Transitioning to Multicore Development

Part 2: Optimizing Parallel Software

A Whitepaper by Rogue Wave Software

Rogue Wave Software
5500 Flatiron Parkway,
Suite 200
Boulder, CO 80301, USA
www.roguewave.com

Transitioning to Multicore Development

Part 2: Optimizing Parallel Software

by Rogue Wave Software

TABLE OF CONTENTS

Introduction

Multicore systems are ubiquitous; it's virtually impossible to buy even commodity computers without a dual, quad, or hex-core processor. It won't be long before many-core processors start to be prevalent as well. Each core in a multicore processor is capable of executing a program, so a quad-core processor can run four separate programs at the same time. That's great if you have many different programs you need to run at one time, but can become a problem when you need performance from a single program. Those four cores can also potentially run one program faster than a single core processor would, but only if the program is written correctly. If you run a sequential (or serial) program written for single core architectures on a multicore platform, it will generally only be able to leverage a single core. Serial programs don't run any faster, and may even run slightly slower, on multicore processors.

There are no silver bullets for automatically converting a sequential application written for a single-core machine to a parallel application that will run quickly on a multicore processor. Instead, developers, who need to get performance out of a multicore processor, need to shoulder the burden of figuring out how to map the work that their program does into multiple threads that can be executed safely in parallel across separate cores. The process of taking a serial application and making it parallel, called parallelization, involves significant redesign and optimization.

Part one of this whitepaper series, "Transitioning to Multicore Development, Part 1: Overcoming the Challenges of Developing Applications for Multicore Systems," outlines a process that shows you how to parallelize a serial application smoothly and correctly. It talks about identifying components to parallelize, task versus data parallelism, discusses race conditions, synchronization, and deadlocks. It emphasizes the importance of testing and troubleshooting parallel programs.

This paper, part two of the series, will focus on how to optimize application performance of a multithreaded program once it is operating correctly. The paper will review different kinds of performance challenges including too many or too few synchronization constructs, bandwidth and latency limits, and cache coherency problems.

Parallelization Workflow

There is a vigorous debate between proponents of designing in performance and those who argue that design should be based on extensibility and correctness and only subsequently focus on optimization within a running system. This series takes the idea that you are working with an already running serial program rather than designing from scratch. In this case, it is recommended that you perform one or more iterations of parallelization, with each iteration focusing on a computationally intense part of the program. You'll want to parallelize that section either in terms of task or data parallelism, focusing first on getting the correct answer and then on optimizing what you've done for speed. If overall performance does not yet meet the requirements after adding parallelization to one set of operations or across one kind of data, then it is necessary to perform a second iteration.

However, if you are designing an application from scratch, then it is recommended to include performance along with correctness and maintainability as a potential goal of the design. Nothing beats actually trying things out and measuring the impact though, and for that you need working code. So even if optimization and performance is a priority during the design, you are best served by planning in time during the later stages of development for focusing on measurement driven optimization.

Amdahl’s Law and Perfect Parallelization

If you are going to measure program performance and try to optimize it, the first thing to consider is what amount of performance should be expected from adding threads to your program. What is the ideal speedup? It is unlikely that you’ll get better than ideal speedup and there is little to be gained from continuing to optimize once the program gets close to that ideal.

It is easy to calculate the ideal speedup that can be gained from parallelization. The key formula is called Amdahl’s law and is expressed as S = 1/(1-P+P/C): S equals the speedup (parallel performance / serial performance); P equals the fraction of total runtime the serial program spent in the section of code that has been parallelized; and C equals the number of threads across which the work has been parallelized.

The key factor in this relationship is the fraction of the program’s work that is being spread across threads. If the entire program can be parallelized and the parallelization is being done over four threads, then C = 4 and P = 1, and theoretically it is expected that the program will run four times as fast. Four threads and half the program is C = 4 and P = 0.5, which yields a program that only runs 1.6 times as fast. If only a small fraction of the program can be parallelized then the performance gains will be relatively modest.

Almost every program has some amount of serial logic that applies to the entire program, but really only needs to be executed once. Often this occurs at start-up, shutdown, and when the program is transitioning from one kind of calculation to another. This serial fraction places a theoretical limit on the speedup that can be obtained through parallelization.

Compiler Optimization

Optimization should ideally happen at several different levels in the program. Some optimizations are best done by the compiler and don’t require any code changes, others may require pragmas but no changes to the actual code, and others require you to break the problem down or re-structure synchronization. This paper focuses on optimization that needs to be done manually, however it is unwise to neglect the compiler’s capability to generate optimized binaries based on the code provided. Typically, a compiler will be able to perform certain optimizations, such as eliding unused variables and redundant calculations, reordering certain instructions to better fit the processors capabilities, and unrolling loops to allow for better pipelining. These optimizations can provide significant benefits. Most of these apply the same to multithreaded programs as they did to the original serial program. When measuring performance to determine how well you are doing compared to ideal scalability, it is good to measure the performance of the compiler-optimized parallel program against a compiler-optimized serial version of the same program.

Performance Challenges

Optimization involves identifying performance bottlenecks and re-designing the program to eliminate those bottlenecks. It is important to know what kinds of bottlenecks to look for and how they can be resolved. Such bottlenecks can involve concurrency, synchronization, memory bandwidth, or cache coherence. This section will discuss difficulties that may occur in these aspects of the program, followed by an examination of ways to make these challenges less daunting.

Concurrency

One of the challenges of tuning a multithreaded application is choosing the right number of threads. Threads are added in order to take advantage of multiple cores. In many cases the optimal number of threads is equal to the number of available cores. However, there will be times where the program runs better with fewer or more threads.

More threads provides a certain amount of future proofing (you can expect the number of cores to increase) and gives the operating system (OS) and hardware the opportunity to swap to another thread when any thread stalls waiting for a resource. However, there is overhead to creating a new thread, and each time that thread starts and stops there is a certain amount of context switching overhead.

Each thread needs to have enough work to keep the processor on which it is running fully occupied. Modern processors are almost always equipped with vector processing capabilities and pipelines, and will function optimally with a certain amount of work. Creating too many threads, or starting and stopping computations too frequently, can prevent the processor from being able to bring all of its functional units to bear and can leave a lot of performance on the table.

Ultimately, experimentation and investigation of resource utilization and time spent creating and context switching may be required to identify the optimal number of threads.

Load Balancing

During the discussion about Amdahl's law, the impact that serial sections of the code have on limiting scalability was mentioned. The same limitation can set in inadvertently when work is partitioned unevenly across multiple threads of execution. Having an uneven workload may cause one or more threads to stall. This effectively means there are parts of the program where the level of concurrency is reduced. In the worst-case scenario, one thread may have more work to do than all the others. Once the others complete their tasks, they will all idle while that one thread is working. That section of the run-time is effectively no longer parallel.

When some threads are given a task that will take longer to perform than tasks on other threads, care should be taken to ensure the threads completing first can either begin working on some other task, or that there are other threads waiting that the OS can schedule into that gap.

Synchronization

Synchronization, which was introduced in the first part of this paper, Part 1: Overcoming the Challenges of Developing Applications for Multicore Systems, is absolutely critical for ensuring parallel programs execute correctly, but it invariably adds additional complexity and computation to any program.

A challenge exists beyond making sure a program is correctly synchronizing around shared data. You must also choose the right granularity for the amount of data covered by a given mutex object. If you utilize very fine-grain mutex objects (many mutex objects for the same data set) then it is possible to potentially maximize the number of things that happen at the same time, but there is an overhead to creating and using mutex objects. At the other extreme, it is possible to use as few as one mutex, and simply require that any thread has it before a read or write occurs to any shared data. The risk here is that it may frequently require threads to wait unnecessarily, when the data being changed isn't the same data they wanted to access.

The rule of thumb here is to pick a middle value, tending towards fewer mutex objects when possible. If there are multiple kinds of data that will all need to be changed at the same time, they should be guarded by a single mutex. For example - in a pipeline code, it is a good idea to have one mutex per communication point. So if data flows from component A to component B and then to component C, a mutex needs to be at the point where component A transfers the data to component B, and another at the point where component B passes the data to component C.

Profiling and tracing can help identify situations in which synchronization objects are either too large or too small. When there are too few mutexes, there will be idle threads waiting for the mutex to be available. When there are too many mutexes, then there is little idle or waiting time, but a lot of time spent in creating, acquiring, and releasing locks.

Accessing Memory on Multicore Hardware

In order for these threads to perform work they need to access data, most of which resides in the main memory. Memory access characteristics, such as memory latency and memory bandwidth, can have a huge impact on performance.

Memory Latency

CPU caches exist because the time required to fetch data from main memory is relatively long. In actual time scales, consider a modern 3GHz processor capable of executing three instructions per cycle has a peak execution speed of nine instructions per nanosecond (ns). The time it takes for a CPU to load data from memory is called the latency of the operation (measured in ns or processor clock cycles). With a latency of 50 ns, this CPU could have executed 450 instructions in the time it was waiting for data. This suggests that if the processor has to wait for each bit of data that it request to come from main memory, it would spend the vast majority of its time stalled, waiting for data.

Each cache acts as a buffer that holds copies of frequently used data close to the processor where the processor can quickly read and write the data. Caches are generally designed to hold some number of cache lines (each cache line is typically 64 bytes long). Ideally, whenever the processor needs data it will already be in the cache, if not it will stall until the data is available in the cache and then proceed. Caches are most effective when the program reads and writes in a regular pattern or when the same addresses are referenced frequently enough that they stay resident in the cache. When the processor is done with a cache line it has been writing to (or when other conditions occur) any updates are written back to memory.

Modern CPUs have multiple levels of cache organized in a hierarchical way. While the caches closest to a core (level 1 or L1) are the fastest, they are also the smallest. Typically, there will be a total of 2 or 3 levels of cache, each getting larger, but slower. The following table summarizes the kinds of latencies that are typically seen for L1 cache, L2-L3 Cache, and Main Memory.

Memory System Level	Relative Latency
L1 Cache	1x
L2-L3 Cache	10x
Main Memory	100x

As the table shows, this can have a huge impact on the performance of a program. If the program is able to leverage the cache well, then most memory accesses won't cause a stall and it will be able to perform instructions one right after the other. If the program behaves in such a way that the cache is ineffective (completely random reads to memory) then every operation will require hundreds of processor cycles. Most often the cache works some of the time, but the impact of each cache miss is enough that it is worth paying attention and improving the cache hit ratio by a small amount. This can have a noticeable impact on performance.

Memory Bandwidth

It is clear that latency is an important factor, but it is not the only memory limit on performance. There is also a limited rate at which data can be read from main memory. There is both a speed limit (latency) and a volume limit (bandwidth). Memory bandwidth is a measure of the size of the "pipe" between the CPU and main system memory. Latency will often be a bottleneck, but if the number of accesses to main memory is too large, the application may reach the memory bandwidth limit. The best way to improve performance in this case is to work on reducing the rate of fetches in the application. If the memory bandwidth limit is reached, optimizations intended to reduce the effects of latency become ineffective and may even impede performance further. For example, adding software prefetches to code to keep the cache full (reducing latency effects), does not reduce the number of memory accesses. In fact, prefetching often increases these accesses since some of the prefetched data is not used. So while prefetching helps performance when latency is the bottleneck, once the bandwidth limit is reached, prefetching makes things worse.

High fetch rates are required to reach the memory bandwidth limit on single-core processes, and this rarely happens. While the overall computational throughput of multicore chips increases as additional cores are added, memory bandwidth typically increases more slowly. Over time, memory bandwidth stands to become a limiting factor for an ever-expanding set of applications.

A single-threaded program that was comfortably below the memory bandwidth limit may run up against the bandwidth limitations when parallelized on multicore processors. Thus, it is common that the first pass at parallelizing an application shows no speedup at all. Some applications may even experience slower performance as additional cores are used, since the multiple threads may end up crowding each other's data out of shared L2 and L3 caches.

Cache Coherence

The paper has discussed latency and bandwidth, which are typically the most important factors for understanding how memory use affects performance. However, multicore codes require extra attention because the cache is split up across the different cores and the hardware needs to ensure that the cache is kept coherent. When multiple threads on multiple cores access the same memory, the cache coherency mechanism (discussed below) invalidates lines and generates additional traffic. If this happens a lot it can impact runtime performance.

Multicore hardware can be complex in terms of the CPU cache. Each core often has its own L1 cache, and shares an L3 cache and access to main memory with other cores (L2 may be private or shared). Cores could have private copies of data in their local caches, but if they modify that data, there needs to be ways to let the other threads (cores) know that other copies of this data are no longer up-to-date. This mechanism is called a cache coherence protocol. There are several different protocols possible, but most modern processors use the MESI protocol.

The name MESI derives from the four different states a cache line can have: Modified, Exclusive, Shared, or Invalid.

Initially, no current data will be in a given cache line, so its state will be Invalid. Once a thread fetches a cache line with data from main memory, the line will be the sole owner of that data and will be flagged Exclusive. This means this copy has not been modified and that a copy of this data exists only in this thread's cache. A thread can write to this cache line with very little overhead, since no other thread has access. Once written to, that line is marked Modified and can continue to be accessed with little overhead. If another thread fetches the same data from main memory, it will retrieve its copy in Shared state, and then the coherence protocol downgrades the first thread's Exclusive or Modified state to Shared. If the first thread's line was Modified, that data must first be written back to main memory, then copied to the second thread's cache, and then both copies are Shared and up-to-date.

The Invalid state is used when a cache line which was shared between two or more threads gets written to by one of the threads. The writing thread keeps a copy of the line with a modified state and the cache line is marked invalid in all the other threads.

Keeping cache lines up-to-date in a multicore system requires communication overhead. In a poorly architected application, the communication overhead can overwhelm the data traffic and performance will suffer. There are three operations that are relatively expensive and should be minimized with the cache coherency protocol in mind:

1) Upgrades: Occur when a cache line is upgraded from Shared to Exclusive or Modified. This can happen when a thread reads in a cache line in Shared state, and then writes to it. In this case, the state of that cache line is changed to Invalid in all other caches that contain it.
2) Coherence Misses: Occur when a thread tries to read a cache line that would have been Shared but instead has been marked as Invalid due to an upgrade by another thread.
3) Coherence Write-Back: Occurs when a memory access in one thread forces a cache line in Modified state in a second thread's cache to be written all the way back to main memory so that the correct data can be read by the first thread.

Note that the unit of concern here is a cache line rather than just a specific data value or data structure. Common cache line sizes are 32, 64, or 128 bytes, and a cache will hold some integer number of cache lines. For example, a 64 kilobyte cache using 64 byte cache lines contains 1,024 cache lines. In multithreaded applications, there are cases where data appears to be used independently in the program, yet performance suffers because the data is shared, thus incurring unexpected coherence overhead.

Consider, for example, a program that computes a fractal image. Computing each pixel is independent of all others, and so this "embarrassingly parallel" program should scale well on multicore hardware. In the main method where variables are declared, the following is found:

```
const double dx=1, dy=2;int const w=3, pixel_number=0;
```

in which dx, dy, and w are all read-only global variables. The pixel_number variable is a global counter that lets threads keep track of where they are working. Since the four variables are declared consecutively, they can (and do in this example) share a cacheline.

The performance challenge in this example can be difficult to spot without knowledge of the cache coherency protocol. In this case, each thread will reference the dx, dy, and w values repeatedly in their calculations and will have a copy of each of these read-only variables in their local cache. Once a thread is done with its calculation block, it will update the pixel_number variable. This causes associated cache lines in all other thread caches to be invalidated and forces them to re-fetch the cache line. A re-fetch of this cache line should not be necessary until a thread is done with its work and wants to update pixel_number. But instead, the thread must re-fetch several times in order to keep using dx, dy, and w, which share the cache line with pixel_number. Once another thread completes its work, the update cycle happens again to all other threads. This behavior is known as "false sharing," where data (e.g. heavily used local copies of read-only dx, dy, and w) appear to be independent with regard to each thread, but are actually linked due to sharing the cache line with global shared variables (e.g. pixel_number).

Other Cache Issues

False sharing issues can be quite difficult to discover until one understands the cache coherency protocol and how cache lines are fetched and utilized. There are many scenarios, as outlined below, and most are not limited to multithreaded applications, but can plague single-threaded programs as well.

Data Layout

For optimal performance, the most frequently used data should be in the cache and fully utilized before they are evicted. However, the way a programmer or compiler organizes variables and objects in memory can lead to a mix of frequently used data in the same cache line as data that are rarely or never used. This poor data layout impacts performance in important ways. First, with bad data layout, the number of cache lines required to hold all the frequently used data increases. This directly correlates to an increase in the number of cache misses and the number of cache line fetches. Similarly, because less useful data can actually be held in the cache, inefficient data layout reduces the effective cache size, and therefore increases the cache miss ratio. And finally, increasing the number of fetches also increases the memory bandwidth requirement of the program - which makes programs near that limit at greater risk for severe performance penalties.

A classic example of what causes poor data layout is a partially used structure. Take, for example, a structure with four integer fields. Certainly somewhere in the application all parts of the structure are required, but consider the case where only two of the fields are used in a computationally intensive loop. When looping through an array of these structures, each access brings in the entire structure and its neighbors into a cache line. However, if only two fields are touched (read or written), then more data is being moved around than is actually being utilized. This section of the program would be using twice the memory bandwidth required and causing twice as many fetches from main memory. If the program can be rewritten to split the structure without affecting the overall design, then when that key loop is reached, twice as many useful values will be in the cache at any given time, significantly reducing the wasted bandwidth. The caveat is the modified code may be harder to understand. But if performance gains are important, some best practices may be sacrificed. This discussion also applies to using a data type that is larger than needed. If a 16-bit data type is sufficient for the elements of an array, using a 64-bit type results in a cache line holding only one quarter the number of elements than if the variable was properly sized.

A potentially more subtle issue is how variables are arranged within a data structure. Most modern processors require memory access to be aligned with addresses in what is known as “natural alignment.” Some systems allow for unaligned accesses, however, better performance can usually be attained when those processors read aligned data. In a natural alignment system, a field (data element) must be stored at an address that is an even multiple of its size. This means a 4-byte field must be stored at an address that is a multiple of 4, and an 8-byte filed field in a multiple of 8. Single byte objects, like chars, can be put anywhere.

Many developers do not immediately give this alignment much thought when they lay out a data record, and instead work with what makes the most syntactic sense or is most convenient at the time. As an example, consider a struct containing two single byte chars (a and c) and a four byte integer (b). If the struct is defined as “char a; int b; char c” the memory layout will start with “a” but then need three bytes worth of padding to allow “b” to be aligned at a multiple of four. Then “c” follows. So this struct containing six bytes worth of variables actually takes up nine bytes because of the poor internal alignment. A better definition would be “int b; char a; char c,” which results in the integer at position zero and the two chars immediately following, using a total of six bytes (as expected).

When designing or redesigning the data layout, manually count offsets to ensure alignment. An easier rule of thumb to follow is to start with the fields that have the greatest requirement in terms of alignment, then continue down to the other fields, i.e. order the fields from biggest to smallest. However, if the structure is so large that it requires more than one cache line, it is important to ensure the most commonly used fields are close together, similar to the first data layout example.

Even though the six byte struct has resolved the internal alignment issue (because it is just six bytes), there will still be an external alignment problem since addresses are powers of two. For an array of these structures, the compiler will still need to add two bytes of padding after each struct element so the next element is properly aligned. This kind of external alignment issue is more challenging to avoid. One possible solution is to pull apart the structure, separating out the less frequently used fields as discussed above.

Access Patterns and Prefetching

Caches work partly because many programs have a characteristic called spatial locality. If a given bit of memory has been accessed, it is likely that the next bit of memory to be accessed will be nearby, often on the same cache line or on a cache line recently fetched. Most processors use a technique called hardware prefetching which builds on this concept. The hardware prefetching kicks in when the program starts reading in a regular access pattern. When this happens it will start requesting memory ahead of the processor. By doing this it pre-loads the cache with lines that are likely to be used next. Ideally, this eliminates the need for the processor to stall while it is waiting for a cache line to be fetched.

This works best with regular access patterns and can be completely undermined when the program is structured in a way that leads to frequent unpredictable (random) memory accesses. Commonly used pointer-based data structures store data at widely-distributed addresses. Traversing these structures often boils down to looking up pointers and dereferencing those pointers over and over again.

Often, structures like these are chosen for specific and valid reasons in designing the application. But if the data is traversed frequently and performance is critical, it may sometimes make sense to choose prefetching friendly data structures. Consider a linked list that is created from dynamically allocated memory regions. These regions can become spread out across memory so that when the list is traversed, memory access is quite unpredictable. As new elements are added individually to the list, the problem gets even worse due to further fragmentation. This challenge can be addressed with a custom allocator that places data close together.

Non-Temporal Data (Single and Multithreaded)

Non-temporal data is accessed and cannot be reused before it gets evicted from the cache. For example, an algorithm that does some transformation on an array - reading an element once from one location, doing a bit of work on it, and then writing it once to another location - falls into this category because there are no data reuse opportunities. Sometimes reuse can be improved for large data sets that do not fit in the cache by using a blocking algorithm, which is a way to do the operation in cache-sized blocks, usually via deeper loop nesting. For some operations (e.g. matrix-multiplication) this offers dramatic performance gains, but for others, this does not always improve the situation because at the innermost loop level, the data are used just once and evicted between iterations. In these cases, it is known that the data will be evicted before reuse, so caching the data in the first place may be pointless. The non-temporal data occupies space that could be better utilized by unrelated data that may be reused before eviction.

To illustrate a case where some steps can be taken to address the issue, consider a single thread with a 3MB cache and an operation that must iterate through both a 2MB array and then an 8MB array a few times. After the first iteration the program starts back at the beginning of the 2MB array and must reload it; having just completed the 8MB iteration, the 2MB array will have been completely evicted from the cache. As it moves on to the beginning of the 8MB array, the 3MB cache will contain the entire smaller array and remnants of the end of the 8MB array, so data will again have to be fetched from main memory.

Each time the program goes through the arrays, cache line fetches of both are required. However, the 8MB array will never fit entirely into the cache. If the processor knew better, it could keep the smaller array in the cache and fetch only smaller pieces of the large one. The program would still get fetches all the time for the larger array, but performance may improve with the good cache hits in the small array. Most modern processors have instructions for including non-temporal hints and telling them not to attempt caching an array that does not fit into cache. With a common replacement policy to evict the least recently used data first, when the program repeatedly iterates over an array that does not fit in the cache, there will be a 100 percent miss ratio because the cache always contains the most recently used data, which will not be used again.

In cases where the data is fairly close to the cache size, there is much room for improvement. For example, if there is the 3MB cache with a single 4MB array to loop over, if non-temporal hints (like prefetches) are added that keep the first 3MB of the data stored in the cache, the miss ratio drops to 25 percent (corresponding to the last 1MB that does not fit). Performance improves considerably for the first 3MB that are retained in the cache. Note that this is an illustrative example; it is not recommended to try to fill the entire cache, but if 1MB is reserved for other uses, the number of fetch misses could be reduced dramatically.

Multithreaded applications may benefit from the same kinds of non-temporal hints, and such hints can also be used for cases when threads share some level of the cache hierarchy, usually L2 or L3. The problems can be similar to those already discussed; for example, if threads are working with different data sets, one small and one large, the thread with the large data set may cause the smaller data to be evicted from the shared cache. Both threads lose efficiency. Since the thread with the larger data set will see cache misses, adding non-temporal hints can prevent it from caching its data and evicting data that the other thread will be reusing.

Most workstations and servers are not dedicated systems, so applications will often not be the only thing running. For embarrassingly parallel applications, it may be other instances of the same application sharing system resources; but for multi-user servers, it may be virtually anything. Under these scenarios, adding non-temporal hints when the data accesses will result in cache misses may help reduce the overall pressure on the cache system and reduce the number of cache fetches in other applications. When analyzing applications for performance on shared systems, it may be necessary to report a smaller cache than actually exists on the hardware. For example, with four instances of the same single-threaded application running, it may be useful to assume just one quarter of the full cache is the most that will ever be available. Keeping concurrency in mind and not optimizing for the ideal situation where the program has 100 percent of the system resources rarely hurts when the program is able to run full tilt on the system.

Tools for Increasing Multicore Optimization

When developing for multicore, it's critical that you are able to see and control what's happening with the threads in the application. Debugging tools should be able to:

- Show what is happening in all the threads;
- Allow the ability to easily define groups of threads and operate on them either as a whole group or some subset of the groups;
- Smoothly switch between groups of threads for evaluation and analysis;
- Synchronize and sequence threads;
- Revisit and examine the entire sequence of thread activity that led to a failure;
- Show how threads are managing memory allocations; and,
- Detect memory leaks and buffer overflows.

While transitioning applications to multicore, sometimes data race conditions occur, so a tool that shows data changes and unexpected behavior is useful. Tools should allow developers to be able to set a watchpoint on variables of interest, and even better, go back in the program to pinpoint the thread or action that caused a crash. It is also useful to be able to automate some of the debugging capabilities and to let a session run unattended at times with a script that can be specifically defined.

With multicore machines, although programs can benefit from additional cores, memory capacity and bandwidth are not necessarily increasing to keep pace with those additional cores. Tools should allow developers to understand memory usage and detect inefficient patterns that can optimize usage.

TotalView

TotalView is a scalable and intuitive debugger for parallel applications written in C, C++, and Fortran. Designed to improve developer productivity, TotalView simplifies and shortens the process of developing, debugging, and optimizing complex applications. TotalView provides a powerful combination of capabilities for pinpointing and fixing hard-to-find bugs, such as race conditions, memory leaks, and memory overruns. Providing developers the ability to step freely, both forwards and backwards, through program execution, TotalView's unique reverse debugging capabilities drastically reduce the amount of time invested in troubleshooting code. To help developers maximize hardware capabilities, TotalView also provides debugging support for NVIDIA® CUDA™, OpenACC®, and the Intel® Xeon® Phi™ coprocessor.

Conclusion

In the first paper, Part 1: Overcoming the Challenges of Developing Applications for Multicore Systems, the benefits and challenges involved in taking a serial application and parallelizing it was discussed. This process is essential for any organization that wants to benefit from the computational power and performance offered by multicore processors. The paper focused on getting parallelism added to the program and getting correct behavior from that newly parallel program.

This second paper took things a step further by discussing multithreaded optimization and focusing on a number of potential pitfalls that need to be avoided in order to achieve peak performance. The paper discussed performance issues related directly to multithreading such as load balancing, choosing the right level of concurrency, along with the right level of synchronization. Also introduced was the cache memory hierarchy and how critical it is to make efficient use of the cache memory subsystem in order to achieve good application performance. Finally, the paper introduced a powerful tool: TotalView. This tool helps developers understand what is going on in programs and helps users more easily identify and resolve performance issues so that applications can achieve the full performance promised by modern multicore processors.

About Rogue Wave Software

Rogue Wave Software, Inc. is the largest independent provider of cross-platform software development tools and embedded components. Rogue Wave's proven technical solutions simplify the growing complexity of building and testing quality software code. Rogue Wave customers improve software quality and ensure code integrity, while shortening development cycle times, and include industry leaders in the Global 2000 as well as leading government institutions and universities. For more information, visit www.roguewave.com.