Optimizing the Performance of Parallel and Concurrent Applications Based on Asynchronous Many-Task Runtimes

Weile Wei
Louisiana State University and Agricultural and Mechanical College

Follow this and additional works at: https://repository.lsu.edu/gradschool_dissertations

Part of the Other Computer Sciences Commons, and the Software Engineering Commons

Recommended Citation
https://repository.lsu.edu/gradschool_dissertations/5884

This Dissertation is brought to you for free and open access by the Graduate School at LSU Scholarly Repository. It has been accepted for inclusion in LSU Doctoral Dissertations by an authorized graduate school editor of LSU Scholarly Repository. For more information, please contact gradetd@lsu.edu.
OPTIMIZING THE PERFORMANCE OF PARALLEL AND CONCURRENT APPLICATIONS BASED ON ASYNCHRONOUS MANY-TASK RUNTIMES

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Division of Computer Science and Engineering

by

Weile Wei
B.S., Shandong Jianzhu University, 2016
M.S., Louisiana State University, 2020
August 2022
Sapere Aude. Quia Veritas Vos Liberabit.

通向真理的道路泥泞满地，凶险异常。莽徒亡命，懦夫沉沦，智者孤独。
Acknowledgments

I would like to take this opportunity to thank my committee members: Dr. Hartmut Kaiser, Dr. Golden G. Richard III, Dr. Feng Chen, and Dr. Fang-ting Tu. Especially, heartfelt gratitude goes to my advisor Dr. Hartmut Kaiser, who has been an invaluable and patient mentor throughout my Ph.D. journey at Louisiana State University (LSU). In the past four years, Dr. Kaiser taught me C++ programming, guided me through the high-performance computing field, and sponsored my travels to academic conferences. I still remember his suggestions and encouragements when I first joined the STE||AR group: “Always ask questions” and “Your sky is unlimited”, which empowered me to conduct my Ph.D. research and continue to positively impact my career and life.

I would like to thank my collaborators Arghya “Ronnie” Chatterjee, Oscar Hernandez, Thomas Maier, Ed D’Azevedo, Peter Doak, John Biddiscombe, Giovanni Balduzzi, Peizhi Mai for their thoughtful research suggestions on the ORNL-LSU collaboration projects.

I would like to thank my colleagues and friends at STE||AR group, Adrian, Ali, Bibek, Bita, Chuanqiu, Katie, Max, Nanmiao, Parsa, Patrick, Rod, Shahrzad, Steve and Tianyi for their constructive discussions, invaluable suggestions, and necessary distractions. I would like to thank the Center for Computation and Technology at LSU for its amazing facilities.

Last but not the least, I would like to give my heartfelt gratitude to my family members for their continuous support and love for my studies in the U.S. I would like to take this opportunity to express my love to my parents using this ancient Chinese quote: “谁言寸草心，报得三春晖”. I would like to thank my wife, Lin, for her unending support in this journey and countless sacrifices to help me get to this point. They always believe in me and encourage me whenever I face difficulties so I can be fearless to overcome any challenges. They let me know that they have my back.

The rest of paragraphs in this chapter are the acknowledgement of funding agencies that support various chapters in this thesis.

For Chapter 2, authors would like to thank Thomas Maier (ORNL), Giovanni Balduzzi
ETH Zurich) and Ed D’Azevedo (ORNL) for their insights during the optimization phase
of DCA++. OLCF system admins Matt Belhorn (ORNL) and Ross Miller (ORNL) assisted
the authors with setting up libraries / programming models on Summit and Wombat. We
would further like to thank Marc Day and Kevin Gott (LBNL) for assisting us with allo-
cation on the CoriGPU at NERSC. This work 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 (ASCR) and Basic Energy Sci-
ences (BES) Division of Materials Sciences and Engineering, as well as the RAPIDS Sci-
DAC Institute for Computer Science and Data under subcontract 4000159855 from ORNL.
This research used resources of the Oak Ridge Leadership Computing Facility, which is
a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.
This research also used resources of the National Energy Research Scientific Computing
Center, a DOE Office of Science User Facility supported by the Office of Science of the
U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

For Chapter 3, authors would like to thank Thomas Maier (ORNL), Giovanni Balduzzi
(ETH Zurich) for their insights during the optimization phase of DCA++. This work 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 (ASCR) and Basic Energy Sciences (BES) Division of Materials Sciences and En-
gineering, as well as the RAPIDS SciDAC Institute for Computer Science and Data under
subcontract 4000159855 from ORNL. This research used resources of the Oak Ridge Lead-
ership Computing Facility, which is a DOE Office of Science User Facility supported under
Contract DE-AC05-00OR22725, and Center for Computation & Technology at Louisiana
State University.

For Chapter 4, the authors would like to thank Manuel Arenaz (Appentra Solutions),
Hartmut Kaiser (Louisiana State University), and Kevin Huck (University of Oregon) for
their guidance and feedback on this work. This work was supported by the Scientific
Discovery through Advanced Computing (SciDAC) program funded by US Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR) and Basic Energy Sciences (BES) Division of Materials Sciences and Engineering. This research was also supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the US Department of Energy Office of Science and the National Nuclear Security Administration, in particular its subproject on Scaling OpenMP with LLVM for Exascale performance and portability (SOLLVE). Notice: This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan). This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-819815).
Table of Contents

Acknowledgements .................................................................................................................. v
List of Tables ........................................................................................................................ viii
List of Figures ........................................................................................................................ ix
Abstract ................................................................................................................................... xi

Chapter
1. Introduction ......................................................................................................................... 1
   1.1. Dissertation Outline ................................................................................................. 1

2. Speed-up Scientific Application Using HPX Threading Back-end .................................. 3
   2.1. Introduction ................................................................................................................ 3
   2.2. Background ............................................................................................................... 4
   2.3. Threading Abstraction Implementation .................................................................... 8
   2.4. Performance Analysis Results ................................................................................. 12
   2.5. Conclusions ............................................................................................................ 20

3. Solve Memory-bound Issue by Ring-based Communication Algorithms .................... 21
   3.1. Introduction ............................................................................................................... 21
   3.2. Background ............................................................................................................. 22
   3.3. Ring-based Communication Algorithms Implementation ................................... 24
   3.4. Results ..................................................................................................................... 31
   3.5. Discussions ............................................................................................................. 35

4. Optimize SIMD Code Using LLVM-based Analysis on Arm Supercomputer ............ 41
   4.1. Introduction ............................................................................................................... 41
   4.2. Case Study: Porting DCA++ to Wombat ................................................................. 41
   4.3. An LLVM Tool Methodology to Generate Efficient Vectorization ..................... 47
   4.4. Automating the Process: The OpenMP Advisor ...................................................... 53
   4.5. Related Work .......................................................................................................... 54
   4.6. Conclusions ............................................................................................................ 54

5. Conclusions ...................................................................................................................... 56
   5.1. Contributions ........................................................................................................... 56

Appendix
A. Copyright Information for Chapter 2 ................................................................................. 58
B. Copyright Information for Chapter 3 ................................................................................. 60
C. Copyright Information for Chapter 4 ................................................................................. 62
List of Tables

2.1. Systems Comparison ........................................................ 12
3.1. Difference between original $G_t$ and accumulate $G^d_t$ over 5 runs .......... 34
List of Figures

2.1. Shows the computation structure of a threaded QMC kernel using the custom-made thread pool in DCA++ running on a single MPI process (rank). ................................................................. 5

2.2. Custom-made thread pool in DCA++ now supports both std::thread (default) and hpx::thread (new feature). Threading options can be toggled at compilation. ................................................... 9

2.3. DCA++ correctness verification across multiple architectures as outlined in Table 2.1......................................................... 14

2.4. Time-to-solution for 100k Monte Carlo measurements with error bars obtained from 5 independent executions on Summit ...................... 15

2.5. Comparison of non-voluntary and voluntary context switches using the APEX performance measurement counters when executing DCA++ with C++ std::thread and hpx::thread versions on Summit. ................................................................. 16

2.6. NVIDIA Nsight System profiler showing CPU utilization; for C++ std::thread (left, shows only 28 active hyper-threads) and hpx::thread (right) versions for DCA++. ...................................... 17

2.7. HPX-APEX profiling results on Summit summarizing CPU and GPU activities............................................................... 18

2.8. HPX-APEX profiling on Summit showing HPX thread idle rate and queue length. ................................................................. 19

2.9. HPX-APEX profiling results on Summit summarizing device memory used (unit: megabyte) over the time. .............................. 19

3.1. Workflow of the QMC DCA++ solver........................................ 22

3.2. Compare original $G_t$ v.s. distributed $G_t^{d}$ implementation. Each rank contains one GPU resource. ........................................... 26

3.3. Workflow of ring algorithm per iteration. ................................. 27

3.4. Workflow of subring algorithm per iteration. .......................... 30

3.5. Architectural layout of a single node on the Summit supercomputer. ......... 32

3.6. cudaMalloc requested size (GBytes) over time visualized by Vampir. 34
3.7. Time for 1400 iterations (per rank) of the subring algorithm using 6 ranks per node on Summit and each message size is 170 MBytes. ...... 36

3.8. Vampir timeline graph shows the processes activities over the time in rank 0 (DCA++ with multi-threaded ring algorithm). ................ 37

3.9. Device memory used (GBytes over time) when using 7 walker-accumulator thread. Visualized by Vampir. .............................. 38

3.10. Device memory used (GBytes over time) when using 1 walker-accumulator thread. Visualized by Vampir. .............................. 39

4.1. DCA++ execution time. .................................................... 44

4.2. DCA++ timing breakdown. ................................................. 45

4.3. PAPI counter for DCA++ runs on A64FX. ............................... 46

4.4. A loop performing a parallel reduction that is not vectorized automatically. ... 49

4.5. A loop performing a memory gather that requires OpenMP SIMD to be vectorized by the ARM compiler. ............................... 50

4.6. A code block using the math library functions $\cos$ and $\sin$. ................. 51

4.7. A loop requiring a source transformation and OpenMP SIMD (left) and its transformed version (right). ............................... 52

4.8. The loops in Figs. 4.6, 4.5, 4.7, and 4.4, respectively, before and after the barriers to SVE execution were remedied. ................. 53
Abstract

Nowadays, High-performance Computing (HPC) scientific applications often face performance challenges when running on heterogeneous supercomputers, so do scalability, portability, and efficiency issues. For years, supercomputer architectures have been rapidly changing and becoming more complex, and this challenge will become even more complicated as we enter the exascale era, where computers will exceed one quintillion calculations per second. Software adaptation and optimization are needed to address these challenges. Asynchronous many-task (AMT) systems show promise against the exascale challenge as they combine advantages of multi-core architectures with light-weight threads, asynchronous executions, smart scheduling, and portability across diverse architectures.

In this research, we optimize the performance of a highly scalable scientific application using HPX, an AMT runtime system, and address its performance bottlenecks on supercomputers. We use DCA++ (Dynamical Cluster Approximation) as a research vehicle for studying the performance bottlenecks in parallel and concurrent applications. DCA++ is a high-performance research software application that provides a modern C++ implementation to solve quantum many-body problems with a Quantum Monte Carlo (QMC) kernel. QMC solver applications are widely used and are mission-critical across the US Department of Energy’s (DOE’s) application landscape.

Throughout the research, we implement several optimization techniques. Firstly, we add HPX threading backend support to DCA++ and achieve significant performance speedup. Secondly, we solve a memory-bound challenge in DCA++ and develop ring-based communication algorithms using GPU RDMA technology that allow much larger scientific simulation cases. Thirdly, we explore a methodology for using LLVM-based tools to tune the DCA++ that targets the new ARM A64Fx processor. We profile all implementations in-depth and observe significant performance improvement throughout all the implementations.
Chapter 1. Introduction

Many High-performance Computing (HPC) scientific applications that run on supercomputers often face challenges including performance, scalability and portability. For years, the complexity of supercomputer architectures have been rapidly increasing, and this challenge will become even more sophisticated as we enter the exascale era, where computers will exceed one quintillion calculations per second. Asynchronous many-task (AMT) systems show promise against the exascale challenge as they combine advantages of multi-core architectures with light-weight threads, asynchronous executions, and smart scheduling.

In this research, we optimize performance of a highly scalable scientific application using HPX, an AMT runtime system. We use DCA++ (Dynamical Cluster Approximation) as a research vehicle for studying performance bottlenecks in parallel and concurrent applications. DCA++ [1–4] is a high-performance research software application that provides a modern C++ implementation to solve quantum many-body problems with a Quantum Monte Carlo (QMC) kernel. QMC solver applications are widely used and are mission-critical across the US Department of Energy’s (DOE’s) application landscape.

HPX is a C++ Library for concurrency and parallelism [5]. It is a runtime system written using modern C++ techniques that are linked as part of an application. HPX exposes extended services and functionalities supporting the implementation of parallel, concurrent, and distributed capabilities for applications in any domain; it has been used in scientific computing, gaming, finances, data mining, and other fields.

1.1. Dissertation Outline

This rest of the dissertation is structured as follows:

- Chapter 2 presents the study of porting and evaluating the performance of HPX-backed quantum monte carlo (QMC) solver in various supercomputer architectures.
- Chapter 3 presents the design and implementation of ring-based GPU RDMA algorithms for solving memory-bound challenges of QMC solver.
- Chapter 4 presents the study of porting and evaluating performance of HPX-based
QMC application in Arm A64fx using LLVM-based tools.

- Chapter 5 concludes the dissertation.
Chapter 2. Speed-up Scientific Application Using HPX Threading Backend

2.1. Introduction

As users move their applications toward accelerated node architectures of different accelerator types and next-generation multi-core systems, they encounter significant challenges in their codes as there are few programming models available on all of these new architectures that can interoperate well with C++ and vendor specific APIs and libraries. Our goal is to examine how successfully we can use the HPX programming model to port codes between architectures, and what lessons we can learn from this experience. HPX also helps raise the level of abstraction in the application’s programming model in order to understand common performance problems across architectures. This helps to identify common optimization opportunities to hide latency, overheads, serializations and wait times while bringing performance improvements “off-the-shelf” to the application originally written using parallelism in C++. In this chapter, we explain which performance issues HPX can address and describe how we use it in the DCA++ application, its evaluation on different platforms, and how we can tune it to target to multiple platforms. With rapidly changing configurations of highly heterogeneous HPC systems, portability of code and performance of scientific applications is paramount for their software design and development efforts and long sustainability of applications.

DCA++ (Dynamical Cluster Approximation) is a high-performance research software framework, providing a modern C++ implementation to solve quantum many-body prob-
lems [1–3]. The DCA++ code currently uses three different programming models (MPI, CUDA, and C++ Standard threads), together with numerical libraries (BLAS, LAPACK and MAGMA), to expose the parallelization in computations.

HPX is a C++ Standard Library for Concurrency and Parallelism [6–9]. It implements all of the corresponding facilities as defined by the C++ Standard. Additionally, in HPX we implement functionalities proposed as part of the ongoing C++ standardization process.

In this chapter, we outline HPX as a potential solution to efficiently porting DCA++ across different architectures.

2.2. Background

Quantum Monte Carlo (QMC) solver applications are common tools and mission critical across the US Department of Energy’s (DOE) application landscape. For the purpose of this manuscript the authors choose to use one of the leading QMC applications, developed primarily at Oak Ridge National Laboratory in collaboration with ETH Zürich, the Dynamical Cluster Approximation (DCA++) algorithm. In recent years DCA++ has been ported and successfully optimized across various platforms (on both host side and accelerator based devices). A production scale scientific problem runs on the DOE’s fastest supercomputer, Summit, at Oak Ridge Leadership Facility (OLCF) on all 4600 nodes equipped with ~28000 NVIDIA Volta V100 GPUs attaining a peak performance of 73.5 PFLOPS with a mixed precision implementation [10].

Although DCA++ has been highly optimized on existing hardware, this is the first effort to focus on the runtime execution level of the application and observe how it performs on each of the already supported systems and newer DOE supported architectures. In this work, the authors enable HPX runtime support to further optimize thread context switching and lower synchronization cost over the usage of C++ standard threads. We further verify such claims using the APEX performance measurement tool.
2.2.1. DCA++

Dynamical Cluster Approximation (DCA++) is a numerical simulation tool that is used to predict behaviors of quantum materials, such as superconductivity, magnetism, etc. It is an iterative convergence algorithm with two primary kernels: (a) Coarse-graining of the single-particle Green's function to reduce the complexity of the infinite size lattice problem to that of an effective finite size cluster problem, and, (b) Quantum Monte Carlo (QMC) based solver for the cluster problem.

![Diagram of computation structure](image)

Figure 2.1. Shows the computation structure of a threaded QMC kernel using the custom-made thread pool in DCA++ running on a single MPI process (rank). We run multiple walker threads concurrently, and after each walker finishes an MC update, an idle accumulator thread is pulled from the head of accumulator waiting queue to compute MC measurement from the walker. After the accumulator finishes its measurement, it’s pushed to the back of the queue.

Most of the application’s performance, workload (computation), memory usage and bottlenecks come from the QMC solver kernel [10]. Fig. 2.1 shows the on-node (per MPI process) computation structure of a threaded QMC simulation using the custom-made thread pool in DCA++. We initialize several instances of independent Markov chains and distribute across nodes (MPI ranks), each node is responsible for that Markov chain assigned \(^1\), computed by a walker object (producer) and an accumulator object (accumulator) that measures single- and two-particle Green’s functions.

\(^1\)On systems with the ability to run multiple MPI ranks per node with one or more GPUs per rank, each process is then only responsible for a portion of the chain assigned to that node.
Each object runs on an independent thread and no communication happens between these threads. We run multiple *walker* threads concurrently, and after each *walker* finishes a Monte Carlo (MC) update (sampling from the Markov chain), the *accumulator* is pulled from the head of *accumulator waiting queue* to compute MC measurement from the *walker*. When each *accumulator* finishes its measurement, it’s pushed into the back of the queue. The queries to the queue are managed by the synchronization primitives (*mutex* and *conditional_variable*).

In this chapter the analysis, optimization, and further performance gains will be discussed in reference only to the QMC solver portion of the DCA++ application.

### 2.2.2. HPX

HPX is a C++ standard library for distributed and parallel programming built on top of an asynchronous many-task runtime system (AMT). It has been described in detail in other publications [5, 8, 11–14]. Such AMT runtimes provide a means for helping programming models to fully exploit available parallelism on complex emerging HPC architectures. The HPX runtime includes the following essential components:

- An ISO C++ standard conforming API that enables wait-free asynchronous parallel programming, including *Futures*, *Channels*, and other primitives for asynchronous execution. The exposed API ensures syntactic and semantic equivalence of local and remote operations, which greatly simplifies writing complex applications [15, 16].
- A work-stealing lightweight task scheduler [7, 17] that enables finer-grained parallelization and synchronization, exposes greatly reduced overheads related to threading, and ensures automatic load balancing across all local compute resources (see 2.3).
- APEX [18], an *in-situ* profiling and adaptive tuning framework (see 2.2.3).
- In its distributed version (not utilized in the presented work), HPX also features an Active Global Address Space (AGAS) [13, 19] that supports load balancing via object migration and enables runtime-adaptive data placement and distributed garbage collection and an active-message networking layer that enables running functions close
to the objects they operate on [7, 20].

In the context of the presented work we use HPX because of its full conformance to the recent C++ standards [21, 22], its reduced thread and synchronization overhead properties, and its sophisticated performance measurement and in-situ profiling capabilities provided by APEX.

2.2.3. HPX-APEX Integration

APEX [18] (Autonomic Performance Environment for eXascale) is a performance measurement library for distributed, asynchronous multitasking runtime systems such as HPX. It provides support for both lightweight measurement and high concurrency. To support performance measurement in systems that employ user-level threading, APEX uses a dependency chain in addition to the call stack to produce traces and task dependency graphs. APEX supports both synchronous (so-called first person) and asynchronous (third person) measurements. The synchronous module of APEX uses an event API and event listeners. Whenever an HPX task is created, started, yielded or stopped, APEX will respectively create, start/resume, yield, or stop timers for measurements. Dependencies between tasks are also tracked. The asynchronous measurement involves periodic or on-demand interrogation of operating system, hardware or runtime states (e.g. CPU utilization, resident set size, memory “high water mark”). HPX counters (e.g. idle rate, queue lengths) are also captured on-demand on a periodic basis.

APEX has native support for performance profiling, in which all tasks scheduled by the runtime are measured and a report is output to disk and/or the screen at the end of execution. The profile data contains the number of times each task was executed and the total time spent executing that type of task. In order to perform detailed performance analysis involving synchronization and/or task dependency analysis, full event traces including event identification and start/stop times have to be captured. To that end, APEX is integrated with the Open Trace Format 2 [23] (OTF2) library – an open, robust format for large scale parallel application event trace data. OTF2 is a robust reader/writer library and
binary format specification that is typically used for high-performance computing (HPC) trace data. In order to capture full task dependency chains in HPX applications, all tasks are uniquely identified by their GUID (globally unique identifier) and the GUID of their parent task. These GUIDs are captured as part of the OTF2 trace output. OTF2 data can be visualized by the Vampir [24] trace analysis tool.

Before the DCA+HPX integration, the first person measurement in APEX was only integrated with a handful of technologies, including the HPX runtime and OpenMP 5.0 runtimes that support the OMPT performance tools interface [25]. The third person measurement in APEX was mostly limited to extracting data from HPX and the Linux /proc virtual filesystem. Because most of the DCA++ computation is offloaded to GPUs using the CUDA library, APEX was integrated with the CUDA Profiling Tools Interface (CUPTI) [26] and the NVIDIA Management Library (NVML) [27]. Synchronous CUDA API callback timers and some counters (e.g. Bytes transferred, bandwidth, vector lanes) from the CUDA runtime and/or device API are captured synchronously, whereas the NVML counters (e.g. utilization, bandwidth, power) are periodically captured asynchronously. Using APEX GUIDs mapped from CUDA Correlation IDs, the GPU activity such as memory transfers and kernel executions are captured and linked to the host-side tasks that launched them. To provide concurrent use of the GPU hardware, memory transfers between the host and GPU and kernels are executed within logical subdivisions of the device, identified by the device, context, and stream IDs. These IDs are associated with the OTF2 virtual “threads” of execution within the trace data, as shown in Fig. 2.7.

2.3. Threading Abstraction Implementation

In this section, we outline our implementation of the high-level threading abstraction layer in DCA++, which supports standard C++ threading and HPX threading implementations\(^2\). The design of HPX integration in DCA++ is presented in Fig. 2.2. Our implementation is non-intrusive to DCA++ code as it does not break the API of the custom-

\(^2\)https://github.com/STEllAR-GROUP/DCA/releases/tag/hpx_thread
made thread pool and we have not modified original DCA++ workflow. It also allows the
application developer to switch between hpx::thread and std::thread via compilation config-
uration. If user prefers HPX threading option, one needs to turn on DCA_WITH_HPX flag
and provide the path of HPX library to the application’s CMake configuration.

Figure 2.2. Custom-made thread pool in DCA++ now supports both std::thread (default)
and hpx::thread (new feature). Threading options can be toggled at compilation.

To parallelize computation tasks, DCA version 1.1.0\(^3\) implemented a multi-threading
strategy using POSIX threads which could cause large overheads when thousands of threads
continuously spawned and joined. DCA version 2.0\(^4\) lowered the overhead with the custom-
made thread pool strategy [10] by maintaining constant number of C++ std::thread objects
during the execution. However, the implementation of the custom-made thread pool strat-
egy was designed to spread worker threads to simultaneous multithreading (SMT) or virtual
cores. Depending on the architecture of the processor, SMT might be a bottleneck if any
of the SMT threads are competing for the shared physical core [28].

We manage to preserve the same API of the ThreadPool implementation in both versions
primarily due to the fact that HPX is fully C++ standard conforming. All synchronization
primitives of the standard C++ library are still valid in the context of HPX. For the C++
std::thread version of the thread pool shown in Listing 2.1, we wrapped all C++ standard
synchronization primitives (i.e. condition_variable, lock_guard, future) into a thread_traits class.

\(^3\)https://github.com/CompFUSE/DCA/releases/tag/paper.2019.old_code
\(^4\)https://github.com/CompFUSE/DCA/releases/tag/paper.2019.new_code
For the HPX-enabled DCA++ shown in Listing 2.2, we construct a similar thread_traits class in a separate header file and replace all the C++ standard synchronization primitives with equivalent HPX synchronization primitives.

Listing 2.1. std::thread version of the thread pool.

```cpp
namespace dca { namespace parallel {
    struct thread_traits {
        template <typename T>
        using future_type = std::future<T>;
        using mutex_type = std::mutex;
        using condition_variable_type = std::condition_variable;
        using scoped_lock = std::lock_guard<mutex_type>;
        using unique_lock = std::unique_lock<mutex_type>;
    };

    class ThreadPool {...};
}}
```

Listing 2.2. hpx::thread version of the thread pool. Note that for the synchronization primitives implemented in class thread_traits, this version differs from the std::thread version only by the used C++ namespace hpx.

```cpp
namespace dca { namespace parallel {
    struct thread_traits {
        template <typename T>
        using future_type = hpx::future<T>;
        using mutex_type = hpx::mutex;
        using condition_variable_type = hpx::condition_variable;
        using scoped_lock = std::lock_guard<mutex_type>;
        using unique_lock = std::unique_lock<mutex_type>;
    };

    class ThreadPool {...};
}}
```
For task-scheduling in the custom-made thread pool implemented in class ThreadPool, the C++ std::thread version of the thread pool [10] maintains an array of std::thread objects and array of queues of work items represented by std::packaged_task objects in a simple round-robin fashion; HPX threading version dispatches tasks asynchronously through hpx::async and manages tasks with its runtime scheduler that has various robust task scheduling methods [29].

For thread affinity, the C++ std::thread version of the thread pool manually sets thread affinity and uses the (SMT) feature to achieve speedup [10]; the hpx::thread version on the other hand handles these scheduling efforts automatically through its runtime system. HPX by default recognizes existing SMT and sets only one hyper-thread per physical processing unit. The runtime schedules user-level lightweight threads on top of operating system threads, which avoids expensive context switches at kernel-level [17].

HPX-threads are implemented as user-level threads. These are cooperatively (non-preemptively) scheduled in user mode by the HPX-thread manager on top of one OS thread per hardware thread (processing unit). By default, the OS threads have their affinities defined such that they run on one processing unit only. The HPX-threads can be scheduled without a kernel transition, which provides a performance boost. Additionally, the full use of the OS’s time quantum per OS-thread is achieved even if an HPX-thread blocks for any reason. In that case, other HPX-threads are scheduled to run immediately. The scheduler is cooperative in the sense that it will not preempt a running HPX-thread until it finishes execution or cooperatively yields its execution. This is particularly important, since it avoids context switches and cache thrashing due to randomization introduced by preemption. The default thread scheduler is implemented as a ‘First Come First Served’ scheduler, where each OS-thread works from its own queue of HPX-threads. Other scheduling policies, e.g. supporting thread priorities, are available as well. If one of the cores runs out of work, it
starts ‘stealing’ queued tasks from neighboring cores, thus enabling load-balancing across all cores [7, 17].

2.4. Performance Analysis Results

2.4.1. Systems Overview

For our evaluation, we have used Oak Ridge Leadership Computing Facility’s (OLCF) Summit supercomputer and the Wombat system; and, National Energy Research Scientific Computing Center’s (NERSC) Cori Supercomputer (for this work we used the new CoriGPU partition). Each system was selected due to its host architecture diversity (shown in Table 2.1) for comparing the performance of DCA++ using the HPX runtime and visualizing the results collected using APEX and visualized by Vampir.

**Summit.** [30] is a 4600 node, 200 PFLOPS IBM AC922 system. Each node consists of 2 IBM POWER9 CPUs with 512 GB DDR4 RAM and 6 NVIDIA V100 GPUs with total of 96 GB high bandwidth memory (divided into 2 sockets), all connected together with NVIDIA’s high-speed NVLink.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Summit</th>
<th>Wombat</th>
<th>CoriGPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>NVIDIA Volta (6 per node)</td>
<td>NVIDIA Volta (2 per node)</td>
<td>NVIDIA Volta (8 per node)</td>
</tr>
<tr>
<td>CPU</td>
<td>IBM POWER9™ (2 Sockets / 21 Cores per socket)</td>
<td>Cavium ThunderX2 (2 Sockets / 28 Cores per socket)</td>
<td>Intel Xeon Gold 6148 (2 sockets / 20 cores per socket)</td>
</tr>
<tr>
<td>CPU-GPU interconnect</td>
<td>NVIDIA NVLINK2 (50 GB/s)</td>
<td>PCIe Gen3 (16 GB/s)</td>
<td>PCIe Gen3 (16 GB/s)</td>
</tr>
</tbody>
</table>

**Wombat.** [32] is a 64-bit ARM cluster with 16 compute nodes, four of which have two NVIDIA V100 GPUs attached. Each compute node has two 28-core Cavium ThunderX2 processors (Cavium is now Marvell), 256 GB RAM (16 DDR4 DIMM’s) and a 480 GB SSD for node-local storage. Nodes are connected with EDR InfiniBand (~100 Gbit/s).

---

5Summit ranked the second place in the TOP500 list in June 2020 [31]
CoriGPU, [33] is a development rack of 18 nodes recently added to the Cori system at NERSC. Each node contains two 20-core Intel Xeon Gold 6148 CPUs with 384GB DDR4 memory and 8 NVIDIA V100 GPUs with 128 GB HBM2 memory (divided into 2 sockets). All GPUs are connected to the CPUs and Infiniband network interface cards via PCIe 3.0.

2.4.2. Correctness Verification Across Systems

To verify the correctness of our work across various HPC architectures, we follow the standard DCA++ protocol\footnote{https://github.com/CompFUSE/DCA/wiki/Tutorial:-Tc} to study superconductivity in the 2D single-band Hubbard model in DCA++. The focus value is the superconducting transition temperature $T_c$, a property of the materials. We choose 100k Monte Carlo measurements as it is representative case to our science problems. The goal is to obtain the same $T_c$ with acceptable statistical noise across all HPC architectures for a specific scientific case as defined under the protocol.

Fig. 2.3a shows DCA++ with C++ std::thread threading generates consistent results across various platforms. It shows the temperature dependence of the leading eigenvalue $\lambda_d$ of the Bether-Salpeter equation. $T_c$ is the temperature where $\lambda_d(T=T_c) = 1$. All $T_c$ are about 0.076 within acceptable statistical range. Similarly, Fig. 2.3b shows DCA++ with hpx::thread also generates accurate results across multiple HPC architectures. We use the DCA++ application with C++ std::thread threading results obtained from runs on Summit as a referencing result, and compare with all other runs of DCA++ using hpx::thread on various platforms. As one might note that we have obtained the same $T_c$ within an acceptable statistical deviation.

2.4.3. Compare Runtime: std::thread v.s. hpx::thread

For this comparison analysis we compared a version of DCA++ with C++ std::thread and one with a hpx::thread implementation on a single Summit node with 6 MPI ranks, each rank mapped to 7 physical cores and 1 Volta V100 GPU. More performance analysis (i.e. performance analysis on other machines) will be uploaded to the public repository \footnote{https://github.com/STEllAR-GROUP/dca} once

---

6https://github.com/CompFUSE/DCA/wiki/Tutorial:-Tc
7https://github.com/STEllAR-GROUP/dca
Here we validate our science case with C++ `std::thread` implementation across three HPC platforms. Additionally, we show the C++ `std::thread` results on Summit as a reference.

For each platform, we compute DCA++ with 100K Monte Carlo measurements (representative case to our science problems) for 5 independent calculations. The random number generator used in all experiments is `std::mt19937_64` from C++ library.

Figure 2.3. DCA++ correctness verification across multiple architectures as outlined in Table 2.1. For our scientific problem we obtain same superconducting transition temperature $T_c$ results (where leading eigenvalue $\lambda_d(T=T_c) = 1$) within acceptable statistical range. For each platform, we compute DCA++ with 100K Monte Carlo measurements (representative case to our science problems) for 5 independent calculations. The random number generator used in all experiments is `std::mt19937_64` from C++ library.
Figure 2.4. Time-to-solution for 100k Monte Carlo measurements with error bars obtained from 5 independent executions on Summit. Using the hpx::thread implementation we observe up to 21% speedup over the C++ std::thread version. Observed performance gain is due to faster context switch and scheduler and less synchronization overhead in HPX runtime system. Lower is better.

Fig. 2.4 shows DCA++ with hpx::thread achieves 21% speedup over the one with C++ std::thread version. The same improvement is also observed in the distributed runs as well. The speedup is mainly due to faster thread context switching and reduced scheduler and synchronization overheads in the HPX runtime system (see Section 2.3). Fig. 2.5 verifies the speedup and shows by the end of the execution, hpx::thread version has much less (∼ 2× lower) voluntary context switches (639 times) relative to std::thread version (1454 times) and ∼ 4× lower non-voluntary context switches (18 times) relative to std::thread version (70 times). For the non-voluntary context switches observed in hpx::thread version, we consider these are most likely caused by the synchronization introduced by CUDA itself as CUDA synchronization is still happening on pthread level.

Fig. 2.6 was generated using the NVIDIA Nsight Systems on Summit. The figure shows two different threading affinity strategies adapted in C++ std::thread (left) and hpx::thread version (right) in DCA+++. Each row in the figure represents average hardware thread utilization. The height of the hardware thread utilization is represented by the height of the black histogram.

For our test case we set the SMT to 4 for both executions. The C++ std::thread version uses 4 hardware threads per physical core; while, HPX-enabled DCA++ by default utilizes only one hardware thread per physical core. Also, if we combine the adjacent 4 hyper-threads (SMT) for each physical core in C++ std::thread version, the overall utilization is not as high as in the hpx::thread version. Moreover, even if the DCA++ is modified to use
Figure 2.5. Comparison of non-voluntary and voluntary context switches using the APEX performance measurement counters when executing DCA++ with C++ std::thread and hpx::thread versions on Summit. We observe that the hpx::thread implementation has much less context switches than std::thread in DCA++ and aides to the performance gains in using HPX over std::thread. Lower is better.

The same affinity settings (which requires explicit changes in the code base) as HPX, the performance is not improved (i.e. the affinity settings do not cause the speedup of HPX). The reason of the speedup is due to the fact that HPX thread management (and context switching in particular) exposes less overheads and lower synchronization overheads. With faster context switch from HPX threads, DCA++ is able to feeds more computing workload into GPU faster. This directly increases the GPU utilization resulting in the observed performance improvement.

We further verified that thread caching malloc (i.e. tcmalloc) is not the cause of the speedup with hpx::thread version which uses tcmalloc. TCMalloc assigns each thread a thread-local cache and reduces lock contention for multi-threaded programs [34]. We performed LD_PRELOAD tcmalloc for DCA++ std::thread version, and the execution time remains the same as the one without tcmalloc. This finding strengthens our conclusion that the 21% speedup seen for the hpx::thread version is due to the fact that user-level context switching is more efficient and synchronization with HPX threads imposes less overhead (see Fig. 2.4).
Figure 2.6. NVIDIA Nsight System profiler showing CPU utilization; for C++ std::thread (left, shows only 28 active hyper-threads) and hpx::thread (right) versions for DCA++. hpx::thread version sets one hyper-thread per physical core to achieve better hardware utilization while std::thread spreads work over 4 hyper-threads per physical core.

2.4.4. HPX-APEX Profiling Analysis

APEX was originally designed to be integrated with the HPX runtime, and enabling APEX support is straightforward. When configuring HPX, flags are passed to CMake in order to enable APEX support and provide the path to library dependencies such as OTF2, CUPTI and NVML. After configuration, build and installation the HPX runtime will have APEX performance measurement enabled. As mentioned in Section 2.2, all HPX tasks are timed by APEX. In addition, tasks defined in the application can be annotated to provide unique labels using the hpx::annotated_function facility in HPX. At runtime, different APEX features (e.g. tracing, output summary format, different counter sets) are enabled/disabled through the use of environment variables, a configuration file, or the APEX programming interface.

For the experiments described below, APEX collected a full event trace to OTF2 and monitored several HPX, operating system, CPU and GPU utilization counters. Counters
(a) Master timeline plot monitored events including CPU and GPU activities

(b) Top 10 time consuming functions. Both annotated functions (user defined kernels) and CUDA API calls can be captured. Exclusive time means the amount of time spent in just this function and no subroutines are included.

Figure 2.7. HPX-APEX profiling results on Summit summarizing CPU and GPU activities. that were particularly useful for these experiments include kernel-level context switches (both voluntary and not), user and system level CPU utilization, GPU utilization and memory consumption, HPX idle rates and queue lengths.

We traced DCA++ with APEX on Summit as shown in Fig. 2.7. We are able to annotate any functions with hpx::annotated_function function wrapper in the code to distinguish their execution time in final profiling data. Here we annotate walker and accumulator functions, as they are the most computation-intensive parts in DCA++ code. From Fig. 2.7b, one can clearly observe that the walker function takes majority of the time in a single DCA++ run. The profiling measurement library can also gather HPX thread idle rate (as seen in Fig. 2.8a) and queue length (as seen in Fig. 2.8b). The idle rate counter indicates how utilized each of the HPX worker threads are during each sampled time period (lower is better). In the context of HPX, it is not a problem having the shown queue lengths as
creating and managing HPX threads (tasks) is generally very cheap (less than 1 µs per thread). The queue depth indicates how much work, in the form of queued tasks, is available for each of the worker threads. The counters are collected on a per-worker basis, and the values shown here represent averages across all worker threads.

In [10] authors reported that while storing two-particle green function \( G_{tp} \) on the device allows condensed matter physicists to explore larger and more complex (higher fidelity) science problems, but we are limited to the device memory size. The data of device memory usage from HPX-APEX shown in Fig. 2.9 can help us track memory usage and provide computational scientists guidance on how to address memory-bound challenge as defined in [10]. We are planning to distribute \( G_{tp} \) across nodes and implement a token ring algorithm to transfer single-particle Green’s function \( G \) between nodes. The implementation will take advantage of high-speed network between devices available on the machine (i.e. NVIDIA NVLink on Summit) in order to transfer device data efficiently.
2.5. Conclusions

In this chapter we used the Dynamical Cluster Approximation (DCA++) one of the leading Quantum Monte Carlo solvers as a research vehicle to test the feasibility of the HPX runtime system and use the abstraction layer in the programming model to understand the performance bottlenecks across multiple architectures (both host side and accelerator based devices).

We observed significant performance benefit (~21% speedup over standard threads) by just using the HPX threading model due to the faster context switches and lower synchronization overheads guaranteed by the HPX runtime. In this work we also validated our claims using the APEX performance measurement library and with the HPX-APEX integration one can observe in-depth analysis of the threading behavior (eg. CPU / GPU utilization counters, device memory allocation over time, kernel level context switches and more).
Chapter 3. Solve Memory-bound Issue by Ring-based Communication Algorithms

3.1. Introduction

DCA++ (Dynamical Cluster Approximation) is a high-performance research software application [1–4] that provides a modern C++ implementation to solve quantum many-body problems. DCA++ implements a quantum cluster method with a Quantum Monte Carlo (QMC) kernel for modeling strongly correlated electron systems. The DCA++ software currently uses three different programming models—message passing interface (MPI), Compute Unified Device Architecture (CUDA), and High Performance ParalleX (HPX)/C++ threading—together with three numerical libraries—BLAS (Basic Linear Algebra Subprograms), (LAPACK) Linear Algebra Package, and MAGMA (Matrix Algebra on GPU)—to expose the parallel computation. In the QMC kernel [36], the two-particle Green’s function ($G_t$) is needed for computing important fundamental quantities, such as the critical temperature ($T_c$), for superconductivity. In other words, a larger $G_t$ allows condensed matter scientists to explore larger and more complex (i.e., higher fidelity) physics cases. DCA++ currently stores $G_t$ in a single GPU device. However, this limits the largest $G_t$ that can be processed within one GPU. A new approach for partitioning the large $G_t$ across the multiple GPUs can significantly increase scientists’ capabilities to explore higher fidelity simulations. This chapter focuses on how the memory-bound issue in DCA++ was successfully addressed by proposing an effective “all-to-all” communication method—a ring algorithm—to update the distributed $G_t$ device array circularly.

3.2. Background

QMC solver applications are widely used and are mission-critical across the US Department of Energy’s (DOE’s) application landscape. For the purpose of this chapter, the authors chose to use one of the major QMC applications, the Dynamical Cluster Approximation (DCA++) code. A production-scale scientific problem runs on DOE’s fastest supercomputer, Summit, at the Oak Ridge Leadership Computing Facility on all 4,600 nodes—with each node containing six NVIDIA Volta V100 GPUs—attaining a peak performance of 73.5 PFLOPS with a mixed precision implementation [36].

Monte Carlo simulations are embarrassingly parallel, which the authors exploited on distributed systems with a two-level (MPI + threading) parallelization scheme (Figure 3.1). Although DCA++ has been highly optimized and is scalable on existing hardware, this is the first effort to focus on solving the memory-bound issue described in Section 5 and further taking advantage of GPU RDMA capability on Summit.

Figure 3.1. Workflow of the QMC DCA++ solver.

Figure 3.1 shows the parallelism hierarchy in one iteration of the QMC solver (MPI distribution + on-node threading parallelism). For example, each rank \( \{R0, \ldots, RN\} \) is
assigned a Markov Chain and the initial input (two particle Green’s function, $G_{t,i}$, where $t$ means “two-particle,” and $i$ is rank index). Each rank spawns multiple independent worker threads (i.e., walkers and accumulators). Most work/computation is performed on the GPU. Each walker thread generates a measurement result ($G_{\sigma,i}$ array, where $i$ is thread ID) by performing nonuniform Fourier transform implemented by matrix-matrix multiplication. Each walker passes its $G_{\sigma,i}$ to its corresponding accumulator thread. In other words, each thread has its own $G_{\sigma,i}$ array, and each rank will have $k$ different $G_{\sigma,i}$ arrays, where $k$ is the number of walker threads per rank. Each accumulator thread then updates $G_{t,i}$ via the formula in Eq. (3.1) to compute and update rank-local $G_{t,i}$ to $G'_{t,i}$. The updated partial $G'_{t,i}$ is then fed into the coarse-graining step for the next measurement. At the end of all measurements, an MPI_Reduce operation will be performed on $G_t$ across all ranks to produce a final and complete $G_t$ in the root rank. $G_t$ is allocated before all measurements start and has a life spanning until the end of the DCA++ program.

3.2.1. Memory-bound issue in DCA++

The results from Balduzzi et al. [36] show that although storing a two-particle Green’s function ($G_t$) on the accelerator device allows condensed matter scientists to explore larger and more complex (i.e., higher fidelity) physics cases, the problem size is limited to the device memory size. Updating the device array $G_t$ is the most time-consuming and memory-intensive process throughout DCA++ computation. A distributed $G_t$ approach is needed to reduce memory allocation and operation in the device.

In the original DCA++ algorithm, $G_t$, is updated by a multiplication between two smaller matrices (single-particle Green’s function, or $G_{\sigma}$). This computation update is in the particle-particle channel and is accumulated according to Eq. (3.1).

$$G_t(K_1, K_2, K_3) += \sum_{\sigma} G_{\sigma}(K_3 - K_2, K_3 - K_1) G_{-\sigma}(K_2, K_1), \quad (3.1)$$

where $K_i$ is a combined index that represents a particular point in the momentum and
frequency space, and $\sigma = +1$ or $-1$ specifies the electron spin value. $G_{\sigma}$ is the single-particle Green’s function that describes the configuration of single electrons.

The ability to handle a larger $G_t$ allows the simulation of complex materials to significantly increase the details, accuracy, and fidelity. In the previous design that kept $G_t$ within one GPU, only a sub-slice of $G_t$ could be computed in a single computation. For the simple single-orbital coarse-grained Hubbard model, physics insights or prior knowledge can be used to decide which sub-slices in $G_t$ contain the important physics and thus avoid the generation of full $G_t$. This simple model allows the generic behavior that comes from electronic corrections in materials to be studied, but it cannot distinguish between different specific materials. Material-specific modeling requires more complex models that include more orbital—and other—degrees of freedom, and this requires a much larger $G_t$. This new distributed ring implementation enables the full large $G_t$ array to be computed in a single computation, even for the more complex multi-orbital models, to ensure that no important physics are overlooked.

3.3. Ring-based Communication Algorithms Implementation

3.3.1. GPU RDMA Technology

GPU RDMA allows direct peer access to multi-GPU memory through a high-speed network. For NVIDIA GPUs, GPUDirect is a technology that allows for the direct transfer of data in GPU device memory to other GPUs on the same node by using the NVLINK2 interconnect and/or between GPUs on different nodes by using RDMA support that can bypass buffers on host memory.

A CUDA-aware MPI\footnote{https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/} implementation can directly pass the GPU buffer pointer to MPI calls. Acceleration support, such as GPUDirect, can be used by the MPI library and allows buffers being sent from the kernel memory to a network without staging through host memory. Various acceleration supports are commercially available, and there are open-sourced CUDA-aware MPI implementations, such as OpenMPI, MVAPICH2, and IBM
3.3.2. Distributed $G_t$ in QMC solver

Before introducing the communication phase of the ring abstraction layer, it is important to understand how we distribute the large device array $G_t$ (two-particle Green’s function) across MPI ranks. We compare original $G_t$ and distributed $G^d_t$ versions (see Figure 3.2).

In the original $G_t$ implementation, the measurements (computed by matrix-matrix multiplication) are distributed statically and independently over the MPI ranks to avoid inter-node communications. Each MPI rank keeps its partial copy of $G_{t,i}$ to accumulate measurements within a rank, where $i$ is the rank index. After all the measurements are finished, a reduction step is taken to accumulate $G_{t,i}$ across all MPI ranks into a final and complete $G_t$ in the root MPI rank. The size of $G_{t,i}$ in each rank is the same size of final and complete $G_t$.

With the distributed $G^d_t$ implementation, we evenly partition this large device array $G_t$ across all MPI ranks and each portion of it is local to each MPI rank. Each rank instead of keeping its partial copy of $G_t$, now keeps an instance of $G^d_{t,i}$ to accumulate measurements of a portion or subslice of the final and complete $G_t$, where the notation $d$ in $G^d_t$ refers to the distributed version and $i$ means the $i$-th rank. Note that the size of $G^d_{t,i}$ in each rank is reduced to $1/p$ of size of final and complete $G_t$ comparing the same configuration in original $G_t$ implementation, where $p$ is the number of MPI ranks used. For example, in Figure 3.2b, there are 4 ranks, and rank $i$ now only keeps $G^d_{t,i}$, which is $1/4$ the size of the original $G_t$ array size.

For the distributed $G^d_t$ implementation, to compute the final and complete $G^d_{t,i}$, each rank needs to see every $G_{\sigma,i}$ from all ranks. In other words, each rank needs to broadcast the locally generated $G_{\sigma,i}$ to the rest of the other ranks at every measurement step. To efficiently perform this “all-to-all” broadcast, we therefore build a ring abstraction layer (Section. 3.3.3), which circulates all $G_{\sigma,i}$ across all ranks.
Figure 3.2. Compare original $G_t$ v.s. distributed $G_{t_d}$ implementation. Each rank contains one GPU resource.
3.3.3. Pipeline Ring Algorithm

We implement a pipeline ring algorithm that broadcasts \((G_\sigma)\) array in a circular fashion during every measurement. The algorithm (Algorithm. 1) is also visualized in Figure 3.3.

**Algorithm 1: Pipeline Ring Algorithm**

```
1 generateGSigma(gSigmaBuf);
2 updateG4(gSigmaBuf);
3 i ← 0;
4 myRank ← worldRank;
5 ringSize ← mpiWorldSize;
6 leftRank ← (myRank − 1 + ringSize) % ringSize;
7 rightRank ← (myRank + 1 + ringSize) % ringSize;
8 sendBuf.swap(gSigmaBuf);
9 while i < ringSize do
10    MPI_Irecv(recvBuf, source=leftRank, tag = recvTag, recvRequest);
11    MPI_Isend(sendBuf, source=rightRank, tag = sendTag, sendRequest);
12    MPI_Wait(recvRequest);
13    updateG4(recvBuf);
14    MPI_Wait(sendRequest);
15    sendBuf.swap(recvBuf);
16    i++;
17 end
```

At the start of every new measurement, a single-particle Green’s function \(G_\sigma\) (Line 1) is generated and then is used to update \(G_{t_i}^e\) (Line 2) using formula in Equation (3.1).
Between Lines 3 to 8, the algorithm initializes the indices of left and right neighbors, and prepares the sending message buffer from previously generated $G_\sigma$ buffer. The processes are organized as a ring so the first rank and last rank are considered as neighbors to each other. A swap operation is used to avoid unnecessary memory copies for $sendBuf$ preparation. Note that walker-accumulator thread allocates an additional $recvBuf$ buffer of the same size as $gSigmaBuf$ to hold incoming $gSigmaBuf$ buffer from leftRank.

The while loop is the core part of the pipeline ring algorithm. For every iteration, each thread in a rank is receiving a $G_\sigma$ buffer from left neighbor rank, and sending a $G_\sigma$ buffer to right neighbor rank. A synchronization step (Line 12) is followed after to ensure each rank receives a new buffer to update the local $G_{t,i}^d$ (Line 13). Another synchronization step is followed to ensure the send all send requests are finalized (Line 14). Lastly, another swap operation is used to exchange content pointers between $sendBuf$ and $recvBuf$ to avoid unnecessary memory copy and prepare for next iteration of communication. Note that in the multi-threaded version (Section 3.3.5), thread of index $i$ only communicates with threads of index $i$ in neighbor ranks and each thread allocates two buffers ($sendBuff$ and $recvBuff$).

The while loop will be terminated after $(ringSize - 1)$ steps. By that time, each locally generated $G_{\sigma,i}$ has traveled across all MPI ranks and updated $G_{t,i}^d$ in all ranks. Eventually, each $G_{\sigma,i}$ has reached to the left neighbor of its "birth rank". For example, $G_{\sigma,0}$ generated from rank 0 will end in last rank in the ring communicator.

Additionally, if the $G_t$ is too large to be stored in one node, it is optional to accumulate all $G_{t,i}^d$ at the end of all measurements. Instead, a parallel write into file system could be taken.

### Subring Optimization

A subring optimization strategy is further proposed to reduce message communication times if the large device array $G_t$ can fit in fewer devices. We visualize the subring algorithm in Figure 3.4.
For the ring algorithm (Section 3.3.3), the size of the ring communicator (mpiWorldSize) is set to the same size of the global MPI_COMM_WORLD, and thus the size of $G_t$ is equally distributed across all MPI ranks.

However, to complete the update $G_{t,i}^d$ in one measurement, one $G_{\sigma,i}$ has to travel mpiWorldSize ranks. In total, there are mpiWorldSize numbers of $G_{\sigma,i}$ being sent and received concurrently in one measurement in the global MPI_COMM_WORLD communicator. This will cause high communication overhead, if the size of $G_{t,i}^d$ is relatively small per rank.

If $G_t$ can be distributed and fitted in fewer devices, then the shorter the travel distance is required for $G_{\sigma,i}$, thus reducing the communication overhead. We then perform one reduction step at the end of all measurements to accumulate $G_{t,s,i}^d$, where $s_i$ means $i$-th rank on the $s$-th subring.

At the beginning of MPI initialization, we partition the global MPI_COMM_WORLD into several new subring communicators using MPI_Comm_split. We pass the new communicator information to DCA++ concurrency class by substituting the original global MPI_COMM_WORLD with this new communicator. Now, only a few minor modifications are needed to transform the Ring Algorithm (Algorithm 1) to subring Algorithm 2. In Line 4, we initialize myRank to subRingRank instead of worldRank, where subRingRank is the rank index in the local subring communicator. In Line 5, we initialize ringSize to subRingSize instead of mpiWorldSize, where subRingSize is the size of new communicator. Note, the general ring algorithm is a special case for subring algorithm because subRingSize of general ring algorithm is equal to mpiWorldSize and there is only one subring group throughout the all MPI ranks.

**Algorithm 2:** Modified ring algorithm to support subring communication

| myRank ← subRingRank;      |
| ringSize ← subRingSize;    |
Figure 3.4. Workflow of subring algorithm per iteration. Every consecutive $S$ ranks form a subring communicator, and no communication happen between subring communicators until all measurements are finished. Here $S$ is the number of ranks in a subring.

3.3.5. Multi-threaded Ring Communication.

To take advantage of multi-threaded QMC model already in DCA++, we further implement multi-threaded ring communication support in the ring algorithm. Figure 3.1 shows that in original DCA++ method, each walker-accumulator thread in a rank is independent to each other, and all the threads in a rank will synchronize only after all rank-local measurements are finished. Moreover, during every measurement, each walker-accumulator thread will generate its own thread-private $G_{\sigma,i}$ to update $G_t$.

The multi-threaded ring algorithm now allows concurrent message exchange, such that threads of same rank-local thread index exchange their thread-private $G_{\sigma,i}$. Conceptually, there are $k$ parallel and independent rings, where $k$ is number of threads per rank, because threads of the same local thread id form a closed ring. For example, thread of index 0 in rank 0 will send its $G_{\sigma}$ to the thread of index 0 in rank 1, and receive another $G_{\sigma}$ from thread index of 0 from last rank in the ring algorithm.

The only changes in the ring algorithm are offsetting the tag values (recvTag and sendTag) by the thread index value. For example, Lines 10 and 11 from Algorithm. 1 are modified to Algorithm. 3.
Algorithm 3: Modified ring algorithm to support multi-threaded ring

\begin{verbatim}
MPI_Irecv(recvBuf, source=leftRank, tag = recvTag + threadId, recvRequest);
MPI_Isend(sendBuf, source=rightRank, tag = sendTag + threadId, sendRequest);
\end{verbatim}

Note, in order to efficiently send and receive $G_\sigma$, each thread will allocate one additional recvBuff to hold incoming gSigmaBuf buffer from leftRank and perform send/receive efficiently. In original DCA++ method, there are $k$ numbers of buffers of $G_\sigma$ size per rank, and now in the multi-threaded ring method, there are $2k$ numbers of buffers of $G_\sigma$ size per rank, where $k$ is number of threads per rank.

3.4. Results

In this section, we evaluate our work from various perspectives, including correctness, memory analysis, scaling, function activities, etc. with the help of APEX profiling tool. We run all experiments on Summit supercomputer.

3.4.1. Summit Node Configuration

Summit is a 4600 node, 200 PFLOPS IBM AC922 system. Each node consists of 2 IBM POWER9 CPUs with 512 GB DDR4 RAM and 6 NVIDIA V100 GPUs with total of 96 GB high bandwidth memory. Each Summit node (Figure 3.5) is divided into 2 sockets, where each socket has one IBM POWER9 CPU and 3 NVIDIA V100 GPUs all connected through NVIDIA’s high-speed NVLINK2 (each NVLINK2 capable of a 25GB/s transfer rate in each direction). Two IBM POWER9 CPUs within a Summit node are connected through PCIe bus (64 GB/s bidirectional). Summit nodes are connected through Network Interface Connector (NIC) (12.5 GB/s in each direction).

3.4.2. APEX

APEX [37] (Autonomic Performance Environment for Exascale) is a performance measurement library for distributed, asynchronous multitasking systems. It provides lightweight measurements without perturbing high concurrency through both synchronous and asynchronous interfaces. To support performance measurement in systems that employ OS or
Figure 3.5. Architectural layout of a single node on the Summit supercomputer.

user-level threading, APEX uses a dependency chain in addition to the call stack to produce traces and task dependency graphs. The synchronous APEX instrumentation API can be used to add instrumentation to a given runtime and includes support for both timers and counters. To support C++ threads on Linux systems, the underlying POSIX threads are automatically instrumented using a preloaded shared object library that intercepts and wraps pthread calls in the application. CUDA host callback and device activity measurements are provided through the NVIDIA CUPTI interface [38]. In addition, the hardware and operating system are monitored through an asynchronous measurement that involves periodic or on-demand interrogation of operating system, hardware or runtime states (e.g. CPU utilization, resident set size, memory “high water mark”). Periodic CUDA device monitoring is provided to APEX by the NVIDIA NVML interface [39]. For this work, APEX was extended to capture additional timers and counters related to CUDA device to device (DtoD) memory transfers, and support for key MPI calls was provided by a minimal implementation of the MPI Profiling Interface [40].
3.4.3. Accuracy Analysis

To verify that our implementation generates correct results, we run the same input configuration for original and ring algorithm methods respectively, and compare the difference between the original $G_t$ and accumulated $G^d_t$ arrays. We use a normalized L1 loss function (Least Absolute Deviations, Equation (3.2)) and normalized L2 loss function (Least Square Errors, Equation (3.3)) to compute the normalized error between original $G_t$ and accumulated $G^d_t$ arrays, where we use the "entrywise" norm\(^2\). Our baseline is that the $L1\_error$ and $L2\_error$ between two arrays should be both smaller than 5e-7 following DCA++ testing protocol, where

\[
L1_{\_error} = \frac{\|\text{vec}(G_t - G^d_t)\|_1}{\|\text{vec}(G_t)\|_1},
\]

\[
L2_{\_error} = \frac{\|\text{vec}(G_t - G^d_t)\|_2}{\|\text{vec}(G_t)\|_2}.
\]

For input configuration, we choose 100k Monte Carlo measurements as it is a representative case to our scientific production run. We configure the cluster size to 6*6 and four-point-fermionic-frequencies to 64, which leads to 212336640 entries in $G_t$. Since each $G_t$ entry is a double-precision complex number, so the $G_t$ memory size is about 3.4 GBytes. This configuration can produce large $G_t$ but still will not hit memory-bound issue on Summit GPU (each GPU has 16 GB) for the regular $G_t$ version. We run such configuration on one Summit node (6 ranks per node and 7 walker-accumulator threads per rank) for 5 times. For the distributed $G^d_t$ version, we set ring size to 6 so there is only one subring during the run. Our results show our implementation generate correct results (Table. ??) as $L1\_error$ and $L2\_error$ on accumulated $G^d_t$ is in acceptable range.

3.4.4. Memory Analysis

Our memory analysis results show that device memory required for $G^d_t$ decreases linearly to size of subring or the number of MPI ranks in the sub communicator, which fits

\(^2\)Entrywise norm as defined in https://en.wikipedia.org/wiki/Matrix_norm
Table 3.1. Difference between original $G_t$ and accumulate $G_t^d$ over 5 runs

<table>
<thead>
<tr>
<th>Error</th>
<th>Real part</th>
<th>Imaginary part</th>
<th>$&lt;5e-7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>$3.71e-09$±$1.74e-18$</td>
<td>$4.61e-09$±$2.16e-18$</td>
<td>True</td>
</tr>
<tr>
<td>L2</td>
<td>$3.10e-10$±$4.19e-18$</td>
<td>$3.37e-10$±$3.89e-18$</td>
<td>True</td>
</tr>
</tbody>
</table>

Figure 3.6. cudaMalloc requested size (GBytes) over time visualized by Vampir.

For example, we compare the requested size in cudaMalloc API between original $G_t$ (Figure 3.6a) and distributed $G_t^d$ (subring size of 3, Figure 3.6b) methods. It shows that distributed $G_t^d$ method produced $3 \times$ less memory allocation than the original $G_t$ device array. Around 7s in both cases, the distributed $G_t^d$ method allocated 1.13 GBytes for $G_{t,i}^d$, while original $G_t$ method allocated 3.40 GBytes for $G_{t,i}$.

3.4.5. Scaling Results

In the pipeline subring algorithm, each rank sends $S - 1$ and receives $S - 1$ messages, where $S$ is the size of subring. Thus the total number of messages scales quadratically as $O(S^2)$ but the number of messages crossing each communication link increases linearly as $O(S)$. Figure 3.7 shows the elapsed computation time for 1400 measurements (per rank).
of the subring algorithm running with 6 ranks per Summit node and where each message is about 170 MBytes. The data are well approximated by a linear least-square line that indicates the elapsed computation time increases linearly with the size of the subring. This suggests the subring algorithm is not constrained by the total volume of messages but restricted by the slowest communication link. The effective bandwidth of the subring algorithm can be estimated as

\[
\text{effective bandwidth} \approx \frac{(170 \times 10^6 \times S \times 1400)}{\text{elapsed time}}
\]

and this is about 6 GBytes/sec using the data for \(S = 60\) on 10 nodes in Figure 3.7. This effective bandwidth is about 50% of the theoretical peak bandwidth (12.5 GBytes/sec per socket) for the Network Interface Connector (NIC) on the Summit node. Further performance improvement may be feasible by using a bidirectional subring algorithm to take advantage of the bi-directional capability of the NIC.

3.5. Discussions

3.5.1. Concurrency Overlapping

We observe that our multi-threaded ring implementation provides sufficient concurrency that overlaps communication and computation. We use APEX profiling tool to collect data on process activities over time, and visualize the data in Vampir.

We run DCA++ with multi-threaded ring support, and obtain the timeline activities in rank 0 at 49 seconds (Figure 3.8). We observe that there is some concurrency overlap in the multi-threaded ring algorithm such that while some threads are blocked in MPI\_Wait, other threads of the same rank are performing useful computation tasks. For example, those short blocks that are not labeled as MPI\_Wait are mostly related to kernel activities.

We also observe that current ring algorithm is a lock-step algorithm where next computation (update \(G_t\)) cannot start until the previous communication step (\(G_\sigma\) message exchange) is finished. To expose more currency, we can take advantage of HPX [9], which
Figure 3.7. Time for 1400 iterations (per rank) of the subring algorithm using 6 ranks per node on Summit and each message size is 170 MBytes.

is a task-based programming model, to potentially overlap the communication and computation. For example, we can wrap DCA++ kernel function into a HPX future, which represents an ongoing computation and asynchronous task, then we can attach or chain the communication tasks to the "futurized" kernel task. In [4], authors reported that DCA++ with HPX user-level [41] threading support achieves 20% speedup over the original C++ threading (kernel-level) due to faster context switching in HPX threading.

3.5.2. Trade-off Between Concurrency and Memory

We also observe that as walker-accumulator threads increase in the multi-threaded ring algorithm, GPU memory usage is also increased due to more device memory is needed for storing extra thread-private $G_{\sigma,i}$ buffers. This might cause a new memory-bound challenge if we use too many concurrent threads. A possible solution is to reduce concurrent threads
Figure 3.8. Vampir timeline graph shows the processes activities over the time in rank 0 (DCA++ with multi-threaded ring algorithm).

in order to achieve more usable device memory.

We run the same configuration for original $G_t$ and distributed $G_t^d$ versions with 7 threads and then with 1 thread, respectively (see Figure 3.9).

For the comparison on 7 threads (Figure 3.9a and 3.9b), the first spike in memory usage increase is due to $G_t$ allocation and second significant wave is because each thread is allocating $G_{\sigma,i}$. Although the $G_d$ allocation is 3 times less in distributed $G_t^d$ (1.3 GB) version than original $G_t$ version (3.4 GB), the maximum device usage is about 1.6 GB larger in distributed $G_t^d$ (11.2 GB) version than original $G_t$ version (9.6 GB). We believe that the extra device memory usage (1.6 GB) is largely contributed by additional message buffer needed per thread for holding send/receive $G_{\sigma,i}$. When we run 7 threads for distributed $G_t^d$ method, each thread essentially has 2 $G_{\sigma}$ buffers (each sized at 170 MBytes), one for $sendBuf$ and another one for $recvBuf$ (see Section 3.3.3), then we have total device memory about $14 \times 170$ MBytes = 2.4 GB for maintaining all $G_{\sigma}$’s per rank. On the other hand, for original $G_t$ version with 7 threads run, we only allocate about $7 \times 170$(MBytes) = 1.2 GBytes.

However, if we use only 1 thread (see Figure 3.10a and 3.10b), the maximum device usage in distributed $G_t^d$ version (3.3 GB) is 1.9 GB less than the one in original $G_t$ version (5.2 GB). We can gain much more usable device memory if we reduce concurrent walker-accumulator threads. For example, the saved device memory from reduced threads can be used to fit larger $G_t$. Further, we run a comparison experiment on one Summit node (6 ranks per node) using the same input configuration (subring size is 3, measurements is 4200 in total) except threading numbers per rank. We observe that the distributed $G_t^d$ with
Figure 3.9. Device memory used (GBytes over time) when using 7 walker-accumulator thread. Visualized by Vampir.
Figure 3.10. Device memory used (GBytes over time) when using 1 walker-accumulator thread. Visualized by Vampir.
7 threads (87 seconds) has $1.3 \times$ speedup than the one with 1 thread (116 seconds). This result suggests that if there is insufficient device memory, the code might use fewer threads with some loss (less than 30%) of runtime performance. We are considering to quantify and model such trade-off in our future research development.

To solve the NIC bottleneck issue and the new memory bound challenge caused by multi-threaded communication (storing additional $G_\sigma$), we are considering another plan to move $G_\sigma$ to the CPU host where the CPU host has more memory. Each Summit node contains 512 GB of DDR4 memory for use by the IBM POWER9 processors while there are only $6 \times 16$ GBytes = 96 GBytes of device memory. On Summit, the NICs are connected to the CPU, not directly connected to GPU. The NVLINK2 connection between CPU and GPU has peak of 50 GBytes/s so it is faster compared to the peak bandwidth (12.5 GBytes/s) of the NIC and may not be the bottleneck. One possible future extension may be to consider keeping $G_t$ on the CPU side instead of in GPU device memory so that we can use a smaller subring or keep the subring on the same single node.
Chapter 4. Optimize SIMD Code Using LLVM-based Analysis on Arm Supercomputer

4.1. Introduction

Program analysis tools are important in helping users understand, improve, and port their applications to new platforms. This is crucial for applications that need tuning and significant code restructuring to exploit new types of hardware devices, such as single instruction/multiple data (SIMD) units and accelerators. Compiler-based tools are crucially important for identifying opportunities to improve application codes as the compiler generates code for different architectures. In particular, the LLVM compiler is an open-source compiler that provides a set of tools for the static analysis and feedback of application code. Static program analysis information can be combined with dynamic information (profile-based) to filter the large amount of information produced by the compiler so that users can focus on the most frequently executed regions of their code.

This chapter presents a methodology for using LLVM-based tools to tune an application to generate efficient SIMD instructions that target the new ARM A64FX processor, as well as describes what is required to achieve good performance.

4.2. Case Study: Porting DCA++ to Wombat

This section describes the authors’ experiences in porting the DCA++ (dynamical cluster approximation) application to the Wombat\(^1\) cluster, an ARM-based heterogeneous cluster at Oak Ridge National Laboratory. This section presents a methodology for using LLVM-based tools to tune the DCA++ application targeting the ARM A64FX and ThunderX2 processors. The goal is to describe what changes are required for the new architecture and generate efficient SIMD instructions that target the new Scalable Vector Extension (SVE)

---

\(^1\)Wombat: www.olcf.ornl.gov/olcf-resources/compute-systems/wombat/
instruction set available in the A64FX processors based on LLVM-based tools information.

4.2.1. Evaluation Environment

The case study used the Wombat test bed with 24 compute nodes. Sixteen compute nodes are based on the Fujitsu A64FX processor with SVE and a theoretical peak performance of 3.3792 TFlops. Each A64FX node has one processor socket with 32 GB of second-generation High-Bandwidth Memory (HBM2). The A64FX-equipped nodes do not have additional Double Data Rate (DDR) memory. Eight compute nodes have two ThunderX2 processors with NEON vector instructions and a theoretical peak performance of 560 GFlops. The ThunderX2 nodes have 256 GB of DDR4 RAM and a 480 GB solid-state drive for node-local storage. All nodes are connected with Enhanced Data Rate InfiniBand (100 Gbit/s). The compilers on the system are the ARM 20.3 compilers and the Clang upstream compiler, which is based on Clang 12. The scientific libraries available on Wombat are the ARM Performance Libraries (APL) version 20.3.

4.2.2. DCA++

Quantum Monte Carlo (QMC) solver applications are popular tools essential to the US Department of Energy-supported scientific software. This chapter studies one cutting-edge QMC application called the DCA++ algorithm. DCA++ [43] implements quantum cluster algorithms to solve quantum many-body problems in condensed matter physics. DCA++ is a highly scalable and performant scientific software written in modern C++ and has been ported to various high-performance computing architectures, including IBM Power9, x86_64, ThunderX2, and ARM A64FX [4]. The DCA++ software currently integrates three different programming models—message passing interface (MPI), Compute Unified Device Architecture (CUDA), and High Performance ParalleX (HPX)/C++ threading—together with numerical libraries (e.g., Basic Linear Algebra Subprograms [BLAS], Linear Algebra Package [LAPACK], and MAGMA) to expose the parallel computation structure.

Wei et al. [4] reported that DCA++ with the HPX run time system [9] has produced a 20% run time speedup over the one with C++ standard threading support. The speedup is
primarily due to the faster thread context switching and reduced scheduler synchronization overheads in the HPX run time. Moreover, Autonomic Performance Environment for Exascale (APEX) [37] is an in situ profiling and adaptive tuning framework to the HPX run time system that can capture operating system and hardware system performance data through various interfaces, such as Performance Application Programming Interface (PAPI) [44]. Because APEX is highly integrated into the HPX run time, for HPX-supported applications, users can easily capture PAPI counter information (e.g., level 2 data cache misses, vector/SIMD instructions, floating point instructions) through HPX function annotation. The overhead introduced by APEX profiling is as low as $\sim 1\%$ [45] compared with the overall application run time.

In DCA++, the QMC solver is the most computation-intensive unit that models strongly correlated electron systems [4]. Computation on the QMC solver is parallelized by using a multithreading scheme that comprises walker (i.e., producer) and accumulator (i.e., consumer) tasks. Each task runs on an independent thread. There are multiple walkers running concurrently. Each walker is responsible for a Monte Carlo (MC) update (sampling from the Markov chain), and then an accumulator is popped from the head of the accumulator waiting queue to compute an MC measurement from the walker. When each accumulator finishes its accumulation measurement, it is pushed back to the end of the queue. The walker-accumulator synchronization is managed by the synchronization primitives mutex and conditional_variable.

4.2.3. Baseline Performance

The following experiments compare DCA++’s performance on Wombat by using its A64FX and ThunderX2 nodes. The performance is measured using 48 accumulators and 48 walkers and using 100,000 measurements, which is a representative scientific simulation case in production. On A64FX, DCA++ is built with two different configuration settings: SVE vectorization and SVE-disabled. The SVE vectorization version of DCA++ means that DCA++ is built with SVE compiler flags enabled and vectorized loops, and it uses
Figure 4.1. DCA++ execution time.

the APL optimized for SVE (i.e., LAPACK, BLAS, Fastest Fourier Transform in the West [FFTW]). The SVE compiler flags are set to “-DNDEBUG -fsimdmath -fopenmp -O3 -mcpu=a64fx” The SVE-disabled version means that DCA++ is built with original DCA++ code and open-source scientific libraries, including Netlib-LAPACK and FFTW. Similarly, on ThunderX2, DCA++ is built with two different configurations: with NEON and NEON disabled.

Figure 4.1 shows DCA++ execution time on A64FX and ThunderX2 architectures. On A64FX, the SVE vectorization version of DCA++ performs ~2× faster than the SVE-disabled version. On ThunderX2, the NEON version of DCA++ is observed to be ~1.66× faster than the NEON-disabled version. Noticeably, the SVE vectorization version of DCA++ on A64FX has ~3.3× speedup over the NEON version on ThunderX2. Meanwhile, the NEON version on ThunderX2 is measured to have ~27 GFlops, and the SVE vectorization version of DCA++ on A64FX reached ~78 GFlops (~2.8×).

These results show the performance gains of DCA++ due to the peak performance improvements of the A64FX processor (e.g., 500 GFlops for ThunderX2 vs. 2.5 TFlops for A64FX).

Figure 4.2 shows the breakdown of DCA++ execution time into four categories: application, scientific libraries, HPX run time, and other activities. Each category only considers functions that have more than 1% overhead shown in the final profiling report generated from perf, a Linux built-in performance profiling tool. The application category includes custom modules developed in the DCA++ source code. The HPX run time category represents necessary scheduling and coordination efforts in HPX threads manager. The scientific
libraries category captures routines from external numerical libraries, such as BLAS, LAPACK, FFTW, and math routines. The other activities category summarizes all other functions that have less than 1% overhead in the final profiling report.

Several observations were made from the timing breakdown shown in Fig. 4.2.

1. With SVE vectorization or NEON optimization, the dominant percentage of the overall execution time is shifted from the external scientific libraries to the application source code. For example, on A64FX, the percentage of application time in the SVE-disabled vectorization version of DCA++ is 26%, whereas the percentage of application time in the SVE version is 57%. A similar percentage shift is also observed on ThunderX2 comparisons. In other words, with APL (SVE vectorization on A64FX or NEON optimization on ThunderX2), less time is spent on scientific libraries because APL are particularly optimized on targeting platforms.
<table>
<thead>
<tr>
<th>Accumulator</th>
<th>% total</th>
<th>L2_DCM</th>
<th>VEC_INS</th>
<th>TOT_CYC</th>
<th>FP_INS</th>
</tr>
</thead>
<tbody>
<tr>
<td>no SVE</td>
<td>30.86</td>
<td>9.29E+09</td>
<td>6.05E+11</td>
<td>1.29E+13</td>
<td>2.73E+12</td>
</tr>
<tr>
<td>standard deviation</td>
<td>0.30</td>
<td>4.27E+07</td>
<td>0.00E+00</td>
<td>2.24E+10</td>
<td>0.00E+00</td>
</tr>
<tr>
<td>SVE vectorization</td>
<td>51.11</td>
<td>9.88E+09</td>
<td>6.53E+10</td>
<td>1.09E+13</td>
<td>2.62E+12</td>
</tr>
<tr>
<td>standard deviation</td>
<td>0.17</td>
<td>3.59E+07</td>
<td>0.00E+00</td>
<td>0.00E+00</td>
<td>0.00E+00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Walker</th>
<th>% total</th>
<th>L2_DCM</th>
<th>VEC_INS</th>
<th>TOT_CYC</th>
<th>FP_INS</th>
</tr>
</thead>
<tbody>
<tr>
<td>no SVE</td>
<td>62.15</td>
<td>6.15E+10</td>
<td>3.99E+12</td>
<td>2.61E+13</td>
<td>8.37E+11</td>
</tr>
<tr>
<td>standard deviation</td>
<td>0.61</td>
<td>2.03E+08</td>
<td>0.00E+00</td>
<td>4.70E+10</td>
<td>0.00E+00</td>
</tr>
<tr>
<td>SVE vectorization</td>
<td>40.14</td>
<td>6.27E+10</td>
<td>5.05E+10</td>
<td>8.56E+12</td>
<td>3.45E+11</td>
</tr>
<tr>
<td>standard deviation</td>
<td>0.14</td>
<td>1.11E+08</td>
<td>0.00E+00</td>
<td>8.87E+09</td>
<td>0.00E+00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total (Acc. + Walker)</th>
<th>% total</th>
<th>L2_DCM</th>
<th>VEC_INS</th>
<th>TOT_CYC</th>
<th>FP_INS</th>
</tr>
</thead>
<tbody>
<tr>
<td>no SVE</td>
<td>93.00</td>
<td>7.08E+10</td>
<td>4.60E+12</td>
<td>3.90E+13</td>
<td>3.57E+12</td>
</tr>
<tr>
<td>standard deviation</td>
<td>0.90</td>
<td>2.46E+08</td>
<td>0.00E+00</td>
<td>6.94E+10</td>
<td>0.00E+00</td>
</tr>
<tr>
<td>SVE vectorization</td>
<td>91.25</td>
<td>7.26E+10</td>
<td>1.16E+11</td>
<td>1.95E+13</td>
<td>2.97E+12</td>
</tr>
<tr>
<td>standard deviation</td>
<td>0.31</td>
<td>1.46E+08</td>
<td>0.00E+00</td>
<td>8.87E+09</td>
<td>0.00E+00</td>
</tr>
</tbody>
</table>

Figure 4.3. PAPI counter for DCA++ runs on A64FX.

2. The HPX run time library imposes minimal overhead to the overall program execution. The overhead is primarily due to a lack of sufficient parallelism from the application so that some HPX worker threads in the kernel level are spinning and waiting for user-level tasks.

Further investigation using hardware performance counters is shown in Fig. 4.3. Here, `hpx::annotated_function()` is used to wrap `accumulator` and `walker` tasks so that their activities (i.e., timing information and PAPI counters) can be distinguished in the final profiling report generated from the HPX-APEX profiling tool. Figure 4.3 shows that the total execution time of `accumulator` and `walker` takes the majority of the overall program execution time (~93.00% in the SVE-disabled version and ~91.25% in SVE vectorization version). Several observations were made from Fig. 4.3.

1. The SVE-disabled version of DCA++ on A64FX has nearly ~40× higher `VEC_INC`, 2× higher `TOT_CYC`, and 1.2× higher `FP_INC` than the SVE vectorization version, where `VEC_INC` is vector/SIMD instructions, `TOT_CYC` is total cycles, and `FP_INC` is floating point instructions. The authors noticed that by using the optimized libraries, the application uses less vector and floating point SVE instructions.
Because SVE has wider 512 bit width, fewer vector instructions are needed in the computation than NEON, which has 128 bit width. Also, the SVE has a more powerful instruction set that uses fewer instructions for the same operation.

2. The L2_DCM (L2 data cache misses) does not change with the SVE optimized version because the SVE optimization does not impact overall memory access patterns. Access to HBM2 remained constant in both versions.

3. Using SVE vectorization on DCA++ shifts timing percentages between accumulator and walker in overall program execution. To perform efficient matrix-related operations, the implementation of walker extensively uses DGEMM routines, which are provided by the scientific libraries. The timing percentage of walker is 62.15% with the SVE-disabled version of DCA++ in overall program execution and is reduced to 40.14% with the SVE vectorization version. The percentage reduction of walker is similar to the percentage reduction of scientific libraries observed in Fig. 4.2.

The results show that to further improve the DCA++ application, the focus must be on tuning the application source code, particularly the accumulator code, to determine which loops need further optimization and which were successfully vectorized by the compiler. This requires significant interaction with the LLVM tools to understand the application hot spots and the opportunities for SVE optimizations.

4.3. An LLVM Tool Methodology to Generate Efficient Vectorization

A64FX performance is highly dependent on how well the source can be mapped to SVE instructions. It is important to determine which application loops are not being vectorized and their impact on the application’s overall performance. The ARM C/C++ compiler is based on the LLVM/Clang compiler, which is also the basis for the authors’ exploration and automation toward vectorizing the most important loops in an application.

Like most modern compilers, LLVM/Clang and its derivatives support profile guided optimization (PGO). The idea is that the compiler inserts profiling instructions into the
target binary to collect information when the application is run. During application shutdown, profiling information is stored on the disk for later use. When the application is recompiled in the future, the collected profiling information is used to drive heuristics (e.g., to determine a suitable unroll count for loops). Such profiling also allows the compiler to approximate how much time was spent in a certain portion of code, also referred to as *code hotness*. The latter makes PGO especially interesting to filter optimization remarks because it allows users to only view remarks emitted for hot code regions. Thus, with PGO, users can be guided toward the loops that would benefit the most from vectorization and avoid overloading them with a plethora of uninteresting remarks.

The authors manually analyzed several loops in the DCA++ application by using the aforementioned method described to determine what was hindering loop vectorization. Some loops required a simple change in vectorization flags, and others required user intervention (e.g., vectorization directives, such as OpenMP SIMD) to assist the compiler. The authors also identified loops that required transformations to make the vectorization more efficient. The following sections present a brief discussion for four hot loops that the compiler was unable to vectorize without user intervention.

### 4.3.1. OpenMP SIMD

When optimizing any loops, the compiler’s vectorization pass must preserve the semantics of the original source code. This usually requires static analyses to verify that the transformation is legal. However, it is not uncommon for a transformation to be correct but unable to be statically verified by the compiler. Since OpenMP 4.0, OpenMP has added support for the SIMD directive, which provides a cross-platform method for statically asserting information about the program’s semantics to the compiler’s vectorization pass [46]. In DCA++, various loops require additional information to be successfully vectorized.

Figure 4.4 shows a classical reduction loop. Because \( x_{\text{val}} \) is a floating point value, any reordering of the iterations (e.g., as part of vectorization) would break strict Institute of Electrical and Electronics Engineers (IEEE) floating point compliance and might introduce
errors in the result. By default, LLVM/Clang will not vectorize the loop but will instead
emit a remark (lower part) that explains how \textit{ffast-math} or vectorization pragmas can be
used to overwrite the IEEE floating point semantics. The Clang pragmas are a less feature-
rich variant of the cross-platform OpenMP SIMD directives, but both explicitly tell the
compiler to allow vector execution for a loop. In the OpenMP variant, users should make
the parallel reduction explicit. Additionally, the authors used the aligned clause to pass
alignment information to the compiler, which can lead to improved performance due to
specialized memory instructions.

```c
#pragma omp simd reduction(-:x_val) aligned(x_val, G_ptr : 64)
for (int i = 0; i < j; i++)
    x_val -= x_ptr[i] * G_ptr[i];
```

Figure 4.4. A loop performing a parallel reduction that is not vectorized automatically.

In line 6 of Figure 4.5, there is a noncontinuous memory load—a gather. ARM’s SVE
supports fast gathering operations; however, the compiler cannot vectorize this loop without
manual intervention because the accessed arrays \(M_{ij}, M, \text{ config}_\text{left}, \) and \(\text{ config}_\text{right}\)
might alias and hence overlap. In these situations, the compiler is often able to version the
loop and generate a vectorized variant guarded by a run time alias check to verify that the
accessed ranges of the arrays do not overlap at run time. However, the support for such
run time alias checks in LLVM/Clang is limited to the case in which the accessed bounds
are known statically [47]. Because the index into the \(M\) array is based on the values loaded
from the configuration arrays, the access range cannot be bound statically. The compiler
remark shown below the loop nest summarizes this discussion in a way that is difficult or
impossible for application developers to understand. Using OpenMP SIMD effectively tells
the compiler that there are no overlapping accesses, allowing the loop to be vectorized.
Care must be taken to ensure that no aliasing actually occurs, otherwise this will result in
incorrect results.
for (int j = start_index_right_[orb_j]; j < end_index_right_[orb_j]; ++j) {
    const int out_j = j - start_index_right_[orb_j];
    #pragma omp simd
    for (int i = start_index_left_[orb_i]; i < end_index_left_[orb_i]; ++i) {
        const int out_i = i - start_index_left_[orb_i];
        M_ij_(out_i, out_j) = M(config_left_[i].idx, config_right_[j].idx);
    }
}

remark: loop not vectorized: Unknown array bounds

Figure 4.5. A loop performing a memory gather that requires OpenMP SIMD to be vectorized by the ARM compiler.

4.3.2. Using the Correct Compiler Flags

Some loops require additional compiler flags to be vectorized. The code shown in Figure 4.6 has two run time calls, line 5 and 6, which prevent the compiler from automatically vectorizing it. A function call usually requires an explicit vector version of the function and compiler support to allow vectorized execution. The ARM compiler provides an optimized math library that includes vector variants of common math functions. Users must explicitly enable such a vector library because it will disturb the precision of the result, similar to the floating point reordering. The ARM compiler provides the \texttt{fsimdmath} option to use its performance libraries, whereas standard Clang requires \texttt{fveclib} to be set to the desired vectorized library. \texttt{ffast-math} or \texttt{fno-math-errno} will allow the compiler to execute the loop out of order, but no vectorized math library is used. This means that the vector lanes are effectively unpacked before the call, and the math function is executed once per vector lane.

Another issue is that the application uses a custom matrix class that performs bounds checking by using assertions in the overloaded access operators. Although assertions are a good software engineering practice, their “complex” semantics must be preserved by the compiler. The problem is that no code is executed after a violated assertion. Thus, if assertions are enabled and present in a loop, the compiler must verify that the assertion cannot trigger to execute any side effects succeeding the assertion (e.g., from the next iteration).
To disable assertions completely, `NDEBUG` can be defined during compilation; however this will cause a tension between “debug” and “release” builds that is often not desirable. For developers to identify issues that stem from assertions and other errors in handling code, the authors added a new remark to the LLVM vectorizer, which is shown below the code. For these experiments, the authors disabled assertions, provided a vectorized math library, and added OpenMP SIMD to allow vectorization, even in the presence of possibly aliasing accesses.

```cpp
for (int j = 0; j < n_v; ++j) {
    #pragma omp simd
    for (int i = 0; i < n_w; ++i) {
        const ScalarType x = configuration[j].get_tau() ∗ w_[i];
        T_[0](i, j) = std::cos(x);
        T_[1](i, j) = std::sin(x);
    }
}
```

Figure 4.6. A code block using the math library functions `cos` and `sin`.

4.3.3. Loop Transformations

The loop in Fig. 4.7 contains gathers from memory at lines 11 and 18. More importantly, the code uses a column-major layout for all its matrices while this loop iterates across a row. This will require expensive scattering operations to distribute the stores to discontinuous memory addresses. This loop can be transformed to better exploit SIMD parallelism. Each iteration of this loop is independent, and the matrices are guaranteed to be square in the code, so this loop can safely be transposed to improve memory accesses. This transformation will also improve performance without vectorizing the loop.

This loop contains conditional expressions that must be transformed into masks to be vectorized. This requires calculating the result of each branch and conditionally moving it into the final register by using a mask. In this case, the true condition of the loop at line 6 is much more computationally expensive than the false condition. If the result was not
for (int i = 0; i < Gamma.Rows(); i++) {
    for (int j = 0; j < Gamma.Cols(); j++) {
        int spin_idx_i = random_vertex_vector[i];
        int spin_idx_j = random_vertex_vector[j];

        if (spin_idx_j < vertex_index) {
            Real delta = (spin_idx_i == spin_idx_j) ? 1. : 0.;
            Real N_ij = N(spin_idx_i, spin_idx_j);
            Gamma(i, j) = (N_ij * exp_V[j] - delta) / (exp_V[j] - 1.);
        } else {
            Gamma(i, j) = G_precomputed(spin_idx_i, spin_idx_j - vertex_index);
        }
    }
}

remark: loop not vectorized: control flow cannot be substituted for a select
remark: loop not vectorized: cannot identify array bounds

Figure 4.7. A loop requiring a source transformation and OpenMP SIMD (left) and its transformed version (right).

needed, then this will be calculated at each iteration of the loop, only to be thrown away.
This problem is even worse for the final update across the diagonal at line 17, which will
only be needed once every iteration of the inner loop but calculated every iteration. This
conditional update can be hoisted from the loop to improve performance significantly.

Another issue is the division at line 14. This could cause a division-by-zero error that
can block vectorization if regular error handling semantics are maintained. This can be
disabled with fast math, but in some cases, the compiler is able to vectorize it by using
masked division instructions. This would be a good application of the assume directive
added in OpenMP 5.1 to assert to the compiler that the division will never cause an error.

4.3.4. Results

The overall impact of these transformations is shown in Fig. 4.8, which shows a signif-
icant speedup in most cases. The loop in Fig. 4.6 had the largest improvement when using
ARM’s vector math support. The reduction loop in Fig. 4.4 yielded no improvement. Upon
further investigation, this was because the loop’s trip count was very small in the average
Figure 4.8. The loops in Figs. 4.6, 4.5, 4.7, and 4.4, respectively, before and after the barriers to SVE execution were remedied. Performance is measured as the total time spent by all the threads in a run using 24 accumulators/walker threads over 100,000 measurements.

case, so the majority of the time was spent doing the final reduction, and work was rarely done in parallel. The other loops saw reasonable improvements, but their performance was limited by the gathering instructions required to vectorize them.

4.4. Automating the Process: The OpenMP Advisor

It is unrealistic but unfortunately still common practice to optimize code and add support for new platforms and features by manually inspecting and modifying the application. Given the increasing complexity when it comes to hardware and the requirement to support multiple heterogeneous platforms simultaneously, the authors must rethink their software engineering practices to ensure that the code is not only correct but also performant and portable. To automate this manual process and boost programmers’ productivity, the authors began developing the OpenMP Advisor. Based on the portable OpenMP directive language, we hope to evolve the OpenMP Advisor over time into a valuable software engineering tool by using and extending LLVM capabilities. During the porting effort of the DCA++ application described here, the authors experienced various issues that require interpretation to derive actionable advice. Using their experience, the authors began automating the parts of the process and improving the compiler remarks that were missing or misleading. As a result, the OpenMP Advisor the authors develop as part of the LLVM
compiler framework will use optimization remarks from multiple optimization passes to report the most performance-critical problems in the code based on the available profiling data.

4.5. Related Work

There are several other tools that analyze source code or provide support for parallelization but with limited support that automatically inserts SIMD directives in the code. These include: CAPO [48] for automatic OpenMP work-sharing directives generation, which supports Fortran 77 and some F90 extensions; Appentra’s Parallware [49], which focuses on parallelizing C/C++ applications by using OpenMP and OpenACC for multicores and accelerators; and Cray Reveal [50], which helps autoscope OpenMP variables and generate OpenMP work-sharing for Fortran and C/C++ for multicore and accelerators. Intel Inspector focuses on OpenMP semantic checking for data race detection. Foresys [51] and the Dragon Analysis tool [52] are legacy tools that supported the maintenance of Fortran code and help with parallelization with OpenMP.

4.6. Conclusions

Porting the DCA++ application to the A64FX processor requires the use of optimized scientific libraries and vectorizing the application hot spots. This process can be overwhelming to users, and tools are needed to automate this process. This work shows that by using LLVM tools, users can easily detect hot spots, determine why loops are not vectorized, and correct the issues by applying the correct compiler flags, transforming the code, or applying OpenMP directives.

Currently, authors are working an OpenMP Advisor tool that is built on top of existing and newly introduced LLVM tooling to automate this process. Ultimately, the authors want to enable application developers to navigate and handle compiler-generated information productively. Optimization reports should pinpoint important opportunities to tune the code (e.g., non-vectorized loops) and simultaneously provide sufficient information and suggestions to allow informed decisions without elaborate studies of compiler and program-
ming language theory. The authors believe that tools can recommend portable annotations, such as OpenMP SIMD directives, when they inform users about the requirements for correctness. Furthermore, compiler analysis and optimizations can directly target the recently proposed OpenMP assume directive to request user feedback. In other words, OpenMP assume directives and the authors’ implementation in the LLVM compiler will enable analyses and transformations to request high-level information from users naturally. The OpenMP Advisor will improve communication in the other direction to present users with important requests and remarks, together with information and examples that translate “compiler language” to “application language.”
Chapter 5. Conclusions

This dissertation presented research on the performance optimization of parallel and concurrent applications based on the asynchronous many-task runtime system HPX. We used DCA++ (Dynamical Cluster Approximation), a real-world and highly scalable HPX application, in our study. Chapter 2 provided a study of adding HPX threading backend in DCA++ by constructing a threading abstraction layer and analyzed performance across various architectures. Chapter 3 investigated the design and implementation of ring-based GPU RDMA algorithms for solving memory-bound challenges of QMC solver. Chapter 4 studied porting and evaluating performance of HPX-backed QMC application in Arm A64fx using LLVM-based tools.

5.1. Contributions

This dissertation presented the following contributions:

1. Ported DCA++ to various HPC architectures (POWER9, x86_64, Arm A64fx).
2. Implemented the HPX threading model for on-node parallelization in DCA++.
3. Profiled DCA++ using performance measurement library APEX, integrated with HPX.
4. Collaborated with APEX performance observation tool team members, providing feedback and driving research
5. Worked with DCA++ domain science application developers driving their new complex science problems with enhanced optimizations.
6. The memory consumption in a QMC solver application was reduced to store a much larger kernel array across multi-GPUs. This significant contribution enables physicists to evaluate larger scientific problem sizes and compute the full kernel array in a single computation, which significantly increases the accuracy/fidelity of the simulation of a certain material.
7. A ring abstraction layer was designed that updates the large distributed kernel array. The ring algorithm was further improved by adding sub-ring communicator and multi-threaded communication to reduce communication overhead and expose more concurrency, respectively.

8. The ring abstraction layer was implemented on top of NVIDIA GPUDirect remote direct memory access (RDMA) for fast device memory transfer.

9. The Autonomic Performance Environment for Exascale (APEX) performance measurement library was extended to support the use case, driving tool development and research.

10. Presented a methodology for using LLVM-based tools to tune an application to generate efficient SIMD instructions that target the new ARM A64FX processor, as well as describes what is required to achieve good performance.
Appendix A. Copyright Information for Chapter 2
Performance Analysis of a Quantum Monte Carlo Application on Multiple Hardware Architectures Using the HPX Runtime

Conference Proceedings:
2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)

Author: Weile Wei
Publisher: IEEE
Date: Nov. 2020

Copyright © 2020, IEEE

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis online.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.
Appendix B. Copyright Information for Chapter 3
Memory Reduction using a Ring Abstraction over GPU RDMA for Distributed Quantum Monte Carlo Solver

Weile Wei
wwei9@lsu.edu
Louisiana State University
Baton Rouge, LA

Eduardo D’Azevedo
dazevedoef@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, TN

Kevin Huck
khuck@cs.uoregon.edu
University of Oregon
Eugene, OR

Arghya Chatterjee∗
ronnie@lbl.gov
NERSC, Berkeley Lab
Berkeley, CA

Oscar Hernandez
oscar@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, TN

Hartmut Kaiser
hkaiser@cti.lsu.edu
Louisiana State University
Baton Rouge, LA

ABSTRACT

Scientific applications that run on leadership computing facilities often face the challenge of being unable to fit leading science cases onto accelerator devices due to memory constraints (memory-bound applications). In this work, the authors studied one such US Department of Energy mission-critical condensed matter physics application, Dynamical Cluster Approximation (DCA++), and this paper discusses how device memory-bound challenges were successfully reduced by proposing an effective “all-to-all” communication method—a ring communication algorithm. This implementation takes advantage of acceleration on GPUs and remote direct memory access (RDMA) for fast data exchange between GPUs. Additionally, the ring algorithm was optimized with sub-ring communicators and multi-threaded support to further reduce communication overhead and expose more concurrency, respectively. The computation and communication were also analyzed by using the Autonomic Performance Environment for Exascale (APEX) profiling tool, and this paper further discusses the performance trade-off for the ring algorithm implementation. The memory analysis on the ring algorithm shows that the allocation size for the authors’ most memory-intensive data structure per GPU is now reduced to $1/p$ of the original size, where $p$ is the number of GPUs in the ring communicator. The communication analysis suggests that the distributed Quantum Monte Carlo execution time grows linearly as sub-ring size increases, and the cost of messages passing through the network interface connector could be a limiting factor.

CCS CONCEPTS

• Applied computing → Physics.

∗Arghya Chatterjee contributed to this work mostly during his previous appointment at Oak Ridge National Laboratory, Oak Ridge, TN.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PASC ’21, July 5–9, 2021, Geneva, Switzerland
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8563-3/21/07...
https://doi.org/10.1145/3468267.3470618

KEYWORDS

DCA++, Quantum Monte Carlo, GPU Remote Direct Memory Access, memory-bound issue, exascale machines

ACM Reference Format:


1 INTRODUCTION

Dynamical Cluster Approximation (DCA++) is a high-performance research software application [2, 3, 6, 10] that provides a modern C++ implementation to solve quantum many-body problems. DCA++ implements a quantum cluster method with a Quantum Monte Carlo (QMC) kernel for modeling strongly correlated electron systems. The DCA++ software currently uses three different programming models—message passing interface (MPI), Compute Unified Device Architecture (CUDA), and High Performance ParalleX (HPX)/C++ threading—together with three numerical libraries—Basic Linear Algebra Subprograms (BLAS), Linear Algebra Package (LAPACK), and Matrix Algebra on GPU (MAGMA)—to expose the parallel computation.

In the QMC kernel [1], the two-particle Green’s function ($G_t$) is needed for computing important fundamental quantities, such as the critical temperature ($T_c$), for superconductivity. In other words, a larger $G_t$ allows condensed matter physicists to explore larger and more complex (i.e., higher fidelity) physics cases. DCA++ currently stores $G_t$ in a single GPU device. However, this limits the largest $G_t$ that can be processed within one GPU. A new approach for partitioning the large $G_t$ across the multiple GPUs can significantly increase scientists’ capabilities to explore higher fidelity simulations. This paper focuses on how the memory-bound issue in DCA++ was successfully addressed by proposing an effective “all-to-all” communication method—a ring algorithm—to update the distributed $G_t$ device array.

1.1 Contributions

The primary contributions of this work are outlined as follows.

1. DCA++ is available at https://github.com/CompFUSE/DCA
Appendix C. Copyright Information for Chapter 4
SPRINGER NATURE LICENSE
TERMS AND CONDITIONS

Jun 20, 2022

This Agreement between Louisiana State University -- Weile Wei ("You") and Springer Nature ("Springer Nature") consists of your license details and the terms and conditions provided by Springer Nature and Copyright Clearance Center.

License Number 5333390010811

License date Jun 20, 2022

Licensed Content Publisher Springer Nature

Licensed Content Publication Springer eBook

Licensed Content Title A Case Study of LLVM-Based Analysis for Optimizing SIMD Code Generation

Licensed Content Author Joseph Huber, Weile Wei, Giorgis Georgakoudis et al

Licensed Content Date Jan 1, 2021

Type of Use Thesis/Dissertation

Requestor type academic/university or research institute

Format electronic
Portion full article/chapter

Will you be translating? no

Circulation/distribution 200 - 499

Author of this Springer Nature content yes

Title Optimizing the Performance of Parallel and Concurrent Applications based on Asynchronous Many-task Runtimes

Institution name Louisiana State University

Expected presentation date Jun 2022

Requestor Location

Louisiana State University
Louisiana State University

BATON ROUGE, LA 70820
United States
Attn: Louisiana State University

Total 0.00 USD

Terms and Conditions

Springer Nature Customer Service Centre GmbH

Terms and Conditions

This agreement sets out the terms and conditions of the licence (the Licence) between you and Springer Nature Customer Service Centre GmbH (the Licensor). By clicking
'accept' and completing the transaction for the material (Licensed Material), you also confirm your acceptance of these terms and conditions.

1. Grant of License

1.1. The Licensor grants you a personal, non-exclusive, non-transferable, world-wide licence to reproduce the Licensed Material for the purpose specified in your order only. Licences are granted for the specific use requested in the order and for no other use, subject to the conditions below.

1.2. The Licensor warrants that it has, to the best of its knowledge, the rights to license reuse of the Licensed Material. However, you should ensure that the material you are requesting is original to the Licensor and does not carry the copyright of another entity (as credited in the published version).

1.3. If the credit line on any part of the material you have requested indicates that it was reprinted or adapted with permission from another source, then you should also seek permission from that source to reuse the material.

2. Scope of Licence

2.1. You may only use the Licensed Content in the manner and to the extent permitted by these Ts&Cs and any applicable laws.

2.2. A separate licence may be required for any additional use of the Licensed Material, e.g. where a licence has been purchased for print only use, separate permission must be obtained for electronic re-use. Similarly, a licence is only valid in the language selected and does not apply for editions in other languages unless additional translation rights have been granted separately in the licence. Any content owned by third parties are expressly excluded from the licence.

2.3. Similarly, rights for additional components such as custom editions and derivatives require additional permission and may be subject to an additional fee. Please apply to Journalpermissions@springernature.com/bookpermissions@springernature.com for these rights.

2.4. Where permission has been granted free of charge for material in print, permission may also be granted for any electronic version of that work, provided that the material is incidental to your work as a whole and that the electronic version is essentially equivalent to, or substitutes for, the print version.

2.5. An alternative scope of licence may apply to signatories of the STM Permissions Guidelines, as amended from time to time.

3. Duration of Licence
3. 1. A licence for is valid from the date of purchase ('Licence Date') at the end of the relevant period in the below table:

<table>
<thead>
<tr>
<th>Scope of Licence</th>
<th>Duration of Licence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Post on a website</td>
<td>12 months</td>
</tr>
<tr>
<td>Presentations</td>
<td>12 months</td>
</tr>
<tr>
<td>Books and journals</td>
<td>Lifetime of the edition in the language purchased</td>
</tr>
</tbody>
</table>

4. Acknowledgement

4. 1. The Licensor's permission must be acknowledged next to the Licenced Material in print. In electronic form, this acknowledgement must be visible at the same time as the figures/tables/illustrations or abstract, and must be hyperlinked to the journal/book's homepage. Our required acknowledgement format is in the Appendix below.

5. Restrictions on use

5. 1. Use of the Licensed Material may be permitted for incidental promotional use and minor editing privileges e.g. minor adaptations of single figures, changes of format, colour and/or style where the adaptation is credited as set out in Appendix 1 below. Any other changes including but not limited to, cropping, adapting, omitting material that affect the meaning, intention or moral rights of the author are strictly prohibited.

5. 2. You must not use any Licensed Material as part of any design or trademark.

5. 3. Licensed Material may be used in Open Access Publications (OAP) before publication by Springer Nature, but any Licensed Material must be removed from OAP sites prior to final publication.

6. Ownership of Rights

6. 1. Licensed Material remains the property of either Licensor or the relevant third party and any rights not explicitly granted herein are expressly reserved.

7. Warranty

IN NO EVENT SHALL LICENSOR BE LIABLE TO YOU OR ANY OTHER PARTY OR
ANY OTHER PERSON OR FOR ANY SPECIAL, CONSEQUENTIAL, INCIDENTAL OR INDIRECT DAMAGES, HOWEVER CAUSED, ARISING OUT OF OR IN CONNECTION WITH THE DOWNLOADING, VIEWING OR USE OF THE MATERIALS REGARDLESS OF THE FORM OF ACTION, WHETHER FOR BREACH OF CONTRACT, BREACH OF WARRANTY, TORT, NEGLIGENCE, INFRINGEMENT OR OTHERWISE (INCLUDING, WITHOUT LIMITATION, DAMAGES BASED ON LOSS OF PROFITS, DATA, FILES, USE, BUSINESS OPPORTUNITY OR CLAIMS OF THIRD PARTIES), AND WHETHER OR NOT THE PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THIS LIMITATION SHALL APPLY NOTWITHSTANDING ANY FAILURE OF ESSENTIAL PURPOSE OF ANY LIMITED REMEDY PROVIDED HEREIN.

8. Limitations

8.1. **BOOKS ONLY**: Where 'reuse in a dissertation/thesis' has been selected the following terms apply: Print rights of the final author's accepted manuscript (for clarity, NOT the published version) for up to 100 copies, electronic rights for use only on a personal website or institutional repository as defined by the Sherpa guideline ([www.sherpa.ac.uk/romeo/](www.sherpa.ac.uk/romeo/)).

8.2. For content reuse requests that qualify for permission under the [STM Permissions Guidelines](https://www.sherpa.ac.uk/romeo/), which may be updated from time to time, the STM Permissions Guidelines supersede the terms and conditions contained in this licence.

9. Termination and Cancellation

9.1. Licences will expire after the period shown in Clause 3 (above).

9.2. Licensee reserves the right to terminate the Licence in the event that payment is not received in full or if there has been a breach of this agreement by you.

Appendix 1 — Acknowledgements:

For Journal Content:
Reprinted by permission from [the Licensor]: [Journal Publisher] (e.g. Nature/Springer/Palgrave) [JOURNAL NAME] [REFERENCE CITATION (Article name, Author(s) Name), [COPYRIGHT] (year of publication)

For Advance Online Publication papers:
Reprinted by permission from [the Licensor]: [Journal Publisher] (e.g. Nature/Springer/Palgrave) [JOURNAL NAME] [REFERENCE CITATION
(Article name, Author(s) Name), [COPYRIGHT] (year of publication), advance online publication, day month year (doi: 10.1038/sj.[JOURNAL ACRONYM].)

For Adaptations/Translations:
Adapted/Translated by permission from [the Licensor]: [Journal Publisher (e.g. Nature/Springer/Palgrave)] [JOURNAL NAME] [REFERENCE CITATION] (Article name, Author(s) Name), [COPYRIGHT] (year of publication)

Note: For any republication from the British Journal of Cancer, the following credit line style applies:

Reprinted/adapted/translated by permission from [the Licensor]: on behalf of Cancer Research UK: [Journal Publisher (e.g. Nature/Springer/Palgrave)] [JOURNAL NAME] [REFERENCE CITATION] (Article name, Author(s) Name), [COPYRIGHT] (year of publication)

For Advance Online Publication papers:
Reprinted by permission from The [the Licensor]: on behalf of Cancer Research UK: [Journal Publisher (e.g. Nature/Springer/Palgrave)] [JOURNAL NAME] [REFERENCE CITATION] (Article name, Author(s) Name), [COPYRIGHT] (year of publication), advance online publication, day month year (doi: 10.1038/sj. [JOURNAL ACRONYM]).

For Book content:
Reprinted/adapted by permission from [the Licensor]: [Book Publisher (e.g. Palgrave Macmillan, Springer etc)] [Book Title] by [Book author(s)] [COPYRIGHT] (year of publication)

Other Conditions:

Version 1.3

Questions? customercare@copyright.com or +1-855-239-3415 (toll free in the US) or +1-978-646-2777.
References


[47] Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. Runtime pointer


Vita

Weile Wei finished his undergraduate study at Shandong Jianzhu University, Jinan, China and Griffith University, Gold Coast, Australia. He has been with STE||AR (Systems Technology, Emergent Parallelism, and Algorithm Research) group at Center for Computation and Technology at LSU. In 2020, he received master degree in Computer Science at Louisiana State University. During his stay at LSU, he has contributed to several open-sourced projects in the field of distributed machine learning, parallel computing and high performance computing, including HPX, Phylanx, DCA++, and LibCDS. His research interests cover high-performance computing, GPU RDMA, runtime system, machine learning, database, and storage. Also, he interned with National Center for Atmospheric Research, Oak Ridge National Lab, Google Summer of Code, Facebook, ByteDance/TikTok.

The following contents describe publications made during my PhD study at LSU. Parts of this dissertation contains previously published materials from IEEE, ACM, Springer Nature, and the following publications have been incorporated throughout the dissertation. Copyright information and permissions for reuse are detailed in the Appendix.


- Chapter 4 is reused from — Huber, Joseph, Weile Wei, Giorgis Georgakoudis, Johannes Doerfert, and Oscar Hernandez. “A Case Study of LLVM-Based Analysis for

The following publications were published during my Ph.D. study in the Computer Science field but not included in this dissertation.


Additionally, I have published several other papers [53,54], and posters [55–57].