博士論文

Techniques for Enabling Highly Efficient Message Passing on Many-Core Architectures

(メニーコア型大規模並列計算機向けの高性能メッセージパッシング型通信技術)

思　敏

ABSTRACT

Since multicore processors have become the most common processor architectures today, the next grade promotion for high end processors is expected to be achieved by improving both thread- and instruction-level parallelism. There are two kinds of architectures dominating the high performance market today, the GPU accelerators and the General Purpose (GP) many-core architectures. In this dissertation, we focuses on the latter. Many-core architecture, such as Intel Xeon Phi and IBM Blue Gene/Q, provides us a massively parallel environment containing dozens of cores and hundreds of hardware threads with powerful wide SIMD units. More and more scientific application developers have begun investigating ways to utilize such architecture for scaling application performance. However, the performance may be restricted in various ways. Unlike traditional CPUs, the performance capability of many-core architectures comes from massive low-frequency cores for better performance-to-energy ratio; thus sequential execution on such hardware could result in performance degradation. Furthermore, the other on-chip resources (e.g., memory) are not growing at the same rate as number of cores, potentially resulting in scalability issue.

Not only hardware architectures, the scientific applications are also moving toward complex hybrid and irregular models. In traditional regular applications (e.g.,Fast Fourier transform), more and more applications start focusing on hybrid programming models comprising a mixture of processes and threads, that allow resources on a node to be shared between the different threads of a process, especially benefiting the execution on many-core architectures. The most prominent of the hybrid models used in scientific computing today is MPI+OpenMP, where multiple OpenMP threads parallelize the computation, while one or more threads utilize MPI for their data communication. On the other hand, despite the well studied regular applications, a number of applications are becoming extremely dynamic and irregular especially in chemistry and bioinformatics domains. MPI-2 and MPI-3 introduced one-sided communication mode, which is more suitable for supporting the data movements in such irregular model rather than the MPI two-sided or group communication modes.

With growing complexity in both computing hardware and scientific applications, various critical communication issues raise up and resulting in severe degradation in application performance. This dissertation focuses on exploiting the capabilities of advanced many-core architectures on widely used message passing model, in order to address the communication problems existing in the popular hybrid programming model and the irregular one-sided mode and consequently contribute efficient communication approaches for various kinds of applications.

Firstly, in hybrid MPI+threads applications, a common mode of operation for such applications involves using multiple threads to parallelize the computation, while one of the threads issues MPI operations. Although such mode extremely improves floating point performance for computation of applications by massive parallelism, it also means that *most of the threads are idle during MPI calls*, which translate to underutilized hardware cores. Furthermore, since *only single low-frequency core is contributing to communication*, it may result in even performance degradation. To address the core idleness issue and improve the performance of communication, we propose an *internally multithreaded MPI* as the first contribution of this dissertation, that transparently coordinates with the threading runtime system to share idle threads with the application in order to fully utilize the computing resources as well as parallelizing MPI internal processing such as derived datatype communication, shared-memory communication, and network I/O operations for better performance.

Secondly, with regard to the irregular one-sided communication, however, the MPI standard does not guarantee that such communication is truly asynchronous. Most MPI implementations still require the remote target to make MPI calls to ensure progress on such operations, consequently the *operation cannot complete at the target without*

explicit processing in software and thus may cause arbitrarily long delays if the target process is busy computing outside the MPI stack. Traditional implementations to ensure asynchronous completion of operations have relied on thread-based or interrupt-based models. Each of these models has several drawbacks, however, such as the inefficient core deployment in the thread model and the expensive overheads caused by multithreading safety in the thread model and by frequent per-message interrupts in the interrupt model. To address these drawbacks, we propose *Casper, a process-based asynchronous progress model for MPI one-sided communication* on multicore and many-core architectures as the second contribution of this dissertation. The central idea of Casper is to keep aside a small, user-specified number of cores on a multicore or many-core environment as "ghost processes," which are dedicated to help asynchronous progress for user processes through appropriate memory mapping from those user processes. Whenever user application issues an RMA operation to a user process Casper then transparently redirects such operation to the ghost process thus ensuring asynchronous completion. This approach has successfully resolved the communication bottleneck in the widely used NWChem quantum chemistry application by achieving up to 30 % performance improvement in the "gold standard" CCSD(T) simulation.

Although Casper provides simple but efficient asynchronous progress for irregular one-sided communication, *the performance might not be optimal in a number of applications that always consist of multiple phases with varying proportion of communication and computation.* Inefficient usage of asynchronous progress may even result in performance degradation. That is, the computation-intensive phase heavily relies on asynchronous progress, however, the communication-intensive phase does not have strong needs of asynchronous progress but more focuses on the load balance for large amount of RMA operations, which might not hold in Casper since the operations are consistently redirected to a few ghost processes. As the third contribution of this dissertation, we propose *a dynamic adaptation mechanism embedded in Casper that transparently adapt the configuration of asynchronous progress for multi-phases applications.*

Finally, apart from the lack of asynchronous progress, many irregular applications also suffer from loss of performance in a number of ways. For example, it is usual in imbalanced communication that an MPI process takes long time to wait for a message to arrive, the core on which it is scheduled is idle and underutilized. To comprehensively address these issues, we plan to investigate *the concept of user-level processes, a way to provide multiple co-scheduled "OS processes" on a single core as the MPI processes, with exploiting the potential optimization in MPI communication runtime, such as better load balancing and light-weight checkpoint migration*, as the future work of this doctoral research.

論文要旨

近年、マルチコアプロセッサが広く普及しているが、消費電力と発熱問題などの理由により、従来通りの動作周波数向上によるプロセッサ処理性能向上は困難になっており、コア数及び SIMD 命令の並列化により性能を向上する傾向になる。インテル社の Xeon Phi や、IBM 社の Blue Gene/Q などのメニーコアアーキテクチャーは数十コア・数百ハードウェアスレッド・ワイド SIMD 搭載のような大規模並列環境を提供している。しかしながら、このようなアーキテクチャーは従来のプログラミングモデルに適していない。普通の CPU プロセサーと違って、メニーコアチップの性能は、省エネルギーのために、大量な低動作周波数のプロセッサコアにより大規模並列計算の形で提供する。それで、アプロケーション全体を大規模並列化しないと逆に性能が悪化することを注意しなければならない。また、コア数が増加する割に、メモリ容量などの他のシステム資源があまり増加していないため、コアごとの資源が少なくなっていることもアプリケーションのスケーラビリティを制限する可能性がある。

一方、ハードウエアの進化とともに、科学計算アプリケーションのほうも複雑なハイブリッド及び非規則計算・通信モデルになる傾向がある。従来の規則系アプリケーション（例えば、高速フーリエ変換）では、メモリやその他のシステム資源に比べて大量のコアが提供されてきている理由から、プロセスとスレッドを混在させるハイブリッドプログラミングモデルに移行している例がある。このようなモデルでは、同一プロセスに所属する複数スレッドがノード内資源を共有することが可能となる。スレッドモデルの代表的処理系である OpenMP と分散メモリシステム上で通信機能を担当する MPI ライブラリの組み合わせがこのモデルの主流である。また、非規則系の計算・通信モデルのアプリケーションも化学・バイオインフォマティクス領域で出現する。MPI-2 及び MPI-3 規格から定義された片方向通信、いわゆる「Remote Memory Access、RMA」は通信の相手の状態に無関係に他のプロセスのデータにアクセスできる通信モデルであり、この非規則系通信モデルに適している。

ハードウエア構造の変化とソフトウエアモデルの複雑化に従い、それぞれのプログラミングモデルにいくつかの通信問題が出現し、アプリケーション全体の性能を極めて制限する。アプリケーションの構成の違いにより、計算・通信プログラミングモデルの最適手法がそれぞれあるが、メニーコア上でそれらのモデルを効率的に実行する処理系は未成熟である。本論文は、科学計算に広く用いられるメッセージパッシング通信モデルを対象としてメニーコアの特徴を活用して通信性能を最大限に発揮し、あらゆるプログラミングモデルの既存課題を解決して全体性能の向上に貢献する。

まず、ハイブリッド MPI+スレッド型アプリケーションでは、複数の OpenMP スレッドが計算を並列化してその中の 1 つが MPI 通信を行うという実行モデルが主流である。このような実行パターンでは、浮動小数点計算を大規模並列化することにより計算部分の性能向上が達成されるが、通信部分では殆どのスレッドがアイドルになり、計算資源が無駄になる。また、1 つのコアだけが通信処理を担当することにより通信性能の劣化原因ともなる。本論文の第一の貢献は、この問題に対してアイドル状態になったユーザアプリケーションが作成したスレッドを再利用して、ユーザ定義データ型通信、共有メモリ通信とネットワーク I/O 作業などの MPI 内部通信作業を並列化する手法を提案する。本手法により、計算部分だけでなく、通信を含めてアプリケーション全体がメニーコア資源を利用でき、

全体性能が向上することを示す。

次に、非同期可能な通信処理がたくさんあっても、MPI 規格ではこのような通信が必ず非同期処理されるとは限らない問題点について取り組む。片方向通信であっても、RMA 通信を完了するためにリモート側プロセスが MPI を呼び出さないと処理が進まない MPI 実装が殆どである。リモート側が MPI を呼び出すまで、通信処理がローカル側で完了できず、更にリモート側が計算中のためローカルプロセスが長時間待ちになる恐れもある。既存研究では、バックグラインドスレッドの手法とシステム割込みに基づく手法が殆どであるが、非効率的な計算コアの配置、マルチスレッドレベルや頻繁的なシステム割込めが生成した重いオーバーヘッドなどの欠点が挙げられる。本論文の第 2 の貢献として、本課題に対してメニーコアの特徴を活用し、プロセスレベルで MPI 非同期通信専用コアを実装し、最適な非同期通信処理手法を提案する。本手法では、ユーザが任意的に計算プロセスに使うコア数と非同期通信を担当するゴーストプロセスに使うコア数を指定でき、従来のアプローチよりマルチスレッドレベルやシステム割込みのオーバーヘットを軽減するとともに、コア配置に対して優れた柔軟性も達成する。更に、PMPI リダイレクト機能を利用した MPI 外部実装方式を採用することにより、あらゆる MPI 実装にも容易にサポートできる利点もある。この非同期手法を利用して、汎用量子化学計算パッケージ NWChem の性能を大幅に向上でき、特に重要な CCSD(T) シミュレーションに対して 30 %ほどの性能向上も達成する。

ところが、大規模計算プログラムにいつもマルチ計算段階が含まれ、通信・計算の比率が変わりつつある理由で、この静的非同期通信の設計は最適とは言えない。例えば、計算が重い段階におけて非同期通信が必要であるが、通信が重くなると、少数の非同期プロセスを使うことより多めの計算プロセスを使って通信負荷分散という手法のほうが効率である。本論文の第 3 の貢献として、NWChem を実例として各段階の計算・通信性能特徴を深く解析し、マルチ計算段階に対して動的に非同期通信を自動適応できる機能を提案する。

最後に、非同期通信の他に、MPI プロセスだけで記述されたアプリケーションに性能を大幅に影響する課題がまだいくつかある。例えば、1MPI プロセスがメッセージを待つ時、メッセージが到着するまでその計算コアはアイドルになり、コア性能を発揮できなくなる。次の研究計画としては、1 コア上で OS プロセスを複数スケジュールできるユーザレベルプロセスのアプローチに基づき、大量の MPI プロセスを 1 コア上で実行して負荷分散やチェックポイントの軽量化などの面から改良手法を着手する予定である。

Acknowledgements

I would like to express my deepest gratitude to Prof. Yutaka Ishikawa, for guiding me to the inspiring high performance computing domain, and for his invaluable and continuous support and encouragement for my life, study and research career over the past five years.

I express my sincere gratitude to Dr. Pavan Balaji and Dr. Antonio J. Peña for the greatest guidance and mentoring leading me into the challenging MPI communication world during my study at Argonne National Laboratory.

My sincere thanks goes to Prof. Reiji Suda, for his considerate and patient guidance in the past year. It is impossible for me to finish the doctoral research remotely without Prof. Suda's help and support.

I thank Dr. Atsushi Hori, Dr. Jeff Hammond, Dr. Masamichi Takagi and Akio Shimada for their support in the collaborative works.

I also thank all the members in Ishikawa lab and in the PMRS group at Argonne for their friendly encouragement and help during my study.

Finally my warmest thanks goes to my dear husband and parents. Without their endless support and encouragement, I would not be able to concentrate on the research and finish this dissertation.

The work in this dissertation has financially supported by (1) the CREST project of the Japan Science and Technology Agency (JST) and the National Project of MEXT called Feasibility Study on Advanced and Efficient Latency Core Architecture and (2) the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357.

The experimental resources for this paper were provided by the Texas Advanced Supercomputing Center (TACC) on the Stampede supercomputer, by the National Energy Research Scientific Computing Center (NERSC) on the Edison Cray XC30 supercomputer and by the Laboratory Computing Resource Center on the Fusion cluster at Argonne National Laboratory.

Contents

Chapter 1

Introduction

Over the past few decades, high performance computing (HPC) has dramatically revolutionized the process of scientific advancement. The power of supercomputers has been heavily used in various scientific fields including climate forecasting, nuclear development, material innovation and so forth. HPC has been considered as the lever that accelerates scientific discovery and shortens the time for technology to benefit real-world.

Thanks to Moore's Law, the speed of HPC performance was growing at exponential rate in the previous decade by improving the density of transistors on single core, which marked the increase of computing power from Terascale (ASCI Red supercomputer, installed at Sandia National Laboratories in 1996 [3]) to Petascale (Roadrunner supercomputer, deployed at Los Alamos in 2008 [4]). This approach could provide most applications immediate benefits without significant change in software since it directly accelerated the speed of single threads. However, such performance improvement has also brought in similar trend in the power consumption [66]. And indeed, the arrival of petaflop supercomputers coincided with processors hitting the power wall whereby any additional increase in the power usage of a processor would result in the processor's components melting or becoming extremely unreliable. Besides, such approach also suffers from physical and economical limitations [7, 54]. Consequently, instead of instruction-level parallelism, processor architects started to move to a higher level (i.e. threads) for continuous performance advancement.

Accordingly, multi-core architectures have become the norm for high-end computing systems now a days [2]. Even personal mobile devices started to use two or more cores to get better performance (e.g., Quad-core Samsung Galaxy Note5, Dual-core iPhone 6). The traditional multi-core processors, however, get hard to increase more cores on chip due to the high risk of contention in shared resources among cores such as bus, cache and memory. In fact, many researches have already looked into this problem on multiple-core systems and had to change their software design for better performance [22, 74].

Parallel with the advancements of multi-core processors, manufacturers started to explore microprocessor design in another direction where hundreds and thousands simple cores are embedded into single chip to form a massive parallel computational environment, called many-core. A broad vision of this kind of architectures can cover any designs that follow the form of massive simple core units, including General-Purpose Graphics Processing Units (GPGPUs) [50] which con-

tribute to highly data parallelizable floating-point computing, the Tilera processors [40] that more focus on commercial networking server farms which rely on high throughput, and the Intel Many Integrated Core (MIC) architectures as the intermediate path between traditional general purpose CPU and the floating-computing concentrated GPU architectures. This dissertation focuses on the Intel MIC architectures.

Table 1.1: Node Configuration in Multi-Core and Many-Core Supercomputers [2].

Year	Name	Cores/Threads	Clock Speed	Memory
2009	Jaguar (Cray XT5)	6/6	2.3GHz	16GB
2011	K (Fujitsu SPARC64)	8/8	2GHz	16GB
2012	Mira (IBM BG/Q)	16/64	1.6GHz	16GB
2012	Stampede (Intel KNC)	61/244	1.1GHz	8GB
2013	Tianhe-2 (Intel KNC)	57/228	1.1GHz	8GB
2016	Cori (Intel KNL)	60+/240+	-	16GB

The many-cores architectures (i.e., Intel Xeon Phi) have entered the HPC market in 2012 [1, 6], providing users a higher degree of massive parallelism with dozens of cores and hundreds of hardware threads with relatively easy-to-start programming environment since all the applications written for traditional CPU systems can be easily executed on this new platform without significant modification in code. Unlike traditional multi-core processors, the on-chip bus interconnection and cache coherency are carefully designed for better sharing among large amount of cores, thus minimizing the contention issues existing in multi-core systems. However, such architecture does not do magic. The many-core parallel environment does not bring us equal improvement in performance and scalability as the increase of cores if we just run our applications as the ways on traditional processors. Application developers have to investigate the appropriate way to fully utilize such hardware in HPC programming. By comparing the hardware configuration in the top-ranked multi-/many-core supercomputers from 2009, Table 1.1 clearly indicates two special trends in the renovation of high-end processors that need to be taken into account.

- Each single core is designed to be simple and low frequency for a better performance-to-watt ratio; thus, execution on a single core could result in extreme performance degradation comparing to that on traditional CPU.
- The number of computing cores is growing at a much faster rate than the other on-chip resources (e.g., memory), thus potentially resulting in scalability limitation.

Not only the restrictions in hardware architectures, the increasing variety of applications also aggravates the complexity in parallel programming. The message passing programming model defines a mechanism that coordinates multiple processes for resolving large computational problems by passing messages between each other. The Message Passing Interface (MPI) [48] standardizes this

model, MPI-1 standard introduced the classical two-sided message passing (e.g., `MPI_Send`/`MPI_Recv`) and the collective communication (e.g., `MPI_Bcast`), MPI-2 and MPI-3 introduced the one-sided communication, as known as the Remote Memory Access (RMA) model. MPI has been the "de facto" industry standard for parallel programming on distributed memory clusters and supercomputers for more than two decades.

The solution of many mathematical problems in scientific applications can always be decomposed into regular meshes and parallelized across all processes (e.g., Fast Fourier transform, LU decomposition). The classical MPI-1 communication functions have perfectly supported the regular data movement in those parallel algorithms for years. Within advanced computing systems, researchers are eagerly looking into larger scale and more complex scientific problems, many of them requiring larger and larger memory capacity [41]. However, as we have compared in Table 1.1, this does not match the trend we have seen in hardwares. To ease the memory crisis among large amount of cores on the many-core systems, application developers are increasingly looking at the hybrid "MPI+X" programming model, comprising a mixture of processes and threads, that allows resources on a node to be shared between the different threads of an MPI process. Such a model, however, also increases the complexity in MPI and results in inefficient communication due to limitations in software and hardware especially on the many-core systems.

Besides these traditional regular applications, a number of applications start to drive more dynamic and irregular data movements, especially in chemistry and bioinformatics domains [17, 47, 76]. Most of these applications always involve extreme big data with enormous irregular communication (e.g, using MPI one-sided operations). However, current HPC systems are not yet ready to efficiently handle these computations, severe performance degradation has been observed in many of those applications. One example is in the quantum chemistry application NWChem [76], the communication overhead can even dominate the entire execution cost by more than 50%. Such communication challenge cannot be resolved by only improving the speed of network interconnection, more importantly, it is limited by the traditional hardware design of network devices. That is, the network devices are connected as PCI-e device, which does not have the control of CPUs for handling incoming message on chip, resulting in arbitrary delay if CPU cores are being used by user applications or other system tasks. Unfortunately, there will be still years to completely bring up the asynchronous capability in network hardware.

This dissertation aims to exploit the capabilities of many-core architectures on widely used message passing model, in order to address these issues existing in different programming models and consequently contribute efficient communication approaches for various kinds of applications.

1.1 Problem Statement

To better utilize the hardware resources on modern multi- and many-core architectures, application developers have studied several approaches for regular and irregular scientific applications in order to achieve better parallelism and resource

sharing. Many of those approaches, however, still face communication problems that result in performance degradation. Here we summarize two critical issues existing in the most popular programming models used in modern applications.

Inefficient communication and core idleness in hybrid MPI+threads model. The hybrid "MPI+threads" programming model has become popular in a rang of applications in recent years. Unlike traditional MPI programming model, it allows resources to be shared between different cores on the node which is especially suitable for parallel programming on many-core environment since the memory capacity per single core is reducing. A common mode of operation in such hybrid models involves using multiple threads to parallelize computation within the node, but using only one thread to issue MPI communication. Although such a mode achieves significant improvement in floating-point computing by massive parallelism without involving heavy thread overhead or complex semantics in MPI, it also means that most of the threads are idle during MPI calls, a situation that can be translated to underutilized hardware cores. Furthermore, since MPI communication performs only on a single low frequency core, this mode may even result in performance degradation.

Lack of asynchronous progress in MPI one-sided communication. An increasing number of applications are looking at the MPI one-sided communication model which provides natural dynamic and irregular semantics of data movements. It is especially important for many-core programming, because many large memory applications rely on a global shared address model that supports the ability to share memory resource across nodes by employing the MPI one-sided model for internal data movements [24]. The MPI-2 and MPI-3 standards [5] introduced the one-sided communication, which allows one process to specify all communication parameters for both sender and receiver. Thus a process can access the memory regions on other processes without the remote process explicitly needing to receive or process the message. Although such communication semantics is able to asynchronously handle communication progress and hence hide communication cost from computation, it is not truly asynchronous in most MPI implementations. For example, although contiguous PUT/GET operations can be implemented in hardware on RDMA-supported networks(e.g., InfiniBand, Fujitsu Tofu, Cray Aries) thus allowing the hardware to asynchronously handle its progress semantics, complex RMA communication such as the heavily used non-contiguous accumulate operation (e.g, an accumulate on a three-dimension double subarray) must still be done in software within the MPI implementation. Consequently, the operation cannot complete at the remote process without explicitly making MPI progress and thus may cause arbitrarily long delays if the remote process is busy computing outside MPI.

1.2 Contributions

This dissertation focuses on the communication optimization in various programming models executed on many-core architectures. We propose efficient solutions to resolve the two critical challenges we have listed in the above section. The contributions of this dissertation can be summarized as follows.

Multithreaded MPI communication. To resolve the problems in the MPI communication of hybrid "MPI+Threads" model, we present MT-MPI [59], an internally multithreaded MPI that transparently coordinates with the threading runtime system to share idle threads with the user application in order to parallelize MPI internal processing such as derived datatype communication, data transfer in shared-memory communication, and network I/O operations.

Process-based asynchronous progress model. To resolve the problem of asynchronous progress in irregular applications, we propose Casper [61], a process-based asynchronous progress model for MPI one-sided communication on multi-core and many-core architectures, that dedicates a small user-specified number of cores as background "ghost processes" to help asynchronous progress. The philosophy of Casper is centered on the notion that since the number of available cores in modern many-core systems is increasing rapidly, some of the cores might not always be busy with user computation and can be dedicated to helping with asynchronous progress.

Dynamic adaptable asynchronous progress. Many of complex scientific problems always require integration of multiple fundamental solvers and algorithms into application execution, each of the phases always performs very different characteristics of communication and computation. Thus it is hard to statically determine whether the asynchronous progress is needed or not in these applications. To achieve the optimal performance for the multi-phases applications, we propose a dynamic adaptation mechanism integrated in the Casper library, providing the capability to dynamically predict the needs of asynchronous progress for different execution phases and transparently adapt asynchronous progress.

1.3 Outline

The rest of this dissertation is organized as follows.

In Chapter 2, we first give an overview of the many-core architecture and introduce the semantics of the popular hybrid programming model and the irregular RMA model with several real applications as the background of this doctoral research.

In the following three main chapters, we then discuss each contribution of this dissertation with detailed description around the motivation, the design challenges and the implementation, and the evaluation from micro- and macro-kernels to real applications. Specifically, Chapter 3 discusses the inefficient communication and the core idleness issue in the hybrid MPI+threads programming model, and presents the multithreaded MPI approah that aims to transparently share user idle threads inside MPI communication. Chapter 4 focuses on the asynchronous progress issue existing in irregular MPI one-sided communication model, and presents the process-based asynchronous progress model, named "Casper". Then Chapter 5 looks into the usability of Casper in complex multi-phases applications, and we present a dynamic adaptation technique that automatically adjust the asynchronous progress for multiple phases of application which involve varying communication characteristics.

Chapter 6 summarizes related works focusing on the hybrid programming models, the MPI one-sided communication or the asynchronous progress models. Finally, we conclude this dissertation in Chapter 7 with discussion for the future works we plan to address for the communication optimization on many-core architectures.

Chapter 2

Background

2.1 Many-Core Architectures

Till the beginning of this century, rapid growing rate of CPU frequency has successfully pushed forward the high performance computing into petascale. However, such improvement is not free, we had to pay for increasing cost of per core power consumption that even raised up the power wall ceasing any frequency growth. Consequently, single processors can no longer become faster, the only way to improve performance for high-end processors is to add more cores and hardware threads.

Many-core architectures provides applications such massively parallel environment and have already being successfully used in several most powerful supercomputers in the world. For example, both the world's No.1 system, Tianhe-2 developed by China's National University of Defense Technology [6], and the No.10 system, Stampede located at Texas Advanced Computing Center [1] use the Intel Xeon Phi coprocessors to accelerate their computation; Mira at Argonne National Laboratory, an IBM BlueGene/Q supercomputer ranked at No.5 in the world, also forms as many-core embedded platform [9]. In this section, we introduce the basic structure and programming environment of a typical many-core product, the Intel Many Integrated Core (MIC) architecture.

The Intel MIC architecture features a large amount of CPU cores inside single chip with Linux-based operating system. It provides applications a similar programming and execution environment as the normal CPU systems, with supporting massive parallelism and vector capability to achieve high floating-point performance. Different from the GPU accelerators, the many-core chip can be run as both floating-point accelerator, and a standalone system.

Intel published the first commercial release of MIC architecture, codenamed Knights Corner (KNC) in 2012 [19, 35]. It provides a minimum of 60 light-weight cores and separate GDDR5 memory embedded on single chip, with each core capable of supporting four hardware threads and a 512-bit SIMD vector processing unit (VPU). As shown in Figure 2.1, all of the cores have fully private and coherent cache: 32 KB instruction + 32 KB data L1, and 512 KB L2 (unified), with high bandwidth bidirectional ring interconnection. Eight dual-channel GDDR5 memory controllers (MC) are symmetrically distributed on the ring to provide high bandwidth access to the 8 GB or more GDDR5 memory from any cores.

The KNC coprocessors is usually connected with host CPU cores and internode communication device (e.g., InfiniBand) via PCI-express on each computing

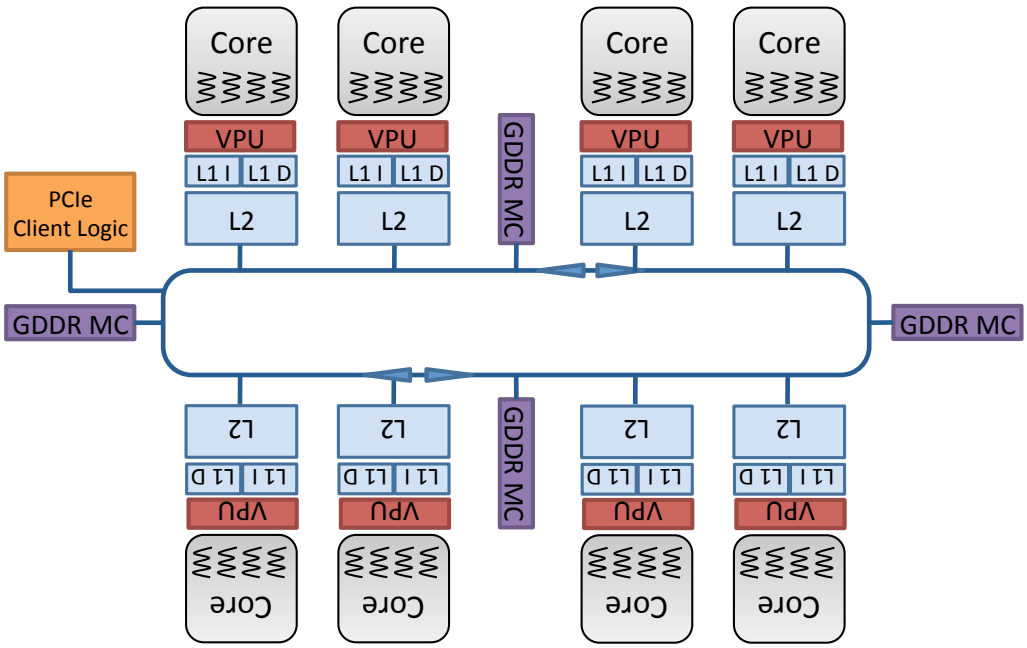


Figure 2.1: Knight Corner Chip Constriction.

node of Xeon Phi based supercomputers as demonstrated in Figure 2.2. For example, the Stampede supercomputer [1] employs one KNC chip (SE10P) connected to two Intel E5 8-core (Sandy Bridge) processors on each node; the Tianhe-2 supercomputer is constructed as three Xeon Phi chips with two Intel Ive Bridge processors per computing node [6].

The KNC provides three programming models for application development: native, offload or symmetric. In the `native mode`, KNC's own micro-Linux operating system manages the on-chip resources and exposes comprehensive system calls to support user programs running with both MPI and threads parallelism. For some of the system calls that cannot be handled directly on the KNC card are transparently forwarded to the host CPU and returned with the result received from host after the execution. Thus the applications can directly run on the KNC environment similar as that on traditional CPU systems. With regard to the internal communication between MPI processes, processes located on the same card communicate with each other through shared memory; processes on the same computing node but located on two KNC cards communicate using the PCIe peer-to-peer capabilities; for the communication outside the node, the capability of direct data transfer without host intervention has been provided on some networks such as InfiniBand. On the other hand, the `offload mode` offers the possibility of running as an accelerator like GPUs, and the `symmetric mode` can be used in MPI applications where processes are distributed on both KNC and host CPUs. We only focus on the `native mode` in this thesis.

Furthermore, Intel has recently also announced the details of its next generation of the Xeon Phi product family, codenamed Knights Landing (KNL). KNL is a fully self-hosted architecture that can offer applications the standalone execution environment similar but more comprehensive compared to the `native mode` on KNC card. Greater than 60 cores with four hardware threads and two powerful VPUs each are embedded on single chip with more complex but high

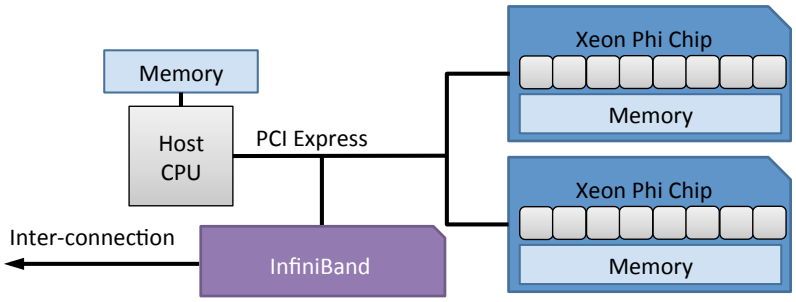


Figure 2.2: Computing Node Structure on Xeon Phi supercomputers.

bandwidth mesh interconnection. This design allows 3x single thread performance compared to KNC and achieve more than 3 TeraFlops peak performance per singe socket node. At least two of the upcoming supercomputers, Cori at the National Energy Research Scientific Computing Center (NERSC) [18], and Theta at Argonne National Laboratory [26], have decided to be constructed using the KNL processors. More detailed information of the KNL architecture can be found at [37].

2.2 Hybrid MPI+Threads Programming

Although the number of cores is rapidly increasing on modern multi- and many-core architectures, the other system resources (e.g., memory, network endpoints) are not growing at the same rate. To efficiently utilize such large amount of threads with better resource sharing, application programmers are increasingly looking at the hybrid MPI + Thread model, where multiple threads are used to parallelize the computation on each computing node and MPI is used for the inter-node data communication. The most prominent of the threading models used in modern scientific computing is OpenMP [21], where applications add annotations in the code with necessary information of the parallelism (e.g., the number of threads, the parallel patterns and the property of variables), then the compiler can translate these annotations into appropriate commands and cooperate with the runtime system for task scheduling. In the rest of this section, we focus on the MPI+OpenMP programming.

Since MPI processes and threads are managed by two separate runtime systems, additional rules have to be made to ensure the thread safety inside MPI without resulting in unnecessary overhead. For example, a message may be concurrently matched by the receive calls from two threads on the same process if the appropriate thread safety is not provided; conversely, we should also avoid over-definition of the thread safety since it can result in significant overhead from heavy usage of memory barriers and lock acquiring/releasing in most MPI implementations even the program does not involve any threads [30].

2.2.1 Programming Model

In this section we introduce the different threading modes defined by MPI for multithreaded environments. The MPI standard provides four levels of thread safety.

```
#pragma omp parallel
{

  /* user computation */

}

MPI_Function();
```

(a) Outside a parallel region

```
#pragma omp parallel
{
  /* user computation */
  #pragma omp master
  {
    MPI_Function();
  }
}
```

(b) Inside omp `master` region

```
#pragma omp parallel
{
  /* user computation */
  #pragma omp critical
  {
    MPI_Function();
  }
}
```

(c) Inside omp `critical` region

```
#pragma omp parallel
{
  /* user computation */
  #pragma omp single
  {
    MPI_Function();
  }
}
```

(d) Inside omp `single` region

Figure 2.3: Different use cases in hybrid MPI+OpenMP.

MPI_THREAD_SINGLE

In this mode, only a single thread exists in every MPI process. This model is commonly referred to as the MPI-only model, where multiple MPI processes communicate with each other and no threads are involved.

MPI_THREAD_FUNNELED

In this mode, multiple threads can be created for parallelizing the computation phases on every MPI process, but only the master thread is allowed to access MPI stack. In an OpenMP program, this can be implemented as either making MPI calls outside the OpenMP parallel region or protecting the MPI calls with OpenMP master regions. Figure 2.3(a) and 2.3(b) demonstrate those implementation respectively.

MPI_THREAD_SERIALIZED

Similar as the funneled mode, multiple threads can be used to parallelize the computation in the serialized mode. For the MPI communication phases, however, any single thread can issue MPI calls at a time. That is, different threads can concurrently perform the computation, but all of them need to be synchronized in order to serialize the MPI calls. In a typical OpenMP program, this can be implemented by making MPI calls within OpenMP critical regions or single regions as shown in Figure 2.3(c) and 2.3(d) respectively.

MPI_THREAD_MULTIPLE

The multiple mode is different from the above levels, multiple threads can concurrently perform both user computation and MPI communication. The MPI implementation is required to provide appropriate synchronization among threads (i.e., lock protection and memory barriers) to protect accesses to shared internal data structures.

2.2.2 Typical Applications

After a brief overview of the hybrid programming model, we then introduce two scientific applications that utilize this model.

2.2.2.1 Quantum Monte Carlo Simulation

Quantum Monte Carlo (QMC) method is one of the most accurate solution to provide accurate and reliable approximation for quantum many-body systems. It helps scientists study the complex electronic structure of realistic world on large-scale computing systems. The algorithm of QMC method is mainly designed around two data objects: enormous "walkers" to represent the dynamic status of each particle, and a large but read-only `ensemble` data that shared among all walkers. The traditional implementation of QMC method utilizes MPI to distribute the walkers among multiple processes and simply replicate the ensemble data on each MPI process. However, such design extremely limits researchers to study larger physical systems or achieve more accurate simulations since the ensemble data is so large that always takes Gigabytes memory per core. Especially on modern mulit- and many-core systems, whose memory capacity per core is actually reducing, a more efficient design is required.

QMCPACK is an open-source QMC package implemented using hybrid MPI+ threads programming model for massively parallel computing system [41]. It utilizes threads to parallelize the walkers inside every physical node thus the essential memory restriction can be addressed since the large `ensemble` data can be shared among threads on every node, and employs MPI for inter-node communication as demonstrated in Figure 2.4. This design also benefits from reduced collective communication among MPI processes that is used for global reduction calculation among walkers, and from less number of large point-to-point communication between paired MPI processes for exchanging walker objects in the load balance step.

2.2.2.2 Computational Fluid Dynamics

Nek5000 is an open-source code that widely used in a broad range of applications such as nuclear reactor cores, ocean modeling and combustion simulation [27]. It provides high order, incompressible Navier-Stokes solver based on the spectral element method. The implementation of Nek5000 is mainly composed of conjugate gradient (CG) solver with efficient preconditioners, which is captured in the Nekbone mini-application with the basic structure and user interface.

The main computational kernel of Nekbone consists of multi-grid matrix-matrix multiplications. Several researches have looked into the optimization for such computation pattern on advanced heterogeneous HPC architectures. For

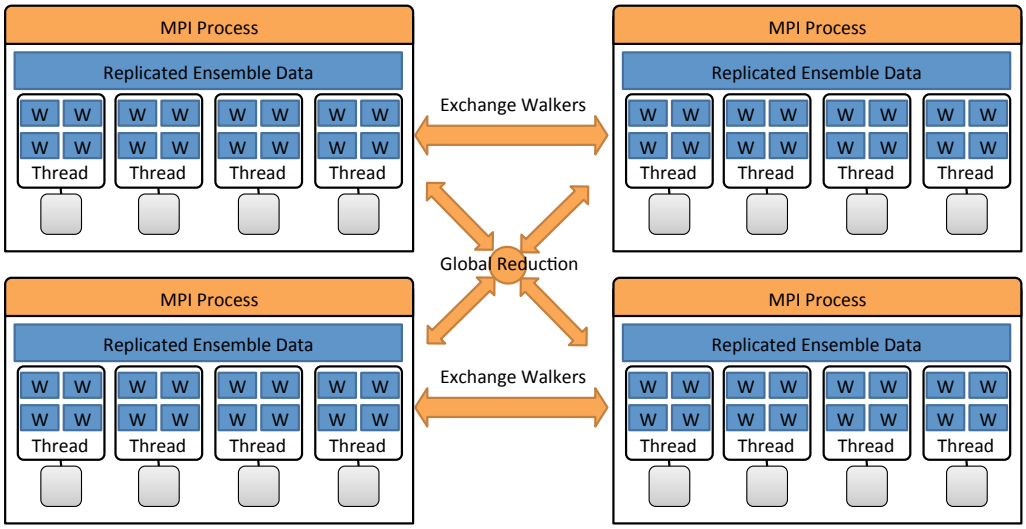
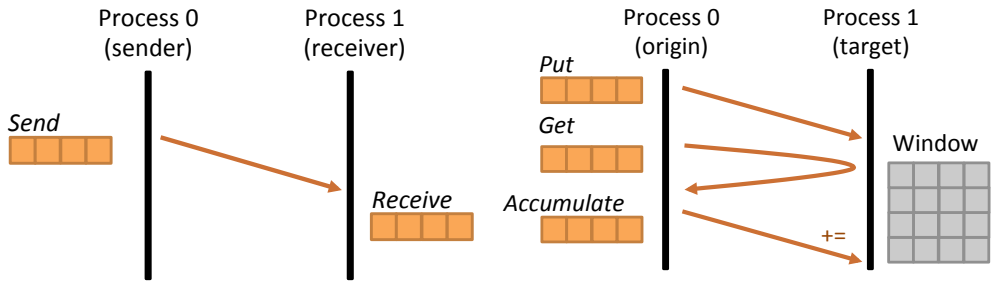


Figure 2.4: Hybrid Implementation of Quantum Monte Carlo Simulation.



(a) Two-Sided Mode. (b) One-Sided Mode.

Figure 2.5: MPI Communication Modes

example, Markidis et al. [46] presented an MPI+OpenACC version of Nek5000 code to highly parallelize the most time-consuming ax3D and gs_op subroutines on GPU-accelerated systems. Hart and Ivanov et al.'s papers [38, 32] have also contributed the MPI+OpenMP version of the Nekbone mini-application for accelerating the local computation on Cray supercomputers.

2.3 MPI One-sided Communication

MPI-2 and MPI-3 introduced the MPI one-sided communication model (also known as remote memory access or RMA). Unlike the well-known two-sided communication (e.g., MPI_Send/MPI_Receive), the one-sided mode allows applications to define more dynamic and data-driven communication patterns where a process can directly access memory in another process (i.e., window) through RMA operations such as `put`, `get` or `update`. Furthermore, all the operations are only issued from the origin process, thus the program running on the remote process does not need to call any MPI routines to match the operations. Figure 2.5 demonstrates the difference between the two-sided mode and the one-sided mode.

2.3.1 Programming and Semantics

Because the second contribution of this thesis is a process-based asynchronous progress model that comprehensively supports the strict MPI one-sided communication semantics which is also the most challenging part in this work, we then introduce the primary semantics of this communication mode in the rest of this section. The semantics of the RMA communication can be divided into following three primary steps: window creation, RMA synchronization and issuing RMA operations.

2.3.1.1 Window Creation

A memory area on a process that is exposed to all other processes in a specified group—allowing direct access to these processes—is called a `window`. MPI-3 provides the following four window initialization functions:

MPI_Win_create
: This routine exposes an RMA window for the memory region which is allocated by user application in advance. The corresponding `MPI_Win_free` call only releases the RMA window, thus user is responsible for releasing the memory region after window is freed.

MPI_Win_allocate
: This routine allows MPI to internally allocate a memory region and expose it to the other processes as a remotely accessible window. The corresponding `MPI_Win_free` call releases both the window structure and the memory buffer.

MPI_Win_allocate_shared
: This routine allows MPI to initialize a shared window among processes located in the same shared memory system (e.g., the same NUMA node) through external system support such as mmap or XPMEM on Cray systems [78]. This shared memory region can be accessed by CPU load/store instructions instead of MPI RMA operations, however, additional synchronization is required to ensure the correctness with other concurrent RMA operations. The start address of a remote window region mapped on the local process can be got from the `MPI_Win_shared_query` call.

MPI_Win_create_dynamic
: This routine allows programs to expose an empty remote accessible window, and then attach/detach one or multiple memory regions in later execution.

The above routines give user different levels of flexibility of window creation. However, the routines with more flexibility also limit the possible internal optimization can be provided from MPI implementations. For instance, the most flexible `MPI_Win_create_dynamic` can rarely get any optimization.

2.3.1.2 RMA Operations

After the remote accessible window is identified, a process can issue `put`, `get`, or `accumulate` operations to access this window. Figure 2.5(b) gives an image to

demonstrate data movement associated to those operations.

MPI_Put
: This operation copies the data in the origin process's buffer to the specified memory location in the window on the target process.

MPI_Get
: This operation copies data from the specified memory location of remote window to the buffer located in the origin process's local memory.

MPI_Accumulate
: This operation first transfers data from the origin process's buffer to the target process, and then performs a update on the target side following the user specified operation (e.g., `MPI_SUM`) and stores the result into the window. We note that, unlike the put/get operations, the accumulate operation is guaranteed to be `ordered` and `atomic` per basic data element.

Beside above three basic RMA operations, there are three other operations also defined in MPI standard: `MPI_Get_accumulate`, `MPI_Fetch_and_op` and `MPI_Compare_and_swap`. The detailed semantics of those operation can be found in [5].

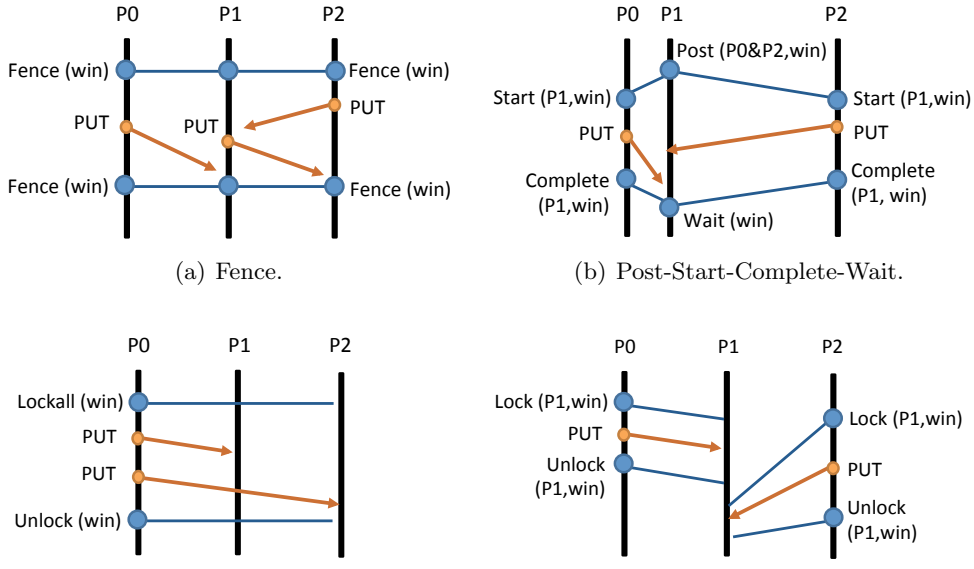
2.3.1.3 RMA Synchronization

All the RMA operations are non-blocking MPI calls, which means the completion of those data movement is not guaranteed at return. In addition, since processes may concurrently access the same RMA window, we also need synchronization among the involved processes in order to avoid any conflicts. MPI defines two kinds of synchronization modes to handle those responsibilities in RMA communication, they are the `active mode` and the `passive mode`. We introduce each of them separately in this section.

Active Mode: This mode provides a similar synchronization as that in two-sided mode, both the origin process and the target process need to explicitly call the synchronization. Two sets of synchronization calls are defined in MPI: `fence` and `post-start-complete-wait`.

- In `fence` synchronization, all the processes in the window must collectively call `MPI_Win_Fence` routine to synchronize with each other (Figure 2.6(a)) similar as `barrier` in the two-sided communication mode. The return from the fence call guarantees: (1) all the processes have arrived at the fence call; (2) all the outstanding RMA operations and local load/store instructions issued on this window have been completed.

- The `post-start-complete-wait` synchronization can be considered as a subset of fence (Figure 2.6(b)). At the beginning of the RMA communication, the target process (P1) calls `MPI_Win_post` to expose its window to one or several processes and the origin process (P0 or P2) performs `MPI_Win_start` to match the post call and then starts the remote access; at the end of the communication, the origin process needs to call



(a) Fence.

(b) Post-Start-Complete-Wait.

(c) Lock_all-Unlock_all.

(d) Lock-Unlock.

Figure 2.6: RMA Synchronization Modes

`MPI_Win_complete` to complete its operations and the target process needs to call `MPI_Win_wait` to ensure all the operations issued on its window have been finished.

Passive Mode: Apart from the semi-dynamic active mode, MPI also offers the passive mode which performs completely dynamic pattern. That is, only the process issuing operations (origin process) is required to explicitly call the synchronization. Two sets of synchronization calls are defined: `lock_all-unlock_all` and `lock-unlock`.

- The `lock_all` serial provides global synchronization similar as the `fence`, however, only the origin process (e.g., P0 in Figure 2.6(c)) issues the `MPI_Win_lock_all` and `MPI_Win_unlock_all` calls. The return from `lock_all` ensures the origin process have acquired the `shared` lock on all the other processes, and the return from `unlock_all` ensures: (1) the locks have been released, and (2) all the operations issued from this process have been completed remotely.

- The `lock` serial can be also considered as a subset of `lock_all` which provides per-target exclusive/shared lock (`MPI_LOCK_EXCLUSIVE` or `MPI_LOCK_SHARED` lock type). We note that two origin processes can concurrently acquire a `shared` lock on the same target window, however, any other lock requests must be serialized with the `exclusive` lock. Figure 2.6(d) shows an example. Simultaneous `lock_all` and `lock` follow the same rule.

- Besides the lock calls, the passive mode also offers two `flush` synchronization routines that only complete the outstanding RMA operations, and a `sync` routine (`MPI_Win_sync`) that synchronizes the data of its local window

updated by remote RMA operations and the one updated by load/store instructions. `MPI_Win_flush` completes the operations issued from the origin process to a single target process, and `MPI_Win_flush_all` completes operations issued from the origin process to any target processes in the window.

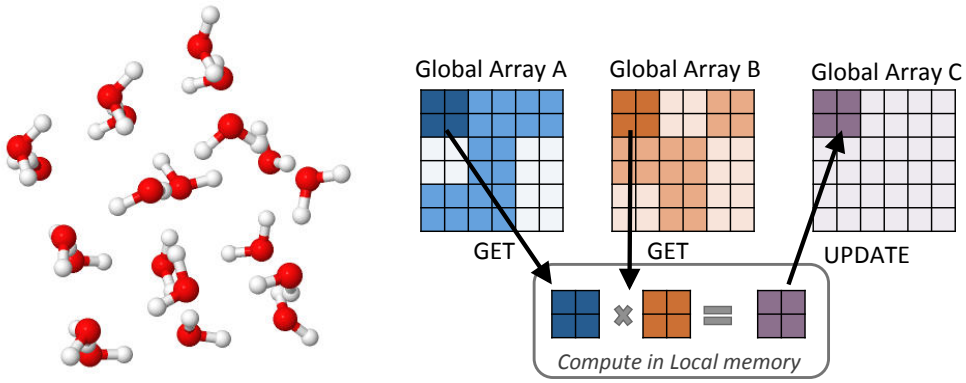
2.3.2 Irregular Applications

In this section, we introduce three scientific applications in chemistry, bioinformatics and nuclear physics fields, all of them involve extremely dynamic and irregular computation and data movement, which can benefit from the one-side communication model.

2.3.2.1 NWChem Quantum Chemistry Application

NWChem is a widely used quantum chemistry application suite that provides a large set of simulation capabilities [76]. Due to the large memory needs in NWChem that often require memory sharing across multiple nodes, it is developed based on the Global Arrays toolkit [51] which provides users with distributed dense arrays that can be accessed through one-sided operations. Figure 2.7(b) demonstrates the typical `get-compute-update` pattern used in NWChem.

Current NWChem has been looking at small-to-medium molecules (e.g., $(H_2O)_{21}$ as shown in Figure 2.7(a)) consisting of 20-100 atoms. Since the coulomb interaction among such small amount of atoms is reasonably large, the computation and communication can be successful scaled on modern supercomputers. However, scientists aim to study more complex molecules that are composed of thousands atoms or even larger thus not only the short-range interactions but also the long-range interactions have to be covered. This means, the diversity of the amount of computation per process can be considerably increased, thus resulting in extremely irregular computation with data movement.

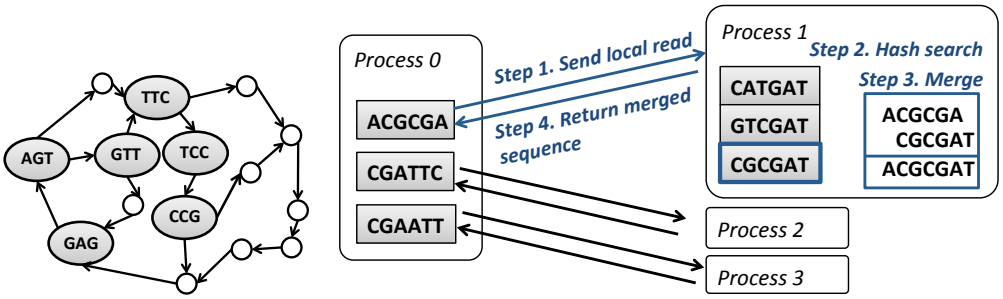


(a) Interaction in $(H_2O)_{21}$ molecule. (b) Get-Compute-Update pattern.

Figure 2.7: Communication in NWChem Application.

2.3.2.2 SWAP-Assembly Bionformatics Application

In bioinformatics, since it is still hard to read the whole genomes in modern DNA sequencing technology, researchers often break down long DNA samples



(a) Graph Reduction. (b) Communication Pattern.

Figure 2.8: Irregular Communication in SWAP-Assembly.

into large amount of small fragments, called "reads", and then read those reads into digital data as the first step. The next step is called assembly, which merges overlapping reads back to one or several contiguous DNA sequences. The sequence assembly technology helps biology scientists analyze DNA sequence, and is especially important for understanding complex environments containing many different microbiomes (e.g., soil and seawater).

The SWAP-Assembler software provides highly scalable assembler that processes the sequence assembly on thousands of cores in parallel [47]. The initial reads are represented as a distributed De Bruijn graph (e.g., Figure 2.8(a)), and final contiguous DNA chains are assembled by executing multiple rounds of graph reduction and error removal over MPI communication. The communication pattern follows the `send-merge-return` mode as shown in Figure 2.8(b). Every process issues each of its local DNA read to a remote process to find the overlapping reads. On the remote process, it first searches the overlapping read for every received message, then merges the reads and finally returns the merged result. If there is no matching read on the remote side, the sending process will try another remote process following the same patter. This processing always involves enormous irregular data movement over Petabytes of data and requires several days or even months of computation. For instance, the largest simulation done to date was at the University of Chicago, where a 2.3-Terabyte sample was assembled on a supercomputer with 18,000 coresthis simulation took 4 days to complete and spent 99.9% of its time idling, because of imbalance between processing units.

2.3.2.3 Greens Function Monte Carlo

Greens Function Monte Carlo (GFMC) is an application in theoretical nuclear physics that provides ab initio calculations for few-nucleon systems [17]. It describes the nuclear structures and reactions by solving the Schrödinger equation and is recognized as the most reliable method for nuclei with 12 or fewer nucleons. The implementation of GFMC utilizes OpenMP to parallelize heavy sparse matrix-vector multiplications and uses MPI to communicate among distributed computing nodes.

The Asynchronous Dynamic Load Balancing (ADLB) library [45] is essentially designed for addressing the load balancing among MPI processes in GFMC on large scale systems that contain more than one hundred thousand computing cores. It provides a general-purpose worker-server model with one-sided Put/Get

operations that helps application codes dynamical share work tasks with assorted work types and priorities. A few server processes are initialized to maintain a distributed shared work queue. Application processes can then submit arbitrary work tasks to the queue with necessary data, and retrieve the results after any task is finished.

Chapter 3

Multithreaded MPI

Publication

This chapter includes the contents that have been published in conference paper [59]. Full article can be found at `http://dl.acm.org/citation.cfm?doid=2597652.2597658`.

The hybrid MPI+Threads programming model has become one of the most popular programming model on many-core systems. The common mode of this hybrid model often uses multiple threads to parallelize the computation on every computing node, and utilizes one of the threads to transfer data across nodes by using MPI. This mode is defined as MPI `FUNNELED` or `SERIALIZED` thread-safety mode as introduced in Section 2.2.1. The most prominent of the threading models used in scientific computing today is OpenMP [21]. In the MPI+OpenMP programming, the application developer can simply add parallel annotations (i.e., `pragma`) on the computation that need to be parallelized by the compiler and the thread runtime system. The compiler, in turn, translates these annotations into semantic information that the runtime system can use to divide and schedule the computing tasks on multiple threads. The MPI communication does not need any code modification, developer can just put it outside the OpenMP parallel region or protects it by using master, critical or singe sections as demonstrated in Figure 2.3.

This mode allows applications to benefit from massive parallelism without large modification in code, it also helps applications scale to larger problem since it allows memory to be shared among large amount of cores on every node. However, the MPI communication in this mode still faces several critical challenges that degrade the performance. This chapter focuses on these challenges and propose efficient solution. In Section 3.1 we first describe the communication issues in the hybrid model. Then in Section 3.2 we present the concept of our solution—an internally multithreaded MPI—, and then list the practical challenges we have to address in implementation. In Section 3.3 we introduce the detailed design and implementation in both OpenMP and MPI libraries, and Section 3.4 provides evaluation results by using several micro and macro-benchmarks.

3.1 Problem Statement

In the common MPI+OpenMP mode, hundreds threads are created in the parallel region for user computation, however, only single thread is used to issue the MPI communication. Such a mode also means that most OpenMP threads are idle during MPI calls, resulting in wasted computational resources. Moreover, since the clock rate of single core on the many-core architecture is always much lower than traditional CPUs, such single thread execution can also result in severe performance degradation in communication.

3.2 Solution

Parallelism is the essential key to reach high performance on many-core systems, MPI communication is no exception. We present MT-MPI, an internally multithreaded MPI implementation that transparently coordinates with the threading runtime system to share idle threads with the application. In this dissertation, we designed MT-MPI in the context of OpenMP, which serves as the most widely used threading runtime system for the applications. MT-MPI transparently employs application idle threads to accelerate MPI communication and data-processing, also achieving better resource utilization. We use the "native mode" of Intel KNC as the architectural testbed where applications are executed directly on the coprocessor. This approach should also be suitable for the next generation of Xeon Phi product (KNL) since it will be built as self-hosting chips in upcoming supercomputers [11, 18].

To demonstrate the performance benefits of the proposed approach, we modified the Intel OpenMP runtime library [34] and the MPICH implementation of MPI [10]. Specifically, we modified the MPI implementation to parallelize its internal processing using OpenMP parallel regions. Figure 3.2 shows the pseudo code of an example following this approach, where MPI routine is called outside the user parallel regions as the MPI `FUNNELED` mode. We also studied new algorithms for various internal processing steps within MPI that are more "parallelism friendly" for OpenMP to use.

```
#pragma omp parallel
{ /* user computation */ }

MPI_Function()
{
  #pragma omp parallel
  { /* internal processing */ }
}

#pragma omp parallel
{ /* user computation */ }
```

Figure 3.1: Pseudo Code of MPI+OpenMP in MT-MPI

In theory, this model would allow both the application and the MPI implementation to expose their parallelism requirements to the OpenMP runtime, which in turn can schedule them on the available computational resources. In practice, however, several challenges exist:

- **Parallel algorithms with insufficient threads.**
  We modify the algorithms used in MPI internal processing for better parallelism (e.g., remove data dependency in for loop). While the modified algorithms are efficient for OpenMP parallelism, they may not as efficient if the number of available OpenMP threads is not sufficient. Consequently, the parallel version can improve performance only when sufficient OpenMP threads is available, we need appropriate trade off according to the number of available threads However, the actual number of available threads at runtime is unknown. Depending on the application's code structure, this can vary from zero to all threads being available for MPI processing. Thus, if not designed carefully, the algorithms can perform even worse than the traditional sequential implementation of MPI.

- **Core oversubscription risk in nested parallel region.**
  The current implementation of the Intel OpenMP runtime does not schedule work units from nested OpenMP parallel regions efficiently. It simply creates new pthreads for each nested parallel region and offload the threads scheduling to the operating system. This can results in core oversubscription since more threads can be created than the available cores, and consequently degrading performance.

To work around these challenges, we modified the Intel OpenMP runtime to understand the status change of threads at runtime and expose the information about the idle threads to the MPI implementation. The MPI implementation then can use this information to choose appropriate algorithms that trade off between parallelism and sequential execution in order to achieve optimal performance, Such information also allows MPI to schedule its internal parallelization only when enough idle threads are available.

3.3 Design and Implementation

In this section we describe the design of MT-MPI, including modifications to the OpenMP runtime system and the MPI implementation. We use MPICH library (v3.0.4) and the Intel OpenMP runtime (version 20130412) as the base code of our implementation.

3.3.1 OpenMP Runtime

understand the status change of threads at runtime and expose the information about the idle threads to the MPI implementation

As described in Section 3.2, we need modify the OpenMP runtime system to expose the number of idle threads to the MPI implementation in order to address the challenges for MPI internal parallelism. To understand how many threads

are idle at current time, the idea is to track how many threads are being used by the application vs. how many threads are idle (e.g., because they are waiting in an OpenMP barrier). Then, the OpenMP runtime can expose this information through a new runtime function. Then the MPI implementation could query for the number of idle threads by calling this runtime function, and use this information to (1) choose the most efficient internal parallelization algorithms and (2) use only as many threads in the nested OpenMP region as there are idle cores, by explicitly guiding the number of threads in OpenMP (using the `num_threads` clause in OpenMP).

We note that the second challenge described above (additional pthreads created in nested OpenMP regions) is an issue only with the current implementation of the Intel OpenMP runtime. An alternative OpenMP runtime implementation (e.g., internally uses user-level threads [52]) may not have this problem. However, since most OpenMP implementations today use pthreads internally, we consider this to be a real issue that needs to be addressed in this research.

3.3.1.1 Threads Idleness

Since we need to track the status change for every internal threads in the OpenMP runtime, we need to first understand the status of threads in the following cases.

- **MPI call made outside the OpenMP parallel regions.**
  As shown in Figure 2.3(a), all threads except the main thread are idle (often equal to `OMP_NUM_THREADS`). Thus, we expect MPI to be able to utilize `OMP_NUM_THREADS` threads in this case.

- **MPI call made in an OpenMP single region.**
  Figure 2.3(d) shows an example for this case. We know that OpenMP single regions provide an `implicit barrier` on exit. Thus, we can expect that all threads waiting in the barrier can be idle if the current thread in the single section queries the information, since all the other threads have to synchronize with the current thread at the `implicit barrier`. In practice, however, it should be careful that not all threads might have arrived the barrier, for example, some threads might still be working in the previous user computation. Consequently, the number of idle threads in this case can vary between zero and the maximum number of threads. We modified the OpenMP runtime to track the status for each thread in order to get the actual number of idle threads when it is queried. In this case, the amount of parallelism available to MPI is not a fixed number. However, for most OpenMP parallel regions where the workload distribution is mostly balanced among threads, we expect the number of idle threads is close to the maximum number of threads (typically equal to `OMP_NUM_THREADS`).

- **MPI call made in an OpenMP master region or single region with a *nowait* clause.**
  Figure 2.3(b) shows an example for this case. It is similar to the previous single region, the only difference is that there is no implied barrier at the end of such regions. In other words, there is no natural synchronization

for the threads during these regions. Nevertheless, depending on how the application is written, it is possible that to user can define an external synchronization point (e.g., OpenMP barrier) that would cause idle threads to be available. Consequently, we follows the strategy we used for the previous case, tracking the number of idle threads. However, we do not expect too many idle threads in this case in practice.

- **MPI call made in an OpenMP critical region.**
  Figure 2.3(c) shows an example of this case. We know that OpenMP critical section involves synchronization among threads in order to ensure only one thread can enter the critical region at a time. While this is not quite an implicit barrier at the entry of critical section, its impact on the availability of threads can be considered as similar to that of the OpenMP single region. To be specific, when the first thread enters the OpenMP critical region, all the remaining threads can be expected to be idle once they have arrived at the entry of critical region; when the second thread enters the critical region, the first thread is no longer available for our use because it has already finished its execution in the critical region and left this section, thus we do not know whether it could be idle or not; when the last thread enters the critical region, none of the remaining threads are expected to be idle because all of them have already left the critical section. Following the same strategy as in the previous cases, we track the number of idle threads inside the OpenMP runtime. We expect that the number of idle threads would be close to maximal number of threads for the first few threads entering the critical section and gradually becomes lower and eventually zero for the last few threads.

As we have discussed in each case, it can be risky to utilize the idle threads in some situations because the status of those threads can change at any time. For example, as shown in Figure 3.2, if we consider the current querying thread is active in the single nowait section (thread 4), threads 2 and 3 can be considered as available threads since they are waiting at the entry of the next critical section, however, their status can change once the thread 1 finished its execution in the critical section. Such status change is unrelated to the current single section, thus it is unknown when those threads become active, and consequently degrading performance if we use them for our parallelism in the single section (suppose the MPI function is called in the single section). Therefore, we distinguish two kinds of thread idleness as follows:

- **Guaranteed idle threads.**
  Any threads that are guaranteed idle until the current thread exist from MPI call. Specifically speaking, if a threads is at one of the following status, we consider it is guaranteed idle: (1) waiting at the barrier for other threads in the team to arrive (i.e., explicit OpenMP barrier or the implicit barrier at the end of single section); (2) waiting to enter a critical section.

- **Temporarily idle threads.**
  Any threads who are currently idle but their status change is not controlled

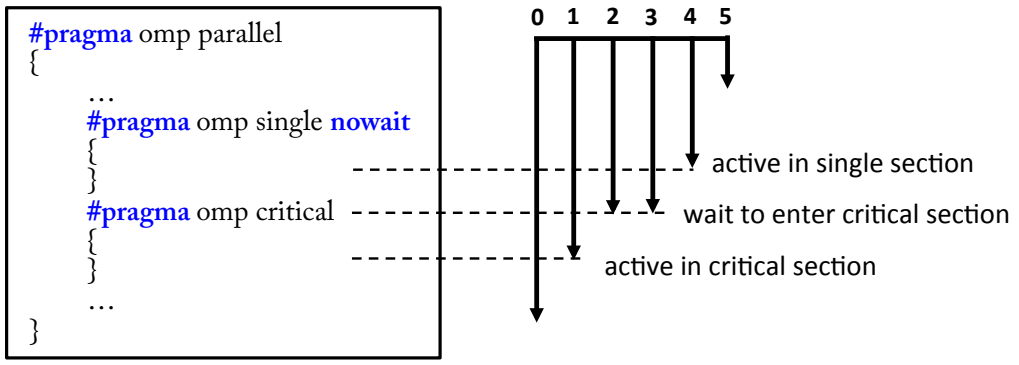


Figure 3.2: An Example of Temporary Idle Threads.

```
#pragma omp parallel
{
  #pragma omp single
  {
    num_idle_threads = omp_get_num_idle_threads();
    if(num_idle_threads < N ){
      /* sequential algorithm */
    }else{
      #pragma omp parallel num_threads(num_idle_threads)
      { /* parallel algorithm */ }
    }
  }
}
```

Figure 3.3: Thread Scheduling in OpenMP barrier Routine.

by the current thread in the MPI call. For example, for the threads waiting in an critical section that is unrelated to the current thread (e.g., thread 2 and 3 in Figure 3.2), we call them temporarily idle threads.

To avoid the risky of performance degradation, we only expose the number of `guaranteed idle threads` to MPI implementation. We define an OpenMP runtime extension `omp_get_num_idle_threads` to return the number of guaranteed idle threads at the querying time. Thus the user thread in an MPI call can easily query such information during through this routine, and safely use the returned value to do trade off between algorithms and also provide guidance for the internal nested parallel region. Figure 3.3 shows an example. We note that our implementation treats a thread as idle only when it is not engaged in any OpenMP activity, including both OpenMP parallel loops and OpenMP tasks. We also note that in our implementation the performance overhead associated with tracking whether a thread is actively being used by the OpenMP runtime is too small to be observed and thus we do not discuss it in this paper.

```
while(time < KMP_BLOCKTIME)
{
  if(done)
    break;
  /* spin loop */
}

sched_yield();
pthread_cond_wait(...);
```

(a) Wait Progress in OpenMP barrier.

```
#pragma omp parallel
{
  #pragma omp single
  {
    set_fast_yield(1);
    #pragma omp parallel
    { ... }
  }
}
```

(b) Fast Yield.

Figure 3.4: Thread Scheduling in OpenMP barrier Routine.

3.3.1.2 Thread Scheduling in Nested Parallelism

As we have discussed about the thread idleness, the threads waiting in an explicit/ implicit barrier should be idle and are available for reuse in MPI. However, this is not exactly as we expected in the OpenMP implementation because of special optimization strategies. We have noticed that, the Intel OpenMP runtime does not put the threads directly into the passive wait status (i.e., being yield or sleep) in the internal wait progress for barriers. Instead, the threads are actively waiting in a spin loop for other threads to arrive for a configurable amount of time `KMP_BLOCKTIME`. Figure 3.4(a) demonstrates this implementation. This is because, for most well-balanced OpenMP parallel loops, thread synchronizations such as barriers are often short-lived since threads tend to arrive at the barrier at approximately the same time. Thus, when a thread arrives at a barrier, if it is immediately go into sleep status while waiting for the other threads to arrive, and is woken up in a short amount of time, performance can be degraded because of the overhead of waking up threads from a sleep state.

Unfortunately, such optimization strategy might not be suitable within the environment of MT-MPI. Especially when a large value is set for `KMP_BLOCKTIME`, it would mean that threads do not become "truly idle" for a long time. While this is not a concern for regular OpenMP parallel loops, it can break our theory of thread idleness in MT-MPI, and thus resulting in degrade performance for nested OpenMP parallel regions since more threads would be active than our estimation and consequently cause core oversubscription for the `KMP_BLOCKTIME` amount of time.

When MPI calls are outside the application OpenMP parallel region (such as in Figure 2.3(d)), this is not a concern since MPI would use the same threads as the application in its parallel region. When MPI calls are inside the application parallel region, however, this would require MPI to use a nested OpenMP parallel region. And since the threads that arrived at the barrier would not yield the available cores immediately, this would either require MPI to utilize lesser parallelism by only using the idle cores or cause thread thrashing on the available cores for `KMP_BLOCKTIME` amount of time. Neither solution is ideal.

In MT-MPI, to be able to employ these resources as soon as possible, we

investigated two possible solutions in the OpenMP runtime: `fast-sleep` and `fast-yield`.

- **Fast-Sleep.**
  We expose a new function `set_fast_sleep` to the MPI implementation, thus MPI could notify OpenMP runtime to force all the threads in the current team to skip the active wait process during the barrier routine and immediately go into sleep status (e.g., call `pthread_cond_wait` function). These sleeping threads can be automatically awaken by receiving a signal through the synchronization from current active thread similar as the original implementation.

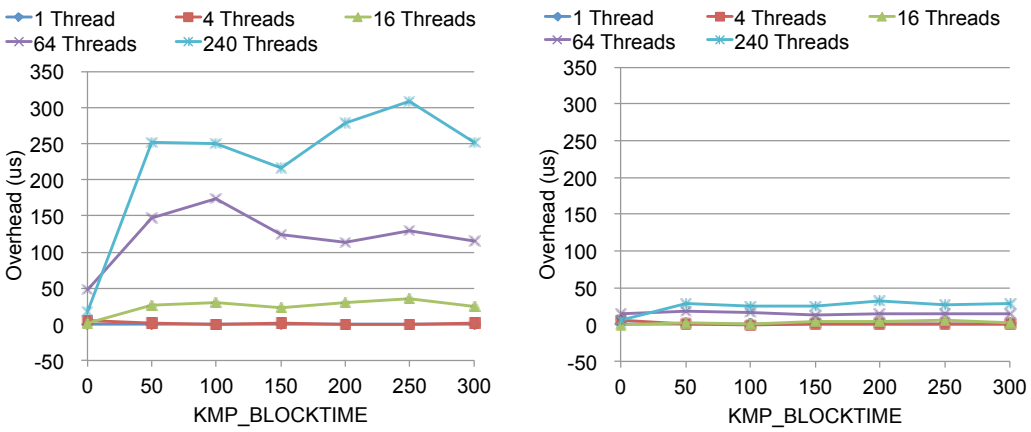
- **Fast-Yield.**
  The second approach is to focus the waiting threads immediately yield the core (e.g., through `sched_yield` system call) instead of spin loop, and then go into sleep after waiting `KMP_BLOCKTIME` amount of time. Similarly, we expose a new function `set_fast_yield` to allow MPI to enable this optimization. Figure 3.4(b) shows an example of its usage.

We note that both the `fast-sleep` and the `fast-yield` approaches follow three rules: (1) they change the thread scheduling in the wait progress only for those threads who are guaranteed to be idle (e.g., threads waiting in an OpenMP barrier); (2) the `set_fast_sleep` and `set_fast_yield` setting is performed internally inside the MPI call and such change is automatically reset once the internal parallelism in MPI is complete, so future OpenMP explicit/implicit barriers are not affected by it; and (3) the proposed thread scheduling optimization is enabled only when MPI uses nested OpenMP parallelism (e.g., for single or critical section) and is not used in the case that the MPI function is called outside the OpenMP parallel region.

Both approaches allow us to eliminate the core oversubscription risk in the waiting progress, however, the overhead of such approaches show difference. Figure 3.5(a) and 3.5(b) compare the overhead of each approach in a single section similar as the code shown in Figure 3.4(b). Obviously, `fast-yield` allows us to manage the cores with a consistent low overhead that only takes 30 μs even with 240 threads, while the `fast-sleep` approach takes much more overhead with increasing number of threads, resulting in more than 200 μs cost at 240 threads. Therefore, we utilize the `fast-yield` approach in MT-MPI.

3.3.2 MPI Internal Parallelism

After we expose the information about the idle threads through our extended OpenMP runtime system, the MPI implementation can then efficiently schedule its internal parallelism to achieve performance improvements. In this section, we look into the MPI internal processing, and demonstrate the benefit of parallelism in various aspects of the MPI processing. We note that we only utilize the idle threads that are guaranteed to be available (`guaranteed idle threads`) in our implementation. When all threads are available (e.g., when the MPI routine is called outside the OpenMP parallel region), we do not explicitly specify the



(a) Fast-Sleep.

(b) Fast-Yield.

Figure 3.5: Overhead of thread scheduling optimization in OpenMP Wait Progress.

number of threads to be utilized by our internal OpenMP parallel regions but simply let OpenMP manage it; if fewer than the maximum number of threads is idle (e.g., MPI is called in single region), we specify the amount of threads used in the internal parallel regions by using the `num_threads` OpenMP clause.

In the rest of this section, we describe the parallelism we achieved in three aspects of MPI internal processing as the showcase. They are derived datatype communication, shared-memory communication, and network I/O operations.

3.3.2.1 Derived Datatype Processing

MPI defines several kinds of derived datatype, such as `vector`, `indexed` and `struct`, to help applications describe noncontiguous regions of memory. Derived datatypes are used to describe complex noncontiguous data layouts and can be directly used in the packing/unpacking processing (i.e., `MPI_Pack`, `MPI_Unpack functions`) to `pack` data from noncontiguous memory locations to contiguous buffer, or `unpack` data from a contiguous buffer to noncontiguous memory locations. Usually the packing/unpacking processing is not directly used in user code, but embedded in MPI data transfer. For example, the well-known halo exchange [77] algorithm can be implemented by using derived datatypes with the MPI send/receive communication. The internal processing of such communication can be divided into three steps: (1) packing noncontiguous user data elements into a internal buffer on the sender side; (2) transfer the packed contiguous data to receive side; (3) unpack the received contiguous data to the noncontiguous memory locations on the receiver side. Figure 3.6 demonstrates such processing for transferring the right edge of a two-dimension matrix which can be defined as the `vector` datatype.

The pack and unpack internal processing consists of a set of local memory copies. A typical implementation often traverses the derived datatype tree and copies each noncontiguous data chunk separately. A well-know optimization that has been utilized in some MPI implementations is representing the datatype tree as a stack structure so that it can be iteratively traversed rather than using

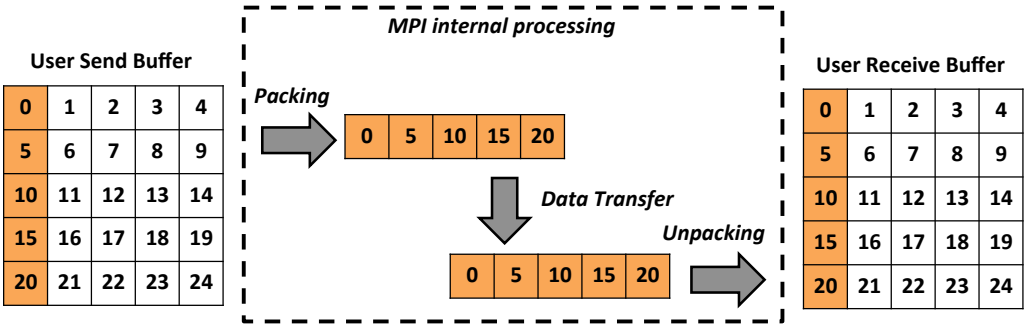


Figure 3.6: Internal Processing for Data Transfer with Derived Datatype.

a recursive traversal [55]. Since each noncontiguous data chunk is copied to a different location and no dependencies exist among the different data elements, such copy processing is a good candidate for OpenMP parallelization. Moreover, thanks to the relatively large private L1/L2 caches per core on the Xeon Phi chips, we expect efficient performance can be obtained in our approach, where different threads concurrently access to separate memory locations. Therefore, we modified the MPI implementation to parallelize internal loop of data copy in the datatype processing using OpenMP. We note that for nested datatypes (e.g., a vector of vectors) only the lowest level is parallelized in our implementation.

```
for(i=0; i<count; i++){
  *dest++ = *src;
  src += stride;
}
```

(a) Sequential implementation.

```
#pragma omp parallel for
for(i=0; i<count; i++){
  dest[i] = src[i*stride];
}
```

(b) Parallel implementation.

Figure 3.7: Sequential and parallel data packing.

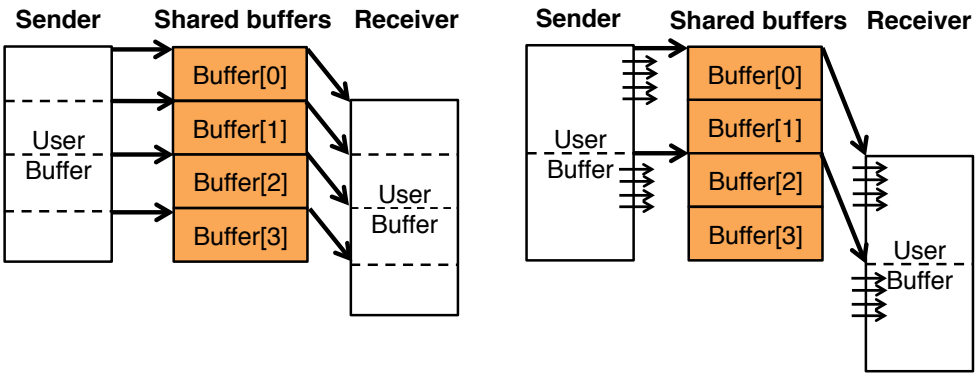
The concept of this optimization is straightforward, however, we observed unexpected performance degradation in this parallelism because of an unintended consequence of the compiler vectorization. Figure 3.7(a) shows the original datatype copy code used in MPICH. While this code works correctly in the sequential copy, it cannot be parallelized by using OpenMP because of data dependency on variable `dest` and `src` that makes compiler cannot understand the constant stride of accesses used through all iterations. Thus we modified the code as shown in Figure 3.7(b). While this new implementation eliminates data dependency and thus the compiler can understand the computation and parallelize it, the implementation also enables the `vectorization` provided by compiler. This automatic optimization itself is not a concern. However, we have observed more than performance degradation in the vectorized loops when the `stride` value is large and the amount of data copied in each loop is small. Specifically, the compiler also automatically enables prefetching in the vectorized version, however, it also causes additional cache misses in such strided loops. Consequently, our modification might perform worse than the sequential version when the benefit from parallelism is small (e.g., when very few threads are available). To work

around this issue, we choose the parallel approach only when sufficiently large amount of threads are available while still keep the vectorization since it is still beneficial in some cases (e.g., small stride or large copy data).

concern. However, the Intel compiler is inefficient in vectorizing strided loops with large stride values when the amount of data copied in each loop is small. Specifically, the compiler does incorrect prefetching in this case, causing additional cache misses and thus losing performance. Consequently, our modification to the code is not always beneficial and can perform worse than the sequential implementation when very few threads are available. To work around this issue, we could either disable vectorization in the parallel implementation or explicitly choose only the parallel approach when a sufficiently large number of threads are available. We chose the latter approach because vectorization is still beneficial in some cases (e.g., when the stride is small or the copy size is large). [add graph.]

3.3.2.2 Shared-Memory Communication

The second MPI internal processing we parallelized in MT-MPI is the data transfer in shared memory communication. When multiple MPI processes are located on the same node, the data transfer between these processes can be implemented based on special shared memory region among processes by utilizing some external techniques such as mmap. Since each process has a different virtual address space, most MPI implementations use a pipelined double-copy algorithm for intra-node communication [15]. This algorithm relies on a shared-memory ring buffer allocated between the sender and the receiver processes, and is implemented as the classical producer-consumer problem. As shown in Figure 3.8(a), the ring buffer is divided into multiple cells, then the sender process tries to get an empty cell and copies part of data from the send buffer into that cell, while the receiver process tries to find a full cell and then copies data from that cell to its receive buffer. The copies on the sender and the receiver processes are pipelined.

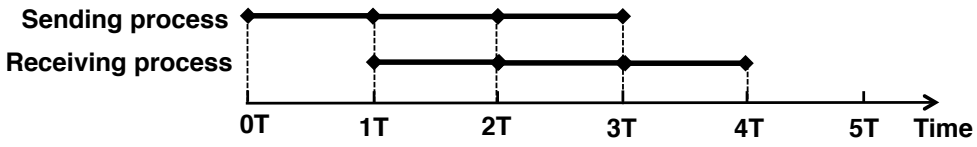


(a) Sequential pipelining.

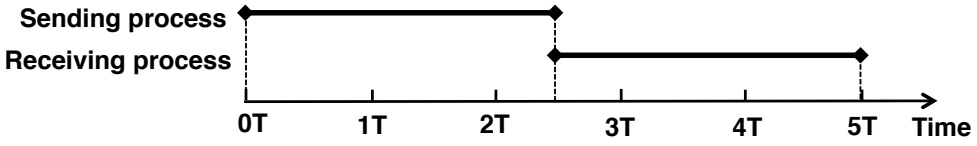
(b) Parallel pipelining.

Figure 3.8: Data movement of parallelization and pipelining.

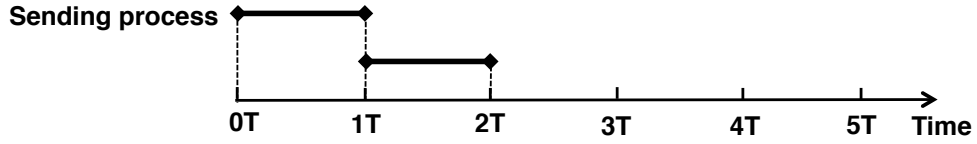
In MT-MPI, we parallelize this copies on both the sender and the receiver processing using the available idle threads. However, the original algorithm always copies data per cell size (32 KB in the MPICH implementation) which is too small for OpenMP parallelism compared to the overhead of threads synchroniza-



(a) Sequential pipelining.



(b) Poor parallelism.



(c) Strong parallelism.

Figure 3.9: Sequential pipelining vs. parallel data copy.

tion. Thus we extended the pipelined double-copy algorithm to ensure sufficient data to be parallelized in every single copy. As shown in Figure 3.8(b), we reserve multiple contiguous available cells as a "large cell", and concurrently copy data from the user buffer to that large cell on the sender side and from the cell to the user buffer on the receiver side. For messages that are larger than the size of the large cell, additional pipelining is still used similar to the sequential version. Consequently, every single copy can have enough work to be parallelized by multiple threads. Compared with the sequential pipelining algorithm, however, this modification may suffer from worse performance in the following cases.

Small messages. As the motivation for our algorithm change, there is not sufficient work can be parallelized by OpenMP in small message transfer. Moreover, the overhead caused by thread management and synchronization with OpenMP can be more expensive than the sequential copy. Thus, we do not expect any performance benefit from parallelization in this case.

Large messages but few available threads. In our parallel algorithm, we reserve as many available shared-memory cells as possible as a large cell and parallelize the copy on that large cell using all of the available idle threads. Although this approach can increase the work for parallelism, it also potentially increases the pipeline unit to a much larger size. That is, if we have enough threads to parallelize each pipeline unit we could obtain very low cost for each unit and thus achieve good performance; conversely, if we only have a few threads available, the cost of every pipeline unit obviously becomes more expensive than the small unit in the sequential version, thus resulting in performance degradation due to reduced pipelining. This trend is illustrated in Figure 3.9. Specifically, compared with the sequential pipelining (Figure 3.9(a)), when the number of threads available to MPI is small, the parallel copy does not improve performance much but delays the receiver process from getting started with its copy due to

increased cost for every single pipeline unit (Figure 3.9(b)). On the other hand, when sufficiently large number of threads are available to MPI, the parallel copy of each unit is much more efficient, thus balancing the loss of performance in pipelining (Figure 3.9(c)). We note that the issue can not be simply worked around by reducing the size of each shared-memory cell or the number of reserved cells in every single copy, because that would also reduce the amount of work distributed to each thread, thus causing similar issue as in the case of small messages.

Few shared-memory cells. Not only small messages but also insufficient shared-memory cells can result in too few work in parallelism. In other words, if the total size of the shared ring buffer is not large enough, it can happen that most cells are still being used for transferring previous message or previous part of the same message (e.g., being copied on the receiver side) and thus cause the work unit of the next copy (e.g., the next chunk copied on the sender side) to be so small that can not show benefit from parallelism compared to the overhead in threads management.

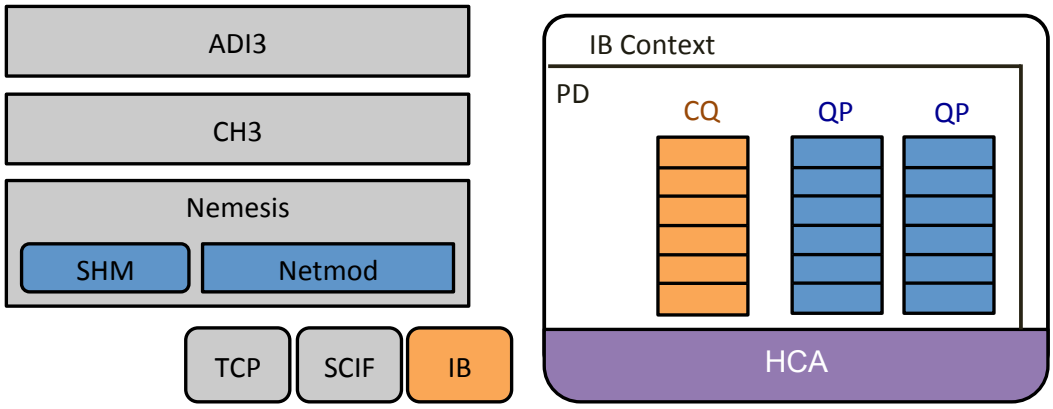
In summary, the parallel approach can improve performance only when (1) the message size is not too small ($\geq$ 64 Kbytes); (2) the number of threads is not too few ($\geq$ 8); and (3) the total size of free cells is not too small ($\geq$ 64 Kbytes). In our implementation, we choose the parallel algorithm for shared-memory communication only when all three conditions are met; otherwise we fall back to the original sequential pipelining. We note that the above thresholds mentioned for each condition are empirically evaluated on our test platform and must be tuned for different platforms.

3.3.2.3 Optimizations for the InfiniBand Network

Several MPI implementations are optimized for a variety of networks through a layered software architecture where one of the layers provides network-specific functionality [10, 71]. In MPICH, the code construction is defined as in Figure 3.10(a), the network-specific layer is called `netmod`. There are several netmod implementations existing for MPI over InfiniBand network (IB), with more-or-less similar functionality and performance. In this dissertation, we utilize the implementation described in [67].

To implement the IB communication, we need to create and manage a number of IB objects, including contexts, protection domains (PDs), queue pairs (QPs), and completion queues (CQs). As shown in Figure 3.10(b), a process can create one or more IB contexts, each of which contains one ore more PDs to define the protection semantics of memory and associated connections. Within a PD, the process can also create one or multiple QPs, each QP consists of a send queue and a receive queue, and is used to communicate between a pair of processes. A PD can also have one or more CQs, each CQ is used to check the completion of data transferring operations on one or more QPs. IB also provides shared queues for better memory management, but for simplicity we do not describe them here.

In the typical MPI IB netmod implementations, every MPI process often initializes one IB context and one PD shared for the connections on all the other processes. Multiple QPs are created on every process, each of which is corre-



(a) Code Construction in MPICH. (b) Components in IB Communication.

Figure 3.10: InfiniBand Netmod in MPICH implementation.

sponding to the connection to a single remote process. A single CQ is created on each process and shared by all the QPs.

3.3.2.3.1 Parallelism in IB Stack The IB software stack is thread-safe [53]. Specifically, when multiple threads access the same QP or CQ object, it internally uses mutexes to maintain state consistency between threads. Such state consistency is expensive and result in performance degradation. Therefore, to efficiently parallelize the data transfer through IB network, we always use different threads to handle the operations issued on different QPs in order to avoid any thread contention. Even with this approach, however, some shared data structures still need to be protected inside IB stack. Before we parallelize the MPI netmod, we first want to see how much performance improvement we can gain by parallelizing the operation posting processing. We first studied several parallel approaches in IB programs and compared the multithreaded point-to-point IB RDMA write bandwidth by using the `ib_write_bw` benchmark from the OpenFabrics Enterprise Distribution (OFED) package [53] with modification for OpenMP parallelism. Specifically, we compare the following three parallelism levels:

- **IB contexts.**
  Each process has 64 IB contexts, and each context has one QP and one CQ. Each thread handles operations on a different context, CQ and QP.
- **QPs and CQs.**
  Each process has a single IB context with 64 QPs and 64 CQs. Each CQ is dedicated to a different QP. Each thread handles operations on different QPs and CQs, but they all share the same context.
- **QPs only.**
  Each process has a single IB context with 64 QPs and one shared CQ. Each thread handles operations on different QPs, but they all share the same context and CQ.

Figure 3.11 shows the IB RDMA write bandwidth with small messages (64 Bytes) between two Intel Xeon phi coprocessors on different nodes with comparison be-

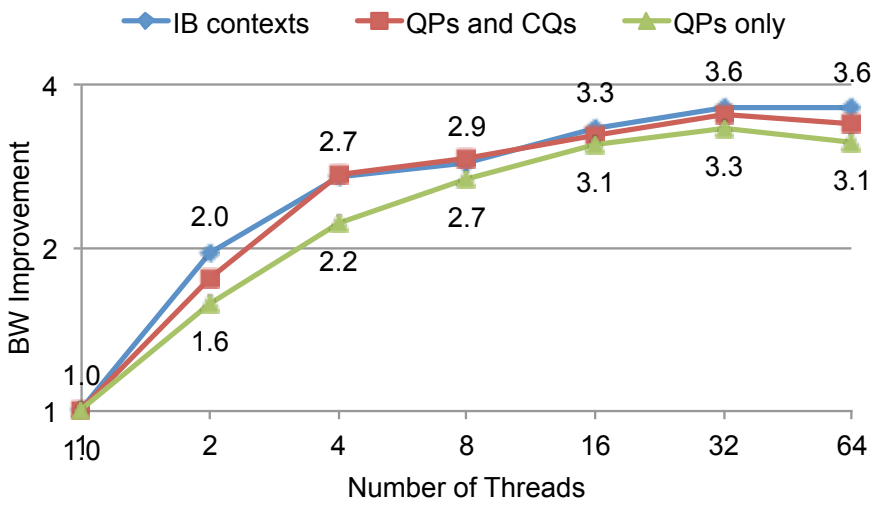


Figure 3.11: Small (64-byte) IB RDMA write bandwidth.

tween above parallelism levels. We conclude two primary observations from the figure.

1. The performance improvement with increasing threads is higher when the number of shared resources is less. For example, when each thread has a separate IB context (`IB contexts` level), with increasing threads, the parallel performance is 3.6-fold higher than the sequential performance. But when the context and the CQ are shared by all threads (`QPs only`), the parallel performance is only 3.1-fold higher than the sequential performance. This is because more sharing usually translates more critical sections and hence resulting in more serialization.

2. The maximum parallelism that the IB-stack can provide is 3.6-fold when all resources are distributed to different threads, and 3.1-fold when the IB context and CQ are shared between all the threads but only QPs are separately accessed by different threads. Most MPI implementations are increasingly utilizing more shared resources (i.e., similar to `QPs only` level) in order to manage the per-process resource usage. Thus, in our MPI implementation, 3.1-fold improvement is the maximum benefit from parallelism that we can expect as the ideal case.

3.3.2.3.2 Parallelism in MPI Netmod After studied the maximal benefit we can get from parallelism in IB communication, then we look into the parallelism strategies in MPI netmod. In our MPI implementation, the internal progress on every MPI process can be divided into two parts: one for sending and other for receiving. In the sending progress, small messages are always firstly copied in to an internal `preregister sending buffer` to avoid heavy overhead for IB memory registration; then an RDMA operation is posted into the corresponding QP to transfer user data from the preregister buffer to a remote process. In the receiving progress, the process firstly polls its internal `preregister receiving buffer` and then copies data from a cell in the receiving buffer to

the corresponding user buffer. Both the copy and the operation posing in the sending progress, the polling and the copy in the receiving progress are separate per connection, thus make them possible be parallelized by OpenMP. Following this notion, we designed our strategies as follows:

- **Posting Operations in Parallel.** As we have mentioned, most netmod implements the IB communication using multiple per-connection QPs and a global shared CQ. To minimize the mutexes required in the IB stack, we only assign each QP to a single thread. That is, multiple QPs might be managed by a single thread, however, a single QP is never managed by multiple threads. We are also carefully ensure that the number of threads used for parallelism is never more than the number of QPs (i.e., by setting `num_threads` clause in OpenMP parallel region), in order to avoid necessary thread synchronization overheads.

- **Copies from/to Preregistered buffer in Parallel.** In small-message communication, user data always needs to be copied into the internal sending buffer on the sender side, and received into the internal receiving buffer and copied out to user receiving buffer on the receiver side. Since each connection uses a separate QP and preregistered buffers, so the data copies on the send and receive side are also part of the parallelism and can be executed concurrently by multiple threads.

Although several places of the IB communication are suitable for parallelism, there are still some factors exist in MPI and limit the parallelism achievable in practice. One factor is that, the number of operations that can be issued to a QP or to the shared CQ is often limited. While the QP or CQ can be configured to allow for a large number of operations, such configuration causes performance degradation due to the internal bookkeeping associated with the data structures required in IB stack (e.g., the send queue in QP). Consequently, the MPICH IB netmod configures this limit to 1,024 for QPs and 32,768 for CQs, thus limiting the maximum number of network operations each thread can post to 1,024, and the maximum number of network operations posted across all threads to 32,768, before thread synchronization is needed. A similar parallelism-limiting factor is the number of preregistered buffers available at the sender and receiver side.

Except the MPI internal design, the application characteristics also constrain the possibility of parallelism. Specifically, since MT-MPI exploits OpenMP parallelism at the granularity of QPs, each of which is corresponding to a different remote process, for ideal parallelism we need the same amount of communication per peer process. In practice, however, this assumption can rarely hold. In most applications the amount of communication can vary dramatically between different processes, thus limiting the available parallelism.

3.4 Evaluation and Analysis

In this section, we evaluate the various techniques designed within MT-MPI. All our experiments are executed on the Stampede supercomputer at the Texas Advanced Computing Center [1]. Stampede consists of 6400 Dell Zeus C8220z

compute nodes, each with two Xeon E5-2680 processors and 32 GB RAM, and an Intel Xeon Phi SE10P coprocessor with 8 GB of on-board RAM connected by an x16 PCIe 2.0 interconnect. The nodes are interconnected by a Mellanox FDR InfiniBand network. All our experiments are executed on the Xeon Phi coprocessor, with every MPI process running on a separate coprocessor following the `native` mode.

3.4.1 Derived Datatype Processing

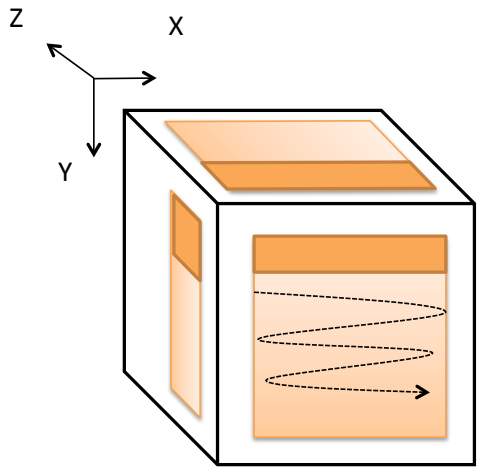
In this section, we describe three types of experiments that stress derived datatype processing to various degrees: (1) derived datatype packing performance, (2) halo data exchange with derived datatypes, and (3) the NAS multigrid benchmark. It is noted that we use a similar for loop for the sequential and parallel version in our comparison. It allows us to have a fair comparison where both modes are vectorizable, and both modes have the same issue with prefetching as described in Section 3.3.2.1. Thus, the improvement shown will be solely due to parallelization.

3.4.1.1 Derived Datatype Packing

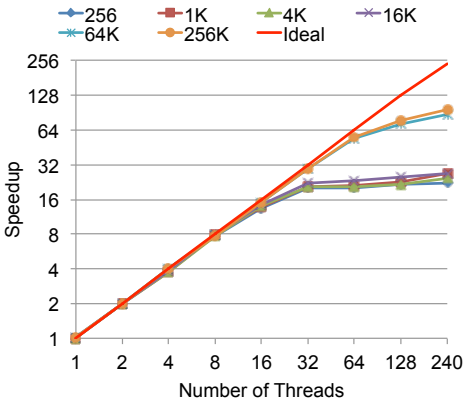
In our experiments with derived datatype packing (using `MPI_PACK`), we utilized a 3D matrix of doubles, with the X dimension as the leading dimension. The matrix volume was fixed at 1 GB, so increasing one dimension would reduce another. Our experiments involved packing different 2D planes of the 3D matrix.

Figure 3.12(b) shows the performance improvement while packing the top surface (X-Z plane). A vector datatype is utilized in this case, with a block length equal to the length of the X dimension and stride equal to the area of the X-Y plane; the Z dimension indicates the vector count. In our experiment, the Y dimension was fixed to 2 doubles, and the Z dimension varied as indicated on the graph legend (X dimension was varied to maintain the matrix volume). As can be seen in the figure, MT-MPI gets a reasonably good speedup with increasing number of threads, achieving a 96-fold improvement compared with the original sequential version when all 240 threads are used. A larger Z dimension provides better speedup because that leads to a larger iteration count for the contiguous copies and hence more parallelism that can be exploited by MT-MPI.

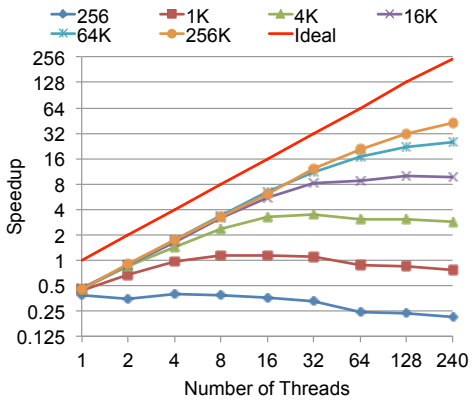
Figure 3.12(c) shows the performance improvement while packing the left surface (Y-Z plane). A two-level datatype comprising a vector of vectors is utilized in this experiment. The X dimension was fixed to 2 doubles, and the Y dimension varied as indicated on the graph legend (the Z dimension was varied to maintain the matrix volume). As shown in the figure, MT-MPI still achieves a relatively good speedup compared with the sequential version (42-fold), although less than what it achieved while packing the top surface. This reduction in performance is because the lowest-level vector datatype always has a block length of one double and a count equal to the Y dimension. This restricts the amount of work that is done within each iteration of the contiguous data copy operation and consequently limits the work done by each thread, especially when the number of iterations (i.e., the Y dimension) is small.



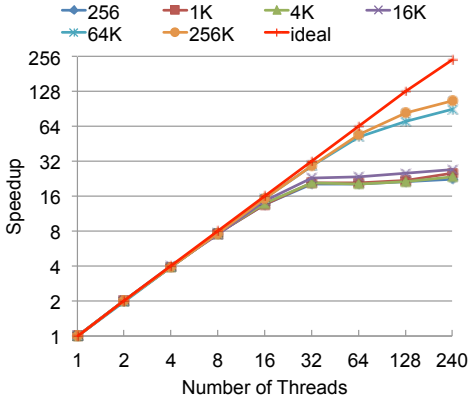
(a) Derived Datatypes in Three-dimension Matrix.



(b) Top surface with varying Z dimension.



(c) Left surface with varying Y dimension.



(d) Front surface with varying Y dimension.

Figure 3.12: Performance of parallel 3D packing.

3.4.1.2 Halo Exchange of Data

In our second set of experiments, we measured the performance of 3D halo exchanges of data as used in stencil computations. Both the data and the processes are partitioned into a 3D space. Each process communicates with its neighboring processes with which it shares a plane. For our experiments we define the following four dimension shapes for the local data on each process: (1) `Cube`, with dimensions 512 × 512 × 512 (doubles); (2) `Large X`, with dimensions 16K × 128 × 64; (3) `Large Y`, with dimensions 64 × 16K × 128; and (4) `Large Z`, with dimensions 64 × 128 × 16K. The MPI processes are evenly distributed in all dimensions.

Figure 3.13 shows the performance improvement achieved by MT-MPI compared with the sequential version when using 64 MPI processes. `Large Y` performs much better than the others, delivering a 23-fold speedup with 240 threads. To understand this behavior, we profiled the communication time for the different dimensions. The halo benchmark sends data in all dimensions simultaneously, so it is hard to profile how much time each dimension takes. Therefore, for profiling purposes, we modified it to serialize communication in one dimension at a time,

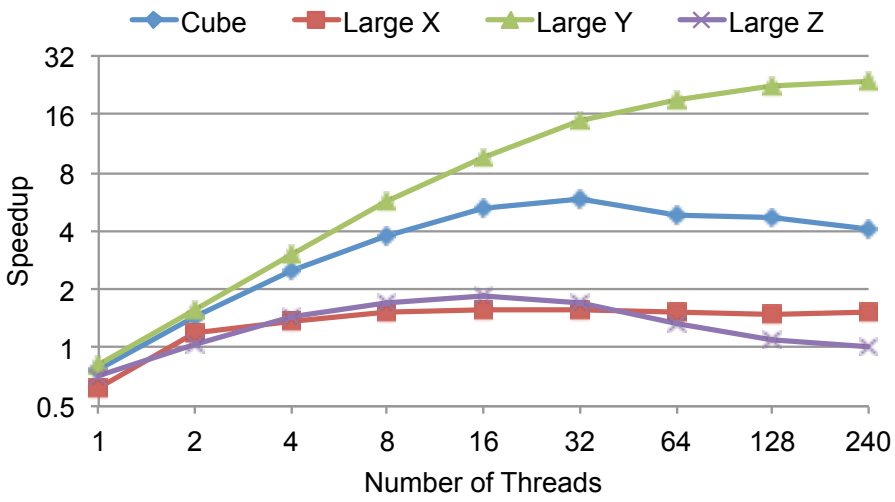


Figure 3.13: 3D internode halo exchange using 64 MPI processes.

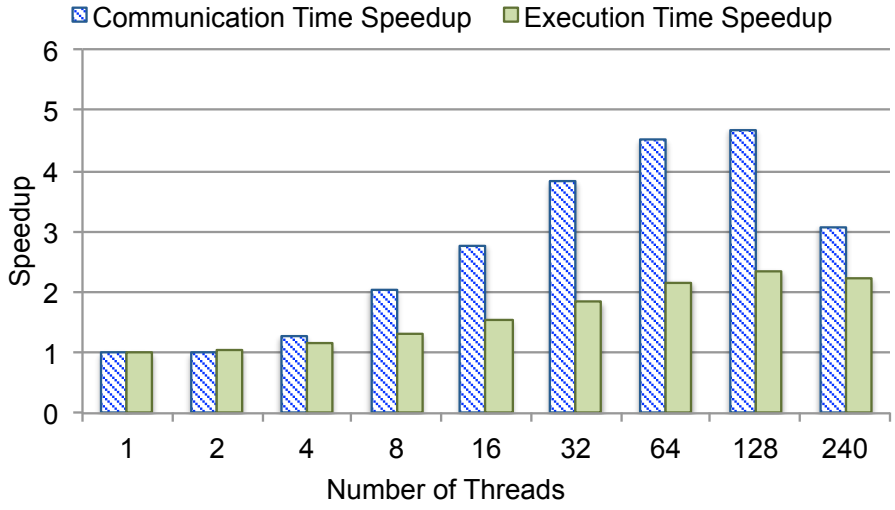
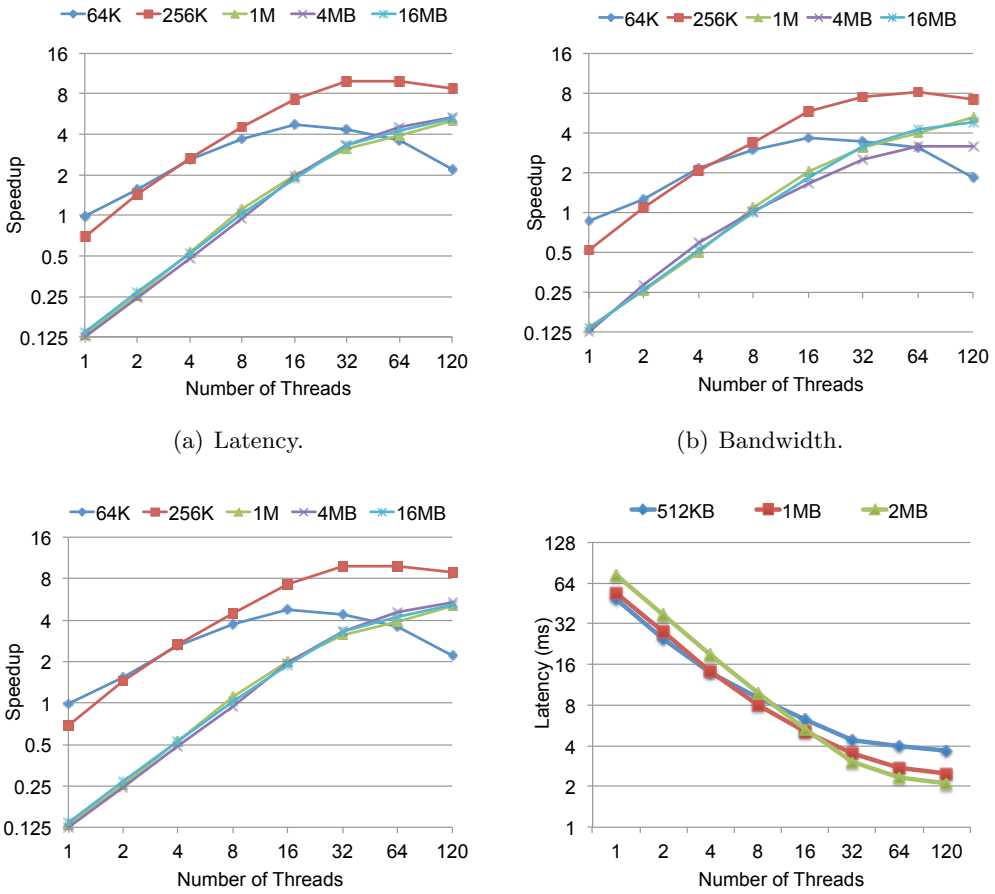


Figure 3.14: Hybrid MPI+OpenMP NAS MG Class E using 64 MPI processes.

and we observed that communication along the Y-Z dimension takes 85% of the time. While this is obviously not entirely indicative of the true halo benchmark that sends data in all dimensions simultaneously, it does give us some idea of the communication cost.

As demonstrated in Figure 3.12(c), a large Y dimension helps improve the performance of packing in the Y-Z dimension by providing better parallelism. This results in a large Y impacting the performance of the halo benchmark to the largest extent. With `Cube`, the Y-dimension is reduced to 512 doubles, thus reducing the speedup to around 5.8-fold as well. With `Large X` and `Large Z`, the Y-dimension further reduces to 128 doubles, which in turn reduces the overall speedup to around 1.6-fold and 1.8-fold, respectively.



(a) Latency.

(b) Bandwidth.

(c) Message rate.

(d) Varying shared buffer size (16MB message).

Figure 3.15: Shared-memory communication performance with varying message size between 2 MPI processes

3.4.1.3 NAS Multigrid Benchmark

We also evaluated a hybrid MPI+OpenMP version of the NAS Multigrid (MG) kernel [12] . The original MG kernel distributed as a part of the NAS parallel benchmarks does not contain a hybrid MPI+OpenMP version, so we modified the MPI version to (1) parallelize the local computation using OpenMP and (2) employ derived datatype communication instead of manual packing. The MG kernel implements a V-cycle multigrid algorithm to solve a 3D discrete Poisson equation. In every iteration of the V-cycle routine, halo exchanges are performed with various dimension sizes (count of double), from 2 to 514 in class E with 64 MPI processes, and so forth. The communication in all dimensions except the X-Y plane is noncontiguous.

Figure 3.14 presents the speedup achieved by MT-MPI compared with the original MPICH in class E (X, Y, Z dimension sizes are each 2K) when employing 64 processes. As shown in the figure, MT-MPI helps improve the communication of MG by 4.7-fold, and the overall execution time by 2.2-fold. The speedup in the communication time is still slightly lower than that of the 3D halo exchanges with the `Cube` shape shown in Figure 3.13. The reason is that the MG also contains

some halo exchanges with very small dimension size whose packing process cannot be parallelized efficiently.

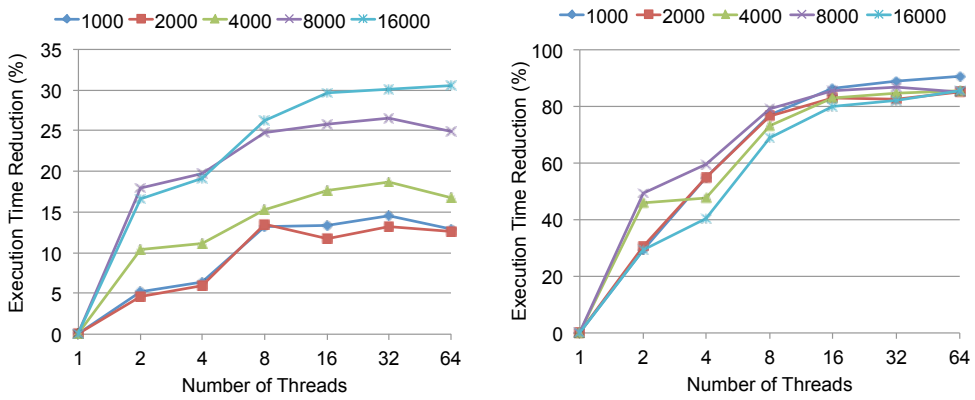
3.4.2 Shared-Memory Communication

To measure the impact of MT-MPI on intranode shared-memory communication, we evaluated the point-to-point communication benchmarks in the OSU MPI microbenchmark suite version 4.1 (http://mvapich.cse.ohio-state.edu/benchmarks/). In particular, we used the latency, bandwidth, and message rate benchmarks. Both the original MPICH and MT-MPI use an internal shared-memory region of 2 MB, with each cell containing 32 KB.

Figure 3.15 illustrates the performance of all three benchmarks; the legends in the graph represent different message sizes. We notice that the performance trends of all three benchmarks are similar, with MT-MPI delivering up to a 5-fold performance benefit for message sizes ≥ 1 MB, given enough parallelism. When the number of idle threads is ≤ 4, however, MT-MPI's performance is worse than that of the original MPICH. As discussed in Section 3.3.2.2, the reason is that MT-MPI loses some of the pipelining capabilities in the original MPICH code in return for thread parallelism. But with a small number of threads, this tradeoff is not beneficial.

Another observation we make in Figure 3.15 is that the speedup of MT-MPI for message sizes 64 KB and 256 KB is much better than that of other message sizes. This, however, is not because of MT-MPI's superior architecture. Rather, it is because the communication protocol thresholds (i.e., eager vs. rendezvous communication thresholds) in MPICH are tuned for regular Xeon systems, by default, and are too large for the Xeon Phi architecture. We did not change the default configuration of MPICH in order to avoid introducing yet another dimension of variance in the paper. Thus, for 64 KB and 256 KB message sizes, the original MPICH ends up using a suboptimal communication protocol, resulting in MT-MPI's performance falsely appearing to be significantly better as compared to other message sizes.

We next study the behavior of our parallel implementation when employing different shared buffer sizes. Our results for the latency benchmark when transferring 16 MB messages are shown in Figure 3.15(d). Other benchmarks expose similar behaviors, whereas this message size ensures we are showing the sustained performance of the pipeline, as several pipeline units are involved. As can be seen in the figure, when only a few of threads are available, smaller shared buffers provide slightly lower latency. We already discussed this issue in Section 3.3.2.2, this is because the parallel implementation reserves many available cells as a large contiguous cell, thus such a larger cell could result in a larger pipelining unit, but such less threads cannot copy them out more efficiently comparing with smaller pipelining units. We avoid this inefficiency by checking the number of idle threads to adjust the maximum combinable buffer size. On the other hand, a large enough number of threads benefits from larger pipeline units, which reduces the proportion of thread synchronization overhead.



(a) Overall Speedup

(b) Send Processing Speedup

Figure 3.16: One-sided communication benchmark with IB using 65 MPI processes.

3.4.3 InfiniBand Communication Operations

In this section we evaluate the performance benefits achieved by MT-MPI with our modifications to the MPICH IB netmod. We performed two types of experiments: (1) a one-sided communication microbenchmark designed to demonstrate the ideal parallelism that can be obtained within MT-MPI and (2) the one-sided version of Graph500 benchmark [49].

3.4.3.1 One-Sided Microbenchmark

We designed a microbenchmark in which one MPI process issues many `MPI_PUT` operations to all other processes. Each `MPI_PUT` operation is for 64 bytes. We measured the execution time of the benchmark using 65 MPI processes; thus each process communicates with 64 other processes and internally maintains 64 IB QPs. Figure 3.16(a) shows the speedup in execution time with MT-MPI compared with the original MPICH. As we increase the number of operations issued from 1,000 to 16,000, MT-MPI delivers an increasing performance benefit, reaching a 1.44-fold speedup when using 64 threads.

This performance benefit, however, is less than the ideal speedup of 3.1-fold that we can get by parallelizing IB communication, as discussed in Section 3.3.2.3. To understand the reason for this less-than-ideal speedup, we measured the execution time of the netmod send-side communication processing at the root process (SP), which consists only of the copy from the user buffer to a preregistered chunk and the posting of the operations to the IB network. Figure 3.16(b) shows that the execution time of SP delivers around 8-fold speedup when using 64 threads, which is as expected—we expect around a 3.1-fold speedup due to the parallelization in the posting of network operations, and some additional improvement due to the parallelized memory copy. Table 3.1 shows the relationship between the time spent in SP and the total execution time when issuing 16,000 operations. Although SP shows the expected performance improvement with MT-MPI, the percentage of time spent in SP is less than 10% when using more than 16 threads. This results in a reduction in the overall performance boost that we achieve.

Table 3.1: Profile of the one-sided communication benchmark.

Nthreads	Execution Time			Speedup	
	Total (s)	SP (s)	SP / Total (%)	Total	SP
1	5.8	2.2	38	1	1
4	4.7	1.3	27	1.2	1.7
16	4.0	0.4	10	1.4	5.0
64	4.0	0.3	8	1.4	6.9

3.4.3.2 Graph500 Benchmark

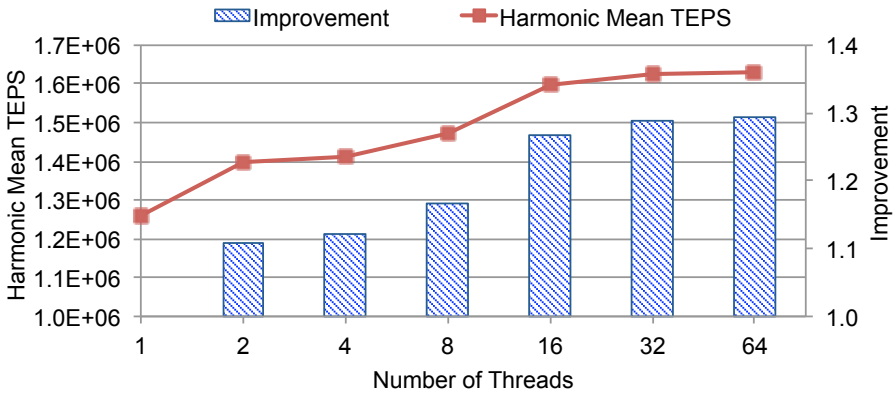


Figure 3.17: Performance of the Graph500 benchmark using 64 MPI processes.

The second benchmark we studied was the Graph500 benchmark [49], which performs a breadth-first vertex-visit operation on large graphs. In particular, we used a scale of 2^{22} and an edge factor of 16 on 64 MPI processes running on different Intel Xeon Phi coprocessors at different nodes. In the one-sided version of the Graph500 benchmark, every process issues many `MPI_Accumulate` operations to the other processes in every breadth-first search iteration.

Figure 3.17 shows the performance improvement of MT-MPI compared with the original MPICH. MT-MPI delivers a 1.3-fold improvement in the harmonic mean of the traversed edges per second (TEPS) when using 64 threads. As expected, this improvement is on par with the performance improvement we see in the one-sided communication benchmark that we discussed in Section 3.4.3.1. The slightly smaller speedup compared with the one-sided communication benchmark (which achieves a 1.44-fold speedup) is because the Graph500 benchmark does not uniformly communicate with all peer processes, thus causing some unevenness in MT-MPI's parallelization.

Chapter 4

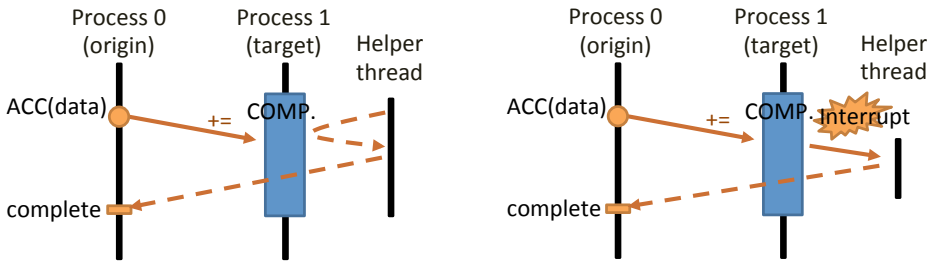
Process-based Asynchronous Progress

Publication

This chapter includes the contents that have been published in conference papers [61][60]. In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of the university of Tokyo's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to `http://www.ieee.org/publications_standards/publications/rights/rights_link.html` to learn how to obtain a License from RightsLink.

An increasing trend has been shown in scientific applications that the computation and communication are moving toward dynamic and data driven. Application developers are investigating the ways to better implement such communication rather than using the traditional send-receive patterns, since it becomes hard to specify matching pair of send/receive calls as in regular applications. MPI one-sided communication, as known as RMA, has been introduced from MPI-2 [5]. Its semantics could allow programmers to specify the communication in a more dynamic way that only the local process (origin) specifies the parameters for a data movement, without requiring a matching "receive" on the remote process (target). Unlike message passing, the data movement in RMA is more close to irregular memory access pattern, in which a process can read from/write to any location in the memory region on the other processes after acquired appropriate permission.

Not only the communication semantics, the RMA model could also provide asynchronous completion of data transfer (i.e., RMA operations in MPI) in order to hide the overhead of communication with computation on the target process. However, such asynchronous completion is not practically achieved in most MPI implementations and consequently limiting the performance of application execution. In this chapter, we will study the critical issue existing in MPI RMA communication. In Section 4.1 we first describe the essential problem in the implementation of RMA data movement, and summarize the status of traditional solutions in Section 4.2 with discussion around their limitations. In section 4.3,



(a) Thread-based Approach. (b) Interrupt-based Approach.

Figure 4.1: Tradition asynchronous progress approaches.

we give an overview of our solution proposed in this dissertation, and describe the detailed design and the practical challenges we resolved in section 4.4 and 4.5 respectively. In Section 4.7, we utilize both mirobenchmarks and a real chemistry application suite to evaluate our solution with comparison to the traditional approaches.

4.1 Problem Statement

While the RMA model is useful for dynamic and irregular communication patterns, the MPI standard does not guarantee that such data movement is truly asynchronous. In reality, most MPI implementations still require the MPI process to make MPI calls in order to ensure communication progress to complete any RMA operations issued on it as a target. Although RDMA (remote direct memory access) supported networks such as InfiniBand, Fujitsu Tofu or Cray Aries, could provide the capability of contiguous PUT/GET operations in hardware thus allowing MPI to offload the corresponding operations and achieve asynchronous communication, more complex operations (e.g., noncontiguous ACCUMULATE operation on 3D subarray) still have to be done in software within the MPI implementation. Consequently, arbitrarily long delay can happen in those operations if the target process is busy computing outside MPI stack and cannot make software progress.

4.2 Traditional Approaches

Several existing researches have looked into this problem and proposed two kinds of approaches—a thread-based mode and an interrupt-based mode —to provide asynchronous progress in MPI thus ensure asynchronous completion of RMA operations. However, none of them have been really used in applications due to several limitations. In this section, we briefly introduce each approach and discuss their limitation as follows.

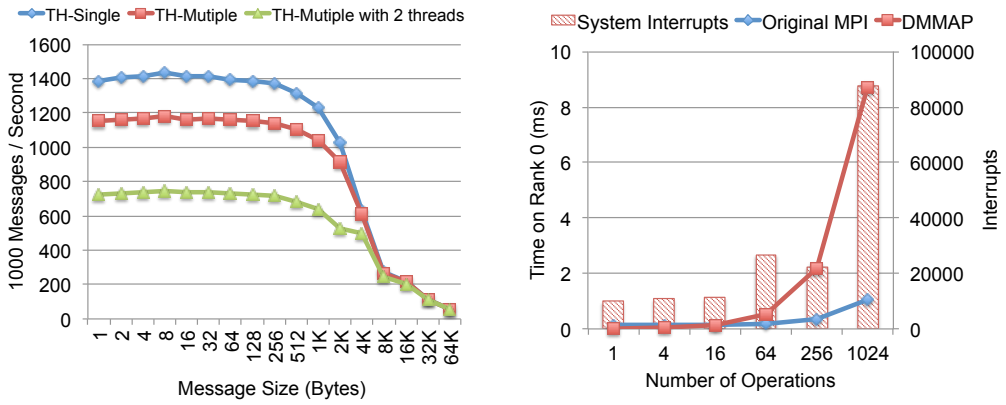
- **Thread-based approach.** In this approach, each MPI process utilizes a background thread in order to handle incoming messages from other processes (Figure 4.1(a)). This model is the most widely used approach and has been implemented in many MPI implementations, including MPICH [10], MVAPICH [70] and Intel MPI [36]. While being a generic approach for various MPI communication models, it raises performance concerns. One is

that a background thread can make progress for only the MPI process that spawned it, thus this model requires deploying at least as many background threads as MPI processes on every computing node. On current MPI implementations, where the progress engine polls repeatedly the network for incoming messages, this approach can waste half the computing resources or force core oversubscription. Another concern is, this model forces MPI implementations to implement multithreaded safety, which can bring further bottlenecks because of thread synchronization requirements. Figure 4.2(a) demonstrates the cost of multithreading based on the point-to-point message rate benchmark in OSU MPI microbenchmark suite with modification for multithreading [69]. Even only enabling the muliththreading safety with only one thread per process (TH-Multiple), significant overhead can be shown compared to the default thread single safety (TH-Single); performance is degraded even more when involving another thread to poll MPI progress (e.g., by waiting for receiving a message from the main thread) on each MPI process (TH-Multiple with 2 threads).

- **Interrupt-based approach.** In the interrupt-based model, hardware interrupts are issued to awaken a thread in order to process the incoming RMA messages. This model is used by Cray MPI [20] when RMA uses the DMAPP conduit (not currently the default); in this case, the interrupt wakes up a kernel thread. MPI on Blue Gene/P [28, Chapter 7] and Blue Gene/Q [43] use special hardware to cause a context switch that cause a special thread to wake up and drive the network when a message arrives; in this case, the thread is a user thread, which allows for arbitrary code to run, unlike a kernel thread. While interrupt-driven asynchrony does not require dedicated resources the way polling threads do, handling per-operation interrupts on cores that are otherwise devoted to computation causes those cores to stop computing temporarily and leads to cache pollution. Figure 4.2(b) demonstrates such cost in the Cray DMAPP asynchronous progress by using a simple RMA communication on two processes where one process does `lockall-accumulate-unlockall` while the other process does a `dgemm` computation. With increasing number of ACCUMULATE operations, it is obvious that the overhead of the DMAPP approach increases with the amount of system interrupts issue on the second process.

4.3 Solution

In this dissertation, we present "Casper," a process-based asynchronous progress solution on multi- and many-core architectures to address the inefficient communication in MPI RMA. Unlike traditional approaches, the philosophy of the Casper framework is centered on the notion that since the number of cores on computing nodes is growing rapidly, not all of the core are always busy computing during the execution, thus dedicating a few of the cores for helping with asynchronous progress might be better than using an interrupt-based model. Figure 4.3 shows an image for such core deployment. Comparing to the thread-based approach, the use of processes rather than threads allows Casper to control the



(a) Overhead of Multithreading.

(b) Overhead of System interrupts.

Figure 4.2: Performance issues in traditional solutions.

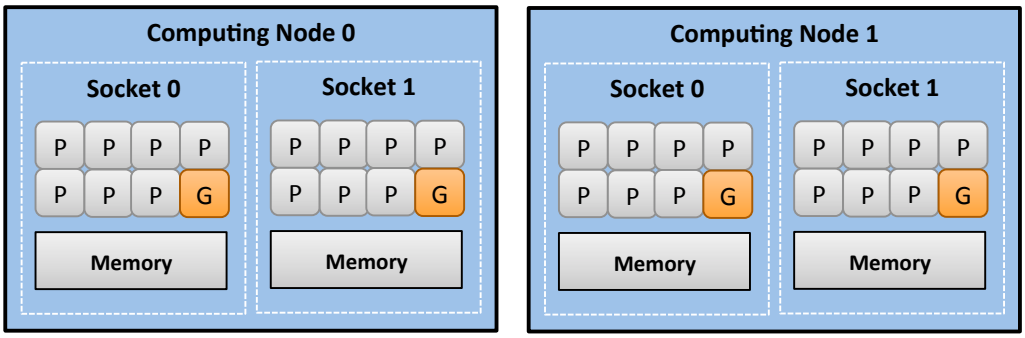


Figure 4.3: Casper core deployment.

amount of sharing thus eliminate the thread-safety overheads, as well as to control the number of cores being utilized for asynchronous progress. In summary, we believe Casper provides a more suitable solution that the traditional approaches for the asynchronous progress on large many-core architectures.

The central idea of Casper is the ability of processes to share memory by mapping a common memory object into their address spaces by using the MPI-3 shared memory windows interface. Specifically, Casper keeps aside a small user-specified number of cores on a multi- or many-core environment as "ghost processes." When the application process tries to allocate a remotely accessible memory window, Casper intercepts the call and maps such memory into the ghost processes' address space. Casper then intercepts all RMA operations to the user processes on this window and redirects them to the ghost processes instead.

Since the user memory regions are not migrated or copied but just mapped into the ghost processes' address space, RMA operations that are implemented in hardware see no difference in the way they behave. On the other hand, RMA operations that require remote software intervention can be executed in the ghost processes' MPI stack on the additional cores kept aside by Casper, without requiring any intervention from the application processes.

Although the core concept of Casper is straightforward, the design and implementation of such a framework must take several aspects into consideration. Most important, the framework needs to ensure that correctness is maintained

as required by the MPI-3 semantics. While this task is easy to manage for simple applications, the wide variety of communication and synchronization models provided by MPI can make the task substantially more complex for applications that are nontrivial. This task is even more complicated for applications that use multiple MPI-3 epoch types (e.g., passive-target and active-target) or multiple windows of remote memory buffers because the same Casper ghost processes need to maintain progress on all of them, thus essentially requiring that they never indefinitely block inside an MPI operation. Furthermore, when more than one ghost process is present, the Casper architecture must ensure that the ordering, atomicity, and memory consistency requirements specified by the MPI-3 standard are met in a way that is transparent to the application.

The Casper architecture hides all this complexity from the user and manages it internally within its runtime system. In some cases, however, such complexity can cause performance overhead. In this paper we present various techniques we used to ensure correctness while retaining the performance of the RMA operations and enabling low-overhead asynchronous progress. In addition to a detailed design of the Casper architecture, we present experiments evaluating and analyzing Casper with various microbenchmarks and a large quantum chemistry application.

4.4 Casper Design Overview

In this section, we present an overview of the design and workings of the Casper architecture. Casper is designed as an external library through the PMPI name-shifted profiling interface of MPI. This allows Casper to transparently link with various MPI implementations, by overloading the necessary MPI functions.

Casper provides three primary functionalities. First, it deploys "ghost processes" on each node that are separate from the main application processes. These ghost processes assist in processing RMA operations targetted to that node. Second, when the user tries to allocate remotely accessible memory, it sets up the allocated memory such that the memory is mapped into the address space of both the application processes as well as the ghost processes, thus allowing any of these processes to access the data. Third, it redirects communication operations intended to a user process to the corresponding ghost processes, thus allowing them to be handled by the ghost processes instead of the application processes. These three steps are described in more detail in the following subsections.

4.4.1 Deployment of Ghost Processes

Ghost processes in Casper are allocated in two steps. In the first step, when the user launches the application with a number of processes, a user-defined subset of these processes is carved aside as the ghost processes at MPI initialization time. The remaining processes form their own subcommunicator called `COMM_USER_WORLD`. The number of ghost processes is user-defined through an environment variable, allowing the user to dedicate an arbitrary number of cores on the node for the ghost processes.

In the second step, Casper overrides all MPI operations that take a communicator argument and replaces any occurrence of `MPI_COMM_WORLD` in all non-RMA functions with `COMM_USER_WORLD` at runtime through PMPI redirection. This

step ensures that all non-RMA communication is redirected to the correct MPI processes, including creation of other subcommunicators from `MPI_COMM_WORLD`.

After initialization, ghost processes simply wait to receive any commands from user processes in an `MPI_RECV` loop. This approach ensures that while the ghost processes are waiting for commands, they are always inside the MPI runtime, thus allowing the MPI implementation to make progress on any RMA operations that are targeted to those ghost process.

One aspect to consider in Casper is the locality of application buffers relative to the ghost processes. Specifically, since a ghost process might be depositing or reading data from the application buffers, how far the ghost process is compared with the buffers can have a serious impact on performance. To handle this issue, we ensure that the ghost processes in Casper are topology-aware. Casper internally detects the location of the user processes and places its ghost processes as close to the application process memory as possible. For example, if a node has two NUMA domains and the user requests two ghost processes, each of the ghost processes places itself in a different NUMA domain and binds itself to either the process ranks or segments in that NUMA domain.

4.4.2 RMA Memory Allocation and Setup

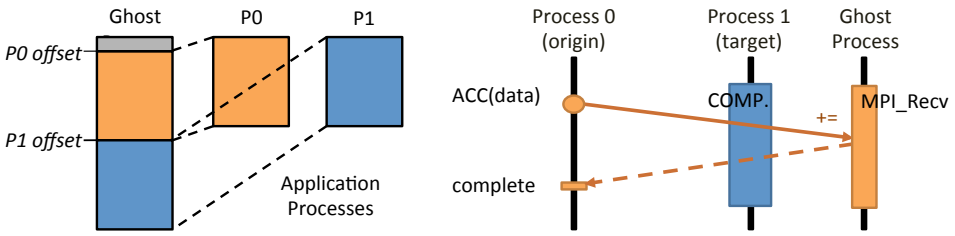
Remote memory allocation in the Casper architecture is tricky in that the allocated memory must be accessible by both the application processes and the ghost processes. MPI provides two broad mechanisms to declare a memory region as remotely accessible. The first is an "allocate" model (i.e., `MPI_Win_allocate` and `MPI_Win_allocate_shared`) in which MPI is responsible for creating such memory, thus allowing the MPI implementation to optimize such allocation (e.g., through shared memory or globally symmetric virtual memory allocation). The second is a "create" model (i.e., `MPI_Win_create` and `MPI_Win_Create_dynamic`), in which the user allocates memory (e.g., using `malloc`) and then exposes the memory as remotely accessible.

While memory sharing between the application processes and the ghost processes can occur in both models, doing so in the "create" model requires OS support to expose such capability. This capability is generally present on large supercomputers such as Cray (e.g., through XPMEM [78] or SMARTMAP [14]) and Blue Gene, but not always on traditional cluster platforms. Thus, for simplicity, we currently support only the "allocate" model.

When the application creates an RMA window using `MPI_Win_allocate`, Casper follows a three-step process:

1. It first allocates a shared-memory region between the user processes and the ghost process on the same node using the MPI-3 `MPI_Win_allocate_shared` function, as depicted in Figure 4.4(a). As shown in the figure, the same memory region that is used by the application is also mapped onto the address space of the ghost process. Thus, such memory is accessible through either process, although care must be taken to keep it consistent.

2. Once the shared memory is allocated, it creates a number of internal windows using `MPI_Win_create` to expose this memory to all user and ghost processes.



(a) Shared buffer mapping.

(b) Operation redirection

Figure 4.4: Casper basic design.

3. Casper creates a new window with the same memory region that contains only the user processes; it then returns the new window handle to the application.

We note that the Casper architecture exposes the allocated shared-memory in multiple overlapping windows. This model provides Casper's runtime system with enough flexibility to manage permissions and communication aspects in a highly sophisticated manner; but at the same time the model requires extreme caution to ensure that memory is not corrupted and is consistent with the user's expectation. In Section 4.5xx, we describe how these internal windows are utilized in Casper.

4.4.3 RMA Operation Redirection

Once the window becomes ready, Casper transparently redirects, through PMPI redirection, all user RMA operations to the ghost processes on the target node. Such redirection needs to translate both the target rank that the RMA operation is addressed to and the target offset where the data needs to be written to or read from (since the offset in the ghost process's memory region might not be the same as the offset in the user process's memory region). For example, based on Figure 4.4(a), if an origin process does an RMA operation at offset "X" of user process P1, Casper will redirect the operation to offset "X + P1's offset in the ghost process address space" on the ghost process.

When multiple ghost processes are available on the target node, Casper attempts to utilize all of them by spreading communication operations across them. This approach allows the software processing required for these operations to be divided between the different ghost processes, thus improving performance. Using such a model with multiple ghost processes, however, requires extra care compared with using a model with a single ghost process. Moreover, it raises a number of correctness issues, as we discuss in Section 4.5.3.

4.5 Ensuring Correctness and Performance

In this section we discuss several corner cases that we need to handle inside Casper in order to maintain correctness as per the MPI-3 standard while achieving high performance. Of these, some of the correctness aspects are specific to the *lock-unlock* epoch type (discussed in Sections 4.5.1,and 4.5.2), while the rest are generic to all epoch types (discussed in Sections 4.5.3, 4.5.4, and 4.5.5).

With respect to performance optimizations, some of the proposed optimizations are automatically detected and handled by the Casper implementation while some others are based on user hints in the form of either *info* hints or *assert* hints, as specified by the MPI standard. Both *info* and *assert* hints are, in essence, user commitments to comply with different restrictions which allow the MPI implementation to potentially leverage different optimizations. *info* arguments are broad-sweeping hints that apply to an entire window and all operations issued on that window. Further, *info* hints are extendable so each MPI implementation can add newer hint capabilities to improve its own performance. *assert* hints, on the other hand, are more focussed in scope and typically apply to each epoch. They are also not as easily extendable to MPI implementation-specific hints.

All *assert* hints used in Casper are MPI-3 standard defined hints that we reuse with the same semantics as the standard. Thus, these hints are compatible with other MPI implementations as well, even though some MPI implementations might not choose to take advantage of them. The *info* hints used in Casper are not defined by the MPI-3 standard and are Casper-specific extensions.

4.5.1 Lock Permission Management for Shared Ghost Processes

Consider an environment where multiple application processes reside on different cores of the same node and thus share a ghost process. In this case, all RMA communication to these application processes would be funneled through the same ghost process. In such an environment if an origin wanted to issue an exclusive lock to more than one application process on the same node, such a step would result in multiple exclusive lock requests being sent from the origin to the same ghost process. This is disallowed by the MPI standard—an origin cannot nest locks to the same target. Similarly, if two origin processes issue exclusive locks to different application processes on the same node, this would result in multiple exclusive lock requests being sent from different origins to the same ghost process. While this is correct according to the MPI standard, it would result in unnecessary serialization of all exclusive locks to processes on the same node, thus hurting performance significantly.

To overcome this issue, Casper internally maintains separate overlapping windows for each user process on the node. In other words, if a ghost process is supporting N user processes, it will create N overlapping windows. Communication to the ith user process on each node goes through the ith window. Thus, the number of internal overlapping windows created is equal to the maximum number of user processes on any node of the system. Such overlapping windows allow Casper to carefully bypass the lock permission management in MPI when accessing different processes but to still take advantage of them while accessing the same process. Since a single RMA communication operation cannot target multiple processes at the same time, we never run into a case where the bypassing of permission management across processes causes an issue.

While this approach ensures correctness, it can be expensive for both resource usage and performance. To alleviate this concern, we allow the user to use the info hint `epochs_used` to specify a comma-separated list of epoch types that the user intends to use on that window. The default value for this info key is all

epoch types (i.e., "fence,pscw,lock,lockall"); but if the user sets this value to a subset that does not include "lock," Casper can use that information to create only a single overlapping window (apart from the user-visible window) for all its internal operations and reduce any overhead associated with lock permission management.

4.5.2 Self Lock Consistency

In general, locks are nonblocking operations in that they do not need to wait till the lock is actually acquired. The MPI implementation only needs to ensure that any future RMA operations are not issued to the target memory before the lock is actually acquired. Self locks (i.e., when a process locks itself), however, are special in that they cannot return till the lock is actually acquired. This requirement is because self locks allow applications to access their local memory directly using load/store operations instead of MPI RMA communication operations. In such cases, the accesses are outside of MPI's control and thus the MPI implementation needs to make sure to acquire the lock before returning from the lock call thus forcing all load/store operations to happen after the lock is acquired.

With Casper, lock operations are redirected to the ghost processes in order to maintain appropriate permissions in case other origins are trying to access the window at the same time. This, however, means that the lock is no longer a self-lock, but to a remote process. Thus, the MPI implementation might choose to delay the lock acquisition or return from the lock call before the lock acquisition is complete. At that point a process issuing load/store operations to itself can cause data correctness issues.

To handle this issue, Casper performs two steps. In the first step, it issues a lock to the ghost process. However, since the acquisition of this lock might be delayed by the MPI implementation, Casper internally issues an additional 1-byte GET from the window and performs a flush to complete that operation. This forced GET would, in essence, block till at least the lock is acquired since the GET operation cannot fetch data before the lock is acquired. One issue with this approach is that the MPI-3 standard (page 456, line 39) states that data consistency is not guaranteed when a GET operation and an update operation (such as a PUT or ACCUMULATE operation) occur simultaneously at the same memory location. Thus, if the user application is updating the same location as the one that is being fetched by the forced GET described above, data consistency is not guaranteed. Such data inconsistency makes sense for the GET operation but seems to be an unnecessary restriction on the update operation, i.e., the update should still be valid in such cases. As active members of the MPI Forum RMA working group, we believe this is an unintended oversight and should be fixed in the upcoming MPI-3.1 or MPI-4.0 standard. However, in order to meet the strict wording of the standard, we need an alternative approach. Therefore, in Casper, we allocate additional "hidden bytes" at the ghost process (depicted by the gray box in Figure 4.4(a)) that are not exposed to the user application. The additional force GET operation is issued on these hidden bytes thus guaranteeing that it cannot cause any potential corruption of user data.

In the second step, now that the permission issue is managed by the lock at

the ghost process, Casper issues a second self-lock at the origin process. This lock does not manage any permissions (since it is guaranteed to be not competed), but is necessary for managing memory consistency through appropriate system-specific memory barriers that the MPI implementation might be required to do.

While the above solution maintains correctness, it clearly adds additional lock acquisition overhead that can have performance implications. In order to alleviate such performance impact, we use the info hint `no_load_store` to let the user specify when she does not intend to access the local window with load/store operations, i.e., all data movement to/from the remotely accessible memory will be done using MPI RMA operations. With this hint, Casper would still issue the lock operation to the ghost process, but does not have to force acquire it. Furthermore, Casper can skip the second self-lock completely since that is only needed for local load/store operations.

The user can also use the `MPI_MODE_NOCHECK` assert (which is already specified in the MPI-3 standard) to help in this case. The `MPI_MODE_NOCHECK` hint tells us that the user is guaranteeing that there will be no contention on the lock and hence on the access permissions to the window, thus removing the necessity for the force lock for permission management.

As a side benefit, this hint also allows us to issue local PUT/GET operations directly instead of forwarding them to the ghost process, since we know that the ghost process would not be receiving any conflicting epochs at the same time. It is still possible that the ghost process will receive a PUT/GET operation to the same location as the local PUT/GET operations, but the MPI standard already states that doing so can result in data corruption, which covers data conflicts within Casper as well. We note that we do not use this optimization for accumulate-style operations since that would break the atomicity constraints enforced by the MPI standard.

4.5.3 Managing Multiple Ghost Processes

In Casper, the user is allowed to configure a node with multiple ghost processes. This allows better sharing of work when the number of operations requiring such asynchronous progress is large. However, such a configuration also requires additional processing to maintain correctness. A simple model where all communication is randomly distributed across the different ghost processes has two issues that need to be handled: (1) lock permissions in the *lock-unlock* epoch and (2) ordering and atomicity constraints for accumulate operations.

When the *lock-unlock* epoch type is used, Casper will internally lock all ghost processes on a node, when a lock operation for a particular application process is issued, in the hope of spreading communication across these helper processes. However, in practice, many MPI implementations might not acquire the lock immediately and delay them to a future time (e.g., when an RMA communication operation is issued to that target). Given this background, consider the sample code demonstrated in Figure 4.5. In this example, Casper might randomly pick a ghost process thus picking one ghost process (G1) on one origin while picking a different ghost process (G2) on another origin. For implementations that delay lock acquisition, this would mean that the two ghost processes would get exclusive

```
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, P1, 0, win);
MPI_Put(..., P1, ...);
MPI_Win_unlock(P1, win);
```

(a) User code

```
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, G1, 0, win);
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, G2, 0, win);

/* Pick a random ghost process */
G = randomly_pick_ghost();
MPI_Put(..., G, ...);

MPI_Win_unlock(G1, win);
MPI_Win_unlock(G2, win);
```

(b) Casper translated code

Figure 4.5: Sample code demonstrating lock acquisitions issues with multiple ghost processes.

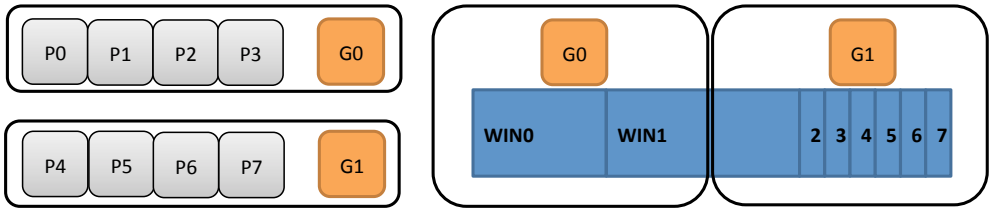
locks from two different origins to access the same memory location. Since the lock management within MPI is unaware of the shared memory buffers in Casper, both exclusive locks would be granted resulting in data corruption.

The second issue is with respect to the atomicity and ordering guarantees provided by the MPI standard for concurrent accumulate operations to the same location (see [5], Section 11.7.1). Each basic datatype element of concurrent accumulate operations issued by the same or different origin processes to the same location of a target process must be performed atomically. Similarly, two accumulate operations from the same origin to the same target at the same memory location are strictly ordered. In Casper, if a user process is served by a single ghost process, such atomicity is already provided by the MPI implementation. However, if a user process is served by multiple ghost processes, they might simultaneously be accessing the same memory region thus breaking both atomicity and ordering.

To address these issue, Casper uses a two-phase solution. The first phase is to provide a base "static binding" model where each ghost process is statically assigned to manage only a subset of the remotely accessible memory on the node. This model ensures correctness as per the MPI standard but can have some performance cost. We propose two static binding approaches in this paper: rank binding and segment binding. The second phase is to identify periods in the application execution where the issuing of some operations to ghost processes can be done in a more dynamic fashion. In this section, we discuss both phases.

4.5.3.1 Static Rank Binding

In this model, each user process binds to a single ghost process, and any RMA operations issued to that user process are always directed to that ghost process.



(a) Rank binding. (b) Segment binding.

Figure 4.6: Static binding strategies.

Therefore, different origins locking the same target would be redirected to the same ghost process thus benefiting from MPI's internal permission management. Similarly, different accumulate operations targetting the same user process would also be redirected to the same ghost process thus benefiting from MPI's internal ordering and atomicity management. This model completely works around the problem with multiple ghost processes since each user application process is only associated with a single ghost process. The disadvantage of this approach, however, is that if the amount of communication to the different user application processes is not uniform, one ghost process might get more work than the others, thus causing load imbalance.

4.5.3.2 Static Segment Binding

In this model, the total memory exposed by all the processes on the node is segmented into as many chunks as the number of ghost processes and each chunk or segment is bound to a single ghost process. Thus, given a particular byte of memory, a single ghost process "owns" it. When the user application issues a lock operation, Casper will still lock all ghost processes, but when the actual RMA communication operation is issued, it is redirected to the appropriate ghost processes that own that segment of the memory. In this model it is possible that different origin processes may get simultaneous access to the same target through different ghost processes. However, they cannot simultaneously update the same memory region, thus making such shared access inconsequential and still guaranteeing application correctness.

A second aspect that must be considered in segment binding is that the segmentation needs to be at a basic-datatype-level granularity in order to maintain MPI's requirements for atomicity. To handle this we must ensure that the segments are divided at an alignment of the maximum size of MPI basic datatypes (i.e., 16 bytes for `MPI_REAL`). This alignment is in order to guarantee no basic datatype is divided between two ghost processes. Thus, although an operation may be divided into multiple chunks and issued to different ghost processes, each basic datatype unit belongs to a single chunk and is directed to a single ghost process, thus guaranteeing atomicity and ordering. This approach would work in most cases since most compilers enable data alignment by default (i.e., a double variable has to be allocated on an address that is a multiple of eight). Hence, it is safe to divide an operation into different aligned segments. However, we note that this approach is not strictly portable. Compilers are allowed to not enforce data alignment or allow users to explicitly disable structure padding, re-

```
struct __attribute__((__packed__)) Foo {
    char a;
    double b;
};
```

Figure 4.7: Padding disabled structure.

sulting in unsafe segmentation (see Figure 4.7 which is a valid datatype with the GNU compilers). Nevertheless, data alignment is always recommended for performance, and some architectures, such as SPARC [63], even require it for correctness.

The advantage of the static segment binding model compared to the static rank binding model is that the load on a given ghost process is determined by the memory bytes it has access to, rather than the process it is bound to. In some cases, such a model can provide better load balancing than the previous model. However, this model has several disadvantages. Most importantly, this solution relies on analyzing the specific bytes on the target process that are being accessed for each RMA operation. For operations using contiguous datatypes that completely fall within one data segment, this model can be straightforward as the operation is simply forwarded to the appropriate ghost process. However, if the data overlaps two or more segments, Casper would need to internally divide the operation into multiple operations issued to different ghost processes. This solution becomes even more complex when the data being transmitted is noncontiguous, in which case the datatype needs to be expanded and parsed before the segments it touches can be determined.

4.5.3.3 Dynamic Binding

In applications that have balanced communication patterns, each target process on a compute node tends to receive approximately equal number of RMA operations. The best performance can be achieved for such patterns by equally distributing the number of processes handled by each ghost process. In such cases, a static binding approach might be a good enough solution for load balancing. However, for applications with more dynamic communication patterns, a more dynamic selection of ghost processes is needed, as long as such an approach does not violate the correctness requirements described above.

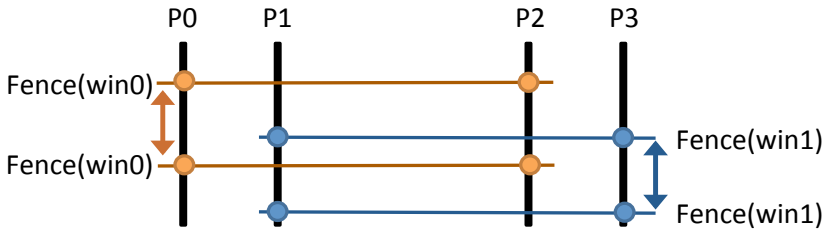
In Casper, to help with dynamic binding, we define "static-binding-free" intervals of time. For example, suppose the user application issues a lock operation to a target—this would translate to a lock operation to the corresponding ghost process to which the process is bound. After issuing some RMA communication operations if the user application flushes the target, at this time the MPI implementation is required to wait for the lock to be acquired and cannot postpone this process any further. The period after the flush operation has completed and before the lock is released is considered a "static-binding-free" period. That is, in this period we know that the lock has already been acquired. In such periods, the Casper implementation no longer has to do lock permission management and is free to load balancing PUT/GET operations to any of the ghost processes with

the same lock type as that specified by the user application process. We note that this optimization is not valid for accumulate-style operations in order to maintain the atomicity and ordering guarantees specified by the MPI standard.

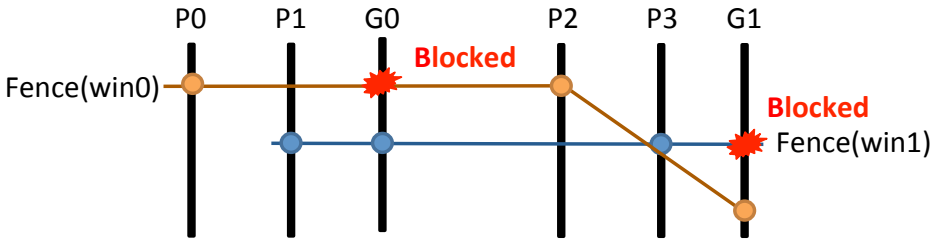
We utilize three dynamic load balancing approaches within Casper. The first is a "random" algorithm that randomly chooses a ghost process from the available ghost processes for each RMA operation. The second is an "operation-counting" or "round-robin" algorithm that chooses the ghost process that the origin issued the least number of operations to. The third is a "byte-counting" algorithm that chooses the ghost process that the origin issued the least number of bytes to.

4.5.4 Dealing with Multiple Simultaneous Epochs

The MPI standard does not allow a process to simultaneously participate in multiple overlapping epoch types on a given window. However, for disjoint sets of processes or for the same set of processes with different windows, no such restrictions exist. Thus, one could imagine an application in which a few of the processes are participating in a *lock-unlock* epoch on one window, while another disjoint set of processes is participating in a *fence* epoch on another window. If more than one of these processes are on the same node, the ghost processes have to manage multiple simultaneous epochs. The primary difficulty with handling multiple simultaneous epochs, especially active target epochs such as *fence* and *PSCW*, is that the epoch opening and closing calls in these epochs are collective over either all or a subset of processes in the window and these calls are blocking with no nonblocking variants. Thus, if a ghost process participates in one epoch opening or closing call, it is stuck in a blocking call and hence loses its ability to help with other epochs for other user processes. Figure 4.8 illustrates such issue in simultaneous fence calls on two disjoint sets of processes which are sharing the same ghost processes (i.e., group [P0, P2] and [P1, P3] share ghost process G0 and G1 on two nodes).



(a) Original MPI.



(b) With ghost process

Figure 4.8: Simultaneous Fence.

To work around this issue, Casper converts all active-target epochs into passive-target epochs on a separate window. Further, it manages permission conflicts between *lockall* and *lock* by converting *lockall* to a collection of *lock* operations in some cases. The following paragraphs describe these changes in more detail.

4.5.4.1 Fence

The *fence* call supports a simple synchronization pattern that allows a process to access data at all processes in the window. Specifically, a fence call completes an epoch if it was preceded by another fence and starts an epoch if it is followed by another fence.

In Casper, we translate *fence* to a *lockall-unlockall* epoch. Specifically, we use a separate window for *fence*; and when the window is allocated, we immediately issue a *lockall* operation. When the user application calls *fence*, we internally translate it to *flushall-barrier*, where the *flushall* call ensures the remote completion of all operations issued by that origin and the *barrier* call synchronizes processes, thus ensuring the remote completion of all operations by all origins. This model ensures that the ghost processes do not need to explicitly participate in any active target synchronization calls, thus avoiding the blocking call issues discussed above.

While correct, this model has a few performance issues. First, a *fence* call does not guarantee remote completion of operations. The return of the *fence* call at a process guarantees only the local completion of operations issued by that process (as an origin) and the remote completion of operations issued to that process (as a target). This is a weaker guarantee than what Casper provides, which is remote completion of all operations issued by all processes. Casper's stricter guarantees, while correct, do cost performance, however. Therefore, such remote completion through *flushall* can be skipped if the user provides the `MPI_MODE_NOPRECEDE` assert indicating that no operations were issued before the *fence* call that need to be flushed.

Second, an MPI implementation can choose to implement *fence* in multiple different ways. For example, one possible implementation of the *fence* epoch is to delay all RMA communication operations to the end of the epoch and issue them only at that time. Thus, if the MPI implementation knows that a *fence* call does not complete any RMA communication operations (e.g., if it is the first fence), it can take advantage of this information to avoid synchronizing the processes. Casper does not have this MPI implementation internal knowledge, however. Thus, it always has to assume that the MPI implementation might issue the RMA communication operations immediately, and consequently it always has to synchronize processes. Again, doing so costs performance. However, if the user specifies the `MPI_MODE_NOSTORE`, `MPI_MODE_NOPUT`, and `MPI_MODE_NOPRECEDE` asserts, Casper can skip such synchronization since there are no store operations before the *fence* and no PUT operations after the *fence* that might impact the correctness of the data.

Third, when *fence* is managed by the MPI implementation, it internally enforces memory consistency through appropriate memory barriers. In Casper, since the *fence* call is translated to passive-target synchronization calls, such

memory consistency has to be explicitly managed. Thus, during each *fence* call, we add an additional call to `MPI_Win_sync` to allow such memory ordering consistency, costing more performance.

4.5.4.2 PSCW

The *PSCW* epoch allows small groups of processes to communicate with RMA operations. It explicitly decouples calls in order to expose memory for other processes to access (exposure epoch) and calls to access memory from other processes (access epoch). The `MPI_Win_post` and `MPI_Win_wait` calls start and end an exposure epoch, while the `MPI_Win_start` and `MPI_Win_complete` start and end an access epoch.

As with *fence*, we translate the *PSCW* epoch to passive-target synchronization calls on the same window (since *fence* and *PSCW* cannot simultaneously occur on the same window). Also as with *fence*, we add additional process synchronization for *PSCW* in Casper. Instead of using *barrier*, however, we use *send-recv* because the processes involved might not be the entire group of processes on the window. Consequently, *PSCW* encounters the same set of drawbacks as *fence* with respect to performance. To help with performance, we allow the user to provide the `MPI_MODE_NOCHECK` assert specifying that the necessary synchronization is being performed before *post* and *start* calls. When this assert is provided, Casper can drop additional synchronization.

4.5.4.3 Lockall

The *lockall* epoch is a passive-target epoch and thus does not require participation from the ghost processes. However, we need to be careful that we do not bypass lock permission requirements when the user uses both *lockall* and *lock* simultaneously from different origin processes. In this case, as discussed in Section 4.5.1, since the *lock* calls are redirected to internal overlapping windows by Casper, one process of the application might end up acquiring a *lockall* epoch while another process of the same application acquires an exclusive-mode *lock* epoch on the same window (we note that the *lockall* epoch is shared-mode only and does not have an exclusive-mode equivalent). This situation is obviously incorrect and can cause data corruption.

To avoid this, Casper internally converts the *lockall* epoch to a series of locks to all ghost processes. Doing so ensures that any accesses are correctly protected by the MPI implementation. Arguably, this solution can add some performance overhead since it serializes lock acquisition. However, most MPI implementations delay lock acquisition until an actual operation is issued to that target, so this might not be much of a concern in practice.

4.5.5 Memory Ordering Consistency

Since Casper allows multiple processes (including the user application process and potentially multiple ghost processes) to access the same memory region, there may be a potential for memory ordering consistency issues. Specifically, without appropriate memory ordering consistency calls, a compiler or processor

```
P0:                     P1:                     G handler:

lock(EXCLUSIVE, P1);
/* redirected to G */
put(x, P1);
unlock(P1);                                     update x;
                        lock(EXCLUSIVE, P1);
                        load x
                        ...
```

Figure 4.9: Concurrent data access between a user and ghost processes.

hardware can freely reorder instructions that do not have data dependencies to improve performance. Of these, compiler reordering is less of an issue since most compilers are conservative, even at high optimization levels, and do not reorder instructions across function calls. Since all MPI RMA operations are function calls, compiler instruction reordering is not a direct concern for us. Hardware architecture reordering, on the other hand, is a concern. Almost every architecture available today permits some level of instruction reordering. Some architectures, such as x86, provide total store ordering (TSO) making such reordering less likely and restricted to fewer instruction patterns, while some architectures, such as Alpha, permit almost all possible reorderings. Thus, for portability, we need to assume that any reordering is possible by the hardware.

Luckily, the instruction reordering concerns of Casper are almost identical to the instruction reordering concerns of MPI RMA, in general, and would anyway need to be addressed by the MPI implementation. For instance, consider the two examples shown in Figures 4.9 and 4.10. In the example shown in Figure 4.9, the load instruction of P1 could be reordered to occur before the lock call since they are independent of each other, resulting in a wrong result of the load because the update on process 'G' may happen later. However, this problem is no different from the traditional RMA model when P0 accesses P1's memory directly (e.g., through shared memory), and the MPI implementation would need to do a memory barrier anyway. Note that this behavior would still be true even if the RMA operation is performed through two-sided communication (over shared memory or over a network) since any form of communication would eventually need to perform a memory barrier to ensure that the data is correctly received.

In the second example shown in Figure 4.10, the update instruction on G0 may also be reordered, hence resulting in a wrong result of the read operation on G1. Again, an MPI implementation would anyway need to do a memory barrier within a flush call to guarantee that the operation is finished on the target process, which means it should have already been reflected in memory.

```
P0:                        G0 handler:  G1 handler:

lock(EXCLUSIVE, P2);
/* redirected to G0 */
put(x, P2);
flush(P2);                 update x;
/* redirected to G1 */
get(x, P2);
unlock(P2);                                read x;
```

Figure 4.10: Concurrent data access between two ghost processes.

Table 4.1: MPI RMA implementations.

MPI Implementation	HW-supported OP	Asynchronous Progress
Cray MPI (regular)	NONE	Thread
Cray MPI (DMMAP)	Contiguous PUT/GET	Interrupt
Mvapich (IB netmod)	Contiguous PUT/GET	Thread
MPICH-SHM	all	NONE

4.6 Experimental Environment

We evaluate Casper on two platforms: the NERSC Edison Cray XC30 supercomputer [1] and the Argonne Fusion cluster [2]. We used these two platforms to demonstrate the impact of varying levels of hardware support for RMA operations as listed in Table 4.1. Specifically, Cray MPI (version 6.3.1) can be executed in two modes: `regular` or `DMAPP-based`. The `regular` version executes all RMA operations in software with asynchronous progress possible through a background thread. The `DMAPP` version executes contiguous PUTs and GETs in hardware, but ACCUMULATEs and noncontiguous operations are executed in software with asynchronous progress through interrupts. On the Fusion platform, we used MVAPICH. MVAPICH (version 2.0rc1[3]) implements contiguous PUT/GET operations in hardware, while using software active messages for ACCUMULATEs and noncontiguous operations with asynchronous progress through a background thread.

We expect Casper to improve asynchronous progress in the cases where RMA operations are implemented as *software active messages* and to perform as well as the original MPI implementation when hardware direct RMA is used.

[1] https://www.nersc.gov/users/computational-systems/edison/configuration/

[2] http://www.lcrc.anl.gov/about/fusion

[3] We had to fix a bug in MVAPICH to allow for true hardware-based RMA for PUT and GET.

4.7 Microbenchmarks Evaluation

In this section, we evaluate Casper by using several microbenchmarks focusing on following three major aspects: (1) analysis of the overheads caused by Casper complex design for guaranteeing correctness; (2) the improvement of asynchronous progress with comparison to other asynchronous progress approaches (an interrupt-based asynchronous progress called DMAPP and MPICH asynchronous thread on Cray X30; and MPICH asynchronous thread on InfiniBand cluster); (3) discussion of load balancing performance optimization.

4.7.1 Overhead Analysis

In this section, we measure two overheads caused by Casper: (1) window allocation and (2) Fence and PSCW.

4.7.1.1 Window allocation.

As discussed in Section 4.4.2, Casper internally creates additional overlapping windows in order to manage lock permissions when a ghost process supports multiple user processes. These can cause performance overhead. However, the amount of overhead can be controlled by setting the info argument `epoch_type` to tell Casper which epoch types are used by the application. Accordingly Casper can decide which internal windows it needs to create. Figure 4.11(a) shows the overhead of `MPI_Win_allocate` on a user process with varying total numbers of processes on a single node of Cray XC30. When no info hints are passed (default `epoch_type` is "fence,pscw,lockall,lock"), Casper can experience substantial performance cost in window creation time. When `epoch_type` is set to "lock," Casper does not have to create the additional window for active target and lockall communication, thus improving performance a little; but the cost is still considerable because Casper has to create one window for every user process on that node. When `epoch_type` is set to "lockall" or "fence" (or any other value that does not include "lock"), Casper has to create just one additional internal window, thus reducing the cost substantially, although the cost is still more than twice that of original MPI.

4.7.1.2 Fence and PSCW.

The second major overhead occurs because of the conversion of fence and PSCW to passive-target epochs and the additional synchronization and memory consistency associated with it. We measure these overheads by using two interconnected processes on Cray XC30. The fence experiment performs `fence--accumulate--fence` on the first process and `fence--fence` on the other, with the first passing the `MPI_MODE_NOPRECEDE` assert and the second fence passing the `MPI_MODE_NOSUCCEED` assert. The PSCW experiment performs `start--accumulate--complete` and `post--wait` on the two processes. Figure 4.11(b) shows the execution time of our experiments on the first process. While the overhead is large (100–200%) for a small number of operations, as the number of operations issued increases, this cost gets amortized and disappears.

4.7.1.3 Self Lock.

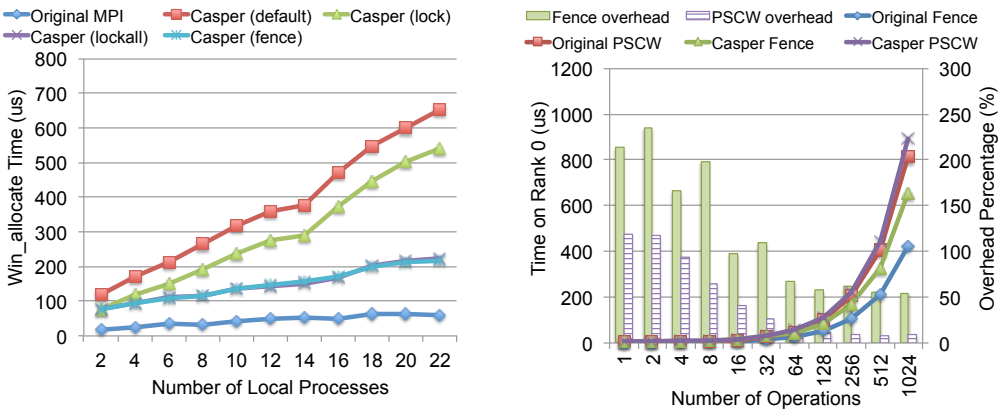
As we have described in Section 4.5.2, Casper issues a `get-flush` and a second *self-lock* for guaranteeing the lock permission correctness and memory consistency, resulting in additional overhead. Two user hints could reduce above overhead: `MPI_MODE_NOCHECK` assert eliminates the first step but still requires the second one for memory consistency of local load/store; `no_local_load_store` info (Casper(NO_LS)) eliminates both steps because neither lock permission nor local memory consistency need to be maintained in this case. Figure 4.11(d) compares the overhead of self lock on both Cray XC30 and on a shared node of Fusion cluster using MPICH-SHM. We measure a simple `lock(self)-put(self)-unlock(self)` microbenchmark with one application process. As shown in this figure, the default self lock always produces the heaviest overhead, 1 μs on Fusion and 1.4 μs on Cray respectively. On Fusion node, the lock with `no_local_load_store` performs the smallest overhead; the `NOCHECK` does not increase overhead much because the additional `get-flush` does not have much overhead on a shared memory node in which RMA operations are performed as direct RMA but the overhead of memory barrier becomes dominant. However, we get different trend on Cray, because Cray MPI still performs RMA operations as remote AM in shared memory node if the window is created by `MPI_Win_create` which degrades local RMA performance and we translate these RMA operations to the process itself if lock is acquired (as in default lock and `NOCHECK`) as a workaround. This is the reason why `NOCHECK` assert delivers the most performance improvement.

4.7.1.4 RMA operation segmentation

The third overhead of Casper comes from the RMA operation segmentation. As we have discussed in Section 4.5.3, when multiple ghost processes exist in the system, we need static binding for lock permission correctness. *Static Segment Binding* is one of the solutions to overcome the permission issue, but with additional overhead. We demonstrate such overhead using a simple microbenchmark with two interconnected processes. Every process first allocates a large window which will be divided to several segments and bound to different local ghosts; then when rank 0 issues a large ACCUMULATE to rank 1, such operation will be divided to multiple operations if the data contains multiple segments. We specify the window size is 1024 count of double, thus it will be divided to two 512 count of double segments with 2 ghost processes, four 256 count of double segments with 4 ghost processes, and eight 128 count of double segments with 8 ghost processes. As shown in Figure 4.11(c), when ACCUMULATE size is 128, which is always smaller than the segment size, there is not much overhead with increasing of ghosts. However, for a 256 and 512 count of double ACCUMULATE, significant overhead occurs with increasing of ghosts processes because more operations have to be produced internally.

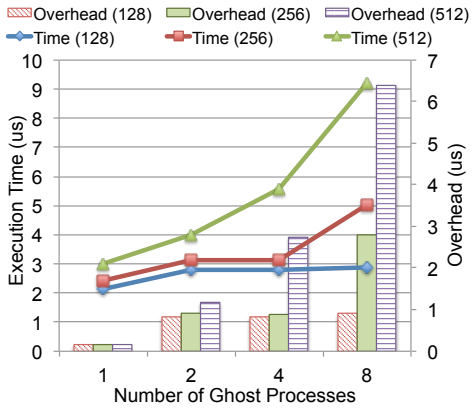
4.7.2 Asynchronous Progress

In the section, we demonstrate the asynchronous progress improvements achieved in various scenarios.

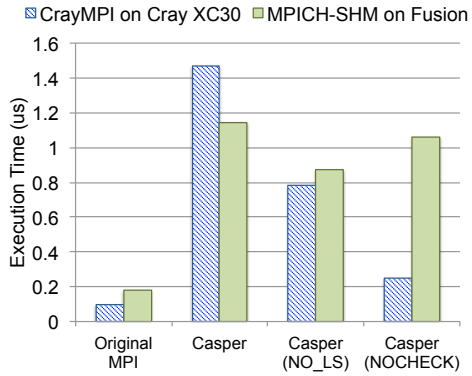


(a) Window allocation overhead.

(b) Fence and PSCW overhead



(c) RMA operation segmentation



(d) Local lock

Figure 4.11: Overhead analysis.

4.7.2.1 Different Synchronization Modes

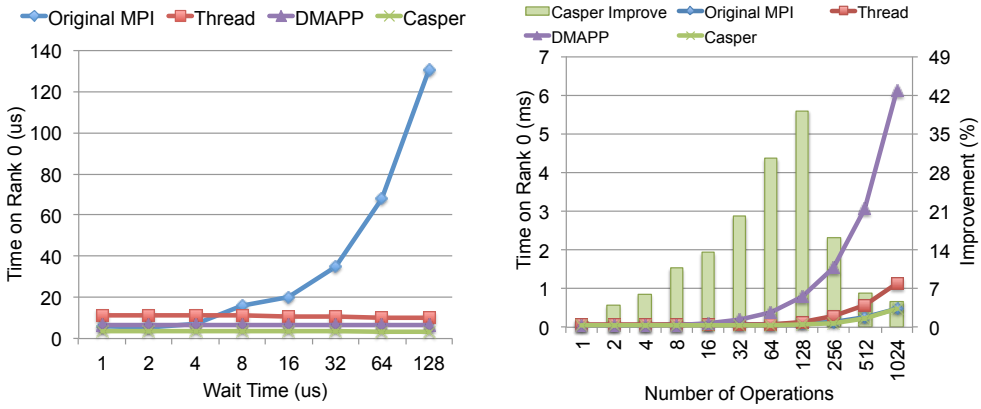
Our first experiment demonstrates the improvement of computation and communication overlap in passive and active-target modes using Casper. Two interconnected processes are used in each mode. In the passive-target mode, one process issues `lockall-accumulate-unlockall` to another process while that process is blocking in computation. Figure 4.13(a) shows the results on the Cray XC30. As expected, with the original MPI the execution time on the origin increases with wait time on the target, which means that the origin is blocked by the computation on the target. All asynchronous progress approaches relieve this issue. We note, however, that both the DMAPP and thread approaches have more overhead than Casper does.

The overhead with using the MPICH asynchronous thread comes from the expensive thread-multiple safety and lock contention. DMAPP-based asynchronous progress, however, does not involve thread-multiple safety and also wakes up background threads only when a message arrives. Therefore, to analyze the reason for this overhead, we performed a test in which one process does `lockall-accumulate-unlockall` and the other process does a `dgemm` computation.

In the active-target mode, since both fence and PSCW require internal synchronization in Casper, the origin has to wait for the completion of the epoch

```
MPI_Win_lock_all(win);
if (rank == 0){
    MPI_Accumulate(...);
    MPI_Win_flush(1, win);
}else{
    while(MPI_Wtime() - start < WAIT_TIME);
    MPI_Test(...);
}
MPI_Win_unlock_all(win);
```

Figure 4.12: Overlap improvement microbenchmark.



(a) Passive-target RMA

(b) Fence RMA

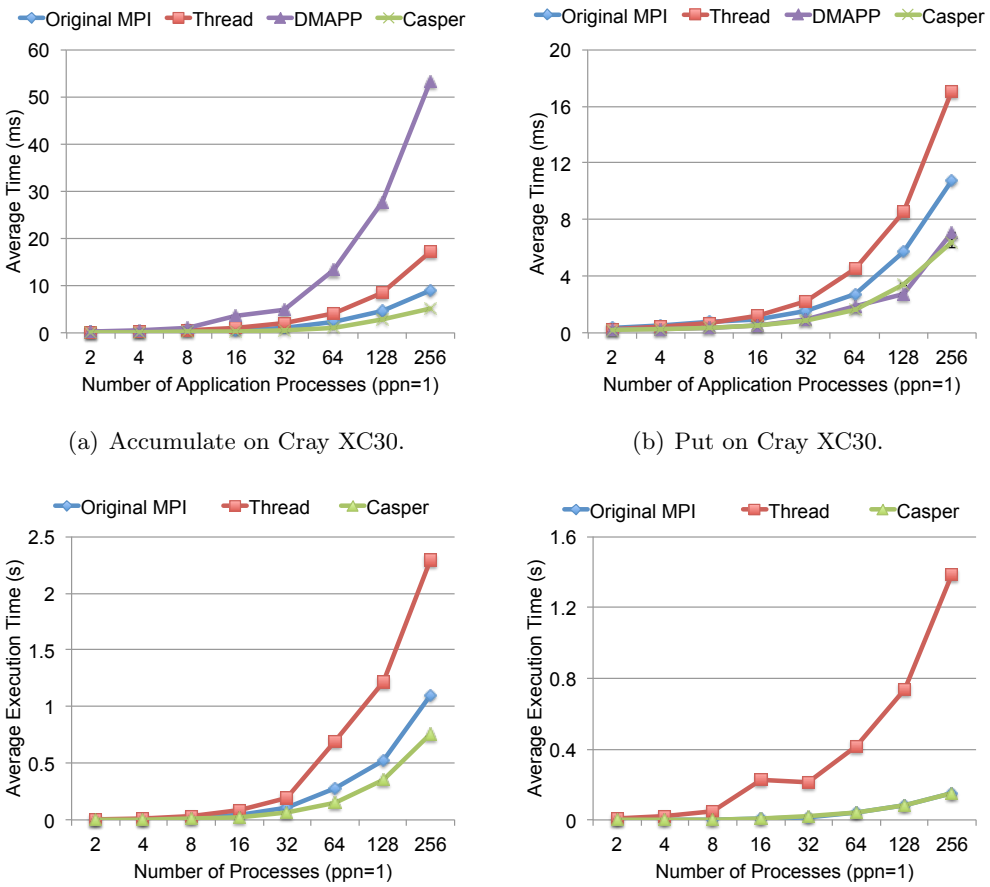
Figure 4.13: Overlap improvement using two interconnected processes on Cray XC30.

on the target. Thus, in an experiment similar to that for the passive mode, we measured the time for `fence-accumulate-fence` on one process while another process performs `fence-100` μ`s busy waiting-fence` as shown in Figure4.13(b). We notice that when a small number of operations are issued during fence, asynchronous progress is beneficial. But when the communication takes more time than the delay on the target, which is the maximum time Casper can overlap (larger than 128 in the figure), the percentage improvement decreases, as expected. PSCW follows a similar trend. Both DMAPP and thread asynchronous progress show significant overhead compared with that of the original MPI execution.

4.7.2.2 Different RMA implementations

The second experiment focuses on the scalability of asynchronous progress with different RMA implementations. In this experiment every process communicates with all the other processes in a `communication-computation-communication` pattern. We use one RMA operation (size of a double) in the first communication, 100 μs of computation, and ten RMA operations (each size double) in the second communication.

On Cray XC30, we use one process per node and scale the number of nodes



(a) Accumulate on Cray XC30.

(b) Put on Cray XC30.

(c) Accumulate on Fusion using MVAPICH.

(d) Put on Fusion using MVAPICH.

Figure 4.14: Asynchronous progress on different platforms.

for both ACCUMULATE and PUT, as shown in Figures 4.14(a) and 4.14(b). We note that DMAPP enables direct RMA for PUT/GET with basic datatypes in Cray MPI, but it involves interrupts for ACCUMULATE operations. Consequently, Casper outperforms the other approaches for ACCUMULATE, while achieving the same performance as that of DMAPP for PUT/GET. The thread asynchronous progress is always expensive and even worse than that of the original MPI when a large number of processes are communicating.

On the Fusion cluster, we compared Casper with MVAPICH by also using one process per node. Figure 4.14(c) indicates that Casper improves asynchronous progress for ACCUMULATE, which is still implemented with software active messages in MVAPICH. The thread asynchronous progress again shows significant overhead. We also measured the performance of PUT/GET operations; as expected, the performance of Casper was identical to that of original MPI since these operations are implemented directly in hardware. The performance numbers are not shown here because of space limitations.

4.7.3 Performance Optimization

Our third set of microbenchmarks focuses on the different load-balancing optimizations discussed in Section 4.5.3.

4.7.3.1 Static Rank Binding

Figure 4.15 shows our measurements with static rank binding on Cray X30. In the first experiment (Figure 4.15(a)) we show the static rank binding with increasing number of processes when each process sends one accumulate message (size of double) to every other process in the system. We use 16 processes per node and evaluate Casper with up to 8 ghost processes on each node. Our results indicate that two ghost processes are sufficient when up to 32 processes communicate; when more processes communicate, however, configurations with larger numbers of ghost processes tend to perform better. The reason is that the number of incoming RMA operations increases with more processes, thus requiring more ghost processes computing to keep up.

Figure 4.15(b) shows a similar experiment but increases the number of accumulate operations while keeping the user process count constant at 32 (2 nodes with 16 processes each). The results show a trend similar to that of the previous experiment, with more ghost processes benefiting when the number of operations per process is larger than 8.

4.7.3.2 Static Segment Binding

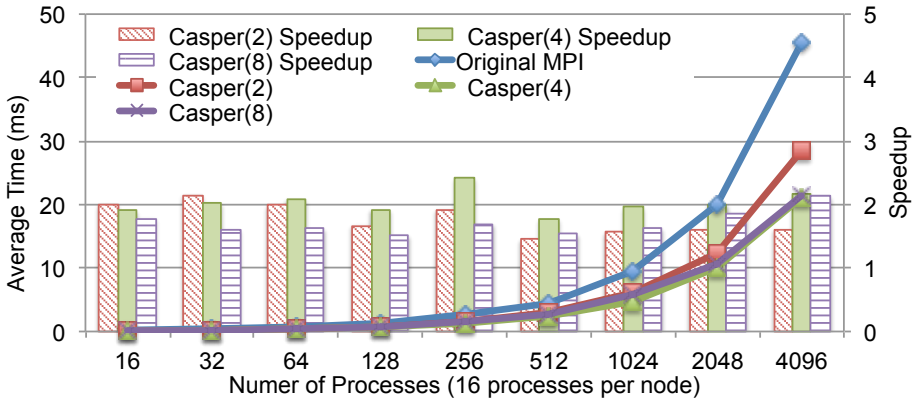
In this experiment we evaluate the performance of the static segment binding approach. Such an approach is expected to be especially beneficial when the application allocates uneven-sized windows and receives a large number of operations that need to be processed in software. Figure 4.15(c) demonstrates this pattern. We used 16 nodes with 16 processes and up to 8 ghost processes per node. The first process of every node allocates a 4-kilobyte window (512 count of double), while the others only allocate 16 bytes. Then each process performs a `lockall-accumulate-unlockall` pattern on all the other processes. We increase the number of ACCUMULATEs to each process whose local rank is 0 while issuing a single operation to other processes. As shown in the figure, performance improves with increasing numbers of ghosts, because the large window is divided into more segments and the communication issued to different segments is handled by different ghosts.

4.7.3.3 Dynamic Binding

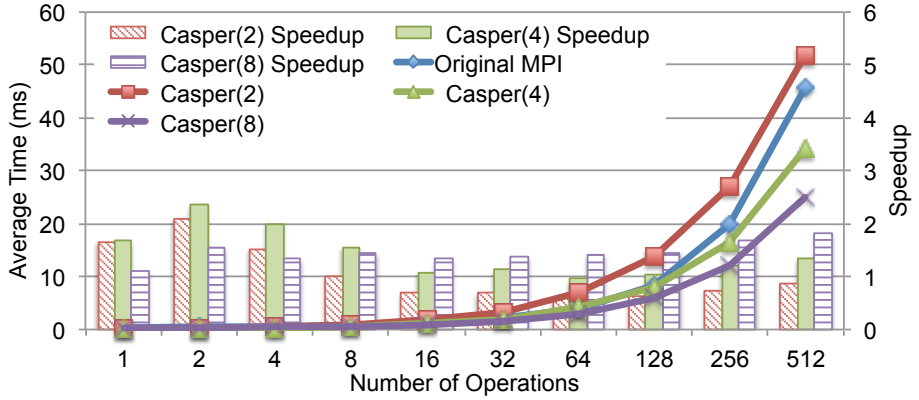
To test our dynamic binding approaches, we designed three microbenchmarks, all of which are executed on 16 nodes with 20 user processes and 4 ghost processes per node.

Figure 4.16(a) shows the results of an experiment in which all processes perform a `lockall-put-unlockall` pattern to all the other processes, but only the first rank of each node receives an increasing number of PUT operations (varied on the x-axis of the graph), while the others receive only one PUT operation. Our random load balancing simply chooses the ghost processes in the order of its local rank for each target process. Thus, all the PUT operations are always equally distributed to the ghosts on each node achieving much better performance that with static binding.

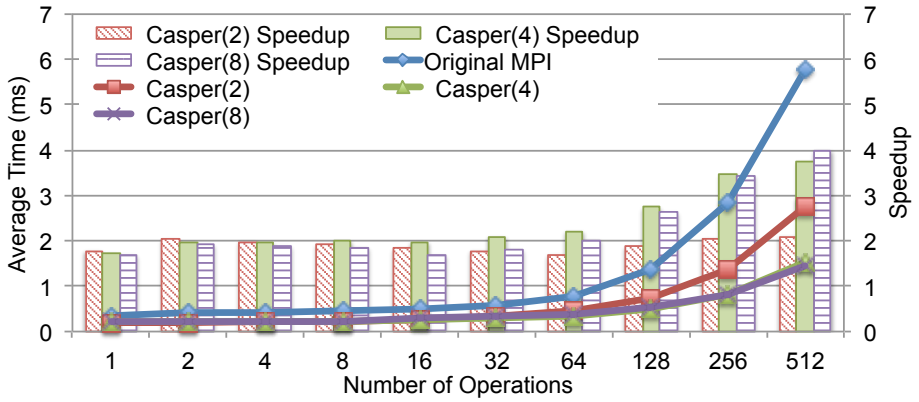
Figure 4.16(b) uses a variant of the previous experiment in which each process performs an uneven `lockall-accumulate-put-unlockall` pattern to all other



(a) Static Rank Binding: Increasing Processes.



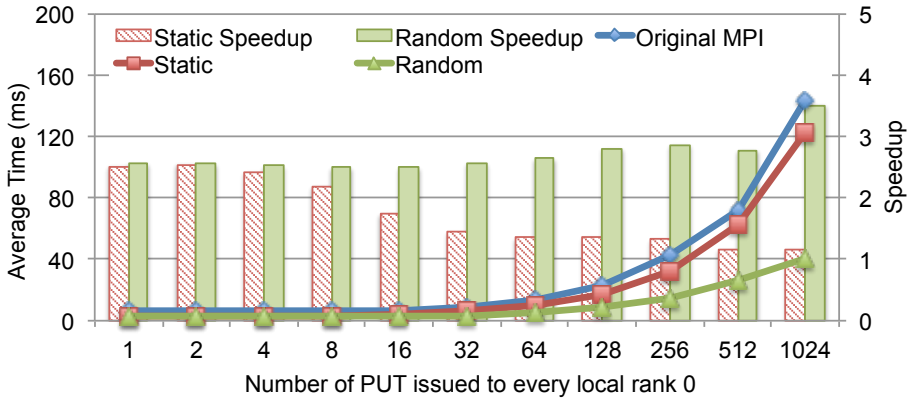
(b) Static Rank Binding: Increasing operations.



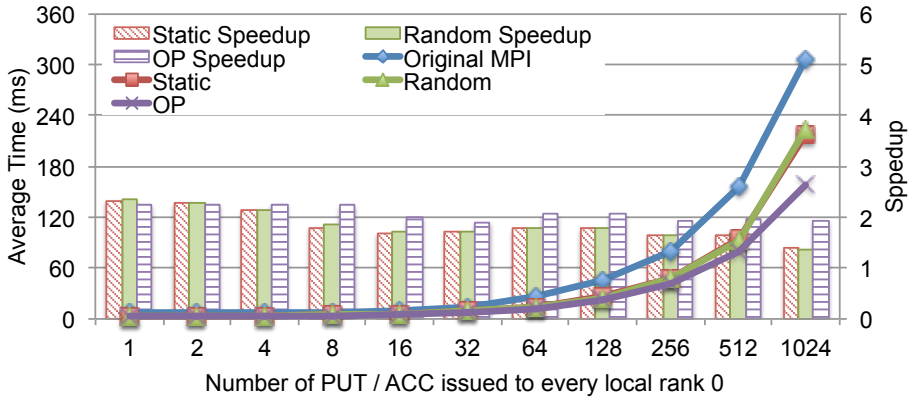
(c) Static Segment Binding: Uneven Window Size

Figure 4.15: Load balancing in static binding on Cray XC30.

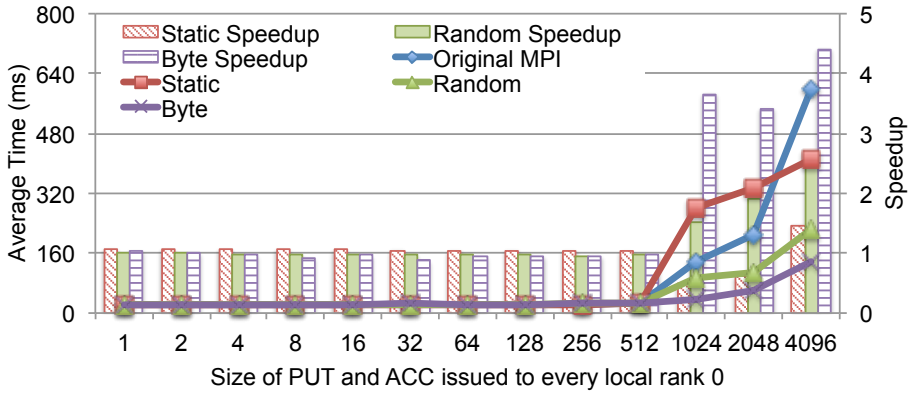
processes. In this case, random load balancing arbitrarily picks the ghost process for each PUT operation but sends all ACCUMULATE operations to the same ghost process (in order to maintain ordering and atomicity guarantees). Thus, the ghost process that is handling both ACCUMULATE and PUT operations would end up having to handle more operations than would the other ghost processes. Our "operation-counting" approach, on the other hand, keeps track of which ghost process has been issued how many operations and balances the operations appropriately, thus allowing it to achieve better performance than the



(a) Random: Uneven Number of Put.



(b) OP-counting: Uneven Number of Put/ACC.



(c) Byte-counting: Uneven Size of Put/ACC.

Figure 4.16: Dynamic load balancing on Cray XC30.

random approach does.

Our third experiment, uses yet another variant of the previous experiments by varying the size of the operations while keeping the number of operations constant. Each process performs a `lockall-accumulate-put-unlockall` pattern, but only the processes whose local rank is 0 receive increasing sizes of PUTs and ACCUMULATEs (varied on the x-axis), while the others receive only one double PUT and accumulate. Figure 4.16(c) shows the results. As expected, neither random nor operation-counting algorithms can handle this case well, although

our "byte-counting" approach outperforms both of them.

4.7.3.4 Memory Locality Comparison

The last microbenchmark measures the overhead when ghost process is located in a different NUMA domain with the application process. We simply use two interconnected processes each with a ghost process on its node, one process issues `lock-100 accumulate-unlock` to the second process while the second one waits in MPI barrier. Figure 4.17 shows that significant overhead occurs on Cray XC30 when the ghost process is located in a different NUMA domain and accesses the memory of application process across domain. The overhead even increases with increasing size of operations, 120% overhead is produced when issuing 512 could of double(4097 Bytes) operation .

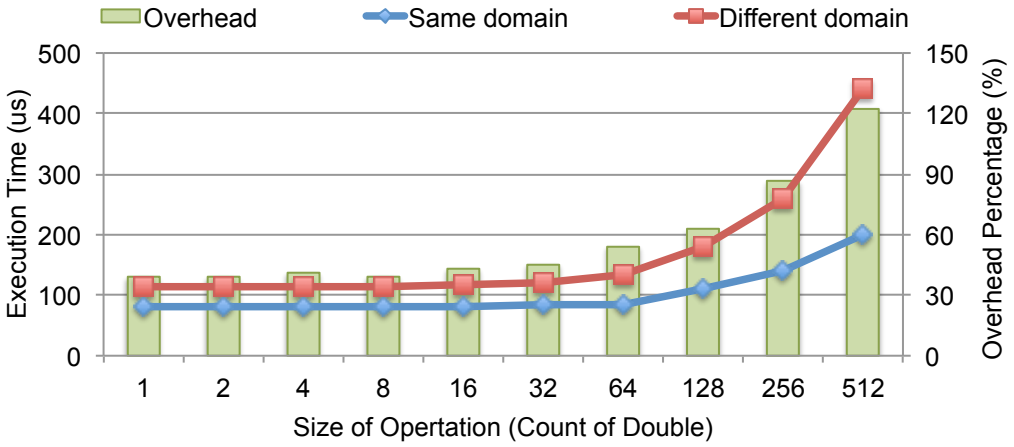


Figure 4.17: Comparison of two cases with different ghost process location.

4.8 NWChem Quantum Chemistry Application

NWChem [16] is a computational chemistry application suite offering many simulation capabilities, with providing extensive functionality [33] and excellent performance [8]. NWChem is developed on top of the Global Arrays [51] toolkit, which provides an abstraction of global shared array that hides the complexity of domain distribution across physical nodes. It is implemented on a number of platforms natively and as a portable implementation over MPI RMA [24].

The coupled cluster (CC) theory is one of the most popular approaches in quantum chemistry for solving electron correlation in atoms and molecules with arbitrary accuracy requirements. NWChem provides highly efficient parallel implementations for a variety of complicated CC methods through the Tensor Contraction Engine (TCE). The "gold standard" *coupled cluster with singles and doubles and perturbative triples* method, known as CCSD(T), is one of the most accurate CC methods applicable to large molecules to date. It is particularly useful for calculating accurate noncovalent interaction energies.

The `get-compute-update` mode is the generic code structure used in all the internal phases of NWChem, which performs large three-dimensional matrix–

```
Global Arrays : A, B, C
Local Buffers : a, b, c
for {each sub block in A, B} {
    GET a from A;
    GET b from B;
    COMPUTE c = a * b + c;
    UPDATE c to C;
    NXTASK;
}
```

Figure 4.18: Generic get-compute-update mode in NWChem.

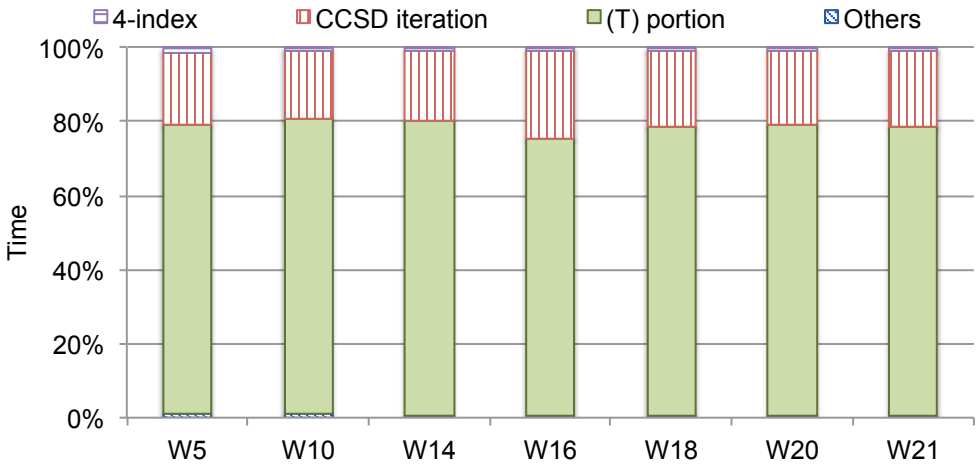


Figure 4.19: Analysis of CCSD(T) internal steps in varying W_n with pVDZ.

matrix multiplications. As demonstrated in Figure 4.18, in the `get-compute-update` mode, each processes first gets sub-domain a and b from global arrays that are located in the memory spaces of remote processes, then performs a local DGEMM and next it updates the result c back to the global memory, accumulating the values. The nxtask module is called after every sub-domain computation to decide the computing process for the next computation.

The CCSD(T) method performs a complex set of multidimensional array computations organized in three internal steps: four-index transformation, CCSD iteration, and the noniterative (T) portion. To understand the performance characteristics of CCSD(T), we compared the time consumed by each step on our experimental platform (see Section 4.6) for three water molecule $(H_2O)_n$ problems ($n = 5, 16, 21$, denoted as W_n) with double-zeta basis sets (pVDZ). As shown in Figure 4.19, the (T) portion consistently dominates the entire cost of CCSD(T) by close to 80%, and the CCSD iteration takes the other 20%; the four-index transformation and other internal steps represent less than 3% of the execution time.

To evaluate the capability of Casper, we use both Casper and the thread-based approaches to optimize the most time-consuming step: (T), a computation-intensive stage that follows the typical `get-compute-update` approach contain-

Table 4.2: Core deployment in NWChem evaluation on Cray XC30.

	Computing Cores	Async Cores
Original MPI	24	0
Casper	23	1
Thread (O)	24	24
Thread (D)	12	12

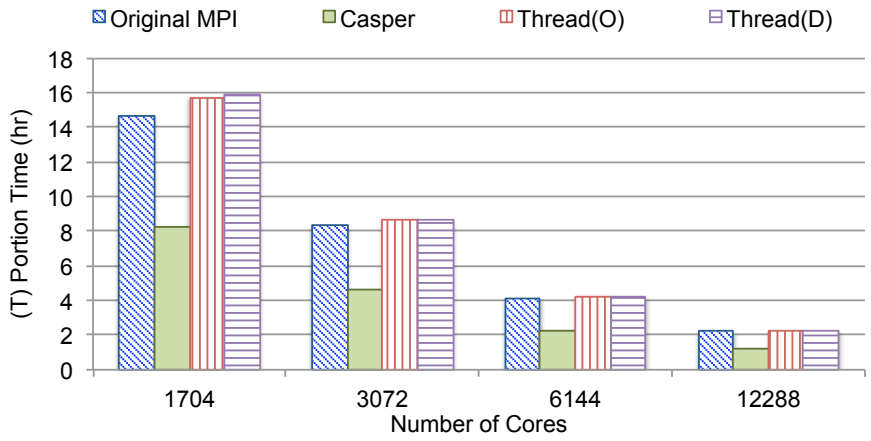
ing large matrix-matrix multiplication operations (`compute`) with numerous one-sided operations (`get`) and a reduce operations (`update`) at the end of computation.

Two molecules are considered in this experiment: a very large water cluster $(H_2O)_{21}$(denoted W_{21} for short), and C_{20} which is obtained from the NWChem QA test suite (`QA/tests/tce_c20_triplet`). For the water cluster, we used double-zeta basis sets (cc-pVDZ from the NWChem basis set library), which are reasonable for this class of problems. We compared Casper with both the original MPI and two thread-based approaches. The first approach employs oversubscribed cores (Thread (O)), where every thread and its MPI process execute on the same core; the second approach uses dedicated cores (Thread (D)), where threads and MPI processes are on separate cores. We used the same total number of cores in all approaches, some of which are dedicated to asynchronous ghost processes/threads as listed in Table 4.2.

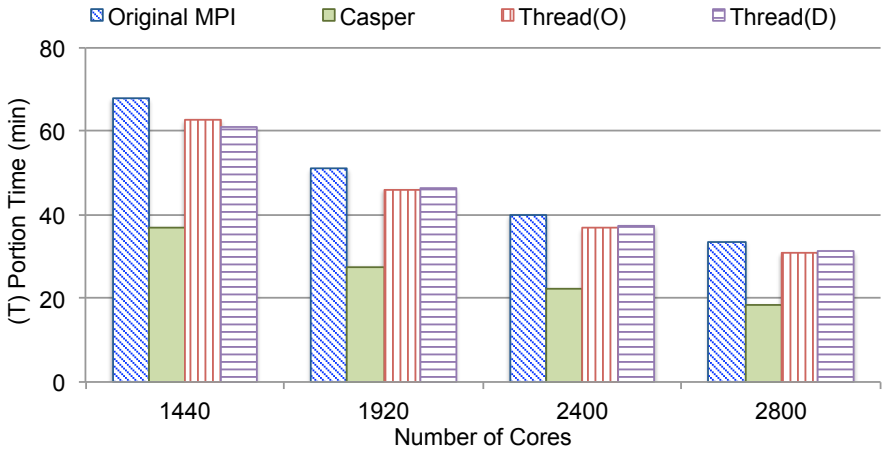
We first focus on the strong-scaling performance in large W_{21} (with pVDZ) problems by using a varying number of cores. Figures 4.20(a) show the execution time. Relative to the original version, Casper is almost twice as fast than the original MPI, whereas the thread-based approaches cannot improve performance and perform even slight worse than the original implementation. Similar trend has been shown in Figure 4.20(b), the CCSD(T) simulation in C_{20} (with pVDZ) problem. Casper with one ghost process consistently improves the performance of (T) portion by 2-folds, but the thread approaches cannot benefit much, showing similar execution time as the original version.

To determine the reason for these results, we measured separately the time consumed by the internal computing operations and that by the RMA communication (`get`) on 1704 cores and on 6144 cores. As shown in Figure 4.21, although both Casper and the thread-based approaches eliminate the delay in RMA communication with asynchronous progress on, the performance of the computation is negatively impacted. Since Casper spends only one core as a ghost process on each node, this degradation is minimized. With the thread-based approaches, however, performance becomes twice as bad because of appropriation of half of the computing cores in the dedicated approach (Thread(D)) and the inefficient core oversubscription in Thread(O) approach.

As the last observation, we also measured the weak scaling of (T) portion with varying water problems (W_n-pVDZ, where n = 5, 10, 14, 16, 18, 20, 21) and cores. As shown in Figure 4.22, similar to the trend we obtained in the strong scaling experiment, Casper consistently doubles the performance in all



(a) (T) portion for W_{21} with pVTZ.



(b) (T) portion for C_{20} with pVTZ

Figure 4.20: NWChem CCSD(T) Simulation on Cray XC30.

problem sizes, while the thread-based approaches impose a performance penalty with respect to the original MPI for all problem sizes because of inefficient core usage or oversubscription.

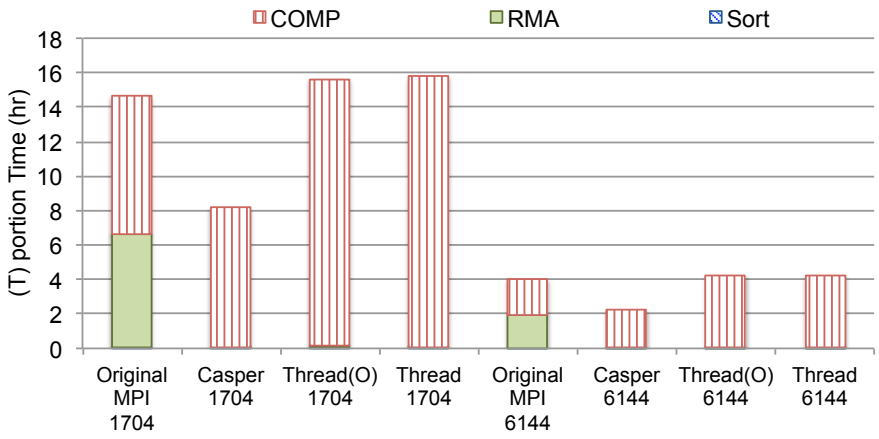


Figure 4.21: (T) portion profiling for W_{21} with pVDZ.

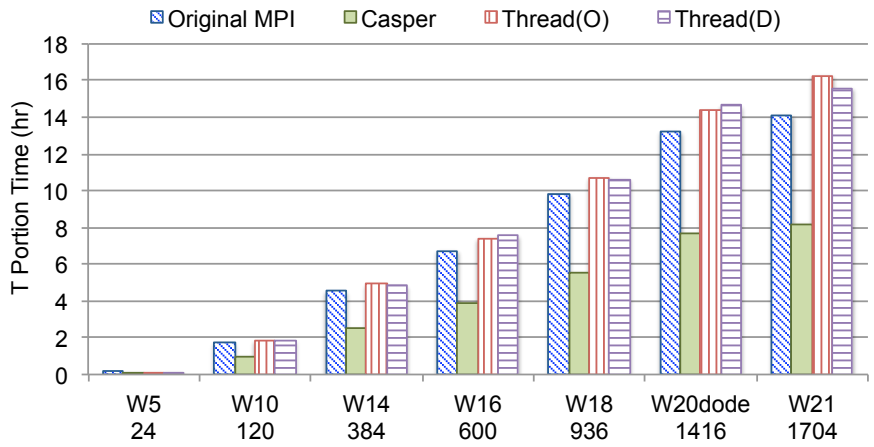


Figure 4.22: (T) portion in NWChem CCSD(T) for varying Wn with pVDZ.

Chapter 5

Dynamic Adaptable Asynchronous Progress

Advances in high end computing systems provide scientists the capability to solve more complex and large-scale problems. Many of those complex scientific problems always require integration of multiple fundamental solvers and algorithms into application execution. To provide highly optimized performance for various scientific applications, researchers have studied and categorized the performance characteristics for large sets of typical applications that are always designed as rigid communication and computation pattern, and proposed corresponding optimization for each kind of applications. However, such static method can rarely benefit those complex applications that require a collection of multiple algorithms, since each of these implementations can performs very different characteristics of communication and computation. Thus it is challenging to find a generic optimization for such kind of applications.

Chapter 4 presented a process-based asynchronous progress approach for supporting one-sided data movements on multi- and many-core architectures by keeping aside a few cores as "ghost process" and dedicating them to help asynchronous progress for user processes through memory mapping and operation redirection techniques. The straightforward design, however, may not deliver optimal performance in multi-phases application and even raise up the concern of load imbalanced communication. In this Chapter, we study this issue by deeply analyze the internal construction of multiple phases in NWChem application as the case study, and propose dynamic adaptation strategies that help Casper better support the challenging multi-phases applications.

The rest of the chapter is organized as follows: Section 5.1 first discuss the limitation of static Casper in general multi-phases applications. Then Section 5.2 present our solution—two dynamic adaptation approaches—for addressing the issue, and Section 5.3 gives the detailed design for each approach. After Section 5.5 evaluates the adaptation approaches using several microbenchmarks, we demonstrate the performance improvement in large quantum chemistry application NWChem in Section 5.6, with deep analysis of the performance characteristics for NWChem's multiple internal phases, and performance comparison with traditional static solution.

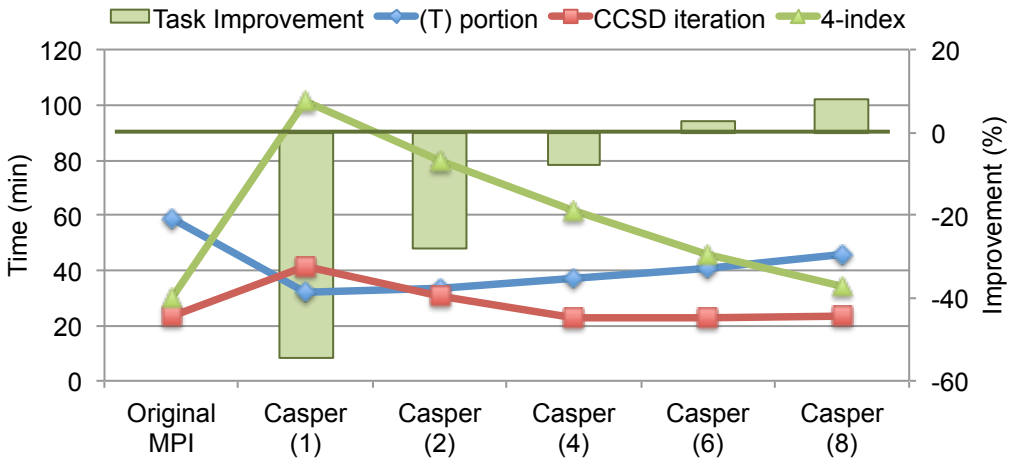


Figure 5.1: Trade-off in NWChem CCSD(T) task for Tet-pVDZ on 240 cores

5.1 Limitation in Static Casper

In Chapter 4, we has proposed "Casper," a process-based asynchronous progress solution for MPI one-sided applications [61]. Although networks such as InfiniBand has provided hardware-handled PUT/GET operations on contiguous data, the other operations, such ACCUMULATION on non-contiguous 3D subarray, still require the MPI software to make progress on the target side to ensure the completion of operations. Such limitation, obviously, can result in arbitrary long delay if heavy computation is involved on the target process. The central idea of Casper is to dedicate a small user-specified number of cores on a multi- or many-core environment as background "ghost processes." Casper transparently intercepts all RMA operations to the user processes through PMPI and redirects them to the ghost processes by utilizing an internal shared window, which is internally initialized at every user window allocation call.

The static operation redirection allows Casper to provide efficient asynchronous progress for software-handled RMA operations without effect the performance of any hardware-handled operations (e.g., contiguous PUT/GET operations on RDMA supported network). However, such static redirection may not be sufficient for some large applications that always compose of multiple internal phases with different proportion of communication. That is, in communication-sparse phases, the ability of asynchronous progress is important because arbitrary long delay can happen when the target processes are so busy in computing that cannot make MPI progress; in communication-intensive phases, however, the asynchronous progress may not be necessary because the target processes can frequently make MPI calls and thus handle the operations issued to them as targets by themselves. Furthermore, when the amount of RMA operations becomes large, redirecting operations to a few of ghost processes may even result in performance degradation in communication, since those operations were originally distributed to many user target processes (the number of user process is always much larger than the amount of ghost processes). Figure 5.1 demonstrates the performance we have observed in NWChem CCSD(T) simulation for the Tetracene ($C_{18}H_{12}$)

molecular. None of the asynchronous progress approaches can benefit all internal phases, the computation-intensive (T) portion gets the largest improvement when using 2 ghost processes (Casper(2)), however, the other phases shows degradation since they are dominated by communication. Trade off has to be done to for the whole execution. In Section 5.6 we will analysis the detail of such inefficiency in the NWChem application.

To better understand the trade off between the asynchronous progress and the load balance of communication, we break down the MPI implementation in a simple RMA communication running on two nodes with two user processes and one ghost process on each node. Each user process on the first node (O1 and O2) communicates with one of the processes on the second node (T1 and T2) following the pattern `lock-ACCUMUALTE-flush-computation-unlock`, while T1 and T2 are just performing computation and waiting for the completion of operations from O1 and O2 respectively. The internal communication processing for the ACCUMULATE operations can be separated as three steps in most MPI implementation: issuing operation on the origin side, handling operation on the target side, and local completing after the internal acknowledgment (i.e., PUT, ACC) or data back (i.e., GET) on the origin side.

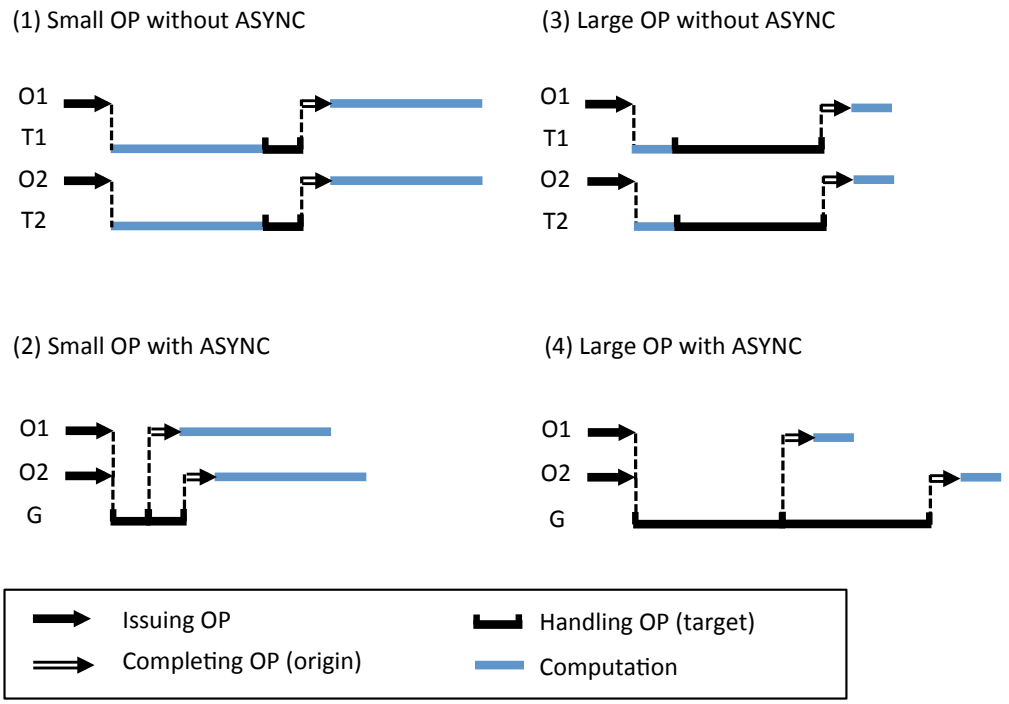


Figure 5.2: Asynchronous Progress and Load Balance.

Figure 5.2 compares the cost of each step with and without asynchronous progress for both small and large operations. As shown in case (1), for small operations, since the operations handling cost is very smaller thus the delay of the operation is almost caused by the computation on the target processes; when asynchronous progress is provided, as shown in case (2), the operations are quickly handled by the ghost process (G) instead of the target processes without any delay caused by the computation, however, the ghost process has to handle all the operations issued to the processes on its node (T1 and T2). Nevertheless,

it is still more efficient than (1) since the the total cost of the handling processing on G is much smaller than the delay caused by the computation on T1 and T2. When the operation size or the number of operations becomes large on every user process, the overhead of the operation handling on the target side is also significantly increased. Thus it can happen that, the waiting time for an operation is longer within Casper on some processes (i.e., O2) if the overhead of the operation handling on the ghost process G is more expensive than the delay caused by user computation. Case (3) and (4) demonstrate such cases.

To distinguish with the adaptable version we propose from the next section, we use static Casper to indicate the basic version with static configuration of asynchronous progress in the remainder of this paper.

5.2 Solution

In this chapter, we propose two dynamic adaptation approaches for Casper asynchronous progress in multi-phases applications: a user-guided approach and a transparent self-profiling based approach, with providing strict correctness per MPI semantics. The adaptation approaches allow application to dynamically redirect all RMA communication to the ghost processes for computation-intensive phase that requiring efficient asynchronous progress, with avoiding inefficient redirection for communication-intensive phase.

5.3 Dynamic Adaptable Asynchronous Progress

A dynamic adaptation mechanism is necessary in order to avoid the inefficient asynchronous progress as described in the above section. In this section, we introduce the design and implementation of Casper's dynamic adaptation.

The notion of asynchronous progress adaptation is to dynamically change the internal target of RMA operations. The operations are sent to ghost process as the target when asynchronous progress is enabled, and to the original user target process when asynchronous progress is disabled. The notion is straightforward, however, a simple implementation can break the ordering and atomicity guarantee for ACCUMULATE-like operations on a single window which is provided in MPI. For example, for two concurrent ACCUMULATE operations, if one is issued to the ghost process but the other is issued to the user target process, then both operations can be concurrently performed on two processes, and consequently resulting in undefined result.

In this section, we introduce two adaptation approaches we have carefully designed for ensuring the correctness per MPI semantics. The first approach relies on the `guidance` from user. It allows user to enable or disable asynchronous progress for each particular internal phase during application execution by passing user hint at the beginning of the target phase. This approach is straightforward and accurate, however, it requires the users have sufficient understanding of the characteristics and code construction of the application. The second approach we have studied is based on the idea of `self-profiling`. We insert profiling code in every MPI call through PMPI to automatically track the change of communication frequency during execution, thus allowing transparent adaptation of

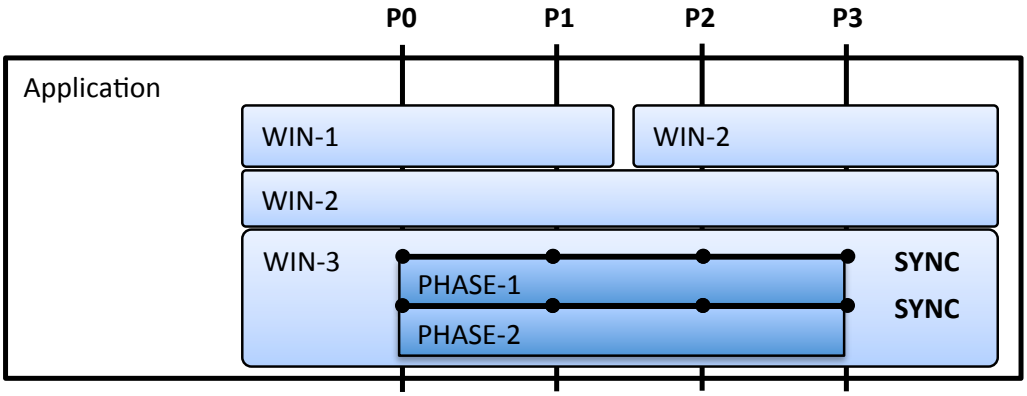


Figure 5.3: Three levels of granularity in RMA application

asynchronous progress. In the following parts of this section, we describe the design of both approaches separately.

5.3.1 User-Guided Adaptation

To ensure any concurrent operations on a given window are always issued to the same internal target, every user process must collectively update the asynchronous progress inside Casper. That is, the change of asynchronous progress configuration must be done either at window allocation time, or at any collective synchronization that guarantees the completion of all outstanding operations on all processes involved in the window (i.e., `MPI_Win_fence`). Thus we define three levels of adaptation granularity in the user-guided approach as shown in Figure 5.3: (1) the global configuration for the entire execution, (2) per-window configuration for the communication performed on a particular window from window allocation to window free, and (3) the per synchronization configuration for controlling a particular phase during two synchronization calls. We describe the detailed design for each level as follows.

Global Configuration: Before execution, user can specify the value of environment variable `CSP_ASYNC_CONFIG` to `ON` or `OFF` (`ON` by default) to enable or disable the asynchronous progress for the entire execution. This value can be overwritten by setting through finer granularity configuration. We note that, the cores dedicated to ghost processes are always kept aside from the user application, even when the asynchronous progress is disabled.

Per Window Configuration: Whenever user processes allocate a window, user can pass the info hint `async_config` (`ON` or `OFF`) to overwrite the configuration of asynchronous progress for this window.

Per Collective Synchronization Configuration: During the communication of a given window, MPI provides several synchronization calls (i.e., fence, lock-unlock, post-start-complete-wait) to ensure the completion of outstanding RMA operations and synchronization between processes. Due to the limitation of ordering and atomicity semantics as we have already discussed, only the fence synchronization allows the internal change of target redirection in Casper, since it is collectively called by all processes in the window and the return from a fence

call guarantees the completion of all outstanding operations on all processes. The window allocation call can be considered as a special collective synchronization.

However, in passive-target mode, there is no synchronization call provides such functionality. We propose a new info hint `symmetric` with value `true` or `false` that can be passed to an active window via the `MPI_Win_set_info` call. This hint is required to be the same value on all processes of the window. The `symmetric=true` info means all processes have locally finished all the outstanding operations on the window and arrived at this call. As an example of its usage, this hint can be passed after a `flush_all-barrier` synchronization performed on all processes. This information allows Casper to provide the chance for safe adaptation similar as that in fence.

5.3.2 Transparent Profiling based Adaptation

Although the user-guided approach provides simple and accurate adaptation, it only benefits a few application experts who have sufficient knowledge in both application implementation and MPI programming. To provide comprehensive support for any application users, a transparent solution becomes necessary. Thus we also studied a self-profiling based approach, which consists of a prediction step determining the needs of asynchronous progress comparing to the load balance of communication for every single user process, and a synchronization step that exchanges the predicting results among all involved processes.

The prediction step replies on the local profiling technique that tracks the proportion of communication and computation in the recent execution phase. Specifically, we estimate the asynchronous progress is more important for the coming execution on a process if the profiling result shows its computation has become so intensive that can significantly degrade the performance of communication from other processes due to lack of asynchronous progress, and hence enable its asynchronous progress. Conversely, we estimate the asynchronous progress is unnecessary on a process if profiling result indicates intensive communication that can increase the progress made by the process itself and additionally result in higher risk of load imbalance, and consequently we disable the asynchronous progress.

The second synchronization step exchanges the predicted results among all user processes, thus allowing every process to gather the configuration of asynchronous progress on every target process for its future communication. This step can be set in every window collective synchronization call to hide the requirement of extra collective synchronization. It also guarantees the ordering and atomicity for ACCUMULATE-like operations since all the processes concurrently and consistently change their local information for any target process. However, such design may result in failure of adaptation if the timing of synchronization in application does not fit the change of communication characteristics. For example, the computation can become heavy after window created and there may not have any window synchronization that allows us to perform the second synchronization step. This issue does not exist in the user-guided approach since the user can appropriately set the hint before the change. Thus we also investigated a more flexible approach for the synchronization step that can address this issue

for PUT/GET operations which do not require strict ordering and atomicity, by offloading to the background ghost processes.

In the rest of this section, we introduce the detailed design of the self-profiling based adaptation by decoupling into following three aspects: the self-profiling base prediction, the user synchronization for all RMA operations, and the ghost-offloaded synchronization for PUT/GET operations.

5.3.2.1 Self-Profiling based Prediction

To predict the needs of asynchronous progress and the communication workload in the next period of execution, we automatically determine the `communication frequency` for every process in the last recent execution and assume the next period follows the same trend. A high frequency means that the process is frequently making MPI call thus is able to handle the receiving operations by itself; a low frequency means the process rarely performs MPI communication thus does not have chance to handle the incoming operations, consequently requiring ghost process to provide asynchronous progress.

We insert timer in every MPI function through PMPI to automatically measure the communication time for any given period of execution from time T_{n-1} to T_n, and use Equation 5.1 to evaluate its communication frequency $Freq(T_n)$ at time T_n:

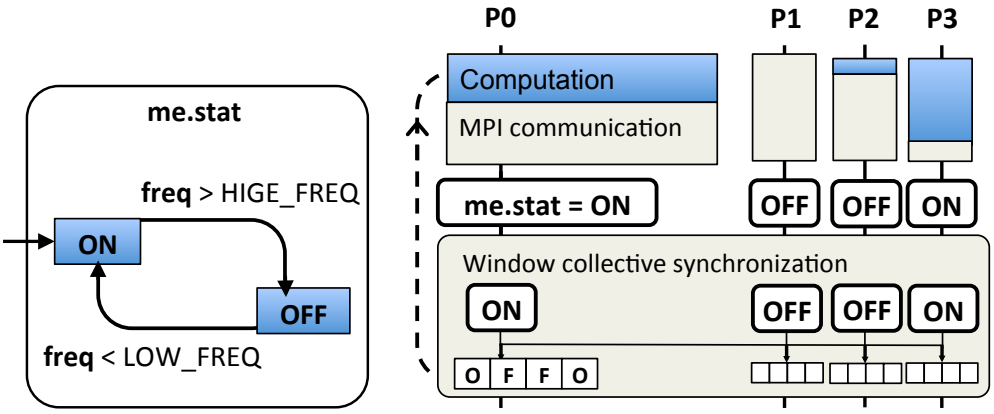
$$Freq(T_n) = \frac{T_{comm_{T_n}}}{T_n - T_{n-1}} \tag{5.1}$$

in which the $T_{comm_{T_n}}$ is the total execution time of MPI calls performed on that process during the period from time T_{n-1} till T_n.

After every process evaluated the communication frequency, it updates the local asynchronous status to indicate its needs of asynchronous progress. As shown in Figure 5.4(a), we define a two-level threshold, `HIGH_FREQ` and `LOW_FREQ`, to ensure a relatively stable status change. The asynchronous status of every user process is `ON` by default; if the observed frequency is higher than the `HIGH_FREQ`, then we set the status to `OFF`; conversely, if the observed frequency becomes lower than the `LOW_FREQ`, then we set the status back to `ON`.

Although the prediction cost is so small that can be ignored in real applications, we still use a threshold `PREDICT_INT` to control the interval between twice local prediction instead of performing it in every MPI call. That is, the local prediction based on $Freq(T_n)$ is only performed when interval $(T_{n-1} - T_n)$ becomes larger than `PREDICT_INT`.

We note that the time inside MPI calls can be taken in two ways: (1) block waiting before message arrive, and (2) MPI internal instructions (i.e., issuing operation and completing operation on the origin side, and handling incoming operations on the target side as shown in Figure5.2), however, the user process can get chance to make MPI progress only in the block waiting time. In addition, we have also discussed that the severe load imbalance issue on ghost processes is due to heavy overhead of the operation handing step. Unfortunately, we cannot distinguish the overhead of each internal communication step of MPI implementations through PMPI, thus we have to predict based on the entire communication time which can reduce the accuracy of prediction.



(a) Status prediction (b) Prediction and Synchronization.

Figure 5.4: Self-profiling based Adaptation

5.3.2.2 User-handled Synchronization

In the user-handled synchronization mode, every user process holds an array for each window to maintain the status of asynchronous progress on all of its target processes. Casper can internally redirect an RMA operation to either the ghost process or the user process according to the target's status. If the status of a target is `ON`, then all the operations issued to that target are redirected to the corresponding ghost process, otherwise they are directly issued to the original user target process.

Figure 5.4(b) demonstrates the adaptation work-flow composing of local prediction step and the user-handled synchronization step. The prediction step simply updates the local status on every single process; then in every window-wide collective synchronization call (i.e., window allocation, fence or symmetric window info setting), processes collectively exchange the latest status with each other and update the local target array, thus ensuring any two operations issued to the same target on the same window must be both sent to the ghost process, or both sent to the user target process. Consequently, the semantics correctness of any RMA communication is guaranteed similar as the user-guided approach.

5.3.2.3 Ghost-offloaded Synchronization for PUT/GET

A more dynamic synchronization is designed for the PUT/GET operations which do not require strict ordering and atomicity. This mode offloads the global information synchronization to the background ghost processes through a two-level cache mechanism as demonstrated in Figure 5.5. Thus the adaptation can happen without relying on any collective synchronization call in user application. This design can be decoupled into following five pieces:

Two-Level Caches: Every user process allocates the level-1 cache from its local memory for fast query in frequent PUT/GET operations; a shared window is allocated among user processes and the first ghost process on every node as the level-2 cache at MPI initialization time for serving the global synchronization. Both the level-1 and level-2 cache is an array that stores the status of all the user processes. The offset of the status for a given user process is consistent on all

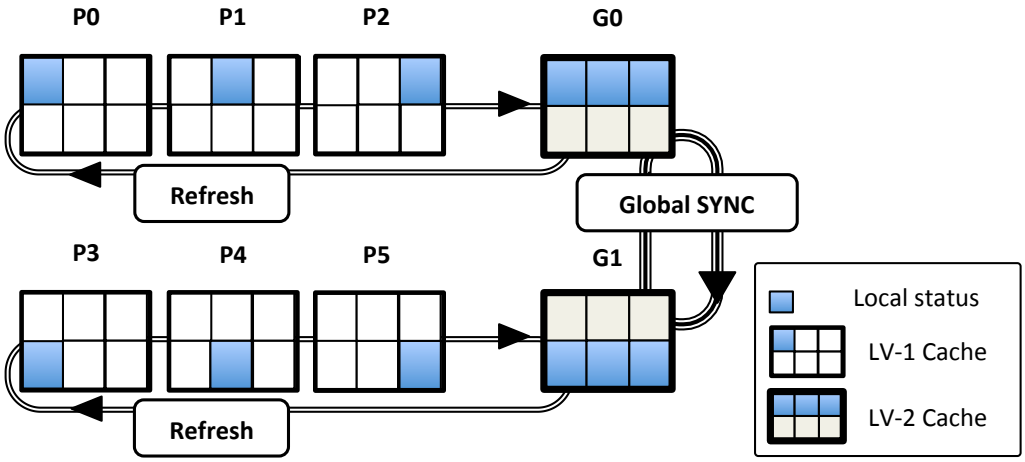


Figure 5.5: Ghost-offloaded synchronization.

processes. Figure 5.5 shows an example of this design with six user processes on two nodes. Thus every cache array contains six integer elements, in which the elements from offset 0 to 5 are responsible for the status from P0 to P5 respectively.

User Local Update: Every user process updates its local status in the prediction step as described in Section 5.3.2.1. It is immediately updated to the corresponding element in the level-1 and level-2 caches if the value is different from the previous status (e.g., changed from `ON` to `OFF`). When updating to the level-2 cache, `MPI_Accumulate` operation is used instead of direct write in order to ensure per-element atomicity when accessing the shared window.

Ghost-offloaded Global Synchronization: Regardless of the execution on user processes, the global status synchronization is performed by the first ghost process on every node at an interval that can be set through the environment variable `GSYNC_INT`. Every ghost process sends out the status of its local user processes (shown as blue blocks in the level-2 cache in Figure 5.5) and receives from others. This operation can be simply implemented by using `MPI_Allgather`. After the completion of a global synchronization, the ghost process needs to notify all the user processes on its node that the level-2 cache has become `DIRTY`, thus the user processes can update its local level-1 cache by reading newer data from the level-2 cache (called "refresh"). The dirty notification can be implemented using `MPI_Ibcast` and the refresh operation must utilize `MPI_Get_accumulate` operation for per basic element atomicity.

Target Local Query: In every PUT/GET operation, the user process first queries the asynchronous progress status of the user target process by directly reading from the local level-1 cache. Similar as that in the user-handled synchronization, if the target's status is `ON` then the user process redirects the operation to the corresponding ghost process for the target, otherwise directly sends to the original user target. The frequent query operation does not involve any heavy overhead since it is just a load of integer from the local memory.

Performance Consideration: Obviously, the ghost-offloaded approach can generate additional overhead, since extra synchronization has been involved. It

is important to avoid unnecessary synchronization. For example, after a user-handled synchronization (e.g., in user window allocation call), the status has already been synchronized, thus the upcoming global synchronization on the ghost processes can be skipped.

We note that the ghost-offloaded approach can only provide user processes a mostly recent status on other processes. It guarantees neither the consistency between a local status and its cached version on remote processes, nor the consistency among any remotely cached status for the same process. This is because, user processes may or may not refresh the level-1 cache concurrently; moreover, a user process can perform the next prediction right after the ghost-offloaded synchronization. Thus we should only apply this adaptation to the PUT/GET operations, ACCUMULATE-like operations can only be adapted by using the user-handled synchronization.

5.3.2.4 Impact on Hardware-handled Operations

The self-profiling based adaptation is relying on the assumption that all RMA operations are handled in the MPI stack that requiring the target process to make progress, thus the asynchronous progress needs to be enabled if we predict low communication frequency on the target process. However, such notion is not held for the hardware handled operations (i.e., contiguous PUT/GET in Cray MPI DMAPP mode) that do not require any software progress on the target process. Nevertheless, we do not distinguish the hardware handled operations in the adaptation, since it does not impact on the performance. That is, in the case with low communication frequency, all the hardware handled operations are also redirected to the ghost process but should delivering the same performance as distributed to multiple user processes on the same node; similarly, it the case with high communication frequency that the asynchronous progress will be disabled, the performance of hardware handled operations should still show no performance difference since they do not rely on the asynchronous progress in software.

5.4 Experimental Environment

All the experiments in this paper are executed on the NERSC Edison Cray XC30 supercomputer the same as the experiments for static Casper (see Section 4.6). For the software environment, a newer version is for all the experiments: the Intel compiler Composer XE 2015.1.133 and the Cray MPI 7.2.1. For NWChem evaluation, we use NWChem version 6.3 with MKL 11.2.1 as the external math library. We note that the Cray MPI contains a `regular mode` and a `DMAPP mode`. The `regular mode` executes all RMA operations in software with asynchronous progress possible through background threads (by setting the `MPICH_ASYNC_PROGRESS` environment variable); the `DMAPP mode` executes contiguous PUT and GET operations in hardware and provides interrupt-based asynchronous progress for other operations (i.e., accumulate, noncontiguous operations). Since in the previous chapter, we have observed significant overhead of frequent interrupts in the `DMAPP mode` that can be difficult to benefit real applications, we only compare the adaptation approaches with the `regular mode`.

5.5 Microbenchmarks

In this section, we analyze the performance of the adaptation approaches by utilizing several microbenchmarks from the aspects of overhead, self-profiling based prediction, the limitation in static Casper, and the adaptation improvement respectively.

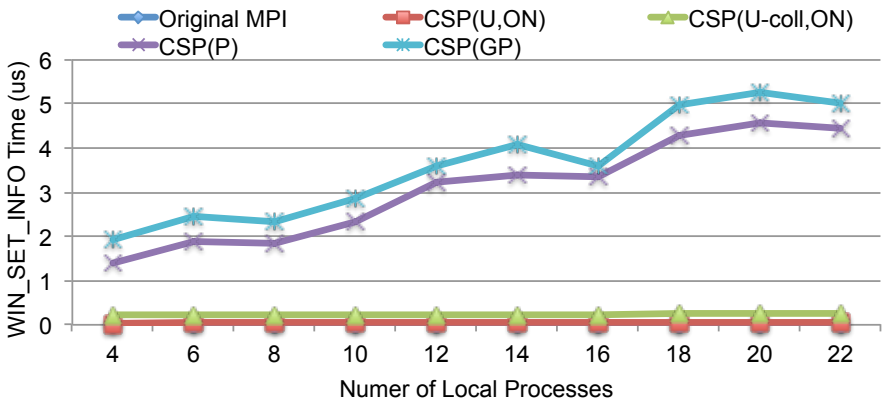
5.5.1 Overhead Analysis

We first evaluate the overhead of the user-guided approach and the self-profiling based approaches for each window collective synchronization call on a single node with one ghost process and increasing number of local processes. For user-guided approaches that only insert hint at window allocation, the names are abbreviated to CSP(U, {ON, OFF}) in which ON or OFF is the hint passed through window info; similarly, CSP(U-coll, {ON, OFF}) is denoted for the user-guided modes that also insert hint in fence or the MPI_WIN_SET_INFO; for self-profiling based approaches, CSP(P) is denoted for the mode that only allows user-handled synchronization, and CSP(GP) is for the mode that also allows ghost-offloaded synchronization for PUT/GET operations.

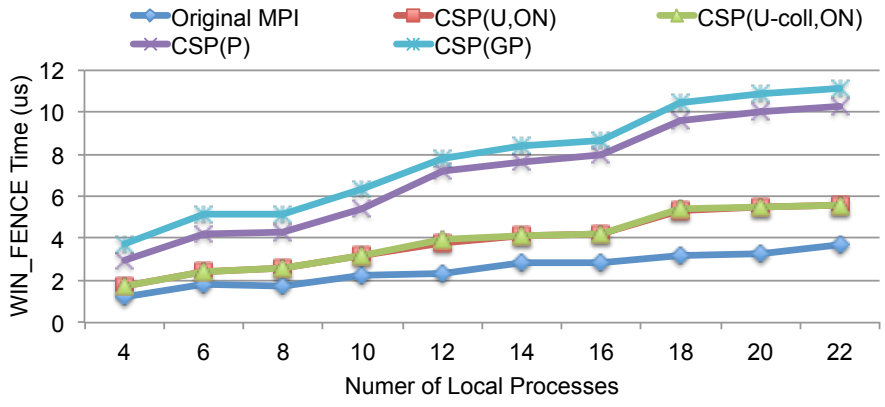
Figure 5.6(a) shows the overhead of `MPI_WIN_ALLOCATE`. When the asynchronous progress is enabled, Casper always experience substantial cost in the window allocation time, since we have to create internal windows and exchange internal information globally. This can be shown as the gap between CSP(U, ON) and the original MPI; the performance of CSP(U, OFF) is very close to that of the original MPI, because we do not perform additional processing and return immediately when the asynchronous progress in the window is disabled; the CSP(U-coll, ON) and CSP(P) approaches perform the same processing as that for CSP(U, ON) thus deliver the same performance. We note that the CSP(GP) mode always updates the local ghost cache, however, it is not significant since the overhead of GET_ACCUMUALTE on shared memory is too small. We omit CSP(U-coll, OFF) in this graph, because it performs the same processing that in CSP(U-coll, ON).

Figure 5.6(b) compares the overhead of each approach at fence call. For CSP(U, ON), it does not perform adaptation in fence, the additional cost comparing to original MPI is because of the passive-mode translation in our Casper implementation as already discussed in our previous work; the CSP(U-coll, ON) approach only updates local window since the user hint is consistent on all processes, thus showing similar cost as that of CSP(U, ON); the CSP(P) approach, however, need to perform the user synchronization that exchanges the status of local asynchronous progress among processes; the CSP(GP) approach needs to also update the data in every local ghost cache after user synchronization, thus consistently showing close to 1μs overhead comparing to CSP(P).

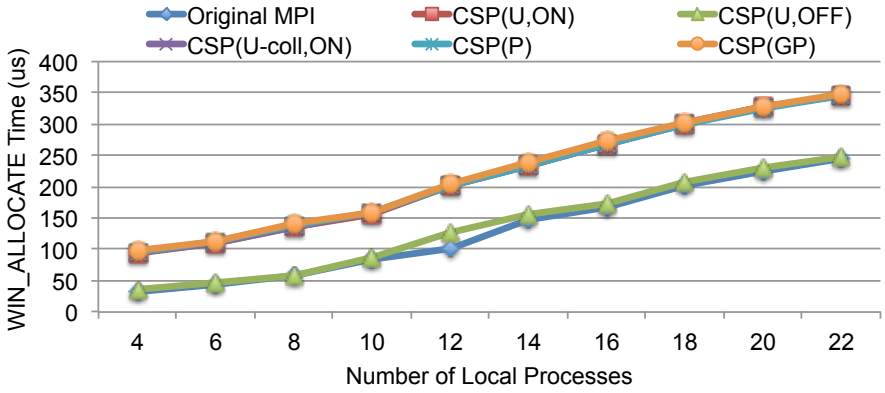
Figure 5.6(c) compares the overhead of each approach at window info setting with symmetric hint. Similar as the fence call, both CSP(U, ON) and CSP(U-coll, ON) do not involve any additional communication; the self-profiling based approaches have to always perform additional all-to-all communication, thus showing increasing overhead with increase of processes. The additional 1μs overhead of CSP(GP) comparing to CSP(P) is the same as we have explained in fence.



(a) Window allocation



(b) Fence



(c) Window info setting with symmetric

Figure 5.6: Adaptation Overhead at Window Collective Synchronization.

5.5.2 Self-Profiling based Prediction

As we have discussed in Section 5.1, the communication load imbalance issue happens when the overhead of operation handling on the ghost processes becomes more expensive than the delay on user processes caused by user computation. The overhead of operation handling is related to three factors: the number of operations, the number of user processes that bound to a single ghost process, and the data size of every operation. In this section, we measure the performance

speedup delivered by Casper with asynchronous progress (shown as ON) comparing to the result with disabling the asynchronous progress (shown as OFF), with increasing proportion of communication in the above ways, and try to quantify the communication frequency on our experiment platform.

5.5.2.1 Increasing Number of Operations

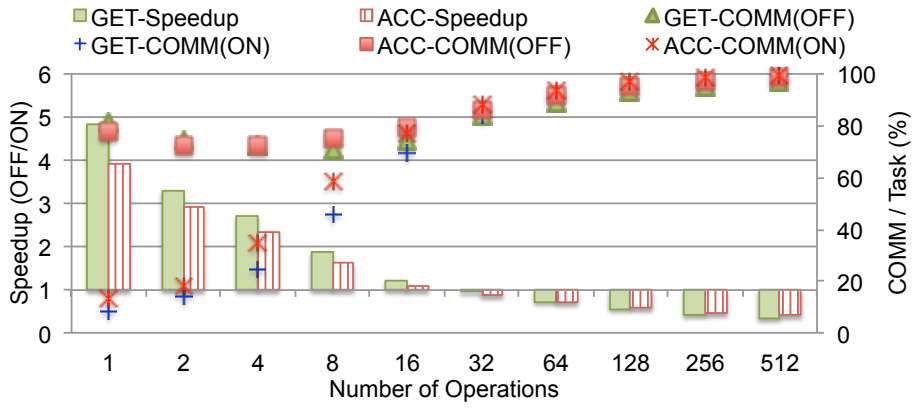
We first measures the experiment of an all-to-all RMA communication with increasing number of operations on 2 nodes with 22 user processes and 2 ghost processes per node. Every user process performs `lockall-N[RMA-flush]-computation-1[RMA-flush]-unlockall` pattern to all the other processes, in which `N` indicates the number of operations and the computation is demonstrated as 5000μ busy waiting. Every RMA operation involves a contiguous buffer with 8 double elements on the origin side, and using subarray datatype to abstract a $2\times2\times2$ three-dimension double submatrix within $8\times8\times8$ window region as the target datatype. Figure 5.7(a) shows the measurements for GET and ACCUMULATE operations separately. Within asynchronous progress, the GET and ACCUMUALTE operations get up to 4.85x and 3.9x speedup when the number of operations is smaller than 16. With increasing operations, the speedup of asynchronous progress gradually reduces and becomes negative when more than 32 operations have been issued, since the ghost process handles too many operations that even heavier than the delay due to computation. This graph also shows the the proportion of communication comparing to the task execution time with reducing speedup. We notice that when the asynchronous progress is disabled (GET-COMM(OFF) and ACC-COMM(OFF)), the value of proportion is much larger than that with asynchronous progress (GET-COMM(ON) and ACC-COMM(ON)) when the number of operations is small, this is because the communication overhead is significantly higher in the disabled case due to lack of asynchronous progress.

5.5.2.2 Increasing Number of Processes

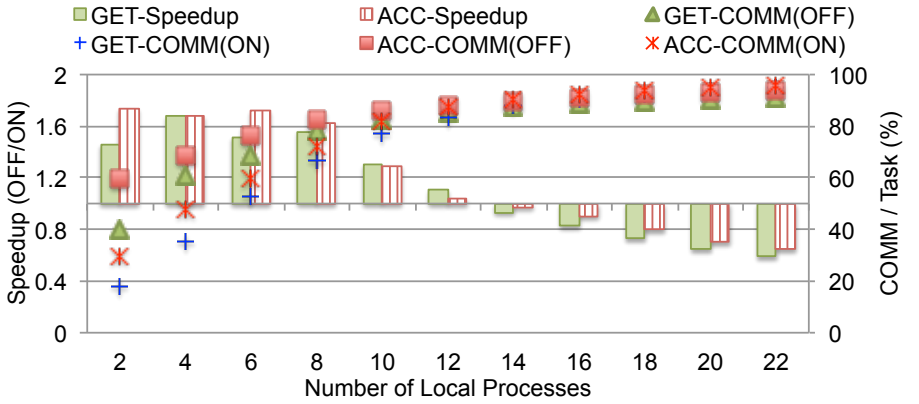
Figure 5.7(c) shows a similar experiment, but scales the number of local processes while keeping the number of GET operations at 100. Similar as the trend in the previous experiment, the speedup of asynchronous progress reduces with increasing number of local processes, and eventually becomes negative at 14 local processes in both GET and ACCUMULATE experiments. The reason is similar as the first experiment, the overhead caused by too heavy workload on the ghost processes in the asynchronous progress enabled case is increasing with larger number of local processes, since every ghost process has to serve more user processes.

5.5.2.3 Increasing Size of Operations

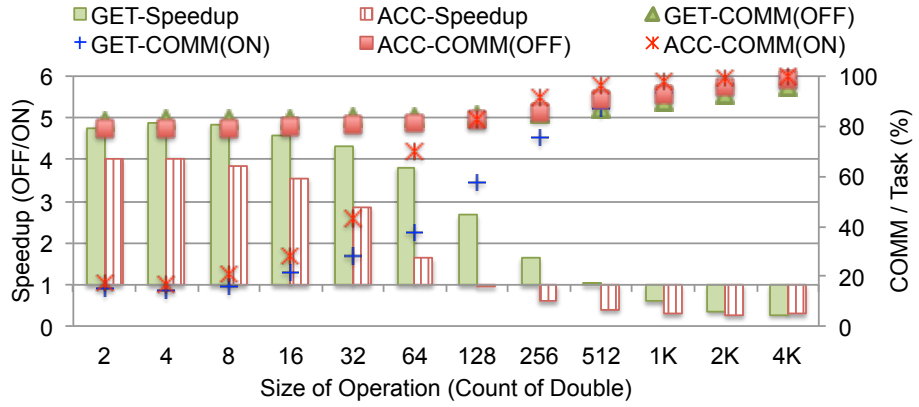
Our third experiment uses yet another variant of the first experiment by varying the length of X-axis (x) in the $x\times2\times2$ submatrix within the $(x+8)\times8\times8$ window region, while keeping the number of operations at 1 for GET and ACCUMULATE operations separately. Every process issues RMA operations following the same pattern with 5000μ busy waiting. Figure 5.7(b) shows the results. This case,



(a) Increasing number of operations



(b) Increasing number of processes



(c) Increasing operation size

Figure 5.7: Efficiency of Asynchronous Progress related to Communication Frequency.

Table 5.1: Profiling Overhead(us) of Single RMA Operation on BreadBoard

OP	$2 \times 2 \times 2$(double)		$128 \times 2 \times 2$(double)		$4096 \times 2 \times 2$(double)	
	Time	H-OP	Time	H-OP	Time	H-OP
ACC	8.6	2.7	48.0	33.9	1341.4	1271.6
PUT	7.7	1.8	28.0	14.4	515.5	447.1
GET	7.7	0.7	28.6	14.2	515.1	445.3

however, shows different trend of speedup among the different operation types. The ACCUMULATE operation starts negative speedup of asynchronous progress after the data size increased to $128 \times 2 \times 2$ counts of double; the GET operation also shows similar trend with increasing operation size, however, the negative speedup starts from $1024 \times 2 \times 2$.

To figure out the difference between these operation types, we measured the overhead of the internal operation handling step on the target side (H-OP) of MPICH on the BreadBoard cluster [1]. As shown in Table 5.1, within the same three-dimension datatype using two inter-connected processes. Only the first process issues operations to the second. The ACCUMUALTE operation always shows heavier overhead of the target handling step than the others because the ACCUMULATE is handled as firstly unpacking received data to a temporary buffer and then accumulating with the original window data following the non-contiguous data structure. Thus its speedup related to the load imbalance is always lower than the GET and PUT which only involve non-contiguous issuing and unpacking received data to non-contiguous window location in the handling step respectively. With increasing X-axis of the target datatype, this gap becomes even larger, thus we can observe the degradation in ACCUMULATE at $128 \times 2 \times 2$ counts of double where GET still shows 2.67-fold speedup. We do not show the results of PUT in the above graphs for simplicity reason since it is performance is very close to GET.

5.5.3 Limitation of Static Casper

Our third set of experiments focus on the limitation of static Casper by comparing two microbenchmarks that demonstrate a typical computation-intensive phase and a communication-intensive phase respectively on 8 nodes. In each experiment, we utilize 24 user processes on every node in the original MPI approach, and vary the number of ghost processes from 1 to 8 in static Casper, each of which serves 23, 22, 20, 18 and 16 user processes per node respectively. Every user process performs all-to-all communication following the pattern as shown in Figure 5.8. In the computation-dominated phase, we use a DGEMM computation with total size $96000 \times 96000 \times 96000$ ($500 \times 500 \times 500$ per process in the original MPI approach) in every iteration, and issues one `ACCUMULATE-flush` in each communication part ($NOP = 1$); in the communication-intensive phase, we set the total size of DGEMM to $192 \times 192 \times 192$ and issues 100 `ACCUMULATE-flush` in the first communication part ($NOP = 100$). Every RMA operation utilizes the same datatype as used in Section 5.5.2.1.

Figure 5.9(a) shows the result of the computation-intensive experiment. The task execution time is reduced from 42.3 seconds to 11.8 seconds when using static Casper with only one ghost process. This is because the communication in the original MPI is significantly degraded due to lack of asynchronous progress. However, with increasing number of ghost processes, the overhead of the DGEMM computation gradually increases because more and more computing cores are dedicated to asynchronous progress, which is necessary.

[1] We cannot access the source of Cray MPI thus cannot directly measure the internal overheads. However, the Cray MPI uses similar RMA implementation as that of MPICH, thus we expect similar trend of the internal overheads with increasing data size.

```
MPI_Barrier();

for(iter=0; iter<50; iter++) {
  for(dst=0; dst<nprocs; dst++) {
    for(i=0; i<NOP; i++) {
      MPI_Accumulate(...,dst,...);
      MPI_Win_flush(dst, win);
    }
  }

  /* DGEMM computation */

  for(dst=0; dst<nprocs; dst++) {
      MPI_Accumulate(...,dst,...);
      MPI_Win_flush(dst, win);
  }
}
```

Figure 5.8: communication/computation-intensive phase microbenchmark.

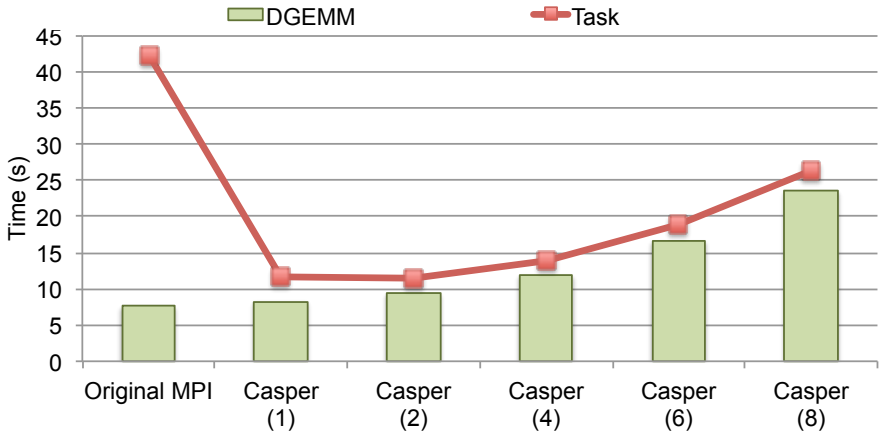
Figure 5.9(b) shows the result of the communication-intensive experiment. Different from the computation dominated case, the performance is 4x worse than the original MPI approach when only use one ghost process within static Casper because of load imbalance issue as we have demonstrated in the previous section. Within more ghost processes, this issue is gradually resolved and shows better performance than the original case from 6 ghost processes since we still have computation on every process.

Obviously, although the static Casper can benefit for any single phase of application by adjusting the number of ghost processes, it cannot achieve the optimal performance if an application contains both the computation-intensive phase and the communication dominated phase. In the next set of experiments, we will focus on such multi-phases case.

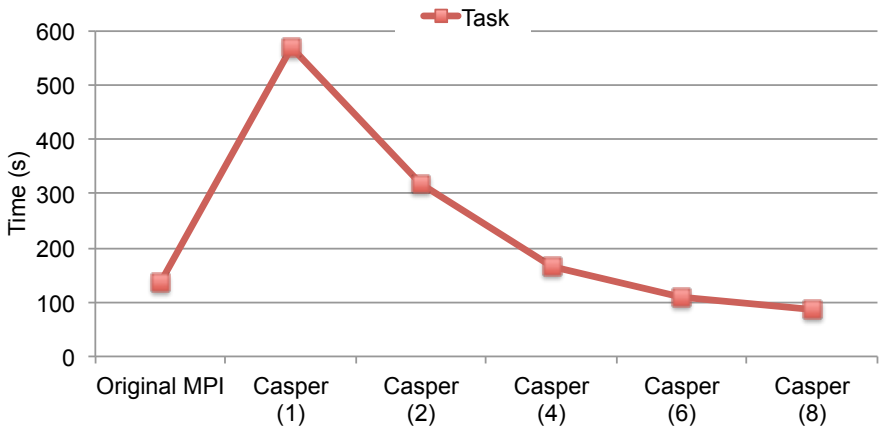
5.5.4 Adaptation Improvement

Our fourth set of experiments focus on the capability of dynamic adaptation by utilizing a microbenchmark that demonstrates the multi-phase applications on 8 nodes with 22 user processes and 2 ghost processes on each node. The microbenchmark is constructed as two window allocation, each of which performs a heavy-computation phase and a heavy-communication phase In each phase, every process performs twice all-to-all communication following the the same pattern as we used in the previous set of experiments (see Figure 5.8).

We compares the static Casper with four proposed adaptation modes: two user-guided modes Casper(U) and Casper(U-coll), and two self-profiling based dynamic adaptation approaches Casper(P) and Casper(GP). In the Casper(U) approach, we only set user hint `async_config` to `ON` in every window allocation; in Casper(U-coll), we add a MPI_WIN_SET_INFO call with user hint `symmetric`



(a) Computation-Intensive Phase

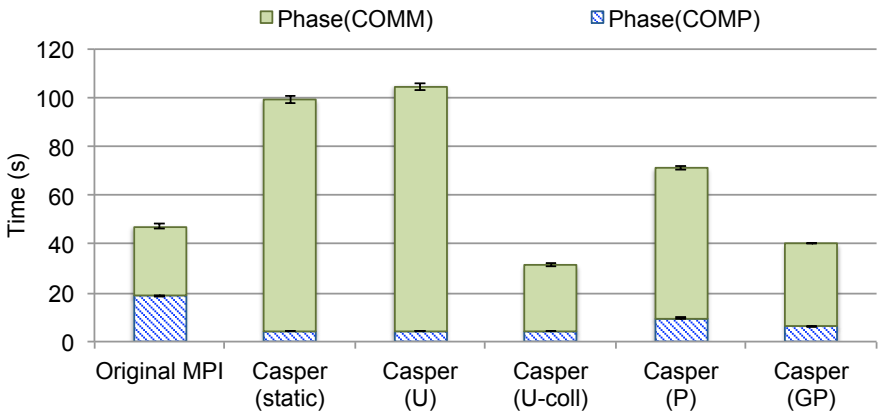


(b) Communication-Intensive Phase

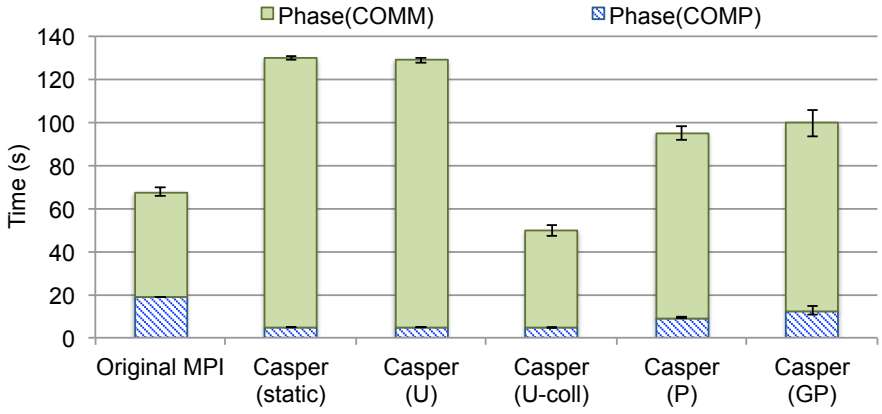
Figure 5.9: Performance Trend of Static Casper with Varying Number of Ghost Processes.

after the barrier in each phase, and set user hint `async_config` to `ON` before the heavy-computation phase, to `OFF` for the heavy-communication phase. With regard to the self-profiling based approaches, the Casper(P) mode only enables the user synchronization, while the Casper(GP) mode also enables the ghost-offloaded synchronization.

Figure 5.10(a) compares the execution time of each internal phase in GET communication with 3D noncontiguous double matrix as the target datatype. Comparing with the original MPI, the static Casper only takes 4 seconds in the heavy-computation phase thus, but also resulting in about 66 seconds degradation in the heavy-communication phase; the user-guided adaptation per window allocation, however, cannot address this issue, since it can only enable asynchronous progress for the first heavy-computation phase, but cannot disable it for the second one that performs heavy communication; within adding user hint in the `MPI_Win_set_info` after the barrier in each phase, the asynchronous progress can be disabled for the heavy-communication phase with also benefiting the asynchronous progress for the heavy-computation, thus delivering the best performance, 4.4 seconds in the heavy-computation phase and 27.1 seconds



(a) Get

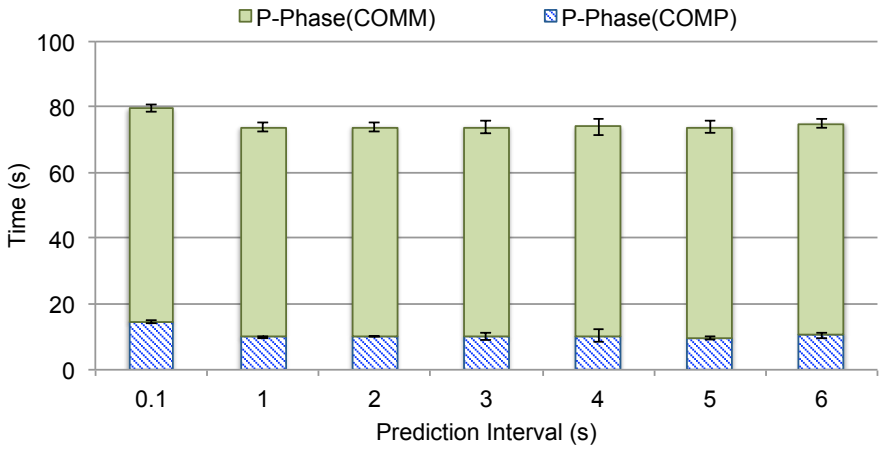


(b) Accumulate

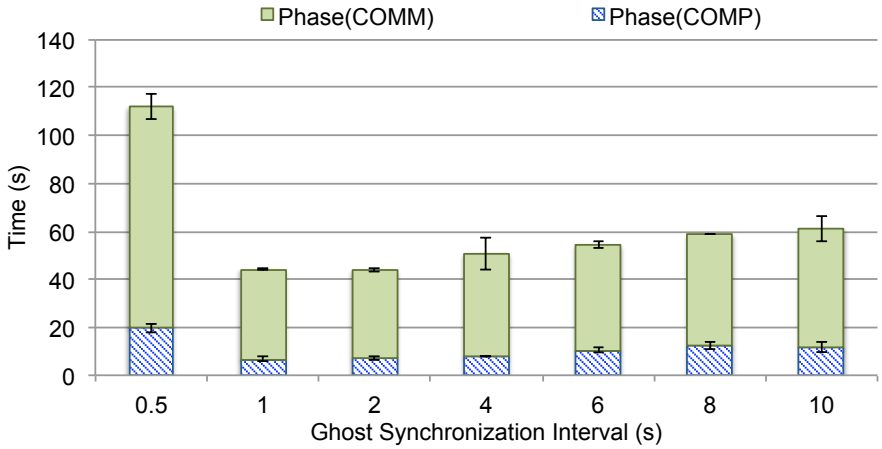
Figure 5.10: Adaptation in multi-phase execution.

in the communication phase. On the other hand, for the self-profiling based approaches, we set {85%,90%} for the frequency thresholds, 1 second as the prediction interval, and 2 seconds as the ghost synchronization in the Casper(GP) mode. Without ghost-offloaded adaptation, each phase shows additional overhead comparing to Casper(U-coll) (5.1 seconds in the computation phase and 34.4 seconds in the communication phase), this is because the prediction in the first `MPI_Win_set_info` is not correct since it is based on the history execution which always performs the opposite pattern (e.g., for the communication-intensive phase, the first prediction is always based on a computation-intensive execution). Within ghost-synchronization added in Casper(GP) mode, each phase can be adapted without relying on user collective calls, thus delivering a better performance which is very close to Casper(U-coll) with only 1.7 seconds and 6.9 seconds overhead in the heavy-computation phase and the heavy-communication phase respectively. This is the time waiting for the prediction and global synchronization with sufficient profiling data on all processes.

Different from the GET communication, the ACCUMULATE communication shows slightly different result. Figure 5.10(b) compares the performance of each approach following the same experiment. Although the Casper(U-coll) also de-



(a) Varying Prediction Interval (GET).



(b) Varying Ghost SYNC Interval (GET).

Figure 5.11: Threshold comparison in multi-phases adaptation.

livers the best performance with benefiting from the asynchronous progress in the heavy-computation phase and avoiding load imbalance in the heavy-communication phase, none of the self-profiling based approaches can achieve similar performance, both of them show about 40 seconds additional overhead in the communication phase since the first half execution of the phase can not be correctly adapted. Moreover, the Casper(GP) mode even results in visible overhead in the computation phase comparing to Casper(U-coll) at 7.8 seconds, this is because the ghost processes have to take time for global synchronization although it does not benefit the ACCUMULATE operations at all.

Besides the evaluation of dynamic adaptation, we also use the same experiment to observe the impact of different interval time. Figure 5.11(a) shows relatively consistent time of Casper(P) in the GET experiment with increasing prediction internal time, thus the impact of the prediction interval is minor. Figure 5.11(b) compares the execution time of Casper(GP) with increasing interval of ghost synchronization. When the interval is set to 0.5 second, substantial communication on the ghost processes results in twice performance degradation in both phases. With increasing interval time the overhead becomes invisible and the phases can benefit from adaptation, the best performance is delivered

at 2 seconds. However, when the interval is even increased, additional overhead gradually appears and finally results in 4.7 seconds degradation in the computation phase and 12.4 seconds in the communication phase at 10 seconds. This is because, the longer the interval we set, the later asynchronous progress status is globally exchanged.

5.6 NWChem Quantum Chemistry Application

In the previous chapter, we have demonstrated significant performance improvement of large chemistry application NWChem with the static Casper for the C_{20} problem and a very large water molecule $(H_2O)_{21}$ in the CCSD(T) simulation. However, we did not look into the break down of the performance in other internal phases. To exploit the maximum performance improvement, we will deeply study the performance characteristics of NWChem internal phases with the static Casper and evaluate the performance with our adaptation solution. In following sections, we first give an overview of the internal phases in the NWChem CCSD(T) task in Section 5.6.1, then in Section 5.6.2 we deeply analyze the performance characteristics and the limitation of static asynchronous progress in the internal phases of the CCSD(T) simulation with two different problem types, a Water molecule ($(H_2O)_n$) series, and an Acenes series, by using the static asynchronous progress approaches including both static Casper and the thread-based approaches. After understanding the shortcoming of static approaches, we demonstrate the benefit of dynamic adaptation in Section 5.6.3.

5.6.1 Overview of Multiple Internal Phases

The CCSD(T) method performs a complex set of multidimensional array computations organized in three internal phases: four-index transformation (4-index), CCSD iteration, and the noniterative (T) portion [31].

Four-index transformation: The 4-index phase requires a number of DGEMM computation with a non-collective global transpose in the middle, containing intensive PUT/GET/ACCUMUALTE communication following the *get-compute-update* mode (see Figure 4.18).

CCSD iteration: This phase evaluates the residual for the complex set of nonlinear CCSD equations. Each of the iterative steps composes of window allocation, a number of global synchronization calls and the typical *get-compute-update* loop that involves large amount of GET operations and a number of ACCUMULATE operations with DGEMM computation.

(T) portion: In the noniterative (T) portion phase, every process only perform one large matrix–matrix multiplication also following the typical *get-compute-updates* approach, containing large COMPUTE operations with numerous GET operations and a reduce operation (update) at the end of local multiplication.

All the above phases contain the nxtask module, in which processes issue atomic FETCH_AND_OP operation on a global shared counter to schedule the computation for the next sub-domain.

Table 5.2: Core deployment in NWChem evaluation on Cray XC30.

	Computing Cores	ASYNC Cores
Casper (1)	23	1
Casper (2)	22	2
Casper (4)	20	4
Casper (8)	16	8

5.6.2 Static Asynchronous Progress

We first analyze the performance of NWChem CCSD(T) simulation with static asynchronous progress approaches.

5.6.2.1 Experiments Overview

We evaluated the improvement of NWChem by comparing the static Casper with both the original MPI and two thread-based approaches. The first Thread (O) approach employs oversubscribed cores where every thread and its associated MPI process execute on the same core; and the second Thread (D) approach deploys dedicated cores where threads and MPI processes are executed on separate cores as listed in Table 4.2. For static Casper, we also compare the configuration with different number of ghost processes, by keeping the same total number of cores as the thread approaches. Table 5.2 lists the their detailed core deployment.

Different from the experiments in the previous chapter, we use the `NTS` task scheduling module in all of our experiments in the evaluation by adding "set tce:nts T" in the input files. The `NTS` module significantly reduces the overhead of `nxtask` scheduling especially in the CCSD iteration.

5.6.2.2 CCSD(T) with Water Molecules

In this section, we focus on the CCSD(T) method with water molecule problems ($(H_2O)_n$)—denoted W_n for short—with double-zeta basis sets (cc-pVDZ from the NWChem basis set library).

To have a global view of the performance effect caused by different asynchronous progress approaches for the entire task execution, we first evaluated Casper with comparison of both the original MPI and thread-based approaches in weak scaling. Figure 5.12 indicates the task time of CCSD(T) method for varying W_n-pVDZ problems ($n = 5, 10, 14, 16, 18, 21$). Casper consistently improves the performance from problem W_{10} to W_{21} by close to 30%. However, the thread-based approaches do not show performance improvement and perform even worse than the original MPI because of oversubscribed cores or appropriation of half of the computing cores. We notice that Casper shows 15% performance degradation in W_5, which is executed on only single node. This is because MPI internally allocates shared window for the processes on the same node, thus all the RMA operations are handled in hardware through the shared memory. Casper does not degrade the performance of communication, however, it takes 4 cores from the user computation.

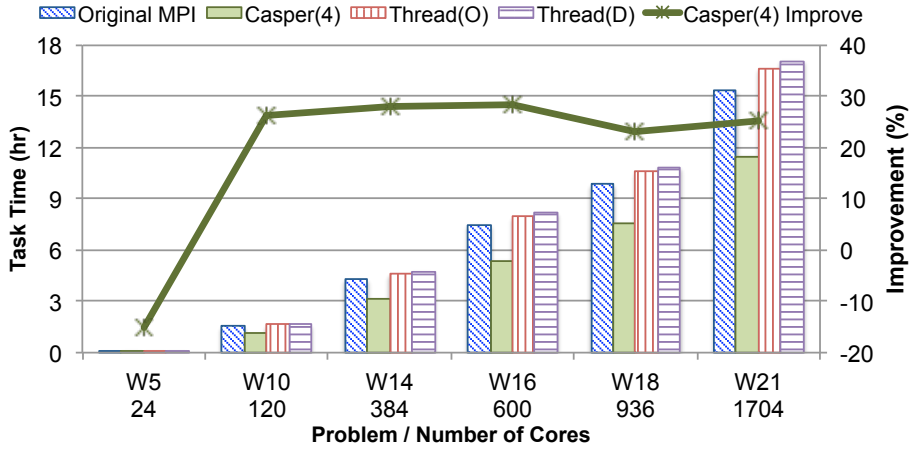


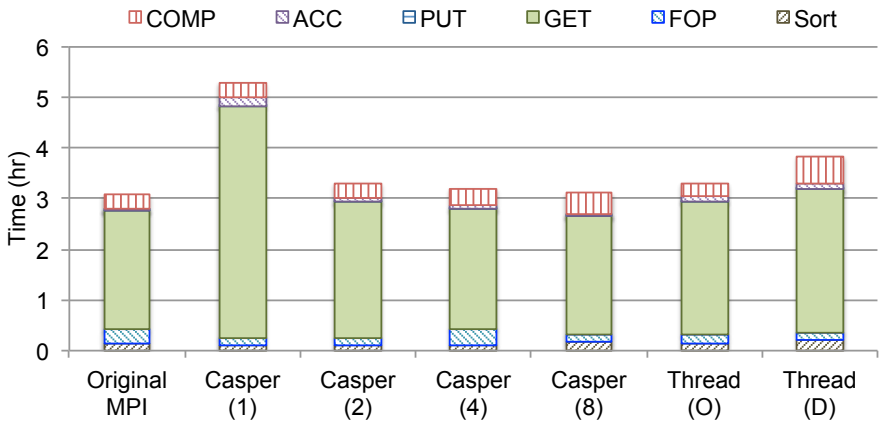
Figure 5.12: CCSD(T) for varying W_n with pVDZ on Cray XC30.

To investigate the impact of asynchronous progress on each internal phase, we next compare the time consumed by each phase for the CCSD(T) task. As we have described in Section 5.6.1, the CCSD(T) method consists of three primary internal phases, 4-index, CCSD iteration and (T) portion. As shown in Figure 4.19, the (T) portion consistently dominates the cost of entire task by close to 80% for all the water problems, and the CCSD iteration takes the other 20%. Therefore, we only focus on the analysis of the CCSD iteration and the (T) portion phases.

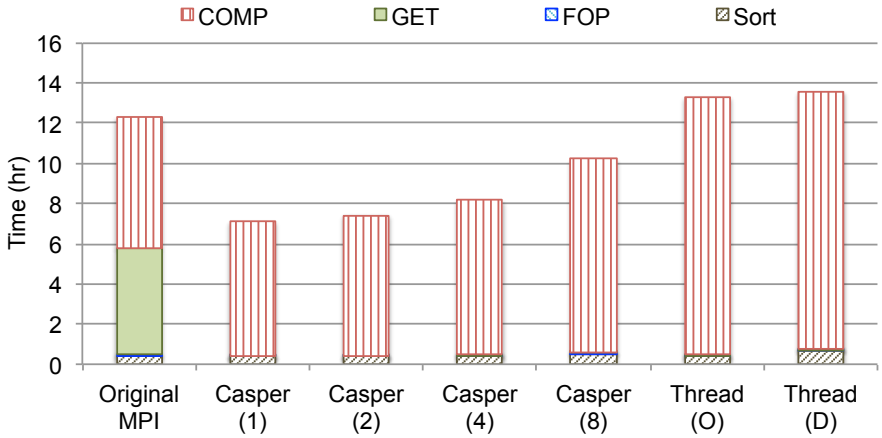
5.6.2.2.1 CCSD Iteration Phase : Figure 5.13(a) shows the profiling of the communication-dominative CCSD iteration phase in the W_{21}-pVDZ problem with 1704 cores. The numerous GET operations dominate the execution time by close to 80 %, while the DGEMM computation (shown as COMP) only takes less than 10 % of the cost and the FETCH_AND_OP (abbreviated to FOP) almost takes the rest 10 %.

Such intensive communication can rarely benefit from the asynchronous progress, however, can even result in performance degradation. When only a single ghost process is used within Casper, the overhead of GET operation becomes twice more expensive than the original MPI, because a large amount of GET operations, which were distributed to 24 user processes on each node, are redirected to a single ghost process, thus resulting in the bottleneck of communication load imbalance. With utilizing more ghost processes, this imbalance issue is resolved, however, the overhead of the DGEMM computation shows 26% increasing when using 8 ghost processes due to loss of computing cores.

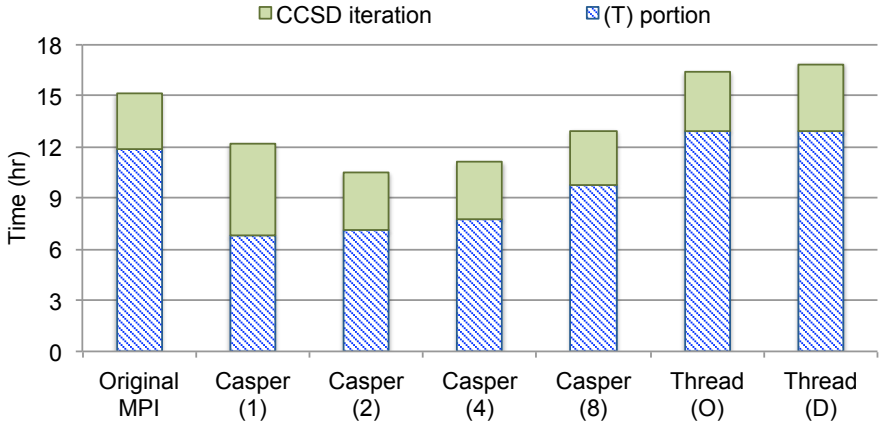
5.6.2.2.2 (T) Portion Phase : We next profile the most time-consuming phase: non-iteration (T) portion. Figure 5.13(b) shows the time consumed in the internal steps of (T) in W_{21}-pVDZ problem with 1704 cores. In the result of original MPI, the heavy computations takes 6.46 hours and the GET communication dominates the other half of cost, taking 5.35 hours. The significant overhead of GET clearly indicates the delay caused by heavy computation on the target processes when no asynchronous progress is provided. Both Casper and



(a) CCSD iteration



(b) (T) portion



(c) Trade off between CCSD iteration and (T)

Figure 5.13: Profiling CCSD iteration and (T) portion for W21 problem with 1704 cores on Cray XC30.

the thread-based approaches asynchronously handles the completion of GET operations thus the overhead of GET dramatically reduced. However, within more number of ghost processes in Casper, the computation resources is gradually reducing, thus resulting in increasing overhead of the computation part (3.18 hour degradation at 8 ghost processes). The thread-based approaches show even twice more expensive overhead in computation because of thread oversubscription and half of computing cores are dedicated to communication.

5.6.2.2.3 Asynchronous Progress Trade-Off : According to above profiling results, it is clear that the needs of asynchronous progress is varying for different phases in a CCSD(T) task. Figure 5.13(c) compares the time of CCSD iteration and (T) with the thread-based approaches and the static Casper with different number of ghost processes. Although the (T) portion gets a great improvement by using only one ghost process, the performance of the CCSD iteration shows 2.18 hours degradation because of load imbalance issue as explained in Figure 5.13(a). Hence, to achieve the optimal performance for the entire task, we need trade off among these internal phases that require different number of asynchronous cores. For example, the optimal number of ghost processes for the W_{21} problem is two on our platform.

5.6.2.3 CCSD(T) with Acenes Molecules

As shown in above section, the CCSD(T) task requires different number of ghost processes to obtain the optimal performance in CCSD iteration and (T) potion phases separately. Thus a static configuration of asynchronous progress can lead to insufficient performance improvement, or even performance degradation such as the performance loss in the CCSD iteration phase while specifying only one ghost process. For water problems, such insufficiency does not significantly impair the performance of the entire execution of task, since the (T) portion, whose performance could be improved by close to 50%, dominates the cost of entire task. However, this might not be true for other molecule problems.

In this section, we also look into another set of molecules, the acenes series including Naphthalene ($C_{10}H_8$), Anthracene ($C_{14}H_{10}$), Tetracene ($C_{18}H_{12}$), Pentacene ($C_{22}H_{14}$) and Hexacene ($C_{26}H_{16}$) molecules, with aug-cc-pVDZ from the NWChem basis set library, which show a different proportion for each internal phase in the execution of CCSD(T) task. These problems are abbreviated to Nap, Ant, Tet, Pent and Hexa respectively for convenient description.

Figure 5.14 compares the execution time of the internal phases in this problem series. Comparing with the results observed in water system, the proportion of (T) portion is significantly reduced, instead, the communication-intensive 4-index shows more overhead. For example, the (T) portion takes only 52% of the entire cost in the Tet problem while the 4-index becomes more expensive and takes close to 26% of the cost, and the CCSD iteration and other internal phases take the other 19% and 3% costs respectively.

Since the proportion of communication-intensive phases is increased, the performance degradation caused by static asynchronous progress in those phases becomes more significant. We then focus on the Tet problem running on 240

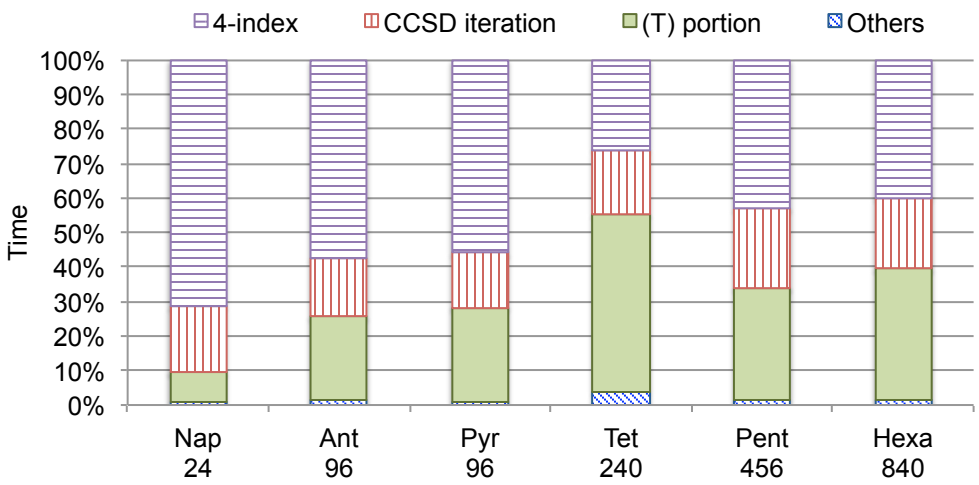


Figure 5.14: Internal phases in CCSD(T) task for Acenes molecules on Cray XC30.

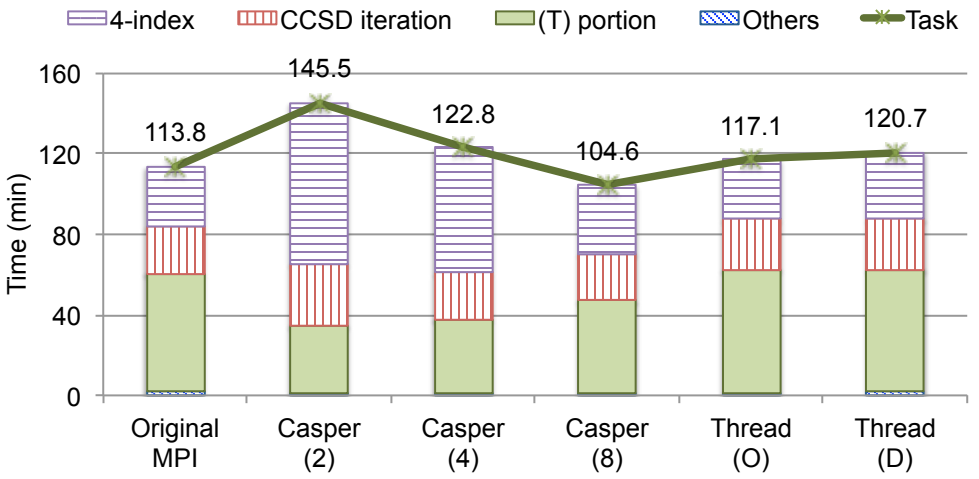


Figure 5.15: CCSD(T) task for Tet-pVDZ problem on 240 cores.

cores to demonstrate such inefficiency. Figure 5.15 compares the execution time of thread-based approaches and static Casper with 2, 4 and 8 ghost processes respectively. Unfortunately, neither Casper nor the thread-based approaches can provide efficient solution for the Tet problem. Although Casper can reduce the overhead of (T) portion by 40 % when using 2 ghost processes, it also causes severe degradation of 4-index by close to 160 %, thus resulting in even worse performance of the entire task execution. Withing increasing number of ghost processes, the degradation of 4-index can be gradually resolved, however, (T) portion shows increasing degradation because of loss of computing resources as studied in Section 5.6.2.2.

To ensure the reason causing different trend in these internal phases, we also profiled the time-consuming 4-index, CCSD, and (T) portion. Figure 5.16 compares the performance of original MPI and that with Casper by using 2 ghost processes. As expected, Casper eliminates most overhead of communication in the computation-intensive (T) portion; however, the overhead of ACCUMULATE

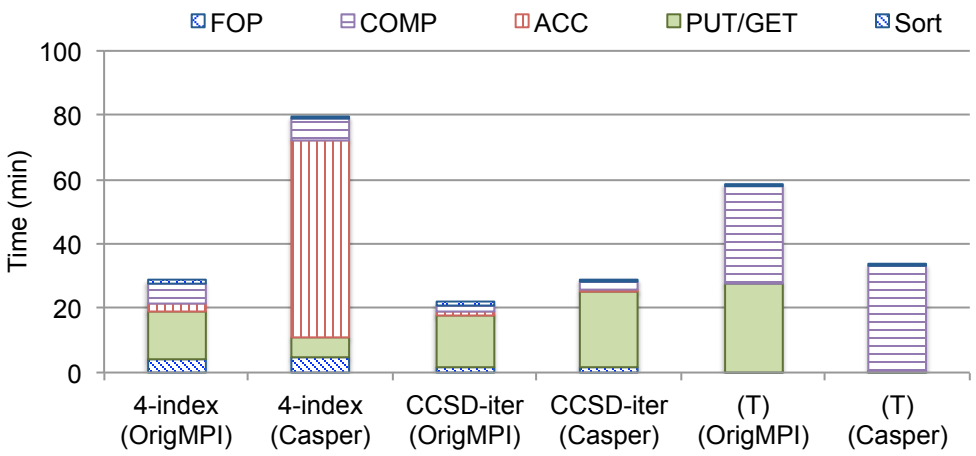


Figure 5.16: Profiling of Tet-pVDZ problem on 240 cores.

in 4-index step increases significantly, thus resulting in performance degradation in 4-index. This is because heavy ACCUMULATE communication which was received by 24 processes on each node, has to be handled by only 2 ghost processes.

5.6.3 Dynamic Adaptation

After comprehensively characterized the performance of multiple phases in NWChem and demonstrated the shortcoming of static asynchronous progress, especially in the acenes problems, we then evaluate the same tasks with our dynamic adaptation approaches.

5.6.3.1 Experiments Overview

As we have discussed in the above section, the static Casper, which only allows fixed number of ghost processes, can rarely benefit the acenes series in CCSD(T) task. In this section, we evaluate the Tet problem with pVDZ on 240 cores using the adaptation extension of Casper, which allows asynchronous progress to be changed in runtime. We compares the static Casper with four proposed adaptation modes as listed in Section 5.5.4: two user-guided modes Casper(U) and Casper(U-coll), and two self-profiling based dynamic adaptation approaches Casper(P) and Casper(GP)). We use two ghost processes in the Tet with pVDZ problem on 240 cores in all experiments.

In the user-guided approaches, we use `ON` as the global default value of asynchronous configuration, and {`OFF`, `OFF`, `ON`} as the value of user info `async_config` passed to window collective calls for the internal phases {4-index, CCSD iteration, (T) portion} respectively. We note that, `MPI_Win_allocate` is the only window collective call used in original NWChem. In Casper(U) approach, we only pass `async_config` info to `MPI_Win_allocate` for each phase; in Casper(U-coll) approach, we add `MPI_Win_set_info` call with user info `symmetric` and `async_config` in every `GA_Sync` call, since it synchronizes and guarantees the completion of all outstanding operations on all processes.

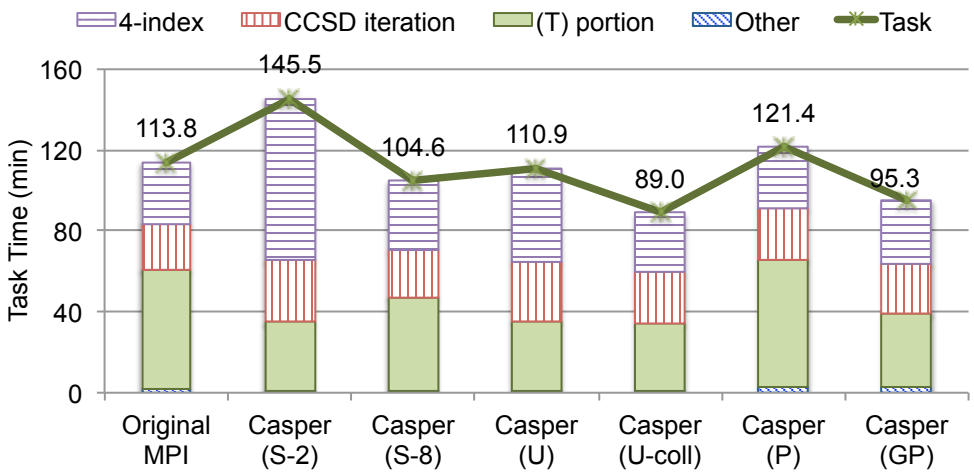
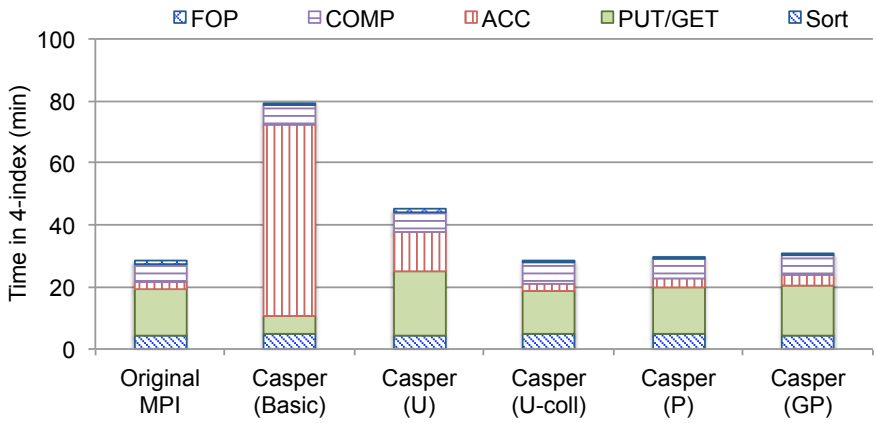


Figure 5.17: CCSD(T) for Tet-pVDZ problem with Casper adaptation on 240 cores.

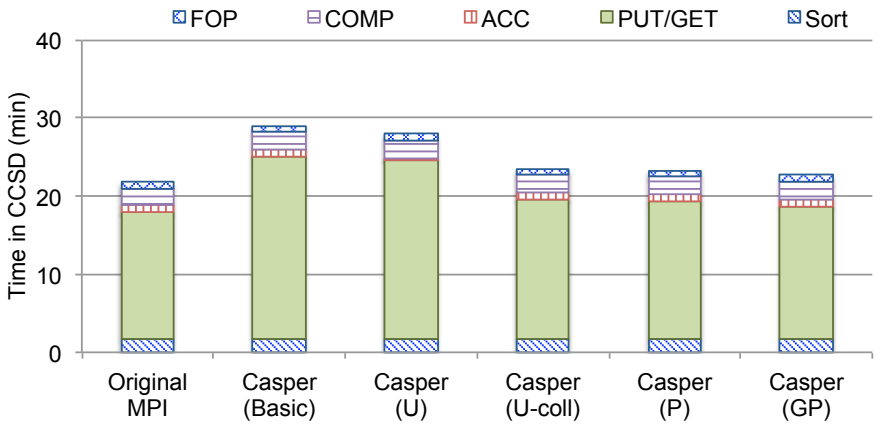
In the self-profiling based approaches, every user process automatically track the communication time taken on itself. In the Casper(P) approach, every user process updates its local asynchronous status according to the profiled communication frequency, and then exchange with other user processes in every window collective call. We use the same modified code as that of Casper(U-coll), thus the local status updating and global exchange phases can happen in both `MPI_Win_allocate` and `MPI_Win_set_info` calls. In the Casper(GP) approach, we control the frequency of adaptation using two factors: (1) every user process updates its local asynchronous status in any communication at specified interval (2 seconds in below experiments); (2) the background synchronization among ghost processes exchanges the latest asynchronous status for their binding user processes at specified interval time (compare 1, 2, 4, 6 minutes in the experiments).

5.6.3.2 Performance Analysis

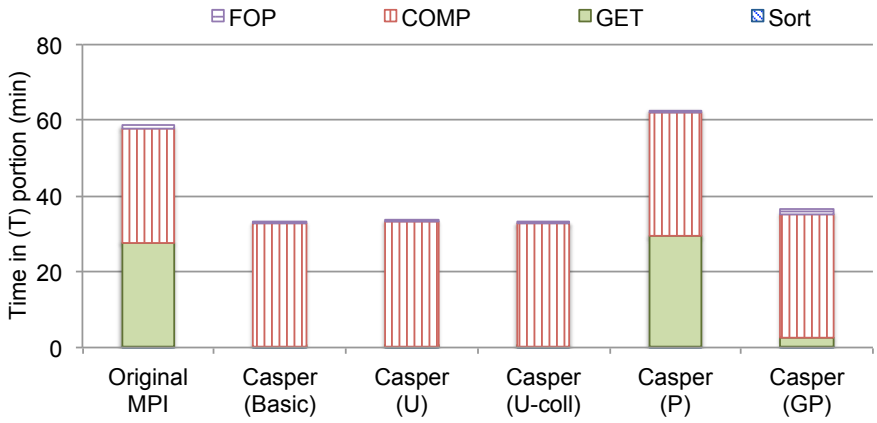
Figure 5.17 compares the task execution time of each adaptation approach with the static Casper. The Casper(U) approach relieves the communication bottleneck in 4-index phase by disabling asynchronous progress, while also improved the performance of (T) portion by re-enabling asynchronous progress. However, the overhead of 4-index and CCSD iteration is still slightly higher than that in the original MPI. Casper(U-coll) approach provides more improvement by inserting more window collective calls which allow Casper to change the asynchronous configuration more frequently and also for existing windows. On the other hand, although Casper(P) approach is able to resolve the over-workload issue in 4-index and CCSD iteration phases, thus delivering the same performance as that in the Casper(U-coll) approach, the overhead of (T) portion is much larger than the user-guided approaches and the static Casper and even slightly worse than the original MPI. The Casper(GP) approach, however, successfully addressed this issue and performs very similar performance as that in Casper(U-coll) approach without any user hint insertion.



(a) 4-Index Profiling



(b) CCSD iteration



(c) (T) portion Profiling

Figure 5.18: Profiling CCSD(T) for Tet-pVDZ problem with Casper adaptation on 240 cores.

We notice that the performance of Casper(GP) approach can vary when using different interval time for the background synchronization among ghost processes. In Figure 5.17, we only show the best performance result with 4 minutes interval, in order to focus on the difference between approaches.

To understand the reason of above performance results, we deeply profile and compare the communication time and computation time taken in each internal phase of CCSD(T) task with different approaches, and with different value of synchronization interval time in the Casper(GP) approach.

5.6.3.2.1 Adaptation Approaches : We first compares the performance of different adaptation approaches in the 4-index and (T) portion internal phases, in which we have observed significant performance change, as shown in Figure 5.18(a) and 5.18(c) respectively. The synchronization interval time is set to 4 minutes in the Casper(GP) approach in these experiments.

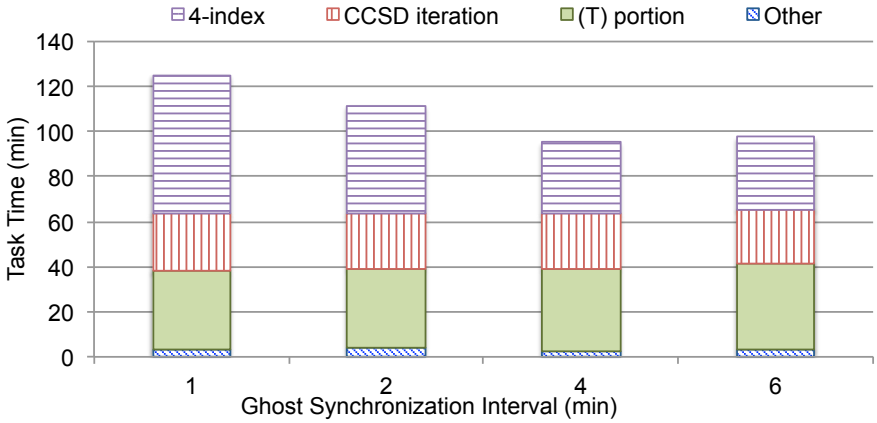
As shown in Figure 5.18(a), the Casper(U) approach can reduce the overhead of ACCUMULATE communication in 4-index phase since it disables the asynchronous progress at window allocation time, however, the overhead is still higher that the cost in original MPI because it does not adapt for existing windows thus load imbalance issue still exists on those existing windows and consequently degrade the performance of ACCUMULATE. The Casper(U-coll), Casper(P) and Casper(GP) approaches completely resolve such performance degradation, this is because both approaches allow asynchronous progress to be changed at both window allocation time for new windows, and at `GA_Sync` time for all existing windows.

For the (T) portion, as shown in Figure 5.18(c), both user-guided approaches significantly reduce the overhead of GET communication as that we have provided in static Casper, however, the Casper(P) approach cannot achieve the same performance and even shows slightly performance degradation comparing with the original MPI. This is because, different from the 4-index and CCSD iteration phases, the (T) portion is a non-iteration phase consisting of only heavy computation and enormous GET-flush operations. The window creation only happens at the start of this phase, and `GA_Sync` only happens at the end. Casper(P) still disables asynchronous progress for all processes at window creation time in this phase, since it is predicted based on the profiling data got from the previous CCSD iteration phase, which is communication-intensive. After a short period of execution, it gets sufficient profiling data to determine the pattern becomes computation intensive, however, can not get any chance to re-enable the asynchronous progress till the end of this phase. Moreover, although Casper(P) disables asynchronous progress, the cores dedicated to ghost processes still could not be reused in user computation, thus it shows slightly performance degradation even comparing with the original MPI.

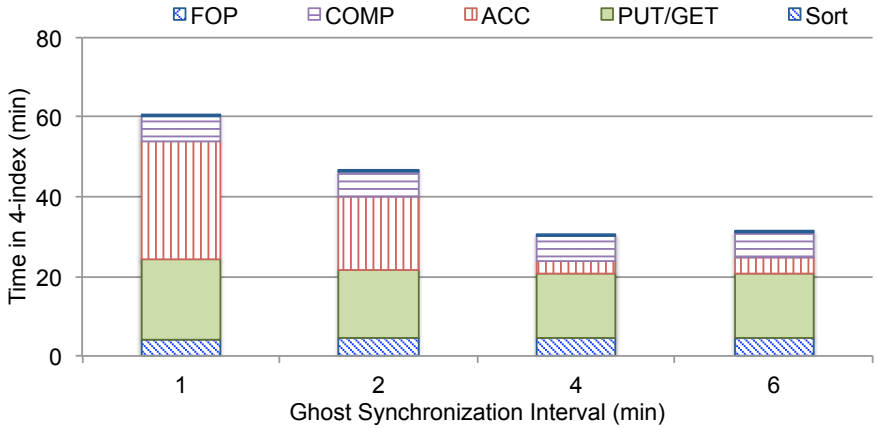
The Casper(GP) approach overcomes the above issue, it reduces most overhead of the GET communication but is still slightly worse than that in static Casper and the user-guided approaches. This is because, each user process does not update its local status collectively, thus it takes several rounds of background ghost synchronization to globally enable the asynchronous progress on all user processes.

With regard to the CCSD iteration phase, both the user-guided approach (Casper(U-coll)) and the profiling-based approaches can disable asynchronous progress appropriately, such avoid inefficient performance in this communication dominated phase.

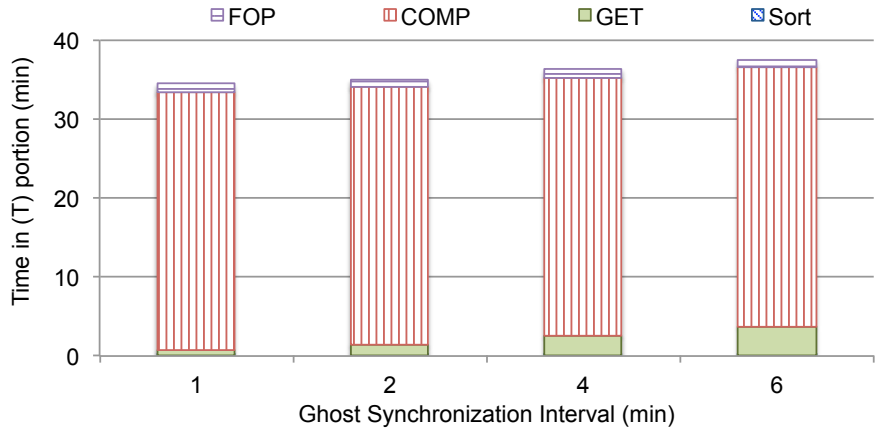
Ghost Synchronization Interval: As we have discussed in Figure 5.17, the ghost-offloaded Casper(GP) approach even improves the self-profiling based adaptation, especially overcomes the issue in the (T) portion phase. However, this approach requires user specified interval for the background synchronization among ghost processes, insufficient synchronization times may delay the change of asynchronous progress, but too frequent synchronization may cause additional communication overhead. The optimal value can vary on different platform and for different computation problems. In this paper, we manually compare the task execution time with 1, 2, 4 and 6 minutes as the interval of background synchronization as shown in Figure 5.19. The best performance is delivered when setting interval to 4 minutes on our platform. The 4-index phase and (T) portion phase show different trend with increasing interval time. As shown in Figure 5.19(b), heavy overhead has been observed in the ACCUMULATE communication with 1 and 2 minutes interval time because too frequent synchronization, such overhead is significantly reduced after reducing the frequency of synchronization by increasing the interval time. Figure 5.19(c) shows different trend for the (T) portion phase. The GET communication benefits from smaller interval time, because it highly replies on the asynchronous progress and frequent synchronization can enable the asynchronous progress earlier.



(a) Task execution time



(b) 4-index profiling



(c) (T) portion profiling

Figure 5.19: CCSD(T) for Tet-pVDZ on 240 cores with Casper(GP) adaptation using varying synchronization interval.

Chapter 6

Related Work

6.1 MPI with Multithreading Environment

The hybrid MPI+OpenMP programming model has been extensively used and studied in the past. For instance, Lusk and Chan [44] explored the performance of such a model on a typical Linux cluster, a large-scale system from SiCortex, and an IBM Blue Gene/P system. The authors concluded that some applications performed better with several MPI-only processes on the same node, while others could benefit from the hybridization. While this situation is still true today, an increasing number of applications are moving to hybrid MPI+OpenMP models, not just for performance, but for per-core resource limitations (in particular, memory). Other studies [64] have, on the other hand, reported satisfactory results in porting the finite-difference time-domain algorithm to the hybrid paradigm to adapt it to SMP compute nodes.

Smith and Kent [62] also found that increasing the number of threads decreased the efficiency of the code when implementing the quantum Monte Carlo algorithm on mixed OpenMP/MPI code on an SGI Origin 2000 system. Although this phenomenon was not attributed to the idle-threads issue we address in this paper, it certainly contributes to the reduced efficiency per thread. [68] performed a comprehensive evaluation of multithreaded MPI communications, pointing to the mutually exclusive regions involved in communication as one of the reasons for the suboptimal performance obtained.

Several researchers have also looked at optimizing the MPI implementation in multithreaded environments. For example, the authors of [13, 25, 29] proposed various techniques to minimize locking within the MPI implementation in order to improve the performance of MPI in `MPI_THREAD_MULTIPLE` environments. They presented various techniques to improve performance on traditional Linux clusters as well as the IBM Blue Gene/P systems. The authors in [23] proposed extensions to the MPI standard that would allow the MPI implementation to minimize contention and improve performance in some cases. However, all these optimizations are for `MPI_THREAD_MULTIPLE` applications. A large fraction of today's hybrid MPI applications, however, still use `MPI_THREAD_FUNNELED` and `MPI_THREAD_SERIALIZED` modes, for which these optimizations are not helpful.

6.2 MPI One-sided Communication and Asynchronous Progress

Asynchronous progress in MPI has been previously explored by the community for both two-sided and one-sided communication using multiple approaches. Sur et al. [65] discussed an interrupt-based design to overlap remote direct memory access read-based rendezvous communication with computation on InfiniBand networks. Kumar et al. [42] improved this work by proposing a signal-based approach to both reduce the number of interrupts and avoid using locks for shared data.

In one-sided communication, although networks such as InfiniBand provide contiguous PUT/GET operations in hardware, noncontiguous data transfers and accumulate operations still require the participation of the target process to perform an unpacking stage from a contiguous receiving buffer into the noncontiguous target location. Jiang et al. [39] proposed a thread-based design to enable asynchronous progress in communications involving noncontiguous data types in one-sided networks. Vaidyanathan et al. [75] improved asynchronous progress on Intel Xeon Phi coprocessors using a similar approach but were able to minimize threading overhead by implementing only a subset of the MPI standard and discarding some requirements of the standard.

PIOMan [73], a multithreaded communication progression engine supporting asynchronous progress, divides I/O communication and rendezvous handshakes into multiple tasks and offloads them to background threads running on idle cores in order to overlap communication and computation. This approach, however, suffers from a nonnegligible overhead derived from the necessary multithreading safety mechanisms [72]. In addition, to the best of our knowledge, the PIOMan project does not target one-sided communications.

Other research has focused on improving communication overlap using network hardware features. Santhanaraman et al. [56] optimized internode one-sided passive-mode synchronization using InfiniBand atomic operations, thus providing applications with improved overlap. Realizing that intranode communication is highly processor demanding, Zounmevo and Afsahi [79] proposed to overlap intra and internode one-sided communications by deferring the former messages falling under a certain message size threshold to the end of the epoch. By issuing network transfers to RDMA-assisted networks in first place, the processor-expensive intranode data movements are can be overlapped when issuing them subsequently.

Chapter 7

Conclusion and Future Work

This chapter concludes this dissertation by summarizing the research and the contributions in this work, with the discussion about two future directions.

7.1 Summary

Modern high end processor are aiming to improve both the thread-level and the instruction level parallelism for achieving the next grade of performance promotion. Many-core architecture is one of the dominating techniques in the high performance market, hundreds hardware threads with wide vector processing units provide applications a massive parallel environment on single chip. However, due to the features of low frequency cores and limited per-core memory capacity, traditional programming models can rarely get benefit on such architecture. On the other hand, the applications are moving to a dynamic and data-driven mode especially in chemistry and bioinformatics fields, such trend also focuses the performance optimization to be restricted by the irregularity of computation/ communication.

With increasing complexity in both hardware architectures and application softwares, application developer are increasingly looking at two programming models: the hybrid "MPI+Threads" programming model for better utilizing the computing power of many-core chips for regular applications, and the one-sided programming model for scheduling the dynamic data movements in irregular applications. However, both popular modes suffer from inefficient communication due to the restrictions in hardware and software, even resulting in severe performance degradation. This dissertation investigated the characteristics of the widely used message passing model on many-core architectures and proposed efficient strategies to optimize the communication in these models for various scientific applications.

Firstly, in the hybrid "MPI+Threads" programming model, the communication performance was degraded because the MPI calls are usually issued only by single lightweight thread in most hybrid applications while massive threads are being used in user computation. Such mode does not only limit the performance of communication but also raise up the computing resources utilization issue since most threads are idle during MPI communication. To address this problem, we proposed an internally multithreaded MPI approach, that allows MPI to transparently share idle threads from user application and parallelize the MPI internal

processing in order to fully utilize the massive parallel environment in communication. This approach is implemented by the modification in OpenMP runtime system and the parallelism in MPI implementation. Specifically, we modified the OpenMP runtime to expose the thread idleness information, and modified the MPI implementation to parallelize three aspects of the internal processing—data packing/unpacking for derived datatypes, large data transfer in shared memory communication and operation posting in InfiniBand network—as the showcase of this approach. Significant performance improvement has been observed through micro- and macro-benchmarks for each aspect.

Secondly, with regard to the irregular one-sided communication, although the semantics allows single process to specify the data movement without explicitly synchronization on the remote process thus potentially allowing asynchronous completion of communication with overlapping to the computation on remote processes, most MPI implementation does not support truly asynchronous operations for such mode. Although some simple operations such as contiguous PUT/GET operations can be offloaded to RDMA-supported network hardware, the other operations such as 3D double subarray accumulate, which is heavily used in scientific applications, still have to be handled in MPI software (i.e., by calling MPI routine to make progress on the remote process). Such limitation could result in arbitrary long delay in the user communication if the remote process is busing computing outside MPI. To resolve this critical issue, we proposed a portable process-based asynchronous progress model, named Casper, for the MPI one-sided communication on multi- and many-core architectures, that could transparently link with most MPI implementations on various platforms. Within this approach, a large quantum chemistry application has shown significant performance improvement which is never achieved in other traditional asynchronous progress approaches.

Finally, after deep study of the performance characteristics of scientific applications, we have identified the issue of static asynchronous progress approach in multi-phases applications. That is, some internal phases of application may perform heavy computation that requires the help of asynchronous progress in communication, while some others may be dominated by enormous data movements rather than computation thus performance can be degraded if such heavy communication which was distributed to multiple user processes is redirected to a few asynchronous cores. To overcome this issue and better support multi-phases application, we presented two dynamic adaptation strategies embedded in the Casper library, allowing the configuration of asynchronous progress to be dynamically adapted for various internal phases with varying proportion of communication and computation.

7.2 Future Work

7.2.1 Process Oversubscription and Dynamic Communication

Although the process-based asynchronous progress has successful improved the communication existing in a part of irregular applications that are implemented in one-sided mode, there are still some other factors that extremely restrict the performance of irregular applications. For example, it is common in imbalanced

communication that an MPI process takes long time to wait for receiving message from others because the other process might not arrive at the desired communication point. It is hard to optimize those issues in communication runtime system, user has to take a lot time to optimize their application in order to minimize such inefficiency.

As one future work of this dissertation, we plan to investigate the process oversubscription concept on the many-core environment, by utilizing user level processes (ULP) where a dozen of "OS processes" could be scheduled on single core with allowing user-level scheduling [58]. Following this concept, MPI can involve a large number of oversubscribed MPI processes and intelligently schedule them inside communication runtime. This approach could potentially provide several unique benefits, for example, an MPI process can switch to another "ready-to-go" process (i.e., an process has received its waiting message) while the currently active process is blocking for message arriving, thus achieving latency hiding.

7.2.2 Improvement in Asynchronous Progress

In Chapter 4 and 5, we have investigated the process-based asynchronous progress model for supporting irregular one-sided communication. This approach can potentially also benefit the other communication modes such as MPI two-sided and group communication. However, as the limitation of this direction, these modes are moving data between user-defined buffers, which can not be internally mapped to the ghost processes without special support from operating system. The `MPI_Win_create` and `MPI_Win_create_dynamic` window types in the one-sided mode are not supported in Casper because of the same limitation.

We plan to utilize several external memory mapping techniques (e.g., XPMEM on Cray machines [78], Partitioned Virtual Address Space [57]), to investigate the capability of asynchronous progress with careful design to ensure the semantics correctness for the rest communication modes in MPI.

Another limitation in Casper is that, we still have to keep aside the cores dedicated to asynchronous progress from the application even when the operation redirection is being disabled in the adaptation mode, hence resulting in waste of core utilization. We plan to also investigate the ULP concept within Casper, that allows us to decouple the view of MPI processes from the physical cores and thus enabling fully core utilization in asynchronous progress and user computation.

References

[1] Texas Advanced Computing Center - Stampede supercomputer. https://www.tacc.utexas.edu/stampede/.

[2] TOP500 supercomputing sites. `http://www.top500.org`.

[3] ASCI Red Supercomputer. `http://www.top500.org/system/168753`, 1996.

[4] Roadrunner Supercomputer. `http://www.top500.org/featured/top-systems/roadrunner-los-alamos-national-laboratory/`, 2008.

[5] MPI: A Message-Passing Interface Standard. `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, September 2012.

[6] Tianhe-2 Supercomputer. `http://www.top500.org/system/177999`, 2015.

[7] K. Ahmed and K. Schuegraf. Transistor Wars. *Spectrum, IEEE*, 48(11):50–66, November 2011.

[8] Edoardo Aprà et al. Liquid Water: Obtaining the Right Answer for the Right Reasons. In *SC*, 2009.

[9] Argonne National Laboratory. Mira Supercomputer. `https://www.alcf.anl.gov/mira`, 2012.

[10] Argonne National Laboratory. MPICH — High-Performance Portable MPI. `http://www.mpich.org`, 2014.

[11] Argonne National Laboratory. Aurora Supercomputer. `http://aurora.alcf.anl.gov`, 2015.

[12] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 1991.

[13] Pavan Balaji, Darius T. Buntinas, David J. Goodell, William D. Gropp, and Rajeev S. Thakur. Toward Efficient Support for Multithreaded MPI Communication. In *Euro PVM/MPI*, 2008.

[14] Ron Brightwell, Kevin Pedretti, and Trammell Hudson. SMARTMAP: Operating System Support for Efficient Data Sharing among Processes on a Multi-Core Processor. In *SC*. IEEE, 2008.

[15] Darius Buntinas and Guillaume Mercier. Implementation and Shared-Memory Evaluation of MPICH2 over the Nemesis Communication Subsystem. In *Euro PVM/MPI*, 2006.

[16] E. J. Bylaska et. al. NWChem, A Computational Chemistry Package for Parallel Computers, Version 6.3, 2013.

[17] J. Carlson. Green's Function Monte Carlo Study of Light Nuclei. *Phys. Rev. C*, 36:2026–2033, November 1987.

[18] National Energy Research Scientific Computing Center. Cori Supercomputer. `https://www.nersc.gov/users/computational-systems/cori/`, 2015.

[19] George Chrysos. Intel Xeon Phi Coprocessor - The Architecture. White paper, Intel Corporation, September 2012.

[20] Cray Inc. Cray Message Passing Toolkit. Technical report, Cray Inc., 2004.

[21] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

[22] J. Diamond, M. Burtscher, J.D. McCalpin, Byoung-Do Kim, S.W. Keckler, and J.C. Browne. Evaluation and Optimization of Multicore Performance Bottlenecks in Supercomputing Applications. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 32–43, April 2011.

[23] James S. Dinan, Pavan Balaji, David J. Goodell, Douglas Miller, Marc Snir, and Rajeev S. Thakur. Enabling MPI Interoperability Through Flexible Communication Endpoints. In *Euro MPI*, 2013.

[24] James S. Dinan, Pavan Balaji, Jeffrey R. Hammond, Sriram Krishnamoorthy, and Vinod Tipparaju. Supporting the global arrays PGAS model using MPI one-sided communication. In *IPDPS*, May 2012.

[25] Gabor Dozsa, Sameer Kumar, Pavan Balaji, Darius T. Buntinas, David J. Goodell, William D. Gropp, Joseph Ratterman, and Rajeev S. Thakur. Enabling Concurrent Multithreaded MPI Communication on Multicore Petascale Systems. In *Euro MPI*, 2010.

[26] Argonne Leadership Computing Facility. Theta Supercomputer. `http://aurora.alcf.anl.gov/`, 2015.

[27] Paul F Fischer, James W Lottes, and Stefan G Kerkemeier. nek5000 web page. *Web page: http://nek5000. mcs. anl. gov*, 2008.

[28] Megan Gilge. *IBM System Blue Gene Solution: Blue Gene/P Application Development.* IBM, June 2013.

[29] David J. Goodell, Pavan Balaji, Darius T. Buntinas, Gabor Dozsa, William D. Gropp, Sameer Kumar, Bronis R. de Supinski, and Rajeev S. Thakur. Minimizing MPI Resource Contention in Multithreaded Multicore Environments. In *IEEE Cluster*, 2010.

[30] William Gropp and Rajeev Thakur. Thread-safety in an MPI Implementation: Requirements and Analysis. *Parallel Comput.*, 33(9):595–604, September 2007.

[31] 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 and Computation: Practice and Experience*, 24(2):135–154, 2012.

[32] Alistair Hart. First Experiences Porting a Parallel Application to a Hybrid Supercomputer with OpenMP4.0 Device Constructs. In Christian Terboven, Bronis R. de Supinski, Pablo Reble, Barbara M. Chapman, and Matthias S. Mller, editors, *OpenMP: Heterogenous Execution and Data Movements*, volume 9342 of *Lecture Notes in Computer Science*, pages 73–85. Springer International Publishing, 2015.

[33] So Hirata. Tensor Contraction Engine: Abstraction and Automated Parallel Implementation of Configuration-Interaction, Coupled-Cluster, and Many-Body Perturbation Theories. *J. Phys. Chem. A*, 107:9887–9897, 2003.

[34] Intel Corporation. Intel(R) OpenMP runtime library. `http://www.openmprtl.org`, 2013.

[35] Intel Corporation. Many Integrated Core (MIC) Architecture — Advanced. `http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html`, 2013.

[36] Intel Corporation. Intel MPI library. `http://software.intel.com/en-us/intel-mpi-library`, 2014.

[37] Intel Corporation. Intel(R) knights landing processor. `https://software.intel.com/en-us/articles/what-disclosures-has-intel-made-about-knights-landing`, 2014.

[38] I. Ivanov, Jing Gong, D. Akhmetova, I. B. Peng, S. Markidis, E. Laure, R. Machado, M. Rahn, V. Bartsch, A. Hart, and P. Fischer. Evaluation of Parallel Communication Models in Nekbone, a Nek5000 Mini-Application. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 760–767, 2015.

[39] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, DhabaleswarK. Panda, Darius Buntinas, Rajeev Thakur, and WilliamD. Gropp. Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters. In *Euro PVM/MPI*, volume 3241 of *Lecture Notes in Computer Science*, pages 68–76. 2004.

[40] Ali Khajeh-Saeed, Stephen Poole, and J Blair Perot. A Comparison of Multi-Core Processors on Scientific Computing Tasks.

[41] Jeongnim Kim, Kenneth P Esler, Jeremy McMinis, Miguel A Morales, Bryan K Clark, Luke Shulenburger, and David M Ceperley. Hybrid Algorithms in Quantum Monte Carlo. *Journal of Physics: Conference Series*, 402(1):012008, 2012.

[42] Rahul Kumar, Amith R. Mamidala, Matthew J. Koop, Gopal Santhanaraman, and Dhabaleswar K. Panda. Lock-Free Asynchronous Rendezvous Design for MPI Point-to-Point Communication. In *Euro PVM/MPI*, volume 5205 of *Lecture Notes in Computer Science*, pages 185–193. 2008.

[43] Sameer Kumar and Michael Blocksome. Scalable MPI-3.0 RMA on the Blue Gene/Q supercomputer. In *Euro MPI*, 2014.

[44] Ewing Lusk and Anthony Chan. Early Experiments with the OpenMP/MPI Hybrid Programming Model. In *OpenMP in a New Era of Parallelism*, pages 36–47. Springer, 2008.

[45] Ewing L Lusk, Steven C Pieper, Ralph M Butler, et al. More Scalability, Less Pain: A Simple Programming Model and Its Implementation for Extreme Computing. *SciDAC Review*, 17(1):30–37, 2010.

[46] Stefano Markidis, Jing Gong, Michael Schliephake, Erwin Laure, Alistair Hart, David Henty, Katherine Heisey, and Paul Fischer. Openacc acceleration of the nek5000 spectral element code. *International Journal of High Performance Computing Applications*, 29(3):311–319, 2015.

[47] Jintao Meng, Bingqiang Wang, Yanjie Wei, Shengzhong Feng, and Pavan Balaji. SWAP-Assembler: Scalable and Efficient Genome Assembly Towards Thousands of Cores. *BMC bioinformatics*, 15(Suppl 9):S2, 2014.

[48] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical report, Message Passing Interface Forum, September 2012.

[49] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. Introducing the Graph 500. In *Proceedings of the Cray User's Group Meeting (CUG)*, May 2010.

[50] B Neelima and Prakash S Raghavendra. Recent Trends in Software and Hardware for GPGPU Computing: A Comprehensive Survey. In *Industrial and Information Systems (ICIIS), 2010 International Conference on*, pages 319–324. IEEE, 2010.

[51] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global Arrays: A Portable "Shared-Memory" Programming Model for Distributed Memory Computers. In *ACM/IEEE conference on Supercomputing*, 1994.

[52] Stephen Olivier, Allan Porterfield, Kyle Wheeler, Michael Spiegel, and Jan Prins. OpenMP Task Scheduling Strategies for Multicore NUMA Systems. *The International Journal of High Performance Computing Applications*, (26(2)):110–124, May 2012.

[53] OpenFabrics Alliance. OpenFabrics Alliance. `http://www.openfabrics.org`, 2013.

[54] Robert Colwell. The Chip Design Game at the End of Moores Law. Keynote Speech, Hot Chips 25 Conference, Auguest 2013.

[55] Robert Ross and Neill Miller. Implementing Fast and Reusable Datatype Processing. In *In EuroPVM/MPI*, pages 404–413. Springer Verlag, 2003.

[56] Gopalakrishnan Santhanaraman, Sundeep Narravula, and Dhabaleswar K. Panda. Designing Passive Synchronization for MPI-2 One-Sided Communication to Maximize Overlap. In *IPDPS*, 2008.

[57] Akio Shimada, Balazs Gerofi, Atsushi Hori, and Yutaka Ishikawa. Proposing a New Task Model Towards Many-core Architecture. In *Proceedings of the First International Workshop on Many-core Embedded Systems*, MES '13, pages 45–48, New York, NY, USA, 2013. ACM.

[58] Akio Shimada, Atsushi Hori, Yutaka Ishikawa, and Pavan Balaji. User-Level Process towards Exascale Systems. *IPSJ SIG Technical Report*, 2014.

[59] Min Si, Antonio J Peña, Pavan Balaji, Masamichi Takagi, and Yutaka Ishikawa. MT-MPI: Multithreaded MPI for Many-Core Environments. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 125–134. ACM, 2014.

[60] Min Si, Antonio J Peña, Jeff Hammond, Pavan Balaji, and Yutaka Ishikawa. Scaling NWChem with Efficient and Portable Asynchronous Communication in MPI RMA. *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium*, 2015.

[61] Min Si, Antonio J Peña, Jeff Hammond, Pavan Balaji, Masamichi Takagi, and Yutaka Ishikawa. Casper: An Asynchronous Progress Model for MPI RMA on Many-Core Architectures . In *Parallel and Distributed Processing, 2015. IPDPS 2015.*

[62] Lorna Smith and Paul Kent. Development and Performance of a Mixed OpenMP/MPI Quantum Monte Carlo Code. *Concurrency Practice and Experience*, 12(12):1121–1129, 2000.

[63] CORPORATE SPARC International, Inc. *The SPARC Architecture Manual (Version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[64] Mehmet F Su, Ihab El-Kady, David A Bader, and S-Y Lin. A Novel FDTD Application Featuring OpenMP-MPI Hybrid Parallelization. In *International Conference on Parallel Processing (ICPP)*, pages 373–379. IEEE, 2004.

[65] Sayantan Sur, Hyun-Wook Jin, Lei Chai, and Dhabaleswar K Panda. RDMA Read based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits. In *PPoPP*, pages 32–39, 2006.

[66] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005.

[67] Masamichi Takagi, Yuichi Nakamura, Atsushi Hori, Balazs Gerofi, and Yutaka Ishikawa. Revisiting Rendezvous Protocols in the Context of RDMA-capable Host Channel Adapters and Many-core Processors. In *Euro MPI*, 2013.

[68] Rajeev Thakur and William Gropp. Test Suite for Evaluating Performance of Multithreaded MPI Communication. *Parallel Computing*, 35(12):608–617, 2009.

[69] The Ohio State University. OSU Micro-Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks, 2013.

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

[71] The Open MPI Development Team. Open MPI: Open source high performance computing. `http://www.open-mpi.org`, 2014.

[72] François Trahay, Élisabeth Brunet, and Alexandre Denis. An Analysis of the Impact of Multi-Threading on Communication Performance. In *9th Workshop on Communication Architecture for Clusters (CAC)*, May 2009.

[73] François Trahay and Alexandre Denis. A Scalable and Generic Task Scheduling System for Communication Libraries. In *IEEE Cluster*, September 2009.

[74] John R Tramm and Andrew R Siegel. Memory Bottlenecks and Memory Contention in Multi-Core Monte Carlo Transport Codes. *Annals of Nuclear Energy*, 2014.

[75] Karthikeyan Vaidyanathan, Kiran Pamnany, Dhiraj D. Kalamkar, Alexander Heinecke, Mikhail Smelyanskiy, Jongsoo Park, Daehyun Kim, Aniruddha Shet, G, Bharat Kaul, Balint Joo, and Pradeep Dubey. Improving Communication Performance and Scalability of Native Applications on Intel Xeon Phi Coprocessor Clusters. In *IPDPS*, 2014.

[76] 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, and W. A. de Jong. NWChem: A Comprehensive and Scalable Open-Source Solution for Large Scale Molecular Simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.

[77] Alan J. Wallcraft and Daniel R. Moore. The NRL Layered Ocean Model. *Parallel Computing*, 23(14):2227–2242, 1997. Parallel computing in regional weather modeling.

[78] Michael Woodacre, Derek Robb, Dean Roe, and Karl Feind. The SGI Altix 3000 Global Shared-Memory Architecture. White paper, Silicon Graphics, Inc, April 2003.

[79] Judicael A Zounmevo and Ahmad Afsahi. Intra-Epoch Message Scheduling to Exploit Unused or Residual Overlapping Potential. In *Euro MPI*, 2014.