RAFTLIB

@ CPPNow 2017

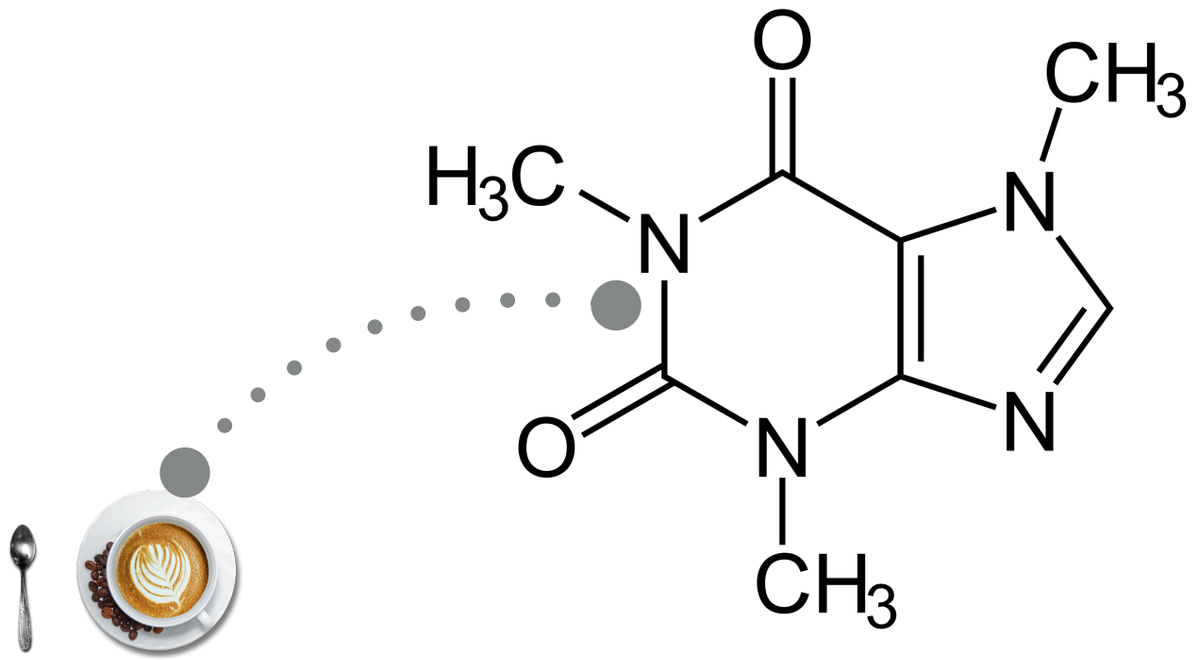
>> SIMPLER PARALLEL PROGRAMMING

Presented By: Dr. Jonathan Beard

All thoughts, opinions are my own. RaftLib is not a product of ARM Inc. Please don’t ask about ARM products or strategy. I will scowl and not answer.

Thank you for your cooperation 😎

Work currently supported by:



Work has (in the past) been supported by:

SBS
Stream Based
Supercomputing Lab
http://sbs.wustl.edu

Washington University in St.Louis

VELOCIDATA

NSF

exegy

Thanks to FairPixels.co for the snazzy new logo

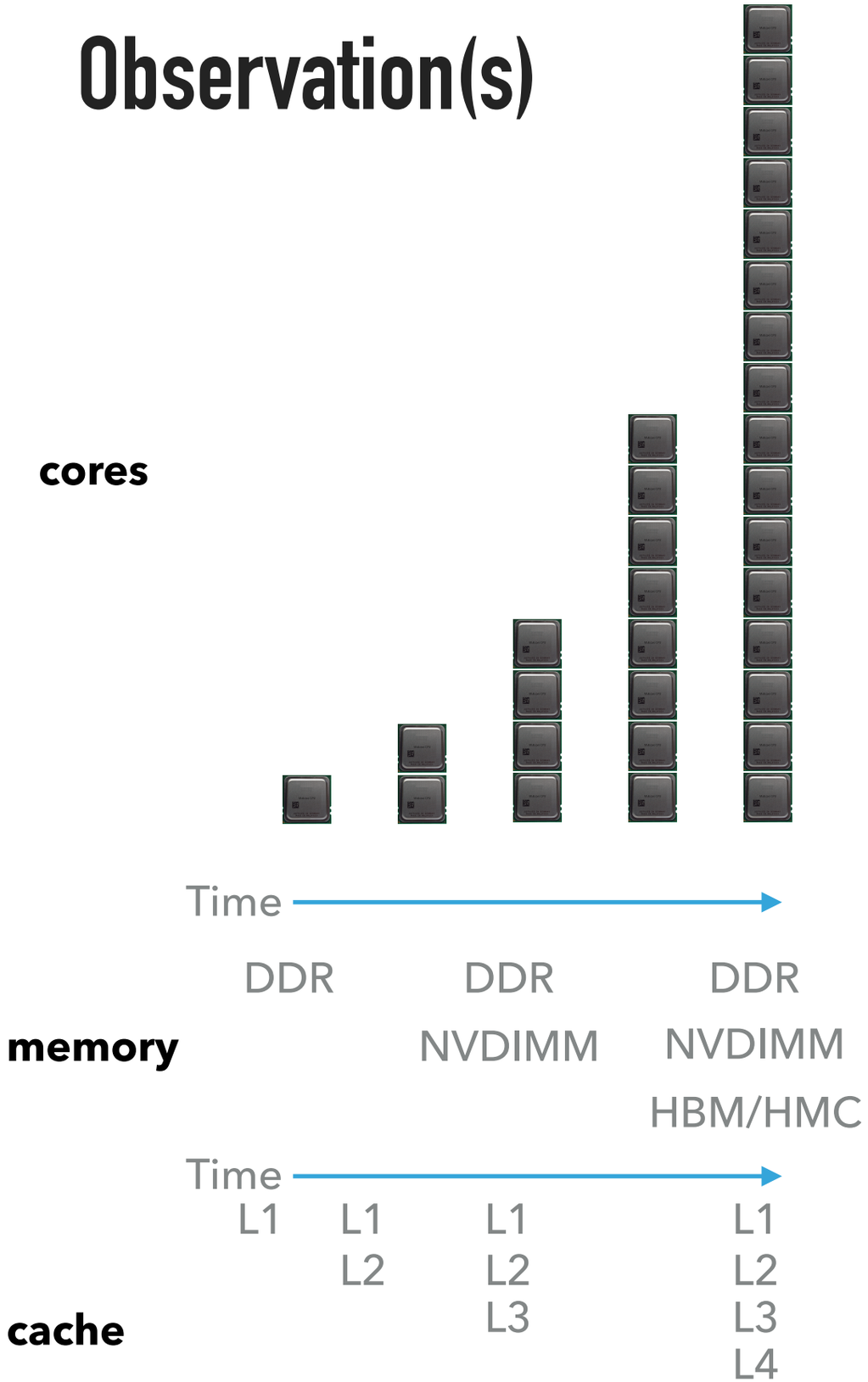
What It Is

- RaftLib is a C++ library that enables data-flow / stream processing using IOstream-like operators
- Header + compiled library
- Enables programmers to link compute kernels with “>>” without having to worry about explicit memory management, thread creation, etc.
- Now runs on OS X, Linux, Unix, and (theoretically) Windows 10

Outline

- Intro / Motivation
- Basic Concepts
- Interfacing
- Allocations and memory model
- Interactive

Observation(s)



- Frequency scaling is basically done
- Core counts are going to get higher
- Memory systems are more heterogeneous
- Hierarchies are getting longer and longer
- Programming constructs to deal with are largely absent

Is this the best we can do?

Move In

Move Out

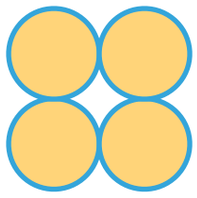
Observation(s)

Single Cell

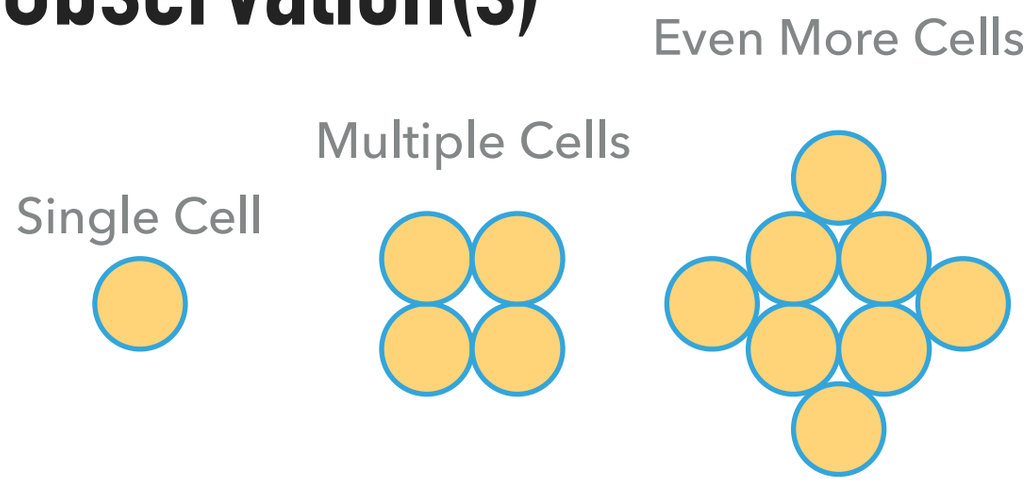
Observation(s)

Single Cell

Multiple Cells



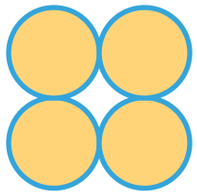
Observation(s)



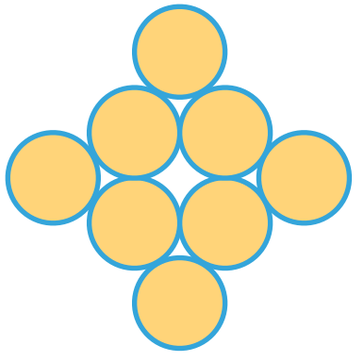
Observation(s)

Single Cell

Multiple Cells



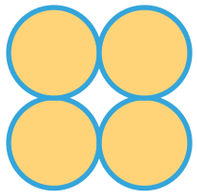
Even More Cells



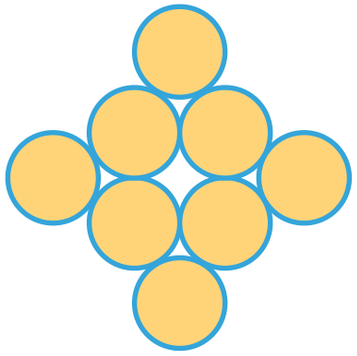
Observation(s)

Single Cell

Multiple Cells



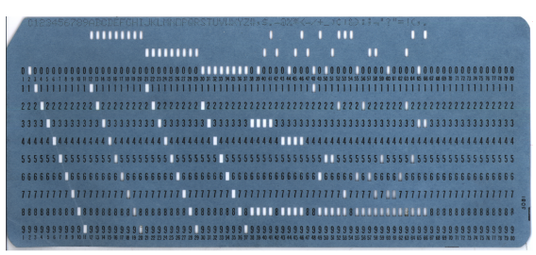
Even More Cells

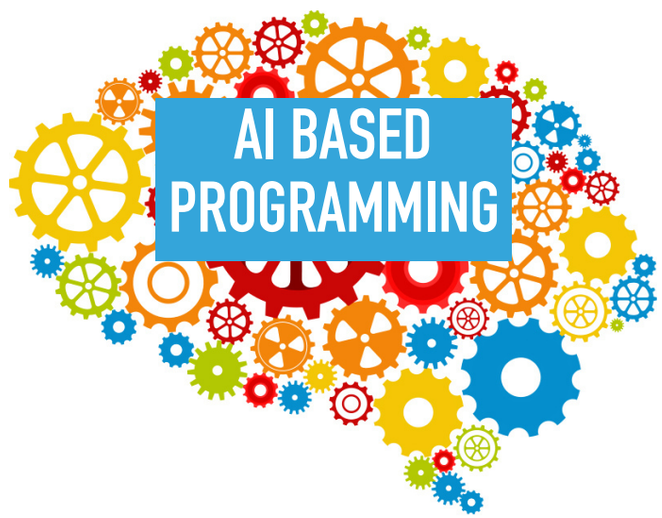


Difference: It’s not the cells. It’s the interaction between them that makes a cat, a cat.

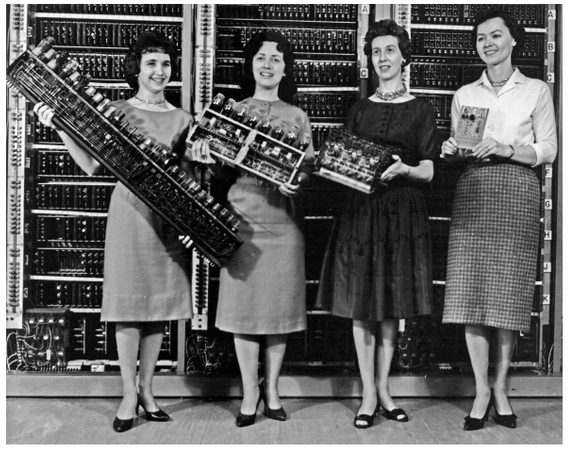
Language Automation

- Switches to ML

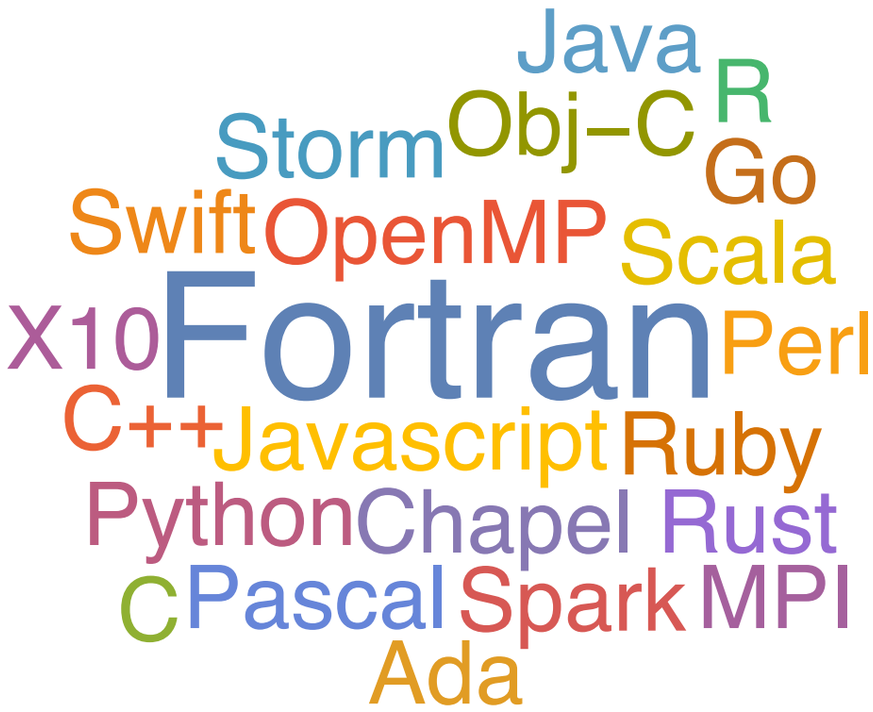




1940's 1950's 1960's ??



source: https://goo.gl/gScHwW



Motivation

- Productivity
- Often code costs more than the machine that it is run on
- Hardware is a commodity resource, coding takes people and time
- Obligatory Moore's Law reference (inserted, check)
  - everything must be parallel going forward
  - must reduce Amdahl quotient to theoretical minimum
  - this takes even more money, time, effort

Motivation

- I hate “boilerplate” code (we all do, right?)
- Example:

```
#ifdef __linux
   /**
    * pin the current thread
    */
   cpu_set_t   *cpuset( nullptr );
   auto cpu_allocate_size( -1 );
#if   (__GLIBC_MINOR__ > 9 ) && (__GLIBC__ == 2 )
   const auto  processors_to_allocate( 1 );
   cpuset = CPU_ALLOC( processors_to_allocate );
   assert( cpuset != nullptr );
   cpu_allocate_size = CPU_ALLOC_SIZE( processors_to_allocate );
   CPU_ZERO_S( cpu_allocate_size, cpuset );
#else
   cpu_allocate_size = sizeof( cpu_set_t );
   cpuset = (cpu_set_t*) malloc( cpu_allocate_size );
   assert( cpuset != nullptr );
   CPU_ZERO( cpuset );
#endif
   CPU_SET( desired_core,
            cpuset );
   errno = 0;
   if( sched_setaffinity( 0 /* calling thread */,
                          cpu_allocate_size,
                          cpuset ) != 0 )
```

Motivation

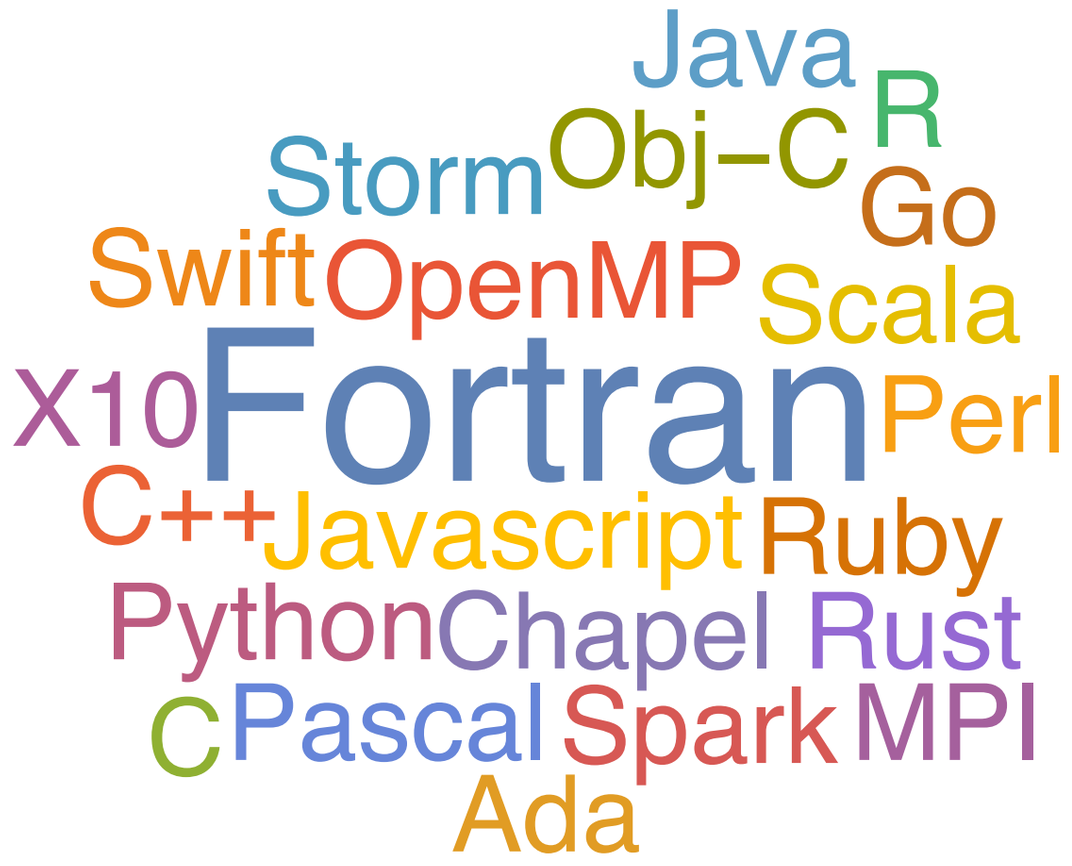
- Most of the steps used to write parallel code are redundant when dependencies are explicit

```
std::thread a( producer, std::ref( producer_key ), std::ref( f ) );
std::thread b( consumer, std::ref( consumer_key ), std::ref( f ) );
a.join();
b.join();
```

- If I specify data dependencies:
  - I can retire thread “a” as soon as it no longer produces data
  - I can retire thread “b” when it is done consuming data from “a”

Motivation

- Big businesses have hundreds of coders
- The cloud equalizes the hardware factor, but not the number of software engineers
- Small businesses need to make their two coders feel like three hundred



Shameless nerdy 300 quote re-work:
Big Business: A thousand coders of the Big Business empire descend upon you. Our code will blot out the sun.
Small Business: Then we will code in the shade.

Miscellaneous Stuff without a Catchy Heading

- Thread libraries might not be the perfect way, but it's a path of less resistance (and more acceptance)
- A full new language might be better (but less accepted)
- Library solutions though provide a nice compromise in a format that is acceptable to the majority of people with C++/C code
- Start looking at programming and interfacing as HCI versus just language theory
  - It's about the humans interacting with the hardware right?

Why not treat languages like we do every other human computer interaction, optimize it for people to use and learn. . .

Dr. Beard

Basic Idea(s) 1/6

- Standard ways of coding for threads don't necessarily scale easily (see example at right)
- Amount of code can easily grow with number of threads and problem size

```
int
main()
{
    std::array< int, length > arr_a;
    std::array< int, length > arr_b;

    /** code to fill array **/


    std::array< std::int64_t, length > arr_out;

    /** manually divide 0-63, 64-127 **/
    std::thread a(  sum,
                    std::ref( arr_a ),
                    std::ref( arr_b ),
                    std::ref( arr_out ),
                    0,
                    63 );
    std::thread b(  sum,
                    std::ref( arr_a ),
                    std::ref( arr_b ),
                    std::ref( arr_out ),
                    64,
                    128 );
    a.join();
    b.join();
  /** do something with arr_out **/
    return( EXIT_SUCCESS );
}
```

Basic Idea(s) 2/6

```
static const auto length( 128 );

static void sum( std::array< int, length > &arr_a,
                 std::array< int, length > &arr_b,
                 std::array< std::int64_t, length > &arr_out,
                 const int start,
                 const int end )
{
    for( auto i( start ); i < end; i++ )
    {
        arr_out[  i ] = arr_a[ i ] + arr_b[ i ];
    }
    return;
}
```

In ceasing to expend energy … **in a process whose main result is to make programs less fit to run on other machine configurations** …, or **to run in company with other programs** …, or to **run with temporarily reduced resources …**, we do more than reduce costs; **we remove self-created obstacles** which today are impeding the development of needed types of systems”

D. Sayre, IBM Yorktown Research (1969)

Basic Idea(s) 3/6

```
class sum : public raft::kernel
{
using type_t = std::int32_t;
public:
   sum() : raft::kernel()
   {
      /** define ports **/
      input.addPort<  type_t >( "input_a", "input_b" );
      output.addPort< type_t >( "sum"      );
   }

   virtual raft::kstatus run()
   {
      type_t a,b;
      /** ports are accessed via names defined above **/
      input[ "input_a" ].pop( a );
      input[ "input_b" ].pop( b );
      /** allocate mem directly on queue **/
      auto c( output[ "sum" ].allocate_s< type_t >() );
      (*c) = a + b;
      /**
       * mem for C pushed to queue on scope exit, proceed
       * signals the run-time to continue
       */
      return( raft::proceed );
   }

};
```

Basic Idea(s) 4/6

- Fewer lines of code overall
- Only marginally more complicated sum function
- Scalable to many threads with basically the same code (or even a single thread with fibers)
- Code can scale-out to heterogeneous devices with the right back-end machinery.

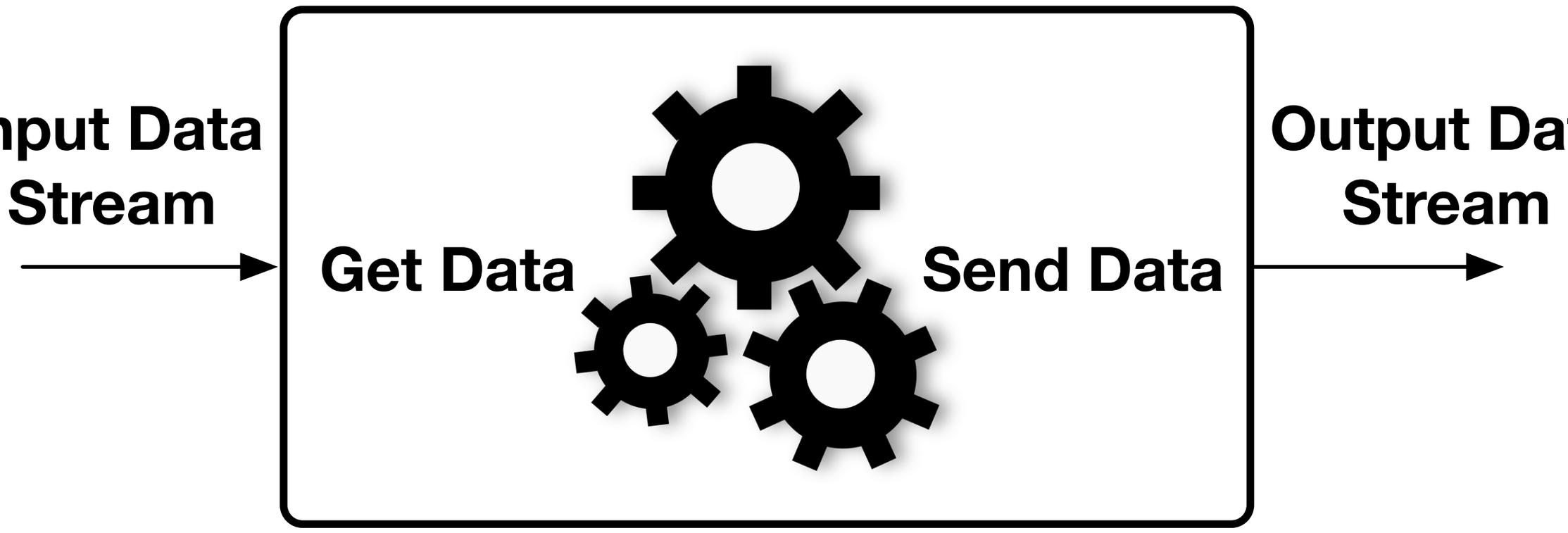
```
gen a( count ), b( count );
sum s;
print p;
raft::map m;
m += a >> s[ "input_a" ];
m += b >> s[ "input_b" ];
m += s >> p;
m.exe();
```

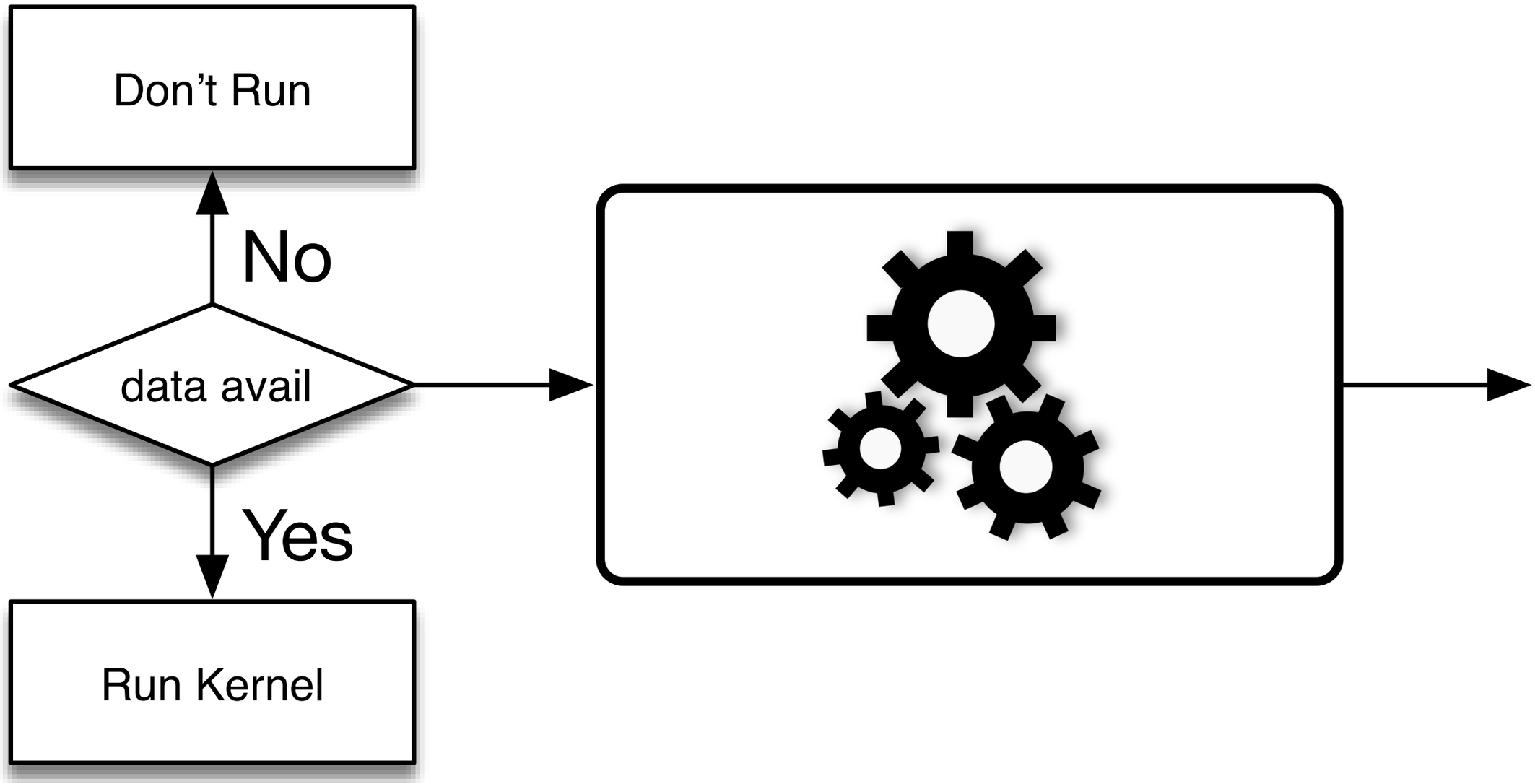
This is HUGE:
Each could be a process, fiber, thread, etc.

Basic Idea 5/6

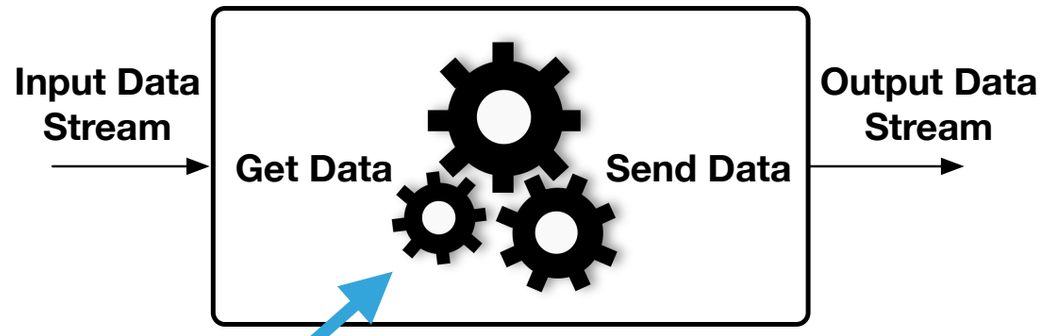
- All needed state is encapsulated inside kernel
- State injected via “port” interfaces
- (not ready yet), but an extension is to provide a definition of ***sum*** in a c++ interface that the runtime recognizes for whatever resource you have
- Provide HDL file that matches that interface (or OpenCL for example)
- Execute CPU-FPGA heterogeneous application (in similar manner to @WUSTL’s original AutoPipe framework)

```
entity sum is
  port(
    clk          : in  std_logic;
    rst          : in  std_logic;

    input_a      : in  unsigned;
    avail_a      : in  std_logic;
    read_a       : out std_logic;

    input_b      : in  unsigned;
    avail_b      : in  std_logic;
    read_b       : out std_logic;

    output_sum   : out unsigned;
    write_sum    : out std_logic;
  );
end sum;

architecture arch of sum is
  signal ack : std_logic;
begin
  output_sum    <= input_a + input_b;
  ack           <= avail_a and avail_b;
  read_a        <= ack;
  read_b        <= ack;
  write_sum     <= ack;
end architecture arch;
```


Input Data Stream
Get Data
Send Data
Output Data Stream


Don’t Run
No
data avail
Yes
Run Kernel

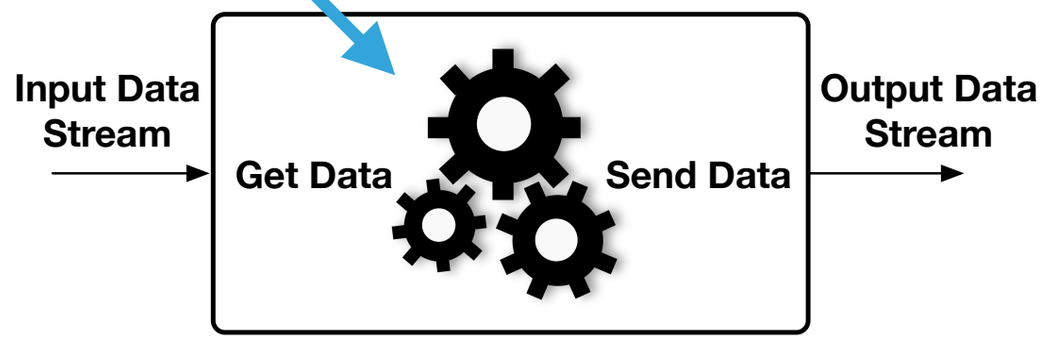
Problem…



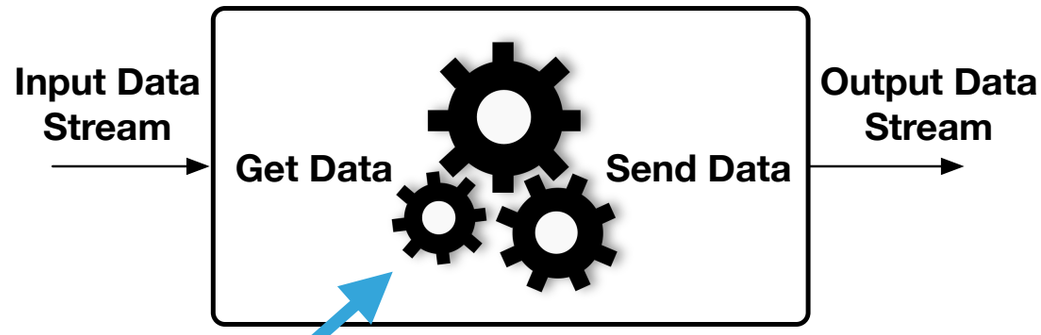
static int a;

State injected

now we have state we can’t move…



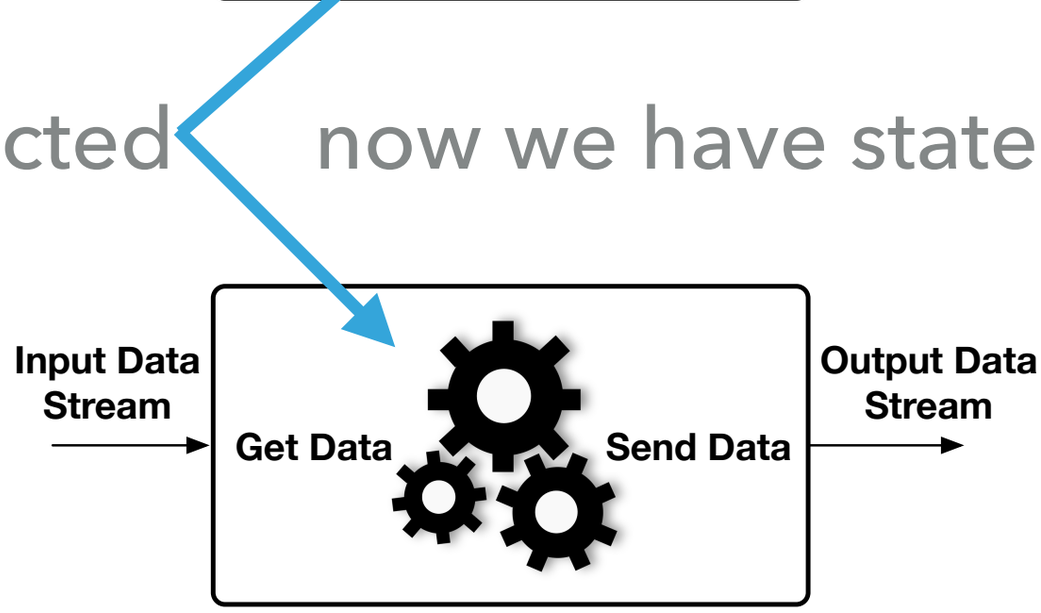
Problem…



static int a;

State injected

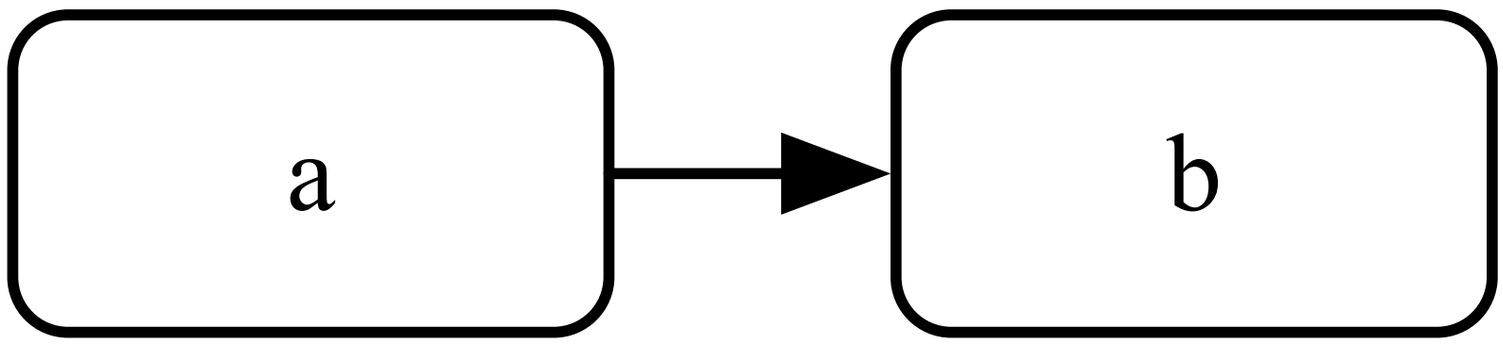
now we have state we can’t move…



First hack solution: If we had something like a “pure” class keyword we could ensure no injected state exists, but likely very hard to implement compiler for with C++ (but not impossible)

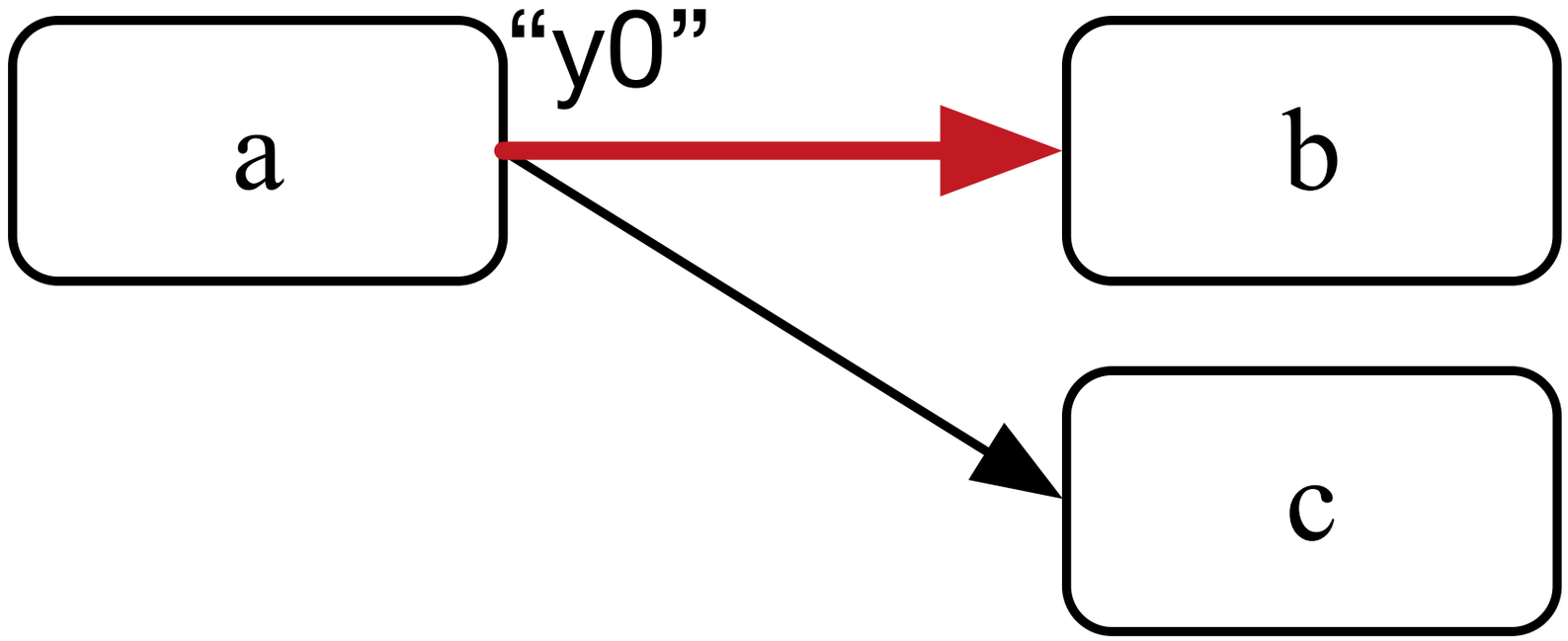
The Basic Link Command

```
raft::map m;
/** example only **/
raft::kernel a, b;
m += a >> b;
```



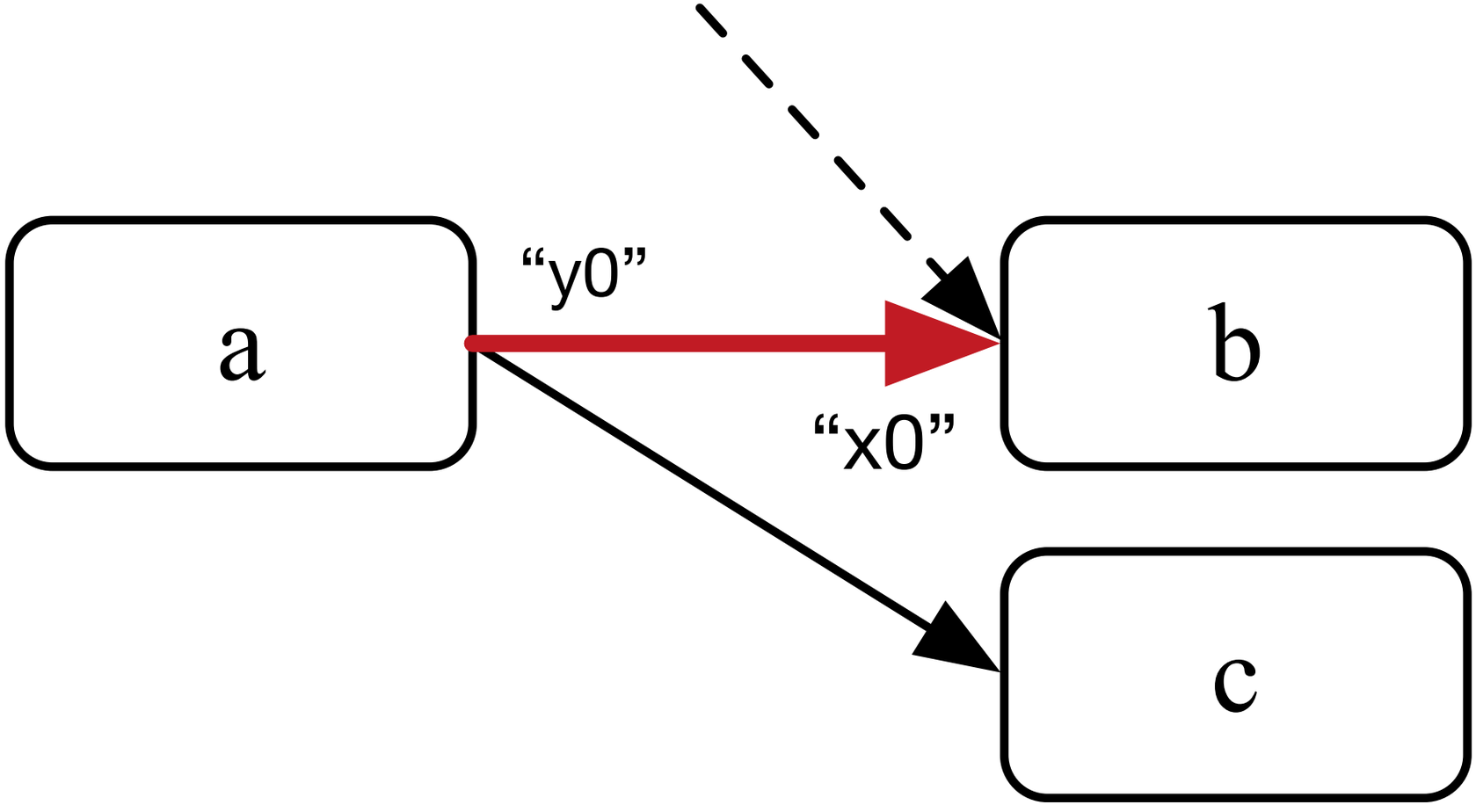
Specifying Ports to Link

```
raft::map m;
/** example only **/
raft::kernel a, b;
m += a[ "y0" ] >> b;
```



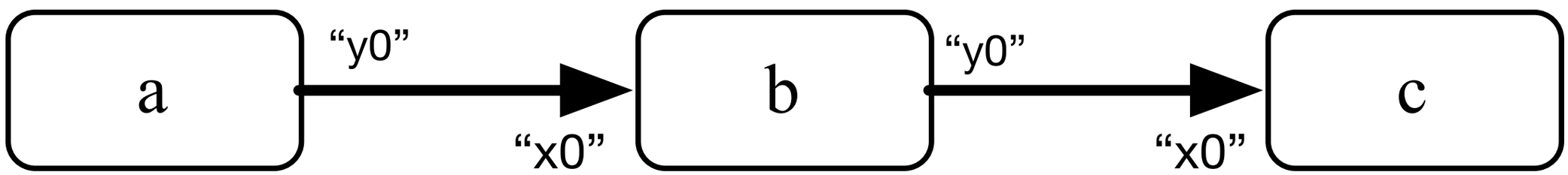
Linking with Two Named Ports

```
raft::map m;
/** example only **/
raft::kernel a, b;
m += a[ "y0" ] >> b[ "x0" ];
```



Linking with Chained Named Ports

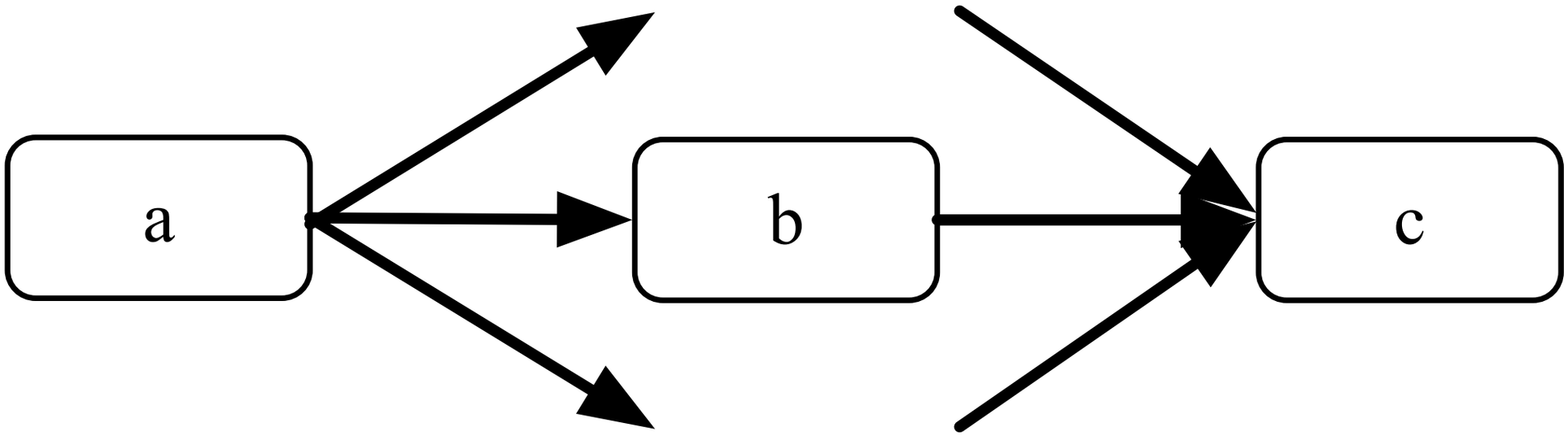
```
raft::map m;
/** example only **/
raft::kernel a, b,c;
m += a[ "y0" ] >> b[ "x0" ][ "y0" ] >> c[ "x0" ];
```



Expansion (reducing duplicate code) 1/2

```
raft::map m;
/** example only **/
raft::kernel a, b, c;
m += a <= b >= c;
```

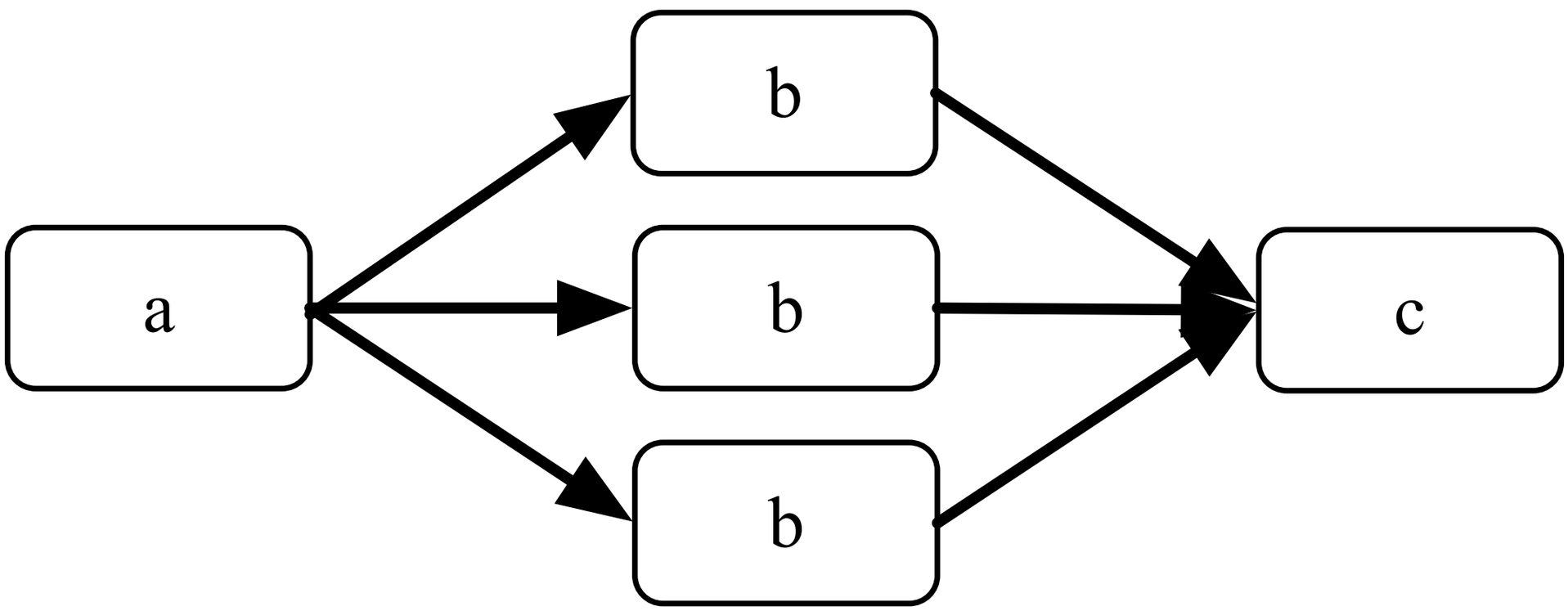
Topology user specifies



Expansion (reducing duplicate code) 2/2

```
raft::map m;
/** example only **/
raft::kernel a, b, c;
m += a <= b >= c;
```

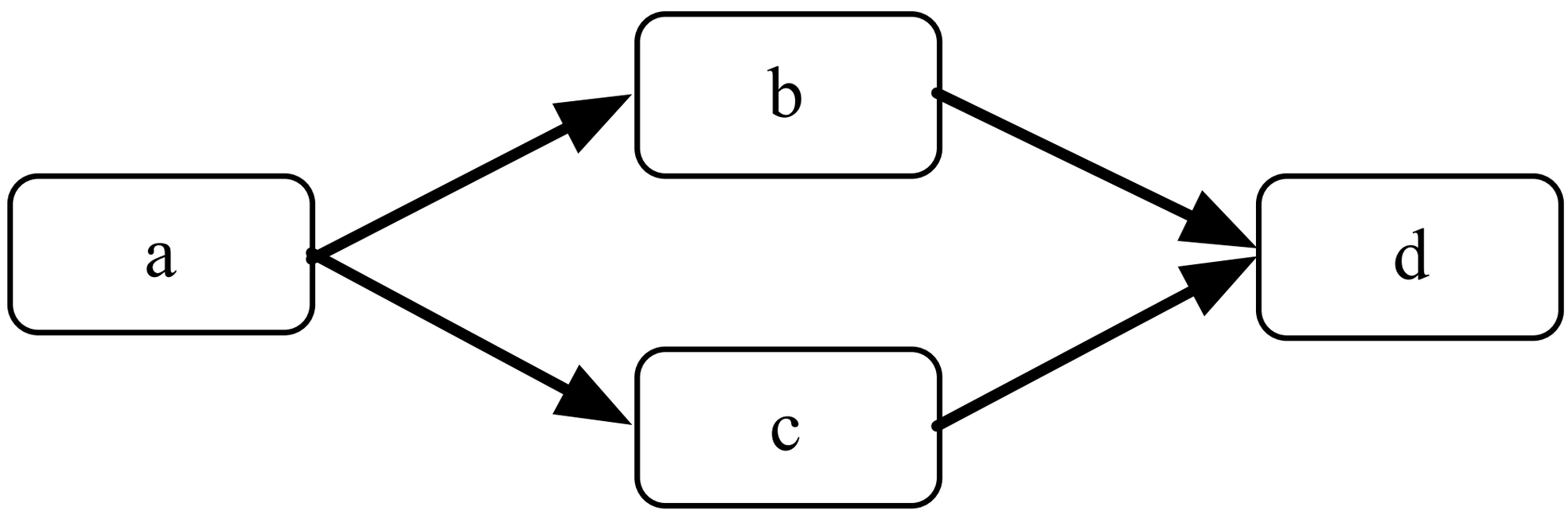
If “a” has 3 output ports and “c” has 3 input ports:



Expansion (only in dev branch, pre-beta version)

```
raft::map m;
/** example only **/
raft::kernel a, b, c, d;
m += a <= raft::kset(b,c) >= d;
```

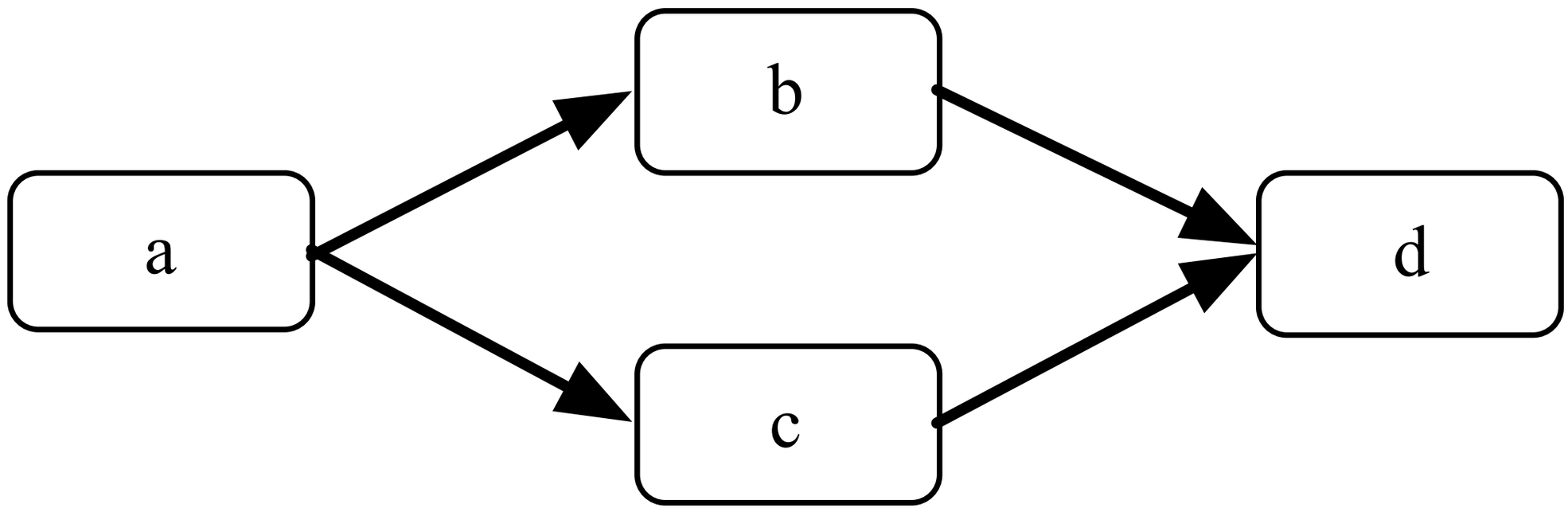
If “a” has 2 output ports and “d” has 2 input ports:



Expansion (only in dev branch, pre-beta version)

```
raft::map m;
/** example only **/
raft::kernel a, b, c, d;
m += a <= raft::kset(b,c) >= d;
```

If “a” has 2 output ports and “d” has 2 input ports:

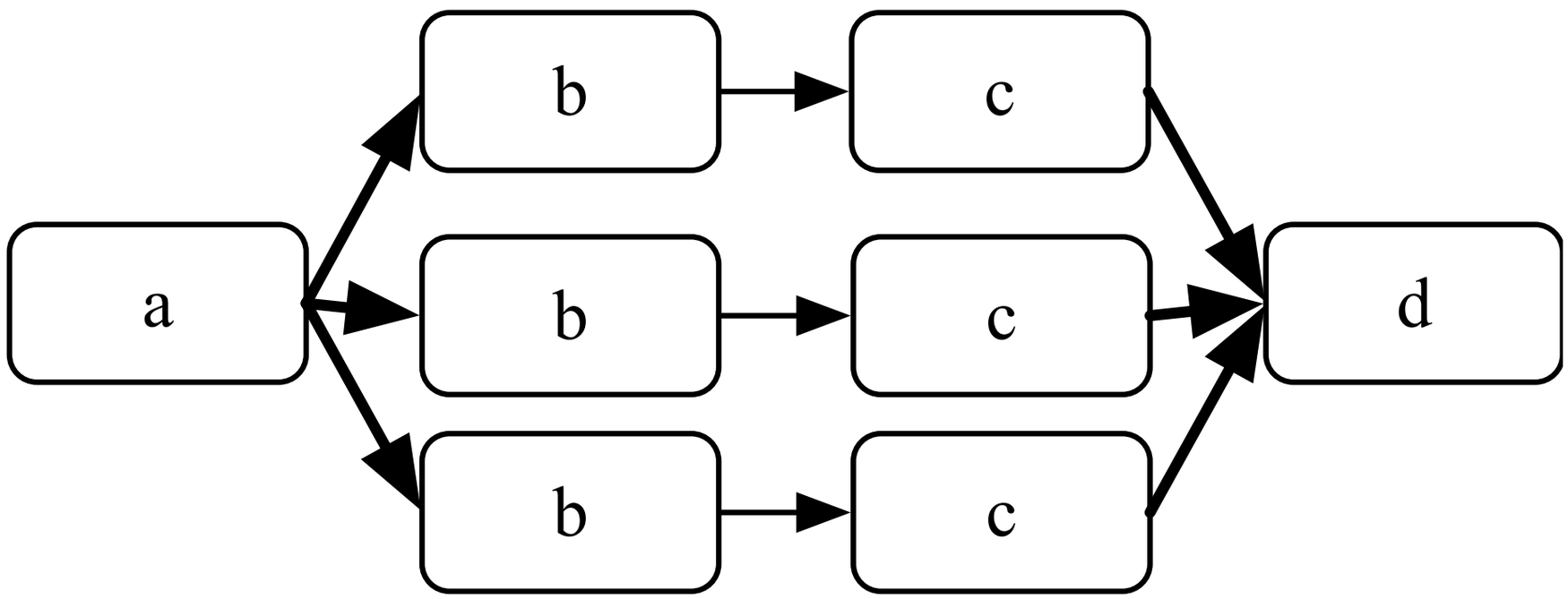



```
m += a <= raft::kset(b[ “x0” ][ “y0 ],c) >= d;
```

Build More Complicated Topologies

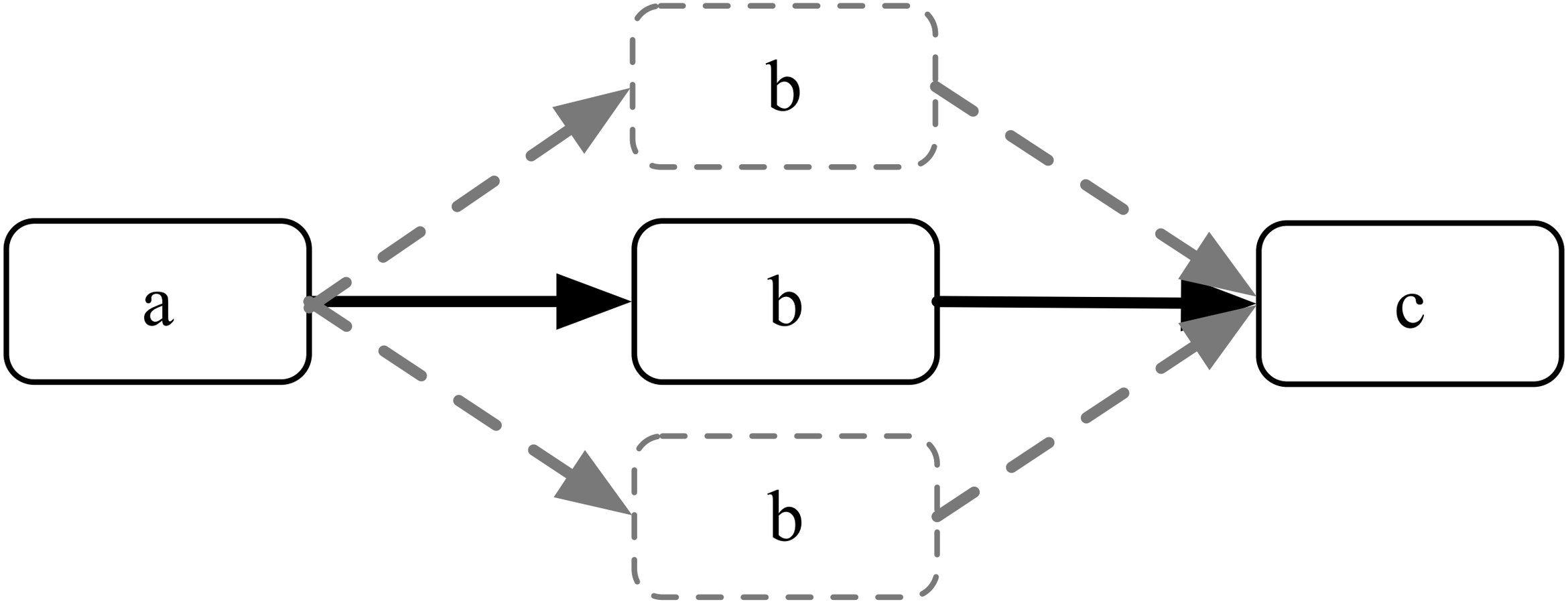
```
raft::map m;
/** example only **/
raft::kernel a, b, c, d;
m += a <= b >> c >= d;
```

RaftLib Turns Into



Parallelizing Out-of-Order Segments

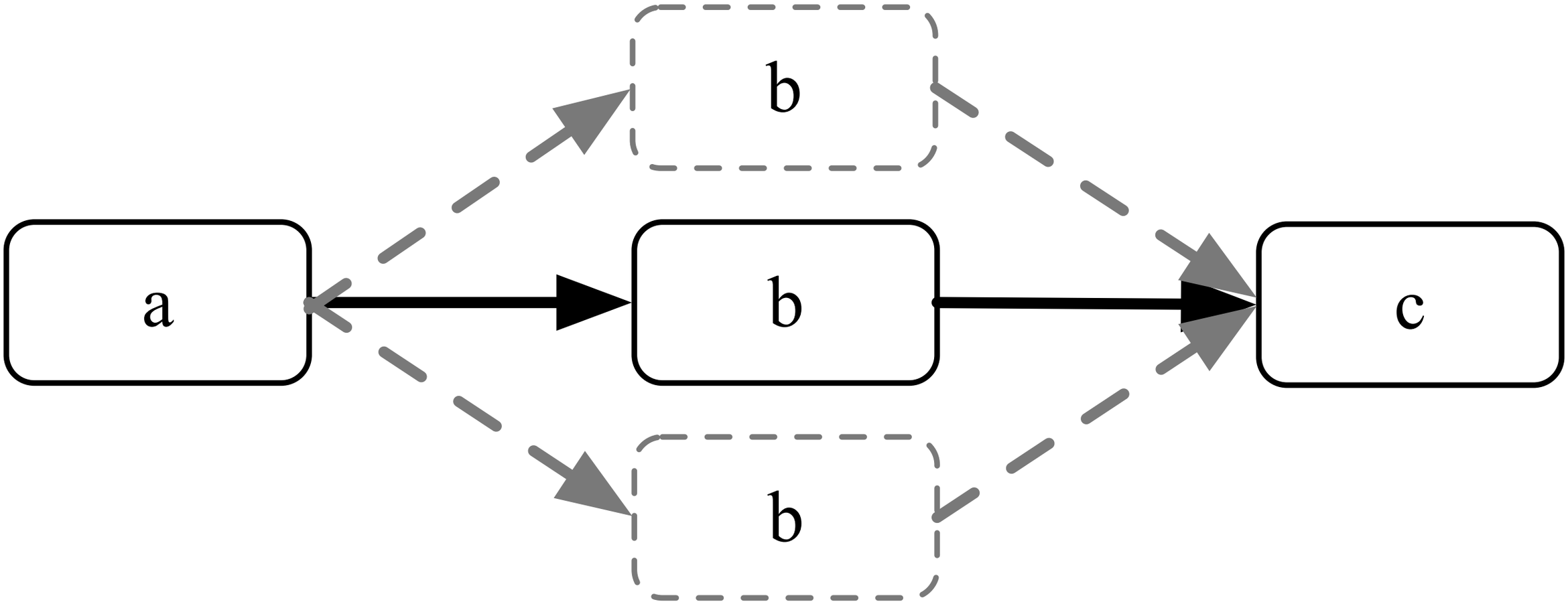
```
raft::map m;
/** example only **/
raft::kernel a, b, c;
m += a >> raft::order::out >> b >> raft::order::out >>  c;
```



Parallelizing Out-of-Order Segments

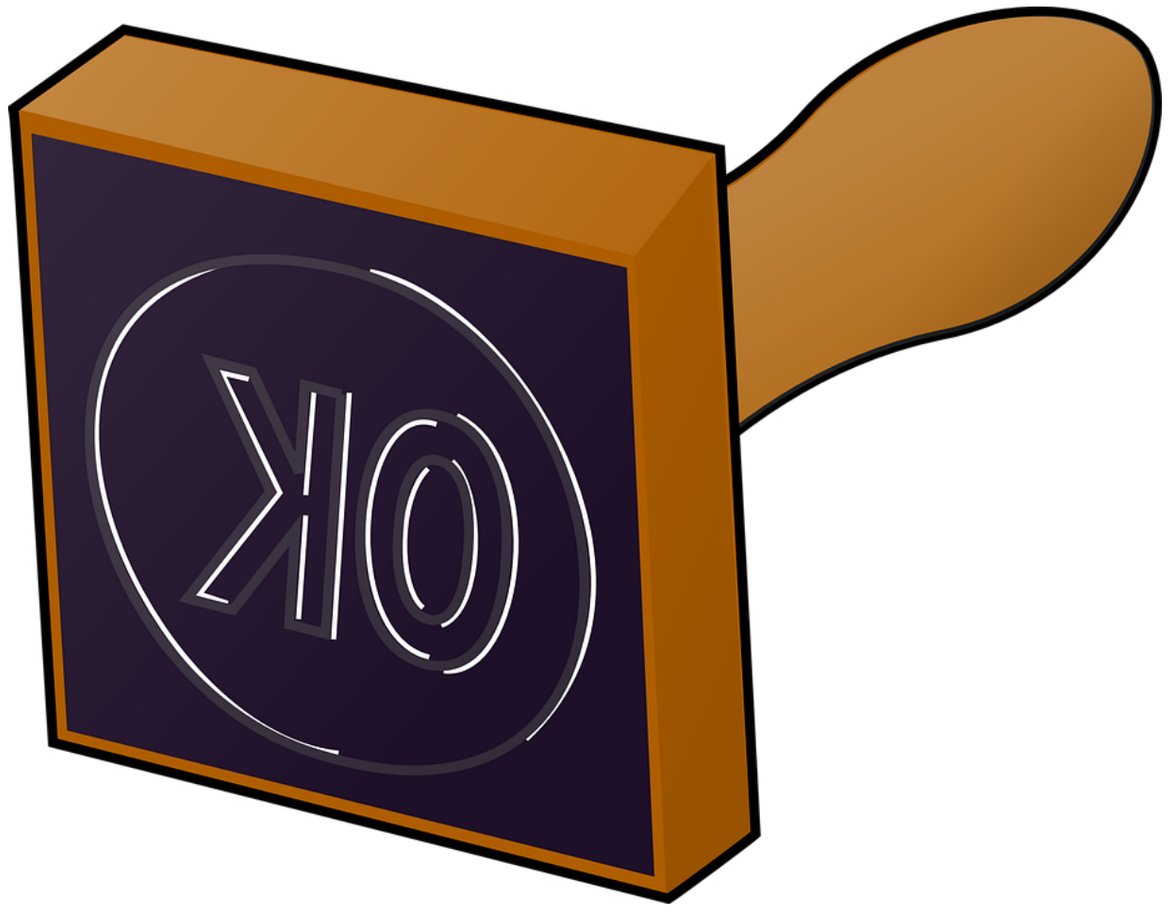
```
raft::map m;
/** example only **/
raft::kernel a, b, c;
m += a >> raft::order::out >> b >> raft::order::out >>  c;
```

Single Entry Point **Single Exit Point**



The Boiler Plate Stamping Machine

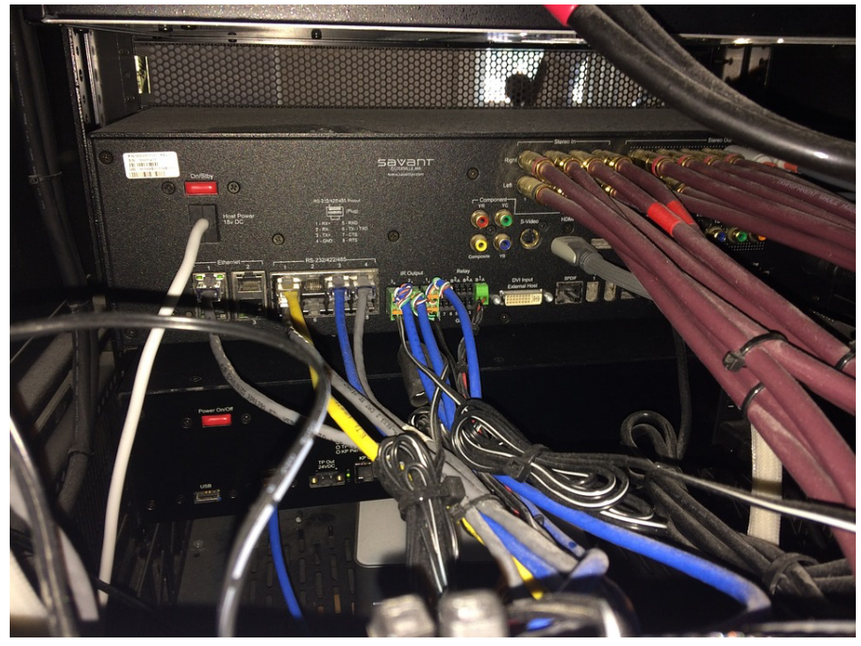
- Most runtimes do a lot of things for you, a lot of boilerplate..RaftLib is no different
- The graph structure combined with state encapsulation enable the runtime to do a bit more
- Where RaftLib shines is allocation, hiding all that memory work that takes so many keystrokes
- We also hide lots of other stuff too



Allocation Management

- With most threaded code, I'd have to initialize my own data structures, e.g. boost lock-free queue
- Managing dozens of these is onerous at best
- Errors in putting these allocations together are easy to make and often hard to find

What we think we code:



What others see it as:

Allocation High level

```
raft::map m;
/** example only **/
raft::kernel a, b;
m += a >> b;
/** here's where we are **/
m.exe()
```

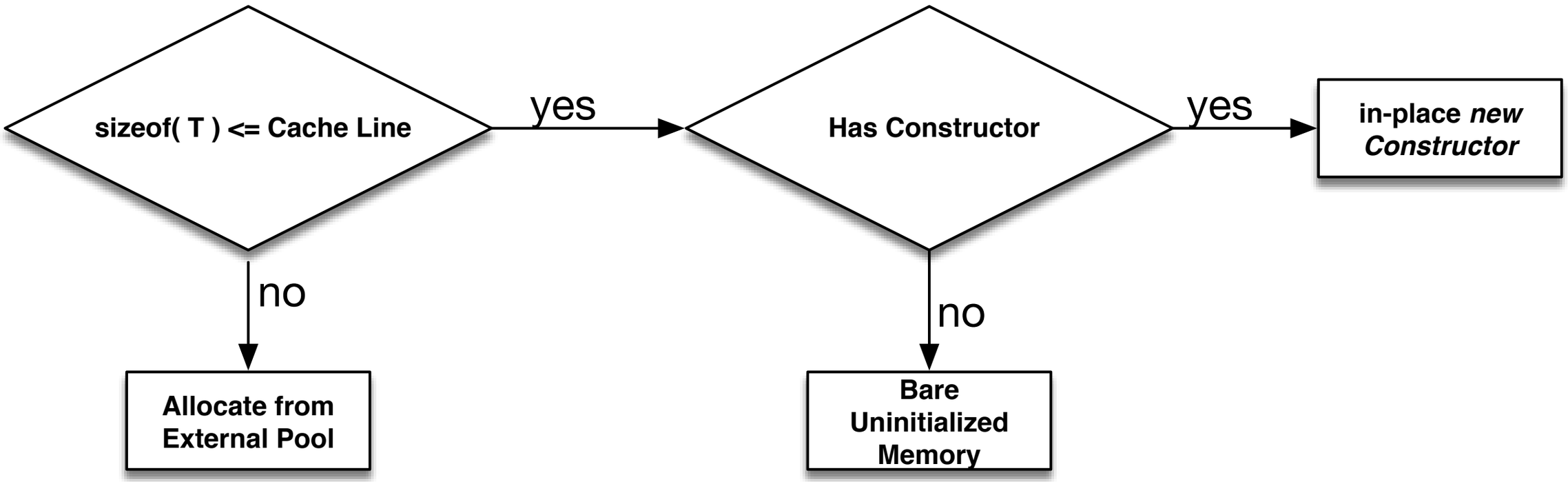
- assume “a” has a single output port (named out) of type *int*
- assume “b” has a single input port (named in) of type *int*

Steps

1. Check types of “in” and “out” dynamically
2. Decide where to run “a” and “b”
3. *Allocate memory for link between “in” and “out”*
4. *Link is ready to roll, threads are launched*

Allocation Mid level

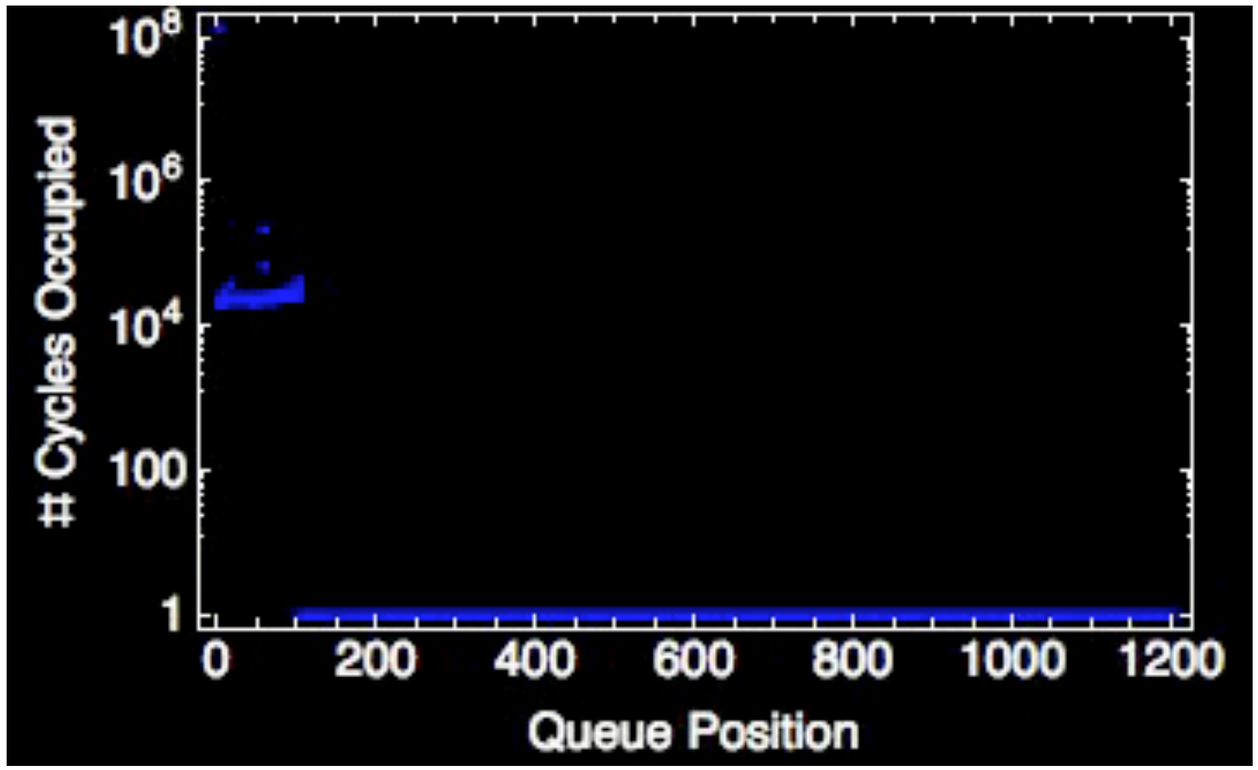
- FIFO between actors can be of many differing configurations
- RaftLib can automatically resize the buffer to make it bigger or smaller depending on service rates and queue occupancy
- Multiple types of FIFOs given type sizing (selected via templates at compile time)



Allocation Mid level

- Why do we need resizing FIFOs?
  - Runtime doesn’t know exactly what the output size of producer or what the consumption requirements of the producer are
  - Even when producer and consumer are rate-matched, they rates can be bursty

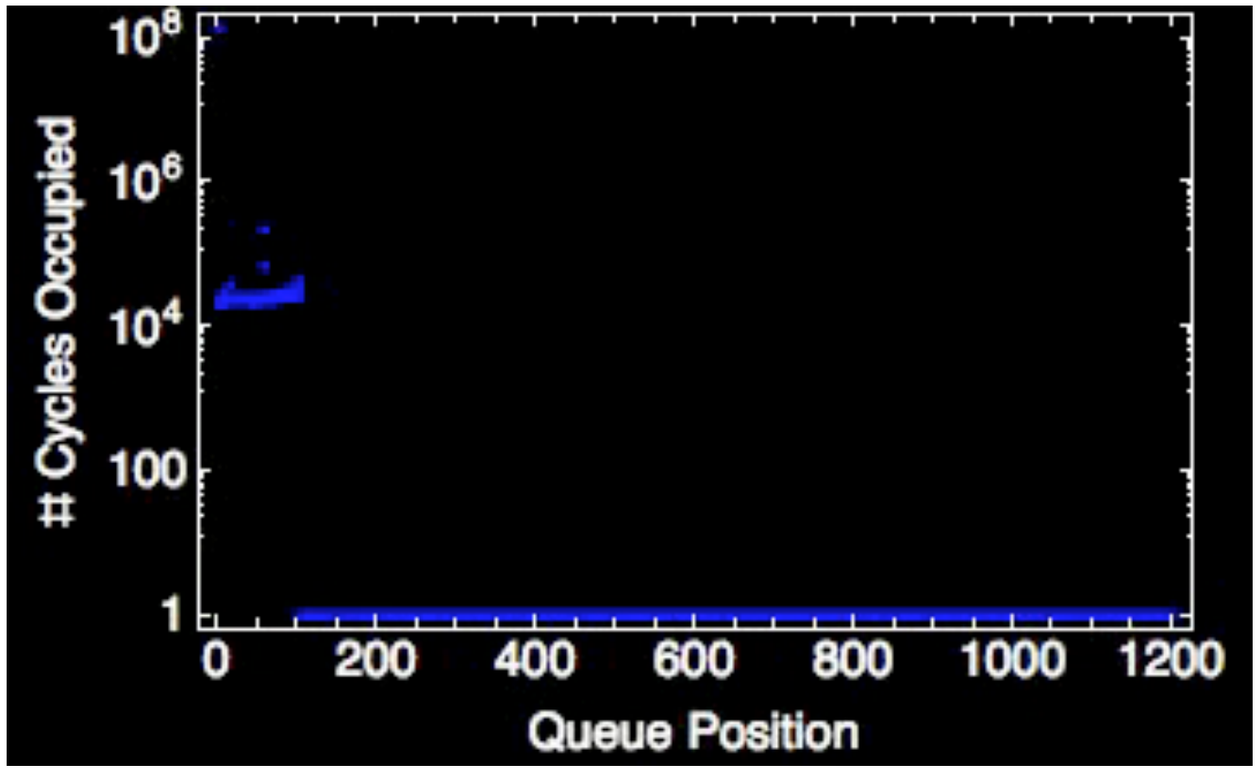
note: behavior at right is driven via exponential random number generator, it’s never this clean in real systems



Allocation Mid level

- Why do we need resizing FIFOs?
  - Runtime doesn’t know exactly what the output size of producer or what the consumption requirements of the producer are
  - Even when producer and consumer are rate-matched, they rates can be bursty

note: behavior at right is driven via exponential random number generator, it’s never this clean in real systems



Allocation Lower Level

- Allocation can be in lots of different locations: Heap, SHM, NV, custom, etc.
- Allocator factories created at class initialization for each extant port, can be added dynamically as well:

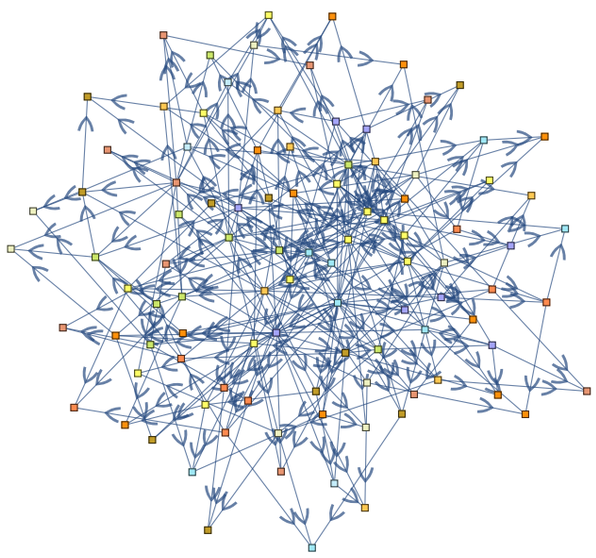
```
/** allocator factory map **/
std::map< Type::RingBufferType , instr_map_t* > const_map;

/** initialize some factories **/
const_map.insert( std::make_pair( Type::Heap , new instr_map_t() ) );

const_map[ Type::Heap ]->insert(
    std::make_pair( false /** no instrumentation **/,
                    RingBuffer< T, Type::Heap, false >::make_new_fifo ) );
const_map[ Type::Heap ]->insert(
    std::make_pair( true /** yes instrumentation **/,
                  RingBuffer< T, Type::Heap, true >::make_new_fifo ) );
….many more
```

Partitioning

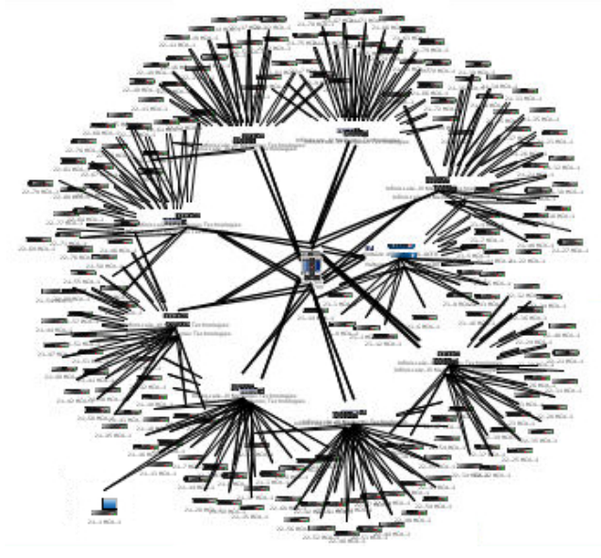
- Where do we run? Another had problem, we normally delegate to the OS. For perf, we usually want to do manually (see FIFO presentation yesterday).

- Last presentation we talked about using Scotch/Metis. RaftLib has since incorporated Sandia's Qthreads as well which can use *hwloc* for hardware aware scheduling.

Application



Application Assigned to Hardware



Why is Partitioning Important?

- With great abstraction comes great ignorance of reality

Why is Partitioning Important?

- With great abstraction comes great ignorance of reality

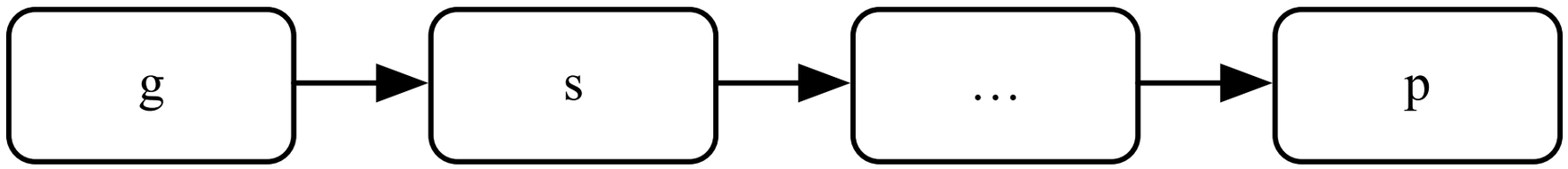
```
raft::map m;
/** make one sub kernel, this one will live on the stack **/
sub s( 1, 1, l_sub );
kernel_pair_t::kernel_iterator_type BEGIN, END;
auto kernels( m += rndgen >> s );
for( int i( 0 ); i < 1000; i++ )
{
    std::tie( BEGIN, END ) = kernels.getDst();
    kernels =
        ( m += (*BEGIN).get() >>
                    raft::kernel::make< sub >( 1, 1, l_sub ) );
}
std::tie( BEGIN, END ) = kernels.getDst();
m += (*BEGIN).get() >> p;
m.exe();
```

Why is Partitioning Important?

- With great abstraction comes great ignorance of reality

```
raft::map m;
/** make one sub kernel, this one will live on the stack **/
sub s( 1, 1, l_sub );
kernel_pair_t::kernel_iterator_type BEGIN, END;
auto kernels( m += rndgen >> s );
for( int i( 0 ); i < 1000; i++ )
{
    std::tie( BEGIN, END ) = kernels.getDst();
    kernels =
        ( m += (*BEGIN).get() >>
                raft::kernel::make< sub >( 1, 1, l_sub ) );
}
std::tie( BEGIN, END ) = kernels.getDst();
m += (*BEGIN).get() >> p;
m.exe();
```

Why is Partitioning Important?



- A thousand threads….not a good idea
- With Qthreads, we can now get performance even for ridiculous numbers of kernels and a decent partition.
  - 1000 kernels on raftlib + Qthreads = 0m2.907s (on this laptop)
  - 1000 kernels on raftlib + pthreads = never finishes (killed after an hour)

Build a Kernel

```
class akernel : public raft::kernel
{
public:
    akernel() : raft::kernel()
    {
        //add input ports
        input.addPort< /** type **/ >( "x0","x1", "x..." );
        //add output ports
        output.addPort< /** type **/ >( "y0", "y1", "y..." );
    }

    virtual raft::kstatus run()
    {
        /** get data from input ports **/
        auto &valFromX( input[ "x..." ].peek< /** type of "x..." **/ >() );
        /** do something with data **/

        const auto ret_val( do_something( valFromX ) );

        output[ "y..." ].push( ret_val );

        input[ "x..." ].unpeek();
        input[ "x..." ].recycle();
        return( raft::proceed /** or stop **/ );
    }
};
```

Receiving Data

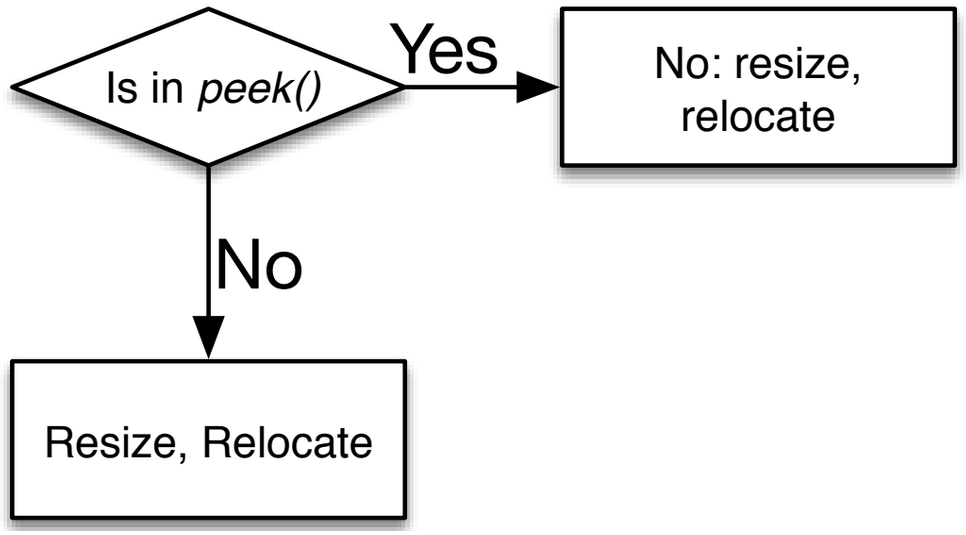
```
/**
 * return reference to memory on
 * in-bound stream
 */
template< class T >
T& peek( raft::signal *signal = nullptr )

template< class T >
autorelease< T, peekrange > peek_range( const std::size_t n )
```

- *Returns object with "special access to stream"*
- *Operator [ ] overload returns auto pair*
- *Direct reference as in peek() for each element*

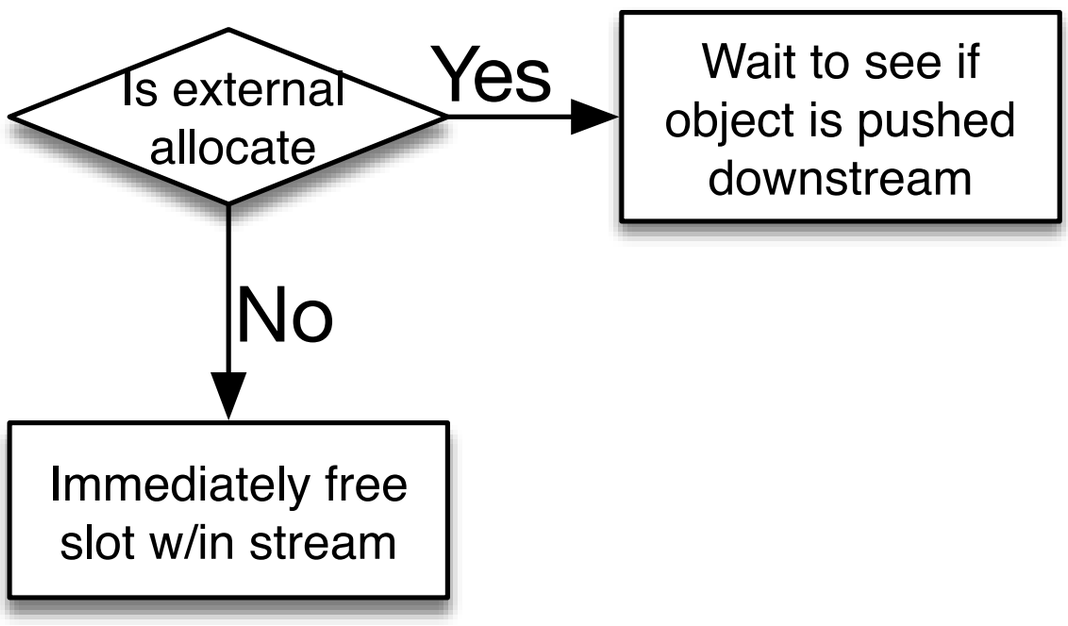

```
template < class T > struct autopair
{
   T                &ele;
   Buffer::Signal &sig;
};
```

Receiving Data

```
/**
 * necessary to tell inbound stream we're done
 * looking at it
 */
virtual
void unpeek()
```




```
/**
 * free up index in fifo, lazily deallocate large objects
 */
void
recycle( const std::size_t range = 1 )
```



The Dangling Reference

- There's a slight problem with peeking at objects followed by calling recycle on the object's port
- A memory leak could result, so just in case RaftLib implements a simple garbage collector to lazily erase/deconstruct dangling references

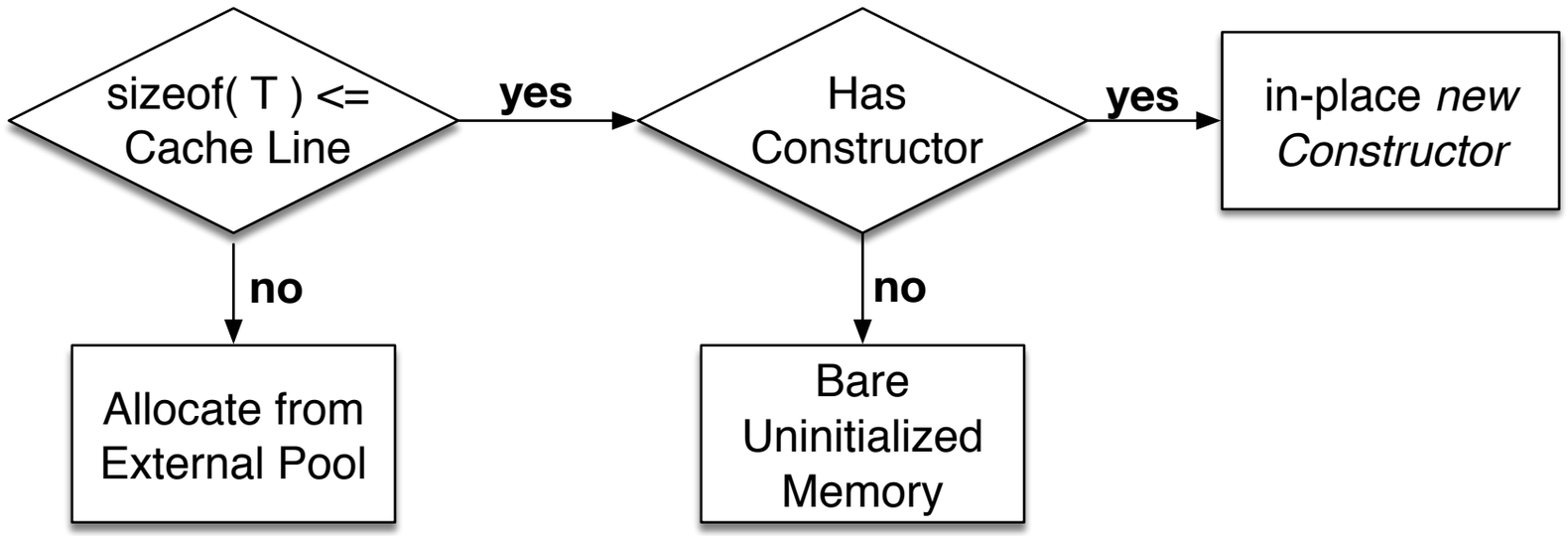
```
auto &mem( input[ "in" ].peek< obj_t >() );

/** do something with mem **/

input[ "in" ].recycle();

/** reference still valid, gc at end of function **/
```

Receiving Data

```
/**
 * these pops produce a copy
 */
template< class T >
void pop( T &item, raft::signal *signal = nullptr )

template< class T > using pop_range_t =
 std::vector< std::pair< T , raft::signal > >;

template< class T >
void pop_range( pop_range_t< T >  &items,
                const std::size_t n_items )

/**
 * no copy, slightly higher overhead, "smart object"
 * implements peek, unpeek, recycle
 */
template< class T >
autorelease< T, poptype > pop_s()
```

Sending Data

```
/** in-place allocation **/
template < class T,
           class ... Args >
T& allocate( Args&&... params )
```




```
/** in-place alloc of range for fundamental types **/
template < class T >
auto allocate_range( const std::size_t n ) ->
        std::vector< std::reference_wrapper< T > >
```

Sending Data

```
/** release data to stream **/
virtual
void send( const raft::signal = raft::none )

/** release data to stream **/
virtual
void send_range( const raft::signal = raft::none )

/** oops, don't need this memory **/
virtual void deallocate()
```

Sending Data

```
/** multiple forms **/
template < class T >
void push( const T &item, const raft::signal signal = raft::none )
```

```
/** insert from container within run() function to stream **/
template< class iterator_type >
void insert(    iterator_type begin,
                iterator_type end,
                const raft::signal signal = raft::none )
```

Progress Since CPPNow2016

- Integrated Qthreads with *hwloc* support
- Rewrite of template language (essentially the stream operators combined with manip modifiers make a full grammar)
- Wiki documentation far more complete
- Thanks to contributors more bugs fixed and Windows 10 support

Comments

- Slowly moving to Beta
- Many issues coming in from alpha users (thanks!)
- More users we have, the better it'll get
- Beta release target likely December 2017 given current rate
- Always looking for contributors

Interactive

- What are somethings we can do to make it better
- What would make you use it?
- Lets walk through some code examples if we have time… black screen with green text, yay :)

REFERENCES / RESOURCES

my website

http://www.jonathanbeard.io

slides at

https://goo.gl/hsTgJF

project page

raftlib.io