Dynamic Adaptable Asynchronous Progress Model for MPI RMA Multiphase Applications

Min Si, Antonio J. Peña, Jeff Hammond, Pavan Balaji, Masamichi Takagi, and Yutaka Ishikawa

Abstract—Casper is a process-based asynchronous progress model for MPI one-sided communication on multi- and many-core architectures. The one-sided communication is not truly one-sided in most MPI implementations: the target process still relies on software progress to complete incoming operations. Casper allows the user to specify an arbitrary number of cores dedicated to background ghost processes and transparently redirects the RMA operations to ghost processes by utilizing the PMPI redirection and MPI-3 shared-memory technologies. Although Casper benefits applications that suffer from lack of asynchronous progress, the operation redirection design might not support complex multiphase applications effectively, which often involve dynamically changing communication density and computing workloads. In this paper, we present an adaptive mechanism in Casper to address the limitation of static asynchronous progress in multiphase applications. We exploit two adaptive strategies, a user-guided strategy and a fully transparent and automatic strategy based on self-profiling and prediction, to dynamically reconfigure the asynchronous progress in Casper according to real-time performance characteristics during multiphase execution. We evaluate the adaptive approaches in both microbenchmarks and a real quantum chemistry application suite, NWChem, on the Cray XC30 supercomputer and an Intel Omni-Path cluster.

Index Terms—MPI, multiphase, one-sided, RMA, adaptation, asynchronous progress

1 Introduction

Advances in high-end computing systems enable scientists to solve complex and large-scale problems with the integration of various fundamental solvers and algorithm modules. Tuning the configuration of runtime systems is a nontrivial task for obtaining highly efficient application performance. This task can be particularly challenging in multiphase applications because of their dynamically changing characteristics of communication and computation during the execution of multiple internal phases, especially when some of the internal phases prefer exactly opposite runtime configurations.

MPI is the dominant parallel programming model on distributed-memory systems. The one-sided communication model (also known as remote memory access, or RMA) allows one process to specify all communication parameters for both the sending and receiving sides. Thus, a process can access a memory region of another process without the target process explicitly needing to receive the message. This asynchronous feature of RMA potentially provides a natural model for some applications that rely on irregular data movement [1], [2], [3], [4]. In practice, however, the RMA communication is not truly asynchronous in most MPI implementations. The reason is that even on RDMA-supported networks such as InfiniBand and the Cray Aries interconnect, most ACCUMULATE operations still have to be handled in MPI software because of the limitations in hardware and MPI semantics. The completion of software-handled operations relies on explicit progress polling on the target process (e.g., making MPI calls). Consequently, an arbitrarily long delay in communication can occur if the target is busy in computation outside MPI.

The traditional approaches to ensure asynchronous progress for MPI communication have relied on two models. One is to utilize the background threads dedicated to each MPI process in order to handle incoming messages from other processes [5]. This model is widely provided in mainstream MPI implementations [6], [7], [8], [9]. However, the fundamental limitation of using this model in real applications is that the thread-based concept requires as many background threads as MPI processes on the system node. Thus, the user has to choose either to lose half of the computing cores or to enable expensive core oversubscription. In addition, this model requires MPI multithreading safety, which is known to be expensive because of the internal thread synchronization [10]. The other model of asynchronous progress is to utilize hardware interrupts to awaken a kernel thread on the target side and process the incoming RMA data within the interrupt context. The interrupt-based model can be found in Cray MPI [24] and in IBM MPI on the Blue Gene/P [12, Chapter 7]. Using this model, however, is limited in that an

- *M. Si and P. Balaji are with the Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL 60439.*
  E-mail: {msi, balaji}@anl.gov.
- *A.J. Peña is with the Barcelona Supercomputing Center (BSC), Barcelona 08034, Spain. E-mail: antonio.pena@bsc.es.*
- *J. Hammond is with Intel Corporation, Santa Clara, CA 95054-1549.*
  E-mail: jeff_hammond@acm.org.
- *M. Takagi and Y. Ishikawa are with the Riken Advanced Institute for Computational Science, Kobe 650-0047, Japan.*
  E-mail: {masamichi.takagi, yutaka.ishikawa}@riken.jp.

Manuscript received 24 Apr. 2017; revised 1 Feb. 2018; accepted 2 Mar. 2018.
Date of publication 13 Mar. 2018; date of current version 8 Aug. 2018.
(Corresponding author: Min Si.)
Recommended for acceptance by D. Arnold.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2018.2815568

interrupt has to be generated on the target side to process each incoming data item, which can be expensive.

In our previous work, we proposed an alternative model for asynchronous progress in MPI one-sided communication, called Casper [13]. Casper enables users to specify a small number of cores as background "ghost processes"; it then transparently intercepts all RMA communication calls to the application processes and redirects them to the hidden ghost processes. Thus, the data movement can be completed by the ghost processes asynchronously while the application processes are tied up by computation outside MPI. Unlike the traditional models, such a process-based model can avoid expensive overheads from either multithreading safety or system interrupts. More important, it allows the user to control the number of cores being utilized for asynchronous progress, which we believe is a more suitable solution for applications running on multi- and many-core architectures.

The basic concept of Casper is to redirect the RMA communication to a few ghost processes. This is suitable for common cases that need only a few cores to trigger data movements while the other resources are used to accelerate computing tasks. However, this design raises a potential bottleneck in that a large number of software-handled operations, which were handled by multiple cores on the node, are aggregated to only one or a few "asynchronous cores" in Casper. Such a small number of progress resources might not be able to complete intensive operations quickly. In particular, when the application does not involve heavy computation and communication becomes dominant, this bottleneck might even counteract the benefit of asynchronous progress and result in poor performance. For single-phase applications, the user of Casper can adjust the number of cores according to the workloads of communication and computation. This method becomes impractical, however, when the application comprises multiple phases, some of which even indicate opposite performance patterns. That is, *the computation-dominant phases can benefit from asynchronous progress with only a small number of asynchronous cores, but some other communication-intensive phases might suffer performance degradation because of the overaggregated operations in Casper*. Thus, there is no way to deliver optimal performance for overall execution.

To address such complications, we present an adaptive mechanism in this paper that allows Casper to dynamically reconfigure the asynchronous progress during the execution of an application's multiple phases. We analyze the performance trade-off with regard to RMA progress, and we design the adaptation to pursue the optimal performance for the overall execution of multiphase applications. We exploit two strategies. One is a user-guided strategy where the user can trigger reconfiguration in every application phase through MPI info hints. The other is a fully transparent strategy that involves automatic self-profiling and performance prediction at application runtime.

We design the framework to ensure strict correctness in accordance with MPI-3 semantics. We evaluate the proposed adaptive approaches through a set of microbenchmarks and a real application suite on a Cray XC30 supercomputer and an Intel Omni-Path Fabric-based cluster. We conclude that the process-based asynchronous progress model is a highly efficient, flexible, and portable approach for MPI RMA.

2 Background

In this section, we briefly introduce the MPI RMA communication model and its implementation limitations on modern network architectures. Here we highlight the important semantics on which this work highly relies. A comprehensive description of RMA semantics can be found in the MPI standard [14] and past papers (e.g., [15]).

MPI RMA Semantics: MPI RMA is introduced in the MPI-2 and MPI-3 standards. To initialize an "RMA conversation," every process in the communicator collectively creates a *window* as the exposure of its local memory region, and a data-transferring phase (called *epoch*) is opened and closed by a set of synchronization calls. During the epoch, a process can access the memory region on a remote process by issuing an RMA operation.

MPI defines an active-target synchronization mode, which includes the *fence* and *post-start-complete-wait (PSCW)* epochs, and a passive-target mode, which includes the *lock* and *lockall* epochs. A fence epoch requires all processes in the window to make the `MPI_WIN_FENCE` synchronization call; a PSCW epoch requires the processes in the origin group and those in the target group to make the `MPI_WIN_START|COMPLETE` and `MPI_WIN_POST|WAIT` calls, respectively; the lock or lockall epoch requires only the origin process to make the `MPI_WIN_LOCK|UNLOCK{ALL}` calls.

The data movement in RMA is defined through the operation calls including PUT, GET, and a set of ACCUMULATE operations (i.e., ACCUMULATE, GET_ACCUMULATE, FETCH_AND_OP, and COMPARE_AND_SWAP, denoted by ACC, GET_ACC, FOP, and CAS, respectively). The ACCUMULATEs guarantee strict *ordering* and *atomicity* for element-wise atomic access to remote memory locations (see page 461 in [14]). A closing synchronization call or an `MPI_WIN_FLUSH{ALL}` call in the passive target mode ensures the completion of operations.

RMA Implementation: The one-sided semantics enables MPI runtime developers to offload the data movement to the hardware of remote direct memory access (RDMA)-supported networks such as InifiBand, Cray Aries, and Fujitsu Tofu. However, the state-of-the-art implementations are usually limited by two factors. First, most RDMA networks are able to process only simple data formats because of the limited processing power on the network interface controller. Complex operations such as computing a multidimensional noncontiguous double array still have to be handled by CPUs in the MPI software. Second, the RMA semantics force the runtime to guarantee ordering and atomicity among ACCUMULATE operations. Thus, none of the ACCUMULATEs can be offloaded as long as a data format is unsupported in hardware. Consequently, the MPI implementations for RDMA networks (e.g., MVAPICH, Cray MPI) usually only offload PUTs and GETs with simple data formats to the hardware and keep the handling of other operations in MPI stack.

3 Casper Overview and Challenge

In this section, we present a brief overview of the Casper framework [13], [16] and discuss the challenge we observed in multiphase applications.

3.1 Overview

Casper is a process-based asynchronous progress model for MPI RMA on multi- and many-core architectures [13], [16]. It allows a few user-defined cores to be kept aside as background "ghost processes," which are dedicated to helping the asynchronous progress for the application processes on the same node.

Casper is designed as an external library through the PMPI name-shifted profiling interface of MPI and transparently provides asynchronous progress for any RMA communication by overloading the necessary MPI functions. This design allows Casper to be platform and MPI implementation independent and enables the user to easily link Casper into the application binary.

When an application process tries to allocate an RMA window, Casper intercepts this call and internally allocates a shared-memory window for all application processes and the ghost processes that are on the same node, using the portable MPI-3 `MPI_WIN_ALLOCATE_SHARED` function. Thus, the ghost processes are able to access the window region located in the memories of the application processes. Then whenever a process tries to issue an RMA operation to the target process, Casper intercepts the call through PMPI and transparently redirects this message to a ghost process on the target node. The ghost processes simply wait in an `MPI_RECV` loop. Therefore, the MPI runtime can quickly make progress for any operations that are handled in the software stack of those ghost processes, and RMA operations that are offloaded to hardware see no difference in the way they behave.

The optimal number of ghost processes is *platform specific and application specific*. Choosing the optimal configuration is essentially a trade-off between the number of resources assigned for computation and that assigned for RMA progress. In practice, using one ghost process per node or one per socket is sufficient for most scientific applications running on CPU cores. This allows the remaining cores to be used to fulfill the heavy computing tasks.

3.2 Challenge in Multiphase Applications

Although using only one or a few asynchronous cores is suitable for most applications, such a small number of progress resources might also lead to a performance bottleneck in some cases. That is, intensive software-handled RMA operations that were completed by a number of application cores on a node are redirected to a few cores in Casper. This processing of overaggregated operations can be slow and might eventually degrade the overall performance of an applications if the following two conditions are met: (1) the portion of computation is less significant than the data movement cost, and (2) the number of dedicated cores is much smaller than that of the remaining computing cores.

In most single-phase applications, the user of Casper can empirically adjust the number of cores for ghost processes in order to avoid the second condition. Such a method becomes impractical, however, when the application involves multiple internal phases and especially when some of the heavy phases perform opposite performance characteristics. For instance, an internal phase may perform extremely expensive computation with little data movement, but the other phase may be dominated by enormous communication. It can be optimal to the former if redirecting communication to only a single asynchronous core in Casper, since the majority of core resources are still used to accelerate the computation; but such a setting can cause a severe overaggregation bottleneck in the latter phase. Unfortunately, a performance trade-off must be made for overall execution.

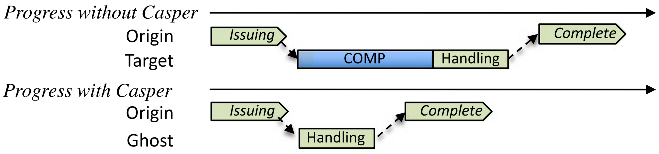


Fig. 1. Decomposing RMA progress.

To provide the optimal overall performance, we need an adaptive mechanism in Casper that dynamically updates the message redirection for different application phases. Specifically, we need to address the following questions.

Q-1. When does an adaptation become necessary?
Q-2. How can we make the adaptation?
Q-3. Where can the adaptation be taken?

4 Decomposing RMA Progress

To answer *Q-1*, we need first to understand the MPI internal overhead for RMA communication. Because the asynchronous progress is needed only when the target process cannot make progress (e.g., computing outside MPI), we consider the simple scenario where the origin process initializes and completes the RMA conversation (e.g., issuing an ACC and waiting in a flush) and the target process does computation. Fig. 1 demonstrates the lifetime of such an RMA progress.

We decompose the completion of an RMA conversation into four portions: the operation issuing taken by the origin process, the network transfer between the origin and the target nodes, the operation handling on the target side, and the local completion on the origin side (e.g., receiving an ACK message in the software handled operation). We abbreviate the cost of each portion as T_{is}, T_{nt}, T_{hd}, and T_{dn}, respectively. Moreover, we consider the worst case that the message arrives on the target just when the target process joined a computation task that takes T_{comp} time; thus the message cannot be handled until the target computation finishes. Consequently, we can formulate the execution time of the original case as $T^{original} = T_{is} + T_{nt} + T_{hd} + T_{comp} + T_{dn}$.

We then formulate the cost when Casper is involved. Unlike the original case, Casper redirects the message to a ghost process. Thus the message can be immediately handled. Therefore, the execution time is $T^{csp} = T_{is} + T_{nt} + T_{hd} + T_{dn}$.

One might expect that T^{csp} should be always smaller than $T^{original}$. In practice, the relationship is actually more complicated. An important factor is that we always have a number of processes and take away only a few of them as ghost processes. Therefore, a ghost process might receive messages instead of multiple target processes, leading to the aggregation of T_{hd}. Let us set the ratio of target processes to a ghost process as r. Thus we finalize the cost with Casper as $T^{csp} = T_{is} + T_{nt} + rT_{hd} + T_{dn}$.

Now we can conclude that *when the aggregated message handling cost in Casper (rT_{hd}) is less significant than the target computation (T_{comp}), the Casper redirection should improve* *performance; in contrast, when rT_{hd} becomes more expensive than T_{comp}, then the overaggregation bottleneck appears.*

Ideally, one would like to measure and compare the overhead of each portion during the execution. However, the MPI standard does not expose a portable interface that allows the user to insert timers and query the information for an arbitrary internal step. Obtaining the cost of the RMA message handling has to rely on the implementation-specific support of MPI tools interface (`MPI_T`).

5 Adaptable Asynchronous Progress

In this section, we focus on *Q-2* and *Q-3* through the design and implementation of the adaptive mechanism in Casper.

5.1 Principle of Adaptation

When the overaggregation risk appears, one way to implement adaptation in Casper is to dynamically assign more resources to asynchronous progress. Thus the aggregated handling cost can be eliminated (i.e., reduced r, the ratio of target processes to a ghost process). The obvious drawback, however, is that we need to dynamically transfer some application processes to the "ghost group," which will result in heavy data repartitioning, or we have to make expensive process oversubscription through MPI dynamic process.[1]

To minimize the overhead of adaptation, we choose a more lightweight approach. That is, we *disable the redirection when the overaggregation issue becomes significant, so that the operation messages can be handled by sufficient processes; when the overaggregation issue disappears and the delay caused by computation becomes dominant, we re-enable the redirection* (*Q-2* is answered).

We notice that this strategy has two potential issues when the redirection becomes disabled. The first is that the messages may suffer from the lack of asynchronous progress again, since they are now handled by the application processes. Given that the disabled setting is needed only when communication becomes dominant, we believe the application processes should be able to make sufficient progress by themselves. The second issue is that the cores dedicated to ghost processes are unutilized. Since the number of dedicated cores is usually small, we consider that this limitation is acceptable.

Following the basic adaptive approach, we then implement the mechanism in two directions. We first study a strategy based on *user guidance*. The assumption is that the user knows the performance characteristics of each internal phase; thus the user can request Casper to enable or disable redirection for each particular phase by passing hints to Casper at the beginning of that phase. The simplified solution allows us to concentrate on the important semantics correctness according to the MPI standard. As the second direction, we design a fully automatic strategy based on the idea of *self-profiling*. In the following sections, we describe the design of each strategy separately.

5.2 User-Guided Adaptation

Casper is required to maintain the consistency of message redirection over all processes in an RMA window. The reason is that any simultaneous operations issued to the same target in that window must always be handled only by a single process, in order to ensure the *ordering* and *atomicity* of ACCUMULATEs. Therefore, we allow the user to reconfigure through an MPI call only when the call guarantees that *all window-wide outstanding operations are completed and all application processes in the window can collectively apply the same change*. Specifically, the reconfiguration can be done either at a window allocation or at a window-wide synchronization call that meets both conditions (*Q-3* is answered). Therefore, we consider three levels of granularity.

Global Configuration: The user can specify a global configuration applied to the entire execution through the environment variable `CSP_ASYNC_CONFIG` with two possible values, `ON` or `OFF`, to enable or disable the redirection in Casper.

Window Configuration: Whenever a process allocates a window, the user can pass the MPI info hint `async_config=ON|OFF`to reconfigure for the communication performed through that window.

Sync-Phase Configuration: Epochs are the natural synchronization phases. However, not all the epochs can perform adaptation. For instance, the synchronization calls in PSCW and the passive target epochs involve only partial processes of the window. Thus, updating in those calls can break the correctness. We can safely reconfigure only in fence. Users can pass the `async_config` info hint for a fence epoch by inserting `MPI_WIN_SET_INFO` before the starting fence call. We require the value of this info to be identical across all processes. Additionally, we propose a new "collective" info hint `symmetric=true|false` that users can pass in `MPI_WIN_SET_INFO`. The `symmetric=true` hint is parsed in Casper, meaning that the user ensures that all outstanding operations in the window have been completed and all processes have arrived at this call. This allows Casper to trigger an adaptation in `MPI_WIN_SET_INFO`; thus it is useful especially within the passive target epochs. For instance, it can be used after a *flush_all-barrier*, which commonly exists in passive target programs.

5.3 Transparent Profiling-Based Adaptation

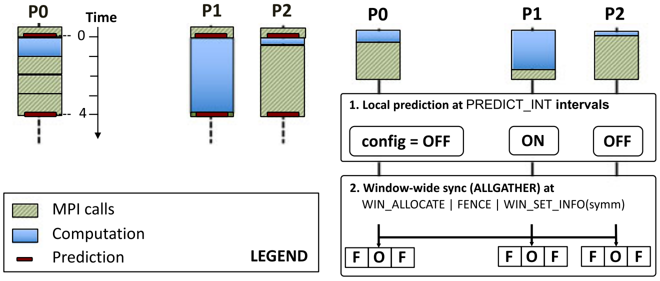
We next introduce techniques to enable the fully automatic adaptation. Instead of user guidance, we want to dynamically predict the impact of redirection in Casper for an application phase. This is based on the notion that the application usually performs a similar communication/computation pattern at a certain period of execution time (e.g., in the same solver). Therefore, we can study the performance of a recent period of execution, assume that the pattern continues for the upcoming period, and use the pattern to trigger adaptation in the Casper runtime.

The key challenge of this approach is that, in order to ensure portability of Casper, we need to obtain the performance information under the constraint that we utilize only the PMPI layer resources. Moreover, we need to address a second challenge, namely, that the dynamically predicted results are applied to all processes in the window consistently. Similar to the user-guided approach, we rely on a

[1] The MPI dynamic process concept allows a program to spawn additional MPI processes during execution. However, the support of dynamic process is limited on HPC systems; for example, the MPICH implementation supports it only on TCP networks.

TABLE 1
Important Factors for RMA Overhead Construction

Factor	Sample Inputs Used in Offline Preprocessing
Data size	Data size in bytes.
Datatype	Contiguous: double; Strided: 3D subarray (double).
Operation type	PUT; GET; ACC; GET_ACC; FOP; CAS.
Blocking pattern	Blocking: flush for every OP; Nonblocking: flush for multiple OPs.
Target pattern (t)	All-to-1-node: Everyone issues OP to the procs on one node; All-to-all: Everyone issues OP to all procs.
Num of procs (n)	Total number of processes.
Num of ghosts (g)	Number of ghost processes on a node.


P0
Time
P1
P2
0
4
▨ MPI calls
🟦 Computation
🟥 Prediction
LEGEND
P0
P1
P2
1. Local prediction at PREDICT_INT intervals
config = OFF
ON
OFF
2. Window-wide sync (ALLGATHER) at
WIN_ALLOCATE | FENCE | WIN_SET_INFO(symm)
F O F
F O F
F O F

(a) CR Calculation. (b) Win-wide Synchronization.

Fig. 2. Self-profiling adaptation.

collective synchronization at the window allocation or synchronization calls for updating the redirection. Although this solution does not show any problem in the user-guided approach, it can result in failure of adaptation in the profiling-based solution if the timing of the synchronization in the application code does not match the change of performance. For instance, the computation can become heavy after a window is allocated, but there may not be any MPI call that allows Casper to perform the synchronization.

In the remainder of this section, we describe our solution as separated into three key components: the self-profiling and local prediction, a basic window-wide synchronization framework, and a special ghost-offloaded synchronization.

5.3.1 Self-Profiling and Local Prediction

Through only the timers inserted in PMPI layer, it is impractical to measure the time of RMA internal portions because they can be processed internally at arbitrary MPI calls. Therefore, instead of focusing on only the message-handling cost rT_{hd}, we try to obtain an approximate relationship between the computation time T_{comp} and the overall communication time T_{comm}. Theoretically, in a specific pattern (e.g., the same operations with the same data size and format), the proportion of T_{hd} related to the other internal portions should be the same on a system with the same MPI environment. Therefore, if $rT_{hd} > T_{comp}$, we should be able to obtain $xT_{comm} > T_{comp}$, where the value of parameter x is approximately identical for a specific pattern.

This notion allows us to build an approximate prediction model. We define a *communication percentage rate* criterion $CR = T_{comm}/(T_{comm} + T_{comp})$ to indicate the proportion of communication time T_{comm} in the overall execution time $(T_{comm} + T_{comp})$. We note that T_{comp} includes any time that is spent outside MPI (e.g., computation, I/O). We employ an offline preprocessing step to determine the reference values of CR that indicate that the communication with asynchronous progress redirection takes the same amount of time as redirection disabled for different communication patterns and for different system deployments. We store the CR reference values for a system and use them as the threshold of real-time adaptation. When the user executes an application, the Casper runtime can perform online profiling and periodically predict the setting based on a corresponding threshold. Below, we describe each step.

Offline Preprocessing: In this step, we design benchmarks to simulate various communication patterns and estimate the CR rate when the condition $(T_{comm} = T^{original} = T^{csp})$ is met. The RMA overhead construction can vary depending on several factors as listed in Table 1. Thus our preprocessing experiments must cover many different sets of those values in order to reduce the deviation. The second column in the table shows the input matrices generated in our benchmark. In Section 7.2 we describe the benchmark details and study the results obtained on our test platforms.

Online Profiling: During the application execution, we periodically measure the real-time CR rate for every period of execution. We insert timers in every MPI function through the PMPI wrapper in Casper to accumulate the time spent in communication T_{comm} and the overall execution time $(T_{comm} + T_{comp})$. As shown in Fig. 2a, we locally calculate the rate of CR on every process for the period during two prediction points by using the accumulated times. For instance, a 75 percent rate is obtained on P0 in the example.

Local Prediction: The next step is to locally predict the new configuration for the upcoming period of a process based on the latest real-time CR rate and the threshold obtained offline. A rate higher than the threshold means that the time the process makes progress in the MPI stack should be sufficiently long to potentially cause operation overaggregation if redirection is enabled (i.e., $rT_{hd} > T_{comp}$). Conversely, a rate lower than the threshold indicates a large proportion of computation (i.e., $rT_{hd} < T_{comp}$) on this process; enabling asynchronous progress in this case becomes more beneficial. We further use a two-level threshold `HIGH_CR` and `LOW_CR` to avoid frequent fluctuations among large varieties of communication patterns and data characteristics. To ensure a sufficient base of profiling time for every prediction, we also define the threshold `PREDICT_INT` in order to control the interval between two predictions.

5.3.2 Window-Wide Synchronization

After the local prediction on every target process, we need to coordinate with the origin side. Thus, the origin process can decide whether to redirect to a ghost process when issuing operation to that target. Similar to the restriction in the user-guided approach, the window-wide synchronization must be done with either a window allocation or special synchronization calls such as `MPI_WIN_FENCE` or `MPI_WIN_SET_INFO` with a symmetric hint.

This component is implemented in a straightforward way such that every process in a window collectively exchanges the last predicted configuration by using `MPI_ALLGATHER` and stores the exchanged data in a local array, as demonstrated in Fig. 2b. Therefore, when the next

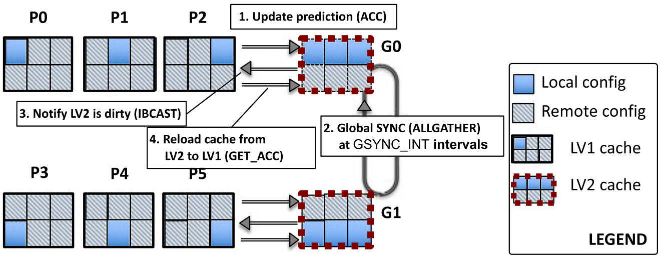

P0
P1
P2
1. Update prediction (ACC)
G0
3. Notify LV2 is dirty (IBCAST)
4. Reload cache from LV2 to LV1 (GET_ACC)
2. Global SYNC (ALLGATHER) at GSYNC_INT intervals
P3
P4
P5
G1
Local config
Remote config
LV1 cache
LV2 cache
LEGEND

Fig. 3. Ghost-offloaded synchronization.

communication starts, any two operations issued to the same target on the window must both be redirected to the ghost process or issued to the original target process.

5.3.3 Ghost-Offloaded Synchronization for PUT/GET

Although the window-wide synchronization ensures semantics correctness, it also limits the adaptation to be valid only at several MPI calls. Unlike ACCUMULATEs, PUT and GET do not require ordering or atomicity. Thus they should be adapted in a more flexible way that does not rely on the existence of special MPI calls in the application code. Therefore, we further investigate a ghost-offloading approach for PUT and GET where the background ghost processes periodically perform an *asynchronous global synchronization* to exchange the predicted results for all processes. To minimize the overhead, we carefully decouple the local synchronization between the application processes and local ghost processes and the global synchronization among ghost processes, by utilizing a *two-level cache* mechanism as shown in Fig. 3. Below, we describe the detailed implementation.

Two-Level Caches: Every application process allocates the level-1 cache (denoted by LV1) on its local memory to ensure lightweight querying at frequent PUT/GET calls; a shared window is then allocated among the application processes and the first ghost process on every node called "sync ghost" as the level-2 cache (denoted by LV2). Each cache is an array that stores the latest configuration of all application processes in the system and is created only once at MPI initialization. The offset of the configuration for a particular process is consistent in all processes' caches. The example shown in Fig. 3 demonstrates the cache construction of six application processes distributed on two nodes. Every cache is a six-integer-elements array, where the elements from offset 0 to 5 are responsible for the cached configuration from process P0 to P5, respectively.

Local Updating: Whenever an application process performs the prediction (see Section 5.3.1) on its local stack, it immediately updates the result to the corresponding element in the LV1 cache. If the value is different from the previous one (e.g., changed from `ON` to `OFF`), the element in the LV2 cache is also updated by issuing an ACC to ensure atomicity when accessing the shared window.

Ghost-Offloaded Global Synchronization: Regardless of the execution on application processes, the sync ghosts perform a global exchange of the LV2 cache at specific intervals defined by the threshold `GSYNC_INT`. Each of the ghost processes sends out the elements corresponding to its local application processes (shown as the solid blue blocks in the LV2 cache in Fig. 3) and receives remote values from others through an `MPI_IALLGATHER` collective call.

TABLE 2
Hardware and Software Configuration on Two Experimental Platforms

	CPU	Memory	Interconnect	HT Enabled
Edison	2×12-core Ivy Bridge	64 GB DDR3	Cray Aries	Yes
Bebop	2×18-core Broadwell	128 GB DDR4	Omni-Path	No
	MPI	**RMA Implementation**	**Default Async**	**Casper**
Edison	Cray MPI 7.2.1 Cray MPI 7.6.0	All SW HW contig PUT/GET	Thread Thread	static/adaptive - *
Bebop	Intel MPI 17.0.4	All SW	Thread	static/adaptive

**Casper cannot execute with the Cray MPI 7.6.0 because of an issue in the MPI implementation that has been reported and is being fixed. Thus we use the Cray MPI 7.2.1 as the primary MPI version on Edison.*

Dirty Notification and Reloading: After a global synchronization, each sync ghost issues a dirty notification to its local processes in an `MPI_IBCAST` call that is periodically tested on each process. Each process then reloads its LV1 data from the LV2 cache by using an GET_ACC.

Per Operation Query: At the issuing of every PUT or GET, the origin process queries the latest configuration of its target through the LV1 cache, in order to decide whether to redirect that operation.

Performance Optimization: The additional synchronization can result in extra overhead in both global synchronization and access to the LV2 caches. Avoiding any unnecessary synchronization is nontrivial. For example, after a window-wide synchronization on most processes in the system such as that with a window allocation call, the first application process on every node can directly update the synchronized data into the node's LV2 cache, and the sync ghosts can skip the upcoming synchronization.

6 Experimental Environment

We performed our experiments on two platforms: the NERSC Edison Cray XC30 supercomputer[2] and the Argonne Bebop cluster.[3] Table 2 summarizes their hardware and software configuration. We highlight two important features: (1) each node of Edison comprises two sockets of the 12-core Intel Xeon E5-2695 v2 processor (Ivy Bridge) with hyper-threading (HT) enabled, whereas the Bebop node uses two sockets of the 18-core Intel Xeon E5-2695 v4 processor (Broadwell) without hyper-threading; and (2) the Cray MPI 7.6.0 on Edison offloads contiguous PUT and GET to hardware and handles other operations in software, whereas the Cray MPI 7.2.1 and the Intel MPI on Bebop handle all operations in software.

For the application case study, we used the large-scale computational chemistry application NWChem version 6.3, with MKL (version 11.2.1 and 2017.3.196 on Edison and Bebop, respectively) as the external math library.

We compare the proposed adaptable Casper with original MPI and several static asynchronous progress approaches as defined in Table 3. The approaches with hardware-offloaded RMA employed the Cray MPI 7.6.0 on Edison; all other

[2] https://www.nersc.gov/users/computational-systems/edison
[3] https://www.lcrc.anl.gov/systems/resources/bebop

TABLE 3
Definition of Evaluation Approaches

OrigMPI	Original MPI with SW RMA and no async support.
OrigMPI/HW	Original MPI with partial HW RMA and no async support.
Static Asynchronous Progress	
CSP(g)	SW RMA with Casper static redirection and g number of ghosts.
TH(D)	SW RMA with thread async, dedicate 50% cores.
TH(O)*	SW RMA with thread async, oversubscribe core.
TH(O)/HW	Partial HW RMA with thread async, oversubscribe core.
Adaptable Asynchronous Progress	
CSP(U)	SW RMA with Casper user-guided adaptation.
CSP(GP)	SW RMA with Casper profiling-based adaptation.
CSP(P)	SW RMA with Casper profiling-based adaptation (only window-wide sync).

Because the Cray MPI 7.2.1 was not available when we measured the thread (O) approach, we used the FALLBACK *mode of 7.6.0, which behaves the same as 7.2.1, handling all operations in software.*

approaches used either the Cray MPI 7.2.1 or the Intel MPI as listed in Table 2. The Cray MPI 7.2.1 supports a *DMAPP mode* that executes contiguous PUT and GET in hardware and provides the interrupt-based asynchronous progress for other operations. It is omitted in the evaluation because of its known overhead due to frequent interrupts; and, in fact, this mode is deprecated in Cray MPI 7.6.0.

7 Microbenchmarks

In this section, we analyze the performance of the adaptive approaches on five microbenchmarks. We use the Cray MPI 7.2.1 as the primary MPI version on Edison.

7.1 Overhead Analysis

We first evaluated the overheads caused by the proposed adaptation in comparison with static Casper on Edison. We ran the experiments on a single node with one ghost process, and we varied the total number of application processes.

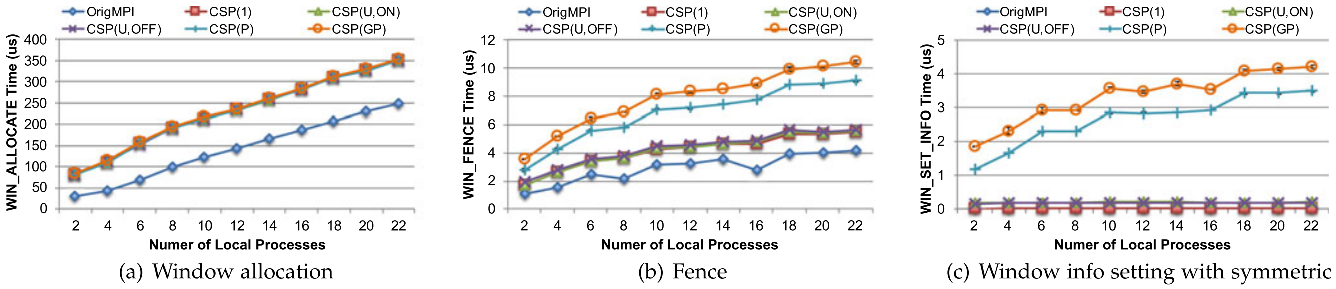
Window Allocation: Fig. 4a shows the overhead of `MPI_WIN_ALLOCATE` on an application process. Since we focus only on the fence or lockall synchronization in our evaluation, we set the `epoch_type=fence,lockall` info at the allocation call for all the Casper approaches. This allows Casper to create only one additional internal window, thus reducing the cost of window allocation (see definition in [13]). We also pass the `async_config` hint with either `ON` or `OFF` in the user-guided approaches (denoted by CSP(U, ON/OFF)). Both adaptive approaches show performance similar to that of static Casper, delivering about 40 to 100 percent overhead compared with the original MPI implementation. We note that this overhead is because of the internal window creation in Casper, which is unrelated to the proposed adaptation. Although the CSP(U,OFF) approach disables the communication redirection, it still suffers from this overhead because we always need to initialize the internal windows in case the user re-enables redirection in the future phase.

Fence: Fig. 4b compares the overhead at `MPI_WIN_FENCE`. The overhead of static Casper compared with the original MPI is due to the passive mode translation in Casper, as discussed in our previous work. The user-guided approaches show performance similar to that of the static version, because they do not involve any additional communication. CSP(P) and CSP(GP) result in extra overhead because of the additional `MPI_ALLGATHER` that exchanges the value of predicted new configurations among all processes. Moreover, we see a consistent gap between CSP(P) and CSP(GP) at close to 1 μs; this is because in CSP(GP) the first application process on the node also updates the synchronized data into the LV2 cache after a window-wide synchronization (see Performance Optimization in Section 5.3.3).

Symmetric Info Setting: When the user passes the `symmetric=true` hint into the `MPI_WIN_SET_INFO` call, we can also perform adaptation. Fig. 4c compares the associated overhead. The profiling-based approaches involve additional `MPI_ALLGATHER` communication, thus showing increasing overhead with increasing numbers of processes. The additional 1 μs overhead of CSP(GP) compared with CSP(P) is the same as in the fence experiment.

7.2 Offline Estimation for Predictive Threshold

We estimated the thresholds of CR rate in the profiling-based adaptation through an offline preprocessing step. We used a benchmark set to demonstrate a common RMA communication pattern where every process performs *RMA-computation-RMA* in multiple iterations following with a barrier. The computation part was simulated as busy waiting, allowing us to flexibly set different computation costs (T_{comp}). The RMA portion was dynamically generated to cover all the combinations of the factor values as listed in Table 1. For each test, the program automatically adjusted the communication cost by increasing the number of operations until the average execution time with asynchronous progress redirection in static Casper was more expensive than the time with redirection disabled. We recorded all



(a) Window allocation (b) Fence (c) Window info setting with symmetric

Fig. 4. Adaptation overhead at window collective synchronization on Edison (average of ten runs, and 1 ghost process is used in all Casper approaches; error is less than 3 percent).

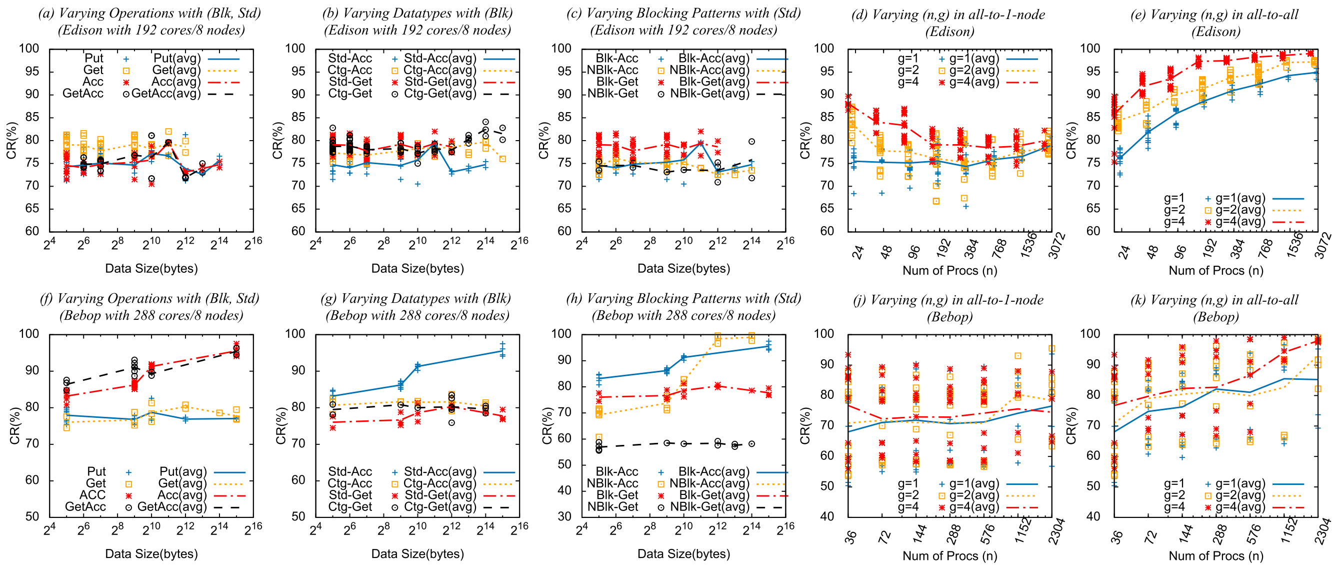


Fig. 5. Analyzing CR rate. (a)–(e)Show the rates measured on Edison, and (f)–(k) show the rates on Bebop. (a)–(c) and (f)–(h) are measured with the all-to-1-node target pattern. List of acronyms: Ctg is contiguous; Std is strided; Blk is blocking; and NBlk is nonblocking.

measured CR rates that indicated an execution time difference in the range of ±5 percent. The benchmark set is available online.[4] We note that performing all the benchmarks is expensive (e.g., consumed close to 260,000 core hours on Bebop). However, we expect such a step is required only once for an MPI environment.

We compared the trend of estimated rates with different sets of factors on both the Edison (with Cray MPI 7.2.1) and the Bebop platforms. We then devised an approach that calculates the thresholds for later experiments.

Varying Operation Types, Datatypes, and Blocking Patterns: We first summarize the trends of estimated CR rates when we change only one of the following factors: operation types, datatypes, and blocking patterns. Figs. 5a, 5b, 5c and 5f, 5g, 5h show the trends on Edison over 192 cores with 1 ghost process per node ($n = 192$, $g = 1$) and the trends on Bebop with ($n = 288$, $g = 1$), respectively. Roughly speaking, the estimated rates on Edison do not show significant differences for different settings. However, the trends on Bebop show significant diversity. For instance, the strided ACCUMULATEs deliver a much higher CR rate than the other types do, and the change of the blocking patterns with GET also results in highly different rates.

Obviously, the diversity of CR rates can be highly platform dependent. To make effective predictions for applications that often involve a mixture of multiple different patterns, we processed the obtained benchmark results following a simple statistical approach. We calculated the mean value of the results for each combination (e.g., a nonblocking strided ACC) called a *basic-mean*. Then we varied the process deployment setting (n, g) with different target patterns and analyzed the distribution of these basic-means.

Varying Process Deployment: Figs. 5d and 5j show the *all-to-1-node*pattern with increasing numbers of n and varying numbers of g on the test platforms. We notice that the basic-means on Bebop are clearly distributed into two ranges. This is the large diversity we observed in the previous comparison. Figs. 5e and 5k show the same measurement but with the *all-to-all* target pattern. We make two observations from the figures. The first is that the greater the number of ghost processes, the higher the CR rate that is required in order to reach a performance bottleneck. This is because the ratio of target processes to a ghost process (abstracted as r in Section 4) is reduced, and thus the bottleneck becomes harder to reach. The second observation is that the larger the number of target processes, the higher the estimated rate. This is because of the reduced proportion of T_{hd} in the overall communication time.

We defined two strategies that calculate the thresholds. In the first strategy, we calculate the overall boundaries of the basic-means for every set of (n, t, g) on the platform. When the deployment of (n, t, g) is specified, we directly use the corresponding boundaries. Since t might change in applications, we also defined a second strategy: taking the average of all (n, t, g) boundaries for every set of (n, g).

7.3 Single-Phase Benchmark

Our third set of experiments focused on the usage of static and adaptive asynchronous progress approaches in two single-phase microbenchmarks. Specifically, the first one performs a typical computation-intensive pattern (denoted by COMP), and the second performs a communication-intensive pattern (denoted by COMM). We defined a *phase* where every process performs *RMA-computation-RMA* in 100 iterations following with a barrier. The phase was executed two times on every process in an *all-to-all* fashion. In the COMP benchmark, we computed DGEMM in every iteration with a total problem size $M = N = K = 192000$, and we issued a single *GET-flush* and *ACC-flush* in the first and second RMA steps, respectively. In the COMM benchmark, we reduced the total size of DGEMM to $M = N = K = 9600$ and increased the number of operations to 100 at the RMA steps. Every RMA operation carries data with a 2^3 3D subarray on the 8^3 window region as the target datatype and 8 contiguous double elements as the origin data structure.

We measured each experiment on 192 cores (8 nodes) on Edison. Because the strided operations are handled in

[4] https://github.com/pmodels/casper-dev/tree/dev-dynamic-schd/preprocess

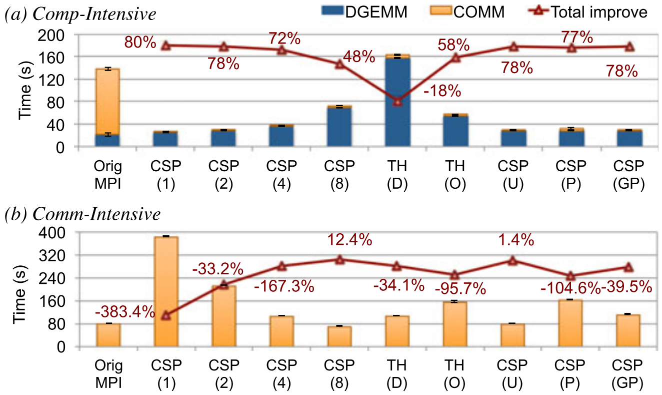


Fig. 6. Performance of asynchronous progress approaches in single-phase benchmarks over 192 cores on Edison (average of three runs, and 2 ghost processes are used in CSP(U), CSP(P), and CSP(GP)).

software in all the MPI versions, we omitted the measurements with OrigMPI/HW and TH(O)/HW. The OrigMPI approach uses 24 processes on every node, and we varied the number of ghost processes from 1 to 8 in static Casper. Each serves 23, 22, 20, and 16 application processes per node, denoted by CSP(1), CSP(2), CSP(4), and CSP(8), respectively. We specified two ghost processes in each adaptive approach. To enable adaptation also in the passive-target communication, we added the win_set_info with symmetric hint after the barrier call (see Section 5.2). In CSP (U), we specified the global `CSP_ASYNC_CONFIG=ON` in the COMP benchmark and set to `OFF` in the other one. In CSP (P) and CSP(GP), we set the CR thresholds to $\{89\%, 95\%\}$ according to the offline estimation for $(n, t, g) = (192, \textit{all-to-all}, 2)$, and we empirically set `PREDICT_INT` to 1 second and `GSYNC_INT` to 2 seconds.

Fig. 6 shows the performance results. Static Casper always reduces the communication cost in the COMP benchmark (Fig. 6a) because of asynchronous progress, but it also degrades the computation performance when using more ghost processes because of losing computing power. TH(D) delivers even worse performance than that of OrigMPI because it occupies 50 percent computing cores. On the other hand, a small number of ghost processes can result in severe degradation in the COMM benchmark (Fig. 6b) because of operation overaggregation. Such overhead can be reduced by using more ghost processes, and the issue can be completely resolved by disabling the redirection, shown as CSP(U). However, the profiling-based adaptations, shown as CSP(P) and CSP(GP), deliver significant overhead in the COMM benchmark. The reason is that they can adapt only after the first barrier, although CSP(GP) can partially help GETs at an earlier time. In both benchmarks, TH(O) does not show better results because of core oversubscription.

7.4 Multiphase Benchmark

Although the user can adjust the setting of static Casper for the execution with a different pattern, achieving optimal performance is impossible if a single execution contains both patterns. Our fourth set of experiments used such a multiphase benchmark. The benchmark contains two sequential windows, each consisting of both a heavy-computing period and a heavy-communicating period (combination of the two single benchmarks in the preceding experiments). Thus, every execution contains eight phases.

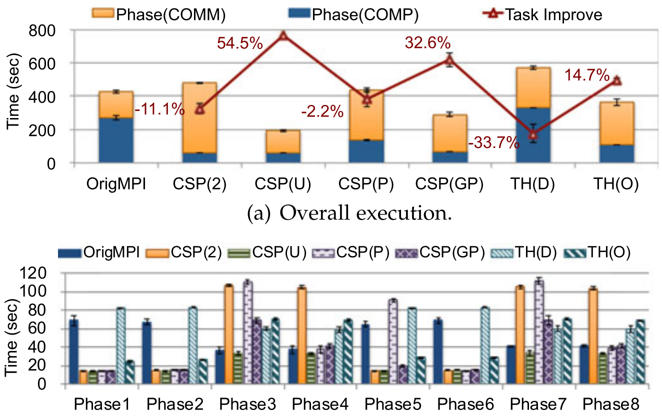
TABLE 4
Expected Adaptation of Casper Configurations (U), (P), and (GP) in Multiphase Benchmark Involving up to 8 Phases

	1	2	3	4	5	6	7	8
Approach	COMP	COMP	COMM	COMM	COMP	COMP	COMM	COMM
(U)	ON	ON	OFF	OFF	ON	ON	OFF	OFF
(P)	ON	ON	ON	OFF	OFF	ON	ON	OFF
(GP)	ON	ON	ON/OFF	OFF	OFF/ON	ON	ON/OFF	OFF

We used two ghost processes in all Casper approaches. In CSP(U), we set the user hint *async_config=ON* at each window allocation call for the upcoming COMP phases, and we set *async_config=OFF* through win_set_info in front of the third and the seventh phases for the next COMM phases. The configuration of CSP(P) and CSP(GP) remained the same as that in the preceding set of experiments.

As listed in Table 4, the three adaptive approaches can result in different reconfigurations in every internal phase. To be specific, CSP(U) can deliver the most precise adaptation that enables redirection (ON) in every COMP phase and disables it (OFF) in every COMM phase. CSP(P), however, cannot promptly adapt to the third, fifth, and seventh phases because it cannot apply the new predicted results until it reaches win_set_info. CSP(GP) partially addresses this issue; for example, the ghost-offloaded synchronization disables redirection at the third and seventh phases only for GETs.

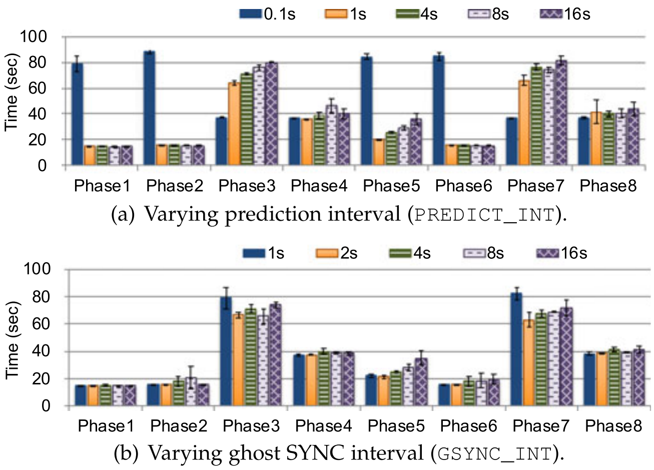
Fig. 7 shows the results. Although the static CSP(2) can reduce the cost of the COMP phases, it also degrades the other phases that perform intensive communication, resulting in an 11.1 percent degradation compared with that of OrigMPI. As expected, CSP(U) achieves the greatest improvement at 54.5 percent; CSP(P) cannot provide appropriate adaptation at Phases 3, 5, and 7, as shown in Fig. 7b, resulting in a 2.2 percent degradation; and CSP(GP) reduces the overhead at those three phases by adapting GETs. The thread approaches suffer from additional cost in either the COMP phases or the COMM phases, similar to our observation in Fig. 6.



(a) Overall execution.

(b) Phases breakdown.

Fig. 7. Comparison of asynchronous progress approaches in the multiphase benchmark over 192 cores on Edison (average of three runs, and 2 ghost processes are used in all Casper approaches).



(a) Varying prediction interval (PREDICT_INT).

(b) Varying ghost SYNC interval (GSYNC_INT).

Fig. 8. Comparison of varying adapting intervals in the multiphase benchmark over 192 cores on Edison (average of three runs).

7.5 Varying Adaptation Intervals

For our fifth set of experiments, we used the same multiphase benchmark to observe the impact of two interval thresholds in the profiling-based approaches: PREDICT_INT (see Section 5.3.1) and GSYNC_INT (see Section 5.3.3).

Fig. 8a compares the per phase costs with varying PREDICT_INT in CSP(GP) with a fixed GSYNC_INT at 2 seconds. We omit the graph of CSP(P) because of space limitations. We notice that the smallest interval, 0.1 seconds, can result in imprecise adaptation especially in the COMP phases (i.e., Phases 1, 2, 5, and 6 in CSP(GP)). The reason is that the fragment of the executed period is so short that it includes only the last few MPI calls; thus the profiling data cannot represent the heavy computing characteristics. With increasing internal time, CSP(GP) shows increasing overhead in Phases 3, 5, and 7, where processes reconfigure for heavy communication, because of the delay in prediction.

Fig. 8b compares the per phase overhead of CSP(GP) with varying GSYNC_INT and a fixed PREDICT_INT at 1 second. It indicates a visible overhead in the COMM phases that are contiguous to the preceding COMP phases (i.e., Phases 3 and 7) when a 1-second interval is set, because of the frequent reloading executed on every application process. Increasing the interval can lead to delays in adaptation, especially in the heavy computing phase (i.e., Phase 5).

8 Case Study: Chemistry Application

In previous work we evaluated the NWChem application with static asynchronous progress by focusing on particular internal phases of the CCSD(T) method [13], [16]. Here we focus on the overall multiphase execution.

NWChem Background: NWChem [1] is a widely used computational chemistry application suite [17], [18]. NWChem is developed on top of the Global Arrays [3] toolkit over MPI RMA [19]. A typical *get-compute-update* mode is widely used in all the internal phases of NWChem, which every process essentially performs by varying the size of matrix-matrix multiplication for multidimensional tensor contraction by coordinating with others through RMA GET/ACC operations. Furthermore, *NXTASK* is the generic task-scheduling component that assigns the "owner" for subdomain computing tasks. It is implemented as a single FOP operation.

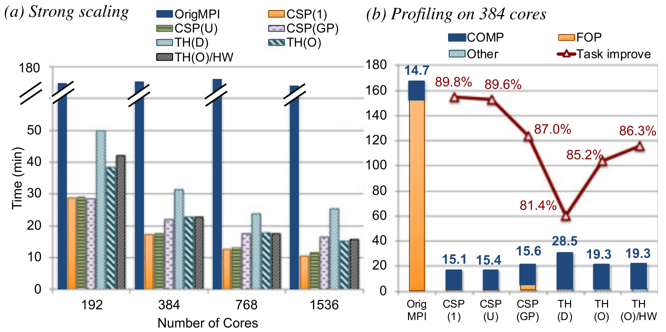


Fig. 9. Single-phase DFT task for C240 with asynchronous progress on Edison. Since OrigMPI/HW cannot complete in 5 hours, it is omitted. All Casper approaches use 1 ghost process per node. COMP values are shown in (b).

We note that most of the RMA operations in NWChem exchange the subblocks of the global matrix. The subblock data is represented as a strided subarray in MPI. Thus, the hardware-offloaded PUT/GET cannot help performance.

Experimental Setup: We inserted a win_set_info with symmetric info at the Global Aarrays GA_SYNC call,[5] since its semantics ensure the completion of all outstanding operations on all processes. CSP(GP) requires three kinds of predefined thresholds: LOW|HIGH_CR, PREDICT_INT, and GSYNC_INT. We used the estimated *CR* thresholds from the results in Section 7.2 following the second strategy; we decided the value of interval thresholds according to the task execution time within OrigMPI. CSP(P) is omitted since it is only for analysis.

We chose two widely used modules of NWChem in our case study: the single-phase density functional theory (DFT) and the multiphase CCSD(T).

8.1 Single-Phase DFT

Density functional theory is one of the most broadly used methods in NWChem. It provides a good mix of efficiency and accuracy to investigate the structural and electronic properties of atoms and molecules. It contains only a single internal phase in the implementation, which follows the *get-compute-update* mode and utilizes *NXTASK* task scheduling.

We evaluated the DFT calculation for Carbon 240 (denoted by C240) with the 6-31G* basis set. We used one ghost process in all Casper approaches; we set CSP_ASYNC_CONFIG=ON in CSP(U); and in CSP(GP) we set PREDICT_INT to 2 seconds, GSYNC_INT to 120 seconds, and a *CR* rate range {75%, 90%}, {75%, 90%}, {80%, 90%}, and {85%, 90%} for 192, 384, 768, and 1,536 cores, respectively.

Fig. 9a compares the strong scaling of both static and adaptive asynchronous progress approaches over a varying number of system cores. OrigMPI does not scale because of the significant delay in the blocking FOP operations in *NXTASK*, as shown in Fig. 9b. All the asynchronous progress approaches can eliminate such overhead; however, the thread-based approaches show increased overhead in computation compared with the Casper approaches as shown in Fig. 9b. This is because TH(D) occupies 50 percent computing cores and TH (O) oversubscribes cores. We compare the static and adaptive approaches in Casper. CSP(1) is clearly the best solution for the single-phase DFT. CSP(U) gives similar performance, but

[5] GA_SYNC internally calls MPI flush_all on all processes followed by a barrier.

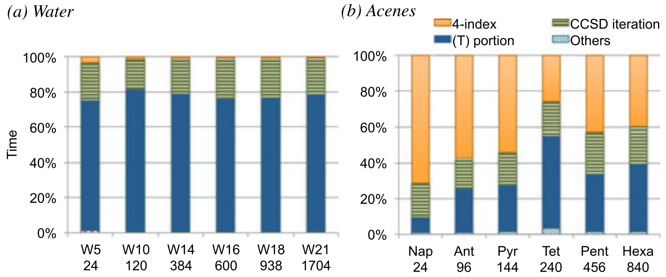


Fig. 10. Internal phases in multiphase CCSD(T) with pVDZ on Edison.

CSP(GP) shows visible communication overhead primarily because of the extra synchronization and prediction error.

8.2 Multiphase CCSD(T)

The coupled cluster theory is one of the most popular approaches in quantum chemistry for solving electron correlation in atoms and molecules with arbitrary accuracy requirements. The "gold standard" *coupled cluster with singles and doubles and perturbative triples* method, known as CCSD(T), is one of the most accurate coupled cluster methods applicable to large molecules to date.

CCSD(T) comprises four internal phases: self-consistent field (SCF), four-index transformation (4-index), CCSD iteration, and the noniterative (T) portion [20]. The overhead proportion among these phases can vary in particular molecular problems. Fig. 10 compares the overhead breakdown of two sets of problems with OrigMPI: the water molecule $(H_2O)_n$ problems (n = 5; 10; 14; 16; 18; 21, denoted as Wn), with the cc-pVDZ basis set, and the Acenes problems, including naphthalene, anthracene, tetracene, pentacene, and hexacene (denoted Nap, Ant, Tet, Pent, and Hexa, respectively) with the aug-cc-pVDZ basis set. Each problem is measured over the appropriate number of cores fitting its memory requirement, as listed below the *x*-axis.

In all the water problems, the (T) portion consistently dominates the cost of the entire task by close to 80 percent, and the CCSD iteration takes the other 20 percent; the remaining phases represent less than 2 percent of the time. The Acenes series shows a different trend in each problem, where the (T) portion indicates only a 52 percent cost in Tet and an even lower proportion in others. Instead, the 4-index contributes more overhead, representing 26–71 percent of the time. We note that the SCF always takes less than 1 percent of the cost; thus it is merged into the "Others" portion for simplicity.

8.2.1 *Analysis with Static Asynchronous Progress*

Next we looked into the performance issue of static asynchronous progress. We chose two problem types: large *W21 molecule over 1,704 cores* and *Tet over 240 cores*. We compared the performance impact on each internal phase by utilizing the static Casper and thread-based approaches. We used the same total number of cores on every computing node in all approaches, some of which are dedicated to asynchronous ghost processes/threads.

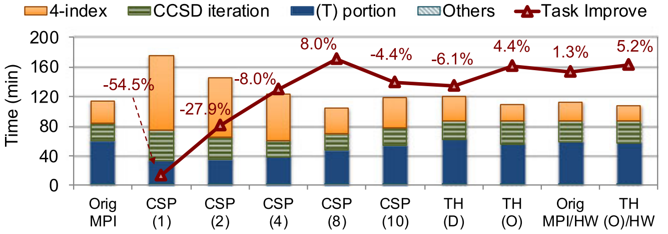


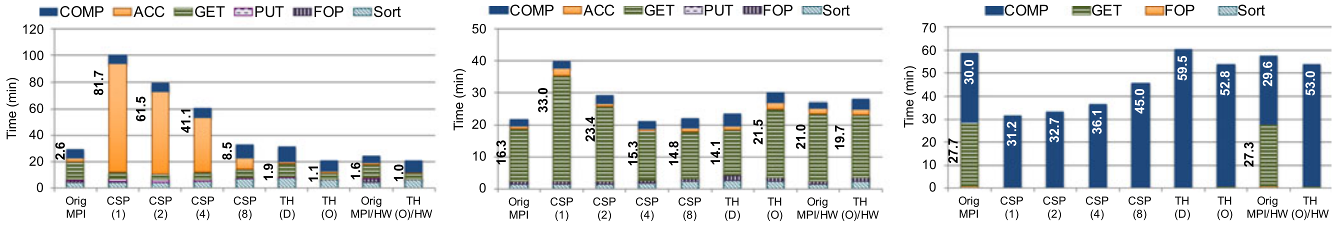
Fig. 11. Trade-off in NWChem CCSD(T) for Tet-aug-cc-pVDZ with static asynchronous progress over 240 cores on Edison. CSP(g) denotes Casper with g ghost processes (g=1; 2; 4; 8; 10). The "Task Improve" rate is calculated based on OrigMPI.

Trade-Off in Overall Execution: Fig. 11 shows the task execution time of the Tet problem. CSP(1) delivers the maximum improvement in the (T) portion by close to 50 percent, but it also leads to more expensive CCSD iteration and 4-index. With increasing numbers of ghost processes such overhead is decreased, but the overhead of the (T) portion increases. TH (D) follows the same trend, because it occupies half of the computing cores. The TH(O) approaches do not perform better because of core oversubscription. As a result, only an 8 percent improvement is achievable with 8 ghost processes.

The internal phases of W21 indicate trends similar to those observed in Tet. Static Casper delivers the best improvement for the overall execution at 28 percent by using 2 ghost processes, because the deduction of the degradations in other phases can be reimbursed by the improvement in the (T) portion, which dominates the entire cost by 80 percent.

Having studied the overall performance trend, we then analyzed each specific internal phase. Since we observed similar trends in each phase in both the W21 and Tet problems, we have omitted the results of W21.

Four-Index Phase: Fig. 12a shows the overhead of the computation and RMA operations in the 4-index phase. The degradation with small numbers of ghost processes is caused mostly by ACCs, which degrade performance by 40x with one ghost process; but the GET portion, which dominates the cost of the 4-index in OrigMPI, can benefit from the redirection in Casper. After careful code reading and profiling, we confirmed that this difference is due to the different target patterns executed in these operations. To be specific, all ACCs are issued as the *all-to-1-node* pattern described in Section 7.2. GETs, on the other hand, are issued following the *all-to-all* pattern.



(a) 4-index phase (showing ACC values) (b) CCSD iteration phase (showing GET values) (c) (T) portion (showing COMP/GET values)

Fig. 12. Profiling multiphase CCSD(T) for Tet-aug-cc-pVDZ with static asynchronous progress over 240 cores on Edison.

TABLE 5
Environment Variables for CCSD(T) with Profiling-Based Adaptation

Edison: {CR(%)}, PREDICT_INT(s), GSYNC_INT(s)				Bebop	
Nap/24:	{81,88}, 60, 2	Tet/240:	{75,90}, 240, 2	Nap/36:	{54,88}, 60, 2
Ant/96:	{78,85}, 120, 2	Tet/384:	{75,90}, 240, 2	Ant/72:	{60,90}, 120, 2
Pyr/144:	{75,90}, 120, 2	Tet/768:	{80,90}, 240, 2	Pyr/108:	{60,90}, 120, 2
Pent/456:	{80,90}, 240, 2	Tet/1536:	{85,90}, 60, 2	Tet/288:	{60,90}, 240, 2
Hexa/840:	{85,90}, 240, 2	Tet/3072:	{85,90}, 60, 2	Pent/576:	{60,90}, 240, 2
	-		-	Hexa/1008:	{65,95}, 240, 2

CCSD Iteration Phase: Fig. 12b shows the profiling of the CCSD iteration phase. Different from the overhead construction in the 4-index, the numerous *all-to-all* GETs dominate the execution time by close to 80 percent, and the DGEMM computation (shown as COMP) takes less than 10 percent. Such intensive communication can rarely benefit from asynchronous progress if the operations are aggregated to only a few ghost processes. Thus, four ghost processes are required in order to balance the overaggregation overhead.

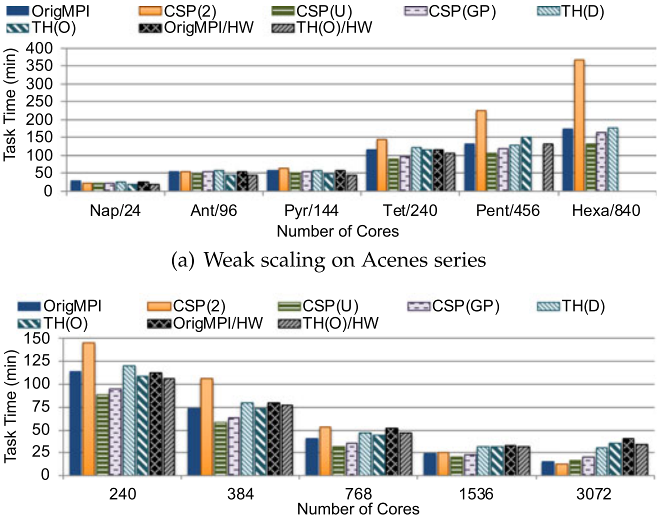
(T) Portion Phase: Fig. 12c shows the overhead profiles of the noniterative phase: (T). With OrigMPI, the heavy computation takes 30 minutes, and GETs dominate the other half of the cost. The overhead of GETs clearly indicates the delay caused by lack of asynchronous progress. All the static approaches can asynchronously complete GET operations; thus such overhead can be eliminated. With more cores dedicated to ghost processes or threads, however, the computation resources are also reduced, resulting in significant degradation in the computation. The TH(O) approaches show similar degradation because of core oversubscription.

8.2.2 Dynamic Adaptation

We next evaluated the dynamic adaptive strategies on both the Edison and Bebop platforms.

Weak and Strong Scaling: We evaluated CSP(U) and CSP (GP) in both weak and strong scaling of the Acenes problems, by comparing them with OrigMPI and the static approaches studied in the preceding section. In both the static and adaptable Casper approaches, we specified two ghost processes per node. In CSP(U), we specified `CSP_ASYNC_CONFIG=ON` and {`OFF`, `OFF`, `ON`} as the value of the `async_config` infos passed to three internal phases: 4-index, CCSD iteration, and (T) portion, respectively. In CSP (GP), we specified the thresholds as listed in Table 5.

Fig. 13 shows the results on Edison. In the weak-scaling graph, we increase the problem sizes and numbers of cores. The static CSP(2) shows significant degradation in the execution of all multinode problems. TH(D) does not achieve any improvement. TH(O) improves the execution by up to 20.0 percent at Ant; however, it also shows a 15.5 percent degradation at Pent. The adaptable CSP(U) and CSP(GP), on the other hand, consistently improve the execution for each problem type by up to 23.2 percent at Hexa and 16.3 percent at Tet, respectively. In the strong-scaling graph, both static Casper and the thread-based approaches show consistent degradation in performance, while CSP(U) and CSP(GP) can resolve such inefficiency. CSP(U) delivers the best performance by utilizing user hints, achieving up to 21.8 percent speedup; CSP(GP) provides a fully automatic solution; and it improves performance up to 16.3 percent. When scaling to 3,072 cores, CSP(2) becomes the best option because the 4-index becomes dominated by numerous *all-to-all* GETs that benefit from asynchronous progress with only two ghost processes.



(a) Weak scaling on Acenes series

(b) Strong scaling on Tet-aug-cc-pVDZ

Fig. 13. Dynamic adaptation with NWChem CCSD(T) on Edison (using 2 ghost processes in all Casper approaches). The unfinished bars in (a) are because of out-of-memory error.

Fig. 14 shows the weak scaling on Bebop. TH(O) delivers significant overhead because it oversubscribes without hyper-threading. CSP(GP) shows higher overhead than the results on Edison because of the overestimated range of *CR* thresholds. For instance, the 60 percent `LOW_CR` used in Ant was generated by the nonblocking GET patterns in preprocessing (see Fig. 5h), which is never used in the application. This caused the delay of adaptation in (T).

Internal Phase Overhead: We chose the Tet problem with 240 cores of Edison as the base of our profiling. We first compared the overhead of each phase with different approaches. We also added CSP(P) (see definition in Table 3) in this experiment. As shown in Fig. 15a, both CSP(U) and CSP(GP) can correctly resolve the overhead in the 4-index and improve the performance for the (T) portion, but CSP(P) cannot improve the overall performance because of the expensive overhead in the (T) portion.

We then compared the overhead distributed in each phase. Fig. 15b indicates that all the adaptive approaches resolve the overhead caused by overaggregated ACCs in the 4-index. With regard to the (T) portion, as shown in Fig. 15c, CSP(U) behaves the same as CSP(2) because it re-enables the redirection at the beginning of (T). CSP(P), on the other hand, cannot reduce the GET overhead because no synchronization call

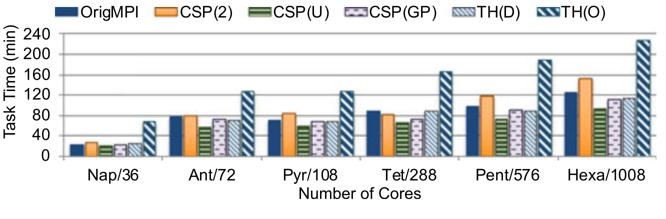
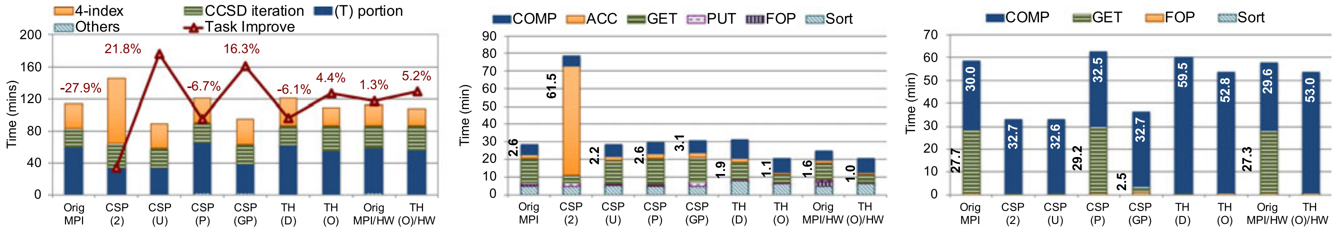


Fig. 14. Weak scaling for dynamic adaptation with NWChem CCSD(T) on Bebop (using 2 ghost processes in all Casper approaches).



(a) Task execution time (b) 4-index phase (showing ACC values) (c) (T) portion (showing COMP/GET values)

Fig. 15. Profiling CCSD(T) for Tet-aug-cc-pVDZ with adaptation over 240 cores on Edison (using 2 ghost processes in all Casper approaches).

exists in the application code. CSP(GP) eliminates the overhead of GET by re-enabling asynchronous progress through ghost-offloaded synchronization. In addition, we notice an overhead of close to 3 minutes in the GET and FOP portions in CSP(GP) compared with CSP(U), because of the interval set for ghost synchronization.

We also observed that although CSP(GP) predicts on each process separately, the majority of the processes always make the same decision (e.g., 99 percent of the processes disabled redirection in the 4-index, and all of them enabled in (T)).

9 Related Work

In this paper, we focus on the use of Casper in multiphase applications, and we propose several adaptive methods to dynamically reconfigure asynchronous progress to resolve operation aggregation imbalance. We divide the related work into two broad topics: communication asynchronous progress and dynamic adaptation for load balancing.

Communication Asynchronous Progress: *Thread-based asynchronous progress* is considered the most common approach for supporting software progress and is found in many MPI implementations such as MPICH and its derivatives [6], [7], [8]. This model allows every MPI process to utilize a background thread to asynchronously handle the incoming messages from other processes. While being a generic approach for various MPI communication models, this approach also raises the restriction that a background thread can make progress only for the process that spawned it. Thus it has to deploy at least as many background threads as MPI processes on every computing node. Consequently, the user must choose either to dedicate half of the computing resources or to involve expensive core oversubscription. Furthermore, this model forces MPI runtimes to support multithreaded safety, which may result in further overheads because of thread synchronization [10].

PIOMan [21] is a multithreaded communication engine supporting thread-based asynchronous progress. It divides rendezvous handshakes into multiple tasks and offloads them to background threads running only on idle cores. This approach, however, also suffers from a non-negligible overhead derived from the necessary multithreaded safety [22].

Vaidyanathan et al. [23] contributed an approach for asynchronous progress in the “MPI+X” model by utilizing a dedicated thread together with a lock-free command queue. The “MPI+X” model often utilizes multiple threads over multi- or many-core systems to parallelize computation and employs only a single MPI process per node for internode communication. Thus, only a single asynchronous thread is required per node.

The other well-known approach in the MPI community is the *interrupt-based asynchronous progress*, which has been supported on both Cray [24] and IBM systems [25][26]. This approach assumes that all processes are busy in external computation, thus utilizing a system interrupt to awaken the kernel thread to asynchronously complete incoming messages. The design is straightforward; however, the implementation often relies on a platform-specific lightweight interrupt engine; otherwise, severe performance degradation might occur because of frequently issued interrupts [20].

Supporting asynchronous progress is an essential task for using the portable MPI in other runtime systems. Daily et al. [27] proposed the approach to build the PGAS ComEX runtime on top of MPI two-sided model, and they designed a progress rank engine in ComEX that splits the MPI world communicator and uses a subset of processes to help communication progress.

Dynamic Adaptation and Load Balancing: Dynamic adaptation is a popular approach to dynamically balance irregular workloads or adapt heterogeneous execution environment and communication methods in both application and runtime systems. Flaherty et al. [28] and Biswas et al. [29] introduced their dynamic load balancer approaches for irregular workloads in mesh applications by repartitioning domains. As examples of runtime-level adaptation, Bhandarkar et al. [30] proposed an MPI implementation on top of the Charm++ environment that provides support for processor visualization and balances the workloads by dynamically measuring idle time or through user hints. Some researchers [31], [32] concentrated on generic autonomic runtime management for workloads on distributed-memory systems by implementing dedicated system modules.

Different from these works, we propose adaptive strategies in a portable MPI asynchronous progress library to resolve the operation overaggregation imbalance when providing asynchronous progress.

10 Conclusion and Future Work

Casper is a portable process-based asynchronous progress model for MPI RMA on multi- and many-core architectures. Our previous work presented the basic framework of Casper that sets aside a small number of cores as background ghost processes and redirects the user RMA operations targeting an application process to the ghost process, thus enabling asynchronous completion of RMA communication. This redirection-based design, however, might also result in operation overaggregation bottlenecks because of the limited progress resources, especially when communication becomes dominant. Therefore, a performance tradeoff has to be made in multiphase applications.

In this paper, we proposed an adaptive mechanism for Casper that resolves the overaggregation issue by disabling operation redirection in communication-intensive phases without affecting the benefit of asynchronous progress in other computation-heavy phases.

We chose an approximate prediction model in the adaptation to detect performance in order to maintain the portability of Casper. This model relies on offline prepossessing to sample the system performance matrices from a large set of benchmarks. However, it might be imprecise if the pattern of an application phase is not covered or large performance diversity exists among different patterns such as the trends observed on the Bebop system. Moreover, the real-time prediction can be further affected by several factors such as system noise or temporary network delay. Although we usually expect such noise to be small on high-performance supercomputers, we should give the issue careful consideration. Therefore, we plan to optimize the prediction model based on dynamic heuristic in future work.

Acknowledgments

This material was based upon work supported by the U.S. Dept. of Energy, Office of Science, Advanced Scientific Computing Research (SC-21), under contract DE-AC02-06CH11357. The experimental resources for this paper were provided by the National Energy Research Scientific Computing Center (NERSC) on the Edison Cray XC30 supercomputer and by the Laboratory Computing Resource Center on the Bebop cluster at Argonne National Laboratory. Antonio J. Peña is co-financed by the Spanish Ministry of Economy and Competitiveness under Juan de la Cierva fellowship number IJCI-2015-23266.

References

[1] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, W. A. de Jong, "NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations," *Comput. Phy. Commun.*, vol. 181, no. 9, pp. 1477–1489, 2010.

[2] T. Shiozaki, "BAGEL: Brilliantly advanced general electronic-structure library," *Wiley Interdisciplinary Rev.: Comput. Molecular Sci.*, vol. 8, no. 1, p. e1331, 2018, [Online]. Available: http://dx.doi.org/10.1002/wcms.1331

[3] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: A portable "Shared-Memory" programming model for distributed memory computers," in *Proc. ACM/IEEE Conf. Supercomputing*, 1994, pp. 340–349.

[4] H. E. Trease, "Code development for NWGrid/NWPhys," *Laboratory Directed Res. Develop. Annu. Rep.-Fiscal Year 2000*, p. 103, 2001, Art. no. 103.

[5] W. Jiang, J. Liu, H.-W. Jin, D. Panda, D. Buntinas, R. Thakur, and W. Gropp, "Efficient implementation of MPI-2 passive one-sided communication on InfiniBand clusters," in *Proc. Euro PVM/MPI*, vol. 3241, pp. 68–76, 2004.

[6] Argonne National Laboratory, "MPICH—High-performance portable MPI," 2014. [Online]. Available: http://www.mpich.org

[7] The Ohio State University, "MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE," 2014. [Online]. Available: http://mvapich.cse.ohio-state.edu

[8] Intel Corporation, "Intel MPI Library," 2014. [Online]. Available: http://software.intel.com/en-us/intel-mpi-library

[9] Cray Inc., "Cray XC series network," Cray Inc., Seattle, WA, USA, Tech. Rep., 2004. [Online]. Available: https://www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf

[10] W. Gropp and R. Thakur, "Thread-safety in an MPI implementation: Requirements and analysis," *Parallel Comput.*, vol. 33, no. 9, pp. 595–604, 2007.

[11] K. Kandalla, P. Mendygral, N. Radcliffe, B. Cernohous, D. Knaak, K. McMahon, and M. Pagel, "Optimizing Cray MPI and SHMEM software stacks for Cray-XC supercomputers based on Intel KNL processors," in *Proc. Cray User Group*, 2016.

[12] M. Gilge, *IBM System Blue Gene Solution: Blue Gene/P Application Development*. New York, NY, USA: IBM, 2013.

[13] M. Si, A. J. Peña, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa, "Casper: An asynchronous progress model for MPI RMA on many-core architectures," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2015, pp. 665–676.

[14] "MPI: A message-passing interface standard," 2015. [Online]. Available: http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[15] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood, "Remote memory access programming in MPI-3," *ACM Trans. Parallel Comput*, vol. 2, no. 2, 2015, Art. no. 9.

[16] M. Si, A. J. Peña, J. Hammond, P. Balaji, and Y. Ishikawa, "Scaling NWChem with efficient and portable asynchronous communication in MPI RMA," in *Proc. 15th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, May 2015, pp. 811–816.

[17] S. Hirata, "Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories," *J. Phys. Chem. A*, vol. 107, pp. 9887–9897, 2003.

[18] E. Aprà, et al., "Liquid Water: Obtaining the Right Answer for the Right Reasons," in *Proc. Conf. High Perform. Comput. Netw. Storage Anal.*, 2009, Art no. 66.

[19] J. S. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju, "Supporting the Global Arrays PGAS Model Using MPI One-Sided Communication," in *Proc. IEEE Int. Parall. Distrib. Process. Symp.*, May 2012, pp. 739–750.

[20] J. R. Hammond, S. Krishnamoorthy, S. Shende, N. A. Romero, and A. D. Malony, "Performance characterization of global address space applications: A case study with NWChem," *Concurrency Comput.: Practice Exp.*, vol. 24, no. 2, pp. 135–154, 2012.

[21] F. Trahay and A. Denis, "A Scalable and Generic Task Scheduling System for Communication Libraries," in *Proc. IEEE Cluster*, 2009, pp. 1–8.

[22] F. Trahay, É. Brunet, and A. Denis, "An analysis of the impact of multi-threading on communication performance," *2009 IEEE Int. Symp. Parallel & Distrib. Process. (IPDPS)*, pp. 1–7, May 2009, doi: 10.1109/IPDPS.2009.5160893.

[23] K. Vaidyanathan, D. D. Kalamkar, K. Pamnany, J. R. Hammond, P. Balaji, D. Das, J. Park, and B. Jo, "Improving concurrency and asynchrony in multithreaded MPI applications using software offloading," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, Nov. 2015, pp. 1–12.

[24] H. Pritchard, D. Roweth, D. Henseler, and P. Cassella, "Leveraging the Cray Linux Environment Core Specialization Feature to Realize MPI Asynchronous Progress on Cray XE Systems," in *Proc. Cray User's Group Meeting*, 2012, https://cug.org/proceedings/attendee_program_cug2012/by_auth.html

[25] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer, "The deep computing messaging framework: Generalized scalable message passing on the blue gene/P supercomputer," in *Proc. 22nd Annu. Int. Conf. Supercomputing*, 2008, pp. 94–103.

[26] S. Kumar, Y. Sun, and L. V. Kale, "Acceleration of an asynchronous message driven programming paradigm on IBM blue gene/Q," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 689–699.

[27] J. Daily, A. Vishnu, B. Palmer, H. van Dam, and D. Kerbyson, "On the Suitability of MPI as a PGAS Runtime," in *Proc. 21st Int. Conf. High Perform. Comput.*, Dec. 2014, pp. 1–10.

[28] J. E. Flaherty, R. M. Loy, C. zturan, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz, "Parallel structures and dynamic load balancing for adaptive finite element computation," *Appl. Numerical Math.*, vol. 26, no. 12, pp. 241–263, 1998.

[29] R. Biswas, S. K. Das, D. J. Harvey, and L. Oliker, "Parallel dynamic load balancing strategies for adaptive irregular applications," *Appl. Math. Model.*, vol. 25, no. 2, pp. 109–122, 2000.

[30] M. A. Bhandarkar, L. V. Kalé, E. de Sturler, and J. Hoeflinger, "Adaptive Load Balancing for MPI Programs," in *Proc. Int. Conf. Comput. Sci.-Part II*, 2001, pp. 108–117.

[31] J. Yang, H. Chen, S. Hariri, and M. Parashar, "Autonomic Runtime Manager for Adaptive Distributed Applications," in *Proc. 14th IEEE Int. Symp. High Perform. Distrib. Comput.*, 2005, pp. 69–78.

[32] T. Estrada and M. Taufer, "On the effectiveness of application-aware self-management for scientific discovery in volunteer computing systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, pp. 1–11.

Min Si received the MS and PhD degrees in computer science from the University of Tokyo, in 2016 and 2012, respectively. She is an Enrico Fermi postdoctoral scholar with the Mathematics and Computer Science Division, Argonne National Laboratory.

Antonio J. Peña is a senior researcher with the Barcelona Supercomputing Center (BSC). He works in the Programming Models group, leading the activity "Accelerators and Communications for HPC". He is also the manager of the BSC/UPC NVIDIA GPU Center of Excellence. He is a Juan de la Cierva fellow and prospective Marie Curie fellow.

Jeff Hammond received the PhD degree in chemistry from the University of Chicago. He is a system architect with Intel. He contributes to the development of open standards for parallel programming, including MPI, OpenMP, and OpenSHMEM, and is an enthusiastic developer of open-source software.

Pavan Balaji is a computer scientist and group lead with Argonne National Laboratory, a fellow with the Northwestern-Argonne Institute of Science and Engineering, Northwestern University, and a fellow with the Computation Institute, University of Chicago.

Masamichi Takagi is a senior scientist and member of the systems software development team with the Riken Advanced Institute for Computational Science, Japan.

Yutaka Ishikawa received the PhD degree in electrical engineering from Keio University. He is the project leader of Post K computer development with the Riken Advanced Institute for Computational Science, Japan. From 1993 to 2001, he was the chief with Parallel and Distributed System Software Laboratory, Real World Computing Partnership. From 2002 to 2014, he was a professor with the University of Tokyo.