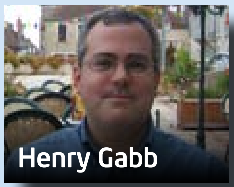
THINKING IN PARALLEL

THREE ENGINEERS' VIEWPOINTS

Effective use of parallelism is the key to achieving application performance as we move toward an era of many-core processors. To share their hard-won expertise with a new generation of coders, three seasoned Intel engineers—with deep experience in parallel programming—share what they've learned about the craft.

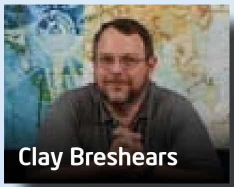
Tapping into the Core of Parallelism

Traditional programming is steeped in its sequential approach; this has to change for the parallel programmer. With serial code development, there is a clear flow of program control. The coder knows how the data is being accessed and changed and understands the dependencies. "In parallel development that gets thrown out the window," said Henry Gabb, a principal engineer in the Intel Software and Services Group. "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."



In the same vein, Tim Mattson, principal engineer at Intel working in the Microprocessor Technology laboratory, said, "There are multiple states occurring and changing. Dependencies change. You have to keep this all straight in your head." In Tim's judgment, this aspect is "at the core of the change you have to go through intellectually in understanding a parallel program."

Change your model, Henry advised. "The key thing to look at is data access," he said. "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. The only order in a parallel program is what the programmer explicitly creates using synchronization."



"I think it's not as big a leap if you do shared-memory parallel programming first versus distributed-memory applications," added Clay Breshears, a course architect for the Intel® Software College. "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."

Parallelize That!

Choosing appropriate code to parallelize can be a challenge. For Clay, the decision is sometimes 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."

When is it worth the effort to parallelize a program? Henry starts with Amdahl's Law and the customer's requirements. "If I think I can get enough theoretical speedup," he said, "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."

Tim relies on his many years of experience in scientific computing. "Typically with the problems I work on, 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."

An algorithm for doing 3D prestack depth migration provides a fitting example. "I know it's going to have a big fast Fourier transform (FFT)," Tim said. "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."

If you're given an unfamiliar application, you still need to understand the algorithm you're trying to parallelize. "Someone needs to walk you through it," Clay said. "Then, if you understand parallelism, you can look for independent operations and visualize where to parallelize."

Exploring Individual Programming Paths

Individual programmers often take a different tack when approaching parallel programming. For example, Henry relies on the Intel® threading tools. Tim typically doesn't (except for Intel® Thread Checker).

Clay employs his own unique approach. "I think in terms of people and work that can be shared when parallelizing in a shared-memory model," Clay said. "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, 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, 16, or 32."

Once Clay gets his code threaded and sees it running, he focuses on optimizing it, using primarily Intel Thread Checker, Intel® Thread Profiler, and Intel® Cluster Tools.

Recognizing Parallel Patterns

"Keeping all concurrent threads straight in your mind is overwhelming," Tim said. "So the programmer creates simpler and more restricted structures to constrain concurrency. These structures become the recurring parallel patterns for exploiting concurrency." This theme is amplified in Tim's book, *Patterns for Parallel Programming*.

"Keeping all concurrent threads straight in your mind is overwhelming, so the programmer creates simpler and more restricted structures to constrain concurrency."

Tim Mattson, Principal Engineer at Intel,
Microprocessor Technology Laboratory

"Over time," Tim said, "programmers build up a catalog of solutions in their heads. 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. The difference between an expert parallel programmer and a novice is the expert has a larger, richer catalog of patterns in his head."

Expert parallel programmers first break down the problem, find and expose concurrency, and then start the notation that exploits the concurrency. Novice programmers who skip the first parts often end up working overtime or don't end up with correctly working code. "By the time I introduce threads," Henry stated, "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."

Avoiding Programming Pitfalls

There are many potential pitfalls for the novice parallel programmer. One of these, Henry said, is: "The serial algorithm the programmer spent so much time optimizing might not be the best parallel algorithm."

Tim sees the lack of experience with what he calls wetware (the intellectual work of understanding the problem and exposing the concurrency) as a significant challenge. "This is a large body of knowledge that takes time to acquire and apply," he said. "Programmers 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." He recommends his book as a practical means to gain familiarity with parallel patterns and how to be effective with them.

"I'd recommend starting with the OpenMP* specification," Henry said. "It's not very long, it catches all the relevant issues in parallel computing, and it has a lot of examples." Clay also recommends OpenMP as the best place to start. Whatever the starting point, however, novice parallel programmers need to actively apply their growing skills to a broad range of concurrency challenges to begin recognizing the tell-tale patterns—where parallelism yields performance or efficiency dividends.

Mastering the Art

What is the best path to parallel programming success? "Programming is an art," Clay said. "And parallel programming is another art. You learn the tools, but it ultimately comes down to practice and intuitively doing things on your own."

"Nobody is going to learn parallel programming without having an application in mind," Henry asserted. "You can read all the literature out there, but until you apply it, you're never going to actually learn."

On one point, these seasoned engineers all agree: the art of parallel programming is best learned by doing. ■

ABOUT THE INTEL ENGINEERS

Henry Gabb is a principal engineer in the Intel Software and Services Group. He has been working on parallel applications for 15 years. 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.

Clay Breshears is currently a course architect for the Intel Software College, specializing in multi-core and multi-threaded 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 20 years.

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 solve many scientific problems.

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. Mr. Strandberg can be reached at ken@kenstrandberg.com.