DCA++: A case for science driven application development for leadership computing platforms

Michael S. Summers, Gonzalo Alvarez, Jeremy Meredith, and Thomas A. Maier
Computer Science and Mathematics Division, Oak Ridge National Laboratory, P. O. Box 2008, Mail Stop 6164, Oak Ridge, TN 37831

Thomas C. Schulthess
Swiss National Supercomputer Center and Institute for Theoretical Physics, ETH Zurich, CSCS MAN E 133, Galeria 2, CH-9628 Manno

schulthess@cscs.ch

Abstract. The DCA++ code was one of the early science applications that ran on jaguar at the National Center for Computational Sciences, and the first application code to sustain a petaflop/s under production conditions on a general-purpose supercomputer. The code implements a quantum cluster method with a Quantum Monte Carlo kernel to solve the 2D Hubbard model for high-temperature superconductivity. It is implemented in C++, making heavy use of the generic programming model. In this paper, we discuss how this code was developed, reaching scalability and high efficiency on the world’s fastest supercomputer in only a few years. We show how the use of generic concepts combined with systematic refactoring of codes is a better strategy for computational sciences than a comprehensive upfront design.

1. Introduction
The development of leadership computing platforms in recent years has been very remarkable. The introduction in 2002 of the Earth Simulator in Japan, resulted in a revitalization of scientific supercomputing in the United States, and the performance of the largest supercomputers available to open science increased by almost three orders of magnitude, from a few teraflop/s in 2002 to a petaflop/s in 2008. But the architectural changes of the computers were, from an application perspective, at least as challenging as the performance improvements. In November 2002, the largest system that was available to open science besides the Earth Simulator sustained the Linpack benchmark with 4.46 teraflop/s and consisted of 3,016 processor cores. Six years later, in November 2008, the largest system available to open science sustained Linpack with 1.06 petaflop/s and consisted of 150,152 processor cores. Put in simple terms, scientific simulations would only benefit from the factor 240 increase in performance if they could scale to 50 times the number of processor cores.

Weak scaling, where the required number of processors increases in proportion to the problem size, is an inherent property of many scientific simulation problems. Codes currently exhibiting this property are the natural choice to run on the next generation of supercomputers. However, limiting application selection to such codes would eventually lead to extinction of supercomputing in science since there would be fewer and fewer scalable applications to run on each generation of
supercomputer. Early science applications for Jagua\textit{r}, the first petaflop/s scale supercomputer at the National Center for Computational Sciences (NCCS), were thus selected according to their potential scientific impact and necessary investments were made to bring these applications to readiness.

The DCA++ code was one of these early science applications. It was selected because the study of models of high temperature superconductivity seemed to be a very promising endeavor for simulations, and advances in understanding the mechanism of high temperature superconductivity could have a big impact on the development of new materials for electric power transmission. But at the time of its selection, in the fall of 2006, it was not at all clear how these simulations would benefit from more than a factor 10 increase in number of cores on a NCCS leadership class system. In the present paper, we discuss how the DCA++ code was developed, and how it became the first scientific application to sustain a petaflop/s under production conditions.

2. Modelling superconductivity and the DCA/QMC method

Soon after high-temperature superconductors were discovered in 1986 [1], it became clear that the physics of superconductivity in these materials should be describable with a simple model, the single band 2 dimensional (2D) Hubbard model [2]. In this model, carriers (electrons or holes) are fermions with spin 1/2. They are able to move between sites on a square lattice with a hopping amplitude $t$ between near neighbor sites, and they repel each other with a Coulomb interaction $U$, when two carriers with opposite spin occupy the same site. Despite its simplicity and the massive amount of attention paid to it by the condensed matter physics community, the problem of describing superconductivity with this model has eluded a solution for nearly two decades. The problem does not have an analytic solution and any numerical solution suffers from the curse of dimensionality inherent to quantum many-body problems. The complexity of the 2D Hubbard model scales with $4^N$, where $N$ is the number of sites. Since superconductivity is a macroscopic quantum effect, nominally $N$ would be of order $10^{23}$. At the same time, numerous experiments indicated that quantum correlations at atomic and nanometer lengths scales play an important role as well. Because of this, an approximate numerical solution of the Hubbard model capable of describing superconductivity, has to address a multi-scale problem that stretches from atomic to macroscopic length scales.

The development of quantum cluster methods [3] in the late 1990s, laid the methodological foundation for solving the 2D Hubbard model. The basic idea is to solve the quantum many-body problem accurately on a cluster that is large enough to account for the nano-meter scale quantum correlations, and to coherently embed the cluster into a mean-field that accounts for the macroscopic length scales. The resulting approximate method has the name “Dynamical Cluster Approximation” (DCA). Numerically, the cluster many-body problem is solved with a variant of the Hirsch Fye Quantum Monte Carlo method (QMC), that has been proposed by Ed D’Azevedo [4] in order to perform well on superscalar processors used in the Cray XT line of supercomputers.

In 2005 [2] it was possible to systematically show for the first time that the 2D Hubbard model describes a superconducting transition with a critical temperature that is comparable to the typical values observed in the cuprates. A series of subsequent simulations revealed details of the pairing mechanism in the model [6].

Meanwhile, a series of remarkable experimental studies were published, in which a relationship between nano-scale inhomogeneities and pair formation was found [7,8]. In particular, the paper from the Princeton group [8] showed that regions exist in the superconductor, in which the superconducting pair-formation persists at temperatures that are significantly higher than the macroscopically observed transition temperature. This indicates that understanding the role of inhomogeneities and disorder effects in the pair formation will be crucial in understanding the cuprate materials. However, these studies would require a new code base, in which variants of the Hubbard model with disorder and inhomogeneities could be quickly introduced, tested, and studied with simulations at scale. Since supercomputers with the necessary performance were on the horizon, it was now time to prepare the DCA/QMC code base for these new science problems that would be studied on petaflop/s scale systems – a decision was made to develop the DCA++ code.
3. **DCA++ Code requirements**

The DCA++ code is a multi-scale quantum field theory application. At the macroscopic scale the “bath” (see Figure 1) 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 the cuprate sample. Each disorder configuration (the cluster of Figure 1) is represented by a particular set of (random) parameters of the single-band 2D Hubbard model. The self-energy of the bath is used to initialize the cluster and the many-body Green’s functions (a complex valued function similar to a many-body correlation function) computed from the cluster can be used to compute a new self-energy. The code iterates until a self-consistent self-energy and many-body Green’s function are found. The computation of each cluster’s many-body Green’s function involves a Monte Carlo integration (represented by dice in Figure 2).

![Figure 1](image1.png)

**Figure 1.** The DCA++ code computes the many-body Green’s function from a model description of the material crystal. Materials properties such as the superconducting transition temperature can be calculated from these many-body Green’s functions.

![Figure 2](image2.png)

**Figure 2.** DCA++ has a natural parallelism structure that can make use of a very large number of processors.

The structure of this problem has at least three levels of hierarchical parallelism (see Figure 2). 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 (the modified Hirsch-Fye algorithm with delayed updates [4]) that can only be handled by a sub-sub-team of processors. The update procedure solves a linear algebra problem that is currently the computational bottleneck of the simulation – the code spends about 95% of the time in this procedure. The sub-sub-teams can be further divided to parallelize the updates or the calculation of observables.

4. **Generic software development in stages: systematic refactoring**

The key aspects of generic programming and its relationship to systematic refactoring are described by Douglas Gregor, et. al. in an OOPSLA paper entitled “Concepts: Linguistic Support for Generic Programming in C++” [10]:

- “Generic programming is a systematic approach to designing and organizing software that focuses on finding the most general (or abstract) formulation of an algorithm together with an efficient implementation. [9]
- The generic programming process derives generic algorithms from families of concrete (non-generic) implementations that exhibit some commonality. We lift away unnecessary requirements on types from an implementation, thereby raising the level of abstraction”. [10]
- “Once many algorithms within a given problem domain have been lifted, we start to see patterns among the requirements. … When this occurs, each set of requirements is bundled
into a concept. … When the Generic Programming process is carefully followed, the concepts that emerge tend to describe the abstractions within the problem domain in some logical way. … Thus, the output of the Generic Programming process is not just a generic, reusable implementation, but a better understanding of the problem domain. “ [11]

For computational scientists this approach simultaneously provides multiple benefits. It permits them to organize their software into concepts that are natural to the physical problems being studied and concepts that are natural to the computational techniques they use to solve these problems. This makes the code easier to verify and extend. The lifting process provides a natural methodology for reorganizing their code so that it continually reflects the results of their research. In our development of DCA++ we have demonstrated that it is possible to do this without sacrificing efficiency and scalability.

Investigations of complex systems by means of simulations are reminiscent of Cold War era research into complex hybrid AI/numerical applications. These projects involved a mix of computer science, applied mathematics, physics, and software engineering. Software development had to be nimble, because no direct path to a solution was obvious in the beginning and thus software had to be developed and re-developed many times. In order to manage complexity and cost, these researchers 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 ultimately reduced the complexity of these codes. It was the identification and use of correct/efficient concepts and their refinements that managed complexity both in the minds of researchers and in the computer code. The structure of the code had to reflect the most efficient way to think about the problem and the best way to think of the problem had to be developed while the code was being developed.

This is the view we took in the DCA++ code, namely, that it is not possible to avoid future rewriting of the code by comprehensive upfront design. Rather, the understanding of what needs to be coded, what the best abstractions are, and how these can be implemented in the most efficient way, is wisdom that has to be allowed to develop and improve. Hence the DCA++ project is designed to allow frequent refactoring of the software, and there is no expectation that there will ever be a final design of the software until the project is completed.

The basic idea of refactoring is presented in Figure 3. As every code is developed, it increases in both functionality and complexity. As the complexity of the code grows, it becomes 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 usually is not. 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 3 by the reduced slope after refactoring.

Trying to avoid refactoring by comprehensive upfront design is usually not a good strategy in computational science research codes. There are several reasons for this:

- The design of the program abstractions involves many trade-offs. Experience with the actual system is needed before good design decisions can be made.
- While the code is being developed, researchers learn about the problem they are trying to solve/explore leading to changes in requirements.
- Algorithm development is usually part of the research project, and the best algorithms for a problem on a particular architecture are only discovered during code development.
- It is efficient to identify refactoring opportunities by looking for redundant code.
- Upfront designs of a system of abstractions/concepts are usually too general, and developers end up writing more code than necessary, leaving much code untested.
5. The first sustained petaflop/s: a practical example of refactoring process

In November 2008, DCA++ was the first application code to sustain a petaflop/s under production conditions on a general-purpose supercomputer. The two-year development time that lead up to this point provides an interesting illustration of the refactoring process.

The DCA++ code was constructed in two stages. The first stage, Version 1 was developed to demonstrate that the team’s approach could be implemented efficiently, correctly, and in a scalable manner to run on 50 thousand 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 minimize the cost of adding future extensions. The second stage, Version 2, represented a complete rewrite to meet all of Version 1’s requirements, and to significantly reduce the cost of developing the many envisioned extensions.

![Figure 3. Refactoring enables us to afford more functionality.](image1)

![Figure 4. DCA++ development time line.](image2)

Figure 4 presents the two-year development timeframe prior to the first petaflop/s scale runs. Version 1 was developed primarily by Gonzalo Alvarez, a computational physicist with extensive background in numerical approaches similar to the ones used in the DCA/QMC method. Initially, the work on Version 2 focused on developing a symmetry package that would be available for use in Version 1 as well. The actual development of the DCA++ application Version 2 started in Fall 2007. Its primary developer was Mike Summers, a mathematician by training with vast software development experience, who was required to become familiar with the basic aspects of quantum field theory used in the DCA/QMC method. About at the same time, work was started by Jeremy Meredith, a computer scientist with experience in computer graphics and visualization, to experiment with Version 1 on graphic processing units (GPU).

The work with DCA++ on GPU showed that promising performance gains could be made on graphic accelerators, and most importantly, that the Hirsch-Fye QMC solver with delayed updates produces satisfactory results with only single precision arithmetic. This was in April 2008, at about the same time when Version 1 demonstrated scalability to the full Cray XT4 system with 8,000 quad-core processors (32 thousand cores in total). Since using a single precision QMC solver would accelerate the DCA++ code on regular CPUs as well, it was decided to introduce some of the changes of the GPU experiment back into Version 1. Making a reliable mixed precision version of the code, where the self-energy evaluation would remain in double precision and the cluster solver would be allowed to run in single precision, was difficult and the production runs with Version 1 were continued with double precision. However, the design requirement to support mixed precision capability was added to Version 2.

In the meantime a very detailed automated testing system was developed to verify that Version 2 produced exactly the same results as Version 1. This lead to a rather swift debugging and validation phase of the code and by fall 2008, Version 2 was ready for production when the Cray XT5 petaflop/s scale supercomputer was ready for initial tests with applications under production conditions. The refactoring had already paid off in two ways:
• Running on 150 thousand processors versus 50 thousand earlier in the summer revealed a weakness in the MPI collective communication approach that was taken in both versions of the DCA++ code. However, Version 2 required changes to only a few lines of code in one inherited method and near perfect scaling was reached up to the full size of the supercomputer. The required changes in Version 1 were more involved.
• The change to a mixed precision version of the DCA/QMC method was fairly trivial in Version 2 – an afternoon’s worth of coding and debugging work – since the possibility to alter precision in different parts of the code had been added to the specification. This change accelerated the time to solution by a factor 1.9, resulting in 1.35 petaflop/s sustained performance of the code under production conditions.

Figure 5 summarizes the basic structure and size of the resulting DCA++ Version 2 code. For basic I/O, the code uses a Javascript Object Notation (JSON) parser. Ideally, a code like DCA++ would be based on XML, for its high-level I/O. However, maintaining the necessary parser system for XML on bleeding edge supercomputers was not practical, and a much lighter JSON parser that we wrote in C++ was used instead. Many of the elementary data structures and algorithms for condensed matter physics simulations have already been abstracted and collected in the PSIMAG library. The symmetry package, which contains all the necessary low-level machinery to apply the DCA method to arbitrary space groups in 2D and 3D, has been added to the PSIMAG library as well, as it will be used by many other condensed matter and materials science codes (this package alone consists of about 200,000 lines of code). Like most other computational science codes, DCA++ relies on libraries such as BLAS and LAPACK for dense linear algebra, MPI for message passing, and CUDA or pthreads for processor dependent fine-grained parallelism. With this structure, the actual DCA++ code ends up being relatively small – only about 25 thousand lines of code of which almost 90% are contained in generic classes.

| DCA++ | Category       | Number | Lines of Code |
|-------|----------------|--------|---------------|
|       | Functions      | 23     | 170           |
|       | Operators      | 29     | 562           |
|       | Generic Classes| 171    | 23,185        |
|       | Regular Classes| 34     | 2,005         |
|       | Total          |        | 25,922        |

| JSON Parser | PSIMAG | Symmetry Package |
|-------------|--------|------------------|
|             |        | BLAS             |
|             |        | LAPACK            |
|             |        | MPI               |

**Figure 5.** DCA++ code structure: 97% is Object Oriented, 89% is generic

6. Summary
The main characteristics of computational science application development is that several issues have to be addressed concomitantly: model-development, the methods to solve these models, and the algorithms to optimally map resulting computing codes onto high-performance computing hardware. Implementing software for such applications has to follow a non-conventional path, with frequent and systematic refactoring steps built into an iterative design process. Relying on rigorous up-front design
of abstractions turns out not to be practical. The DCA++ code development successfully followed such an approach, and after a two-year development process resulted in a production code that implements bleeding edge algorithms to study superconductivity in models of the cuprate high-temperature superconductors at scale. The team that developed DCA++ has a mixed background from many disciplines, bringing together experience in condensed matter and computational physics, applied mathematics, and computer science. It was the first code to sustain a petaflop/s under production conditions on a general-purpose supercomputer.

Acknowledgement
This work was supported by the Laboratory Directed Research and Development fund of Oak Ridge National Laboratory and the Center for Nanophase Material Sciences, which is sponsored by the Division for Scientific User Facilities Division of the Office of Basic Energy Sciences, as well as a grant from the National Center for Computational Sciences, which is funded by the Facilities Division of the Office of Advanced Scientific Computing Research (ASCR).

References
[1] J. G. Bednorz and K. A. Müller, “Possible high-Tc superconductivity in the Ba-La-Cu-O system”, Zeitschrift f. Physik B-Cond. Matt. 64, 189-193 (1986)
[2] F. C. Zhang and T. M. Rice, “Effective Hamiltonian for the superconducting Cu oxides”, Phys. Rev. B 41, 7243 (1988).
[3] T. A. Maier, M. Jarrell, T. Pruschke, and M. H. Hettler, “Quantum Cluster Theories”, Rev. Mod. Phys. 77, 1027 (2005).
[4] Gonzalo Alvarez, Michael S. Summers, Don E. Maxwell, Markus Eisenbach, Jeremy S. Meredith, Jeffrey M. Larkin, John M. Levesque, Thomas A. Maier, Paul R. C. Kent, Eduardo F. D’Azevedo, Thomas C. Schulthess, Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008, November 15-21, 2008, Austin, Texas, USA. IEEE/ACM 2008, ISBN 978-1-4244-2835-9
[5] T. A. Maier, M. Jarrell, T. C. Schulthess, P. R. C. Kent, and J. B. White, “Systematic study of d-wave superconductivity in the 2D repulsive Hubbard model”, Phys. Rev. Lett. 95, 237001 (2005).
[6] T. A. Maier, M. Jarrell, and D. J. Scalapino, “Structure of the pairing interactions in the two-dimensional Hubbard model”, Phys. Rev. Lett. 96, 047005 (2006).
[7] K. McElroy, Jinho Lee, J. A. Slezak, D.-H. Lee, S. Uchida, and J. C. Davis, “Atomic-scale sources and mechanism of the nanoscale electronic disorder in Be₂Sr₂CaCu₂O₈₊δ”, Science 309, 1048-1052 (2005).
[8] K. K. Gomes, A. N. Pasupathy, A. Pushp, S. Ono, Y. Ando, and A. Yazdani, “Visualizing pair formation on the atomic scale in the high-Tc superconductor Bi₂Sr₂CaCu₂O₈₊δ”, Nature 447, 569 (2007).
[9] Mehdi Jazayeri, Rudiger Loos, David Musser, and Alexander Stepanov. Generic Programming. In Report of the Dagstuhl Seminar on Generic Programming, Schloss Dagstuhl, Germany, April 1998. Compiler Construction, pages 86–101, London, UK, 2001. Springer-Verlag.
[10] Douglas Gregor, Jaakko Jarvi, Jeremy Siek,Bjarne Stroustrup, Gabriel Dos Reis,Andrew Lumsdaine, OOPSLA’06 October 22-26, 2006, Portland Oregon, USA.
[11] http://www.generic-programming.org/