Real-World Case Studies: Threading Games for High Performance on Intel® Processors

by Sara Sarmiento, Software Solutions Group, Intel Corporation

February 24, 2005
The information contained in this document is provided for informational purposes only and represents the current view of Intel Corporation on the date of publication. Intel makes no commitment to update the information contained in this document, and Intel reserves the right to make changes at any time, without notice.

DISCLAIMER. THIS DOCUMENT AND ALL INFORMATION CONTAINED HEREIN IS PROVIDED AS IS. INTEL MAKES NO REPRESENTATIONS OF ANY KIND WITH RESPECT TO PRODUCTS REFERENCED HEREIN, WHETHER SUCH PRODUCTS ARE THOSE OF INTEL OR THIRD PARTIES. INTEL EXPRESSLY DISCLAIMS ANY AND ALL WARRANTIES, IMPLIED OR EXPRESS, INCLUDING WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, AND ANY WARRANTY ARISING OUT OF THE INFORMATION CONTAINED HEREIN, INCLUDING WITHOUT LIMITATION, ANY PRODUCTS, SPECIFICATIONS, OR OTHER MATERIALS REFERENCED HEREIN. INTEL DOES NOT WARRANT THAT THIS DOCUMENT OR THE INFORMATION CONTAINED HEREIN IS FREE FROM ERRORS, OR THAT ANY PRODUCTS OR OTHER TECHNOLOGY DEVELOPED IN CONFORMANCE WITH THIS DOCUMENT WILL PERFORM IN THE INTENDED MANNER, OR WILL BE FREE FROM INFRINGEMENT OF THIRD PARTY PROPRIETARY RIGHTS, AND INTEL DISCLAIMS ALL LIABILITY THEREFOR.

INTEL DOES NOT WARRANT THAT ANY PRODUCT REFERENCED HEREIN OR ANY PRODUCT OR TECHNOLOGY DEVELOPED IN RELIANCE UPON THIS DOCUMENT, IN WHOLE OR IN PART, WILL BE SUFFICIENT, ACCURATE, RELIABLE, COMPLETE, FREE FROM DEFECTS OR SAFE FOR ITS INTENDED PURPOSE, AND HEREBY DISCLAIMS ALL LIABILITIES THEREFOR. ANY PERSON MAKING, USING OR SELLING SUCH PRODUCT OR TECHNOLOGY DOES SO AT HIS OR HER OWN RISK.

Licenses may be required. Intel and others may have patents or pending patent applications, trademarks, copyrights or other intellectual proprietary rights covering subject matter contained or described in this document. No license, express, implied, by estoppels or otherwise, to any intellectual property rights of Intel or any other party is granted herein. It is your responsibility to seek licenses for such intellectual property rights from Intel and others where appropriate.

Limited License Grant. Intel hereby grants you a limited copyright license to copy this document for your use and internal distribution only. You may not distribute this document externally, in whole or in part, to any other person or entity.

LIMITED LIABILITY. IN NO EVENT SHALL INTEL HAVE ANY LIABILITY TO YOU OR TO ANY OTHER THIRD PARTY, FOR ANY LOST PROFITS, LOST DATA, LOSS OF USE OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, OR FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF YOUR USE OF THIS DOCUMENT OR RELIANCE UPON THE INFORMATION CONTAINED HEREIN, UNDER ANY CAUSE OF ACTION OR THEORY OF LIABILITY, AND IRRESPECTIVE OF WHETHER INTEL HAS ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING THE FAILURE OF THE ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

Intel, the Intel logo, Pentium, Intel Xeon, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2005 Intel Corporation
Introduction

The evolution of the multi-threaded processor design is the trend for next generation desktop processors. The introduction of Hyper-Threading Technology (HT Technology), where two logical processors simulate two physical processors and share the underlying hardware resources, is an indication that the processor is moving toward this multi-core model. This model will continue to evolve to support not only duplicated logical processors, but multiple physical processors in future generations.

Why should you thread your game? Parallelism is the easiest and most cost effective way to achieve scaling on multiple platforms. There are inherent performance benefits to threading a software application when the processor can support more than one thread. On a uniprocessor system, it is difficult to see the performance benefits of threading because the process or swapping threads in and out can be far too expensive. However, threading on a processor that support multiple threads, such as the ones that are currently available on the market today, offer benefits for threaded applications. By threading the code base it allows an application to take better advantage of under utilized hardware resources. Threading also introduces an application’s ability to scale on the next generation processors that have multiple cores, and can also support threaded applications. As the hardware evolves and continually improves so will the performance of threaded software.

Preamble to Threading

There are two different types of parallelism, domain decomposition (data level parallelism), and functional decomposition (task level parallelism). Data level parallelism is defined by performing the same independent operation on different sets of data. Task level parallelism is independent work being mapped to different threads to execute asynchronously. This paper will go into these methods in detail, and give examples of game engines and titles that have seen significant performance gains by introducing threading into their application.

Data Level Parallelism

Picture the drive-thru window at a local fast food establishment. Now, imagine the line of cars waiting to place their order, pay the cashier, and get their meal. If everything was done with just one order window and through one line of cars, it would take longer than if you had two windows and could divide the line of cars between the two. The throughput would increase close to 2X. The same principle can be applied to data processing through data level parallelism.
Data can be divided into \( \Pi \) number of partitions, depending on the number of threads a processor can handle, to help increase the instruction throughput. There are two different methods in distributing data to threads, static and dynamic distribution. In static distribution, iterations are divided into chunks of size \( \text{chunk}_\text{size} \), and the chunks are statically assigned to threads in the team in a round-robin fashion in the order of the thread number. Note that the last chunk to be assigned may have a smaller number of iterations. When no \( \text{chunk}_\text{size} \) is specified, the iteration space is divided into chunks, which are approximately equal in size and each thread is assigned at the most one chunk. In dynamic distribution, iterations are divided into chunks of size \( \text{chunk}_\text{size} \), and each chunk is assigned when a thread requests for it. The thread executes the chunk of iterations, then requests for another chunk, until no chunks remain to be assigned. Note that the last chunk to be assigned may have a smaller number of iterations. (OpenMP* Specifications). Static distribution has very low CPU overheads while dynamic distribution offers potentially better load balancing of the work at the expense of a slightly higher CPU overhead. Below are two case studies of game engines that saw significant performance gains by implementing data level parallelism in their solution.

**Case Study: Codemasters/SixByNine**

Colin McRae Rally 4*, originally an Xbox* title, is an off road driving simulation, ported to the PC. The 3D engine was designed to support multiple platforms which made it difficult to change the design of the game at an architectural level. The goal for the developers was to find a specific part of the engine that could benefit from data level parallelism without affecting the surrounding code. The section identified for threading was the weather system computations in order to increase the amount of particles generated by the 3D engine.
The weather system consisted of two loops that benefited from data level parallelism. The first loop that calculates new positions for each frame and a second loop used to determine the collision of the particles generated by the engine with the surrounding terrain. In order to implement threading within these loops the developers of this title removed all the global variables from within the particle generation loops, and instead created local variables in order to avoid race conditions with the data being used. By using the high performance setting available with the Intel® Compiler, the loop received a 5% performance increase in speed. The data level parallelism within the particle system added an additional 7-8% increase in speed to the weather system. The developer also used the Intel® Thread Profiler to check load imbalance and any synchronization overhead within the threading implementation.

Using the speed gained with the optimizations and the higher clock speeds available on a HT Technology based PC, the developer was able to increase in areas affected by particles by 4x, and increase in particle density but 8x compared to the initial Xbox version of the game. This created a significantly better visual affect within game play.
**Task Level Parallelism**

Imagine that same drive-thru window example that was used earlier, but instead of creating two lines of cars, creating two different windows. Each window is responsible for a different job in the fast food drive-thru process. The first window is responsible for taking the order at the beginning and taking payments while the second window is responsible for distributing the order to the customer.

In this case, the tasks are divided into N number of threads. Tasks should be relatively independent to avoid synchronization overhead. Generally, performance benefits are determined by data dependencies and the load balance between threads. If a threaded application is in the situation where one thread is doing significantly more work than another thread, that application will see little performance gain. It is dependent on the software designer and developer to understand the way tasks are handled in the application and allocate those tasks evenly among threads. Below are a few examples of game applications that see performance benefit on processors that support multiple threads that implemented task level parallelism in their solution.

**Case Study: Lego/Argonaut Bionicle**

Bionicle is a third person action game produced by Lego, developed by Argonaut Software, and based off to Lego’s toy franchise with the same name.
Bioncle tasks were run on different threads. The main thread was for Update World and rendering. A worker thread was spawned to handle Update Sky. The sky could have multiple cloud layers (typically two, but potentially up to four). For each layer there was one thread as shown in the figure below.
The interface to the main game was simple because they only needed to issue messages to the Update Sky Module to update graphics and textures for rendering. The resource loader continually streamed data based on commands from the main game thread.

The advantage for this game in terms of threading performance was that it created a common code base among different operating systems, and was able to reduce the amount of code complexity. In addition, by including the resource loader and decompression on a separate thread there is some additional performance gained by multi-threading blocked I/O. The key is that they use normal blocking IO OS calls, but on a secondary thread. Most operating systems support some kind of Overlapped IO or non-blocking IO, but the implementations and API’s are very different. In this case threads are used for file loading in order to provide a common asynchronous API for both Win9X* and Windows* XP. The disadvantage of this method is that it makes it difficult to abort File loading operations, which will have an impact on the program’s ability to switch streams at will.

**Case Study: Codemasters/1C/KD-Lab**

Perimeter* is an advanced science fiction real time strategy game. It has highly detailed and realistic landscape, buildings, and game play units where the terrain is constantly reforming and changing in real time to enhance the consumer’s game play experience.

This game uses a single thread, which it detects through CPUID when there is only one processor available. It uses two threads when there are two or more processors detected. In the single threaded model, as seen in the figure below, Update World and Rendering occur on the same thread. In the multi-processor case, Update World that in game logic and AI run on a separate thread to the rendering engine, which includes the rendering as well as the graphical user interface.
The threaded method required data dependencies between the threads to be handled and synchronized prior to updating graphical data. Logic within Update World needs to communicate changes in position to logical objects to the rendering engine so that these changes are streamed to the update buffer and then updated at the next time interval. All the relevant logic information also needs to be updated within the GUI, and at the same time player actions are batched and given to the logic unit within Update World to handle the changes.

As a result of threading, Update World and the Render World game play saw a 25-50% increase in speed in frame rate on a HT Technology based system. In addition to an increase in frame rate, an additional side effect of threading the rendering was that the frame rate became more uniform. This can be argued that while harder to qualify it is the most important factor when assessing the end users experience. In a single threaded system the frame rate would fluctuate when update world was called sequentially between rendering frames, in a multi-threaded design. This is no longer the case with frame rate largely independent of the time taken by the logic.

Synchronization between the logic and GUI is not easy, and dependencies could limit the amount of performance that can be achieved. Reducing the amount of synchronization overhead to as few as possible and ensuring that there is a balance between threads will ensure the best results. A clean modular design will help isolate the essential data needed to be synchronized between threads.

**Case Study: Virtools Dev 2.1**

Virtools Dev 2.1 is a 3D authoring tool used in the deployment of 3D engines and games. In this particular example, we focus on Update World, but on the particle system in particular.
In the figure below is the original design of the engine. Within Update World, it was determined that there is a significant portion of computations done on the Particle Systems prior to rendering the objects to the screen.

Task level parallelism was used to thread this application. On the main thread was the Update World computations, and a worker thread was spawned to compute the Particle System. While the Particle System was being computed, Update World could continue performing it’s rendering. Final rendering of time $t+1$ waits until the Particle system renders and the results are synchronized. It’s worth pointing out that under this system the rendering system, whether DirectX* or OpenGL*, does not have to be run in a multi-threaded mode. The primitive list for the particles is built and possibly even alpha sorted on the secondary thread but the drawing can still be done on the primary thread.
The caveat to utilizing this model would be if there was a load imbalance. If the computations on Update World out weighed the computations for the Particle Systems and vice versa, the engine would not see significant performance gains. However, for Virtools, the division of work done on each thread was relatively balanced, and it saw a significant performance gain on processors that support multiple threads. It is also true that you will get the most gain at exactly the time you need it most, that is, when you suddenly have a large particle effect triggered.

As shown in the figure below, the Virtools Dev 2.1 application saw a 13% performance gain on a HT Technology based system and a 33% performance gain on a dual processor system.

![Figure](image.png)

**Figure.** Dell Dual 2.2 GHz Xeon with Hyper-Threading; 512MB RAM; NVIDIA GeForce* 3 64MB; Processor Config BIOS switch-able (SP, HT, DP)

**Case Study: Dreams Interactive Lejendary Adventures**

Lejendary Adventures is a game created by Dreams Interactive. By design, this application was threaded and scaled well on processors that supported multiple threads. In this game, Update World was broken up into two threads, the first to handle the rendering of Update World and the other to handle DirectX 3D rendering.
Working with Intel, they were able to see additional performance gain by creating a third thread to handle Terrain and Procedural Textures as a separate task from the Update World threads.

By doing this the game title was able to see additional performance gains on a HT Technology enabled processor and Dual processor system. For the HT Technology enabled processor the game saw a 19% performance gain, and for the dual processor system a 24% gain.
Case Study: SCI / Pivotal Conflict Vietnam*

Conflict Vietnam is a Frames Per Second (FPS) game that takes place in the jungle terrain of Vietnam. The game consists of varying missions that take place in numerous different graphic environments, such as dense jungle and river assaults. This game incorporates cutting edge graphics techniques, such as blooming and depth of field, using DirectX 9.

In the original game design the main thread performs the logic and the rendering while a separate thread performs mixing of compressed audio. Synchronization overheads between the two threads are kept to a minimum as the commands issued by the primary thread to the audio thread are kept light weight and handled using low overhead Critical Sections. On PC’s that report multi-processors the rendering of a procedural sky textures is implemented on a separate worker thread, this replaces a static texture used on single processor machines.

Figure. Dell Dual 2.2 GHz Xeon with Hyper-Threading; 512MB RAM; NVIDIA GeForce 3 64MB; Processor Config BIOS switch-able (SP, HT, DP)
The developers for this game used the Intel Thread Profiler to identify areas in the threading implementation which could be fine tuned. In the case of the procedural sky, Thread Profiler showed that main thread was being blocked by the update sky thread when the sky texture was of a high resolution or the update world calculations were particularly light weight. To further improve the threading performance, the solution that was determined was to reduce the amount blocking time spent in the main thread due to the sky texture computations by threading the update sky routine against more of the calculations done in the main thread. Rather than thread the update sky against just the game logic the sky was changed to thread in parallel with both the render world and the game logic being called in update world.
This change in the threading implementation increased the performance by 15% bringing the total performance win compared to implementing the procedural sky in a sequential manner to 25% on an HT Technology enabled system. In addition, the change to a procedural sky compared to the initial static texture implementation allowed for greater control over the look and feel of the sky.

**Conclusion**

Implementing threading in games is not an easy task. There are several factors that play into determining if threading is feasible both from a technical and business standpoint. On an existing application, implementing threading generally requires a high level design change. Incorporating that understanding when planning out future releases will help to take advantage of next generation processors. Keep in mind that the code base itself has to be flexible and modular to allow for threading to be introduced. Game engines that are sequential by design and those that contain multiple global variables and data dependencies between modules are difficult to thread.

Support for multi-threading exists today and will continue in future generations of Intel® desktop processors. It is important that you understand the underlying architecture and how it handles instructions in order to take advantage of the multi-processor design. Working with Intel and researching Intel’s technical software support sites to find tips and techniques to take advantage of the multi-threading capabilities of the Intel desktop processors is one way to do this.
Additional References


About the Author

Sara Sarmiento is a Technical Marketing Engineer in the Software Solutions Group at Intel Corporation. A graduate from Oregon State University, Sara holds a Bachelors of Science degree in Computer Science. She joined Intel in October 2000, and is currently working on software enabling for current and future generation client desktop processors.