Evaluating Collectives in Networks of Multicore/Two-level Reduction

U.S. Wickramasinghe*, Luke D'Alessandro*, Andrew Lumsdaine†, Ezra Kissel*, Martin Swany* and Ryan Newton*
*Indiana University, Bloomington, USA
Email: *{uswickra,ldalessa,ezkissel,swany,rrnewton}@indiana.edu
†Pacific Northwest National Laboratory, Seattle, USA
Email: †andrew.lumsdaine@pnnl.gov

Abstract—**As clusters of multicore nodes become the standard platform for HPC, programmers are adopting approaches that combine multicore programming (e.g., OpenMP) for on-node parallelism with MPI for inter-node parallelism—the so-called "MPI+X". In important use cases, such as reductions, this hybrid approach can necessitate a scalability-limiting sequence of independent parallel operations, one for each paradigm. For example, MPI+OpenMP typically performs a global parallel reduction by first performing a local OpenMP reduction, followed by an MPI reduction across the nodes. If the local reductions are not well-balanced, which can happen in the case of irregular or dynamic adaptive applications, the scalability of the overall reduction operation becomes limited. In this paper we study the empirical and theoretical impact of imbalanced reductions on two different execution models: MPI+X and AMT (Asynchronous Many Tasking), with MPI+OpenMP and HPX-5 as concrete instances of these respective models. We explore several approaches of maximizing asynchrony with MPI+OpenMP, including using OpenMP tasking, as well as the case of MPI only, detaching X altogether. We study the effects of imbalanced reductions for microbenchmarks and for the Lulesh mini-app.Despite maximizing MPI+OpenMP asynchrony, we find that as scale and noise increases, scalability of the MPI+X model is significantly reduced compared to the AMT model.**

I. Introduction

The standard HPC platform today is a cluster of multicore nodes (perhaps also including GPUs). Historically (and perhaps obviously), programmers have used shared memory approaches for parallel programming of multicore machines and have used distributed memory approaches (aka MPI) for programming clusters. Thus, the obvious approach for programming clusters of multicore machines is to marry the two approaches that have separately worked so well in the shared and distributed memory worlds. The general moniker for the resulting combination is "MPI+X" to reflect the fact that there are a multiplicity of shared memory approaches (but only one MPI).

Of course the expectation (or at least the hope) is that the effect of "MPI+X" will provide the compounded benefits of each and enable scalability on today's largest machines as well as into future exascale machines. The problem with MPI+X, as has been famously noted, is in the "+"[1]. That is, there are numerous problems in combining two separate parallel programming paradigms as each carries its own interface, runtime system, and high-performance programming idioms. It is unlikely to expect independent approaches to simply compose.

In this paper we study one important use case in parallel programming, namely global reduction, and investigate the impact of the "+" in MPI+X — in our case we focus on MPI+OpenMP in particular. For MPI+OpenMP, a global parallel reduction can be performed by first performing a local OpenMP reduction, followed by an MPI reduction across the nodes. However, this approach imposes a serialization (albeit a coarse one) of the operations in the parallel reduction — i.e., it requires a reduction of the local variables followed by a further reduction over those intermediate values. Such a coarse serialization may not appear to be detrimental and, as the obvious approach presented by the two systems, would also seem to be the best possible approach. However, if the local reductions are not well-balanced, which can happen in the case of irregular or dynamic adaptive applications, this serialization can cause problems and limit the scalability of the overall reduction operation.

Collective communication is known to propagate and even amplify noise effects and numerous studies have been conducted on the effects of external noise [1]–[5] on application scalability and the propagation of delays in the face of collective communication or global synchronization barriers. With MPI+X, there is a necessary sequence of operations (local plus global) to realize a single compound operation. In that case, the effect of computational irregularity becomes isomorphic to that of system noise – but potentially orders of magnitude larger. Given the importance of irregular and dynamic adaptive applications, it is important to understand the effects of MPI+X on the scalability of global reduction operations (and ultimately other collectives).

One approach to ameliorating the effect of noise on collective operations is to make the collective operation non-blocking. This becomes problematic with the MPI+X approach because only the MPI collective operation is readily transformable into a non-blocking operation. That is, only the second half of the compound operation can be overlapped with other work – with the straightforward implementation the local portion is not overlapped. Using a more sophisticated – and complicated – approach to asynchrony allows the local work to

[1]Quote attributed to Bill Gropp

also be overlapped. However in this case, we are moving away from strictly MPI+X to implementation of another paradigm using MPI+X, namely AMT, which we discuss next.

Asynchronous Many Tasking (AMT) is an alternate approach to MPI+X for programming clusters of multicore systems. The basic paradigm of AMT is to expose and exploit maximum parallelism through large numbers of lightweight threads. Moreover, existing AMT systems such as Charm++ [6], OCR [7], HPX-5 [8], and Legion [9] support shared and distributed memory with a single runtime, programming model, and paradigm. Although AMT is still a topic of active research, and these systems are still under active development, they do show promise for improved scalability, particularly for irregular applications.

In this paper we study the empirical and theoretical impact of imbalanced reductions on two different execution models: MPI+X and AMT, with MPI+OpenMP and HPX-5 as concrete instances of these respective models. We explore several approaches of maximizing asynchrony with MPI+OpenMP, including using OpenMP tasking. We study the effects of imbalanced reductions for microbenchmarks and on synthetically noise injected Lulesh [10] mini-app. Despite maximizing MPI+OpenMP asynchrony, we find that as scale and noise increases, scalability of the MPI+X model is significantly reduced compared to the AMT model.

This paper makes the following contributions:

- We implement and evaluate a portable framework and an API, to profile and instrument load variation with various distributions into parallel regions of an distributed memory application on different runtimes.
- We implement a high performance unified collective interface on a representative AMT called HPX-5.
- We empirically analyze effects of load variation on multiple runtime execution models namely, MPI+OpenMP, MPI and our AMT instance using a tunable collective microbenchmark.
- We analyze mini-app Lulesh, by injecting load at various points and on various runtime conditions and results compared with MPI+OpenMP and AMT implementation of Lulesh.

The rest of the paper is organized as follows. Section III, depicts a deeper look at the problem at hand. Section IV, we consider the HPX-5 runtime and its collective implementation. We discuss the overall design and features offered under its asynchronous collectives framework. In Section V, we present the details of the load injection benchmark and discuss the usage scenarios subjected under three runtime systems. Then, in Section VI, we study the load injected Lulesh application and benchmarks which simulate performance characteristics of 2 phase reduction under irregular workload.

II. Related Work

Hoefler et al. have conducted a detailed analysis on impact of external noise effects on communication synchronization. These effects include operating system noise [3] and network noise [1]. Studies such as [5] further shed light on modeling noise to acquire analytical perspective to explore the effect of noise on scalability of collective operations. Ferreira, Bridges, and Brightwell [3] use noise-injection techniques to assess the impact of noise on several large scale applications using extremely lightweight kernels. Beckman et al. [4] characterized sources of noise (both internal and external) and analyzed the performance on BlueGene/L systems, using a synthetic noise injecting benchmark known as 'selfish detour' benchmark.

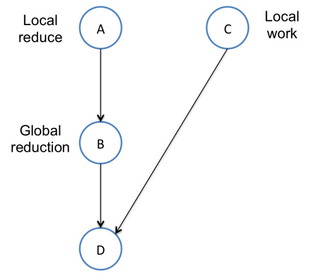


Fig. 1: **General outline for a common computation graph found in HPC applications**

Research work such as [11]–[14], report on MPI+OpenMP usage patterns, how they can be applied to existing applications and possible challenges that may be encountered. Based on this evidence only a handful of hybrid execution patterns have deemed to be successful in practice mainly due to the inherent integration complexity and performance implications across the two runtime system boundaries. The dynamic load balancing potential of MPI+OpenMP runtime execution model has regularly been a subject of interest. Notably, Tafti et al. [15], [16] have reported its early adoption on AMR based irregular applications. More recently newer AMT runtimes [17] have grown in popularity for tackling such large scale irregular problems.

III. Background

```
compute_region() {
  while (some_condition()){
    #pragma omp parallel
    {
       //execute shared memory parallel region
    }
    //global reduction
    MPI_Allreduce()
  }
}
```

Listing 1: **General MPI+OpenMP pattern for a two-phase reduction**

Listing 2 and Figure 1 report a very simple but commonly found execution pattern of a two-phase reduction barrier in MPI+OpenMP programming model. One of the limitations of executing such a program is the strict ordering of the local reduction phase—using a *fork-join* model of parallel execution, which is followed by a global synchronization primitive such as a collective operation. Figure 1 further illustrates this limitation more thoroughly. The data flow graph depicts Independent regions **A** and **C** (ie:- no directed edge) and regions **B, D** as dependent. Region **B** relies on the output of region **A** and then region **D** on both **B** and **C**. For irregular load conditions, it would be especially beneficial to overlap work of region **C** with **B**. However the implicit synchronization barrier presents a limiting factor that makes it impossible to hide irregularities in region **A**. Therefore naive MPI+OpenMP model of programming can make it difficult to fully utilize available processing resources for applications with similar parallel data dependency characteristics. Newer Implementations of (OpenMP version 3.0 and on) have attempted to mitigate these issues by embedding dynamic loop scheduling and task parallelism techniques, for example using latest additions to programming constructs like `pragma omp task`, `pragma omp sections` and nested regions. However introducing these newer constructs to applications and transforming them have become a matter of software complexity. Issues of effectively controlling nested parallelism and obscure details of performance tuning across hybrid runtime boundaries may become *hard* problems and a matter of concern for many *hybrid* application developers.

AMTs are the newer breed of distributed shared memory runtime systems that have had a significant influence on dataflow driven parallel programming. We contend that AMTs provide a uniform approach to collectives even under uneven load conditions. This is largely due to the asynchronous design of AMT runtimes. For example considering the information in Figure 1, AMTs can effectively overlap communication and computation of region (**A, B**) and combine them with region **C**, thus avoiding wait time for any costly intermediate synchronization steps and increasing throughput. Threads, in terms of early finishers, can compensate for late comers by taking up more work while waiting for a collective communication operation to complete. Features such as Active Messages [18], over subscription, and global address spaces can deliver AMT runtimes with additional options to balance load by utilizing latency hiding and pipelining. Such runtimes can provide collective communication primitives which may be at least theoretically as efficient as any MPI implementation available. Moreover, AMTs such as CILK and TBB have shown comparable if not better SMP performance than OpenMP. More important to application developers, however, is that AMTs are by design a unified programming model, and so they avoid poor cross-runtime usability and performance issues that can be present in hybrid execution models.

IV. HPX-5

For this work we have selected the HPX-5 exascale runtime as our representative adaptive multithreaded runtime. HPX-5 is based on the ParalleX execution model [19]. HPX-5 applications are written as diffusing shared memory programs where threads explicitly send active messages to global addresses (GAS) where they become new lightweight threads. Threads may block on globally allocated local synchronization objects (e.g., futures, dataflow, etc) for control and data synchronization, and may also perform non-blocking memory transfers (puts and gets) with the GAS directly. This model of execution permits the runtime to tolerate latency through concurrency within and across threads. The reference implementation of HPX-5 [20] implements a conventional work stealing scheduler [21] for local lightweight thread scheduling, a high performance Partitioned GAS (PGAS) for active message addressing and RDMA operations, and uses the Photon RDMA library [22] for network transport.

Importantly, for our purposes on HPX-5, we implemented a non-blocking collectives interface that operates at the lightweight thread level. As with MPI, threads interact with the collective through two phases, first joining the collective and then later testing the collective for completion.[2] This allows threads to overlap collective communication with computation and tolerate latency and irregularity.

Unlike MPI, there is no limit to the number of lightweight threads participating in a locality, nor is there any external synchronization required. HPX-5 collective scheduling is naturally integrated into the HPX-5 runtime. This unified behavior eliminates model-imposed barriers that are fundamental to all MPI+X instantiations, and will be shown in Section VI to be superior for tolerating the noise and irregular behavior expected in exascale systems.

The HPX-5 reference process based allreduce implementation is equipped with local collectives combined with number of virtual network typologies (binary, binomial trees, hierarchical, etc) for for efficient global communication.

A. A Collective Communication Framework

The HPX-5 programming model envelops the necessary constructs required for developing a communication framework for collectives. In the following sections we provide an overview of the HPX-5 programming constructs and refer to the primary API's that are required in this context.

1) Task Parallelism in Synchronous Domains: Intra-node level task parallelism can be achieved in a number of different ways within the HPX-5 runtime. A work segment or a compute task generally is enveloped by an HPX-5 action. Therefore before any task is executed by the runtime, user or library needs to register their respective actions. This is achieved via the HPX-5 action registration interface either statically or dynamically at runtime. Another consideration that needs to be taken into account is the locality, or global address, at which the set of tasks are spawned. Identifier `HPX_HERE` indicates that tasks are spawned in a respective action's local domain. In other cases, a parent task must ensure that the virtual address range it signals for spwawning belongs to its local synchronous domain.

In general, all `hpx_*_call_()` API invocations causes a parent action to spawn new parallel actions asynchronously. A LCO synchronization object is passed onto the function call as a parameter, except in the situations where the synchronous variation of an interface is invoked. This LCO object will act

[2]In fact, collectives in HPX-5 are data-driven and not execution driven. The identity of the joining threads is inconsequential, and the completion of a collective operation triggers a set of registered continuations. This style of operation is consistent with ParalleX and can encode arbitrarily complex dataflow.

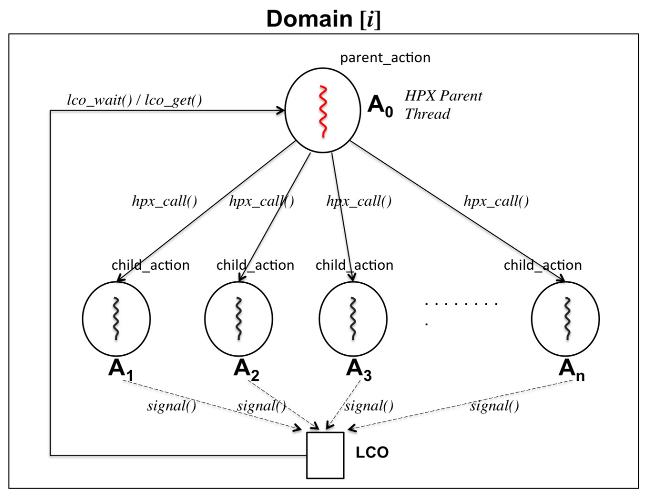


Fig. 2: HPX-5 Parent action spawns *n* parallel tasks and waits on a LCO.

as a handle for some action, often the parent task, which performs a join operation that may cause it to wait until all spwaned child tasks created have terminated. The code segment in Figure 2 illustrates this concept where a parent action A_0 spawns $A_1, A_2...A_n$ actions locally and A_0 waits until all child actions have exited.

HPX-5 includes the ability to create a logical *and* LCO which provides safe access to producer and consumer (i.e., the parent in this case) actions to signal the completion of a parallel task. The `hpx_lco_wait` clause provides an association within the *and* LCO, which will cause a wait for *n* number of inputs to complete. The scheduler is expected to yield the calling threads at `hpx_lco_wait` and only wake up when the values are ready to be read, providing opportunity for other tasks to do useful work. Intra-node task parallelism can also be achieved by the HPX `hpx_par_for` and `hpx_par_call` interfaces. These calls are more suited to parallelize *for* loops and nested *for* loops that occur commonly in many matrix multiplication and linear algebra applications .

```
//main action
parent_action (..){
..
..
  hpx_addr_t done = hpx_lco_and_new(size);

  for (int i = 0; i < size; i++) {
    hpx_call(HPX_HERE, _child_action,..,
             done, args, ..);
  }
  hpx_lco_wait(done);
  hpx_lco_delete(done, HPX_NULL);
..
..
}
```

Listing 2: **HPX-5 C code for parallel invocation of an *Action* named _child_action.**

2) Communicating Domains: An Efficient inter-node communication strategy is essential to realize a collectives framework for a task parallel runtime. HPX-5 exhibits diversity in this area in terms of unifying task level parallelism inside a node and facilitating communicating domains by introducing both native blocking and non blocking message passing methods in addition to as MPI-style SPMD communication across nodes.

HPX-5 comprises a rich set of interfaces that can be used for general point-to-point communication between domains. HPX-5 also supports fully synchronous, locally synchronous, and fully asynchronous interfaces. In terms of locally synchronous API's, HPX-5 provides local completion semantics where the ownership of local buffers is released back to the user at the end of a successful invocation.

The communication API consists of several categories, namely a) remote procedure call interface for point to point communication b) parallel loop interface for direct parallel actions c) process level interface for ParalleX process creation, management, and communication d) collectives interface for group communication d) Remote Memory Access (RMA) based interface for direct GAS interactions using *put/get*, and e) low-level network interface for fine grained communication.

We do not intend to discuss all interfaces in this paper, however, Table I lists those that are relevant to this work. All asynchronous and partially synchronous API's require an LCO to be passed into a action call API. Here, an LCO acts as a handle to *wait* or *test* until sufficient progress is made to acquire the result of the action(s). Some of the API's use continuations as a versatile method to enforce synchronization constraints such as to expose *wait until* and *continue after* semantics in communication. HPX-5 uses a *thread_continue* interface within a running action to pass on the output value to a desired continuation. A continuation will always be associated with a particular domain along with an action to be executed.

3) Collective Operations: The HPX-5 collective framework is built upon three categories of collective types supporting both intra-node and inter-node levels. These can be categorized into the following, a) LCO based collectives, b) process level collectives, and c) network level collectives. As briefly described above, a process in HPX-5 encompasses a nested group of child processes or threads operating within their distributed memory space. Each process associates itself with a termination group which is analogous to a communicator in MPI/SPMD and can be waited or tested for process termination through a termination LCO. All HPX-5 inter-node collectives created under this level of abstraction falls into the process level category. Network level collectives operate at the low level transport layer and follows more MPI-like SPMD model for group communication. We first discuss synchronization of LCO based collectives and then move into the native process based and combined hierarchical approach of inter-node and intra-node collective model.

Remote Procedure Calls	**Blocking**	**Description**
hpx_call	false	Locally synchronous send
hpx_call_sync	true	Fully synchronous send
hpx_call_async	false	Fully asynchronous send
hpx_call_with_continuation	false	Locally synchronous send with continuation action
hpx_call_when	false	Locally synchronous send executed dependent on an LCO control signal
Parallel Loop Calls	**Blocking**	**Description**
hpx_par_for	false	Locally synchronous parallel for loop
hpx_par_for_sync	true	Fully synchronous parallel for loop
hpx_par_call	false	Locally synchronous generic parallel loop
LCO-Collective Calls	**Blocking**	**Description**
hpx_lco_reduce_new	true	Returns a synchronized LCO object for reduce
hpx_lco_allreduce_new	true	Returns a synchronized LCO object for allreduce
hpx_lco_alltoall_new	true	Returns a synchronized LCO object for alltoall
hpx_lco_allgather_new	true	Returns a synchronized LCO object for allgather
Process-Collective Calls	**Blocking**	**Description**
hpx_process_collective_subscribe	false	Start a subscription phase to signal participation in a collective group
hpx_process_collective_finalize	false	Stop subscription phase and prepare for group communication
hpx_process_broadcast	false	Start a domain wide broadcast w.r.t. process/collective group
hpx_process_allreduce	False	Start a domain wide allreduce w.r.t. process/collective group
Network-Collective Calls	**Blocking**	**Description**
coll_init	true	Initialize collective communication group on network level
coll_sync	true	Fully synchronous collective call on network
coll_async	false	Fully asynchronous collective call on network

TABLE I: List of HPX-5 API's for collectives and their functions.

```
main_action(){
...
  allreduce = hpx_lco_allreduce_new(inputs, outputs,
                                    ...,..., rop);
  sum_result = hpx_lco_reduce_new(n,...,..., rop);

  //call allreduce join
  hpx_call(.., join_synchronous_action,...,
           &sum_result,.. )
}

join_synchronous_action(hpx_addr_t allreduce,
                        int id, int input,
                        hpx_addr_t sum_lco) {
  int result;
  //do the actual join here
  hpx_lco_allreduce_join_sync(allreduce, id,
                              sizeof(input),
                              &input, &result);

  //set lco to result
  hpx_call_cc(sum_lco, hpx_lco_set_action,
              ..., &result, sizeof(result));
}

join_asynchronous_action(hpx_addr_t allreduce,
                         int id, int input,
                         hpx_addr_t sum_lco) {
  int r;
  //create future to hold allreduce sum
  hpx_addr_t f = hpx_lco_future_new(0);
  //we pass a future for async join
  hpx_lco_allreduce_join_async(allreduce, id,
                               sizeof(input),
                               &input, &r, f);

  //wait for value to be available
  hpx_lco_wait(f);
  hpx_lco_delete(f, ...);

 //set lco to result
  hpx_call_cc(sum_lco, hpx_lco_set_action,
              ..., &result, sizeof(result));
}
```

Listing 3: **HPX-5 C code for LCO based collective invocation**

Listing 2 illustrates a simple LCO based allreduce collective in both synchronous and asynchronous forms. As mentioned in Section 2, an important distinction between the two methods is that the latter will always provide some synchronization LCO for lazy evaluation of the result while the former will block the invocation until acquisition of the final result is made available. In this example the main action initiates an `allreduce` LCO with required arguments. The arguments include number of producers who will continue to input values, consumers who will eventually read the reduced values and a pointer to a *reduction* operation that will applied at the point of *joining* of the input values. HPX-5 supports custom commutative and associative reduction operators for this purpose. Once the collective has been initialized, users may proceed with a collective *join* operation of choice such as illustrated by `join_synchronous_action` and `join_asynchronous_action` methods. The allreduce LCO object will be responsible for synchronization and execution of the reduction.

Allreduce LCO and other LCOs are synchronized state machines that interact with the HPX-5 scheduler and thread runtime. This mode of operation is very similar to traditional monitors and semaphores used in other systems. By default it supports *signal_wait* semantics where producers are being waited until all inputs are ready. And result is propagated once all consumers are available to read them. The state of all reduce LCO objects transits between **REDUCE** and **SIGNAL** phases during synchronization phases. Furthermore, should the need arise to propagate allreduce result to other actions, HPX-5 provides auxiliary mechanisms such as *reduce* and *future* LCO's as in the case of `sum_result` in Listing 2. This is another instance where data flow programming constructs in

HPX-5 can be useful for application routines.

However, it should be noted that this mode of collectives is not always *scalable*. The overhead introduced by *monitor* and *lock* based synchronization, and the sequential bottleneck on a single reducer, are significant when the number of domains and task parallelism increases. One of the new contributions HPX-5 provides in the collective communication space is the process based collective paradigm. Figure 3 illustrates a process based allreduce example which encapsulate this concept.

```c
main_action(){
  ...
  allreduce = hpx_process_collective_allreduce_new(
                   data_size,..., rop);
  sum_result = hpx_lco_reduce_new(HPX_LOCALITIES,
                   ...,..., rop);

  //call allreduce subscribe
  for (int i = 0; i < HPX_LOCALITIES; ++i) {
    hpx_call(..., subscribe_action,...,
             &allreduce);
  }
  ...
  //call allreduce join
  for (int i = 0; i < HPX_LOCALITIES; ++i) {
    hpx_call(..., reduce_Nblcoks, reduce,
             &allreduce);
  }
  ...
  //get result
   hpx_lco_get_reset(reduce,.., &result);
}

static int subscribe_action(...,
                      hpx_addr_t allreduce) {
  for (int i = 0; i < N; ++i) {
    ...
    future f;
    int local_id =
        hpx_process_collective_subscribe(
             allreduce, hpx_lco_set_action, f);
  }
  return HPX_SUCCESS;
}

static int reduce_action(...,
             hpx_addr_t allreduce,
             input_value) {
  hpx_process_collective_allreduce_join(
       allreduce,..., &input_value);
  hpx_lco_get_reset(future,..., &result);
  HPX_THREAD_CONTINUE(result);
}
```

Listing 4: **HPX-5 C code for process based collective invocation, both synchronous and asynchronous classes of API usage is shown**

HPX-5 collective model integrates an intra-node accumulation stage followed by a network level reduction stage for process level invocation. As illustrated by Figure 4 the main action should initiate `hpx_process_collective_allreduce_new` or similar interface to instantiate the group communication constructs. This preparation step is necessary to contextualize information required to save the collective operational state during the run. Then the subscribe

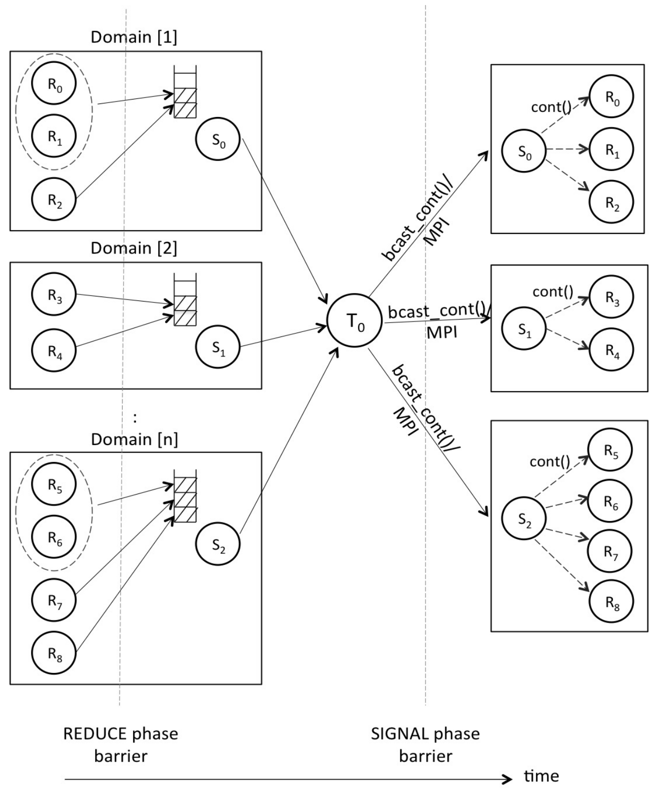


Fig. 3: Organization of a process level collective operation. A domain may be oversubscribed (indicated by the groups surrounded by dashed circle).

step is used to signal to a parent network node that a particular domain has been registered under it. This is particularly important since we eventually perform a **reduce-broadcast** type operation for allreduce.

Results are communicated by extending continuations on each leaf domain and parent network domain at the subscription phase. For example, each leaf domain will contain user level *continuations* such as *futures* to propagate the result back to the callee action, while the parent network domain will contain a *continuation* for each leaf node to broadcast inter-domain join output to its children. The subscription phase is synchronized such that no two subscribers from the same domain is registered twice under the same parent. Although this synchronization is a requirement, it must be invoked once for a respective collective invocation, thus resulting in minimal overhead if the same collective is repeatedly reused.

Once the reduction phase starts, each domain action will invoke its local join operation using `hpx_process_collective_allreduce_join`, which is executed under action `reduce_action`. Each of the local reductions will take place in a lock free environment using thread local buffers. When all input related to a domain has been made available to a reducer, and the final result is ready, the reduced value is communicated to the parent which then performs the final reduction. Finally, using registered continuations (e.g., at subscribe phase) domain node(s) would broadcast the final result back to leaf domains which will in turn propagate values to any application using user continuations. This workflow of a process level reduction/join is detailed in Figure 3 with leaf domains indicated by $R_0, R_1 ... R_i$ and phase domains indicated by $S_0, S_1 ... S_j$. The parent phase domain T_0 will contain the final join result after the output of all join operations are synchronized.

The process level local domain reduction phase avoids using LCO synchronization altogether, instead it relies on thread local buffers to store the partial reductions. This is useful since synchronization overhead due to lock contention when a domain is oversubscribed with multiple workers trying to access the same shared buffer would be significant. Here, more importantly, we introduce two implementation strategies for inter-node collective communication for an asynchronous task runtime: i) A continuation driven hierarchical collective implementation, and ii) a direct collective call using an existing library supporting network-optimized collective communication. Option (i) is the more native approach to HPX-5 design and its programming constructs, while in (ii) we use a more traditional approach that stems from inter-process joins on SPMD clusters.

In the continuation-based hierarchical collective implementation we systematically extend each continuation action from local domains to remote domains using native HPX-5 parcels. Each parcel will contain a destination address (i.e, a future or a GAS target), an action (`hpx_lco_set_action`) and data primarily encoding the reduction value and its state. As shown in Listing 3, the point at which this happens is dictated by invocation of the `hpx_process_collective_subscribe` call. This call provides users a subscription window to register user and network level continuations until a corresponding join is invoked. The continuation hierarchy can be considered as a simplified overlay network where input values for a reduction, join or broadcast values will be disseminated up the network and computed results are pushed downwards depending on the nature of the collective. Within the current reference implementation we have implemented single level hierarchiesto other Multi-level hierachical networks such as N-ary tree, binomial tree. Other topological in the likes of dissemination and "Bruck" are in the experimentation phase.

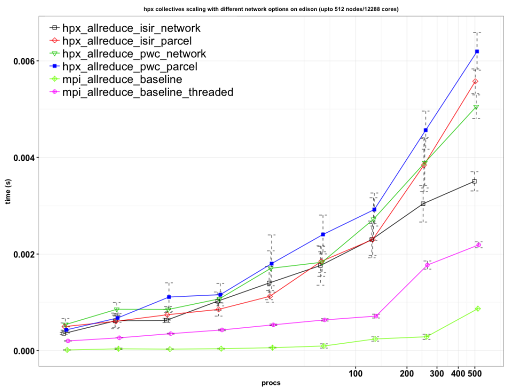


Fig. 4: **Microbenchmark performance for HPX-5 Allreduce with different HPX-5 specific implementations (parcel, network), on Edison cluster (512 nodes/12000+ cores)**

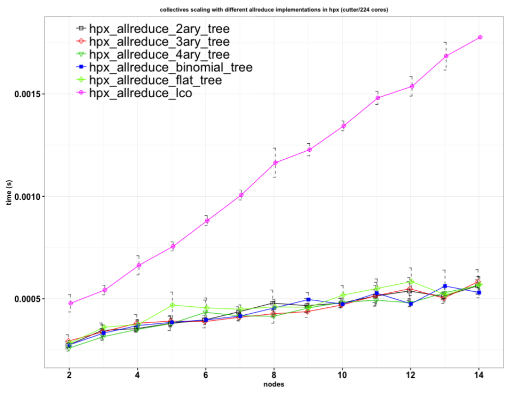


Fig. 5: **Microbenchmark performance for HPX-5 Allreduce with different HPX-5 parcel implementations (flat tree, N-ary tree, binomial trees), on Cutter cluster (14 nodes/224 cores)**

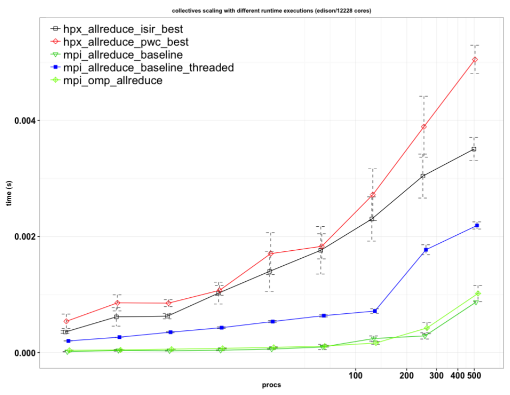


Fig. 6: **Microbenchmark performance for HPX-5 , MPI, MPI+X Allreduce on Edison cluster (512 nodes/12000+ cores)**

B. Network Collective Implementation

Process based collectives by default use our native parcels based implementation to create a continuation tree. Thus drawing back from the current state of the network transports available for HPX-5, users can decide to deploy two different variations of collective runtimes, namely 1) **PWC** network based collectives 2) **ISIR** network based collectives. Either one of the mode can be enabled at HPX-5 compile time by using the switches, **-enable-photon** or **-enable-mpi** respectively. This does not however affect application level collective interfaces in anyway and hence users can transparently decide between the 2 implementations, depending on the performance characteristics of the system.

HPX-5 traditionally supported point-to-point communication via its low-level network interfaces. These interfaces are well defined in 'xport_mpi.h' and 'xport_photon.h' headers. Beside the apparent use of point to point in HPX-5, these were also used as vehicles for collective communication. For example native process based collectives has used point-to-point interfaces underneath, albeit being the general modus of operand for the runtime. We need to emphasize that point to point communication is no less powerful than a collective interface, in fact is the inverse - efficient point to point communication is the basis for a robust collective framework. However our expectation is that runtime system may be able to use optimized transports from network (ie:- efficient transports from MPI and other RDMA libraries such as Photon) and reduce any overhead involved.

Thus being true to this fact, we have implemented a new native collective transport interface that will support inter collectives natively at transport layer. HPX-5 network interface defines three basic methods for network layer collectives on 2 network transports namely, **MPI** and **Photon**. For **MPI**, we used underlying *MPI_** collective interfaces while for **Photon**, we implemented native collective library using **LibNBC** (Non blocking collective library used in OpenMPI) as the substrate. As listed on Table 1 these include interfaces to initialize collectives and perform both synchronous and asynchronous collective patterns. Figure 4 report the performance evaluation of *Allreduce* operation with different HPX-5 collective transports available. This includes network level collectives discussed in this section and pure parcel based implementation. Figure 5 evaluates LCO based collective implementation with other hierarchical parcel implementations. Figure 6 report the performance comparison of HPX-5 allreduce collective (we picked the best case here) with other runtime implementations such as **MPI** and **MPI+X**.

V. Emulating Load Imbalance

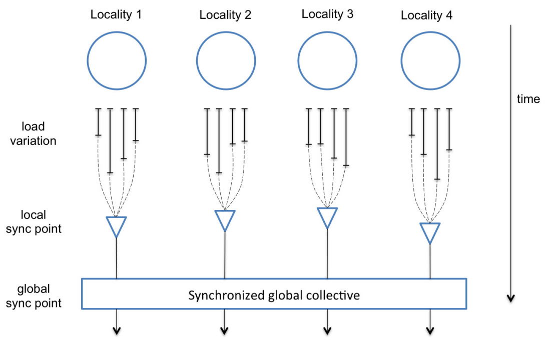


Fig. 7: **Benchmark overview for computation load injection**

To perform our study, we developed a framework that can inject various amounts of load into existing programs. It was important that this framework capture the core ideas we wanted to highlight in irregular applications, irrespective of the underlying runtime execution model. Our design is based on several criteria identified below.

- Enabling injection of load at varying amounts (amplitude) or conforming to a particular distribution
- Enabling injection of load at identified points - locality or light weight process of an distributed memory application
- Enabling portability across different runtime execution models such as MPI+OMP and HPX-5
- Enabling a lightweight profiler/tracer to identify regions of significance for load injection

We describe the implementation in detail in the following sections.

A. A Framework for Load Injection

Our framework uses the method of Fixed Work Quantum (FWQ) to inject and measure load across application regions. FWQ assumes that the minimum time t_u or *unit work* as we refer to it from this point on, represents the perfectly balanced execution of a program region. However all other times *unit work* can be perturbed by t_i - t_u or an *overhead* time which we will refer to as t_o. Earlier work [1], [3] used similar techniques in their noise injection benchmarks that emulate minuscule amounts of system noise. However unlike the measurement of external background noise in benchmarks, we do not try to keep work quantum t_u to a bare minimum since our benchmark will specifically try to mimic actual workload on real world applications. Therefore we have implemented load injection of a resolution that can be measured in anything from microseconds to any order of magnitude of seconds , and that can be directly correlated to constant factor of compute cycles.

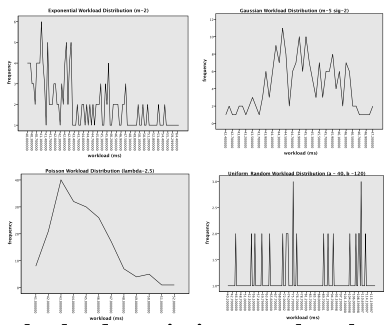


Fig. 8: **Sample load variation on benchmark 16-node infiniband cluster**

1) Input for Load Injection: Our benchmark allows input of basic load distribution properties at runtime using `--base` arguments which take unit work (t_u), amount of maximum overhead (t_u/t_o %) to inject as an percentage of unit work, number of threads on each locality to inject into (`tpn`) and a time or work resolution unit. Injected load can be emulated either using a sleep such as a high resolution system timer based `nanosleep()` system call or with regular spinning by cpu instructions. Although both modes are supported, In our study we argue that using regular spinning using no-op instructions is much more effective since cpu yielding methods such as `sleep` are susceptible to underlying operating system and runtime system optimizations that will interfere with the anticipated outcome. Moreover, idle cpu cores are highly capable of absorbing noise from other parts of the system [23], which may result in skewed measurements.

Figure 7 is an illustration of an overview of a load-injection framework and the base MPI+OpenMP benchmark work. In MPI+OpenMP we would like to emulate varying load at local reduction region or a `#pragma omp parallel` region, thus making sure we observe effects such as delays in parallel threads can propagate into global synchronization step. The offset triangle emphasizes the time difference between the end of the local reduction phase barrier and the start of the global synchronization point. Our benchmark essentially calculates workload assignment at the initialization stage and then transports the respective work assignment via `run_kernel(work_ledger_t* work)` interface. This interface abstraction provides each runtime system or application a mechanism to implement and emulate a irregular compute-heavy region.

Our benchmark was designed to support different load/noise distributions as well. The *uniform* injection mode acts as the perfectly load balanced base case. Other distributions allow load/noise variation by varying random distribution parameters for mean, standard deviation, etc. A *scaled* version of distribution will scale just one load assignment with an overhead by a specified percentage relative to unit work and all others will remain uniform. Others follow the statistical properties of their respective distributions and will depend on the input parameters (specified by `-r` $[a, b]$ for *uniform random* , `-g` (μ, σ) for *gaussian* , `-p` (λ^{-1}) for *poisson* and `-e` λ for *exponential* distribution). We depend on known algorithms such as Multiplicative LCG (Linear Congruential Generator) and Box Mueller transform for psuedo-random generation of distributions. Figure 8 report a scatter plot at injecting load at an MPI+OpenMP execution region, on a small 16 node cluster which emulates the effect of load variation under different random distributions.

2) Input for Overlapped Work: As illustrated by Figure 1, different runtime systems may enable *communication and computation overlap* to increase throughput by allowing independent work region **C** to be executed during network communication. that can be overlapped with core parallel computation region. Our framework allows emulating overlapping work by calling `run_overlapped_work(uint64_t qw, uint64_t ow)`. An overlapped work segment may present itself in 2 flavours, a) non parallalizable b) parallalizable. A non parallalizable overlapped work segment may have too many data dependencies such that any parallalization of respective code region is either impossible or impractical. We identify this as *sequential* overlap. Consequentially we identify a parallalizable work segment as *parallel* overlap.

Parameters for overlapped execution is input by arguments --mode [`mode`] [q_w] [o_w], which takes a mode switch, total overlapped region and overlapped work quantum, both relative to the unit work. The `mode` switch can enable different versions of overlap (*sequential* or *parallel*) on three runtime execution environments, HPX-5, MPI and MPI+OpenMP.

B. MPI+OpenMP Benchmark

Listing 5 report MPI+OpenMP benchmark with load being injected at the parallel region. The benchmark first calculates the load to be assigned to each thread spawned by initializing the load-injection framework at root rank. Then the assignment data are scattered to respective ranks using `MPI_Scatter()`. To ensure all simulated MPI processes start at the same time from the benchmark code region, we enforce an `MPI_Barrier()` based synchronization point just before timing starts and threads are spawned to emulate respective injected workloads.

```
run_kernel(){
  ...
  result = run_omp_kernel(work, ... )
  MPI_Iallreduce(&result, ... , &mpi_request)
  while(not done)
    MPI_Test(&mpi_request, &test, .. )
    test &= run_overlapped_work()
}
```

Listing 5: **MPI+OpenMP pseudo code example for overlapping parallel regions**

1) Overlapping parallel regions: This benchmark supports 3 modes of overlapped execution against both *sequential* and *parallel* (cf. Section V-A2) overlap segments, a) Regular parallel OpenMP regions b) OpenMP `sections` with nested regions and c) OpenMP `task` with nested regions. The first mode of overlapping does not require any special OpenMP constructs. With the implicit synchronization barriers at local parallel regions in MPI+OpenMP, it is not advisable to backfeed any extra (overlapped) work into a parallel region during the local reduction phase. Therefore, the default *modus operandi* for MPI+OpenMP is to overlap with global collective operation. This needs to be implemented with non-blocking versions of the MPI collective communication repertoire. However, most non-blocking MPI collective algorithms do require multiple iterations of network communication operations [24] to complete, and so they require the `MPI_Test` to be invoked progressively. Therefore as in *sequential* case, if an overlapped region is too large it may hinder the collective network progression and decrease overall throughput of the program. Listing 5 report the respective pseudo code.

```
run_kernel(){
  ...
#pragma omp parallel
  // all threads are falling through this line
  #pragma omp single
    // only one thread is here
    #pragma omp task
      result = run_omp_kernel(work, .. )
      MPI_Allreduce(&result, .. )
    #pragma omp task
      run_overlapped_work()
  // execute reduction+overlapped work
}
```

Listing 6: **MPI+OpenMP pseudo code for overlapped parallel region with OpenMP `task`**

```
run_kernel(){
  ...
#pragma omp sections
  // start execute reduction+overlapped work
  #pragma omp section
    result = run_omp_kernel(work, .. )
    MPI_Allreduce(&result, .. )
  #pragma omp section
    run_overlapped_work()
// other threads may branch around here
 ...
```

Listing 7: **MPI+OpenMP pseudo code for overlapped parallel region with OpenMP `section`**

OpenMP 3.0 parallel tasks and sections implementation enables execution of independent parallel regions by mutually exclusive teams of threads. Nested parallelism and *friends* are ably supported by modern OpenMP implementations including both **gcc** and **intel** compilers - although the native **intel** version widely regarded to have an edge. Listings 6, 7 present the the pseudo code for executing an overlapped kernel with either OpenMP `section` or OpenMP `task` region.

The difference between tasks and sections is subtle but lies in the time of execution of the parallel regions. Tasks will be queued and always will execute in a differed fashion, for example when a thread encounters an implicit or explicit synchronization point.[3] OpenMP `sections` however, will spawn only the required number of threads for the enclosed the sections construct and threads will not leave it until (unless specified by a `nowait` clause) all sections have been executed. For this reason, to allow parallel regions within them, we enabled nested parallelism using `omp_set_nested()` API or by setting env variable `OMP_NESTED` to TRUE.

```
run_kernel(){
  ...
#pragma omp parallel
  MPI_Iallreduce(&do_work(),&result, .. , &req[tid])
  run_overlapped_work()
  #pragma omp single
    // progress network
    MPI_Waitall(num_r, req, status);
  ...
```

Listing 8: **MPI benchmark pseudo code with MPI per thread communicator and parallel threads implemented by OpenMP**

C. MPI Benchmark

We developed an MPI benchmark for irregular workloads that performs the same 2-level hierarchical reduction task with overlap segments but without an OpenMP kernel. Thus MPI benchmark avoids a parallel region with an implicit synchronization barrier. The benchmark was implemented by creating a single parallel region after `MPI_Init_thread(MPI_THREAD_MULTIPLE)` that performed the reduction operation. Next, the overlapped segments were placed on the parallel region either in `omp single` or all parallel regions, depending on overlapped segment at hand is either *sequential* or *parallel*, respectively. Since logical divisions within a communicator is impossible with current MPI standard, our design required a per threaded communicator for reduction operations. However this design can be supplemented by proposed MPI-4 endpoints [25] proposal if it is incorporated into the standard in the future. Listing 8 report the pseudo code for aforementioned benchmark.

[3]`omp pragma parallel` region is an implicit synchronization point in Listing 6

D. HPX-5 Benchmark

In HPX-5 all the functionality of a 2-phase reduction can be encompassed by a single collective invocation(cf. Section IV). As an example for allreduce operation the corresponding invocation is the process based collective interface `hpx-_process_collective_allreduce_join` [20].

```
struct elem[
  int id
  long work
  long overlap]

// main parallel reduction action
hpx_work_action(elem, allreduce){
  hpx_process_collective_allreduce_join(
    do_work(elem->id), allreduce, ...)
}
run_kernel_action(elem_addr[], N, allreduce) {
  // Create an array of futures for asynchronous
  // threads for this iteration.
  Future sync[N + 1]
  for ( i = 0 --> N )
    // Reduce asynchronously across the local work
    // (reduce joins the collective)
    sync[i] = hpx_call(elem_addr[i],  hpx_work_action,
                       &allreduce)
  // Post the asynchronous overlaped work.
  sync[N]=hpx_call(HPX_HERE, overlap, and, elem_addr)
  // Wait for this iteration to complete.
  hpx_wait_all(sync)
}
```

Listing 9: **HPX-5 inspired pseudo code shows the parallel execution of 2 phase reduction operation and overlapped work region.**

HPX-5 benchmark start with the initialization phase at `root` locality by initializing the load-injection framework and then performing a asynchronous broadcast operation with the use of `hpx_call()` interface. Overlapping work is a trivial task in HPX-5 due to the asynchronous nature of operations. HPX-5 will spawn a single or multiple threads for the overlapped work depending on overlapped segment is either *sequential* or *parallel*, respectively. Listing 9[4] report the pseudo code for the benchmark. HPX-5 action `reduce` correspond to the local/global reduction region while `overlap` action refers to the parallel independent work region. HPX-5 **Futures** are used to detect the termination of all work regions.We also collect timing data by writing directly to a shared memory region on locality `root` from all other localities (including `root` itself) using one sided operation `hpx_gas_memput_rsync()`.

VI. Results and Discussion

We performed a number of synthetic benchmarks with varying, scale and load injection characteristics to investigate effect of outliers to a global reduction. Apart from the synthetic benchmarks, we choose a bulk synchronous parallel application, LULESH to test load injection. All benchmarks were carried out on a small scale HPC cluster "Cutter" at IU (**gcc**

[4]The actual HPX-5 version requires more lines of code but performs exactly this parallelization

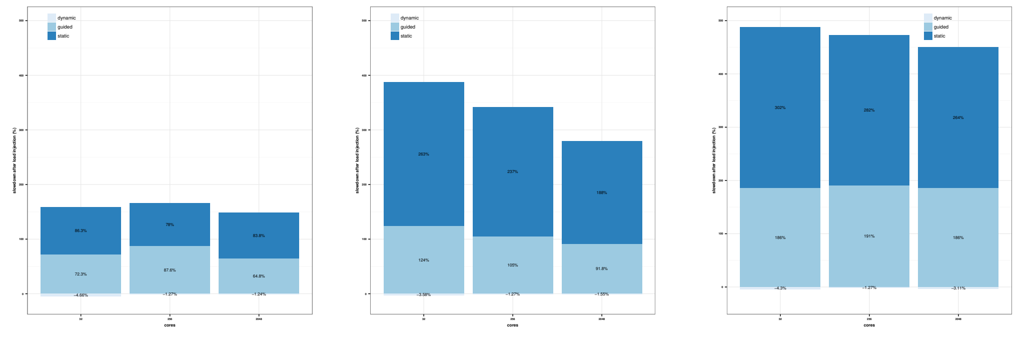
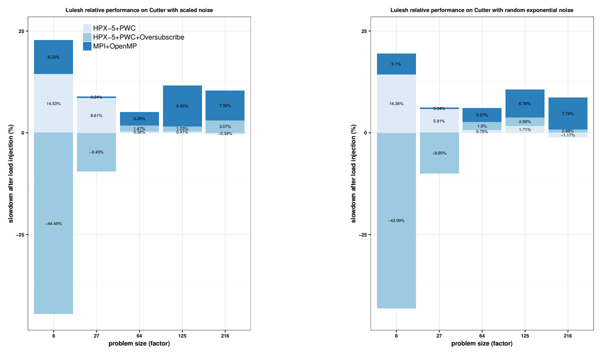

(a) uniform (b) scaled (c) exponential

Fig. 9: **Lulesh MPI+OMP version relative performance (slowdown) with different noise Injection and *OMP par for* loop scheduling methods on Cori (upto 2048 cores)**

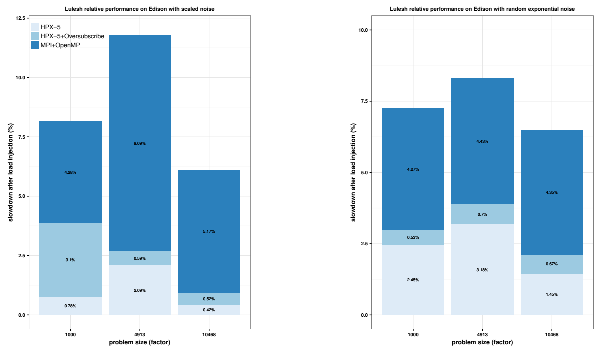
Region ID	Function Name	Average Time (%)
5	`CalcHourglassControlForElems`	45.20
3	`CalcFBHourglassForceForElems`	34.65
1	`InitStressTermsForElems`	14.39
12	`CalcKinematicsForElems`	0.80
2	`IntegrateStressForElems`	0.78
13	`CalcLagrangeElements`	0.60

TABLE II: **List of functions with largest time spent (descending order) on *OMP par for* regions in Lulesh MPI+OMP version**



(a) scaled (b) exponential

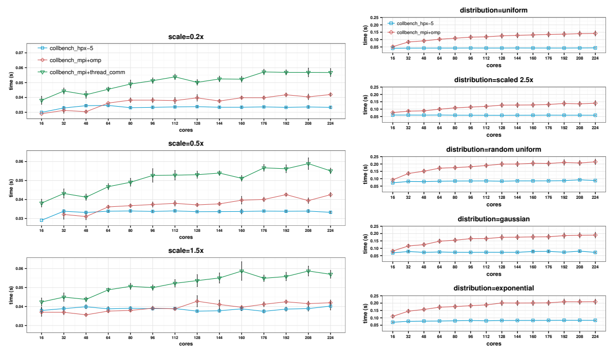
Fig. 10: **Lulesh Slowdown on Cutter cluster under scaled and exponential noise injection (upto 224 cores)**



(a) scaled (b) exponential

Fig. 11: **Lulesh Slowdown on Edison cluster under scaled and exponential noise injection (upto 12000+ cores)**

/ **open-mpi** *env*) and on a large scale HPC clusters "Edison", "Cori" (**intel / cray-mpich** *env*) on NERSC Berkley cluster.



(a) scaled 0.2x-1.5x (b) random distributions

Fig. 12: **Microbenchmark performance results on various noise/load distributions on Cutter cluster (upto 224 cores)**

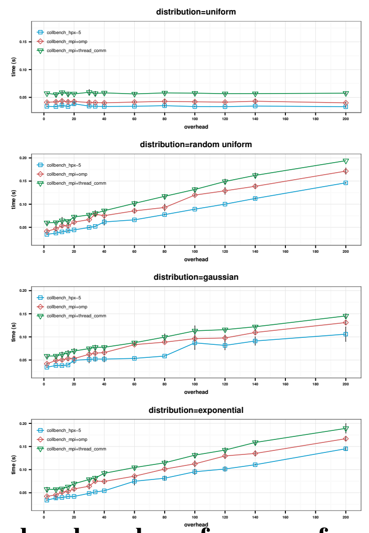


Fig. 13: **Microbenchmark performance for scaling amplitude of load/noise with gaussian,poisson,uniform random, exponential distributions on Cutter cluster (224 cores)**

At small scale we ran up to 16 nodes/256 cores on Intel Xeon E5 16 core 2.1GHz processors supported by an dual Infiniband QLogic/Mellanox Network. Large scale experiments were followed on 'Edison' and 'Cori', upto 1024 nodes/24576 cores on Cray X30 Intel 'Ivy Bridge' 24-core, 2.4 GHz processor

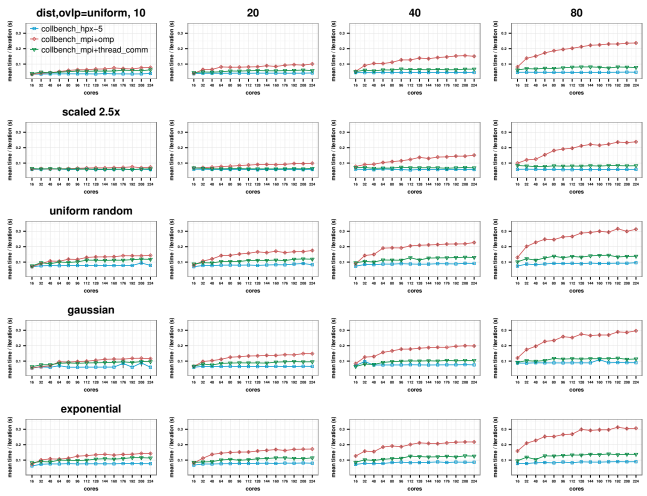


Fig. 14: **Microbenchmark scaling performance with *sequential* overlap work segment size (fixed overhead) under multiple distributions on Cutter cluster (upto 224 cores)**

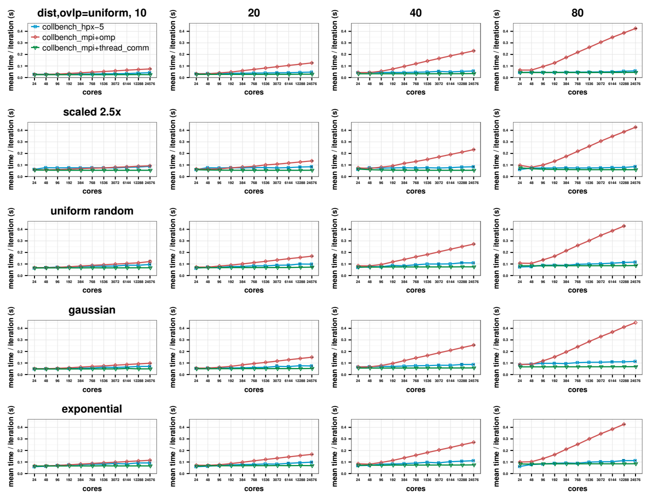


Fig. 15: **Microbenchmark scaling performance with *sequential* overlap work segment size (fixed overhead) under multiple distributions on Edison cluster (upto 24000+ cores)**

nodes and Intel 'Haswell' 32-core, 2.3 GHz processor nodes respectively, backed up by a Cray 'Aries' Dragonfly network with aggregated MPI bandwidth of ∼8GB/s.

A. Case Study - Lulesh

We used MPI+OpenMP version of LULESH, a proxy application for shock hydrodynamics to study the effects of irregular load at scale with different OpenMP loop scheduling options . Furthermore we analyzed the ability to absorb irregular noise by two (2) implementation versions of LULESH namely, a) MPI+OpenMP b) AMT implementation of HPX-5 (also called '*LULESH parcels*' version). HPX-5 '*parcels*' version was adapted directly from the MPI version of the LULESH code. In MPI+OpenMP version of LULESH, MPI handles coarse-grained domain-level parallelism and OpenMP is used for more finer grain parallelism within each MPI

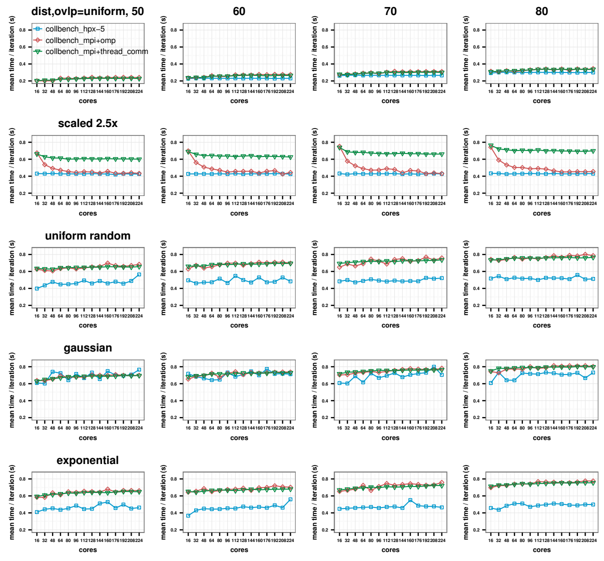


Fig. 16: **Microbenchmark scaling performance with *parallel* overlap work segment size (fixed overhead) under multiple distributions on Cutter cluster (upto 224 cores)**

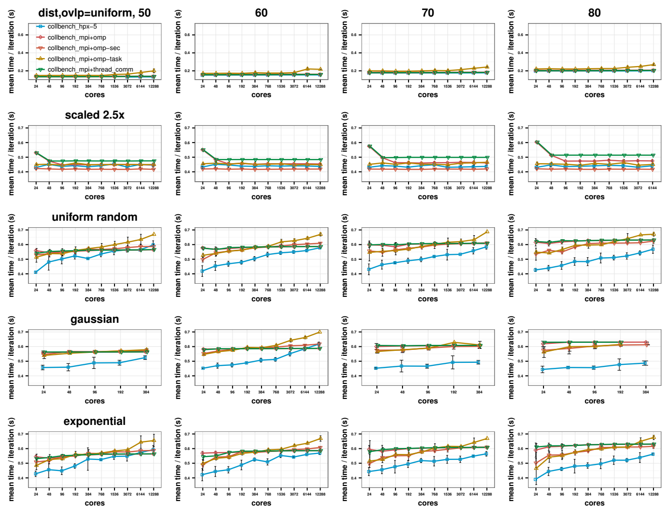


Fig. 17: **Microbenchmark scaling performance on five runtime environments (HPX-5, MPI, MPI+OpenMP, MPI+OpenMP Sections and MPI+OpenMP Tasks), with *parallel* overlap work segment size (fixed overhead) under various noise/load distributions on Edison cluster (upto 12000+ cores)**

process. The MPI+OpenMP LULESH implementation uses MPI across nodes and OpenMP for cores in-node, allowing reductions of the hydro and Courant constraints to be calculated within the threads [10] on a task before messages are sent between tasks. We observed that all the parallel compute regions contributing to *lagrange leap frog* computation was followed by a time constraint calculation which involved a global reduction operation. Thus this usage pattern was consistent with the cases we discussed above (cf. Listing 2)

One observation from the MPI+OpenMP LULESH code was that OpenMP parallel regions were kept separate from

MPI communication, hence it was easier for us to select parallel regions that could be injected with artificial load. There were possible 30 OpenMP parallel regions in Lulesh code but we identified five (6) most significant parallel regions to inject load into by profiling all regions with our framework as shown in Table II. Percentage time spent on these parallel regions were observed from an average $\sim$1 to $\sim$45% in the selected six regions for number of problem sizes and scales we tested on different clusters. Even-though LULESH does not have any inherent load imbalance problem, with load injection we were able to observe behavior that can occur due to irregular noise in parallel regions of applications. More importantly we could characterize its ability to absorb such pressure on different runtime choices.

Figure 9 report a sample execution run of Lulesh on "Cori" HPC cluster spanning upto 2048 cores. We chose a fixed problem size of 48 and executed 100 timecycles to take average running time on Lulesh execution. Four types of experiments were tested on MPI+OpenMP LULESH. First we kept a base experiment which did not inject noise into LULESH, and subsequently tested LULESH performance on varying amounts of load injection with *scaled*, *uniform*, and *uniform random* distributions. Additionally we tested its performance degradation scheduled under 'static', 'dynamic' and 'guided' OpenMP modes. We injected a load of 2us units on each experiment with overhead being kept constant at 200% relative to t_u. We understandably noted a gradual slow down in LULESH performance with different noise distributions, worst being with 'exponential' noise outliers 9. We observed that 'static' OpenMP parallel loop schedules performed the worst with $\sim$265 to $\sim$300% slowdown in 'exponential' noise injection. LULESH OpenMP instances that were powered by 'dynamic' OpenMP loop scheduling, were the best performers with average slowdown ranging as low as from $\sim$1 to $\sim$5%.

Figure 10 and 11 report LULESH relative running time differences (w.r.t. running time of LULESH without any noise injection) on three implementation choices, under scaled and exponential noise outliers in their respective parallel regions. We analyzed LULESH on MPI+OpenMP and HPX-5 ('Parcels') implementations scaling up to 216 cores on "Cutter" and 12000+ cores on "Edison". 'Parcels' version of LULESH was executed on two (2) different modes, *regular* and *oversubscribed*. In all cases *total problem size* was kept at same value [5] for each experiment instance, however in *oversubscribed* mode we increased the number of domains per each HPX-5 processor node. The *oversubscribed* mode enabled HPX-5 work stealing scheduler to efficiently work in overdrive, which was evident from the observed results. In this mode HPX-5 was able to absorb the noise outliers most effectively, sometimes showcasing speedup of $\sim$45% for smaller problem sizes and negligible slowdown of $\sim$1% to $\sim$3% for larger problem sizes. MPI+OpenMP version of LULESH reported the worst mean running time with $\sim$1% to $\sim$10% slowdown on Cutter and $\sim$5% to $\sim$10% on Edison.

B. Microbenchmarks

We developed benchmarks that were purpose built to compare aforementioned properties under different runtime execution models HPX-5, MPI+OpenMP and MPI (cf. Sections V-D, V-B and V-C). In particular all microbenchmark experiments were based upon 2 categories of execution. a) Load characteristics when outliers are present in parallel compute region followed by global synchronization b) Load characteristics when outliers are present in parallel compute region and also extra overlapped work region is embedded. Our initial set of results belong to case a) while the figures followed later will focus on case b). It is important to note that on all microbenchmark experiments that follows, unit work t_u , was kept at constant 40 units. Furthermore all experiments we will show error bars of of 90% confidence interval.

Figure 12a report results of experiments where a single noise outlier were injected into the threads in the parallel region. AMT reported better performance in majority of instances compared to MPI and MPI + OpenMP execution. We observed a maximum mean slowdown of about 66% in MPI while 28% in MPI+OpenMP version w.r.t. AMT execution at 224 cores. We also noticed that HPX-5 can absorb pressure from a maximum single outlier about 1.5 times the unit work.

Several benchmarks tested load injection into the parallel kernel of the MPI+OpenMP and HPX-5 execution following specific random probability density function of which parameters we discussed above (cf. Section V-A1). Each experiment pertaining to a particular random distribution was supplied with an maximum overhead t_{omax} value to 2x times unit work t_u. [6]

Next we adjusted statistical parameters accordingly to fit between the scaled range, for example gaussian mean was set at the mid range $2(t_u)$ and sigma parameter was set $1(t_u)$. Similarly random uniform distribution parameters $[a, b]$ were set between t_u and $3(t_u)$, and so on. We observed that our AMT version was able to complete the reduction faster than MPI+OpenMP counterpart (Figure 12b). We also noticed that the mean latency variation when scaling was significant in MPI+OpenMP, 90% increase from initial value, compared to 19% increase in HPX-5 under exponential noise injection.

Furthermore we tried scaling amplitude of noise outliers, t_{omax} value under different distributions. Results on small scale cluster Cutter can be seen in Figure 13. The maximum overhead t_{omax} parameter was scaled from 2 units to 200 units under various distributions at a constant number (14 nodes/224 cores) of processors (while making sure necessary adjustments to the random probability distributions were made accordingly). Increased overhead understandably caused linear slowdown with different random distributions, yet AMT showed better adaptability with increased amplitude of noise.

A compelling case for AMTs is seen when 2 phase reduction was overlapped with an additional compute region. Most interesting effects from our benchmark can be spotted in Figure

[5]All experiments conducted were weak scaling on LULESH. Total problem size was given by `num of domains . problem size`^{3}

[6]Each parallel load injection t_i was scaled between t_u and $3.t_u$

14, 15 for *sequential* overlap and Figure 16, 17 for *parallel* overlap (cf. Section V-A2). We started with a base case with uniform noise/load injection with a lower overlap segment size setting. On this base case, reduction performance of all runtimes were about the same ($+/-5\%$ to 15% gap in running time), as being shown by the above figures. In both *sequential* and *parallel* overlap case, MPI+OpenMP generally performed poorly with scale, compared to the execution of either MPI or AMT. However in *parallel* overlap case, MPI+OpenMP implementation performed better than its *sequential* setting, especially when uniform and scaled noise outliers were present. We observed slightly different performance characteristics in MPI+OpenMP execution on the two cluster environments. Mainly, various differences in runtime implementations of MPI+OpenMP (**open-mpi** vs **cray-mpich** and **gcc** vs **intel** respectively) have contributed towards this behavior. A rapid performance degradation of MPI+OpenMP execution in some instances were due to the cascading effect of implicit synchronization barrier on MPI+OpenMP local reduction. As discussed earlier, such situations prohibit processors from engaging in other useful work or progressing the network.

Figure 14, 15 report a significant slowdown in regular MPI+OpenMP w.r.t MPI and AMT execution, when the overlap percentage value being gradually increased. For *sequential* overlapped segments on Cutter, MPI+OpenMP reported a maximum of $\sim$2X to $\sim$4X slowdown (on different distributions) while MPI exhibited only a maximum of $\sim$0.65X slowdown compared to the AMT running time. On Edison cluster however scaling upto 24000+ cores, we noticed that MPI threaded execution outperforming our AMT instance marginally. In this case MPI+OpenMP reported a maximum of $\sim$3X to $\sim$6.5X slowdown while MPI reported a $\sim$0.2X to $\sim$0.4X speedup compared to AMT execution. When the same benchmarks were carried out for *parallel* overlapped segments on Cutter (Figure 16), we observed a relative slowdown around $\sim$10% to $\sim$50% w.r.t. AMT execution for gaussian, random and exponential distributions. In the presence of scaled noise outliers however, MPI threaded mode on average performed 45% slower than MPI+OpenMP and 55% slower than AMT. In this case MPI synchronization overhead at smaller scales with native MPI thread modes failed to amortize over relatively small parallalization gain.

Our final benchmark (Figure 17) tested 2 additional modes of execution for MPI + OpenMP, namely asynchronous *sections* and *tasks*, on Edison scaling upto 12000+ cores. AMT execution performed better than other runtime options in most of these instances with *parallel* overlap segments and more importantly we noted that it was able to absorb pressure at these large scales while maintaining a relative running time significantly lower than to that of all MPI+OpenMP modes of 2 phase reduction. We observed a mean slowdown of $\sim$2% to $\sim$35% in MPI+OpenMP *task* execution while MPI+OpenMP *sections* was $\sim$2% to $\sim$25% w.r.t. AMT execution. However MPI+OpenMP *sections* benchmark displayed the best performance in the presence of scaled noise outliers with $\sim$6% speedup against AMT. Both MPI only threaded mode and MPI+OpenMP regular version behaved similarly at scale with relative slowdown ranging from $\sim$2% to $\sim$30% against AMT execution mode.

VII. Conclusion

Naively combining MPI+X in a bulk synchronous parallel way necessarily includes a sequential barrier bottleneck between the X collective and the MPI collective. More involved combinations (e.g., using OpenMP tasks) can eliminate that barrier, but expose the disjoint nature of the MPI and X schedulers. As systems increase in size and real problems become more irregular, these effects will impact the scalability of applications using MPI+X. An AMT runtime with integrated collective support has no sequential bottleneck and has unified scheduling and is therefore not subject to these same scalability limitations.

Our results indicate that given above situations, MPI + OpenMP performance varied rapidly across different execution modes and environments. Effectiveness of other alternatives such as threaded MPI largely depended on the size and structure of irregularity. More importantly, results demonstrate that on both small and large scales a reference AMT implementation was able to outperform MPI + OpenMP 3.0 on a simple collective microbenchmark that simulates load imbalance. Furthermore, as applications are able to expose overlapped work with the non-blocking collective this benefit increased, as the HPX-5 reference runtime effectively used this overlapped work to tolerate system noise and workload irregularities. These results indicate the importance of further development, implementation, and investigation of AMT as an effective approach for current and future large-scale heterogeneous parallel computing.

References

[1] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *Proceedings of SC 2010*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.12

[2] ——, "The impact of network noise at large-scale communication performance," in *IPDPS 2009*, May 2009, pp. 1–8.

[3] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to os interference using kernel-level noise injection," in *Proceedings of SC 2008*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 19:1–19:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413390

[4] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj, "Benchmarking the effects of operating system interference on extreme-scale parallel machines," *Cluster Computing*, vol. 11, no. 1, pp. 3–16, Mar. 2008. [Online]. Available: http://dx.doi.org/10.1007/s10586-007-0047-2

[5] S. Agarwal, R. Garg, and N. K. Vishnoi, "The impact of noise on the scaling of collectives: A theoretical approach," in *International Conference on High-Performance Computing*. Springer, 2005, pp. 280–289.

[6] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[7] T. Mattson, R. Cledat, V. Z. Budimlic, Cave, S. Chatterjee, B. Seshasayee, R. van der Wijngaart, and V. Sarkar., "OCR the open community runtime interface, version 1.0.0," June 2015.

[8] T. Sterling, M. Anderson, P. K. Bohan, M. Brodowicz, A. Kulkarni, and B. Zhang, "Towards exascale co-design in a runtime system," in *EASC 2014*, Stockholm, Sweden, Apr 2014.

[9] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, “Legion: Expressing locality and independence with logical regions,” in *Proceedings of SC 2012*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 66:1–66:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389086

[10] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. Devito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. H. Still, “Exploring traditional and emerging parallel programming models using a proxy application,” in *IPDPS 2013*, May 2013, pp. 919–932.

[11] J. Corbalan, A. Duran, and J. Labarta, “Dynamic load balancing of mpi+openmp applications,” in *ICPP 2004*, Aug 2004, pp. 195–202 vol.1.

[12] F. Cappello and D. Etiemble, “Mpi versus mpi+openmp on the ibm sp for the nas benchmarks,” in *Supercomputing, ACM/IEEE 2000 Conference*, Nov 2000, pp. 12–12.

[13] S. Bova, C. Breshears, R. Eigenmann, H. Gabb, G. Gartner, B. Kuhn, B. Magro, S. Salvini, and V. Vatsa, “Combining message-passing and directives in parallel applications,” *SIAM News*, vol. 32, no. 9, 1999.

[14] E. Lusk and A. Chan, “Early experiments with the openmp/mpi hybrid programming model,” in *Proceedings of IWOMP’08*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 36–47. [Online]. Available: http://dl.acm.org/citation.cfm?id=1789826.1789831

[15] W. Huang and D. Tafti., “A parallel computing framework for dynamic power balancing in adaptive mesh refinement applications.” in *Proceedings of Parallel Computational Fluid Dynamics*, 1999.

[16] W. Huang and D. K. Tafti, “A parallel adaptive mesh refinement algorithm for solving nonlinear dynamical systems,” *International Journal of High Performance Computing Applications*, vol. 18, no. 2, pp. 171–181, 2004.

[17] J. S. Firoz, M. Barnas, M. Zalewski, and A. Lumsdaine, “Comparison of single source shortest path algorithms on two recent asynchronous many-task runtime systems,” in *ICPADS 2015*. IEEE, 2015, pp. 674–681.

[18] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, “Active messages: A mechanism for integrated communication and computation,” in *Proceedings of ISCA 1992*. New York, NY, USA: ACM, 1992, pp. 256–266.

[19] H. Kaiser, M. Brodowicz, and T. Sterling, “Parallex an advanced parallel execution model for scaling-impaired applications,” in *Proceedings of ICPPW 2009*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 394–401.

[20] “HPX-5,” available from http://hpx.crest.iu.edu.

[21] R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999.

[22] E. Kissel and M. Swany, “Photon: Remote memory access middleware for high-performance runtime systems,” in *IPDPSW 2016*, May 2016, pp. 1736–1743.

[23] F. Petrini, D. J. Kerbyson, and S. Pakin, “The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q,” in *Proceedings of SC 2003*. New York, NY, USA: ACM, 2003, pp. 55–. [Online]. Available: http://doi.acm.org/10.1145/1048935.1050204

[24] T. Hoefler and A. Lumsdaine, “Design, Implementation, and Usage of LibNBC,” Open Systems Lab, Indiana University, Tech. Rep., Aug. 2006.

[25] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur, “Enabling mpi interoperability through flexible communication endpoints,” in *Proceedings of EuroMPI 2013*. New York, NY, USA: ACM, 2013, pp. 13–18. [Online]. Available: http://doi.acm.org/10.1145/2488551.2488553