

<http://www.hpcwire.com>[\(javascript:void\(printArticle\(\)\);\)](#)[Click to Print](#)[SAVE THIS \(javascript:void\(open\('http://www.savethis.clickability.com/s/saveThis?partnerID=317787&urlID=28597308&origin=11','click','height=450,width=510,title=no.location=no.scrol'libars=yes,r](#)[\(javascript:void\(printArticle\(\)\);\)](#)

September 28, 2007

Programming Models for Scalable Multicore Programming

by Michael D. McCool, Chief Scientist, RapidMind

Multicore devices will quickly evolve in both architecture and core count. This will motivate software developers to decouple the code from the hardware, in order to enable applications to move between different architectures and automatically scale as new processor generations are introduced. An appropriate programming model can enable this decoupling while maintaining -- and even enhancing -- performance.

Moore's Law is a statement about transistor density increasing over time. It has become harder and harder to squeeze extra performance out of a single core by using more transistors, and the fact that power consumption increases rapidly and nonlinearly with clock rate blocks further increases in performance by scaling to higher gigahertz ratings. Therefore, all major processor vendors have now switched to an explicitly parallel, multicore processor strategy. By combining multiple small, efficient cores onto a single chip, it is possible to get much higher overall performance and simultaneously improve power efficiency.

Unfortunately, only parallelized applications can exploit this additional performance. In fact, since the individual cores on a processor are often slower than the large single-core processors of the past, non-parallelized applications may in fact be slower on multicore processors. Also, since the number of cores will grow exponentially over time (under the new interpretation of Moore's Law), any application, in order to grow in performance, must be written to use any number of cores in a scalable fashion.

Autoparallelization tools are unlikely to help. Modern processors already exploit internally much of the implicit parallelism in an application, in the form of low-level instruction level parallelism (ILP). It has been shown that most applications have relatively small amounts of such implicit parallelism, and that this is already nearly fully utilized by modern processors.

However, there are further complications. The memory system is actually the chief bottleneck in many applications. In order to take advantage of the increased computational performance of a processor, the data must be moved onto the chip and off again as efficiently as possible. If the data rate cannot keep pace with the computational performance, than any increase in on-chip computational performance is useless.

In a multicore processor, all cores on a processor must share a finite off-chip bandwidth, making memory access even more of a bottleneck. Also, accessing main memory from the processor, for data that is not in cache, can take hundreds of processor clock cycles to complete. This latency can severely degrade performance since in the worst case the processor must stall while waiting for the memory access to complete.

There is a solution to this: even more parallelism! If the processor has extra, independent work to do while waiting for long-latency operations to complete, then it can run more efficiently. Single-core simultaneous multithreading, also called hyperthreading, is really a mechanism to hide latency. By having multiple concurrent tasks on a single core, it is possible to switch from one to another when one task encounters a long-latency operation, such as a memory access.

Little's Law states that for efficient execution, the number of concurrent tasks "in flight" at any point in time should be equal to the latency times the parallelism. A modern four-core processor with the ability to issue four floating-point operations (using SSE instructions or some other form of instruction-level-parallelism) at once has a total parallelism of 16, since it can issue 16 operations per clock. Suppose in general that we access main memory for every 8 numerical operations, which is an optimistic value. With a main memory latency of 128 cycles -- again optimistic -- we need 256 separate, independent tasks in order to fully utilize the processor.

In other words, multicore processing is only exacerbating an already challenging problem. Most software today is grossly inefficient, because it is not written with sufficient parallelism in mind. Breaking up an application into a few tasks is not a long-term solution. First, lots and lots of parallelism is actually needed for efficient execution: much more than the number of cores, actually. Second, with the number of cores increasing exponentially, more and more parallelism will be needed over time.

The solution to this dilemma is data parallelism. In data parallelism, the structure of the data is used to drive the creation of more and more parallel tasks as needed. Since larger problems with more data naturally result in more parallel tasks, a data-parallel approach results in a scalable solution that can automatically take advantage of more and more cores. Data parallel programming models, since they also focus on the data and its movement, also result in predictable memory access patterns and this can also be used to improve the efficiency of memory access.

There is some concern about the general applicability of data parallelism, but it is important to understand that there are a variety of data parallel programming models available. Some of the simplest forms can only be used on very regular problems, but the most flexible models are capable of dealing with a variety of different kinds of irregularities, and are equivalent in expressive power to task parallelism.

The SIMD (Single Instruction, Multiple Data) model is the simplest data parallel model but is also the most limited. In this model, a sequence of operations is applied in parallel to all the elements of a collection of data, such as an array or set. A naive implementation of this model is not very efficient on modern memory-constrained architectures, since it reads and writes memory for each simple operation. In practice, several operations should be combined together into a more arithmetically intense kernel, and the kernel should be unrolled and vectorized to exploit instruction-level parallelism as well as multicore parallelism.

Even so, the basic SIMD model cannot handle irregularities of control or data access. It cannot, for instance, avoid doing unnecessary work, since every element of the data collection will have exactly the same sequence of operations applied. If there are special cases, such as boundary conditions, that require more or less work, the SIMD model cannot take advantage of this fact, and has to do the worst-case computation all the time. This can degrade performance unacceptably.

The SPMD (Single Program, Multiple Data) model is better. In this model, every kernel can also include control flow, which allows a kernel to do more or less work, as the situation demands. This type of model can handle irregularity in workload, but also requires a more sophisticated runtime that can move heavier parts of the workload to more lightly loaded cores.

Collective operations can be added to this model to support irregularity in communication and data access. A general scatter/gather collective operation can handle any irregular memory access pattern, but as a parallel operation. The combination of an SPMD model for kernels with scatter/gather is equivalent in computational power to

threading, but is more structured and more naturally leads to the massive parallelism required for performance. It also has certain advantages in safety; for example, it is impossible to express programs with deadlock in this model. Certain refinements to this model are possible, for example it can be extended to nested or recursive parallelism, but even without these refinements it is applicable to a wide range of applications, as has been shown by extensive research over the last twenty years.

Finally, we come to the problem of programming mechanisms. Efficient implementation of a program on a parallel computer requires the coordinated generation of low-level machine language to exploit instruction level parallelism as well as a high-level runtime to support such operations as load balancing. This is because, as pointed out above, processors actually support multiple parallelism mechanism over a range of scales, and exploiting them all simultaneously is crucial for performance.

Many parallel programming languages have been devised, but these systems are not in wide use, and most existing application code is written using languages such as C and C++ that do not natively support parallelism. Tuned parallel libraries can be used, but these are only suitable for the most common stereotypical tasks, and even library writers need something to program with. Frameworks can also be used, but these typically will only address one level of parallelism at once, such as multiple cores, and cannot coordinate code generation with the runtime, since one is unaware of the other.

In developing the RapidMind platform, we have taken a different approach. We start with a programming model, SPMD data-parallelism, that we know is both general and efficient. We provide access to this from existing C++ compilers, but without using the native C++ code generator: instead we provide our own, so that the generated code can be coordinated with the runtime. This allows us to exploit multiple granularities and mechanisms for parallelism simultaneously. The simplicity of the SPMD model means that the programmer can continue to work with familiar concepts, like functions and arrays, but can also directly express parallel algorithms in a natural and efficient way. We have also been able to map this programming model to widely divergent architectures with excellent performance, including GPUs, the Cell BE, and of course multicore CPUs. This portability is useful today but is also crucial in order to future-proof application code against likely changes in processor architectures. Our system takes a high-level abstraction of parallelism and maps it to what is available, and can do so efficiently.

In summary, efficient performance on modern multicore processors requires an aggressive approach to parallelism. There are many performance mechanisms in modern processors, including but not limited to multiple cores, that depend on parallelism. In addition, memory bandwidth and latency can severely degrade performance if not managed, but "spare" parallelism can be used to hide latency. Data-parallel programming models can be used to express the required level of parallelism but also to expose coherent memory access patterns, which can be used to optimize memory bandwidth. A sufficiently general data-parallel computational model, such as the SPMD model, is as powerful as a task-parallel programming model, so no generality is lost in using this more efficient and scalable approach. Finally, the fact that this model is capable of providing portability means that application logic can be decoupled from hardware deployment, providing more choices to the software developer and providing a measure of security that their code will continue to perform well on future massively multicore processors.

About the Author

Michael McCool is an Associate Professor at the University of Waterloo and co-founder of RapidMind. He continues to perform research within the Computer Graphics Lab at the University of Waterloo. Professor McCool has a diverse set of published papers, and his research interests include high-quality real-time rendering, global and local illumination, hardware algorithms, parallel computing, reconfigurable computing, interval and Monte Carlo methods and applications, end-user programming and metaprogramming, image and signal processing, and sampling. Michael has degrees in Computer Engineering and Computer Science.

[http://markets.hpcwire.com/taborcomm?Account=hpcwire&Page=QUOTE&Ticker=\\$2](http://markets.hpcwire.com/taborcomm?Account=hpcwire&Page=QUOTE&Ticker=$2)


[http://markets.hpcwire.com/taborcomm?Account=hpcwire&Page=QUOTE&Ticker=\\$2](http://markets.hpcwire.com/taborcomm?Account=hpcwire&Page=QUOTE&Ticker=$2)

[http://markets.hpcwire.com/taborcomm?Account=hpcwire&Page=QUOTE&Ticker=\\$2](http://markets.hpcwire.com/taborcomm?Account=hpcwire&Page=QUOTE&Ticker=$2)

[http://markets.hpcwire.com/taborcomm?Account=hpcwire&Page=QUOTE&Ticker=\\$2](http://markets.hpcwire.com/taborcomm?Account=hpcwire&Page=QUOTE&Ticker=$2)

Find this article at:

<http://www.hpcwire.com/topic/deprecated/17902939.html>

 Click to Print

[SAVE THIS \(javascript:void\(open\('http://www.savethis.clickability.com/st/saveThis?partnerID=317787&urlID=28597308&origin=11','click','height=450,width=510,title=no.location=no.scrol lbars=yes,r](#)

[\(javascript:void\(printArticle\(\)\)\);](#)

Check the box to include the list of links referenced in the article.