

# ENABLING PARALLEL ABSTRACTION LAYER TO DCA++ USING HPX AND GPUDIRECT

A Project Report

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  
Master of Science in Computer Science

in

The Department of Computer Science and Engineering

by  
Weile Wei  
May 2020

## Acknowledgments

I would like to take this opportunity to thank my committee members: Dr. Hartmut Kaiser, Dr. Golden G. Richard III, Dr. Bijaya B. Karki. Especially, heartfelt gratitude goes to my advisor Dr. Hartmut Kaiser, who has been an invaluable and patient mentor throughout my journey of Master in Computer Science at LSU. During the past two years, Dr. Kaiser helped me learn C++ programming, guided me through the high-performance computing field, and allowed me explore the world.

It is a pleasure also to thank my collaborators Arghya “Ronnie” Chatterjee (ORNL), Oscar Hernandez (ORNL), Thomas Maier (ORNL), Ed D’Azevedo (ORNL), Peter Doak (ORNL), John Biddiscombe (CSCS), Giovanni Balduzzi (ETH Zurich), and Ying Wai Li (LANL) for their thoughtful research suggestions on this project. I would like to express my sincerest gratitude to Ronnie for his timely and invaluable support throughout my ASTRO intern program at ORNL.

I would like to thank my colleagues and friends at STE||AR group, Adrian, Ali, Bibek, Bita, Katie, Max, Nanmiao, Parsa, Patrick, Rod, Shahrzad, Steve and Tianyi for their constructive discussions, invaluable suggestions, and necessary distractions. I would like to thank 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 parents and my family members for their patience and warm supports for my study in the U.S. 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. I would like to take this opportunity to express my love and say thanks to my family in my mother tongue: 谢谢家人的支持，我爱你们！.

## Table of Contents

Acknowledgements .....	ii
List of Tables .....	iv
List of Figures.....	v
List of Listings .....	vi
Abstract .....	vii
Chapter	
1.    Introduction .....	1
2.    HPX Threading Abstraction .....	5
2.1.  HPX Runtime System and its Facilities .....	5
2.2.  HPX Threading Implementation in DCA++ .....	9
2.3.  Experiment and Results .....	10
3.    GPUDirect .....	14
3.1.  Memory Bound Issue and Solution .....	14
3.2.  GPUDirect on Summit .....	15
3.3.  MPI Ping-pong Benchmark .....	16
3.4.  Bandwidth Measurement on NVLink on Summit .....	22
3.5.  Pipeline Ring Algorithm .....	25
4.    Conclusion and Future Work .....	29
References .....	30
Vita .....	32

## List of Tables

3.1. Compare NVLink bandwidth speedup on Summit .....	23
---	----

## List of Figures

1.1.	DCA++ workflow .....	2
1.2.	DCA++ Quantum Monte Carlo Solver .....	2
2.1.	user-level light-weight HPX thread and kernel thread .....	6
2.2.	What is a future? .....	8
2.3.	New threading abstraction layer in DCA++ .....	9
2.4.	Difference between custom-made threadpool and HPX threading abstraction in DCA++ .....	10
2.5.	Compare the computation accuracy of HPX-enabled DCA++ and original DCA++ .....	11
2.6.	Compare DCA++ execution time between custom-made threadpool and hpx threadpool (threads=7) .....	12
2.7.	Compare DCA++ execution time among different number of HPX threads .....	13
3.1.	Roofline plot of a NVIDIA V100 GPU running DCA++ at production level on Summit. Figure adapted from [1] .....	14
3.2.	Memory bound issue with G4 matrix .....	15
3.3.	GPU data transfer between GPUs using traditional MPI GPU to Remote GPU method .....	16
3.4.	GPU data transfer between GPUs using GPUDirect .....	18
3.5.	On-node bandwidth comparisons: DCA++ mini-application using NVLink and Up-and-down method) .....	20
3.6.	Distributed G4 with NVLink enabled .....	22
3.7.	Compare bandwidth of data transfer on Summit using NVLink .....	23
3.8.	Two nodes are located in two different sides of the facility room .....	24
3.9.	Two nodes are located close to each other in the facility room .....	24

## Listings

3.1. MPI GPU to Remote GPU .....	16
3.2. GPUDirect peer to peer.....	18
3.3. pipeline ring algorithm.....	26

## Abstract

DCA++ is a high-performance research software framework providing modern C++ implementation to solve quantum many-body problems. Currently, DCA++ is facing two challenges: 1) DCA++ needs to prepare portability layer for Frontier, the next-generation exascale supercomputer at Oak Ridge National Lab and 2) DCA++ kernel computation is memory-bound on Summit Supercomputer’s NVIDIA V100’s.

High-Performance ParalleX (HPX), which is a C++ runtime-support system for parallel computation and has been ported to many operating systems, is a good candidate for the portability layer for DCA++. We have built a threading abstraction layer that can choose DCA++’s threading dependency between the custom-made thread pool in DCA++ and HPX light-weight thread pool by passing the threading compilation flag such that chosen thread-pool is used for the whole source-code in DCA++. We compared the runtime performance of the two thread-pools and our experimental runs show HPX-enabled DCA++ is slightly faster than the original DCA++ on supercomputer Summit.

To address the second challenge, we explored GPUDirect communication technique using NVIDIA’s NVLink interconnect (a high bandwidth low latency switched fabric interconnect method) on Summit. We designed a mini-app for comparing the NVLink approach and regular communication method and have observed up to 17x speedup using NVLink approach on the min-app on Summit. We further analyzed NVLink bandwidth in different scenarios on Summit using different hardware resources (i.e. NVLink across nodes and racks). This NVLink approach enabled us to drastically reduce memory usage in DCA++ by distributing arrays across nodes.

We plan to apply GPUDirect high bandwidth technique into DCA++ code to address the memory bound issue. Also, we plan to integrate GPUDirect into HPX so we can construct task dependency and attach more task continuations by overlapping network communication and computation to fully utilize hardware resources and improve code performance.

## Chapter 1. Introduction

Legacy scientific applications must be ported to large scale and highly heterogeneous HPC systems. Portability and scalability of such applications are paramount to the design and implementation [2, 3]. In this project, we present the new strategies opted to enable high-level parallel abstraction to a legacy-condensed matter physics application using High-performance ParalleX and GPUDirect to exploit the application’s portability and scalability.

Developed by Oak Ridge National Lab, Swiss National Supercomputing Centre (CSCS), and ETH Zurich, DCA++ (Dynamical Cluster Approximation) [4,5] is a high-performance research software framework, providing modern C++ implementation to solve quantum many-body problems. It’s a numerical simulation tool developed to understand the behaviors (such as superconductivity, magnetism, etc.) of co-related quantum materials. DCA++ has been successfully ported to the world’s largest supercomputers, e.g. Titan (Oak Ridge Leadership Computing Facility), Summit (OLCF) & Piz Daint (CSCS) sustaining many petaflops of performance. Recent work shows [1] a perfect strong and weak scaling for the Quantum Monte Carlo solver in DCA++ on Summit at 4600 nodes at 73.5 PFLOPS (mixed precision), but our solver computation is at memory bound on the NVIDIA V100’s.

Figure 1.1 shows the primary workflow in DCA++. It takes the initial Green’s function & Markov Chain as input parameters. After several iterative computations, DCA++ will converge and output a two-particle Green’s function (G4 array). The iteration process involves two critical computation-intensive steps, including coarse-graining and Quantum Monte Carlo solver (QMC Solver). QMC solver is the most computation intensive step in DCA++ and hence is our focus to optimize its performance.

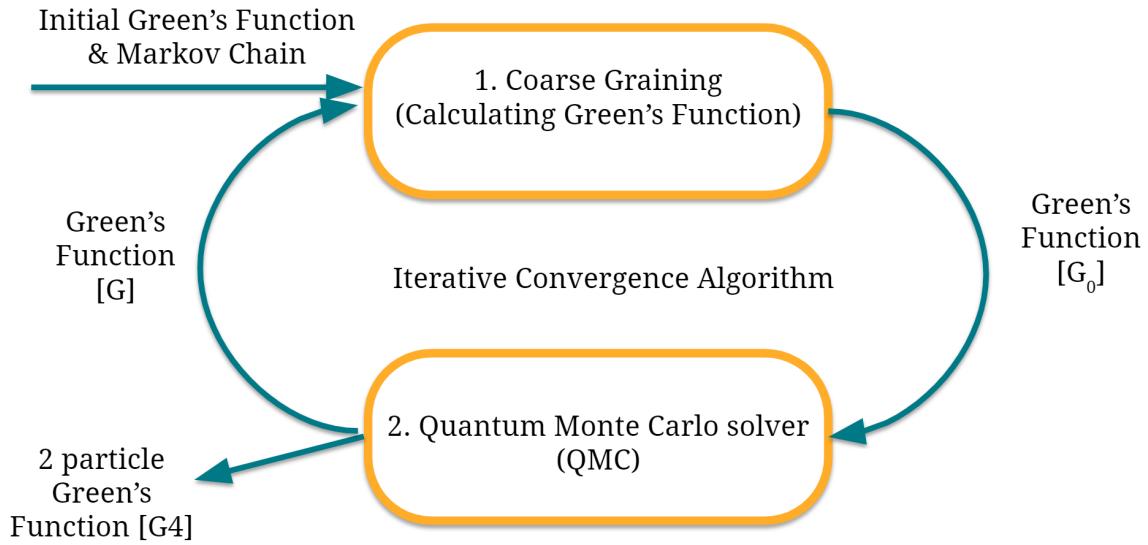


Figure 1.1. DCA++ workflow

The details of QMC solver is shown in Figure 1.2. DCA++ application starts in multiple MPI ranks. Each MPI rank is first performing its independent computation and communicates with each other at the end through calling `mpi_all_reduce` routine. During the independent computation stage, a custom-made thread-pool is used to run tasks in parallel (i.e. walker, accumulator).

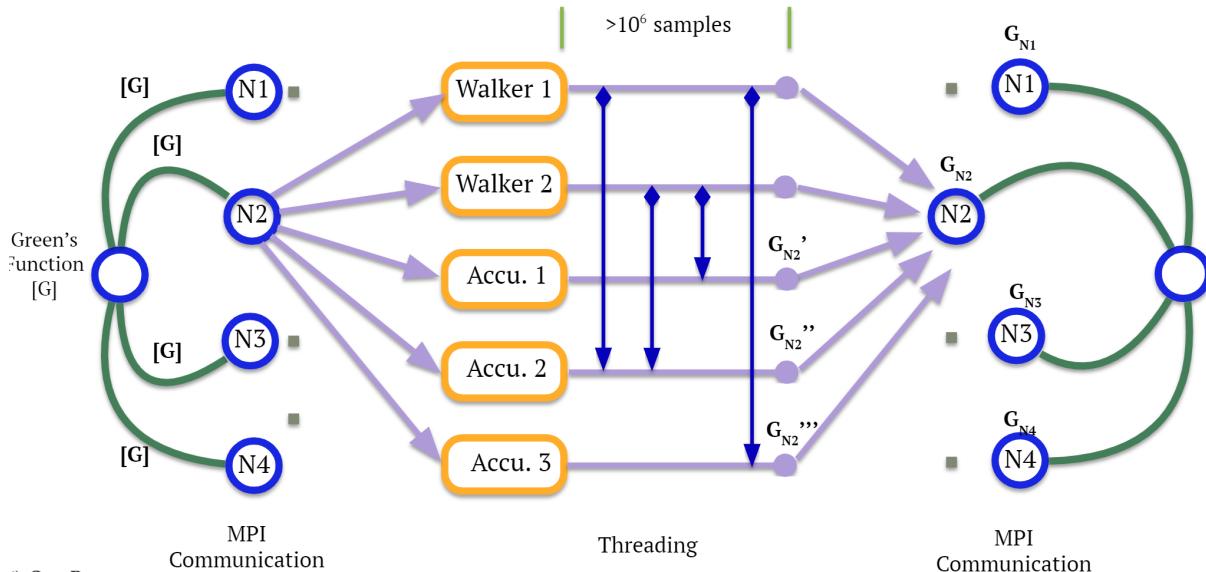


Figure 1.2. DCA++ Quantum Monte Carlo Solver

With regard to the increasing heterogeneity of modern computing machines due to

energy constraints and changes to chips and memory architecture, porting scientific software to new hardware and architectures has become a grand challenge. This requires advanced programming techniques in software development and design. Ensuring long term platform stability and portability is very important for scientific codes.

HPX [6], a C++ Standard Library for Parallelism and Concurrency, provides an innovative implementation of C++ runtime system architecture with high-level parallelization abstractions. In this project, we have built a light-weight user threading model provided by HPX to DCA++, porting the current application to asynchronous task-based execution model. Our current result shows that HPX-enabled DCA++ produces comparable accuracy to the originally implemented custom-made thread pool in DCA++ and has slightly faster runtime performance.

In the project, we also attempt to address the memory bound challenges in DCA++ [1] and discuss our DCA++ lite benchmark (mini-application expressing the expensive computational motif with a relatively higher memory usage in V100 GPU) on comparing GPUDirect communication technique (a high-bandwidth, low-latency switched fabric interconnect method), and regular communication technique. We have observed 17x bandwidth speedup using GPUDirect for the benchmark application on the Summit supercomputer. This newly added GPUDirect approach proved the feasibility of constructing another high-level parallel abstraction layer in DCA++ that can take advantage of GPUDirect. This could enable us to easily switch from original network communication pattern in DCA++ to this new approach. More importantly, the GPUDirect approach will drastically reduce memory usage in DCA++ by distributing arrays across nodes, and therefore, we could utilize more kernel memory to conduct more scientific measurements. With the scalability support provided by GPUDirect approach, one can explore more scientific computation in DCA++ to study high-temperature superconductivity in material science.

In this report, chapter 2 introduces HPX and analyzes the performance of HPX threading mechanism in DCA++; chapter 3 presents the bandwidth evaluations of GPUDirect

techniques under varies hardware configuration on Summit; lastly, chapter 4 summarizes this research project and provides an outlook of our future work.

## Chapter 2. HPX Threading Abstraction

### 2.1. HPX Runtime System and its Facilities

HPX (High Performance ParalleX) is a fully Asynchronous Many Task (AMT) runtime system extending the C++ programming language. HPX runtime system is implemented on lightweight user-level tasks manager running on top of kernel threads. HPX is the first implementation of the ParaleX execution model [7], which essentially solves critical challenges that prevent effective usage of new HPC systems: Starvation, Latency, Overheads, and Waiting for Contention.

HPX programming model exposes a C++ standard API entirely conforming to interfaces as defined by C++11/C++14/C++17 and extended the C++ standard to distributed and heterogeneous computing scenarios, which makes HPX easy to use and very portable. HPX aligns itself with the ongoing C++ standardization proposals with a goal of providing a uniform interface, in particular, related to parallelism and concurrency. This project presents our input of various language features into proposals to current C++ standardization process. It is critical that we provide this user feedback to C++ standards committee so that the future language standard will natively support HPC use cases and eventually benefit our HPC community.

#### 2.1.1. HPX Light-weight thread

The HPX light-weight threading system provides user-level threads, which enables fast context switching [8]. Smaller overheads allow the program to be broken up into smaller tasks, which in turns helps the runtime system to dynamically schedule these smaller tasks into physical processing units whenever the units have free resources therefore fully utilizing all processing units. With lower overheads (less time spent on creating, scheduling, executing and destroying threads) per thread, programs are able to create and schedule a large number of tasks with little penalty.

In the context of multitasking, the word "context switch" refers to the process of storing

the state for one process, so that it can be paused (and reloaded when required) and another process resumed. The process of context switching can have a negative impact on system performance [9]. Switching from one process to another requires a certain amount of time – changes of registers, program counter, stack, etc.

The idea behind light-weight thread is to switch one task to another as quickly as possible. This lightweight thread approach enables the operating system to take a thread and match it to one or more lightweight threads. Lightweight threads are thousands of times faster than kernel level threads (creating, scheduling, destroying, etc.). Figure 2.1 shows the relation between user-level light-weight HPX thread, and kernel thread. In user-level space, HPX has its own thread scheduler to manage light-weight threads, and this scheduler can smartly assign its threads to kernel threads when appropriate by using scheduling algorithms [10].

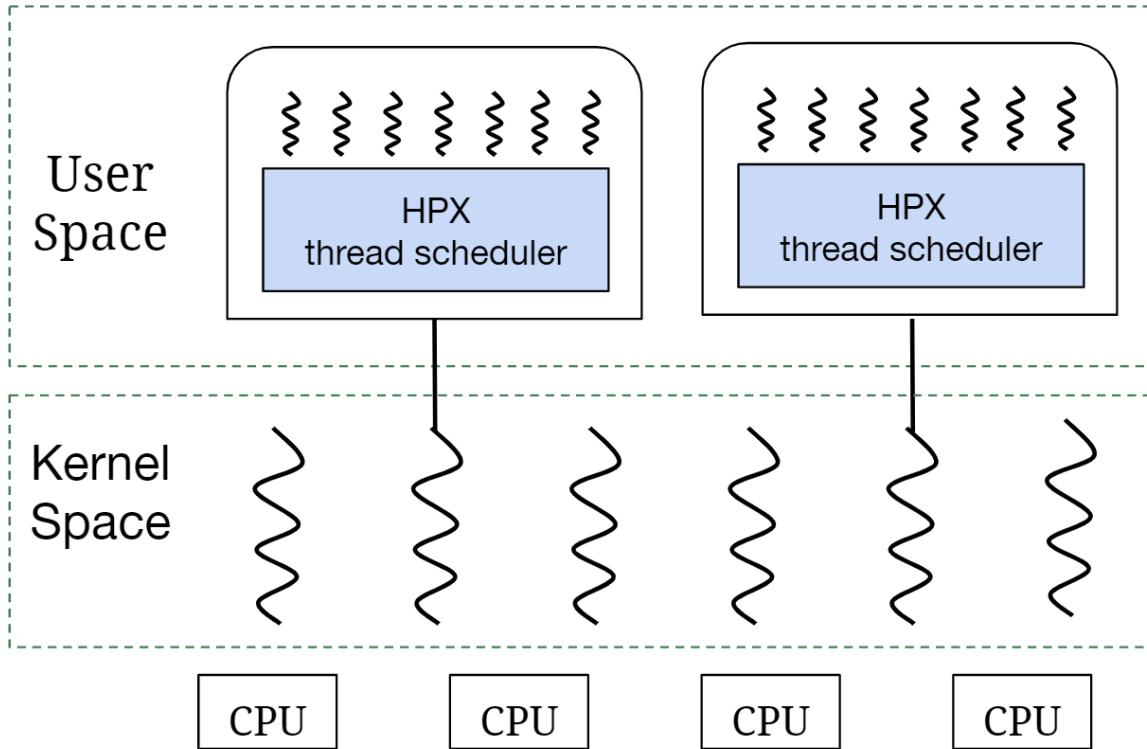


Figure 2.1. user-level light-weight HPX thread and kernel thread

The advantage of HPX lightweight threading system combined with the future func-

tionality in HPX facilitates auto-parallelization in a highly efficient fashion as such combination allows the direct expression of the generated dependency graph as an execution tree generated at runtime.

### 2.1.2. Future

A future is an object representing a result which has not been computed yet. The future [11–13] enables a transparent synchronization between the producer and its consumer; hides the notion of directly dealing with threads; makes asynchrony manageable as a future represents a data dependency; allows coordination of asynchronous execution of several tasks; supports and encourages a programming style which favors parallelism over concurrency.

Figure 2.2 shows the future object in the perspective of consumer-producer relationship. In consumer thread, a `future` object is created by calling `async` function and is dispatched to a producer thread to be computed. Calling `get()` function on a `future` object is a way to communicate and synchronize consumer and producer. By calling `get()`, two possible results can happen: 1) if the computation is finished, then program will handle results back to function caller; 2) if the computation is ongoing, the program will put the current thread to sleep and let current threads wait until computation has finished.

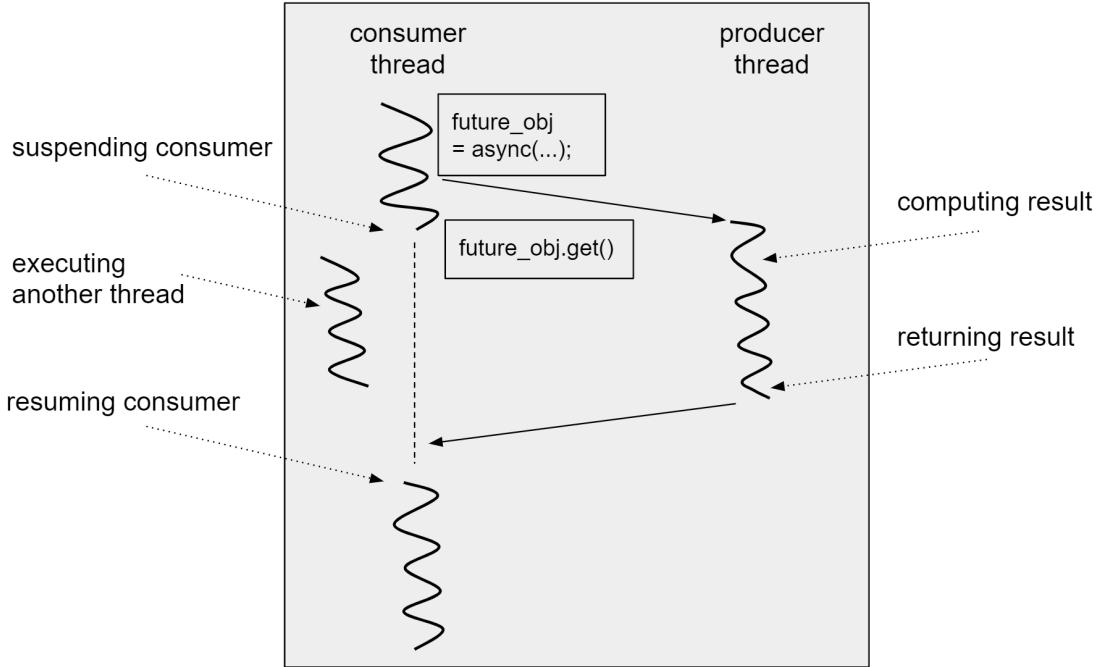


Figure 2.2. What is a future?

One of main differences between using `std::future` and `hpx::future` is that `hpx::future` is running in HPX lightweight thread while `std::future` is running in kernel thread. If calling `get()` on an `hpx::future` object and this object has not finished its computation, then the current lightweight thread will be put on sleep; meanwhile, the underlying kernel threads can still schedule other tasks if there are any. However, in the same situation for `std::future` will block kernel thread and leave physical core to be idle. Therefore, HPX light-weight thread combined with the future functionality can achieve lower overhead and better hardware utilization. This also means that with the help of HPX threading scheduler, we can easily oversubscribe thousands of millions of tasks to keep the hardware busy if needed.

### 2.1.3. Task continuation in HPX

Apart from HPX lightweight thread and `hpx::future` we introduced above, HPX has other facilities that can't be performed using C++ standard library to support task-based programming, such as `then` (a method of `hpx::future<T>`), `hpx::dataflow`, etc.

In asynchronous many-task programming, it is common for one asynchronous operation, once completed, to invoke a second function along with data [11]. For example, the current C++ standard does not allow connecting such continuation to a `future`. With `hpx::future`, instead of waiting for the returning result, a continuation can be “attached” to the asynchronous operation, which is invoked when the result is returned. `then` function will help to avoid blocking waits (i.e. `.get()`) or wasting threads on polling, greatly improving the responsiveness of a program. Even more conveniently, one can create a chaining futures if needed by attaching `hpx::future` after `hpx::future`. By creating continuation to a single `hpx::future`, an implicit dependency graph is therefore constructed.

## 2.2. HPX Threading Implementation in DCA++

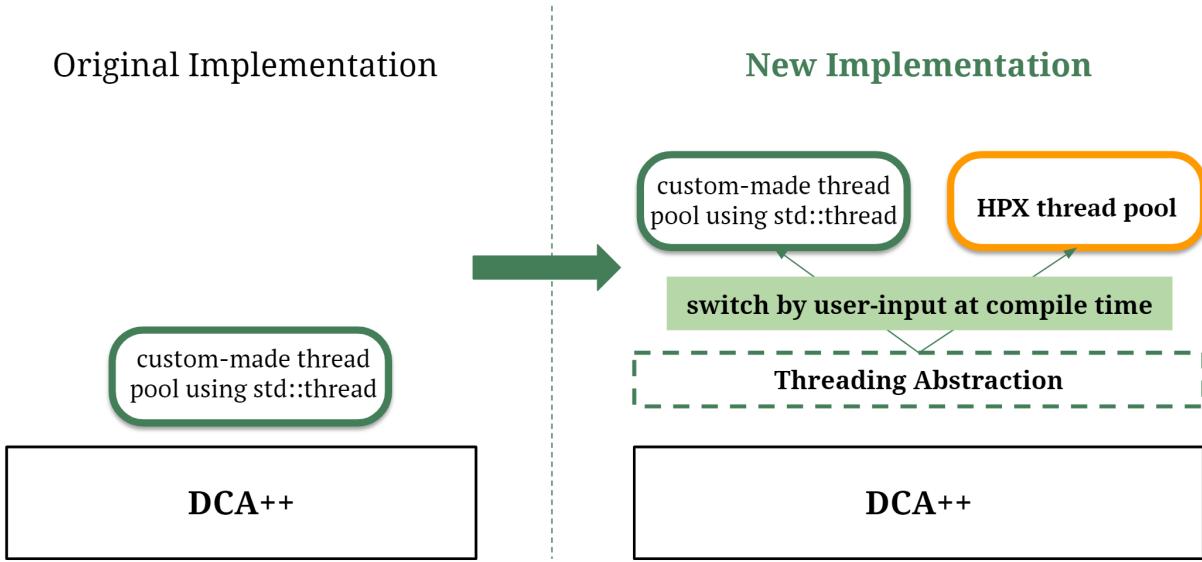


Figure 2.3. New threading abstraction layer in DCA++

Figure 2.3 shows new threading abstraction layer that can choose DCA++’s threading pool between the custom-made thread pool in DCA++ and HPX light-weight thread pool by a user’s compilation flag such that chosen thread-pool is used for the whole source-code in DCA++.

Figure 2.4 shows the code that how HPX threading abstraction layer replaces custom-made thread pool. First, instead of creating a vector of `std::future<void>` objects, we create

a vector of `future_type<void>` objects to represent a list of to-be-compute tasks. Then, instead of creating a static thread-pool in custom-made thread-pool approach, we skip this stage because HPX has its own thread-pool manager underneath and therefore does not need to again create a thread-pool. When launching tasks, original DCA++ inserts each task into the custom-made thread-pool; in HPX version, we use `hpx::async` to launch tasks. At the end, original DCA++ calls `get()` method on each task to join the result of all given futures, while HPX version just calls `hpx::wait_all()` to join all the results.

```

162
163 - std::vector<std::future<void>> futures;
164
165     dca::profiling::WallTime start_time;
166
167 - auto& pool = dca::parallel::ThreadPool::get_instance();
168     for (int i = 0; i < thread_task_handler_.size(); ++i) {
169         if (thread_task_handler_.getTask(i) == "Walker")
170 -         futures.emplace_back(pool.enqueue(&ThisType::startWalker, this, i));
171         else if (thread_task_handler_.getTask(i) == "accumulator")
172 -         futures.emplace_back(pool.enqueue(&ThisType::startAccumulator, this, i));
173         else if (thread_task_handler_.getTask(i) == "Walker and accumulator")
174 -         futures.emplace_back(pool.enqueue(&ThisType::startWalkerAndAccumulator, this,
175             i));
176         else
177             throw std::logic_error("Thread task is undefined.");
178     }
179
180     try {
181         for (auto& future : futures)
182             future.get();
183     }
184     catch (std::exception& err) {
185         print_metadata();
186     }
187
188     @@ -188,8 +188,9 @@ void StdThreadQmcSolver::integrate() {
189
190         try {
191             for (auto& future : futures)
192                 future.get();
193
194         }
195         catch (std::exception& err) {
196             print_metadata();
197         }
198
199         try {
200             for (auto& future : futures)
201                 future.get();
202
203             hpx::wait_all(futures);
204
205         }
206         catch (std::exception& err) {
207             print_metadata();
208         }
209
210     }
211
212     print_metadata();
213
214     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
215
216         try {
217             for (auto& future : futures)
218                 future.get();
219
220             hpx::wait_all(futures);
221
222         }
223         catch (std::exception& err) {
224             print_metadata();
225         }
226
227     }
228
229     print_metadata();
230
231     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
232
233         try {
234             for (auto& future : futures)
235                 future.get();
236
237             hpx::wait_all(futures);
238
239         }
240         catch (std::exception& err) {
241             print_metadata();
242         }
243
244     }
245
246     print_metadata();
247
248     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
249
250         try {
251             for (auto& future : futures)
252                 future.get();
253
254             hpx::wait_all(futures);
255
256         }
257         catch (std::exception& err) {
258             print_metadata();
259         }
260
261     }
262
263     print_metadata();
264
265     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
266
267         try {
268             for (auto& future : futures)
269                 future.get();
270
271             hpx::wait_all(futures);
272
273         }
274         catch (std::exception& err) {
275             print_metadata();
276         }
277
278     }
279
280     print_metadata();
281
282     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
283
284         try {
285             for (auto& future : futures)
286                 future.get();
287
288             hpx::wait_all(futures);
289
290         }
291         catch (std::exception& err) {
292             print_metadata();
293         }
294
295     }
296
297     print_metadata();
298
299     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
300
301         try {
302             for (auto& future : futures)
303                 future.get();
304
305             hpx::wait_all(futures);
306
307         }
308         catch (std::exception& err) {
309             print_metadata();
310         }
311
312     }
313
314     print_metadata();
315
316     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
317
318         try {
319             for (auto& future : futures)
320                 future.get();
321
322             hpx::wait_all(futures);
323
324         }
325         catch (std::exception& err) {
326             print_metadata();
327         }
328
329     }
330
331     print_metadata();
332
333     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
334
335         try {
336             for (auto& future : futures)
337                 future.get();
338
339             hpx::wait_all(futures);
340
341         }
342         catch (std::exception& err) {
343             print_metadata();
344         }
345
346     }
347
348     print_metadata();
349
350     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
351
352         try {
353             for (auto& future : futures)
354                 future.get();
355
356             hpx::wait_all(futures);
357
358         }
359         catch (std::exception& err) {
360             print_metadata();
361         }
362
363     }
364
365     print_metadata();
366
367     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
368
369         try {
370             for (auto& future : futures)
371                 future.get();
372
373             hpx::wait_all(futures);
374
375         }
376         catch (std::exception& err) {
377             print_metadata();
378         }
379
380     }
381
382     print_metadata();
383
384     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
385
386         try {
387             for (auto& future : futures)
388                 future.get();
389
390             hpx::wait_all(futures);
391
392         }
393         catch (std::exception& err) {
394             print_metadata();
395         }
396
397     }
398
399     print_metadata();
400
401     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
402
403         try {
404             for (auto& future : futures)
405                 future.get();
406
407             hpx::wait_all(futures);
408
409         }
410         catch (std::exception& err) {
411             print_metadata();
412         }
413
414     }
415
416     print_metadata();
417
418     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
419
420         try {
421             for (auto& future : futures)
422                 future.get();
423
424             hpx::wait_all(futures);
425
426         }
427         catch (std::exception& err) {
428             print_metadata();
429         }
430
431     }
432
433     print_metadata();
434
435     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
436
437         try {
438             for (auto& future : futures)
439                 future.get();
440
441             hpx::wait_all(futures);
442
443         }
444         catch (std::exception& err) {
445             print_metadata();
446         }
447
448     }
449
450     print_metadata();
451
452     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
453
454         try {
455             for (auto& future : futures)
456                 future.get();
457
458             hpx::wait_all(futures);
459
460         }
461         catch (std::exception& err) {
462             print_metadata();
463         }
464
465     }
466
467     print_metadata();
468
469     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
470
471         try {
472             for (auto& future : futures)
473                 future.get();
474
475             hpx::wait_all(futures);
476
477         }
478         catch (std::exception& err) {
479             print_metadata();
480         }
481
482     }
483
484     print_metadata();
485
486     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
487
488         try {
489             for (auto& future : futures)
490                 future.get();
491
492             hpx::wait_all(futures);
493
494         }
495         catch (std::exception& err) {
496             print_metadata();
497         }
498
499     }
500
501     print_metadata();
502
503     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
504
505         try {
506             for (auto& future : futures)
507                 future.get();
508
509             hpx::wait_all(futures);
510
511         }
512         catch (std::exception& err) {
513             print_metadata();
514         }
515
516     }
517
518     print_metadata();
519
520     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
521
522         try {
523             for (auto& future : futures)
524                 future.get();
525
526             hpx::wait_all(futures);
527
528         }
529         catch (std::exception& err) {
530             print_metadata();
531         }
532
533     }
534
535     print_metadata();
536
537     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
538
539         try {
540             for (auto& future : futures)
541                 future.get();
542
543             hpx::wait_all(futures);
544
545         }
546         catch (std::exception& err) {
547             print_metadata();
548         }
549
550     }
551
552     print_metadata();
553
554     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
555
556         try {
557             for (auto& future : futures)
558                 future.get();
559
560             hpx::wait_all(futures);
561
562         }
563         catch (std::exception& err) {
564             print_metadata();
565         }
566
567     }
568
569     print_metadata();
570
571     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
572
573         try {
574             for (auto& future : futures)
575                 future.get();
576
577             hpx::wait_all(futures);
578
579         }
580         catch (std::exception& err) {
581             print_metadata();
582         }
583
584     }
585
586     print_metadata();
587
588     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
589
590         try {
591             for (auto& future : futures)
592                 future.get();
593
594             hpx::wait_all(futures);
595
596         }
597         catch (std::exception& err) {
598             print_metadata();
599         }
600
601     }
602
603     print_metadata();
604
605     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
606
607         try {
608             for (auto& future : futures)
609                 future.get();
610
611             hpx::wait_all(futures);
612
613         }
614         catch (std::exception& err) {
615             print_metadata();
616         }
617
618     }
619
620     print_metadata();
621
622     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
623
624         try {
625             for (auto& future : futures)
626                 future.get();
627
628             hpx::wait_all(futures);
629
630         }
631         catch (std::exception& err) {
632             print_metadata();
633         }
634
635     }
636
637     print_metadata();
638
639     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
640
641         try {
642             for (auto& future : futures)
643                 future.get();
644
645             hpx::wait_all(futures);
646
647         }
648         catch (std::exception& err) {
649             print_metadata();
650         }
651
652     }
653
654     print_metadata();
655
656     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
657
658         try {
659             for (auto& future : futures)
660                 future.get();
661
662             hpx::wait_all(futures);
663
664         }
665         catch (std::exception& err) {
666             print_metadata();
667         }
668
669     }
670
671     print_metadata();
672
673     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
674
675         try {
676             for (auto& future : futures)
677                 future.get();
678
679             hpx::wait_all(futures);
680
681         }
682         catch (std::exception& err) {
683             print_metadata();
684         }
685
686     }
687
688     print_metadata();
689
690     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
691
692         try {
693             for (auto& future : futures)
694                 future.get();
695
696             hpx::wait_all(futures);
697
698         }
699         catch (std::exception& err) {
700             print_metadata();
701         }
702
703     }
704
705     print_metadata();
706
707     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
708
709         try {
710             for (auto& future : futures)
711                 future.get();
712
713             hpx::wait_all(futures);
714
715         }
716         catch (std::exception& err) {
717             print_metadata();
718         }
719
720     }
721
722     print_metadata();
723
724     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
725
726         try {
727             for (auto& future : futures)
728                 future.get();
729
730             hpx::wait_all(futures);
731
732         }
733         catch (std::exception& err) {
734             print_metadata();
735         }
736
737     }
738
739     print_metadata();
740
741     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
742
743         try {
744             for (auto& future : futures)
745                 future.get();
746
747             hpx::wait_all(futures);
748
749         }
750         catch (std::exception& err) {
751             print_metadata();
752         }
753
754     }
755
756     print_metadata();
757
758     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
759
760         try {
761             for (auto& future : futures)
762                 future.get();
763
764             hpx::wait_all(futures);
765
766         }
767         catch (std::exception& err) {
768             print_metadata();
769         }
770
771     }
772
773     print_metadata();
774
775     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
776
777         try {
778             for (auto& future : futures)
779                 future.get();
780
781             hpx::wait_all(futures);
782
783         }
784         catch (std::exception& err) {
785             print_metadata();
786         }
787
788     }
789
789     print_metadata();
790
791     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
792
793         try {
794             for (auto& future : futures)
795                 future.get();
796
797             hpx::wait_all(futures);
798
799         }
800         catch (std::exception& err) {
801             print_metadata();
802         }
803
804     }
805
805     print_metadata();
806
807     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
808
809         try {
810             for (auto& future : futures)
811                 future.get();
812
813             hpx::wait_all(futures);
814
815         }
816         catch (std::exception& err) {
817             print_metadata();
818         }
819
820     }
821
821     print_metadata();
822
823     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
824
825         try {
826             for (auto& future : futures)
827                 future.get();
828
829             hpx::wait_all(futures);
830
831         }
832         catch (std::exception& err) {
833             print_metadata();
834         }
835
836     }
837
837     print_metadata();
838
839     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
840
841         try {
842             for (auto& future : futures)
843                 future.get();
844
845             hpx::wait_all(futures);
846
847         }
848         catch (std::exception& err) {
849             print_metadata();
850         }
851
852     }
853
853     print_metadata();
854
855     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
856
857         try {
858             for (auto& future : futures)
859                 future.get();
860
861             hpx::wait_all(futures);
862
863         }
864         catch (std::exception& err) {
865             print_metadata();
866         }
867
868     }
869
869     print_metadata();
870
871     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
872
873         try {
874             for (auto& future : futures)
875                 future.get();
876
877             hpx::wait_all(futures);
878
879         }
880         catch (std::exception& err) {
881             print_metadata();
882         }
883
884     }
885
885     print_metadata();
886
887     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
888
889         try {
890             for (auto& future : futures)
891                 future.get();
892
893             hpx::wait_all(futures);
894
895         }
896         catch (std::exception& err) {
897             print_metadata();
898         }
899
900     }
901
901     print_metadata();
902
903     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
904
905         try {
906             for (auto& future : futures)
907                 future.get();
908
909             hpx::wait_all(futures);
910
911         }
912         catch (std::exception& err) {
913             print_metadata();
914         }
915
916     }
917
917     print_metadata();
918
919     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
920
921         try {
922             for (auto& future : futures)
923                 future.get();
924
925             hpx::wait_all(futures);
926
927         }
928         catch (std::exception& err) {
929             print_metadata();
930         }
931
932     }
933
933     print_metadata();
934
935     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
936
937         try {
938             for (auto& future : futures)
939                 future.get();
940
941             hpx::wait_all(futures);
942
943         }
944         catch (std::exception& err) {
945             print_metadata();
946         }
947
948     }
949
949     print_metadata();
950
951     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
952
953         try {
954             for (auto& future : futures)
955                 future.get();
956
957             hpx::wait_all(futures);
958
959         }
960         catch (std::exception& err) {
961             print_metadata();
962         }
963
964     }
965
965     print_metadata();
966
967     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
968
969         try {
970             for (auto& future : futures)
971                 future.get();
972
973             hpx::wait_all(futures);
974
975         }
976         catch (std::exception& err) {
977             print_metadata();
978         }
979
980     }
981
981     print_metadata();
982
983     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
984
985         try {
986             for (auto& future : futures)
987                 future.get();
988
989             hpx::wait_all(futures);
990
991         }
992         catch (std::exception& err) {
993             print_metadata();
994         }
995
996     }
997
997     print_metadata();
998
999     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1000
1001         try {
1002             for (auto& future : futures)
1003                 future.get();
1004
1005             hpx::wait_all(futures);
1006
1007         }
1008         catch (std::exception& err) {
1009             print_metadata();
1010         }
1011
1012     }
1013
1013     print_metadata();
1014
1015     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1016
1017         try {
1018             for (auto& future : futures)
1019                 future.get();
1020
1021             hpx::wait_all(futures);
1022
1023         }
1024         catch (std::exception& err) {
1025             print_metadata();
1026         }
1027
1028     }
1029
1029     print_metadata();
1030
1031     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1032
1033         try {
1034             for (auto& future : futures)
1035                 future.get();
1036
1037             hpx::wait_all(futures);
1038
1039         }
1040         catch (std::exception& err) {
1041             print_metadata();
1042         }
1043
1044     }
1045
1045     print_metadata();
1046
1047     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1048
1049         try {
1050             for (auto& future : futures)
1051                 future.get();
1052
1053             hpx::wait_all(futures);
1054
1055         }
1056         catch (std::exception& err) {
1057             print_metadata();
1058         }
1059
1060     }
1061
1061     print_metadata();
1062
1063     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1064
1065         try {
1066             for (auto& future : futures)
1067                 future.get();
1068
1069             hpx::wait_all(futures);
1070
1071         }
1072         catch (std::exception& err) {
1073             print_metadata();
1074         }
1075
1076     }
1077
1077     print_metadata();
1078
1079     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1080
1081         try {
1082             for (auto& future : futures)
1083                 future.get();
1084
1085             hpx::wait_all(futures);
1086
1087         }
1088         catch (std::exception& err) {
1089             print_metadata();
1090         }
1091
1092     }
1093
1093     print_metadata();
1094
1095     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1096
1097         try {
1098             for (auto& future : futures)
1099                 future.get();
1100
1101             hpx::wait_all(futures);
1102
1103         }
1104         catch (std::exception& err) {
1105             print_metadata();
1106         }
1107
1108     }
1109
1109     print_metadata();
1110
1111     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1112
1113         try {
1114             for (auto& future : futures)
1115                 future.get();
1116
1117             hpx::wait_all(futures);
1118
1119         }
1120         catch (std::exception& err) {
1121             print_metadata();
1122         }
1123
1124     }
1125
1125     print_metadata();
1126
1127     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1128
1129         try {
1130             for (auto& future : futures)
1131                 future.get();
1132
1133             hpx::wait_all(futures);
1134
1135         }
1136         catch (std::exception& err) {
1137             print_metadata();
1138         }
1139
1140     }
1141
1141     print_metadata();
1142
1143     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1144
1145         try {
1146             for (auto& future : futures)
1147                 future.get();
1148
1149             hpx::wait_all(futures);
1150
1151         }
1152         catch (std::exception& err) {
1153             print_metadata();
1154         }
1155
1156     }
1157
1157     print_metadata();
1158
1159     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1160
1161         try {
1162             for (auto& future : futures)
1163                 future.get();
1164
1165             hpx::wait_all(futures);
1166
1167         }
1168         catch (std::exception& err) {
1169             print_metadata();
1170         }
1171
1172     }
1173
1173     print_metadata();
1174
1175     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1176
1177         try {
1178             for (auto& future : futures)
1179                 future.get();
1180
1181             hpx::wait_all(futures);
1182
1183         }
1184         catch (std::exception& err) {
1185             print_metadata();
1186         }
1187
1188     }
1189
1189     print_metadata();
1190
1191     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1192
1193         try {
1194             for (auto& future : futures)
1195                 future.get();
1196
1197             hpx::wait_all(futures);
1198
1199         }
1200         catch (std::exception& err) {
1201             print_metadata();
1202         }
1203
1204     }
1205
1205     print_metadata();
1206
1207     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1208
1209         try {
1210             for (auto& future : futures)
1211                 future.get();
1212
1213             hpx::wait_all(futures);
1214
1215         }
1216         catch (std::exception& err) {
1217             print_metadata();
1218         }
1219
1220     }
1221
1221     print_metadata();
1222
1223     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1224
1225         try {
1226             for (auto& future : futures)
1227                 future.get();
1228
1229             hpx::wait_all(futures);
1230
1231         }
1232         catch (std::exception& err) {
1233             print_metadata();
1234         }
1235
1236     }
1237
1237     print_metadata();
1238
1239     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1240
1241         try {
1242             for (auto& future : futures)
1243                 future.get();
1244
1245             hpx::wait_all(futures);
1246
1247         }
1248         catch (std::exception& err) {
1249             print_metadata();
1250         }
1251
1252     }
1253
1253     print_metadata();
1254
1255     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1256
1257         try {
1258             for (auto& future : futures)
1259                 future.get();
1260
1261             hpx::wait_all(futures);
1262
1263         }
1264         catch (std::exception& err) {
1265             print_metadata();
1266         }
1267
1268     }
1269
1269     print_metadata();
1270
1271     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1272
1273         try {
1274             for (auto& future : futures)
1275                 future.get();
1276
1277             hpx::wait_all(futures);
1278
1279         }
1280         catch (std::exception& err) {
1281             print_metadata();
1282         }
1283
1284     }
1285
1285     print_metadata();
1286
1287     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1288
1289         try {
1290             for (auto& future : futures)
1291                 future.get();
1292
1293             hpx::wait_all(futures);
1294
1295         }
1296         catch (std::exception& err) {
1297             print_metadata();
1298         }
1299
1300     }
1301
1301     print_metadata();
1302
1303     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1304
1305         try {
1306             for (auto& future : futures)
1307                 future.get();
1308
1309             hpx::wait_all(futures);
1310
1311         }
1312         catch (std::exception& err) {
1313             print_metadata();
1314         }
1315
1316     }
1317
1317     print_metadata();
1318
1319     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1320
1321         try {
1322             for (auto& future : futures)
1323                 future.get();
1324
1325             hpx::wait_all(futures);
1326
1327         }
1328         catch (std::exception& err) {
1329             print_metadata();
1330         }
1331
1332     }
1333
1333     print_metadata();
1334
1335     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1336
1337         try {
1338             for (auto& future : futures)
1339                 future.get();
1340
1341             hpx::wait_all(futures);
1342
1343         }
1344         catch (std::exception& err) {
1345             print_metadata();
1346         }
1347
1348     }
1349
1349     print_metadata();
1350
1351     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1352
1353         try {
1354             for (auto& future : futures)
1355                 future.get();
1356
1357             hpx::wait_all(futures);
1358
1359         }
1360         catch (std::exception& err) {
1361             print_metadata();
1362         }
1363
1364     }
1365
1365     print_metadata();
1366
1367     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1368
1369         try {
1370             for (auto& future : futures)
1371                 future.get();
1372
1373             hpx::wait_all(futures);
1374
1375         }
1376         catch (std::exception& err) {
1377             print_metadata();
1378         }
1379
1380     }
1381
1381     print_metadata();
1382
1383     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1384
1385         try {
1386             for (auto& future : futures)
1387                 future.get();
1388
1389             hpx::wait_all(futures);
1390
1391         }
1392         catch (std::exception& err) {
1393             print_metadata();
1394         }
1395
1396     }
1397
1397     print_metadata();
1398
1399     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1400
1401         try {
1402             for (auto& future : futures)
1403                 future.get();
1404
1405             hpx::wait_all(futures);
1406
1407         }
1408         catch (std::exception& err) {
1409             print_metadata();
1410         }
1411
1412     }
1413
1413     print_metadata();
1414
1415     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1416
1417         try {
1418             for (auto& future : futures)
1419                 future.get();
1420
1421             hpx::wait_all(futures);
1422
1423         }
1424         catch (std::exception& err) {
1425             print_metadata();
1426         }
1427
1428     }
1429
1429     print_metadata();
1430
1431     @@ +188,9 +188,10 @@ void StdThreadQmcClusterSolver::integrate() {
1432
1433         try {
1434             for (auto& future : futures)
1435                 future.get();
1436
1437             hpx::wait_all(futures);
1438
1439         }
1440         catch (std::exception& err) {
1441             print_metadata();
1442         }
14
```

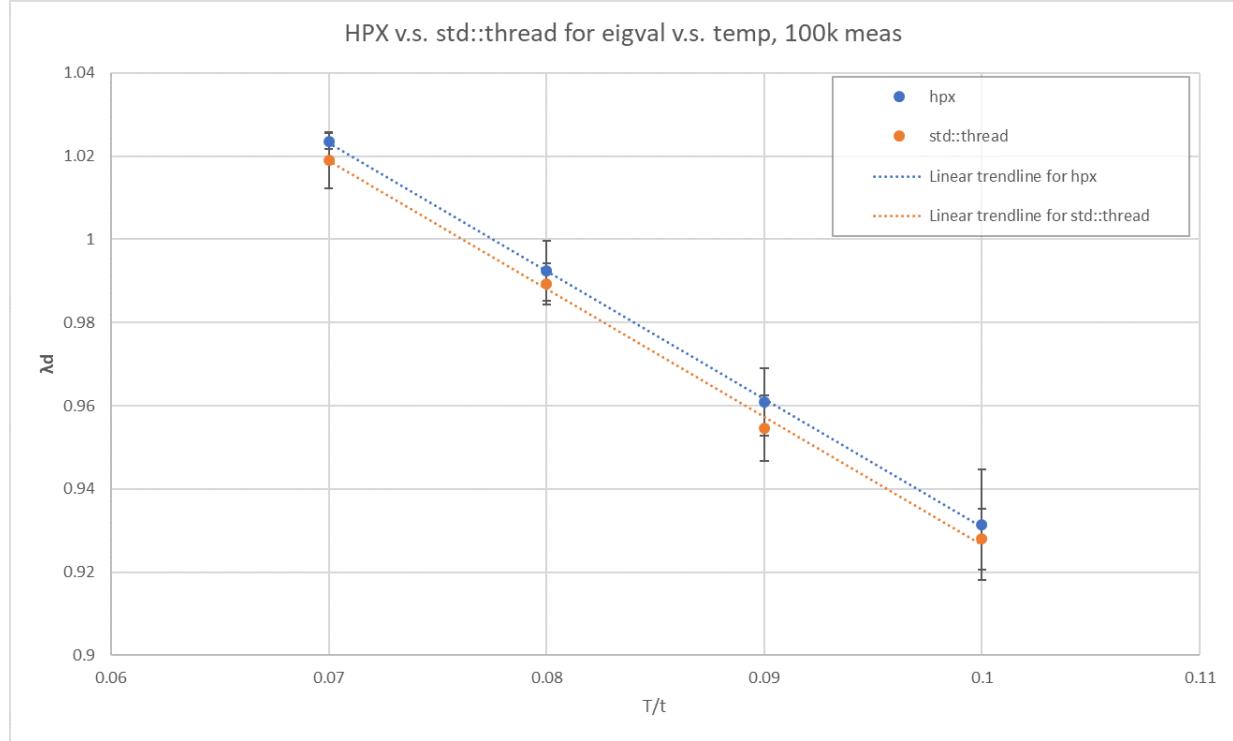


Figure 2.5. Compare the computation accuracy of HPX-enabled DCA++ and original DCA++

We measure the DCA++ run-time for custom-made thread-pool and HPX thread-pool for 5 times, respectively. Figure 2.6 shows that HPX-enabled DCA++ runs 3.6s faster than the custom-made thread-pool version. This means that HPX-enabled DCA++ is just slightly 1% faster than the custom-made thread-pool version in DCA++ and we consider this 1% speedup is inside the error margin.

The reason for this can be explained by DCA++’s threading configuration both from Summit hardware perspective and context switch perspective. Further explanation about the Summit hardware configuration can be found in the next paragraph and Figure 2.7. Here we analyze the context switch part. DCA++ originally does not use many threads (7 threads) to parallelize tasks, and basically no kernel context switch happens in the code. When we switch the threading mechanism to HPX light-weight thread-pool, we have not changed any underlying tasks ordering or code structure. So far, we simply replace standard threads to HPX light-weight user-level threading, the benefit of faster context

switch among HPX threads does not gain us much speedup. In the future, we plan to have finer size of computation tasks and oversubscribe tasks than the core capability in HPX-enabled DCA++ version, from there we will see more speedup and performance gain.

### Compare DCA++ execution time between custom-made threadpool and hpx threadpool

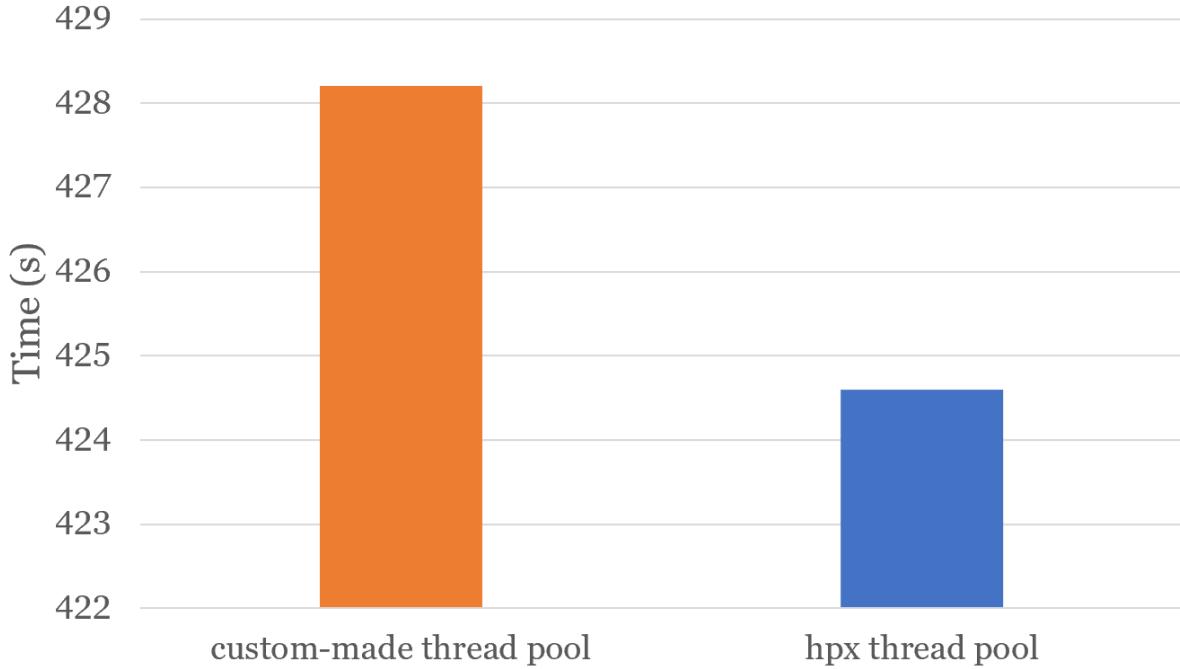


Figure 2.6. Compare DCA++ execution time between custom-made threadpool and hpx threadpool (threads=7)

We compare the DCA++ run-time among different number of HPX threads shown in Figure 2.7. The optimal run-time is obtained when number of HPX threads is set to 7. Similar behaviors are also found in DCA++ that 7 threads provides the optimal run-time results. This result is largely due to the fact that the design of DCA++ software is tightly customized to Summit hardware resources. On each compute node of Summit, DCA++ runs 6 MPI ranks and each rank has 1 GPU and 7 CPU cores. This method is an even division of 42 CPU cores on a node, as well as an optimal solution to utilize GPU resource on a node when no communication between GPUs happens. If each physical CPU core only runs 1 kernel thread, then no expensive context switch will occur. When setting `hpx:threads=7`, we tell HPX to use 7 worker threads in total and use only one worker thread

per CPU core.

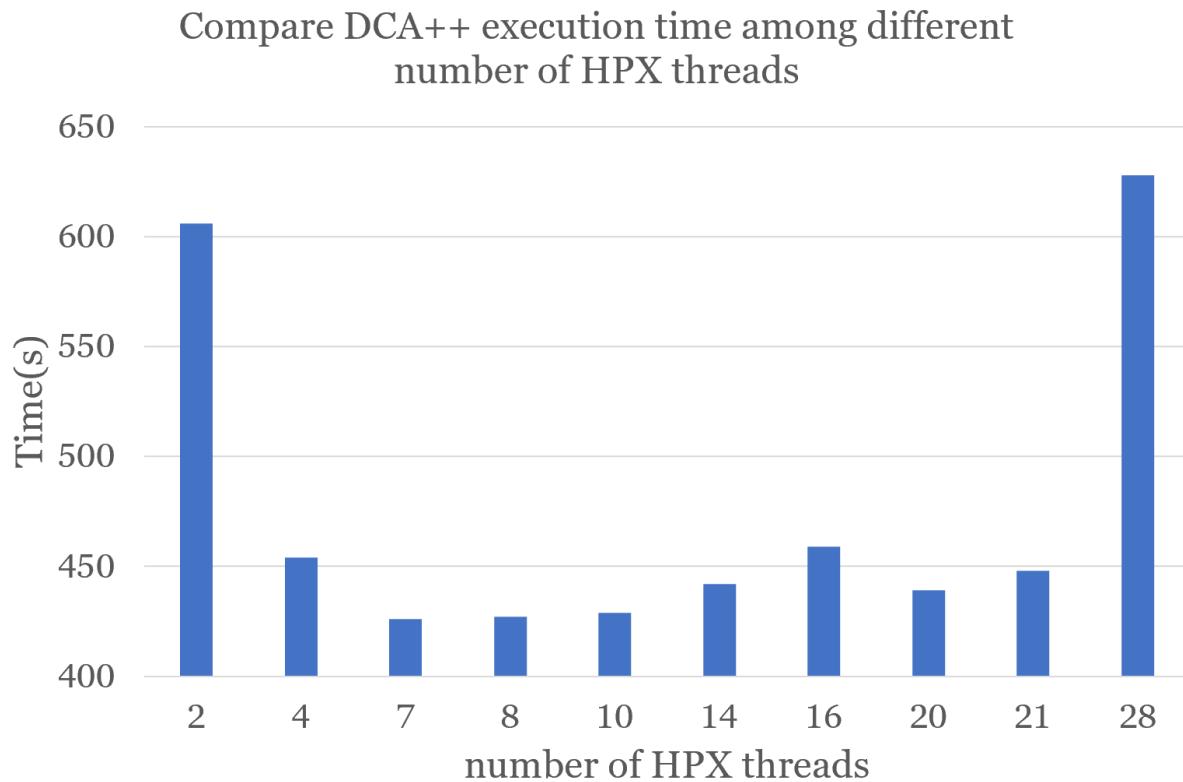


Figure 2.7. Compare DCA++ execution time among different number of HPX threads

## Chapter 3. GPUDirect

### 3.1. Memory Bound Issue and Solution

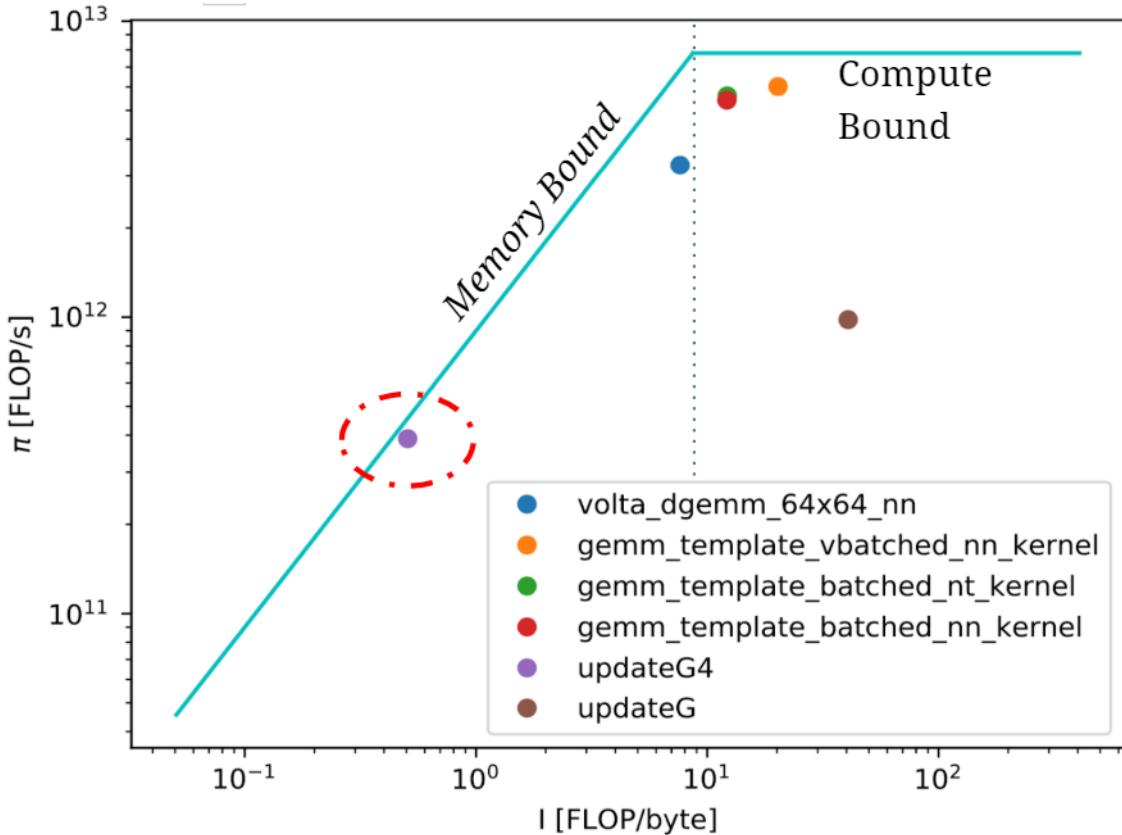


Figure 3.1. Roofline plot of a NVIDIA V100 GPU running DCA++ at production level on Summit. Figure adapted from [1]

Previous work [1] indicated that DCA++ is facing memory bound issue caused by the size of G4 matrix (two particle function). Figure 3.1 shows Performance of the most FLOP-intensive kernels compared against the roofline model of an NVIDIA V100 GPU on Summit. Each NVIDIA V100 GPU on Summit has 16GB High Bandwidth Memory 2, and the size of G4 matrix is estimated to take up about 12 GB HBM2 space in each GPU shown in a red dash circle in Figure 3.1. If a larger input size was provided, then each GPU might not be able to process and store the entire G4 matrix thereby we are limited to device memory size.

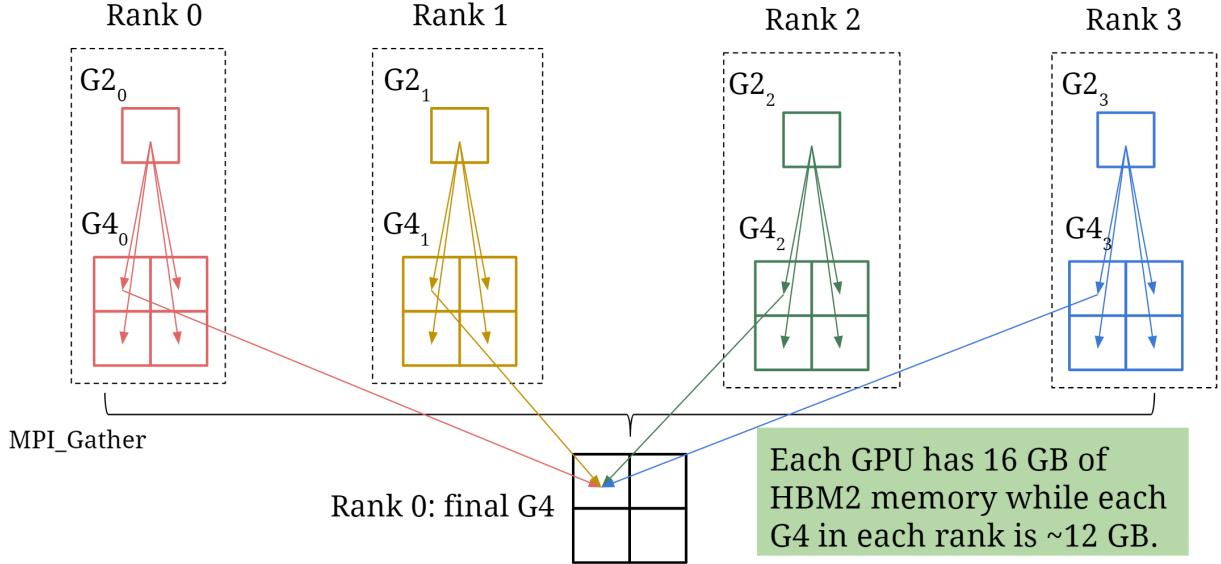


Figure 3.2. Memory bound issue with G4 matrix

We will further visualize the procedure on how we have memory bound issue in DCA++.

Currently, each MPI rank (with 1 GPU resource) keeps a private and full copy of G4 matrix (yet complete) and all MPI ranks do sum reduction at the end to form final G4 matrix on root rank via `mpi_all_reduce` shown as in Figure 3.2. Each GPU resource has its own private copy of G4, which is a size of 12 GB, and therefore hits memory bound.

### 3.2. GPUDirect on Summit

It is a necessary step to explore techniques and develop algorithm for a distributed version of G4 matrix in preparation of larger science runs when G4 matrix cannot fit on a single GPU. To alleviate the memory bound challenge in [5] on the V100s, we explore the NVIDIA NVLink interconnects on Summit nodes to take advantage of the high bandwidth to address memory bound issue.

NVIDIA's GPUDirect peer-to-peer communication enables peer-to-peer memory access, transfers and synchronization between two GPUs [14]. For example, with GPUDirect  $GPU_0$  can read and write remote  $GPU_1$  memory (load/store). Also, with GPUDirect, `cudaMemcpy()` initiates Direct Memory Access (DMA) copy from  $GPU_0$  memory to  $GPU_1$  memory.

### 3.3. MPI Ping-pong Benchmark

To test the bandwidth advantage of using GPUDirect on Summit, we develop a mini-application for comparing GPUDirect communication method and MPI GPU to Remote GPU method. A MPI ping pong program is used in this mini application. In this program, processes use MPI\_Send and MPI\_Recv to continually bounce messages off of each other until they decide to stop.

Figure 3.3 visualizes MPI GPU to Remote GPU method and Listing 3.1 shows the pseudo code. For the process of transferring an device allocated array from  $GPU_0$  to  $GPU_1$ , it goes through several processes: allocates memory on the  $GPU_0$ , copies data from  $GPU_0$  to the  $host_0$  in the same rank, transfers data from  $host_0$  to remote  $host_1$  through MPI\_send and MPI\_receive, and finally copies data from  $host_1$  to  $GPU_1$ . The implementation of this method can be found in [https://github.com/weilewei/Ring\\_example\\_MPI\\_CUDA/blob/master/up\\_n\\_down.cpp](https://github.com/weilewei/Ring_example_MPI_CUDA/blob/master/up_n_down.cpp).

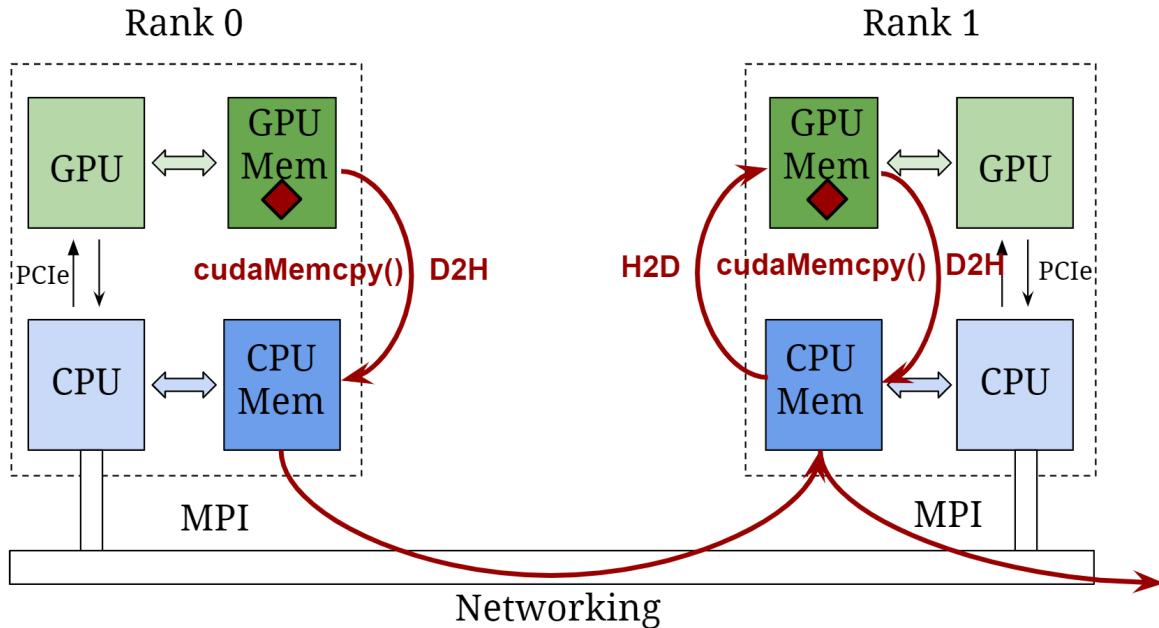


Figure 3.3. GPU data transfer between GPUs using traditional MPI GPU to Remote GPU method

Listing 3.1. MPI GPU to Remote GPU

```

char* s_d_array; char* r_d_array; char* s_h_array; char* r_h_array;

for (long long size : std::vector<long long>{1, 2, 10, 100, 1000, 1000000, 30000000,
    100000000, 1000000000}) {
    alloc_d_char(size, &s_d_array);
    init_d(size, s_d_array, 'a');
    alloc_d_char(size, &r_d_array);

    s_h_array = (char*) malloc(size * sizeof(char));
    r_h_array = (char*) malloc(size * sizeof(char));

    bool ping = 0;
    startTimer();
    for (int i = 0; i < times; ++i) {
        if (rank == ping)
        {
            cudaMemcpy(s_h_array, s_d_array, size, cudaMemcpyDeviceToHost);
            MPI_CHECK(MPI_Send(s_h_array, size, MPI_CHAR, !ping, 0, MPI_COMM_WORLD));
        }
        else
        {
            MPI_CHECK(MPI_Recv(r_h_array, size, MPI_CHAR, ping, 0, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE));
            cudaMemcpy(r_d_array, r_h_array, size, cudaMemcpyHostToDevice);
        }
        ping = !ping;
    }
    auto time = endTimer() / times;
}

```

---

```

if(rank == 0) {

out << size << "\t" << time << "\t" << size / time * 1e-9 << "\n";

std::cout << size << "\t" << time << "\t" << size / time * 1e-9 << "\n";

}

```

---

Figure 3.4 visualizes GPUDirect method and Listing 3.2 shows the pseudo code. For the process of transferring an device allocated array from  $GPU_0$  to  $GPU_1$ , it goes through several processes: allocates memory on the  $GPU_0$ , transfers data from  $GPU_0$  to remote  $GPU_1$ . CUDA-aware MPI is used in this case. The implementation of this method can be found in [https://github.com/weilewei/Ring\\_example\\_MPI\\_CUDA/blob/master/gpuDirect.cpp](https://github.com/weilewei/Ring_example_MPI_CUDA/blob/master/gpuDirect.cpp).

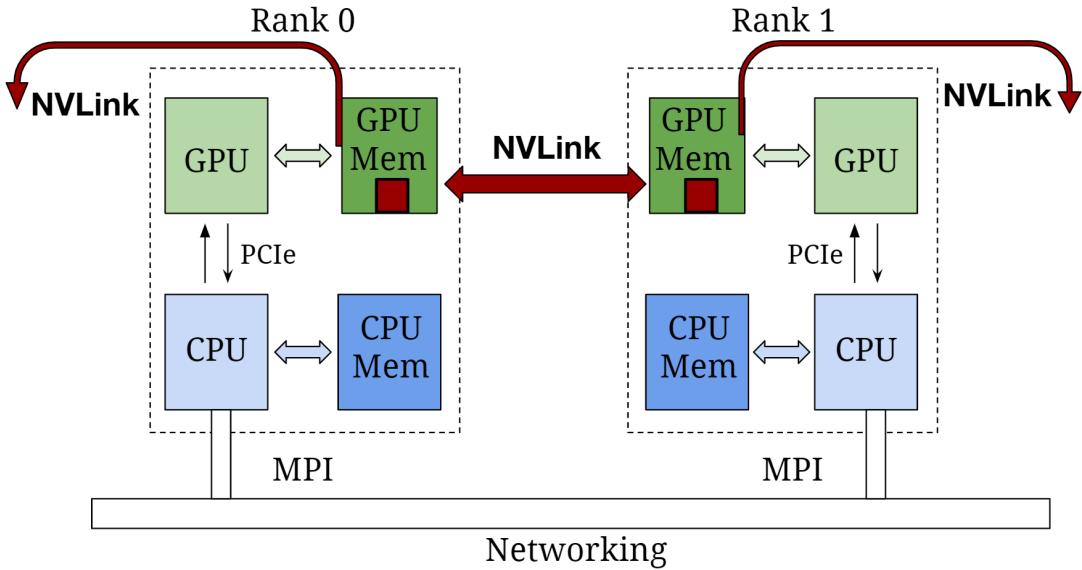


Figure 3.4. GPU data transfer between GPUs using GPUDirect

---

Listing 3.2. GPUDirect peer to peer

---

```

char* s_array; char* r_array;

for (long long size : std::vector<long long>{1, 2, 10, 100, 1000, 1000000, 30000000,
1000000000, 10000000000} {

```

```

alloc_d_char(size, &s_array);

init_d(size, s_array, 'a');

alloc_d_char(size, &r_array);

bool ping = 0;

startTimer();

for (int i = 0; i < times; ++i) {

if (rank == ping)

    MPI_CHECK(MPI_Send(s_array, size, MPI_CHAR, !ping, 0, MPI_COMM_WORLD));

else

    MPI_CHECK(MPI_Recv(r_array, size, MPI_CHAR, ping, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE));

ping = !ping;

}

auto time = endTimer() / times;

if(rank == 0) {

out << size << "\t" << time << "\t" << size / time * 1e-9 << "\n";
std::cout << size << "\t" << time << "\t" << size / time * 1e-9 << "\n";
}

}

```

---

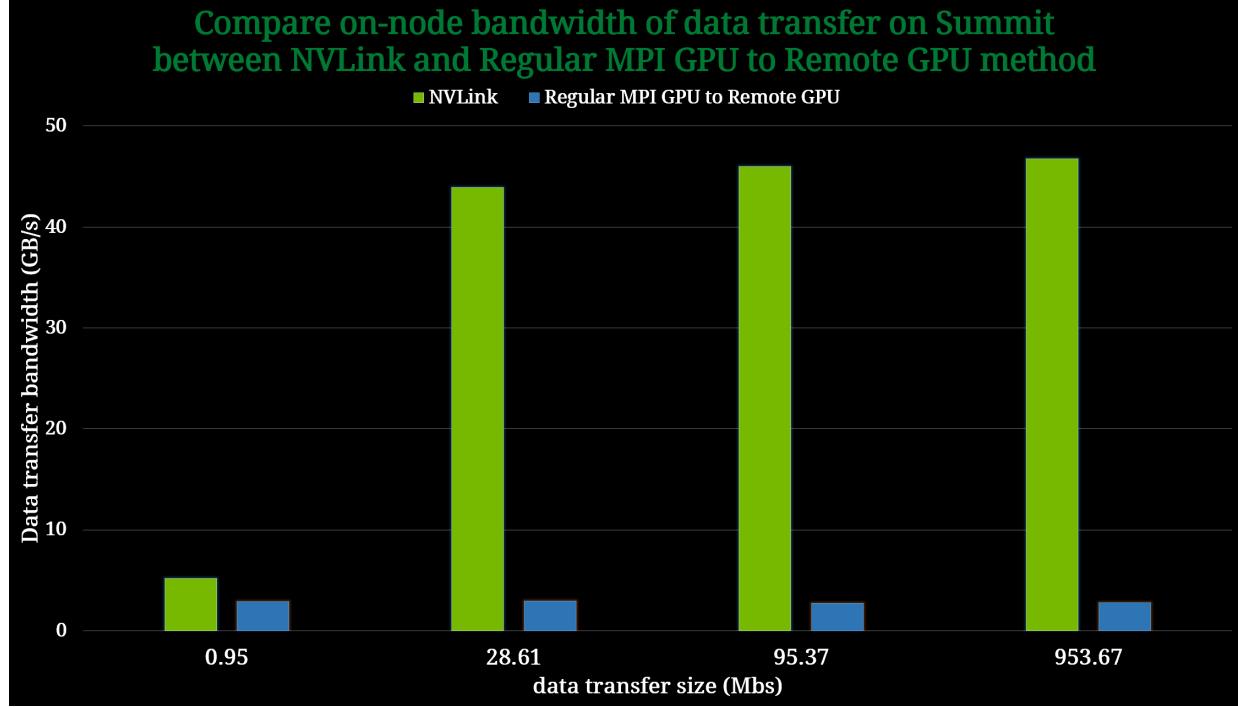


Figure 3.5. On-node bandwidth comparisons: DCA++ mini-application using NVLink and Up-and-down method)

Figure 3.5 shows up to 17x speedup observed by DCA++ Lite mini-application using GPU Direct (NVLink) over regular communication (MPI GPU to remote GPU). One of the main reasons resulting in the bandwidth difference is that too many data copy operations occurred in regular communication method: data copy between  $GPU_0$  to  $CPU_0$ ,  $CPU_0$  to  $CPU_1$ , and  $CPU_1$  to  $GPU_1$ , which greatly slows down data transfer process.

Figure 3.6 shows workflow of distributed G4 with NVLink enabled. Since using NVLink in the data communication phase would greatly speedup the process, we plan to integrate NVLink into DCA++ code. In the original implementation shown in Figure 3.2, in each MPI rank, each G2 is performing product operation to generate G4, which is a private and full copy (yet complete) of final G4 matrix. Then, all MPI ranks do sum reduction at the end to form final and complete G4 matrix on root rank via `mpi_all_reduce`. With new implementation, each local G2 will also conduct product operation but only contribute to a small portion of G4. Then each locally generated G2 in different ranks will be sent to other different ranks, such that, each rank will see every G2. Every time when each rank

receives a G2, it will perform product operation to contribute to a small portion of its local G4. After every rank see every G2 and finish its update of its small portion of G4, now each rank owns a small but complete portion of G4. At the end, we will perform one more step as in the original implementation, all MPI ranks do sum reduction at the end to form final G4 matrix on root rank via `mpi_all_reduce`. Now, at the root rank, we will have a full and complete final G4 matrix, the result we want.

The following example can help further illustrate the new implementation. Just like what we have in Figure 3.6. Let's assume we would like to have a final G4 matrix of size 2x2. In each MPI rank, instead of computing (product operation) a full but not complete G4 in previous implementation, now each rank only computing a small portion of G4. For example, in rank 0, it only computes  $G4(0,0)$ , which is one forth of final G4 size; in rank 1, it only computes  $G4_{(0,1)}$  and so on. Then we will have a "travel phase", meaning that every G2 will be traveling and exposing in each rank to update a small portion of G4 in that local rank. For example,  $G2_1$  will be sent to rank 0, and do product operation to update  $G4_{(0,0)}$ , so does the same to  $G2_2$ ,  $G2_3$ . In other words, rank 0 will receive 4 different G2s and complete the computation or update of  $G4_{(0,0)}$ . After the "travel phase", each G2 has been traveled across the MPI world to finish their product operation in each MPI rank, in other words, each MPI rank now has a complete update of its local G4, a small portion of final G4 array. At the end, we will perform one more step as in the previous implementation, all MPI ranks perform sum reduction at the end to form final G4 matrix on root rank via `mpi_all_reduce`. Now, in rank 0, we have a final and complete G4 array of size 4x4.

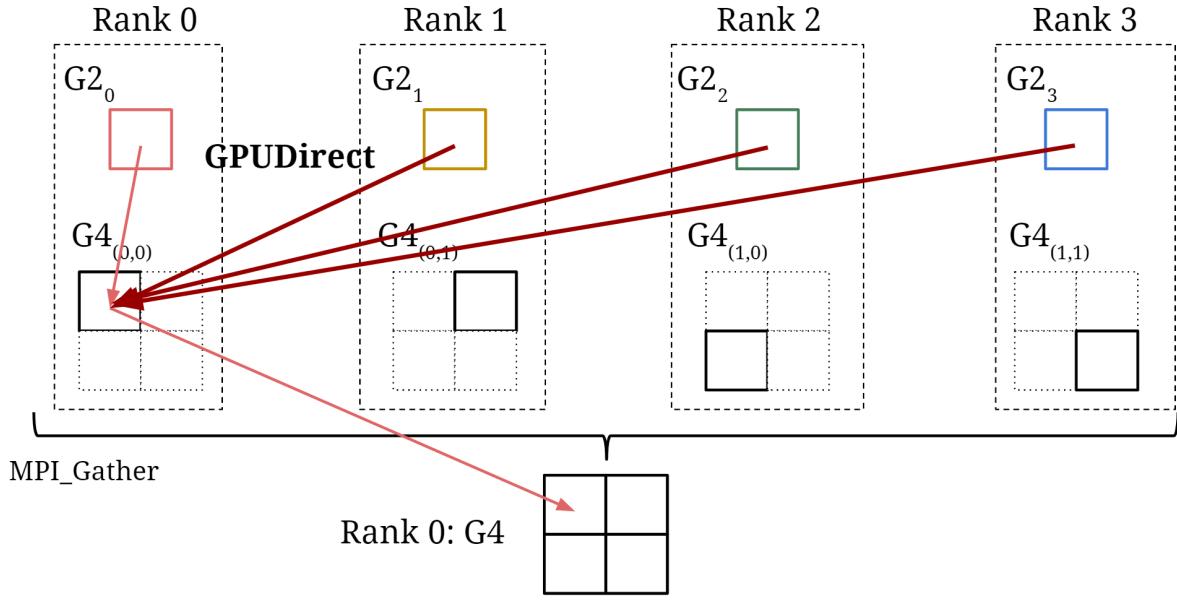


Figure 3.6. Distributed G4 with NVLink enabled

### 3.4. Bandwidth Measurement on NVLink on Summit

In order to use NVLink on Summit at scale, such as transferring data across multiple nodes and racks/cabinets, bandwidth measurement on NVLink with this regard is needed. Figure 3.7 shows the comparison results on NVLink bandwidth across nodes and racks/cabinets. We are focused on when the size of data transfer is about 28.61 Mb as the G2 array size in DCA++ is about 25 MB. The result shows that on-node GPUDirect has highest bandwidth than rest of the scenarios. On-node GPUDirect is about 2.2 speedup than two-node GPUDirect in same rack and 2.36 speedup than two-node in different racks.

We further submitted more job runs and observed that, for an array of size 953 Mb, 8 out of 10 runs have similar speed ( 23 Gb/s), one run is 20 Gb/s, another one is as slow as 16 Gb/s. For the slowest run result, two racks are just several aisles apart in the room, visualized in 3.9 (link: <https://jobstepviewer.olcf.ornl.gov/summit/952157-3>). The 23 Gb/s one also happens when racks are located on two sides of the room, visualized in 3.8 (link: <https://jobstepviewer.olcf.ornl.gov/summit/951401-3>). It seems the distance between racks (i.e. how racks are physically located in the facility room) would not impact

the bandwidth.

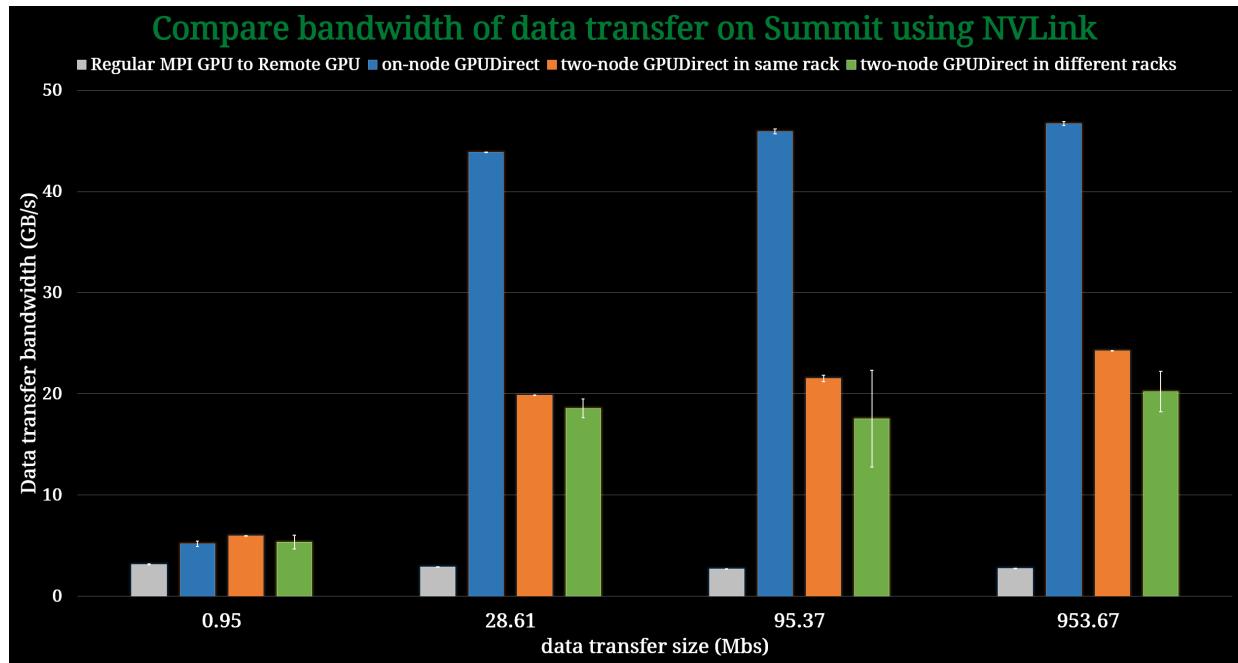


Figure 3.7. Compare bandwidth of data transfer on Summit using NVLink

Table 3.1. Compare NVLink bandwidth speedup on Summit

Size of data transferred (Mb)	0.95	28.61	95.37	953.67
Speedup (GB/s) for on-node / off-node	0.87	2.21	2.14	1.93

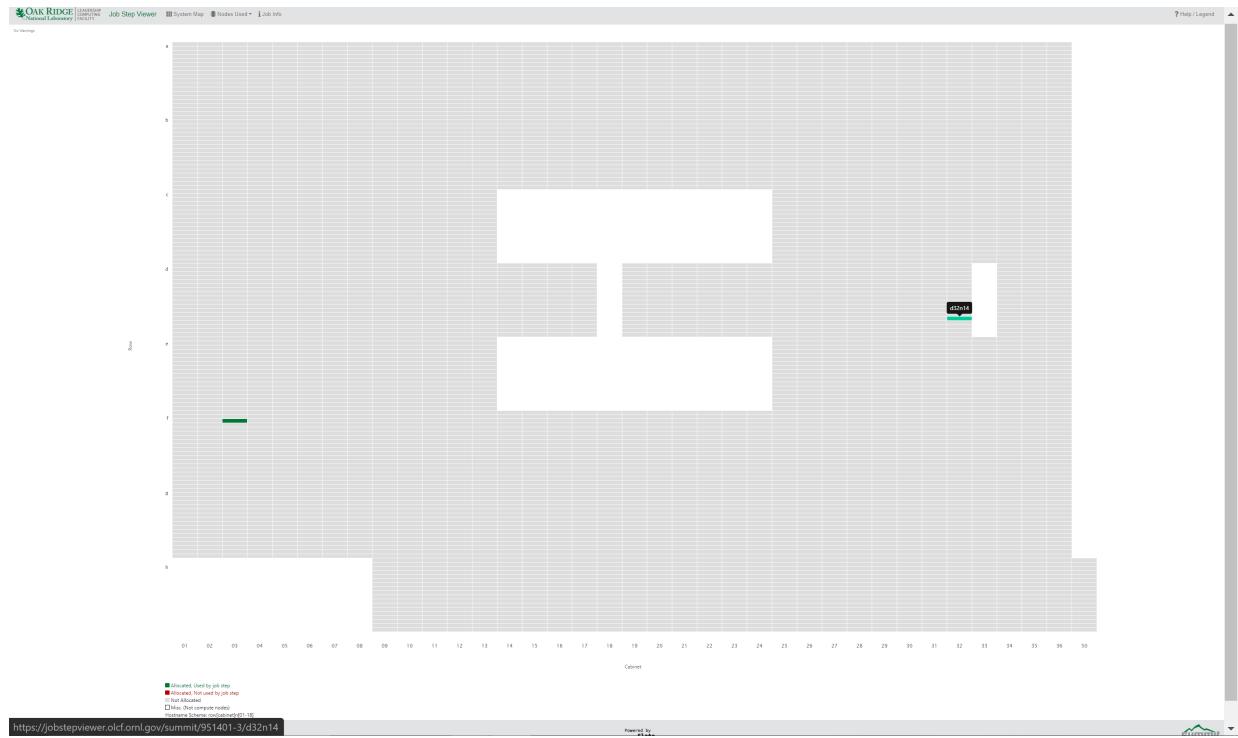


Figure 3.8. Two nodes are located in two different sides of the facility room

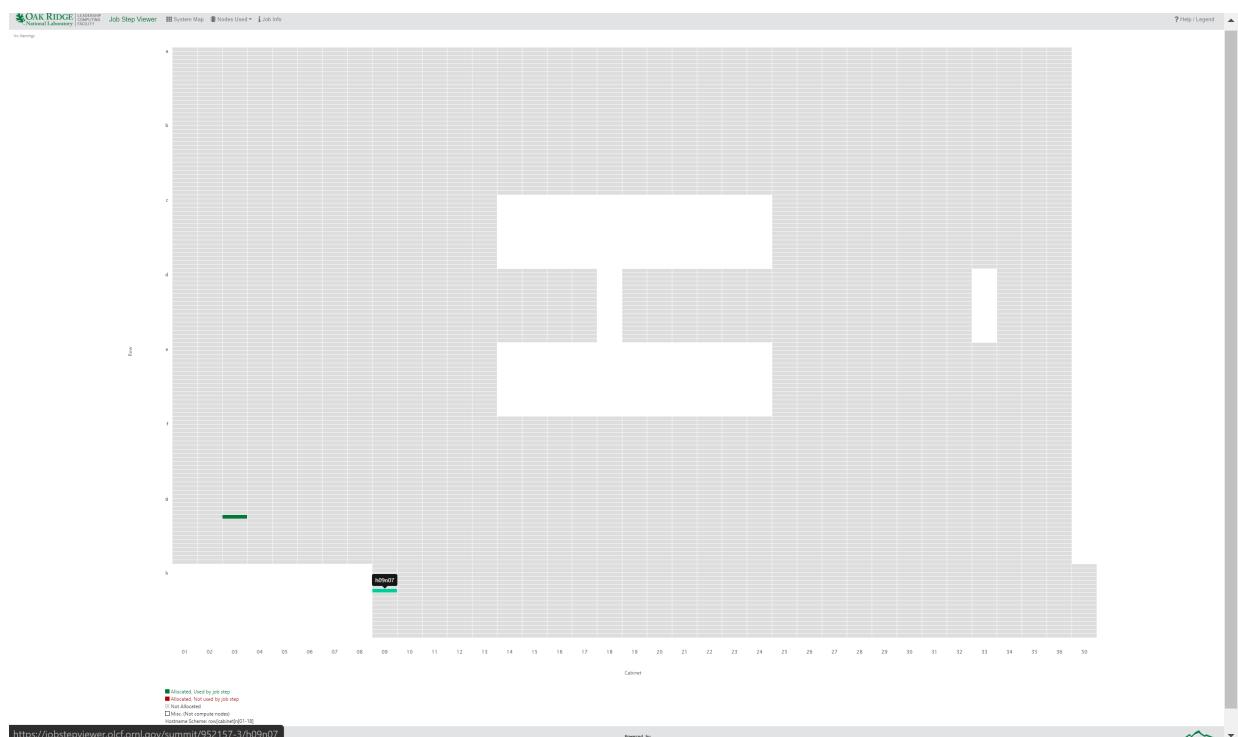


Figure 3.9. Two nodes are located close to each other in the facility room

The reason is Summit has a non-blocking, three-level fat-tree [15] (i.e., folded Clos)

topology. This means is that there are three levels of switches: first-, second-, and third-level for simplicity's sake. On Summit, we call these Top Of Rack (TOR) switches, Leaf switches, and Spine switches, respectively. The first level switches connect to 18 nodes in the rack. Each client has two ports and each port connects to a different first-level switch. Clients connected to the same switch do not experience congestion (overloaded links in the middle of the network), but they can experience endpoint contention (e.g., multiple hosts sending to one host). Traffic between two hosts in the same rack using the same port should see almost no variability in performance because there is one switch hop and the switch is non-blocking.

Each first level switch also has 18 links for connecting to second-level switches. Groups of first-level switches will connect to 18 different second-level switches. Traffic between clients within a group that are in separate racks (i.e. connected to different first-level switches in different racks) or traffic between two clients in the same but connected to opposite client ports (i.e., connected to different first-level switches albeit in the same rack) has to traverse three switch hops (first-level, second-level, first-level). In this case, traffic flows can experience both congestion and endpoint contention. This leads to variability. On Summit, nodes are grouped in sets of 324 (18 nodes per rack x 18 racks per group).

Lastly, each second-level switch also connects to 18 third-level switches. Again, they are grouped in sets of 18 second-level switches. Any traffic that cannot be serviced by a first- or second-level switch must traverse five switch hops and is susceptible to more opportunities to encounter congestion or endpoint contention leading to the highest variability.

Summit's network uses adaptive routing to change the path if a flow detects congestion. This can also lead to variation if it happens mid-flow on some but not all tests.

### 3.5. Pipeline Ring Algorithm

In order to address memory bound issue in DCA++, we design a pipeline ring algorithm to process MPI communication. This algorithm assumes that there are  $P$  ranks, and each rank locally generates  $p$  number of  $G2$  array, so each rank will see all  $P$  copies of  $G2$  after

the pipeline ring algorithm. This algorithm will work well if every MPI rank has similar time cost for generating G2 and updating G4, and each MPI rank generates the same number of G2 arrays.

The pseudo code show in 3.3 has a for-loop assuming each rank generate P number of G2. In this algorithm, each MPI rank takes a newly generated G2 array to update G4 array, makes a copy of this G2 array to an outgoing array that will be asynchronously sent to its right neighbor later, receives a copy of G2 array from its left neighbor that will be copied into local G2. Overall, this algorithm will be conducted in P steps, and in each time-step, each MPI rank will send out one G2 array, use local G2 to update G4 and send out one G2 array.

In our first attempt to implement the pipeline ring algorithm, we only store one local copy of G2, one buffer copy of outgoing copy of G2, and one buffer copy of incoming copy of G2. In the future, we are considering storing all the P copies of G2s (or some of G2s), which is just an variant of the lock step algorithm. One possible limitation of storing P copies of G2s is increasing device memory as value of P could be hundreds of thousands so large that it will trigger another memory bound issue. Also, we are considering attaching task continuation and adding more overlap between communication and computation to fully utilize hardware resources. Which version is more efficient may also depend on the relative cost of generating G2, cost of updating G4, and cost of MPI communication.

---

Listing 3.3. pipeline ring algorithm

---

```
#define MOD(x,n) ((x) % (n))

left_neighbor = MOD((myrank-1 + mpi_size), mpi_size);
right_neighbor = MOD((myrank+1 + mpi_size), mpi_size);

for(iter=0; iter < niter; iter++)
{
```

```

// each rank generate same number of G2

generateG2(G2);

update_local_G4(G2);

// get ready for send

sendbuf_G2 = G2; // copy into buffer

send_tag = 1 + myrank;

send_tag = 1 + MOD(send_tag-1, MPI_TAG_UB); // MPI_TAG_UB is largest tag value

for(icount=0; icount < (mpi_size-1); icount++)
{
    // encode the originator rank in the message tag as tag = 1 + originator_irank

    originator_irank = MOD((myrank-1)-icount + 2*mpi_size), mpi_size);

    recv_tag = 1 + originator_irank;

    recv_tag = 1 + MOD(recv_tag-1, MPI_TAG_UB); // 1 <= tag <= MPI_TAG_UB

    MPI_Irecv(recvbuf_G2, source=left_neighbor, tag = recv_tag, &recv_request);

    MPI_Isend(sendbuf_G2, dest=right_neighbor, tag=send_tag, &send_request);

    mpi_wait(recv_request); // wait for G2 to arrive

    G2 = recvbuf_G2; // copy from buffer

    update_local_G4(G2);

    mpi_wait(send_request); // wait for sendbuf_G2 to be available again

    // get ready for send

    sendbuf_G2 = G2;

    send_tag = recv_tag;

}; // end for icount

}; // end for iter

```



## Chapter 4. Conclusion and Future Work

In this project report, we introduced Dynamic Clustering Algorithm (DCA++) and analyzed how we use High-performance ParalleX (HPX) and NVLink to build high-level parallel abstraction layers with a goal of improving performance of DCA++ software.

For threading abstraction layer, we added a threading abstraction layer using HPX user-level light-weight threading model such that users can switch between custom-made thread pool and hpx thread pool via compile time input without changing too much codes in DCA++. We also presented the results of HPX implementation: HPX-enabled DCA++ 1) produced same results to custom thread pool in DCA++ and 2) has slightly faster runtime performance than the custom-made thread pool option in DCA++, but we consider the 1% speedup is within the error margin. In the future, we will add more tasks continuation methods to better utilize hardware resources, such as, facilities in HPX: `dataflow`, `then`, ect.

For NVLink method, we evaluated NVLink bandwidth on Summit, using ping-pong algorithm, comparing regular MPI GPU to remote GPU method, and also analyzing NVLink bandwidth under different scenarios (on-node, two-node in same/different racks on Summit). We concluded that using NVLink would speedup to our min-application. We also introduced memory bound in DCA++ and detailed how we integrate pipelined ring algorithm and NVLink to address the memory bound issue. In the future, we plan to implement pipelined ring algorithm to distribute G4 array and profile our implementation.

Also, we plan to integrate GPUDirect into HPX so we can construct task dependency and attach more task continuations by overlapping network communication and computation to fully utilize hardware resources and improve code performance. For example, GPUDirect function call is attached to MPI API (i.e. `MPI_Isend`, `MPI_Irecv`), and we can wrap return value of MPI call into `hpx::future` and then attach continuation with that future. In this way, once the `MPI_Irecv`, for instance, has finished, it will automatically continue to execute next task, if available.

## References

- [1] Giovanni Balduzzi, Arghya Chatterjee, Ying Wai Li, Peter W Doak, Urs Haehner, Ed F D’Azevedo, Thomas A Maier, and Thomas Schulthess. Accelerating dca++ (dynamical cluster approximation) scientific application on the summit supercomputer. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 433–444. IEEE, 2019.
- [2] Gregor Daiß, Parsa Amini, John Biddiscombe, Patrick Diehl, Juhan Frank, Kevin Huck, Hartmut Kaiser, Dominic Marcello, David Pfander, and Dirk Pfüger. From piz daint to the stars: simulation of stellar mergers using high-level abstractions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–37, 2019.
- [3] Thomas Heller, Patrick Diehl, Zachary Byerly, John Biddiscombe, and Hartmut Kaiser. Hpx—an open source c++ standard library for parallelism and concurrency. *Proceedings of OpenSuCo*, page 5, 2017.
- [4] Urs R Hähner, Gonzalo Alvarez, Thomas A Maier, Raffaele Solcà, Peter Staar, Michael S Summers, and Thomas C Schulthess. Dca++: A software framework to solve correlated electron problems with modern quantum cluster methods. *Computer Physics Communications*, 246:106709, 2020.
- [5] Urs R Hähner, Giovanni Balduzzi, Peter W Doak, Thomas A Maier, Raffaele Solcà, and Thomas C Schulthess. Dca++ project: Sustainable and scalable development of a high-performance research code. In *Journal of Physics: Conference Series*, volume 1290, page 012017. IOP Publishing, 2019.
- [6] Thomas Heller, Bryce Adelstein Lelbach, Kevin A Huck, John Biddiscombe, Patricia Grubel, Alice E Koniges, Matthias Kretz, Dominic Marcello, David Pfander, Adrian Serio, et al. Harnessing billions of tasks for a scalable portable hydrodynamic simulation of the merger of two stars. *The International Journal of High Performance Computing Applications*, 33(4):699–715, 2019.
- [7] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. Paralex an advanced parallel execution model for scaling-impaired applications. In *2009 International Conference on Parallel Processing Workshops*, pages 394–401. IEEE, 2009.
- [8] Bibek Wagle, Samuel Kellar, Adrian Serio, and Hartmut Kaiser. Methodology for adaptive active message coalescing in task based runtime systems. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1133–1140. IEEE, 2018.
- [9] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 2015.
- [10] Tianyi Zhang, Shahrzad Shirzad, Patrick Diehl, R Tohid, Weile Wei, and Hartmut Kaiser. An introduction to hpxmp: A modern openmp implementation leveraging hpx, an asynchronous many-task system. In *Proceedings of the International Workshop on OpenCL*, pages 1–10, 2019.

- [11] Thomas Heller. Extending the c++ asynchronous programming model with the hpx runtime system for distributed memory computing.
- [12] Michael Wong, Hartmut Kaiser, and Thomas Heller. Towards massive parallelism (aka heterogeneous devices/accelerator/gpgpu) support in c++ with hpx. 2015.
- [13] Patrick Diehl, Madhavan Seshadri, Thomas Heller, and Hartmut Kaiser. Integration of cuda processing within the c++ library for parallelism and concurrency (hpx). In *2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, pages 19–28. IEEE, 2018.
- [14] NVDIA. Gpudirect. <https://developer.nvidia.com/gpudirect>.
- [15] Mellanox Technologies. Reaching the summit with infiniband: Mellanox interconnect accelerates world’s fastest hpc and artificial intelligence supercomputer at oak ridge national laboratory (ornl). [https://www.mellanox.com/related-docs/solutions/hpc/CS\\_ORNL\\_Summit\\_InfiniBand.pdf](https://www.mellanox.com/related-docs/solutions/hpc/CS_ORNL_Summit_InfiniBand.pdf).

## Vita

Weile Wei finished his undergraduate studies at Shandong Jianzhu University, Jinan, China and Griffith University, Gold Coast, Australia. In May 2018 he had opportunities to pursue graduate studies in Computer Science at Louisiana State University and work as a graduate software developer with STE||AR group at Center for Computation and Technology located at LSU. During his stay at LSU, he has completed several projects in the field of distributed machine learning, parallel computing and high performance computing. Also, he had software developer internships experience with National Center for Atmospheric Research for developing file system library, and with Oak Ridge National Lab for researching programming model for Dynamic Clustering Algorithm (DCA++). His researches has been presented in several international conference papers and posters.