Kahn process networks are a flexible alternative to MapReduce

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

Abstract

Experience has shown that development using shared-memory concurrency, the prevalent parallel programming paradigm today, is hard and synchronization primitives nonintuitive because they are low-level and inherently non-deterministic. To help developers, we propose Kahn process networks, which are based on message-passing and shared-nothing model, as a simple and flexible tool for modeling parallel applications. We argue that they are more flexbile than MapReduce, which is widely recognized for its efficiency and simplicity. Nevertheless, Kahn process networks are equally intuitive to use, and, indeed, MapReduce is implementable as a Kahn process network. Our presented benchmarks (word count and k-means) show that a Kahn process network framework permits alternative implementations that bring significant performance advantages: the two programs run by a factor of up to ~ 2.8 (word-count) and ~ 1.8 (k-means) faster than their implementations for Phoenix, which is a MapReduce framework specifically optimized for executing on multicore machines.

1 Introduction

Developing applications that exploit multiple computing resources, be it multiprocessors, chip-level multiprocessors or co-processors is challenging. Since such systems have become a commodity, the number of developers expected to face this challenge is increasing. Implicit communication, through shared data structures, is the reigning paradigm despite the recognition that standard concurrency control mechanisms (e.g., mutexes, semaphores and condition variables) are difficult to use correctly [12]. Their non-deterministic nature also makes the learning curve steeper – for example, many newcomers to multithreaded programming are surprised to learn that mutexes or condition variables wake up threads in arbitrary instead of FIFO order. Hence, several frameworks and higher-level abstractions (designed to ease parallelization) have been proposed, such as software transactional memory (STM) [9], MapReduce [6] and Dryad [10].

Each of these proposed frameworks, however, has some drawbacks: STM has large overheads; MapReduce is very rigid (in the sense that computation steps are predetermined); Dryad supports non-deterministic constructs and disallows cycles in the communication graph and is thus, like MapReduce, unable to model iterative algorithms.

We therefore propose that Kahn process networks (KPN) [11] be used for expressing parallelism in programs. KPNs are *the least restrictive message-passing model that yields provably deterministic programs*, i.e., programs that yield always the same output given the same input, regardless of the order in which individual processes are scheduled. Determinism is achieved by placing restrictions on processes; it has been proven formally that the resulting model is no longer deterministic if one or more of these restrictions are removed. Using KPNs for development of parallel applications brings several benefits:

- *Sequential coding of individual processes.* Processes are written in the usual sequential manner; synchronization is implicit in explicitly coded communication primitives (message send and receive).
- *Composability.* Connecting the output of a network computing function $f(x)$ to the input of a network computing $g(x)$ guarantees that the result will be $g(f(x))$. Thus, components can be developed and tested individually, and later assembled together to achieve more complex tasks.
- *Reliable reproduction of faults.* Because of determinism, it is possible to *reliably* reproduce faults (otherwise notoriously difficult), which will greatly ease debugging.

While MapReduce and Dryad also have most of the above benefits, KPNs have several additional key properties that make them suitable for modeling and implementing a wider range of problems than MapReduce and Dryad:

- *Arbitrary communication graphs.* Whereas MapReduce and Dryad restrict developers to the structure of figure 1 and directed acyclic graphs (DAGs), respectively, 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 [13].

- *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.

The flexibility of KPNs allows the implementation of the full semantics of MapReduce, so they are as easy (or hard) to use as MapReduce. However, we and others [4] have noticed that the structure of the MapReduce computation is not always a good match to the task at hand, and, as we will show in this paper, can adversely impact performance. Because a single, multi-core machine has much more limited resources than a large cluster, we deem that it is important to investigate MapReduce alternatives that will be more efficient on a small scale.

To investigate whether direct KPN modeling has advantages over modeling with MapReduce, we have implemented the word count, which is the canonical MapReduce example, and k-means applications. Each application is implemented in two variants, KPN-MR and KPN-FLEX (4 programs in total). The KPN-MR variant constructs a MapReduce network topology, while the KPN-FLEX variant constructs a topology tailored to the problem. In our benchmarks, KPN-FLEX outperforms KPN-MR by a factor of up to 1.3 for the k-means program, and by a factor of up to 6.7 in the word count program. Similarly, KPN-FLEX runs faster by a factor of up to ~ 2.8 (word-count) and ~ 1.8 (k-means) than their equivalents for Phoenix [13], which is a MapReduce framework specifically optimized for executing on multicore machines.

2 Related work

MapReduce [6] is a popular framework for processing of large amounts of data on a cluster of machines. Its basic structure is a pipeline, but each stage consists of several concurrent processes (see figure 1). The splitter process (S) splits the input into roughly equal chunks and sends them to m processes in the map stage. Processes in the map stage apply a user-defined function to each record, consisting of a key and a value,[1] and send the resulting records to n processes in the reduce stage. Records with identical keys are always sent to the same reduce process, so that the final value of the reduce function for the given key can be computed without communicating with other reduce processes.

[1]The key-value paradigm stems from the MapReduce pattern; it is not necessary to use it when modeling a problem as a KPN.

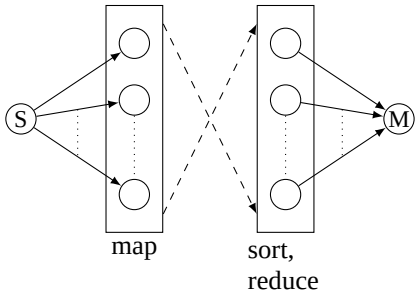

Figure 1. Structure of MapReduce computation represented as a KPN. Dahsed lines represent connections between all pairs of processes in the two stages. Merge (M) and split (S) processes can be omitted when several MapReduce computations are chained.

The reduce stage first sorts the data items by their key, and then applies another user-defined function over each group of data items with identical keys. The final stage of the computation is the merge process (M) which merges n sorted outputs from reduce processes into a single sorted output stream of data items.

Dryad [10] is a system for describing and executing, in a potentially distributed manner, computations whose communication patterns are expressed by directed *acyclic* graphs. Main features, namely message-passing and sequential programming model of individual processes, are shared with the KPN model. However, there are a number of differences. First, Dryad is not based on any formal foundation. Second, Dryad introduces asynchronous interfaces which may, as a consequence, result in nondeterministic programs (indeed, Dryad's nondeterministic merge is built upon these interfaces). Third, loops in the communication graph are not allowed, which makes it impossible to model iterative algorithms such as k-means (see section 5).

While there are other parallel programming models, due to space constraints we have been able to describe only MapReduce and Dryad in detail. Other models cover shared-memory models, pipelines, and unstructured point-to-point communication, as available through the MPI programming interface. In this taxonomy, while (parallel) pipelines and DAGs have limited expressiveness, i.e., do not allow cycles in the graph. This is possible with unstructured communication, but inadvertent problems – such as non-determinism and deadlocks – occur easily. In this taxonomy, KPNs are the most flexible model which still guarantees determinism, which is the reason for our deeper investigation of their properties.

3 Kahn process networks

A KPN [11] has a simple representation in the form of a *directed graph* with *processes* as nodes and *communication*

channels as edges (see Section 5 for examples). A process encapsulates data and a single, sequential control flow, independent of any other process. Processes are not allowed to share data and may communicate only by sending messages over channels. Channels are *infinite* FIFO queues that store discrete *messages*. Channels have *exactly one* sender and one receiver process on each end (1:1), and every process can have multiple *input* and *output* channels. Sending a message to the channel always succeeds, but trying to receive a message from an empty channel *blocks* the process until a message becomes available. It is not allowed to poll a channel for presence of data. These properties fully define the *operational semantics* of KPNs and make the Kahn model *deterministic*, i.e., the history of messages produced on the channels does not depend on the order in which the processes are executed, provided the scheduling is *fair*, i.e., that execution of a ready process will not be indefinitely postponed.

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 construction of KPNs which need unbounded space for their execution, but 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, but which would continue running in the theoretical model. The algorithm of Geilen and Basten [7] resolves the deadlock by traversing the cycle to find the channel of least capacity and enlarging it by one message, thus resolving the deadlock.

4 KPN implementation

Our KPN execution environment is implemented in C++, and it runs on Windows and POSIX operating systems (Solaris, Linux, etc.) Our implementation[2] of the run-time environment for executing KPNs consists of a Kahn process (KP) scheduler, message transport and deadlock detection and resolution algorithms.

To the best of our knowledge, there exist only two other general-purpose KPN runtime implementations: YAPI [5] and Ptolemy II [3]. 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. 40 kB in our implementation), and we were unable to make it use multiple CPUs.

[2]Available code: `http://simula.no/research/networks/software`

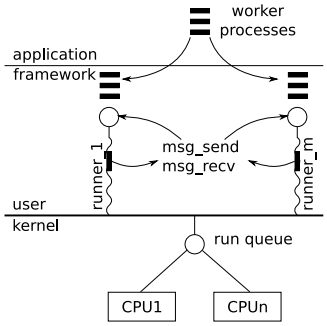

Figure 2. Two-level KPN scheduling

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-switch.

4.1 Process scheduler

The scheduler may be configured, at compile-time, in two ways. In the first configuration, each KP is run in its own OS-thread. Channels are protected by blocking mutexes, and notifications are done by using condition variables. However, our earlier measurements have shown that the native mechanisms suffer from high overheads, so we have also implemented an optimized KP scheduler.

The second configuration uses our own work-stealing scheduler. When the KPN is started, m runner threads are created and scheduled by the OS onto n available CPUs (see figure 2). Each runner implements a work-stealing policy [2], i.e., it has a private run queue of ready KPs, and 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 [2]; instead we use ordinary mutexes. Context-switch between KPs is implemented in user-mode. On Solaris and Linux running on AMD64 architecture we employ hand-crafted assembly code; on other platforms we use OS-provided facilities, which often incur some additional overhead.

The channels are protected with *polling* mutexes: if a KP cannot obtain the channel's lock, it will spin, explicitly yielding to the scheduler between iterations, until it has finally obtained the lock. Waiting and signaling are implemented by a protocol between KPs and the scheduler. The protocol is optimized for the case of having at most one sleeping KP, which is possible because channels are 1:1, and at most one process can ever be blocked on a channel.

4.2 Message transport

Channels have a two-fold role: to transport messages and to interact with the scheduler, i.e., block and unblock pro-

cesses on either side of the channel. Message send/receive is implemented by copying the messages to/from channel buffers. Zero-copy transfer would require dynamic memory allocation, and we have measured that copying is less expensive as long as messages are smaller than ~ 256 bytes.

We have also extended channels with *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, but the next receive will immediately return false instaed of blocking. A further attempt to receive a message from the channel will permanently block the process.

An alternative approach to EOF signaling is sending a message with specific contents as the last message on the channel. The disadvantage is that all values of the channel's type (e.g., `int`) might be meaningful in a given context, so no value could be used to encode the EOF value. In such cases, one would be forced to use more cumbersome solutions that also potentially impose additional overhead.

4.3 Deadlock detection and resolution

Deadlock detection and resolution is a mechanism which allows execution of KPNs in finite space. Since communication is 1:1, every cycle of blocked KPs is a ring; a property which greatly simplifies detection. Whenever the currently running KP would block on send, the algorithm is invoked to check whether an artificial deadlock occurred. If no cycle is found, the current KP is blocked and this fact is recorded in the blocking graph data structure. Otherwise, the capacity of the smallest channel in the cycle is increased by one, as suggested by [7]. If the channel belongs to the current KP, the current KP is immediately resumed; else the KP on the channel's send side is unblocked, and the current KP is blocked. Similarly, receiving from a full channel unblocks the KP on the sending side and removes an edge from the blocking graph.

Our current implementation uses a centralized data structure for the blocking graph, so the above operations must run while holding a single global mutex. Despite this, we have not noticed significant scalability issues on up to 8 CPUs on the workloads described in this paper. This is because of the interaction of the following elements:

- KPs that do not need to block or wake up another KP continue to run undisturbed.
- Since the number of KPs is usually much larger than the number of runners, it is to be expected that many KPs will be ready and little work will be stalled.
- The time the global mutex is held for in the worst case is proportional to the size of the largest potential cycle, which is small in our examples.

5 Modeling with KPNs

Parallelizing an application with KPNs (as well as with Dryad and MapReduce) entails three steps:

1. Identifying independent subtasks (components) and their corresponding inputs and outputs.
2. Implementing subtasks, possibly by "filling in the blanks" in off-the-shelf components (e.g., Reduce and n-way merge).
3. Determining the number of subtasks and communication between them; the latter will be largely dependendent on the previous step.

We demonstrate this methodology on the word count and k-means programs. Even though word count is the "canonical" MapReduce example, its implementation via MapReduce is rather inefficient, in terms of the amount of unnecessary extra work done. k-means is an example of an *iterative* data-parallel algorithm, and is because of that a rather bad match for MapReduce, as is also noted in [13].

5.1 Word count

The word count program counts the number of occurrences of each word in a given text and outputs them sorted in the order of decreasing frequency.

The MapReduce and KPN-MR solutions require *two* MapReduce instantiations connected in series, such that the output of the Reduce stage is sent directly to the input of the Map stage of the second instantiation, without an intervening merge stage. Schematically, the network looks like this (note the pipeline structure):

$$\mathrm{S} \rightarrow \mathrm{MR}_1 \rightarrow \mathrm{MR}_2 \rightarrow \mathrm{M}$$

where each MapReduce has a structure corresponding to the one shown in figure 1.

The output of MR_1 is a list of word-count pairs sorted alphabetically by word. MR_2 then reverses word-count pairs to count-word pairs, count now being the key, in order to sort them by frequency. The reduce function of MR_2 is identity and the only role of this stage is sorting.

The subtasks of our KPN-FLEX solution (see Figure 4) are operationally identical to those of the KPN-MR solution: splitting the (memory-mapped, copy-on-write) input file into chunks, counting word frequency in individual chunks, summing and sorting partial word frequencies of chunks, and finally merging partial counts into a single sorted list.

The code that implements the count stage is shown in Figure 3; note that this is *real code*, taken from a working program (see appendix of [6] for Google's implementation). Kahn processes are in our framework implemented

```
1  void count::parse_word(text_chunk &chunk)
2  {
3    char *b = get<0>(chunk), *e = get<1>(chunk);
4    char *w;
5
6    // Skip leading non-letters.
7    while((b < e) && !inword(*b)) ++b;
8
9
10   // Parse word and convert to uppercase
11   for(w = b; (w < e) && inword(*w); ++w)
12     *w -= ((*w >= 'a') && (*w <= 'z'))*('a'-'A');
13
14   // Insert new word, or increase count
15   if(b != w) {
16     std::pair<count_hash::iterator,bool> rv =
17       counts_.insert(count_hash::value_type(
18         word(b, w), 1));
19     if(!rv.second)
20       ++rv.first->second;
21   }
22   get<0>(chunk) = w;
23 }
24
25 void count::behavior()
26 {
27   text_chunk chunk;
28   boost::hash<word> h;
29   count_hash::iterator it;
30
31   in(0).recv(chunk);
32   while(get<0>(chunk) != get<1>(chunk))
33     parse_word(chunk);
34   for(it = counts_.begin();
35     it != counts_.end(); ++it)
36       out(h(it->first) % out_count()).send(it);
37   eof_all();
38 }
```

Figure 3. C++ code for the count stage of the word count program in KPN-FLEX implementation (see also figure 4). The code in slanted font (lines 31 and 34–37) are the only necessary additions to the sequential version of the algorithm.

as classes which must derive from the `actor` class and implement the pure virtual `behavior` method, which is the entry point.

The `count` process initially receives a text chunk which is represented as a (start, end) tuple of pointers. The following `while` counts individual words in the text chunk: it keeps calling the `parse_word` routine which extracts the next word from the chunk and updates the hash table of counts. When all words from the chunk have been extracted, the hash table is iterated and iterators to each (word, count) pair are sent to the sum process. To distribute the load approximately evenly, the destination output is calculated by taking the hash of each word modulo the number of outputs. Finally, the `eof_all()` statement sets an EOF indication on all output channels.

5.2 k-means

k-means is an iterative algorithm used for partitioning a given set of points in multidimensional space into k groups;

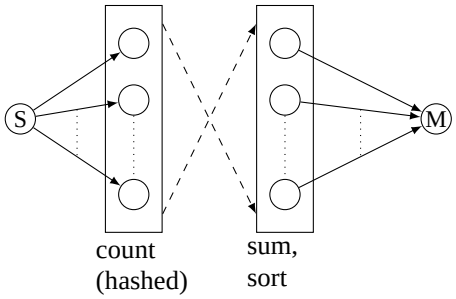

Figure 4. KPN-FLEX for solving the word count problem.

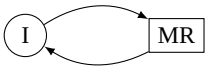

Figure 5. KPN-MR for solving the k-means problem. We had to introduce additional process (I) to iterate the MR block, whose constituent parts are shown in Figure 1.

it is used in data mining and pattern recognition. The algorithm consists of the following steps:

1. Make initial guess for the center of gravity (centroid) of each of the k groups.
2. Assign each point from the dataset to the group which centroid is closest to the point.
3. Recalculate new centroids based on the new assignment of points to clusters.
4. Repeat from step 2, now with new centroid coordinates, until the process converges (i.e., the centroids do not move by more than a preset threshold).

In the KPN-MR implementation, the Map stage computes the index i of the nearest mean for each point p and emits (i, p) as the intermediate key-value pair. The Reduce stage computes the new mean values from the itermediate key-value pairs. Since MapReduce alone cannot model iterative algorithms, we construct the KPN shown in Figure 5. The MapReduce implementation is optimized in the same way as for word count: the Map stage and Reduce stage send out pointers to vectors instead of individual points.

KPN-FLEX that executes the k-means algorithm is shown in Figure 7. The I (iterate) process has two phases: initialization and iteration. The initialization phase generates a vector of random points, selects new k random points as the starting centroids, and partitions the point vector into as many approximately equal parts as there are worker processes (unlabeled in the figure). Then, it sends a partition of the point set to each worker as a pair of (start, end) iterators, and sends all k centroids, each in own message, to every worker. When the initialization phase has finished, the iteration phase, described below, begins.

A worker receives on its input the current location of the means and computes the new cluster assignment for its part of the point set; each time a point is assigned to a new cluster, the centroid corresponding to the partial point set assigned to the worker is updated. When a worker has finished with its points, it sends partial sums for centroids to the iteration process (I). Since mean is a linear operation, the iteration process can take the partial sums and counts to compute the new locations of the centroids. If the new locations are equal to the locations from the previous iteration, the process has converged and the program terminates.

The full code executed by the worker process is shown in Figure 6. The additional code necessary to transform the sequential algorithm into a Kahn process is shown in slanted text. Also note how the code in slanted font resembles programming with *ordinary files*: a process reads its input, item by item, from input ports (files opened for reading), does some processing, writes results to its output ports (files opened for writing), and exits when EOF is detected on some input port (file).

```
1  void kmeans::behavior()
2  {
3    size_t i;
4
5    // Receive the point set.
6    in1.recv(points_);
7
8    while(1) {
9      point_vec partial_sums(cur_means_.size(),
10       ZEROPOINT);
11
12     // New iteration: get recalculated means
13     for(i = 0; i < cur_means_.size(); ++i)
14       if(!in2.recv(cur_means_[i]))
15         return;
16
17     point_vec::iterator it;
18     for(it = get<0>(points_);
19       it != get<1>(points_); ++it)
20     {
21       // Recompute point's cluster.
22       int c = std::min_element(
23           cur_means_.begin(), cur_means_.end(),
24           bind(&distance, *it, _1) <
25           bind(&distance, *it, _2))
26         - cur_means_.begin();
27
28       // Assign point to the new cluster and
29       // update cluster's partial sum
30       it->c = c;
31       for(int k = 0; k < DIM; ++k)
32         partial_sums[c].v[k] += it->v[k];
33       ++partial_sums[c].c;
34     }
35
36     // Send partial sums to the I process.
37     for(i = 0; i < partial_sums.size(); ++i)
38       out.send(partial_sums[i]);
39   }
40 }
```

Figure 6. C++ code for the worker processes of the k-means program in KPN-FLEX implementation. The code lines typeset in slanted font (lines 13–15, 37–38) are the only necessary additions to the sequential algorithm.

6 Evaluation

To evaluate advantages of modeling with KPNs, we have implemented the word count and k-means programs as KPN-FLEX and KPN-MR networks (see Section 5). All test programs have been compiled as 64-bit with GCC 4.3.2 and maximum optimizations (`-m64 -O3 -march=opteron`). 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 program has been run 10 times on 1, 2, 4 and 8 CPUs with differing number of worker processes. We present the average *wall-clock (real) time* of 10 runs together with error lines showing the standard deviation. Note that we compare the *unoptimized* (in terms of the number of exchanged messages) KPN implementations with the *optimized* MapReduce (over KPN) implementations.

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

6.1 Implementation considerations

Our *unoptimized* MapReduce implementation of the word count program was slower by an order of magnitude than the optimized implementation, for which we present the results. In the unoptimized case, each key-value pair was sent in its own message, and the number of sent messages was dominated by the *total* number of words in the input file. The *optimized* KPN-MR implementation allocates as many vectors holding key-value pairs as there are outputs from the Map and Reduce stages. Individual pairs are distributed across the vectors, taking care that pairs with the same key are placed in the same vector. Afterwards, a *single* message with a pointer to the vector is sent to each output. We could have optimized the KPN-FLEX implementations in a similar way, but we decided against it because it would somewhat obscure similarity with the sequential algorithm.

We have furthermore optimized the KPN-MR k-means implementation by introducing the extra process that iterates the MapReduce block (see Figure 5), thus avoiding KPN startup and shutdown costs which Phoenix suffers in each iteration.

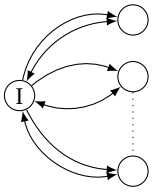

Figure 7. KPN-FLEX for solving the k-means problem. Edges with double arrows represent two channels, one in each direction. "Parallel" channels are carrying messages of different types.

6.2 Word count

We have run the word count program on files of size 10, 50 and 100MB, which is also the dataset used to benchmark Phoenix. Both implementations have been run on 1, 2, 4 and 8 CPUs, and with the number of runners in each stage varying from 8 to 128 in steps of 8. Figure 8 shows running time of the KPN-FLEX solution for the three datasets in the left column, and of KPN-MR solution in the right column; our findings are also summarized in Table 1.

We can see 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. The variation in running times is rather small between experiments, except for the KPN-MR solution with 2 runner threads; as of now we cannot explain this variation. With respect to the number of workers, the situation is more complicated: the running time *decreases* as the number of workers *increases* up to a certain point. Word count is a communication-intensive task and having more workers improves performance by more evenly distributing the load among CPUs and reducing contention in the work stealing scheduler: the more processes there are in the system, the smaller probability that a runner will need to steal a process from another runner.

Furthermore, as described in Section 4, processes acquire locks by busy-waiting and yielding between successive attempts. When there are enough ready processes, waiting on a lock will yield to another ready process which will be able to perform some useful work. When the number of processes increases even more, the performance starts dropping again for two reasons: 1) context-switch overheads (which also increase pressure on CPU caches), and 2) we conjecture that an even more important factor is contention over the single deadlock detection lock.

Table 1 gives an insight into scalability of KPN-MR and KPN-FLEX implementations with respect to the number of CPUs, which turns out to be approximately *logarithmic*: the running time decreases *linearly* with each *doubling* of the number of runner threads. We see also that KPN-FLEX is consistently faster than KPN-MR, by a factor of 3 – 6.7, and that the speedup is *proportional with the problem size*.

Size	KPN-MR			KPN-FLEX			
	n	t	α	n	t	α	f
10/1	96	1.24	1.00	8	0.36	1.00	3.41
10/2	64	0.80	1.55	16	0.21	1.71	3.80
10/4	48	0.46	2.70	16	0.14	2.57	3.24
10/8	32	0.32	3.88	16	0.11	3.27	2.95
50/1	120	8.57	1.00	8	1.54	1.00	5.57
50/2	128	5.14	1.67	16	0.87	1.77	5.91
50/4	64	2.92	2.93	24	0.52	2.96	5.57
50/8	48	1.86	4.61	24	0.37	4.16	5.06
100/1	128	20.00	1.00	8	3.17	1.00	6.30
100/2	88	12.22	1.64	32	1.81	1.75	6.74
100/4	104	6.65	3.01	32	1.08	2.94	6.14
100/8	56	4.23	4.73	32	0.74	4.28	5.69

Table 1. Word count experiment summary: number of workers n that achieves the best running time t (in seconds), relative speedup over one CPU (α) and speedup factor of KPN-FLEX over KPN-MR (f).

	Points	Groups	Iterations
A	100000	100	97
B	200000	50	140
C	200000	100	139

Table 2. k-means problem sizes and number of iterations until algorithm converges. The amount of work performed in each iteration is proportional to the product of the number of points and groups.

6.3 k-means

The k-means program generates a number of 3-dimensional points with random integer coordinates from the cube $[0, 1000)^3$. The random number generator is always initialized with the same seed, so each run executes the same number of iterations. We have measured performance of the program on three problem instances differing in the number of points and groups (see Table 2) and on 1, 2, 4, and 8 CPUs. Since this is a CPU-intensive benchmark with little communication, we have set the number of workers to be equal to the number of runner threads; very little experimentation was needed to establish that this was the optimal choice.

Again, the KPN-FLEX solution outperforms the KPN-MR solution, although the speedup is not as drastic as in the word count example. The main reason for this is that, unlike with word count, very little unnecessary work is performed by KPN-MR, namely only sorting before comput-

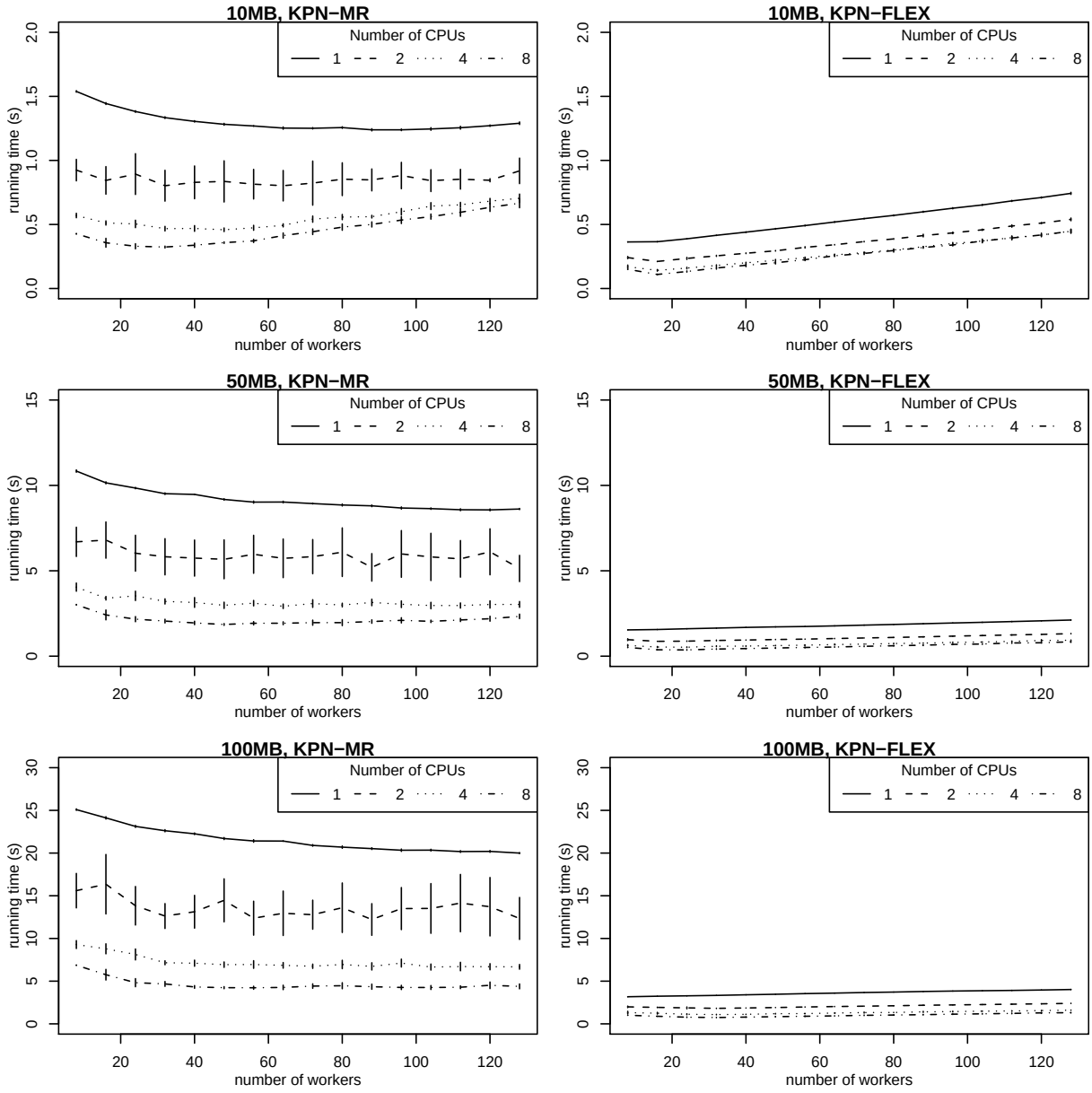

Figure 8. Running times for the word count program run on three different input files (10, 50 and 100 MB), implemented as KPN-FLEX and KPN-MR; vertical lines represent standard deviation. The number of workers for the KPN-MR version is *per stage*, so there are four times as many processes in total (two MR blocks, each consisting of two stages, each stage having the same number of workers).

ing the new means for the next iteration. Table 3 shows speedup of the KPN-FLEX solution over KPN-MR. We see that the speedup is ~ 1.2 (20%) for the benchmarks with 100 groups, and ~ 1.4 (40%) for the benchmark B with 50 groups. Since KPN-MR is recalculating means in parallel, this leads us to believe that the single I process which recalculates new means is the bottleneck in the KPN-FLEX solution. Nevertheless, unlike the word count example, the k-means program shows *almost perfect linear scalability* with the number of CPUs.

6.4 Comparison with Phoenix

We have also benchmarked our KPN-FLEX and KPN-MR against Phoenix [13], which is a MapReduce implementation designed specifically for multicore machines.

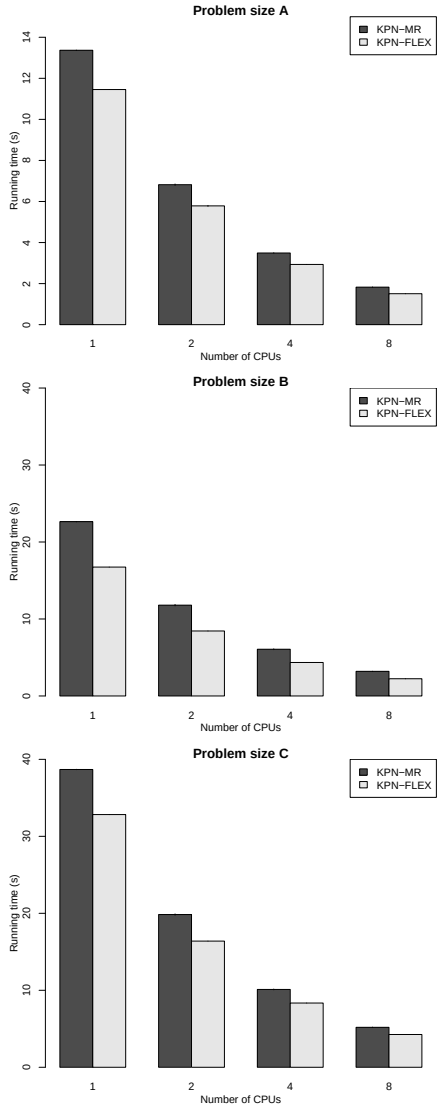

Figure 9. Benchmark results for the k-means program run on three different problem sizes, implemented as KPN-MR and KPN-FLEX. Vertical lines (barely visible) represent standard deviation. KPN-FLEX solution has as many worker process as the number of runners that were used in the benchmark. Similarly, the KPN-MR solution has the same number of workers as CPUs in each stage.

Phoenix is implemented with pthreads and uses by default as many threads in each of the Map and Reduce stages as there are CPUs on the machine. We had to compile Phoenix programs in 32-bit mode because their code uses non-portable casts between pointers and integers, which causes crashes in 64-bit mode. This is certainly to Phoenix's advantage because it allocates arrays of pointers, which would take twice as much space in the 64-bit mode, which causes fewer cache and TLB misses than in 64-bit mode.

Size	KPN-MR		KPN-FLEX		
	t	α	t	α	f
A/1	13.37	1.00	11.45	1.00	1.17
A/2	6.82	1.96	5.79	1.98	1.18
A/4	3.49	3.83	2.93	3.91	1.19
A/8	1.83	7.31	1.51	7.58	1.21
B/1	22.64	1.00	16.74	1.00	1.35
B/2	11.80	1.92	8.44	1.98	1.40
B/4	6.07	3.73	4.35	3.85	1.40
B/8	3.20	7.08	2.24	7.47	1.43
C/1	38.68	1.00	32.83	1.00	1.18
C/2	19.84	1.95	16.39	2.00	1.21
C/4	10.11	3.83	8.35	3.93	1.21
C/8	5.19	7.45	4.26	7.71	1.22

Table 3. k-means experiment summary: mean running time (t), relative speedup over one CPU (α) and and speedup factor of KPN-FLEX over KPN-MR (f).

Size	Phoenix	KPN-MR		KPN-FLEX	
	t (σ)	t	f	t	f
Word Count					
10	0.30 (0.02)	0.32	0.94	0.11	2.73
50	0.98 (0.02)	1.86	0.53	0.37	2.65
100	2.04 (0.05)	4.23	0.48	0.74	2.76
k-Means					
A	2.39 (0.05)	1.83	1.31	1.51	1.58
B	3.92 (0.21)	3.20	1.23	2.24	1.75
C	5.43 (0.22)	5.19	1.05	4.26	1.28

Table 4. Mean running times in seconds (t) of the word count and k-means programs for Phoenix, KPN-MR and KPN-FLEX; in case of KPNs, the smallest time on 8 CPUs is shown. σ is standard deviaton and f is speedup factor over Phoenix.

Table 4 shows Phoenix, KPN-MR, and KPN-FLEX running times for word count and k-means programs on all problem sizes. All Phoenix experiments have been run on the same machine 10 times, and the mean and standard deviation has been calculated.

In the word count benchmark, the KPN-MR program is consistently slower than Phoenix, by a factor of 1.06 on the 10MB file, and by a factor of $\sim$ 2 on 50MB and 100MB files. This result is not very surprising since Phoenix is optimized for running MapReduce programs, while our KPN runtime assumes nothing about the network topology. 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 2.6 times better performance than Phoenix.

Somewhat more surprisingly, both our KPN-MR and KPN-FLEX solutions outperform Phoenix on the k-means benchmark, by factors of 1.05 – 1.3 and 1.28 – 1.75, respectively. The KPN-FLEX solution performs better because it

avoids doing unnecessary work. However, we are as of yet unsure about the reasons for the unexpectedly good performance of the KPN-MR solution. We believe that the main reason for good performance of the KPN-MR solution is the elimination of framework (de)initialization that Phoenix has to perform on each iteration.

7 Conclusion and future work

In this paper we have reviewed in detail MapReduce and Dryad frameworks and KPNs. We have identified KPNs as the most flexible abstraction that presents a sequential programming model, while still ensuring deterministic execution of programs. We have also demonstrated that the MapReduce paradigm, widely recognized for its simplicity, is only a specially crafted KPN with fixed communication patterns.

To investigate advantages of KPN models, we have implemented the word count and k-means programs using two KPN topologies: one closely implementing the semantics of MapReduce (KPN-MR), and another which is specially crafted to the problem at hand (KPN-FLEX). In our benchmarks, on a 64-bit 8-core machine, KPN-FLEX implementation always outperforms the KPN-MR implementation (up to a factor of 1.3 for the k-means program and up to a factor of 6.7 for the word count program). Compared with Phoenix [13], KPN-FLEX implementations of word count and k-means programs are faster by a factor of up to 2.7 and 1.7. KPN-MR implementation of the word count program is about 2 times slower than Phoenix, but the KPN-MR implementation of the k-means program is, surprisingly, up to a factor of 1.3 faster than Phoenix.

Our future work includes performing more extensive tests to better understand performance characteristics of KPNs, especially in relation to the total number of processes and increasing scalability on machines with many CPUs. Work on scalability can be done by experimenting in three areas: using lock-free queues for channel communication (e.g., the one described in [8]), evaluating trade-offs between using lock-free queues or locks with exponential backoff in the work stealing scheduler, as well as implementing a distributed (as opposed to the current centralized) deadlock detection and resolution algorithm [1].

Acknowledgments

We thank to Håvard Espeland for constructive discussions during writing of this paper.

References

[1] 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)*, 2:II–33–II–36, April 2007.

[2] 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)*, pages 119–129, New York, NY, USA, 1998. ACM.

[3] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii). Technical Report UCB/EECS-2008-28, EECS Department, University of California, Berkeley, Apr 2008.

[4] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of ACM international conference on Management of data (SIGMOD)*, pages 1029–1040, New York, NY, USA, 2007. ACM.

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

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

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

[8] 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*, pages 43–52, New York, NY, USA, 2008. ACM.

[9] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP: Proceedings of the ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[10] 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*, pages 59–72, New York, NY, USA, 2007. ACM.

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

[12] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[13] 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)*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.