DCA++: Winning the Gordon Bell Prize with Generic Programming

Michael S. Summers

May 3 2009

Abstract

The 2009 Gordon Bell prize was won by the DCA++ code, the first code in history to run at a sustained 1.35 petaflop rate. While many GB prize wining codes have been written in FORTRAN, the DCA++ code is fully object-oriented C++ and makes heavy use of generic programming. This paper discusses how the DCA++ team simultaneously achieved world class performance and also the maintainability and elegance of modern software practice.

1 DCA++: The Rest of the Story

The DCA++ team is a large diverse team with a joint skill set that covers all aspects of designing, developing, hosting, running and using HPC applications. The story of this team's scientific pursuits and how they won the Gordon Bell prize has been well told by the team's leadership. For a quick overview, the funding sources, and the full list of team members see Thomas Schulthess's "The DCA++ story" [TSCH09]. What has been mentioned but not presented in detail is the software development story behind the DCA++.

The team's software development experience influenced its development. The author's own experience started in the early 70's, using FORTRAN on a CDC 6600 (designed by Seymour Cray). In the 80's he researched prototype command and control systems for the Air Force. Some of these systems ran on artificial intelligence machines such as the Symbolics 3600. Finally his experience came full circle back to scientific computing with Version 2 of the DCA++ code.

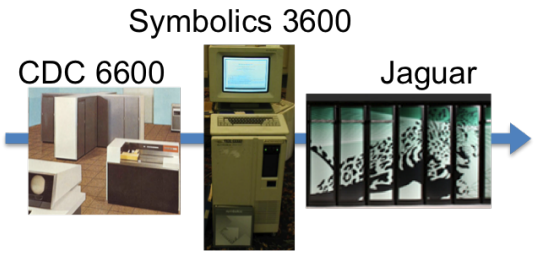

Figure 1: The code of DCA++ Version 2 was designed using skills first learned on a Symbolics Lisp Machine.

Researchers working in the 80's with Symbolics systems were asked to demonstrate very complex hybrid AI/numerical applications. Because of the nature of the problems they faced, the research often involved a mix of computer science, physics, and engineering. Since they needed to explore many possibilities they also had to develop and re-develop their software very quickly.

These researchers recognized that they were jointly developing both procedural and ontological knowledge and that they were encoding this knowledge into the software they developed. They understood that in order to manage the complexity and cost of their software they needed the ability to define software abstractions, and they developed languages and tools to this end.

However, it was not the languages and tools that managed the complexity of their code. It was the identification and use of correct/efficient abstractions (concepts in generic programming terminology) that managed complexity both in their minds and in their code. The structure of the code had to reflect the most efficient way to think about the problem. In cases where there were real time requirements the abstraction process had to be inter woven with performance confederations.

Looking back, the Symbolics represents more than a hardware and software platform. It represents the collective software development wisdom of its time.

This paper is the story of the application of this wisdom to the DCA++ code.

2 DCA++ Requirements

The DCA++ code is a multi-scale, quantum field theory application. At the bulk scale the "bath" (see Figure 2) is represented with a self energy function. At the atomic scale it uses a very large set of disorder configurations to model the many impurities that would be found in a super-conducting material. Each disorder configuration (the cluster of Figure 2) is represented by a single band 2D Hubbard model.

The self energy of the bath is used to initialize the clusters and the correlation functions computed from the clusters can be used to compute a new self energy. The code iterates until a self consistent self energy and correlation function are found. The computation of each cluster's correlation functions involves a Monte Carlo integration (represented by the dice in the figure).

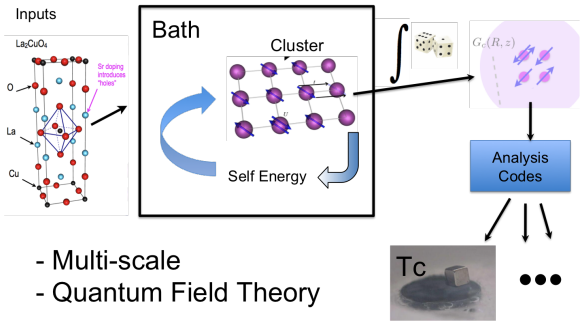

Figure 2: The DCA++ code computes correlation functions from a description of the material lattice. Material properties such as the superconducting transition temperature, Tc, can be calculated from the correlation functions.

The structure of this problem has at least three levels of hierarchical parallelism (see Figure 3). The available processors are first divided up into teams, one for each disorder configuration. Each disorder team can then be divided into Monte Carlo integration sub-teams.

Each integration sub-team uses a separate Markov chain to generate a sequence of integrand values. The processing of each Markov chain involves an expensive update procedure which can optionally be handled by sub-sub-team of processors. The update procedure involves some linear algebra which is the current bottleneck in the system. The code spends 95% of its time in these procedures. Depending on the hardware it may be appropriate for the bottleneck sub-sub-teams to share memory.

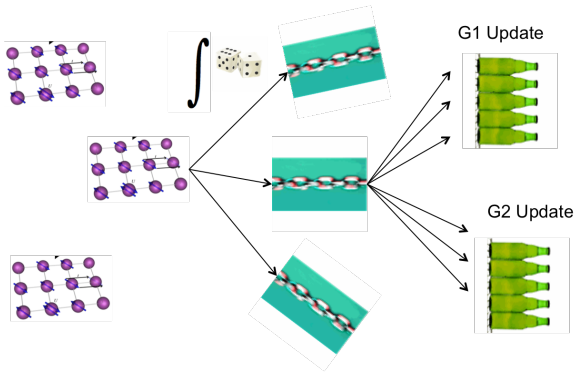

Figure 3: The DCA++ has a natural parallelism structure that can make use of a very large number of processors.

The DCA++ code was constructed in two stages. The first stage, Version 1, was developed to demonstrate that the team's algorithms could be implemented efficiently and that the code could be organized to scale to 50k or more processors. While Version 1 was written in C++ and made some use of object orientation and generic programming, it was not written in a way that would reduce the cost of adding future extensions.

As Figure 4 shows, Version 2, a complete rewrite, was required to meet all of Version 1's requirements and to also be written in a way that would reduce the cost of developing the many envisioned extensions. The allocation of requirements between Version 1 and 2 was also chosen to match the work loads and skills of the team's principle software developers. The principal developer of Version 1, Gonzalo Alvarez, an excellent and efficient coder, was foremost a physicist. The version that he produced provided the author (only a "wannabe" quantum mechanic) with the basis he needed to 'refactor' Version 1 into Version 2. This pairing of skill sets has worked well for us.

3 Refactoring

The basic idea of refactoring is presented in Figure 5. As every code is developed, it increases in both functionality and complexity. As the complexity of the code grows it becomes harder and harder to change until it reaches a ceiling where adding new capabilities is too costly. While it may appear that the code is at an impasse at this point, it is usually not so.

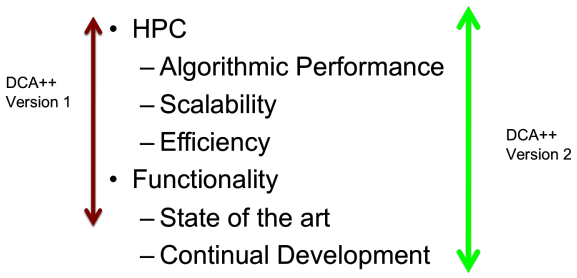

Figure 4: The DCA++ Requirements distributed between versions.

Much of the complexity of the code is not intrinsic to the problem the code is solving. During the refactoring process code is rewritten (often just a portion of it) so that 1) the non-intrinsic complexity is greatly reduced and 2) the new abstractions used in the code change the rate at which complexity grows with additional functionality. This is illustrated in Figure 5 by the reduced slope after refactoring.

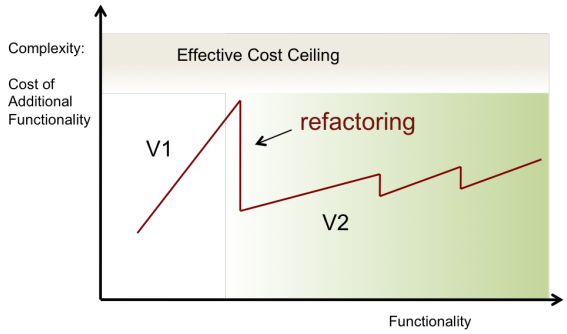

Figure 5: Refactoring enables us to afford more functionality.

It is usually not possible to avoid refactoring by comprehensive upfront design. This is especially true for research code. There are several reasons for this:

- The design of the programs abstractions involve many trade-offs. We often need experience with actual systems before we can make good design decisions.
- As researchers we are learning about the problem we are trying to solve as we develop the code.
- As researchers we are also discovering the best algorithms to use as we develop the code.
- Many refactoring opportunities can be identified by simply looking for redundant (or nearly redundant) code.
- Upfront designs of a system of abstractions/concepts are usually too general. Developers end up writing more code than necessary and leave this code untested.

4 The Race to the Finish

Our experience developing the DCA++ and submitting it for the Gordon Bell prize provides an interesting illustration of the refactoring process.

Figure 6 presents the two year time frame prior to the Gordon Bell DCA++ runs in November of 2008. During the first year Gonzalo Alvarez was writing Version 1 and the author was working on the underlying symmetry package which is shared by both systems. When the symmetry package was finished and integrated into Version 1, work on Version 2 began.

Development of Version 1 continued. As it demonstrated its functionality and performance it became the production version. It was used to demonstrated that it could scale to the order of 50k processors and was expected to be the Gordon Bell code.

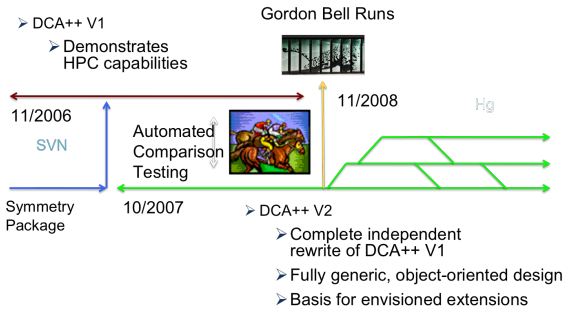

Figure 6: The time line of the DCA++ development.

Initially the performance runs were to be on the "old" Jaguar XT4, later in the year it became clear that the XT5 (with it's 150k processors) would be available for Gordon Bell Prize runs. There was a narrow time window in which the new machine would be available for this purpose. By that time several things had changed.

- We had developed a very detailed automated testing system to verify that Version 2 was producing exactly the same results as Version 1.

This system showed that Version 2 was very close to meeting its functional requirements.

- After sorting out some issues, Version 2 appeared to be running as fast as Version 1.
- We had decided that we should modify the codes so that the Monte Carlo integration would operate in single precision. Due to the nature of the integration process, we knew it was unnecessary to run this part of the code in double precision. It was also clear that our performance would increase significantly if we made this change.

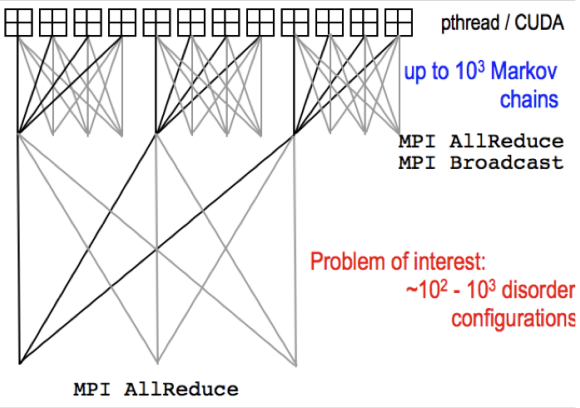

Figure 7: During the Gordon Bell runs a need to modify the collective MPI approach was discovered.

In the end it was very close race, Version 2 passed all of its acceptance test days before the window on the XT5 opened. However during the runs the refactoring paid off in two ways:

- The precision change turned out to be fairly trivial for Version 2 (an afternoon's work) and difficult for Version 1. This change resulted in 1.35 petaflops/s instead of .7 petaflops/s.
- Running on 150k processors versus 50k processors revealed a weakness in the MPI collective communication approach that we had previously taken. The same approach was used by both versions. However, Version 2 required a change to only a few lines of code in one inherited method, whereas the same change in Version 2 had to be made in various places.

As a result, Version 1 never actually crossed the finish line in terms of petaflop performance.

- ~~*FieldType* = double~~
  FieldType = float
- MonteCarloIntegration<*FieldType*>
- Automatically caused:
  dgemm -> sgemm
  dgemv -> sgemv

Figure 8: The use of generic programming made the change of precision easy in Version 2. The actual change from dgemm to sgemm did not require any code changes. It was performed automatically at compile time by the C++ template and overloaded function matching.

5 After the Gordon Bell Runs

After the Gordon Bell runs, Version 2 became the production code. Gonzalo Alvarez now maintains this code and jealously guards it against unvalidated modifications since he (and the other Physicists on the team) use the code for their work. The author works on the development versions. Just before the Gordon Bell runs we switched our version control system from Subversion(SVN) to Mercurial(Hg) (a distributed system). As Figure 6 shows, we now use Hg to manage all of the development and production branches of the code, periodically merging development capabilities back into the production version.

The development branches provide functionality which:

- Replaces the MpiSplit-based processor topology with a multi-dimensional MpiCart topology. (See Figure 9) This allows us to construct multi-dimensional disorder configurations. It also allows us to add another hierarchical layer in the parallel processing. This will permit us to experiment with combining results from many DCA++ runs each of which has its momentum structure shifted from the other.
- Provide the capability to exhaustively generate many combinations of disorder configurations.
- Provide a lightweight Java Script Object Notation (JSON) parser written in C++ with a few special features that allow us to handle large arrays efficiently.

- Replace the Delayed Update Algorithm with a much faster algorithm.

During all of this development activity low-level refactoring continues as we learn new ways to make the code simpler and more succinct.

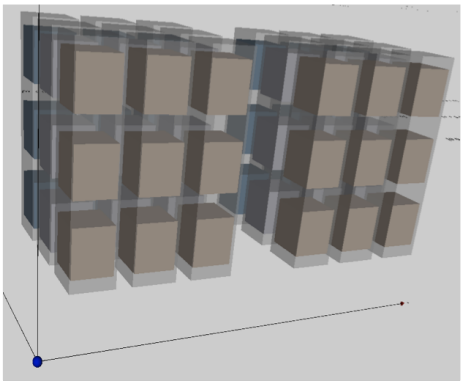

Figure 9: The development branch of Version 2 has changed from an MpiSplit-based topology to a Cartesian topology.

We have also been experimenting with the "D" programming language. The "D" language is simpler than C++, it's compilers are easier to write and it has a much cleaner template system. The very useful meta programming capabilities of C++ were accidental in nature. Because of this they are unnecessarily awkward. This makes "D"'s template system very attractive.

So far we have been able to install the GDC compiler on smoky.ccs.ornl.gov, construct the equivalent of mpic++, and write a small test MPI application. This application links with MPI, BLAS, and LAPACK which is all we require of DCA++. The next step is to write some small benchmark codes and evaluate them viz. a viz. their C++ equivalent.

Figures 10 and 11 contain summery information about the current development branch. As the figures show, the system has a minimum of external dependencies (requiring only MPI and linear algebra packages) and is almost entirely generic.

As we approach Version 3, we will be reworking our testing framework. Comparison runs with Version 1 are now insufficient since we will be testing many new capabilities. We will be automating and extending our unit testing framework. The theoretical side of the team will be identifying new system level tests. However, as we continue to refactor the system we find that its clarity makes it easier to verify through simple audit.

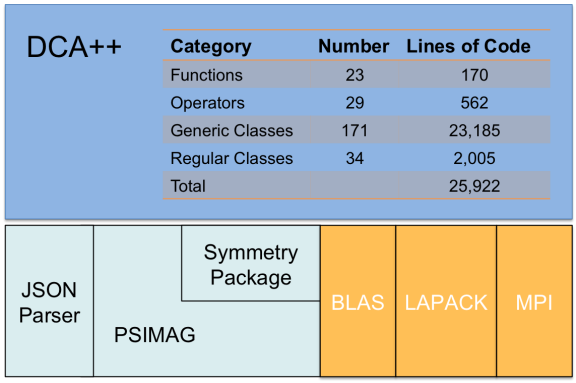

Figure 10: 97% of the DCA++ code is Object Oriented, 89% of it is Generic.

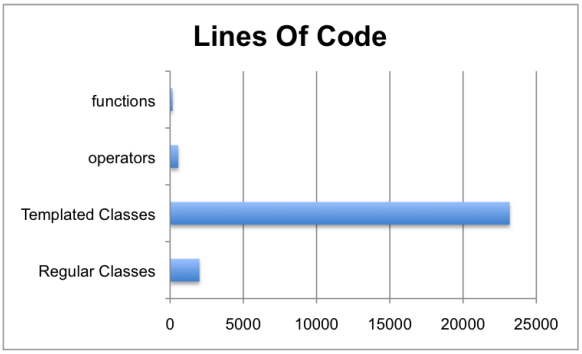

Figure 11: 97% of the DCA++ code is Object Oriented, 89% of it is Generic.

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult. It demands the same skill, devotion, insight, and even inspiration as the discovery of the simple physical laws which underlie the complex phenomena of nature." [Hoare1980]

6 Summary

Physicists generally understand that working with complex physical phenomena requires a foundation in mathematics. Analogously, we have come to understand that the development of good scientific code also requires a foundation. This foundation is a mastery of software technologies like generic programming.

The mathematics that a physicist uses to describe nature makes what he writes clearer to the mathematically trained, but at the same time it makes it more difficult to understand by those who do not have the requisite training. Similarly, the use of software abstraction technologies makes the code easier to work with for those who have training and harder to work with for those who don't.

Our assumption is that in the case of scientific codes such as DCA++, the use of "SWAT" development teams will be the most productive. Clearly those who learn the mathematics of quantum mechanics may also learn the analogous software technologies.

It has been our experience that it is relatively easy to develop software that is both high performance and well designed. Performance issues do affect the design of the code and the choice of the abstractions used. The developer must be constantly aware of the performance implications of his design decisions. Fortunately, compile time technologies such as generic programming help to improve performance more often than they cause problems.

7 About the Author

Michael Summers
Oak Ridge National Laboratory
S&T Staff
Computer Science and Mathematics Div.
1 Bethal Valley Road, Oak Ridge, TN
(865) 576-4488
x7u@ornl.gov

References

[TSCH09] Thomas C. Schulthess,
schulthess@cscs.ch,
The DCA++ Story.
Advance Scientific Computing,
Advisory Committee Meeting,
Washington DC, March 3-4, 2009.
http://www.er.doe.gov/ASCR/ASCAC/Meetings/Mar09/Schulthess.pdf

[Hoare1980] Charles A. R. Hoare,
1980 Turing Award Lecture;
Communications of the ACM 24 (2),
(February 1981): pp. 75-83.