INFO TO GO

- Failures can lurk in incredibly obscure places.
- You could need 4 billion computations to find 2 errors caused by 1 bug.

Exhausting Your Test Options

by Douglas Hoffman

SOME TIME AGO, I WORKED AS QA MANAGER FOR a startup that was making massively parallel systems. Each computer system had between 1,024 and 65,536 processors. The most powerful systems we made were about twenty times as powerful as the largest single-CPU systems at the time (in raw CPU cycles). With that much raw CPU power, we could address some problems that were otherwise unthinkable at the time. (For example, one of our systems was used to correct some of the images from the Hubble Space Telescope while the lenses were being fixed.) That much raw CPU power also created some testing problems and opportunities that were unthinkable at the time. Let me give you a bit of background so you can better understand the challenges we faced.

The system processor array was laid out much like a spreadsheet, with rows and columns of processors that had neighbor processors at each of eight relative positions, as shown in Figure 1. The processors could send and receive data with each of the neighbors directly and with any other processor in the array indirectly. Each processor received the same machine instructions from the program, but had unique data. (Such machines are called "SIMD," for Single Instruction, Multiple Data.) SIMD machines work well for problems, such as weather modeling, that can be broken up into a large number of chunks of data, each of which is handled the same way. The algorithm is the same everywhere, but each processor's outcome is uniquely defined in its own private data.

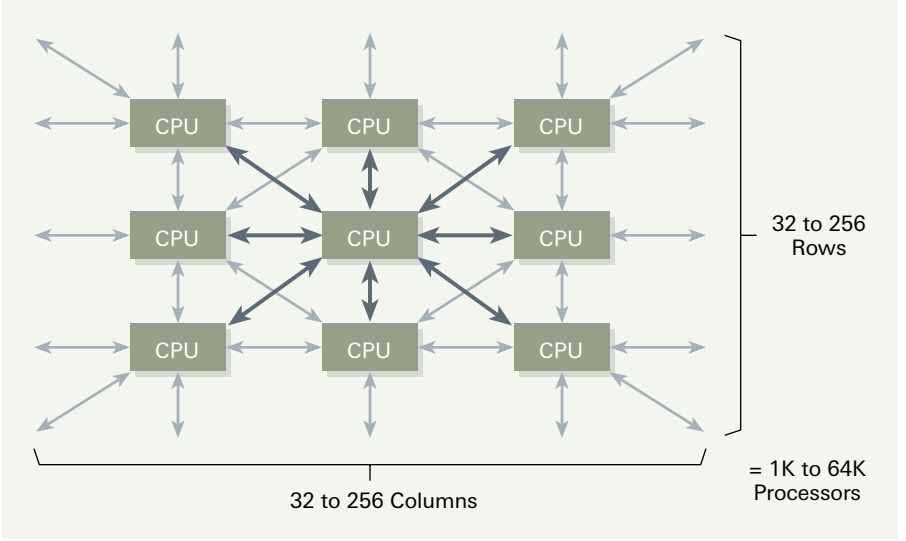

Figure 1: Massively Parallel Processor Configuration

We had several levels of software to deal with. There were some ordinary applications. There were programmer tools like compilers, debuggers, and data visualizers. There was an operating system that controlled the array. And there was even software embedded in the processor chips. Each instruction a processor executed was actually a small program, written in what's called "microcode." With all these levels to test, this was definitely an environment that kept the test and QA group hopping.

Dinner Napkins and Microcode

I was having dinner at the office one evening with one of the two principal architects of the system. (As is typical at most startups, we had dinner brought in five or six nights a week to encourage us to stay, reduce the time we had to take to eat, and provide more opportunities for us to talk shop.) We were in the final stages of defining the processor instruction set and were trading ideas about validating it. Our conversation eventually came around to the problem of dealing with the most complex machine instruction—square root.

Most of the machine instructions were very simple programs with a few dozen microcode steps. We had confidence in these from code inspections and use. The square root functions, however, were much more complex, running hundreds of steps. Inspection was complicated by the fact that only the two principal architects understood the details of the processor architecture and machine micro instruction set, so they were the only people qualified to analyze the code in depth. Others of us participated in some of the inspections, but were limited by our fuzzy understanding of the internals of the machines.

Between bites of dinner, we discussed the number of hours it was going to take to inspect the code and when we might be able to squeeze it into our schedules. As we bemoaned the loss of time and all the other work we had to do, an application program-

mer passing our table asked us a question. How long would it take to generate all the values using brute force? We had a very powerful system for some kinds of problems—and mathematical computations was one of them. It would not take much programming effort to get each of the 65,536 processors to compute square roots for different values, so the computation of all 4,294,967,296 values for the 32-bit square root instruction would not take much time at all. The slowest and most difficult part of such a test would be figuring out whether or not the computed result was correct. We were also worried about common mode problems. (Those show up when the thing you're testing and the thing that generates the expected result both use the same component or algorithm. If there is an error in the shared component, both the code under test and the generated expected result will have the same wrong answer.) Scratching on the back of a napkin (literally), we figured out that even using a different algorithm on a neighboring processor was only going to take a few minutes.

We designed a test where each of the 64K CPUs started with a different value and tested each value 65,536 greater than the last one. Then, each CPU would ask a neighbor to compute the square root differently to check the original result.

Two Errors in Four Billion

Two hours later, we made the first run using our biggest processor configuration. The test and verification ran in about six minutes or so, but we were confused a bit when the test reported two values in error. Both the developer and I scratched our heads, because the two values didn't appear to be special in any way or related to one another. Two errors in four billion. The developer shook his head and said, "No…This is impossi…Oh!…Okay?" He thumbed through the microcode source listing and stabbed his finger down on an instruction. "Yeah—there it is!" There was only one thing that could cause those two values to be wrong, and a moment's thought was all it took to discover what that was!

He gleefully tried to explain that the sign on a logical micro instruction shift instruction was incorrect for a particular bit pattern on a particular processor node at 11 P.M. on the night of a full moon (or some such—it didn't make a lot of sense to me at the time—but we caught, reported, and fixed the defect, fair and square). We submitted the defect report, noting only the two actual and expected values that miscompared and attaching the test program. Then, in just a few minutes, he fixed the code and reran the test, this time with no reported errors. After that, the developers ran the test before each submission, so the test group didn't ever see any subsequent problems.

I later asked the developer if he thought the inspection we had originally planned would likely have found the error. After thinking about it for a minute, he shook his head. "I doubt if we'd have ever caught it. The values weren't obvious powers of two or boundaries or anything special. We would have looked at the instructions and never noticed the subtle error."

Be Like Don Quixote

Failures can lurk in incredibly obscure places. Sometimes the only way to catch them is through exhaustive testing. The good news was, we had been able to "exhaustively test" the 32-bit square root function (although we checked only the expected result, not other possible errors like leaving the wrong values in micro registers or not responding correctly to interrupts). The bad news was that there was also a 64-bit version, with four billion times as many values—that would take about 50,000 years to test exhaustively. Maybe between now and 50,000 years from now, we'll come up with better tests to catch these potential errors. In the meantime, I think we should continue to challenge our assumptions of what's possible in our never-ending quest to build better software. Who knows what giants we might slay? STQE

Douglas Hoffman is the principal consultant with Software Quality Methods, LLC, in Silicon Valley, providing planning, management guidance, and training to transform products and organizations. He is a Fellow of the ASQ and has earned his BACS, MSEE, and MBA degrees.

***STQE* magazine is produced by Software Quality Engineering.**