An Adaptive Control Scheme for Multi-threaded Graphics Programs

Nakhoon Baek*
School of EECS
Kyungpook Nat'l Univ.
Daegu 702-701
Korea

Choong-Gyoo Lim
Digital Contents Lab.
ETRI†
Daejon 305-700
Korea

Kwang-Ho Yang
Digital Contents Lab.
ETRI†
Daejon 305-700
Korea

Youngsul Shin
School of EECS
Kyungpook Nat'l Univ.
Daegu 702-701
Korea

Abstract: In these days, multi-core CPU's are easily available, and thus, multi-threaded programs are preferred. In the field of computer graphics, there is still no reference programming model for the multi-threaded environments. A typical graphics program may have a simulation-rendering loop. When we parallelize this loop structure, we will have performance improvements. Furthermore, we can use a set of threads to more accelerate it with multi-core CPU's. In this paper, we select a sample problem, and represent its parallelized version, to finally present an adaptive control scheme for the number of simulation and rendering threads. Our adaptive scheme is designed to maintain the optimal performance through controlling the number of threads dedicated to different tasks. Simulation results show that this scheme is better than the fixed number of threads cases.

Key–Words: Multi-core CPU, Acceleration, Multithreading, Hyper-Threading, Simulation, Rendering.

1 Introduction

Traditionally, our PC's have a single CPU and a single graphics card to provide multimedia contents including graphics outputs and computer games. Along with the development of semi-conductor technology, we now meet the technical limit of CPU clock speed and it is impossible to speed up it dramatically at this time. Therefore, in these days, we need to apply the parallelism to PC's and its related software[1].

Mainframe computers already met this kind of limits, and they developed various technologies including multi-threading, super-threading and hyper-threading. Nowadays, PC's and game consoles are ready to effectively support multi-processing schemes. For example, dual-core CPU's including Intel Pentium D or Xeon processors are now available.

For these multi-core CPU's, the serialized programming models for traditional single-core CPU's are not suitable for using full CPU powers. Thus, we need to develop multi-threaded programs based on a parallelized programming model.

In addition to improvements in general areas, we also need to focus specialized improvements. In the area of computer graphics, we have GPU's for vertex shader and pixel shader, which are already parallelized. In contrast, research results in usual multi-processing areas focused on the symmetric multi-processor (SMP) architectures, while graphics programs may handle asymmetric cases such as load-balancing between CPU and GPU. Therefore, we need parallelized programming models which are more specialized to interactive computer graphics programs[2, 3].

In this paper, we provide a new parallelized programming model, which uses an adaptive control scheme on the number of threads dedicated to specific parts. In the next section, we present background issues. In section 3, we present our programming model and section 4 shows the implementation results for an example problem. Conclusions are followed in section 5.

2 Problem Formulation

From the view point of hardware, we need advanced technologies such as multi-core CPU's, hyper-threading, etc. Current PC platforms are ready to satisfy these requirements. In particular, major CPU vendors, Intel and AMD already introduced new server and desktop CPU's with dual-core support. Additionally applying hyper-threading technology to dual-core CPU's, we get 4 physical threads at a time[4].

In the case of graphics cards, major graphics chip vendors now provide multi-GPU environment, including nVIDIA's quad-SLI chips. Thus, we can exe-

*corresponding author
†Electronics and Telecommunications Research Institute

cute 4 GPU's simultaneously. In the future, these trends to multi-processor environment will be more emphasized[5]. More recently, the physics processing unit(PPU) is newly introduced. These PPU's are very similar to GPU's and thus will be parallelized in near future[6].

Thus, current PC platforms are already multi-processor environments with three different types of processors: CPU's, GPU's and PPU's. Our focus is now software for these asymmetric multi-processor environments. We need a cooperative way of fully utilizing these heterogeneous processors.

Supports for multi-processing are focused on mainframe computers. In the UNIX operating system, they introduced thread concept as *more lighter process* and now several multi-threading libraries are available in UNIX and PC platforms. *OpenMP* is an API for writing multi-threaded applications, which is implemented as a set of compiler directives and library routines for parallel application programmers. This API is based on the traditional fork-join parallelism[7, 8].

For an efficiently multi-threaded program, we need to naturally reflect the parallelism into the threads, rather than simply dividing the existing program into a set of threads. Along to the development of these thread-smart environments, they will focus on the interactive computer graphics and game programming area. In other words, typical office applications are hard to accelerate with multi-core CPU's, mainly due to their serialized complex software architectures. In contrast, graphics and animation programs have a typical architecture consisting of various modules for simulation, rendering, data loading, audio processing, etc. We may implement all these modules as a set of threads to get the benefits from parallel programming models.

In last few years, several commercial programs already proved that multi-threading can be a solution for various problems. For example, the game *Dungeon Siege* shows that lags during data loading can be avoided through effectively balancing thread loads, even with single-core CPU's[9]. Game engines like *Unreal Engine 3* provide multi-threaded asynchronous background loading techniques, and in near future, more effective game engines will be developed on the basis of multi-threading[10]. In the case of Xbox 360, some games already say that they fully utilized the triple-core CPU in the Xbox 360.

3 Programming Models

To write a parallelized application program, we need to decompose the original problem into a set of tasks which can be simultaneously executed. In this section, we will examine a set of conceptual programming models.

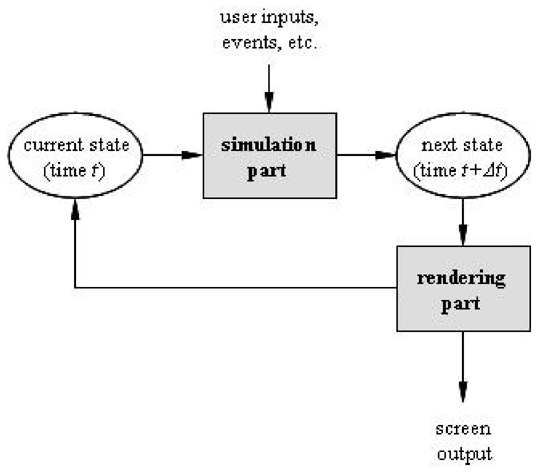

Figure 1: A serialized programming version

3.1 A Serialized Model

In the field of computer graphics, a typical interactive program needs to display the current state on the screen periodically and also needs to response to the user inputs. Considering these aspects, typical graphics programs can be roughly modeled as shown in Figure 1.

Based on the current state at time t, the *simulation part* will calculate the next state at time $t+\Delta t$, and this new state is rendered on the screen by the *rendering part*. Since these steps are usually implemented as an infinite loop, the new state will become the current state in the next iteration[11].

From the view point of processing orders, it is a serialized loop, which is used in most graphics applications. It has a constraint that the simulation and rendering parts should be executed in an alternating sequence. Therefore, we can locally parallelize the internal codes of the simulation and rendering parts, while they cannot be simultaneously executed.

3.2 Its Parallelized Version

In the previous section, the serialized loop structure makes the rendering part output the new state at time $t+\Delta t$ on the screen. When it outputs the current state at time t instead of the newly calculated state, we can achieve more parallelization. As shown in Figure 2, using the current state, the simulation part calculates the next state, while the rendering part simultaneously displays the current state.

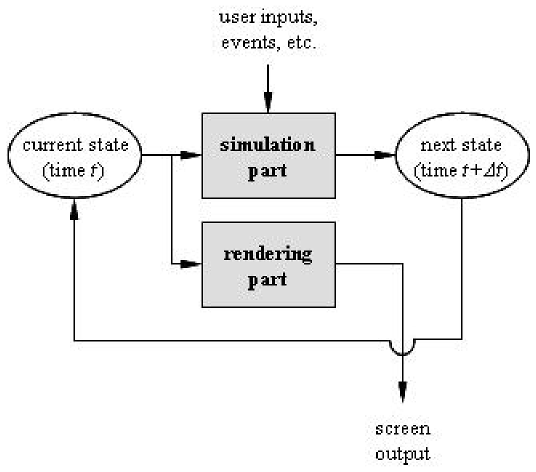

Figure 2: A parallelized version

In this case, we have a drawback of delayed displays, which makes us to see the graphics output delayed as time interval of Δt, in comparison to the previous version. In contrast, we can simultaneously execute both parts. In particular, when we have more than two CPU cores, the total execution time will be maximum value of the execution time for simulation part and rendering part, rather than the summation of them. Thus, we can speed up the overall process seriously, when we can well balance the simulation and rendering parts.

3.3 Applying Multithreading

Applying the multi-thread concepts to the parallelized version shown in the previous section, each part can be executed by a set of threads rather than a single process, as shown in Figure 3. This multi-threading feature even speeds up the overall process. For example, when the simulation part and rendering part are executed in two threads, respectively, the overall running time will be the maximum of the running times of totally 4 threads, and will be even less than the running time of the non-threaded version.

3.4 Adaptive Control of Theads

Through mixing up all the simulation and rendering threads into a large thread pool as shown in Figure 4, we can adaptively control the number of threads for simulation and rendering. This is our final goal of fully-threaded programming model.

This adaptive control model have an assumption: The simulation and rendering part can be split into a set of symmetric threads, respectively. In this case, the total works by the simulation part will be firstly divided into a set of almost equal sizes and each thread

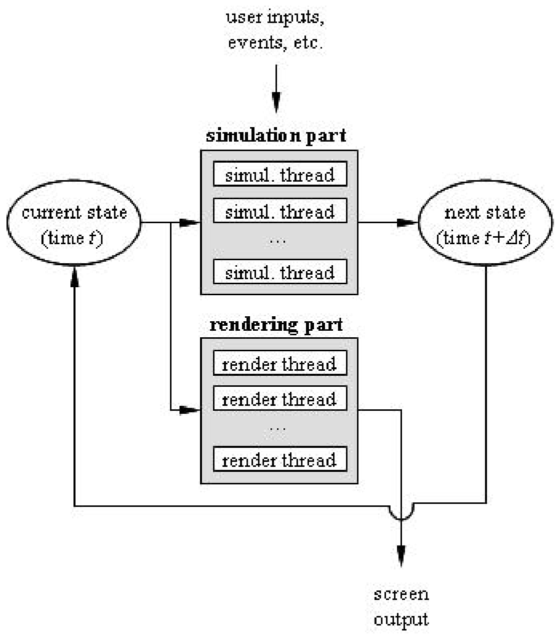

Figure 3: A multi-threaded version

does its own one in a parallelized manner. Similarly, the rendering part draws n objects using m threads, though assigning n/m objects to each thread, expecting all threads will be completed in almost same time.

Our adaptive control strategy will increase or decrease the number of threads for the simulation and rendering parts, according to the work load of them. For each iteration step, we measure CPU times consumed by each thread, to dynamically decide which

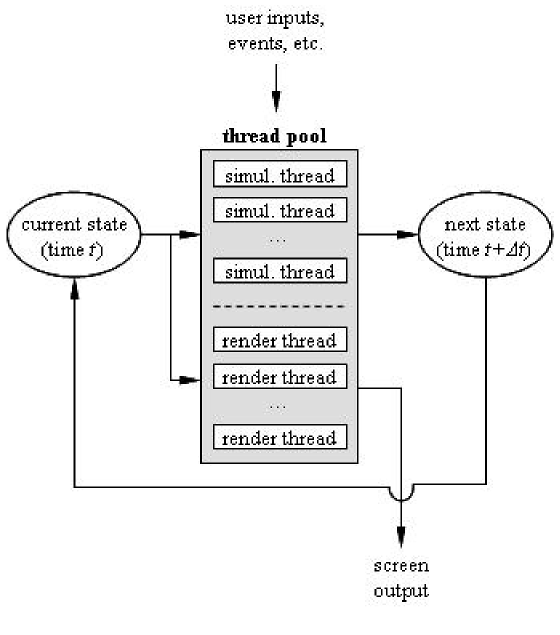

Figure 4: An adaptive control of threads version.

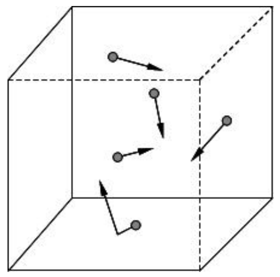
Figure 5: n particles in the box.

kind of thread should be increased or decreased. In the next section, we will introduce an example graphics problem, and build up an algorithm with our adaptive control method for the number of threads. The result will be followed.

4 Simulation Results

In this section, we will show an example implementation of our adaptive control method and its results.

4.1 An Example Simulation

To apply our adaptive control method, we need an example in which the simulation and rendering parts can be symmetrically divided into a set of threads. Additionally, the simulation part had better to have a reasonable time complexity to demonstrate our method.

In this paper, we use a simplified particle simulation. As shown in Figure 5, there are totally n particles in a rectangular box. Each particle constructs its own field, which is similar to the electromagnetic field, and has influences on all other particles[12].

Now, at a specific time t, we calculate the force from all other particles, and get its position at the time $t + \Delta t$. Since a particle will be influenced by remaining $(n-1)$ particles, all the process will be done in $O(n^2)$ time.

For the containing box, we need additional particle-to-plane collision detection and responses.

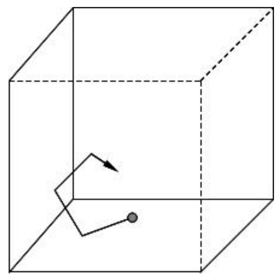
Figure 6: Multiple collisions with the bounding planes.

For the given time slice of Δt, a particle can collide the bounding plane several times, as shown in Figure 6. For simplicity, we adjusted the time slice Δt and particle velocities, and make a particle to collide bounding planes at most 3 times, in the worst case.

For each iteration, rendering threads will draw totally n particles, and thus its time complexity will be $O(n)$.

To control the number of threads adaptively, we will start from some measures for the CPU loads. Let $N^{\text{threads}}_{\text{simulation}}$ and $N^{\text{threads}}_{\text{rendering}}$ be the number of threads dedicated to simulation and rendering, respectively. The total number of possible threads $N^{\text{threads}}_{\text{total}}$ will be decided by the given hardware environments. For example, a dual-core Xeon processor with hyper-threading support can be regarded to have 4 physical CPU cores, and thus $N^{\text{threads}}_{\text{total}} = 4$.

Now we have the following simple constraints on the number of threads:

$$\begin{aligned} N^{\text{threads}}_{\text{simulation}} &\geq 1, \\ N^{\text{threads}}_{\text{rendering}} &\geq 1, \\ N^{\text{threads}}_{\text{total}} &\geq N^{\text{threads}}_{\text{simulation}} + N^{\text{threads}}_{\text{rendering}}. \end{aligned}$$

Let $T_{\text{simulation}}$ and $T_{\text{rendering}}$ be the total CPU time used by all the simulation threads and all the rendering threads, respectively. Now, the expected execution time for each simulation and rendering thread can be calculated as follows:

$$\begin{aligned} t_{\text{simulation}} &= \frac{T_{\text{simulation}}}{N^{\text{threads}}_{\text{simulation}}}, \\ t_{\text{rendering}} &= \frac{T_{\text{rendering}}}{N^{\text{threads}}_{\text{rendering}}}. \end{aligned}$$

Finally, the expected overall execution time will be the maximum of $t_{\text{simulation}}$ and $t_{\text{rendering}}$:

$$\begin{aligned} t_{\text{total}} &= \max\{t_{\text{simulation}}, t_{\text{rendering}}\} \\ &= \max\left\{\frac{T_{\text{simulation}}}{N^{\text{threads}}_{\text{simulation}}}, \frac{T_{\text{rendering}}}{N^{\text{threads}}_{\text{rendering}}}\right\}. \end{aligned}$$

For each iteration step, we measure $T_{\text{simulation}}$ and $T_{\text{rendering}}$, and calculate the optimal values for $N^{\text{threads}}_{\text{simulation}}$ and $N^{\text{threads}}_{\text{rendering}}$ to minimize t_{total}. Thus, we adaptively control the number of threads, based on the values of $T_{\text{simulation}}$ and $T_{\text{rendering}}$.

4.2 Results

To verify our idea, the adaptive control method in Section 4.1 was implemented as shown in Figure 7. Red and green spots are particles rendered by two independent rendering threads, and white lines indicate the bounding box.

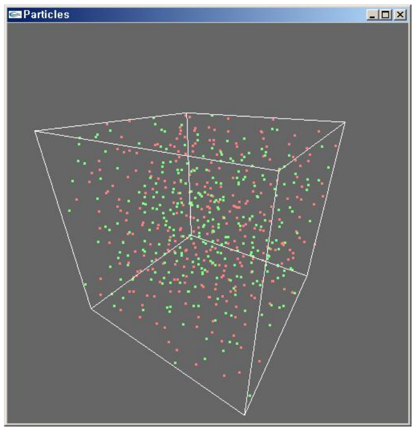

Figure 7: Our implementation program.

We executed this program on a system with a single 3.0GHz Pentium 4 CPU with hyper-threading support. Although this CPU executes at most 2 physical threads at a time, we measure the actual CPU time rather than the running time of threads, and thus, without loss of generality, we can simulate our adaptive control method on this system.

Table 1 shows the simulation results for 900 and 3,000 particle cases. Letting $N_{\text{total}}^{\text{threads}} = 4$, the number of threads are changed to measure $t_{\text{simulation}}$, $t_{\text{rendering}}$ and t_{total}. All the time values are averaged over the 100 time slices.

To demonstrate the adaptive control method, we need to change the portion of simulation and rendering times during its execution. As an example, we start from 900 particles for 25% of its running time, and then change the number of particles to be 3,000 for following 50% of its running time and finally 900 particles again for remaining 25% of its running time.

Table 1: $n = 900$ and $n = 3,000$ cases.

(unit: sec)

n	$N_{\text{simulation}}^{\text{threads}}$ / $N_{\text{rendering}}^{\text{threads}}$	$T_{\text{simulation}}$ / $T_{\text{rendering}}$	$t_{\text{simulation}}$ / $t_{\text{rendering}}$	measured t_{total}
900	1	0.071	0.071	0.042
	3	0.206	0.069	
	2	0.081	0.041	0.051
	2	0.042	0.021	
	3	0.077	0.026	0.066
	1	0.066	0.066	
3,000	1	0.586	0.586	0.586
	3	0.237	0.079	
	2	0.890	0.445	0.451
	2	0.119	0.060	
	3	0.890	0.297	0.304
	1	0.057	0.057	

Table 2: Various number of particles case.

(unit: sec)

n	$N_{\text{simulation}}^{\text{threads}}$ / $N_{\text{rendering}}^{\text{threads}}$	$T_{\text{simulation}}$ / $T_{\text{rendering}}$	$t_{\text{simulation}}$ / $t_{\text{rendering}}$	measured t_{total}
900 $\rightarrow$ 3,000 $\rightarrow$ 900	2	0.487	0.244	0.265
	2	0.124	0.062	
	3	0.481	0.244	0.188
	1	0.064	0.244	
	varying	0.472	N/A	0.186
		0.087		

Our implementation shows that the number of simulation threads are changed from 2 to 3 and finally to 2 again. This adaptive control result is compared with fixed number of threads cases. As shown in Table 2, the adaptive control case is better than fixed number of thread cases.

4.3 Conclusion

To fully utilize today's multi-core CPU's, we present a parallelized programming model. From the typical simulation-rendering loop in computer graphics programs, we tried to parallelize its loop structure and applied multi-threading features to it. Our final programming model is based on the pool of asymmetric threads. In other words, a set of threads are used to different tasks.

To realize our adaptive control scheme, we select a sample problem of n-particle simulation. After representing its parallelized version, we present an adaptive control scheme for the number of simulation and rendering threads. These number of threads are increased or decreased according to the measured execution time of them.

Our adaptive scheme is designed to maintain the optimal performance through controlling the number of threads dedicated to different tasks. Simulation results show that this scheme is better than the fixed number of threads cases.

To verify our adaptive control scheme more seriously, we plan to perform various experiments for additional problems. Executions on multi-processor systems are also required. Using two dual-core hyper-threading CPU's, we can use at most 8 physical threads, and we need more experiments on these advanced systems. Improvements on the adaptation algorithms are also required.

References:

[1] Intel, *Software Insight*, July 2005 Issue, 2005.

[2] A. Watt and F. Policarpo, *3D Games: Real-time Rendering and Software Technology*, Addison-Wesley, 2001.

[3] A. Rollings and D. Morris, *Game Architecture and Design*, Coriolis, 1999.

[4] http://www.intel.com/

[5] http://www.nvidia.com/

[6] http://www.ageia.com/

[7] T. Mattson and R. Eigenmann, *OpenMP: an API for Writing Portable SMP Application Software*, 2004.

[8] OpenMP Web site, *OpenMP tutorial on Super-Computing '99*, 1999.

[9] S. Bilas, "The Continuous World of Dungeon Siege", *Game Developers Conference '02*, 2002.

[10] http://www.unrealtechnology.com/

[11] D. M. Bourg, *Physics for Game Developers*, O'Reilly, 2001.

[12] D. Baraff and A. Witkin, *Physically Based Modeling*, SIGGRAPH'03 Course Note #12, 2003.