2009 Sixth IFIP International Conference on Network and Parallel Computing

The *Nornir* run-time system for parallel programs using Kahn process networks

Željko Vrba, Pål Halvorsen, Carsten Griwodz, Paul Beskow
Simula Research Laboratory, Oslo, Norway
Department of Informatics, University of Oslo, Norway
{*zvrba,paalh,griff,paulbb*}*@ifi.uio.no*

Dag Johansen
Dept. of Computer Science, University of Tromsø, Norway
Fast Search and Transfer, Norway
{*dag@cs.uit.no*}

***Abstract*—Shared-memory concurrency is the prevalent paradigm used for developing parallel applications targeted towards small- and middle-sized machines, but experience has shown that it is hard to use. This is largely caused by synchronization primitives which are low-level, inherently nondeterministic, and, consequently, non-intuitive to use. In this paper, we present the *Nornir* run-time system. Nornir is comparable to well-known frameworks like MapReduce and Dryad, but has additional support for process structures containing cycles. It is based on the formalism of Kahn process networks, which we deem as a simple and deterministic alternative to shared-memory concurrency. Experiments with real and synthetic benchmarks on up to 8 CPUs show that performance in most cases improves almost linearly with the number of CPUs, when not limited by data dependencies.**

I. Introduction

It is widely recognized that developing parallel and distributed programs is inherently more difficult than developing sequential programs. As a consequence, several frameworks that aim to make such development easier have emerged, such as Google's MapReduce [1], Yahoo's Pig latin [2] which uses Hadoop[1] as the back-end, Phoenix [3], and Microsoft's Dryad [4]. All of these frameworks are gaining in popularity, but they lack a feature that is critical to our application domains: the ability to model iterative algorithms, i.e., algorithms containing feedback loops in their data-path.

Much of our research focuses on the execution of complex parallel programs, such as real-time encoding of 3-D video streams and data encryption, where cycles are more rule than the exception (see figure 2). Thus, we cannot use any of the existing frameworks, so we have turned towards the flexible formalism of Kahn process networks (KPN) [5]. KPNs retain the nice properties of MapReduce and Dryad, but in addition support cycles. Even though KPNs are an inherently distributed model of computation, their implementation for shared-memory machines and its performance is worth studying for many reasons (see section II), the main ones being determinism and, consequently, composability. Determinism guarantees that a program, given the same input, will behave identically on each run. This significantly eases debugging, which is an otherwise a notoriously hard problem with parallel and distributed computations. Composability guarantees that assembling together independently developed components will yield the expected result.

In our earlier paper [6], we have evaluated implementation options for KPNs on shared-memory architectures. In a follow-up paper [7], we have presented case studies of modeling with KPNs and their comparison with MapReduce. We showed that KPNs allow more natural problem modeling than MapReduce, and that implementations of real-world tasks on top of Nornir outperform the corresponding MapReduce implementations. In this paper, we give implementation details about the new version of Nornir, which is internally significantly different from the one described in [6]; this same implementation is also used for evaluation in [7]. We also investigate performance and scalability characteristics of Nornir using a set of benchmarks on a workstation-class machine with 8 cores. Our performance experiments reveal some weaknesses in our current implementation, but nevertheless indicate that KPNs are a viable programming model for parallel applications on shared-memory architectures.

II. Kahn process networks

KPNs, MapReduce and Dryad have in common two important features, both of which significantly simplify development of parallel applications: 1) communication and parallelism are explicitly expressed in the application graph; 2) individual processes are written in the usual sequential manner, and do not have access to each other's state. In addition, KPNs have a unique combination of other desireable properties:

- *Determinism.* KPNs are deterministic, i.e., each execution of a network produces the same output given the same input,[2] regardless of scheduling strategy.
- *Reproducible faults.* One consequence of determinism is that faults are consistently reproducible, which is otherwise a notoriously difficult problem with parallel and distributed systems. Reproducibility of faults greatly eases debugging.
- *Composability.* Another consequence of determinism is that processes can be composed: connecting the

[1] An open-source MapReduce implementation in Java, available at http://hadoop.apache.org/

[2] Provided that processes themselves are deterministic.

978-0-7695-3837-2/09 $26.00 © 2009 IEEE
DOI 10.1109/NPC.2009.19

IEEE computer society

output of a process computing $f(x)$ to the input of a process computing $g(x)$ is guaranteed to compute $g(f(x))$. Therefore, processes can be developed and tested individually and later put together to perform more complex tasks.
- *Deterministic synchronization.* Synchronization is embodied in the blocking receive operation. Thus, developers need not use other, low-level and non-deterministic synchronization mechanisms such as mutexes and condition variables.[3]
- *Arbitrary communication graphs.* Whereas MapReduce and Dryad restrict developers to a parallel pipeline structure and directed acyclic graphs (DAGs), KPNs allow *cycles* in the graphs. Because of this, they can directly model iterative algorithms. With MapReduce and Dryad this is only possible by manual iteration, which incurs high setup costs before each iteration [3].
- *No prescribed programming model.* Unlike MapReduce, KPNs do not require that the problem be modeled in terms of processing over key-value pairs. Consequently, transforming a sequential algorithm into a Kahn process often requires minimal modifications to the code, consisting mostly of inserting communication statements at appropriate places.

A KPN [5] has a simple representation in the form of a *directed graph* with *processes* as nodes and *channels* as edges, as exemplified by figure 1. 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 communication), 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. These properties 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 how the process execution order. Less restrictive models, e.g., those that allow non-blocking reads or polling, would result in non-deterministic behavior.

The theoretical model of KPNs 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. These assumptions are somewhat problematic because they allow for the construction of KPNs that need unbounded space for their execution. However, any real implementation is constrained to run in finite memory. A

[3]Many inexperienced developers expect that mutexes and CVs wake up waiting threads in FIFO order, whereas the wake-up order is in reality non-deterministic.

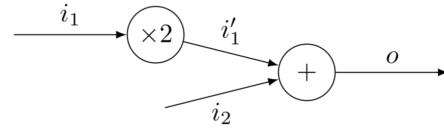


Figure 1. An example KPN. i_1 and i_2 are external input channels to the network (assumed to be numbers), o is the external output channel, and i'_1 is an internal channel. The inputs and the output are related by the formula $o = 2i_1 + i_2$

common (partial) solution to this 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 blocks on send, which would continue running in the theoretical model. The algorithm of Geilen and Basten [8] resolves the deadlock by traversing the cycle to find the channel of least capacity and enlarging it by one message, thus resolving the deadlock.

It also is worth noting that KPNs are not a universal solution for what is an inherently difficult problem of developing parallel and distributed applications. Even though determinism is a desirable property from the standpoint of program design and debugging, it limits the applications for KPNs. A disk scheduler serving many clients is a very simple example of a use-case that is inappropriate for KPNs. It must periodically serve all clients in some order to preserve fairness, say round-robin, but since read is blocking, absence of requests from one client can indefinitely postpone serving of requests from other clients. Such use-cases mandate use of other frameworks, or extending the KPN formalism by non-deterministic construct(s) such as $m : n$ channels and/or polling.

III. NORNIR

The Nornir run-time system is implemented in C++, and it runs on Windows and POSIX operating systems (Solaris, Linux, etc.). The implementation[4] consists of a Kahn process (KP) scheduler, message transport and deadlock detection and resolution algorithms.

A. Process scheduler

Since KPNs are deterministic, they give a great freedom in implementing the process scheduler: any *fair* scheduler will result in a KPN execution that generates the full output.[5] In this context, fairness means that the execution of a ready process will not be indefinitely postponed.

KPN networks could be built on top of native OS mechanisms, with each KP being an OS-thread. Channels would be protected by blocking mutexes, and condition variables

[4]Code is available at: http://simula.no/research/networks/software

[5]If the scheduler is not fair, the output will be correct, but possibly shorter than it would be under a fair scheduler.

would be used as the sleep / wakeup mechanism. However, we have investigated this approach in an earlier paper [6] and found that user-mode scheduling of many KPs over few kernel threads is considerably more efficient.

Nornir can be configured to use different scheduling policies. In addition to classical work-stealing [9], we have also implemented a policy based on graph-partitioning [10] which tries to reduce the amount of inter-CPU synchronization and communication. Here, we describe only the work-stealing policy because experiments on the same workloads as used in section V have shown that it leads to much better application performance.

When the KPN is started, m runner threads ("runners") are created and scheduled by the OS onto the available CPUs. Each runner implements a work-stealing policy, which our experiments have shown to have best performance for computationally intensive tasks on few CPUs. With work stealing, each runner has a private run queue of ready KPs. If this queue is empty, it tries to steal a KP from a randomly chosen runner. For simplicity, we do not use the non-blocking queue described in [9]; instead we use an ordinary mutex per run queue. This might become problematic on machines with many cores, but we deemed that introducing the additional complexity of a non-blocking queue was unnecessary at this stage of our research.

The work-stealing scheduler uses a user-mode context-switch. On Solaris and Linux running on AMD64 architecture, we employ optimized, hand-crafted assembly code for context-switch; on other platforms we use OS-provided facilities: fibers on windows, and `swapcontext()` on POSIX. The latter is inefficient because each context switch requires a system call to save and restore the signal mask.

B. Message transport

In KPNs, channels have a two-fold role: 1) to interact with the scheduler, i.e., block and unblock processes on either side of the channel, and 2) to transport messages. The initial capacity of the channel may be specified when the channel is first created; if omitted, the default capacity taken from an environment variable is used.

Interaction with the scheduler is needed for the cases of a full or empty channel. Receiving from an empty channel or sending to a full channel must block the acting process. Similarly, receiving from a full channel or sending to an empty channel must unblock the process on the other side of the channel.

Message transport over channels is protected by *busy-wait* mutexes: if a KP cannot obtain the channel's lock, it will explicitly yield to the scheduler between iterations, until it has finally obtained the lock. Busy-waiting allows other processes to proceed with their computations, while avoiding the complexities of a full-fledged sleep/wakeup mechanism. Furthermore, since channels are 1:1, at most two processes will compete for access to any given channel, so the expected number of spins in the case of contention on a channel is very small.

Since KPs are executing in a shared address space in our implementation, it is still possible that they modify each others state[6] and thus ruin the KPN semantics. There are at least two ways of implementing a channel transport mechanism that lessens the possibility of such occurrence:

- A message can be dynamically allocated and a pointer to it sent in a class that implements *move semantics* (e.g., `auto_ptr` from the C++ standard library).
- A message can be physically copied to/from channel buffers which is, in our case, done by invoking the copy-constructor.

We have initially implemented the first approach, which requires dynamic memory (de-)allocation for every message creation and destruction, but is essentially zero-copy. Our current implementation uses the second approach because measurements on Solaris have shown that memory (de)allocation, despite having been optimized by using Solaris's *umem* allocator, has larger overhead than copying as long as the message size is less than ~ 256 bytes. As of now, our implementation cannot choose between different mechanisms depending on the message size, so we recommend that large blocks be transfered as pointers.

Since C++ is a statically-typed language, our channels are also *strongly-typed*, i.e., they carry messages of only a single type. Since *communication ports* (endpoints of a channel; used by processes to send and receive messages) and channels are parametrized with the type of message that is being transmitted, compile-time mechanisms prevent sending messages of wrong types. Furthermore, the run-time overhead of dynamic dispatch based on message type are eliminated. Nevertheless, if dynamic typing is desired, it can be implemented by sending byte arrays over channels, or in a more structured and safe manner by using a standard solution such as *Boost.Variant* (see http://www.boost.org).

As KPs have only blocking read at their disposal, it is useful to provide an indication of no more messages arriving on the channel (EOF). One way of doing this is to send a message with specific value that will indicate EOF. However, all values of a type (e.g., `int`) might be meaningful in a certain context, so no value is available to encode the EOF indication. In such cases, one would be forced to use solutions that are more cumbersome to use and impose additional overhead (for example, dynamic memory allocation with `NULL` pointer value representing EOF). We have therefore extended channels by introducing support for *EOF indication*: the sender can set the EOF status on the channel when it has no more messages to send. After EOF on the channel has been set, the receiver will be able to read the remaining buffered messages. After all messages

[6]C++ is an inherently unsafe language, so there is no way of *preventing* this.

have been read, the next call to the port's `recv` method will immediately return false (without changing the target message buffer), and the next `recv` call will permanently block the process.

C. Deadlock detection and resolution

Deadlock detection and resolution makes it possible to execute KPNs in a finite space. Each time a process would block, either on read or on write, a deadlock detection routine is invoked. Since communication is 1:1, every cycle of blocked KPs is a ring; a property which greatly simplifies detection. The deadlock detection and resolution algorithm in our current implementation uses a centralized data-structure (the blocking graph) and thus must run while holding a single global mutex. If no cycle is found, the KP is blocked and this fact is recorded in the blocking graph. Otherwise, the capacity of the smallest channel in the cycle is increased by one, as suggested by [8]. Similarly, receiving from a full channel unblocks the sending side and removes the corresponding edge from the blocking graph.

D. Accounting

We have implemented a detailed accounting system that enables us to monitor many different aspects of Nornir's runtime performance, such as cpu time used by each process, number of context-switches, number of loop iterations in waiting on spinlocks, number of process thefts, number of messages sent to processes on the same or different CPU. We have measured (see [6] for methodology) that a single transaction consisting of [send $\rightarrow$ context switch $\rightarrow$ receive] takes $1.4\mu s$ with accounting enabled. When accounting is disabled, this time drops to $\sim 0.68\mu s$. The largest overhead in our accounting mechanism stems from the measurement of per-process CPU time, which requires a system call immediately before a process is run and immediately after a process returns to scheduler.

IV. Case study: H.264 encoding

KPNs are especially well suited for stream-processing applications. Indeed, each process is a function that takes as input one or more input data streams and produces one or more streams as output. H.264 is a modern, lossy video-compression format that offers high compression rates with good quality. Our KPN representation of an H.264 video encoder (see figure 2) is a slight adaptation of the encoder block diagram found in [11], with functional blocks implemented as KPs.

The input video consists of a series of discrete frames, and the encoder operates on small parts of the frame, called macroblocks, typically 16×16 pixels in size. The encoder consists of "forward" and "reconstruction" datapaths which meet at the prediction block (P). The role of the prediction block is to decide whether the macroblock will be encoded by using intra-prediction (relative to macroblocks in the *current* frame F_n) or inter-prediction (relative to macroblocks in the *reference* frame(s) F_{n-1}). The encoded macroblock goes through the forward path and ends at the entropy coder (EC), which is the final output of the encoder. The decision on whether to apply intra- or inter-prediction is based on factors such as the desired bandwidth and quality. To be able to estimate quality, the codec needs to apply transformations inverse to those of the forward path, and determine whether the *decoded* frame satisfies the quality constraints.

This example demonstrates that feedback loops are not only a matter of *convenience*, but actually *essential* for the expressive power of a programming framework. Thus, as already argued in the introduction, neither MapReduce nor Dryad can be used to implement the H.264 encoder.

V. Performance evaluation

To evaluate the performance and scalability of applications modeled as KPNs, we have used the H.264 KPN network (see section IV and Figure 2), a random network and a pipeline with artificial workloads, as well as AES encryption with a real workload. The test programs have been compiled as 64-bit with GCC 4.3.2 and maximum optimizations (`-m64 -O3 -march=opteron`). All benchmarks have been configured to use the work-stealing scheduling policy, initial channel capacity of 64 messages. Nornir has been compiled with accounting turned on, since this is necessary to study performance effects of deadlock detection. We have run them on an otherwise idle 2.6 GHz AMD Opteron machine with 4 dual-core CPUs, 64 GB of RAM running Linux kernel 2.6.27.3. Each data point is an average of 10 consecutive measurements of the total real (wall-clock) running time. This 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.

A. Results

H.264: For the purpose of evaluation of Nornir, we have used an artificial workload consisting of loops that consume the amount of CPU time which are on average used by a real codec in the different blocks. To gather this data, we have profiled x264, an open-source H.264 encoder, with the cachegrind tool and mapped the results to the H.264 block-diagram (see figure 2). The benchmark "encoded" 30 frames at rate of 1 frame per second (fps), with the number of workers varying from $1 \ldots 512$ in successive powers of two. From the results in figure 3 we can see that the performance gets slightly better as the number of workers per stage increases up to the number of CPUs, and remains constant afterwards. The best achieved speedup on 8 CPUs is only ~ 2.8; this limitation is caused by data-dependencies in the algorithm.

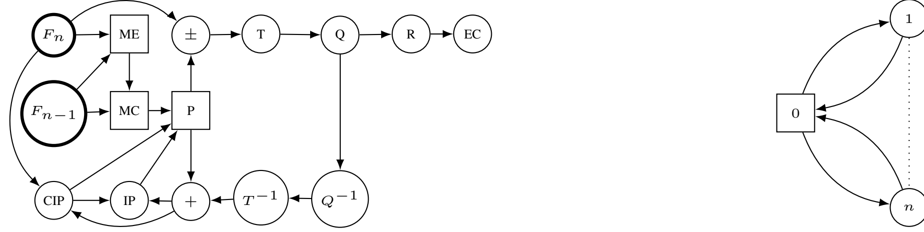


Figure 2. H.264 block-diagram, adapted from the H.264 whitepaper [11]. Inputs to the codec are current and reference frames (F_n and F_{n-1}). Since P, MC and ME blocks are together using over 50% of the total processing time and are perfectly parallelizable, we have parallelized each block by dividing the work over n workers. The figure on the right exemplifies parallelization of a single block.

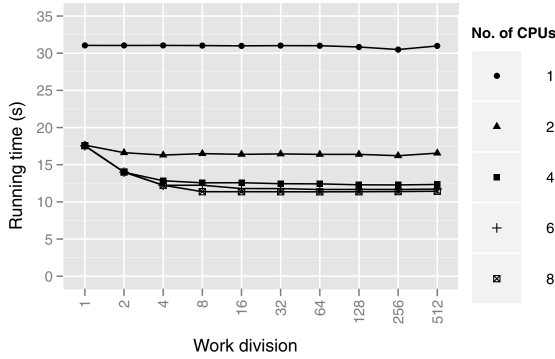


Figure 3. H.264: performance of "encoding" 30 frames at ~ 1 fps.

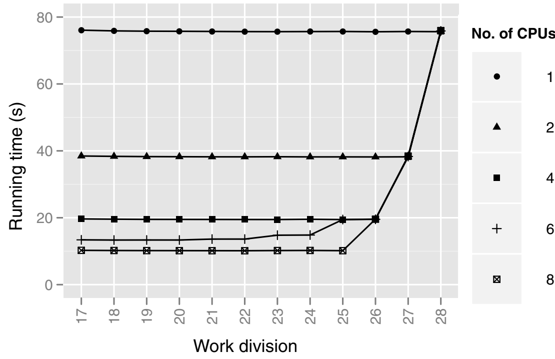


Figure 4. AES encryption benchmark.

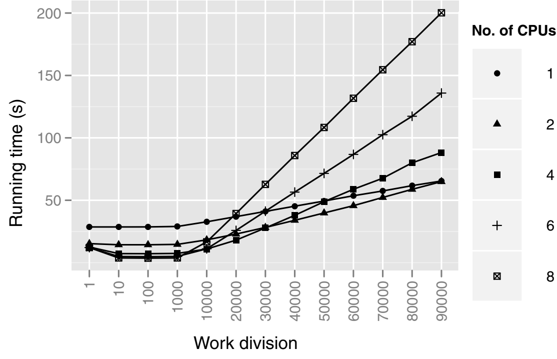
AES: The AES KPN has the same topology as the network on the right in figure 2; this topology is in fact common for algorithms that can process different chunks of input independently. The source KP (denoted "0") hands out equally-sized memory chunks to n worker KPs (denoted "1" …"n") which reply back when they have encrypted the given chunk. The exchanged messages are carrying only pointer-length pairs (16 bytes in total).

In this benchmark, we have set the total block size to 2^{28} bytes (256 MB), and the chunk given to each individual worker has been varied from to 2^{17} to 2^{28} and the total number of workers has been varied from 2048 to 1. The number of encryption passes that each worker will perform over its assigned chunk has been set to 40. The results, shown in figure 4, show perfect linear speedup with the number of CPUs, as soon as the number of workers becomes greater or equal to the number of CPUs. Note that the number of workers increases to the *left* in the figure, when given work division w (x-axis), the number of workers is 2^{28-w}. For $w = 28$ there cannot be any speedup on multiple CPUs because there is only a single worker process encrypting the whole chunk.

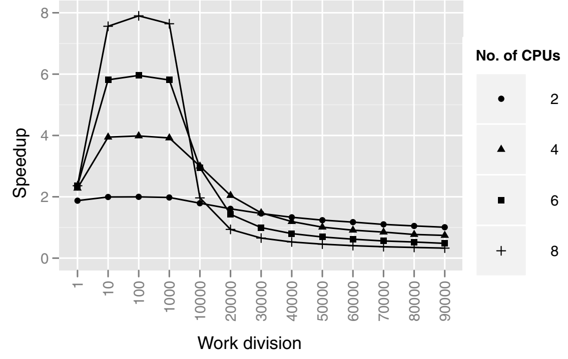
Random network: A random network is a directed graph consisting of a source KP, a number of intermediate KPs arranged in n_l layers (user specified) and a sink KP. The number of KPs in each layer is randomly selected between 1 and the user-specified maximum number m. The intention of this construction is to mimic, with fine granularity, network protocol graphs or parallelism in algorithms. The network may have additional b back-edges which create cycles. Each node is constrained to have at most one back-edge, be it outgoing or incoming.

The workload of the random network is determined by the formula nT/d, where n is the number of messages sent by the source, T is a constant that equals ~ 1 second of CPU-time, and d is the work division factor. In effect, each single message sent by the source (a single integer) carries a work amount equalling approximately $w = T/d$ seconds of CPU time. The workload w is distributed in the network (starting from the source KP) with each KP reading n_i messages from all of its in-edges. Once all messages are read, they are added together to become the t units of CPU-time which the KP is to consume before distributing t to its n_o forward out-edges. Then, if a process has a back-edge, a message is sent/received (depending on the edge direction) along that channel. As such, the workload w distributed from the source KP will equal the workload w collected by the sink KP. Messages sent along back-edges do not contribute to the network's workload; their purpose is solely to generate more complex synchronization patterns.

Figure 5(a) shows the absolute running times of a random



(a) Running time.



(b) Speedup over 1 CPU.

Figure 5. Benchmark of a random directed graph with 212 nodes, 333 edges and 13 cycles in 50 layers. The x-axis is not uniform.

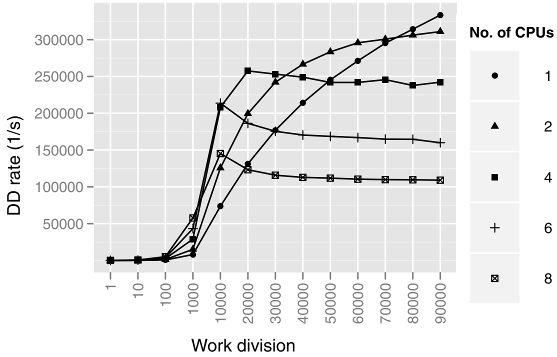


Figure 6. Deadlock detection rate on the random graph benchmark.

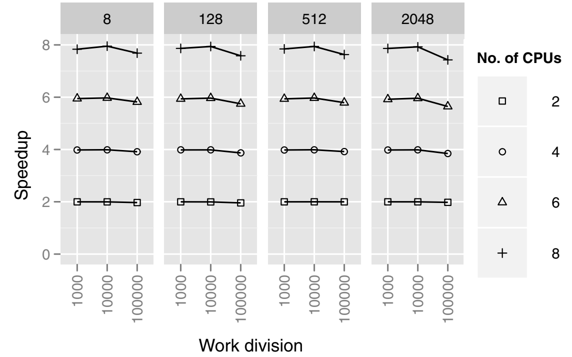


Figure 7. Pipeline speedup for message sizes of 8,...,2048 bytes and three different work divisions (d).

graph with cycles at different values of d, whereas figure 5(b) shows speedup over running time on 1 CPU. In our experiments, we have set $n = d$. As d increases to 100, the available parallelism in the network increases, and the running time decreases; in figure 5(b) we see that $d = 100$ achieves perfect speedup on multiple CPUs. At $d = 1000$, running time starts increasing linearly with d, and grows faster on more CPUs. This is caused by frequent deadlock detections, as witnessed by figure 6 which shows the number of started deadlock detections per second of running time. Since deadlock detection uses a global lock to protect the blocking graph, this limits Nornir's scalability on this benchmark. A possible way of avoiding this problem, in our current implementation, is to increase default channel capacity to a larger value. A long-term solution is implementing a distributed deadlock detection and resolution algorithm [12].

Pipeline: A pipeline does the same kind of processing as the random network, except that each layer has exactly one process and each process takes its only input from the preceding process in the pipeline. In all previous benchmarks, the communicated messages have been rather small (less than 32 bytes). We have used a pipeline consisting of 50 stages to study the impact of message size on performance. As previously, d messages have been generated by the source process, each containing $1/d$ seconds of CPU time. A noticable slow-down (see figure 7) happens regardless of message size and only at $d = 10^5$, which is equivalent to 10 μs of work per message, which is only 7 times greater than the time needed for a single transaction (see section III-D). As expected, the drop in performance at $d = 10^5$ is proportional to message size, but also to the number of CPUs. Larger message sizes take more time to copy, which causes greater contention over channel locks with increasing number of CPUs.

B. Summary

We have evaluated several aspects of Nornir: scalability of the scheduler with number of processes and CPUs,

overheads of copying message-passing and overheads of centralized deadlock-detection and resolution. Our findings can be summarized as follows:

- Nornir can efficiently handle a large number of processes. Indeed, in the AES benchmark, it achieved an almost perfect linear speedup of 7.5 on 8 CPUs with 2048 processes.
- Message sizes up to 512 bytes have negligible impact on performance. The cost of message copying starts to be noticeable at message size of 2048 bytes. Protecting channels with mutexes has negligible performance impact on 8 CPUs.
- As shown by the pipeline benchmark, context-switch and message-passing overheads start to have a noticeable impact on the overall performance when the amount of work per message is less than ~ 7 times the transaction time (see section III-D).
- The centralized deadlock detection and resolution algorithm can cause serious scalability and performance problems on certain classes of applications. In our evaluation, this was the case *only* for the random graph benchmark.
- Again, as shown by the random graph benchmark, the default channel capacity of 64 bytes, which we have used in our benchmark, can be too small in certain casses. Increasing it would mitigate overheads of deadlock detection, but it would also increase memory consumption.
- Performance can be further increased by turning off detailed accounting in cases where it is not needed.
- We have not noticed any scalability problems with using mutexes to protecting the scheduler's queues instead of using the non-blocking queue of [9].

Although there is room for improvement in Nornir (especially in deadlock detection), our results indicate that message-passing and KPNs in particular are a viable programming model for high-performance parallel applications on shared-memory architectures.

VI. Related work

Very few general-purpose KPN runtime implementations exist, among them YAPI [13] and Ptolemy II [14]. 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. 50 kB in our implementation), and the implementation did not have inherent support for multiple-CPUs. Ptolemy II is a Java-based prototyping platform for experimenting with various models of computation, and it spawns one thread for each Kahn process, which is rather inefficient for large networks. The amount of code that the JVM consists of would make it prohibitively difficult to experiment with low-level mechanisms, such as context-switches. PNRunner, a part of the Sesame project [15], is an event-driven *simulator* of embedded systems, which employs KPNs for application modeling and simulation. As such, it is not suitable for executing KPNs where performance is important.

Phoenix [3] is a MapReduce implementation optimized for multi-core architectures. We have reimplemented word count and k-means examples using Nornir, and found that our implementation outperforms that of Phoenix by factors of up to 2.7 and 1.7, respectively [7]. The main reason for this is that, unlike MapReduce, KPNs allow us to use algorithms and build a processing graph that are well matched to the structure of the underlying problem.

StreamIt [16] is a language for simplifying implementation of stream programs described by a graph consisting of computational blocks (filters) having a single input and output. Filters can be combined in fork-join patterns and loops, but must provide bounds on the number of produced and consumed messages, so a StreamIt graph is actually a synchronous dataflow process network [17]. The compiler produces code which can exploit multiple machines or CPUs, but their number is specified at compile-time, i.e., a compiled application cannot adapt to resource availability.

Pig latin [2] is a language for performing ad-hoc queries over large data sets. Users specify their queries in a high-level language which provides many features of operators found in SQL. Unlike SQL, which is declarative and heavily relies on query optimizer for efficient query execution, Pig latin allows users to precisely specify *how* the query will be executed. In effect, users are constructing a dataflow graph which is then compiled into a pipeline of MapReduce programs and executed on a Hadoop cluster, which is an open-source, scalable implementation of MapReduce. All of the Pig latin operators, such as `FILTER` and `JOIN`, are directly implementable as Kahn processes. Taking our experimental results [7] into consideration, we believe that compiling Pig latin programs into Nornir graphs would be advantageous for their performance on multi-core machines.

VII. Conclusion and future work

In this paper we have described implementation details of Nornir, our run-time environment for executing parallel applications specified in the high-level framework of Kahn process networks, which allow cycles in the communication graph of the program. Since this feature is crucial for implementing iterative algorithms such as H.264 encoding, Nornir complements existing frameworks such as MapReduce and Dryad.

We have evaluated Nornir's efficiency with several synthetic (H.264 encoding, random KPN, pipeline) and one real (AES) application on an 8-core machine. Our results indicate that Nornir can scale well, but that in certain cases (random KPN) the centralized deadlock detection is detrimental for performance, and that default channel capacity of 64 bytes is too small for some applications. We have also found that copying semantics of message-passing starts

having a slight, but noticeable impact on performance at message sizes of ~ 2048 bytes. Furthermore, Nornir can support parallelism at fine granularity: its overheads become noticeable at processing time of $10\mu s$ per message, which is ~ 7 times greater than the combined overhead of scheduling and message-passing.

The first, and most important, step in our future work is increasing Nornir's scalability by replacing a centralized deadlock detection algorithm with a a distributed one [12]. This will also be the first step towards a distributed version of Nornir, executing on a cluster of machines. Further performance increases can be gained by using non-blocking data structures. In the scheduler, we might need to use the non-blocking queue of [9] instead of mutexes in order to support scalability beyond 8 CPUs. We might also use a single-producer, single-consumer FIFO queue [18] to avoid yielding between en-/dequeue attempts. Since yielding incurs switching to new control flow and new stack, we expect that this improvement will further increase performance by reducing pressure on data and instruction caches.

References

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *Proceedings of Symposium on Opearting Systems Design & Implementation (OSDI)*. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[2] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2008, pp. 1099–1110.

[3] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 13–24.

[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems*. New York, NY, USA: ACM, 2007, pp. 59–72.

[5] G. Kahn, "The semantics of a simple language for parallel programming." *Information Processing*, vol. 74, 1974.

[6] Željko Vrba, P. Halvorsen, and C. Griwodz, "Evaluating the run-time performance of Kahn process network implementation techniques on shared-memory multiprocessors," in *Proceedings of the International Workshop on Multi-Core Computing Systems (MuCoCoS)*, 2009.

[7] Željko Vrba, P. Halvorsen, C. Griwodz, and P. Beskow, "Kahn process networks are a flexible alternative to MapReduce," in *Proceedings of the International Conference on High Performance Computing and Communications*, 2009.

[8] M. Geilen and T. Basten, "Requirements on the execution of kahn process networks," in *Programming Languages and Systems, European Symposium on Programming (ESOP)*. Springer Berlin/Heidelberg, 2003, pp. 319–334.

[9] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *Proceedings of ACM symposium on Parallel algorithms and architectures (SPAA)*. New York, NY, USA: ACM, 1998, pp. 119–129.

[10] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, 2007, also available as Sandia National Labs Tech Report SAND2006-6450C.

[11] I. E. G. Richardson, "H.264/mpeg-4 part 10 white paper," Available online., http://www.vcodex.com/files/h264_overview_orig.pdf.

[12] G. Allen, P. Zucknick, and B. Evans, "A distributed deadlock detection and resolution algorithm for process networks," *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, vol. 2, pp. II–33–II–36, April 2007.

[13] E. de Kock, G. Essink, W. J. M. Smits, R. van der Wolf, J.-Y. Brunei, W. Kruijtzer, P. Lieverse, and K. Vissers, "Yapi: application modeling for signal processing systems," *Proceedings of Design Automation Conference*, pp. 402–405, 2000.

[14] 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)," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-28, Apr 2008. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-28.html

[15] M. Thompson and A. Pimentel, "Towards multi-application workload modeling in sesame for system-level design space exploration," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, vol. 4599/2007, 2007, pp. 222–232. [Online]. Available: http://www.springerlink.com/content/u4265u6r0u324215/

[16] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2006, pp. 151–162.

[17] E. Lee and T. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.

[18] J. Giacomoni, T. Moseley, and M. Vachharajani, "FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue," in *PPoPP: Proceedings of the ACM SIGPLAN Symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2008, pp. 43–52.