Eurographics Symposium on Parallel Graphics and Visualization (2009)
J. Comba, K. Debattista, and D. Weiskopf (Editors)

Interactive Physical Simulation on Multicore Architectures

Everton Hermann[1], Bruno Raffin[1] and François Faure[2]

[1]Inria Grenoble Rhône-Alpes, France
[2] Grenoble Universities, France

Abstract

In this paper we propose a parallelization of interactive physical simulations. Our approach relies on a task parallelism where the code is instrumented to mark tasks and shared data between tasks, as well as parallel loops even if they have dynamic conditions. Prior to running a simulation step, we extract a task dependency graph that is partitioned to define the task distribution between processors. To limit the overhead of graph partitioning and favor memory locality, we intend to limit the partitioning changes from one iteration to the other. This approach has a low impact on physics algorithms as parallelism is mainly extracted from the coordination code. It makes it non parallel programmer friendly. Results show we can obtain good performance gains.

Categories and Subject Descriptors (according to ACM CCS): Software [D.1.3]: Parallel programming—, Computer Graphics [I.3.7]: Animation—

1. Introduction

The goal of interactive physics is to simulate the dynamics of virtual objects submitted to mechanics laws at an interactive refresh rate. It is a challenge for many applications like video-games, virtual training for manufacturing processes or surgery. The simulation of complex heterogeneous scenes requires combining many models and methods, leading to computationally intensive applications. For instance, virtual surgery requires simulating a human body with articulated rigid objects (bones), deformable objects (flesh) and fluids (blood) in contact. To conform to interactive-time constraints, to simulate more complex scenes, and to take advantage of the new processor architectures, physical simulation software libraries will increasingly have to rely on parallelism.

Parallelism can be applied to non-colliding objects, which can be simulated independently [YFR06, Mir00]. However, when all the objects make a single connected group, as in surgery scenes, no parallelization is possible at this level, and a finer grain is necessary. Recently, important speedups have been obtained using the Graphics Processing Unit (GPU) as co-processor [bul]. However, this requires to explicitly parallelize the methods at very fine level using specialized libraries, which is difficult for non-experts.

The development of advanced physical simulations, such as surgery simulations, requires the collaboration of specialists in various fields such as mechanics of continuous media, collision detection, numerical methods, geometry, haptics. In practice, most of them are only able to program sequentially. These specialists also need to easily experiment various models and algorithms, combined with each other's contributions. In this context, parallelism should have the following properties :

- Non-invasive : The constructs used to express parallelism, i.e. to define task boundaries and data dependencies, should have a reduced impact on the code. It makes it friendly to algorithm developers and application developers who are usually not experts in parallel computing.
- Compatible : The parallelization should be generic enough not to impair further lower level parallelizations. For instance it should be possible to rewrite one specific algorithm to defer part of the work load on a GPU while keeping the benefit of the current parallelization. It makes it complementary with the popular GPU-based parallelizations.
- Externalized : The scheduling and mapping of tasks on processors should not be embedded in the physical simulation code. It eases code development, the programmer not being concerned about these aspects. It also en-

ables to evaluate different scheduling strategies without code modifications.

Relying on the KAAPI middleware [GBP07], we instrument the code of a generic simulation library to identify tasks (corresponding to methods calls) as well as the data shared between tasks. We obtain a mid-to-coarse grain parallelism that affects the coordination code but not the internal algorithm codes. Developers can also use these keywords to express parallelism inside their algorithms.

Prior to running a simulation step, we extract a graph of tasks representing the data dependencies between the tasks. Those tasks are then grouped in partitions that are mapped on the processors. During a simulation step, processors execute tasks in parallel, suspending their execution if required to respect data dependencies. We thus avoid the spurious synchronizations otherwise induced by parallelizations that do not rely on an explicit identification of shared data. This is the first specific contribution of this paper. Additionally, some high-level algorithms such as iterative equation solvers include loops with dynamics conditions, *i.e.* loops coordinating the execution of several tasks with a break condition that may depend on the computations of several tasks. We introduce a construct to enable to extract parallelism from such loops, which is the second specific contribution of this paper.

Task mapping and scheduling is not embedded into the code. The KAAPI middleware takes care of this aspect. It supports different scheduling algorithms that can be modified if required. Base scheduling algorithms include a static task partitioning computed from the data dependency graph, and a dynamic work-stealing [FLR98].

We propose a modified static scheduling algorithm that limits the graph partitioning overhead and favors memory locality. The partitioning of the previous simulation step is reused unless new collisions occurred. In this case, new interaction forces modify the data dependencies. Graph partitioning is them recomputed accordingly, while avoiding moving previously existing tasks to a different processor.

The rest of this paper is organized as follows. Section 2 provides the necessary background on physical simulation, and briefly summarizes previous work on parallel implementations. Section 3 details our parallelization approach. Results are presented and discussed in Section 4 and a conclusion is finally drawn in Section 5.

2. Background

2.1. Physical Simulation

The physical simulation pipeline is an iterative process where a sequence of steps is executed to advance the scene forward in time (Figure 1). The time-dependent variables are basically the position and velocity of each Degree of Freedom (DOF), stored in state vectors. The pipeline includes a collision detection step, used to dynamically create or delete interactions between objects, based on geometry intersection. Time integration consists in computing a new state (i.e. position and velocity vectors), starting from the current state and integrating the forces in time. Finally, the new scene state is rendered and displayed or sent to other devices. Usually a synchronization occurs before and after the time integration step to obtain a consistent scene state.

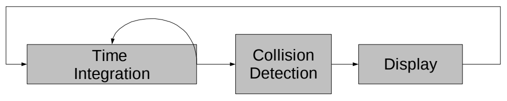


Figure 1: *Simulation Pipeline*

In this paper, we focus on time integration. The DOF are updated by solving ordinary differential equations. Most traditional parallel physical simulation methods focus on optimally partitioning complex objects, such as an atmospheric model for weather forecast [STF*02], or interacting media in multiphysics applications [MCW*02]. Interactive mechanical simulators can involve objects of different kinds, interacting with each other using forces, as in [AR06]. The objects are simulated independently, using their own encapsulated simulation methods. Interaction forces are periodically updated based on the current states of the objects. This approach provides a high flexibility since arbitrary objects can be combined. But, it is limited to explicit time integration, where each object evaluates its net force at the current time to straightforwardly derive its next position and velocity. The objects can not anticipate the variations of the interaction forces, since these forces depend on several objects. This is sometimes called *weak coupling*. Consequently, divergence occurs unless sufficiently small time steps are applied, which can result in very slow simulations when stiff interaction forces are applied. Unfortunately, high stiffness is generally required for contact forces to avoid visible object intersections.

This well-known stability problem can be avoided using *strong coupling* such as implicit time integration [BW98], where force variations are anticipated, allowing large time steps and high performance. However, this requires setting up and solving an equation system involving the objects and their interaction forces, which is not possible when the objects are simulated independently.

Our approach combines flexibility and performance, using a new efficient approach for the parallelization of strong coupling between independently implemented objects. We extend the SOFA framework [ACF*07] we briefly summarize here. The simulated scene is split into independent sets of interacting objects. Each set is composed of objects along with their interaction forces, and monitored by an implicit differential equation solver. The object are made of components, each of them implementing specific operations related to forces, masses, constraints, geometries and other parameters of the simulation.

A collision detection pipeline creates and removes contacts based on geometry intersections. It updates the independent object sets accordingly, so that each one can be processed independently of the others. Within each set, the equation solver processes an arbitrary number of abstract objects and interaction forces, by traversing a data structure using visitors. Visitors apply specific tasks at each component, such as force accumulation or state vector operations.

We straightforwardly associate the elementary tasks with the virtual methods overloaded by the components. The experts in various disciplines who collaborate on the development of the library can thus safely ignore parallelism issues if they respect the framework interface and only access components data through this interface. This does not prevent a fine grain parallelisation at a task level, for deferring computations to a co-processor for instance.

Algorithm 1 Simulation Pipeline with Explicit Time Integration

```
1: loop
2:     Compute Force
3:     Compute Acceleration
4:     v+=a*dt
5:     x+=v*dt
6:     Collision Detection
7:     Display
8: end loop
```

In our first parallelization attempt, each branch of the data structure was processed in parallel by the visitors the high-level algorithms fire. Unfortunately, due to the sequential code of the high-level algorithms (see, *e.g.*, Algorithm 1), synchronizations happened at the end of each visit to make sure that the data used by the next visitor is available, as illustrated in the left of Figure 2, which dramatically decreases the speedup. We relied on a data dependency graph analysis to circumvent this issue as presented in Section 3.1. Another difficulty is due to the loops with dynamic exit conditions, which take place in iterative equation solvers. This control structure was not available in our parallel programming environment, preventing access to this parallelism. We therefore introduced a new parallel loop construct as presented in Section 3.3. Finally, dynamic scene changes induced by collisions require dynamic scheduling. However, when no new collision appears and no contact disappears, the data dependency remains the same and we wish to reuse the previous scheduling. This point is discussed in Section 3.2.

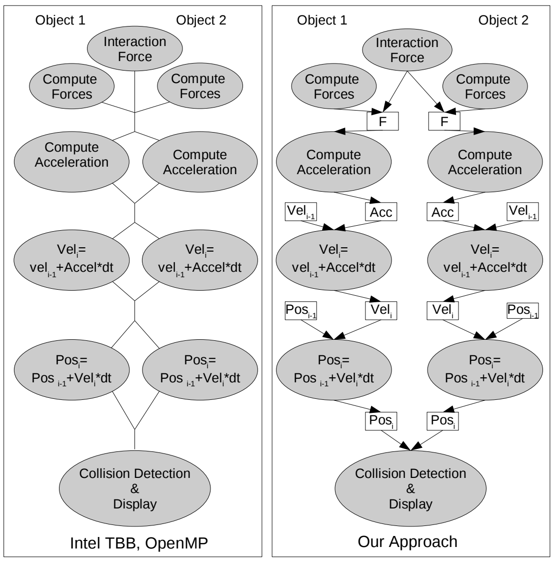


Figure 2: *Task graph of the time integration step presented in algorithm 1. Left : a recursive or data parallel approach adds a synchronization after each instruction of the algorithm. Right : we avoid the unnecessary synchronizations performing a data dependency analysis.*

2.2. Parallel Programing Environments

Parallel environments like Cilk [Ran98], Intel TBB [Rei07] and OpenMP [DM98] have some similarities to the KAAPI execution environment we use. All of them support a loop over independent data and have some mechanisms to specify the data that are shared between the different threads. The main difference is in the way the tasks are executed. In our approach we can exploit the implicit independence between tasks and avoid synchronization barriers, like shown on Figure 2. The control of the outermost loop is not centralized avoiding to have the main thread as a bottleneck. To obtain a similar result using the aforementioned programming environment, we would need to reorganize the code to explicitly express that there is no data dependency between the tasks associated with different objects.

A data dependency approach is also employed by [ZLM07] as a way to extract parallelism from a single threaded application. Data dependencies are computed at compiling time. Specific processor instructions, supported by the multicore architecture they propose, enable to control the execution flow at runtime.

2.3. Parallel Physical Simulation

The basic approach usually used for games is to construct a dependency graph from the different subsystems present in the main loop. For instance it can lead to the execution of the physics engine concurrently with the artificial intelligence [RCE05]. This kind of approach reflects the structure of an application, and is independent from the scenarios. But the amount of parallelism extracted is very limited and does not scale as there is a restricted number of independent tasks.

[YFR06] proposes a parallel version of the Open Dynamics Engine (ODE). The main idea is to group colliding objects in islands and distribute the islands over the available threads. This approach works well when there are enough independent islands and especially with rigid objects that are simple to solve. If we consider colliding soft bodies, as it is usually the case for medical simulations, this approach would result in a single thread and would not bring any performance gain.

For soft body simulations, most of the researches are focused on a fine grain data parallelism [JTS*07, TPB08]. Some physical simulators like PhysX [phy] and Bullet [bul] defer computations to a GPU or SPU co-processor. Such fine grain parallelization can be very efficient but it requires rewriting each algorithm.

This paper deals with the parallelization of the time integration step. But collision detection can account for a significant part of the overall iteration time. Parallel algorithms have been developed by reworking sequential ones like the recursive coordinate bisection algorithm [BAPH00] or regular voxel-based approaches [LK02].

3. Parallel Simulation using Task Graphs

3.1. Graph Generation

The simulation pipeline is organized around a main external loop that makes the simulation advance forward in time (Figure 1). In this paper we focus on the parallelization of the time integration step. This step may implement its own internal iterative process, especially when relying on *strong coupling* such as implicit time integration. We consider a parallelization based on the identification of tasks and data dependencies from the sequence of steps involved in one iteration of time integration. The data are vectors representing the physical states of an object, like positions, velocities and forces. The tasks are operations on those vectors like accumulating forces, computing positions from velocities, etc. The data dependencies between tasks and data represent graph edges. Figure 2 corresponds to a simplified graph of one explicit time integration step where two objects are colliding. In response to the collision an interaction force is set between both objects.

We produce data flow graphs corresponding to the different operations triggered by the equation solvers, rather than directly performing these operations. Each visitor fired by the algorithm produces tasks, and the complete sequence of operations performed by the algorithm is represented as an assembled graph that precisely models data dependencies at component granularity. With a good partitioning of this graph we can avoid the undesirable synchronizations previously discussed, and significantly improves performance.

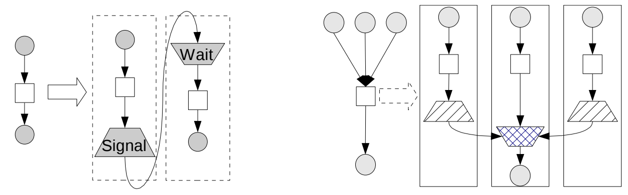
3.2. Graph Partitioning

Once we have a task graph representing the simulation operations, we can schedule this graph by assigning the tasks to a set of processors using a partitioning algorithm. In most of the cases we can deduce the task affinity from the scene structure. In Figure 2 all the tasks following the force computation modify the data of only one object. In this case the tasks can automatically be mapped in the same partition gathering all the tasks modifying a given object. The remaining tasks that are not directly associated to one simulation object, like the *interaction force*, are grouped with other tasks that access the same group of objects.

If needed, we can also employ a dedicated partitioner like SCOTCH [PR96] or METIS [KK98], to better optimize the partitioning. One situation that may need a deeper analysis of the dependency graph is for extracting parallelism from the tasks that access the same object. But using such a partitioning we lose the control on the number of partitions, as it depends on the number of objects in a scene and the distribution of the objects in space at a given time step.

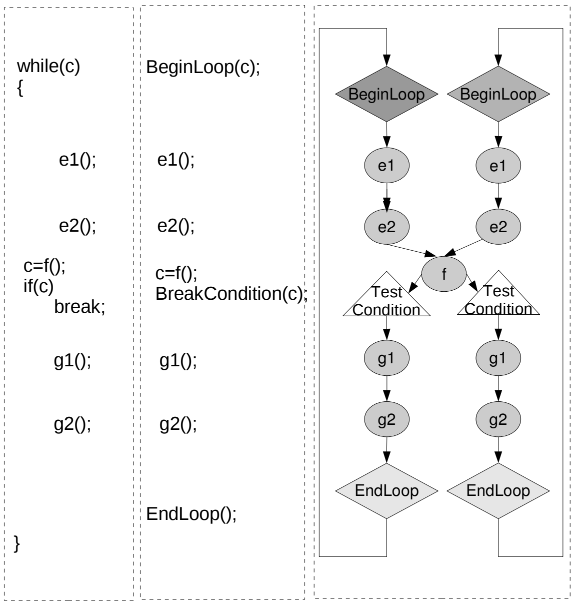
In opposite, gathering in the same partition the tasks that access the same data (Owner Compute Rule) leads to a partitioning that corresponds to the scene structure. It enables to rebalance the object distribution over the processors without requiring to repartition the graph. We can move a computationally expensive object from an overloaded processor to an idle one only by reassigning the partition related to this object. Also we can associate the data to the processors, and the tasks that access a given set of data will be placed in the right processor even if the graph is repartitioned, which improves the locality of the accessed data.

After partitioning the tasks, we create the threads that will be executed in parallel. When a task needs to access some data produced by another partition, we add control structures to signal when the data is ready. In a Writer/Reader profile like shown in Figure 3(a) a *signal* task is inserted after the writer task. This task signals all readers when the data is ready. A *wait* task is placed just before the first reader of each partition, to passively wait for the signal. To support a parallel cumulative write, that occurs for instance when computing forces (Figure 2), we decompose it into two phases : accumulation and reduction. At the beginning each writer accumulates its value to a temporary buffer, so they can run without concurrency issues (Figure 3(b)). The reducer waits for all writers and sums up the accumulated temporary buffers. The *Reducer* is considered as the real writer of the data, as the writers only accumulate to temporary buffers. The result produced by the reducer will be accessed by all subsequent readers.



(a) Write Read Operation (b) Cumulative Write

Figure 3: *Control tasks employed to guarantee data access coherence between different partitions. On a write/read operation the reader waits for the data to be produced by the writer. In a cumulative write operation all the writes are performed in a temporary buffer that will be further accumulated by the Reducer.*



```
while(c)
{
        e1();
        e2();
  c=f();
  if(c)
        break;
        g1();
        g2();
}
```

```
BeginLoop(c);
  e1();
  e2();
  c=f();
  BreakCondition(c);
  g1();
  g2();
EndLoop();
```

Figure 4: *Dynamic loop. Functions e1, e2, g1, g2 change multiple objects. Left : standard code. Middle : our modified code. Right : Graph representation of our implementation.*

Because most of the steps are repeated between iterations, the graph can be replayed, to save graph partitioning overheads and to favor data affinity by limiting data movements between iterations. However each time the graph changes we need to recompute the data dependency to be sure that the parallel run will have the same result as the sequential algorithm.

3.3. Dynamic Loops

As stated on Section 2.1, some integration processes iterate up to comply with a given convergence criterion. To be able to extract parallelism from such loop, we need to mark such loops and conditional breaks so they can be identified in the task graph.

We introduce two special tasks, *EndLoop* and *BeginLoop*, to delimit the loop boundaries. All tasks created between the *BeginLoop* and the *EndLoop* are considered part of the loop body and are redeployed at each loop iteration. This loop can be controlled through a condition variable or an iteration counter. The main difference is that a condition variable is shared by all the loops, while an iteration counter can be incremented locally by each loop without the need of synchronization. We also introduce a *ConditionalBreak* task as a break instruction that can be very convenient to exit the loop.

The condition is tested each time the *BeginLoop*, *EndLoop* or *ConditionalBreak* tasks are executed. When reaching a *EndLoop* task , all the tasks of the loop body are redeployed if the loop condition is still valid. Otherwise, we must exit the loop. The next task to be executed will be the task just after the *EndLoop*.

To partition a loop we take the same assignment pattern that for other tasks as explained on Section 3.2. If a loop is decomposed onto several partitions, the loop control tasks are replicated on all the partitions, including the *ConditionalBreak* tasks. If the break condition is a boolean, it is treated as a data that is shared between all the loop control structures. Otherwise, the condition can be evaluated locally for each partition. In this case, each loop executed by each partition can run freely without the need for extra synchronization.

In Figure 4 we show a loop executed onto two partitions. The task *F* updates the condition variable at each iteration, and all the loops that depend on this variable are readers of this shared data. By considering a condition variable like any other shared data, we guarantee that the access control is made automatically by the *Writer/Readers* mechanism explained in previous section.

4. Results

We tested our approach with different simulation scenarios, going from identical objects that are completely independent to heterogeneous scenes of colliding objects. The tests were performed on one PC equiped with 16 cores (8 dual-core 2.2GHz Opteron processors) with 32GB of RAM. Each processor has direct access to 4 GB of memory and uses Hypertransport to access the other processor memory, leading to non-uniform memory accesses.

As the parallelization of the collision detection step is out of the scope of this paper, we only consider the performance

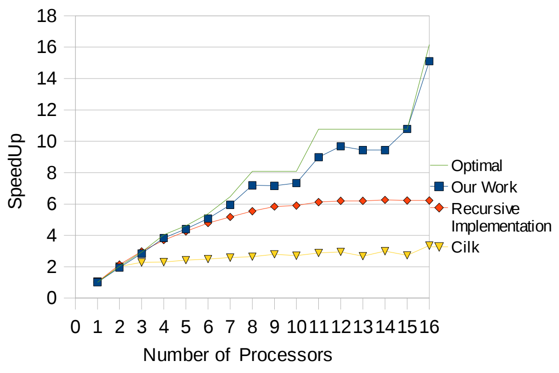


Figure 5: *Speedup using 64 identical bars of size 16x4x4 without collision.* $T_1 = 150ms$

of the time integration step. The speedup is computed taking the sequential execution time T_1 as the reference. Task graph partitioning (that may not occur at each iteration) is always included in the measured time.

The video associated with this paper shows executions that include a sequential collision detection and a scene rendering.

4.1. Independent Similar Objects

First test is a scene composed of non colliding identical objects simulated independently. Each object is a soft bar that is attached at one end and get deformed due to the gravity force. This test highlights the overhead induced by the parallelization.

The test runs with 64 bars, each one is composed by 256 particles that are simulated using hexahedral finite elements (Figure 5). This size of object is large enough not to fit in cache, and at the same time small enough to avoid having the memory as a bottleneck.

We compared our approach with a Cilk based parallelization relying on work-stealing, and an implementation using recursive parallelism following the Cilk divide and conqueror approach. In both cases there is no parallel dynamic loop. All the iterative algorithms were expressed using parallel static loops, forcing all the tasks to resynchronize before starting a new operation. Neither of them ensures the affinity between tasks and processor. From one step to the other, there is no guarantee that a task will execute on the same processor. For this simple case, work-stealing was expected to be slower than our approach as the limited amount of available parallelism was not able to compensate for the overhead of the dynamic load-balancing.

Our implementation is close to the optimal. We succeeded to manage the most important issues that are not treated

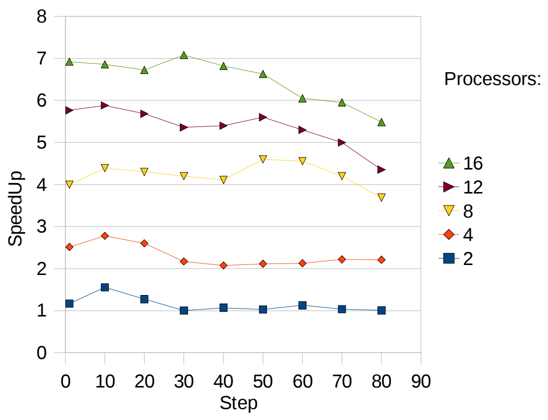


Figure 6: *Speedup on a heterogeneous scene containing different objects simulated using different mechanical models.*

by previous work : unnecessary synchronization barriers and locality coherence over iteration steps.

4.2. Complex Scene

The parallelization of a scene composed of independent objects can be obtained using simple approaches, like simulating each object in a different thread or even in a different process. However when objects are colliding, such method can only take advantage of a reduced amount of parallelism. This is for this kind of simulation that our approach shows its potential.

In Figure 8 we have the initial and final state of a scene of heterogeneous objects falling under gravity and colliding (see the video for a full simulation). The different objects have surface meshes that go from 400 to 2.500 triangles. The method employed to simulate the object varies : there are rigid bodies and soft bodies using mass springs, finite elements or deformable grids models.

During the first steps there is almost no collision between objects, and as explained on Section 2.1, they can be solved by separate instances of the solvers. Also, we obtain a high speedup as there is few tasks that access data from different objects. However, as we approach the final state, objects get closer, and at the end they are all part of one single collision group. It means that they all must share the same iterative loop and break conditions, as the instance of the solver employed on all the objects is the same to avoid instabilities (*strong coupling*). Because of the collisions, many tasks access multiple objects, creating synchronization points.

When collision groups change, the task graph is updated. The cost of this procedure is proportional to the number of contacts in the scene. In the scene from Figure 8 the mean cost of updating the graph is close to 30ms, which is about

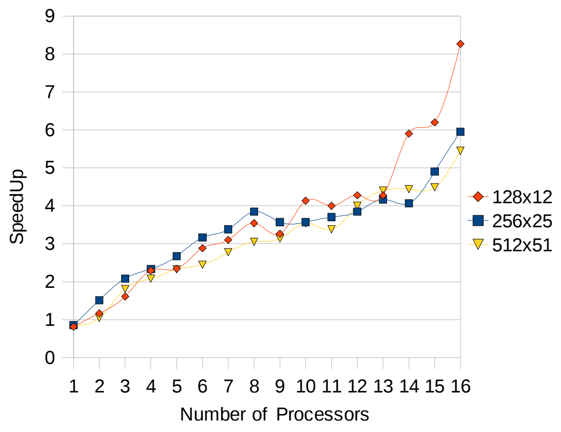


Figure 7: *Speedup using a mass-spring mesh simulation by domain decomposition.The sequential time for each case is : 125ms (128x128), 500ms (258x258), 2.2s (512x512)*

the same time required to execute a timestep. This overhead is usually not noticeable (see the video). It is compensated by the overall performance improvement and is often amortized over a few timesteps.

The speedup obtained here did not require any effort from the application developer nor the physics algorithm developer. The physics algorithms can evolve independently from the parallel code, reducing the lag between the development of sequential algorithm and the execution of this algorithm in a parallel architecture.

4.3. Domain Partition

Our approach targets a coarse grain parallelism without looking to the internal implementation of a given method. In a scene with few objects or with a huge time consuming object, it would be harder to obtain good performance gains. As SOFA enables to have tightly connected objects into a scene, we can explicitly decompose a large object into smaller connected ones. SOFA guarantees that both physical simulation will be identical. Constraints ensure that two points from different objects will be considered as a single one during the simulation. To evaluate this kind of domain decomposition we used a mass-spring mesh that is cut into smaller objects like shown in Figure 9(a). With this simple decomposition, the speedup can be significant (Figure 7). We obtainted better results with smaller objects due to the overhead introduced by memory accesses on larger scenes. Notice that like for the previous test, all the objects share the same iterative loop for the convergence phase of the conjugate gradient.

4.4. Medical Simulation

The last test focuses on a more realistic scene simulating a torso model with all organs. The bones were simulated as rigid bodies, lungs and liver are simulated finite elements, and the intestine uses springs. We ran this test on a dual Quad Intel Xeon architecture, with a total of 8 cores. The sequential integration time is 50ms, and the parallel time using 8 cores is 14ms, resulting on a speedup of about 3.5. Note that this speedup is automatically available to any user, including non experts in parallelism.

5. Conclusion and Future Work

In this paper we presented a framework for coarse-to-mid grain parallelism that takes advantage of the parallelism between different objects in a scene. We extract tasks from the sequential algorithm to generate a dataflow graph. This graph is then partitioned to be executed in parallel. The changes on the physical simulator are restricted to the system core, and all the physics routines are kept unchanged.

To be able to parallelize more complex simulations, we introduced new control structures to represent loops in a graph. Those loops can be partitioned and executed on multiple processors, giving access to extra coarse parallelism in the scene. Unlike traditional parallel loops, we can create loops whose number of iteration is dynamic. Additionally, conditional breaks are supported inside the loop body.

This approach was tested on different scenarios going from similar independent objects to complex scenes. Tests shows that we can obtain good performance results, achieving in some cases near optimal speedups. In all the scenes the gain on performance is obtained transparently to the physics developer.

Future work focuses on using a work-stealing load balancing approach as a way to integrate our multicore implementation with co-processor accelerated code. Another work look at optimizing the task graph updating, to avoid reconstructing the whole graph each time it changes.

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)* (Long Beach, California, Etats-Unis, February 2007), pp. 1–6.

[AR06] Allard J., Raffin B. : Distributed physical based simulations for large vr applications. *Virtual Reality Conference, 2006* (March 2006), 89–96.

[BAPH00] Brown K., Attaway S., Plimpton S., Hendrickson B. : Parallel strategies for crash and impact simulations. *Computer Methods in Applied Mechanics and Engineering 184*, 2 (2000), 375–390.

[bul] Bullet Physics Library. http://www.bulletphysics.com.

[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* (New York, NY, USA, 1998), ACM, pp. 43–54.

[DM98] DAGUM L., MENON R. : Openmp : an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE 5*, 1 (Jan-Mar 1998), 46–55.

[FLR98] FRIGO M., LEISERSON C. E., RANDALL K. H. : The implementation of the cilk-5 multithreaded language. *SIGPLAN Not. 33*, 5 (1998), 212–223. http ://supertech.csail.mit.edu/papers/cilk5.pdf.

[GBP07] GAUTIER T., BESSERON X., PIGEON L. : Kaapi : A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *PASCO '07 : Proceedings of the 2007 international workshop on Parallel symbolic computation* (New York, NY, USA, 2007), ACM, pp. 15–23.

[JTS*07] JERABKOVA L., TERBOVEN C., SARHOLZ S., KUHLEN T., BISCHOF C. : Exploiting multicore architectures for physically based simulation of deformable objects in virtual environments. In *Virtuelle und Erweiterte Realität, 4. Workshop der GI-Fachgruppe VR/AR, Weimar, Germany* (2007).

[KK98] KARYPIS G., KUMAR V. : A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput. 20*, 1 (1998), 359–392.

[LK02] LAWLOR O. S., KALÉE L. V. : A voxel-based parallel collision detection algorithm. In *ICS '02 : Proceedings of the 16th international conference on Supercomputing* (New York, NY, USA, 2002), ACM, pp. 285–293.

[MCW*02] MCMANUS K., CROSS M., WALSHAW C., CROFT N., WILLIAMS A. : Parallel performance in multi-physics simulation. In *ICCS '02 : Proceedings of the International Conference on Computational Science-Part II* (London, UK, 2002), Springer-Verlag, pp. 806–815.

[Mir00] MIRTICH B. : Timewarp rigid body simulation. In *Proc. of ACM SIGGRAPH* (2000), pp. 193–200.

[phy] Nvidia PhysX. http://www.nvidia.com/physx.

[PR96] PELLEGRINI F., ROMAN J. : Scotch : A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *HPCN'96* (Bruxelles, 1996), Springer, pp. 493–498.

[Ran98] RANDALL K. : *Cilk : Efficient Multithreaded Computing*. Tech. rep., Cambridge, MA, USA, 1998.

[RCE05] RHALIBI A. E., COSTA S., ENGLAND D. : Game engineering for a multiprocessor architecture. In *DIGRA Conf.* (2005).

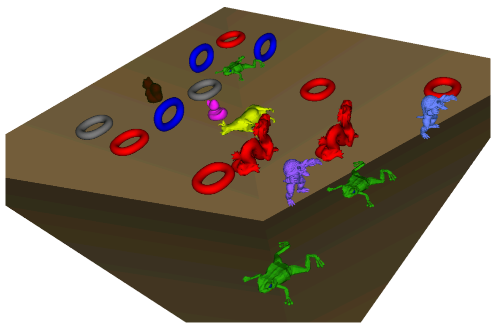
[Rei07] REINDERS J. : *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[STF*02] SHINGU S., TAKAHARA H., FUCHIGAMI H., YAMADA M., TSUDA Y., OHFUCHI W., SASAKI Y., KOBAYASHI K., HAGIWARA T., ICHI HABATA S., YOKOKAWA M., ITOH H., OTSUKA K. : A 26.58 tflops global atmospheric simulation with the spectral transform method on the earth simulator. In *Supercomputing '02 : Proceedings of the 2002 ACM/IEEE conference on Supercomputing* (Los Alamitos, CA, USA, 2002), IEEE Computer Society Press, pp. 1–19.

[TPB08] THOMASZEWSKI B., PABST S., BLOCHINGER W. : Special section : Parallel graphics and visualization : Parallel techniques for physically based simulation on multi-core processor architectures. *Comput. Graph. 32*, 1 (2008), 25–40.

[YFR06] YEH T. Y., FALOUTSOS P., REINMAN G. : Enabling real-time physics simulation in future interactive entertainment. In *Sandbox '06 : Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames* (New York, NY, USA, 2006), ACM, pp. 71–81.

[ZLM07] ZHONG H., LIEBERMAN S., MAHLKE S. : Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *Proc. 2007 International Symposium on High Performance Computer Architecture* (February 2007).


(a) Initial State

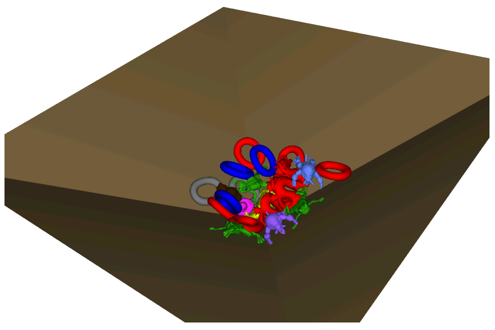
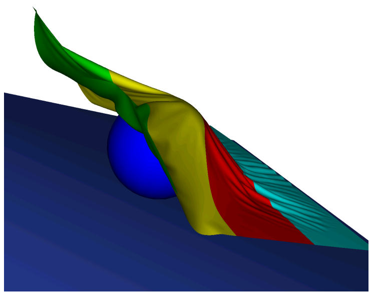

(b) Final State

Figure 8: *Scene used for the complex simulation tests. Objects are simulated using different methods, like Finite Elements, Springs and Regular Grids*


(a) Domain decomposition.

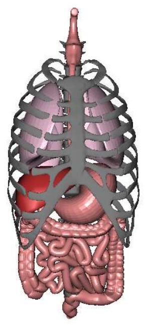

(b) Medical simulation.

Figure 9: *Scenes used on performance tests*