Software Testing and Verification in Climate Model Development

Thomas L. Clune, NASA Goddard Space Flight Center

Richard B. Rood, University of Michigan

// *Recent advances in commercial software tools and methodologies have created a renaissance for systematic fine-grained testing. This opens new possibilities for testing climate-modeling-software methodologies.* //

TESTING, VERIFICATION, EVALUATION, and validation are vital aspects of climate model construction. However, these processes are so ingrained into the cultures of modeling centers that we don't often recognize them separately.[1] We define *validation* as comparison with observations and *verification* as comparison with analytic test cases and computational products.[2] (*Testing* and *evaluation* are more generic; we discuss them later.)

The climate-modeling community is dominated by software developed by scientists rather than software engineers.[3] The community invests heavily in system-level testing, which exercises most portions of the underlying code base with realistic inputs. These tests verify that multiple configurations of the model run stably for modest time intervals and for certain key invariants such as parallel reproducibility and checkpoint restart.

On the other hand, the routine application of fine-grained, low-level tests (unit tests) is nearly nonexistent. Notable exceptions are limited to infrastructure software (such as that of the Earth System Modeling Framework). Infrastructure typically supports data organization and movement, for parallelism, I/O, and information transfer (for example, from ocean grid to atmosphere grid).[4]

The lack of use of unit tests in climate model development starkly contrasts with the practices for commercially developed software. This is because the community isn't aware of pervasive unit testing's benefits and has misconceptions about the nature and difficulty of implementing unit tests. Analyzing the model evaluation process reveals the elements of the ingrained culture of testing and evaluation, letting us define a more comprehensive approach to software testing. The benefits of this include improved software quality, software reliability, and model developer productivity.

Climate Model Practice

Model development is a two-phase cycle alternating between development and evaluation. During development, programmers extend the model's capabilities by producing new code. During evaluation, they verify whether those changes achieve the desired results without breaking other aspects of the implementation. Different scales of code changes warrant different types of verification, and all successful verifications are tentative (because later verifications might reveal flaws).

Developers naturally scale the verification phase to match the computational resources and time required for the process. For example, if the verification technique requires lengthy simulations, developers are unlikely to verify each new line of source code. Rather, they're likely to spend one or more days programming before submitting a verification run. Likewise, if the verification process is merely to ensure no compilation errors, developers might perform verification frequently throughout the day.

Our focus is ultimately on model evaluations that aren't comparisons with observations—that is, aren't validation. However, it's useful to disentangle testing, verification, and validation more thoroughly. Validation is a controversial issue in climate-modeling centers. Indeed, Naomi Oreskes and her colleagues argued that formally validating geophysical models of complex natural systems is impossible.[5] More recent papers studying the philosophy and social science of the culture of climate and weather modeling suggest that the extensive comparison with observations and the extensive review of climate assessments do stand as validation.[6]

Because of formal programmatic requirements, the Data Assimilation Office (DAO; now the Global Modeling and Assimilation Office) at NASA's Goddard Space Flight Center had to produce a formal validation plan.[7] This plan defined a quantitative baseline of model performance for a set of geophysical phenomena. The DAO broadly studied these phenomena and simulated them well enough that they described a credibility threshold for system performance. This validation approach is defined by a specific application suite, it formally separates validation from development and relies on both quantitative and qualitative analysis.

We focus here on the testing and evaluation that are specific to the software itself—that is, testing and evaluation that should be completed before submitting the system for validation. The cost of fixing software defects decreases substantially if they're detected early.[8] A primary verification mechanism is a short-duration run of the full modeling system. Even when such tests can be performed in minutes, they're often too crude to detect all but the most basic types of errors. So, defects fester until more thorough runs are performed and detailed analyses of the results are examined.

If the verification technique requires lengthy simulations, developers are unlikely to verify each new line of source code.

When possible, finer-grained tests that directly verify the specific changes being attempted not only execute faster but also have a higher probability of detecting undesired behavior early. Professional software engineers have recognized these characteristics of software development and have generally moved toward development processes that provide rapid cycling between coding and verification.

Software-Testing Categories

Software testing can be applied at various levels in a climate model. Coarse-grained tests (those encompassing large portions of the implementation) can detect the existence of defects but generally have a limited ability to isolate specific causes. Fine-grained tests can isolate and diagnose specific defects, but the number of tests necessary to cover an entire application might be prohibitively expensive. Coarse-grained tests of modern parallel climate models might also require substantial computational resources, whereas fine-grained tests, with appropriate care, require minimal computational resources. In practice, coarse-grained tests are run nightly or weekly; fine-grained tests are run more frequently to support ongoing development.

Here, we describe three categories of testing, ranging from coarse- to fine-grained. Projects must consider each category's relative costs and benefits when formulating a testing plan.

System Tests

System testing is the coarsest level of testing in that individual tests exercise all or nearly all of the underlying implementation. Most system tests are regression tests—that is, they reveal errors in existing functionality. Regression tests commonly used in climate modeling seek to answer questions such as:

- Do the primary model configurations run to completion?
- Are the results independent of parallel decomposition?
- Does checkpoint/restart alter the results?
- Has the computational performance significantly changed?

Although the regression-testing process is similar to studying the consequences of scientific changes to the model, the distinction is important. A regression test is objective and can be automated.[9] A scientific change involves trade-offs on subtle qualities of a model and requires a variety of investigations into the model's behavior.

One impediment to system testing in most, if not all, climate models is the many configurations that could be tested. For example, models can be configured with or without components such as an ocean, atmospheric chemistry, and aerosols, each of which might have alternate implementations. The number of combinations grows exponentially with the number of components and can limit the extent to which you can thoroughly test a system.

Despite the difficulties induced by the proliferation of model configurations, system-level testing is where climate models have the greatest investment and maturity. For example, ModelE (the climate model from the Goddard Institute for Space Studies)[10] employs continuous integration. This allows nightly automated tests to automatically obtain the latest development branch and compile and execute about 10 model configurations under a variety of parallel decompositions. If a configuration fails to compile or execute or if the results aren't strongly reproducible, the system emails an alert. Even when the test reports are ignored in real time, experience shows that these tests often bracket the dates on which a defect was introduced and greatly reduce the effort to identify and fix the problem. Other climate teams typically have longer integration cycles but compensate by increasing the intensity of testing at those boundaries.

Integration Testing

Integration testing is the testing of software aggregates and is intermediate between system testing and fine-grained unit testing. Although there's much to be said about integration testing, the distinction is of little consequence to the arguments in this article.

Unit Testing

Unit tests are the finest granularity of testing; they verify an implementation's low-level behaviors. Generally, a unit test exercises a single unit (function or subroutine) by comparing output generated from a set of synthetic input values with corresponding output values. Unit-testing concerns that arise specifically for climate modeling include parallelism, checkpoint/restart, and numerical computations. Automated fine-grained testing is limited in current climate models (although there are a few counterexamples generally associated with infrastructure layers). Until quite recently, developers dreaded creating unit tests. They thought the tests took too much effort and distracted them from producing "real" software. Two innovations have changed this attitude. The first was language-specific unit-testing frameworks (such as JUnit), which make creating and executing tests and reporting test results easier. The second innovation was test-driven development (TDD),[11] which reverses the conventional software development process: developers create tests before creating the software to be tested.

The change in developer perception of unit tests is perhaps best typified by Michael Feathers: "The main thing that distinguishes legacy code from non-legacy code is tests, or rather a lack of tests."[12] He implies that the existence of a sufficiently robust suite of tests completely alters the experience, productivity, and overall risk when working with a software system.

Spurred on by JUnit's success, developers have created unit-testing frameworks for most modern programming languages. There are at least three independent frameworks for Fortran, including one that supports testing parallel implementations based on MPI (Message Passing Interface), pFUnit. Although these frameworks vary from language to language, their basic architectures are similar. Each framework provides a suite of assertion routines, which developers can use to express the expected state of output variables after a call to the routine being tested. When an assertion fails to hold, the framework logs the name of the test that fails along with the assertion's location and any accompanying user-provided messages for that assertion.

Test-Driven Development

TDD is a major element of agile development. It consists of a short development cycle that alternates between developing tests and developing code that enables the tests to pass. First, the developer creates or extends a test, which then fails because the necessary functionality hasn't yet been created. Then, the developer produces just enough code to pass the test. Finally, the developer cleans up, particularly removing any incidental redundancy. The process encourages progress in the form of rapid, incremental steps and is made practical through unit-testing frameworks.

To better understand the TDD workflow and its application in the context of numerical software, consider developing a simple 1D linear-interpolation procedure. The first step is to create a test. Usually this test is extremely simple and is intended primarily to specify the desired interface for the procedure to be implemented. For example, we could use

```
assert(0 == interp(x=[1,2],y=[0,0],at_x=1)).
```

With the test in place, we implement the interpolation procedure, but only so far as to enable successful compilation and execution of the test. A trivial implementation that returns zero will suffice.

Next, we choose data having a simple linear relationship:

```
assert(1 == interp(x=[0,2],y=[0,2],at_x=1)).
```

This test will compile and execute but will fail owing to the overly simplistic implementation after the first test. We proceed by extending the implementation to pass both tests.

Additional tests could then check that the behavior is correct when the data contain multiple intervals, when the interpolation is outside the domain, when the data are degenerate, and so on. After each test, we extend the interpolate procedure to ensure that all tests pass. For many developers, proceeding in such minute steps will initially be counterintuitive, but with practice these steps happen quickly and lead to steady, predictable development of complex features.

TDD practitioners tout its many benefits. Chief among these is the improvement in overall developer productivity (despite the substantial increase in the total lines of code for a given piece of functionality). The improved productivity stems from the comprehensive test coverage, which in turn leads to far fewer defects. Tests under TDD run continuously and can therefore serve as a form of *maintainable* documentation. Developers using TDD might also be more productive owing to the reduced stress and improved confidence that arise from the immediate feedback as they extended the functionality. Finally, TDD leads to higher-quality implementations. This somewhat surprising claim stems from the observation that software designed to be testable tends to contain smaller procedures with shorter argument lists.

High-resolution grids don't achieve anything other than slower test execution and less available memory.

Barriers to Testing

Scientists often perceive testing as an attempt to execute a model, a component, or an algorithm with maximal fidelity. Their instinct is to use representative grid resolutions, realistic input values, and so on. Software testing uses synthetic input values and small grids that exercise specific lines of code to produce output values that can be verified by inspection. High-resolution grids don't achieve anything other than slower test execution and less available memory. The use of realistic input values is appropriate only to avoid branches that are unsupported by the implementation, or when independent realistic output values are available. Working with synthetic inputs requires care to ensure that important cases are covered. For example, you can detect frequent programming errors (such as index permutations and wrong offsets) by imposing a spatial dependence on test inputs.

The largest technical barrier to unit testing in climate-modeling software is the legacy nature of the code base. Large procedures (more than 1,000 lines of source code) that rely heavily on global variables are difficult to characterize with unit tests. One possible test might be to store representative input and output data of a procedure in an external file. You could then develop tests that exercise the procedure with the stored data and notify you of any changed output values. However, such tests would be of limited value other than for refactoring.

Teams that desire to enhance software testing mitigate the problem of legacy software by limiting modification of existing routines. Rather than "wedging" new code into a large procedure, they develop a new testable subroutine. In the legacy procedure, they insert only a call to the new routine. Groups also look for opportunities to extract small, testable routines from large legacy procedures.

Beyond the burdens of the legacy code base, numerical algorithms present difficulties absent from many other software categories. The most basic of these arise from numerical errors originating from both truncation and rounding off of floating-point operations. These errors are problematic because the corresponding tests must not only specify expected values for output parameters but also provide tolerances for acceptable departures from those values. If the tolerances are too tight, tests will appear to fail despite producing acceptable results; if the tolerances are too loose, tests lose value as they fail to detect some errors. In most cases, developers don't have suitable values a priori for such tolerances, even when they know the asymptotic form of the truncation error for an algorithm.

Fortunately, the difficulty of specifying suitable tolerances isn't as severe in practice as the previous discussion indicates. The tests are verifying the implementation, not the asymptotic form of the algorithm. When you break an algorithm into sufficiently small steps, the resulting pieces are often amenable to simple, even trivial error analysis. Complicated error bounds are largely a consequence of the how errors compound through a calculation's subsequent stages. From a testing perspective, you can avoid this compounding by examining each step independently.

An important concern when dealing with climate models is whether small changes to an implementation result in a different basin of attraction for the trajectories, leading to a different climate. The chaotic nature of the underlying dynamics effectively limits unit tests' ability to constrain the system in this regard; only long control runs can protect against undesirable changes. Likewise, so long as all local tests pass, long control runs can't identify any particular aspect of the implementation as being responsible for an incorrect result (although it will assume the most recent change is the offender). Over time, researchers identify additional constraints (and thus unit tests) that decrease the frequency of incorrectly altering the simulated climate, but verification will always require control runs.

Another major objection to the

testing of some numerical algorithms is the general lack of known analytic solutions for realistic algorithms that could be used to derive appropriate values to use in tests. Unless an equivalent and entirely independent algorithm is available, tests of such algorithms tend to have no real value because they're redundant with the implementation. As with our previous discussion of numerical errors, you can largely eliminate this concern by decomposing algorithms into distinct, simple steps for which synthetic test values are readily apparent.

For pedagogical purposes, consider how you would implement testing for a procedure that computes the area of a circle, `area = areaOfCircle(radius)`. You could choose a small set of trial values for the input radius (such as 0, 1, and 2) and verify the results (in this case, 0, 3.14159265, and 12.5663706, respectively). These results aren't obvious from inspection because mental arithmetic with π is problematic. You could replace these literal values with expressions involving π, but the resulting test would be nearly redundant with the algorithm being tested.

However, from a software-engineering perspective, a different option exists: introducing π as a second parameter into the interface. You can use simple values for π such as 1 or 2 for testing the procedure's implementation. You would implement the usual interface for `areaOfCircle(radius)` by passing a hardcoded value of π to a procedure that accepts two arguments. You can test this value by inspection or by trigonometric identities.

This approach for decomposing numerical schemes might seem overly burdensome at first. However, compared to the effort required for developing a scientific parameterization, the effort for expressing algorithms in this fine-grained manner would be modest. In addition, the return could be immeasurable in the form of early detection of software defects.

This approach also helps sidestep the issue of providing realistic input values for complicated physical parameterizations based on empirical parameters and curve fitting. Of course, if realistic input and output data are available, you should create corresponding tests. More often, though, high-level tests for such parameterizations should verify the sequence of substeps rather than try to compose a full numerical comparison. This approach is more natural with abstract data types and object-oriented programming and might require reengineering a relatively small number of interfaces in a model.

More broadly, splitting scientific models into small procedures to enable unit testing raises concerns about computational efficiency. If you split algorithms into very small procedures with short parameter lists, the performance overhead for real code could be substantial. Bootstrapping can overcome this difficulty. That is, you can test an optimized implementation (large monolithic optimized procedure) against the slower fine-grained implementation, which is in turn covered by simple, clean unit tests. Although this technique compounds development overhead, the fine-grained implementation can be a useful reference for attempts to further optimize an algorithm.

TDD Applications

At the Goddard Space Flight Center's Software Integration and Visualization Office (SIVO), we've aggressively applied TDD to several development projects, including two that are directly relevant to scientific modeling. The first was Gtraj (Giga-Trajectories), a parallel application that calculates the trajectories of billions of air parcels in the atmosphere. The other was Snowfake, a parallel implementation of a numerical model that simulates the growth of virtual snowflakes.[13] The projects' success demonstrates that unit testing in general, and TDD in particular, can be applied in modeling. (The projects also revealed the challenges senior scientific programmers encounter when repressing established programming habits and methodologies.)

Gtraj

Gtraj is a complete rewrite of a legacy application, the Goddard Trajectory Model,[14] originally developed in IDL (Interface Definition Language). To achieve portability and scalability, SIVO applied TDD to rewrite the application in C++ and MPI using a team of senior scientific programmers who were relatively inexperienced with unit testing and TDD. The primary numerical algorithms used in Gtraj are numerical interpolations of gridded meteorological data and temporal integrations of the interpolated flow field experienced by a given parcel of air. The original model was primarily tested by measuring the discrepancy in time reversal of a five-day integration. Although time-reversibility might seem like a stringent requirement, it actually can't detect many types of errors.

During the rewrite, the new fine-grained tests revealed two important errors in the original. First, the binning algorithm, which selects which grid points to use for interpolation, was incorrect in some regions with sharp changes in the pressure heights. The "interpolations" would then become extrapolations, occasionally resulting in extreme inaccuracies. The correction would have been difficult in the original monolithic implementation but was straightforward in the version developed under TDD.

The second error was the Runge-Kutta (RK) algorithm's behavior when parcels pass near the poles in a latitude-longitude grid. Examination of the large errors for such trajectories using synthetic test data led researchers to realize that the usual RK algorithm must be reformulated for curvilinear coordinate systems. With this correction, the accuracy of trajectories that pass near the poles improved by orders of magnitude. This in turn permitted substantially larger time steps and faster application execution.

At several points during Gtraj's creation, the process of following TDD challenged the primary developers. The temptation to immediately begin implementing an algorithm was difficult to resist, especially early in the project. Unfortunately, the pressure to skip tests is typically highest when they do the most good—when the design requires further thought or when the expected behavior is harder to specify. The developers also tended to use realistic input values in tests rather than "enlightening" values, but they rapidly improved.

Snowfake

Unlike Gtraj, the initial serial implementation of Snowfake was developed by a single developer experienced in using TDD for model infrastructure components. The project benefited from the precise mathematical specification of the model[15]; the developer translated this specification into a set of basic tests.

The initial implementation took approximately 12 hours of development time and comprised approximately 700 lines of source code and 800 lines of tests. The assembled application ran to completion on the first try and produced simulated snowflakes in excellent agreement with the reference paper. The simple microdebugging at each TDD stage apparently eliminated the substantial debugging typically experienced during the integration phase of even less complex programs. Other encouraging statistics are that procedures averaged under 12 lines and fewer than two arguments.

As you might expect, the initial implementation's computational performance was suboptimal. However, we used TDD to develop optimized implementations of key algorithms by using simple tests that compared the results of alternative implementations against the those of the baseline. For example, one significant cache-based optimization came from fusing the substeps associated with vapor diffusion. The test for the fused procedure was simple: compare its results with a sequential execution of each original substep. The implementations and tests of both the baseline and optimized implementations were retained in the software repository to support further development.

Confidence in the scientific predictions of climate models is contingent on confidence in those models' underlying implementations. Our experience with our two applications suggests that this testing methodology is applicable to climate modeling, but more relevant examples are needed. Changing established processes in any community is difficult, and scientists are understandably skeptical about our approach's costs and benefits.

In the near future, we intend to apply TDD to create a parameterized model component from scratch. The early phases of such a component's development are the ideal time to express the various physical requirements and constraints as unit tests. The results should serve as a strong demonstration of this technique's potential.

Many technical difficulties must be overcome before unit testing can become pervasive in climate models. Chief among these is the extreme difficulty of introducing fine-grained tests into the typical procedures of legacy science code. In many cases, we can sidestep this issue by implementing changes and extensions to models as new modules rather than wedging changes into large procedures. This approach is particularly valuable for bug fixes because the constructed test prevents reintroduction of the defect.

In the long term, developers need more powerful tools that let them efficiently extract disjoint bits of functionality from the legacy layer. In other programming languages, especially Java, new generations of powerful software development tools have proven effective. Photran[16] is an attempt to introduce some of these refactoring capabilities for Fortran. However, we need greater investments to bring appropriate tools to a state of maturity suitable for routine application to climate models.

References

1. S. Shackley, "Epistemic Lifestyles in Climate Change Modeling," *Changing the Atmosphere*, C.C. Miller and P.N. Edwards, eds., MIT Press, 2001, pp. 107–133.
2. D.E. Post and R.P. Kendall, "Software Project Management and Quality Engineering Practices for Complex, Coupled Multiphysics, Massively Parallel Computational Simulations: Lessons Learned from ASCI," *Int'l J. High Performance Computing Applications*, vol. 18, no. 4, 2004, pp. 399–416.
3. S.M. Easterbrook and T.C. Johns, "Engineering the Software for Understanding Climate Change," *Computing in Science and Eng.*, vol. 11, no. 6, 2009, pp. 64–74.
4. C. Hill et al., "Architecture of the Earth System Modeling Framework," *Computing in Science and Eng.*, vol. 6, no. 1, 2004, pp. 18–28.
5. N. Oreskes, K. Shrader-Frechette, and K. Belitz, "Verification, Validation, and Confirmation of Numerical Models in the Earth Sciences," *Science*, vol. 263, no. 5147, 1994, pp. 641–646.
6. H. Guillemot, "Connections between Simulations and Observation in Climate Computer Modeling: Scientist's Practices and 'Bottom-Up Epistemology' Lessons," *Studies in History and Philosophy of Modern Physics*, vol. 41, no. 3, 2010, pp. 242–252.
7. R.B. Rood et al., *Algorithm Theoretical Basis Document Version 1.01*, tech. report, Data Assimilation Office, Goddard Space Flight Center, 1996; http://eospso.gsfc.nasa.gov/eos_homepage/for_scientists/atbd/docs/DAO/atbd-dao.pdf.
8. RTI, *The Economic Impacts of Inadequate Infrastructure for Software Testing*, tech. report, US Nat'l Inst. Standards and Technology, 2002; www.nist.gov/director/planning/upload/report02-3.pdf.
9. S. Vasquez, S. Murphy, and C. DeLuca, "Earth System Modeling Framework Software Developer's Guide," 29 Aug. 2011; www.earthsystemmodeling.org/documents/dev_guide.
10. G.A. Schmidt et al., "Present-Day Atmospheric Simulations Using GISS ModelE: Comparison to In Situ, Satellite, and Reanalysis Data," *J. Climate*, vol. 19, no. 2, 2006, pp. 153–192.

11. K. Beck, *Test-Driven Development: By Example*, Addison-Wesley, 2003, p. 240.
12. M. Feathers, *Working Effectively with Legacy Code*, Prentice-Hall, 2004, p. 456.
13. J. Gravner and D. Griffeath, "Modeling Snow Crystal Growth: A Three-Dimensional Mesoscopic Approach," *Physical Rev. E.*, vol. 79, no. 1, 2009, doi:10.1103/PhysRevE.79.011601.
14. M.R. Schoeberl and L. Sparling, "Trajectory Modeling," *Diagnostic Tools in Atmospheric Physics*, J.C. Gille and G. Visconti, eds., IOS Press, 1995, pp. 419–430.
15. J. Gravner and D. Griffeath, "Modeling Snow Crystal Growth: A Three-Dimensional Mesoscopic Approach," *Physical Rev. E*, vol. 79, no.1, 2009.
16. "Photran—an Integrated Development Environment and Refactoring Tool for Fortran," Eclipse Foundation, 2011; www.eclipse.org/photran.

cn Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.

ABOUT THE AUTHORS

THOMAS L. CLUNE is the head of the Advanced Software Technology Group in the Earth Science Division at NASA's Goddard Space Flight Center. His research interests include techniques to improve the quality, the performance, and tests of complex scientific software. Clune has a PhD in physics from the University of California, Berkeley. He's a member of the American Geophysical Union. Contact him at thomas.l.clune@nasa.gov.

RICHARD B. ROOD is a professor in the University of Michigan's Department of Atmospheric, Oceanic and Space Sciences and School of Natural Resources and Environment. His research interests include problem solving in climate change. Rood has a professional degree in meteorology from Florida State University. He serves on the US National Academy of Sciences Committee on a National Strategy for Advancing Climate Modeling, is a Fellow of the American Meteorological Society, and has received the World Meteorological Organization's Norbert Gerbier Award. Contact him at rbrood@umich.edu.

QUALCOMM®

Senior Software Engineer/Graphics

(5775 Morehouse Drive, San Diego, CA 92121)

As the world leader in next generation mobile technologies, Qualcomm is focused on accelerating mobility around the world. Qualcomm has the following position available in San Diego, CA. This position is responsible for the research, design, development, integration, testing and performance optimization of software applications and computer software systems and features in conjunction with hardware product development for wireless telecommunications, analog and digital, multimedia, device drivers, connectivity, semiconductor and other high-tech electronics applications. Determine systems specifications, requirements and working parameters for hardware/software compatibility. Coordinates design of subsystems and integration of total system. Identifies, analyzes and resolves program support deficiencies. Develops and recommends corrective actions. Will accept a Master's degree or foreign academic equivalent in Computer Science, Computer Engineering, Electrical Engineering or related and one year of experience in a related occupation. Will also accept a Bachelor's degree or foreign academic equivalent in Computer Science, Computer Engineering, Electrical Engineering or related and five (5) years of progressive experience in a related occupation. Candidate must have academic or industry experience and demonstrated proficiency in Performance Analysis & Optimization and Parallel Computing Algorithms & Architecture over multi-cores and academic or industry experience in five (5) of the following skills: Data Structures; C or C++ Programming; Computer Architecture; Operating Systems; GPUs or 3D Graphics; Multithread Programming; GPGPU or Cuda. $83,595.00 - $120,504.00. Standard corporate benefits. 40 hrs/wk (9 to 5). Please send resumes to Recruitment and Employment Office, QUALCOMM, Inc., Attn: Job Ref #: QUA69760, P.O. Box 56625, Atlanta, GA 30303. E/O/E

SC11

International Conference for High Performance Computing, Networking, Storage and Analysis

12-18 November 2011

Seattle, Washington, USA

Register today!

http://sc11.supercomputing.org/

IEEE

IEEE computer society