A parallel runtime system for Kahn process networks

Nhat Minh Lê Adrien Guatto Robin Morisset Albert Cohen

INRIA and ENS Paris

Libkpn

- A lean and portable C library, written in C11
- Designed for modern multicore architectures (relaxed memory model)
- Hand-written proofs for important parts (C11 formal memory model)
- For parallel algorithms
- And it's fast!

The plan

1. A programmer's introduction to libkpn
2. Inside libkpn
3. Comparisons and applications

A programmer's introduction to libkpn

Kahn process networks with bounded channels

Simple deterministic concurrent programming model

- Sequential processes
- Single-producer single-consumer FIFO communication
- Blocking reads (pop) from channels
- Blocking writes (push) because of bounded channels

KPN runtime

Lots of prior work on compilation of synchronous languages

- Such as Lucid Synchrone
- With semantics based on KPNs
- Compiled down to more general-purposes languages (e.g., C)

Libkpn is different

- KPNs with processes written directly in C
- And channels as C objects and functions

Why not use system processes with pipes?
Why not threads with shared-memory concurrent queues?

Why a runtime system?

Designed for performance and low overhead
Libkpn is a specialized scheduling system

- Cooperative (Kahn processes cannot be forcefully preempted)
- No isolation nor protection (shared memory and address space)
- Limited fairness
- No support for communication other than channels

In particular, does not need to support:

- Multi-channel *select*
- Disk and network I/O
- Locks
- Busy waiting (spin locks)

When to use libkpn

Performance and low overhead for parallel algorithms

- Written as KPNs in libkpn
- Called from main C (or C-compatible) application
- Executed on dedicated libkpn threads

Main application (sequential or threaded) can offload heavy parallel calculations to the KPN runtime

Example use cases

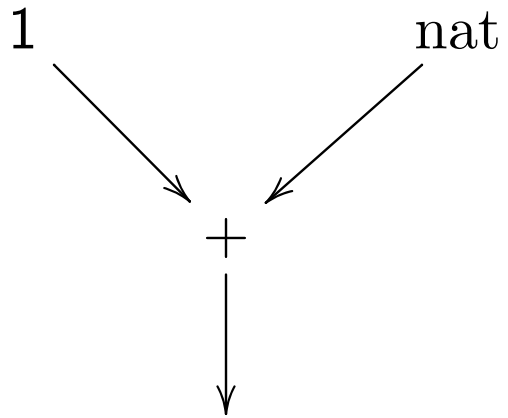
Examples of things we've used libkpn for:

- Linear algebra (we have dense matrix factorization competitive with LAPACK and Intel MKL)
- Error-correcting codes (we have a simple LDPC decoder)
- Signal processing (e.g., à la StreamIt)
- Work-in-progress: stencils, XML queries, ...

Basic terminology

- A *Kahn process* is basically a sequence of instructions that may operate on channels (a special kind of lightweight thread, if you will)
- A *scheduler thread* is a C thread (e.g., pthread) managed by libkpn that schedules and executes Kahn processes
- A *user thread* is any other C thread, not managed by libkpn
- *Runtime code* is everything that is part of libkpn and not written by the user

A simple Kahn process network



1	1	1	1	1	...
nat	0	1	2	3	...
+	1	2	3	4	...

- 3 Kahn processes: 1, nat, and +
- 3 channels: $1 \to +$, $\mathrm{nat} \to +$, and the final output from +

The 1 process in libkpn

```
#define BUF_SIZE (16 * sizeof(int))

void one_proc(void *kp) {
  for (;;)
    kpn_push(kp, 0, BUF_SIZE, &(int){ 1 }, sizeof 1);
}
```

- `one_proc` is called by the scheduler
- `kp` is a handle representing the Kahn process
- `kpn_push(kp, 0, BUF_SIZE, &(int){ 1 }, sizeof 1)`
  memcpy `sizeof 1` bytes pointed to by `&(int){ 1 }`
  into the 0-th output channel of Kahn process `kp`
  which is a bounded ring buffer of capacity `BUF_SIZE`
- `kpn_push` aborts and *longjmp* back to the scheduler if the channel is full

The nat process in libkpn

```
void nat_proc(void *kp) {
  struct nat_mem *m = kp;
  for (;; ++m->i)
    kpn_push(kp, 0, BUF_SIZE, &m->i, sizeof m->i);
}
```

- A Kahn process is sequential and stateful
- `nat_proc` may be called multiple times
- `nat_proc` may only be interrupted by *longjmp* from an aborted `kpn_push`
- `m` is a pointer to the user-defined state of the Kahn process identified by `kp`
- `m->i` is a Kahn process-local variable whose value persists between calls to `nat_proc`

The + process in libkpn (1)

```
void plus_proc(void *kp) {
  struct plus_mem *m = kp;
  BEGIN(m);
  for (;;) {
    CHECKPOINT(m, 1);
    kpn_pop(kp, 0, BUF_SIZE, &m->arg0, sizeof m->arg0);
    CHECKPOINT(m, 2);
    kpn_pop(kp, 1, BUF_SIZE, &m->arg1, sizeof m->arg1);
    CHECKPOINT(m, 3);
    int sum = m->arg0 + m->arg1;
    kpn_push(kp, 0, BUF_SIZE, &sum, sizeof sum);
  }
  END(m);
}
```

The + process in libkpn (2)

- `kpn_pop(kp, 0, BUF_SIZE, &m->arg0,sizeof m->arg0)`
  memcpy `sizeof m->arg0` bytes to location `&m->arg0`
  from the `0`-th input channel of Kahn process `kp`
  which is a bounded ring buffer of capacity `BUF_SIZE`
- `BEGIN`, `END` and `CHECKPOINT` macros provide poor man's coroutine support through Duff's device
- When embedded in a conventional C application, the final output could be displayed with `printf` or written to shared memory instead

Building the process network (1)

```
void simple_kpn(void) {
  KPN_TASK *one, *nat, *plus;
  /* Create Kahn processes. */
  one = kpn_spawn(one_proc, sizeof(KPN_TASK));
  nat = kpn_spawn(nat_proc, sizeof(struct nat_mem));
  plus = kpn_spawn(plus_proc, sizeof(struct plus_mem));
  /* Add channels. */
  kpn_pipe(one, plus, BUF_SIZE);
  kpn_pipe(nat, plus, BUF_SIZE);
  kpn_pipe(plus, NULL, BUF_SIZE);
  /* Schedule Kahn processes. */
  kpn_post(one);
  kpn_post(nat);
  kpn_post(plus);
}
```

Building the process network (2)

Slightly simplified API calls (normal versions have more optional parameters); error handling omitted

- `kpn_spawn(f, n)` creates a Kahn process running function `f` with a state structure of size `n >= sizeof(KPN_TASK)`
- `kpn_pipe(producer, consumer, n)` creates a channel of capacity `n` bytes from Kahn process `producer` to Kahn process `consumer`
- `kpn_post(kp)` marks Kahn process `kp` as fully initialized and ready for scheduling by the runtime

Standalone run

```
int main(void) {
  kpn_init(0, KPN_THREADING_STANDALONE);
  kpn_start_threads(0);
  simple_kpn();
  kpn_stop_threads(0);
  kpn_quit();
}
```

- `kpn_init(n, KPN_THREADING_STANDALONE)` starts `n` (0: auto) scheduler threads on stand by
- `kpn_start_threads(0)` instructs all scheduler threads to start accepting work; they run until all Kahn processes have terminated (their functions have returned)
- `kpn_stop_threads(0)` joins all scheduler threads
- `kpn_quit()` frees all resources allocated for the runtime

Inside libkpn

What does libkpn do?

Similar to the process and scheduling subsystem in an OS

- Execute Kahn processes on available scheduler threads . . .
- . . . with load-balancing
- Manage ready/waiting Kahn processes
- Provide efficient scheduler-aware channel operations for communication

Runtime code lives in:

- The scheduling loop, which fetches and executes work
- Runtime functions called from Kahn process procedures (including channel operations such as push and pop)

Scheduling loop

Each scheduler thread runs a scheduling loop

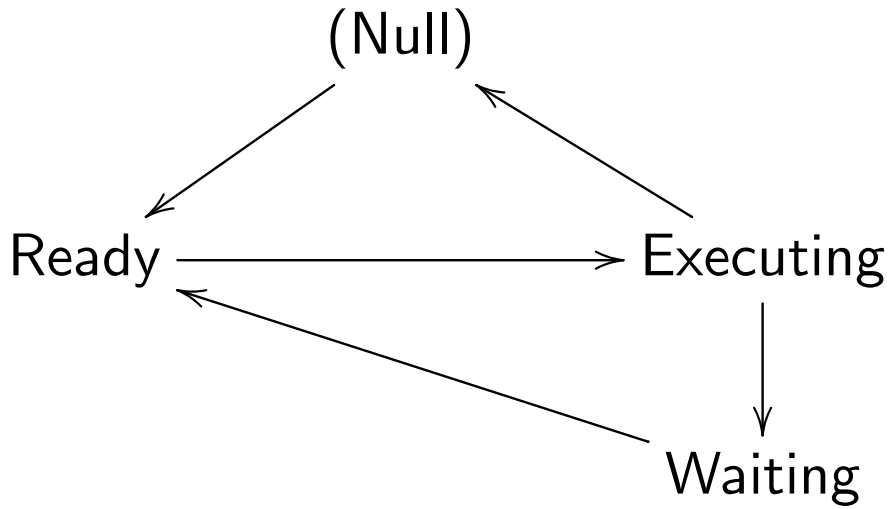
```
void schedule(int thread) {
  KPN_TASK *kp;
  while ((kp = fetch(thread)) != NULL) {
    kp->thread = thread;
    kp->proc(kp);
  }
}
```

- `thread` identifies the thread among KPN scheduler threads
- `fetch(thread)` returns a Kahn process to be executed on KPN scheduler thread `thread`
  or `NULL` if `kpn_quit()` has been called and all Kahn processes have terminated
- `kp->proc(kp)` calls the Kahn process procedure (e.g., `plus_proc`)

Process states

The runtime system manages Kahn processes
Once created, each Kahn process is either

- Currently executing on a scheduler thread
- Not running but ready to be executed
- Or not executing and waiting to perform a channel operation



When is a Kahn process considered in a given state?

Sequential implementation (1)

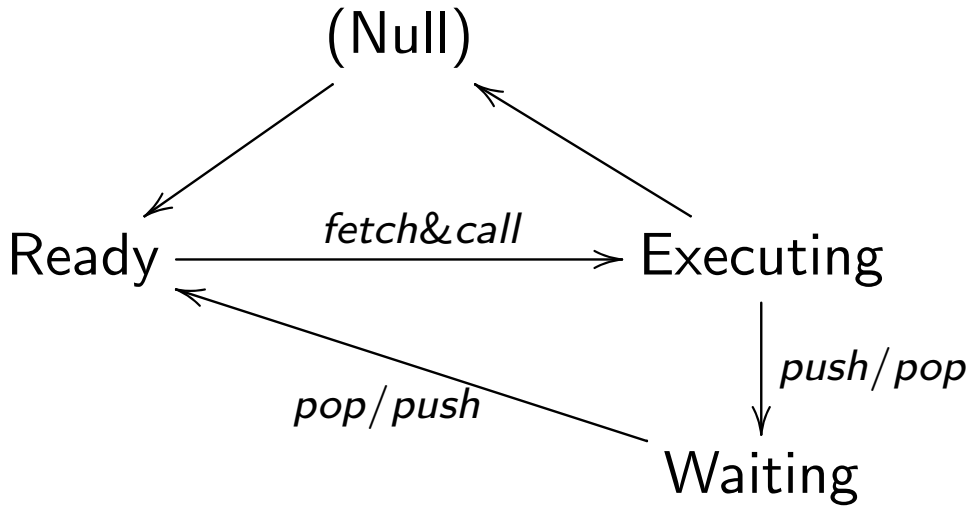
In an implementation with a single scheduler thread

- A single Kahn process at a time is *executing*, when control is currently not in runtime code
- A *ready set* holds all Kahn processes ready to be executed
- Each channel has a *read waiting set* that is either empty or contains a single Kahn process waiting to *pop*
- Analogously each channel has a *write waiting set*

Sequential implementation (2)

Runtime code moves Kahn processes from one set to another

- The scheduling loop makes one *ready* Kahn process *executing*
- An unsatisfied pop (resp. push) operation moves the requesting Kahn process from *executing* into the *read* (resp. *write*) *waiting set*
- A successful push or pop operation moves any Kahn process from the opposite *waiting set* into the *ready set*



Multiple scheduler threads (1)

In an implementation with multiple scheduler threads

- Each scheduler thread may execute a different Kahn process simultaneously (parallelism)
- All threads equally need to fetch *ready* Kahn processes, which may generate contention
- Therefore, we want to distribute the *ready set* data structure across scheduler threads
- Conversely, *waiting sets* have at most one element and gain nothing by being distributed

Multiple scheduler threads (2)

Thus, given N scheduler threads

- There are at most N *executing* Kahn processes at a time
- There are N *ready sets*
- There are still only two *waiting sets* per channel, shared by all N scheduler threads

What could go wrong?

If data structures representing the sets of Kahn processes are always only accessed atomically (with a lock) by every thread, then there is no problem

What if we want to work without locks?

- Half-complete accesses may interleave
- In a relaxed memory model, different states of the shared memory may be visible to different threads
- Both situations yield inconsistent views of the data structures in different threads

What could go wrong?

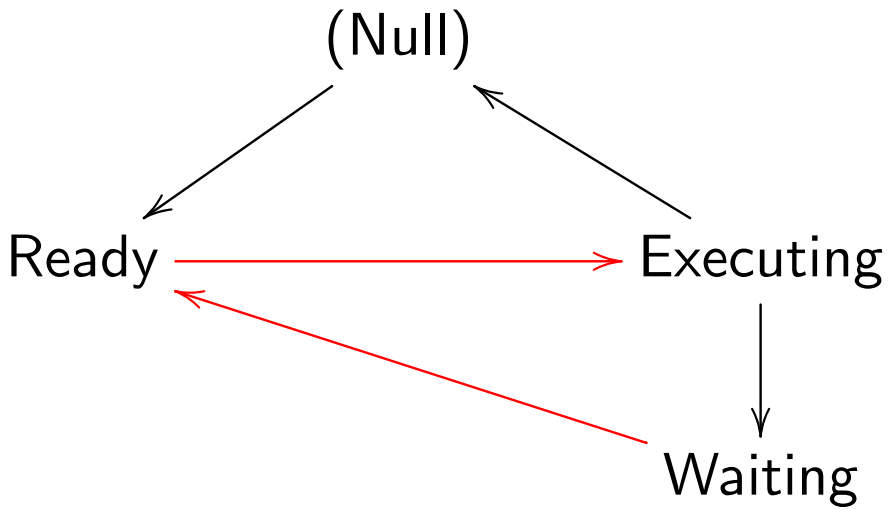
- Duplication of a Kahn process
- Loss of a Kahn process

Duplication

Two cases of possible duplication

- Of ready Kahn processes removed from a *ready set*
- Of waiting Kahn processes removed from a *waiting set*

Requires atomic (exclusive) transitions between Ready and other meaningful states

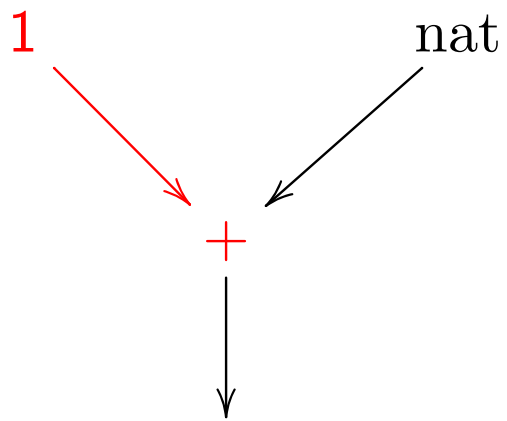


Non-duplication in libkpn

- We use work-stealing deques to represent *ready sets* (previous work on Chase–Lev queue in a relaxed memory model)
- Two atomic pointers per channel, to represent the *read* and *write waiting sets*
- Hand-written proof in C11 that duplication does not occur: invocations of the same Kahn process are totally ordered by *happens-before* (across all schedulers threads)

Let's take a step back (to our example)

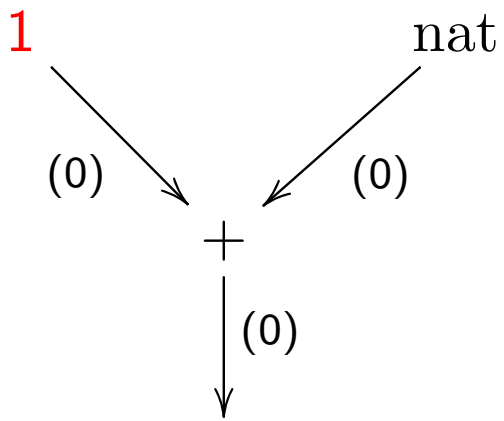
What about losses?
We recall our simple example

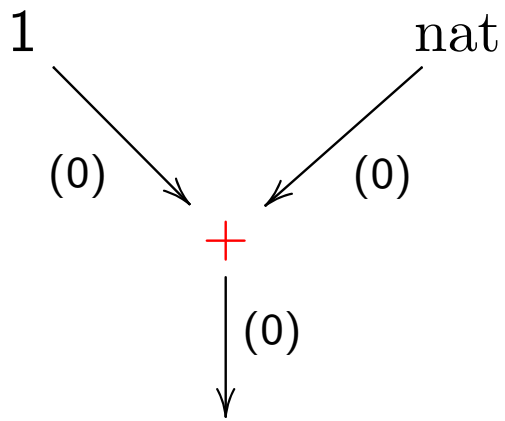


We focus on the case where $+$ may need to wait for 1 to produce more values

Loss

Executing Waiting



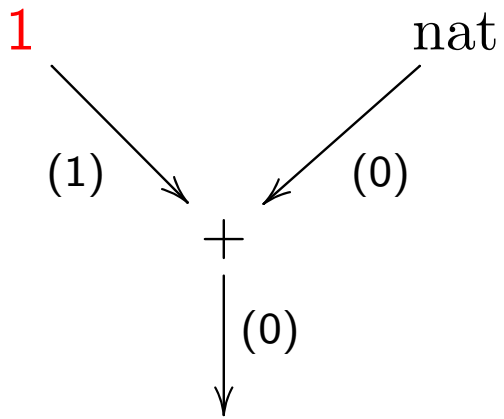


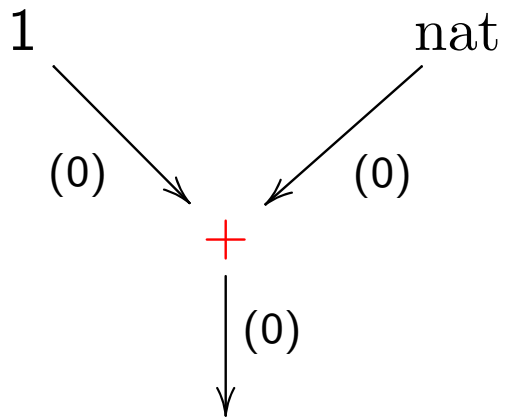
1 is executing on A

$+$ is executing on B

Loss

Executing Waiting



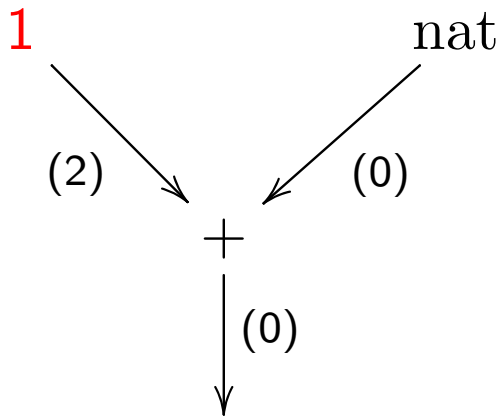


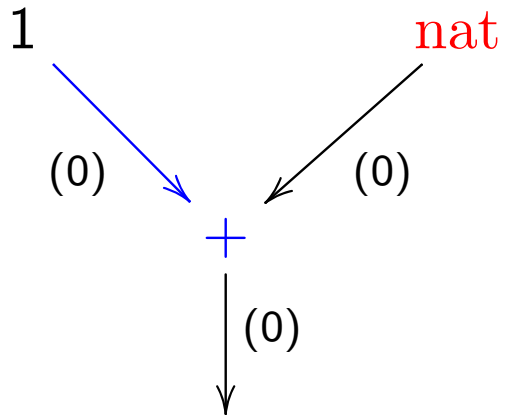
1 pushes the first number to its output channel

$+$ attempts to pop the first number from its left input channel and fails

Loss

Executing Waiting



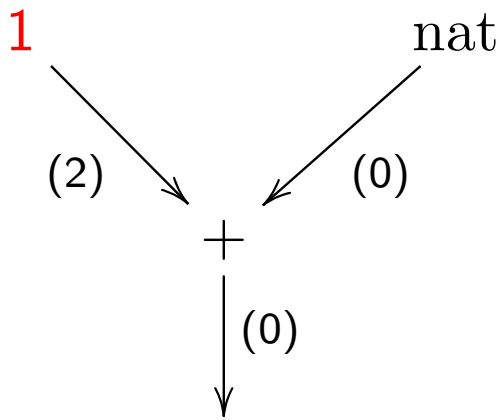


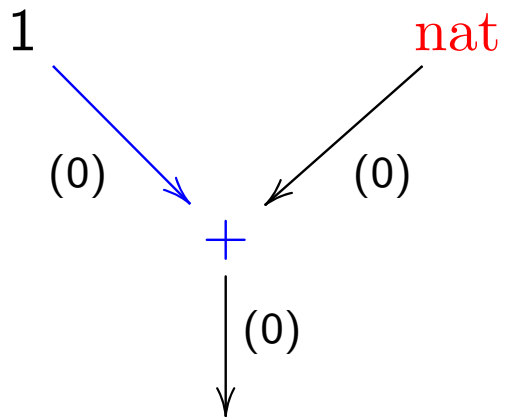
A is not (yet) aware of + being in the *waiting set*

B returns to its scheduling loop and puts + into the *waiting set* of its left input channel

Loss

Executing Waiting





A is not (yet) aware of + being in the *waiting set*

B is not (yet) aware of the first push

Which thread should ready + now?

When can we ready a waiting process?

To remove a Kahn process from a *waiting set*

- There (obviously) needs to be a Kahn process waiting
- And an opposite operation needs to occur
- Once a thread confirms both conditions, it may attempt to move the waiting Kahn process into its *ready set*
- How soon is "once?"

Whose responsibility is it to ready waiting Kahn processes?

- A thread executing the requesting Kahn process?
  (In the previous example, B)
- A thread executing the opposite Kahn process?
  (In the previous example, A)
- Another thread?

Loss prevention strategies

- Different strategies (who, when) possible
- We present two: ASAP and (somewhat) lazy
- Both proven (by hand) not to induce losses, including in the presence of cycles: not all Kahn processes in a KPN are waiting in a deadlock if at least one of the waited on channels has enough space or data

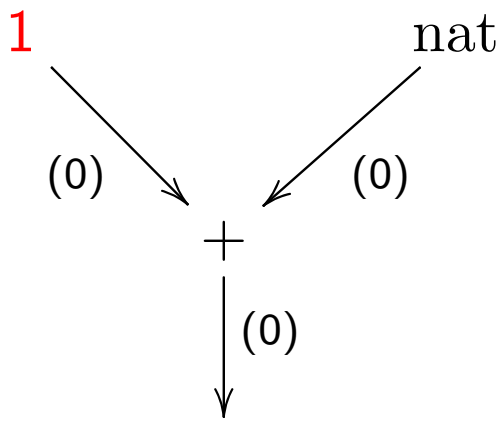
ASAP strategy

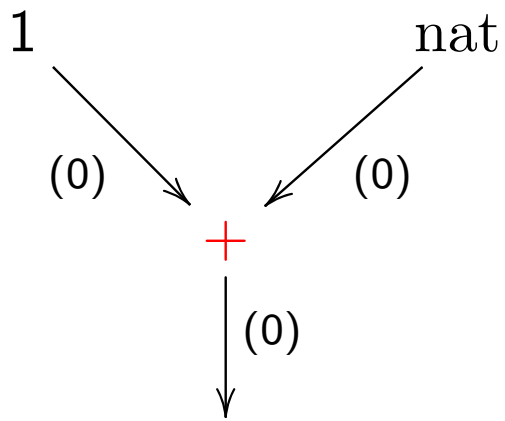
Atomic (sequential) waiting and publishing operations
At the end of the previous example scenario

- Both A and B are responsible for readying $+$ if possible
- After the succesful push of the i-th number, A checks whether the *waiting set* is empty
- After moving $+$ to the *waiting set*, B rechecks whether the i-th number is now available
- Both of these checks are atomic with respect to each other: they are **totally ordered**
- If either check succeeds, an attempt is made to atomically remove $+$ from the ready set (compare-and-swap)

ASAP strategy illustrated

Executing Waiting



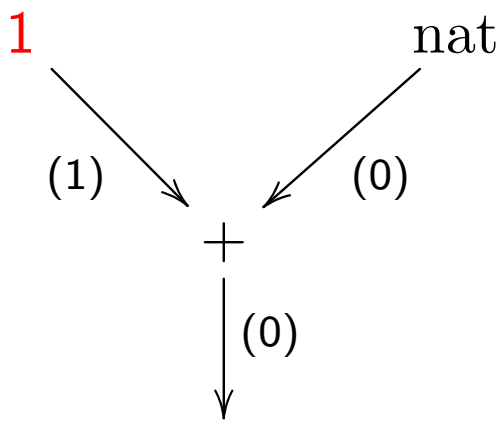


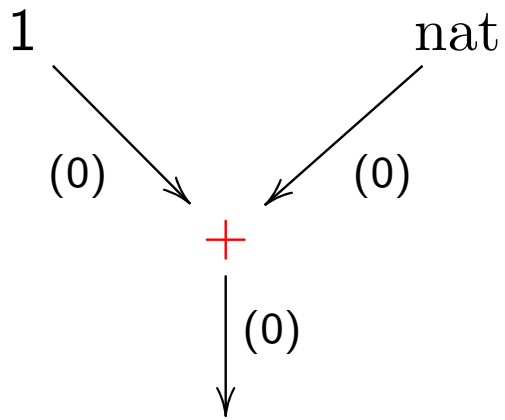
1 is executing on A

$+$ is executing on B

ASAP strategy illustrated

Executing Waiting



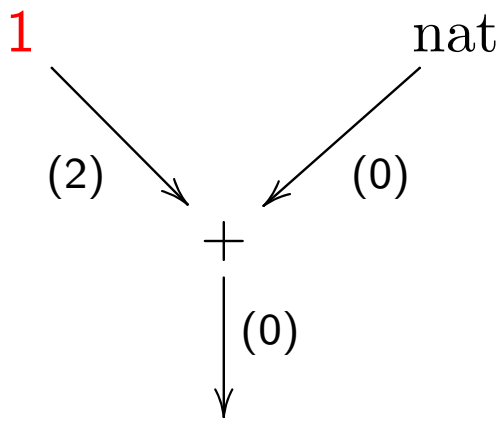


1 pushes the first number to its output channel

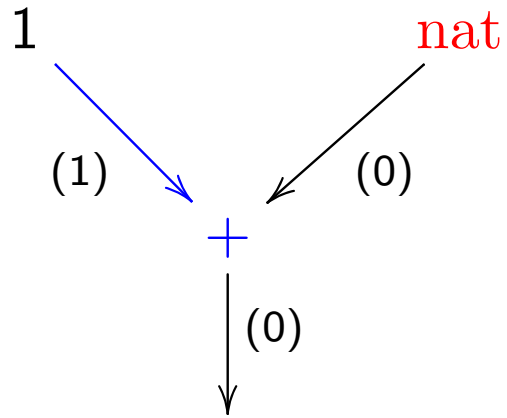
$+$ attempts to pop the first number from its left input channel and fails

ASAP strategy illustrated

Executing Waiting



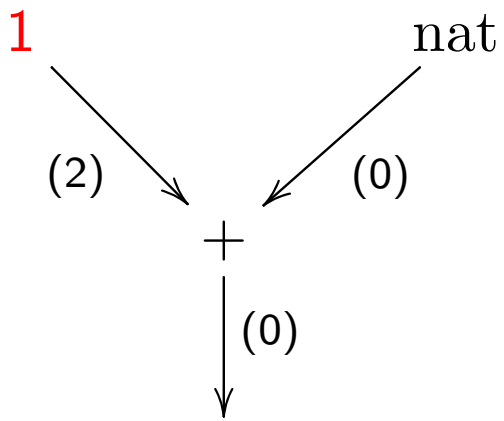
A is not (yet) aware of + being in the *waiting set*

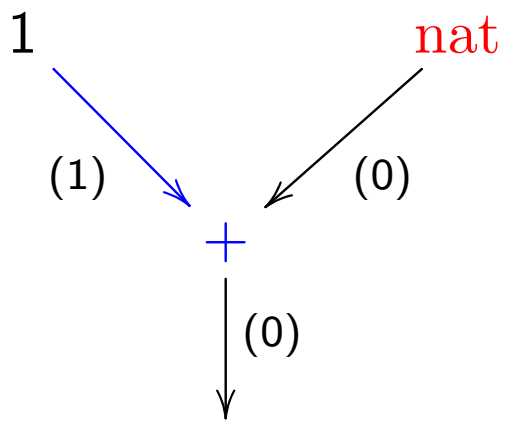


B returns to its scheduling loop and puts + into the *waiting set* of its left input channel, and **rechecks**

ASAP strategy illustrated

Executing Waiting





A is not (yet) aware of + being in the *waiting set*

This time, *B* is aware of the first push, and will ready +

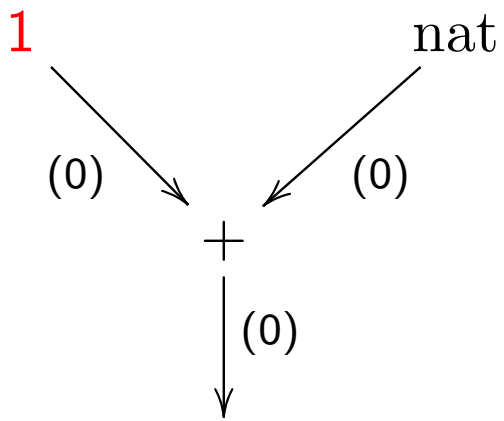
Lazy strategy

Idea: atomic checks are costly
Replace costly checks with **relaxed tests** as much as possible

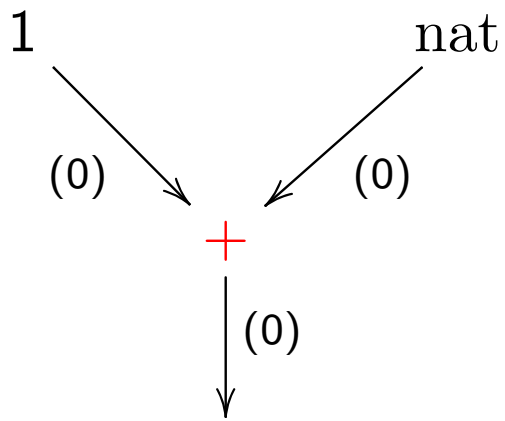
- Only A is responsible for readying $+$
- After the succesful push of the i-th number, A non-atomically checks whether the *waiting set* is empty
- Let's call this non-atomic check *probing*
- Then A may probe again every now and then
- When A finishes executing $+$, it can stop probing and perform an atomic check

Lazy strategy illustrated

Executing Waiting



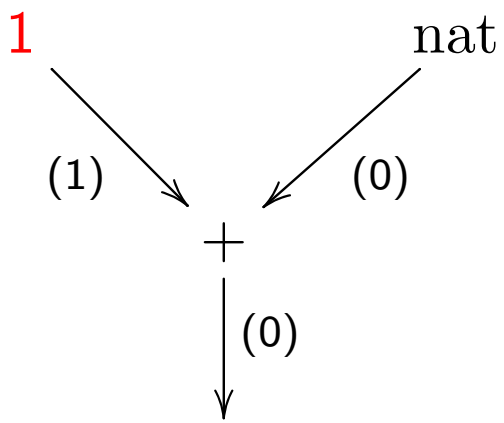
1 is executing on A

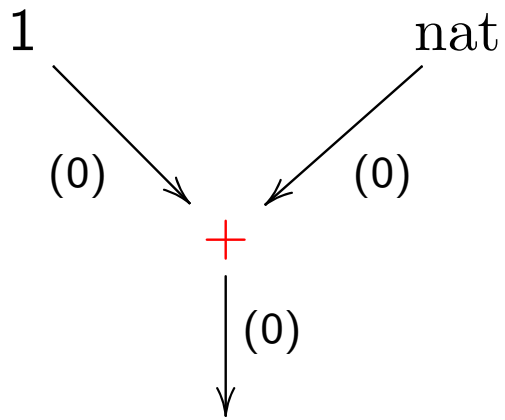


$+$ is executing on B

Lazy strategy illustrated

Executing Waiting



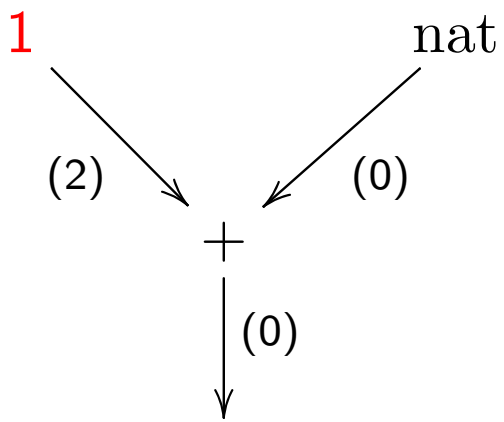


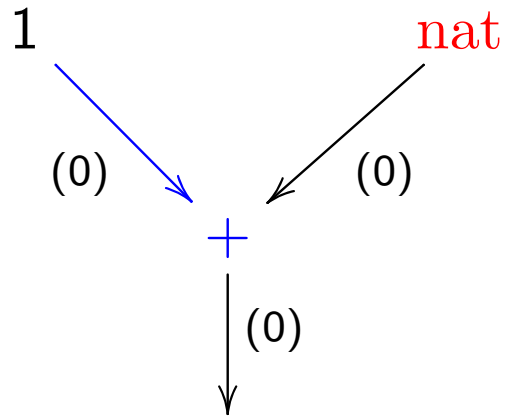
1 pushes the first number to its output channel

$+$ attempts to pop the first number from its left input channel and fails

Lazy strategy illustrated

Executing Waiting



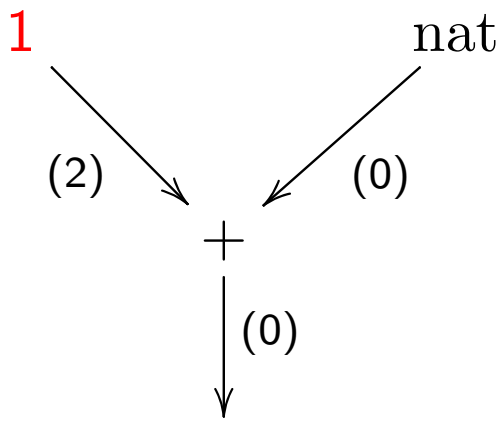


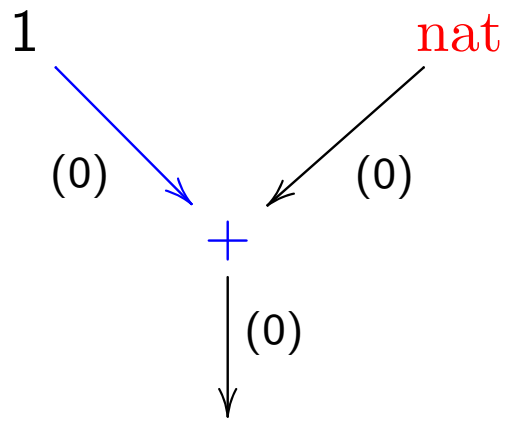
A is not (yet) aware of + being in the *waiting set*

B returns to its scheduling loop and puts + into the *waiting set* of its left input channel

Lazy strategy illustrated

Executing Waiting



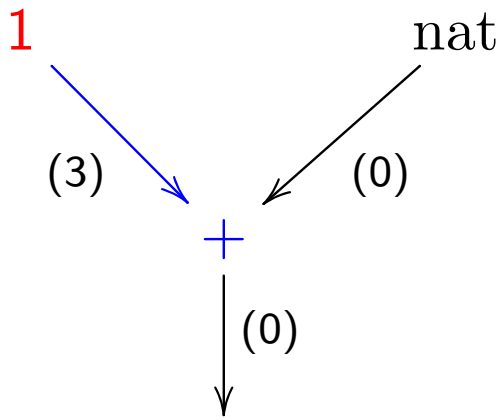


A is not (yet) aware of + being in the *waiting set*

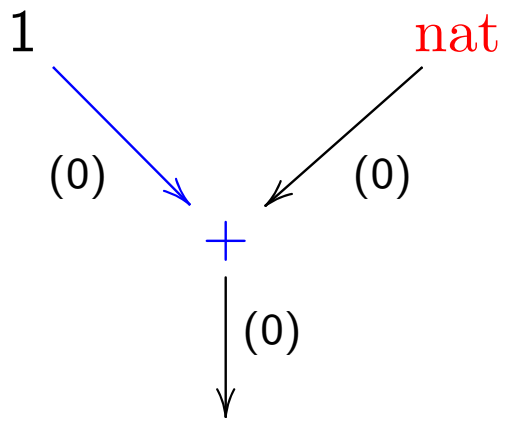
B is not (yet) aware of the first push

Lazy strategy illustrated

Executing Waiting



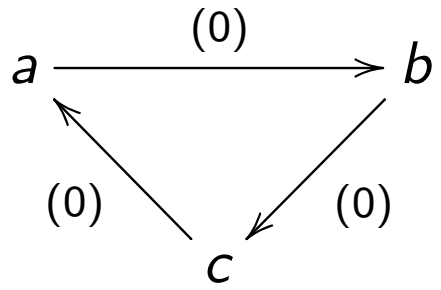
1 pushes a third time and probes:
A **is now aware of** + **being in the waiting set**



B is still not aware of the first push

Atomic checks in the lazy strategy

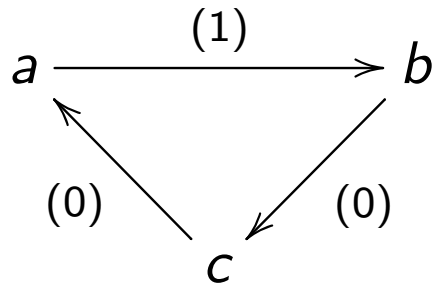
Which Kahn processes do we atomically check?



- *b* attempts to pop from *a* but fails

Atomic checks in the lazy strategy

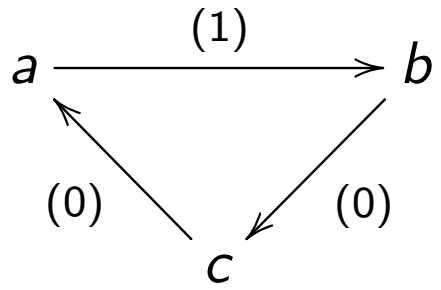
Which Kahn processes do we atomically check?



- *b* attempts to pop from *a* but fails
- *a* pushes to *b*, checks but *b* is not waiting yet

Atomic checks in the lazy strategy

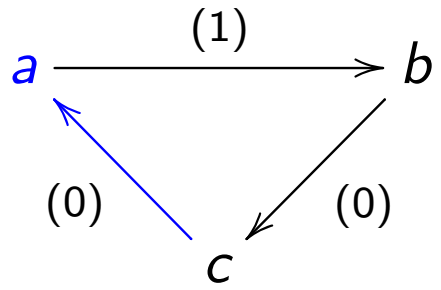
Which Kahn processes do we atomically check?



- *b* attempts to pop from *a* but fails
- *a* pushes to *b*, checks but *b* is not waiting yet
- *a* attempts to pop from *c* but fails

Atomic checks in the lazy strategy

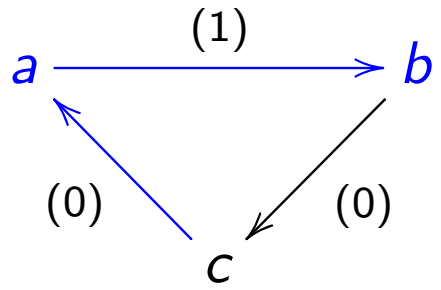
Which Kahn processes do we atomically check?



- *b* attempts to pop from *a* but fails
- *a* pushes to *b*, checks but *b* is not waiting yet
- *a* attempts to pop from *c* but fails
- *a* stops to wait (atomic check)

Atomic checks in the lazy strategy

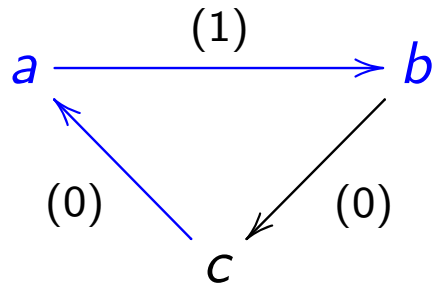
Which Kahn processes do we atomically check?



- *b* attempts to pop from *a* but fails
- *a* pushes to *b*, checks but *b* is not waiting yet
- *a* attempts to pop from *c* but fails
- *a* stops to wait (atomic check)
- *b* stops to wait (atomic check)

Atomic checks in the lazy strategy

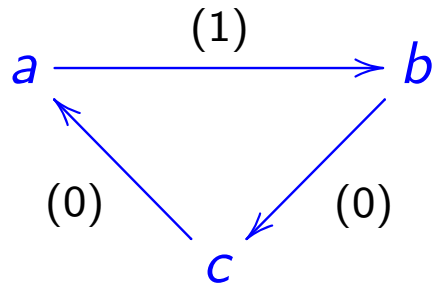
Which Kahn processes do we atomically check?



- *b* attempts to pop from *a* but fails
- *a* pushes to *b*, checks but *b* is not waiting yet
- *a* attempts to pop from *c* but fails
- *a* stops to wait (atomic check)
- *b* stops to wait (atomic check)
- *c* attempts to pop from *b* but fails

Atomic checks in the lazy strategy

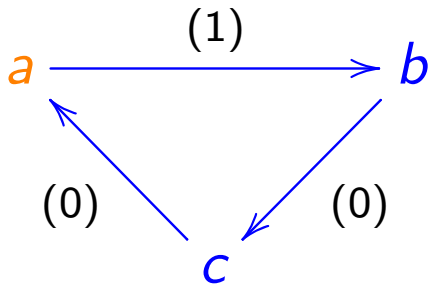
Which Kahn processes do we atomically check?



- *b* attempts to pop from *a* but fails
- *a* pushes to *b*, checks but *b* is not waiting yet
- *a* attempts to pop from *c* but fails
- *a* stops to wait (atomic check)
- *b* stops to wait (atomic check)
- *c* attempts to pop from *b* but fails
- *c* stops to wait (atomic check)

A non-working strategy

Suppose we only check atomically channels we have operated on



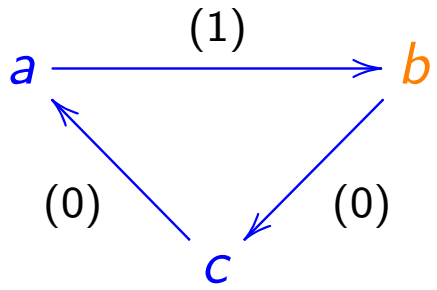
Atomic checks total order: a, b, c

- a has pushed to b thus checks atomically for b, but b is not waiting yet; then waits

Result: deadlock

A non-working strategy

Suppose we only check atomically channels we have operated on



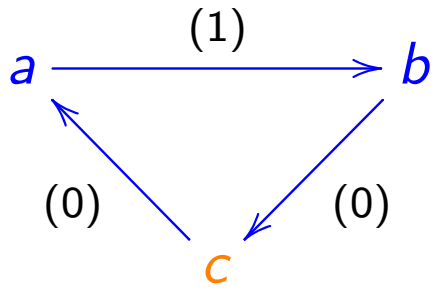
Atomic checks total order: a, b, c

- a has pushed to b thus checks atomically for b, but b is not waiting yet; then waits
- b checks nothing because it didn't push anything; then waits

Result: deadlock

A non-working strategy

Suppose we only check atomically channels we have operated on



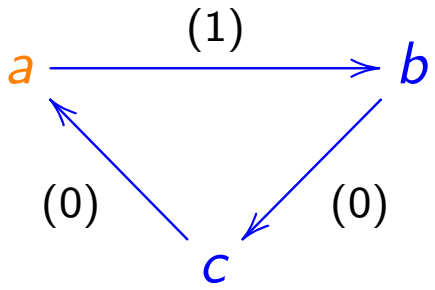
Atomic checks total order: a, b, c

- a has pushed to b thus checks atomically for b, but b is not waiting yet; then waits
- b checks nothing because it didn't push anything; then waits
- c checks nothing because it didn't push anything; then waits

Result: deadlock

A working strategy

Suppose we check atomically all connected channels



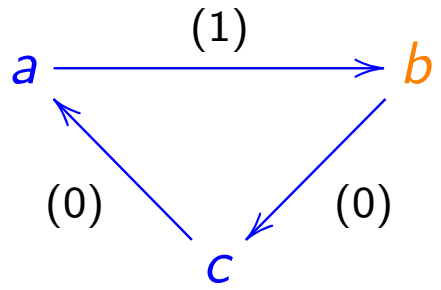
Atomic checks total order: a, b, c

- a checks atomically for b, but b is not waiting yet; then waits

Result: b will make progress when it is scheduled

A working strategy

Suppose we check atomically all connected channels



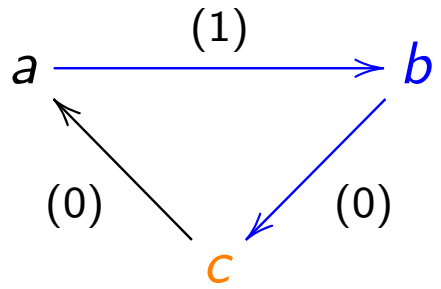
Atomic checks total order: a, b, c

- a checks atomically for b, but b is not waiting yet; then waits
- b checks atomically for c, but c is not waiting yet; then waits

Result: b will make progress when it is scheduled

A working strategy

Suppose we check atomically all connected channels



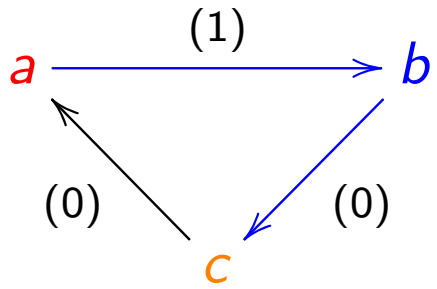
Atomic checks total order: a, b, c

- a checks atomically for b, but b is not waiting yet; then waits
- b checks atomically for c, but c is not waiting yet; then waits
- c checks atomically for a, and readies a; then waits

Result: b will make progress when it is scheduled

A working strategy

Suppose we check atomically all connected channels



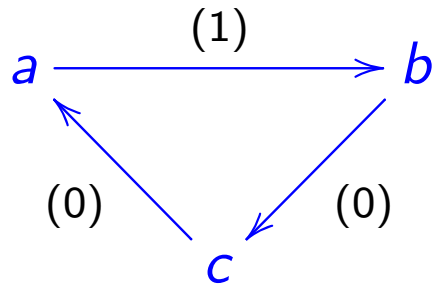
Atomic checks total order: a, b, c

- a checks atomically for b, but b is not waiting yet; then waits
- b checks atomically for c, but c is not waiting yet; then waits
- c checks atomically for a, and readies a; then waits
- a is executed but cannot make progress

Result: b will make progress when it is scheduled

A working strategy

Suppose we check atomically all connected channels



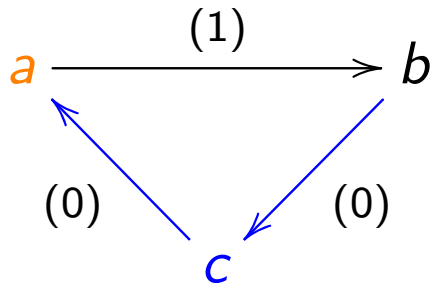
Atomic checks total order: a, b, c

- a checks atomically for b, but b is not waiting yet; then waits
- b checks atomically for c, but c is not waiting yet; then waits
- c checks atomically for a, and readies a; then waits
- a is executed but cannot make progress
- a stops to wait

Result: b will make progress when it is scheduled

A working strategy

Suppose we check atomically all connected channels



Atomic checks total order: a, b, c

- a checks atomically for b, but b is not waiting yet; then waits
- b checks atomically for c, but c is not waiting yet; then waits
- c checks atomically for a, and readies a; then waits
- a is executed but cannot make progress
- a stops to wait
- a checks atomically for b, and readies b; then waits

Result: b will make progress when it is scheduled

Summary of loss prevention strategies

	ASAP	Lazy
Check per push/pop	atomic	relaxed
Check on wait	atomic	atomic
Combined checks on push/pop	no	yes
Must check all adjacent processes	no	yes
Faux-ready	no	yes

Comparisons and applications

Task-parallel models

- Libkpn can be seen as implementing a task-parallel model where tasks are continuous execution intervals of Kahn processes
- Libkpn implements dynamic Kahn process creation and channels as first-class objects (can only be bound to a single producer and a single consumer at a time)
- The KPN model thus implemented supersedes Cilk, SMPSs, as well as hybrid models with SMPSs-style dependencies between children of Cilk-like tasks

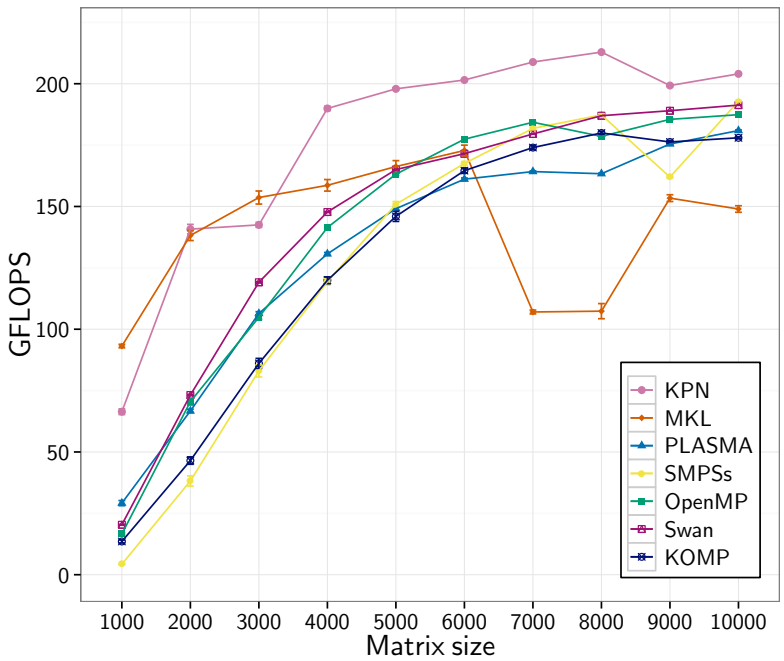
Extensions to task-parallel models

From a purely task-oriented algorithm

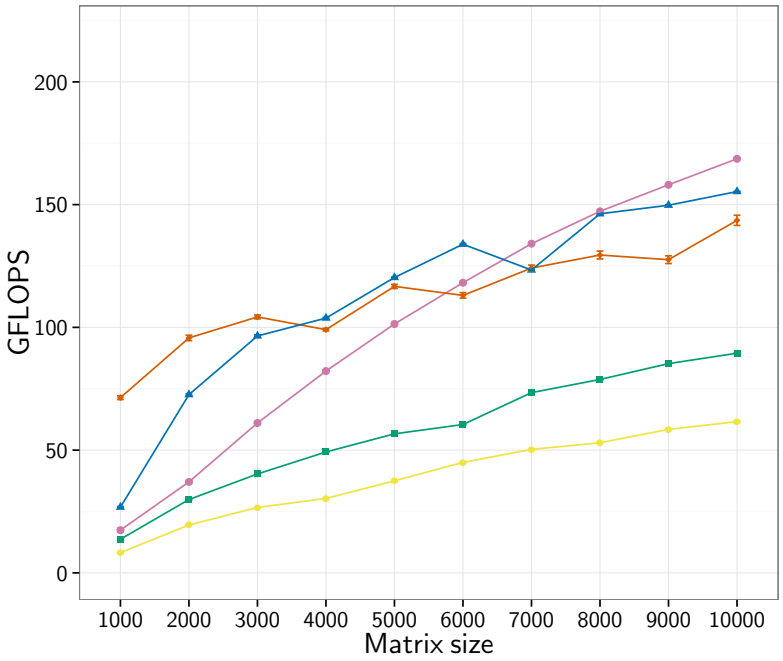
- Kahn processes can be used to aggregate tasks
- Channels can be used to aggregate dependencies
- In practice, libkpn offers efficient and realistically usable buffered communication and synchronization ("streaming") to task-parallel programs

Cholesky and LU on x86-64

Cholesky



LU



Manually optimized to use Kahn processes and channels:

- Gains from aggregation
- Better creation balancing

Conclusion

- We've got that cool library at PARKAS called libkpn
- It is built using modern C, is portable, and we are reasonably sure its core algorithms aren't totally faulty
- It lets you structure parallel C programs as KPNs
- It can compete with existing task-parallel runtimes while allowing streaming capabilities that are actually usable
- If you take advantage of its features, it can run your KPNs really fast, and we have a couple of applications to show off

Extra

To probe or not to probe (1)

How does A remember to probe?
When does A forget and stop probing?

- The KPN should not deadlock if a Kahn process network can be readied (missed checks)
- As much as possible, we do not want multiple threads checking the same condition (redundant checks)
- We do not want to ready a Kahn process unless the channel it waits on meets the waiting conditions (faux-ready)

To probe or not to probe (2)

As long as a Kahn process is running, only its executing thread pushes to its output channels and pops from its input channels

- No other thread can ready Kahn processes waiting on those channels (no redundant checks)
- If a probe misses, as long as the Kahn process is running, waiting conditions do not change (no faux-ready)

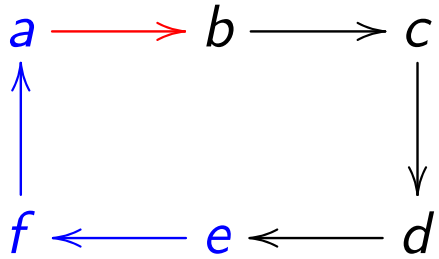
Atomic checks

- When a Kahn process stops and waits, its execution may continue on another thread, thus further probing may be redundant or induce faux-ready; hence, we should stop checking
- When a Kahn process terminates, we want to release memory allocated to it, including information on its channels; hence, we should stop probing
- When we stop probing, we must atomically check and ready any waiting adjacent Kahn process

We ready even those adjacent Kahn processes whose waiting conditions are not satisfied; why is that?

Why is the lazy strategy correct?

If there is a cycle of Kahn processes waiting on one another and at least one of the connecting channel has enough data to keep the network running, then it will eventually be found by the rippling effects of atomic checks



Any Kahn process waiting between the last Kahn process to stop (e) and the first to be able to make progress (b) will be faux-readied