Building Parallel Abstractions to DCA++ Scientific Software by Taking Advantage of HPX and GPUDirect

Weile Wei¹, A. Chatterjee², O. Hernandez², H. Kaiser¹

1. Louisiana State University
2. Oak Ridge National Laboratory

Authors would like to thank:

Giovanni Balduzzi (ETH Zurich)  
John Biddiscombe (CSCS)  
Thomas Maier (ORNL)  
Ed. D'Azevedo (ORNL)  
Ying Wai Li (LANL) [former ORNL]

- **The parallel abstraction optimization** was supported by the **Scientific Discovery through Advanced Computing (SciDAC)** program funded by U.S. DOE, Office of Science, Advanced Scientific Computing Research (ASCR) and Basic Energy Sciences (BES), Division of Materials Science and Engineering.

- **This research used resources of the Oak Ridge Leadership Computing Facility (OLCF) at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.**

CompFUSE: [https://compfuse.ornl.gov](https://compfuse.ornl.gov)
Outline

○ Quantum Monte Carlo solver application: DCA++

○ Threading abstraction using HPX to parallelize computations
  ○ HPX runtime system
  ○ Performance implications of using HPX over C++ Standard threads

○ Optimized distributed computing abstraction on & across Summit nodes
  ○ Using GPUDirect RDMA (NVLink)
  ○ To address memory bound challenges in DCA++

○ Ongoing efforts
  ○ Multi platform support for QMC applications
  ○ Using APEX + HPX Runtime → in depth visualization of kernels
DCA ++ (Dynamical Cluster Approximation)

- Scientific software for solving quantum many-body problems
- A numerical simulation tool to predict behaviors of co-related quantum materials (such as superconductivity, magnetism)
- Ported to world’s largest supercomputers, e.g. Titan, Summit, Cori, Piz Daint (CSCS) sustaining many petaflops of performance.

DCA++: Primary workflow

1. Coarse Graining (Calculating Green’s Function)

2. Quantum Monte Carlo solver (QMC)

Initial Green’s Function & Markov Chain

Coarse-grained Green’s Function [G]

two particle Green’s Function [G4]

Green’s Function [G0]

Iterative Convergence Algorithm
DCA++ : Quantum Monte Carlo Solver

Imagine: 2D space with lots of points on it (measurements)

Walkers → 1. picks these measurements at random
  2. performs computation (mostly DGEMMs)
  3. sends matrices to accumulator (Producer)

Accumulators → 1. Feeds in the matrices from the walkers
  2. Computes \([G_2]\) for next iteration (Consumer)
  3. Also computes \(G_4 \rightarrow [G_2] \ast [G_2]\)

[all computation happens on both GPU and CPU sides]
DCA++ : Threaded QMC solver

MPI communication among different nodes

Threading on node

MPI communication among different nodes

Walker 1

Walker 2

Acc. 1

Acc. 2

Acc. 3

>10⁶ samples

Green's Function [G]

MPI communication among different nodes

Threading on node

MPI communication among different nodes
Threading abstraction w/ HPX runtime system
Threading abstraction for QMC Solver

std::thread

HPX::thread

swtich at compile time via user-input
HPX_DIR=$HPX_PATH
DCA_WITH_HPX=ON

Custom-made Thread pool

Figure: workflow of thread-pool. Adding hpx option does not change API of custom-made thread pool in DCA++ due to HPX is C++ standard compliant
HPX - A General Purpose Runtime System

- Widely portable (Platforms / Operating System)
- Unified and standard-conforming C++ API and more ...
- Explicit support for hardware accelerators and vectorization
- *Boost license* and has an open, active, and thriving developer community
- Domains: Astrophysics, Coastal Modeling, Distributed Machine Learning
- Funded through various agencies:

Yes, we accept Pull Requests !!!

https://github.com/STEllAR-GROUP/hpx
HPX Runtime System

API

C++1y Parallelism APIs

Threading Subsystem

Active Global Address Space (AGAS)

Performance Counter Framework

Local Control Objects (LCOs, Synchronization)

Parcel Transport Layer (Network)

Policy Engine/Policies

OS
HPX - C++ standard compliant and more

- C++ standard library API compatible: (selected)
  - std::thread        hpx::thread
  - std::mutex         hpx::mutex
  - std::future        hpx::future
  - std::async         hpx::async
  - std::function      hpx::function
  -
  - Extend standard APIs where needed (compatibility is preserved)
HPX thread pool

User Space

- HPX thread scheduler

Kernel Space

- CPU
- CPU
- CPU
- CPU

Nanosecond level

HPX thread is a lightweight user-level thread
- ~1000x faster context switch than OS thread

Microsecond level
QMC solver w/ custom-made thread pool

1. // original implementation w/ custom thread pool
2. std::vector<std::future<void>> futures;
3.
4.
5. auto& pool = dca::parallel::ThreadPool::get_instance();
6.
7.
8. for (int i = 0; i < thread_task_handler_.size(); ++i) {
9.     if (thread_task_handler_.getTask(i) == "walker")
10.         futures.emplace_back(pool.enqueue(&ThisType::startWalker, this, i));
11.
12.     // else if handle other conditions...
13. }
QMC solver w/ threading abstraction

1. // new implementation w/ threading abstraction
2. std::vector<dca::parallel::thread_traits::future_type<void>> futures;
3. // switch to std::future or hpx::future at compile time
4. 
5. auto& pool = dca::parallel::ThreadPool::get_instance();
6. 
7. 
8. for (int i = 0; i < thread_task_handler_.size(); ++i) {
9. if (thread_task_handler_.getTask(i) == "walker")
10. futures.emplace_back(pool.enqueue(&ThisType::startWalker, this, i));
11. 
12. // else if handle other conditions...
13. }
Synchronization primitives in thread-pool class

std::thread
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>;
}
} // namespace parallel
} // namespace dca

HPX thread
namespace dca { namespace parallel {

struct thread_traits {
    template <typename T>
    using future_type = hpx::lcos::future<T>;
    using mutex_type = hpx::lcos::local::mutex;
    using condition_variable_type = hpx::lcos::local::condition_variable;
    using scoped_lock = std::lock_guard<mutex_type>;
    using unique_lock = std::unique_lock<mutex_type>;
}
} // namespace parallel
} // namespace dca
Runtime Comparison

- Configuration: 1 Summit node (6 MPI ranks; 7 CPUs + 1 GPU per rank)

- Results for 100k monte carlo measurements with error bars obtained from 5 independent executions.

- We observed **21% speedup using HPX threading** in DCA++ threaded QMC solver on Summit over C++ std threads.

- The speedup is due to **faster context switch and scheduler and less synchronization overhead** in HPX runtime system.
Optimized distributed computing with NVIDIA GPUDirect RDMA on Summit
Memory bound challenge and solution

Focus: Memory usage of the two-particle Green’s Function (G4) computation

- In general, size of G2 is ~30 MB, while G4 is 12 GB.
- V100 HBM on Summit: 16 GB
- Solution: *Broadcasting* each $G_2[][]$ matrix to all other ranks:
  - Traditional method
  - GPUDirect RDMA

Fig. : Roofline plot of a single NVIDIA V100 GPU running DCA++ at production level on Summit (OLCF).

Summit Node Layout

- NVLink 50 GB/s
- Nvidia V100 GPU
- IBM CPU
Transfer G2 around

Traditional method

Rank 0

Rank 1

cudA memcpy()

Memory copy everywhere: Device2Host, H2D, network transfer, etc.

GPUDirect method

Rank 0

Rank 1

GPUDirect

We avoid expensive memory copies and use high-speed network, the NVLink.

CUDA array  NVLink  50 GB/s  Nvidia V100 GPU  IBM P9
Performance Comparison

Compare bandwidth between gpuDirect and traditional methods

- on-node: traditional
- on-node: GPUDirect

up to 17x speedup!
Distributed G4 implementation

Original DCA++

Each rank keeps a private and full copy of G4.

DistG4 solution

Memory usage reduced. Each rank keeps 1/n G4 (n = #ranks).
## Enabling MORE science

<table>
<thead>
<tr>
<th></th>
<th>Original method</th>
<th>Distributed G4 w/ <strong>same</strong> science case</th>
<th>Distributed G4 w/ <strong>larger</strong> science case</th>
<th>What if much larger</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Memory usage per GPU</strong></td>
<td>Full, no more</td>
<td>$\frac{1}{6}$ is used, a lot left</td>
<td>Each GPU is fully used</td>
<td>X nodes</td>
</tr>
<tr>
<td></td>
<td><img src="image1" alt="Full, no more" /></td>
<td><img src="image2" alt="Distributed G4 w/ same science case" /></td>
<td><img src="image3" alt="Distributed G4 w/ larger science case" /></td>
<td><img src="image4" alt="X nodes" /></td>
</tr>
<tr>
<td><strong>Max. memory available for storing final G4 on one Summit node</strong></td>
<td>16 GB</td>
<td>96 GB</td>
<td><img src="image5" alt="96 * X (nodes) GB" /></td>
<td></td>
</tr>
<tr>
<td></td>
<td><img src="image6" alt="16 GB" /></td>
<td><img src="image7" alt="96 GB" /></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>6 X more per node w. DistG4 implementation!</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Then much more!
Summary

• **HPX light-weight threads**
  ○ Added HPX threading support and maintained same API of the thread pool in DCA++
  ○ **21% speedup using HPX threading** in DCA++ threaded QMC solver on Summit over C++ std threads

• **GPUDirect RDMA**
  ○ Implemented ring algorithm using GPUDirect capability enabling us to explore **large and complex science** problems
Ongoing work

• Multi platform effort:
  ○ Porting DCA++ w/ HPX to Arm64, Intel x86-64 and more...

• HPX task continuation:
  ○ Wrapping DCA++ cuda kernel into HPX future → overlapping communication and computation

• APEX + HPX Runtime:
  ○ Profiling DCA++ to identify bottlenecks and potential improvement in performance

• GPUDirect RDMA:
  ○ Adding bidirectional ring communication methods to utilize more bandwidth
Building Parallel Abstractions to DCA++ Scientific Software by Taking Advantage of HPX and GPUDirect

Weile Wei
PhD Student, Louisiana State University
Research Collaborator, Oak Ridge National Laboratory
wwei9@lsu.edu