Channel-based Work Stealing

Andreas Prell and Thomas Rauber

Department of Computer Science
University of Bayreuth, Germany
{andreas.prell,thomas.rauber}@uni-bayreuth.de

Abstract. Tasks have become a useful abstraction in parallel programming; one that relies on runtime systems to provide efficient task scheduling and load balancing. The scheduling strategy of choice is work stealing, in which idle workers try to take some work from busy workers. Current work-stealing schedulers are dependent on shared state in the form of concurrent task queues, most often deques, while other implementations of work stealing based on message passing have largely been geared towards distributed environments. This paper explores an alternative message-passing approach to work stealing that is centered around communication over channels, concurrent building blocks dating back to the idea of Communicating Sequential Processes. We propose a new scheduler, in which worker threads operate on private deques and communicate with each other by sending steal requests and tasks over channels; no shared state is assumed internally. Our benchmark results on a 48-core multiprocessor and a 60-core Intel Xeon Phi show that there is no significant slowdown due to channel communication: on average, the channel-based scheduler is only a few percent slower than a scheduler based on concurrent deques.

1 Introduction

Tasks have become a useful abstraction in parallel programming; one that allows programmers to focus on finding potential parallelism and leave the job of managing it to the runtime system. The scheduling strategy of choice is work stealing, in which idle workers, called thieves, try to "steal" work from busy workers, called victims, thereby load balancing a computation [8]. Work stealing was popularized by Cilk [16] and has since found its way into many task-parallel programming systems [29, 31, 23] and languages [10, 9, 4].

Work stealing was designed to work well on conventional shared-memory machines. Typical implementations use a set of concurrent data structures, most often double-ended queues (deques), that are accessed by all workers in the system. Implementations for distributed systems are either directly based on MPI [12, 30] or layered on top of a global address space abstraction [28, 13]. In both cases, the communication involved is very different from a shared-memory implementation. Schedulers that store tasks in private deques, even though shared memory is available, have been shown to achieve good performance while avoiding expensive synchronization operations required by concurrent deques [5].

Recent trends in programming languages include a renewed interest in explicit communication. Modern concurrent languages, such as Go [1] and Rust [2], support lightweight threads, which communicate by sending messages over channels, rather than reading from and writing to shared variables. The idea of channels goes back to Tony Hoare's Communicating Sequential Processes (CSP) [20]. Originally meant to synchronize execution of independently running processes, channels are useful for both synchronous and asynchronous messaging.

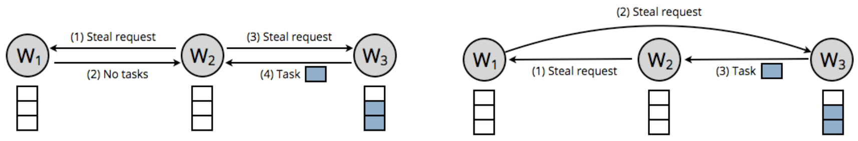
On the processor architecture side, we can expect to see a rise of new designs, as inefficiencies in general-purpose chips are increasing [17, 15]. A recent example is Intel's Single-Chip Cloud Computer (SCC) [24]. Although a research chip and not a commercial product, the SCC provides a glimpse of how manycore chips might evolve in the future: many smaller cores arranged in a grid, exchanging messages to communicate and share data. With increasing core counts, the case for "sharing by communicating" is only getting stronger [21].

In this paper, we explore work stealing based on channels. Similar to other message-passing schedulers, all communication happens through sending messages over channels; no shared state is assumed internally, a valuable property for the portability of the scheduler. Since task-parallel programming depends on efficient runtime system support, we deem it unacceptable if channel communication incurred significant overhead and a loss of scalability and performance. We test our scheduler on two shared-memory multiprocessors to see how it fares against work stealing with concurrent deques. The results outlined in this paper make us confident that channel-based work stealing is a viable alternative to previous work-stealing implementations.

Our main contributions are thus: (1) We present a new work-stealing scheduler centered around private task queues and explicit communication over channels. Channels, inspired by recent programming languages, such as Go, are a lightweight message passing abstraction, simple enough to allow the scheduler to be ported to other platforms in the future. Work-stealing requests, tasks, and termination detection messages are all delivered over channels. (2) We evaluate our scheduler on two shared-memory multiprocessors, a 48-core AMD Opteron SMP and a 60-core Intel Xeon Phi, by comparing performance with a work-stealing scheduler based on concurrent deques. We find that channel communication comes at a reasonable cost in terms of performance: in most benchmarks, the channel-based scheduler is 3-7% slower than the deque-based scheduler. In a few benchmarks, however, the situation is reversed, and the channel-based scheduler manages to outperform the deque-based scheduler.

2 Channel-based Work Stealing

Channels, in the original sense of CSP, serve the dual purpose of communication *and* synchronization: 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 channel sends and receives should never block



(a) Reactive work-stealing (b) Proactive work-stealing

Fig. 1. 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 avoids any acknowledgment messages; steal requests keep being forwarded until tasks are found. Steal requests and tasks are sent over separate channels.

worker threads from making progress. The Go programming language, for example, supports buffered channels for asynchronous sends and receives.

Channels behave like FIFO queues and hence preserve message order. We use only a small number of channel functions, including alloc/free, send/receive, and peek. Unbuffered channels that serve as synchronization points between threads need not be supported; buffered channels are the default.

2.1 Steal Request Messages

Work stealing without sharing task queues requires cooperation between victims and thieves. Rather than dealing out tasks on a regular basis [18], workers send tasks in reaction to steal requests they receive. A steal request is fundamentally a message containing the thief's ID and a channel for sending tasks from victim to thief. Additional information concerning the victim selection strategy or the number of requested tasks may be included in a steal request. To bound the number of steal requests in the system, we allow only one steal request per worker to be pending at any given time. Every worker receives steal requests and tasks independently of other workers, so that there is never more than one receiver per channel, which in turn enables efficient channel implementations, for example, when shared memory is available. The total number of channels grows linearly with the number of workers: n workers communicate over $2n$ channels.

Unlike in other schedulers designed for use in message-passing environments [12, 30], workers do not wait for an acknowledgment after sending a steal request to some victim. This design decision reduces the number of messages and allows workers to send steal requests ahead of time, before running out of local work.

Figure 1 illustrates our approach. Suppose worker W_1 receives a steal request from W_2, but has zero tasks left. Rather than returning the steal request to W_2, saying the steal has failed (Figure 1(a)), W_1 forwards the steal request to another potential victim, W_3 (Figure 1(b)). If W_3 has tasks to spare, it can send some using the channel contained in the steal request. Otherwise, W_3 must forward the steal request, or, if there is no potential victim left, send it back to W_2 (for reasons discussed in the next section). Effectively, W_2 can start stealing *before* it strictly needs to. For instance, W_2 can send a steal request after dequeuing the last task and before executing it. Ideally, when W_2 runs out of tasks, new work has already arrived, and W_2 can continue without delay.

Randomly selecting some worker as a target for a steal request may be efficient when tasks are equally distributed among workers. However, if only a few workers have tasks left, steal requests are likely to pass through idle workers, requiring more channel operations along the way and increasing overall work-stealing latency. In deque-based work stealing, 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 implement a similar strategy using channels, we allow channels to be closed, like in the Go language, and, in addition, to be reopened. Unlike in Go, however, closing a channel means no further values will be *received* from the channel (rather than sent to the channel), conveying to potential senders that this channel should be skipped. Channels remain open as long as workers have tasks to share. Closed channels are reopened once tasks become available again.

2.2 Termination Detection

Workers are either active executing tasks or inactive searching for tasks. Termination detection is the problem of determining when all workers are inactive, which means a task-parallel computation has finished or can proceed to the next phase. Because of the nature of work stealing, inactive workers may become active again at any point in time, as long as there are tasks in the system.

Termination detection is relatively straightforward to implement with shared memory [19]. In our runtime, however, workers communicate exclusively over channels; there is no shared state that can be used to collect information about the system. Instead, we take the following approach: One worker keeps track of steal requests and decides whether termination has occurred yet by counting the number of inactive workers. We call this worker *manager*. There are two complicating factors: (1) workers may send steal requests while still being active (result of proactive stealing) and (2) work stealing happens between workers, without the manager's knowledge (a prerequisite for scalability). To deal with (1), we extend the information contained in a steal request with a boolean `idle` that tells whether a thief is in fact idle. Steal requests are eventually returned to their original senders if no tasks could be found, allowing thieves to update their steal requests to reflect any state change. Only a steal request with `idle == true` counts towards the number of inactive workers. (2) may lead to early termination detection, violating the safety property. Consider the following situation: Worker I is idle, but is about to receive new tasks from worker J. As a result, I changes state from idle to working. The manager, however, let's say worker K, has no way of knowing that I is no longer idle. Assuming J runs out of tasks and becomes idle shortly after, the manager may falsely conclude that termination has occurred. The solution to this problem is to inform the manager of the worker's state change: whenever a worker sends a task to another worker that is idle, it must also send an update message to the manager.

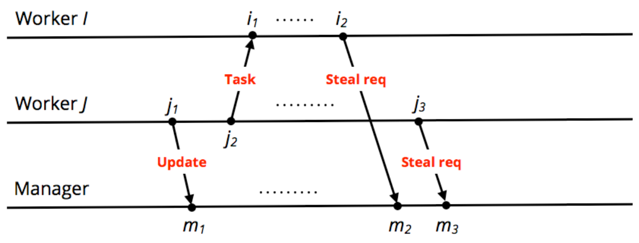


Fig. 2. Updating the manager about the state of workers. i_x, j_x, and m_x are events denoting the sending or the receipt of a message. The receipt of the update message m_1 is guaranteed to happen before the receipt of any subsequent steal request from worker I or worker J.

Figure 2 illustrates a possible ordering of message-send and message-receive events. Expressed in terms of "happened before" [22], $j_1 \rightarrow j_2$ (the update must be sent before any task) and $m_1 \rightarrow m_3$ (the update must be received before any subsequent steal request from worker J). The FIFO property of channels guarantees that $m_1 \rightarrow m_2$ and $m_1 \rightarrow m_3$, if and only if update and steal requests travel over the *same* channel to the manager. If we used separate channels, we would open the door for race conditions, which could lead to early termination detection by the manager as described above. Note that, if worker I is not known to be idle, there is no need to update the manager, and the message is omitted.

2.3 Synchronization

Synchronization is required to coordinate the execution of tasks. The task model we use is similar to that of OpenMP 3.0 and later [3, 7], with two primary synchronization constructs, a full task barrier and a task barrier for child tasks. The latter is built on top of *futures*, for which we need an implementation that does not rely on shared state.

The important insight is that futures can be viewed as channels. In other words, a future opens up 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 if the value is not computed yet.

Suppose we want to call function `f` asynchronously. `f` takes two integers as arguments and returns an integer as a result. Creating a task to run `f` and waiting for `f`'s result involves allocating a channel and storing a reference to the channel in the task descriptor (part of the `async` macro and not shown here):

```
Channel *ch = channel_alloc(sizeof(int));
async(f, a, b, ch);
...
while (!channel_receive(ch, &x, sizeof(int)) ;
```

Note that channel `ch` should be buffered, or otherwise the sender blocks until the matching receive occurs. The thread that scheduled the task uses the channel reference in the task descriptor to send the result to the waiting thread:

```
int tmp = f(a, b); channel_send(ch, &tmp, sizeof(int));
```

The implementation in our runtime differs slightly from the basic code above. We disallow busy-waiting until the future is set, so instead of trying to receive from the channel in a tight loop, we pass control to a runtime library function that schedules other work until the result is ready to be received:

```
Channel *ch = channel_alloc(sizeof(int));
async(f, a, b, ch);
...
await(ch, &x);
```

3 Experimental Evaluation

In this section, we show that our scheduler is practical and can achieve performance comparable with conventional work stealing on recent shared-memory systems. The implementation is tested on an AMD Opteron SMP and an Intel Xeon Phi coprocessor. The SMP is equipped with four Opteron 6172 processors, each with 12 cores running at 2.1 GHz, and 128 GB of memory. The Xeon Phi card has 60 cores clocked at 1.053 GHz and 8 GB of memory. Some of the benchmarks allow us to explicitly set the task size t. In these cases, we choose three different values for t, representing fine-grained, medium-grained, and coarse-grained parallelism: $10\mu s$, $100\mu s$, and $1000\mu s$. On the Xeon Phi, we double these task sizes to account for the lower clock speed (1.053 GHz vs 2.1 GHz). The benchmarks used are:

SPC A simple producer-consumer benchmark. A single worker produces n tasks, each running for time t. We choose $n = 100\,000$.

BPC A variation of a producer-consumer benchmark with two kinds of tasks, producer and consumer tasks [13]. Each producer task creates another producer task followed by n consumer tasks, until a certain depth d is reached. Consumer tasks run for time t. We choose $n = 9$ and $d = 10\,000$, a particularly challenging input for small values of t.

Treerec A simple tree-recursive computation, similar in structure to Fibonacci, which is often used to estimate task scheduling overheads [14]. 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 time t before returning. We choose $n = 25$, generating a total of 242 784 tasks.

Matmul Block matrix multiplication of two 2048×2048 matrices of doubles. The matrix block size is 32.

LU Block LU decomposition of a sparse 4096×4096 matrix of doubles. The code is based on the OpenMP version from the BOTS project [14]. The matrix block size is 64.

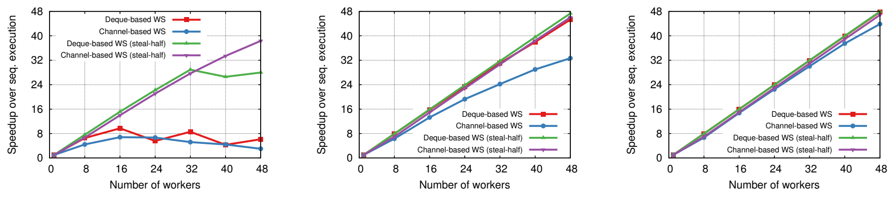
Quicksort A recursive algorithm that performs an in-place sort of an array of 100 million integers. For sub-arrays ≤ 100 elements, the algorithm falls back to using insertion sort, which is faster on small inputs.

UTS An algorithm that performs an exhaustive search on a highly unbalanced tree [27]. The tree is generated implicitly; each child node is constructed from the SHA-1 hash of the parent node and child index. As input, we choose geometric trees T1L and T2L and binomial tree T3L, which all have about 100 million nodes.

In the following text, we use the abbreviation DQ to refer to the reference work-stealing scheduler and CH to refer to the channel-based implementation. DQ has one concurrent deque per worker, selects victims at random, and implements the termination detection barrier described in [19]. For better comparison, DQ uses the same implementation of future tasks as CH; see Section 2.3 for details. CH uses two types of channels, as summarized in Table 1: multiple-producer, single-consumer (MPSC) channels for exchanging steal requests and single-producer, single-consumer (SPSC) channels for transferring tasks and task return values. A task contains a function pointer, the function's arguments, and a reference to its parent task. In both DQ and CH, a successful steal avoids copying heap-allocated tasks, but rather passes ownership of stolen tasks between workers by copying references only. Steal requests in CH are not heap-allocated and are thus copied when sent and received from channels. Update messages in CH are treated as steal requests; see Section 2.2 for why this is required. Only SPSC channels are wait-free; MPSC channels are blocking (sender side). Shared deques in DQ and channels in CH are implemented as circular arrays. The array sizes are large enough to prevent resizing at runtime. The private deques in CH use a simple list-based implementation. All code is compiled with `gcc -O3` (GCC 4.7.1, revision 189773) on the Opteron and `icc -O2 -mmic` (ICC 14.0.1.106, build 20131008) on the Xeon system. The reported execution times are the result of taking averages over 20 runs.

Table 1. Overview of messages in the channel-based scheduler

Message type	Description	Size (bytes)	Channel type
Steal request	Request task (work-stealing attempt)	32	MPSC
Idle updates	Tell manager a worker is no longer idle	32	MPSC
Task	Send task to requesting worker	8	SPSC
Task return value	Communicate task return value	depends	SPSC



(a) Task size $10\mu s$ (b) Task size $100\mu s$ (c) Task size $1000\mu s$

Fig. 3. SPC benchmark performance results of deque-based and channel-based work-stealing schedulers on the 4-way AMD Opteron SMP.

3.1 Importance of Chunking

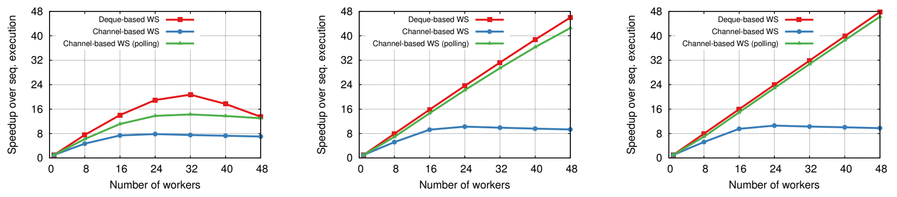
First, we take a closer look at SPC. Figure 3 shows speedups over sequential execution across different task sizes on the AMD Opteron SMP. We notice that CH does not scale as well as DQ, which already achieves near-optimal performance on $100\mu s$ tasks. Going further towards fine-grained parallelism, neither CH nor DQ are able to scale beyond 16 workers. At this point, the time it takes to distribute tasks from a single worker to all other workers outweighs the benefit of parallel execution. Stealing a single task only to run out of work again shortly after is obviously not the best strategy for scheduling fine-grained parallelism.

Figure 3 includes the results of stealing up to half of a victim's tasks with a single steal. Perhaps surprisingly, we see that CH scales better than DQ in Fig. 3(a). To understand this behavior, we examine work-stealing statistics collected at runtime. With 32 threads, roughly 3000 successful steals are counted in DQ, compared to 1500 in CH. With 48 threads, the number of steals has already increased by more than a factor of ten to 37 000 in DQ, whereas in CH, the number remains relatively low at 5000. In fact, 50% of all steals in DQ have a chunk size of 1 or 2; only 10% transfer more than 16 tasks at once. For comparison, 50% of all steals in CH have a chunk size between 1 and 12, and 10% succeed in chunking more than 130 tasks. On average, steal-half transfers larger chunks of tasks between workers in CH, resulting in less load balancing activity and better absolute performance than DQ.

3.2 Importance of Polling

Figure 4 compares BPC performance of DQ and CH on the AMD Opteron SMP. We observe a striking difference between the schedulers. Except for small tasks, DQ has no problems balancing this workload. CH, on the other hand, plateaus around a speedup of 8, regardless of task size. Tasks are equally distributed among workers, yet account for only 28% of the workers' execution time in Fig. 4(c). The rest is spent exchanging steal requests and waiting for tasks.

In a sense, BPC is the perfect adversary to CH because BPC is designed to stress the ability to quickly locate new work [13]. Producer tasks bounce back and forth among workers so that there is no stable producer of tasks that could be



(a) Task size 10μs (b) Task size 100μs (c) Task size 1000μs

Fig. 4. BPC benchmark performance results of deque-based and channel-based work-stealing schedulers on the 4-way AMD Opteron SMP.

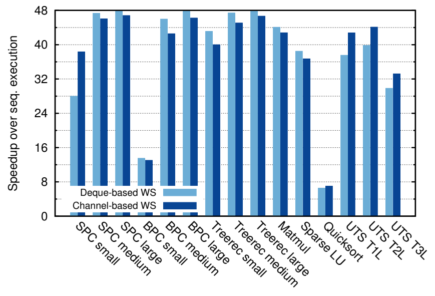
targeted reliably. Rather, performance is dependent on short message-handling delays. Long-running tasks must be interruptible or otherwise steal requests start to pile up, effectively blocking workers from making further progress. Figure 4 includes the results of a version that uses software polling to periodically yield control to the runtime system and check for steal requests. While not as robust as generating interrupts on message receive, polling succeeds as a low-overhead mechanism for improving scheduler performance. The remaining overhead due to channel communication in Fig. 4(c) is reduced to under 2%.

3.3 Summary

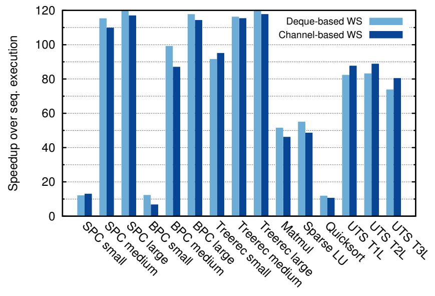
Figure 5 summarizes all benchmark results on our test systems. Matmul and Sparse LU are characterized by parallel phases separated by barrier synchronization. Sparse LU proves to be a challenging workload for CH because parallelism is decreasing over the course of the computation so that communication delays become visible. In later stages of the matrix decomposition, there are simply not enough tasks left to keep every worker busy, with the consequence that some steal requests circulate indefinitely. Failed steals are actually a bigger problem in CH than in DQ because idle workers interrupt busy workers by sending messages that must be handled, even though many messages may simply end up being passed on to other workers. We have found that a simple backoff strategy in fact improves performance in the case of Sparse LU: when stealing fails repeatedly, idle workers back off and resend steal requests at a later time. Termination detection is not affected by the backoff as long as only idle workers are allowed to hold back and resend steal requests.

In general, CH performs well on divide-and-conquer-type applications in which tasks are created recursively. Such applications tend to generate large numbers of tasks, and even if the load is highly unbalanced, finding a worker that can fulfill a steal request is less of a problem than in applications with flat parallelism. CH even outperforms DQ on all tested UTS trees. UTS generates a large number of very small tasks and, like SPC, benefits from steal-half. Again, we notice the efficiency of chunking tasks in CH.

BPC, perhaps the most challenging workload in our setting, makes it clear that performance of CH is sensitive to message-handling delays. In our tests,



(a) AMD Opteron SMP



(b) Intel Xeon Phi

Fig. 5. Summary of 48-thread and 120-thread speedups on a 4-way AMD Opteron SMP and Intel Xeon Phi compared to sequential execution. On the Xeon Phi, we made sure that every core was running two threads.

we used polling to check for and service incoming steal requests. Generating interrupts on message receive and jumping to the handling code in the runtime would be a more robust solution, but unless it can be done very efficiently, polling likely incurs less overhead, especially in unbalanced applications that generate a constant stream of messages. The decision whether to use polling or interrupts involves a trade-off one way or the other. In a perfect world, we would like to see this problem addressed at the hardware level [32].

4 Related Work

Since the rise of multicore processors, now almost a decade ago, task parallelism has taken center stage in the design of new libraries [29, 31, 23] and languages [4, 10, 9]. Cilk [16] has pioneered many runtime techniques that are of importance today, first and foremost the scheduling of tasks by work stealing [8]. Work stealing has become the algorithm of choice on shared-memory multiprocessors. There is an increasing body of work on work stealing with concurrent deques [6, 11, 25] (only to name a few), but work stealing is not confined to shared memory. Dinan et al. describe work-sharing and work-stealing implementations of the UTS benchmark using MPI [12]. In later papers, Dinan et al. show the scalability of work stealing in the context of PGAS [13]. Runtime systems that build on one-sided or two-sided communication libraries can also be found in [28] and, more recently, in [26]. Ravichandran et al. combine conventional work stealing with message passing at the node level to target clusters of multicore processors [30]. Saraswat et al. introduce a load balancing extension to reduce the performance penalty of work stealing on large-scale machines [33]. Sanchez et al. propose hardware support for exchanging asynchronous messages between threads in order to build efficient task schedulers for future manycore chips [32].

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 [5]. The paper proposes a receiver-initiated algorithm, based on steal requests, and a sender-initiated algorithm, similar to work dealing [18]. In both algorithms, worker threads communicate through a set of shared variables, relying on atomic operations to distribute tasks. In our scheduler, worker threads communicate exclusively over channels, to the extent that porting the scheduler to a new platform becomes a matter of writing a channel implementation for it.

5 Conclusion

In this paper, we have described a new work-stealing scheduler in which worker threads coordinate scheduling and transfer tasks through channels. Channels are a simple message passing abstraction, valuable for increasing the portability of the scheduler. Our benchmarks have shown that there is no significant abstraction penalty due to channel communication on two recent shared-memory machines: in most cases, the channel-based scheduler performs within 3-7% of a deque-based scheduler. This conclusion is important because an inefficient scheduler would defeat the purpose of task-parallel programming. Efficient message-passing schedulers will make it easier to target future manycores as well as distributed environments.

Acknowledgment

This work is supported by the Deutsche Forschungsgemeinschaft (DFG).

References

1. The Go Programming Language. `http://golang.org`
2. The Rust Programming Language. `http://rust-lang.org`
3. OpenMP Application Program Interface Version 3.1. `http://www.openmp.org/mp-documents/OpenMP3.1.pdf` (July 2011)
4. A Brief Overview of Chapel. `http://chapel.cray.com/papers/BriefOverviewChapel.pdf` (January 2013)
5. Acar, U.A., Charguéraud, A., Rainey, M.: Scheduling Parallel Programs by Work Stealing with Private Deques. pp. 219–228. PPoPP '13 (2013)
6. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread Scheduling for Multiprogrammed Multiprocessors. pp. 119–129. SPAA '98 (1998)
7. Ayguadé, E., et al.: The Design of OpenMP Tasks. IEEE Trans. Parallel Distrib. Syst. 20, 404–418 (March 2009)
8. Blumofe, R.D., Leiserson, C.E.: Scheduling Multithreaded Computations by Work Stealing. pp. 356–368. FOCS '94 (1994)
9. Cavé, V., Zhao, J., Shirako, J., Sarkar, V.: Habanero-Java: the New Adventures of Old X10. pp. 51–61. PPPJ '11 (2011)
10. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. pp. 519–538. OOPSLA '05 (2005)

11. Chase, D., Lev, Y.: Dynamic Circular Work-Stealing Deque. pp. 21–28. SPAA '05 (2005)
12. Dinan, J., Olivier, S., Sabin, G., Prins, J., Sadayappan, P., Tseng, C.W.: Dynamic Load Balancing of Unbalanced Computations Using Message Passing. pp. 1–8. IPDPS '07 (2007)
13. Dinan, J., Larkins, D.B., Sadayappan, P., Krishnamoorthy, S., Nieplocha, J.: Scalable Work Stealing. pp. 53:1–53:11. SC '09 (2009)
14. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. pp. 124–131. ICPP '09 (2009)
15. Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., Burger, D.: Dark Silicon and the End of Multicore Scaling. pp. 365–376. ISCA '11 (2011)
16. Frigo, M., Leiserson, C.E., Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language. pp. 212–223. PLDI '98 (1998)
17. Hameed, R., Qadeer, W., Wachs, M., Azizi, O., Solomatnikov, A., Lee, B.C., Richardson, S., Kozyrakis, C., Horowitz, M.: Understanding Sources of Inefficiency in General-Purpose Chips. pp. 37–47. ISCA '10 (2010)
18. Hendler, D., Shavit, N.: Work Dealing. pp. 164–172. SPAA '02 (2002)
19. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2008)
20. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1985)
21. Kumar, R., Mattson, T., Pokam, G., Wijngaart, R.: The case for message passing on many-core chips. In: Hübner, M., Becker, J. (eds.) Multiprocessor System-on-Chip, pp. 115–123. Springer New York (2011)
22. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Commun. ACM 21(7), 558–565 (Jul 1978)
23. Leijen, D., Schulte, W., Burckhardt, S.: The Design of a Task Parallel Library. pp. 227–242. OOPSLA '09 (2009)
24. Mattson, T.G., et al.: The 48-core SCC Processor: the Programmer's View. pp. 1–11. SC '10 (2010)
25. Michael, M.M., Vechev, M.T., Saraswat, V.A.: Idempotent Work Stealing. pp. 45–54. PPoPP '09 (2009)
26. Min, S.J., Iancu, C., Yelick, K.: Hierarchical Work Stealing on Manycore Clusters. PGAS '11 (2011)
27. Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P., Tseng, C.W.: UTS: An Unbalanced Tree Search Benchmark. pp. 235–250. LCPC '06 (2007)
28. Olivier, S., Prins, J.: Scalable Dynamic Load Balancing Using UPC. pp. 123–131. ICPP '08 (2008)
29. Peierls, T., Goetz, B., Bloch, J., Bowbeer, J., Lea, D., Holmes, D.: Java Concurrency in Practice. Addison-Wesley Professional (2005)
30. Ravichandran, K., Lee, S., Pande, S.: Work Stealing for Multi-core HPC Clusters. pp. 205–217. Euro-Par'11 (2011)
31. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media (2007)
32. Sanchez, D., Yoo, R.M., Kozyrakis, C.: Flexible Architectural Support for Fine-Grain Scheduling. pp. 311–322. ASPLOS '10 (2010)
33. Saraswat, V.A., Kambadur, P., Kodali, S., Grove, D., Krishnamoorthy, S.: Lifeline-based Global Load Balancing. pp. 201–212. PPoPP '11 (2011)