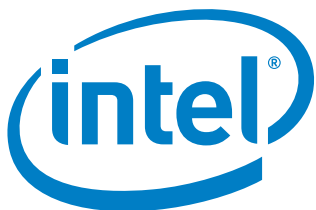


101001010110101101011010110110010101001
01001010101001010100011100101010010
0100101110010010010101001001001
010010101001010101101010101
011010010101101010110110
0100100111001010100100001010101

Managing Multi-Core Projects





Contents

Part 1: HPC on the Parallelism Frontier	2
<i>by Steve Apiki</i>	
Part 2: HPC Project Implementation	5
<i>by Steve Apiki</i>	
Part 3: Multi-Core Development in the Enterprise	9
<i>by Steve Apiki</i>	
Part 4: The Enterprise Development Cycle Meets Multi-Core	12
<i>by Steve Apiki</i>	
Part 5: Desktop Software Considerations	16
<i>by Steve Apiki</i>	
Part 6: Diving Deeper on Consumer Software	19
<i>by Steve Apiki</i>	

Part 1: HPC on the Parallelism Frontier

HPC may be on the leading edge, but key advice like going parallel early, thinking strategically, and spreading knowledge throughout the team applies to all development managers.

by Steve Apiki

HPC was parallel when parallel wasn't cool. In high-performance computing, where developers have long experience with parallel computing and where large clusters are often the target platform, the multi-core driven concurrency revolution isn't catching anyone by surprise. From these pioneers, we can learn that parallelism makes a competitive difference-and that it doesn't happen overnight.

In HPC, as in other software, all signs point to increasing parallelism as the surest path to improved system performance and to competitive advantage. Just when you were warming up to programming SMP within the nodes of a cluster, asymmetry between types of core processors will add additional complexity. But whatever the architecture of the next generation system, whether SMP (symmetric multiprocessing) systems based on multi-core processors, FPGA supercomputers, hybrid GPU designs, or other asymmetric configurations, all can be approached with some proven principles for managing parallel software projects.

Parallelism is a defining feature of HPC, as are large data sets and long run times, sometimes measured in days or weeks. Typical HPC applications divide the dataset among multiple processors, achieving parallelism through data decomposition. Data decomposition can be an effective technique in games and video applications as well.

HPC platforms may be threaded, shared-memory systems, or they may rely on message passing for communication and coordination among large collections of more independent nodes. Both concurrency techniques may be used together. Although not as performance sensitive, multithreaded enterprise applications face similar architectural complexity in terms of program correctness.

Threaded SMP systems enjoy a tremendous bandwidth and latency advantage over distributed memory systems, but scaling is limited. Multi-core processors raise the scaling limit of SMP, increasing its applicability to parallel programming problems, whether in HPC or in the enterprise.

Starting Points

Revisions in HPC applications are frequently driven by a user requirement for greater capacity (the ability to handle larger datasets). That may mean more threading to improve performance. In addition, look for ways to apply parallelism to add new capability (the ability to solve new problems with additional resources). The new prevalence of multi-core will make it easier to find third-party parallel components that help in this regard. Both added capacity and added capability are desirable, but while the former keeps you ahead of your competition, the latter can put you in a whole new market.

For development managers, the challenge is not so much to introduce parallelism as to plan development approaches that continually target scalability. That challenge has to be met not only at the start of a new project, but through successive upgrades.

If you're planning the next version of an application, add additional parallelism along with other changes. One approach is to concentrate on the new modules that implement new features, making sure these make the best use of parallelism. For this to be effective, you need to make sure that new code is sufficiently isolated from old code. This won't always be possible, but it's a further argument for walling off new features in separate modules. By keep-

Additional Resources

1. *Intel Threading Tools*
2. *Intel Trace Collector and Intel Trace Analyzer*
3. *Intel VTune*

Managing Multi-Core Projects

ing new functions modular, you can aggressively add new parallel code while limiting its impact on existing code and limiting the scope of required regression testing.

Modularity is a desirable goal unto itself. Because the interfaces between systems are so sharply defined, message-passing systems tend to be more modular than threaded programs.

Amdahl's law is a well-known principal that describes the benefit you can expect from moving portions of a project from serial to parallel processing. The more direct approach to better performance is to follow where Amdahl's law leads and to go after serial regions in existing code. Attacking this problem requires a thorough performance analysis, which historically has meant reading through code, but automated tools can improve the process by increasing coverage. Intel VTune Performance Analyzer's call graph profiling can help here.

Other tools can help to measure the performance of large cluster systems. For applications using MPI (Message Passing Interface), Intel Trace Collector and Intel Trace Analyzer can analyze performance on cluster systems of over 1,000 processors.

There's no working around a bad design, but that doesn't mean that a good design can't be improved by some of the same tools and techniques that you might apply to legacy code. Testing modules for performance as well as correctness is important before introducing the complexity of a fully integrated system. Intel Thread Checker has unique capabilities for debugging threaded applications that are useful at this stage. For performance analysis, Intel Thread Profiler can compare threaded performance of several versions.

Real-World Conditions

There's no way to cover every case with simple rules. What might be a pragmatic solution that keeps a project on schedule might, under different conditions, be a shortsighted fix that hampers long-term performance. Tactical judgment needs to be applied in a strategic context, and there's no substitute for experience in developing that judgment.

Bob Kuhn, Intel's Technical Marketing Director for Advanced Parallel Software Platforms, is a parallel-computing expert and a veteran of many a parallel programming development effort. Kuhn says that many HPC projects at first sought to increase performance by optimizing away the current bottleneck, using the easiest mechanism, then attacking the next bottleneck that cropped up in a similar fashion. "For pragmatists," says Kuhn, "that may provide sufficient performance."

But Kuhn cautions that such an approach has a point of diminishing returns-what he terms the "project manager's version of Amdahl's law." Eventually, the most egregious bottlenecks are eliminated, and each successive target of optimization delivers a lower marginal performance benefit for the same amount of development resources. Kuhn describes a more sustainable approach. "Analyzing the goal with Amdahl's Law, start by saying everything in the application must eventually be in the parallel region to reach your goal," he says. "What data structures must be parallel and without synchronization?" Improvements along these lines may show lesser short-term speedups per developer-hour, but they have greater prospect for long-term gain, with the benefit coming from data decomposition. Optimization often makes an application structurally more complex, making it harder to improve overall parallelization after optimizations have been made. "After many changes, you find you have to do much more to switch to data decomposition," says Kuhn.

On the other hand, according to Kuhn, sometimes you have to consider options other than data decomposition, even in HPC. For example, a workflow model might be a more practical first-pass way to quickly integrate third-party programs in your HPC application than a deep parallel integration. In this case, the clean stdin/stdout interface of a workflow approach avoids a number of bugs that would surely crop up in a shared-memory integration of two large, complex pieces of code from different sources.

The Dream Team

Part of the planning phase should be an evaluation of the skills of project team members. It's important to have familiarity with parallel programming throughout the team, from those responsible for the initial design to those providing field support.

In the ideal case, a "dream team" would deliver applications that delight the user with new functionality, scale up to the number of cores on the newest processors, all on schedule with market availability. In building that dream team, you'll get better results teaching an engineer with domain knowledge the principles of parallelism than training a computer scientist in what your users expect. Work toward a team with these skills:

- **Parallelism Architect:** Needs expertise in parallelism, with experience in parallel algorithm design, and deep knowledge of your application. Grow one of your application experts through classroom training on principles, best practices, and tools.
- **Computational Scientist:** An individual that combines scientific domain knowledge with parallel-computing expertise. Bring in one of this new breed to discover how the latest parallel methods and structures can increase your functional and performance advantage.
- **Application Developers:** Developers should be trained in the principles of parallel processing, know parallel tools, and be able to build thread-safe component interfaces.
- **Test Engineers:** Test engineers should be quite strong in parallel debugging skills and familiar with parallel analysis and profiling tools. Key adversarial testing skills in parallelism vary not only the number of threads, but the order in which they execute.
- **Field Support Engineers:** These engineers need some parallel debugging skills and should also have knowledge of parallel tools. In highly scalable software, your customer may have a larger cluster than you can afford at headquarters. Field engineers need parallel application skills to work with these customers.

Of course, reality rarely reflects the ideal case, but the critical point here is that all team members need to have experience with parallel computing.

Grow three types of knowledge: knowledge of what your users want in each function, horizontal knowledge of the architecture in clean synchronizing interfaces, and vertical knowledge in building and using thread-safe components. Don't have a developer that's added parallelism in one function move on to adding parallelism in another. The function's developer knows the code best and will have the best intuition on what is and what is not parallel.

Finally, whether you're developing an HPC application or productivity software, the fundamental things apply: Target some features where you can add parallelism today. Think strategically about parallelism in the whole application. And develop parallel skills in every member of the development team.

In the next installment of this series, we'll move beyond the planning phase and explore the management issues surrounding implementation, test, and debug of parallel HPC systems.

About the author: Steve Apiki is senior developer at Appropriate Solutions, Inc., a Peterborough, NH consulting firm that builds server-based software solutions for a wide variety of platforms using an equally wide variety of tools. Steve has been writing about software and technology for over 15 years.

Part 2: HPC Project Implementation

Directing a parallel-programming development project requires knowing the components, knowing the tools to use, and knowing your customers. Part 2 of our management series spotlights some of the lessons learned in fielding parallel HPC applications.

by Steve Apiki

Parallelism affects everything about the way you manage a development project, from the skills you need to develop in your team to the debugging tools you'll use in maintenance. In part one of this series, we focused on the up-front challenges, including staffing, planning, and design. Here we'll get into the rest of the process, looking at how parallel programming changes implementation, test, and debug. As in part one, the nominal topic here is high-performance computing. But many of the lessons learned in HPC, where parallel development experience is deepest, are fundamental-lessons like using thread-safe libraries as a foundation for concurrency, or continually improving scalability over time. These apply as well to software fields where multi-core is just starting to raise the issue of effective management of parallel projects.

Multi-core changes the competitive landscape by making threading a critical source of performance advantage. In HPC, parallelism may be built in at two levels—at a low level, with threads in a shared-memory system, or at a higher level, with message passing. Shared memory systems can finesse the bandwidth restrictions and added latency of message passing, but they scale only up to the number of processors you can put in a single node. By putting a multiplier before that number, multi-core processors make threading an important source of additional parallelism and additional performance. This is true in enterprise software as well as in HPC, where more threading can be applied to improving single-transaction performance in a server farm.

Building on Libraries

Delivering fast, reliable threaded software takes working from a good design and a team with the right skills, but it also takes starting the implementation at the right place. Pre-tested threading libraries such as OpenMP and Intel's Threaded Building Blocks (TBB) can shorten both development time and time spent in debug. These libraries provide a higher-level and high-performance approach to threading and remove some of the mechanical, error-prone work that come with the territory of SMP.

OpenMP is a threading API and a set of compiler pragmas for C++ and Fortran development of shared-memory systems. Intel's recently-introduced TBB is a C++ template library optimized for Intel processors that does not require specific compiler support. Both reduce the burden of explicit data partitioning and explicit synchronization compared to working with native Windows or POSIX threads.

It's good practice to work with foundational libraries like these, OpenMP and Intel TBB for shared-memory systems, MPI libraries like Intel MPI for message-passing systems. As we'll see in some industry examples below, MPI libraries can play a critical role in overall system performance.

You might be able to further reduce development costs by incorporating additional libraries of common parallel algorithms. The growth of multi-core, and therefore of parallel programming, is beginning to make these kinds of components more plentiful. It's worth researching third-party libraries that can get you off to a faster start, or help you replace portions of homegrown code that may be under-performing or hard to maintain.

Some of the functionality you can get off the shelf includes numeric, statistical, and other math-oriented routines, and media-related functions like video coding and signal processing. The NAG libraries (from NAG) and IMSL libraries (from Visual Numerics) are broad functional libraries, which are both available in thread-safe versions. NAG

Additional Resources

1. *Intel Math Kernel Library*
2. *Intel's Threaded Building Blocks*

Managing Multi-Core Projects

offers a version that is itself internally threaded. Intel partitions its functional libraries into two packages, Intel Math Kernel Libraries (MKL) and Intel Integrated Performance Primitives (IPP). Intel's libraries are also thread-safe and internally threaded (using OpenMP).

You can, and should, verify the thread-safety of third-party libraries through your own testing. Even thread-safe libraries that are internally threaded can be the source of thread conflicts with the calling program if they aren't used correctly. Intel Thread Checker is an excellent tool for testing the thread safety of libraries and for finding threading problems in the system as a whole.

Test and Support

There's more to test in a shared-memory design than in a message passing system, if for no other reason than the greater coupling of SMP leads to a proliferation of test cases. Tests of highly-threaded applications should vary thread execution order and timing as well as other conditions. The right tools are critical for effective test and debug of parallel systems. Again, Intel Thread Checker may be used to isolate problems that are uncovered in testing.

In consumer software, testing can't cover the variety of customer hardware configurations, but it can cover the number of cores in a customer system. In HPC, there may be fewer configurations to test, but it's possible that a customer will deploy on more cores than can be tested on the development cluster. Scaling limitations or bugs related to interactions between a large number of nodes may only surface at a customer site.

In these cases, final testing will necessarily occur with the customer, at deployment time. This scenario makes it especially important to have field support engineers with parallel testing and debugging skills, and with experience using parallel tools.

In the Field: Examples from the Oil and Gas Industry

Let's walk through some real examples. We'll use two case studies from the oil and gas industry to illustrate some of the points we've been discussing, both in this article and in part one of this series. Each is a development effort involving Intel engineers working with industry customers.

High-performance clusters are a critical part of the array of technologies deployed in the oil and gas industry to locate oil and gas deposits. There are a variety of scientific applications that run on these clusters, using message passing, threading, or a combination of these mechanisms.

In our first example case, a seismic company sought to speed up a long-running seismic imaging application that consumed 40 percent of total compute cycles in the data center. The application ran on a cluster of several hundred nodes, using MPICH, an open-source MPI library. The company viewed even a small gain as worth a significant amount of development time, for two reasons. First, reducing the load on the cluster would speed up other software and thus the overall process. Second, since the seismic imaging application run and data interpretation could take up to a month, even a small percentage gain could mean a day or two of real time.

The company's initial plan was to replace the MPICH 2.x library with Intel MPI. This alone led to a 2x speedup, but during the phase-in of Intel MPI, engineers discovered that they could further boost performance with some additional steps. These were a faster sparse-matrix multiply routine, which made for better single-node performance, and an added data partitioning stage, which reduced bandwidth consumption. Finally, Intel engineers were also able to improve global data communication performance, an improvement that was later rolled back into Intel MPI itself.

Each of these changes was built into a library. The company then integrated the libraries into its application, making for a relatively low impact on existing code. While Intel engineers were making changes, the company's non-parallel code continued to evolve, so there was additional by-hand merging of the two code bases that needed to be done before the project was completed.

Managing Multi-Core Projects

Table 1. Steps taken in each of our example projects, broken down by phase of development.

	Seismic Imaging Speedup	Multi-User Visualization
Design	(1) Use faster MPI library; (2) Add data partitioning step; (3) Improve single-node performance; (4) Improve global data communication.	Thread each node in MPI cluster to support multiple clients.
Implement	Build changes as libraries, then integrate.	No thread safe MPI libraries; use asynchronous messaging.
Test	Realize 4.5 - 5x speedup.	Tools (Intel Thread Checker) help compensate for less threading experience.
Tune	Reduce number of libraries by rolling performance changes into MPI library.	Bottleneck-by-bottleneck, using Intel VTune.

The company saw an overall improvement of 4.5-5x on the seismic imaging application over the two-year span of the project. The project involved the geophysicist in charge of the application, two engineers from the seismic company, and two Intel engineers, one a computational engineer and one a library expert.

The second case wasn't directly driven by a demand for greater performance. Instead, the company added the capability to support several users on a highly-parallel visualization system that initially served a single visualization client. The system allows geophysicists to interactively explore segments of the earth's crust, running 550 MB of geological survey data through a 64-node cluster. At the start, the single visualization client communicated through a single master node on the cluster.

That arrangement underutilized the cluster. The project required threading each node in the cluster to support multiple visualization clients, and interleaving visualization tasks with computation tasks.

Combining threads and MPI proved to be a problem as the project progressed to implementation, because at the time there were no thread-safe MPI libraries. (There are now two options, Critical Soft WMPI and MPICH). The company settled on asynchronous messaging using an MPI lprobe polling loop.

The company made heavy use of Intel Thread Checker during the debugging phase. Having a threading debug tool was important at this point because the development team was much more familiar with MPI than with multi-threaded development. Nevertheless, they were able to use Thread Checker to its fullest, even bumping up against a limitation of the tool itself in working with memory-mapped files.

The team tuned the software using a classic bottleneck-by-bottleneck approach. They made considerable use of Intel VTune in this phase.

There were no performance metrics on this project other than the goal to enable multi-users without individual users seeing any performance degradation. The project team of nine (part-time, the equivalent of 2 engineers full time), plus one or two Intel engineers completed the work to support multiple users in about six months.

Each of these projects saw some significant benefit in terms of the "four Cs:" capacity, capability, creativity, and cost of parallel programming projects. In the seismic imaging case, better performance meant improved capacity and reduced cost, both in run times and in reduced impact on the data center. In the visualization project, threading nodes to support multiple clients meant an added multi-user capability, greater opportunity for end-user creativity, and reduced cost in better cluster utilization.

Managing Multi-Core Projects

HPC projects like these are where all the parallel development action has been, but multi-core is changing all that, of course. In part three, we'll start looking into managing multi-core development projects in enterprise software.

About the author: Steve Apiki is senior developer at Appropriate Solutions, Inc., a Peterborough, NH consulting firm that builds server-based software solutions for a wide variety of platforms using an equally wide variety of tools. Steve has been writing about software and technology for over 15 years.

Part 3: Multi-Core Development in the Enterprise

Additional Resources

1. *Ensure High Quality SOA Applications*
2. *Intel Open Fire Forum on DevX*

This chapter on managing multi-core development focuses on finding parallelism in service-oriented systems that will come after the client-server era. Multi-core means opportunities for a better customer experience- if you take the right perspective.

by Steve Apiki

There are many different ways to look at the advantages of multi-core processors, so as you watch the rollout of quad-core processors this year you need to make sure that you keep the pair of glasses on that gives you just the right perspective on them. While it's true that quad-core will mean a more efficient data center, more flops per watt, and more transactions per second per square foot, what that means isn't the same to everybody.

For an application or service development manager, those particular metrics are examples of thinking from the wrong perspective. That's because for a computer center manager, it's all about efficiency, and that means maximizing throughput. But for a development manager, it has to be all about customer experience, and that means minimizing latency. Multi-core processors can provide the processing power to keep up with customer demands, but only to the extent that you can apply parallel programming to build faster-responding services and applications.

SOA and grids represent a kind of "macro concurrency," concurrency expressed in business services or compute nodes. Multi-core represents an opportunity for "micro concurrency," which brings parallelism down to a single server or on a single node. You'll need to take advantage of this low-level parallelism to get the most performance out of each transaction and out of the overall system.

In the brave new world of SOA, Web 2.0, and SaaS, how do you make use of multi-core servers to deliver the best customer experience? For each of these architectures, performance depends on efficient communication among a set of loosely-coupled services. It's in managing that communication overhead, in minimizing I/O latency as well as maximizing compute performance, that the new multi-core servers will have the greatest effect.

Two Kinds of Latency

Take the trendiest of trends, a Web 2.0 mashup, as an example. Say you build a Web application that combines data from several sources and provides an integrated dashboard, with some visualization tools. In addition to providing your customers with new capability, you've shifted the burden of page assembly from the clients to your new service.

Depending on the kinds of data manipulation required, combining data in this way can be data intensive, I/O intensive, or both. The mashup can't respond any faster than the services we're drawing from, so the slowest of these determines our minimum latency. What we need to do is to minimize both compute latency and the I/O latency of concurrent communication with other services. We can address both kinds of latency with threading, using data decomposition to structure compute threads and functional decomposition to structure I/O threads.

If the service is calculation intensive, look for parallel segments that can be split across threads using data decomposition and thread pools. Use asynchronous messaging to minimize the impact of I/O on overall latency. At a high level, proper I/O threading for interconnected services means making the process as asynchronous as possible and designing the service so that the main processing thread continues to run with minimal blocking on I/O.

At a low level, and where I/O performance is critical-in communication between nodes of a grid, for example-threading can reduce I/O latency by increasing the efficiency of messaging. Messaging is an abstraction that makes development of parallel programs easier, but a poor messaging library implementation can doom performance with excessive consumption of memory and memory bandwidth. It takes careful threading and a good API to minimize

Managing Multi-Core Projects

latency in getting the message off the network and into memory that the handling thread can access.

When working with a third-party messaging library, it's hard to gauge efficiency unless you can run benchmarks. As a starting point, look for asynchronous messaging when comparing libraries.

When you thread for data parallelism, you can evaluate the result of the parallel computation and develop confidence in the correctness of your implementation. With functional decomposition for I/O concurrency, executing services concurrently, correctness is just as important, but more difficult to evaluate. You'll want to schedule significant test time early in the development cycle, with simulated services, so you can work out bugs in I/O threads before introducing dependencies on real online services.

Thread pools are a familiar part of any server system. Thread pools reduce thread management overhead, balancing the time impact of expensive thread creation with the memory impact of maintaining idle threads. Thread pools should be considered in any multithreaded design, including those for multi-core systems.

Where possible, take advantage of thread pool mechanisms provided by the platform (by the .NET CLR or by OpenMP, for example, or the Java 5 `ThreadPoolExecutor`). On .NET, using the CLR thread pool can provide a significant performance advantage to managed code versus unmanaged code using Win32 threads. The advantage can be enough that, contrary to what you might expect, a threaded C# application can be faster than an equivalent C++ implementation.

Getting There

The next round of high-end servers is going to be based on multi-core processors, so it's important to start thinking about how multi-core affects your development process now. Whether you're updating single-threaded legacy J2EE code or starting a new service development project from scratch, work out your multi-core strategy at the very start, before you even get into planning the details of the project itself.

Choosing the right approach starts with recognizing where you are, in terms of both the code you're starting with and the threading experience and expertise of your team. If your team has little experience, reduce the scope of the project by paring features to an absolute minimum. Forget the fine details and focus on major functionality. Find the shortest path through development and get the project to market, or into the hands of an early-adopter, as quickly as you can. In this way, you'll get the product out the door and begin to develop some threading expertise throughout your team. Because it's a small project, you'll also minimize the impact of the almost inevitable schedule slip-page that comes with learning new development practices, and have an opportunity to improve your own scheduling skills for future multithreaded projects.

The minimal project approach can be applied to performance improvements of single-threaded legacy code as well as to new development efforts. With legacy code, you'll also need to plan time up front to explore how threading for data or functional decomposition might better performance. Tools such as Intel's VTune can help you find serial sections that might benefit from parallelism. The trend to more core and less clock makes threading so much more critical to performance that you might revisit threaded approaches that you'd considered not worth the development effort on single-core, single-processor servers.

With a code base that's already threaded, focus on incremental improvements. Again, multi-core increases the value of threading when measured against other performance strategies. Look for other areas that could be threaded, ways to reduce message overhead, or more highly-parallel data processing algorithms. Tools can help here, too, in particular profilers like Intel Thread Profiler.

Regardless of where you're starting from, you'll get the greatest advantage from multi-core if you can create an emphasis on threading and parallelism throughout the project lifecycle. That means not just planning for multi-core, but developing an integrated approach where the parallelism model that's adopted early in the process is improved

Managing Multi-Core Projects

through feedback from testing and tuning.

In our next installment, we'll push further for an integrated approach to parallelism. Then we'll break down the development cycle for enterprise software projects and discuss the steps you can take to in each phase to ensure the best multi-core performance.

About the author: Steve Apiki is senior developer at Appropriate Solutions, Inc., a Peterborough, NH consulting firm that builds server-based software solutions for a wide variety of platforms using an equally wide variety of tools. Steve has been writing about software and technology for over 15 years.

Part 4: The Enterprise Development Cycle Meets Multi-Core

How does multi-core change the development cycle for enterprise software? In part 4 of our series on managing multi-core development, we discuss how an integrated approach to parallelism changes both the process and the manager's role.

by Steve Apiki

Every successful company, maybe even every successful team, has a development process that works--what team members think of as "their" process. Making some adjustments to the process to accommodate the development of parallel software can have a big impact, keeping the process working as you transition to multi-core with the same high level of productivity you enjoy today.

Intel's Developer Products Division (DPD) has a process, too, one tuned for the development of parallel software and honed by over ten years of experience working with customers, primarily in HPC domains. The DPD process works for multi-core development just as it has worked for other parallel processing environments. It may be helpful to look at its approach, not as a model, but as a sample, something to crib from as you consider how multi-core may change your own development practices.

Intel DPD breaks its process down into four steps. These four steps are intended to be applied continuously, version over version, improving parallelism with continued revision.

- **Discover:** Find the natural modes of parallelism in an application. Determine which problems are appropriate for parallel decomposition, how functions might be refactored so that different sections can run simultaneously.
- **Express:** Design and build an implementation structured around the parallel decomposition you've developed in the discovery step.
- **Confidence:** Test the implementation to develop confidence in its working correctly. Use your findings to improve your implementation or update the model. Take this feedback into the earlier steps.
- **Optimize:** Tune at the end of the incremental revision, after you're satisfied with the correctness and baseline performance of the implementation.

These four steps are general, and can be related to any development methodology. Within an agile development framework, as an example, the process can be followed through in a single development iteration, restricted only to the sections of the software actively under development, and completed within a release cycle. More generally, if you have a refactoring process, think about adding this threading process as a special case.

Parallel in Every Phase

Developing for multi-core changes the structure of the process. It also means some new considerations for each step along the way. What follows are some pointers to follow (and pitfalls to avoid) in directing a multi-core development project. Let's consider the project step-by-step.

In the discovery step, consider tools early. Look for the threading and messaging tools that best suit your application. Tools are a key element of success in developing parallel applications, and they aren't all alike, so put in the effort up front. Find the right analysis and debugging tools as well as the right threading or messaging libraries. Test components and libraries for thread safety, relying as little as possible on vendor claims.

Threading is a design consideration, not an optimization. Include discussions of threading and thread coordination as you work through the project design. This will help to minimize threading conflicts as you get to implementation. If you can use it, data decomposition should be preferred over strictly functional threading, as data decomposition will

Managing Multi-Core Projects

scale better to more cores.

Train application experts in threading and parallel techniques. These are the developers that will do most of the implementation in the expression step. If you're going back and threading an existing program, use the original developer of a module to thread his or her own code, rather than a parallel-programming expert.

The confidence step in a threaded project introduces thread-specific testing requirements. In addition to testing additional thread-interaction scenarios, test the threaded version of the application for consistency with a single-threaded implementation, if one exists. This becomes more important as more users deploy on multi-core systems.

Optimization is critical for best parallel performance. Don't expect that just implementing threads will create a dramatic performance improvement. Spend the time on the back end to get the most out of the design by tuning locking, shared memory, cache interactions, and other performance parameters. Automated tools such as Intel's VTune and Thread Profiler can help a great deal in this part of the process.

Creating a fast threaded application or threading existing code takes attention to parallelism in every phase. If you skip the front end of the process, you'll end up with threading added as an optimization, less comprehensive and more likely to introduce bugs. If you skip the back end of the process, you end up with an under-tested, under-performing implementation. View each round of parallel performance enhancement as a continuous task that runs through a release cycle, rather than a feature that can be introduced late or back-burnered under schedule pressure.

In the Field: Commercial Software

As we did with HPC earlier in this series, let's relate some of these points to the development of a few actual business software products. We'll discuss two projects, one a revision to a voice communication and collaboration application that is now complete, and the other the ongoing threading of a large desktop productivity application. Intel engineers played, or are playing, a consulting role on these development efforts and they provided us with these brief case studies.

The voice application had an unusual goal. Instead of using multiple cores to better performance, developers sought to distribute the workload among cores as evenly as possible. With more even core utilization, CPU clock frequency could be dropped, conserving power.

The project team included two company engineers and one Intel engineer. At the start of the project, the application was functionally threaded only, and core utilization was uneven. The discovery step began with a review of the entire system, from the driver level up, to find opportunities for parallelism. Engineers settled on applying data decomposition techniques to the software's audio codec as the best approach.

In the design, pooled threads both encode and decode audio data. The number of threads in the pool is proportional to the number of cores, for even loading. The main thread pulls threads from the pool and uses them in the codec as needed, returning them to the pool when they complete their task. Each thread runs at the same priority, again for most even distribution of active threads among cores.

As the expression step began and developers began to thread the codec, they found that a third-party library they had been using was not thread-safe. Rather than replace the entire library, which is used throughout the application, developers chose to build thread-safe replacements for only the library functions that were used in the threaded sections. That complete, developers found only a few synchronization bugs as they began to test for confidence.

Although the voice project was focused on core utilization and not application performance, developers proceeded to an optimization step after verifying that the threaded codec was working properly. They focused particularly on serial optimization within the threaded section, since that would reduce maximum core utilization. The team used Intel's VTune and Thread Profiler tools in tuning.

Managing Multi-Core Projects

The second project-at a very different stage- is the threading of a large desktop productivity application. In this project, still in the discovery step, engineers are unraveling an inefficient threaded implementation before they can begin to look for better, more natural opportunities for both functional and data parallelism. This amounts to forensic work on the application, using VTune and a debugger to find dependencies and to map out the work done by the main thread.

The desktop application has close to 40 threads, but the main thread is doing over 95% of the work. Engineers need to first address some basic performance issues, such as reducing the time spent in system calls, before moving on to discover new opportunities for threading. The initial threading plan is to decompose the main thread, almost starting from the same point as one might with a single-threaded application.

It's hard to say how the desktop application project will proceed, but the rough schedule is for the three engineers working on the project to spend three months in further discovery (while simultaneously working on basic performance) and then the following three months on the threaded implementation.

The Development Manager's Role

In their 1937 Papers on the Science of Administration, Gulick and Urwick described a manager's role in terms of seven activities. Their's is a seminal work, one that helped to define the discipline of management. In this series, we've focused quite a bit on how multi-core changes development process and development practice. To wrap up this installment, let's take a look at multi-core from another perspective, focusing instead on how the development of parallel software changes the manager's job. We'll break it down according to the seven functions provided by Gulick and Urwick.

Planning. In planning a project, consider how parallelism will change your development process. Take an integrated approach to parallel programming that runs through every phase of development.

Organizing. Organize the team according to the roles we've discussed earlier in this series. Include a lead architect that understands threading, and application experts that understand the problem domain. Include testers that can verify that parallel code works on the range of target hardware, varying the number and speed of processing cores. **Staffing.** Start with application experts and train them in parallel techniques. Attempt to develop parallel skills throughout the development team. As a goal, the overall team's expertise should be 80% in the application domain and 20% in parallelism.

Directing. Guide the project through your development process, as modified to accommodate multi-core. Know the exit signals from each step of the process. As in the voice application case study, know when changing conditions (a library that's not thread safe) require a change to the implementation (the development of thread-safe replacements).

Coordinating. For large applications, you'll need to carefully coordinate the threading team's changes with those of other teams making functional or architectural changes. If you don't have internal threading expertise, coordinate your team's efforts so that they follow the technology lead of experts in your application area.

Reporting. Parallel programming is seen as a performance activity, so you'll often be targeting a performance metric, but correctness is always important. Parallelism adds additional testing requirements. Track whether bugs are due to parallel interactions or present in a single thread.

Budgeting. Budget for staff requirements discussed above. Consider the tradeoff between the software costs of buying threading components vs. the development costs of building parallelism into the application itself. The best budgeting news is on the hardware side, where multi-core is the new mainstream for desktop and laptop systems. That means that even the natural obsolescence of development and test machines will put more and more developers and testers onto the target multi-core platform.

Managing Multi-Core Projects

Multi-core is the new mainstream for more than business customers-it's time to start putting those cores to good use in consumer software, too. In our next installment, we'll look at the management issues around the development of multi-core games and other consumer software projects.

About the author: Steve Apiki is senior developer at Appropriate Solutions, Inc., a Peterborough, NH consulting firm that builds server-based software solutions for a wide variety of platforms using an equally wide variety of tools. Steve has been writing about software and technology for over 15 years.

Part 5: Desktop Software Considerations

Programming for parallelism is something new in desktop software, and multi-core systems are wide-open territory for performance improvement. Make the most of the opportunity by carefully choosing your approach to threading legacy applications, and by building the right kind of team for parallel programming.

by Steve Apiki

Think about parallelism in desktop software, and you should be thinking small-small systems, of course, but also the small intervals of time you can run parallel between user interactions, a small number of cores (for now), and, in most cases, a small threaded codebase. Some of these "small" factors are the reasons that multi-core processors will make an even more decisive impact on client and consumer applications than they have on server software. On desktop and laptop systems, the move to parallel software is just getting underway, so the opportunity to differentiate with parallel performance has never been greater. And if you can keep it scalable, your application can continue to dazzle users as these machines track new processor generations with more cores per package.

From a half-empty perspective, you could say the opportunity for parallel improvement is so great because desktop teams are starting with so little. In comparison to HPC and server software, few client or consumer applications are threaded. Threaded desktop applications are most commonly threaded by task, with tasks such as communicating with a long-running server process, or loading graphics textures, or handling I/O, relegated to a background thread to keep the main thread from blocking. That's not the kind of threading, the threading of compute intensive operations, that's the key to scalable performance on multi-core systems.

Before multi-core, there was little incentive for data parallelism on the desktop. Now it's critical if you intend to scale out to more cores, not only in obvious application categories like games and graphic effects, but in thick client and productivity packages as well. As the development manager, how do you guide your team through that transition? In these last two segments of our management series, we'll discuss approaches to introducing parallelism, and some of the factors that make the desktop so different from parallel HPC and server configurations.

Finding the Right Targets

Let's look at a single-threaded legacy application from two perspectives, code-centric and data-centric. You can find opportunities for threading from both perspectives-but each also imposes its own constraints on how threading changes can be made.

From a code-centric perspective, an application looks like a stack of modules, core modules on the bottom, application-specific modules that make use of core services on top. The natural break for parallelism occurs at different levels in this stack for different applications. A couple of examples may make this more concrete.

In a game, the first level of parallelism is very coarse. You've got critters running around, you want each critter to operate in parallel. You can thread a game like this at a high level, if not one thread per critter, then perhaps one thread for a group of interacting critters. In any case, the change occurs far away from the core modules, up in game-specific logic.

On the other hand, you won't find such high-level parallelism in a media player-you're only playing back one DVD, after all. Effects are instead better handled in a pipeline, with a thread per stage. Unlike the game, the natural place to thread the media player is at a very low level, down in the video codec itself.

The level at which parallelism naturally occurs will determine how you can best introduce threading. If there's high-level parallelism, you can thread at that level with little direct impact on the rest of the application. (There may be an indirect impact, though, if you find that underlying modules are not thread-safe and would need to be made thread-

Managing Multi-Core Projects

safe or replaced with thread-safe versions.) If parallelism is at a low level, and you need to thread a core module, the direct cost of the change may be greater. The benefits of threading the application must be measured against that cost.

Similarly, from a data-centric perspective, threading changes must be measured against how they affect the data model. Rearranging the data model to accommodate more parallel structure may be a high-cost choice, or it may simply not be an option in a legacy system. However, it may be possible to thread the application without modifying the data model, if application code can be refactored so that the threaded sections work only with isolated portions of the data. In this case, the data model can be partitioned into threaded-access and non-threaded access segments.

One or Two Experts

As we've discussed before in this series, the ideal parallel software development team has parallel experience at every position, from the software architect to the test engineer. It's hard to get near that ideal in client and consumer software, where threading veterans are hard to find. Instead, most desktop development teams must rely on the advice and experience of one or two threading experts as they make the move to threaded development.

A common practice when there's a large team and a few threading experts is to give the experts global access to the code and simply have them make all thread-related changes. There are two problems with this approach. First, threading experts can't have deep experience with most modules, yet they are expected to make changes with deep impact. Second, it's difficult to coordinate thread-related and feature-related or functional changes when both must be done in the same revision cycle, which is often the case.

A better first place to cultivate parallel expertise is in the application expert, the member of the technical team with the greatest understanding of the application domain and user expectations. This position may have different titles in different companies, but the intent is to move parallel understanding to the team member that first translates new creative ideas into technical designs.

By making the application expert the threading expert, you are more likely to get the parallelism model right, to get threading applied at the right level, and have the least impact on the data model. With limited expertise to go around, that is a better option than focusing experts on threaded implementation at the expense of design.

Digital Content Creation and the Threading Process

Digital Content Creation (DCC) software includes tools for 3D modeling, rendering, graphic effects, and animation. DCC software demands performance, both to respond quickly to user input and to cut run times on effects generation and high-quality renders. Intel application engineers have worked with several software vendors in this category to thread their applications for effective multi-core performance.

A typical experience is illustrated by one such vendor who has enjoyed significant success in threading portions of its DCC application since kicking off its multi-core threading effort nearly two years ago. Its product is an integrated package with a number of modules, and the development team's choices of where to introduce threading (and where to leave it out) are especially interesting.

The team's first target was a partial differential equation solver used in simulations for graphics effects. The solver runs equations across large array data and is a natural for threading through domain decomposition. The solver was an ideal place to introduce threading, not only because of its readily-parallelized data, but because of its limited impact on the core data model. It took two person-months to thread this first module.

With new releases of the product, additional sections have been threaded. But some key elements have not been (and may never be) threaded. These are elements that touch a great many nodes in the data model. Any modification would have implications throughout the system, and the development team decided that the benefits of thread-

Managing Multi-Core Projects

ing these modules, at least to this point, weren't worth the impact on the product.

The team, with Intel engineers in a consulting role, followed a pattern of slow and careful introduction of parallelism, in actual threading implementation as well as in choosing what to thread. They found that OpenMP was an excellent way to quickly prototype parallel changes. Since OpenMP is so easy to turn on and off, it also provided an easy means to compare parallel performance to that of the serial implementation.

When implementing multiple pipelines, engineers prototyped parallel sections with OpenMP and later went back and re-coded using Intel TBB. In this case, Intel TBB gave an added performance edge that the company's engineers decided was worth the additional coding effort. TBB is still relatively new, and the team expects to be able to use TBB for prototyping as it becomes more established.

Introducing threading is a significant challenge, but it's one that can be met with a careful, pragmatic approach. The first steps are choosing the right level for threading and educating your team. In our next installment we'll look at some additional multi-core desktop issues, including changes to the development cycle, and some design considerations unique to client and consumer software.

About the author: Steve Apiki is senior developer at Appropriate Solutions, Inc., a Peterborough, NH consulting firm that builds server-based software solutions for a wide variety of platforms using an equally wide variety of tools. Steve has been writing about software and technology for over 15 years.

Part 6: Diving Deeper on Consumer Software

Threading raises some not-so-obvious design issues in desktop software, issues that might have been overlooked before the era of multi-core. Part 6 concludes the series with a discussion of these issues, and with a closer look at how the threading process relates to development methodology.

by Steve Apiki

This final installment in our multi-core management series is all about diving deeper. We've discussed Intel Developer Products Division's (DPD) steps for threaded development before, but here we'll go further into how we might adapt these steps for multi-core development to a specific development methodology. Then, returning our focus to consumer and client software, we'll get into some of the architectural and design issues of threading desktop applications.

By way of review, Intel DPD has a four-step design and development process for parallel programs. Intel DPD's process has emerged from the years of experience the group has in threading HPC, visualization, and other customer applications. Briefly, the four steps are to discover the natural mode of parallelism for an application, to express that mode as a parallel programming model, to gain confidence that the model is effective, and finally to optimize for performance.

The four-step process is general enough that it can be applied within any programming system. It's a way to emphasize the importance of the parallelism model and the importance of continuous revision in the creation of threaded programs.

Let's take a look at how these four steps might work within Extreme Programming (XP). Even if you're in an XP shop, your team may follow some, but not all XP practices, and not all XP practices are going to be sensitive to parallel vs. serial development (for example, keeping a sustainable pace is unrelated to the type of code you are creating). What follows are selected XP practices with notes on how they relate to Intel DPD's four-step process. The items in this list are adapted from the Rules and Practices page at ExtremeProgramming.org.

Small Releases: XP breaks larger projects down into deliverable iterations that can be completed in a short period of time. Keep the scope of threading changes small by focusing on single modules if possible. In that way, programmers can run through each step within a single XP iteration.

Spike Solutions: Spike solutions are small, independent programs written to explore solutions outside the main body of code. These may be used as part of the discover step to test approaches to parallel decomposition. The "parallel design patterns" you develop through spike solutions are important. They help in understanding what sections are scalable in estimating performance improvements using Amdahl's law. They can also help outline where synchronization is needed, and what synchronization methods are appropriate. Often, the best way to improve parallel performance is to restructure the data or algorithm to reduce synchronization.

No Early Functionality: The natural mode of parallelism may vary from module to module. In the express step, programmers should focus on proper implementation of the parallel model, not on performance optimization or general-purpose routines.

Refactor: The parallel development process is iterative. Refactor the parallel model when appropriate, moving back to further discovery when you see potential for improvement later on. Refactoring to avoid synchronization is a good example. In a larger sense, "refactoring" may apply to the threading of serial functions. That larger refactorization from serial to parallel would include working through all four steps.

Unit Test First: In XP, initial unit tests are created with the code. This means either in discovery, to test the scalability

Managing Multi-Core Projects

of the parallel model, or as part of the express step, to ensure that threaded code properly represents the model. Additional tests will be needed to find threading conflicts as you test for confidence.

Optimize Last: Both XP and Intel DPD's four steps advise against premature optimization. Only after testing has developed confidence in the model should you move on to tuning and optimization.

XP and discover, express, confidence, and optimize share the central idea that development is an iterative process. In building parallel software, it's essential that you get the parallel model right. Ideally this happens in discovery, but the model can be adjusted later on in the process. Use the four-step process within an XP iteration to deliver threaded changes built on a tested parallel model. Further refine and refactor the parallel model in subsequent XP iterations.

Architectural Options

Desktop applications are unlike server software and typical HPC programs in lots of ways, but the critical difference for our purposes is that desktop applications must be interactive. There's a user on the other side of that monitor and keyboard, and users like to see real-time responses. In an interactive application, you have only short intervals in which you can run parallel before you need to synchronize threads to update the user's view. The shorter the interval, the more responsive your application will appear, but the less time it can spend in pure parallel sections.

Games and animation packages will generally need to synchronize once per frame, for two reasons. First, the view needs to be in a consistent state at the start of each frame so that it is ready for display. Second, both OpenGL and DirectX require that only a single thread access each library. With a frame rate of 60-100 FPS, the parallel intervals are short, and synchronization overhead may become significant. Contrast this artificial synchronization requirement to that of a long-running HPC process, where threads or nodes are only synchronized as the algorithm requires.

You may not need to synchronize once per frame in a productivity package, but you've still got to be responsive. We know anecdotally that users prefer longer execution time with feedback to absolute minimum wall-clock time-to-solution. Sometimes it will be necessary to synchronize threads in order to provide that feedback on screen, even at the expense of greater parallelism.

The synchronization requirement limits your architectural options in designing the parallel model for desktop systems. One option is data parallelism, where data is split across several threads during a parallel section but all data threads rejoin the main thread at the end of each interval (each frame, for a game). Then the main thread synchronizes with a rendering thread to update the display. This approach is relatively straightforward, and it's a natural fit both for OpenMP's fork-join model and for Intel Threading Building Blocks (TBB).

Data parallelism is effective on dual-core systems, but performance improvements begin to level out on more than two cores. With data parallelism, all data threads must wait on the slowest data thread as they synchronize at the end of each interval, limiting scalability.

For games, or other performance-critical software, domain decomposition provides a more scalable alternative. In domain decomposition, related tasks are grouped into domains ("related" so that communication, and therefore synchronization, is minimized between domains). The application repartitions data across threads for load-balancing at the start, and periodically after that, but not with every frame. Domain decomposition eliminates synchronization between data threads and the main thread, leaving only the rendering thread synchronization to be completed for each frame.

Design for Threading

Design new code for threading. That's not the same as saying that new code should be threaded wherever possible-while in many cases that might be a good idea, there are cases where it's not. If you're fielding a new, complex algorithm, it's probably better to make sure the serial version works as expected before attempting to derive a parallel implementation.

Managing Multi-Core Projects

But, as we've said before, parallelism is a design consideration, not a performance optimization. So, whether code is threaded or not, design it to be threadable. For example, partition modules so that tasks with frequent shared-memory communication are in a single module, and minimize communication between modules.

It's often the case that new features or lead customer requirements are compute-intensive and anticipate the latest hardware. This may help to raise the importance of considering threading issues in design, among both developers and customers.

If your desktop application supports plug-ins for user customizations, you know that plug-in compatibility puts a constraint on changes you can make to your software, both internally and to the plug-in API. Multi-core adds another consideration: plug-ins may themselves be threaded, and plug-in threads plus internal threads makes oversubscription more likely.

Consider exposing thread management as part of your plug-in API. That way, you can ensure that both application and plug-in threads come from a common pool of the appropriate size. You'll also make threading easier for plug-in developers.

Take an evolutionary approach to adding thread support. Start with the basics, including just a `GetThread` call, a `ReleaseThread`, and a critical section. You can follow that with advanced capabilities and other synchronization objects in later releases.

Multi-Core and Management

There are a few points that we've made throughout this series that apply as well to HPC and server software as they do to client and consumer applications. No matter what kind of development project you're steering, you can take best advantage of the power of multi-core if you keep these items in mind:

Develop Parallel Skills Throughout the Team: Parallelism needs to be considered in every phase of development. The team is made much more effective when each member has the right amount of parallel expertise for his or her role. Take an Incremental Approach: Discover the natural parallel model for your project, then incrementally improve the model as you test and gain experience.

Consider Libraries and Tools: Don't underestimate the importance of libraries, debuggers, and optimization tools. These are especially valuable as your team makes the transition from single-threaded to multi-threaded, and as you thread legacy applications.

Parallelism is Design, not Optimization: It's most important that you get the parallel model right, for performance, for scalability, and to reduce conflicts. Threading is too fundamental a change to be handled as an optimization.

About the author: Steve Apiki is senior developer at Appropriate Solutions, Inc., a Peterborough, NH consulting firm that builds server-based software solutions for a wide variety of platforms using an equally wide variety of tools. Steve has been writing about software and technology for over 15 years.