Embracing Explicit Communication in Work-Stealing Runtime Systems

Von der Universität Bayreuth
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von

Andreas Prell

aus Kronach

1. Gutachter:	Prof. Dr. Thomas Rauber
2. Gutachter:	Prof. Dr. Claudia Fohry

Tag der Einreichung: 14. Juli 2016
Tag des Kolloquiums: 20. September 2016

Abstract

Parallel computers are commonplace. The trend of increasing the number of processor cores highlights the importance of parallel computing: a single-threaded program uses a fraction of a modern processor's resources and potential, and that fraction will only decrease over the coming processor generations.

Existing abstractions for writing parallel programs, such as threads and mutual exclusion locks, are difficult to understand, use, and reason about, making them a poor choice for mainstream parallel programming. Higher-level abstractions aim to achieve a more favorable division of labor between programmers and compilers/runtime systems, with programmers expressing and exposing parallelism and compilers/runtime systems managing parallel execution.

A popular and effective abstraction is that of a *task*, a piece of work, usually a function or a closure, that is safe to execute in parallel with other tasks. Scheduling decisions, including the mapping of tasks to threads, are made by the runtime system and are not imposed on the programmer.

Tasks are well-suited to express fine-grained parallelism, but whether fine-grained parallelism brings performance gains depends on the runtime system and its implementation. State-of-the-art runtime systems employ the scheduling and load balancing technique of work stealing, which is known to be efficient, both in theory and practice. In work stealing, idle workers, called thieves, request tasks from busy workers, called victims, thereby balancing the load. Most implementations of work stealing take advantage of shared memory by letting thieves "steal" tasks from the double-ended queues (deques) of their victims.

Modern multiprocessors feature increasingly complex architectures that make it challenging to implement efficient yet flexible work-stealing schedulers. Future manycore processors may have limited support for shared memory, or may rely on message passing for scalable inter-core communication, such as Intel's SCC research processor, a recent example of a "cluster-on-a-chip".

This thesis aims to put work stealing based on message passing on a better, more practical foundation, developing techniques to rival the performance of concurrent deque-based implementations, while remaining more flexible. Work stealing based on message passing has been studied before, notably in the context of distributed systems, where MPI still dominates. We present a work-stealing scheduler in which workers communicate with each other through channels, a lightweight message passing abstraction that goes back to Hoare's Communicating Sequential Processes (CSP). Channels feature prominently in modern programming languages such as Go and Rust, which advocate messages to communicate, synchronize, and share state between threads. The advantage of using channels, as opposed to using low-level synchronization primitives, is that channels decouple the scheduler from processor-specific features, thereby increasing its flexibility. Large parts of this thesis are dedicated to making channel-based work stealing perform well on modern shared-memory multiprocessors.

We describe an implementation in which workers exchange asynchronous steal requests and tasks by passing messages over channels. Termination is detected as a consequence of forwarding steal requests instead of requiring additional control messages to be passed between workers. Dependencies between tasks, most importantly, between parent and child tasks, are expressed with futures, which can be implemented efficiently in terms of channels.

Private task queues are more flexible than concurrent ones. We show a simple extension that provides support for adaptive stealing—the ability to switch the stealing strategy at runtime. Fine-grained parallelism requires not only efficient work stealing, but also granularity control to overcome the overhead of task creation and scheduling. Similar tasks, such as iterations of a parallel loop, can be combined into a single task ready to split whenever parallelism is needed. We extend previous work on lazy splitting, integrate it with channel-based work stealing, and demonstrate performance comparable to dedicated loop schedulers in OpenMP. Finally, we provide experimental evidence that channel-based work stealing performs on par with runtime systems based on concurrent deques.

Zusammenfassung

Parallelrechner auf Basis von Mehrkernprozessoren sind heutzutage allgegenwärtig. Da anzunehmen ist, dass die Anzahl der Prozessorkerne, die auf einem Chip Platz finden, weiter steigen wird, besteht Handlungsbedarf. Ohne Parallelverarbeitung bleibt das Potenzial eines modernen Rechners zunehmend ungenutzt. Aus diesem Grund gewinnen Techniken der parallelen Programmierung mehr und mehr an Relevanz.

Die klassische Thread-Programmierung gilt als zu diffizil, um ein geeignetes Programmiermodell zu bieten, das auch von Nicht-Experten effektiv eingesetzt werden kann. Eine vielversprechende Alternative ist die Benutzung von Tasks anstelle von Threads. Ein Task bezeichnet eine beliebige Berechnung innerhalb eines Programms, die unabhängig von anderen Berechnungen und damit parallel ausgeführt werden kann. Der Programmierer hat die Aufgabe, Tasks zu spezifizieren, während das Laufzeitsystem für deren Ausführung sorgt. Dabei übernimmt das Laufzeitsystem viele kritische Funktionen einschließlich der Verwaltung von Threads, der Zuweisung von Tasks an Threads und der Lastverteilung.

Tasks sind leichtgewichtiger als Threads und somit einfach zu erzeugen, selbst für relativ feingranulare Aufgaben. Task-parallele Programme generieren in der Regel eine Vielzahl von Tasks, mit dem Ziel, diese möglichst gleichmäßig an die ausführenden Worker Threads zu verteilen. Wie feingranular die Tasks dabei sein dürfen, hängt von der Effizienz des Laufzeitsystems ab.

Von besonderer Bedeutung ist das Work Stealing, eine Scheduling-Technik, bei der Worker Threads, die keine Tasks mehr haben, anderen Worker Threads durch Stehlen Arbeit abnehmen, wodurch eine dynamische Lastverteilung erzielt wird. Üblicherweise besitzt jeder Worker Thread eine eigene Queue-Datenstruktur (Deque), in die Tasks abgelegt und zur Ausführung entnommen werden, und die von anderen Worker Threads zugegriffen werden kann, um Stehlen zu ermöglichen. Solche Implementierungen sind oft aus Effizienzgründen auf eine bestimmte Hardware-Architektur zugeschnitten. Da die Zukunft Cluster-ähnlichen Vielkernprozessoren gehören dürfte, ist davon auszugehen, dass Work Stealing Scheduler an diesen Umstand angepasst werden müssen. Eine zu starke Plattformabhängigkeit, was zum Beispiel das Vorhandensein bestimmter Synchronisationsoperationen betrifft, kann auf lange Sicht eine Portierung erschweren.

Die vorliegende Arbeit verfolgt das Ziel, eine effiziente und gleichzeitig flexible Alternative zum klassischen Work Stealing im gemeinsamen Adressraum zu entwickeln. Zu diesem Zweck wird ein Laufzeitsystem entworfen, in dem Worker Threads ausschließlich über Channels miteinander kommunizieren. Direktes Stehlen ist nicht mehr möglich: Worker Threads senden Steal Requests, die mit

Tasks beantwortet oder abgelehnt werden.

Channels bieten ein einfaches Interface für den Nachrichtenaustausch zwischen Threads, das von der Hardware-Architektur abstrahiert und effizient implementiert werden kann. Gepufferte Channels ermöglichen asynchrone Kommunikation, so dass Worker Threads in der Lage sind, Steal Requests untereinander auszutauschen, ohne Antworten abwarten zu müssen. Das Senden eines Steal Requests ist dadurch vergleichbar mit einem asynchronen Aufruf, der eventuell einen Task über einen separaten Channel zurückliefert. Die Terminierung einer task-parallelen Berechnung kann aus Steal Requests abgeleitet werden und erfordert kein verteiltes Protokoll mit zusätzlichem Nachrichtenaustausch. Taskabhängigkeiten, zum Beispiel zwischen Eltern- und Kindtasks, werden durch Futures ausgedrückt, welche eng mit Channels korrespondieren.

Worker Threads verwalten Tasks in privaten Deques. Dies vereinfacht die Realisierung flexibler Strategien wie zum Beispiel adaptives Stehlen, bei dem jeder Worker Thread selbst entscheidet, wieviele Tasks gestohlen werden sollen.

Im Mittelpunkt der Arbeit steht die effiziente Ausführung feingranularer Tasks. Um den Overhead der Taskverwaltung zu reduzieren, ist es möglich, ähnliche Tasks, insbesondere Iterationen paralleler Schleifen, so zusammenzufassen, dass weitere Tasks nur nach Bedarf erzeugt werden. Überschüssige Tasks werden automatisch sequentialisiert und verursachen keinen Overhead. Das vorgestellte Laufzeitsystem implementiert und erweitert das sogenannte Lazy Splitting, welches ermöglicht, parallele Schleifen ähnlich effizient auszuführen wie mit OpenMP, ohne auf die Unterstützung eines Loop Schedulers angewiesen zu sein.

Mithilfe der entwickelten Techniken lässt sich trotz expliziter Kommunikation gute Performance erzielen. Bei einem Vergleich auf drei unterschiedlichen Systemen landet das vorgestellte Laufzeitsystem vor Cilk Plus und Intel OpenMP und nur knapp hinter einer Variante mit Chase-Lev Deques.

Danksagung

An erster Stelle richte ich meinen Dank an Prof. Dr. Thomas Rauber für seine Anleitung und seine Unterstützung, die mir geholfen haben, mich an eine Promotion zu wagen. Professor Raubers Vorlesungen weckten damals mein Interesse für Rechnerarchitektur und Parallelverarbeitung. Umso mehr weiß ich es zu schätzen, dass ich nach dem Studium die Gelegenheit bekam, noch tiefer in diese Gebiete einsteigen zu dürfen.

Gleichermaßen bedanke ich mich bei meinen jetzigen und ehemaligen Kolleginnen und Kollegen für die gute Zusammenarbeit und das gute Miteinander am Lehrstuhl, das eine so wichtige Rolle spielt.

Danke an Simon Melzner und Monika Glaser, ohne deren Hilfe so manches technische und organisatorische Problem schwierig zu bewältigen gewesen wäre.

Nicht zuletzt habe ich vieles meinen Eltern zu verdanken, die mich stets unterstützen und mir nie einen Stein in den Weg legten. In den letzten Jahren darf außerdem ein Name nicht fehlen: Miwako, itsumo arigatō.

Contents

List of Figures

List of Tables

List of Code Listings

1 Introduction

For almost a decade, since the introduction of the first commercial multicore processors, parallelism has been the primary method of improving processor performance [38]. Today, the computing landscape is dominated by multicores. Even mobile processors as found in laptops, tablets, and smartphones have two or more cores. And yet, despite the ubiquity of parallel computers, writing programs that take advantage of multiple cores remains challenging. This is unfortunate, because the performance gap between simple, sequential code and well-parallelized code has been growing over the last processor generations and will likely continue to grow [220].

1.1 The Changing Microprocessor Landscape

While the number of cores per chip has not turned out to double every two years, it is slowly but steadily increasing [190]. Current top-of-the-line Intel Xeon server processors have up to 24 cores, which, coupled with simultaneous multithreading (SMT), can execute instructions from 48 independent hardware threads at the same time (24 cores $\times$ 2 threads per core) [13]. Modern coprocessors including GPGPUs integrate many small cores, not ideal for running sequential programs, but, taken together, capable of accelerating highly parallel workloads. Intel's first commercial manycore processor, the Xeon Phi, has between 57 and 61 Pentium-class cores [12] and requires at least two threads per core to fulfill its true potential (see [126], Chapter 8, pp. 249–250).

Researchers are experimenting with hundreds to thousands of cores on a single chip [54, 127, 52]. It seems likely that the trend of increasing parallelism will continue, although at some point a fraction of the cores may have to be powered off and turned into “dark silicon” [84]. To an increasing extent, processor architectures embrace heterogeneity to overcome the inefficiency of general-purpose, power-constrained hardware [135, 123, 108, 237]. Unfortunately, a growing diversity of microprocessors will only add to the challenge of parallel programming.

Whether hardware-managed cache coherence is here to stay, or whether it will be gradually abandoned on the road to more and more cores, remains to be seen. Some researchers argue that the communication overhead associated with cache coherence protocols will grow to the point where the amount of traffic between caches saturates the interconnect and practically limits the number of cores that can be put on a chip [42, 124, 134]. If this prediction turns out to be true, message-passing chips with non-cache-coherent memories might start to replace their cache-coherent counterparts, so that, if needed, coherence might have to be enforced by software. Others remain optimistic that directory-based protocols can scale to large numbers of cores [161].

What is undisputed, however, is the importance of scalable on-chip interconnects [164, 55, 189, 192]. Early multicore processors were based on the idea of modular tiles containing processor cores and communication switches with the goal of scaling the number of cores as transistor budgets increase [252, 153, 180, 238]. A more recent example of a tiled architecture is the Single-Chip Cloud Computer (SCC), the second of Intel's Terascale Research processors [110], which connected 24 dual-core tiles in a two-dimensional mesh network [124]. The majority of the system memory was mapped as private, turning the SCC into a "cluster-on-a-chip" with a message passing programming model. A small amount of on-chip memory—16 KB per tile—was set aside to accelerate communication between cores. Programming the SCC had a lot in common with programming a distributed system, starting, of course, with the preference for sending and receiving messages over modifying shared state.

1.2 The Growing Importance of Parallel Programming

The end of frequency scaling has led to the realization that parallelism is essential for continued performance improvements. Whether shared memory or message passing, parallel programming techniques are more important than ever for a simple reason [233]: single-threaded applications use only one core, a fraction of a processor's resources and potential. On top of that, single-threaded performance no longer improves at the same rate as it did in the past. In fact, it may not improve at all if clock rates are lowered to reduce power consumption. If performance is a concern, applications must be written to use multiple, mostly independent threads of execution.

"Threads and locks" are sometimes described as the "assembly language" of shared-memory concurrency [36, 128]: a low-level programming model that reflects how multicore processors operate. The fundamental problem with this level of abstraction is that it is extremely difficult to reason about all possible thread interleavings for anything but the most trivial programs, so it becomes hard to write code that is free of deadlocks, data races[1], and race conditions in general [146, 30, 262]. Even so called "benign" data races [182] suffice to make program behavior impossible to predict, which is to say, all bets are off in the presence of a data race [51].

The challenge is to find ways to lower the barrier to parallel programming while producing correct and efficient programs for multi- and manycore platforms [38]. Achieving this goal requires raising the level of abstraction above "threads and locks". While threads remain important, especially in I/O-intensive applications that involve blocking, it is often easier to think in terms of logical tasks rather than directly in terms of threads. Task-centric approaches are a promising way to deal with parallelism because they offload the burden of thread management, task creation, task scheduling, and load balancing to the runtime system/library, instead of imposing it on the programmer. Delegating these responsibilities to a library frees the programmer from concerns about multithreading—task-parallel programs rest on top of thread pools—and from the need to devise ad-hoc solutions for task scheduling and load balancing. Besides these benefits, tasks can be used where threads may not be profitable, for example in programs with many independent but short-lived computations.

Task-based abstractions are seeing widespread adoption. Java [193], C# [62], and C++11 [257] provide library support for asynchronous computations in the form of tasks, encouraging programmers to prefer tasks to threads when possible. Scott Meyers, for example, argues that the higher level of abstraction that tasks embody "frees you from the details of thread management", such as dealing with oversubscription[2] and load balancing [167]. Unlike threads, tasks "provide a natural way to examine the results of asynchronously executed functions".

The parallel programming languages X10 [65, 217], Chapel [1, 69], and Habanero Java [64] have been designed from the beginning to support tasks. X10 and Chapel were funded by DARPA's High Productivity Computing Systems (HPCS) program and aim to improve the programmability of large-scale machines by providing partitioned global address space (PGAS) abstractions [27] on top of distributed memory [263, 216]. Habanero Java, which is based on an earlier version of X10, consists of a set of parallel extensions to the Java language, compatible with recent versions of the Java virtual machine. All three languages share the approach of expressing parallelism in terms of tasks rather than threads. In Chapel, for instance, all parallel constructs, including data-parallel `forall` loops, are implemented on top of tasks.

Task-based programs rely on efficient runtime support. This is a point worth em-

[1]Concurrent but unsynchronized access to mutable data.

[2]Having more runnable threads than available processors increases the scheduling overhead.

phasizing: a task abstraction is only as good as its implementation. Runtime systems must be able to handle large numbers of fine-grained tasks without much overhead; otherwise, task-based programs may fail to achieve the desired performance, with little hope of running efficiently on more and more cores in the future.

State-of-the-art runtime systems employ a scheduling technique called work stealing as a means of load balancing: idle workers become thieves that “steal” tasks from busy workers, thereby balancing work as needed. Work stealing is often based on concurrent data structures, such as double-ended queues (deques), which grant thieves access to the tasks of their victims. Comparatively few implementations are based on message passing, despite the benefits of keeping tasks in private data structures that rule out concurrent access [28]. The goal of this dissertation is to strengthen the case for runtime systems in which worker threads communicate via messages rather than shared memory.

1.3 Motivation

Our previous work [120, 121, 200] has led us to the conclusion that work-stealing schedulers are difficult to port to architectures with unusual characteristics, such as limited support for shared memory or lack of universal synchronization primitives [113]. Both Cell [123] and SCC [165] processors supported shared memory, but their architectures made it impossible to implement concurrent data structures without overhead such as issuing multiple DMA transfers to update a value (on Cell) or invalidating certain cache lines to avoid reading stale data (on SCC). The SCC, for example, provided only a small number of test-and-set registers to compensate for the lack of atomic operations, requiring a combination of shared state and message passing to reduce contention [253]. In addition, the small size of the on-die message-passing buffers made it impractical to keep many tasks close to cores, where they could be accessed efficiently [200].

As scaling requires less sharing and more distribution, it seems reasonable to assume that manycore architectures will draw inspiration from clusters. We argue that, in light of the importance of scalable inter-core communication, parallel runtime systems will benefit from adopting message passing, in terms of portability and performance. To facilitate the transition, we propose a work-stealing runtime system in which workers communicate exclusively over channels. (We will often shorten the term worker thread to just worker.) Such a runtime system requires workers to send “steal requests” in order to receive tasks, which involves explicit cooperation between thieves and victims: thieves initiate steals, but depend on victims to send tasks. Work-stealing deques, on the other hand, assume that thieves and victims cooperate implicitly by following the same synchronization protocol.

Channels are well-known building blocks for concurrent systems: they permit threads to communicate and synchronize execution by exchanging messages through buffered or unbuffered message queues, without dictating a specific implementation [224]. As a simple message passing abstraction, channels can be used in programs that are intended to run on a wide variety of platforms, including those where MPI, for example, would incur too much overhead. How channels are implemented depends on the platform: shared-memory channels are often concurrent FIFO queues; distributed-memory channels are built on top of lower-level messaging primitives. An implementation of channels for the SCC would use the processor's message-passing features [195, 201] or leverage the native communication library [165]. Using channels for communication, or message passing in general, has another practical advantage: concurrent deques become redundant and can be replaced by private data structures, including lists and trees. Channels that need not support an arbitrary number of senders or receivers are amenable to optimization [206]. Additionally, by limiting the number of messages that workers are allowed to send, channels are strictly bounded, and sending can be guaranteed to always succeed without blocking a worker.

This dissertation explores a work-stealing runtime system in which workers communicate by exchanging messages over channels. In particular, we aim to

- make all inter-worker communication explicit by sending and receiving messages instead of modifying shared state to improve the architectural flexibility of work stealing,
- demonstrate comparable or better performance to existing runtime systems based on concurrent deques. This requires that channel communication does not affect the runtime system's ability to exploit fine-grained parallelism.

1.4 Contributions

While work stealing based on message passing is not a new idea, we present, to the best of our knowledge, the first scheduler that uses channels with the goal of decoupling task scheduling and load balancing from the choice of low-level communication. Large parts of this dissertation are dedicated to making this scheduler perform well under stressful workloads.

- We introduce a work-stealing scheduler in which n workers communicate through $2n$ channels. Every worker has two channels for receiving messages from other workers:

one channel for steal requests (using many-to-one communication) and one channel for tasks (using one-to-one communication). All channels have bounded capacity and limited concurrency, which simplifies their implementation.

- Our work-stealing scheduler handles steal requests differently than other schedulers: when a steal fails, the request is not returned to the thief, but forwarded to another victim, resulting in an attempt to steal on behalf of the thief. This reduces the number of messages and allows a worker to initiate a steal and continue working while the steal is carried out by coworkers.

- We highlight the importance of stealing multiple tasks and present a shared-memory implementation of steal-half—stealing half of a victim's tasks—without increasing the number of messages among workers. Using a simple heuristic that allows workers to choose and switch between steal-one and steal-half at runtime, we are able to combine the best of both strategies in order to achieve robust performance beyond tree-structured computations. This is a good example of how private deques afford the flexibility to implement new work-stealing strategies, without having to resort to customized data structures [28].

- We describe a new algorithm for termination detection that leverages asynchronous steal requests instead of requiring separate control messages. We show how this algorithm can be turned into a task barrier with little additional communication overhead. Tasks may have to wait for the results of other tasks. Such data dependencies are best expressed with futures. We describe a channel-based implementation of futures for nested parallelism, which achieves comparable performance to Cilk Plus.

- Fine-grained parallelism on the order of a few thousand CPU cycles can overwhelm a runtime system with the sheer number of tasks to create, schedule, and distribute. The ideal runtime system guarantees load balance without creating more tasks than necessary by increasing the granularity of tasks to a degree that permits efficient scheduling. Tzannes et al. proposed Lazy Binary Splitting (LBS) to defer the creation of tasks until workers are assumed to benefit from additional parallelism [245]. We describe splittable tasks—bundles of similar but independent tasks, such as, perhaps most importantly, iterations of a parallel loop—and evaluate different splitting strategies based on LBS in the context of concurrent deques and in the context of private deques. We find that our implementations come within 2.3% of the performance of loop scheduling in OpenMP (averaged over all benchmarks on a 48-core system) without the need to choose a chunk size, thus combining good performance with ease of programming.

- The success of channel-based work stealing depends on the efficiency of its implementation. We demonstrate competitive performance to traditional work-stealing schedulers on a set of task-parallel benchmarks and workloads using 24, 48, and 240 threads. Channel-based work stealing is on average faster than Cilk Plus and Intel OpenMP and only slightly slower than using Chase-Lev deques. These results make us confident that channels are useful building blocks for constructing work-stealing runtime systems.

1.5 Context

Channel-based work stealing grew out of the difficulties that we encountered in porting task-based runtime systems to different processor architectures. We started to embrace the idea of using explicit communication when we experimented with task-parallel programming on the SCC processor, whose lack of cache coherence and universal synchronization primitives proved challenging for shared-memory concurrency. We implemented a number of message-based schedulers, some of them still sharing deques, others using mailboxes instead of channels. After the MARC program had ended and our access to the SCC had expired in late 2013 [14], we went on to pursue channel-based communication on more conventional multiprocessors to be able to draw performance comparisons with popular work-stealing schedulers such as Cilk Plus and Intel's OpenMP runtime library. The scheduler that we describe in this thesis has evolved considerably from our early prototypes, which is why we omit any preliminary experiments on the SCC.

In its current state, our implementation is likely not as scalable as schedulers that target large-scale systems. The programming language X10, for example, distinguishes between intra-node and inter-node load balancing [100, 218, 265]. While we focus on intra-node load balancing, channel-based work stealing is flexible enough to cross node boundaries. That said, scaling out to multiple nodes will be easier if workers are grouped into partitions or places, which enable hierarchical work stealing and termination detection. For the purpose of this thesis, we can think of our scheduler as operating within a single, implicitly defined partition.

1.6 Outline

The main text is structured as follows:

Chapter 2 describes the notion of tasks in parallel programming, and explains why tasks are an effective abstraction on top of "threads and locks". With a task abstraction comes the need for a runtime system that hides lower-level details, including thread management, task scheduling, and load balancing. We look at task pools, typical task pool implementations, and the scheduling technique of work stealing.

Chapter 3 describes a work-stealing scheduler that employs private task queues and shared channels for communication between worker threads. Channels provide a message passing abstraction that allows the scheduler to operate on any system that is capable of supporting message queues.

Chapter 4 deals with constructs for termination detection in task-parallel computations: a task barrier to wait for the completion of all tasks and futures to support tree-structured computations including strict fork/join parallelism in the style of Cilk. Both constructs are based on channels.

Chapter 5 focuses on fine-grained parallelism. We introduce a heuristic for switching stealing strategies at runtime and propose extensions to the lazy scheduling of splittable tasks that achieve comparable performance to dedicated loop schedulers.

Chapter 6 compares the performance of channel-based work stealing with three work-stealing schedulers that use concurrent deques, both lock based and lock free, on a set of task-parallel benchmarks and workloads, demonstrating that channel communication does not prevent efficient scheduling of fine-grained parallelism.

Chapter 7 concludes by summarizing our findings and proposing ideas for future work.

2 | Technical Background

This chapter provides the necessary background on task parallelism, task-parallel programming, and runtime systems based on work-stealing scheduling.

Recent years have witnessed the growing importance of parallel computing. Section 2.1 draws an important distinction, that between concurrency and parallelism. Tasks make it easier to express parallelism, without giving concurrency guarantees. Sections 2.2–2.4 deal with threads and tasks, the benefits of programming with tasks compared to programming with threads, and the task model we are going to use, which offers portable abstractions for writing task-parallel programs.

The supporting runtime system is responsible for mapping tasks to threads. Section 2.5 contrasts static with dynamic scheduling. Section 2.6 describes the data structures behind dynamic schedulers —task pools—whose implementations can be centralized or distributed. Central task pools limit the scalability of dynamic schedulers. Distributed task pools solve this scalability problem, but add complexity in the form of load balancing. Section 2.7 elaborates on load balancing techniques, primarily on work stealing, and summarizes the pioneering results of Cilk that continue to influence the design and implementation of task schedulers [21]. Section 2.8 concludes with a list of task-parallel benchmarks and a few words about performance.

2.1 Concurrency and Parallelism

Due to the proliferation of microprocessors with increasing numbers of cores, concurrency and parallelism are becoming more and more important, as is the search for better programming abstractions than “threads and locks” [233]. While threads have long been used as building blocks for concurrent and parallel systems, higher-level abstractions tend to be designed with either concurrency or parallelism in mind [133].

Concurrency and parallelism are related but distinct concepts (see, for example, the introductory chapters in [156], [244], and [60], or refer to [213] for a thorough discussion of concurrency as used in different programming paradigms). In practice, however, the distinction is often obscured by a tendency to view both concurrency and parallelism as a means to improve performance, despite the fact that concurrency is a way to structure programs and not necessarily a recipe for parallel speedup [155, 109, 197].

Concurrency refers to multiple activities or threads of execution that overlap in duration [212]. Consider two threads T_1 and T_2. If one of the two threads, say T_1, completes before the other thread, T_2, starts running, T_1 and T_2 execute in sequence without interleaving. If T_2 starts running before T_1 completes, T_1 and T_2 happen logically at the same time; both threads have started and neither has completed [221]. We say T_1 and T_2 happen concurrently. It is left to the implementation whether T_1 and T_2 happen physically at the same time, that is, in parallel.

Parallelism results from simultaneous execution of two or more independent computations. By contrast, concurrency describes the structure of systems, programs, and algorithms in terms of threads and their interactions through memory. In that sense, concurrency *facilitates* parallelism: a concurrent program is easily turned into a parallel program by executing two or more threads simultaneously, for example by binding threads to different cores of a multicore processor. When forced to run on a single core, a program can be concurrent without being parallel.

Multiple threads are often a prerequisite for parallel execution, but parallelism is not tied to threads. At the machine level, independent instructions may execute in parallel (instruction-level parallelism), and SIMD instructions operate on multiple data elements packed into vectors (data parallelism). Because concurrency can be seen as dealing with more than one thing at the same time, we might think of parallelism as an instance of concurrency [56, 224]; programs must exhibit concurrency at some level of abstraction to make use of parallelism. For this reason, concurrency is usually considered to be a more general concept than parallelism.

Modern systems based on multicore processors benefit from both data and task parallelism. Data parallelism can be considered a subset of task parallelism [36]. It is possible to express a data-parallel computation as a task-parallel computation in which tasks are set up to perform the same operations on different elements of the data. Task and data parallelism are not mutually exclusive. Consider for example a blocked matrix multiplication that creates a task per matrix block and uses vector operations to speed up block-wise multiplications.

2.2 Tasks and Threads

Multithreaded programming has received a great deal of attention, but remains regarded as challenging, perhaps too challenging to make parallel programming accessible to a wide range of programmers. Higher-level abstractions than “threads and locks” are needed to reduce complexity and enable programmers to be more productive.

When we talk about multithreaded programming, we refer to the use of multiple, preemptively scheduled native (kernel) threads that share a common address space[1]. Multithreaded programming is *thread-centric*. Programmers are required to think in terms of threads—independent sequences of instructions—and how these threads may work together to achieve their purpose. Threads provide control over which computations are carried out in parallel, but at the cost of introducing complexity that must be dealt with and pitfalls that must be avoided [146, 166].

According to Leung [150], a good programming model (1) is less error prone than using threads directly, (2) makes it easy to identify independent computations, and (3) runs on current and future parallel hardware with increasing numbers of cores. The most promising approach is to raise the level of abstraction and make threads an implementation detail hidden from the programmer. Programs that utilize multiple threads in a way that is transparent to the programmer are implicitly multithreaded. Such programs are composed of *tasks*.

Since task is a very general term, we start with a simple definition: in the context of parallel computing, a task is a sequence of instructions that may be executed in parallel with other tasks (see [163], Section 2.4, page 16). In general, tasks denote pieces of code, usually functions or function objects, and all the arguments needed for execution. Tasks are *potentially* parallel (see [166], Section 2.3, page 44): a task is an opportunity for parallel execution, a hint to the runtime system that some computation can be done in parallel. Key to the idea of using tasks is to identify enough such opportunities and let the runtime system decide how to distribute the work. Intel’s developer documentation puts it this way [10]:

> *Design your programs to try to create many more tasks than there are threads, and let the task scheduler choose the mapping from tasks to threads.*

A task-parallel program with sufficient potential for parallel execution can achieve portable performance; it can run efficiently on different systems with different numbers

[1]Another form of multithreading is based on cooperatively scheduled threads, which are typically implemented in user space. An example library is GNU Portable Threads [83].

```go
func recurse(depth int) int {
    if depth < 2 {
        return compute()
    }

    x := make(chan int, 1)

    // Create task
    go func() {
        x <- recurse(depth - 1)
    }()

    y := recurse(depth - 2)

    // Wait for child task to finish
    return <-x + y + 1
}
```

```haskell
recurse :: Integer -> IO Integer
recurse depth
    | depth < 2 = return compute
    | otherwise = do

        mvx <- newEmptyMVar


        forkIO $ do
            x <- recurse (depth-1)
            putMVar mvx x

        y <- recurse (depth-2)
        x <- readMVar mvx

        return (x + y + 1)
```

Listing 2.1: Task-parallel tree recursion in the Go (left) and Haskell (right) programming languages. A word of caution: `return` and `<-` have different meanings in Go and Haskell. What looks like imperative code in Haskell is actually translated into a chain of function calls.

of cores as the runtime system takes care of allocating machine resources. (Assuming the runtime system does not turn into a bottleneck.) Theoretically, task-parallel programs may scale up to the point where all potential parallelism is converted to actual parallelism. When the number of tasks exceeds the number of hardware threads, some of the tasks will be queued and run later.

Scott defines tasks as passive entities, implying that tasks are scheduled by threads, which he defines as active computations [221]. Scott's definition has some appeal, although in practice, tasks may be indistinguishable from user-level threads if implementations choose a direct mapping from tasks to threads [99, 236]. In fact, concurrent programming languages lend themselves to writing programs in a task-parallel style if their runtimes permit user-level threads to execute in parallel (that is, if M user-level threads are scheduled across N kernel threads, see [203], Section 3.8.2.2, pp. 150–151).

Listing 2.1 shows an example of a tree recursion in Go. Tasks are mapped one-to-one to goroutines[2], while goroutines are multiplexed onto native threads by the Go-internal scheduler. Listing 2.1 also includes the same tree recursion written in Haskell using lightweight threads (created with `forkIO`[3]) and synchronization variables (`MVar`s). Unless the number of threads reaches into the tens or hundreds of thousands, and as long as tasks are sufficiently coarse grained, ad-hoc approaches to task parallelism may be surprisingly efficient.

Implementation-wise, it may help to think of tasks as deferred function calls. Listing

[2]A goroutine is a lightweight thread with a variable-sized stack that grows and shrinks as needed.
[3]Haskell has an operator that creates "sparks", which are the equivalent of passive tasks [158].

```c
#include <stdio.h>

typedef struct {
    int (*f)(int, int);
    int a, b;
} Task;

int sum(int a, int b)
{
    return a + b;
}

int main(void)
{
    Task t = {sum, 1, 2};

    printf("%d\n", t.f(t.a, t.b));

    return 0;
}
```

```cpp
#include <iostream>
#include <functional>

struct Task {
    using fun = std::function<int(int, int)>;

    Task(fun f, int a, int b)
        : f_(f), a_(a), b_(b)
    {}

    int operator()() const
    { return f_(a_, b_); }

    fun f_;
    int a_, b_;
};

int main()
{
    Task t([](int a, int b) {
        return a + b;
    }, 1, 2);

    std::cout << t() << "\n";
}
```

Listing 2.2: Tasks as deferred function calls. Examples in C (left) and C++11 (right), whose support for closures makes it possible to create tasks that refer to anonymous functions.

2.2 shows two examples of packaging a function of signature `int(int, int)` to be called at a later time. Packaging a function means storing the function (pointer or closure) along with its arguments in a task descriptor. To achieve parallelism, the function must be called from a different thread context, which requires moving the task, for example by handing it off to a new thread. We will come back to the subject of implementation in Section 2.4.

2.3 Task-parallel Programming

Task-parallel programming shifts the focus from threads to tasks. Programmers can concentrate on finding independent computations and enforcing synchronization where necessary. How these computations map onto actual threads is an implementation detail. To appreciate the difference between threads and tasks, consider the code in Listing 2.3.

Both programs look similar except for their verbosity and different ways of returning values from asynchronously executed functions. But the difference is less a matter of syntactic convenience than a matter of semantics. Task-parallel programs begin with a single thread of execution that *logically* forks into two threads whenever a task is encountered. Tasks can be viewed as hints to the compiler and runtime system

```c
void *do_this(void *arg)
{
    // Compute x
    *(int *)arg = x;
    return NULL;
}

void *do_that(void *arg)
{
    // Compute y
    *(int *)arg = y;
    return NULL;
}

int do_sth_else(void)
{
    // Compute z
    return z;
}

int main(void)
{
    pthread_t thrds[2];
    int x, y, z;

    pthread_create(&thrds[0], NULL,
                   do_this, &x);
    pthread_create(&thrds[1], NULL,
                   do_that, &y);

    z = do_sth_else();

    pthread_join(thrds[0], NULL);
    pthread_join(thrds[1], NULL);

    // Do something with x, y and z

    return 0;
}
```

```c
int do_this(void)
{
    // Compute x
    return x;
}

int do_that(void)
{
    // Compute y
    return y;
}

int do_sth_else(void)
{
    // Compute z
    return z;
}

int main(void)
{
    int x = spawn do_this();
    int y = spawn do_that();
    int z = do_sth_else();

    sync;

    // Do something with x, y and z

    return 0;
}
```

Listing 2.3: The difference between programming with threads and programming with tasks is less a matter of syntax than a matter of semantics. A task is a candidate for parallel execution and as such not guaranteed to run in a separate thread. Examples in C with POSIX Threads (left) and C extended with constructs for task parallelism (right).

about which computations are candidates for parallel execution. As such, a task is not guaranteed to run in a separate *physical* thread. It may be deferred to run at a later time, or it may be executed sequentially to avoid surplus parallelism. Consequently, there is no guarantee that any two tasks will run concurrently, or in parallel, for that matter. In fact, scheduling is the responsibility of the runtime system, and part of the idea of using tasks is to trust the runtime system to make efficient scheduling decisions. Because efficiency is deemed more important than fairness, tasks are usually not preempted but run to completion [10].

Tasks are meant to make it easy to express fine-grained parallelism, which applications must exhibit to benefit from increasingly parallel hardware [219]. Programs may create millions of tasks, rendering a direct mapping from tasks to threads impractical in general due to the cost of thread creation and context switching. Instead, tasks are executed by a pool of worker threads, mirroring the available hardware parallelism. A common approach is to create a worker thread for each physical or logical processor. Irregular algorithms, for which the amount of parallelism may not be known until runtime, tend to create large numbers of tasks ranging from very fine grained (on the order of a few 1000 processor cycles) to coarse grained (on the order of milliseconds to seconds). The challenge of extracting parallelism from a set of tasks lies in efficient scheduling and load balancing, happening dynamically at runtime.

Common to all task-parallel programs is the need to create tasks and synchronize their execution to be able to express meaningful computations. The example in Listing 2.3 uses two language keywords, which were introduced by Cilk: `spawn f(...)` runs `f(...)` as a task, possibly in parallel with the rest of the program, and `sync` waits for the completion of all tasks created in the scope of the function. While details vary between implementations, similar constructs appear in every programming model that is based on tasks.

Roughly classified, task-parallel programming models are either 1) language based or 2) library based [63]. Language-based approaches to task parallelism include new languages, such as X10 [65, 217] and Chapel [1, 69], and extensions to existing languages, such as the Cilk extensions to C [95], the Intel Cilk Plus extensions to C/C++ [6], and the Habanero Java extensions to Java [64]. The most prominent and widely used libraries for task parallelism are the Java Concurrency Utilities (JUC) [145, 193], Intel's Threading Building Blocks (TBB) [205], Microsoft's Task Parallel Library (TPL) for .NET [149], and Apple's Grand Central Dispatch (GCD) [4].

Beginning with version 3.0, OpenMP has started to support tasks to better handle unstructured parallelism [40, 17, 18]. The next section will introduce a task model similar to that of OpenMP[4], which, as we see it, strikes a good balance between expressiveness and potential performance.

2.4 Implementing a Task Model

For the programmer, it is important that a task model is easy to use and flexible enough to express common task-parallel patterns such as fork/join. The implementer's job is to find a good trade-off between the flexibility of a task model and the efficiency of its implementation. The task model is supported by a runtime system/library that manages parallel execution, including task creation, scheduling, and load balancing. We assume programs are written in C, which is also the language of our runtime system.

2.4.1 Interface

We keep the interface fairly simple, relying on preprocessor macros to hide low-level details that are of no concern to the programmer. Macros provide a simple layer of abstraction that would otherwise require compiler support in the form of language constructs, such as Cilk's **`spawn`** and **`sync`**. Without compiler support, however, we have to assume that tasks refer to named functions rather than arbitrary function-like objects, owing to the lack of closures in C. We provide the following macros:

`TASKING_INIT()`

Initializes the runtime system, creating worker threads, setting up data structures, and causing worker threads to wait for tasks. The number of worker threads is taken from the environment variable `TASKING_NUM_WORKERS` and defaults to the number of available processors minus one, since one thread is already running.

`TASKING_EXIT()`

Finalizes the runtime system, completing remaining tasks, cleaning up resources, and joining worker threads.

`TASKING_BARRIER()`

Executes a task barrier, which blocks the caller until all tasks created prior to the barrier, including tasks created transitively, have finished execution. A task barrier boils down to detecting termination of a task-parallel computation—a problem that will be discussed in depth in Sections 4.1 and 4.2.

[4]More precisely, the task model is similar to that of *tied* tasks in OpenMP. In addition, OpenMP supports *untied* tasks, which, when suspended, can resume execution on any thread in the team.

`ASYNC(f, a, b, ...)`

Creates a task for calling `f(a, b, ...)` and adds it to the task pool. Every function f that is called like this must have a corresponding declaration of the form `ASYNC_VOID_DECL`(f, τ_1 x_1; $\ldots$; τ_n x_n; x_1, $\ldots$, x_n), where x_i is the ith parameter of f, and τ_i is the type of x_i. Note that f is assumed to have no return value.

`ASYNC(f, a, b, ..., &res)`

Creates a task for calling `res = f(a, b, ...)` and adds it to the task pool. Every function f that is called like this must have a corresponding declaration of the form `ASYNC_DECL`(τ_f, f, τ_1 x_1; $\ldots$; τ_n x_n; x_1, $\ldots$, x_n), where τ_f is the return type of f, x_i is the ith parameter of f, and τ_i is the type of x_i. Synchronization is needed before the caller can use the value of `res`.

Creating a task involves adding it to the task pool, the set of tasks that are ready to run [203]. An asynchronous function that returns a value to the caller can be generalized to a *future* [91]:

`FUTURE(f, a, b, ...)`

Creates a task for calling `res = f(a, b, ...)`, adds it to the task pool, and returns a value of type `future`, a placeholder for the result `res`, whose implementation will be revealed in Section 4.3. Every function f that is called like this must have a corresponding declaration of the form `FUTURE_DECL`(τ_f, f, τ_1 x_1; $\ldots$; τ_n x_n; x_1, $\ldots$, x_n), where τ_f is the return type of f, x_i is the ith parameter of f, and τ_i is the type of x_i. The future's result must be awaited before it can be used.

`AWAIT(fut, typ)`

Blocks the caller until the result of future `fut` is available and returns it. This result is assumed to have type `typ`. A regular, blocking function call such as `int s = f(a, b)` can be simulated with `int s = AWAIT(FUTURE(f, a, b), int)`.

First introduced in functional programming languages, such as Multilisp, a parallel dialect of Scheme [107], futures have found their way into numerous other programming languages [65, 149, 230, 193]. Essentially, a future is a placeholder for an asynchronous computation, a proxy for the eventual result of that computation. In the context of task parallelism, a future represents the pending result of a task. “Forcing” or “touching” a future means waiting for the result to be determined.

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "tasking.h"

int sum(int a, int b)
{
    return a + b;
}

ASYNC_DECL(int, sum, int a; int b, a, b);

int main(void)
{
    int s, a = 4, b = 2;

    // Start of parallel region
    TASKING_INIT();

    ASYNC(sum, a, b, &s);

    TASKING_BARRIER();

    assert(s == 6);

    // End of parallel region
    // Implicit barrier
    TASKING_EXIT();

    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <omp.h>

int main(void)
{
    int s, a = 4, b = 2;

    // Start of parallel region
    #pragma omp parallel
    {
        #pragma omp master
        {
            #pragma omp task \
            firstprivate(a, b) \
            shared(s)

            s = a + b;
        }

        #pragma omp barrier

        assert(s == 6);

    } // End of parallel region
    // Implicit barrier

    return 0;
}
```

Listing 2.4: Basic structure of using tasks with our runtime system compared to OpenMP 3.x. The main difference between the two tasking models is that the code between `TASKING_INIT` and `TASKING_EXIT` is executed by a single thread, thread 0, whereas in OpenMP, a parallel region is executed by every thread that is created at the enclosing **`#pragma`** `omp parallel` (`OMP_NUM_THREADS-1` threads, unless overriden by a call to `omp_set_num_threads` or by a `num_threads` clause).

Tasks and futures have a lot in common. Both refer to asynchronous computations that provide opportunities for parallelism. Coming from functional programming, futures are associated with having a result that is used later in the program. Technically speaking, futures are allowed to return nothing, in which case only their side effects matter (see, for example, the concept of a **`void`** future in [167], pp. 262–271). Distinguishing between tasks and futures makes it possible to optimize the former while retaining the flexibility of the latter. Unlike futures, tasks need not return handles, signal completion, or deliver results since execution can be synchronized by means of other constructs. This is also what distinguishes tasks in our implementation from tasks in X10. The latter are associated with an enclosing finish scope that tracks the number of pending tasks and must be updated accordingly [137].

We will describe the implementation of futures in Section 4.3 and focus on `ASYNC` tasks for now. Listing 2.4 shows a toy example in which the main thread of control, called master, creates a task for evaluating `s = a + b`. The code on the left interfaces with our runtime system, the code on the right uses OpenMP compiler directives. Whether the sum is actually calculated in parallel is determined at runtime[5]. After the barrier in line 22, the task is done and its result can be used (documented with an assertion in line 24). In our implementation, a task barrier is initiated by the master, whereas in OpenMP, a task barrier requires a thread barrier, which means that all workers must synchronize in addition to executing tasks. We will have more to say about task barriers and synchronization in Chapter 4.

2.4.2 Implementation

Listing 2.5 shows the OpenMP version of the program after **`#pragma`** `omp` lowering and expansion[6]. What the compiler does is rewrite the program to allow for parallel execution by multiple threads. Parallel regions and tasks are outlined into separate functions. Library calls are inserted to create and join threads, define tasks, and synchronize execution.

Listing 2.6 shows our version of the program after preprocessing, in abbreviated form to make it more readable. Every function that may be called asynchronously must have a corresponding declaration, which generates the necessary code for interfacing with the runtime library. We distinguish between asynchronous functions that return values to the caller, as in this example, and functions that return nothing. Thus, we have different types of declarations, `ASYNC_VOID_DECL` and `ASYNC_DECL`, the latter requiring the return type as the first macro argument. Tasks are created with `ASYNC`, which is so named because it marks an asynchronous function call that returns immediately, without waiting for the function to complete. Notice the similarity to Listing 2.2. A task refers to a function and must save the arguments with which the function will be called. At this point, the arguments are captured, but the function call is deferred. Strictly speaking, the last value passed to `ASYNC` is not an argument of `sum`, but reserved for `sum`'s return value, which is required by the caller. This is analogous to how OpenMP compilers treat `shared` variables.

Task functions are auto generated and type erased to **`void(void *)`**, which makes them easy to call from the runtime library, but requires that all arguments be packed into a data structure and unpacked again in preparation for running the task. Tasks are enqueued via `rts_push`, which is not exposed to the programmer. By the time `ASYNC` "returns", the new task may have already started running on a different worker.

[5]Parallel execution is unlikely because there is only one task and its result is immediately required.
[6]`gcc -fopenmp -fdump-tree-ompexp-all` (GCC 4.8.1, openSUSE 13.1)

```c
struct omp_data {
    int a, b, s;
};

struct omp_task_data {
    int a, b, *s;
};

// Outlined task
void omp_task_fn_0(struct omp_task_data *task_data)
{
    int a, b, *s;

    a = task_data->a;
    b = task_data->b;
    s = task_data->s;

    *s = a + b;
}

// Outlined parallel region
void main_omp_fn_0(struct omp_data *omp_data)
{
    int s, a, b;

    struct omp_task_data task_data;

    if (omp_get_thread_num() == 0) {
        task_data.a = omp_data->a;
        task_data.b = omp_data->b;
        task_data.s = &omp_data->s;
        GOMP_task(omp_task_fn_0, &task_data, NULL, 16, 8, 1, 0);
    }

    GOMP_barrier();

    s = omp_data->s;

    assert(s == 6);
}

int main(void)
{
    int s, a = 4, b = 2;

    struct omp_data omp_data;

    omp_data.a = a;
    omp_data.b = b;
    omp_data.s = s;

    GOMP_parallel_start(main_omp_fn_0, &omp_data, 0);

    main_omp_fn_0(&omp_data);

    GOMP_parallel_end();

    a = omp_data.a;
    b = omp_data.b;
    s = omp_data.s;

    return 0;
}
```

Listing 2.5: Program 2.4 (right) after GCC's source-level transformations from OpenMP to multithreaded code (abbreviated and added comments to make it more readable).

```c
int sum(int a, int b)
{
    return a + b;
}

// ASYNC_DECL expands to a data structure to hold the task's arguments
struct sum_task_data {
    int a; int b; int *s;
};

// and a task function that wraps the call to sum
void sum_task_func(struct sum_task_data *d)
{
    typeof((d)->a) a = (d)->a;
    typeof((d)->b) b = (d)->b;
    typeof((d)->s) s = (d)->s;

    int tmp = sum(a, b);
    *s = tmp;
}

int main(void)
{
    int s, a = 4, b = 2;

    tasking_init();

    do { // ASYNC creates a task and enqueues it
        Task *__task = task_alloc();
        struct sum_task_data *__d;
        __task->parent = get_current_task();
        __task->fn = (void (*)(void *))sum_task_func;
        __d = (struct sum_task_data *)__task->data;
        *(__d) = (typeof(*(__d))){ a, b, &s };
        rts_push(__task);
    } while (0);

    rts_barrier();

    assert(s == 6);

    tasking_exit_signal();
    tasking_exit();

    return 0;
}
```

Listing 2.6: Program 2.4 (left) after preprocessor macro expansion (abbreviated and added comments to make it more readable).

Initialization, finalization, and barrier synchronization expand directly to runtime library calls as in OpenMP. The main semantic difference between OpenMP tasking and our implementation comes from `TASKING_INIT`. In OpenMP, parallel regions are outlined as functions and executed by *every* thread in the team (hence the pragma guarding task creation). In our implementation, the master goes on to execute the code between `TASKING_INIT` and `TASKING_EXIT`, whereas newly created worker threads enter a scheduling loop in the runtime library. The program remains sequential until an asynchronous function call creates a task to be picked up by one of the worker threads or the master itself. Eventually, the master orders termination by calling `TASKING_EXIT` and joins the worker threads as they complete. We can think of the code between `TASKING_INIT` and `TASKING_EXIT` as the top-level or root task, and all tasks that are created during the execution of a program as child tasks. Every child task has a parent task, and, in fact, we store a reference to the parent in the task descriptor. We will see how this reference is used when we look at futures in Section 4.3.

2.5 Task Scheduling

Task-parallel programming raises the level of abstraction from managing threads to specifying tasks, which may or may not run in parallel, depending on decisions made at runtime. These decisions are the responsibility of the task scheduler, which determines the mapping from tasks to worker threads. Consequently, much of the efficiency of using tasks depends on the scheduler and its ability to handle workloads, which differ in the number, granularity, and order of tasks, including potential dependencies. The granularity of a task is defined by the amount of computation in relation to communication/synchronization. Tasks that compute very little are fine grained. Dividing a workload into many fine-grained tasks is good for load balancing and scalability, as long as the cost of scheduling does not outweigh the benefit of parallel execution. The finer the granularity, the more attention must be paid to low overheads.

Depending on workload parameters, we roughly distinguish between static and dynamic workloads. Static workloads are predictable, not subject to change at runtime, and lend themselves to static scheduling. Dynamic workloads depend on input or other program parameters not known until runtime and tend to vary as the computation unfolds, making static scheduling impractical in most cases.

2.5.1 Static Scheduling

The scheduling algorithm determines how tasks are mapped to threads for execution. In static scheduling, the assignment of tasks to threads is fixed, either as part of the compilation process or at runtime. As an example, consider the static scheduling (`schedule(static)`) of loop iterations in OpenMP: a parallel loop of N iterations is compiled such that every thread executes N/T iterations, assuming N is an even multiple of T, the number of threads. Thread 0 will then execute the first N/T iterations, thread 1 will execute the next N/T iterations, and so on. This division incurs (almost) no runtime overhead, but is only efficient as long as N/T iterations roughly equal N/T percent of the total work. If that is not the case and load imbalance arises, static scheduling has no means to counter it.

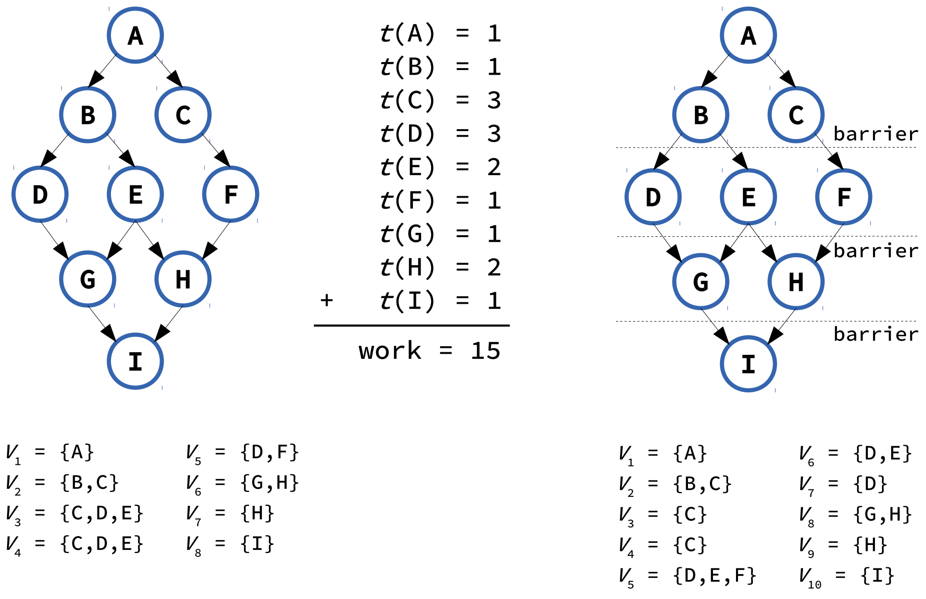
2.5.2 Dynamic Scheduling

Dynamic scheduling, on the other hand, assigns tasks to threads as needed, thereby maintaining load balance. The price to pay for dynamic scheduling is increased runtime overhead. Returning to the example of scheduling parallel loops in OpenMP, it is a good idea to use dynamic scheduling (`schedule(dynamic)`) whenever the amount of work per iteration is unknown or varies widely. The dynamic scheduler keeps track of iterations in a shared location, so that idle workers can claim new iterations and come back for more once they have finished their current batch of work. The higher runtime overhead compared to static scheduling is the result of fetching iterations, including contending for iterations with other threads.

The shared location can be generalized to a task pool, a data structure that holds all tasks that are ready to run, not necessarily in any particular order. Tasks are added to the task pool and retrieved for execution [203].

2.5.3 Task Graphs

Tasks and their dependencies form a task graph: a directed acyclic graph (DAG), where every node represents a task, and a directed edge from node A to node B indicates that B depends on results computed by A [34]. Figure 2.1 shows an example task graph that can be scheduled with the help of a task pool. We use similar notation to Blelloch et al. [46]: Each vertex v represents a task of duration $t(v)$. The sum of all $t(v)$ is the *work* of the graph, or the time it takes to execute all tasks sequentially. A dependency (u, v) between two tasks u and v means that u must complete before v can start execution. Hence, v is not ready until all the tasks it depends on have been



```
  t(A) = 1
  t(B) = 1
  t(C) = 3
  t(D) = 3
  t(E) = 2
  t(F) = 1
  t(G) = 1
  t(H) = 2
+ t(I) = 1
-----------
  work = 15
```

V_1 = {A}	V_5 = {D,F}
V_2 = {B,C}	V_6 = {G,H}
V_3 = {C,D,E}	V_7 = {H}
V_4 = {C,D,E}	V_8 = {I}

V_1 = {A}	V_6 = {D,E}
V_2 = {B,C}	V_7 = {D}
V_3 = {C}	V_8 = {G,H}
V_4 = {C}	V_9 = {H}
V_5 = {D,E,F}	V_{10} = {I}

Figure 2.1: Example task graph showing tasks (vertices) with their dependencies (edges). The graph on the right-hand side indicates that explicit synchronization may be used to schedule tasks in an order that respects their dependencies, though possibly at the cost of losing parallelism, as in this example. The resulting schedule is non-*greedy* [48], unlike the schedule on the left-hand side.

completed. For simplicity, we assume a perfect task pool with zero overhead for task insertion and removal, so that tasks are scheduled as soon as they enter the task pool.

Figure 2.1 includes two P-processor schedules, $P \geq 3$. A schedule is a sequence $(V_1, V_2, \cdots, V_T)$, where V_i is the set of tasks that are running during time step i. For a complete formal definition, we refer the reader to [46]. The *greedy* schedule on the left-hand side assumes that tasks are created and scheduled as soon as their dependencies are satisfied [48]. Task H, for example, depends on both E and F and is guaranteed to be scheduled during the time step following the completion of either E or F, whichever finishes last. More generally, if E is scheduled during time step i and F is scheduled during time step j, H will be scheduled during time step $\max(i + t(E), j + t(F))$. This does not hold for the schedule on the right-hand side, which relies on explicit synchronization to ensure that tasks are scheduled in an order that respects their dependencies. A task may be ready, but its creation may be deferred until other, possibly unrelated tasks have finished execution, due to explicit synchronization. The potential loss of parallelism is often accepted as a trade-off for a simpler programming model and a runtime system that avoids task dependencies and the associated overhead of determining when tasks are ready to be scheduled.

2.6 Task Pools

Task pools can be distinguished in two categories depending on their implementation: central and distributed [131, 258]. A central task pool is a global container for tasks, which workers are free to enqueue and dequeue. A distributed task pool contains a set of local task pools, usually one per worker. Workers operate on their local task pools and rely on load balancing when running out of work.

Central task pools provide implicit load balancing, but require synchronization for every task pool access. Frequent task pool access leads to contention, which can harm scalability; a consequence of N workers trying to access a shared data structure.

Distributed task pools solve the scalability problem of central task pools, but with distribution comes the need for load balancing to divide work evenly among available workers. Local task pools may be completely unsynchronized, in which case load balancing must be realized by sending tasks among workers.

The limited scalability of central task pools is problematic even on hardware with relatively modest core counts. Figure 2.2 compares two OpenMP tasking implementations using the UTS benchmark (described at the end of this chapter) [184, 25]. GNU's libgomp [3] implements a central task pool; Intel's runtime library [9] assigns a task queue to every worker thread and uses work stealing for load balancing. We will take a closer look at work stealing in the next section. GCC's implementation of OpenMP performs best with up to eight workers; above eight, every worker taking part in the computation increases task pool contention and causes execution to slow down. Intel's implementation on the other hand demonstrates that scalability is not elusive. With 48 workers, the performance gap between central and distributed task pools has reached a factor of 223. Central task pools provide a simple solution for small-scale systems, but their limitations are apparent. For this reason, we will focus on distributed task pools and will not further consider central task pools.

2.7 Load Balancing

Distributed task pools introduce the need for load balancing. What happens when some workers exhaust their local task pools while others still have tasks left? There are two ways to distribute tasks: (1) Share some of the local tasks with underutilized workers, or (2) let idle workers take the initiative to "steal" some of the local tasks of other workers. The former is called work sharing, the latter work stealing [171, 49].

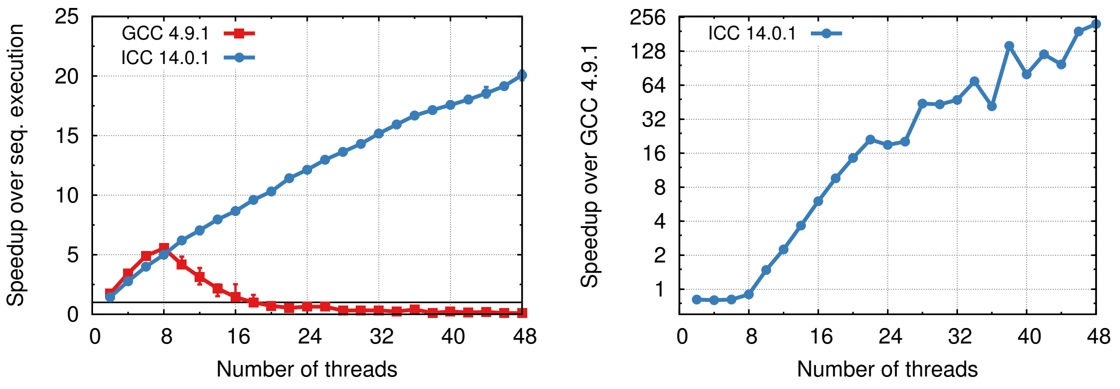


Figure 2.2: Testing the OpenMP tasking implementations of GNU and Intel C compilers: GNU libgomp versus Intel OpenMP Runtime Library. The benchmark that is used here is UTS with binomial tree T3L, which has about 111 million nodes (80% leaves) and a depth of 17 844. Benchmark programs are detailed in Section 2.7. In the graph on the left, speedups are based on median execution times of three individual runs and refer to slightly different sequential baselines, depending on which compiler is used: 37.8 seconds in the case of GCC and 39.5 seconds in the case of ICC. The graph on the right plots the ratio of execution times. At 48 threads, the performance gap between GNU's and Intel's runtime libraries has reached a factor of 223. (GCC 4.9.1, `-O3`, ICC 14.0.1, `-O3`, AMD Opteron multiprocessor)

2.7.1 Work Sharing

Work sharing can be implemented by maintaining a separate pool of tasks, which workers can turn to after running out of work [131, 200]. Workers with tasks to spare must prevent this pool from draining, or else idle workers will not be able to go back to work. A central task pool represents the simplest possible implementation of work sharing because every task is shared once it is enqueued, hence the reason why the load is considered to be balanced at all times.

2.7.2 Work Stealing

Work stealing was named after the fact that concurrent data structures, most notably concurrent deques, allow "thieves" to "steal" tasks from "victims" without interfering with the victims' execution[7]. In systems with distributed address spaces, work stealing requires cooperation between victims and thieves: victims send tasks in reaction to steal requests they receive. Because of this explicit message exchange, some authors prefer the term work requesting [178]. Work sharing and work stealing are instances of sender-initiated and receiver-initiated load balancing [81, 44]. Tasks are transferred

[7]This is a bit of a simplification because thief and victim may have to synchronize.

from sender to receiver as a result of either the sender's actions (work sharing) or the receiver's actions (work stealing).

The Cilk multithreaded runtime system The idea of work stealing originated from research on parallelism in functional programming languages in the early 1980's [59, 106, 107]. Much of the groundwork that influenced the design and implementation of schedulers was laid in the Cilk project at MIT [21]. Cilk established a provably efficient, randomized work-stealing scheduler for fully-strict computations, in which child tasks are required to synchronize with their parents (well-structured fork/join computations) [95, 49].

Tasks and child tasks A multithreaded program can be viewed as a DAG of computations (vertices) linked by dependencies (edges) [48]. Spawn edges create child tasks, which represent potential parallelism, and join edges introduce synchronization. In a fully-strict DAG, every task Γ must synchronize with its parent, the task that spawned Γ. In a strict DAG, every task Γ must synchronize with one of its ancestors, the parent of Γ or, recursively, an ancestor of the parent of Γ. A DAG in which every task ends with a join edge is said to be terminally strict [32]. Figure 2.3 shows examples of fully-strict and strict multithreaded computations that are also terminally strict. Terminal strictness is often implied when waiting for completion is the only means of synchronization between tasks and child tasks.

Let T_1 be the execution time of a fully-strict computation DAG on one processor, T_P be the execution time on P processors, and T_∞ be the execution time on an infinite number of processors. The latter is also known as the critical path length: the theoretically shortest execution time resulting from the longest chain of sequential dependencies. Cilk's work-stealing scheduler executes a computation on P processors in expected time $T_1/P + O(T_\infty)$, assuming the two bounds $T_P \geq T_1/P$ and $T_P \geq T_\infty$ are met. The result is near-optimal linear speedup if $T_1/P \gg T_\infty$, or equivalently, if $T_1/T_\infty \gg P$, that is, the potential parallelism far exceeds the number of processors, which highlights the importance of breaking down a program into many independent tasks. T_1 is known as the work of the computation, and the ratio T_1/T_S describes the work overhead relative to the serial elision with execution time T_S[8].

The work-first principle Cilk's *work-first* principle states that the work overhead should be minimized, since it has a big impact on performance, whereas overheads on

[8]The serial elision of a Cilk program is obtained by removing all Cilk keywords [95].

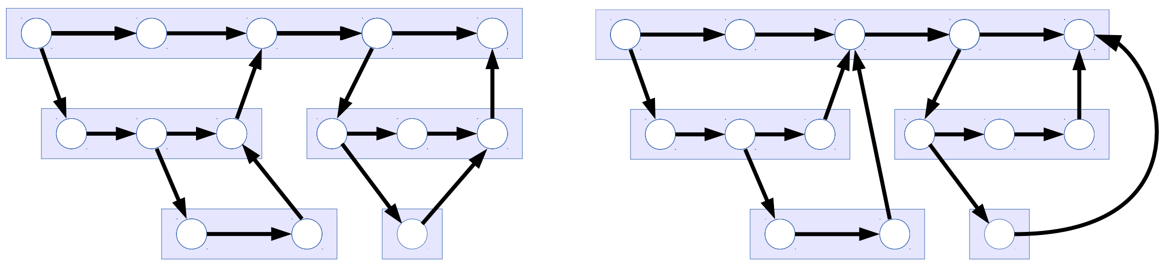

(a) Fully-strict computation DAG **(b)** Strict computation DAG

Figure 2.3: The DAG model for multithreading. Tasks, drawn as rectangles around sequential computations (vertices) and continuations (horizontal edges), are connected by spawn and join edges. A spawn edge creates a new task, which may execute in parallel with other tasks. A join edge ends a task after synchronizing with the parent (a) or an ancestor (b). The *work* is the time it takes to execute all computations in a DAG, and the *span*, or *critical-path length*, is the time it takes to execute the longest path of dependencies. The ratio of work to span gives the maximum possible speedup for any number of processors. It is an indication of how much potential parallelism a DAG contains. Strictness does not require synchronization between parent and child tasks, allowing computations at different levels of the spawn tree to execute in parallel without unnecessary dependency constraints. There are two such opportunities in this example, with the result that the strict computation DAG contains more potential parallelism than its fully-strict counterpart. In (a), the ratio of work to span is $14/12 = 1.1\overline{6}$, whereas in (b), it is $14/10 = 1.4$.

the critical path T_∞ are much less important as long as sufficient parallelism exists, and steals are rare. In other words, optimizations should target the common case, the execution path where no work is stolen, even if that means adding overheads to the critical path. Specifically, as much of the scheduling cost as possible should be shifted to idle workers, since idle workers have no other work to do. Similar time, space, and communication bounds have been proved for the more general class of strict multithreaded computations, in which child tasks are required to synchronize with their ancestors but may outlive their parents [85, 32].

Work-first scheduling Cilk's task creation strategy is the result of strict adherence to the work-first principle. As in lazy task creation [172], a task is turned into a function call, while the continuation following the spawn operation is pushed onto the deque for idle workers to steal [95]. Upon returning from the task, the worker that pushed the continuation checks if the continuation has been stolen, and if so, becomes a thief itself. Otherwise, the worker picks up and resumes the continuation.

Help-first scheduling The alternative to Cilk's task creation strategy is to queue the task and execute the continuation. Because this strategy neither assumes compiler support nor a continuation-passing programming style, it is the strategy of choice for many tasking libraries, including our own. Guo et al. call the "steal child" approach help-first, to suggest that help is needed to run a task, and to distinguish it from Cilk's "steal parent" approach, which is a consequence of the work-first principle [101]. Help-first is less space efficient than work-first. In theory, help-first may require unbounded space and may overflow heap memory, whereas work-first provably requires at most S_1P space, a multiple of the space required by the serial execution S_1. A simple example may help to visualize the difference between help-first and work-first. Consider the following loop, in which a single thread spawns `N` tasks before running them in sequence at the **`sync`** statement, all under the assumption that no thieves are present [208]:

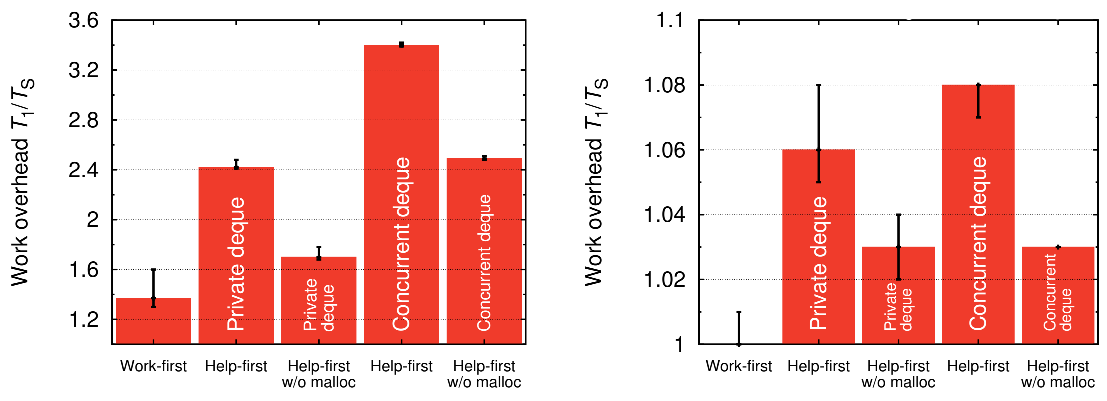
```
for (i = 0; i < N; i++)
    spawn f(i);
sync;
```

With help-first, the program requires space proportional to `N`, because `N` tasks are created and enqueued before the **`sync`** statement is reached, at which point tasks are scheduled for execution. With work-first, the program runs in constant space, deferring only the continuation of the loop in each iteration. Suppose the loop spawns one million tasks, each taking up 192 bytes[9]. Running the program on one processor will allocate 192 MB of memory to store the tasks, which will make the loop noticeably slower than its sequential version. Using our runtime system, for example, we measure a work overhead of 2.42, compared to 1.37 for Cilk Plus, when function `f` does nothing else but return (see Figure 2.4 (a)). Half of the work overhead can be attributed to allocating memory. By allocating memory ahead of time, the work overhead drops to 1.7. The remaining overhead compared to sequential execution is caused by task creation and deque operations, which cannot be eliminated without serializing tasks.

For comparison, we show how using a concurrent deque, similar to Cilk's implementation [95], can affect the work overhead. Private deques have the advantage of efficient operations, but require that victims steal on behalf of thieves, which is at odds with the work-first principle as formulated by Cilk.

Figure 2.4 (b) indicates that work overheads tend to be small for non-empty tasks. Given tasks of one microsecond, which we count as fine-grained parallelism, work-first has no measurable overhead on average, while help-first adds 6% to the sequential execution time.

[9]The actual size of a task in our implementation.



(a) Task length of $0\mu s$ **(b)** Task length of $1\mu s$

Figure 2.4: Work overhead T_1/T_S for running a sequential loop that spawns one million tasks using work-first and help-first task creation strategies. Operating a concurrent deque is more expensive than operating a private deque; hence the greater work overhead. (ICC 14.0.1, `-O2`, Intel Core i7-4770)

Advantages of help-first are less stack pressure and better performance when steals are frequent. (Remember that one of the assumptions underlying the work-first principle is that steals are rare.) A more practical advantage of help-first is that it tends to be easier to reason about than work-first. Imagine a sequence of statements that looks like this:

```
S1;
spawn S2;
S3;
```

Under work-first, `S1` and `S3` are executed by different threads if a thief steals the continuation following `S2` while the worker is still busy with `S2`. For this reason, we may observe that a procedure is called by a thread T_1, but returns on a different thread T_2 [210]. Under help-first, `S2` may be executed by a different thread, but `S1` and `S3` are guaranteed to run on the same thread (intuitive function call/return).

Guo et al. have developed an adaptive work-stealing scheduler for Habanero Java that switches between help-first and work-first depending on the stealing rate and recursion depth [102]. If steals are rare, the scheduler operates under the work-first policy in order to guarantee bounded use of space. Otherwise, if steals are frequent, or if stacks exceed a certain depth, the scheduler prefers to create tasks according to the help-first policy. Scheduling decisions are reevaluated periodically.

A simpler way to try to combine the benefits of work-first and help-first is to bound the number of tasks per deque [170]. Bounded deques make it impossible to enqueue tasks beyond a certain threshold, which bounds the memory consumption and work overhead because tasks that cannot be enqueued are executed sequentially without being created. Tasks are created only when deque space is available. Such a scheduler can be viewed as alternating between help-first and work-first if we think of work-first as invoking a task directly, without creating a stealable continuation.

An interesting load balancing strategy, which can be labeled work stealing in retrospect, was proposed by Rudolph et al [214]. Every processor has a local "workpile" (task queue) on which it operates. Whenever a processor accesses its own workpile, it performs load balancing with a probability that is inversely proportional to the size of its workpile. Load balancing consists of choosing a random workpile and equalizing the number of tasks in the two workpiles. When a processor's local workpile contains fewer tasks than the randomly chosen workpile of another processor, equalizing the number of tasks in the workpiles amounts to stealing work from the other processor.

2.7.3 Data Structures for Work Stealing

Many implementations of work stealing assume that workers communicate through shared memory. The usual implementation is as follows: Each worker maintains a deque of tasks, which it treats as a stack, pushing and popping tasks at the bottom [37, 111, 66]. When a worker finds its deque empty, it becomes a thief, randomly selects a victim among the other workers, and tries to steal a task from the top of the victim's deque. If the steal fails, for example, because the victim has no tasks that could be stolen, the thief selects a new victim and tries again, repeating the procedure until a steal succeeds.

Work-stealing deques provide three main operations: *push*, *pop*, and *steal*. One thread pushes/pops tasks to/from one end of the deque, while other threads are allowed to steal tasks from the other end. The work-first principle suggests that *push* and *pop* should be fast operations because they contribute to the work overhead. Cilk introduced the THE protocol to manage deques [95]. In Cilk, each deque has an associated lock, but only *steal* requires locking; *pop* avoids locking in the common case, requiring only a memory fence, unless victim and thief contend for the last task, in which case locking is used to resolve the conflict. Arora et al. removed the need for locking with non-blocking (lock-free) deques[10] [37]. Chase and Lev improved on Arora et al.'s implementation by using a ring buffer that can grow to prevent overflow [66].

[10]Lock freedom gives a progress guarantee: among all threads trying to perform a deque operation, one is guaranteed to succeed [114]. Lock freedom implies that no thread can stall the system by preventing other threads from making progress on their own.

Non-blocking deques require the use of a universal synchronization primitive such as compare-and-swap (CAS) [113]. A *steal* involves one CAS. A *pop* involves one CAS if it tries to take the last task and must prevent a race with a concurrent *steal*; otherwise, a *pop* may require only a memory fence analogous to the THE protocol.

Michael et al. have shown that memory fences can be avoided if workers are allowed to remove a task more than once [169]. Morrison et al. have found a way to avoid memory fences by deriving a reordering bound under a total store ordering (TSO) memory model [176]. The reordering bound depends on the size of the processor's store buffer and the number of stores between invocations of *pop*, which can be determined with the aid of static code analysis. When a thief tries to steal a task, but the number of tasks is below the reordering bound, thief and victim might be contending for the same task. To maintain the safety property, the thief either aborts or seeks to synchronize with the victim to verify that the task is stealable (basically sending a steal request). Option one, aborting, relaxes the semantics of work stealing, as steals may fail not only in the presence, but also in the absence of conflicts. Option two, synchronization, introduces the need for polling to detect when steals are stalled until acknowledged by the victim.

Classic work stealing with *push* and *pop* operating in a LIFO manner and *steal* operating in a FIFO manner is biased towards divide-and-conquer algorithms, in which tasks close to the root of the computation carry large chunks of the total work. A single steal can thus transfer a substantial amount of work, and a few steals may suffice for load balancing. Some work-stealing deques support stealing multiple tasks at once, usually up to half of the tasks in a deque [111]. Deques, or task queues in general, may be split into private and public parts to allow unsynchronized access to a subset of the tasks [131, 74, 121, 247]. While split deques permit efficient implementations of *pop*, they require their owners to regularly check (poll) if there are tasks available to thieves and, if not, try to move some tasks from the private to the public region. Implementation-wise, split deques are halfway between shared and private deques. Thieves can steal on their own as long as tasks are publicly available. If there is nothing left to steal, thieves post "split requests" to which recipients should respond in a timely manner, hence the need for polling [247].

2.7.4 Distributed Work Stealing

Early in the history of work stealing, researchers concluded that shared memory would enable more efficient work-stealing policies than possible with message passing [229]. In distributed-memory environments, the necessity to communicate by passing messages dictates a certain amount of cooperation between victims and thieves.

Distributed work stealing is either centered around explicit message passing, most often in the form of MPI-like send/receive [93], or implemented on top of higher-level global address space abstractions, which hide the communication from the parallel runtime. Dinan et al. describe work-sharing and work-stealing implementations of the UTS benchmark using MPI [75]. In later work, Dinan et al. show the scalability of work stealing in the context of PGAS [73, 74]. Communication libraries such as ARMCI [183] and GASNet [53], or the recently released MPI-3 standard [16], provide APIs for one-sided remote memory access (RMA) operations that facilitate the implementation of work stealing on distributed-memory systems. Work-stealing schedulers written in GAS/PGAS languages interface with the same communication libraries, if not directly, at least indirectly [185, 222, 170].

Even though message passing schedulers have the benefit of flexibility and portability, they are seldom considered in the context of shared memory. Feeley argued that a message passing implementation of Lazy Task Creation (LTC) [172] is both simpler and more efficient than an implementation that relies on shared memory [89]. He showed that a private stack of suspended continuations can be cached efficiently (write-back), unlike a shared stack, which causes writes to main memory in the absence of cache coherence (write-through). In addition to improving parallel performance, he was able to reduce the work overhead of LTC.

Lazy Threads [99] and StackThreads/MP [236], both inspired by LTC, reduce the cost of thread creation by allocating stacks lazily. In Lazy Threads, for example, a child executes on the stack of its parent. The parent is suspended until the child blocks or returns, or a steal request causes the parent to be migrated to another processor. Both Lazy Threads and StackThreads/MP rely on polling to detect steal requests.

Hendler et al. proposed work dealing, an alternative load balancing strategy akin to work sharing [112]. Instead of operating on deques, workers send every task they create to one of n producer-consumer buffers; every worker has n such buffers from which it consumes tasks. Tasks are distributed according to fixed policies, and once distributed, tasks are never reassigned. Hendler et al. emphasize the locality-oriented nature of work dealing, but also admit that having to traverse up to n buffers before finding work becomes increasingly inefficient for larger values of n. At least on small-scale systems, this overhead seems to be negligible.

Load balancing based on sending tasks between workers is implemented in the Tascell framework [117] and in the Manticore programming language [92, 202]. Tascell is unique in that it does not create tasks in the usual sense, but rather maintains information about points in the program's execution where tasks can be spawned if the need arises. Whenever a worker receives a steal request, it backtracks to the oldest potential spawn point, spawns a task, sends it to the requesting worker, and returns to the point from which it backtracked and resumes execution. Manticore is a parallel variant of Standard ML with a work-stealing runtime system based on private deques and steal request messages. Manticore's decision to use private deques was motivated by the desire to not compromise the performance of garbage collection [202].

In a recent paper, Acar et al. prove that work-stealing algorithms with private deques guarantee the same theoretical bounds as work-stealing algorithms with concurrent deques [28]. The paper proposes a receiver-initiated algorithm based on steal requests and a sender-initiated algorithm similar to work dealing. In both algorithms, worker threads communicate through a set of shared variables, relying on atomic operations to distribute tasks. This limits the flexibility of the schedulers. In the receiver-initiated algorithm, for example, a victim can receive at most one steal request at a time. A steal request is simply a thief's ID written to a victim's request cell. We explore a more general approach than Acar et al. For example, channels allow steal requests to be queued, enabling strategies such as lazy adaptive splitting (see Chapter 5), which greatly contribute to the efficiency of channel-based work stealing.

Work stealing with private deques can be thought of as delegation, in which a server thread does work on behalf of others [61]: a thief requests a task by delegating the steal to a victim, the only thread allowed to access a given deque. One of the benefits of delegation—the abstraction layer that it introduces—makes a case for message-passing schedulers, especially when portability is a concern.

2.7.5 Hardware Support for Work Stealing

Thanks to Moore's Law [173], transistor budgets have grown to the point that some hardware can be dedicated to inter-thread communication and synchronization. Examples include Intel's Single-Chip Cloud Computer (SCC) [124, 165], Tilera's TILE-Gx processor family [256], Kalray's MPPA accelerators [70], and the PRAM-based XMT architecture [255, 250].

Petrovic et al. showed that hardware support for message passing can improve thread synchronization, leading to more efficient data structures including stacks and queues [196]. Calciu et al. observe that delegation-based data structures can sometimes outperform their concurrent counterparts [61]. Sanchez et al. proposed asynchronous messages independent of the cache hierarchy for fast communication between threads [215]. They showed that a few message-passing primitives combined with user-level interrupts upon message arrival suffice to implement flexible and scalable task schedulers. Prior work proposing hardware task queues along with automated load balancing promised significant speedups over software schedulers, but lacked the flexibility that software schedulers provide [136]. Rosas-Ham et al. integrated hardware task queues with control logic to raise and handle steal requests without processor involvement [211]. Like [136], they accept hardwired scheduling for the benefit of simpler runtime systems. Steal request and response messages are sent independently of a software scheduler's operation. Hardware-accelerated queues, in contrast to custom hardware queues, have also been proposed [129, 148].

2.8 Benchmark Programs

This section introduces the set of benchmarks and microbenchmarks on which we evaluate our runtime system throughout the remaining chapters. We do not claim that these programs (written in C) exhibit the best performance among all possible implementations. In particular, matrix multiplication and LU decomposition cannot compete with optimized routines from linear algebra libraries such as Intel's Math Kernel Library [8]. We are primarily interested in programs with challenging task structures and their parallel speedups, which we use to compare the performance of different runtime systems. The variables that appear in the benchmark descriptions correspond to parameters that can be changed by the user.

SPC A **S**imple **P**roducer-**C**onsumer benchmark. A single worker produces n tasks, each running for t microseconds. This benchmark allows us to test how many concurrent consumers a single producer can sustain.

BPC A **B**ouncing **P**roducer-**C**onsumer benchmark, which is a producer-consumer benchmark with two kinds of tasks, producer and consumer tasks [74]. Each producer task creates another producer task followed by n consumer tasks, until a certain depth d is reached. Consumer tasks run for t microseconds. The smaller the values of n and t, the harder it becomes to exploit the available parallelism. This benchmark stresses the ability of the scheduler to find and load-balance work.

Treerec A simple tree-recursive computation, similar in structure to Fibonacci, which is often used to estimate task scheduling overheads [78]. Each task $n \geq 2$ creates two child tasks $n - 1$ and $n - 2$ and waits for their completion. Leaf tasks $n < 2$ perform some computation for t microseconds before returning. This simulates a cut-off, as if tasks were inlined after reaching a certain recursion depth.

Matmul A blocked matrix multiplication of two $N \times N$ matrices of doubles, each partitioned into $(\frac{N}{B})^2$ $B \times B$ blocks [139]. The block size B determines the task granularity and must be a divisor of N.

LU A blocked LU decomposition of a sparse $N \times N$ matrix of doubles, partitioned into $(\frac{N}{B})^2$ $B \times B$ blocks. The block size B determines the task granularity and must be a divisor of N. The sparsity of the matrix—the fraction or percentage of blocks that contain only zeros—increases with the number of blocks in each dimension. Blocks that contain only zeros are not allocated. The code is based on the OpenMP version from the Barcelona OpenMP Tasks Suite (BOTS) [78, 20].

Quicksort A recursive algorithm that performs an in-place sort of an array of n integers by partitioning it into two sub-arrays according to some pivot element and recursively sorting the sub-arrays. The pivot is chosen as the median of the first, middle, and last array elements. For sub-arrays ≤ 100 elements, the algorithm falls back to using insertion sort, which is usually faster on small inputs.

Cilksort A recursive algorithm inspired by [35] that sorts an array of n integers by dividing it into four sub-arrays, recursively sorting the sub-arrays, and merging the sorted results back together in a divide-and-conquer fashion to expose additional parallelism. For sub-arrays ≤ 1024 elements, the algorithm performs a sequential quick sort with median-of-three pivot selection and partitioning, and for sub-arrays ≤ 20 elements, the algorithm falls back to using insertion sort. The code is based on the version distributed with MIT Cilk [21].

NQueens A recursive backtracking algorithm that finds all possible solutions to the N-Queens problem of placing N queens on an $N \times N$ chessboard so that no queen can attack any other queen. The code is based on the OpenMP version from the BOTS project [78, 20], which in turn is derived from the version distributed with MIT Cilk [21].

UTS The **U**nbalanced **T**ree **S**earch: an algorithm that counts all nodes in a highly unbalanced tree [184]. The tree is generated implicitly; each child node is constructed from the SHA-1 hash of the parent node and child index. The code is based on the Pthreads version from the official UTS repository [25].

All benchmarks have the form `init(); compute(); fini();`, with application-specific initialization/finalization in `init`/`fini`, as well as the required calls to start/stop the runtime system. When we talk about a benchmark's execution time, we mean the execution time of `compute`.

Benchmark	**Tasks**	**Median task length**,	**IQR**	**Phases**	**Category**
Matmul 2048 32	262 144	77μs,	1μs	64	flat
LU 4096 64	23 904	556μs,	15μs	128	flat
— `fwd`	1024	363μs,	3μs		
— `bdiv`	1024	332μs,	4μs		
— `bmod`	21 856	557μs,	17μs		
Quicksort 10^8	1 697 314	5μs,	11μs	1	recursive
Cilksort 10^8	1 070 421	34μs,	57μs	1	recursive
— `cilksort`	87 380	82μs,	294μs		
— `cilkmerge`	983 041	19μs,	23μs		
NQueens 14	27 358 552	7μs,	12μs	1	recursive
UTS T1L	102 181 081	$< 1\mu$s,	1μs	1	recursive
UTS T2L	96 793 509	1μs,	1μs	1	recursive
UTS T3L	111 345 630	$< 1\mu$s,	1μs	1	recursive

Table 2.1: Workload characteristics of selected benchmarks. LU and Cilksort comprise different types of tasks, as itemized above. Task lengths were measured on one core of the AMD Opteron multiprocessor, using GCC 4.9.1 with all optimizations enabled (`-O3`). Parallel phases end with barrier synchronization to ensure that all tasks have run to completion.

2.8.1 Speedup and Efficiency

Sequential execution can mean two things: running a sequential version of `compute` or running a parallelized version of `compute`, but using a single thread to do so. To avoid confusion, we follow the convention of Cilk and denote sequential execution times with T_S and T_1. Unlike T_S, T_1 includes the overhead of parallelization so that $T_1 \geq T_S$.

We define the speedup over sequential execution as

$$S_P = \frac{T_S}{T_P}, \tag{2.1}$$

where T_P is the parallel execution time of `compute` with P processors or threads [82]. An alternative definition of speedup over sequential execution would be

$$S_P = \frac{T_1}{T_P}. \tag{2.2}$$

It is usually safe to assume that T_S/T_P gives a lower bound for T_1/T_P.

Intuitively, a program is considered scalable if additional processors speed up the program's execution. If we increase the number of processors by a factor of N, and the program runs roughly N times faster as a result, we speak of linear scaling. Efficiency is defined as the speedup divided by the number of processors used to obtain that speedup [82]:

$$E_P = \frac{S_P}{P}. \tag{2.3}$$

Expressed as a value between zero and one, efficiency indicates how well parallel processors are utilized, compared to how much effort is spent on communication and synchronization. High efficiency means good utilization with little overhead. Programs that scale linearly have an efficiency close to 1.

A note on the graphs in this thesis: when we report execution times, speedups, or efficiencies, we plot the median of ten program runs, except where noted, along with 10th and 90th percentiles, where visible. The difference between the 10th and 90th percentiles, also known as the interdecile range, covers 80% of the dispersion of a data set and is a useful measure of spread around the median. As such, it gives a sense of the variability that we observe between program runs, after excluding minimum and maximum execution times.

2.9 Summary

Task-parallel programming revolves around tasks—independent units of work eligible for parallel execution. Tasks simplify the expression of fine-grained parallelism, while shifting the burden of task creation, management, scheduling, and load balancing to the runtime system, a library that provides these services in addition to maintaining a pool of worker threads to take advantage of multicore systems. The programmer remains responsible for defining tasks and inserting synchronization operations to join tasks, await results, and orchestrate execution.

Task-parallel programs may create millions of short-lived tasks, much more than can possibly run in parallel. This gives the runtime system the freedom to utilize different numbers of cores and helps programs achieve scalable performance.

The scheduling and load balancing technique of choice is work stealing, which is commonly implemented with concurrent deques, using shared memory or a global address space abstraction. Message-based implementations are usually geared toward cluster systems. As it appears increasingly likely that future microprocessors will share some similarities with clusters, message passing may become the most efficient way for threads to communicate with each other. Starting with the next chapter, we will explore an implementation of work stealing that is based on exchanging messages over channels, which lend themselves to efficient message passing on different systems.

3 | Channel-based Work Stealing

The previous chapter ended with a brief summary of the status quo: tasks have emerged as a useful abstraction in parallel programming, and work stealing is becoming an indispensable component of efficient runtime systems. We noted that strict dependence on shared memory may pose a problem when faced with the task of porting a runtime system to a less conventional processor architecture. While we cannot predict the future[1], we argue that runtime systems will benefit from changes that increase their flexibility without sacrificing performance.

This chapter introduces a work-stealing scheduler in which worker threads communicate by sending messages over channels. Section 3.1 starts with a description of channels and our motivation for using them to exchange steal requests and tasks among workers. Before we dive into steal requests in Section 3.3, we give a short overview of the scheduler in Section 3.2. Central to work stealing is choosing victims and deciding how many tasks to steal. Victim selection is discussed in Section 3.4. Stealing multiple tasks to reduce the frequency of load balancing operations is the focus of Section 3.5. Section 3.6 emphasizes the importance of software polling, without which long-running tasks would prevent workers from handling steal requests.

3.1 Channels

Channels for interprocess communication can be traced back to Tony Hoare's Communicating Sequential Processes (CSP), a formal notation for describing concurrent systems [118, 212]. CSP influenced the design of a number of programming languages, including Occam, Amber, Newsqueak, Alef, Limbo, and, most recently, Go [23]. Newsqueak and its descendants leading up to Go treat channels as typed, first-class values: they can be assigned to variables, passed as parameters, returned from functions, and sent over channels like any other value. Concurrent ML (CML), an extension of Standard ML, goes one step further and makes synchronous operations in-

[1] *"Prediction is very difficult, especially about the future."* —Niels Bohr

volving channels first-class, with the result that channel communication can be hidden behind user-defined protocols [207].

Channels, in the original sense of CSP, serve the dual purpose of communication *and* synchronization: CSP channels are unbuffered so that sender and receiver synchronize at the point of message exchange. While this behavior makes reasoning about concurrent programs easier, it is too restrictive for use in a performance-critical runtime system, where worker threads should not be hindered from making progress by waiting on channel operations to complete. Go [23] and Rust [24], for example, support buffered channels for asynchronous sends and receives, allowing messages to be queued and later received. Manticore and MultiMLTon, two multicore-aware variants of Standard ML, incorporate CML-style concurrency and add support for asynchronous messages and events [92, 223]. Barrelfish is a multikernel operating system that runs an independent kernel on every core of a multicore processor, using buffered channels for inter-core communication [42]. The authors of Barrelfish are careful to note that absence of shared state at the OS level does not preclude applications from sharing memory [42]. The same can be said for a work-stealing runtime system that prefers explicit communication. We will, however, see a few examples where it is useful to be able to share state between workers. Channel communication can be used to ensure that a worker has exclusive access to some data until ownership is given to another worker. This enables important optimizations, such as avoiding to copy heap-allocated objects on shared-memory systems.

3.1.1 Why Channels?

Channels are the building blocks of our choice, because channels

- provide a message passing abstraction that is easy to use, yet flexible enough to support the types of messages that need to be sent among worker threads,
- can be implemented efficiently on top of different communication mechanisms, including coherent and non-coherent shared memory and distributed memory,
- are amenable to optimization, such as compile-time specialization of send or receive operations, if the number of senders or receivers can be determined statically [206],
- appear as first-class constructs or standard library components in modern programming languages, such as Go and Rust, and most importantly,
- allow us to design task schedulers that are largely decoupled from low-level communication details.

These reasons are first and foremost practical reasons. After all, worker threads must be able to communicate efficiently, and efficiency demands lightweight abstraction. Channels allow us to replace concurrent deques with private data structures as simple as arrays or linked lists. Furthermore, channels can have specialized and more efficient operations if their usage deviates from the general pattern of many-to-many messaging. As an example, consider a channel that is used by N client threads to send messages to a server thread. If the server thread is the only one to receive messages from the channel, it is allowed to forego synchronization that would otherwise be needed to coordinate access if more threads wanted to receive a message.

Modern programming languages with a systems focus, such as Go and Rust, encourage programmers to "share memory by communicating", rather than to "communicate by sharing state" [97]. Channels let us apply this motto to runtime systems and libraries.

3.1.2 The Channel API

Channels behave like FIFO queues, and the runtime system assumes that messages are received in the order they were sent. We focus on buffered channels for asynchronous messaging. The channel API is small; the functions we use are summarized in the following list:

```
Channel *channel_alloc(size_t sz, unsigned int n, chan_t t);
```

Allocates and returns a channel of type `t` for elements $\leq$ `sz` bytes. Valid types are defined by the enumeration constants `MPMC`, `MPSC`, and `SPSC`, which describe common patterns of communication:

- `MPMC` (multiple producers, multiple consumers): denotes a channel that permits multiple threads to send and receive messages. This is the most general type of channel, suitable for many-to-many communication.
- `MPSC` (multiple producers, single consumer): allows multiple threads to send, but at most one thread to receive messages.
- `SPSC` (single producer, single consumer): limits communication to one sender and one receiver.

The parameter `n` defines the channel's capacity—the maximum number of elements that can be buffered internally. The interface suggests that all channels are *bounded*, that is, have a finite capacity. Unbounded channels with resizable buffers could be used in place of bounded channels, but are not needed for our purpose. As we shall see later, we can give an upper bound on the number of messages in the scheduler so that channels can be sized appropriately at allocation time to guard against the possibility of blocking sends or message loss.

```
void channel_free(Channel *ch);
```

Frees the memory associated with channel `ch`.

```
bool channel_send(Channel *ch, void *data, size_t sz);
```

Sends an element of `sz` bytes at address `data` to channel `ch`. Returns `false` if the channel is full, `true` otherwise.

```
bool channel_receive(Channel *ch, void *data, size_t sz);
```

Receives an element of `sz` bytes from channel `ch`. The element is stored at address `data`. Returns `false` if the channel is empty, `true` otherwise.

```
unsigned int channel_peek(Channel *ch);
```

Returns the number of buffered items in channel `ch`. This function checks for available messages without receiving them.

There is no distinction between the two endpoints of a channel, which means that every thread that holds a reference to a channel may use that reference to send and receive messages. In situations where the communication behavior can be analyzed or is known beforehand, specialized channels (MPSC, SPSC) may be used in place of more general ones (MPMC) to reduce overhead and improve performance [206].

3.1.3 Channel Implementation

Channels have a practical advantage over work-stealing deques: channels—and FIFO queues in general—are easier to implement than deques with concurrent *push*, *pop*, and *steal* operations. Deques require expensive synchronization operations that cannot be eliminated without relaxing the semantics of the work-stealing algorithm [169, 39] or assuming bounded store/load reordering [176]. Channels, on the other hand, can be implemented efficiently, especially under the assumption of limited concurrency [71, 39, 144]. As it turns out, single-consumer queues (MPSC, SPSC) suffice to construct efficient schedulers.

Shared Memory Listing 3.1 sketches a simple implementation of SPSC channels on a typical shared-memory multiprocessor [142, 116]. Channels are implemented as ring buffers with head and tail pointers (actually buffer indices). Elements are added to the tail and removed from the head, so that head lags behind tail. We prefer ring buffers to linked lists because channels need not be resizable; they always contain a bounded number of elements. The memory barriers in lines 21 and 38 ensure that writes to head and tail occur in program order. If left unconstrained, non-sequentially-consistent[2] architectures are free to reorder loads and stores in accordance with their memory models [177, 31], potentially violating the correctness of the algorithm [142].

Shared-memory channels are suitable for passing values as well as references. Consider the following example to send a large data structure over a channel:

```
Channel *chan = channel_alloc(sizeof(Large_data_structure), 1, SPSC);
Large_data_structure *d = ...
channel_send(chan, d, sizeof(Large_data_structure));
```

Instead of copying `sizeof(Large_data_structure)` bytes into and out of the channel, first, when sending, and second, when receiving, data can be moved between threads without copying. Assuming `d` is allocated from heap memory, a send can be used to transfer ownership of the referenced data:

```
Channel *chan = channel_alloc(sizeof(Large_data_structure *), 1, SPSC);
Large_data_structure *d = ...
channel_send(chan, (void *)&d, sizeof(Large_data_structure *));
```

Notice how the channel has changed from storing values to storing pointers. A thread must not access data for which it has relinquished ownership, in much the same way that data must not be touched after it has been freed.

Distributed Memory Channels can be used to exchange data among processes in a distributed environment. A possible implementation of SPSC channels is shown in Listing 3.2, where we take advantage of MPI's nonblocking send and receive operations instead of maintaining channel buffers ourselves. Point-to-point messages as in MPI enforce the property that only one process receives from a channel. Additionally, MPI's semantics guarantee that messages are received in the order they were sent (see [15], Section 3.5, pp. 42–45), provided the channel is used as intended[3]. Handing over a channel between two processes requires copying the channel descriptor and changing the value of `receiverID` to point to the new receiver. Each channel must tag its

[2]Sequential consistency [141] forbids observable reordering of memory operations.
[3]Using the same implementation for MPSC messaging would violate the FIFO property of channels!

```c
typedef struct channel {
    unsigned int cap;
    size_t itemsize;
    unsigned int head;
    // Appropriate padding to avoid false sharing ...
    unsigned int tail;
    char *buffer;
} Channel;

bool channel_send(Channel *chan, void *data, size_t size)
{
    unsigned int newtail;

    if (IS_FULL(chan))
        return false;

    assert(size <= chan->itemsize);
    memcpy(chan->buffer + chan->tail * chan->itemsize, data, size);

    newtail = INCREMENT(chan->tail, chan->cap);
    memory_barrier();
    chan->tail = newtail;

    return true;
}

bool channel_receive(Channel *chan, void *data, size_t size)
{
    unsigned int newhead;

    if (IS_EMPTY(chan))
        return false;

    assert(size <= chan->itemsize);
    memcpy(data, chan->buffer + chan->head * chan->itemsize, size);

    newhead = INCREMENT(chan->head, chan->cap);
    memory_barrier();
    chan->head = newhead;

    return true;
}
```

Listing 3.1: Implementation sketch of SPSC channels on a typical shared-memory multiprocessor. Elements are added to the tail and removed from the head of a circular array of bounded size. For performance reasons, it is important that head and tail occupy separate cache lines, or otherwise, sender and receiver end up constantly invalidating each other's cached values of head and tail (false sharing). Memory barriers (affecting both compiler and hardware) prevent reordering of prior loads and stores with updates of head and tail.

```
 1 typedef struct channel {
 2     int receiverID;
 3     int tag;
 4     size_t itemsize;
 5 } Channel;
 6
 7 bool channel_send(Channel *chan, void *data, size_t size)
 8 {
 9     MPI_Request req;
10
11     MPI_Isend(data, size, MPI_BYTE, chan->receiverID, chan->tag, MPI_COMM_WORLD, &req);
12     MPI_Wait(&req, MPI_STATUS_IGNORE);
13
14     return true;
15 }
16
17 bool channel_receive(Channel *chan, void *data, size_t size)
18 {
19     MPI_Request req;
20     int flag;
21
22     MPI_Iprobe(MPI_ANY_SOURCE, chan->tag, MPI_COMM_WORLD, &flag, MPI_STATUS_IGNORE);
23     if (flag) {
24         MPI_Irecv(data, size, MPI_BYTE, MPI_ANY_SOURCE, chan->tag, MPI_COMM_WORLD, &req);
25         MPI_Wait(&req, MPI_STATUS_IGNORE);
26     }
27
28     return (bool)flag;
29 }
```

Listing 3.2: Channels as thin wrappers around two-sided communication operations using the example of nonblocking send and receive in MPI. The `MPI_Wait` in line 12 waits until the data has been copied out of the send buffer; it does not necessarily mean that the message has been received. Similarly, the `MPI_Wait` in line 25 waits until the message has arrived in the receive buffer. The receiver first probes whether a message is available that matches the channel's tag before it initiates the receive operation.

messages with a unique identifier so that receivers are able to distinguish messages belonging to different channels.

Channels may be thin wrappers around message passing primitives, as we have seen in Listing 3.2. Lower-level implementations may utilize remote memory access (RMA) to send and receive messages. As part of our work in [201], we have implemented channels based on one-sided put and get operations between the local message passing buffers on Intel's SCC processor. Figure 3.1 shows the communication latencies for bouncing a small, cache-line-sized message between a pair of cores. This "ping-pong" benchmark measured the round-trip latency between core number 0, at the bottom left corner of the chip, and a second core that varied from being 0 to 8 hops away. General MPMC channels added 23–41% overhead on top of the native communication library's send and receive operations (RCCE). MPSC and lock-free SPSC channels, however, allowed faster communication than RCCE, showing the benefit of specializing an implementation to the communication pattern [206].

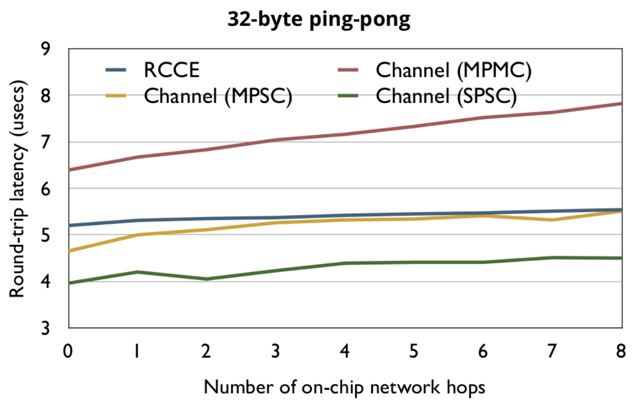


Figure 3.1: Round-trip latencies in microseconds on the Intel SCC for passing a 32-byte message back and forth between core 0 and a second core that varies from being 0 to 8 hops away. A distance of 0 corresponds to core 0 communicating with core 1 of the same tile, a distance of 8 corresponds to core 0 communicating with core 47. The results show the fastest of ten trials, each trial being the average of 1000 round trips. The chip was operated in its default Tile533/Mesh800/DDR800 configuration (all values in MHz).

Distributed-memory implementations of channels have copy semantics, regardless if a machine supports shared memory. Send and receive operations are required to copy messages between private memory and channel buffers. Friedley et al. propose an API for ownership passing that enables move semantics[4] in MPI programs [94]. Combined with other recent developments, such as the extended RMA model of MPI-3 [119] and improved producer-consumer communication [43], MPI may permit increasingly efficient channel implementations.

3.2 Scheduler Overview

When we introduced the task model in Section 2.4 of the previous chapter, we mentioned that `TASKING_INIT` creates a number of worker threads that wait for tasks to become available. The master thread is responsible for supplying initial tasks. Every thread, including the master thread, maintains a private deque of tasks, which represents pending work. Figure 3.2 depicts the main scheduling loop.

Workers repeatedly pop tasks from the bottom of their private deques ① until they have no work left to do. Whenever a worker i creates a task t, it pushes t onto the bottom of its private deque (not shown in Figure 3.2) so that t will be popped next

[4]We borrow the term "move semantics" from C++11, where it implies that resources may be moved between scopes without copying.

```
SCHEDULE()
    Let $Q_i$ be the private deque of tasks of worker $i$,
        $C_i$ be the channel for sending steal requests to worker $i$,
        $T_i$ be the channel for sending tasks to worker $i$
    while TRUE
        while $Q_i$ is not empty
            Pop task $t$ from the bottom of $Q_i$                ①
            while $C_i$ is not empty
                Receive and handle steal request                ②
            Run task $t$
        Select a worker $j$, $j \neq i$, at random              ③
        Send a steal request to channel $C_j$
        while $T_i$ is empty
            if $C_i$ is not empty
                Receive and handle steal request                ④
            if master has signaled termination
                return
        Receive task $t$ from channel $T_i$                     ⑤
        Run task $t$
```

Figure 3.2: The main scheduling loop that every worker thread keeps executing until the master thread signals termination.

```c
typedef struct steal_request {
    int thief;
    Channel *chan;
    // Possibly other fields
} StealRequest;
```

Listing 3.3: A minimal steal request message contains the thief's ID (`thief`) and a reference to a channel for sending tasks (`chan`).

unless a steal causes t to be sent to another worker j. Steal requests are handled after every push and after every pop ②. A successful steal requires worker i to remove a task from the top of its private deque and to send the "stolen" task to the thief. A failed steal implies that worker i has no tasks left, in which case stealing continues after selecting a new victim. How this works is described in detail in the following section.

Figure 3.2 shows an instance of a *parsimonious* work-stealing algorithm [228]: a worker i sends a steal request to another worker j, $j \neq i$, only when its private deque is empty ③. But steal requests are more flexible than that: a worker may initiate a steal at any time by sending a message (asynchronously) and continue where it left off, without waiting for a response. Whether sent before or after running out of work, steal requests may incur a delay that cannot be hidden, during which a worker has nothing else to do but to service incoming messages ④. A successful steal ends with the receipt of a task ⑤. Running this task may result in new tasks being created and pushed to the bottom of the private deque, completing an iteration of the scheduling loop.

3.3 Steal Requests

Stealing tasks without being able to access other workers' deques requires cooperation between victims and thieves. When a worker runs out of tasks, it becomes a thief by sending steal requests to selected victim workers, which either reply with tasks or signal that they have no tasks left. A steal request is a message containing the thief's ID, a reference to a channel for sending tasks from victim to thief, and possibly other fields carrying additional information, as shown in Listing 3.3.

When the runtime system starts up, every worker allocates two channels: a channel for receiving steal requests and a channel for receiving tasks. A reference to the latter is stored in steal requests, and workers use this reference to send tasks. By "owning" two channels, workers are able to receive steal requests and tasks independently of other workers, which in turn enables efficient channel implementations based on single-consumer queues [39]. The total number of channels grows linearly with the number of workers: n workers allocate $2n$ channels to communicate with each other.

3.3.1 Number of Steal Requests

When we introduced the channel API earlier in this chapter, we said that all channels have bounded capacity. This, of course, requires that the number of messages is bounded as well, given that workers should never block trying to send messages to full channels. Matching traditional work stealing, we allow one outstanding steal request per worker. This decision has two important consequences: (1) The number of steal requests is bounded by n, the number of workers. (2) A thief will never receive tasks from more than one victim at a time. It follows from (1) that a channel capacity of $n - 1$ is sufficient to deal with other workers' steal requests since no more than $n - 1$ thieves may request tasks from a single victim. We actually increase the capacity to n so that steal requests can be returned to their senders, for instance, in case of repeated failure. (2) implies that, at any given time, a task channel has at most one sender and one receiver, meeting the requirements for an SPSC implementation.

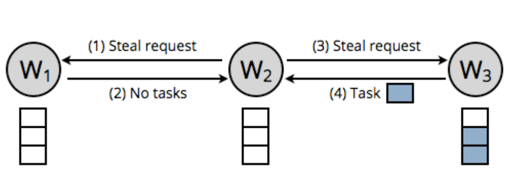
Suppose all n workers have issued steal requests, and one of them, worker i, starts to create tasks. Let m be the number of tasks that are sent in response to a successful steal request. Worker i will handle at most $n-1$ steal requests by sending $(n-1)\cdot m$ tasks. It may or may not be able to discard its own steal request before sending the last of those tasks. The total number of messages is therefore bounded by $(n-1)\cdot m+1 = n\cdot m-m + 1$. Assuming m is constant[5], communication grows linearly with the number of workers. The amount and frequency of communication is a major factor in determining the work stealing overhead.

3.3.2 Handling Steal Requests

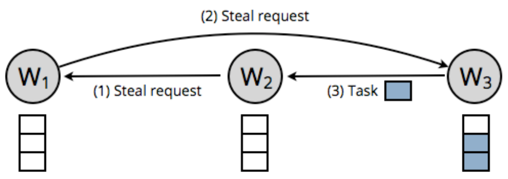
When a worker has no tasks left to send in response to a steal request, it must react in some way to make sure that stealing can continue. In other message-passing schedulers, every steal request is acknowledged to inform a thief about the outcome of a steal [75, 204]. A positive acknowledgment message is followed up with tasks, a negative acknowledgment message prompts the thief to select another victim and try again.

It seems natural to acknowledge steal requests, but the problem with this approach is twofold: First, steals that succeed after t tries involve $2t$ messages, t steal requests plus t acknowledgments. Ideally, t tries should involve no more than t messages. Second, workers should respond to every message, including every acknowledgment, as promptly as possible, because otherwise, stealing comes to a halt. This makes it difficult to overlap stealing with other work, unless workers regularly check for incoming

[5]We will later describe an implementation that has $m = 1$.



(a) Acknowledging every steal request



(b) Forwarding failed steal requests

Figure 3.3: Possible message flows for steal requests. With (a), every attempt at stealing involves two messages: a request and an answer, either negative (no task) or positive (task). We implement (b), which omits acknowledgment messages. Steal requests are forwarded until tasks are found. Effectively, victims assume the role of thieves and send steal requests on their behalf. Steal requests and tasks are sent over separate channels.

steal requests *and* acknowledgments.

Our solution is to eliminate acknowledgment messages altogether by having victims *forward* steal requests they cannot handle themselves. In other words, victims resend steal requests on behalf of thieves if necessary, as if they intended to steal. Forwarding a steal request, however, does not mean that the steal request is "hijacked" as it still points to the original thief.

Figure 3.3 illustrates our approach. Suppose worker W_1 receives a steal request from worker W_2, but has nothing left to share. Rather than return a message to W_2, saying the steal has failed (Figure 3.3 (a)), W_1 forwards the steal request to another potential victim, worker W_3 (Figure 3.3 (b)). Because W_3 has tasks to spare, it will pass some of its work on to W_2 using the channel contained in the steal request. Otherwise, if the steal failed again, W_3 would select another victim and forward the steal request, or, alternatively, if stealing is unlikely to succeed, return the steal request to W_2, which might choose to back off from stealing and try again at a later time.

The forwarding of steal requests makes it easier for workers to start stealing before they strictly need to. For example, a worker may initiate a steal by sending a request after popping the last task from its deque (and before running the task). Once initiated, the steal is carried out by the victim, or other victims after that. Ideally, when the worker finishes its last task and runs out of work, new work has already arrived and can be picked up immediately. Sending a steal request becomes an asynchronous operation that, like a future, can be waited for when its result, a task, is needed to continue execution. Stealing ahead of time can mask communication latency and reduce the time spent waiting to receive new work. In the absence of acknowledgment messages, the time between initiating a steal and receiving tasks depends primarily on the victims' responsiveness—their ability to handle steal requests in a timely manner.

```
HandleStealRequest() // First version
    Let $Q_i$ be the private deque of tasks of worker $i$,
        $C_i$ be the channel for sending steal requests to worker $i$,
        $S$ be the steal request to handle
1   if $Q_i$ is not empty
2       Pop task $t$ from the top of $Q_i$
3       Send task $t$ to channel $S.chan$
4   else
5       Select a worker $j$, $j \neq i \wedge j \neq S.thief$, at random
6       Send $S$ to channel $C_j$
```

Figure 3.4: When a worker receives a steal request but cannot send a task in return, it selects another worker to which it forwards the steal request.

Figure 3.4 summarizes how workers respond to steal requests. Tasks are popped, oldest first, from the top of the local deque and sent to channel $S.chan$, which belongs to worker $S.thief$. If worker i's deque is empty and the steal request must be rejected, worker i picks a new victim to which it forwards the steal request, leaving $S.chan$ and $S.thief$ unchanged.

Recall the workers' scheduling loop in Figure 3.2. Having sent a steal request, a worker that becomes idle may have to wait until the steal succeeds. While waiting for tasks to arrive, the worker keeps forwarding steal requests from other workers because it has no tasks left. Implementation-wise, there is no difference between initiating a steal (lines 7–8 of Figure 3.2) and forwarding a steal request (lines 5–6 and 10–11 of Figure 3.4) apart from the set of potential victims and the contents of the steal request, which identify the initial sender as the actual thief who will receive the stolen tasks, if any are found. Thus, by forwarding steal requests, a worker is actively stealing, not for itself but on behalf of other workers. Seen from this angle, the workers' scheduling loop follows conventional work stealing: workers execute local tasks and try to steal after running out of work.

Forwarding steal requests appears both simpler and more efficient than sending acknowledgment messages. We would expect a measurable difference in performance when work stealing happens frequently and latency matters. Figure 3.5 shows the result for a sample BPC workload with fine-grained tasks. Forwarding yields in fact between 17% and 33% better overall performance because of increased work-stealing efficiency. Even with empty tasks and no useful work to do, close to 80% of all steal requests result in tasks being sent to thieves.

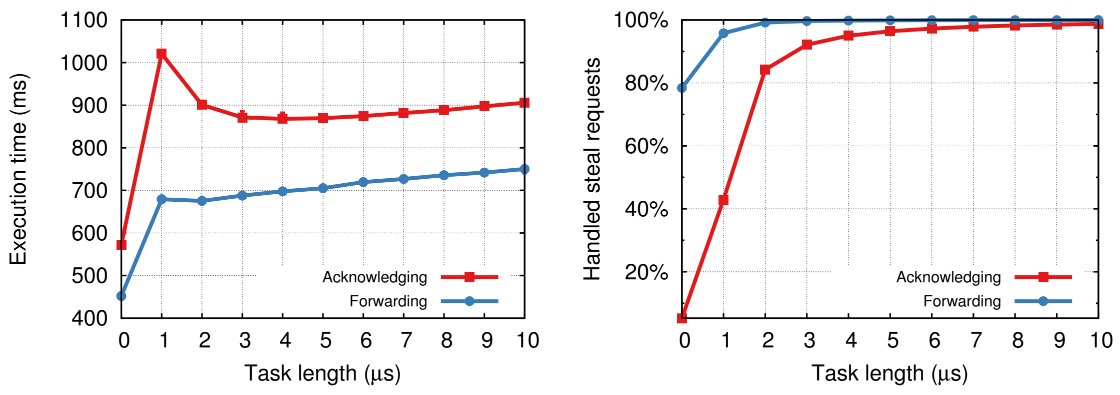


Figure 3.5: Acknowledging failed steal attempts versus forwarding steal requests. The two figures examine the results of running BPC with $d = 100\,000$, $n = 9$, and t between 0 and 10 microseconds. The figure on the right shows the percentage of steal requests that were answered with tasks. (GCC 4.7.1, `-O3`, AMD Opteron multiprocessor, 48 worker threads)

3.4 Victim Selection

Before a worker can send a steal request, it must pick a victim. Cilk has proved the efficiency of work stealing by selecting victims uniformly at random, and other schedulers have largely followed suit. On large-scale systems where it becomes impractical to try to steal from every potential victim, work stealing is usually restricted to selecting victims from predefined sets [234] or is guided by locality information to favor nearby victims [170].

3.4.1 Random Victim Selection

Random victim selection makes it easy to steal on behalf of other workers. A thief can, for example, use its steal request to pass along a copy of its random number state to have victims generate the same random sequence. Notice the close correspondence between line 7 of Figure 3.2 and line 5 of Figure 3.4. We have looked at deterministic strategies, such as selecting victims round robin based on rank or worker ID, but our experiments have led us to the conclusion that the robustness of a randomized strategy is hard to rival. Recent work by Perarnau et al. studying the UTS benchmark confirms that performance increases significantly by replacing deterministic with random victim selection [194]. Prior to that, Faxén et al. proposed alternative strategies to random victim selection, but found that neither sampling a number of victims nor constraining the set of victims to steal from made a noticeable difference in performance in their tests with up to 64 workers [88].

There is, however, a potential problem that we must consider when randomly selecting victims: the latency due to failed steals. With concurrent deques, a thief can probe a victim's deque before committing a steal. If a deque turns out to be empty, the thief proceeds to the next victim. To some extent, this is like aborting a steal that is bound to fail: an optimization to compensate for the lack of direction in choosing victims. Private deques permit no such optimization in the absence of shared state. Every steal that fails generates another message that must be sent, received, and handled, adding up to the time it takes to find work. Even stealing ahead of time might not suffice to cover the latency that results from a large number of failed steals. It is therefore important to avoid unnecessary communication to the best extent possible.

Suppose only one worker has tasks left. Under random victim selection, the probability of picking this worker among n potential victims is $\frac{1}{n}$. The probability of picking this worker after k unsuccessful attempts is $(1 - \frac{1}{n})^k \cdot \frac{1}{n}$, $k \in \{0, 1, 2, \ldots\}$. The number of unsuccessful attempts follows a geometric distribution with probability of success $p = \frac{1}{n}$ and expected value $n - 1$.

Random victim selection can be improved by removing victims after unsuccessful attempts. Failing once increases the probability of success to $\frac{1}{n-1}$. Failing twice increases the probability of success to $\frac{1}{n-2}$, and so on. In the best case, it takes a single attempt to find the right victim. In the worst case, it takes n attempts to sample every victim. On average, it takes $\frac{1+2+\cdots+n}{n} = \frac{n+1}{2}$ attempts, which include $\frac{n-1}{2}$ failures. Saving $(n - 1) - \frac{n-1}{2} = \frac{n-1}{2}$ attempts can make a difference in practice, but does not change the fact that the expected number of failures increases linearly with the number of potential victims.

3.4.2 Remembering the Last Victim

Figure 3.6 shows the inefficiency of random victim selection on the example of the matrix multiplication benchmark in which a single worker creates all tasks. Instead of choosing victims independently at random, we avoid choosing a victim more than once, as described above. This can be done in two ways:

The first is to store the set of previously selected victims in the steal request and update it in case of failure. If there are many potential victims, it may be more practical to focus on the m, $m < n$, most recently selected victims, rather than trying to fit n worker IDs into a steal request[6].

The second is to take advantage of shared memory. Every worker i keeps a list of potential victims, which it starts to shuffle to pick the first victim [80]. If the steal

[6] A more compact representation such as a bitset could help.

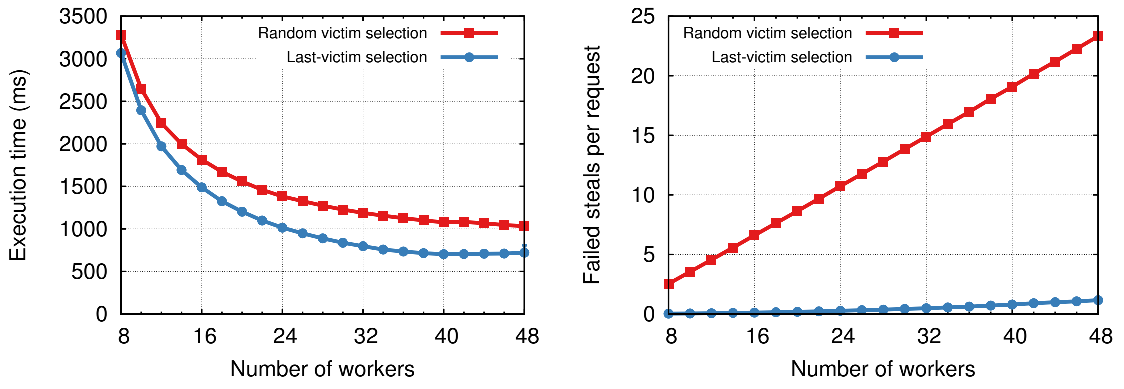


Figure 3.6: Random victim selection may not be the best strategy when a single worker creates all tasks, as in this example of multiplying two 2048 × 2048 matrices using blocks of size 32 × 32. The figure on the right shows the numbers of failed attempts before a steal request succeeded (medians of averages from ten program runs). (GCC 4.9.1, `-O3`, AMD Opteron multiprocessor)

request fails, the first victim, which now assumes the role of thief, continues the shuffling of worker i's list to pick the second victim, and so on, until a single victim remains. A pointer to this (partially shuffled) list of victims can be passed along with each steal request, sharing state by communicating [97].

Figure 3.6 (b) confirms that, on average, a steal request succeeds after $\frac{n-1}{2}$ failures. It seems pointless to ask the same workers over and over again if only one of them can possibly send tasks. We can devise a simple strategy without necessarily knowing which worker we are looking for: remember the victim of the last successful steal, and target this victim first when running out of tasks next time. If this last-victim check fails to have the desired effect because the victim in question has run out of tasks in the meantime, victim selection proceeds in the same way as previously. The result is up to 35% better performance compared to completely random selection, as shown in Figure 3.6 (a).

Is it always preferable to send steal requests to the last victim if we know that other workers will decline? The answer is, perhaps surprisingly, negative. The following analysis assumes that out of n potential victims ($n+1$ workers), only one has tasks that workers are trying to steal, and that, under random victim selection, a steal request is expected to succeed after $\frac{n-1}{2}$ failures.

Let t_{sendR} be the time it takes to send a steal request and t_{sendT} be the time it takes to send a task. In addition, let t_{recvR} be the time it takes to receive a steal request. We further define t_{lat}, t_{sel}, and t_{steal} to be the message handling latency, the time it takes to select a victim, and the time it takes to steal (dequeue) a task, respectively. The time for a failed steal that results in a forwarded request can be broken down as $t_{fail} = t_{lat} + t_{recR} + t_{sel} + t_{sendR}$, while a successful steal amounts to $t_{succ} = t_{lat} + t_{recR} + t_{steal} + t_{sendT}$. Randomly selecting victims, the time it takes to receive a task becomes

$$\begin{aligned} t_{ws}^{R} &= t_{sel} + t_{sendR} + \frac{n-1}{2} \cdot t_{fail} + t_{succ} \\ &= \frac{n+1}{2} \cdot (t_{sendR} + t_{lat} + t_{recR} + t_{sel}) + t_{steal} + t_{sendT}. \end{aligned} \tag{3.1}$$

By timing the different operations, we observed that $t_{sel} \approx t_{steal}$, whereas communication is an order of magnitude more expensive. (Recall that neither t_{sel} nor t_{steal} involves synchronization.) To simplify, we drop t_{sel} and t_{steal} and say that

$$t_{ws}^{R} \approx \frac{n+1}{2} \cdot (t_{sendR} + t_{lat} + t_{recR}) + t_{sendT}. \tag{3.2}$$

In other words, work stealing is dominated by the cost of communication, including message handling latencies.

Now consider that thieves may prevent further failure (assuming enough tasks are available) by sending steal requests to the victim from which they received their last tasks. When n steal requests are lined up (worst case), and all orderings are equally likely, the expected time it takes to receive a task is

$$t_{ws}^{LV} = t_{sel} + c \cdot t_{sendR} + t_{lat} + \frac{n+1}{2} \cdot (t_{recR} + t_{steal} + t_{sendT}), \tag{3.3}$$

with $c \geq 1$ accounting for the possibility that sending steal requests to a single victim may increase contention among thieves, and t_{lat} being amortized over $\frac{n+1}{2}$ steal requests, which can be handled in succession. We omit t_{sel} and t_{steal} just like we did above and conclude that

$$t_{ws}^{LV} \approx c \cdot t_{sendR} + t_{lat} + \frac{n+1}{2} \cdot (t_{recR} + t_{sendT}). \tag{3.4}$$

Last-victim selection may not be able to reduce the number of messages, but it increases the efficiency with which steal requests are handled. Random victim selection is sensitive to variations in t_{lat}. Workers that are busy running tasks cannot respond to steal requests, causing latency to increase, which in turn increases linearly with the number of workers. On the other hand, workers that are idle have nothing to do besides handling messages, in which case latency becomes negligible. If tasks are so short that

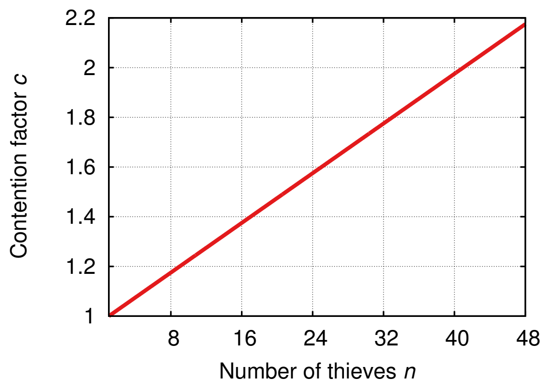


Figure 3.7: When steal requests cause contention among thieves, the overhead of last-victim selection may exceed that of random victim selection if the time required to send a steal request increases by more than a factor of c. The graph shows how c increases with the number of concurrent thieves, assuming that steal requests are 5% more expensive to send than tasks. (Steal requests use MPSC channels, whereas tasks use faster SPSC channels.)

t_{lat} is of no significance, random victim selection incurs a communication overhead of $\frac{n+1}{2} \cdot (t_{sendR} + t_{recR}) + t_{sendT}$, compared to $c \cdot t_{sendR} + \frac{n+1}{2} \cdot (t_{recR} + t_{sendT})$ for last-victim selection. Assuming ideal channels, that is, assuming $c = 1$ and $t_{sendR} = t_{sendT}$, both strategies have the same communication cost. But more realistically, we expect $c > 1$ and possibly $t_{sendR} \geq t_{sendT}$ because steal requests are sent over MPSC channels, whereas tasks are sent over SPSC channels. Therefore, $c \cdot t_{sendR} + \frac{n+1}{2} \cdot t_{sendT} > \frac{n+1}{2} \cdot t_{sendR} + t_{sendT}$ leads to last-victim selection having higher communication cost than random victim selection. Under the assumption of observable contention ($c > 1$), $t_{sendT} \geq t_{sendR}$ or $t_{sendR} > t_{sendT}$ and $c > \frac{(n+1)/2 \cdot (t_{sendR} - t_{sendT}) + t_{sendT}}{t_{sendR}}$ suffice for last-victim selection to be outperformed by random victim selection.

As an example for the latter case, suppose that sending a steal request takes 5% longer than sending a task. Figure 3.7 plots the values of c above which last-victim selection would incur more overhead in terms of channel operations. Judging from these numbers, it is entirely possible that, for sufficiently short tasks, random victim selection provides better load balancing, despite the number of failed attempts caused by sending steal requests to random workers.

The results of testing our hypothesis on the SPC benchmark are shown in Figure 3.8. Up to a task length of roughly 25 microseconds, random victim selection is preferable to last-victim selection because it achieves a better distribution of work, as measured by the number of tasks assigned to each consumer. For longer tasks and thus longer message handling latencies, the opposite is true, with last-victim selection providing better load balancing and performance. Interestingly, last-victim selection is fast for

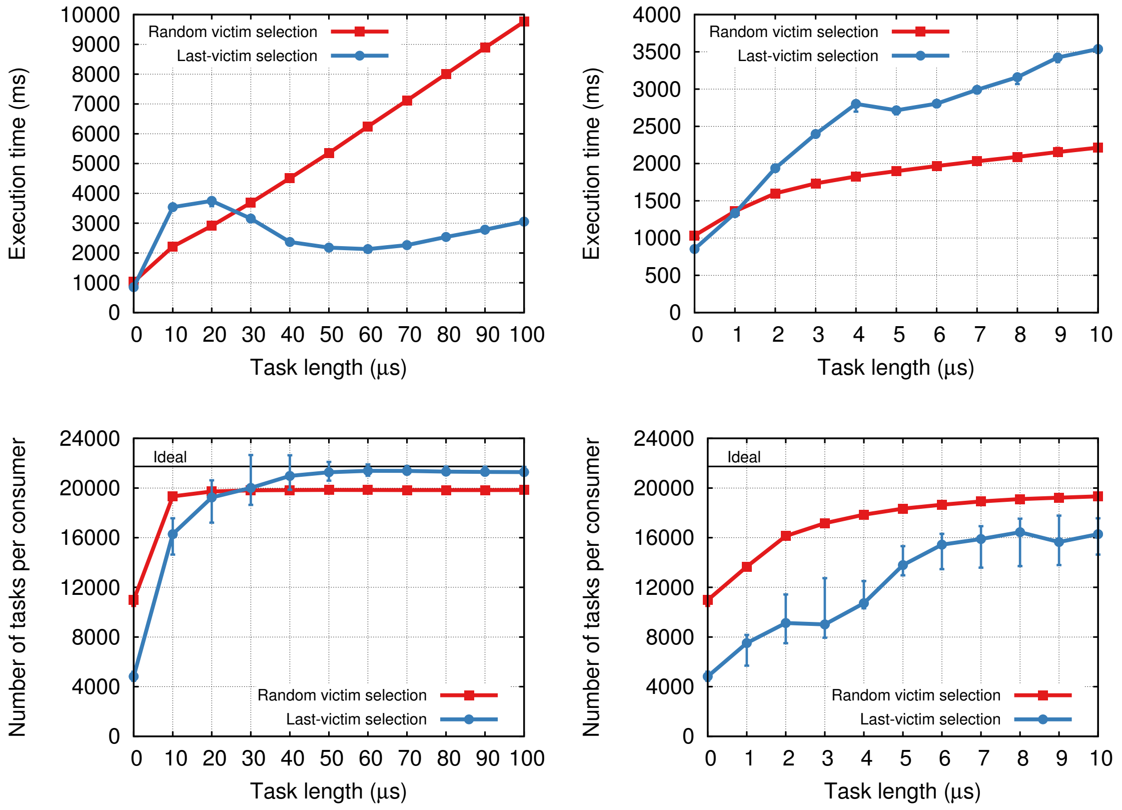


Figure 3.8: In a single-producer and multiple-consumers setting, as in this example of running SPC with $n = 10^6$ and t between 0 and 100 microseconds, last-victim selection leads to a poor distribution of work when scheduling fine-grained tasks of up to roughly 25 microseconds. Above that task length, however, it achieves a better distribution of work than random victim selection. The bottom figures show the numbers of tasks executed per consumer (medians of 460 data points, along with 10th and 90th percentiles). A horizontal line labeled "Ideal" indicates a perfectly even distribution of work. (GCC 4.9.1, `-O3`, AMD Opteron multiprocessor, 48 worker threads)

empty tasks because it fails to distribute many of them. Empty tasks are a special case for which load balancing is guaranteed to be detrimental to performance; there is simply no parallelism to take advantage of.

3.4.3 Limitations

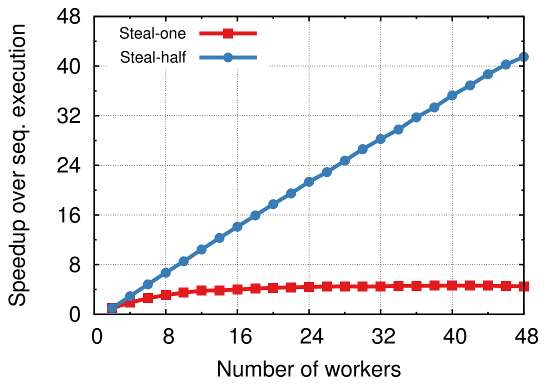
Random victim selection tends to work well when the load is fairly balanced, that is, when most workers can send tasks in response to steal requests. When workers frequently run out of work, however, last-victim selection may reduce the number of failed steals, *unless* the last victim frequently runs out of work too. If that happens to be the case, last-victim selection yields no improvement over random victim selection.

The problem is that, without further communication, workers have no way to know whether a potential victim has tasks or not. Probing a victim's deque to make sure it is non-empty before deciding to send a steal request requires support for shared memory and atomic operations to prevent data races. Another potential optimization with the same goal is to allow workers to close their channels (temporarily) while they are idle. Closed channels could then be skipped when searching for victims. Unlike in Go, where a closed channel indicates that no more values will be sent, a worker would signal that no more steal requests will be received and handled until the channel is reopened. Again, this is easy to implement if shared memory is available, in which case determining whether a channel is closed amounts to the same as determining whether a deque contains tasks: both approaches involve reading shared state to distinguish idle from busy workers. Idle workers are not selected as victims.

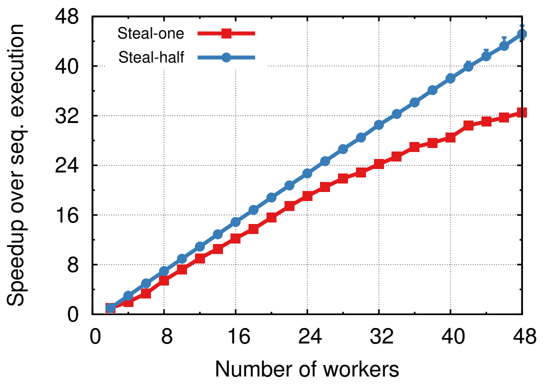
Sanchez et al. proposed a work-stealing scheduler in which managers oversee groups of workers [215]. Receiving updates from workers, managers keep approximate task counts and initiate steals based on this information. Update messages serve the purpose of notifying managers about changes in the task distribution as well as triggering steal requests, which are sent by managers whenever a worker appears to be running out of tasks. Steal requests are sent to the workers with the most tasks to minimize the chance of failure. It is unclear, however, to which extent this scheduler is able to scale without the supporting hardware primitives presented in the paper.

3.5 Importance of Steal-Half for Fine-grained Parallelism

Different victim selection strategies may affect the overhead associated with stealing a task, but may not prevent frequent stealing in unbalanced computations. The cycle of stealing a task, executing it, and running out of work again becomes increasingly



(a) SPC with $n = 10^6$ and $t = 10\mu s$



(b) SPC with $n = 10^6$ and $t = 100\mu s$

Figure 3.9: Being able to reduce the work-stealing overhead is essential for scheduling fine-grained parallelism. Informed by the results of Section 3.4, we combined steal-one with random victim selection in (a) and last-victim selection in (b). (GCC 4.9.1, `-O3`, AMD Opteron multiprocessor)

inefficient with decreasing task lengths.

Suppose m workers are trying to balance n tasks. Let t be the time it takes to run a task (task length) and t_{ws} be the time it takes to steal a task. (Recall Equations (3.2) and (3.4).) If $t \gg t_{ws}$, that is, if $t + t_{ws} \approx t$, the speedup of the computation will approach the number of workers m. Using a concrete example, ten workers are able to execute 100 tasks, each taking one second, in ten seconds. If $t \approx t_{ws}$ such that $t + t_{ws} \approx 2t$, the speedup will not exceed $m/2$. Generally speaking, the speedup is bounded by $\frac{n \cdot t}{(n \cdot (t+t_{ws}))/m}$, which simplifies to $\frac{t \cdot m}{t+t_{ws}}$. We expect no speedup if t is approximately equal to $\frac{t_{ws}}{m-1}$. If t falls below $\frac{t_{ws}}{m-1}$, the parallel computation will end up being slower than the sequential one; the speedup will turn into a slowdown. Theoretically, if t were so small that $t + t_{ws} \approx t_{ws}$, the speedup would tend towards zero.

3.5.1 Stealing Single Tasks

Figure 3.9 uses the SPC benchmark as an example to show the practical limitations of stealing single tasks, a strategy we call steal-one. Judging from the speedup curves, it takes more than $10\mu s$ to receive a task in Figure 3.9 (a) and less than $100\mu s$ in Figure 3.9 (b). We can estimate t_{ws} using the speedup formula above, but must be aware of the underlying assumption that n tasks are distributed evenly among m workers, which may not hold true in practice. Solving $S = \frac{t \cdot m}{t+t_{ws}}$ for t_{ws} gives $t_{ws} = \frac{t \cdot m}{S} - t$. With 48 threads, one of them managing termination detection, we measure speedups of 4.5 and 32.5 for $t = 10\mu s$ and $t = 100\mu s$, respectively. Substituting the values of S, t, and m, we get $t_{ws} = t_{ws}^{R} = 94.4\mu$s for $t = 10\mu$s and $t_{ws} = t_{ws}^{LV} = 44.6\mu$s for $t = 100\mu$s. (We use different victim selection strategies, see Figure 3.9.) Though difficult to determine precisely, we measure a median latency of 121μs for $t = 10\mu$s and 44μs for $t = 100\mu$s. (The measured values show a high variability; the 50% central ranges are 142μs for $t = 10\mu$s and 49μs for $t = 100\mu$s.) One value is very close to the estimate, the other value is off by 28%. In fact, $t = 10\mu$s yields an uneven distribution of work, unlike $t = 100\mu$s, as hinted at in Figure 3.8. Worker 0, which creates all tasks, also runs many more tasks than other workers. The actual overhead associated with work stealing is thus higher than the estimate, which does not account for load imbalance.

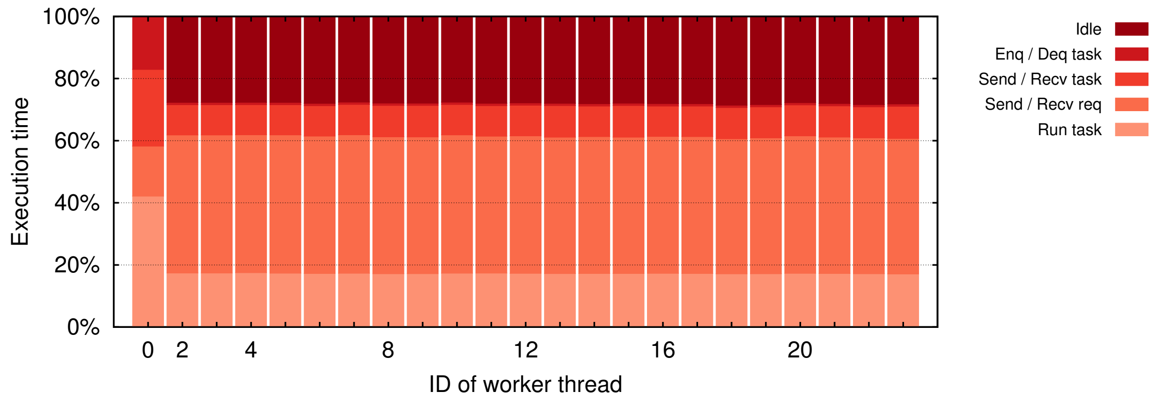
We should be convinced by now that steal-one is inadequate for scheduling fine-grained parallelism, unless a few steals suffice to maintain load balance. Reducing the cost of work stealing also means reducing the frequency with which workers need to steal. The idea is simple: workers may try to steal multiple tasks at a time to amortize the cost of stealing over n, $n > 1$, tasks that can be executed in sequence. Stealing multiple tasks helps spread the work, especially when dealing with large numbers of tasks. With more workers having tasks to spare, subsequent steals are more likely to succeed, even if victims are selected randomly.

3.5.2 Stealing Multiple Tasks

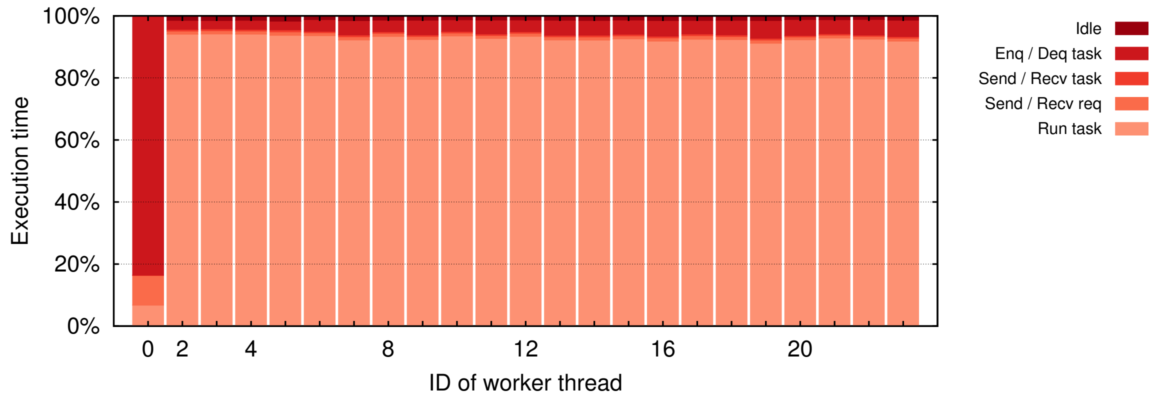
There are static and dynamic approaches to stealing multiple tasks. A possible strategy is to steal a fixed amount of work, say n tasks, and fall back to steal-one when a victim has less than n tasks left. Such a strategy, though easy to implement, becomes difficult to use when the workload is unknown or changes at runtime. Finding a value of n that is neither too small nor too large requires experimentation.

Dynamic strategies solve this problem by choosing n based on the number of tasks m in a victim's deque such that $n = f(m)$, where f is a monotonically non-decreasing function [44]. The more tasks are there for thieves to steal, the larger the value of n will tend to be. In this way, thieves are able to adapt to the workload. Among the different possibilities, stealing half of a victim's tasks, that is, $f(m) = \lfloor \frac{m}{2} \rfloor$, has emerged as a robust strategy [111, 44]. The idea behind steal-half is to transfer half of a victim's remaining work with a single steal, assuming a correlation between the number of tasks and the amount of work to do.

The results of using steal-half are included in Figure 3.9. Efficiency is up from 10% in (a) and 69% in (b) to 88% in (a) and 96% in (b). To understand where the large difference in performance comes from, Figure 3.10 shows execution profiles. Execution time is broken down into the time spent running tasks, enqueuing/dequeuing tasks,



(a) steal-one



(b) steal-half

Figure 3.10: Execution time profile of SPC with $n = 10^6$ and $t = 10\mu s$ under (a) steal-one and (b) steal-half work stealing. Worker 1 was dedicated to termination detection and is excluded from the list of worker IDs. The different sections of code were timed by inserting `RDTSC` instructions to read the processors' time-stamp counters (see [125], Chapter 17). This slowed down execution by 10% and 2% compared to the uninstrumented versions of steal-one and steal-half. (GCC 4.9.1, `-O3`, AMD Opteron multiprocessor, 24 worker threads)

sending/receiving tasks, sending/receiving steal requests, and idling. The time spent for work stealing, directly or on behalf of other workers by means of forwarding steal requests, is derived from handling steal requests and tasks. A worker is idle while waiting to receive tasks.

Under steal-one, worker 0 does more work than necessary, executing over twice as many tasks as other workers, which spend less than 20% of their time on useful work. By contrast, under steal-half, workers get to run a lot more tasks as the cost of scheduling and work stealing shrinks to 5–8%. Apart from worker 0, which is busy creating and distributing tasks, workers spend more than 90% of their time running tasks. This vastly better utilization of workers explains the significant speedup of steal-half over steal-one.

3.5.3 Implementing Steal-Half with Private Deques

Private deques have the benefit that stealing multiple tasks is a relatively straightforward extension. The same cannot be said for concurrent deques, which must resort to coarse-grained locking or implement intricate synchronization protocols [111].

In our implementation, every worker has a doubly-linked list that serves as a private deque, with *push* and *pop* operating on one end, and *steal* operating on the other end. Steal-half requires a worker to traverse the list, find the middle element, split the list in half, and send the second half to the thief using the channel reference in the steal request. Sending tasks sequentially, one by one, incurs overhead proportional to the length of the list. In addition, it is no longer possible to guarantee that `channel_send` never blocks as victims may try to send more tasks than a channel can buffer.

Shared memory permits an efficient solution with constant overhead: a single pointer-sized message can move a list of tasks between workers without copying. Stealing proceeds as follows: The victim splits its deque in half, intending to give away one half to the thief. Splitting a deque with head h_v creates a new deque with head h_t whose ownership is transferred by sending h_t (a pointer) to the thief. This requires that `channel_send` and `channel_receive` copy pointers rather than tasks by using the technique outlined in Section 3.1.3. Upon completing the steal, the victim's local pointer goes out of scope, permitting only the thief to access tasks through h_t. The thief receives h_t, prepends the list of stolen tasks to its own deque, and continues working. While copying tasks is inevitable in the absence of shared memory, similar optimizations may be used to limit the number of messages in a distributed environment, provided that tasks are stored in a way to facilitate one-sided access [185].

A doubly-linked list is not the most efficient data structure when it comes to splitting [29], but in practice, workers rarely pile up huge numbers of tasks while other workers remain idle. Tasks need not be stored in lists; they can also be stored in trees, which permit asymptotically efficient implementations of steal-half at the expense of adding overhead to task insertion and removal [122]. Using forests of binomial trees, for example, would enable thieves to steal the largest trees in the forests. Because task queues are private, no synchronization is needed, no matter how complex the underlying data structures are.

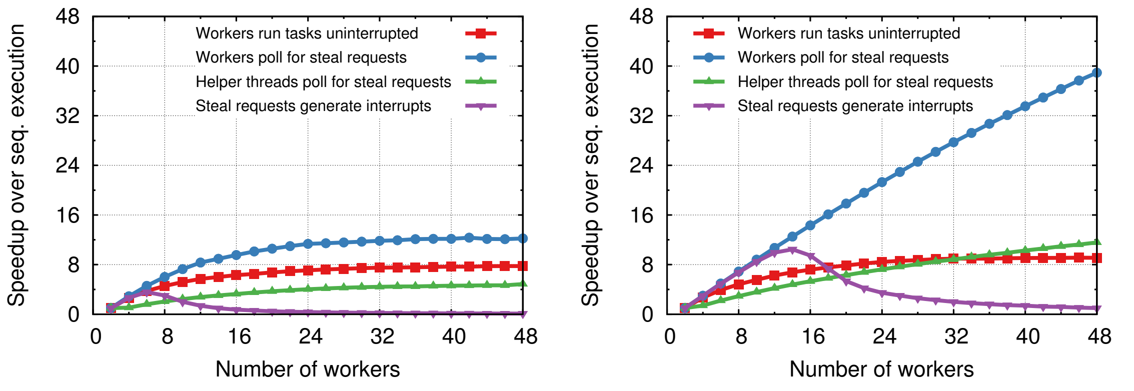
An alternative to arranging tasks in trees is building up nested lists. Cong et al. insert tasks into lists before making them available to thieves [67]. This task creation strategy saves deque operations and supports efficient stealing, but does not expose parallelism while a batch of tasks is being created. A batch is unsplittable and must be executed sequentially. The challenge is to schedule batches that are neither too small nor too large for the problem at hand. Cong et al. use a bounded exponential growth function to generate small batches when tasks are needed for load balancing and large batches when enough tasks are available to keep workers busy.

3.6 Importance of Polling for Coarse-grained Parallelism

We noted in Section 3.4.2 that message handling delays affect the time it takes to steal a task. Especially random victim selection is prone to poor performance when steal requests take a long time to get processed. (Recall Equation (3.2).) To stay responsive, workers must regularly check for steal requests. Such checks are easily integrated with deque operations, primarily push and pop, which we assumed are called frequently enough to guarantee progress. But even if most tasks are indeed fine grained, this assumption is bound to become a problem.

Suppose a worker schedules a single long-running task that does not call back into the runtime system. While the worker is busy running the task, any steal request that arrives is put on hold until the task is finished. The performance implications are shown in Figure 3.11 using the BPC benchmark. Let us concentrate on the red curve first. The other curves represent three attempts to improve performance—two based on software polling, one based on interrupts—which are described in turn in the following subsections.

Whether tasks run for 10μs or 100μs has practically no effect on scalability, a result that seems to contradict the expectation that coarse-grained parallelism is easier to exploit than fine-grained parallelism. On closer inspection, it becomes clear that idle workers are stalled until their steal requests are handled, which requires waiting for



(a) BPC with $d = 10\,000$, $n = 9$, and $t = 10\mu$s **(b)** BPC with $d = 10\,000$, $n = 9$, and $t = 100\mu$s

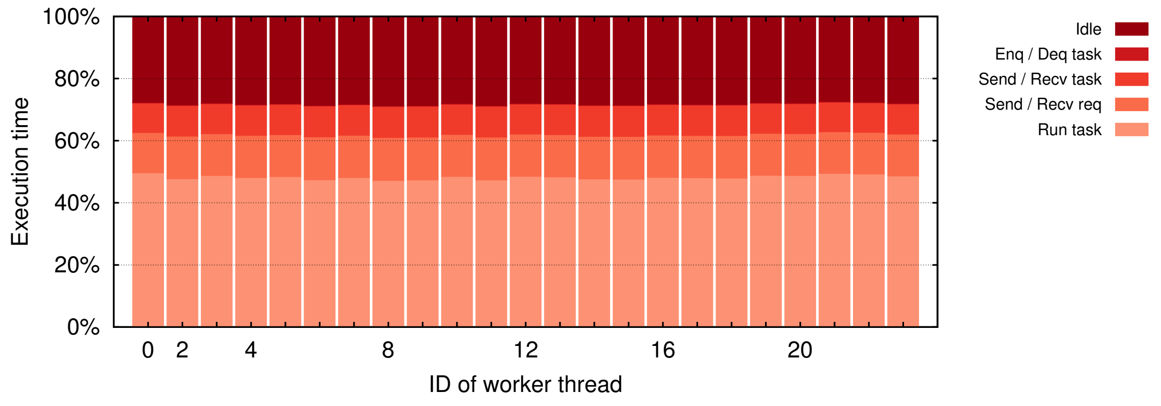
Figure 3.11: The longer it takes to execute a task, the more important it becomes to check for pending steal requests. (GCC 4.9.1, `-O3`, AMD Opteron multiprocessor)

other workers to finish their current tasks. It is not hard to imagine a scenario where n workers are sitting idle until the only worker with tasks to spare returns from a long-running task. The longer it takes to complete a task, the more important it becomes to check if steal requests are waiting to be handled, and, if so, handle them.

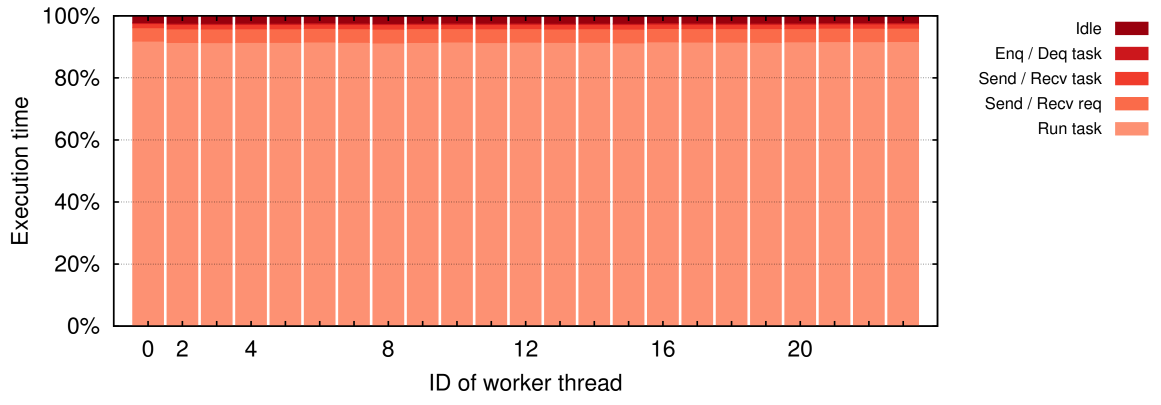
3.6.1 Software Polling

Software polling necessitates a new function `rts_poll`, which, when called, handles pending steal requests. Programmers can incorporate this function in long-running tasks and have workers periodically check for steal requests. This guarantees that message latencies are bounded and independent of task length. In the case of BPC, it makes sense to add polling to the compute-intensive consumer tasks, which neither create child tasks nor synchronize with other tasks and thus do not invoke the runtime system. A consumer task contains a loop to which we add a call to `rts_poll` on every, say, tenth iteration. Alternatively, we may install a timer to poll at regular intervals. The blue curve in Figure 3.11 reveals that polling is simple and effective: scalability improves with increasing task length, as one would expect.

To understand the influence of polling on the performance of work stealing, Figure 3.12 breaks down the execution time for each worker in a 24-thread run. Not paying attention to steal requests while scheduling an unbalanced computation such as BPC can impede the performance of workers. As long as tasks are sufficiently fine grained, workers can respond to steal requests in a timely manner. The situation changes, the longer it takes to complete a task: the message latency grows proportionally to the task

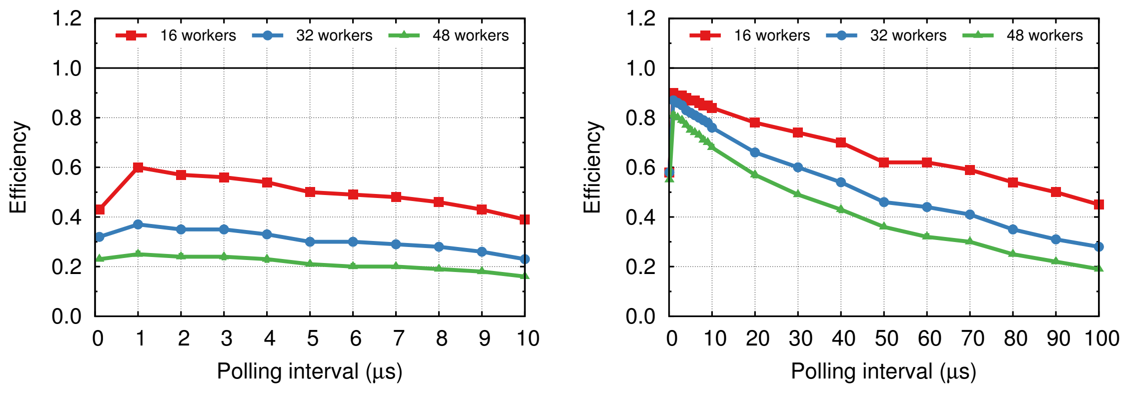


(a) Workers run tasks uninterrupted



(b) Workers poll for steal requests once per microsecond

Figure 3.12: Execution time profile of BPC with $d = 10\,000$, $n = 9$, and $t = 100\mu$s showing the performance implications of (a) deferring steal requests until after task completion and (b) polling once per microsecond for the duration of a task. Worker 1 was dedicated to termination detection and is excluded from the list of worker IDs. As in Figure 3.10, the code was instrumented by inserting `RDTSC` instructions (see [125], Chapter 17), which led to 2% slower execution. (GCC 4.9.1, `-O3`, AMD Opteron multiprocessor, 24 worker threads)



(a) BPC with $d = 10\,000$, $n = 9$, and $t = 10\mu$s **(b)** BPC with $d = 10\,000$, $n = 9$, and $t = 100\mu$s

Figure 3.13: Parallel efficiency varies with the time between two polling operations. The rightmost values, 10μs in (a) and 100μs in (b), correspond to no polling while tasks are running. (GCC 4.9.1, `-O3`, AMD Opteron multiprocessor)

length. Increasing the task length is akin to decreasing the responsiveness of workers, which is a major factor in the performance of load balancing. In practice, tasks can vary in length by multiple orders of magnitude, from microseconds to seconds, and runtime systems are expected to handle a wide range of parallel workloads.

On top of the delay, stealing often fails; on average, eight attempts are needed to receive new work. The net result is that workers spend less than 50% of their time on tasks, in contrast to more than 90% when workers poll once every microsecond for the duration of a task. The difficulty that comes with polling is to strike a balance between polling frequently enough and polling too frequently. On the one hand, steal requests must be handled promptly, or stealing stalls and performance suffers. On the other hand, polling incurs overhead that gets in the way of other work. Failure to keep this overhead in check can degrade performance.

Figure 3.13 shows how efficiency varies with the polling interval. Short intervals down to 1μs are feasible because checking for steal requests is relatively cheap. In the absence of steal requests, `rts_poll` returns after calling `channel_peek`, which takes around 200 cycles on our test system.

3.6.2 Polling with a Background Thread

An interesting question to consider is: can we take advantage of software polling without requiring the programmer to manually insert polling statements? The idea is to use a background thread to process messages while a worker is busy running tasks.

Our implementation is as follows: Every worker creates a helper thread, pinned to the same processor, that sleeps until being notified. When woken up, the helper thread assumes ownership of the worker's deque and channels and starts handling steal requests by running a loop that calls `rts_poll` before yielding control back to the worker. As long as the helper thread is tasked with handling steal requests, the worker must not access its deque, poll for steal requests, or touch any data structure belonging to the runtime system. For this reason, the helper thread is only allowed to service steal requests while the worker is running user code. It is put to sleep when a task is finished. Thus, every long-running task starts with a call to `wake_helper` and ends with a call to `stop_helper`.

Unfortunately, context switches and synchronization come with a cost that is reflected in Figure 3.11. That helper threads succeed in handling the majority of steal requests is practically irrelevant, given how much overhead is introduced. Repeating the experiment on a 12-core, 24-thread Intel Xeon E5-2630 processor and assigning helper threads to the logical processors afforded by Hyper-Threading did not change the results.

3.6.3 Interrupts

Software interrupts triggered by signals provide an alternative to polling. Typically used to notify processes of exceptional events, such as the notorious segmentation violation, signals allow processes or threads to communicate without exchanging data, except for the signal number. Processes can catch signals by defining and registering signal handlers, procedures that are invoked upon receipt of the signals they are registered to handle. The advantage of interrupt-driven communication is that processes do not have to wait or poll for messages. Signals are handled as soon as they arrive, unless these signals are blocked or otherwise ignored.

We install a process-wide signal handler for the user-defined signal `SIGUSR1` and cause every steal request to send a signal via `pthread_kill(thr, SIGUSR1)`, where `thr` is the thread ID of the selected victim. The signal handler wraps a call to `rts_poll`, which, because it involves deque and channel operations, cannot safely execute in runtime system context[7]. For this reason, and to keep complexity under control, signals are blocked by default and must remain blocked for as long as a worker executes runtime system code. When a worker enters a task function, a call to `enable_interrupts` unblocks signals and allows interrupts to occur. When a worker leaves a task function,

[7]If a signal interrupts the execution of a non-reentrant function, and the signal handler happens to call the same function, the behavior of the program is undefined.

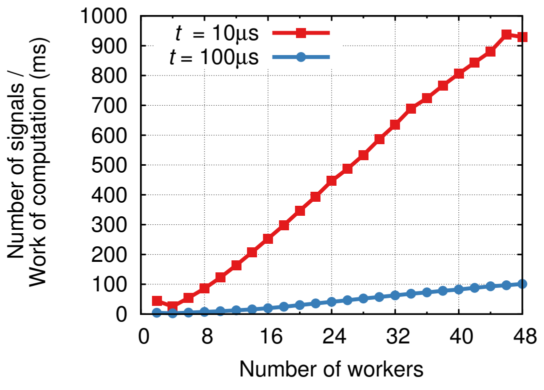


Figure 3.14: The problem with signals: Unbalanced computations cause frequent interrupts, as in this of example of running BPC with $d = 10\,000$ and $n = 9$. The corresponding speedups are included in Figure 3.11. (GCC 4.9.1, `-O3`, AMD Opteron multiprocessor)

a call to `disable_interrupts` makes sure that signals are blocked again.

Figure 3.11 shows that interrupts fail to scale beyond a dozen workers. For the fine-grained workload, we measure a maximum speedup of 3.57. Starting from six workers, every additional worker slows down execution. With 14 workers, the parallel program is already slower than its sequential version. The medium-grained workload scales slightly better, achieving a speedup of 10.45 with 14 workers.

Unbalanced computations such as BPC generate many steal requests. Figure 3.14 gives an idea of the resulting signal overhead by plotting the number of signals sent divided by the work of the computation. Hundreds of signals on average per millisecond interrupt the program far too often. Judging from these numbers, between 20 and 50 signals per millisecond can be tolerated. The gentle slope resulting from $t = 100\mu s$ suggests that interrupts might be a viable alternative to polling, given sufficiently long-running tasks and small worker pools.

Again, repeating the experiment on a 12-core, 24-thread Intel Xeon E5-2630 processor, we measured speedups of 4.54 ($t = 10\mu s$) and 14.26 ($t = 100\mu s$) with six and 18 workers. Beyond these numbers of workers, performance degraded, albeit slower than in Figure 3.11.

3.6.4 Polling versus Interrupts

In the end, like many decisions, choosing between polling and interrupts involves a trade-off. Both approaches have their strengths and weaknesses. Polling succeeds as a low-overhead mechanism for handling steal requests, at the cost of shifting some burden from the runtime system to the programmer. Interrupts are raised and handled transparently, but the associated overhead can be overwhelming. It is well known that coarse-grained parallelism favors interrupts, while fine-grained parallelism requires polling [226, 57, 143, 154].

Ideally, we would like to be able to combine the performance of polling with the reliability of interrupts. Addressing this problem in hardware might lead to an efficient solution, such as an interrupt-based mechanism that executes entirely in user space [215]. Building on top of cache coherence protocols, communication could be accelerated by exposing cache events, such as updates to specific cache lines to indicate the availability of messages, that trigger interrupts and cause the processor to call predefined handler functions [225, 181]. For now, we stick with polling for practical reasons. Without some hardware support, an interrupt-driven implementation is hard to reconcile with the idea of a runtime system that targets fine-grained parallelism:

- Interrupts have a certain runtime overhead, exacerbated by frequent transitions between user mode and kernel mode, which renders them impractical for frequent events such as steal requests. Part of the overhead may be amortized by merging signals close in time to each other [22], but apart from causing fewer interrupts, the potential for optimization is limited.

- Interrupt handling complicates (an already complex) runtime system as interrupts happen at unpredictable times and demand immediate attention. Suppose a worker is interrupted while pushing tasks onto its deque. The interrupt might have occurred when the worker was updating the head, leaving the deque in an inconsistent state until control is transferred back from the interrupt handler. To prevent interrupt handlers from accessing inconsistent data structures, interrupts must be disabled on entry into critical sections and restored on exit. Forgetting to do so will lead to unpredictable results. There may be no practical alternative but to leave interrupts disabled, except when running user code.

- Relying on the operating system to interrupt the execution of workers runs counter to our stated goal of improving portability without sacrificing performance, unless we can guarantee the efficiency of the operating system.

Software polling can be integrated with local deque operations, such that every push/pop has the side effect of handling steal requests. In case of no steal requests, polling overhead is determined by `channel_peek`. When scheduling fine-grained parallelism, workers push/pop tasks frequently enough to obviate the need for `rts_poll` in most cases. Likewise, programs that create many child tasks, such as implementations of divide-and-conquer algorithms, tend to achieve short message latencies. For long-running tasks it is up to user code to determine the frequency of polling.

Therein lies the biggest downside of polling: it affects the performance of the runtime system if used incorrectly. The programmer must be aware of this limitation and work around it when necessary by inserting (or removing) calls to the runtime library. The problems with imperfect or "leaky" abstractions are well known [130, 227]. However, as pointed out by Spolsky [227], all non-trivial abstractions tend to leak some implementation details that are considered part of the abstraction. Mytkowicz et al. argue that every task abstraction is leaky in the sense that programmers must reason about task granularity to avoid creating too many fine-grained tasks [179]. Even the most carefully optimized runtime system carries some overhead that can neither be hidden nor ignored. Whether polling can benefit performance is likewise a matter of task granularity.

Many runtime systems that need to handle messages prefer polling over interrupts. X10 provides a method called `Runtime.probe` to handle pending activities including steal requests [218, 100]. Manticore integrates polling with garbage collection [202]. Lazy Threads [99] and StackThreads/MP [236] depend on polling to know when to migrate threads to other processors. In sufficiently advanced implementations, the compiler may help with inserting runtime checks into application code [90].

Polling and interrupts can be combined, motivated by the observation that the dynamic behavior of some programs complicates a choice between the two approaches [143, 154]. Maquelin et al. proposed Polling Watchdog, a hardware extension that generates a (hardware) interrupt when the network is not polled within a certain period of time after the arrival of a message [154]. The work-stealing scheduler of Acar et al. creates an additional thread that issues interrupts as frequently as every 200μs [28]. This is akin to polling at regular intervals without assistance from the programmer. To approach the performance of polling, however, more control may be needed than the runtime system allows. We have seen in Figures 3.11 and 3.13 that a polling interval of 200μs is too long for our purposes.

We end the discussion by revisiting the results of section 3.4.2. Recall that, given a single producer, multiple consumers, and steal-one work stealing, random victim selection is preferable to last-victim selection when confronted with fine-grained parallelism of up to 25μs per task. (See Figure 3.8.) Above 25μs, however, last-victim selection wins because it is less sensitive to message handling delays. What happens when we introduce polling? After all, polling serves to reduce message handling delays, so we

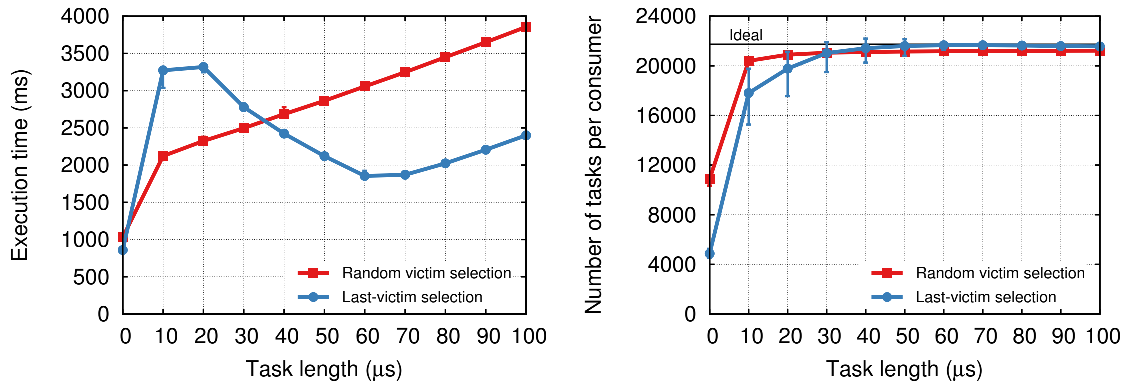


Figure 3.15: Influence of polling on the steal-one strategies of work stealing, continuing our study of SPC with $n = 10^6$ and t between 0 and 100 microseconds. More details and previous results are found in Figure 3.8. The polling interval was set to one microsecond. (GCC 4.9.1, `-O3`, AMD Opteron multiprocessor, 48 worker threads)

expect performance to improve, especially when victims are selected randomly. Figure 3.15 confirms: polling narrows the performance gap between the two strategies, with speedups of up to 2.52× over the previous results in Figure 3.8. In terms of relative performance, we reach the same conclusion as in section 3.4.2, except for the point of intersection, which has shifted from $25\mu s$ to $35\mu s$.

3.7 Summary

This chapter has focused on steal requests and tasks—the data that is exchanged over channels. Every worker allocates two specialized channels: an MPSC channel where it receives steal requests and an SPSC channel where it receives tasks. Steal requests are forwarded rather than acknowledged, letting workers steal on behalf of others upon receiving steal requests that cannot be handled. Random victim selection fits in well with forwarding steal requests, but may cause a lot of communication if only few workers have tasks left. Stealing half of a victim's tasks—steal-half—is straightforward to implement with private task queues, especially when shared memory is available, in which case tasks do not need to be copied. While steal-half is important to tackle fine-grained parallelism, polling is necessary to achieve short message handling delays when workers schedule long-running tasks.

4 | Synchronization Mechanisms

Synchronization is required to coordinate the execution of tasks. Our task model is inspired by OpenMP 3.0 [40, 17, 18], which offers two synchronization constructs: a full task barrier and a task barrier for child tasks. The former detects when all tasks are done; the latter waits only for immediate child tasks. We support a more general synchronization construct—futures [91]—for which we present a portable and efficient implementation based on channels.

Section 4.1 starts with an algorithm for termination detection in preparation for describing task barriers in Section 4.2. Fine-grained synchronization is achieved with the help of futures, whose implementation is detailed in Section 4.3. Futures lend themselves to parallelizing divide-and-conquer algorithms. Whether they are lightweight enough to be able to compete with Cilk Plus is evaluated in Section 4.4.

4.1 Termination Detection with Steal Requests

Workers are either busy, executing tasks, or idle, searching or waiting for tasks. Termination detection is the problem of determining when all workers are idle, meaning that every task created up to this point in the program has been completed. Because of the nature of work stealing, idle workers may continue working at any point in time, as long as there are tasks in the system.

Termination detection is relatively straightforward to implement with shared memory or a shared memory abstraction [116, 67]. The basic idea is to count the number of idle workers (or busy workers) to detect when all deques are empty and all work is done. Initially, the count of idle workers is equal to the number of workers N. A count of N indicates that the computation has terminated (or has not been started). When a worker starts working, it decrements the count. When a worker runs out of tasks and starts stealing, it increments the count. Before a worker tries to steal a task from a non-empty deque, it decrements the count to avoid a race condition with the victim, which might run out of tasks and declare itself idle before the thief has a chance to get back to work. Without a prior decrement, the victim could trigger false termination detection by incrementing the count to N, signaling completion while the computation is still in progress. If a thief has decremented the count, but stealing fails, it must increment the count again.

Distributed termination detection is based on collecting votes to decide if termination has occurred [162]. In Dijkstra's classic algorithm, N processors are arranged in a ring [72]. An idle processor, say processor 0, initiates termination detection by sending a token to its neighbor, processor 1, which receives the token, colors it if the computation is still in progress, and passes it along to processor 2. This token passing continues until the token is returned to processor 0, which, depending on the color of the token and its own vote, can conclude termination or start a new round of termination detection at a later time.

Dijkstra's token-passing algorithm has been used by distributed work-stealing schedulers [75, 204]. Dinan et al. have implemented a variation of this algorithm in which workers are arranged in a tree, rather than in a ring [73, 74]. Tokens are passed down the tree, asking for votes, and up the tree, combining votes from subtrees. Termination is detected when the root receives a positive vote; otherwise, if one or more workers disagree, a new round of voting is started.

4.1.1 Managing Idle Workers

In this section, we introduce a termination detection barrier that does not depend on shared state nor burdens workers with separate control messages, which are required by distributed memory algorithms. The former is the result of using channel communication; the latter is achieved by collecting information about steal requests. There is no need to inject more messages into the system when steal requests are already passed between workers. Additional data, if needed, can be piggybacked with steal requests. In addition, forwarding makes sure that workers enter termination detection only when it is likely that no work remains [185].

We take the following approach: A worker is assigned the task of determining whether termination has occurred by keeping track of steal requests and counting the number of idle workers. We call this worker *manager*. But simply counting every steal request towards the number of idle workers may lead to early termination detection. There are two complications that must be dealt with: First, workers may send speculative steal requests, expecting to run out of work but not knowing for sure. Receiving a steal request does not necessarily mean that the thief is idle. It may be that the thief is trying to prefetch some work. Second, work stealing happens between workers, without the manager's knowledge. If the manager were involved in every steal, it could quickly become a bottleneck for scalability, especially with frequent load balancing.

The first problem to address is that of counting only idle workers. To distinguish idle from busy workers, we extend steal requests with a field, *status*, that indicates whether a worker is *working*, *idle*, or *registeredIdle* by the manager. Initially, a new steal request identifies a worker as *working*, unless, of course, the worker is already idle when it starts to steal. A worker is idle if it has nothing to do besides handling messages and waiting for tasks.

Generally speaking, there are two possible outcomes of a steal request: it either succeeds or fails. If it fails repeatedly, say n times, it is returned to allow the thief to change the steal request to *idle* (if true). This is a point worth emphasizing: if the steal request were not returned, the transition from *working* to *idle* could not happen; only the thief itself is in a position to confirm that it has no tasks left.

Figure 4.1 shows the changes to the handling of steal requests. In addition to *status*, a steal request is extended with another field, *failed*, that counts how many times a steal request has been forwarded. This count is used to decide when a steal request is returned to the thief (lines 29–31). A returned steal request is either discarded if the worker has tasks (lines 1–5), forwarded again if the worker is still busy, but likely running out of work soon (lines 21–23), or sent to the manager if the worker is idle. (lines 18–20).

The manager handles steal requests just like other workers, except for the additional requirement to examine *status* in order to keep track of idle workers. When the manager runs out of tasks and receives a steal request that points to an idle worker, it changes the steal request to *registeredIdle*, updates the set of idle workers, and checks for termination, that is, if every worker is registered as idle (lines 9–12). Afterwards, it selects a new victim for the steal request. Note that, according to Figure 4.1, the manager will eventually send a message to itself, passing up the opportunity to directly register itself as idle. An implementation of HandleStealRequest should consider this optimization. It is also possible to use a dedicated manager that forwards every steal request. Because a dedicated manager has no need for a deque, nor for sending own steal requests, it makes sense to write two versions of HandleStealRequest so that manager and workers avoid unnecessary runtime checks.

The second problem is related to work stealing: idle workers, including those identified as *registeredIdle*, may receive tasks and start working again. Clearly, we do not want to put the manager in a position to acknowledge every single steal, but we still have to eliminate any possibility of detecting termination on the basis of outdated in-

```
HandleStealRequest() // Second version
    Let $Q_i$ be the private deque of tasks of worker $i$,
        $C_i$ be the channel for sending steal requests to worker $i$,
        $C_m$ be the channel for sending steal requests to manager $m$,
        $S$ be the steal request to handle,
        $n$ be the number of steals to attempt
 1  if $Q_i$ is not empty
 2      if $i == S.thief$
 3          // Own steal request is no longer needed
 4          Discard $S$
 5          return
 6      Pop task $t$ from the top of $Q_i$
 7      Send task $t$ to channel $S.chan$
 8  else
 9      if $i == m \wedge S.status == idle$
10          $S.status = registeredIdle$
11          Add $S.thief$ to the set of idle workers
12          Check for termination
13      if $S.failed == n$
14          // Steal request must have been returned
15          assert $i == S.thief$
16          // Start new round of stealing (alternatively, back off if $S.status == registeredIdle$)
17          $S.failed = 0$
18          if worker $i$ is idle
19              $S.status = idle$
20              Send $S$ to channel $C_m$
21          else
22              Select a worker $j$, $j \neq i$, at random
23              Send $S$ to channel $C_j$
24      else
25          $S.failed = S.failed + 1$
26          if $S.failed < n$
27              Select a worker $j$, $j \neq i \wedge j \neq S.thief$, at random
28              Send $S$ to channel $C_j$
29          else
30              // Return steal request to $S.thief$
31              Send $S$ to channel $C_{S.thief}$
```

Figure 4.1: Because steal requests can be sent ahead of time, a worker must confirm that it is idle before it can be counted as such by the manager.

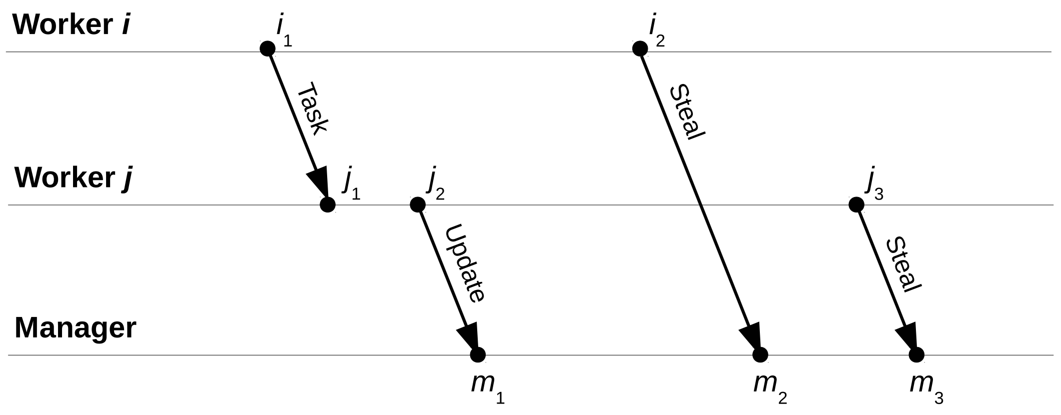


Figure 4.2: Updating the manager about the state of workers: a correct execution in the presence of a race condition. i_x, j_x, and m_x are events denoting the sending or the receipt of a message x. The receipt of the update message m_1 is *not* guaranteed to happen before the receipt of a subsequent steal request from worker i. Specifically, since i_2 and j_2 are concurrent events, neither $i_2 \rightarrow j_2$ nor $j_2 \rightarrow i_2$. As a result, the order of m_1 and m_2 depends on the timing and sequence of the other workers' operations.

formation. To illustrate the problem of early termination detection, suppose worker i sends a task to worker j, which is idle and registered as such by the manager. (Worker j's steal request has *status == registeredIdle*.) There is no way the manager could know that worker j has picked up a new task without simultaneously creating a race condition with an undesirable outcome if, subsequently, worker i runs out of tasks and becomes idle itself. The solution to this problem is to inform the manager of a worker's state change. We have two options: (1) have worker j send a message upon receiving tasks, or (2) have worker i send a message on behalf of worker j, either before or after sending tasks to j.

4.1.2 Updating the Manager

Let us first consider option (1). Figure 4.2 illustrates a possible ordering of sends and receives among worker i, worker j, and the manager. In this diagram, threads are drawn as horizontal lines, with time progressing from left to right. Dots denote events, such as the sending or receipt of a message, and arrows indicate the direction of communication between threads.

Four messages are shown: worker i sends a task to (idle) worker j, worker j sends an update to the manager, and, at a later time, both worker i and worker j send steal requests to the manager. The following discussion relies on two assumptions: all messages are sent over channels, and messages sent over the same channel are not reordered. Let a_x and b_x denote the sending and the receipt of a message x between workers a and b. Expressed in terms of "happens before" ($\rightarrow$) [140], if $a_x \rightarrow b_x$ (trivially) and $a_x \rightarrow a_y$, that is, worker a sends another message y after message x, then $b_x \rightarrow b_y$ by the FIFO property of the channel.

Looking at Figure 4.2, we see that $m_1 \rightarrow m_2$. The manager receives the update about worker j being no longer idle before a subsequent steal request sent by worker i. Provided the manager acts on the update in the time between m_1 and m_2, the possibility of early termination detection after m_2 is eliminated. Can we prove that $m_1 \rightarrow m_2$, implying $m_2 \nrightarrow m_1$ (strict partial order)?

We know that $i_1 \rightarrow j_1$, $j_1 \rightarrow j_2$, because worker j is responsible for sending the update to the manager, and $j_2 \rightarrow m_1$. By transitivity, $i_1 \rightarrow j_2$ as well as $i_1 \rightarrow m_1$. We also know that $i_1 \rightarrow i_2$, $i_2 \rightarrow m_2$, and, again, by transitivity, $i_1 \rightarrow m_2$. On the other hand, neither $i_2 \rightarrow j_2$ nor $j_2 \rightarrow i_2$. We say the two events are concurrent, denoted $i_2 \parallel j_2$. If we cannot order i_2 and j_2, there is no guarantee that $m_1 \rightarrow m_2$. However unlikely, it is possible that, in a given execution, i_2 races with j_2, causing m_2 to happen before m_1. If the manager concludes that worker i is idle, but is still unaware that worker j has picked up a new task, it may falsely detect termination.

Note that we have to be pessimistic: $m_2 \rightarrow m_1$ does not pose a problem if the manager has tasks itself, or if m_2 is the receipt of a steal request that indicates that worker i is still working. Such a steal request does not count towards the number of idle workers.

Given that option (1) violates the safety property of termination detection, we are left with option (2): have worker i send the update in place of worker j. Figure 4.3 shows a possible ordering of events, drawn like Figure 4.2 so that $m_1 \rightarrow m_2$. Again, can we prove that this is true for every possible ordering?

We now have $i_1 \rightarrow m_1$, $i_1 \rightarrow i_2$, because worker i is responsible for sending the update to the manager, and $i_2 \rightarrow j_1$. Of course, $i_1 \rightarrow i_2$ and $i_2 \rightarrow i_3$, just as $i_1 \rightarrow i_2$ earlier in Figure 4.2. Since $i_1 \rightarrow i_3$ by transitivity, $m_1 \rightarrow m_2$ if update message and steal request use the same channel. Provided that we implement update messages in terms of steal requests and thereby unify both message types, option (2) guarantees the correctness of the algorithm by taking advantage of the FIFO property of channels.

Meeting this requirement is straightforward. In fact, there is no need to distinguish between update messages and steal requests in the first place. When worker i has tasks and receives a steal request from worker j that is *registeredIdle*, worker i changes the steal request back to *working*, forwards it to the manager as an update, and sends a task to worker j, completing the steal. The manager receives the repurposed steal

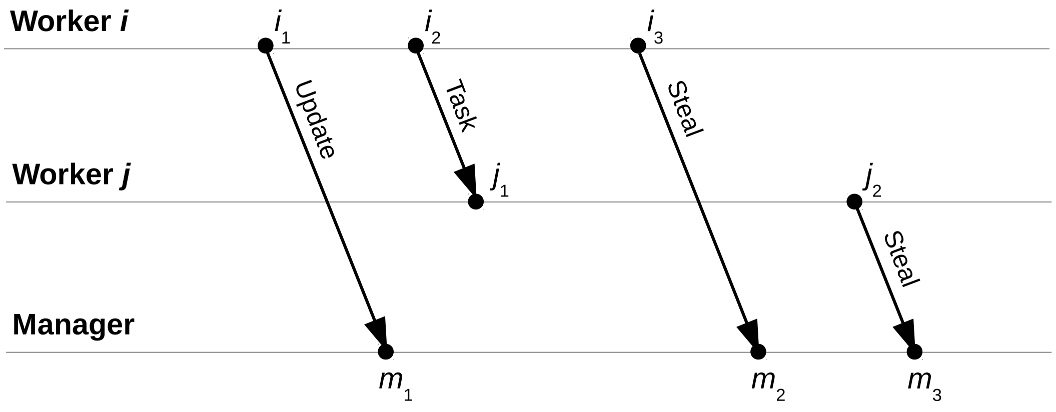


Figure 4.3: Updating the manager about the state of workers: a correct execution with no potential race condition. i_x, j_x, and m_x are events denoting the sending or the receipt of a message x. The receipt of the update message m_1 is guaranteed to happen before the receipt of any subsequent steal request from worker i or worker j if update message and steal requests are received on the same channel.

request and applies the update by removing worker j from the set of idle workers. This requires the manager to distinguish updates from regular steal requests, which, unlike updates, must be forwarded in case the manager has no task to send. If worker j's steal request is not *registeredIdle*, an update is not needed and simply omitted because worker j is not in the set of idle workers, so nothing needs to be corrected.

Note that it does not matter which of the two messages—update to manager or task to worker j—comes first. As long as worker i sends the update before a subsequent steal request, the update will be received first.

Figure 4.4 highlights the final changes to the handling of steal requests. Only the manager receives updates, which are handled differently from regular steal requests (lines 1–3). If the manager handles a steal request that is *registeredIdle*, an update message is omitted (lines 8–10). By writing two versions of HandleStealRequest, one for $i == m$ and one for $i \neq m$, we can eliminate the corresponding runtime checks and simplify the code.

The fact that steals may generate update messages that are sent to the manager has a consequence. Suppose worker i handles a steal request from worker j, which is registered as idle. Worker i, having tasks, updates the steal request and forwards it to the manager as required by the algorithm. It then reactivates worker j by sending a task. Worker j receives the task, executes it, runs out of work again, and sends another steal request, coincidentally, directly to the manager. While worker j's new steal request cannot overtake the old one (the update message), it may still be the case

```
HandleStealRequest() // Final version
    Let $Q_i$ be the private deque of tasks of worker $i$,
        $C_m$ be the channel for sending steal requests to manager $m$,
        $S$ be the steal request to handle
if $i == m \wedge S.update ==$ TRUE
    Remove $S.thief$ from the set of idle workers
    return
// S must be a regular steal request
assert $S.update ==$ FALSE
if $Q_i$ is not empty
    if $S.status == registeredIdle$
        if $i == m$
            // Manager omits sending an update message to itself
            Remove $S.thief$ from the set of idle workers
        else
            // Send an update message to the manager
            $S.status = working$
            $S.update =$ TRUE
            Send $S$ to channel $C_m$
    if $i == S.thief$
        // Own steal request is no longer needed
        Discard $S$
        return
    Pop task $t$ from the top of $Q_i$
    Send task $t$ to channel $S.chan$
else
    // Same as in Figure 4.1
```

Figure 4.4: A worker notifies the manager when it reactivates another worker. Consequently, the manager must distinguish between updates and regular steal requests.

that both messages arrive in close succession. This implies that a channel capacity of n, with n being the number of workers, is no longer sufficient to guarantee non-blocking operations when trying to send steal requests to the manager. To account for the worst case of $2n$ simultaneous steal requests/updates, we must double that channel's capacity. Other channels are not affected since updates are only sent to the manager.

Termination detection does affect the maximum number of pending messages. Suppose all workers including the manager are registered as idle, and worker i starts to create tasks and handle steal requests. Let m be the number of tasks that worker i sends to a thief. Given that worker i does not send tasks to itself, it will send at most $(n-1)\cdot m$ tasks (see Section 3.3.1) plus $n-1$ notifications. What happens with worker i's steal request? It is either discarded, requiring another notification, or forwarded. Discarding it might cause worker i to send a new steal request shortly thereafter. The total number of messages is therefore bounded by $(n-1)\cdot m+n+1 = n\cdot m-m+n+1$. If $m = 1$, there are at most $2n$ pending messages ($n-1$ tasks plus $n+1$ updates/steal requests), doubling the previous value of n ($n-1$ tasks plus one steal request).

4.1.3 Performance

Termination detection latency We will refer to the time it takes to detect termination after its occurrence as termination detection latency or delay. What interests us is how explicit communication affects this delay. Figure 4.5 compares our algorithm to an adaptation of Herlihy and Shavit's algorithm [116]. In this implementation, there is no manager; its role is filled by a shared variable `count` that workers increment and decrement using atomic operations. A worker confirms that it is idle by incrementing `count`. Whenever a worker sends a task to an idle thief, it decrements `count`. (Recall that whether a thief is idle or not is recorded in its steal request.) Termination is detected when `count` is found to equal the number of workers.

As a consequence of sending steal requests, workers must confirm that they are idle before they are allowed to increment `count`. The steals attempted in between help reduce the number of atomic operations and associated contention [185], but can increase the latency to termination [218]. This is apparent in Figure 4.5 (on the right), which shows worst-case latencies as a result of sending steal requests only after the computation has already terminated. (The microbenchmark does not create tasks and uses a thread barrier to line up all workers before they have a chance to send steal requests.) Although this situation is unlikely to occur in practice, given that workers are allowed to send steal requests ahead of running out of work, some workers may have been busy and cannot declare themselves idle as long as their steal requests are being

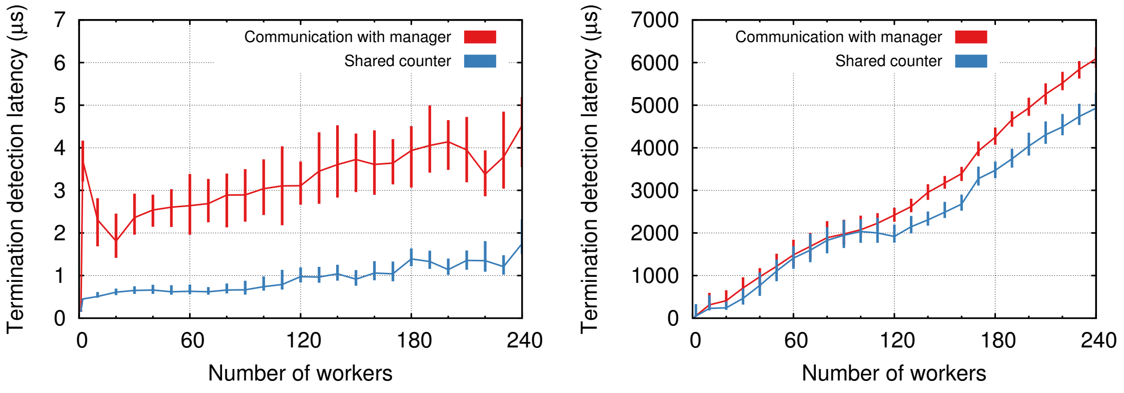
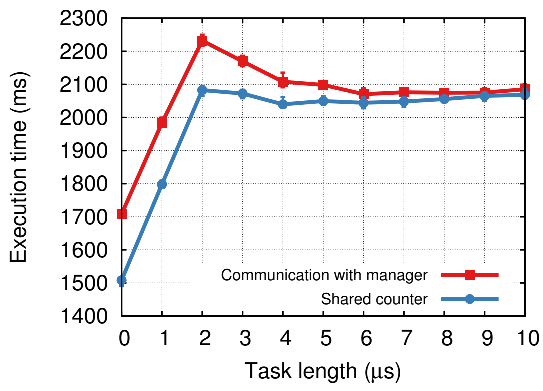


Figure 4.5: Termination detection latency in the best case, when all workers are already idle (left), and in the worst case, when no steal request has been sent yet (right). Lines connect the median latencies of 100 data points. Vertical bars show interdecile ranges. (ICC 14.0.1, `-O2`, Intel Xeon Phi)

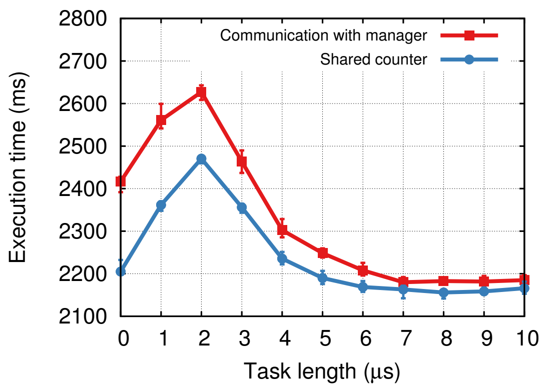
passed between coworkers. The resulting latency grows linearly with the number of workers unless steal requests are returned after a constant number of failed attempts. Assigning the task of counting idle workers to a manager increases latency by up to 22% compared to using shared state and atomic operations.

Figure 4.5 shows termination detection to be orders of magnitude faster when workers are already idle. (The microbenchmark detects termination twice, measuring how long it takes the second time.) This is not surprising as most communication has taken place at this point. The difference in latency stems from worker 0 waiting to receive a "termination detected" message, which is redundant when using shared state. Another way to make these channel operations redundant is to have worker 0 detect termination in the capacity of manager. While this is certainly possible, it may not represent the best configuration, considering that worker 0 is intended to run the root task, whose execution may conflict with runtime system duties, such as keeping track of idle workers. Other workers are either running mostly short tasks or are handling steal requests, which makes them better candidates for overseeing termination detection. In this and the following experiments, worker 1 serves as manager. On a 60-core, 240-thread Xeon Phi, the manager is forced to share a core with up to three coworkers.

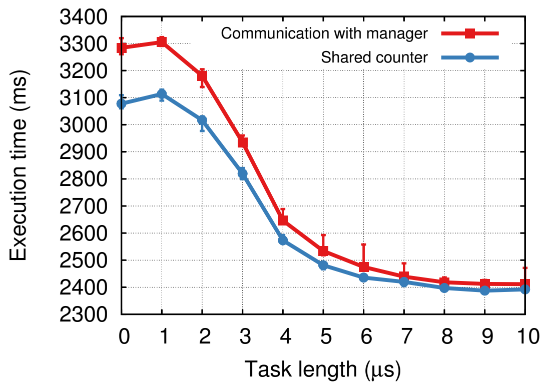
Impact of update messages What also interests us is how update messages affect performance. To this end, we pick the benchmark that generates the most traffic in terms of update messages and determine the performance penalty for avoiding shared state. In the presence of shared state, neither manager nor update messages are needed,



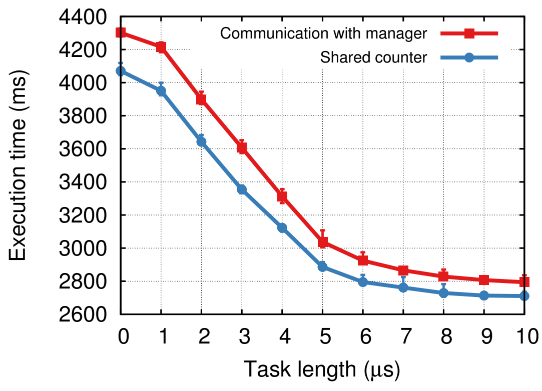
(a) 60 threads



(b) 120 threads



(c) 180 threads



(d) 240 threads

Figure 4.6: Performance difference between termination detection involving explicit communication and termination detection using shared state and atomic operations. The four graphs show the results of running BPC with $d = 100\,000$, $n = 9$, and t between 0 and 10 microseconds using 60, 120, 180, and 240 threads. (ICC 14.0.1, `-O2`, Intel Xeon Phi)

and atomic operations suffice to signal that idle workers are about to receive new tasks. In the presence of a manager, reactivating an idle worker entails two channel operations (not counting the channel operations needed to transfer tasks) and some bookkeeping by the manager to maintain correctness.

Figure 4.6 shows BPC execution times as a function of task length. BPC is a perfect adversary as it causes workers to send a significant number of update messages to the manager, placing a lot of burden on both workers and manager. This burden decreases with increasing task length: we count up to 88 748 messages when a task immediately returns (0 microseconds), but “only” up to 27 137 messages when a task lasts 10 microseconds. Not entirely unexpected, we find the performance impact of sending update messages to be more pronounced towards fine-grained parallelism.

Worker-manager communication increases execution time by at most 6–13% compared to reading and writing shared state. At the other end of the task spectrum, which still represents fairly fine-grained parallelism, the performance penalty is reduced to 1–3%. With 60 threads, the manager has a core to itself. With 120, 180, and 240 threads, it shares a core with one, two, and three coworkers, respectively.

4.1.4 Limitations

The algorithm, as we have described it, has two primary limitations: (1) It is not resilient to losing steal requests. (2) While any worker, including worker 0, can function as manager, having a single manager oversee an increasing number of workers will eventually create a bottleneck, in much the same way as a single shared variable will eventually limit the scalability of Herlihy and Shavit's algorithm [116]. Luckily, both limitations can be addressed. We will not go into too much detail and just briefly describe what we think is interesting future work.

Our reasoning about liveness assumed that the number of workers is known to the manager[1] and that messages are neither dropped nor held up forever. If messages, specifically steal requests, were forgotten or held up indefinitely for some reason, liveness would not be guaranteed, and termination would not be detected reliably without additional communication between manager and workers. For example, messages could be resent upon request by the manager, rather than periodically after a certain amount of time. In any case, we would have to give up the idea of a single steal request per worker to support these changes. Fault tolerance in the face of worker failures or crashes requires checkpointing and/or task recovery facilities, which are beyond the scope of this work [50, 152, 259, 58].

So far, we assumed that a single worker is in charge of termination detection. As the number of workers increases, keeping track of steal requests requires more and more work, potentially overwhelming the manager with too many messages at the same time. Heterogeneous systems may help mitigate this problem by placing the manager on a latency-oriented core while running other workers on smaller, less powerful cores [135]. On larger systems, workers are often grouped into distinct places or locality domains so that work stealing algorithms are able to prioritize local victims over remote victims [102, 170, 235]. This makes it possible to initiate global load balancing only after local load balancing has failed [218, 191]. In the same way, termination can be detected locally within each place before communicating with other places.

[1]The number of workers in the worker pool need not be fixed. Workers may join or leave the computation at any time, provided they notify the manager.

Our algorithm can be made hierarchical by introducing the notion of places and setting up one manager per place. Intra-place termination is detected as described above, while inter-place (global) termination is decided among managers, without involving other workers in the process. Places themselves can form trees that model the complex memory hierarchies of large-scale systems and determine the flow of termination detection messages between managers [260, 170]. For this thesis, however, we continue to focus on workers within a single place.

4.2 Task Barriers

Task barriers enable us to write programs containing parallel regions: a parallel region is characterized by a number of tasks, which must complete before execution can continue. A simple example is the following piece of code, in which a task barrier ensures that a loop will run to completion by waiting for every task created in the loop body:

```
for (i = 0; i < N; i++) {
    if (some_condition(i)) {
        ASYNC(f, i);
    }
}
TASKING_BARRIER();
```

It is important to note that a task barrier says nothing about the order of execution of tasks. All tasks are assumed to be independent and are scheduled accordingly.

A task barrier marks a global synchronization point—a point in the program where all tasks are required to complete. We restrict such synchronization points to the implicit root task, the code between `TASKING_INIT` and `TASKING_EXIT`; it is impossible to invoke a task barrier from a child task or from any task that is a descendant of the root task without violating the barrier semantics.

OpenMP has slightly different barrier semantics, which is not surprising given that barriers existed before the introduction of explicit tasks in version 3.0 of the standard. In OpenMP, a barrier requires that all threads of a team, created by a parallel region, meet at the barrier and complete all tasks of the enclosing region before any thread of the team is allowed to proceed. Invoking a barrier from a child task without executing a nested parallel region will result in a deadlock, as indicated in Listing 4.1.

Recall that `TASKING_INIT` is roughly equivalent to `#pragma omp parallel` followed by `#pragma omp master`. Ending a parallel region is roughly equivalent to calling `TASKING_EXIT`; in both cases, an implicit barrier makes sure that all work is done before any of the worker threads terminate. As a result of binding the root task to the

```c
#include "tasking.h"

int sum(int a, int b)
{
    TASKING_BARRIER(); // Runtime error
    return a + b;
}

int main(void)
{
    // Start of parallel region
    TASKING_INIT();


    // Create child tasks
    // running sum()


    // Explicit barrier
    TASKING_BARRIER();

    // End of parallel region
    // Implicit barrier
    TASKING_EXIT();

    return 0;
}
```

```c
#include <omp.h>

int sum(int a, int b)
{
    #pragma omp barrier // Deadlock
    return a + b;
}

int main(void)
{
    // Start of parallel region
    #pragma omp parallel
    {
        #pragma omp master
        {
            // Create child tasks
            // running sum()
        }

        // Explicit barrier
        #pragma omp barrier

    } // End of parallel region
    // Implicit barrier

    return 0;
}
```

Listing 4.1: Task barriers in child tasks are forbidden in our task model (left) and in OpenMP 3.x (right).

master thread, only the master thread is allowed to call `TASKING_BARRIER`, turning a team barrier into a single-worker barrier with no possibility for contention: the master thread waits until all tasks have executed to completion and is then released from the barrier. In other words, the master thread waits until termination is detected before returning from the barrier.

4.2.1 Extending Termination Detection

Task-based programs may contain several parallel phases separated by task barriers. The pattern of task creation often resembles the following structure:

```c
// Create tasks
...
TASKING_BARRIER();
...
// Create tasks
...
TASKING_BARRIER();
```

Suppose worker 0 leaves a task barrier upon detecting termination. Returning from `TASKING_BARRIER`, worker 0 is the only worker that can be considered busy, namely with the root task, but there is a pending steal request that indicates the opposite: worker 0 is still counted as idle. This may lead the manager to detect termination again; a direct consequence of outdated worker information.

The problem is that termination can no longer be considered a one-off event. The manager cannot be oblivious to the fact that execution continues after a barrier without raising the possibility of race conditions that undermine the safety property of our algorithm. We can close the race window by putting worker 0 in charge of termination detection. In general, however, any worker may function as manager, and worker 0 must somehow signal "end of barrier" to convey to the manager that execution continues. It may not be immediately obvious why additional communication is needed. If the manager assumes that worker 0 is busy between barriers, it must wait for worker 0 to send a new steal request; otherwise, it has no way of knowing when its previous assumption about worker 0 being busy is no longer valid. Termination detection is put on hold until that message arrives. What complicates matters is that worker 0 is not allowed to send a new steal request while the current one has not been handled.

The code on the left-hand side of Listing 4.1 serves as a good example because it contains two task barriers in a row. (The second task barrier is hidden inside `TASKING_EXIT`.) If the manager expects a message after completing the first task barrier, but none is generated, termination will never be detected for the second task barrier. Even creating new tasks does not guarantee that the manager will be notified, unless a steal succeeds, or worker 0 gets a chance to revoke its steal request. In both cases, the result would be an update sufficient to let the manager conclude that execution continues, knowing that one worker must be busy. Recall that an update takes the form of a steal request. To simplify the discussion, we keep referring to these steal requests as updates.

Revoking worker 0's steal request This suggests a first solution: a guaranteed way to trigger an update is to make sure that worker 0 revokes its pending steal request. Doing so entails an update message, notifying the manager that worker 0 is no longer idle. As a result of this update, termination can no longer be detected on the basis of outdated information.

Figure 4.7 depicts the additional communication after detecting termination. The time between detecting termination and sending an update to the manager $(w_2 - w_1)$ is the time it takes worker 0 to revoke its steal request. We will refer to this time as task barrier latency or delay. Note that worker 0 is not required to wait until the manager has received the update if the manager knows about the barrier and awaits a message.

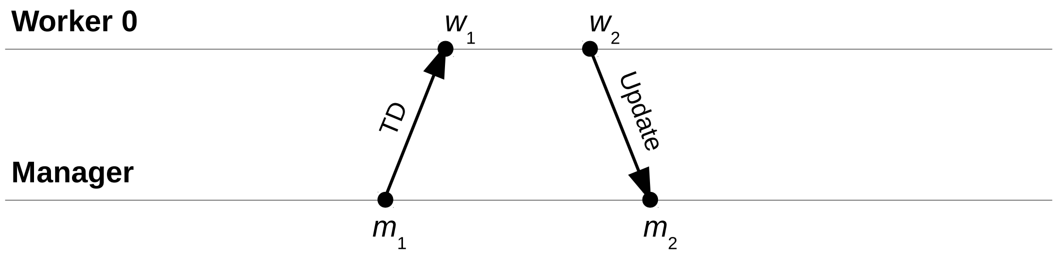


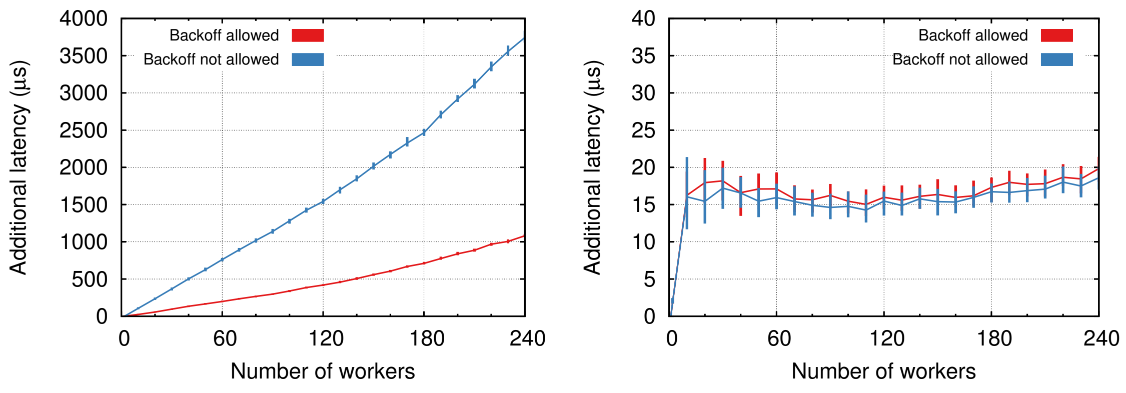
Figure 4.7: Additional communication between worker 0 and the manager after detecting termination: m_1 and w_1 denote the sending and the receipt of a “termination detected” message. w_2 and m_2 denote the sending and the receipt of an update message. After applying the update, all but one worker are registered as idle, and normal termination detection continues. Worker 0 returns from the task barrier immediately after w_2.

Worker 0 cannot cancel its steal request without receiving it first. In fact, worker 0 may have to wait until its steal request is returned by a coworker. This may involve a non-constant amount of communication, depending on the number of attempted steals, since worker 0’s coworkers, except the manager, are oblivious to the barrier. (Recall that only worker 0 executes `TASKING_BARRIER`.) Couldn’t worker 0 just cancel a random steal request to minimize waiting time? All that matters is that one worker is counted as busy after a barrier, whether it is worker 0 or one if its coworkers. From the point of view of the manager, it would appear as if worker 0 handed over the root task.

Revoking a random worker’s steal request Suppose worker 0 receives a steal request from worker i, $i \neq 0$, and wants to cancel it. Since worker i must be aware that its steal request vanishes, worker 0 fakes a successful steal by sending an empty dummy task. Having acknowledged the steal, worker 0 leaves the barrier and continues with the root task, knowing that termination cannot be detected while worker i appears to be busy. But worker i will not stay busy for long because the dummy task contains no real work. If worker 0 were not involved in handling worker i’s subsequent steal request, termination could be detected: worker i could transition from working to idle and lead the manager to conclude that every worker is idle, hence causing termination to be detected. Thus, worker 0 must not be bypassed.

Our solution is simple: we modify the dummy task such that the receiver sends a steal request to worker 0[2], which, being busy running the root task, will keep the steal request until it can reply with a real task or has no work left. In the former case, there is no need for an update message since worker i has not been counted as idle. In the

[2] `void dummy_task(void) {/*send steal request to worker 0*/}`



(a) Revoke worker 0's steal request **(b)** Revoke a random worker's steal request

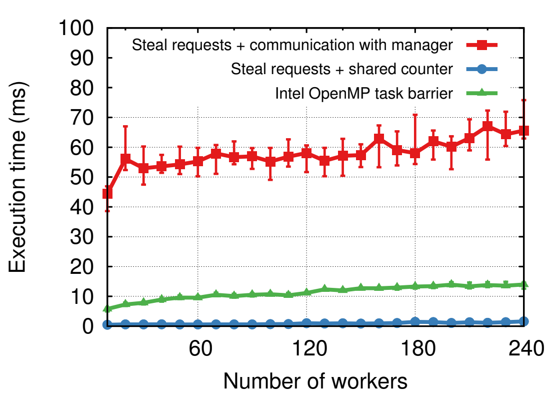
Figure 4.8: Latency between detecting termination and worker 0 returning from the task barrier. In this experiment, all workers, except worker 0, are allowed to back off from stealing after failing n times, where n is the number of workers. When a worker backs off, it refrains from sending a new steal request for a certain period of time, initially set to 100 microseconds. Other workers' steal requests are not delayed and are forwarded as usual. If stealing continues to fail, the backoff period is doubled. If stealing succeeds, the backoff period is reset to its initial value. Lines connect the median latencies of 100 data points. Vertical bars show interdecile ranges. (ICC 14.0.1, `-O2`, Intel Xeon Phi)

latter case, worker i's steal request may be handled by a coworker if one has been given tasks in the meantime. (Some steal requests may have arrived ahead of worker i's.) If, however, every attempt to steal fails, worker i confirms that it is idle, and termination may be detected. In the special case where worker 0 receives its own steal request, the dummy task is elided, and the only message that is sent is an update to the manager.

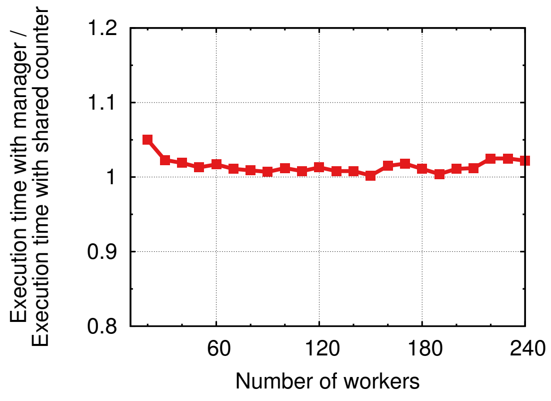
4.2.2 Performance

Task barrier latency We expect a measurable benefit from being able to revoke random steal requests. Figure 4.8 confirms that waiting for worker 0's steal request to be returned is a source of increasing latency. What is more interesting is that other workers can affect this latency if they back off from stealing after failing a number of times. A backoff is a useful strategy to limit contention for a few remaining tasks as well as to adjust the number of messages in a terminating computation. Every worker, with the exception of worker 0, may back off between steals, provided it has been registered as idle to not interfere with termination detection. For the duration of a backoff, a worker refrains from sending a new steal request, but keeps forwarding those of its coworkers because it is not allowed to make backoff decisions on behalf of them.

On the one hand, the more workers back off and postpone their steal requests, the



(a) Microbenchmark measuring the execution time of 1000 task barriers



(b) LU factorization of a 4096 × 4096 matrix using blocks of 64 × 64 elements

Figure 4.9: Task barrier overhead due to explicit communication versus using shared state and atomic operations. Figure (a) quantifies task barrier latency with the help of a microbenchmark; Figure (b) compares performance on a more realistic benchmark with $4096/64 \cdot 2 = 128$ task barriers. In both tests, workers are allowed to back off using the same strategy as in Figure 4.8. The shared counter based task barrier has no discernible overhead beyond that of termination detection. (ICC 14.0.1, `-O2`, Intel Xeon Phi)

smaller the communication overhead required to return worker 0's steal request. The result is a 71–75% reduction of latency, as shown in Figure 4.8 (a).

On the other hand, the more workers back off, the longer it may take before worker 0 receives one of the remaining steal requests. While we do see a slight increase in latency in Figure 4.8 (b), the difference is never more than 12 microseconds, comparing best and worst execution times. Median execution times are within two microseconds of each other.

Whether or not workers are allowed to back off from stealing, the conclusion is the same: performance-wise, it is far preferable to send a dummy task than to wait for worker 0's steal request to be returned. We measure between 33% (10 threads) and 98% (240 threads) lower latencies than the better results of Figure 4.8 (a).

Impact of explicit communication Worker-manager communication inevitably adds some overhead to a task barrier. To quantify this overhead, we stress-test our implementation with a microbenchmark. In addition, for a more realistic use case, we pick the benchmark with the most task barriers and determine the performance impact of using our implementation. The shared counter based algorithm serves as a baseline where task barriers incur zero overhead beyond that of termination detection. Results are shown in Figure 4.9.

Figure 4.9 (a) depicts how long it takes to complete 1000 successive task barriers. The results are interesting insofar as they highlight the overhead associated with steal requests. On average, a task barrier requires at least 58 microseconds because of the communication involved: worker 0 must receive a steal request from worker i, send an update to the manager, send a dummy task to worker i, wait for worker i to receive and run the dummy task, and forward worker i's subsequent steal request to the manager to make sure the condition for termination is met again. The first three operations take 15–20 microseconds, as we have seen in Figure 4.8 (b). The remaining 38–43 microseconds give an idea of the minimum time required to detect termination after a task barrier. For comparison, Figure 4.9 (a) also includes the performance of Intel OpenMP. The combined thread/task barrier incurs more overhead than a pure task barrier, but is still less expensive than explicit communication.

Figure 4.9 (b) shows the performance impact of worker-manager communication in a more realistic setting. We pick the LU factorization benchmark because it can be written with two task barriers per iteration, so efficient synchronization is of key importance. Despite frequent communication, the increase in runtime is just between 0.5% and 5%, demonstrating that channel communication has a reasonable cost.

4.3 From Tasks to Futures

In general, tasks may have arbitrary dependencies that must be respected. As in [166], Section 2.1, page 39, we will not distinguish between control and data dependencies and simply say that task B depends on task A if A must precede B, either because of A's side effects, or because A produces data that B consumes. Formally, A precedes B is written $A \prec B$, which tells us that A and B are ordered and forbidden to execute in parallel.

4.3.1 Channel-based Futures

Consider a simple Fibonacci-like tree recursion (without the base case):

```
int x = spawn f(n-1); // Create task for f(n-1)
int y = f(n-2);       // Proceed recursively with f(n-2)
sync;                 // Wait until result of f(n-1) is available
return x + y;
```

Here, a task depends on the result of its child task, which in turn depends on the result of its child task, and so on. Cilk and OpenMP provide constructs to suspend a task until its children have finished execution. The same can be achieved with futures:

```
future fx = FUTURE(f, n-1); // Create a future for f(n-1)
int y = f(n-2);             // Proceed recursively with f(n-2)
int x = AWAIT(fx, int);     // Wait for future's result
return x + y;
```

The important insight is that futures can be viewed as channels: a future opens a channel over which the result will be delivered. Setting the value of a future is equivalent to sending the value to the channel. Forcing a future is equivalent to receiving the value from the channel. When the value is needed, it is simply received from the channel, blocking the receiver until the value is determined.

Creating a future for `f(n-1)` involves allocating a channel, creating a task, and storing a reference to the channel in the task descriptor. The latter is taken care of by `ASYNC` (cf. lines 28–36 in Listing 2.6):

```
Channel *ch = channel_alloc(sizeof(int), 1, SPSC);
ASYNC(f, n-1, ch);
```

The channel should be buffered (capacity > 0) to avoid the possibility of a blocking send when a worker uses the channel reference after evaluating `f(n-1)`. This means that we cannot use `ASYNC_DECL` to generate the task function for `f` because it would insert code to dereference `ch` (cf. lines 12–20 in Listing 2.6). We need a modified version, `FUTURE_DECL`, that inserts a call to `channel_send` instead:

```
// At the end of the task function
int tmp = f(n-1);
channel_send(ch, &tmp, sizeof(int));
```

Before the future's result can be used, it must be received from the channel. Until the value is available, the task is suspended:

```
while (!channel_receive(ch, &x, sizeof(int))) suspend();
channel_free(ch);
```

This is known as data flow synchronization: waiting for data to become available, rather than waiting for a task to finish execution. While a thread is blocked on a future, it can try to schedule other work by calling back into the runtime system:

```
rts_force_future(ch, &x, sizeof(int));
channel_free(ch);
```

In this case, the runtime system takes care of receiving a value from channel `ch`. Finally, by hiding channels behind a `future` type, we arrive at the macros that we introduced back in Section 2.4.1:

```
future fx = FUTURE(f, n-1);
```

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "tasking.h"

int sum(int a, int b)
{
    return a + b;
}

FUTURE_DECL(int, sum, int a; int b, a, b);

int main(void)
{
    int s, a = 4, b = 2;

    TASKING_INIT();

    future f = FUTURE(sum, a, b);

    s = AWAIT(f, int);

    assert(s == 6);

    TASKING_EXIT();

    return 0;
}
```

Listing 4.2: A minimal task-parallel program with future-based synchronization.

```
...
int x = AWAIT(fx, int);
```

Listing 4.2 repeats the toy example from Section 2.4, replacing the task barrier with future-based synchronization. Note the use of `FUTURE_DECL` in the declaration of `sum`. With help from the compiler, synchronization could be made implicit by figuring out when a future's result is needed and forcing it upon first touch.

Listing 4.3, which shows Listing 4.2 after macro expansion, reveals the underlying channel operations. After creation (line 39), the future is stored in the task descriptor (line 43) and later retrieved to send the result (line 27). Forcing a future translates into a call to `rts_force_future` followed by freeing the associated channel. Note the use of wrapper functions that act as getters and setters for channels. A future is a handle to a channel and may contain different data, depending on how channels are passed between workers. On the SCC, for example, we used a pair of integers[3] to identify a channel [201], hence the need for converting a "portable reference" to a regular `Channel *` and vice versa. Shared-memory futures are raw pointers to channels and can be cast as such.

[3](ID of channel owner, byte offset into owner's message passing buffer)

```c
int sum(int a, int b)
{
    return a + b;
}

// FUTURE_DECL expands to a data structure to hold the task’s arguments,
struct sum_task_data {
    int a; int b; future f;
};

// a function to allocate a future/channel,
static inline future make_sum_future(void)
{
    future f;
    channel_set(&f, __channel_alloc_impl__3(sizeof(int), 1, SPSC));
    return f;
}

// and a task function that wraps the call to sum
void sum_task_func(struct sum_task_data *d)
{
    typeof((d)->a) a = (d)->a;
    typeof((d)->b) b = (d)->b;
    typeof((d)->f) f = (d)->f;

    int tmp = sum(a, b);
    channel_send(channel_get(f), &tmp, sizeof(tmp));
}

int main(void)
{
    int s, a = 4, b = 2;

    tasking_init();

    future f = ({ // FUTURE creates a task, enqueues it, and returns a future
        Task *__task = task_alloc();
        struct sum_task_data *__d;
        future __f = make_sum_future();
        __task->parent = get_current_task();
        __task->fn = (void (*)(void *))sum_task_func;
        __d = (struct sum_task_data *)__task->data;
        *(__d) = (typeof(*(__d))){ a, b, __f };
        rts_push(__task);
        __f;
    });

    s = ({ // AWAIT forces the future and returns its result
        int __tmp;
        rts_force_future(channel_get(f), &__tmp, sizeof(__tmp));
        channel_free(channel_get(f));
        __tmp;
    });

    assert(s == 6);

    tasking_exit_signal();
    tasking_exit();

    return 0;
}
```

Listing 4.3: Program 4.2 after preprocessor macro expansion (abbreviated to make it more readable). Both `FUTURE` and `AWAIT` macros use statement expressions `({...})`, a GNU extension supported by GNU, Clang, and Intel compilers [2].

4.3.2 Futures for Nested Parallelism

What is left to explain is the implementation of `rts_force_future`. Although futures can express more than nested parallelism [228], we experiment with a specialized implementation as sketched in Listing 4.4. This implementation operates under the assumption that if a task creates a future, the future's result will be needed later on. Herlihy et al. have shown that "well-structured futures" incur fewer deviations from sequential execution than general, unstructured futures, resulting in better cache locality, as measured by the number of cache misses [115]. Critical to this is using futures in a disciplined way: making sure that every future is touched only once, either by the task that created it or by a descendant of the task that created it. As a result, a well-structured future is always created prior to being touched. Creating a future and passing it around is certainly possible, but not the use case that `rts_force_future` is trying to address, namely that of structured local-touch computations [115].

When forcing a future, we first check if the future's result is already computed, and if so, just return (lines 5–6). If not, we try to resolve the future by running all child tasks of the current task, until the future's result is available or no child tasks are left (lines 8–12). Finally, we have to assume that the corresponding task has been stolen and switch to work stealing (lines 14–22). We can safely return from `rts_force_future` in line 19 because `send_steal_request` preserves the invariant of one pending steal request per worker and will not generate a new request until the thief has successfully received a task and cleared the channel.

To summarize, there are three possibilities: (1) The future has been evaluated in parallel, and its result can be received. (2) The future has not been started yet, in which case it is evaluated sequentially. (3) The future is being evaluated by another worker, in which case other work is picked up until the result can be received.

Forcing a future may have the side effect of evaluating other futures. For example, imagine a worker creates three futures f_1, f_2, and f_3, in this order, pushing each task onto the bottom of its deque, and then forces f_2, which we assume has not been stolen. Because f_3's task sits on top of f_2's task in the worker's deque, `rts_force_future` evaluates f_3 before it evaluates f_2, with the result that a subsequent touch of f_3 will immediately return its value. Again, this is only reasonable if every future will be touched, and there is no priority involved. It would be undesirable to evaluate f_3 if its result were not needed, or if f_2 had a higher priority. A more flexible implementation would have to defer f_3's task when touching f_2.

Unrestricted work stealing in `rts_force_future` cannot guarantee that a task will return from the function as soon as the value it is waiting for is available, leading

```c
void rts_force_future(Channel *chan, void *data, unsigned int size)
{
    Task *task;

    if (channel_receive(chan, data, size))
        return;

    while ((task = pop_child()) != NULL) {
        run_task(task);
        if (channel_receive(chan, data, size))
            return;
    }

    while (!channel_receive(chan, data, size)) {
        send_steal_request();
        while (!channel_receive(chan_tasks, (void *)&task, sizeof(task))) {
            handle_steal_requests();
            if (channel_receive(chan, data, size))
                return;
        }
        run_task(task);
    }
}
```

Listing 4.4: Forcing a future involves channel communication.

to "buried joins" [87] and associated stack pressure through deep recursion: waiting for a future may depend on completing stolen work, which in turn may depend on completing stolen work, and so on [147]. For this reason, some implementations prefer to use leapfrogging [251, 86] or other forms of depth-restricted work stealing [205, 33], but risk losing parallelism and thus performance[4] [232]. Van Dijk et al. have measured stack depths for the UTS benchmark with input T3L using 48 workers on an AMD Opteron multiprocessor, which comes close to our configuration [247]. They found that unrestricted stealing creates 36%–223% deeper stacks than leapfrogging, requiring between 0.3 and 1 MB of additional memory per worker.

4.4 Efficient Fork/Join Parallelism

Divide-and-conquer algorithms lend themselves to parallel execution: when a problem is recursively divided into subproblems that can be solved independently, the number of tasks after a few recursive steps provides plenty of opportunity for parallelism.

4.4.1 Cilk-style Fork/Join

Figure 4.10 shows an instance of a parallel divide-and-conquer algorithm. Until the problem is small enough to be solved directly (lines 1–2), it is subdivided into smaller

[4]Intel TBB has since moved away from depth-restricted to unrestricted work stealing [159, 160].

```
SOLVE($problem$)
1 if $problem$ is small enough
2     return SOLUTION($problem$)
3 else
4     $problem_i, problem_j$ = DIVIDE($problem$)
5     $solution_i$ = fork SOLVE($problem_i$)
6     $solution_j$ = SOLVE($problem_j$)
7     join $solution_i$
8     $solution$ = COMBINE($solution_i, solution_j$)
9     return $solution$
```

Figure 4.10: Pseudocode of a parallel divide-and-conquer algorithm, in which a problem is recursively divided into subproblems until the problems are small enough to be solved directly. In general, a problem is divided into $k \geq 2$ subproblems.

problems (lines 3–4), which are solved in parallel (lines 5–6). Synchronization is needed (line 7) to combine the partial solutions into a total solution (lines 8–9).

Fork/join as a way of structuring parallel programs has been around for some time [68]. Generally speaking, every program that forks into two or more concurrent threads of execution and later joins (combines) the threads' results is following the fork/join-model of execution. Cilk has shown that forking tasks instead of threads and employing workers to execute tasks enables efficient implementation of parallel divide-and-conquer algorithms.

Figure 4.11 shows the typical structure of a task graph created by Cilk-style fork/join. The proper nesting of tasks and child tasks is the result of strict synchronization: a procedure that creates tasks is not allowed to return until it has joined all of its children. If absent, joins are inserted or performed implicitly before every return. A join is usually collective and synchronizes with every child spawned in the parent, rather than allowing fine-grained control over which child to join and in which order.

The properties of strict fork/join are summarized concisely in Halpern's proposal for including task-parallel constructs in future revisions of the C++ standard [103]:

- Tasks can create child tasks, which may execute in parallel with other tasks and the tasks that created them.
- A task can wait for its children to complete. A task *cannot* synchronize execution with any other task that is not its child.
- A task is not allowed to return until all of its children have returned. This restriction

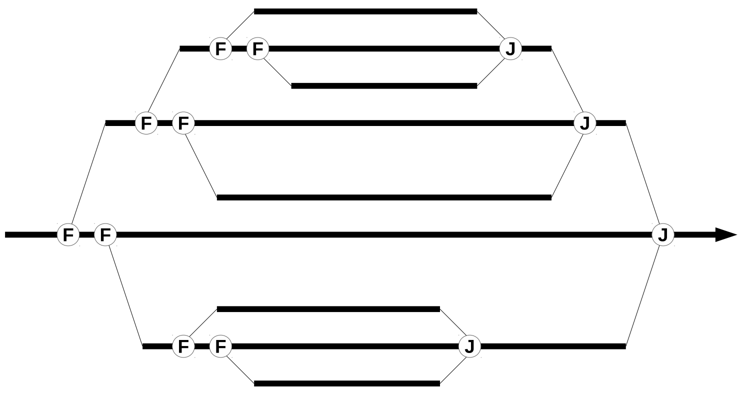


Figure 4.11: Example task graph of a multithreaded program based on Cilk-style fork/join. Child tasks are properly nested and joined collectively with their parents. This style of fork/join is sometimes called spawn/sync in reference to the Cilk keywords of the same name (see, for example, [221], pp. 116–118).

guarantees proper nesting of tasks as illustrated in Figure 4.11 and, by extension, safe access to variables in the parent's stack frame,[5] assuming, of course, that workers share the same address space.

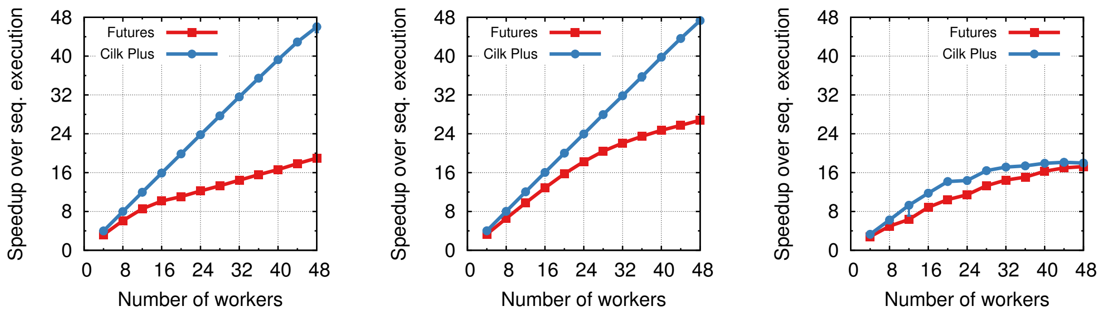
4.4.2 Fork/Join with Futures

Futures can express the same computations as Cilk's **`spawn`** and **`sync`** constructs [115]. Every **`spawn`** creates a future, and a **`sync`** is equivalent to forcing every future created by a task. Ideally, we want the resulting programs to be as efficient as their Cilk counterparts. Figure 4.12 shows the performance of our scheduler on three benchmarks with strict fork/join parallelism: Treerec, N-Queens, and Cilksort. Except on Cilksort, where the difference is less pronounced, future-based synchronization comes with a high cost: with 48 threads, Cilk Plus is 2.4 times faster on Treerec and 1.76 times faster on N-Queens than our scheduler. Performance profiling of N-Queens identifies `__channel_alloc_impl__3` as a hot spot (cf. line 15 in Listing 4.3) [11]. Our implementation spends a significant amount of execution time allocating channels as a result of creating futures[6]. The program that finds all solutions to the 14-Queens problem unfolds into 27 358 552 tasks, each of which allocates (and frees) a future/channel, averaging around 10.5 million channels per second when using 48 threads.

Reusing memory that was allocated from the heap by caching channels in per-thread free lists is an effective way to improve performance as shown in Figure 4.13. With most heap allocations gone, overall performance is much closer to that of Cilk Plus.

[5]Referring to the lifetime of objects, not implying that access is exempt from coordination.

[6]According to the profiler [11], the heap itself is not the bottleneck, that is, heap contention is low.



(a) Tree recursion with $n = 34$ and $t = 1\mu s$

(b) N-Queens 14

(c) Cilksort of 100 million integers

Figure 4.12: Performance of channel-based futures in our runtime system compared to Intel Cilk Plus. Cilksort was run under `numactl --interleave=all`, which makes sure that memory is allocated evenly among all NUMA nodes. (ICC 14.0.1, `-O2`, AMD Opteron multiprocessor)

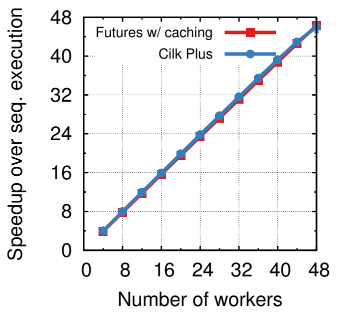
Our scheduler has caught up on Treerec and remains 8% behind on N-Queens with 48 threads—a reasonable result considering that all communication happens through channels. Note that the definitions of `FUTURE_DECL` and `AWAIT` must be modified to accommodate the use of free lists (cf. lines 15 and 51 in Listing 4.3).

Despite their good performance, futures are less convenient than **`spawn`** and **`sync`** when joining more than a few child tasks. Consider the body of a recursive function `f` that creates `n` child tasks and eventually collects their results:

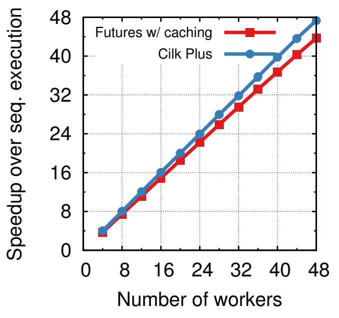
```
future children[n];
int result = 0;
for (i = 0; i < n; i++) {
    children[i] = FUTURE(f, depth-1);
}
...
for (i = 0; i < n; i++) {
    result += AWAIT(children[i], int);
}
```

Such a function appears in N-Queens, for example, where a task spawns up to N child tasks depending on how freely the next queen can be placed on the current board. Using **`spawn`** and **`sync`** instead of futures removes bookkeeping and better expresses intent. A single **`sync`** statement replaces up to N invocations of `AWAIT`:

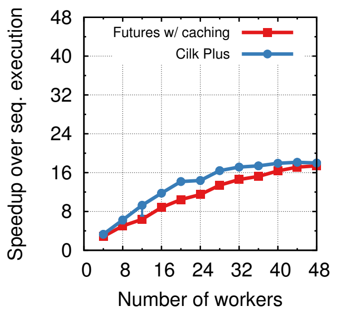
```
int results[n], result = 0;
for (i = 0; i < n; i++) {
```



(a) Tree recursion with $n = 34$ and $t = 1\mu s$



(b) N-Queens 14



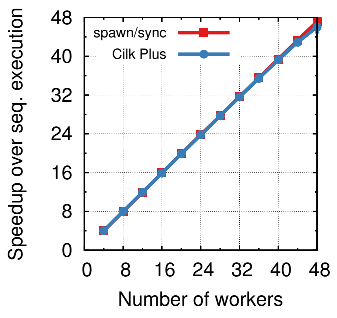
(c) Cilksort of 100 million integers

Figure 4.13: Improved performance of channel-based futures in our runtime system compared to Intel Cilk Plus. Cilksort was run under `numactl --interleave=all`, which makes sure that memory is allocated evenly among all NUMA nodes. (ICC 14.0.1, `-O2`, AMD Opteron multiprocessor)

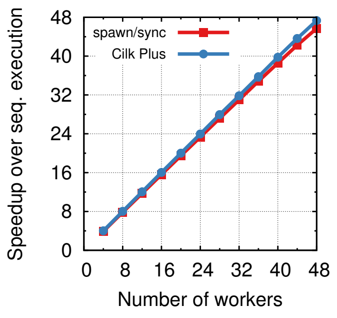
```
        results[i] = spawn f(depth-1);
    }
    ...
    sync;
    for (i = 0; i < n; i++) {
        result += results[i];
    }
```

We have implemented **`spawn`** and **`sync`** macros to experiment with an implementation that trades off future handles for collective joins. This implementation takes advantage of one specific property of strict fork/join parallelism, namely that child tasks may access variables in their parents' stack frames, provided that workers share the same address space (see above). Instead of using a channel, a parent may then pass a reference to a local variable to its child, which will hold the child's result once written. Return values are no longer sent over channels to be received by parents, but stored directly in the parents' stack frames. We can understand this implementation as another specialization, whereby one-shot channels are replaced with synchronization variables that can be written exactly once [46, 157].

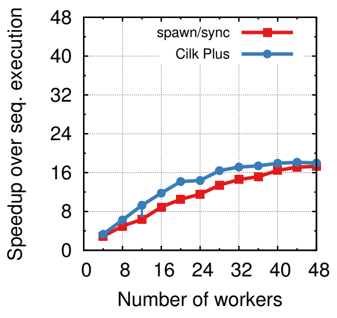
A **`sync`** must wait for the completion of all children of the current task. We implement this construct with the help of atomic join counters [47]. Every task keeps a counter, which it increments when it creates a child, and which is decremented when a child returns. A count of zero indicates no pending children, allowing the parent to return from **`sync`**. Some implementations of OpenMP's `taskwait` work in the same way



(a) Tree recursion with $n = 34$ and $t = 1\mu s$



(b) N-Queens 14



(c) Cilksort of 100 million integers

Figure 4.14: Performance of Cilk-like spawn and sync constructs in our runtime system compared to Intel Cilk Plus. Cilksort was run under `numactl --interleave=all`, which makes sure that memory is allocated evenly among all NUMA nodes. (ICC 14.0.1, `-O2`, AMD Opteron multiprocessor)

[241, 138]. It is worth noting that **`sync`** is oblivious to the grandchildren of a task. The operation of a deep **`sync`** that waits until all transitively spawned tasks have finished execution depends on child tasks to decrement a join counter only after all their own children have finished execution. The last child that decrements a join counter is then responsible for decrementing its ancestors' join counters further up the tree [241]. Cilk enforces strictness by automatically inserting a **`sync`** at the end of every procedure that contains a **`spawn`**. Otherwise, in the absence of synchronization, tasks may outlive their ancestors, in which case care must be taken to avoid dangling references.

Figure 4.14 shows that **`spawn`** and **`sync`** can help improve performance when the full generality of futures is not needed. Our scheduler is now slightly faster than Cilk Plus on Treerec. Using 48 threads, Cilk Plus comes out ahead with 3% better performance on N-Queens and 4% better performance on Cilksort. We conclude that channel communication among worker threads does not hinder efficiency when scheduling fork/join computations. The results suggest that our scheduler can compete with Cilk Plus, a mature work-stealing scheduler that has been carefully designed and optimized for parallel divide-and-conquer algorithms.

One of the goals of Cilk was to address the shortcomings of using futures for fine-grained parallelism [104, 105]. Whether performance should be counted among the shortcomings is debatable. Our experiments so far have not borne out that futures are only suitable for coarse-grained parallelism.

We could further optimize our implementation of futures to avoid channel communication in the common case, namely when forcing a future that has not been stolen. In fact, sequential execution is so common that it accounts for 96–99.9% of all future-related channel operations in Treerec, N-Queens, and Cilksort (Figure 4.13). In other words, workers mostly communicate with themselves, using channels to return values from functions. To avoid this needless overhead, we could associate a future with a reference to a local variable and replace this reference with a channel only when the task is stolen. We leave this optimization for future work. Special care must be taken if a worker is allowed to create a future and pass it on to another worker, in which case channel communication cannot be elided.

4.5 Summary

This chapter has shown how termination can be derived from asynchronous steal requests without additional control messages. The algorithm we described keeps track of steal requests rather than workers as they run out of work and become idle. A task barrier that requires all tasks to finish can await termination before letting execution continue. Futures are easily expressed in terms of channels, allowing fine-grained synchronization between tasks as well as strict fork/join parallelism in the style of Cilk. Some overhead remains, but there is room for optimization.

Besides futures, there are other synchronization constructs that could leverage channels, such as semaphores, synchronization variables [46, 69], and cyclic barriers [193, 217]. Channels are useful building blocks not only for sending tasks between workers, but also for exchanging data between tasks.

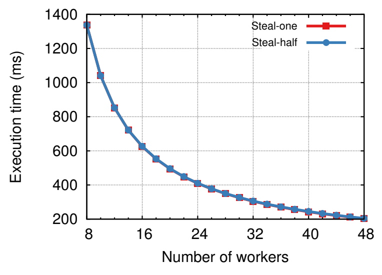
5 | Scheduling Fine-grained Parallelism

Work stealing has evolved into the scheduling technique of choice for fine-grained task parallelism. While there are still many systems in use that are well served by central task pools, core counts continue to increase, and runtime systems are expected to take advantage of the available hardware. Recall, for example, the experiment in Figure 2.2, which showed the limited scalability of a central task pool compared to a distributed task pool. But work stealing alone does not guarantee efficient scheduling. As a rule of thumb, the fewer steals are necessary, the better the performance, which is why victim selection and stealing strategies receive much emphasis in the design and implementation of work stealing algorithms. There are many parameters to tune, especially when scheduling fine-grained parallelism.

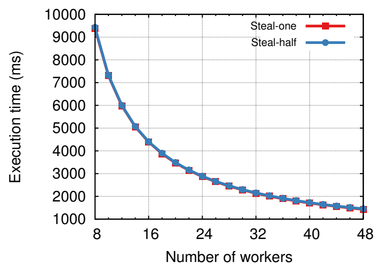
In Section 5.1, we see that steal-half is not always beneficial. It turns out a better default is to let the runtime system decide which strategy to use. We propose an adaptive strategy that is able to switch between steal-one and steal-half as needed. Section 5.2 shows the difficulty of creating many fine-grained tasks. Similar tasks, such as iterations of a parallel loop, can be combined and split at runtime to increase the granularity of parallel work. We focus on Lazy Binary Splitting (LBS), which promises robust performance without parameter tuning, unlike other strategies that are sensitive to the choice of chunk size. Section 5.3 discusses different splitting strategies and demonstrates their performance. We also show that LBS enables efficient loop scheduling with performance close to dedicated loop schedulers.

5.1 Adaptive Work Stealing

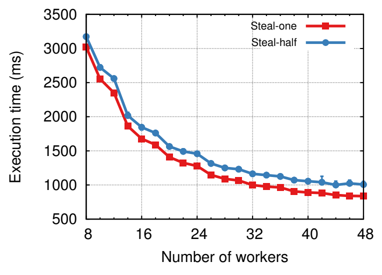
We have implemented the two most common work-stealing strategies: steal-one—stealing a single task from a victim—and steal-half—stealing half of a victim's tasks. Steal-half is itself an adaptive strategy. The more tasks a worker has enqueued, the more tasks are returned by a successful steal. Conversely, if tasks are rare, steal-half may return single tasks to the same effect as using steal-one.



(a) Tree recursion with $n = 34$ and $t = 1\mu s$



(b) N-Queens 14



(c) Cilksort of 100 million integers

Figure 5.1: The work-stealing strategy makes little to no difference in performance when scheduling divide-and-conquer algorithms. Tasks are synchronized using futures; associated channels are kept in free lists rather than returned to the heap, for reasons discussed in the previous chapter. Cilksort was run under `numactl --interleave=all`, which makes sure that memory is allocated evenly among all NUMA nodes. (GCC 4.9.1, `-O3`, AMD Opteron multiprocessor)

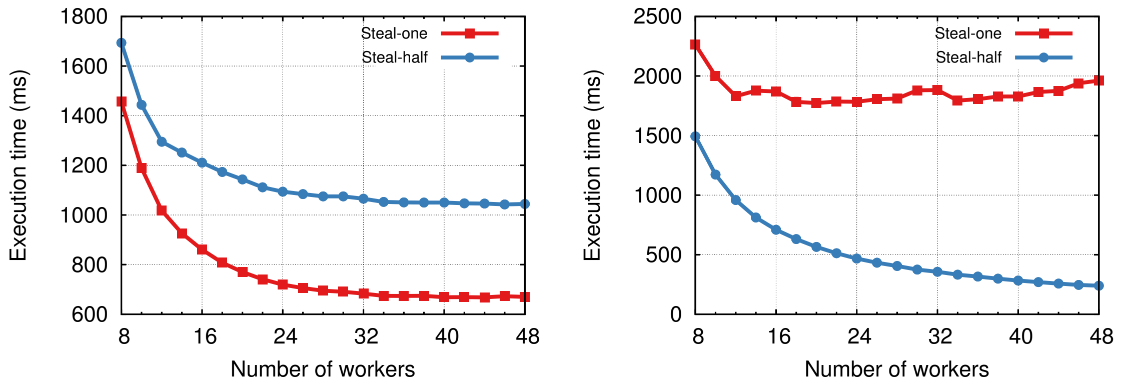
5.1.1 Choosing Between Steal-One and Steal-Half

In section 3.5, we have seen that steal-half is preferable to steal-one when workers need to deal out large numbers of tasks. As a rule of thumb, shallow task trees lead to frequent load balancing, which is bound to become a bottleneck unless the overhead is amortized by stealing tasks in batches. Parallel divide-and-conquer algorithms, on the other hand, are typically scheduled using steal-one because stealing the oldest task already yields a significant portion of the work[1]. As a result, steals are infrequent, leaving little room for improvement by choosing a strategy such as steal-half.

Consider, for example, the benchmarks with which we measured fork/join performance towards the end of the previous chapter. For better comparison with Cilk Plus, we configured our scheduler to use steal-one. Figure 5.1 attests that there is no advantage to be gained from using steal-half. In fact, performance is virtually identical except for Cilksort, where steal-half is consistently 150–180 milliseconds slower than steal-one, accounting for a 20% difference in performance with 48 threads.

We might be tempted to conclude that steal-half would be a good default strategy, seeing that it outperforms steal-one on flat parallelism without losing too much on nested parallelism. But let us look at another benchmark. Figure 5.2 shows the result of running BPC with two very different inputs. In Figure 5.2 (a), parallelism is exposed in short bursts of task creation, as every producer task creates only nine consumer tasks.

[1]Our discussion assumes the classic work-stealing algorithm in which workers treat their deques as stacks, but steal tasks from other deques in FIFO order.



(a) BPC with $d = 100\,000$, $n = 9$, and $t = 10\mu\text{s}$ **(b)** BPC with $d = 1$, $n = 999\,999$, and $t = 10\mu\text{s}$

Figure 5.2: The choice of work-stealing strategy may depend on input values, which are not known until runtime. In (a), tasks are created in small numbers at a time—nine consumer tasks for each of the 100 000 producer tasks. In (b), a single producer task creates 999 999 consumer tasks, all in sequence. (GCC 4.9.1, `-O3`, AMD Opteron multiprocessor)

Figure 5.2 (b) presents the exact opposite in terms of task creation: a single producer task creates the entirety of consumer tasks, turning BPC into SPC, with unsurprisingly similar results to Figure 3.9 (a). Forty-eight threads send an order of magnitude more steal requests when stealing single tasks compared to stealing in batches. On top of that, steal-one generates an average of 22.5 failed steals per request, a number that can be reduced by last-victim selection, but not without affecting performance in other ways. (Refer back to Section 3.4.2 for a detailed discussion of last-victim selection.) Regardless of victim selection, steal-half ends up being much more efficient: a steal succeeds within two attempts and moves an average of 35 tasks.

The situation is different with multiple producer tasks and limited parallelism at each stage. First of all, steal-half provides little benefit over steal-one when no worker has more than a few tasks to spare. In fact, steal-half only manages to move an average of 1.5 tasks, yet this seemingly small difference between steal-half and steal-one amounts to 11% more steal requests, together with a 66% increase in the number of failed attempts before tasks are discovered. Steal-half ends up being less efficient than steal-one because steal-half increases the need for rebalancing; being greedy and stealing multiple tasks whenever possible may actually interfere with load balancing and hurt performance.

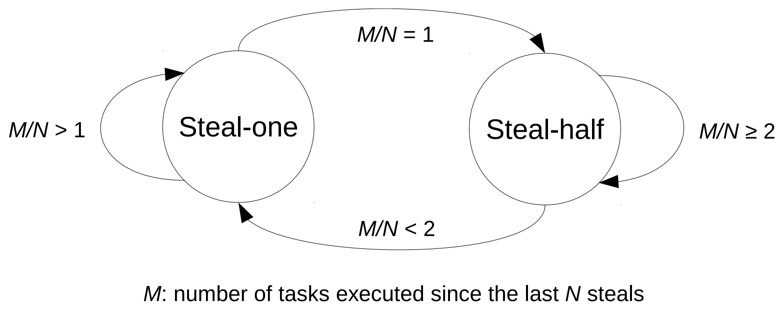


Figure 5.3: State diagram showing the conditions for switching between steal-one and steal-half after every N steals. The smaller the value of N, the more transitions may occur. The quality of a strategy is inversely proportional to the frequency of work stealing. Initially, workers have to start with either steal-one or steal-half.

5.1.2 Adapting the Choice at Runtime

Choosing a strategy for every instance of a problem is tedious, let alone finding the strategy that works best. Moreover, it is possible for an application to create different task graphs over the course of several parallel phases. A single strategy, be it steal-one or steal-half, may not be able to schedule all available parallelism efficiently, giving rise to suboptimal performance.

If we assume that neither steal-one nor steal-half is strictly better than the other, we might be able to get the best of both worlds by letting the runtime system decide which strategy to use under what circumstances. This implies that workers switch between steal-one and steal-half, depending on which strategy is deemed more effective. Implementation-wise, the chosen strategy is stored in a binary flag that is added to the steal request structure definition. A victim reads the value of the flag to perform the desired steal on behalf of the thief. The main question that remains is, how should workers decide which strategy to use?

We follow a simple heuristic: Prefer steal-one to steal-half when parallelism is limited or created recursively; otherwise choose steal-half. Put the other way round, prefer steal-half to steal-one when steals are frequent despite abundant potential parallelism[2].

Figure 5.3 illustrates the process of choosing a stealing strategy. A worker is in one of two states corresponding to steal-one and steal-half. Depending on the outcome of the previous N steals, a worker may pursue a different strategy or keep using the current strategy for the next N steals.

Frequent stealing is an indication that a better strategy is needed. How do we

[2]Cilk and its descendants are built on the assumption that, given sufficient parallelism, steals are rare. It is thus understandable that Cilk-style work stealing lacks support for steal-half.

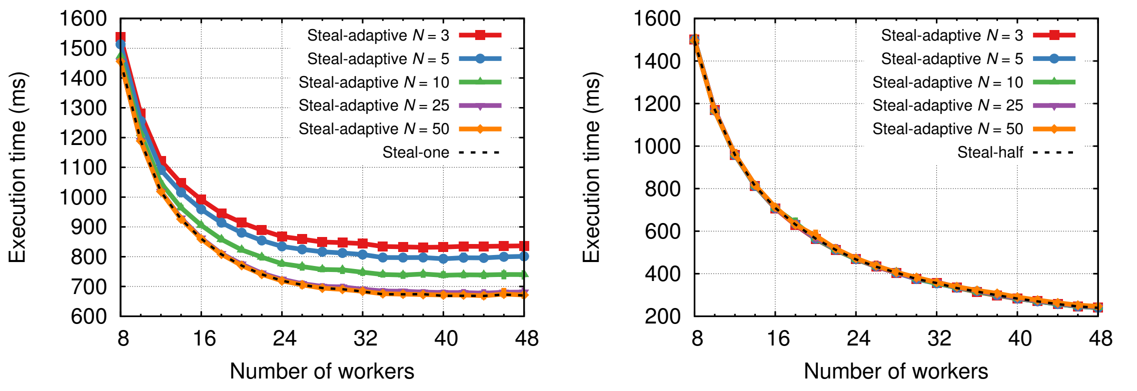
define “frequent”? Our main concern is fine-grained parallelism—tasks that are small enough that efficient scheduling matters. Coarse-grained parallelism tends to dwarf the time spent scheduling, to the point that work stealing may have little effect on overall performance. Fine-grained parallelism, in contrast, is highly sensitive to the choices made at runtime. In general, the fewer the number of steals, the better the performance, provided that the work remains balanced throughout the computation. That is to say, performance benefits from a high ratio of executed tasks to steals (not stolen tasks!). The smaller the ratio, however, the stronger the case for a change of strategy to try to reduce the work-stealing overhead.

Returning to Figure 5.3, we see that every transition is determined by the ratio of executed tasks to steals. For a worker to calculate this ratio, the interval is limited to the last N steals, during which a worker executes M tasks. A transition from steal-one to steal-half as a result of $M/N = 1$ usually means that all M tasks had to be stolen, or that other workers were quick to steal every task that was created recursively. Similarly, a transition from steal-half to steal-one as a result of $M/N < 2$ means that steal-half has failed to reduce the work-stealing overhead by averaging less than two tasks per steal. A worker takes $M/N \geq 2$ as a sign that enough parallelism exists to be able to benefit from steal-half. It may well be the case that some workers continue to use steal-half, while others switch to steal-one, or vice versa. This helps avoid situations like the motivating example above, where there is not enough parallelism for every worker to steal more tasks than needed.

The value of N determines the number of steals and, as such, the length of the interval after which a worker reevaluates its stealing strategy. The smaller the value of N, the more transitions may occur. If steals are frequent, intervals tend to be short, allowing workers to adapt their strategies continually. If steals are rare, intervals tend to be longer, as workers are busy running tasks until the next load imbalance arises.

5.1.3 Performance

Figure 5.4 shows how adaptive work stealing compares to the best-performing strategies from Figure 5.2. Before we discuss the results, it is time to fill in the last blank in our algorithm, namely the choice of initial strategy. Referring back to Figure 5.3, workers have the choice of starting with steal-one or steal-half. In fact, different workers might start with different strategies, should some workloads warrant it. In our tests, at least, it does not matter whether workers start with the one or the other; performance is identical. For presentation, we pick the results of starting with steal-one and omit the other results to avoid duplication.



(a) BPC with $d = 100\,000$, $n = 9$, and $t = 10\mu$s **(b)** BPC with $d = 1$, $n = 999\,999$, and $t = 10\mu$s

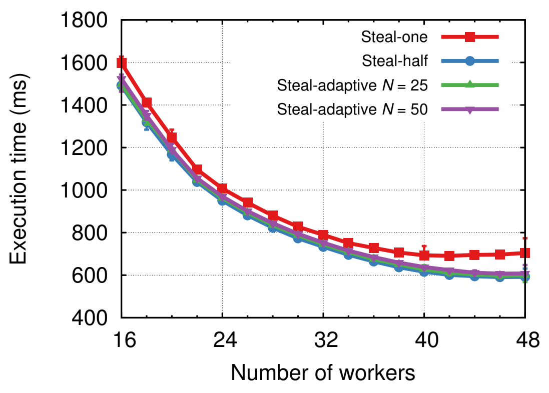
Figure 5.4: Adaptive work stealing versus the best-performing strategy for each of the two workloads from Figure 5.2. The value of N is the number of steals after which a worker reevaluates its stealing strategy (see Figure 5.3). Whether workers start out with steal-one or steal-half does not make a measurable difference, so we omit the latter. (GCC 4.9.1, `-O3`, AMD Opteron multiprocessor)

Most importantly, adaptive work stealing can match steal-one in Figure 5.4 (a) and steal-half in Figure 5.4 (b). The latter results suggest that any reasonably small value for N serves to approximate steal-half; larger values (>100) slowly start to affect performance (not included in the figure). With 48 threads, there is practically no difference between $N = 3$ and $N = 50$. Workers rely on steal-half 93.2% of the time, $\pm 0.4\%$ for $N = 3$ and $\pm 0.7\%$ for $N = 50$ (standard error of the mean).

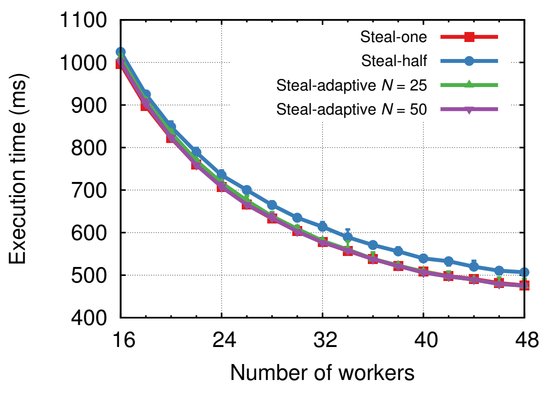
Figure 5.4 (a) highlights the importance of choosing a good value for N. Among the plotted values, only $N = 25$ and $N = 50$ lead to performance on par with steal-one. Smaller values increase the chance of switching to steal-half, given the workload's 9:1 ratio of consumer to producer tasks. For $N = 3$, workers fall back to steal-half $37.1\% \pm 0.02\%$ of the time (48 threads). As we increase N, it becomes clear that steal-one is winning over steal-half. For $N = 5$ and $N = 10$, the percentage of steal-half drops to $28.2\% \pm 0.02\%$ and further to $14.6\% \pm 0.03\%$. A value of 25 is large enough that workers dismiss steal-half and opt for steal-one $98.4\% \pm 0.02\%$ of the time.

To summarize Figure 5.4, the number of steals between intervals should be neither too large nor too small. Workers need to base their decisions on more than a few steals, but at the same time be quick to adapt when a strategy turns out to be inefficient.

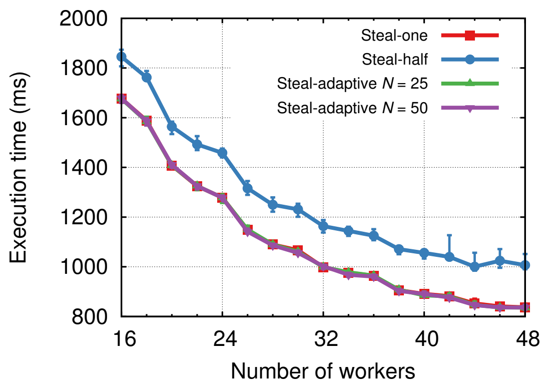
Figure 5.5 shows four more benchmarks where the right strategy makes some, albeit small, difference in performance. Matrix multiplication and UTS benefit from steal-half; LU decomposition and Cilksort benefit from steal-one. The results make



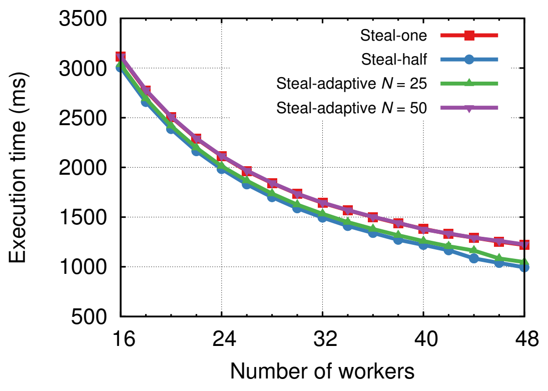
(a) Multiplication of two 2048 × 2048 matrices partitioned into blocks of 32 × 32 elements



(b) LU factorization of a 4096 × 4096 matrix partitioned into blocks of 64 × 64 elements



(c) Cilksort of 100 million integers



(d) UTS with tree T3L

Figure 5.5: Adaptive work stealing combines steal-one and steal-half to select the better-performing strategy at runtime. Matrix multiplication and LU factorization were configured to use “last-victim-first” instead of random victim selection. (GCC 4.9.1, `-O3`, AMD Opteron multiprocessor)

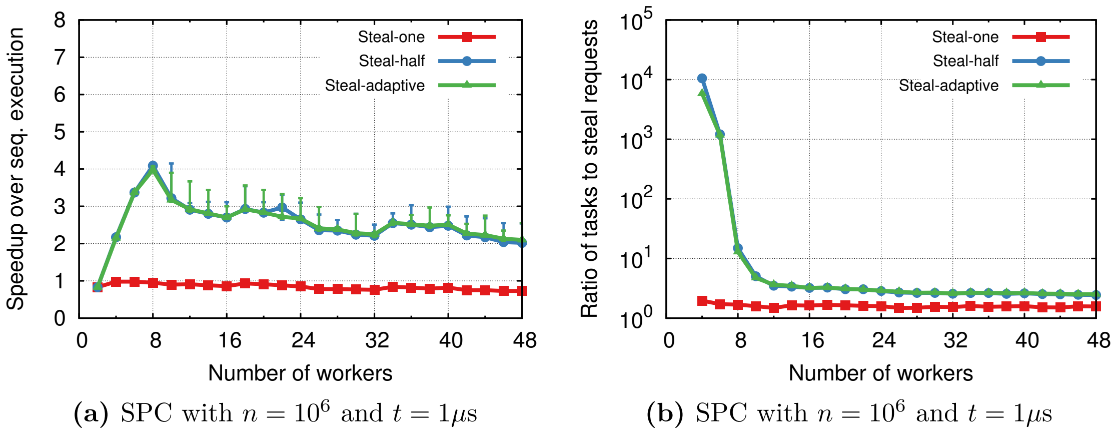


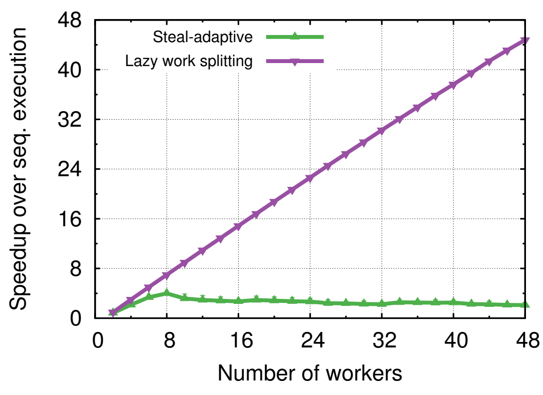
Figure 5.6: Creating a large number of very fine-grained tasks poses a problem to either stealing strategy. (GCC 4.9.1, `-O3`, AMD Opteron multiprocessor)

us confident: steal-adaptive manages to combine the best of both strategies, relatively independent of the value of N, save for UTS, where a larger N means fewer chances for steal-half as workers err on the side of steal-one. With $N = 25$, workers are led to choose steal-half 20.8% $\pm$ 0.98% of the time, in stark contrast to less than one percent with $N = 50$ (48 threads). The problem is not so much that steal-one is inefficient (UTS includes recursive task creation), but rather that steal-half is sometimes preferable, hence the need for shorter intervals to take advantage of steal-half, if only temporarily. We will use $N = 25$ for all remaining experiments.

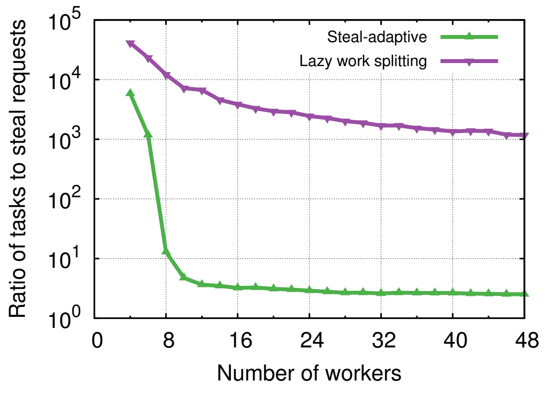
5.2 The Case for Splittable Tasks

Stealing multiple tasks is often a runtime system's primary means to amortize work stealing overheads. We have seen examples where steal-half improves load distribution and overall performance, but we also remarked that, without sufficient parallelism, steal-half may offer little advantage over steal-one.

Let us revisit the SPC benchmark from Section 3.5. Figure 5.6 (a) shows the result of decreasing the task length to one microsecond. Now even steal-half fails to scale beyond eight workers. Adaptive stealing can switch to steal-half, but cannot improve its performance. The average ratio of tasks to steal requests plotted in Figure 5.6 (b) gives an idea of the scheduling overhead and hints at decreasing batch sizes, which limit the distribution of work. While private deques reduce the task creation overhead, they cannot eliminate it. A worker executing the equivalent of



(a) SPC with $n = 10^6$ and $t = 1\mu s$



(b) SPC with $n = 10^6$ and $t = 1\mu s$

Figure 5.7: Creating a single task and splitting it into smaller tasks is much more efficient than creating and scheduling a large number of tasks. (GCC 4.9.1, `-O3`, AMD Opteron multiprocessor)

```
for (i = 0; i < N; i++)
    spawn f(i);
```

risks creating too much parallelism to be useful or too little parallelism to be able to schedule efficiently, depending on the number of iterations and the granularity of `f`. In case of fine-grained parallelism, it may help to create tasks in parallel [239] or offload task creation to a subset of the workers [121]. Ideally, however, a worker should create only as much parallelism as needed to minimize runtime overheads.

5.2.1 Bundling Tasks

When a worker steals a number of tasks, the tasks are assumed to be unrelated. Every task points to a function and contains its own data, reflecting the fact that tasks are created dynamically at runtime. The for-loop above results in N tasks that share both code *and* data except for different values of i. Instead of creating a large number of almost identical tasks, we may create a single task and record that i ranges over $[0, N)$[3]. The resulting task may be *split* into "smaller" tasks, possibly down to single iterations. More importantly, splitting may be *lazy*, meaning that it allows delaying task creation until parallelism is needed. If no parallelism is needed, no tasks are created. This brings us closer to our ideal scheduler that increases the granularity of tasks while maintaining load balance.

Figure 5.7 compares lazy work splitting to adaptive stealing. The difference in

[3] $[a, b)$ denotes the half-open range from a (inclusive) to b (exclusive).

efficiency is striking. At 48 workers, we count 8517 unique steal requests over the course of ten program runs, compared to 3 969 873 for steal-adaptive. Lazy work splitting succeeds in executing most iterations sequentially, without creating tasks. Less than 0.1% of all potential tasks are actually created and stolen. The remaining 99.9% are executed with little runtime overhead.

Naturally, creating and scheduling fewer tasks has an effect on the work overhead T_1/T_S. In Section 2.7.2, we measured a work overhead of 2.42—significantly more than Cilk Plus's 1.37. Lazy work splitting reduces this overhead to 1.03. This is only possible because of granularity control [174]: unless needed for load balancing, tasks are combined in an attempt to reduce scheduling overheads. It is thus no surprise that single-worker execution comes close to sequential execution, in terms of performance and semantics[4].

Much of the work on controlling task granularity in parallel programs is orthogonal to work splitting. Parallel divide-and-conquer algorithms tend to create so many tasks that it is often feasible to apply a cut-off based on recursion depth or estimated task demand [151, 76, 254, 242]. Tasks that are cut off are executed sequentially as part of their parent tasks, which makes them candidates for inlining.

Cong et al.'s work-stealing scheduler, mentioned in Section 3.5.3, supports a form of bundling that involves collecting tasks in a list before pushing the list onto a deque [67]. Thieves steal bundles, but cannot split them into smaller bundles to recover parallelism if needed. Whether a task is bundled is based on the same reasoning as load-based inlining [132]: a deque that contains only few tasks indicates that more parallelism is needed, hence bundles should be small to maintain load balance; a deque that contains many tasks indicates that workers are already busy and unlikely to benefit from more parallelism, hence bundles should be large to reduce task creation and deque overheads. Cong et al.'s adaptive strategy causes the size of a bundle to grow exponentially with the size of a deque, up to some threshold.

5.2.2 The Structure of a Splittable Task

Before we begin to discuss strategies for work splitting, we must first establish the structure of a splittable task. Such a task has two fields to represent the interval $[a, b)$ as well as a field to represent the next value in the range. The latter identifies which task is being worked on. For simplicity, we assume $a, b \in \mathbb{Z}$ and $a < b$ so that $[a, b)$ denotes the $b - a$ tasks between a and b, including a and excluding b. We also

[4]Loop iterations are executed sequentially in program order, regardless of task creation being work-first or help-first.

assume that splitting an interval $[a, b)$ produces two smaller intervals $[a, c)$ and $[c, b)$ with $a < c < b$, that is, splitting requires $b - a > 1$. A task ceases to be splittable when $b - a = 1$, in which case it is treated as a regular task.

Work stealing does not need to distinguish between regular and splittable tasks, to place no restrictions on which tasks can be stolen. In particular, enqueued tasks are not split by steals. (They may be split afterwards.) Workers must be able to determine the task type by inspection. Other than that and the three integers described above, a splittable task has the same fields as a regular task.

We introduce a macro `ASYNC_FOR` that wraps the creation of a splittable task:

```
ASYNC_FOR(f, a, b, args...);
```

creates a task that calls `f(args...)`, passing in [`a`,`b`) internally to enable splitting while running `f`. To replace

```
for (i = 0; i < N; i++)
    ASYNC(f, i);
```

with a splittable task, we would write

```
ASYNC_FOR(f, 0, N);
```

and redefine `f` as

```
void f(void)
{
    long i;
    for_each_task (i) {
        // Inlined body of task
    }
}
```

using another helper macro for iterating and splitting. This is not the best or most convenient interface, owing to the limitations of the C preprocessor, but works well enough for experimentation, and serves to illustrate the code a compiler might generate.

Figure 5.8 depicts the splitting process, which involves cloning the current task, calculating `split`, and adjusting the upper and lower bounds of each task. We assume that workers continue execution where they left off before splitting by making the upper interval `[split,end)` available for stealing. Subtracting `start` from `next` gives the number of tasks that have already been consumed, so in general, not `(end - start) > 1` but `(end - next) > 1` determines whether a task is further splittable.

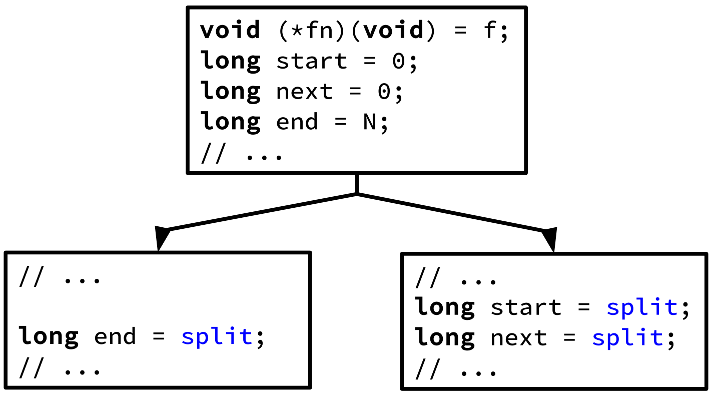


Figure 5.8: The size of a task shrinks as it is split into smaller tasks. The value of `split`, which falls between `next` and `end`, determines the number of tasks in each half.

5.3 Strategies for Work Splitting

In Cilk Plus, a parallel loop, such as

```
cilk_for (i = 0; i < N; i++)
    f(i);
```

is translated into a recursive procedure similar to

```
void loop(long start, long end)
{
    long count = end - start;
    if (count > K) {
        long mid = start + count / 2;
        spawn loop(start, mid);
        loop(mid, end);
    }
    for (long i = start; i < end; i++)
        f(i);
}
```

which splits the iteration range in half until the tasks have reached a predefined chunk size K, equal to some fraction of the number of tasks N divided by the number of workers P [26]. Tzannes et al. call this Eager Binary Splitting (EBS), to emphasize that splitting proceeds eagerly, regardless of how many tasks are actually needed [245]. Creating approximately N/K tasks means that K should be chosen according to the workload. Too many small tasks may overwhelm the runtime system; only few large tasks may cause load imbalance and limit scalability.

Intel's Threading Building Blocks (TBB) adopted a more robust strategy that tries to split an iteration range into $K \cdot P$ chunks, where K is a small constant [209]. In addition, every stolen range is further split into at least V more chunks (if possible), where V is on the order of K. TBB has settled on $K = 4$ and $V = 4$ [209]. This so called auto partitioning derives the chunk size from the number of workers and the fixed parameters K and V, rather than from the number of iterations, but is still eager to create tasks, whether needed or not.

The alternative to EBS is to try to split a task as late as possible, when parallelism is needed to keep workers busy. Tzannes et al. introduced Lazy Binary Splitting (LBS), where a worker does not split unless its deque is empty [245]. As a result, LBS is able to adapt to different workloads, without requiring the programmer or the runtime system to specify a chunk size.

By being lazy, LBS shares some similarities with load-based inlining [132] and lazy task creation [172]. Unlike the former, LBS tends to inline only small tasks before it reevaluates the decision to create parallelism. Unlike the latter, LBS does not create a stealable continuation in preparation for running a task, which explains why the work overhead can drop below that of Cilk Plus. LBS achieves the same effect as indolent closures [231]: it reduces overhead by creating tasks on demand, rather than eagerly whenever a spawn operation is encountered.

We will henceforth use the term work splitting to encompass eager and lazy splitting. EBS is an instance of eager splitting, while LBS is an instance of lazy splitting. Both are work-splitting strategies in which workers, or victims, are responsible for splitting, unlike thieves, which follow the usual work-stealing algorithm.

The opposite approach is to combine splitting with stealing. Orozco et al. propose task queue extensions to support splittable tasks, which they call polytasks [188]. Enqueuing a polytask makes N tasks available with a single push. Dequeuing a (non-empty) polytask does not remove it from the queue, but splits off one task. This is reminiscent of dynamic loop scheduling in OpenMP and raises the same questions about contention and scalability. Hardware task queues [136] or enqueue/dequeue primitives [129, 148] might be useful in this context.

Durand et al. make a case for work-stealing loop schedulers by showing that an adaptive strategy outperforms dynamic and guided schedulers while being easier to use for a programmer [79]. Their scheduler assigns every worker an equal number of iterations in the beginning to avoid stealing when possible. When a worker runs out of iterations, it selects a victim and steals half of its iterations, similar to steal-half.

Gautier et al. describe X-Kaapi, a work-stealing runtime system for tasks with dependencies [96]. X-Kaapi schedules parallel loops by splitting them into chunks of equal size upon stealing. If there are k thieves trying to steal from the same victim,

```
SPLIT-HALF($T$)
    Let $Q_i$ be the shared deque of tasks of worker $i$
1   // Preconditions: Deque is empty and task is splittable
2   assert $Q_i$ is empty $\wedge$ $T$ is splittable
3   $count = T.end - T.next$
4   $split = T.next + count/2$
5   $T' = T$ // Make a copy of T
6   $T'.start = T'.next = split$
7   Push $T'$ onto the bottom of $Q_i$
8   $T.end = split$
```

Figure 5.9: When a worker schedules a splittable task and finds that its deque is empty, it splits the task in half, enqueues the upper half, and continues work on the lower half.

one of them is elected to create $k+1$ chunks. We will later describe a similar strategy, but with victims in charge of splitting (and sending tasks to thieves), eliminating the need for synchronization between workers contending for chunks.

5.3.1 Using Concurrent Deques

Work splitting can be understood as a scheduling extension. As such, it is not tied to any particular scheduler. We will focus on lazy splitting, starting with Tzannes et. al's description of LBS, which assumes a work-stealing scheduler based on concurrent deques. Building on that, we propose alternative splitting strategies, first for use with concurrent deques, and then for use with private deques and steal requests. Comparing these extensions will give us a better sense of the utility of explicit communication in supporting work splitting.

Lazy binary splitting LBS makes two assumptions: (1) It does not make sense to continue creating tasks if workers are already busy and unlikely to benefit from more parallelism. (2) An empty deque is an indication that more parallelism is needed, in which case tasks are potentially valuable for maintaining load balance. Thus, when a worker has scheduled a splittable task and finds that its deque is empty, it splits the current task in half, enqueues one half to allow stealing, and continues work on the other half, as detailed in Figure 5.9. We will call this strategy *split-half* because it is based on the same idea as steal-half; it takes away half of the remaining tasks, assuming that they constitute about half of the remaining work.

To combine very small tasks, a compiler may choose a profitable parallelism threshold (*ppt*) as a minimum chunk size for execution [245]. Lacking the necessary compiler support, we will assume $ppt = 1$ so that workers reevaluate the decision to split upon completing a task. This may add some runtime overhead in the form of frequent deque checks, but these are cheap operations compared to the cost of creating tasks. As long as the work is balanced and splitting is postponed, tasks are executed sequentially, with deque checks being the only source of overhead.

Figure 5.7, for which we used split-half, is proof that fine-grained parallelism can be scheduled efficiently using LBS. Unfortunately, we cannot always rely on tasks being equally fine-grained. A few long-running tasks can indeed pose a problem for split-half. Consider the following example: Worker 0 schedules a splittable task, denoted by the interval $[0, 16)$, finds that its deque is empty, splits off task $[8, 16)$, and starts to execute task 0. Worker 1 is idle, steals task $[8, 16)$ from worker 0, splits off task $[12, 16)$ (being idle implies an empty deque), and starts to execute task 8. If, at this point, worker 0 continues to be busy, only task $[12, 16)$ can be stolen. Hence, if more than four workers are idle, some will be forced to wait until worker 0 or worker 1 split again.

We can think of two solutions to avoid this load imbalance: (1) resort to software polling, which amounts to inserting deque checks into user code, or (2) choose an alternative splitting strategy that achieves a better distribution of work than split-half. Both solutions can be combined, but we may want to avoid polling, considering that concurrent deques are meant to eliminate the need for it.

Lazy guided splitting The preceding example illustrates a potential problem of splitting a task in half. Some workers may end up with too many tasks while others are starving. Suppose a worker keeps only a fraction of the tasks by splitting $[i, j)$ into $[i, i + K)$ and $[i + K, j)$ such that K, the chunk size, is equal to $\frac{1}{P}$ of the tasks, where P is the number of workers. By following this strategy, worker 0 enqueues $\frac{P-1}{P}$ of the tasks, worker 1 enqueues $\frac{P-2}{P-1}$ of the remaining tasks, and so on, until the $(P - 1)$th split has the same effect as split-half.

Figure 5.10 shows an implementation of this strategy. We introduce a field, *chunks*, initialized to P if $P > 1$ and 2 otherwise[5], to determine the value of *split* and thus the chunk size. The chunk size is simply $split - T.next$, which is equal to $\frac{T.end - T.next}{T.chunks}$. Splitting a task T with $T.chunks > 2$ creates a task T' with $T'.chunks = T.chunks - 1$. In other words, T' has all but one of T's chunks, reducing T to one chunk. Splitting a task T with $T.chunks == 2$ results in split-half; it creates a task T' with $T'.chunks = 2$,

[5]In the special case of $P = 1$, splitting can be entirely omitted, so setting $T.chunks$ to 2 (or any other value, for that matter) may have no consequence.

```
SPLIT-GUIDED($T$)
    Let $Q_i$ be the shared deque of tasks of worker $i$,
        $P$ be the number of workers
1   // Preconditions: Deque is empty and task is splittable
2   assert $Q_i$ is empty $\wedge$ $T$ is splittable
3   assert $T.chunks \geq 2 \wedge T.chunks \leq P$
4   $count = T.end - T.next$
5   if $count < T.chunks$
6       $T.chunks = count$
7   $split = T.next + count / T.chunks$
8   $T' = T$ // Make a copy of T
9   $T'.start = T'.next = split$
10  $T'.chunks = \max(T'.chunks - 1, 2)$
11  Push $T'$ onto the bottom of $Q_i$
12  $T.end = split$
13  $T.chunks = 2$
```

Figure 5.10: Lazily splitting a task into P chunks and reverting to split-half for each chunk.

which, in turn, will be split in half.

Guided splitting guarantees that a splittable task with sufficient parallelism can be executed by P workers. This guarantee does not come for free. Unless most of the workers are idle, producing P chunks may incur more overhead than necessary. To see why, suppose worker 1 steals a splittable task $[0, 1024)$ from worker 0. There are six more workers, that is, $P = 8$. Suppose none of them attempts to steal. Worker 1 begins by splitting $[0, 1024)$ into $[0, 128)$ and $[128, 1024)$, pushing the latter task onto its deque. After completing the first chunk, worker 1 pops $[128, 1024)$ from its deque, splits off $[128, 256)$, pushes $[256, 1024)$ back onto its deque, and executes the next chunk. This is repeated until worker 1 fetches the last chunk, at which point the strategy reverts to split-half. In total, worker 1 splits 14 times before it finishes executing the last task, compared to 10 times if split-half was used from the beginning. While still small in this example, the difference will increase with the number of workers, as illustrated in Figure 5.11.

Frequent splitting may affect performance when it comes to fine-grained parallelism. In fact, as the number of workers increases, it becomes more likely that guided splitting produces small chunks, potentially increasing the scheduling overhead. In the worst case, workers split off and execute one task at a time while passing the remaining chunks back and forth, causing frequent steals and limiting parallelism. This problem

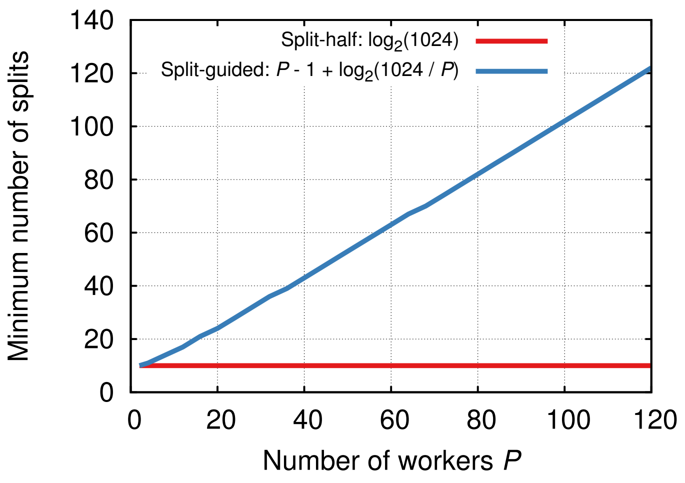


Figure 5.11: The minimum number of splits involved in scheduling a splittable task of size 1024 as a function of the number of workers P.

can be mitigated to some extent by creating tasks up front. We can specialize

```
ASYNC_FOR(f, a, b, args...);
```

to

```
for (i = a; i < b; i++)
    // Specialized version of ASYNC_FOR
    ASYNC_FOR_SINGLE(f, i, args...);
```

if `b - a <= N` at the cost of losing the ability to benefit from work splitting below a certain number of tasks. If tasks are very fine grained, however, neither guided splitting nor single tasks guarantee efficient scheduling.

Lazy adaptive splitting The solution we would like to have can be described as follows: "Use guided splitting when needed to facilitate the distribution of work; otherwise, fall back to using split-half to bound the number of splits." This strategy would schedule small chunks when workers are likely to steal and large chunks when workers are mostly busy, thus combining the benefits of split-half and split-guided. We can achieve this by counting the number of idle workers. The more workers are idle, the more tasks are needed to keep them busy; hence, a larger fraction of tasks is made available for them to steal. The appeal of this strategy is derived from its easy integration with termination detection based on Herlihy and Shavit's algorithm [116]. There is no need to introduce more synchronization if the runtime system already provides the required functionality. Figure 5.12 shows an implementation of adaptive splitting. We will focus on line 5, the main difference from the previous strategy.

```
SPLIT-ADAPTIVE($T$)
     Let $Q_i$ be the shared deque of tasks of worker $i$,
         $P$ be the number of workers,
         $P_{idle}$ be the current number of idle workers
 1   // Preconditions: Deque is empty and task is splittable
 2   assert $Q_i$ is empty $\wedge$ $T$ is splittable
 3   assert $T.chunks \geq 2 \wedge T.chunks \leq P$
 4   // A worker that splits cannot be idle
 5   assert $P_{idle} \geq 0 \wedge P_{idle} < P$
 6   $count = T.end - T.next$
 7   $\underline{T.chunks = \max(P_{idle} + 1, T.chunks)}$
 8   if $count < T.chunks$
 9       $T.chunks = count$
10   $split = T.next + count/T.chunks$
11   $T' = T$ // Make a copy of T
12   $T'.start = T'.next = split$
13   $T'.chunks = \max(T'.chunks - 1, 2)$
14   Push $T'$ onto the bottom of $Q_i$
15   $T.end = split$
16   $T.chunks = 2$
```

Figure 5.12: Lazily splitting a task into as many chunks as there are idle workers. In contrast to guided splitting, this strategy removes the implicit assumption that *all* workers are idle when scheduling a splittable task.

The value of $T.\mathit{chunks}$, which is subsequently used to determine the chunk size, is updated if the number of idle workers P_{idle} has increased such that more than $T.\mathit{chunks}$ chunks may be needed to correct the load imbalance. Because the value is not updated if $P_{\mathit{idle}} < T.\mathit{chunks}$, it serves as an upper bound for the chunk size. The reason for this upper bound is best explained with an example.

When a splittable task T is first created, $T.\mathit{chunks}$ is set to $\max(P_{\mathit{idle}} + 1, 2)$, reflecting the number of idle workers at the time of T's creation. If no worker is idle, $T.\mathit{chunks}$ is set to 2, which means that split-half may be used. It does not mean that split-half *should* be used when T is scheduled for execution because other workers may become idle in the meantime, in which case the strategy deems it necessary to enqueue a larger fraction of the tasks, proportional to the number of idle workers. Conversely, the more workers are idle at the time of T's creation, the more likely it is that T is immediately stolen and split based on a value that is unlikely to change much in the short time since T's creation.

Recall from Section 4.1 that idle workers must decrement P_{idle} when trying to steal, despite the possibility that steals may fail. As a result, it may appear that some workers are working when in fact they are not. It is more accurate to say that P_{idle} *estimates* the number of idle workers. With that said, suppose worker 0 creates and enqueues a splittable task T with $T.\mathit{chunks} == 7$, estimating that there are six idle workers, that is, $P_{\mathit{idle}} == 6$. Worker 1, being idle, steals T, goes on to split, estimates that there are five idle workers, that is, $P_{\mathit{idle}} == 5$, calculates $T.\mathit{chunks} = \max(6, 6) = 6$, and splits accordingly. This is one possible outcome. Another possible outcome is that worker 0 is led to conclude that fewer workers are idle due to concurrent steals causing P_{idle} to drop temporarily. If the chunk size only depended on P_{idle}, worker 1 would underestimate the load imbalance and claim a large chunk of T.

For this reason, the strategy errs on the side of smaller chunks by factoring in worker 0's estimate. This in turn relies on worker 0's estimate being close to the actual number of idle workers. If both worker 0 and worker 1 underestimated the load imbalance, worker 1 might initially enqueue too few tasks, depending on how far off both estimates were. In any case, using the maximum is better than deferring to worker 1. Subsequent splitting allows workers to adjust the chunk size to account for an increase in P_{idle}, giving adaptive splitting more leverage to counter load imbalance than is possible otherwise.

5.3.2 Using Private Deques and Steal Requests

With concurrent deques, workers base splitting decisions on deque checks. If a worker discovers that its deque is empty, it tries to split the current task to enqueue new work. The assumption that other workers are likely to steal is reasonable, but does not hold in every case so that workers may end up splitting more often than necessary. In Figure 5.11, we have seen that binary, guided, and, by extension, adaptive splitting cannot completely avoid creating tasks. If splitting were truly lazy, workers would not create tasks unless asked to do so. This brings us back to our channel-based scheduler.

Using private deques workers can postpone splitting until they receive steal requests. This implies that workers may never split. Let us focus the discussion on adaptive splitting. Binary and guided splitting are similar enough to Figures 5.9 and 5.10 that we will not duplicate the algorithms here. Note, however, that splitting ends with sending the new task to the thief rather than pushing it onto the victim's deque: with binary splitting, the thief receives half of the victim's tasks; with guided splitting, the thief receives a fraction $f \geq \frac{1}{2}$ of the victim's tasks, depending on the size of the chunks that have already been split off.

Figure 5.13 shows adaptive splitting in the context of private deques and steal requests, which allow workers to know exactly how many chunks to create. The assertions in the first lines document important preconditions that are shared by all lazy strategies: a worker will only split after it has received a steal request, found that its deque is empty, and confirmed that the current task is splittable. Except for the addition of steal requests, they are the same preconditions as before (cf. lines 1–2 in Figures 5.9, 5.10, and 5.12). As a consequence, a worker will not split unless its deque is empty. If the deque contains leftover work, that work will be distributed first.

Adaptive splitting is straightforward to implement: given N steal requests, a worker tries to split its current task $[a, b)$ into $N + 1$ chunks of equal size, leaving one chunk for itself. Up to $b - a - 1$ steal requests can be handled before the task ceases to be splittable, that is, if $N \geq b - a$, the last $N - (b - a - 1)$ steal requests cannot be handled and must be passed on to other victims. Note that, if $N \geq b - a - 1$, the task is broken down into single tasks, effectively reversing the bundling. The difference between breaking down a bundle into single tasks and creating single tasks in the first place is that the former eliminates all but two deque operations, while the latter requires two deque operations per task.

```
SPLIT-ADAPTIVE($T$)
    Let $Q_i$ be the private deque of tasks of worker $i$,
        $P$ be the number of workers,
        $S$ be the set of steal requests to handle
    // Preconditions: Deque is empty and task is splittable
    assert $Q_i$ is empty $\wedge$ $T$ is splittable
    // Own steal request $\notin S$
    assert $|S| > 0 \wedge |S| < P$
    $chunks = |S| + 1$
    for each steal request $s \in S$
        $count = T.end - T.next$
        if $count == 1$
            // $T$ is no longer splittable
            // Forward steal request (see Figure 4.1)
            continue
        $chunksize = \max(count/chunks, 1)$
        $split = T.end - chunksize$
        $T' = T$ // Make a copy of $T$
        $T'.start = T'.next = split$
        if $s.status == registeredIdle$
            // Notify manager (see Figure 4.4)
        Send $T'$ to channel $s.chan$
        $T.end = split$
        $chunks = chunks - 1$
```

Figure 5.13: Lazily splitting a task into as many chunks as there are pending steal requests.

5.4 Performance of Work Splitting

Lazy splitting has been proposed to dynamically control the granularity of tasks, reduce runtime overhead, and improve performance, with an eye towards nested parallelism, to which eager splitting is oblivious [245]. How that performance translates to flat parallelism, namely non-nested loops, will be addressed in this section.

5.4.1 Loop Scheduling

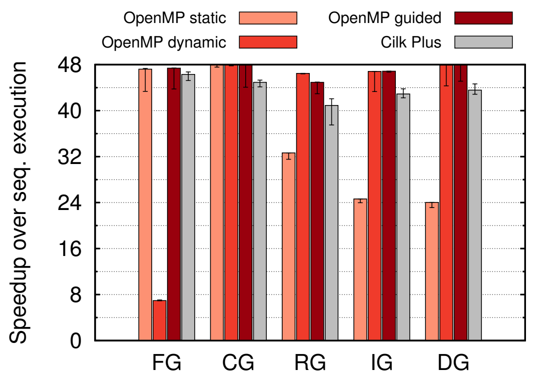
Loops with independent iterations are an important source of parallelism in many applications. Task-parallel runtime systems may choose to model parallel loops in terms of tasks. The obvious advantage of doing so is that iterations can be scheduled by work stealing, providing dynamic load balancing, which is important for many unbalanced loops. But not every loop is unbalanced, and paying the overhead of work stealing may affect performance compared to much simpler, static scheduling, especially when all iterations take the same amount of time. Fine-grained parallelism requires that iterations are combined into chunks of useful work that a scheduler can exploit. We have seen that lazy splitting is an efficient solution to this problem. Being able to create tasks “on demand” makes chunking implicit and eliminates the need for parameter tuning. This raises the question of whether lazy splitting has the potential to replace dedicated loop schedulers. As a benchmark set, we consider the following non-nested parallel loops, ranging from balanced to unbalanced, and spanning fine-grained and coarse-grained parallelism:

FG Fine-grained parallelism: 10 000 000 iterations, each lasting one microsecond.

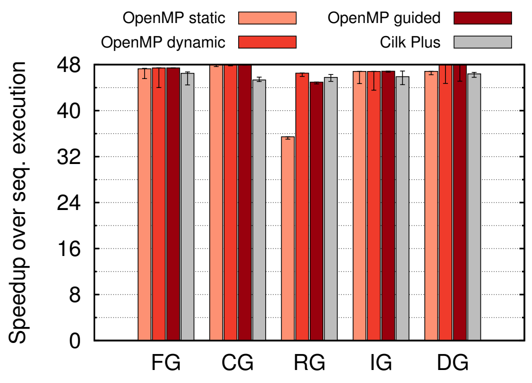
CG Coarse-grained parallelism: 960 iterations, each lasting ten milliseconds.

RG Randomly chosen granularity: 10 000 unbalanced iterations, biased towards fine-grained parallelism. There are five different granularities, from one microsecond up to ten milliseconds, separated by factors of ten. Approximately one third of the iterations are very fine-grained; 6.6% are very coarse-grained. In between, 26.6% of the iterations last ten microseconds, 20% last 100 microseconds, and 13.3% last one millisecond.

IG Linearly increasing granularity: 2000 iterations between one microsecond (first iteration) and ten milliseconds (last iteration). The increment between subsequent iterations is five microseconds.



(a) Default configuration



(b) Best results

Figure 5.14: Performance of OpenMP static, dynamic, and guided schedulers, and EBS as implemented in Cilk Plus. Figure (a) shows the speedups obtained by choosing the default chunk size for each scheduler. Figure (b) shows the speedups that resulted after manually tuning the chunk sizes. (ICC 15.0.3, `-O2`, AMD Opteron multiprocessor, 48 threads/cores)

DG Linearly decreasing granularity: 2000 iterations between ten milliseconds (first iteration) and one microsecond (last iteration). The decrement between subsequent iterations is five microseconds.

All loops contain sufficient parallelism; some may benefit from chunking.

Figure 5.14 shows the performance of OpenMP static, dynamic, and guided [199] schedulers as well as Cilk Plus, which splits eagerly by recursively dividing the iteration range in half, creating tasks along the way. Static scheduling incurs no runtime overhead besides calculating the iteration range based on a worker's ID and the total number of workers. Dynamic and guided scheduling require calls to the runtime library to fetch iterations from a shared counter using atomic operations, such as fetch-and-add or compare-and-swap, or locking as a fallback mechanism.

The first thing to note is the importance of choosing good chunk sizes for static and dynamic scheduling and, to a lesser extent, Cilk Plus. Only guided scheduling achieves near-optimal speedups across the board using its default chunking policy. Intel OpenMP schedules $\frac{1}{2P}$ of the remaining iterations, until their number is less than $2P \cdot (K + 1)$, where P is the number of workers and K is the desired minimum chunk size ($K = 1$ if unset)[6]. At this point, the remaining work is scheduled in chunks of K iterations, after switching from guided to dynamic scheduling.

[6]See `src/kmp_dispatch.cpp` in the Intel OpenMP runtime library [9].

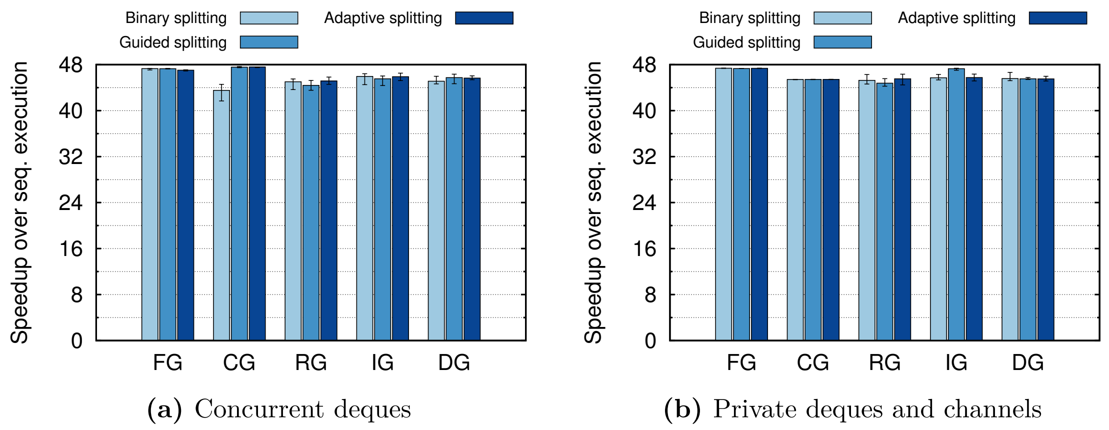


Figure 5.15: Performance of different work-splitting strategies in combination with work stealing based on concurrent deques (a) and private deques and channels (b). (ICC 15.0.3, `-O2`, AMD Opteron multiprocessor, 48 threads/cores)

Cilk Plus calculates the chunk size according to $\min(2048, \lceil \frac{N}{8P} \rceil)$, where N and P are the numbers of iterations and workers[7]. When $N \leq 8P$, the chunk size is set to one. When $N > 16384P$, the chunk size is capped at 2048 iterations. As Figure 5.14 (a) shows, this heuristic works well for most loops, but leaves some room for improvement when iterations exhibit vastly different execution times. The speedups in Figure 5.14 (b) are the result of manually trying out each chunk size $K \in \{2^i \mid 0 \leq i \leq 10\}$ and picking the best for each scheduler. For **RG**, for example, Cilk Plus performs best with a chunk size of one. The default chunk size of 27 turns out to be too large for this kind of unbalanced loop.

Figure 5.15 compares the performance of binary, guided, and adaptive splitting using concurrent and private deques. There are several things worth noting. First, lazy splitting delivers on its promise of robust performance without tuning parameters. Second, explicit communication is as efficient as sharing splittable tasks in a traditional work-stealing scheduler, with one exception: coarse-grained parallelism. Parallel loops such as **CG** demand a perfectly equal distribution of work, or else performance will suffer. This is difficult to achieve without knowing which worker has the largest chunk of remaining iterations, which must be split evenly to provide every worker with the same amount of work. Shared deques allow workers to find this chunk, steal it, and split accordingly. Guided and adaptive splitting work well for **CG**, but less so when combined with private deques. Binary splitting is a solid choice for fine-grained paral-

[7]See `runtime/cilk-abi-cilk-for.cpp` in the Intel Cilk Plus runtime library [6].

Benchmark	**Speedup**		**Performance of work splitting**			
	OpenMP	Cilk Plus	Concurrent deques		Private deques	
FG	47.43	46.49	−0.3%	+1.7%	−0.1%	+1.9%
CG	47.93	45.33	−0.7%	+5.0%	−5.2%	+0.2%
RG	46.51	45.78	−2.9%	−1.4%	−2.1%	−0.5%
IG	46.86	45.90	−1.9%	+0.1%	+0.9%	+3.0%
DG	47.93	46.40	−4.6%	−1.4%	−4.9%	−1.8%

Table 5.1: Summary of the best median results from Figures 5.14 (b) and 5.15. Lazy work-splitting schedulers, whether based on concurrent or private deques, achieve performance close to the best scheduler + chunk size combinations. (ICC 15.0.3, `-O2`, AMD Opteron multiprocessor, 48 threads/cores)

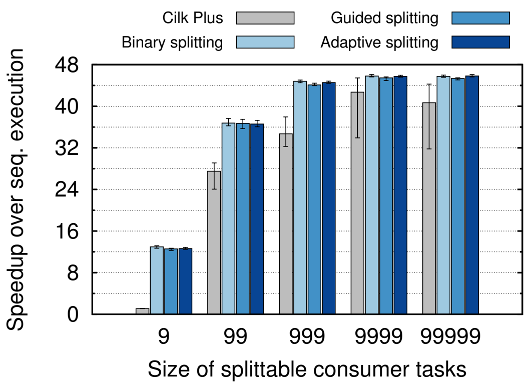
lelism, but suboptimal for coarse-grained parallelism, unless supported by polling, as Figure 5.15 (b) shows.

Steal requests make work splitting truly lazy, which we expect to be reflected in the number of splits at runtime. In fact, **FG** causes binary, guided, and adaptive splitting to perform, respectively, 11 295, 29 729, and 30 086 splits on average when using concurrent deques, compared to 1072, 2947, and 1056 splits when using private deques. Similarly but less pronounced, **RG** results in 1488, 2742, and 2730 splits on average when using concurrent deques, compared to 428, 1360, and 406 splits when using private deques.

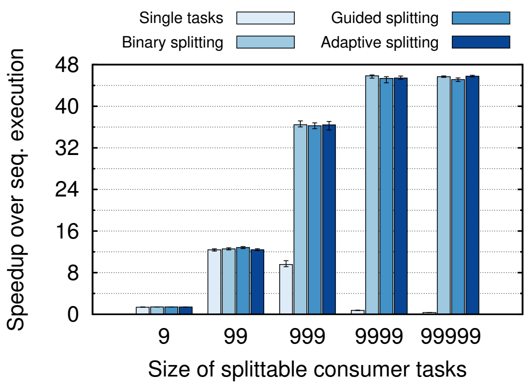
The loop scheduling results are summarized in Table 5.1. Lazy splitting comes close to the performance of loop scheduling in OpenMP and is marginally faster than eager splitting in Cilk Plus, whether using concurrent deques or private deques and steal requests (averaged over all benchmarks). Thus, lazy splitting manages to combine good performance with ease of programming through implicit chunking.

5.4.2 Mixing Tasks and Splittable Tasks

Regular tasks and splittable tasks can be freely mixed. An interesting example is a variation of BPC in which consumer tasks are bundled together so that a producer task has to create only two tasks instead of $n + 1$. (Recall that each of d producer tasks creates another producer task followed by n consumer tasks.) We can thus reduce BPC to $2d + s$ tasks, which, depending on s, the number of splits performed, may be significantly less than $(n + 1) \cdot d$. For example, if we assume that $d = 1000$, $n = 9$, and that each splittable task is split twice, that is, $s = 2000$, the total number of tasks is



(a) $t = 10\mu s$



(b) $t = 1\mu s$

Figure 5.16: Work splitting benchmarked using a variation of BPC in which consumer tasks are bundled and scheduled as splittable tasks. We tested different workloads, starting with $d = 10\,000$, $n = 9$, and $t = 10\mu s$ in Figure (a) and $d = 100\,000$, $n = 9$, and $t = 1\mu s$ in Figure (b), successively decreasing d by a factor of ten while increasing n such that the total number of tasks stays constant: 100 000 in Figure (a) and 1 000 000 in Figure (b). (ICC 15.0.3, `-O2`, AMD Opteron multiprocessor, 48 threads/cores)

$2 \cdot 1000 + 2000 = 4000$, instead of $(9 + 1) \cdot 1000 = 10\,000$ if regular tasks were used. Reducing task creation by 60% will have a noticeable effect on the program's efficiency as long as load balancing does not suffer as a result.

The addition of splittable tasks does not change the breadth-first nature of work stealing. As an example, suppose worker i has created a producer task followed by a splittable consumer task and needs to schedule new work. The splittable consumer task, pushed last, is popped first. If worker i receives a steal request from worker j, it will not split, knowing that the oldest task—the producer task—should be stolen first. In fact, sending the producer task will allow worker j to create tasks itself, and may cause fewer steals than if worker j received a fraction of worker i's consumer tasks.

We start our mixed-task BPC experiment with splittable tasks of limited parallelism ($n = 9$). Since BPC is designed to stress dynamic load balancing, work splitting may happen frequently enough to negate the benefit of bundling tasks for small values of n. As we increase the number of consumer tasks per splittable task (up to $n = 99\,999$), we expect that fewer splits are required to achieve load balance, giving workers more opportunities to serialize tasks and reduce overhead.

Figure 5.16 shows 48-core speedups over sequential execution for different values of n and task lengths of $10\mu s$ and $1\mu s$. We focus on lazy splitting in the context of private deques and steal requests. For comparison, we include Cilk Plus (using its default chunking policy) in Figure 5.16 (a) and a regular, non-work-splitting implementation of BPC in Figure 5.16 (b). We were not able to make Cilk Plus function correctly with $d = 100\,000$, which caused Cilk Plus to crash regardless of stack size.

Figure 5.16 (a) shows that lazy splitting outperforms eager splitting. For $n = 9$ and $n = 99$, Cilk Plus calculates a chunk size of one, whereas adaptive splitting, for example, manages to reduce the number of tasks from 100 000 to 61 509 ($n = 9$) and 24 520 ($n = 99$) on average. This means that 43% ($n = 9$) and 76% ($n = 99$) of all consumer tasks are executed sequentially. Starting with $n = 999$, Cilk Plus calculates chunk sizes greater than one, but remains behind lazy splitting.

Reducing the task length to 1μs makes it difficult to achieve good speedups for small values of n, whether using splittable tasks or not. The more tasks appear in sequence, however, the easier it becomes for lazy splitting to distribute the work by sending out large chunks. Contrast this with creating single tasks: the more tasks appear in sequence, the higher the likelihood of a task creation bottleneck. For $n = 999$, adaptive splitting is already 3.8× faster than our non-work-splitting implementation, highlighting the importance of bundling tasks, which, in this case, allows workers to serialize 96% of all consumer tasks.

To summarize this experiment, we note that lazy splitting is most effective when there is plenty of potential parallelism, such as task-generating loops, that can be bundled. The larger the bundles, the more potential for improvement. This does not mean that we should avoid creating small bundles. Adaptive splitting, for example, will convert a splittable task back into single tasks if there are sufficient steal requests. The advantage of creating tasks lazily as opposed to eagerly is that no deque operations are required beyond an initial push and pop.

Task-generating loops have received special support in OpenMP 4.5, the latest version of the standard at the time of writing [19]. The `taskloop` construct, which "specifies that the iterations of one or more associated loops will be executed in parallel using OpenMP tasks" (see [19], Section 2.9.2, page 87), addresses the problem of distributing work to all threads in a team in the context of nested parallelism [240]. Threads can pick up tasks created by other team members, but cannot participate in parallel loops they do not encounter[8]. Unless all threads in a team encounter a parallel loop, iterations must be packaged as tasks to be able to distribute work, hence the need for efficient task-generating loops [240]. Whether tasks are created eagerly or lazily is

[8]OpenMP distinguishes between implicit and explicit tasks [17]. A parallel region containing a parallel loop creates a team of threads, assigns an implicit task to each thread, and schedules iterations as part of these implicit tasks, which are assumed to cooperate in executing the loop. Explicit tasks are those created by the `task` construct.

left to the implementation. We hope to have shown that `taskloop`s would benefit from lazy splitting.

It is interesting to note that Tzannes et al.'s original implementation of LBS [245] opted against breadth-first work stealing, with the result of causing frequent steals for deeply nested parallelism [246]. In their follow-up work, Tzannes et al. used two deques per worker: a private deque, where most tasks are kept, and a public, shared deque to enable work stealing [246]. When a worker's public deque is empty, the worker tries to share its oldest task by moving it to the public deque or splitting it if possible. This so called lazy scheduling combines private and public deques in a way that resembles the use of channels: a public deque becoming empty is akin to a steal request waiting to be received. Sharing a task is like sending it to a channel; stealing a task is like receiving it from a channel. An important difference remains: steal requests make splitting truly lazy; shared deques, even if empty, can only suggest—not guarantee—that parallelism is needed.

5.5 Summary

The task abstraction encourages programmers to express fine-grained parallelism towards improving performance. But any runtime system comes with overhead, and too many short tasks may hurt performance. The primary sources of overhead are task creation, scheduling, and load balancing in the form of work stealing. Steal-half may help reduce the frequency of work stealing, but in some cases, being greedy and stealing many tasks is counterproductive. Robust performance requires that workers adapt their strategy if needed, based on their recent work-stealing history.

Adaptive stealing allows workers to choose and switch between steal-one and steal-half at runtime, but cannot reduce the overhead of task creation. An effective way to control the number of tasks by increasing their granularity is to bundle similar tasks, such as iterations of a parallel loop. Instead of creating, enqueuing, and dequeuing N tasks, a worker may create, enqueue, and dequeue a single task and defer splitting until workers attempt to steal. We find that lazy strategies enable efficient scheduling of fine-grained parallelism including parallel loops.

6 | Performance Comparison

The previous chapters introduced a work-stealing scheduler in which worker threads communicate by sending messages over channels. We have already seen a few performance results along the way. This chapter will focus on a performance comparison with three work-stealing schedulers, all of which are based on concurrent deques. Our goal is to demonstrate that explicit communication enables practical and efficient schedulers, even on shared-memory systems, where the channel abstraction is not strictly needed for threads to communicate.

Sections 6.1 and 6.2 give an overview of the competing runtime systems and explain our setup. Sections 6.3 through 6.9 present speedups for our set of task-parallel benchmarks and discuss the most interesting results. Section 6.10 concludes by looking at the average performance across all benchmarks.

6.1 Competing Runtime Systems

We have focused on developing a runtime system based on private deques and channels, but we also looked at other runtime systems to assess the impact of explicit communication on the performance of work stealing. The following list reviews all runtime systems that have been used throughout or in parts of this thesis:

Channel-based work stealing (Channel WS) Every worker has a private deque of tasks and two channels to communicate with other workers. The private deque is implemented as a doubly-linked list with head and tail pointers. Dequeuing a single task, whether through *pop* or *steal*, is a constant-time operation. Stealing half of a victim's tasks has linear complexity (see Section 3.5.3). Channels are concurrent FIFO queues implemented as circular arrays of fixed size. We can afford to use a relatively simple channel implementation because channels are either MPSC (steal requests) or SPSC (tasks), and the number of messages is bounded. Sending a steal request involves locking; receiving a steal request is lock free. Sending a task is lock free, as is receiving a task. A steal request takes up 32 bytes and is copied between threads. A task takes up 192 bytes and is moved between threads to avoid copying: only a pointer is sent, sufficient to transfer ownership of a task or list of tasks from victim to thief. We choose worker 1 to be in charge of termination detection in addition to being a regular worker. Futures allocate and reuse SPSC channels as described in Chapter 4. We configure the scheduler to use adaptive stealing and adaptive splitting as proposed in Chapter 5.

Chase-Lev work stealing (Chase-Lev WS) This scheduler implements our task model, but uses concurrent deques instead of private deques and channels. We use an implementation of Chase and Lev's non-blocking (lock-free) deque [66] from the High Performance ParalleX (HPX) Library developed at Indiana University [5]. This implementation issues atomic CAS operations. We allocate enough space for the deques to avoid triggering reallocations at runtime. Termination detection follows Herlihy and Shavit's algorithm [116]. Futures are implemented on top of SPSC channels analogous to the description in Chapter 4. Since we propose support for work splitting in Chapter 5, we configure the scheduler to use adaptive splitting.

Intel Cilk Plus Cilk Plus is faithful to the work-first principle (see Section 2.7.2). When a worker encounters a `spawn` statement, it defers the continuation of the parent task and invokes the child task. The other schedulers take the opposite approach: they defer the child task and continue to execute the parent task. Cilk Plus inherits Cilk's THE protocol for implementing deques [95]: *steal* requires locking; *pop* is mostly lock free, unless victim and thief contend for the same task. Work splitting (EBS) is provided by `cilk_for`, which produces chunks of 1 to 2048 iterations using a simple heuristic (see Section 5.4.1).

Intel OpenMP OpenMP has evolved considerably over the last years, and task-based parallelism is well established. Intel's OpenMP runtime library includes a work-stealing scheduler, which, while not mandated by the standard, is important for fine-grained parallelism, as we have seen in Chapter 2 using the example of UTS. Every worker maintains a deque of tasks implemented as a circular array of fixed size (256 in ICC 15.0.3). When a worker has filled its deque and fails to enqueue a task, it executes the task immediately. This can be seen as a form of granularity control, where parallelism is cut off to prevent workers from piling up an excessive number of tasks. All deque operations—*push*, *pop*, and *steal*—require locking. Nevertheless, Intel's runtime library provides one of the best-performing implementations of OpenMP tasks [186, 187, 198].

It is important to benchmark channel-based work stealing against different concurrent deques. The selected schedulers provide us with the following deques, in order of increasing sophistication: blocking (lock-based) deques, THE deques, and Chase-Lev deques. While differing in implementation details, all schedulers share the same breadth-first approach to work stealing, with victims selected at random. Workers treat their own deques as stacks and execute tasks in LIFO order to preserve locality.

6.2 Setup

Section 2.8 introduced the benchmark programs. We make sure that every runtime system is properly initialized before timing a computation so that none of the runtime systems is put at a disadvantage. As the number of threads increases, so does the cost of initializing (and finalizing) the runtime systems. In the case of Cilk Plus, we spawn and synchronize a dummy task to force the creation of worker threads. Cilk Plus provides an API function `__cilkrts_init`, which, when called, initializes the runtime system, but does so *without* creating worker threads. These are created when the internal function `__cilkrts_start_workers` is called the first time a task is spawned; hence, spawning a dummy task during initialization has the side effect of creating worker threads in addition to setting up the runtime system. In the case of OpenMP, we create a parallel region and insert a barrier, either explicitly or implicitly, to ensure that all workers are up and running when the clock is started.

To avoid thread migration, we pin workers to processor cores, except for Cilk Plus, where thread affinities are not configurable (yet)[1]. Intel OpenMP allows to set thread affinities via the environment variable `KMP_AFFINITY`. We use

```
KMP_AFFINITY=granularity=fine,scatter
```

on the Intel Xeon system,

```
KMP_AFFINITY=granularity=fine,compact
```

on the AMD Opteron system, and

```
KMP_AFFINITY=granularity=fine,balanced
```

on the Intel Xeon Phi. The latter is MIC Architecture specific and combines features of `compact`—grouping together consecutive threads— and `scatter`—distributing threads evenly across all cores. Until the number of threads exceeds the number of cores,

[1]Recent versions of the Intel Cilk Plus runtime library, up to and including Build 4420 released in November 2015 [7], contain a comment in `runtime/sysdep-unix.c: create_threads` indicating that binding threads to cores should be made an option.

`balanced` has the same effect as `scatter`, placing a thread on every core, starting with core 0. If there are more threads than cores, the first two threads are placed on core 0, the next two threads are placed on core 1, and so on, depending on the number of threads. Consequently, if the system is fully loaded, threads 0–3 share core 0, threads 4–7 share core 1, and so on. This is the same as using `compact`, but is different from using `scatter`, which keeps assigning threads to cores round robin. We should not fail to mention that the order of assignment may be less relevant when all hardware threads are used, as is the case in our experiments. That said, `scatter` and `balanced` are sensible defaults for systems that support SMT.

We measure performance on three systems that are summarized in Appendix A: a 2-socket Intel Xeon system, a 4-socket AMD Opteron system, and a Xeon Phi coprocessor. We will refer to these systems as *Small*, *Medium*, and *Large*, based on the number of hardware threads that they support and the number of worker threads that we intend to start. This means we will look at 24-thread, 48-thread, and 240-thread executions, corresponding to *Small*, *Medium*, and *Large* thread counts. The majority of the benchmarks are not able to utilize 240 threads efficiently, which means that these benchmarks also test how well the runtime systems cope with large numbers of contending workers.

6.3 SPC

The SPC benchmark contains a single task-generating loop, which allows us to take advantage of work splitting. With Intel OpenMP, we can create a task for each loop iteration, but doing so neglects a better option: loop scheduling. We therefore use guided scheduling, which avoids tasking-related overheads and has demonstrated good performance in Section 5.4. The results are shown in Figure 6.1.

On systems *Small* and *Medium*, Channel WS and Chase-Lev WS deliver comparable performance to Intel OpenMP's guided scheduling, in line with the results of Section 5.4. Cilk Plus is slightly slower. On system *Large*, however, work splitting, whether eager or lazy, loses efficiency relative to guided scheduling. Moreover, Chase-Lev WS falls short of Channel WS. Both succeed in load balancing, but overheads differ: Chase-Lev WS performs roughly an order of magnitude more steals than Channel WS. The reason is that Chase-Lev WS splits frequently, which impacts performance on fine-grained parallelism. This problem can be solved by using binary splitting, which manages to cut down the number of tasks while maintaining load balance. With Channel WS, adaptive splitting is truly lazy and not outperformed by other strategies, demonstrating the

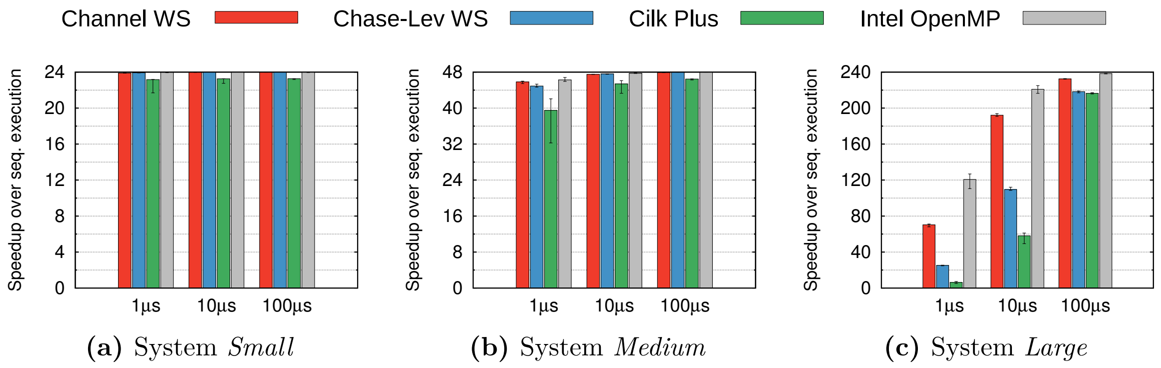


Figure 6.1: 24-thread, 48-thread, and 240-thread speedups for SPC with $n = 1\,000\,000$ (1 000 000 tasks) and $t = 1\mu$s, 10μs, and 100μs. (ICC 15.0.3, `-O2`)

robustness of work splitting using private deques and steal requests.

Increasing work-stealing overheads limit the efficiency of Channel WS on system *Large*. Using 60 workers, Channel WS is at most 8% slower than Intel OpenMP. Using 240 workers widens the gap to 42%. The number of failed attempts per steal increases by a factor of 4.1. This suggests a difficulty in finding victims to steal from; a problem that is absent from guided scheduling, which keeps all tasks in a central location.

6.4 BPC

Figure 6.2 shows speedups for BPC. We have chosen a moderately challenging workload with $d = 1000$ and $n = 999$ because we noticed that Cilk Plus performs increasingly poorly for larger values of d. With $d = 10\,000$, for example, Cilk Plus barely surpasses sequential performance, no matter which task length and how many worker threads are used. The other runtime systems, including Intel OpenMP, do not exhibit this behavior.

Not surprisingly, Intel OpenMP has trouble with very fine-grained parallelism, owing to the lack of efficient granularity control. Cilk Plus benefits from eager splitting, but schedules smaller chunks than the adaptive strategies of Channel WS and Chase-Lev WS, which achieve the lowest overhead. On system *Small*, Channel WS is on par with Chase-Lev WS. On system *Medium*, Channel WS is 23% ahead of Chase-Lev WS, but starts to fall behind as the task length is increased. The difference is more pronounced on system *Large*, where scalability is key to good performance. Channel WS does not scale as well as Chase-Lev WS and remains 24%, 11%, and 10% behind for tasks of 1μs, 10μs, and 100μs. To understand where this difference comes from, we try to quantify the efficiency of work stealing by counting the number of failed steal

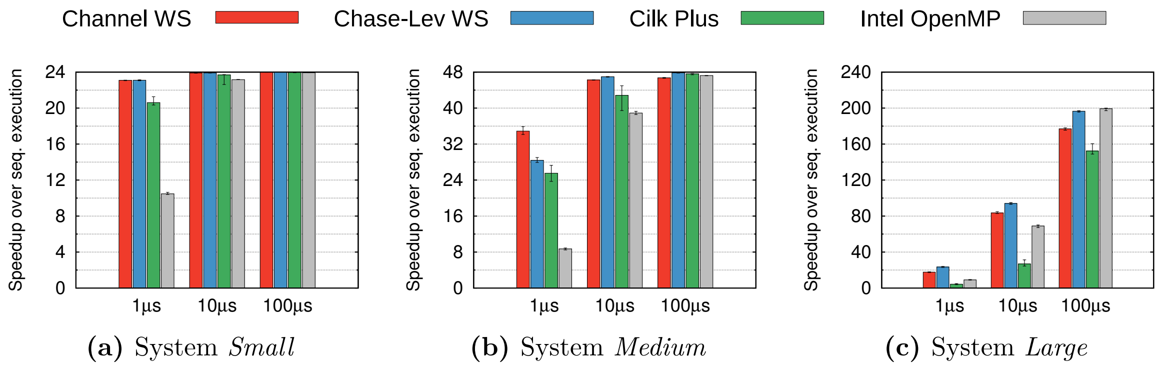


Figure 6.2: 24-thread, 48-thread, and 240-thread speedups for BPC with $d = 1000$, $n = 999$ (1 000 000 tasks), and $t = 1\mu$s, 10μs, and 100μs. (ICC 15.0.3, `-O2`)

attempts, which determine the time it takes until new tasks arrive. On systems *Small* and *Medium*, the average steal request incurs less than one failed attempt, indicating that tasks are well balanced among workers. This is not the case on system *Large*, where the collective number of failed attempts has increased by more than 10× compared to the smaller systems. Chase-Lev WS incurs failed attempts as well, but far fewer than Channel WS because workers check their victims' deques before trying to steal. In fact, we should not be too surprised that sending steal requests to random workers may cause more communication as the number of workers increases. Since BPC is meant to stress the runtime systems' ability to locate tasks, it becomes more difficult to maintain load balance at scale without efficient victim selection.

6.5 Treerec

In Section 4.4, we have used Cilk Plus as a reference for evaluating the performance of our runtime system in the context of strict fork/join parallelism. Cilk Plus excels at tree-structured computations that benefit from work-first scheduling. Figure 6.3 shows speedups for Treerec with $n = 32$.

On system *Small*, Cilk Plus achieves perfect speedups, even for very fine-grained tasks. This efficiency is only slightly reduced on system *Medium*, which matches our earlier result in Section 4.4. On system *Large*, however, Cilk Plus suffers from similar performance degradation as with BPC, leaving it behind Channel WS and Chase-Lev WS. Channel WS is never slower than Chase-Lev WS, suggesting that steal requests are handled efficiently. But this alone does not explain why Channel WS is able to outperform Chase-Lev WS when tasks are very fine grained. Both runtime systems allocate the same amount of memory for channel-based futures. Though Chase-Lev

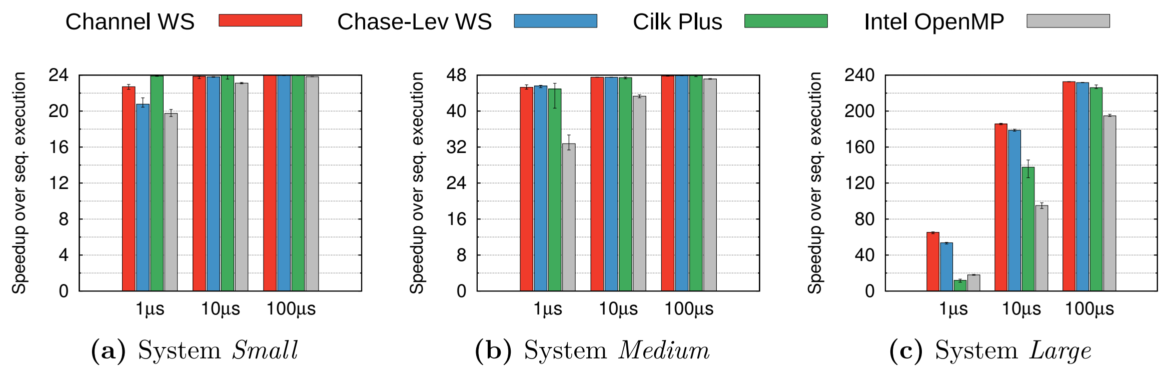


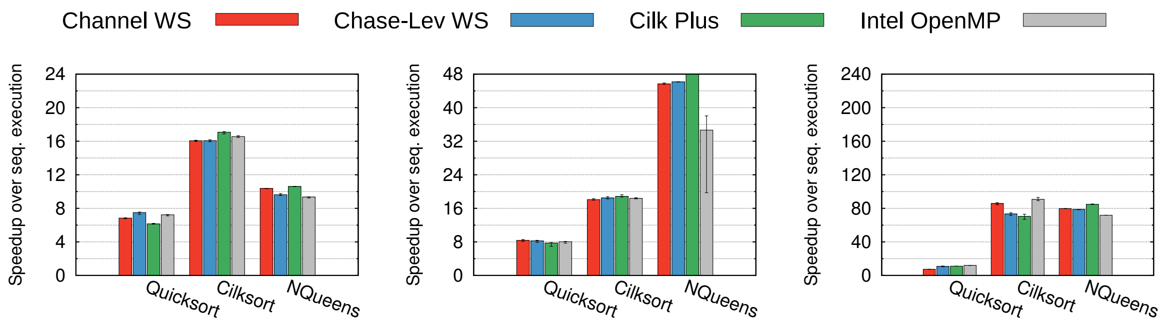
Figure 6.3: 24-thread, 48-thread, and 240-thread speedups for Treerec with $n = 32$ (3 524 577 tasks) and $t = 1\mu$s, 10μs, and 100μs. (ICC 15.0.3, `-O2`)

WS lacks support for steal-half, and, by extension, adaptive stealing, Channel WS does not attempt to steal multiple futures; workers always opt for steal-one. We do notice that workers initiate more steals. On system *Large*, for example, Channel WS exchanges 26% more tasks than Chase-Lev WS without decreasing the percentage of successful steals, which is roughly the same between Channel WS and Chase-Lev WS. More steals imply more communication and thus overhead, which we must assume is offset by improved load balancing.

Profiling reveals that workers spend less time waiting for stolen futures to be fulfilled. This is a consequence of asynchronous steal requests, which allow workers to return from `rts_force_future` even if a steal is pending, or tasks are sent concurrently (see Section 4.3 and Figure 4.4 in particular). Unlike Channel WS, Chase-Lev WS cannot interrupt a steal to return from `rts_force_future`: stealing either succeeds or fails. The best we can do is fail fast, for example, by trying to steal from a single victim before testing again if a future is fulfilled, and, in fact, doing so helps improve performance somewhat.

6.6 Sorting and N-Queens

Figure 6.4 shows three more benchmarks that exhibit strict fork/join parallelism: Quicksort, Cilksort, and N-Queens. With a few exceptions, we see comparable performance. Cilk Plus, for example, excels at N-Queens, achieving speedups of 10.6, 48, and 84.8 on systems *Small*, *Medium*, and *Large*, but channel WS comes close to these numbers; it is only 2–7% slower. What we think is interesting and not immediately obvious is that Channel WS can be fast for Cilksort but slow for Quicksort. To understand why, let us concentrate on system *Large*.



(a) System *Small* **(b)** System *Medium* **(c)** System *Large*

Figure 6.4: 24-thread, 48-thread, and 240-thread speedups for Quicksort of 100 million integers (1 697 314 tasks), Cilksort of 100 million integers (1 070 421 tasks), and N-Queens with $N = 14$ (27 358 552 tasks). Quicksort and Cilksort were run under `numactl --interleave=all` on systems *Small* (two NUMA nodes) and *Medium* (four NUMA nodes). (ICC 15.0.3, `-O2`)

Quicksort with sequential partitioning is the least scalable of our benchmarks; its speedup on P processors is bounded by $\frac{P \log_2 N}{2P-2+\log_2(N/P)}$ [243]. For $N = 10^8$ elements, we should not expect a speedup of more than 13. Thus, Quicksort does not benefit from having many workers compete for tasks. In fact, adding workers beyond a certain number may result in a slowdown, regardless of which runtime system is used. If we double the number of workers from 60 to 120, Chase-Lev WS, for example, takes 10% longer to run. If we double the number of workers again, from 120 to 240, Chase-Lev WS ends up running 44% longer than with 60 workers. Channel WS is initially on par with Chase-Lev WS, but performance degrades faster. Doubling the number of workers from 60 to 120 increases execution time by 17%. Doubling the number of workers further from 120 to 240 increases execution time by 77%. That is, going from 60 workers to 240 workers more than doubles the execution time. This amounts to a performance difference of 31% between Channel WS and Chase-Lev WS. For comparison, the difference is 7% with 120 workers and 1% with 60 workers.

As the number of workers increases, Channel WS becomes less efficient. Quicksort shows a 10-fold increase in the number of steal requests with 240 workers instead of 60 workers, suggesting that it takes effort to achieve load balance. In addition, steal requests fail twice as often, since workers have fewer tasks. The result is a 20-fold increase in the number of messages compared to using 60 workers. Cilksort, on the other hand, shows a 4-fold increase, adding less overhead as a result. We also find that Chase-Lev WS achieves a more even distribution of tasks in Quicksort, with 7072 ± 677 tasks/worker, compared to 7072 ± 2560 tasks/worker when using Channel

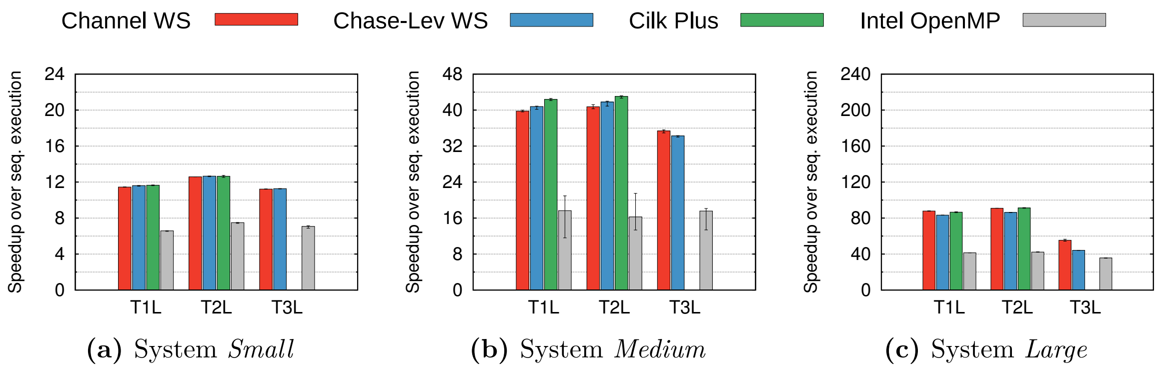


Figure 6.5: 24-thread, 48-thread, and 240-thread speedups for UTS with input trees T1L (102 181 081 tasks), T2L (96 793 509 tasks), and T3L (111 345 630 tasks). (ICC 15.0.3, `-O2`)

WS. Again, this is not the case in Cilksort, where both runtime systems achieve a similar distribution of tasks (4460 ± 167 tasks/worker for Chase-Lev WS and 4460 ± 122 tasks/worker for Channel WS).

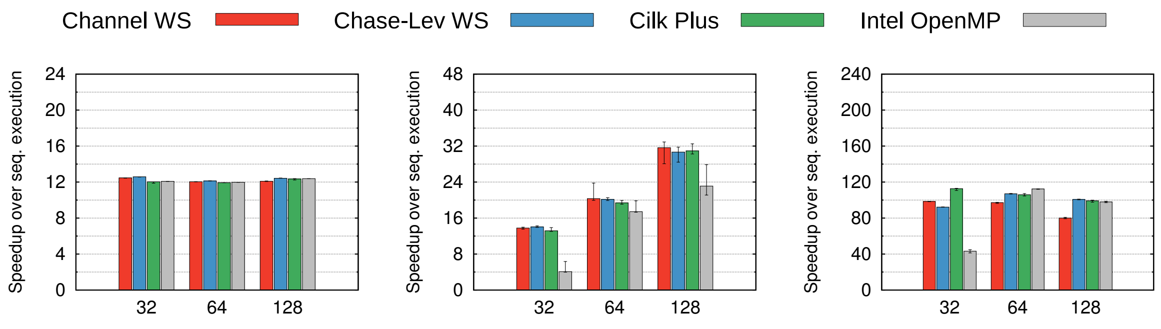
6.7 UTS

Figure 6.5 shows speedups for three UTS workloads: the geometric trees T1L and T2L and the binomial tree T3L, all having around 100 million nodes. The nodes in a geometric tree follow a geometric distribution in which the expected size of a subtree increases towards the root. Geometric trees tend to be shallow: T1L has a depth of 13 and 80% leave nodes; T2L employs a less rigid cut off, resulting in a depth of 67 and 55.57% leave nodes.

The nodes in a binomial tree have either m children with probability q or no children with probability $1 - q$, where m and q are parameters that determine the shape of a tree. The root is an exception, and has a specific branching factor b_0. In the case of T3L, the root has much more children ($b_0 = 2000$) than other internal nodes ($m = 5$, $q = 0.200014$)[2]. In contrast to geometric trees, binomial trees tend to be deep: T3L has a depth of 17 844 and 80% leave nodes.

UTS creates millions of very fine-grained tasks, thereby exposing the runtime overhead involved in scheduling and load balancing. As in previous benchmarks with fine-grained parallelism, Intel OpenMP falls short in terms of scalability, owing to the fact that even the most frequent deque operations—*push* and *pop*—require locking. Cilk Plus achieves good speedups on T1L and T2L, but fails on T3L with an internal

[2]The complete set of parameters used to generate the trees along with tree statistics can be found in the file `sample_trees.sh` included in the UTS distribution [25].



(a) System *Small* **(b)** System *Medium* **(c)** System *Large*

Figure 6.6: 24-thread, 48-thread, and 240-thread speedups for multiplying two 4096 × 4096 matrices of doubles using different block sizes: 32 × 32 (2 097 152 tasks), 64 × 64 (262 144 tasks), and 128 × 128 (32 768 tasks). (ICC 15.0.3, `-O2`)

runtime system error, which is why Figure 6.5 is missing three bars. Increasing the size of worker stacks makes no difference. Even a smaller binomial tree, such as T3, which has a depth of 1572 and roughly 4 million nodes, causes Cilk Plus to abort execution.

Channel WS and Chase-Lev WS achieve similar performance within 5% of each other, except on system *Large*, where T3L is 20% faster when using Channel WS. In Figure 5.5, we have seen that steal-half reduced the execution time of T3L by 17% because it managed to balance tasks with fewer steals than steal-one (48 workers). The same happens here on system *Large*. Channel WS performs 55% fewer steals than Chase-Lev WS, without sacrificing load balance, thanks to workers being able to switch to steal-half. In fact, workers are too conservative in their choice of strategy: steal-half is still 17% faster than adaptive stealing, which, nevertheless, provides 25% better performance than steal-one. In contrast to T3L, the geometric trees T1L and T2L do not benefit from steal-half so that workers stick to steal-one.

6.8 Matrix Multiplication

The remaining benchmarks exhibit flat parallelism. Figure 6.6 shows speedups for multiplying two 4096 × 4096 matrices of doubles using different block sizes. The block size determines the task granularity. A small block size, such as 32 (1024 elements), increases the number of tasks in each parallel phase. A large block size, such as 128 (16 384 elements), has the opposite effect. A block size in between, such as 64 (4096 elements), may be preferable in practice since it strikes a balance between the number of tasks and the amount of useful work per task. Channel WS is supported by polling. We add a call to `rts_poll` to the outermost loop of the multiplication kernel so that

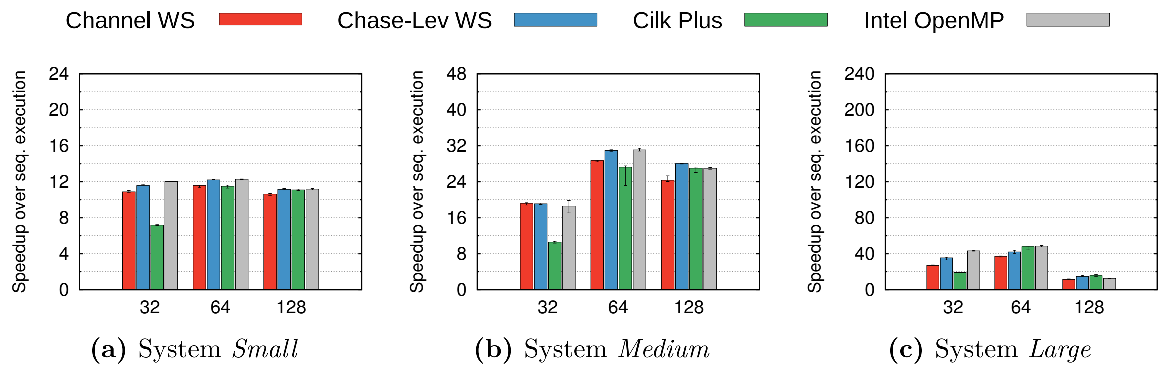


Figure 6.7: 24-thread, 48-thread, and 240-thread speedups for the LU decomposition of a sparse 4096 × 4096 matrix of doubles using different block sizes: 32 × 32 (182 976 tasks), 64 × 64 (23 904 tasks), and 128 × 128 (3248 tasks). The percentage of nonzero blocks is 11% for $B = 32$, 13% for $B = 64$, and 18% for $B = 128$. (ICC 15.0.3, `-O2`)

workers check once per iteration if there are pending steal requests. As a consequence, the number of runtime library calls per task is equal to the block size.

Channel WS is among the best runtime systems on systems *Small* and *Medium*, but falls behind on system *Large* as the block size is increased. Larger block sizes result in fewer tasks being created in each parallel phase. Thus, workers have fewer tasks to spare, and an increasing percentage of steal requests cannot be handled. Choosing a block size of 128 on system *Large* causes the percentage of successful steals in Channel WS to drop below 1%. Chase-Lev WS benefits from deque checks, which greatly reduce the number of steal attempts. The percentage of successful steals in Chase-Lev WS remains above 50%.

It is worth noting that Chase-Lev WS does not improve the distribution of tasks; it achieves the same level of load balancing as Channel WS, but does so with significantly less effort. Channel WS is up to 25% slower than Chase-Lev WS. Creating single tasks rather than splittable tasks reduces the performance difference to 8% if combined with last-victim selection. This reaffirms the importance of victim selection when parallelism is limited.

6.9 Sparse LU

Figure 6.7 shows speedups for the LU decomposition of a sparse 4096 × 4096 matrix of doubles using different block sizes. As in the previous benchmark, a single worker is responsible for creating tasks. This LU decomposition differs by using futures rather than task barriers because synchronization is frequent, and the number of tasks shrinks with each step of the decomposition so that task barriers become needlessly expensive. The OpenMP implementation is likewise based on `task`s and `taskwait`s [41, 40]. Again, the block size determines the granularity and number of tasks. Later steps of the decomposition have insufficient parallelism with fewer tasks than workers, causing significant contention. The sparsity of the matrix makes it difficult to profit from dynamic loop scheduling, whereas tasks are only created for nonzero blocks [41]. Channel WS is supported by the same polling strategy as above: every kernel contains a call to `rts_poll` so that the block size determines the number of runtime checks per task.

Channel WS is mostly slower than Chase-Lev WS, up to 31% on system *Large*. Interestingly, Chase-Lev WS is itself slower than Intel OpenMP on system *Large*. The performance difference between the two increases with the number of workers. Although futures are reused, they must be created first, causing a (potentially large) number of allocations in the first step of the decomposition. Preallocating futures or using **`spawn`** and **`sync`** (see Section 4.4) would indeed improve performance for smaller block sizes, up to 10% in our measurements.

Channel WS would likewise benefit (12%) from fewer allocations. But these are not the only overhead. More than above, random victim selection causes a huge number of steal attempts. The larger the block size, or the larger the number of workers, the smaller the percentage of successful steals. Even on system *Small*, stealing requires five attempts on average to succeed; and more than that in case of longer but fewer tasks. Most of the time, workers attempt to steal single tasks. Only the smallest block size generates enough tasks to convince workers to switch from steal-one to steal-half. The more workers are used, however, the less viable it becomes to steal multiple tasks, as evidenced by the decreasing number of tasks received per steal, from an average of five tasks on system *Small* to two tasks on system *Large*. Choosing a larger block size than 32 leaves little to no room for workers to make use of steal-half.

We tried different stealing strategies, such as steal-one combined with last-victim selection, but none was able to improve upon adaptive stealing. Lastly, we implemented victim checks analogous to the deque checks of Chase-Lev WS and others, and indeed, these help improve performance. Once again, this shows the importance of victim selection when parallelism is limited.

Choosing the largest block size increases contention over tasks to the point of slowing down the benchmark, regardless of which runtime system is used. Channel WS, Chase-Lev WS, Cilk Plus, and Intel OpenMP all take 83%, 81%, 60%, and 108% longer to run with 240 instead of 60 workers.

Overall, this is a challenging benchmark that benefits from large matrices and small block sizes. Additional parallelism can be uncovered by expressing task dependencies and taking advantage of fine-grained synchronization, which has been shown to improve scalability [77, 249].

6.10 Summary

Tables 6.1, 6.2, and 6.3 summarize the median speedups for every benchmark. These are not necessarily the best speedups that we measured. Some benchmarks are far from utilizing 240 threads efficiently, and some perform better with fewer threads and less contention. Nevertheless, we want to stress the runtime systems as much as possible, and that involves using all available hardware threads.

In addition to speedups, the tables include deviations (in percent) from the best results. The lower the deviation, that is, the closer it is to zero, the better the performance relative to the other runtime systems. For a final ranking, we determine the average performance of each runtime system on the set of 21 benchmarks/workloads (20 in the case of Cilk Plus):

System *Small*		**System *Medium***		**System *Large***	
1. Chase-Lev WS	−1.6%	1. Chase-Lev WS	−2.2%	1. Chase-Lev WS	−13.7%
2. Channel WS	−2.4%	2. Channel WS	−2.4%	2. Channel WS	−13.7%
3. Cilk Plus	−4.6%	3. Cilk Plus	−6.9%	3. Intel OpenMP	−22.2%
4. Intel OpenMP	−10.7%	4. Intel OpenMP	−21.8%	4. Cilk Plus	−28.1%

Channel WS comes out ahead of Cilk Plus and Intel OpenMP on all three systems. Only Chase-Lev WS achieves better performance. We conclude that Channel WS is competitive with state-of-the-art runtime systems based on concurrent deques.

Demonstrating good performance does not mean that there is no room for improvement. We have seen on multiple occasions that random victim selection becomes increasingly inefficient when parallelism is limited or only few workers are able to create tasks. A simple extension such as last-victim selection mitigates this problem in some cases, but does not represent a general solution. As a consequence, adding more workers than can be used increases the number of steal requests and the percentage of those that fail, which tends to degrade performance.

By using a simple channel implementation that requires locking when sending steal requests, Channel WS may incur more synchronization overhead than other runtime systems. Consider that a steal request takes t attempts to succeed, acquiring and releasing t locks in the process. Lock-free MPSC channels may be needed to match the performance of lock-free deques such as Chase and Lev's [168, 175, 261].

Benchmark	**Speedup on System *Small***							
	Channel WS		Chase-Lev WS		Cilk Plus		Intel OpenMP	
SPC 1μs	23.89	(−0.3%)	23.93	(−0.2%)	23.17	(−3.3%)	23.97	
SPC 10μs	23.89	(−0.4%)	23.99		23.26	(−3.0%)	23.99	
SPC 100μs	23.99		23.99		23.27	(−3.0%)	24.00	
BPC 1μs	23.09		23.10		20.61	(−10.8%)	10.47	(−54.7%)
BPC 10μs	23.90		23.91		23.69	(−0.9%)	23.16	(−3.1%)
BPC 100μs	23.99		23.99		23.97	(−0.1%)	23.93	(−0.3%)
Treerec 1μs	22.71	(−5.0%)	20.77	(−13.1%)	23.90		19.74	(−17.4%)
Treerec 10μs	23.85	(−0.6%)	23.80	(−0.8%)	23.99		23.11	(−3.7%)
Treerec 100μs	23.98	(−0.1%)	23.97	(−0.1%)	24.00		23.82	(−0.7%)
Quicksort 10^8	6.84	(−8.8%)	7.50		6.15	(−18.0%)	7.23	(−3.6%)
Cilksort 10^8	16.06	(−6.0%)	16.07	(−5.9%)	17.08		16.56	(−3.0%)
N-Queens 14	10.37	(−2.2%)	9.62	(−9.2%)	10.60		9.35	(−11.8%)
UTS T1L	11.45	(−1.7%)	11.61	(−0.3%)	11.65		6.58	(−43.5%)
UTS T2L	12.59	(−0.6%)	12.66		12.65	(−0.1%)	7.49	(−40.8%)
UTS T3L	11.22	(−0.4%)	11.27		N/A		7.07	(−40.8%)
MM 4096 32	12.47	(−1.0%)	12.59		12.03	(−4.4%)	12.08	(−4.1%)
MM 4096 64	12.04	(−0.7%)	12.13		11.93	(−1.6%)	11.98	(−1.2%)
MM 4096 128	12.09	(−2.7%)	12.42		12.35	(−0.6%)	12.38	(−0.3%)
LU 4096 32	10.89	(−9.6%)	11.58	(−3.8%)	7.19	(−40.3%)	12.04	
LU 4096 64	11.58	(−5.8%)	12.22	(−0.6%)	11.51	(−6.3%)	12.29	
LU 4096 128	10.64	(−4.9%)	11.16	(−0.3%)	11.10	(−0.8%)	11.19	

Table 6.1: Median speedups and relative differences (in parentheses) on system *Small*. A value of $-x$ percent means that a speedup is x percent *lower* than the best median speedup for the given benchmark. (ICC 15.0.3, `-O2`)

Benchmark	**Speedup on System *Medium***			
	Channel WS	Chase-Lev WS	Cilk Plus	Intel OpenMP
SPC 1μs	45.83 (−1.0%)	44.90 (−3.0%)	39.50 (−14.6%)	46.27
SPC 10μs	47.50 (−0.5%)	47.61 (−0.3%)	45.38 (−4.9%)	47.73
SPC 100μs	47.90 (−0.1%)	47.97	46.46 (−3.1%)	47.97
BPC 1μs	34.89	28.39 (−18.6%)	25.53 (−26.8%)	8.70 (−75.1%)
BPC 10μs	46.26 (−1.5%)	46.97	42.83 (−8.8%)	38.90 (−17.2%)
BPC 100μs	46.73 (−2.3%)	47.84	47.62 (−0.5%)	47.23 (−1.3%)
Treerec 1μs	45.28 (−0.7%)	45.59	44.92 (−1.5%)	32.76 (−28.1%)
Treerec 10μs	47.53	47.52	47.44 (−0.2%)	43.31 (−8.9%)
Treerec 100μs	47.80 (−0.3%)	47.91	47.93	47.14 (−1.6%)
Quicksort 10^8	8.36	8.24 (−1.4%)	7.73 (−7.5%)	8.04 (−3.8%)
Cilksort 10^8	18.09 (−4.1%)	18.49 (−2.0%)	18.86	18.39 (−2.5%)
N-Queens 14	45.67 (−6.5%)	46.15 (−5.5%)	48.83	34.66 (−29.0%)
UTS T1L	39.78 (−6.1%)	40.79 (−3.8%)	42.38	17.64 (−58.4%)
UTS T2L	40.74 (−5.3%)	41.80 (−2.9%)	43.04	16.26 (−62.2%)
UTS T3L	35.37	34.28 (−3.1%)	N/A	17.56 (−50.4%)
MM 4096 32	13.79 (−1.7%)	14.03	13.17 (−6.1%)	4.09 (−70.8%)
MM 4096 64	20.32	20.16 (−0.8%)	19.38 (−4.6%)	17.43 (−14.2%)
MM 4096 128	31.64	30.67 (−3.1%)	30.94 (−2.2%)	23.11 (−27.0%)
LU 4096 32	19.13	19.12 (−0.1%)	10.60 (−44.6%)	18.61 (−2.7%)
LU 4096 64	28.70 (−7.7%)	30.47 (−2.0%)	27.27 (−12.3%)	31.10
LU 4096 128	24.36 (−13.1%)	28.03	27.04 (−3.5%)	27.02 (−3.6%)

Table 6.2: Median speedups and relative differences (in parentheses) on system *Medium*. A value of $-x$ percent means that a speedup is x percent *lower* than the best median speedup for the given benchmark. (ICC 15.0.3, `-O2`)

Benchmark	**Speedup on System *Large***			
	Channel WS	Chase-Lev WS	Cilk Plus	Intel OpenMP
SPC 1μs	70.32 (−41.7%)	25.23 (−79.1%)	6.27 (−94.8%)	120.60
SPC 10μs	192.10 (−13.0%)	109.50 (−50.4%)	58.09 (−73.7%)	220.92
SPC 100μs	232.49 (−2.5%)	218.28 (−8.4%)	216.46 (−9.2%)	238.38
BPC 1μs	17.71 (−24.4%)	23.44	4.33 (−81.5%)	9.09 (−61.2%)
BPC 10μs	83.52 (−11.2%)	94.04	26.83 (−71.5%)	68.99 (−26.6%)
BPC 100μs	176.95 (−11.2%)	196.54 (−1.4%)	152.33 (−23.6%)	199.28
Treerec 1μs	65.24	53.67 (−17.7%)	11.89 (−81.8%)	17.98 (−72.4%)
Treerec 10μs	185.70	178.75 (−3.7%)	137.68 (−25.9%)	95.02 (−48.8%)
Treerec 100μs	232.68	231.69 (−0.4%)	226.05 (−2.8%)	194.77 (−16.3%)
Quicksort 10^8	7.41 (−38.7%)	10.76 (−10.9%)	11.08 (−8.3%)	12.08
Cilksort 10^8	85.76 (−5.5%)	73.37 (−19.2%)	70.42 (−22.4%)	90.78
N-Queens 14	79.67 (−6.1%)	78.63 (−7.3%)	84.85	71.80 (−15.4%)
UTS T1L	87.82	83.24 (−5.2%)	86.58 (−1.4%)	41.42 (−52.8%)
UTS T2L	90.91 (−0.4%)	86.26 (−5.5%)	91.25	42.23 (−53.7%)
UTS T3L	55.34	44.20 (−20.1%)	N/A	35.68 (−35.5%)
MM 4096 32	98.45 (−12.6%)	92.19 (−18.2%)	112.65	43.27 (−61.6%)
MM 4096 64	97.19 (−13.5%)	106.98 (−4.8%)	105.92 (−5.7%)	112.37
MM 4096 128	80.01 (−20.5%)	100.66	99.15 (−1.5%)	97.98 (−2.7%)
LU 4096 32	27.05 (−37.5%)	35.53 (−17.9%)	19.24 (−55.5%)	43.27
LU 4096 64	37.16 (−23.5%)	42.09 (−13.3%)	47.80 (−1.6%)	48.57
LU 4096 128	11.55 (−25.9%)	14.87 (−4.6%)	15.58	12.74 (−18.2%)

Table 6.3: Median speedups and relative differences (in parentheses) on system *Large*. A value of $-x$ percent means that a speedup is x percent *lower* than the best median speedup for the given benchmark. (ICC 15.0.3, `-O2`)

7 | Conclusion and Future Work

This final chapter recapitulates our results and outlines some possible directions for future work.

7.1 Conclusion

We set out to create a work-stealing runtime system using private deques and channels to eliminate the need for concurrent deques, which, in our experience, limit the flexibility of work stealing. Our goal was to explore a general message-passing implementation that can compete with, or even outperform, concurrent deques.

Private deques are more flexible than concurrent deques by virtue of being private: tasks may be stored in lists, trees, or other sequential containers, avoiding the complications of concurrency. Explicit communication improves portability by removing the dependency on shared memory. Workers exchange steal requests and tasks over buffered channels. Asynchronous steal requests provide opportunities to overlap communication with computation, for example, by initiating steals shortly before running out of work. Since channels can be used to transfer ownership of data between workers, it is possible to steal multiple tasks without increasing the number of messages. Long-running tasks may have to poll to ensure that steal requests are handled in a timely manner. Fine-grained parallelism mostly obviates the need for polling.

We derived an algorithm for termination detection that combines features of shared-memory barriers and distributed-memory protocols. Termination is detected once all workers are idle, as evidenced by their steal requests, without propagating additional control messages. We extended the termination detection barrier to a task barrier with little additional communication.

Tasks often depend on the results or side effects of other tasks, motivating the use of futures, which can be viewed as channels connecting parent and child tasks. A worker is free to pick up other tasks while waiting for a future's result. With some tuning, such as reducing the number of heap allocations, channel-based futures can compete with Cilk Plus on fork/join parallelism, while being more flexible in general.

We paid particular attention to fine-grained parallelism. Adaptive stealing combines the benefits of steal-one and steal-half by choosing the appropriate strategy at runtime. Though private deques reduce the cost of task creation, large numbers of fine-grained tasks still necessitate granularity control. Splittable tasks address the problem of overexposing parallelism by bundling and splitting off tasks as needed. Perhaps most importantly, splittable tasks simplify loop scheduling, allowing the runtime system to schedule chunks of iterations by work stealing. Lazy splitting has been shown to achieve better performance portability than eager splitting. We developed extensions to make lazy splitting more robust and demonstrated comparable performance to dedicated loop schedulers.

Lastly, we showed that channel-based work stealing can compete with concurrent deques on current multi- and manycore systems. In fact, channel-based work stealing performed better on average than Intel Cilk Plus and Intel OpenMP.

7.2 Ideas for Future Work

Section 3.4 mentioned some alternatives to random victim selection. In sampling victim selection [88] or the closely related group-based victim selection [45], a worker samples n potential victims to determine and pick the one with the most tasks. Thanks to asynchronous steal requests, sampling victim selection can be implemented by sending and then forwarding a steal request $n - 1$ times and recording which worker has the most tasks. After the sampling is done, the steal request is forwarded once again, this time to the designated victim, which handles it like a normal steal request. Because random victim selection can be thought of as sampling a single victim, it should be possible to devise an adaptive strategy that varies n depending on the number of failed attempts that a steal request has caused, with the goal of improving the effectiveness of steal-half and, by extension, adaptive stealing. We are not aware of previous work that has investigated the combination of sampling victims and steal-half.

When parallelism is limited, workers can reduce contention by backing off from stealing. Saraswat et al. have proposed an interesting strategy in which workers back off by sending steal requests to other workers, establishing "lifelines" that determine how new work will be distributed [218]. The basic idea of remembering steal requests is easy to implement in our runtime system. The more interesting question is whether the benefits of work stealing and work sharing can be combined without having to precompute suitable lifelines for each worker.

Channel-based work stealing can be used on any system that is capable of supporting channels through shared memory, message passing, or a combination of both. This includes future multi- and manycore processors as well as manycore clusters. To target the latter, we may run an instance of channel-based work stealing on each node and have managers relay messages between nodes. This makes it easy to specialize channels for intra-node and inter-node communication to reduce latency where possible.

Since managers are responsible for termination detection, they can pass those steal requests that could not be handled within their nodes on to other managers, initiating global load balancing when local load balancing has failed. If we assume that only managers are able to communicate with other nodes, managers act as proxies for inter-node steals. Neither thief nor victim need to take special action; steal requests are flexible enough to be "hijacked", meaning a worker can change the channel reference contained in a steal request and intercept tasks. By doing so, managers are able to forward tasks from their nodes to other nodes and from other nodes to workers within their nodes.

Hierarchical work stealing can help exploit locality in the presence of increasingly complex memory hierarchies, including those of manycore clusters [170, 264]. Workers running on cores in close proximity can be grouped together into places [102], with managers being in charge of local termination detection and inter-place communication. If workers can communicate directly with other places, for instance, within a single node, managers need not participate in inter-place steals.

Work stealing may suffer from long message latencies. If parallelism is not the limiting factor, workers can try to prefetch tasks by sending steal requests further ahead of time. While prefetching did not improve performance in our tests, its potential for hiding latency is worth exploring on more systems. It has been shown in the past that prefetching benefits load balancing in high-latency networks [248].

Another way to prefetch tasks is to continue stealing even after succeeding. Workers can forward steal requests until the desired number of tasks has been prefetched. Since there is still only one steal request per worker, it is easy to ensure that tasks are never sent concurrently so that workers can keep using SPSC channels. This would not be possible if workers were allowed to send multiple steal requests. An alternative would be to allocate a second SPSC channel to support two concurrent steal requests per worker. For example, a worker could initiate a local and a remote steal request, the latter for the purpose of prefetching, similar to the wide-area work-stealing strategy of van Nieuwpoort et al [248].

We mentioned that steal requests may be hijacked in order to intercept tasks. Suppose worker i is idle. While waiting for tasks, it forwards steal requests from other workers. Some steal requests, especially those meant for prefetching, can be considered less urgent than others. Worker i could hijack such a steal request, hoping to reduce its own time spent waiting. This would allow starving workers to get back to work faster, without changing the upper bound for the number of steal requests, at the cost of requiring MPSC channels for sending tasks between workers.

This much is certain: channel-based work stealing opens up many interesting avenues for future work, which we look forward to exploring.

A | CPU Architectures

The following tables summarize the CPU architectures on which we have run our tests. Most of the information is taken from the `lscpu` command and from `/proc/cpuinfo`. Minimum and maximum clock speeds are determined by reading

```
/sys/devices/system/cpu/cpu*/cpufreq/cpuinfo_{min,max}_freq.
```

The Intel Core i7 is included for completeness; it is used only in Figure 2.4. The processor topologies in Figures A.1 and A.2 are gathered from `lstopo` with

```
lstopo --no-legend --no-io.
```

For lack of space, we omit similar topology information for the 240-thread Intel Xeon Phi and point the interested reader to the Portable Hardware Locality (*hwloc*) project's web page at `https://www.open-mpi.org/projects/hwloc`, which contains a number of examples, including the graphical output of running `lstopo` on a Xeon Phi coprocessor.

A.1 Intel® Core™ i7-4770 Processor

CPUs	8
Threads per core	2
Cores per socket	4
Sockets	1
NUMA nodes	1
CPU MHz min	800.000
CPU MHz max	3900.000
L1d cache	32 KB
L1i cache	32 KB
L2 cache	256 KB
L3 cache	8192 KB
Operating system	openSUSE 13.1 (x86_64)
Kernel release	3.12.53-40-desktop

A.2 2× Intel® Xeon® Processor E5-2630

CPUs	24
Threads per core	2
Cores per socket	6
Sockets	2
NUMA nodes	2
CPU MHz	2900.000
L1d cache	32 KB
L1i cache	32 KB
L2 cache	256 KB
L3 cache	15360 KB
Operating system	SUSE Linux Enterprise Server 11 (x86_64)
Kernel release	3.0.101-0.40-default

A.3 4× AMD Opteron™ Processor 6172

CPUs	48
Threads per core	1
Cores per socket	12
Sockets	4
NUMA nodes	8
CPU MHz min	800.000
CPU MHz max	2100.000
L1d cache	64 KB
L1i cache	64 KB
L2 cache	512 KB
L3 cache	5118 KB
Operating system	openSUSE 13.1 (x86_64)
Kernel release	3.11.10-21-desktop

A.4 Intel® Xeon Phi™ Coprocessor 5110P

CPUs	240
Threads per core	4
Cores	60
CPU MHz min	842.104
CPU MHz max	1052.630
L1d cache	32 KB
L1i cache	32 KB
L2 cache	512 KB
Operating system	GNU/Linux-based microkernel
Kernel release	2.6.38.8+mpss3.4.1

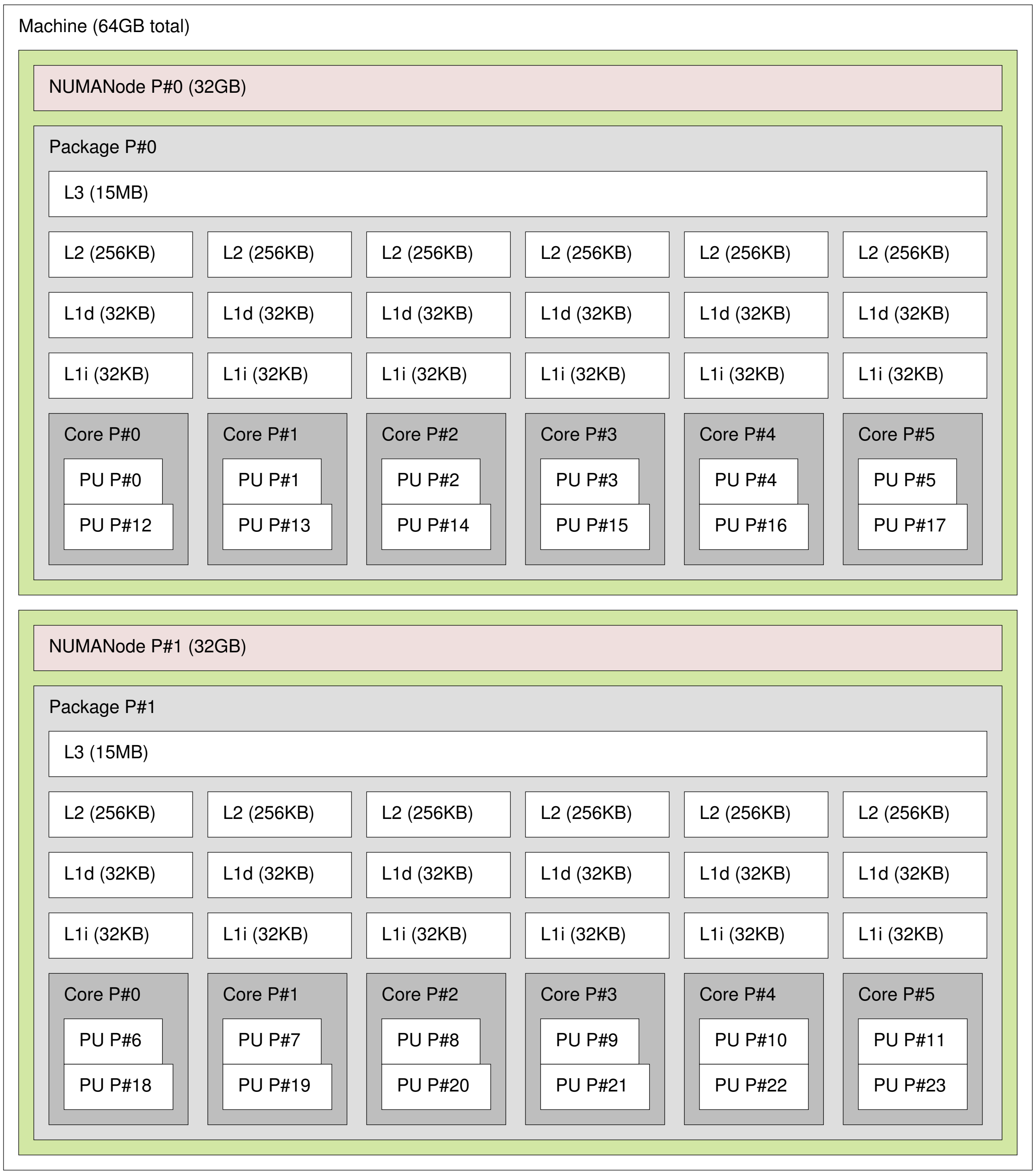


Figure A.1: Processor topology of the 24-thread Intel Xeon system.

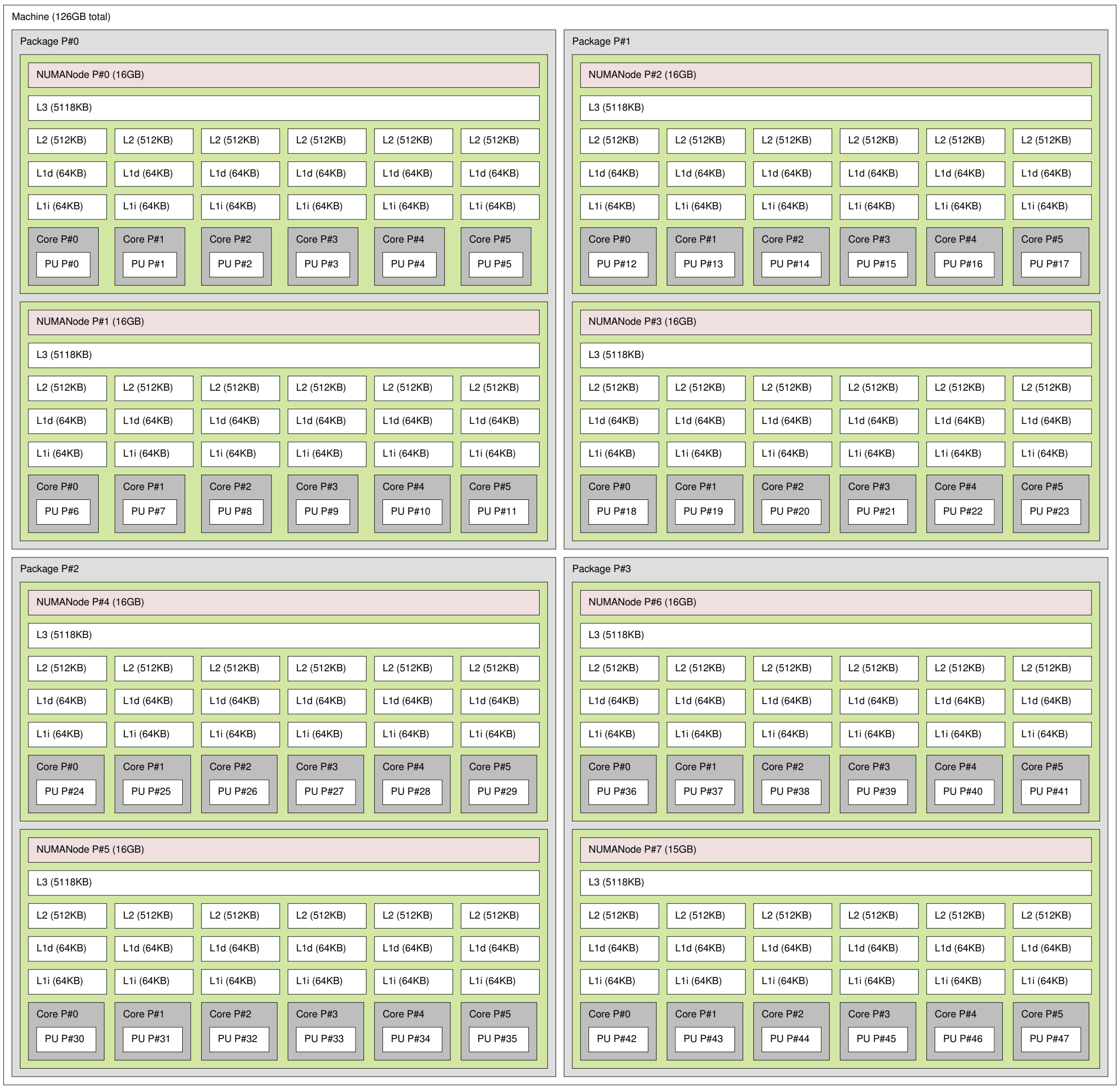


Figure A.2: Processor topology of the 48-thread AMD Opteron system.

Bibliography

[1] A Brief Overview of Chapel. `http://chapel.cray.com/papers/BriefOverviewChapel.pdf` (last visited July 5, 2016).

[2] GCC Online Documentation. `https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html` (last visited July 5, 2016).

[3] GNU libgomp. `https://gcc.gnu.org/onlinedocs/libgomp` (last visited July 5, 2016).

[4] Grand Central Dispatch (GCD) Reference. `https://developer.apple.com/library/mac/documentation/Performance/Reference/GCD_libdispatch_Ref/index.html` (last visited July 5, 2016).

[5] High Performance ParalleX. `https://hpx.crest.iu.edu` (last visited July 5, 2016).

[6] Intel® Cilk™ Plus. `http://www.cilkplus.org/` (last visited July 5, 2016).

[7] Intel® Cilk™ Plus Build 4420. `https://www.cilkplus.org/sites/default/files/runtime_source/cilkplus-rtl-004420.tgz` (last visited July 5, 2016).

[8] Intel® Math Kernel Library (Intel® MKL). `https://software.intel.com/en-us/intel-mkl` (last visited July 5, 2016).

[9] Intel® OpenMP Runtime Library. `https://www.openmprtl.org` (last visited July 5, 2016).

[10] Intel® Threading Building Blocks Documentation. `https://software.intel.com/en-us/node/506100` (last visited July 5, 2016).

[11] Intel® VTune™ Amplifier XE. `https://software.intel.com/en-us/intel-vtune-amplifier-xe` (last visited July 5, 2016).

[12] Intel® Xeon Phi™ Coprocessor x100 Product Family. `http://ark.intel.com/products/family/92649/Intel-Xeon-Phi-Coprocessor-x100-Product-Family` (last visited July 5, 2016).

[13] Intel® Xeon® Processor E7 v4 Family. `http://ark.intel.com/products/family/93797/Intel-Xeon-Processor-E7-v4-Family` (last visited July 5, 2016).

[14] Many-core Applications Research Community. `https://communities.intel.com/community/marc` (last visited July 5, 2016).

[15] MPI: A Message-Passing Interface Standard Version 2.2. `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf` (last visited July 5, 2016).

[16] MPI: A Message-Passing Interface Standard Version 3.0. `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf` (last visited July 5, 2016).

[17] OpenMP Application Program Interface Version 3.0. `http://www.openmp.org/mp-documents/spec30.pdf` (last visited July 5, 2016).

[18] OpenMP Application Program Interface Version 3.1. `http://www.openmp.org/mp-documents/OpenMP3.1.pdf` (last visited July 5, 2016).

[19] OpenMP Application Programming Interface Version 4.5. `http://www.openmp.org/mp-documents/openmp-4.5.pdf` (last visited July 5, 2016).

[20] The Barcelona OpenMP Task Suite (BOTS) Project. `https://pm.bsc.es/projects/bots` (last visited July 5, 2016).

[21] The Cilk Project. `http://supertech.csail.mit.edu/cilk/` (last visited July 5, 2016).

[22] The GNU C Library (glibc). `https://www.gnu.org/software/libc/manual/html_node/Merged-Signals.html` (last visited July 5, 2016).

[23] The Go Programming Language. `http://golang.org` (last visited July 5, 2016).

[24] The Rust Programming Language. `http://rust-lang.org` (last visited July 5, 2016).

[25] The Unbalanced Tree Search Benchmark. `http://sourceforge.net/p/uts-benchmark/wiki/Home` (last visited July 5, 2016).

[26] User and Reference Guide for the Intel® C++ Compiler 15.0. `https://software.intel.com/en-us/node/522593` (last visited July 5, 2016).

[27] Partitioned Global Address Space (PGAS) Languages. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1539–1545. Springer US, 2011.

[28] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling Parallel Programs by Work Stealing with Private Deques. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, pages 219–228, New York, NY, USA, 2013. ACM.

[29] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Theory and Practice of Chunked Sequences. In Andreas S. Schulz and Dorothea Wagner, editors, *Algorithms - ESA 2014*, volume 8737 of *Lecture Notes in Computer Science*, pages 25–36. Springer Berlin Heidelberg, 2014.

[30] Sarita Adve. Data Races Are Evil with No Exceptions: Technical Perspective. *Commun. ACM*, 53(11):84–84, November 2010.

[31] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *Computer*, 29(12):66–76, December 1996.

[32] Shivali Agarwal, Rajkishore Barik, Dan Bonachea, Vivek Sarkar, Rudrapatna K. Shyamasundar, and Katherine Yelick. Deadlock-free Scheduling of X10 Computations with Bounded Resources. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 229–240, New York, NY, USA, 2007. ACM.

[33] Kunal Agrawal, I-Ting Angelina Lee, and Jim Sukha. Brief Announcement: Serial-Parallel Reciprocity in Dynamic Multithreaded Languages. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 186–188, New York, NY, USA, 2010. ACM.

[34] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Executing Task Graphs Using Work-Stealing. In *2010 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–12, April 2010.

[35] Selim G. Akl and Nicola Santoro. Optimal Parallel Merging and Sorting Without Memory Conflicts. *IEEE Trans. Comput.*, 36(11):1367–1369, November 1987.

[36] Diego Andrade, Basilio B. Fraguela, James Brodman, and David Padua. Task-Parallel Versus Data-Parallel Library-Based Programming in Multicore Systems. In *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, PDP '09, pages 101–110, Washington, DC, USA, 2009. IEEE Computer Society.

[37] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM.

[38] 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, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report.

[39] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot be Eliminated. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 487–498, New York, NY, USA, 2011. ACM.

[40] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The Design of OpenMP Tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20:404–418, March 2009.

[41] Eduard Ayguadé, Alejandro Duran, Jay Hoeflinger, Federico Massaioli, and Xavier Teruel. An Experimental Evaluation of the New OpenMP Tasking Model. In Vikram Adve, María Jesús Garzarán, and Paul Petersen, editors, *Languages and Compilers for Parallel Computing*, pages 63–77. Springer-Verlag, Berlin, Heidelberg, 2008.

[42] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[43] Roberto Belli and Torsten Hoefler. Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization. In *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 871–881, May 2015.

[44] Petra Berenbrink, Tom Friedetzky, and Leslie Ann Goldberg. The Natural Work-Stealing Algorithm is Stable. *SIAM J. Comput.*, 32(5):1260–1279, May 2003.

[45] Abhishek Bhattacharjee, Gilberto Contreras, and Margaret Martonosi. Parallelization Libraries: Characterizing and Reducing Overheads. *ACM Trans. Archit. Code Optim.*, 8(1):5:1–5:29, February 2011.

[46] Guy E. Blelloch, Phillip B. Gibbons, Yossi Matias, and Girija J. Narlikar. Space-Efficient Scheduling of Parallelism with Synchronization Variables. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 12–23, New York, NY, USA, 1997. ACM.

[47] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[48] Robert D. Blumofe and Charles E. Leiserson. Space-efficient Scheduling of Multithreaded Computations. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 362–371, New York, NY, USA, 1993. ACM.

[49] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, 46:720–748, September 1999.

[50] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and Reliable Parallel Computing on Networks of Workstations. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '97, pages 10–10, Berkeley, CA, USA, 1997. USENIX Association.

[51] Hans-J. Boehm. How to Miscompile Programs with "Benign" Data Races. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

[52] Brent Bohnenstiehl, Aaron Stillmaker, John Pimentel, Timothy Andreas, Bin Liu, Anh Tran, Emmanuel Adeagbo, and Bevan Baas. KiloCore: A 32nm 1000-Processor Array. *Hot Chips 2016 Symposium on High Performance Chips*, 2016. to appear.

[53] Dan Bonachea. GASNet Specification, V1.1. Technical report, Berkeley, CA, USA, 2002.

[54] Shekhar Borkar. Thousand Core Chips: A Technology Perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.

[55] Shekhar Borkar and Andrew A. Chien. The Future of Microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[56] Clay Breshears. *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications.* O'Reilly Media, Inc., 2009.

[57] Eric A. Brewer, Frederic T. Chong, Lok T. Liu, Shamik D. Sharma, and John D. Kubiatowicz. Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 42–53, New York, NY, USA, 1995. ACM.

[58] Marco Bungart, Claudia Fohry, and Jonas Posner. Fault-Tolerant Global Load Balancing in X10. In *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 471–478, Sept 2014.

[59] F. Warren Burton and M. Ronan Sleep. Executing Functional Programs on a Virtual Tree of Processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, FPCA '81, pages 187–194, New York, NY, USA, 1981. ACM.

[60] Paul Butcher. *Seven Concurrency Models in Seven Weeks: When Threads Unravel*. Pragmatic Bookshelf, 1st edition, 2014.

[61] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra Marathe, and Mark Moir. Message Passing or Shared Memory: Evaluating the Delegation Abstraction for Multicores. In Roberto Baldoni, Nicolas Nisse, and Maarten van Steen, editors, *Principles of Distributed Systems*, volume 8304 of *Lecture Notes in Computer Science*, pages 83–97. Springer International Publishing, 2013.

[62] Colin Campbell, Ralph Johnson, Ade Miller, and Stephen Toub. *Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures*. Microsoft Press, 1st edition, 2010.

[63] Vincent Cavé, Zoran Budimlić, and Vivek Sarkar. Comparing the Usability of Library vs. Language Approaches to Task Parallelism. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 9:1–9:6, New York, NY, USA, 2010. ACM.

[64] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: the New Adventures of Old X10. PPPJ '11, pages 51–61, 2011.

[65] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[66] David Chase and Yossi Lev. Dynamic Circular Work-Stealing Deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM.

[67] Guojing Cong, Sreedhar Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay Saraswat, and Tong Wen. Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 536–545, Washington, DC, USA, 2008. IEEE Computer Society.

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

[69] Cray Inc. Chapel Language Specification, Version 0.95. `http://chapel.cray.com` (last visited July 5, 2016), April 2014.

[70] Benôit D. de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benôit Ganne, Pierre G. de Massas, François Jacquet, Samuel Jones, Nicolas M. Chaisemartin, Frédéric Riss, and Thierry Strudel. A Clustered Manycore Processor Architecture for Embedded and Accelerated Applications. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6, Sept 2013.

[71] Eric D. Demaine. Protocols for Non-Deterministic Communication over Synchronous Channels. In *Proceedings of the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, IPPS/SPDP '98, pages 24–30, Washington, DC, USA, 1998. IEEE Computer Society.

[72] Edsger W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. Derivation of a Termination Detection Algorithm for Distributed Computations. *Information Processing Letters*, 16(5):217–219, 1983.

[73] James Dinan, Sriram Krishnamoorthy, D. Brian Larkins, Jarek Nieplocha, and P. Sadayappan. Scioto: A Framework for Global-View Task Parallelism. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 586–593, Washington, DC, USA, 2008. IEEE Computer Society.

[74] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable Work Stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 53:1–53:11, New York, NY, USA, 2009. ACM.

[75] James Dinan, Stephen Olivier, Gerald Sabin, Jan Prins, P. Sadayappan, and Chau-Wen Tseng. Dynamic Load Balancing of Unbalanced Computations Using Message Passing. In *2007 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IPDPS '07, pages 1–8, 2007.

[76] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. An Adaptive Cut-off for Task Parallelism. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 36:1–36:11, Piscataway, NJ, USA, 2008. IEEE Press.

[77] Alejandro Duran, Roger Ferrer, Eduard Ayguadé, Rosa M. Badia, and Jesus Labarta. A Proposal to Extend the OpenMP Tasking Model with Dependent Tasks. *International Journal of Parallel Programming*, 37(3):292–305, 2009.

[78] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguadé. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[79] Marie Durand, François Broquedis, Thierry Gautier, and Bruno Raffin. An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines. In Alistair P. Rendell, Barbara M. Chapman, and Matthias S. Müller, editors, *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 141–155. Springer Berlin Heidelberg, 2013.

[80] Richard Durstenfeld. Algorithm 235: Random Permutation. *Commun. ACM*, 7(7):420, July 1964.

[81] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing. *Performance Evaluation*, 6(1):53 – 68, 1986.

[82] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup Versus Efficiency in Parallel Systems. *IEEE Trans. Comput.*, 38(3):408–423, March 1989.

[83] Ralf S. Engelschall. Portable Multithreading: The Signal Stack Trick for User-Space Thread Creation. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, pages 20–20, Berkeley, CA, USA, 2000. USENIX Association.

[84] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[85] Panagiota Fatourou and Paul G. Spirakis. A New Scheduling Algorithm for General Strict Multithreaded Computations. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 297–311, London, UK, UK, 1999. Springer-Verlag.

[86] Karl-Filip Faxén. Wool - A Work Stealing Library. *SIGARCH Comput. Archit. News*, 36(5):93–100, June 2009.

[87] Karl-Filip Faxén. Efficient Work Stealing for Fine Grained Parallelism. In *2010 39th International Conference on Parallel Processing (ICPP)*, pages 313 –322, September 2010.

[88] Karl-Filip Faxén and John Ardelius. Manycore Work Stealing. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 10:1–10:2, New York, NY, USA, 2011. ACM.

[89] Marc Feeley. A Message Passing Implementation of Lazy Task Creation. In *Proceedings of the US/Japan Workshop on Parallel Symbolic Computing: Languages, Systems, and Applications*, pages 94–107, London, UK, UK, 1993. Springer-Verlag.

[90] Marc Feeley. Polling Efficiently on Stock Hardware. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 179–187, New York, NY, USA, 1993. ACM.

[91] Cormac Flanagan. Futures. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 749–753. Springer US, 2011.

[92] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Manticore: A Heterogeneous Parallel Language. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 37–44, New York, NY, USA, 2007. ACM.

[93] Message Passing Forum. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.

[94] Andrew Friedley, Torsten Hoefler, Greg Bronevetsky, Andrew Lumsdaine, and Ching-Chen Ma. Ownership Passing: Efficient Distributed Memory Programming on Multi-core Systems. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 177–186, New York, NY, USA, 2013. ACM.

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

[96] Thierry Gautier, Fabien Lementec, Vincent Faucher, and Bruno Raffin. X-Kaapi: A Multi Paradigm Runtime for Multicore Architectures. In *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ICPP '13, pages 728–735, Washington, DC, USA, 2013. IEEE Computer Society.

[97] Andrew Gerrand. Share Memory By Communicating. `http://blog.golang.org/share-memory-by-communicating` (last visited July 5, 2016).

[98] Diana Göhringer, Michael Hübner, and Jürgen Becker, editors. *3rd Many-core Applications Research Community (MARC) Symposium. Proceedings of the 3rd MARC Symposium, Ettlingen, Germany, July 5-6, 2011.* KIT Scientific Publishing, Karlsruhe, 2011.

[99] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy Threads. *J. Parallel Distrib. Comput.*, 37(1):5–20, August 1996.

[100] David Grove, Olivier Tardieu, David Cunningham, Ben Herta, Igor Peshansky, and Vijay Saraswat. A Performance Model for X10 Applications: What's Going on Under the Hood? In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, pages 1:1–1:8, New York, NY, USA, 2011. ACM.

[101] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. In *2009 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–12, May 2009.

[102] Yi Guo, Jisheng Zhao, Vincent Cavé, and Vivek Sarkar. SLAW: A Scalable Locality-aware Adaptive Work-stealing Scheduler. In *2010 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–12, April 2010.

[103] Pablo Halpern. Strict Fork-Join Parallelism. C++ Standard Committee Papers: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3409.pdf` (last visited July 5, 2016), September 2012.

[104] Pablo Halpern. Cilk runtime, first impressions.... Cplex mailing list: `http://www.open-std.org/pipermail/cplex/2014-January/000303.html` (last visited July 5, 2016), January 2014.

[105] Pablo Halpern. Cilk runtime, first impressions.... Cplex mailing list: `http://www.open-std.org/pipermail/cplex/2014-January/000310.html` (last visited July 5, 2016), January 2014.

[106] Robert H. Halstead, Jr. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 9–17, New York, NY, USA, 1984. ACM.

[107] Robert H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.*, 7:501–538, October 1985.

[108] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding Sources of Inefficiency in General-Purpose Chips. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 37–47, New York, NY, USA, 2010. ACM.

[109] Robert Harper. Parallelism is not Concurrency. `http://existentialtype.wordpress.com/2011/03/17/parallelism-is-not-concurrency` (last visited July 5, 2016), March 2011.

[110] Jim Held. "Single-chip Cloud Computer", an IA Tera-scale Research Processor. In *Proceedings of the 2010 Conference on Parallel Processing*, Euro-Par 2010, pages 85–85, Berlin, Heidelberg, 2011. Springer-Verlag.

[111] Danny Hendler and Nir Shavit. Non-Blocking Steal-Half Work Queues. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 280–289, New York, NY, USA, 2002. ACM.

[112] Danny Hendler and Nir Shavit. Work Dealing. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '02, pages 164–172, New York, NY, USA, 2002. ACM.

[113] Maurice Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[114] Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, November 1993.

[115] Maurice Herlihy and Zhiyu Liu. Well-Structured Futures and Cache Locality. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 155–166, New York, NY, USA, 2014. ACM.

[116] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[117] Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. Backtracking-based Load Balancing. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 55–64, New York, NY, USA, 2009. ACM.

[118] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[119] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. Remote Memory Access Programming in MPI-3. *ACM Trans. Parallel Comput.*, 2(2):9:1–9:26, June 2015.

[120] Ralf Hoffmann, Andreas Prell, and Thomas Rauber. Dynamic Task Scheduling and Load Balancing on Cell Processors. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 205–212, Washington, DC, USA, 2010. IEEE Computer Society.

[121] Ralf Hoffmann, Andreas Prell, and Thomas Rauber. Exploiting Fine-Grained Parallelism on Cell Processors. In Pasqua D'Ambra, Mario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing*, volume 6272 of *Lecture Notes in Computer Science*, pages 175–186. Springer Berlin Heidelberg, 2010.

[122] Ralf Hoffmann and Thomas Rauber. Fine-Grained Task Scheduling Using Adaptive Data Structures. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par '08, pages 253–262, Berlin, Heidelberg, 2008. Springer-Verlag.

[123] H.Peter Hofstee. Heterogeneous Multi-core Processors: The Cell Broadband Engine. In Stephen W. Keckler, Kunle Olukotun, and H. Peter Hofstee, editors, *Multicore Processors and Systems*, Integrated Circuits and Systems, pages 271–295. Springer US, 2009.

[124] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabrice Pailet, Shailendra Jain, Tiju Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas Apel, Kersten Henriss, Tor Lund-Larsen, Sebastian Steibl, Shekar Borkar, Vivek De, Rob van der Wijngaart, and Timothy Mattson. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *Proceedings of the 2010 IEEE International Solid-State Circuits Conference*, pages 108–109, 2010.

[125] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 3 (3A, 3B, 3C & 3D): System Programming Guide. April 2016.

[126] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[127] Daniel R. Johnson, Matthew R. Johnson, John H. Kelm, William Tuohy, Steven S. Lumetta, and Sanjay J. Patel. Rigel: A 1,024-Core Single-Chip Accelerator Architecture. *IEEE Micro*, 31(4):30–41, 2011.

[128] Hartmut Kaiser. Plain Threads are the GOTO of Today's Computing. `http://stellar.cct.lsu.edu/pubs/Plain_Threads_are_the_GOTO_of_Todays_Computing_MeetingCpp_2014.pdf` (last visited July 5, 2016), December 2014.

[129] John H. Kelm, Daniel R. Johnson, Matthew R. Johnson, Neal C. Crago, William Tuohy, Aqeel Mahesri, Steven S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 140–151, New York, NY, USA, 2009. ACM.

[130] Gregor Kiczales. Towards a New Model of Abstraction in Software Engineering. In *Proceedings of the International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*, pages 1–11, 1992.

[131] Matthias Korch and Thomas Rauber. A Comparison of Task Pools for Dynamic Load Balancing of Irregular Algorithms. *Concurrency and Computation: Practice and Experience*, 16(1):1–47, 2004.

[132] David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: A High-performance Parallel Lisp. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 81–90, New York, NY, USA, 1989. ACM.

[133] Yossi Kreinin. Parallelism and concurrency need different tools. `http://yosefk.com/blog/parallelism-and-concurrency-need-different-tools.html` (last visited July 5, 2016), May 2013.

[134] Rakesh Kumar, Timothy G. Mattson, Gilles Pokam, and Rob Van Der Wijngaart. The Case for Message Passing on Many-Core Chips. In Michael Hübner and Jürgen Becker, editors, *Multiprocessor System-on-Chip*, pages 115–123. Springer New York, 2011.

[135] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous Chip Multiprocessors. *Computer*, 38(11):32–38, November 2005.

[136] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 162–173, New York, NY, USA, 2007. ACM.

[137] Vivek Kumar, Daniel Frampton, Stephen M. Blackburn, David Grove, and Olivier Tardieu. Work-stealing Without the Baggage. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 297–314, New York, NY, USA, 2012. ACM.

[138] James LaGrone, Ayodunni Aribuki, Cody Addison, and Barbara Chapman. A Runtime Implementation of OpenMP Tasks. In Barbara M. Chapman, William D. Gropp, Kalyan Kumaran, and Matthias S. Müller, editors, *OpenMP in the Petascale Era*, volume 6665 of *Lecture Notes in Computer Science*, pages 165–178. Springer Berlin Heidelberg, 2011.

[139] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 63–74, New York, NY, USA, 1991. ACM.

[140] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[141] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[142] Leslie Lamport. Specifying Concurrent Program Modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, April 1983.

[143] Koen Langendoen, John Romein, Raoul Bhoedjang, and Henri Bal. Integrating Polling, Interrupts, and Thread Management. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '96, pages 13–, Washington, DC, USA, 1996. IEEE Computer Society.

[144] Nhat Minh Lê, Adrien Guatto, Albert Cohen, and Antoniu Pop. Correct and Efficient Bounded FIFO Queues. In *Proceedings of the 2013 25th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '13, pages 144–151, Washington, DC, USA, 2013. IEEE Computer Society.

[145] Doug Lea. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM.

[146] Edward A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, May 2006.

[147] I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson. Using Memory Mapping to Support Cactus Stacks in Work-stealing Runtime Systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 411–420, New York, NY, USA, 2010. ACM.

[148] Sanghoon Lee, Devesh Tiwari, Yan Solihin, and James Tuck. HAQu: Hardware-Accelerated Queueing for Fine-Grained Threading on a Chip Multiprocessor. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 99–110, Feb 2011.

[149] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The Design of a Task Parallel Library. OOPSLA '09, pages 227–242, 2009.

[150] Ted Leung. A Survey of Concurrency Constructs. `http://www.oscon.com/oscon2009/public/schedule/proceedings` (last visited July 5, 2016), July 2009.

[151] Hans-Wolfgang Loidl and Kevin Hammond. On the Granularity of Divide-and-Conquer Parallelism. In *Proceedings of the 1995 International Conference on Functional Programming*, FP'95, pages 135–144, Swinton, UK, UK, 1995. British Computer Society.

[152] Wenjing Ma and Sriram Krishnamoorthy. Data-driven Fault Tolerance for Work Stealing Computations. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 79–90, New York, NY, USA, 2012. ACM.

[153] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 161–171, June 2000.

[154] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xin-Min Tian. Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA '96, pages 179–188, New York, NY, USA, 1996. ACM.

[155] Simon Marlow. Parallelism /= Concurrency. `http://ghcmutterings.wordpress.com/2009/10/06/parallelism-concurrency` (last visited July 5, 2016), October 2009.

[156] Simon Marlow. Parallel and Concurrent Programming in Haskell. In *Proceedings of the 4th Summer School Conference on Central European Functional Programming School*, CEFP'11, pages 339–401, Berlin, Heidelberg, 2012. Springer-Verlag.

[157] Simon Marlow, Ryan Newton, and Simon Peyton Jones. A Monad for Deterministic Parallelism. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 71–82, New York, NY, USA, 2011. ACM.

[158] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime Support for Multicore Haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 65–78, New York, NY, USA, 2009. ACM.

[159] Andrey Marochko. TBB Scheduler Clandestine Evolution. `https://software.intel.com/en-us/blogs/2010/12/09/tbb-scheduler-clandestine-evolution` (last visited July 5, 2016), December 2010.

[160] Andrey Marochko and Alexey Kukanov. Composable Parallelism Foundations in the Intel Threading Building Blocks Task Scheduler. In *Applications, Tools and Techniques on the Road to Exascale Computing, Proceedings of the conference ParCo 2011, 31 August - 3 September 2011, Ghent, Belgium*, pages 545–554, 2011.

[161] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why On-chip Cache Coherence is Here to Stay. *Commun. ACM*, 55(7):78–89, July 2012.

[162] Jeff Matocha and Tracy Camp. A Taxonomy of Distributed Termination Detection Algorithms. *J. Syst. Softw.*, 43(3):207–221, November 1998.

[163] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming.* Addison-Wesley Professional, first edition, 2004.

[164] Timothy G. Mattson, Rob Van der Wijngaart, and Michael Frumkin. Programming the Intel 80-core network-on-a-chip Terascale Processor. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 38:1–38:11, Piscataway, NJ, USA, 2008. IEEE Press.

[165] Timothy G. Mattson, Rob F. van der Wijngaart, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. The 48-core SCC Processor: the Programmer's View. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[166] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[167] Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14.* O'Reilly Media, Incorporated, 2014.

[168] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[169] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent Work Stealing. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 45–54, New York, NY, USA, 2009. ACM.

[170] Seung-Jai Min, Costin Iancu, and Katherine Yelick. Hierarchical Work Stealing on Manycore Clusters. In *In Fifth Conference on Partitioned Global Address Space Programming Models*, PGAS '11, 2011.

[171] Michael Mitzenmacher. Analyses of Load Stealing Models Based on Differential Equations. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 212–221, New York, NY, USA, 1998. ACM.

[172] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 185–197, New York, NY, USA, 1990. ACM.

[173] Gordon E. Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, 38(8), April 1965.

[174] José E. Moreira, Dale Schouten, and Constantine Polychronopoulos. The Performance Impact of Granularity Control and Functional Parallelism. In Chua-Huang Huang, Ponnuswamy Sadayappan, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, volume 1033 of *Lecture Notes in Computer Science*, pages 581–597. Springer Berlin Heidelberg, 1996.

[175] Adam Morrison and Yehuda Afek. Fast Concurrent Queues for x86 Processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 103–112, New York, NY, USA, 2013. ACM.

[176] Adam Morrison and Yehuda Afek. Fence-Free Work Stealing on Bounded TSO Processors. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 413–426, New York, NY, USA, 2014. ACM.

[177] David Mosberger. Memory Consistency Models. *SIGOPS Oper. Syst. Rev.*, 27(1):18–26, January 1993.

[178] Cornelius Müller, David Camp, Bernd Hentschel, and Christoph Garth. Distributed Parallel Particle Advection using Work Requesting. In *Large-Scale Data Analysis and Visualization (LDAV), 2013 IEEE Symposium on*, pages 1–6, Oct 2013.

[179] Todd Mytkowicz and Wolfram Schulte. Waiting for Godot? The Right Language Abstractions for Parallel Programming Should Be Here Soon. *Ubiquity*, 2014(June):4:1–4:12, June 2014.

[180] Ramadass Nagarajan, Karthikeyan Sankaralingam, Doug Burger, and Stephen W. Keckler. A Design Space Evaluation of Grid Processor Architectures. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 40–51, Washington, DC, USA, 2001. IEEE Computer Society.

[181] Vijay Nagarajan and Rajiv Gupta. ECMon: Exposing Cache Events for Monitoring. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 349–360, New York, NY, USA, 2009. ACM.

[182] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 22–31, New York, NY, USA, 2007. ACM.

[183] Jarek Nieplocha and Bryan Carpenter. ARMCI: A Portable Remote Memory Copy Libray for Distributed Array Libraries and Compiler Run-Time Systems. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 533–546, London, UK, UK, 1999. Springer-Verlag.

[184] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. UTS: An Unbalanced Tree Search Benchmark. In *Proceedings of the 19th international conference on Languages and compilers for parallel computing*, LCPC '06, pages 235–250, Berlin, Heidelberg, 2007. Springer-Verlag.

[185] Stephen Olivier and Jan Prins. Scalable Dynamic Load Balancing Using UPC. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 123–131, Washington, DC, USA, 2008. IEEE Computer Society.

[186] Stephen L. Olivier and Jan F. Prins. Evaluating OpenMP 3.0 Run Time Systems on Unbalanced Task Graphs. In *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, IWOMP '09, pages 63–78, Berlin, Heidelberg, 2009. Springer-Verlag.

[187] Stephen L. Olivier and Jan F. Prins. Comparison of OpenMP 3.0 and Other Task Parallel Frameworks on Unbalanced Task Graphs. *International Journal of Parallel Programming*, 38(5):341–360, 2010.

[188] Daniel Orozco, Elkin Garcia, Robert Pavel, Rishi Khan, and GuangR. Gao. Polytasks: A Compressed Task Representation for HPC Runtimes. In Sanjay Rajopadhye and Michelle Mills Strout, editors, *Languages and Compilers for Parallel Computing*, volume 7146 of *Lecture Notes in Computer Science*, pages 268–282. Springer Berlin Heidelberg, 2013.

[189] John D. Owens, William J. Dally, Ron Ho, D.N. Jayasimha, Stephen W. Keckler, and Li-Shiuan Peh. Research Challenges for On-Chip Interconnection Networks. *Micro, IEEE*, 27(5):96–108, Sept 2007.

[190] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[191] Jeeva Paudel, Olivier Tardieu, and José Nelson Amaral. Hybrid Parallel Task Placement in X10. In *Proceedings of the Third ACM SIGPLAN X10 Workshop*, X10 '13, pages 31–38, New York, NY, USA, 2013. ACM.

[192] Li-Shiuan Peh, Stephen W. Keckler, and Sriram Vangal. On-Chip Networks for Multicore Systems. In Stephen W. Keckler, Kunle Olukotun, and H. Peter Hofstee, editors, *Multicore Processors and Systems*, Integrated Circuits and Systems, pages 35–71. Springer US, 2009.

[193] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[194] Swann Perarnau and Mitsuhisa Sato. Victim Selection and Distributed Work Stealing Performance: A Case Study. In *Proceedings of the 2014 IEEE 28th International Parallel and*

Distributed Processing Symposium, IPDPS '14, pages 659–668, Washington, DC, USA, 2014. IEEE Computer Society.

[195] Simon Peter, Adrian Schüpbach, Dominik Menzi, and Timothy Roscoe. Early experience with the Barrelfish OS and the Single-Chip Cloud Computer. In Göhringer et al. [98], pages 35–39.

[196] Darko Petrović, Thomas Ropars, and André Schiper. Leveraging Hardware Message Passing for Efficient Thread Synchronization. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 143–154, New York, NY, USA, 2014. ACM.

[197] Rob Pike. Concurrency is not Parallelism. `http://talks.golang.org/2012/waza.slide#1` (last visited July 5, 2016), January 2012.

[198] Artur Podobas, Mats Brorsson, and Karl-Filip Faxén. A Comparative Performance Study of Common and Popular Task-centric Programming Frameworks. *Concurr. Comput. : Pract. Exper.*, 27(1):1–28, January 2015.

[199] Constantine D. Polychronopoulos and David J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Trans. Comput.*, 36(12):1425–1439, December 1987.

[200] Andreas Prell and Thomas Rauber. Task Parallelism on the SCC. In Göhringer et al. [98], pages 65–67.

[201] Andreas Prell and Thomas Rauber. Go's Concurrency Constructs on the SCC. In *MARC Symposium*, pages 2–6, 2012.

[202] Michael Alan Rainey. *Effective Scheduling Techniques for High-Level Parallel Programming Languages.* PhD thesis, Department of Computer Science, University of Chicago, August 2010.

[203] Thomas Rauber and Gudula Rünger. *Parallel Programming for Multicore and Cluster Systems.* Springer Publishing Company, Incorporated, 2nd edition, 2013.

[204] Kaushik Ravichandran, Sangho Lee, and Santosh Pande. Work Stealing for Multi-core HPC Clusters. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11, pages 205–217, Berlin, Heidelberg, 2011. Springer-Verlag.

[205] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism.* O'Reilly Media, 2007.

[206] John Reppy and Yingqi Xiao. Specialization of CML Message-Passing Primitives. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 315–326, New York, NY, USA, 2007. ACM.

[207] John H. Reppy. CML: A Higher-order Concurrent Language. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 293–305, New York, NY, USA, 1991. ACM.

[208] Arch Robison. A Primer on Scheduling Fork-Join Parallelism with Work Stealing. C++ Standard Committee Papers: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3872.pdf` (last visited July 5, 2016), January 2014.

[209] Arch Robison, Michael Voss, and Alexey Kukanov. Optimization via Reflection on Work Stealing in TBB. In *2008 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, April 2008.

[210] Arch D. Robison and Ralph E. Johnson. Three Layer Cake for Shared-memory Programming. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, pages 5:1–5:8, New York, NY, USA, 2010. ACM.

[211] Demian Rosas-Ham, Isuru Herath, Paraskevas Yiapanis, Mikel Luján, and Ian Watson. Architectural Support for Exploiting Fine Grain Parallelism. In *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, pages 61–70, June 2012.

[212] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[213] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, Cambridge, MA, USA, 2004.

[214] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '91, pages 237–245, New York, NY, USA, 1991. ACM.

[215] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. Flexible Architectural Support for Fine-Grain Scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 311–322, New York, NY, USA, 2010. ACM.

[216] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. The Asynchronous Partitioned Global Address Space Model. In *Proceedings of The First Workshop on Advances in Message Passing*, AMP '10, June 2010.

[217] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 Language Specification, Version 2.4. `http://x10-lang.org` (last visited July 5, 2016), May 2014.

[218] Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove, and Sriram Krishnamoorthy. Lifeline-based Global Load Balancing. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 201–212, New York, NY, USA, 2011. ACM.

[219] Vivek Sarkar, William Harrod, and Allan E Snavely. Software Challenges in Extreme Scale Systems. In *Journal of Physics: Conference Series*, page 012045, 2009.

[220] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Hideki Saito, Rakesh Krishnaiyer, Mikhail Smelyanskiy, Milind Girkar, and Pradeep Dubey. Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications? In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 440–451, Washington, DC, USA, 2012. IEEE Computer Society.

[221] Michael L. Scott. Shared-Memory Synchronization. *Synthesis Lectures on Computer Architecture*, 8(2):1–221, 2013.

[222] Aniruddha G. Shet, Vinod Tipparaju, and Robert J. Harrison. Asynchronous Programming in UPC: A Case Study and Potential for Improvement. In *APGAS'09: Proceedings of the 1st Workshop on Asynchrony in the PGAS Programming Model, co-located with the 23rd International Conference on Supercomputing (ICS)*, Yorktown Heights, NY, 2009.

[223] K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. MultiMLton: A Multicore-aware Runtime for Standard ML. *Journal of Functional Programming*, 24:613–674, 2014.

[224] Matthew Sottile, Timothy G. Mattson, and Craig E. Rasmussen. *Introduction to Concurrency in Programming Languages*. Chapman & Hall/CRC, 1st edition, 2009.

[225] Michael F. Spear, Arrvindh Shriraman, Hemayet Hossain, Sandhya Dwarkadas, and Michael L. Scott. Alert-on-Update: A Communication Aid for Shared Memory Multiprocessors. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 132–133, New York, NY, USA, 2007. ACM.

[226] Ellen Spertus, Seth Copen Goldstein, Klaus Erik Schauser, Thorsten von Eicken, David E. Culler, and William J. Dally. Evaluation of Mechanisms for Fine-grained Parallel Programs in the J-machine and the CM-5. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 302–313, New York, NY, USA, 1993. ACM.

[227] Joel Spolsky. The Law of Leaky Abstractions. `http://www.joelonsoftware.com/articles/LeakyAbstractions.html` (last visited July 5, 2016), November 2002.

[228] Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. Beyond Nested Parallelism: Tight Bounds on Work-Stealing Overheads for Parallel Futures. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 91–100, New York, NY, USA, 2009. ACM.

[229] Mark S. Squillante and Randolph D. Nelson. Analysis of Task Migration in Shared-memory Multiprocessor Scheduling. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '91, pages 143–155, New York, NY, USA, 1991. ACM.

[230] Bjarne Stroustrup. *A Tour of C++*. Addison-Wesley Professional, 2013.

[231] Volker Strumpen. Indolent Closure Creation. Technical report, Cambridge, MA, USA, 1998.

[232] Jim Sukha. Brief Announcement: A Lower Bound for Depth-Restricted Work Stealing. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 124–126, New York, NY, USA, 2009. ACM.

[233] Herb Sutter and James Larus. Software and the Concurrency Revolution. *Queue*, 3(7):54–62, September 2005.

[234] Olivier Tardieu, Benjamin Herta, David Cunningham, David Grove, Prabhanjan Kambadur, Vijay Saraswat, Avraham Shinnar, Mikio Takeuchi, and Mandana Vaziri. X10 and APGAS at Petascale. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 53–66, New York, NY, USA, 2014. ACM.

[235] Olivier Tardieu, Haichuan Wang, and Haibo Lin. A Work-Stealing Scheduler for X10's Task Parallelism with Suspension. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 267–276, New York, NY, USA, 2012. ACM.

[236] Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. StackThreads/MP: Integrating Futures into Calling Standards. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99, pages 60–71, New York, NY, USA, 1999. ACM.

[237] Michael B. Taylor. Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1131–1136, June 2012.

[238] Michael B. Taylor, Walter Lee, Jason E. Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul R. Johnson, Jason S. Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matthew I. Frank, Saman Amarasinghe, and Anant Agarwal. Tiled Multicore Processors. In Stephen W. Keckler, Kunle Olukotun, and H. Peter Hofstee, editors, *Multicore Processors and Systems*, pages 1–33. Springer Science+Business Media, LLC, 2009.

[239] Christian Terboven, Dirk Schmidl, Tim Cramer, and Dieter an Mey. Task-Parallel Programming on NUMA Architectures. In Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 638–649. Springer Berlin Heidelberg, 2012.

[240] Xavier Teruel, Michael Klemm, Kelvin Li, Xavier Martorell, StephenL. Olivier, and Christian Terboven. A Proposal for Task-Generating Loops in OpenMP*. In Alistair P. Rendell, Barbara M. Chapman, and Matthias S. Müller, editors, *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 1–14. Springer Berlin Heidelberg, 2013.

[241] Xavier Teruel, Priya Unnikrishnan, Xavier Martorell, Eduard Ayguadé, Raul Silvera, Guansong Zhang, and Ettore Tiotto. OpenMP Tasks in IBM XL Compilers. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, pages 16:207–16:221, New York, NY, USA, 2008. ACM.

[242] Peter Thoman, Herbert Jordan, and Thomas Fahringer. Adaptive Granularity Control in Task Parallel Programs Using Multiversioning. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, pages 164–177, Berlin, Heidelberg, 2013. Springer-Verlag.

[243] John Thornley. Performance of a Class of Highly-Parallel Divide-and-Conquer Algorithms. Technical report, Pasadena, CA, USA, 1995.

[244] Aaron Turon. *Understanding and Expressing Scalable Concurrency.* PhD thesis, College of Computer and Information Science, Northeastern University, April 2013.

[245] Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. Lazy Binary-Splitting: A Run-Time Adaptive Work-Stealing Scheduler. PPoPP '10, pages 179–190, 2010.

[246] Alexandros Tzannes, George C. Caragea, Uzi Vishkin, and Rajeev Barua. Lazy Scheduling: A Runtime Adaptive Scheduler for Declarative Parallelism. *ACM Trans. Program. Lang. Syst.*, 36(3):10:1–10:51, September 2014.

[247] Tom van Dijk and Jaco C. van de Pol. Lace: Non-blocking Split Deque for Work-Stealing. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8806 of *Lecture Notes in Computer Science*, pages 206–217. Springer International Publishing, 2014.

[248] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient Load Balancing for Wide-Area Divide-and-Conquer Applications. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 34–43, New York, NY, USA, 2001. ACM.

[249] Philippe Virouleau, Pierrick Brunet, François Broquedis, Nathalie Furmento, Samuel Thibault, Olivier Aumage, and Thierry Gautier. *Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite*, pages 16–29. Springer International Publishing, Cham, 2014.

[250] Uzi Vishkin. Using Simple Abstraction to Reinvent Computing for Parallelism. *Commun. ACM*, 54(1):75–85, January 2011.

[251] David B. Wagner and Bradley G. Calder. Leapfrogging: A Portable Technique for Implementing Efficient Futures. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '93, pages 208–217, New York, NY, USA, 1993. ACM.

[252] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring It All to Software: Raw Machines. *Computer*, 30(9):86–93, Sep 1997.

[253] Ivan Walulya, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. Concurrent Data Structures in Architectures with Limited Shared Memory Support. In Luís Lopes, Julius Žilinskas, Alexandru Costan, Roberto G. Cascella, Gabor Kecskemeti, Emmanuel Jeannot, Mario Cannataro, Laura Ricci, Siegfried Benkner, Salvador Petit, Vittorio Scarano, José Gracia, Sascha Hunold, Stephen L. Scott, Stefan Lankes, Christian Lengauer, Jesus Carretero, Jens Breitbart, and Michael Alexander, editors, *Euro-Par 2014: Parallel Processing Workshops*, volume 8805 of *Lecture Notes in Computer Science*, pages 189–200. Springer International Publishing, 2014.

[254] Lei Wang, Huimin Cui, Yuelu Duan, Fang Lu, Xiaobing Feng, and Pen-Chung Yew. An Adaptive Task Creation Strategy for Work-Stealing Scheduling. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 266–277, New York, NY, USA, 2010. ACM.

[255] Xingzhi Wen and Uzi Vishkin. FPGA-Based Prototype of a PRAM-On-Chip Processor. In *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, pages 55–66, New York, NY, USA, 2008. ACM.

[256] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27(5):15–31, September 2007.

[257] Anthony Williams. *C++ Concurrency in Action: Practical Multithreading.* Manning Publications, Shelter Island, NY, 2012.

[258] Alexander Wirz, Michael Süß, and Claudia Leopold. *A Comparison of Task Pool Variants in OpenMP and a Proposal for a Solution to the Busy Waiting Problem*, pages 397–408. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[259] Chenning Xie, Zhijun Hao, and Haibo Chen. X10-FT: Transparent Fault Tolerance for APGAS Language and Runtime. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '13, pages 11–20, New York, NY, USA, 2013. ACM.

[260] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing*, LCPC'09, pages 172–187, Berlin, Heidelberg, 2010. Springer-Verlag.

[261] Chaoran Yang and John Mellor-Crummey. A Wait-free Queue as Fast as Fetch-and-Add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 16:1–16:13, New York, NY, USA, 2016. ACM.

[262] Junfeng Yang, Heming Cui, and Jingyue Wu. Determinism Is Overrated: What Really Makes Multithreaded Programs Hard to Get Right and What Can Be Done About It? In *Presented as part of the 5th USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2013. USENIX.

[263] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. Productivity and Performance Using Partitioned Global Address Space Languages. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07, pages 24–32, New York, NY, USA, 2007. ACM.

[264] Richard M. Yoo, Christopher J. Hughes, Changkyu Kim, Yen-Kuang Chen, and Christos Kozyrakis. Locality-aware Task Management for Unstructured Parallelism: A Quantitative Limit Study. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 315–325, New York, NY, USA, 2013. ACM.

[265] Wei Zhang, Olivier Tardieu, David Grove, Benjamin Herta, Tomio Kamada, Vijay Saraswat, and Mikio Takeuchi. GLB: Lifeline-based Global Load Balancing Library in X10. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, PPAA '14, pages 31–40, New York, NY, USA, 2014. ACM.