CSPACER: A Reduced API Set Runtime for the Space Consistency Model

Khaled Z. Ibrahim
Computational Research Division, Lawrence Berkeley National Laboratory
Berkeley, CA 94720, USA
kzibrahim@lbl.gov

ABSTRACT

We present our design and implementation of a runtime for the Space Consistency model. The Space Consistency model is a generalized form of the full-empty bit synchronization for distributed memory programming, where a memory region is associated with a counter that determines its consistency and readiness for consumption. The model allows for efficient implementation of point-to-point data transfers and collective communication primitives as well. We present the interface design, implementation details, and performance results on Cray XC systems. Our runtime adopts a reduced API design to provide low-overhead initiation and processing of communication primitives, enable threaded execution of runtime functions, and provide efficient pipelining, thus improving the computation-communication overlap. We show the performance benefits of using this runtime both at the microbenchmark level and in application settings.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**; • **Theory of computation** → **Parallel computing models**.

KEYWORDS

Runtime Systems, Parallel Programming Models

ACM Reference Format:
Khaled Z. Ibrahim. 2021. CSPACER: A Reduced API Set Runtime for the Space Consistency Model. In *The International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2021), January 20–22, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3432261.3432272

1 INTRODUCTION

Efficient communication in HPC platforms is instrumental for performance and scalability. Achieving efficiency requires not only improvements in interconnect designs but also improvement in runtime designs. The multicore architectures were introduced a couple of decades ago as the most viable architecture to tackle the demise of Dennard's scaling. Single-core performance has not improved significantly since then, while the number of cores per chip has increased from few cores to tens of cores. In most architectures, the single-core performance is traded for increased core parallelism. The degree of leveraging such architectural shift has a striking contrast between the application layer and communication runtime layer. Application developers could easily leverage multi/many-core architectures using either data sharing parallelism models, such as OpenMP, or could use virtual splitting of computational resources using the process abstraction. At the runtime level, there were fewer attempts to leverage multicore architectures to improve communication libraries. A notable effort is the MT-MPI [31] work, which leverages OpenMP threads for internal MPI processing. The majority of MPI-related efforts are mainly focusing on techniques [4, 16] to reduce the overhead supporting concurrent thread access to the runtime layer. Except for progress threads, none of the efforts to leverage multicore architectures for accelerating internal MPI processing are part of production-quality implementations.

Languages and libraries supporting one-sided communication, such as UPC [9] and MPI RMA [15], provides the opportunity for decoupling synchronization from data transfer mechanisms. While they may provide low-overhead support for threaded access [22], they typically lack efficient support for collectives communication primitives, especially over large vectors of elements. Additionally, they do not leverage multicore architectures for internal processing.

Our work attempts to leverage multicore computational power for various runtime operations, especially collectives over a vector of elements. Heterogenous systems of many-core accelerated by GPUs, *e.g.* OLCF Summit, are becoming more prevalent in HPC designs. Having accelerators dedicated for computation could allow more cores to do complex runtime activities such as resource scheduling and communication activities. Ideally, we tap into these compute resources while preserving other performance techniques such as the ability to overlap computation with communication, pipeline activities, etc.

To achieve these goals, we find it necessary to revisit the interface between the application and the runtime. We adopt a reduced API design principle for constructing the CSPACER runtime[1], where we decompose complex runtime operations into simpler ones, including creation, planning, and initiation of a transfer. This study shows how such decomposition allows efficient processing and pipelining of transfer and introduces sparsified traffic patterns to the interconnect. We leverage the Space Consistency abstraction, a mechanism for performing producer-consumer relations in a distributed environment using one-sided semantic [20, 21]. We introduce an implementation of the Consistency Space model through

[1]We draw the analogy of the CSPACER runtime design principle with the reduced instruction set in ISA designs for microprocessors to improve the execution efficiency.

HPC Asia 2021, January 20–22, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8842-9/21/01.
https://doi.org/10.1145/3432261.3432272

a low-overhead runtime, called CSPACER[2]. The abstraction allows reasoning about the readiness of a memory region for consumption as a single unit, releasing the runtime from requiring to track individual transfers. Its relaxed semantics enables a wide range of runtime optimizations, including threaded-execution of communication primitives and lock-free concurrent injections from threaded regions, among others. CSPACER provides a mechanism to leverage lightweight many-core architectures and extend the use of one-sided semantic to collective operations. We provide common complex runtime operations as communication patterns that could easily be integrated into applications compute kernels to accelerate critical data movement routines. These skeletons include data reduction, irregular all-to-all exchange, one-sided `broadcast`, irregular `allgather`, etc. We support these patterns over teams, a subset of communicating ranks. We evaluate the model on Cray XC40 systems both at the microbenchmark level and in application settings.

The CSPACER runtime deliver up to 3.8× for `broadcast` compared with MPI, and up to 3.9× for `allreduce`. We show the performance advantage of using this runtime for matrix multiplication libraries and Lattice QCD simulations. The introduced runtime also interoperates efficiently with the MPI runtime, such that the majority of MPI code would require no change. Only hotspots within an application code will be the likely candidate to migrate to our proposed runtime. Therefore, CSPACER could be thought as a runtime accelerator to MPI-based applications. The CSPACER runtime and communication patterns are publicly available [32].

The rest of this paper is organized as follows. Section 2 introduces the space consistency abstraction for distributed memory programming. We introduce the proposed API in Section 3. Then we present the evaluation method in section 4. We present the performance evaluation in Section 5. We then present related work in Section 6 before presenting our conclusion in Section 7.

2 THE SPACE CONSISTENCY PROGRAMMING ABSTRACTION

For the runtime developed in this study, the target programming abstraction is the space consistency model [20, 21]. The abstraction defines consistency guarantees at the granularity of memory spaces. These memory spaces are symmetric objects allocated across a team of ranks, where each object has an associated counter that records the volume of received data. The counter gets incremented atomically with the amount of data received due to a transfer or a collective operation.

A space becomes consistent, *i.e.*, ready for consumption, when the counter reaches a specific value, called a consistency tag. In a sense, this mechanism provides a generalized form of the full/empty synchronization mechanism, supported by some architecture such as MIT Alewife [1]. The model provides consumers with the ability to check for the consistency of space, while no mechanism is provided for tracking individual transfer completions for the producers (the transfer initiators). The consistency tag is either calculated independently or communicated in another communication step in case of irregular patterns. For irregular communication, one needs to size the space carefully to meet the worst-case data volume.

[2]CSPACER stands for Consistent SPACE Runtime.

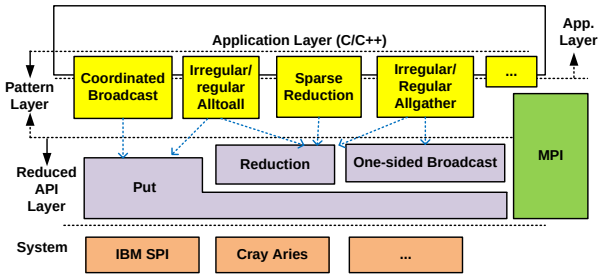

Figure 1: The CSPACER software layers for accelerating MPI applications. Three primitives are provided as core APIs: put, broadcast, and reductions. Other functionalities are provided as patterns that could be integrated into compute phases within the application for performance critical regions.

This abstraction's main strength is harnessing the one-sided semantics to have low-overhead data transfers while providing the ability to reason about producer-consumer relations similar to two-sided transfers. For optimal performance, the abstraction would require an interconnect with RDMA support and NIC with atomic increment capability that is ideally coherent with respect to the CPU. Prior work leverages hardware support in the underlying interconnect architectures. This work fully implements the abstraction using software mechanisms, relying on non-coherent NIC atomics.

3 THE CONSISTENCY SPACE APPLICATION INTERFACE

Figure 1 depicts our MPI-interoperable software stack. One main core functionality is dependent on the system, which is the chained one-sided put operation. A put operation involves an update to the consistency counter associated with a memory space. We provide two additional core functionalities, reduction and broadcast. The rest of the communication primitives are provided as patterns that should preferably be integrated with the application's computational kernels. This integration of the pattern layer with the application facilitates pipelining and overlap of computation with communication.

3.1 Application Interaction with Runtime and API Design

In this section, we revisit the interface between an application with a runtime during execution. The key question is how to design an interface that improves the effectiveness of supporting pipelining, asynchronous progress, leveraging multicore architectures, *etc.*

The contrast between reduced API runtimes with complex API runtime designs manifests in the way applications interact with runtime systems. For instance, in MPI, an application would construct a complex operation, such as MPI `allreduce` over a vector of variables using a single call. This reduction would require multiple stages of processing within the MPI software stack depending on the algorithm, which is influenced by the vector length and the rank count. Generally, the runtime provides an interface for providing the data, description of the operation, and the meta-data about the operation, such as types and vector length, using a single API. The runtime also provides an API to check or wait for the

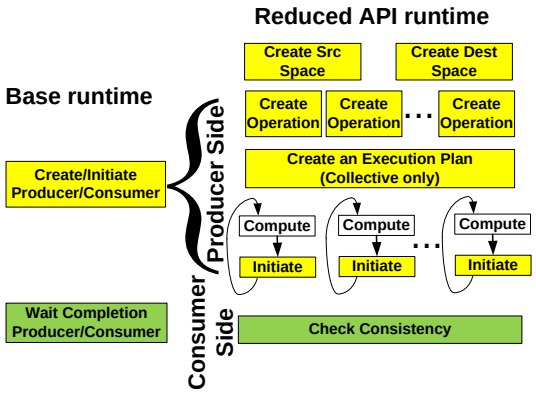

Figure 2: Contrasting complex API with reduced API set runtime in implementing communication primitives. A reduced API set runtime would require creating multiple operations to accomplish a single complex functionality, would decouple transfer meta-data from data availability, and would support efficient pipelining of transfer operations.

completion of the operation, if the application chooses the non-blocking interface. Given that many activities could be involved in performing the request, depending on the algorithm used by the runtime, the application needs to progress the runtime to complete the request. This progress could either be done asynchronously, using a progress thread or at the time the application request or check for completion.

By a reduced API design, shown in Figure 2, we mean decomposing the complex operation into multiple simple operations to enable efficient execution of the communication primitive. First, the application is required to declare the intent of memory region usage, *e.g.* for reduction, during the communication buffer allocation. This gives the runtime a chance to allocate all scratch memory needed for internal processing to optimize the performance. The application then creates one or multiple operations to perform a transfer operation, for instance, to allow parallel injection by independent threads or to concurrently deposit data to the runtime for processing. Afterward, the runtime requires identifying the meta-data about the operation before the data is readily available for communication. The runtime could decide what algorithm to best serve the application request and how internal runtime tasks are split between ranks before the data is readily available. Finally, the data is deposited to the runtime in a pipelined fashion as they get produced. This way, the interaction between the runtime and the application becomes more frequent, allowing for better progress of the operation, improving the tolerance to imbalanced arrival to collectives, and reducing the burstiness of injecting traffic to the interconnect. The presented approach does not entail implementing the collective algorithm at the application layer. For instance for reduction, the runtime fully implements the algorithm, including the distribution and the execution of the work, and how to schedule communications. Similarly for broadcast, our runtime uses a topology-aware tree transparently. This reduced API approach merely creates the interface for a pipelined and threaded interaction of the application computation with the runtime upon the readiness of a data unit for processing by the collective algorithm.

Reduced API design requires a low-overhead, in the range of 100s of ns, implementation of the runtime. Otherwise, the potential advantages of the mechanisms described above could be overtaken by runtime overheads. More importantly, the programming abstraction should support relaxed consistency semantics that tolerate out-of-order processing of transfers and tolerate relaxed consistency guarantees rather than associating consistency with individual transfer requests. This relaxed semantic enables a wide set of runtime optimizations, discussed in the following sections.

3.2 CSPACER Threaded Support

In the multicore era, there are multiple ways an application could leverage a library supporting a threaded implementation. As shown in Figure 3, the interaction between the application level threading and an external library threading could happen in multiple ways. Libraries, such as MAGMA [18], adopt a single interface for non-threaded and threaded implementations, relying on special initialization and/or linking with a specialized library to control the threading support within the library. Such a unified interface simplifies the application development cycle but forces the application and the library to use independent fork-join models. For communication runtime, we may have limited interconnect resources that do not match the number of threads supported at the application level. As such, protecting shared resources within the runtime through locking or lock-free data structure, required for correctness, could severely impact performance.

CSPACER supports two main threading models. The first is intended to support concurrent access with no runtime serialization due to lock or atomic primitives, Figure 3.a. The second is cooperative models, Figure 3.d, where the application-level threads simultaneously call the runtime to accomplish a single communication operation, whether this involves computation such as reduction operations or data movement involving copying of data across ranks within a node. The first model requires designing the application such that it adapts to the concurrency level supported by the runtime. We provide the supported concurrency level as queriable resources that should be explicitly used in creating a transfer operation. For the second model, Figure 3.d, our design relies on leveraging application-level threading by the runtime for internal processing, which alleviates the need to handle multiple threading models across layers of the software stack. This mode requires a set of threads within the application to call the same APIs with their unique id. Valid ids depend on the number of thread count registered with the runtime. To enable such support, we provide special API variant for each supported primitive. CSPACER runtime is oblivious to the threading library used by the application and could interact with any application threading model. This interaction model is uncommon in providing threading support for both the application layer and external libraries to the best of our knowledge. The primary motivations for having this model are avoiding the complexity associated with a) coordinating communication and threading runtimes in case of leveraging a common threading runtime for application computation and runtime communication tasks; b) handling thread-binding on HPC platforms in case of using different threading runtimes at both layers.

A special case of Figure 3.c is the sharing of threads between two layers of runtimes, *e.g.* MT-MPI [31] thread sharing between

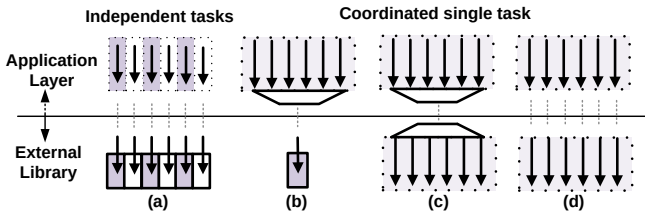

Figure 3: The interaction between the application layer and another library could be done through multiple mechanisms. Typically, applications use an independent threading model from the runtime. Our communication runtime supports independent injection of transfer (mode (a)) or coordinated runtime call (mode (d)) where application-level threads progress the runtime complex computational tasks.

OpenMP and MPI. MT-MPI requires modifying the OpenMP runtime to enable querying and using idle threads by the MPI runtime. As such, the MPI and threading runtimes need to coordinate their work, which ideally require a standardized interaction mechanism. On the other hand, MT-MPI preserves the application/MPI interface as a single thread need to call the MPI runtime.

The Space Consistency abstraction guarantees lock-free, and atomic-free, support for the concurrency level reported by the runtime query. For concurrent access, Figure 3.a, the application needs to control the level of concurrent access to the runtime, making at most one thread access per lane[3]. Consequently, two levels of concurrency need to be managed by the application one for computation and another one for communication. Fortunately, OpenMP standards support controlling the level of concurrency for each parallel region independently, in addition to supporting nested parallelism with control of the concurrency degree at each level. The main advantage of this mechanism to the application layer is improving performance predictability. Earlier proposal such as MPI endpoint provides such thread support for an arbitrary level of application concurrency. Therefore, correct execution would require lock to guard access to shared data structures, which involves added overhead and serialization in accessing the interconnect resources. The problem is typically more challenging with MPI two-sided semantics because of the tag matching complexity [19].

3.2.1 CSPACER Data Copying Avoidance. Multicore architectures made multithreading and shared memory programming very prevalent in HPC. Communication runtime often needs to do copying or sharing of data, especially while implementing collective primitives. If multiple ranks within a team, or communicator, reside in the same node, the runtime would need to do intra-node broadcast (through copying) following the internode broadcast. Requiring such intra node broadcast may not be needed if all the ranks are doing read-only access to the broadcasted data, and the runtime would provide a mechanism for sharing the node leader memory with other members of a team within a node. Typically, we could achieve this using a shared memory segment that is mapped to all the ranks within the node. Avoiding such copying overhead could provide a noticeable performance advantage, especially for light-weight core design such as Intel KNL, where more than half the operation time could be spent for local data copying.

[3]A lane [20] is lock-free runtime resources for accessing the interconnect.

```
10  csr_get_channel_count(&channel_count);
    ...
12  csr_get_team_handle(CSR_SELF_TEAM,&h_self_team);
    csr_get_team_handle(CSR_COLUMN_TEAM,&h_col_team);
14
    csr_sym_allocate_memory(h_col_team, space_size,
16          CSR_REDUCTION64_SPACE,&rmt_cspace_hndl);
    csr_sym_allocate_memory(h_self_team, space_size,
18           CSR_DEFAULT_SPACE,&src_cspace_hndl);
    csr_get_space_ptr(rmt_cspace_hndl,(void **)&rmt_space_ptr);
20  csr_get_space_ptr(src_cspace_hndl,(void **)&src_space_ptr);

22  csr_operation_t op = CSR_ADD_FP64;
    csr_construct_reduction (rmt_cspace_hndl,op,channels[0],
24          tx_handles+0);

26  csr_plan_reduction(element_count,CSR_DEFAULT_REDUCTION_CHUNK,
          pattern,INPLACE_REDUCTION,tx_handles[0]);
28  csr_set_team_helper_thrd(h_col_team,first_level_omp_threads);

30 /* Active Producers */
   #pragma omp parallel
32 {
     for(t=0;t<chunk_count;t++) {
34     /* Produce chunk of data */
        #pragma omp for nowait
36      for(i=start;i<end;i++)
          src_space_ptr[i] = f(i);
38     /* Start the reduction of the produced chunk */
       csr_deposit_reduction_thrd(rmt_cspace_hndl,src_cspace_hndl,
40        start,chunk_size,tid,nthreads,sync_mode);
     }
42 }
   /* Consumers, wait for the consistency resulting from all
      deposits. */
44 #pragma omp parallel
   {
46      csr_wait_consistent_thrd(rmt_cspace_hndl,tag,tid,nthreads)
      ;
   }
48 /* Now, consumers could access the data. */
   for(i=0;i<size;i++)
50    local[i]=g(rmt_space_ptr[i]);
```

Figure 4: The computation-communication pattern for an `allreduce` operation. The pattern involves creating operations, planning through providing type, size, and algorithm, and finally issuing the transfer requests while doing computation within a threaded region. The consistency is checked at the end by the consumers.

CSPACER provides applications the choice to access leader data pointer while waiting for a space to be consistent. Unfortunately, we do not have an efficient mechanism for enforcing read-only access to the data. As such, it is the responsibility of the application to ensure that the leader shared copy is not overwritten. Multiple numerical algebra algorithms would benefit from avoiding copying, including distributed matrix-matrix multiplication, and LU factorization. In both cases, blocks of data are broadcasted across a team of ranks and are read-only accessed.

3.3 Construction of Communication Pattern

A major objective of most of the design choices of the CSPACER runtime is to enable computation and communication overlap while leveraging multicore architectures, efficiently. Some of the complex operations, such as `allreduce`, `allgather`, etc, are decomposed into multiple application/runtime phases rather than having a single phase of interaction. We will demonstrate a couple of simple examples to clarify the steps of creating these patterns.

The `allreduce` example, shown in Figure 4, illustrates the phases of constructing the operation over a symmetric space. It starts with the allocation of special space for processing a reduction, *line* : 15 – 18. Then, the application creates an operation for reduction, *line* : 23, and provides information about the intended

```
10  csr_init_runtime(argc,argv,config_file_name);
    csr_get_lane_count(&lane_count);
12
       /* Only the root inject data  */
14  if(root) {
       /* Create multiple transfers, one per thread. */
16     for(i=0;i<lane_count;i++) {
          csr_get_lane_handle(i, lanes+i);
18        csr_construct_broadcast(rmt_cspace_hndl, 0,
                                  src_cspace_hndl,0,chunk,
20                                lanes[i],tx_handles+i);
       }
22    #pragma omp parallel num_threads(lane_count)
       {
24        csr_broadcast_handle_t tx_hndl = tx_handles[tid];
          csr_lane_handle_t mylane = lanes[tid];
26
          for(t=0;t<tx_count;t++) {
28         /*
           #pragma omp for num_threads(second_level_omp_threads)
30         {
            Threaded compression of broadcasted data.
32         }
           */
34         csr_update_broadcast(thr_start,chunk_size,tx_hndl);
           while(csr_initiate_broadcast(tx_hndl) != CSR_SUCCESS)
36            /*retry*/;
          }
38        /* Without hardware support, we need this lane flush. */
          csr_flush_lane(mylane);
40     }
    }
42
    /*Consumers, rather than copying, could read the node leader's
       data */
44  /*Tag may require another broadcast, if the data size is not
       known to consumers apriori. */
    csr_wait_leader_consistent(rmt_cspace_hndl, tag);
46  csr_get_node_leader_space_ptr(rmt_cspace_hndl, dest_space_ptr);
```

Figure 5: Concurrent Computation-communication pattern for one-sided broadcast. Root could do threaded injections of different chunks of data in out-of-order fashion.

operation, transfer size, etc, *line* : 26. The application alternates between compute, *line* : 36, and data deposition to the runtime, *line* : 39, within an OpenMP parallel region. These frequent interactions between the application and the runtime provide a better opportunity for progressing the runtime and better tolerance for imbalanced arrival to the collective. Threading provides more compute power to finish the reduction task. Finally, the data consumer wait for the space to become consistent, *line* : 46, before accessing the data.

Figure 5 shows how to perform a one-sided broadcast operation, where only the root knows the amount of broadcasted data while other team members only know the worst case volume of broadcast. We create a transfer operation for each thread, *line* : 18, such that each thread could progress independently, allowing concurrent injection of data, *line* : 34 – 36. Unlike the first example, each thread independently chooses the transfer size and offsets at the source and the destination. If the broadcasted data are read-only, then the consistency is checked for the team's node leader, *line* : 46. Once consistent, the leader copy of the data is used by all ranks within the team, rather than creating private copies. If the broadcast operation involves some compression, then the metadata would require another broadcast operation, which is issued after the compression but need to be completed first. The metadata broadcast enables establishing the consistency tag of the irregular broadcast.

Our implementation [32] include other patterns, such as `alltoall` or `allgather`, based on the use of put or broadcast operation, respectively. For the irregular variant of these operations, we use an additional communication phase for establishing the consistency tag following the main data communication phase. An application developer could integrate these skeletons into their code, specifically for communication-bound hotspots.

Due to the lack of hardware support for collectives over a vector of elements in the Cray Aries interconnect, we introduce a variant of the broadcast operation that would require the involvement of all ranks within a team, and we call this variant a coordinated broadcast. This primitive allows concurrent forwarding of the broadcast data by the whole team to improve the efficiency of the operation while still adopting the one-sided broadcast semantics where only the root injects data and decides the amount of data to broadcast. The consumers could progress the operation by checking the consistency of the broadcast space against the worst-case broadcast volume. We also built a coordinated `allgather` on top of the coordinated broadcast primitives. Both regular and irregular broadcast use the same API, similarly for `allgather`. The consistency mechanism is slightly more involving in the irregular case and would typically require an additional communication step of the metadata.

Internally, CSPACER uses a simple algorithm for reduction, where it assigns the nodes participating in the reduction chunks of the data in a round-robin fashion. The reduction mechanics is decided based on the information provided in the reduction plan call. Moreover, ranks within a node transparently split the work of making the reduction. As chunks get deposited to the runtime, the CSPACER runtime performs local reduction before the inter-node one. Only node leaders participate in the second phase of reduction, and they broadcast the results back to all node leaders. Non-leader ranks finally copy the results if needed. The CSPACER implementation of the coordinated broadcast uses a simple cut-through forwarding using put operation on a topology-aware binary tree of ranks within the team. This cut-through forwarding could easily be offloaded to a hardware-accelerated engine. CSPACER also leverages shared memory bypassing using xpmem [3].

3.4 Runtime Management of Interconnect Resources

Frequently, the runtime provides communication resources, such as communicators, to the application layers as allocate-able resources. It may also try to support concurrent access to an arbitrary number of threads at the application level. At the interconnect level, these resources could be limited and frequently shared by all ranks using the same network interface, in case of concurrent access, or the network switch, in case of supporting hardware collectives. As such, the communication runtime could face a challenge mapping software resources onto the limited hardware resources. Moreover, the application may encounter an accelerated runtime path if certain conditions are satisfied, making performance predictability a challenge. On the other hand, developing applications against virtual interconnect resource abstraction provides development productivity and portability across systems.

In designing our runtime, we rely on presenting performance-critical resources as being `query-able`, rather than `allocate-able`. For instance, the number of injection lanes that are supported by the runtime depends on the underlying architecture and the number of ranks sharing the node. As such, the application needs to query such critical resource and structure the code dealing with the possibility

of having variable lane count. Similarly, for hardware-accelerated collectives, the application should not assume a particular availability of these resources, similar to software allocated resources. Often, if the resources are limited, the application needs to choose which communication pattern is critical to its performance and uses scarce resources for critical communication patterns.

The presented approach makes it a necessity to interoperate with a general-purpose programming model, such as MPI. This helps in identifying whether communication bottlenecks exist within the code. Porting communication hotspot into the CSPACER runtime is done guided by an application profiling phase. In general, judging where communication resources should be used is essential to reaching optimal performance. Typically, the amount of code that needs rewriting using this approach is expected to be a small percentage of the MPI-based application code.

4 EVALUATION METHODOLOGY

4.1 Platform

We conducted our experiment on the Cray Cori (XC40) system at NERSC. The system has a Cray Aries interconnect with Dragonfly topology. Two kinds of compute nodes are connected to the interconnects. The system has two types of compute nodes: 9,688 Intel Knights Landing compute nodes and 2,388 Intel Xeon Haswell. The KNL nodes operate at 1.4 GHz and have 68 cores per node, 4 hardware threads per core, 1.4 GHz, 512-bit vector units, 96 GB DRAM, and 16 GB on-package MCDRAM. The Haswell compute nodes operate at 2.3 GHz, and have 32 cores/node and 128GB of DDR4 memory. The experiment conducted in this study used the following list of programming environment: intel/19.0.3.199, PrgEnv-intel/6.0.5, cray-mpich/7.7.10, craype-mic-knl, craype-hugepages2M, dmapp/7.1.1, xpmem/2.2.20, ugni/6.0.14.0, and SHMEMX/9.0.0.

4.2 Microbenchmarks

The OSU benchmarks [27] provides a comprehensive set of benchmarks for testing various functionalities for the MPI programming model, along with other programming models. We used OSU benchmarks, v. 5.6.2, to assess point-to-point, one-sided, and collective performance. We used Cray SHMEMX, for evaluating the Shmem performance on Cray systems. For SHMEM benchmarking, we used a slightly modified version of SHMEM-MT [37], where thread synchronization during injection is removed because it adds unnecessary overhead given the Cray thread-hot support.

Although the benchmarks assess the performance for various transfer sizes, we focused solely on performing communication on large data sets, where the data transfer is usually bandwidth limited. On the Cray systems, communication primitives become bandwidth-limited at roughly 4-32KB. On Haswell-based nodes, the bandwidth-limited transfers are smaller than the KNL nodes, which require larger transfers to reach the same efficiency of utilizing the interconnect. Unless stated otherwise, we measured the latency to communicate or to reduce 64 MB of data.

4.3 CSPACER Use in Math Libraries

We evaluated the effectiveness of the presented runtime on distributed matrix multiplication. Although matrix multiplication should be asymptotically compute-bound, the change of machine balance across supercomputer generations [24] makes matrix multiplication communication-bound for matrix sizes of interest. For distributed matrix multiplication, typically implemented using ScaLAPACK, we used two popular algorithms, the Cannon [23], and the SUMMA [36] algorithms. We used the 2.5 D decomposition [33], proposed by Solomonik for being communication efficient. Both 2.5D Cannon and 2.5D SUMMA rely on having replicated state of the input matrices that would reduce the volume of communication. Each group of ranks creates a 2D plane involving multiple communication phases. The 2.5D decomposition reduces the data movement across ranks, through replication of the input and output matrices across groups of ranks. These groups independently calculate partial results that are collectively reduced at the end of the computation. The communication pattern within a plane depends on the algorithm. For SUMMA, the data is broadcasted across the rows and columns from the root owning the source data. For CANNON, the data moves circularly across rows and columns. As such, SUMMA relies on the performance of broadcast, while CANNON depends on the performance of point-to-point primitives. Both rely on the reduction of partial results. We find these algorithms to provide representative test cases for the performance of the developed runtime because they stress the support for point-to-point, broadcast, and reduction.

To improve the efficiency of executing these algorithms, we introduced a pipelined variant of the algorithm, where while doing computation on a matrix panel, another panel is being communicated. For MPI, we used the non-blocking interface for send/receive or broadcast for CANNON and SUMMA, respectively. This pipelined approach stressed the effectiveness of overlapping computation with communication. Using MPI, we need to construct communicators for all three dimensions. Two communicators are across rows and columns, and a third across planes. The memory requirements increase for the case where the three dimensions are close in size. The 2.5D allows controlling the tradeoffs between memory requirement and performance, providing a rich set of configurations compared with the 3D case [2].

The pipelined version is the one we extended to use the CSPACER. Additionally, we used the threaded injection strategy of transfer and pipelined the injection while packing the transfers. We observed that sending data in chunks allows more time-spaced injection of data and correlate with better performance. For CSPACER broadcast and reduction, we use the threaded version of transfer initiation and consistency checking. For point-to-point, we leverage concurrent independent injection of the transfers. We did not use a similar strategy for the MPI variant because the OSU benchmarks show a significant performance penalty for using threaded injection. For instance, on KNL nodes, the latency increases from 3 us to 25 us, for 8 B messages, when we increase the number of threads from 1 thread to 4 threads. At 8 threads, the latency reaches 67 us. In general, we observed that the latency increases super-linearly with the number of threads, which is a behavior that affects not only small transfers but also large transfers, as well.

4.4 Application Use of CSPACER

We study the use of CSPACER with the MIMD Lattice Computation (MILC) [7], which is one of the software packages in the

USQCD framework [35] used to study the lattice quantum chromodynamics (QCD) theory. The lattice QCD theory models the strong interactions of subatomic particles, quarks, and gluons to form hadrons. The Lattice QCD computation involves discretized four-dimensional SU(3) lattice gauge, three dimensions are in the space, and the fourth is the time dimension. In distributed environment, the lattice structure is distributed by decomposing the lattice across ranks in one or more of the lattice dimensions.

The MILC code has several specialized computational kernels. In this work, we use the dynamical rhmc code (relational hybrid monte carlo algorithm, or su3_rhmd_hisq), version 7.8.1, for benchmarking. The MILC package has multiple specialized implementations depending on the target platform. For Intel-based systems, it leverages the Intel QPhiX library for the conjugate gradient solver, which provides an optimized vector implementation for various Intel architectures. In a distributed computing environment, the library also performs point-to-point halo exchange between various sites. The communication pattern is a simple nearest neighbor pattern, where depending on the problem decomposition, each rank communicates with up to 8 neighbors using non-blocking primitives.

We modified the two boundary exchange logic within the package responsible for communication for the gauge force (GF) and the Kogut-Susskind (KS) calculation. In both cases, the library provides a threaded function for packing and unpacking lattice faces to exchange with neighbors. We also used a threaded injection for transfer, which allows threaded copying if the neighbor resides within the node. We also used a separate communication lane for each communication direction to allow independent injection and progress of these lanes.

5 PERFORMANCE EVALUATION

This presents the performance evaluation of the presented runtime using microbenchmarks and when integrated into scalable solver libraries or with an application.

5.1 Runtime Overheads

Very often, a newly developed runtimes report on the latency of performing a data transfer operation. The CSPACER runtime does not rely on establishing consistency at the granularity of a data transfer. As such, it views consistency as an aggregate measure over a memory space. Therefore, to assess our runtime efficiency, we report the latency to initiate a transfer and the latency to check whether a transfer is not possible due to lack of resources or the NIC being busy processing prior requests. On KNL architecture, our runtime takes around 280 *ns* in the non-contended case, reaching 490 *ns* in the contended case, to initiate a put transfer. On Haswell, the latency is 32 *ns* and 72 *ns*, for the non-contended and the contended cases, respectively. We believe that the large difference between the two architectures is influenced by the cost of executing a memory fence to control the PCIe-based NIC and the difference in core execution capability. A failed attempt to initiate a transfer could take up to 50 *ns* on KNL and as little as 10 *ns* on Haswell. These low overheads allow the application to invoke the runtime while producing the data more frequently.

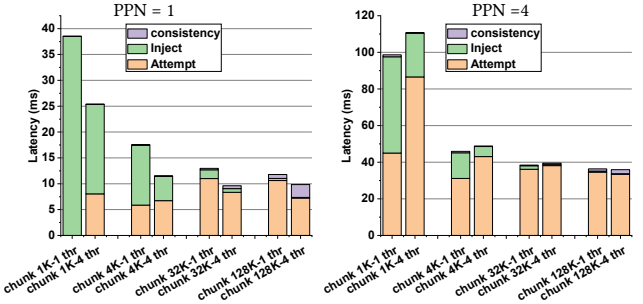

Figure 6: The time decomposition of concurrent injection of put transfers with different levels of process concurrency for 64MB of data. Threading improves the performance, especially for small chunk size, and small number of ranks sharing a node.

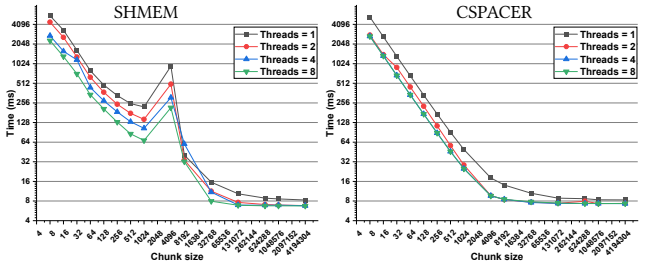

Figure 7: The latency to send 64MB between two processes using different chunk sizes at different threading levels for CSPACER and SHMEM. CSPACER achieves mimimum latency at smaller chunk and lower concurrency.

5.2 CSPACER Runtime Efficiency

We report on the performance improvement with concurrent injection in Figure 6. We show the impact for different chunk sizes, thread concurrency level, and rank concurrency levels per node. For 1KB chunk, the performance improvement of using four threads compared with a single thread is as high as 50%. The performance improvement gets smaller as we increase the chunk size because the bandwidth to the interconnect becomes the major bottleneck. We notice that attempts to inject transfers take the majority of time in the experiment when the chunk size increases. In our experiment, we use back-to-back injection, *i.e.*, a sort of flooding pattern. In practice, large chunks would require larger computation time, and as such larger inter-arrival time inject traffic to the runtime.

As we start using more ranks per node, we notice that concurrent injection could be associated with a 5-10% performance penalty with our flooding pattern. In contrast, the thread multiple support on MPI results in a super-linear increase in latency as we increase the number of threads injecting traffic. For instance, for 1KB transfers the latency increase from 4.5 *us* to about 30 *us* with four threads, according to the `osu_latency_mt` benchmark. These experiments clearly show that concurrent injection could be beneficial, especially in reaching bandwidth saturation of the interconnect. The concurrent injection could also be beneficial when the target rank resides within the node. In this case, the use of more threads improves the copying efficiency of the data.

One of the most efficient programming models that efficiently support threading on Cray systems is SHMEM, especially with

Cray SHMEMX hot-thread. In Figure 7, we contrast the performance of CSPACER with Cray SHMEMX with hot-threads, which is one of the most efficient low-overhead models on Cray systems. CSPACER uses only BTE-based protocol for all message sizes, and could reach the best performance with two threads only, matching the number of BTE channels available for point-to-point communication on the Cray Aries interconnect. With concurrent injections, 4KB is enough to drive the interconnect at full speed, while a single thread would need 128KB. Cray SHMEMX requires a chunk of at least 32KB for threaded implementation to reach the same efficiency because it combines the transfer creation and initiation. We also observe the impact of switching between the FMA and the BTE protocols for SHMEM, which degrades the performance for chunk sizes of 1 – 32KB. The CSPACER transfer for small transfer is as low as FMA protocol due to splitting the transfer into two phases: creation and initiation. With MPI, we noticed performance degradation with multithreading support relative to the use of single-thread, similar to those reported in earlier studies [19].

5.3 Performance of Collectives

In this section, we show the performance (latency) of performing multiple collective operations using CSPACER and contrast that with the performance observed using MPI and Cray SHMEMX.

5.3.1 Broadcast Performance. In Figure 8, we show the latency of broadcasting a 64MB of data using MPI, SHMEM, and CSPACER. For MPI, we observe a significant increase in latency when the number of nodes increases. Given the large transfer size, we would expect that the latency of propagating the broadcast will be bandwidth limited. Optimized latency for the broadcast of large messages [29, 34] should ideally be $(log_2(p)+p-1)\alpha+2\frac{(p-1)}{p}n\beta)$, where p is the number of ranks, α is the startup time, β is the time per byte, and n is the number of bytes. This time complexity [34] assumes scattering the data across all nodes, followed by an `allgather` operation. In the first part of the equation, the latency term should not significantly dominate the latency for large transfer. Instead, we observed an increase in latency proportional to the $log_2(p)$. We explored multiple variants, including the use of Cray DMAPP, change of collective protocols, use of asynchronous progress threads. We observed performance variation of up to 14% between these algorithmic choices. We decide to report on the default setting of MPI because the differences are not significant. Moreover, as we increase the number of ranks per node, we also observed an increase in the latency for performing the broadcast. With SHMEM, we observed similar behavior for broadcast. By exploring different settings for SHMEM, we observed a slightly accelerated performance for single rank per node, which is not shown for brevity.

The CSPACER implementation delivers a significant improvement in latency and scaling. Our implementation is based on pipelined cut-through forwarding using a binary tree organization of the ranks within a team. Receiving nodes keep forwarding data as they receive them. As such, the latency do not change significantly with the number of nodes. We reduce this latency by using multiple threads while checking for the completion of the space. Overall, the performance advantage of CSPACER on 64 nodes is up to 3.8× compared with MPI.

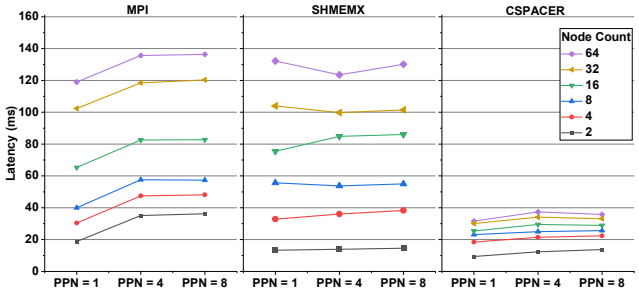

Figure 8: The latency for broadcasting 64MB of data, for MPI, SHMEMX, and CSPACER, respectively. CSPACER has lower latency compared with other models and introduces lower latency increase as we scale.

5.3.2 AllReduce Performance. We tested the performance of an `allreduce` operation using MPI, SHMEM and CSPACER. While for MPI and SHMEM, we report on the time to do the collective operation solely, for CSPACER, we include the time to setup the array value as it interleaves with the communication. As shown in Figure 4 *line* : 39, the reduction is performed as a series of deposit operations. One of the algorithmic choices is deciding the chunk size for depositing reduction data. In Cray XC40 system, we observed the best performance with 32KB chunks.

The impact of using threaded implementation is illustrated in Figure 9, where we show the performance. Comparing with the MPI performance, shown in Figure 9, we notice a significant performance advantage to CSPACER. For instance, for 16 nodes with 4 processes per node, CSPACER delivers 3.9× latency improvement. The performance advantage depends on the number of ranks per nodes and the level of threading. We observe increase latency for the MPI implementation between 16 and 64 nodes for both MPI and CSPACER. We conjuncture that the node placement on Aries Dragonfly network starts making the performance dependent on traversing more global links.

Compared with SHMEM and MPI, CSPACER provides a performance advantage only when threading is enabled. The single-thread performance of SHMEM and MPI are higher than CSPACER, but they do not have the option of threaded processing of the reduction operation. CSPACER has a single algorithmic choice for reduction and is amenable for further tuning to improve the single-thread performance, a task we will pursue in future work.

5.4 Matrix Multiplication Performance

Most new supercomputing machines have higher flop/byte machine balance, making communication a growing challenge for performance [24], even for the traditionally compute-bound problems such as distributed matrix multiplication. We explored the performance of two variants of matrix multiplication using CSPACER- and MPI-based implementations.

We report on the best implementation for each programming model. For computation, we used the same thread concurrency for both implementations. For communication, because a threaded implementation is advantageous for CSPACER, we adopt it. While for MPI, we used the non-threaded communication implementation because it delivers the best performance.

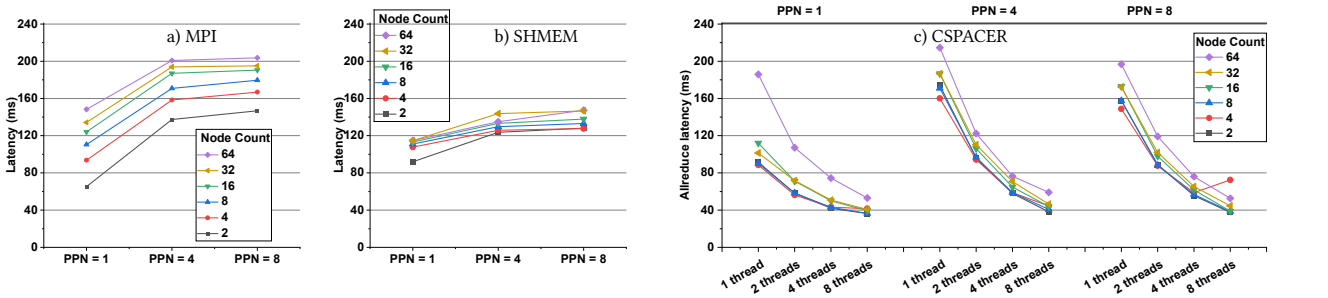

Figure 9: The performance of the `allreduce` operation for MPI, SHMEM, and CSPACER, on 64MB data using various ranks per node. For CSPACER, we vary the threads per rank. In general, CSPACER threaded version improves with the use of more threads, and is associated with lower latency than other models.

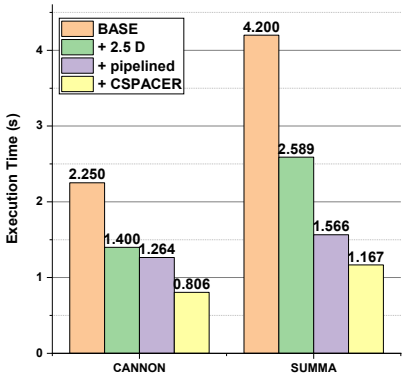

Figure 10: Matrix multiplication using 2.5D Summa and Cannon algorithm for square matrices of size $32K^2$.

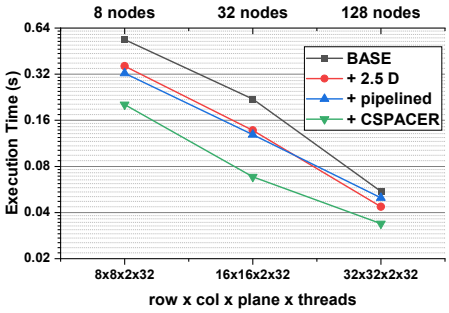

Figure 11: Strong scaling of matrix multiplication using the Cannon algorithm for square matrices of size $8K^2$.

Depending on the number of nodes, we construct 3D teams. The layout is expressed as $row \times col \times plane \times threads$. For instance, on 32 nodes, we use team layout of $8 \times 8 \times 2 \times 32$, while we used $4 \times 4 \times 2 \times 32$ layout on 8 nodes. We have four ranks per node in all cases, which delivers the best performance and stress both inter- and intra- node transfer mechanisms.

As shown in Figure 10, testing the performance of multiplication of matrices of size $32K^2$ on 128 nodes, the CSPACER-based implementation improves the performance over the best MPI-variant by 1.56× and 1.34× for SUMMA and CANNON, respectively. The CANNON algorithm delivers better performance compared with SUMMA, but algorithmically it requires perfectly square decomposition. The SUMMA algorithm does not impose such constraints.

We tested the strong scaling performance for matrix multiplication $8K^2$ on 8-128 nodes, using the CANNON-based implementation. As shown in Figure 11, the CSPACER-based implementation significantly improves the performance compared with the best MPI-variant. The speedup increases from 1.6× for 8 nodes to 1.89× using 32 nodes. At 128 nodes, the performance improvement is roughly 1.48×. Note that we did not fully leverage computation communication overlap, and we merely overlapped data packing with communication.

5.5 Lattice QCD Performance

The conjugate gradient kernel is one of the most computationally demanding in Lattice QCD computations. Figure 12 shows the performance of the CG component for a) MPI and b) CSPACER-based variants while strong-scaling the calculation of lattice of size $64^3 \times 64$. As shown in Figure, CSPACER improves over MPI by up to 1.58× on 16 nodes and by up to 1.41× at 128 nodes. The figure also shows the variability in CG performance which is inversely proportional to the quark masses being simulated. CSPACER exhibits a higher variability than MPI at low node code. We found the correlation of the CSPACER runs to be higher, *i.e.* more predictable than MPI at low node count. At high node count, system variability significantly influences the execution time making the variability less predictable for both CSPACER and MPI.

Figure 13 shows the strong scaling behavior for the whole application for 16-128 nodes, using four ranks per node. For both MPI- and CSPACER-based variants, using 256 hyperthreads per node delivers the best performance at low node count. At high node count, 128 hyper-threads delivers the best performing configuration. Comparing the best performing configuration of both MPI and CSPACER, we measure and an end-to-end speedup of 1.19× and 1.28× at 16 and 128 nodes, respectively. In all reported experiments, we modified a subset of the MPI call involved in the halo exchange. The rest of the communication code still uses the MPI APIs. The performance advantage of CSPACER is due to the efficient use of pipelined concurrent injection of data by different threads.

6 RELATED WORK

The Space Consistency model adopts PGAS semantic adopted by a wide class of programming models including SHMEM [30], UPC [9, 10], UPC++ [5, 40], Global Arrays [25], Corray Fortran [26], etc. These models relies on various libraries such as GASNet [8, 38], MPI RMA [15], ARMCI [25]. The majority of PGAS models provide the ability to do direct remote access for applications in distributed programming environments. They differ in the base language, C, C++, or Fortran, whether they require symmetric heap allocation, the synchronization and consistency semantic, etc. The consistency space

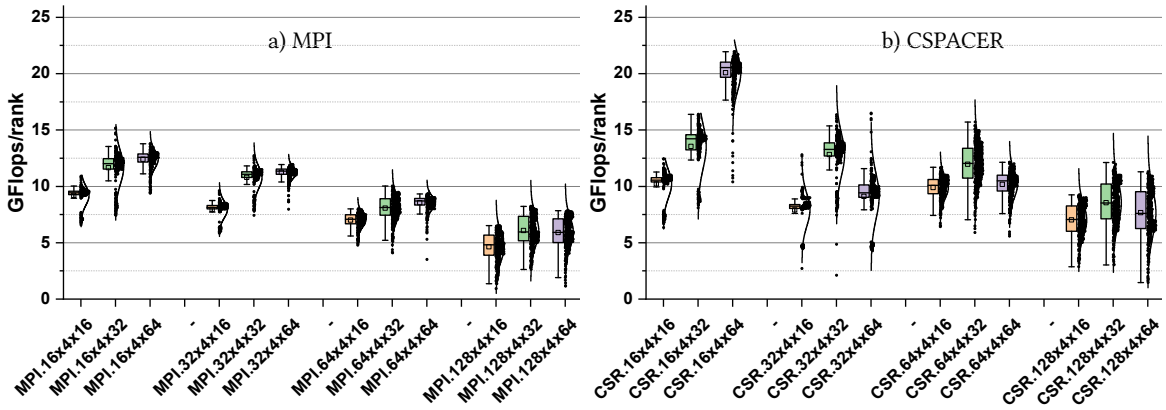

Figure 12: The throughput of the conjugate gradient of the lattice QCD calculation using MILC package on KNL architectures on a lattice of the size $64^3 \times 64$ for a) MPI and b) CSPACER. We notice the performance improvement of using CSPACER over baseline using MPI by up to 49%.

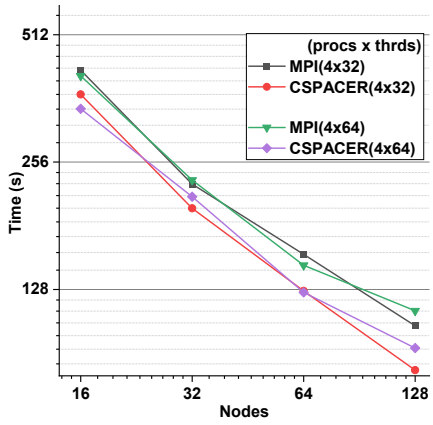

Figure 13: Strong scaling the lattice QCD calculation using MILC package on KNL architectures on a lattice of the size $64^3 \times 64$.

programming abstraction leverages many of the concepts developed in other programming models, including the use of symmetric heap on team-based rank grouping, simple producer-consumer relation based on counting events to support efficient point-to-point and collective communication.

The support for threading in MPI is a well-investigated problem [4, 12, 13, 16, 28, 39]. Most of the efforts are geared towards reducing the overhead of processing transfers in threaded environments. The performance improvement is transparent to the user, but is best efforts as well, and very often not-optimal [28]. The MPI-MT runtime leverages pipelining internally for processing large data, but it does not provide a mechanism for overlapping the production of these data with their processing. NewMadeleine [14] runtime allows efficient handling of many requests through decoupling the application requests from network activities. The runtime uses aggregation, message splitting, and reorder to improve the efficiency of transfers. NewMadeleine explored mainly two-sided point-to-point APIs. Similar efforts have been explored in PGAS language [22]. In these cases, the threading support preserves the sequential interface.

CSPACER provides mechanisms to create application pipelining as a part of a plan, allowing the application to deposit data as they are produced to improve overlapping computation and communication activities. These plans enable the runtime to optimize based on knowing the application's overall data size involved in the operation. An application-centric pipelining, through the creation of independent operations, may not help the runtime choosing the best collective algorithm unless the runtime is equipped with an accurate speculation mechanism of the application intent.

The idea of using one-sided semantic in implementing producer-consumer relation has been explored using signaling put (notifying the receiver of the put) in Split-C programming model [11]. The idea of counting put has also been explored for OpenSHMEM [17] and in MPI notified access [6]. This work extends these concepts to collectives processing using threaded implementation.

7 CONCLUSIONS

In this paper, we present the design and implementation for a runtime that implements the Space Consistency model on Cray XC systems. The design leverages multicore architectures to accelerate communication for point-to-point and collective operations. The CSPACER API design relies on decomposing complex communication primitives, such collective operations, into simpler primitives. These primitives could then be used to construct communication patterns to be integrated into the application computation.

We show efficient point-to-point and collective communication, providing speedup up to 3.8× for `broadcast`, and up to 4× for `allreduce`. We show that we could use CSPACER to improve applications relying on point-to-point communication, such as QCD, and those relying on collective, such as communication-avoiding matrix multiplication algorithms. The CSPACER runtime improves the LQCD conjugate gradient computation by up to 50% and reduce the overall execution time for 2.5D Cannon algorithm by up to 32%.

In future work, we would like to extend the support of the CSPACER runtime to other interconnect, such as Infiniband. Automating the transformation of MPI-based codes to CSPACER is likely to be beneficial for developers' productivity and code maintenance. Moreover, a coding tool that provides design patterns to the developers may also facilitate the adoption of this abstraction and reduce the entry barrier. Another likely desired feature is a mapping of CSPACER to MPI primitives for systems where CSPACER is not supported.

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, Beng-Hong Lim, K. Mackenzie, and D. Yeung. 1995. The MIT Alewife machine: architecture and performance. In *Proceedings 22nd Annual International Symposium on Computer Architecture*. 2–13.

[2] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. 1995. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development* 39, 5 (1995), 575–582.

[3] Bob Alverson, Edwin Froese, Larry Kaplan, and Duncan Roweth. [n.d.]. XPMEM Linux kernel module. https://github.com/hjelmn/xpmem.

[4] Abdelhalim Amer, Huiwei Lu, Yanjie Wei, Pavan Balaji, and Satoshi Matsuoka. 2015. MPI+Threads: Runtime Contention and Remedies. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Francisco, CA, USA) *(PPoPP 2015)*. 239–248.

[5] J. Bachan, S. B. Baden, S. Hofmeyr, M. Jacquelin, A. Kamil, D. Bonachea, P. H. Hargrove, and H. Ahmed. 2019. UPC++: A High-Performance Communication Framework for Asynchronous Computation. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 963–973.

[6] Roberto Belli and Torsten Hoefler. 2015. Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization. *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* 00 (2015), 871–881. https://doi.org/doi.ieeecomputersociety.org/10.1109/IPDPS.2015.30

[7] C. Bernard, T. Burch, C. DeTar, J. Osborn, Steven Gottlieb, E. B. Gregory, D. Toussaint, U. M. Heller, and R. Sugar. 2005. QCD thermodynamics with three flavors of improved staggered quarks. *Physical Review D* 71, 3 (Feb 2005). https://doi.org/10.1103/physrevd.71.034504

[8] Dan Bonachea. 2002. *GASNet Specification, v1.1.* Technical Report UCB/CSD-02-1207. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2002/5764.html

[9] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. 1999. *Introduction to UPC and Language Specification.* Technical Report. Technical report CCS-TR-99-157, IDA Center for Computing Sciences.

[10] UPC Consortium, Dan Bonachea, and Gary Funck. 2013. UPC Language and Library Specifications, Version 1.3. *Berkeley Lab reports* (11 2013).

[11] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. 1993. Parallel programming in Split-C. *Supercomputing ́93.* (Nov 1993), 262–273.

[12] H. Dang, S. Seo, A. Amer, and P. Balaji. 2017. Advanced Thread Synchronization for Multithreaded MPI Implementations. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 314–324.

[13] Hoang-Vu Dang, Marc Snir, and William Gropp. 2016. Towards Millions of Communicating Threads. In *Proceedings of the 23rd European MPI Users' Group Meeting*. 1–14.

[14] A. Denis. 2019. Scalability of the NewMadeleine Communication Library for Large Numbers of MPI Point-to-Point Requests. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 371–380. https://doi.org/10.1109/CCGRID.2019.00051

[15] James Dinan, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. 2013. An Implementation and Evaluation of the MPI 3.0 One-Sided Communication. ANL/MCS-P4014-0113.

[16] James Dinan, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev Thakur. [n.d.]. Enabling MPI Interoperability through Flexible Communication Endpoints. In *Proceedings of the 20th European MPI Users' Group Meeting* (Madrid, Spain) *(EuroMPI '13)*. 13–18.

[17] James Dinan, Clement Cole, Gabriele Jost, Stan Smith, Keith Underwood, and RobertW. Wisniewski. 2014. Reducing Synchronization Overhead Through Bundled Communication. *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools* 8356 (2014), 163–177.

[18] Jack Dongarra, Mark Gates, Azzam Haidar, Yulu Jia, Khairul Kabir, Piotr Luszczek, and Stanimire Tomov. 2015. HPC Programming on Intel Many-Integrated-Core Hardware with MAGMA Port to Xeon Phi. *Sci. Program.* 2015, Article 9 (Jan. 2015). https://doi.org/10.1155/2015/502593

[19] Ryan E. Grant, Matthew G. F. Dosanjh, Michael J. Levenhagen, Ron Brightwell, and Anthony Skjellum. 2019. Finepoints: Partitioned Multithreaded MPI Communication. In *High Performance Computing*, Michèle Weiland, Guido Juckeland, Carsten Trinitis, and Ponnuswamy Sadayappan (Eds.). Springer International Publishing, 330–350.

[20] K. Ibrahim. 2019. Optimizing Breadth-First Search at Scale Using Hardware-Accelerated Space Consistency. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 23–33.

[21] Khaled Z. Ibrahim. 2018. Space Consistency for Distributed Memory Systems. In *PMAW'18, The Programming Models and Algorithms Workshop, IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*.

[22] Khaled Z. Ibrahim and Katherine Yelick. 2014. On the Conditions for Efficient Interoperability with Threads: An Experience with PGAS Languages Using Cray Communication Domains. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS '14)*. 23–32.

[23] Hyuk-Jae Lee, James P. Robertson, and José A. B. Fortes. 1997. Generalized Cannon's Algorithm for Parallel Matrix Multiplication. In *Proceedings of the 11th International Conference on Supercomputing*. 44–51.

[24] Daichi Mukunoki and Toshiyuki Imamura. 2018. Performance Analysis of 2D-compatible 2.5D-PDGEMM on Knights Landing Cluster. In *Computational Science – ICCS 2018*, Yong Shi, Haohuan Fu, Yingjie Tian, Valeria V. Krzhizhanovskaya, Michael Harold Lees, Jack Dongarra, and Peter M. A. Sloot (Eds.). 853–858.

[25] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Aprà. 2006. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *The International Journal of High Performance Computing Applications* 20, 2 (2006), 203–231. https://doi.org/10.1177/1094342006064503

[26] Robert W. Numrich and John Reid. 1998. Co-Array Fortran for Parallel Programming. *SIGPLAN Fortran Forum* 17, 2 (Aug. 1998), 1–31. https://doi.org/10.1145/289918.289920

[27] OSU benchmarks. [n.d.]. OMB 5.6.2. Network-Based Computing Laboratory, Ohio State University, http://mvapich.cse.ohio-state.edu/benchmarks/.

[28] T. Patinyasakdikul, D. Eberius, G. Bosilca, and N. Hjelm. 2019. Give MPI Threading a Fair Chance: A Study of Multithreaded MPI Designs. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–11.

[29] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. 2005. Performance analysis of MPI collective operations. In *19th IEEE International Parallel and Distributed Processing Symposium*. 8 pp.–.

[30] Stephen W. Poole, Oscar Hernandez, Jeffery A. Kuehn, Galen M. Shipman, Anthony Curtis, and Karl Feind. 2011. *OpenSHMEM - Toward a Unified RMA Model*. Springer US, Boston, MA, 1379–1391.

[31] Min Si, Antonio J. Peña, Pavan Balaji, Masamichi Takagi, and Yutaka Ishikawa. 2014. MT-MPI: Multithreaded MPI for Many-Core Environments *(ICS '14)*. Association for Computing Machinery, New York, NY, USA, 125–134. https://doi.org/10.1145/2597652.2597658

[32] Berkeley Laboratory Software. 2020. CSPACER: Consistent SPACE Runtime. https://bitbucket.org/kibrahim/cspacer_xc/

[33] Edgar Solomonik and James Demmel. 2011. Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms. In *Euro-Par 2011 Parallel Processing*, Emmanuel Jeannot, Raymond Namyst, and Jean Roman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 90–109.

[34] Rajeev Thakur and William D. Gropp. 2003. Improving the Performance of Collective Operations in MPICH. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Jack Dongarra, Domenico Laforenza, and Salvatore Orlando (Eds.). 257–267.

[35] US Lattice Quantum Chromodynamics. [n.d.]. https://www.usqcd.org/index.html.

[36] Robert A. van de Geijn and Jerrell Watts. 1995. *SUMMA: Scalable Universal Matrix Multiplication Algorithm.* Technical Report. University of Texas at Austin, USA.

[37] Hans Weeks, Matthew G. F. Dosanjh, Patrick G. Bridges, and Ryan E. Grant. 2016. SHMEM-MT: A Benchmark Suite for Assessing Multi-threaded SHMEM Performance. In *OpenSHMEM and Related Technologies. Enhancing OpenSHMEM for Hybrid Environments*, Manjunath Gorentla Venkata, Neena Imam, Swaroop Pophale, and Tiffany M. Mintz (Eds.). 227–231.

[38] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. 2007. Productivity and Performance Using Partitioned Global Address Space Languages. In *The 2007 International Workshop on Parallel Symbolic Computation*. 24–32.

[39] R. Zambre, A. Chandramowlishwaran, and P. Balaji. 2018. Scalable Communication Endpoints for MPI+Threads Applications. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. 803–812.

[40] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. 2014. UPC++: A PGAS Extension for C++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 1105–1114.