Using OpenMP. In *Third European Workshop on OpenMP* (2001).

[LVDY04] LEE B. C., VUDUC R. W., DEMMEL J. W., YELICK K. A.: Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In *ICPP '04: Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 169–176.

[MKE03] MEZGER J., KIMMERLE S., ETZMUSS O.: Hierarchical Techniques in Collision Detection for Cloth Animation. *Journal of WSCG 11*, 2 (2003), 322–329.

[OBHL02] OLIKER L., BISWAS R., HUSBANDS P., LI X.: Effects of ordering strategies and programming paradigms on sparse matrix computations. *Siam Review 44:3* (2002).

[O'H97] O'HALLARON D.: Spark98: Sparse matrix kernels for shared memory and message passing systems, 1997.

[RRZ00] ROMERO S., ROMERO L. F., ZAPATA E. L.: Fast Cloth Simulation with Parallel Computers. In *Euro-Par* (2000), pp. 491—499.

[Saa03] SAAD Y.: *Iterative Methods for Sparse Linear Systems*, 2nd ed. SIAM, 2003.

[She94] SHEWCHUCK J. R.: An Introduction to the Conjugate Gradient Method Without the Agonizing Pain, 1994. http://www.cs.cmu.edu/˜quake-papers/painless-conjugate-gradient.ps.

[TB06] THOMASZEWSKI B., BLOCHINGER W.: Parallel simulation of cloth on distributed memory architectures. In *Proc. of Eurographics Symposium on Parallel Graphics and Visualization 2006* (Braga, Portugal, May 2006).

[THM*05] TESCHNER M., HEIDELBERGER B., MANOCHA D., GOVINDARAJU N., ZACHMANN G., KIMMERLE S., MEZGER J., FUHRMANN A.: Collision Handling in Dynamic Simulation Environments. In *Eurographics Tutorials* (2005), pp. 79–185.

[ZFV02] ZARA F., FAURE F., VINCENT J.-M.: Physical Cloth Animation on a PC Cluster. In *Fourth Eurographics Workshop on Parallel Graphics and Visualisation* (2002).

[ZFV04] ZARA F., FAURE F., VINCENT J.-M.: Parallel simulation of large dynamic system on a pcs cluster: Application to cloth simulation. *International Journal of Computers and Applications* (march 2004). special issue on cluster/grid computing.