RAFTLIB

Presented by: Jonathan Beard
To: C++Now 2016

RAFTLIB

Alternate Titles:

This thing I started on a plane

RAFTLIB

Alternate Titles:

This thing I started on a plane

What's this RaftLib Thingy?

RAFTLIB

Alternate Titles:

This thing I started on a plane

What's this RaftLib Thingy?

OMG, Another Threading Library...

RAFTLIB

Alternate Titles:

This thing I started on a plane

What's this RaftLib Thingy?

OMG, Another Threading Library…

Why I hate parallel programming

RAFTLIB

Alternate Titles:

This thing I started on a plane

What's this RaftLib Thingy?

OMG, Another Threading Library...

Why I hate parallel programming

A self help guide for pthread anxiety

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 😎

ABOUT ME

my website

http://www.jonathanbeard.io

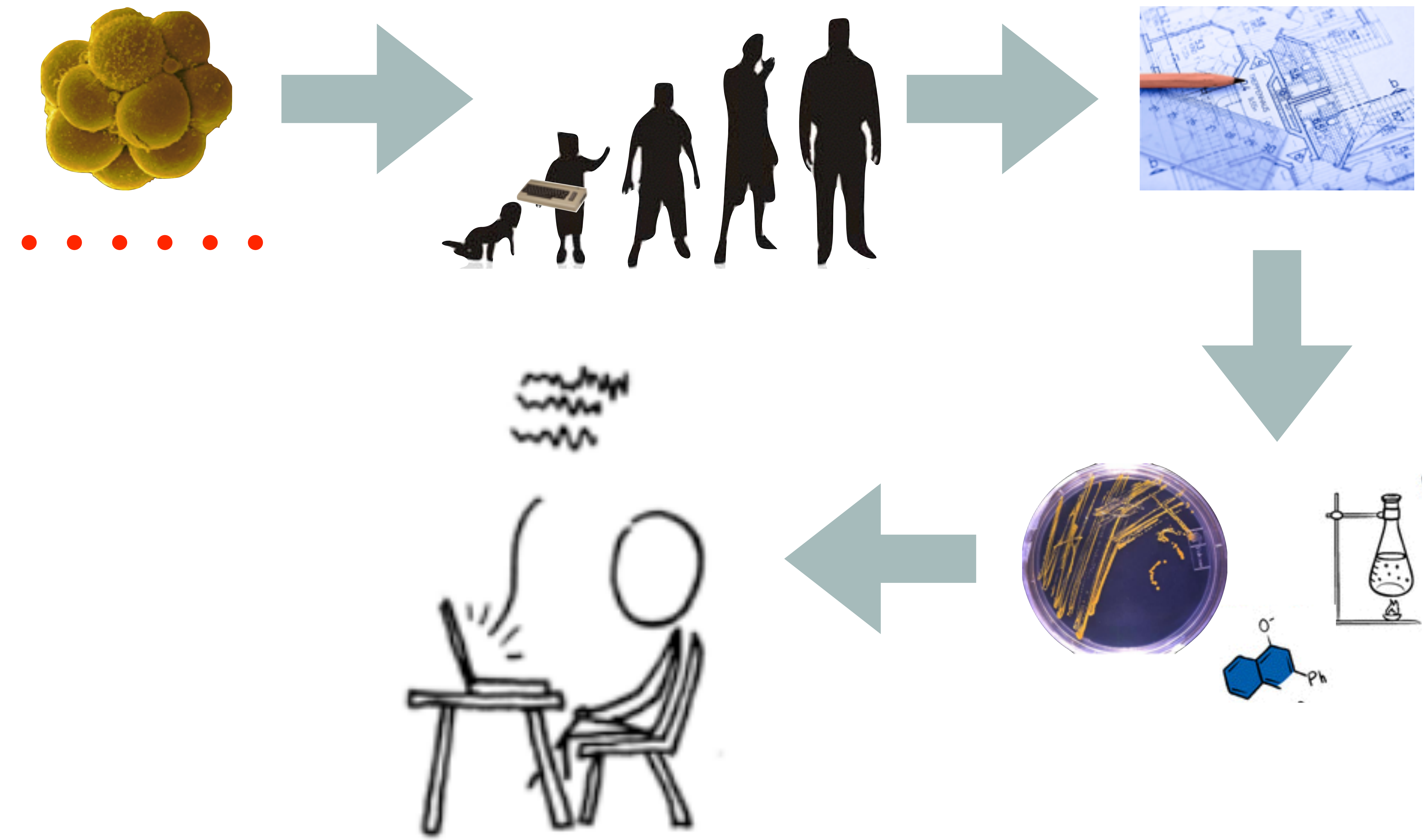
slides at

http://goo.gl/cwT5UB

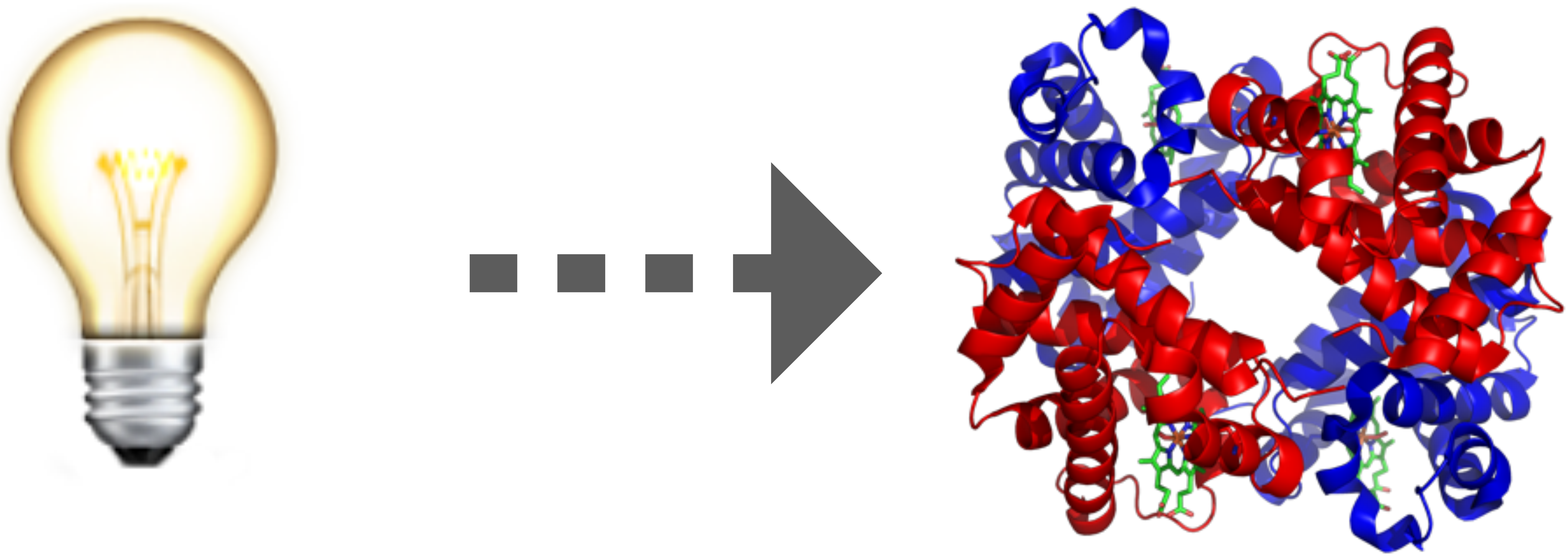
project page

raftlib.io

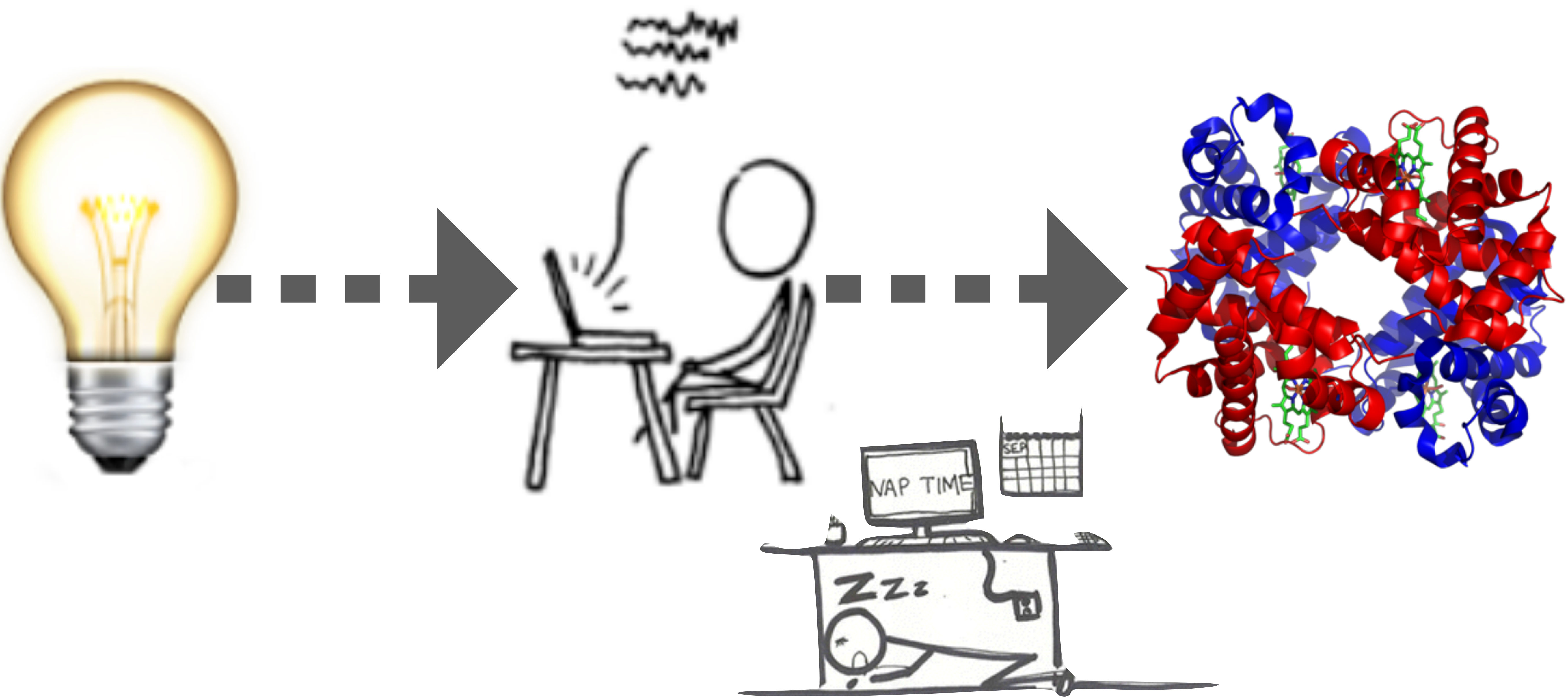
WHERE I STARTED



WHERE I STARTED



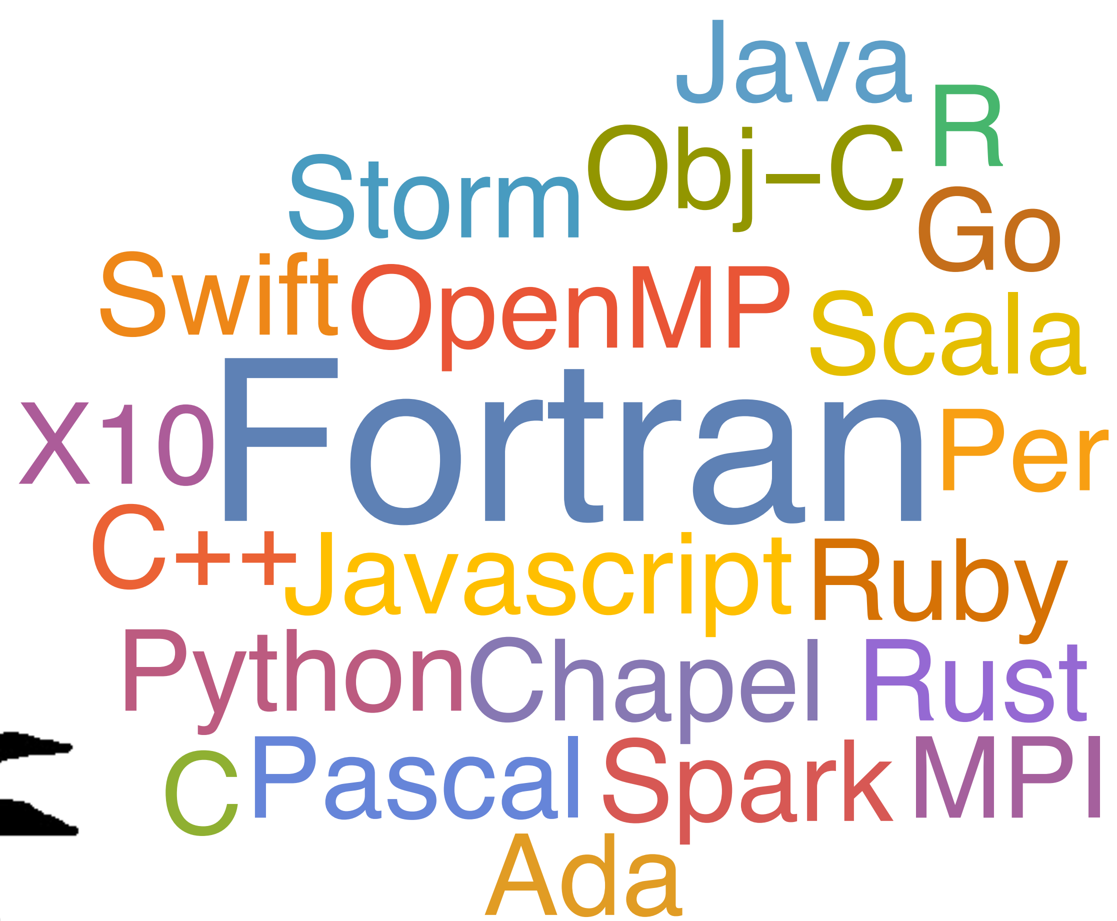
WHERE I STARTED



WHERE I STARTED

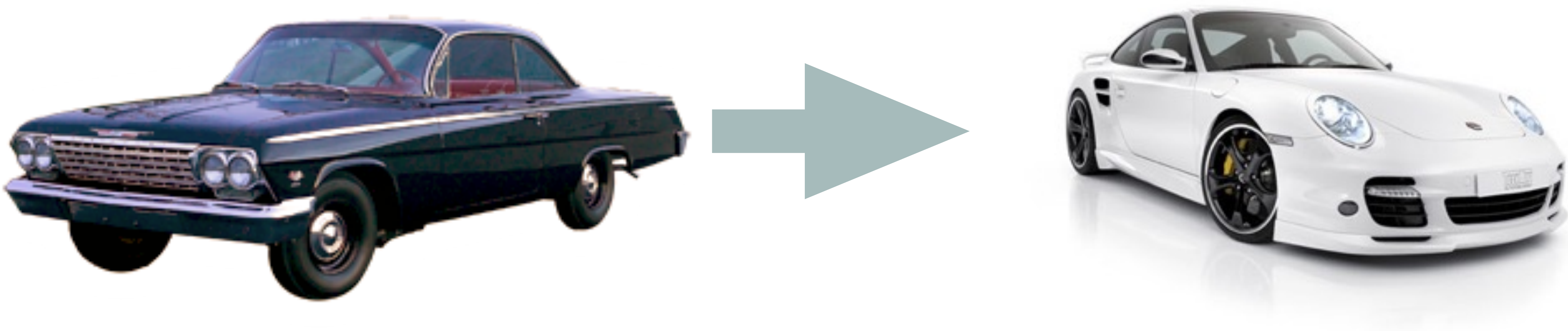
```
const uint8_t fsm_table[STATES][CHARS]=
   {
      /* 0 - @        */ {[9]=0x21},
      /* 1 - NAME     */ {[0 ... 8]=0x11 ,[11]=0x32},
      /* 2 - SEQ      */ {[1 ... 2]=0x42 ,[5]=0x42,[8]=0x42,[11]=0x13},
      /* 3 - \n       */ {[1 ... 2]=0x42 ,[5]=0x42,[8]=0x42,[10]=0x14},
      /* 4 - NAME2    */ {[0 ... 8]=0x54 ,[11]=0x15},
      /* 5 - SCORE    */ {[0 ... 10]=0x65,[11]=0x16},
      /* 6 - SEND/SO */ {[9] = 0x71,[11] = 0x16}
   };
```

NECESSITY DRIVES IDEAS

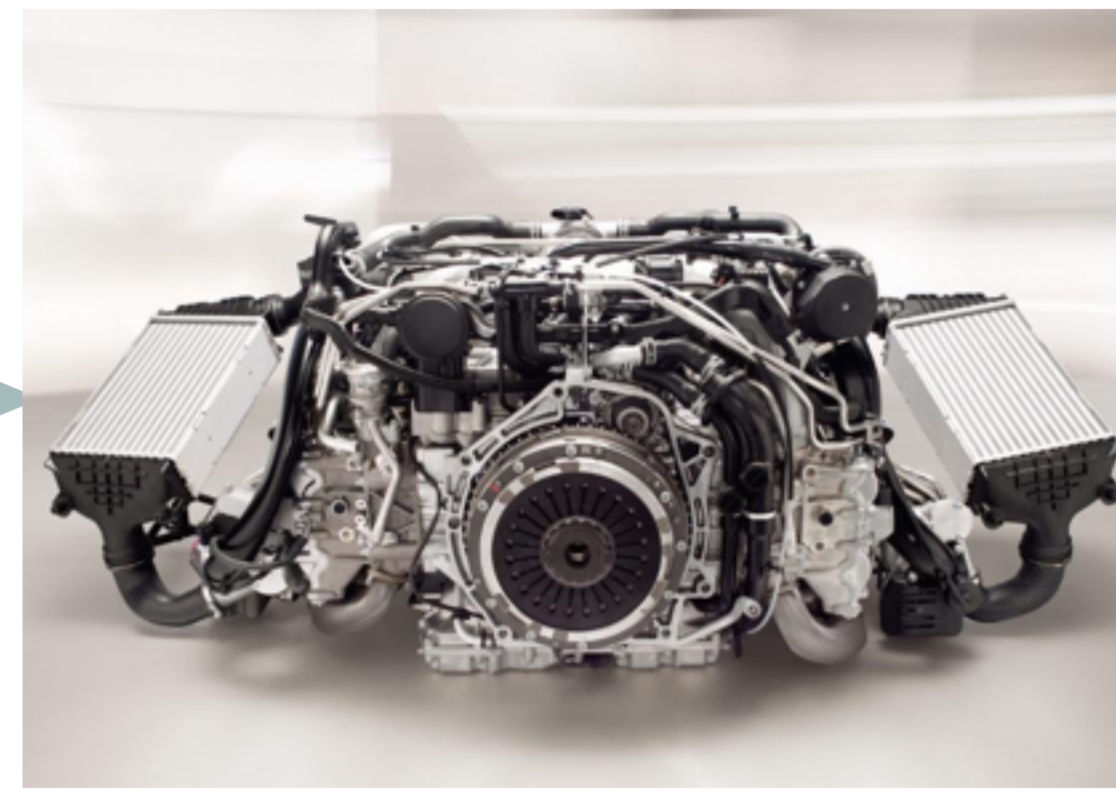


NECESSITY DRIVES IDEAS

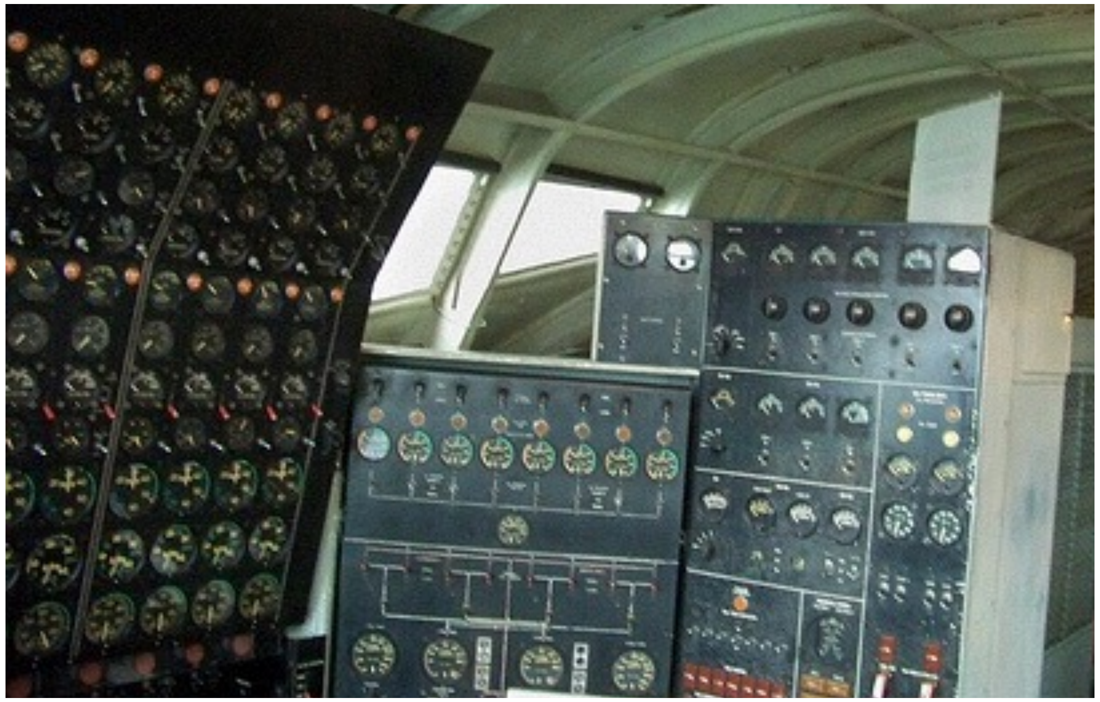
AN (PERHAPS BAD) ANALOGY



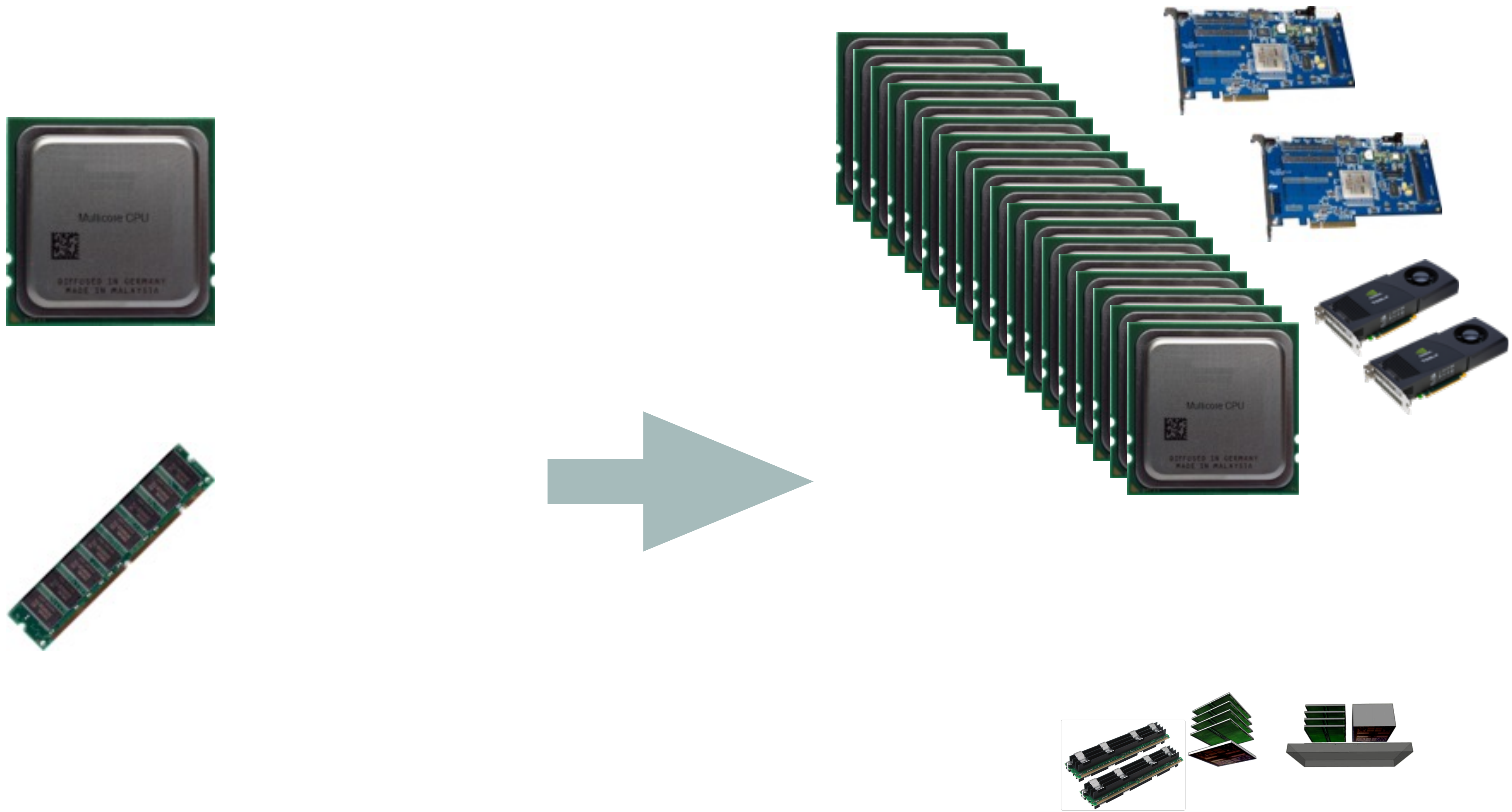
AN (PERHAPS BAD) ANALOGY



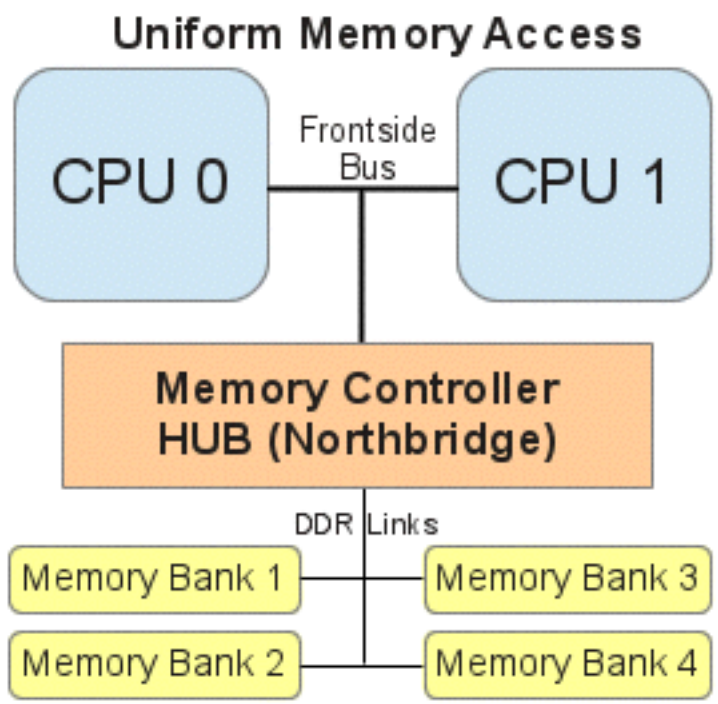
AN (PERHAPS BAD) ANALOGY



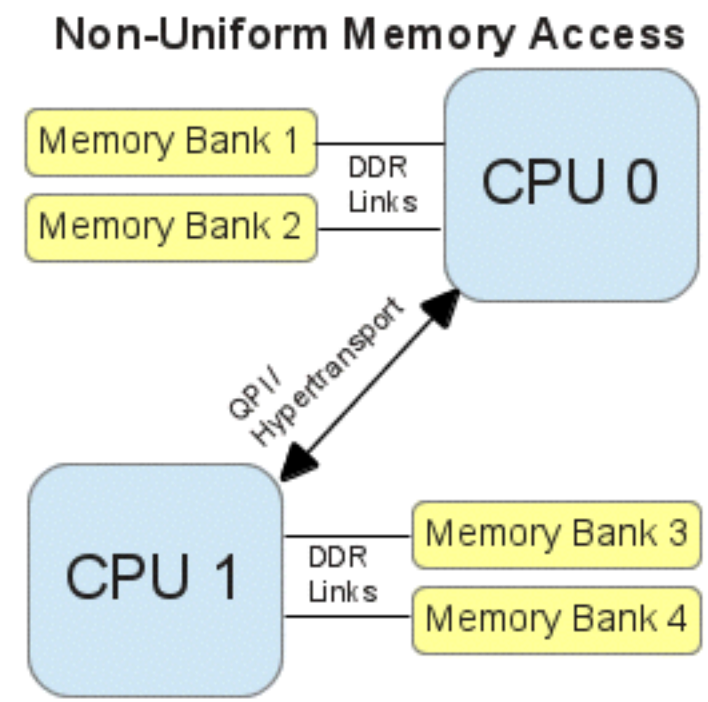
ANALOGY PART TWO



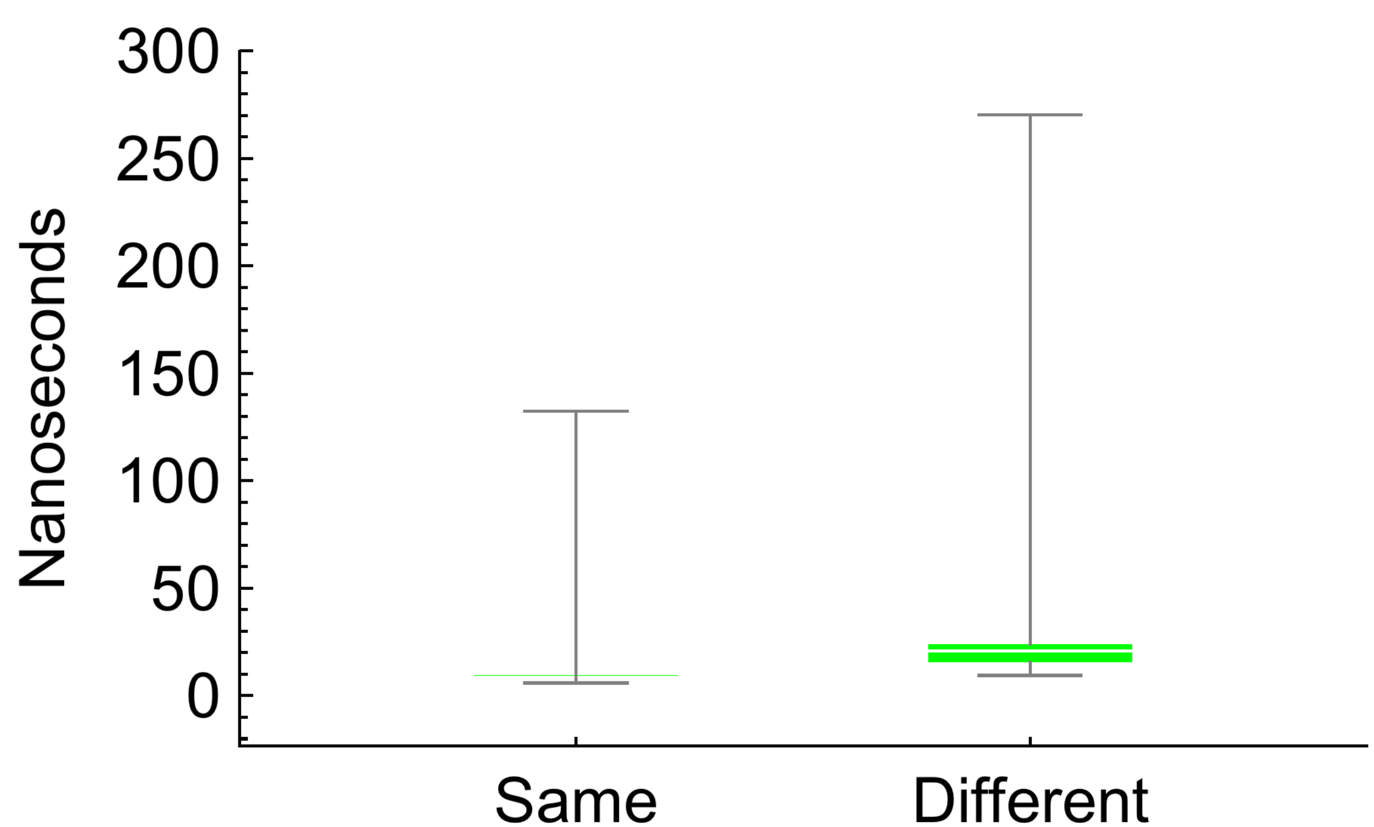
TOPOLOGY



Uniform Memory Access
CPU 0
Frontside Bus
CPU 1
Memory Controller HUB (Northbridge)
DDR Links
Memory Bank 1
Memory Bank 2
Memory Bank 3
Memory Bank 4



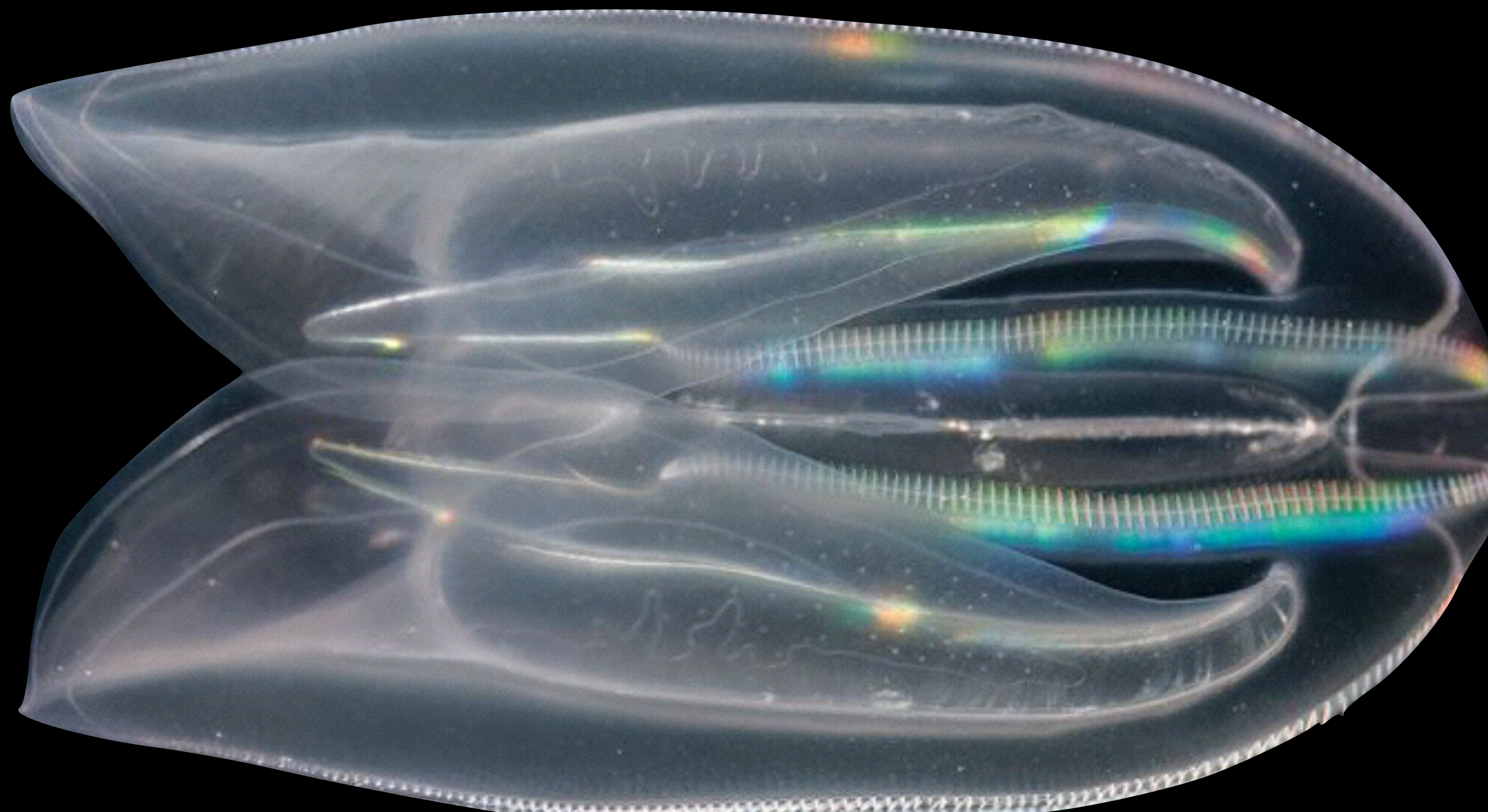
Non-Uniform Memory Access
Memory Bank 1
Memory Bank 2
DDR Links
CPU 0
QPI / Hypertransport
CPU 1
DDR Links
Memory Bank 3
Memory Bank 4



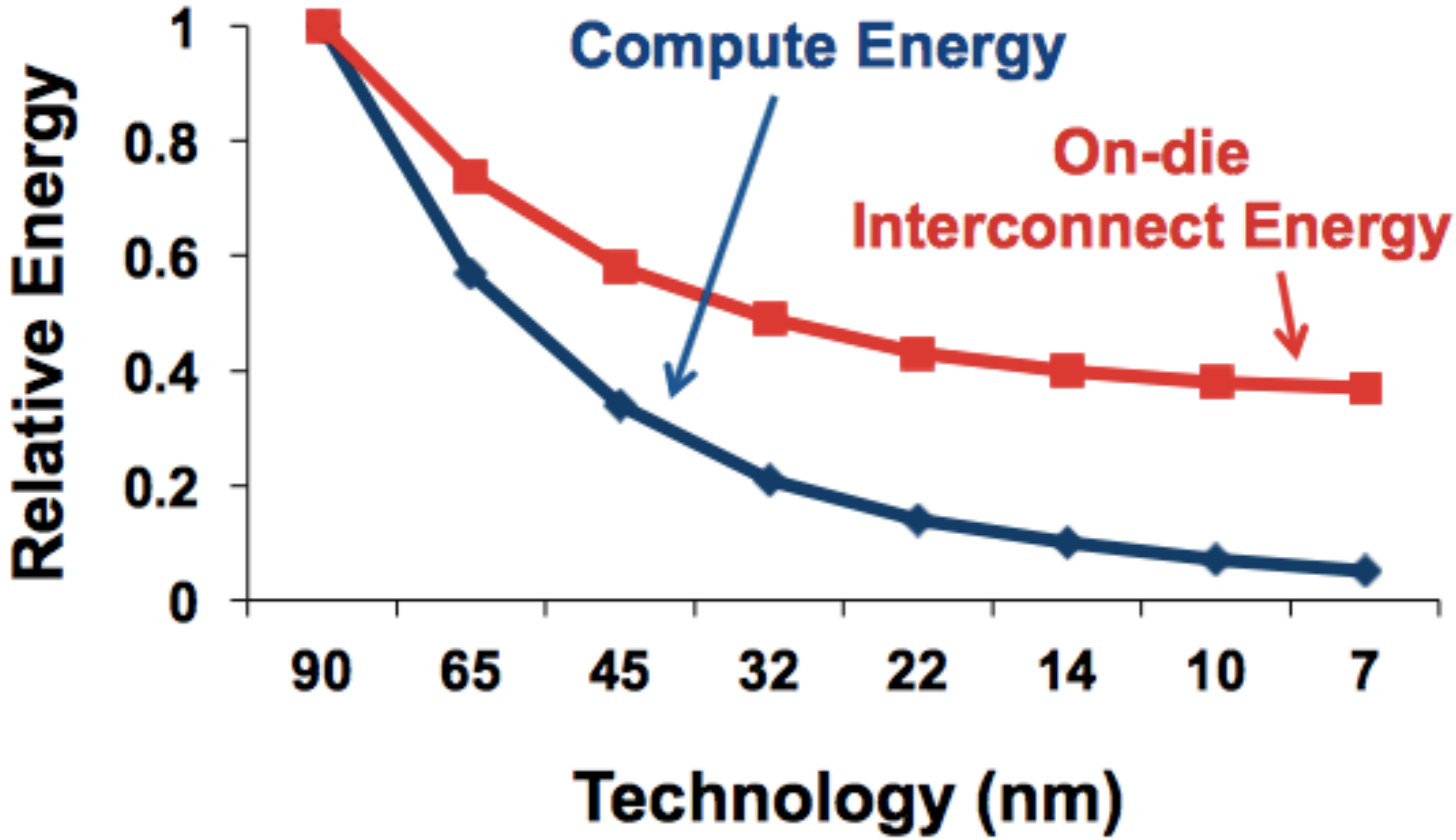
Nanoseconds
300
250
200
150
100
50
0
Same
Different

THE JELLYFISH

THE FIRST ORGANISM TO OVERLAP ACCESS AND EXECUTION

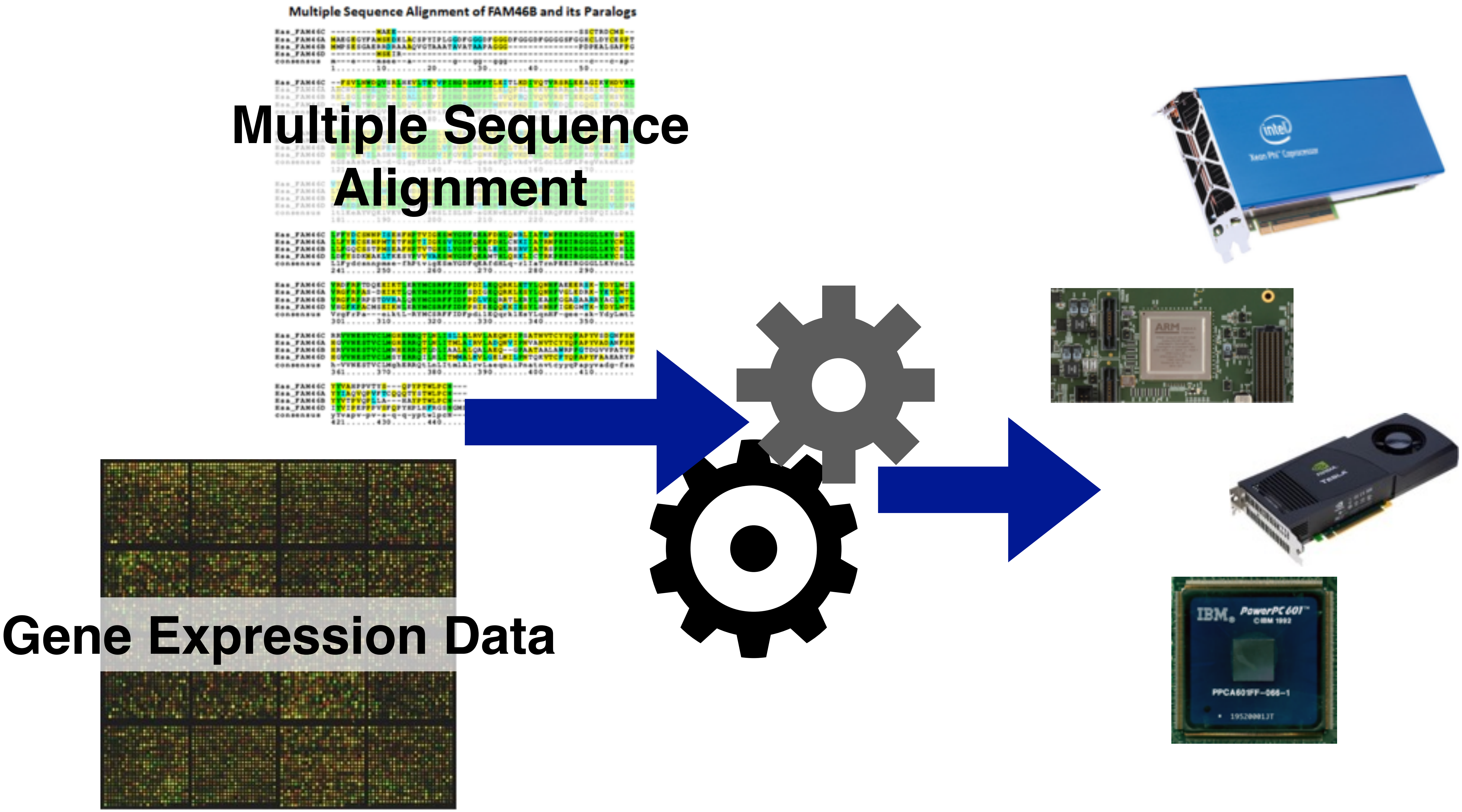


DATA MOVEMENT DOMINATES

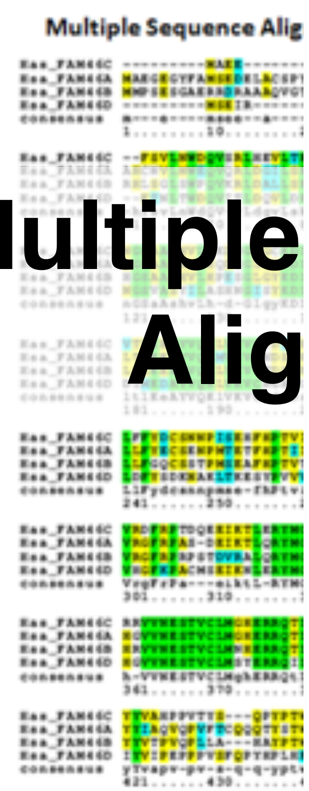


Source: Shekhar Borkar, Journal of Lightwave Technology, 2013

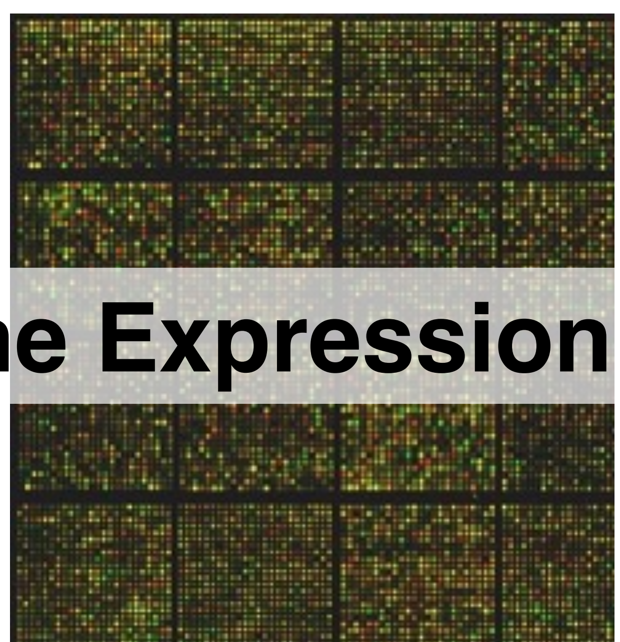
I SHOULDN'T HAVE TO CARE



I SHOULDN'T HAV



Multiple
Alig



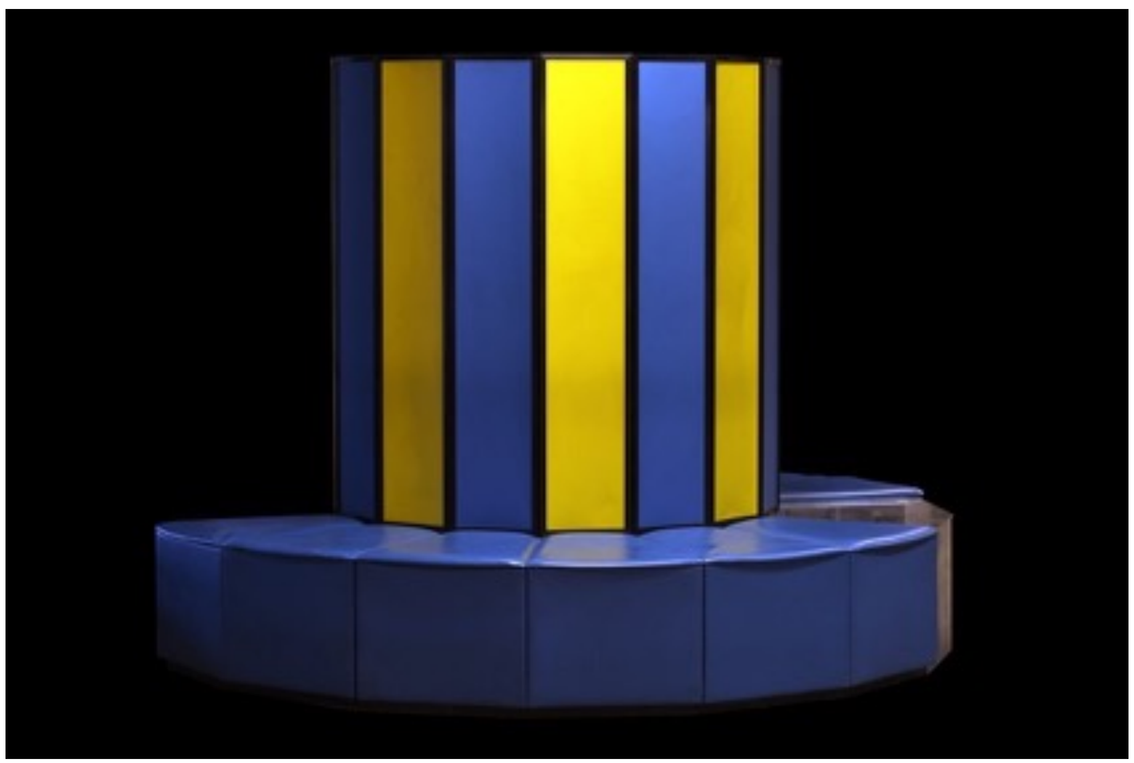
Gene Expression

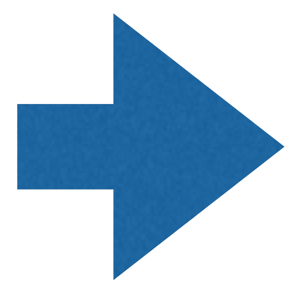
HARDWARE

1984

Cray X-MP/48

$19 million / GFLOP

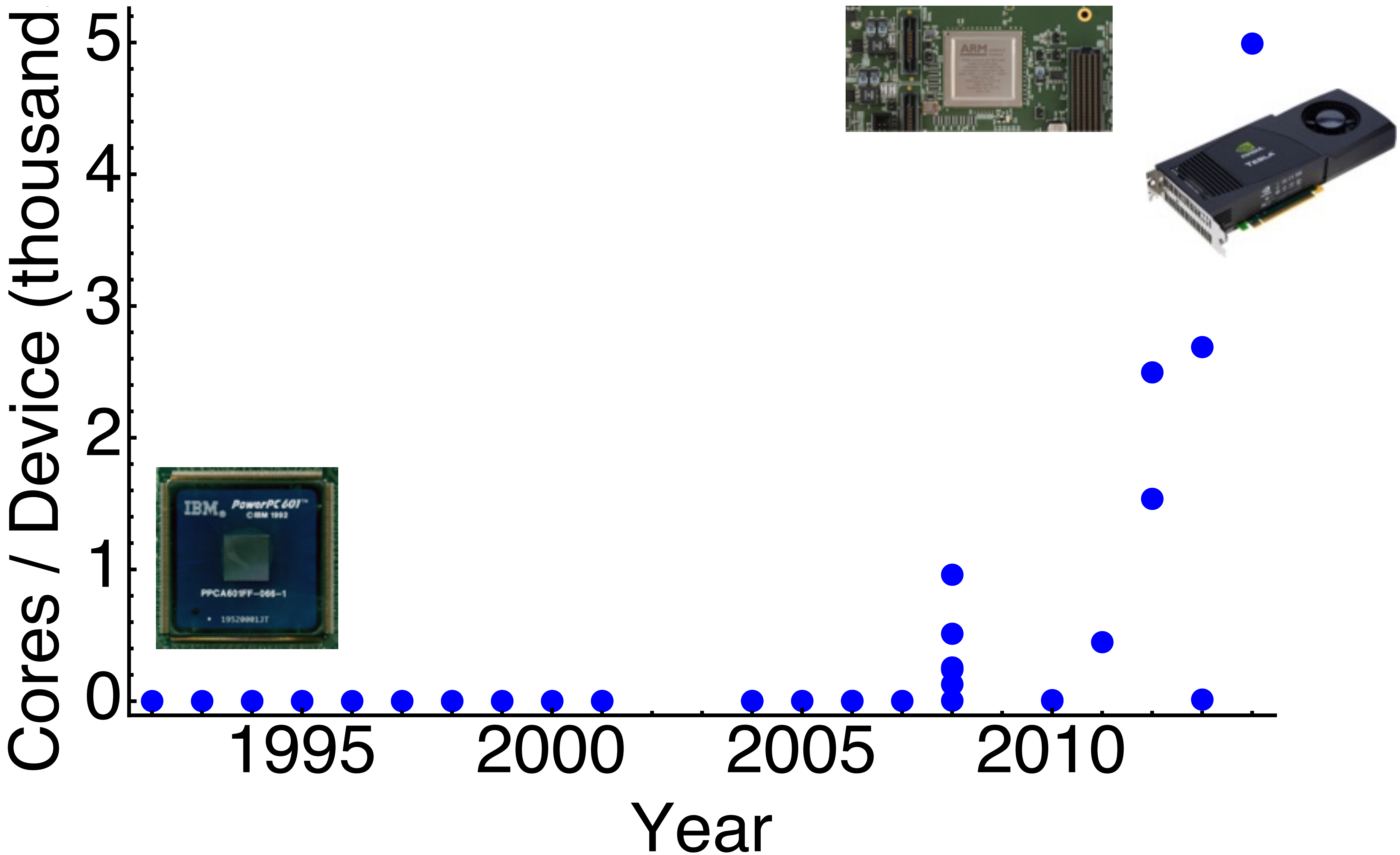




2015

$.08 / GFLOP

CORES PER DEVICE



FINANCIAL INCENTIVE

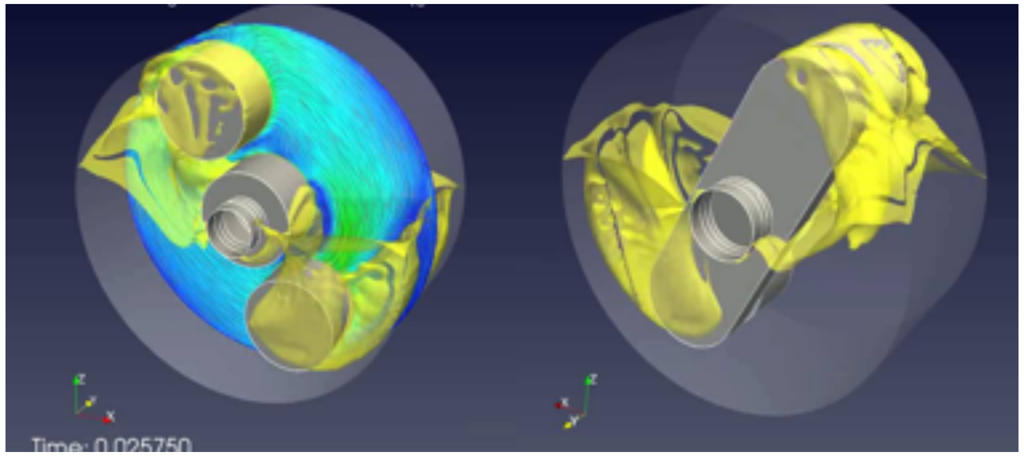
Sequential JS: $4-7/line

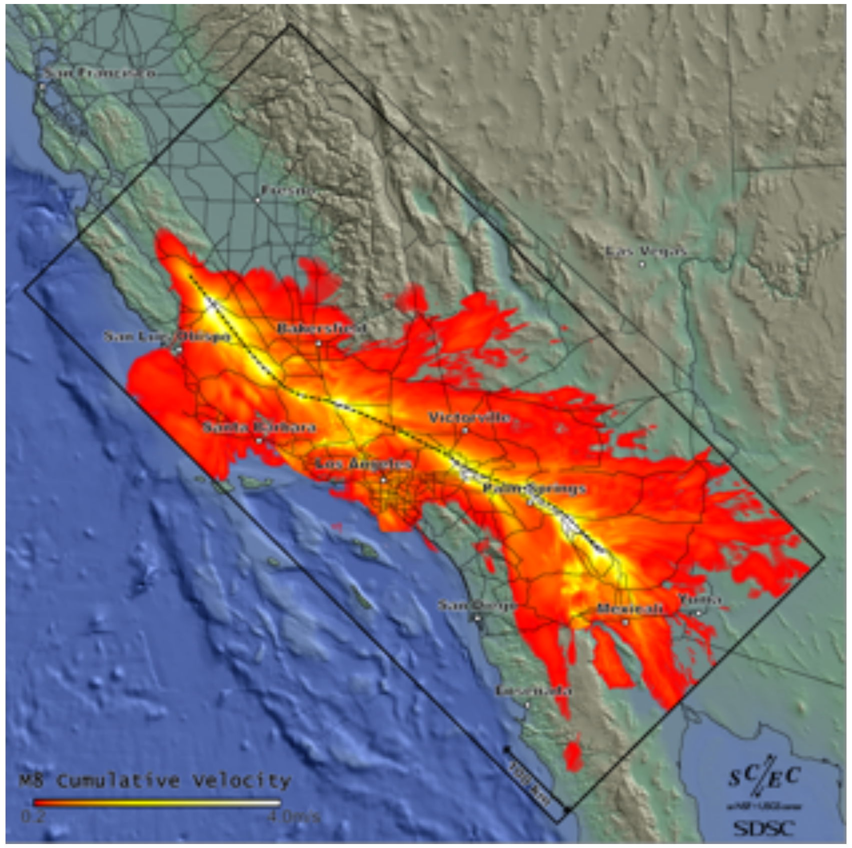
Sequential Java: $5-10/line

Embedded Code: $30-50/line

HPC Code: $100/line

WHY YOU SHOULD CARE



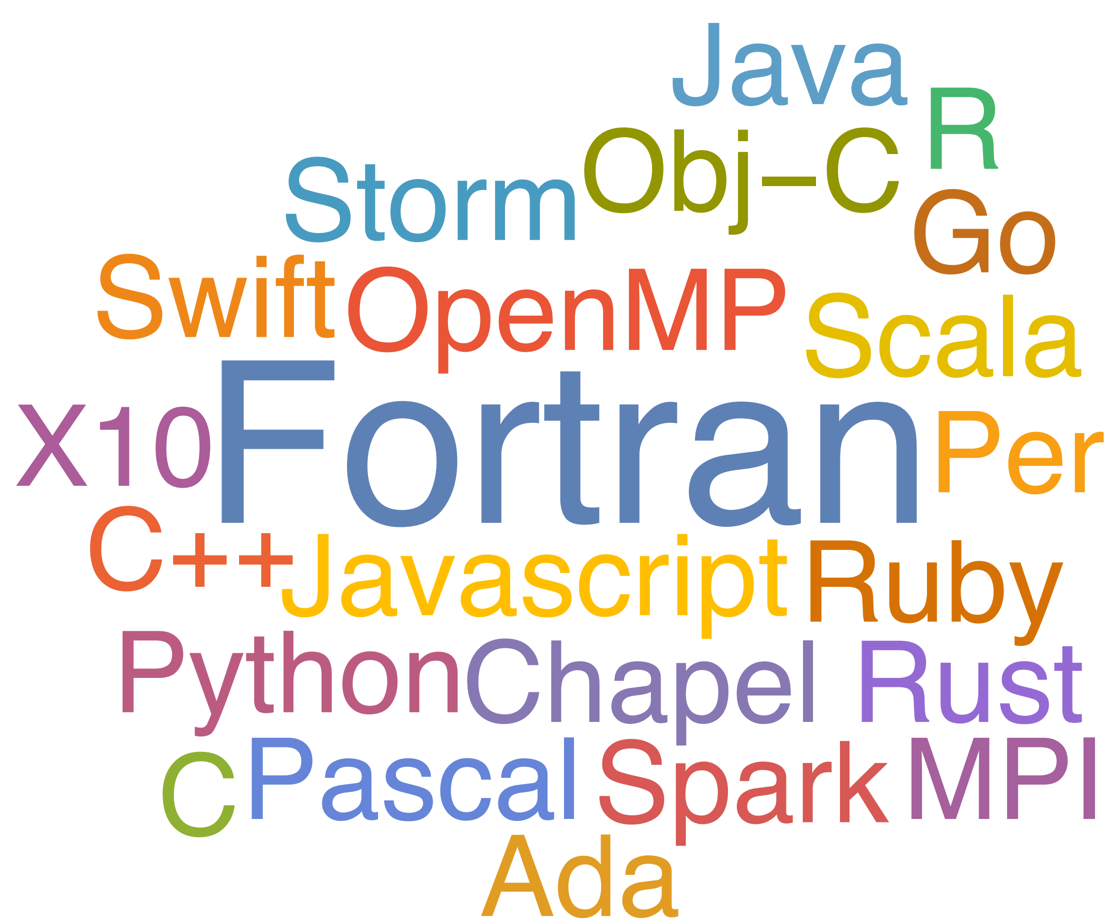


PRODUCTIVITY / EFFICIENCY AN EQUALIZER

- Titan SC estimates for porting range from 5, to > 20 million USD
- Most code never really optimized for machine topology, wasting $$ (time/product) and energy
- **Getting the most out of what you have is an equalizer**

amazon web services™

Google Cloud Platform



HAVES AND HAVE NOTS

Gov't / Big Business

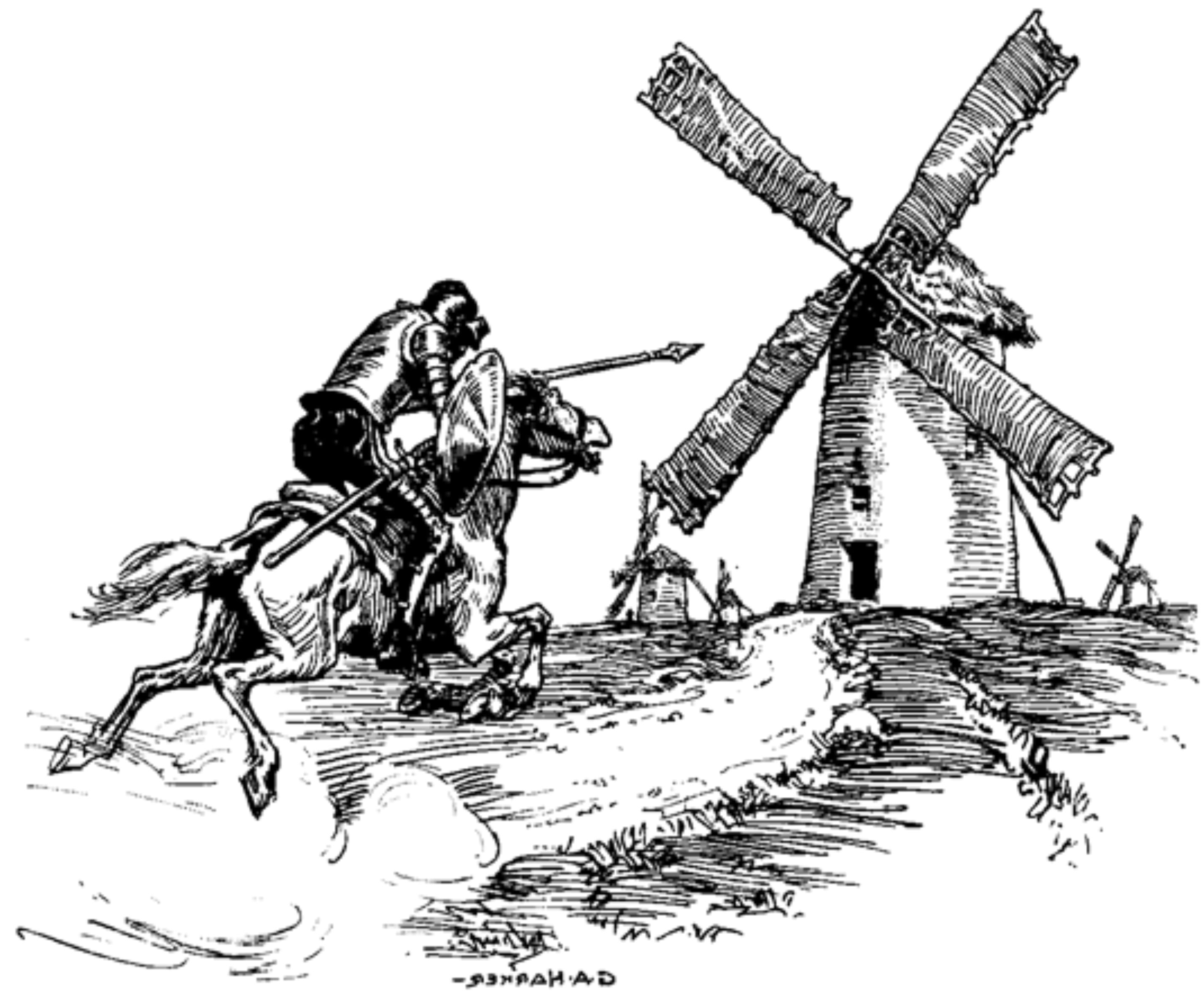
- *Lots of $$*
- *Can hire the best people*
- *Can acquire the rest*
- *Plenty of compute resources*

Start-up / Small Government

- *Not a lot of $$*
- *Often can't hire the best people*
- *Left to the mercy of cloud providers*

AN IDEA

Let's make computers super fast, and easy to program



WHERE TO START

Brook for GPUs: Stream Computing on Graphics Hardware

Ian Buck Tim Foley Daniel Horn Jeremy Sugerman Kayvon Fatahalian Mike Houston Pat Hanrahan

Stanford University

WASHINGTON UNIVERSITY

SCHOOL OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

X-SIM AND X-EVAL:

TOOLS FOR SIMULATION AND ANALYSIS OF HETEROGENEOUS PIPELINED ARCHITECTURES

by

Saurabh Gayen

Prepared under the direction of Professor Mark A. Franklin

SPADE: The System S Declarative Stream Processing Engine

Bugra Gedik, Henrique Andrade, Kun-Lung Wu — IBM Thomas J. Watson Research Center, Hawthorne, NY, 10532, USA — bgedik@us.ibm.com, hcma@us.ibm.com, klwu@us.ibm.com

Philip S. Yu — Department of Computer Science, University of Illinois, Chicago, IL 60607, USA — psyu@cs.uic.edu

MyungCheol Doo — College of Computing, Georgia Institute of Technology, GA, 30332, USA — mcdoo@cc.gatech.edu

ABSTRACT

In this paper, we present SPADE − the System S declarative stream processing engine. System S is a large-scale, distributed data stream processing middleware under development at IBM T. J. Watson Research Center. As a front-end for rapid application development for System S, SPADE provides (1) an intermediate language for flexible composition of parallel and distributed data-flow graphs, (2) a toolkit of type-generic, built-in stream processing operators, that support scalar as well as vectorized processing and can seamlessly inter-operate with user-defined operators, and (3) a rich set of stream adapters to ingest/publish data from/to outside sources. More importantly, SPADE automatically brings performance optimization and scalability to System S applications. To that end, SPADE employs a code generation framework to create highly-optimized applications that run natively on the Stream Processing Core (SPC), the execution and communication substrate of System S, and take full advantage of other System S services. SPADE allows developers to construct their applications with fine granular stream operators without worrying about the performance implications that might exist, even in a distributed system. SPADE's optimizing compiler automatically maps applications into appropriately sized execution units in order to minimize communication overhead, while at the same time exploiting available parallelism. By virtue of the scalability of the System S runtime and SPADE's effective code generation and optimization, we can scale applications to a large number of nodes. Currently, we can run SPADE jobs on ≈ 500 processors within more than 100 physical nodes in a tightly connected cluster environment. SPADE has been in use at IBM Research to create real-world streaming applications, ranging from monitoring financial market feeds to radio telescopes to semiconductor fabrication lines.

Categories and Subject Descriptors

H.2.4 [Database Management]: Systems—*Distributed databases*; H.2.3 [Database Management]: Languages—*Data manipulation languages*

General Terms

Design

Keywords

Distributed Data Stream Processing

1. INTRODUCTION

On-line information sources are increasingly taking the form of *data streams*, that is time ordered series of events or readings. Example data streams include live stock and option trading feeds in financial services, physical link statistics in networking and telecommunications, sensor readings in environmental monitoring and emergency response, and satellite and live experimental data in scientific computing. The proliferation of these sources has created a paradigm shift in how we process data, moving away from the traditional "store and then process" model of database management systems toward the "on-the-fly processing" model of emerging data stream processing systems (DSPSs). This paradigm shift has recently created a strong interest in DSPS-related research, in academia [1, 4, 6, 7] and industry [8, 17, 21, 25] alike.

In this paper we describe the design of SPADE, which is the declarative stream processing engine of the massively scalable and distributed System S − a large-scale stream processing middleware under development at IBM Research. SPADE provides a rapid application development front-end for System S. Concretely, SPADE offers:

1. *An intermediate language for flexible composition of parallel and distributed data-flow graphs.* This language sits in-between higher level programming tools and languages such as the System S IDE or stream

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.
Copyright 2008 ACM 978-1-60558-102-6/08/06 ...$5.00.

1123

WHERE TO START

~~*P ft Languag*~~

- ~~Ob t ented~~
- ~~Mult inheritance~~
- ~~Si ple t s~~

~~…. lots mo~~

Library

Language
Java
C
$\boxed{\text{C++}}$
C#
Python

RAFTLIB

C++ Streaming Template Library

Simplifies parallelization of code

Abstracts the details

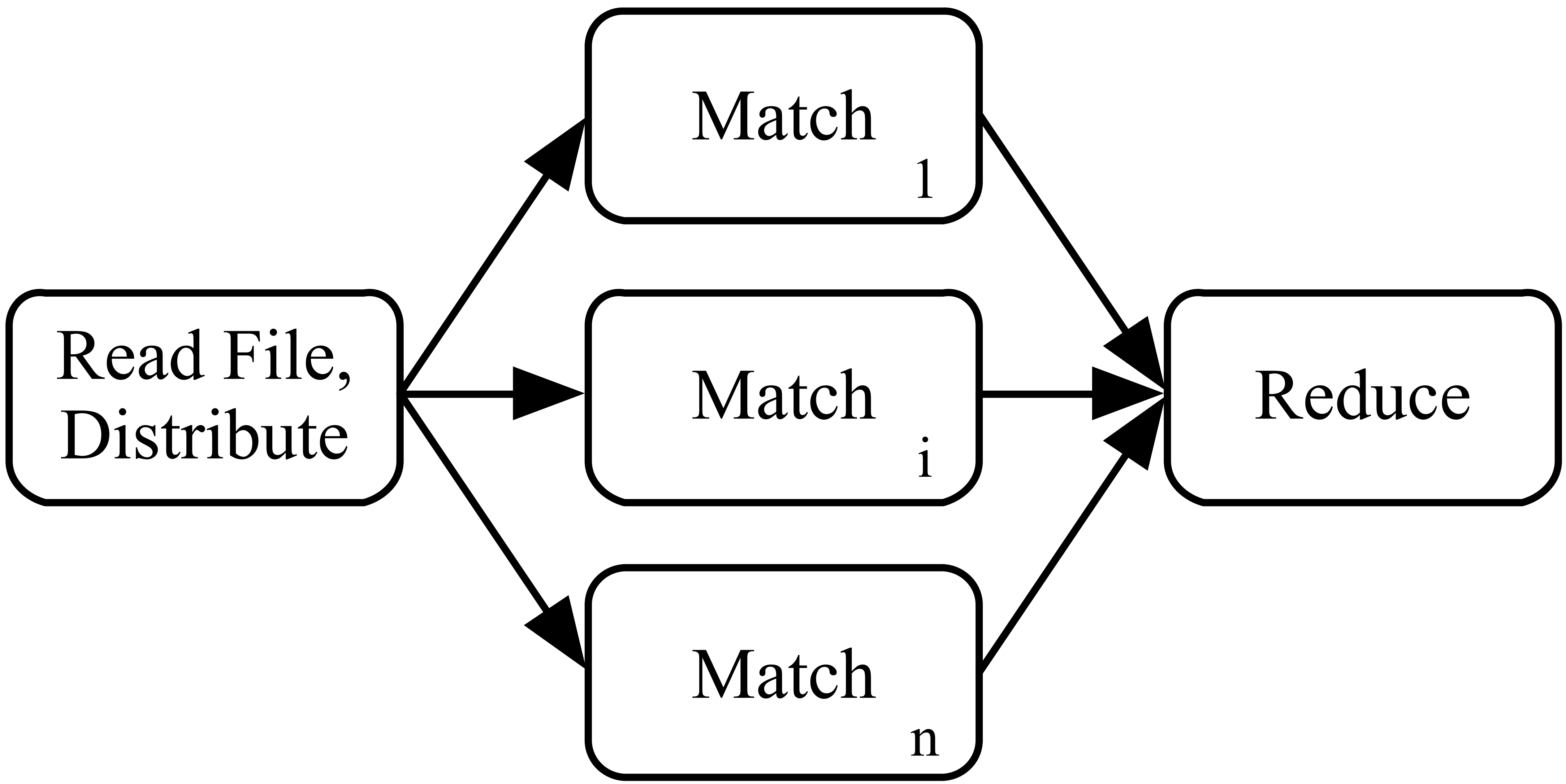
Auto-manages allocation types, data movement.

software download: http://raftlib.io

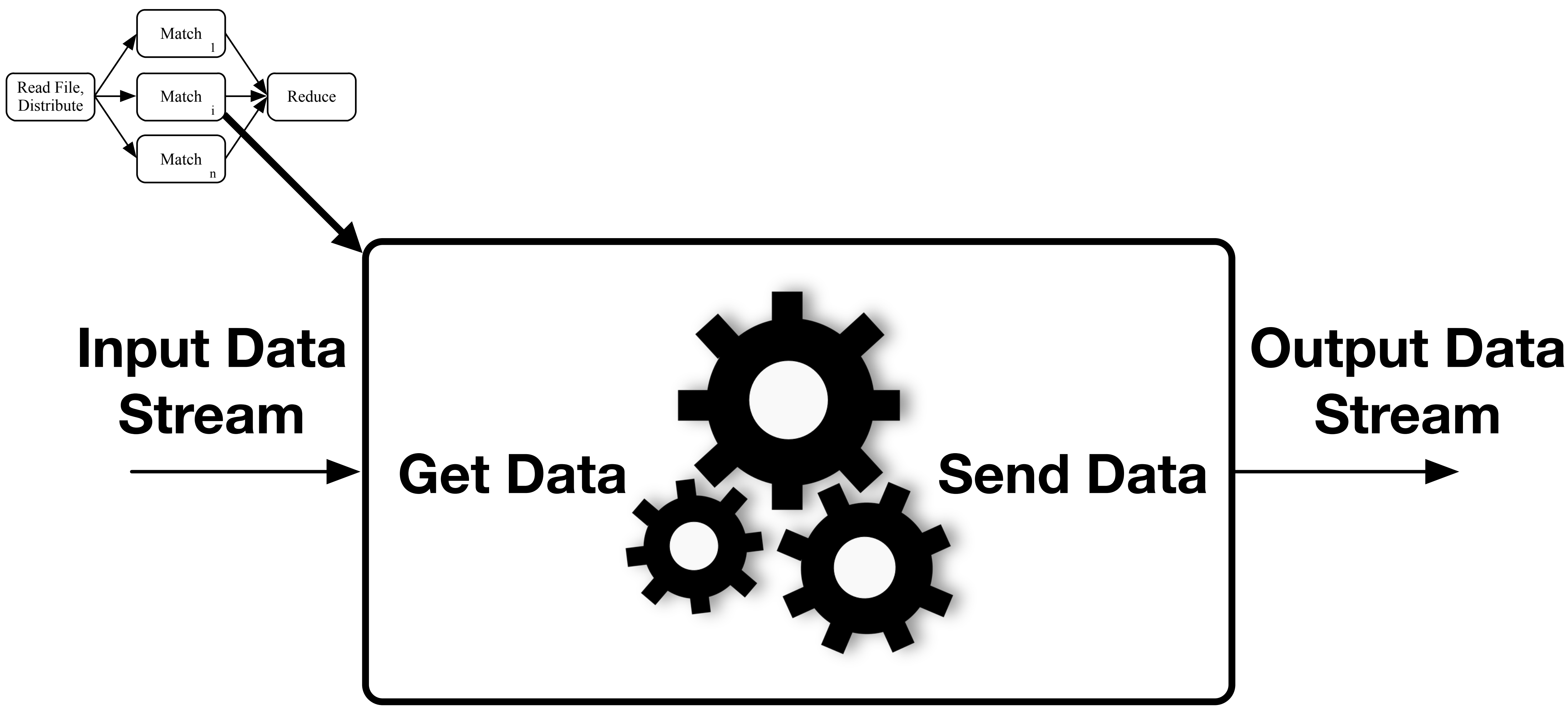
ISSUES THAT NEED SOLUTIONS

- Usable Generic API
- Where to run the code
- Where to allocate memory
- How big of allocations to start off with
- How many user vs. kernel threads
- …Gilligan had a better chance of having a 3 hour tour
- …than we have of fitting all the issues on a single slide

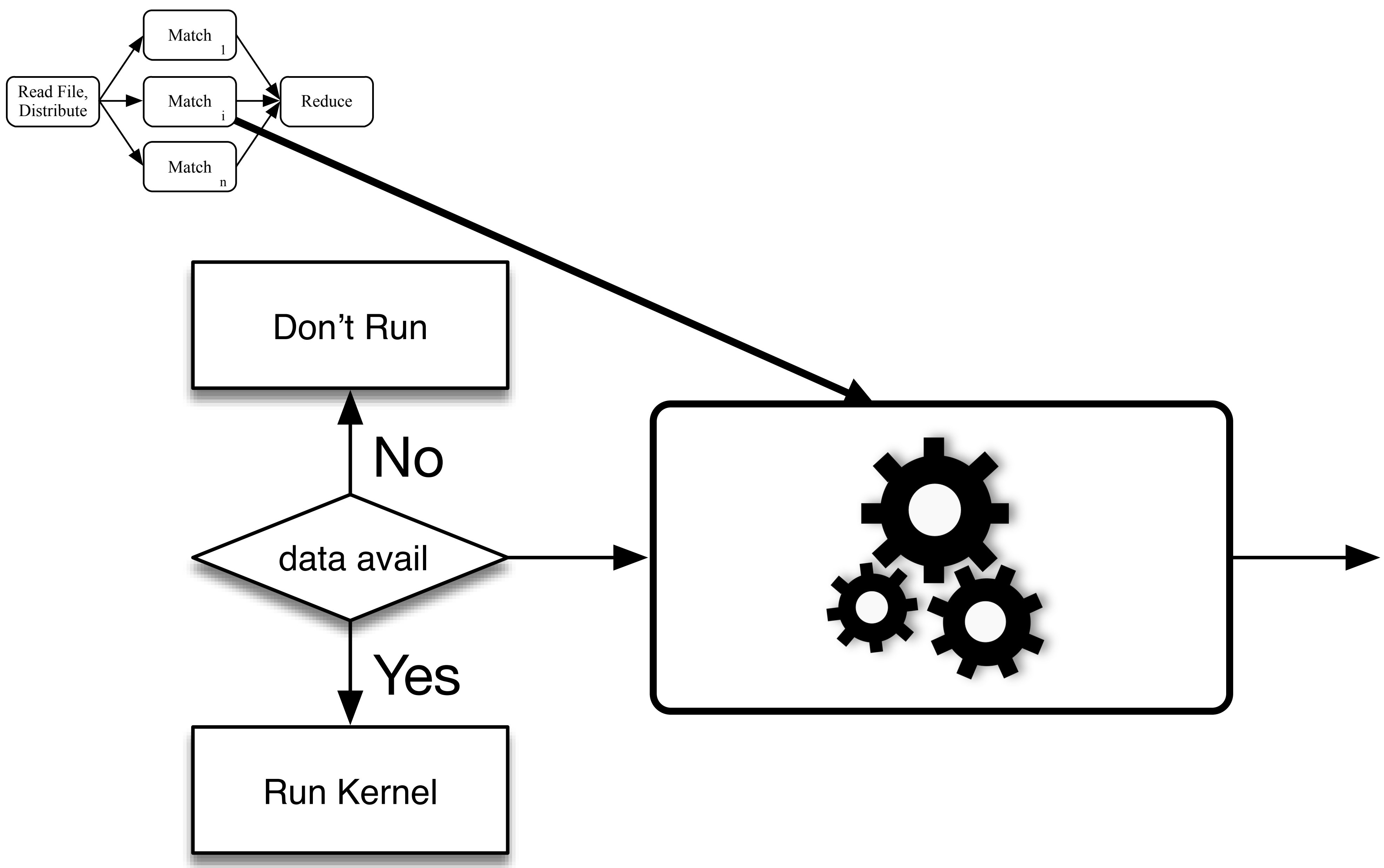
STREAM PROCESSING (STRING SEARCH EXAMPLE)



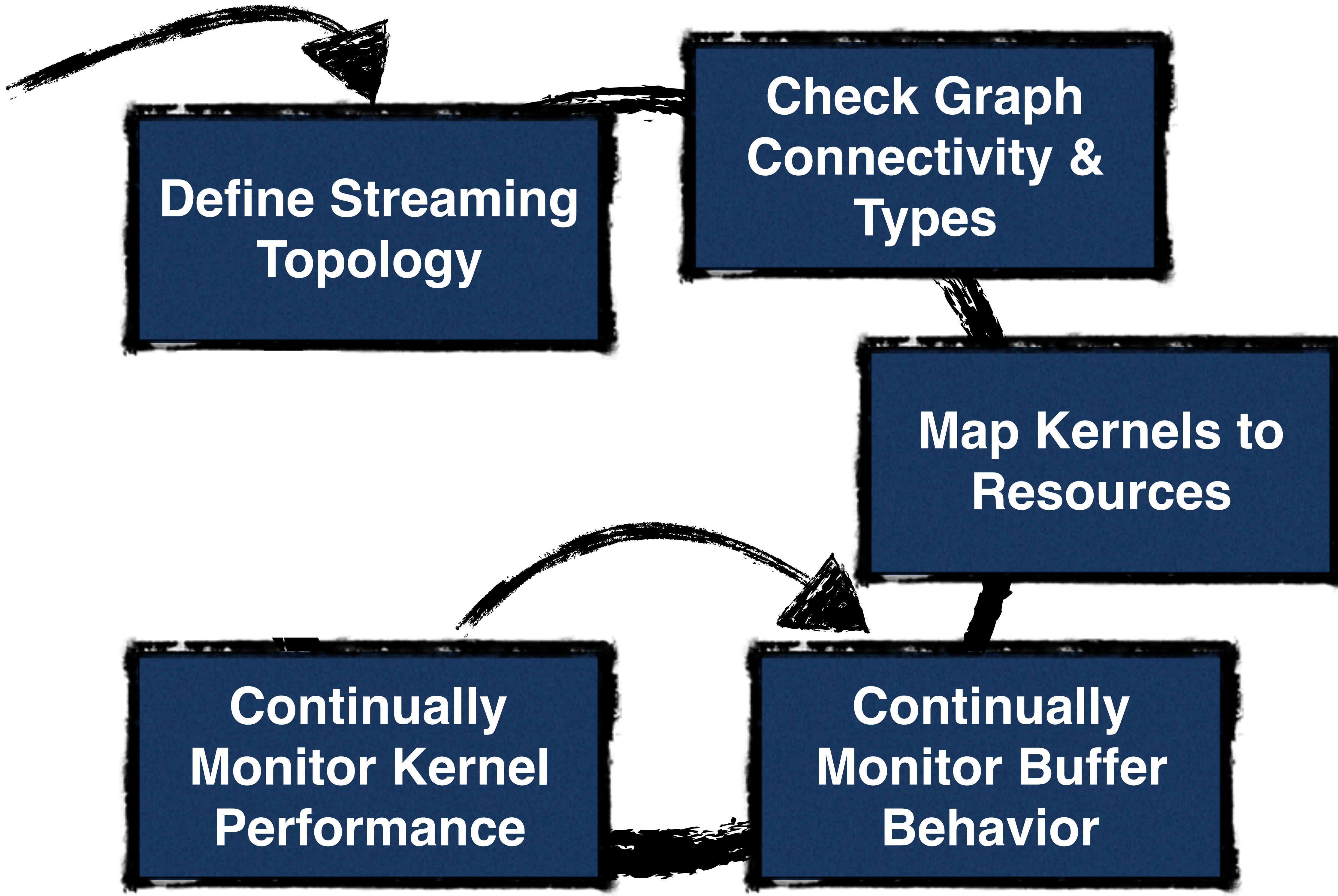
STREAM PROCESSING (STRING SEARCH EXAMPLE)



STREAM PROCESSING (STRING SEARCH EXAMPLE)



HOW IT WORKS (HIGH LEVEL)



Stream Processing (Search Ex)

```
template < class T > class search : public raft::kernel
{
public:
   search( const std::string && term ) : raft::kernel(),
                                         term_length( term.length() ),
                                         term( term )
   {
      input.addPort<  T >( "0" );
      output.addPort< std::size_t >( "0" );
   }

   search( const std::string &term ) : raft::kernel(),
                                       term_length( term.length() ),
                                       term( term )
   {
      input.addPort<  T >( "0" );
      output.addPort< std::size_t >( "0" );
   }

   virtual ~search() = default;

   virtual raft::kstatus run()
   {
      auto &chunk( input[ "0" ].template peek< T >() );
      auto it( chunk.begin() );
      do
      {
         it = std::search( it, chunk.end(),
                           term.begin(), term.end() );
         if( it != chunk.end() )
         {
            output[ "0" ].push( it.location() );
            it += 1;
         }
         else
         {
            break;
         }
      }
      while( true );
      input[ "0" ].unpeek();
      input[ "0" ].recycle( );
      return( raft::proceed );
   }
private:
   const std::size_t term_length;
   const std::string term;
};
```

Stream Processing (Search Ex)

```
template < class T > class search : public raft::kernel
{
public:
   search( const std::string && term ) : raft::kernel(),
                                         term_length( term.length() ),
                                         term( term )
   {
      input.addPort<  T >( "0" );
      output.addPort< std::size_t >( "0" );
   }

   search( const std::string &term ) : raft::kernel(),
                                       term_length( term.length() ),
                                       term( term )
   {
      input.addPort<  T >( "0" );
      output.addPort< std::size_t >( "0" );
   }

   virtual ~search() = default;

   virtual raft::kstatus run()
   {
      auto &chunk( input[ "0" ].template peek< T >() );
      auto it( chunk.begin() );
      do
      {
         it = std::search( it, chunk.end(),
                           term.begin(), term.end() );
         if( it != chunk.end() )
         {
            output[ "0" ].push( it.location() );
            it += 1;
         }
         else
         {
            break;
         }
      }
      while( true );
      input[ "0" ].unpeek();
      input[ "0" ].recycle( );
      return( raft::proceed );
   }
private:
   const std::size_t term_length;
   const std::string term;
};
```

Stream Processing (Search Ex)

```
template < class T > class search : public raft::kernel
{
public:
   search( const std::string && term ) : raft::kernel(),
                                         term_length( term.length() ),
                                         term( term )
   {
      input.addPort<  T >( "0" );
      out
   }

   search

   {
      inp
      out
   }

   virtua

   virtua
   {
      aut
      aut
      do
      {

         {
            break;
         }
      }
      while( true );
      input[ "0" ].unpeek();
      input[ "0" ].recycle( );
      return( raft::proceed );
   }
private:
   const std::size_t term_length;
   const std::string term;
};
```

```
public:
   search( const std::string && term ) : raft::kernel(),
                                         term_length( term.length() ),
                                         term( term )
   {
      input.addPort<  T >( "0" );
      output.addPort< std::size_t >( "0" );
   }

   search( const std::string &term ) : raft::kernel(),
                                       term_length( term.length() ),
                                       term( term )
   {
      input.addPort<  T >( "0" );
      output.addPort< std::size_t >( "0" );
   }
```

Stream Processing (Search Ex)

```
template < class T > class search : public raft::kernel
{
public:
   search( const std::string && term ) : raft::kernel(),
                                         term_length( term.length() ),
                                         term( term )
   {
      input.addPort<  T >( "0" );
      output.addPort< std::size_t >( "0" );
   }

   search( const std::string &term ) : raft::kernel(),
                                       term_length( term.length() ),
                                       term( term )
   {
      input.addPort<  T >( "0" );
      output.addPort< std::size_t >( "0" );
   }

   virtual ~search() = default;

   virtual raft::kstatus run()
   {
      auto &chunk( input[ "0" ].template peek< T >() );
      auto it( chunk.begin() );
      do
      {
         it = std::search( it, chunk.end(),
                           term.begin(), term.end() );
         if( it != chunk.end() )
         {
            output[ "0" ].push( it.location() );
            it += 1;
         }
         else
         {
            break;
         }
      }
      while( true );
      input[ "0" ].unpeek();
      input[ "0" ].recycle( );
      return( raft::proceed );
   }
private:
   const std::size_t term_length;
   const std::string term;
};
```

Stream Processing (Search Ex)

```
template < class T > class search : public raft::kernel
{
public:
   search( const s

   {
      input.addPor
      output.addPo
   }

   search( const s

   {
      input.addPor
      output.addPo
   }

   virtual ~search

   virtual raft::k
   {
      auto &chunk(
      auto it( chu
      do
      {
         it = std:

         if( it !=
         {
            output
            it +=
         }
         else
         {
            break;
         }
      }
      while( true
      input[ "0" ]
      input[ "0" ]
      return( raft
   }
private:
   const std::size
   const std::string term;
};
```

```
virtual raft::kstatus run()
{
   auto &chunk( input[ "0" ].template peek< T >() );
   auto it( chunk.begin() );
   do
   {
      it = std::search( it, chunk.end(),
                        term.begin(), term.end() );
      if( it != chunk.end() )
      {
         output[ "0" ].push( it.location() );
         it += 1;
      }
      else
      {
         break;
      }
   }
   while( true );
   input[ "0" ].unpeek();
   input[ "0" ].recycle( );
   return( raft::proceed );
}
```

Stream Processing (Search Ex)

```
int
main( int argc, char **argv )
{
    using chunk = raft::filechunk< 256 >;
    using fr    = raft::filereader< chunk, false >;
    using search = search< chunk >;
    using print = raft::print< std::size_t, '\n'>;

    const std::string term( "Alice" );
    raft::map m;

    fr   read( argv[ 1 ], 1, term.length() );
    search find( term );
    print p;

    m += read >> raft::order::out >> find >> raft::order::out >> p;
    m.exe();

    return( EXIT_SUCCESS );
}
```

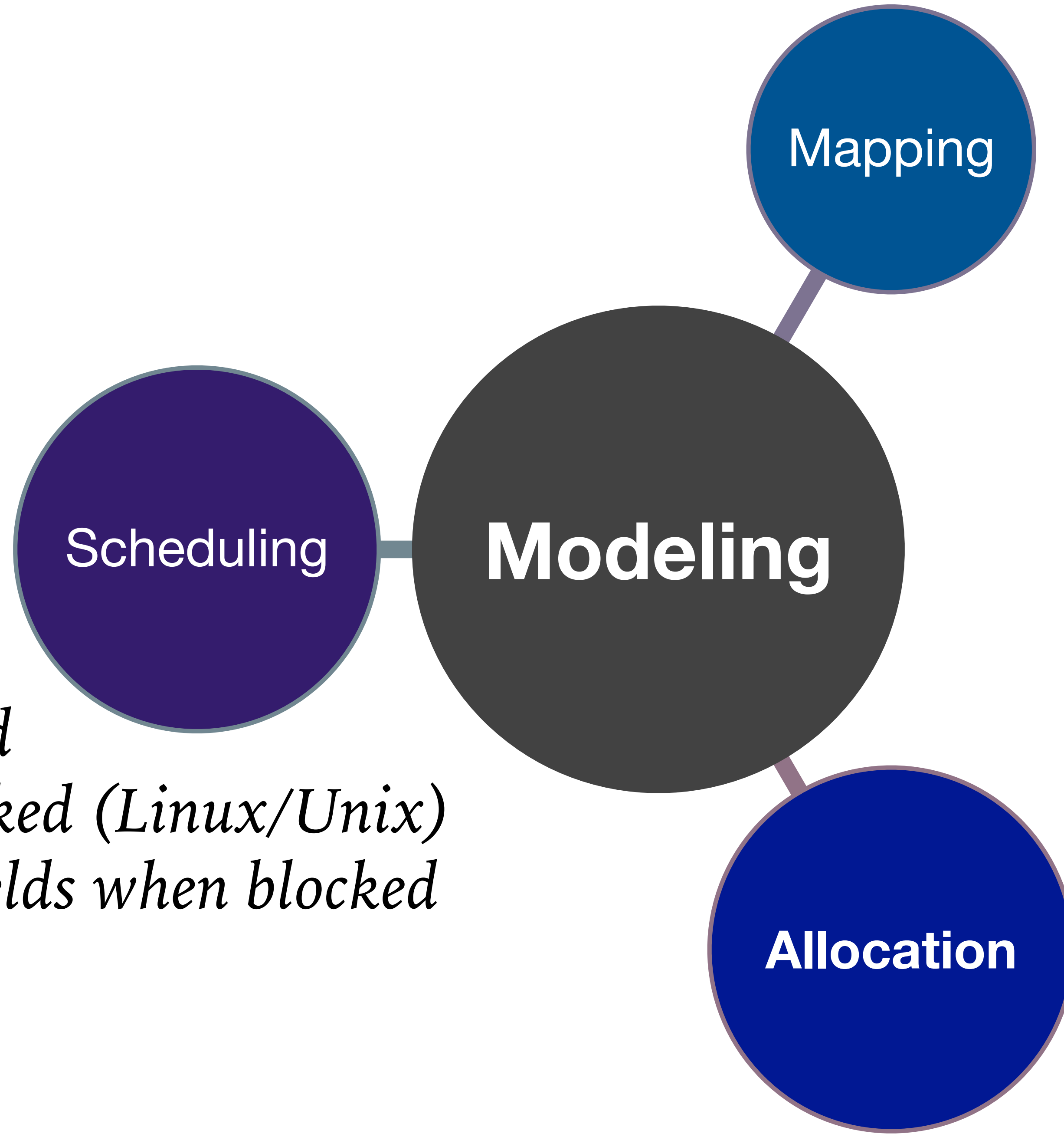
CHOOSE YOUR ADVENTURE

Models

Interface

Implementation

MODELING

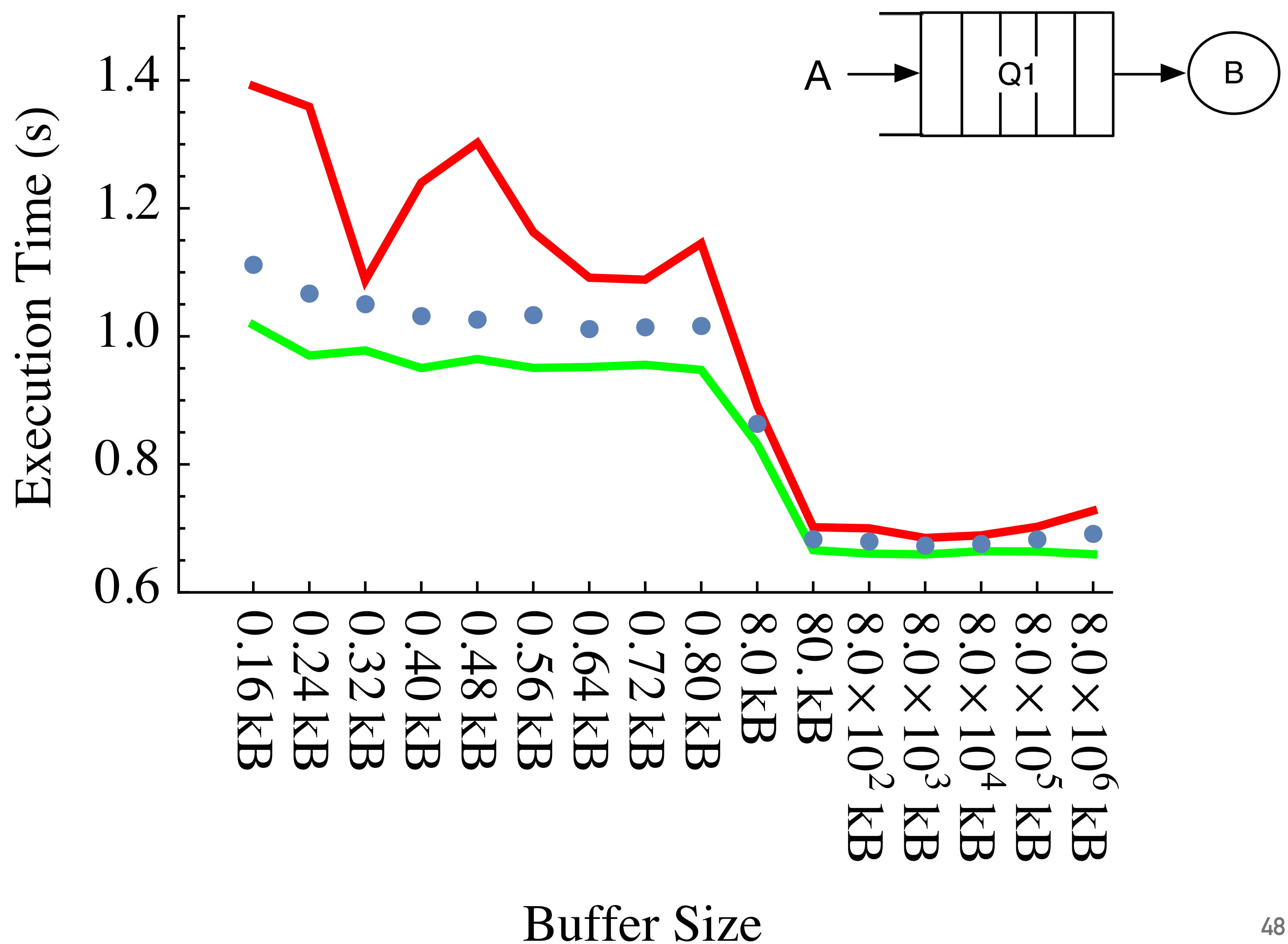


- *RaftLib optionally uses Scotch*
- *Otherwise basic affinity*

- *OS Managed*
- *Affinity Locked (Linux/Unix)*
- *Run-time yields when blocked*

Mostly my work, which we can talk about

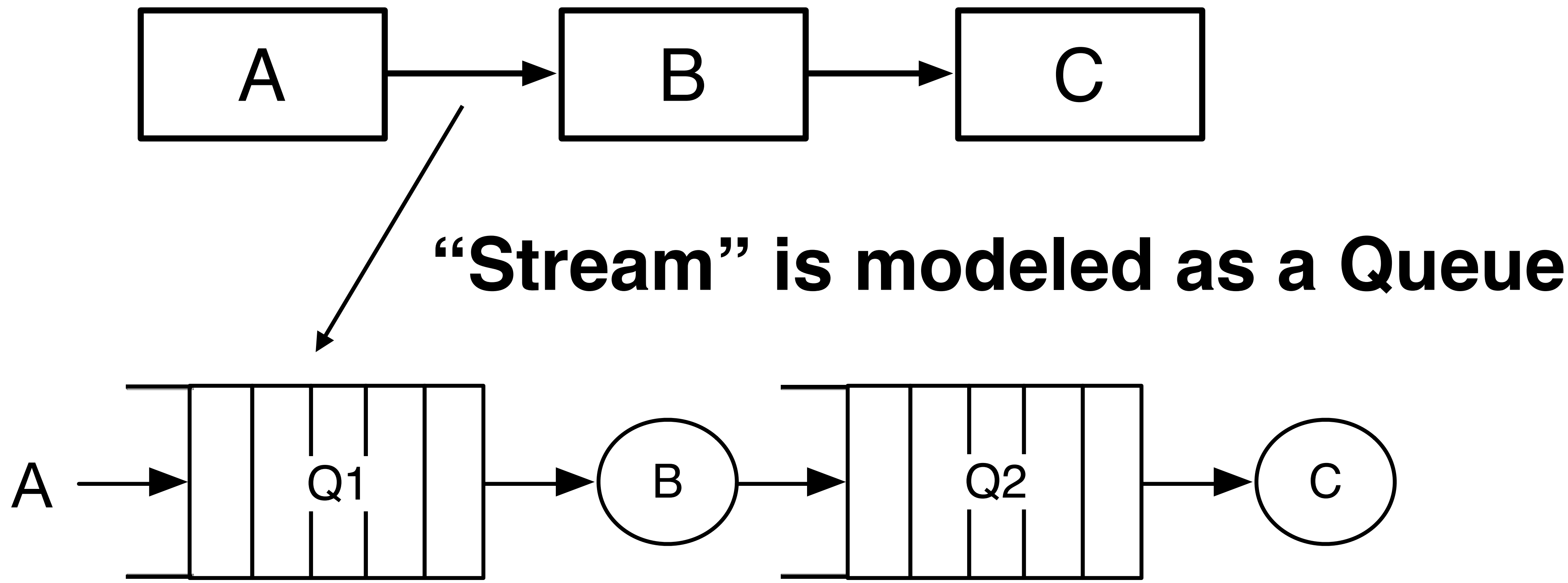
MODELING ISSUES – SIZE OF Q1



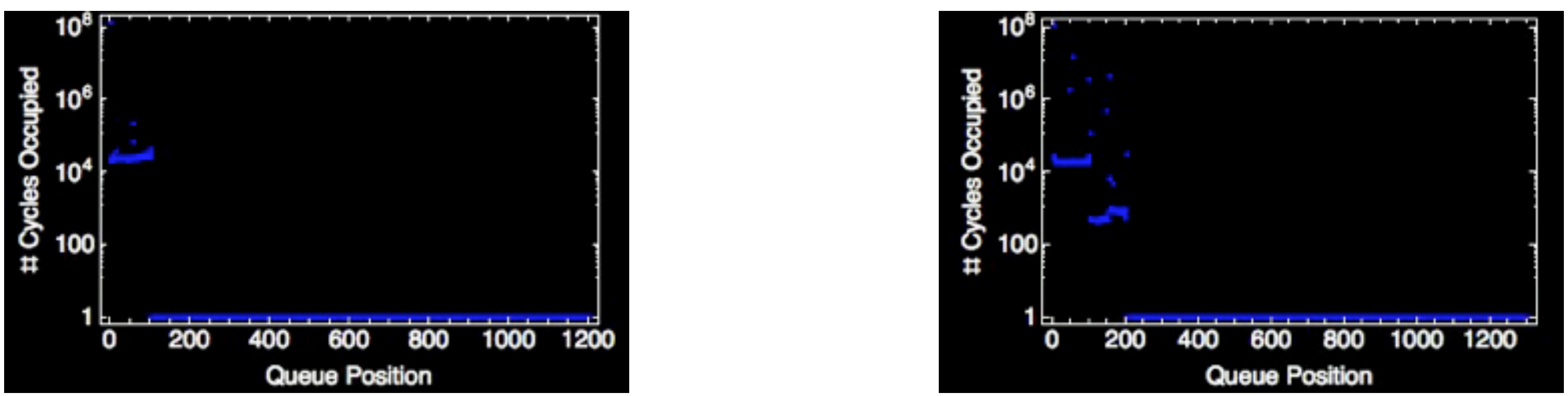
WHY DO WE NEED TO SOLVE

- Buffer allocations take time and energy
- Programmers are horrible at deciding (too many parameters)
- Hardware specific locations matter (NUMA)
- Re-allocating with an accelerator takes even more time (bus latency, hand-shakes, etc.)
- Must be solved in conjunction with partitioning/scheduling/placement

MODELING

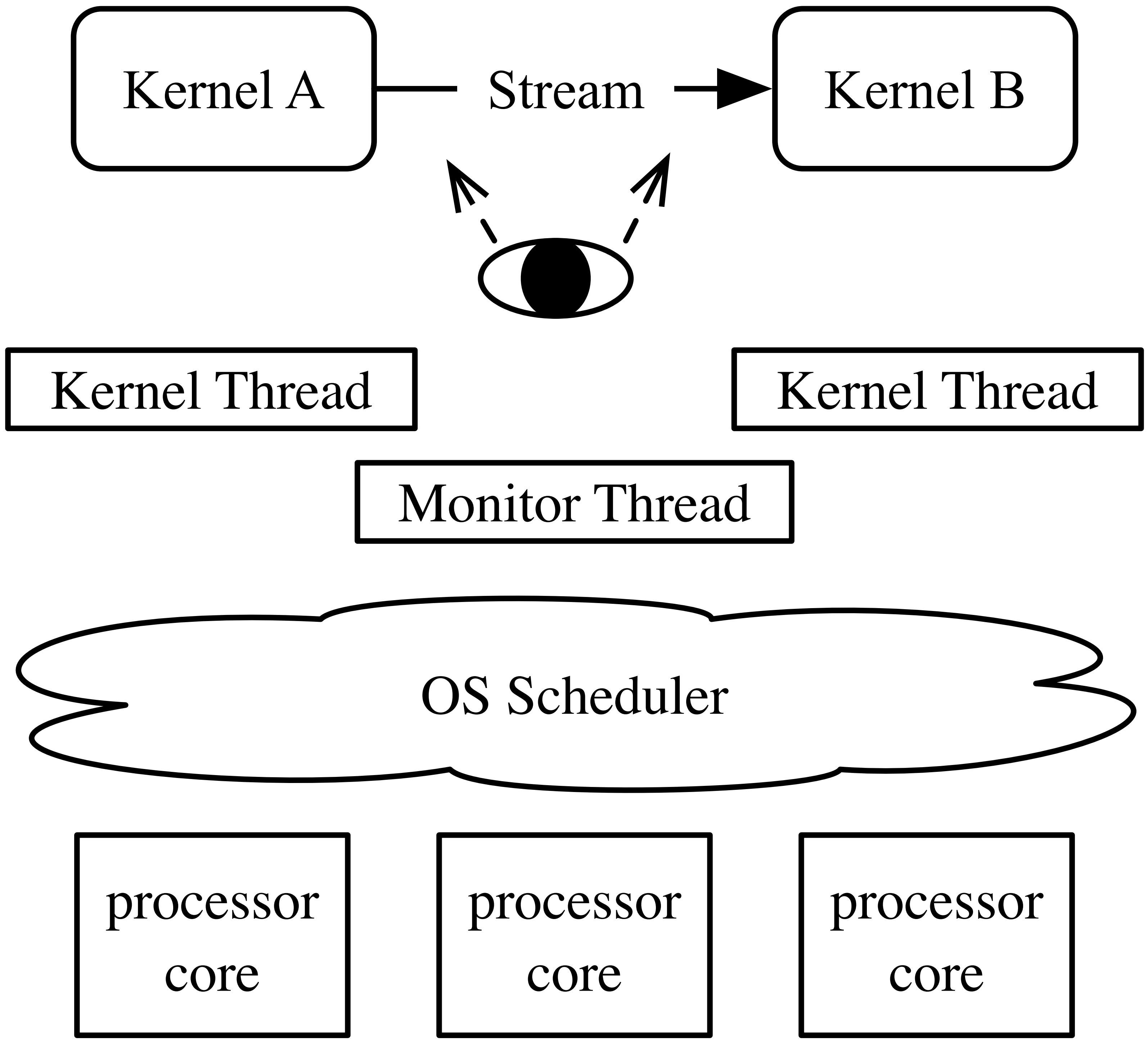


A
B
C
"Stream" is modeled as a Queue
A
Q1
B
Q2
C

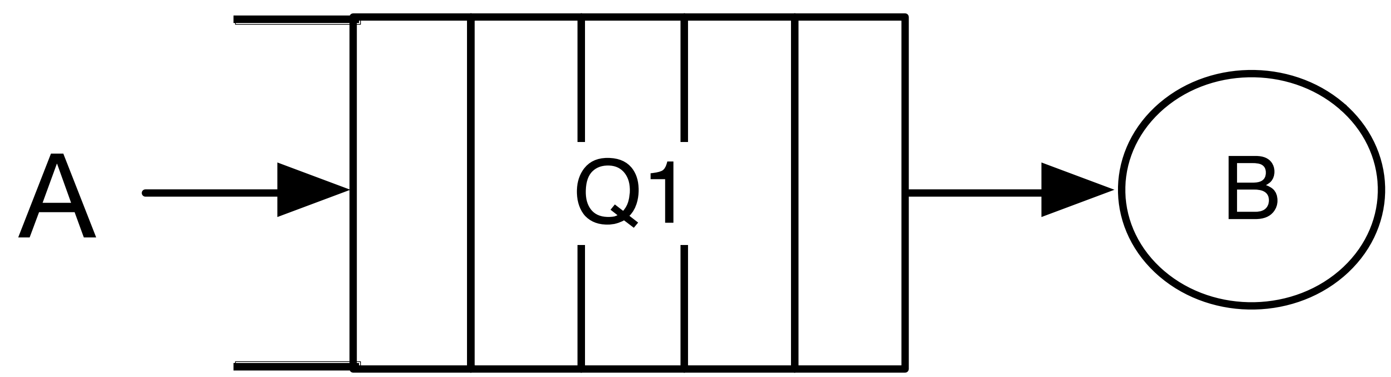


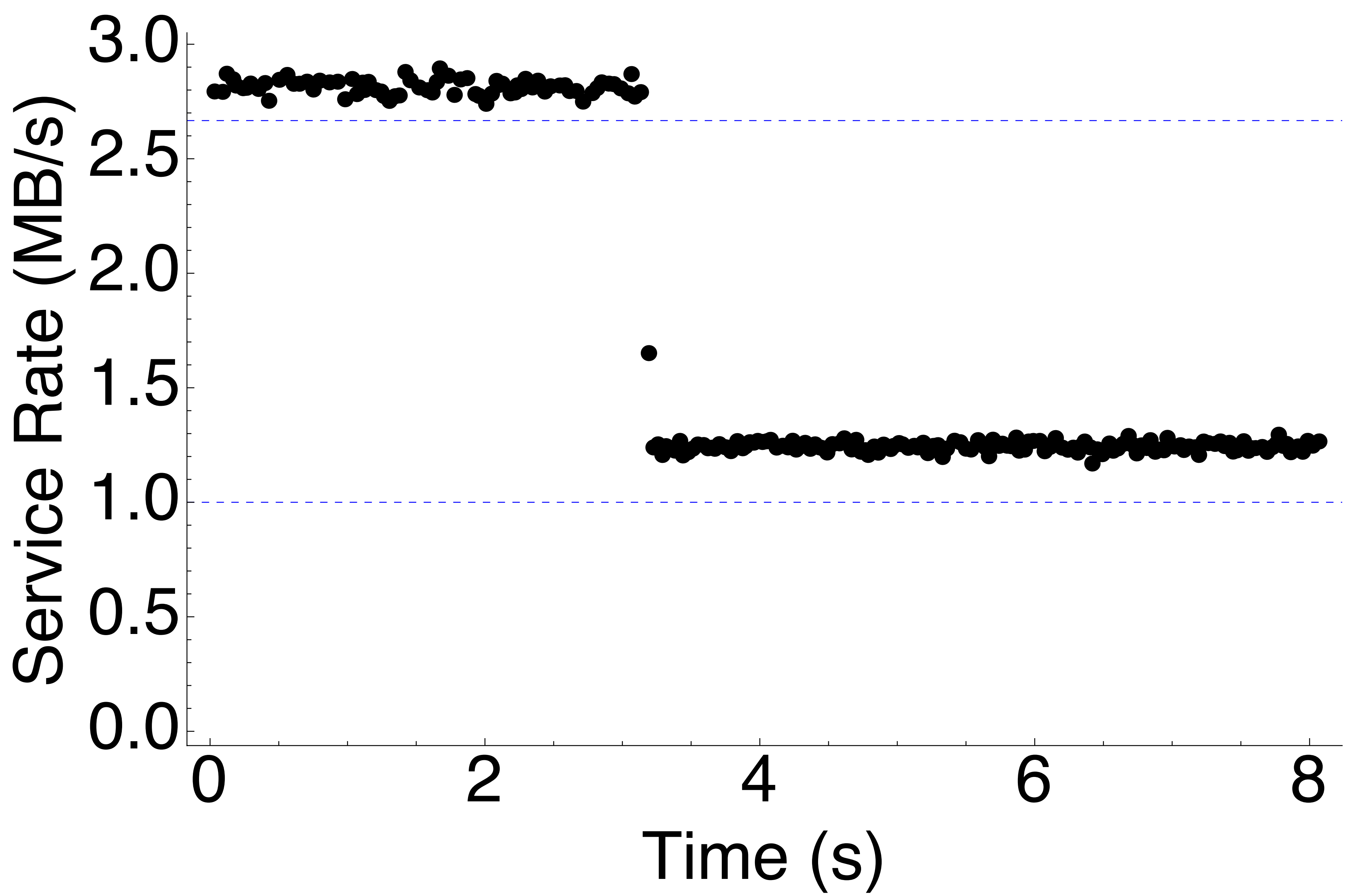
Cycles Occupied
Queue Position
Cycles Occupied
Queue Position

MONITOR

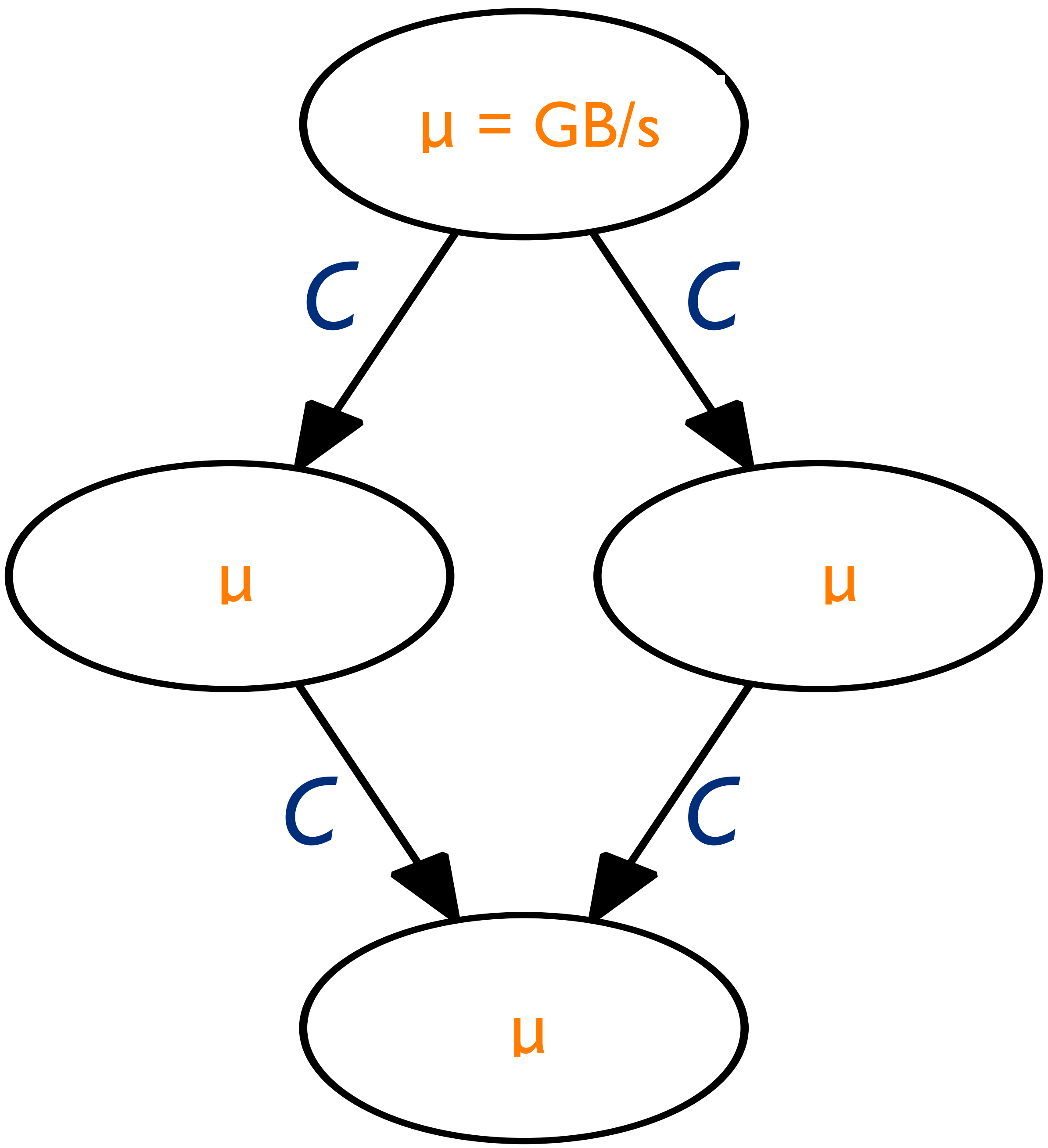

Kernel A
Stream
Kernel B
Kernel Thread
Kernel Thread
Monitor Thread
OS Scheduler
processor core
processor core
processor core

APPROXIMATE INSTRUMENTATION





SOLVE FOR THROUGHPUT QUICKLY



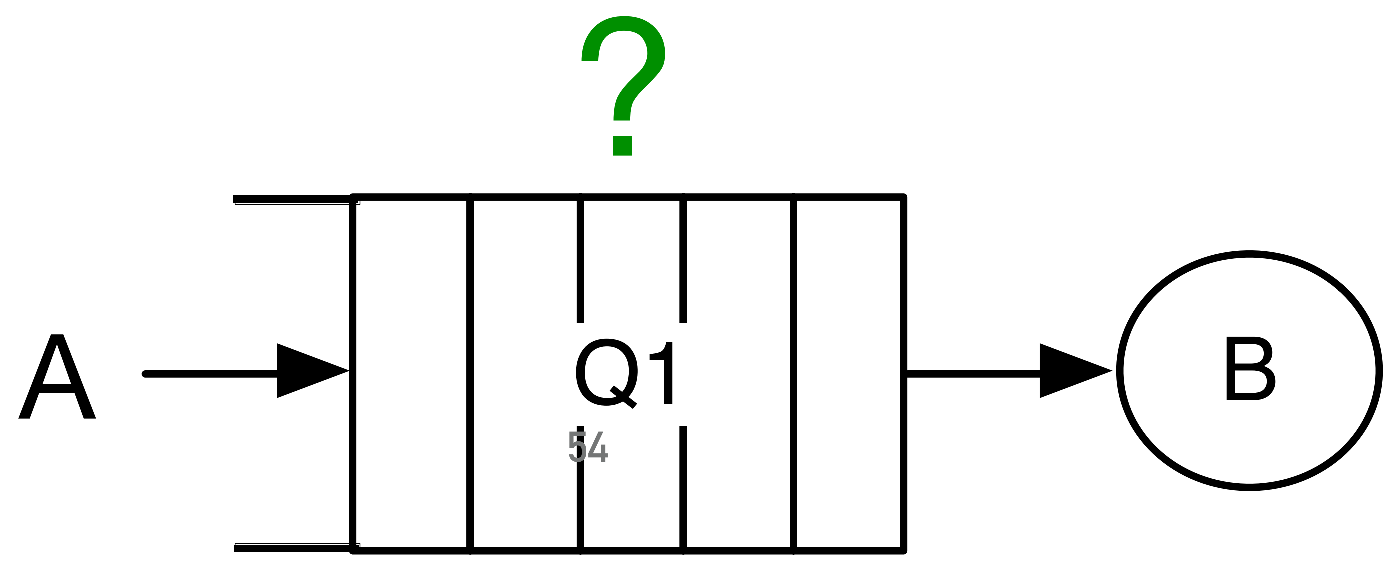
Use network flow model to quickly estimate flow within a streaming graph

Decompose queueing network and solve each queueing station independently

HOW WOULD YOU GET THE RIGHT BUFFER SIZE?

Queueing Model to Solve for Buffer Size

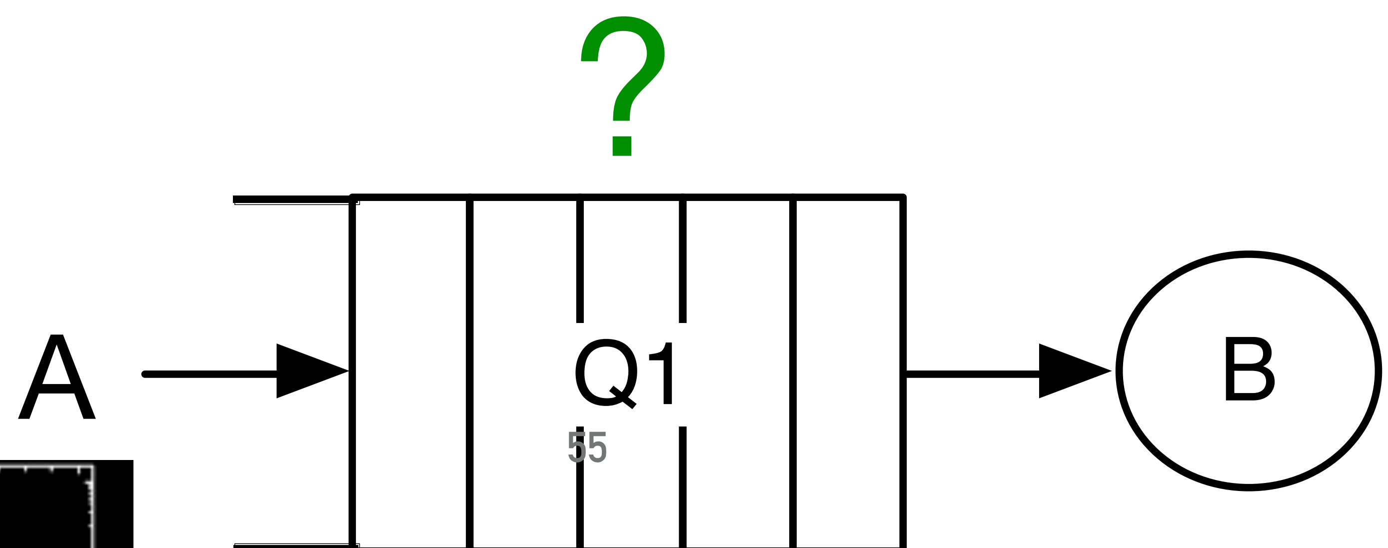
M/D/1 M/M/1 **None**

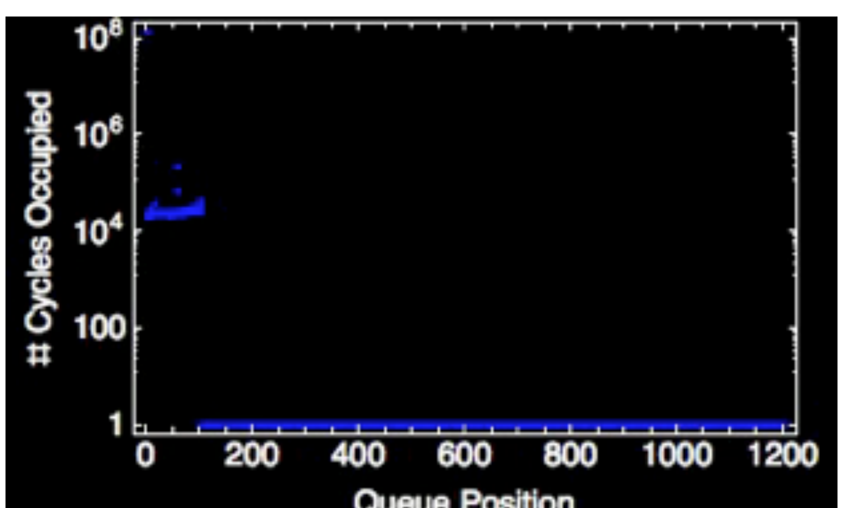


HOW WOULD YOU GET THE RIGHT BUFFER SIZE?

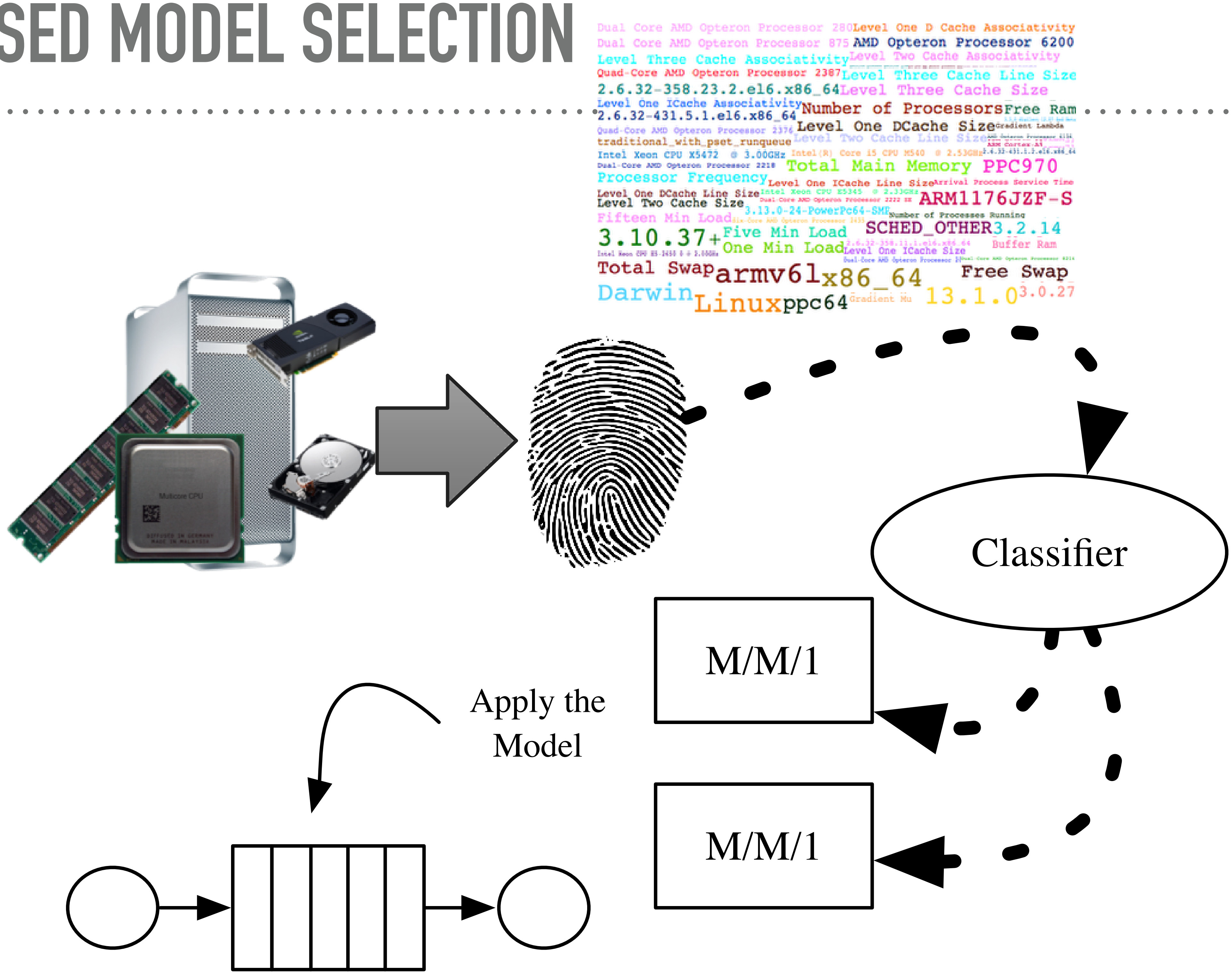
Queueing Model to Solve for Buffer Size

M/D/1 M/M/1 **None**





ML BASED MODEL SELECTION



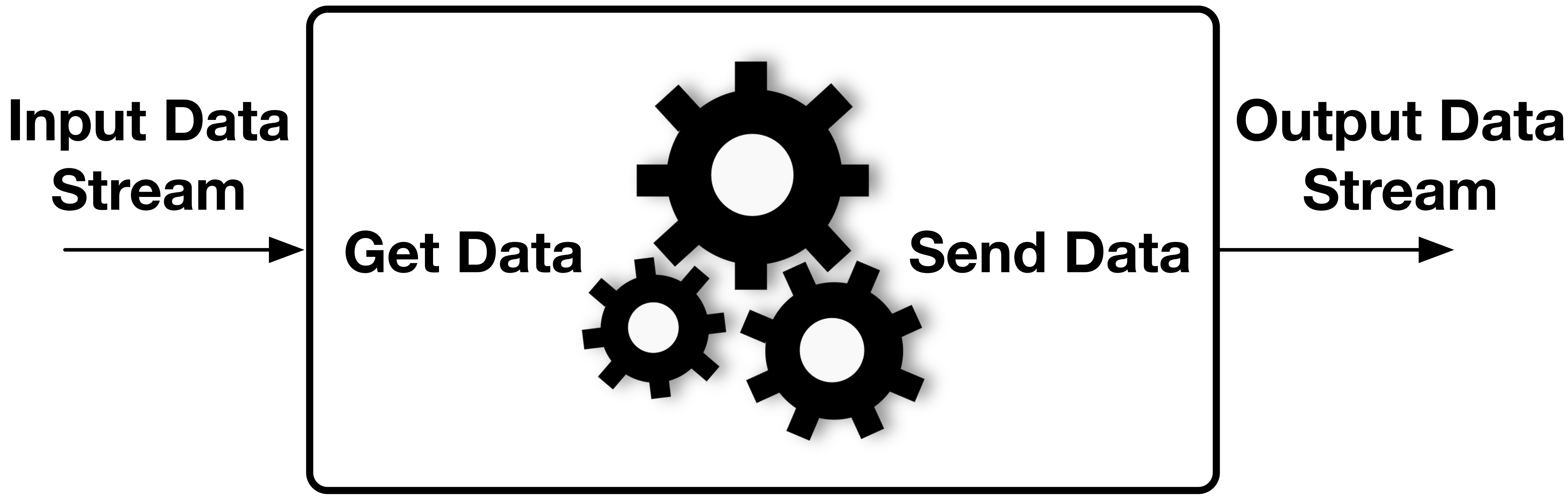
CHOOSE YOUR ADVENTURE

[Models]

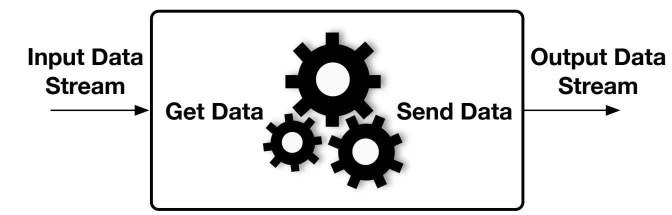
[Interface]

[Implementation]

COMPUTE KERNEL



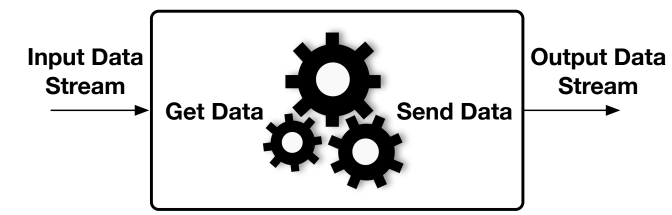
COMPUTE KERNEL

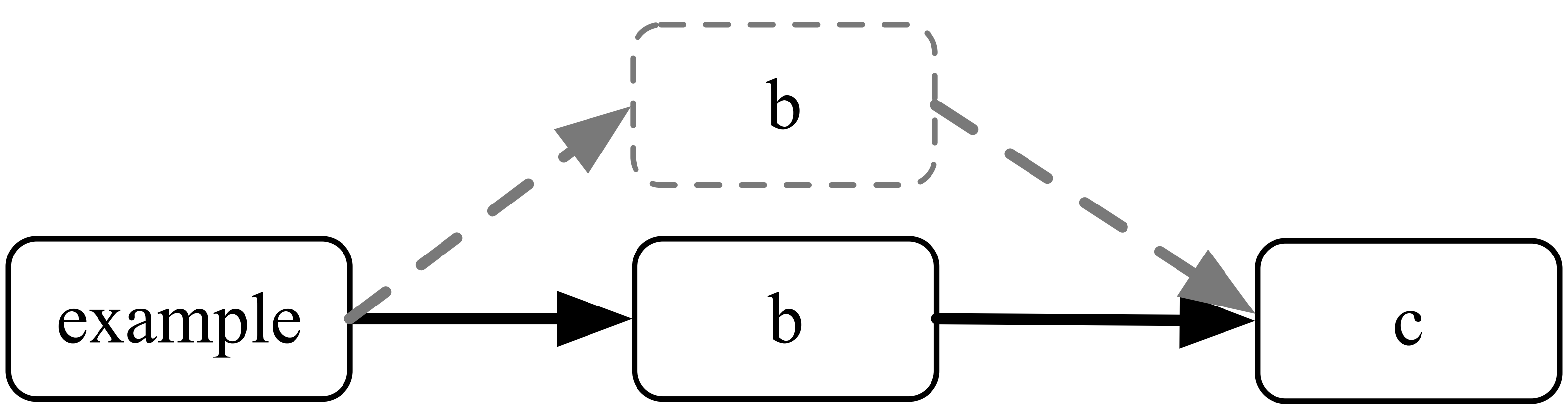



```
class akernel : public raft::kernel
{
public:
    akernel() : raft::kernel()
    {
        //add input ports
        input.addPort< /** type **/ >( "x0","x1", "x..." );
        //add output ports
        output.addPort< /** type **/ >( "y0", "y1", "y..." );
    }

    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 **/ );
    }
};
```

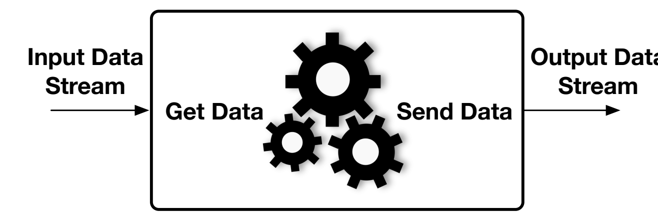
COMPUTE KERNEL

```
class example : public raft::parallel_k
{
public:
   example() : parallel_k()
   {
      input.addPort< /** some type **/ >( "0" );
      /** add a starter output port **/
      addPort();
   }

   /** implement virtual function **/
   virtual std::size_t  addPort()
   {
     return( (this)->addPortTo< /** type **/ >( output /** direction **/ ) );
   }

"..."
```





RECEIVING DATA

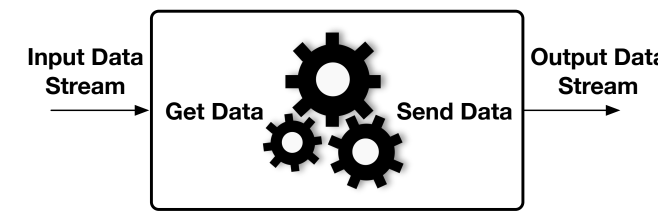



```
/**
 * return reference to memory on
 * in-bound stream
 */
template< class T >
T& peek( raft::signal *signal = nullptr )

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

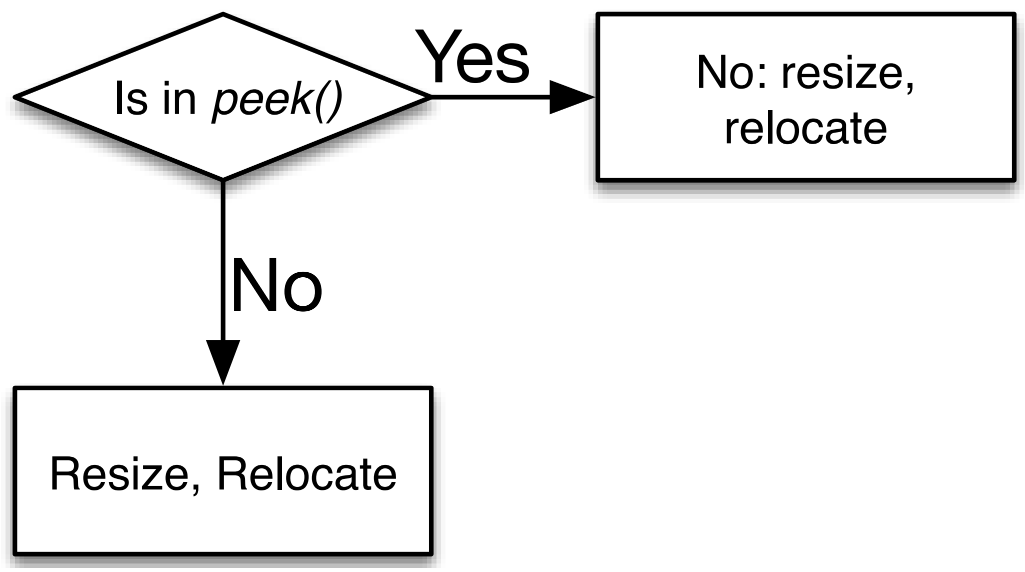
- *Returns object with "special access to stream"*
- *Operator [ ] overload returns auto pair*
- *Direct reference as in peek() for each element*

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

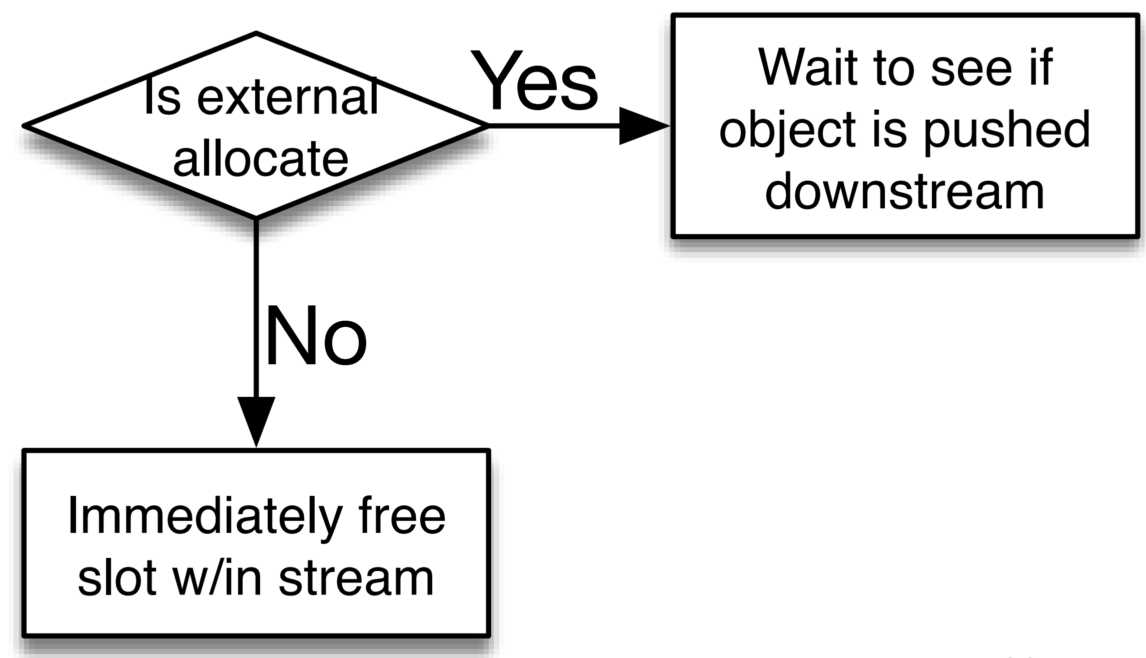
RECEIVING DATA



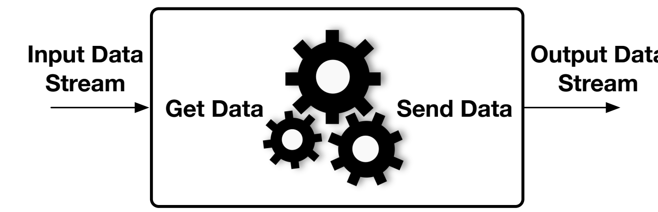

```
/**
 * necessary to tell inbound stream we're done
 * looking at it
 */
virtual
void unpeek()
```




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



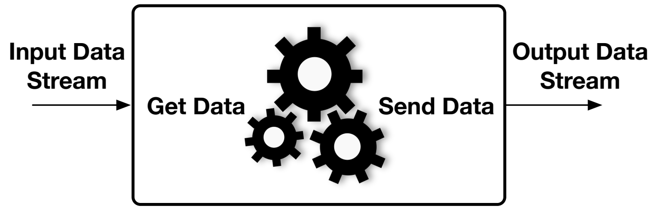
RECEIVING DATA

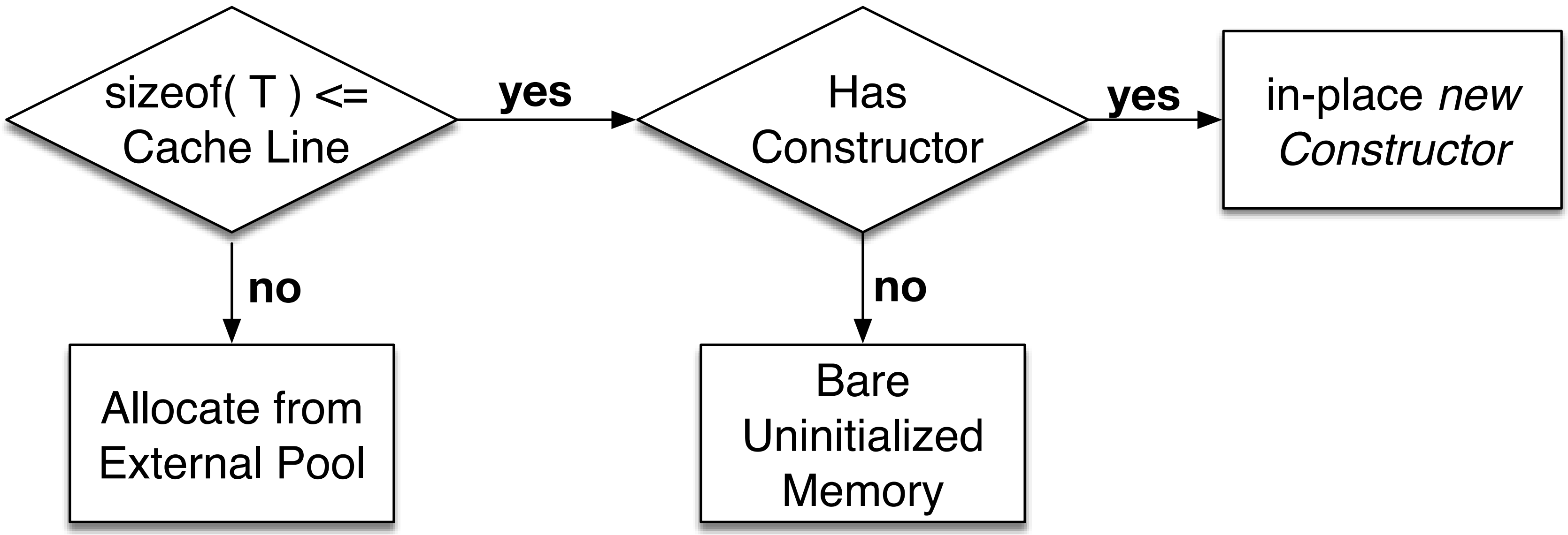



```
/**
 * these pops produce a copy
 */
template< class T >
void pop( T &item, raft::signal *signal = nullptr )

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

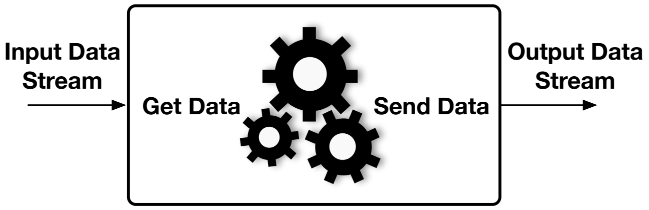
```
/** in-place allocation **/
template < class T,
           class ... Args >
T& allocate( Args&&... params )
```





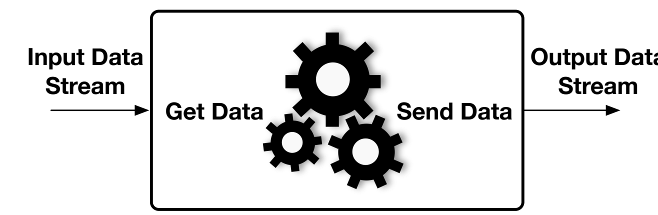

```
/** in-place alloc of range for fundamental types **/
template < class T >
auto allocate_range( const std::size_t n ) ->
       std::vector< std::reference_wrapper< T > >
```

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



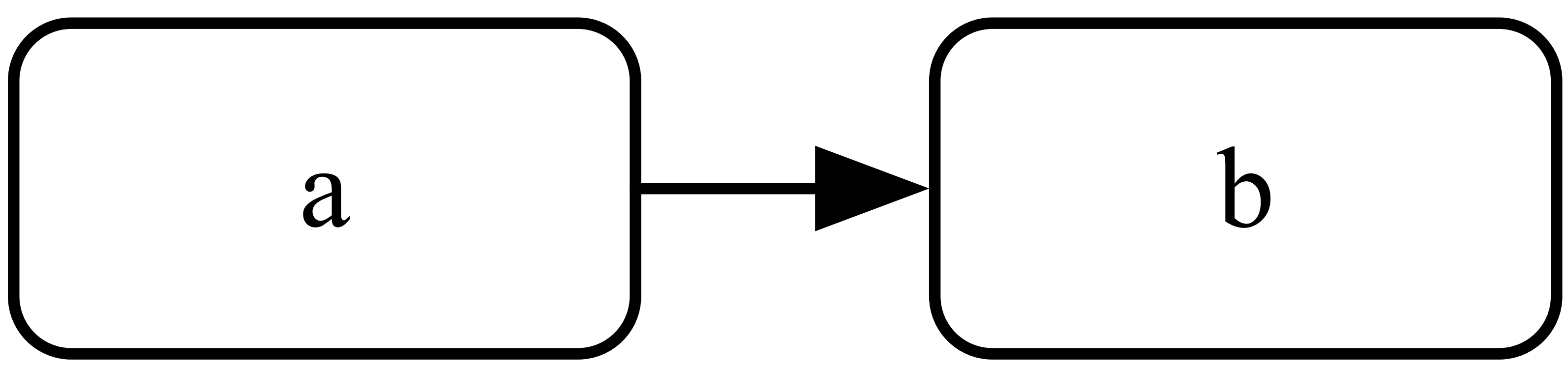

```
/** multiple forms **/
template < class T >
void push( const T &item, const raft::signal signal = raft::none )


/** 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 )
```

INCLUDED KERNELS

```
/**
 * thread safe print, specialization for '\n' vs. '\0'
 */
template< typename T, char delim = '\0' > class print

/** read from iterator to streams **/
static
raft::readeach< T, Iterator >
read_each( Iterator &&begin,
           Iterator &&end )

/** write from iterator to streams **/
template < class T, class BackInsert >
static
writeeach< T, BackInsert >
write_each( BackInsert &&bi )
{
    return( writeeach< T, BackInsert >( bi ) );
}
```

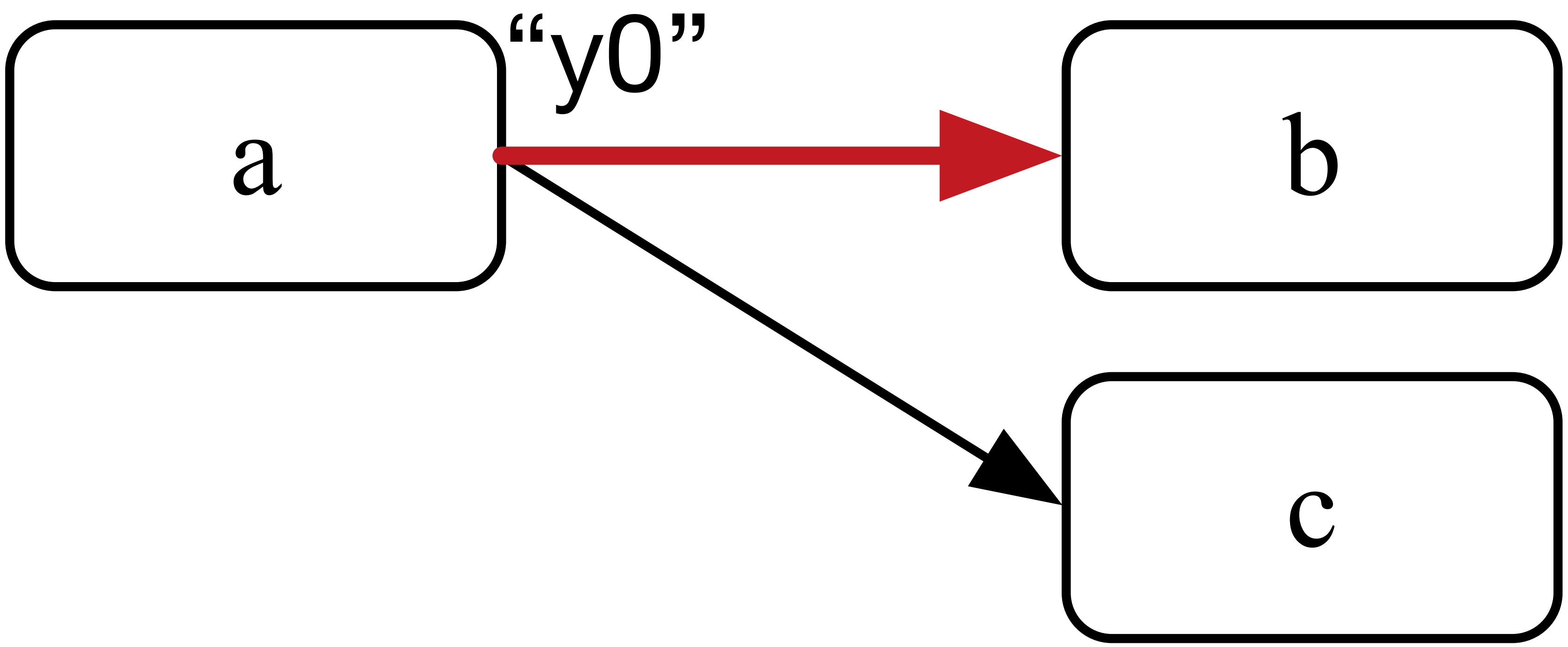
CONNECTING COMPUTE KERNELS

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



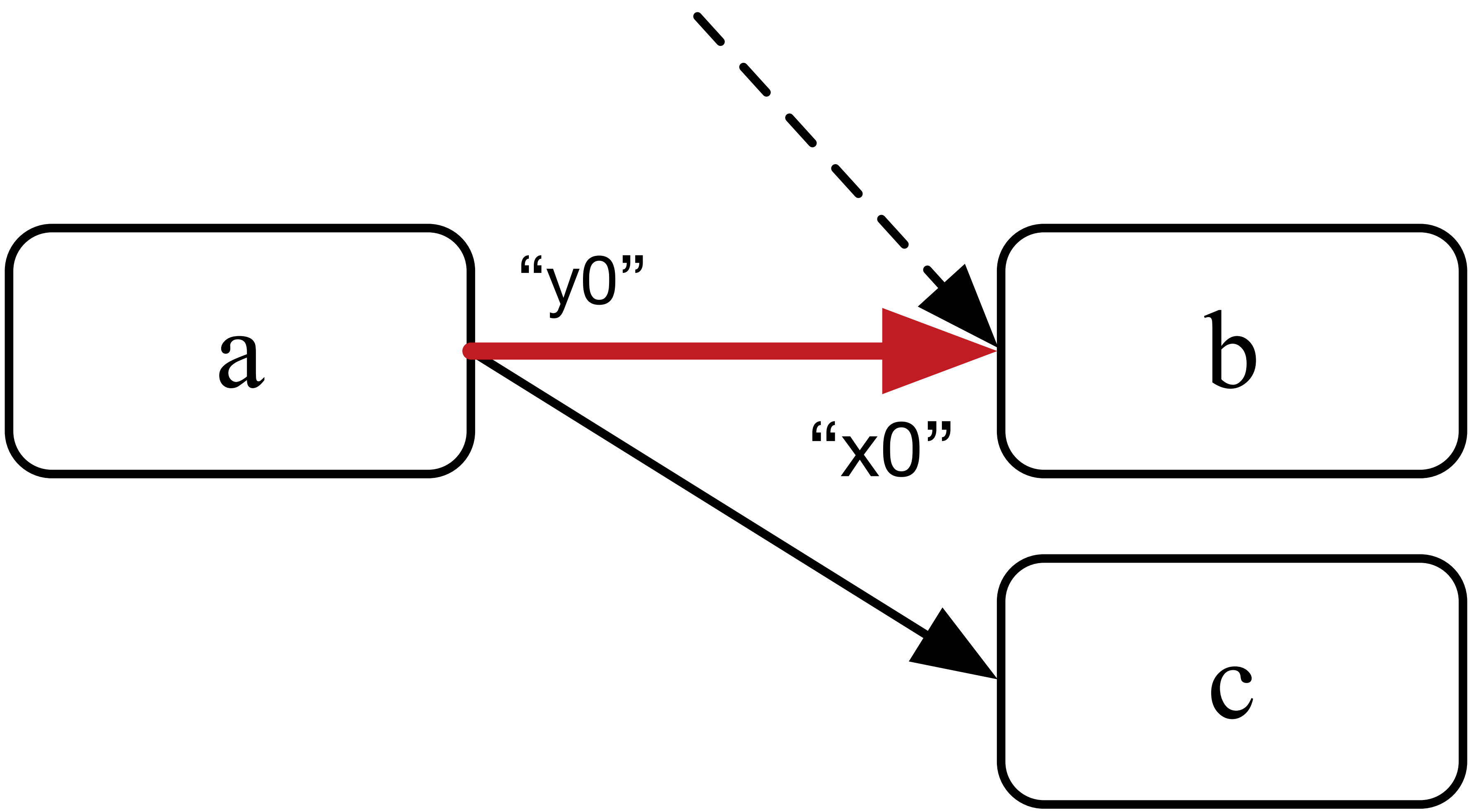
CONNECTING COMPUTE KERNELS

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



CONNECTING COMPUTE KERNELS

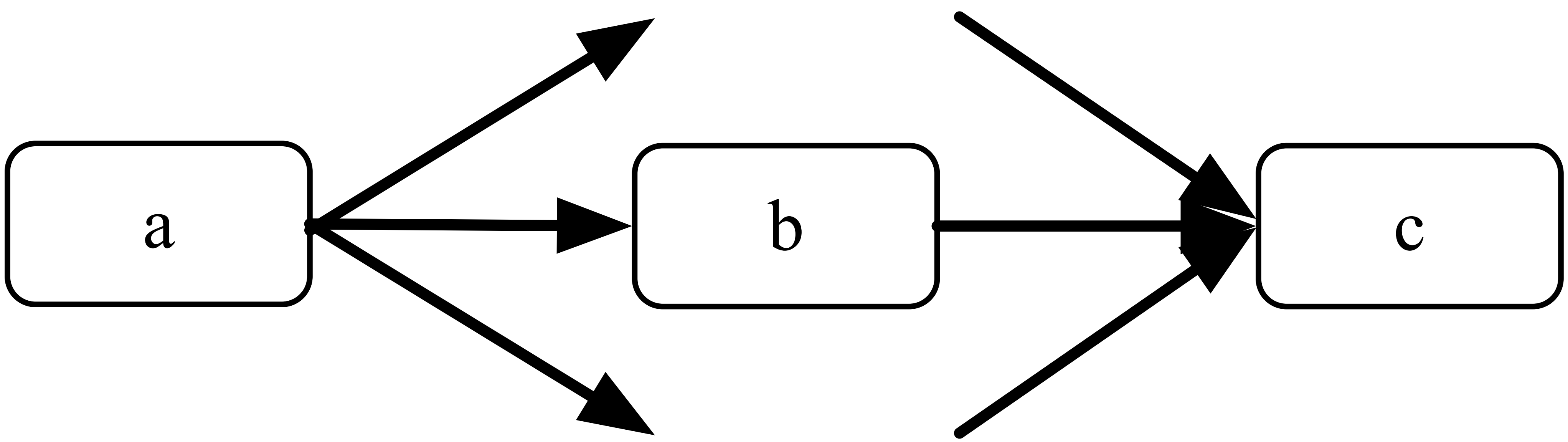
```
raft::map m;
/** example only **/
raft::kernel a, b;
m += a[ "y0" ] >> b[ "x0" ];
```



CONNECTING COMPUTE KERNELS

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

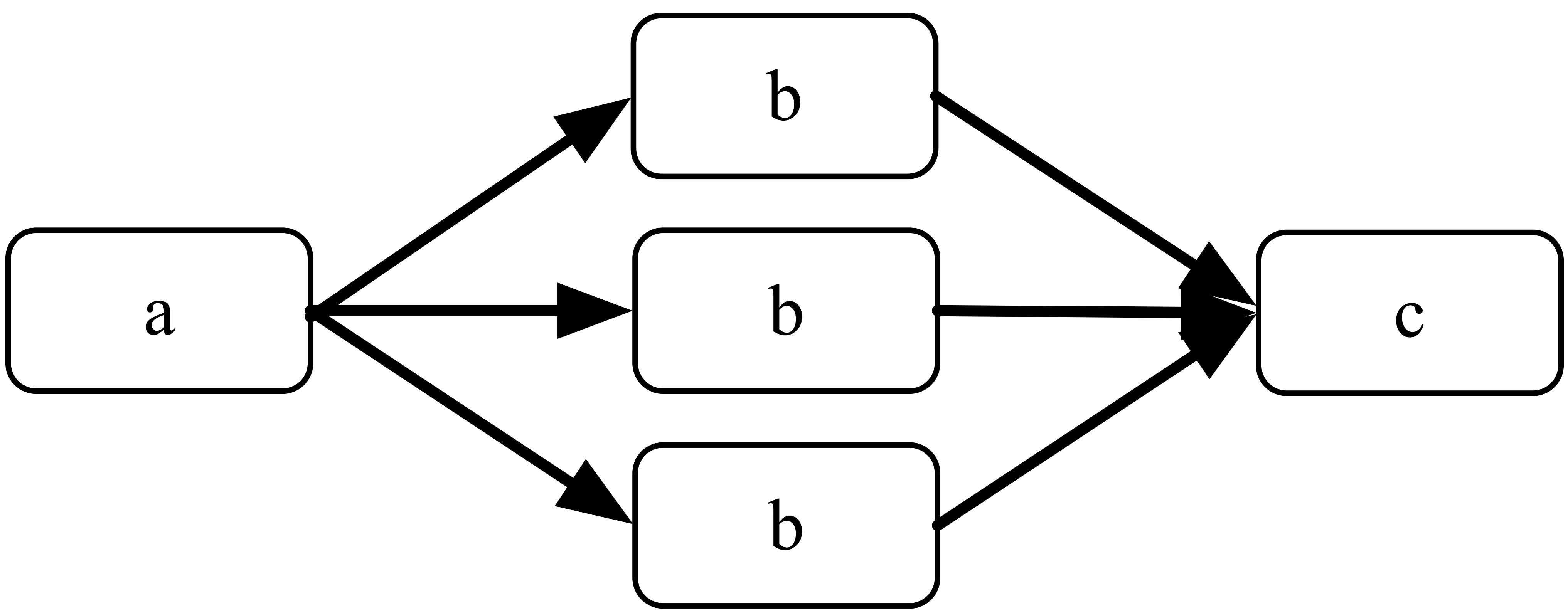
Topology user specifies



CONNECTING COMPUTE KERNELS

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

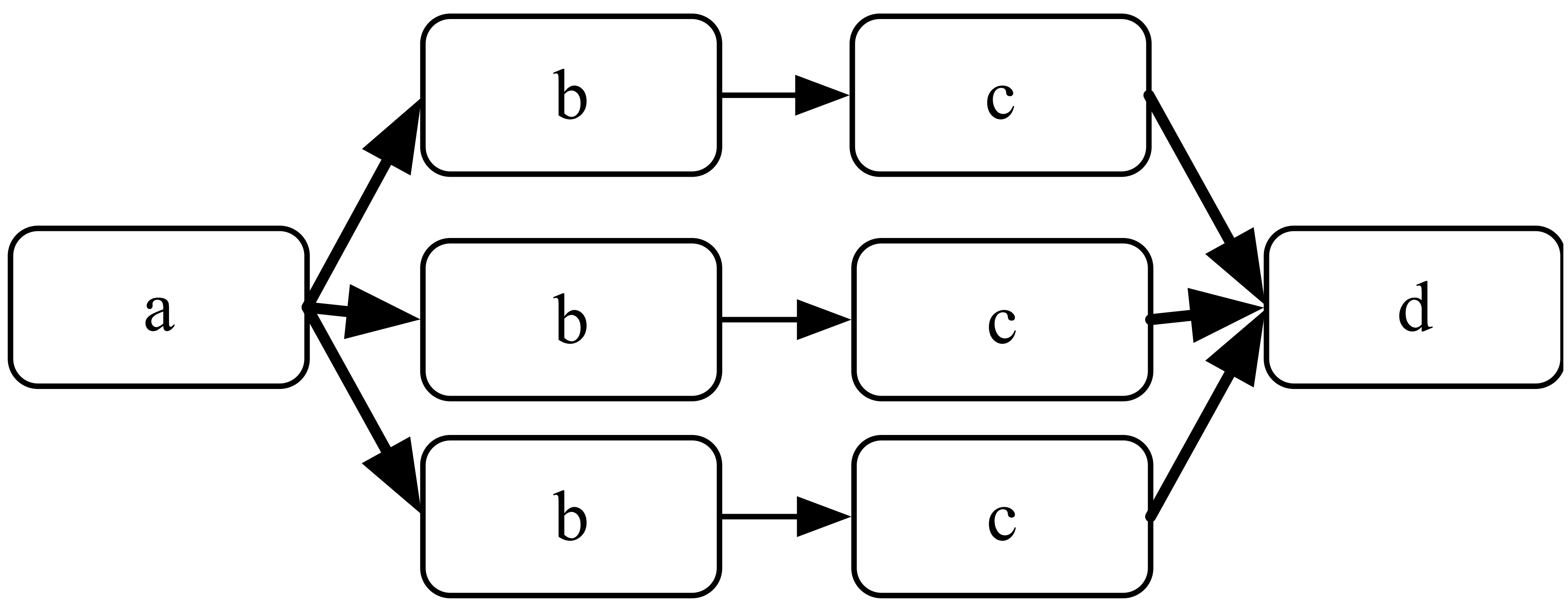
RaftLib Turns Into



CONNECTING COMPUTE KERNELS

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

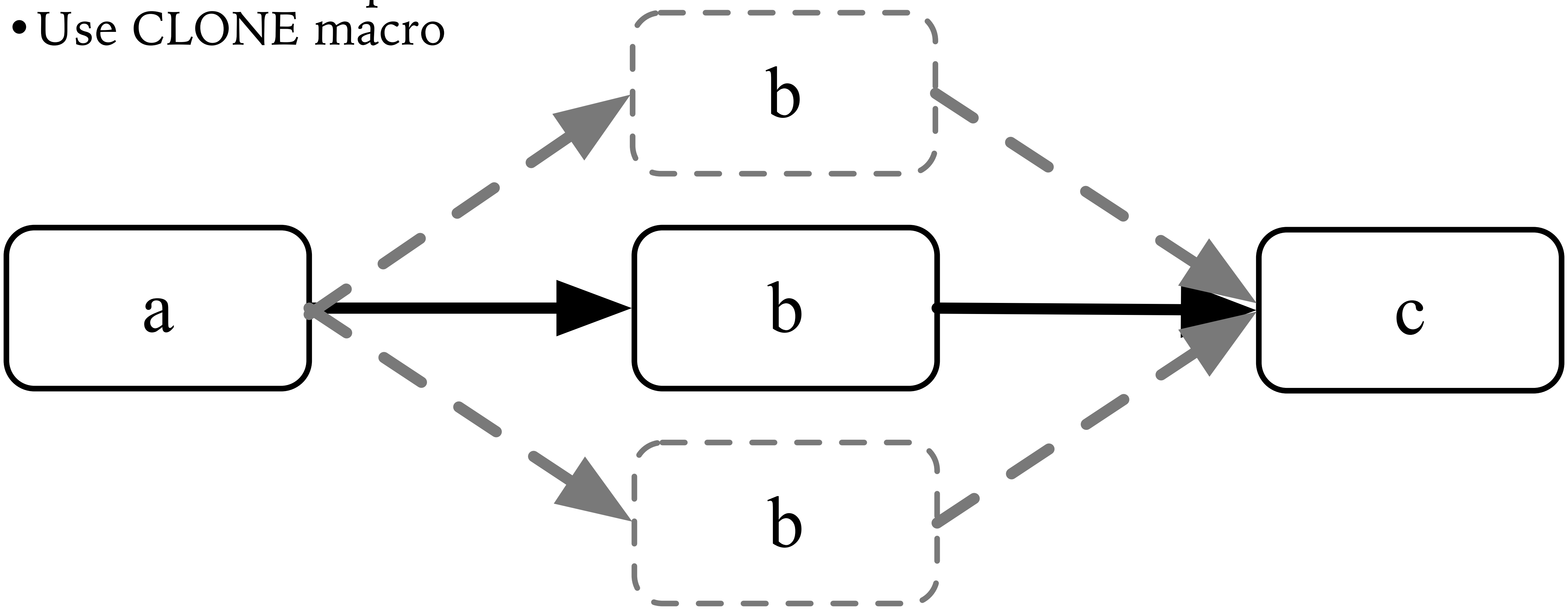
RaftLib Turns Into



CONNECTING COMPUTE KERNELS

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

- Clone at SESE points
- Use CLONE macro

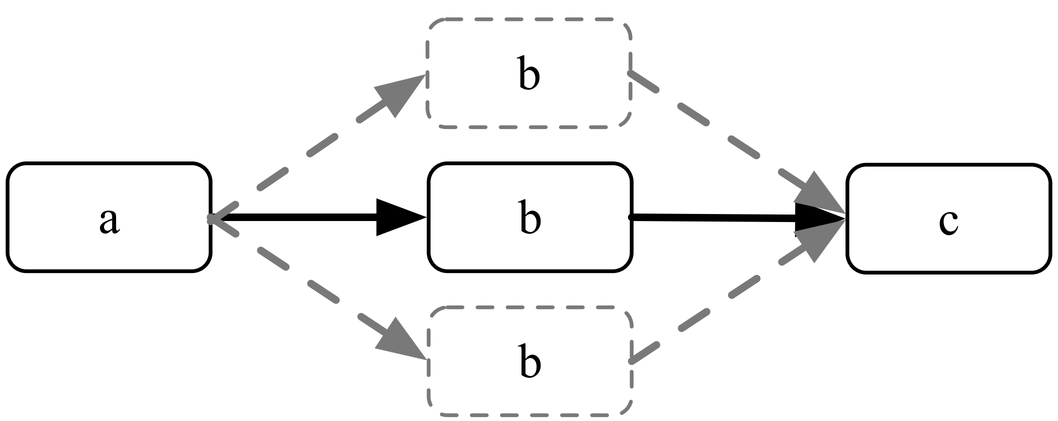


AUTO-PARALLELIZATION

1) RaftLib figures out methodology

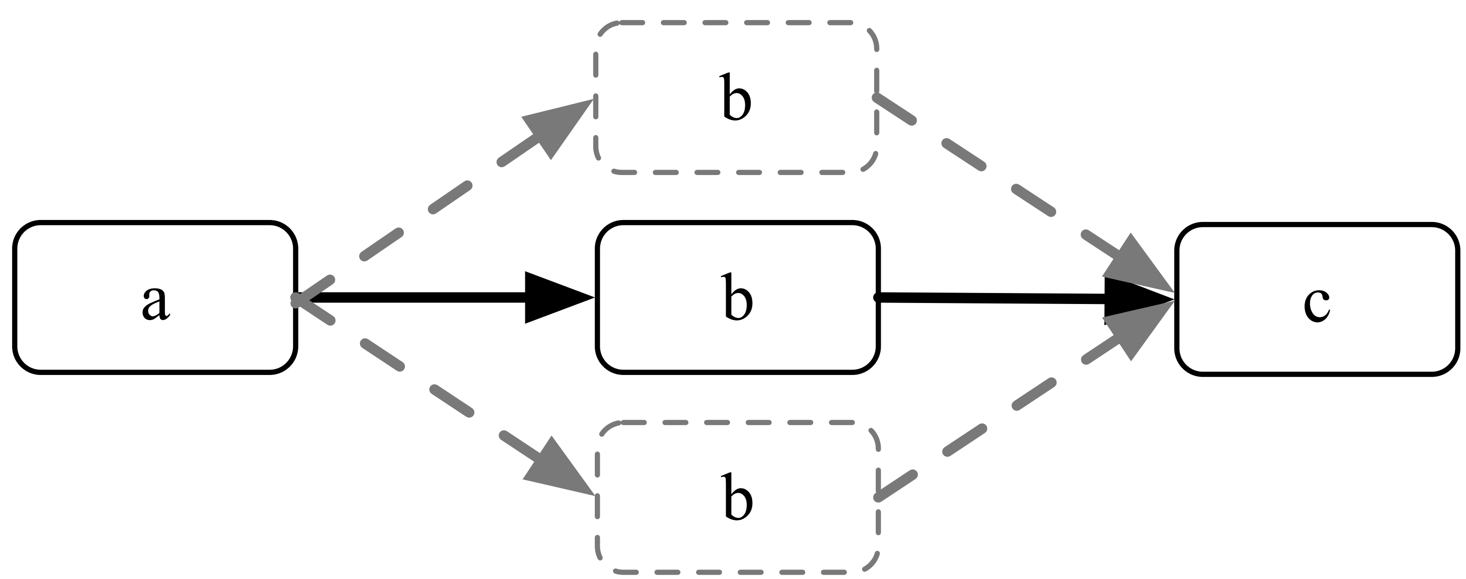
2) Runtime calls CLONE() macro of kernel “b”

```
#define CLONE()\
virtual raft::kernel* clone()\
{ \
   auto *ptr( \
      new typename std::remove_reference< decltype( *this ) >::type( ( *(\
   (typename std::decay< decltype( *this ) >::type * ) \
   this ) ) ) );\
   /** RL needs to dealloc this one **/\
   ptr->internal_alloc = true;\
   return( ptr );\
}
```



AUTO-PARALLELIZATION

3) Lock output port container of “a”
4) Register new port
5) Decide where to run it
6) Allocate memory for stream
7) Unlock output container of kernel “a”
//do same on output side of “b”



CONNECTION TODO ITEMS

- Better SESE implementation
- Decide on syntax for set of kernels

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

- Address space stream modifier (new VM space)

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

- Anything else?

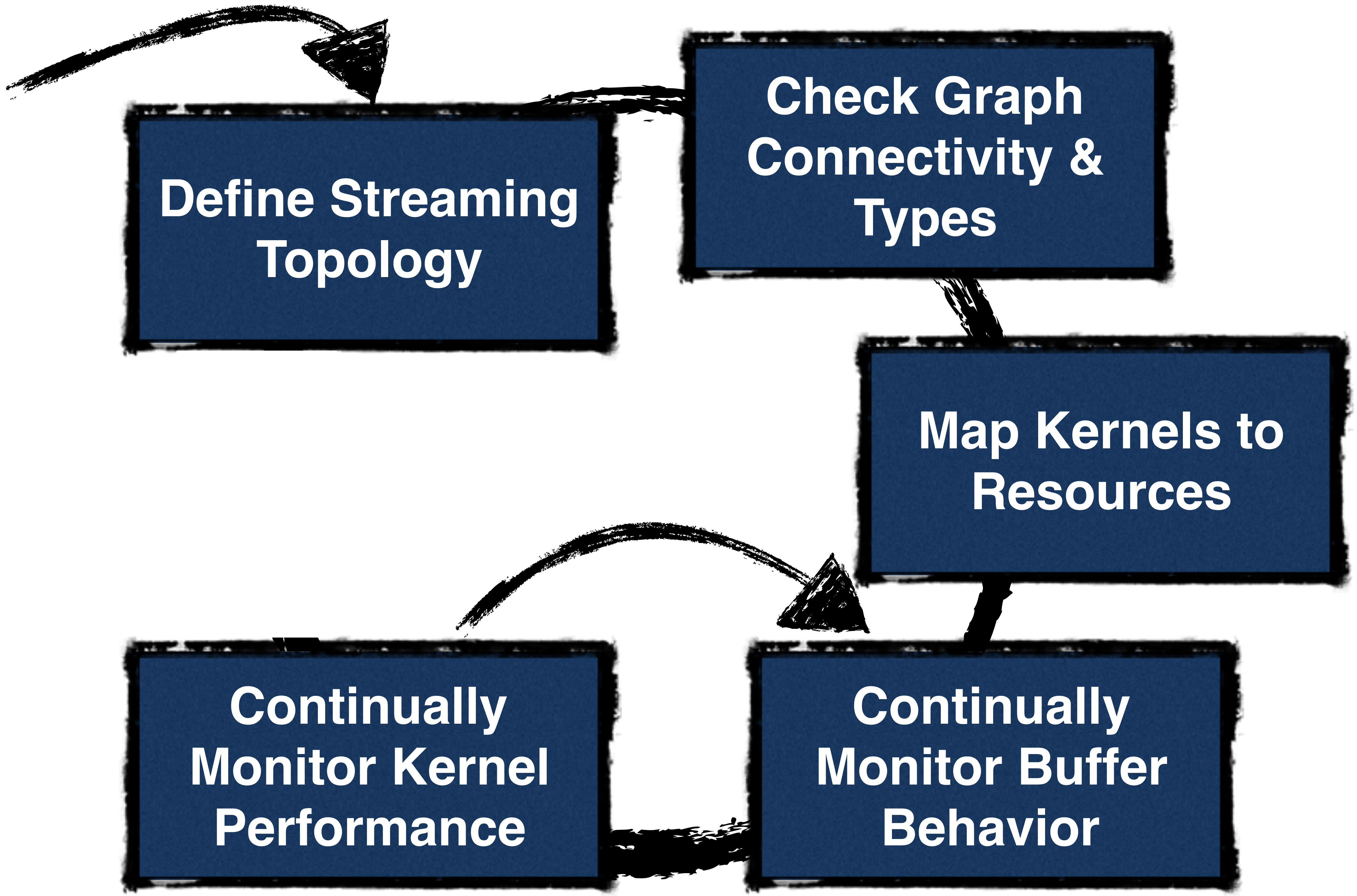
CHOOSE YOUR ADVENTURE

Models

Interface

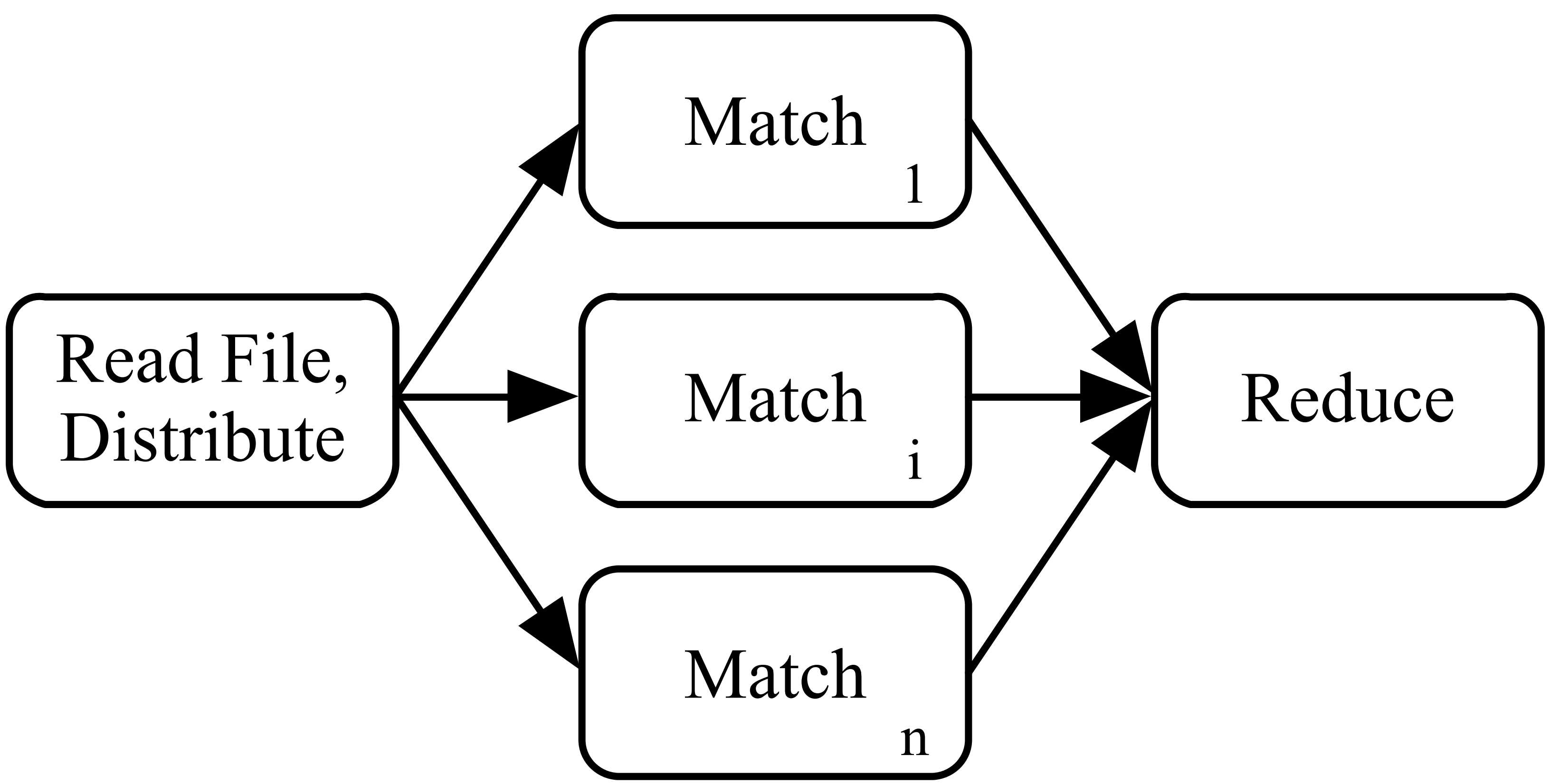
Implementation

IMPLEMENTATION DETAILS



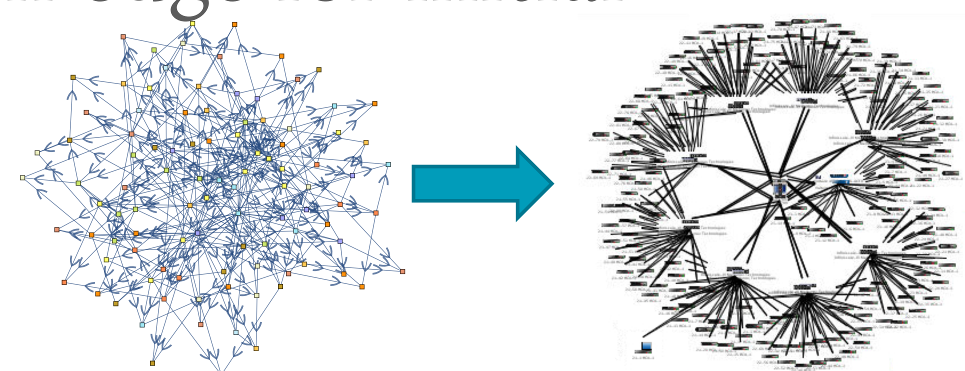
TOPOLOGY CHECK

- Add kernels to map
- Check type of each link (potential for type optimization)
- Handle static split/joins (produce any new kernels)
- DFS to ensure no unconnected edges



PARTITION (LINUX / UNIX)

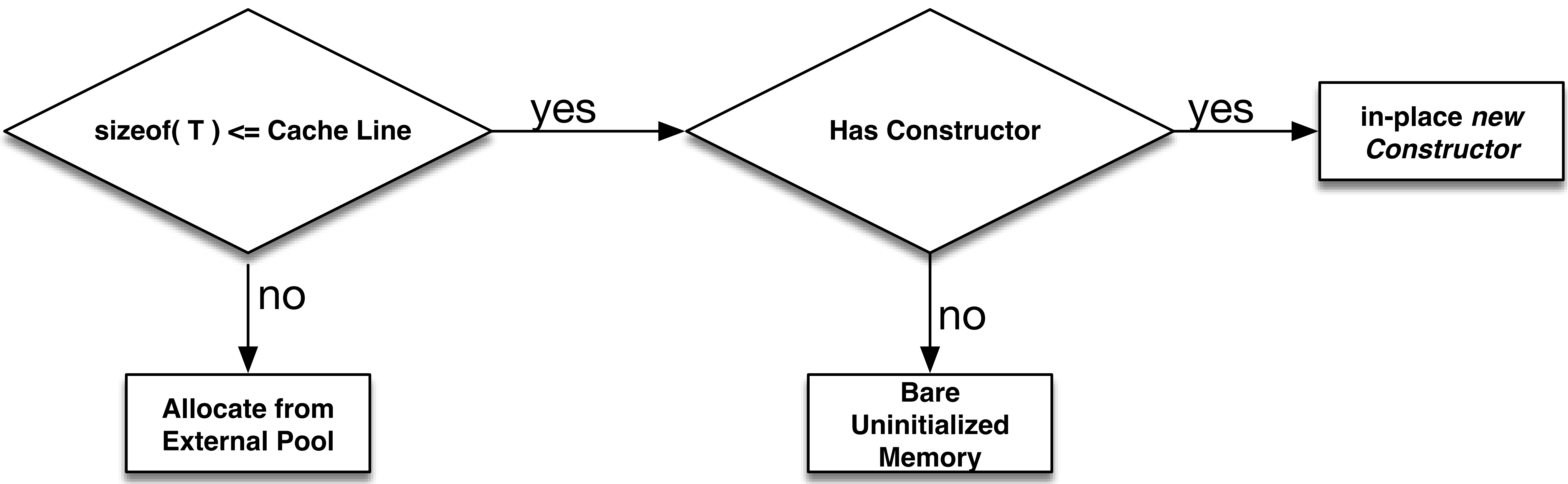
- Take RaftLib representation, convert to Scotch format
- Use fixed communications cost at each edge for initial partition



  - 2 main dimensions:
    - Flow between each edge in the application graph
    - Bandwidth available between compute resource
- Set affinity to partitioned compute cores
- Repartition as needed @ run-time
- TODO: incorporate OpenMPI utilities *hwloc* and *netloc* to get cross-platform hardware topology information

CHOOSE ALLOCATIONS

- Alignment
  - SIMD ops often require memory alignment, RaftLib takes an alignment by default approach for in-stream allocations
- In-stream vs. External Pool Allocate (template allocators)



MONITOR BEHAVIOR - ALLOCATORS

- Two options for figuring out optimal buffer size while running
  - model based (*discussed on modeling adventure path*)
  - branch & bound search
- separate thread, exits when app done
- pseudocode:

```
while( not done )
{
    if( queue_utilization > .5 )
    {
        queue->resize();
    }
    sleep( ALLOC_INTERVAL );
}
```

RUN-TIME LOCK FREE FIFO RESIZING

```
enum access_key : key_t { allocate       = 0,
                          allocate_range = 1,
                          push           = 3,
                          recycle        = 4,
                          pop            = 5,
                          peek           = 6,
                          size           = 7,
                          N };
}
struct ThreadAccess
{
   union
   {
      std::uint64_t whole = 0; /** just in case, default zero **/
      dm::key_t     flag[ 8 ];
   };
   std::uint8_t padding[ L1D_CACHE_LINE_SIZE - 8 /** padding **/ ];
}
#if defined __APPLE__ || defined __linux
__attribute__((aligned( L1D_CACHE_LINE_SIZE )))
#endif
volatile thread_access[ 2 ];

/** std::memory_order_relaxed **/
std::atomic< std::uint64_t >  checking_size = { 0 };
```

RUN-TIME LOCK FREE FIFO RESIZING

- Optimization…wait for the right conditions

```
if( rpt < wpt )
{
    //perfect to copy w/std::memcpy
}
```

- Factory allocators

```
/** 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
```

MONITOR BEHAVIOR – PARALLELIZATION

- Mechanics covered in interface, simple model here
- Run in separate thread, term on exit

```
/** apply criteria **/
if( in_utilization > .5 && out_utilization < .5 )
{
    //tag kernel
    auto &tag( ag[ reinterpret_cast< std::uintptr_t >( kernel ) ] );
    tag += 1;
    if( tag == 3 )
    {
        dup_list.emplace_back( kernel );
    }
}

//after checking all kernels, handle duplication
```

IMPLEMENTATION TODO ITEMS

- Find fast SVM library to integrate (research code used LibSVM) for buffer model selection
- Integrate more production-capable network flow model for run-time re-partitioning choices
- Performant TCP links….
- RDMA on wish list
- QThreads Integration (see pool scheduler)
- *hwloc* and *netloc* integration (see partition_scotch)
- Perf data caching (useful for initial partition)

CHOOSE YOUR ADVENTURE

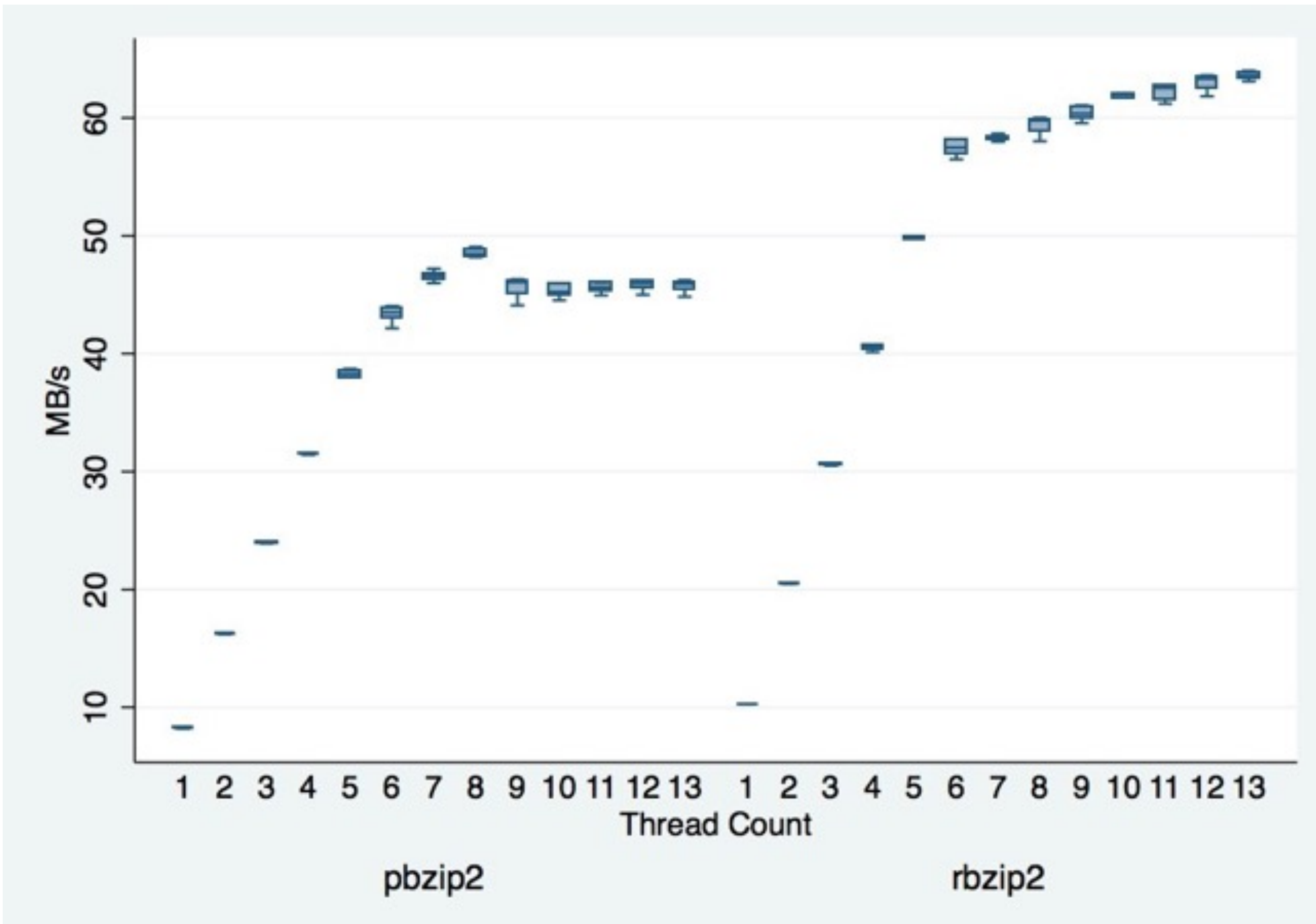
Interface

Models

Implementation

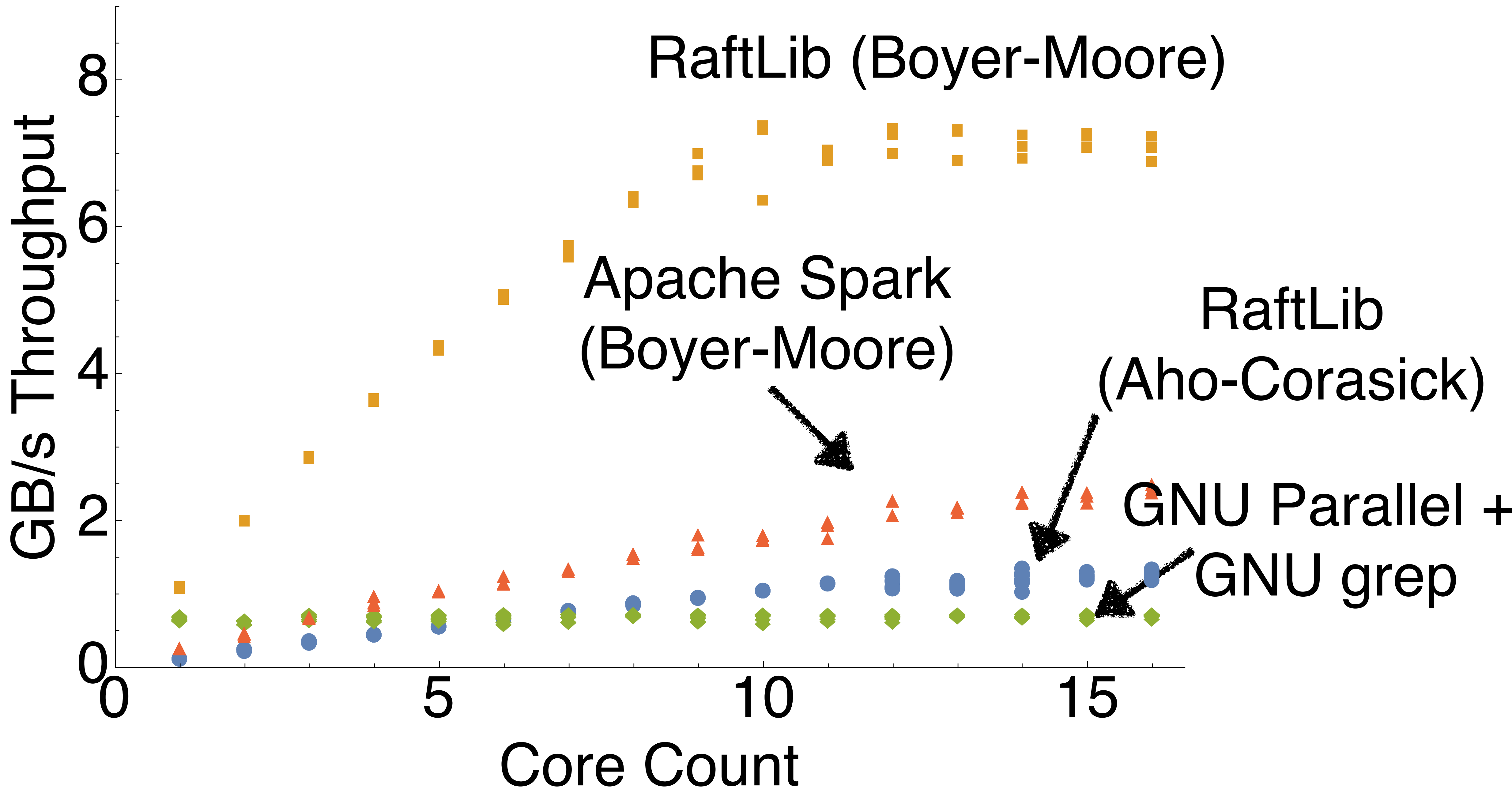
PERFORMANCE

- Decent compared to pthread stock implementation of pbzip2
- Parallel Bzip2 Example: https://goo.gl/xyQAhm



PERFORMANCE

- Fixed string search compared to Apache Spark, GNU Parallel + GNU Grep




POINT OF NO RETURN

ABOUT ME

my website

http://www.jonathanbeard.io

slides at

http://goo.gl/cwT5UB

project page

raftlib.io

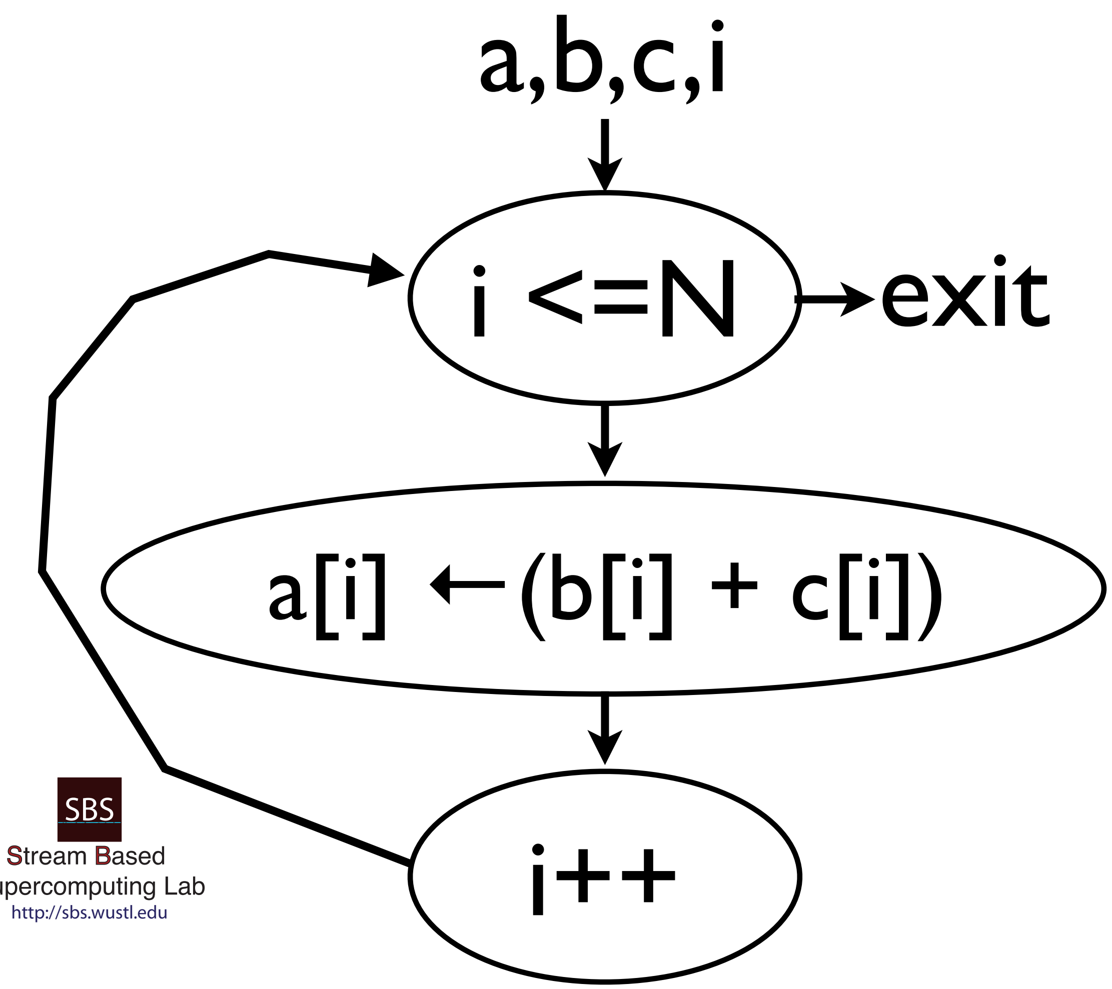
video of talk given at #CppNow2016

http://goo.gl/mbxAwK

Stream Processing

```
for i←0 through N do
   a[i] ←(b[i] + c[i])
   i++
end do
```

Traditional Control Flow



Streaming