Computer Physics Communications 246 (2020) 106709

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

DCA++: A software framework to solve correlated electron problems with modern quantum cluster methods[☆]

Urs R. Hähner[a,*], Gonzalo Alvarez[b], Thomas A. Maier[b], Raffaele Solcà[c], Peter Staar[d], Michael S. Summers[b], Thomas C. Schulthess[a,b,c]

[a] Institute for Theoretical Physics, ETH Zurich, 8093 Zurich, Switzerland
[b] Computational Science and Engineering Division, Center for Nanophase Materials Sciences, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
[c] Swiss National Supercomputing Centre, ETH Zurich, 6900 Lugano, Switzerland
[d] IBM Research – Zurich, 8803 Rüschlikon, Switzerland

ARTICLE INFO

Article history:
Received 12 August 2018
Received in revised form 14 December 2018
Accepted 1 January 2019
Available online 17 January 2019

Keywords:
Strongly correlated electron systems
Quantum cluster algorithms
Dynamical cluster approximation
Continuous-time quantum Monte Carlo
Extreme-scale computing
Sustainable software development

ABSTRACT

We present the first open release of the DCA++ project, a high-performance research software framework to solve quantum many-body problems with cutting edge quantum cluster algorithms. DCA++ implements the dynamical cluster approximation (DCA) and its DCA$^+$ extension with a continuous self-energy. The algorithms capture nonlocal correlations in strongly correlated electron systems, thereby giving insight into high-T_c superconductivity. The code's scalability allows efficient usage of systems at all scales, from workstations to leadership computers. With regard to the increasing heterogeneity of modern computing machines, DCA++ provides portable performance on conventional and emerging new architectures, such as hybrid CPU–GPU, sustaining multiple petaflops on ORNL's Titan and CSCS' Piz Daint supercomputers. Moreover, we show how sustainable and scalable development of the code base has been achieved by adopting standard techniques of the software industry. These include employing a distributed version control system, applying test-driven development and following continuous integration.

Program summary
Program Title: DCA++
Program Files doi: http://dx.doi.org/10.17632/482jm5cv77.1
Licensing provisions: BSD-3-Clause
Programming language: C++14 and CUDA
Nature of problem: Understanding the fascinating physics of strongly correlated electron systems requires the development of sophisticated algorithms and their implementation on leadership computing systems.
Solution method: The DCA++ code provides a highly scalable and efficient implementation of the dynamical cluster approximation (DCA) and its DCA$^+$ extension.

© 2019 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Introduction

Computer simulations have become indispensable across all areas of science and physics in particular. In condensed matter physics significant computational effort is devoted to solving the quantum many-body problem of interacting electrons. Strong correlations between electrons are the origin of a variety of fascinating phenomena, most prominently high-temperature superconductivity. The enormous complexity of the quantum many-body problem – the Hilbert space grows exponentially with the system size – and lack of approximations and simplifications in the physically relevant regime pose a great challenge to date.

Progress in solving the many-electron problem has been made by introducing models that comprise the fundamental mechanisms that lead to the observed phenomena. The two-dimensional Hubbard model, for instance, is believed to capture the important physics in the superconducting cuprates [1,2]. The Hubbard Hamiltonian describes fermions on a lattice, where the particles are allowed to hop between lattice sites and interact through Coulomb repulsion when they occupy the same site. Despite the simple structure, there exists no exact solution except for the one-dimensional case.

To get insight into the physics described by the Hubbard model, one has to resort to numerical methods. Exact diagonalization (ED) and a variety of quantum Monte Carlo (QMC) algorithms solve

[☆] This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).
[*] Corresponding author.
E-mail address: haehneru@itp.phys.ethz.ch (U.R. Hähner).

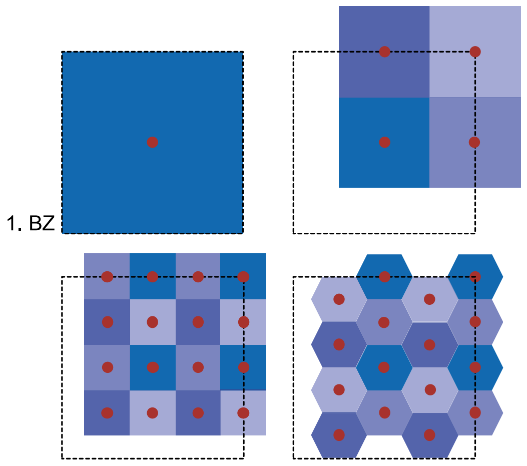


Fig. 1. (Color online) Cluster momenta **K** (red dots) and standard coarse-graining patches for different cluster sizes and shapes: $N_c = 1$ (DMFT) (top left), $N_c = 4$ (top right), $N_c = 16A$ (bottom left), and $N_c = 16B$ (bottom right).

the model on a finite-size lattice. While ED is restricted to small systems due to the exponential scaling of the problem size with the number of lattice sites, the negative sign problem prevents QMC calculations on large lattices or at low temperatures. Finite-size effects arise from the truncation of the infinite lattice to a finite number of sites.

Mean-field theories choose a different approach and are formulated in the thermodynamic limit, that is on the infinite lattice. In dynamical mean-field theory (DMFT) [3] the infinite lattice problem becomes tractable by reducing it to the self-consistent solution of an effective impurity model. Underlying the single-site approximation is the assumption that the self-energy is local in real space, which holds in the limit of infinite dimensions. However, spatially nonlocal correlations are often essential to describe phase transitions, which is the reason why DMFT has been extended by a number of quantum cluster algorithms [4]. The dynamical cluster approximation (DCA) [4–6] maps the bulk lattice problem to a finite-size periodic cluster that is self-consistently embedded in a dynamical mean-field. The effective cluster problem is best solved by means of continuous-time QMC methods [7]. Compared to finite-size QMC calculations, DCA's mean-field approach not only reduces the fermion sign problem [6], but also generally leads to results that converge faster to the thermodynamic limit [6,8].

The DCA++ project provides an efficient and highly scalable C++ implementation of the DCA algorithm and the DCA$^+$ extension that introduces a continuous lattice self-energy [9]. The code has been developed by a team at ORNL and ETH Zurich, which, besides modern software design, has the focus on developing and implementing sophisticated algorithms [10,11] and exploiting the benefits of emerging architectures, such as hybrid CPU–GPU systems. In addition to setting benchmarks for extreme scale scientific applications [12,13], the DCA++ code has provided the numerical evidence for a series of scientific publications in the field of strongly correlated electron systems. Studies of the pairing mechanism in unconventional superconductors [14] and the influence of near neighbor Coulomb repulsion on d-wave superconductivity [15], for example, were based on DCA calculations with DCA++.

In software-driven research, scientific productivity is strongly coupled to software productivity. Hence, the scientific output of a research group can be held by challenges such as new computer architectures, advanced algorithms or changing teams of developers. To prevent this productivity collapse, software development needs to be sustainable and scalable by producing comprehensible, maintainable, and extensible code. At the same time, it is essential to release changes to the code base, from new features to bug fixes, rapidly to the users, an ability that usually falls under the term *continuous delivery*. Last but not least, the scientific standard demands correctness, credibility and reproducibility of numerical results in published work. To fulfill these requirements in a challenging environment formed by complex algorithms, performance sensitive codes, the diversity of architectures, and the multidisciplinary of teams, the DCA++ project employs well-proven tools and successful techniques of the software industry [16]. While adopting these methods can require an effort, we believe that they represent a substantial factor for a research code to become a long-lived software project.

This paper accompanies the first open release of the DCA++ code and is structured as follows: Section 2 reviews the underlying DCA and DCA$^+$ algorithms, briefly discusses the continuous-time auxiliary-field and continuous-time hybridization expansion QMC algorithms, and outlines the computation of two-particle correlation functions. Section 3 gives an overview of the main components of the DCA++ code, addresses prerequisites and the CMake-based building routine, and explains how to run a DCA or DCA$^+$ calculation. Section 4 explores the software development practice that is followed in the DCA++ project, but whose applicability extends to many other collaborative scientific software efforts. Section 5 provides two typical use cases supplemented by performance analyses. The paper concludes with an outline of future development plans and a summary of ways to contribute back to the DCA++ project.

2. Formalism and methods

2.1. The dynamical cluster approximation

Like finite-size methods, the DCA algorithm replaces the infinite real-space lattice by a finite-size cluster to reduce the complexity of the problem. The reduction of the infinite lattice to a cluster of N_c sites corresponds to a discretization of the first Brillouin zone into a set of N_c cluster momenta $\mathbf{K}$ (see Fig. 1). In contrast to finite-size methods, which solve the cluster in isolation, the DCA embeds the cluster in a mean field to retain information about the degrees of freedom of the lattice that are not contained in the cluster. This is achieved through a coarse-graining procedure in momentum space, where these degrees of freedom are averaged out. For this purpose, the first Brillouin zone (BZ) is divided into N_c patches (see Fig. 1). Each patch is centered around a unique cluster momentum $\mathbf{K}$ and represented by a patch function $\phi_{\mathbf{K}}(\mathbf{k})$, the characteristic function of the patch,

$$\phi_{\mathbf{K}}(\mathbf{k}) = \begin{cases} 1, & \mathbf{k} \in \mathbf{K}^{\text{th}} \text{ patch}, \\ 0, & \text{otherwise}. \end{cases} \tag{1}$$

In addition to the orthogonality condition,

$$\frac{N_c}{V_{\text{BZ}}} \int_{\text{BZ}} d\mathbf{k}\, \phi_{\mathbf{K}}(\mathbf{k}) \phi_{\mathbf{K}'}(\mathbf{k}) = \delta_{\mathbf{K},\mathbf{K}'} \,, \tag{2}$$

we require the patches to have equal size and shape,

$$\phi_{\mathbf{K}}(\mathbf{k}) = \phi(\mathbf{k} - \mathbf{K}) \,, \tag{3}$$

and to exhibit inversion symmetry,

$$\phi(\mathbf{k} - \mathbf{K}) = \phi(\mathbf{K} - \mathbf{k}) \,. \tag{4}$$

Despite these restrictions, the choice of patches is far from unique and it is even possible to create interleaved shapes [17]. In the standard coarse-graining the patches are defined as the Brillouin zones of the superlattice as illustrated in Fig. 1.

The underlying assumption of the DCA is that the lattice self-energy $\Sigma(\mathbf{k}, i\omega_n)$ is a slowly varying function in momentum space

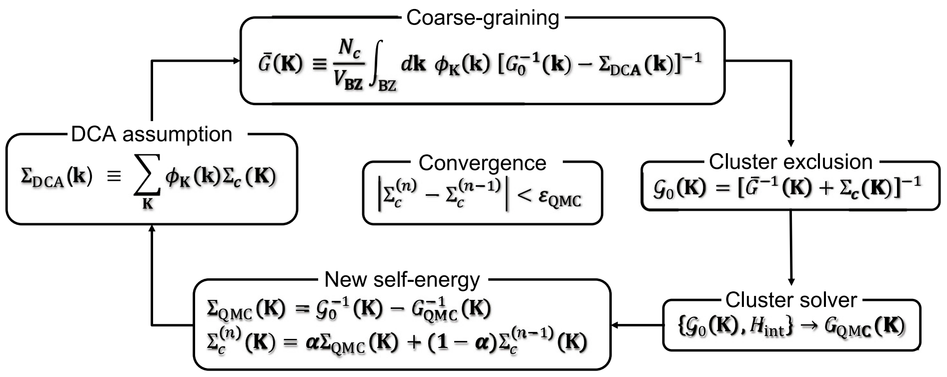

Coarse-graining
$\bar{G}(\mathbf{K}) \equiv \frac{N_c}{V_{\mathrm{BZ}}}\int_{\mathrm{BZ}} d\mathbf{k}\; \phi_{\mathbf{K}}(\mathbf{k})\,[\mathcal{G}_0^{-1}(\mathbf{k}) - \Sigma_{\mathrm{DCA}}(\mathbf{k})]^{-1}$
DCA assumption
$\Sigma_{\mathrm{DCA}}(\mathbf{k}) \equiv \sum_{\mathbf{K}} \phi_{\mathbf{K}}(\mathbf{k})\Sigma_c(\mathbf{K})$
Convergence
$\left|\Sigma_c^{(n)} - \Sigma_c^{(n-1)}\right| < \varepsilon_{\mathrm{QMC}}$
Cluster exclusion
$\mathcal{G}_0(\mathbf{K}) = [\bar{G}^{-1}(\mathbf{K}) + \Sigma_c(\mathbf{K})]^{-1}$
New self-energy
$\Sigma_{\mathrm{QMC}}(\mathbf{K}) = \mathcal{G}_0^{-1}(\mathbf{K}) - G_{\mathrm{QMC}}^{-1}(\mathbf{K})$
$\Sigma_c^{(n)}(\mathbf{K}) = \alpha\Sigma_{\mathrm{QMC}}(\mathbf{K}) + (1-\alpha)\Sigma_c^{(n-1)}(\mathbf{K})$
Cluster solver
$\{\mathcal{G}_0(\mathbf{K}), H_{\mathrm{int}}\} \to G_{\mathrm{QMC}}(\mathbf{K})$

Fig. 2. The DCA self-consistency loop: In accordance with the DCA assumption, the lattice self-energy $\Sigma_{\mathrm{DCA}}(\mathbf{k})$ is constructed as a piecewise constant continuation of the cluster self-energy $\Sigma_c(\mathbf{K})$. The former is then used to calculate the coarse-grained Green's function $\bar{G}(\mathbf{K})$. Given $\bar{G}(\mathbf{K})$ and $\Sigma_c(\mathbf{K})$, the bare cluster Green's function $\mathcal{G}_0(\mathbf{K})$ is obtained from the Dyson equation. $\mathcal{G}_0(\mathbf{K})$ complemented by the interaction Hamiltonian H_{int} defines the effective cluster problem, which is solved, in general, by means of QMC techniques. The self-consistency loop is closed by producing a new cluster self-energy $\Sigma_c(\mathbf{K})$.

and in the vicinity of a cluster momentum $\mathbf{K}$ well approximated by the self-energy of the cluster $\Sigma_c(\mathbf{K}, i\omega_n)$,

$$\Sigma(\mathbf{k} = \mathbf{K} + \tilde{\mathbf{k}}, i\omega_n) \approx \Sigma_c(\mathbf{K}, i\omega_n)\,. \tag{5}$$

Based on this assumption, we can approximate the lattice self-energy $\Sigma(\mathbf{k}, i\omega_n)$ by a piecewise constant continuation of the cluster self-energy $\Sigma_c(\mathbf{K}, i\omega_n)$ by means of the patch functions $\phi_{\mathbf{K}}(\mathbf{k})$,

$$\Sigma_{\mathrm{DCA}}(\mathbf{k}, i\omega_n) \equiv \sum_{\mathbf{K}} \phi_{\mathbf{K}}(\mathbf{k})\, \Sigma_c(\mathbf{K}, i\omega_n)\,. \tag{6}$$

Unlike finite-size methods, the DCA, using above approximation of the lattice self-energy, replaces the Green's function of the cluster $G_c(\mathbf{K}, i\omega_n)$ with the coarse-grained average of the lattice Green's function,

$$\begin{aligned} G_c(\mathbf{K}, i\omega_n) &\to \bar{G}(\mathbf{K}, i\omega_n) \\ &= \frac{N_c}{V_{\mathrm{BZ}}} \int_{\mathrm{BZ}} d\mathbf{k}\, \phi_{\mathbf{K}}(\mathbf{k}) \left[G_0^{-1}(\mathbf{k}, i\omega_n) - \Sigma_{\mathrm{DCA}}(\mathbf{k}, i\omega_n)\right]^{-1}\,. \end{aligned} \tag{7}$$

$G_0(\mathbf{k}, i\omega_n)$ is the non-interacting lattice Green's function given by

$$G_0(\mathbf{k}, i\omega_n) = [i\omega_n - \varepsilon_{\mathbf{k}} + \mu]^{-1}\,. \tag{8}$$

To formulate the effective cluster problem of the DCA, we need the corresponding bare propagator of the cluster $\mathcal{G}_0(\mathbf{K}, i\omega_n)$. We obtain it by reversing the Dyson equation in the *cluster-exclusion step*,

$$\mathcal{G}_0(\mathbf{K}, i\omega_n) = \left[\bar{G}^{-1}(\mathbf{K}, i\omega_n) + \Sigma_c(\mathbf{K}, i\omega_n)\right]^{-1}\,. \tag{9}$$

$\mathcal{G}_0(\mathbf{K}, i\omega_n)$ is fundamentally different from the free lattice propagator evaluated at the cluster momenta $\mathbf{K}$, $G_0(\mathbf{K}, i\omega_n)$. It incorporates the coupling of the cluster to the mean-field that represents the remaining degrees of freedom of the lattice. The effective cluster problem is completed by the interacting part of the Hamiltonian H_{int} and best solved by QMC methods to calculate a new cluster self-energy $\Sigma_c(\mathbf{K}, i\omega_n)$,

$$\{\mathcal{G}_0(\mathbf{K}, i\omega_n), H_{\mathrm{int}}\} \xrightarrow{\mathrm{QMC}} \Sigma_c(\mathbf{K}, i\omega_n)\,. \tag{10}$$

This closes the DCA self-consistency loop.

To start the loop, one has to provide an initial cluster self-energy $\Sigma_c(\mathbf{K}, i\omega_n)$, which in practice is either zero or the result of a previous calculation for a slightly higher temperature (see Section 3.3). Steps (6), (7), (9) and (10) are then iterated until convergence is reached, where the criterion for convergence is dictated by the stochastic error of the QMC solver $\varepsilon_{\mathrm{QMC}}$,

$$|\Sigma_c^{(n)} - \Sigma_c^{(n-1)}| < \varepsilon_{\mathrm{QMC}}\,. \tag{11}$$

To stabilize convergence of the loop, the self-energy produced by the QMC solver $\Sigma_{\mathrm{QMC}}(\mathbf{K}, i\omega_n)$ is often mixed with the previous value of the cluster self-energy $\Sigma_c(\mathbf{K}, i\omega_n)$,

$$\Sigma_c^{(n)} = \alpha\, \Sigma_{\mathrm{QMC}} + (1-\alpha)\, \Sigma_c^{(n-1)}\,, \quad 0 < \alpha \le 1\,. \tag{12}$$

The complete DCA algorithm is summarized in Fig. 2.

Relying on QMC methods, the DCA algorithm inherits their negative sign problem. Nevertheless, the sign problem turns out to be less severe than in finite-size lattice QMC simulations, an effect the dynamical mean-field has been credited with [6].

Like finite-size methods, the DCA yields the exact result of the infinite lattice problem as $N_c \to \infty$. Consequently, if clusters large enough to extrapolate to infinite cluster size are accessible, we can obtain exact results for the thermodynamic limit by finite-size scaling. In doing so, convergence with respect to cluster size indicates that the current size of the cluster captures all relevant nonlocal correlations. In the opposite limit, i.e. for $N_c = 1$, the DCA recovers DMFT.

2.2. The DCA$^+$ algorithm

The cluster shape dependence represents one of the major drawbacks of the DCA algorithm (see Fig. 4). The expansion of the lattice self-energy $\Sigma_{\mathrm{DCA}}(\mathbf{k}, i\omega_n)$, Eq. (6), evidently depends on the shape of the coarse-graining patches and is therefore sensitive to the definition of the cluster. A way to cure this problem is to abandon the representation of the lattice self-energy in terms of step functions and find a continuous description. Simple interpolation schemes of the cluster self-energy, however, have been shown to lead to causality violations [5]. Instead, we start from a coarse-graining equation for the self-energy, equivalent to the expression for the Green's function in Eq. (7). Inverting Eq. (6) with the help of the orthogonality condition of the patch functions, Eq. (2), we obtain the relation,

$$\Sigma_c(\mathbf{K}, i\omega_n) = \frac{N_c}{V_{\mathrm{BZ}}} \int_{\mathrm{BZ}} d\mathbf{k}\, \phi_{\mathbf{K}}(\mathbf{k})\, \Sigma_{\mathrm{DCA}^+}(\mathbf{k}, i\omega_n)\,. \tag{13}$$

This equation implies that we need to generate a lattice self-energy $\Sigma_{\mathrm{DCA}^+}(\mathbf{k}, i\omega_n)$ whose coarse-grained average equals the cluster self-energy $\Sigma_c(\mathbf{K}, i\omega_n)$. Note that the piecewise constant form of the standard DCA algorithm, Eq. (6), trivially satisfies this condition. Finding the lattice self-energy $\Sigma_{\mathrm{DCA}^+}(\mathbf{k}, i\omega_n)$ in DCA$^+$ is a two step procedure. First, we interpolate the cluster self-energy $\Sigma_c(\mathbf{K}, i\omega_n)$ between the cluster momenta $\mathbf{K}$,

$$\left\{\mathbf{K}, \Sigma_c(\mathbf{K}, i\omega_n)\right\} \to \left\{\mathbf{k}, \tilde{\Sigma}(\mathbf{k}, i\omega_n)\right\}, \tag{14a}$$

$$\text{with} \quad \tilde{\Sigma}(\mathbf{K}, i\omega_n) \equiv \Sigma_c(\mathbf{K}, i\omega_n)\,. \tag{14b}$$

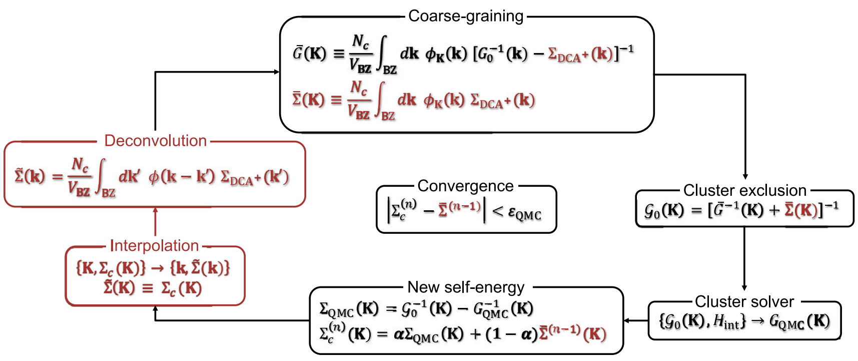

Coarse-graining
$\bar{G}(\mathbf{K}) \equiv \frac{N_c}{V_{BZ}}\int_{BZ} d\mathbf{k}\ \phi_{\mathbf{K}}(\mathbf{k})\ [G_0^{-1}(\mathbf{k}) - \Sigma_{DCA^+}(\mathbf{k})]^{-1}$
$\bar{\Sigma}(\mathbf{K}) \equiv \frac{N_c}{V_{BZ}}\int_{BZ} d\mathbf{k}\ \phi_{\mathbf{K}}(\mathbf{k})\ \Sigma_{DCA^+}(\mathbf{k})$
Deconvolution
$\tilde{\Sigma}(\mathbf{k}) = \frac{N_c}{V_{BZ}}\int_{BZ} d\mathbf{k}'\ \phi(\mathbf{k}-\mathbf{k}')\ \Sigma_{DCA^+}(\mathbf{k}')$
Convergence
$|\Sigma_c^{(n)} - \bar{\Sigma}^{(n-1)}| < \epsilon_{QMC}$
Cluster exclusion
$\mathcal{G}_0(\mathbf{K}) = [\bar{G}^{-1}(\mathbf{K}) + \bar{\Sigma}(\mathbf{K})]^{-1}$
Interpolation
$\{\mathbf{K}, \Sigma_c(\mathbf{K})\} \rightarrow \{\mathbf{k}, \tilde{\Sigma}(\mathbf{k})\}$
$\tilde{\Sigma}(\mathbf{K}) \equiv \Sigma_c(\mathbf{K})$
New self-energy
$\Sigma_{QMC}(\mathbf{K}) = \mathcal{G}_0^{-1}(\mathbf{K}) - G_{QMC}^{-1}(\mathbf{K})$
$\Sigma_c^{(n)}(\mathbf{K}) = \alpha\Sigma_{QMC}(\mathbf{K}) + (1-\alpha)\bar{\Sigma}^{(n-1)}(\mathbf{K})$
Cluster solver
$\{\mathcal{G}_0(\mathbf{K}), H_{int}\} \rightarrow G_{QMC}(\mathbf{K})$

Fig. 3. (Color online) The DCA$^+$ self-consistency loop with differences to the standard DCA algorithm highlighted in red. Due to the non-trivial lattice mapping consisting of an interpolation followed by a deconvolution, the DCA$^+$ algorithm distinguishes between cluster and coarse-grained self-energy, $\Sigma_c(\mathbf{K}, i\omega_n)$ and $\bar{\Sigma}(\mathbf{K}, i\omega_n)$, respectively . (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

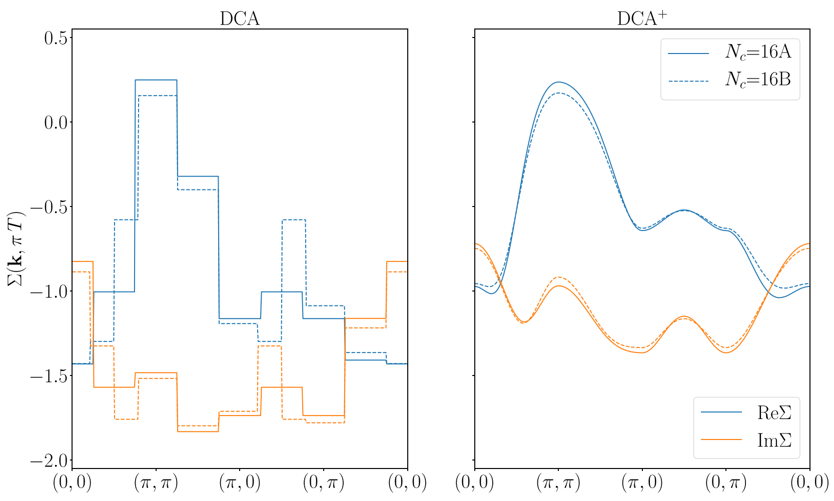


Fig. 4. (Color online) Momentum dependence of the DCA (left) and DCA$^+$ (right) lattice self-energy at the lowest Matsubara frequency for a 2D Hubbard model, Eq. (45), with $U = 7t$ and $\langle n\rangle = 0.95$ at a temperature $T = 0.2t$. Results are shown for the two different 16-site clusters of Fig. 1.

Next, we extend the coarse-graining condition, Eq. (13), to the interpolated lattice self-energy $\tilde{\Sigma}(\mathbf{k}, i\omega_n)$,

$$\tilde{\Sigma}(\mathbf{k}, i\omega_n) = \frac{N_c}{V_{BZ}}\int_{BZ} d\mathbf{k}'\, \phi(\mathbf{k}-\mathbf{k}')\, \Sigma_{DCA^+}(\mathbf{k}', i\omega_n)\,, \tag{15}$$

where we used the translation property, Eq. (3), and inversion symmetry, Eq. (4), of the patch function $\phi_{\mathbf{K}}(\mathbf{k})$. According to Eq. (15), $\tilde{\Sigma}(\mathbf{k}, i\omega_n)$ is related to $\Sigma_{DCA^+}(\mathbf{k}, i\omega_n)$ by a convolution with the patch function $\phi(\mathbf{k}-\mathbf{k}')$. Consequently, the lattice self-energy $\Sigma_{DCA^+}(\mathbf{k}, i\omega_n)$ is determined by solving the deconvolution problem. We want to point out that the convergence of both the interpolation and the deconvolution depends on the size of the real-space cluster. More precisely, if the assumption already the DCA algorithm is based on is violated, i.e. the self-energy of the lattice is longer-ranged, the lattice mapping problem, $\Sigma_c(\mathbf{K}, i\omega_n) \rightarrow \Sigma_{DCA^+}(\mathbf{k}, i\omega_n)$, is ill-defined and convergence poor [9]. This sensitivity to the behavior of the real-space self-energy defines the current limits of the DCA$^+$ algorithm.

From $\Sigma_{DCA^+}(\mathbf{k}, i\omega_n)$ we again calculate the coarse-grained Green's function $\bar{G}(\mathbf{K}, i\omega_n)$ and this time also a coarse-grained self-energy $\bar{\Sigma}(\mathbf{K}, i\omega_n)$, which should be close to the cluster self-energy $\Sigma_c(\mathbf{K}, i\omega_n)$,

$$\bar{G}(\mathbf{K}, i\omega_n) \equiv \frac{N_c}{V_{BZ}}\int_{BZ} d\mathbf{k}\, \phi_{\mathbf{K}}(\mathbf{k})\, \left[G_0^{-1}(\mathbf{k}, i\omega_n) - \Sigma_{DCA^+}(\mathbf{k}, i\omega_n)\right]^{-1}\,, \tag{16a}$$

$$\bar{\Sigma}(\mathbf{K}, i\omega_n) \equiv \frac{N_c}{V_{BZ}}\int_{BZ} d\mathbf{k}\, \phi_{\mathbf{K}}(\mathbf{k})\, \Sigma_{DCA^+}(\mathbf{k}, i\omega_n)\,. \tag{16b}$$

The coarse-grained self-energy $\bar{\Sigma}(\mathbf{K}, i\omega_n)$ enters the calculation of the bare Green's function of the effective cluster problem in DCA$^+$,

$$\mathcal{G}_0(\mathbf{K}, i\omega_n) = \left[\bar{G}^{-1}(\mathbf{K}, i\omega_n) + \bar{\Sigma}(\mathbf{K}, i\omega_n)\right]^{-1}\,. \tag{17}$$

As in the DCA algorithm, the QMC cluster solver closes the self-consistency loop by generating a new cluster self-energy $\Sigma_c(\mathbf{K}, i\omega_n)$. The full DCA$^+$ algorithm is illustrated in Fig. 3, in which we also point out the differences to the standard DCA loop.

Distinguishing the coarse-grained self-energy $\bar{\Sigma}(\mathbf{K}, i\omega_n)$ from the cluster self-energy $\Sigma_c(\mathbf{K}, i\omega_n)$, the DCA$^+$ algorithm requires a modified convergence criterion. Self-consistency is reached when the QMC solver produces a cluster self-energy $\Sigma_c^{(n)}$ that agrees, within the statistical Monte Carlo error ε_{QMC}, with the coarse-grained self-energy $\bar{\Sigma}^{(n-1)}$ used to set up the cluster problem,

$$|\Sigma_c^{(n)} - \bar{\Sigma}^{(n-1)}| < \varepsilon_{QMC}\,. \tag{18}$$

The benefits of the DCA$^+$ algorithm in terms of full momentum resolution and independence of the results on the cluster shape are illustrated in Fig. 4, where the momentum dependence of the DCA and DCA$^+$ lattice self-energy is shown for the two different 16-site clusters of Fig. 1. The DCA self-energy is characterized by jump discontinuities between the coarse-graining patches and shows a strong cluster shape dependence in the region between $(\pi, 0)$ and $(0, \pi)$. In contrast, the DCA$^+$ algorithm provides a self-energy with smooth momentum dependence and results for the two clusters that agree well. In addition to curing the cluster shape dependence, the DCA$^+$ algorithm further reduces the negative sign problem of the underlying QMC method. This was attributed to the removal of the apparent discontinuities in the DCA self-energy, which had caused artificial long-range correlations [9].

2.3. Continuous-time auxiliary-field quantum Monte Carlo

Developed by Gull et al. [18] specifically for quantum impurity problems, the continuous-time auxiliary field QMC algorithm (CT-AUX) is the state of the art method to solve the effective cluster problem in DCA and DCA$^+$ calculations. CT-AUX can treat Hamiltonians containing all types of density–density interactions, including spatially non-local Coulomb repulsion and multiple orbital models, but the algorithm is most useful for large cluster calculations of the single-band Hubbard model. Instead of describing the algorithm at full length, we want to focus on the relevant parts for a DCA++ user and refer the interested reader to the original paper as well as the detailed review on continuous-time QMC methods given in Ref. [7].

We consider a generalized Hubbard model given by the Hamiltonian

$$H = H_0 + H_{\text{int}}\,, \tag{19}$$

$$H_0 = \sum_{ij,\sigma} t_{ij} c^\dagger_{i\sigma} c_{j\sigma} - \mu \sum_{\mu} n_\mu + \frac{1}{2}\sum_{\mu>\nu} U_{\mu\nu}\left(n_\mu + n_\nu\right)\,, \tag{20}$$

$$H_{\text{int}} = \sum_{\mu>\nu} U_{\mu\nu}\left[n_\mu n_\nu - \frac{1}{2}\left(n_\mu + n_\nu\right)\right]. \tag{21}$$

The indices i and j run over sites and orbitals to allow for the most general type of electron hopping. μ and ν represent combined indices of spin, orbital and site, and the sum $\sum_{\mu>\nu}$ runs over all pairs of correlated spin–orbitals ($U_{\mu\nu} \neq 0$), whose number we denote by N_{corr}. Note that for the traditional single-band Hubbard model with only on-site Coulomb repulsion, the number of correlated orbitals is just the number of sites, $N_{\text{corr}} = N_c$. Based on this decomposition of the Hamiltonian H into a non-interacting part H_0 and the interaction term H_{int}, we can formulate the partition function $Z = \text{Tr}[e^{-\beta H}]$ in the corresponding interaction representation,

$$Z = e^{-K}\,\text{Tr}\left[e^{-\beta H_0} T_\tau e^{-\int_0^\beta d\tau\,(H_{\text{int}} - K/\beta)}\right]. \tag{22}$$

The time-ordered exponential is now expanded with the parameter K, introduced in Eq. (22), controlling the expansion order,

$$\begin{aligned} Z = e^{-K} \sum_{k=0}^{\infty} \left(\frac{K}{\beta}\right)^k \int_0^\beta d\tau_1 \ldots \int_{t_{k-1}}^\beta d\tau_k\,\text{Tr} \\ \times \left[e^{-(\beta-\tau_k+\tau_1)H_0}\left(1 - \frac{\beta H_{\text{int}}(\tau_k)}{K}\right)\ldots e^{-(\tau_2-\tau_1)H_0} \right. \\ \left. \times \left(1 - \frac{\beta H_{\text{int}}(\tau_1)}{K}\right)\right]. \end{aligned} \tag{23}$$

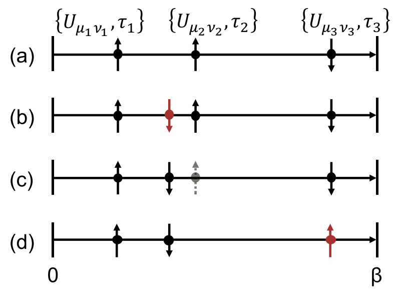
The interaction terms $(1 - \beta H_{\text{int}}(\tau)/K)$ are decoupled with the help of auxiliary spins s_i [19],

$$1 - \frac{\beta H_{\text{int}}}{K} = \frac{1}{2N_{\text{corr}}} \sum_{\mu>\nu} \sum_{s=\pm 1} e^{\gamma_{\mu\nu} s (n_\mu - n_\nu)}\,, \tag{24}$$

$$\cosh(\gamma_{\mu\nu}) = 1 + \frac{U_{\mu\nu}\beta N_{\text{corr}}}{2K}\,. \tag{25}$$

As a result, the partition function Z becomes a high-dimensional integral,

$$Z = e^{-K} \sum_{k=0}^{\infty} \sum_{\mu>\nu} \sum_{s_1,\ldots,s_k=\pm 1} \int_0^\beta d\tau_1 \ldots \int_{\tau_{k-1}}^\beta d\tau_k$$



$\{U_{\mu_1\nu_1}, \tau_1\}$
$\{U_{\mu_2\nu_2}, \tau_2\}$
$\{U_{\mu_3\nu_3}, \tau_3\}$
(a)
(b)
(c)
(d)
0
β

Fig. 5. (Color online) CT-AUX vertex configuration of a $k = 3$ order term (a) and the three types of single spin updates: (a) to (b) vertex insertion, (b) to (c) vertex removal, (c) to (d) flip of an auxiliary spin.

$$\times \left(\frac{K}{2\beta N_{\text{corr}}}\right)^k Z_k(\{\mu, \nu, \tau, s\}_k)\,, \tag{26}$$

$$\begin{aligned} Z_k(\{\mu, \nu, \tau, s\}_k) &= \text{Tr} \prod_{i=1}^{k} e^{-\Delta\tau_i H_0} e^{\gamma_{\mu\nu} s (n_\mu - n_\nu)} \\ &= Z_0 \prod_\sigma \det N_\sigma^{-1}(\{\mu, \nu, \tau, s,\}_k)\,, \end{aligned} \tag{27}$$

which can be evaluated by a Markov Chain Monte Carlo procedure.

Following the Metropolis algorithm, we sample the configuration space $\mathcal{C}$, in which an element $c \in \mathcal{C}$ consists of a set of k vertices, $\{v = (\mu_i, \mu_j, \tau, s)\}_k$. Each vertex possesses an auxiliary spin $s = \pm 1$ and represents an interaction between two correlated spin–orbitals, $\mu_i = \{\sigma_i, \alpha_i, \mathbf{r}_i\}$ and $\mu_j = \left\{\sigma_j, \alpha_j, \mathbf{r}_j\right\}$, acting at a time $\tau \in [0, \beta)$. A configuration $c \in \mathcal{C}$ in the Markov chain is updated by inserting a new vertex, removing an existing vertex, or flipping an auxiliary spin. These single spin updates correspond to rank-1 operations on the N_σ-matrices defined in Eq. (27). The dimension of the $N_\uparrow(N_\downarrow)$-matrix is given by the number of $\uparrow$ $(\downarrow)$-electron spins contained in the current configuration. If interactions only occur between spins of opposite type, both $N_\uparrow$ and $N_\downarrow$ are $k \times k$ matrices. Fig. 5 shows a typical configuration of auxiliary spins on the imaginary time axis and the three types of single spin updates.

While we sample configurations on the imaginary time axis, measuring the Green's function is most accurately performed in discrete Matsubara frequency space. Time measurements would lead to binning errors as the vertices can occupy arbitrary positions in the imaginary time interval. From a vertex configuration $\{v\}_k$ and its respective N_σ-matrices we first compute a function M, whose Fourier coefficients are then accumulated. The Green's function is linearly related to M,

$$\langle G(\mathbf{k}, i\omega_n)\rangle = \mathcal{G}_0(\mathbf{k}, i\omega_n) - \frac{1}{\beta}\mathcal{G}_0(\mathbf{k}, i\omega_n)\,\langle M(\mathbf{k}, i\omega_n)\rangle\,\mathcal{G}_0(\mathbf{k}, i\omega_n)\,. \tag{28}$$

The self-energy $\Sigma(\mathbf{k}, i\omega_n)$ is finally gained from the measured Green's function $\langle G(\mathbf{k}, i\omega_n)\rangle$ and the bare propagator of the cluster impurity problem $\mathcal{G}_0(\mathbf{k}, i\omega_n)$ using the Dyson equation,

$$\Sigma(\mathbf{k}, i\omega_n) = \mathcal{G}_0^{-1}(\mathbf{k}, i\omega_n) - \langle G(\mathbf{k}, i\omega_n)\rangle^{-1}\,. \tag{29}$$

In addition to the single-particle Green's function $G(\mathbf{k}, i\omega_n)$, also the two-particle Green's function can be obtained within the CT-AUX algorithm. The measurement of the latter involves significantly higher computational cost, though.

The CT-AUX algorithm scales as the cube of the linear size of the N_σ-matrices, i.e. $\mathcal{O}(\langle k\rangle^3)$. The average expansion order $\langle k\rangle$, in turn, grows linearly with the expansion parameter K, the inverse temperature β, the strength the interactions $U_{\mu\nu}$, and the number of interaction terms N_{corr}. In simulations of the Hubbard model away from half-filling, CT-AUX exhibits a negative sign problem. The signal-to-noise ratio decreases exponentially when the system size or the strength of the Coulomb interaction is increased, or the temperature lowered. Therefore, the behavior of both the average expansion order and the sign problem makes CT-AUX calculations of large systems, with strong coupling and at low temperatures computationally most demanding.

Submatrix updates. An important improvement in terms of computational efficiency of the CT-AUX algorithm has been achieved with the introduction of *submatrix updates* [20]. Combining k_s successive single spin updates into a rank-k_s operation, submatrix updates substantially improve data locality and thus benefit from the cache hierarchy of modern processors.

The formalism of submatrix updates assumes that individual spins are changed only once within a submatrix update and spins that do not satisfy this condition require special treatment at the cost of a small performance loss. One can either update these spins with the help of Bennett's algorithm [21] or abort the current submatrix update when a spin is selected twice and propose the same spin again at the beginning of the next update. For systems with large expansion orders, for which the probability of touching the same spin twice within a submatrix update is small, ignoring proposals of already chosen spins is a valid option, too. This approximation avoids the performance loss, but violates the detailed balance condition.

Non-equidistant fft measurements. A remarkable speed-up in the measurement of single- as well as the two-particle functions has been accomplished by developing optimized non-equispaced fast Fourier transform (NFFT) methods [11]. In the context of CT-AUX, the methods are used to measure and accumulate the M function, but their applicability extends to other continuous-time QMC methods as well.

Since vertex configurations are not sampled on an equispaced grid but at random time locations, we cannot directly apply the fast Fourier transform (FFT) algorithm on the raw samples. To circumvent this problem, the NFFT algorithm [22] makes use of the convolution theorem,

$$f_\omega \varphi_\omega = \int_0^\beta d\tau\, e^{i\omega\tau} \left[\int_0^\beta d\tau'\, \varphi(\tau - \tau') f(\tau') \right] . \tag{30}$$

Given samples f_i at random times $\tau_i \in [0, \beta)$, we first transform the data-set onto an equispaced imaginary time grid by convoluting it with a localized, β-periodic kernel φ. We are then able to use the FFT algorithm to compute the Fourier transform of the convoluted data. According to Eq. (30), we obtain the Fourier components f_ω of the original data after normalizing with the Fourier coefficients of the kernel φ_ω, which finalizes the NFFT algorithm,

$$f_\omega \leftarrow \frac{1}{\varphi_\omega} \mathrm{FFT} \left[\left\{ \bar{f}_l = \sum_i \varphi(\tau_i - \frac{l\beta}{mN_\omega}) f_i \mid l \in 0, 1, \ldots, mN_\omega - 1 \right\} \right] . \tag{31}$$

In Eq. (31), N_ω denotes the desired number of positive Matsubara frequencies and m is an oversampling factor.

The linearity of the Fourier transform allows us to delay it, meaning we can accumulate the equispaced data points $\bar{f}_l$ and perform a single FFT on the accumulated data at the end of the Monte Carlo integration. The algorithm can be further accelerated by approximating the kernel φ with a low order polynomial at the cost of a controllable interpolation error.

While the delayed-NFFT scheme cannot be applied to the measurement of the two-particle Green's function as Fourier transformation and accumulation are not interchangeable there, a specifically tuned 2D NFFT algorithm can still lead to a considerable speed-up compared to the standard 2D non-equidistant discrete Fourier transform [11].

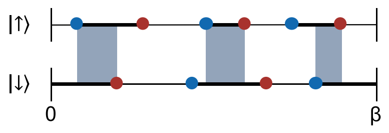


Fig. 6. (Color online) A segment-CT-HYB configuration of a $k = 5$ order term for a single-orbital impurity. Thick lines represent segments, which are bounded by creation (blue dots) and annihilation operators (red dots). As shown for the spin-down orbital, segments can wrap around β. Shaded areas indicate time intervals in which both spin–orbitals are occupied and the Coulomb interaction U is acting between them. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

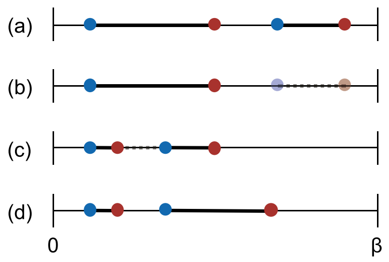


Fig. 7. (Color online) Three examples of segment updates: (a) to (b) segment removal, (b) to (c) anti-segment insertion, (c) to (d) shift of segment endpoint.

2.4. Continuous-time hybridization expansion quantum Monte Carlo

With the continuous-time hybridization expansion QMC algorithm (CT-HYB) [23,24], DCA++ provides an alternative continuous-time impurity solver. While CT-AUX is the favorable algorithm for large cluster DCA and DCA$^+$ simulations, CT-HYB is the method of choice for single-site multi-orbital impurity models in the strong coupling regime ($U/t \approx 10 - \infty$). The CT-HYB algorithm is capable of treating general interactions, but provides a particularly efficient *segment formulation* for density–density interactions. We will restrict ourselves to this formulation and again refer readers to the original papers and Ref. [7] for a complete discussion.

We consider a quantum impurity model, which describes a small system, the impurity, embedded in an infinite non-interacting bath,

$$H = H_{\mathrm{imp}} + H_{\mathrm{bath}} + H_{\mathrm{hyb}} . \tag{32}$$

According to its name, the CT-HYB algorithm is based on an expansion of the partition function Z in the hybridization H_{hyb}, which couples the impurity to the bath. In analogy to the procedure in CT-AUX, the partition function is transformed into a high-dimensional integral, which is sampled by means of a Markov chain Monte Carlo simulation. In the segment picture the configuration space $\mathcal{C}$ is represented by multiple imaginary time lines. Each line spans the interval $[0, \beta)$ and corresponds to a flavor (spin, orbital, site) of the impurity. A configuration $c \in \mathcal{C}$ consists of a set of segments on these time lines during which the corresponding spin–orbitals are occupied. For the Markov chain to be ergodic, two types of updates are needed: segment insertions and removals. Additional updates in form of insertion and removal of "anti-segments" and segment shifts, however, can improve the sampling efficiency. Segment swaps, which exchange the time lines of two flavors, may be necessary in certain cases. A typical configuration and examples of segment updates are illustrated in Figs. 6 and 7, respectively.

The CT-HYB algorithm accumulates the Matsubara coefficients of the Green's function $G(i\omega_n)$. In addition, it has proven useful to measure $G(i\omega_n)\Sigma(i\omega_n)$. Although it is not necessary to accumulate this product, since the self-energy can be obtained from the Dyson equation (29), the approach leads to much smaller statistical errors [25].

For a diagonal hybridization term H_{hyb}, the CT-HYB algorithm in the segment formulation is sign problem free and scales as $\mathcal{O}(N\beta^3)$, where N is the number of spin-degenerate orbitals.

2.5. Calculation of two-particle correlation functions

A possible approach to study phase transitions, for example to the superconducting state, is the extension of the DCA and DCA$^+$ frameworks to the two-particle level. In this section we want to briefly outline this formalism for the case of the particle–particle channel and refer to the article by Jarrel et al. [6] for a more detailed derivation in the DCA framework, to the work of Staar, Maier, and Schulthess [26] for the extension to DCA$^+$, and to the didactic summary by Maier [27], which we follow here.

DCA. We start from the Bethe–Salpeter equation (BSE), which in the particle–particle channel (abbreviated with "pp") is given by

$$\begin{aligned} G_2(k, q-k, q-k', k') &= G_\uparrow(k)G_\downarrow(q-k)\delta_{k,k'} \\ &- \frac{T}{N}\sum_{k''} G_\uparrow(k)G_\downarrow(q-k)\,\Gamma^{pp}(k, q-k, q-k'', k'') \\ &\times G_2(k'', q-k'', q-k', k')\,. \end{aligned} \tag{33}$$

It relates the two-particle Green's function G_2 to the irreducible vertex function of this channel Γ^{pp}, and can be regarded as the two-particle level analog of the Dyson equation. In Eq. (33) we introduced a combined index for momentum and Matsubara frequency, $k \equiv (\mathbf{k}, i\omega_n)$, with the corresponding summation, $\frac{T}{N}\sum_k \equiv \frac{T}{V_{\text{BZ}}}\int d\mathbf{k}\sum_{i\omega_n}$. The variable $q \equiv (\mathbf{q}, i\nu_m)$ represents the transferred momentum and transferred (bosonic) frequency. Applying the underlying assumption of the DCA at the two-particle level, we expand the irreducible vertex function of the cluster onto the coarse-graining patches to obtain a piecewise constant approximation of the lattice irreducible vertex function,

$$\Gamma^{pp}_{\text{DCA}}(k, k', q) \equiv \sum_{\mathbf{K},\mathbf{K}'} \phi_{\mathbf{K}}(\mathbf{k})\,\Gamma^{pp}_c(K, K', q)\,\phi_{\mathbf{K}'}(\mathbf{k}')\,, \tag{34}$$

where $\Gamma^{pp}(k, k', q) \equiv \Gamma^{pp}(k, q-k, q-k', k')$ was used to simplify the notation. The cluster vertex function $\Gamma^{pp}_c(K, K', q)$, in turn, satisfies the cluster version of the Bethe–Salpeter equation (33),

$$\begin{aligned} G_{2,c}(K, K', q) &= G^{0,pp}_{2,c}(K, K', q) - \frac{T}{N_c}\sum_{K''} G^{0,pp}_{2,c}(K, K', q) \\ &\times \Gamma^{pp}_c(K, K'', q)\,G_{2,c}(K'', K', q)\,, \end{aligned} \tag{35}$$

with the non-interacting two-particle Green's function $G^{0,pp}_{2,c}(K, K', q) \equiv G_{c,\uparrow}(K)G_{c,\downarrow}(q-K)\delta_{K,K'}$. Solving for the irreducible vertex function yields (matrix notation in K, K' for fixed q),

$$\boldsymbol{\Gamma}^{pp}_c = -\frac{N_c}{T}\left[\left(\mathbf{G}^0_{2,c}\right)^{-1} - \left(\mathbf{G}_{2,c}\right)^{-1}\right]. \tag{36}$$

The single- and two-particle Green's functions are obtained from the cluster solver.

Possible phase transitions can be discovered from the eigenvalues and left and right eigenvectors of the Bethe–Salpeter kernel,

$$-\frac{T}{N}\sum_k \tilde{g}_\alpha(k)\,\Gamma^{pp}(k, k', q)G_\uparrow(k')G_\downarrow(q-k') = \lambda_\alpha\tilde{g}_\alpha(k')\,, \tag{37}$$

$$-\frac{T}{N}\sum_{k'} \Gamma^{pp}(k, k', q)G_\uparrow(k')G_\downarrow(q-k')\,g_\alpha(k') = \lambda_\alpha g_\alpha(k)\,, \tag{38}$$

where we dropped the dependence of λ_α, $g_\alpha(k)$ and $\tilde{g}_\alpha(k)$ on the transferred momentum and frequency q, as well as on the considered channel. This can be seen when the two-particle Green's function is expressed in terms of these eigenvalues and eigenvectors,

$$G^{pp}_2(k, k', q) = \sum_\alpha G_\uparrow(k)G_\downarrow(q-k)\frac{g_\alpha(k)\tilde{g}_\alpha(k')}{1-\lambda_\alpha}\,. \tag{39}$$

The eigenvalues λ_α indicate an instability of the system: Eq. (39) diverges when the leading eigenvalue approaches 1. The eigenvectors $g_\alpha(k)$ and $\tilde{g}_\alpha(k)$ describe the momentum and frequency structure of the instability.

By using the approximation of the lattice irreducible vertex function by step functions, Eq. (34), and restricting the momentum resolution of the right eigenvectors $g_\alpha(\mathbf{k}, i\omega_n)$ to the cluster momenta $\mathbf{K}$, we can transform Eq. (38) into a coarse-grained eigenvalue equation,

$$-\frac{T}{N_c}\sum_{K'} \Gamma^{pp}_c(K, K', q)\,\chi^{pp}_0(K', q)\,g_\alpha(K') = \lambda_\alpha g_\alpha(K)\,, \tag{40}$$

with

$$\chi^{pp}_0(K, q) \equiv \frac{N_c}{V_{\text{BZ}}}\int d\mathbf{k}\,\phi_{\mathbf{K}}(\mathbf{k})\,G_\uparrow(k)G_\downarrow(q-k)\,. \tag{41}$$

Eq. (40) is easily solvable by standard diagonalization techniques.

DCA$^+$. To restore the momentum resolution of the lattice, we need to find a lattice irreducible vertex function $\Gamma^{pp}(k, k', q)$ that is continuous in $\mathbf{k}$ and $\mathbf{k}'$. In analogy to the derivation of DCA$^+$ in Section 2.2, we start from a coarse-graining equation for the vertex function,

$$\Gamma^{pp}_c(K, K', q) = \frac{N_c^2}{V_{\text{BZ}}^2}\int d\mathbf{k}d\mathbf{k}'\,\phi_{\mathbf{K}}(\mathbf{k})\,\Gamma^{pp}(k, k', q)\,\phi_{\mathbf{K}'}(\mathbf{k}')\,, \tag{42}$$

which requires the coarse-grained average of the lattice function to match the corresponding cluster quantity. Consistent with the computation of the continuous lattice self-energy $\Sigma_{\text{DCA}^+}(\mathbf{k}, i\omega_n)$, we obtain $\Gamma^{pp}_{\text{DCA}^+}(k, k', q)$ from an interpolation of the cluster vertex function,

$$\left\{\mathbf{K}, \Gamma^{pp}_c(K, K', q)\right\} \to \left\{\mathbf{k}, \tilde{\Gamma}^{pp}(k, k', q)\right\}\,, \tag{43}$$

followed by a deconvolution,

$$\begin{aligned} \tilde{\Gamma}^{pp}(k_1, k_2, q) &= \frac{N_c^2}{V_{\text{BZ}}^2}\int d\mathbf{k}'_1 d\mathbf{k}'_2\,\phi(\mathbf{k}_1 - \mathbf{k}'_1) \\ &\times \Gamma^{pp}_{\text{DCA}^+}(k'_1, k'_2, q)\,\phi(\mathbf{k}_2 - \mathbf{k}'_2)\,. \end{aligned} \tag{44}$$

From $\Gamma^{pp}_{\text{DCA}^+}(k, k', q)$ we can compute the Bethe–Salpeter eigenvectors and eigenvalues with full momentum resolution (see Eq. (38)).

In some cases it has proven useful to project the Bethe–Salpeter kernel onto a set of crystal harmonics corresponding to localized lattice vectors. This *folding* not only leads to much smaller matrices to diagonalize, but more importantly reduces noise by projecting onto the physically relevant subspace.

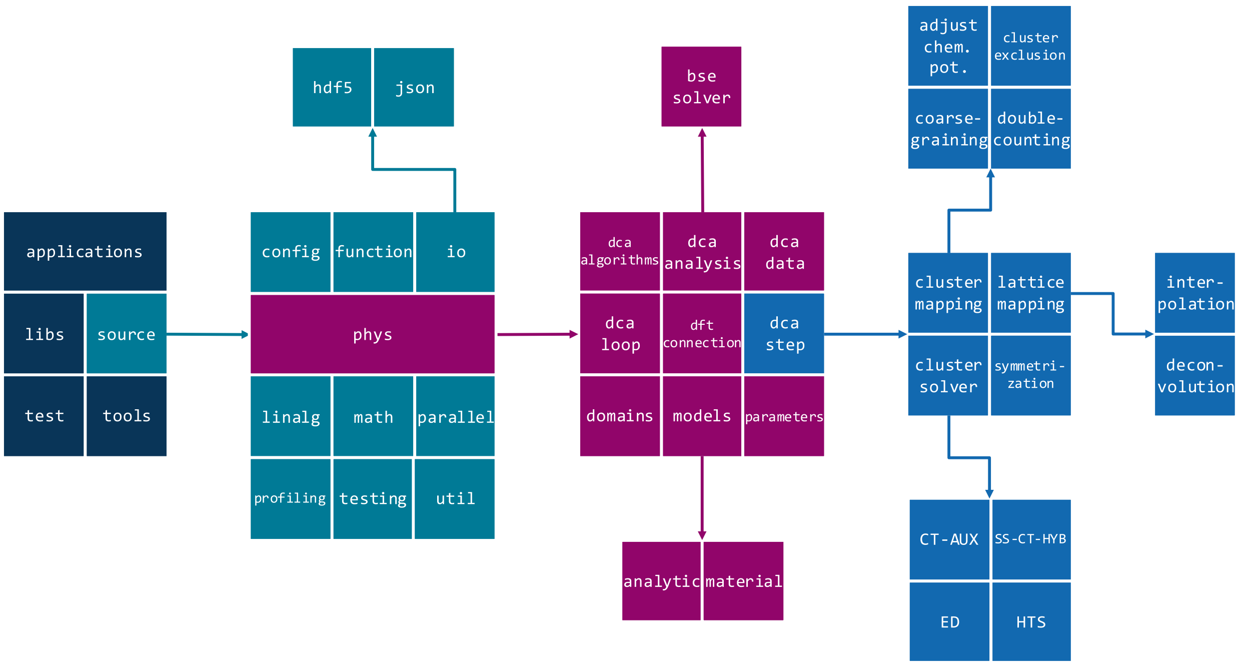


Fig. 8. (Color online) Schematic overview of the code base. The modular and hierarchical structure reflects the generic design of the DCA++ code.

3. The DCA++ code

3.1. Main components

The main purpose of this section is to give an overview of the key parts and main capabilities of the DCA++ code. A visual summary of the code base is shown in Fig. 8.

We have chosen C++ as the primary implementation language for its generic programming model, which, besides easy extensibility, allows us to hide architectural details. The C++14 standard, which DCA++ adopts, provides many modern language features for effective and safe programming, while still having a wide compiler support. On top of C++, we use CUDA to utilize GPU resources in performance critical parts of the code.

Cluster mapping. Distinguishing DCA and DCA$^+$ from finite-size methods, the coarse-graining procedure is a central element of the code. In addition to the traditional coarse-graining, which partitions the first Brillouin zone according to the Brillouin zones of the superlattice, the interlaced coarse-graining developed by Staar et al. [17] is implemented with adjustable number of periods, that is variable degree of interleaving. Besides the coarse-graining routines, the cluster mapping module contains methods to update the chemical potential and execute the cluster-exclusion step. In LDA+DMFT calculations the double-counting problem is addressed by the correspondent submodule with a constant correction term [28], specified as an input parameter.

Lattice mapping. The framework of DCA$^+$ requires a non-trivial lattice mapping to generate a lattice self-energy and irreducible vertex function with full momentum resolution. After interpolation of the cluster functions onto the fine lattice grid, the lattice mapping becomes a deconvolution problem. In the case of the self-energy, we employ the iterative Richardson–Lucy deconvolution algorithm as explained in [9]. For the deconvolution of the irreducible vertex function we have chosen a simpler approach. After Fourier transformation of the convolution equation (44) to real space, the deconvolution problem presents itself as a simple division.

Cluster solver. Solving the effective cluster problem and computing a new estimate of the cluster self-energy is the most time consuming part in a DCA or DCA$^+$ calculation. For this reason, lots of effort has been put in the development of a cutting edge implementation of the CT-AUX algorithm including submatrix updates and efficient NFFT measurements. With the aim of featuring optimal methods for the entire spectrum of quantum impurity models, we provide a complementary cluster solver to CT-AUX in the form of the segment formulation of CT-HYB restricted to single-site problems (SS-CT-HYB). Besides the two continuous-time QMC methods, DCA++ implements two other cluster solvers, each bound to a special purpose. A high temperature series expansion (HTS) solver may help improve convergence of the lattice mapping by computing a fourth order perturbation expansion of the self-energy with respect to the on-site Coulomb interaction U. To validate QMC methods and new models, an ED solver can be employed to compute exact results on small lattices in finite-size calculations (no mean-field).

Models. DCA++ is able to simulate standard, extended and frustrated Hubbard models on various types of lattices. 2D square and triangular lattices, and a bilayer square lattice corresponding to a two-orbital model are currently supported, but new lattices and models can easily be added upon need. In particular, the code's generic structure based on C++ templates permits to easily treat 1D and 3D lattices, too. In the framework of LDA+DMFT real materials such as CuO_2 and NiO can be studied in ab initio calculations.

Input/output. Running simulations requires reading input files and writing out results. The input and output (I/O) module supports two data formats, HDF5 [29] and JSON [30]. While JSON's human-readability makes it very suitable for specifying input parameters, we recommend using the binary HDF5 format to write output. HDF5 features high speed in reading and writing data, and requires less storage size compared to other data formats.

Parallelization. The computationally most expensive part of a DCA or DCA$^+$ calculation is the QMC step. Fortunately, Monte Carlo

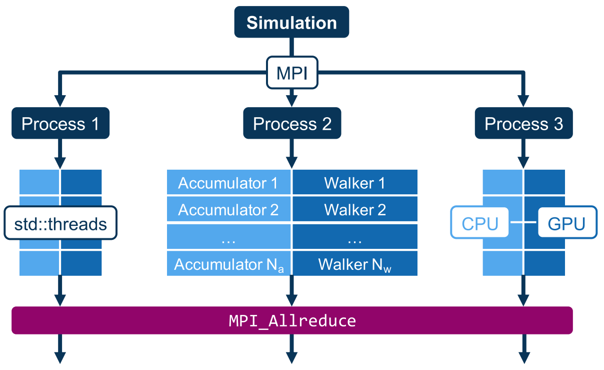


Fig. 9. (Color online) MPI + threading parallelization scheme of a DCA or DCA$^+$ calculation. If GPU resources are available, the computationally heavy matrix operations performed by a CT-AUX walker are executed on them. Communication between MPI processes is limited to a few `MPI_Allreduce` calls at the very end of the Monte Carlo integration.

simulations are embarrassingly parallel, which we exploit on distributed multi-core machines with a two level (MPI + threading) parallelization scheme, illustrated in Fig. 9.

Building DCA++ with MPI enabled allows to run the DCA or DCA$^+$ calculation with multiple MPI processes. Each MPI process performs an independent Monte Carlo integration and results are accumulated at the end in few collective reductions. In addition, a threaded Monte Carlo wrapper allows each MPI process to spawn multiple Monte Carlo walker and accumulator threads. Each Monte Carlo walker processes its own Markov chain and, as soon as it has executed the required number of updates per measurement, sends its current configuration over to a queue of idle Monte Carlo accumulators. A free accumulator then takes measurements on the sampled configuration. The total number of measurements during the Monte Carlo step of a single DCA or DCA$^+$ iteration is given by

$$\begin{aligned}\text{total number of measurements} &= \text{number of MPI processes}\\ &\times \text{number of accumulators}\\ &\times \text{number of measurements per accumulator}\,.\end{aligned}$$

GPU. In addition to conventional multi-core architectures, DCA++ ports to the emerging hybrid CPU–GPU systems. Comprising large matrix–matrix multiplications, the CT-AUX cluster solver with submatrix updates particularly benefits from the computing power of the GPU. If DCA++ is built with GPU support enabled, all Monte Carlo walkers of a process are moved to its dedicated GPU. This leads to a considerable speed-up compared to the multi-core implementation [13].

Applications. `main_dca` is the central application of DCA++. An input parameter allows to choose at runtime whether the DCA or DCA$^+$ algorithm should be executed. One can turn off the mean-field and do a finite-size QMC calculation, too. Depending on input parameters, either a fixed number of iterations are carried out, or the DCA or DCA$^+$ algorithm is iterated until the desired accuracy is reached. The main output of the application is the cluster's single-particle Green's function and self-energy. The lattice self-energy can be written out, too, when DCA$^+$ is used. If required, the application additionally measures and stores the two-particle Green's function of the cluster.

The output of `main_dca` is post processed by the application `main_analysis`. This application computes the eigenvectors and eigenvalues of the Bethe–Salpeter equation, which provide information about nature and extent of instabilities in the system and possible phase transitions. Like `main_dca`, `main_analysis` can be run in DCA or DCA$^+$ mode. While in DCA mode the momentum resolution of the BSE eigenvectors is restricted to the cluster momenta, the BSE solver produces eigenvectors with continuous momentum dependence when DCA$^+$ is used.

3.2. Building

Prerequisites. Building DCA++ requires a set of tools and libraries:

- a C++14 compiler (tested: clang $\geq$ 3.5, gcc $\geq$ 4.9),
- the CMake build system [31] version 3.3 or higher,
- the HDF5 library with the C++ interface (tested: 1.8.13 and 1.10),
- FFTW3 *or* an FFT library with the FFTW3 interface (e.g. Intel MKL),
- BLAS and LAPACK libraries, and
- MPI, if requested.

To enable the GPU support, one needs to provide

- the CUDA Toolkit (tested: 7.0, 7.5 and 8.0), and
- the MAGMA library (tested: 2.0.0 with CUDA 7.0 and 2.2.0 with CUDA 7.5 and 8.0).

Further optional requirements are

- the EasyBuild framework [32],
- Python, NumPy, SciPy, Matplotlib, and the h5py package to use the provided python scripts.

Building steps. The DCA++ project is maintained in a public GitHub repository at https://github.com/CompFUSE/DCA. Cloning the repository allows to obtain the latest version of the master branch. Alternatively, one can download a specific version from the release page. The recommended way of configuring and building DCA++ across all platforms is to use the CMake build system. The building procedure starts with creating a clean build directory. The `cmake` command followed by the path to the source and appropriate options generates the build files. Finally, by evoking `make` one can compile the applications and tests, if enabled. The building steps are summarized in Listing 1. More details on the building procedure can be found on the correspondent Wiki page in the GitHub repository, while a complete list of all relevant CMake options with descriptions and their default value is provided in Appendix A.

3.3. Running

Input files. There are two types of input files. All applications read *simulation parameters*, e.g. the DCA cluster, the temperature

```
$ git clone https://github.com/CompFUSE/DCA.git dca_source
$ mkdir build && cd build
$ cmake ../dca_source -D<variable1>=<value1> -D<variable2>=<value2> ...
$ make
$ make test
```

Listing 1. Summary of the DCA++ building steps. The `cmake` command requires the path to the source (here `../dca_source`) and can be complemented by a list of options using the `-D` flag. If tests have been built, they can be executed with the command `make test`.

or output filenames, from a JSON-formatted input file, in which parameters are thematically grouped and sub-grouped in JSON objects. Appendix B provides descriptions of all input parameters including their type and default value.

In addition to providing simulation parameters, *data* needs to be transferred between individual runs. A `main_dca` run uses the output of a previous calculation to initialize the self-energy with a non-zero value (see below) and the application `main_analysis` requires a `main_dca` output file containing the cluster single-particle and two-particle Green's functions, as well as the self-energy. While the file format for data can either be HDF5 (recommended) or JSON, it has to match between consecutive runs.

Parallelization. On modern systems with multi-core nodes we usually run up to two MPI tasks per node and one Monte Carlo walker or accumulator thread per core. The optimal distribution of Monte Carlo walker and accumulator threads, on the other hand, is strongly problem dependent. Note that memory restrictions on the GPU might limit the number of walker threads a process can spawn. The environment variable `OMP_NUM_THREADS` can be used to enable threading in BLAS, LAPACK and FFTW routines. One should be aware, however, that some of the calls to these routines are already within parallel regions.

Cooldown. A full DCA or DCA$^+$ calculation is usually performed by slowly *cooling down* the system. In a series of `main_dca` runs the temperature is gradually lowered to study its dependence on fluctuations leading to phase transitions. Sometimes this procedure is also necessary to guarantee convergence of the DCA and DCA$^+$ loop at low temperatures.

The calculation is started at a high temperature ($T \approx 1.0t$), at which a vanishing self-energy is a good initial guess, and the system then slowly cooled down. Each following `main_dca` run is initialized with the result of the previous temperature. While for the first temperature up to eight iterations are required until convergence is reached, three to six iterations are usually enough for the subsequent runs, given the temperature steps are sufficiently small. Lowering the temperature is usually done with decreasing step size.

In the beginning only single-particle (sp) functions, i.e. the cluster Green's function and self-energy, are computed. When we reach the relevant temperature regime in which the system exhibits an instability, we additionally measure the required cluster two-particle Green's function in a separate two-particle (tp) run, which only performs a single DCA or DCA$^+$ iteration. Subsequent to the `main_dca` runs, we separately analyze the output of all tp-runs with `main_analysis`. Listing 2 provides a schedule of jobs for a typical cooldown procedure.

We provide a python script that generates directories, input files and batch scripts for a cooldown. By default, this script and the provided template input files are configured for DCA or DCA$^+$ calculation of the 2D single-band Hubbard model with on-site Coulomb interaction U and fixed electron filling d, but it can easily be adjusted for other problems. Usage of the script is explained in the Wiki.

4. Sustainable and scalable software development

The DCA++ project follows several well-proven methods in software engineering to develop and maintain a high-performance research code in a multidisciplinary team. This section is devoted to sharing these methods and the tools that allow to implement them. In addition to the dedicated article [16], we want to call the interested reader's attention to the *"How to" documents* of the IDEAS Scientific Software Productivity Project [33], which collect best practices in scientific software development, and which we partially follow here.

4.1. Software testing and test-driven development

Systematic testing is an essential element in software development to produce correct code. In addition to detecting errors, software tests provide an important form of documentation by describing correct usage and defining requirements and limits with edge cases.

Software tests can be divided according to their level into *unit*, *integration*, and *system-level tests*. While unit tests operate at the smallest code scale testing single functions and classes, integration tests check the interaction between these units, and system-level tests validate and verify the behavior of the full code. Orthogonal to this classification, we can categorize tests by their aspect into *verification*, *validation*, *no-change*, and *performance tests* [33]. Verification tests determine whether the code satisfies specific requirements or conditions. They answer the question "Are we building the product right?" [34]. Verification testing at the unit level is the tool kit of test-driven development (see below). Validation tests, in contrast, are motivated by the question "Are we building the right product?" [34]. These tests evaluate whether the software satisfies the intended purpose. No-change tests define the current behavior of the software and are indispensable for refactoring legacy code. Passing this type of software tests requires reproducing predefined results close to machine precision. Performance tests are used to compare alternative algorithms, to monitor the code's performance, or to benchmark computing systems.

All of our software tests are automated and part of the regression test suite. The regression test suite is divided with respect to required computational resources into *fast tests* and *extensive tests*, while performance tests are kept separated. Fast tests include unit tests and lightweight integration tests. Since they build and run fast, they can be executed frequently in the code development

```
mpirun -np 8 ./main_dca ./T=1/input_sp.json     # Initial self-energy: "zero".
mpirun -np 8 ./main_dca ./T=0.5/input_sp.json   # Initial self-energy: T=1/dca_sp.hdf5.
mpirun -np 8 ./main_dca ./T=0.25/input_sp.json  # Initial self-energy: T=0.5/dca_sp.hdf5.

mpirun -np 8 ./main_dca ./T=0.1/input_sp.json   # Initial self-energy: T=0.25/dca_sp.hdf5.
mpirun -np 8 ./main_dca ./T=0.1/input_tp.json   # Initial self-energy: T=0.1/dca_sp.hdf5.
mpirun -np 8 ./main_dca ./T=0.09/input_sp.json  # Initial self-energy: T=0.1/dca_sp.hdf5.
mpirun -np 8 ./main_dca ./T=0.09/input_tp.json  # Initial self-energy: T=0.09/dca_sp.hdf5.
mpirun -np 8 ./main_dca ./T=0.08/input_sp.json  # Initial self-energy: T=0.09/dca_sp.hdf5.
mpirun -np 8 ./main_dca ./T=0.08/input_tp.json  # Initial self-energy: T=0.08/dca_sp.hdf5.

mpirun -np 8 ./main_analysis ./T=0.1/input_tp.json    # Reads T=0.1/dca_tp.hdf5.
mpirun -np 8 ./main_analysis ./T=0.09/input_tp.json   # Reads T=0.09/dca_tp.hdf5.
mpirun -np 8 ./main_analysis ./T=0.08/input_tp.json   # Reads T=0.08/dca_tp.hdf5.
```

Listing 2. Workflow of a DCA or DCA$^+$ cooldown.

to detect problems early. Extensive tests contain computationally more expensive integration tests and system-level tests, and are usually executed before a piece of code gets back into the main trunk (see workflow below). They provide additional evidence that the code's behavior has not been unintentionally altered.

Test-driven development (TDD) [35] is a software development technique in which automated tests steer the design process. Based on the two rules of only writing new code if there is a failing test and removing duplication, TDD consistently follows a three-step cycle:

1. Create an automated test that fails.
2. Make the test pass quickly by any means necessary.
3. Refactor to eliminate duplication introduced by getting the test to pass.

As any functional code is protected by at least one unit test, TDD prevents regression and delivers high confidence in the correctness of the software. Writing tests first has the additional benefit that it encourages the programmer to think about the interface of the code first. This way, TDD fosters designs that are high in cohesion and low in coupling. The rhythm of TDD allows to work in small, safe steps and to focus on one task at a time. The tests provide immediate feedback.

In the DCA++ project, we follow TDD whenever the existing code allows it and refactor code to make it more testable. Where unit tests cannot be put in place, system-level no-change tests act as a global safety net. Moreover, we require defects to be reported in the form of minimal, failing tests. Our goal is a growing test suite and an increasing percentage of test driven code.

We implement tests using the Google C++ Testing Framework (Google Test) [36]. Google Test builds on the xUnit architecture and provides benefits both at writing and executing tests. It supports various fatal and non-fatal assertions, including death tests, and allows tests to share resources that are expensive to set up and not modified by the tests. Moreover, tests can be value- or type-parameterized to cover a wide range of cases without duplicating test code.

4.2. Software project management and collaborative development

Tools. Successfully maintaining any software project begins with keeping the code base under version control. By storing old versions of the software and monitoring changes to the code base, a version control system crucially contributes to reproducibility of simulation results and safe code development. Version control systems that support branching allow the isolated implementation of features in dedicated branches, while the master branch, which contains the production code, is always operational. We use the popular Git version control system [37], in which each developer clones their own local repository. Besides being distributed, Git offers fast and easy branching capabilities, which makes it well suited for collaborative software development. Our Git repository is hosted on the GitHub platform, which, in addition to the repository, provides a set of tools for managing a software project enabling us to implement the workflow explained below.

While GitHub helps us to maintain the DCA++ source, we employ the EasyBuild framework to manage builds of DCA++ on production systems. All dependencies of the DCA++ code, such as external libraries and compilers, but also the CMake build system, are specified in a build recipe. Bundled in the so-called "easyconfig" file, the dependencies can be installed with a one-line command into a single module[1]. This module can be loaded and used by other

[1] More commonly, EasyBuild is used to not only build the dependencies but the full application or library.

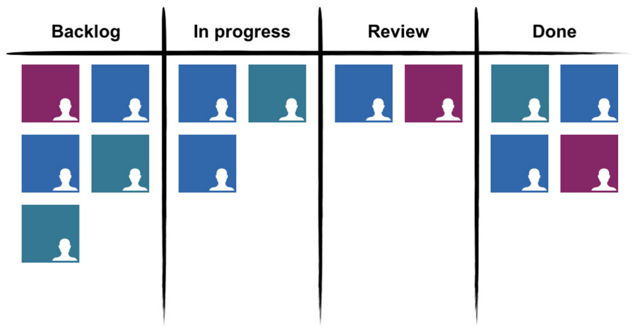


Fig. 10. (Color online) Kanban board for software development. Each note corresponds to an issue. During its lifetime, an issue moves across the board from left to right.

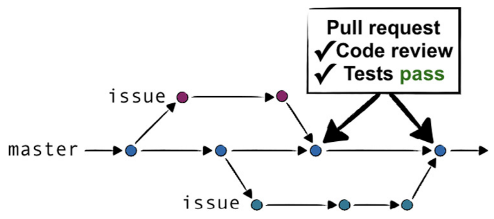


Fig. 11. (Color online) Issue branch workflow. Each issue is implemented in a separate branch, while the master branch is constantly operational. Code reviews and automated testing at the end of an issue cycle ensure quality and correctness of the new code before it arrives in the master branch.

collaborators to easily build the DCA++ code. We currently provide easyconfig files for CSCS' Piz Daint (hybrid Cray XC50/XC40) and ORNL's Titan (Cray XK7) supercomputers.

To automate the quality control of the code, we run a Jenkins server [38] hooked up to DCA++'s GitHub repository. Triggered by changes to the repository, Jenkins builds all applications and tests, and runs the test suite. A test report allows to immediately detect any degradation and to document the status of the code for later reference.

Workflow. It is our goal to continuously and frequently deliver updates to the DCA++ code without compromising its quality. To achieve this, we organize the code development by tracking each task, bug fix or other enhancement as an issue. GitHub's issue tracker offers to categorize issues with labels and to assign them to a team member. By associating issues with a milestone, the process to a specific goal can be monitored. The implementation of a single issue should not exceed few weeks and more complex features, which we call *epics*, should be broken down into smaller, independent tasks. To visualize the status of all issues and to prioritize work, we use a Kanban board, a popular tool in agile development. The board is organized in columns with each column representing a stage in the lifetime of an issue, i.e. *Backlog*, *In progress*, *Review* and *Done*. A typical Kanban board for a software project is shown in Fig. 10.

By developing each issue in a dedicated branch, we guarantee that the master branch is kept stable at all times. As part of the quality control, the finished work has to be submitted in form of a pull request. An experienced developer then reviews the code and requests changes if necessary. If the reviewer approves the new code and Jenkins confirms that all tests pass, it will be merged into the master branch and become production code. Fig. 11 illustrates this *issue branch workflow*.

For a successful realization of the workflow it is important to have early discussions about design decisions and implementation details. This assures that the development of a new issue starts

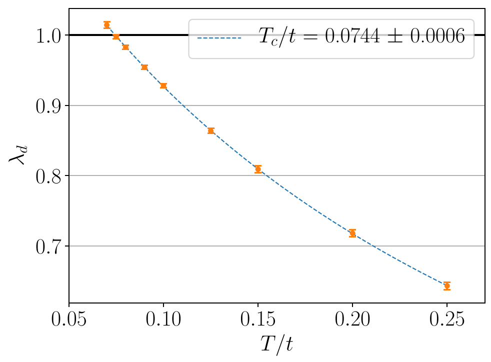


Fig. 12. (Color online) Leading (d-wave) eigenvalue of the particle–particle Bethe–Salpeter kernel in the DCA calculated for $U = 8t$, $\langle n\rangle = 0.95$ and $N_c = 4$. The intersection of $\lambda_d(T)$ with one yields the transition temperature T_c. Standard deviations illustrated as error bars were computed from 10 independent simulations that each performed 1 000 000 Monte Carlo measurements per iteration.

immediately in the right direction. Collaborative development in large teams once more requires keeping individual issues small. As part of continuous integration, we demand developers to rapidly check in their changes via pull requests to master. This allows them to get feedback frequently and prevents a scenario, called *merge hell*, in which it involves considerable effort to integrate a far diverged branch back into the main line. Short issue cycles also mean that new features and bug fixes are quickly available to team members and users. This can, in particular, avoid duplication of code work.

5. Examples

The following two standard use cases study superconductivity in the 2D Hubbard model. The lattice model is described by the Hamiltonian

$$H = -t \sum_{\langle i,j\rangle,\sigma} c^\dagger_{i\sigma} c_{j\sigma} + U \sum_i n_{i\uparrow} n_{i\downarrow} \,, \tag{45}$$

where $c^\dagger_{i\sigma}$ ($c_{i\sigma}$) creates (annihilates) an electron with spin σ on lattice site i, $n_{i\sigma} = c^\dagger_{i\sigma} c_{i\sigma}$ is the corresponding number operator and U denotes the on-site Coulomb repulsion. In the form above, where $\langle i, j\rangle$ indicates that the first sum only runs over pairs of nearest neighbor sites, the model is restricted to nearest neighbor hopping with isotropic amplitude t. The simulations are performed on a square lattice.

In the first example, we determine the superconducting transition temperature T_c for a small 2×2 cluster employing the DCA method. The second example provides a large-scale DCA$^+$ calculation in which we converge T_c with respect to the cluster size N_c. In both cases we use the CT-AUX QMC algorithm to solve the effective cluster problem.

5.1. Computing the superconducting transition temperature T_c with DCA

As explained in Section 2.5, we can analyze the transition to the superconducting phase by computing the leading eigenvalue of the Bethe–Salpeter equation in the particle–particle channel. The superconducting state of the 2D Hubbard model is characterized by d-wave symmetry [26,39]. In Fig. 12 we plot the leading d-wave eigenvalue λ_d as a function of temperature for strong coupling $U = 8t$, electron filling $\langle n\rangle = 0.95$ and $N_c = 4$. As the temperature is lowered, the eigenvalue increases and the system approaches the phase instability. After fitting the data points with a function of the form

$$\lambda_d(T) = \frac{a}{(T-b)^c} \,, \tag{46}$$

we can solve for the transition temperature T_c, defined as the temperature for which $\lambda_d(T)$ crosses one, i.e. $\lambda_d(T_c) = 1$.

We performed the computation on Piz Daint's multi-core partition, a Cray XC40 system equipped with two 18-core Intel Xeon processors (2.1 GHz) per node. Using a single compute node, results for 1 000 000 Monte Carlo measurements per iteration were obtained in 3.5 h. The time-to-solution (TTS) can be greatly reduced by exploiting the code's parallel performance and increasing the number of nodes. To quantify this, we measured the *strong scaling* efficiency $E_s(n)$ as a function of the number of nodes n. It is defined as

$$E_s(n) = \frac{TTS(1)}{n \times TTS(n)} \times 100\% \,. \tag{47}$$

Results are shown in the left plot of Fig. 13. Even though the computational demand of the problem is small, DCA++ allows to efficiently use up to 200 compute nodes, for which the TTS reduces to less than two minutes.

We also examined the *weak scaling* behavior, where the problem size (number of Monte Carlo measurements) per node is kept constant. Weak scaling efficiency $E_w(n)$ as a function of the number of nodes n is computed as

$$E_w(n) = \frac{TTS(1)}{TTS(n)} \times 100\% \,. \tag{48}$$

The right plot of Fig. 13 presents the results for 1 000 000 Monte Carlo measurements per iteration and node. While overall, we observe weak scaling efficiency close to the ideal case of 100%, the 10 node result actually shows a deviation to "better than ideal". This can be explained by the fact that we only kept the problem size with respect to the number of Monte Carlo measurements constant per node. Other parts of the code, in particular the coarse-graining step, benefited from a speed-up when the number of nodes was increased.

5.2. Cluster size scaling of T_c with DCA$^+$

The reduced fermionic sign problem in the DCA$^+$ algorithm makes cluster sizes accessible that have been out of reach for standard DCA or finite-size QMC calculations. This allows to enter a regime of asymptotic convergence, in which the transition temperature T_c becomes independent of the cluster size. Here, we consider the weak-coupling case of $U = 4t$ at 10% doping. Fig. 14 shows the temperature dependence of the leading d-wave eigenvalue $\lambda_d(T)$ for cluster sizes between 24 and 56 and Fig. 15 plots the resulting values for T_c versus cluster size N_c. We observe convergence of the transition temperature T_c for clusters larger than 32 sites.

Error bars in Figs. 14 and 15 represent, with the exception of $N_c = 36$, the statistical error of the QMC algorithm. They tend to grow with the size of the cluster. This originates from the behavior of the fermionic sign problem, which becomes more severe with increasing cluster size leading to a larger statistical error.

For the case of $N_c = 36$, we used four different cluster shapes to illustrate the error associated with the choice of cluster for given cluster size. While the cluster shape dependence is substantially reduced by the introduction of DCA$^+$ [9], the corresponding spread in results still exceeds the statistical uncertainty of the Monte Carlo algorithm. Ref. [26] extends this analysis to more cluster sizes.

Large expansion orders resulting in large matrix operations make this problem computationally demanding, but also well suited for Piz Daint's Cray XC50 hybrid compute nodes. They provide, in addition to a 12-core Intel Xeon processor (2.6 GHz),

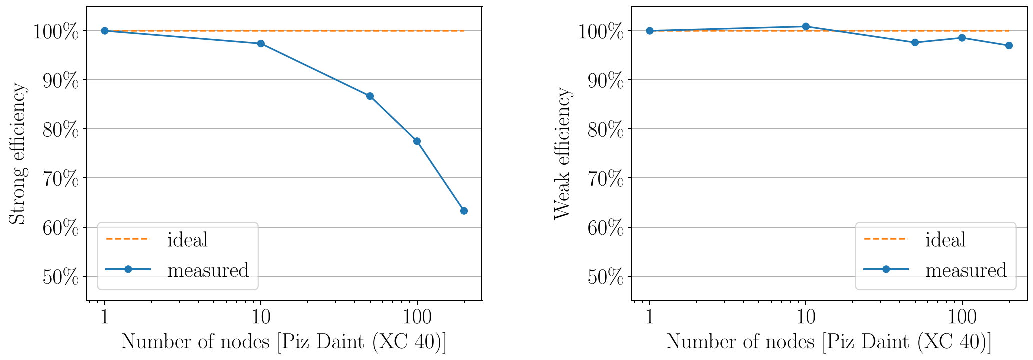


Fig. 13. (Color online) Strong (left) and weak (right) scaling efficiency for 1 to 200 Cray XC40 multi-core nodes. Results represent time-to-solution of the full cooldown. We used 1 000 000 Monte Carlo measurements per iteration for the strong scaling analysis, while weak scaling is based on 1 000 000 Monte Carlo measurements per iteration and node.

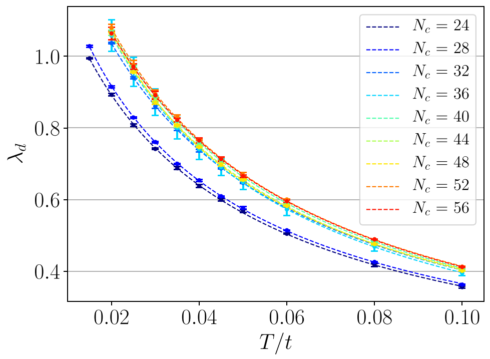


Fig. 14. (Color online) Leading (d-wave) eigenvalue of the particle–particle Bethe–Salpeter kernel in the DCA$^+$ framework for $U = 4t$, $\langle n \rangle = 0.9$ and various cluster sizes. Error bars present, with the exception of $N_c = 36$, the standard deviation of four independent simulations with 1 000 000 Monte Carlo measurements each. For $N_c = 36$, the error bars depict the standard deviation of results for four different cluster shapes.

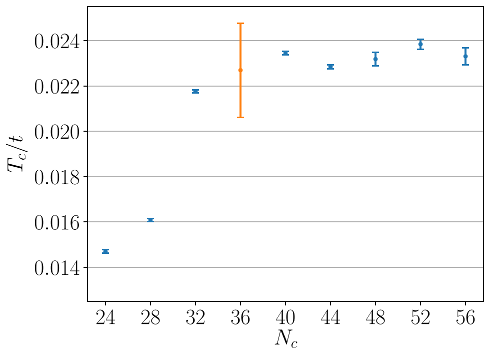


Fig. 15. (Color online) DCA$^+$ results for the cluster size dependence of the transition temperature T_c for $U = 4t$ and $\langle n \rangle = 0.9$. Error bars present, with the exception of $N_c = 36$, the standard deviation of four independent simulations with 1 000 000 Monte Carlo measurements each. For $N_c = 36$, the error bar depicts the standard deviation of results for four different cluster shapes.

an NVIDIA Tesla P100 GPU accelerator. This leadership computing system was used to examine the large-scale parallel performance of the DCA++ code. We measured the TTS of the QMC integration for the computing intensive case of $N_c = 56$ and $T = 0.02t$. Due to the problem size, we chose a minimum of 500 compute nodes to limit the run time. For illustration, the full cooldown for $N_c = 56$ takes about 5.2 h on 500 nodes. We performed 1 000 000 Monte Carlo measurements for each node count to investigate the strong scaling behavior and used 2000 Monte Carlo measurements per node in the weak scaling analysis. Fig. 16 presents strong and weak scaling results for up to 5000 compute nodes. The observed strong scaling efficiency implies that we achieve a considerable speed-up over 500 nodes when using nearly the entire machine. As in the multi-core case, weak scaling efficiency excels reaching almost 100%.

We want to conclude this example by looking at the speed-up that a Cray XC50 hybrid node provides over a Cray XC40 multi-core node. As in the scaling analysis, we compare the TTS of the QMC integration for the 56-site cluster at $T = 0.02t$. The total number of Monte Carlo measurements per node was fixed to 2000. We can see in Fig. 17 that the speed-up in a single-particle run is close to threefold, while the two-particle run is accelerated by a factor of 2.5. The smaller speed-up when the computationally expensive cluster two-particle Green's function is accumulated accounts to the fact that the hybrid implementation of the CT-AUX solver performs the measurements on the host CPU.

6. Final remarks

Future development. DCA++ is an active software project with constantly extended functionality. The next release will add full multi-orbital support and a state of the art implementation of the continuous-time interaction expansion QMC algorithm (CT-INT) [40] to allow for simulations of multi-orbital models of systems such as the iron-based superconductors.

Since Monte Carlo algorithms are based on random sampling, they are non-deterministic by nature. While implementations of these algorithms can still be tested at the unit level, it is often difficult to design conventional integration or system-level tests due to the random behavior. For this reason, we have developed a framework based on hypothesis testing, which is able to validate stochastic programs. This framework is currently being integrated into DCA++.

The workload balance in the CT-AUX solver on hybrid CPU–GPU systems, i.e. running the Monte Carlo walker on the GPU and accumulating measurements on the CPU, has proven to be efficient on current systems. However, to respond to the emerging GPU-dominated node configurations, the next release of DCA++ will provide the capability to move the CT-AUX accumulator to the GPU, too. In the long term, the growing diversity of HPC architectures will make it necessary to revisit the on-node parallelization strategy altogether. We plan to follow a task based parallelization model based on the HPX-3 system [41] to ensure portability and efficiency for the future.

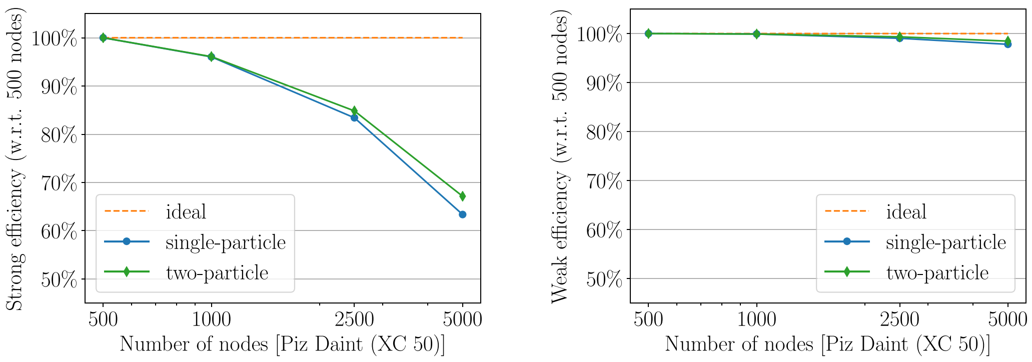


Fig. 16. (Color online) Strong (left) and weak (right) scaling efficiency for 500 to 5000 Cray XC50 hybrid CPU–GPU nodes. Results represent time-to-solution of the QMC integration for $N_c = 56$ and $T = 0.02t$. Efficiencies are computed with respect to the 500 nodes result. We used 1 000 000 Monte Carlo measurements for the strong scaling analysis, while weak scaling is based on 2000 Monte Carlo measurements per node. Data distinguish between single-particle runs (blue dots), in which only single-particle quantities were accumulated, and two-particle runs (green diamonds), in which also the cluster two-particle Green's function was measured.

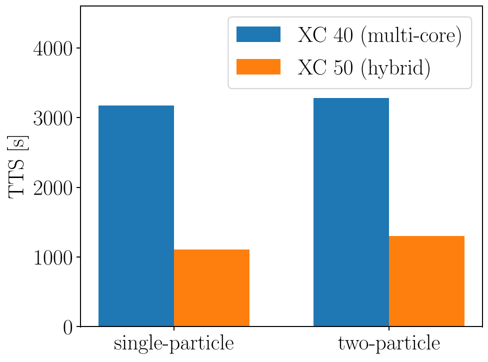


Fig. 17. (Color online). Comparison of the time-to-solution of the QMC integration for $N_c = 56$, $T = 0.02t$ and 2000 Monte Carlo measurements between a Cray XC40 multi-core node and a Cray XC50 hybrid CPU–GPU node. Results are shown for a single-particle run (left), in which only single-particle quantities were accumulated, and a two-particle run (right), in which also the cluster two-particle Green's function was measured.

Getting started. The DCA++ code is licensed under the BSD 3-Clause License. Hence, it is open source and a version of the code can easily be obtained from the public GitHub repository at https://github.com/CompFUSE/DCA. We advise prospective users to visit the Wiki section of the repository, which, in particular, provides a small tutorial. Questions about building or running DCA++ can be posted to the repository's built-in issue tracking system.

Contributing back. There are various ways to contribute to the DCA++ project. Reports about bugs and other problems can be submitted via the issue tracking system. The issue tracker is also the preferred communication channel for any other request concerning the code, including the suggestion of new features. We welcome code contributions, too. They are best submitted by creating a pull request in the GitHub repository.

Citation guideline. If the DCA++ code contributes to your research, we kindly ask to acknowledge this in any resulting scientific publication by citing this paper.

Acknowledgments

The authors would like to acknowledge the contributions of Giovanni Balduzzi, Peter W. Doak, Mi Jiang, Andrei Plamada and Bart Ydens to the DCA++ project. We would like to thank John Biddiscombe for his contributions to refactoring the code base, Guilherme Peretti-Pezzi for his assistance in setting up continuous integration and his support with EasyBuild, and Philipp Werner for providing the CT-HYB QMC solver. U.R.H. acknowledges the support by the National Center of Competence in Research (NCCR) MARVEL, funded by the Swiss National Science Foundation. The work of T.A.M. was supported by the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research and Basic Energy Sciences, Division of Materials Sciences and Engineering. This research used resources of the Oak Ridge Leadership Computing Facility (OLCF) awarded by the INCITE program, and of the Swiss National Supercomputing Center (CSCS). OLCF is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

Appendix A. CMake options

The DCA++ build can be configured with several CMake options. These options can either be appended to the `cmake` command with the `-D` flag:

```
$ cmake
   ../dca_source -D<variable1>
       =<value1> -D<variable2>=<value2> ...
```

or specified using the `ccmake` interface, which is launched in the existing build directory (after `cmake` was called) with:

```
$ ccmake .
```

Table A.1 lists some general and all DCA++ specific CMake options.

Appendix B. Input parameters

This appendix section provides short descriptions of each input parameter using the format:

- "`parameter-name`": type (`default value`)
  Description.

In addition, Listing 3 shows a sample input file as used for the DCA$^+$ calculation of Section 5.2.

Output parameters

Group "`output`":

- "`directory`": string ("`./`")
  Directory to write the output to.
- "`output-format`": string ("`HDF5`")
  File format of the output files. Options are: "`HDF5`", "`JSON`".
- "`filename-dca`": string ("`dca.hdf5`")
  Filename for the output of the application `main_dca`.

Table A.1
Relevant CMake options for building DCA++.

Option	Default value	Description
CMAKE_BUILD_TYPE		The build type. Options are: [empty], Debug, Release, RelWithDebInfo, MinSizeRel. According to the build type CMake sets different optimization and debug compiler flags. Release is recommended for production runs.
FFTW_INCLUDE_DIR		Path to fftw3.h.
FFTW_LIBRARY		The FFTW3(-compatible) library. If the -D flag is used, multiple libraries have to be separated by semicolons and enclosed by double quotes.
DCA_HAVE_LAPACK		Set to TRUE if you use a compiler wrapper that automatically links against BLAS and LAPACK libraries, to prevent CMake from searching for these libraries.
DCA_WITH_MPI	ON	Enable MPI. If MPI is enabled, set the environment variable CXX to the MPI compiler wrapper before running CMake.
DCA_WITH_CUDA	OFF	Enable GPU support.
DCA_WITH_PINNED_HOST_MEMORY	OFF	Enable pinned host memory. *(advanced)*
CUDA_GPU_ARCH	sm_60	Name of the *real* architecture to build for.
CUDA_TOOLKIT_ROOT_DIR		Path to the CUDA Toolkit. Determined by CMake. Set it manually, if CMake cannot find it.
MAGMA_DIR		Path to the MAGMA installation directory. Hint for CMake to find MAGMA.
DCA_BUILD_DCA	ON	Build main_dca.
DCA_BUILD_ANALYSIS	ON	Build main_analysis.
DCA_BUILD_CLUSTER_SOLVER_CHECK	OFF	Build cluster_solver_check, an application that tests a QMC solver against exact diagonalization.
DCA_WITH_TESTS_FAST	OFF	Build DCA++'s fast tests.
DCA_WITH_TESTS_EXTENSIVE	OFF	Build DCA++'s extensive tests.
DCA_WITH_TESTS_PERFORMANCE	OFF	Build DCA++'s performance tests. (Only in Release mode.)
TEST_RUNNER		Command for executing (MPI) programs. Required if MPI is enabled or on systems that use a command for launching executables (e.g. aprun or srun).
MPIEXEC_NUMPROC_FLAG	-n	Flag used by TEST_RUNNER to specify the number of processes.
MPIEXEC_PREFLAGS		Flags to pass to TEST_RUNNER directly before the executable to run.
DCA_CLUSTER_SOLVER	CT-AUX	The cluster solver for the DCA/DCA$^+$ calculation. Options are: CT-AUX, SS-CT-HYB.
DCA_WITH_THREADED_SOLVER	ON	Use multiple walker and accumulator threads in the cluster solver.
DCA_LATTICE	square	Lattice type, options are: bilayer, square, triangular.
DCA_POINT_GROUP	D4	Point group symmetry, options are: C6, D4.
DCA_MODEL	tight-binding	Model type, options are: tight-binding.
DCA_RNG	std::mt19937_64	Random number generator, options are: std::mt19937_64, std::ranlux48, custom. See Random number generator section in the Wiki for details.
DCA_RNG_CLASS		Class name including namespaces of the *custom* random number generator.
DCA_RNG_HEADER		Header file of the *custom* random number generator.
DCA_RNG_LIBRARY		*Custom* random number generator library.
DCA_PROFILER	None	Profiler type, options are: None, Counting, PAPI.
DCA_WITH_AUTOTUNING	OFF	Enable auto-tuning. Requires a profiler type other than None. *(advanced)*
DCA_WITH_GNUPLOT	OFF	Enable Gnuplot.
DCA_WITH_SINGLE_PRECISION_MEASUREMENTS	OFF	Measure in single precision.
DCA_WITH_SINGLE_PRECISION_COARSEGRAINING	OFF	Coarsegrain in single precision. *(advanced)*
DCA_WITH_QMC_BIT	OFF	Enable QMC solver built-in tests. *(advanced)*

- "filename-analysis": string ("analysis.hdf5")
  Filename for the output of the application main_analysis.
- "filename-ed": string ("ed.hdf5")
  Filename for the ED output in the application cluster_solver_check.
- "filename-qmc": string ("qmc.hdf5")
  Filename for the QMC output in the application cluster_solver_check.
- "filename-profiling": string ("profiling.json")
  Filename for the profiling output. The file format is always JSON.
- "dump-lattice-self-energy": boolean (false)
  Write out the lattice self-energy $\Sigma(\mathbf{k}, i\omega_n)$ in the application main_dca.
- "dump-cluster-Greens-functions": boolean (false)
  Write out various cluster Green's functions in the application main_dca.
- "dump-Gamma-lattice": boolean (false)
  Write out the lattice irreducible vertex function $\Gamma(k, k', q)$ in the application main_analysis.
- "dump-chi-0-lattice": boolean (false)
  Write out $\chi_0(k)$ in the application main_analysis.

Physics parameters

Group "physics":

- "beta": double (1.)

Inverse temperature.
- `"density"`: double `(1.)`
  Target electron density.
- `"chemical-potential"`: double `(0.)`
  Initial value of the chemical potential. This value is overwritten, if an output file with an initial self-energy is read (see section DCA parameters).
- `"adjust-chemical-potential"`: boolean `(true)`
  Adjust the chemical potential to obtain the specified density.

Model parameters

◇ *Single-band Hubbard model*
Used if `square` lattice or `triangular` lattice are selected.
Group `"single-band-Hubbard-model"`:

- `"t"`: double `(0.)`
  Nearest neighbor hopping parameter.
- `"t-prime"`: double `(0.)`
  Next nearest neighbor hopping parameter.
- `"U"`: double `(0.)`
  On-site Coulomb repulsion.
- `"V"`: double `(0.)`
  Nearest neighbor coulomb repulsion for opposite spins.
- `"V-prime"`: double `(0.)`
  Nearest neighbor coulomb repulsion for same spins.

◇ *Bilayer Hubbard model*
Used if `bilayer` lattice is selected.
Group `"bilayer-Hubbard-model"`:

- `"t"`: double `(0.)`
  Nearest neighbor hopping parameter.
- `"t-prime"`: double `(0.)`
  Next nearest neighbor hopping parameter.
- `"t-perp"`: double `(0.)`
  Hopping parameter between layers.
- `"U"`: double `(0.)`
  On-site Coulomb repulsion.
- `"V"`: double `(0.)`
  Coulomb repulsion between layers for opposite spins.
- `"V-prime"`: double `(0.)`
  Coulomb repulsion between layers for same spins.

◇ *Material model*
Group `"material-model"`:

- `"t_ij-filename"`: string `("t_ij.txt")`
  Name of the CSV file containing the hopping matrix t_{ij}.
- `"U_ij-filename"`: string `("U_ij.txt")`
  Name of the CSV file containing the Coulomb repulsion matrix U_{ij}.

DCA parameters

Group `"DCA"`:

- `"initial-self-energy"`: string `("zero")`
  Either the name of the file (including path) with the initial self-energy (usually the `main_dca` output file of the previous temperature) or `"zero"` indicating that the initial self-energy should be zero.
- `"iterations"`: integer `(1)`
  Number of DCA/DCA$^+$ iterations.
- `"accuracy"`: double `(0.)`
  Stop the DCA/DCA$^+$ loop if this accuracy has been reached.
- `"self-energy-mixing-factor"`: double `(1.)`
  Parameter α of Eq. (12).
- `"interacting-orbitals"`: array of integers `([0])`
  Indices of orbitals that are treated interacting orbitals. This parameter must be consistent with the model that is used.
- `"do-finite-size-QMC"`: boolean `(false)`
  Do a finite-size QMC calculation (no mean-field).
- Subgroup `"coarse-graining"`:

  - `"k-mesh-recursion"`: integer `(0)`
    Number of recursion steps in the creation of the momentum space mesh. See Ref. [17] for details.
  - `"periods"`: integer `(0)`
    Number of periods of the interlaced coarse-graining patches, i.e. degree of their interleaving. Restricted by `k-mesh-recursion`. See Ref. [17] for details.
  - `"quadrature-rule"`: integer `(1)`
    Determines the quadrature rule used in the coarse-graining.
    `quadrature-rule` < 0: use a flat mesh.
    `quadrature-rule` $>= 0$: use the Grundmann–Moeller rule [42] of index = `quadrature-rule`.
    A rule of index s is exact for polynomials up to order $2s + 1$.
  - `"threads"`: integer `(1)`
    Number of threads used in the coarse-graining.
  - `"tail-frequencies"`: integer `(0)`
    Number of tail frequencies used for updating the chemical potential.

- Subgroup `"DCA+"`:

  - `"do-DCA+"`: boolean `(false)`
    Use the DCA$^+$ algorithm.
  - `"deconvolution-iterations"`: integer `(16)`
    Maximum number of iterations in the deconvolution step.
  - `"deconvolution-tolerance"`: double `(1.e-3)`
    Termination criteria for the deconvolution step.
  - `"HTS-approximation"`: boolean `(false)`
    Use a high-temperature series approximation in the lattice mapping.
  - `"HTS-threads"`: integer `(1)`
    Number of threads used in the HTS-solver.

Domains parameters

Group `"domains"`:

- Subgroup `"real-space-grids`:

  - `"cluster"`: array of arrays of integers (lattice basis, e.g. in 2D: `[[1, 0], [0, 1]]`)
    Real space DCA cluster. Given as coordinates with respect to the lattice basis.
    To choose a cluster of a given size, consult the exhaustive list of 2D and 3D clusters in `tools/cluster_definitions.txt`.
  - `"sp-host"`: array of arrays of integers (lattice basis, e.g. in 2D: `[[1, 0], [0, 1]]`)
    Real space host grid for single-particle functions, also called *(sp-)lattice*. Given as coordinates with respect to the lattice basis.
  - `"tp-host"`: array of arrays of integers (lattice basis, e.g. in 2D: `[[1, 0], [0, 1]]`)
    Real space host grid for two-particle functions, also called *tp-lattice*. Only used in DCA$^+$.

Given as coordinates with respect to the lattice basis.

- Subgroup "`imaginary-time`":
  - "`sp-time-intervals`": integer (`128`)
    Discretization of the imaginary time axis.
  - "`time-intervals-for-time-measurements`": integer (`1`)
    Discretization of the imaginary time axis for additional time measurements.
- Subgroup "`imaginary-frequency`":
  - "`sp-fermionic-frequencies`": integer (`256`)
    Number of fermionic Matsubara frequencies for single-particle functions.
  - "`HTS-bosonic-frequencies`": integer (`0`)
    Number of bosonic Matsubara frequencies in the HTS-solver.
  - "`four-point-fermionic-frequencies`": integer (`1`)
    Number of fermionic Matsubara frequencies for four-point functions.
- Subgroup "`real-frequency`":
  - "`min`": double (`-10.`)
    Minimum real frequency.
  - "`max`": double (`10.`)
    Maximum real frequency.
  - "`frequencies`": integer (`3`)
    Number of real frequencies.
  - "`imaginary-damping`": double (`0.01`)
    Small positive shift η, used to shift the real frequencies ω away from the real axis, $\omega \to \omega + i\eta$.

Monte Carlo integration parameters

Group "`Monte-Carlo-integration`":

- "`seed`": integer or string (`985456376`)
  Seed for the random number generator(s) used in the Monte Carlo integration. Instead of an integer, the string "`random`" can be passed to generate a *random* seed.
- "`warm-up-sweeps`": integer (`20`)
  Number of warm-up sweeps.
- "`sweeps-per-measurement`": double (`1.`)
  Number of sweeps per measurement.
- "`measurements-per-process-and-accumulator`": integer (`100`)
  Number of independent measurements each accumulator of each process performs.
- Subgroup "`threaded-solver`":
  - "`walkers`": integer (`1`)
    Number of Monte Carlo walkers.
  - "`accumulators`": integer (`1`)
    Number of Monte Carlo accumulators.

Monte Carlo solver parameters

◇ *CT-AUX*
Used if `CT-AUX` is selected as the cluster solver.
Group "`CT-AUX`":

- "`expansion-parameter-K`": double (`1.`)
  The perturbation order in the CT-AUX algorithm increases linearly with the expansion parameter K. While K is only subject to the restriction of being positive, values of the order of one have proven to be a good choice [18].
- "`initial-configuration-size`": integer (`10`)
  The CT-AUX solver is initialized with `initial-configuration-size` random *interacting* vertices.
- "`initial-matrix-size`": integer (`128`)
  Initial size of the CT-AUX matrices.
- "`max-submatrix-size`": integer (`128`)
  Maximum number of single spin updates per submatrix update.
- "`neglect-Bennett-updates`": boolean (`false`)
  Neglect the two types of Bennett updates:
  1. Removal of an interacting spin that has already been proposed for removal.
  2. Removal of a non-interacting spin that has been proposed for insertion.

  Turning on this option leads to a larger number of delayed spins ($\approx$`max-submatrix-size`) at the cost of a systematic error due to the violation of detailed balance.
- "`additional-time-measurements`": boolean (`false`)
  Do additional time measurements.

◇ *SS-CT-HYB*
Used if `SS-CT-HYB` is selected as the cluster solver.
Group "`SS-CT-HYB`":

- "`self-energy-tail-cutoff`": integer (`0`)
  Cut-off parameter for the imaginary frequency tail of the self-energy.
- "`steps-per-sweep`": double (`0.5`)
- "`shifts-per-sweep`": double (`0.5`)
  The fraction of insert/removal steps is given by `steps-per-sweep`/(`steps-per-sweep` + `shifts-per-sweep`), while the fraction of segments shifts is `shifts-per-sweep`/(`steps-per-sweep` + `shifts-per-sweep`).

Four-point parameters

Group "`four-point`":

- "`type`": string ("`NONE`")
  Four-point type, options are:
  - "`NONE`"
  - "`PARTICLE_PARTICLE_UP_DOWN`"
  - "`PARTICLE_HOLE_CHARGE`"
  - "`PARTICLE_HOLE_MAGNETIC`"
  - "`PARTICLE_HOLE_TRANSVERSE`"
- "`momentum-transfer`": array of doubles (null vector with the dimension of the lattice, e.g. for a 2D lattice: `[0., 0.]`)
  Transferred momentum $\mathbf{q}$. Must be an element of the reciprocal lattice of the DCA cluster ($\|\mathbf{q} - \mathbf{K}_{\text{DCA}}\|_2 < 10^{-3}$).
- "`frequency-transfer`": integer (`0`)
  Transferred frequency ω_n. Given as the index n of the bosonic Matsubara frequency $\omega_n = 2n\pi/\beta$.

Analysis parameters

Group "`analysis`":

- "`symmetrize-Gamma`": boolean (`true`)
  Symmetrize the cluster and lattice irreducible vertex functions according to the symmetry group and diagrammatic symmetries.

- "`Gamma-deconvolution-cut-off`": double (`0.5`)
  Cut-off parameter for the deconvolution of cluster irreducible vertex function. The deconvolution is done by Fourier transforming to real space, dividing by the coarse-graining patch functions in real space, $\phi(r)$, and then transforming back to reciprocal space. The real space patch functions are only inverted down to the value of this parameter,

$$\phi^{-1}_{\text{cut-off}}(r) = \begin{cases} \phi^{-1}(r), & \text{if } \phi^{-1}(r) > \texttt{Gamma} - \texttt{deconvolution} - \texttt{cut} - \texttt{off}, \\ 0, & \text{else.} \end{cases}$$

- "`project-onto-chrystal-harmonics`": boolean (`false`)
  Project the product $\Gamma(k, k')\chi_0(k)$ onto the crystal harmonic functions and diagonalize in this subspace.
- "`projection-cut-off-radius`": double (`1.5`)
  For the projection use crystal-harmonic functions of lattice vectors $\mathbf{r}$ with $\|\mathbf{r}\|_2 <$ `projection-cut-off-radius`.

ED-solver parameters

Only required if the exact diagonalization solver is used, e.g. in the application `cluster_solver_check`.
Group "`ED`":

- "`eigenvalue-cut-off`": double (`1.e-6`)
  Only keep energy eigenvalues E for which $e^{-\beta E} >$ `eigenvalue-cut-off`.

Double-counting parameters

Only required for LDA+DMFT.
Group "`double-counting`":

- "`method`": string ("`none`")
  The double-counting method to use, options are:
  - "`none`": no double-counting correction
  - "`constant-correction-without-U-correction`"
  - "`constant-correction-with-U-correction`"
- "`correction`": double (`0.`)
  The value of the double-counting correction.

Listing 3: JSON-formatted sample input file for a DCA$^+$ calculation.

```
{
    "output": {
        "directory": "./T=0.02/",
        "output-format": "HDF5",
        "filename-dca": "dca_tp.hdf5",
        "filename-analysis": "analysis.hdf5",
        "filename-profiling": "profiling.json",
        "dump-lattice-self-energy": false,
        "dump-cluster-Greens-functions": true,
        "dump-Gamma-lattice": false,
        "dump-chi-0-lattice": false
    },

    "physics": {
        "beta": 50,
        "density": 0.9,
        "chemical-potential": 0.,
        "adjust-chemical-potential": true
    },

    "single-band-Hubbard-model": {
        "t": 1.,
        "U": 4
    },

    "DCA": {
        "initial-self-energy": "./T=0.02/dca_sp.hdf5",
        "iterations": 1,
        "accuracy": 0.,
        "self-energy-mixing-factor": 1.,
        "interacting-orbitals": [0],

        "coarse-graining": {
            "k-mesh-recursion": 3,
            "periods": 2,
            "quadrature-rule": 1,
            "threads": 1,
            "tail-frequencies": 0
        },

        "DCA+": {
            "do-DCA+": true,
            "deconvolution-iterations": 16,
            "deconvolution-tolerance": 1.e-2
        }
    },

    "domains": {
        "real-space-grids": {
            "cluster": [[4, 6],
                        [8, -2]],
            "sp-host": [[20, 20],
                        [20,-20]],
            "tp-host": [[8, 8],
                        [8,-8]]
        },

        "imaginary-time": {
            "sp-time-intervals": 1024
        },

        "imaginary-frequency": {
            "sp-fermionic-frequencies": 1024,
            "four-point-fermionic-frequencies": 16
        }
    },

    "Monte-Carlo-integration": {
       "seed": 985456376,
       "warm-up-sweeps": 100,
       "sweeps-per-measurement": 1,
       "measurements-per-process-and-accumulator":
       100,

       "threaded-solver": {
            "walkers": 2,
            "accumulators": 20
        }
    },

    "CT-AUX": {
        "expansion-parameter-K": 1.,
        "initial-configuration-size": 64,
        "initial-matrix-size": 128,
        "max-submatrix-size": 256,
        "neglect-Bennett-updates": false,
        "additional-time-measurements": false
    },

    "four-point": {
        "type": "PARTICLE_PARTICLE_UP_DOWN",
        "momentum-transfer": [0., 0.],
        "frequency-transfer": 0
    },

    "analysis": {
        "symmetrize-Gamma": true,
        "Gamma-deconvolution-cut-off": 0.5,
        "project-onto-crystal-harmonics": false
    }
}
```

References

[1] P.W. Anderson, Science 235 (4793) (1987) 1196–1198.
[2] F.C. Zhang, T.M. Rice, Phys. Rev. B 37 (7) (1988) 3759–3761.
[3] A. Georges, G. Kotliar, W. Krauth, M.J. Rozenberg, Rev. Modern Phys. 68 (1) (1996) 13–125.
[4] T. Maier, M. Jarrell, T. Pruschke, M.H. Hettler, Rev. Modern Phys. 77 (3) (2005) 1027–1080.
[5] M.H. Hettler, M. Mukherjee, M. Jarrell, H.R. Krishnamurthy, Phys. Rev. B 61 (19) (2000) 12739–12756.
[6] M. Jarrell, T. Maier, C. Huscroft, S. Moukouri, Phys. Rev. B 64 (19) (2001) 195130.
[7] E. Gull, A.J. Millis, A.I. Lichtenstein, A.N. Rubtsov, M. Troyer, P. Werner, Rev. Modern Phys. 83 (2) (2011) 349–404.
[8] S. Moukouri, M. Jarrell, Phys. Rev. Lett. 87 (16) (2001) 167010.
[9] P. Staar, T. Maier, T.C. Schulthess, Phys. Rev. B 88 (11) (2013) 115101.
[10] P.K.V.V. Nukala, T.A. Maier, M.S. Summers, G. Alvarez, T.C. Schulthess, Phys. Rev. B 80 (19) (2009) 195111.
[11] P. Staar, T.A. Maier, T.C. Schulthess, J. Phys. Conf. Ser. 402 (1) (2012) 012015.
[12] G. Alvarez, M.S. Summers, D.E. Maxwell, M. Eisenbach, J.S. Meredith, J.M. Larkin, J. Levesque, T.A. Maier, P.R.C. Kent, E.F. D'Azevedo, T.C. Schulthess, Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, in: SC'08, IEEE Press, Piscataway, NJ, USA, 2008, pp. 61:1–61:10.
[13] P. Staar, T.A. Maier, M.S. Summers, G. Fourestey, R. Solca, T.C. Schulthess, Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, in: SC'13, ACM, New York, NY, USA, 2013, pp. 1:1–1:11.
[14] P. Staar, V. Mishra, U. Chatterjee, J.C. Campuzano, D.J. Scalapino, T.A. Maier, Nature Commun. 7 (2016) 11875.
[15] M. Jiang, U.R. Hähner, T.C. Schulthess, T.A. Maier, Phys. Rev. B 97 (18) (2018) 184507.
[16] U.R. Hähner, G. Balduzzi, P.W. Doak, T.A. Maier, R. Solcà, T.C. Schulthess, J. Phys. Conf. Ser. (2019) (in press).
[17] P. Staar, M. Jiang, U.R. Hähner, T.C. Schulthess, T.A. Maier, Phys. Rev. B 93 (16) (2016) 165144.
[18] E. Gull, P. Werner, O. Parcollet, M. Troyer, Europhys. Lett. 82 (5) (2008) 57003.
[19] S.M.A. Rombouts, K. Heyde, N. Jachowicz, Phys. Rev. Lett. 82 (21) (1999) 4155–4159.
[20] E. Gull, P. Staar, S. Fuchs, P. Nukala, M.S. Summers, T. Pruschke, T.C. Schulthess, T. Maier, Phys. Rev. B 83 (7) (2011) 075122.
[21] J.M. Bennett, Numer. Math. 7 (3) (1965) 217–221.
[22] J. Keiner, S. Kunis, D. Potts, ACM Trans. Math. Software 36 (4) (2009) 19:1–19:30.
[23] P. Werner, A.J. Millis, Phys. Rev. B 74 (15) (2006) 155107.
[24] P. Werner, A. Comanac, L. de' Medici, M. Troyer, A.J. Millis, Phys. Rev. Lett. 97 (7) (2006) 076405.
[25] H. Hafermann, K.R. Patton, P. Werner, Phys. Rev. B 85 (20) (2012) 205106.
[26] P. Staar, T. Maier, T.C. Schulthess, Phys. Rev. B 89 (19) (2014) 195133.
[27] T.A. Maier, in: E. Pavarini, E. Koch, P. Coleman (Eds.), Many-Body Physics: From Kondo to Hubbard, in: Modeling and Simulation, vol. 5, Forschungszentrum Jülich GmbH, Jülich, Germany, 2015.
[28] G. Kotliar, S.Y. Savrasov, K. Haule, V.S. Oudovenko, O. Parcollet, C.A. Marianetti, Rev. Modern Phys. 78 (3) (2006) 865–951.
[29] The HDF Group, Hierarchical data format, version 5, 1997–2018, https://www.hdfgroup.org/HDF5/.
[30] JSON, https://www.json.org/.
[31] K. Martin, B. Hoffman, Mastering CMake: A Cross-Platform Build System (CMake 3.1), Kitware, Inc., New York, NY, USA, 2015.
[32] K. Hoste, J. Timmerman, A. Georges, S.D. Weirdt, Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, in: SCC '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 572–582.
[33] IDEAS Scientific Software Productivity Project. https://ideas-productivity.org/resources/howtos/.
[34] B.W. Boehm, IEEE Softw. 1 (1) (1984) 75–88.
[35] K. Beck, Test Driven Development: By Example, Addison-Wesley Professional, Boston, MA, USA, 2002.
[36] Google test. https://github.com/google/googletest.
[37] Git, https://git-scm.com.
[38] Jenkins, https://jenkins.io.
[39] T.A. Maier, M. Jarrell, T.C. Schulthess, P.R.C. Kent, J.B. White, Phys. Rev. Lett. 95 (23) (2005) 237001.
[40] A.N. Rubtsov, A.I. Lichtenstein, J. Exp. Theor. Phys. Lett. 80 (1) (2004) 61–65.
[41] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, D. Fey, Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, in: PGAS '14, ACM, New York, NY, USA, 2014, pp. 6:1–6:11.
[42] SIMPLEX_GM_RULE: Grundmann-Moeller quadrature rules for the simplex in m dimensions. https://people.sc.fsu.edu/~jburkardt/f_src/simplex_gm_rule/simplex_gm_rule.html.