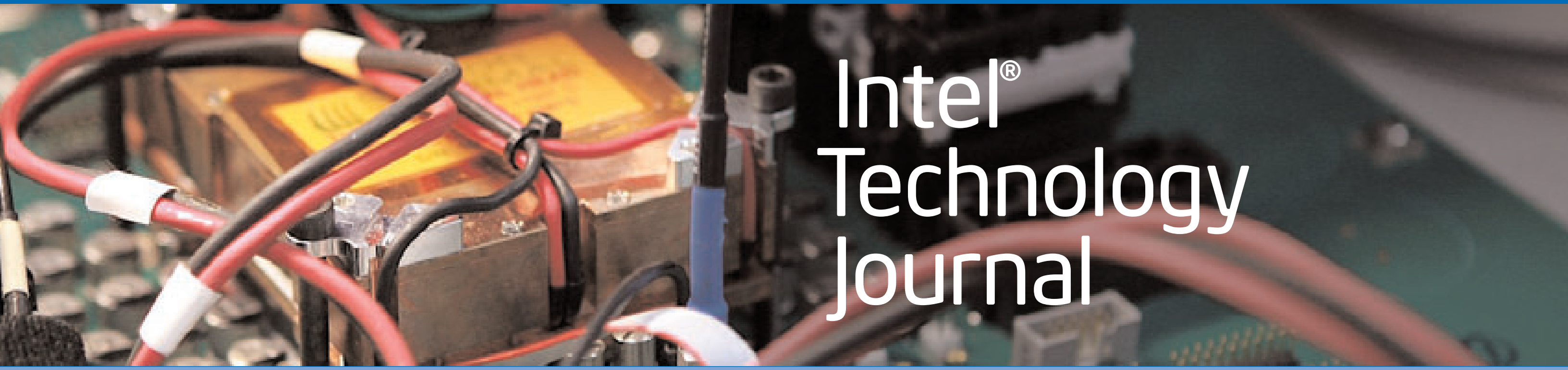
Volume 11 Issue 03 Published, August 22, 2007 ISSN 1535-864X DOI: 10.1535/itj.1103.08



Tera-scale Computing

High-Performance Physical Simulations on Next-Generation Architecture with Many Cores

High-Performance Physical Simulations on Next-Generation Architecture with Many Cores

Yen-Kuang Chen, Corporate Technology Group, Intel Corporation
Jatin Chhugani, Corporate Technology Group, Intel Corporation
Christopher J. Hughes, Corporate Technology Group, Intel Corporation
Daehyun Kim, Corporate Technology Group, Intel Corporation
Sanjeev Kumar, Corporate Technology Group, Intel Corporation
Victor Lee, Corporate Technology Group, Intel Corporation
Albert Lin, Corporate Technology Group, Intel Corporation
Anthony D. Nguyen, Corporate Technology Group, Intel Corporation
Eftychios Sifakis, Corporate Technology Group, Intel Corporation
Mikhail Smelyanskiy, Corporate Technology Group, Intel Corporation

Index words: Physical simulations, chip multiprocessor, many cores, parallel scalability, memory bandwidth

ABSTRACT

Physical simulation applications model and simulate complex natural phenomena. The computational complexity of real-time physical simulations far exceeds the capabilities of modern unicore microprocessors, which are limited to only tens of billions floating-point operations per second (FLOPS). However, the advent of multi-core architectures promises to soon make processors with trillions of FLOPS available. Such processors are also known as tera-scale processors. Physical simulations can exploit this huge increase in computational capability to increase realism, enable interactivity, and enrich a user's visual experience.

In this work, we study physical simulation applications in two broad categories: production physics and game physics. After parallelization, the benchmark applications achieve parallel scalabilities of 30×–60× on a simulated chip-multiprocessor with 64 cores.

We examine the memory requirements of physical simulation applications and find that they require cache sizes in excess of 128MB and main memory bandwidths in excess of hundreds of GB/s for real-time performance. A radical re-design of the memory hierarchy may be necessary for the multi-core tera-scale era to provide good scaling for this type of application.

INTRODUCTION

The booming computer games and visual effects industries continue to drive the graphics community's seemingly insatiable desire for increased realism, believability, and speed. In the past decade, physical simulation has become a key to achieving the realism expected by audiences of games and movies. Physical simulation models the laws of physics to simulate life-like movement and interaction among objects, such as rigid and deformable bodies, human faces, cloth, and water.

Physical simulation can be used in a variety of settings such as weather prediction, movie special effects, and computer games. Complex natural phenomena such as ocean waves crashing on a shore, a flag waving in the wind, or bricks falling from a collapsing tower are modeled by means of numerical simulation of physical laws. Modeling different natural phenomena requires a diverse set of techniques, algorithms, and data structures, making physical simulation both complex and general. Computation and memory requirements are extremely demanding. This makes the workloads a challenging target for current as well as future architectures.

In this paper, we examine applications involving physical simulation for production environments and for gaming. For production physical simulation, we study the PhysBAM package from Stanford University [5, 11], which is used by several special-effects and film production companies, including Pixar and Industrial

Light and Magic. The goal is to recreate the visual experience of a human observing a natural phenomenon. For gaming physical simulation, we study the open source ODE package [13]. This package provides similar functionality to the widely used commercial Havok Effect package from Havok. The goal of physical simulation in gaming is to make real-time interactions between objects as accurate as possible. The difference in goals for the two physical simulation domains leads to different choices for algorithms and data structures. However, these two domains do have many similar characteristics.

One common characteristic of production and gaming physical simulation is a need for significant acceleration. On a 4-way Intel® Xeon® processor 3.0GHz system, with 16GB of DDR2-3200 and three levels of cache on each processor (16KB L1, 1MB L2, and 8MB L3), the production physics workloads take 5 to 188 seconds to process a single frame. These workloads have hundreds of thousands to a few million entities (tetrahedra/grid cells) interacting with each other. In contrast, for game physics workloads, only a thousand objects can currently interact in real time. Acceleration by an order of magnitude or more will allow improved accuracy, modeling of new effects, and even interactive or real-time production applications. Multi-core processors are now common, and we expect the number of cores to increase steadily for the foreseeable future, so that multi-core processors capable of executing applications tens of times faster than today's processors are on the horizon. Such processors would improve the speed and realism of production-quality or real-time game physical simulation applications. However, for an application to harness the computational power of such a multi-core processor, it must effectively utilize multiple threads. Parallelization of a large code base as used by production or game physics applications is not trivial, especially when the target parallel scalability is tens of threads.

Another similarity in requirements for the two categories of physical simulation applications is high-bandwidth requirements. The size of the data scales with increasing resolution or number of objects in the simulation. Input sizes are often millions of volume elements or tens of thousands of objects. This leads to memory footprints that are tens of megabytes (i.e., larger than typical caches). These applications therefore require either much larger caches or a large main memory bandwidth.

Our contributions are as follows:

- We have parallelized six state-of-the-art physical simulation applications (fluid dynamics [4], human face simulation [12], and cloth simulation [2] for production physics and convex body collision [1, 3], game cloth [7], and game fluids [9] for game physics). In parallelizing these workloads, we employed various techniques which include parallelizing loops/graph operations and using alternative algorithms for better scalability.
- We simulated and analyzed the scalability of these applications using cycle-accurate simulation of a chip-multiprocessor with 64 cores. The workloads studied achieved a parallel scaling of 30× to 60× for 64 cores.
- We perform a detailed analysis of the memory requirements of these applications. Our study finds that future physical simulation workloads demand cache sizes close to 100 megabytes or physical main memory bandwidths in the hundreds of GB/s.

PHYSICS SIMULATION PIPELINE

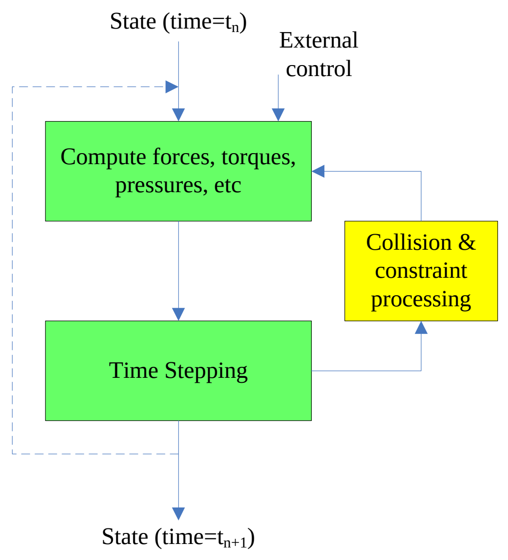

State (time=t_n)
External control
Compute forces, torques, pressures, etc
Collision & constraint processing
Time Stepping
State (time=t_{n+1})

Figure 1: Overview of physical simulation

Figure 1 shows a typical time step in a physical simulation application. For each time step of a simulation, a physical simulation application takes as input the state of the simulated scene (e.g., positions, orientations, and velocities of all objects), as well as external control information (e.g., what the player is doing in a game). The application then computes the physical processes that potentially lead to an updated state (e.g., force, torque, or pressure generation). Depending on the scheme used, this information will be used to advance the state forward in time (e.g., time integration of the laws of motion), yielding a new candidate state. Should phenomena such as collisions or other constraints be triggered, the state may be updated in response to this collision or constraint, and the force computation/time stepping phases will be repeated, possibly with a smaller time step.

While game and production physics share the same iterative process, they exhibit important differences. These differences stem from the execution time requirements of their domains. *Production physics* is primarily used for special effects in movies and other off-line simulations. The execution time limit for these environments is typically a few minutes per frame in order to simulate the complete effect in a reasonable amount of time (e.g., less than a day). *Game physics*, on the other hand, is concerned with real-time simulation used in computer games. Thus, the execution time limit is at most tens of milliseconds per frame. Both areas of physical simulation have the goal of providing maximum visual plausibility within their execution time requirements.

We now describe how we model some specific phenomena.

Fluid Simulation

Production physics: Simulated water volumes are key elements in an increasing number of feature films, making fluid simulation (a.k.a., Computational Fluid Dynamics, or CFD) very common in the special effects industry today. Our production-quality fluid simulation application models a body of water with a free surface (as opposed to water flowing in a pipe or other airtight container). The application uses a combination of a three-dimensional grid and a set of particles [4]. The simulation tracks the velocity and pressure of the water in each grid cell. It computes how velocity and pressure change at each time step using incompressible Navier-Stokes equations. This is very computationally expensive, and it becomes much more so as the number of grid cells goes up. Unfortunately, unless prohibitively large grid resolutions are to be used, the grid cannot accurately represent intricate geometrical features of the water surface (such as thin sheets and droplets). Therefore, particles are sprinkled around the surface and advected along with the fluid. The updated positions and velocities of these particles are used to enhance the resolution of the water surface.

Game Physics: While CFD is the method of choice for high-fidelity simulation of fluids, its high computational requirements necessitate off-line rendering. Game physics therefore uses much faster, although less accurate, techniques. Smoothed Particle Hydrodynamics (SPH) has recently emerged as a popular technique for interactive simulation of fluids [9]. The SPH method represents a fluid as a set of discrete particles and models a resistance to density changes: when particles get too close to one another, a repulsive force separates them; when they get too far from each other, an attractive force brings them together. If a pair of particles is far enough apart, no forces act between them. SPH discretizes the Navier-Stokes equations and samples its solution at a finite number of such particles in space and time. While in the grid-based method the position of these sample points is fixed, in the SPH the particles are free to move around. This difference fundamentally changes the way the Navier-Stokes equations are solved and generally leads to much smaller complexity and ease of implementation, making SPH more suitable for interactive environments.

Cloth Simulation

Production physics: Cloth simulation models a cloth surface that can deform under the influence of external forces such as gravity or forced stretching, and internal forces such as the elastic response to tensile stress, shearing, and bending [2]. This application also models collisions of the piece of cloth with itself and other elements in the environment. The deformable cloth is modeled as a set of mass particles connected to form a triangle mesh. The mesh is endowed with a network of spring elements aligned with all triangle edges and altitudes, as well as between adjacent triangles. These springs model the cloth's resistance to various forms of deformation. Collision detection and resolution is a key part of this application. After the velocities and positions of the cloth particles are updated, collisions are detected. If the collisions cannot be resolved, the application undoes the updates from this iteration and re-executes it with a smaller time step.

Game Physics: Similar to production physics, game physics models a cloth object as a set of particles [7]. Each particle is subject to external forces, such as gravity, wind and drag, as well as various constraints. These constraints are used to maintain the overall shape of the object (spring constraints), and to prevent interpenetration with the environment (collision constraints). The particle's equation of motion resulting from applying the external forces is integrated using explicit Verlet integration. The above-mentioned constraints create a system of equations linking the particles' positions together. This system is solved at each simulation time step by relaxation, that is, by enforcing the constraints one after another for a fixed number of iterations. This method is less accurate but faster than the Conjugate Gradient solver [7] used in production physics, which enables the game cloth to simulate in real time. In addition, self-collisions are typically ignored.

Face Simulation

Production physics: Face simulation animates a model of a human face to provide an anatomically correct visualization of a person speaking or making facial expressions [12]. The application we examine assumes that inertia has a negligible effect on human faces in typical situations, and it therefore models facial motion as a sequence of steady states. Each state is defined by facial muscle activation and the position of the cranium and jawbone. The face is modeled as a tetrahedral mesh, which is driven by the facial musculature and the motion of the jawbone. The application takes as input a time sequence of muscle activation values and kinematics parameters for the jaw motion. The finite element method is used to define the forces (elastic deformation resistance and active muscle contraction) that act on the face and determine its shape.

Rigid Body Simulation

Game Physics: Rigid body dynamics [3] simulates motion and interaction of non-deformable objects when forces and torques are present in the system. Rigid body dynamics is the most commonly used physical simulation in video games today. Examples of rigid bodies in games are vehicles, rag dolls, cranes, barrels, crates, and even whole buildings. The traditional approach solves a system of ordinary differential equations, which represent Newton's second law of motion, `F=ma`, where `m` is the mass of an object, `a` is its acceleration, and `F` is the applied force. The applied force determines the acceleration of the object, so velocity and position are obtained by integration of the above equation. The main computational challenge comes from the fact that rigid bodies' motion is constrained due to their interaction with the environment. For example, consider a destructive environment in a video game where 1000s of rigid objects explode, collapse, and collide, resulting in 100,000s of interactive contacts. To realistically simulate such a scene requires determination of collisions, calculation of collision contact points, and physically correct computation of the contact forces that result from these contacts. To accelerate collision detection relies on spatial partitioning data structures, such as grids or bounding volume hierarchies. To determine contact forces that result from collision contact, we model the contact as a linear complementary problem [1].

Rigid body simulation in games today assumes that rigid bodies cannot break. In general, this assumption is not true in production physics. Today's films use animation of elasticity and fracture. However, these techniques are too slow for interactive use.

PARALLELIZATION METHODOLOGY

The applications we study are all computationally demanding—on a 4-way Intel Xeon processor 3.0GHz system, with 16GB of DDR2-3200 and three levels of cache on each processor (16KB L1, 1MB L2, and 8MB L3), they take on average 188, 14, and 5 seconds to process a single frame for production fluid, face, and cloth simulations, respectively. Similarly, high-complexity scenes in game rigid body dynamics, fluid and cloth take on average 1, 0.4, and 0.1 seconds to process a single frame. While game physics performance may seem much better than production physics, one needs to perform at least 30 frames per second for real-time interactive experiences. Since they will all benefit from a large performance boost, we parallelize the applications, targeting a multi-core processor with tens of cores.

We took the conventional approach to parallelizing large code bases. We first profiled each application to determine the most expensive modules in a serial execution. After that, we prioritize the modules of each application and parallelize them in decreasing order of importance.

The applications were parallelized using the fork-join model in which the program consists of alternating serial and parallel sections. This model is attractive because it allows one to start with a serial program and selectively parallelize the most profitable portions of the program until satisfactory performance is achieved. We use a standard task queue technique [8], similar to Intel Thread Building Blocks (TBB) [6] and OpenMP [10], to parallelize all modules.

In the rest of this section, we discuss how the various modules were parallelized to scale to a large number of cores.

Parallelizing Loops

The majority of modules were parallelized via loop parallelization. These modules typically involve operations on arrays of elements, such as grid cells of a 3D grid (production fluid), an array of particles (game fluid), vertices of a triangle mesh (production cloth), and contact constraints (game rigid bodies).

In most of the cases, the iterations of the loops are independent of each other. For instance, computing the aggregate force on a vertex of the triangular mesh (production cloth) requires simply adding all the forces on that vertex. These loops are parallelized by partitioning the iterations of the loop among the cores. In a few instances, multiple iterations update the same piece of data. However, even in these instances, the final result is independent of the ordering of the iterations. These loops are also parallelized by splitting iterations among cores while using fine-grained locking to guard updates on the shared data.

Parallelizing Graph Operations

A few modules have more complex forms of parallelism and typically incur more parallelization overheads.

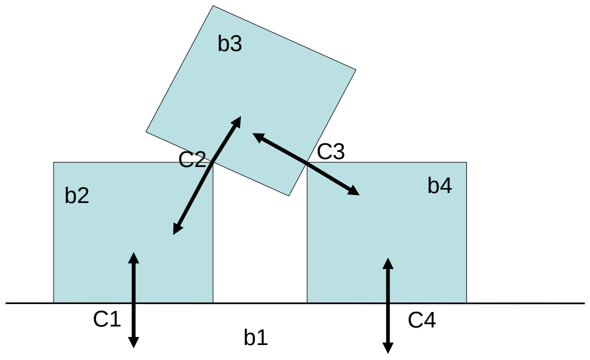


Figure 2: Scene configuration

Bodies → / Constraints ↓	b1	b2	b3	b4
	C1	C1		
		C2	C2	
			C3	C3
	C4			C4

Figure 3: Constraints between various objects

	b1	b2	b3	b4
Time 1	C1	C1	C3	C3
Time 2	C4	C2	C2	C4

Figure 4: Reordered constraints into two batches to expose parallel computation

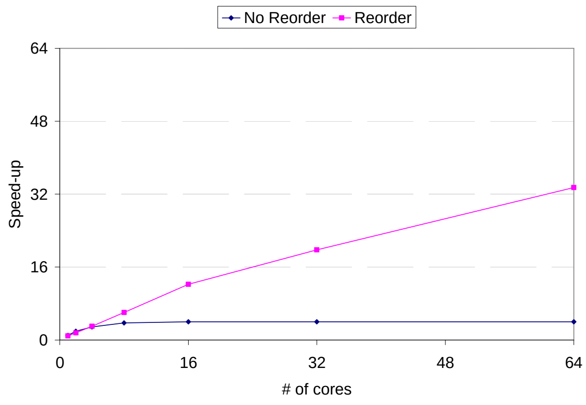


Figure 5: Relative speedups for the constraint solver with and without reordering of the constraints

An example of this is the broad-phase in collision detection. Collision detection requires checking every pair of objects for collisions. Since only a small fraction of the objects actually collide at any given instant, collision detection is performed in two phases: a *broad-phase* that performs quick checks to rule out a large fraction of the object pairs, and a *narrow-phase* that performs the exact (and more computationally expensive) check on the remaining pairs. A standard technique to accelerate the broad-phase is to build a bounding volume hierarchy (a tree) containing the objects. A leaf node of this tree consists of a single object. The computation starts at the root and traverses down. At each step, it checks pairs of nodes of the tree. If the bounding volumes at the two nodes do not overlap, then none of the objects in the first subtree can possibly collide with any object in the second subtree. Otherwise, more checks have to be performed using the children of the two subtrees. Each of these pairs of subtrees represents independent computation and can be performed in parallel. Consequently, in the broad-phase, each unit of parallel work can spawn off more parallel work.

Using Alternative Algorithms

Sometimes, the best serial algorithm has poor parallel scalability. In such cases, we often use an alternative algorithm whose serial version is not as efficient as the original, but whose parallel version scales much better. Sometimes, we use an additional phase to reorder data and expose more parallelism. In this section, we describe two specific examples in detail.

The first example is from rigid-body dynamics from game physics. During the execution of the physical simulation pipeline, the collision detection phase computes the pairs of colliding bodies, which are used as inputs to the constraint solving phase. The physics solver operates on these pairs and computes the separating contact forces, which keep the bodies from inter-penetrating each other. In Figure 2, we show one such case involving four bodies (three boxes and one ground plane), where the corresponding pairs of colliding bodies are listed in Figure 3. The resulting constraints C1, C2, C3, and C4 need to be resolved to update the body positions.

To parallelize this phase, we would ideally like to distribute the constraints amongst the available threads and resolve them in parallel. However, there is often an inherent dependency between consecutive constraints. In our example, constraints C1 and C2 both involve body b2 and thus cannot be resolved in parallel. These dependencies can force a significant serialization of the computation. However, we can reorder the constraints into different batches such that there are no conflicting constraints in each batch. That is, each batch will contain at most one constraint that refers to any given body.

Reordering algorithms traverse the constraints, maintaining an ordered list of partially filled batches. Each constraint is assigned to the earliest batch with no conflicting constraints. As a result, all constraints within a batch can be processed in parallel, while the different batches have to be processed sequentially.

For example, we reorder the constraints in Figure 3 and obtain two batches, (C1, C3) and (C4, C2), as shown in Figure 4. Note that C1 and C3 from the first batch do not refer to any body more than once and can be resolved in parallel. A similar observation holds for C4 and C2. As a result C1 and C3 in the first batch are solved in parallel and the results are fed as part of the input to the second batch. The bottom curve in Figure 5 shows the speedup of the physics solver using the original order of the constraints relative to the serial version for up to 64 cores. The top curve shows the speedup using reordered constraints. We see that without reordering, the speedup is limited to 4× on 64 cores. However, reordering the constraints enables a speedup of 35×, including the overhead for reordering. This example highlights the case where some extra computation needs to be performed to expose the parallelism.

The second example of the need for alternative algorithms is from fluid simulation for production physics. Our fluid simulation application implicitly tracks the interface (boundary) between the air and the fluid. For each grid cell in the modeled space, it computes the distance to the interface. The most common technique to do this is the Fast Marching Method (FMM), which iteratively advances the wave front. For each iteration, it finds and updates the closest grid cell to the front that is not already on or behind the front (Figure 6). This is inherently serial. However, these distance values are required only for a narrow band around the interface. Thus, we parallelize FMM by dividing the grid into overlapping blocks, padded by the width of the narrow band, and working on each block independently (Figure 7). This works well for a small number of threads. However, the total overlap region becomes large quickly as the number of blocks increases. As a result, the application scales relatively poorly, achieving a scaling of around 21× on 64 threads.

We instead use an alternative scheme known as the Fast Sweeping Method (FSM) [5] (Figure 8). FSM traverses the grid cells in all eight possible combinations of the X, Y, and Z directions. Each “sweep” updates the distance of a cell from the distances computed for its neighbors in the previous sweeps. We obtain the correct distance for each cell after completing all eight sweeps. We parallelize FSM in a similar manner to FMM (i.e., with overlapping blocks). The serial version of FSM is around 30% slower than the serial version of FMM. However, FSM has more parallelism since the sweeps are independent. Thus, we achieve scaling of around 55× on 64 threads. In Figure 9, we compare the speedup of FSM relative to FMM. Up to 16 cores, FMM provides higher performance, but beyond 16 cores, FSM is better, giving about 2× the performance of FMM on 64 cores.

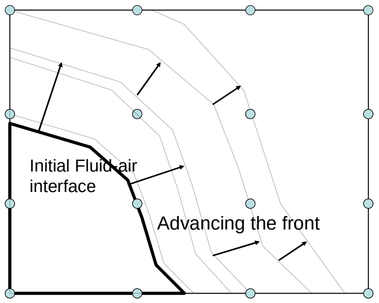


Figure 6: Fast Marching Method (FMM) advances the front to incrementally compute the signed distance of nodes from the interface

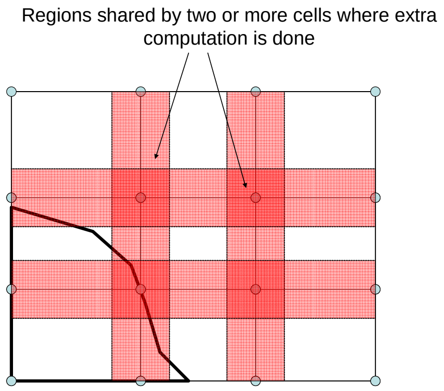

Regions shared by two or more cells where extra computation is done

Figure 7: Parallelizing the algorithm by dividing the region into overlapping cells

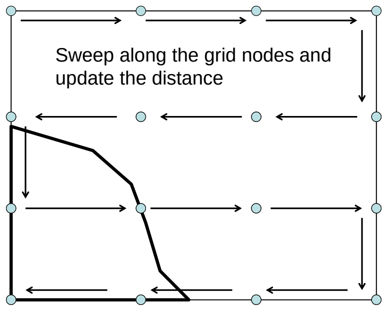


Figure 8: Fast Sweeping Method (FSM) traverses the grid nodes and incrementally updates the minimum distance to the interface

PARALLEL SCALABILITY RESULTS

Figure 10 shows the parallel scalability for our applications for up to 64 cores. Since no large-scale CMP is available for us to experiment with, we use cycle-accurate simulation to measure performance and characterize the parallelized workloads. Details of our simulator can be found in [5]. We assume a very high main memory bandwidth so that we do not artificially limit scalability. The speedups are obtained against the one thread version of the parallelized code. On 64 cores, we achieve 30× to 56× speedup for production physics and 36× to 61× speedup for game physics.

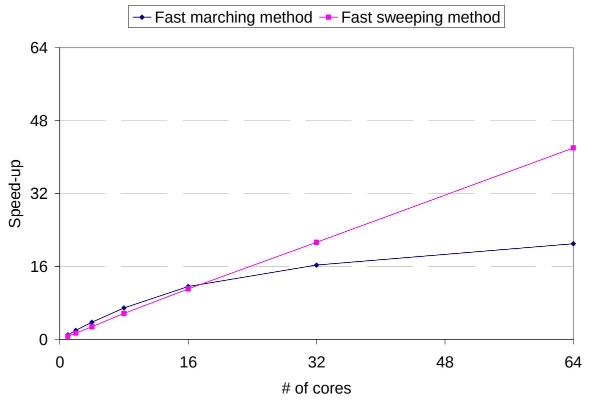


Figure 9: Speedup of FSM relative to FMM

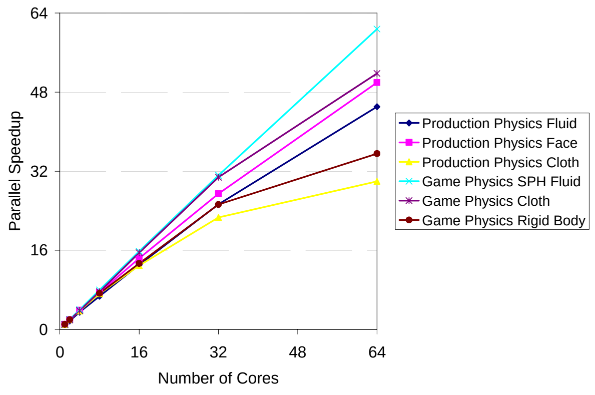


Figure 10: Parallel scalability of production physics and game physics

Next, we discuss important issues regarding scalability. Amdahl's law determines the theoretical maximum scalability. Load balancing and synchronization overheads impact how close we can be to the theoretical limit.

Serial Sections: Amdahl's law dictates that the parallel scalability is limited by the size of the serial sections. In most of the production and game physics modules, the serial section accounts for much less than 1% of execution time for one core. As a result, it does not significantly impact the parallel scalability in our study of up to 64 cores. However, as the number of cores increases, more aggressive parallelization will be needed to keep serial code from limiting parallel scalability.

Load Imbalance: The load imbalance is a function of the variability of task size as well as the number of tasks. The lower the variability, the fewer tasks are needed to obtain good load balance. Unfortunately, some modules exhibit high variability, which requires many tasks for good load balance, resulting in high parallelization overhead. Therefore, we should make a tradeoff between good load balance and low parallelization overhead. Under certain instances (e.g., Figure 7), we are forced to minimize the number of tasks to keep the amount of replication and redundant computation at an acceptable level. However, this comes at a cost of significant load imbalance that may limit parallel scaling.

Task Queuing: We implement a task queue to distribute parallel tasks across the cores. For some modules, the task queue overhead becomes a bottleneck for the scalability. In our implementation of task queues, all tasks are enqueued before we enter the parallel section. Therefore, if the number of tasks is large and/or the parallel section is small, the enqueue overhead becomes significant. Note that an alternative implementation of task queues might solve the problem, one of which is discussed in [8].

Locking: Grabbing and releasing locks incurs synchronization overhead. However, we observe that the locking overhead does not increase with the number of threads. Since there is little contention on the locks, locking does not significantly limit scalability. Nevertheless, accessing an uncontended lock still incurs parallelization overhead as it is extra work that is not required in a serial code.

In addition to the reasons listed above, parallel scaling is also affected by the memory behavior, which is covered in detail in the next section.

IMPLICATIONS FOR THE MEMORY SUBSYSTEM

Memory bandwidth requirements grow proportionally to the number of cores on a multi-core chip. Furthermore, as applications and workloads evolve, memory bandwidth requirements are expected to grow. Current server memory bandwidth projections are mostly based on traditional benchmarks such as TPC-C, SPECjAppServer (SJAS), and SPECjbb (SJBB). Unfortunately, these benchmarks do not accurately reflect future important workloads such as our physical simulation applications.

Figure 11 shows the projected external memory bandwidth requirements for five different sizes of last-level cache (the other caches are assumed to be small and inclusive). The projection is based on running the workloads at 64 giga-instructions-per-second (GIPS). We analyze the bandwidth requirements for all important modules and compare them to TPC-C, SJAS, and SJBB. For each cache size, the modules are sorted according to their bandwidth requirements. The bandwidth requirements for the traditional benchmarks are highlighted for comparison.

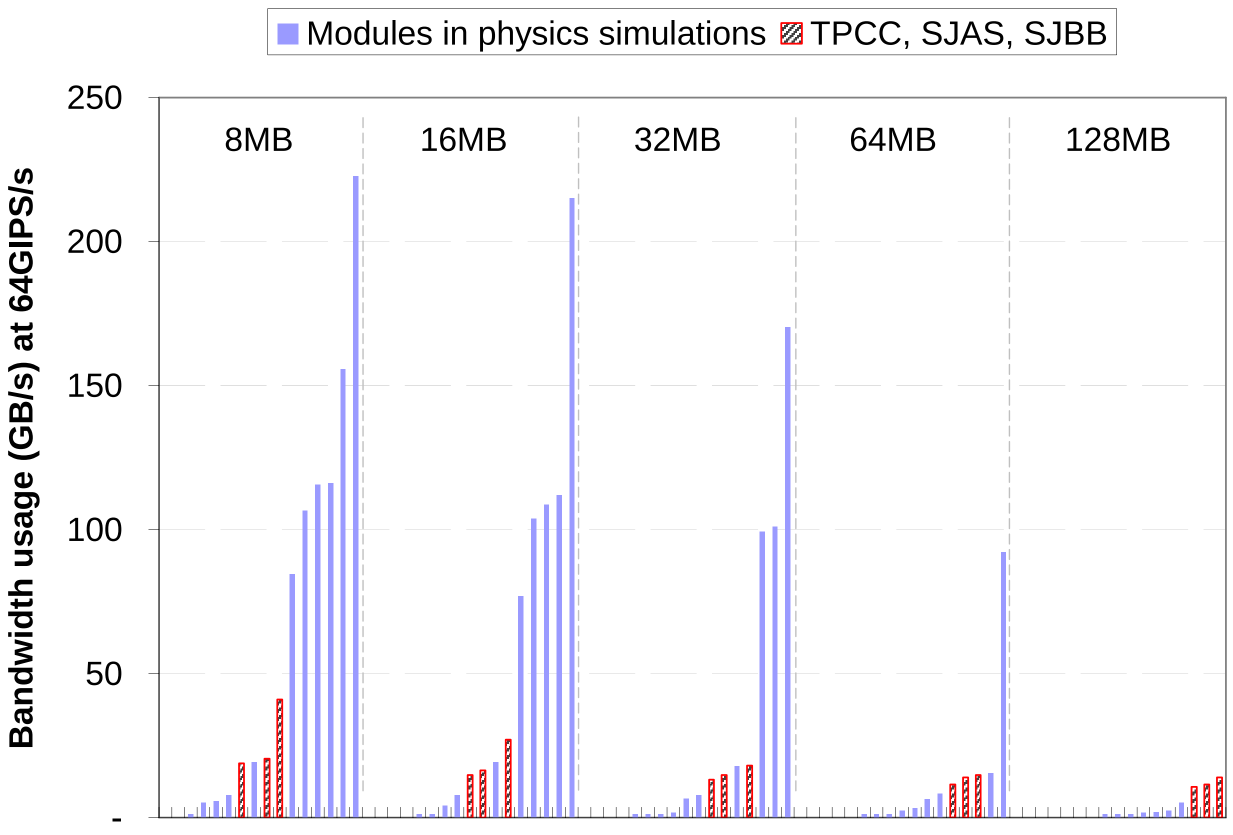


Figure 11: Projection of external memory bandwidth requirements (GB/s) for a given last-level on-die cache size

The results show the following behaviors:

(1) If we have less than 128MB of last-level cache, modules in physical simulation have a wide range of bandwidth requirements, ranging from a few gigabytes per second to over 200GB/s. The bandwidth usage of traditional benchmarks, on the other hand, is much lower than that (maximum of 40GB/s, even if we have only 8MB of cache).

(2) To put the results into context, we compare the requirements to projected available bandwidth in 2010. Memory bandwidth typically grows at 30% per year, so we expect the available bandwidth to be about 48GB/s in 2010. Workloads with bandwidth requirements greater than this will suffer performance-wise. Some of our modules have bandwidth requirements that greatly exceed 48GB/s unless the last-level cache is at least 64MB.

(3) The average bandwidth usage for each of the applications is significantly lower than the peak bandwidth usage. This is because each application is made up of modules with different bandwidth requirements. The scalability of the module with the highest bandwidth requirement often limits the scalability of the entire application.

(4) Our physical simulation modules benefit significantly more than traditional benchmarks do from a large last-level cache. When an application's entire working set fits into cache, the external memory bandwidth usage becomes minimal. For our applications, this happens when the cache is 128MB.

(5) One of our most memory-intensive modules is the incomplete Cholesky Preconditioned Conjugate Gradient (PCG) method from production fluid simulation. PCG is used to solve a system of equations arising from the discretization of the Poisson Equation.[1] It consists of a number of

[1] PCG is one of the most popular approaches for solving large symmetric positive-definite systems of equations because it is more robust than direct solvers and converges fast. As such, PCG is of great importance beyond the study of this application.

operations performed sequentially on a set of two matrices and a number of vectors. The solver iterates tens of times until the solution converges. During each iteration, both matrices (which occupy about 40MB each) are streamed over. Thus, we see a huge bandwidth requirement when the last-level cache cannot hold the matrices. When the last-level cache is big enough to hold both matrices (and all the vectors), the bandwidth requirement is greatly reduced.

CONCLUSION

We consider two broad categories of physical simulation applications: production physics and game physics. Production physics is used by movie studios for creating special effects that may take many minutes to process a single frame. In contrast, game physics is used by the gaming industry and has a more stringent real-time requirement of about 30-60 frames per second. The difference in execution time requirements affects the choice and design of algorithms for the two categories of physical simulation.

We have parallelized applications in both categories and achieve parallel scalability of 30-60× on a cycle-accurate simulator of a multi-core chip with 64 cores. Many modules of these applications require extensive effort to achieve good performance scaling. In some cases, the best serial algorithms have poor parallel scalability. For these, we use alternative algorithms which are slower on one core, but have more parallelism. In other cases, we modify the algorithm to expose more parallelism. The overhead of exposing the parallelism is often small compared to the benefits of improved scaling.

While our applications scale well, some modules are far from the theoretical maximum scaling. This is primarily due to overheads in the task queues and to imperfect load balancing.

Some modules also have significant overheads from locking, but these overheads do not grow with the number of cores (i.e., the locks have low contention), and therefore do not impact scalability. However, the cost of locking still has a significant impact on the overall performance of the parallelized application.

We find that future physics workloads will require large last-level caches (i.e., 128MB) or main memory bandwidths in excess of 100GB/s. This is due to the applications' use of streaming access patterns combined with large data sets (e.g., tens of thousands of objects for game physics and hundreds of thousands to a few million objects for production physics).

We also find that physical simulation applications have very different memory characteristics than traditional benchmarks such as TPC-C, SPECjAppServer, and SPECjbb. These traditional benchmarks do not get a big boost from a large last-level cache since their working sets are extremely large. However, physical simulation applications benefit greatly from a 128MB cache since it can fit the whole working set of all application modules.

ACKNOWLEDGMENTS

We thank Radek Grzeszczuk, Matthew J. Holliman, Richard Lee, Andrew P. Selle, and Jason Sewall for their contributions to the project. We also thank Pradeep Dubey who encouraged us to look into this problem.

REFERENCES

[1] D. Baraff, "Physically Based Modeling: Principals and Practice," *Online Course Notes, SIGGRAPH*, 1997.

[2] R. Bridson, S. Marino, and R. Fedkiw, "Simulation of Clothing with Folds and Wrinkles," in *Proceedings of the Eurographics Symposium on Computer Animation*, 2003.

[3] D. H. Eberly, *Game Physics*, Morgan Kaufmann/Elsevier, San Francisco, 2003.

[4] D. P. Enright, S. R. Marschner, and R. P. Fedkiw, "Animation and Rendering of Complex Water Surfaces," *ACM Transactions on Graphics*, 21(3):736–744, July 2002.

[5] C. J. Hughes, R. Grzeszczuk, E. Sifakis, D. Kim, S. Kumar, A. P. Selle, J. Chhugani, M. Holliman, and Y.-K. Chen, "Physical Simulation for Animation and Visual Effects: Parallelization and Characterization for Chip Multiprocessors," in *Proceedings of the 34th International Symposium on Computer Architecture*, June 2007.

[6] *Intel® Thread Building Blocks Reference*, 2006, Version 1.3.

[7] T. Jacobsen, "Advanced Character Physics," *Game Developers Conference*, 2001.

[8] S. Kumar, C. J. Hughes, A. Nguyen, "Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors," in *Proceedings of the 34th International Symposium on Computer Architecture*, June 2007.

[9] M. Muller, D. Charypar, and Markus Gross, "Particle-based fluid simulation for interactive applications," in *Proceedings of the Eurographics Symposium on Computer Animation*, 2003.

[10] *OpenMP Application Program Interface*, May 2005, Version 2.5.

[11] *PhysBAM physical simulation package*, at http://graphics.stanford.edu/~fedkiw*

[12] E. Sifakis, A. Selle, A. Robinson-Mosher, and R. Fedkiw, "Simulating Speech with a Physics-Based Facial Muscle Model," in *Proceedings of the Eurographics Symposium on Computer Animation*, 2006.

[13] R. Smith, "*Open Dynamics Engine*, at http://www.ode.org*

AUTHORS' BIOGRAPHIES

Yen-Kuang Chen is a Principal Engineer in the Corporate Technology Group. His research interests include developing innovative multimedia applications, studying the performance bottleneck in current architectures, and designing next-generation microprocessors/platforms. He is one of the key contributors to Supplemental Streaming SIMD Extension 3 in the Intel® Core™2 processor family. He received his Ph.D. degree from Princeton University. His e-mail is yen-kuang.chen at intel.com.

Jatin Chhugani is a Staff Researcher in the Corporate Technology Group. His research interests include developing algorithms for interactive computer graphics, parallel architectures, and image processing. He received his Ph.D. degree from The Johns Hopkins University, Baltimore, MD. His e-mail is jatin.chhugani at intel.com.

Christopher J. Hughes is a Staff Researcher in the Corporate Technology Group. His research interests are emerging workloads and computer architectures, with a current focus on parallel architectures and memory hierarchies. He received his Ph.D. degree from the University of Illinois at Urbana-Champaign. His e-mail is christopher.j.hughes at intel.com.

Daehyun Kim is a Senior Research Scientist in the Corporate Technology Group. His research interests include parallel computer architecture, intelligent memory systems, and emerging workloads. He received his Ph.D. degree from Cornell University. His e-mail is daehyun.kim at intel.com.

Sanjeev Kumar is a Staff Researcher in the Corporate Technology Group. His research interests are parallel architectures, software, and workloads especially in the context of chip-multiprocessors. He received his Ph.D. degree from Princeton University. His e-mail is sanjeev.kumar at intel.com.

Victor Lee is a Senior Staff Research Scientist in the Corporate Technology Group. His research interests are computer architecture and emerging workloads. He is currently involved in defining next-generation chip-multiprocessor architecture. He received his S.M. degree from the Massachusetts Institute of Technology. His e-mail is victor.w.lee at intel.com.

Albert Lin is a graduate intern in the Corporate Technology Group. His work is primarily in memory systems for future processors with many cores. He received his B.S. and M.Eng. degrees in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology. While at MIT, he was a recipient of the Siebel Scholar Fellowship and a student member of the American Academy of Achievement. He is currently an Electrical Engineering doctoral candidate at Stanford University. His e-mail is albert.c.lin at intel.com.

Anthony D. Nguyen is a Senior Research Scientist in the Corporate Technology Group. His research interests include developing emerging applications for architecture research and designing the next-generation chip-multiprocessor systems. He received his Ph.D. degree from the University of Illinois, Urbana-Champaign. His e-mail is anthony.d.nguyen at intel.com.

Eftychios Sifakis is a visiting researcher in the Corporate Technology Group. He received B.Sc. degrees in Computer Science and Mathematics from the University of Crete, Greece in 2000 and 2002, respectively, and he received his Ph.D. degree in Computer Science from Stanford University in 2007. His research focuses on simulation and analysis of human body and face motion and simulation algorithms for deformable solids. He has been working with Intel since 2005 on the mapping of physics-based simulation on chip-multiprocessors. His e-mail is eftychios.d.sifakis at intel.com.

Mikhail Smelyanskiy is a Senior Research Scientist in the Corporate Technology Group. His research focus is on building and analyzing parallel emerging workloads to drive the design of the next-generation parallel architectures. He received his Ph.D. degree from the University of Michigan, Ann Arbor. His e-mail address is mikhail.smelyanskiy at intel.com.

THIS PAGE INTENTIONALLY LEFT BLANK

For further information visit:

developer.intel.com/technology/itj/index.htm