STAPL-RTS: A RUNTIME SYSTEM FOR MASSIVE PARALLELISM

A Dissertation

by

IOANNIS PAPADOPOULOS

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Lawrence Rauchwerger
Committee Members,	Nancy M. Amato
	Jennifer L. Welch
	Marvin L. Adams
Head of Department,	Dilma Da Silva

May 2016

Major Subject: Computer Science

ABSTRACT

Modern High Performance Computing (HPC) systems are complex, with deep memory hierarchies and increasing use of computational heterogeneity via accelerators. When developing applications for these platforms, programmers are faced with two bad choices. On one hand, they can explicitly manage machine resources, writing programs using low level primitives from multiple APIs (e.g., MPI+OpenMP), creating efficient but rigid, difficult to extend, and non-portable implementations. Alternatively, users can adopt higher level programming environments, often at the cost of lost performance.

Our approach is to maintain the high level nature of the application without sacrificing performance by relying on the transfer of high level, application semantic knowledge between layers of the software stack at an *appropriate level of abstraction* and performing optimizations on a per-layer basis. In this dissertation, we present the *STAPL Runtime System* (STAPL-RTS), a runtime system built for portable performance, suitable for massively parallel machines. While the STAPL-RTS abstracts and virtualizes the underlying platform for portability, it uses information from the the upper layers to perform the appropriate low level optimizations that restore the performance characteristics.

We outline the fundamental ideas behind the design of the STAPL-RTS, such as the always distributed communication model and its asynchronous operations. Through appropriate code examples and benchmarks, we prove that high level information allows applications written on top of the STAPL-RTS to attain the performance of optimized, but ad hoc solutions. Using the STAPL library, we demonstrate how this information guides important decisions in the STAPL-RTS, such as multi-protocol

communication coordination and request aggregation using established C++ programming idioms.

Recognizing that nested parallelism is of increasing interest for both expressivity and performance, we present a parallel model that combines asynchronous, one-sided operations with isolated nested parallel sections. Previous approaches to nested parallelism targeted either static applications through the use of blocking, isolated sections, or dynamic applications by using asynchronous mechanisms (i.e., recursive task spawning) which come at the expense of isolation. We combine the flexibility of dynamic task creation with the isolation guarantees of the static models by allowing the creation of asynchronous, one-sided nested parallel sections that work in tandem with the more traditional, synchronous, collective nested parallelism. This allows selective, run-time customizable use of parallelism in an application, based on the input and the algorithm.

To my bride, Adrea.

To my parents, Antonis and Despina, and my brother, Miltiadis-Alexios.

To my grandparents.

And to all my friends that were there for me.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Dr. Lawrence Rauchwerger, for his support and guidance throughout my PhD studies. Were it not for his insight and his support, I would not have been able to make it through this journey. Our lengthy discussions about research and life have shaped me both as a researcher and as a person. Through constant mentoring, he instilled me with confidence in my abilities as an independent researcher, the importance of academic honesty and of the scientific method, and the ability to follow a structured approach to any problem, from the high level description to the implementation details.

I am also grateful to Dr. Nancy M. Amato for her guidance in my research and her support, especially in stressful situations. By focusing on the algorithmic aspect of my research, she greatly contributed to the successful conclusion of my studies. And whenever a paper submission seemed hopeless, she was there to advise and help me overcome any adversity.

I would like to thank Dr. Jennifer L. Welch for dedicating considerable amount of time and effort into guiding me through one of the most difficult sections of my work, that of message ordering and consistency models. Her knowledge and comments gave me new directions and insight about this exciting area of research.

I am thankful to Dr. Marvin L. Adams for his ideas and perspective. His view helped me see my work through the eyes of a user. His performance requirements pushed me to explore new solutions. More importantly, he showed me the implications of my research to real life problems.

I would also like to thank the research staff of the Parasol Laboratory, Dr. Timmie Smith and Dr. Nathan Thomas. Through lengthy discussions about features,

research topics, concepts, code and implementation, I became a better researcher, collaborator, and programmer. Being new PhD graduates themselves, they could relate to my worries and anxieties and help me overcome them. On multiple occasions they have dedicated time, effort and countless nights collaborating on research and publications without which this dissertation would not have existed.

As a member of the Parasol Laboratory, I had the opportunity to work alongside some amazing people. They have all contributed to my education and understanding of parallel computing through the development of STAPL and everyday interaction. Firstly, I would like to thank Adam Fidel for collaborating with me on multiple publications. Thanks also to all my fellow students, Justin Frye, Harshvardhan, Glen Hordemann, Dielli Hoxha, Alireza Majidi, Colton Riedel, Junjie Shen, Brandon West and Mani Zandifar, and former students, Antal Buss, Nicolas Castet, Vincent Marsy, Chidambareswaran (Chids) Raman, Olivier Rojo, Shishir Sharma, Xiabing Xu and Shuai Ye. I will never forget the time we all spend together and I hope our paths will cross again. Finally, I would like to thank my former colleagues, Dr. Mauro Bianco, Mr. Robert Metzger and Dr. Gabriel Tanase for all their help on multiple occasions.

A big special thank you to my family and friends. I thank my parents, Antonis and Despina, my grandma, Zoe, and my brother, Miltiadis-Alexios, for all their support and advice throughout my studies; I would not have made it without you. My soon-to-be wife, Adrea, has been by my side through good and bad times ever since we met. Her support and encouragement allowed me to push forward and eventually succeed. I am so excited about the new chapter in our life! I am also very grateful to all my friends. Going through a PhD is a difficult, yet gratifying experience, but you have made it unforgettable.

This research is supported in part by NSF awards CNS-0551685, CCF-0702765, CCF-0833199, CCF-1439145, CCF-1423111, CCF-0830753, IIS-0916053,

IIS-0917266, EFRI-1240483, RI-1217991, by NIH NCI R25 CA090301-11, by DOE awards DE-AC02-06CH11357, DE-NA0002376, B575363, by Samsung, IBM, Intel, and by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST). This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

NOMENCLATURE

HPC	High Performance Computing
MPI	Message Passing Interface
OpenMP	Open Multi-Processing
PE	Processing Element
PGAS	Partitioned Global Address Space
RMI	Remote Method Invocation
SPMD	Single Program Multiple Data
STAPL	Standard Template Adaptive Parallel Library
STAPL-RTS	Standard Template Adaptive Parallel Library Runtime System
STL	Standard Template Library

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Programming in a general and portable way has always been a hard task. Programming in parallel is an even harder task, as performance has to be guaranteed not only for current parallel hardware, but also for future ones, without compromising portability. This task becomes even more difficult considering the complex hierarchy of current petascale and future exascale machines.

SmartApps [1] attempts to address this complexity by adopting *application-centric* computing. The application is responsible for transferring information from the top (application) to the bottom (machine). The application is at the forefront and flows contextual information that each layer can use to adapt, increasing opportunities for optimization.

In this dissertation, we present our approach to supporting the SmartApps philosophy of "measure, compare, and adapt if beneficial" at the runtime system level. We will describe the motivation, concept, design, and implementation of the *STAPL Runtime System* (stapl-rts), a runtime system targeted to user-friendly and portable programming frameworks for High Performance Computing (HPC).

Our research is focused on providing a layer that abstracts and virtualizes the underlying platform and provides a shared-memory view of the system, a communication model based on Remote Method Invocations (RMIs) on distributed objects, support for nested parallelism, and the ability to easily partition the machine. The stapl-rts provides a common interface for both shared-memory and distributed memory platforms, while still fully exploiting the platform capabilities, providing portability and performance.

We will describe a runtime system that is modular and configurable, so that it

can be easily ported onto new platforms, and adaptive, so as to take into account runtime parameters and offer the best performance possible. The STAPL-RTS will be evaluated both in isolation and through its use by the *Standard Template Adaptive Library* (STAPL) [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16], a framework developed in C++ for parallel programming that uses the STAPL-RTS to offer parallel algorithms, distributed data structures and abstract data types that decouple a container interface from the underlying storage that have interfaces similar to the (sequential) ISO C++ Standard Template Library (STL) [17].

1.1 Contributions

The runtime system presented in this dissertation, called STAPL-RTS, makes the following novel contributions.

- **Exploiting machine hierarchy through nested parallelism.** We will present an execution model that takes advantage of the machine hierarchy through enabling nested parallel algorithms to be mapped and execute on hierarchical machines. Our model allows the asynchronous creation of nested parallel sections as opposed to prior work, enabling new applications for nested parallelism.

- **Transfer of application semantics to the runtime.** We employ annotations based on common programming idioms, for example move semantics [18] and immutable sharing [19], as well as algorithm driven optimizations, to perform *instance* specific optimizations, such as leveraging shared memory and relaxing communication ordering guarantees.

- **Unification of shared and distributed memory communication using asynchronous primitives.** The STAPL-RTS offers a Remote Method Invocation (RMI) based communication model on distributed objects over both shared

and distributed memory. It abstracts the machine hierarchy levels, offering a unified interface that releases the users from the intricacies of hybrid communication solutions (e.g., hybrid OpenMP and MPI) without sacrificing performance.

- **Causal communication ordering with support for explicit relaxation.** The STAPL-RTS provides causal ordering for all communication with the ability to explicitly relax it if an algorithm or data structure allows. We will discuss how causal ordering is used in real use cases to provide a consistency model, such as in distributed containers, and we will show the effects that algorithmic driven relaxation of ordering requirements can have on performance.

1.2 Outline

This dissertation is organized as follows. In Section 2 we begin with a discussion of the challenges and the requirements that need to be met to support massively parallel machines with complex hierarchies. We will describe the qualities that a runtime system has to exhibit to fulfill those requirements, our approach to meeting those requirements, and a brief comparison with related work.

In Section 3 we describe the design of the STAPL-RTS, focusing on its general concepts, the communication and task execution interfaces offered, while Section 4 will provide a short overview of the experimental setup used in this dissertation.

Section 5 introduces the Standard Template Adaptive Parallel Library (STAPL) as a use case of the concepts, execution model, and primitives of the STAPL-RTS.

In Section 6 we demonstrate how to get advantage of shared memory by passing high level information from the application, through STAPL, to STAPL-RTS.

Section 7 will present a novel execution model based on asynchronously created concurrent nested parallel sections. We describe the concepts, interfaces and how the execution progresses when mapping an hierarchical algorithm, expressed using

nested parallelism, to the physical machine hierarchy.

Finally, Section 8 outlines the communication ordering guarantees and how they are used by STAPL components and evaluates the effects of relaxing these guarantees. We conclude this dissertation with Section 9, which will summarize our findings. The appendices provide useful insight on how code using the STAPL-RTS looks like compared to MPI, some implementation details of the STAPL-RTS and a high level overview of the code organization.

2. RUNTIME SYSTEMS FOR HIGH PERFORMANCE COMPUTING

2.1 Runtime System Definition

Programming languages and frameworks often specify an execution model that dictates how execution takes place during the lifetime of an application. The component that implements part of the execution model and offers features that support it is the *runtime system*. As framed in [20], a runtime system is a software component that provides essential services to a language or a library and applications implemented on top of them.

Runtime systems are usually domain and perhaps language specific. They are developed around specific needs and requirements of the upper layers, offering an abstraction layer of the physical system. For example, various languages, such as FORTH [21], Lisp [20] and even modern ones, such as Microsoft .NET [22], depend on a specialized runtime system that offers the environment required for applications written in those languages to execute.

For the domain of High Performance Computing (HPC), runtime systems face multiple challenges. They have to support parallel execution through a model that can be easy to program and reason about while not hindering scalability. They are required to abstract a wide range of platforms, which can range from small workstations, or even embedded systems, to large, networked clusters of nodes, without compromising performance.

Runtime systems targeting HPC applications abstract, or virtualize, the underlying platform, providing a layer that promotes application portability and ease of development, while at the same time attempt to take advantage of all the features of the platform, offering *portable performance*. For the rest of this dissertation, the

term runtime system refers to any runtime system for HPC applications.

In brief, a good runtime system for HPC applications should:

- support an *execution model* that is suitable for scalability and performance,
- offer *platform abstraction* to decouple user applications from the underlying platform, e.g., by providing a unified communication layer,
- be *adaptive* and facilitate *vertical integration* through abstract communication methods between the top layer (application) and the bottom layer (hardware), e.g., by downstream transfer of application information and upstream run-time conditions reporting, so that each layer can adapt dynamically,
- be *modular* and *extensible* to allow integration with other runtime systems and support new application requirements, and
- be *portable* and *configurable* so that it supports different and future platforms without sacrificing performance.

In this dissertation, we will show that the STAPL-RTS fulfills all of the above requirements through the description of its design, appropriate code samples that show its capabilities, and microbenchmarks, kernels, and applications demonstrating its performance compared to the state-of-the-art.

2.2 Design Decisions

The design decisions that have to be made for a runtime system are the fundamental principles of the execution model. In this section we outline the fundamental design decisions behind the STAPL-RTS and the execution model it supports and the reasoning behind them.

2.2.1 Partitioned Global Address Space

Typically, the computing requirements of HPC applications surpass that of a single processor, requiring machines that consist of interconnected nodes, each with its own processing and memory resources. Supporting *massively parallel* [23] machines and operating under a model that can take advantage of them is paramount for performance and scalability. The *Single Program Multiple Data* (SPMD) model [24] is an intuitive way of programming massively parallel machines. While SPMD may seem more restrictive compared to the *Multiple Program Multiple Data* (MPMD) model, it is more intuitive and easy to program with [25].

Shared memory machines, while easy to program, have limited processor and memory scalability, requiring costly and complex mechanisms for maintaining cache coherence. On the other hand, distributed memory machines, are more difficult to program, but easier to scale.

Various models have been developed to bring the ease of use of shared memory programming models to distributed memory machines. For example, the *Distributed Shared Memory* (DSM) model [26] presents a global address space by allowing memory pages to be shared across different physical address spaces and managing coherence automatically. However, DSM encounters some performance issues [27].

The *Partitioned Global Address Space* (PGAS) model addresses the performance issues of DSMs by distinguishing between local and remote memory in SPMD programming models. Local memory is directly accessible, using regular memory accesses, while remote memory is accessed through a communication layer. Its use in a number of parallel programming languages, such as UPC [28] and Co-Array Fortran [29] has proven its performance potential and programmability.

Using a PGAS model coupled with locality information allows the runtime sys-

tem to efficiently utilize hierarchical machines. The knowledge of if communication happens in shared or distributed model is replaced with appropriate high level information about locality. Higher level programming frameworks can then be written by relying on qualitative information about the communication, e.g., what are the relative latencies, rather than on quantitative, such as the latency to read or write to a specific object in memory, which is machine and execution dependent.

2.2.2 Nested Parallelism

Function invocation is important in program development, as it is the foundation of software composition. The ability to provide *nested function invocation* plays a major role in programmer productivity and reuse. Function composition is especially important in parallel programming models, as it allows users to express parallel algorithms in a natural way by composing other, simpler parallel algorithms. This ability is more commonly known as *nested parallelism*, the "ability to take a parallel function and apply it over multiple instances in parallel." [30]

Sequential support for nested algorithm invocation is straightforward: appropriate state (e.g., registers) is saved, the call stack is initialized according to convention, and control is transferred to the target function until it returns. However, parallel programming models present a more challenging scenario. Nested parallel algorithm invocations must be efficiently mapped onto the processing elements while taking data locality into account. Furthermore, by definition, multiple such nested invocations occur concurrently, meaning a coordination of activities is required.

Supporting nested parallelism in both static and dynamic applications requires flexible support. Since the nature of the parallelism is not always known *a priori*, nested algorithms are usually implemented as a series of dynamically spawned tasks. In order to achieve clean algorithm expressivity, avoid user managed nested par-

allelism and provide reasonable performance, the mapping of these tasks must be handled by the runtime system.

If nested parallelism is combined with a PGAS/SPMD model then it provides the basis for maintaining the algorithm structure, allowing better control on how execution happens on an hierarchical machine. The user expresses her application as a composition of parallel algorithms on distributed data structures, providing this information to the lower levels all the way to the runtime, which can then help map each algorithm instance to the appropriate subset of the machine.

2.2.3 Differentiate between Work and Communication

Programs consist of algorithms invoked on data. While supporting both through the same mechanisms increases reuse and reduces complexity, putting both work and data under the same concept is challenging and can create performance or usability issues. It is important to maintain a distinction between communication and work, or communication tasks and computation tasks, respectively.

In order to provide a general model, any task can generate a task of any other type. However, each task type has its own characteristics. Computational tasks are governed by scheduling policies that are mandated by the algorithm, whereas communication tasks obey ordering rules, such as those described in Section 8, that allow users to reason about the order of reads and writes.

2.2.4 Asynchronous Remote Method Invocation

It is known that latency lags bandwidth [31], creating challenges when scaling an application to an increasing number of cores. Synchronous, or blocking, operations limit scalability by blocking progress on cores that wait for data to arrive. This fact makes latency one of the biggest obstacles to sustaining performance.

Asynchronous, or non-blocking, communication mechanisms have been proven

effective at offering a solution by hiding latency. While all software managed asynchrony has inherent overheads, its benefits outweigh its costs. Asynchronous communication primitives allow the overlap of computation and communication, thus providing latency hiding.

Asynchronous communication using a *Remote Method Invocation* (RMI) based model is an exciting approach. RMIs allow one to move data, work or both. Users can call arbitrary functions on remote targets asynchronously, allowing greater flexibility than merely asynchronous reads or writes. We will discuss more about the benefits of RMIs and we will address some of their drawbacks in Sections 3 and 6.

2.2.5 Implicit and Explicit Parallelism

Implicitly parallel models, such as those in HPF [32] and NESL [30], remove the burden of managing parallelism from the user, increasing productivity. User applications are written using high level algorithms and the rest of the stack takes care of data and work distribution, communication and synchronization. On the other hand, *explicitly parallel models* allow users more fine grain control over the applications, leading to greater scalability and performance, as demonstrated by the success of models such as MPI [33].

It is more beneficial if a runtime system offers an implicitly parallel model to the upper layers paired with an explicit data communication model. For example, algorithms always run in parallel and can choose when communication happens, while the runtime abstracts how it happens. It is up to the runtime to provide the necessary tools to assist algorithms and data structures in minimizing communication, offering an implicitly parallel model with explicit data movement to the end user.

2.3 Related Work

Many parallel programming languages and frameworks are supported by runtime systems that attempt to provide portable performance. We briefly present a number of them, saving a more direct comparison with our work in each individual chapter.

2.3.1 Shared Memory

The common characteristic of all runtime systems that support shared memory only is that communication is implicit; all threads have access to any object they are given access to, either implicitly or explicitly, without being required to go through a communication library. This is an intuitive programming model, however it relies on DSM techniques to be able to work on distributed memory, limiting its performance.

C++11 [34] offers thread creation and synchronization primitives, such as atomic types and mutexes. It also provides interfaces for creating tasks and retrieving values asynchronously, i.e., futures. It is a low level framework that focuses on concurrency rather than efficient parallel execution.

OpenMP [35] is built on the fork-join parallelism model [36] and is a set of directives and library routines that provide support for shared memory parallel programming in C, C++ and Fortran. Users annotate their sequential code with directives that the compiler uses to parallelize it, either by decomposing loops or by specifying tasks. While OpenMP has had nested parallelism capabilities since its inception and performance gains have been reported [37], the `collapse` keyword in OpenMP 3.0 that flattens nested parallel sections attests to the difficulty of gaining performance from nested parallelism in OpenMP. OpenMP focuses on computational tasks, since communication is implicit through shared memory. This approach has limited OpenMP to shared memory, as attempts to bring it to distributed memory had scalability issues, as demonstrated by Cluster OpenMP [38].

Cilk/Cilk++ [39, 40] are extensions of C and C++ respectively that provide primitives to create tasks, called *Cilk procedures*, executed by worker threads. Each Cilk procedure can create new procedures, thus supporting a form of nested parallelism. The runtime system implements work-stealing techniques and the work-first principle to adapt to run-time parameters. Silkroad [41] was an attempt to execute Cilk programs on distributed memory.

Intel Threading Building Blocks (TBB) [42] is a library for C++ that has similar nested parallelism and work-stealing properties as Cilk. It provides parallel algorithms and thread-safe containers with the runtime system providing a task parallel model. TBB tasks, a bundle of work and data, can be further partitioned in smaller tasks by the runtime system. TBB is also limited to shared memory and its task based, work-stealing runtime system does not retain the algorithm structure.

Habanero-Java [43], and its siblings Habanero-C and Habanero-C++, used in shared memory parallel programming, provide primitives for asynchronous function invocation, communication and synchronization. They all have work-stealing mechanisms and they extend the Cilk model with explicit task affinity control. Communication and computation are handled by the same primitives without a mechanism to differentiate between the two.

Table 2.1 summarizes the main characteristics of each presented language and library for shared memory parallelism.

2.3.2 Distributed Memory

Distributed memory only runtime systems are communication libraries that are designed to offer an abstraction layer for other, higher level libraries. While many have optimizations for intranode communication, they still retain their distributed character, forcing the user to copy data between the different address spaces of the

Name	Model	Nested Parallelism	Communication	Synchronization
OpenMP	Fork-join	Yes	Object sharing	Mutexes
C++11	MPMD	No	Object sharing, futures	Mutexes, futures
Cilk/Cilk++	MPMD	Yes	Object sharing	Continuations, Hyperobjects
TBB	MPMD	Yes	Object sharing	Continuations
Habanero-Java/C/C++	MPMD	No	Object sharing, futures	Continuations, Phasers

Table 2.1: Shared memory parallel languages and libraries

processing elements.

MPI (Message Passing Interface) [33] implementations are ubiquitous on all HPC platforms. They offer a wide range of primitives for message passing, both point-to-point and collective, and Remote Memory Access (RMA) operations [44], such as put, get and accumulate. MPI has support for nested parallelism through its subgroup and process spawning support. MPI was designed for data communication in distributed memory machines and while most implementations have optimizations for shared memory, they do not offer any control over the program execution.

Active Messages (AM) [45] is a library that provides message passing and allows one to specify a handler on the receiving process to process the message. AM was used as the basis for languages that offer a PGAS model, for example in Split-C [46].

ARMCI [47] and its successor ComEx [48] are libraries for RMA operations. They provide request aggregation dynamically at run-time and are configurable to fully utilize its platform's capabilities. Along with GASNet [49], they are used to offer a PGAS model to parallel frameworks, rather than being offered to end-users.

Table 2.2 summarizes the main characteristics of each presented library for distributed memory parallelism.

Name	Model	Subgroups	Communication	Collectives	Synchronization
MPI	SPMD	Yes	Message Passing, RDMA	Blocking / Non-blocking	Blocking / Non-blocking
GASNet	SPMD	No	RDMA	Proposed	Non-blocking
ARMCI	SPMD	No	RDMA	Blocking	Blocking
AM	MPMD	No	Active Messages	No	Non-blocking

Table 2.2: Distributed memory parallel libraries

2.3.3 Hybrid Shared and Distributed Memory

Hybrid shared and distributed memory runtime systems are usually built by combining a distributed memory runtime system that provides a PGAS model with a shared memory task parallel runtime system. The level of integration varies. There are frameworks that have a loose integration, such as Tpetra and Kokkos from the Trilinos package [50], in which Tpetra uses Kokkos, but the latter is unaware of the former. Others are more closely coupled, blurring the lines between shared and distributed memory and offering a single interface.

Charm++ [51] is a language based on C++ which provides a message-driven execution model. Messages invoke functions on *chares*, active objects with associated data that can be migrated automatically by the Charm++ runtime system, Converse. Charm++ allows users to associate multiple chares between them into collections, with each collection potentially representing a part of an hierarchical machine, offering some form of hierarchical mapping. Charm++ offers an MPMD model with a relaxed consistency model and does not distinguish between computation and communication tasks.

In Chapel (Cascade High Productivity Language) [52] a program executes on a number of *locales*, each with its own locality information and mapped to a level of the hierarchy (e.g., socket, core, etc.). Chapel provides a PGAS based model

called Asynchronous PGAS (APGAS) that permits only asynchronous operations. Each locale can spawn asynchronous tasks on any other locale, with the runtime system managing shared and distributed memory communication. Both computation and communication is performed through tasks and nested parallelism is supported through recursive task spawning. X10 [53], a Java-based language and predecessor to Habanero-Java, has a similar execution model.

HPX [54], an implementation of the ParalleX system in C++, shares the characteristics of Chapel and X10. It offers methods for creating lightweight tasks on specific threads, which are scheduled from the HPX system and provides an Active Global Address Space (AGAS) view, which is PGAS with the ability to move objects between physical addresses without having to update their virtual address.

UPC [28], UPC++ [55], Co-Array Fortran [29], and Titanium [56, 57] all provide an SPMD programming model with a PGAS view. All of them allow explicit affinity control, as they expose the locality of data, and they allow nested invocation of SPMD algorithms on a subset of the processing elements of the invoking algorithm.

Global Arrays [58] offers a programming model that resembles as much as possible that of shared memory models while being based on PGAS through ComEx [48]. It supports the creation of distributed arrays and uses RDMA for data transfers. Nested parallelism is not supported.

A few projects have been abandoned but they are worth mentioning for their contributions. Nexus [59] is a task parallel runtime system that provides *remote service requests*, essentially non-blocking Remote Procedure Calls (RPCs), for spawning tasks on processors, requiring the user to explicitly define affinity.

Split-C [46] is a parallel extension of C that uses the SPMD model and provides a PGAS view through AM [45]. Split-C supports nested parallelism, and provides data locality information but it targets regular applications.

Name	Model	Subgroups	Nested Parallelism	Consistency
Charm++	RMI	No	No	Weak
Chapel	APGAS	No	Yes	Weak
X10	APGAS	No	Yes	Weak / Sequential
UPC	PGAS	Proposed	Limited	Weak
Titanium / UPC++	RSPMD	Yes	Yes	Weak / Strict
HPX	MPMD	No	No	Weak
Global Arrays	PGAS	Yes	Yes	Weak with ordered loads/stores to overlapping addresses
Nexus	SPMD (RPC)	No	No	Platform dependent
Split-C	PGAS	No	No	Processor

Table 2.3: Hybrid distributed and shared memory parallel libraries

NESL [30] was the first language that supported expressing algorithms using nested parallelism. The user expresses her algorithm as a composition of other parallel algorithms. Since subgroup support is not offered, the NESL compiler performs flattening to transform the nested parallel algorithms into a flat data parallel model.

Table 2.3 summarizes the main characteristics of each presented library for hybrid shared and distributed memory parallelism.

3. THE STAPL RUNTIME SYSTEM

In Section 2 we have outlined the characteristics that a good runtime should exhibit. In this chapter we will present our approach for a portable, scalable, and efficient runtime system. One of the key design goals is portable performance: users must be able to write one version of the code that exhibits performance on different systems with no per platform optimization.

The *STAPL Runtime System* (STAPL-RTS) is an abstraction layer for parallel frameworks. It supports an SPMD execution model with task parallelism capabilities. For scalability and correctness, we employ a *distributed Remote Method Invocation* (RMI) model on distributed shared objects, called `p_objects`. The STAPL-RTS offers the same set of asynchronous primitives for communication over both shared and distributed memory, thus mitigating the effects of high memory latency and hiding architectural complexities.

Each processing element together with a logical address space forms an isolated computational unit called a *location*. Locations only have access to their own address space and communicate with other locations using RMIs on `p_objects`. This eliminates accidental data sharing and by extension, race conditions.

The STAPL-RTS presents a uniform communication interface that transparently employs both shared and distributed memory primitives, something that we refer to as *mixed-mode*. This approach is distinct from the standard hybrid (e.g., MPI+OpenMP) models, where a distinct shared memory implementation is maintained within a single process of the distributed program.

The scheduling of runnable tasks is handled by the EXECUTOR component. The EXECUTOR*s* present a task execution engine to the user and allow the scheduling of

runnable tasks with customizable scheduling policies.

Finally, the STAPL-RTS is written in the C++11 programming language [18] and only requires standard components, such as off-the-shelf compilers and established communication libraries such as MPI.

3.1 Execution Model

The goal of the STAPL-RTS is to support a scalable execution model that fulfills the requirements set in Section 2. Based on the positive experience at scaling applications using PGAS and SPMD models, we adopt and extend these models. In this section we describe several aspects of the execution model of the STAPL-RTS.

3.1.1 Execution Environment

Algorithms and applications built on top of the STAPL-RTS are executed always in SPMD sections. Each SPMD section is executed cooperatively by a set of *locations*.

Definition 1. *A* location *consists of a processing element (PE) combined with a virtual address space. This address space is logically isolated and cannot be directly accessed by other locations in user code.*

Each location is mapped to a PE and is not allowed to be migrated. When a location wishes to modify or read a remote location's memory, this has to be done through the appropriate STAPL-RTS functions, *even if the two locations reside in shared memory.*

Definition 2. *A* gang *is a collection of N locations with identifiers in the range $[0, \ldots, N-1]$ in which an SPMD section executes.*

Each gang has the necessary metadata for resolving location IDs to the corresponding PE. Two locations in the same gang cannot be mapped to the same PE. However, different gangs can have locations that are mapped to the same PE; gangs

are allowed to overlap and their locations are cooperatively scheduled on the PE that they are mapped to.

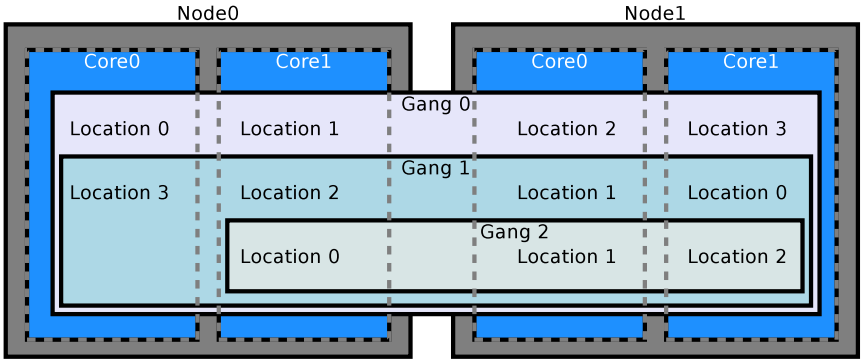


Figure 3.1: Execution example

Figure 3.1 shows an instance of a program in which gangs are overlapping over the same set of PEs. User code is unaware if a remote location is on the same PE, on shared memory or reachable only through distributed memory. Location naming is virtualized, further abstracting the application from the underlying platform.

3.1.2 Distributed Shared Memory

All memory accesses between locations are expressed via Remote Method Invocations (RMIs) on distributed objects. RMIs give us the ability to move work, data or both, offering a more flexible infrastructure than PGAS based DSMs that only focus on data transfer.

Definition 3. *A* p_object *is a distributed object defined over a set of locations. Each location owns a piece of the* `p_object`*, called a* representative*, and all the pieces are logically associated with each other to form the* `p_object`*.*

`p_object`s are created within a gang, and as such, each `p_object` is associated with exactly one gang and is distributed across its locations. A gang can have any number of `p_object`s. The gang metadata that resolve location IDs to PEs are used to direct RMIs to the right representative of a `p_object`, as well as a additional information for supporting collective operations.

This relationship between gangs and `p_object`s is what offers *isolation* and *virtualization* to higher level components, as they allow creating containers and invoking algorithms on a subset of the resources without algorithm or code modifications forming the basis of container composition and nested parallelism support. This interaction is further explored in Section 7.

The ability to perform RMIs on `p_object`s is the basis for distributed shared memory (DSM) communication. When a location wishes to modify or read a remote location's memory, the work must be expressed via RMIs on distributed `p_object`s, *even if the two locations reside in shared memory.* This means that *data races cannot occur*, as only one PE can directly access memory and RMI atomicity is guaranteed by the STAPL-RTS.

3.1.3 Asynchronous Communication Primitives

RMIs can be synchronous (blocking) or asynchronous (non-blocking). They are non-preemptive and are atomically executed as long as a *scheduling point* is not encountered.

Definition 4. *A* scheduling point *is a point in the execution of the instruction stream of user code where control is returned to the* STAPL-RTS*. This happens when* STAPL-RTS *primitives are called, such as for example blocking while waiting for a value or invoking an RMI.*

Asynchrony allows us to minimize the effects of high latency by enabling com-

munication and computation overlapping. We further desire to minimize any state associated with the RMI from the initiating location after invoking the non-blocking RMI, allowing the location to proceed with other, potentially unrelated tasks while it awaits any return value.

We facilitate the use of asynchronous RMIs by providing *RMI argument copy semantics*. We enforce pass-by-value semantics for all arguments passed to an RMI. A private copy of any argument passed to a remote function call is presented to the receiver; any mutation on the argument either at the sender or at the receiver will not be visible to the other.

3.1.4 Causal RMI Ordering

In order to present a coherent model, RMIs are *causally ordered*. A happened-before relationship is established between RMIs that are invoked from the same source location to the same destination location if they are issued in the same context without requiring extra synchronization. These ordering guarantees may be stricter than required by some algorithms, making them a good candidate for application driven optimization. Causal RMI ordering is discussed in Section 8.

3.1.5 Customizable Execution

RMIs are able to move work, data or both, but they have to respect causal ordering and lack the ability for user configurable scheduling. In order to enhance the execution capabilities, we offer the ability to schedule runnable *computational tasks* with arbitrary scheduling policies. The STAPL-RTS provides the necessary interfaces to create and schedule tasks for execution on a per location basis.

- RMIs are single threaded and execute in the SPMD section of their target `p_object`. Their execution order is mandated by the causal ordering and arbitrary scheduling is not allowed.

While RMIs can be used to perform computation, their execution can be interrupted by other RMIs at scheduling points with negative performance impact, e.g., because of cache eviction if they operate on different data. As such, RMIs are used for relatively short-lived operations, such as data read, write and update operations.

Collective operations, as well as `p_object` creation, are not allowed while executing RMIs.

- Computational tasks executed in their own SPMD section and are concurrent, allowing to specify user defined, arbitrary scheduling policies.

  When a task is declared runnable, e.g., by a task dependence graph built on top of the STAPL-RTS, appropriate scheduling information is passed along to place the task in a location specific task queue called the *EXECUTOR*. Although concurrent, tasks execute atomically; another task cannot preempt an executing task. This reduces potentially negative performance implications.

  Typically, computational tasks perform lengthier operations than RMIs and since they operate in their own SPMD section, they are allowed to make collective calls and to create `p_object`s.

Despite their differences, tasks and RMIs share a lot of commonalities. There are no restrictions regarding the code that they can contain and both are allowed to make blocking and non-blocking RMIs to `p_object`s they have access to.

3.2 Component Overview

The runtime system features a highly modular design, depicted in Figure 3.2*, that allows it to be customized and tuned as needed for different platforms.

*White text signifies components that are user accessible, black text marks components for internal use only.

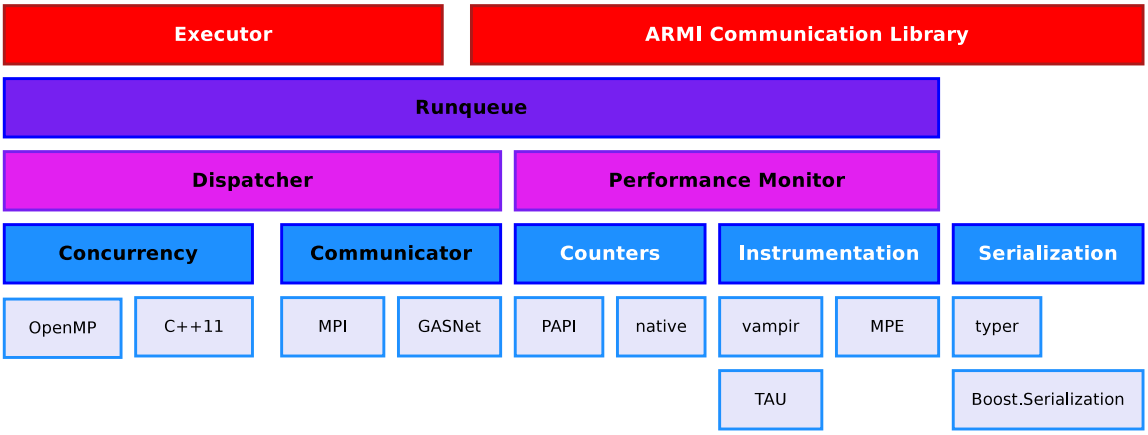


Figure 3.2: STAPL-RTS components

3.2.1 Adaptive Remote Method Invocation (ARMI)

Adaptive Remote Method Invocation (ARMI) [60] provides primitives for creating `p_objects` and invoking RMIs on them. ARMI makes use of *future* and *promise* objects [61] to allow the asynchronous return of values from RMIs. ARMI is the basis for building distributed containers for storing and accessing data and task dependence graphs to perform the computation.

3.2.2 EXECUTOR

The EXECUTOR allows users to schedule runnable tasks for execution with associated scheduling information. Scheduling is influenced by user-defined scheduling policies, such as First In First Out (FIFO), priority-based scheduling and others.

Work-stealing is also supported [62] through a work-stealing scheduler and various stealing policies. Finally, the EXECUTOR framework is complemented by the *terminators*, objects that can be closely tied with task dependence graphs and EXECUTORs and decide when an algorithm has finished executing (terminated).

3.2.3 Runqueue and Dispatcher

The *runqueue* and the *dispatcher* are responsible for executing tasks and RMIs. They are responsible for passing RMI requests to the right communication channel, e.g., communication libraries for distributed memory communication or to the multithreading library for shared memory communication. The runqueue has a list of all pending RMIs per location, while the dispatcher is responsible for giving control to the RMI selected for execution. The runqueue and the dispatcher cooperate with the EXECUTORs so that there is coordinated execution of tasks and RMIs.

Closely coupled with the dispatcher are the *concurrency* component which abstracts the threading capabilities of the platform, offering functions to create threads and it exposes the computing resources that are available, for example the number of cores and the processing element hierarchy. Currently, two back-ends are offered, one based on the C++11 thread support [34] and one that is built on top of OpenMP [35].

Finally, communication between different address spaces is achieved through the *communicator* component, which is a low-level distributed memory communication layer wrapper. It offers point-to-point, collective and multicast communication capabilities and currently it uses MPI as its back-end.

3.2.4 Performance Monitoring

The *performance monitoring* module consists of various independent components that are related to measuring run-time variables. Its basis is the *counters* component that offers high-level interfaces to the platform's native counters and timers, such as Linux timers or PAPI [63] and even energy consumption on machines that provide this level of information.

The *instrumentation* component is responsible for the tracing and profiling capabilities through the integration of libraries such as TAU [64] and MPE [65]. ARMI

and EXECUTORs are annotated with compile-time enabled instrumentation calls that make callbacks to supported third party libraries (TAU, MPE and others) and allow users to understand the behavior of their application.

3.2.5 Serialization

The *serialization* module is an independent module that provides C++ object marshalling capabilities for communication or storage. The STAPL-RTS relies on the serialization module both for internal object marshalling, as well as user defined data structure marshalling.

The `typer` [60] is our approach to serialization through intrusive marshalling. Some of the functionality offered is:

- automatic support for basic types, such as empty classes, primitive types, and plain old data structures (PODs),
- arbitrary object support via a per-class user-defined function (`define_type`), that enables marshalling for objects with pointers, inheritance and members that should not be packed, but rather default constructed for each object instance (transient members), and
- the ability to decide if an object can introduce data races, e.g., communicating an `std::shared_ptr` through shared memory, something that is used in zero-copy described in Section 6.

Figure 3.3 shows an example of providing serialization support for a simple vector-like class. The marshalling mechanism relies on the `define_type` functions to recursively traverse the object structure and is similar to PUP from Charm++ [51]. Each statement in the `define_type` is evaluated in a depth-first manner, until a basic type is encountered, in which the recursion stops. The `typer` is responsible for

```
1  template<typename T>
2  class vector {
3    std::size_t m_size;
4    T*          m_data;
5
6    // constructors, accessors, etc.
7
8  public:
9    void define_type(stapl::typer& t) {
10     t.member(m_size);
11     t.member(m_data, m_size);
12   }
13 };
```

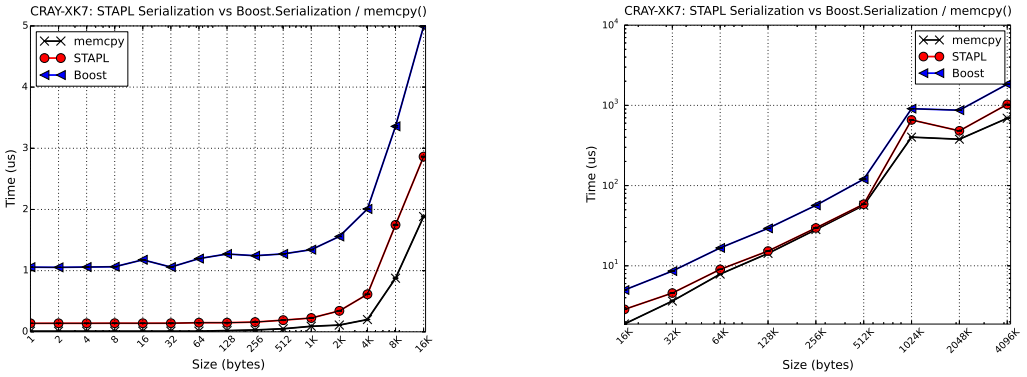
Figure 3.3: Marshaling example

finding the required size for an object, packing and unpacking it, relying on `memcpy` for the simple types and using the additional information from the `define_type` to fix pointers for complex types.

As mentioned, the `typer` provides intrusive marshalling. Sometimes this is not desired. For example, an already good marshalling solution exists through a third-party library or an intrusive solution is not possible, as the user has no access to the class code. For these reasons, the serialization component is extensible and supports seamless integration with other marshalling libraries, such as Boost.Serialization [66]. The STAPL-RTS automatically tries to fallback to Boost.Serialization if a suitable STAPL-RTS-based marshalling method has not been defined.

We compare the `typer`-based serialization of the example data structure from Figure 3.3 against Boost.Serialization and `memcpy` of a C array with the same number of elements on an AMD Opteron 6272 Interlagos processor[†] in Figure 3.4. The `typer`-based serialization, while slower than `memcpy` of a C array, it is significantly faster than Boost.Serialization, a fact that is attributed to the intrusive mechanisms that the `typer` uses. While both the `typer` and Boost.Serialization have to serialize the

[†]More information about the experimental set-up can be found in Section 4.



(a) Small objects (b) Large objects (log y-axis)

Figure 3.4: STAPL-RTS serialization cost vs `memcpy()` and Boost.Serialization

data structure into a buffer, at deserialization the `typer` can unpack the object by fixing the internal pointers of the data structure, effectively performing an *in place* unpacking, whereas Boost.Serialization has to default construct a new object and fill it with the serialized values.

3.3 `p_object`s

As mentioned, a `p_object` is a distributed object defined on a set of locations, each of which has a local representative of the object. Each `p_object` is identified by an `rmi_handle` which acts as its virtual address. A representative of a `p_object` is identified by the `rmi_handle` of the `p_object` it belongs to and the ID of the location it lives on. An `rmi_handle` is generated programmatically through the creation of a `stapl::rmi_handle` object.

3.3.1 Virtual Addressing

Constructing a `p_object` generates a virtual address unique in the system, the `rmi_handle`. This virtual address is shared among all the locations that participate in the `p_object` construction and has the necessary information for accessing the `p_object` using RMIs.

Typically, an `rmi_handle` is heavier than a C++ pointer. It consists of the ID of the gang the `p_object` was created in and an integral ID unique in the gang. Additional information includes flags that modify the behavior of the `p_object`, e.g., if communication is allowed to be aggregated or not. For gangs with 1 location, the integral ID is the same as the physical address of the `p_object`, an optimization that avoids some of the overhead inherent to generating and storing `rmi_handle`s.

During communication, any C++ pointers and references to `p_object`s are automatically converted to `rmi_handle`s at the sender and back to physical address at the receiver. This is managed by the STAPL-RTS and the serialization module. Depending on the requirements, two different internal representations of an `rmi_handle` are used:

- the `stapl::rmi_handle::reference` is a complete virtual address that allows both translation to the physical address of a `p_object` to retrieve the representative on a location and allows RMI communication, and
- the `stapl::rmi_handle::light_reference` which is a partial virtual address with a smaller space footprint that only allows translation to physical address.

3.3.2 Construction

We support two ways of declaring an object as `p_object` as shown in Figure 3.5:

- by having a member variable `stapl::rmi_handle` in the class that associates the object with an `rmi_handle` and providing the function `get_rmi_handle` to retrieve the handle or
- extending from the `stapl::p_object` class that itself uses an `rmi_handle` internally.

```
1  // using stapl::rmi_handle
2  class A {
3    stapl::rmi_handle m_handle;
4
5  public:
6    A(...) { }
7
8    stapl::rmi_handle::const_reference get_rmi_handle() const { return m_handle; }
9
10   stapl::rmi_handle::reference get_rmi_handle() { return m_handle; }
11 };
12
13 // using stapl::p_object
14 class B
15 : public stapl::p_object
16 {
17 public:
18   B(...) { }
19 };
20
21 stapl::error_code stapl_main(int, char**) {
22   A a;                          // create instance of A
23   B b;                          // create instance of B
24   std::unique_ptr<B> p{new B}; // create instance of B on the heap
25   ...
26   return EXIT_SUCCESS;
27 }
```

Figure 3.5: `p_object` declaration

For both methods, when a new instance of `A` or `B` is created, it is automatically registered with the STAPL-RTS, creating a virtual address that can be used for communication. By extending from the `stapl::p_object` class, the user is provided with additional capabilities such as copy and move constructors that handle registration automatically and support for polymorphic type hierarchies.

Currently, we require SPMD creation of `p_object`s on all the locations of a gang as described in [60]. `stapl_main` is the application entry point and executes on a number of locations, acting as the primordial SPMD section. Users can create `p_object`s as shown in Figure 3.5. `p_object` construction is a collective, SPMD operation in which each location is responsible for constructing its representative of the `p_object`. It is worth noting that during construction no communication is required in the STAPL-RTS to guarantee registration.

```
1  // Create a p_object of type A by passing args to the constructor, in a new gang
       over locations 0,2,3 and return a future to its handle
2  stapl::future<stapl::rmi_handle::reference> f1 =
3    stapl::construct<A>(stapl::location_range({0, 2, 3}), args...);
4
5  // Get object handle
6  auto h = f1.get();
7
8  // Create a p_object of type B in a new gang that fully overlaps with the gang of h
9  stapl::future<stapl::rmi_handle::reference> f2 =
10   stapl::construct<B>(h, stapl::all_locations, args...);
11
12 // Delete first object
13 stapl::p_object_deleter<A> d;
14 d(h);
```

Figure 3.6: Asynchronous, one-sided section creation

An alternative way of creating p_objects is via an asynchronous, one-sided mechanism. The construct primitive, an example of which is shown in Figure 3.6, creates a new gang over a set of resources and constructs a new p_object in it. Multiple variations are supported, such as creating sections on arbitrary ranges of locations (or all) of either the current parallel section or that of another p_object.

The STAPL-RTS is responsible for translating the virtualized specification of resources, i.e., a range of location IDs, to PEs and for building a suitable broadcast tree on the PEs which it uses to construct the associated p_object. The return of construct is always a future object, similar to C++11 futures [34], that allows consumption of a return value from an asynchronous function; see also Section 3.4.1. This asynchronous mechanism is the basis for providing nested parallelism for irregular applications and will be presented more in depth in Section 7.

3.3.3 Destruction

Similarly to construction, destruction of p_objects is also an SPMD operation. However, unlike construction, the destruction order of p_objects can vary between locations. This has the obvious advantage that p_objects can be deleted one-sided

using RMIs, something that would not be possible with SPMD unregistration.

We achieve this by extending the work in [60] through allowing `p_objects` to unregister in any order on each location. When a representative of a `p_object` is deleted, then the associated `rmi_handle` is added to a location specific *unregistration reorder buffer*. The `rmi_handle` is not reused until the reorder buffer is committed.

We release the `rmi_handles` in the same lexicographical order on all locations only at the next synchronization point. This unregistration mechanism requires no communication but it still requires that all locations have to delete the same `p_objects` in between two subsequent synchronization points. This requirement can be relaxed by assigning a location responsible for a block of `rmi_handles` and releasing an `rmi_handle` for reuse only when all the representatives of the `p_object` have been destroyed, avoiding the reorder buffer. We plan to explore this option in the future.

To complement the one-sided construction through `construct`, an asynchronous, one-sided destruction mechanism is offered. The `p_object_deleter` follows the concept of the `std::deleter` [34] and allows to call the destructor and release the memory for `p_objects` that are either heap allocated or created through a `construct` call. The `p_object_deleter` is shown in Figure 3.6.

3.4 ARMI

Adaptive Remote Method Invocation (ARMI) provides a unified communication model to users based on RMIs on `p_objects`. It transparently employs both shared and distributed memory primitives, something that we refer to as *mixed-mode*. This approach is distinct from the standard hybrid (e.g., MPI+OpenMP) models, where a distinct shared memory implementation is maintained within a single process of the distributed program.

Primitive	Description
One-Sided Primitives	
`void async_rmi(dest, h, f, args...)`	Issues an RMI that calls member function `f` of the `p_object` associated with the `rmi_handle` `h` on location `dest` with the given arguments, ignoring the return value. Synchronization calls or other RMI requests that do not ignore the return value can be used to guarantee its completion.
`future<Rtn> opaque_rmi(dest, h, f, args...)`	Calls `f` and returns a `future` object for retrieving the return value of `f`.
`Rtn sync_rmi(dest, h, f, args...)`	Calls `f` and waits for the return value (blocking primitive).
`void try_rmi(dest, h, f, args...)`	Issues an asynchronous RMI that calls `f` iff the target `p_object` is still alive; otherwise it is safely ignored. Can be used for data prefetching.
One-sided Collective Primitives	
`void async_rmi(all_locations, h, f, args...)`	Calls `f` of the `p_object` associated with `h` on all of the locations it exists on.
`futures<Rtn> opaque_rmi(h, f, args...)`	Calls `f` on all locations of `h` and returns a `futures` object to retrieve the return value from each location.
`future<Rtn> reduce_rmi(op, h, f, args...)`	Calls `f` and returns the result of the reduction using operator `op` when applied to the return values.
Collective Primitives	
`futures<Rtn> allgather_rmi(h, f, args...)`	Collectively calls `f` on all locations of `h`. The return values are retrieved through the `futures` object.
`future<Rtn> allreduce_rmi(op, h, f, args...)`	Collectively calls `f` and returns the result of the reduction using operator `op` when applied to the return values.
`future<Rtn> broadcast_rmi(h, f, args...)`	Caller (root) location calls `f` and broadcasts the return value to all other locations. Non-root locations have to call `broadcast_rmi(root, f)` to receive the value.
Synchronization Primitives	
`void rmi_fence()`	Guarantees that all invoked RMI requests have been processed using an algorithm similar to [67].
`void rmi_barrier()`	Performs a barrier operation.
`void p_object::advance_epoch()`	Advances the epoch of the `p_object`, as well as the epoch of the location. It can be used for synchronization without communication, avoiding the `rmi_fence()` or `rmi_barrier()` primitives.
Information Primitives	
`location_id get_location_id()`	Returns the ID of the calling location.
`location_id get_num_locations()`	Returns the number of locations in the gang of the calling location.

Table 3.1: ARMI primitives

A synopsis of the current interface can be found in Table 3.1[‡]. We have substantially extended the API from [60] and all our primitives, apart from `sync_rmi`, are asynchronous (non-blocking). Asynchrony allows us to minimize the effects of high latency by enabling communication and computation overlapping. In Section 3.4.1 we show how asynchronous RMI return values are handled without requiring blocking, allowing users to move on to another computation while awaiting said value.

The asynchronous operation is possible because of the RMI argument copy semantics. Any argument passed to a remote function call is a private copy of the receiver; any mutation on the argument either at the sender or at the receiver will not be visible to the other. Copy semantics simplify reasoning about parallel programs, as they remove potential side-effects, but they can affect performance by adding unnecessary copying, something that will be addressed in Section 6.

References and pointers to `p_objects` are translated automatically to reference the representative of the `p_object` at the receiver. If the `p_object` has no representative at the receiver, an error is raised.

ARMI primitives are divided in four categories:

- **point-to-point**, where the communication is performed between two endpoints (locations),
- **one-sided collectives**, one-to-many communication patterns, in which one source location invokes an RMI to multiple destination locations,
- **collectives**, in which all locations of a gang participate in the RMI,
- **synchronization** that provide guarantees regarding the state of SPMD execution and RMI execution, and

[‡]All primitives exist in the `stapl` namespace that is omitted for brevity.

```
1  struct A : public stapl::p_object {
2   int m_value;
3   void write(int t) { m_value = t; }
4   int read() const { return m_value; }
5  };
6
7  foo(...) {
8    A a;
9    auto h = a.get_rmi_handle();
10   int t = 5;
11   stapl::async_rmi(1, h, &A::write, t);
12   t = 6;
13   stapl::future<int> f = stapl::opaque_rmi(1, h, &A::read);
14   int y = f.get();
15
16   assert(y==5); // guaranteed by RMI argument copy semantics
17 }
```

Figure 3.7: Basic usage of ARMI primitives

- **information** that provide information about the execution environment and the system.

In Figure 3.7 we give an example of ARMI usage. Function `foo` is executing on a location which wishes to communicate with location 1. The shared `p_object` `a` is accessed through a handle `h`, which represents the distributed object with a representative on the destination. The corresponding instance of `a` on location 1 is updated via a call to `A::write`. Note that pass by value semantics guarantee that the callee sees 5 and not 6. Also, assuming that no other locations send updates to location 1, y will be set to 6 since the ordering of RMI invocations from a single source is enforced by default according to the guarantees described in Section 8.

A more complete example use of the primitives with collectives and synchronization is shown in Figure A.2 in Appendix A.1. It compares a 1-D Jacobi solver in hybrid MPI+OpenMP against an ARMI-based one, showing that the unified communication interface of the STAPL-RTS provides a simpler programming model than that of the dual interfaces required to implement the hybrid MPI and OpenMP version.

3.4.1 Future / Promises

```
template<typename T>
class promise {
public:
  // Sets the result
  void set_value(T const&);

  // Returns a future to retrieve the
    result
  future<T> get_future();
};
```

(a) `stapl::promise`

```
template<typename T>
class future {
public:
  // Checks if the result is available
  bool valid() const;

  // Waits for the result to become
    available
  void wait() const;

  // Returns the result
  T get();

  // Invokes f when the result becomes
    available
  template<typename F>
  void async_then(F&& f);
};
```

(b) `stapl::future`

```
template<typename T>
class futures {
public:
  // Returns the number of expected
    resutlts
  std::size_t size() const;

  // Checks if all results are
    available
  bool valid() const;

  // Checks if the n-th result is
    available
  bool valid(std::size_t n) const;

  // Waits for all results to become
    available
  void wait() const;

  // Waits for the n-th result to
    become available
  void wait(std::size_t n) const;

  // Returns all results
  std::vector<T> get();

  // Returns the n-th result
  T get(std::size_t n);

  // Invokes f when all results become
    available
  template<typename F>
  void async_then(F&& f);
};
```

(c) `stapl::futures`

Figure 3.8: Asynchronous value retrieval interfaces

The RMI interfaces in Table 3.1 offer the ability to write data (*put operations*) through the supplied arguments. For asynchronous *get operations* we draw inspiration from the future / promise mechanisms [61]. We have modeled our future/promise support on C++11 offerings [18] and their interface is presented in Figure 3.8.

A *future* is a mechanism to retrieve the result of an asynchronous primitive that does not ignore the result of the invoked function, e.g., `opaque_rmi`. The *promise*

```
stapl::future<T> send_request(...) {
  stapl::future<T> f = stapl::opaque_rmi(0, h, &A::get_value);
  return f;
}

process_request(T& t, stapl::future<T> f) {
  if (f.valid()) {
    t = f.get();
    return true;
  }
  return false;
}

stapl_main(...) {
  T t;
  stapl::future<T> f = send_request();
  while (!process_request(t,f))
  { ... // perform other work }
  foo(t); // use t;
}
```

(a) Example `stapl::future` and `stapl::promise` usage

```
send_request(...) {
   future<T> f1 = opaque_rmi(1, h, &A::get_value)
   f1.async_then([](future<T> f2) { foo(f2.get()); });
}

stapl_main(...) {
  send_request();
  ... // proceed with other work
};
```

(b) Example `stapl::future::async_then` usage

Figure 3.9: Asynchronous value retrieval examples

is a placeholder for an incoming value, which can be set at the end of complex, multi-hop communication patterns; the value is retrieved through a `future` object. For collective operations, the `futures` object extends future support by providing interfaces to retrieve multiple values.

A usage example of our API that highlights our asynchronous primitives and our future/promise support is shown in Figure 3.9. While C++ versions are for shared memory, our implementation provides similar semantics transparently in distributed memory without any additional intervention from users. The promise/future mechanisms provide a standard idiom to facilitate *gets*, enable optimizations, e.g., zero

copy, and delegate responsibility for receiving the return value from the RMI to code outside the calling context. One example usage is shown Figure 3.9(a).

This trivial example shows how a return value from an RMI can be handled outside the context of the RMI invocation. A different computational activity can occur while waiting for the internal STAPL-RTS `promise` associated with the `future` to be fulfilled. We also support continuations on future objects through the `future::async_then` function [68], an extension that has been proposed for C++17 [69]. Again, we follow the proposed interface, but provide our own implementation that provides a uniform interface for both shared and distributed memory. Together with a lambda expression, this feature is used to refine the previous example as shown in Figure 3.9(b). In this case, the consuming function of the RMI return value is specified at the RMI call site, and will be called by STAPL-RTS when the corresponding promise is fulfilled. Other local computation proceeds immediately after the initial RMI request is made.

3.4.2 Synchronization

In our previous work [60], we only supported the `rmi_fence` primitive, that ensures that all pending RMI requests prior its call have been processed. For an asynchronous system, this is a very strict operation that can limit scalability.

For that reason, we have implemented an *epoch* support in our framework to provide a cheap, communication-less synchronization mechanism. Our implementation relies on logical clocks [70]. Each location has a local epoch counter and `p_objects` are associated with the epoch of the gang they are created in. By default, creating a `p_object` always advances the epoch, therefore each `rmi_handle` is associated with a specific epoch.

Each location advances the epoch in an SPMD way. This happens implicitly

```
1  struct A : public stapl::p_object {
2    int m_value;
3    void write(int t) { m_value = t; }
4    int read() const { return m_value; }
5  };
6
7  foo(...) {
8    A a;
9    auto h = a.get_rmi_handle();
10
11   a.write(5);
12
13   auto f1 = stapl::opaque_rmi(1, h, &A::read);
14   int y1  = f.get();
15   // assert(y1, 5); unclear if the a.write(5) on the destination has been executed
16
17   a.advance_epoch();
18
19   auto f2 = stapl::opaque_rmi(2, h, &A::read);
20   int y2  = f2.get();
21   assert(y2, 5); // when opaque_rmi executes,
22                  // it is guaranteed that a.write(5) has finished
23 }
```

Figure 3.10: Epoch guarantees

when the STAPL-RTS has to guarantee that `p_object`s are in a consistent state, for example when registering them or after an `rmi_fence`, or explicitly through the `p_object::advance_epoch`.

The synchronization guarantees are achieved through checking the epochs of the incoming RMIs and that of the location and deciding if the RMI can be executed. RMI requests that arrive from the same or past epoch are allowed to be processed, while RMI requests coming from a future epoch are being deferred until the location advances the epoch. Relying on this mechanism frees the user from always to have to guarantee RMI execution through `rmi_fence` calls and promotes a more asynchronous model.

Figure 3.10 shows an example use of the epoch support. All locations create the `p_object` a and then make an `opaque_rmi` call to retrieve the value from location 1. However, there is no guarantee that `A::write` has been executed before the arrival

of any of the RMIs, except for location 1. Calling `p_object::advance_epoch` guarantees that for any RMI that arrives from a future epoch (i.e., the `opaque_rmis`), it's execution will deferred until the epoch is advanced locally.

3.5 Interoperability

Software reuse is crucial for HPC applications. To improve productivity and performance, it is essential to allow the use existing libraries that have been fine-tuned, rather than reinvent them. The STAPL-RTS was designed to be interoperable with other libraries, either through providing the appropriate interfaces to call external libraries or allowing the STAPL-RTS to be called from other applications.

3.5.1 Calling Legacy Code

Shared Memory libraries. The STAPL-RTS can use other multithreaded libraries transparently, e.g., fftw [71], since the latter operate on data in shared memory and are assumed to access data in a safe manner.

For the cases that a library need to call ARMI primitives, the STAPL-RTS has to be notified about the threads of the legacy code. The `external_thread` object notifies of the existence of an externally managed thread that requires to call STAPL-RTS primitives. Figure 3.11 shows a small example of how code in an OpenMP section could invoke RMIs on a `p_object` safely.

Distributed Memory libraries. For distributed memory code, e.g., libraries that use MPI internally, a different approach is used. We offer the `external_call` function that halts the execution of an application in a consistent state and transfers control to the the third party library.

`external_call` waits for all communication to quiesce and all pending RMIs to finish. It then disables the communication layer so that it cannot be used through any ARMI primitives. Finally, it elects one location per process to act as the leader

```
1  struct A : public stapl::p_object {
2    void f(...) { ... }
3  };
4
5  stapl::error_code stapl_main(int, char**) {
6    A a;
7
8    // create OpenMP parallel section
9  #pragma omp parallel shared(a)
10   {
11     // inform STAPL-RTS that this is a thread from an external library
12     stapl::external_thread t;
13
14     // accessing a directly managed by OpenMP
15 #pragma omp critical
16     a.f(...);
17
18     // calling RMIs managed by STAPL-RTS
19     stapl::async_rmi(..., a.get_rmi_handle(), &A::f, ...);
20     ...
21     stapl::rmi_fence();
22   }
23   return EXIT_SUCCESS;
24 }
```

Figure 3.11: Calling RMIs from legacy code

for calling the external library; the rest of the locations block waiting for the leader to finish. Since the STAPL-RTS is in a halted state, the user is responsible for transferring all the data to the leader and ensuring that no ARMI primitives will be called.

In Figure 3.12 user code is calling MPI to perform a blocking `MPI_Allreduce` that is called from one location per process. The `external_call` has been used successfully by applications built on top of STAPL-RTS, such as the Graph 500 implementation of the SGL [12].

3.5.2 Integrating with Legacy Code

To promote adoption of libraries built on top of the STAPL-RTS and combine them with existing libraries, the STAPL-RTS can be initialized and invoked from existing applications as another library. In this case, the STAPL-RTS is being driven by the application, rather than being the driver.

```
std::pair<int,int> call_mpi_allreduce(int i)
{
  int j = i;
  MPI_Allreduce(MPI_IN_PLACE, &j, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

  int size = MPI_PROC_NULL;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  return std::make_pair(size, i);
}


stapl::error_code stapl_main(int, char**) {

  // associative container with all the leaders
  const auto leaders = external_callers();

  // transfer control to external library; only the leader will make the call
  auto p = external_call(&call_mpi_allreduce, 1);

  if (c.find(stapl::get_location_id())!=c.end()) {
    // this location is a leader, value is valid
    assert(bool(p));

    // each leader contributes 1, total result is the number of MPI processes
    assert(p->first==p->second);
  }
  else {
    // this location is not a leader, does not have a value
    assert(bool(p));
  }

  return EXIT_SUCCESS;
}
```

Figure 3.12: Invoking distributed memory code

```
int main(int argc, char* argv[]) {
  MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, ...);

  // other MPI code

  // options for STAPL-RTS
  auto opt = stapl::option{argc, argv} & stapl::option{ MPI_Comm  , comm};

  // initialization of STAPL-RTS and execution of function
  stapl::initialize(opts);
  stapl::execute(pdt_entry_point_wf{my_generated_data});
  stapl::finalize();

  // other MPI code

  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

Figure 3.13: Invoking STAPL-RTS from legacy code

Figure 3.13 shows a simplified code example from PDT [72], a parallel particle transport code. MPI is initialized and used to initialize the application data structures. The STAPL-RTS is given an arbitrary MPI communicator and some work to do. Once the work is done, the STAPL-RTS finalizes itself and returns control to `main`.

4. PERFORMANCE EVALUATION

In this chapter, we describe the methodology used to evaluate the performance of components for the STAPL-RTS and the cooperation between the STAPL-RTS and frameworks build on top of it, such as STAPL [9]. We will look at the performance using microbenchmarks, kernels, and real-world applications.

4.1 Experimental Setup

We conducted our experimental studies on various parallel machines comprising various processor architectures and network interconnects.

CRAY-XK7. This is a Cray XK7m-200 system which consists of twenty-four compute nodes with AMD Opteron 6272 Interlagos 16-core processors at 2.1 GHz. Twelve of the nodes are single socket with 32 GB of memory, and the remaining twelve are dual socket nodes with 64 GB. Our codes have been compiled with `gcc` 4.9.1.

IBM-BG/Q. This IBM BG/Q system available at Lawrence Livermore National Laboratory has 24,576 nodes. Each node is populated by a 16-core IBM PowerPC A2 processor clocked at 1.6 GHz and 16 GB of memory. The compiler was `gcc` 4.8.4.

X86-CLUSTER. This machine is an `x86`-based commodity cluster that consists of 311 nodes with different processor and memory configurations. The slice of the system that we used for our experiments is 128 nodes. Each node has two AMD Opteron 2350 2.5 GHz processors, with each processor having 4 cores, for a total of 8 cores and 32 GB per node. We used `gcc` 4.8.2.

4.2 Benchmarks

The STAPL-RTS is evaluated both in isolation and as used by other frameworks. Whenever possible, we choose to implement established benchmarks that are well

known and understood, modified to fit the STAPL-RTS programming model.

4.2.1 Microbenchmarks

Microbenchmarks are small programs that target a specific component or primitive. They are artificial benchmarks that attempt to evaluate the performance of a small part of the STAPL-RTS in isolation and set the bounds on expected performance. Their added benefit is that their results can be used with an appropriate parallel computation model such as LogP [73] to derive a *machine model*.

Due to their nature, microbenchmarks have often execution restrictions, e.g., being able to used only on specific number of PEs. Our microbenchmarks include adaptations of the Ohio State University Microbenchmarks (OMB) [74] and the set of benchmarks presented in [75].

4.2.2 Kernels

Kernels are benchmarks that abstract common computational or communication patterns of real world applications. Kernels are considerably smaller than the applications that are based on but exhibit similar characteristics. In this work, we focus mainly on computation kernels implemented either directly using ARMI or using STAPL, such as the NAS parallel benchmarks [76] and Graph 500 [77].

4.2.3 Applications

Finally, the STAPL-RTS is evaluated using real world applications built on top of STAPL. The latter is an advanced parallel framework that takes advantage of the STAPL-RTS functionality without exposing the latter to the user, proving that the STAPL-RTS is a runtime system that can offer performance, without sacrificing portability or ease-of-use.

5. STAPL OVERVIEW

Before continuing with the presentation of our work, we will present a framework that adopts the execution model presented in Section 3.1 and that is using the STAPL-RTS as its platform abstraction layer.

The *Standard Template Adaptive Parallel Library* (STAPL) [9] is a framework developed in C++ for parallel programming. STAPL is a library, requiring only a C++ compiler (e.g., `gcc`) and uses the STAPL-RTS for expressing communication and computation. An overview of its major components are presented in Figure 5.1. The generic design of STAPL is based on that of the C++ Standard Template Library (STL) [17], extended and modified for parallel programming.

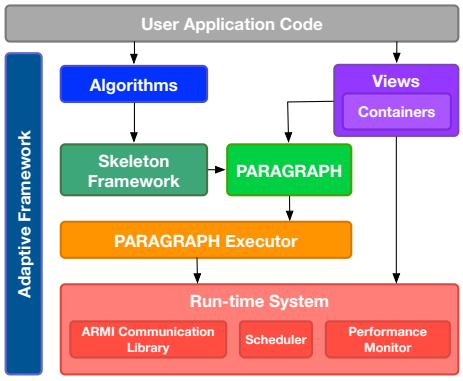


Figure 5.1: STAPL components

STAPL provides *parallel algorithms* and *distributed data structures* [11, 12] with interfaces similar to the STL. Instead of using iterators, algorithms are written with *views* [10] that decouple the container interfaces from the underlying storage. The *skeletons framework* [14, 16] allows the user to express an application as a composi-

tion of simpler parallel patterns (e.g., map, reduce, scan and others).

Algorithmic skeletons are instantiated at runtime as task dependence graphs by the `PARAGRAPH`, STAPL's data flow engine. It enforces the specified task dependencies and is responsible for the transmission of intermediate values between tasks.

The *STAPL Runtime System* (STAPL-RTS) [78, 15, 79], the focus of this dissertation, provides portable performance by abstracting the underlying platform with the concept of *locations* as explained in Section 3. The STAPL-RTS abstracts the platform and its resources, providing a uniform interface for communication and computation.

Throughout this dissertation, we will explain how STAPL components are using the STAPL-RTS for creating user friendly, portable, and efficient parallel applications.

5.1 Containers and Views

STAPL containers are distributed data structures that offer a shared memory inspired interface. They have interfaces similar to their STL counterparts for accessing and mutating stored data and metadata, e.g., size of the container, distribution and others. They are extensible and composable through regular C++ inheritance and template instantiation mechanisms.

Various containers are offered that have similar characteristics as the STL containers they model, such as `array`, `vector`, `map`, `set`, `unordered_map`, `unordered_set`, `list`, etc. There are also containers that are not to be found in STL such the `matrix` [7] and the `graph` [12].

The container consists of the distribution metadata of its elements (`metadata`) that itself is also a `p_object` and the base containers (`base_container`) that are non-`p_object`s that store the actual data. The container's metadata has information about the distribution of the elements, or mapping of element index to location ID, locating transparently local and remote elements.

```
1  template<typename T>
2  class array : public stapl::p_object {
3    metadata       m_meta;
4    base_container m_bcontainer;
5
6    void set_element(std::size_t n, T const& t) {
7      auto lid = m_meta.location_of(n); // find in which location the element exists
8      if (lid==this->get_location_id())
9        m_bcontainer[m_meta.find_index(n)] = t; // element on this location
10     else
11       stapl::async_rmi(lid, this->get_rmi_handle(), &array::set_element, n, t);
12   }
13
14   stapl::future<T> get_element(std::size_t n) const {
15     auto lid = m_meta.location_of(n); // find in which location the element exists
16     if (lid==this->get_location_id())
17       return stapl::make_ready_future<T>(m_bcontainer[m_meta.find_index(n)]);
18     else
19       return stapl::opaque_rmi(lid, this->get_rmi_handle(), &array::get_element, n);
20   }
21
22   std::size_t local_size() const { return m_meta.size(); }
23
24   stapl::future<std::size_t> size() const {
25     return stapl::allreduce_rmi(std::plus<std::size_t>{}, this->get_rmi_handle(),
26                                 &array::local_size);
27   }
28 };
29
30 stapl_main(...) {
31   array<int> a(100);
32   if (stapl::get_location_id()==0) {
33     a.set_element(99, 1);
34     auto f = a.get_element(99);
35     assert(f.get()==1);
36   }
37 }
```

Figure 5.2: Simple container

A simplified container is shown in Figure 5.2. To provide a shared memory view to the user, containers are declared as `p_objects`. RMIs are used to read and write elements. They are also used to access and mutate metadata, for example getting the total size of the container relies on the reduction primitives of ARMI (`reduce_rmi` and `allreduce_rmi`).

Figure 5.3 shows graphically the interaction between the containers and the STAPL-RTS. The light gray color is user code, whereas purple is container code and red is STAPL-RTS calls. Through the addition of minimal primitives, the STAPL-

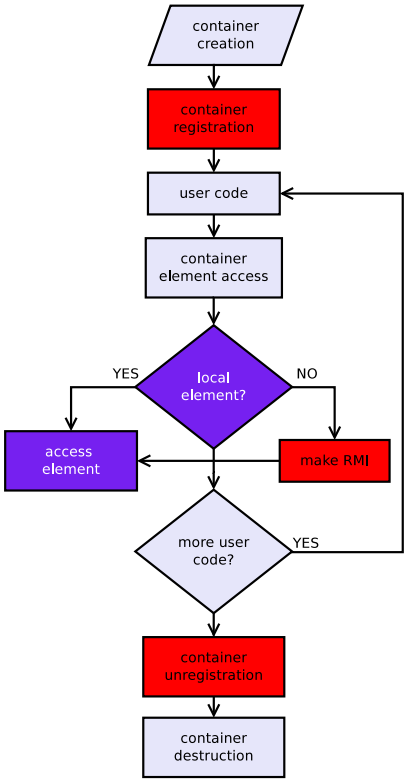


Figure 5.3: Container and STAPL-RTS interaction

RTS transforms data structures to distributed containers, attesting to its power as a solution for creating parallel frameworks.

Container composition is supported through creating the inner containers in their own isolated gang, different from that of the parent container. The support for container composition will be explored in Section 7.

STAPL views [10] are also inherently distributed objects. Several of their internal parts, such as the underlying container and domain are `p_objects`, thus views are `p_objects` as well. In fact, since the domain is a form of metadata, the container is actually a view with associated, per-location storage. The interaction between views and the STAPL-RTS is similar as that of the containers.

5.2 PARAGRAPHs

Users write their application as a composition of algorithms, expressed through skeletons. These composed algorithms are used in conjuction with the views to the data they work on to create a distributed task graph. The `PARAGRAPH` is the data flow engine of STAPL. It is a distributed task graph that is responsible for placing tasks, resolving dependencies and flow values between producer and consumer tasks, generated by the skeletons and the views.

A `PARAGRAPH` is essentially a distributed container of tasks. It is a `p_object` and has an associated set of locations on which its tasks are mapped for execution. `PARAGRAPH`s use RMIs to place tasks, resolve dependencies and flow values between producer and consumer tasks that are not on the same location. Additionally, runnable tasks are scheduled through the EXECUTOR framework.

The STAPL-RTS and the `PARAGRAPH`, while distinct and with clear interface separation, have a close relationship. The STAPL-RTS lacks the task dependence resolution capabilities of the `PARAGRAPH` and the `PARAGRAPH` requires an abstraction layer for communication and task scheduling. They complement each other, making the `PARAGRAPH` a *higher level* runtime system.

In Section 6 we will show the how the `PARAGRAPH` interacts with the STAPL-RTS to take advantage of shared memory in an abstract way, while in Section 7 we will present how they work together to provide generic nested parallelism for regular and irregular applications.

6. LEVERAGING SHARED MEMORY*

The current state of the art in HPC is a distributed memory machine comprised of nodes with accelerators and multiple processor sockets, each with a multi-core chip. Application development for these platforms is usually evolutionary: a scalable, distributed programming model (usually MPI [33]) is used for the initial implementation, with the memory hierarchy largely ignored. To increase performance, the implementation is extended with another library (e.g., OpenMP [35]), with threading for finer grain parallelism and shared memory with explicit synchronization to replace communication between processing elements. Writing such programs decorated with primitives from multiple low level APIs is an inherently non-scalable way to write software. Without a *separation of concerns*, only small programs written by expert developers actually achieve greater efficiency. The implementations are also rigid, difficult to extend, and not portable.

This lack of abstraction clearly detracts from code reuse and program composability. However, developers are often faced with no other choice if they wish to gain even some fraction of the peak performance modern systems offer. Efficiently mapping applications to such architectures requires semantic information that is usually lost when higher level programming models are used. In this chapter, we describe how user-level information is transferred to the STAPL-RTS to leverage shared memory when offered by the platform.

One of the key design goals of STAPL is portable performance: users must be able to write one version of the code that has good performance on different systems

*Part of this chapter is reprinted with permission from "STAPL-RTS: An Application Driven Runtime System" by Ioannis Papadopoulos, Nathan Thomas, Adam Fidel, Nancy M. Amato, Lawrence Rauchwerger, 2015. *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15*, 425–434, Copyright 2015 by ACM.

with minimal per-platform effort. The layered component architecture of the library supports this objective, with each component responsible for abstracting some area of concern in parallel programming, such as data distribution, computation specification, work scheduling, and communication. Key to obtaining performance is a *transfer of contextual information* between these components, while still maintaining the proper abstractions necessary for software reuse.

The STAPL-RTS presents a *unified interface* for both intra-node and inter-node communication to support performance portability. Internally the *mixed-mode* implementation uses both standard shared and distributed memory communication protocols when appropriate. Our *distributed Remote Method Invocation* (RMI) model guarantees scalability and correctness.

Each processing element together with a logical address space forms a *location* (isolated computational unit). Hence, parameters to RMIs are passed by value, maintaining strict copy semantics with no user-visible sharing. This approach provides safety to the user by guarding against data races. However, as with other features of higher level languages, it can introduce runtime overhead, in this case from excessive copying of large data structures. We show in this chapter how *copy elision* (i.e. removing unnecessary copying of objects) can eliminate this performance penalty, via simple annotations inserted by STAPL based on information from higher levels of the software stack.

This chapter makes the following contributions:

- **Transfer of application semantics to the runtime.** We employ annotations based on common programming idioms. As the STAPL-RTS is implemented in C++11 [18], the annotations are similar to C++ STL interfaces.
- **Copy removal via move semantics and immutable sharing**. To demonstrate

application driven optimization, we transparently avoid copies usually incurred when maintaining isolation of computational activities. We employ the commonly known idioms of move semantics [18] and immutable sharing [19], leveraging shared memory for communication between activities whenever possible. STAPL programs are expressed as task dependence graphs, with consumer tasks receiving read-only access to produced values. We use this graph representation to transparently insert annotations whenever possible: tasks with single consumers can direct the runtime to *move* their copy directly to the producer, which it will do if the tasks exist in shared memory. Tasks with multiple consumers can request *immutable references* to the value be transmitted to other locations where these successors exist.

6.1 Shared Memory Optimization Opportunities

We describe several aspects of the execution model of STAPL, motivating design decisions and pinpointing opportunities for application driven optimization.

6.1.1 Execution Environment

A STAPL application is always implicitly parallel and executes on a number of locations. Each location has an isolated, virtual address space which is not directly accessible by other locations. When a location wishes to modify or read a remote location's memory, the work must be expressed via RMIs on distributed `p_objects`, *even if the two locations reside in shared memory.* This design has the following ramifications:

- **Data Races Cannot Occur.** With only one processing element able to directly access memory and RMI atomicity guaranteed by the runtime system, users do not have the ability to create data races as is usually possible in shared memory parallel execution models.

- **Isolation causes copying.** One cost of the added safety is object copying between locations, even if they share a common address space. Maintaining isolation means values returned from RMIs between locations must be copied. We discuss in Section 6.2.1 how to minimize this overhead when in shared memory.

6.1.2 RMI Argument Copy Semantics

We enforce pass-by-value semantics for all arguments passed to an RMI. A private copy of any argument passed to a remote function call is presented to the receiver; any mutation on the argument either at the sender or at the receiver will not be visible to the other. Again, if this is done without high level information, the runtime may introduce unnecessary copies to enforce pass-by-value semantics.

6.2 Application Driven Optimization

We now give examples of how high level information is transferred from STAPL programs into the STAPL-RTS to guide optimization using `PARAGRAPH` directed copy elision between locations in shared memory. The information is provided at an appropriate level of abstraction (i.e., they need not be aware of how and if STAPL-RTS uses this information) through well known programming idioms, derived from standard C++ language features or library interfaces.

6.2.1 Argument Copy Elision in Shared Memory

Copy semantics simplify the reasoning about parallel programs, as they remove potential side-effects. However they can introduce significant runtime overhead. We relax our implementation of copy semantics with assistance from the `PARAGRAPH`. We describe three RMI annotations that allow STAPL-RTS to remove copies. They include transfer via `move`, return storage specification via `promise` and `future` objects, and the use of shared, immutable data references. We first describe how application

contextual information flows to the PARAGRAPH to guide copy elision and then discuss the annotations as well as their implementation.

6.2.1.1 PARAGRAPH Direction of Copy Elision

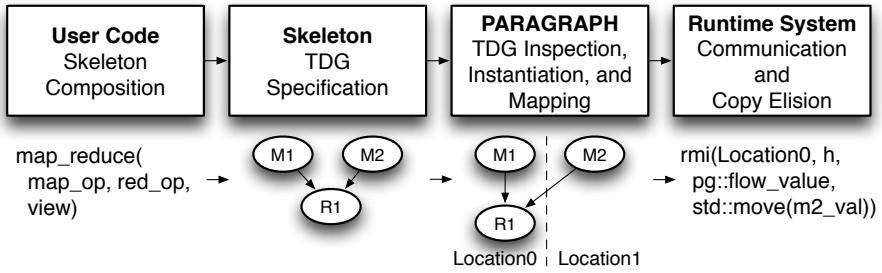


Figure 6.1: Copy elision in STAPL

Copy elision annotations are not inserted by STAPL application programmers. Instead the elision is directed by the PARAGRAPH, and Figure 6.1 depicts this process. First an application writer employs an algorithmic skeleton, which the skeleton framework uses to generate a task dependence graph specification. At run-time, this graph is instantiated by the PARAGRAPH and mapped onto a set of locations for execution. The PARAGRAPH also performs an inspection of the graph to detect where copy elision can be used. In Figure 6.1 the result of the map operation on location 1 can be transferred (i.e., moved) to the reduction task on location 0, as it is the only consumer of the value. The PARAGRAPH uses the following set of rules to identify elision opportunities:

- `move` **annotation**. If a task has single consumer and it is on a remote location (i.e., different than where the task executes), pass the value to `async_rmi` via `std::move`.

```
1 void produce() {
2   std::vector<int> v(N);
3   ... // populate v
4   async_rmi(dest, h, obj::consume, v);
5 }
```

(a) RMI invocation with copy of `v`.

```
1 void produce(...) {
2   std::vector<int> v(N);
3   ... // populate v
4   async_rmi(dest, h, obj::consume, std::move(v));
5 }
```

(b) RMI invocation with move of `v`

Figure 6.2: Move semantics for RMI arguments

- `immutable_shared` **annotation**. If a task has multiple consumers and at least one is on a remote location, use an immutable shared reference. The reference is passed to associated RMIs and also used to service local consumers.

- **No annotation**. If all consumers are on the producer's location, the value is managed locally with no RMIs.

6.2.1.2 Using Moves for RMI Parameter Passing

Consider the code in Figure 6.2(a) which calls `async_rmi`. A location executes function `produce` that creates a vector and sends it to another location via RMI. In this case, the copy of the vector parameter into the runtime is unnecessary. The source location produces the value solely for consumption at the destination location. This is an *object transfer* pattern present in many parallel algorithms (e.g., reductions). This type of value transfer is also desirable in sequential computing. C++11 [18] addresses this problem with language support for *rvalue references* and an associated library function `std::move`.

The STAPL-RTS supports the direct use of these move semantics with RMI parameter passing, so that the unnecessary copies can be completely avoided. The

trivially modified code in Figure 6.2(b) has been annotated to express the transfer of `v`. The parameter is passed without any copying *when the source and destination reside in the same address space* and is presented to `consume_value` as an rvalue reference, that is as an `std::vector<int>&&`. During execution, the parameter is moved from user space into the runtime, serialization is avoided, and control bits are inserted into the RMI request to forward the rvalue reference to the callee.

6.2.1.3 Immutable Object Sharing

There are times that basic *data transfer* between locations is insufficient. If there is still a local consumer of the value to be transmitted remotely, we can employ *immutable data sharing* to avoid overhead while still preserving copy semantics. This admittedly does not cover all cases (i.e., if the receiver wants to mutate the value, they must still copy it), but when it can be used, it gives similar savings as the zero-copy data transfers discussed in the previous two sections. We currently offer two variations of immutable sharing:

Permanently immutable objects. Values placed in an immutable wrapper via `make_shared_immutable` are guarded against mutation for the remainder of their lifetime. This primitive mimics the behavior of a `std::shared_ptr<const T>` and is used to safely share values between locations in shared memory. When the destination location resides in another address space, a new copy is initialized there to back the immutable wrapper. In each address space, the underlying copy is deleted when the last reference is deleted, using standard reference counting.

Figure 6.3(a) depicts an example of permanently immutable sharing. Assume that location 1 resides in the same address space as the location executing `A::put`, while location 2 does not. Location 1 shares a copy of `t` with the lifetime managed by STAPL-RTS. When `A::put` exits and references on location 1 are destroyed, the

```
foo(...) {
  auto t = stapl::make_immutable_shared<T>(...);
  stapl::async_rmi(1, h, &A::put, t);
  stapl::async_rmi(2, h, &A::put, t);
  T const& ref = t.get();
}
```

(a) Example `immutable_shared` usage

```
t = ...;
stapl::async_rmi(1, h, &A::put, immutable(t));
stapl::rmi_fence();
t = ...;
```

(b) Example `immutable` usage

Figure 6.3: Immutable object sharing in STAPL-RTS

copy is destroyed. Location 2 receives a wrapper to its own copy which is read and subsequently shared, if desired, with other locations.

Temporarily immutable objects. Objects can be tagged using the `immutable` function. A reference to such an object can be given to a destination location in shared memory, instead of copying it. This annotation, demonstrated in Figure 6.3(b), allows the caller to regain mutability rights of the object after the next synchronization point. Using the `immutable` tag, the caller guarantees that `t` will not be updated until after the `rmi_fence` collective synchronization call. Afterwards the variable can be safely modified.

6.2.2 Return Value Copy Elision

All return values from RMIs are returned only by copy in order to discourage the user from returning pointers or references to objects. The exception to this rule is returning a reference or pointer to a `p_object` from an intragang RMI. The reasons behind this design choice are

- to enforce data locality,
- to maintain portability by forcing the user to think in terms of distributed

memory and

- to avoid possible memory leaks, as returning a pointer or a reference requires a heap allocated object which the user has to delete.

Returning objects that themselves manage pointers, e.g., smart pointers such as `std::unique_ptr`, `std::shared_ptr` and others, is allowed.

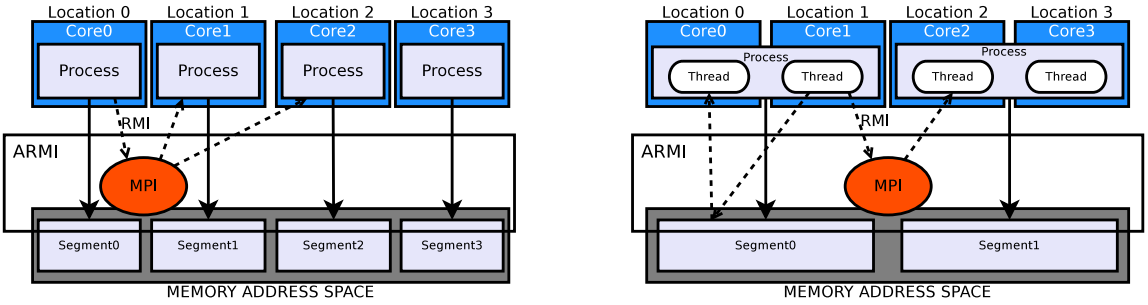
6.3 Mixed-mode Communication

To support the aforementioned optimizations, the STAPL-RTS is specialized when in shared memory. Our *mixed-mode* communication unifies shared and distributed memory communication under the same set of primitives.

Mixed-mode presents a consistent interface rather than mixing the two different paradigms. Locations communicate using RMIs on `p_object`s regardless of their relative position in the memory hierarchy. The STAPL-RTS internally changes its implementation and specializes RMI handling for each of these approaches. Maintaining our always distributed model is achieved by allowing the creation of locations on threads and avoiding the communication layer when communicating between them.

In mixed-mode, the program is executed on locations where some of them are on the same physical address space, or same node, and others are on different nodes. The STAPL-RTS can create more than one locations on each node that are capable of communicating between them via shared memory, whereas locations on different nodes communicate through a communication library. As far as the user is concerned, locations are still isolated and the only valid communication means is via RMIs on `p_object`s and thread safety is ensured by the STAPL-RTS.

Figure 6.4 shows the difference between executing an application on 4 MPI processes vs 2 MPI processes with 2 threads each on a 2 node, 2 core/node machine. While the machine configuration did not change, changing the *execution configura-*



(a) Distributed memory only (b) Mixed-mode

Figure 6.4: Mixed-mode execution vs distributed memory only

tion allows the STAPL-RTS to utilize shared memory optimizations.

6.3.1 Threading Backends

Through the concurrency component (Section 3.2.3), the STAPL-RTS abstracts the threading capabilities of the platform. Each different threading library is plugged in to the STAPL-RTS as a different *multithreading back-ends*. Currently, we support backends based on C++11 threads [34] and OpenMP [35]. Additional ones can be supported with the only restriction that only one can be active at any given moment.

6.3.2 Communication Protocol

When an RMI request is issued, the STAPL-RTS needs to find where the target location exists in the system. This location-to-PE mapping is available internally in the STAPL-RTS, but is partly distributed. To promote scalability, we aggregate the information about where a location is at a process level. The STAPL-RTS knows globally the process that a location is on, but not at which PE. The exact location-to-PE mapping is only available on a per process level; each process knows only of the exact mapping for the locations it hosts.

Figure 6.5 shows a high level overview of the location resolution when issuing an RMI. The metadata answers on which process a location is. Locations in shared

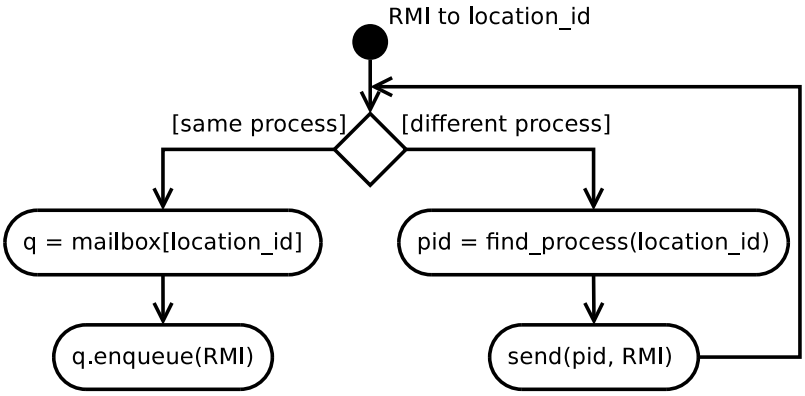


Figure 6.5: Enqueuing RMIs

memory communicate between them using a shared *mailbox*. The mailbox is a hash table that associates location IDs to lock-free buffer queues. Locations in shared memory have direct access to the mailbox and they are allowed to push a buffer containing RMIs directly to the destination location, using its ID as the key.

If the source and target locations are not on the same process, communication is delegated to the communicator. The request gets forwarded to the process that the metadata has said that knows where the location is. Upon arrival of the forwarded request, the metadata is queried again to determine the right queue. As mentioned, the metadata has the full location-to-process mapping on each process.

This design can easily be adapted to have an hierarchical resolution mechanism that answers where a location is, for example per node or per rack, reducing metadata duplication to the cost of more communication hops. In this case, the protocol will keep forwarding the request until the location is reached. Providing such support does not require placing restrictions to the number of forwardings allowed, since locations are pinned to their respective core and thus are never migrated, requiring a deterministic number of hops to be reached.

6.3.3 Shared Objects

Shared memory also enables the STAPL-RTS to use shared memory optimized implementations for common operations, e.g., reductions. Using these primitives relies on the ability to share objects between locations in shared memory, for example a thread-safe accumulator.

Previous approaches, such as OpenMP [35] require shared objects to be declared prior to creating the parallel section it is shared in. Others, such as Habanero-Java [43] require a reference to the object to be passed explicitly to each thread. For systems that support dynamic applications, these approaches are either cumbersome to use or impose unnecessary overhead.

The STAPL-RTS has an internal mechanism for creating shared objects dynamically, without requiring a priori declaration. This mechanism relies on the fact that the `rmi_handle` of a `p_object` is a unique value in a gang. Upon first request of a shared object, it is constructed and placed in a hashtable indexed with an `rmi_handle`, along with a reference count of how many locations will be sharing it. The `p_object` associated with it is inconsequential, as the `rmi_handle` is the key for that shared object. All locations that want to share the same object have to provide the same `rmi_handle`. The only limitation of this mechanism is that only one shared object is allowed per `p_object`.

6.3.4 Accessing Communication Layer

6.3.4.1 Outbound Communication Access

Each location that has to communicate with another location over distributed memory must go through the communicator module. Our current communication layer is based on MPI and we ensure that only one thread on the shared memory is allowed to make calls to the MPI layer at any given moment. This way, we only

require `MPI_THREAD_SERIALIZED` instead of the slower `MPI_THREAD_MULTIPLE` [75].

In order to avoid contention on that layer as each thread attempts to acquire a lock around the communication layer, we introduce a deferred task mechanism. A location that wants to use the communicator attempts to lock it. If the lock fails, it places a task in a lock-free queue that has all the information regarding the requested communication, such as the buffer, the destination and other metadata and returns. If the lock succeeds, then the location will send everything that is currently in the lock-free queue and then send its own buffer.

6.3.4.2 Inbound Communication Access

Since most low level communication libraries are polling-based, rather than interrupt-based, they have to be occasionally checked for incoming communication. In the STAPL-RTS, while a location waits for a value, it is free to process its pending RMIs. If it has no RMIs, then it can poll the distributed memory communicator for any incoming messages. We implement this functionality using busy-waiting. However, if multiple locations on the same shared memory node are idle or are blocked waiting for a value, then it is easy to have contention at the lock in the communicator.

Resolving contention can be done either by electing specific threads that are allowed to access a shared resource or by allowing all threads to attempt to access the shared resource and backing off if another thread is accessing it. In the STAPL-RTS we offer a three distinct policies:

- **Master thread policy** which allows only the locations that are mapped to the master thread of the process to access the communicator. While this approach has the lowest overhead, it leads to unfair work distribution as locations on the master thread have to always bear the cost of receiving and pushing requests to the correct queue, even if other locations are idle.

- **First thread policy** where the first thread that manages to lock the communicator takes care of inbound communication. In order to reduce contention, as multiple threads try to be the first, we use an hierarchical lock that uses the PE hierarchy from the concurrency component along with an exponential back-off mechanism.

- **Dedicated thread policy** in which a dedicated, separate thread is the only one allowed to access the communication layer.

6.4 Related Work

In [80] the authors propose Kanor, a declarative language for writing parallel programs in partitioned address spaces. Users annotate how data flows between address spaces to describe communication. The Kanor source-to-source compiler uses these annotations to do a more informed data-flow analysis and appropriately promote objects to global shared objects in shared memory, achieving zero-copy without explicit synchronization from the user's side, while maintaining the isolation features of distributed memory. The resulting code targets either multithreaded code or MPI code, but not a mix of both. The immutable object support we present is similar to performing the globalization optimization on a variable in Kanor but with the added benefit that it works in mixed-mode as well. A compiler such as Kanor's could easily leverage the immutable object support we offer.

A framework for taking advantage of immutable objects is introduced in [81] for code optimization. The authors describe a set of immutability annotations for Java that can be added to local or member variables. These can be used by the compiler to perform optimizations such as relaxing bounds checking, and load eliminations.

Several papers explore reference and object immutability for type safety reasons, with the potential to enable optimizations using those guarantees. Javari [82] and

IGJ [83] extend the Java language with reference and object immutability qualifiers. Using them, they provide *symbolic constants*. While it is mentioned that these qualifiers can enable code optimizations, this opportunity is not explored. The authors of [19] extend the concepts of object and reference immutability to build a type system that offers immutability guarantees for objects, for the purpose of exploiting it in parallel execution. However, the paper does not expand on the performance implications. The presented immutable object support is similar to this work, but our focus is exploiting shared memory, rather than enforcing type safety.

There have been previous attempts to provide a unified communication model. Treadmarks [84] and Intel Cluster OpenMP attempted to expand shared memory models to distributed memory. While novel and popular at the time, such shared memory approaches suffer inherent scalability issues which make them infeasible for large distributed systems. Additionally, it has been suggested that MPI should become aware of shared memory through the use of the RMA functionality [85], but this has yet to be approved in the MPI specification. However, the use of all these primitives must be explicitly set up and managed by the user, making it effectively a multi-protocol approach.

MPI implementations [86, 87] detect intra-node communication and use optimized methods for copying data. While the optimizations take advantage of the node memory hierarchy, data copying is still required between MPI processes.

Hybrid OpenMP+MPI solutions have been used in applications [88, 89] with success. Almost all of these applications have sequences of parallel OpenMP sections followed by sequential sections that perform communication using MPI. The reason for this configuration is that while MPI implementations allow threads to communicate with each other under the `MPI_THREAD_MULTIPLE` mode [90], it has been shown that this negatively affects performance [75].

Habanero-C with MPI (HCMPI) [91] introduces distributed memory communication in Habanero-C. It uses the Habanero task programming model for intra-node computation and synchronization, while introducing new functions based on MPI for the inter-node equivalents. While HCMPI has better performance than MPI or hybrid MPI+OpenMP, it presents two different programming models to the user. Intra-node, HCMPI uses the Habanero-C interface, while inter-node it relies on a MPI-like message passing interface.

HPX [92] supports hybrid-mode, allowing threads to communicate in distributed memory with asynchronous primitives. Future and promise mechanisms are provided for synchronization and data retrieval. The distributed memory communication employs MPI. In shared memory, it allows arguments passed by reference, significantly reducing communication latency. In distributed memory the arguments are copied, exposing a slightly different view to the user depending on the target of the communication primitive.

Charm++ [51] provides support for hybrid mode, either by having multiple threads per node to process messages or by declaring functions threaded and allowing them to block while waiting for a value to arrive. The user has to be aware of the threading capabilities, as data structures that may be accessed by multiple threads have to be properly protected.

Chapel [52] has been designed to specifically fit multicore distributed systems. As such, they offer support for creating tasks asynchronously either on distributed or shared memory. There is not enough information on how Chapel performs in mixed shared and distributed memory applications. However, shared-memory only performance indicates that Chapel may not adequately optimized [93].

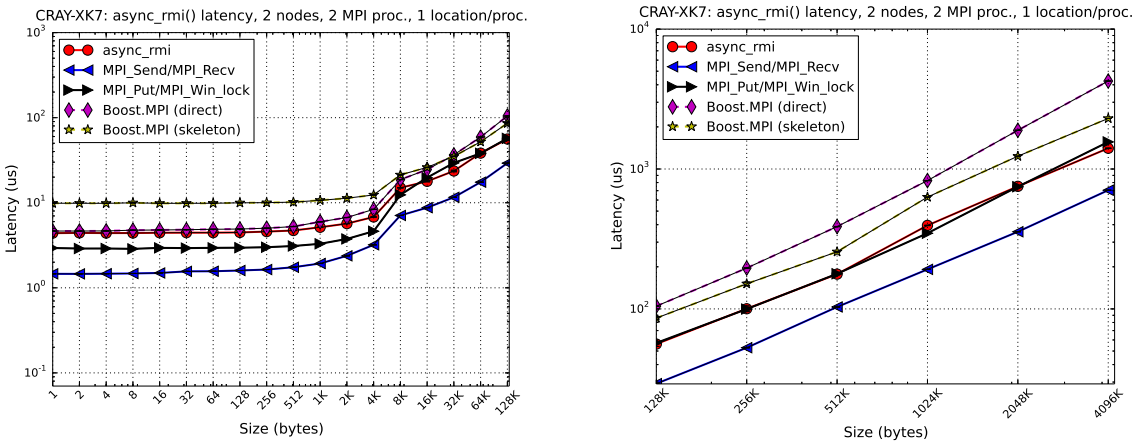
Tpetra [50] is a linear algebra package from Trilinos. Tpetra supports hybrid-mode parallelism through Kokkos; communication in distributed memory uses MPI,

while shared memory communication and computation employ various Kokkos back-ends (Pthreads, OpenMP, Intel TBB, CUDA). Tpetra resembles a hybrid MPI and OpenMP system, as Kokkos tasks cannot communicate through distributed memory.

6.5 Experimental Evaluation

For all our experiments, we configured the STAPL-RTS with the OpenMP-based concurrency back-end with each location mapped to one OpenMP thread. In all cases, threads are pinned to one core. The default ARMI buffer size is 8 KB which means that the one-way handshake protocol (see Appendix B.2) will be used in distributed memory communication for payloads that are bigger than 8136 bytes, as 56 bytes are used for ARMI bookkeeping and metadata. This means that the STAPL-RTS sends two MPI messages rather than one, increasing the latency but allows truly asynchronous communication without payload size restrictions.

6.5.1 Point-to-Point Latency



(a) Small objects (log y-axis)

(b) Large objects (log y-axis)

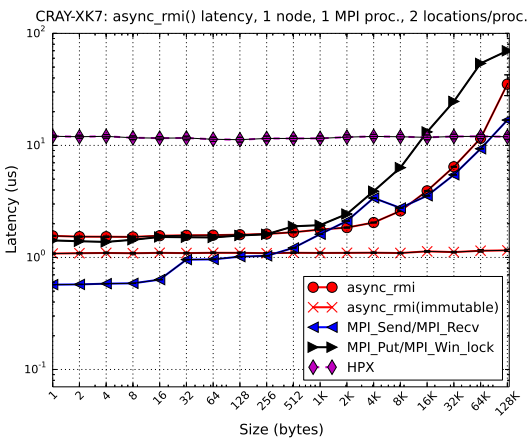
Figure 6.6: `async_rmi` latency against MPI and Boost.MPI

Figure 6.6 compares `async_rmi` against `MPI_Send`/`MPI_Recv`, one-sided `MPI_Put`/ `MPI_Win_lock`[†], and Boost.MPI [66] using the OMB [74] latency benchmarks. For the Boost.MPI benchmark, we implemented two versions, one that serializes and sends the data structure containing the payload directly (`Boost.MPI (direct)`) and one that creates and sends the MPI datatype of the data structure, followed its data (`Boost.MPI (skeleton)`). For the latter, we are creating and transmitting the datatype for each object sent to better simulate what would happen in an actual application with varying data structures. All benchmarks where performed with `MPI_THREAD_SERIALIZED`.

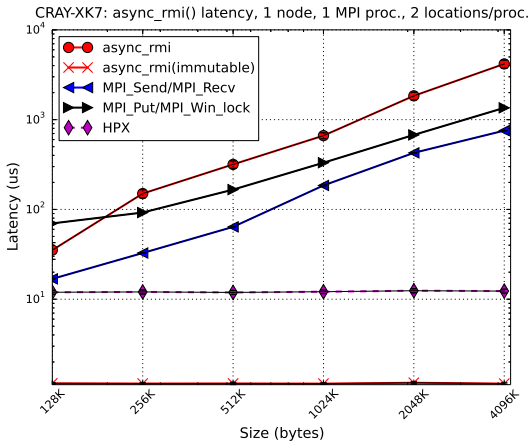
The pair of `MPI_Send`/`MPI_Recv` experiences the least latency, which is attributed to the fact that they are blocking and they have direct access to the buffer that contains the data to be transmitted, therefore requiring only minimal bookkeeping information. Boost.MPI performance suffers since it has to serialize and deserialize the data in an internal buffer prior to sending it and after receiving it, introducing an extra copy at both the sender and the receiver. Similarly, the `async_rmi` has to copy the data to an internal buffer in a serialized form, not only to preserve the copy semantics of the RMI, but also to be able to transmit arbitrary objects. While we cannot match the latency of `MPI_Send`/`MPI_Recv`, as we are using their non-blocking counterparts to perform communication in ARMI, we offer similar latency to the one-sided MPI primitives and considerably better latency than Boost.MPI.

In Figure 6.7, we compare our latency on one node, with the STAPL-RTS and HPX benchmarks executing on shared memory (2 threads in the same process), against MPI on two processes. Our latency for `async_rmi` is on par with the one-sided MPI. It is clear though that we pay the overhead of serialization, as the pair `MPI_Send` / `MPI_Put` can do a `memcpy` from one address space to another, whereas we have

[†]This was the fastest combination on our system.

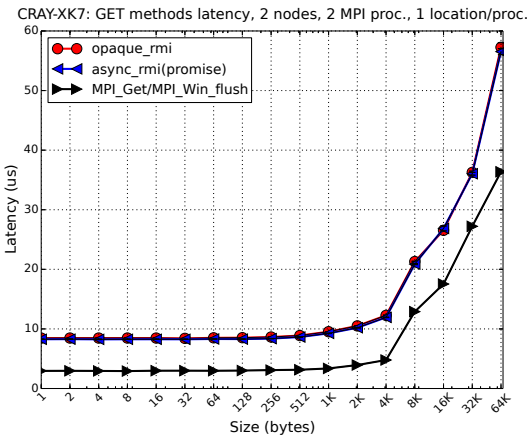


(a) Small objects (log y-axis)

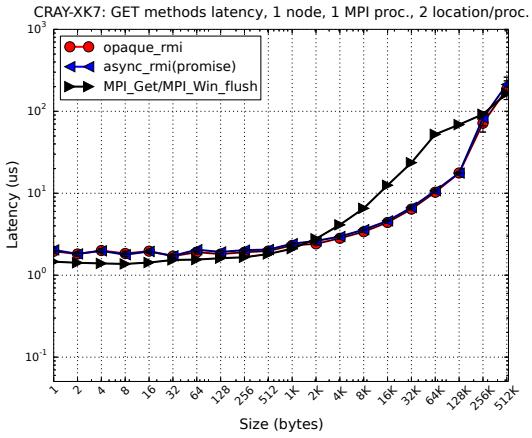


(b) Large objects (log y-axis)

Figure 6.7: `async_rmi` latency against MPI and HPX on one node



(a) Distributed memory



(b) Shared memory (log y-axis)

Figure 6.8: `opaque_rmi`/`async_rmi(promise)` latency against one-sided MPI

to serialize the object, that involves among others a `memcpy` for its data. HPX on the other hand sends a reference to the data, resulting in constant latency for any payload size.

6.5.2 Asynchronous Return Values

RMIs offer additional flexibility compared to MPI for value retrieval. MPI Remote Memory Access (RMA) [90] use is complex, requiring explicit memory registration and synchronizations. RMIs expose a simple, high level interface which allows one to either wait for values or pass them to a continuation via `future::async_then`, with data transfer details managed by the STAPL-RTS.

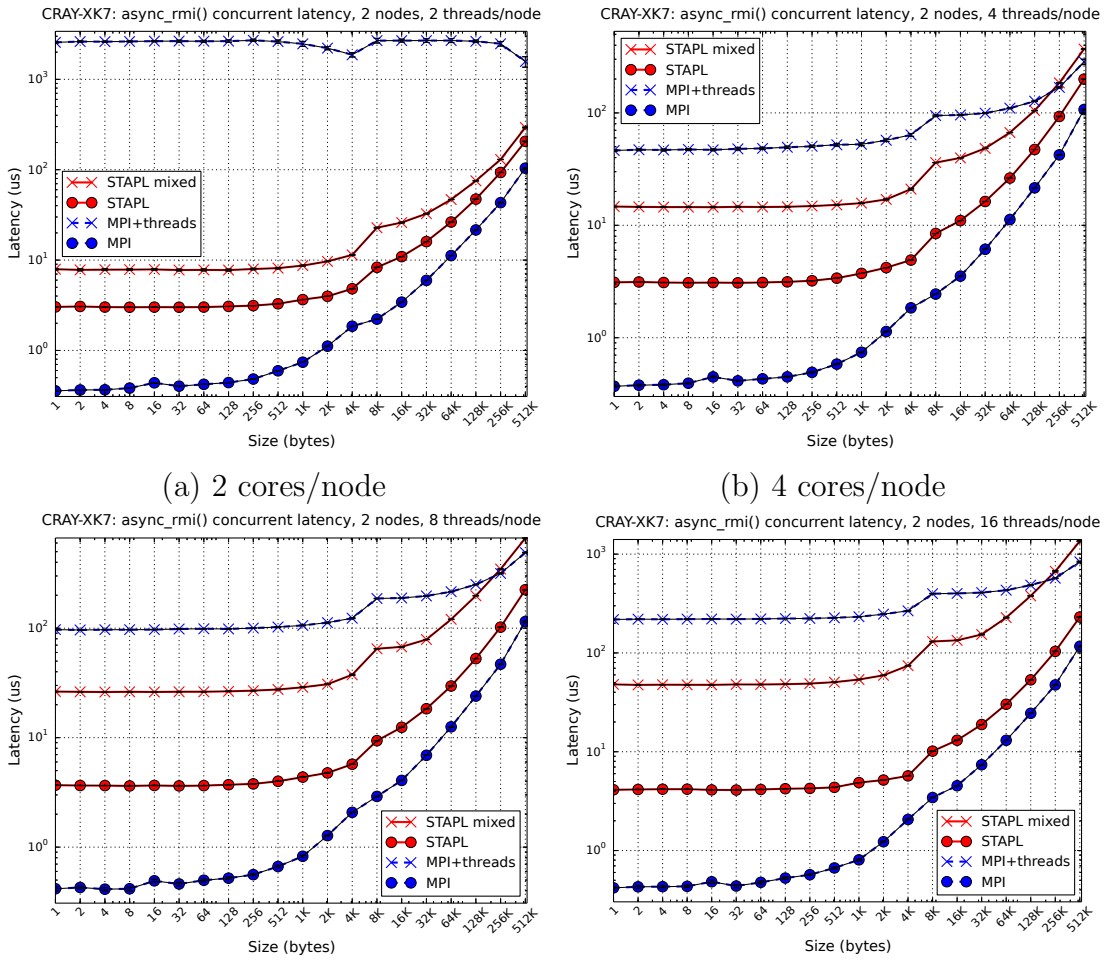
Figure 6.8 presents the latency of our primitives employing futures and promises (Section 3.4.1). We compare against one-sided MPI with `MPI_Get`/`MPI_Win_flush`[‡] under distributed and shared memory. Creating MPI windows is done through `MPI_Win_create_dynamic`, as it provides the most flexible form of RMA memory registration for non-trivial applications [94].

Both our methods (`opaque_rmi` and `async_rmi` with `promise`) have similar latency. They are competitive with MPI in distributed memory. However as the object size increases past 64 KB, serialization begins to noticeably affect perceived latency. In shared memory, especially for medium object sizes (2 KB - 256 KB), we outperform MPI, as we can automatically elide one memory copy that MPI has by performing in-place construction of the object in the receiver's address space. MPI starts to outperform us after 512 KB.

6.5.3 Concurrent Latency

In Figure 6.9, we evaluate our system under maximum contention using the concurrent latency benchmark from [75]. We configure our experiment to run on 2 processes with one process per node, and 1, 2, 4, 8 and 16 locations per process. It is worth noting that changing the number of locations per process requires to set the appropriate environment variable (`STAPL_NUM_THREADS`) at run-time and does not

[‡]This was the best performing combination on our system.



(a) 2 cores/node

(b) 4 cores/node

(c) 8 cores/node

(d) 16 cores/node

Figure 6.9: `async_rmi` concurrent latency against MPI and MPI+threads

require any reconfiguration.

Each location is communicating with a location in the other process, forcing all threads to go through the communicator. Similarly, in the MPI experiment, each process has multiple threads that communicate through the same MPI communicator using different tags. Therefore, the MPI layer is configured with `MPI_THREAD_MULTIPLE`. Communication patterns like this that can appear in applications with unpredictable communication (e.g. graph applications). In these applications, introducing a hy-

brid MPI+OpenMP solution that tries to extract communication in a separate phase might be problematic and the only solution may be a thread-safe MPI. Our system has very competitive latency compared to MPI, especially for smaller payload sizes where the serialization overhead is minimal. The effects of the one-way handshake protocol (see Appendix B.2) are visible at $> 4\,\text{KB}$ and our latency increases due to object serialization at $> 64\,\text{KB}$.

6.5.4 Graph 500

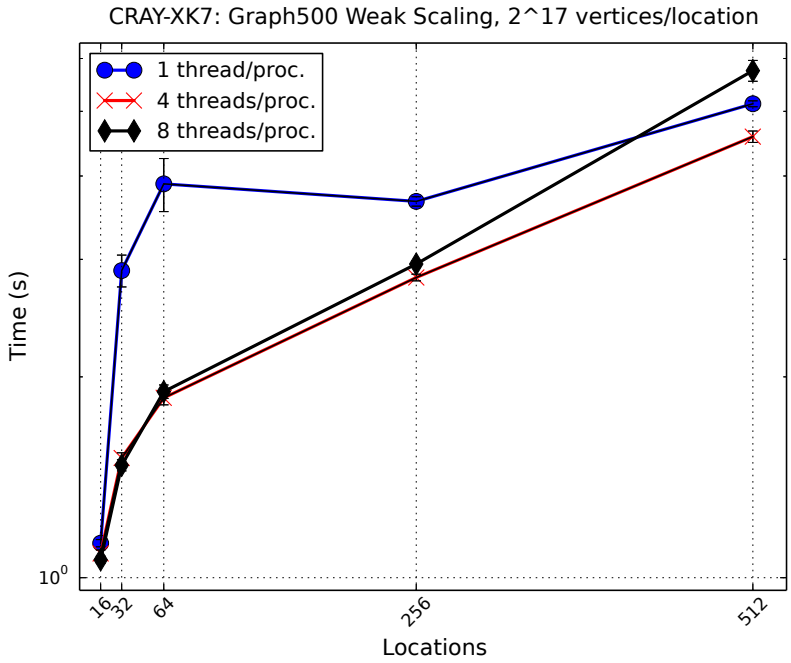


Figure 6.10: Graph 500 in distributed memory only and mixed-mode

In Section 6.5.3 we showed results when all locations have to go through the communication layer. For a more realistic communication model, we evaluate mixed-mode using the Graph 500 benchmark [77] as presented in [12]. In Figure 6.10

we compare the same implementation running with 1 thread/process (blue line), 4 threads/process (red line) and 8 threads/process (black line). For each horizontal graph grid line, the number of locations is the same, the only thing that changes is the configuration. So for 256 locations, the blue line means that the application is invoked on 256 MPI processes, the red line on 64 MPI processes with 4 threads each and the black line on 32 MPI process with 8 threads per process.

Without modifying the implementation at the higher levels and simply by changing the configuration, we are able to perform better than the distributed memory only version for the 4 thread case.

When using 8 threads, the contention at the communicator eliminates our gains. Graph 500 has a random communication pattern and the more locations exist, the less probable is for a location to communicate with another location in shared memory. This increases the number of locations trying to access the communicator. It is evident that mixed-mode benefits applications that favor neighbor communication patterns and those neighbors reside on the same shared memory node.

Currently, our Graph 500 implementation does not take advantage of zero-copy. It does not use move semantics or immutable sharing, minimizing the opportunities for eliding copies in shared memory. We expect that by introducing these zero-copy techniques and further reducing contention will allow us to exhibit better results.

6.5.5 Jacobi Solver

For more structured communication, we evaluate the STAPL-RTS against hybrid MPI+OpenMP using the Jacobi solver presented in Appendix A.1. We run the solver for 100 iterations on a 23040×23040 matrix for 8 and 16 threads per process for up to 512 processors (32 nodes) and present the results in Figure 6.11. The hybrid solution is able to directly copy data from one address space to another when in

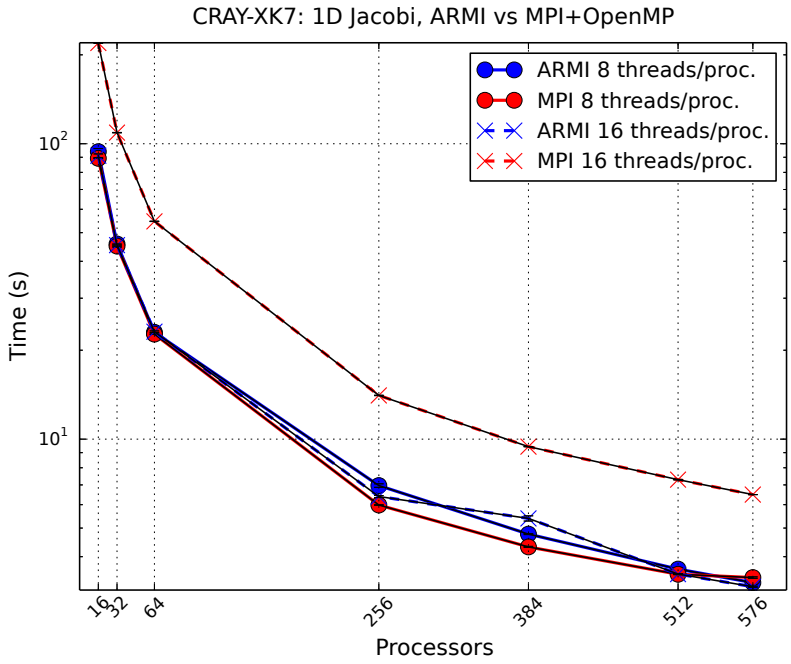


Figure 6.11: Jacobi solver in mixed-mode ARMI and hybrid MPI+OpenMP

distributed memory and access the data directly when in shared memory. The ARMI implementation always creates copies of a range of values using the `make_range_n` primitive (see Figure A.2).

In the smaller thread counts (8 threads), while we scale similarly to the hybrid solution, we experience higher overhead, as our always asynchronous, distributed nature forces us to always make data copies, either in shared or distributed memory, something that the hybrid version can avoid. However, in higher thread counts (16 threads), the fork-join model of the shared memory parallelism in the hybrid code takes its toll, increasing the overall processing time. This is where we expect our work to provide clear benefits: as the number of cores increases, the fork-join implementations have to spend more time creating parallel sections, whereas our model avoids this.

6.5.6 Copy Elision in K-means Clustering

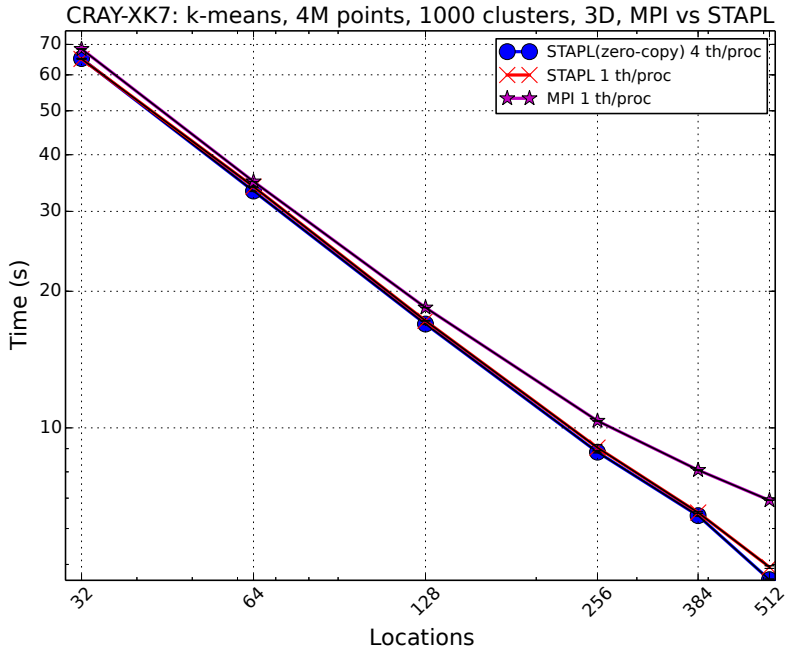


Figure 6.12: K-means algorithm with 4 M points, 1000 clusters in 3 D space on Cray-XK7

K-means clustering [95] is a widely used data mining algorithm. Given a set of vectors in an n-dimensional space, the algorithm assigns vectors which are similar to one another to a specific cluster. The "K" refers to the number of clusters, which is specified by the user, at the start of the algorithm. The "means" refers to the computation for associating the vectors. Each cluster is represented by a single point in the space, which is referred to as a *cluster means* or *cluster centroid*. Dhillon and Modha [96] present a sequential and analogous parallel implementation of k-means. The parallel algorithm is implemented using MPI.

We implemented the Dhillon and Modha MPI version of k-means in C++ and

then created a STAPL implementation of the same algorithm. It employs an algorithmic skeleton performing a `map` operation followed by an `all_reduce`. Binary reduction tasks use moves on one of the inputs to co-locate data for the operation. The broadcast portion of the allreduce operation uses shared immutability to avoid unnecessary copies during dissemination of new cluster centroids. These optimizations are enabled by the PARAGRAPH using the rules outlined in Section 6.2.1.1.

The scalability of the STAPL version (`STAPL 1th/proc`) as shown in Figure 6.12 surpasses that of the MPI implementation, due to other optimizations besides copy elision. Despite being primarily a computation kernel, the mixed-mode execution with copy elision (`STAPL(zero-copy) 4th/proc`) sees gains of up to 6.2% over the basic STAPL implementation.

7. NESTED PARALLELISM*

Writing parallel applications is difficult, and many programming idioms taken for granted in sequential computing are often unavailable. One of these tools, program composition via *nested function invocation*, is not present in many parallel programming models, at least not in a general form that is abstracted from the target architecture. Indeed, while *nested parallelism*, the "ability to take a parallel function and apply it over multiple instances in parallel" [30], is a natural way to express many applications, employing it is often constrained by the deep memory hierarchies and multiple communication models of modern HPC platforms.

Sequential support for nested algorithm invocation is straightforward: appropriate state (e.g., registers) is saved, the call stack is initialized according to convention, and control is transferred to the target function until it returns. However, parallel programming models present a more challenging scenario. Nested parallel algorithm invocations must be efficiently mapped onto the processing elements (PEs) with the locality of data it accesses considered. Furthermore, by definition, multiple nested invocations occur concurrently, meaning a coordination of activities is required.

While the efficient mapping of the application's hierarchy of algorithms onto the machine's hierarchy is important for performance, we believe requiring developers to explicitly coordinate this effort is overly burdensome. Furthermore, direct management leads to ad-hoc solutions that significantly decrease *software reuse*, which is key to addressing the difficulties of parallel programming.

Previous work tried to make the problem tractable by focusing on specific types

*Reprinted with permission from "Asynchronous Nested Parallelism for Dynamic Applications in Distributed Memory" by Ioannis Papadopoulos, Nathan Thomas, Adam Fidel, Dielli Hoxha, Nancy M. Amato, Lawrence Rauchwerger, 2015. *Lecture Notes in Computer Science*, 9519, 106–121, Copyright 2015 by Springer International Publishing Switzerland.

of applications. Some target more regular applications, where the mapping can be done in a static, globally coordinated manner. Invocation happens collectively, with processing elements together creating a new isolated section for the nested algorithm. Though restricted in the applications they support, these models tend to be both expressive and exhibit good performance.

Dynamic applications require more flexible support. Since the nature of the parallelism is not known *a priori*, nested algorithms are usually implemented as a series of dynamically spawned tasks. The mapping of these tasks must be either managed by the user or handled by the runtime system, making clean algorithm expressivity and performance difficult to achieve together.

This chapter describes the support for nested parallelism in the STAPL-RTS and how it extends to the creation of nested parallel sections that execute STAPL algorithms. These nested SPMD (Single Program Multiple Data) sections provide an isolated environment from which algorithms, represented as task dependence graphs, execute and can spawn further nested computation. Each of these sections can be instantiated on an arbitrary subgroup of PEs across distributed and shared memory.

While the STAPL-RTS supports collective creation of nested parallel sections, we will mostly focus on the *one-sided* interface. The one-sided interface allows a local activity (e.g., visiting a vertex in a distributed graph) on a given location to *spawn* a nested activity (e.g., following all edges in parallel to visit neighbors). Both the creation and execution of this nested activity are asynchronous: calls to the STAPL-RTS are non-blocking and allow local activities to proceed immediately. Hence, the one-sided, asynchronous mechanism is particularly suitable for dynamic applications.

Nested sections are also used to implement *composed data structures* with data distributed on arbitrary portions of the machine. Together, this support for nested algorithms and composed, distributed containers provides an increased level of sup-

port for irregular applications over previous work. In the experimental section, we demonstrate how the algorithms and data interact in a STAPL program, initially with finding the minimum element on composed containers created such that computation is imbalanced. We also use a distributed graph with vertex adjacency lists being stored in various distributed configurations. Without any changes to the graph algorithm, we are able to test a variety of configurations and gain substantial performance improvements (2.25x at 4K cores) over the common baseline configuration (i.e., sequential storage of edge lists).

Our contributions include:

- **Uniform nested parallelism with controlled isolation.** Support for arbitrary subgroups of processing elements (i.e., locations) across distributed memory. The sections are logically isolated, maintaining the hierarchical structure of algorithms defined by the user. For instance, RMI ordering and traffic quiescence is maintained separately for each nested section.

- **Asynchronous, one-sided creation of parallel sections.** The ability to asynchronously create nested parallel sections provides latency hiding which is important for scalability. We combine one-sided and asynchronous parallel section creation, presenting a simple and scalable nested parallel paradigm.

- **Separation of algorithm specification and mapping.** STAPL-RTS provides services to multiple components of upper layers (e.g., STAPL), which enable the specification of an algorithm to remain independent of the mapping. The latter is managed by STAPL, providing the appearance to users that nested function invocation proceed in a manner similar to that of sequential programming models. We build on STAPL's unified communication model [15] and we offer virtualized affinity and creation of parallel sections while maintaining information for

hierarchical-aware placement.

- **Use of STAPL-RTS to implement dynamic, nested algorithms.** We use our primitives to implement several fundamental graph algorithms, and demonstrate how various distribution strategies from previous work can be generalized under a common infrastructure using our approach to nested parallelism.

Results are presented for both static and dynamic benchmarks, demonstrating the flexibility of the approach and its performance at scale of up to 16K cores.

7.1 Design Considerations

In order to take advantage of nested parallelism and realize its full potential, we have made several design decisions that influence our implementation including:

7.1.1 Expressiveness

Users express algorithms as a composition of simpler parallel algorithms using algorithmic skeletons [16]. This specification is independent of any target architecture. The responsibility for mapping it onto the machine is left to the library, though it can be customized by more experienced users at an appropriate level of abstraction.

7.1.2 Preserving Algorithm Structure

We maintain the hierarchy of tasks defined by the application when mapping it to the machine. Hence, each nested section's tasks remain associated with it and are subject to its scheduling policy. Each algorithm invocation is run within an SPMD section, from which both point-to-point and collective operations are accounted for independently of other sections. The SPMD programming model has been chosen since scaling on distributed machines has favored this programming model (e.g., MPI [33]) more than fork-join or task parallel models.

7.1.3 Parallel Section Isolation

Parallel sections created from the STAPL-RTS exhibit controlled isolation for safety and correctness. The uncontrolled exchange of data between parallel sections is potentially unsafe due to data races. Performance can be impacted, as isolation means that collective operations and data exchanges are in a controlled environment. We discuss techniques to mitigate these overheads in [15]. Users have to explicitly the data available for access in each section.

7.1.4 Asynchronous, One-sided Parallel Section Creation

We support both *partitioning* (collective creation) of existing sections and *spawning* (one-sided creation) of new sections. Partitioning existing parallel sections is beneficial for static applications but is difficult to use in dynamic applications. On the other hand, one-sided creation may not give optimal performance for static applications where the structure of parallelism is more readily known.

In this work we will focus on the one-sided creation as it is a more flexible approach than the collective creation for dynamic applications. One-sided creation is also fully asynchronous, allowing us to effectively hide latency and better support our always distributed memory model.

7.2 Flow of Execution

As with STL programs, a typical STAPL application begins with the instantiation of the necessary data structures. Each container has its own distribution and thus defines the affinity of its elements. Container composition is supported, as well as complete control over the distribution of each container (e.g., balanced, block cyclic, arbitrary) irrespectively of where it exists in the composition hierarchy.

Users write applications with the help of skeletons [16] and views [10], that ab-

stract the computation and data access, respectively. The skeletons provide a platform independent specification. The views provide element locality information, projecting it from the underlying container. The responsibility of mapping the application onto the machine is left to the library, though it can be customized by more experienced users at an appropriate level of abstraction.

An algorithm's execution is performed by a `PARAGRAPH`, a distributed task dependence graph responsible for managing task dependencies and declaring which tasks are runnable. Each `PARAGRAPH` executes in an isolated parallel section, with data access provided by the views. Each task may itself be a parallel algorithm, for which a new nested parallel section is created. A default policy *places* a `PARAGRAPH` for execution based on the locality of the data it accesses, or one can define a custom policy at `PARAGRAPH` creation.

Parallel sections exhibit controlled isolation for safety and correctness. The uncontrolled exchange of data between parallel sections is potentially unsafe due to data races. Performance can be negatively impacted by isolation because isolation requires that collective operations and data exchanges are in a controlled environment. Techniques to mitigate these overheads are discussed in Section 6. Users provide views to define the data available for access in each section.

Figure 7.1 shows graphically the flow of execution. As in Section 5.1, light gray color is user code, whereas purple is `PARAGRAPH` code and red is STAPL-RTS calls. While the STAPL-RTS manages the creation of the isolated parallel sections, the decision of if a parallel section is created is left to the `PARAGRAPH`. While the STAPL-RTS allows arbitrary levels of nested parallel algorithm invocations, the `PARAGRAPH` is ultimately responsible for stopping the recursion (flattening).

The `PARAGRAPH` decides on both if a new parallel section should be created as well as its placement based on locality provided by the views. This can lead to multiple

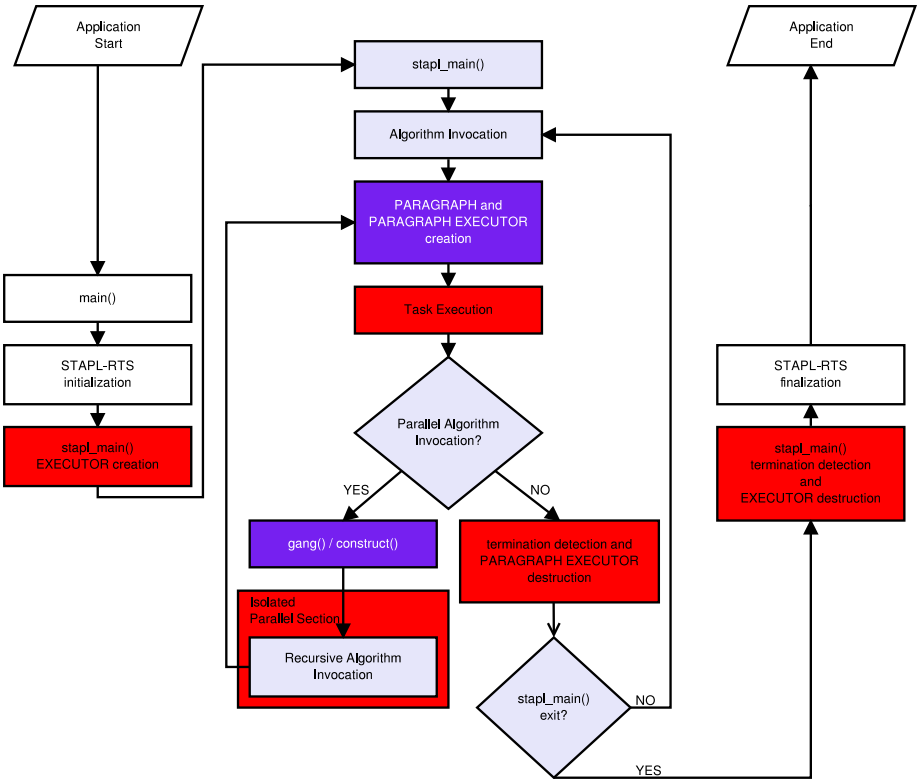


Figure 7.1: Flow of execution

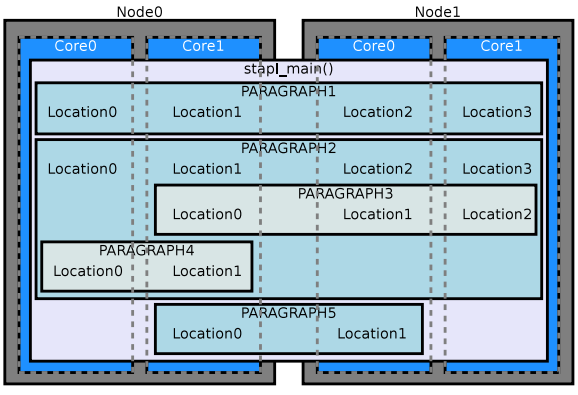


Figure 7.2: Execution model with nested parallel sections

parallel sections being mapped onto the same PEs. Figure 7.2 shows an example execution instance of an application that has a number of `PARAGRAPH` invocations in isolated parallel sections over the same set of hardware resources.

7.2.1 Container Composition and Nested Parallelism

Containers and `PARAGRAPH`s are both distributed objects (i.e., `p_object`s) and they use RMIs for communication. Containers use RMIs to read and write elements, whereas `PARAGRAPH`s use them to place tasks, resolve dependencies, and flow values between tasks that are not on the same location.

Distributed containers can contain other distributed containers. The inner containers can have their own type, distribution and they can be constructed in their own parallel section, different from that of their parent container. More importantly, in turn, they can be containers of containers, making container composition a first-class citizen, with the STAPL-RTS providing the necessary support for creating them and handling communication.

Nested parallelism is supported by allowing `PARAGRAPH` tasks to invoke nested parallel sections, creating new `PARAGRAPH`s in the process that are defined in their own parallel section. We can view the nested `PARAGRAPH` hierarchy and container composition as two different expressions of *object composition*. Thus it is natural that support for both is provided through the same mechanisms.

7.2.2 Parallel Sections

The STAPL-RTS allows the creation of new parallel sections that are independent of the section that created them via

- *spawning* a new parallel section, whereby one location creates a new section in an *asynchronous* and *one-sided* manner, using a subset of the resources of an existing section and

- *partitioning* the resources of an existing parallel section through *collective* primitives over the resources that participate in the new section.

7.3 Gangs

In Section 3.1 we introduced the concept of a *gang*. Gangs are the STAPL-RTS subgroup support. Each parallel section executes inside one. While the locations of a gang execute a single SPMD task, they communicate asynchronously independently of each other, making them a more loosely knit group than, for example, MPI groups or Titanium/UPC++ teams.

Subgrouping gives the ability to invoke algorithms on a subset of the resources without algorithm or code modifications. It allows partitioning the machine, so as to create associations of existing processing elements. This can lead to lower communication costs for collective operations as well as synchronization.

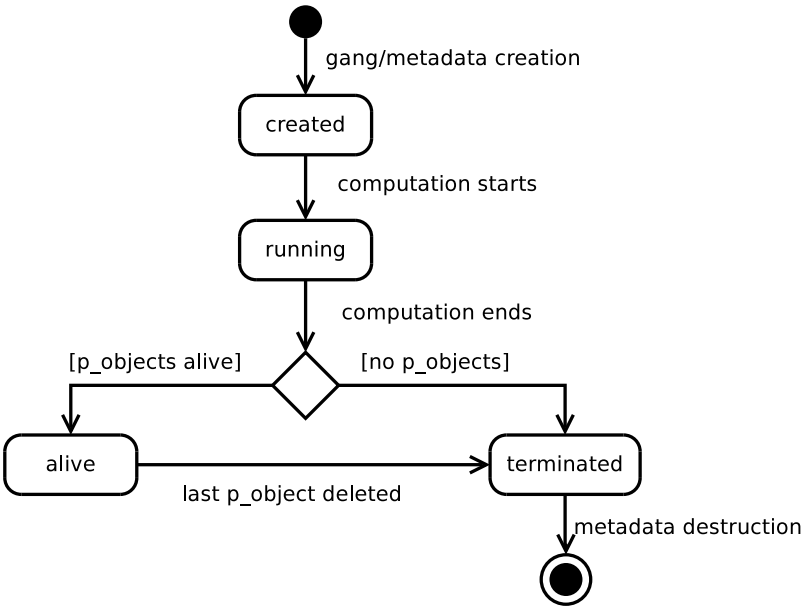


Figure 7.3: Gang state transitions

p_objects can be created within a gang, and as such, each p_object is associated with exactly one gang and is distributed across its locations. A gang can have any number of p_objects and its lifetime is tied to that of the p_objects present in it. Figure 7.3 presents a state transition diagram of the life of a gang.

- Upon construction, the gang is *created*. The gang metadata are generated and everything is set up to execute the SPMD task.

- When the task executes, the gang is declared *running*. While the task executes, p_objects can be created and they are automatically associated with the gang. The scope of the automatic p_objects (stack allocated) is the scope of the SPMD task, however heap-allocated p_objects can outlive it.

- If the task finishes and there are no associated p_objects, the gang is *terminated* and its metadata are deleted.

- If there are still p_objects associated with the gang, then it is declared *alive* and its metadata preserved. The gang remains alive until the last p_object is deleted. RMIs can still be invoked on the p_objects.

This relationship between gangs and p_objects is what offers *isolation* and *virtualization* to the higher level components of STAPL, as they allow creating containers and invoking algorithms on a subset of the resources through PARAGRAPHs without algorithm or code modifications forming the basis of container composition and nested parallelism support. Parallel sections are isolated, and a parallel section can be given access to any p_object only through its creator gang (parent). Ability to move p_object references between gangs, parent to child, between siblings only if they go through the parent.

7.4 Gang Creation

To create a new gang, the user can:

- Partition an existing gang with a collective call over the locations that participate in the gang. The user has the ability to provide a name for the new gang to have an $O(1)$ creation cost. If such a name cannot be provided, the STAPL-RTS will perform a $O(\log n)$ communication phase to generate a new name, where n is the number of locations participating in the new gang.

- Spawn a gang via an asynchronous and one-sided manner over an existing gang. Spawning always requires $O(\log n)$ communication, where n is the number of locations participating in the new gang.

```
1  if (condition) {
2    // Create a p_object of type T in a new gang that was created collectively by
       partitioning the existing one
3    gang g1{mapping-function1, resolution-function1};
4    T t{args...};
5  }
6  else {
7    // Create a p_object of type U on the rest of the resources
8    gang g2{mapping-function2, resolution-function1};
9    U u{args...};
10 }
```

Figure 7.4: Collective gang creation

7.4.1 Collective Gang Creation

Partitioning parallel sections is supported through the creation of a `gang` object, where a provided mapping function determines which resources participate in the new section. An example of its usage is shown in Figure 7.4.

The `mapping-function` provides a way to translate from the parent gang location id to the newly created gang location id. `resolution-function` is the inverse of `mapping-function`. For example, if one wishes to create a new gang that has all the odd locations of the parent gang, then

$$\text{mapping-function}(x) = x/2$$

and

$$\text{resolution-function}(x) = 2x + 1$$

Using functions, instead of location enumeration, as MPI and other systems do, has the benefit that if a closed-form solution exists for the translation, then $O(1)$ space is used for doing the location-to-PE resolution.

```
1  // Create a p_object of type T by passing args to the constructor, in a new gang
       over the given locations and return a future to its handle
2  future<rmi_handle::reference> f1 =
3    construct<T>(location-range, args...);
4
5  // Get object handle
6  auto h = f1.get();
7
8  // Create a new p_object of type U on a new section co-located with the section of
       the first object
9  future<rmi_handle::reference> f2 =
10   construct<U>(h, all_locations, args...);
```

Figure 7.5: Asynchronous, one-sided gang creation

7.4.2 Asynchronous, One-sided Gang Creation

Spawning a parallel section is supported through the `construct` primitive as shown in Section 3.3.2. A more general example is shown in Figure 7.5. `construct` accepts a range of locations through the `location-range` argument. This does not

need to be an enumeration, as special ranges are accepted such as `all_locations` or the `level(n)` where it abstracts a level of an hierarchical machine (e.g., nodes, sockets). In all cases, the STAPL-RTS is responsible for translating location IDs to processing element (PE) IDs and for building a suitable multicast tree on the PEs which it uses to construct the section and the associated `p_object`.

7.5 Gang Metadata

A gang is identified by its ID, a system-wide unique unsigned integral value, that is used to index its *gang metadata*. The gang metadata provide facilities such as mapping of the gang's locations to PEs, as well as bookkeeping for synchronization operations. It is a distributed object that has pieces on each process that a gang has locations on.

The ID space is partitioned among the processes at the time of application execution using a block-cyclic distribution. It is guaranteed that location 0 of a gang is on the process that is the owner of the gang ID of that gang.

Since the distribution is a closed-form solution, any location can make RMIs to a `p_object` that was created in another gang G'. If the gang metadata of G' are present on the process of the source location, then it is easy to find the destination queue for the RMI, based on the protocol discussed in Section 6.3.2.

However, if the gang metadata of G' are not available, we utilize the gang ID distribution information to find where location 0 is and forward the RMI to it. That location upon receipt of the RMI will be responsible for re-routing the RMI to the actual destination. While this may create a bottleneck, we expect that gangs that frequently communicate (e.g. parent and child gangs) have metadata that are on the same subset of processes.

7.5.1 Asynchronous Creation and Destruction of Metadata

Both gang metadata creation and destruction are designed to be asynchronous operations to provide scalability. This has certain ramifications on the design of the STAPL-RTS. Firstly, since gang metadata construction is asynchronous, it may occur that RMIs arrive before the metadata has been created on a process. In order to avoid the need for synchronization during creation, we queue all RMIs until the gang metadata has been created. All incoming RMIs remain in a *deferred request buffer* and will be queued for execution once the gang metadata and the associated locations are created.

7.5.2 Gang ID Reuse

Gang IDs can only be reused when all the pieces of the gang metadata object have been destroyed. Once a gang is terminated (i.e. the SPMD task has finished and has no more `p_object`s associated with it) then its metadata are destroyed. The gang metadata are organized in a binomial tree whose root is the process of location 0. Each metadata deletion incurs a notification to its parent node in the tree. When all the metadata pieces have been destroyed (i.e. every leaf and node in the tree has sent a destruction notification) the gang ID is available for reuse.

7.5.3 Gang Metadata Sharing

A lot of times user code ends up creating a gang over the same set of PEs with the same mapping. This case raises the opportunity for sharing metadata between gangs. If the STAPL-RTS determines that two or more gangs are over the same PEs, e.g., when a new gang is created over all the locations of another gang, then part of the gang metadata is shared between all these gangs using a reference counted mechanism. This sharing not only reduces the space required, but also reduces any

potential pre-processing that happens upon metadata creation.

For the corner case that a new gang has only one location, we perform a more extreme optimization. No metadata is created, unless a `p_object` is registered or an RMI is issued. Additionally, a lot of operations, e.g., synchronization, become no-ops. This is a useful optimization, as since `PARAGRAPH` tasks are typically SPMD sections of one location.

7.6 Intragang and Intergang Communication

Communication between locations in the same gang (*intragang*) is straightforward, but the STAPL-RTS also provides the flexibility of *intergang* communication. It is possible that a parallel section must perform an operation on a container that has been created in a different section. For that reason, we allow the invocation of RMIs between locations of different gangs, assuming that the source location has an `rmi_handle` to a `p_object` of a different gang, for example through a view to a container. This way, we can control the flow of references based on the algorithm hierarchy, as a parallel section can be given an `rmi_handle` only from its parent parallel section.

An RMI that is being invoked while executing an intragang RMI is intragang if it is on a `p_object` that is part of the current gang or intergang otherwise. All RMIs that are being invoked while executing intergang RMIs are intergang.

The STAPL-RTS maintains copy semantics on for all arguments to RMIs, except `p_objects`. `p_objects` are distributed objects and they can be created in any gang G and referenced from some other gang G', as long as G gave access to its `p_objects`. A `p_object` can be passed as a C++ reference or pointer in any intragang RMI and the STAPL-RTS will take care of the conversion to and from an `rmi_handle` automatically. On the other hand, for intergang RMIs, a `p_object` can only be

passed as an `rmi_handle` reference, as the `p_object` may not have a representative on the destination location of G'. As long as a location has a reference to a `p_object` or to its `rmi_handle`, it can call RMIs on it.

7.7 Quiescence

We offer an RMI quiescence mechanism (`rmi_fence`), a collective synchronization mechanism, that guarantees that all RMIs, both intragang and intergang, have been executed. To account for intragang RMIs we use an algorithm similar to [67]. The number of sent and processed RMIs is counted at each location and when the sum of the difference of those counts for all locations is 0 for two consecutive times, then intragang quiescence has been reached.

For intergang traffic we employ a different protocol. A gang G can be given `rmi_handle`s to `p_object`s that are in different gangs and G's locations can do intergang RMIs to any of the locations in those gangs. Instead of keeping information on G about all locations that intergang RMIs have been sent to, we implement a protocol that relies on *back-edge coherence traffic*. Whenever intergang RMIs are sent from a location L in gang G to a location L' in gang G', L' sends *coherence traffic* back to L when the RMI has been executed. In order to reduce the amount of coherence traffic, multiple back-edge messages are *combined* on L'. Furthermore, when `rmi_fence` is called in G, each of its locations participates in a reduction tree that combines all the partial coherence information to location 0, which can then verify if all intergang RMIs have finished.

Quiescence is an operation private to a gang. This is an important property as it offers isolation in the communication traffic: two unrelated gangs have independent accounting of traffic and therefore, their respective quiescence is independent from one another. Composed containers and nested `PARAGRAPH`s rely on this compartmen-

talization to avoid performance pitfalls. For example, when all the writes initiated from a parallel algorithm have to become visible, then the `PARAGRAPH` responsible for executing the algorithm can issue an `rmi_fence` and continue when only the related RMIs have finished.

7.8 Virtualization of Resources

Creating multiple gangs over the same set of PEs effectively virtualizes the computing resources and it bears close resemblance to $M : N$ schedulers. Locations for different gangs may be mapped to the same PE.

Containers are aware of the locality of their elements, which they convey to the `PARAGRAPH` through views, so that the former can perform efficient task placement according to its placement policy, a process called *localization*. Naturally, localization is important for performance.

While the STAPL-RTS is able to answer in a lot of instances the question of if two different locations are on the same PE and make localization straightforward. However, given the asynchronous nature of the STAPL-RTS, the fact that containers and `PARAGRAPH`s may live in different subsets of the machine and that the location-to-PE mapping is distributed, the answer to this question is challenging.

In order to compare localities, we have developed a lightweight PE description scheme called an *affinity tag* which encodes the PE's place in the machine hierarchy. A location mapped to a specific PE inherits the affinity tag of the latter.

When a container is created, each one of its pieces tagged with the affinity of the location it is created on. Views are using these affinity tags to provide information on data locality, which the `PARAGRAPH` uses to do task placement. Should multiple views that reference data in different affinities exist, the `PARAGRAPH` has various task placement algorithms it can employ to perform the placement, which is beyond the

scope of this dissertation.

As an illustrative example of an affinity tag, assume that we have a machine with 2 nodes, 2 sockets per node and 16 cores per socket, we can fully describe the machine by creating affinity tags with 6 bits (1 bit for the nodes, 1 bit for the socket and 4 bits for the cores). The 4th core of the 2nd socket in the 1st node is tagged with 010011. Up to 64 bits are allowed in this representation, allowing us to describe even the largest available machine. This representation has the advantage that a simple subtraction gives the distance between two different processing elements, and a simple integer comparison can tell if any two locations are on the same PE.

Currently, we rely on OpenMP for creating affinity tags. The OpenMP runtime determines thread-to-core binding, either automatically, using pragmas or environment variables. The STAPL-RTS inherits the binding and provides it to the rest of the framework as an affinity tag. This affinity tag is fundamentally logical, as it relies on the information provided from a lower level runtime system. In the future, we plan to support affinity tags that utilize information from libraries such as `hwloc` [97].

7.9 Scheduling

`PARAGRAPH`s may create parallel sections that overlap on the same PEs. In order to enforce isolation and preserve the algorithm structure, we employ a *hierarchical scheduling* approach. Each `PARAGRAPH` is associated with its own EXECUTOR, a distributed object that receives runnable tasks and scheduling information from the `PARAGRAPH` and dispatches said tasks when appropriate. While a default First-In/First-Out (FIFO) policy is provided, the scheduling policy can be changed by the algorithm developer at the point of `PARAGRAPH` instantiation.

`PARAGRAPH` tasks run to completion and therefore priority inversion is avoided as tasks cannot be preempted, except in cases where a lower priority task is runnable,

and a higher priority task is not because of unsatisfied dependencies. These cases can be resolved by special scheduling policies that disallow runnable lower priority tasks from executing until the higher priority tasks have been scheduled. We delegate this to the scheduling policy, since this information has to be provided from the application itself.

Each executor with runnable tasks (therefore, a runnable EXECUTOR) is inserted with associated scheduling information to the parallel section EXECUTOR, called a GANG_EXECUTOR. Since we support the notion of *non-blocking* EXECUTORs a GANG_EXECUTOR may have more than one EXECUTORs to schedule. The GANG_EXECUTOR is responsible for dispatching EXECUTORs according to the scheduling policy of the parallel section and each dispatched EXECUTOR dispatches its runnable tasks. This hierarchical scheme guarantees that the scheduling policy is conserved within an EXECUTOR and across all EXECUTORs in a parallel section.

`PARAGRAPH` tasks can invoke parallel algorithms by creating new `PARAGRAPH`s. This will result in the creation of new parallel sections with their own GANG_EXECUTORs. GANG_EXECUTORs create an hierarchy based on the parallel section hierarchy. A GANG_EXECUTOR that has runnable EXECUTORs is inserted to the GANG_EXECUTOR of its parent parallel section.

7.10 Related Work

Nested parallelism was first used for expressiveness, as in NESL [30]. The NESL compiler applies flattening, transforming all nested parallelism algorithms to a flat data parallel version, which may limit performance. Other parallel programming systems use nested parallelism for performance. Users express algorithms using nested sections for the sole purpose of exploiting locality.

The treatment of nested parallelism in modern parallel programming models can

be used to divide the previous work on the topic into the following categories: user managed, structured nested parallelism, and recursive task spawning.

User managed. Models such as MPI [33] are generally single level, requiring programmers to flatten any nested algorithm calls manually. Support that is provided to support nested structure (e.g., communicators) is provided by primitives whose use are intertwined with the program specification.

MPI [90] allows creating new MPI communicators by partitioning existing ones or by spawning additional processes. This functionality can be used to map nested parallel algorithms to the machine hierarchy. It is well suited to static applications. Applications with input sensitive nested parallelism are difficult to express because communicator creation is always collective and each process must know through which MPI communicator it should communicate at any given point in the program.

OpenMP [35] has had nested parallelism capabilities since its inception. There is work on nested parallelism for performance [37]. However, the `collapse` keyword in OpenMP 3.0 that flattens nested parallel sections attests to the difficulty of gaining performance from nested parallelism in OpenMP.

With hybrid protocol approaches to nested parallelism such as hybrid MPI and OpenMP [35, 88, 89], the application code becomes more complex, with ad-hoc solutions for data and computation placement, as performance gains are realized. Looking toward exascale computing, we do not believe that the user managed approach is feasible, as the required user effort will increase greatly with the deep system hierarchies that must considered for performance.

Structured nested parallelism. Several models including [98, 99, 100] have been successful in allowing programmers to use nested algorithms for static applications. Nested algorithm invocations can exist in the code without explicit management of mapping (this is either done automatically, based on a data distribution,

or through a separate, user specified mapping class as in [99]). These invocations usually occur in isolated sections so that the application level algorithmic structure is maintained.

Systems in this category enhance the MPI approach, while simplifying the programming model. Neststep [101] is a language that extends the BSP (bulk synchronous parallel) model and allows the partitioning of the processing elements that execute a superstep to subgroups that can call any parallel algorithm. These subgroups must finish prior to the parent group continuing with the next superstep. UPC [28] and Co-Array Fortran [102] have similar restrictions.

Titanium [98] and UPC++ [55] introduce the Recursive Single Program Multiple Data (RSPMD) model and provide subgrouping capabilities, allowing programmers to call parallel algorithms from within nested parallel sections that are subsets of the parent section. Similarly to Neststep, they also require that the nested sections finish before resuming work in the parent section.

The Sequoia [99] parallel programming language provides a hierarchical view of the machine, enforcing locality through nested parallelism and thread-safety with total task isolation: tasks cannot communicate with other tasks and can only access the memory address space passed to them. This strong isolation, in conjunction with execution restrictions to allow compile-time scheduling of task scheduling and task movement, limit its usefulness in dynamic applications.

Phalanx [103] provides capabilities to asynchronously spawn SPMD tasks that execute on multiple threads. Programmers allocate memory explicitly on the supported devices (CPU, GPU, etc.) and invoke tasks on them, creating parallel sections. Phalanx has a versatile programming model and is the most similar related work to the STAPL-RTS. Its main difference from the STAPL-RTS is that Phalanx requires explicit control of resources. Data and task placement needs to be statically specialized with

the target (e.g., GPU, thread, process), transferring the responsibility of resource management to the user and creating the need for multi-versioned code.

Recursive task spawning. Dynamic applications are not easily expressible in many of these static models, as this type of nested parallelism typically requires more centralized coordination for the creation, execution, and quiescence of nested sections (i.e., they are blocking collectives). When the need for parallelism is not known *a priori* due to input sensitivities, more flexible methods to dynamically spawn more work are required. Task based models are effective in creating the desired parallelism. However, they are often more difficult to program due to the loss of isolation based on program structure. The scheduling and execution of tasks is usually flattened, and synchronizations become more coarse grained, as the semantic information grouping tasks for a given nested computation is lost. These shortcomings inevitably affect the expressiveness provided by these programming models.

Several systems support task-based parallelism, allowing the user to spawn tasks from other tasks. The programmer can thus express nested parallelism with the system responsible for placement. These include Intel Thread Building Blocks [42] and Cilk [104]. Since task placement is done in absence of knowledge about locality, one of the benefits of nested parallelism is lost.

X10 [105], Habanero-Java [43], HPX [92], Fortress [106] and Grappa [107] all offer task-based parallelism, going a step further and allowing control over task placement. However, they suffer from loss of structure of the algorithms during execution, as tasks are independent of each other. Building on top of Habanero, Otello [100] addresses the issue of isolation in nested parallelism. While maintaining a task parallel system, Otello protects shared data structures through analysis of which object each task operates on and the spawning hierarchy of tasks.

The Charm++ [51] developers have extended their messaging model to shared

memory communication as shown in [108]. In [109] the authors explore the potential nested parallelism capabilities of Charm++ using a kd-tree benchmark that introduces an ad-hoc fork-join model in Charm++.

Chapel [52] is a multi-paradigm parallel programming language and supports nested parallelism. While it supports data and task placement, users are given only two parallel algorithms (parallel for, reduce). Other parallel algorithms have to be implemented explicitly using task parallelism.

Legion [110] retains Sequoia's strong machine mapping capabilities and it relaxes many of the latter's assumptions, making it a good fit for dynamic applications. It follows a task parallel model in which tasks can spawn subtasks with controlled affinity. However, this process leads to loss of information about the structure of the parallel sections, which is a common issue in other task parallel systems.

From the Trilinos package [50], Kokkos supports nested parallelism by allowing users to divide threads in a team. Teams can be further divided and threads that belong to a team are concurrent. However, teams cannot execute concurrently, and only three algorithms (parallel for, reduce and scan) are available to be invoked from within a nested parallel section.

Table 7.1 summarizes the main differences between STAPL's support of nested parallelism and similar approaches.

7.11 Experimental Evaluation

In this section, we measure the performance of our one-sided nested parallelism primitives on both static application kernels, as well as several dynamic ones, such as graph algorithms.

Name	SPMD NP sections	Asynchronous	Locality Aware	Any algorithm allowed in NP section
MPI	Yes	No	Manual	Yes
UPC++, Co-Array Fortran, Titanium	Yes	No	Manual	Yes
Sequoia	Yes	No	Compile-time	Yes
Habanero, X10	No	Yes	Yes	Yes
Chapel	No	Yes	Yes	No
Charm++	No	Yes	Yes	Yes
Legion	No	Yes	Yes	Yes
Phalanx	Yes	Yes	Manual	Yes
STAPL	Yes	Yes	Yes	Yes

Table 7.1: Nested Parallelism (NP) capabilities comparison

7.11.1 Gang Creation

In Figure 7.6 we present the results of a microbenchmark that compares the one-sided (`construct`) and collective (`gang`) section creation against the collective `MPI_Comm_create`[†] over the same number of processes, when the global parallel section is 512 processes on Cray-XK7. The combined effect of asynchronous creation and deletion result in competitive performance against MPI. Moreover, it shows that the one-sided parallel section creation is a scalable approach.

The last line, `construct(all_locations)`, takes advantage of the gang metadata reuse mentioned in Section 7.5. This happens in the benchmark by creating initially a gang over a subset of the locations and every subsequent call uses a `p_object` created in that gang to create a new using the `construct` call. This results in reduction of the overhead to create a new section, as it shares the metadata with an already existing one, removing the need to recreate the metadata. We expect that finding more opportunities for metadata reuse will reduce the collective and one-sided section creation cost.

[†]MPI does not offer one-sided communicator creation functionality.

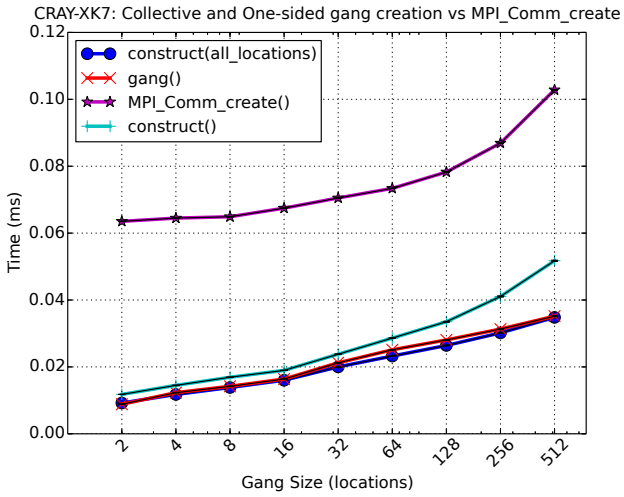


Figure 7.6: One-sided (`construct`) and Collective (`gang`) vs MPI on 512 processes on Cray-XK7

7.11.2 Intragang vs Intergang Communication

In Figure 7.7 we evaluate the overhead of intergang RMIs over intragang RMIs using the MPI latency benchmark from [74] on Cray-XK7. Intergang RMIs incur an additional overhead of about 1 us due to some additional bookkeeping required, as discussed in Section 7.7. MPI results are shown using both `MPI_Send`/`MPI_Recv` pairs and one-sided MPI-2 calls (`MPI_Put`/`MPI_Win_flush`).

7.11.3 SAXPY

SAXPY stands for "Single-Precision AX Plus Y" and is a Level 1 routine in the standard Basic Linear Algebra Subprograms (BLAS) library [111]. SAXPY uses two input vectors of 32 bit floats X and Y with N elements each, and a scalar value A and it multiplies each element $X[i]$ by A and adds the result to $Y[i]$.

We evaluate the overhead of creating nested parallel sections using a `SAXPY` kernel written directly using STAPL-RTS primitives in Figure 7.8. We compare a straightforward SAXPY implementation ("flat") against a version that recursively divides the

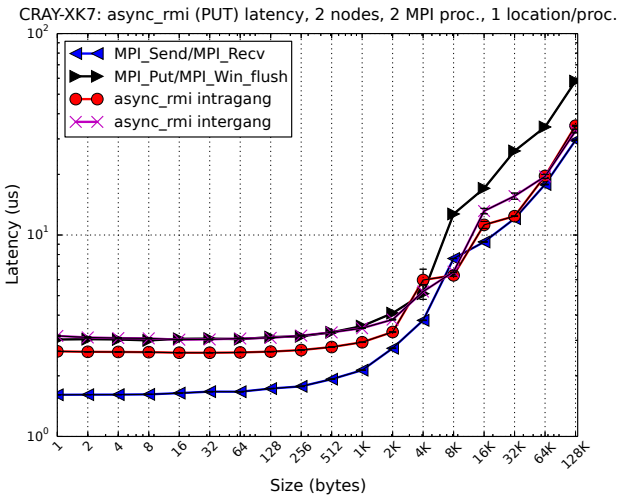


Figure 7.7: Intragang vs intergang asynchronous RMI latency

input vectors in half 4 times by creating nested parallel sections using the `construct` primitive. This figure is complimentary to Figure 7.6, as the latter shows the cost of one-sided creation with minimal computation, and shows that the overhead of nested parallel section creation is minimal.

7.11.4 NAS Conjugate Gradient

The NAS Conjugate Gradient (CG) benchmark [76] estimates the largest eigenvalue of a symmetric positive definite sparse matrix using the inverse power method. It employs the conjugate gradient method which uses matrix vector multiplication.

We compare a STAPL CG implementation using one-sided nested parallelism against the reference NAS CG MPI implementation on IBM-BG/Q. The STAPL-RTS shared memory optimizations [15] are disabled to provide a fairer comparison as there is not a hybrid implementation (i.e., MPI+OpenMP) of the reference.

The reference implementation distributes the matrix in a 2D block manner and uses MPI communicators for ad-hoc nested parallelism, dividing the P processors evenly into $\sqrt{P}$ groups. In the matrix vector multiplication, each processor performs

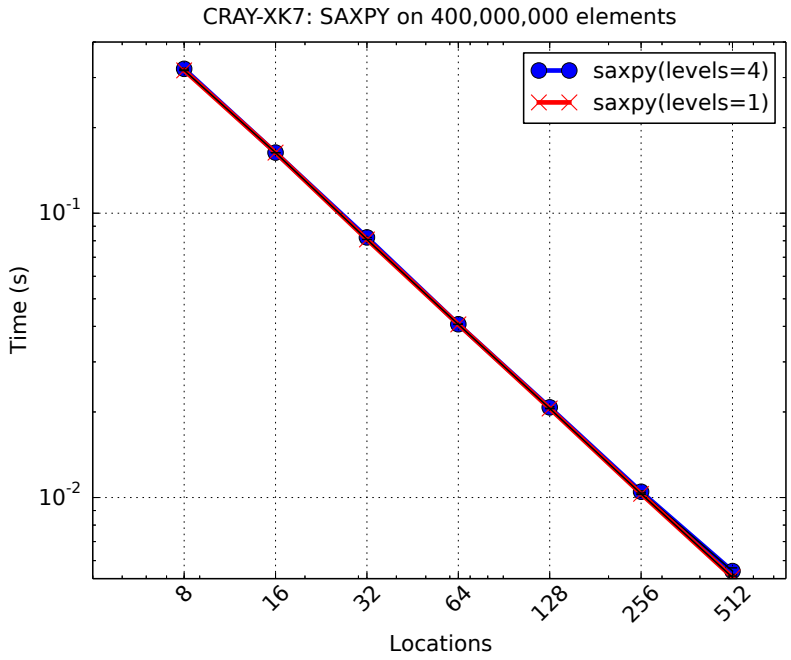


Figure 7.8: SAXPY with no nested parallel sections ("flat") and 3 nested parallel sections created ("nested") on Cray-XK7 (log-log graph)

local work before a per group recursive doubling phase. A global pairwise exchange is done next to effect a transposition of the vector.

The STAPL implementation uses a composed container (i.e., array of arrays) for the row-wise vectors: each inner array is distributed in a gang of size $\sqrt{P}$. The matrix is distributed as in the reference implementation. A nested invocation of `map(inner_product())` performs the matrix vector multiplication. This is followed by $\sqrt{P}$ parallel broadcasts to implement the vector transpose.

The implementations of NAS CG perform the same data distribution and computation. The advantage of the STAPL implementation is the separation of the specification of the computation from its mapping to the system. The MPI implementation intertwines the data distribution and explicit communication together with the computation of the algorithm. STAPL separates the data distribution specification from

the communication pattern for the vector transpose, and the conjugate gradient implementation.

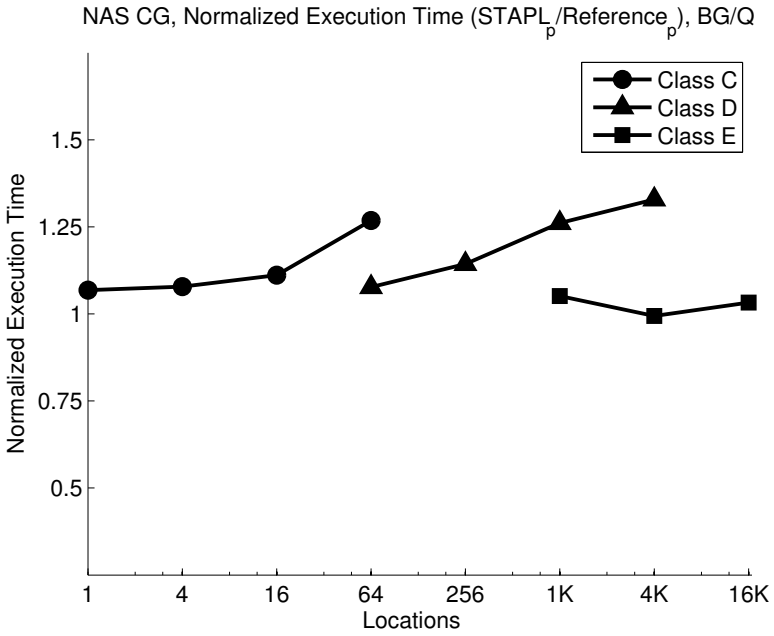


Figure 7.9: NAS CG Class C, D, and E on IBM-BG/Q

Results for CG on IBM-BG/Q are shown in Figure 7.9. Given the wide range of core counts (1..16K), a weak scaling would be preferred but is not possible as NAS problem sizes are fixed across core counts. We use 3 NAS problem classes (C, D, E) to span the range of processors. The plots show the STAPL execution time normalized to that of the reference implementation. Besides a single data point (Class E, 4K processors), the STAPL version is slightly slower than the reference (up to 25% at Class C, 4K processors).

In addition to the results shown, the STAPL version also ran at 65K cores, but the times could not be shown as the reference failed to compile (a new binary is generated

for each combination of problem class and core count). The STAPL implementation is performing on par with the reference implementation and doesn't require parameters to be hardwired when the code is compiled.

7.11.5 Minimum Element using Composed Containers

In this section, we compare a nested parallel implementation of finding the minimum element over a composed container (`stapl::array<stapl::array<int>>`) against a flat parallel implementation with a distributed container of non-distributed containers (`stapl::array<std::vector<int>>`).

For the `stapl::array<std::vector<int>>` version we find the minimum element by invoking a parallel `stapl::min_element` algorithm over the results of `std::min_element` calls over the inner `std::vector<int>` containers whereas for the container composition version (`stapl::array<stapl::array<int>>`) we recursively call `stapl::min_element` over the outer and the inner containers.

In Figure 7.10 we compare the two versions in which the outer container, C, is a `stapl::array` with n elements, where n is the total number of locations that the experiment is run on. The inner containers c are either `std::vector<int>` for the flat implementation of minimum element or `stapl::array<int>` for the nested parallel implementation.

Each inner container c_i has $10,000 + 400,000 * i$ `int`s, where i is the index of c in the outer container C ($c_i = C[i]$). For example for the data point on 512 locations, the C has 512 inner containers where the c_0 has $10,000$ `int`s and c_{511} has $204,810,000$ `int`s.

This example intentionally creates imbalance, where the higher the location ID, the more elements the inner container has. In the case of the composed STAPL containers `stapl::array<stapl::array<int>>` the inner containers are distributed

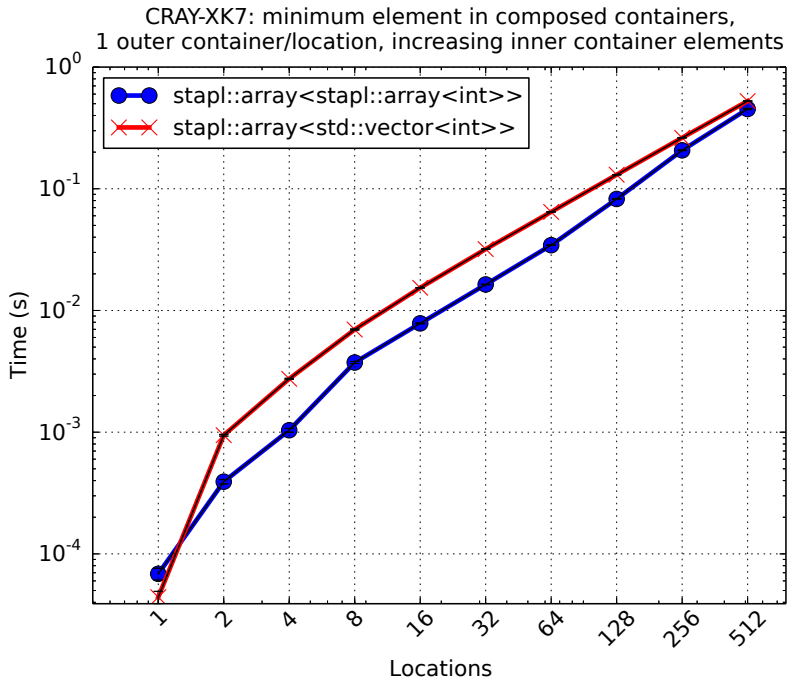


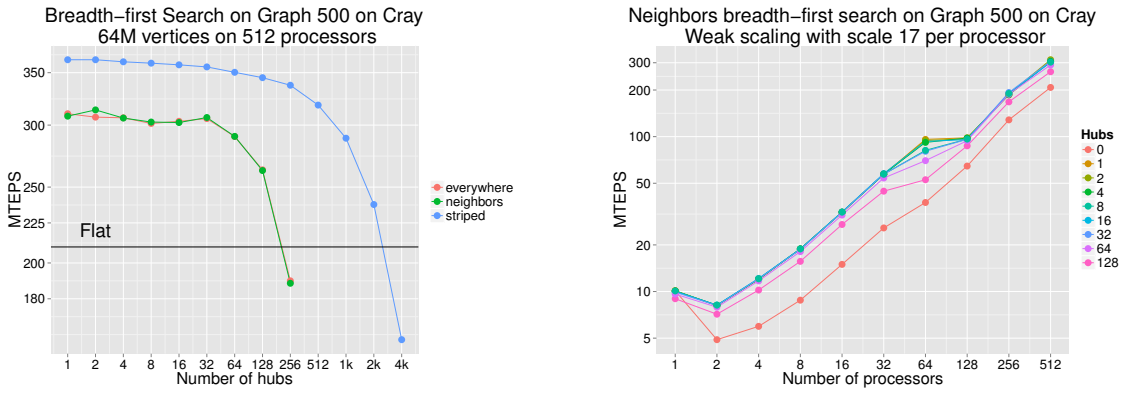
Figure 7.10: `min_element` on `array<array<int>>` (log-log graph)

across all n locations. While we over-distribute the inner containers, in turn increasing the number of nested parallel sections that find the minimum element in the inner containers, invoking nested parallel algorithms presents the benefit of more efficiently distributing the work across the system resulting in better performance.

7.11.6 Breadth First Search

Processing large-scale graphs has become a critical component in a variety of fields, from scientific computing to social analytics. An important class of graphs are *scale-free* networks, where the vertex degree distribution follows a power-law. These graphs are known for the presence of *hub vertices* that have extremely high degrees and present challenges for parallel computations.

In the presence of hub vertices, simple 1D partitioning (i.e., vertices distributed,



(a) (b)

Figure 7.11: Graph 500 breadth-first search on CRAY-XK7 varying (a) the number of hubs on 512 processors and (b) the number of processors for a weak scaling experiment.

edges collocated with corresponding vertex) of scale-free networks presents challenges to balancing per processor resource utilization, as the placement of a hub could overload a processor. More sophisticated types of partitioning have been proposed [112, 113, 114, 115], however these strategies often change both the data representation as well as the algorithm.

We represent the graph as a distributed array of vertices, with each vertex having a (possibly) distributed array of edges. Using `construct` we define several strategies for distributing the edges of hub vertices, that can be interchanged without changing the graph algorithm itself. The first distribution strategy (EVERYWHERE) places a hub's adjacency list on all locations of the graph's gang. The second (NEIGHBORS) places the edges only on locations where the hub has neighbors. This strategy is especially dynamic as the distribution of each hub edge list is dependent on the input data. Thus, we rely heavily on the arbitrary subgroup support of STAPL-RTS. The last strategy (STRIPED) distributes the adjacency list on one location per shared-memory node in a strided fashion to ensure that no two hubs have edges on the same

location. Even though the distribution strategy of the edges changes, the edge visit algorithm remains unchanged.

To validate our approach, we implemented the Graph 500 benchmark [77], which performs a parallel breadth-first search on a scale-free network. In Figure 7.11(a), all three edge distribution strategies fare well over the baseline of non-distributed adjacency lists for modest number of hubs, and then degrade in performance as more vertices are distributed. The EVERYWHERE and NEIGHBORS strategies behave similarly, as the set of locations that contain any neighbor is likely to be all locations for high-degree hub vertices. The EVERYWHERE and NEIGHBORS strategies are 49% and 51% faster than the baseline, respectively. The STRIPED strategy performs up to 75% faster than the baseline, which is a further improvement over the other strategies. On CRAY-XK7, cores exhibit high performance relative to the interconnect, and thus even modest amounts of communication can bring about large performance degradation. The STRIPED strategy reduces the amount of off-node communication to create the parallel section from the source vertex location, bringing the performance of the algorithm above the other two strategies. We are investigating this phenomenon to derive a more rigorous model for distributing edge lists.

Figure 7.11(b) shows a weak scaling study of the neighbor distribution strategy on CRAY-XK7. As shown, the flat breadth-first search scales poorly from 1 to 2 processors due to an increase in the amount of communication. By distributing the edges for hubs, we reduce this communication and provide better performance than the flat algorithm. The number of distributed hubs must be carefully chosen: too few hubs will not provide sufficient benefit in disseminating edge traversals, whereas too many hubs could overload the communication subsystem.

In order to evaluate our technique at a larger scale, we performed a breadth-first search on the Graph 500 graph on IBM-BG/Q in Figure 7.12(a). We found that

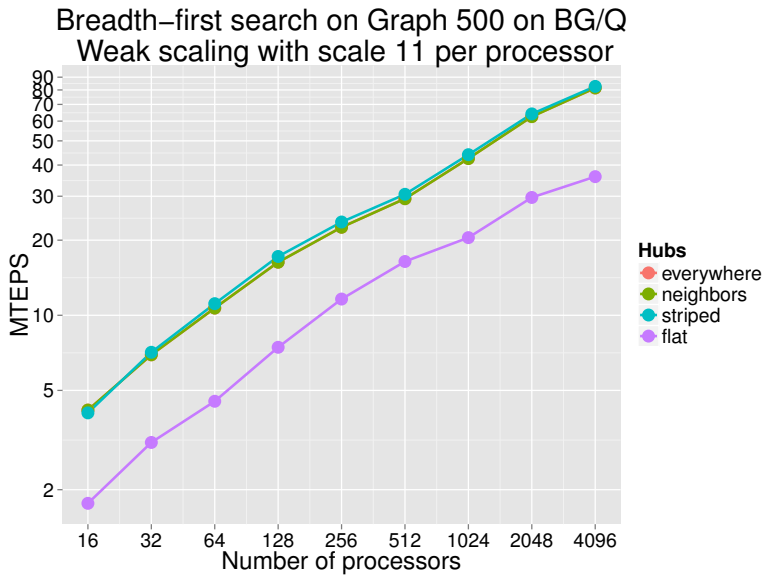


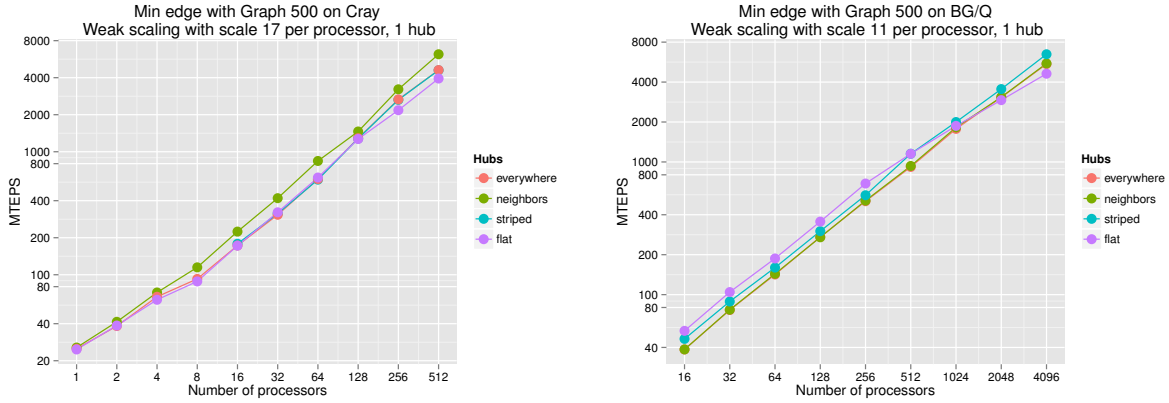
Figure 7.12: Graph 500 breadth-first search with various adjacency distributions on IBM-BG/Q

although faster than the flat version, all three distribution strategies performed comparably with each other. At 4,096 processors, the distributed adjacency list versions of breadth-first search are 2.25x faster than the flat baseline. Hence, the distribution strategy is machine-dependent, further reinforcing the need for a modular and algorithm-agnostic mechanism to explore the possible configuration space for nested parallelism in parallel graph algorithms.

7.11.7 Minimum Edge Weights

Finding the incident edge of each vertex with the minimum edge weight is an important operation that occurs in various graph algorithms, including Boruvka's minimum spanning tree algorithm [116]. This operation is a natural fit for nested parallel execution, as each vertex can spawn an asynchronous nested parallel algorithm to find the minimum edge weight amongst all of the edges in its adjacency list.

Following the same optimization of distributing only hub vertices, we are able to dynamically choose between a sequential or nested parallel reduction on a per-vertex basis, dependent on the degree of the vertex itself.



(a) (b)

Figure 7.13: Minimum weight edge with the Graph 500 input on (a) CRAY-XK7 and (b) IBM-BG/Q

Figure 7.13 compares the throughput of finding minimum edge weights in a weak-scaling experiment on the Graph 500 input. On CRAY-XK7, we find that the NEIGHBORS strategy performs best and provides a 1.5x improvement at 512 cores. On the other hand, the flat strategy on IBM-BG/Q is initially better than all distributed strategies, but is outperformed by the STRIPED strategy at scale and we see a 1.4x improvement at 4,096 cores.

8. CAUSAL RMI ORDERING*

The STAPL-RTS abstracts the platform, providing a unified communication interface over shared and distributed memory. As explained in Section 3, our communication abstraction is based on Remote Method Invocations (RMIs) on distributed objects. While allowing RMIs to execute in any order is a tempting proposition, as it can expose a lot of performance opportunities, this would create a cumbersome, difficult to program model which can lead to unnecessary synchronization.

Thus, it is important that a runtime system offers a virtualization layer with well defined communication ordering guarantees. An appropriate model will allow the user to reason about the order of operations without excessive synchronization. In addition, to support the application-centric computing vision, an application should have the appropriate mechanisms to influence the model to enable optimizations, such as relaxing the ordering constraints if the algorithm can tolerate that.

The STAPL-RTS RMI ordering scheme, *Causal RMI Ordering* (CRMIO), attempts to combine traditional message ordering schemes with the dynamic nature of RMIs. In this chapter we will

- *motivate* why ordering RMIs is a desirable property,
- *formalize* the RMI ordering guarantees as provided by the STAPL-RTS and compare it against similar schemes,
- *describe* its use cases, such as in the STAPL framework, and

*Part of this chapter is reprinted with permission from "STAPL-RTS: An Application Driven Runtime System" by Ioannis Papadopoulos, Nathan Thomas, Adam Fidel, Nancy M. Amato, Lawrence Rauchwerger, 2015. *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15*, 425–434, Copyright 2015 by ACM.

- *evaluate* the performance benefits of relaxing those guarantees when this can be safely communicated from the algorithm to the STAPL-RTS.

8.1 Related Work

The STAPL-RTS is primarily a distributed system that offers a form of distributed shared memory (DSM) [26] via presenting a shared object view to its users. Therefore the related work cannot be constrained to distributed systems only, but must extend to the memory consistency models to fully understand the implications of the RMI ordering guarantees of the STAPL-RTS.

Starting with shared memory and DSMs, a number of memory consistency models have been described. Stricter consistency models are simpler to program with and reason about, while weaker ones offer performance benefits, especially as memory latency increases, to the expense of a more convoluted programming model.

Sequential Consistency (SC) [117] defines a memory to be sequentially consistent if "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." Data-race free C++ programs are SC [118].

In Linearizability [119], writes and reads can overlap and are not instantaneous, and a total order is imposed on the operations in the concurrent execution. According to this order, each read returns the value written by the latest preceding write. The total order is consistent with the order of non-overlapping operations in the concurrent execution. While linearizability seems as prohibitive in terms of performance, in [120] the authors use linearizability guarantees through futures to provide thread-safe containers with better performance than lock-free solutions.

In Processor Consistency (PC) [121], processors agree on the order of writes from

each processor but can disagree on the order of writes from different processors to different memory locations.

Under the Pipelined Random Access Memory (PRAM) [122] consistency model, writes from a processor are seen by all other processors in the order in which they were issued, but writes from different processors may be seen in a different order by different processors. Using an implementation example, each processor has a local copy of the global memory and reads are performed on that local copy. When a processor makes a write, it broadcasts the new value to all the processors' copies, including its own, and all writes are processed in order, hence the term "pipelined".

Causal Consistency (CC) [123] is based on Lamport's concept of potential causality [70]. CC is a consistency model that is between SC and PRAM. Causality of operations is determined by program order and a *writes-into* order. If o_1 and o_2 are two operations, then program order means that $o_1 \underset{i}{\rightarrow} o_2$ for some processor p_i, if o_1 precedes o_2 in processor i. Writes-into order ($\mapsto$) associates a write operation with each read operation ($o_1 \mapsto o_2$, with write operation $o_1 = w(x)v$ and read operation $o_2 = r(x)v$). A causality order between o_1 and o_2 ($o_1 \rightsquigarrow o_2$) exists if either program order or write-into order exists between o_1 and o_2 or an operation o' exists such that $o_1 \rightsquigarrow o' \rightsquigarrow o_2$. A history of operations H is causal if for each process i there is a serialization of the operations that respects $\rightsquigarrow$. A memory is causal if it admits only causal histories of operations.

Relaxed consistency models such as Weak Consistency (WC) [124] offer sequentially consistent access to synchronization variables, while accesses to the synchronization variables act as barriers across which accesses cannot be reordered. MPI RMA (Remote Memory Access) [125] is an example of WC in DSMs; users are required to issue explicit synchronization operations to guarantee that reads and writes have completed.

Release Consistency (RC) [126] extends WC with acquire, release and non-synchronizing accesses. Acquire accesses prohibit future accesses only, while release accesses ensure that past accesses have finished. Munin [127] uses RC, requiring the users to explicitly define which variables were under RC and which were not to achieve performance at the expense of usability.

Building on top of RC, Lazy Release Consistency (LRC) [128] defers all updates to memory until the next acquire operation, reducing overall communication. An example of LRC use is Treadmarks [84].

Entry Consistency (EC) used in the Midway system [129] requires shared variables to be associated with a synchronization variable, allowing RC on such variables. Midway, like Munin, requires the programmer to declare which variables are protected.

Location Consistency (LC) [130] relaxes the *memory coherence* assumption [126][†] and offers a partially ordered set of writes for each object (memory location). It is more relaxed than RC or EC and requires that locations have to be explicitly listed for acquire and release operations to establish an order of operations between different processors for the same object.

Distributed memory communication libraries such as MPI [33], GASNet [49] and Active Messages (AM) [45], provide FIFO guarantees for communication through messages. For example, in MPI, two MPI messages send through the same communicator to the same rank and tag, will be received in the issuing order. Similar guarantees apply for GASNet and AM. These guarantees, while they are enough when dealing with programming models similar to Communicating Sequential Processes (CSP) [131], are usually not enforced in shared memory, requiring the programmer

[†] "All writes to the same location are serialized in some order and are performed in that order with respect to any processor."

to distinguish between shared and distributed memory.

All parallel programming systems respect the consistency model of the underlying language for shared memory execution, however they offer little to no guarantees when it comes to task or communication request execution order, especially over distributed memory. The user is required to use either point-to-point, e.g., acknowledgment mechanisms, or collective synchronizations, e.g., quiescence detection [67] or phasers [132].

For example, Charm++ [51] applications are written by sending messages to migratable objects called *chares*. Each chare lives on a processing element (PE) but is globally addressable. Messages are typically processed in the order they arrive (First In First Out or FIFO). However, if the target object (chare) is migrated, Charm++ will automatically start forwarding messages, providing no guarantee that two messages from the same source will be executed in order.

Similarly, in work-stealing languages and frameworks, including but not limited to Cilk/Cilk++ [39, 40], TBB [42], Habanero-Java [43] and its other variants, the user is responsible for enforcing ordering between tasks.

X10 [53], Chapel [52], UPC [28] and others, offer a SC model for local accesses and remote accesses that happen between the same source and destination. While they do provide constructs that give SC guarantees even to tasks (e.g. Chapel's `coforall`), in general they offer relaxed consistency.

Distributed systems on the other hand often present a more intuitive model. Based on the work in [70], happened-before relations are established between events or messages that allow users to create a partial order of events. As mentioned, the STAPL-RTS operates always as a distributed system, taking advantage of shared memory transparently. Therefore, its RMI ordering guarantees have a lot in common with [70] and the Medium Futures Linearizability (Medium-FL) model presented

in [120]. The main differences are that the primitives offered by the STAPL-RTS create their own execution context, extending the notion of process from [70], and ordering is maintained for asynchronous primitives (e.g., `async_rmi`) that never return its result, as opposed to [120].

8.2 Preliminaries

As mentioned previously, the STAPL-RTS abstracts a processing element (PE) using the concept of a *location*, a virtual isolated address space with associated execution capabilities (e.g. thread). Locations communicate between each other through *Remote Method Invocations* (RMIs) on *distributed objects* (`p_object`s).

A `p_object` consists of logically associated objects, or *representatives*, each of which is owned by a location. The location can read and write to the local representative of a `p_object` directly, however any remote reads and writes happen only through RMIs. An RMI call targets one or more representatives of a `p_object` as shown in Section 3.4. Point-to-point RMIs call the function on one representative, while collective and one-sided collective operations call the same function on multiple representatives. For now we will only consider the case of point-to-point RMIs.

RMIs are executed on the corresponding destination location on the representative of the `p_object` that resides on that location (*owner-computes*). RMIs are executed *sequentially* and *atomically*, as the execution of an RMI is not preempted, unless an STAPL-RTS primitive is encountered.

8.2.1 Direct Memory Access

An RMI can make an arbitrary number of reads and writes on the target `p_object` representative and, as a consequence, to the location's memory address space. Apart from the obvious reads/writes on the target `p_object`, the arguments to the RMI can reference other `p_object`s as well. An example is given in Figure 8.1. When

```
1  struct A : public stapl::p_object {
2    int m_value;
3
4    void write(int t) { m_value = t; }
5
6    int read() const { return m_value; }
7
8    void write_direct(int t, A* p) {
9       m_value = t;
10      p->m_value = (t+1);
11   }
12 };
13
14 foo(...) {
15   A a1;
16   A a2;
17
18   stapl::async_rmi(1, a2.get_rmi_handle(), &A::write, 5);
19   stapl::async_rmi(1, a1.get_rmi_handle(), &A::write_direct, 6, &a2);
20
21   int r1 = stapl::sync_rmi(1, a1.get_rmi_handle(), &A::read);
22   int r2 = stapl::sync_rmi(1, a2.get_rmi_handle(), &A::read);
23
24   assert( r1==6 && r2==7 );
25 }
```

Figure 8.1: Direct `p_object` access

the RMI to the function `A::write_direct` executes, the writes to `a1` and `a2` are performed atomically.

8.2.2 Indirect Memory Access

During the execution of an RMI, other RMIs may be invoked that target the same or other `p_objects`. An example of this is shown in Figure 8.2. While the value $r1$ is expected to be 6, without any quiescence (e.g. `rmi_fence`) it is uncertain what the value of $r2$ is. However, if the value of $r2$ is 7, then we can be sure that the value of $r1$ is 6, a fact that implies that there is a *causality* between the effects of the execution of different RMIs. We refer to RMIs that have been called from other RMIs as *nested RMIs*.

```
1  struct A : public stapl::p_object {
2    ...
3
4    void write_indirect(int t, A* p) {
5       m_value = t;
6       stapl::async_rmi(2, p->get_rmi_handle(), &A::write, t+1);
7    }
8  };
9
10 foo(...) {
11   A a1;
12   A a2;
13
14   stapl::async_rmi(2, a2.get_rmi_handle(), &A::write, 5);
15   stapl::async_rmi(1, a1.get_rmi_handle(), &A::write_indirect, 6, &a2);
16
17   int r1 = stapl::sync_rmi(1, a1.get_rmi_handle(), &A::read);
18   int r2 = stapl::sync_rmi(2, a2.get_rmi_handle(), &A::read);
19
20   assert( r1==6 && (r2==5 || r2==7) );
21 }
```

Figure 8.2: Indirect `p_object` access

8.3 Causal Remote Method Invocation Order

As mentioned in previous chapters, RMIs may return the result of the invocation, e.g., `opaque_rmi`, `sync_rmi`, or they may discard it, e.g., `async_rmi`.

Claim 1. *An RMI request call that discards results behaves the same as an RMI request call that returns the result if we choose to discard said result.*

Additionally, RMI request calls that return the result of the function can be blocking, e.g., `sync_rmi` or non-blocking, e.g., `opaque_rmi`. A non-blocking RMI can always become blocking if the value is requested immediately, for example by calling `future::get` on the returned `future` from an `opaque_rmi`.

Claim 2. *A blocking (synchronous) RMI can be treated the same as a non-blocking (asynchronous) RMI if the result is immediately requested.*

Up until now, we have only looked at point-to-point primitives. Collective and

one-sided collective operations have similar effects and properties as their point-to-point counterparts, but they operate on multiple locations.

Claim 3. *Collective and one-sided collective operations can be logically replaced by a series of point-to-point operations without loss of generality.*

Each RMI is effectively a sequential instruction stream that executes uninterrupted until a STAPL-RTS primitive is encountered. When such a primitive is called, control is transferred to the STAPL-RTS which is free to schedule any other request for execution. Only after the STAPL-RTS is finished control is returned to the caller.

Definition 5. *A* context *is an environment in which a sequential instruction stream, such as an RMI, executes.*

Locations that execute SPMD (Single Program Multiple Data) [24] parallel sections have a context for each instance of the section[‡], that is each location is associated with its own context C_l, where l is the location ID.

SPMD sections can invoke RMIs and RMIs can invoke other nested RMIs (Section 8.2.2). RMIs are executed in a context and multiple RMIs can be executed in the same context, provided that their execution is serialized. An implicit *causal order* is established between local reads and writes and RMIs and between RMIs themselves that forms the STAPL-RTS RMI ordering guarantees.

An SPMD function `f` executes on a location l in context C_l. While the location is executing `f`, it encounters an RMI, RMI_1 (e.g., an `async_rmi`). The execution of RMI_1 logically happens after any local reads and writes in `f` that preceded the invocation of RMI_1.

If during the execution of `f` two RMIs, RMI_2 and RMI_3, are invoked to the same

[‡]Section 3.1 describes the STAPL-RTS execution model.

destination location l', then RMI_2 and RMI_3 will be executed in the invocation order in the same context $C_{RMI_{l'}}$ irrespectively of the target `p_object`.

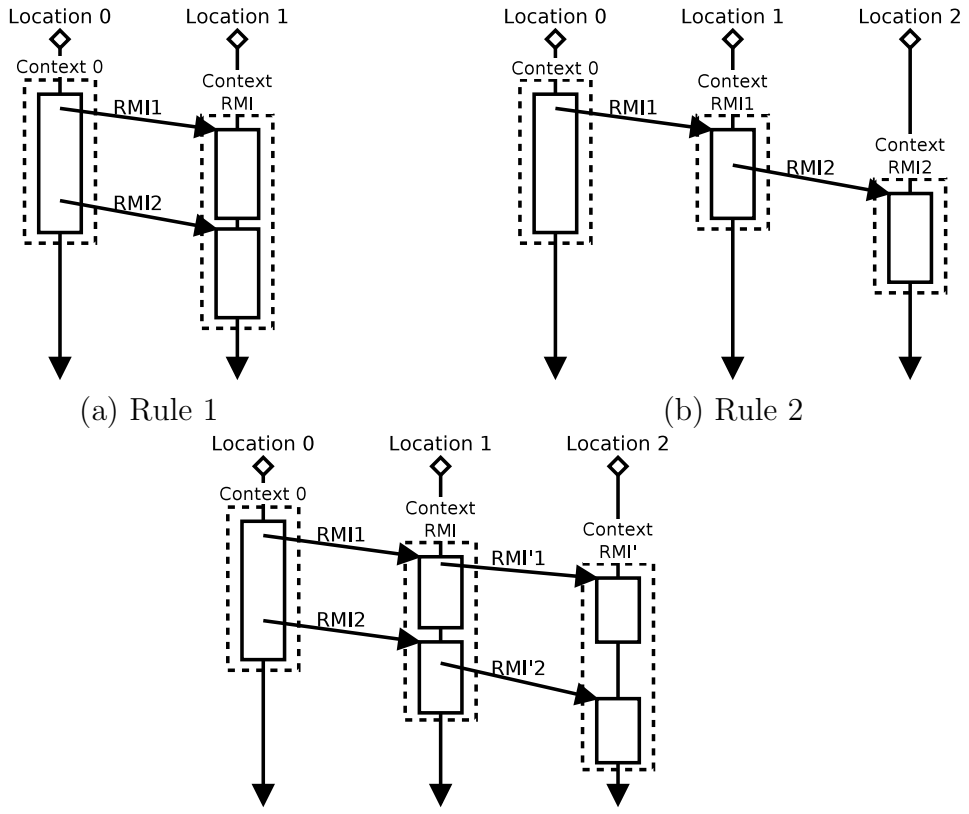
An RMI RMI_4 executes on a location in its own context C_{RMI_4} and has access to that location's `p_object` representatives. If during the execution of RMI_4 another RMI, RMI_5, is invoked, then the execution of the latter happens after the reads and the writes of RMI_4 that preceded the invocation of RMI_5. Additionally, if RMI_4 also invokes an RMI RMI_6 to the same destination location l'' as RMI_5, RMI_5 and RMI_6 will be executed in their invocation order in the same context $C_{RMI_{l''}}$.

In [70] the *"happened-before"* relation ($\rightarrow$) is established between events in a system of communicating processes: i) if a and b are events in the same process, and a precedes b, then $a \rightarrow b$, ii) if a is the sending of a message from one process to another process and b is the receipt of that message, then $a \rightarrow b$ and iii) if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$. Two events a and b are *concurrent* if $a \not\rightarrow b$ and $b \not\rightarrow a$; it is also noted that $a \not\rightarrow a$ for any reasonable system.

We extend and adapt the definition of $\rightarrow$ to suit the dynamic nature of asynchronous RMIs; RMIs are invoked from a source location to a destination location, without requiring the destination location to explicitly post receive requests for the RMIs.

Definition 6. Causal RMI Order *(CRMIO) is defined based on the "happened-before" relation ($\rightarrow$) between RMIs. RMI RMI_1 is said to have happened before an RMI RMI_2 ($RMI_1 \rightarrow RMI_2$) if*

1. *RMI_1 and RMI_2 are RMIs invoked from the same context to the same destination location and RMI_1 is invoked before RMI_2, or*
2. *RMI_2 was invoked from the context that RMI_1 was executing in, or*
3. *$RMI_1 \rightarrow RMI_i$ and $RMI_i \rightarrow RMI_2$ for some RMI RMI_i.*



(a) Rule 1

(b) Rule 2

(c) Rule 3

Figure 8.3: Causal RMI Ordering

We can show this definition using space-time diagrams, such as Figure 8.3. The vertical direction represents time with earlier times appearing higher, while the horizontal space. The solid line blocks are RMIs and SPMD functions, while the dotted line blocks denote a context the former execute in. The lines are the invocation of RMIs from a context to a location on a target `p_object` in the destination location.

In Figure 8.3(a), the RMIs are ordered because of rule 1 in Definition 6; $RMI_1 \rightarrow RMI_2$ since both RMI_1 and RMI_2 are invoked from the same context ($Context_0$). The second rule is visualized in Figure 8.3(b). $RMI_1 \rightarrow RMI_2$ since RMI_2 is invoked while executing RMI_1. Finally, the third rule is shown in Figure 8.3(c). The first rule dictates that $RMI_1 \rightarrow RMI_2$ and $RMI'_1 \rightarrow RMI'_2$, whereas the

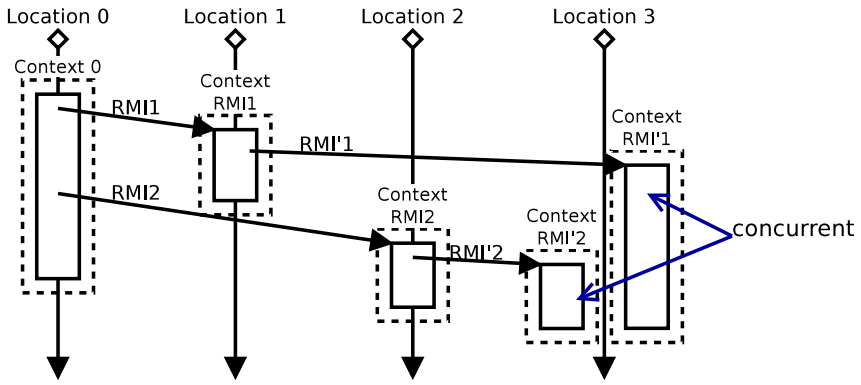


Figure 8.4: Concurrent RMIs

second rule guarantees $RMI_1 \rightarrow RMI_1'$ and $RMI_2 \rightarrow RMI_2'$. The transitivity property introduced by the third rule guarantees that $RMI_1 \rightarrow RMI_2'$.

On the other hand, Figure 8.4 shows an instance where ordering is not guaranteed for RMIs $RMI_{1'}$ and $RMI_{2'}$. The first rule is violated, as RMI_1 and RMI_2 have different destination locations and thus $RMI_1 \not\rightarrow RMI_2$.

8.4 Implementing Causal RMI Ordering

8.4.1 Atomic Execution of RMIs

We can rely on the C++ memory model [118] that if data races do not exist, the execution of a C++ program is sequentially consistent. Under this assumption, as long as a location is prevented from accessing the memory of another location directly, then RMIs can execute atomically.

Indeed, since the only way for locations to communicate is through RMIs on `p_objects`, then it is guaranteed that unless there is a call to the STAPL-RTS, an RMI will execute atomically.

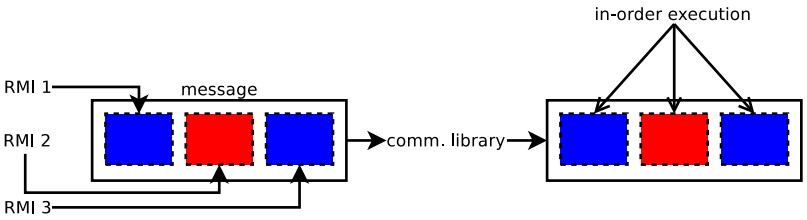


Figure 8.5: In-order RMI processing

8.4.2 Sequential and In-order Execution of RMIs

Distributed systems that guarantee causal message order often rely on mechanisms such as vector clocks [70] or timestamp mechanisms, such as MARS [133] and Δ-protocols [134]. All these algorithms work under the assumption that messages between processes can arrive out-of-order. In the HPC domain, communication libraries such as MPI and GASNet guarantee that messages will be delivered in the issuing order. As long as FIFO order is enforced for queued RMIs, then they will execute in the order they were received.

Our implementation of Causal RMI Order (CRMIO) is based around RMIs and their association with contexts. RMIs are aggregated in messages, or buffers, as shown previously in [60] and in Appendix B.1. Messages are transmitted in distributed memory through some communication library (e.g., MPI, GASNet) that guarantees that messages between two processes are received in the issuing order. Our mailbox data structure (see Section 6.3.2) provides the same guarantees in shared memory.

Upon receipt of a message with aggregated RMIs, it is placed in a FIFO queue that is associated with the context it will execute in. Upon processing of each message, the RMIs aggregated in it are executed in the stored order and in the context the queue is associated with. Therefore, the execution of RMIs that execute in the same

context is serialized and is in the issuing order, as shown in Figure 8.5. This satisfies the first rule in Definition 6.

In order to guarantee that RMIs are properly executed in order, it is enough to guarantee that they are associated with the same context and thus queued in the same queue. We define the *context ID* as a tuple of location IDs that the predecessor RMIs that lead to received RMI have executed on. The first RMI in the tuple always comes from an SPMD section and as such, the first element in the tuple is called the *originator* location ID. When an RMI is received, a new context ID is created by suffixing the context ID of the sending context with the location ID of the destination location. Therefore, the context ID is a history of the *invocation chain* of locations that resulted in the currently received RMI.

For RMIs that invoke other nested RMIs, if the context IDs of two contexts C and C', created by RMIs RMI and RMI' respectively, are denoted with the tuples T_C and $T_{C'}$ and $l_i, i \in \mathbb{N}$ is a location ID then

$$C \to C' \Leftrightarrow T_C = (l_i, l_j, \ldots, l_n), T_C = (l_i, l_j, \ldots, l_n, \ldots)$$

which satisfies the second rule in Definition 6.

As an example of nested RMIs, in Figure 8.6, the chain when function `A::i` is called is $(0, 2, 1, 4)$; `stapl_main` calls `A::f` on location 0, `A::f` calls `A::g` on location 2, `A::g` calls `A::h` on location 1 which finally calls `A::i` and `A::j` on location 4. This chain is the *context ID* and the location that started the chain, in this example location 0, the *originator*. `A::i` and `A::j` will execute in issuing order as their context IDs are the same. Additionally, `A::f` $\to$ `A::g` $\to$ `A::h` $\to$ `A::i`, which follows the invocation order of the respective functions.

Based on our experience, it is relatively uncommon to have nested RMIs that

```
1  struct A : public stapl::p_object {
2    void f() {
3      // context ID = (0)
4      stapl::async_rmi(2, this->get_rmi_handle(), &A::g);
5    }
6
7    void g() {
8      // context ID = (0, 2)
9      stapl::async_rmi(1, this->get_rmi_handle(), &A::h);
10   }
11
12   void h() {
13     // context ID = (0, 2, 1)
14     stapl::async_rmi(4, this->get_rmi_handle(), &A::i);
15     stapl::async_rmi(4, this->get_rmi_handle(), &A::j);
16   }
17
18
19   void i() {
20     // context ID = (0, 2, 1, 4)
21     // executes before j() on location 4
22   }
23
24   void j() {
25     // context ID = (0, 2, 1, 4)
26     // executes after i() on location 4
27   }
28 };
29
30 stapl_main(...) {
31   A a;
32
33   if (a.get_location_id()==0) {
34     // context ID = (0)
35     a.f();
36   }
37 }
```

Figure 8.6: Nested RMIs

have a context ID longer than 6 elements, while context IDs with 2 levels are the most prevalent. RMIs are used to put and get data, update metadata and flow values in task dependence graphs. In order to minimize latency, most algorithms on top of the STAPL-RTS are developed to keep nested invocation of RMIs to a minimum. As such, the more deeply nested an RMI is, the less the probability that it will invoke a new RMI.

Maintaining the context ID as a full tuple of all the visited locations can create performance issues. For each nested RMI the context ID would keep increasing by one location ID, requiring $\Theta(n)$ space, with n the number of locations in the tuple. The header size of a message that stores the context ID increases as the nesting level of the RMI increases. This in return makes RMI queuing more complex, as the context ID has to be parsed dynamically.

We therefore choose to compress context IDs so that we can have fast RMI scheduling for simple cases of 1 or 2 levels of RMIs and a more complex algorithm for all other cases. To bound the context ID to $\Theta(1)$ space, given a tuple $T = (l_i, \ldots, l_m, l_n)$ we use the tuple

$$< o, s, d, a, n >$$

as a compressed form of T:

- o is the *originator location* that started the chain (l_i),
- s is the *source location* that was the last location that created a context in the chain (l_m),
- d is the *destination location* where S created the context on (l_n),
- a is the *arbiter*, an integral value to differentiate between two contexts that,

although they have the same o, s, d and n, are actually concurrent, something that we explain more in depth in later paragraphs, and

- n is the length of the chain, or the *nesting level* of T ($|T|$).

By replacing the full tuple of location IDs with information about three locations and the nesting level, we effectively compress the chain with a lossy algorithm. The scheme can be relaxed to allow more locations in the compressed form.

However, discarding information may create artificial causality between contexts that are concurrent. For example, two RMIs that execute in contexts with chains $T' = (l_0, l_1, l_2, l_3)$ and $T'' = (l_0, l_4, l_2, l_3)$ will have the same compressed context ID at l_3, which would be $< l_0, l_2, l_3, 3 >$. RMIs in contexts that have the same ID will be processed in the order they arrive. In this case, RMIs that are concurrent will be incorrectly identified as causally related.

The *arbiter* is an integral value that breaks unintentional causality between contexts due to the compression of the invocation chain. It is an automatically generated integral value that captures the information that although two different context IDs have the same o, s, d, and n, they are in fact concurrent.

The arbiter is generated at the sender prior to sending the RMI when it is detected that the compressed context ID may lead to concurrent RMIs declared as causally related. When a new RMI RMI' is invoked from a context C, a new context ID is created that will identify the context C' that the RMI' will execute in. The context ID of C' is derived from the context ID of C; if the context ID of C is $< o, s, d, a, n >$ then the created context ID will be $< o, d, d', a', n + 1 >$, where o is the originator, d is the location C executes in and the source of C', d' is the location C' executes in and n is the nesting level. a and a' are the arbiter numbers and their generation is explained below:

- Each location has a hash table H that uses compressed context IDs as keys and hash tables h as the stored values.
- Each inner hash table h_i uses context IDs as keys and stores arbiter numbers.
- If the nesting level n of C is $n > 2$, then it is assumed that an artificial relationship will be imposed between C' and some other future context C'' that are both generated from C, as in the example with the tuples $T' = (l_0, l_1, l_2, l_3)$ and $T'' = (l_0, l_4, l_2, l_3)$ above.
- While creating the context ID of C', H is queried with a proposed context ID of C', which includes all the information of a compressed ID as discussed above minus the arbiter number. This requires to access entry $h_{C'}$.
- If entry $h_{C'}$ does not exist, then the arbiter number is chosen as 0, and $h_{C'}$ is initialized and added to H. Additionally, the arbiter number is assigned to the entry for the current context C using $h_{C'}[C] = 0$.
- If entry $h_{C'}$ exists, then $h_{C'}[C]$ is retrieved. If it exists, then number that is returned is used as the arbiter. If $h_{C'}[C]$ does not exist, a new entry is created and it is initialized with the size of $h_{C'}$, $h_{C'}[C] = h_{C'}.size()$. The newly created $h_{C'}[C]$ is the arbiter.

Since entries are never removed, the size of the inner hash tables are monotonically increasing and therefore generate a unique number for each pair of created and current context IDs. Using the example from previous paragraphs, in l_2 unique arbiter numbers a' and a'' are generated for both T' and T'', as the proposed context IDs would be the same for both ($< l_0, l_2, l_3, 3 >$) but the contexts that create them are not (T' is created by context $< l_0, l_1, l_2, 2 >$, whereas T'' is created by context $< l_0, l_4, l_2, 2 >$).

The trade-off using this technique is that while we maintain constant space complexity for communicating the context ID, we introduce $\Theta(km)$ space requirements per location, where k are the contexts of the location with nesting level $n > 2$ that invoke RMIs[§] and m are the created context IDs the RMIs are going to execute in.

8.5 Differences over Previous Work

ARMI [60] always relied on guarantees from the communication layer (e.g., MPI) to provide RMI ordering. However, as noted in [60], the user was responsible to write her programs in such a way that deadlock would not be possible. With the introduction of automatic work and data migration in STAPL [62], avoiding deadlocks became a more daunting task.

Additionally, the earlier implementation of ARMI did not allow arbitrary scheduling of RMIs that are concurrent. RMIs had to be executed in the order they were received from the communication layer, potentially causing starvation if a location was sending more RMIs than other locations.

8.5.1 Scheduling of RMIs

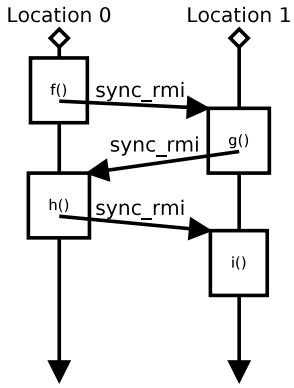
In this work, RMIs are queued in FIFO order based on the context they execute in. RMIs in the same queue are causally related and therefore cannot be arbitrarily scheduled. However, RMIs that are in different queues can follow any scheduling policy as they are concurrent.

The STAPL-RTS uses a round-robin scheduling scheme that executes 1 message from each queue every time that user code encounters a scheduling point (Section 3.1.3). If it is detected that either too many messages are queued (default value of 256) or are pending in the outgoing queue of the communication layer (default value of $2,048$) or a synchronization primitive was called (`rmi_fence`), then the

[§]Contexts that do not invoke RMIs do not require an entry.

```
1  struct A : public stapl::p_object {
2    int f() {
3      return stapl::sync_rmi(
4               1, this->get_rmi_handle(), &A::g);
5    }
6
7    int g() {
8      return stapl::sync_rmi(
9               0, this->get_rmi_handle(), &A::h);
10   }
11
12   int h() {
13     return stapl::sync_rmi(
14              1, this->get_rmi_handle(), &A::i);
15   }
16
17   int i() {
18     return 42;
19   }
20 };
21
22 stapl_main(...) {
23   A a;
24
25   if (a.get_location_id()==0)
26     a.f();
27 }
```

(a) Example code



(b) RMI call graph

Figure 8.7: Deadlock example

scheduling policy changes to execute all messages from one queue before moving to the next.

Choosing scheduling policies for RMIs is a new capability that opens a new domain for experimentation. This does not only include finding the optimal values for influencing the current scheduling policy, but even changing the scheduling based on the type of RMI received (e.g., process blocking RMIs first to minimize the effect on the waiting location).

8.5.2 Deadlock Avoidance

The more flexible scheduling capabilities also remove the possibility of a deadlock compared to prior work.

Consider the example in Figure 8.7. In each function a blocking `sync_rmi` call is

invoked to the other location. The chain of RMIs ends with the call to `A::i` and the value 42 will be flowed back to location 0 that started the chain.

Under [60], this code would lead to a deadlock. It was using the invocation order to create a single FIFO RMI queue, and since `A::g` was blocked waiting for a return value, `A::i` would never execute. Using CRMIO, `A::g` and `A::i`, while they are causally related (`A::f` → `A::g` → `A::h` → `A::i`), they are queued under different runqueues. The STAPL-RTS will properly execute `A::i`, return the value and allow it to flow back to location 0.

8.6 Causal RMI Ordering Use Cases

8.6.1 STAPL Container Consistency Model

The main STAPL component that depends on the CRMIO is the container framework. The *container Memory Consistency Model* (`pContainer` MCM) that the STAPL containers provide is described in [135].

In brief, the containers operate under an SPMD model as described in Section 5.1. Users declare a distributed container(i.e., `p_object`) over a set of locations and can invoke `Read` and `Write` operations on elements of a container. Additional operations are supported that are collective in the SPMD section (*Coll*), such as getting the size of the container. All operations are implemented as RMIs on `p_objects`. Successful completion of operations is guaranteed either by synchronization operations (*Synch*) or by implicit RMI ordering. Irrespectively of the data distribution or the number of locations, two writes to the same container element will be seen in the same order from any location in the system. This holds true even when elements are migrated.

Containers provide a more restrictive model to the user than the one that the STAPL-RTS supports. While internally they make full use of the RMI capabilities, externally they only allow reads, writes and container information operations on an

SPMD model. As such, all user invoked operations happen from a location context, rather than from within RMIs.

In order to show exactly how the CRMIO guarantees the `pContainer` MCM, we will annotate the latter with the guarantees provided by CRMIO. Normal text is the consistency model as outlined in [135], whereas bold text explains how the CRMIO makes the `pContainer` MCM possible:

The `pContainer` MCM: For an execution E, a container guarantees that there is a permutation P of all method invocations in E such that:

1. The methods in P occur sequentially (no overlapping). **The STAPL-RTS offers atomic and sequential execution of RMIs on a location. While concurrent RMIs may be executed in any order, they will not execute in parallel.**

2. For each element x, the restriction of P to just those methods on x, denoted $P|x$, satisfies the specification of the data type of x. (E.g., if x is a register that supports `Read` and `Write`, then each `Read` returns the value of the latest preceding `Write` invoked on x.) **Since x lives on one location and `Read` and `Write` functions are expressed using RMIs, CRMIO guarantees that RMIs are executed in invocation order, as long as the operations are invoked from the same location.**

3. For each thread i, the restriction of E to just the $Coll$ (collective) and $Synch$ (synchronization) methods invoked by i, denoted $E|(Coll \cup Synch)|i$, must equal $P|(Coll \cup Synch)|i$. That is, the permutation P has all the collective and synchronous methods by i in the same order as they were invoked. However, no guarantee is given as to how $Synch$ methods at different locations are ordered in P. **Ordering guarantees apply to both point-to-point**

and collective operations. CRMIO makes no ordering guarantees for *Synch* methods.

4. For each element x and each thread i, the restriction of P to the methods on x invoked by i, denoted $P|x|i$, consists of all the *Synch*, *Asynch*, and *Split Phase* methods on x invoked by i in E, in the order of their invocation. **CRMIO does not reorder operations from the same context.**

5. Consider any element x and let O_i and O_j be two operations on x in E such that O_i is invoked by some thread i, O_j is invoked by some other thread j, and O_i completes (i.e., receives its ACK) before O_j is invoked. Then O_i is ordered in P before O_j. **The STAPL-RTS does not need to guarantee anything additional for this.**

Other container components that require CRMIO is the *base container ordering* and *base container ranking*. Depending on the distribution policy and the partitioning information, each container, itself a `p_object`, has one or more base containers per location. These base containers are non-distributed objects that are the container's element storage. Each base container stores elements that have consecutive IDs. However, in order to support view operations, such as random access and the ability to create linear views over any container, the base containers themselves have to be ordered. The base container ordering operation builds a distributed double-connected linked list of all the base containers of a container that connects base containers that have consecutive elements. The base container ranking generates an increasing rank per base container that reflects its index in the linked list and is used to offer random access. Both operations are implemented directly using RMIs and rely on the CRMIO for correct execution.

8.6.2 Implementing Causal Consistency

In this section, we will implement Causal Consistency (CC) as presented in [123] to prove that CRMIO is applicable to use cases outside STAPL.

An algorithm that provides CC is outlined and proved correct in [123] and the reader can see it in Figure 8.8 in C++ pseudocode for some value type T. The main idea of the algorithm is the following:

- The machine consists of n reliable processes that communicate between them with messages through reliable communication channels that can reorder message delivery.

- Each process i has a private copy of the shared causal memory called M and a private copy of the vector clock t.

- A read from process i for an address x happens through the private copy of the shared memory M (`read`).

- A write from process i for an address x happens in the private copy of the shared memory M (`write`). This includes also updating its part of the vector clock ($t[i]$) and writing information about the write, such as which process, the address, the value and the private copy of the vector clock, called a write-tuple, to the *OutQueue*.

- The pair of functions `send_writes` and `receive_writes` are used to communicate updates to the shared memory through broadcasting the updates from each process. `send_writes` sends the *OutQueue* of a process to all other processes. `receive_writes` receives the *OutQueue* from a process and queues it in the process private copy of *InQueue*, a priority queue that orders elements based on the vector clock copy (timestamp), the fourth type in the write-tuple.

```
n = ... // number of processes
i = ... // process id

// Initialization
M = ... // private copy of shared causal memory
for (auto& x : M)
  x = {};
std::vector<int> t(n, 0); // vector clock
std::deque<std::tuple<process_id, std::size_t, T, std::vector<int>>> OutQueue = {};
std::priority_queue<std::tuple<process_id, std::size_t, T, std::vector<int>>>
    InQueue = {};

// Read from x
T read(std::size_t x) {
  return M[x];
}

// Write v to x from process i
void write(std::size_t x, T v) {
  t[i] = t[i] + 1;
  M[x] = v;
  OutQueue.emplace_back(i, x, v, t);
}

// Send action: execute infinitely often
void send_writes() {
  if (!OutQueue.empty()) {
    auto A = std::move(OutQueue);
    send(all_processes, A);
  }
}

// Receive action: upon receipt of A from process i
void receive_writes(A) {
  for (auto const& e : A)
    InQueue.push(e);
}

// Apply action: executed infinitely often
void update_private_memory() {
  if (!InQueue.empty()) {
    process_id j;
    std::size_t x;
    T v;
    std::vector<int> s;
    std::tie(j, x, v, s) = InQueue.front();
    if (s[j] = t[j] + 1) {
      for (std::size_t k = 0; k < n; ++k) {
        if (k!=j && s[k] > t[k])
          return;
      }
      InQueue.pop();
      t[j] = s[j];
      M[x] = v;
    }
  }
}
```

Figure 8.8: Causal consistency memory implementation

- The function `update_private_memory` updates the private copies of M and t on process i. The update happens only if the write-tuple on the head of the queue reflects no other write that process i is not aware of.

In STAPL-RTS, CC can be implemented more easily. Reliable processes (locations) and communication channels (RMIs) are provided by the STAPL-RTS and the underlying communication libraries (e.g., MPI). All RMIs coming from a context are processed in-order, something that removes the need for tracking which write happened when, as this is tracked by the STAPL-RTS. Each location has a private piece of the shared causal memory (representative of a `p_object`). CRMIO guarantees that if two operations are causally related they will be seen as such; reads/writes from the same context are ordered and a "happened-before" relation is established between them and between an RMI and its invoked RMI, as described earlier.

In Figure 8.9 we have implemented a shared causal memory on top for the STAPL-RTS. We did not strive for an optimal implementation, but rather a proof-of-concept that shows that the STAPL-RTS is applicable for use in other cases apart from STAPL.

Each location i has a copy of the shared memory. A `causal_memory::read` calls an RMI to return a value from the local copy[¶]. A `causal_memory::write` from location i sends out an update to all locations, including i.

Since the reads and writes happen through RMIs, the reads and writes are ordered per invocation order on the location. Writes from different locations are seen in the order they were performed. Finally, if a location reads a value, it can assume that all the operations that preceded the write of that value have finished, as per CRMIO. This satisfies all the requirements of Causal Consistency.

[¶]A location invoking an RMI to itself still follows CRMIO.

```
1  using namespace stapl;
2
3  template<typename T>
4  class causal_memory
5  : public p_object
6  {
7  private:
8    // private copy of shared memory
9    ... M;
10
11   T read_impl(std::size_t x) const {
12     return M[x];
13   }
14
15   void write_impl(std::size_t x, T v) {
16     M[x] = v;
17   }
18
19 public:
20   future<T> read(std::size_t x) const {
21     return opaque_rmi(this->get_location_id(), this->get_rmi_handle(),
       &A::read_impl, x);
22   }
23
24   void write(std::size_t x, T v) {
25     async_rmi(all_locations, this->get_rmi_handle(),  &A::write_impl, x, v);
26   }
27 };
```

Figure 8.9: STAPL-RTS-based causal memory implementation

8.7 Unordered Primitives

In some cases, components do not require the default RMI ordering guarantees that the STAPL-RTS offers. Examples of these cases are finding the size of a container via the one-sided collective `reduce_rmi` and propagating values to be consumed in the `PARAGRAPH` via `async_rmi` to multiple locations. For these cases we offer implementations of popular one-sided collective primitives, such as `reduce_rmi` and `async_rmi` in the `unordered` namespace that override the CRMIO.

An example of the difference between ordered and unordered RMI requests can be found in Figure 8.10(a) and Figure 8.10(b) respectively. In Figure 8.10(a) the `async_rmi(all_locations)` and the `opaque_rmi` respect the implicit ordering by CRMIO. For the `unordered::async_rmi(all_locations)` (Figure 8.10(b)), com-

```
struct A : public stapl::p_object {
  int m_value;

  A() : m_value(0) { }
  void write(int t) { m_value = t; }
  int read() const { return m_value; }
};

stapl_main(...) {
  A a;

  if (stapl::get_location_id()==0) {
    // set the int stored in p_object a on all locations to 42
    stapl::async_rmi(stapl::all_locations, a.get_rmi_handle(), &A::write, 42);

    // read the value from location 1
    auto r = stapl::opaque_rmi(1, a.get_rmi_handle(), &A::read);

    assert( r.get()==42 );
  }
}
```

(a) Ordered RMI request

```
stapl_main(...) {
  A a;

  if (stapl::get_location_id()==0) {
    // set the int stored in p_object a on all locations to 42
    stapl::unordered::async_rmi(stapl::all_locations, a.get_rmi_handle(),
    &A::write, 42);

    // read the value from location 1
    auto r = stapl::opaque_rmi(1, a.get_rmi_handle(), &A::read);

    // assert( r.get()==42 ); not guaranteed, unordered::async_rmi may
    // have not executed
  }

  stapl::rmi_fence();

  if (stapl::get_location_id()==0) {
    // read the value from location 1
    auto r = stapl::opaque_rmi(1, a.get_rmi_handle(), &A::read);

    assert( r.get()==42 ); // guaranteed, unordered::async_rmi has executed
                           // due to rmi_fence
  }
}
```

(b) Unordered RMI request

Figure 8.10: Ordered and unordered RMI requests

pletion can only be guaranteed through explicit synchronization, in this case the `rmi_fence` call, which ensures that all previously invoked RMI calls have finished.

Unordered RMIs offer the same execution guarantees as the ordered; an RMI executes sequentially, and atomically unless an STAPL-RTS primitive is invoked. However, they relax the ordering guarantees for performance. For example, unordered one-sided collective RMIs can use simpler multicast algorithms and previous and subsequent RMIs do not have to wait for the unordered RMI to execute.

8.8 Application Driven Ordering Relaxation

As an example of application driven runtime optimization, we tune the aggregation of RMIs, an optimization that has been shown to be important for fine-grained asynchronous messaging models [51, 60, 136]. We create ad-hoc communication channels to efficiently aggregate sequences of RMIs sharing common and constant parameters such as destination and target method. As we will show, the technique can allow relaxed ordering for collections of requests that are logically associated with a given computational activity. This technique can have a dramatic effect on application performance, as demonstrated using a common graph traversal algorithm.

As is usually the case with asynchronous communication models, STAPL encourages fine grain communication. Previous work [51, 60, 136] has shown aggregating these requests generally leads to overall better performance. In STAPL-RTS, we aggregate multiple RMI requests to the same destination location in the same outgoing buffer. We further enhance this mechanism by implementing *request combining*, a lightweight compression technique for requests that have the same triplet of target `p_object`, function and destination as the previous requests in the buffer. If this triplet is the same, then we need only append the arguments of the request to the aggregation buffer. This is explained more in depth in Appendix B.1.

```
for (int idx = ...)
  stapl::async_rmi(dest, handle, A::set_element, idx, rand());
```

(a) Default request aggregation

```
auto tunnel = stapl::bind(stapl::async_rmi, dest, handle, A::set_element, _1, _2);
for (int idx = ...)
  tunnel(idx, rand());
```

(b) Partial function evaluation of `async_rmi`

Figure 8.11: Application customized aggregation in STAPL-RTS

Consider the code in Figure 8.11(a) which updates a sequence of values in a remote object with random values. If the STAPL-RTS were to generate an MPI request for every `async_rmi` invocation in this tight loop, performance would suffer as the overhead of request transmission would greatly outweigh the cost of the requested updates. In this case, however, the STAPL-RTS is free to employ not only basic aggregation but combining as well (see AppendixB.1), without violating the CRMIO.

Note however in this case there is information trivially available to the user that would aid the runtime in this activity. The fact that the object handle, destination location and target method remain constant is immediately clear in the calling context. Using *partial function evaluation*, a common generic programming operation, we can fix one or more arguments of the STAPL-RTS primitives such as `async_rmi`, creating a new function with reduced arity. This new function contains typing information about which parameters have been fixed. To accomplish this, we can use a `bind` function with an interface similar to that of the C++ STL. The operation creates a custom communication channel as shown in Figure 8.11(b) based solely on algorithm level information. Using this RMI *tunnel* has the following effects:

- **Relaxed request ordering**. Tunnels define a new logical route to destination location with ordering guarantees independent of the default route (the atom-

icity of all RMIs is maintained). A tunnel defined with only a bound location maintains the same basic causal ordering as previously described.

- **Less runtime overhead and more efficient aggregation.** By binding additional arguments during the partial evaluation, we reduce redundancy in the message; only a single copy of the bound parameter is stored in the aggregated message instead of a copy for every RMI. Other optimizations are enabled by different combinations of bound parameters. For example binding both the object handle and target function enables combining at compile time, eliminating the overhead of runtime detection.

Though not necessary for the example shown in Figure 8.11(a), dedicated per method tunnels enable greater use of combining in some instances. Consider the case where a small number of non-combinable RMIs are interspersed in an otherwise homogeneous sequence of RMI invocations. By creating a tunnel for the homogeneous requests, the other requests do not interfere with the combining operation.

One use of tunnels in STAPL is in graph traversals, where a vertex *visitor* function passed to the algorithm is repeatedly applied on vertices throughout the `pGraph` data structure. When graph edges cross location boundaries, RMIs are issued to complete the visitation. The algorithm specifies a set of tunnels for these fine grain method invocations through this interface. As we show below, this high level annotation can have a dramatic effect on performance and scalability.

We evaluate this technique using a parallel connected components (CC) graph algorithm. The algorithm computes the connected components – i.e., the subgraph wherein any two vertices in the subgraph can be connected through some path – for each vertex, and the ID representing the component is assigned to the vertex. It is a label-propagation algorithm similar to the work presented in [137], wherein nodes

```
for (auto&& u : neighbors(v))
  stapl::async_rmi(location_of(u), handle, Graph::visit<cc_visitor>,
    cc_visitor(v.id()), u);
```

(a) Default request aggregation

```
auto tunnel = stapl::bind(stapl::async_rmi, _1, handle, Graph::visit<cc_visitor>,
    _2, _3);
for (auto&& u : neighbors(v))
  tunnel(location_of(u), cc_visitor(v.id()), u);
```

(b) Partial evaluation of `async_rmi`

Figure 8.12: Customized request aggregation for connected components

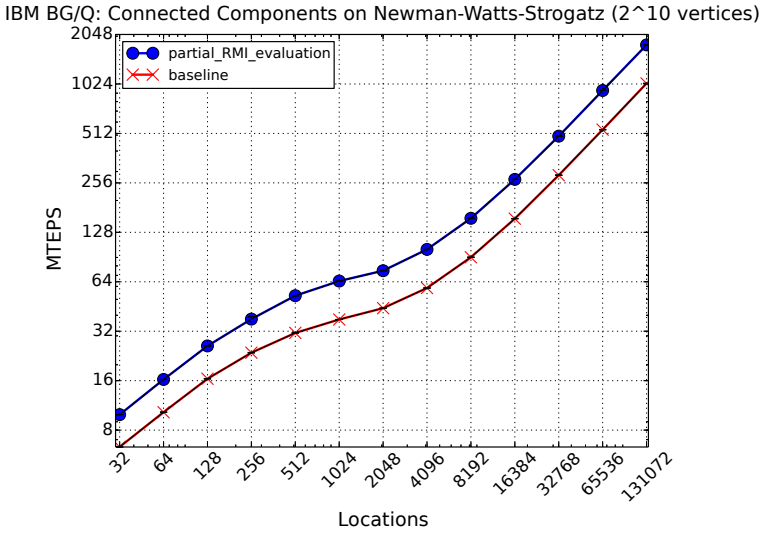


Figure 8.13: Connected components on IBM-BG/Q

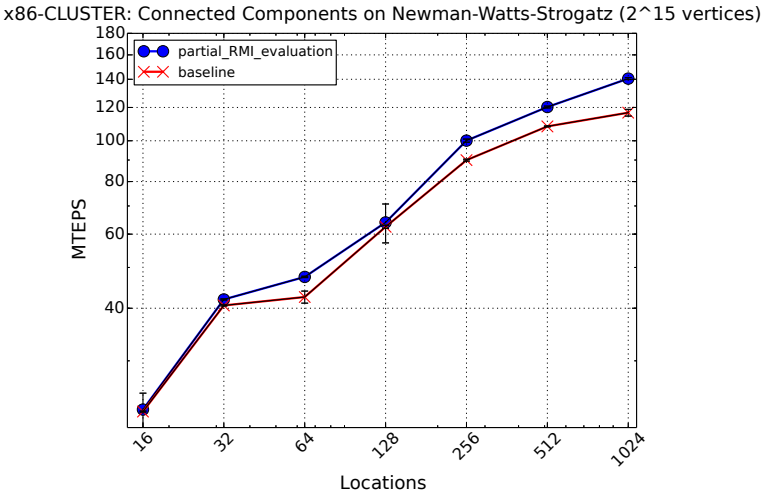


Figure 8.14: Connected components on x86-CLUSTER

set their CC ID to the lowest CC ID of their neighbors iteratively for k rounds. The CC algorithm is widely used to study the connectivity and basic topology of graphs.

In the standard expression of the algorithm, each vertex in parallel visits all other vertices in its neighborhood to propagate its ID as a candidate CC ID. In Figure 8.12(a), we achieve this by issuing `async_rmi`s to the locations of all neighbor vertices to apply the visitation function computing the CC. At the algorithm's level, both the handle of the graph data-structure and the method to apply visitor functions is constant, so we can form a tunnel to aid combining based aggregation of these requests (Figure 8.12(b)).

Figure 8.13 evaluates the algorithm's performance in terms of throughput (millions of traversed edges per second or MTEPS), both with and without tunneling on up to $131,072$ processors on IBM-BG/Q for a Newman-Watts-Strogatz graph of 2^{10} vertices per core. We see a $1.5\times$ improvement in throughput at lower core counts, which grows to $1.7\times$, suggesting that tunneling is not only increasing throughput but also improving scalability. Figure 8.14 shows the performance for the same type of graph with 2^{15} vertices per core on X86-CLUSTER with smaller but still noticeable improvement of about 20% over the approach without tunneling.

9. CONCLUSION AND FUTURE WORK

The past few years increases in single thread performance have been marginal. As needs for computational power increase, multicore and multiprocessor approaches often coupled with accelerators become more relevant in mainstream computing. Additionally, the scale up approach, adding more computing resources to a single computer, will eventually hit an upper bound, forcing to adopt more scale out solutions.

Programming these complex architectures has been proven difficult. Traditionally, to achieve the best performance, experts write highly tuned but non-portable applications. On the other side of the spectrum, non-experts rely on higher level programming paradigms that trade easier programming, portability and expressivity for some performance loss.

The SmartApps approach puts productivity and the application in the center of attention. The programmer is responsible for expressing her application using high level constructs. The application is responsible for establishing a top-to-bottom information flow, information which every software layer can mine to perform optimizations. In this dissertation we focus on the STAPL Runtime System (stapl-rts), a runtime system built to support higher level programming frameworks. The stapl-rts abstracts massively parallel platforms and provides platform-independent interfaces that allow to restore performance lost due to the abstractions, building the foundation for the SmartApps vision.

We began our discussion by establishing what is a runtime system and what are some desired qualities for a runtime that supports scale out systems. We then presented the programming model the stapl-rts supports and its main components, focusing on the asynchronous communication model it offers. In order to give a

more concrete example of how the STAPL-RTS can be used, we presented the STAPL framework and how it takes advantage of the STAPL-RTS.

We continued with our mixed-mode support, showing how abstracting shared and distributed memory communication under a unified communication model relieves the programmer from the nuances of hybrid solutions. We expanded on that and offered high-level interfaces that allow programmers to abstractly express data consumption, information that the STAPL-RTS uses to take advantage of shared memory transparently and increase performance, both in isolation and in STAPL.

Turning our focus back to the programming model, we delved into the support for nested parallelism. We presented a model that combines the traditional blocking, collective subgroup support with a novel asynchronous, one-sided subgroup creation and how it is used to support container composition and nested parallelism. Using a natural, nested parallel expression for dynamic algorithms we proved that using asynchronous, one-sided nested parallelism can lead to significant performance improvements in graph applications.

We have also presented the ordering guarantees for RMIs, Causal RMI Ordering, that provides an intuitive way to reason about communication request execution. Using appropriate abstractions, we showed how an application can relax this ordering to enhance performance.

This work is only the start for the STAPL-RTS. Its modular design has allowed porting it from smartphones to the largest supercomputers. However, we have only scratched the surface of the potential application-driven optimizations. While we have abstracted and maintained the machine hierarchy information, we have not fully taken advantage of the shared memory node hierarchy or the machine topology. Introducing hierarchy awareness to the STAPL-RTS will allow us to fine-tune shared memory optimizations. Taking into account topology information can not only lead

to more efficient collective operations but also allow us to control common operations, such as the dissemination of data, in such a way that we fully utilize the machine's network capabilities.

Our asynchronous, one-sided nested parallelism support would be an interesting approach to harness the power of accelerators. The nested parallel programming model that the STAPL-RTS offers can be trivially expanded to allow the creation of parallel sections on accelerators, allowing higher level frameworks, such as STAPL, to take advantage of accelerators with minimal changes to the frameworks themselves.

Finally, another research dimension would be to provide different guarantees along the causal RMI ordering, such as object consistency. Object consistency can couple the existing ordering guarantees on a `p_object` basis, providing more opportunities for communication and computation overlap, and quiescence segregation.

REFERENCES

[1] Lawrence Rauchwerger, Nancy M. Amato, and Josep Torrellas. SmartApps: An Application Centric Approach to High Performance Computing. In *Int. Workshop on Languages and Compilers for Parallel Computing (LCPC), in Lecture Notes in Computer Science (LNCS)*, volume 2017, pages 82–96. Springer-Verlag, 2001. LCPC 2000.

[2] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Timmie Smith, Gabriel Tanase, Nathan Thomas, Nancy M. Amato, and Lawrence Rauchwerger. STAPL: A Standard Template Adaptive Parallel C++ Library. In *Proc. of the International Workshop on Advanced Compiler Technology for High Performance and Embedded Processors (IWACT)*, Bucharest, Romania, Jul 2001.

[3] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Timmie Smith, Gabriel Tanase, Nathan Thomas, Nancy M. Amato, and Lawrence Rauchwerger. STAPL: An adaptive, generic parallel programming library for C++. In *Int. Workshop on Languages and Compilers for Parallel Computing (LCPC), in Lecture Notes in Computer Science (LNCS)*, volume 2624, Cumberland Falls, KY, USA, Aug 2001.

[4] Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M. Amato, and Lawrence Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, pages 277–288, Chicago, IL, USA, 2005. ACM.

[5] Gabriel Tanase, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. The STAPL pArray. In *Proceedings of the 2007 Workshop on Memory Perfor-*

mance (MEDEA), pages 73–80, Brasov, Romania, 2007.

[6] Gabriel Tanase, Chidambareswaran Raman, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. Associative parallel containers in STAPL. In *Int. Workshop on Languages and Compilers for Parallel Computing (LCPC), in Lecture Notes in Computer Science (LNCS)*, volume 5234, pages 156–171, Urbana-Champaign, IL, USA, 2008.

[7] Antal A. Buss, Timmie Smith, Gabriel Tanase, Nathan Thomas, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. Design for interoperability in STAPL: pMatrices and linear algebra algorithms. In *Int. Workshop on Languages and Compilers for Parallel Computing (LCPC), in Lecture Notes in Computer Science (LNCS)*, volume 5335, pages 304–315, Edmonton, Alberta, Canada, July 2008.

[8] Gabriel Tanase, Xiabing Xu, Antal A. Buss, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie G. Smith, Nathan Thomas, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. The STAPL plist. In *Languages and Compilers for Parallel Computing, 22nd International Workshop, LCPC 2009, Newark, DE, USA, October 8-10, 2009, Revised Selected Papers*, pages 16–30, 2009.

[9] Antal A. Buss, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie G. Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. STAPL: standard template adaptive parallel library. In *Proceedings of of SYSTOR 2010: The 3rd Annual Haifa Experimental Systems Conference, Haifa, Israel, May 24-26, 2010*, pages 1–10, New York, NY, USA, 2010. ACM.

[10] Antal A. Buss, Adam Fidel, Harshvardhan, Timmie G. Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. The STAPL pview. In *Languages and Compilers for Parallel Computing - 23rd International Workshop, LCPC 2010, Houston, TX, USA, October 7-9, 2010. Revised Selected Papers*, pages 261–275, 2010.

[11] Gabriel Tanase, Antal A. Buss, Adam Fidel, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie G. Smith, Nathan Thomas, Xiabing Xu, Nedal Mourad, Jeremy Vu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. The STAPL parallel container framework. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pages 235–246, 2011.

[12] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. The STAPL Parallel Graph Library. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 46–60. Springer Berlin Heidelberg, 2012.

[13] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. KLA: A new algorithmic paradigm for parallel graph computations. In *Proc. Intern. Conf. Parallel Architecture and Compilation Techniques (PACT)*, PACT '14, pages 27–38, New York, NY, USA, 2014. ACM. Conference Best Paper Award.

[14] Mani Zandifar, Nathan Thomas, Nancy M. Amato, and Lawrence Rauchwerger. The STAPL Skeleton Framework. In *Int. Workshop on Languages and Compilers for Parallel Computing (LCPC), in Lecture Notes in Computer Science (LNCS)*, Hillsboro, Oregon, USA, 2014.

[15] Ioannis Papadopoulos, Nathan Thomas, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. STAPL-RTS: An Application Driven Runtime System. In *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015*, pages 425–434, 2015.

[16] Mani Zandifar, Mustafa Abdul Jabbar, Alireza Majidi, David Keyes, Nancy M. Amato, and Lawrence Rauchwerger. Composing algorithmic skeletons to express high-performance scientific applications. In *Proceedings of the 29th ACM International Conference on Supercomputing (ICS)*, ICS '15, pages 415–424, New York, NY, USA, 2015. ACM. Conference Best Paper Award.

[17] David Musser, Gillmer Derge, and Atul Saini. *STL Tutorial and Reference Guide, Second Edition*. Addison-Wesley, 2001.

[18] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley Professional, 2013.

[19] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 21–40, New York, NY, USA, 2012. ACM.

[20] Andrew W. Appel. A runtime system. *Lisp and Symbolic Computation*, 3(4):343–380, 1990.

[21] Alan FT Winfield. *The complete FORTH*, volume 3. Wiley Press, 1983.

[22] Don Box and Ted Pattison. *Essential. Net: the common language runtime*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[23] Kenneth E. Batcher. Design of a massively parallel processor. *Computers, IEEE Transactions on*, 100(9):836–840, 1980.

[24] Michel Auguin and François Larbey. OPSILA: an advanced SIMD for numerical analysis and signal processing. In *Microcomputers: developments in industry, business, and education, Ninth EUROMICRO Symposium on Microprocessing and Microprogramming*, volume 16, pages 311–318, Madrid, Spain, September 1983.

[25] Jiannong Cao. Clustergop: A high-level parallel programming environment. In *Parallel Processing Workshops, 2004. ICPP 2004 Workshops. Proceedings. 2004 International Conference on*, pages 158–158, Aug 2004.

[26] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[27] Honghui Lu, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. Quantifying the performance differences between PVM and treadmarks. *J. Par. Dist. Comp.*, 43(2):65–78, 15 June 1997.

[28] UPC Consortium and others. UPC language specifications v1.2. *Lawrence Berkeley National Laboratory*, 2005. `http://escholarship.org/uc/item/7qv6w1rk`.

[29] Robert W. Numrich and John Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[30] Guy Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-93-129, Carnegie Mellon University, 1993.

[31] David A. Patterson. Latency lags bandwidth. In *ICCD*, pages 3–6, Oct 2005.

[32] David B. Loveman. High performance fortran. *IEEE Parallel and Distributed Technology*, 1:25–42, 1993.

[33] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The complete reference*. The MIT Press, Cambridge, MA, 1996.

[34] International Organization for Standardization (ISO). *IEC 14882: 2011 Information technology – Programming languages – C++*, volume 27. 2012.

[35] OpenMP, ARB. OpenMP Application Program Interface. Technical report, 2011.

[36] Melvin E. Conway. A multiprocessor system design. In *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*, AFIPS '63 (Fall), pages 139–146, New York, NY, USA, 1963. ACM.

[37] Alejandro Duran, Raúl Silvera, Julita Corbalán, and Jesús Labarta. Runtime adjustment of parallel nested loops. In *Shared Memory Parallel Programming with OpenMP*, pages 137–147. Springer, 2004.

[38] Christian Terboven, Dieter An Mey, Dirk Schmidl, and Marcus Wagner. First experiences with Intel Cluster OpenMP. In *OpenMP in a New Era of Parallelism*, pages 48–59. Springer, 2008.

[39] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[40] Charles E. Leiserson. The Cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, New York, NY, USA, 2009. ACM.

[41] Liang Peng, Weng-Fai Wong, Ming-Dong Feng, and Chung-Kwong Yuen. Silkroad: A multithreaded runtime system with software distributed shared memory for smp clusters. *Cluster Computing, IEEE International Conference on*, 0:243, 2000.

[42] James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[43] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: The New Adventures of Old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.

[44] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. MPI-2: Extending the message-passing interface. In *Euro-Par'96 Parallel Processing: Second International Euro-Par Conference Lyon, France, August 26–29 1996 Proceedings, Volume I*, pages 128–135. Springer Berlin Heidelberg, 1996.

[45] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 256–266, New York, NY, USA, 1992. ACM.

[46] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten Von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Supercomputing'93. Proceedings*, pages 262–273. IEEE, 1993.

[47] Jarek Nieplocha, Vinod Tipparaju, Manojkumar Krishnan, and Dhabaleswar K. Panda. High performance remote memory access communication: The ARMCI approach. *International Journal of High Performance Computing Applications*, 20(2):233–253, 2006.

[48] Jeff Daily, Abhinav Vishnu, Bruce Palmer, Hubertus van Dam, and Darren Kerbyson. On the suitability of mpi as a pgas runtime. In *High Performance Computing (HiPC), 2014 21st International Conference on*, pages 1–10. IEEE, 2014.

[49] Dan Bonachea. GASNet Specification, v1.1. Technical Report UCB/CSD-02-1207, University of California at Berkeley, 2002.

[50] Christopher G. Baker and Michael A. Heroux. Tpetra, and the use of generic programming in scientific computing. *Scientific Programming*, 20(2):115–128, 2012.

[51] Laxmikant V. Kalé and Sanjeev Krishnan. CHARM++: A portable concurrent object oriented system based on C++. *SIGPLAN Not.*, 28(10):91–108, 1993.

[52] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The Cascade High Productivity Language. In *The Ninth Int. Workshop on High-Level Parallel Programming Models and Supportive Environments*, volume 26, pages 52–60, Los Alamitos, CA, USA, 2004. IEEE.

[53] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[54] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. Parallex. *2012 41st International Conference on Parallel Processing Workshops*, 0:394–401, 2009.

[55] Yili Zheng, Amir Kamil, Michael B. Driscoll, Hongzhang Shan, and Katherine Yelick. UPC++: a PGAS Extension for C++. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1105–1114. IEEE, 2014.

[56] Paul N. Hilfinger, Dan Bonachea, David Gay, Susan Graham, Ben Liblit, Geoff Pike, and Katherine Yelick. Titanium language reference manual. *Science*, (UCB/EECS-2005-15), 2005.

[57] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. In ACM, editor, *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.

[58] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2):203–231, May 2006.

[59] Ian Foster, Carl Kesselman, and Steven Tuecke. The Nexus task-parallel runtime system. In *Proc. 1st Intl Workshop on Parallel Processing*, pages 457–462, 1994.

[60] Steven Saunders and Lawrence Rauchwerger. ARMI: an adaptive, platform independent communication library. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, pages 230–241, San Diego, California, USA, 2003. ACM.

[61] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. *SIGPLAN Not.*, 12(8):55–59, August 1977.

[62] Adam Fidel, Sam Ade Jacobs, Shishir Sharma, Nancy M. Amato, and Lawrence Rauchwerger. Using load balancing to scalably parallelize sampling-based motion planning algorithms. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, Phoenix, Arizona, USA, May 2014.

[63] Shirley Browne, Jack Dongarra, Nathan Garner, George Ho, and Philip Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.

[64] Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20:287–331, 2006.

[65] C. Eric Wu, Anthony Bolmarcich, Marc Snir, David Wootton, Farid Parpia, Anthony Chan, Ewing Lusk, and William Gropp. From trace generation to visualization: A performance framework for distributed parallel systems. In *Proc. of SC2000: High Performance Networking and Computing*, November 2000.

[66] Robert Demming and Daniel J. Duffy. *Introduction to the Boost C++ Libraries; Volume I-Foundations*. Datasim Education BV, 2010.

[67] Amitabh B. Sinha, Laxmikant V. Kalé, and Balkrishna Ramkumar. A dynamic and adaptive quiescence detection algorithm. Technical report, University of Illinois at Urbana-Champaign, Urbana-Champaign, 1993.

[68] Colin Campbell and Ade Miller. *A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures*. Microsoft Press, 1st edition, 2011.

[69] Niklas Gustafsson, Artur Laksberg, Herb Sutter, and Sana Mithani. Improvements to std::future<T> and Related APIs. *ISO/IEC JTC 1, Information Technology Subcommittee SC 22, Programming Language C++*, 2013.

[70] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[71] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[72] W. Daryl Hawkins, Timmie Smith, Michael P. Adams, Lawrence Rauchwerger, Nancy M. Amato, and Marvin L. Adams. Efficient massively parallel transport sweeps. *Transactions American Nuclear Society*, 107:477–481, 2012.

[73] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, volume 28, pages 1–12. ACM, 1993.

[74] Dhabaleswar K. Panda et al. OSU Microbenchmarks. `http://mvapich.cse.ohio-state.edu/benchmarks/`.

[75] Rajeev Thakur and William D. Gropp. Test suite for evaluating performance of multithreaded mpi communication. *Parallel Computing*, 35:608–617, 11/2008 2008.

[76] D. Bailey et al. The NAS parallel benchmarks. Technical Report NAS RNR-94-007, NASA Ames Research Center, http://www.nas.nasa.gov/npb/, 1994.

[77] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 2010.

[78] Nathan Thomas, Steven Saunders, Timmie G. Smith, Gabriel Tanase, and Lawrence Rauchwerger. ARMI: a High Level Communication Library for STAPL. *Parallel Processing Letters*, 16(2):261–280, 2006.

[79] Ioannis Papadopoulos, Nathan Thomas, Adam Fidel, Dielli Hoxha, Nancy M. Amato, and Lawrence Rauchwerger. Asynchronous nested parallelism for dynamic applications in distributed memory. In *Int. Workshop on Languages and Compilers for Parallel Computing (LCPC), in Lecture Notes in Computer Science (LNCS)*, Raleigh, NC, USA, September 2015.

[80] Fangzhou Jiao, Nilesh Mahajan, Jeremiah Willcock, Arun Chauhan, and Andrew Lumsdaine. Partial globalization of partitioned address spaces for zero-copy communication with shared memory. In *High Performance Computing (HiPC), 2011 18th International Conference on*, pages 1–10, Dec 2011.

[81] Igor Pechtchanski and Vivek Sarkar. Immutability specification and its applications: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(5-6):639–662, April 2005.

[82] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 35–49, Vancouver, BC, Canada, October 26–28, 2004.

[83] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kiezun, and Michael D. Ernst. Object and reference immutability using java generics. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 75–84, New York, NY, USA, 2007. ACM.

[84] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, February 1996.

[85] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. MPI + MPI: A New Hybrid Approach to Parallel Programming with MPI Plus Shared Memory. *Computing*, 95:1121–1136, 2013.

[86] Darius Buntinas and Guillaume Mercier. Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem. In *Proceedings of the International Symposium on Cluster Computing and the Grid*, pages 521–530. IEEE Computer Society, 2006.

[87] Sandia National Labs. Portals Message Passing Interface. `http://www.sandia.gov/Portals`.

[88] Franck Cappello and Daniel Etiemble. MPI Versus MPI+OpenMP on IBM SP for the NAS Benchmarks. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.

[89] Juan Sillero, Guillem Borrell, Javier Jiménez, and Robert D. Moser. Hybrid openMP-MPI Turbulent Boundary Layer Code over 32K Cores. In *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI'11, pages 218–227, Berlin, Heidelberg, 2011. Springer-Verlag.

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

[91] Saptarshi Chatterjee, Sagnak Tasırlar, Zoran Budimlic, Vincent Cave, Milind Chabbi, Max Grossman, Vivek Sarkar, and Yonghong Yan. Integrating Asynchronous Task Parallelism with MPI. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 712–725, May 2013.

[92] Thomas Heller, Hartmut Kaiser, Andreas Schäfer, and Dietmar Fey. Using HPX and LibGeoDecomp for Scaling HPC Applications on Heterogeneous Supercomputers. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '13, pages 1:1–1:8, New York, NY, USA, 2013. ACM.

[93] Sebastian Nanz, Sam West, Kaue Soares da Silveira, and Bertrand Meyer. Benchmarking usability and performance of multicore languages. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 183–192. IEEE, 2013.

[94] Dan Bonachea and Jason Duell. Problems with Using MPI 1.1 and 2.0 As Compilation Targets for Parallel Language Implementations. *Int. J. High Perform. Comput. Netw.*, 1(1-3):91–99, August 2004.

[95] Xindong Wu and Vipin Kumar. K-means. In *The Top Ten Algorithms in Data Mining*. CRC Press, Boca Raton, FL, USA, 2009.

[96] Inderjit S. Dhillon and Dharmendra S. Modha. A data-clustering algorithm on distributed memory multiprocessors. In *Large-Scale Parallel Data Mining*, volume 1759 of *LNAI*, pages 245–260. Springer-Verlag, 2002.

[97] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 180–186, Feb 2010.

[98] Amir Kamil and Katherine A. Yelick. Hierarchical Computation in the SPMD Programming Model. In Calin Cascaval and Pablo Montesinos, editors, *LCPC*, volume 8664 of *Lecture Notes in Computer Science*, pages 3–19. Springer, 2013.

[99] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[100] Jisheng Zhao, Roberto Lublinerman, Zoran Budimlić, Swarat Chaudhuri, and Vivek Sarkar. Isolation for Nested Task Parallelism. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 571–588, New York, NY, USA, 2013. ACM.

[101] Christoph W. Keßler. NestStep: Nested Parallelism and Virtual Shared Memory for the BSP Model. *The Journal of Supercomputing*, 17(3):245–262, 2000.

[102] John Mellor-Crummey, Laksono Adhianto, III William N. Scherer, and Guohua Jin. A new vision for coarray Fortran. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 5:1–5:9, New York, NY, USA, 2009. ACM.

[103] Timothy D. R. Hartley, Erik Saule, and ímit V. Çatalyürek. Improving Performance of Adaptive Component-based Dataflow Middleware. *Parallel Comput.*, 38(6-7):289–309, June 2012.

[104] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, volume 30, pages 207–216, New York, NY, USA, July 1995. ACM.

[105] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an Object-Oriented Approach to Non-Uniform Cluster Computing. In *Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[106] Guy L. Steele Jr., Eric E. Allen, David Chase, Christine H. Flood, Victor Luchangco, Jan-Willem Maessen, and Sukyoung Ryu. Fortress (Sun HPCS Language). In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 718–735. Springer, 2011.

[107] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, Santa Clara, CA, July 2015. USENIX Association.

[108] Chao Mei, Gengbin Zheng, Filippo Gioachin, and Laxmikant V. Kalé. Optimizing a parallel runtime system for multicore clusters: A case study. In *Proceedings of the 2010 TeraGrid Conference*, TG '10, pages 12:1–12:8, New York, NY, USA, 2010. ACM.

[109] Pritish Jetley and Laxmikant V. Kalé. Optimizations for message driven applications on multicore architectures. In *18th annual IEEE International Conference on High Performance Computing (HiPC 2011)*, December 2011.

[110] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE Computer Society Press, Nov 2012.

[111] Chuck L. Lawson, Richard J. Hanson, David R. Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[112] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 65:1–65:12, New York, NY, USA, 2011. ACM.

[113] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Scaling techniques for massive scale-free graphs in distributed (external) memory. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 825–836, Washington, DC, USA, 2013. IEEE Computer Society.

[114] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Faster parallel traversal of scale free graphs at extreme scale with vertex delegates. In *Proceedings of*

the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14, pages 549–559, Piscataway, NJ, USA, 2014. IEEE Press.

[115] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[116] Sun Chung and Anne Condon. Parallel implementation of boruvka's minimum spanning tree algorithm. *Parallel Processing Symposium, International*, 0:302, 1996.

[117] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, Sep. 1979.

[118] Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM.

[119] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[120] Alex Kogan and Maurice Herlihy. The future(s) of shared data structures. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 30–39, New York, NY, USA, 2014. ACM.

[121] James R. Goodman. *Cache consistency and sequential consistency.* University of Wisconsin-Madison, Computer Sciences Department, 1991.

[122] Richard J. Lipton and Jonathan S. Sandberg. Pram: A scalable shared memory. Technical Report TR-180-88, Department of Computer Science, Princenton University, September 1988.

[123] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[124] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ISCA '86, pages 434–442, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[125] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface.* MIT Press, Cambridge, MA, USA, 1999.

[126] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 15–26, New York, NY, USA, 1990. ACM.

[127] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Trans. Comput. Syst.*, 13(3):205–243, August 1995.

[128] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 13–21, New York, NY, USA, 1992. ACM.

[129] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical report, Carnegie Mellon University, Pittsburg, PA, USA, September 1991.

[130] Guang R. Gao and Vivek Sarkar. Location consistency-a new memory model and cache consistency protocol. *IEEE Trans. Comput.*, 49(8):798–813, August 2000.

[131] Charles Antony Richard Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[132] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: A unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 277–288, New York, NY, USA, 2008. ACM.

[133] Hermann Kopetz. Sparse time versus dense time in distributed real-time systems. In *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*, pages 460–467. IEEE, Jun 1992.

[134] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. *Information and Computation*, 118(1):158–179, 1995.

[135] Ilie Gabriel Tanase. *The STAPL Parallel Container Framework*. PhD thesis, Texas A&M University, 2 2012. `http://hdl.handle.net/1969.1/ETD-TAMU-2010-12-8753`.

[136] Jeremiah James Willcock, Torsten Hoefler, Nicholas Gerard Edmonds, and Andrew Lumsdaine. AM++: A Generalized Active Message Framework. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 401–410, New York, NY, USA, 2010. ACM.

[137] U Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *Proceedings of the IEEE International Conference on Data Mining*, pages 229–238. IEEE, 2009.

[138] 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 *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 32–39, New York, NY, USA, 2006. ACM.

[139] Mohammad J. Rashti and Ahmad Afsahi. A Speculative and Adaptive MPI Rendezvous Protocol Over RDMA-enabled Interconnects. *International Journal of Parallel Programming*, 37(2):223–246, 2009.

[140] Christian Bell and Dan Bonachea. A New DMA Registration Strategy for Pinning-Based High Performance Networks. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 198.1–, Washington, DC, USA, 2003. IEEE Computer Society.

[141] Wolfgang E. Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. VAMPIR: Visualization and analysis of MPI resources. http://www.tu-dresden.de/zih/vampir.

APPENDIX A

ARMI EXAMPLES

A.1 1-D Jacobi Stencil

An example use of the ARMI primitives presented in Section 3 is shown in Figures A.1 and A.2. This simple example implements a basic Jacobi solver* for the Laplace equation using two dimensions with finite differences.

The primitives used are explained in Table 3.1. The `async_rmi` is used for a point-to-point put, whereas the `allreduce_rmi` is used in place of `MPI_Allreduce`. `make_range_n` is a helper function to describe to the STAPL-RTS that it has to sent the first n objects starting from iterator `it`, rather than the whole container. Notice the use of `rmi_fence` that guarantees that all RMIs have been executed and the values to `xnew` have been written.

While the number of lines is similar, the unified communication interface of the STAPL-RTS (Figure A.2) provides a simpler programming model than that of the dual interfaces required to implement the hybrid MPI and OpenMP version shown in Figure A.1.

*Based on Jacobi solver from `http://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiexmpl/src/jacobi/C/main.html`

```
int main(int argc, char* argv[]) {
  using matrix_t = ...; // matrix or multiarray type of doubles

  MPI_Init(&argc, &argv);
  MPI_Comm comm = MPI_COMM_WORLD;
  int size = MPI_PROC_NULL;
  int rank = MPI_PROC_NULL;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);

  std::size_t maxn = ...; // matrix dimensions
  matrix_t xlocal  = ...; // matrix to solve, (maxn/size + 2) × maxn per process

  // xlocal[][0] is lower ghostpoints, xlocal[][maxn+2] is upper
  std::size_t i_first = 1;
  std::size_t i_last  = maxn/size;
  // top and bottom processes have one less row of interior points
  if (rank == 0)
    i_first++;
  if (rank == size - 1)
    i_last--;

  double gdiffnorm = DBL_MAX;
  for (int itcnt=1; itcnt<100 && gdiffnorm>1.0e-2; ++itcnt) {
    if (rank < size - 1)
      MPI_Send(xlocal[maxn/size], maxn, MPI_DOUBLE, rank+1, comm);
    if (rank > 0)
      MPI_Recv(xlocal[0], maxn, MPI_DOUBLE, rank-1, comm);
    if (rank > 0)
      MPI_Send(xlocal[1], maxn, MPI_DOUBLE, rank-1, comm);
    if (rank < size - 1)
      MPI_Recv(xlocal[maxn/size+1], maxn, MPI_DOUBLE, rank+1, comm);

    matrix_t xnew; // temporary matrix, (maxn/size + 2) × maxn per process

#pragma omp parallel for reduction(+:diffnorm)
    for (std::size_t i=i_first; i<=i_last; i++) {
      for (std::size_t j=1; j<maxn-1; j++) {
        xnew[i][j] = (xlocal[i][j+1] + xlocal[i][j-1] +
                      xlocal[i+1][j] + xlocal[i-1][j]) / 4.0;
        diffnorm += pow(xnew[i][j], xlocal[i][j], 2.0);
      }
    }

#pragma omp parallel for
    for (std::size_t i=i_first; i<=i_last; i++)
      for (std::size_t j=1; j<maxn-1; j++)
        xlocal[i][j] = xnew[i][j];

    MPI_Allreduce(&diffnorm, &gdiffnorm, 1, MPI_DOUBLE, MPI_SUM, comm);
    gdiffnorm = std::sqrt(gdiffnorm);
  }

  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

Figure A.1: MPI + OpenMP Jacobi solver

```
struct jacobi : public p_object {
  using matrix_t = ...; // matrix or multiarray type of doubles

  std::size_t maxn    = ...; // matrix dimensions
  matrix_t    xlocal  = ...; // matrix to solve, (maxn/size + 2) × maxn per process
  std::size_t i_first = ...; // same as in MPI version
  std::size_t i_last  = ...; // same as in MPI version
  double      diffnorm;

  template<typename Range>
  void recv_lower(Range const& v)
  { std::copy(v.begin(), v.end(), xlocal[0]); }

  template<typename Range>
  void recv_upper(Range const& v)
  { std::copy(v.begin(), v.end(), xlocal[m_maxn/this->get_num_locations()+1]); }

  double get_diffnorm() const { return diffnorm; }

  double do_iteration() {
    auto h    = this->get_rmi_handle();
    auto size = this->get_num_locations();
    auto rank = this->get_location_id();

    if (rank < size - 1)
      async_rmi(rank+1, h, &jacobi::recv_upper,
                make_range_n(xlocal[maxn/size], maxn));
    if (rank > 0)
      async_rmi(rank-1, h, &jacobi::recv_lower, make_range_n(xlocal[1], maxn));

    rmi_fence(); // wait for all writes

    matrix_t xnew; // temporary matrix, (maxn/size + 2) × maxn per process

    this->diffnorm = 0.0;
    for (std::size_t i=i_first; i<=i_last; i++) {
      for (std::size_t j=1; j<m_maxn-1; j++) {
        xnew[i][j] = (xlocal[i][j+1] + xlocal[i][j-1] +
                      xlocal[i+1][j] + xlocal[i-1][j]) / 4.0;
        this->diffnorm += pow(xnew[i][j], xlocal[i][j], 2.0);
      }
    }

    auto f = allreduce_rmi(std::plus<double>{}, h, &jacobi::get_diffnorm);

    for (std::size_t i = i_first; i <= i_last; ++i)
      for (std::size_t j=1; j<m_maxn-1; ++j)
        xlocal[i][j] = xnew[i][j];

    return std::sqrt(f.get());
  }
};

exit_code stapl_main(int, char**)
{
  jacobi_computation m;
  for (int itcnt=1; itcnt<100 && m.do_iteration()>1.0e-2; ++itcnt);
  return EXIT_SUCCESS;
}
```

Figure A.2: ARMI-based Jacobi solver

APPENDIX B

COMMUNICATION MECHANISM DETAILS

B.1 Communication Coarsening

Programming with asynchronous RMIs exposes the user to a fine-grain communication model. As it has been shown in previous publications [51, 60, 136], aggregating requests coarsens the fine-grain communication that is encountered in asynchronous systems, leading to overall better performance.

In the STAPL-RTS, we aggregate multiple RMI requests to the same location in the same outgoing buffer, or message [60]. We further enhance this mechanism by implementing *request combining*, a compression technique for RMI requests that target the same `p_object` representative and call the same member function.

At the sender, every time that an RMI is invoked, the triplet of location, target `p_object` and member function is checked against the triplet from the last aggregated RMI request. If the triplets differ, then we proceed with request aggregation. However, if the triplets are equal, we only aggregate the arguments of the request, eliminating duplicate data, a process called *request combining*. At the receiver, while processing RMIs contained in a message, each RMI is checked if it has participated in combining. If it is, then the translation from `rmi_handle` to `p_object` representative happens once and the same member function is invoked as many times as the combined requests. Thus, the overhead of request execution is reduced, as the translation happens only once.

Figure B.1 shows a simple distributed array implementation. A location invokes multiple RMI calls to `array::write` to the same location but with different data.

```
1  struct array : public stapl::p_object {
2   int m_value[...];
3   void write(std::size_t index, int t) { m_value[index] = t; }
4   int read(std::size_t index) const { return m_value[index]; }
5  };
6
7  stapl_main(...) {
8    array a;
9    auto h = a.get_rmi_handle();
10
11   if (stapl::get_location_id()==0) {
12     stapl::async_rmi(1, h, &A::write, 0, 10);
13     stapl::async_rmi(1, h, &A::write, 1, 20);
14     stapl::async_rmi(1, h, &A::write, 4, 30);
15   }
16 }
```

Figure B.1: RMI combining

Under aggregation, the outgoing buffer would have 3 instances of the target information (`rmi_handle` of the array and the pointer to member function) and the arguments, i.e., the index in the array and the value to be set. With request combining, it is automatically detected that the target (location, `p_object`, and member function) is always the same and therefore only on instance of the target information is sent along with the three sets of arguments.

Combining is possible only when the triplet of destination location, `p_object` and function match for two subsequent requests. For example, in Figure B.2 only Example 2 can combine the two call to `array::write`, since there is no other call between them. Combining the `array::write` calls in Example 1 would violate the ordering guarantees presented in Section 8. If preserving the RMI ordering is not required, then this type of combining would be possible, something that we plan to explore in future work. Figure B.3 presents the state of the STAPL-RTS internal buffers after aggregation only (Example 1), and combining and aggregation (Example 2).

Combining reduces the overall size of the buffer for a small cost at the sender location, which is offset by the reduced work that happens on the receiver, as the

```
1  stapl_main(...) {
2    array a;
3    auto h = a.get_rmi_handle();
4
5    if (stapl::get_location_id()==0) {
6      // Example 1: three aggregated requests, no combining possible
7      stapl::async_rmi(1, h, &A::write, 0, 10);
8      ... = stapl::opaque_rmi(1, h, &A::read);
9      stapl::async_rmi(1, h, &A::write, 1, 20);
10   }
11
12   if (stapl::get_location_id()==0) {
13     // Example 2: two combined requests and one aggregated
14     stapl::async_rmi(1, h, &A::write, 0, 10);
15     stapl::async_rmi(1, h, &A::write, 1, 20);
16     ... = stapl::opaque_rmi(1, h, &A::read);
17   }
18 }
```

Figure B.2: RMI combining opportunities

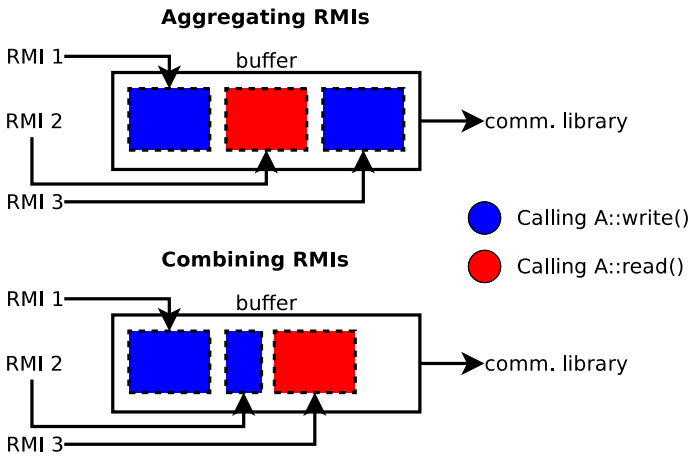
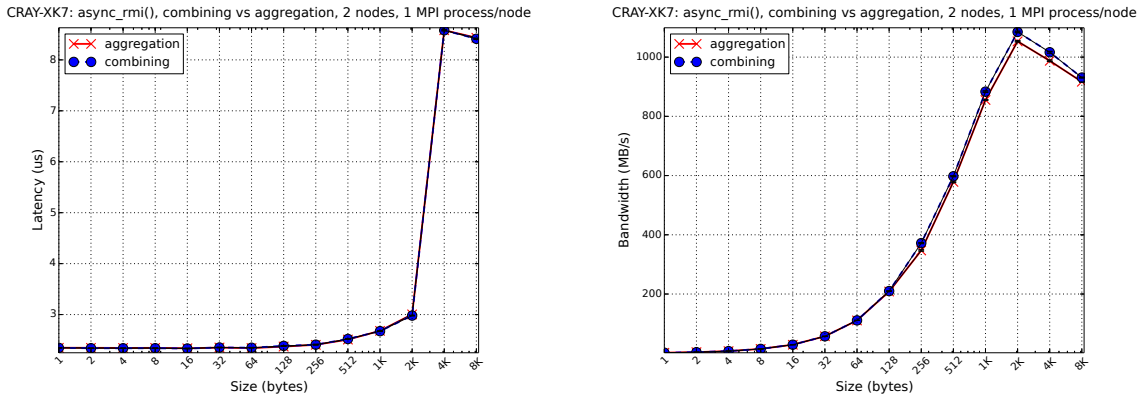


Figure B.3: Aggregation vs combining



(a) Latency (b) Bandwidth

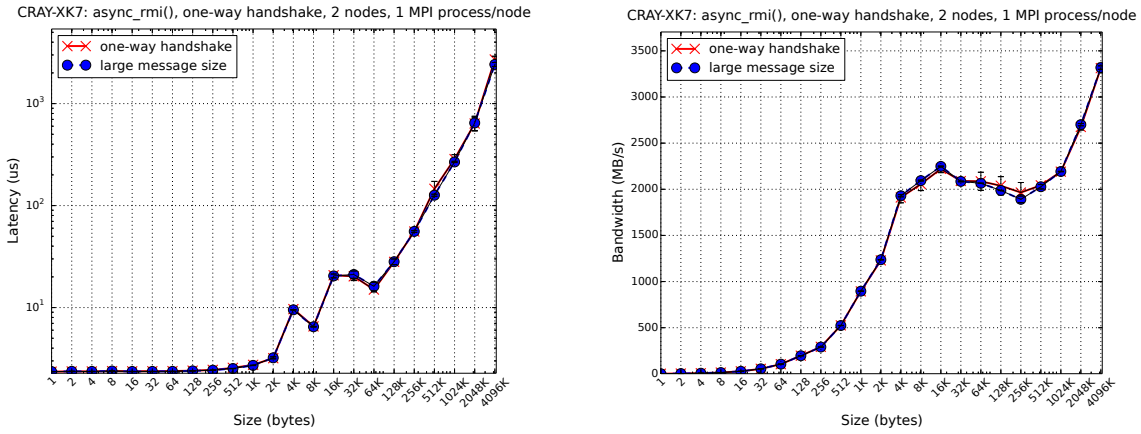
Figure B.4: Combining vs aggregation on Cray-XK7

`rmi_handle`-to-pointer translation happens once and a virtual function call is elided for each one of the combined RMI requests. Figure B.4 shows the latency and bandwidth achieved on two nodes of Cray-XK7 with (`combining`) and without (`aggregation`) combining using `async_rmi`. Since the overhead is minimal, latency is unaffected; a slow combining technique would have increased the time to create an RMI request, manifesting itself as latency. Bandwidth improves with combining, as more requests can fit in the same buffer. The abnormal increase in latency between from 2 KB to $4 - 8$ KB is due to the MPI implementation, something that can be seen in Section 6.5.1 for `MPI_Send`/`MPI_Recv` as well*.

B.2 One-way Handshake Protocol

Another issue faced by frameworks that support one-sided transfers coupled with function invocations (RMI, RPC, Active Messages, etc.) is how to copy data of arbitrary size from one address space to another. While the sender is aware of the data payload size to be communicated, the receiver also has to be ready to receive

*For this experiment, all RMIs have to go through the MPI layer, as the locations are in different nodes.



(a) Latency (log-log plot) (b) Bandwidth

Figure B.5: One-way handshake vs large message sizes on CRAY-XK7

such a payload. A number of protocols have been invented such as rendez-vous protocols [138, 139], bounce buffers, firehose [140] and others to address this issue.

Our current MPI-based communicator implementation uses pairs of `MPI_Isend` and `MPI_Irecv` calls for all distributed memory communication. We pre-post some messages using `MPI_Irecv` using a predetermined tag T in a STAPL-RTS owned MPI communicator. These messages have a default size that is configurable at run-time, with a default value of two pages (typically 8 KB).

All messages that are of smaller or equal size as the default can be trivially sent and received. For messages larger that the default size, we have implemented a lightweight communication protocol, called *one-way handshake*. In the case that the message is larger than the default size, then the sender will only send the first 8 bytes of the header of the buffer in the default channel, using the tag T. These bytes contain the actual size of the message and are only part of the full message header, that contains additional information. Then it sends the full message in a different channel with a predefined tag $T', T' \neq T$.

The receiver, upon receiving just 8 bytes and not the full header, posts a new `MPI_Irecv` in the other channel with MPI tag T', to receive the large message. In order to avoid message reordering that would violate the ordering presented in Section 8 and protect the process from being overwhelmed with large messages and run out of memory, we do not allow any other message to be received until the large message has been received. When the large message is received, normal communication resumes.

Figure B.5 evaluates the one-way handshake by using a default message size of 8 KB (`one-way handshake`) against using a default message size of 4 MB (`large message size`), which would avoid the one-way handshake for all payloads, both for latency (Figure B.5(a)) and bandwidth (Figure B.5(b)). The one-way handshake imposes some overhead and does exhibit higher variability than using the larger message size, something that is expected, as the former sends two messages to complete the operation. However, with the one-way handshake, the STAPL-RTS is not required to post the largest message expected, thus reducing memory use.

APPENDIX C

STAPL-RTS CODE ORGANIZATION

In this section we briefly present a high level overview of the code organization of the STAPL-RTS as of April 21, 2016 and revision `r12272` of the STAPL SVN repository. A partial code directory structure can be seen in Figure C.1.

- `collective`: Contains helper `p_object`s for implementing collective operations. Normally, each collective RMI is backed by such a `p_object`, for example `allgather_rmi` relies on the `allgather_object`.
- `communicator`: Backends for communication layers, such as MPI, as well as distributed memory collective operations implementations. This code is used in the implementation of the runqueue.
- `concurrency`: Multithreaded backends based on existing multithreading libraries, such as OpenMP and C++11 threads. This directory contains also optimized shared memory collective operations (e.g., reductions, barriers).
- `config`: Configuration headers to automatically recognize platform capabilities (e.g., data alignment) and define the various internal STAPL-RTS types, such as the gang and location IDs.
- `counter`: Counter infrastructure implementation (see Section 3.2) that uses platform dependent libraries, such as PAPI-backed counters. It includes configuration files for automatic and user-guided counter discovery.
- `executor`: Implementation of the EXECUTORs and their associated scheduling and work-stealing capabilities (`scheduler` subdirectory). Task dependence

Figure C.1: Code organization tree

graph objects (i.e., `PARAGRAPH`) are the main users of this code.

- `instrumentation`: Support for instrumentation and bindings to TAU [64], MPE [65], and vampir [141].
- `non_rmi`: High-level interfaces of non-RMI primitives, such as `construct` and `external_call`.
- `request`: Building blocks for implementing both RMI and non-RMI requests. This directory contains code for implementing the primitives offered in `non_rmi` and `rmi` directories.
- `rmi`: Implementation of interfaces for all RMI primitives, such as `async_rmi` and `opaque_rmi`.
- `serialization`: Marshalling support for C++ objects that includes `typer` and Boost.Serialization integration as described in Section 3.2.5.
- `type_traits`: Template-based interfaces to query or modify types at compile-time, similar to the C++11 [34] and Boost [66] type traits support.
- `utility`: Utility classes such as implementations of type-erased ranges, functions that behave as input iterator ranges, and specialized allocators.
- `aggregators.hpp, context.hpp, runqueue.hpp`: Support for RMI creation, aggregation and combining, and execution.
- `gang_md.hpp, gang_md_registry.hpp, location_md.hpp`: Metadata classes for gang support and SPMD sections, and location-to-core mapping.
- `immutable_range.hpp, immutable_ref.hpp, immutable_shared.hpp, range.hpp`: Support classes for zero-copy as presented in Section 6.