Practical Game Architecture for Multi-core Systems
By Katrina Archer

Today’s developers create games for a wide array of platforms—from mobile devices and the Web to sophisticated multi-core processor systems like PCs and consoles such as Microsoft* Xbox 360* and Sony* PlayStation* 3. When creating games for a single platform with fixed hardware, you have the luxury of making the best possible use of the hardware configuration. Because of market realities, many game developers do not possess the resources to build special-purpose game engines for each platform. Given the rapid advance in PC hardware and the number of years the Xbox 360* and PlayStation* 3 consoles have been on the market, today’s personal computers often easily outperform the consoles. However, with a bit of ingenuity it’s possible to design a flexible game architecture that performs well on multiple, different platforms.

Is Memory Architecture Important?

Personal computer systems as well as Xbox 360* and PlayStation* 3 consoles offer the power of multi-core processor systems. These days, a PC can possess an arbitrary number of processors, each with multiple cores. Three cores sit inside an Xbox 360* console, whereas the PlayStation* 3 has a single main core but eight synergistic processing elements (SPEs), of which six are usable by developers.

Both PCs and the Xbox 360* have a Unified Memory Architecture (UMA). Conversely, the PlayStation* 3 has a Non-unified Memory Architecture (NUMA). Each SPE in the PlayStation* 3 comes with an synergistic processing unit (SPU), 256 KB of local memory storage—all manually managed—and no conventional memory cache. Data must be transferred using direct memory access (DMA) and pulled from main memory into local memory on the SPE before any processing occurs on the SPU.

Optimal Cross-platform Coding Considerations

UMA-style development does not work well on a NUMA system: The algorithms are not efficient. On the PlayStation* 3, incorrectly structured data leads to wasted cycles trying to move data back and forth between main memory and the SPEs. If more than one SPE needs access to the same data set in main memory, the risk of deadlocks increases, resulting in a need for complicated scheduling algorithms. The size of the data set becomes important: Keeping data in small chunks avoids memory access deadlocks.

An algorithm optimized for processor speed that runs quickly on UMA but ignores data movement costs results in wasted cycles on a NUMA system. Random access of memory consumes too many cycles performing data transfers to local memory. Happily, discretizing data for optimal NUMA
DMA transfers produces better locality and boosts performance on a UMA platform because of greater cache coherence.

Because of the limitations of the PlayStation* 3 design, developers writing games typically find that it’s better to write code for the PlayStation* 3 and adapt it for the Xbox 360* and the PC. Doing so involves using similar algorithms and data structures on all platforms, then on the Xbox 360* and PC conditionally compiling out the specialized DMA code that performs the data transfer for the PlayStation* 3.

**Coarse- and Fine-grained Parallelism**

Different games require different basic architectures. Many single-player games and even non-networked multiplayer games do a certain amount of work in a given period of time—either a frame of rendering time or another convenient time step. Some threads, such as the renderer, may run asynchronously. The game reads the controller input, updates the player-controlled characters, artificial intelligence (AI) characters, and the game’s overall state machine, and then sends the results to the renderer.

A coder can use the different processors and their cores, assigning various game systems to run in parallel on different threads. The key is to keep each core or SPE as busy as possible while minimizing bottlenecks and idle time. A networked game, as a result of latency and deterministic issues, uses different techniques beyond the scope of this article.

**Coarse-grained Parallelism**

*Coarse-grained parallelism* divvies up whole game systems at the thread level. Larger systems are assigned to their own thread. On the Intel® Core™ i7 and Xbox 360*, each core offers two hardware threads. The CPU maintains two contexts or register sets—one for each hardware thread—and swaps quickly between each thread as needed.

One way of breaking down game systems puts all simulation work—controller, game state and logic, and AI—in one thread, while rendering and audio are each in their own threads. For example, on the Xbox 360*, simulation, rendering, and audio each get their own hardware thread on a separate core (Figure 1).
The PlayStation* 3 requires a more hybrid approach (Figure 2). You can use the two hardware threads on the single main core, and then allocate threads to each of the SPEs. One option puts the simulation thread on one of the main core threads, with rendering on the other. However, doing so causes the renderer to compete with the simulation thread, because unlike on Xbox 360*, they are running on the same core. Because games are often render-bound, it helps to allocate an entire SPE to aiding the main rendering thread by pipelining and sending work out to the graphics processing unit (GPU) on jobs that require heavy communication with the GPU.

Figure 2. PlayStation* 3 thread layout
Audio is an important component to parallelize, because neither the Xbox 360* nor the PlayStation* 3 has much dedicated audio. In addition, unlike simulation and rendering, where the user experience degrades linearly with loss of performance, audio degrades all at once, usually in the form of clicks and pops. Therefore, on Xbox 360, the audio thread is often assigned to its own core; audio has its own SPE on the PlayStation* 3.

**Fine-grained Parallelism**

*Fine-grained parallelism* breaks down work at the job or task level. A batching system manages the distribution and queuing of jobs or tasks to different processors.

On the PC or the Xbox 360*, jobs are batched to the unused hardware threads on each available core (Figure 3). A developer who uses three threads for the larger systems described above leaves three secondary hardware threads on the Xbox 360* for fine-grained job queues. On the PC, where you allocate jobs depends on how many cores you have at your disposal. On the PlayStation 3, the jobs are farmed out to one of the remaining unused SPEs.

**Figure 3. PC thread layout**

Good tasks to parallelize in this manner are physics, visibility culling, particle effects, and animation.

With different high-level systems (e.g., simulation, rendering) creating finer-grained job batches (e.g., physics, culling) to put into the queue, a savvy coder can facilitate load balancing by designing the size of each batch to take roughly the same amount of computation time. In practice, this requires extensive profiling and tuning to eliminate bottlenecks.

A good batching heuristic normalizes the cost of each job based on the operational expense for each primitive type. For example, in physics collision tests, sphere calculations are cheap, cylinders are more expensive, and convex hulls require the most processing. The batch size sent to the fine-grained job queue changes depending on the type of primitive: The heuristic restricts how many convex hulls appear in a single batch. Farming out smaller-sized but more expensive packets
allows each processor more opportunities to gather work. Packets with wildly varying execution times cause starvation periods. With normalization, no one high-level system waits very long for another system’s job to finish, resulting in higher overall throughput.

Physics’ computationally intensive calculations involve integrating over constraints, doing collision detection, and then resolving interpenetrations of objects. Often, you must iterate over a number of game objects to determine whether object A intersects object B. The first parallelization step assembles the data for an arbitrary number of game objects into packets of collision queries. The job queue then sends the filled packets off for collision detection processing on the fine-grained thread.

Visibility and occlusion culling offload well to a separate thread. In the case of the PlayStation* 3, a brute-force linear culling algorithm caters more to the strengths of the architecture than more sophisticated hierarchical techniques such as binary space partitioning (BSP) trees, octrees, spatial partitioning, portals, or clustering. Tree algorithms are cache-bound and have poor data locality. A simple linear box-to-box visibility test takes as few as five cycles on a single instruction, multiple data (SIMD) processor like the SPU. Packaging hundreds of boxes together allows very quick culling of thousands of objects. This type of test is also stateless and doesn’t require knowing the previous positions of any other object. From frame to frame, the cost remains consistent, unlike the hierarchical algorithms, which require tree rebalancing or pruning of the data set.

Animation jobs, batched on a per-character basis, also parallelize well. Types of operations to throw onto the queue include inverse kinematics (IK), head-tracking, skeletal matrix evaluation, and pose blending. Asynchronous evaluation of particle systems is also a perfect task for another thread.

**Discussion**

It would be easy to conclude that the PlayStation* 3, with one main core and six usable SPEs, offers a significant advantage in performance over the Xbox 360* system’s three main cores. However, with careful tuning, the coarse-grained primary thread work runs faster on the Xbox 360* because of the general-purpose nature of each core. On the PlayStation* 3, the coarse-grained threads’ performance on the main CPU suffers because of core contention.

Given the rapid advance in PC hardware and the number of years the Xbox 360* and PlayStation* 3 consoles have been on the market, today’s personal computers often easily outperform the consoles. The overall performance on a PC depends on the total number of cores in the system and how well the game is threaded.
Why Use a Mixed Coarse/Fine Approach Instead of a Completely Distributed Approach?

It may seem tempting to develop a game system using a completely distributed approach by breaking down each task or job into similarly sized chunks of work: Certainly, it looks simple on the surface and algorithmically elegant. A fine-grained system is infinitely scalable—an advantage as computer architectures add more and more cores. This is a consideration for developers creating games to run on personal computers.

As the granularity becomes smaller, more work can be parallelized, providing a wider array of choices for scheduling and load balancing. Increasing the number of tasks, however, comes with a cost: more overhead for context switching and synchronization and increased memory usage.

The variety of algorithms, dependencies, CPU, and memory usage patterns in game engines makes it difficult to design a one-size-fits-all general tasking system. Many operations have a strict order of dependence. Without complex priority queues, increased parallelism can introduce non-determinism into the game loop, making the flow of code more difficult to follow—an important consideration during the quality assurance phase of a product when large numbers of bugs must be fixed in a few short months.

To produce compelling onscreen characters that behave in interesting ways, the game loop does a large amount of conditional event evaluation that can look at any random game object. The intertwined subsystems in AI and simulation access disparate data in a cache-unfriendly manner. To properly parallelize these operations requires making each and every simulation variable thread-safe. The theoretical performance gained by parallelizing the AI tasks tends to be offset by the cache misses and latency induced by the thread mutexes. Little available PlayStation* 3 or Xbox 360* performance is left “on the table” by keeping the lion’s share of simulation and AI in a single thread. This is partly why even though rendering performance increased dramatically with the hardware shift from single processor to multi-core, gamers haven’t seen the same exponential increase in the number of characters shown on screen for the PlayStation* 3 or Xbox 360*.

The coarse/fine system described earlier scales fairly well, because you can keep adding fine-grained parallelism, if needed. For example, on a PC with eight cores instead of two or four cores, you could offload more particle simulation tasks to an individual core rather than letting the rendering thread deal with those tasks on its own.

Conclusion

In gaming, it’s important to architect a flexible system, because tasks wind up taking different amounts of time depending on what the player does at any given moment. When large explosions fill the screen with particle effects, the game becomes rendering-bound. But when more characters spawn to attack the player or during large crowds scenes, the game becomes simulation-bound.
In the mixed approach described earlier, a well-designed queuing system determines which coarse-grained thread needs to offload more fine-grained tasks at any given time. When combined with accurate profiling results from tools like Intel® Graphics Performance Analyzers, this offers a developer greater choice in making the best use of each core to give the player the most immersive experience possible.

For More Information