Characterizing Performance of Imbalanced Collectives on Hybrid and Task Centric Runtimes for Two-Phase Reduction

Udayanga Wickramasinghe[1(✉)] and Andrew Lumsdaine[2]

[1] Indiana University, Bloomington, IN, USA
uswickra@indiana.edu

[2] Pacific Northwest National Laboratory, Richland, USA
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 impact of imbalanced reductions on two different execution models: MPI+X and Asynchronous Many Tasking (AMT), with MPI+OpenMP and HPX-5 as concrete instances of these respective models. We explore several approaches to maximizing asynchrony with the HPX-5 and MPI+OpenMP collective programming interfaces and characterize the imbalance using a specialized set of microbenchmarks. Despite maximizing MPI+OpenMP asynchrony, we find situations where scalability of the MPI+X programming model is significantly impaired for two-phase reductions. We report from 0.5X to 6.5X relative performance degradation of MPI+X in the AMT instance.

1 Introduction

The standard HPC platform today is a cluster of multicore nodes, perhaps also including some number of GPU resources. Historically, programmers have used shared-memory approaches for parallel programming of multicore machines and have used distributed-memory approaches (e.g. 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 with shared and distributed memory. The general moniker for the resulting combination is "MPI+X" to reflect the fact that there is a multiplicity of shared-memory approaches but only one MPI.

L. Rauchwerger (Ed.): LCPC 2017, LNCS 11403, pp. 129–144, 2019.
https://doi.org/10.1007/978-3-030-35225-7_10

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 on 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, run-time system, and high-performance programming idioms. It is unlikely to expect independent approaches simply to compose a coherent system (programming or otherwise).

```
compute_kernel() {
  do {
    #pragma omp parallel private(val) reduction(+:sum)
    for (i = 0 ; i < C*nthreads ; i++) {
      val = work(i); //execute shared memory parallel region
      sum += val;
    }
    MPI_Allreduce(sum, global_sum, .., MPI_SUM, ... ); // allreduce SUM
  } while (time_step())
}
```

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

In this paper we study one important use case in parallel programming, namely two-phase reduction, and we investigate the impact of the "+" in MPI+X—in our case we focus on MPI+OpenMP in particular. For example, in MPI+ OpenMP, a global parallel reduction can be performed by first performing a local OpenMP reduction, followed by an MPI reduction across the nodes (1). 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. Any significant variation of thread arrival times at the OpenMP implicit synchronization barrier (that happens just before MPI_Allreduce in Listing 1) may cause cascading delays [1–3] across the overall operation. Additionally, such serialization constraints may reduce the amount of parallelism possible in a severely imbalanced local reduction.

One approach to ameliorating the effect of imbalance on collective operations is to make the collective operation non-blocking. This becomes problematic with a standard MPI+OpenMP 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 – the local

[1] Quote attributed to Bill Gropp.

portion is not overlapped. However, when using more sophisticated approaches to asynchrony, an MPI+OpenMP programmer may work around this restriction.

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. In this paper we present a representative AMT system called "HPX-5" (based on the ParalleX execution model [4]) coupled with a fully asynchronous high-performance collective framework that is well suited for heavily imbalanced global reductions. We also compare and quantify the impact of imbalanced reductions on MPI+X vs AMT, with MPI+OpenMP and HPX-5 as concrete instances of these respective models.

This paper makes the following contributions:

– We analyze the problem of imbalance in two-phase reduction in detail for the MPI+X and AMT programming models. We also propose a generalized formal model that can be utilized to characterize imbalance for such configurations (Sect. 2).
– We implement a high-performance unified collective interface on a representative AMT called "HPX-5" and a portable framework to profile and instrument imbalance with various real-world load distributions into parallel regions of a distributed-memory application on MPI+OpenMP and AMT (Sect. 3).
– We empirically analyze effects of load variation (Sect. 4) on multiple runtime execution models, namely: MPI+OpenMP, MPI, and our AMT instance using a tunable collective microbenchmark.

2 Motivation

Collective communication is known to propagate and even amplify noise effects within an application life cycle. Numerous studies report the effects of external noise [1–3,5,6] 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) for realizing a single compound operation. The effect of computational irregularity becomes isomorphic to that of system noise but potentially orders of magnitude larger.

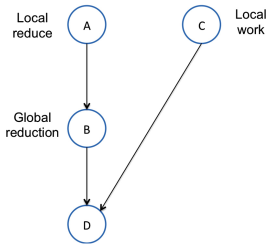

Fig. 1. General dataflow graph found in HPC

Figure 1 reports a simple but commonly found execution pattern of a two-phase reduction operation. One of the limitations of executing such a program in MPI+OpenMP is the strict ordering of the local reduction phase. The data flow graph depicts independent regions **A** and **C** (i.e. no directed edge) and regions **B** and **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 the work of region **C** with **B**. However, the implicit synchronization barrier presents a limiting factor that makes it impossible to hide irregularities

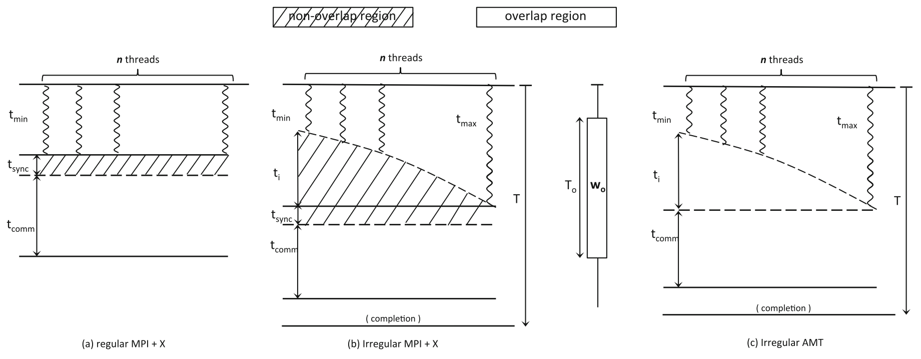

Fig. 2. Generic communication and computation models for two-phase reduction in MPI+X and AMT

in region **A**. Therefore a naive MPI+OpenMP model of programming can make it difficult to utilize available processing resources fully for applications with the aforementioned parallel data dependency characteristics.

Newer implementations of OpenMP (Version 3.0 and on) have attempted to mitigate some of these issues by embedding dynamic loop scheduling, work queues, and task parallelism techniques, e.g. by using new additions to its programming constructs like `pragma omp task`, `pragma omp sections`, and nested regions. An MPI-only approach for two-phase reduction may also use OpenMP as a thread substrate. In such cases special MPI per-threaded communicators[2] should be formed to accommodate local reductions on shared memory buffers. However, effectively controlling parallelism (task scheduling, granularity, etc.), and obscure details of performance tuning across different runtime boundaries (MPI and OpenMP) and compilers (Intel, gcc, etc.) may introduce many *performance* problems and significant discomfort for MPI+X application developers.

AMTs are the newest breed of distributed shared memory runtime systems that have had a significant influence on data-flow-driven parallel programming. We contend that AMTs provide a uniform approach to two-phase collectives even under severe imbalance. For example, due to the asynchronous nature of AMT runtimes, they can effectively overlap communication and computation of regions **A** and **B** (Fig. 1) and combine them with region **C**, thus avoiding wait time for any costly intermediate synchronization steps and thereby increasing throughput. Threads, in terms of early finishers, can compensate for late-comers by taking up any additional work while waiting for network completion.

2.1 Analysis of Two-Phase Reduction

We refer to Fig. 2 for a detailed analysis of the above problem. The leftmost diagram (in Fig. 2) shows a two-phase reduction by n threads conducted in MPI+X runtime under regular conditions (i.e. no outliers). The next two diagrams feature

[2] This can be superseded by MPI-4 Endpoints [16] if the proposal is accepted.

two-phase reduction when outliers are present in MPI+X and AMT, respectively. We state the following definitions and paramters for our evaluation.

Definition 1. ***Overlap Region*** *– An overlap region is a time window at which at least one idle processor is available to process any additional independent compute work (independent w.r.t. this reduction operation).*

Definition 2. ***Non-overlap Region*** *– A non-overlap region is a time window at which no idle processor is available (e.g. due to high contention or model-imposed constraint) to process any additional independent compute work effectively (independent w.r.t. this reduction operation).*

Definition 3. ***Sequential Overlap*** *– A sequential overlap is compute work executed on an overlap region that will run only on a single processor.*[3]

Definition 4. ***Parallel Overlap*** *– A parallel overlap is compute work executed on an overlap region that may run on any number of processors. A parallel overlap is assumed to be embarrassingly parallel.*

T	An upper-bound on *total elapsed time* to complete a single two-phase reduction with an independent work of size W_o, across N number of nodes;
T_o	*sequential latency* of a work of size W_o;
t_{min}	*minimum latency* to complete a local reduction by one or more of total n worker threads;
t_{max}	*maximum latency* to complete a local reduction by one or more of total n worker threads;
t_{sync}	*synchronization overhead* incurred when switching from one runtime boundary to another;
t_{comm}	average *communication latency* for a global reduction operation;
t_i	*barrier latency* defined as time (remaining) to complete the local reduction barrier relative to the fastest thread. This implies that for regular reduction $\forall\ t_i \rightarrow 0$.

Our analysis of two-phase reduction is based on maximizing the overall parallelization (or minimizing latency) possible for MPI+X or AMT in the presence of extra computation work. However, as highlighted before, MPI+X and AMT differ in the execution of the overlap region. MPI+X has a smaller time window to leverage any overlap due to the coarse serialization of local and global reduction phases; it will only be able to overlap work during communication time (t_{comm} in Fig. 2(b)), whereas AMT's overlap region (Fig. 2(c)) is much larger. Therefore, quantification of such potential differentiation is necessary and useful for estimation of this behaviour. We have introduced a notation in-terms of t_{sync} which describe the synchronization overhead when control transfers

[3] We model sequential work as a compute segment with too many data dependencies such that any parallelization of respective code regions is either impossible or impractical.

between different executions, MPI and OpenMP in this particular case. A switch between heterogeneous execution environments may sometimes incur significant runtime overheads due to factors such as fork-join-like model-imposed barriers, implementation-specific constraints[4] as well as system-specific overheads related to coherency issues, instruction and data-level cache, TLB, and page misses. The probability of such occurrence in a homogeneous execution environment such as AMT is low, so we assume $t_{sync} \rightarrow 0$.

Table 1. Maximum overlap region sizes allowed in AMT two-phase reduction for optimal latency hiding on different load distributions

Distribution	Parallel overlap	Sequential overlap
Uniform (regular)	$(n-1) \cdot t_{comm}$	t_{comm}
Scaled	$(n-1) \cdot t_{comm} + n \cdot (t_{max} - t_{min})$	$t_{comm} + n \cdot (t_{max} - t_{min})$
Random uniform	$(n-1) \cdot t_{comm} + \frac{n \cdot (t_{max} - t_{min})}{2}$	$t_{comm} + \frac{n \cdot (t_{max} - t_{min})}{2}$
Gaussian	$(n-1) \cdot t_{comm} + (\frac{n}{\sqrt{2\pi\sigma^2}}) \cdot \int_0^{t_{max}} t \cdot e^{-\frac{(t-\mu)^2}{2 \cdot \sigma^2}} dt$	$t_{comm} + (\frac{n}{\sqrt{2\pi\sigma^2}}) \cdot \int_0^{t_{max}} t \cdot e^{-\frac{(t-\mu)^2}{2 \cdot \sigma^2}} dt$
Exponential	$(n-1) \cdot t_{comm} + n \cdot \int_0^{t_{max}} \frac{t \cdot e^{-\lambda} \lambda^t}{\Gamma(\lambda+1)} dt$	$t_{comm} + n \cdot \int_0^{t_{max}} \frac{t \cdot e^{-\lambda} \lambda^t}{\Gamma(\lambda+1)} dt$

2.2 Evaluating MPI+X

For MPI+X, minimum latency is achievable when additional parallel work W_o, is overlapped with the non-blocking MPI communication. However the overlap region starts only after t_{max}. Therefore the following relation holds true for total elapsed time when MPI+X reduction is executed with parallel overlap:

$$T_{par\_ov} = t_{max} + t_{sync} + \max\ (t_{comm},\ \frac{T_o}{n-1}) \tag{1}$$

For sequential overlap, additional work cannot be executed in parallel (by definition). Therefore, work segment W_o must be delegated to a single thread.[5]

$$T_{seq\_ov} = t_{max} + t_{sync} + \max\ (t_{comm},\ T_o) \tag{2}$$

Understandably, the potential for communication and computation overlap for MPI+X is higher in the case of parallel overlap, since the overlap region can be masked in the communication region of t_{comm}. However, time to complete the overall operation will depend on additional work when the amount of work (or noise) becomes sufficiently large $\frac{T_o}{n-1} > t_{comm}$ and $T_o > t_{comm}$ for parallel and sequential overlap, respectively. Furthermore, if significant synchronization overheads are incurred (t_{sync}), then potential for communication and computation overlap will decrease.

[4] For example MPI would need to execute in MPI_THREAD_MULTIPLE mode with OpenMP which may induce certain penalties compared to regular mode.

[5] Amdhal's Law can be applied for all other cases when both sequential and parallel code regions are present in W_o. However, this evaluation goes beyond the scope of this paper.

2.3 Evaluating AMT

For AMT, minimum latency is achievable when all or part of W_o overlaps with local reduction. Therefore the overlap region starts just after t_{min} (Fig. 2(c)). We formulate the following definition to continue our analysis.

Definition 5. *T_l* ***Local Overlap*** *– Local overlap is the maximum amount of work (in time units) that can be executed by all available idle processors during a local reduction phase (t_i). We model $t_i \in \{\mathbb{R} \mid t_i \geq 0\}$ as a continuous random variable, with a probability density function of $f(t)$. Let n be the total number of threads available, and $E(t)$ be average projected work (i.e. expected value) per thread, then T_l can be calculated by the following.*

$$\begin{aligned} T_l &= n \cdot E(t_i) \\ &= n \cdot \int_0^{t_i} t \cdot f(t) dt \end{aligned} \tag{3}$$

Accordingly, for an imbalanced two-phase reduction in AMT, $T_l > 0$. Thus, for this case the Local Overlap facilitates the communication and computation overlap by reducing the amount of work that need to be parallelized during communication phase the (t_{comm}). By including T_l the following relation can be formulated for the total time of parallel overlap.

$$T_{par\_ov} = t_{max} + max\ (t_{comm},\ \frac{T_o - T_l}{n-1}) \tag{4}$$

Similarly, the following relation holds true for sequential overlap case.

$$T_{seq\_ov} = t_{max} + \max\ (t_{comm},\ T_o - T_l) \tag{5}$$

Equations 4 and 5 highlight the significance of additional work size for the purpose of latency hiding. AMT two-phase reduction reaches optimal overlap[6] when the time required for additional work does not exceed global communication – that is $(n-1) \cdot t_{comm} + T_l$ and $t_{comm} + T_l$ for parallel and sequential overlap, respectively. Thus, the best case for parallel overlap on AMT reduction allows overlap of an additional work region up to a size of $(n-1) \cdot t_{comm} + T_l$, which is much larger than sequential case. Table 1 reports the maximum overlap allowed on AMT for many different probability distributions of load/noise. Barrier latencies (i.e. t_i) follow statistical properties of respective distributions as suggested by the formulae in Table 1, for example: $[t_{min}, t_{max}]$ for *uniform random* , (μ, σ) for *gaussian*, and λ for *exponential.*

2.4 MPI+X Vs AMT

Our model (cf. Eqs. 1 to 5) show that when $t_{comm} > T_o$ and the load configurations are the same AMT's imbalanced two-phase reduction enables execution of

[6] Optimal solution found when $T = t_{max} + t_{comm}$.

a larger overlap region than does MPI+X. This enables AMT to execute a global reduction very efficiently even under severe noise or load variation. Furthermore, even for the case in which heavy computation work ($T_o > t_{comm}$) is overlayed with the reduction, AMT performs better than MPI+X, as $(\frac{T_o - T_l}{n-1}) << \frac{T_o}{n-1}$. Therefore, according to our analysis AMT appears to be the best fit for use cases where distributed irregular reduction will be overlayed with useful parallel compute work. More importantly, such advantage of AMT has become increasingly evident as the amount of parallelism possible per node and the scale increases.

3 A Task-Centric Approach

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

hpx_reduce_action(elem, allreduce){ /* two-phase reduction action */
   hpx_process_collective_allreduce_join( do_work(elem->id), allreduce);
}
run_kernel_action(elem_addr[], N, allreduce) {
   Future sync[N + 1] // create an array of futures to acquire results
   for ( i = 0 to N ) { /* call reduce action on each global element */
      sync[i] = hpx_call(elem_addr[i], hpx_reduce_action, &allreduce); }
   sync[N]=hpx_call(HPX_HERE, overlap, and, elem_addr); // parallel work
   hpx_wait_all(sync);
}
```

Listing 2: **HPX-5 pseudocode shows the parallel execution of two-phase reduction kernel and extra parallelizable work.**

In this paper we have selected the HPX-5 exascale runtime as our representative adaptive multithreaded runtime. Listing 2 reports a code listing for a global reduction of two-phase nature written in HPX-5 pseudo code. The reference implementation of HPX-5 [7] implements a conventional work-stealing scheduler [8] for local lightweight thread scheduling, a high performance Partitioned Global Address Space (PGAS) for active messaging and Remote Direct Memory Access (RDMA) operations, and uses a Photon RDMA library [9] for network transport. Importantly for this work, we introduce a novel non-blocking collective interface in HPX-5, which is assisted by its fully asynchronous lightweight thread runtime. As with MPI+OpenMP, threads interact via

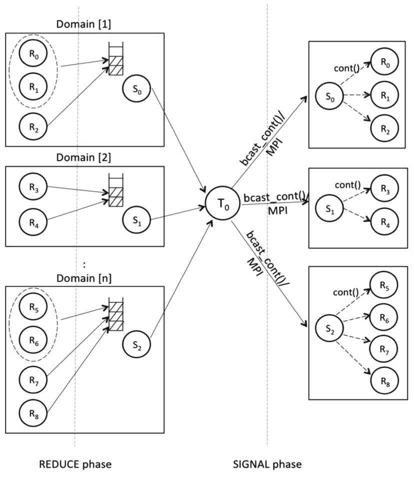

Fig. 3. Continuation driven collective design in HPX-5 for reduction operations

collectives through two phases: first joining the local domains and then communicating globally to arrive at the final value. However, unlike MPI+X, HPX-5 threads do not block during any stage of the collective reduction operation. Once the final reduced result is available, the HPX-5 scheduler will signal all suspended threads for completion.[7] This allows threads to overlap collective communication with computation and to tolerate both latency and irregularity. Our HPX-5 collective implementation has a *continuation-driven* [7] design, as shown in Fig. 3. Additionally, this implementation consists of optimizations such as shared memory, thread-local buffers, and virtual network topologies (binary, binomial trees, etc.) typical for any contemporary high-performance collective interface, implementation details [10] of which are beyond the scope of this paper. HPX-5 implements reduction operations using the HPX-5 collective interface `hpx_process_collective_allreduce_join` [7]. 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 in Sect. 4 will be observed to be well-suited for tolerating noise and irregular behavior expected in exascale systems.

A Framework for Noise Injection. We developed a framework that injects various amounts of load into existing parallel programs to perform our analysis. Our framework uses the method of Fixed Work Quantum (FWQ) [1] to inject and measure load across application regions. FWQ assumes that minimum time t_u, or *unit work* (t_{min} in Fig. 2), represents the perfectly balanced execution of a program region. Our framework can inject $t_{max} - t_i$ delays into an application region. Thus, *unit work* is perturbed by injecting small delays, which we will refer to as "overhead" time or "t_o". Earlier work [2,5] used similar techniques in their noise-injection benchmarks that emulate minuscule amounts of system noise. We identified several important criteria to emulate imbalance. First, we enabled injection of load at varying amounts (amplitude) or conforming to a particular distribution. Second, we enabled injection of load at identified points – locality or light-weight processes/tasks of a distributed memory application. Finally, we implemented runtime-specific extensions for MPI, MPI+OpenMP, and HPX-5. Our model consists of a number of load distribution parameters for instrumentation: unit work (t_u), maximum overhead units (t_u/t_o %) to inject as a percentage of unit work, number of threads to inject in each locality (`tpn`), and a time or work resolution unit. Our emulation system varies load/noise amplitude by adjusting random distribution (Uniform Random, Gaussian, Poisson, etc.) parameters such as mean and standard deviation. A *scaled* version of distribution will scale just one load assignment with an overhead by a specified percentage relative to unit work. The *uniform* injection mode emulates the perfectly load-balanced base scenario.

[7] 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.

4 Results and Discussion

Our experiments have been conducted using a set of customized benchmarks that evaluate two-phase reduction in irregular conditions for three programming/execution models: MPI, MPI+OpenMP and HPX-5. We tested the maximum number of processing elements possible in each cluster in every experiment. We also tested realistic workload distributions with varying frequencies and amplitudes that may naturally occur within load-imbalanced applications. For statistical significance, each measure was repeated 100 times and special care was taken to limit any external interference on performance measures. We conducted all our experiments on two platforms: the small-scale HPC cluster "Cutter" (Intel Xeon E5 2.1 GHz processors, 16 cores per node, up to 256 cores with a **gcc/open-mpi** environment) at Indiana University and the large scale HPC cluster "Edison" (Cray X30 Intel 'Ivy Bridge' 2.4 GHz processors, 24 cores per node, up to 24576 cores with an **intel/cray-mpich** environment) on NERSC at Berkeley.

All microbenchmark experiments are based upon two categories of execution. First, we executed a two-phase allreduce operation[8] when outliers were present in parallel compute regions. Second, we executed the same experiment with an additional parallel work region (Fig. 1). We injected noise outliers for each thread (MPI+OpenMP), process (MPI-only), or task (HPX-5) using our emulation framework (cf. Sect. 3). For overlap we used model parameters, total overlapped region size (W_o), and overlapped work quantum ($\frac{T_o}{n-1} \sim$ work per thread). An overlapped region was emulated either with a sequential or a parallel region. The benchmarks currently implement different variants of these overlap regions for MPI, MPI+OpenMP, and HPX-5 via the `run_overlapped_work(uint64_t qw, uint64_t ow)` interface.

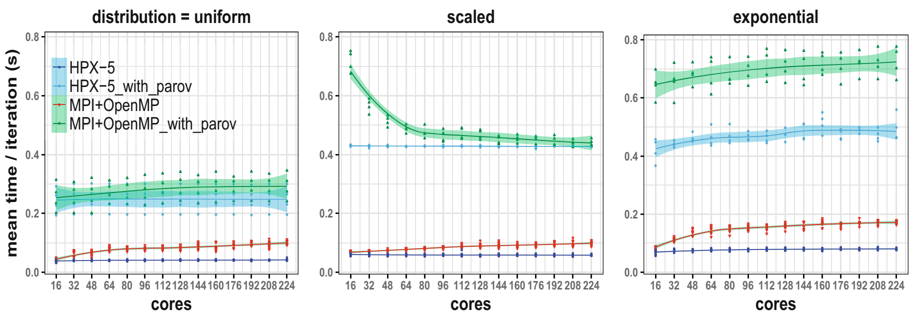

Fig. 4. Microbenchmark scaling with and without *parallel* overlap, for three distributions on Cutter (upto 224 cores)

[8] A collective (i.e. tree-based) algorithm was consistent across all experiments and runtime modes.

Each experiment pertaining to a particular random distribution was injected with a maximum overhead workload of t_{omax} value equating to 2x unit work t_u.[9] Next we adjusted statistical parameters accordingly to fit in the scaled range. For example the Gaussian mean was set at the mid range $2t_u$ and the sigma parameter was set at t_u. Similarly, random uniform distribution parameters $[a, b]$ were set between t_u and $3t_u$, etc. Using experimentation and empirical techniques as tools, we determined a minimum threshold where the cascading effects of noise irregularities became significant. For all microbenchmark experiments that followed, unit work t_u was determined at a constant value of 40 time units. Furthermore, all experiments report regression regions or error bars of a 90% confidence interval.

We first evaluated the performance of two-phase reduction with irregular noise on the Cutter cluster. Two experiments were conducted excluding and including parallel regions (the parallel overlap) on the reduction kernel. On each scatter plot in Fig. 4, we display the fitted lines (evaluated by non parametric LOESS regression) and confidence regions for MPI+OpenMP and HPX-5 for each case. The AMT instance completed the reduction faster than MPI+OpenMP on both these experiments. The uniform case (zero outliers) reported approximately same running times at a single node ($t_{comm} \rightarrow 0$), but, as the frequency of outliers and the scale increased, relative variance in running times became more significant. For exponential outliers HPX-5 reported a ~2.2X speedup when parallel regions were excluded, and when parallel regions were included HPX-5 reported a speedup of ~1.6X w.r.t. MPI+OpenMP.

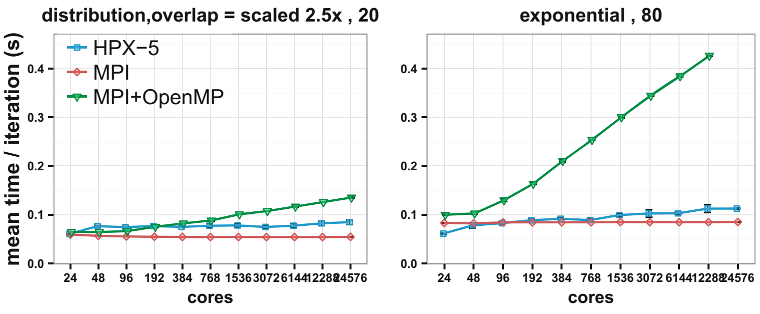

Fig. 5. Microbenchmark scaling results with *sequential* overlap on Edison (upto 24000+ cores)

Even though MPI+X and HPX-5 spent roughly the same time in global reduction (t_{comm}) on Cutter, MPI+X displayed higher synchronization costs ($\uparrow\ t_{sync}$), creating higher latencies than the AMT instance. For parallel overlap AMT showed a greater speedup than MPI+X on account of the higher potential for latency hiding when $T_o > t_{comm}$. We also noticed that the average latency

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

variation with exponential noise was more significant in MPI+OpenMP than in HPX-5; the fitted model for MPI+OpenMP only explained 20% while HPX-5 explained 30% of the variability of data (Fig. 4 *exponential* plot). Interestingly, the MPI+OpenMP allreduce benchmark with a parallel region and a single outlier (Fig. 4 *scaled* plot) displayed resilience by absorbing noise pressure as the number of nodes increased. MPI+OpenMP was able to hide relatively smaller delays (where $T_{comm} \simeq T_o$) by overlapping compute work in parallel regions. Overall results suggested that HPX-5 was better at absorbing noise delays than MPI+OpenMP for two-phase reduction on smaller node counts.

We show the performance of two-phase reduction on the Edison cluster with scaling up to 24000+ cores in Fig. 5. More specifically, we tested a sequential overlap region and started with a base case of scaled noise/load injection with a lower overlap segment size setting (20 time units). In the base case running times of two-phase reduction were about the same (within +/−5%). As expected, MPI+OpenMP two-phase reduction performed poorly with scale as compared to MPI or AMT. For sequential overlapped segments on Edison, MPI+OpenMP reported a maximum slowdown of ∼3X to ∼6.5X (on different distributions), while MPI exhibited a marginal speedup of ∼0.25X w.r.t. HPX-5. Unlike the case where overlap regions are parallelizable, the addition of a sequential region imposed a sequential delay for MPI+OpenMP at the implicit barrier. At this point, when $T_o >> t_{comm}$ the overhead gap generated by MPI+X – ($t_{sync} + T_o$), was much larger than in AMT ($T_o - T_l$). This resulted in amplification of communication overheads with scale and thus a significant slowdown w.r.t. HPX-5 and MPI allreduce.

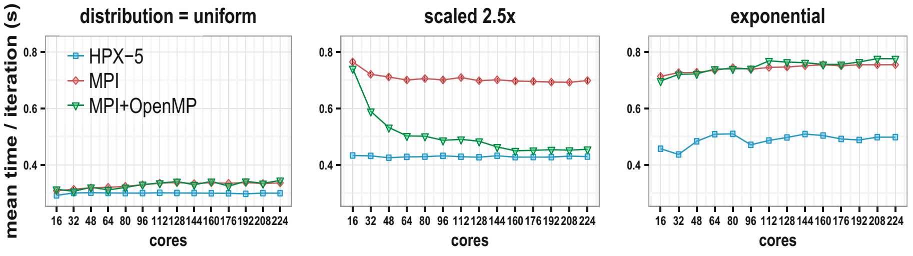

Fig. 6. Microbenchmark scaling with *parallel* overlap work ($t_u = 50$) for multiple distributions on Cutter (upto 224 cores)

For parallel overlap cases, MPI+X performed much better than its sequential configuration (i.e. when uniform and scaled noise outliers were present), an observation which matches the inferences derived by our model (cf. Eqs. 1 and 2). On Cutter (Fig. 6) both the MPI and MPI+OpenMP benchmarks reported a relative slowdown of ∼10% to ∼50% compared to HPX-5 when outliers were present. Here MPI+OpenMP absorbed noise pressure better when a single noise outlier (scaled injection) was present. Interestingly, MPI proved slower in execution times, ∼10% to ∼50% w.r.t. MPI+OpenMP and HPX-5.

Our benchmark on Edison (up to 12000+ cores, Fig. 7), tested two additional modes of execution for MPI + OpenMP: OpenMP `sections` and `tasks`. The AMT instance performed better than the MPI and MPI+X runtime instances in a majority of these configurations. Importantly, we noted that it was able to absorb noise pressure at these large scales while maintaining a running time better than that of all MPI+OpenMP modes for two-phase reduction. More importantly, these observations are consistent with our model as well; AMT has better latency (T) than MPI+X, since $t_{max} + (\frac{T_o - T_l}{n-1}) << t_{max} + t_{sync} + \frac{T_o}{n-1}$. However, Fig. 7 shows that *exponential* and *random* distribution have relatively stable running times for the MPI+OpenMP two-phase reduction. This is because MPI+X does not depend on function $T_l(t_i)$ (Eq. 1), whereas AMT may be affected by cascading delays induced by variation of noise distributions ($T_l(t_i)$) across different nodes (cf. Table 1). Thus we observe that at larger scales the structure of noise between nodes is as important as the mode of execution for an global reduction. In contrast, this behaviour was not visible at smaller scales (Fig. 6), because for a small number of nodes significant amplification of delays is unlikely.

We observed a mean slowdown of ∼2% to ∼35% in MPI+OpenMP `task` execution while slowdown in MPI+OpenMP `sections` was ∼2% to ∼25% w.r.t. AMT. However, the MPI+OpenMP `sections` benchmark displayed its best performance in the presence of scaled noise outliers with ∼6% speedup against our AMT instance. Both the MPI-only threaded mode and MPI+OpenMP regular version behaved similarly at scale with relative slowdowns ranging from ∼2% to ∼30% w.r.t. HPX-5. We also noticed some variance in performance characteristics in MPI+ OpenMP on the two cluster environments. Mainly, differences in runtime implementations of MPI+OpenMP (i.e. **open-mpi** vs **cray-mpich**) and programming environments (i.e. **gcc** vs **intel**) may have contributed towards this behavior. Synchronization costs when $\uparrow\ t_{sync}$, (costs on per-threaded communicators, progress engine, etc.) causes MPI execution higher penalties in certain cases (i.e. that of scaled outliers in Fig. 7) resulting in worse performance than MPI+OpenMP.

5 Related Work

Hoefler et al. have conducted a detailed analysis on the impact of external noise effects on communication synchronization. These effects include operating system [2] and network noise [5]. Other studies [6] shed further light on modeling noise to gain a more analytic perspective on the effect of noise on the scalability of collective operations. Ferreira et al. [2] use noise-injection techniques to assess the impact of noise on several large-scale applications using extremely lightweight kernels. Beckman [1] characterized sources of noise and analyzed performance on BlueGene/L systems, using a synthetic noise-injecting benchmark called “selfish detour”.

Other research reports on MPI+OpenMP usage patterns [11–13] and 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 been deemed successful in practice. The MPI+OpenMP programming

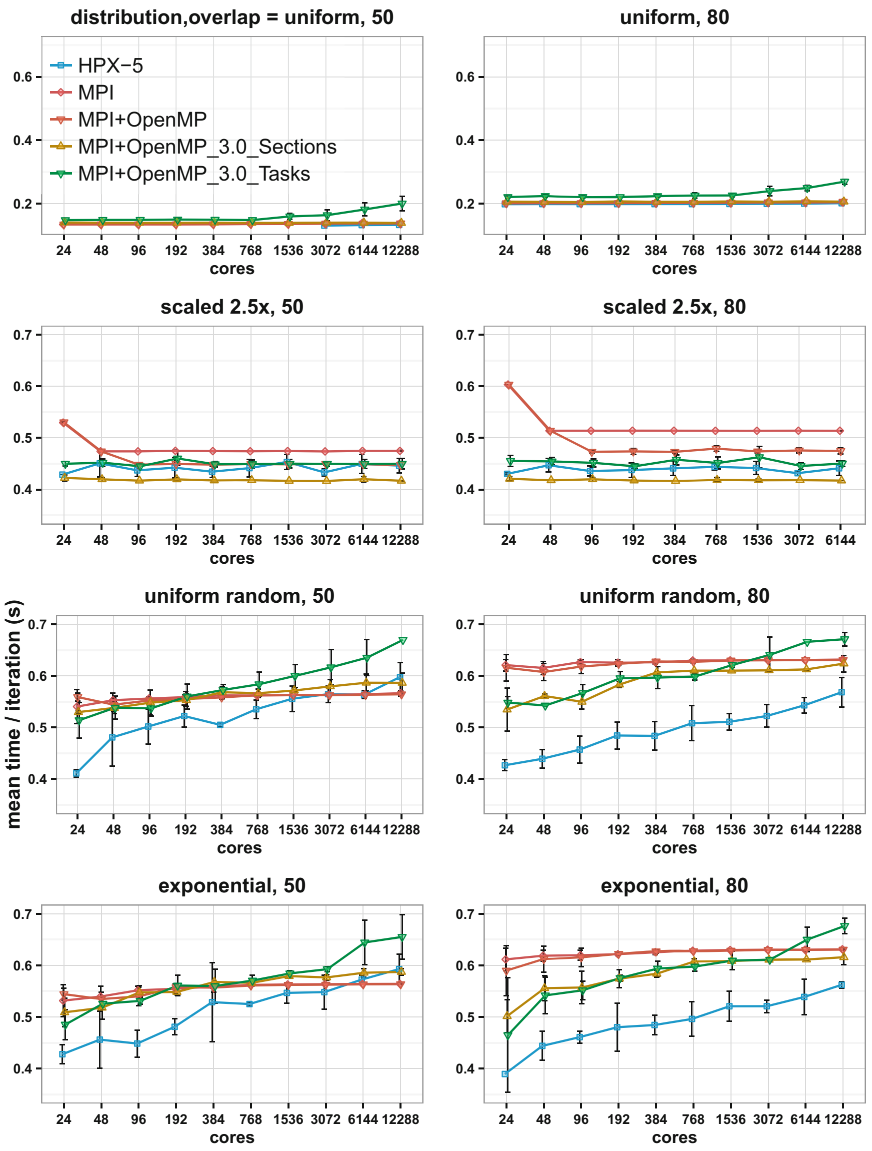

Fig. 7. Microbenchmark scaling with *parallel* overlap work segment size for 4 noise/load distributions on Edison (up to 12000+ cores).

model has been used for irregular application domains. Notably, Tafti et al. [14] have reported its early adoption for AMR. More recently, newer AMT runtimes, such as Legion [15] and OCR [17], too have grown in popularity for tackling large-scale irregular problems. However, the impact of irregularities and imbalance on the performance and scalability of applications has not been thoroughly studied on these systems. Our work differentiates from above efforts in that we focus our attention on the effects of imbalance for two-phase reduction in both the MPI+X and AMT execution models. Furthermore, we formalize imbalance in terms of a probabilistic model and characterize performance under a varying number of configurations.

6 Conclusion

Combining MPI and X for a two-phase reduction in a naive manner introduces a sequential barrier bottleneck between the X collective and the MPI collective. More involved combinations (e.g. by 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 unified scheduling, no sequential bottleneck, and is therefore not subjected to these same scalability limitations.

Our results indicate that given the above situations, MPI + OpenMP performance varied rapidly across different execution modes and environments. MPI+X asynchronous variants, such as OpenMP `tasks` and `sections`, performed better compared to the naive MPI+X implementation. The effectiveness of other alternatives, such as threaded MPI, largely depended on the size and structure of the irregularity. More importantly, on both small and large scales a reference AMT collective implementation was better able to withstand the pressure exerted by simulated noise than any implementation of MPI+X. However, both AMT and asynchronous MPI+X (i.e. OpenMP `tasks`) variants may not be entirely immune to noise at very large scales. We learned that, if the structure and the distribution of the noise changes significantly across nodes, then the tendency to cascade delays may influence the overall scalability of a two-phase reduction. Thus, we recognize that proper characterization of irregularity is essential to understanding the limits of existing parallel systems and address the issues of similar nature. We look forward to building on this framework to design accurate performance models, which will allow us to implement better parallel programming models and paradigms in the face of observed levels of irregularity in HPC systems.

References

1. Beckman, P., Iskra, K., Yoshii, K., Coghlan, S., Nataraj, A.: Benchmarking the effects of operating system interference on extreme-scale parallel machines. Cluster Comput. **11**(1), 3–16 (2008). https://doi.org/10.1007/s10586-007-0047-2
2. Ferreira, K.B., Bridges, P., Brightwell, R.: Characterizing application sensitivity to OS interference using kernel-level noise injection. In: Proceedings of SC 2008, pp. 19:1–19:12. IEEE Press, Piscataway (2008). http://dl.acm.org/citation.cfm?id=1413370.1413390

3. Hoefler, T., Schneider, T., Lumsdaine, A.: The impact of network noise at large-scale communication performance. In: IPDPS 2009, pp. 1–8 (2009). https://doi.org/10.1109/IPDPS.2009.5161095
4. Kaiser, H., Brodowicz, M., Sterling, T.: Parallex an advanced parallel execution model for scaling-impaired applications. In: Proceedings of ICPPW 2009, pp. 394–401. IEEE Computer Society, Washington, DC (2009). https://doi.org/10.1109/ICPPW.2009.14
5. Hoefler, T., Schneider, T., Lumsdaine, A.: Characterizing the influence of system noise on large-scale applications by simulation. In: Proceedings of SC 2010, pp. 1–11. IEEE Computer Society, Washington, DC (2010). https://doi.org/10.1109/SC.2010.12
6. Agarwal, S., Garg, R., Vishnoi, N.K.: The impact of noise on the scaling of collectives: a theoretical approach. In: Bader, D.A., Parashar, M., Sridhar, V., Prasanna, V.K. (eds.) HiPC 2005. LNCS, vol. 3769, pp. 280–289. Springer, Heidelberg (2005). https://doi.org/10.1007/11602569_31
7. CREST: HPX-5. http://hpx.crest.iu.edu
8. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**(5), 720–748 (1999)
9. Kissel, E., Swany, M.: Photon: remote memory access middleware for high-performance runtime systems. In: IPDPSW 2016, pp. 1736–1743 (2016). https://doi.org/10.1109/IPDPSW.2016.120
10. Wickramasinghe, U., DAlessandro, L., Lumsdaine, A., Kissel, E., Swany, M., Newton, R.: Evaluating collectives in networks of multicore/two-level reduction. Technical report, Indiana University, School of Informatics and Computing (2017)
11. Bova, S., et al.: Combining message-passing and directives in parallel applications. SIAM News **32**(9), 10–14 (1999)
12. Cappello, F., Etiemble, D.: MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks. In: Supercomputing, ACM/IEEE 2000 Conference, p. 12 (2000). https://doi.org/10.1109/SC.2000.10001
13. Corbalan, J., Duran, A., Labarta, J.: Dynamic load balancing of MPI+OpenMP applications. In: ICPP 2004, vol. 1, pp. 195–202 (2004). https://doi.org/10.1109/ICPP.2004.1327921
14. Huang, W., Tafti., D.: A parallel computing framework for dynamic power balancing in adaptive mesh refinement applications. In: Proceedings of Parallel Computational Fluid Dynamics, pp. 249–256 (1999)
15. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: expressing locality and independence with logical regions. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 66. IEEE Computer Society Press (2012)
16. Dinan, J., et al.: Enabling communication concurrency through flexible MPI endpoints. Int. J. High Perform. Comput. Appl. **28**(4), 390–405 (2014)
17. Dokulil, J., Sandrieser, M., Benkner, S.: OCR-Vx-an alternative implementation of the open community runtime. In: International Workshop on Runtime Systems for Extreme Scale Programming Models and Architectures, in Conjunction with SC15, Austin, Texas (2015)