Flow

A Multithreaded KPN language

Mitchell Gouzenko | Hyonjee Joo | Adam Chelminski | Zach Gleicher

Determinism in KPNs

- Determinism is important - there are many subtle ways to break it
  - Mutable global variables
  - Mutable objects
  - Querying channels for size
- Important to keep in mind as we developed language features

The Basics: Functions

```
int foo(char bar, double baz){

    …

}
```

Functions are declared c-style, with their return types and arguments

The Basics: Lists

- Declaration and initialization:

```
list<int> foo = [1, 2, 3];
```

- Adding to the head returns a new list:

```
foo = 0::foo;  // foo is [0, 1, 2, 3]
```

- Getting the tail:

```
foo = ^foo;  // foo is [1, 2, 3]
```

- Getting the length:

```
int length = #foo;  // length = 3
```

The Basics: The Main Entrypoint

- The entry point into a flow program is the main function
- Must be declared as `int main(){ … }`
- The first process invocations take place in main
  - Functions and processes can then go on to invoke other processes
- Under normal circumstances, programmer SHOULD NOT return from main
  - Return from main will cause the entire Kahn Process Network to stop running
    - Useful in the event of an unfavorable condition
    - Allows other programs to check exit value

The Basics: Processes

```
proc foo(in int bar, out int baz){

    …

}
```

- Declared like functions, but with “proc” pseudo return value
- Procs can accept channels as arguments
  - Only “in” channels can be read from, and only “out” channels can be written to
- Invoking a process is just like invoking a function
  - Difference is that process starts on a separate thread
- `return` – can be issued to stop the process

H

The Basics: Working with Channels

```
channel<int> foo;
```

- Declaring a channel heap-allocates it immediately
- Channels can be written to and read from processes
- Writing to channels:

```
1 -> foo;
```

- Reading from channels:

```
int bar = @foo;
```

H

Program Termination

- When does the program end?
    - When all channels are empty?
        - Wrong - channels might be empty, but a process might be about to issue a write
    - SIGINT? Timer?
        - Not elegant - processes might be in the middle of performing work
    - When all threads are terminated?
        - But how does a process know when to stop?
    - Solution: programmer can explicitly poison channels

M

Channel Poisoning

```
poison ochan;
```

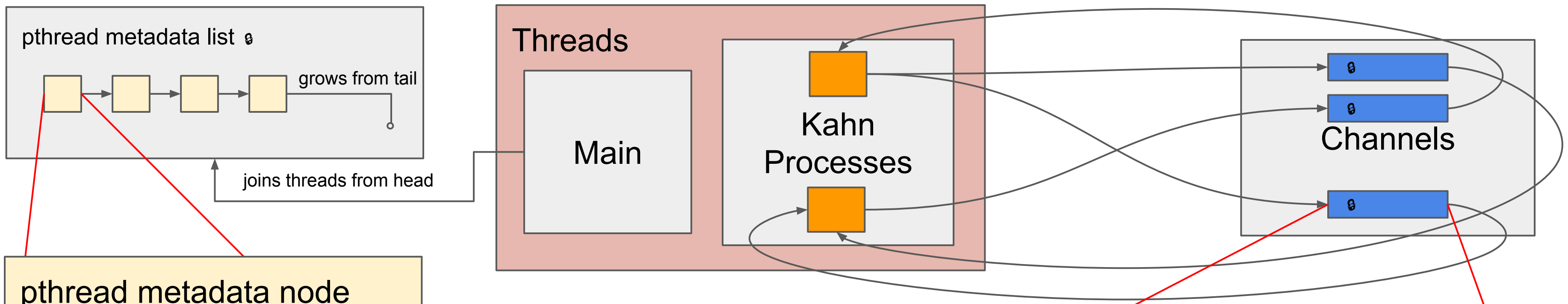
- Forbids future writes
- Reading process can consume remaining tokens by checking if channel “alive”
    - Checking is done in conditionals

```
if(chan){ … }      while(chan){ … }
```

- Channels used in a boolean context
    - Evaluate to “false” only if channel is poisoned, and there are no tokens left
    - Evaluate to “true” if channel has tokens to read
        - Blocks if channel is empty (but not poisoned)
        - Reading from an empty channel is a runtime error
- If a process exits without poisoning its channels, they’re poisoned automatically

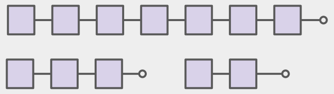
H

The Runtime Environment



pthread metadata node

- thread id
- name of running proc
    - used for printing read/write violations (i.e. two processes interacting with the same end of a channel)
- list of channels claimed for writing
    - channels are poisoned if process exits without explicitly doing so.



Lists

- Immutable - so processes cannot use them to communicate.
- Grow from head only
- Linked list; O(1) to add

Channel Struct

- channel lock - must be acquired for reading, writing, and poisoning
- wrap around queue of tokens
- read-ready and write-ready condition variables
- thread ids of reading/writing threads, used to enforce runtime checks that each channel is used by no more than one thread on each end.

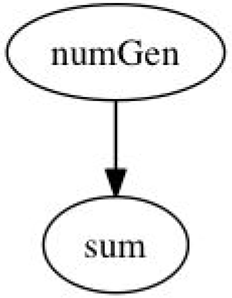
The Runtime Environment

- Only one process can write to a channel, and only one can read from it
    - Enforced at runtime; impossible to enforce at compile time due to lists of channels
    - First process that reads/writes to a channel “claims” the channel for reading/writing
    - Thread id of process that issued read/write must always be checked for validity
- FIFO implemented as a producer-consumer queue with condition variables
    - Poisoning a channel amounts to setting a single flag
- Added bonus: runtime can detect when both ends of a channel are bound
    - Allows visualization of the KPN via Graphviz’s Dot
    - Feature is built into the runtime and toggled with “-d” during compilation
        - Outputs KPN in dot format through stderr

M

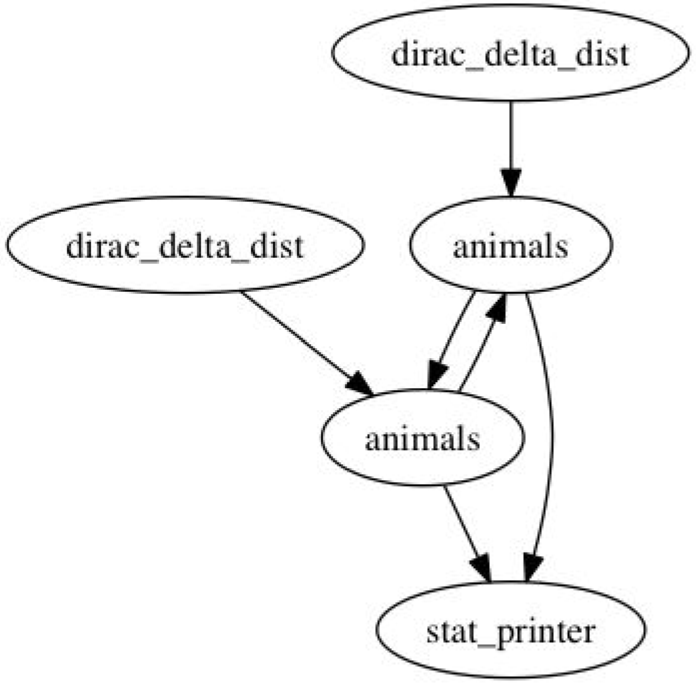
Demo Programs

Sum



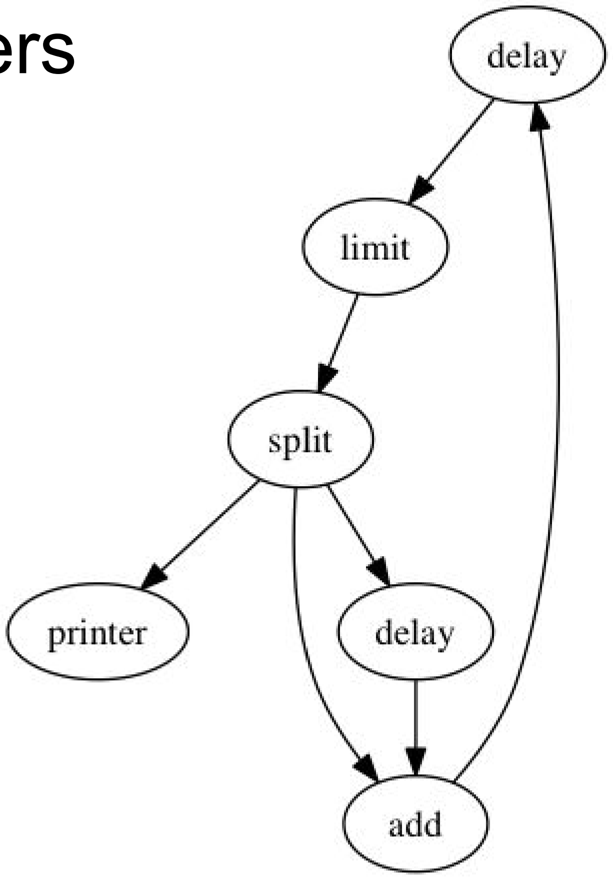
```
proc numGen(out int ochan){
  list <int> test = [1, 2, 3, 4, 5];

  while(#test > 0) {
    @test -> ochan;
    test = ^test;
  }

  poison ochan;
}

proc sum(in int chan) {
  int sum = 0;
  while(chan) {
    sum = sum + @chan;
  }
  print_int(sum);
}

int main() {
  channel<int> chan;
  numGen(chan);
  sum(chan);
}
```

Population Modeling



H

Fibonacci numbers

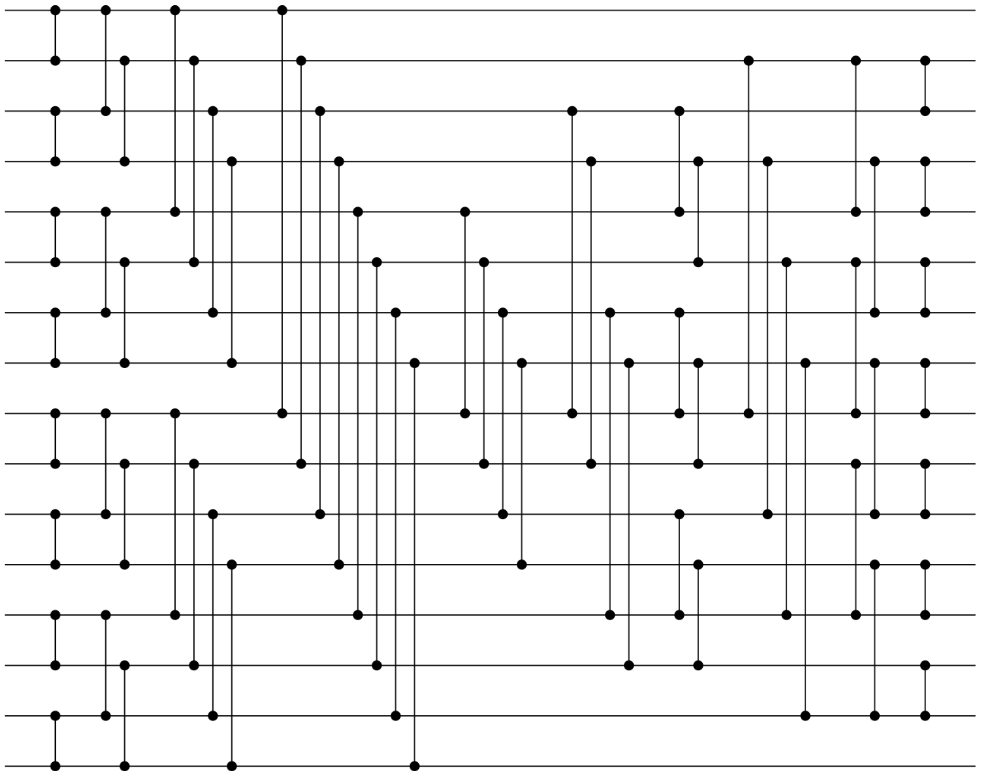


A

Recursive Demo Programs

Bitonic Sort: How it Works

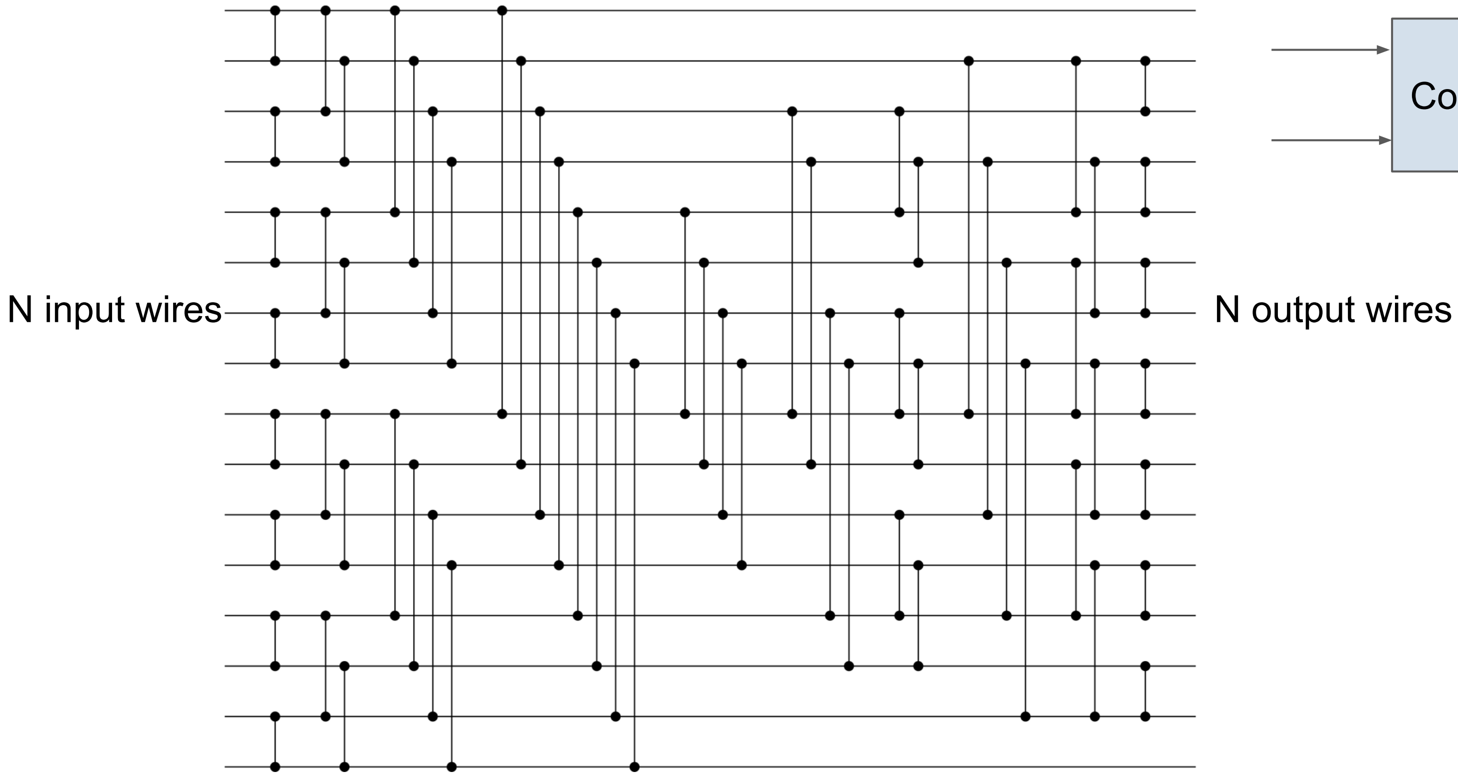
N input wires

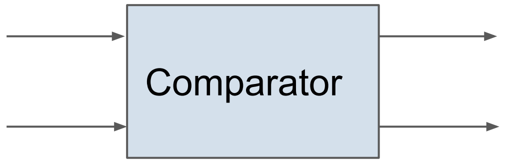


N output wires

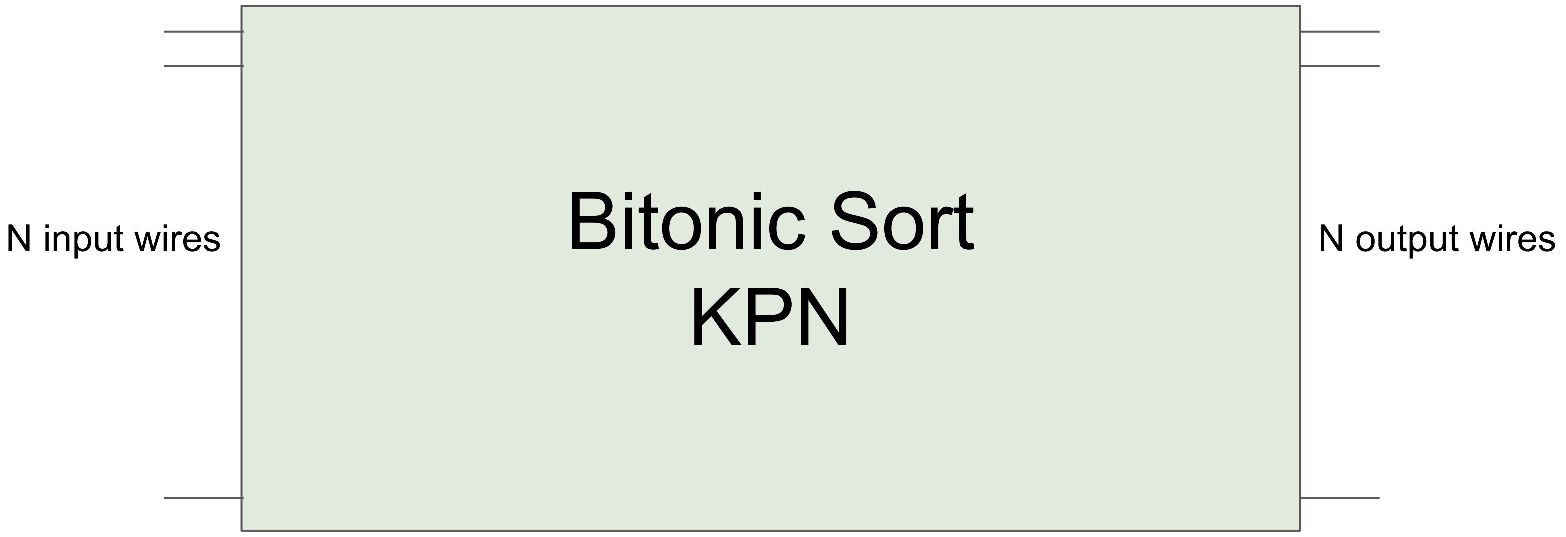
A

Bitonic Sort: How it Works



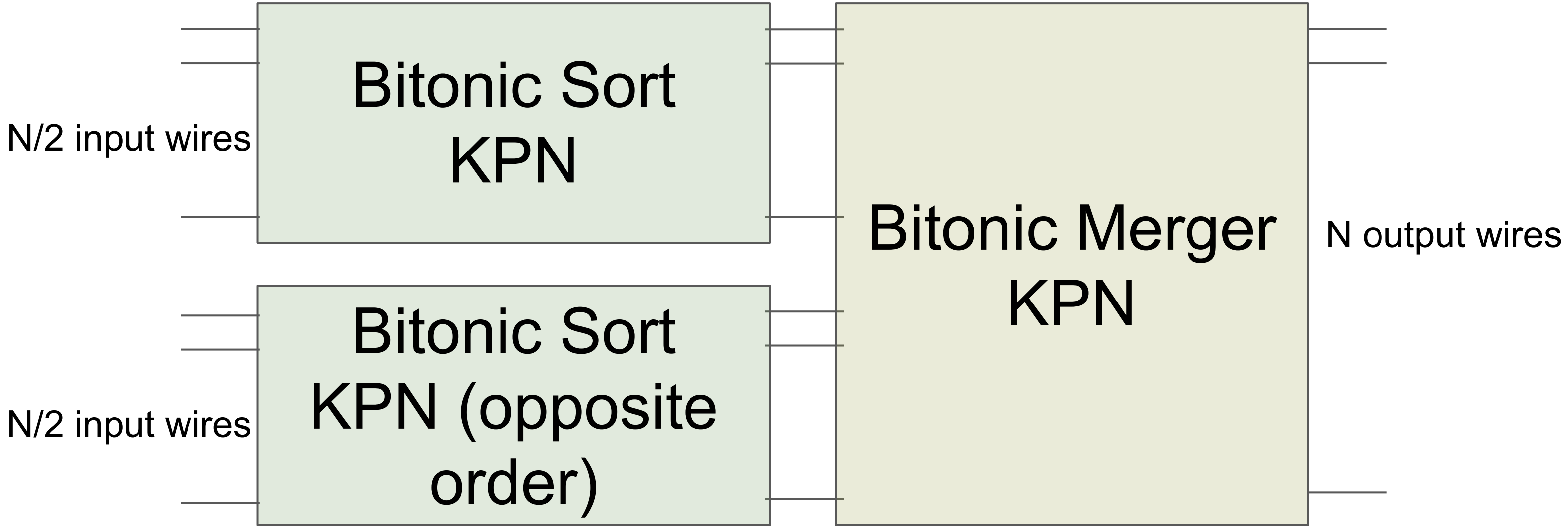


Bitonic Sort: How it Works

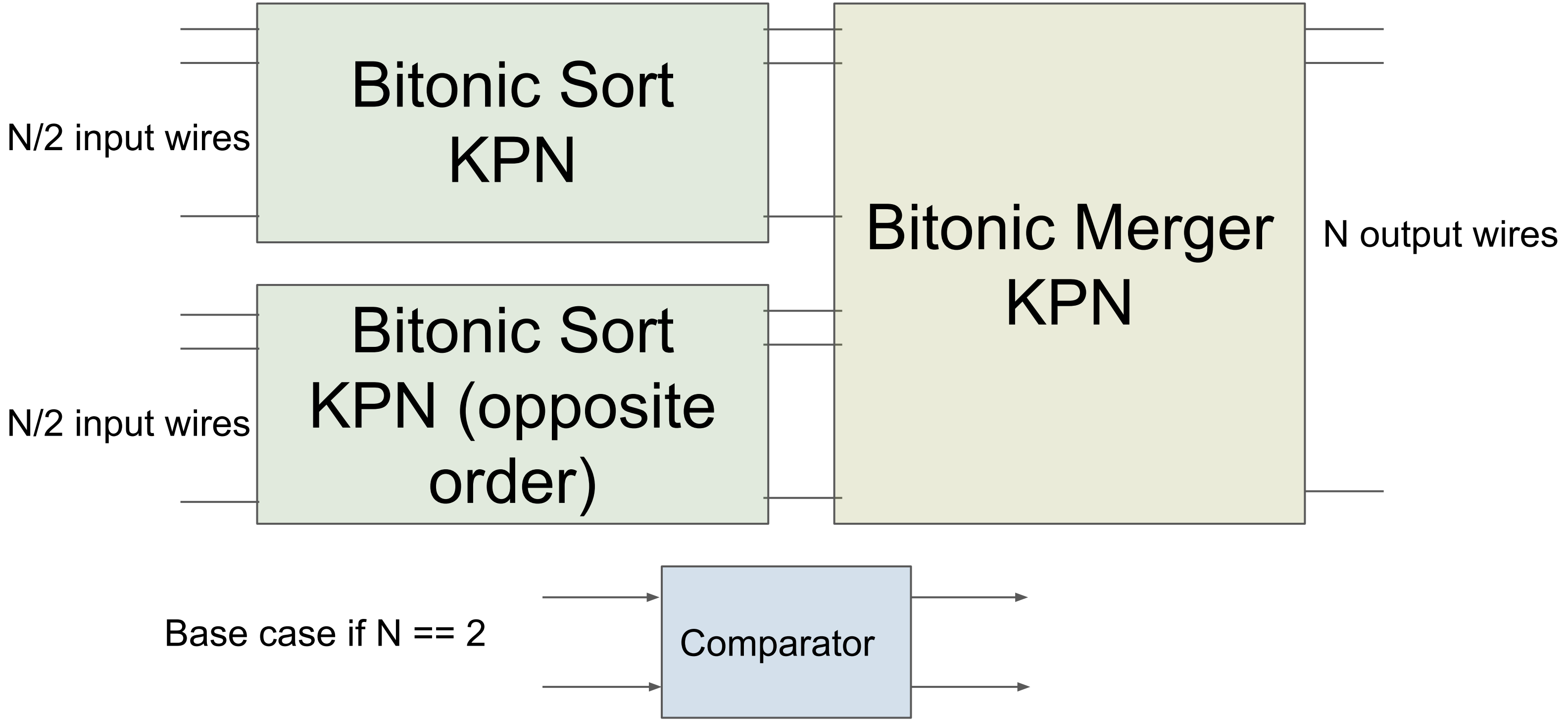


A

Bitonic Sort: How it Works

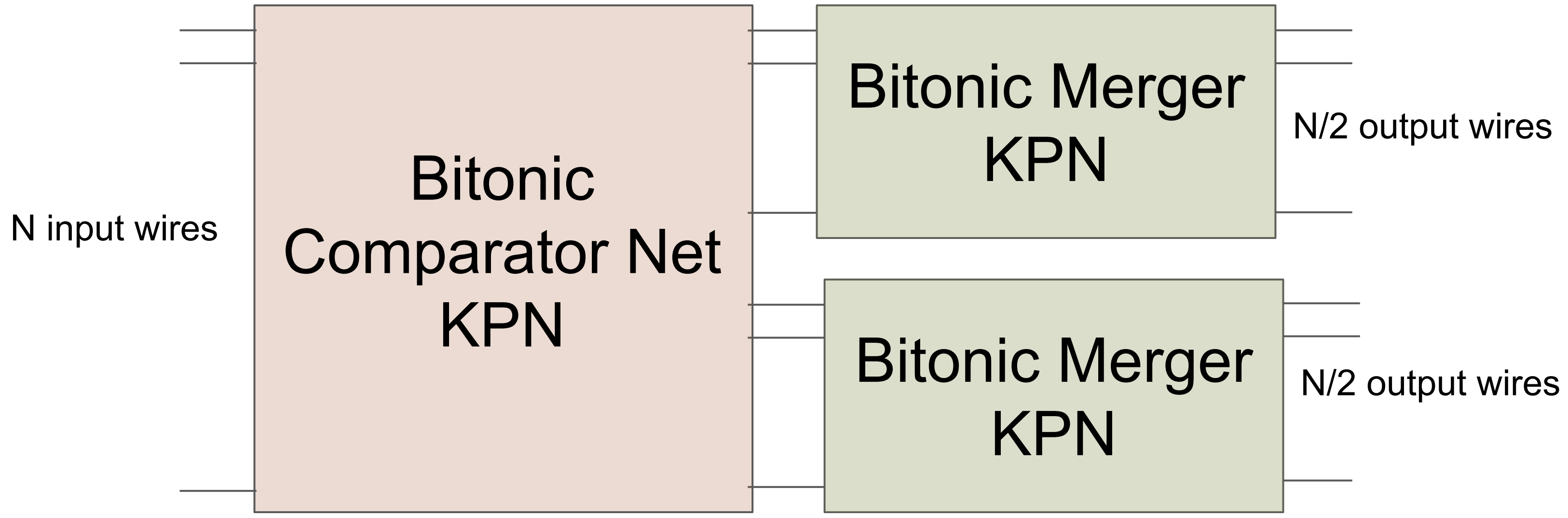


Bitonic Sort: How it Works

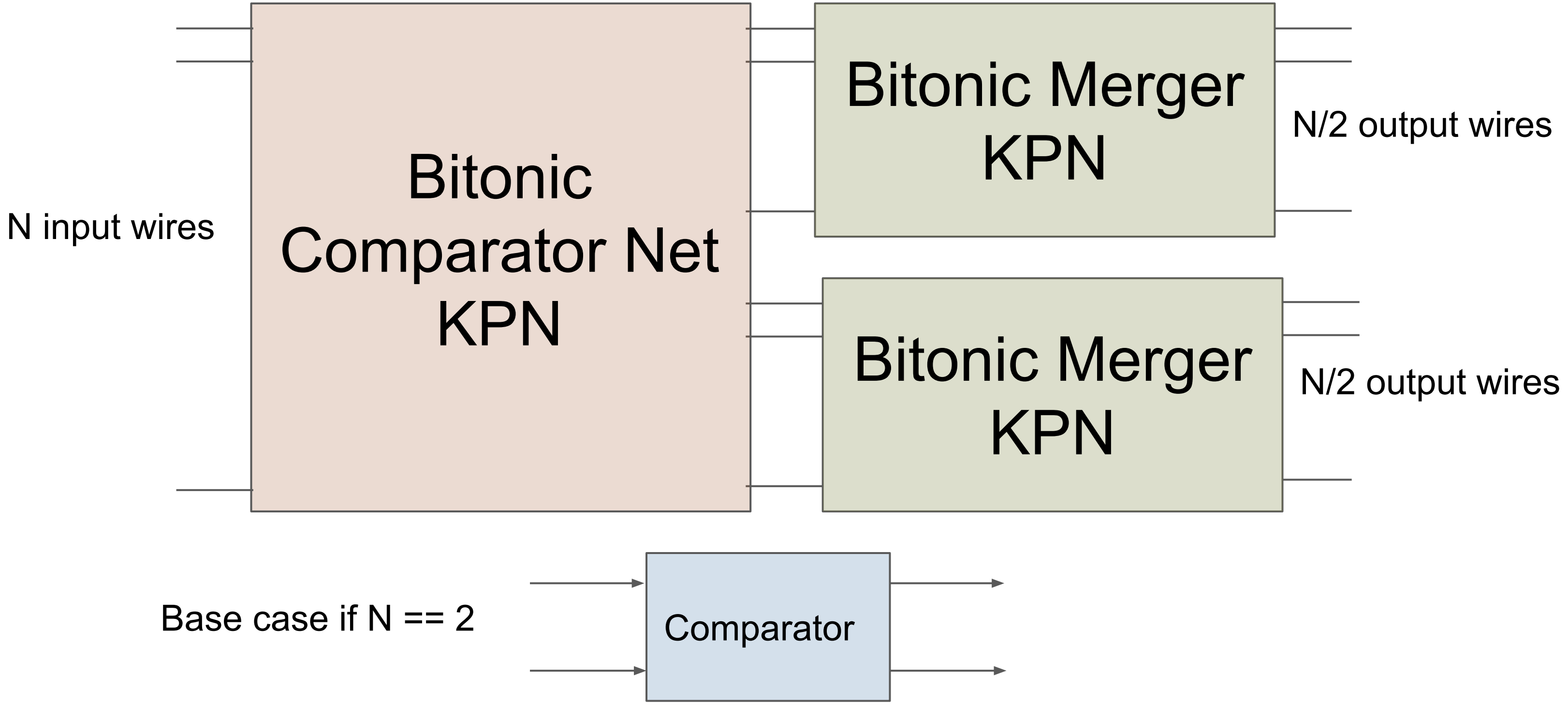


A

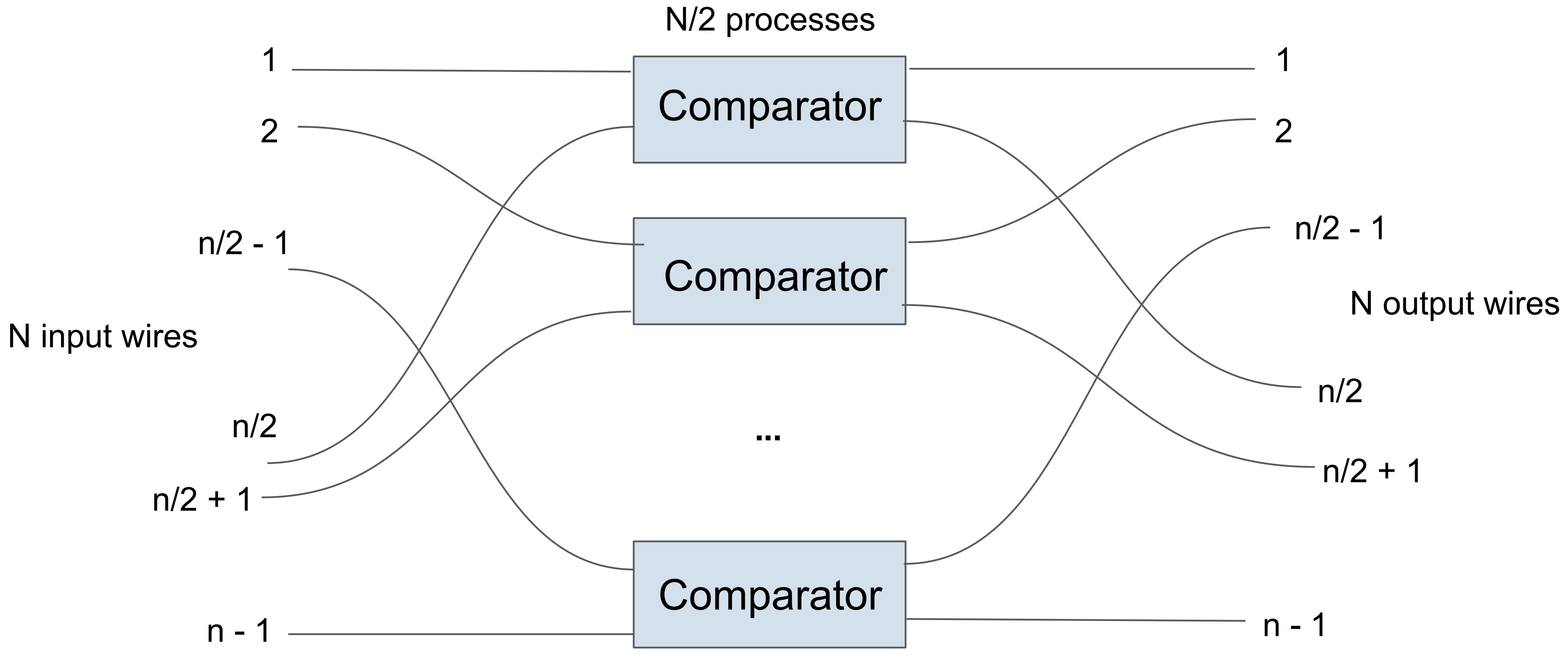
Bitonic Merger: How it Works



Bitonic Merger: How it Works

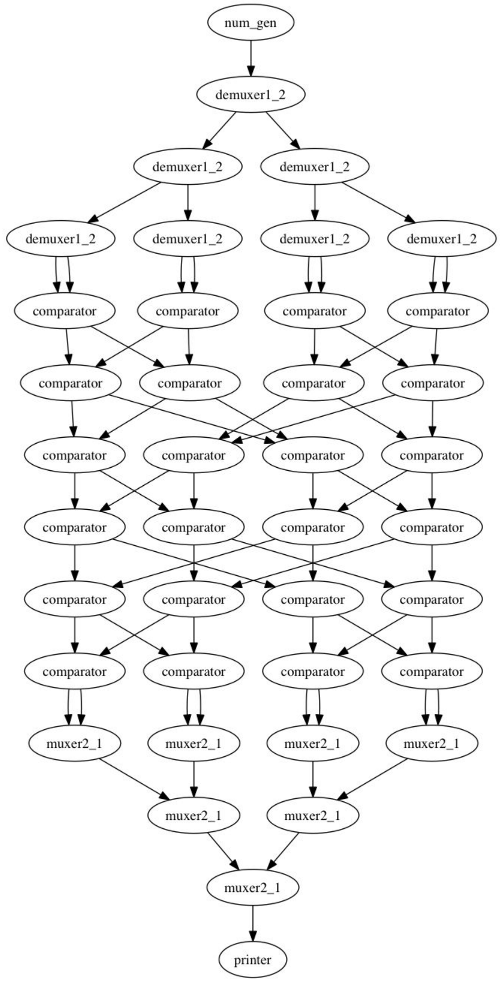


Bitonic Comparator Network: How it Works

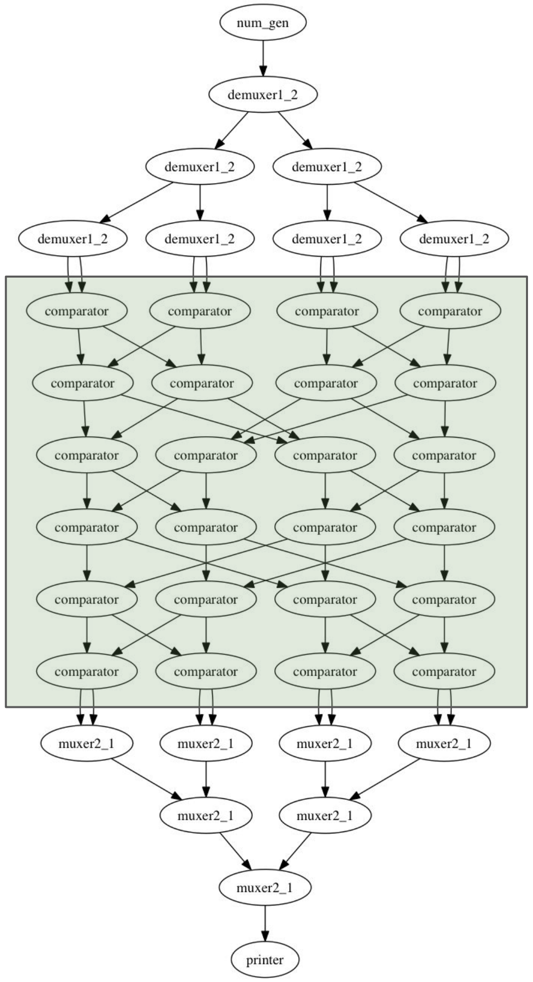


Recursive Bitonic Sort (8 elements)

Approx. 250 lines of flow translates into 830+ lines of c



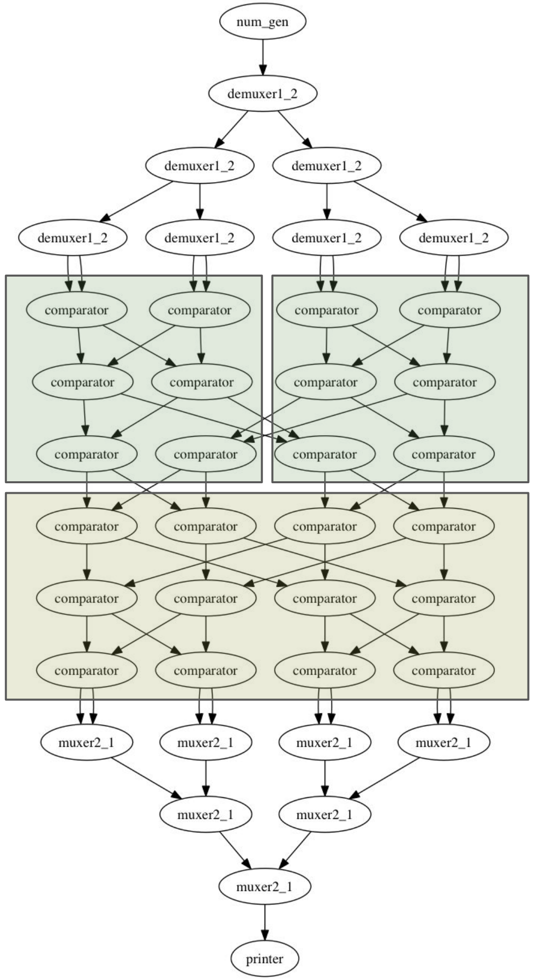
Recursive Bitonic Sort (8 elements)



Bitonic Sorter KPN (size 8)

A

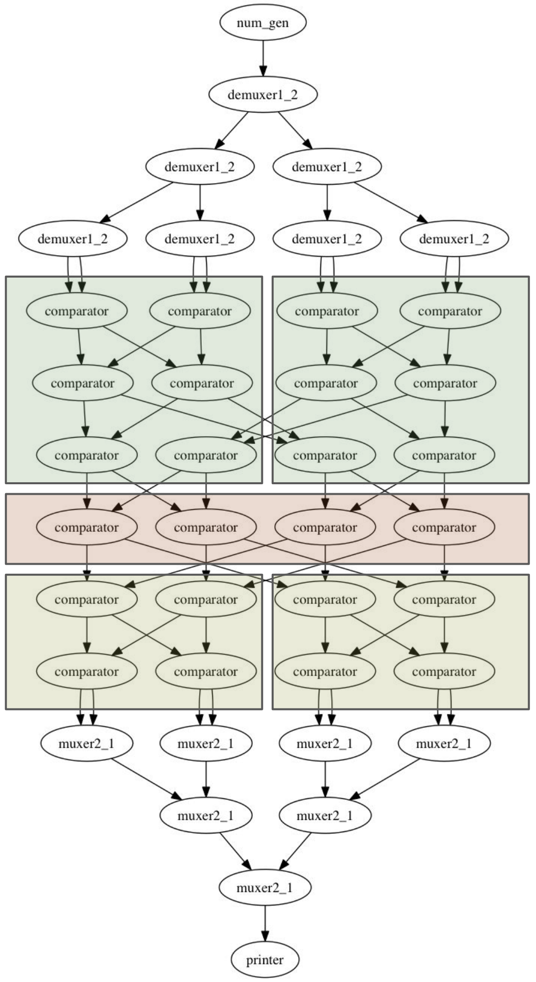
Recursive Bitonic Sort (8 elements)



Bitonic Sorter KPNs (size 4)

Bitonic Merger KPN (size 8)

Recursive Bitonic Sort (8 elements)

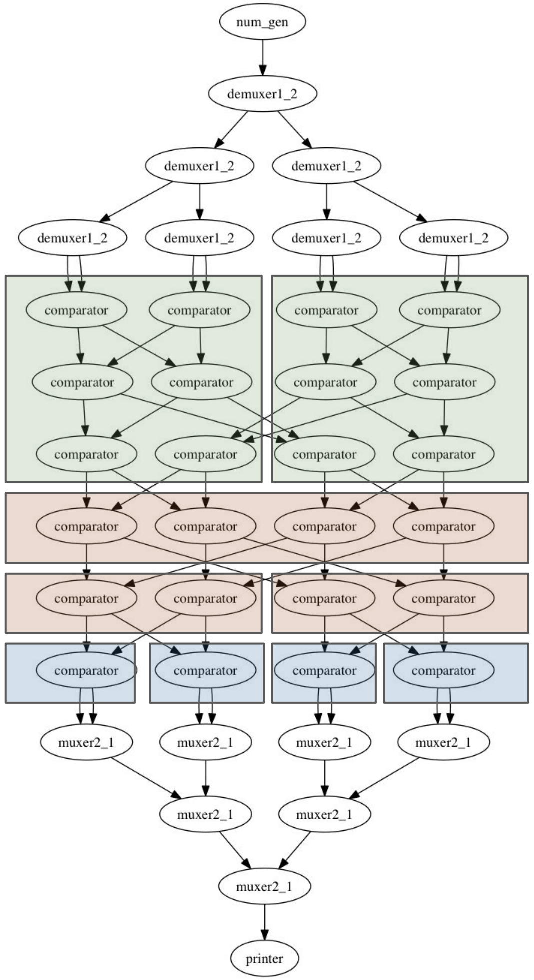


Bitonic Sorter KPNs (size 4)

Bitonic Comparator Net KPN (size 8)

Bitonic Merger KPNs (size 4)

Recursive Bitonic Sort (8 elements)



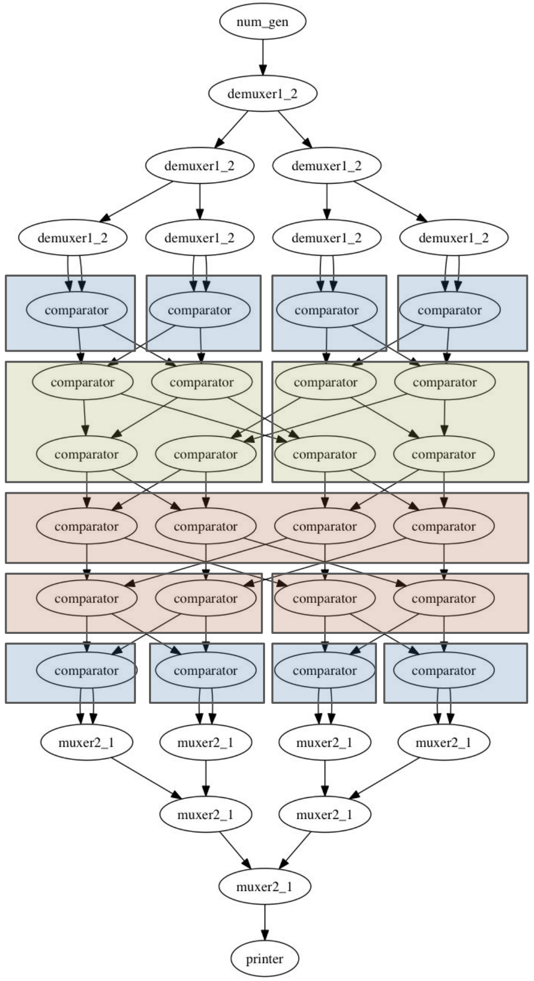
Bitonic Sorter KPNs (size 4)

Bitonic Comparator Net KPN (size 8)

Bitonic Comparator Net KPNs (size 4)

Comparator Processes

Recursive Bitonic Sort (8 elements)



Comparator Processes

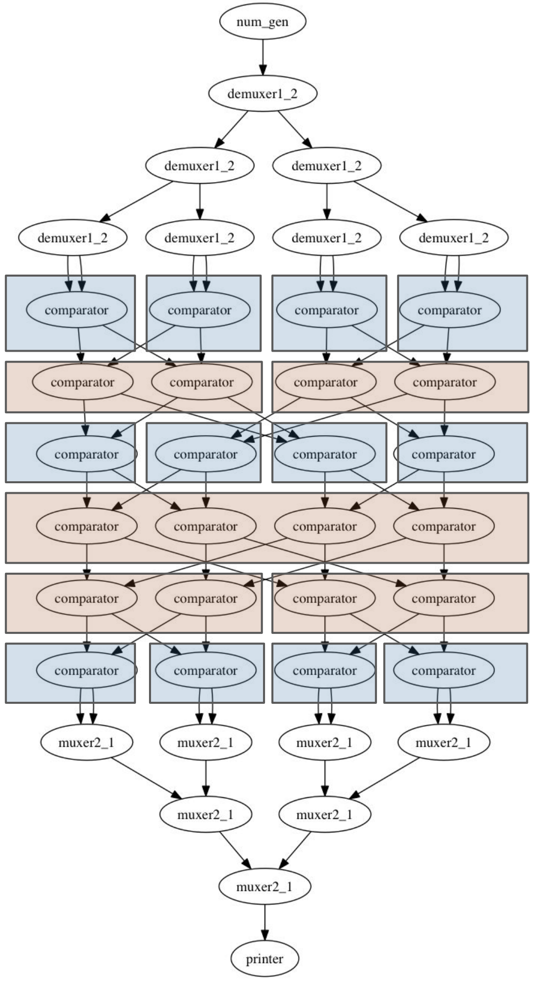
Bitonic Merger KPNs (size 4)

Bitonic Comparator Net KPN (size 8)

Bitonic Comparator Net KPNs (size 4)

Comparator Processes

Recursive Bitonic Sort (8 elements)



Comparator Processes

Bitonic Comparator Net KPNs (size 4)

Comparator Processes

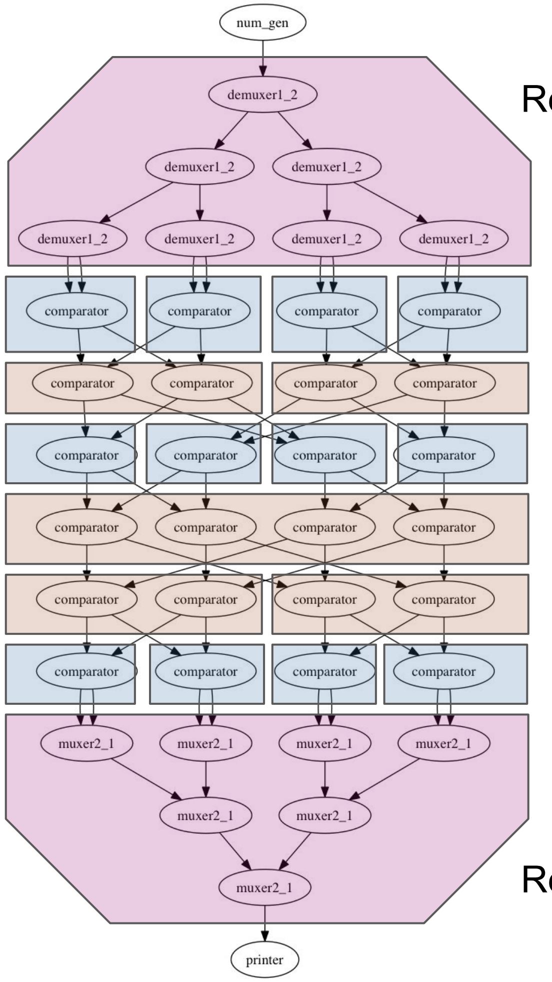
Bitonic Comparator Net KPN (size 8)

Bitonic Comparator Net KPNs (size 4)

Comparator Processes

A

Recursive Bitonic Sort (8 elements)



Recursively generated demuxer

Comparator Processes

Bitonic Comparator Net KPNs (size 4)

Comparator Processes

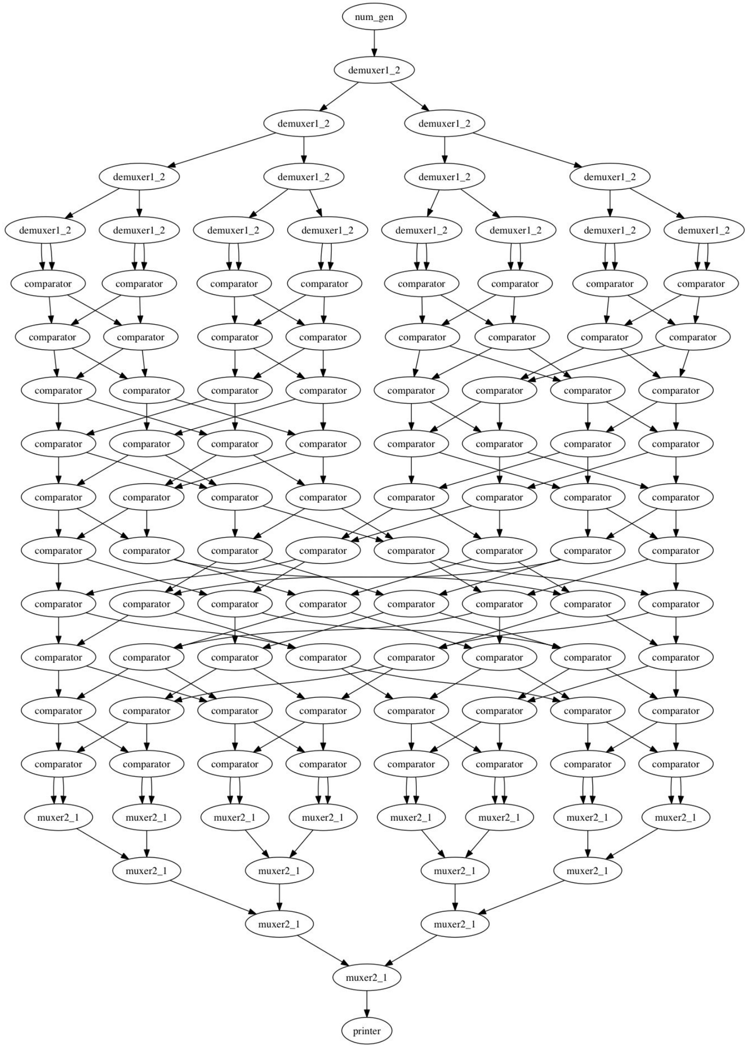
Bitonic Comparator Net KPN (size 8)

Bitonic Comparator Net KPNs (size 4)

Comparator Processes

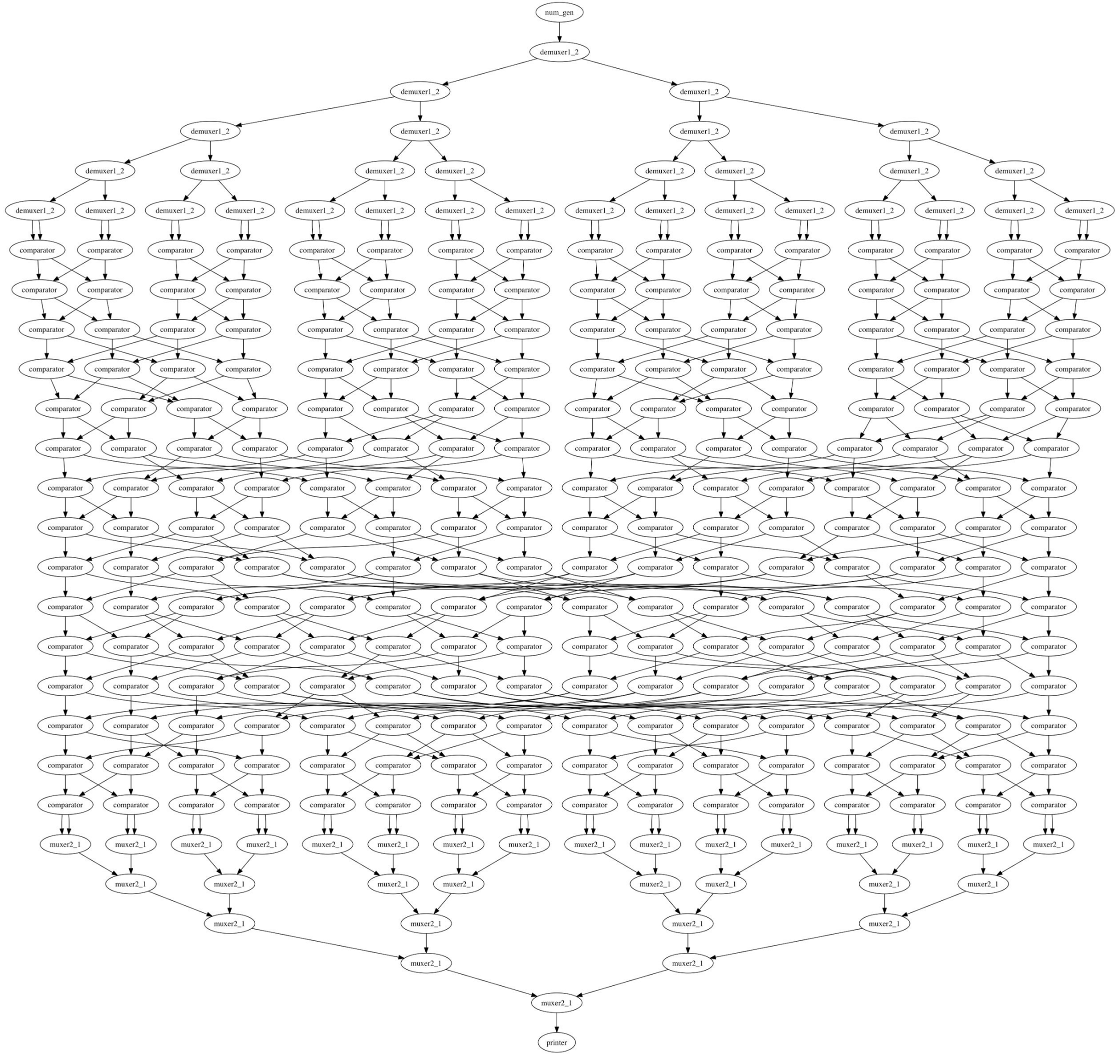
Recursively generated muxer

Recursive Bitonic Sort (16 elements)



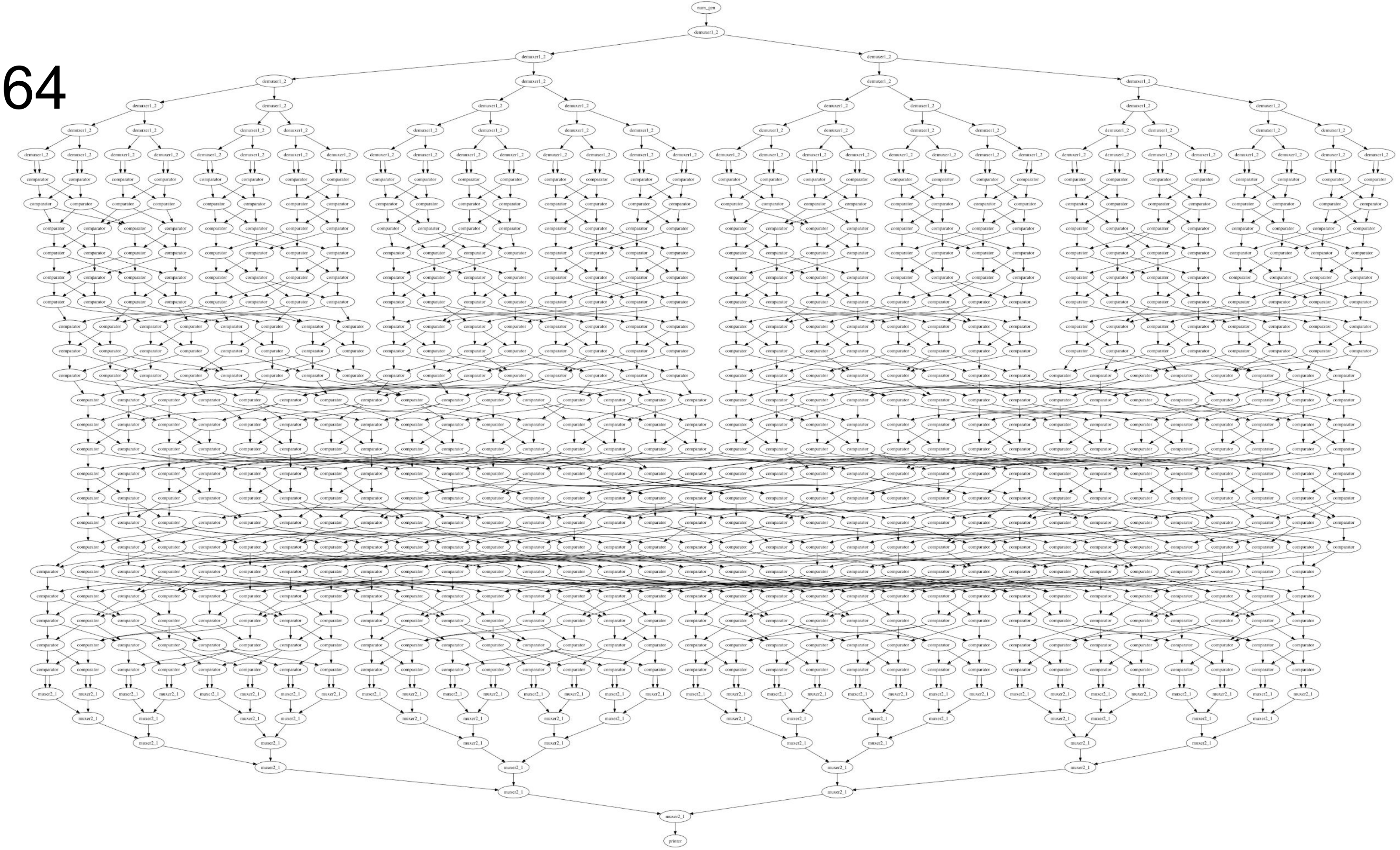
A

32



A

64



A

To Do

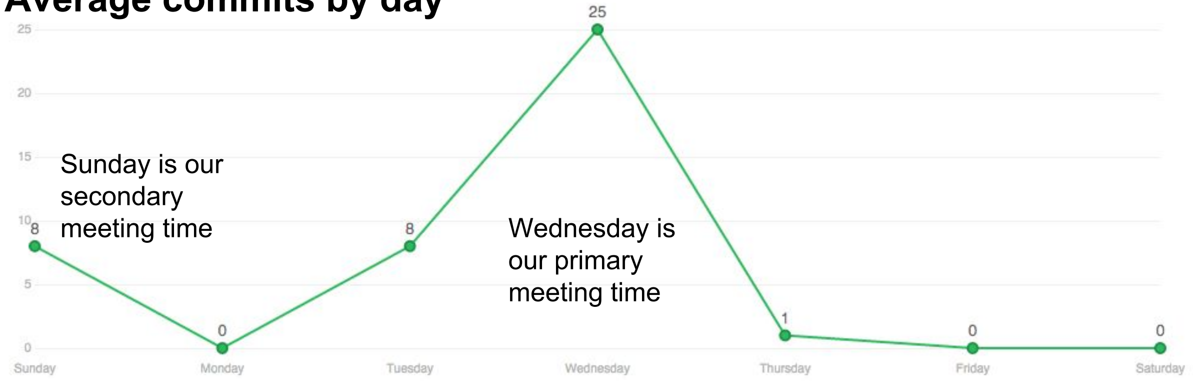
- Implement reference counting for lists
    - Tricky, because there are so many contexts in which the count should be adjusted
        - Functions that return lists
        - Anonymous lists: `int foo = @(bar()); // If bar returns a list...`
        - Moving towards the tail: `foo_list = ^foo_list`
        - Adding to the head: `foo_list = 1::foo_list`
    - Must all be done in a thread-safe context, since multiple threads share list tails
        - Threads compete for other global resources; easy to accidentally deadlock
            - Locks on channels
            - Locks on global list of thread metadata
    - Can be done, with more time and extreme care
- Implement operations on strings (along with reference counting)
- Separating compilation and linking

M

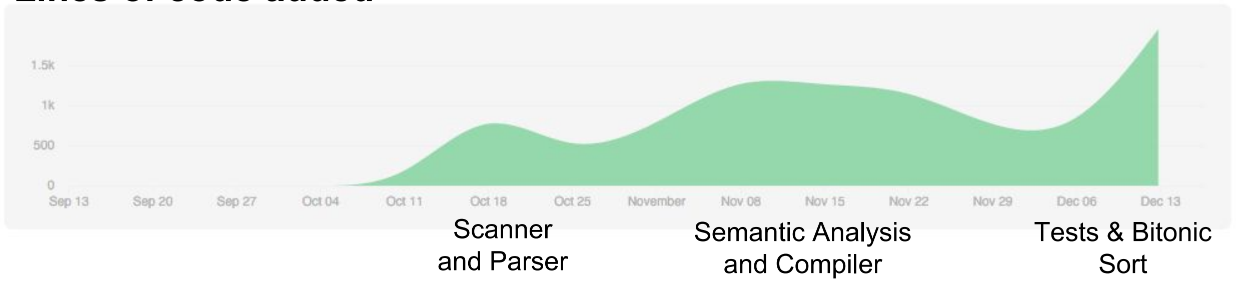
Stats

- 219 commits
- 54 unique cloners (?)

Average commits by day



Lines of code added



Component	**LOC**
tests (80 in total)	1671
semantic_analysis.ml	453
c_runtime.c	344
compile.ml	290
parser.mly	186
testall.sh	177
scanner.mll	71
Makefile	63
flowc.ml	35
Total	**3262**

A

Lessons Learned

Mitchell

Much of our time seemed at first to be unproductive: we spent 80% of it talking, planning, and brainstorming. 20% of our time together was spent on programming. But, I came to realize that communication is VERY important. Everyone needs to be on the same page. Discussion often exposes potential pitfalls, thereby lessening our chances of succumbing to them.

Hyonjee

Set up regular weekly meeting times (preferably more than one).

Discuss design and come up with a general implementation plan before writing code.

Set up a test framework early and write tests as you go -- with multiple people contributing code, this is the most effective way to make sure you don’t break things in the system. Tests can also give you goals and direction as you near the end of the project.

Adam

When in doubt, restart your computer
Draw and hand simulate algorithms you don’t understand
When programming in a different paradigm (object-oriented, functional, dataflow), it seems hard at first, but you just have to think about problems in an entirely different way.

Zach

Overly broad github issues such as “clean code” will never be closed - Writing tests that break the program is a much better way to prioritize things that need to be fixed.
If your team has a designated time to meet, make sure the group meets. Saying “let’s just work individually” can be interpreted as “I have other work I need to do.” If one or two group members cannot make a meeting, still meet, and do not cancel - doing a little each week keeps momentum going and is much more effective than pushing back work for another week.
Great teams make all the difference. These are people I would happily work with after college!