論文 / 著書情報
Article / Book Information

題目(和文)	大規模並列システム上でのスレッディングモデルにおける並列性、データ移動、同期
Title(English)	Parallelism, Data Movement, and Synchronization in Threading Models on Massively Parallel Systems
著者(和文)	AMERABDELHALIM
Author(English)	Abdelhalim Amer
出典(和文)	学位:博士（理学）, 学位授与機関:東京工業大学, 報告番号:甲第9744号, 授与年月日:2015年3月26日, 学位の種別:課程博士, 審査員:松岡　聡,遠藤　敏夫,吉瀬　謙二,増原　英彦,渡辺　治,ﾊﾞﾗｼﾞ ﾊﾟﾊﾞﾝ
Citation(English)	Degree:, Conferring organization: Tokyo Institute of Technology, Report number:甲第9744号, Conferred date:2015/3/26, Degree Type:Course doctor, Examiner:,,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Parallelism, Data Movement, and Synchronization in Threading Models on Massively Parallel Systems

by

Abdelhalim Amer

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Science

in

Mathematical and Computing Sciences

in the

Graduate School of Information Sciences and Engineering

of

Tokyo Institute of Technology

Committee in charge:

Professor Satoshi Matsuoka, Chief Examiner
Associate Professor Toshio Endo
Associate Professor Kenji Kise
Professor Hidehiko Masuhara
Professor Osamu Watanabe
Group Lead Dr. Pavan Balaji

Spring 2015

Parallelism, Data Movement, and Synchronization in Threading Models on Massively Parallel Systems

Abdelhalim Amer

Abstract

Parallelism, Data Movement, and Synchronization in Threading Models on Massively Parallel Systems

by

Abdelhalim Amer

Tokyo Institute of Technology

The trend in high-end and supercomputing system node designs is towards higher core densities and deeper memory hierarchies. Furthermore, several node resources are not scaling at the same rate as the core density such as memory capacity, memory bandwidth, and the number of network endpoints. Consequently, the memory per core available for applications is becoming scarce and data movements are considered the major source of performance bottlenecks and energy consumption. As a result, application developers are increasingly adopting programming models and runtime systems that allow applications to share node resources and reduce data movements. In this regard, threading models offer opportunities to efficiently exploit the memory hierarchy thanks to their resource-sharing ability. Threads of execution are at the heart of many programming models and frameworks to expose data and task-level parallelism. In addition, hybrid MPI+threads models are gaining importance and popularity, where application threads share resources within the same address space and interprocess communication is ensured by the MPI library.

Multithreaded applications running on highly parallel systems, however, may experience several inefficiencies. Withing the same address space, these shortcomings can be attributed to various factors from the underutilisation of resources because of computational units idleness and load-imbalance, to expensive intranode data movements due to NUMA and contention for the memory subsystem, and synchronization overheads that result from protecting global states from corruption by means of locking, atomic operations, and memory barriers.

In this work, we tackle these inefficiencies at the application level as well as at runtime system level. From an application perspective, we address the challenge of reducing idle-

ness and data movement to efficiently exploit thread-level parallelism. First, we perform a cost-effectiveness analysis of the commonly used naive bulk-synchronous and data-driven models and show that both are not scalable on modern multicore systems. Given that optimal task assignment to compute resources is a scheduling problem known to be NP-complete, we propose a combination of tiling computational patterns to improve data locality while relying on dynamic scheduling, including a data-driven approach, to maximize resource utilization. This strategy proved to be a practical solution to the problem and showed substantial improvements over the naive methods. From a runtime system perspective, we address thread synchronization overheads that occur in applications at the level of the MPI communication layer. We first show that contention in the MPI runtime is a serious scalability bottleneck. We follow by analyzing the effectiveness of a popular MPI implementation in a multithreaded environment and found that the common use of Pthread mutexes to ensure thread-safety incurs resource monopolization and impedes communication progress. We propose a solution to the problem by first ensuring fairness in critical section arbitration. We then refine the arbitration so that it adapts to the MPI runtime workload by favoring threads making progress. We show by extensive analysis and evaluation that our solutions improve performance substantially while incurring minimal implementation costs.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor Prof. Satoshi Matsuoka, one of the brightest minds of our time, for his guidance, research and career advices, and his patience during my doctoral studies where I learned lessons that I will never forget.

I thank the Matsuoka Laboratory members for their constant support throughout my stay at Tokyo Tech. as well as during my student life in Japan.

I am also thankful for the people at the "Programming Models and Runtime Systems" (PMRS) group and Argonne National Laboratory, where I had one of the best intellectual, professional, and human experiences during my internship and research visit.

I am particularly thankful to Dr. Pavan Balaji, one of the most brilliant scientists and teachers I ever met, for his teaching, energy, and for giving me chances to grow as a better scientist through his advices and guidance.

I also want to thank the committee members in charge of my thesis: Prof. Satoshi Matsuoka, Prof. Toshio Endo, Prof. Kenji Kise, Prof. Hidehiko Masuhara, Prof. Osamu Watanabe, and Dr. Pavan Balaji, for their precious time and feedback that resulted in constantly improving and polishing this thesis.

This dissertation would never had seen light without the support, help, feedback, and constant interactions with several colleagues: Dr. Naoya Maruyama, Dr. Miquel Pericas, Prof. Kenjiro Taura, Dr. Rio Yokota, Dr. Huiwei Lu, Prof. Yanjie Wei.

I would like to thank many friends and colleagues from Tokyo Tech. and ANL: Leo, Alex, Kevin, Chisato, Kento, Keisuke, Shirahata, Sangmin, Toni, Wesley, Junchao, Xin, Min, and others that I apologise for not mentioning, for their professional support and their friendship.

Finally, and most importantly, I would never thank enough my family for their love, sacrifices, and their constant support for my education and my growth as an engineer, as a scientist, and as a person.

Chapter I

Introduction

1 Motivation

Power has become a major obstacle for computer architects to improve performance of new chip designs. This led to the emergence of processor packages incorporating several computational cores such as multi and manycore architectures. Consequently, this maintains Moore's law of scaling the number of transistors but puts a barrier on single-core performance because of the breakdown in frequency scaling. In particular, massively parallel systems targeted to high-end and supercomputing are equipped with multiple processing cores within node packages that are often arranged in a cluster and interconnected by a fast network fabric such as InfiniBand. Future systems are likely to follow the trend of fitting more cores inside processor packages with deeper memory hierarchies with each new generation.

To run efficiently within a single node, in the sens of time to solution, applications have to expose multiple levels of concurrency such as data and thread-level parallelism [44]. However, time to solution is no longer considered the primary metric for efficiency and new metrics have emerged such as energy efficiency and memory consumption rates. That is, time to solution matters less if the system exceeds its power budget or if the problem size of the workload the system was designed for cannot fit in the available memory. In addition, data movement is one of the largest contributors to application execution time and its energy consumption. As a result, researchers are increasingly looking for programming models and runtime systems that allow exposing parallelism to utilize the computational resources while avoiding expensive data movements and reducing memory requirements.

Interprocess data movement and synchronization is often implemented using the Message Passing Interface (MPI) [6]. MPI is the most widely used library by applications

running on high performance computing (HPC) systems. Running exclusively with MPI on shared-memory can effectively utilize the computational resources, but does not often efficiently exploit shared resources such as the memory subsystem. Consequently, application developers are increasingly adopting hybrid MPI+X programming, where X designates a shared-memory programming model. One derivative of such hybrid model is the combination of threading models with MPI, such as MPI+OpenMP[25], which is becoming a popular way to run on clusters of multicore systems.

2 Problem Statement

An application that relies on a threading model for shared-memory parallelism may experience inefficiencies that are related to one or a combination of several factors among which we address the following: (1) underutilisation of resources because of computational units idleness and load-imbalance; (2) expensive intranode data movements due to NUMA and contention for the memory subsystem (3) synchronization overheads that result from protecting global states from corruption by means of locking, atomic operations, and memory barriers. We give a brief description of these issues below.

To exploit thread-level parallelism, an application has to divide the execution into smaller streams that we refer as *tasks*. Optimal task assignment to computational units aims at minimizing idleness and communication costs that constitute the makespan of an application task graph. This is a scheduling problem known to be NP-complete even in simple scenarios such as with static scheduling, two processors, and one or two time units for task weights [82]. Thus, taking into account additional constraints, such as dynamic scheduling and irregularities in task execution times and communication costs that characterize modern parallel applications makes it even harder to find an optimal schedule. Due to this complexity, programming models and their underlying runtime schedulers rely on heuristics for task assignment (e.g. work-first and random work-stealing).

Bulk-synchronous approaches that expose parallelism by embarrassingly parallel large stages (e.g. OpenMP work-sharing constructs) serialized through global synchronizations are among the simplest and most popular models for parallel execution. In case each stage performs a static partitioning of the tasks, big chunks of work are issued to the processing cores and load-imbalance may occur due to irregularities in the workload and the data access latencies. In case tasks within stages are scheduled dynamically, idleness can be reduced but often not completely eliminated because the work within the same stage may not be enough to saturate the computational cores. In addition, the non-determinism of the dynamic execution may introduce more irregularity in the data access patten and generally hinders data locality. This observation also applies to when breaking the global

synchronizations and expose fine-grained concurrency, such as in a data-driven execution model. Although this latter model exhibits some degree of producer-consumer data reuse, other forms of data locality are often not well exploited. Consequently, reducing data movement in thread-level parallelism often goes against exposing parallelism, thus, leading to difficult trade-offs. An execution model that approaches the optimal balance is still an open question and affects directly the progress towards designing efficient programming models and runtime systems.

The increasing popularity of MPI+X programming is not due to productivity improvements but rather because of hardware constrains. There is still a large body of applications that rely exclusively on MPI for parallel execution. For applications to move to a hybrid implementation, there requires enough cost-effectiveness evidence that justified the move. Unfortunately, such evidence is scarce especially at very large scale. Thus, large scale characterisations of this hybrid model is of utmost importance for application developers and also will serve as feedback to the programming models and system software communities for further refinement and improvement.

One of the most serious drawbacks of threading models is their synchronization costs that are necessary to guarantee a consistent execution when threads access global states. Because of the relaxed memory consistency model of modern hardware, software developers have to rely on locks, atomic operations and memory fences to ensure thread-safety. These mechanisms are often expensive and hinder parallel efficiency. In particular, MPI+threads models require careful interaction between MPI and application threads. Therefore, this interaction may be a source of overhead because of the added thread-safety support. Thread safety overheads can be inevitable but their magnitude can vary depending on many parameters such as the critical sections granularity and the synchronization mechanism used to implement them. In particular, when contention occurs at the critical section level, the common practice is to use scalable locks and to reduce the critical section granularity to bring down the chances of serialization. The way a critical section is arbitrated and how it affects MPI runtimes, however, is not well understood because it received little attention from the community and requires careful investigation.

3 Contributions

To address the previously mentioned challenges, we claim the following contributions:

1. We perform an in-depth cost-effectiveness analysis of two major execution models for thread-level parallelism: *bulk-synchronous*, as illustrated by OpenMP work-sharing constructs, and *data-driven* . We choose these models as they offer very different

trade-offs in terms of parallelism and data-locality. We find that, while the load-balancing issues of a bulk-synchronous approach are obvious, eliminating this issue through fine-grained concurrency without careful data-locality optimizations is not a scalable approach. Although the data-driven approach exhibits producer-consumer temporal data reuse, it loses other forms of temporal reuse and spatial locality. In particular, in the presence of memory intensive kernels, data-movement become the major bottleneck and a fine-grained data-driven execution loses against the simple bulk-synchronous model.

2. The problem of choosing the optimal trade-off between parallelism and data locality is hard and depends on my parameters. However, this problem is not fatal and we demonstrate that auto-tuning tiled and dynamically scheduled implementations can overcome the issue. We argue that tiling or blocking computation patters is an important abstraction that allows optimizing for data locality on deep memory hierarchies. For portable performance, however, auto-tuning can be necessary to obtain optimal task and block sizes. We demonstrate this principle with the OpenMP programming model by tiling the computational patterns of dynamic bulk-synchronous and data-driven implementations. We use the Fast Multipole Method (FMM) as a case study and expose the *task granularity* as well as a *blocking factor* for a communication intensive kernel. Our approach emphasizes on the importance of tiling tasks and kernels in both the temporal and spatial dimensions. The evaluation of our auto-tuned implementations showed close to linear scalability with the new data-driven implementation outperforming all the other approaches.

3. We perform a characterization study of MPI-only and MPI+threads at very large scale (512K cores) by using the communication intensive Breath First Search algorithm on a BG/Q system. We show that the coarse-grained nature of interprocess communication of hybrid MPI+threads approach can alleviate some scalability issues of an MPI-only, but does not fix the root problems. We show practical ways to solve the bottleneck issues including using the recent MPI-3 nonblocking barrier synchronization. We also show that, thread contention in the MPI runtime is a serious issue that needs to be addressed.

4. We address the contention problem in a multithreaded MPI runtime by analyzing the effect of the critical section arbitration on communication performance. We show that relying on Pthread mutexes to ensure thread-safety, an approach adopted by most MPI libraries, incurs resource monopolization and constitutes a major bottleneck. We show that establishing fairness in critical section arbitration improves

the overall progress. In addition, we further improve the design by adapting the arbitration to the MPI workload. This is achieved by establishing a two level priority arbitration policy, where threads likely to yield more useful work are favored. We then perform an extensive evaluation of all methods using micro-benchmarks, important kernels, and genome-assembly application and show up to 8x improvements of our solutions over the baseline Pthread mutex design.

4 Document Organization

The document is organized as follows. We discuss the context of this work by giving some background knowledge in Chapter II. We then enumerate related works in Chapter III and discuss how we complement previous research on the topic or how our approaches to solve the problems differ. We follow by the body of the work in the next three chapters. Chapter IV relates our analysis and solution to the inefficiency of parallel execution models on multicore systems. Chapter V analysis MPI+Threads applications at large scale using the BFS algorithm as a case study. This Chapter also introduces the issue of runtime contention when threads access the MPI runtime. The mutithreaded MPI runtime contention is investigated in more detail in Chapter VI where we propose solutions to the problem and their evaluation with several benchmarks and applications. Finally, we make concluding remarks in Chapter VII along with future directions.

Chapter II

Background

This dissertation focuses on the challenges related to using threads for application parallel execution in the context of modern multi-core based systems. Here we provide the requisite background necessary to grasp the context of this work and its pertinence.

1 Technology Trends

Recent hardware design noticed a fundamental switch from improving performance of single execution streams to providing more processors in a single package or even a single die. Here we discuss the reasons for this shift and the new technological trends.

1.1 Fundamental Hardware Limits

Moore's law of scaling the number of transistors is still holding. This is due to the combined effect of decreasing transistor feature sizes and increasing the chip area. This scalability was correlated with the steady increase in CPU frequency and thus yielding better performance with each new hardware generations without extra software development costs [44].

The trend, however, had radically changed after hitting the *power wall*. That is, the technology allows to fit more transistors than can be powered and building single core processors with higher clock frequencies becomes impractical [12] [13]. As a result, the industry moved away from single-core design in favor of multicore architectures.

1.2 Increasing Core Densities

Before the shift to multicore architectures, the increase in the number of transistors served to improve single-thread performance through instruction-level parallelism (ILP), speculative execution, and deep pipelining. Current designs, however, are using the excess

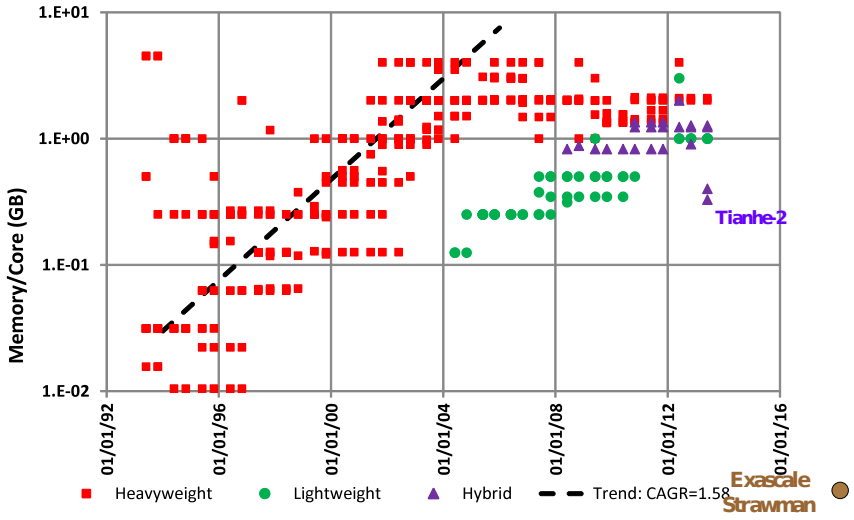


Figure II.1: Evolution of the memory capacity per core for the machines on the Top500 list, by Kogge et al. [49]

.

in transistors to integrate more parallel processing units or *cores*. Consequently, it follows that Moore's law will translate into increasing the number of cores per die. As of this writing, more than 50 cores can be found on Intel Xeon Phi accelerators and larger core counts are to be expected on future hardware generations.

1.3 Scalability of Other Resources

When CPU clock frequency was still scaling, the speed to access memory was a major bottleneck [44]. After the improvement in clock frequencies flattened, the shift to multi-core architectures affected other resources differently. Many resources of the memory subsystem are not scaling at the same rate as the core density (ex. memory capacity and bandwidth). For instance, researchers had identified scaling memory capacity and bandwidth as major challenges toward exascale systems. In fact, as demonstrated by Kogge et al. in Figure II.1, the memory capacity per core is reducing and is affecting directly problem sizes that applications can fit in the main memory [49]. In addition, we can observe that the projections for exascale systems is much worse than current systems.

This hardware trend suggests that applications have to rely on sharing those resources to alleviate their overheads. Multithreading is a common method that allows sharing, thus several programming models rely on multithreading whether by directly using kernel-threads such as POSIX threads, or building on top of them an abstraction layer to improve users productivity and ease performance portability. In the next section we introduce shared-memory programming using threading models.

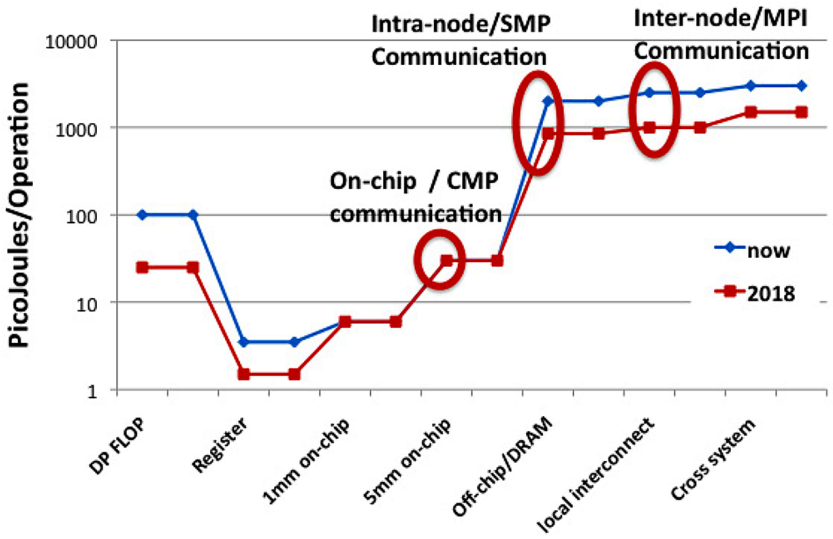


Figure II.2: Energy of moving data with respect to the distance and projections to 2018 systems, by Shalf et al. [71]

.

1.4 Data Movement and Energy

Data movement is a serious obstacle for both performance and energy efficiency. For instance, Shalf et al. showed that one to three orders of magnitude more energy is required to move data to/from off-chip DRAM compared to accessing on-chip memory (see Figure II.2). This implies that applications have to be implemented with careful data-locality considerations and programming models and runtime systems need to offer the possibility to handle data locality transparently to the programmer. We also point out that there should be a priority when optimizing applications for data locality. That is, application developers have to optimize for on-chip memory before moving higher in the memory hierarchy and optimize for socket-level data locality for instance. According to Figure II.2, if an application fails to exploit on-chip data locality, a factor of 10-1000x more energy is paid while penalties below 10x are usually incurred for intersocket (local interconnect) data movements.

In this work, we do not optimize directly for energy efficiency. However, our thread-level parallelism optimizations aim at reducing data movements and in general time to solution which affects directly energy consumption.

2 Parallel Execution Models

We define a *Parallel Execution Model* (PEM) as a model that captures the execution pattern of a parallel algorithm. We differentiate it from *programming models* and *abstract models* in the following sens: a programming model can express different execution models, while an abstract model captures not only an execution model but abstracts architectural parameters such as latency and bandwidth. For instance, a Bulk-Synchronous Parallel model (BSP) [39] is an abstract model that captures the bulk-synchronous execution model while abstracting the number of processors, the latency, and periodicity of an architecture. A programming model, such as OpenMP, can express a bulk-synchronous execution model through the usage of parallel work-sharing constructs and global barriers.

We regard the fork-join PEM as equivalent to a bulk-synchronous in the absence of nesting, since the join operation if equivalent to a global barrier. Thus, from hereon, bulk-sychronous and fork-join are interchangeable. In this chapter, we consider two execution models: bulk-synchronous and fine-grained data-driven. We differentiate the granularity of a data-driven execution with the rate at which tasks are executed. A fine-grained data-driven will typically execute more than a million tasks per second such as in our experiments.

3 Multithreaded Shared-Memory Parallelism

The reducing memory capacity per core is constraining software designers to explore new models, such as avoiding memory-hungry process-only parallel models, and exploiting other levels of the memory hierarchy such as flash memory. In this section we present important concepts related to shared-memory parallelism. We first describe the different forms of parallelism from application and hardware perspectives following the description by Hennessy et al [44].

3.1 Application Perspective

We distinguish two forms of parallelism found in algorithms: *data-level parallelism* and *task-level parallelism*. On the one hand, data-level parallelism occurs when many data elements can be operated on in parallel. On the other hand, task-level parallelism occurs when many tasks of work are created and can largely operate in parallel

3.2 Hardware Perspective

The hardware can execute concurrently the available parallelism from an application. We distinguish the levels of parallelism described in Table II.1.

Table II.1: Hardware levels of parallelism

Level of Parallelism	**Description**
Instruction-Level Parallelism	exploits data-level parallelism through mechanism such as pipelining and speculative execution
Vectorization	exploits data-level parallelism by executing one instruction on multiple data elements at the same time (SIMD)
Thread-Level Parallelism	exploits data-level parallelism and task-level parallelism using multiple threads

In this work, we focus on thread-level parallelism. In this case, data-level parallelism can be expressed though OpenMP work-sharing constructs or by explicitly partitioning the work by spawning and then joining a team of kernel threads. Thus, several tasks that are executed sequentially with each task exploiting data-level parallelism are executing in bulk-synchronous fashion. Task-parallelism, on the other hand, can be expressed though OpenMP tasks and lightweight or use-level threads. With OpenMP 4, it is possible to express arbitrary task graphs using the `depend` clause of the `task construct`. It also possible to implement manually tasks dependencies where tasks are expressed as user-level threads, as will be shown later. In these cases, the resulting parallel execution model is data-driven.

4 Threading Models

In this section, we introduce a taxonomy of threading models and explain which models our work targets and how future works can extend ours to other models. We also describe how multithreaded applications can be characterized to expose their parallel inefficiencies, which is used as a base for our analysis in Chapter IV.

4.1 Classes of Models

We present here the classes of threading models according to how they map to kernel threads.

1:1 Mapping In this model, an independent piece of work is mapped to a kernel thread. This is typically the case when using directly kernel threads to express parallelism such as when using POSIX threads.

M:1 Mapping In this model, multiple pieces of work are mapped to the same kernel thread. This is often targeted to single-processor systems.

M:N Mapping This is the most general model. Multiple pieces of work, called *tasks*, *chunks*, *user-level threads*, or *lightweight-threads* are mapped to a set or a team of N threads. This model is used by many programming models such as OpenMP[25], Cilk [17], Intel Threading Building Blocks (TBB) [67], etc.

In this work, we study the *M:N* model from an application perspective and the *1:1* model from a runtime system perspective when combined with MPI. Because the most widely used MPI implementations view only kernel threads and they are not aware of higher abstraction levels such as the *M:N* thread mapping.

4.2 Characterizing Multithreaded Application Performance

Given the different factors that influence multithreaded parallel executions, researchers have tried to characterize and mitigate these factors. Previous works have identified the factors that contribute to the execution time of task parallel applications [65][75] as summarized in Table II.2

Table II.2: Parallel execution characterization

idle time	occurs when a thread has no work because of load-unbalance or task-dependencies
runtime overhead	is the time spent by the runtime in task creation and scheduling
work time	is the time spend doing actual work.
work time inflation	The work time in a parallel execution often surpasses the work time of a sequential execution due to different memory performance. This difference is referred to as *work time inflation*.

Thus, to achieve efficient parallel execution, one needs to minimize idle time, runtime overhead, and reduce the inflation of the work time. Unfortunately, these properties are often orthogonal, where, for instance, reducing idleness through fine-grained tasks increases scheduling overhead and makes difficult to preserve data locality.

5 Hybrid MPI+Threads Programming

MPI+X programming, where "X" denotes a shared-memory programming model, is a two-level hybrid model that employs MPI for a coarse-grained process-level parallel ex-

ecution, and a fine-grained parallel execution within the same address space (process) using a shared-memory programming model. This hybrid model has recently emerged as a viable model for many-core architectures. A common variant is to use OpenMP or Pthreads for "X," where multiple threads are used for intranode parallelism while internode communication is done with one or more communication threads. Such a model, however, has its own pros and cons compared with those of an MPI-only model. In the following, we give a conceptual comparison between the two models followed by short description on how MPI handles thread-safey.

5.1 MPI-only vs. MPI+Threads Paradigms

Although MPI is the predominant programming system on high-end and HPC platforms, MPI alone is often insufficient to take full advantage of a system resources. An MPI-only model uses one MPI process per system core and is capable of effectively utilizing the available processing units but fails to fully utilize the memory hierarchy. Specifically, each process's memory is private, thus requiring message passing to move data between cores, and often duplication of data between processes to improve local access. In Figure II.3 we present a conceptual comparison between the MPI-only and the hybrid model. We observe that the MPI process model has a finer-grained internode communication model, and enforces explicit interprocess communication even within the same physical node. In addition, data sharing is not possible between address spaces, a feature that reduces further the ability of the model to exploit the memory hierarchy. Moreover, the available memory for the application is reduced compared with that of a hybrid model because of often-duplicated boundary data and the memory requirements of the MPI runtime.

Due to the previously stated shortcomings of the MPI-only model, application developers are increasingly looking at using hybrid MPI+X programming to utilize the computational units while sharing memory. A hybrid model can handle intranode parallelism more effectively and, perhaps more important, hybrid MPI+threads models alleviate some of the interprocess data movement constraints associated with the MPI-only model by allowing a more coarse-grained "node-to-node" data movement (e.g., if threads share all memory on a node), rather than having to send a separate message to every core. Arguably, however, the threads' shared-memory view may incur overheads to maintain consistent execution through locking, synchronization, and memory barriers.

The superiority of either model highly depends on the target machine specification, such as performance and density of the cores, the latencies and bandwidths of the memory and network, and the topology of the components, as well as the characteristics of the application, such as its computation and communication requirements, the amount of

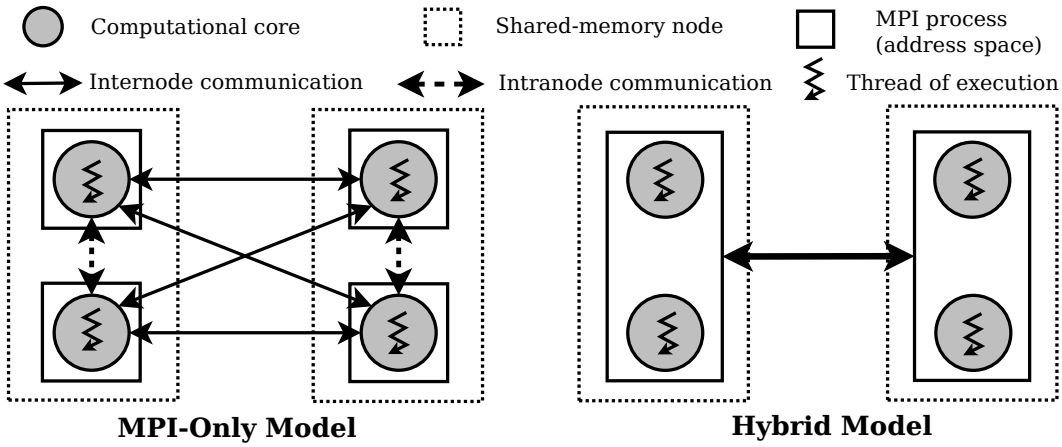


Figure II.3: Conceptual comparison between the MPI-only model and the hybrid MPI+threads model

parallelism, and the application's ability to map to the machine topology. However, the technology trend suggests that hybrid models are likely to be better at handling large-scale machines. The reason is primarily due to the increasing core density in cluster nodes that requires efficient shared-memory programming and, combined with the growth in the number of nodes, makes fine-grained core-to-core MPI programming difficult to scale in terms of performance and memory consumption.

The implications of using either model are still not well understood, however, and require further investigation with various architectures, algorithms, and applications. Thus, in Chapter V, we study the two models using the breadth-first search algorithm on graph structures in order to expose the limits of the models in the hope to guide future programming models and runtime systems design.

5.2 Interoperation between MPI and Threads

Since in a hybrid MPI+threads model the MPI library is a shared resource, thread safety can be necessary in order to maintain a consistent execution. To alleviate applications from unnecessary thread-safety overheads, the MPI standard defines four levels of threading support—`MPI_THREAD_SINGLE`, `MPI_THREAD_FUNNELED`, `MPI_THREAD_SERIALIZED`, and `MPI_THREAD_MULTIPLE`—to be specified when initializing MPI [6]. These levels are listed in increasing order of threading support, where the user can choose the desirable level of thread safety depending on the needs of the target application.

Chapter III

Related Work

In this chapter, we discuss some related work that tried to address similar challenges. We describe the related works in question, how they differ from our methods, and in some cases how the related work and ours complement each other.

1 Task-Level Parallelism on Multicores

A large body of work has been dedicated to task-level parallelism. Parallelism and data locality is well known trade-off that makes writing parallel algorithms harder and efficient task scheduling, an already NP-hard problem, difficult to achieve.

Yan et al. abstracted the memory hierarchy using the Hierarchical Place Trees (HPT) model [89]. In the HPT model, the user specifies the desired view of the memory with a configuration specification, where the memory is abstracted as a tree of *places*. Here, both tasks and the data are mapped to places and data movements between places obey well established mechanisms. The authors pointed out that the optimal configuration depend on the application, the hardware architecture, and the desired trade-off between locality and load-balancing where auto-tuning techniques can be used to search the best configuration parameters.

However, a programming model that uses HPT is not flexible enough to express arbitrary dependencies between tasks. In addition, while some temporal data reuse can be exploited by running tasks operating of common data on the same place, arguably, it is more difficult to control data locality since this likely depend on the order of task creations and their execution. For instance, the data that a task is operating on that had been previously loaded by another task running on the same place may be evicted from the cache memory because of compulsory misses. Thus, we argue that the notion of "places" is not enough to take advantage of the memory hierarchy and a more powerful

abstraction is needed.

Olivier et al. explored the concept of *locality domains* similar to places by extending OpenMP with runtime routines to allow users to implement locality-aware divide-and-conquer algorithms. Previous works focused on loop parallelism or divide-and-conquer task parallel algorithms while ignoring arbitrary-dependencies-type of tasking models. In addition there is a lack of rationalization regarding spacial locality. That is, two tasks running in the same place or locality-domain may exhibit reuse, but does not necessary mean that the order in which tasks are scheduled on the same core will exhibit high spacial locality. More specifically, in order to preserve a high software and hardware prefetching success rate, tasks operating on contiguous data to each other must be scheduled successively and in the right order. This issue is especially important when going towards fine-grained task execution.

Bauer et al. introduced *Legion* [16], a programming model that targets heterogeneous architectures with deep and complex memory hierarchies. Legion and its runtime system are articulated around *logical regions* that allow expressing locality and independence of data, and tasks that operate on those regions. The main difference compared to our work is that we target threading models on homogeneous multicore systems while using production programming models (OpenMP) as well as experiment with user-level threads-based data-driven implementations.

Tasilar et al. introduced the implementation of Data Driven Tasks (DDT) as an extension to the *async-finish* model to allow arbitrary runtime task graphs execution [78]. However, this previous work focused mostly on the syntax and semantic of the model. Although a comparison study of multiple runtime schedulers was performed, no particular attention was payed to data locality.

Towards understanding the performance of multi-core machines, some authors used stencil-computation and sparse matrix-vector multiplication kernels and optimized them for state-of-the-art multi-core architectures[26] [85]. These works use small kernels as benchmarks while going deeper in architectural details in order to get insight into performance trade-offs, although no particular attention was payed to the efficiency of different execution models. In our work (Chapter IV), we use FMM, a sizable algorithm which uses multiple kernels, as a benchmark to get insight into the performance of different execution models.

2 Auto-tuning

Auto-tuning, short for automatic performance tuning, is a well know practice that tries to automate the process of finding the optimal set of parameters for an algorithm when

running on a target machine. The tuning parameters are various and can range from compiler options, such as levels of loop unrolling, to cache blocking factors, and other application specific parameters. It is often associated with tuning numerical libraries such as the mathematical library ATLAS [84] or Fast Fourier Transform libraries [38, 64]. Given the increasing complexity of parallel architectures, auto-tuning has gained more interest toward optimizing parallel algorithms. For instance, an extensive work has been conducted on auto-tuning stencil algorithms on multi and many-core architectures [87, 47]. However, to the best of our knowledge, none has reported work related to auto-tuning more complex algorithms such as the Fast Multipole Method (FMM) or data-driven implementations. Our auto-tuning approach in Chapter IV generates a single binary for each target execution model and application. During the auto-tuning step, each binary is run with different application specific as well as execution model specific parameters. The goal is to choose the parameters of the fastest run for production usage.

3 Hybrid MPI+Threads

3.1 Characterization of MPI+Threads Applications

Considerable work has been conducted on studying the pros and cons of using MPI-only and shared-memory methods and their hybrid derivatives. Early experiments with the hybrid approach revealed that depending on the target application an MPI-only model can be more suitable for performance, while others may benefit from a hybrid approach [19, 46, 52, 24, 88]. Some authors shed light on the practical implications of these models and showed cases where a hybrid model can be superior [68]. We summarise relevant previous studies in Table III.1. Here we only provide information about the maximum scale in core count, the used benchmark or application, and the main conclusion of the work. For more details, the reader can refer to the related work in the first column of the table. The main conclusion we draw from the outlined history is that the number of cases in favor of the hybrid model are increasing over time and are proportional to the scale of the testbed and the abundance of intranode parallelism.

Our characterization efforts in Chapter V aims at exposing the coarse-grained vs. fine-grained communication of the MPI-only and MPI-threads, respectively. In addition, we push further the scalability of the study by running on up to 512K cores. Furthermore, we enrich our study by adding a new variable to the equation and allowing thread concurrent access to MPI in the hybrid model. This exposes some overheads of the hybrid model where the MPI implementation has to guarantee thread-safety. Previous case studies where threads participate in communication are scarce. This ability gained importance

Table III.1: History of MPI-only and Hybrid MPI+X comparison studies

Work	Max core count	Benchmark	Overall conclusion
[46]	81	NAS BT benchmark	MPI-only often better especially on fast networks.
[19]	128	NAS benchmarks	MPI-only often better. Hybrid is better with a slow network and enough intranode parallelism
[24]	512	DL POLY (molecular dynamic)	MPI-only better at small scale, hybrid better at large scale
[68]	8K	NAS BT-MZ and ST-MZ benchmarks	Hybrid better at large scale.
[88]	10K	NAS BT and ST benchmarks	Hybrid better at large scale
Our study	512K	Graph500 benchmark	Hybrid better at large scale

given the benefit of driving the network through multiple endpoints. Among these few studies, Cappello et al. investigated the benefit of having threads help with the internal MPI computation part [19]. Others have also proposed a solution to the idleness of threads when only a single core is driving the communication in a hybrid MPI-OpenMP model [72]. The authors addressed the challenge by exposing and exploiting computation-idle OpenMP threads to improve communication.

In Chapter V, we complement previous characterization studies by provideding insight into multithreaded communication performance issues in a hybrid model. In addition, we stressed on the core-to-core communication model and MPI-only and showed its overhead analytically and experimentally.

3.2 Thread-Safety in MPI

Several researchers have addressed the issue of thread safety challenges in MPI implementations. Gropp et al. [43] presented an exhaustive thread-safety requirement analysis of MPI functions and their implementation issues. Performance implications of thread safety were exemplified with an efficient algorithm for generating context ids. Thakur et al. also proposed a method to obtain insight into performance of multithreaded MPI implementations [80]. The solution consists of a test suite composed of multiple benchmarks that simulate typical application scenarios. This method is useful for comparing different multithreaded MPI implementations or measuring the impact of certain optimizations, such as using a dedicated progress thread. However, it gives only a general performance feedback and does not pinpoint the exact performance bottleneck.

Efficiency and thread safety can be orthogonal objectives that are difficult to achieve at the same time when designing a software library. Goodell et al. showed that concurrent accesses from multiple threads to MPI objects can be a bottleneck when using reference counts [40] and proposed more scalable solutions. Hoefler et al. identified that multithreaded MPI messaging involving `MPI_Prob` is thread unsafe and that a conventional lock-based implementation is not scalable [45]. They proposed an efficient solution that goes beyond the implementation level and requires changing the MPI standard.

3.3 Critical-Section Granularity in MPI

The granularity of a critical section is an important parameter, since it influences the degree of parallelism allowed to concurrent thread executions. That is, the longer a critical section is, the more it incurs serialization and thus hinders parallel performance. In the context of MPI, researchers have proposed different thread-safe MPI implementations with different levels of critical-section granularity and their performance and implementation complexity implications [14, 33]. The conclusion was that moving to fine-grained critical sections is a complex task, but communication could substantially be improved. However, performance results were still suboptimal compared with those from a process-driven communication. These results indicate that contention and thread-safety overheads are still not totally eliminated.

In Chapter VI, we demonstrate how a complementary approach to reducing critical section granularities can improve tremendously communication performance. Our approach exploits the fact that the way critical sections are arbitrated affects communication progress, and thus explores better arbitration policies than how common Pthread mutex implementations are ordering access to critical sections.

3.4 Extensions to the MPI Standard

The trade-off between using threads for efficient intranode computation and relying on multiple processes to drive the network make tuning a hybrid implementation a difficult task. For instance, application developers try to tune the number of number of processes per node and the number of threads per processes in order to improve performance. This challenge, and a number of programmability considerations, pushed the community to consider extending the MPI standard to better support multithreaded communication. To overcome the thread-safety issues, many solutions were proposed to redesign MPI to allow more flexibility and concurrency but are not standard-compliant. For instance, some researcher proposed to promote threads as MPI ranks instead of limiting the rank assignment to processes [28, 77, 70]. Other concepts such as MPI Endpoints emerged as an attractive solution, which ensures contention-free multithreaded communication through independent endpoints [31, 32]. However, MPI Endpoints are not yet part of the MPI standard, and no MPI implementation supports it at the moment of this writing. These exists, however, a library-based implementation of MPI endpoints that can be used on top of existing MPI implementations [73].

Chapter IV

Efficient Parallel Execution on Multi-Cores

We postulate in a broad sens that the parallelism/data-locality trade-off follows the description in Figure IV.1. That is, bulk-synchronous PEM often optimizes for locality, except when dynamic scheduling is performed, while the more an execution is dynamic the more the model aims at eliminating idleness to the detriment of data-locality. We emphasize that, this is just a broad postulate, and contradicting examples can be found when a bulk-synchronous execution can observe very little idleness while a dynamic data-driven execution can exhibit idleness and superior data-locality.

This chapter summarizes and extends the work published in the paper "Fork-join and data-driven execution models on multi-core architectures: Case study of the FMM" [11]. Here, we aim at providing empirical cost-effectiveness analysis of several execution models, which we believe will help application developers and system software experts reason about them and build more scalable software. In the following we use a highly optimized Fast Multipole Method (FMM) as a case study. In addition, we proceed in two large steps in our study; first, we characterize naive implementations of the bulk-synchronous and data-driven models; and second, we propose optimized and more scalable implementations of the same models.

1 About the FMM Case Study

Nbody problems can be encountered in many disciplines such as mathematical physics, machine learning, approximation theory, etc. The problem is how to efficiently evaluate

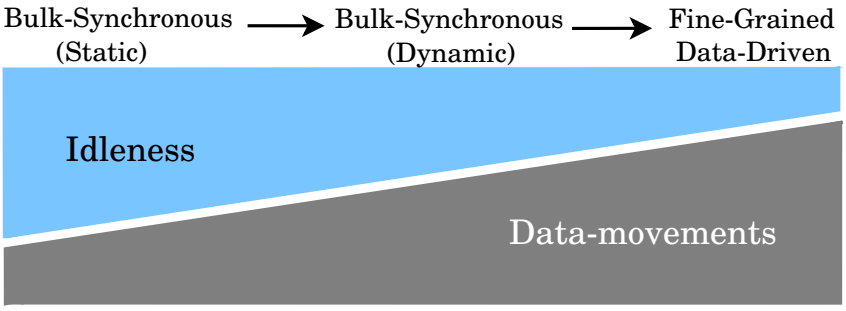


Figure IV.1: Trade-off between idleness (or its dual: parallelism) and data locality in parallel execution models

pairwise interactions between N bodies. It can be formally described as follows:

$$f(x_i) = \sum_{j=1}^{N} K(x_i, y_j)s(y_j), i = [1..N] \tag{IV.1}$$

where $f(x_i)$ is the potential at the target x_i resulting from the sources y_j, s the source density, and K the interaction kernel. A direct computation results in a $O(N^2)$ complexity which makes it very expensive for large problem sizes. First attempts towards a faster method brings the complexity to $O(NLogN)$ like the Barnes-Hut method[15]. The FMM was proposed as an even faster solution that uses a rapidly convergent method achieving a $O(N)$ complexity [42].

Most of FMMs rely on analytic expansions to evaluate pairwise interactions. Analytical expansions are problem dependent, not always available, and difficult to build. In this work we use the Kernel-Independent FMM (KIFMM) developed by Ying et al. which relies only on kernel evaluations, thus enabling FMMs to a wider range of engineering and scientific problems [90, 91]. In KIFMM, the domain is represented by an octree of cells or boxes, where interaction lists are built for each cell following Greengard notation [41], namely: U-list, V-list, W-list, and X-list. The KIFMM implements the force evaluation through the following large stages: U-list, Upward, V-list, X-list, W-list, and Downward. These stages are synchronized by global barriers, are embarrassingly parallel, and traverse the tree cells independently (for the list computations) or level-by-level (Upward and Downward). We distinguish two independent flows of computation: the near field *direct evaluation* represented by the U-list computation, and the *far-field approximation* starting from the Upward stage, computing V-list, W-list, and X-list stages and finishing by the Downward stage. We note that the W-list and X-list computations are negligible for a uniform distribution of bodies.

2 Naive PEM Implementations

In this section, we discuss and characterize the KIFMM that is implemented using the naive bulk-synchronous and data-driven parallel execution models for modern multicore machines.

2.1 Bulk-Synchronous Implementation

The bulk-synchronous KIFMM is the baseline implementation and highly optimized for multicore architectures [21, 22]. All the previously mentioned stages are executed in a bulk-synchronous way. In particular, the upward and downward steps ensure level-to-level dependencies through global barriers. We illustrate how the simplified far-field computation is carried out in Figure IV.2a.

2.2 Data-Driven Implementation

In this section we discuss the implementation of our data-driven solution. That is, the flow of execution goes from the sources to the targets where the far-field and direct evaluation computations are merged into a single flow by starting the Upward and the direct evaluation at the same time.

From target centric to source centric

In KIFMM, the data structures are built from a target centric point of view, thus these data structures need to be rebuilt from a source centric view to enable a data-driven execution. Although in theory if a cell A interacts with cell B, B will interact with A whether symmetrically (U and V lists) or dually (W and X lists), in practice it depends on how a cell's neighbors are determined. Indeed, a cell's interaction lists are only built around a neighborhood, and in KIFMM this neighborhood does not ensure bidirectional interactions between two cells. In order to maintain a correct behavior of the algorithm in a data-driven execution, we compute these lists from a source point of view.

Thread-based data-driven implementation

The dependencies in the data-driven execution can be seen as a producer-consumer synchronization problem as shown in the simplified FMM far-field computation task dependency graph in Figure IV.2b. In our implementation, each task is aware of the tasks that depend on it and may trigger their creations upon termination. Moreover, the task dependencies are satisfied using a combination of *recursive calls* and *atomic counters*. For instance, an atomic counter is used at the `Down` task dependency in Figure IV.2b which

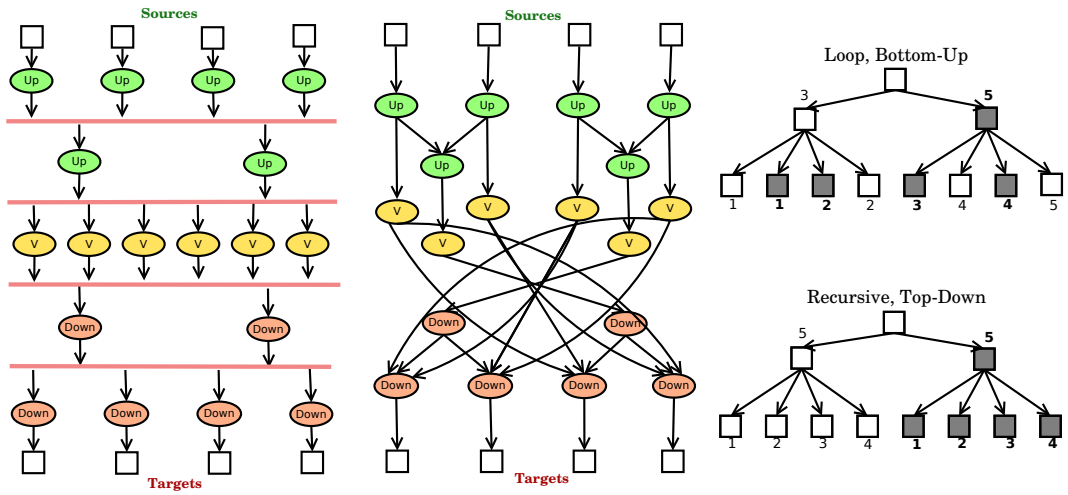
is updated by other `Down` tasks or `V` tasks. In the following we give an example on how a V-list task is executed for a source cell (`src`) after it was called by an `Up` task:

```
void* V (src){
  for(trg in Vlist(src))              //Compute the contribution of src
  {                                   //to all the target cells that
     compute_V(trg,src);              //depend on it
     trg.down_counter++;              //Atomic incrementation of
                                      //the synchronization counter
     /* Test if all dependencies are satisfied */
     if(trg.down_counter = nb_input_depend(trg))
           create_task(Down, trg);    //Create Down computation task
     }
  }
}
```

This pseudo-code shows the V-list and Downward computation tasks (`V` and `Down` resp.) and the target's synchronization counter (`trg.down_counter`) between them. In the Massivethreads library [61], tasks are embedded in lightweight threads scheduled to be executed by *workers*. Task scheduling in Massivethreads follows by default Cilk-like policies. That is, each *worker* is an OS-thread and has a private queue of ready tasks which is managed by a LIFO (Last In First Out) scheduler and a FIFO (First In First Out) work stealing policy between *workers* is adopted. As a result, the `Down` task will be executed first and the `V` task goes at the front of the worker's ready queue. We note that the creation of tasks is incremental and done at the worker level, while the first created tasks, which are situated at the back of the ready queue, may be stolen by other workers. Thus, this work-stealing mechanism ensures good load-balancing. Since each *worker* is scheduling the tasks to be executed independently from the others and uses a private task queue, this method results in a *distributed scheduling* scheme that avoids the drawbacks of a *centralized scheduler*, which may constitute a scalability bottleneck. We note that, being oblivious of which stage in KIFMM, contributions from many cells may be reduced at a target cell. While this is naturally serialized in the original target approach, in our source approach, we serialize these updates by using the *locks* provided by the lightweight thread library. Our experiments showed that contention for the locks occur rarely, which discards these critical sections from being scalability bottlenecks.

2.3 Data-Locality Analysis

The dependency between a `V` and a `Down` task, as described in Section 2.2, corresponds to a *read* after *write* hazard and results in temporal data reuse. Such data reuse can be observed along the paths going from sources to targets. However assessing the spatial

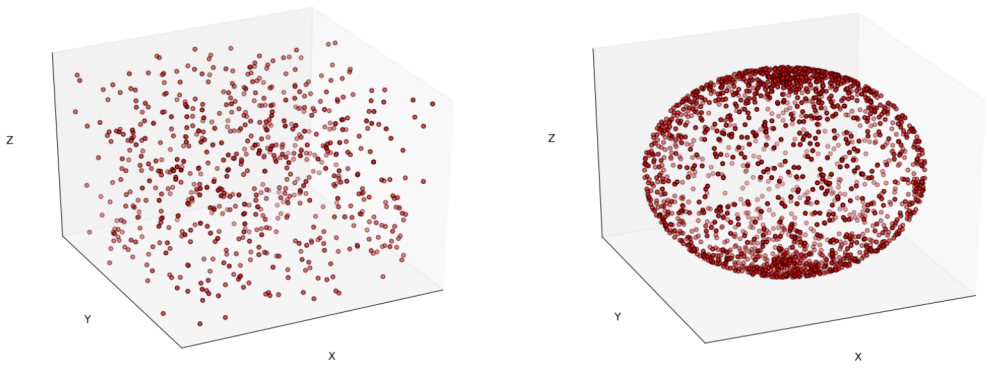


(a) FMM Bulk-Synchronous **(b)** FMM DAG **(c)** Data Locality

Figure IV.2: (a,b) Simplified FMM far-field computation with the bulk-synchronous and data-driven methods. `Up`,`V`, and `Down` stand for the fine-grained tasks that operate at the octree cell level and the upward, v-list, and downward stages, respectively. (c) Simple example of the Upward tasks executed by two *workers*: white for the first *worker* and gray for a second *worker*. The numbering shows a possible task execution order.

data reuse is more subtle since it depends on how the tasks are scheduled. First, one may implement the task graph of Figure IV.2b by traversing the leaf boxes and spawning the Upward and V-list tasks. This method requires *synchronization counters* where each Upward task atomically increments its parent's *counter* upon termination, and triggers the Upward task of its parent if it is the last child. In addition, *workers* will likely access non-contiguous cells in the tree. Indeed, the *serial* code to create all the leaf tasks is also considered as a task which will be preempted and put in the *worker*'s ready queue. A second *worker* will steal that task and create the second leaf task and so on. As a result, the *workers* will access randomly the data as shown by the upper part of Figure IV.2c.

To overcome this issue, we use a top-down recursive algorithm to spawn the tasks as shown in the lower part of Figure IV.2c. In this approach, the work stealing happens in the upper levels of the tree and results in a sub-tree working-set per *worker*, thus, ensuring a better spatial and temporal locality and avoiding additional synchronization variables.



(a) Random uniform **(b)** Elliptical with uniform angles

Figure IV.3: 3-Dimensional visualization of the input particle distributions in a cubic domain

Table IV.1: Target machine specifications. We report the memory bandwidth as the maximum value achieved by the Stream benchmark [54]

	Sandy-Bridge-EP	Nehalem-EX	Magny-Cours
Processor	Xeon E5-2620	Xeon X7550	Opteron 6172
CPU Frequency (Ghz)	2.0	2.0	2.1
# Sockets	2	4	4
# NUMA-Nodes	2	4	8
#Cores/NUMA-Nodes	6	8	6
L3 Cache size (MB)	15	18	6-1
Memory BW (MB/s)	52590.4	68827.3	74720.4
Compiler	GCC 4.4.6	ICC 11.1	GCC 4.4.5

2.4 Performance Evaluation

Experimental Setup

We choose to follow the same input problems as in [21]. That is, we simulate the evaluation of a single step with 4 million bodies following two distributions: a unit cube uniform and an elliptical non-uniform distribution as shown in Figure IV.3. As for the interaction kernel we use the Laplace kernel. For each target machine, we manually tune the maximum number of bodies per cell parameter. We only consider double-precision computation because of its higher pressure on the memory subsystem that we consider more insightful. As for the target architectures, we select representatives of NUMA multi-core architectures, with a 2 socket Intel Sandy-Bridge-EP, a 4 socket Intel Nehalem-EX, and a 4 socket 8 NUMA-nodes AMD Magny-Cours with their detailed specifications given in Table IV.1.

In this section we will conduct a performance analysis of the original KIFMM design

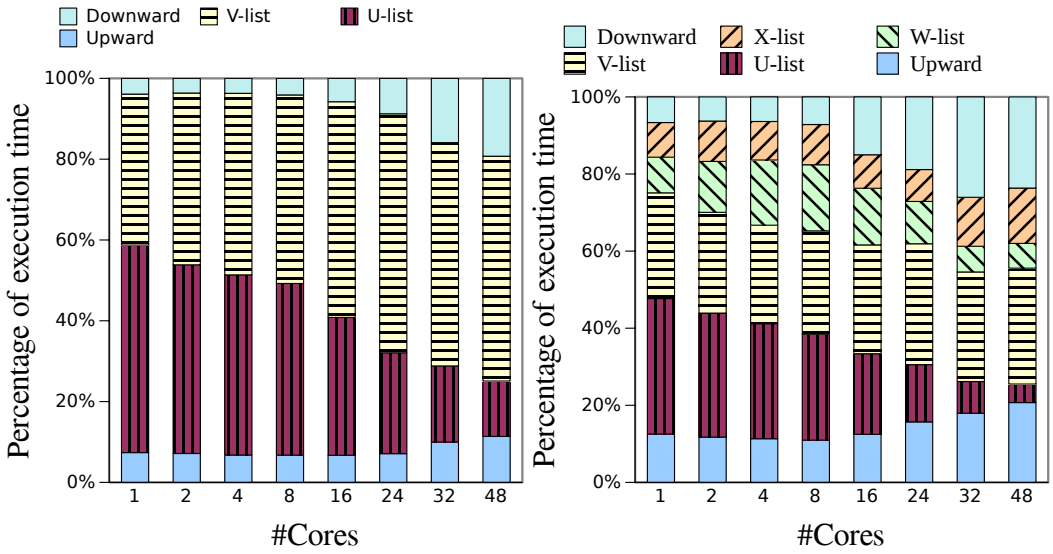
approach as described in [21] and [22]. However we do not consider the intermediate and advanced tuning techniques introduced in [22], as these techniques can also be adapted for a data-driven execution. Our methodology is guided by the high-level knowledge of the application and also relying on hardware performance monitoring tools. For the latter purpose, we use the Vampir tool-set [48, 18] combined with native hardware counters accessible through the PAPI library [60, 3]. In addition, we use the VTune tool to report memory-bandwidth measurements on the Sandy-Bridge-EP machine [4].

KIFMM stages at large scale

It is well known in the FMM literature that the U-list and V-list computations dominate the serial execution time. However, after parallelization, not all of the stages scale at the same rate, and the dominant stages at larger scale may differ. To verify our assumptions, we run strong scaling simulations using the original implementation on the Magny-Cours machine and we report the percentage of execution time taken by each stage as shown in Figure IV.4. These results were reported using high resolution timers without tracing the execution in order to avoid unnecessary overheads. We observe that although the U-list stage takes the longest time when running sequentially, at full concurrency it takes the smallest amount of time while the opposite is observed for the other stages. In the following section, we shed light on the reasons behind this disparity in parallel efficiency of the stages while performing a deep comparative analysis of both KIFMM implementations on the target machines.

Comparative analysis

To decrease the negative effects of NUMA in the data-driven execution, we use the `numactl` command to interleave the memory allocation on the NUMA-nodes where there exists a MassiveThreads *worker*. For both implementations, the OS-threads are scattered across the sockets and bound to the cores to optimize the memory bandwidth utilization. Figure IV.5 shows the strong scaling of both implementations on each machine using both distributions and indicates an overall better scaling of the data-driven execution. We observe that both implementations exhibit very limited speed-ups after using more than half the cores. In particular, both methods have the worst scaling on the Magny-Cours machine likely due to a smaller last level cache and the deepest memory hierarchy among all machines. Also, for the elliptical distribution, there is a 22%, 18%, and 10% speed-up of the data-driven execution over the bulk-synchronous PEM when using half of the cores on the Sandy-Bridge-EP, Nehalem-EX, and Magny-Cours machine, respectively. To bet-



(a) Uniform distribution **(b)** Elliptical distribution

Figure IV.4: Percentage of execution time for each stage on a the Magny-Cours machine for uniform and elliptical distributions.

ter understand this scaling disparity, a deeper analysis of the latter case is performed as it showed the greatest gap between the two methods.

We record statistics for each stage and also for the total force evaluation as shown in Table IV.2. The computation times do not include scheduling and synchronization overheads thus, the differences between the methods are only due to data movements. We used native counters rather than PAPI preset counters which were not enough to gather the information of interest. Native counters are machine dependent, thus we follow the guidelines of the hardware manufacturers to derive our metrics for the Magny-Cours [35] and the Sandy-Bridge-EP [7] machines. However, to the best of our knowledge, similar guidelines are not available for the Nehalem-EX machine, thus we do not report its memory-bandwidth measurements.

We can observe that the data-driven computation time is close to that of the original implementation, indicating that the synchronization overheads eliminated by the data-driven method did not detrimentally affect the data locality. In this experiment we observed improvements of 14%, 17.3%, and 14% resp. for the data-driven which deviate slightly from the above mentioned speed-ups due likely to tracing overheads and operating system noise.

We notice that the data-driven method ensures a better locality for the Upward computations, which hides the slower V-list execution time. In order to verify the locality benefit of using sub-tree based working-sets, a similar approach was implemented for

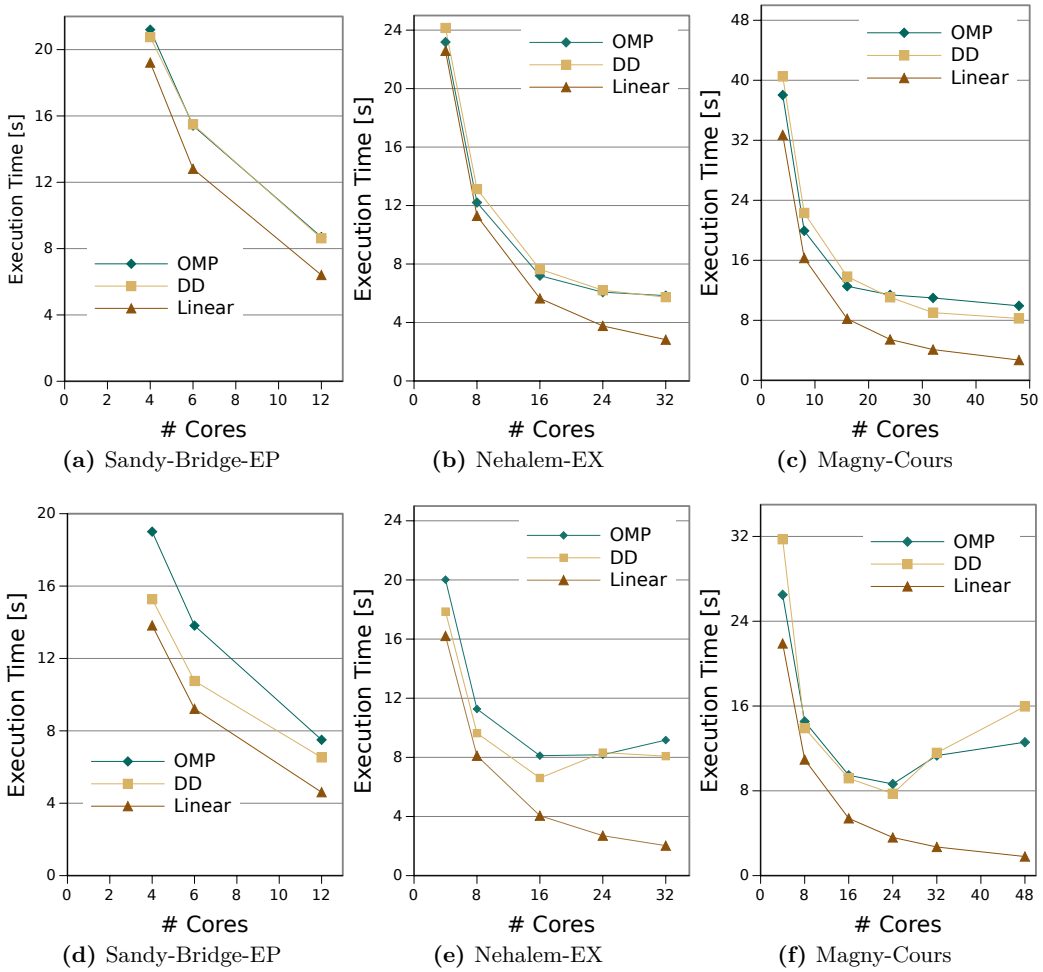


Figure IV.5: Strong scaling of the OpenMP bulk-synchronous (OMP) and the data-driven (DD) implementations for uniform (a,b,c) and elliptical (d,e,f) distributions. To better appreciate the scaling results we added a linear scaling plot.

the Upward stage using the bulk-synchronous model by manually partitioning the tree among the threads. The results, as reported in the last row of Table IV.2, show that the computation runs faster at the cost of a very large synchronization overhead. We also observe that most of the synchronization overheads stem from the X-list and W-list computations. This is not surprising since these computations exhibit the highest variation in terms of work per cell and results in high load-imbalance. An attempt to fix this by means of a `dynamic` or a `guided` OpenMP scheduler resulted in worsening the data locality and increasing the OpenMP scheduling overhead leading to a longer execution time. In this case, the data-driven approach achieves a better trade-off between locality and synchronization overhead. Later, we will perform a more thorough analysis for static

Table IV.2: Computation time (without scheduling and synchronization overheads), OpenMP synchronization overhead, average bandwidth consumption, and relative computation time of the data-driven execution per machine for the elliptical distribution running on half the cores. Note that important information is highlighted. Abbreviations: DD (Data-Driven), SB (Sandy-Bridge-EP), NH (Nehalem-EX), MC (Magny-Cours), and N/A (Not Available). The DD relative time is computed as: $\frac{Time_{OMP}-Time_{DD}}{Time_{OMP}}$, where positive values mean speed-up in favor of the DD-PEM and negative values mean slow downs.

	Comput. Time(s)			Sync. Overhead(%)			Bandwidth(GB/s)			DD Relative Time(%)		
	SB	NH	MC	SB	NH	MC	SB	NH	MC	SB	NH	MC
U-list	27	30.5	38.5	7	14.8	14	0.1	N/A	0.4	-2.96	-2.30	-11.69
Upward	9.56	15.7	43.2	1.2	0.2	7.36	1.2	N/A	0.68	**-4.60**	**17.20**	**44.44**
V-list	13.42	24.7	36.7	1.8	8	1.34	**6**	N/A	**6.8**	**-8.05**	**-10.12**	**-32.15**
W-list	7.3	8.05	10.67	**56.7**	**62**	**61**	0.1	N/A	0.2	0.00	-4.60	-7.78
X-list	7	7.96	15	**29.4**	**25.5**	**24**	0.1	N/A	0.38	-2.86	-3.27	23.00
Downward	5.9	14.9	51.9	2.1	0.2	1.9	1.8	N/A	0.4	0.00	-0.54	1.54
Total OpenMP	70.18	101.8	195.9	**15.6**	**18.8**	**15**	1.3	N/A	1.8	**-3.59**	**-1.19**	**3.22**
Data-Driven	72.7	103	190	0	0	0	**2.7**	N/A	**4.4**			
Upward static	**9.32**	**13.4**	**18.58**	**55**	**24.5**	**45.3**						

and dynamic bulk-synchronous implementations as well as we report results with the data-driven approach.

For a uniform distribution, we observed less synchronization overheads, a worse data locality, and more bandwidth consumption, due to a larger V-list computation, which reduces the effectiveness of the data-driven execution.

Analysis of the memory bandwidth consumption

According to the memory bandwidth measurements of Table IV.2, most of the memory traffic comes from the V-list stage which is known to be memory bound. The memory behavior of this computation can be explained as follows: V-list target-source interactions can be seen as a sparse matrix pattern with high spatial locality and temporal reuse regions at the diagonal of the target particle distributions [22]. These regions are limited (roughly half of the total sources for a uniform distribution) while the rest of the sources are streamed in a non-unit-stride fashion. In addition to the source cells, V-list uses translation vectors, which are also accessed in a non-unit-stride pattern, and further increases the working-set size. We conclude that the V-list bandwidth is consumed by streaming a large working-set following mostly a non unit-stride memory access pattern. Furthermore, reading the sources and translation vectors in a NUMA-aware fashion is not guaranteed.

The data-driven execution of FMM resulted in a homogeneous memory bandwidth consumption rather than concentrated only in the V-list computation. Thus, on a machine with a low memory bandwidth, this execution model will help reduce localized high memory traffic and the performance may improve as long as the overall data locality

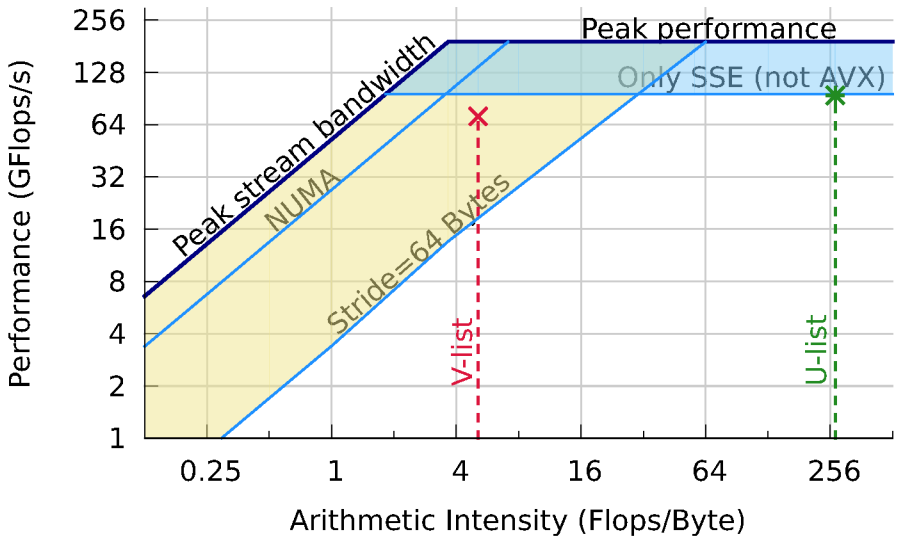


Figure IV.6: Roofline of the Sandy-Bridge-EP machine. The NUMA memory ceiling was obtained using the Stream benchmark with only remote accesses, which was then augmented with 64 bytes strided accesses to plot the stride ceiling. We use SSE vector instructions and do not exploit AVX instructions which halves the computational power. The arithmetic intensity of the computations were derived from machine counters

is not severely hindered. However, for our target machines this was not observed when comparing the V-list bandwidth in Table IV.2 with the Stream bandwidth in Table IV.1 for each machine. The limited scaling of V-list can be explained by the Roofline model [86]. We draw the Roofline plot for the the Sandy-Bridge-EP machine along with the performance achieved by U and V-list computations at full concurrency in Figure IV.6. V-list has a low arithmetic intensity, as opposed to the compute bound U-list, a mixture of unit-stride and non-unit-stride memory accesses, and also local and remote DRAM accesses. Hence, V-list is partially affected by each memory ceiling in the Roofline plot which explains the limited achievable bandwidth and the performance.

3 Tiling Computational Patterns

The previous section exposed the limits of implementing naively the KIFMM using bulk-synchronous and data-driven PEMs. In short, the implementations only optimize in one dimension while the problem of data-locality and idleness is two dimensional. In this section, we show how to optimize in both dimensions. Our approach exploits the principle *tilling computational patterns* to optimize for data locality while relying on *dynamic scheduling* to reduce idleness. In the following we will characterize naive bulk-synchronous implementations and show our optimized OpenMP-based bulk-synchronous

```
void* V-list-phase (){
  #pragma omp parallel for schedule(OMP_SCHED)
  for(trg=0; trg < trgNodeMax; trg++)    //Traverse all target nodes
  {
     for(src in  Vlist(trg))   //Accumulate the contribution of all
       compute_V(trg,src);     //source nodes into the target
  }
}
```

Figure IV.7: Example of how the *V-list* computation is implemented with OpenMP for loop. `OMP_SCHED` is a macro that controls the scheduling policy and takes the values `static` or `dynamic`

and data-driven solutions.

3.1 Static vs. Dynamic Scheduling in the Bulk-Synchronous PEM

The baseline bulk-synchronous KIFMM implemented the phases with different scheduling policies. *Upward* and *downward* used a guided scheduling. *U-list* and *V-list* were using a static scheduling where the target tree nodes were partitioned among the threads with the goal of having the same workload per thread. Finally, *X-list* and *W-list* implement static scheduling. This variety in scheduling methods renders difficult the task of analysing a bulk-synchronous implementation. In order to expose the trade-off between parallelism and data locality, we used two opposite scheduling methods to implement all phases: (1) a static approach that divides the node list equally among the threads without taking into account the workload variation and (2) a dynamic approach that distributes dynamically the workload. Figure IV.7 shows an example implementation of *V-list* stage. We also note that we generate different binaries depending on the value of `OMP_SCHED` passed to the compiler.

We run the rest of the experiments of this section on a dual-socket Sandy-Bridge machine equipped with 8 cores per socket with Simultaneous Multi-Threading (SMT) is disabled. The 8 cores share a 20 MB of L3 cache memory. We run our new implementations with 2^{23} particles and 128 particles per box and show the trace of their respective executions in Figures IV.10a and IV.10b. Here we clearly notice the idleness of threads in all the phases when using static scheduling. Although the dynamic approach fixes the idleness, we notice that to execution time of the computational parts stretched, especially with the *V-list* phase, indicating data locality issues. This suggest a different method that offers a better balance between idleness and data-locality is necessary for scalability.

3.2 Tasking Through Temporal and Spatial Blocking

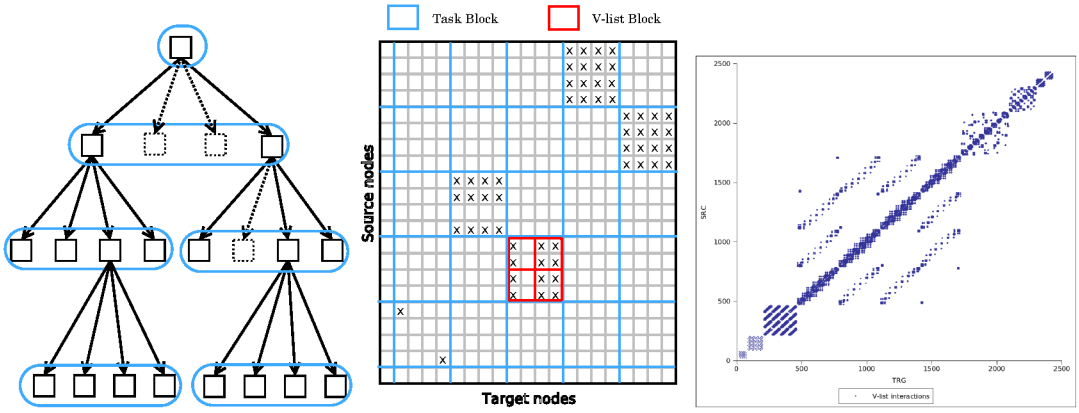
The naive dynamic approach uses fine-grained computations (computation = loop iteration corresponding to work on a single octree box) as schedule able units. Instead, we argue that, with a large enough problem size, it is possible to expose enough parallelism with bigger chunks. We emphasize here that, however, an implementation has to take into account both temporal and spacial locality in scheduling the computations to optimize for cache performance. For an application that does not have memory intensive kernels, exposing the task granularity might be enough, since it will mostly impact on idleness. Otherwise, if a subset of the kernels experience bottlenecks in the memory subsystem, it might be necessary for the user to expose those kernel key parameters to the auto-tuner.

We give a real example of the V-list computation pattern when using an elliptical distribution in Figure IV.8c. This pattern is highly dependent on the input parameters: type of distribution, number of particles, and number of particles per box. Thus, one needs to tune the execution for each set of parameters. Our partitioning scheme suggests to perform a one-dimensional tiling of the source data and another one-dimensional tiling of the target data. The resulting partitioning is a two-dimensional tiling that clusters computational patterns that operate on contiguous data and exhibit high temporal and spatial locality. We illustrate how the data is portioned in KIFMM in Figures IV.8a and IV.8b. Hereon, we use only one tiling factor parameter for the tasks and for all stages. Thus, because of the parent-children dependencies in the octree, the data is not partitioned uniformly and rather done in a way to respect those dependencies.

3.3 Tiled Bulk-Synchronous Approach

In order to implement the previous idea of larger chunks with high data-locality potential, we implement a bulk-synchronous approach that uses *tiles* that form one dimensional blocks with the target data and another one dimensional block for the source data in case of the list computations (all phases except *Upward* and *Downward*). We illustrate this approach with the *V-list* computation phase in Figure IV.9.

This approach, however, requires some tuning since the optimal tile size is unknown. We show the results of tuning this parameter with a uniform and an elliptical distributions in Figure IV.11. We used the optimal tuning parameter and traced the execution of this approach in Figure IV.10c. We notice that the execution time of the computational part is at least as fast as with the static scheduling approach while most of the idleness was eliminated. Some idleness subsists in the *Upward* and *Downward* phases because the upper levels of the tree inherently lack computational work and exhibit many synchronizations. We present in the next section a tiled data-driven implementation that solves



(a) Partitioning of the tree data structure

(b) Applying the partitioning scheme to the V-list computation

(c) Real-world V-list computation pattern

Figure IV.8: Example of a V-list computation pattern (a) and the data partitioning method (b,c)

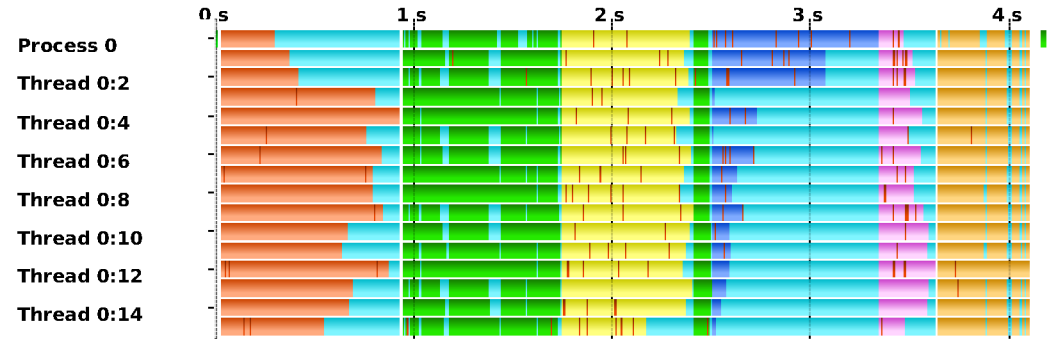
this problem. We finally show strong scaling results in Figure IV.12a and IV.12b and notice the superiority of the tiled-dynamic approach with up to 38% improvements.

```
void* V-list-phase (){
  #pragma omp parallel for schedule(dynamic)
  for(i=0; i < trgNodeMax; i+=BS) {  //Traverse all target blocks
   for(j=0; j < srcNodeMax; j+=BS) { //Traverse all source blocks
     for(trg=i; trg < BS; trg++) {   //Traverse the targets in the block
      for(src=j; src < BS; src++) {  //Traverse the sources in the block
        if(src in  Vlist(trg))       //Accumulate the contribution of all
            compute_V(trg,src);      //source nodes into the target
      }
    }
  }
}
```

Figure IV.9: Example of a bulk-synchronous tiled *V-list* computation with a dynamic scheduling. `BS` denotes the *block size*

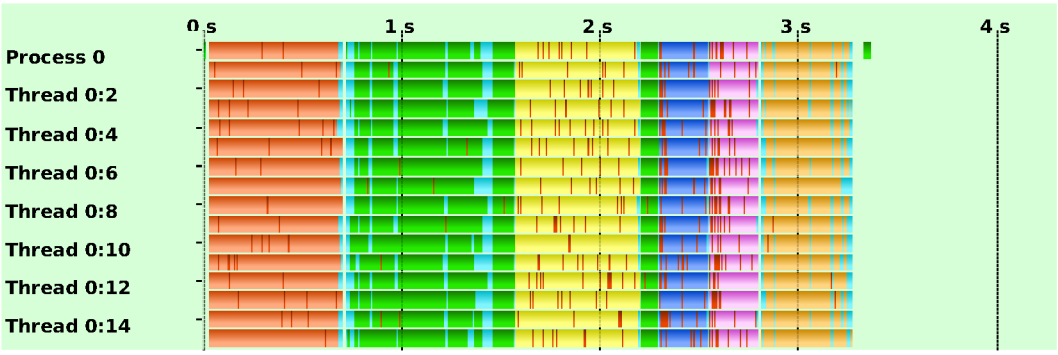
3.4 Auto-Tuning Data-Driven Implementations

In this section we discuss our method of tuning data-driven implementations to achieve a superior balance between parallelism and data locality than what the previous approaches offered. We first describe our methodology to expose key parameters to the auto-tuning step, and then demonstrate auto-tuning results and performance comparisons using KIFMM.



(a) Static scheduling

(b) Dynamic scheduling



(c) Tiling with dynamic scheduling

Figure IV.10: Execution trace of the different bulk-synchronous implementations with an elliptical distribution. The light blue color (`OMP_SYNC`) is the time spent in OpenMP synchronization barriers. The stages are ordered as follows: *U-list*, *Upward*, *V-list*, *X-list*, *W-list*, and *Downward*

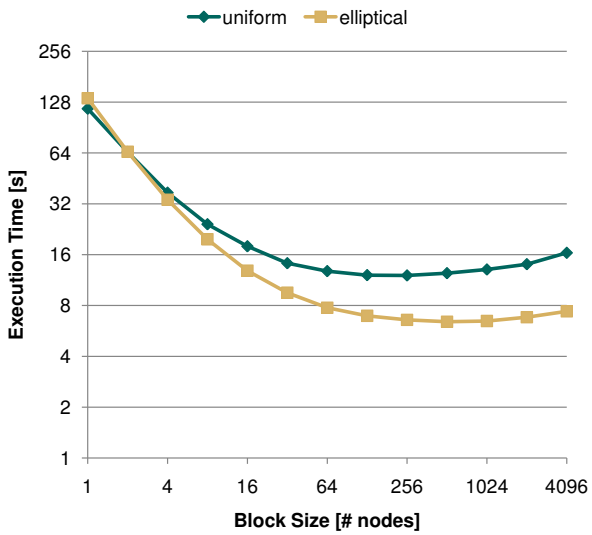
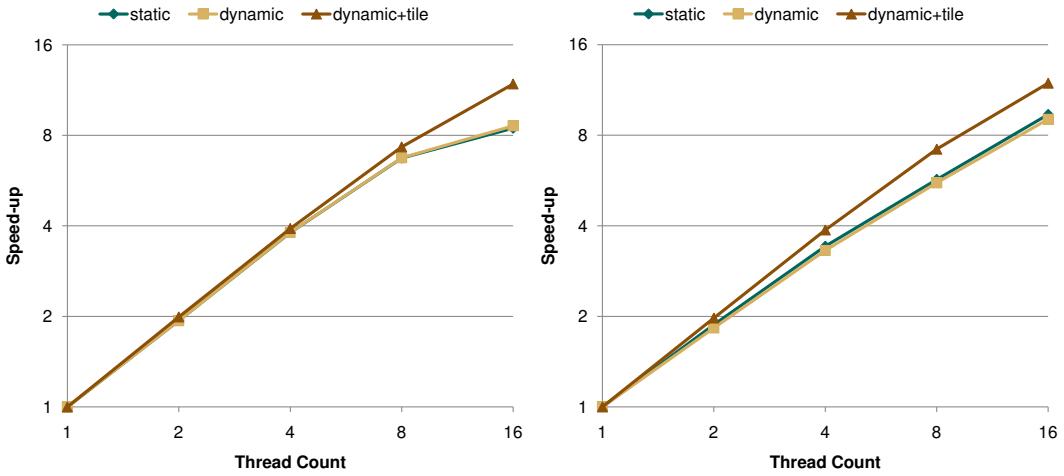


Figure IV.11: Tile size tuning for the tiled dynamic bulk-synchronous implementation



(a) Strong scaling with a uniform distribution **(b)** Strong scaling with an elliptical distribution

Figure IV.12: Performance comparison of the tiled bulk-synchronous implementation with the naive static and dynamic implementations

Tiled Data-Driven implementation

In addition to tiling chunks of computation as we did with the tiled bulk-synchronous method, we expose another blocking factor for *V-list* since it is the main bottleneck of the computations. The justification is that, communication intensive kernels such as V-list often perform worse in a data-driven execution because of sharing the cache with other computations that incurs more compulsory cache misses. This is for instance observed in the previous characterization section. Although not observed in our experiments, a different setting (application and hardware) would exhibit high producer-consumer data reuse and thus would alleviate the compulsory misses issue. We illustrate how the data is portioned in KIFMM in Figure IV.8b with a particular attention to the V-list blocks. The unit of the block granularity is expressed in terms of number of tree nodes. Depending on the type of computation carried out, the real size of the data accessed by the task will depend on the stride of the array accessed by that task. Our implementation relies on the OpenMP 4 tasks with the `depend` construct that allows explicit data-dependency specification in the task clause. These directives are similar to what can be found in the research prototype of the OMPSs programming model [36]. More specifically, we decorate the key kernels with `IN/OUT` dependencies on array regions. The listing in Figure IV.13 shows how we implemented the V-list computation. Those tasks are integrated into a global task graph to allow a full data-driven execution.

```
#define DATA_OUT eff_val[beg_eval:trg_stride]
#define DATA_IN  eff_den[beg_eden:src_stride]
void* V-list-phase (){
  for(i=0; i < trgNodeMax; i+=BS) {  //Traverse all target blocks
   int trg_stride =  eff_trg_size*BS;
   int beg_eval = trg_stride*i;
   for(j=0; j < srcNodeMax; j+=BS) { //Traverse all source blocks
     int src_stride =  eff_src_size*BS;
     int beg_eden = src_stride*j;
     #pragma omp parallel task depend(out: DATA_OUT) depend(in: DATA_IN)
     for(trg=i; trg < BS; trg++) {   //Traverse the targets in the block
      for(src=j; src < BS; src++) {  //Traverse the sources in the block
        if(src in  Vlist(trg))       //Accumulate the contribution of all
            compute_V(trg,src);      //source nodes into the target
      }
     }
   }
  }
}
```

Figure IV.13: Example of a tiled *V-list* computation in the data-driven implementation using OpenMP tasks. `eff_val` and `eff_den` are the input and output vectors, respectively.

Auto-tuning Approach

One of the limits of auto-tuning methods is the rapid growth of the design space with the number of parameters and the ranges of discrete values they can take. To reduce this complexity, the auto-tuner can operate in several steps starting from tuning single-threaded performance then moving to tuning the parallel data-driven implementation. In the case of KIFMM, single-threaded performance was manually tuned in previous works [21, 22]. The only single-threaded tuning operation we perform is regarding the number of particles per box since it can vary highly between input problems and hardware specifications. The most important step here it to explore the design space of the task granularity and the V-list block size. We show the results of tuning two input problems on the Sandy-Bridge machine in Figures IV.14a and IV.14b respectively. We observe that performance is highly variable depending on the task granularity and the V-list block size. In addition, we notice that the optimal parameters are different for the input distributions.

Performance Results

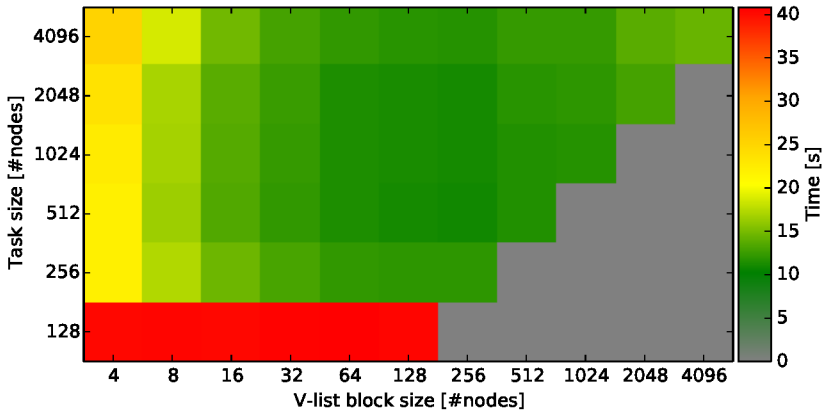
After the previous tuning step, we perform a strong scaling experiment and show the results in Figure IV.15. We notice that we achieve an average of 15% improvement over the tiled bulk-synchronous approach with 16 threads across the input problems.

PEM Comparative Analysis

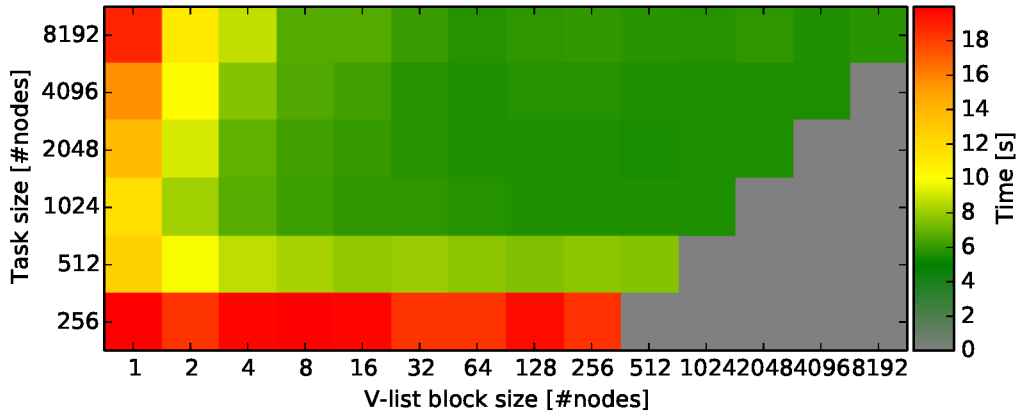
The performance results showed that the data-driven implementation outperformed the bulk-synchronous one, and we speculate that it is likely because of the reduction in thread idleness. Here, we provide some empirical evidence.

The bulk-synchronous execution traces obtained with the Vampir tool-chain offers the reader an intuitive way to attribute performance losses to one of the factors mentioned in Chapter II: idleness shown as `OMP_SYNC`, work time inflation can be obtained by comparing the parallel execution time of the computational kernels and their sequential execution, but scheduling overhead is less intuitive but often negligible for bulk-synchronous type of execution. However, as of this writing and to the best of our knowledge, no tool can profile or trace OpenMP codes that use the task construct with the `depend` clause. Thus, we rely on manual profiling to shed light on the performance results previously obtained.

We first show that both tiled bulk-synchronous and data-driven methods have comparable cache performance by measuring work time inflation (WTI). This is straight forward, since it only requires measuring application kernel execution times and compare them with their corresponding sequential runs. Figure IV.16 shows the WTI of the



(a) Uniform Distribution

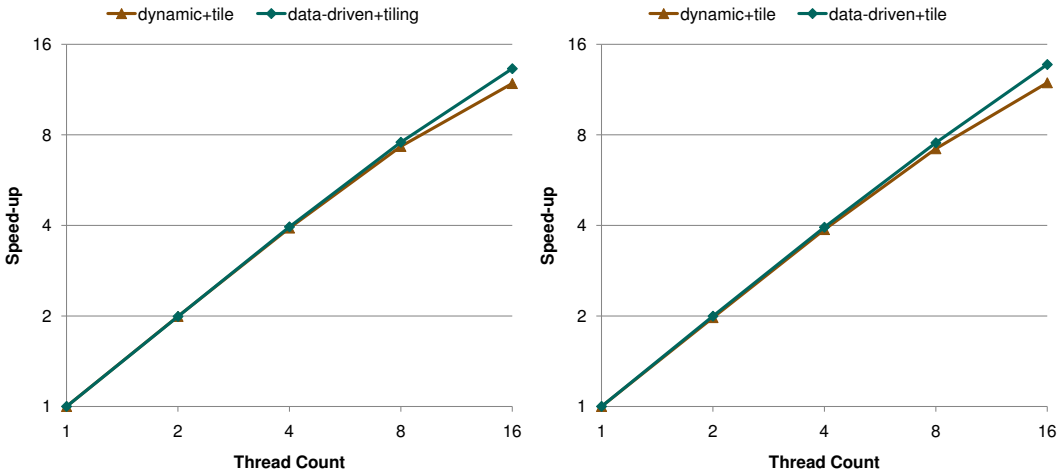


(b) Elliptical distribution

Figure IV.14: 2D tuning space exploration results on the Sandy-Bridge machine with the uniform and elliptical distributions

low level kernels when using all methods. The *Direct* kernel is compute intensive and called by all the higher level kernels except *V-list*. This latter calls intensively the memory intensive *Pointwise*. We notice that all methods have little inflation except for the bulk-synchronous method with dynamic scheduling and the *Pointwise* kernel. This was already observed in our previous traces. Most importantly, the tiled bulk-synchronous and data-driven methods are performing similarly. Scheduling overheads should be higher for the data-driven implementation especially given the data dependency tracking costs. Consequently, the performance improvement must have resulted from reducing idleness.

In order to verify this assumption, we manually instrumented the implementations to record the number of tasks running in parallel per interval of time. This metric would indicate idle threads if the number of running tasks is less than the number of threads. We use for this purpose POSIX timers to implement a sampling based approach. With sample intervals of 5 milliseconds, we show the results in Figure IV.17c. We confirm that



(a) Speed-up over single-threaded performance with a uniform particle distribution

(b) Speed-up over single-threaded performance with an elliptical particle distribution

Figure IV.15: Strong scaling performance results after the auto-tuning step

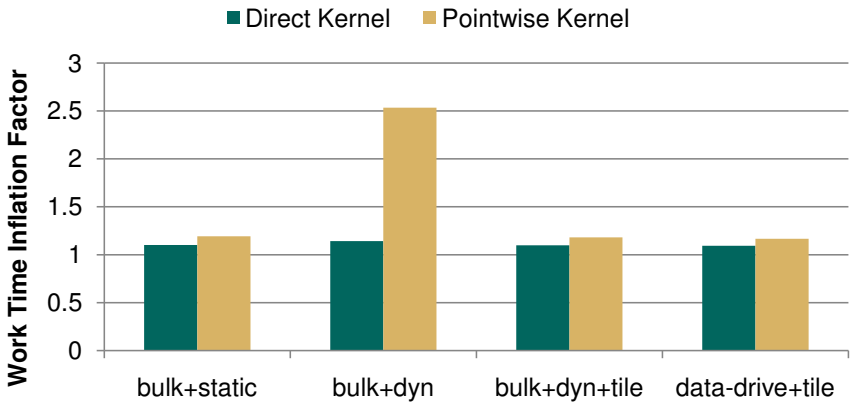
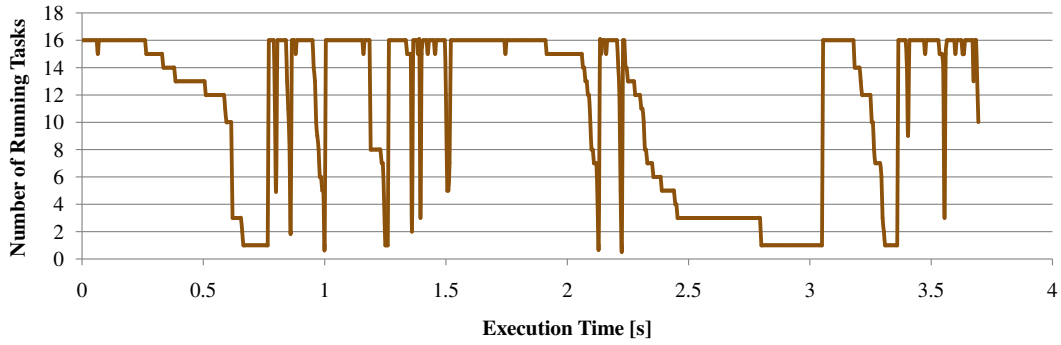


Figure IV.16: Work time inflation comparison between the different implementations

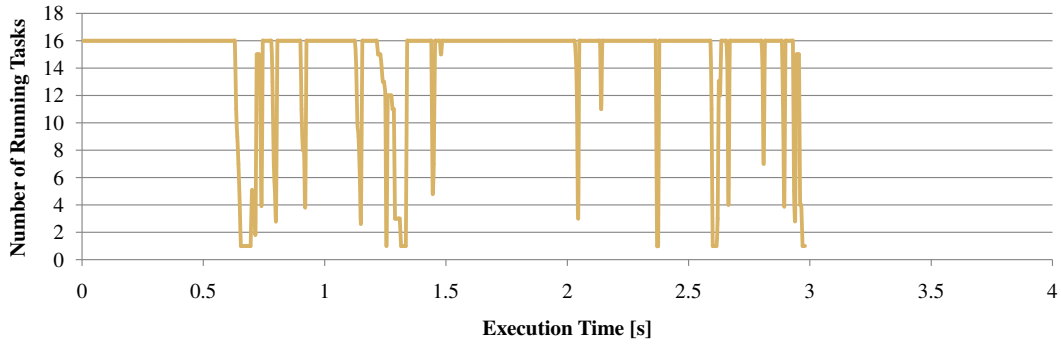
the data-driven approach exhibits the least idle threads among all approaches and that it is the major advantage of the bulk-synchronous approach even after tuning the block sizes. Note that we do not count the sequential code and the task creation operations performed by one of the threads (most likely the master thread) as tasks. As a result, we observe that the tiled-data-driven implementation has one or two less parallel tasks, which map to those operations.

4 Parallel FMM Related Work

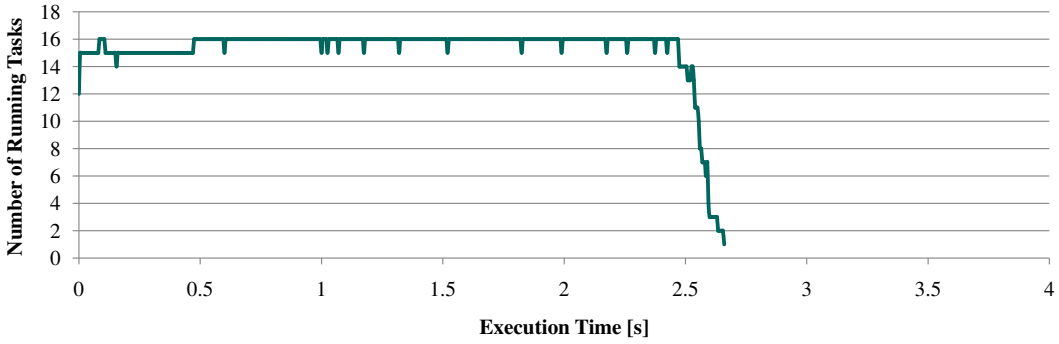
Data-driven execution of FMM is not novel. Yokota et al. [51] proposed a data-driven execution of FMM in order to overcome load-balancing issues. Agullo et al. proposed



(a) Bulk-Synchronous with Static Scheduling



(b) Tiled Bulk-Synchronous with Dynamic Scheduling



(c) Tiled Data-Driven

Figure IV.17: Profiling idleness with an elliptical distribution and 2^{22} bodies

to pipeline the FMM computations on heterogeneous architectures over a runtime [9]. Pericàs et al. implemented a data-driven execution of ExaFMM, a fast open-source FMM [66]. Although these works used a data-driven approach to implement the FMM, their objectives were to load-balance the work among the computational units. Our work also achieves this goal, proposes a novel distributed scheduling scheme, and further presents an in-depth comparison with a bulk-synchronous execution model. Using also the MassiveThreads library, Taura et al. described parallel recursions as an alternative

to parallel loops for implementing ExaFMM [79]. Our methodology can be applied to this work in order to evaluate the implications of using recursions to implement task parallelism.

5 Conclusion

In this chapter, we characterized bulk-synchronous and a data-driven parallel execution models on multicore systems using the KIFMM as a case study. Our initial characterization of naive approaches that optimize only in one of the data-locality/parallelism dimensions proved that such methods are not scalable. Although a fine-grained data-driven implementation showed some degree of temporal data reuse, it loses other forms of data locality and performs similar or worse than a bulk-synchronous. Since both approaches are optimizing in one dimension, we proposed a method that explores the sweet spot in both parallelism and data locality dimensions. We show that by exposing key parameters of dynamic implementations, performance could substantially improved.

In the next section we address another important aspect of using threading models on modern parallel systems: that is, the hybrid MPI+threads model. We study this model from an application perspective, by showing the pros an cons of the model compared to a process-only, and also emphasize on the consequences of runtime contention.

Chapter V

Characterizing MPI+ Threads Applications at Scale

This chapter serves at exposing the pros and cons of a hybrid MPI+threads model by comparing it to the well established MPI-only process model to implement parallel algorithms on clusters of multicores. This serves at identifying the scalability issues that MPI+threads can alleviate, but also shows the limits of this model, e.g. it cannot fix fundamental algorithmic scalability issues that require addressing the root causes. We also demonstrate some bottlenecks that arise because of the data sharing nature of using threads which will serve as an introduction to the more detailed runtime analysis of Chapter VI.

We use along this chapter the breath-first search algorithm as a case study because of its importance and its communication intensive nature.

1 The Case with Parallel BFS

Here we describe the breadth-first search algorithm and its baseline MPI-only distributed implementation that will be used to derive the hybrid MPI+threads implementation.

1.1 The Breadth-First Search Algorithm

Given a graph $G(V, E)$ composed of a set of vertices V and a set of edges E and a root vertex $r \in V$, the BFS algorithm explores the edges of G, in order to traverse all the vertices reachable from r, and produces a breadth-first tree rooted at r. Most parallel graph traversal algorithms are performed level by level, where the vertices of a level are situated at the same distance from the root. Because synchronization is required between levels this approach is referred to as *level synchronized*. We choose the baseline

BFS algorithm from the Graph500 benchmark [5, 1] because it is widely accepted as the reference benchmark to evaluate graph processing capabilities of parallel systems and is well studied [81, 74].

1.2 Baseline MPI-Only Implementation

The baseline implementation is described in Algorithm 1 along with the details of the functions provided in Algorithms 2 and 3.[1]

Algorithm 1: Pseudo algorithm executed by each process for the Graph500 BFS reference simple implementation

```
1  ENQUEUE(CQ, root);
2  while True do
3      IRECV(AnySource);
4      for u ∈ CQ do
5          CHECKINCOMMSGS;
6          for v ∈ Neighbors(u) do
7              O ← Owner(v);
8              if (R = O) then UPDATE(u,v) ;
9              else
10                 if (Pending Send to O) then
11                     WAITPENDSEND(O)
12                 Buffer(v, u, O)              ▷ Put (v,u) in O's buffer;
13                 if (O's buffer full or last message) then
14                     Isend(O)                 ▷ Send buffer content to O
15     SYNCHRONIZE;
16     count = |NQ|;
17     ALLREDUCE(count);
18     if (count = 0) then
19         break                                ▷ NQ of all processes is empty;
20     SWAP(CQ, NQ);
```

Algorithm 1 assumes that the graph is already partitioned between the processes. `CQ` and `NQ` are queues that store the vertices at the current level and the ones that will be visited at the next level, respectively. When a vertex is visited, it is marked in the `visited` vector, and its parent is recorded in the `pred` vector. Each process traverses the graph level by level starting from the root vertex at level 0. First, the root of the graph is enqueued in `CQ` (line 1). The graph traversal is done by the main loop at line 2 with each loop iteration corresponding to one level. Given a vertex in `CQ`, its neighbors are

[1] We use a naming convention for the communication routines similar to the MPI standard. The function names, however, are truncated in order to keep the pseudo algorithm simple.

Algorithm 2: Routine details of Algorithm 1

```
1  UPDATE:
2      if (visited[v] = 0) then
3          visited[v] ← 1;
4          pred[v] ← u;
5          ENQUEUE(NQ, v);
6  CHECKINCOMMSGS:
7      CHECKREQUESTS;
8  WAITPENDSEND:
9      while (Pending Send to p) do
10         CHECKREQUESTS;
11 SYNCHRONIZE:
12     for (p ∈ P) do
13         ISEND(EmptyMessage, p)                    ▷ Signal I am done
14     repeat CHECKREQUESTS until All processes done;
```

Algorithm 3: Communication Progress Routine

```
1  CHECKREQUESTS:
2      if TESTRECV() then
3          for ((v, u) ∈ RecvBuff) do UPDATE(u,v) ;
4          IRECV(AnySource)
5      for p ∈ P do
6          if TESTSEND(p) then                   ▷ Test Send requests ;
7          FREEBUFF(p) ;
```

processed in the loop at line 6. The newly visited vertices will be enqueued in `NQ` (line 8 in `Update`). At the end of each level, `CQ` and `NQ` are swapped so that `CQ` contains the new vertices to be processed while `NQ` reuses the `CQ` memory storage to contain the next-level vertices. The algorithm stops by breaking out of the main loop after `NQ` of all processes are empty.

Computation is carried out by the `Update` routine, and the bulk of interprocess communication is ensured by nonblocking point-to-point communication. The algorithm simulates an event-driven execution by polling for incoming communication (`CheckIncomMsgs`) and for pending send operations (`WaitPendSend`). After all the vertices of `CQ` are processed, the processes synchronize globally by exchanging empty messages (`Synchronize`). All these functionalities use the `CheckRequests` communication progress routine that polls for both incoming and outgoing communication.

It is worth noting that this algorithm puts a lot of stress on the core-to-core commu-

nication model of the MPI-only model because each process has to communicate with all other processes. This is valid for both point-to-point vertex exchanges and the collective `Allreduce` operation (line 17). In the rest of the document, we refer to the problem size by the graph *scale*, where $scale = \log_2 |V|$. In addition, we use Kronecker graphs as input data sets during our experiments [50].

1.3 Hybrid BFS Design and Implementation

Our solution builds on top of the baseline algorithm described in the previous section. The main difference is that both computation and communication are driven by OpenMP threads. A thread is considered here as an independent unit with an execution flow similar to the process execution flow of the baseline method. Algorithm 4 shows where the team of threads is spawned (line 3) and later joined. That is, threads do all the work in parallel except the `Allreduce` operation. We describe below how computation and communication are implemented.

Algorithm 4: Threading extension to Algorithm 1

```
1  ENQUEUE(CQ, root);
2  while True do
3      foreach thread parallel do
4          IRECV(AnySource);
5          for u ∈ CQ do
6              ▷ Do local computation, send operations, and process incoming
                messages
7          SYNCHRONIZE;
8      count = |NQ|;
9      ALLREDUCE(count);
10     if (count = 0) then
11         break                    ▷ NQ of all processes is empty;
12     SWAP(CQ, NQ);
```

Computation

Various optimizations were explored during the past decade to improve the scalability of the BFS algorithm on shared-memory architectures. Since the goal of our work focuses mostly on the communication part, we implement only a subset of existing optimizations that have the highest benefit.

First, we inherit the `visited` vector bitmap representation to reduce the memory footprint of the graph and reduce the amount of data movement [8]. Second, we update

the shared `visited` and `pred` vectors and the `CQ` and `NQ` queues in a lock-free and atomic-free manner. More precisely, the `CQ` is read-only so it does not cause any issue. Writing to `NQ`, however, needs to be protected. Instead of using locks or a lock-free queue that are not scalable, we use private queues per thread and then merge them at the end of each level. This strategy also improves data locality because vertices visited by the same thread will be gathered contiguously in `NQ`, and it increases the opportunities for data reuse when reading the vertices from `CQ`.

Our method shares many similarities to the shared-memory BFS implementation of Chhugani et al. [23]. One major difference is that we do not maintain an array for the depth of the vertices that was exploited by their algorithm to confirm whether a vertex is visited. Instead, we initialize the parent of each vertex with a negative value and test against it to achieve the same atomic-free algorithm as shown in Algorithm 5. Although the algorithm exhibits possible data races, it guarantees the generation of a correct BFS tree on architectures that ensure atomic loads/stores (see [23] for a detailed explanation).

Figure V.1a shows a single-node performance comparison between the baseline MPI-only design and our multithreaded implementation on a Blue Gene/Q node. While both methods scale with the number of cores, the multithreaded method achieves close to 2x better performance. This gives an empirical estimation of the drawbacks of using message-passing for shared-memory parallelism. Although the shared-memory performance of both methods can be improved, we do not optimize the computation part any further since our primary goal is a large-scale study. In fact, we show later that the primary bottlenecks are related to the communication model rather than the computation performance.

Algorithm 5: Pseudo algorithm for the multithreaded `Update` method

```
1 UPDATE:
2     if visited[v] = 0 then
3         visited[v] ← 1;
4         if pred[v] = −1 then
5             pred[v] ← u;
6             ENQUEUE(NQ_i, v);
```

Communication

First, we note that threads in our BFS implementation concurrently perform computation and communication in order to maximize throughput and minimize idleness. Thus, we require the `MPI_THREAD_MULTIPLE` threading support from the MPI library.

To avoid thread contention at the application level, we manage the communication following the same methodology as with the computation part. Since accessing the communication buffers is critical, we use thread-private buffers mapped to remote processes. This approach ensures asynchronous progress and thread independence where the only synchronization point is the implicit barrier at the end of the parallel region. In particular, synchronization inside the parallel region is avoided through the use of OpenMP `nowait` clauses. The `Synchronize` step in Algorithm 2 is also performed by all threads. Each thread sends to all processes a termination message and expects the same type of message from each remote process. In the following, we model the communication analytically in order to compare the communication costs of running with only processes or with one process and multiple threads per node.

Communication Characterization Assuming P processes, $|V|$ vertices in the graph, and e the edge factor, we estimate $\mathcal{C}$, the total number of vertices communicated, as

$$\mathcal{C}(P, V) = 4e|V|\frac{P-1}{P} \tag{V.1}$$

The derivative

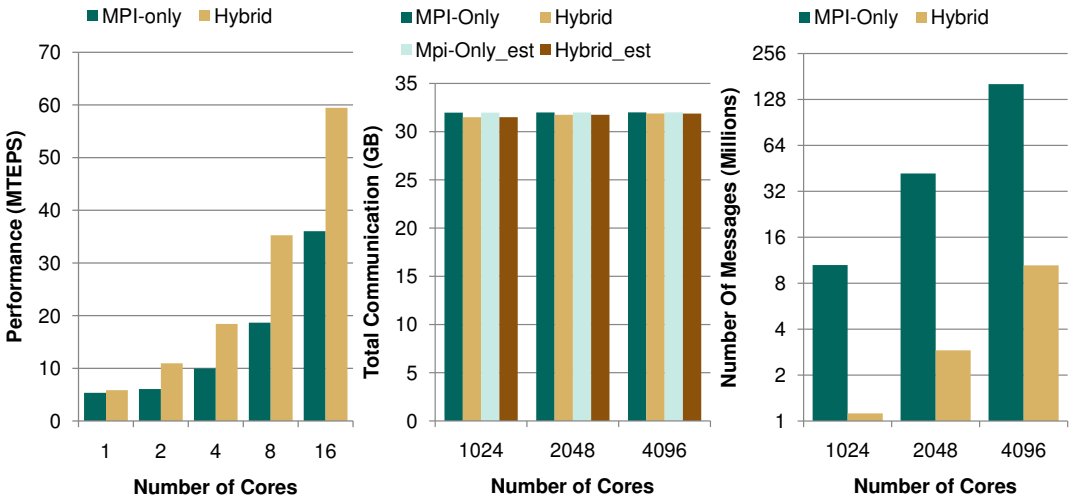
$$\frac{\delta\mathcal{C}}{\delta P} = 4e|V|\frac{1}{P^2} \tag{V.2}$$

is always positive and proves that $\mathcal{C}$ grows proportionally with the number of processes. If we assume that P in an MPI-only model is larger than that of a hybrid model by a factor α, then the growth in communication can be estimated by

$$\frac{\mathcal{C}(\alpha P, V)}{\mathcal{C}(P, V)} = \frac{\alpha P - 1}{\alpha(P-1)} \tag{V.3}$$

We note that for systems where the number of nodes is significantly larger than the core density ($P \gg \alpha$), the increase in communication of the MPI-only over a hybrid method is negligible. For a small-diameter cluster with high core-density nodes, however, the reduced internode communication of the hybrid model can be significant. For instance, with 128 processes and 16 threads per process, we estimate and confirm experimentally the total amount of communication and show the message count in Figures V.1b and V.1c, respectively. Although the gain by the hybrid method in terms of reduced communication is not large in this case, the MPI-only method incurs a larger communication overhead by sending more messages. This message count increase stems from finer-grained graph partitioning between the processes and the global synchronizations.

Scalability of the Global Synchronization In our design, a thread sends an empty message to each process, to signal that it is done sending and expects to receive an empty



(a) Single node performance **(b)** Total communication **(c)** Total message count

Figure V.1: (a) Performance comparison between the MPI-only model and the hybrid mode on a single BG/Q node. (b) Total amount of communication aggregated across all processes. (c) Corresponding total message count with a problem size 26, one process/thread per core for the MPI-only/hybrid method, respectively. `MPI-Only_est` and `Hybrid_est` are the estimation of communication for the MPI-only and the hybrid methods, respectively.

message from each process. Although we could join the threads beforehand and perform the global synchronization by a single thread, we choose to involve all the threads in this step to process incoming messages in parallel. Despite the fact that all threads participate in the synchronization, this method is more scalable than in the MPI-only case. Let us assume M cores per node, N nodes, one process per core for the MPI-only method, and one process per node and one thread per core for the hybrid method. Then, the number of empty messages scales as $O(M^2N^2)$ for the MPI-only method and as $O(MN^2)$ for the hybrid method. As a result, the hybrid method reduces by M-fold the overhead of the global synchronization.

2 Evaluation and Analysis

In this section we present our initial evaluation of both approaches followed by a series of analyses of the bottlenecks and their corresponding optimizations. The evaluation was conducted on a Blue Gene/Q system (Table V.1) while interprocess communication was ensured by using MPICH 3.1.1. Because of the larger memory footprint and the suboptimal performance when using more that one process per core, we use only one hardware thread per core. For fairness reasons, the MPI-only method dictates the problem sizes and thread count per node for the hybrid method. The maximum achievable performance

Table V.1: Target platform specification

Architecture	Blue Gene/Q
Processor	PowerPC A2
Clock frequency	1.6 GHz
Cores per node	16
HW threads/Core	4
Number of nodes	49152
Interconnect	Proprietary
Topology	5D Torus
Compiler	GCC 4.4.7
Network driver	BG/Q V1R2M1

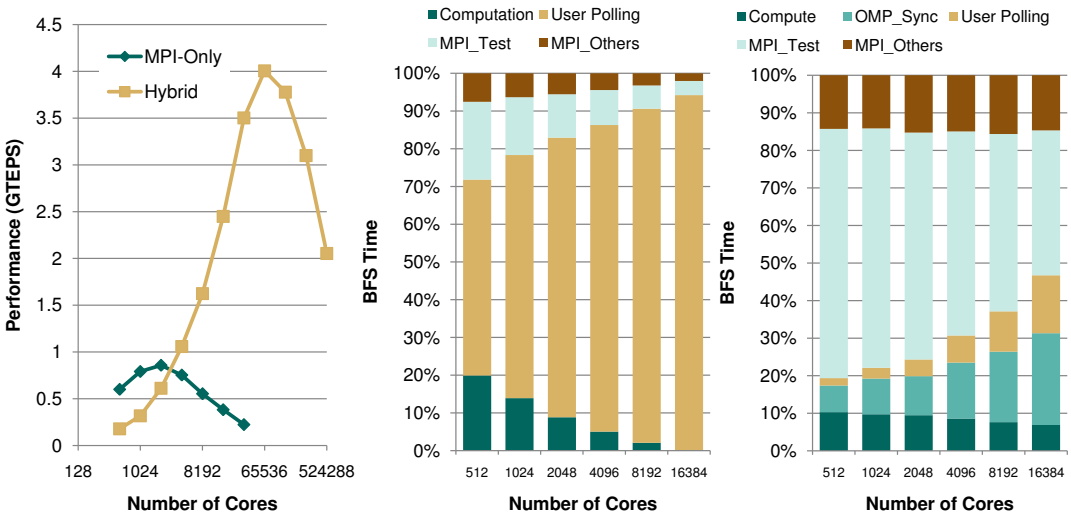
by the latter method is higher than what is shown because bigger problems can be run and better results were observed with 32 threads instead of 16. Thread-safety in MPICH is guaranteed through a global critical section. Although MPICH supports fine-grained critical sections on Blue Gene systems, it has a higher overhead on the fast path, that is, the execution path free of lock contention. Hence, without proof of contention in the runtime, the application developer is advised to configure MPICH with the global critical section support.

2.1 Preliminary Evaluation

We performed a comparative analysis between the baseline process implementation and our hybrid method. We show in Figure V.2a weak-scaling performance results. We make the following observations: (1) at small scale, the hybrid method performs worse than MPI-only; and (2) at larger scale (≥4K cores), the MPI-only implementation stops scaling, whereas the hybrid method performs better and stops scaling only after $64K$ cores. We do not show data points after $32K$ cores with the reference implementation because it runs out of memory. This issue arises during the graph construction when using a flat-MPI model and has been reported by previous authors [81]. The trend is obviously toward worse performance. Most of the inefficiency of the hybrid method at small scale stems from runtime contention and will be discussed in Section 2.4.

To understand the source of the performance breakdown, we profiled the execution of the previous experiment. We show the results in Figures V.2b and V.2c. Thread load imbalance is estimated by the average time spent between the end of the global synchronization step and the end of the parallel region and is shown as `OMP_Sync` in the hybrid model profiling figures. The non-overlapped communication cost is estimated by summing the time spent in the MPI runtime (`MPI_Test`, and `MPI_Others`[2]) and the

[2] `MPI_Isend`, `MPI_Irecv`, `MPI_Allreduce`, and other routines involved in the global synchronization



(a) Performance comparison **(b)** Profile for MPI-only **(c)** Profile for Hybrid

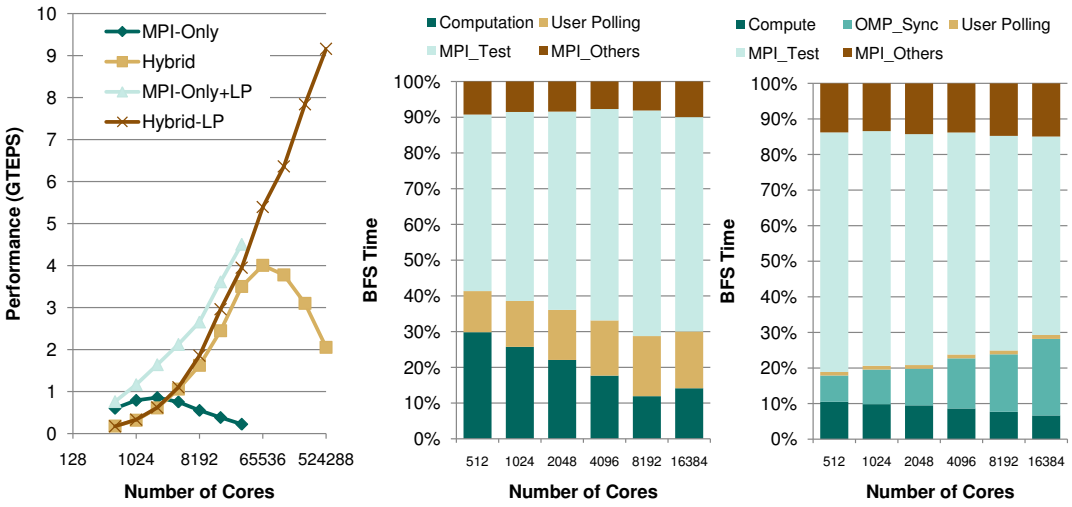
Figure V.2: (a) Preliminary weak-scaling performance results with problem sizes from 25 to 35, with 16 processes per node (PPN) for MPI-only and 1 PPN and 16 TPN (threads per node) for the hybrid version. (b) Execution breakdown of the weak-scaling experiment for the MPI-only model. (c) Execution breakdown of the weak-scaling experiment for the hybrid model.

time spent polling at the user level. Figure V.2b shows that the main bottleneck in the reference implementation is polling for communication progress outside the MPI runtime. The analysis in Section 1.3 showed that the difference in communication volume between the two methods is small and does not justify such a gap in the communication cost. We notice that the loop for checking outgoing requests in the communication progress routine (Algorithm 3) scales as $O(P)$. Although the same routine is used by the hybrid solution, it is more scalable because P is smaller by a factor equal to the number of cores per node. Nevertheless, since the hybrid method scales with the number of nodes, this issue is only delayed; and the model breaks down at $64K$ cores. Hence, fixing the root issue is necessary.

2.2 Reducing Synchronization between Endpoints

Polling for outgoing requests completion with `CheckRequests` is performed at several execution points. In short, the algorithm eagerly checks the requests in order to mark buffers as *free* as soon as possible. We point out that doing so is not necessary, however, because a buffer may effectively be needed only at a later time. Although regularity in checking for incoming messages is essential, for outgoing messages we propose a heuristic that delays polling until the buffer is needed. That is, we avoid checking outgoing requests

implementation of Section 2.3



(a) Performance comparison **(b)** Profile for MPI-only **(c)** Profile for Hybrid

Figure V.3: Weak-scaling results and profiling after using the lazy polling method.

at each iteration of the main loop and at the global synchronization step. In addition, waiting for a buffer to be freed involves only the corresponding outgoing request and avoids polling for $O(P)$ requests. We refer to this method as *lazy polling (LP)* and illustrate the changes to the routines in Algorithm 6. The performance gain of this optimization is shown in Figure V.3a. We notice that with LP, both the MPI-only and MPI+threads methods are more scalable. The profiling results in Figures V.3b and V.3c confirm that the *lazy polling* policy substantially reduces the polling overhead. We also notice, however, that significant time is spent in communication with both methods.

Algorithm 6: Lazy polling implementation

```
1 CheckIncomMsgs:
2     if (TestRecv()) then                          ▷ Test Recv requests
3         for ((v,u) ∈ RecvBuff) do Update(u,v) ;
4         Irecv(AnySource) ;
5 WaitPendSend:
6     while (Pending Send to p) do
7         CheckIncomMsgs                ▷ Make progress on Recv;
8         if (TestSend(p)) then                     ▷ Test Send request
9             FreeBuff(p)
```

2.3 Efficient Global Synchronization

As mentioned in Section 1.3, the original design incurs an overhead due to empty messages that scales as $O(M^2N^2)$. Although the hybrid method reduces this overhead by a factor M, it is still not scalable because the overhead grows quadratically with the number of nodes. In Figure V.4, we depict the distribution of the messages in a BFS run according to their type—*full*, *incomplete*, and *empty*—in a weak-scaling experiment. Ideally only full messages would be exchanged. We observe, however, that the ratio of full messages decreases at scale, even though we performed a weak-scaling experiment. We also confirm experimentally that the multithreaded implementation inherits the same issue but incurs fewer empty messages than does the MPI-only method. These results encourage implementing a better global synchronization algorithm.

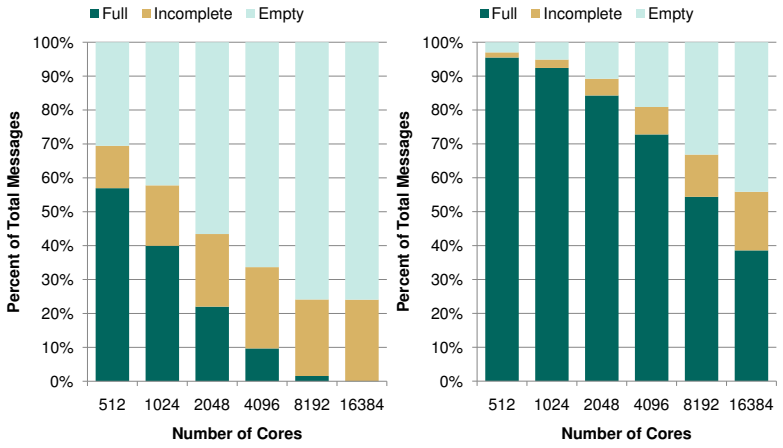
The difficulty here is to ensure a global barrier-like synchronization while processing incoming messages to avoid deadlocks. We propose the new `Synchronize` routine implementation in Algorithm 7. Our solution assumes the availability of an implementation of the recently released MPI-3 standard, which supports nonblocking barriers. Most supercomputers, including Blue Gene/Q, Cray, and InfiniBand platforms, support MPI-3 at this point. In addition, since handling incoming messages involves computation (`Update` operation) and internal MPI processing, the hybrid method uses multiple threads during this step. Here, a single thread is responsible for calling the nonblocking barrier. When one of the threads detects the barrier completion, it sets the `done` flag that signals the end of the synchronization step for the other threads. Assuming that a barrier implementation incurs $O(PLogP)$ message exchanges, where P equals the number of processes, we estimated the cost of the global synchronization as $NLogN$ instead of the original MN^2 for the hybrid implementation.

We measured performance and profiling data after using the optimized global synchronization [3], as shown in Figure V.5. We observe in Figure V.5a that the scalability of both methods has improved.

Algorithm 7: New global synchronization implementation

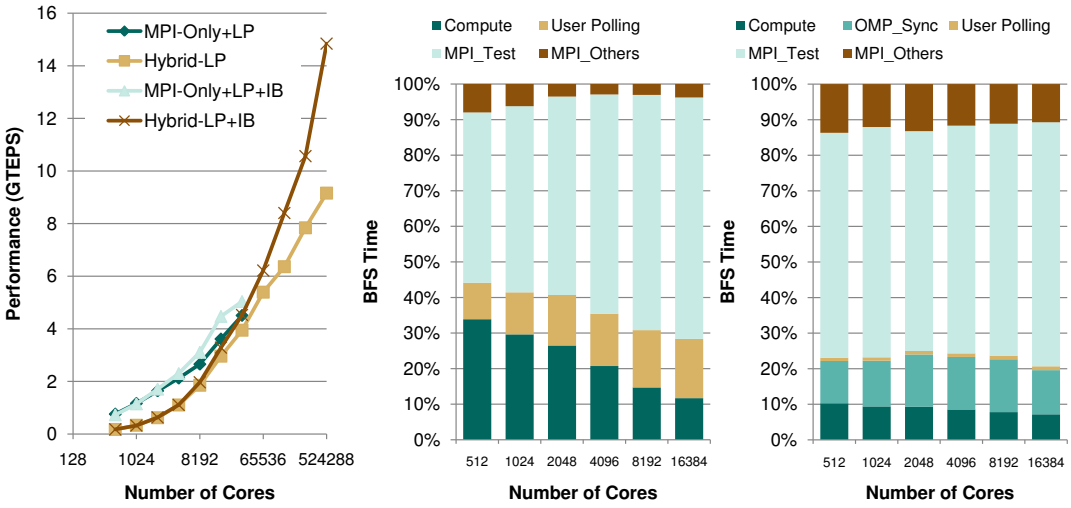
```
1 SYNCHRONIZE:
2     done ← False;
3     OMPBARRIER()              ▷ Threads must complete sending;
4     IBARRIER()                ▷ Executed by a single thread;
5     while (done = False) do
6         CHECKINCOMMSGS()              ▷ Make progress on Recv;
7           ▷ Test barrier Completion by a single thread done ← TESTIBARRIER();
```

[3] Referred to as IB as in `Ibarrier`



(a) MPI-only **(b)** Hybrid

Figure V.4: Distribution of the messages according to their content in a weak-scaling experiment with problem sizes from 24 to 30 and with 256 edges per message. *Empty* messages stem from `Synchronize`, and *incomplete* messages result from flushing the last vertices inside the buffers at the end of each level.



(a) Performance comparison **(b)** Profile for MPI-only **(c)** Profile for Hybrid

Figure V.5: Weak-scaling performance and profiling results after using a nonblocking barrier to implement the global synchronization step.

2.4 Reducing MPI Runtime Contention

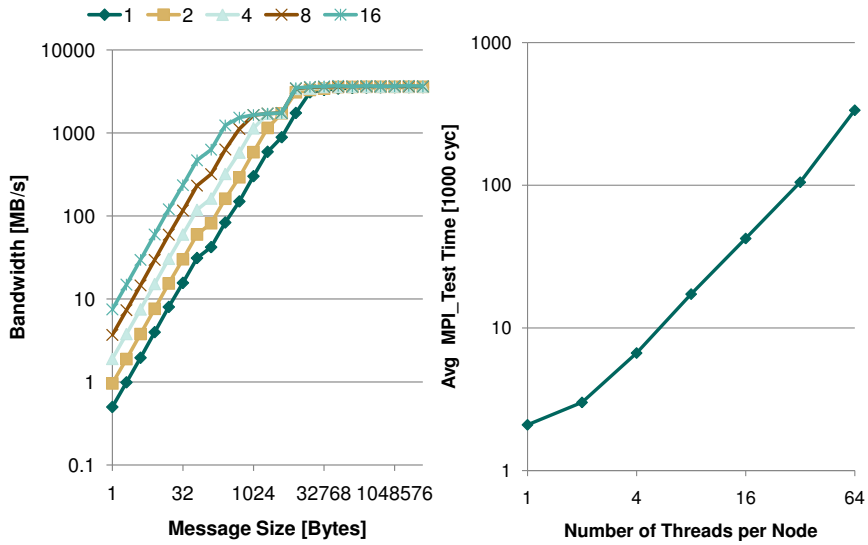
Although the multithreading version is more scalable, it performs worse than the MPI-only method at a smaller scale. Despite the reduction in execution time, however, the execution breakdown in Figures V.5b and V.5c shows that significant time is still spent in communication. In particular, the communication costs for both methods are comparable, which contradicts our analysis, in which we concluded that the communication costs

should be higher for the MPI-only method.

We therefore hypothesized that either the multitheaded communication is serialized at the network level or the threads suffer contention at the software stack. To test the first hypothesis, we measured the concurrent bandwidth when varying the number of cores (one process per core). The results are shown in Figure V.6a. We observe that by driving the communication using multiple cores, throughput can be improved by more than an order of magnitude as compared with a single core. This result disproves the first hypothesis.

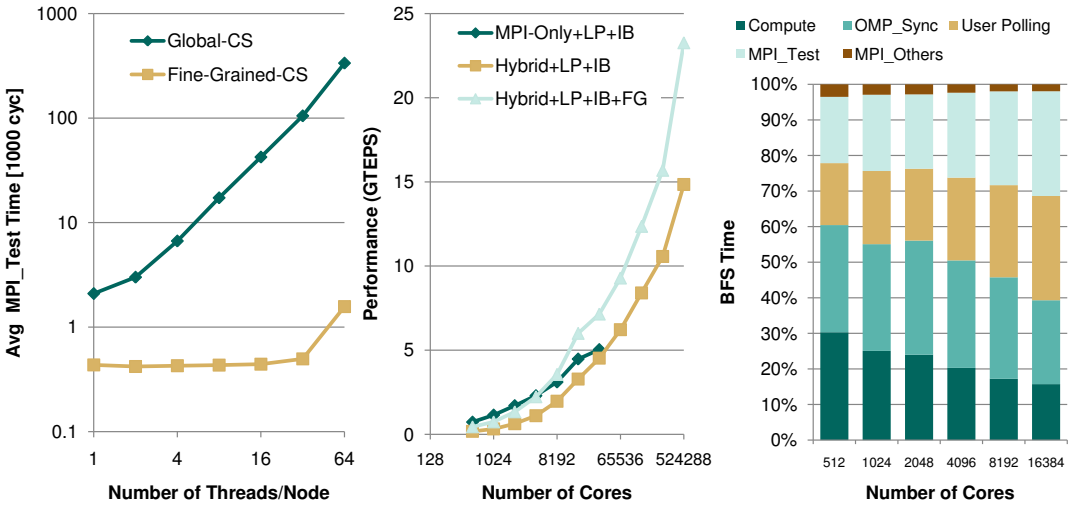
To test the second hypothesis, we measured the average time per `MPI_Test` call during a BFS run and plotted the graph in Figure V.6b. We notice that the time per `MPI_Test` scales with the number of threads, suggesting contention occurs in the MPI runtime. We remind that MPICH was built by using a *global critical section (CS)* internally to optimize the fast path. However, the proof of contention encourages the use of fine-grained critical sections. Previous works already targeted reducing runtime contention through fine-grained concurrency in the MPICH runtime and obtained significant improvement in both multithreaded communication throughput and latency [14, 33]. Because of the cost of implementing fine-grained critical sections, however, most production MPI implementations rely on coarse-grained locking; to the best of our knowledge, only MPICH on Blue Gene systems supports fine-grained locking in production environments. To enable a more efficient multithreaded runtime on a wider range of architectures including commodity HPC clusters, we investigated a different, but complementary, approach that reduces contention regardless of the CS granularity in the next chapter. We showed that by using locks with CS arbitration policies different from that of Pthread mutex, contention can be reduced substantially. Indeed, several fold improvements were observed with several benchmarks and applications.

Hence, since our platform is a Blue Gene/Q system, we rebuilt MPICH to support *fine-grained* (FG), or per-object, critical sections. We profiled the performance of `MPI_Test` with our hybrid method and show comparative results in Figure V.7a. We notice that the performance improves considerably, with the cost of polling with `MPI_Test` being almost constant. The overall performance is improved and outperforms the MPI-only approach (Figure V.7b), providing a total performance improvement of *35-fold* on 16K cores over the original MPI-only approach. The final profiling data in Figure V.7c shows that the communication cost is reduced considerably, while the time spent outside the MPI runtime dominates the overall execution time. We observe, however, that for fewer than 4K cores the hybrid approach is slightly worse than MPI-only, indicating that the hybrid method may still suffer from contention and incurs overheads, such as thread load-imbalance, shown in Figure V.7c.



(a) Concurrent Bandwidth **(b)** `MPI_Test` performance

Figure V.6: (a) Point-to-point bandwidth on a Blue Gene/Q system with respect to the number of cores involved in the communication. The data was obtained by using the `osu_mbw_rm` benchmark from the OSU microbenchmarks suite (http://mvapich.cse.ohio-state.edu/benchmarks/). (b) `MPI_Test` performance with the BFS hybrid method when scaling the number of TPN with 512 cores



(a) `MPI_Test` performance **(b)** Performance comparison **(c)** Profile for Hybrid

Figure V.7: (a) `MPI_Test` performance when scaling the number of TPN with 512 cores. (b) Weak-scaling performance comparison after using fine-grained critical sections (FG). (c) Execution breakdown after using fine-grained critical sections.

3 Hybrid BFS Related Work

Many parallel BFS implementations have emerged during the past few decades, most of which use a hybrid MPI-threads model. Koji et al. [81] proposed an MPI+OpenMP parallelization with advanced 2D partitioning, cache-friendly optimizations, and compression techniques to reduce remote communication time. Satish et al. [69] used their highly optimized shared-memory BFS as a basis for a distributed solution that also uses some compression techniques and handles manually the progress on communication by means of a communication thread. Lv et al. [53] used a design in which the master thread handles communication and traversal threads perform the computation. From the perspective of programmability, these latter solutions add a layer of complexity because of the heterogeneity created by classes of communication and computation threads and the producers-consumer relationship between them. Moreover, we point out some drawbacks such as losing some threads for computation, and possibly underutilizing the network resources since not all cores are involved in the communication. Our implementation can be combined with previous approaches for more scalability and greater performance.

The goal of these previous works was to traverse graphs at the highest possible rate. Our goal is the comparison of two programming models instead, and performance measurement were just used for comparison purposes.

4 Concluding Remarks

We studied in this chapter the MPI-only and hybrid MPI+threads models using the BFS algorithm at very large scale. In our hybrid BFS implementation, threads are the main units that handle both computation and communication concurrently. The motivation behind this design is to offer a better trade-off between memory usage, intranode parallelism, and communication performance than what the MPI-only model offers. As a result, we exposed many important parameters that users need to take into account when choosing the best model that fits the application and target platform. Such parameters are the scale of the target machine, where larger scale favors the hybrid model because of the superior node-to-node communication model, and the importance of runtime contention when threads participate in communication. However, the hybrid model is not a silver bullet that fixes all scalability issues. Although it reduces some of the overheads associated with interprocess communication, thus delaying some of the scalability bottlenecks, it does not completely avoid them. We therefore proposed various enhancements and, through a detailed experimentation and analysis, demonstrated that our techniques can reduce the overheads at very large scale and improve the performance of BFS by

35-fold cores on 16K cores while scaling to 512K cores of a Blue Gene/Q system.

This study revealed contention in the MPI runtime as an important bottleneck that may occur when multiple threads access the MPI layer. Fine-grained thread-safety proved to be effective on the BG/Q system, however, other platforms do not support this level of granularity. We address this challenge in the next chapter and propose solutions to the runtime contention that are accessible on many platforms.

Chapter VI

Mitigating Contention in Multithreaded MPI Runtimes

The MPI standard defines how hybrid applications interoperate with an MPI library. In particular, MPI implementations that offer multithreading support must guarantee thread safety. Such thread-safety is guaranteed through a combination of processor atomic operations and critical sections in virtually every MPI implementation today. In order to maintain performance, however, two orthogonal dimensions of optimizations need to be investigated—(1) granularity of the critical sections and (2) the synchronization mechanism used by those critical sections. The synchronization mechanism itself has two important properties. First, the latency of handing-off the critical section to the next thread, which was investigated for many decades by various researchers and led to several scalable locks and other synchronization mechanisms. Second, the resulting arbitration of a critical section, i.e. the access order of the threads to the critical section. Most MPI implementations use Pthread mutex-based coarse-grained critical sections due to its portability and relative simplicity, though some implementations have investigated fine-grained critical sections as well [43, 14, 33]. However, all existing implementations ignore the orthogonal dimension of critical section arbitration.

This chapter summarizes and extends on the work titled "MPI+threads: runtime contention and remedies" [10]. Here, we investigate how critical section arbitration can affect communication performance.

1 Thread Safety in MPI

In this section we provide background information about thread safety and its relation to MPI, and we discuss thready-safety efforts in MPICH.

1.1 MPI Requirements for Thread Safety

The MPI standard defines four levels of thread safety: `MPI_THREAD_SINGLE`, `MPI_THREAD_FUNNELED`, `MPI_THREAD_SERIALIZED`, and `MPI_THREAD_MULTIPLE`. These levels are listed in increasing order of thread safety. Given the performance costs of thread safety, these levels allow adjusting the MPI implementation to the needs of the target application without incurring unnecessary overheads. In this work, we target the least restrictive level, `MPI_THREAD_MULTIPLE`, which allows multiple threads to call MPI routines concurrently.

Many aspects are considered when designing a thread-safe MPI library. Critical-section granularity is an important parameter that influences the degree of parallelism allowed for concurrent thread executions. That is, the longer a critical section is, the more it incurs serialization and thus hinders parallel performance. In previous works, the authors investigated different levels of critical-section granularity and their implications in terms of performance and implementation complexity [14]. Figure VI.1 summarises the different levels of granularity explored by Balaji et al. We observe that the granularity becomes increasingly fine-grained from left ("Global") to right ("Lock-free"). In practice, most MPI implementations use a combination of "Global" and "Lock-free", with the latter applying to atomic updates of reference counters that are required for managing internal MPI objects. A combination of fine-grained locking and lock-free granularity was implemented in MPICH for Blue Gene/Q systems [33], which was used in Chapter V of this document.

Regardless of the granularity, however, contention to enter a critical section can still occur, and serialization is inevitable. Several mechanisms, such as mutexes and spin-locks, have been developed to implement mutual exclusion in order to protect a shared resource. Such mechanisms usually differ in the way they handle contention, the unnecessary lock/unlock overhead they incur in scenarios where there is no contention, reducing synchronization granularities, and time to transfer ownership of the lock. However, the serialization order of threads accessing a critical section and its effect on an MPI runtime performance are less understood. To the best of our knowledge, no existing work has studied thread synchronization in communication runtimes from this perspective.

1.2 Thread Safety in MPICH

In this section, we analyze thread-safety in MPI implementations in the context of MPICH, though the analysis is largely true for most other MPI implementations as well. On our platform, MPICH thread safety is ensured by using a Native POSIX Threading Library (NPTL) [59] global pthread mutex. Locking a default mutex in NPTL is im-

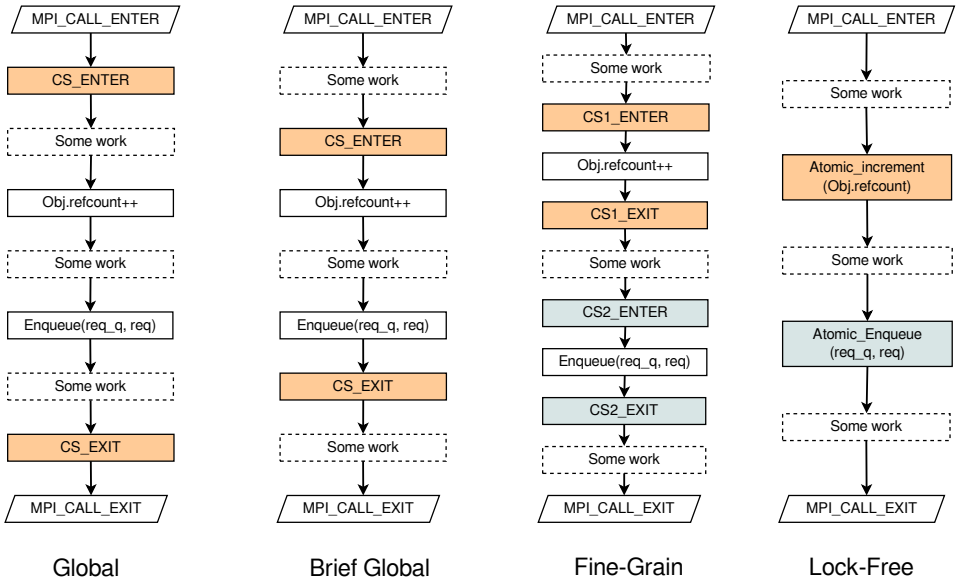


Figure VI.1: Critical-section granularity. `CS_ENTER` and `CS_EXIT` denote the protocol operations executed at the entry and exit of a critical section, respectively. `Atomic_OP` refers to a lock-free implementation of the operation `OP`.

plemented as follows. First, a thread tries to acquire the lock in the user space with an atomic compare-and-swap operation. If not successful, the thread goes to sleep in the kernel space using the `FUTEX_WAIT` operation of the futex (fast user-space mutex) system call [37, 34]. The mutex holder wakes up at most one thread in the futex queue, with the `FUTEX_WAKE` operation, when leaving the critical section. The thread that wakes up again competes to acquire the lock and the same process repeats if the lock acquisition fails.

From an arbitration perspective two distinct arbitration policies can be identified. In the kernel space the futex wake-up operations are performed according to the kernel-scheduling policy. On most, if not all, HPC systems, users do not have privileges to alter the priorities of their processes and threads. Thus on these systems, to the best of our knowledge, the scheduling policy is first-in first-out. In the user space, however, the policy obeys the fastest-thread-first rule because it is based on compare-and-swap operation that does not offer any means to order the threads. We note two major implications. First, the overall arbitration is dominated by the user-space policy since threads must try to acquire the lock every time they leave the kernel space. Second, the current user space arbitration model is unfair since no mechanism to maintain fairness is used. In Section 3, we characterize fairness in the MPICH runtime and analyze the effects of the Pthread mutex arbitration on communication performance.

Table VI.1: Target machine specifications

	Fusion	Tsubame Thin Nodes	Tsubame Fat Nodes
Architecture	Nehalem	Nehalem	Nehalem-EX
Processor	Xeon E5540	Xeon 5670	Xeon 7550
Clock frequency	2.6 GHz	2.93 GHz	2.0 GHz
Number of sockets	2	2	4
Cores per socket	4	6	8
L3 Size	8 MB	12 MB	18 MB
L2 Size	256 KB	256 KB	256 KB
Number of nodes	310	1408	10
Interconnect	InfiniBand QDR	InfiniBand QDR	InfiniBand QDR

2 Testing Platforms

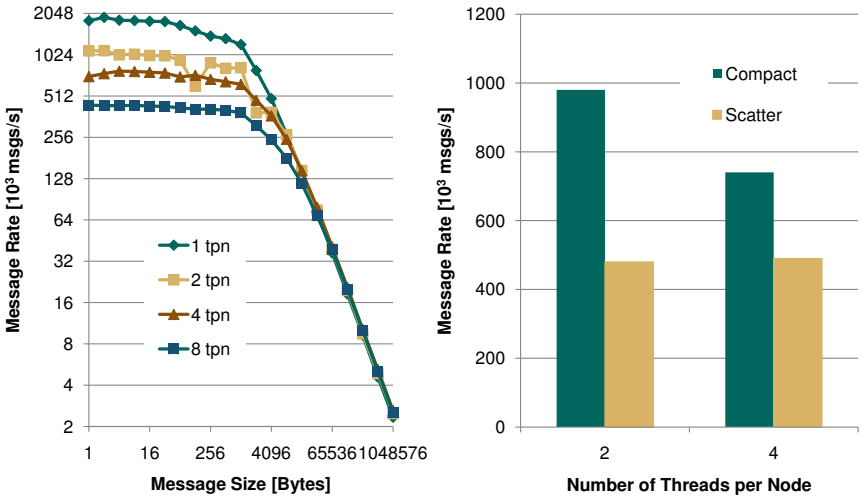
We perform most of our experiments on an Intel processor-based cluster named Fusion at Argonne National Laboratory, where each node is equipped with 2 Nehalem processors each with 4 cores and the nodes are interconnected with a Mellanox InfiniBand QDR fabric. We also report several results on the Tsubame 2.5 supercomputer at Tokyo institute of Technology. This latter machine has a higher core density, with 12 cores and 32 cores/node for the thin and fat nodes, respectively. The detailed specifications of these machines are summarized in Table VI.1. We used MPICH 3.2a1, which is configured to run with the Mellanox Messaging Accelerator (MXM) interface.

3 MPI Runtime Analysis

In this section we first consider MPICH as a black box and perform a simple performance evaluation and analysis in a multithreaded scenario on the Fusion cluster. We then present a detailed profiling in order to shed light on the runtime inefficiencies. Unless specified otherwise, we bind the first four threads to cores on the first socket and the rest to cores on the second.

3.1 Initial Performance Evaluation

Here we present a simple evaluation that reflects the runtime contention experienced with MPICH. We measure the point-to-point throughput using a benchmark derived from `osu_bw` of the OSU microbenchmarks suite [2]. We modified the benchmark to fork a team of threads to perform the communication in parallel. Since threads are not doing computation and MPICH uses a global critical section, we expect to see serializaction in communication which would result in no speedup. Figure VI.2a shows the message rate as a function of message sizes and the number of threads per node. We observe performance



(a) Multithreaded throughput performance with varying message sizes; *tpn* stands for thread(s) per node

(b) Effect of thread concurrency and NUMA on probability performance

Figure VI.2: Preliminary evaluation of multithreaded communication performance

degradation proportional to the number of threads reaching up to a four-fold reduction in throughput. This experiment shows that there is contention in the MPI runtime although it does not expose the primary source of that contention. For large messages, network communication delay dominates and renders runtime inefficiencies negligible.

3.2 Impact of NUMA on Critical Sections

Here we present the results of our experiments on throughput performance when threads are bound to cores on the same socket (`Compact`) vs. when they are bound to cores on different sockets (`Scatter`). The goal was to determine the effect of nonuniform memory access (NUMA) on the runtime contention. The results are shown in Figure VI.2b. We observe that throughput is 1.5 to 2-fold worse with a scatter binding. This suggests that the runtime contention is sensitive to NUMA. Given that mutex uses a compare-and-swap operation in user space, we point out two major drawbacks that are amplified by NUMA. First, the hand-off time[1] between threads is amplified by the intersocket latency. Second, since mutex does not guarantee fairness, ownership of the lock may not be passed fairly between threads. We speculate that NUMA biases the arbitration in favor of the threads on the same socket—more analysis on this speculation is presented in the next Section.

[1] The elapsed time between when a lock holder marks the lock as free and when the next owner detects it

3.3 Arbitration Fairness Analysis

The speculation of critical section arbitration being biased by NUMA architectures, as mentioned in Section 3.2, stems from the nonuniform proximity of threads to the memory hierarchy (caches and memory). Specifically, the thread that releases the lock dirties the cache line holding the lock, which makes it most favorable for other threads closest to this cache to acquire the lock.

To assess the degree of unfair arbitration, we manually instrumented MPICH to trace the lock acquisition, processed the collected data, and compared mutex to an oracle fair arbitration to analyse to which extent mutex biases the arbitration. We further considered the role of the memory hierarchy by analysing two levels at which the arbitration may be biased. At the *core level*, given T threads waiting to enter the critical section, we estimate the probability P_c that the same thread reacquires the lock successively. At the *socket level*, given N sockets, T_i the number of threads on socket i waiting to enter the critical section, we estimate the probability P_s that the new owner of the lock runs on the same socket j as the previous owner. We statistically estimate these quantities for both mutex and a fair arbitration as follows:

$$P_c = (\sum_{l=1}^{L} X_l)/L$$
$$P_s = (\sum_{l=1}^{L} Y_l)/L$$

Where, for a mutex arbitration:

$$X_l = \begin{cases} 1 & \text{if same owner as } l-1 \\ 0 & \text{else} \end{cases}$$
$$Y_l = \begin{cases} 1 & \text{if same socket as } l-1 \\ 0 & \text{else} \end{cases}$$

and for a fair arbitration:

$$X_l = 1/T_l$$
$$Y_l = T_{j,l}/(\sum_{i=0}^{N-1} T_{i,l})$$

We discretized the execution at the lock acquisition level using the subscript "l" in our equations. That is, at each lock acquisition, we take a snapshot of the critical section in terms of the number of threads blocked at the entrance. L denotes the total number of lock acquisitions and X_l and Y_l are the probability of electing a thread running on respectively the same core and the same socket during lock acquisition l. T_l and $T_{i,l}$ represent respectively the total number of waiting threads and the number of waiting threads on socket i during lock acquisition l. We then analyzed the arbitration with the

point-to-point throughput benchmark on our dual-socket system for each message size. We compute P_c and P_s using the previous equations and derive the ratio of the mutex probabilities over the fair arbitration probabilities that we refer as *Bias Factors*. Theses factors indicate to which degree mutex biases the arbitration where a fair arbitration would have a bias factor of 1. The results of our analysis are shown in Figure VI.3. We notice that mutex biases the arbitration by 2x at the core level and 1.25x at the socket level on average across message sizes. These results confirm our assumption regarding the unfair arbitration and our belief that NUMA amplifies the problem.

In the next section we relate the unfair arbitration to the MPI implementation internals and thus explain the performance consequences.

3.4 Message Requests Analysis

Before we discuss our profiling method, we first describe how requests are handled inside the runtime. For simplification, we consider only nonblocking receive requests. Figure VI.4a shows the state diagram of a receive request. When a user calls `MPI_Irecv`, a request is issued internally. If the corresponding message was already received in the *unexpected queue,*[2] then the request is marked as completed. Otherwise, the request is posted in the *posted queue*. Later, if a message arrives and matches the request, it will be marked as completed. `MPI_Wait` or `MPI_Test` or their derivatives will be called and block the caller until the request is found completed and then gets freed.

The point-to-point throughput benchmark performs two-sided nonblocking communication (`MPI_Isend` and `MPI_Irecv`) on a window of 64 requests, and `MPI_Waitall` is used to wait for all the requests in batches (see Figure VI.4b). In our multithreaded version each thread manages its own window. In order to make progress on communication, all requests need to be detected as completed and then freed before moving to a new window. In a multithreaded scenario, threads can mark other threads' requests as completed inside the runtime, but only the thread that waits for the completion of a request can free it. In addition, we do not tag messages so that threads can match any message from the same process and communicator.

Our profiling method relies on the notion of *dangling requests*, that is, requests that are completed but not yet freed. We track the number of these requests at the lock acquisition granularity. We then define our metric as the average number of dangling requests during the program execution. To make rapid progress on communication, threads should detect completed requests early, free them, and generate new requests to feed the runtime and the network. Thus, this metric should be kept low. In fact, with a fair arbitration, one

[2] A queue for incoming message handlers without matching receive requests

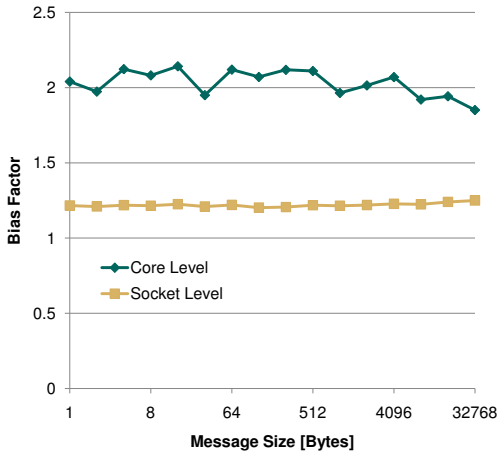
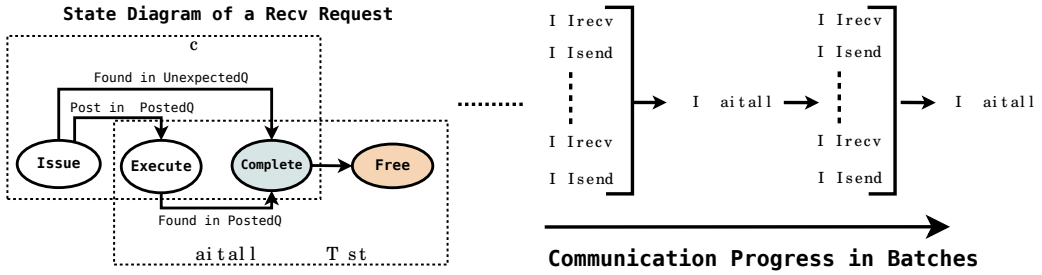


Figure VI.3: Mutex arbitration fairness analysis at the core and socket levels with the point-to-point throughput benchmark



(a) State machine of a receive request

(b) Comunication progress with a window of requests

Figure VI.4: MPI internal details in a throughput oriented scenario

would expect this metric to be low since requests will be issued in batches, communicated through the network, and detected by the threads evenly. The profiling results with the throughput benchmark are plotted in Figure VI.5. We notice that the number of dangling requests is high (approx. 40% of the maximum number of requests that can exit inside the runtime). The reason is that the window of a starving thread will likely take longer to be filled with completed requests and that the requests inside incomplete windows are counted as dangling. A chain reaction will result by delaying the next requests to be issued and thus their network communication and the matching process. Consequently, the overall communication progress will be slowed.

4 Reducing Contention

The analysis in the previous section confirmed that threads have unfair access to the critical section. Here we explore two alternative locking methods that take into account

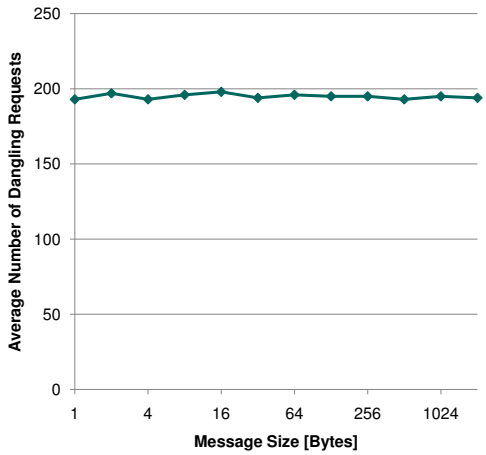


Figure VI.5: Dangling requests analysis with the point-to-point throughput benchmark

fairness and communication progress in the arbitration policy.

4.1 First-In First-Out Arbitration

The first alternative ensures fairness through a first-in first-out critical section arbitration. In Section 1 we pointed out that changing the scheduling policy in the kernel space does not affect the arbitration in the user space. To alter the lock arbitration, we rely on a locking method that is entirely implemented in the user-space through busy waiting. Several locking methods, for example, MCS [56] and ticket [55], establish a first-in first-out threads ordering in the user space. A recent study by David et al. showed that the *ticket lock* performs well on most modern many-core architectures and in various contention scenarios [27]. The principle behind the ticket lock is simple. Each thread gets a ticket at the entry of a critical section and waits for its turn. Figure VI.6 shows a basic implementation. We note that only one atomic operation (`fetch_and_increment`) is needed. In addition, the busy-wait condition does not involve extensive cache traffic, unlike a `compare_and_swap`–based spinlock. Thus, our first step is to use a ticket lock to implement the MPICH global critical section.

After integration in the MPICH runtime, we compare the new design to the mutex-based runtime by analyzing dangling requests. The results are shown in Figure VI.7a. We observe that using ticket keeps the number of dangling requests very low and, according to our analysis, should have a positive effect on performance. We investigated the effect of the degree of concurrency and the NUMA effect by scaling the number of threads per core. The results are shown for compact and scatter bindings in Figure VI.8. In a compact binding, the ticket method reduces contention compared with mutex and improves the performance by 68% with four threads. With scatter binding and two

```
int next_ticket;
int now_serving;

ticket_acquire_lock()
{
  my_ticket = fetch_and_incr(next_ticket);
  /* Wait for my turn */
  while(my_ticket!=now_serving) ;
}

ticket_release_lock()
{
  now_serving++;
}
```

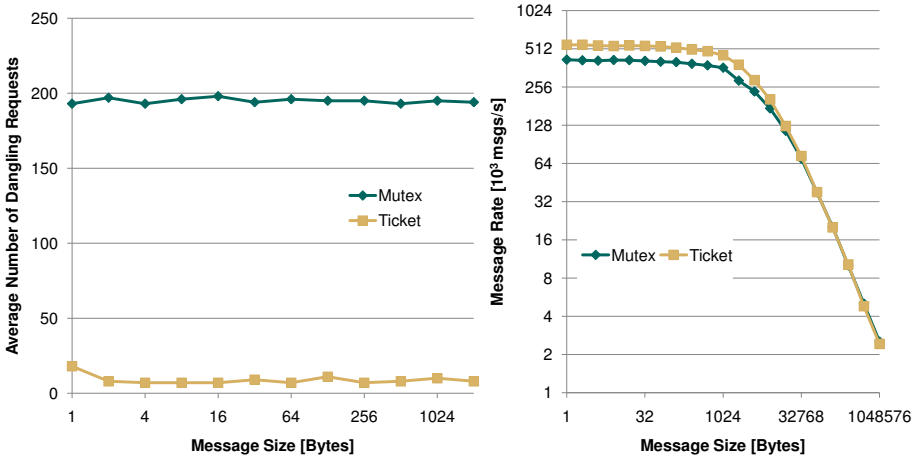
Figure VI.6: Pseudo-algorithm for a ticket lock

threads per node, the ticket method loses slightly to mutex, suggesting that the ticket method incurs more intersocket synchronization. Indeed, the mutex socket-level lock monopolization exhibits more locality of reference and reduces consequently the amount of intersocket synchronization compared with that of the ticket method. However, the benefit of fair arbitration grows with the degree of concurrency. We conclude that for a scatter binding and a low degree of concurrency, the overhead of intersocket thread synchronization may outweigh the benefit of fair arbitration. A common method for solving this issue is to spawn one process per socket and to use threads within a socket to avoid intersocket synchronization and data movements. In such cases, the ticket method will be more effective than mutex. Another possibility is for the MPI implementation to use a NUMA-aware lock such as hierarchical queue locks that offer fairness and scalability [29, 20]. Figure VI.7b shows throughput comparison between the two methods with respect to the message size. We notice that the ticket method outperforms mutex by 30% on average for messages below 4 KB. The gap between the methods decreases until reaching 32 KB, from which point performance differences become negligible.

4.2 Priority Locking Scheme

In this section, we describe our custom lock implementation, which takes into account MPICH internal details to prioritize threads with higher chances of making progress. We follow with an example benchmark where this method is superior to a flat fair arbitration.

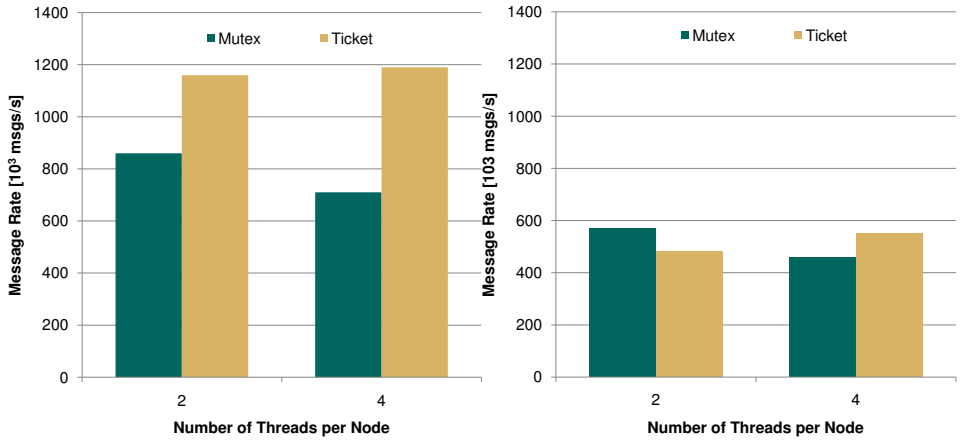
The main consequence of unfair arbitration is the possibility that threads monopolizing the lock may not yield useful work while penalizing a starving thread that can make progress. It is not trivial to know a priori which thread is going make progress after getting the lock, since this situation depends in many cases on external events such as message



(a) Dangling requests analysis

(b) Throughput performance with varying message sizes and 8 threads per node

Figure VI.7: Analysis and performance comparison between using mutex or the ticket lock with the throughput benchmark



(a) Compact binding

(b) Scatter binding

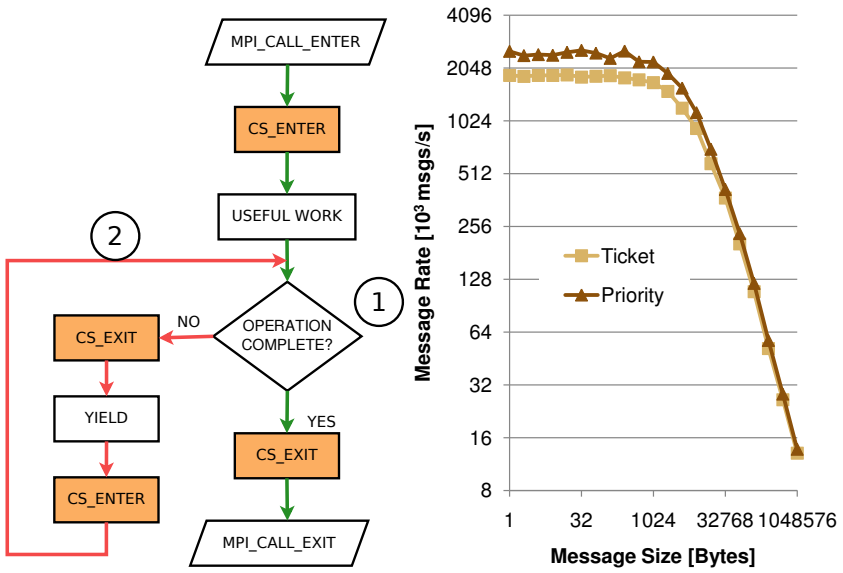
Figure VI.8: Effect of NUMA on thrpughput performance of MPICH with different lock implementations (mutex or ticket)

reception. Nevertheless, we can identify unbalanced execution paths within the runtime that yield different amounts of work within the critical section. Our examination of the MPICH runtime exposed two coarse-grained execution paths as shown in the simplified flowchart of Figure VI.9a: (1) a *main path* that each thread-safe routine implements differently and (2) a *progress loop* that some MPI routines enter to poll for communication progress. For instance, `MPI_Irecv` may allocate resources and enqueue the request in a queue in the *main path*, while not going through the *progress loop*. On the other hand,

a blocking call such as a `MPI_Wait` will enter the communication progress engine until the corresponding request completes. An important observation is that threads in the *main path* have more chances to yield useful work and thus are unlikely to waste a lock acquisition. There is an exception, however, for some MPI operations that may not make progress even in the main path such as `MPI_Test`. The ticket lock gives threads equal chances to acquire the lock independently from the path they take, for example, in cases where threads are blocked at the entry of the *main path* while waiting for threads in the *progress loop* to release the lock. Such cases may also yield to wasted lock acquisitions.

Given these observations, we propose a custom locking scheme that favors threads in the *main path*. This can be achieved by assigning a high priority to threads at the entry of MPI routines, then lowering their priorities if they enter the *progress loop*. To avoid lock monopolization, we ensure fair arbitration between threads of the same priority. To achieve this goal without using a heavyweight locking mechanism, we implement the lock acquisition and relinquish operations with ticket locks (Figure VI.10). More specifically, we use three ticket locks: `ticket_H` for the high-priority threads in the main path; `ticket_L` for the low-priority threads in the progress loop; and `ticket_B` for the high-priority threads to block low-priority ones. We emphasize that using a ticket lock for the high-priority threads to block the low-priority ones is necessary. Failing to do so may generate lock monopolization in favor of low-priority threads.

A consequence of prioritizing the main path is feeding the runtime with requests at a higher rate than with the ticket lock. Our design also maintains a low overhead in case the prioritization does not improve the communication progress by incurring at most one more atomic operation in the critical path. Thus, we expect to have some cases with performance improvements and others with performance similar to that of the flat ticket or with a slight overhead. In the following experiment we performed an all-to-all communication pattern named N2N. The benchmark is derived from the multithreaded throughput benchmark with the exception that each process sends/receives a continues stream of messages to and from all the other processes. The results in Figure VI.9b show that the priority lock improves the throughput of the N2N benchmark by 33% on average for messages below 32 KB. Here, executing the main path is more critical than with the point-to-point communication pattern. In the point-to-point throughput benchmark, threads can match any message since they all come from the same source, the same communicator, and all have the same tags. In the N2N benchmark, however, threads cannot match certain messages. For instance, if a thread is blocked at the entrance of the main path while attempting to post a receive for a certain process, another thread inside the runtime may process the incoming message and put it in the unexpected queue. As a result, the request matching is delayed, slowing the issue and matching of requests and



(a) Thread safety in a MPI runtime **(b)** N2N throughput performance with 4 processes

Figure VI.9: (a) Simplified flow diagram of a generic thread-safe MPI implementation with a global critical section. We distinguish two main paths: (1) a *main path* and (2) a *progress loop*. (b) Performance comparison of the ticket and priority locks with the N2N throughput benchmark.

thus the overall communication progress. Prioritizing the main path solves this issue by promoting request generation.

5 Evaluation

In this section we compare the performance of the original design with that of our methods using microbenchmarks, application kernels, and a genome assembly application.

5.1 Microbenchmarks

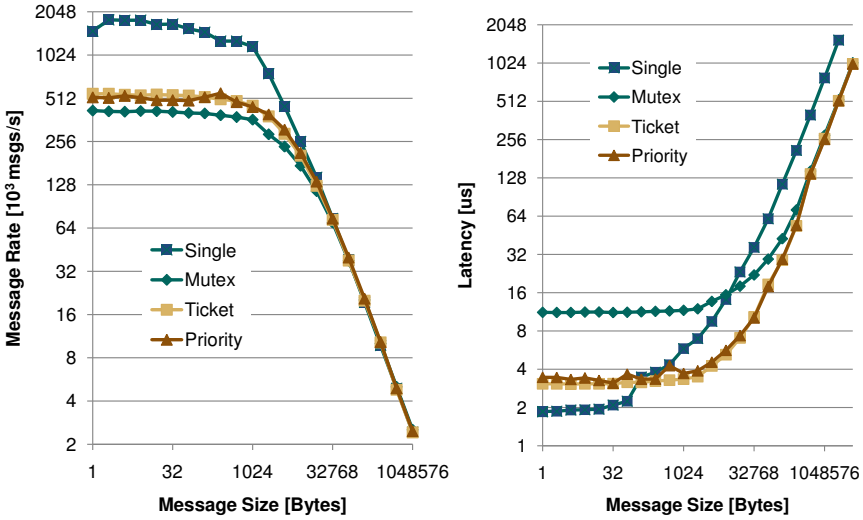
We begin by presenting results with multithreaded two-sided point-to-point throughput and latency benchmarks. We also evaluate MPI one-sided operations with asynchronous progress.

Two-Sided Communication

Figure VI.11 summarizes the results with throughput and latency benchmarks of all methods using 8 threads per node. In addition, we provide a comparison with single-threaded performance (i.e. `MPI_THREAD_SINGLE`). The latency benchmark was derived from `osu_latency` of the OSU microbenchmarks suite. We observe that the throughput

```
/* High/Low priority ticket locks*/
ticket_lock_t ticket_H;
ticket_lock_t ticket_L;
/* Lock to block lower priority  */
ticket_lock_t ticket_B;
/* Let high priority go through  */
int already_blocked;

high_priority_acquire_lock()
{
  ticket_acquire_lock(ticket_H);
  if(!already_blocked)
  {
    ticket_acquire_lock(ticket_B);
    already_blocked = true;
  }
}

low_priority_acquire_lock()
{
  ticket_acquire_lock(ticket_L);
  ticket_acquire_lock(ticket_B);
}

high_priority_release_lock()
{
  if(last high priority thread)
  {
     /* Let low priority pass */
     ticket_release_lock(ticket_B);
     already_blocked = false;
  }
  ticket_release_lock(ticket_H);
}

low_priority_release_lock()
{
  ticket_release_lock(ticket_B);
  ticket_release_lock(ticket_L);
}
```

Figure VI.10: Priority locking pseudo-algorithm with two levels of priority



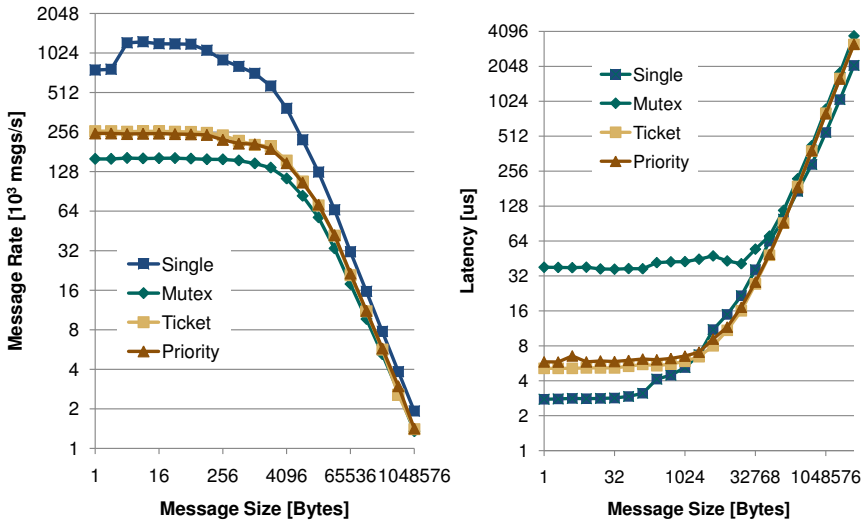
(a) Throughput **(b)** Latency

Figure VI.11: Performance comparison between mutex, ticket, priority, and single-threaded execution with the two-sided point-to-point throughput and latency benchmarks

achieved with the ticket and priority methods are similar (Figure VI.11a) and outperform mutex but are only 36% that of single-threaded performance. Figure VI.11b shows that the ticket method reduces latency by up to 3.5x over mutex. The priority method adds around 11% overhead over the ticket method for small messages but performs similarly with large messages. The latency with the ticket method is 1.66x that of the single-threaded approach with messages below 128 bytes. Surprisingly, however, multithreaded latency with the ticket and priority methods is up to 3.6x better than with a single thread for messages larger than 128 bytes. The reason is that the multithreaded communication issues several communication operations to the runtime instead of an individual operation, as in the single-threaded case, and helps feed the network resources. For small messages, thread synchronization and serialization hide this benefit. When moving to a higher core density on the Tsubame 2.5 fat nodes (32 cores/node), we notice in Figures VI.12a and VI.12b that the benefit or our methods is even higher (up to 8x improvement in latency). We notice also, however, that the gap between our methods and the single-threaded case is even larger in terms of throughput and the latency for larger messages does not improve with higher core densities.

Remote Memory Access with Asynchronous Progress

Remote memory access (RMA) provides a powerful model where processes can access memory outside their address space and even outside the physical node they are running on. This concept is at the heart of many programming models, such as Global Arrays

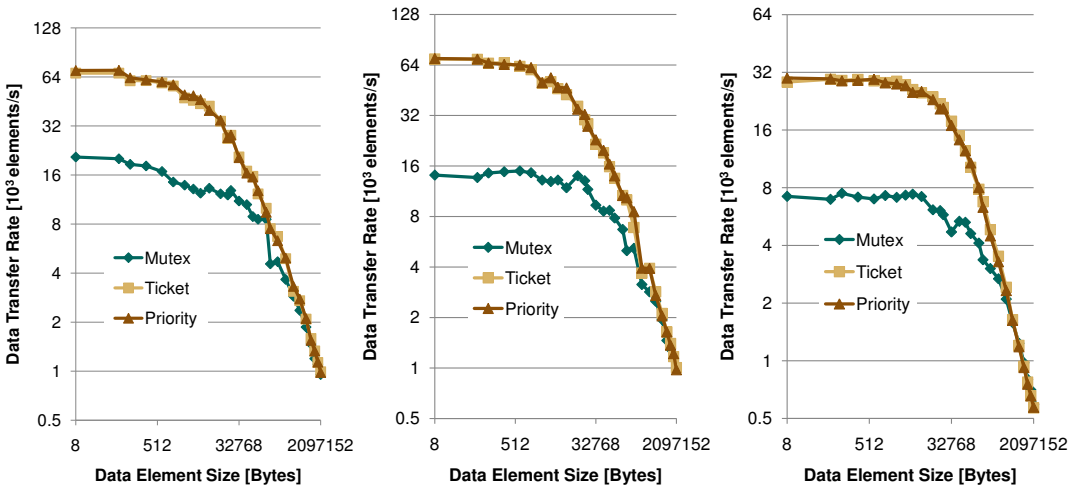


(a) Throughput **(b)** Latency

Figure VI.12: Performance comparison between mutex, ticket, priority, and single-threaded execution with the two-sided point-to-point throughput and latency benchmarks with 32 cores/node of the Tsubame 2.5 supercomputer fat nodes

(GA) [63], that offer a *shared-memory* view on top of distributed-memory systems. Here we evaluate the performance of ARMCI-MPI [30], an implementation of the Aggregate Remote Memory Copy Interface (ARMCI) [62], which uses MPI one-sided operations. ARMCI-MPI is an important interface that is used, for instance, by the chemistry package NWChem [83].

We conducted an experiment in which one process does RMA operations (put, get, and accumulate) to or from other processes on contiguous data. This benchmark is single threaded; however, we enabled MPICH asynchronous progress that triggers progress on communication in the background by forking an additional thread. Thus, when asynchronous progress is enabled, MPICH uses internally `MPI_THREAD_MULTIPLE` since two threads are running concurrently inside the runtime. The results of this experiment are shown in Figure VI.13 when running with 8 processes. Although only two threads are running concurrently, we notice a substantial performance difference between mutex and our methods. The reason is that the progress thread, which is most of the time in the progress loop, heavily monopolizes the lock since it does not do useful work most of the time. Thus, enforcing fairness produces a tremendous speedup. We also note that our methods improve performance up to 5x over mutex. Similar to the two-sided communication results, the difference between ticket and priority is not perceptible.



(a) Put **(b)** Get **(c)** Accumulate

Figure VI.13: Performance comparison of all methods when doing RMA contiguous data transfer using ARMCI-MPI with asynchronous progress

5.2 Kernels

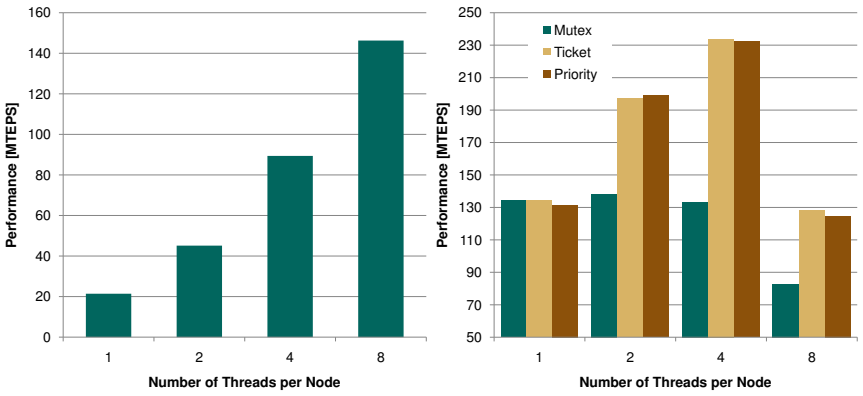
In this section we evaluate some computational kernels often encountered in real applications.

The Graph500 Benchmark BFS

The Graph500 benchmark is a communication-intensive code used for ranking large-scale systems in terms of graph processing capabilities [5]. It is composed of multiple kernels, but we consider only breadth-first search (BFS) in this work. More specifically, the baseline algorithm is an MPI-only level-synchronized BFS that relies on nonblocking point-to-point MPI communication for data exchanges. Our hybrid MPI+OpenMP implementation extends the MPI-only design by allowing multiple threads to cooperate for computation and independently communicate with remote processes. Moreover, both computation and communication are lock-free and atomic-free, inspired by the single-node implementation of Chhugani et al. [23]. Each thread maintains outgoing buffers corresponding to each remote process and one buffer for incoming messages. Threads repeatedly check for completed requests using `MPI_Test` and eventually do computation and generate new outgoing requests. For more details, the reader can refer to Chapter V.

First, we evaluated the performance of our implementation on a single node (Figure VI.14a). The problem size is expressed by the scale of the graph in terms of number of vertices.[3] We notice that our implementation scales linearly up to 4 cores and loses

[3]We use a logarithmic scale: $scale = \log_2(\#vertices)$

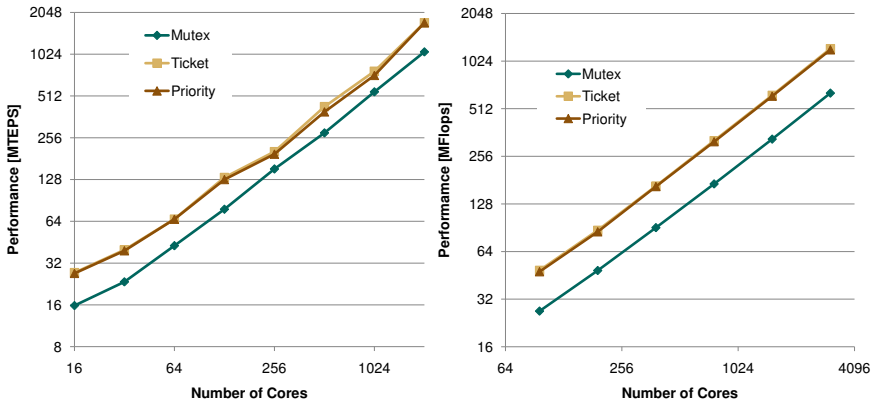


(a) Performance scalability on a single node with a problem size of 24

(b) Thread scaling performance with a compact binding, 16 processes, and a problem size of 28

Figure VI.14: Performance comparison of all methods with the Graph500 BFS kernel. In (a) the single-node results do not involve MPI processes.

10% efficiency with 8 cores. The loss in efficiency is likely due to intersocket data movements since our implementation is not socket-aware. We conducted simple thread-scaling experiments similar to those of Section 4.1. Here, the difference with the throughput benchmark is that threads are doing computation in addition to MPI communication. Consequently, the time spent by one thread inside the MPI runtime may overlap with the computation of another thread. Combined with a parallelized computation, speedups over a single-threaded execution may occur. Indeed, the results in Figures VI.14b show that speedups do occur when threads are located on the same socket and a fair lock is used with up to 4 threads. With mutex no speedup is apparent, suggesting that the unfair arbitration generates contention and consequently wastes the speedup of the parallel computation. When both sockets are involved, thread synchronization across sockets is an obvious bottleneck; but unlike mutex, our methods avoid slowdowns compared with using a single-threaded method. Figures VI.15a and VI.15b show the results of weak-scaling performance comparisons of all methods. We notice performance improvements close to 2x for our methods. As expected, the benefit of our locks are higher on Tsubame 2.5 because of the higher core density. The priority method does not show signs of superiority in this case. We explain this by the fact that threads do not busy wait in the progress loop since they use only immediate `MPI_Test` calls. That is, all threads are always in the main path and have the same high priority inside the runtime.



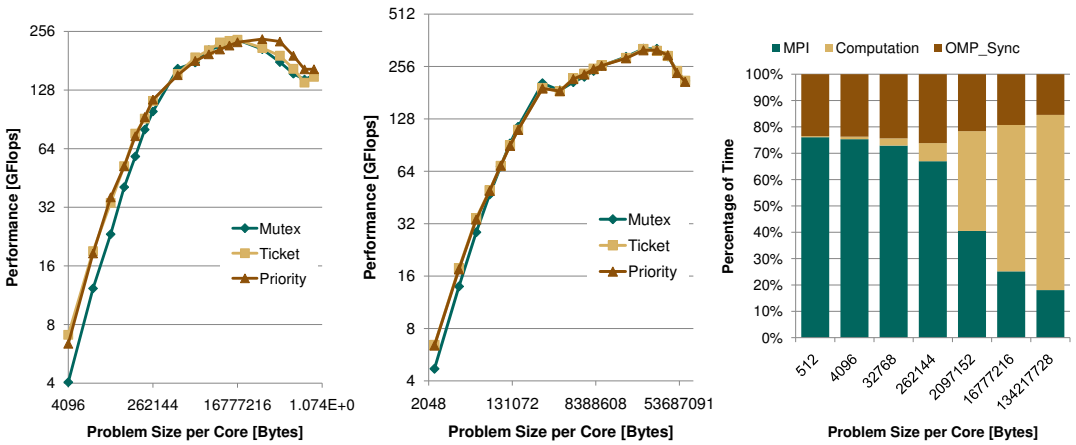
(a) Nehalem with 8 threads/node **(b)** Tsubame 2.5 thin nodes with 12 threads/node

Figure VI.15: Weak-scaling performance with one process per node and problem sizes from 25 to 32

3D 7-Point Stencil Kernel

Stencil codes are a class of iterative methods found in many scientific and engineering applications. Here, the problem domain is iteratively updated by using the same computational pattern, called a *stencil*. We implemented a hybrid MPI+OpenMP 3D 7-point stencil code that simulates a heat equation. Our decomposition methodology tries to reduce the internode communication by dividing the domain along all dimensions, while we avoid splitting the process subdomain along the most strided dimensions for better cache performance. The basic communication method is to perform nonblocking send/receive operations at each iteration followed by `MPI_Waitall` to wait for all the requests. Common hybrid stencil codes typically require the `MPI_THREAD_FUNNELED` threading support, where the computation is done in parallel by a team of threads but only the main thread is driving the communication. In our implementation, all threads independently do computation and communication and synchronize only at the end of an iteration.

We conducted a strong-scaling experiment on 64 nodes, using all 8 threads per node, while increasing the problem size. The results in Figures VI.16a and VI.16b show that our methods improve performance for relatively small problems (<= 1 MB/core on the Nehalem machine and <= 128 KB on Tsubame 2.5 thin nodes). The reason is that, as demonstrated by our microbenchmarks, the runtime contention is more critical for relatively small messages; in addition, the computation takes more time than does the communication, as shown in Figure VI.16c. Arguably, the benefit of our methods is less pertinent for stencil applications on this typical platform, since bigger problems are often run in production. Nevertheless, given the increase in core counts and the trend



(a) 3D-stencil strong scaling on the Nehalem machine

(b) 3D-stencil strong scaling on Tsubame 2.5

(c) 3D-stencil execution breakdown

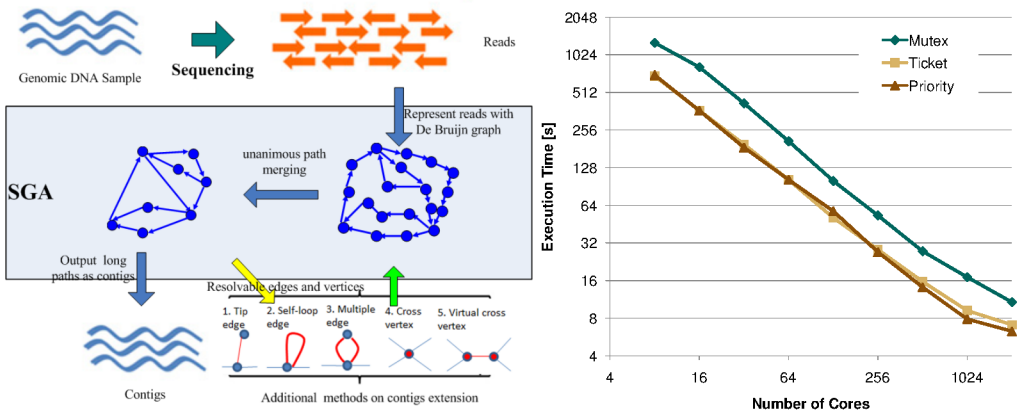
Figure VI.16: Strong-scaling comparison of all methods with the 3D 7-point stencil kernel

of reduced memory per core, we expect that our solutions will play a more important role when running stencil applications on platforms with less memory per core, such as current systems equipped with many-core accelerators and future systems. We note that the priority locking does not improve performance over the ticket lock method. The reason is that since threads have few requests (8 receive and send operations), the rate at which the critical section is entered from the main path is negligible compared with polling for communication in the progress loop. That is, most threads will fall back to low priority, which is equivalent to the ticket method. As a result, giving priority to the main path has negligible effect.

5.3 Genome Assembly Application

Genome assembly is an important process for many fields, such as biological research and virology. It refers to the process of reconstructing a long DNA sequence, such as the chromosome of an organism, from a set of *reads* (short DNA sequences) by aligning and merging them together. The reads originate from automated sequencing machines, which can generate billions of reads to be processed by assembly applications. As a result, these applications exploit high-performance computing systems and explore efficient parallel solutions in order to cope with the ever-increasing generated sequencing data.

We evaluated our methods using the SWAP-Assembler [57], an application that targets processing massive sequencing data on large-scale parallel architectures. It abstracts the genome assembly problem with a multistep bidirected graph and relies on a scalable framework, SWAP (Small World Asynchronous Parallel) [58], to perform computational



(a) Genome assembly process in SWAP-Assembler [57]

(b) SWAP-Assembler strong scaling

Figure VI.17: Description of the genome assembly process and performance of the SWAP-Assembler using all methods

work in parallel (Figure VI.17a). The SWAP framework is implemented on top of MPI to ensure interprocess communication. Each process spawns two threads, one for sending and another for receiving data from other processes using blocking `MPI_Send`/`MPI_Recv` operations. We performed a strong-scaling experiment with a synthetic sequence of 1 million reads, where each read contains 36 nucleotides. The results are shown in Figure VI.17b. For each data point we used four processes per node and two threads per process to utilize all cores. We observe an average 2x speedup independent of the core count. We note that this improvement in processing time did not incur any modification in the application or the underlying hardware. This fact is important for applications in production environments, since no additional investment is required to speed up the time to solution.

6 Discussion

In addition to the lower overhead of the ticket lock compared with the priority lock (fewer atomic operations), cases may arise where the ticket method will perform better because of fair arbitration. MPI runtimes are sensible to the number of queued requests because the associated internal data structures and algorithm complexity are proportional. Since the priority lock gives high priority to feeding the runtime with requests, ticket may reduce the rate of issuing requests and thus their associated overhead. However, there exists another related issue that affects the cache performance of the MPI runtime and applications. The order in which threads acquire resources can affect the data locality of MPI internal structures, such as shared queues, and thus runtime performance. Similarity,

it can also affect the computation part of an application. Assuming that the aggregated threads working sets cannot all fit in the last level of cache, critical section arbitration might impact the amount of data reuse and cache line evictions. These intricacies require further analysis and experimentation.

The idea of using a socket-aware high-priority method that prioritizes threads on the main path and the same socket before moving to another socket seems attractive for reducing intersocket synchronization. However, this approach may lead to starvation. For instance, if the user issues nonblocking operations and waits for them by polling with `MPI_Test`, threads on the same socket will monopolize the lock. We also did not consider a mutex-based priority arbitration. Changing the kernel scheduling is not effective, as discussed in Section 1. Using three mutex locks to establish a two-level hierarchy, similar to our priority lock, is also not effective because mutexes do not guarantee fairness within the same priority class and, worse, low-priority threads can monopolize the lock over the high-priority threads.

7 Related Work

In a different context, many works have addressed similar issues, such as starvation, preemption, and deadlocks, in the context of multitasking systems [76]. In addition, establishing a priority between classes of tasks can be critical in some real-time systems. In our work the shared resources belong to the communication runtime, and the goal is to maintain fast progress on communication. Although the context is different, we demonstrated that some existing heuristics in this field (FCFS and priority with aging) can be beneficial to multithreaded communication.

8 Conclusion

We addressed in this chapter the MPI runtime inefficiency with multithreaded communication. Our analysis demonstrated that one of the major drawbacks is unfair thread arbitration inside the MPI runtime caused by using a mutex-based critical section. We then addressed this unfair arbitration by ensuring FCFS access to the critical section. In addition, we proposed another method that biases the scheduling to favor threads with higher probability of making progress. Our results with various benchmarks and applications showed large improvements.

Despite those benefits, the performance difference between our methods and a pure-MPI setting with a simple throughput benchmark suggests that room for improvement remains. We are currently investigating even more flexible methods that target reducing

wasted time by the threads inside the MPI runtime. Examples include selective thread wake-up triggered by events such as message arrival. We also are considering combining fine-grained critical sections with custom lock arbitrations, which should be beneficial if contention occurs within some of the critical sections. As mentioned before, more scalable locks, such as NUMA aware locks, can be combined with a better arbitration to achieve better results on heavy NUMA systems.

Chapter VII

Conclusion and Future Directions

1 Summary

In this thesis, we addressed important challenges that high performance and high-end computing are facing from a threading model perspective. We first presented from an application perspective the two-dimensional trade-off between exposing parallelism and maintaining high degree of data locality to exploit thread-level parallelism. We showed that models that optimize only in one direction, using bulk-synchronous and fine-grained data-driven models, are not scalable approaches. We demonstrated that auto-tuning dynamically scheduled bulk-synchronous and data-driven implementations that exploits tiled computation patterns can be more effective. Second, we tackled the challenge of efficient interoperation between application threads and MPI runtimes. We first characterized the MPI+threads model at large scale and showed how it can alleviate certain scalability issues of an MPI-only model. We also exhibited, however, the contention that can take place when multiple threads access concurrently the MPI runtime. We followed with an analysis of this contention on commodity systems and showed that a mutex-based arbitration of the critical section incurs critical section mono plication and hinders the progress of the system. We showed how an FCFS arbitration establishes fairness and improves threads progress and the performance of many benchmarks and applications. In addition, by prioritizing threads in the main path of the runtime, we demonstrated further improvement.

2 Insights and Future Directions

In the following, we provide some directions that we judged worth considering in the context of threading models on highly parallel systems.

2.1 Improving Support for Task and Data-Parallelism

We showed numerous ways of implementing a given application in parallel with the current state-of-the-art of programming models and their runtime systems. Here we use the lessons learned previously in order to provide insight into extensions and improvements that would be important for future programming models to program share-memory machines.

At the Programming Model Level

The programming model constrains the application developer with a set of constructs that sometimes are not flexible enough to express efficient communication reducing algorithms while exposing abundant parallelism. An example that is recurrent is the high-level abstraction of the memory hierarchy, where many researchers claim that the memory hierarchy needs to be exposed to the programmer. However, the optimal way that the memory hierarchy should be exposed is still an open question. For instance, many programming models implement the notion of *locale*. That is, an abstract region that combines data and the processor that will operate on it. However, this approach is not ideal in many cases. A strict mapping of tasks to locals may show signs of load unbalance while a dynamic load-balancing will hinder the locality that the user wanted from the first place.

Many works targeted NUMA-node-aware type of locality optimizations. We argue that, since keeping data within the on-chip memory is more critical than avoiding inter-NUMA-node data movements, better abstractions that aim at this direction are required. For instance, the concept of *tilling* proved to be powerful to exploit both spacial and temporal locality. Thus, future programming models should incorporate *tilling abstractions* that allow easy-to-implement and elegant constructs combined with smart optimizations from the underlying runtime.

At the Runtime Level

Programming models that allow arbitrary task dependency expressions are equipped with runtime systems to track task dependencies and schedule ready tasks. Expressing data dependencies is ensured by the programmer by giving input and output data accessed by a given task. This same information, however, can be used for more than just dependency tracking. It can also serve at giving hints to the runtime to perform some locality aware task scheduling. One would imagine a runtime system that would cluster fine-grained tasks together into coarse-grained ones to optimize for temporal and spacial locality. However, the overhead of such runtime scheduler would likely be higher

than a simple random work-stealing algorithm, and thus would require careful design and implementation to minimize this overhead.

2.2 MPI+Threads

Given the importance of the hybrid MPI+Threads model and the persistent thread-safety overheads, more work is required to characterize and optimize the interoperation between threads and MPI. Here we list some directions that we consider worth investigating in the future.

Advanced Critical Section Arbitration

In our work we presented simple heuristics that proved to be more efficient than when using mutex. As our understanding of the influence of critical section arbitration on MPI runtimes is improving, better heuristics are feasible. A heuristic that we believe is promising is a *message-driven arbitration*. That is, message handlers would carry a thread identifier that would serve at selectively waking up that specific thread so that it can carry the work on that message such as free the corresponding request. I case the thread corresponding to the message handler is not inside the MPI runtime, or in case no communication progress is made, one would fall into a fair and scalable arbitration. Such scheme is more complex and a careful design and implementation is required.

Combining Fine-Grained Concurrency with Custom Arbitration

In our introduction to thread-safety in Section 1, we considered a critical-section granularity as orthogonal to how it is arbitrated. That is, regardless of the granularity, resource acquisition–related issues such as starvation may occur, and appropriate arbitration methods will be required. Thus, we believe that combining those approaches will have a synergistic effect on reducing the runtime contention. However, a cost-effectiveness study of both methods on the same testbed is still warranted. Such a study can guide the development process of a thread-safe library. A possible model would be to start with a global critical section, explore effective arbitration methods, reduce granularity if high contention persists, and repeat the process.

MPI+Threads with an M-to-N Model

We addressed the challenge of MPI critical sections arbitration from a kernel thread perspective. In other words, the MPI library views the threading model as a one-to-one mapping between application threads and kernel threads (Pthreads). It is possible,

however, for an MPI library to support other threading models such as the more generic M:N mapping. Such model can offer new opportunities to reduce runtime contention. For instance, application threads mapped to the same kernel thread can hand-off the critical section by a simple context switch without releasing the lock. This avoids the overhead of atomic operations and memory barriers that a traditional hand-off mechanism incurs. However, such model introduces new challenges, such as avoiding lock monopolization by the application threads running on the same kernel thread, and will likely require different arbitration policies than the one-to-one model.

Bibliography

[1] The Graph 500 list. http://www.graph500.org/.

[2] OSU microbenchmarks suite. http://mvapich.cse.ohio-state.edu/benchmarks.

[3] Papi. http://icl.cs.utk.edu/PAPI/.

[4] Vtune. http://software.intel.com/en-us/intel-vtune-amplifier-xe.

[5] *Introducing the Graph 500.* Cray User's Group (CUG), May 2010.

[6] Mpi: A message-passing interface standard version 3.0, September 2012.

[7] Intel®xeon®processor e5-2600 product family uncore performance monitoring guide, March 2012.

[8] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. SuperComputing '10, pages 1–11, Nov 2010.

[9] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Pipelining the fast multipole method over a runtime system, 2012.

[10] Abdelhalim Amer, Huiwei Lu, Yanjie Wei, Pavan Balaji, and Satoshi Matsuoka. Mpi+ threads: Runtime contention and remedies. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 239–248. ACM, 2015.

[11] Abdelhalim Amer, Naoya Maruyama, Miquel Pericàs, Kenjiro Taura, Rio Yokota, and Satoshi Matsuoka. Fork-join and data-driven execution models on multi-core architectures: Case study of the fmm. In *Supercomputing*, pages 255–266. 2013.

[12] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research:

A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[13] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, et al. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[14] Pavan Balaji, Darius Buntinas, D. Goodell, W. D. Gropp, and Rajeev Thakur. Fine-grained multithreading support for hybrid threaded MPI programming. *Int. J. High Perform. Comput.Appl.*, 24:49–57, Feb 2010.

[15] Josh Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449, December 1986.

[16] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 66. IEEE Computer Society Press, 2012.

[17] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. *Cilk: An efficient multithreaded runtime system*, volume 30. ACM, 1995.

[18] Holger Brunst and Andreas Knüpfer. Vampir. In *Encyclopedia of Parallel Computing*. Springer, October 2011.

[19] Franck Cappello and Daniel Etiemble. Mpi versus mpi+ openmp on the ibm sp for the nas benchmarks. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 12–12. IEEE, 2000.

[20] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. High performance locks for multi-level numa systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 215–226. ACM, 2015.

[21] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc. Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, april 2010.

[22] Aparna Chandramowlishwaran, Kamesh Madduri, and Richard Vuduc. Diagnosis, tuning, and redesign for multicore performance: A case study of the fast multipole

method. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society.

[23] J. Chhugani, N. Satish, Changkyu Kim, J. Sewall, and P. Dubey. Fast and efficient graph traversal algorithm for CPUs: Maximizing single-node efficiency. pages 378–389, May 2012.

[24] Martin J Chorley and David W Walker. Performance analysis of a hybrid mpi/openmp application on multi-core clusters. *Journal of Computational Science*, 1(3):168–174, 2010.

[25] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[26] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization and performance modeling of stencil computations on modern microprocessors, 2009.

[27] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48, New York, NY, USA, 2013. ACM.

[28] Erik D 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. Citeseer, 1997.

[29] David Dice, Virendra J Marathe, and Nir Shavit. Lock cohorting: a general technique for designing numa locks. In *ACM SIGPLAN Notices*, volume 47, pages 247–256. ACM, 2012.

[30] J. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju. Supporting the global arrays pgas model using mpi one-sided communication. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 739–750, May 2012.

[31] James Dinan, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev Thakur. Enabling mpi interoperability through flexible communication endpoints. In *Proceedings of the 20th European MPI Users' Group Meeting*, pages 13–18. ACM, 2013.

[32] James Dinan, Ryan E Grant, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev Thakur. Enabling communication concurrency through flexible mpi endpoints. *International Journal of High Performance Computing Applications*, 28(4):390–405, 2014.

[33] 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.

[34] Ulrich Drepper. Futexes are tricky. Dec 2011.

[35] Paul J. Drongowski. Basic performance measurements for amd athlontm 64, amd opterontm and amd phenomtm processors, 25 September 2008.

[36] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[37] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *AUUG Conference Proceedings*, page 85, 2002.

[38] Matteo Frigo and Steven G Johnson. Fftw: An adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384. IEEE, 1998.

[39] Alexandros V Gerbessiotis and Leslie G Valiant. Direct bulk-synchronous parallel algorithms. *Journal of parallel and distributed computing*, 22(2):251–267, 1994.

[40] David Goodell, Pavan Balaji, Darius Buntinas, Gabor Dozsa, William Gropp, Sameer Kumar, Bronis R. de Supinski, and Rajeev Thakur. Minimizing mpi resource contention in multithreaded multicore environments. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, CLUSTER '10, pages 1–8, Washington, DC, USA, 2010. IEEE Computer Society.

[41] Leslie Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems*, volume 52. MIT Press, 1988.

[42] Leslie Frederick Greengard. *The rapid evaluation of potential fields in particle systems*. PhD thesis, Yale University, New Haven, CT, USA, 1987. AAI8727216.

[43] William Gropp and Rajeev Thakur. Thread-safety in an mpi implementation: Requirements and analysis. *Parallel Comput.*, 33:595–604, Sep 2007.

[44] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[45] T. Hoefler, G. Bronevetsky, B. Barrett, B. R. de Supinski, and A. Lumsdaine. Efficient mpi support for advanced hybrid programming models. In *Recent Advances in the Message Passing Interface (EuroMPI'10)*, volume LNCS 6305, pages 50–61. Springer, Sep. 2010.

[46] Gabriele Jost, Haoqiang Jin, Dieter an Mey, and Ferhat F Hatay. Comparing the openmp, mpi, and hybrid programming paradigms on an smp cluster. In *Proceedings of EWOMP*, volume 3, page 2003, 2003.

[47] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS), 2010*, pages 1–12, April 2010.

[48] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The Vampir Performance Analysis Tool-Set. In Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing*, pages 139–155. Springer Berlin Heidelberg, 2008.

[49] Peter Kogge. Pim & memory: The need for a revolution in architecture. The Argonne Training Program on Extreme-Scale Computing (ATPESC), 2013.

[50] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In *Knowledge Discovery in Databases: PKDD 2005*, pages 133–145. Springer, 2005.

[51] Hatem Ltaief and Rio Yokota. Data-driven execution of fast multipole methods, 2012.

[52] Ewing Lusk and Anthony Chan. Early experiments with the OpenMP/MPI hybrid programming model. In *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*, IWOMP'08, pages 36–47, Berlin, Heidelberg, 2008. Springer-Verlag.

[53] Huiwei Lv, Guangming Tan, Mingyu Chen, and Ninghui Sun. Understanding parallelism in graph traversal on multi-core clusters. *Comput. Sci.*, 28(2-3):193–201, may 2013.

[54] John D McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture TCCA Newsletter*, pages 19–25, 1995.

[55] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[56] John M Mellor-Crummey and Michael L Scott. Synchronization without contention. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 269–278. ACM, 1991.

[57] Jintao Meng, Bingqiang Wang, Yanjie Wei, Shengzhong Feng, and Pavan Balaji. Swap-assembler: scalable and efficient genome assembly towards thousands of cores. *BMC Bioinformatics*, 15(Suppl 9):–2, 2014.

[58] Jintao Meng, Jianrui Yuan, Jiefeng Cheng, Yanjie Wei, and Shengzhong Feng. Small world asynchronous parallel model for genome assembly. In JamesJ. Park, Albert Zomaya, Sang-Soo Yeo, and Sartaj Sahni, editors, *Network and Parallel Computing*, volume 7513, chapter Lecture Notes in Computer Science, pages 145–155. Springer Berlin Heidelberg, 2012.

[59] Ingo Molnar. The native POSIX thread library for Linux. Technical report, 2003.

[60] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[61] Jun Nakashima and Kenjiro Taura. Massivethreads: A thread library for high productivity languages. In *Concurrent Objects and Beyond*, pages 222–238. Springer, 2014.

[62] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda. High performance remote memory access communication: The armci approach. *Int. J. High Perform. Comput. Appl.*, 20(2):233–253, may 2006.

[63] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Aprà. Advances, applications and performance of the global

arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2):203–231, may 2006.

[64] Akira Nukada and Satoshi Matsuoka. Auto-tuning 3-d fft library for cuda gpus. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 30. ACM, 2009.

[65] Stephen L. Olivier, Bronis R. De Supinski, Martin Schulz, and Jan F. Prins. Characterizing and mitigating work time inflation in task parallel programs. *Scientific Programming*, 21(3):123–136, 2013.

[66] Miquel Pericas, Abdelhalim Amer, Keisuke Fukuda, Naoya Maruyama, Rio Yokota, and Satoshi Matsuoka. Towards a dataflow fmm using the ompss programming model. *136th IPSJ Conference on High Performance Computing*, 2012.

[67] Chuck Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[68] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436, 2009.

[69] N. Satish, Changkyu Kim, J. Chhugani, and P. Dubey. Large-scale energy-efficient graph traversal: A path to efficient data-intensive supercomputing. pages 1–11, Nov 2012.

[70] Gaurav Saxena. Thread safety for hybrid programming in thread-as-rank model. *Master's thesis, The University of Edinburgh*, 2013.

[71] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *High Performance Computing for Computational Science–VECPAR 2010*, pages 1–25. Springer, 2011.

[72] Min Si, Antonio J. Peña, Pavan Balaji, Masamichi Takagi, and Yutaka Ishikawa. MT-MPI: Multithreaded MPI for many-core environments. *ACM International Conference on Supercomputing (ICS 2014)*, 2014.

[73] Srinivas Sridharan, James Dinan, and Dhiraj D Kalamkar. Enabling efficient multithreaded mpi communication through a library-based implementation of mpi endpoints. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 487–498. IEEE Press, 2014.

[74] T. Suzumura, K. Ueno, H. Sato, K. Fujisawa, and S. Matsuoka. Performance characteristics of graph500 on large-scale distributed environment. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 149–158, Nov 2011.

[75] Nathan R Tallent and John M Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *ACM Sigplan Notices*, volume 44, pages 229–240. ACM, 2009.

[76] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[77] Hong Tang and Tao Yang. Optimizing threaded mpi execution on smp clusters. In *Proceedings of the 15th international conference on Supercomputing*, pages 381–392. ACM, 2001.

[78] Sagnak Tasirlar and Vivek Sarkar. Data-driven tasks and their implementation. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 652–661, 2011.

[79] Kenjiro Taura, Rio Yokota, and Naoya Maruyama. A task parallelism meets fast multipole methods. In *Proceedings of SCALA'12*, November 2012.

[80] Rajeev Thakur and William Gropp. Test suite for evaluating performance of multithreaded mpi communication. *Parallel Computing*, 35(12):608–617, 2009.

[81] Koji Ueno and Toyotaro Suzumura. Highly scalable graph search for the graph500 benchmark. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 149–160, New York, NY, USA, 2012. ACM.

[82] Jeffrey D. Ullman. NP-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975.

[83] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, and W. A. de Jong. Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.

[84] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.

[85] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 38:1–38:12, New York, NY, USA, 2007. ACM.

[86] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[87] Samuel Webb Williams. *Auto-tuning performance on multicore computers*. ProQuest, 2008.

[88] Xingfu Wu and Valerie Taylor. Performance characteristics of hybrid mpi/openmp implementations of nas parallel benchmarks sp and bt on large-scale multicore supercomputers. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):56–62, 2011.

[89] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *Languages and Compilers for Parallel Computing*, pages 172–187. Springer, 2010.

[90] Lexing Ying, G. Biros, D. Zorin, and H. Langston. A new parallel kernel-independent fast multipole method. In *Supercomputing, 2003 ACM/IEEE Conference*, page 14, nov. 2003.

[91] Lexing Ying, George Biros, and Denis Zorin. A kernel-independent adaptive fast multipole algorithm in two and three dimensions, 2003.