Performance Analysis of a Quantum Monte Carlo Application on Multiple Hardware Architectures Using the HPX Runtime

Weile Wei  @weile_wei
Ph.D. student
Louisiana State University

ScalA20 @ SC20, Nov 12, 2020.
Performance Analysis of a Quantum Monte Carlo Application on Multiple Hardware Architectures Using the HPX Runtime

Authors:

Weile Wei
wwei9@lsu.edu

Arghya Chatterjee

Kevin Huck

Oscar Hernandez

Hartmut Kaiser
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.

Support for UO through the U.S. Department of Energy, Office of Science, ASCR, RAPIDS SciDAC Institute for Computer Science and Data under subcontract 4000159855 from ORNL.

CompFUSE: https://compfuse.ornl.gov
Outline

○ Quantum Monte Carlo solver application: DCA++

○ Threading abstraction using HPX
  ○ HPX runtime system
  ○ Performance optimization using hpx::thread over C++ Standard threads
  ○ Porting code to multi architectures (x86_64, Arm V8, Power9, … )
  ○ APEX +HPX runtime analysis -> in depth visualization of kernels

○ Ongoing efforts
  ○ Dynamic abstraction layer using GPUDirect RDMA on and across nodes
Dynamical Cluster Approximation (DCA++)
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.
- Gordon Bell Prize Winner 2008, a highly scalable application

DCA++: Primary workflow

\[
\text{Input} \\
\text{Initial Green's Function} \\
\& \text{Markov Chain}
\]

\[
\text{Coarse Graining} \\
(\text{Calculating Green's Function})
\]

\[
\text{Coarse-grained Green's Function} \\
[G]
\]

\[
\text{Iterative Convergence Algorithm}
\]

\[
\text{Quantum Monte Carlo solver} \\
(QMC)
\]

\[
\text{Output} \\
two \text{particle Green's Function} \\
[G_4]
\]

\[
\text{Green's Function} \\
[G_0]
\]
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 $G4. \rightarrow [G_2] * [G_2]$  

[most computation happens on GPU]

Walkers and accumulators are threaded
Threaded QMC workflow

Accumulator waiting queue

Acc.  ●●●  Acc.  Acc.

Multiple walkers

Walker  →  Producer

Walker

send results to

Acc.

Walker

→  Consumer

Pop acc. from acc. queue head once one walker finishes MC update

Push acc. to acc. queue’s back once it finishes MC measurement
Threading abstraction w/ HPX runtime system
Threading abstraction for QMC Solver

std::thread
(C++ standard library)

hpx::thread

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

DCA++ Custom-made Thread pool

Figure: workflow of thread-pool.
Note, adding hpx support does not change API and workflow 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
- 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 !!!

[GitHub](https://github.com/STEllAR-GROUP/hpx)
HPX Runtime System

- API
  - C++1y Parallelism APIs

- Policy/Engine/Policies
  - Threading Subsystem
  - Active Global Address Space (AGAS)

- OS

- Local Control Objects (LCOs, Synchronization)

- Parcel Transport Layer (Network)

- Performance Counter Framework
HPX - C++ standard compliant and more

- C++ standard library API compatible: (selected)
  
  - std::thread
  - std::mutex
  - std::future
  - std::async
  - std::function
  - hpx::thread
  - hpx::mutex
  - hpx::future
  - hpx::async
  - hpx::function

- Extend standard APIs where needed (compatibility is preserved)
HPX thread pool

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

- hpx::thread is user-level thread

- std::thread is kernel-level thread (or pthread)

Nanosecond level

Microsecond level
QMC solver w/ custom-made thread pool

1. // original implementation w/ custom thread pool
2. std::vector<std::future<void>> futures;
3. auto& pool = dca::parallel::ThreadPool::get_instance();
4. for (auto& task: thread_task_handler) {
5.     if (task.type() == "walker")
6.         futures.emplace_back(pool.enqueue(&ThisType::startWalker,
7.                                         this, task.id() ));
8.     // else if handle other conditions...
9. }
10. // else if handle other conditions...
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 (auto& task: thread_task_handler) {
9.    if (task.type() == "walker")
10.       futures.emplace_back(pool.enqueue(&ThisType::startWalker, this, task.id() ));
11. }
12. // else if handle other conditions...
Synchronization primitives in thread-pool class

```cpp
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::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>;
}
} // namespace parallel
} // namespace dca

https://github.com/STEllAR-GROUP/DCA/releases/tag/hpx_thread
```
Performance Analysis in-depth
Experiment setup

- Hardware configurations:

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Summit @ ORNL</th>
<th>Wombat @ ORNL</th>
<th>CoriGPU @ NERSC</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>Arm 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>

- For Summit run, each node has 6 MPI ranks; each rank has 7 CPUs + 1 GPU; each rank has 7 threads. This config. is the most optimal config. after parameters sweep.
Correctness on multi-architectures

- Obtained correct results from DCA++ runs on various architectures

(a) we validate our science case with C++ std :: thread implementation across three HPC platforms.

(b) Validation using the same case with hpx:: thread implementation across the same three systems. Additionally, we show the C++ std :: thread results on Summit as a reference.
Runtime Comparison

- Configuration: 1 Summit node
- Each node has 6 MPI ranks; each rank has 7 CPUs + 1 GPU; each rank has 7 threads. This config. is the most optimal config. after parameters sweep.

- 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. Same speedup observed in multi-node run.

- The speedup is due to faster context switch and scheduler and less synchronization overhead in HPX runtime system.
NVIDIA Nsight System Profiling for CPU utilization

- hpx::thread uses hwloc to achieve thread-affinity
  - sets one hyper-thread per physical core.
  - Overhead ➖
  - CPU Utilization ➔

- std::thread spreads work over 4 hyper-threads per physical core.
Autonomic Performance Environment for eXascale (APEX)
APEX

- Autonomic Performance Environment for eXascale
- **Performance Measurement** - designed for distributed, asynchronous tasking runtimes with task pre-emption and high concurrency (i.e. HPX)
  - Performance profiling & tracing support, task dependency graphs
  - Hardware / OS monitoring (utilization, power, HPX, others)
  - Native support for: OpenMP, NVIDIA CUDA, kokkos, OpenACC, C++

- **Policy Engine** - infrastructure for feedback/control, runtime optimization and auto-tuning (not used in this work)

https://github.com/khuck/xpress-apex
APEX Tracing HPX & CUDA, visualized in Vampir*

(above) Top 10 time consuming functions.

(left) Master timeline plot monitored events including CPU and GPU activities. APEX writes natively to OTF2, the trace format supported by Vampir. *https://vampir.eu
APEX monitoring of native HPX Performance Counters*

(a) HPX thread idle rate (unit: 0.01%), lower is better.

(b) HPX queue length (unit: number of available tasks), higher is better.

* https://hpx-docs.stellar-group.org/tags/1.3.0/html/manual/optimizing_hpx_applications.html
APEX monitoring of NVIDIA NVML data* & more

(a) APEX monitoring results on Summit summarizing device memory used (unit: megabytes) over time

(b) more GPU/CPU profiling options

Ongoing work

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

• **GPUDirect RDMA:**
  ○ Building ring abstraction over GPUDirect to optimize distributed quantum monte carlo solver

• **Arm A64fx evaluation:**
  ○ Evaluate DCA++ performance on Arm A64fx cluster (Wombat @ORNL)