Parallel Rendering Mechanism for Graphics Programming on Multicore Processors

Satyadhyan Chickerur, Department of Information Science and Engineering, B. V. Bhoomaraddi College of Engineering and Technology, Hubli, Karnataka, India
Shobhit Dalal, HealthAsyst Pvt. Ltd, Bangalore, Karnataka, India
Supreeth Sharma, Akamai Technologies, Bangalore, Karnataka, India

ABSTRACT

The present day technological advancement has resulted in multiple core processors coming into desktops, handhelds, servers and workstations. This is because the present day applications and users demand huge computing power and interactivity. Both of these reasons have resulted in a total design shift in the way the processors are designed and developed. However the change in the hardware has not been accompanied with the change in the way the software has to be written for these multicore processors. In this paper, we intend to provide the integration of OpenGL programs on a platform which supports multicore processors. The paper would result in clear understanding how graphics pipelines can be implemented on multi-core processors to achieve higher computational speeds up with highest thread granularity. The impacts of using too much parallelism are also discussed. An OpenMP API for the thread scheduling of parallel task is discussed in this paper. The tool Intel VTune Performance Analyzer is used to find the hotspots and for software optimization. Comparing both the serial and parallel execution of graphics code shows encouraging results and it has been found that the increase in frame rate has resulted due to parallel programming techniques.

Keywords: Multicore, Open Graphics Library (OpenGL), Open Multi-Processing (OpenMP), Parallel Programming, Parallel Rendering

INTRODUCTION

Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently (“in parallel”). There are different forms of parallel computing: bit-level, instruction level, data, and task parallelism (Culler, Singh, & Gupta, 1998). Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to the physical constraints preventing frequency scaling for improving performance of microprocessors. As power consumption (and consequent heat gen-
operation) by computers has become a concern in recent years, parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multicore processors.

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism - with multi-core and multi-processor computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task. Specialized parallel computer architectures are sometimes used alongside traditional processors, for accelerating specific tasks. Parallel computer programs are more difficult to write than sequential ones, because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common. Communication and synchronization between the different subtasks are typically one of the greatest obstacles to getting good parallel program performance. The maximum possible speedup of a program as a result of parallelization is observed as Amdahl’s law.

OpenMP is very successful in reducing the time required for the computation of various mathematical expressions. It was developed in the year 1997 and has been growing since then. Various versions have been developed and the last release was Version 3.1 in July 2011. OpenMP has a very good interface to develop parallel programs as the applications are getting larger and more complex. It has basically provided shared memory multiprocessing and multi-threading in commonly used programming languages which include C, C++ as provided in the documentation (Developing Multi threaded Applications, 2005). The programs developed can be scaled to a desktop or a supercomputer. OpenMP uses pragmas and work sharing constructs to divide the workload of a thread into multiple sub-threads so that more threads work on a single task than a single thread trying to do the entire work.

Parallelism can be done at two levels. One is process level parallelism in which many processes run simultaneously. A typical example of process level parallelism is running many applications together on an operating system. You can run media player to listen to music, edit files in a word document, copy data to and from a CD to the hard disk location simultaneously. The second is thread level parallelism which means that several threads run concurrently. For example, within a document several options like editing text, cut, paste, and preview options are handled by different thread in parallel. We use thread level parallelism rather than process level parallelism because of various reasons which are discussed in the Table 1.

From Table 1 we can infer that it’s beneficial to use thread level parallelism. OpenGL enables developers of software for PC, workstation, and supercomputing hardware to create high-performance, visually compelling graphics software applications, in markets such as CAD, content creation, energy, entertainment, game development, manufacturing, medical, and virtual reality. OpenGL exposes all the features of the latest graphics hardware. OpenGL graphics programming is highly used by game developers. Now-a-days many gaming industries use OpenGL to program their gaming packages. OpenGL was developed in the year 1992 and

<table>
<thead>
<tr>
<th>PROCESS</th>
<th>THREAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Different address space is available for each process.</td>
<td>Same address space is allocated for different threads.</td>
</tr>
<tr>
<td>Processes are heavy weight.</td>
<td>Threads are light weighted than process.</td>
</tr>
<tr>
<td>They have more overhead.</td>
<td>They have less overhead than processes.</td>
</tr>
<tr>
<td>Context switching is higher.</td>
<td>Context switching is lower.</td>
</tr>
<tr>
<td>Inter-process communication is high.</td>
<td>Inter-thread communication is low.</td>
</tr>
</tbody>
</table>
Fault Tolerance
Valentin Cristea, Ciprian Dobre, Corina Stratan and Florin Pop (2010). Large-Scale Distributed Computing and Applications: Models and Trends (pp. 168-193). www.igi-global.com/chapter/fault-tolerance/43107?camid=4v1a

Error Recovery for SLA-Based Workflows Within the Business Grid