FG-MPI: Fine-Grain MPI

by

Humaira Kamal

M.Sc., The University of British Columbia, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

The Faculty of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

July 2013

Abstract

The Message Passing Interface (MPI) is widely used to write sophisticated parallel applications ranging from cognitive computing to weather predictions and is almost universally adopted for High Performance Computing (HPC).

Many popular MPI implementations bind MPI processes to OS-processes. This runtime model has closely matched single or multi-processor compute clusters. Since 2008, however, clusters of multicore nodes have been the predominant architecture for HPC, with the opportunity for parallelism inside one compute node. There are a number of popular parallel programming languages for multicore that use message passing. One notable difference between MPI and these languages is the granularity of the MPI processes. Processes written using MPI tend to be coarse-grained and designed to match the number of processes to the available hardware, rather than the program structure. Binding MPI processes to OS-processes fails to take full advantage of the finer-grain parallelism available on today's multicore systems. Our goal was to take advantage of the type of runtime systems used by fine-grain languages and integrate that into MPI to obtain the best of these programming models; the ability to have fine-grain parallelism, while maintaining MPI's rich support for communication inside clusters.

Fine-Grain MPI (FG-MPI) is a system that extends the execution model of MPI to include interleaved concurrency through integration into the MPI middleware. FG-MPI is integrated into the MPICH2 middleware, which is an open source, production-quality implementation of MPI. The FG-MPI runtime uses coroutines to implement light-weight

MPI processes that are non-preemptively scheduled by its MPI-aware scheduler. The use of coroutines enables fast context-switching time and low communication and synchronization overhead. FG-MPI enables expression of finer-grain function-level parallelism, which allows for flexible process mapping, scalability, and can lead to better program performance.

We have demonstrated FG-MPI's ability to scale to over a 100 million MPI processes on a large cluster of 6,480 cores. This is the first time any system has executed such a large number of MPI processes, and this capability will be useful in exploring scalability issues of the MPI middleware as systems move towards compute clusters with millions of processor cores.

Preface

Following is a list of publications arising from the work presented in my dissertation. All of this work was done under the supervision of Dr. Alan S. Wagner, who is a co-author on these publications. The dissertation does not contain work by any other collaborators.

[81] Humaira Kamal and Alan Wagner. An integrated fine-grain runtime system for MPI. *Computing*, pages 1–17, 2013.

[77] H. Kamal and A. Wagner. Added concurrency to improve MPI performance on multicore. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 229–238, 2012.

[80] H. Kamal and A. Wagner. An integrated runtime scheduler for MPI. In Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 7490 of *Lecture Notes in Computer Science*, pages 173–182. Springer Berlin Heidelberg, 2012.

[78] Humaira Kamal, Seyed M. Mirtaheri, and Alan Wagner. Scalability of communicators and groups in MPI. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 264–275, New York, NY, USA, 2010. ACM.

[79] Humaira Kamal and Alan Wagner. FG-MPI: Fine-Grain MPI for multicore and clusters. In *11th IEEE Intl. Workshop on Parallel and Distributed Scientific and En-*

gineering Computing (PDSEC) held in conjunction with IPDPS-24, pages 1–8, April 2010.

Chapter 2 presents an extended description of the work that was published in [81]. The discussion in Section 3.1 was partly published in [78] and provides a more detailed description of the system components. The rest of the sections in Chapter 3 have not appeared in publications elsewhere.

The system description and experiments presented in Chapter 4 were published and presented at [77]. The work presented in Section 5.1 appeared in [79], and that in Section 5.2 was published in [81]. The code example in Appendix B was provided by Dr. Alan Wagner. The remaining parts of the dissertation have not been published before.

Table of Contents

Appendices

List of Tables

List of Figures

Acknowledgements

I owe a debt of gratitude to my supervisor, Dr. Alan Wagner, for his support and mentoring throughout my research work. I thank him for his generosity with his time and advice and his constant encouragement and optimism. I am grateful for the opportunity of having worked with him.

I thank Dr. Mark Greenstreet, Dr. Eric Wohlstadter, Dr. Norm Hutchinson and Dr. Matthew Choptuik for their support and the time they spent reading my thesis and giving me useful feedback.

I thank my friends for the wonderful time I have spent with them. You have always made my days warmer and brighter.

Words are inadequate for me to express my gratitude to my family. They have always cheered me on and always been there when I needed them. There is not much I can say except that I am blessed to have a family like you. *Thank You.*

Dedication

Dedicated to my parents with love and gratitude.

Chapter 1

Introduction

> The lessons from MPI can be summed up as follows: It is more important to make the hard things possible than it is to make the easy things easy.
>
> William D. Gropp

Over the last two decades, MPI (Message Passing Interface) [97] has been hugely successful for High Performance Computing (HPC). It is almost universally adopted for writing sophisticated scientific parallel code and mature numerical libraries that can scale to run on some of the world's largest supercomputers. Most application developers for supercomputing centers regard using MPI as essential for their software [15, 49]. Some of the factors contributing to MPI's wide adoption include its support for software libraries, composibility, portability and consistent performance over a diverse set of architectures and network fabrics. The MPI standard is actively maintained by the MPI Forum, which is a broad consortium of national laboratories, universities, vendors, and library developers. The MPI standard owes its success to precisely-defined and forward-looking features based on active collaboration with implementers, vendors, and users [55]. There are a number of vendor-specific as well as open source MPI implementations including MPICH2 [9] and Open MPI [141], which are extensively used worldwide.

The architecture of modern microprocessors has changed dramatically since the introduction of multicore processors. Microprocessors today have multiple cores and the number of cores per chip is increasing. There is considerable interest in the MPI middleware as it relates to multicore processors [12, 47, 55, 93]. The focus of my thesis is on the following problem.

1.0.1 Problem Statement

> *Many popular implementations of the MPI library bind MPI processes to OS-processes. Processes written using MPI, as a result, tend to be coarse-grained and designed to match the available hardware rather than the program structure. Binding to OS-processes fails to take full advantage of the finer-grain parallelism available on today's multicore systems.*

The first definition of the MPI standard dates back to 1993, when many machines had a single processor that consisted of multiple chips, and one user process was run on each processor [55]. The MPI standard does not define the term "process", however, many MPI library implementations [9, 141] bind MPI processes to OS-processes. This runtime model has been successful since it closely matched single or multi-processor compute clusters. Since 2008, however, clusters of multicore nodes have been the predominant architecture for HPC [145], with the opportunity for parallelism inside one compute node[1]. As a result of binding to OS-processes, processes written using MPI tend to be coarse-grained and programmed to make it easy to match them to the available hardware, rather than to the structure of the program. Although fine-grain parallelism may have benefited single processor systems, the need for it is more critical for exploiting parallelism available on today's multicore systems. It is not easy to exploit fine-grain parallelism and hybrid approaches have become common where MPI is used as a communication mechanism between nodes, combined with pre-emptive threads or APIs such as OpenMP [108] on a node. The pre-emptive thread model, however, introduces programming complexity [60, 86, 91] as well as

---

[1] We will be using the terms "node" and "machine" interchangeably in this document to refer to a single computational node with multiple processor cores, operating under a single operating system.

significant system overheads for the MPI middleware to support thread safety [13, 59, 140].

Mixing MPI with a pthread-based API such as OpenMP introduces another set of issues related to resource contention and coordination. The interface between MPI and other programming models is not well defined and there is no clear separation of concerns [57]. This results in different runtime systems attempting to optimize the use of threads and processor cores independently of each other and potentially leading to contention. A recent post in the MPICH2 mailing list highlights the problem of mixing MPI with pthreads [73]. In this user-case, a thread on the critical path delays message communication due to the difficulty of coordinating accesses of multiple pthreads to the middleware. The locking overhead in the middleware in this case resulted in considerably low performance. The issues of mixing programming models as well as the programming complexity will magnify as systems scale.

There are a number of multicore languages such as Erlang [11], Haskell [45], Scala [105] and Go [48], that support fine-grain parallel programming as a way to take advantage of multicore processors. The ability to express fine-grain parallelism makes it possible to seamlessly scale with the number of cores in the node. However, these languages do not have the extensive middleware support needed for messaging and communication over a variety of network fabrics as provided by MPI.

MPI, by itself, is perceived to be difficult to program with the message passing model considered as too low-level [55] and the binding to OS-processes as heavy-weight. These perceptions, however, are not due to a requirement of the MPI specification, but a property of the implementation of the MPI library and its runtime system. The MPI specification does not equate an MPI process to an OS-process and does not preclude a fine-grain interpretation. Our goal was to take advantage of the type of runtime systems used by fine-grain languages and to integrate that into MPI to obtain the best of these programming models; the ability to have fine-grain parallelism, while maintaining MPI's rich support for communication between machines.

As part of my thesis, I have designed and implemented Fine-Grain MPI (FG-MPI), a system that allows execution of millions of MPI processes on a node or communicating between nodes inside a cluster [77–81]. FG-MPI is integrated into the MPICH2 [9] middleware, which is a popular open source implementation of the MPI library from Argonne National Laboratory. FG-MPI decouples the MPI processes from the hardware and implements a user-level runtime system that is integrated into the middleware to allow for interleaved execution of multiple concurrent MPI processes inside an OS-process. This adds a new dimension to mapping processes onto nodes, which enables us to allocate some processes to run interleaved within an OS-process and others to run in parallel in different OS-processes on other cores. Coroutines are used in FG-MPI to implement light-weight MPI processes that are non-preemptively scheduled by its runtime system. The benefits of coroutine-based non-preemptive threads include fast context-switching time, low communication and synchronization overhead and ability to support thousands of MPI processes inside a single OS process (see Chapters 4 and 5). The communication between the MPI processes is seamless, whether it is between two processes in the same OS-process or between processes on different cores or machines. By decoupling MPI processes from an OS process, it is possible to vary the number of MPI processes to match the problem rather than the hardware. Moreover, the same MPI program can be flexibly mapped to OS-processes and cores and executed, without changes, on a laptop or on a cluster.

1.0.2 Thesis Statement

> *Fine-Grain MPI extends the execution model of MPI to include interleaved concurrency through integration into the MPI middleware. FG-MPI enables expression of finer-grain function-level parallelism, which allows for flexible process mapping, scalability, and can lead to better program performance.*

1.1 Challenges

One of the goals of the FG-MPI project was to enable transfer of technology and have an impact in the HPC community through extending an existing open source implementation of the MPI standard. The MPI community follows a rigorous practise of implementing, validating, and testing any extensions before adopting them. Our objective was to provide a proof of concept for the effectiveness of a fine-grain execution model through a working system. We chose to integrate our system in the MPICH2 middleware due to its extensive and successful adoption by national laboratories, academia, industry vendors, and supercomputer centers worldwide. MPICH2 is a high quality production system and has won awards for excellence and innovation. Integration of FG-MPI into MPICH2 enables it to leverage its portability and rich support for communication over a variety of network fabrics.

There were a number of challenges in our decision of integrating FG-MPI into MPICH2. Firstly, it is not straightforward to make conceptual design changes to an existing system while leveraging their optimized code base. The little documentation available was high-level and I had to read the code extensively in order to extend it. Secondly, it was not known what roadblocks I would encounter and whether the project would be successful. This was not a project that required extensions to a particular component of the middleware. A challenge with integration into MPICH2 was that many changes were needed across all the layers of the MPICH2 architecture (Figure 2.1, page 20) before it was possible to execute even the simplest MPI program.

Scalability and sharing of middleware structures was another challenging aspect of the project. There were a number of low-level design issues and subtle corner cases that had to be addressed. Debugging a system with hundreds and thousands of processes posed difficulties of its own. In order to manage the complexity of the project, I adopted a phased approach to the implementation, starting with point to point communication routines and gradually adding other functionality.

1.2 Contributions

The main contributions of my thesis are as follows.

- FG-MPI is integrated into MPICH2, a popular open source implementation of the MPI library, and uses its low-latency and scalable communication subsystem for intra-node and inter-node communication. Integration in MPICH2 allows FG-MPI to leverage its optimized code base and rich support for network fabrics in cluster environments. The design and implementation of the FG-MPI system is described in Chapters 2 and 3.
- We investigated scalability issues related to MPI groups and communicators and defined new efficient algorithms for communicator creation and storage of process maps. Communicators in MPI allow code composition and enable development of parallel libraries by defining separate communication contexts. Communicators provide a scoping mechanism that ensures that communication within one software component will not interfere with another component. The ability to structure and group MPI processes becomes more important when dealing with millions of MPI processes (Chapter 3).
- FG-MPI runtime system uses light-weight coroutines to expose massive concurrency. The use of coroutines enables fast context-switching time as well as low communication and synchronization overhead. (See Chapter 4).
- We designed techniques to exploit the locality of MPI processes for communication efficiency and implement optimized communication between concurrent processes in the same OS-process (Chapters 3 and 4).
- FG-MPI implements a MPI-aware user-level scheduler that works in concert with MPICH2's progress engine and is responsive to events occurring inside the middleware.

The design of the scheduler is described in Chapter 2.

- FG-MPI's runtime scheduler can be used to simplify programming by avoiding non-blocking communication that often makes MPI programming difficult. The scheduler relieves the programmer from scheduling computation and communication inside the application and brings the performance part outside of the program specification into the runtime (Chapter 5).

- FG-MPI adds an additional dimension to mapping processes onto nodes to allow for interleaved execution of multiple concurrent MPI processes inside an OS-process. MPI processes can be flexibly mapped to OS-processes and cores and seamlessly executed without requiring changes, whether it is on a laptop or a cluster (Chapter 3).

  This added degree of freedom can be used to match the granularity of processes to better fit the cache and improve the performance of existing algorithms. This is described in Chapter 4.

- FG-MPI enables a task-oriented programming approach that makes it easier to exploit function-level parallelism. In Chapter 5, we compare FG-MPI with other multicore parallel languages and runtime systems and show it achieves equal or better performance.

- We have demonstrated FG-MPI's ability to scale to over a 100 million MPI processes on a large cluster of 6,480 cores. This is the first time any system has executed such a large number of MPI processes, and this capability will be useful in exploring the performance and scalability issues of the MPI middleware as systems move towards exascale. (Chapter 5).

1.3 Related Work

There is past work on thread-based implementations of MPI which introduce their own runtime system. Prior to 1999, projects like TOMPI [36] and TMPI [130, 136] were developed for execution on single workstations or multi-programmed shared memory machines. The main focus of these projects was to implement MPI processes as threads and use shared memory to reduce communication and context switching overhead among them.

The objective of TOMPI (Threads Only MPI) [36] project was to facilitate development of parallel programs on a single workstation. TOMPI was designed to run as a single Unix process and uses semaphores and condition variables for synchronization among threads. TOMPI is a prototype that implements a subset of MPI routines and is primarily for testing purposes. It does not address any middleware communication or scheduling issues.

TMPI (Threaded MPI) [130, 136], is a project designed for shared memory machines in late 1990s with the objective to improve performance of MPI in the presence of space/-time multiprogramming. A later version of TMPI [137] targeted Linux SMP clusters and borrowed many design ideas from the MPICH [51, 58] implementation. TMPI maps MPI processes to pthreads inside a single address space on a cluster node. It introduces an additional daemon thread on each cluster node for buffering and serialization of incoming messages. The authors identify the use of a single daemon receive thread for all incoming messages as a potential bottleneck for scalability. The experimental results presented for TMPI are only for dedicated SMP clusters, where the number of MPI processes does not exceed the number of processors (numbers reported in [137] are for up to 24 MPI processes).

AzequiaMPI [37] is a more recent thread-based implementation with objectives similar to TMPI. AzequiaMPI is layered on top of Azequia runtime system and uses pthreads mutex and condition variables for synchronization of communication among threads. Interestingly, the authors stress the use of locks in their implementation for portability reasons in [37], but a later poster paper [121] mentions using lock-free queues and borrows several design concepts from the MPICH2 implementation. Apart from implementing more MPI routines, it is unclear what their design offers over other pre-emptive thread-based implementations. The results reported by AzequiaMPI are for benchmarks on a single multicore node with up to eight MPI processes [121], where the number of MPI tasks are equal to the number of cores.

MPI Accelerator (MPIActor) [90] is another work that implements MPI processes as pre-emptive threads for the same reasons mentioned above, i.e., to improve intra-node communication efficiency through shared memory. They implement a runtime system layered on top of MPI, with one MPI process on each compute node. The underlying MPI library is aware of only a single MPI process on each machine and is used as an inter-node communication mechanism. All the MPI communication routines in the program are intercepted and replaced by their own MPIActor routines, which provides the abstraction of MPI processes mapped onto threads. There are a number of limitations to their approach. The first is that it requires the underlying MPI library to be thread-safe with support for `MPI_THREAD_MULTIPLE`[2] level. The second limitation is their choice of using the tag parameter in the MPI point-to-point communication routines as a way to map different MPI process ranks onto threads inside an MPI process. The authors use the 32 bits in the tag to store the communication type, the ranks of the source and the destination, as well as the tag value. Not only does it change the semantics of the tag argument, it is not portable to a system where the tag may be stored in a fewer or greater number of bits. As well, the uniqueness of the tag argument cannot be guaranteed in a multi-module library, and the authors do not explain how their system handles different communicators.

While the above projects have focused on implementing thread-based MPI for communication efficiency on shared memory systems, other researchers have looked at alternative solutions for improving intra-node communication for existing process-based MPI

---

[2] `MPI_THREAD_MULTIPLE` is the highest of the four levels of thread safety defined by MPI. It allows multiple threads in a process to make MPI calls simultaneously.

implementations through efficient data transfer techniques [22, 23, 75]. A number of intra-node communication mechanisms are available such as user-space shared memory segments, NIC level loop-back, and kernel-based memory mapping for efficient shared message passing [23]. Techniques for mapping the memory of one process into another process's virtual address space through `mmap` or XPMEM [4] are well known. Kernel-based mapping avoids intermediate system buffer copies that are involved in passing messages across different address spaces, and recent research has focused on improving its performance and portability. LiMIC (Linux kernel module for MPI Intra-node Communication) [75] is a portable kernel module interface that enables a direct copy from one process to another. LiMIC is implemented as a runtime loadable module and was designed to provide portability across different interconnects. The authors report substantial improvement in bandwidth and latency with LiMIC on their system. KNEM [46] is another kernel module that supports Direct Memory Access (DMA) copy and the work in [94, 99] shows promising results for improving the performance of intra-node communication in MPI.

There is significant diversity in the design and architectures of multicore systems by different vendors, each providing multiple levels of cache hierarchies and topologies. Work by Dongarra *et. al.* [93] proposes a framework that discovers information about the hardware at runtime, based on the locality and topology of cores, and uses this information to tune the point-to-point intra-node communication in the MPI library. The FG-MPI implementation can also make use of these kernel-based and topology-aware optimization schemes for communication among the OS-processes. FG-MPI uses the Nemesis [21, 24] channel in the MPICH2 implementation which employs a number of techniques for efficient data transfer such as lock-free shared memory queues, a double-buffering mechanism to enable the receiver to copy data out of a shared memory region while the sender copies it in, and use of non-temporal store operations for memory copy for high bandwidth transfer while reducing the impact on the application's data in the receiver's cache [24]. The MPICH2 group has also explored kernel-based data transfers with the Nemesis channel and shown that it can lead to speedup and better cache efficiency [22]. FG-MPI can leverage any communication optimizations that are introduced into Nemesis in the future.

MPI-LITE [17], MPC [112] and AMPI [64] are three projects that enable over-subscription of user-level MPI tasks through process virtualization. MPI-LITE is an early prototype developed in 1997 that implements MPI processes as user-level threads within an OS-process. MPI-LITE uses MPI for communication between OS-processes and its own runtime for inter-thread communication and requires special suffixes for MPI routines to distinguish between the two types of communication. This project was developed for IBM SP2 systems and is no longer active. MPC is a recent project that supports user-level MPI tasks through process virtualization. The main motivation of the MPC (MultiProcessor Communications environment) project [25, 111, 112] is to improve the performance of hybrid OpenMP+MPI programming. This work points to the complexities of mixing two programming models whose runtime systems may work independently of each other and compete for the same resources. MPC proposes a unified runtime that supports both OpenMP and MPI through their own MPC API. MPC requires a modified GCC compiler that transforms the OpenMP directives into MPC internal calls. It implements a custom user-level $M \times N$ thread library for intra-node communication where threads are spawned for MPI tasks and OpenMP parallel regions and managed by their scheduler. Inter-node communication is done through an underlying MPI library, which must support the `MPI_THREAD_MULTIPLE` level. As with other thread-based approaches, MPC has to deal with synchronization for inter-task communication among threads on different cores.

Adaptive MPI (AMPI) [64] is another system that supports added concurrency by implementing MPI layered on top of Charm++ [76], which is an object oriented system based on C++. A virtual MPI process in AMPI is represented as a collection of objects, which passes messages to another MPI process by invoking an entry method on the remote object. AMPI defines a number of virtual processors (VPs) and the program computation is divided among them. These virtual processors are mapped onto the physical processors and managed by their runtime system. There are benefits to implementing MPI on top of a runtime system like Charm++ since it becomes possible to support process migration, which is the main focus of AMPI. FG-MPI takes a completely different approach that integrates interleaved concurrency directly into the MPICH2 implementation. Our approach reduces the overheads that arises from over-subscription of cores so that the benefits of adding concurrency can be extended to more problems. In Chapter 4 we compare the point-to-point messaging and barrier costs in FG-MPI to those in AMPI and show that FG-MPI outperforms it by a large margin. The performance advantages of AMPI have been reported on a subset of small problem sizes of the NAS parallel benchmarks [103], but not on a wider range and larger sizes of the benchmarks and they do not discuss issues pertaining to multicore. We have carried out an extensive set of experiments on the complete set of the NAS parallel benchmarks over large problem sizes and discuss the trade-offs between the added costs and benefits (Section 4.2).

Our approach in FG-MPI is very different from the thread-based implementations described in this section. Our use of non-preemptive, MPI-aware scheduling avoids many of the synchronization and locking overheads that arises with threads [140], while providing fast context switching through coroutines. It is also interesting to note that sharing variables in a thread-based implementation can result in cache performance penalties depending on the access pattern of the variable and its placement in a NUMA system. There are trade-offs between reducing data duplication and the increase in cache coherency traffic for variables that are modified often [138]. Projects like Barrelfish [16, 128, 135] have shown that an operating system that uses message passing instead of shared data communication is scalable and offers tangible benefits, even on present day systems. As well, several cache-coherent multicore systems available today use message passing hardware instead of a single shared interconnect for scalability reasons [30, 69]. From the programming perspective, message passing simplifies reasoning about the program's state and is portable to multicore systems with or without support for cache-coherence.

A summary of the different MPI projects discussed in this section is presented in Appendix C. The earlier attempts at thread-based MPI implementations are more than ten years old and there has been little activity on this front until the emergence of multicore systems. As Appendix C shows these projects were designed with different goals and most have focused on developing (a) a runtime system from scratch for intra-node communication while using MPI as a mechanism for inter-node communication, or (b) layering an MPI implementation on top of an existing runtime system. FG-MPI takes a different approach by integrating into MPICH2 with a focus on exposing large-scale, function-level parallelism.

1.4 Thesis Synopsis

The organization of my thesis is as follows. In Chapter 1, I discuss the motivation for the FG-MPI project and describe the contributions of my work. I also present a discussion of the related work and how their systems differ from our implementation. The design of the FG-MPI runtime system is presented in Chapters 2 and 3. Chapter 2 discusses the main design issues in our integrated approach of adding interleaved concurrency to the MPICH2 implementation. I describe the motivation for using coroutines and non-preemptive threads, the issues involved in integrating FG-MPI into a production quality middleware, and the design of the runtime scheduler and its interactions with the MPI progress engine. Chapter 3 describes low-level design and implementation issues of the different system components. In Section 3.1, I discuss the scalability of communicators and groups and the techniques we have employed for sharing the group information and design of scalable algorithms for communicator creation. The design of communication queues is presented in Section 3.2, along with a discussion of how they are shared among the co-located processes. In Section 3.3 I discuss the design and implementation of zero-copy communication routines to optimize data transfer within a single address space.

FG-MPI enables a task-oriented programming approach and support for MPMD (Multiple Program Multiple Data) is provided through a second dimension of mapping of the MPI processes. This extra degree of freedom and process deployment in FG-MPI is discussed in Section 3.4. Limitations of using non-preemptive multitasking within a single address space are described in Section 3.5.

We discuss the benefits of added concurrency to improve the performance of MPI programs on multicore systems in Chapter 4. One of the key issues for added concurrency to be effective is to reduce the overheads related to context switches, scheduling and extra message passing. In Sections 4.1.1 and 4.1.2 we measure these overheads and show that FG-MPI achieves fast context switches and low message passing overhead among co-located processes. We also show that there are potential advantages in passing more smaller messages instead of larger messages. Messaging costs and context switch overhead can have a significant impact on collective operations. In Section 4.1.3 we describe a location-aware implementation of the barrier operation that takes advantage of the single address space to offset these overheads and speed up execution.

Substantial performance gains are possible by adjusting the grain-size of a program to better fit the cache. In Section 4.1.4 we carry out a series of experiments to evaluate the effect of finer-granularity in the program on the cache behavior. In Section 4.2, we evaluate the use of FG-MPI on the complete set of the NAS parallel benchmarks [27] over large problem sizes and show that substantial improvements are possible by adjusting the program granularity. We also discuss the characteristics of the benchmarks with regards to trade-offs between the added costs and benefits.

In Chapter 5, we discuss FG-MPI's support for function-level parallelism and show how it can be used as a tool for structuring parallel programs to better match the problem. Section 5.1 describes a real-world example from the CoSMoS [31] project that was modified to use FG-MPI and models emergent behavior through thousands of MPI processes. We compare the performance of FG-MPI with several fine-grain multicore languages and show that it compares favourably with them. In Section 5.2, we use FG-MPI to re-structure a typical use of non-blocking communication in MPI and show that having multiple MPI processes per OS-process, with a runtime scheduler, can be used to simplify MPI programming and achieve performance without adding complexity to the program. Finally in Section 5.3, we test the ability of FG-MPI to scale to massively parallel programs and run over a 100 million MPI processes on the WestGrid computing facility [50]. Conclusions and directions for future work are presented in Chapter 6.

Chapter 2

A Fine-Grain Integrated Runtime System

> If I have seen further it is by standing on the shoulders of giants.
>
> Bernard of Chartres

In this chapter we describe the design of the FG-MPI runtime system and discuss the integrated approach to adding concurrency to MPI programs by making it possible to have multiple MPI processes per OS-process. We describe the main design issues in FG-MPI that made it possible to support additional concurrency: (a) the use of coroutines and non-preemptive threads, (b) the integration of FG-MPI into existing middleware (MPICH2) rather than a layer running on top of MPI and (c) the design of an integrated scheduler and its interactions with the MPI progress engine.

In order to avoid any ambiguity we will use the term "*OS-process*" when referring to operating systems processes and at all other places the terms *process*, *fine-grain process* and *MPI process* will be used interchangeably. MPI processes sharing the same address space are referred to as *co-located* processes.

2.1 FG-MPI Runtime

One major decision in the design of FG-MPI and the support for multiple MPI processes within an OS-process was the use of coroutines as a basis for non-preemptive scheduling of the processes. Use of coroutines to implement cooperative (i.e., non-preemptive) multitasking provides a number of advantages such as minimizing the need for synchronization, avoidance of conflicts due to race conditions [60, 100], and ease of reasoning about the concurrency in the program [5]. Due to these benefits, there has been renewed interest in the use of coroutines in a number of systems [5, 39, 42, 84, 134, 148]. Capriccio and other systems [84, 132, 149] have shown that coroutine-based threads have fast context-switching time, low communication and synchronization overhead and scale to support large numbers of threads. The benefits of coroutines at the language level are well-known and they are supported in many languages (Python [127, 142], Lua [67, 68]) including parallel languages used on multicore (Erlang [11], Go [48], Occam-pi [106]).

MPI processes in FG-MPI are implemented as light-weight non-preemptive threads which, along with the runtime scheduler, are implemented on top of coroutines. Our system uses a modular approach and is capable of making use of different coroutine libraries through a configuration option. We currently support Toernig's coroutine library [143], and PCL (Portable Coroutine Library) [89]. MPI has an added advantage in the context of cooperative multitasking that makes coordination among MPI processes easier. Writing an MPI program requires decomposing the work among processes and specifying the communication between them. The messaging-passing and calls to the middleware provide a natural yield point for the non-preemptive threads, where one process can cooperatively yield control to another. Since only one co-located process is active, it was possible to share the middleware without using locks and ensure that the middleware is in a consistent state between scheduling points.

Use of non-preemptive processes was crucial with regard to managing large-scale concurrency in the middleware. Programming with pre-emptive threads is widely acknowledged to be difficult and even though mechanisms like locks, semaphores and monitors are available to manage synchronization, errors often lurk in even carefully designed code. As multicore architectures scale up and increase the amount of parallelism, finer thread interleavings will occur and these errors will show up more frequently to cause system crashes [86, 91]. One of the drawbacks of using pre-emptive threads is its inherent non-determinism where most of the programmers' effort and time is spent. Lee [86] makes a strong case for building programs out of deterministic components and then introducing non-determinism where needed.

Hybrid programming models like MPI+OpenMP, which use pre-emptive multi-threading with MPI, introduce a number of challenges. Mixing models leads to contention of resources, where each runtime system may try to optimize the use of cores independently of each other [55]. A comparative performance study of OpenMP and MPI on a large variety of parallel architectures identifies lack of control over memory locality as one of the factors affecting OpenMP's performance [133]. OpenMP does not allow direct expression of data and work locality or specification of dependencies between tasks. The work in [119] points to the lack of transparency provided by OpenMP constructs and how a simple integration of those constructs in an MPI program does not yield good performance and often code restructuring may be required [118].

Hybrid programming models also impose many thread safety requirements on the MPI implementation and as shown in [13, 38, 59, 117, 140], there are several trade-offs between providing thread safety in an MPI implementation and the performance. The overhead of using locks to enable thread-safety is clearly evidenced in [13], which explores the use of coarse-grain and fine-grain locks in detail. Their experiments show the detrimental effects of resource contention on performance and point to the need for careful optimizations so that too much performance is not sacrificed. Moreover, according to the MPI standard, it is the user's responsibility to ensure that multiple threads in the same address space avoid race conditions and do not make conflicting MPI calls [59]. It is interesting to note that leading HPC systems limit the number of threads to one per core due to the overheads and program complexity introduced by the thread model [55]. The challenges and overheads of thread-safety of the MPI middleware are well known and it is an important problem but the use of coroutines circumvents the need for locks to support multitasking and the guaranteed atomicity makes it easier to reason about the state of the middleware.

The second major design decision was integration of FG-MPI directly into the MPI middleware rather than an attempt to design a new implementation of MPI or to use coroutines and layer it on top of MPI. Adaptive MPI is an implementation of MPI that supports fine-grain processes, however, AMPI [64] implements the MPI library on top of Charm++ [76] rather than directly into an existing MPI implementation. This requires their own implementation of MPI and the Charm++ runtime also needs a communication layer. This can result in an MPI sandwich, with MPI running on top of Charm++ which in turn runs over MPI. In FG-MPI, all MPI communication directly invokes the corresponding lower level MPI implementation of the call in the middleware, whereas in the layered approach only a subset of the MPI communication in the lowest layer is used. More importantly, a scheduler layered on top of MPI operates independently from the lower level MPI progress engine. The result is multiple independent control loops and schedulers, where it is difficult to coordinate their activities with regards to the scheduling of asynchronous and synchronous messages, which was a problem highlighted by Jacobson and Felderman in the design of the TCP stack [74].

We integrated FG-MPI into the MPI library by extending the MPICH2 middleware. FG-MPI was implemented with the view to enable technology transfer and from the onset we strongly believed that for our project to be successful and have an impact in the HPC community we needed to integrate into an existing open source implementation of MPI, rather than simply build a system from scratch. The MPI community follows the practise of implementing, testing and validating any extensions before accepting it into the standard.

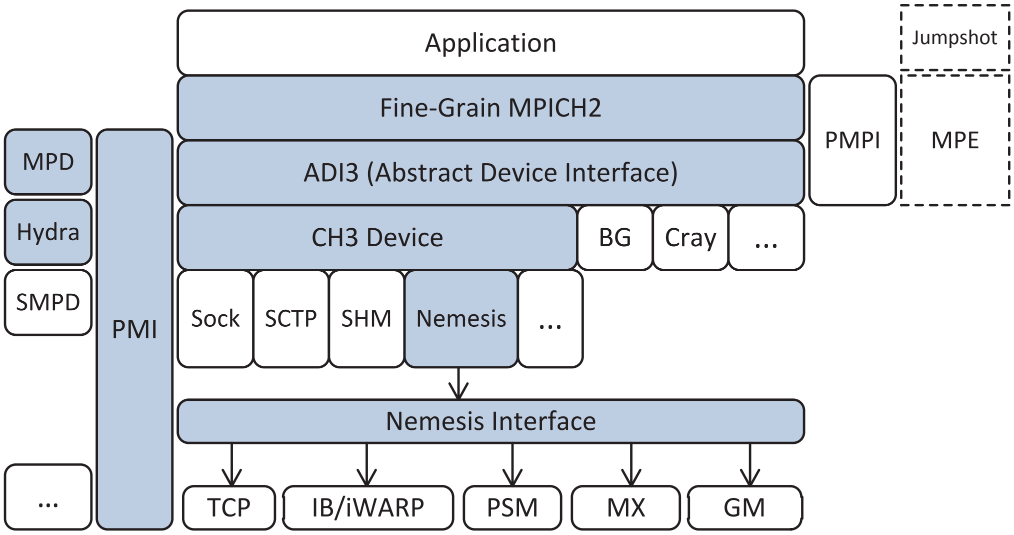


Figure 2.1: FG-MPI Architecture. Shading shows the layers of MPICH2 that were augmented in the FG-MPI implementation. Figure adapted from [10].

Integration into MPICH2 enabled us to build on a widely used, production quality implementation of the MPI standard and show that it can be used to improve the performance of existing MPI programs as well as exploit function-level parallelism. MPICH2 efficiently supports diverse computation and communication platforms ranging from commodity systems to high-end supercomputers. It is an optimized and well-engineered production system with over 530,000 lines of code and was awarded the R&D 100 award as a mark of excellence and innovation in 2005. The MPICH2 code forms the foundation of the majority of commercial and research implementations of MPI available today. Libraries such as IBM® Platform™MPI [72] for Blue Gene, Intel® MPI library [71], MPT Cray [32], MPICH-MX [101], and OSU MVAPICH [83] are among the several implementations that are derived from MPICH2 [10, 153].

Figure 2.1 shows the integration of FG-MPI in the layered modular architecture of MPICH2, with the shaded regions indicating the layers that FG-MPI augments. The first layer, below the application, defines the MPI API and implements user abstractions such as MPI data types and communicators. The second `ADI3` (Abstract Device Interface) layer contains the progress engine and provides abstract middleware services to support the functionality of the first layer. Representation in this layer is in terms of MPI requests/ messages and the functions for manipulating those requests. The third layer provides the device interface such as communication protocols and implements the ADI for the channels. FG-MPI uses the Nemesis `CH3` channel as the communication subsystem [21]. The Nemesis communication subsystem is designed for scalability and low shared-memory communication overhead, making it suitable for our fine-grain system. Communication among MPI processes in different OS-processes, on the same physical node, takes place through Nemesis's low latency, lock-free shared memory queues. The communication through shared memory employs optimizations to reduce L2 cache misses and techniques such as *fastboxes* that bypass message queues, to speed up message transfers [21]. Communication across different physical nodes is enabled through its multi-network support and integration allows FG-MPI to leverage MPICH2's rich support for network fabrics in cluster environments. As well, we exploit the locality of MPI processes in the system and implement optimized communication between concurrent processes in the same OS-process.

One of the main considerations in FG-MPI was to support large amounts of concurrency through scalable sharing of MPI middleware structures among the coroutines. To this end, a large number of MPI storage structures such as posted receive queues, unexpected messages queues, communicator and request pools are shared by the coroutines. Figure 2.2 shows a high-level picture of the FG-MPI runtime system and the shaded regions show the shared structures. Figure 2.2 contains a number of key components of the middleware and in the following sections we describe how FG-MPI augments the middleware to support large-scale concurrency with multiple MPI processes within an OS-process.

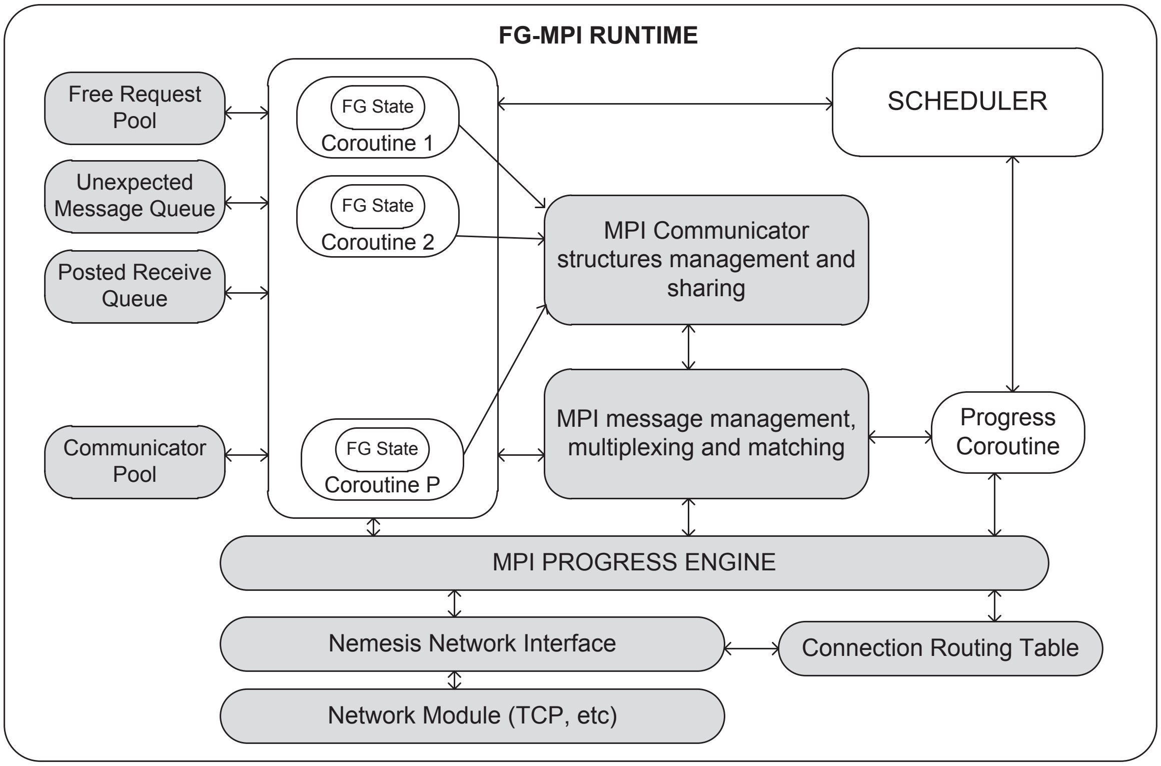


Figure 2.2: FG-MPI Runtime System. Shaded regions show the MPICH2 middleware structures shared among the co-located MPI processes.

2.1.1 Separation of Namespaces

FG-MPI decouples the MPI processes from the OS-processes, which requires separating the namespace of the OS process's network point of attachment from the namespace of the MPI process ranks. In this section we describe how connection information is stored in MPICH2 and our approach to decoupling the MPI process names from the connection routing information.

In MPI, communicators are used to define separate communication contexts where communication within one group of processes cannot interfere with another group. In order for a process to communicate with another it must provide a communication context (i.e., communicator) and the local rank of the process in that communicator. Each process also needs to maintain connection information about the other processes with which it communicates. The MPICH2 implementation stores the connection state information in a virtual connection (`VC`) object, and creates one `VC` for every MPI process. MPICH2 maintains a virtual connection reference table (`VCRT`) for each communicator and translates the communication context and the local rank of the process to the appropriate `VC` object. The `VCRT` is stored as a dense array of pointers to the `VC` objects and is indexed by the local process rank in the communicator. The process rank, in this case, is tightly associated with the connection routing information.

In FG-MPI, we decouple the MPI processes from the hardware and have taken a different approach than MPICH2 to storing process connections. The MPI process groups are not coupled to the `VCRT`, but instead we maintain two tables in each OS-process:

(a) a single `VCRT` for all the OS-processes in the execution environment that maps OS-processes to the virtual connections. The size of `VCRT` is proportional to the number of OS-processes.

(b) a process name table that uses `MPI_COMM_WORLD`[3,4] ranks and maps the MPI processes to OS-processes.

All the MPI processes co-located within an OS-process are assigned the same point of attachment. There is a single shared instance of each of these tables in every OS-process.

There remains the issue of efficiently storing the process group in each communicator. As the process group is decoupled from the `VCRT`, this allows us to employ a number of techniques and representations to reduce the amount of space required to store the process groups. We describe the implementation of these techniques in FG-MPI in Sections 2.1.3 and 3.1. The ability to represent communicators more efficiently by decoupling the `VCRT`

---

[3] `MPI_COMM_WORLD` is a pre-defined communicator in the MPI standard, which represents the group of all MPI processes

[4] We do not support MPI dynamic process management functionality.

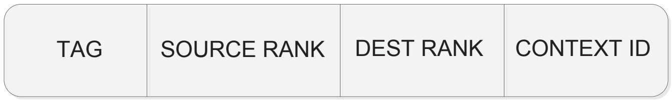


Figure 2.3: Structure of the message envelope.

in MPICH2 from the upper level code was also proposed in [47], which appeared after our initial work.

The separation of the two namespaces requires that we have a two-level hierarchy of ranks: (a) an OS-process namespace for virtual connection management and routing and (b) a namespace for MPI processes in `MPI_COMM_WORLD`. We emphasize the separation of these two namespaces because it is an example of the importance of naming in a distributed system [125]. For example in the Internet the tight association between the name of a node (hostname) and its network attachment point (IP address) was identified as a disadvantage by Saltzer and others [61, 124].

Although we have not yet considered process mobility, the ability to support additional concurrency that is decoupled from the hardware should simplify that as well. We have experimented with new ways of supporting dynamic processes by introducing a pool of co-located processes that stay dormant until one or more processes are activated by sending a message to them. This creates processes that can be scheduled “on demand” without incurring the overhead of spawning OS-processes [28]. The idea is for the processes to execute a task and then be suspended again upon its completion until the next activation. This scheme is enabled by dis-associating the structure of the program from the OS-processes and allowing additional concurrency to be managed by FG-MPI’s runtime scheduler.

2.1.2 Message Multiplexing

An interesting issue related to separation of namespaces is the message match header (envelope) in MPICH2. This header (see Figure 2.3) is appended to each message that is communicated between processes and contains the MPI tag, rank of the sender (source) process and the context ID of the communicator. In MPICH2, the message envelope does not contain the rank of the receiver (destination) process because that information is implicit from the OS-process identifier and hence the corresponding virtual connection used for message transmission. In FG-MPI, since there may be multiple MPI processes inside an OS-process, the destination rank of the process is necessary to de-multiplex the message from the OS-process network point of attachment to the MPI process. As a result we had to extend the message envelope as well as increase the packet header size to include the destination rank. This destination rank is the rank of the receiver in `MPI_COMM_WORLD` and uniquely identifies it.

A second issue is that the MPICH2 version we used as our code base used `int16_t` for storing the rank. This was not sufficient for supporting an environment with millions of MPI processes. We use 32-bit integers for both the source and destination ranks. There are trade-offs in extending the message envelope. MPICH2's motivation for using 16-bit ranks was to fit the entire message header in a 64-bit field to allow 64-bit instruction comparisons on platforms that support it and also slow communication links can benefit from a smaller header size [14]. Although we have not done a low-level comparison we have not noticed any performance differences at the application layer as a result of our extension.

2.1.3 Scalability of Communicators

Communicators in MPI provide an important mechanism to support code composition and enable development of parallel libraries [62]. Communicators provide a scoping mechanism that ensures that communication within one software component will not interfere with another component. FG-MPI's ability to expose large-scale concurrency allows for more opportunities to structure and group processes, which makes it more important to support efficient creation and storage of communicators. In this section we give a high-level overview

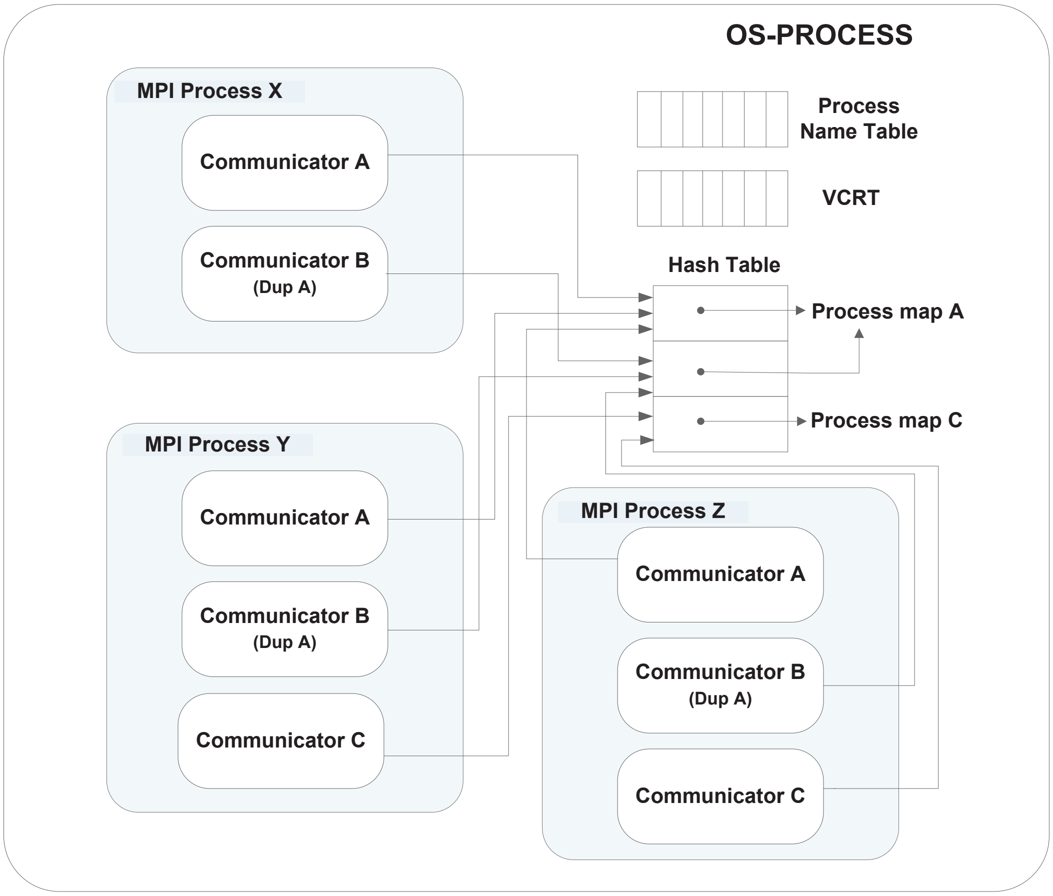


Figure 2.4: Sharing of process group map inside communicators among co-located MPI processes.

of our group sharing and communicator creation techniques and in Section 3.1 we describe the design of the different components in detail.

In MPI, all processes belong to groups and a communicator encapsulates the communication context and the process group in one object. As mentioned in Section 2.1.1, the MPI processes in FG-MPI are not coupled to the `VCRT` and we store a process group map in each communicator that maps the local rank of that process in that communicator to its rank in `MPI_COMM_WORLD`. In order for FG-MPI to expose large-scale concurrency the creation of communicators and storage of the process maps must be scalable and space efficient. A simple calculation shows that keeping separate process maps as arrays for P processes takes $\mathcal{O}(P^2)$ space, which is not feasible in a system with millions of MPI processes. We employ three techniques for communicator creation and storage that is both time and space efficient: (1) We enable sharing of process maps among co-located processes that are part of the same communicator. (2) We use different memory reduction techniques for storage of process maps and provide a framework that allows selection of different storage structures as a configurable option. (3) We define new efficient algorithms for communicator creation `MPI_Comm_split` and `MPI_Comm_create` and an algorithm for creation of a globally unique context ID. The process map for `MPI_COMM_WORLD` is simply an identity vector and stored as a function.

Figure 2.4 shows an example of three co-located processes which share the maps of different communicators. During the communicator creation operation, one of the members of the communicator creates the process map and stores a pointer to it in a global hash table. The context ID of each communicator, as described in Section 3.1.1, is globally unique in our implementation and is used as the key for the hash table. Other co-located processes of the same communicator lookup the pointer to this process map in the hash table and cache it locally in their communicator structures. We thus store a single process map per OS-process for each communicator. Sharing is also enabled if a routine like `MPI_Comm_dup` is used to duplicate any communicator. In Figure 2.4, communicator `B` is created by duplicating communicator `A`. The entry in the hash table corresponds to the new context ID for communicator `B`, however, the process map is shared. Notice that the hash table lookup is only done once to access the pointer to the process map, during communicator creation, and then the pointer is cached locally. This allows us to use reference counting to keep track of how many MPI processes are sharing a map. If communicator `A` is freed, the entry corresponding to it in the hash table is removed, however, process map `A` remains until all references to it are removed.

MPI provides routines like `MPI_Comm_group` to access the process group associated with a communicator. We use a uniform definition for process maps inside MPI group and communicator structures. For certain routines like `MPI_Comm_group`, this allows us to use reference counting, in a way similar to that described for the `MPI_Comm_dup` routine, to share the process maps across communicators and associated MPI groups. In general, creation of MPI groups from existing groups through routines like `MPI_Group_incl` are not scalable, as these are local operations and store their individual maps. However, if a group is used to create a new communicator as in `MPI_Comm_create`, then we de-allocate all of the individual maps of the group members that are co-located and share a single map with the new communicator. However, as mentioned in [52], routines to create new groups from existing ones are rarely needed, and the use of `MPI_Comm_split` is recommended for creation of communicators. We discuss low-level system details of the design and implementation of communicators in Chapter 3.

2.1.4 MPI Environment Initialization and Synchronization

MPICH2 uses external agents called process managers to launch and manage parallel jobs. These agents communicate with MPI processes through an interface called PMI (Process Manager Interface) via the `mpiexec` command. We extended the PMI to support an `nfg` (**n**umber of **f**ine-**g**rain) flag to the `mpiexec` command. Using `nfg`, the user can choose how many MPI processes to run per OS-process in combination with the `n` flag specifying the number of OS-processes. The FG-MPI runtime system inside each OS-process is initialized through a call to a function called `FGmpiexec()`. At the beginning of the program execution there exists a single main coroutine (`MAIN_CO`), which communicates with the process manager and gathers the environment settings. The main coroutine plays an additional initialization and synchronization role in the `MPI_Init()` and `MPI_Finalize()` calls, but otherwise behaves identically to the other co-located coroutines during program execution.

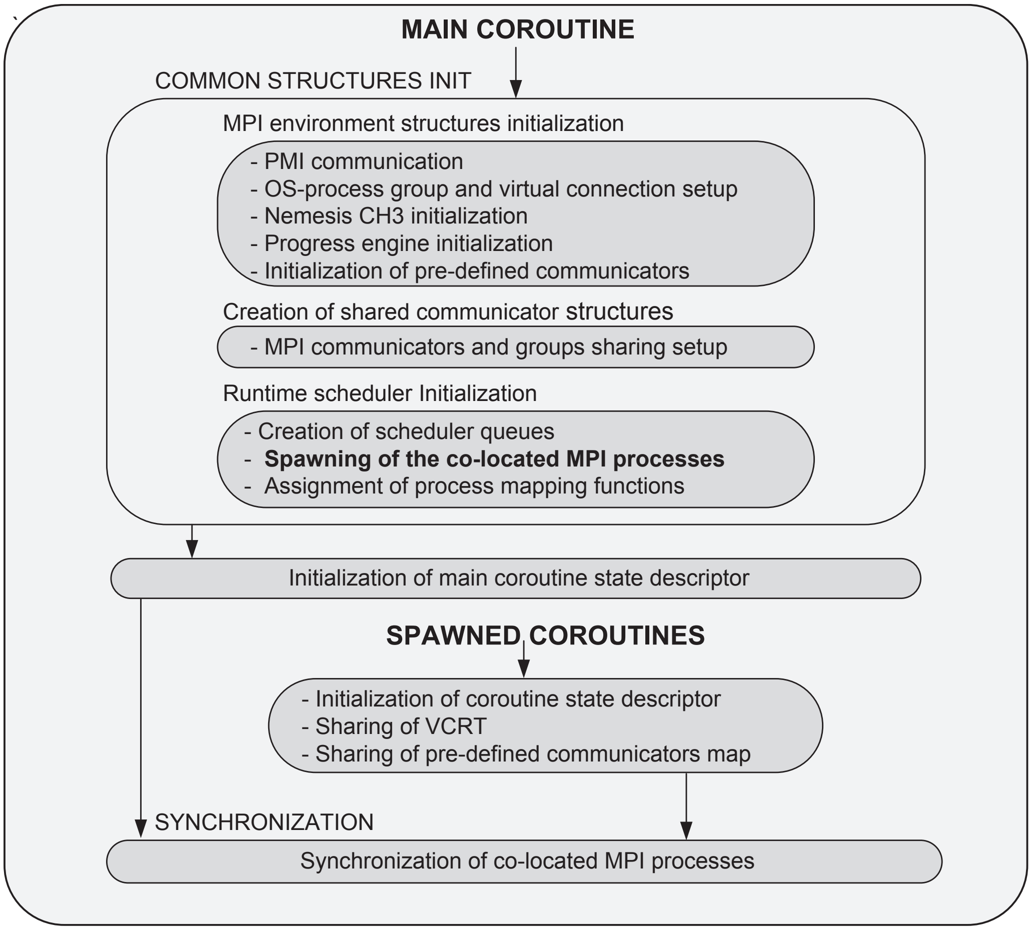

MAIN COROUTINE
COMMON STRUCTURES INIT
MPI environment structures initialization
- PMI communication
- OS-process group and virtual connection setup
- Nemesis CH3 initialization
- Progress engine initialization
- Initialization of pre-defined communicators
Creation of shared communicator structures
- MPI communicators and groups sharing setup
Runtime scheduler Initialization
- Creation of scheduler queues
- Spawning of the co-located MPI processes
- Assignment of process mapping functions
Initialization of main coroutine state descriptor
SPAWNED COROUTINES
- Initialization of coroutine state descriptor
- Sharing of VCRT
- Sharing of pre-defined communicators map
SYNCHRONIZATION
Synchronization of co-located MPI processes

Figure 2.5: Initialization of the MPI environment during `MPI_Init`.

When the `FGmpiexec()` function is called by the main coroutine, it performs two functions: firstly, it initializes the MPI execution environment and secondly it spawns the other coroutines.

Figure 2.5 shows the major structures that are created and initialized by the main coroutine and subsequently shared among all the co-located MPI processes in an OS-process. These include the Nemesis communication subsystem queues and shared memory segments, progress engine initialization and creation of `VCRT` and communicator hash table. The MPI storage structures such as posted receive queues, unexpected messages queues and request pools are also global in the middleware and shared among the coroutines. Sharing of the request queues and progress engine not only enables scaling to large number of co-located processes, it also allows one MPI process to cooperatively progress messages for other co-located MPI processes. We describe message progression in more detail in Section 2.2.

The last step of the MPI initialization is the creation of scheduler queues and the spawning of the co-located MPI processes in each OS-process. Each of the newly spawned processes are assigned the function that they will be executing. These functions are written as standard MPI programs, complete with the `MPI_Init` and `MPI_Finalize` routines. The mapping of functions to the MPI processes is done through a user-defined binding function[5], which maps the `MPI_COMM_WORLD` rank of each process to a function pointer. The main coroutine yields at this point and each of the spawned processes are run by the scheduler and they initialize their coroutine state (described in Section 2.1.5) and share the common structures created by the main coroutine. All co-located MPI processes synchronize at the end of their `MPI_Init` calls and are queued for scheduling.

2.1.5 Coroutine State Descriptor

Each coroutine maintains a descriptor to store the state of execution of the MPI process associated with it. This MPI state information per coroutine is 1,328 bytes on a 64-bit machine and consists of the fields shown in Figure 2.6. This consists of the process's unique

Process rank in MPI_COMM_WORLD
Pointer to built-in communicators
Pointer to communicator context ID bitmap
State of initialization

Figure 2.6: Coroutine State Descriptor.

---

[5] The binding function is passed as an argument to `FGmpiexec`. More details are given in Section 3.4.

rank in `MPI_COMM_WORLD`, a pointer to the pre-defined built-in communicators, a pointer to its context ID bitmap indicating the available and used communicator context IDs for this process (see Section 3.1.1) and the state of initialization, i.e., whether it has called `MPI_Init` (and similarly for `MPI_Finalize`). Each coroutine has its own stack with a default size of 128 Kbytes. The lower bound on the stack size for the Toernig's and the PCL coroutine libraries is 1K bytes and 4K bytes, respectively. We currently have a fixed stack size for each coroutine, however, it is possible to extend `FGmpiexec` to provide the stack size as an argument.

Finally, note that FG-MPI extends MPICH2 and the FG-MPI runtime is only set-up when there is more than one MPI process in an OS-process and it is possible to freely mix OS-processes with one process with those having multiple processes. In Section 2.2 we describe FG-MPI's user-level scheduler and its interactions with the middleware's progress engine.

2.2 Integrated MPI Scheduler

We maintain a run queue and a blocked queue for co-located MPI processes inside each OS-process. Scheduling events inside the middleware invoke the scheduler, which according to the scheduling policy, blocks the current process or adds it back onto the run queue, and chooses the next process to resume. We provide a scheduler framework that allows us to add new policies as the need may arise. The selection of the scheduler is provided as a command line option to `mpiexec`. The most interesting aspect of the scheduler is its integration into the MPI middleware and interaction with events occurring inside the progress engine.

As Figure 2.2 (page 22) shows, many of the key data structures in the middleware, such as the message queues, the request pools and the communicator pool, are shared among all of the co-located MPI processes. In FG-MPI, communication can be both internal (among co-located processes) and external (between processes that are not co-located). When a process makes an MPI call it progresses its request as far as possible. For example, consider the case of standard communication through `MPI_Send` and `MPI_Recv` between co-located processes. If the receiver process runs first, it queues its request in the posted receive queue and yields to the scheduler, which blocks it and resumes another co-located process to run. When the corresponding sender process runs, it matches the pre-posted receive request and completes its call. The sender sends a notification to the scheduler to unblock the receiver process and continues executing. However, in case the matching receive has not yet been posted, the send request is placed in the unexpected message queue and the sender yields to the scheduler so that the receiver can run. When the corresponding receiver executes, it finds and completes the matching request and continues execution.

For the case of communication between processes that are not co-located, when a process sends a message it initiates a communication transfer over the external link to the receiver. Depending on the size of the send request, it may be able to complete the transfer and continue executing or it may require an acknowledgement from the receiver to complete the call, as for example in a long message rendezvous transfer. In the latter case, the sender's message is queued among the pending sends in the progress engine, the sender yields to the scheduler and another co-located process runs. Similarly, a message arriving over the network at the message matching layer may complete a pending request or in the case of an unexpected message will be queued until a matching receive request arrives.

In case of non-blocking MPI calls, the process does not block but continues executing until, for example, the corresponding wait routine is called. Depending on the state of the process's request at that point it may be able to complete the call or yield to another co-located process.

One important advantage to sharing the state of the progress engine is that MPI processes can cooperatively progress pending messages for other co-located processes and notify the scheduler. The scheduler based on these notifications can add processes to the run queue. An example of cooperation between co-located processes is that of a pre-posted receive request for which a ready-to-send (RTS) arrives to initiate the long message handshake. It is possible that the MPI process which posted that receive request is not currently executing, but a clear-to-send (CTS) can be sent by the currently executing process on its behalf.

One effect of non-preemptive scheduling is that a process that is busy computing blocks the progress of all other co-located processes. One assumes that as long as the process is busy it is making progress; however, we added `MPIX_Yield()` to handle cases when a fairer scheduling is needed. `MPIX_Yield()` is a FG-MPI specific routine which allows the calling process to voluntarily yield control to the scheduler.

Internal communication is optimized to take advantage of a single address space, and it is an opportunity for the scheduler, depending on the type of the communication, to block one process until the communication can be completed after which both processes can proceed. For co-located processes, the scheduler follows a natural order where a send message schedules the corresponding receive process that can continue to progress the message chain. The communication among co-located processes involves a single `memcpy`, avoiding any intermediate system copies. Similarly for external events, once a message is received and completed the corresponding MPI process is scheduled to continue advancing the computation. As well, for collectives such as `MPI_Barrier`, the last co-located process completing the barrier can gang-schedule all of the processes in the barrier since they can now all proceed.

In many cases we have found that even the basic round-robin (RR) scheduler, which keeps all the processes on the run queue, is adequate. Because the scheduling overhead is relatively small, the RR scheduler works well as long as the co-located processes are easy to keep busy. Another nice property of the RR scheduler is that it is deterministic and gives more predictable executions. The deterministic property of RR has also been useful as a tool for debugging programs. This is the advantage to introducing a user-level scheduler instead of scheduling by the OS where the programmer has less control over how the processes are scheduled.

It is not sufficient to have only RR since there are simple cases where RR performs extremely poorly. For example, consider the simple ring program, the forward communication of messages works well when it is the same order as the scheduling order, however, communication in the reverse direction is slow due to re-scheduling delay of all of the processes on the run queue. More generally, it was important to introduce a scheduling framework rather than one or more fixed policies. The policy ultimately depends on the application where ideally processes on the critical execution path are scheduled first. Finally, note that the scheduling policy is local to an OS-process and the runtime inside each OS-process can select its own scheduling policy.

One interesting problem that arises with the scheduler, that allows blocking of MPI processes, is indefinite waiting of the processes in the scheduler's blocked queue. Indefinite blocking can occur, for instance, when all of the co-located processes are blocked on a receive call, waiting for an external event, and there is no runnable process that can check for the arrival of messages and unblock those processes. One alternative is simply not to block all processes or to simply keep one or more processes on the queue. Deciding on whether or not to block a process depending on the state of other co-located processes is complicated. There are a large number of MPI calls and different scenarios that would need to be considered including analysis of corner cases involving collectives and the different MPI communication modes, where a change in the implementation could inadvertently introduce subtle problems. However, there is a simple and scalable solution to this scheduling problem.

We solved the potential indefinite blocking problem by introducing a progress coroutine in our runtime that comes into existence the first time an MPI process blocks on a receive call. Once created, the progress coroutine remains on the run queue. When called, this coroutine executes the progress-loop in the middleware and progresses pending incoming and outgoing messages. Whenever there is a receive that could be matched by a message from a remote process it ensures that we poll the external link for more data and on arrival of such a message wakes up the blocked process. As discussed, a clear-to-send (CTS) may be sent by the progress coroutine for a pre-posted receive. A progress coroutine avoids the checking that would have been necessary when blocking processes and also provides an easy way to measure the idle time and "slackness" during runtime.

2.3 Summary

FG-MPI extends the execution model of MPI to include interleaved concurrency through integration into the MPICH2 middleware. It decouples the notion of a process from that of the hardware and makes it possible to adjust the granularity of the programs independently from the hardware. Integration of the FG-MPI runtime into the MPI middleware reduces the overhead of adding concurrency and provides an alternative to the hybrid MPI+X model.

FG-MPI's tight integration into MPICH2 enables sharing of the middleware structures, optimizing communication for co-located processes and implementation of a runtime scheduler that works in concert with the events occurring inside the middleware. These benefits cannot easily be obtained by layering a system on top of MPI with no visibility into the state of the middleware. FG-MPI runs on commodity operating systems and does not require any special support.

Chapter 3

System Components

> The best design tool is a long eraser with a pencil at one end.
>
> Marty Neumeier

In this chapter we describe the low-level system and implementation details of the FG-MPI system. Prior to discussing the design aspects of the different system components, we introduce the following notation to classify different sizes of systems in terms of the communication hierarchy (see Figure 3.1). We specify the hierarchical structure of an MPI execution in terms of P, the number of MPI processes per OS-process as given by `nfg`, O, the number of OS-processes per machine, and M, the number of machines. N = P×O×M is the number of MPI processes in a [P;O;M] execution. The standard "one MPI process per OS-process" model corresponds to a [1;O;M] execution. In general, the number of fine-grain processes (P) inside an OS-process can vary, but for our discussion we will assume the same P for every O. Given this notation, concurrency can be added by over-subscription (increasing O to be larger than the number of cores per machine) and/or increasing P.

3.1 Scalability of Communicators and Groups

The ability of FG-MPI to expose large-scale concurrency necessitates addressing issues related to the scalability of MPI to a system with millions of fine-grain MPI processes. Communicators and groups are an integral part of MPI and scalability of their implementa-

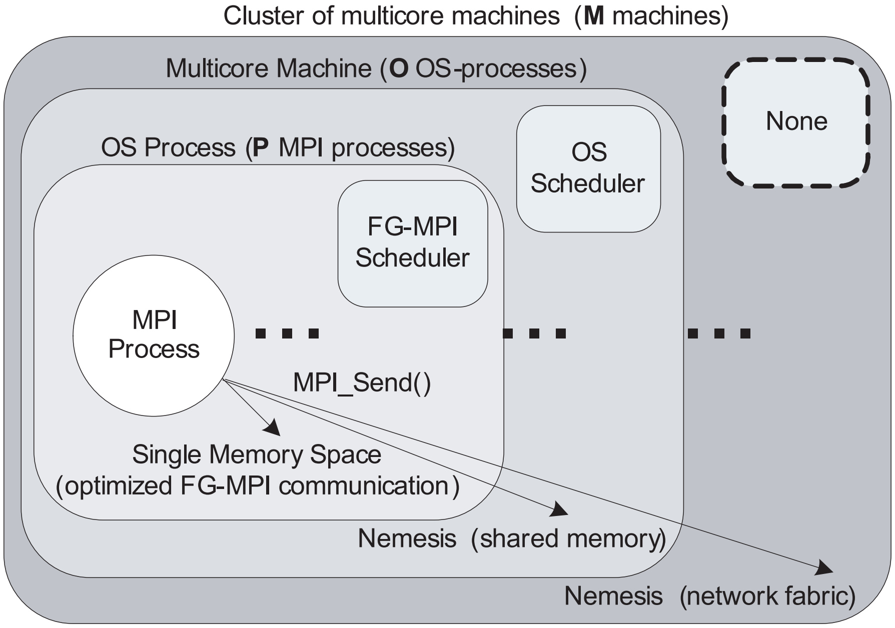

Cluster of multicore machines (**M** machines)
Multicore Machine (**O** OS-processes)
None
OS Process (**P** MPI processes)
OS Scheduler
FG-MPI Scheduler
MPI Process
MPI_Send()
Single Memory Space
(optimized FG-MPI communication)
Nemesis (shared memory)
Nemesis (network fabric)

Figure 3.1: Hierarchical view of FG-MPI runtime environment with N = P×O×M MPI processes.

tion is a challenging aspect of the MPI middleware. In this section, we describe techniques to allow for sharing of group information inside OS-processes and the design of scalable operations to create the communicators. The techniques described here were implemented for intra-communicators, however, they can be extended to inter-communicators.

Communicators and groups exist to support the development of higher level libraries [56, 62]. A communicator is an opaque object and every point-to-point and collective communication routine in MPI takes a communicator as a parameter. Communicators divide the communication into disjoint communication contexts such that a message sent in one context can only be received by a communication routine with a communicator in the same context. Every process inside a communicator is assigned a rank from 0 to the group size minus one. In particular, since all new communicators are derived from the pre-defined communicator `MPI_COMM_WORLD`, the group of all processes, every process in a system with

N processes is assigned a "world" rank from 0 to $N-1$.

In order to route messages to their destination a communicator needs to know the location of the destination process. In a $[1, O, M]$ system the endpoint of a communication is an OS-process and, as done in MPICH2, one can store the information about the destination as an array of pointers mapping group rank to a communication endpoint object. In FG-MPI, since the communication endpoints are no longer OS-processes but MPI processes inside OS-processes, we use a process's world rank to uniquely identify the destination of a message. When a message is sent in FG-MPI we add the world rank of the destination process and the Nemesis endpoint associated with the OS-process to the message envelope. Using the endpoint object, Nemesis routes the message to the correct OS-process and the middleware inside the OS-process uses the world rank as part of the MPI matching criteria to deliver the message to the correct receive buffer. Most implementations of MPI store the mapping from group rank to destination process as an array for every destination [29, 47, 146], thus a group of size $\mathcal{O}(N)$ requires $\mathcal{O}(N^2)$ space in total. Given that in general MPI programs may contain many groups of various sizes the amount of space consumed by group maps becomes prohibitive. Given $N = P \times O \times M$ processes in a $[P, O, M]$ system, one goal was to reduce the non-scalable $\mathcal{O}(N^2)$ space and communication time to $\mathcal{O}(PO^2M^2)$. Note that here we use P as the maximum number of MPI processes per OS-process and O as the maximum number of OS-processes per machine, to give upper bounds on the time and space with respect to the total number of processes.

The space consumed by the group maps becomes all the more challenging as one attempts to scale up the number of MPI processes to millions. For example, consider a $[1000, 100, 10]$ system with one million processes. Assuming that we can optimally encode each destination using $\lceil \lg(10^6) \rceil = 20$ bits, a new communicator which re-maps `MPI_COMM_WORLD` requires 2.5 MBytes of storage per process, 2.5 GBytes per OS-process and 250 GBytes per machine and 2.5 TBytes in total! The techniques described in this section reduce the memory requirements for this example to 2.5 GBytes in total, and possibly significantly less depending on the mapping.

It is reasonable to expect that the size of systems will continue to grow and systems with more than 1,500,000 cores already exist today [144]. Issues related to the scalability of the MPI specification and its implementation to millions of MPI processes are a subject of active research [12, 47]. At this scale, and even on many smaller systems, memory saving techniques are essential to take advantage of the benefits of communicators. Although we do not have access to a system of this size, we are able to implement and evaluate a number of memory saving techniques for communicators inside FG-MPI for $[P, O, M]$ systems of this scale.

The first step to reduce the space requirements for communicators is to share the group map among co-located processes. This allows us to reduce the space requirements from $\mathcal{O}(N^2)$ to $\mathcal{O}(PO^2M^2)$. In Section 3.1.1, we discuss the issues and optimizations involved in sharing of data structures among processes. The second strategy, which benefits all $[P, O, M]$ systems, is to reduce the N factor through different memory reduction techniques for storage of the group map. These memory reduction techniques were implemented as part of a Master's thesis by Seyed Mirtaheri [98] under the supervision of Dr. Alan Wagner and involved introduction of a framework for the use of various compact representations of the group map. A key element of this framework is the decomposition of the group map into a set and permutation. The representations investigated included compression, implicit representations based on Binary Decision Diagrams (BDDs) [20], and succinct data structures. The details of these memory reduction techniques can be found in [78]. Mirtaheri's work was implemented as an external library that can be linked to the FG-MPI system as a configuration option. I provided the API for this library and implemented the interface part for FG-MPI as well as the build and configuration systems.

3.1.1 Sharing the Group Map

The two main routines for creating intra-communicators in MPI are `MPI_Comm_split` and `MPI_Comm_create`. Communicators are created with respect to an outer communicator, where the new communicator or communicators is a subset of the processes of the outer communicator. These routines are collective operations over the processes in the outer communicator. The routines return with a local handle to the new communicator, which contains among other fields, (a) a pointer to the process group map, and (b) a context identifier. The group map identifies the members of the group associated with the communicator and the context identifier (context ID) is a fixed sized field inside the communicator that must match on all communications using the communicator. As long as the context ID of a communicator for a process is unique, the process cannot mistakenly receive a message sent to it using a different communicator. Thus context IDs provide a scoping mechanism and are essential to avoiding communication errors that can arise in writing of parallel libraries in MPI.

To enable sharing the group map structure among the co-located processes belonging to the new communicator, a hash table is used to coordinate the allocation of memory for the structure. Consider Figure 3.2 showing two OS-processes X and Y inside one machine where processes $\{2, 3, 4, 5\}$ are creating communicator A. One of the processes inside each OS-process creates the group map structure and stores a pointer to the structure inside the global hash table. Subsequently, the other co-located process belonging to the same communicator uses this hash table to lookup the group map pointer and cache the pointer inside its own communicator structure during the communicator creation operation. For sending messages, a process accesses the group map to find the world rank of the destination to put in the message envelope and then finds the OS-process endpoint through the process name table to route the message to the appropriate OS-process.

An important precondition to the use of the hash table is that all co-located processes

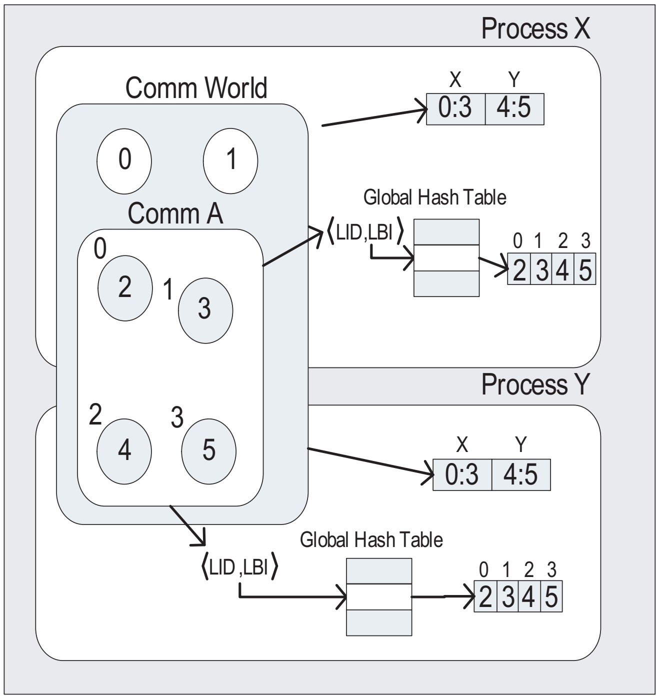


Figure 3.2: The sharing of the process group inside communicator A with 6 processes inside two OS-processes X and Y of one machine M.

belonging to the same new communicator need to have the same key to lookup the shared group map pointer in the global hash table. Furthermore, these keys need to be different for different communicators. Given that a context ID has to be created as part of the communicator and the properties of a context ID are similar to that of a key, we chose to generate a context ID that can serve both purposes.

During a communicator create routine, the context ID for a new communicator A is constructed from a combination of a leader identifier (LID) and leader bit index (LBI) denoted by $\langle LID, LBI\rangle$ where

- LID is the world rank of a representative chosen from the group underlying A, and
- LBI is an integer, uniquely chosen for every LID.

For LBI, every process maintains a bitmap of size 2^k, for some constant k, where a set bit in location i of the bitmap signifies the use of that position, the bit is unset otherwise. For a given LID, LBI is the index of a free position in the bitmap which is then set to one. The context ID is constructed by concatenating the L bits needed to represent a world rank and k bits[6] needed to represent LBI, where each process can be chosen leader for as many as 2^k communicators. Context ID $\langle LID, LBI \rangle$ is globally unique since LID and LBI are equal only when they belong to the same communicator. It is thus also OS-process unique and therefore can be used as a key for sharing the group map for the communicator.

The pseudo-code for `MPI_Comm_split` is given in Procedure 1. The routine uses a leader-based approach where one process (the root) in the outer communicator gathers and then distributes the necessary information to all the other processes. In line 5, the color and key information is gathered to the root. We also gather the bitmap information into B, which is used later in Line 11. We employ an optimization that reduces the amount of data sent to the root process by sending only the count of set bits and the location of the lowest set bit in LBI, instead of sending the entire bitmap. The gathered data, together with ranks of the processes stored in S, is sorted by color, key and rank. Processes belonging to the same color class belong to the same new communicator. In lines 5 to 20, the root computes the context ID. Although any bitmap with an unset bit can be chosen, we balance the choice of leader by finding the one that has been chosen the least number of times. We use the CID value to also define a local leader for each group. The root uses the world rank array to determine which processes are co-located and for each group, and for each OS-process it chooses a local leader. Finally, in lines 22 to 32, the root distributes the context ID and group information to all the processes. All processes receive the context ID, but since the group is shared, the root distributes the group map only to the local leaders as identified in the context ID information. The remaining processes yield and are not re-scheduled until after the local leader has created the group map. For an outer communicator (`comm`) of size

---

[6] See implementation notes in Section 3.1.2 for details on the value of k.

Procedure 1 `MPI_Comm_split(comm,color,key,ncomm)`

```
Let: Let $N$ be the size of comm and rank 0 the root.
if root then
    // Arrays to store Colors, Keys and Bitmaps
    Allocate $C[N]$, $K[N]$, and $B[N]$
end if
MPI_Gather($[C, K, B], root$)
if root then
    Allocate $S[N]$ // initially S[i]=i
    Sort $S, C, K, B$ with respect to $C$, $K$ and rank.
    for each color class $C_i$ do
        Find $k$ in $B$ with the fewest number of bits set
        $LBI \leftarrow$ first unset bit in $B[k]$
        Allocate $CID[N]$ // info for context ID
        for each $j$ in the color class $C_i$ do
            if $S[k]$ is the first co-located process for group then
                $CID[j] \leftarrow \{S[k], LBI, LEADER = yes\}$
            else
                $CID[j] \leftarrow \{S[k], LBI, LEADER = no\}$
            end if
        end for
    end for
end if
MPI_Scatter({$CID$,group-rank,group-size,local-leader},$root$)
Create $\langle LID, LBI\rangle$ from CID values
if root then
    for each color class $C_i$ do
        MPI_Send(group membership info to local leaders)
    end for
else if local leader of a group for an OS-process then
    MPI_Recv(group membership info from root)
else
    yield to wait for local leader
end if
Use $\langle LID, LBI\rangle$ to create or get pointer to group map
```

N, it takes $\mathcal{O}(N \lg N)$ time to sort the data at the root, $\mathcal{O}(N)$ communications, and $\mathcal{O}(N)$ space.

The pseudo-code for `MPI_Comm_create` is given in Procedure 2. The main difference between the `MPI_Comm_split` and `MPI_Comm_create` routines is that in `MPI_Comm_create` each process already knows the group information. As well, since MPI 2.2, the group parameter to `MPI_Comm_create` can differ, which can result in multiple communicators. Because processes already have the group information, the only information missing is the context ID. In lines 1 to 8, rank 0 of the group gathers the bitmap information from the rest of the group members. Rank 0 computes $\langle LID, LBI\rangle$, and in lines 9 to 17 sends the context

Procedure 2 `MPI_Comm_create(comm,group,ncomm)`

Require: Let G be group map from parameter `group` and N be the size of `comm`.

```
 1: if I am rank 0 of $G$ then
 2:    Allocate $B[|G|]$ where $B[0] \leftarrow$ bitmap of rank 0
 3:    for each $i = 1$ to $|G| - 1$ do
 4:       MPI_Recv(B[i], from rank $i$ in $G$)
 5:    end for
 6: else
 7:    MPI_Send(bitmap, to rank 0 in $G$)
 8: end if
 9: if I am rank 0 of $G$ then
10:    Find $k$ in $B$ with the fewest number of bits set
11:    $LBI \leftarrow$ first unset bit in $B[k]$, $LID \leftarrow k$
12:    for each $i = 1$ to $|G| - 1$ do
13:       MPI_Send({$\langle LID, LBI\rangle$, group rank}, rank $i$ in $G$)
14:    end for
15: else
16:    MPI_Recv({$\langle LID, LBI\rangle$, group rank}, rank 0 in $G$)
17: end if
18: Use $\langle LID, LBI\rangle$ to create or get pointer to group map.
```

ID information back to the group members. Because there can be more than one group, we cannot use collectives, and instead use explicit sends and receives to the group elements. For multiple groups, all of the groups can be actively sending and receiving messages at the same time since the groups are non-overlapping. In line 18 the process with the smallest group rank uses it to create the group map and the remaining processes simply cache a pointer to their group map.

3.1.2 Implementation Notes

In our implementation, all the fields of the message envelope (see Figure 2.3, page 24) are stored as 32-bit integers. This allows the envelope to fit in two 64-bit fields, potentially allowing for fast comparisons on 64-bit architectures. For the 32-bit context ID, we set $k = 6$ and $L = 21$, because MPICH2 reserves 5 bits in the context ID for other purposes [8]. The choice of k=6 bits keeps the size of the bitmap small, allows each process to be a chosen as a leader 64 times, while leaving 21 bits for the leader ID. This choice has no hard dependencies for the system design, which allows for increasing the size of context ID in the future for very large program sizes.

The MPICH2 middleware uses a lazy instantiation approach for storing handles to the communicator objects in the middleware.[7] This allows the middleware to specify a large pool of objects, however, memory is only allocated when needed. It uses a two-level storage scheme consisting of: (a) a direct array that can hold a small number of pre-allocated objects, and (b) a large indirect array of pointers, each of which points to a block of storage. These blocks are lazily allocated when they are needed. The default value of number of pointers in the indirect array in MPICH2 is 1024 and each of the blocks can contain 256 objects, allowing for allocation of 256K communicator handles in each OS-process. This communicator pool is shared among the co-located MPI processes (Figure 2.2,

---

[7] MPICH2 uses the same storage scheme for handles to other types of MPI objects as well, such as group, request and datatype etc.

page 22). The number of communicators (num_comms) that can be created inside an OS-process, before exhausting this pool, is given by ($\sum_{i=1}^{num\_comms} num\_colocated_i = 256K$), where $num\_colocated_i$ is the number of processes in $communicator\_i$ that are co-located inside that OS-process. The usage of the communicator pool depends on the dynamic characteristics of the application. We currently use MPICH2's default configuration values for the indirect array size in FG-MPI which can be increased, if needed.

3.1.3 Discussion

Assignment of a unique identifier to the process groups in MPI communicators can be viewed as a hypergraph problem, where the design of the context ID algorithm determines the graph structures that can be created. In this section we discuss the different graph structures used by our context ID generation algorithm and compare it to MPICH2's algorithm.

In an MPI environment, the set of vertices V in a hypergraph $H(V, E)$ represent the set of MPI processes in `MPI_COMM_WORLD` and the set of edges E, such that $E \subseteq \mathcal{P}(V)$, represent the process groups.[8] Our context ID creation algorithm, where each edge $e \in E$ in a hypergraph can be a leader at most 2^k times, is equivalent to the problem of s-orientability. S-orientability of a hypergraph is defined as follows [43].

Definition 3.1. A hypergraph $H = (V, E)$ is called s-orientable, if there is an assignment of each edge $e \in E$ to one of its vertices $v \in e$ such that no vertex is assigned more than s edges.

Our algorithm balances the choice of leader by finding the one that has been previously selected as the leader the least number of times. Ties are broken by selecting the leader with the smallest local rank in the group. The structure of the hypergraph is dynamically determined based on the program execution and our algorithm uses a greedy approach that finds the local minimum.

---

[8] The empty set can be viewed as corresponding to `MPI_COMM_NULL`, the null communicator.

In MPICH2, the context ID is directly generated from a bitmap of size 2^k that is allocated for each process at start-up with all bits initially set to one.[9] A new context ID is generated by performing a bitwise `AND` in `MPI_Allreduce`, where all processes in the outer communicator of the operation obtain the result of the bitwise `AND` operation where a one bit in the i^{th} position is an available context ID. As long as all processes agree on which bit to choose, then they will all agree on the same k-bit value to use to generate the context ID (the lowest set bit is selected in MPICH2 [59]). The MPICH2 algorithm is equivalent to the edge coloring of a hypergraph, which is defined as follows [110].

Definition 3.2. A s-edge coloring of hypergraph $H = (V, E)$ is an assignment of s colors to the edges of H so that distinct intersecting edges receive different colors.

Notice, that in the case of `MPI_Comm_split` every new communicator receives the same context ID. This is sufficient to ensure “safe” communication because the underlying groups for the new communicators are all disjoint, thus the context ID is guaranteed to be unique for all the communicators of which a process is a member.

The algorithm used for context ID generation determines the structure of the hypergraph that is dynamically created during the execution of the program. In the case of MPICH2, a process can be a member of at most 2^k communicators, independent of the size of the world. As well, the limit on the number of context IDs can only be increased by increasing k, which essentially doubles the size of the bitmap and the amount of data for `MPI_Allreduce`. The context ID we have defined can be created as long as there is *one* process in the new communicator that has not been a leader 2^k times. Thus, the opportunity to create new context IDs grows with the size of the group which in turn depends on the size of the world. The difference is, in our case, the number of context IDs scales with the size of `MPI_COMM_WORLD` and k does not need to scale. Illustrations of the hypergraph structures

---

[9] MPICH2 stores the context ID as a 16 bit integer, where 5 bits are reserved for other purposes [8]. In MPICH2, k=11, allowing the context ID to fit into 16 bits. The size of the bitmap is 2 Kbits.

and the limiting cases for MPICH2 and FG-MPI context ID algorithms are presented in the boxed section on pages 49 and 50.

The other major difference in communicator creation is the use of leader-based communication where one process (the root) in the outer communicator gathers and then distributes the necessary information to all the other processes. This avoids the "ALL" type collectives, like `MPI_Allgather`, which in the worst case consumes $\mathcal{O}(N^2)$ for an outer communicator G of size N. As well, our `MPI_Comm_create` algorithm does not perform any collective operations on the outer communicator and uses local leaders within each of the multiple groups. This allows the processes in each of the groups to communicate independently of each other.

An advantage of MPICH2's context ID is that it can be generated with a single `MPI_Allreduce` operation, and in the case of `MPI_Comm_create` no further communication is needed. However, in the case of `MPI_Comm_split`, this is possible only after an earlier `MPI_Allgather` operation which temporarily requires $\mathcal{O}(N^2)$ space. This is a good example of the trade-offs between communication time and space where originally a $[P, O, M]$ system uses $\mathcal{O}(P^2O^2M^2)$ communications and $\mathcal{O}(P^2O^2M^2)$ space now takes $\mathcal{O}(POM)$ communication and $\mathcal{O}(PO^2M^2)$ space. The same techniques can be used to extend group sharing to OS-processes, but it would incur some additional shared memory synchronization overheads.

The use of leaders is a key part of the design of FG-MPI. Not only does it avoid the scalability problems for creating communicators, leaders played an important role in defining location-aware collectives that can take advantage of the hierarchical communication structure in a $[P, O, M]$ system.

Hypergraph structures

The context ID generation algorithms used by MPICH2 and FG-MPI exploit very different hypergraph structures. The s-edge coloring algorithm used by MPICH2 allows context ID reuse for non-overlapping groups, however, the total number of unique context IDs that can be generated is a fixed value, irrespective of the number of processes in `MPI_COMM_WORLD`. The limiting case for MPICH2's algorithm occurs if at least one MPI process is part of every communicator, where a total of 2^k context IDs (i.e., edge colors) can be generated (see Figure 3.3).

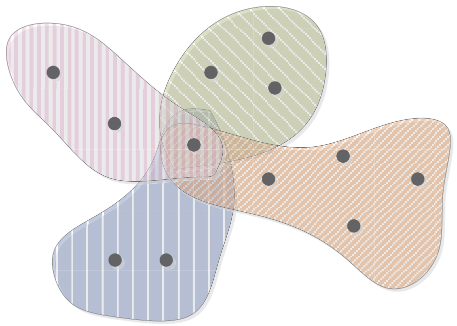


Figure 3.3: An example of s-edge coloring of hypergraph using MPICH2's context ID algorithm, where one process is member of all four communicators.

FG-MPI's s-orientability algorithm incorporates a process's `MPI_COMM_WORLD` rank as part of the context ID, which gives us an important property of being able to scale with the size of the group. The leader-based approach uses a greedy algorithm to balance the orient of the hypergraph to one that has been selected the leader the least number of times.

Continued on page 50.

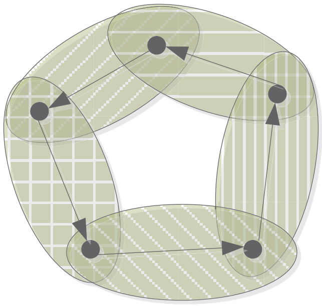


Figure 3.4: An example of s-orientability of hypergraph using FG-MPI's context ID algorithm, where the communicators form a ring structure in which each member of a group of size two is part of two communicators.

The limiting case for FG-MPI are small groups that are duplicated $2^k\times$ group-size. For example, Figure 3.4 shows groups of size two overlapping in a ring structure. The maximum number of times that this ring structure can be duplicated is $2^k\times 2$ before running out of potential leaders. This example, where we are using a uniform group size, is similar to (l, s)-orientation of hypergraphs, which is defined as follows [87].

Definition 3.3. A h-uniform hypergraph $H = (V, E)$ is called (l, s)-orientable if there exists an assignment of each hyperedge $e \in E$ to exactly l of its vertices $v \in e$ such that no vertex is assigned more than s hyperedges.

Although the algorithms used by MPICH2 and FG-MPI create different hypergraph structures it is not clear which one can create more. Moreover, the group structure is dynamically determined by the application. As mentioned in Section 3.1.2 (on page 45), the value of k, indicating the size of the bitmap, can be increased when necessary.

3.2 Message Matching Queues

MPICH2 has a modular architecture as shown in Figure 2.1 (page 20) and the implementation of the communication part is spread over the `CH3` device layer and the channel layer below it. The `CH3` device layer defines the device interface such as the communication protocols and the interface to the channel layer. The channel layer implements device specific features.

The message matching inside the middleware broadly works as follows. The middleware creates an MPI `Request` object, which describes the type of the communication, whenever a message is posted for sending or receiving by a process. MPICH2 maintains two main queues for message matching: (a) a posted receive queue (`PRQ`) for pending receives posted by a process and, (b) an unexpected message queue (`UMQ`) for messages arriving for a process that has not yet posted a corresponding receive. The `Request` object is searched in one of these queues for a possible match depending on whether it is a send request or a receive request. For example, in case of a receive call by the application, the `CH3` layer first searches the unexpected message queue to check if a matching message has already arrived. If so, then the data is copied into the receiver's user buffer and the request is marked as completed.[10] However, if a match cannot be found in the unexpected message queue, then the receive request is queued in the posted receive queue. A similar search happens when a message arrives for a process. This message is checked for a match in the posted receive queue. If found, the request is completed, otherwise the incoming message is queued in the unexpected message queue.

In MPICH2, the unexpected message queue and the posted receive queue are both implemented as singly linked-lists. In the MPICH2 implementation these queues were meant

---

[10] Copying of data and completion of request is for the simple case when the message being transferred is a short message, which is sent eagerly and is buffered by the middleware. For other cases, there are a number of events that can occur depending on the type of the send or receive call, the size of the message, whether the message transfer was buffered or not and the communication protocol (e.g., rendezvous, etc.) used for the message transfer.

for a single MPI process and a linked-list structure provides an efficient queueing and search mechanism as long as the queues remain small. In FG-MPI, it is possible to share the existing MPICH2's queues to store pending requests for all co-located processes. However, since there are now `nfg` co-located processes using the linked-list structure, the lists can become large, adding to the overhead in matching. Depending on the communication pattern the search time in either of the queues may become as large as $\mathcal{O}(\sum_{i=1}^{nfg} num\_requests_i)$, where $num\_requests_i$ is the number of un-matched requests for *process_i* (MPI process with world rank `i`).

The problem is how to reduce the overhead in matching requests in these queues while allowing for large number of co-located processes. We had the following objectives in the design of the `UMQ` and `PRQ` queues.

- Reduce the search time overhead for each $process_i$ so that it is $\mathcal{O}(\texttt{num\_requests}_i)$.
- Match and complete requests as soon as possible, irrespective of whether the co-located process that posted the request is currently scheduled or not. To allow cooperation among co-located processes, the `UMQ` and `PRQ` structures should be shared among them.
- Structure the queues so that the search overhead is the same, whether it is a wildcarded receive or not. The request matching criteria in MPI is based on the context ID, tag and the sender process and receiver process ranks as shown in Figure 2.3 (page 24). MPI allows a process issuing a receive request to use wildcards such as `MPI_ANY_SOURCE` that allows them to receive from any sending process.

In order to achieve $\mathcal{O}(\texttt{recv\_requests}_i)$ search time, we de-multiplexed the single linked-list in MPICH2 of `Request` objects into separate linked-lists for each of the co-located processes. As shown in Figure 3.5, the `UMQ` and `PRQ` queues are implemented as an array of linked-lists, with the size of the array equal to `nfg`.

Selecting the array index in these queues is key to limiting the search, for a possible

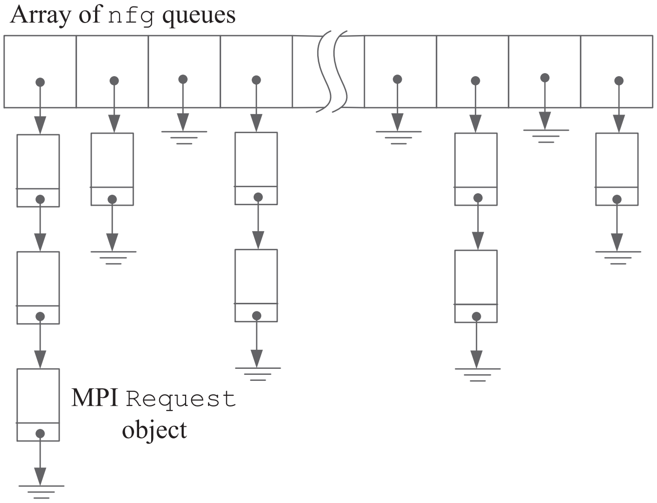


Figure 3.5: Structure of the Unexpected Message Queue (`UMQ`) and the Posted Receive Queue (`PRQ`).

match to a `Request` object, to only one of these linked-lists. This is particularly important in the case of `MPI_ANY_SOURCE` wildcards since we want to avoid a linear search through all the linked-lists. To solve this problem we selected the array index to be what is always known in any `Request` object and that is the receiving process's world rank (see Figure 2.3 on page 24). The `UMQ` is indexed by the destination rank in the incoming message header, while the `PRQ` is indexed by the world rank of the receiver process who issued the receive call. The index choice is based on the fact that the sender process always specifies the destination (receiver's) rank when sending a message and an incoming message when matched against `PRQ` should only check that receiver's list. A receiver process may not know the rank of its matching sender but always knows its own rank. Whenever a receiver $process_i$ posts a receive request that cannot be matched immediately, it can use its own world rank `i` to enqueue it at the end of its list in `PRQ`. The case for `UMQ` is similar. While separate lists are maintained for the `Request` objects for each of the co-located processes in the `UMQ` and the `PRQ`, the complete structure itself is accessible by all co-located processes. This allows an

MPI process to progress and complete requests by other processes.

In terms of space complexity, the array of linked lists introduces an extra overhead of $\mathcal{O}(\texttt{nfg})$ over a single linked-list, but provides considerable speedup with respect to search time. One limitation of using an array is that it assumes that the co-located processes have contiguous `MPI_COMM_WORLD` ranks. This, however, is not a limitation of the design and the array can be replaced by a hash table.

Even in the case of a single MPI process per OS-process, as in MPICH2, optimizations have been proposed for reducing the search time in `UMQ` and `PRQ` through further levels of de-multiplexing based on the context ID and tag [156]. These techniques can be applied to the FG-MPI implementation as well. However, the impact of the search time in message queues on the latency has been shown to have a wide variation depending on the application characteristics [82, 147]. It has been proposed in [82] that the queue usage be used to detect and remove bottlenecks in the application and avoid accumulation of messages. One advantage of having many fine-grain processes is that each of them will likely not post a large number of outstanding messages and the individual linked-lists for each of them remain small, reducing the search overhead. In Section 5.2, we present an example of using FG-MPI to re-structure a typical use of non-blocking communication in MPI, where the number of outstanding requests for each of the fine-grain processes is kept small.

3.2.1 Nemesis Queues

MPICH2 supports a number of channels as shown in Figure 2.1, with the Nemesis channel designed specifically for scalability and low latency of communication. The design of the Nemesis communication subsystem was based on a number of goals with the following priority: (a) scalability, (b) low intra-node communication overhead, (c) low inter-node communication overhead, and (d) support for multi-network inter-node communication [24]. Nemesis uses a number of techniques to achieve this, such as, (a) lock-free queues to avoid locking overhead, (b) maintaining a single receive queue for each process to avoid the overhead of polling multiple queues, which is important for scalability on large SMP systems, and (c) several optimizations to reduce cache misses.

In FG-MPI, we leverage Nemesis's design for communication among OS-processes mapped to different cores and machines. The Nemesis implementation provides a uniform interface for sending messages, whether they are sent to a remote OS-process on a different node or to a local OS-process on the same node. For intra-node communication, each OS-process has a single lock-free receive queue that other OS-processes use to enqueue messages through shared memory on the node. For inter-node communication, its network module implements a send queue and messages are enqueued in it in the same way that they are onto a local OS-process's receive queue. In FG-MPI, the Nemesis queues are shared among all the co-located MPI processes in an OS-process and in a $[P, O, M]$ system, the number of Nemesis queues is $\mathcal{O}(O)$.

3.3 Zero-copy Communication Among Co-located Processes

A common criticism of MPI's communication routines for shared address space is that they require making one or more[11] copies of the message buffer [44, 102]. In FG-MPI, standard MPI communication among co-located processes is optimized to a single `memcpy`, avoiding intermediate system copies, however, it still corresponds to passing messages by value, requiring a deep copy of the message buffer. Given that the co-located processes share a single address space, there is a potential to avoid this copy overhead through passing a pointer to the message buffer instead. Conceptually, it is similar to defining the message buffer as a "single-owner" resource, where only the current owner has exclusive read or write access to it. The sending process explicitly hands over ownership of this resource

---

[11] Implementations of some routines may require intermediate system copies due to hardware or operating system limitations [53].

to the receiver process through an MPI routine and agrees not to access it again. Single ownership has been explored for mobile-space objects in the Occam-pi language [106] and in use of contracts to guarantee safe passing of references in systems like Singularity OS [65]. Enabling of safe ownership transfer is also well studied in the Actor model of message passing, where messages can be efficiently sent using pointers, without the introduction of a shared state between the concurrent entities. Ownership type systems to annotate messages are used in libraries like Scala Actors and Kilim [132], and tools have been proposed to infer safe transfer of ownership of messages without annotation [102].

We investigated extending MPI to support zero-copy communication among co-located MPI processes.[12] The potential benefits are in avoiding deep copies of data structures that incur the overhead of serialization and de-serialization. Enabling zero-copy communication in MPI involved looking at a number of design issues, which we discuss below.

3.3.1 Symmetry and Interoperability

MPI defines a large and rich set of library routines, however, these routines were designed around a small number of concepts, keeping simplicity and symmetry in mind. As mentioned by Gropp [53], one of the objectives of the MPI design was to require the user to learn only a few concepts in order to use MPI. Our goal also was to define the zero-copy routines in terms of the existing point-to-point routines and support the same properties of simplicity and symmetry. Some of the main considerations in defining the zero-copy routines were as follows.

- Interoperability of zero-copy routines with existing MPI communication routines, i.e., be able to use different routines on the sender and receiver sides. This functionality already exists for the standard MPI routines and is supported by the zero-copy

---

[12] The term "zero-copy transfer" is somewhat loosely used in the context of MPI. Passing messages through a direct copy of data between receiver and sender, without intermediate copies, is also sometimes referred to as zero-copy transfer [53]. In our discussion, however, we define it as passing a pointer to the data buffer in a message, with the sender process relinquishing the ownership of the buffer to the receiver process.

implementation as well.

- Interoperability of different communication modes (such as blocking and non-blocking etc.,) on the sender and receiver sides.

- Transparency of the communication to the user, i.e., the user need not be concerned about whether the sender and receiver are co-located or not. If the communication is among processes that are not co-located then the zero-copy routines will communicate like the standard MPI routines. This property is important to support seamless mapping of MPI processes onto OS-processes, cores and nodes. This property also enables the use of `MPI_ANY_SOURCE` wildcards on the receiver side, as is possible with existing routines.

```
int MPIX_Zrecv(void ** buf_handle, int count, MPI_Datatype datatype,
      int source, int tag,  MPI_Comm comm, MPI_Status *status)

int MPIX_Zsend(void **buf_handle, int count, MPI_Datatype datatype,
                 int dest, int tag, MPI_Comm comm)

int MPIX_Izrecv(void ** buf_handle, int count, MPI_Datatype datatype,
      int source, int tag, MPI_Comm comm, MPI_Request *request)

int MPIX_Izsend(void **buf_handle, int count, MPI_Datatype datatype,
      int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

Figure 3.6: Syntax of the zero-copy routines

- Zero-copy routine call semantics are similar to those of other point-to-point routines and thus already familiar to the user. Figure 3.6 shows the syntax of the new routines. The user can also continue to make use of the rich set of MPI datatypes due to this symmetry in the design.

Figure 3.6 shows both the blocking and non-blocking versions of the zero-copy routines. The only syntactical difference between the zero-copy routines and the corresponding MPI ones is that the sender process passes a reference to the message buffer pointer (`void**`) as the first argument. The message buffer must be allocated on the heap and the sender transfers ownership of the buffer to the receiver by agreeing not to access it again. The implementation of zero-copy involved extension to the request structure and introduction of a zero-copy request type inside the middleware. The message matching component was augmented to incorporate the new request type into the search procedure for the posted receive queue and the unexpected message queue. If both the sender and receiver processes are co-located and communicate through the zero-copy routines, then the reference to the message buffer pointer is passed to the receiver and the communication avoids memory copy of the data. For interoperability and backward compatibility, if only one of the sender or receiver processes uses the zero-copy routines, or they are not co-located, then the communication will be the same as if both processes had used the corresponding standard MPI routines. The sender buffer is de-allocated by the middleware in case of a `MPI_Zsend`/`MPI_Izsend` pairing with a non-zero-copy receive routine or with a receiver process that is not co-located. Recall that the receiver buffer is allocated by the user in case of standard `MPI_Recv`/`MPI_Irecv` calls. If `MPI_Zrecv`/`MPI_Izrecv` is paired with a non-zero-copy send routine or with a sender process that is not co-located, then the receiver buffer is allocated by the middleware. As a safety check, `MPI_Zsend` and `MPI_Izsend` routines check that the `buf_handle` does not specify a null buffer, and likewise the `MPI_Zrecv` and `MPI_Izrecv` routines check that the `buf_handle` has been initialized to `NULL`.

MPI provides a set of predefined basic datatypes as well as routines to allow the users to construct their own datatypes, which can be used to describe non-contiguous data [52]. User-defined datatypes allow description of complex data structures, however, they have an associated serialization and de-serialization cost when used with standard MPI routines. The zero-copy routines avoid both the copying and serialization costs for user-defined datatypes for the case when both the sender and receiver are co-located. In this case only the pointer to the data buffer is passed to the receiver and the layout information of the data in the MPI `datatype` field of the zero-copy routines (Figure 3.6, page 57) is ignored as it is not needed for the pointer transfer. Note that although information about the layout of the user-defined datatype is not used for co-located zero-copy communication, the user must construct a proper datatype to allow the middleware to interpret and serialize/ deserialize the data items when the sender and receiver are not co-located. MPI also allows specialized functionality where datatypes on the sender and receiver can be placed in a different memory order [44]. The zero-copy functionality does not cover this scenario.

3.3.2 Discussion

In this section we discuss some design choices we considered when implementing the zero-copy functionality. One consideration was whether to introduce a special "pointer" datatype (e.g., `MPI_PTR`) for the MPI communication routines that would indicate that a pointer to the data buffer is being passed, instead of introducing new `MPIX_Zxxxx` routines. One basic limitation of the datatype approach is that the zero-copy transfer indicates a communication mode, which is semantically different from the concept of a datatype. The MPI send/receive semantics are based on message copying and the message matching is agnostic to the type of the data sent. Information about the datatype is not part of the message header or the request matching logic. The middleware would, in essence, treat a `MPI_PTR` type as a 8 byte message[13] with the onus placed on the programmer to treat it as a pointer. This requires the programmer to be cognisant of whether the sender and receiver are co-located before using a special datatype to communicate among them. This restricts the flexibility of process mapping and the ability to specify a different mapping of MPI processes to OS-processes on the command-line. In our approach of defining `MPIX_Zxxxx` routines, the communication

---

[13] Assuming an 8 byte pointer on 64-bit machines.

mode is transparent to the user and the middleware ensures interoperability with existing MPI routines.

Recent work by Friedley *et. al.* [44] have considered providing zero-copy functionality as a user-level library (OPI-HMPI) that runs on top of MPI. Independently to our work, they defined a syntax of the zero-copy routines that is similar to the one specified in Figure 3.6. Their implementation approach is fundamentally different from ours in that their library is a wrapper that is layered between the application and the underlying MPI implementation [18]. The HMPI wrapper library implements MPI processes as pre-emptive threads and intercepts messages that are communicated among MPI processes on the same node and uses a single address space to send pointers to the message data among them. For inter-node communication, it uses the standard MPI routines for communication. This layered approach uses two different communication paths for intra-node and inter-node communication, with the consequence that separate message passing mechanisms have to be implemented for intra-node along with management and allocation of memory buffers and maintenance of communication buffer pools in a thread-safe manner. OPI-HMPI implements one memory pool per MPI process to avoid global synchronization and reduce locking overheads. In our design, which can expose massive concurrency, one memory pool per MPI process does not scale. We chose to implement the zero-copy functionality by extending the existing message matching engine and request structure inside the middleware. This has the important property of providing a single communication path and a uniform method for sending and receiving messages for both intra-node and inter-node communication. As discussed in Section 3.2 (page 51), the request queues and communication structures are shared which allows the system to scale. As well, the zero-copy routines in Figure 3.6 can interoperate with other standard MPI routines, which relieves the programmer from the necessity of replacing all routines like `MPI_Send` and `MPI_Recv` in the program with special user-level library calls, which is the approach used by OPI-HMPI. One advantage of the OPI-HMPI scheme is that it works across cores on the same machine and it would be interesting to investigate if MPICH2's intra-node communication can be extended to use zero-copy communication across OS-processes.

3.4 Process Deployment

Process deployment is an important part of the process management environment. We had two main goals for process deployment in FG-MPI: (1) to make the system flexible and transparent to allow the programmers to optimize process mapping, and (2) to allow easy porting of existing MPI programs to FG-MPI.

MPICH2 uses the `mpiexec` command-line utility to launch MPI processes and the mapping of processes to cores and machines can be flexibly configured through a `hostfile`.

```
mpiexec -f hostfile -n Y program
```

A model of execution commonly used is SPMD (Single Program Multiple Data), where, for example, all the `Y` MPI processes launched in the above command execute `program`. Inside the program, statements like `if-else` or `switch` may be used to selectively assign a set of instructions to processes based on their ranks. This adds extra complexity to the program. Often these condition statements based on the MPI process rank are used in places where process behavior is different for different ranks. These type of condition statements are scattered throughout the code making the program difficult to read.

Support for MPMD (Multiple Program Multiple Data) is through the colon notation along with `mpiexec` to assign different executables to different processes.

```
mpiexec -f hostfile -n Y prog1 :  -n Z prog2
```

FG-MPI adds another dimension by mapping multiple MPI processes to OS-processes. The user can use `mpiexec` with the `nfg` flag to specify the number of MPI processes per OS-process in addition to the `n` flag specifying the number of OS-processes.

For example, the command:

```
mpiexec -f hostfile -nfg X -n Y program
```

starts up $X \times Y$ MPI processes with X MPI processes inside each of the Y OS-processes. In the current implementation, the co-located MPI processes are assigned ranks in consecutive blocks. In the above example, the first OS-process will contain MPI processes of ranks $[0 \ldots X-1]$ and the next one will have processes with ranks $[X \ldots 2X-1]$ and so on. It is possible to specify a different number of MPI processes in each OS-process using the colon notation.

```
mpiexec -nfg T -n U prog1 :  -nfg V -n W prog2
:  -nfg X -n Y prog3
```

One important feature of this approach is the flexibility of mapping co-located MPI processes among OS-processes, from the one extreme of executing them all inside a single OS-process to the other extreme of having only one MPI process per OS-process. The number of MPI processes that can be launched per OS-process is limited only by the available memory on the system. The `mpiexec` command is backward compatible, so omitting the `nfg` parameter equates one MPI process with one OS-process.

The second level of mapping introduces new opportunities for executing MPMD programs, where each of the co-located MPI processes invoke functions instead of main programs. We treat SPMD as a special case of the MPMD program, where each process invokes the same function. The boilerplate code in Listing 3.1 gives an example of converting an existing SPMD MPI program to a FG-MPI program. There are two steps required: (1) Rename `main` to `FG_Process`[14] and, (2) Add the boilerplate code in Listing 3.1 at the beginning of the program.

The boilerplate contains two user-defined functions; `binding_func` and `map_lookup`. The `binding_func` function takes the `MPI_COMM_WORLD` rank as input and maps it to the function pointer that the corresponding MPI process will be executing. The `map_lookup`

---

[14] The `FG_Process` function name is simply being used as an example in the boilerplate and there is no restriction that this particular name be chosen.

function takes a string as its third parameter and uses it to select a binding function. The purpose of the `map_lookup` and `binding_func` combination is to allow the ability to determine mapping of processes at runtime in addition to compiled mappers. This allows experimentation with different bindings of the MPI processes to functions without re-writing and re-compiling the program. Since the mapping is localized to each OS-process, it is also possible to specify a different binding function for each OS-process.

```
/******* FG-MPI Boilerplate begin *********/
#include "fgmpi.h"
int FG_Process( int argc, char** argv ); /*forward declaration*/
FG_ProcessPtr_t binding_func(int argc, char** argv, int rank){
    return (&FG_Process);
}
FG_MapPtr_t map_lookup(int argc, char** argv, char* str){
    return (&binding_func);
}

int main( int argc, char *argv[] )
{
    FGmpiexec(&argc, &argv, &map_lookup);
    return (0);
}
/******* FG-MPI Boilerplate end *********/
```

Listing 3.1: Boilerplate code to convert an existing SPMD MPI program to a FG-MPI program.

In the SPMD example in Listing 3.1, the `binding_func` maps all process ranks in `MPI_COMM_WORLD` to `FG_Process`. However, the real flexibility of mapping comes from the ability to assign different functions for each MPI process. Listing 3.2 gives a simple MPMD example where the odd numbered ranks are mapped to `ProcessA` function and the rest to

`ProcessB` function.

The `str` parameter of the `map_lookup` function is not being used in the examples in Listings 3.1 and 3.2 and a single `binding_func` is returned. The `str` parameter can be specified on the `mpiexec` command line to allow selection of different binding functions at runtime.

```
/******* FG-MPI Boilerplate begin *********/
#include "fgmpi.h"
int ProcessA( int argc, char** argv ); /*forward declaration*/
int ProcessB( int argc, char** argv ); /*forward declaration*/
FG_ProcessPtr_t binding_func(int argc, char** argv, int rank){
    if (rank % 2) return (&ProcessA);
    else return (&ProcessB);
}
FG_MapPtr_t map_lookup(int argc, char** argv, char* str){
    return (&binding_func);
}

int main( int argc, char *argv[] )
{
    FGmpiexec(&argc, &argv, &map_lookup);
    return (0);
}
/******* FG-MPI Boilerplate end *********/
```

Listing 3.2: An example of a simple MPMD mapping

Apart from the boilerplate, the MPI routines used in the program are exactly the same as in a standard MPI program, without any special prefixes or suffixes. All the functions invoked by the MPI processes e.g., `FG_Process` in Listing 3.1 and `ProcessA` and `ProcessB` in Listing 3.2, are written as regular MPI applications beginning with `MPI_Init` and ending with `MPI_Finalize` routines. The `argc` and `argv` arguments provided to these functions are the same that are passed to the `main` function in a MPI program. In Appendix B, we present the complete code of a small application to demonstrate writing a program using FG-MPI.

3.5 Limitations

In this section we discuss the limitations of implementing MPI processes as user-level non-preemptive threads sharing a single address space. One limitation of multiple co-located MPI processes is that global and static variables in the program can cause undesired side effects. Such variables, unless they are read-only, must be removed from the code for it to work correctly. The side effects of using shared global variables and static variables is an issue that has been well studied in other systems that allow multiple user-level threads per core. There are tools available for both `FORTRAN` [114] and `C` [2] that re-factor the source code to privatize global variables. We use one such tool to re-factor the NAS parallel benchmarks for our experimentation in Chapter 4.

The current implementation of FG-MPI maintains a fixed-sized stack of size 128K bytes for each coroutine. There is a potential for stack overflows if the programmer allocates large sized stack-based arrays. It is recommended that large allocations be made on the heap instead of the stack. However, as mentioned in Section 2.1.5 (page 30), this stack size can be provided as a configurable option. It is also possible to replace fixed-sized stacks with those that grow dynamically, similar to the linked stacks proposed by Capriccio [149].[15]

One effect of non-preemptive scheduling is that a computationally intensive process may block the progress of other co-located processes. We added a `MPIX_Yield()` routine that allows one process to voluntarily yield to the scheduler. This can be used to balance the amount of computation and communication in the application. If the currently executing

---

[15] It is interesting to note, however, that a choice between stackless and stack-based implementations for languages like Python [3] and Scheme [40] are available.

process is the only one on the run queue of the scheduler, then `MPIX_Yield()` is simply a `no-op` operation.

Execution of blocking file I/O by one process is another operation that can impede the progress of other co-located processes. One possibility is to structure the code so that I/O system calls are placed in an OS-process of their own. This can help avoid blocking other processes during I/O execution. Another scheme that is used for cooperative multitasking is to wrap the I/O library function so that the process initiating the I/O operation yields control to another process. The wrapper is then responsible for notifying the original process when the I/O completes [5, 67]. MPI, as well, provides the ability for users to define non-blocking operations through a feature called generalized requests. Work by Latham *et. al.* [85] proposes extending the generalized request interface to simplify the implementation of non-blocking I/O operations by the users.

The advantages and dis-advantages of user-level threads are well-known and solutions like scheduler activations [7] exist. Scheduler activations are not supported in Linux, however, kernel extensions like FlexSC and others [34, 131] exist that decouple the execution of a system call from its invocation. FlexSC provides a M-on-N user-mode scheduler that implements cooperative scheduling among the user-mode threads where the system calls act as yield points. It would be interesting to use FlexSC to schedule and batch system calls to the kernel making it possible for the user-level scheduler to coordinate with the OS scheduler.

3.6 Summary

In this chapter we discussed the low-level system components of the FG-MPI runtime system. We described the techniques used to share the process group maps and algorithms for creation of communicators. We also discussed communication among co-located processes and the structure of message queues in the middleware and how they are shared among the co-located MPI processes. We defined a way to avoid copying of messages among co-located processes through zero-copy routines that can be used to pass a pointer to the message buffer instead. These routines can interoperate with existing MPI point-to-point communication routines. Process deployment is an important part of the MPI execution environment and we described how MPI processes are mapped to OS-processes and cores and support for deployment of MPMD programs. Lastly, limitations of implementing MPI processes as user-level threads were discussed.

Chapter 4

Experimentation

> The process is really just Iterate, Iterate, Iterate.
>
> Chris Clark

MPI programs are typically written as SPMD where the program is parameterized by "N", the number of MPI processes. Parameter N determines the granularity of the program and gives the amount of available concurrency. In executing MPI programs, one typically matches the number of MPI processes to the number of cores, the amount of available parallelism. Matching the concurrency to the available parallelism fixes the granularity of the program to make it as coarse-grain as possible. However, maximizing the granularity is not always optimal because of the effect it has on the cache behavior and the number and sizes of the messages sent and received. There are also MPI programs where N is partly determined by the problem size and may not exactly match the parallelism available in the machine. For these reasons it should be possible to be able to adjust the granularity independently from the amount of parallelism and be able to expose more concurrency than can be executed in parallel.

Introducing added concurrency by over-decomposing the problem is a well-known performance optimization technique for SPMD scientific computing programs. Since data decomposition is typically hard-coded and difficult to change, over-decomposition for MPI programs depends on its runtime environment. One simple technique that is commonly used is to over-subscribe the number of cores by starting more MPI processes and thereby more OS-processes. A recent paper [66] studied the use of over-subscription on multicore machines and report a 10% performance degradation for the NAS parallel benchmarks with MPI. As discussed in their paper, a problem with increasing the amount of over-subscription is that the OS scheduler is not aware of the cooperative nature of the parallel application on a dedicated machine. "Over-subscription" in FG-MPI is done through interleaved concurrency managed by the user-level scheduler integrated with the MPICH2 progress engine and the effects of OS-noise [41] is reduced by not over-subscribing cores.

The key issue in a system that adds more concurrency than available parallelism is to minimize the overheads in order to maximize the benefits and make it effective over a wider range. The tight integration of FG-MPI runtime system in the MPI middleware made it possible to address overhead issues related to context switches, MPI-aware scheduling, as well as added synchronization costs, which was also highlighted as a problem in [66].

In this chapter we evaluate the benefits of added concurrency for cache awareness and message size and show that performance gains are possible by using FG-MPI to adjust the grain-size of a program to better fit the cache and potential advantages in passing smaller versus larger messages. In Section 4.1.1, we discuss the scheduling of non-preemptive threads built on top of coroutines and show that switching between them is an order of magnitude faster than an OS-level context switch. In Section 4.1.2 we measure the overheads of the extra message passing that can occur to show that it is similar to that of a memory copy when using standard MPI routines. This overhead can be further reduced by using zero-copy routines. Adding concurrency also affects the communication time for collectives since now these collectives are over a larger collection. In Section 4.1.3 we describe the use of location-aware implementation of collectives that takes advantage of the single address space to speed up the collectives for co-located processes. We measure this added overhead for the `MPI_Barrier()` call. In Section 4.1.4 we look at the potential performance improvements for better cache behavior. Lastly, in Section 4.2, we evaluate the use of FG-MPI on the complete set of the NAS parallel benchmarks over large problem sizes and discuss the characteristics of the benchmarks with regards to trade-offs between the added costs and benefits.

4.1 Overheads in Added Concurrency

For the experiments the test setup consisted of a cluster with 16 nodes connected by a 10GigE Ethernet interconnection network. Each of the nodes in the cluster is a quad-core, dual socket (8 cores per node) Intel Xeon® X5550, 64-bit machine, running at 2.67 GHz. All machines have 12 GB of memory and run Linux kernel 2.6.18-194.8.1.el5.

4.1.1 FG-MPI Context Switch

One important cost that is a consequence of adding concurrency is the time taken to switch between processes. FG-MPI's non-preemptive runtime is build on top of coroutines, each with it own stack. The runtime is partially derived from Capriccio [149], a scalable thread library for high-concurrency servers. We use Toernig's coroutine (`coro`) library [143], which provides highly efficient yield for switching context between coroutines. We also extended the system to be able to use any similar type of coroutine package and provide the option for using PCL (Portable Coroutine Library) [89].

In FG-MPI every MPI call is a potential de-scheduling point where, depending on the call and the state of the middleware, the scheduler chooses the next process to run. MPI communication calls provide a natural yield point for switching between coroutines where one process when it enters the middleware can progress messages for all of the co-located processes.

Switching between processes involves switching to the runtime scheduler, selection of next runnable MPI process by the scheduler and switching to the new process. We measured this switching time for both of the coroutine packages that FG-MPI can use. The `coro` library provides the fastest switching time ($0.13\,\mu$s) while the PCL library switching time is $0.81\,\mu$s. This time is an order of magnitude faster than OS-level context switch which takes $6.85\,\mu$s. Our results are similar to the numbers reported for threading benchmarks in Capriccio. In a comprehensive study, they demonstrated that coroutines outperform

NPTL (Native Posix Thread Library) and LinuxThreads (Linux kernel threads) for both raw performance and scalability [148, 149].

Coroutines provide very efficient context switch time in comparison to other types of threads and this makes it easier to scale to support massive amounts of concurrency. We have tested FG-MPI with thousands of co-located processes and, when not constrained by memory, have used it for debugging with `gdb` to run an entire MPI application in a single OS-process.

4.1.2 Messaging Costs

In FG-MPI one of the effects of adding more concurrency is that we send more smaller sized messages. A potential overhead is the cost of the inter-process communication that is not present in a coarse-grain program. However, the conventional wisdom that it is faster to share a memory location versus copying is not always the case on multicore. Not only is cache coherence a power hungry operation that is constantly on, as shown in the Barrelfish [128] project, in modern multicore machines passing a message can be faster than shared memory access between cores associated with different caches.

In the case of communication between co-located processes, we provide two mechanisms. The first is through standard MPI communication routines, where we exploit the single address space to optimize point-to-point communication by simply doing a `memcpy`. The second is through the zero-copy routines described in Section 3.3, that pass a pointer to the message buffer from the sender to the receiver process. Our objective was to achieve low messaging overhead through synchronous communication between co-located processes and avoid intermediate system copies.

We first discuss the messaging costs of the standard MPI routines. For co-located processes, there are two cases to consider depending on whether the sender or the receiver executes first. The two cases are symmetric, therefore, we describe in detail the one where a receiver process executes a receive call before it can be completed; i.e., the sender process has not yet executed a matching send call.

In this case, the receiver process first determines that the sender has not yet executed a send call by looking in the unexpected-message-queue and queues its request in the posted-receive-queue. It then blocks waiting for the data arrival. When the sender process executes the send call, it first checks in the posted-receive-queue and finds the matching receive. The sender then copies the data directly into the receiver's buffer and unblocks it. Note that there is only one memory copy in this operation and no intermediate system copies are made. The additional overheads over a simple `memcpy` in this case are the context switch costs, looking in the unexpected-message-queue and queueing the receive request in the posted-receive-queue by the receiver and finding the matching request in the posted-receive-queue by the sender.

For processes that are not co-located, MPICH2 treats messages of 64 KBytes or above as long messages that require a rendezvous protocol for communication. For co-located communication we avoid the extra cost of rendezvous and the communication between two co-located processes is the same as described above, irrespective of the size of the message.

In order to measure the cost of message communication within co-located MPI processes, we designed a benchmark that compares sending MPI messages, picked randomly from a memory location, between two co-located processes with the cost of doing a memory copy inside one process. We measured the overhead (difference between a message send/receive and a `memcpy`) for different messages sizes. The reason for randomly picking from a memory location was to isolate the effect of messaging only (i.e., without any cache effects). We also ran a test where the same message is being sent and the overhead of messaging over `memcpy` compared to the random selection case was less, but we report the numbers for the random selection as it is closer to what we would expect in an application. Our results showed that the average overhead of co-located messaging in comparison to a `memcpy` operation was $0.43\,\mu$s with a standard deviation of $0.065\,\mu$s for messages ranging in size from 2 bytes to 128 Kbytes. For messages in the range 256 Kbytes to 1 Mbytes, the message overhead increased from 1 μs to 2.3 μs. Note these measurements include the context switching and scheduling time to complete the operation.

It was also interesting to measure the co-located messaging cost for a system like AMPI that layers MPI on top of the Charm++ runtime system. Using the same benchmark, we measured the cost for AMPI. For message sizes in the range 2 bytes to 256 bytes, AMPI was slightly faster than FG-MPI, the difference between messaging times for FG-MPI and AMPI was in the range 0.12 μs to 0.15 μs. For messages 512 bytes and above the difference increased sharply with FG-MPI outperforming AMPI by a large margin as shown in Table 4.1. Interestingly, AMPI failed to execute for message sizes 64 Kbytes and higher, reporting that it cannot allocate memory. With FG-MPI we successfully tested for message sizes of several megabytes on the same machine without any problems.

Message size	FG-MPI time	AMPI time
2 bytes	0.47 μs	0.32 μs
32 bytes	0.475 μs	0.34 μs
64 bytes	0.48 μs	0.36 μs
256 bytes	0.57 μs	0.44 μs
512 bytes	0.587 μs	0.585 μs
1 Kbytes	0.6 μs	0.8 μs
8 Kbytes	1.4 μs	3.6 μs
32 Kbytes	4.23 μs	15.1 μs
64 Kbytes	7.9 μs	fails

Table 4.1: Comparison of FG-MPI and AMPI co-located messaging times.

As discussed in Section 3.3, further optimization is possible with co-located processes in a single address space by using zero-copy message passing where processes pass a reference to the data. We tested the above benchmark in FG-MPI by replacing the standard MPI send and receive routines with the corresponding zero-copy routines. The messaging cost in this case was 0.4 μs for all message sizes.

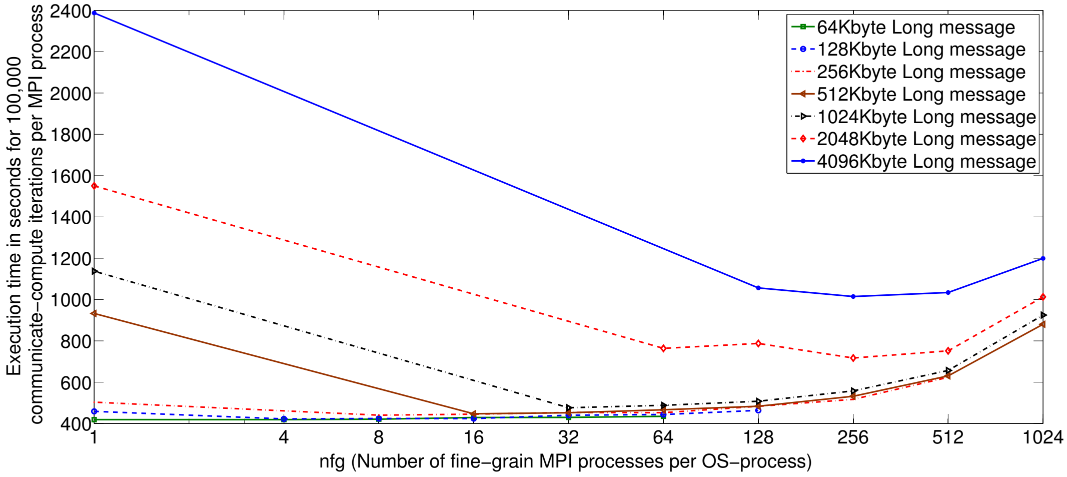
One potential benefit to adding concurrency for communication between remote processes is that by sending more smaller messages, rather than one large one, we can avoid the long-message protocol that requires a rendezvous between the processes. In effect, the added concurrency acts like packetization where the packets (smaller messages) can be pipelined between the two OS-processes [23]. Messages can be split up into smaller pieces and the MPI processes at the receiver can begin working on the smaller pieces without having to wait for the entire message to arrive. In order to study this effect, we designed a benchmark where processes alternate between computation and communication for a number of iterations. In the coarse-grain MPI case we have two processes doing a fixed amount of computation (`Cp`) and communication (`Cm`) per iteration. In the finer-grain case, `Cp` and `Cm` per OS-process is divided among the number of processes defined by the `nfg` flag while keeping the total volume of computation and computation the same in both cases. In coarse-grain MPI fewer number of large messages are communicated whereas in FG-MPI the size of messages is smaller (divided by P=`nfg`), but the total number of messages communicated is multiplied by the `nfg` parameter.

Figure 4.1(a) shows the effect of sending fewer long messages versus several short messages[16] over the network. The advantages of avoiding the rendezvous protocol for network communication are clear in this figure. We also see that the overhead of additional messaging in FG-MPI is small (even for substantially large values of `nfg`). Figure 4.1(b) shows the best times achieved in FG-MPI for different communication volumes compared with MPI.

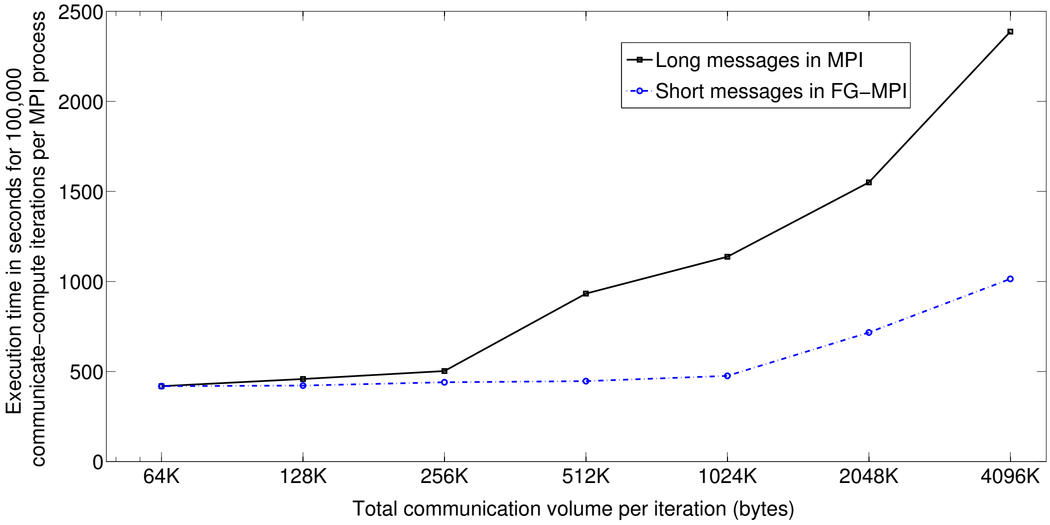
In conclusion, FG-MPI makes it possible to adjust the message size independently from the size of the machine and the added concurrency results in a more fluid communication with more messages flowing potentially requiring less synchronization with more overlap between communication and computation. We expect this benefit to become more evident

---

[16] MPICH2 treats messages of 64 KBytes or above as long messages that require a rendezvous protocol for communication. For the fine-grain results, The `nfg` parameter is chosen to divide the message size so that it is less than 64 KBytes.



(a)



(b)

Figure 4.1: (a) Execution time of sending long messages in MPI versus short messages in FG-MPI for the same total amount of computation and communication volume. (b) The best times achieved in FG-MPI compared to MPI for part (a).

on programs computing on larger datasets.

4.1.3 Collective Communication

Concurrency adds overhead to the collectives since it results in more MPI processes and hence more messages and more context switches. The cost of synchronization has a significant impact on MPI programs that use collectives and may offset any of the potential advantages of added concurrency. Collective operations in FG-MPI are implemented in terms of point-to-point operations and it leverages the algorithm optimizations for collectives employed by MPICH2. However, it is possible to reduce the synchronization overhead by taking advantage of the single address space and optimize the collective communication for those processes that are co-located. We demonstrate this approach for the `MPI_Barrier` operation. For synchronization of MPI processes on the same node, we used a counter and signal based mechanism typically used in shared memory barrier algorithms [96].

For `MPI_Barrier`, one leader per OS-process is selected from the co-located MPI processes and one leader is selected for all of the OS-processes on the machine. Inside the OS-process, a shared variable is used to count the co-located processes that enter the barrier. Processes inside the OS-process increment the counter and block waiting for the co-located leader to clear the barrier and re-schedule them. Similarly, the OS-process leaders uses MPICH2 Nemesis's shared memory to coordinate with the leader of all the OS-processes on the machine. Once processes on each multicore node have synchronized then all those leaders communicate after which the leader of the OS-processes signals that they can leave the barrier which in turn allows the co-located processes to leave the barrier. This type of location-aware implementation of the `MPI_Barrier` is optimized for each level of the communication hierarchy and is supported by the scheduler to minimize its interactions with the middleware. The naive approach of simply relying on the MPI's point-to-point can take advantage of `memcpy` for the communication among co-located processes but it results in more interactions than necessary with the middleware and the progress engine.

Table 4.2 presents the barrier latency as concurrency (`P`) per OS-process in a [P;O;M] system is increased from 1 to 256 for [P;1;1], [P;8;1] and [P;8;16] executions. O (the number of OS-processes) is kept equal to the number of physical cores per machine to reduce the influence of the OS scheduler. In [66] over-subscription was found to have a detrimental effect on the performance of MPI programs containing collectives.

P	**[P;1;1]**		**[P;8;1]**		**[P;8;16]**	
(`nfg`)	time (μs)	N	time (μs)	N	time (μs)	N
1	-	1	1	8	122	128
2	1	2	1	16	126	256
4	1	4	2	32	129	512
8	1	8	3	64	129	1024
16	3	16	4	128	137	2048
32	5	32	10	256	149	4096
64	10	64	20	512	183	8192
128	22	128	41	1024	259	16384
256	45	256	84	2048	438	32768

Table 4.2: Barrier latency time (μs) with varying concurrency (P) for three [P;O;M] executions, where P is the number of MPI processes per OS-process, O is the number of OS-processes per node and M is the number of nodes. N=P×O×M is the total number of MPI processes.

Table 4.2 shows that the increase in barrier latency with concurrency is sublinear, e.g., on a single multicore node ([P;8;1]), the latency increased from 1μs to 84μs as concurrency increased from 1 to 256. These results were obtained by iteratively calling the barrier operation several thousand times and averaging the result. As such, they are a lower bound estimate on the cost of barrier communication. It is also interesting to compare this approach to AMPI. In AMPI we have MPI on top of the Charm++ runtime which in turn uses MPI as a communication layer. Its implementation of MPI can still take advantage of the single address space, but the Charm++ scheduler and lower level MPI progress engine operate independently with the potential for the same types of delays that occur between the progress engine and the OS scheduler. Using the same program, we measured the cost of `MPI_Barrier` for AMPI on [P;8;1]. For P=1 the time was 18 μs, versus 1 μs for FG-MPI, and for P=256 was 662 μs versus 84 μs for FG-MPI. On multicore, this shows the advantage of reducing the layers by supporting concurrency inside the communication middleware.

4.1.4 Cache Behavior

In order to demonstrate the potential benefits of using added concurrency to improve the cache behavior, we designed several experiments to measure the cache effects. We focused on block structured algorithms as they provide the best case for measuring the extent to which we can improve performance.

Block structured algorithms in MPI are commonly used for parallel scientific computations. The blocks generally represent the working set size of a task and blocking is used to exploit both temporal and spatial locality for efficient execution. Today's systems of multiple cores on a single chip have a multi-level hierarchical memory model with smaller caches per core [6]. Applications need to be able to express fine-grain parallelism with smaller working set sizes to fit in caches. FG-MPI provides the ability to achieve better memory locality and cache hit ratios simply through the use of the `nfg` parameter on the command line.

The tests are run on a single multicore machine with the specification described in Section 4.1. The multicore machine has 8 cores with three-level cache; L1(data/instr) is 32K/32K, L2 cache is 256K/core and L3 cache is 8M /socket. The total memory on the machine is 12 GB.

Experiment A

This benchmark algorithm takes two square matrices as input and partitions them into square blocks of size determined by the total number of MPI processes. It then assigns each pair of sub-matrices to the MPI processes. Each process computes on these sub-matrices and then exchanges its sub-matrices with its neighbours.[17] After the exchange, each process computes with new values in its sub-matrices and this process repeats for a number of iterations. This benchmark requires the total number of MPI processes to be a square because the sub-matrices are evenly distributed among them.

Figure 4.2 shows the effect of added concurrency on execution times compared with MPI on input matrices of size 4096. The P=`nfg` parameter on the bottom axis is the concurrency for different [P;O;1] executions for a total number of P×O MPI processes. The hashed bars correspond to the traditional MPI [1;O;1] executions, while added concurrency is represented by solid bars. Notice that the hashed bars for MPI are not present for [1;2;1] and [1;8;1] because they are not equal to a square number of processes. FG-MPI, however, can use all the cores by appropriately setting P=`nfg` so that the product of P and O is a square.

As the total number of the MPI processes (P×O) increases, the block sizes that each of them operate on decreases. Figure 4.2 shows substantial performance improvements by adjusting the block size through added concurrency, without any serial cache blocking optimizations. The best time obtained with FG-MPI is `12.55` seconds with a [512;8;1] execution. Note that our node has 8 physical cores and the FG-MPI best results correspond to the case where the amount of parallelism equals the number of cores. For interest, Figure 4.2 also shows the effects of over-subscription for this example using [P;16;1] (two OS-processes per core). The time achieved with MPI is `70.49` seconds with [1;16;1] execution, which is more than 5 times slower than the best time of `12.55` seconds achieved with FG-MPI mentioned above. Note that serial blocking techniques can be used to improve cache hit ratios but such techniques require modifications to the algorithms and are tuned for particular architectures and are not portable.

In order to quantify the effect of cache on our results, we collected memory access data using Intel's® VTune™Amplifier XE [70] advanced hardware analysis for the Nehalem®

---

[17] The computation is Cannon's matrix multiplication in our benchmark.

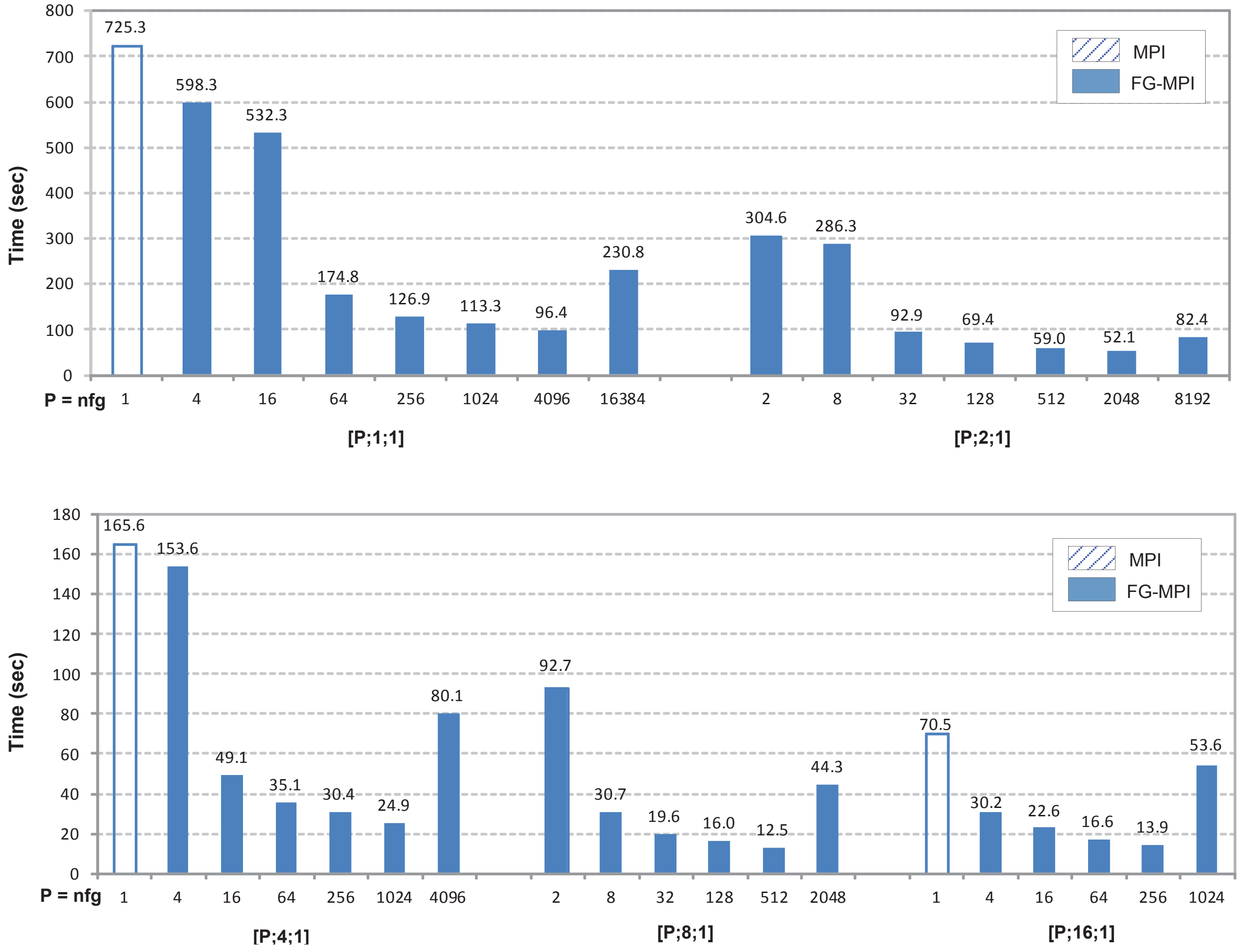


Figure 4.2: Execution times (sec) with different values of concurrency (`nfg`) for Experiment A on one multicore node.

micro-architecture. This analyzer has low overhead and uses event-based sampling for data collection. Table 4.3 shows there is a significant difference in the last-level (LL) cache[18] misses for the best times in Figure 4.2 with added concurrency, which shows that the performance benefits are the result of better cache locality.

	Time (sec)	LL Cache misses
MPI [1;16;1]	70.49	396,000,000
FG-MPI [512;8;1]	12.55	75,000,000

Table 4.3: LL cache misses for the best times achieved with MPI and FG-MPI best times in Experiment A.

The effect of added concurrency was repeated in matrices of other sizes, as shown in Table 4.4, with the performance improvement over MPI increasing with larger matrix sizes. This is expected as in the MPI case, the block granularity is fixed to the amount of parallelism and with larger matrices the mismatch with the cache size increases.

Matrix size	FG-MPI [P;O;1]			MPI [1;16;1]
S	P=nfg	O	Time (sec)	Time (sec)
2048	128	8	1.47	3.7
4096	512	8	12.55	70.49
8192	2048	8	134.5	944.58

Table 4.4: Best execution times for MPI and FG-MPI on different matrix sizes for Experiment A.

It is interesting to note from Table 4.4, that the $(\frac{S^2}{P \times O})$ ratio is the same for the best times achieved for different matrix sizes. This indicates that it may be possible to analytically determine the value of P.

---

[18] The last-level (L3 in this case) cache misses influence the runtime the most because it masks accesses to the main memory [104].

Experiment B

In general, block structured algorithms, especially matrix multiplication, use libraries like the BLAS for matrix operations. Although the previous experiment may be indicative of more general algorithms, there is the question of whether the use of these libraries may eliminate the need to add concurrency for block-structured matrix operations.

Matrix size	BLAS (FG-MPI)			Intel MKL
	P=nfg	O	Time (sec)	Time (sec)
2048	2	8	0.31	0.25
	8	8	0.31	
	32	8	0.44	
	128	8	0.73	
4096	2	8	1.78	1.27
	8	8	1.81	
	32	8	1.88	
	128	8	3.20	
8192	2	8	12.70	8.14
	8	8	11.95	
	32	8	11.43	
	128	8	13.15	
16384	2	8	107.33	58.04
	8	8	86.02	
	32	8	80.03	
	128	8	81.39	

Table 4.5: Effect of added concurrency while using BLAS GEMM routines on a single multicore machine

We implemented a version of the Cannon's Matrix Multiplication algorithm, where FG-MPI is used to decompose the input square matrices into blocks and each of the blocks are multiplied through the ATLAS (Automatically Tuned Linear Algebra Software) BLAS GEMM [152] serial routine. We set O to 8 to take advantage of all the cores on the multicore machine and increased the concurrency (`nfg`) from 2 to 128. Because the number of processes needs to be a square `nfg`=2 is the closest to executing it with just MPICH2.[19] The middle column in Table 4.5 shows that for the smaller matrix sizes using the minimum amount of concurrency is best, however, for larger matrix sizes we see an improvement centered around $P = 32$. Even with the use of the BLAS routines, added concurrency improves performance for larger matrices.

It is also interesting to compare this to other techniques such as the Intel's MKL library that is specific to Intel architectures with a completely different runtime. The MKL library is optimized for the architecture on our Intel Xeon® system and it was compiled with the optimizations enabled. In the last column of Table 4.5, we report the results with Intel MKL's GEMM threaded parallel routine.[20] Although FG-MPI is slower than Intel's MKL, the performance is within 70% of the Intel's optimized runtime. The Intel MKL library is based on OpenMP and runs on one machine whereas for FG-MPI there is the ability distribute computation across multiple nodes for matrix sizes greater than 16K, which may not fit on a single node.

4.2 NAS Benchmarks

The NAS Parallel Benchmarks (NPB2.4) [27] are a set of eight standard benchmarks that are used to evaluate the performance of parallel systems. Each of the eight benchmarks can be compiled for different problem classes (`CLASS`) and the number of MPI processes (`NPROCS`). The problem classes range from `A` (smallest) to `D` (largest). Class `D` was introduced to provide more challenging benchmark sizes for high-performance computer systems that have grown significantly in size and capacity in the last decade. Class `D` benchmark involves about 20 times as much work, and a data set that is approximately 16 times larger than the Class `C` benchmark. The NAS benchmarks do not contain an implementation for `IS` class `D`,

---

[19] The maximum square matrix size tested was 16K due to the amount of memory available on our node.

[20] Threading in Intel MKL is based on OpenMP specification and the GEMM routines used all 8 cores on the node.

and `SP` and `BT` run on a square number of MPI processes. Most of the NAS benchmarks are written in Fortran and we used a tool called Photran [103] to privatize the global variables in the benchmarks.[21] In this section we evaluate the effects of added concurrency on the performance of the NAS benchmarks. For the experiments, we use the cluster described in Section 4.1.

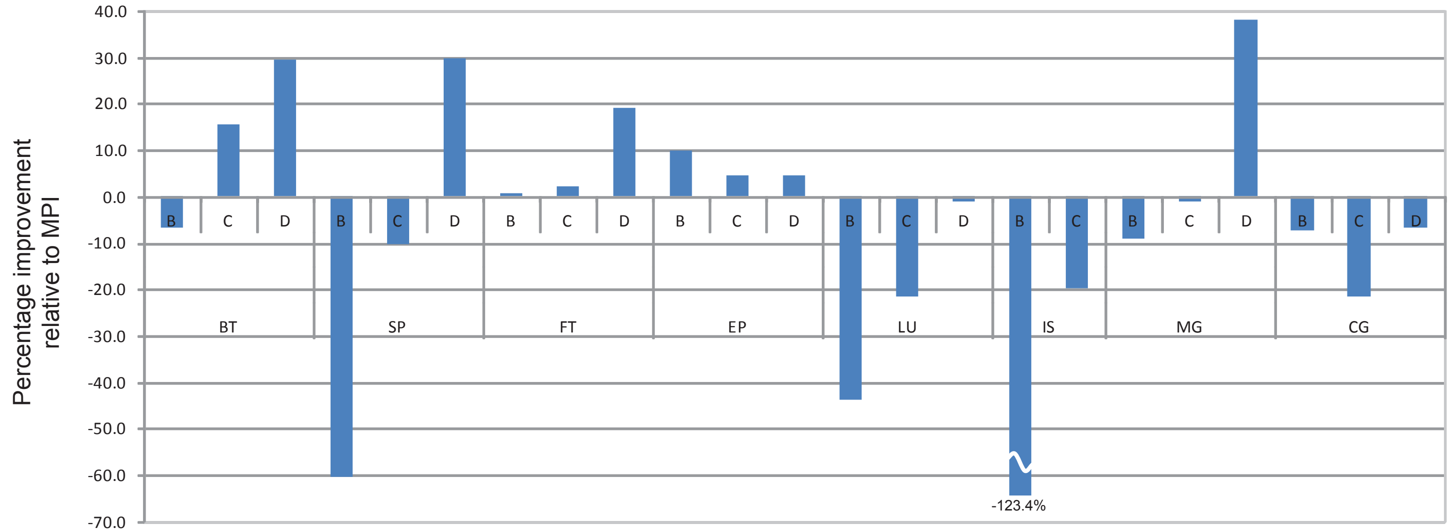
4.2.1 Performance of the NAS Benchmarks

We ran an extensive set of experiments for each of the benchmarks using different `CLASS` and [P;O;M] combinations. We explored the execution space for the 32 problem sizes (4 classes: A,B,C,D for the 8 benchmarks) with P (concurrency) ranging from 1 to 1024, O (OS-processes per node) in the range 1 to 8 and M (number of nodes) varying from 1 to 16. The maximum number of OS-processes (O) per node was fixed to the number of physical cores per node. The mapping of OS-processes to nodes was done in blocks of eight, with the first eight on the first node and next eight on the next, etc. We do not over-subscribe OS-processes to cores for reasons discussed in [66, 88], which report performance degradation for the NAS benchmarks with MPI. Our goal in these experiments is to focus on the effects of added concurrency on MPI performance.

We view `nfg` as a variable to the execution that can be used to adjust cache locality and message sizes independently from the size of the cluster. For each of the benchmark problem sizes, we experimentally varied `nfg` to determine the best FG-MPI performance for different [P;O;M] ($P > 1$) executions, and compared that to the MPI performance achieved with [1;O;M] executions. We present our results by normalizing the MPI performance and reporting the FG-MPI results as a percentage increase or decrease. We roughly characterize problem sizes running under a minute as short-lived and above that as long-lived. For the benchmarks considered, our results show that added concurrency has different effect on short

---

[21] We wish to thank Stas Negara, developer of Photran, who shared the tool with us and answered questions related to it.



MPI time (sec)	15.9	63.2	1354.9	10.9	45.6	1021.8	5.5	17.0	365.0	0.7	2.6	41.4	6.6	18.9	333.8	0.47	2.3	0.3	1.6	33.5	3.9	9.2	229.0
Cores = OxM	64	64	64	64	64	64	64	128	128	128	128	128	128	128	128	32	32	128	128	128	128	128	128
P = nfg	4	4	16	4	4	4	4	4	4	8	8	8	2	2	2	2	8	2	4	4	2	4	4

Figure 4.3: Percentage effect of added concurrency ([P;O;M]) on the end-to-end performance of NAS benchmarks normalized to the best MPI performance [1;O;M]. O=8 per M (machine).

and long-lived problem sizes. Figure 4.3 presents our results.[22] We are omitting results for class A as this is a small problem size that is mostly relevant for testing purposes. Each of the bars represent the percentage effect of added concurrency on the execution time of the benchmarks over the best performance achieved for MPI [1;O;M]. Cores=O×M represent the number of physical cores for the best MPI performance. Concurrency in [P;O;M] executions is the value of P=`nfg` per OS-process for the same values of O and M as above. Figure 4.3 also shows the MPI execution times in seconds to allow differentiation between short and long-lived applications.

4.2.2 Discussion of Results

Figure 4.3 shows a clear trend for short and long-lived applications, which are discussed below.

- For the long-lived benchmarks like BT, SP and FT the performance improvement increases with larger class sizes. The results discussed in Section 4.1.4 on cache behavior corroborate these findings that added concurrency can result in lower working set sizes and better cache locality for general programs with a mix of communication and computation. Also, as shown in Section 4.1, these benefits outweigh the extra overheads due to messaging and context switches.
- We achieve substantial performance improvements of 30% for problem sizes like BT-D and SP-D which execute for more than 15 minutes under MPI and 20% improvement for FT-D which executes for around 6 minutes with MPI.
- The performance improvements for long-lived applications were achieved with concurrency per OS-process that ranged from 4 to 16.

---

[22] The NAS benchmarks do not contain an implementation for `IS` class D.

- For short-lived applications like IS, MG-B, MG-C, CG-B and CG-C, there was performance decrease with added concurrency ranging from 2 to 8. The amount of performance degradation, however, decreased with larger class sizes and for MG-D we see an improvement of 38%. Note that IS and MG both lie towards the lower spectrum of short-lived; i.e., their execution times for class B were a fraction of a second and class C less than 2 seconds. Because of the small data size and the relatively short execution time, the opportunity to take advantage of cache locality and smaller message is slight and, as expected, there is only the added costs of the added concurrency. The short execution time magnifies the overheads in terms of relative performance. In absolute terms, the degradation ranges from 0.03 to 1.99 seconds, however, in this case there is no advantage to using added concurrency.

- EP (Embarrassingly Parallel) application is computationally intensive and was mostly agnostic to added concurrency. We saw modest improvements of around 4% with this benchmark.

- LU-D was an exception to the performance improvement trend seen for long-lived applications. LU-C which ran in around 18 sec and LU-D which ran in 336 sec (5.6 mins) with MPI, showed a performance decrease of 21% and 1% respectively. The amount of degradation, however, was substantially lower for the larger class D. One reason that the LU benchmark behaves differently may be due to the inherent load imbalance of the application [154]. It is also more communication intensive compared to the other benchmarks and as hypothesized in [154], large synchronization time due the load imbalance may mask the benefits of added concurrency.

- The CG benchmark showed an interesting behavior in terms of resource utilization, which is discussed below.

There are two performance benefits to using FG-MPI that are not shown in Figure 4.3. The first one is due to the decoupling of the problem from the hardware. For benchmarks like BT and SP that have a restriction that the number of MPI processes be square, it is possible for us to run on a non-square number of cores (e.g. 128) as long as the P×O×M product is a square. The performance improvements with [P;8;16] over MPI [1;8;8] go up significantly, ranging from 36.7% to 63.5%, with SP and BT, class sizes C and D.

Another interesting result is for the CG benchmark. We noticed FG-MPI can achieve performance comparable to MPI for CG while using fewer cores. That is, FG-MPI provides better resource utilization while achieving similar execution times.

CLASS		P=nfg	O×M	Time (sec)
A	MPI	1	32	0.14
	FG-MPI	2	32	0.17
B	MPI	1	128	3.92
	FG-MPI	2	64	4.03
C	MPI	1	128	9.17
	FG-MPI	2	64	9.35
D	MPI	1	128	229.5
	FG-MPI	8	64	271.5

Table 4.6: Comparing MPI and FG-MPI for CG benchmark.

Table 4.6 shows the results for CG, where FG-MPI uses half the number of cores (O×M) as compared to MPI. This indicates that communication costs may be the limiting factor for CG and keeping it more localized helps.

In conclusion, for the benchmarks considered, added concurrency showed substantial benefits for the long-lived applications and larger problem sizes. For short-lived applications and more communication intensive applications FG-MPI did not provide any improvements. FG-MPI makes it easy to add concurrency to the runtime execution of MPI programs, since it is a runtime parameter it can be adjusted, without re-compilation, to different problem sizes and one can always omit the `nfg` option when its use is not indicated.

4.3 Summary

A problem is often mapped onto MPI processes based on the number of physical processors available, with one process assigned to each processor. Often we come across algorithms that will only run on a certain number of processors. This approach inhibits a natural decomposition of the problem that may be more simply described, algorithmically, using finer granularity. From a software engineering point of view, it is important to decouple the problem decomposition from the number of processing units.

In this chapter we have shown the effectiveness of decoupling the MPI processes from the underlying hardware. We measured the benefits of added concurrency for cache awareness and message sizes and described ways in which FG-MPI minimizes the resulting overheads for context switching and communication on multicore machines. Using the NAS benchmarks we showed that substantial performance gains are possible on long-lived benchmark programs where the gains out-weigh the overheads. For existing MPI programs, FG-MPI provides a flexible runtime parameter to improve cache locality and message-size independently from the number of cores and machines in the cluster.

Chapter 5

Concurrency >> Parallelism

> Controlling complexity is the essence of computer programming.
>
> — Brian Kernigan

Concurrency is a tool for structuring programs [126]. Concurrency is an application level property and a programming environment that supports concurrency allows a program to be expressed as a set of tasks and the synchronization relationships between them. Parallel execution is a property of the machine and the mapping of concurrent tasks onto physical processors defines which tasks are executed in parallel and which are interleaved on the same processor.[23] Many real world problems are concurrent and expressing additional concurrency in a program can often simplify the problem by matching the processes to the program structure instead of the hardware [126, 150]. In this chapter we explore the use of added concurrency per OS-process to enable function-level parallelism in FG-MPI.

There are a number of popular parallel programming languages for multicore [11, 48, 105] that use message passing. One notable difference between MPI and these parallel languages is the granularity of the MPI processes. Processes in MPI are coarse-grained and programmed to make it easy to match the number of processes to the available hardware, whereas many parallel languages support finer grain to match processes to the structure of the program. By fine-grain we mean function-level parallelism where processes may have

---

[23] Sebesta [129] defines two categories of concurrency: (a) Physical concurrency that is characterized by multiple independent processors and multiple threads of control and, (b) Logical concurrency where concurrent units are executed in an interleaved manner by time-sharing a single processor.

tens of instructions rather than the thousands of instructions in coarse grain program-level parallelism. One can have function-size programs in MPI but it is not done because oversubscribing processes to nodes is inefficient due to the context switch time between OS-level processes and because the OS scheduler is unaware of the cooperative nature of the MPI processes.

In the first two sections of this chapter we examine how added concurrency in FG-MPI can be used to structure parallel programs. In Section 5.1, we use FG-MPI to implement an application from the CoSMoS [31] project that models emergent behavior and compare the results with several popular fine-grain parallel languages on a multicore system. In Section 5.2, we use FG-MPI to re-structure a typical use of non-blocking communication in MPI and show that having multiple MPI processes per OS-process, with a runtime scheduler, can be used to simplify MPI programming and achieve performance without adding complexity to the program. Lastly, in Section 5.3, we test the ability of FG-MPI to scale to massively parallel programs and present the results of running over a 100 million MPI processes on the WestGrid [50] computing facility.

5.1 Comparison with Fine-grain Multicore Languages

In this section we compare the performance of the FG-MPI implementation with Pthreads and three other languages that support fine-grain parallelism: Erlang, Haskell, and Occam-pi [106]. We wanted to show that it is not only possible but easy to express an application, requiring thousands of fine-grain tasks, through MPI processes. To make the test concrete, we have intentionally chosen an application with the type of fine-grain parallelism favourable to languages like Erlang, Haskell and Occam-pi in a shared memory environment. In fact, this application was developed for process oriented programming and presented in [122] and its dynamic nature is well suited for the automatic process migration strategies supported in the runtime system for Occam-pi. As well, all tests were run on a shared memory system, which favoured the concurrency techniques used in some of the languages. We describe the application in the next section.

5.1.1 Simulation of Flocking Behavior

The application is based on modelling of emergent behavior as part of the CoSMoS [31] project and a benchmark has been developed based on occoids (Occam-pi processes) to simulate bird flocking behavior. A detailed description of the benchmark can be found in [122] and its source code, implemented in different languages, is available at [107].

In this application, a number of birds move in a space represented by a two-dimensional torus. Each grid point in the torus represents a location and the position of birds is defined with respect to the center of their current location. The birds are able to see each other within nine adjacent locations. Using an internal bias that produces randomized behavior, the birds calculate a repulsive force from all those visible and use that force to decide if they should move to a new location or stay at the current one. The bias depends on the position of the bird and how many other birds are visible to it.

We developed an MPMD MPI application to simulate the bird flocking behavior. From the implementation perspective, each bird is represented as an MPI process. Each grid point on the two-dimensional torus is a location process and it maintains information about all the birds that are currently present at that location. Another process called view is also associated with each location, and all the birds at that location communicate with it. At each simulation step, the birds query the view associated with their current location for information about visible birds at the adjacent locations. The view communicates with all the adjacent locations and compiles an aggregated list of birds and sends that to the querying birds. That information is then used by the birds to calculate the repulsive forces and decide if they should move to another location or not. In either case, the birds communicate their decision to their location process, so that it can update its information.

Barrier synchronizations are used between communication phases in each simulation step, so that each bird has a consistent view of the environment.

5.1.2 Test Setup

Our test setup consists of a single socket, 4-core Intel® Core™ i7 Nehalem workstation. The system has 6 GB of memory and the processors run at 2.67GHz. The workstation runs Linux kernel 2.6.28-15-Generic with hyper-threading enabled.

The following versions of the software for the languages reported in our test results were used. We used the Glasgow Haskell Compiler (GHC) version 6.10.4 for multicore systems and compiled with optimization options. The Haskell code was run with the `+RTS -N` option so that it could run on multiple CPUs. We experimented with different values for `N`, the number of threads, and `-N8` gave the best results. Erlang version 5.6.5, both with and without HiPE [113], and the POSIX threads interface available through GNU C version 4.3.3 were used. For Occam-pi, we used the KRoC compiler version 1.5.0-pre5 with the CCSP runtime system. All tests were run to take advantage of the thread-level parallelism available on all processors with hyper-threading. For FG-MPI tests, we evenly distributed the fine-grain processes across eight OS-processes.

5.1.3 Performance Results

The application was run for different grid sizes and in each case the mean value of three independent test runs is reported. The measurements were stable with very little variation across different runs.[24] The runtime is reported for a total of 1024 simulation steps. The number of birds at each location at the start of the simulation is twelve. Table 5.1 shows the number of MPI processes with increasing grid sizes.

---

[24] The Pthreads application runs showed some variation at certain grid points. In those cases the mean of a total of 6 independent runs was taken.

$$(G \times G) \times (B + 2)$$

where G is the grid size and B is the initial number of birds in each grid area. Sample experiments for $B = 12$ included:

Grid size	MPI processes
5x5	350
10x10	1,400
15x15	3,150
20x20	5,600

Table 5.1: Number of MPI processes with increasing grid sizes.

Figure 5.1 compares the performance of FG-MPI implementation with the other languages. As the figure shows, FG-MPI outperforms Pthreads, Haskell and Erlang and is competitive with Erlang-HiPE. The only language that out-performed FG-MPI is Occam-pi. For this application, there are a number of optimizations that the other implementations use, which ours does not. Firstly, the FG-MPI application communicates through standard message passing routines and each process owns its data. In Pthreads, for example, the application maintains a linked list of the birds at each location and communicates that by simply passing pointers. There is no ownership of data as one thread can change variables used by another. We do not use zero-copy routines in this test and a sending process constructs a message, which is then copied into the receive buffer of the receiver. Our MPI application also does not use any grid resizing strategies to adapt to the dynamic nature of this application. Secondly, Haskell, Erlang, Pthreads, and Occam-pi all use compiler optimizations, which we do not. The performance difference between Erlang with and without HiPE shows the potential for compiler optimizations.

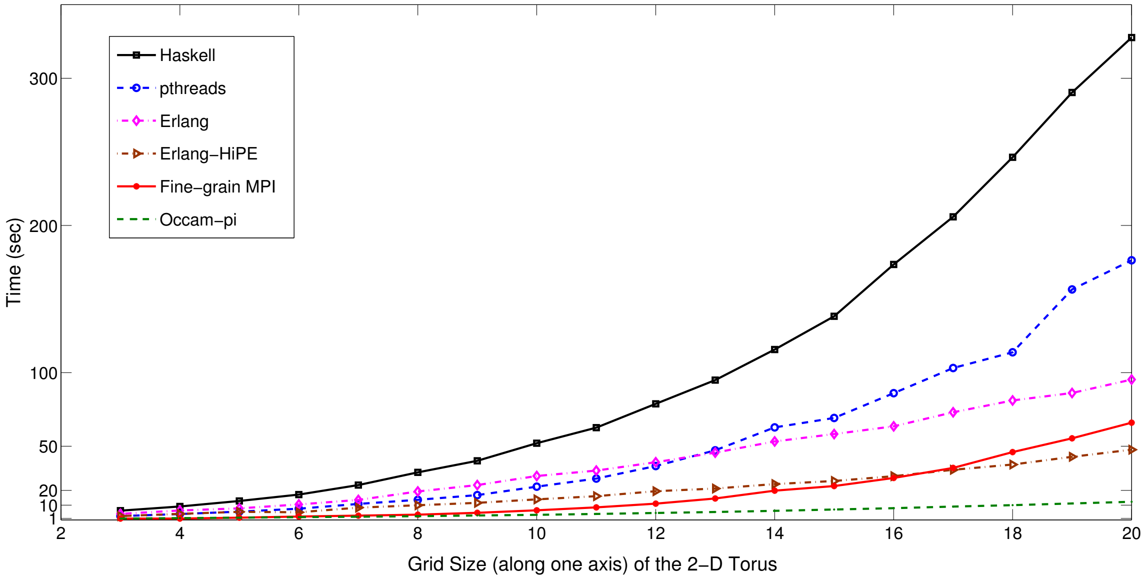


Figure 5.1: Comparison of the execution times of the bird flocking application for different grid sizes.

Despite these optimizations by the other implementations, FG-MPI does better than many of them. For example, for the `15x15` grid size, Erlang takes `58.28` seconds, Erlang-HiPE takes `26.42` seconds while FG-MPI takes `22.95` seconds. The run times for Pthreads (`69.23`) and Haskell (`138.27`) are much higher. The code for MPI processes has a low context switching overhead and scales well.

The Pthreads results in Figure 5.1 show that the synchronization overheads of using locks and condition variables quickly become very large and it does not scale. Haskell performs even worse and as mentioned in [45], much work needs to be done on optimizing placement of threads.

The language that consistently did better than FG-MPI is Occam-pi. They employ a highly optimized runtime system and scheduler. The KRoC compiler provides support for the language features and optimizations for code generation specific to Occam-pi. Occam-pi results clearly demonstrate the performance potential of using cooperating non-preemptive light-weight processes. Unlike Occam-pi, MPI is a library not a language, however, code transformations to improve performance of MPI programs through compiler optimizations is an active area of research and the types of techniques used in Occam-pi may benefit MPI in future [33, 116].

We ran all tests on a single shared memory multicore machine. This setup is especially favourable to the implementations we compared against. FG-MPI, however, has the benefit that the same application can execute in a distributed memory environment without any code modifications. The other languages require modification to their codes to enable them to run in a distributed environment.

In a cluster environment, where the program is distributed between cores on one machine as well as across machines, MPI has the advantage of performing well over a wide variety of network fabrics in various cluster environments. Given the many years worth of effort in ensuring performance portability of MPI programs in distributed environments we expect MPI programs to perform better in comparison to a multicore language with added on network support.

5.2 Programmability and Non-blocking Communication

One technique widely used in MPI programs that adds to their complexity is the use of non-blocking communication. Non-blocking communication makes it possible to have multiple outstanding messages that increases asynchrony and allows one to overlap communication with computation. This can reduce the idle time that results when processes are blocked waiting for a message to arrive. To avoid idle time the programmer tries to post messages as soon as possible, overlap that with some computation while periodically checking for new messages to process as well as posting new ones. Optimizing the messaging in this manner to reduce idle time and increase “slackness” breaks the cohesion of the program structure, adds complexity, and is less portable with respect to performance. As well, in MPI programs written as a collection of modules, there is no simple way to schedule communication with computation without exposing the module's functionality. In this section we describe how FG-MPI can be used to reduce reliance on non-blocking communication without adding complexity to the code.

```
int main( int argc, char *argv[] )
{  ...
  MPI_Irecv(...,recvRequests[0]);
  MPI_Irecv(...,recvRequests[1]);
  do {
    compute_local(...);
    MPI_Waitany(2, recvRequests, &index, recvStatus);
    switch(recvStatus->MPI_TAG) {
    case tag1:
      compute_A();
      MPI_Send(...);
      MPI_Irecv(...,recvRequests[index]);
      break;
    case tag2:
      compute_B();
      MPI_Send(...);
      MPI_Irecv(...,recvRequests[index]);
      break;
    }
  }while(...);
}
```

Listing 5.1: Scheduling communication and computation by non-blocking operations

Non-blocking communication typically involves structuring the code into stages and scheduling these stages inside the application code. Exposing and scheduling even a modest number of stages in the application can result in complex code that is difficult to read and maintain [54, 55, 95]. One of the advantages of our approach is that it reduces the need for non-blocking communication.

Consider the program in Listing 5.1, showing a simple use of non-blocking communication, which tries to post as many messages as possible to keep the process busy. There are three main parts to the program: (a) allocating and managing message request buffers, (b) checking for message completions and then processing the messages, (c) a compute part that may or may not depend on the messages sent and received. Some of the complexities in Listing 5.1 are:

(i) The compute and communication parts of the code are interleaved and the programmer needs to balance the computation with the polling of the link via the middleware.

(ii) The user needs to manage the request buffers for the multiple outstanding messages. The programmer also needs to be aware of all the different types of outstanding messages and how messages are matched. This often results in the use of `MPI_ANY_SOURCE` and `MPI_ANY_TAG`.

With FG-MPI, as shown in Listing 5.2, we can achieve a similar overlap by re-organizing the code fragment into three smaller processes: `compute_local()`, `process_A()` and `process_B()`. As opposed to Listing 5.1, there are no non-blocking requests and associated structures in Listing 5.2 and no need to remember that the posted requests have to be checked for completion. Listing 5.1 has requests that are global over the entire program and no clear demarcation between different types of requests. FG-MPI places all of corresponding computation and communication code pertaining to one activity into one process. This adds to the cohesiveness of the program and makes it easier to read and change the code.

The purpose of the control loop in Listing 5.1 is to schedule different parts of the code based on the message events from `MPI_Waitany()`. In the FG-MPI version of the code there is no `MPI_Waitany()`. The control loop is now handled by the FG-MPI scheduler, which

```
int main( int argc , char *argv [] ){
    FGmpiexec(&argc , &argv , &binding_func );
    return (0);
}
int process_A( int argc , char** argv ){
    MPI_Init(...);     ...
    do{
      MPI_Recv(...,tag1 ,...);
      compute_A();
      MPI_Send(...);
    }while(...);
    MPI_Finalize();
}
int process_B( int argc , char** argv ){
    MPI_Init(...);    ...
    do{
      MPI_Recv(...,tag2 ,...);
      compute_B();
      MPI_Send(...);
    }while(..);
    MPI_Finalize();
}
int compute_local( int argc , char** argv ){
    MPI_Init(...);     ...
    do{ ...
      if (...) MPIX_Yield();
    }while(...);
    MPI_Finalize();
}
```

Listing 5.2: Defining MPI processes as concurrent functions all mapped to the same OS-process. Each MPI process also calls `MPI_Init` and `MPI_Finalize`.

acts as an abstraction device, so that the programmer does not have to hand-code it into the program.

In Listing 5.2, should `process_A()` now require we receive two messages rather than one, we only need to add another `MPI_Recv()`, however, for Listing 5.1 there are questions as to whether we need to introduce another case and tag and how it might be matched. In both listings it is important that the `compute_local()` code invoke the progress engine sufficiently often to not unduly delay the remaining computation and communication. In Listing 5.2, `MPIX_Yield()` can be appropriately placed when needed to provide an explicit de-scheduling point that automatically resumes at the proper place. In Listing 5.1, changing the rate at which the network is polled requires reorganizing the computation, which is yet another complication.

Expressing additional concurrency in the program gives us the opportunity to exploit it, however, it does require additional effort to structure the code and map MPI processes to functions using a MPMD process model. In Listing 5.2, we encapsulate different receive actions in separate MPI processes and the sender process needs to use the appropriate receiver process's rank to trigger the right computation. As discussed in Section 3.4 (page 61), `FGmpiexec` spawns the co-located MPI processes and the mapping of process ranks to functions is specified through the user-defined `binding_func`, which takes as input the `MPI_COMM_WORLD` rank of a process and returns a pointer to the function that the process is bound to. The extra-level of mapping gives us more flexibility in mapping to OS-processes and cores. As well, we can match the OS-processes to the cores to minimize the effect of OS-noise [41] and not rely on the OS scheduler, which introduces yet another control loop that is unaware of the synchronization between the MPI processes. Finally, FG-MPI extends MPI so the programmer can manage as little or as much of the non-blocking communication as they wish.

We created a benchmark program, similar to the codes in Listings 5.1 and 5.2, to evaluate the effect of exposing function-level parallelism and the overhead of the scheduler in FG-

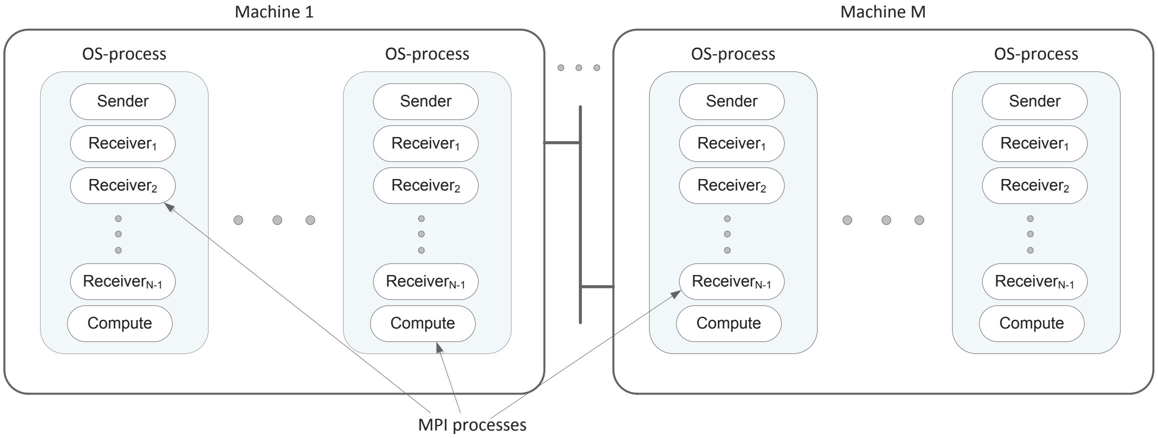


Figure 5.2: Mapping of one sender process, $N - 1$ receiver processes and one compute process to each OS-process in the FG-MPI code of the benchmark.

MPI. The MPI code of the benchmark launches N MPI processes as OS-processes, each of which uses non-blocking `MPI_Isend` and `MPI_Irecv` calls to pre-post send and receive requests for all the other processes. The MPI code carries out some local computation and then calls `MPI_Waitany` on these requests to progress them. The local computation and progression of the requests is done in a loop until all the requests are completed. The FG-MPI code also launches N OS-processes, however, each of these OS-processes contain $N+1$ co-located MPI processes (see Figure 5.2). The MPI processes within each OS-process are organized as follows. (A) There is one MPI sender process that calls `MPI_Send` $N - 1$ times to send to one receiver in each of the N OS-processes. (B) There are $N - 1$ receiver processes that call `MPI_Recv`. Each of these receivers is matched with the sender in one of the OS-processes. (C) There is one compute process that does local computation similar to `compute()` function in Listing 5.2. To allow computation and communication overlap, all messages sent in this benchmark are long messages that use the rendezvous protocol in MPICH2. The MPI code introduces asynchrony by pre-posting non-blocking operations and then manages scheduling of the computation and the communication through explicit calls to the middleware via the `MPI_Waitany` calls. On the other hand, the FG-MPI version restructures the code so that there is a separate MPI process for each of the receive calls. The scheduling of computation and communication is outside of the application specification and is managed by the runtime scheduler. In order to isolate the effects of introducing additional concurrency by mapping MPI processes to functions, we did not introduce any dependency between the computation and the communication in this benchmark.

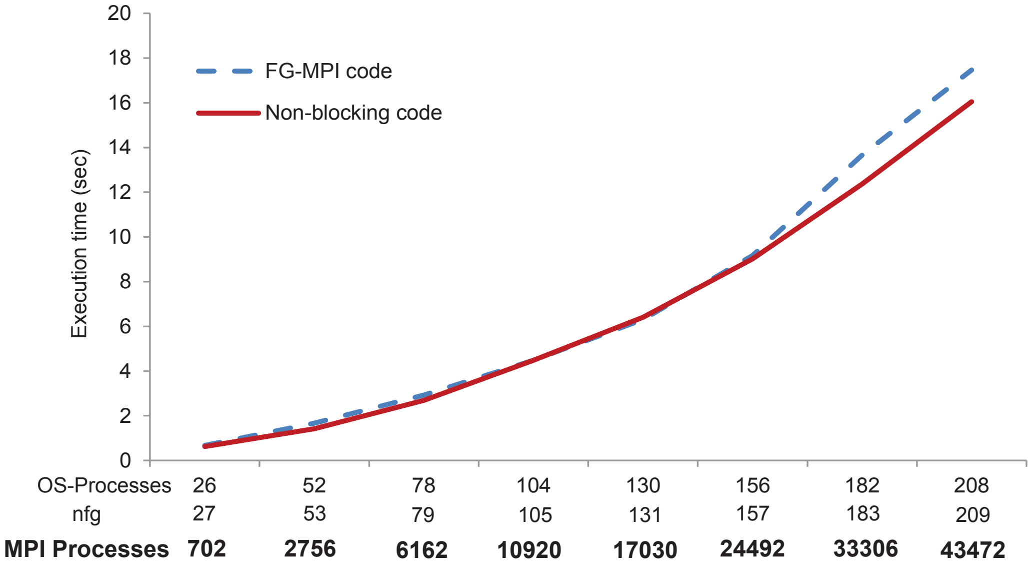


Figure 5.3: Performance comparison of non-blocking code using `MPI_Waitany` with function-level parallelism in FG-MPI. Number of OS-Processes is same in both cases.

Figure 5.3, shows the results of this benchmark. For the MPI non-blocking code, the number of MPI processes are equal to the number of OS processes, while in the FG-MPI code the number of MPI processes are a multiple of the `nfg` parameter and the number of OS-processes. The time reported is for ten iterations of the benchmark. Our results show that even with the introduction of more than 24,000 fine-grain MPI processes compared to 156 coarse-grain processes, the performance remains the same.

As the number of MPI processes increase beyond this to more than 43,000 processes, there is a small overhead of 8.7%. We are not sure of the reason for the deviation after 24,000 processes, however, the MPI code has an advantage in this benchmark, since it pre-posts all the send requests. Multiple of these pre-posted send requests can be progressed in the `MPI_Waitany` call. The FG-MPI code has a single sender process which makes `MPI_Send` calls one after the other. This benchmark stress tests asynchrony at a large scale and shows that the overhead incurred by exposing function-level parallelism remains low.

5.3 A 100 million MPI processes on Westgrid

The past few years have seen rapid advances and growth in computer hardware. The latest TOP500 [144] list reports systems with more than 1,500,000 cores in operation today. An interesting statistic on the current state of the art is a brain simulation experiment[25], that was run on the world's second fastest supercomputer at the Lawrence Livermore National Laboratory (LLNL). This experiment was executed on 1,572,864 processor cores using 98,304 MPI processes with 64 threads per process (a total of 6,291,456 threads) [120, 155]. Development is underway of exascale systems with a 100 million cores that are capable of executing an exaflop and are expected by the end of this decade. These systems bring with them significant challenges for the development of software that can scale and exploit the computing power of these systems. Development of software models for these systems is an open and active area of research [1].

MPI faces a number of challenges in this changing parallel computing environment and key among them are issues of scale [12, 55, 92, 139]. We wanted to test the scalability of FG-MPI to see whether we would be able to execute a 100 million MPI processes on thousands of cores. In collaboration with the Western Canada Research Grid (WestGrid)

---

[25] This work was among the six finalists for the Best Paper Award at Supercomputing 2012 (SC12).

computing facility[26], we were able to access a large number of cores and ran a series of experiments in several stages and successfully scaled to over a 100 million MPI processes using 6,480 processor cores [63, 151].[27]

Our test setup on Westgrid consisted of a cluster of 540 nodes, where each of the nodes is a dual-socket 6-core (12 cores per node) Intel Xeon X5650, 64-bit machine, running at 2.67 GHz, connected by an InfiniBand network and runs IPoIB on the QDR IB links. All nodes have 24 GB of memory and run Linux kernel 2.6.18-194.el5.

We ran a number of programs that used both point-to-point and collective MPI routines to test the scalability and performance of FG-MPI. These experiments included a pi calculation program that uses `MPI_Bcast` and `MPI_Reduce`, and a `MPI_Barrier` benchmark. Each of these programs were run on 6,480 cores where the number of OS-processes was equal to the number of cores, with 16,000 co-located MPI processes in each OS-process. The total number of MPI processes in each application was thus 16,000 x 6,480 = 103,680,000.

The pi-calculation program took 9.84 seconds to execute. This includes the time for `MPI_Bcast`, local computation and `MPI_Reduce` operations on `MPI_COMM_WORLD`. In this experiment communication takes place at all levels of the communication hierarchy, i.e., within co-located MPI processes, between MPI processes in different OS-processes on the same node and across different nodes.

Our location-aware implementation of `MPI_Barrier` is optimized for the three levels of the communication hierarchy (Section 4.1.3). The time to do a barrier operation on 103 million MPI processes took only 0.207 seconds.

In the absence of exascale machines, our tests approximated execution of massively parallel programs. It shows that FG-MPI's approach allows the basic routines in MPI and in particular the MPICH2 middleware to scale to 100 million processes - something that

---

[26] We wish to acknowledge the help of WestGrid's support staff Roman Baranowski and Brent Gawryluik in setting up our experiments

[27] FG-MPI's 100 million MPI processes experiment was featured on the HPCwire and WestGrid websites as well as on the Argonne National Laboratory's MPICH homepage.

no other system has demonstrated. As bigger systems become available, FG-MPI makes it possible to explore the performance and scalability issues of the MPICH2 middleware. As well, it will enable the development of algorithms and applications that can benefit from millions of processes without requiring as many cores.

5.4 Summary

Fine-Grain MPI (FG-MPI) supports function-level parallelism by having multiple MPI processes per OS-process. We compared FG-MPI with several commonly used fine-grain multicore languages. Our experimental results show that for a highly dynamic fine-grain application, well-suited to these languages, FG-MPI outperformed Pthreads and Haskell, and achieved similar performance to that of Erlang.

We gave an example of using FG-MPI to re-structure a typical use of non-blocking communication in MPI. FG-MPI enables a task-oriented programming approach and support for MPMD that makes it easier, by exposing more concurrency, to overlap communication with computation. This relieves the programmer from scheduling computation and communication inside the application and focus on what needs to be scheduled rather than how to manage it.

Lastly, we demonstrated that FG-MPI's approach can scale to hundreds of millions of MPI processes and reported the results of our experiments on Westgrid.

Chapter 6

Conclusions

> What we call the beginning is often the end. And to make an end is to make a beginning. The end is where we start from.
>
> T.S.Elliot

One of the main challenges of the FG-MPI project involved integrating it into the MPICH2's middleware. Our goal from the beginning of this project was to have an impact in the HPC community through extending an existing open source implementation of MPI, rather than write our own library or layer it on top of another runtime system. There were a number of challenges in integrating FG-MPI into an existing production-quality middleware and it was not known what problems we would encounter. In addition to the major design issues like decoupling of the namespace, extension of the runtime and scalability, there were a host of low-level system details that had to be addressed including extension of the message format, sharing of MPI structures, management of two-level process ranks, interfacing with external libraries, and development of build and debug subsystems. Based on our experience we believe this type of integration is possible with other implementations of MPI. In the next sections I discuss some of our experiences in using FG-MPI and future work.

6.1 Our Experiences with FG-MPI

6.1.1 Vehicle to Investigate Scalability Issues

We have found FG-MPI to be a useful vehicle for investigating and testing scalability issues of the MPI middleware. During its development, I identified a number of structures in the MPICH2 middleware which did not scale, such the tight-coupling of the process names and the routing tables, the creation and storage of process maps and communicators and the message envelope format and matching. FG-MPI's ability to expose large-scale concurrency allows us to explore different designs and algorithms inside an MPI implementation without access to a facility with hundreds and thousands of processing cores. The are a number of issues at scale related to the MPI API, the middleware implementation and the standard itself, that are highlighted by Balaji *et. al.* in [12]. These are challenging issues and we believe FG-MPI can help in development and testing of scalable algorithms to address some of these issues.

From the application perspective, it enables programmers to develop and test applications that can scale to millions on their laptops or desktops without requiring as many physical cores. Access to resources in a supercomputing center can often be limited, time consuming and costly. This allows testing on a smaller system before deployment on a larger one.

6.1.2 Deterministic Execution

Debugging parallel programs is notoriously difficult, and their non-deterministic nature makes it hard to predict interactions or reproduce errors [91, 123]. Programmers can benefit from the ability to restrict the amount of non-determinism and test with a limited number of program interleavings [86, 134]. FG-MPI provides the ability to flexibly move the boundary of the interleaved concurrency and parallelism through command-line, without recompilation. The selection of scheduler can, as well, be specified on the `mpiexec` command-line and it is possible to test the execution of the program with different schedulers. This helps to the control the program interleavings and judiciously introduce non-determinism for testing. We have found it particularly useful to run all the MPI processes co-located inside a single OS-process for detecting program safety issues such as deadlock. We have also found using a deterministic scheduler like round-robin as a useful tool to reproduce program executions.

6.1.3 Porting of Existing Programs

The boiler-plate code described in Section 3.4 was developed to simplify porting of existing MPI programs to FG-MPI. In addition to the NAS benchmarks we have ported a number of libraries to test the ease of porting existing programs to FG-MPI. One of those libraries is MR-MPI [115] library from Sandia National Laboratories, which is an open source implementation of the MapReduce model popularized by Google [35]. MR-MPI is a very well structured library, and very few source code changes were required to use it with FG-MPI programs. We successfully ran FG-MPI programs using this library and were able to specify and control the mapping of fine-grain processes by a simple invocation of the `nfg` parameter on the command-line. Our experience has been that, barring the removal of global and static variables in the existing code, porting is straight forward and requires the addition of only a few lines of the boilerplate code discussed in Section 3.4. A number of tools [2, 114] are available for automatic privatization of global variables that can help reduce the programmer's effort.

6.1.4 Matching the Processes to the Program Structure

Expressing additional concurrency in a program can often make it simpler and lead to greater efficiency and enhanced opportunities for structuring of parallel programs [126]. We have looked at a number of applications that would normally not be implemented using MPI due to its coarse-grained nature. These include the bird flocking application presented in Section 5.1 as well as graph applications where each node is modelled as an MPI process. Our experience has been that the ability to match MPI processes to the program structure simplifies the program logic. Over the past two years, FG-MPI has been used by other Master's students at UBC in their theses work to develop novel program designs. These include design of a CSP-like (Communicating Sequential Process) program that uses MPMD-style of programming where multiple processes communicate using Pilot [26], an MPI library that provides CSP-like channel abstraction [19]. Another work involved investigation of different scalable storage representations for process maps using FG-MPI as a testbed [98]. Currently, work is being done on the design and implementation of concurrent data structures using FG-MPI. We are also experimenting with new ways to support dynamic processes in MPI by leveraging our runtime scheduler.

6.2 Moving Forward

Our hope is that the FG-MPI design and its proof of concept in a working system may provide a way for other MPI implementations to augment MPI to support this fine-grain model. Secondly we hope, by way of illustration in this thesis, that extending MPI's execution model to fine-grain can make MPI programming easier and a better overall solution that can scale from a multicore node to multiple machines in a cluster. The potential benefits of our fine-grain approach are for multicore as the number of cores increase. As well, it is based on message-passing and it will be portable to multicore nodes with or without support for cache-coherence.

The success of MPI is due in a large part to the MPI Forum's focus on concisely defined features and a forward-looking view of the parallel computing landscape [55]. MPI faces a number of challenges as the needs of parallel computing change in ways that are fundamentally different from the past. The MPI Forum continues to introduce enhancements to the standard and has recently approved the MPI 3.0 standard and the Argonne National

Laboratories (ANL) has released its latest version of MPICH2. The Forum has always followed a formal practise of seeking input from library implementers, vendors and users prior to introducing any enhancements into the standard. The enhancements are based on validation through a working implementation and the strength of the use-cases. Through our work on FG-MPI, we hope to work closely with the MPI Forum and the MPICH2 team at ANL and integrate FG-MPI in the latest version for release to the research community.

Bibliography

[1] 10^{18} - International exascale software project. Available from `http://www.exascale.org/iesp/Main_Page`, accessed on March 4, 2013.

[2] Elsa: An elkbound based C++ parser. Available from `http://www.cs.berkeley.edu/~smcpeak/elkhound`, accessed on March 4, 2013.

[3] Stackless Python. Available from `http:http://www.stackless.com`, accessed on March 4, 2013.

[4] XPMEM: Cross-Process Memory Mapping. Available from `https://code.google.com/p/xpmem/`, accessed on March 4, 2013.

[5] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management or, event-driven programming is not the opposite of threaded programming. In *Proceedings of the 2002 Usenix ATC*, 2002.

[6] Francesc Alted. Why modern CPUs are starving and what can be done about it. *Computing in Science and Engineering*, 12:68–71, 2010.

[7] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.*, 10(1):53–79, February 1992.

[8] Argonne National Laboratory. Communicators and Context IDs. Available from `http://wiki.mcs.anl.gov/mpich2/index.php/Communicators_and_Context_IDs`, accessed on March 4, 2013.

[9] Argonne National Laboratory. MPICH2: A high performance and portable implementation of MPI standard. Available from `http://www.mcs.anl.gov/research/projects/mpich2/index.php`, accessed on March 4, 2013.

[10] Argonne National Laboratory. MPICH2: Performance and portability. MPICH2 flyer at SC07. Available from `www.cels.anl.gov/events/conferences/SC07/presentations/mpich2-flyer.pdf`, accessed on March 4, 2013.

[11] Joe Armstrong, Bjarne Dacker, Thomas Lindgren, and Hakan Millroth. Erlang - White Paper. Available from `http://erlang.org/white_paper.html`, accessed on March 4, 2013.

[12] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing L. Lusk, Rajeev Thakur, and Jesper Larsson Träff. MPI on a million processors. In *PVM/MPI*, pages 20–30, 2009.

[13] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. Toward efficient support for multithreaded MPI communication. In *Proc. of the 15th Euro PVM/MPI Users' Group Meeting*, pages 120–129, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] Pavan Balaji and Dave Goodell. Using 32-bit as rank. Available from `https://trac.mcs.anl.gov/projects/mpich2/ticket/42/`, accessed on March 4, 2013., 2008.

[15] Victor R. Basili, Jeffrey C. Carver, Daniela Cruzes, Lorin M. Hochstein, Jeffrey K. Hollingsworth, Forrest Shull, and Marvin V. Zelkowitz. Understanding the high-

performance-computing community: A software engineer's perspective. *IEEE Softw.*, 25(4):29–36, 2008.

[16] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[17] Punit Bhargava. MPI-LITE: Multithreading support for MPI. Available from `http://pcl.cs.ucla.edu/projects/sesame/mpi_lite/mpi_lite.html`, accessed on March 4, 2013., 1997.

[18] G. Bronevetsky, D. Quinlan, A. Lumsdaine, and T. Hoefler. Compiled MPI: Cost-effective exascale applications development. *Lawrence Livermore National Laboratory Technical Report*, April 2012.

[19] Cody R. Brown. Supporting a process-oriented model in MPI through fine-grain mapping. Master's thesis, University of British Columbia, April 2012.

[20] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[21] D Buntinas, W Gropp, and G Mercier. Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. In *Proc. of the Sixth IEEE Intl. Symp. on Cluster Computing and the Grid (CCGRID)*, pages 521–530, Washington, DC, USA, 2006. IEEE Computer Society.

[22] Darius Buntinas, Brice Goglin, David Goodell, Guillaume Mercier, and Stephanie Moreaud. Cache-efficient, intranode, large-message MPI communication with MPICH2-Nemesis. In *ICPP*, pages 462–469. IEEE Computer Society, 2009.

[23] Darius Buntinas, Guillaume Mercier, and William Gropp. Data transfers between processes in an SMP system: Performance study and application to MPI. In *Proceedings of the 2006 International Conference on Parallel Processing*, ICPP '06, pages 487–496, Washington, DC, USA, 2006. IEEE Computer Society.

[24] Darius Buntinas, Guillaume Mercier, and William Gropp. Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the Nemesis communication subsystem. *Parallel Comput.*, 33(9):634–644, 2007.

[25] Patrick Carribault, Marc Pérache, and Hervé Jourdren. Enabling low-overhead hybrid MPI/OpenMP parallelism with MPC. In Mitsuhisa Sato, Toshihiro Hanawa, Matthias S. Müller, Barbara M. Chapman, and Bronis R. de Supinski, editors, *IWOMP*, volume 6132 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2010.

[26] J. Carter, W.B. Gardner, and G. Grewal. The pilot approach to cluster programming in C. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, April 2010.

[27] NASA Ames Research Center. Numerical aerodynamic simulation (NAS) parallel benchmark (NPB) benchmarks. Available from `http://www.nas.nasa.gov/Software/NPB/`, accessed on March 4, 2013.

[28] Márcia C. Cera, Guilherme P. Pezzi, Elton N. Mathias, Nicolas Maillard, and Philippe Olivier Alexandre Navaux. Improving the dynamic creation of processes in MPI-2. In Bernd Mohr, Jesper Larsson Träff, Joachim Worringen, and Jack Dongarra, editors, *PVM/MPI*, volume 4192 of *Lecture Notes in Computer Science*, pages 247–255. Springer, 2006.

[29] Mohamad Chaarawi and Edgar Gabriel. Evaluating sparse data storage techniques

for MPI groups and communicators. In *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part I*, pages 297–306, Berlin, Heidelberg, 2008. Springer-Verlag.

[30] Pat Conway and Bill Hughes. The AMD Opteron Northbridge Architecture. *IEEE Micro*, 27(2):10–21, March 2007.

[31] CoSMoS. Complex systems modelling and simulation infrastructutre. Available from `http://www.cosmos-research.org/`, accessed on March 4, 2013.

[32] Cray The Supercomputer Company. MPT - Cray's MPI Library. Available from `https://www.olcf.ornl.gov/wp-content/uploads/2013/02/MPI_MPT-HP.pdf`, accessed on March 4, 2013.

[33] Anthony Danalis, Lori L. Pollock, D. Martin Swany, and John Cavazos. MPI-aware compiler optimizations for improving communication-computation overlap. In Michael Gschwind, Alexandru Nicolau, Valentina Salapura, and José E. Moreira, editors, *ICS*, pages 316–325. ACM, 2009.

[34] Vincent Danjean, Raymond Namyst, and Robert D. Russell. Integrating kernel activations in a multithreaded runtime system on top of Linux. In José D. P. Rolim, editor, *IPDPS Workshops*, volume 1800 of *Lecture Notes in Computer Science*, pages 1160–1167. Springer, 2000.

[35] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[36] Erik Demaine. A threads-only MPI implementation for the development of parallel programs. In *Proceedings of the 11th International Symposium on High Performance Computing Systems*, pages 153–163, 1997.

[37] J. C. Díaz Martín, J. A. Rico Gallego, J. M. Álvarez Llorente, and J. F. Perogil Duque. An MPI-1 compliant thread-based implementation. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 327–328, Berlin, Heidelberg, 2009. Springer-Verlag.

[38] Gábor Dózsa, Sameer Kumar, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Joe Ratterman, and Rajeev Thakur. Enabling concurrent multithreaded MPI communication on multicore petascale systems. In *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, EuroMPI'10, pages 11–20, Berlin, Heidelberg, 2010. Springer-Verlag.

[39] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06, pages 29–42, New York, NY, USA, 2006. ACM.

[40] R. Kent Dybvig. *Three Implementation Models for Scheme.* PhD thesis, University of North Carolina, Chapel Hill, April 1987.

[41] Kurt B. Ferreira, Patrick Bridges, and Ron Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 19:1–19:12, Piscataway, NJ, USA, 2008. IEEE Press.

[42] Jeffrey Fischer, Rupak Majumdar, and Todd Millstein. Tasks: Language support for event-driven programming. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '07, pages 134–143, New York, NY, USA, 2007. ACM.

[43] Nikolaos Fountoulakis and Konstantinos Panagiotou. Orientability of random hypergraphs and the power of multiple choices. In *Proceedings of the 37th international colloquium conference on Automata, languages and programming*, ICALP'10, pages 348–359, Berlin, Heidelberg, 2010. Springer-Verlag.

[44] A. Friedley, T. Hoefler, G. Bronevetsky, and A. Lumsdaine. Ownership Passing: Efficient Distributed Memory Programming on Multi-core Systems. February 2013. Accepted at PPoPP'13.

[45] GHC. Data Parallel Haskell. Available from `http://www.haskell.org/haskellwiki/GHC/Data_Parallel_Haskell`, accessed on March 4, 2013.

[46] Brice Goglin and Stéphanie Moreaud. KNEM: a generic and scalable kernel-assisted intra-node MPI communication framework. *Journal of Parallel and Distributed Computing (JPDC)*, 73(2):176–188, February 2013.

[47] David Goodell, William Gropp, Xin Zhao, and Rajeev Thakur. Scalable memory use in MPI: a case study with MPICH2. In *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, EuroMPI'11, pages 140–149, Berlin, Heidelberg, 2011. Springer-Verlag.

[48] Google, Inc. The Go Programming Language. Available from `http://golang.org/`, accessed on March 4, 2013.

[49] Ganesh Gopalakrishnan, Robert M. Kirby, Stephen Siegel, Rajeev Thakur, William Gropp, Ewing Lusk, Bronis R. De Supinski, Martin Schulz, and Greg Bronevetsky. Formal analysis of MPI-based parallel programs. *Communications of the ACM*, 54(12):82–91, December 2011.

[50] WestGrid: Western Canada Research Grid. Compute Canada's regional supercomputing facility. `http://http://www.westgrid.ca`, accessed on March 4, 2013.

[51] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[52] W.D. Gropp, E.L. Lusk, and A. Skjellum. *Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface*. Scientific and Engineering Computation Series. Mit Press, 1999.

[53] William Gropp. Learning from the success of MPI. In *Proceedings of the 8th Intl. Conf. on High Performance Computing*, HiPC '01, pages 81–94, London, UK, 2001. Springer-Verlag.

[54] William Gropp. Half full or half empty? Invited panel presentation for Thomas Sterling's multicore panel at Supercomputing 2006. Available from `http://www.cs.uiuc.edu/homes/wgropp/bib/talks/2006/multicore-panel-final.pdf`, accessed on March 4, 2013., November 2006.

[55] William Gropp. MPI 3 and beyond: Why MPI is successful and what challenges it faces. In Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, editors, *EuroMPI*, volume 7490 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2012.

[56] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambridge, MA, 1994.

[57] William D. Gropp, Laxmikant V. Kale, David A. Padua, Arch Robison, and Marc Snir. A single programming model for clusters and multiprocessor nodes: Dream, nightmare, reality, or vision. Panel discussion in Workshop on Charm++ and its Applications. Available from `http://charm.cs.uiuc.edu/workshops/charmWorkshop2009/program.html`, accessed on March 4, 2013., April 2009.

[58] William D. Gropp and Ewing Lusk. *User's Guide for MPICH, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[59] William D. Gropp and Rajeev Thakur. Issues in developing a thread-safe MPI implementation. In *PVM/MPI*, pages 12–21, 2006.

[60] Andreas Gustafsson. Threads without the pain. *ACM Queue*, 3(9):34–41, 2005.

[61] Bernard M. Hauzeur. A model for naming, addressing and routing. *ACM Trans. Inf. Syst.*, 4(4):293–311, December 1986.

[62] Torsten Hoefler and Marc Snir. Writing parallel libraries with MPI - common practice, issues, and extensions. In Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra, editors, *EuroMPI*, volume 6960 of *Lecture Notes in Computer Science*, pages 345–355. Springer, 2011.

[63] HPCwire. British columbia researchers notch milestone in pursuit of exascale computing. Available from `http://www.hpcwire.com/hpcwire/2013-01-15/british_columbia_researchers_notch_milestone_in_pursuit_of_exascale_computing.html`, accessed on March 4, 2013., January 2013.

[64] Chao Huang, Orion Lawlor, and L. V. Kale. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing LCPC 03*, pages 306–322, 2003.

[65] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007.

[66] C. Iancu, S. Hofmeyr, F. Blagojevic, and Yili Zheng. Oversubscription on multicore processors. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –11, april 2010.

[67] Roberto Ierusalimschy. *Programming in Lua.* lua.org, December 2003. ISBN 85-903798-1-7.

[68] Roberto Ierusalimschy and Luiz Henrique de Figueiredo andWaldemar Celes. *Lua 5.1 Reference Manual.* Lua.org, 2006.

[69] Intel Inc. QuickPath architecture white paper, 2008.

[70] Intel. Intel VTune Amplifier XE. Available from `http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/`, accessed on March 4, 2013.

[71] Intel Corporation. Intel MPI Library 4.1. Available from `http://software.intel.com/en-us/intel-mpi-library`, accessed on March 4, 2013.

[72] International Business Machines Corporation. IBM Platform MPI. Available from `http://www-03.ibm.com/systems/technicalcomputing/platformcomputing/products/mpi/index.html`, accessed on March 4, 2013.

[73] Geoffrey Irving. Low performance in an asynchronous, mixed MPI/pthreads application. Discussion on the MPICH mailing list. Available from `http://lists.mpich.org/pipermail/discuss/2013-January/000259.html`, accessed on March 4, 2013., January 2013.

[74] Van Jacobson and Bob Felderman. Speeding up networking *or* A modest proposal to help speed up and scale up the Linux networking stack. Talk at linux.conf.au. Available from `http://www.lemis.com/grog/Documentation/vj/lca06vj.pdf`, accessed on March 4, 2013., January 2006.

[75] Hyun-Wook Jin, Sayantan Sur, Lei Chai, and Dhabaleswar K. Panda. LiMIC: Support for high-performance MPI intra-node communication on Linux cluster. In *ICPP*, pages 184–191. IEEE Computer Society, 2005.

[76] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: A portable concurrent object oriented system based on C++. Technical report, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1993.

[77] H. Kamal and A. Wagner. Added concurrency to improve MPI performance on multicore. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 229–238, 2012.

[78] Humaira Kamal, Seyed M. Mirtaheri, and Alan Wagner. Scalability of communicators and groups in MPI. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 264–275, New York, NY, USA, 2010. ACM.

[79] Humaira Kamal and Alan Wagner. FG-MPI: Fine-Grain MPI for multicore and clusters. In *11th IEEE Intl. Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC) held in conjunction with IPDPS-24*, pages 1–8, April 2010.

[80] Humaira Kamal and Alan Wagner. An integrated runtime scheduler for MPI. In Jesper Larsson Träff, Siegfried Benkner, and Jack J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, volume 7490 of *Lecture Notes in Computer Science*, pages 173–182. Springer Berlin Heidelberg, 2012.

[81] Humaira Kamal and Alan Wagner. An integrated fine-grain runtime system for MPI. *Computing*, pages 1–17, 2013.

[82] Rainer Keller and Richard L. Graham. Characteristics of the unexpected message queue of MPI applications. In *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, EuroMPI'10, pages 179–188, Berlin, Heidelberg, 2010. Springer-Verlag.

[83] Matthew J. Koop, Terry Jones, and Dhabaleswar K. Panda. MVAPICH-Aptus: Scalable high-performance multi-transport MPI over infiniband. In *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*, pages 1–12. IEEE, 2008.

[84] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *Proc. of the USENIX Annual Technical Conference*, pages 7:1–7:14, Berkeley, CA, USA, 2007. USENIX Association.

[85] Robert Latham, William Gropp, Robert B. Ross, and Rajeev Thakur. Extending the MPI-2 generalized request interface. In Franck Cappello, Thomas Hérault, and Jack Dongarra, editors, *PVM/MPI*, volume 4757 of *Lecture Notes in Computer Science*, pages 223–232. Springer, 2007.

[86] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[87] M. Lelarge. A new approach to the orientation of random hypergraphs. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA ’12, pages 251–264. SIAM, 2012.

[88] Tau Leng, Rizwan Ali, Jenwei Hsieh, Victor Mashayekhi, and Reza Rooholamini. An empirical study of hyper-threading in high performance computing clusters. *Linux HPC Revolution*, 2002.

[89] Davide Libenzi. Portable coroutine library. Available from `http://www.xmailserver.org/libpcl.html`, accessed on March 4, 2013.

[90] Zhiqiang Liu, Junqiang Song, and Shaoliang Peng. MPIActor: A thread-based MPI program accelerator. In *IWQoS*, pages 1–2. IEEE, 2010.

[91] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of*

the 13th international conference on Architectural support for programming languages and operating systems, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.

[92] Ewing Lusk. MPI on a hundred million processors...Why not? Talk at Clusters and Computational Grids for Scientific Computing 2008, Available from `http://www.cs.utk.edu/~dongarra/ccgsc2008/talks/Talk10-Lusk.pdf`, accessed on March 4, 2013., September 2008.

[93] Teng Ma, George Bosilca, Aurelien Bouteiller, and Jack J. Dongarra. Locality and topology aware intra-node communication among multicore CPUs. In *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, EuroMPI'10, pages 265–274, Berlin, Heidelberg, 2010. Springer-Verlag.

[94] Teng Ma, George Bosilca, Aurelien Bouteiller, Brice Goglin, Jeffrey M. Squyres, and Jack J. Dongarra. Kernel assisted collective intra-node MPI communication among multi-core and many-core CPUs. In *Proceedings of the 40th International Conference on Parallel Processing (ICPP-2011)*, Taipei, Taiwan, September 2011.

[95] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 5–16, New York, NY, USA, 2010. ACM.

[96] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[97] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, 1993.

[98] Seyed M. Mirtaheri. Scalability of communicators in MPI. Master's thesis, University of British Columbia, March 2011.

[99] S. Moreaud, B. Goglin, R. Namyst, and D. Goodell. Optimizing MPI communication within large multicore nodes with kernel assistance. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–7, April.

[100] Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, 31:6:1–6:31, February 2009.

[101] Myricom - Extreme Performance Ethernet. MPICH-MX. Available from `https://www.myricom.com/support/downloads/mx/mpich-mx.html`, accessed on March 4, 2013.

[102] Stas Negara, Rajesh K. Karmani, and Gul Agha. Inferring ownership transfer for efficient message passing. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 81–90, New York, NY, USA, 2011. ACM.

[103] Stas Negara, Gengbin Zheng, Kuo-Chuan Pan, Natasha Negara, Ralph E. Johnson, Laxmikant V. Kale, and Paul M. Ricker. Automatic MPI to AMPI Program Transformation using Photran. In *3rd Workshop on Productivity and Performance (PROPER 2010)*, number 10-14, Ischia/Naples/Italy, August 2010.

[104] Nicholas Nethercote. Cachegrind: a cache and branch-prediction profiler. Available from `http://valgrind.org/docs/manual/cg-manual.html`, accessed on March 4, 2013.

[105] Martin Odersky. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[106] University of Kent. Occam-pi: Blending the best of CSP and the pi-calculus. Available from `http://www.cs.kent.ac.uk/projects/ofa/kroc/`, accessed on March 4, 2013.

[107] University of Kent. Source code for bird flocking simulation. Available from `http://projects.cs.kent.ac.uk/projects/kroc/svn/kroc/trunk/tests/ccsp-comparisons`, accessed on March 4, 2013.

[108] OpenMP. The OpenMP Application Program Interface. Available from `http://openmp.org/wp/about-openmp/`, accessed on March 4, 2013.

[109] Avneesh Pant, Hassan Jafri, and Volodymyr Kindratenko. Phoenix: A runtime environment for high performance computing on chip multiprocessors. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:119–126, 2009.

[110] Viji Paul and K. A. Germina. On Edge Coloring of Hypergraphs and ERDöS-Faber-LováSZ conjecture. *Discrete Math., Alg. and Appl.*, 4(1), 2012.

[111] Marc Pérache, Patrick Carribault, and Hervé Jourdren. MPC-MPI: An MPI implementation reducing the overall memory consumption. In Matti Ropo, Jan Westerholm, and Jack Dongarra, editors, *PVM/MPI*, volume 5759 of *Lecture Notes in Computer Science*, pages 94–103. Springer, 2009.

[112] Marc Pérache, Hervé Jourdren, and Raymond Namyst. MPC: A unified parallel runtime for clusters of NUMA machines. In *Proceedings of the 14th international Euro-Par conference on Parallel Processing*, Euro-Par '08, pages 78–88, Berlin, Heidelberg, 2008. Springer-Verlag.

[113] Mikael Pettersson, Konstantinos F. Sagonas, and Erik Johansson. The HiPE/x86

Erlang Compiler: System description and performance evaluation. In *FLOPS*, pages 228–244, 2002.

[114] Photran. An Integrated Development Environment and Refactoring Tool for Fortran. Available from `http://www.eclipse.org/photran`, accessed on March 4, 2013.

[115] Steve Plimpton and Karen Devine. MapReduce-MPI Library. Sandia National Laboratories. Available from `http://www.sandia.gov/~sjplimp/mapreduce.html`, accessed on March 4, 2013.

[116] Robert Preissl, Martin Schulz, Dieter Kranzlmller, Bronis R. de Supinski, and Daniel J. Quinlan. Transforming MPI source code based on communication patterns. *Future Generation Computer Systems*, 26(1):147 – 154, 2010.

[117] Boris V. Protopopov and Anthony Skjellum. A multithreaded message passing interface (MPI) architecture: Performance and program issues. *Journal of Parallel and Distributed Computing*, 61(4):449 – 466, 2001.

[118] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, PDP '09, pages 427–436, Washington, DC, USA, 2009. IEEE Computer Society.

[119] Ashay Rane and Dan Stanzione. Experiences in tuning performance of hybrid MPI/OpenMP applications on quad-core systems. In *Proceedings of 10th LCI Int'l Conference on High-Performance Clustered Computing*, March 2009.

[120] IBM Research. Cognitive computing milestone: IBM simulates 530 billon neurons and 100 trillion synapses. Available from `http://www.zmescience.com/research/`

`technology/cognitive-computing-ibm-simulation-brain-0942343/`, accessed on March 4, 2013.

[121] Juan-Antonio Rico-Gallego and Juan-Carlos Díaz-Martín. Performance evaluation of thread-based MPI in shared memory. In *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, EuroMPI'11, pages 337–338, Berlin, Heidelberg, 2011. Springer-Verlag.

[122] Carl G. Ritson, Adam T. Sampson, and Fred R. M. Barnes. Multicore scheduling for lightweight communicating processes. In *COORDINATION*, pages 163–183, 2009.

[123] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 47–56, New York, NY, USA, 2010. ACM.

[124] Jerome Saltzer. On the naming and binding of network destinations. Network Working Group. Available from `http://tools.ietf.org/html/rfc1498`, accessed on March 4, 2013., 1993.

[125] Jerome H. Saltzer. Naming and binding of objects. In *Operating Systems, An Advanced Course*, pages 99–208, London, UK, 1978. Springer-Verlag.

[126] Adam Sampson. *Process-Oriented Patterns for Concurrent Software Engineering.* PhD thesis, Computing, University of Kent, CT2 7NF, September 2008.

[127] N. Schemenauer, T. Peters, and M. Hetland. PEP 255 Simple Generators. Available from `http://www.python.org/peps/pep-0255.html`, accessed on March 4, 2013., 2001.

[128] Adrian Schpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the Barrelfish manycore

operating system. In *Proceedings of the Workshop on Managed Many-Core Systems (MMCS)*, June, 2008.

[129] Robert W. Sebesta. *Concepts of Programming Languages (7th Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[130] Kai Shen, Hong Tang, and Tao Yang. Adaptive two-level thread management for fast MPI execution on shared memory machines. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 49, New York, NY, USA, 1999. ACM.

[131] Livio Soares and Michael Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[132] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for Java. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 104–128, Berlin, Heidelberg, 2008. Springer-Verlag.

[133] Alexandros Stamatakis and Michael Ott. Exploiting fine-grained parallelism in the phylogenetic likelihood function with MPI, Pthreads, and OpenMP: A performance study. In *Proceedings of the Third IAPR International Conference on Pattern Recognition in Bioinformatics*, PRIB '08, pages 424–435, Berlin, Heidelberg, 2008. Springer-Verlag.

[134] Christopher A. Stone, Melissa E. O'Neill, and The O. C. M. Team. Observationally cooperative multithreading. In Cristina Videira Lopes and Kathleen Fisher, editors, *OOPSLA Companion*, pages 205–206. ACM, 2011.

[135] QNX Software Systems. QNX Neutrino realtime operating system. System architecture. Available from `http://support7.qnx.com/download/download/20970/sys_arch.pdf`, accessed on March 4, 2013., 2010.

[136] Hong Tang, Kai Shen, and Tao Yang. Compile/run-time support for threaded MPI execution on multiprogrammed shared memory machines. *SIGPLAN Not.*, 34(8):107–118, 1999.

[137] Hong Tang and Tao Yang. Optimizing threaded MPI execution on SMP clusters. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 381–392, New York, NY, USA, 2001. ACM.

[138] Marc Tchiboukdjian, Patrick Carribault, and Marc Pérache. Hierarchical local storage: Exploiting flexible user-data sharing between MPI tasks. In *IPDPS*, pages 366–377. IEEE Computer Society, 2012.

[139] R. Thakur, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, and J. L. Traeff. MPI at Exascale. In *Procceedings of SciDAC 2010*, Jun. 2010.

[140] Rajeev Thakur and William Gropp. Test Suite for Evaluating Performance of MPI Implementations That Support MPI_THREAD_MULTIPLE. In *Proc. of the Euro PVM/MPI*, pages 46–55, September 2007.

[141] The Open MPI Development Team. Open MPI: A high performance message passing library. Available from `http://www.open-mpi.org/`, accessed on March 4, 2013.

[142] C. Tismer. Continuations and stackless python. In *Proceedings of the 8th International Python Conference*, 2000.

[143] Edgar Toernig. Coroutine library. Available from `http://www.goron.de/~froese/coro/coro.html`, accessed on March 4, 2013.

[144] TOP500. Top 500 supercomputing sites. Available from `http://www.top500.org/`, accessed on March 4, 2013.

[145] TOP500. Top 500 supercomputing sites - highlights for June 2008. Available from `http://www.top500.org/lists/2008/06/highlights/`, accessed on March 4, 2013.

[146] Jesper Larsson Träff. Compact and efficient implementation of the MPI group operations. In *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, EuroMPI'10, pages 170–178, Berlin, Heidelberg, 2010. Springer-Verlag.

[147] Keith D. Underwood and Ron Brightwell. The impact of MPI queue usage on message latency. In *Proceedings of the 2004 International Conference on Parallel Processing*, ICPP '04, pages 152–160, Washington, DC, USA, 2004. IEEE Computer Society.

[148] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association.

[149] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: scalable threads for internet services. In *SOSP '19*, pages 268–281, New York, NY, USA, 2003. ACM.

[150] Peter H. Welch. Process oriented design for Java: Concurrency for all. In Peter M. A. Sloot, Chih Jeng Kenneth Tan, Jack Dongarra, and Alfons G. Hoekstra, editors, *International Conference on Computational Science (2)*, volume 2330 of *Lecture Notes in Computer Science*, page 687. Springer, 2002.

[151] WestGrid. A signpost on the road to exascale: UBC researchers use westgrid to explore exascale computing. Available from `http://www.westgrid.ca/westgrid_news/`

`2013-01-14/ubc_researchers_use_westgrid_explore_exascale_computing`, accessed on March 4, 2013., January 2013.

[152] R. Clint Whaley. Automatically Tuned Linear Algebra Software (ATLAS). Available from `http://math-atlas.sourceforge.net/`, accessed on March 4, 2013.

[153] Wikipedia. MPICH derivatives. Available from `http://en.wikipedia.org/wiki/MPICH#MPICH_derivatives`, accessed on March 4, 2013.

[154] Frederick C. Wong, Richard P. Martin, Remzi H. Arpaci-Dusseau, and David E. Culler. Architectural requirements and scalability of the NAS parallel benchmarks. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '99, New York, NY, USA, 1999. ACM.

[155] Theodore M. Wong, Robert Preissl, Pallab Datta, Myron Flickner, Raghavendra Singh, Steve K. Esser, Emmett McQuinn, Horst D. Simon Rathinakumar Appuswamy, William P. Risk, and Dharmendra S. Modha. 10^{14}. *IBM Research Report*, November 2012.

[156] Judicael A. Zounmevo and Ahmad Afsahi. An efficient MPI message queue mechanism for large-scale jobs. In *ICPADS*, pages 464–471. IEEE Computer Society, 2012.

Appendix A

FG-MPI Specific Routines

Form follows function.

Louis Sullivan

Following is a list of the `MPIX` routines, introduced by FG-MPI, to provide additional functionality related to co-located MPI processes. The boiler-plate code was described in Section 3.4.

MPIX_Yield

The calling processes performs a voluntary yield to the scheduler.
Prototype:

```
void MPIX_Yield(void)
```

MPIX_Usleep

The calling processes yields to the scheduler which blocks it for at least *utime* microseconds before placing it back on the run queue.
Prototype:

```
int MPIX_Usleep(unsigned long long utime)
```

MPIX_Get_collocated_size

Determines the number of co-located MPI processes in an OS-process, as specified by `nfg` flag with `mpiexec`.

Prototype:

```
int MPIX_Get_collocated_size(int *size)
```

`size` is the number of co-located MPI processes (integer)

MPIX_Get_collocated_startrank

Determines the smallest `MPI_COMM_WORLD` rank from among the co-located processes in an OS-process.

Prototype:

```
int MPIX_Get_collocated_startrank(int *startrank)
```

`startrank` is the smallest rank (integer).

MPIX_Comm_translate_ranks

Translates the ranks of MPI processes in one communicator to another communicator.

Prototype:

```
int MPIX_Comm_translate_ranks(MPI_Comm comm1,
                              int n, int *ranks1,
                              MPI_Comm comm2, int *ranks2)
```

The input parameters are `comm1` (communicator handle), `n` is the number of ranks in `ranks1` and `ranks2` arrays (integer), `comm2` (communicator handle). The output parameter is `ranks2` which is an array of corresponding ranks in `comm2`.

Following are the zero-copy routines discussed in Section 3.3.

```
int MPIX_Zrecv(void ** buf_handle, int count,
               MPI_Datatype datatype, int source,
               int tag, MPI_Comm comm,
               MPI_Status *status)


int MPIX_Zsend(void **buf_handle, int count,
               MPI_Datatype datatype, int dest,
               int tag, MPI_Comm comm)


int MPIX_Izrecv(void ** buf_handle, int count,
                MPI_Datatype datatype, int source,
                int tag, MPI_Comm comm,
                MPI_Request *request)


int MPIX_Izsend(void **buf_handle, int count,
                MPI_Datatype datatype, int dest,
                int tag, MPI_Comm comm,
                MPI_Request *request)
```

FG-MPI does not currently support inter-communicators, dynamic process management functionality and remote memory access operations.

Appendix B

Example of a Program Using FG-MPI

In this section we present the code of a small application to demonstrate writing a program using FG-MPI. This application creates the sieve of Eratosthenes by composing several fine-grain MPI processes to form a pipeline. A pipeline is a commonly used pattern in process-oriented environments for creating process networks within programs and is also used in dataflow applications [126]. This application demonstrates the use of two different mapping functions for composing the pipeline of processes. These mappings can be selected on the command-line to allow experimentation with load-balancing of processes without recompiling the code.

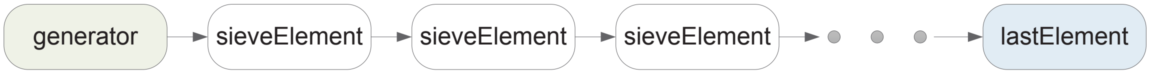


Figure B.1: A pipeline of processes, where numbers generated by the `generator` are streamed through the chain to be processed by each element.

As Figure B.1 shows we have three types of MPI processes:

(a) A `generator` process that generates odd numbers that are passed down the pipeline. The generator keeps the prime number 2 for itself.

(b) `sieveElement` process that keeps the first prime number it sees and filters the remaining numbers by either discarding them or passing them to the next process in the chain.

(c) A `lastElement` process that terminates the prime number generation at the end of the sieve.

The number of prime numbers generated is equal to the length of the pipeline (i.e., the total number of MPI processes in this application). For example, the following command will generate 40 prime numbers.

```
mpiexec -nfg 10 -n 4 ./primeSieve
```

The listing on pages 138 and 139 shows the code for the three functions `generator()`, `sieveElement()` and `lastElement()` bound to these processes. As discussed in Section 3.4 (page 61), `map_lookup` function takes a string as its third parameter and uses it to select a binding function. In this example we provide two different binding functions:

(a) `sequential_mapper` that binds the `generator` to process rank 0, `lastElement` to the highest rank in `MPI_COMM_WORLD` and the remaining processes are all of type `sieveElement`. The pipeline is composed by specifying the previous and next neighbours of a process in the chain (see `who_are_my_neighbors` function on page 139). In this case we have a sequential assignment where process of $rank - 1$ is the previous neighbour and process $rank + 1$ is the next neighbour. Process rank i will generate the $i + 1$th prime number. Due to the packed assignment of MPI process ranks inside OS-processes, this mapping is not the most efficient since the processes appearing later in the pipeline do not become active until a large number of primes are found. The `random_mapper` binding function addresses this problem.

(b) `random_mapper` uses a technique similar to the shuffling of a deck of cards to create a random chain sequence. It uses two user-defined parameters (`seed` and `cuts`) to randomize the next and previous neighbours of a process. This allows a more even distribution among processes and the parameters can be used for experimentation with load-balancing.

We also define two special constants `MAP_INIT_ACTION` and `MAP_FINALIZE_ACTION` that can be used inside the binding functions. `MAP_INIT_ACTION` can be used by the user for any initialization of structures or actions prior to the actual binding of functions to process ranks. An example of this is in the `random_mapper` binding function, where `MAP_INIT_ACTION` is used to allocate a temporary array for storing the random permutation of ranks that is later read by all the newly spawned processes when they start executing and call the `who_are_my_neighbors()` function. `MAP_FINALIZE_ACTION` can be used by the user in the binding function for any action subsequent to the binding operation. Note that at this stage only the binding of functions to MPI process ranks has been completed, but the processes have not executed yet.

The environment variable `FGMAP` is used to select a binding function on the `mpiexec` command line. For example, the random function can be specified as follows.

```
mpiexec -nfg 10 -n 4 -genv FGMAP random ./primeSieve
<seed> <cuts>
```

Whereas, the following specifies the sequential binding function.

```
mpiexec -nfg 10 -n 4 -genv FGMAP seq ./primeSieve
```

It is possible to specify a scheduling policy in a similar manner. For example, the block scheduler discussed in Section 2.2 (page 31) can be specified as follows (the round-robin (`rr`) is the default scheduler if none is specified).

```
mpiexec -nfg 10 -n 4 -genv SCHEDULER block -genv FGMAP
random ./primeSieve <seed> <cuts>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <mpi.h>
#include "fgmpi.h"

#define MAXINT 1000000
#define FALSE 0
#define TRUE !FALSE
#define DATA_TAG 111
#define STOP_TAG 999
/* forward declarations */
int lastElement(int argc, char **argv);
int sieveElement(int argc, char **argv);
int generator(int argc, char **argv);
int who_are_my_neighbors(int rank, int size,
                         int *prevproc_ptr, int *nextproc_ptr);
FG_ProcessPtr_t sequential_mapper(int argc, char** argv, int rank);
FG_ProcessPtr_t random_mapper(int argc, char** argv, int rank);

FG_MapPtr_t map_lookup(int argc, char** argv, char* str) {
    /* Two mapping functions */
    if ( str && !strcmp(str, "random")) {
        return (&random_mapper);
    } else if ( str && !strcmp(str, "seq")) {
        return (&sequential_mapper);
    }
    /* Default mapper if FGMAP environment variable is not specified */
    return (&sequential_mapper);
}

int main( int argc, char *argv[] )
{
    FGmpiexec(&argc, &argv, &map_lookup);
    return (0);
}

int sieveElement(int argc, char **argv) {
    int nextproc, prevproc;
    int rank, size;
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    who_are_my_neighbors(rank, size, &prevproc, &nextproc);

    uint32_t num,myprime=0;
    int notdone = TRUE;
    while ( notdone ) {
        MPI_Recv(&num,1,MPI_INT,prevproc,MPI_ANY_TAG,MPI_COMM_WORLD,&st
atus);
        if ( status.MPI_TAG == DATA_TAG ) {
            if ( myprime == 0 ) {
                myprime=num;
                printf("%u, ",myprime);
            }
            else if ( num % myprime ){/*not divisable by this prime*/
                MPI_Send(&num,1,MPI_INT,nextproc,DATA_TAG,MPI_COMM_WORL
D);
            }
            else { ; }
        } else if ( status.MPI_TAG == STOP_TAG ) {
            notdone = FALSE;
            /* Send the terminate_TAG*/
            num=1;
            MPI_Send(&num,1,MPI_INT,nextproc,STOP_TAG,MPI_COMM_WORLD);
        } else  {
            fprintf(stderr, "ERROR ERROR bad TAG \n");
            MPI_Abort(MPI_COMM_WORLD,rank);
        }
    }
    MPI_Finalize();
    return 0;
}

int generator(int argc, char **argv) {
    int nextproc, prevproc;
    int rank, size;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Request request;
    MPI_Status  status;
    who_are_my_neighbors(rank, size, &prevproc, &nextproc);

    uint32_t myprime=2;
    uint32_t num=myprime+1;
    printf("%u, ",myprime);
    /* Set up a MPI_Irecv to stop the sieve */
    MPI_Irecv(&num,1,MPI_INT,prevproc,STOP_TAG,MPI_COMM_WORLD,&request)
;
    while ( num <= MAXINT ) {
        int result=FALSE;
        MPI_Send(&num,1,MPI_INT,nextproc,DATA_TAG,MPI_COMM_WORLD);
        num+=2;
        MPI_Test(&request,&result,&status);
        if ( result == TRUE) break;
    }
    num=0;
    MPI_Send(&num,1,MPI_INT,nextproc,STOP_TAG,MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}

int lastElement(int argc, char **argv) {
    int nextproc, prevproc;
    int rank, size;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Status status;
    uint32_t num,myprime=0;
    who_are_my_neighbors(rank, size, &prevproc, &nextproc);

    int notdone = TRUE;
    while ( notdone ) {
        MPI_Recv(&num,1,MPI_INT,prevproc,MPI_ANY_TAG,MPI_COMM_WORLD,&st
atus);
        if ( status.MPI_TAG == DATA_TAG ) {
            if ( myprime == 0 ) {
                myprime=num;
                printf("%u \n",myprime);
                /* Send STOP_TAG to generator */
                num=1;
                MPI_Send(&num,1,MPI_INT,nextproc,STOP_TAG,MPI_COMM_WORL
D);
            }
        } else {
            /* Received a STOP_TAG */
            notdone = FALSE;
        }
    }
    MPI_Finalize();
    return 0;
}

FG_ProcessPtr_t sequential_mapper(int argc, char** argv, int rank){
    int worldsize;
    MPI_Comm_size(MPI_COMM_WORLD, &worldsize);
    if ( (rank == MAP_INIT_ACTION) || (rank == MAP_FINALIZE_ACTION) )
        return (NULL);
    if ( 0 == rank ) return (&generator);
    if ( worldsize-1 == rank ) return(&lastElement);
    return (&sieveElement);
}

/* proc is a temporary shared array that holds the random
 * permutation of the processes created by random mapper.
 * This array is only read by the co-located processes once
 * to discover their previous and next neighbors and is
 * de-allocated after that */
int *proc = NULL;

FG_ProcessPtr_t random_mapper(int argc, char** argv, int rank){
    int first=FALSE;
    int last=FALSE;
    int nfg, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPIX_Get_collocated_size(&nfg);

    int cuts= (size/nfg)-1;
    int seed=0;
    if ( rank == MAP_INIT_ACTION ) {
        if ( argc == 2 ) {
            seed = atoi(argv[1]);
        } else if ( argc ==  3 ) {
            seed = atoi(argv[1]);
            cuts = atoi(argv[2]);
        } else {
            printf("USAGE:  primeSieve [seed] [cuts]\n");
            exit(-1);
        }
        /* do the cuts and swaps */
        srand(seed);
        int i;
        proc = malloc(sizeof(int)*size);
        int *procswap = calloc(sizeof(int),size);
        for ( i=0; i<size; i++) proc[i] = i;
        for ( i=0; seed && i<cuts; i++){
            int k = rand() % size;
            /* swap the k to size portion of array with 0 to k-1 */
            if ( k != 0 ) {
                memcpy(procswap,&(proc[k]), sizeof(int)*(size-k));
                memcpy(&(procswap[size-k]),proc, sizeof(int)*k);
                int *tmp=proc; proc = procswap; procswap=tmp;
            }
        }
        free(procswap);
        return (NULL);
    }
    if ( rank == MAP_FINALIZE_ACTION )
        return (NULL);
    if  ( proc[size-1] == rank ) { last=TRUE; }
    if  ( proc[0] == rank ) { first=TRUE; }
    if ( first )
        return (&generator);
    else if ( last )
        return (&lastElement);
    else
        return (&sieveElement);
}

int who_are_my_neighbors(int rank, int size,
                         int *prevproc_ptr, int *nextproc_ptr){
    int prevproc = -1, nextproc = -1;
    char *mapstr = getenv("FGMAP");

    if ( mapstr && !strcmp(mapstr, "random")){
        static int times_called = 0;
        int i;
        times_called++;
        for ( i=1; i<size-1; i++)
            if ( proc[i] == rank)
                { prevproc = proc[i-1]; nextproc = proc[i+1]; }
        if  ( proc[size-1] == rank )
            { prevproc = proc[size-2]; nextproc = proc[0];}
        if  ( proc[0] == rank )
            { prevproc = proc[size-1]; nextproc = proc[1];}
        if (times_called == size){
            free(proc);
        }
    }
    else {
        prevproc = (0==rank) ? size-1 : rank-1;
        nextproc = (size-1==rank) ? 0 : rank+1;
    }
    *prevproc_ptr = prevproc;
    *nextproc_ptr = nextproc;
    return (0);
}
```

Appendix C

Summary of Different MPI Projects

Figure C.1 on page 141 presents a summary of the different thread-based MPI projects discussed in Section 1.3 (page 8).

Project Name	**Year**	**Objectives**	**Target Environment**	**Runtime system**	**Custom software requirements**	**Status**
TOMPI (Threads-only MPI)	1997	TOMPI is designed to run MPI programs on a single computer. Its objective is to enable efficient development of parallel programs on a single workstation by reducing context-switching overhead among MPI processes.	Single workstation	Runs as a single Unix process and supports POSIX and Solaris threads and Cthreads. Uses semaphores and condition variables for synchronization.	Requires thread packages such as Cthreads for user-level thread support.	Source code last updated in March 1998.
MPI-LITE	1997	Maps MPI processes to user-level threads to reduce communication overhead between threads mapped to the same processor.	IBM SP2	Uses MPI to communicate between OS-processes and its own runtime for inter-thread messaging.	Requires special suffixes for MPI routines to distinguish between inter-thread and inter-process communication. Implementation for IBM SP2.	Inactive. Link to source code broken.
TMPI (Threaded MPI)	1999-2001	Implements MPI processes as pre-emptive threads to reduce overheads arising from process context switches, synchronization and copying.	Linux SMP clusters. Earlier versions for multiprogrammed shared memory machines (SGI Machines).	TMPI maps MPI processes to pthreads inside a single address space on a cluster node. Uses daemon threads for communication between cluster nodes.	Readme file reports that gcc header files are non-ANSI compliant.	Source code last updated in May 2002.
AMPI (Adaptive MPI)	2003	Implements a MPI library that uses process virtualization and is layered on top of the Charm++ framework. It supports dynamic load balancing and migration of MPI tasks.	Commodity clusters	AMPI is layered on top of Charm++ runtime.	Charm++ runtime system	Active
MPC (MultiProcessor Communication)	2008	Provides a unified runtime to improve performance of hybrid approaches such as OpenMP + MPI.	Commodity clusters	Uses a custom MPC MxN thread library for intra-node communication and MPI for inter-node communication.	Compilation requires mpfr and gmp libraries. Uses a modified version of GCC for OpenMP. Not safe to mix Posix threads with MPC Posix threads.	Active
Phoenix	2009	The focus of this work is to implement a runtime tuned for chip multiprocessors. It uses processor virtualization and attribute tagging to identify different execution characteristics such as compute, network and I/O to schedule on a group of virtual processors.	Chip Multiprocessors	MPI layered on top of Phoenix runtime. Phoenix defines the basic unit of execution as execution contexts (EC), which are implemented as a pthreads. In order to negate the effects of the OS-scheduler, Phoenix uses condition variables and semaphores so that the number of runnable ECs matches the physical processors.	Unavailable	Unavailable
AzequiaMPI	2009	Implements MPI tasks as preemptive threads to reduce context switch time and avoid intermediate system copies.	Linux clusters and MMU-less machines	AzequiaMPI is layered on top of the Azequia runtime system.	Requires Azequia for AzequiaMPI can run on top of it. Azequia in turn requires MPD daemons on each node to execute applications.	AzequiaMPI is active but maintenance of Azequia has been discontinued since April 2010.
MPI Accelerator	2010	Pre-emptive thread-based MPI to improve intra-node communication.	Linux clusters	Runtime system layered on top of another MPI implementation. The underlying MPI library is used as inter-node communication mechanism.	Requires a custom MPIActor C compiler (MAcc).	Early prototype. Unavailable.
FG-MPI	2010	Extends the execution model of MPICH2 to support large-scale, fine-grain function-level parallelism.	Commodity clusters	Integrated into the MPICH2 middleware.	None.	Active

Figure C.1: A summary of the different MPI projects discussed in Section 1.3 (page 8).