EECS 583 – Class 19
Research Topic 2
Exploiting Parallelism in Streaming Applications

University of Michigan

November 28, 2011

Announcements & Reading Material

- Exams – not graded yet, will be passed back next Monday
- Final projects
  - Each group will sign up for a 30 min presentation/demo slot
  - Presentation days: Dec 13-16, 19
- Today's class reading
  - "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," M. Gordon, W. Thies, and S. Amarasinghe, *Proc. of the 12th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.
- Next class reading
  - "Orchestrating the Execution of Stream Programs on Multicore Platforms," M. Kudlur and S. Mahlke, *Proc. ACM SIGPLAN 2008 Conference on Programming Languages Design and Implementation*, Jun. 2008, pp. 114-124.

Discussion Items

- What explicitly parallel programming models have you used?
  - What was good, what was bad about them?
- What are the advantages of stream programming?
- What are the limitations of stream programming?
- Are static rates sufficient? What changes with dynamic rates?
- Is stream programming the right approach?
  - Why isn't it more popular?
  - What is the better option?

Homework 2 Contest: Rules

- Correctness
  - Only eligible if your optimizer works on all correctness testcases
- Timing
  - Raw execution time: Average across 3 runs
  - Run on Core 2 Quad system (no other users)
- Winners (need to come to class to see)
  - Per performance benchmark
  - Overall winner (Geometric mean of speedups across performance benchmarks)
    - Note: Overall winner had to work on all performance benchmarks to be eligible

Congratz to Those Who Passed All the Correctness Testcases!!

- aabooth
- athuls
- basir
- benselb
- chardson
- dadick
- ddevec
- dpopoff
- durgesh
- egnorka
- hsinhao
- jasonjk
- jdkasten
- jiehou
- jlafonta
- joemp
- joshlzh
- kuper
- mspivak
- mviscomi
- nehaag
- ricardoj
- sanae
- shrupad
- wlthoma

Winners

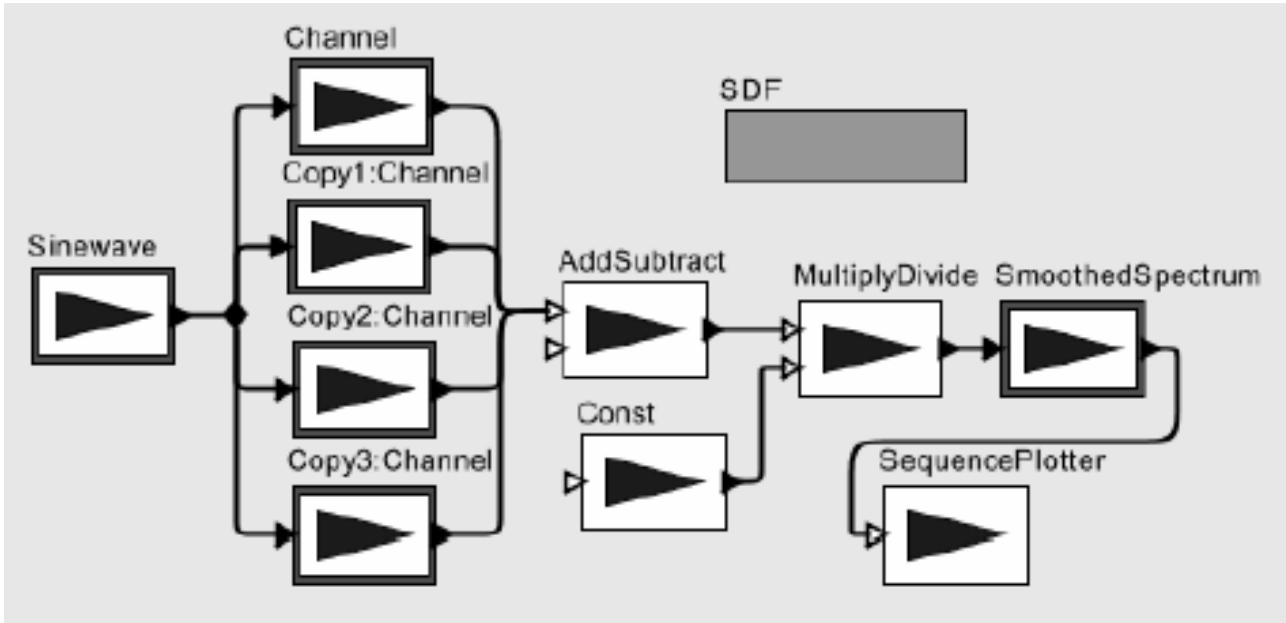
- Gotta come to class to see the winners!

StreamIt: A Language for Streaming Applications [Thies 02]

Streaming Application Domain

- Based on streams of data
- Increasingly prevalent and important
  - Embedded systems
    - Cell phones, handheld computers, DSP's
  - Desktop applications
    - Streaming media
    - Software radio
    - Real-time encryption
    - Graphics packages
  - High-performance servers
    - Software routers
    - Cell phone base stations
    - HDTV editing consoles

Synchronous Dataflow (SDF)



- Application is a graph of nodes
- Nodes send/receive items over channels
- Nodes have static I/O rates

  ➡ Can construct a static schedule

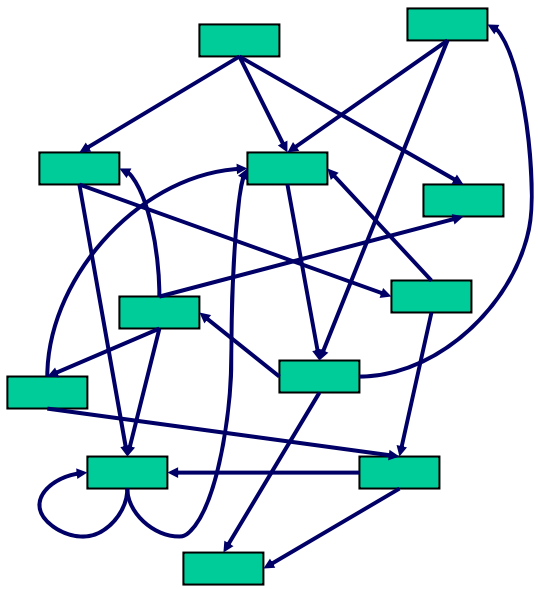
The StreamIt Language

- Also a synchronous dataflow language
  - With a few extra features
- Goals:
  - High performance
  - Improved programmer productivity
- Language Contributions:
  - Structured model of streams
  - Messaging system for control
  - Automatic program morphing

ENABLES Compiler Analysis & Optimization

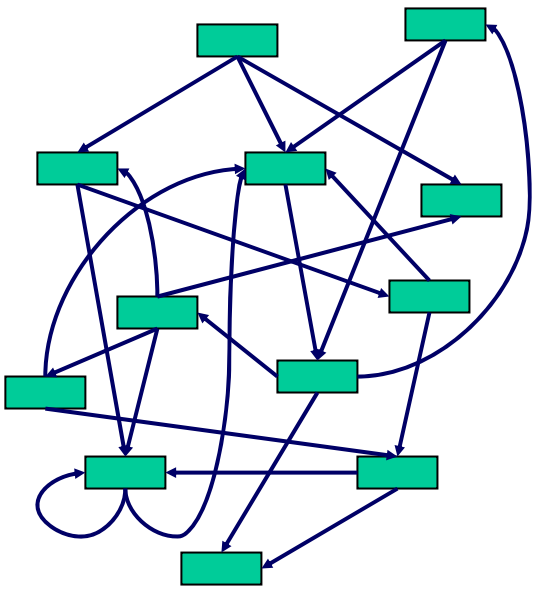
Representing Streams

- Conventional wisdom: streams are graphs
  - Graphs have no simple textual representation
  - Graphs are difficult to analyze and optimize

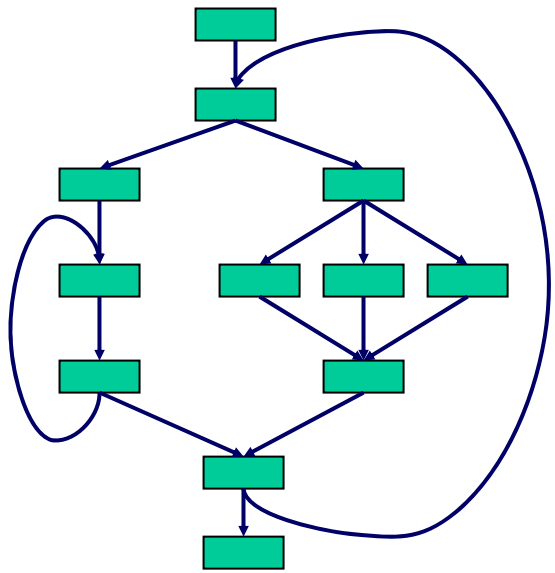


Representing Streams

- Conventional wisdom: streams are graphs
  - Graphs have no simple textual representation
  - Graphs are difficult to analyze and optimize
- Insight: stream programs have structure



unstructured

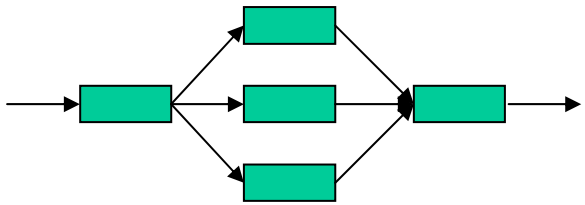


structured

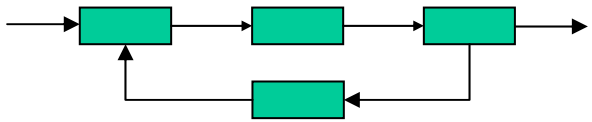
Structured Streams

- Hierarchical structures:
  - Pipeline

  - SplitJoin



  - Feedback Loop

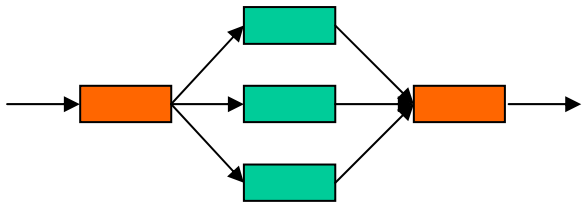


- Basic programmable unit: Filter

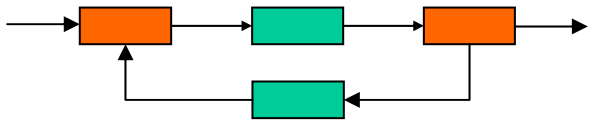
Structured Streams

- Hierarchical structures:
  - Pipeline

  - SplitJoin



  - Feedback Loop



- Basic programmable unit: Filter

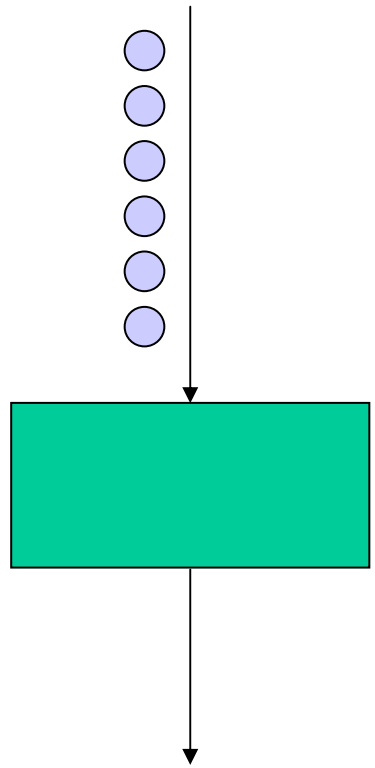
- Splits / Joins are compiler-defined

Representing Filters

- Autonomous unit of computation 
  - No access to global resources
  - Communicates through FIFO channels
    - pop() - peek(index) - push(value)
  - Peek / pop / push rates must be constant
- Looks like a Java class, with
  - An initialization function
  - A steady-state “work” function
  - Message handler functions

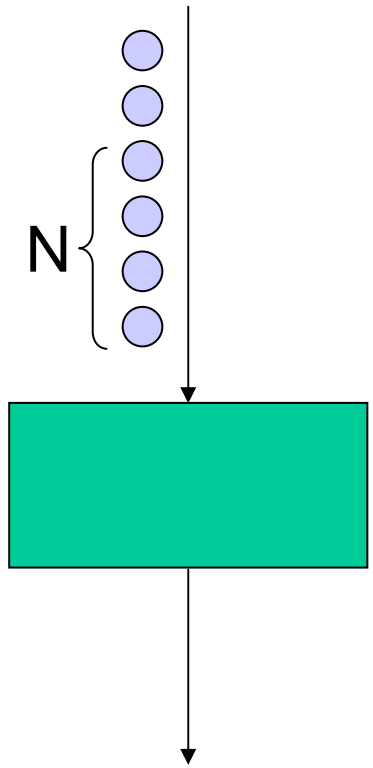
Filter Example: LowPassFilter

```
float->float filter LowPassFilter (float N) {
    float[N] weights;

    init {
        weights = calcWeights(N);
    }

    work push 1 pop 1 peek N {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
}
```



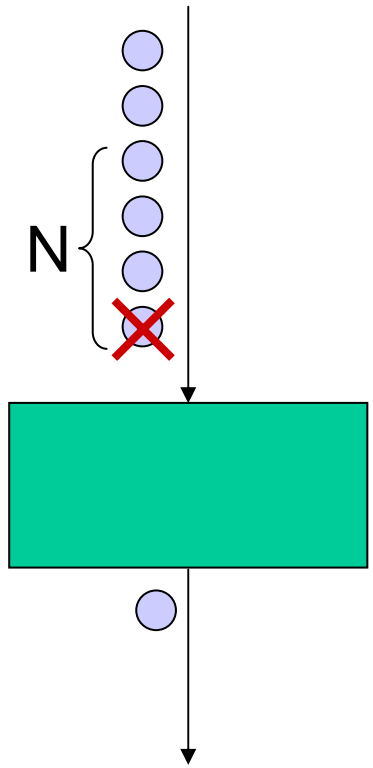
Filter Example: LowPassFilter

```
float->float filter LowPassFilter (float N) {
    float[N] weights;

    init {
        weights = calcWeights(N);
    }

    work push 1 pop 1 peek N {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
}
```



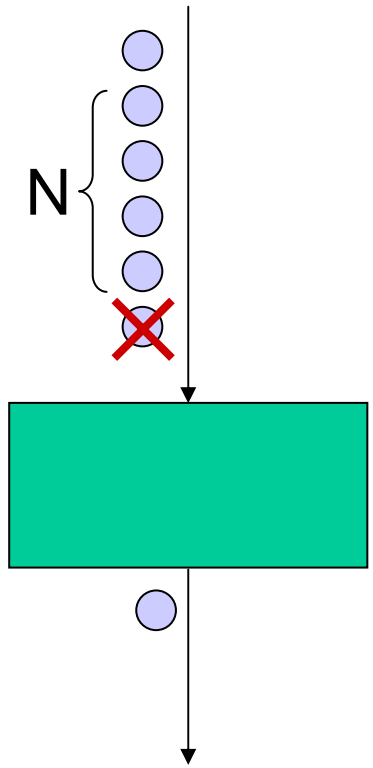
Filter Example: LowPassFilter

```
float->float filter LowPassFilter (float N) {
    float[N] weights;

    init {
        weights = calcWeights(N);
    }

    work push 1 pop 1 peek N {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
}
```



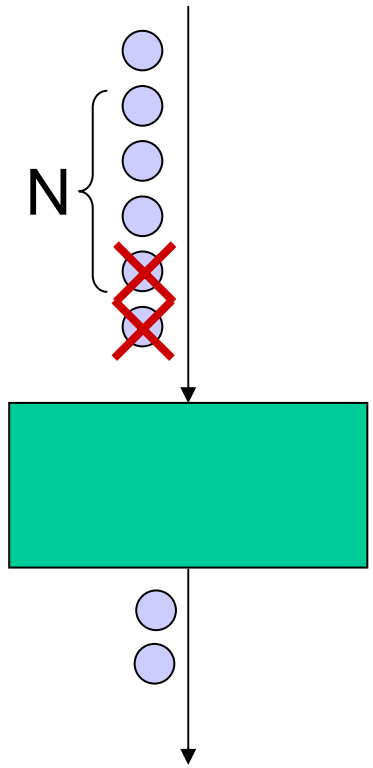
Filter Example: LowPassFilter

```
float->float filter LowPassFilter (float N) {
    float[N] weights;

    init {
        weights = calcWeights(N);
    }

    work push 1 pop 1 peek N {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
}
```



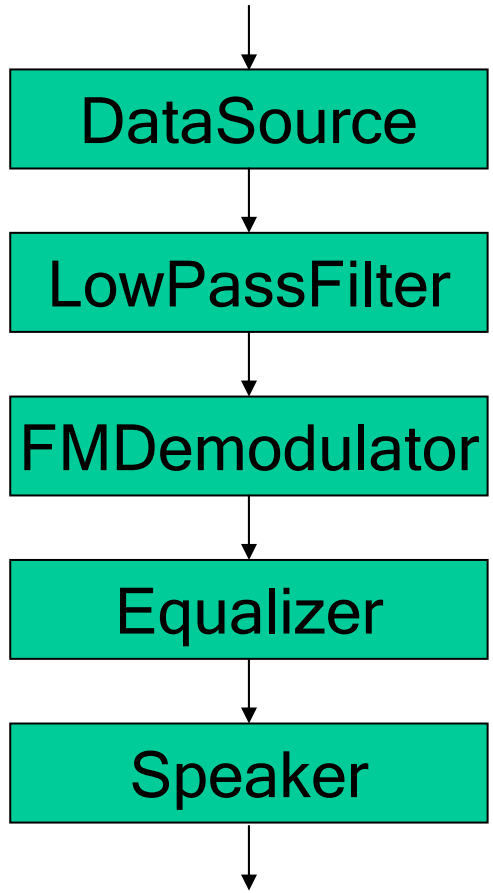
Filter Example: LowPassFilter

```
float->float filter LowPassFilter (float N) {
    float[N] weights;

    init {
        weights = calcWeights(N);
    }

    work push 1 pop 1 peek N {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
}
```



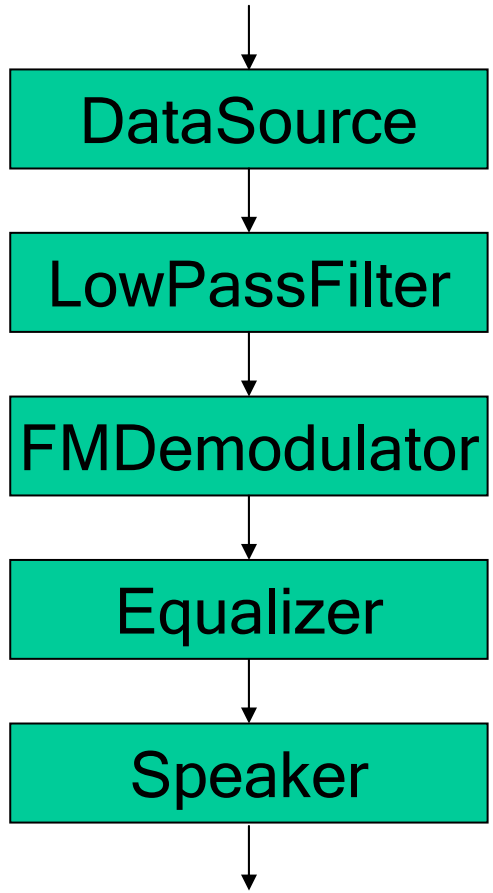
Pipeline Example: FM Radio

```
pipeline FMRadio {
    add DataSource();
    add LowPassFilter();
    add FMDemodulator();
    add Equalizer(8);
    add Speaker();
}
```



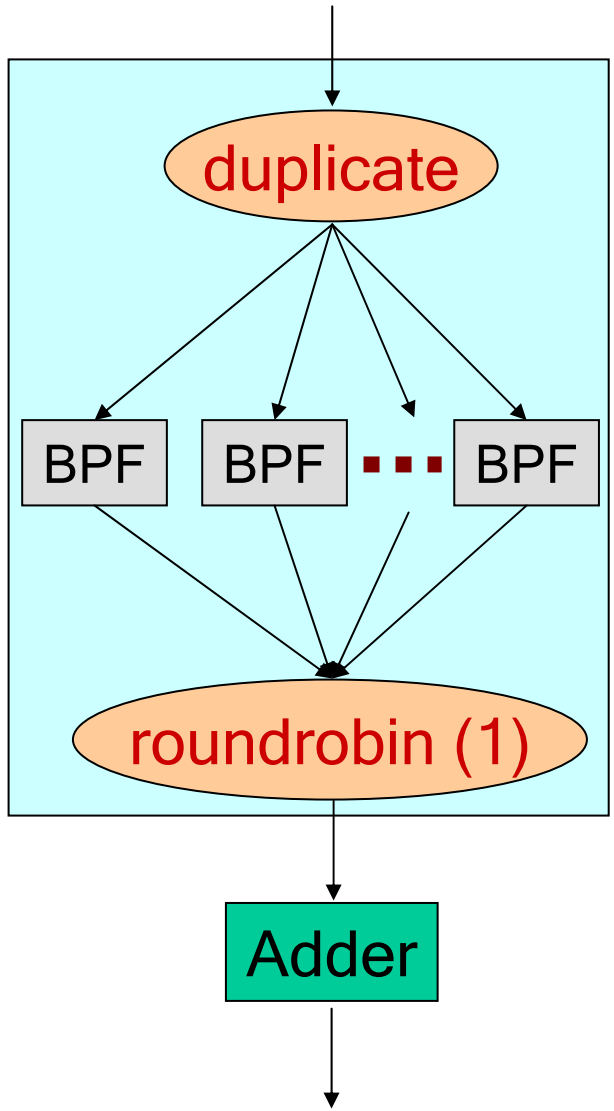
Pipeline Example: FM Radio

```
pipeline FMRadio {
    add DataSource();
    add LowPassFilter();
    add FMDemodulator();
    add Equalizer(8);
    add Speaker();
}
```



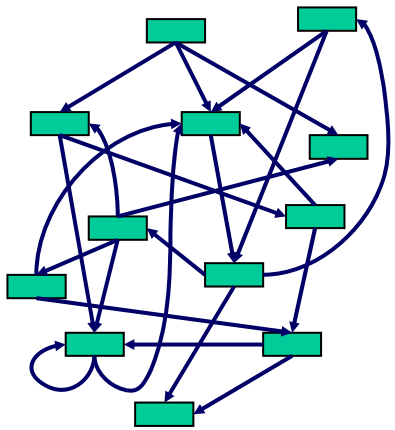
SplitJoin Example: Equalizer

```
pipeline Equalizer (int N) {
  add splitjoin {
    split duplicate;
    float freq = 10000;
    for (int i = 0; i < N; i ++, freq*=2) {
      add BandPassFilter(freq, 2*freq);
    }
    split roundrobin;
  }
  add Adder(N);
  }
}
```

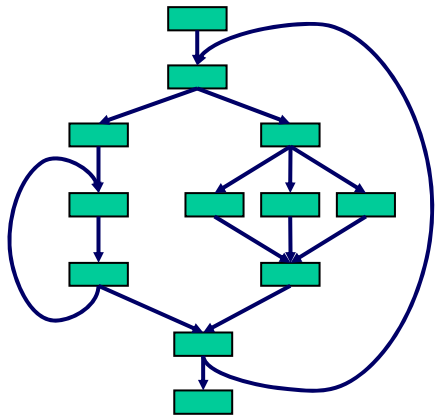


Why Structured Streams?

- Compare to structured control flow



GOTO statements



If / else / for statements

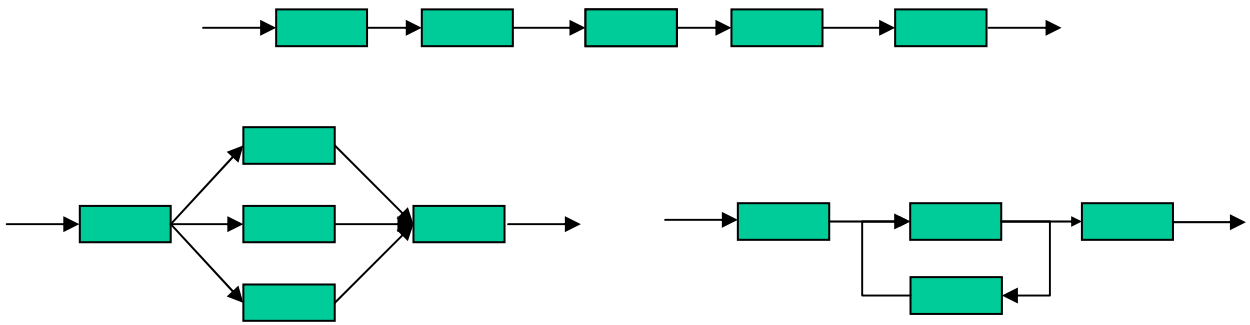
- Tradeoff:

PRO: - more robust - more analyzable

CON: - “restricted” style of programming

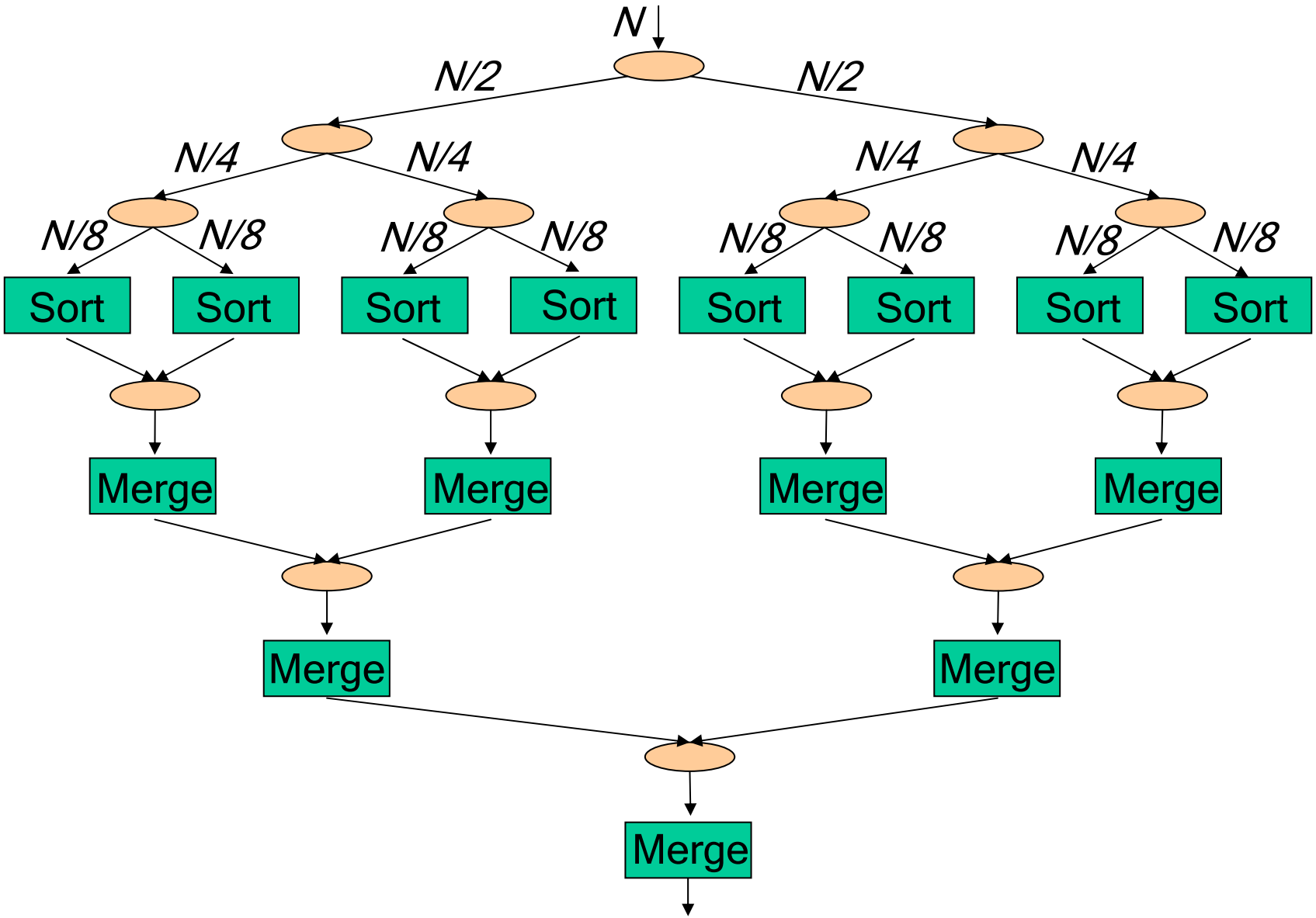
Structure Helps Programmers

- Modules are hierarchical and composable
  - Each structure is single-input, single-output



- Encapsulates common idioms
- Good textual representation
  - Enables parameterizable graphs

N-Element Merge Sort (3-level)



N-Element Merge Sort (K-level)

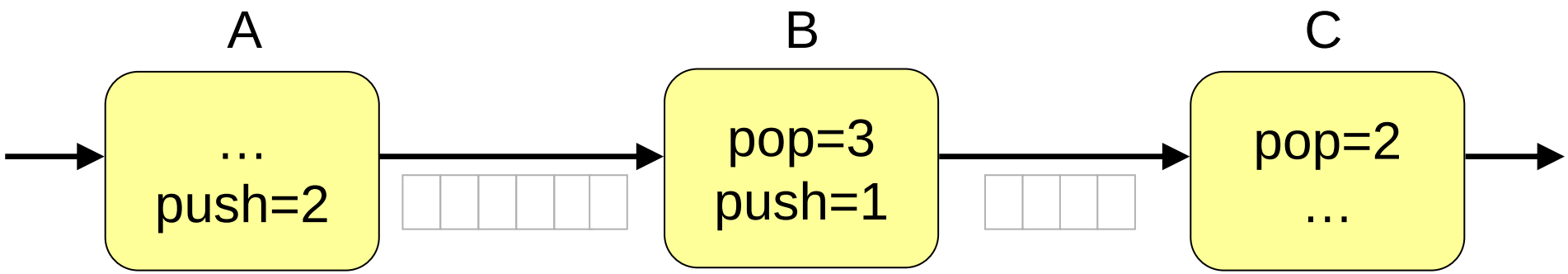
```
pipeline MergeSort (int N, int K) {
    if (K==1) {
        add Sort(N);
    } else {
        add splitjoin {
            split roundrobin;
            add MergeSort(N/2, K-1);
            add MergeSort(N/2, K-1);
            joiner roundrobin;
        }
    }
    add Merge(N);
  }
}
```

Basics of Stream Compilation [Gordon 06]

Rate Matching

- All data pop/push rates are constant
- Can find a Steady-State Invocation Count
  - # of items in the buffers are the same before and the after executing the sequence
  - There exist a unique minimum execution rate

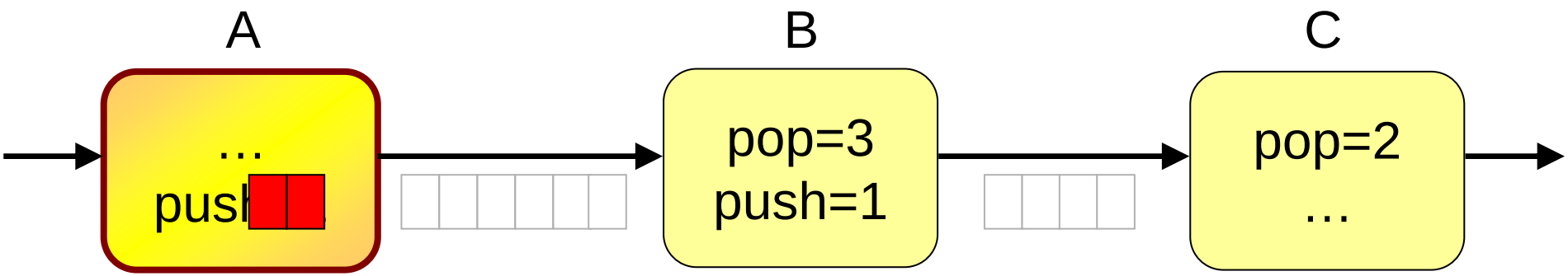
- Execution = { }



Rate Matching

- All data pop/push rates are constant
- Can find a Steady-State Invocation Count
  - # of items in the buffers are the same before and the after executing the sequence
  - There exist a unique minimum execution rate

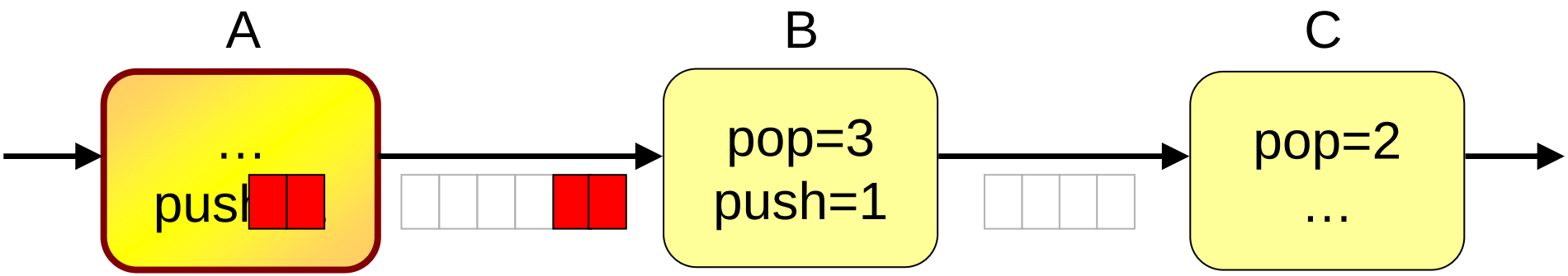
- Execution = { A }



Rate Matching

- All data pop/push rates are constant
- Can find a Steady-State Invocation Count
  - # of items in the buffers are the same before and the after executing the sequence
  - There exist a unique minimum execution rate

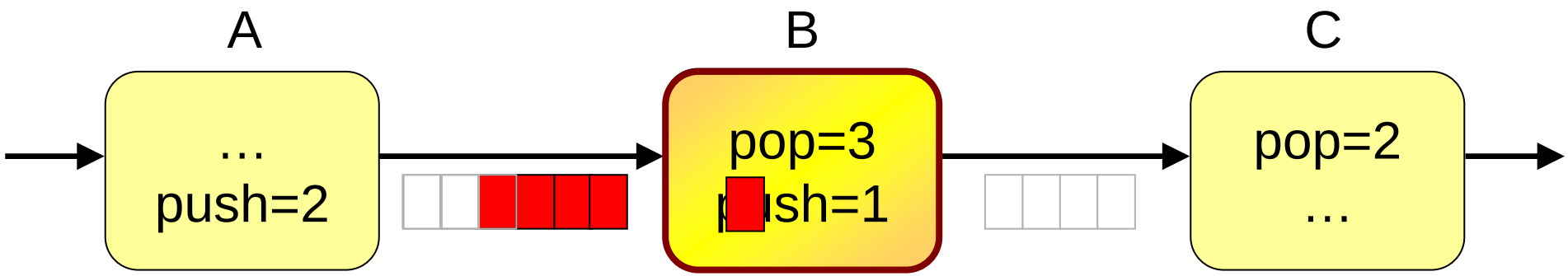
- Execution = { A, A }



Rate Matching

- All data pop/push rates are constant
- Can find a Steady-State Invocation Count
  - # of items in the buffers are the same before and the after executing the sequence
  - There exist a unique minimum execution rate

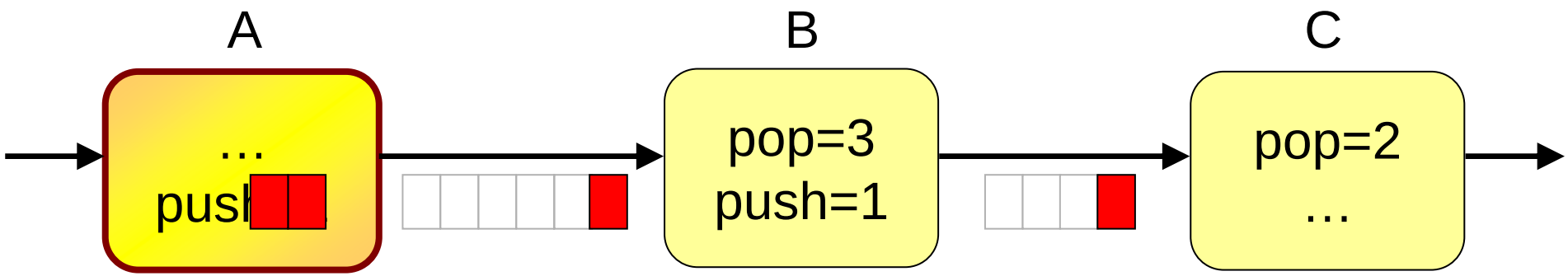
- Execution = { A, A, B }



Rate Matching

- All data pop/push rates are constant
- Can find a Steady-State Invocation Count
  - # of items in the buffers are the same before and the after executing the sequence
  - There exist a unique minimum execution rate

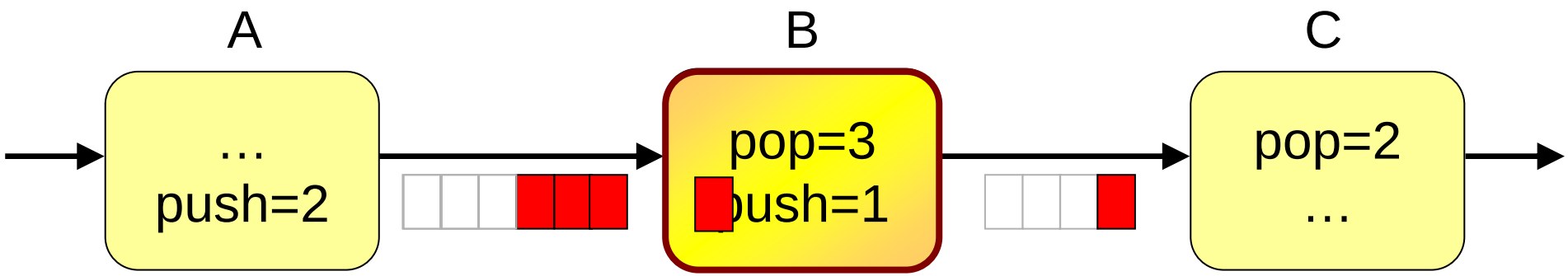
- Execution = { A, A, B, A }



Rate Matching

- All data pop/push rates are constant
- Can find a Steady-State Invocation Count
  - # of items in the buffers are the same before and the after executing the sequence
  - There exist a unique minimum execution rate

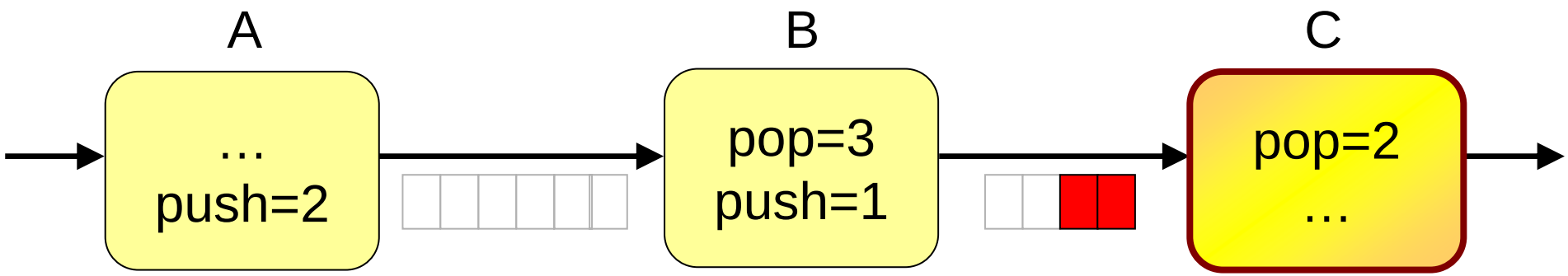
- Execution = { A, A, B, A, B }



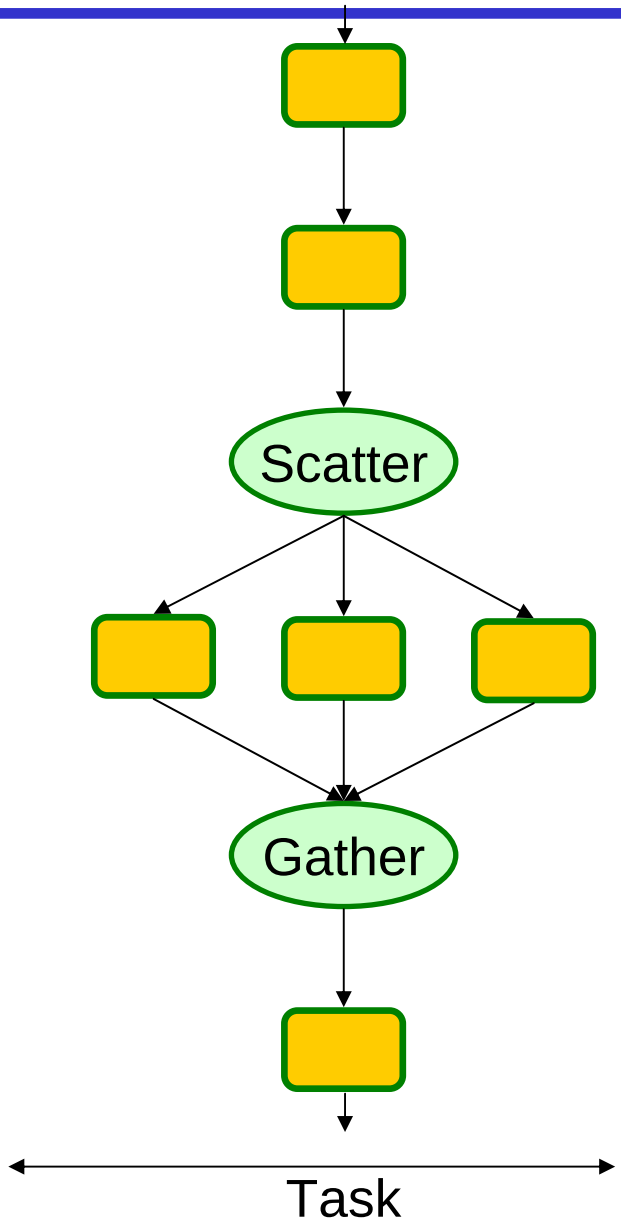
Rate Matching

- All data pop/push rates are constant
- Can find a Steady-State Invocation Count
  - # of items in the buffers are the same before and the after executing the sequence
  - There exist a unique minimum execution rate

- Execution = { A, A, B, A, B, C } -> 3A, 2B, C



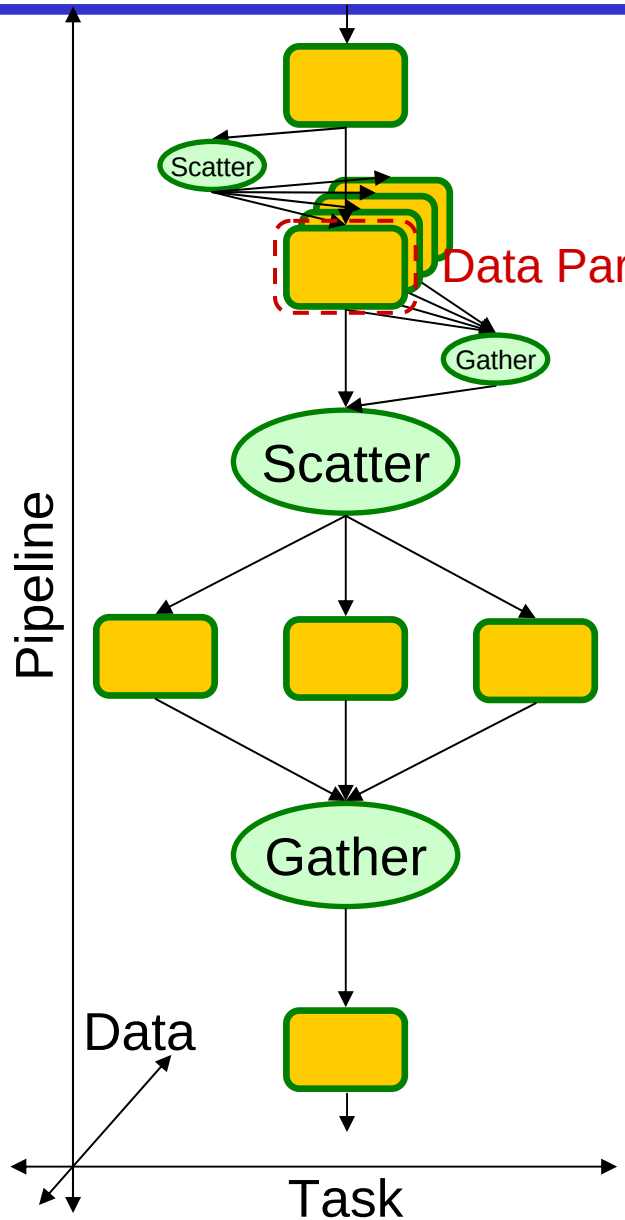
Types of Parallelism



Task Parallelism

- Parallelism explicit in algorithm
- Between filters *without* producer/consumer relationship

Types of Parallelism



Task Parallelism

- Parallelism explicit in algorithm
- Between filters *without* producer/consumer relationship

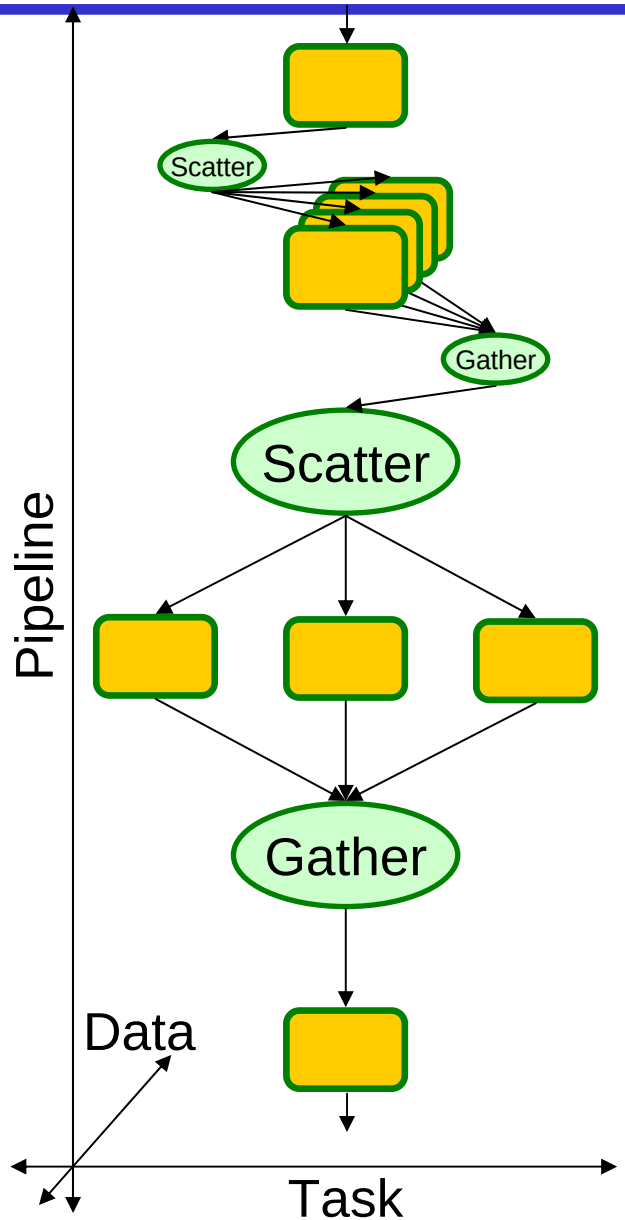
Data Parallelism

- Between iterations of a *stateless* filter
- Place within scatter/gather pair (*fission*)
- Can't parallelize filters with state

Pipeline Parallelism

- Between producers and consumers
- *Stateful* filters can be parallelized

Types of Parallelism



Traditionally:

Task Parallelism

- Thread (fork/join) parallelism

Data Parallelism

- Data parallel loop (**forall**)

Pipeline Parallelism

- Usually exploited in hardware

Problem Statement

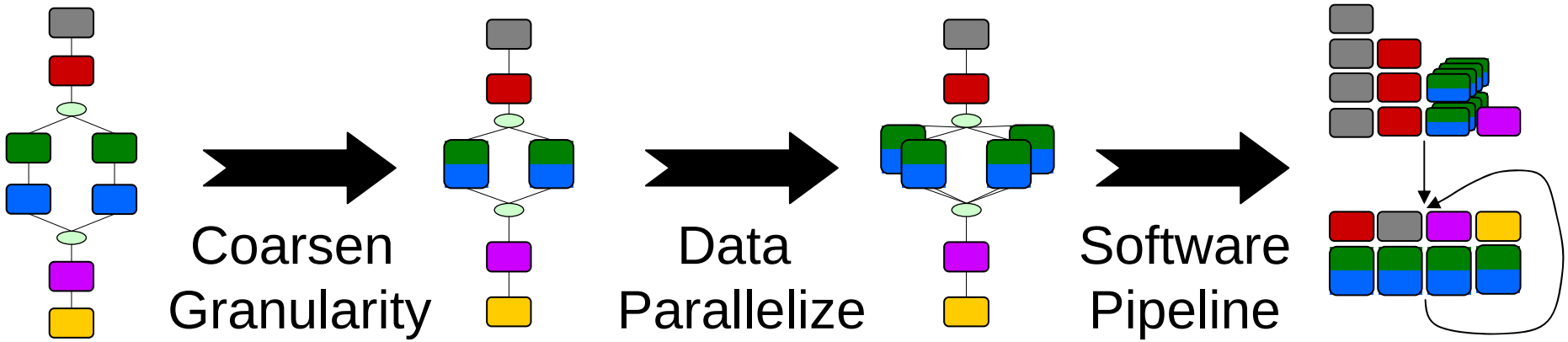
Given:

- Stream graph with compute and communication estimate for each filter
- Computation and communication resources of the target machine

Find:

- Schedule of execution for the filters that best utilizes the available parallelism to fit the machine resources

3-Phase Solution

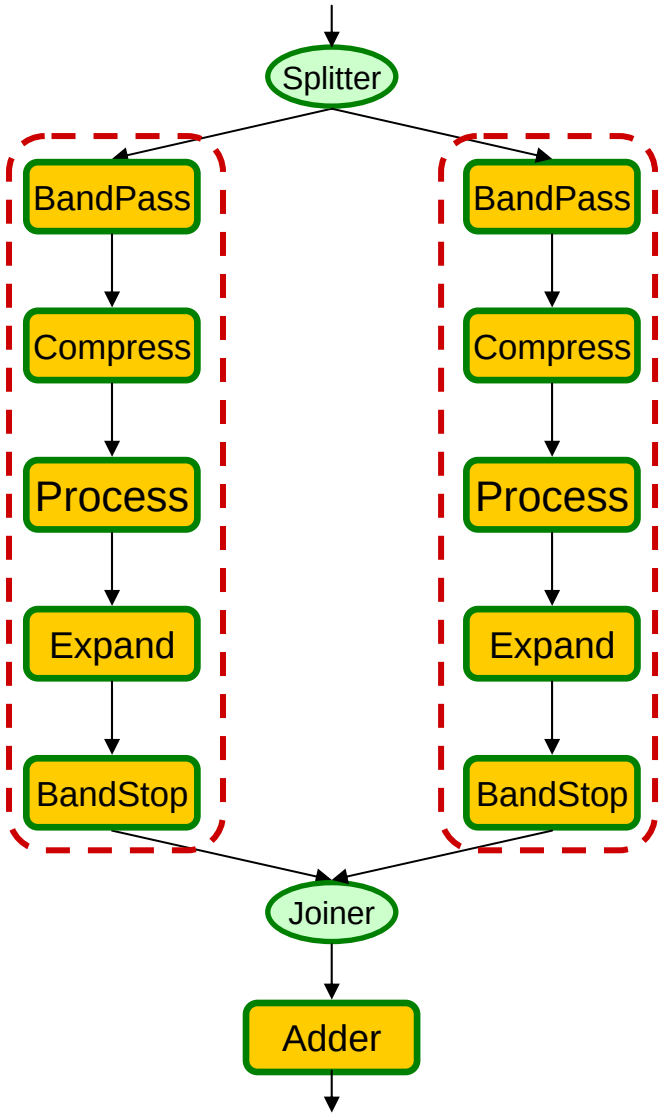


1. Coarsen: Fuse stateless sections of the graph
2. Data Parallelize: parallelize stateless filters
3. Software Pipeline: parallelize stateful filters

Compile to a 16 core architecture

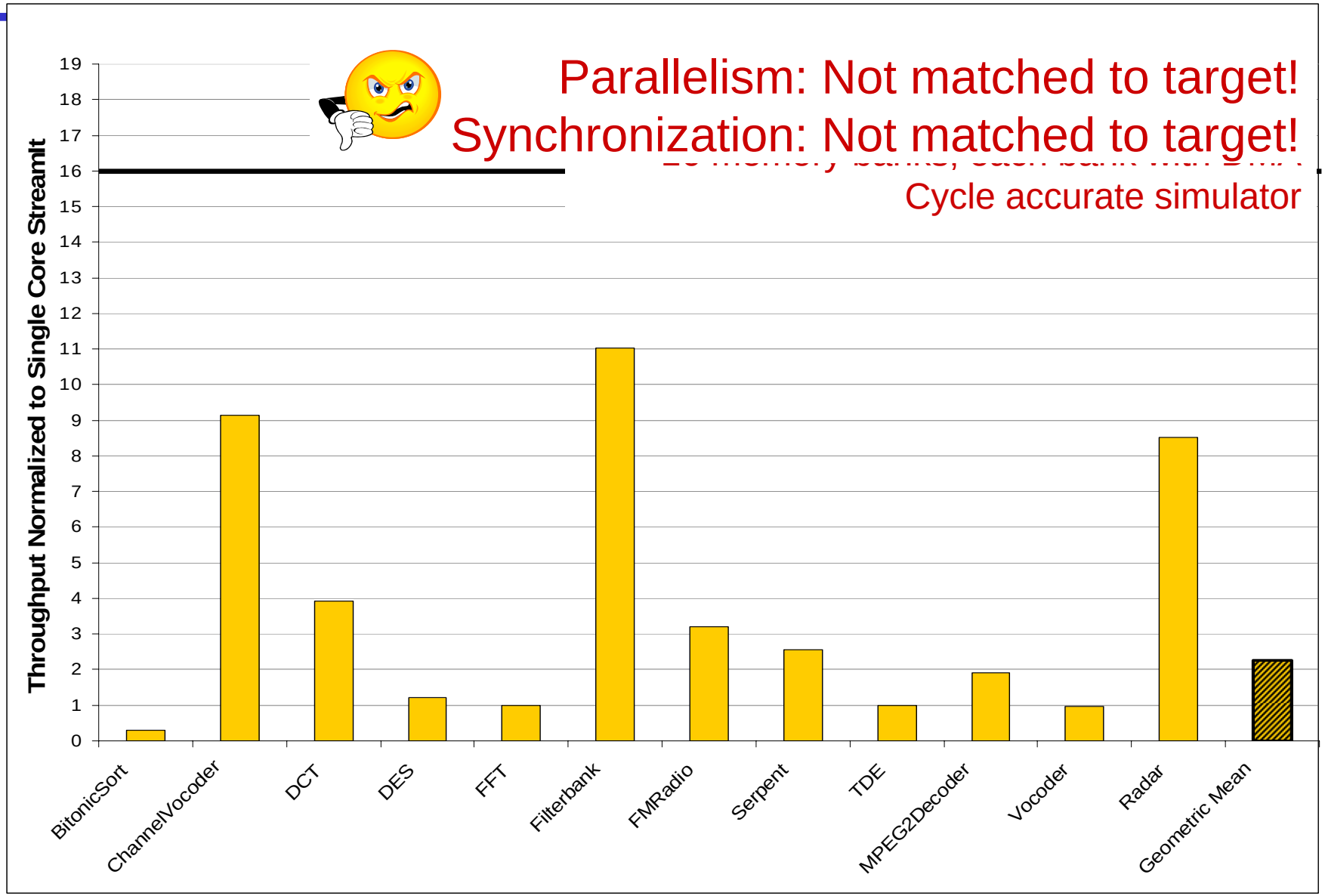
- 11.2x mean throughput speedup over single core

Baseline 1: Task Parallelism

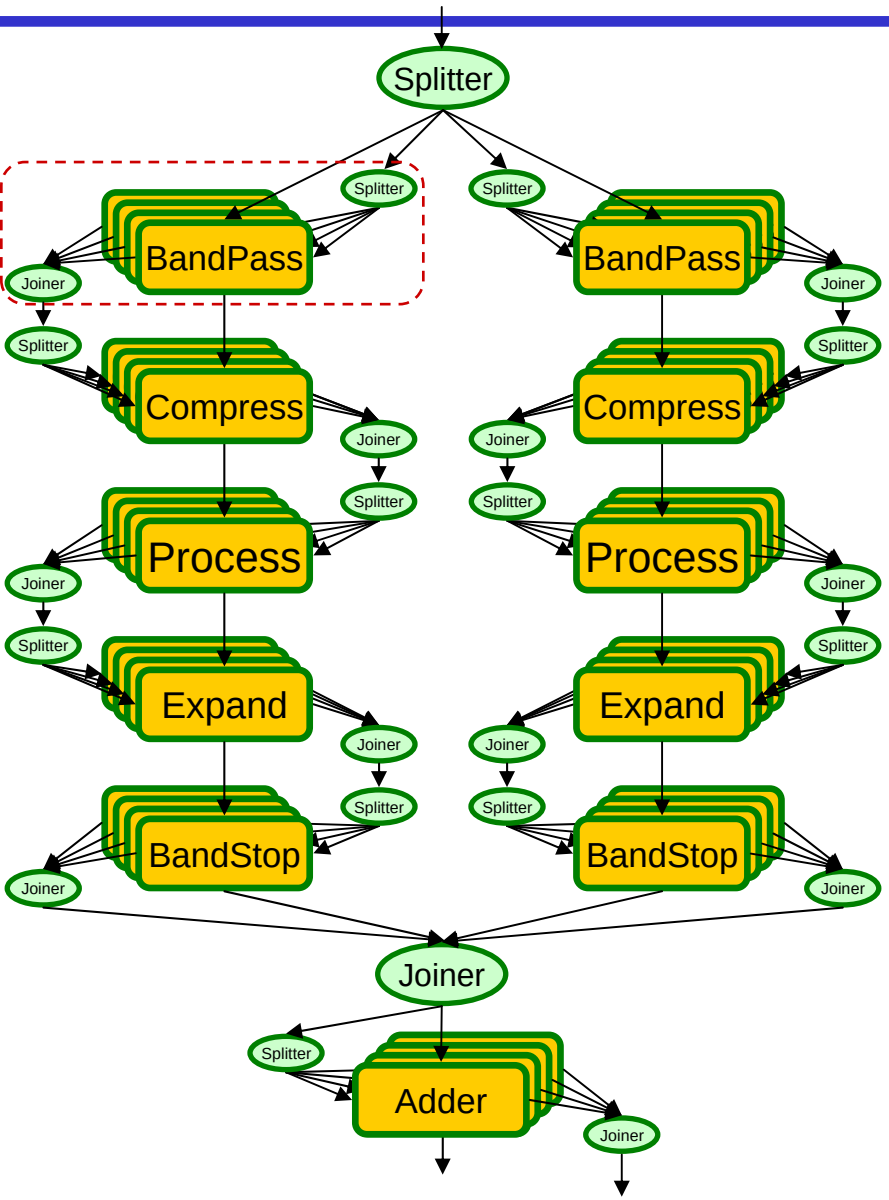


- Inherent task parallelism between two processing pipelines
- Task Parallel Model:
  - Only parallelize explicit task parallelism
  - Fork/join parallelism
- Execute this on a 2 core machine ~2x speedup over single core
- What about 4, 16, 1024, … cores?

Evaluation: Task Parallelism

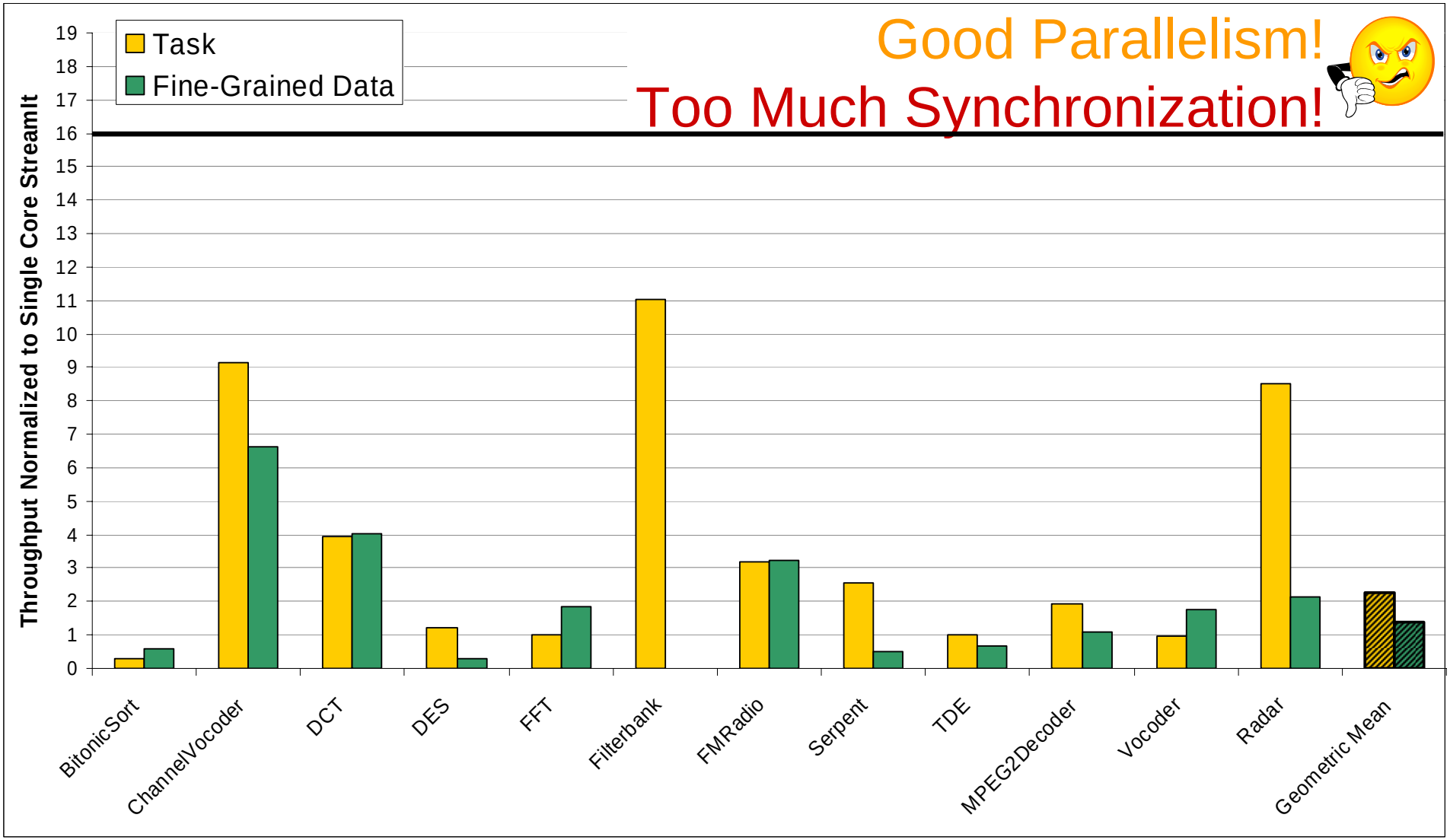


Baseline 2: Fine-Grained Data Parallelism

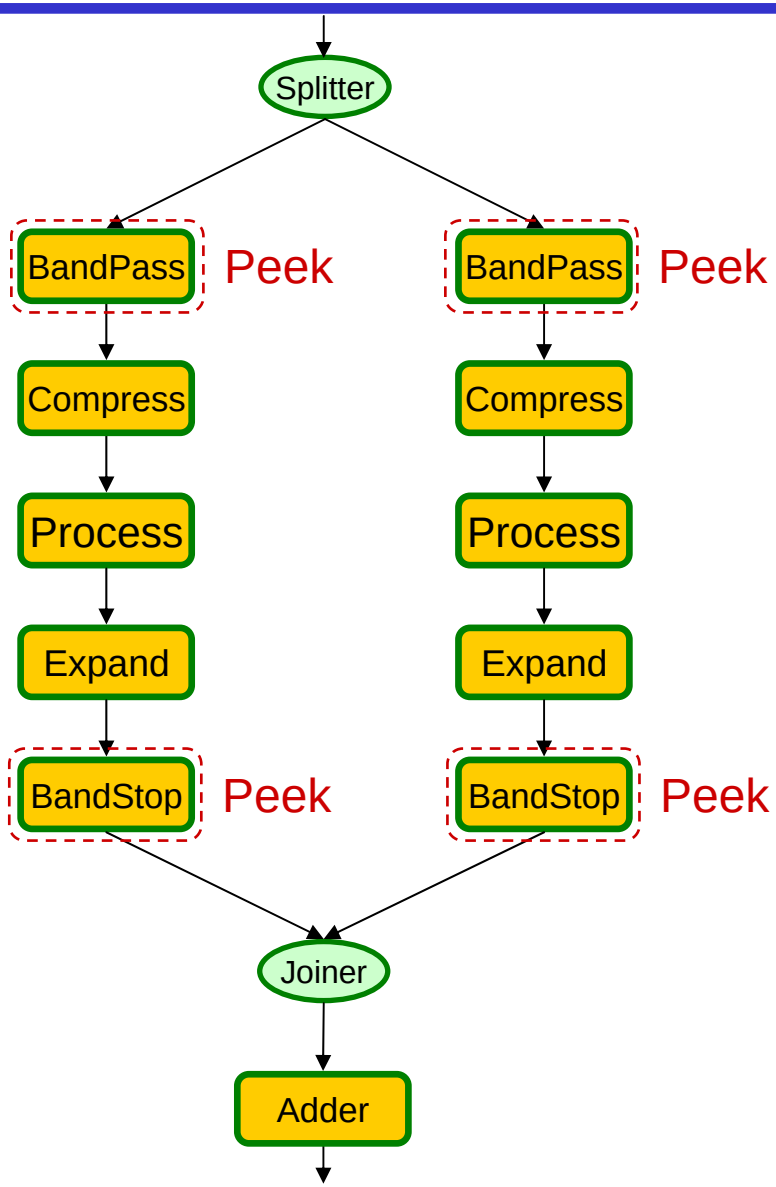


- Each of the filters in the example are stateless
- Fine-grained Data Parallel Model:
  - *Fiss* each stateless filter *N* ways (*N* is number of cores)
  - Remove scatter/gather if possible
- We can introduce data parallelism
  - Example: 4 cores
- Each fission group occupies entire machine

Evaluation: Fine-Grained Data Parallelism

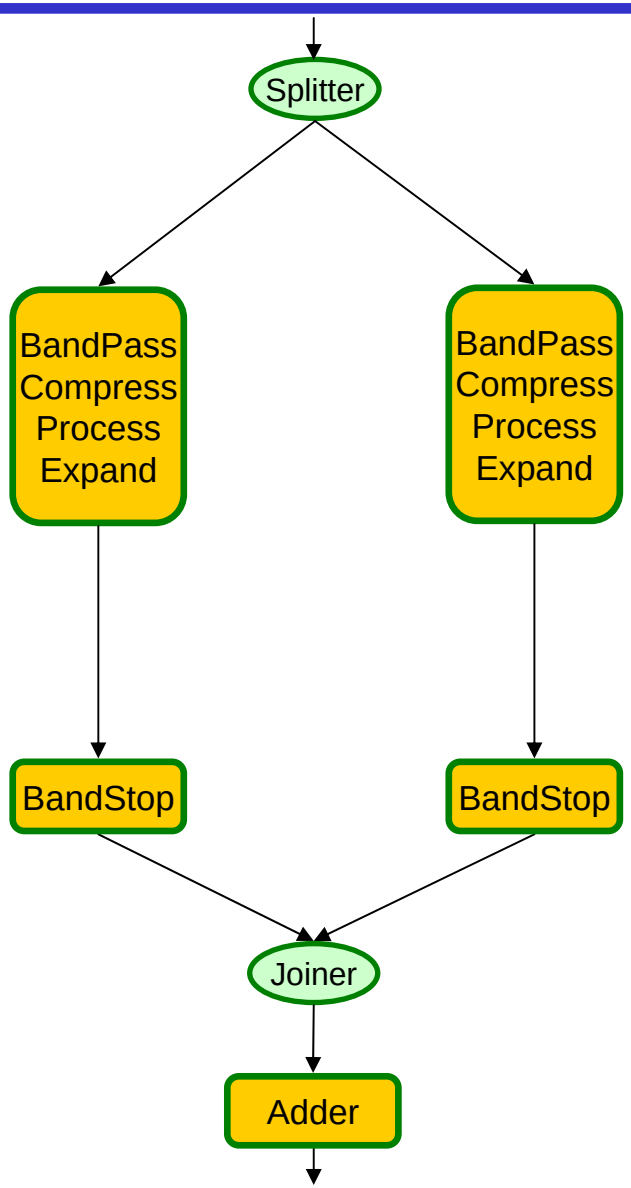


Phase 1: Coarsen the Stream Graph



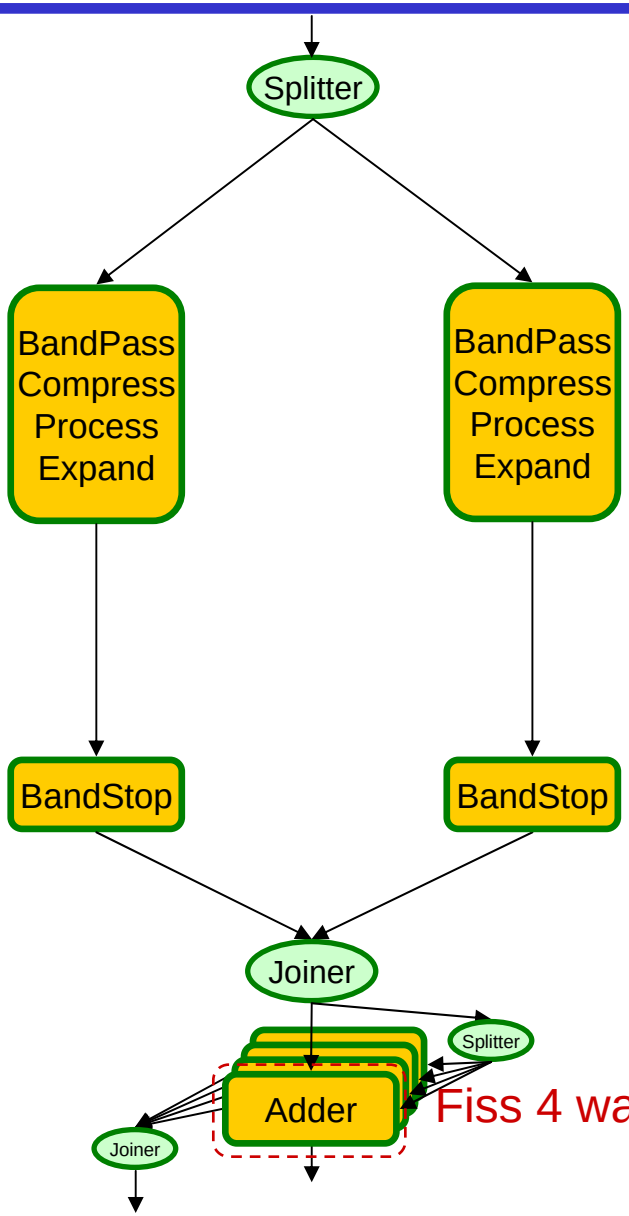
- Before data-parallelism is exploited
- *Fuse* stateless pipelines as much as possible without introducing state
  - Don't fuse stateless with stateful
  - Don't fuse a peeking filter with anything upstream

Phase 1: Coarsen the Stream Graph



- Before data-parallelism is exploited
- *Fuse* stateless pipelines as much as possible without introducing state
  - Don't fuse stateless with stateful
  - Don't fuse a peeking filter with anything upstream
- Benefits:
  - Reduces global communication and synchronization
  - Exposes inter-node optimization opportunities

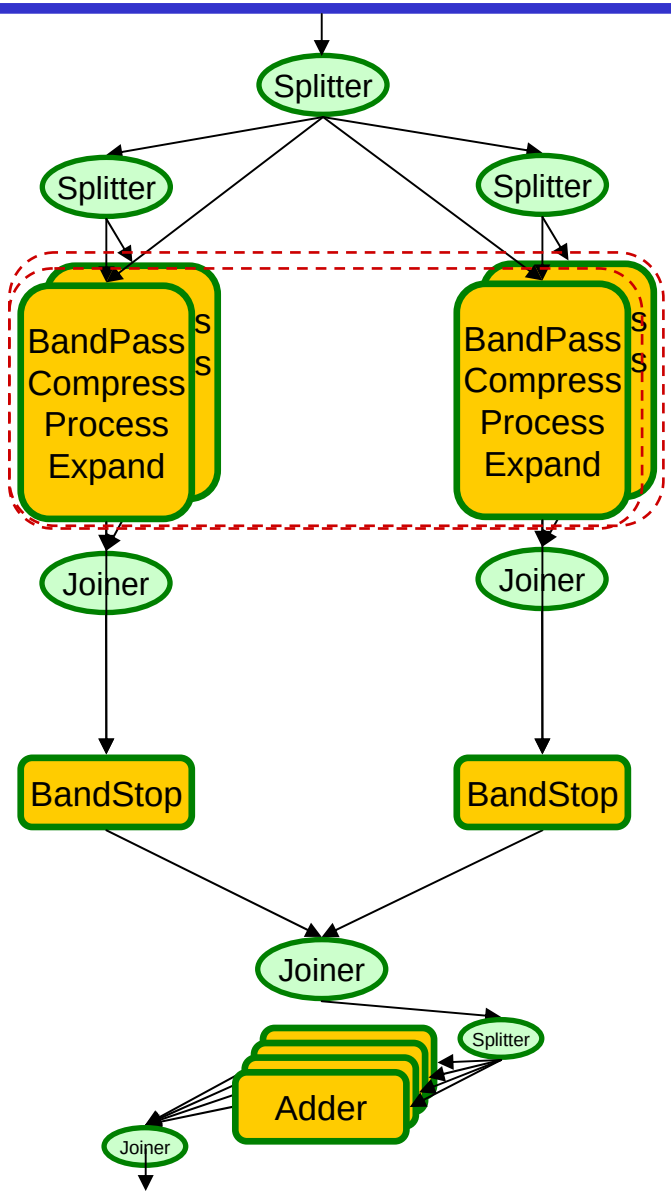
Phase 2: Data Parallelize



Data Parallelize for 4 cores

Fiss 4 ways, to occupy entire chip

Phase 2: Data Parallelize



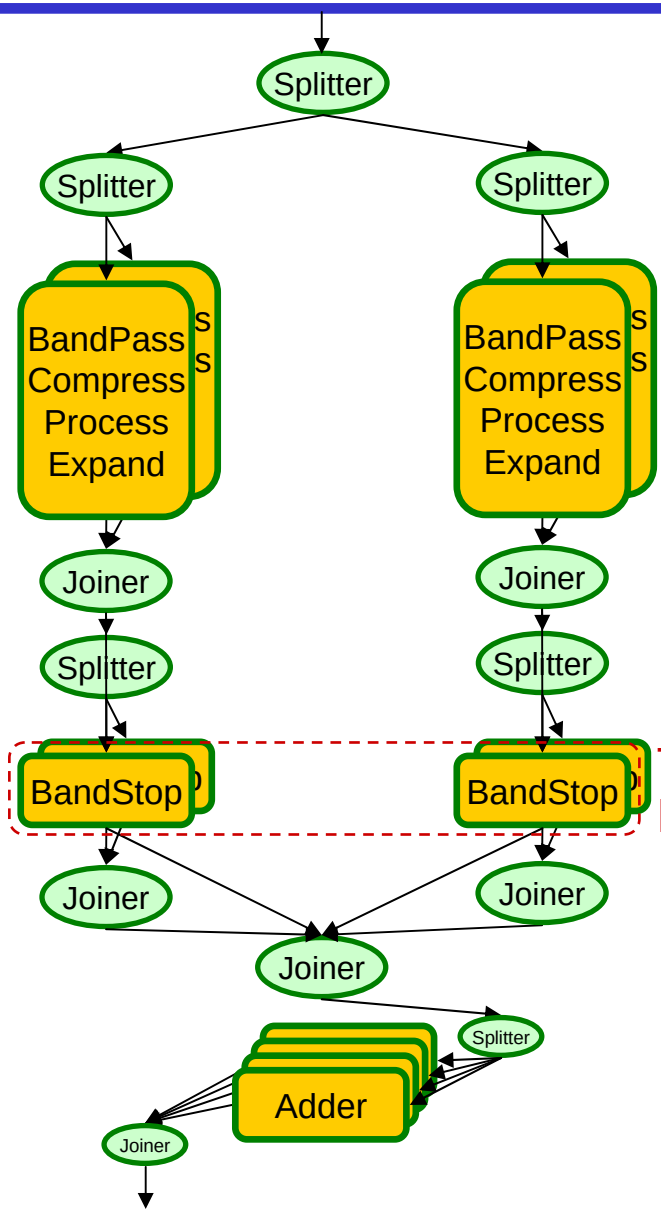
Data Parallelize for 4 cores

Task parallelism!

Each fused filter does equal work

Fiss each filter 2 times to occupy entire chip

Phase 2: Data Parallelize

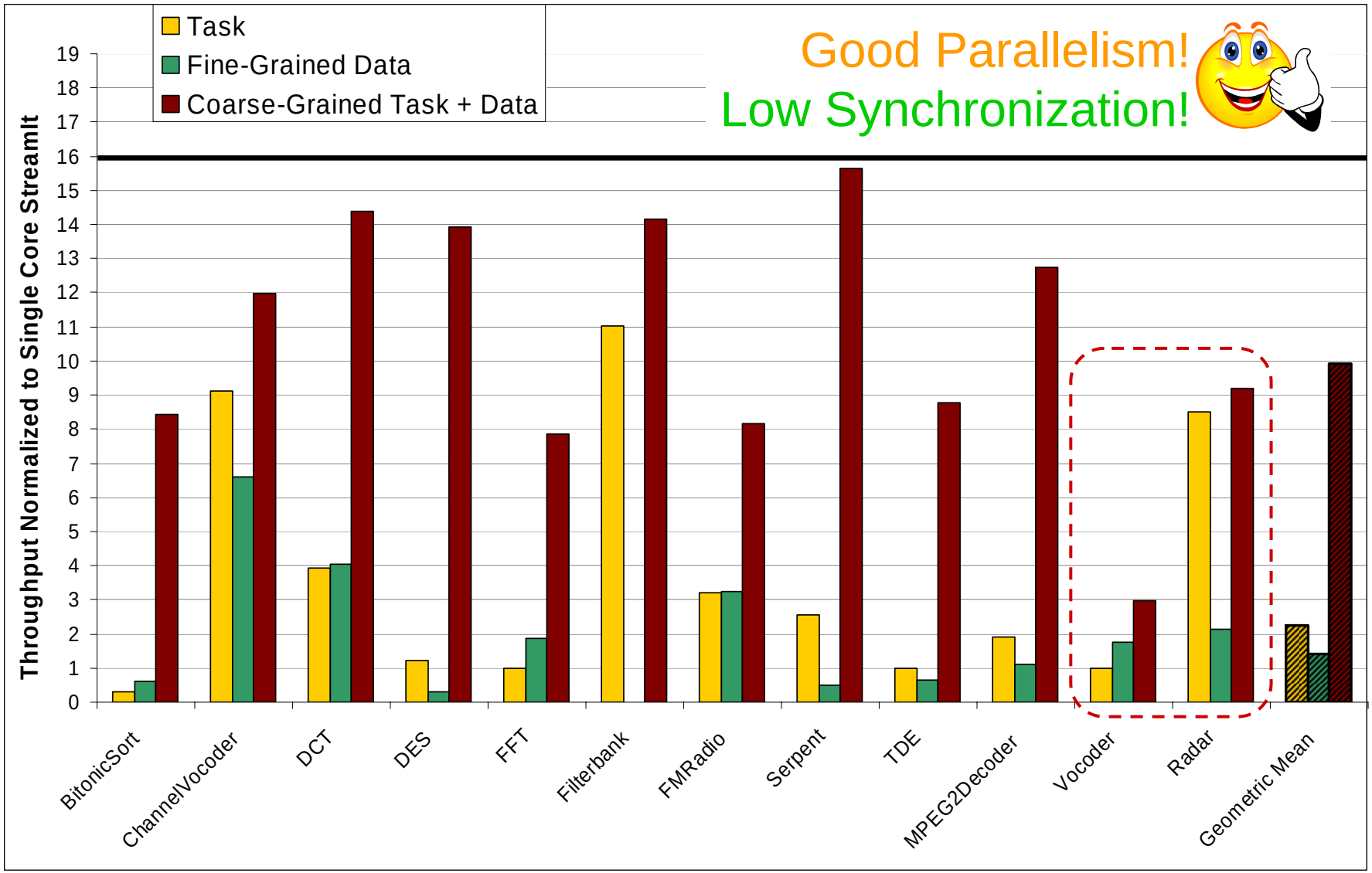


Data Parallelize for 4 cores

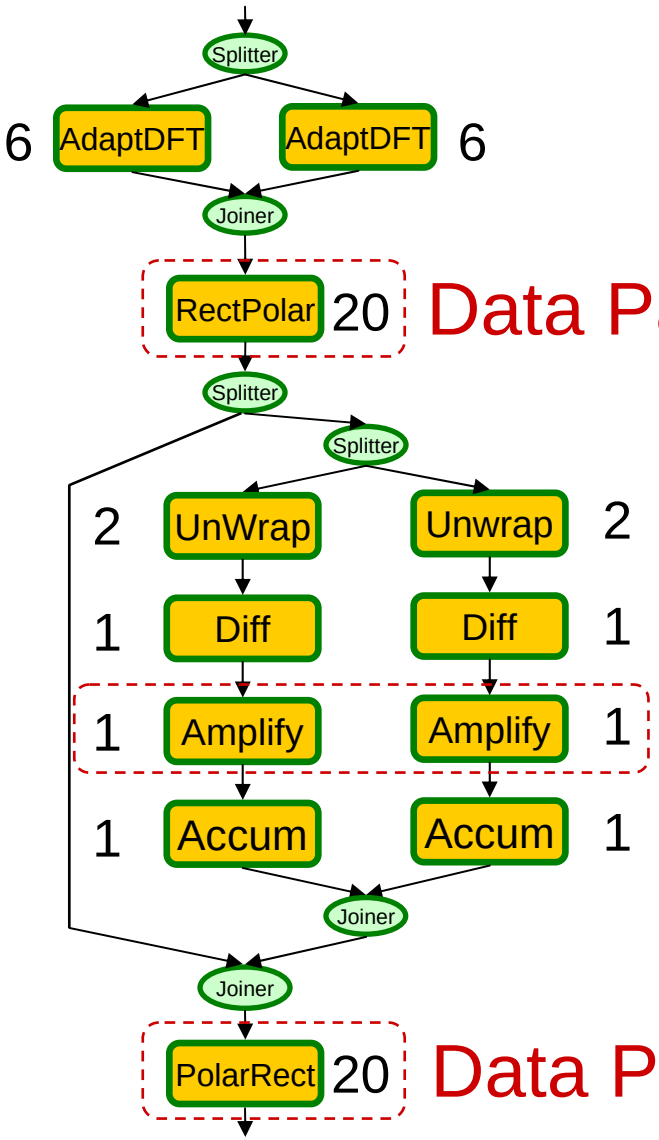
- Task-conscious data parallelization
  - Preserve task parallelism
- Benefits:
  - Reduces global communication and synchronization

Task parallelism, each filter does equal work
Fiss each filter 2 times to occupy entire chip

Evaluation: Coarse-Grained Data Parallelism



Simplified Vocoder



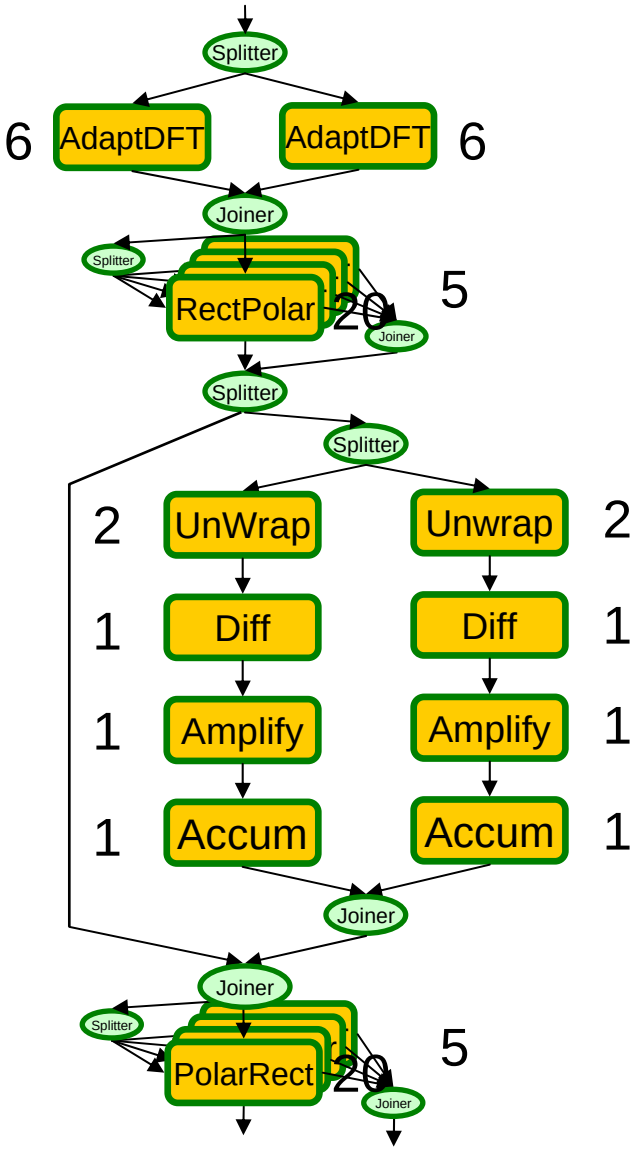
Data Parallel

Data Parallel, but too little work!

Data Parallel

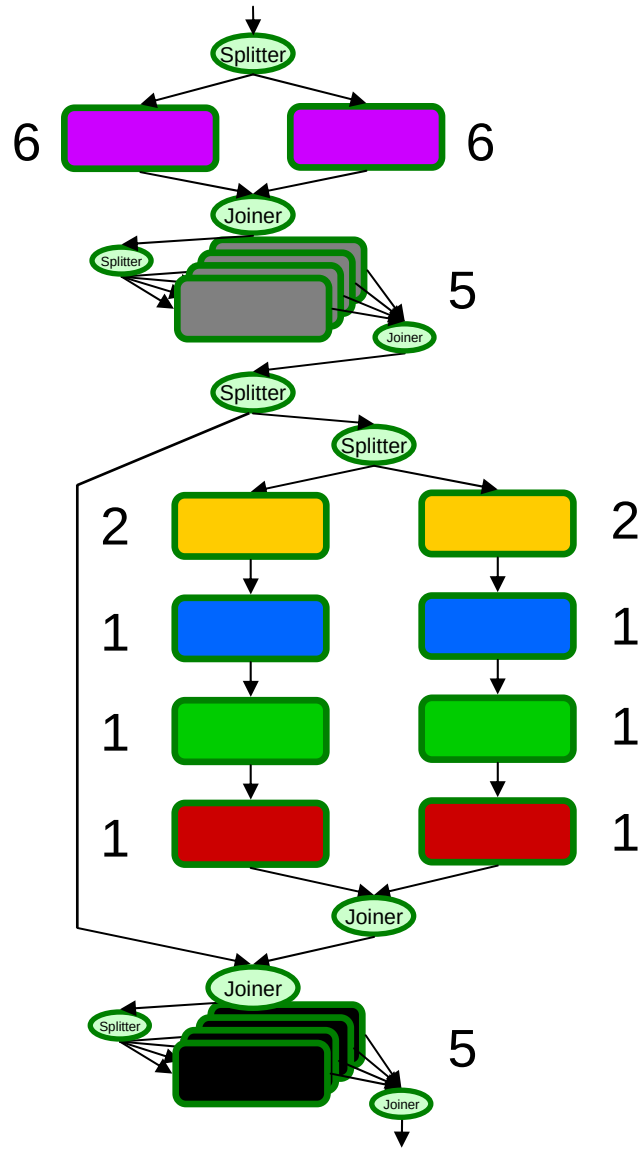
Target a 4 core machine

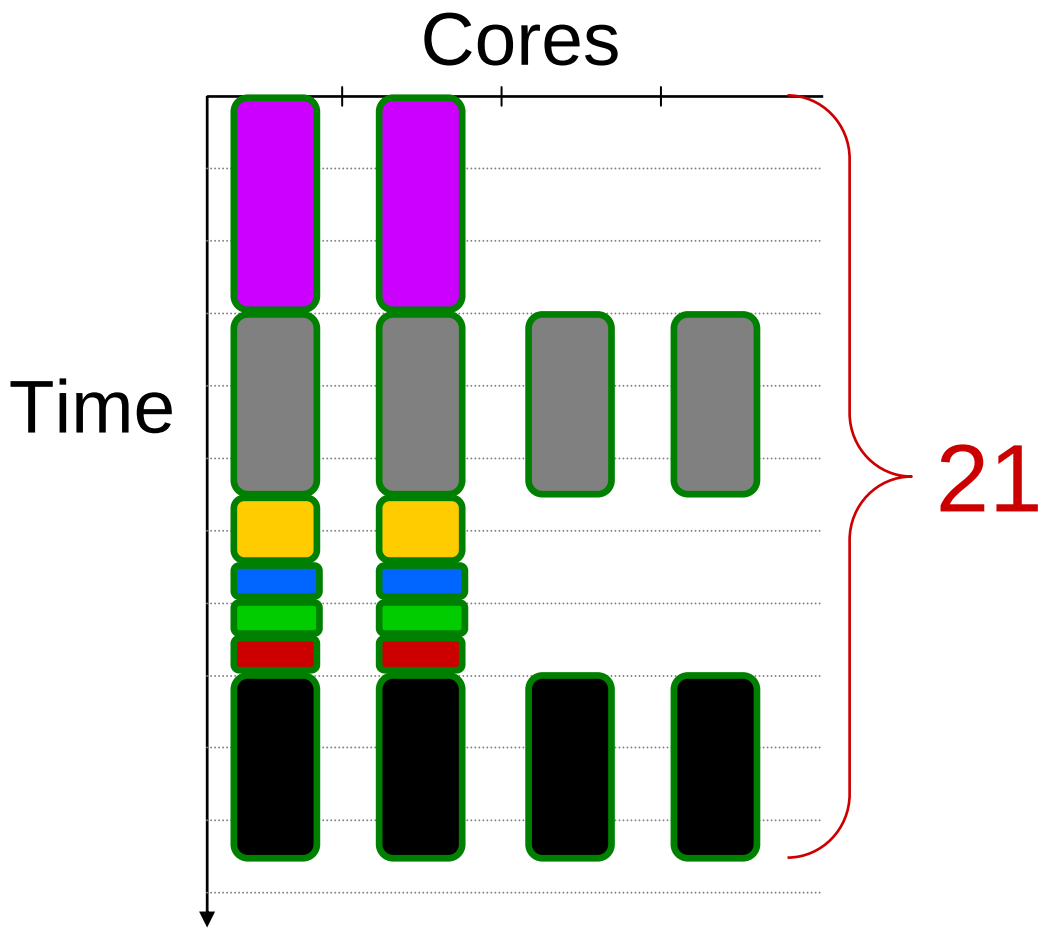
Data Parallelize



Target a 4 core machine

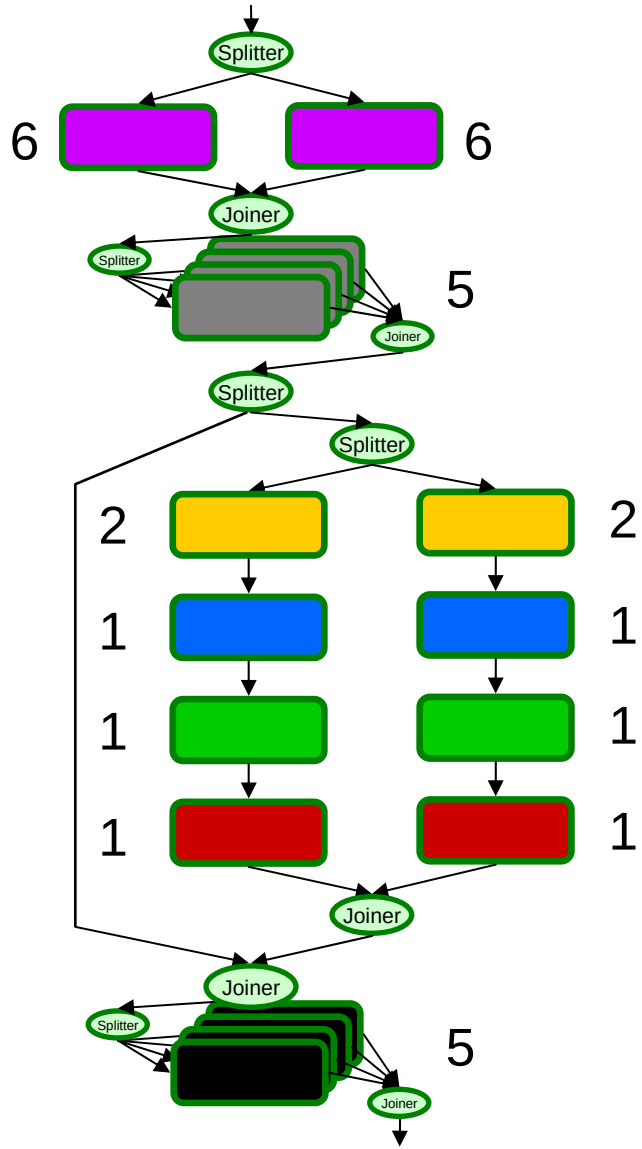
Data + Task Parallel Execution

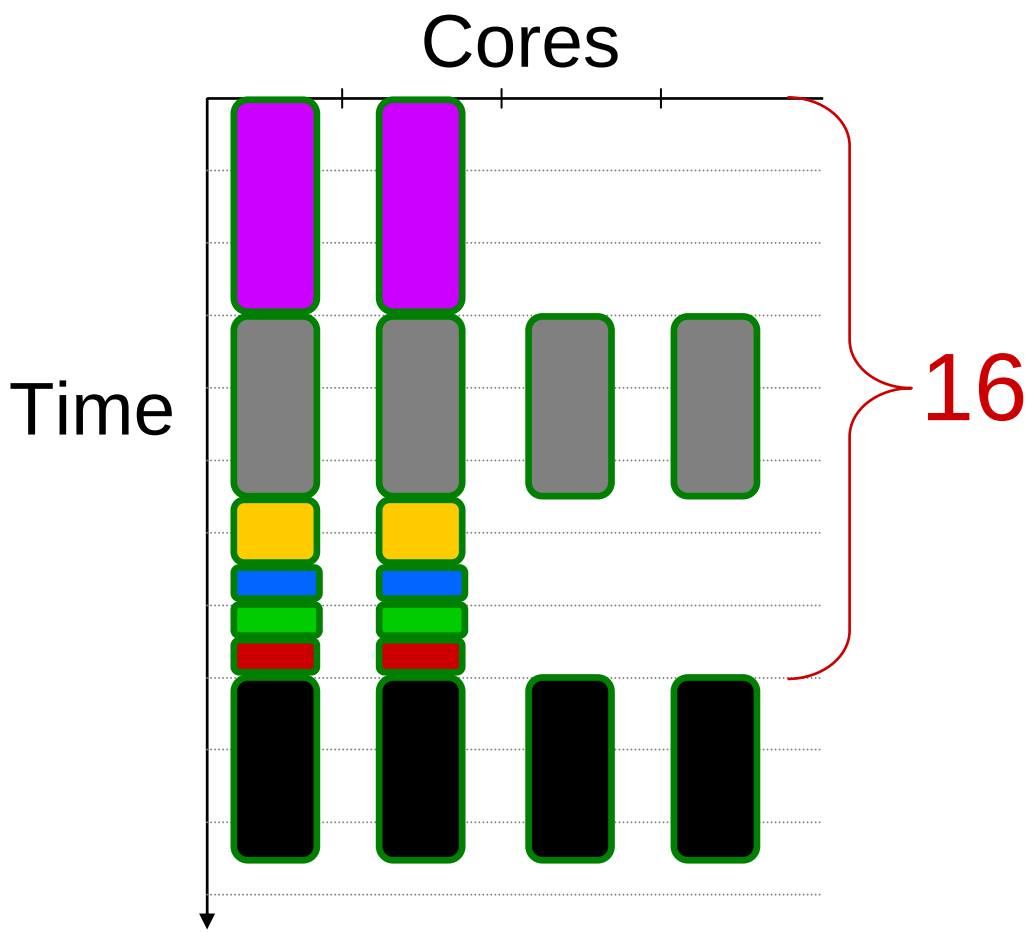




Target 4 core machine

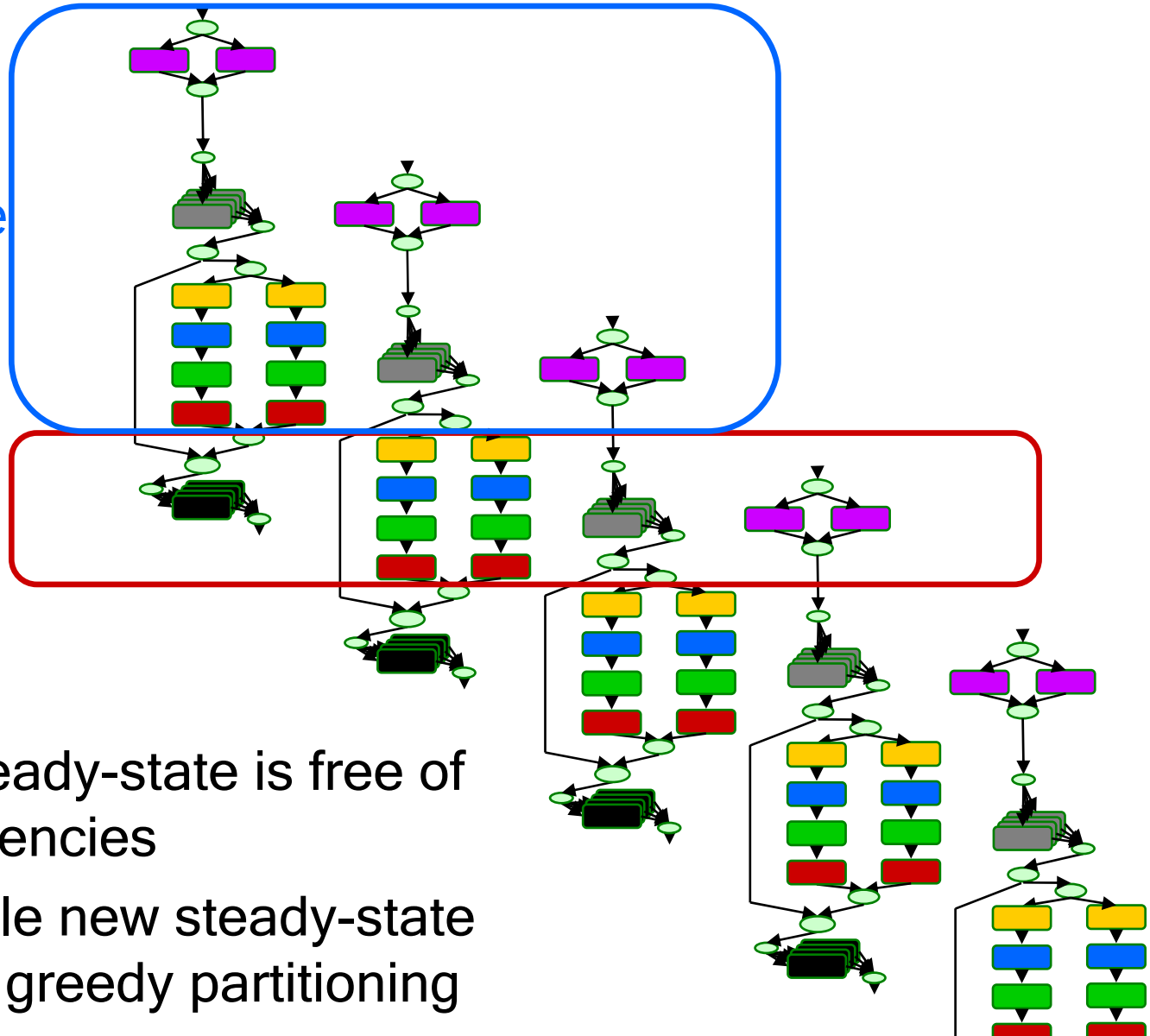
We Can Do Better!





Target 4 core machine

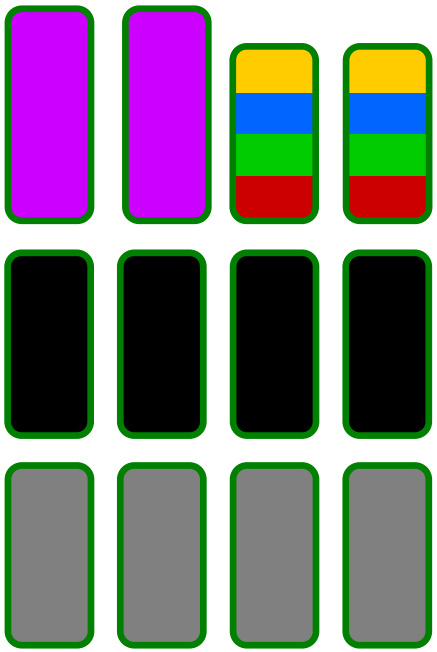
Phase 3: Coarse-Grained Software Pipelining

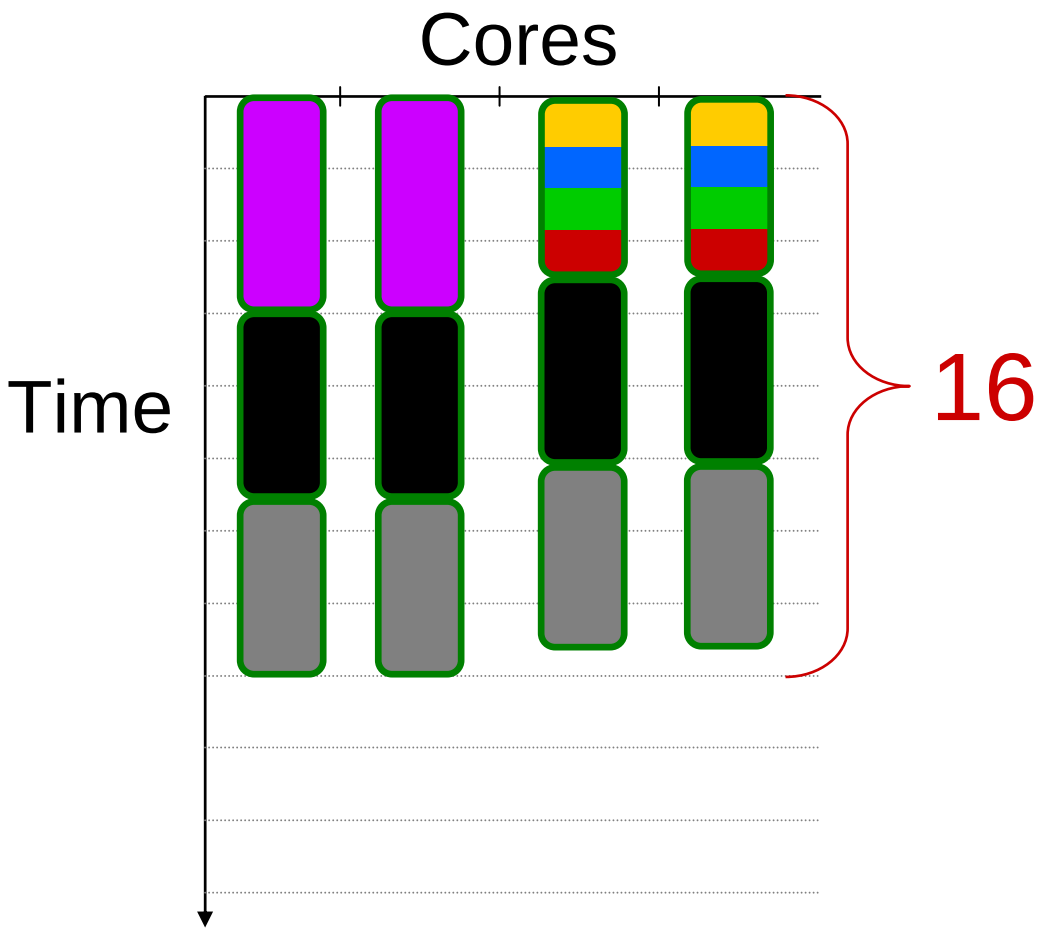


- New steady-state is free of dependencies
- Schedule new steady-state using a greedy partitioning

Greedy Partitioning

To Schedule:





Target 4 core machine

Evaluation: Coarse-Grained Task + Data + Software Pipelining