

Evaluating the run-time performance of Kahn process network implementation techniques on shared-memory multiprocessors*

Željko Vrba, Pål Halvorsen, Carsten Griwodz
Simula Research Laboratory and University of Oslo, Norway
{zvrba,paalh,griff}@ifi.uio.no

Abstract

Software development tools have not adapted to the growing popularity of multi-core CPUs, and developers are still “stuck” with low-level and high-cost thread abstractions. The situation is becoming even more complicated with the advent of heterogenous computing. In this article, we point out some drawbacks of high-level abstractions currently in use, and propose Kahn process networks (KPN) as a more high-level and efficient abstraction for developing parallel applications. We show that the native POSIX mechanisms (threads and message queues) perform suboptimally as an implementation vehicle for KPNs, and we present an implementation of a run-time environment that can execute KPNs with less overhead. Our evaluation shows the advantages and disadvantages of statically mapping Kahn processes to CPUs.

1 Introduction and related work

As CPU manufacturers have hit the frequency wall, they are adding performance along another dimension by increasing the number of computing cores on a single chip. This change, although superficially simple, has far-reaching consequences for application development: the developers can no longer expect that their code will automatically run faster on newer generations of CPUs. Instead, applications have to be designed from the ground-up with parallelism in mind, which is rather hard today because the abstractions in wide-spread use (threads, locks and condition variables) are *non-deterministic* and too low-level. Non-determinism makes problems very hard to diagnose, reproduce and fix [8].

Several abstractions, some of which, unlike threads, do not presuppose a shared-memory model, have been proposed in order to ease development of parallel applica-

tions and remedy this situation. Futures [6], also called promises, are an abstraction of asynchronous function calls. A future immediately returns an encapsulation of the result (“promise”) and continues to execute asynchronously; evaluation of the promise synchronizes with the completion of the future. Polyphonic C# [1] extends the C# programming language with asynchronous methods and chords, concepts based on formalisms of the join calculus. A chord is a synchronization device whose header specifies a pattern of asynchronous methods, and whose body is executed after all methods of the header have been completed. If several chords are eligible for execution, one is chosen *non-deterministically*; as a consequence, even a *single-threaded* program can behave non-deterministically. Both need to be used carefully to avoid deadlocks and race conditions. With software transactional memory (STM) [5], operations over shared data protected by locks are replaced with `atomic{}` code-blocks, where it is guaranteed that either all memory operations in the atomic block succeed or fail, i.e., the operations are roll-backed in case of conflict (hence the name transaction). The compiler replaces memory references (both reads and writes) inside atomic blocks with calls to the STM run-time system, which resolves conflicts. Finally, there are libraries, such as Intel Threading Building Blocks (TBB), which encapsulate parallelized versions of common algorithms into a library usable by application developers.

Parallel programming is today further complicated by the steadily increasing significance of heterogenous computing. A significant factor that should be explicit in the abstractions is *communication*, of which *data sharing* is just a special case, because it is very costly when parts of a program reside on different CPUs (e.g., data transfer between main memory and graphics card). We want an abstraction that is *deterministic*, does not presuppose a shared memory model, and allows developers to easily express both parallelism and communication; none of the above-mentioned proposals satisfies all these criteria.

Therefore, we turn towards the simple, yet powerful, abstraction of Kahn process networks (KPNs) [7] which have strong formal foundations and unify expressive power, de-

*This work has been financed by the INSTANCE II project (Norwegian Research Council’s Contract No. 147426/431) and Department of Informatics, University of Oslo.

terministic behavior, and asynchronous communication and execution. Furthermore, their graph representation makes it easy to reason about communication patterns and to employ advanced scheduling and load-balancing algorithms. All of these properties should contribute to easier application development, debugging and, possibly, better run-time performance because schedulers have more information.

Since processes in a KPN execute asynchronously, parallelism with fine granularity (more processes) may lead to more parallelism at run-time, provided there are enough CPUs. Furthermore, fine granularity is desirable because otherwise the deterministic nature of KPNs may result in much blocking. However, fine granularity also presupposes an efficient run-time environment that can support many processes with low overhead, even on few CPUs.

In this paper, we focus on the performance that this paradigm can achieve on shared-memory multiprocessors, since these systems have the highest importance on today's computing market. We show that the native POSIX mechanisms (threads and message queues), when used in the obvious way, are not well suited for implementing a run-time environment for executing KPNs. We have therefore implemented an optimized KPN run-time environment which is shown by benchmarks to have significant performance improvements (~ 2 times faster message passing, up to ~ 6.5 times faster scheduling) compared to the native mechanisms.

2 Kahn process networks

A KPN [7] consists of *processes* and *channels*, and it has a simple representation as a *directed graph* (see figure 2 for examples). A process encapsulates data and a single, sequential control flow, independent of any other process. Processes are not allowed to share data and may communicate only by sending messages over channels. Channels are *infinite* FIFO queues that store discrete *messages*. Channels have *exactly one* sender and one receiver process on each end (1:1), and every process can have multiple *input* and *output* channels. Sending a message to the channel always succeeds, but trying to receive a message from an empty channel *blocks* the process until a message becomes available. It is not allowed to poll a channel for presence of data. These properties fully define the *operational semantics* of KPNs and make the Kahn model *deterministic*, i.e., the history of messages produced on the channels does not depend on the order in which the processes are executed, provided the scheduling is *fair*.

The theoretical model of KPN described so far is idealized in two ways: 1) it places few constraints on process behavior, and 2) it assumes that channels have infinite capacities. The first assumption makes it possible to construct a KPN that *under no circumstances* can be executed in finite

space. We shall therefore consider only *effective* networks, i.e., networks where each sent message is eventually read from the channel.

The second assumption presents an obvious difficulty because any actual KPN implementation runs in finite memory. A common (partial) solution is to assign *capacities* to channels and redefine the semantics of send to *block* the sending process if the delivery would cause the channel to exceed its capacity. Under such send semantics, an *artificial deadlock* may occur, i.e., a situation where a cyclically dependent subset of processes is blocked, but which would continue running in the theoretical model. The algorithm of Geilen and Basten [4] resolves the deadlock by traversing the cycle to find the channel of least capacity and enlarging it by one message, thus resolving the deadlock.

3 KPN implementation

To the best of our knowledge, there exist only two other general-purpose KPN runtime implementations: YAPI [3] and Ptolemy II [2]. YAPI is not a pure KPN implementation, as it extends the semantics and thus introduces the possibility of non-determinism, its code-base is too large for easy experimentation (120 kB vs. 23 kB in our implementation), and we were unable to make it use multiple CPUs. Ptolemy II is implemented in Java, which makes it difficult to experiment with low-level mechanisms such as context-switching and scheduling.

The easiest way to implement a KPN run-time is to use existing OS facilities (see table 1), but, as shown in this paper, such approach suffers from high overheads. We have therefore implemented a KPN execution environment in C++, with small portions specific to the Solaris OS and the AMD64 architecture.¹

When the KPN is started, m runner threads are created and scheduled by the OS onto n available CPUs (see figure 1). A runner thread ("runner") schedules KPs from a private run-queue by a fair algorithm;² KPs are assigned and pinned to runners at KPN creation time. The scheduling on each CPU is *cooperative* with yield points being message send/receive operations. Context switching is implemented in assembly. When there are no runnable KPs, the runner sleeps on a private counting semaphore until woken up by a KP belonging to another runner.

Channels are implemented in two layers: an upper layer which handles interaction with KPs and the scheduler, and a lower layer where the actual transport mechanism is implemented. When a KP wants to block, it cannot use standard OS mechanisms because doing so would block the whole

¹The code is available at <http://simula.no/research/networks/software>

²The current prototype uses the round-robin policy for its simplicity, but this is easily changed.

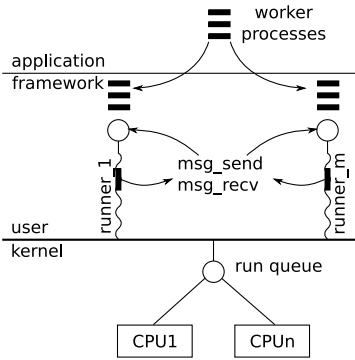


Figure 1. Two-level KPN scheduling

runner thread and prevent dispatching of another KP. Instead, our scheduler provides a sleep/wake-up synchronization object that is optimized for 1:1 communication.

Since KPs are executing in a shared address space, it is still possible that they modify each other’s state and thus ruin the Kahn semantics. To minimize the possibility of this happening, the message class contains only a pointer to message contents in shared memory and implements *move semantics*: send operation nullifies the pointer, making message contents inaccessible to the sender. This allows for a zero-copy message transport where a channel is a linked list of messages, and links are embedded in the message body. The trade-off is that message creation, destruction and assignment require dynamic memory (de)allocation which we optimize by using Solaris’s *libumem* allocator.

We have not implemented any special support for I/O operations yet because our current research interests lie in adaptive scheduling and load-balancing of KPNs. Using the usual blocking UNIX I/O calls in KPs can block the runner which will then be unable to dispatch another KP. One possible solution to this problem is to have separate runners for KPs doing I/O; another is to use asynchronous I/O API. We use the first approach, since it is the simplest.

4 Benchmarks

We have implemented several benchmarks (see figure 2) to compare the efficiency of scheduling and message transport mechanisms shown in table 1. The bottleneck in the “ring” benchmark is message-passing and scheduling, while the bottleneck in the advanced encryption standard (AES) benchmark is CPU bandwidth. The third benchmark simulates an execution of an H.264 encoder based on data obtained by profiling a real encoder.³

The benchmarks are compiled with Sun’s C++ compiler with maximum optimizations in 64-bit mode and executed

³x264; available at <http://www.videolan.org/developers/x264.html>

Code	Parameter
P/U	Pthread scheduler (P) is the native scheduler with dynamic load-balancing over available CPUs and 1 thread / KP. Our scheduler (U), so far, only supports static assignment of KPs to CPUs.
1/2/n	Number of runner threads for our scheduler; n for the pthread scheduler.
O/N	Our optimized transport; zero-copy (see section 3) / Native transport (POSIX message queues; copies message contents).

Table 1. Benchmark parameters.

on an otherwise idle 2.2 GHz dual-core AMD Athlon with 2 GB of RAM running Solaris 10. Each data point is an average of 10 consecutive runs and reports the elapsed real (wall-clock) time⁴ measured by a high-resolution timer (`gethrtime` function). To reflect the true cost of communication, measurements also include the overhead of dynamic memory (de)allocations (see section 3). The graphs have min/max error bars, but the variability of the measurements was so small that they are barely visible.

4.1 Ring: context-switch and message-passing performance

Since KPNs rely heavily on message passing, we measure the duration of the composite transaction [send → context switch → receive]. We employ the same method as the *lat_ctx* benchmark from the *lmbench* suite.⁵ KPs $P_0 \dots P_{n-1}$, are created and connected into a ring topology (see figure 2(a)). P_0 sends an initial message with payload of 4 bytes (so that copying overhead for native transport is negligible) and measures the time it takes to make m round-trips.

The number of KPs n is varied from 10 to 1000 in steps of 50. The number of round-trips m is adjusted so that the total number of transactions is approximately 10^6 , i.e., $m = \lfloor 10^6/n \rfloor$, which gives a sufficiently long execution time and number of transactions to cancel out the the noise of scheduling effects such as interrupt processing. This benchmark is also sensitive to KP placement: with good placement (G), inter-CPU synchronization every $\lfloor n/k \rfloor$ transactions, k being the number of CPUs. With bad placement (B), synchronization on every transaction. Benchmark results for three different configurations (cf. table 1) are shown in figure 3.

⁴The wall-clock time metric is most representative because it accurately reflects the real time needed for task completion, which is what the end-users are most interested in. We have also measured system and user times (`getrusage`), but do not use it to present our results because 1) it does not reflect the reduced running time with multiple CPUs, and 2) resource usage does not take into account sleep time.

⁵<http://www.bitmover.com/lmbench/>

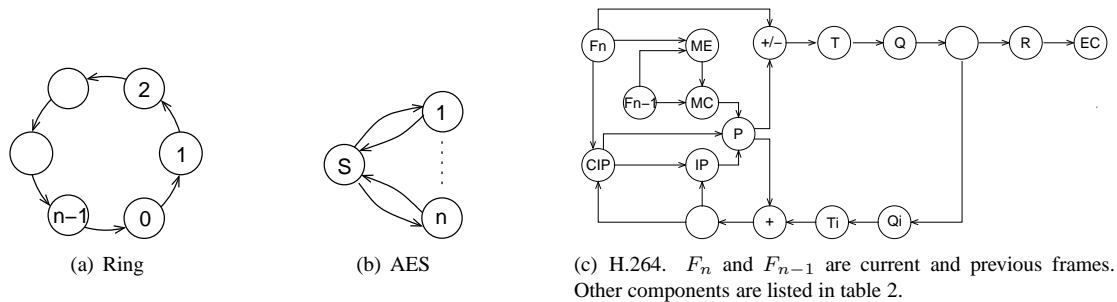


Figure 2. KPNs for the “ring”, “AES” and H.264 benchmarks.

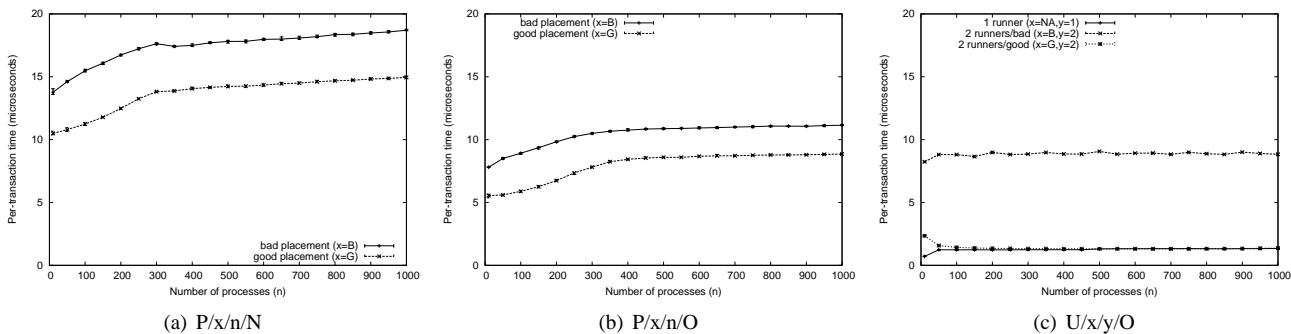


Figure 3. Performance of the “ring” test in different configurations. x and y are parameter placeholders whose values are given in figure legends (see also table 1 for explanation of parameters).

We can draw several conclusions from these tests. First, inter-CPU synchronization is expensive, which is confirmed by observing the difference in running times of the benchmark with good and bad KP placement. We believe that it is also the cause for the anomaly of the U/G/2/O configuration in figure 3(c), where it can be seen that the running time *increases* as the number of KPs *decreases* below 100.

Second, the test shows that our optimized transport is ~ 1.7 – 1.9 times faster than the native transport. The exact speedup is dependent on the workload, KP placement and message size. The native transport has constant performance for messages up to 128 bytes, after which it degrades proportionally with the message size, due to its copying semantics (not shown in the graphs).

Third, our user scheduler performs best with a single runner because lock contention never happens. In the U/B/2/O configuration, it performs slightly worse than the pthread scheduler for less than ~ 100 KPs. As the number of KPs grows, it starts to outperform the pthread scheduler, whose performance becomes worse while ours remains constant, and it is 1.25 times faster for $n = 1000$ KPs. With good KP placement (U/G/2/O configuration) and $n = 1000$, our scheduler is 6.5 times faster than the pthread scheduler (P/G/n/O configuration), and achieves nearly the same per-

formance as with single runner.

4.2 AES: scheduling overheads

Application overdecomposition (having more workers than CPUs) is important because communication latencies and blocking times may overlap with computation, and because performance will transparently improve as more processors are added. One of the goals of our run-time is to enable overdecomposition without incurring large performance penalty on compute-intensive tasks, such as 128-bit AES encryption.

The source KP S hands out equally-sized memory chunks to n worker KPs which reply back when they have encrypted the given chunk (see figure 2(b)). The exchanged messages are carrying only pointer-length pairs (16 bytes in total). We fix the total block size to 2^{28} bytes (256 MB, to avoid possible swapping), fix the number of passes through each to 8 (to achieve sufficiently long running time), and vary the base-2 logarithm of memory chunk size, k , from 14 to 28. A worker KP is created for every chunk of memory, so there are $n = 2^{28-k}$ KP workers in total. In the U/2/O configuration, KPs are assigned to CPUs in a round-robin manner to evenly distribute the load among CPUs.

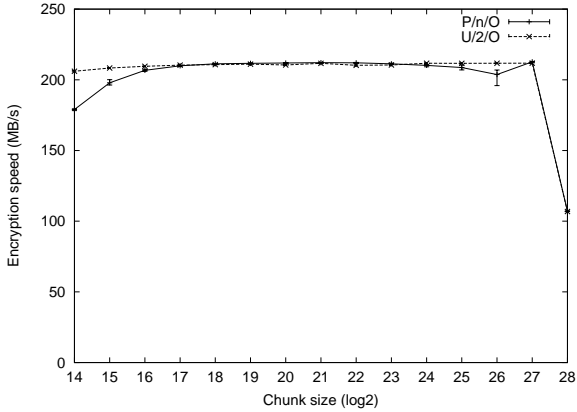


Figure 4. Performance of AES encryption.

Figure 4 shows the encryption throughput versus the number of workers; note that the number of workers increases to the left, as the chunk size decreases. The 50% drop in speed at $n = 28$ occurs because the task is serially executed on a single CPU.

As the number of KPs increases beyond 2048 (chunk size less than 2^{17}), the encryption speed decreases, but much more sharply for the pthread scheduler. By running *vmstat* in parallel, we have confirmed that this was not caused by swapping. Furthermore, when using the pthread scheduler with 16384 worker KPs, there are approximately 25000 context-switches, 35000 system calls and 30000 instruction TLB misses per second. In contrast, our scheduler with 16384 worker KPs has < 500 system calls and context switches per second (which almost coincides with the numbers on an idle system), and < 2000 instruction TLB misses per second (< 1000 on an idle system).

While we have not been able to find the exact causes of this behavior, we believe, based on real-time monitoring by advanced tools such as *dtrace* and *truss*, that they stem from the combination of the (preemptive) pthread scheduler and lock contention. Whatever the root cause, it is obvious that the pthread scheduler has some trouble as the number of threads increases beyond 2048. These results indicate that our scheduler is better suited for executing overdecomposed problems than the native OS scheduler.

4.3 H.264 encoder

Our KPN representation of an H.264 video encoder (see figure 2(c)) is only a slight adaptation of the encoder block diagram found in [9]. Each functional unit is implemented as a KP, with additional KPs inserted at the branch points of the original diagram. Since the P, MC and ME stages are together using over 50% of the CPU, each of these has been parallelized (not shown in the figure) in the same way as AES.

Stage	Time %
Choose inter-/intra-prediction (CIP)	8.08
Intra-analysis (IP)	5.02
Inter-analysis (MC+ME)	34.36
Prediction (P)	23.83
Macroblock encoding (T,Ti,Q,Qi)	8.69
Entropy encoding (EC)	6.72
Other, including reordering (R)	13.3

Table 2. Cachegrind profiling data of x264.

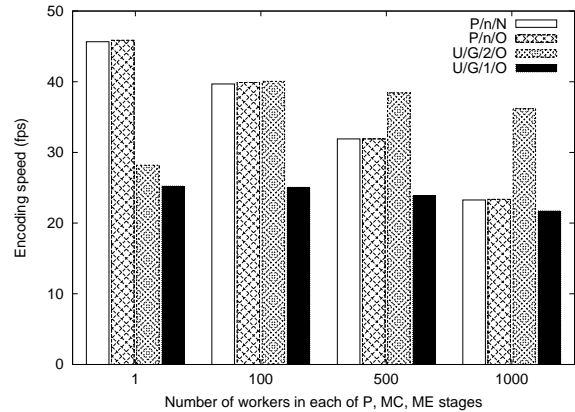


Figure 5. Performance of the H264 encoder.

For simplicity, encoding operations are replaced with loops that consume a percentage of CPU time corresponding to that used by the x264 codec at respective stages. The amount of time spent in each x264 function has been profiled with the cachegrind tool, and mapping of these results to the block diagram is shown in table 2. The loops have been adjusted so that the “frames” are “encoded” at approximately 25 fps in real-time. We measure the time needed to “encode” 300 frames (12 seconds) in different configurations. For each configuration, the benchmark is executed with 1, 100, 500 and 1000 workers in each of the P, MC and ME stages (up to 3000 additional workers in total).

We draw the following conclusions from the results in figure 5: 1) data dependencies in KPN prohibit linear scalability with the number of cores; 2) message transport (P/N vs. P/O) has no influence on running time because the workload is CPU-bound and few messages are exchanged; 3) with few KPs and little available parallelism, the native scheduler’s dynamic load-balancing outperforms our static assignment of processes; and 4) when the application is overdecomposed and KPs are carefully placed, our scheduler performs as well as, or better than the native.

5 Conclusion and future work

In this paper, we have motivated the use of the KPN formalism as an alternative to other abstractions intended to simplify development of parallel applications. Compared to futures, transactional memory and joins, the KPN framework unifies determinism and explicit modeling of communication. It does not rely on shared memory, so it is also suitable for designing distributed systems. We have described our implementation, whose cornerstones are an optimized zero-copy message transport and a two-level scheduler. Benchmarking on the Solaris 10 OS shows that our implementation makes significant scalability and performance improvements over native mechanisms (POSIX message queues and kernel thread scheduler): in our tests, message transport is 1.7–1.9 times faster, and the scheduler is up to 6.5 times faster.

The “AES” and “H.264” benchmarks show that our scheduler scales much better than the pthread scheduler in situations with a large number of CPU-intensive processes (see figures 4 and 5). This is best illustrated on the H.264 benchmark with 1 and 1000 workers in each of the P, MC and ME stages: our scheduler suffers a performance drop of $\sim 10\%$, while the performance of the pthread scheduler is nearly halved. Experiment 3 shows also that static assignment of KPs to CPUs can lead to blocking that cancels out the benefits of more efficient scheduler.

We are currently working on extending our scheduler with dynamic load-balancing strategies that take CPU requirements of processes as well as their communication patterns into account. Scheduling according to communication intensity is significant because some of today’s multi-core systems have non-uniform memory architecture, where it is desirable to place heavily communicating processes on nearby CPUs. Such scheduling would be even harder to design and implement for abstractions that are less structured than KPNs (e.g., transactional memory).

References

- [1] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for c#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.
- [2] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii). Technical Report UCB/EECS-2008-28, EECS Department, University of California, Berkeley, Apr 2008.
- [3] E. de Kock, G. Essink, W. J. M. Smits, R. van der Wolf, J.-Y. Brunei, W. Kruijtzter, P. Lieverse, and K. K.A. Vissers. Yapi: application modeling for signal processing systems. *Design Automation Conference, 2000. Proceedings 2000. 37th*, pages 402–405, 2000.
- [4] M. Geilen and T. Basten. Requirements on the execution of kahn process networks. In *Programming Languages and Systems: 12th European Symposium on Programming, ESOP 2003*, pages 319–334. Springer Berlin/Heidelberg, 2003.
- [5] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP ’05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.
- [6] J. Henry G. Baker and C. Hewitt. The incremental garbage collection of processes. Technical report, Cambridge, MA, USA, 1977.
- [7] G. Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 74, 1974.
- [8] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.
- [9] I. E. G. Richardson. H.264/mpeg-4 part 10 white paper, 2002.