White Paper

How Do Parallel Programmers Think?

By Ken Strandberg

I visited a software forum recently, and one of the topics revolved around "Do parallel programmers think differently from sequential programmers? Do they use different models? Do they approach their work differently?" I had the privilege of interviewing three parallel programmers at Intel with 60 years of combined parallel programming experience to answer those questions and offer a glimpse into the mind of a parallel programmer.

Introduction

Parallel programming is a well-established science with a large body of knowledge and a library of literature. It is also the future direction of programming required for applications to take full advantage of the multi-core and many-core chips Intel knows it can build. Yet, to many programmers, both those just beginning their careers and experienced sequential programmers, how expert parallel programmers create their applications, while not a mystery, is certainly foreign. Yet it doesn't have to be.

Three of Intel's expert parallel programmers generously gave me some of their busy time to talk about thinking like a parallel programmer. Their short bios are at the end of this article. And, those paragraphs don't begin to give justice to their knowledge, experience, and the roads they took to acquire it. They're all seasoned experts. Two are pioneers in parallel programming, each coming into it from different angles and backgrounds at the beginning of parallel computing. The bottom line is, while all these experts haven't seen it all, they've seen a lot, and that's shaped the way they think about parallel programming.

Certainly, there's plenty of literature on the issues of parallel computing in a shared memory model: variables, race conditions, deadlocks, etc. And there's good information on the techniques of writing parallel code: private variables, synchronization, etc. We also have good, well-established notations and APIs, like Intel® Threading Building Blocks, OpenMP*, Windows* Threads and Pthreads*. If you understand the issues, have introduced yourself to the techniques, and played with the notation, what else is there? Plenty.

Parallelism at the Core

Obviously, parallel programs are different from sequential ones. For today's programmer, everybody pretty much starts with serial codes. With sequential programs, there's a clear flow of program control – instruction 1 executes, then instruction 2, then instruction 3, and so on. You can always depend on knowing what's happening when, and therefore, know how the data is being accessed and changed, and clearly understand the dependencies. "In parallel development that gets thrown out the window," states Henry Gabb. "You now have a temporal component to consider. You have to think about multiple instructions executing simultaneously and what that does to your data structures, your variables, your algorithms. Everything."

Tim Mattson describes it in terms of a single-threaded versus multi-threaded reading of code. In one, you clearly see the control flow. In the other, multiple, different flows can be executing at any time. "There are multiple states occurring and changing. Dependencies change. You have to keep this all straight in your head." This change, this temporal component, where a number of control flows are executing simultaneously is "at the core of the change you to have to go through intellectually in understanding a parallel program," says Tim.

Thus, your model has to change. Henry advises, "The key thing to look at is data access. You have to be very careful about data corruption. There is no set order in which threads will execute and when they will access data. The operating system schedules the threads, and it doesn't know anything about data access patterns. It just schedules threads as it sees fit. The only order in a parallel program is what the programmer explicitly creates, using synchronization. But then, the program is no longer parallel at synchronization points."

Though parallel programming is not a paradigm shift in the classical sense, there is a definite shift in thinking. How difficult that shift can be depends on where you start. "I think it's not as big a leap if you do shared-memory parallel programming first versus distributed-memory applications," says Clay Breshears. "If you've got one thread, and you can consider things being shared, adding another thread and envisioning two workers dividing the work is not a huge leap. It's just different, with different issues to watch out for."

Parallelize That!

There's no question that parallelizing applications is paramount for future performance out of today's codes. But which codes should be parallelized? For Clay Breshears, the answer is often pretty simple: "Usually when a project comes to me, my manager says 'parallelize that'. Of course, when you get into it, it might not be as simple as it sounds."

The core of any parallel decision comes from the ability to get enough concurrency out of the problem to make it worthwhile to parallelize it, and whether that's enough for the customer. Amdahl's law is the guiding principle – or law – here. When Henry gets a project, he looks for reasons to disqualify it. The project must pass three tests before he considers there's enough value in parallelizing to go through the work. "I start with Amdahl's Law and customer

requirements. If I think I can get enough theoretical speedup, and it's enough for the customer, then I look at load balance. How evenly can the parallel work be divided among threads? If the load balance is reasonable, then I consider the granularity. Is there enough work per thread to justify the cost of creating the thread in the first place? Any one of these tests can disqualify a project. If it passes all the tests, then it's a candidate for parallelization."

All three of these professionals approach projects somewhat differently. Henry first looks at the profile: how much of the code is absolutely sequential. If it's too much, then the serial factor is too high and there's not enough theoretical speedup to make it worthwhile to parallelize. Again, it comes down to Amdahl's Law here.

Tim relies on his many years experience in scientific computing. "Typically with the problems I work on, looking at the problem, I can figure out what the program is doing, where the concurrency is, and where the compute intensive portions of the code are. If I can't parallelize those portions, there's no sense in continuing." He offers an example of an algorithm doing 3D prestack depth migration used in seismic applications found in oil and gas exploration. "I know it's going to have a big FFT. I know it's going to consume a lot of time, because I know the mathematics of an FFT. And I know the structure of the data as it loads and processes the collections of records. So, I can deduce where I need to parallelize the code. But I can get away with that, because, for the problems I tend to work on, this is all well defined mathematically, and I know the mathematics."

But what if you're handed an application you know nothing about. That's the situation with many application engineers when an ISV says, "parallelize that!"

"In all cases, you have to understand the algorithm that you're trying to parallelize," says Clay. "Someone needs to walk you through it. That's usually the customer. If you don't, you're just reading code. Sure, you'll eventually figure it out, but it'll take longer. Then, if you understand parallelism, you can look for independent operations and visualize where to parallelize."

Seeing Parallel

The overall parallel programming process – identifying the concurrency, exposing the concurrency, and exploiting it in source code – is the same for all parallel programmers, according to Tim Mattson. But, like anything, it's a human activity, with individual nuances in the approach. Henry relies on the Intel threading tools; Except for Intel® Thread Checker, Tim typically doesn't.

Clay has found that visualization helps him organize his code. "I think in terms of people and work that can be shared when parallelizing in a shared-memory model. If I can say 'here is a pile of things and here are the tasks,' and I can visualize two people coming in and dividing up the work, whether or not they agree on how the division goes or the methods they use; then I can come up with algorithms that solve the problem. And, if I can see it for two people, then I can visualize it for four, eight, sixteen, or thirty-two." At each division of the work, Clay has to ask how the additional workers affect the rest of his scenario. Does each worker have enough work? Is there contention for the same information? Can he instruct people to work more independently and share information occasionally, as opposed to constantly? When programming for a distributed-memory environment, instead of people and work, he sees mailboxes and mailmen passing messages around.

Once Clay gets his code threaded and sees it running, he uses tools to optimize it, including Intel® Thread Checker, Intel® Thread Profiler, Intel® Cluster Tools, and TotalView® Debugger for MPI from TotalView Technologies.

Parallel Patterns

The original question for this article was, 'How do parallel programmers think?' According to Tim Mattson, computer scientists really don't know much about the cognitive psychology behind programming – parallel or serial. They have largely ignored this field. Fortunately, cognitive psychologists have looked at this problem, having used programming to study human cognition in general. From these efforts, and a handful of collaborations with computer scientists, a high level picture of the cognitive psychology of programming is emerging.

The human mind is an abstraction engine. We build models; and, in a problem that demands our immediate attention, we make features explicit, while hiding the rest behind a high level abstraction. You can see this when working with concurrency. "Keeping all concurrent threads straight in your mind is overwhelming," says Tim. "So the programmer creates simpler and more restricted structures to constrain concurrency in a problem. These structures become the recurring parallel patterns for exploiting concurrency." Tim wrote a book about parallel patterns and is currently furthering this work to a larger range of software engineering problems. "What happens with programmers is over time they build up a catalog of solutions in their heads. Psychologists call these *plans*; in computer science, we call them *patterns*. As an expert programmer continually looks at a problem, he acquires enough information about the problem to match a pattern to a part of the problem. At that point, he applies the plan to progress toward a solution."

This pattern matching step, according to Tim, is part of what he calls wetware, the intellectual work of understanding the problem, identifying the concurrency, and reorganizing the problem to expose that concurrency. "The difference between an expert parallel programmer and a novice is the expert has a larger, richer catalog of plans – or patterns – in his head."

Every expert parallel programmer does this wetware before starting the step of notation that exploits the concurrency. Novice programmers who skip this part of the work often end up working overtime or don't end up with correctly working code. "By the time I introduce threads," states Henry, "I've done a lot of groundwork. I've seen lots of cases where an engineer gets the code, profiles it, and then starts threading without regard for the underlying algorithms. If you go about it that way and get a scalable parallel code, it's blind luck."

According to all three of my interviewees, part of the wetware work is figuring out whether the serial algorithm handed to you is the best one for parallelization. When a project begins failing Henry's tests for qualification, he considers alternatives with the customer to achieve the desired results with a different, more "parallel" algorithm. Clay does the same thing when he sees things like the data can't be decomposed effectively, or there's too much contention for the same variable and there's no obvious solution. "That's when I consider an alternative algorithm that might not be as efficient serially, but parallelizes and scales easier with threads."

Some Sound Advice

As these three experts have expressed, parallel programming, for the sequential programmer, is new territory. There are many difficulties the novice parallel programmer faces, according to Henry. "Inexperience with parallelism in general. The serial algorithm the programmer spent so much time optimizing might not be the best parallel algorithm. The fact that they're parallelizing legacy code that wasn't designed for parallelism in the first place."

Tim believes the biggest challenge is the wetware – where programmers are going to find the concurrency in the problem – and building up their catalog of plans. "This is a large body of

knowledge that takes time to acquire and apply. They need to get comfortable with the basic terminology and basic classes of algorithms, understand the patterns, grasp the challenges. Notation is the easy part. Understanding the parallel algorithm is the hard part."

But, they all agree the path to parallel programming is well marked. And they have some suggestions of their own to novice parallel programmers to get started.

"There are a lot of good books out there," says Henry. "I'd recommend starting with the OpenMP specification. It's not very long, it catches all the relevant issues in parallel computing, and it has a lot of examples." Clay suggests that OpenMP is also the best place to start, assuming the programmer has an existing knowledge with C, FORTRAN, or C++. Henry cautions though that OpenMP isn't enough. "OpenMP is not broadly applicable. You're going to run into problems down the road if you try to make it fit into every problem."

Tim says to buy his book. Not because he's the author, but because it's the only book available that focuses primarily on the recurring patterns that experienced parallel programmers use over and over again. "You have to build up the body of knowledge; make it yours. That's what my book is about. And it provides examples in MPI, OpenMP and Java* threads. Seeing how a single solution is handled with three different notations is a valuable way to learn the ideas behind a solution."

Conclusion

Parallel programming is the future of programming. It's going to take some time for today's programmers to become expert and mature parallel programmers. Tim and Clay agree that there's a lot more information on parallel programming today than when they started out in the 1980s. It's a mature science. And it has its naysayers and evangelists.

Clay offers the following: "Don't be put off by all the doom and gloom talk. Programming is an art. And parallel programming is another art. You learn the tools, but it ultimately comes down to practice and intuitively doing things on your own." Henry reminds us, "Nobody is going to learn parallel programming without having an application in mind. You can read all the literature out there, but until you apply it, you're never going to actually learn."

The art of parallel programming is best learned by doing.

Understanding Granularity

Granularity is a term used in different contexts in parallel programming. It is somewhat subjective. In this context, granularity can be loosely defined as the ratio of computation to synchronization. For Henry Gabb, parallel granularity is one of his tests to disqualify a potential threading project. If he can't find enough parallel work to justify thread creation, then he disqualifies the project.

He looks at the level of threading to make sure it's at the right place in the call tree. "That's a critical part of the process. If you thread too low in the call tree, you can end up hurting performance. A colleague once worked on a program where the profile showed 99 percent of the runtime in a single function. He immediately threaded the main loop in this function then wondered why the parallel code performed worse than the original serial code. Unfortunately, this function was called millions of times and the individual calls didn't do enough work to justify the overhead from threading. He should have added the threads one level higher in the call tree."

In synchronization, if you lock too coarsely, you can end up blocking too many threads accessing necessary data to gain the intended performance. Henry pays close attention to synchronization while designing a parallel program. "I worked on a project where the customer's application used a tree structure," Henry describes. "Whenever a thread read or modified a leaf in the tree, it locked the entire tree. They asked us, 'What are we doing wrong?' First, locking the entire tree serialized far too much of the application, so we changed the lock granularity to branches and even individual leaves. Next, we discovered that 90 percent of the threads were only reading the tree rather than writing to it. Since there's no conflict when multiple threads read the same data, we created reader/writer locks to minimize contention. These basic modifications improved performance tremendously. In fact, these modifications are so well-known that they're addressed in just about every book on threading."

About the Parallel Programmers

Henry Gabb is a Principal Engineer in the Intel Software and Services Group. He has been working on parallel applications and parallel performance issues since joining Intel in 2000. Prior to joining Intel, Henry was Director of Scientific Computing at the U.S. Army Engineer Research and Development Center MSRC, a Department of Defense high-performance computing site. Henry holds a B.S. in biochemistry from Louisiana State University and a Ph.D. in molecular genetics from the University of Alabama Birmingham School of Medicine. He has several peer-reviewed research papers on high-performance computing and the computational life sciences.

Henry Gabb has written several articles useful to parallel programmers.

Gabb and Kakulavarapu (Editors), Intel Software Network, "Developing Multithreaded Applications: A Platform Consistent Approach" http://cache-www.intel.com/cd/00/00/05/15/51534_developing_multithreaded_applications.pdf

Gabb, *Linux Magazine*, "Common Concurrent Programming Errors" http://www.linux-mag.com/id/996

Gabb and Magro, *Dr. Dobb's Journal*, "Faster Image Processing with OpenMP" http://www.ddj.com/architect/184405586

Gabb and Lake, *Gamasutra*, "Threading 3D Game Engine Basics" http://www.gamasutra.com/features/20051117/gabb_01.shtml

Clay Breshears is currently a Course Architect for the Intel Software College, specializing in multi-core and multithreaded programming and training. He received his Ph.D. in Computer Science from the University of Tennessee,

Knoxville, in 1996, but has been involved with parallel computation and programming for over twenty years. Six of those years were spent in academia. Clay started his tenure at Intel as a Senior Parallel Application Engineer at the Intel Parallel Applications Center in Champaign, IL, implementing multithreaded and distributed solutions in customer applications. He can be reached at clay.breshears@intel.com.

Timothy Mattson is a Principal Engineer at Intel working in the Microprocessor Technology laboratory. He started writing parallel software in 1985 during a post-doc at the California Institute of Technology, while working with the Cosmic Cube (one of the first hypercube parallel computers). Since then he has used parallel computers to make chemicals react, shake up proteins, find oil, understand genes, and solve many other scientific problems. Tim's long term research goal is to make sequential software rare by bringing today's sequential programmers into the realm of parallel development.

Tim has published over 40 papers on different topics in parallel computing, including his most recent book, *Patterns for Parallel Programming* authored with Beverly A. Sanders and Berna L. Massingill. It's published by Addison-Wesley.

About the Author

Ken Strandberg writes technical articles, white papers, seminars, and a host of other content for trade publications, emerging technology companies, Fortune 100 enterprises, and multi-national corporations. He writes about Software, Industrial Technologies, Design Automation, Networking, Medical Technologies, Semiconductor, and Telecom. Mr. Strandberg can be reached at ken@kenstrandberg.com.