Impact of Thread Synchronization and Data Parallelism on Multicore Game Programming

Abu Asaduzzaman, Hin Y. Lee, and Deepthi Gummadi
Department of Electrical Engineering and Computer Science
Wichita State University, Wichita, Kansas, USA

Abstract—Xbox-360 has three cores with six logical threads and the PlayStation-3 has one master core and six independent worker cores. According to the current design trends, multicore processors will be ubiquitous in every game computer. A game engine has many ‘components’ and multithreading is an important technique to parallelize the execution of these components. However, effective programming of multiple threads in multicore systems has challenges including concurrent processing, thread synchronization, data and task level parallelism, and load balancing. In this paper, we investigate the challenges and benefits of thread synchronization and data level parallelism on multicore game engine programming. We implement a multi-object interactive game engine in an 8-core workstation using single-threaded model (STM), multithreaded asynchronous model (MAM), multithreaded synchronous model (MSM), and multithreaded synchronous model with data parallelism (MSMDP). Experimental results show that MSMDP is the best and it reduces the execution time up to 50%.

Keywords: Data level parallelism; game engine; multicore architecture; multithreaded programming; thread synchronization;

1. Introduction

There are many components in a simple modern game engine. According to the flow of operations, important components in a single threaded game engine are: Input, Game Logic, Artificial Intelligence (AI), Physics (engine for collision detection/response), Audio (for sound), and 3D Graphics. A rendering engine called “renderer” is required for 2D or 3D graphics. A graphics package may include scene graph, culling and sorting, skeletal animation and rendering. Inside a component, there may be many subcomponents that ‘glue’ together to form a complete package. Some of these components can be middleware to make programming easier. An operation from start to finish is known as a clock cycle.

In addition to standalone game machines, game engines are nowadays being used for educational, engineering, and scientific applications [1]. To fulfill the high performance requirements, game engines are adopting new hardware technologies like multicore CPUs [2] and software technologies like multithreaded parallel programming [3], [4]. Many parallel programming techniques are available; one or more of them can be used in game engine programming. When components in a game engine are originated from many different middleware, the design of the library will most likely dictate which one is more suitable to be used. Some middleware such as Bullet Physics library includes multithreading in their application programming interface (API). Depending on the type of multithreading model used, some level of data redundancy is required to improve performance. Therefore, a mechanism to ensure data/cache coherency is needed in the implementation.

As the number of cores in a processor increases but the speed of the core has not changed much in the recent years, multithreading can be very helpful to get as much performance out of a system as possible to the advancement of video game technologies. Currently available middleware used in high-level API, like Open MPI, make the parallel implementation a challenge. Therefore, various methods should be evaluated when implementing multithreading in a game because the components usually never work the same way. One multithreading technique might not be suitable for a particular API of a component because of the way it is built; optimization of multithreaded game engines requires a lot of experimentations.

In this work, we implement a multi-object video game engine using middleware from different vendors. Various multithreaded asynchronous and synchronous models, with and without data parallelism, are implemented to study their effectiveness to improve the performance of multicore game engines.

The rest of the paper is organized as follow: Section 2 reviews some related published articles. Section 3 explains the impact of data/task parallelism, and synchronization. A multi-object multithreaded video game engine implementation is presented in Section 4. Some simulation results are discussed in Section 5. Finally, this paper is concluded in Section 6.

2. Background Study

The game industry has surpassed the movie and music industry in U.S. in 2005 and 2007, respectively. In 2008, the game industry surpassed the music industry in the U.K. and is expected to surpass DVD sales in the future. The desire for
more complex game elements is driving the game industry forward. Multithreaded parallel programming has potential to implement complex game engine. However, multicore CPU (not manycore GPU) is a relatively new technology, especially in the game development world.

Multicore architecture is a recent design trend and most vendors are adopting multicore processors to their products. Multilevel cache memories are common in multicore processors [5]. The cache memory hierarchy normally has level-1 cache (CL1), level-2 cache (CL2), and main memory. In most cases, CL1 is split into instruction cache (I1) and data cache (D1) and CL2 is a unified cache [2]. Performance and power consumption are impacted by cache misses, increased usage of main memory, and poor cache memory arrangement. Using communication that is too fine grained can cause the cache to be underutilized [6]. A thread reading the data can receive the set of multiple data objects first and then process all of them. The effective size of the set of data objects can be calculated by the size of each line and the size of the cache line size of the cores. When communication is too coarse grained, capacity misses could happen when there are large amount of objects being copied that are larger than the cache size. Some processors use shared cache (like shared CL2) that can be accessed by multiple cores. When more cores access the same cache, there will be overhead of managing the use of the cache by multiple processors. Multicore systems are very suitable for multithreaded processing as multiple threads can be executed on multiple cores at the same time [5].

Task level parallelism is a popular method for game engine multithreading, where components run asynchronously in their own loop or synchronously in a single loop with multiple forks and joins. An asynchronous model of game engines has been introduced in [7]. In this model, as soon as a task is done, it will run immediately from the beginning. Data sharing could limit the effectiveness of this model depending on the amount of synchronization required. The multithreaded game engine introduced in [7] is an asynchronous model that uses multiple render states to buffer data. In this implementation, there is one ‘world’ state and three ‘render’ states. For a game engine, data parallelism is where the same type of data in a component is parallelized in multiple threads. The use of this in a game engine is when a component spawns multiple worker threads to process one type of data. To properly scale a multicore game engine, task parallelism and data parallelism have to be employed as introduced in [8].

When the cores are used more evenly, it gives more opportunities for developers to implement more distributed and parallel game play elements [9]. Intel Corporation has used a ‘thread pool’ mechanism to manage tasks as discussed in [10]. In a thread pool, each component has one or more tasks that will be queued and threads that are idle or have finished a task will retrieve a task from the queue to run next. This ensures that there will be a maximum amount of threads that can run at the same time. A multicore architecture is integrated to expose multithreading concept to game programmers to different number of cores without recompilation of code.

RedLynx has implemented multithreading in their game Trials HD [11]. It uses the Bullet Physics Engine for physics simulation. The library is optimized in-house for the Xbox 360 CPU and the vector units. One of the new features is the threaded asynchronous resource loading [12]. Developers can load rendering resources in a thread-safe way and use them concurrently with the rendering operation.

In this work, we use thread pooling technique and graphics rendering API to develop a multithreaded game engine to evaluate the impact of synchronization and data parallelism on performance of the game engine.

3. Important Techniques for Game Engine

Some important techniques used in game engines to improve performance are briefly explain in the following subsections.

3.1 Data Level Parallelism

Data parallelism is the distribution of the same type of data to process across different threads. For a game engine, data parallelism is where the same type of data in a component is parallelized in multiple threads [13]. This is used in a game engine when a component spawns multiple worker threads to process one type of data. If only data parallelism is employed, the series of different types of operations are sequential, only the data of a type of operation are processed concurrently at one stage. If the type of data requires communication among themselves, a thread safe communication system has to be implemented. This method scales well for many number of processors because the size of the data for each thread can be divided equally. It may also be easy to balance the load among multiple cores because there is only one type of object being processed concurrently. When the data type does not share data with each other, this method of parallelism can easily be implemented to scale on any number of threads. Communication among the threads can be reduced by grouping the objects that are most likely to interact with each other in the same thread [14].

3.2 Task Level Parallelism

Task parallelism is the distribution of different task across different threads. The use of task parallelism in game engine is by running each component task in its own thread [15], [16]. There are two model of execution for this method: the synchronous model and asynchronous model. The synchronized model is where all the tasks of the components must finish in a single clock cycle. At the end of the clock
cycle, the application will loop to the beginning to start the operations in the same order every time. The asynchronous model is where the tasks of the components can run and finish at their own time. To share data among the threads, a synchronization stage can be used in between the clock cycle in this model. In an asynchronous model all the components run in their own loop. Some components that do not always have new data for a single frame are usually implemented asynchronously such as resource loading, player input, and networking.

3.3 Data and Task Level Parallelism

To properly implement a game engine that will scale properly and fully utilize parallelism for various numbers of cores, both data parallelism and task parallelism have to be employed. A mixture of task and data parallelism is an optimum approach to exploit multithreading in game engines [17]. In this combination, each task can run in parallel with another task and may spawn several worker threads. A system may have number of cores less or more than the number of parallelizable components. In task parallelism, if there are more cores than the number of types of components to be parallelized, then if each of the types of component runs in a single core, there will be cores that are not used. Therefore, to maximize parallelism, data parallelism should also be employed to maximize the use of all cores. A highly data-parallelism design would make it easier to manage tasks that are sequential as there may only be race condition among the same type of data being parallelized but may not fully utilize the concurrency advantage for some components that are decoupled from each other. A highly task-parallelism design would cause some cores to be unused as there may be more cores than the number of different types of tasks that can run at the same time but having a synchronization stage with no mutex locking can be easily implemented if it is the synchronous model. Mixing task and data parallelism takes advantage of the fact that not all components and data objects of a game engine are completely dependent.

3.4 Synchronization

Synchronization with respect to multithreading is basically data synchronization. Synchronization is used to make sure that the same data are not executed at the same time by two threads. One method for synchronization is with the use of mutex (i.e., mutual exclusion). Use of a mutex locking in a game engine depends on the multithreading model. The main drawbacks with mutex locks are overhead, deadlocks, contention, and priority inversion [18].

In some cases, lockless algorithm can be used. In those cases, a game engine is designed so that mutex locking is entirely avoided. The easiest method is to have a synchronization stage where all processes must run in sequence. Another method is to use a message passing system between threads. This avoids the use of mutex locking when passing data. Some other synchronization techniques include reader-writer lock and read-copy-update [9].

4. Test Game Engine

We implement a multi-object game engine: Tower Defense Game (TDG) in our laboratory to investigate the impact of thread synchronization and data parallelism on multicore game engine performance.

4.1 Game Policy

The objective of the game is to defend a main base structure. The player has to build defensive structures that will destroy waves of enemies trying to destroy the main base structure. The player will try to survive as many waves as possible. Enemies will get harder after every wave. For every enemy destroyed, the player gains credits which could be used to build more defenses. Currently, there are three possibilities to terminate the game - (i) the player defends the main base structure for 3 minutes (the player wins the game), (ii) the main base structure is destroyed (the player loses the game), and (iii) abnormal termination.

4.2 Different Implementations

The video game engine is implemented using single-threaded model and various multithreaded models (with and without synchronization). We briefly explain different implementations below.

- **Single Threaded Model (STM):**
  The first operation is to capture input events and put it in a buffer; this is an input / output (I/O) operation with the operating system. The order of operations in the single threaded implementation is:
  1) Capture input
  2) Update input operation
  3) Update game logic
  4) Update AI
  5) Update physics
  6) Process navigational mesh updates
  7) Simulate physics
  8) Render graphics

- **Multithreaded Asynchronous Model (MAM):**
  There are two threads; each thread has independent clock cycle. The update graphics stage in this model reads the data buffered from the updates of the other thread. The order of operations in the single threaded implementation is:
  1) Capture input
  2) Update game logic
  3) Update AI
  4) Update physics
  5) Process navigational mesh updates
  6) Simulate physics
Thread 2:
1) Update graphics
2) Render graphics

- Multithreaded Synchronous Model (MSM):
In this lockless implementation, data synchronization is done in the serial stage only. In the parallel stage, all the operations run in parallel, each on a different thread. The order of operations is:
Serial Stage:
1) Capture input
2) Update logic
3) Update AI
4) Update physics
5) Update graphics
Parallel stage: (One possible order)
1) Process navigational mesh updates
2) Simulate physics
3) Render graphics

- Multithreaded Synchronous Model with Data Parallelism (MSMDP):
This implementation is a combination of task and data parallelism using the multithreaded synchronous model. It is similar to the synchronous model but the physics will have 2 worker threads to process collision detection. In the physics simulation thread, two more threads are spawned during the collision detection stage. This is considered parallelism within a component. The order of the operations is:
Serial stage:
1) Capture input
2) Update logic
3) Update AI
4) Update physics
5) Update graphics
Parallel stage: (One possible order)
1) Update navigational mesh
2) Simulate physics
3) Perform collision detection on object batch 1
4) Perform collision detection on object batch 2
5) Render graphics

5. Experimental Results
In this work, we develop a multi-object game engine using C++ in a multicore computer to explore how the performance of a multicore game engine is influenced due to data parallelism and thread synchronization.

5.1 Important System Parameters
The workstation used in this experiment is an 8-core (dual-processor, quad-core per processor) system from Intel, runs at 2.13 GHz, and has 6 GB of RAM. The operating system used is the Linux Debian 6.0. Output parameters include: the number of frames generated per minute, processing time for each frame, processing time for each component, and speedup factor, $S(P) = T(S)/T(P)$. Where, $T(S)$ is the best sequential time and $T(P)$ is the run time due to the parallel implementation.

5.2 Results and Discussion
We start with exploring the generation of different frames by a single-threaded multi-object game engine. Then we study various multithreaded models of the game engine. We compare the performance and speedup due to various implementations: STM, MAM, MSM, and MSMDP.

First, we examine the number of frames generated due to various implementations for a 3-minute simulation of the game. As illustrated in Figure 1, the multithreaded synchronous model with data parallelism game console generates more frames per minute than any other consoles. Single-threaded game console generates about 4,750 frames per minute, multithreaded asynchronous and synchronous both generate about 5,000 frames per minute, and multithreaded synchronous with data parallelism generates about 5,050 frames per minute.

Second, we investigate the maximum time required to process different frames. As shown in Figure 2, the single-threaded game console takes the maximum amount of time to process a frame when compared with other multithreaded models. MAM shows improvement over STM. Building a proper asynchronous multithreaded engine requires a lot of time and it will be more complex than a synchronized model in most cases. The asynchronous model of execution will most likely require more memory to implement and as such it is only recommended in cases where user experience can be improved by components running at their own clock cycle. It is also observed that multithreaded synchronous model with data parallelism console takes the minimum amount of time to process a frame.
Third, we study the maximum time required to process different components. As shown in Figure 3, the components are found to have different amount of time for processing. The physics component takes up most of the processing time; this is because the game uses a lot of physics objects and operations.

Fourth and finally, we observe the speedup due to the multithreaded implementations over the single-threaded implementation of the test game engine. According to the experimental results, the speedup increases as the number of threads increases (see Figure 4). More than 14x speedup is achieved due to the multithreaded synchronous model with data parallelism for 512 threads. This speedup is impressive. Future video games are expected to be much more complex than what we have today; speedup of such complex systems can be significantly increased by applying MSMDP like GPU computing. In GPU technology, thousands of threads can be generated for computation extensive systems; the threads are then run concurrently in parallel on a multicore CPU with manycore GPU system.

6. Conclusion

Single-processor multithreaded game engines struggle to improve performance due to the lack of hardware support. Recently introduced multicore systems have the potential to improve the performance of multithreaded game engines. However, programming multithreaded game engines for multicore architectures introduces various challenges including data parallelism, thread synchronization, task parallelism, and load balancing. In this work, we explore the challenges due to thread synchronization and data parallelism by developing a multicore video game console (called Tower Defense Game). We develop C++ programs for single-threaded and several multithreaded models with and without data parallelism. Experimental results support the fact that the multithreaded models outperform the single threaded model. Although different game components take different amount of processing times (see Figure 3), the multithreaded synchronous model with data parallelism generates more frames and takes less amount of time to process the frames (see Figures 1 and 2).

On an 8-core system, the speedup due to multithreaded synchronous program with data parallelism is about 14 (see Figure 4) with respect to the single-threaded implementation. It is expected that a much higher speedup will be needed for future game engines, which can be achieved by applying concurrent/parallel programming techniques like GPU computing.

We plan to implement the entire test game engine (i.e., Tower Defense Game) on a CPU/GPU platform to explore the performance and power consumption in our next endeavor.
References


