The Journal of Supercomputing manuscript No.
(will be inserted by the editor)

The *Nornir* run-time system for parallel programs using Kahn process networks on multi-core machines – A flexible alternative to MapReduce

Željko Vrba · Pål Halvorsen · Carsten Griwodz · Paul Beskow · Håvard Espeland · Dag Johansen

Received: date / Accepted: date

Abstract Even though shared-memory concurrency is a paradigm frequently used for developing parallel applications on small- and middle-sized machines, 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 such as MapReduce and Dryad that are recognized for their efficiency and simplicity. Unlike these frameworks, Nornir also supports process structures containing branches and cycles. Nornir is based on the formalism of Kahn process networks, which is a shared-nothing, message-passing model of concurrency. We deem this model 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 scales almost linearly with the number of CPUs, when not limited by data dependencies. We also show that the modeling flexibility allows Nornir to outperform its MapReduce counter-parts using well-known benchmarks.

Keywords Parallel processing · Kahn process networks

1 Introduction

The introduction of commodity multi-core processors has spawned a wide interest in parallel programming. It is, however, widely recognized that devel-

Željko Vrba, Pål Halvorsen, Carsten Griwodz, Paul Beskow, Håvard Espeland
Simula Research Laboratory, Oslo, Norway
Department of Informatics, University of Oslo, Norway
E-mail: {zvrba,paalh,griff,paulbb,haavares}@ifi.uio.no

Dag Johansen
Department of Computer Science, University of Tromsø, Norway
E-mail: dag@cs.uit.no

oping parallel and distributed programs is inherently more challenging than developing sequential programs. As a consequence, several frameworks that aim to make such development easier have emerged, such as Google's MapReduce [16], Yahoo's PigLatin [30], which uses Hadoop [2] as the back-end, Phoenix [32] and Microsoft's Dryad [24]. All of these frameworks are gaining popularity, but they lack a feature that is critical to our application domains: the ability to model branching and 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, where cycles are more the rule than the exception (see figure 6 for an example). Thus, we cannot use any of the existing frameworks, so we have turned towards the flexible formalism of Kahn process networks (KPN) [25]. 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, the main ones being determinism and composability. Determinism guarantees that a program, given the same input, will behave identically on each run. This significantly eases debugging, which is otherwise a notoriously hard problem with parallel and distributed computations. Composability guarantees that assembling independently developed components yields the expected results.

In an earlier paper [38], we evaluated implementation options for KPNs on shared-memory architectures. In a follow-up paper [40], we presented case studies of problem modeling with KPNs and compared the resulting KPN models with MapReduce models. 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 optimized for multi-core machines. This paper expands our previous paper [41] with more detailed descriptions and further performance experiments. The implementation details of Nornir, which we describe in this paper, are somewhat different than the implementations of our previous KPN runtimes described in [38,40]. Most notably, we have added support for pluggable scheduling policies to allow easy experimentation. We also investigate the performance and scalability characteristics of Nornir using a set of benchmarks on a workstation-class machine with 8 cores. The experiments reveal some weaknesses of our current implementation, but indicate nevertheless that KPNs are a viable programming model for parallel applications on shared-memory architectures.

The rest of this paper is organized as follows. In the next section, we present a brief overview of related work. In section 3, we present the properties and basic ideas behind KPNs, and follow up with the description of Nornir implementation details in section 4. In section 5, we describe the benchmarks we designed and report our performance results. We summarize the paper and present concluding remarks in section 6.

2 Background and Related work

A lot of research has been dedicated to addressing the challenge of parallel and distributed programming, which has led to the development of many tools, programming languages and frameworks. Here, we give a short overview of this work – for a more detailed summary, please refer to [37].

2.1 Low-Level Tool Support

Low-level tools can be roughly categorized into those enabling shared-state concurrency and those enabling message-passing concurrency. Shared-state concurrency has traditionally been most easily accessible from imperative programming languages such as C, C++ and FORTRAN. These languages have little built-in support for concurrency, so the developers must use the operating system's threading facilities or 3rd-party libraries, such as threading building blocks [23], which encapsulate common concurrency patterns. Another choice is OpenMP [4], which is an extension of C, C++ and FORTRAN oriented mostly towards loop-level parallelism. μC++ [9] is another extension of the C++ programming language, which introduces a number of concurrent programming concepts, such as cooperatively-scheduled co-routines [26] and tasks that run in parallel and are scheduled preemptively.

Message-passing concurrency has traditionally been used for implementing programs running in different address spaces or on different machines. Erlang [6] is an example of a pure functional programming language with built-in support for message-passing concurrency. In Erlang, processes (actors) communicate by sending messages via mailboxes, and the run-time provides extensive support for failure detection and recovery.

In most other languages, message-passing concurrency is delegated to library support, the most prominent examples of which are parallel virtual machine (PVM) and message passing interface (MPI). PVM [1] aims at support for heterogeneous concurrent computing, creating an illusion of a distributed virtual machine. It thus provides means for process and resource control and fault-tolerance along with communication primitives. MPI [3], on the other hand, is somewhat narrower in scope than PVM. Its main task is to support efficient point-to-point communication and some complex collective operations. Data types may be automatically serialized for network transport, and newer MPI versions have added many new features like dynamic process management and parallel I/O.

The tools described above provide sufficient *mechanisms* for building distributed and parallel applications, but they leave the definition of higher-level abstractions and policies to developers. For example, PVM and MPI enable inter-task communication, but do not endorse any particular application structure or concurrency model. In this respect, large players (Google and Microsoft in particular) have identified common patterns of distributed computations and packaged them in easier-to-user frameworks, which we describe in the next section.

2.2 High-Level Frameworks

Industrial actors have developed solutions for simplified distributed processing of large data quantities. Examples are Google's MapReduce [16], Yahoo's PigLatin programming language [30], Microsoft's Dryad [24] and Cosmos systems as well as the programming language Scope [11], and IBM's System S and programming language SPADE [17]. Dryad, Cosmos and System S have many properties in common: all use directed graphs to model computations and execute them on a cluster of machines. In addition, System S supports cycles in graphs, while Dryad supports non-deterministic constructs. Thus, the deterministic subset of the Dryad system is also a subset of the KPN framework, while the expressiveness of System S is equivalent to that of KPNs. However, not much is known about these systems and their availability is limited.

MapReduce [16] has become one of the most cited paradigms for expressing parallel computations. Unlike the above systems, which define task-parallel models, MapReduce defines a data-parallel model based on keys and values. While the original MapReduce paper specifies the programming model rather informally, Lämmel [27] has used the Haskell [22] programming language to rigorously define the semantics of the MapReduce model and Sawzall programming language [31]; there he also discusses parallellization issues. Google's MapReduce implementation supports fault-tolerant distributed execution in clusters. Others have reimplemented MapReduce for clusters (Hadoop [2], an open-source implementation in Java), multi-core machines [32], the Cell BE architecture [14] and even for GPUs [21], all of which bear witness of its popularity. Map-Reduce-Merge [12], adds a merge step to efficiently process data relationships among heterogeneous datasets, and to be able to execute in parallel join algorithms of relational algebra, operations not directly supported by the plain MapReduce model. The same issues are also addressed by Oivos [35], which in addition provides a more expressive, declarative programming model. Finally, reducing the added overhead of layering software on top of MapReduce is the goal of Cogset [36] where the processing architecture is changed to increase performance.

In our work, however, we are generally interested in real-time processing of multimedia content. This requires high performance and repeated non-trivial sequences of processing steps, but the above frameworks are mostly targeted towards off-line batch processing of data. MapReduce, Dryad and Cosmos are not able to model iterative algorithms; in addition the rigid MapReduce semantics are not a good fit for all problems [12, 37], which may lead to unnaturally expressed solutions and decreased performance [40]. Finally, Google's recent patent on MapReduce [15] may prompt commercial actors to look for an alternative framework.

In this respect, the KPN framework is a viable alternative, but, in practice, very few general-purpose KPN run-time implementations exist. Known examples include the Sesame project [34], the process network framework [5, 29], YAPI [13] and Ptolemy II [8]. PNRunner, a part of the Sesame project, is an event-driven simulator of embedded systems that employs KPNs for application modeling and simulation. However, since an event-driven simulation significantly slows down execution, Sesame is not suitable for executing KPNs where performance is important. The process network framework supports distributed execution and deadlock detection in KPNs, but only a 1:1 scheduling model. It would require substantial extensions to introduce support for m:n scheduling. YAPI is not a pure KPN implementation, as it extends the semantics and thus introduces the possibility of non-determinism. In addition, it is unclear whether the implementation can use multiple-CPUs.[1] Ptolemy II is a Java-based prototyping platform for experimenting with various models of computation, and it spawns one thread for each Kahn process. The amount of code that comprises the JVM would make it prohibitively difficult to experiment with low-level mechanisms, such as context-switches.

Another framework based on the process network paradigm is StreamIt [20]. It is a language and a run-time system 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 data-flow process network [28]. The compiler produces code that can exploit multiple machines or CPUs, but their number is specified at compile-time, i.e., a compiled application cannot adapt to resource availability.

Thus, all of the existing frameworks have some short-comings that are difficult to address, which has motivated us to implement a new KPN run-time from scratch. Before describing the implementation details of Nornir, we shall first describe the basic KPN semantics in the next section.

3 Kahn process networks

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

[1] Inspection of source code indicates that some support for multiple CPUs is built into YAPI. Experiments on Solaris, however, have revealed that YAPI never actually uses more than a single CPU.

– *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 output of a process computing $f(x)$ to the input of a process computing $g(x)$ is guaranteed to compute $g(f(x))$. Therefore, small KPNs 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 [37] 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 [32].
– *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 [25] 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 only allowed to share data 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 (from the KPN model point of view), but trying to receive a message from an empty channel *blocks* the process until a message becomes available. Testing for available messages is not permitted. 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 the process execution order. Every model that relaxes one of these conditions, e.g. allowing non-blocking reads or polling, results in non-deterministic behaviour.

[2] Provided that processes themselves are deterministic.

[3] Inexperienced developers expect that mutexes and condition variables wake up waiting threads in FIFO order, whereas the wake-up order is in reality imlementation-dependent and often non-deterministic.

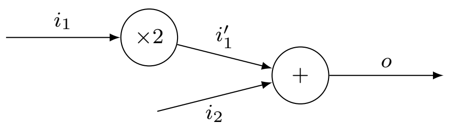


Fig. 1 An example KPN. i_1 and i_2 are external input channels to the network (assumed to be numbers), o is the external output channel, and i'_1 is an internal channel. The inputs and the output are related by the formula $o = 2i_1 + i_2$

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 the construction of KPNs that need unbounded space for their execution. However, any real implementation is constrained to run in finite memory. A 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, although they would continue running in the theoretical model. The algorithm of Geilen and Basten [18] resolves the deadlock by traversing the cycle to find the channel of least capacity and enlarges it by one message, thus resolving the deadlock.

It also is worth noting that KPNs are not a universal solution for the inherently difficult problem of developing parallel and distributed applications. Even though determinism is a desirable property from the standpoint of program development, it limits the application areas for KPNs. For example, a disk scheduler is a simple use-case that cannot be appropriately modeled with KPNs. The scheduler must periodically serve all clients in some order, say round-robin, to preserve fairness. However, 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 relaxing the KPN formalism by introducing non-deterministic construct(s) such as $m : n$ channels and/or polling, which would reduce its conceptual value.

4 Nornir

The Nornir run-time system is implemented in C++, and 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.

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

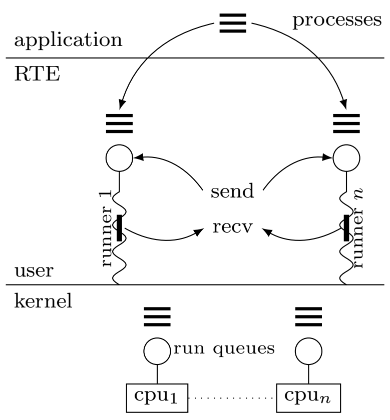


Fig. 2 KP scheduling: each runner has its own private run queue (small circles) containing ready KPs (small black squares), and is executing at most one KP at any given time. Runners in Nornir are bound to different CPUs, so they never compete with each other, but they compete with other threads and processes on the machine.

4.1 Process scheduler

Nornir implements both preemptive 1:1 and cooperative m:n scheduling models. In the 1:1 model, one OS thread is used for each KP, blocking mutexes are used to protect the channels, and condition variables are used as the sleep/wakeup mechanism. We have investigated this approach in an earlier paper [38] and showed that the m:n scheduling model is considerably more efficient. Because of this, we focus here on the m:n scheduling model, which we have also used for our performance evaluations.

With an m:n model, many KPs are time-multiplexed onto one OS thread, which we call *runner*. Since the m:n scheduler is cooperative, a KP runs uninterrupted until it exits or encounters a blocking point. In our implementation, KPs yield implicitly only in message send and receive operations. However, since KPs may use all UNIX system calls, they can perform I/O in the usual way. In that case, I/O operations may block without causing the calling KP to block or yield. The runner executing the KP will be blocked und unable to dispatch another KP. This can be avoided by using either asynchronous I/O or dedicated runners for KPs that perform I/O.

The m:n scheduler uses user-mode code to switch contexts between a KP and the scheduler core. On Solaris and Linux running on AMD64, 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.

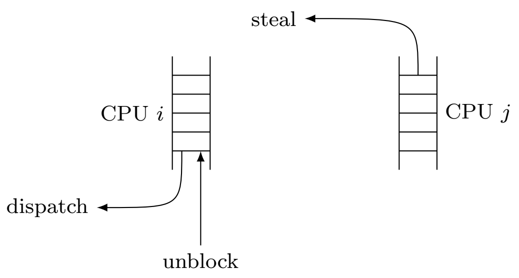


Fig. 3 Queue operations in work stealing. CPU i accesses its own queue only at the front, while it steals tasks from other CPUs only from the back.

4.2 Scheduling policies

Since KPNs are deterministic, they provide great freedom in implementing the process scheduler: any *fair* scheduler results 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.

We have thus made Nornir configurable with different scheduling policies. In addition to the classical work-stealing [7] algorithm, we have also implemented and evaluated a modified work-stealing algorithm and a scheduling algorithm based on graph-partitioning [10]. The latter tries to balance the computational load evenly while reducing the amount of inter-CPU synchronization and communication.

The *work-stealing* algorithm [7] is illustrated in figure 3. When the KPN is started on a system with m CPUs, m runner threads ("runners") are created and pinned to different CPUs. Each runner has a private run queue of ready KPs and executes the KP at the front of its queue (*dispatch*). If this queue is empty, it tries to *steal* a KP from the back of the queue of a randomly chosen runner. KPs that become ready (*unblock*) are added at the front of a queue. We use two algorithms for determining this queue. The original algorithm always prepends a newly unblocked KP to the queue of the CPU that executes that currently running KP which triggered the unblocking (*WS-CUR*). This algorithm causes performance problems for some workloads, including the scatter-gather kind of load. They are caused by high contention over run-queues when stealing. We have therefore modified the work-stealing algorithm so that an unblocked KP is placed on the last CPU that executed it [39] (*WS-LAST*). We do not use the non-blocking queue described in [7]; instead we use an ordinary mutex per run queue. This not only simplifies the implementation, but is also *required* for implementing our WS-LAST modification since the non-blocking queue does not support all of the operations needed to support the modified algorithm. This might become problematic on machines

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

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 *graph-partitioning* algorithm assigns weights to the nodes and edges in the process graphs. The weight of a node is proportional to the processing power required by the KP and the weight of an edge is proportional to the communication volume between the two processes incident to that edge. The algorithm tries to partition the graph into disjoint sets of nodes so that all node subsets have approximately equal total node weights. In addition, the algorithm tries to minimize the cut cost, i.e., the total weight of edges crossing partitions. Graph partitioning is an NP-hard problem, so we have implemented a heuristic based on the work presented by Devine et al. [10].

4.3 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, a default capacity is used.

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

Message transport over channels is protected by mutexes that are essentially implemented by *busy-waiting*, but yield to the scheduler for every failed locking attempt. Combined with the user-space work stealing scheduler, this is less wasteful than a spinlock and much faster than a full-fledged sleep/wakeup mechanism. Furthermore, since channels are 1:1, at most two processes 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 other's 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 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

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

copying as long as the message size is less than ~ 256 bytes. We have not tested suballocators separately; the performance would likely be very similar to message copying with a small constant message size.

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 is 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 or C++ with run-time type information or *Boost.Variant* (see `http://www.boost.org`).

As KPs have only blocking read at their disposal, it is useful to provide an indication when no more messages arrive on the channel (EOF). One way of doing this is to send a message with specific value that indicates 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 and that 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 is able to read the remaining buffered messages. After all messages have been read, the next call to the port's `recv` method returns false immediately (without changing the target message buffer), and the next `recv` call block the process permanently.

4.4 Deadlock detection and resolution

Deadlock detection and resolution make it possible to execute many[7] KPNs in 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, read and write are blocking and polling is forbidden, it is straight-forward to notice when a ring of blocked processes is closed. 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 [18]. Similarly, when a process receives from a full channel, the upstream KP is unblocked and the blocking graph updated.

[7] As demonstrated in [37], it is possible to construct KPNs that are under no circumstances executable in finite space.

4.5 Accounting

We have implemented a detailed accounting system that enables us to monitor many different aspects of Nornir's run-time 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 a different CPU. We have measured (see [38] for methodology) that a single transaction, consisting of [send $\rightarrow$ context switch $\rightarrow$ receive] operations, 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-KP CPU time, which requires a system call immediately before a KP is run and immediately after a KP returns to scheduler.

5 Performance evaluation

We have evaluated performance aspects of Nornir in particular, but also of the KPN modeling paradigm in general:

- Effects of scheduling policies on performance.
- Scalability of Nornir with respect to KPN size and parallelism granularity.
- Performance consequences of using MapReduce or KPNs for modeling parallel applications.

To investigate these issues, we have designed and implemented four synthetic workloads: an H.264-encoding process graph, an Advanced Encryption Standard (AES) encryption algorithm, a random process network and a pipeline. In addition, we have implemented the canonical MapReduce example, word-frequency program, on top of Nornir in two ways: 1) by implementing the exact MapReduce semantics within the KPN framework and using it to solve the word-frequency problem, and 2) by designing and implementing a KPN that is best suited for the problem. Lastly, we have compared the performance of both of our solutions to the solution implemented on top of Phoenix [32], which is a MapReduce implementation designed specifically for multi-core machines.

In the following subsections, we first describe our experimental setup and methodology and proceed to present results in the order outlined above. Workloads are described in detail as they are encountered.

5.1 Methodology

The test programs have been compiled as 64-bit with GCC 4.3.2 and maximum optimizations (`-m64 -O3 -march=opteron`). 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 *median*[8] of 9 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.

All benchmarks, except those of section 5.2, have been configured to use our modified work-stealing algorithm and initial channel capacity of 64 messages, with deadlock detection enabled.

5.2 Choice of Scheduler

As stated in section 4.1, a KPN yields identical results when run by any fair scheduling algorithm, but the running time may be highly dependent on the chosen algorithm. In this section, we investigate the relative merits of the original work-stealing algorithm (WS-CUR), our modified version (WS-LAST) and the method based on graph partitioning (GP). In order to separate performance effects of scheduling and deadlock detection, the latter has been turned off for this series of experiments. Nevertheless, all benchmarks did finish without any artificial deadlock problems.

5.2.1 Original vs. modified work-stealing

We have performed extensive comparison of application running times under WS-CUR and WS-LAST policies. We have found that both algorithms lead to very similar application performance[9], *except* on a specific type of workload where WS-LAST is clearly superior.

The KPN for this workload, which we have named *scatter-gather*, is shown in figure 4. This structure is typical for embarrassingly-parallel problems, i.e., problems that can be split into parts that can be processed independently of each other. The AES encryption benchmark described later is an example of such a problem. There are also workloads that proceed in stages, where all sub-problems of one stage must be finished before starting work on the next stage; in such cases KP P_0 acts as a barrier. An example of such a situation is the synchronization between Map and Reduce stages of a MapReduce computation. The scatter-gather benchmark executes in m rounds. In each round, P_0 first sends a single message to each of the n workers, and then waits to receive the same number of replies from each worker. The workers are oblivious to the round-based execution: a worker just receives a message, uses a fixed amount of CPU time t and sends a reply to P_0. For the results shown in

[8] We have investigated also correlation with other quantities, such as rate of steal attempts. Using median allowed us to use the values of other measured variables *as is*. Mean value would not correspond to any particular measurement, and it would be unclear how the values of other measured quantities could be interpolated. The observed variation in running times is small, so the difference between using median and mean is negligible.

[9] We have used also the other workloads described later in this paper for our comparisons. On all of our measurements, the difference between mean running times under WS-CUR and WS-LAST was within 2%, most often in favor of WS-LAST.

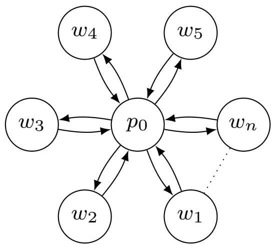


Fig. 4 Scatter-gather topology.

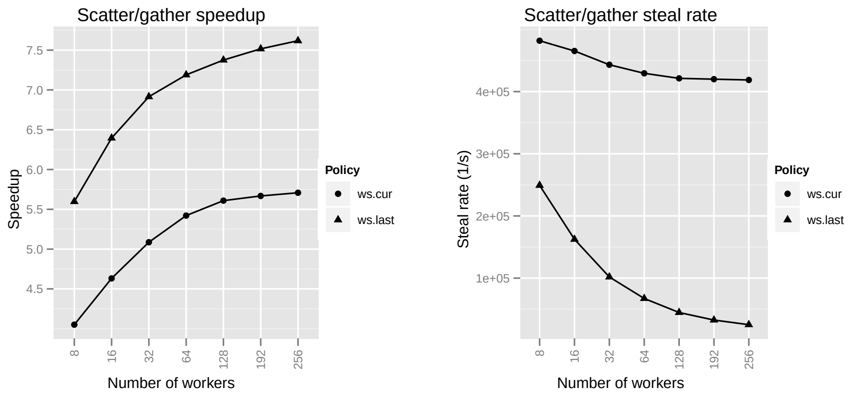


Fig. 5 Scatter/gather on 8 CPUs: speedup and steal rate vs. number of workers.

figure 5, we have used $m = 12500$, $t = 10^{-4}$ seconds, and varied n over the set $n \in \{8, 16, 32, 64, 128, 192, 256\}$. Thus, the total useful work performed by all workers is $W = nmt = 1.25n$ CPU seconds.

Compared to running the whole workload on a single core, we can observe that speedup increases with the number of workers. For example, using 256 workers, WS-CUR achieves a speedup of 5.7 whereas WS-LAST achieves a speedup of 7.6. As can be observed in figure 5, the speedup is directly related to the reduction in the number of steal attempts per second. The number of steal attempts is reduced because WS-LAST actively distributes the load over all CPUs, thus significantly reducing contention over the run-queues. WS-CUR, on the other hand, always wakes up *all* workers to the *same* CPU, i.e., the one on which P_0 is running. This creates significant contention over a *single* run-queue at the beginning of each round, which leads to significant loss of performance. This contention-reducing property will be even more important for future applications as the number of cores in computer systems is steadily increasing.

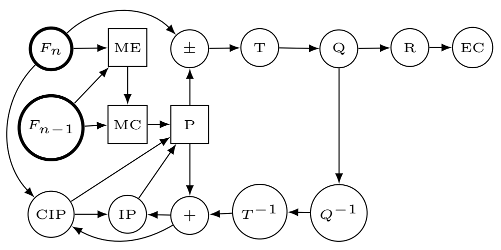


Fig. 6 H.264 block-diagram, adapted from the H.264 whitepaper [33] where the inputs to the codec are the current (F_n) and reference (F_{n-1}) frames. Here, each square block assumes the role of P_0 in figure 4 and has attached n workers.

5.2.2 Modified work-stealing vs. graph partitioning

Scheduling based on GP needs a mechanism that can detect load-imbalance and trigger repartitioning when the load-imbalance has become intolerably high. For that purpose, we measure the idle time accumulated by all runners and trigger repartitioning when it exceeds a preset constant threshold τ.

GP has complex performance characteristics which required running an extensive set of benchmarks across different parameter sets. Due to the complexity of accurately characterizing GP behavior, we only list our main conclusions and refer the reader to [37] for full details.

Application running time. Compared to GP, WS-LAST yields in general best performance on all workloads. Provided that there is enough parallelism in the program and that a process performs a sufficient amount of work each time it is scheduled, WS-LAST always achieves speedup almost proportional to the number of CPUs. Speedup using GP, however, is always significantly lower – in some pathological cases, programs running under GP on multiple CPUs takes *longer* to execute than when running on a single CPU.

To illustrate this, we used the H.264 benchmark[10] that simulates the data flow in an H.264 codec. Figure 6 shows our H.264 KPN, which is only a slight adaptation of the encoder block diagram found in [33]. The functional blocks of the H.264 encoding process are implemented as KPs using synthetic loops to spend the same amount of time for each "encoded" frame that would be used by a real codec. We have obtained these timings by profiling x264, which is an open-source H.264 encoder implementation, and mapping the results to the shown process graph. Since the P, MC and ME stages together use more than 50% of the total CPU time, we have parallelized each of these stages in the same way as shown in figure 4. 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

[10] This example demonstrates that feedback loops are not only a matter of convenience, but actually *essential*. The H.264 benchmark is a synthetic workload 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.

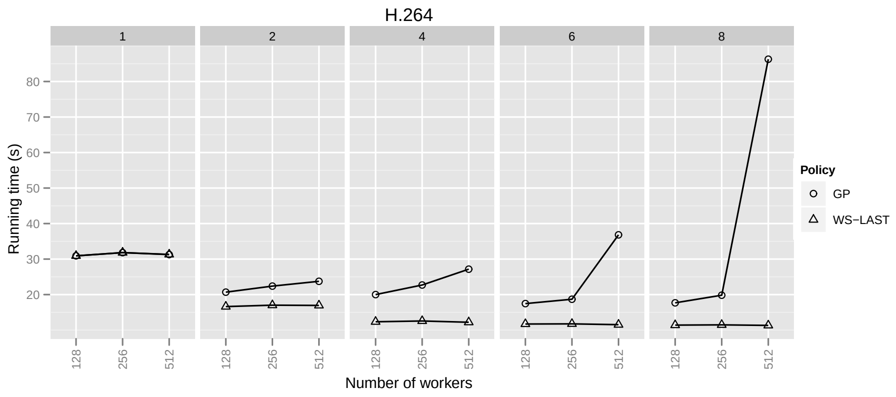


Fig. 7 Median running times for WS-LAST and GP on 1,2,4,6 and 8 CPUs.

"forward" and "reconstruction" data paths which meet at the prediction block (P). The role of the prediction block is to decide whether the macroblock is 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.

In figure 7, we plot the performance results for both GP and WS-LAST running the H.264 benchmark. The main reason of performance degradation with increasing number of workers is the limited parallelism available in the H.264 network. Under GP, runners accumulate idle time faster than our simple heuristics for deciding when to rebalance the load is able to catch up. This means frequent repartitioning, which is protected by a single mutex, and process migration. These operations are performed hundreds of times per second, leading to severe performance degradation.

Variance of application running times. Our experiments have confirmed that GP indeed fulfils its design goal of reducing traffic between processes on different CPUs. However, we have also observed that its load-balancing properties are rather poor. This is illustrated using a random network that 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 (see figure 8 for an example). 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 back-edges that create cycles, with each node being constrained to have at most one back-edge, be it outgoing or incoming. The

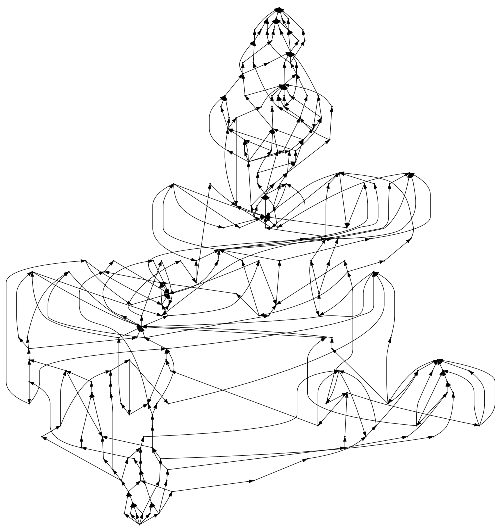
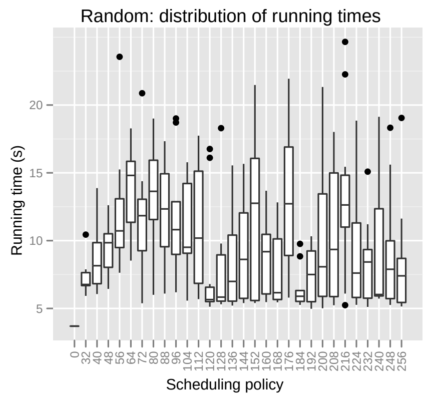


Fig. 8 A random graph with 212 nodes, 333 edges and 13 cycles with the source node on bottom and sink node on top.

workload is generated by sending and receiving messages that contain plain integers; each integer denotes the amount of CPU time that the receiving KP spends on processing, using a synthetic loop, before sending a reply or receiving another message. More specifically, the source KP sends n messages, each containing a fixed amount of work corresponding to w seconds of CPU time. A KP reads a single message from each of its n_i inputs and adds them together to compute the total amount t of CPU-time that is to be consumed. Then, the KP runs a synthetic loop to use the number of CPU seconds proportional with t, and then it distributes t to its n_o forward out-edges. A message representing $\lfloor t/n_o \rfloor$ seconds of CPU time is sent to each of the n_o forward out-edges; if t is not divisible by n_o then the remainder is added to the last outgoing edge. Lastly, if a KP has a back-edge, a message is sent/received (depending on the edge direction) along that channel. As such, the workload w in a single message sent from the source KP equals 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. The total CPU load W generated by the network is determined by the formula $W = nT/d$, with each single message representing a CPU load of $w = T/d$ CPU seconds. Here, T is the number of iterations of our synthetic loop that uses 1 second of CPU time on the benchmark machine, and d is a quantity that we have named *work division factor*. For example, when $d = 10000$, each message by the source represents 10^{-4} seconds (100μs) of



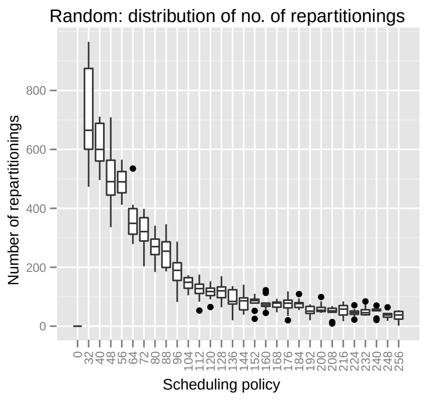


Fig. 9 Illustration of GP variance in running time for the Random workload on 8 CPUs and $d = 1000$, which is one step past the WS peak speedup. x-axis is the value of the idle time parameter τ for the GP policy, and $\tau = 0$ shows the results for WS.

CPU time. This allows us to control the granularity of chunks into which the total workload W is divided.[11]

Figure 9 shows a box-and-whiskers plot of the distribution of running times and number of repartitionings for different values of the τ parameter; $\tau = 0$ corresponds to WS-LAST. As expected, larger τ causes fewer repartitionings. However, it has very little influence on the running time. When $\tau \geq 72$, the *minimal* measured running times are approximately constant, and around ~5s (WS-LAST achieves running time of ~2.5s), while the *maximal* running times vary widely. In general, the *variance* of running times for any given τ is large, i.e., the running time of any given experiment run is very unpredictable and uncorrelated with the repartitioning frequency. We deem that this is the biggest drawback of GP-based scheduling and therefore recommend that users perform extensive performance tests with their particular workloads before employing GP-based scheduling in production.

Traffic patterns. As explained earlier, the GP algorithm is designed to satisfy two mutually incompatible constraints: balance the load across multiple CPUs and minimize the traffic between processes on different CPUs. We have observed that there is much more local traffic under GP scheduling. For example, on the random graph benchmark under GP on 8 CPUs, *at least* 70% of all messages are communicated between KPs on the same CPU for all values of work division factor. On the same benchmark under WS-LAST, *at most* 20% of all messages constitute local traffic, and this ratio decreases as work division increases.

[11] Note that T/d equals the *maximum* work performed by any single KP. The *lower* bound on the amount of work performed by any KP is T/md where m is the maximal number of KPs in any layer.

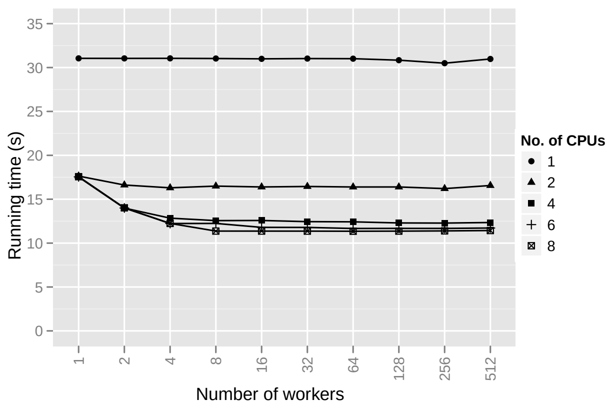


Fig. 10 H.264: performance of "encoding" 30 frames at ∼ 1 fps.

Based on these results, we use WS-LAST for the rest of the experiments evaluating different application scenarios. Note, however, that we are running on a single, multi-core machine. When the communication cost increases, for example by extending the investigation to a compute cluster, the benefit of using graph partitioning will increase (but this is subject to further experiments).

5.3 Scalability of Nornir

We have used several different benchmarks to evaluate scalability of Nornir as well as the overheads of centralized deadlock detection. To this end, we present results from H.264, AES, random graph and pipeline benchmarks. We are generally interested in speedup over one CPU and in factors that cause the speedup to be less than the ideal.

5.3.1 H.264

In the first experiment, we have used the artificial H.264 encoding workload described in section 5.2.2 (see block diagram in figure 6). We have configured the benchmark to encode 30 video frames at rate of 1 frame per second (fps), with the number of workers varying from 1 to 512 in steps of powers of two. From the results in figure 10, we can observe 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.

5.3.2 AES

The AES encryption is performed in ECB mode, resulting in a typical example of an embarrassingly parallel workload, i.e., a task that can be divided into

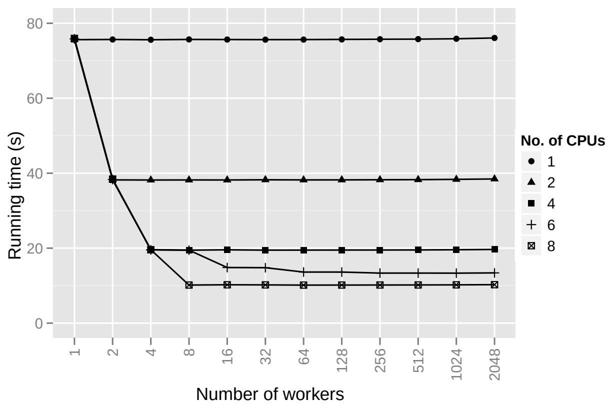


Fig. 11 AES encryption benchmark.

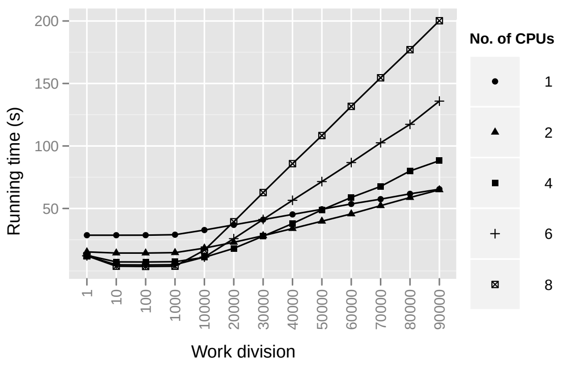
subtasks that can execute completely independently of each other.[12] The AES KPN is the same as the KPN shown in figure 4; this topology is in fact common for all embarrassingly parallel algorithms. The source P_0 hands out equally-sized memory chunks to n workers where the exchanged messages are carrying only pointer-length pairs (16 bytes in total). When a worker receives a message, it encrypts the chunk in-place with the 128-bit AES algorithm using several passes and sends the reply message back to the source KP.

In this benchmark, we have set the total block size to 2^{28} bytes (256 MB), and the total number of workers has been varied from 1 to 2048. P_0 (refer to figure 4) partitions the memory block into as many non-overlapping chunks as there are workers and sends each chunk to a worker for encryption. The number of encryption passes has been set to 40 so that the total running time is sufficiently high. The results, shown in figure 11, show perfect linear speedup with the number of CPUs (running times of 80 and 10s on 1 and 8 CPUs) , as soon as the number of workers becomes greater or equal to the number of CPUs. When $w < 8$ the speedup is limited only by the lack of work to be assigned to all available CPUs; in particular, for $w = 1$, the speedup is 1 because there is only a single worker process encrypting the whole block.

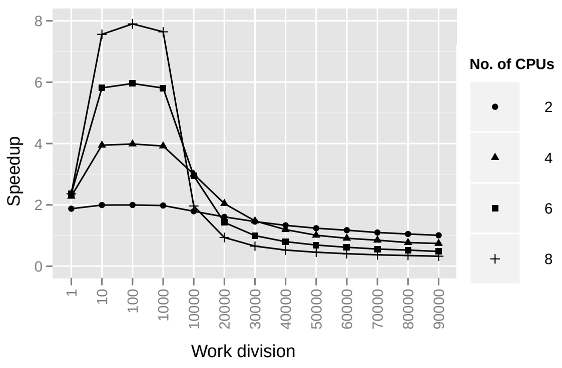
5.3.3 Random network

The scalability of Nornir using the Random benchmark described in section 5.2.2 is shown in figure 12(a). This plot shows the absolute running times of a random graph with cycles at different values of d, whereas figure 12(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 12(b) we see that $d = 100$

[12] ECB mode should never be used in practice; CTR mode is far more secure and also yields embarrassingly parallel workload.



(a) Running time.



(b) Speedup over 1 CPU.

Fig. 12 Benchmark results of a the random directed graph shown in figure 8 in 50 layers. The x-axis is not uniform.

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 13, 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 [5].

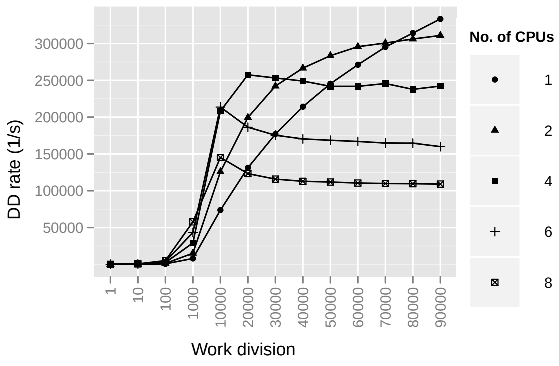


Fig. 13 Deadlock detection rate on the random graph benchmark.

5.3.4 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 noticeable slow-down (see figure 14) 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 (defined in section 4.5). The drop in performance is proportional to message size and the number of CPUs. Copying larger messages consumes more time and causes greater contention over channel locks with increasing number of CPUs.

5.4 Comparison with MapReduce

KPNs are generally more flexible than MapReduce in that more general process graphs, e.g., with branches and cycles, can be processed. However, an interesting issue is how these two models perform for applications they both can execute. To evaluate whether the KPN model has any performance advantages over the MapReduce model, we have therefore implemented the word count application, which is the canonical MapReduce example (more examples is given in [37]). We used Nornir to process a MapReduce-version of Word Count (KPN-MR) by constructing a MapReduce topology. In figure 15, KPN-MR is compared with Nornir and its more flexible process topology tailored to the problem (KPN-FLEX). We processed a data-set that is distributed with Phoenix and used for Phoenix MapReduce benchmarks [32], where the files

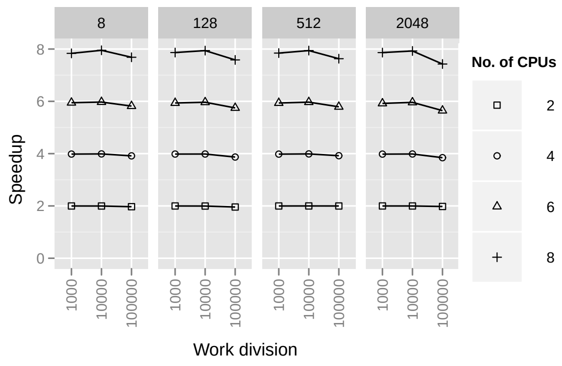


Fig. 14 Pipeline speedup for message sizes of 8,. . . ,2048 bytes and three different work divisions (d).

had the sizes 10, 50 and 100 MB. We can observe that the running time decreases as more CPUs are used and that the KPN-FLEX version has several *times* better performance than the KPN-MR version. Our results indicate that both the KPN-MR and KPN-FLEX implementations scale approximately logarithmically with the number of CPUs: the program running time decreases approximately linearly with each doubling of the number of runners. We believe that the speedup is sub-linear because of two factors: algorithmic complexity of sorting, which is $O(n \log n)$, and the effects of deadlock detection. We also observe that KPN-FLEX is consistently faster than KPN-MR, by a factor of 3 – 6.7, and the speedup is proportional with the problem size.

We have also compared the performance of our KPN-FLEX and KPN-MR implementations with Phoenix [32], which is a MapReduce implementation designed specifically for multi-core machines. We have run Phoenix with its default settings, which means that it uses as many threads in each of the Map and Reduce stages as there are CPUs on the machine. Table 1 compares the Word Count results of Phoenix, KPN-MR and KPN-FLEX. The KPN-MR program is consistently slower than Phoenix, by a factor of 1.06 on the 10MB file, and by a factor of about 2 on 50MB and 100MB files. This result is not very surprising since Phoenix is optimized for running MapReduce programs, while our KPN run-time includes supports for arbitrary directed graphs of communicating processes. However, the KPN-FLEX program, by having the ability to use data structures that are more suited to the given task (hash tables) and avoiding unnecessary work that MapReduce semantics requires (extra sort), achieves about 2.7 times better performance than Phoenix.

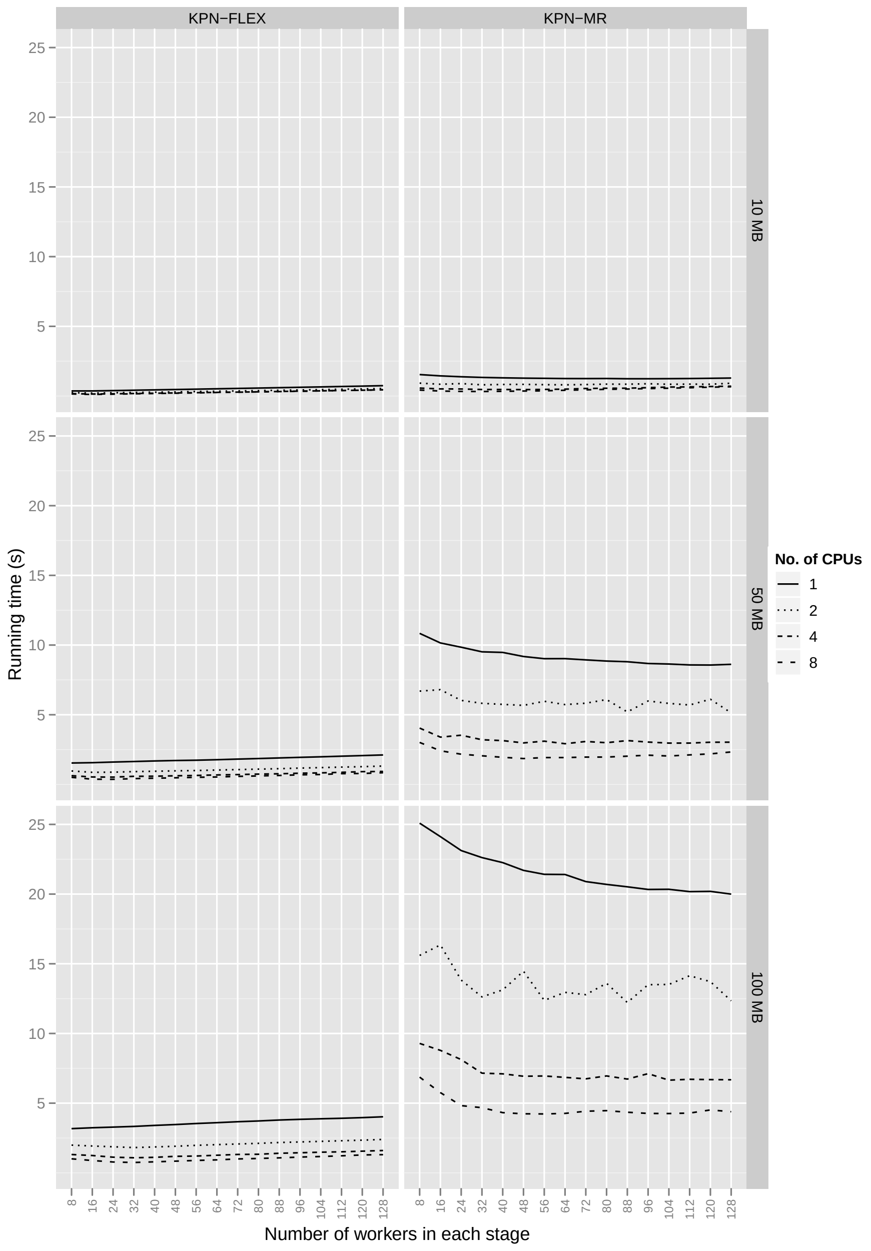


Fig. 15 Running times of the word frequency program on 10, 50 and 100 MB data-sets, using KPN-FLEX and KPN-MR solutions.

Size	Phoenix	KPN-MR		KPN-FLEX	
	t	*t*	*speedup*	*t*	*speedup*
10	0.30	0.32	0.94	0.11	2.73
50	0.98	1.86	0.53	0.37	2.65
100	2.04	4.23	0.48	0.74	2.76

Table 1 Mean running times in seconds (t) of the word frequency programs for Phoenix, KPN-MR and KPN-FLEX. *Speedup* is the speedup factor over Phoenix.

5.5 Summary

We have evaluated several aspects of Nornir: scalability of the scheduler with the number of processes and CPUs, overhead of message-copying and overhead 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 embarrassingly parallel 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 sizes 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 overhead starts 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 4.5).
- The centralized deadlock detection and resolution algorithm can cause serious scalability and performance problems for 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 cases. 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 using mutexes for protecting the scheduler's queues instead of using non-blocking queues proposed by Arora et al. [7], except in the scatter-gather benchmark. There, our WS-LAST variant largely mitigates the problem.
- Compared to MapReduce models, Nornir using KPNs is more flexible, and outperforms both applications running in Nornir following the MapReduce model and the Phoenix implementation of MapReduce [32] for multi-core machines.

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.

6 Conclusion and future work

In this paper, we have described the implementation details of Nornir, our run-time environment for executing parallel applications specified in the high-level framework of Kahn process networks. KPNs allow branches and cycles in the communication graph of the program, a feature crucial for implementing iterative algorithms such as H.264 encoding. Nornir thus *complements* existing frameworks such as MapReduce and Dryad. We have shown that even for problems without branches and cycles, the flexibility of KPNs makes it possible to design solutions that outperform the solutions cast into the MapReduce framework.

We have evaluated Nornir's efficiency with several synthetic applications (H.264 encoding, random KPN, pipeline, AES) 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 a default channel capacity of 64 bytes is too small for some applications. We have also experienced that copying semantics of message-passing have a slight, but noticeable impact on performance at message sizes of $\sim$ 2048 bytes. Furthermore, Nornir can support parallelism at fine granularity: its overhead becomes noticeable at processing time of $10\mu s$ per message, which is $\sim$ 7 times greater than the combined overhead of scheduling and message-passing.

Our study of scheduling policies has lead to two results that are significant also outside the context of KPNs. First, we have developed a simple improvement of the work-stealing algorithm that performs as well as the original algorithm, but does not exhibit pathologically bad performance with scatter-gather types of workloads. Second, we were the first to evaluate performance of unstructured workloads on multi-core machines when scheduled by a scheduler based on graph-partitioning. We have shown that graph partitioning not only performs worse than work-stealing, but also has very unpredictable running times, an aspect not discussed by Devine et al [10]. The latter finding is relevant also in the context of resource provisioning in distributed systems where the same types of algorithms are used.

The first, and most important, step in our future work is increasing Nornir's scalability by replacing a centralized deadlock detection algorithm with a distributed one [5]. This will also be the first step towards a distributed version of Nornir, executing on a cluster of machines. For building a distributed version of Nornir, we plan to use MPI to implement message-passing between KPs running on different 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 presented in [7] instead of mutexes in order to support scalability beyond 8 CPUs. We might also use a single-producer, single-consumer FIFO queue [19] to avoid yielding between en-/dequeue attempts.

Since yielding implies switching to a new control flow and a new stack, we expect that this improvement will further increase performance by reducing pressure on data and instruction caches.

References

1. PVM (Parallel Virtual Machine), Accessed August 2010. `http://www.csm.ornl.gov/pvm/`.
2. Apache Hadoop, Accessed July 2009. `http://hadoop.apache.org/`.
3. Message passing interface forum, Accessed July 2009. `http://www.mpi-forum.org/`.
4. The OpenMP API specification for parallel programming, Accessed July 2009. `http://openmp.org/wp/`.
5. Allen, G., Zucknick, P., and Evans, B. A distributed deadlock detection and resolution algorithm for process networks. *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) 2* (April 2007), II–33–II–36.
6. Armstrong, J. A history of Erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages* (New York, NY, USA, 2007), ACM, pp. 6–1–6–26.
7. Arora, N. S., Blumofe, R. D., and Plaxton, C. G. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of ACM symposium on Parallel algorithms and architectures (SPAA)* (New York, NY, USA, 1998), ACM, pp. 119–129.
8. Brooks, C., Lee, E. A., Liu, X., Neuendorffer, S., Zhao, Y., and Zheng, H. Heterogeneous concurrent modeling and design in Java (volume 1: Introduction to Ptolemy II). Tech. Rep. UCB/EECS-2008-28, EECS Department, University of California, Berkeley, Apr 2008.
9. Buhr, P. A., and Stroobosscher, R. A. The μ system: Providing light-weight concurrency on shared-memory multiprocessor computers running UNIX. *Software – Practice and Experience 20*, 9 (1990), 929–964.
10. Catalyurek, U., Boman, E., Devine, K., Bozdag, D., Heaphy, R., and Riesen, L. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)* (2007), IEEE. Also available as Sandia National Labs Tech Report SAND2006-6450C.
11. Chaiken, R., Jenkins, B., Larson, P.-Å., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow. 1*, 2 (2008), 1265–1276.
12. chih Yang, H., Dasdan, A., Hsiao, R.-L., and Parker, D. S. Map-Reduce-Merge: simplified relational data processing on large clusters. In *Proceedings of ACM international conference on Management of data (SIGMOD)* (2007), pp. 1029–1040.
13. de Kock, E., Essink, G., Smits, W. J. M., van der Wolf, R., Brunei, J.-Y., Kruijtzer, W., Lieverse, P., and K.A. Vissers, K. Yapi: application modeling for signal processing systems. *Proceedings of Design Automation Conference* (2000), 402–405.
14. de Kruijf, M., and Sankaralingam, K. MapReduce for the Cell BE architecture. *University of Wisconsin Computer Sciences Technical Report CS-TR-2007 1625* (2007).
15. Dean, J., and Ghemawat, S. System and method for efficient large-scale data processing. US Patent no. 7650331, Jan 2010.
16. Dean, J., and Ghemawat, S. MapReduce: simplified data processing on large clusters. In *Proceedings of Symposium on Opearting Systems Design & Implementation (OSDI)* (Berkeley, CA, USA, 2004), USENIX Association, pp. 10–10.
17. Gedik, B., Andrade, H., Wu, K.-L., Yu, P. S., and Doo, M. Spade: the system s declarative stream processing engine. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2008), ACM, pp. 1123–1134.
18. Geilen, M., and Basten, T. Requirements on the execution of Kahn process networks. In *Programming Languages and Systems, European Symposium on Programming (ESOP)* (2003), Springer Berlin/Heidelberg, pp. 319–334.

19. GIACOMONI, J., MOSELEY, T., AND VACHHARAJANI, M. 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, 2008), ACM, pp. 43–52.
20. GORDON, M. I., THIES, W., AND AMARASINGHE, S. 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, 2006), ACM, pp. 151–162.
21. HE, B., FANG, W., LUO, Q., GOVINDARAJU, N. K., AND WANG, T. Mars: a MapReduce framework on graphics processors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques* (New York, NY, USA, 2008), ACM, pp. 260–269.
22. HUDAK, P., HUGHES, J., JONES, S. P., AND WADLER, P. A history of Haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages* (New York, NY, USA, 2007), ACM, pp. 12–1–12–55.
23. INTEL CORPORATION. Threading building blocks. `http://www.threadingbuildingblocks.org`.
24. ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. of the ACM SIGOPS/EuroSys European Conference on Computer Systems* (New York, NY, USA, 2007), ACM, pp. 59–72.
25. KAHN, G. The semantics of a simple language for parallel programming. *Information Processing 74* (1974).
26. KNUTH, D. E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison-Wesley Professional, 1997.
27. LÄMMEL, R. Google's MapReduce programming model — revisited. *Sci. Comput. Program. 68*, 3 (2007), 208–237.
28. LEE, E.A.; PARKS, T. Dataflow process networks. *Proceedings of the IEEE 83*, 5 (May 1995), 773–801.
29. OLSON, A., AND EVANS, B. Deadlock detection for distributed process networks. In *ICASSP: Proc. of IEEE IEEE International Conference on Acoustics, Speech, and Signal Processing* (March 2005), vol. 5, pp. 73–76.
30. OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. 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, 2008), ACM, pp. 1099–1110.
31. PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program. 13*, 4 (2005), 277–298.
32. RANGER, C., RAGHURAMAN, R., PENMETSA, A., BRADSKI, G., AND KOZYRAKIS, C. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 13–24.
33. RICHARDSON, I. E. G. H.264/MPEG-4 part 10 white paper. Available online. `http://www.vcodex.com/files/h264_overview_orig.pdf`.
34. THOMPSON, M., AND PIMENTEL, A. Towards multi-application workload modeling in sesame for system-level design space exploration. In *Embedded Computer Systems: Architectures, Modeling, and Simulation* (2007), vol. 4599/2007, pp. 222–232.
35. VALVÅG, S. V., AND JOHANSEN, D. Oivos: Simple and efficient distributed data processing. In *Proceeedings of IEEE International Conference on High Performance Computing and Communications (HPCC)* (2008), pp. 113–122.
36. VALVÅG, S. V., AND JOHANSEN, D. Cogset: A unified engine for reliable storage and parallel processing. In *Proceedings of IFIP International Conference on Network and Parallel Computing Workshops (NPC)* (2009), pp. 174–181.
37. Ž. VRBA. *Implementation and performance aspects of Kahn process networks.* PhD thesis, Department of Informatics, University of Oslo, Norway, Dec. 2009. Dissertation no. 903.
38. Ž. VRBA, HALVORSEN, P., AND GRIWODZ, C. Evaluating the run-time performance of Kahn process network implementation techniques on shared-memory multiprocessors.

International Conference on Complex, Intelligent and Software Intensive Systems (CISIS) - International Workshop on Multi-Core Computing Systems (MuCoCoS) (2009), 639–644.

39. Ž. Vrba, Halvorsen, P., and Griwodz, C. A simple improvement of the work-stealing scheduling algorithm. *International Conference on Complex, Intelligent and Software Intensive Systems (CISIS) - International Workshop on Multi-Core Computing Systems (MuCoCoS)* (2010), 925–930.
40. Ž. Vrba, Halvorsen, P., Griwodz, C., and Beskow, P. Kahn process networks are a flexible alternative to MapReduce. *IEEE International Conference on High Performance Computing and Communications (HPCC)* (2009), 154–162.
41. Ž. Vrba, Halvorsen, P., Griwodz, C., Beskow, P., and Johansen, D. The Nornir run-time system for parallel programs using Kahn process networks. In *6th International Conference on Network and Parallel Computing (NPC)* (October 2009), IEEE Computer Society, pp. 1–8.