6.189 Multicore Programming Primer, January (IAP) 2007

6.189 IAP 2007

Lecture 12

StreamIt Parallelizing Compiler

Common Machine Language

- Represent common properties of architectures
  - Necessary for performance
- Abstract away differences in architectures
  - Necessary for portability
- Cannot be too complex
  - Must keep in mind the typical programmer
- C and Fortran were the common machine languages for uniprocessors
  - Imperative languages are not the correct abstraction for parallel architectures.
- What is the correct abstraction for parallel multicore machines?

Common Machine Language for Multicores

- Current offerings:
  - OpenMP
  - MPI
  - High Performance Fortran
- *Explicit* parallel constructs grafted onto imperative language
- Language features obscured:
  - Composability
  - Malleability
  - Debugging
- Huge additional burden on programmer:
  - Introducing parallelism
  - Correctness of parallelism
  - Optimizing parallelism

Explicit Parallelism

- Programmer controls details of parallelism!
- Granularity decisions:
  - if too small, lots of synchronization and thread creation
  - if too large, bad locality
- Load balancing decisions
  - Create balanced parallel sections (not data-parallel)
- Locality decisions
  - Sharing and communication structure
- Synchronization decisions
  - barriers, atomicity, critical sections, order, flushing
- For mass adoption, we need a better paradigm:
  - Where the parallelism is natural
  - Exposes the necessary information to the compiler

Unburden the Programmer

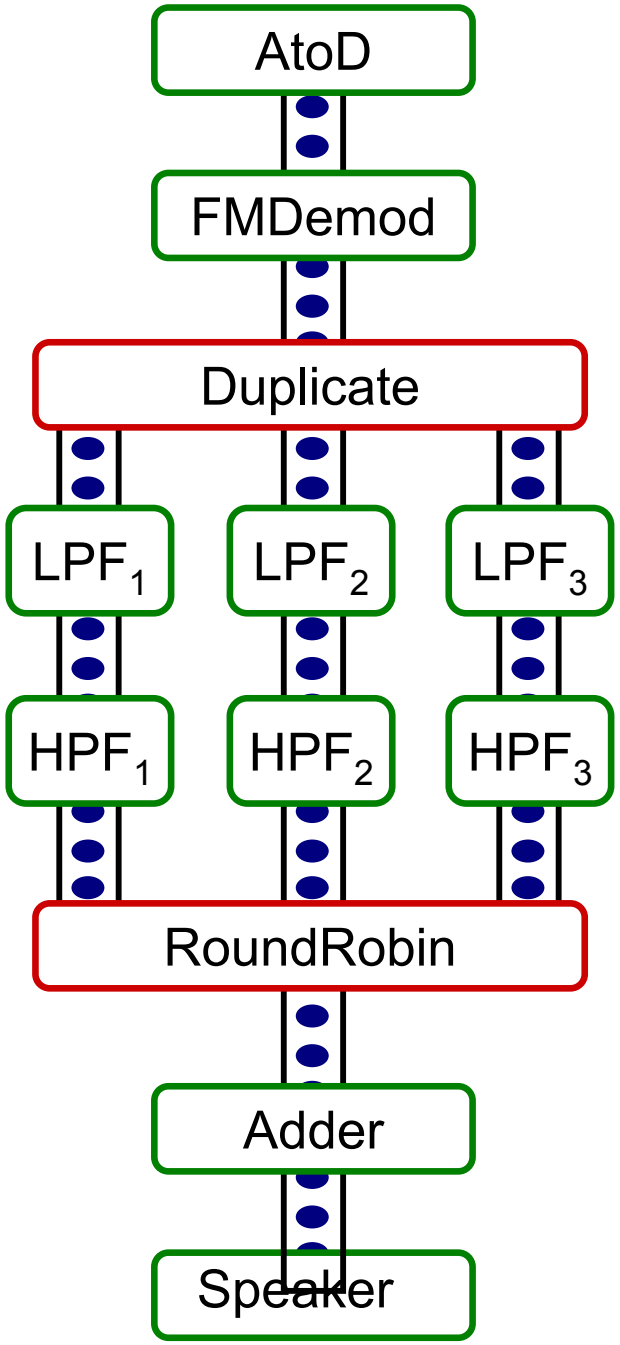
- Move these decisions to compiler!
  - Granularity
  - Load Balancing
  - Locality
  - Synchronization
- Hard to do in traditional languages
  - Can a novel language help?

Properties of Stream Programs

- Regular and repeating computation
- Synchronous Data Flow
- Independent actors
  with explicit communication
- Data items have short lifetimes

Benefits:

- Naturally parallel
- Expose dependencies to compiler
- Enable powerful transformations



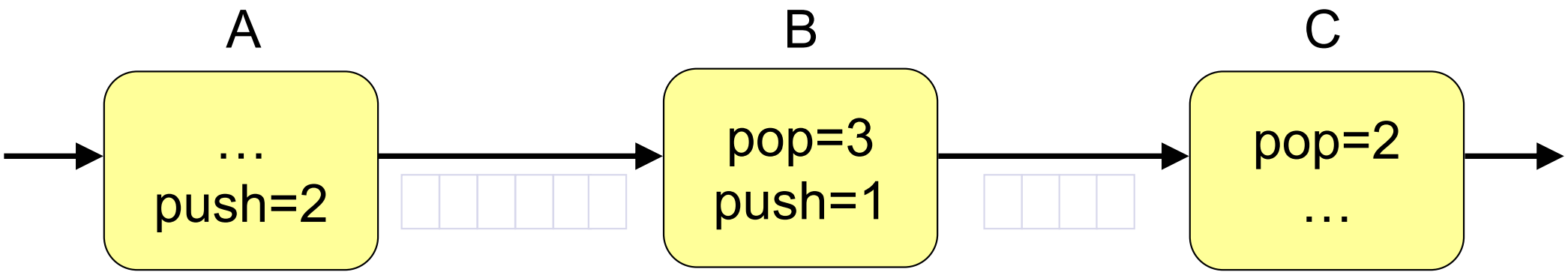
Outline

- Why we need New Languages?
- **Static Schedule**
- Three Types of Parallelism
- Exploiting Parallelism

Steady-State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule

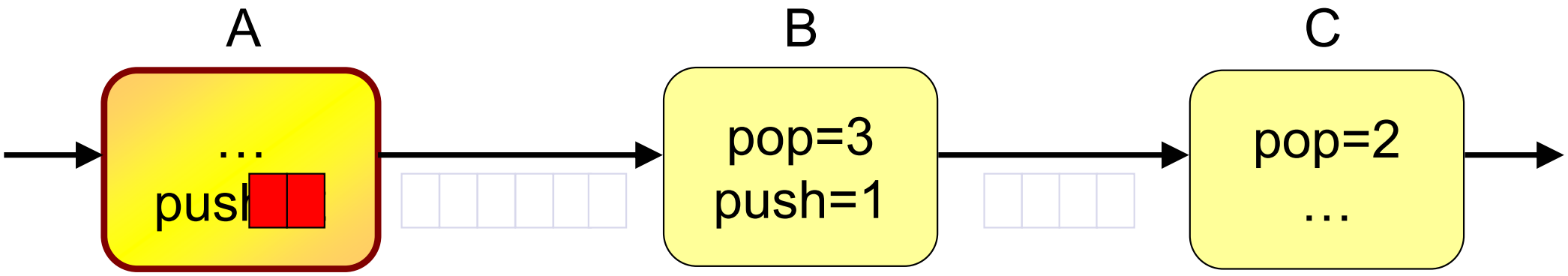
- Schedule = { }



Steady-State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule

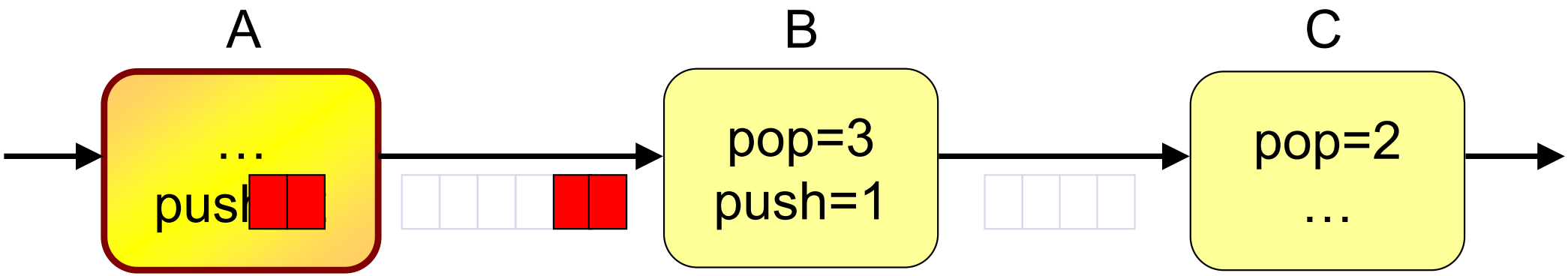
- Schedule = { A }



Steady-State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule

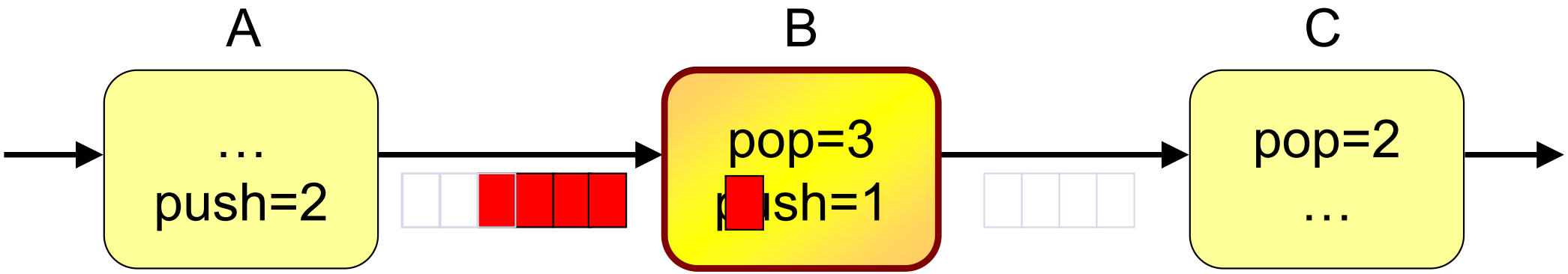
- Schedule = { A, A }



Steady-State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule

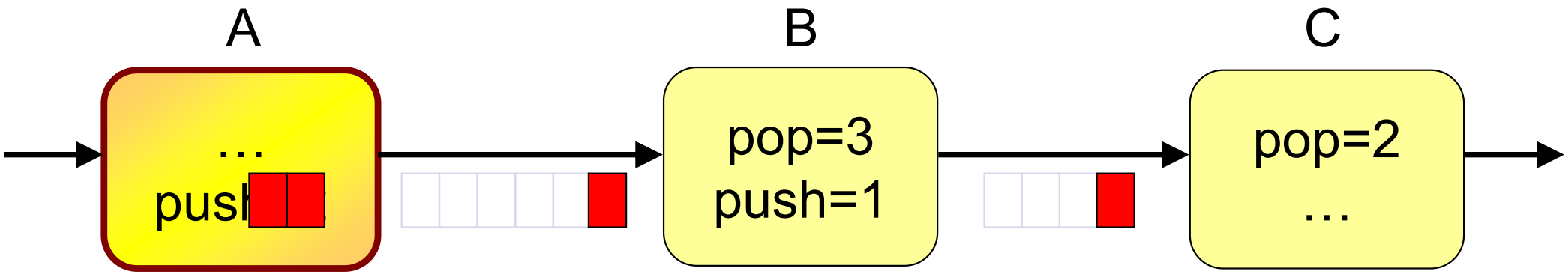
- Schedule = { A, A, B }



Steady-State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule

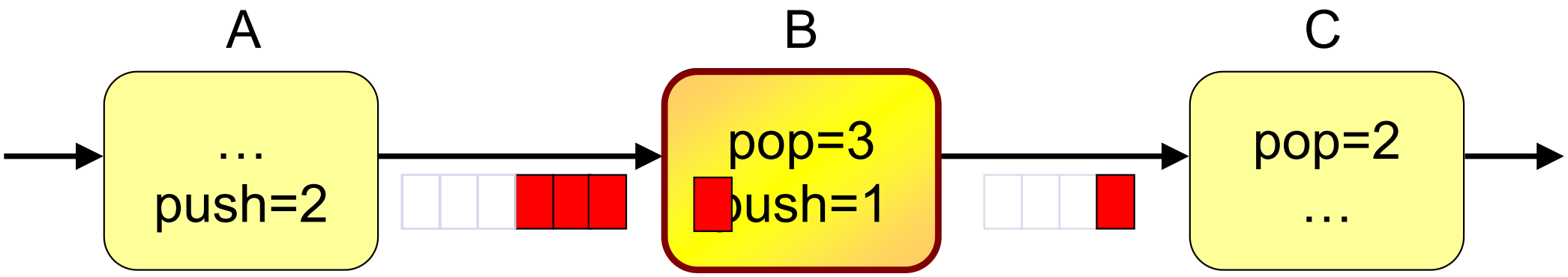
- Schedule = { A, A, B, A }



Steady-State Schedule

- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule

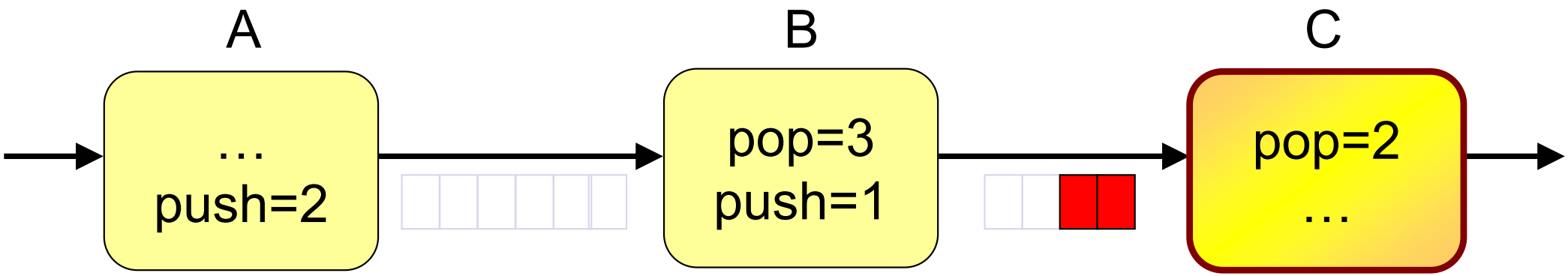
- Schedule = { A, A, B, A, B }



Steady-State Schedule

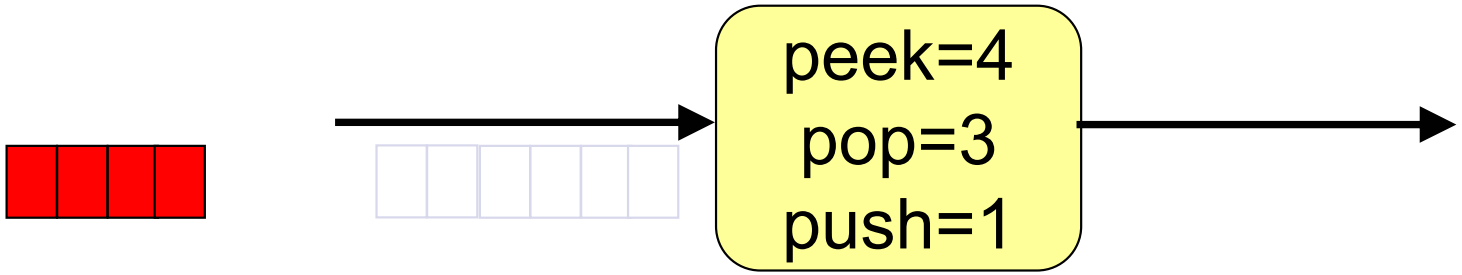
- All data pop/push rates are constant
- Can find a Steady-State Schedule
  - # of items in the buffers are the same before and the after executing the schedule
  - There exist a unique minimum steady state schedule

- Schedule = { A, A, B, A, B, C }



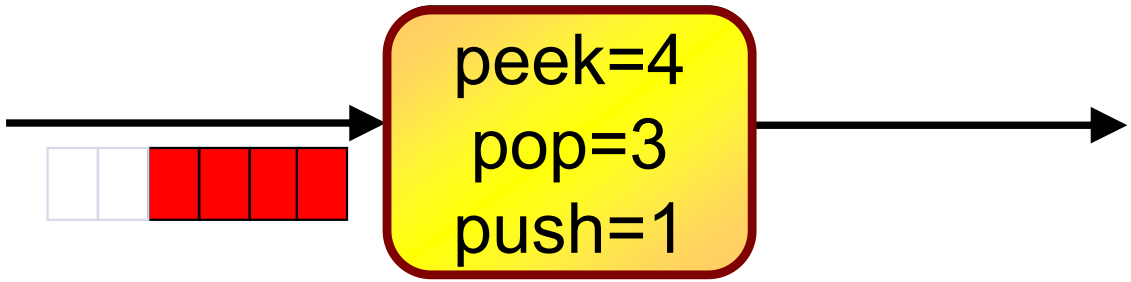
Initialization Schedule

- When peek > pop, buffer cannot be empty after firing a filter
- Buffers are not empty at the beginning/end of the steady state schedule
- Need to fill the buffers before starting the steady state execution



Initialization Schedule

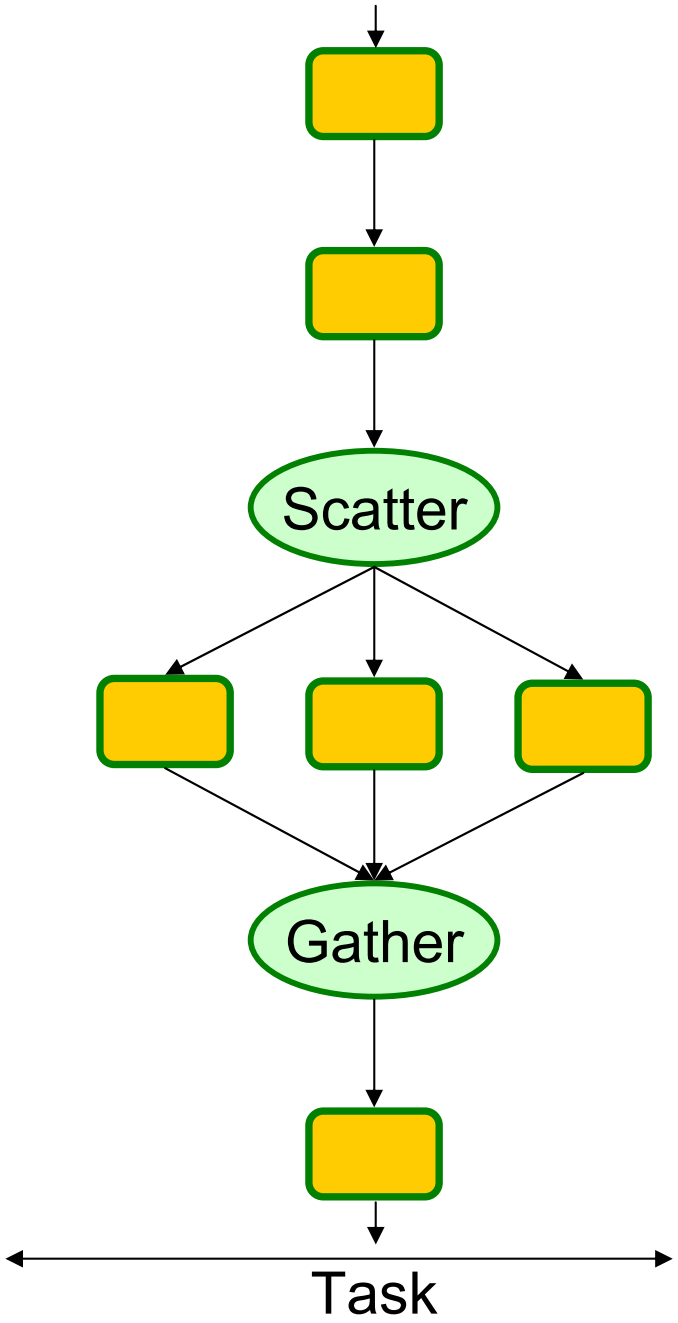
- When peek > pop, buffer cannot be empty after firing a filter
- Buffers are not empty at the beginning/end of the steady state schedule
- Need to fill the buffers before starting the steady state execution



Outline

- Why we need New Languages?
- Static Schedule
- **Three Types of Parallelism**
- Exploiting Parallelism

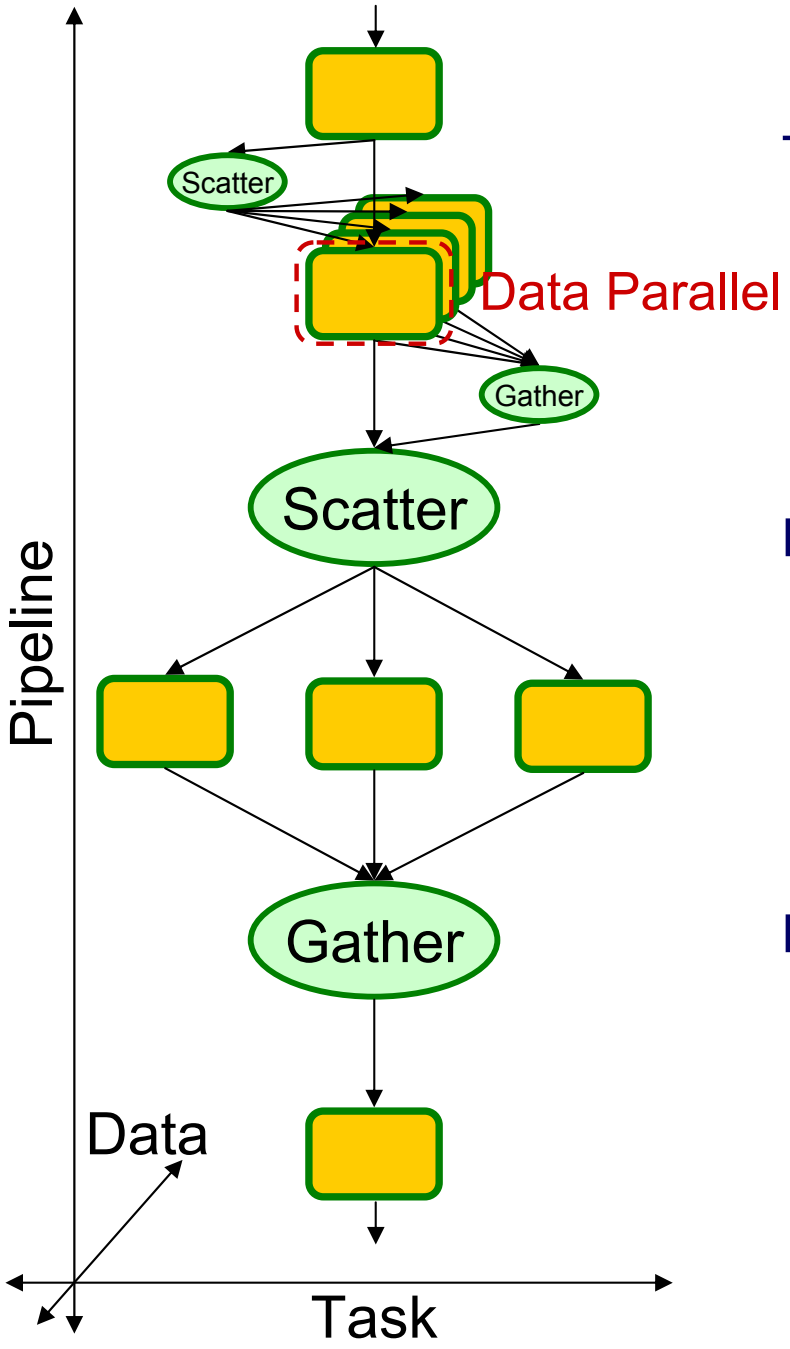
Types of Parallelism



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

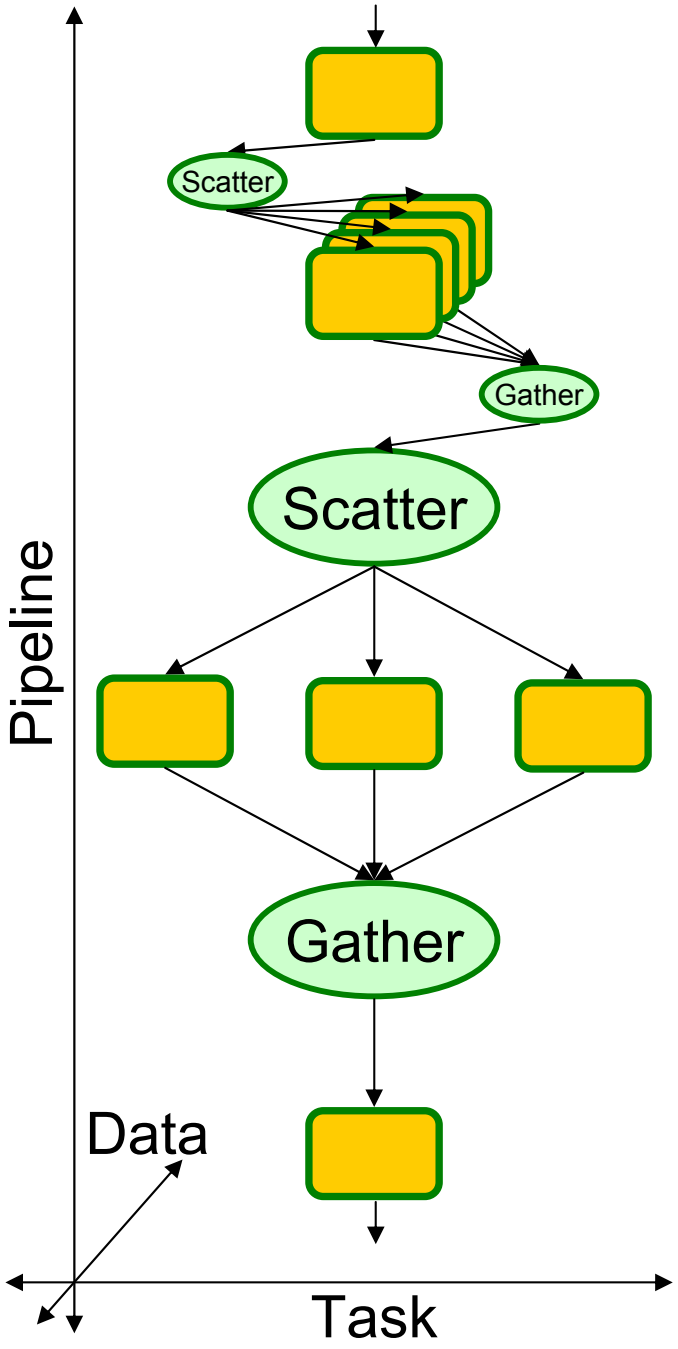
Data Parallelism

- 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**)

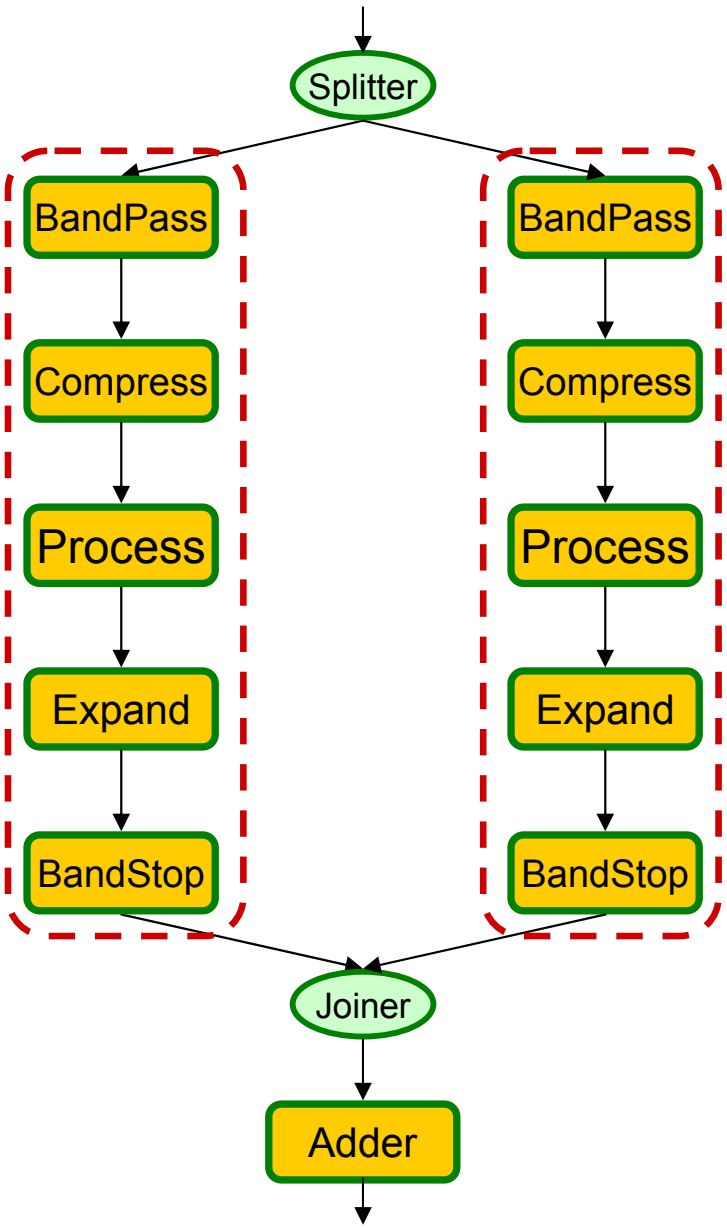
Pipeline Parallelism

- Usually exploited in hardware

Outline

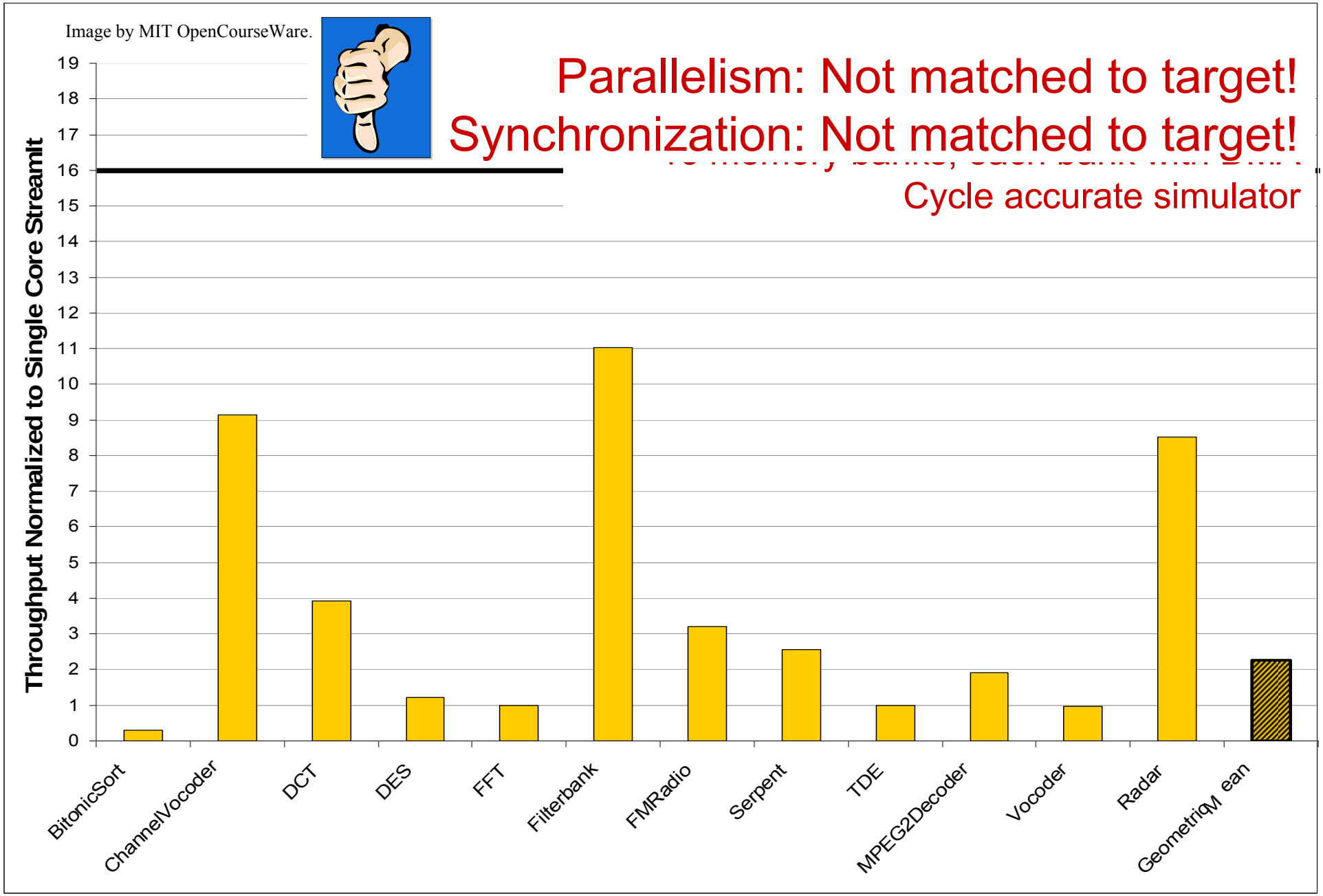
- Why we need New Languages?
- Static Schedule
- Three Types of Parallelism
- **Exploiting Parallelism**

Baseline 1: Task Parallelism

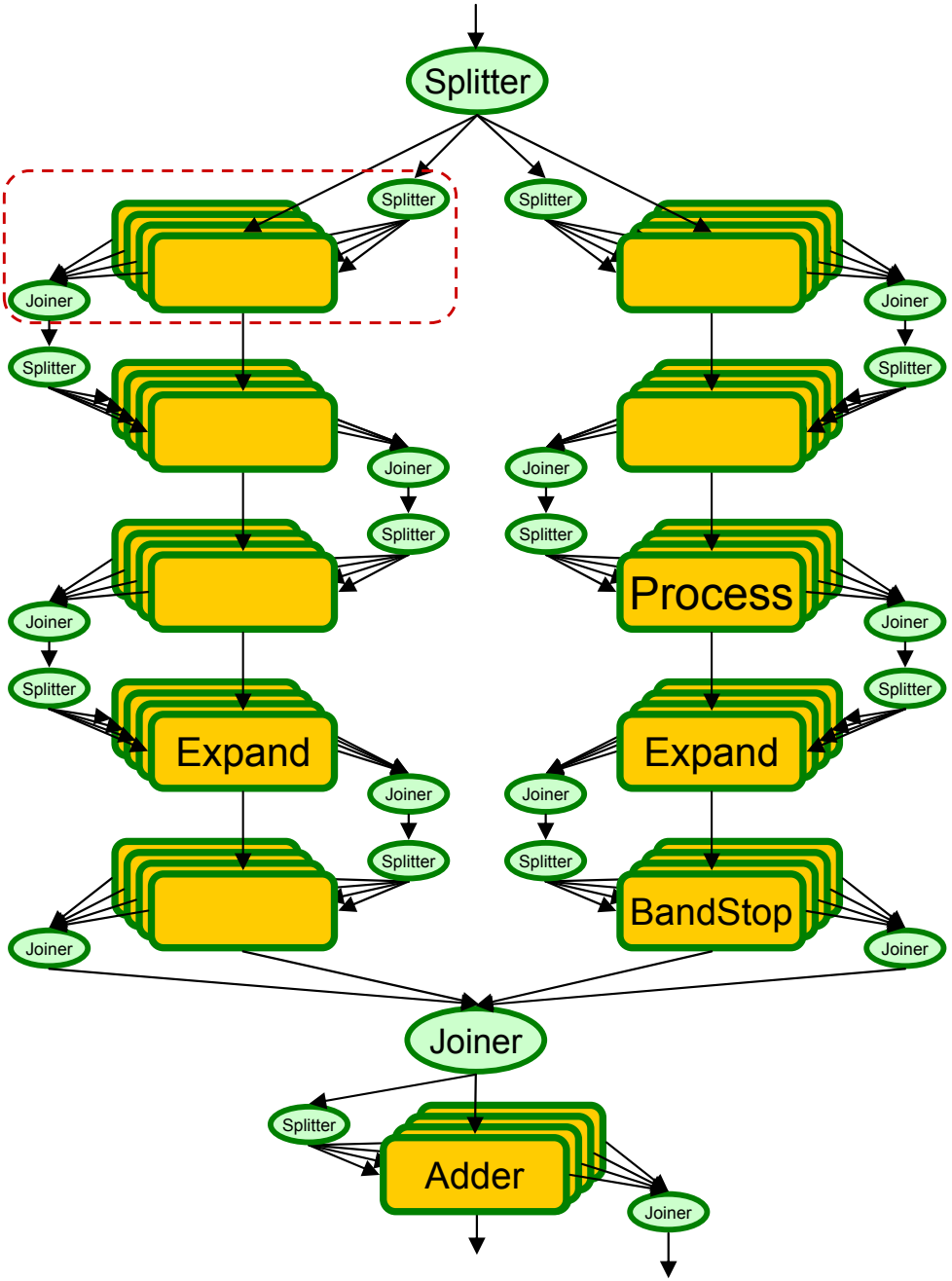


- Inherent task parallelism between two processing pipelines
- Task Parallel Model:
  - Only parallelize explicit task parallelism
  - Fork/join parallelism
- Execute this on a 2 core machine ~2x speedup over single core
- What about 4, 16, 1024, … cores?

Evaluation: Task Parallelism

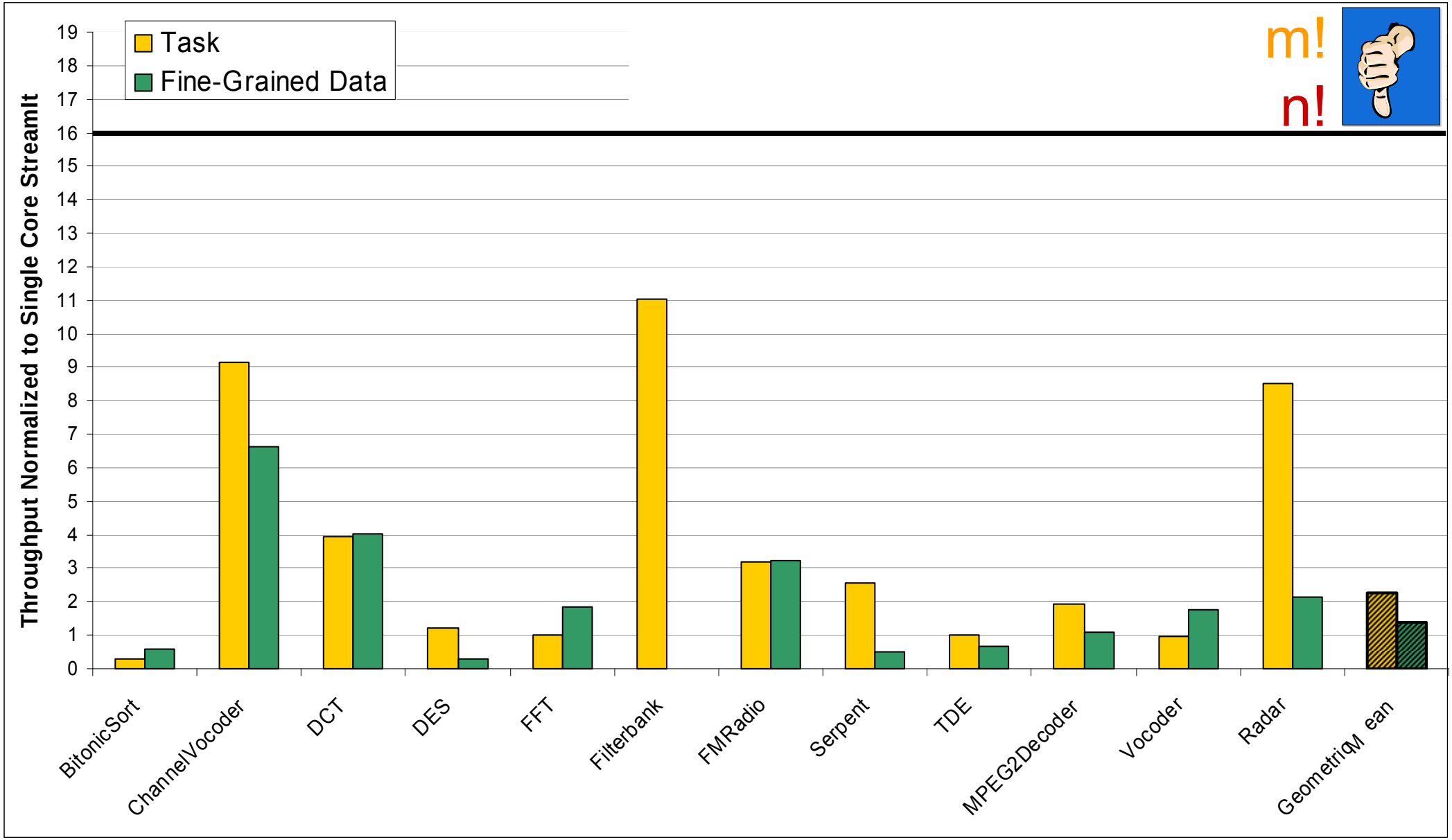


Baseline 2: Fine-Grained Data Parallelism

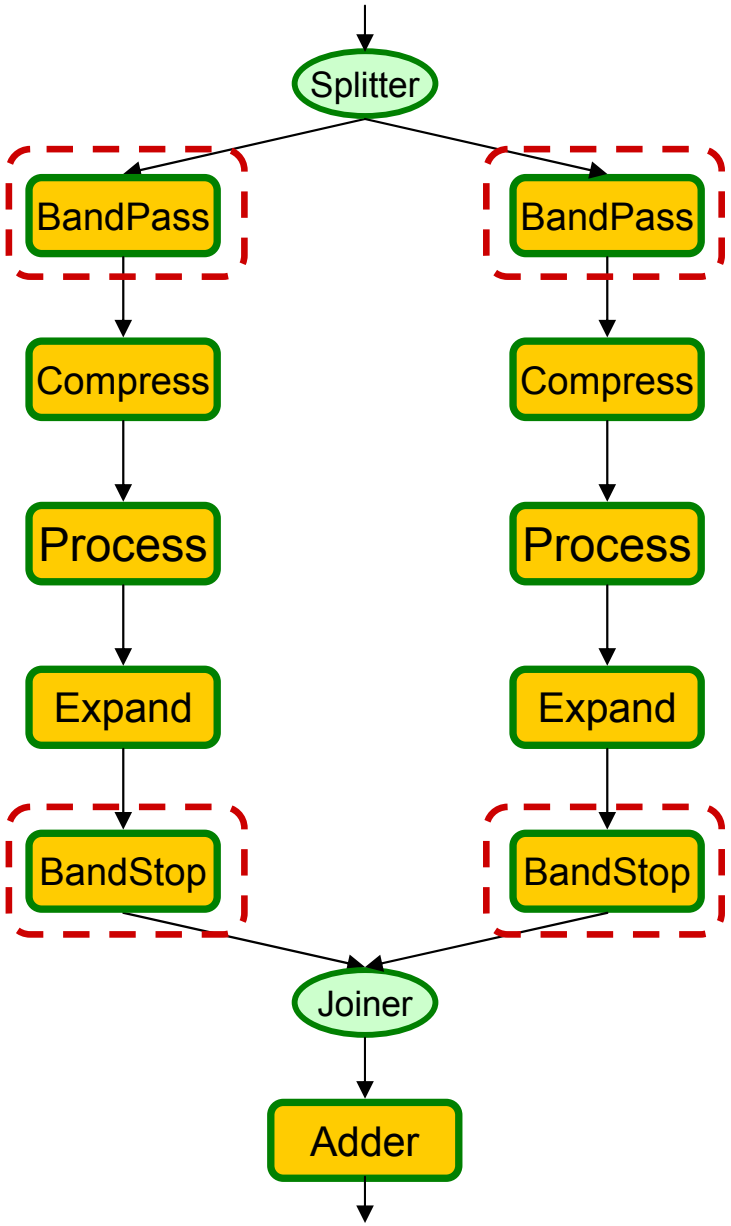


- Each of the filters in the example are stateless
- Fine-grained Data Parallel Model:
  - *Fiss* each stateless filter *N* ways (*N* is number of cores)
  - Remove scatter/gather if possible
- We can introduce data parallelism
  - Example: 4 cores
- Each fission group occupies entire machine

Evaluation: Fine-Grained Data Parallelism

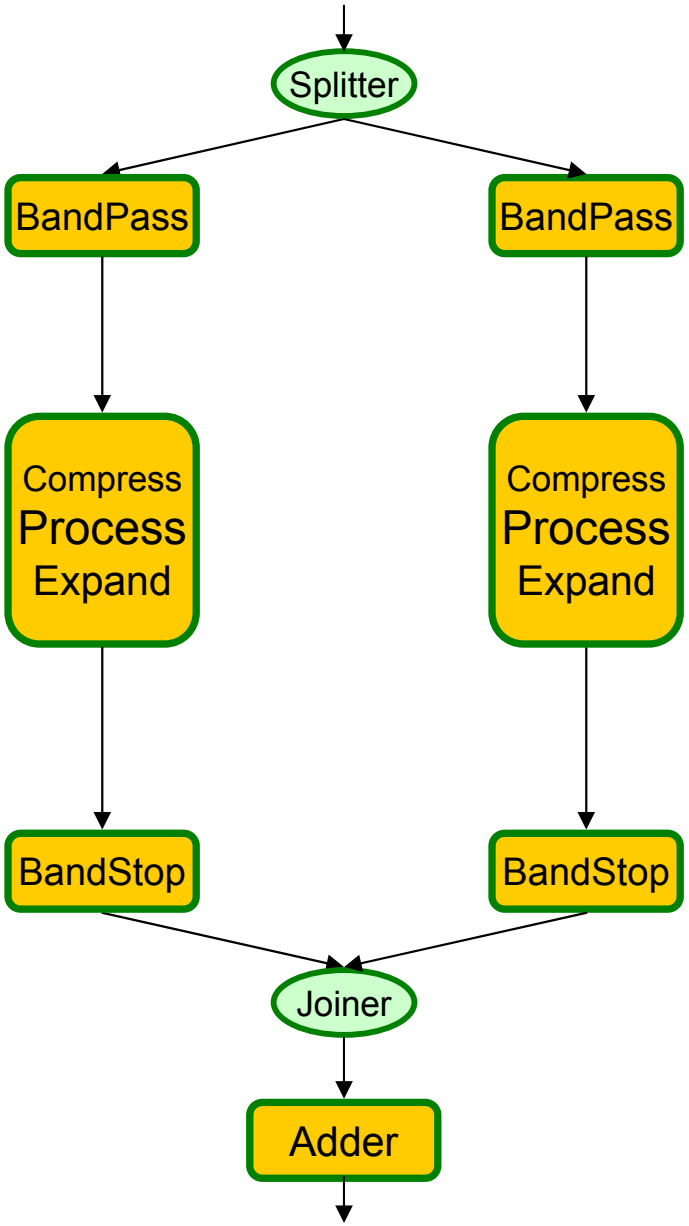


Baseline 3: Hardware Pipeline Parallelism



- The BandPass and BandStop filters contain all the work
- Hardware Pipelining
  - Use a greedy algorithm to fuse adjacent filters
  - Want # filters <= # cores
- Example: 8 Cores

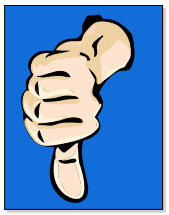
Baseline 3: Hardware Pipeline Parallelism



- The BandPass and BandStop filters contain all the work
- Hardware Pipelining
  - Use a greedy algorithm to fuse adjacent filters
  - Want # filters <= # cores
- Example: 8 Cores
- Resultant stream graph is mapped to hardware
  - One filter per core
- What about 4, 16, 1024, cores?
  - Performance dependent on fusing to a load-balanced stream graph

Evaluation: Hardware Pipeline Parallelism

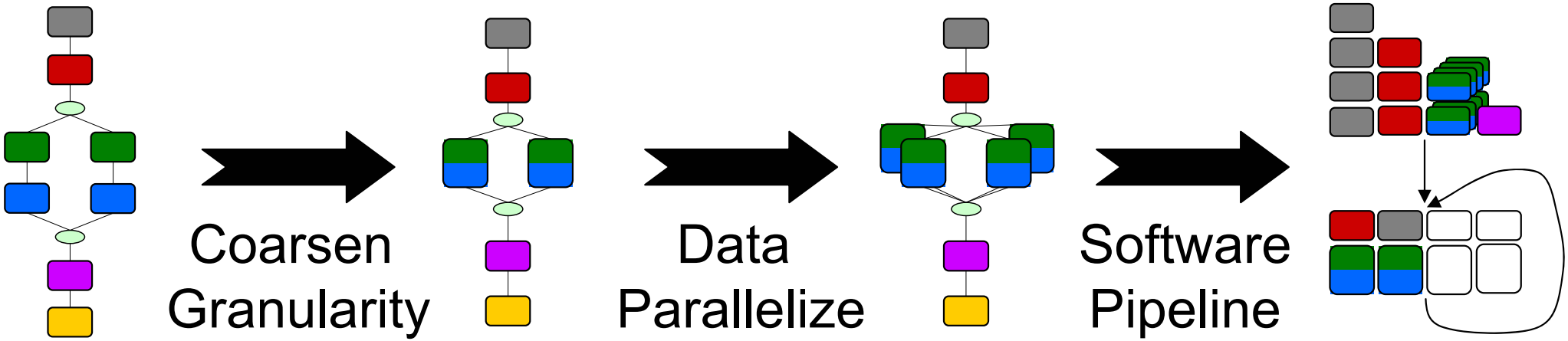
Image by MIT OpenCourseWare.



Parallelism: Not matched to target!

Synchronization: Not matched to target!

The StreamIt Compiler

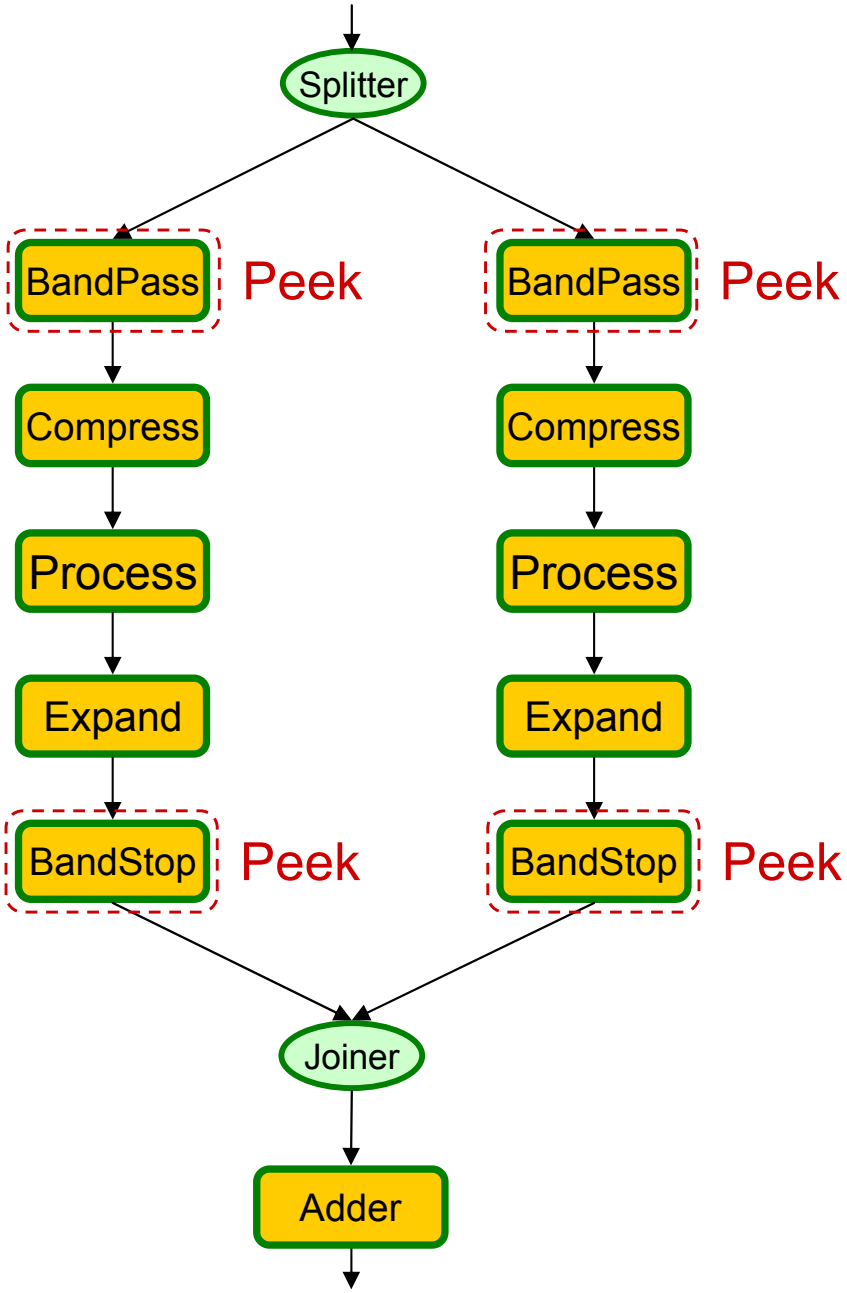


1. Coarsen: Fuse stateless sections of the graph
2. Data Parallelize: parallelize stateless filters
3. Software Pipeline: parallelize stateful filters

Compile to a 16 core architecture

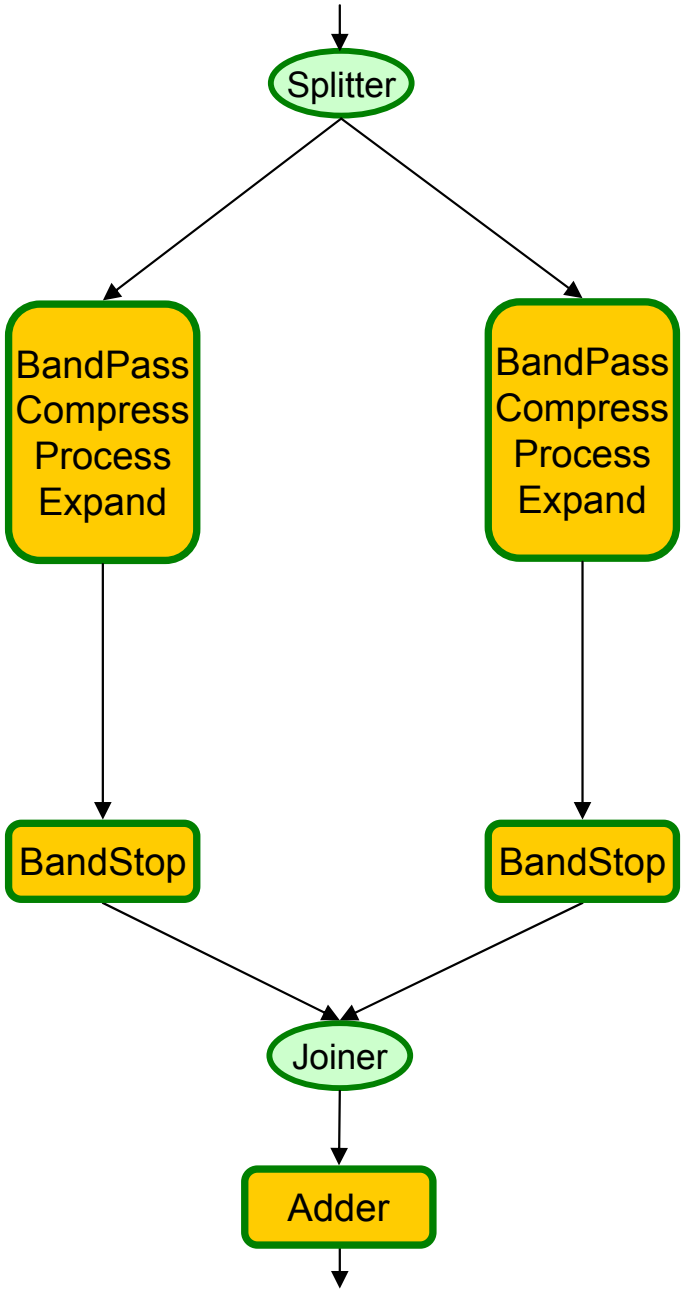
- 11.2x mean throughput speedup over single core

Phase 1: Coarsen the Stream Graph



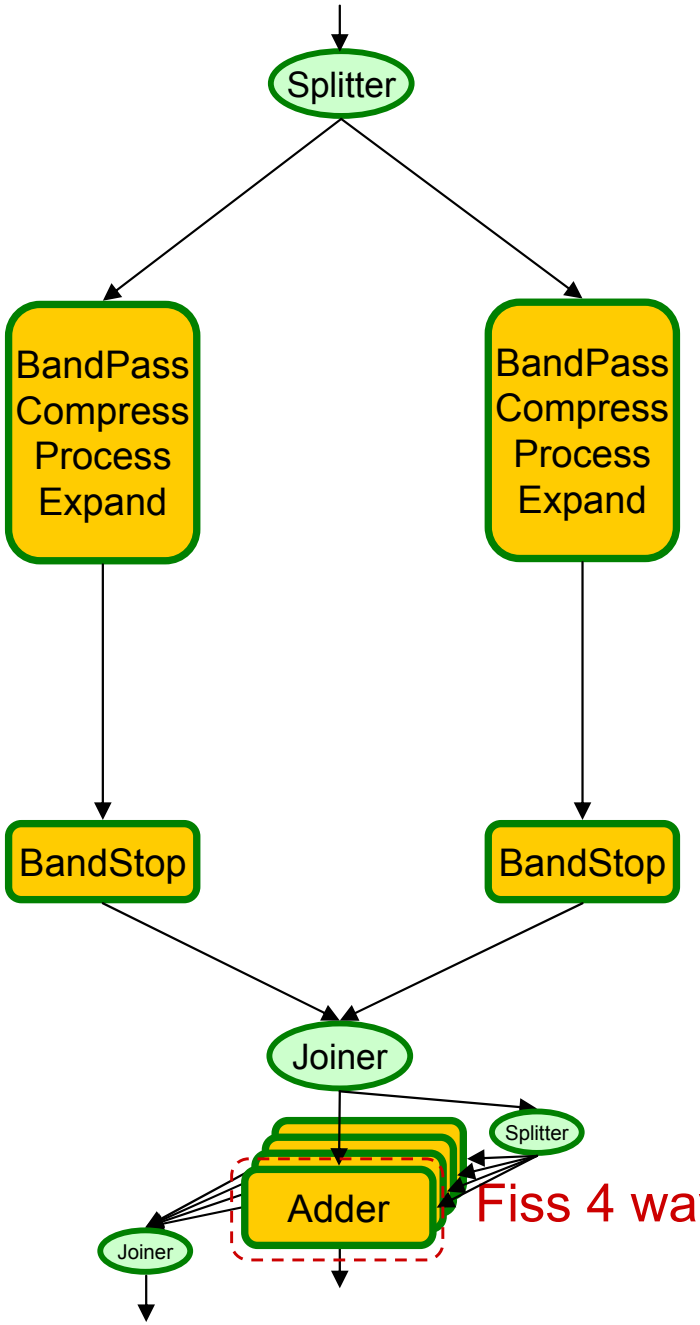
- Before data-parallelism is exploited
- *Fuse* stateless pipelines as much as possible without introducing state
  - Don't fuse stateless with stateful
  - Don't fuse a peeking filter with anything upstream

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

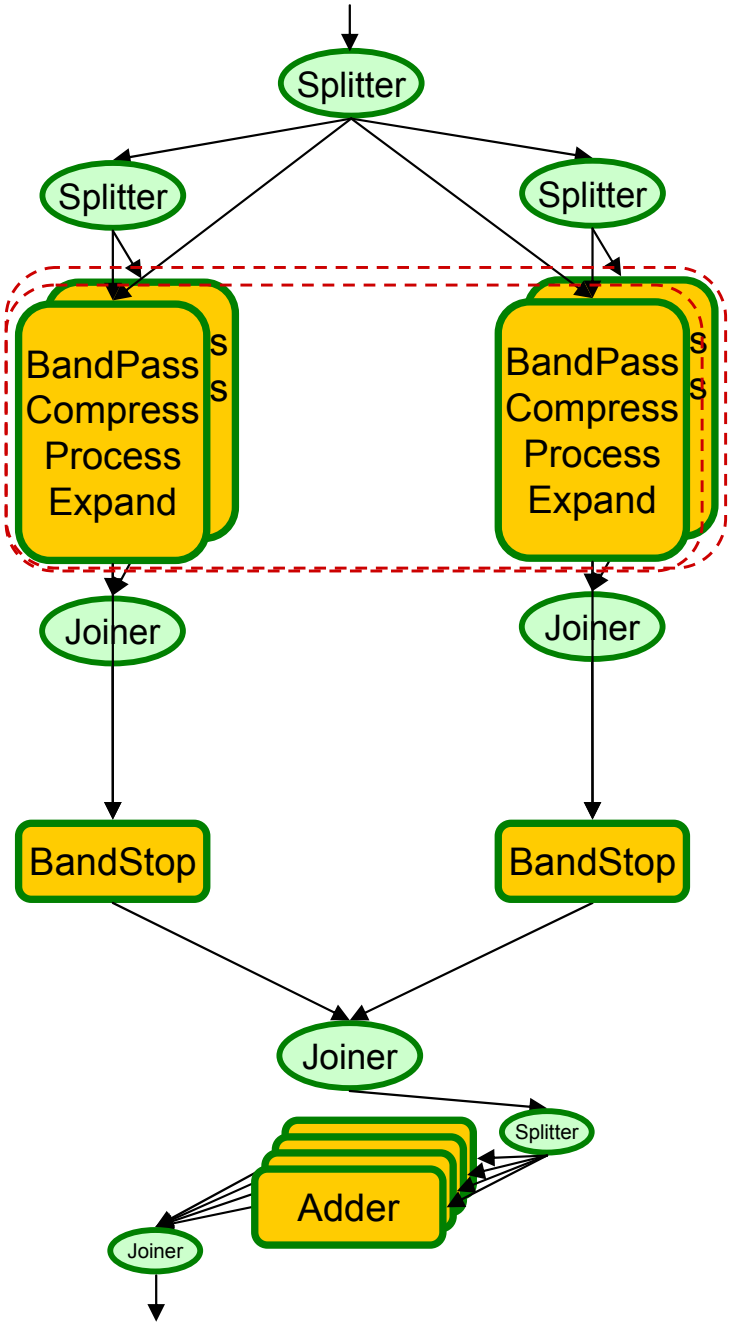
Phase 2: Data Parallelize



Data Parallelize for 4 cores

Fiss 4 ways, to occupy entire chip

Phase 2: Data Parallelize



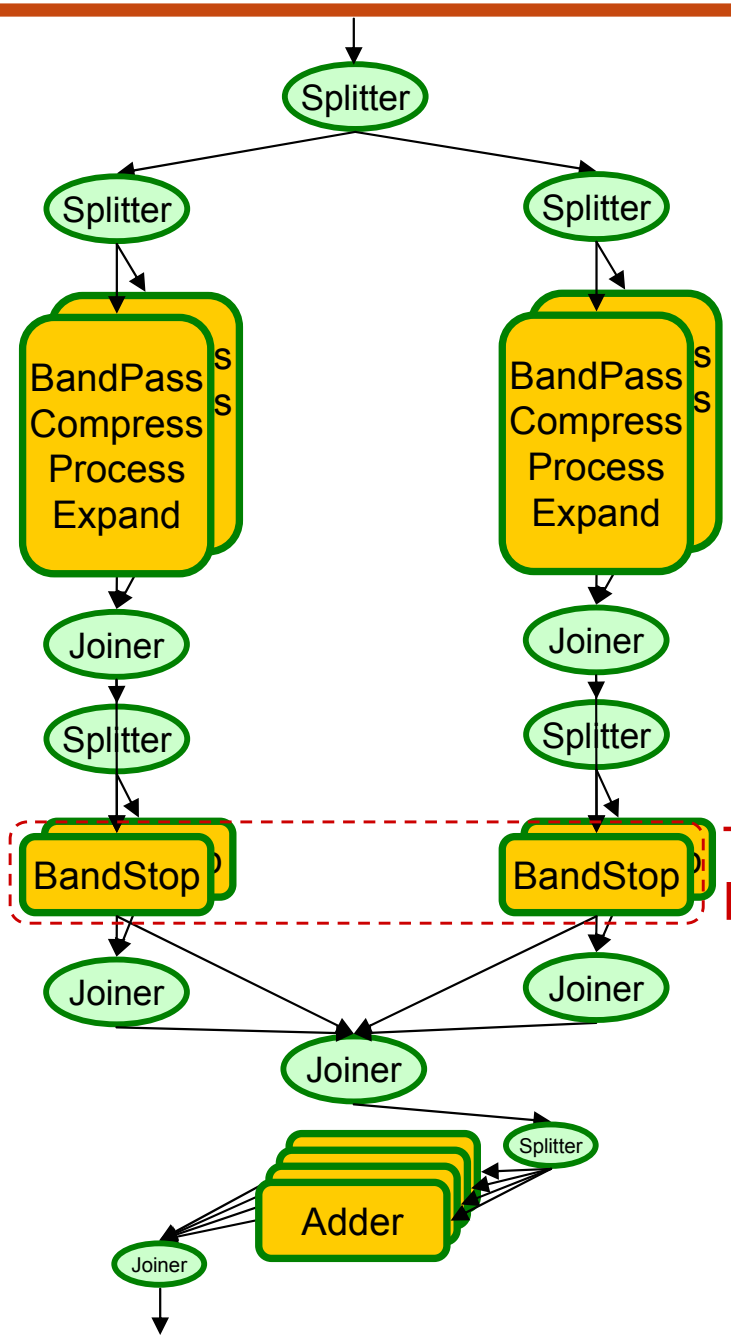
Data Parallelize for 4 cores

Task parallelism!

Each fused filter does equal work

Fiss each filter 2 times to occupy entire chip

Phase 2: Data Parallelize

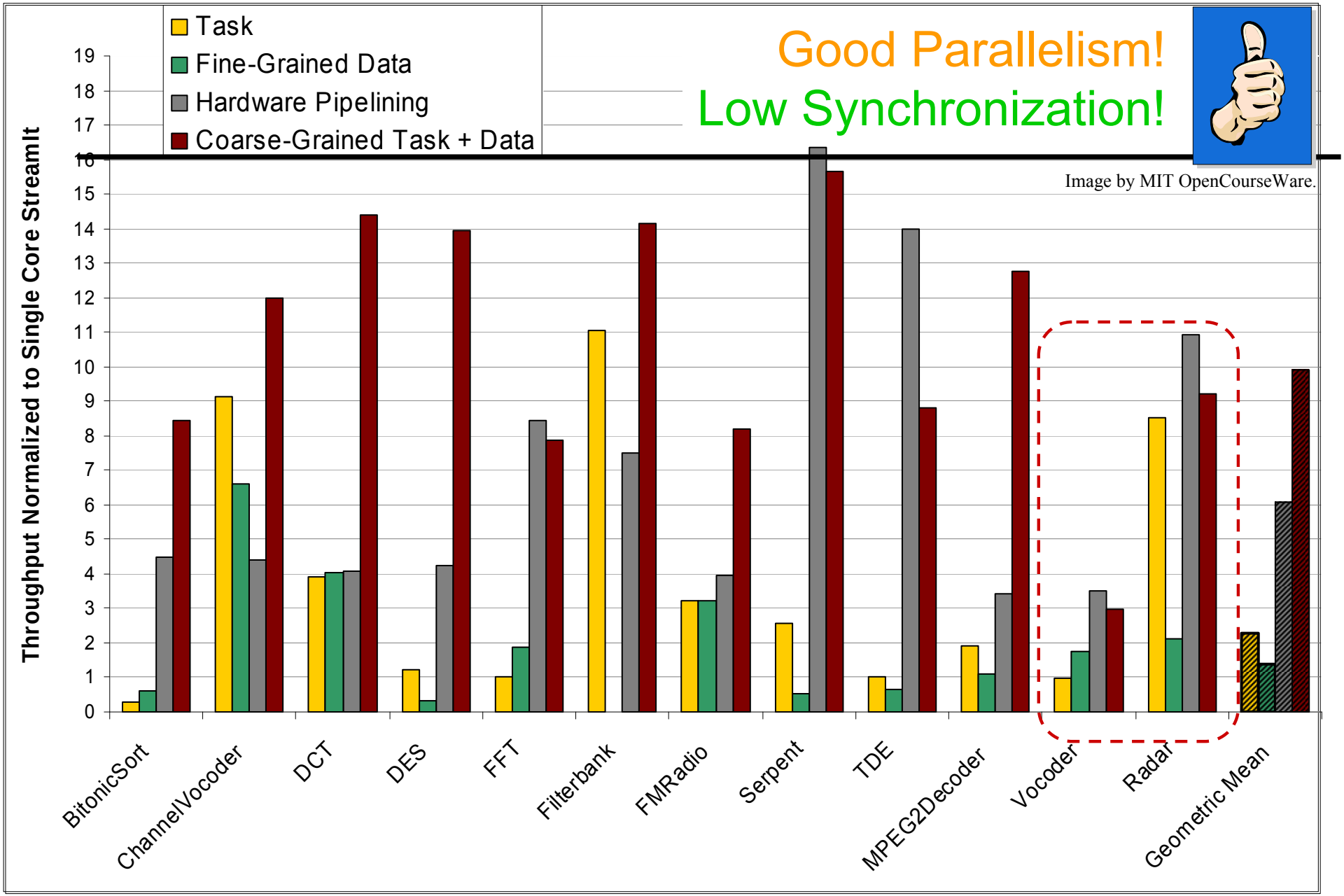


Data Parallelize for 4 cores

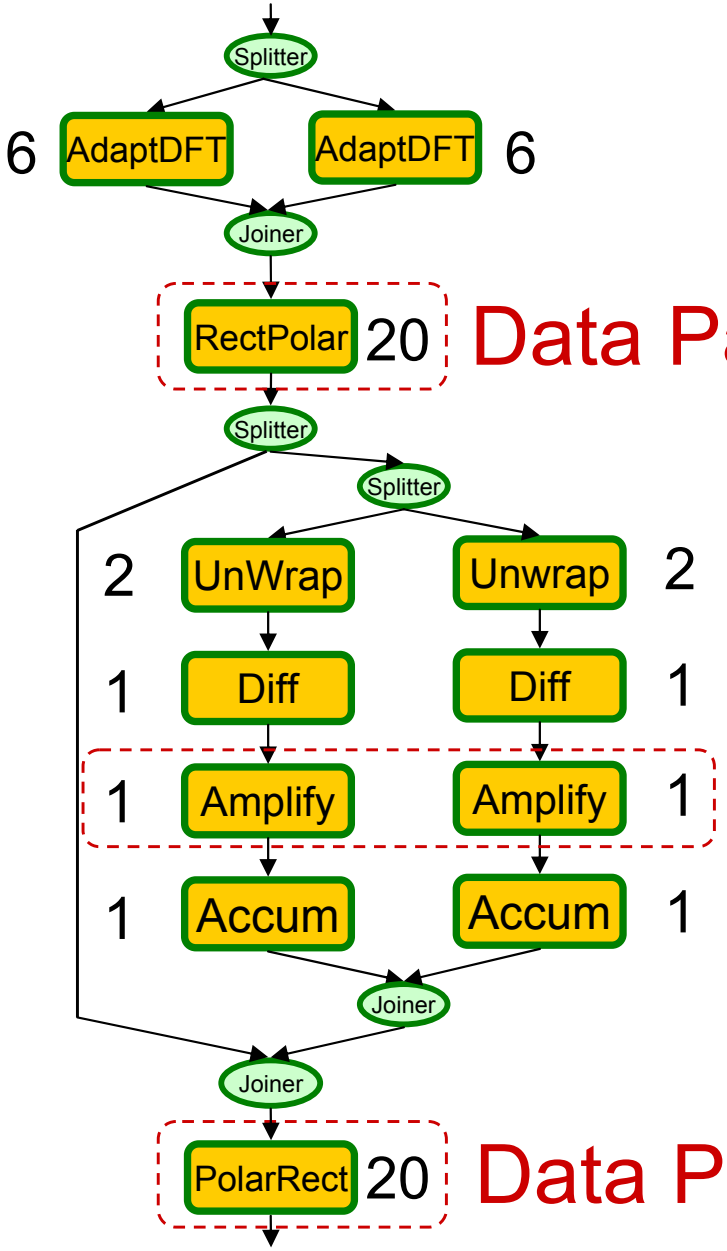
- Task-conscious data parallelization
  - Preserve task parallelism
- Benefits:
  - Reduces global communication and synchronization

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

Evaluation: Coarse-Grained Data Parallelism



Simplified Vocoder



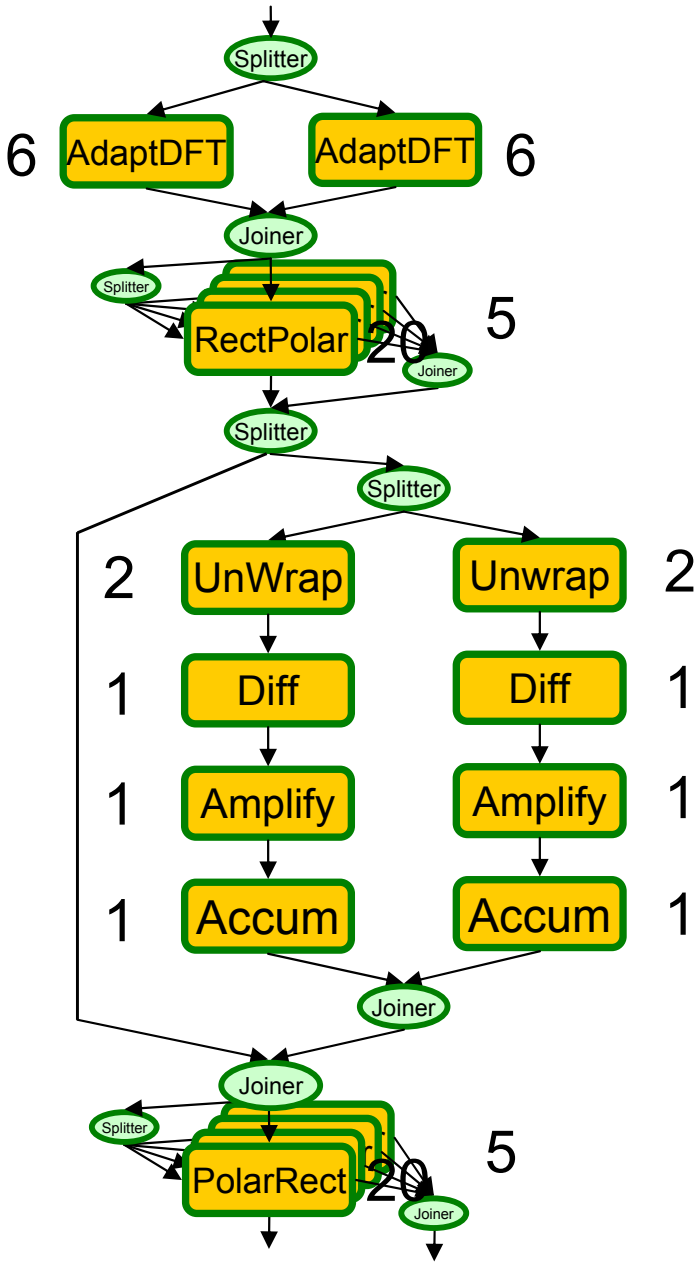
Data Parallel

Data Parallel, but too little work!

Data Parallel

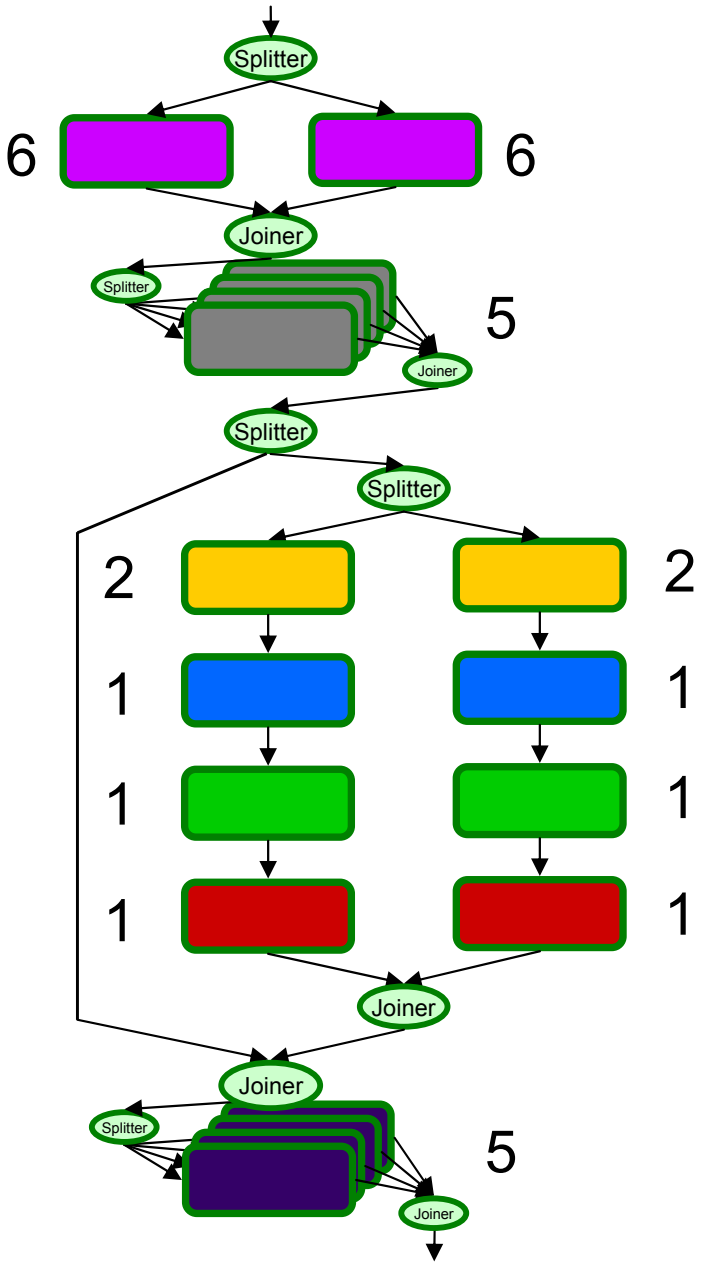
Target a 4 core machine

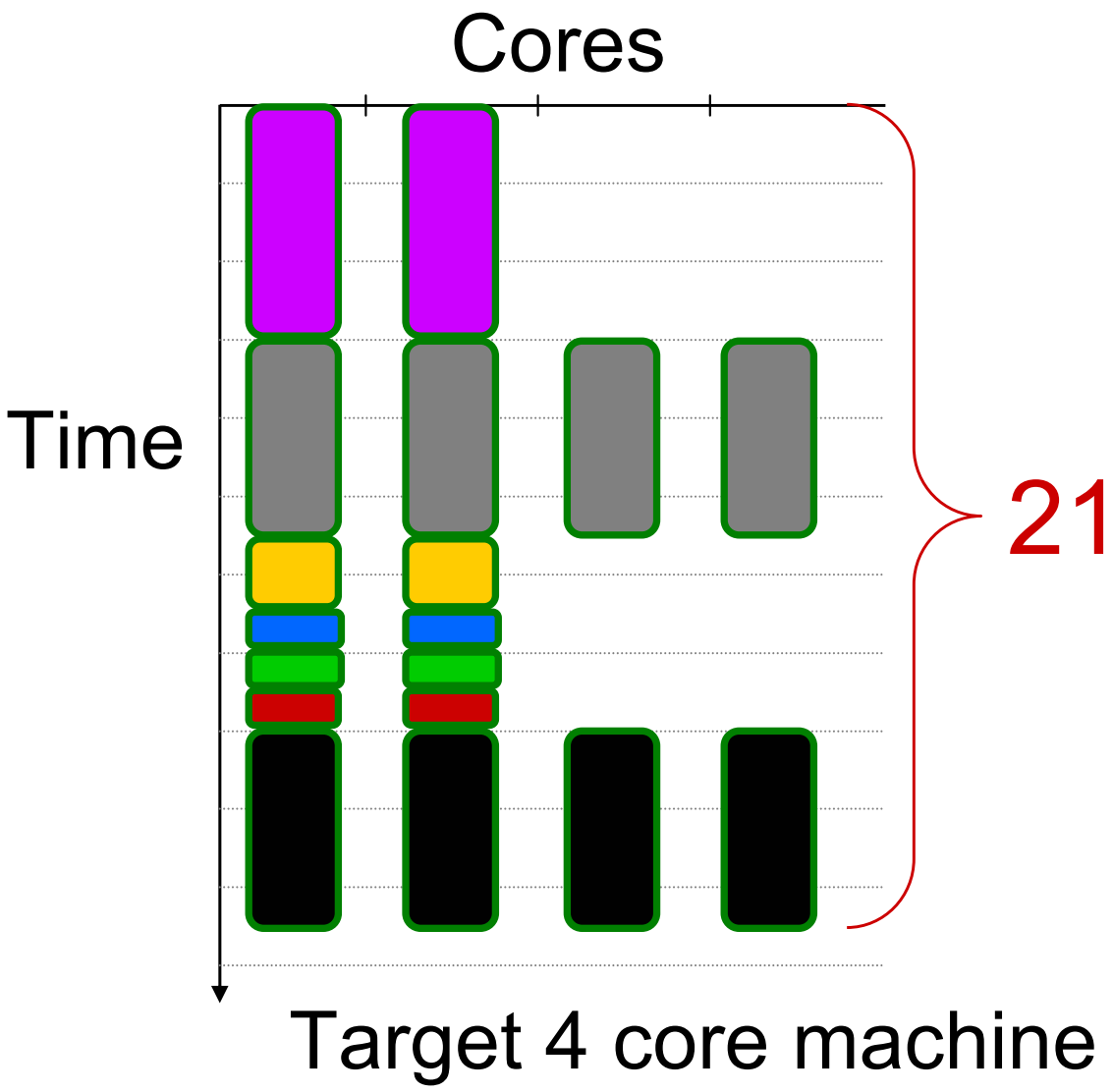
Data Parallelize



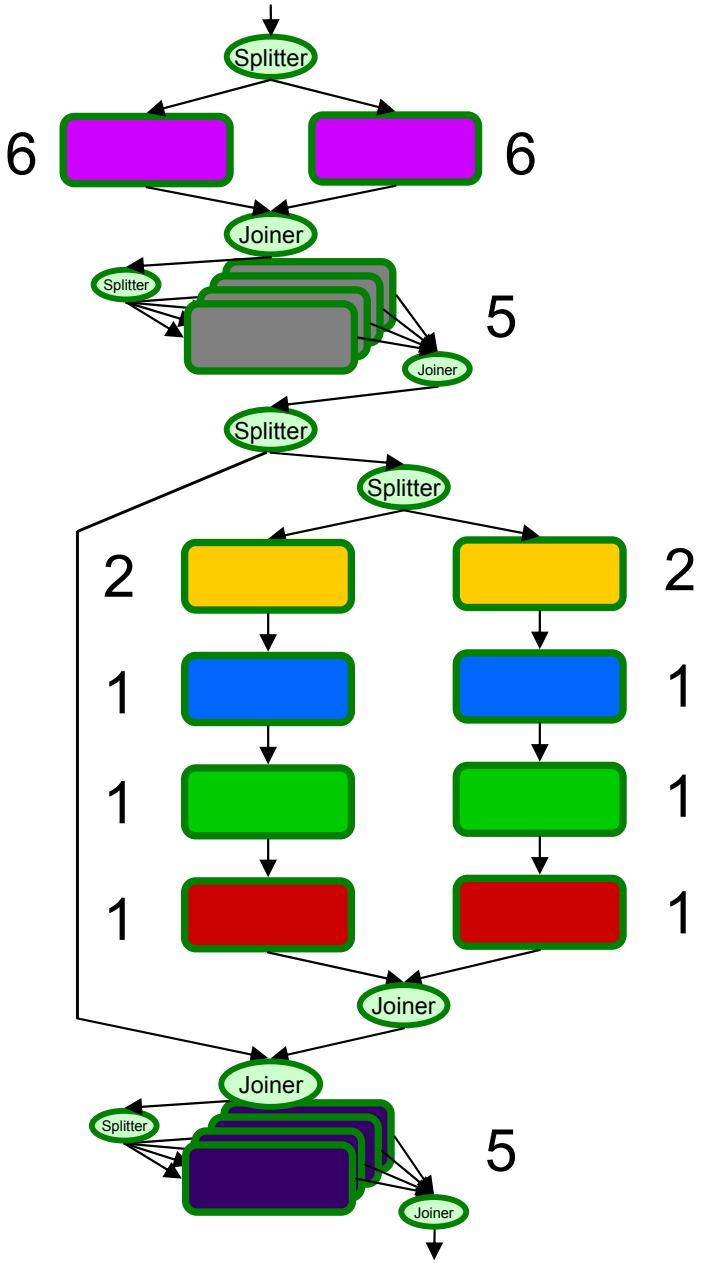
Target a 4 core machine

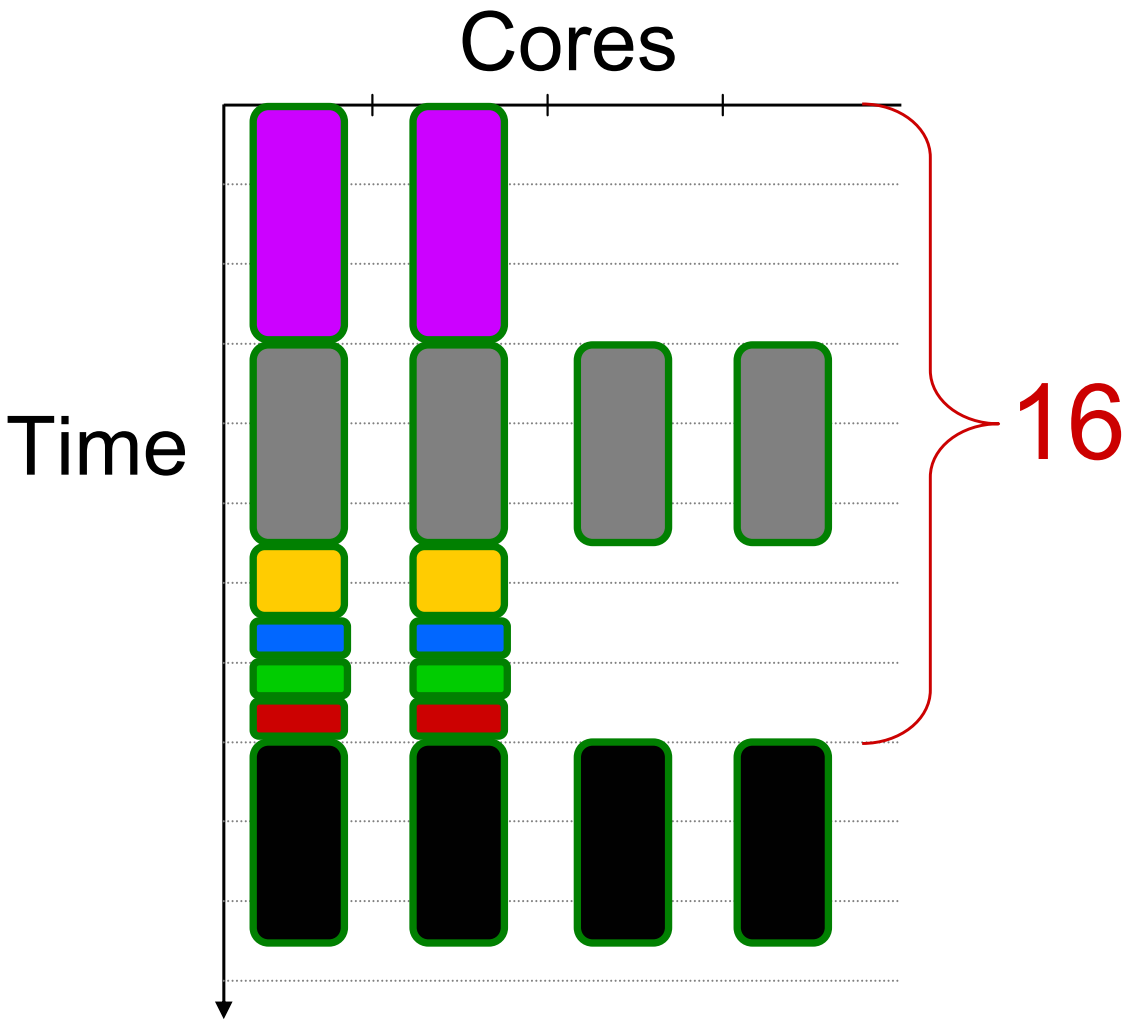
Data + Task Parallel Execution





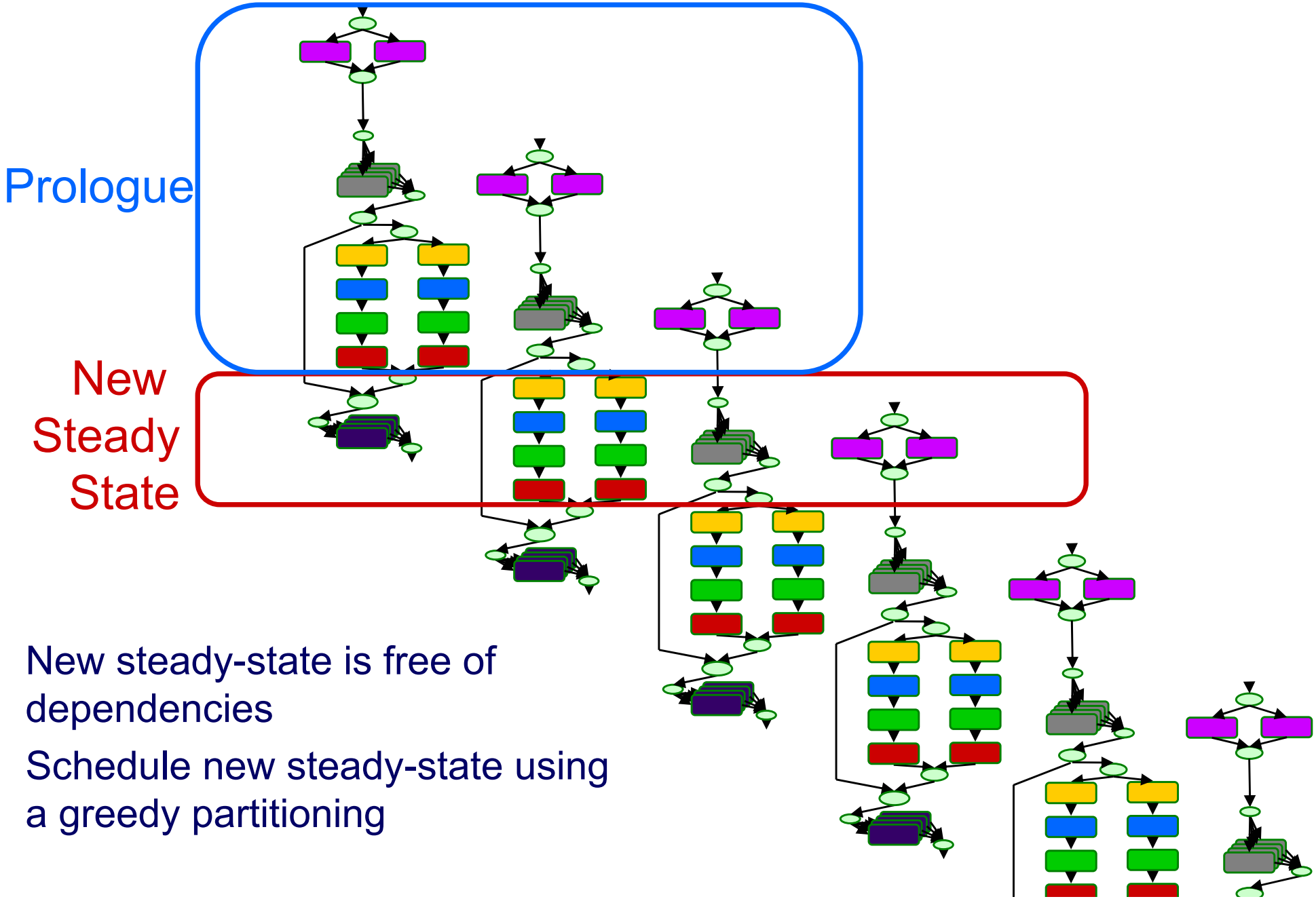
We Can Do Better!





Target 4 core machine

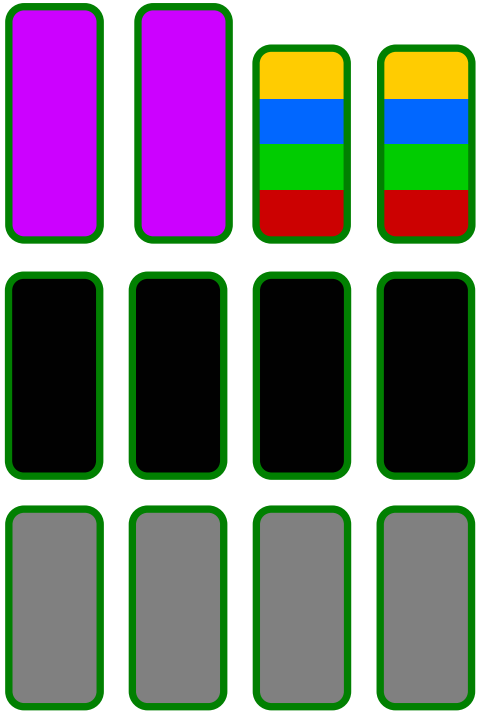
Phase 3: Coarse-Grained Software Pipelining

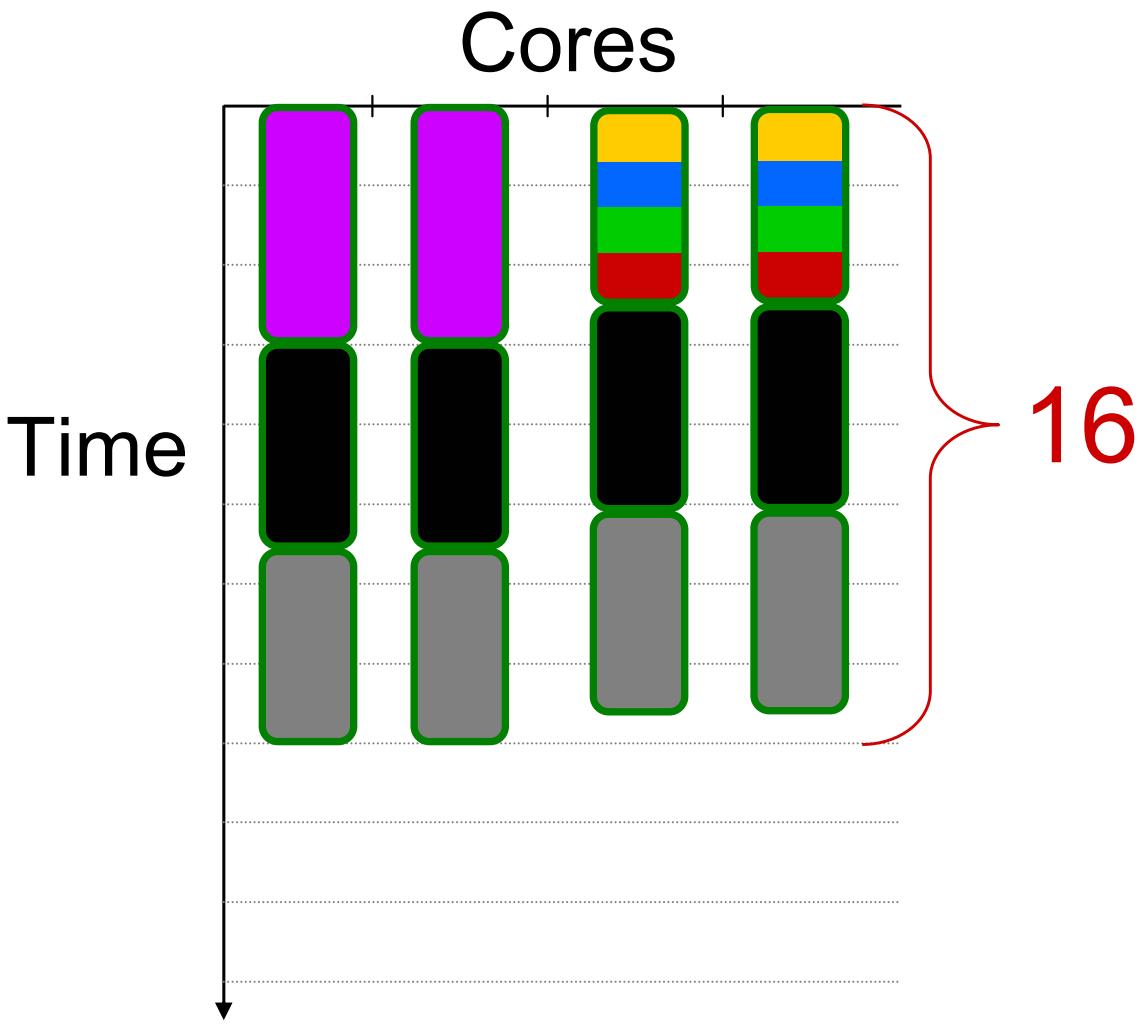


- New steady-state is free of dependencies
- Schedule new steady-state using a greedy partitioning

Greedy Partitioning

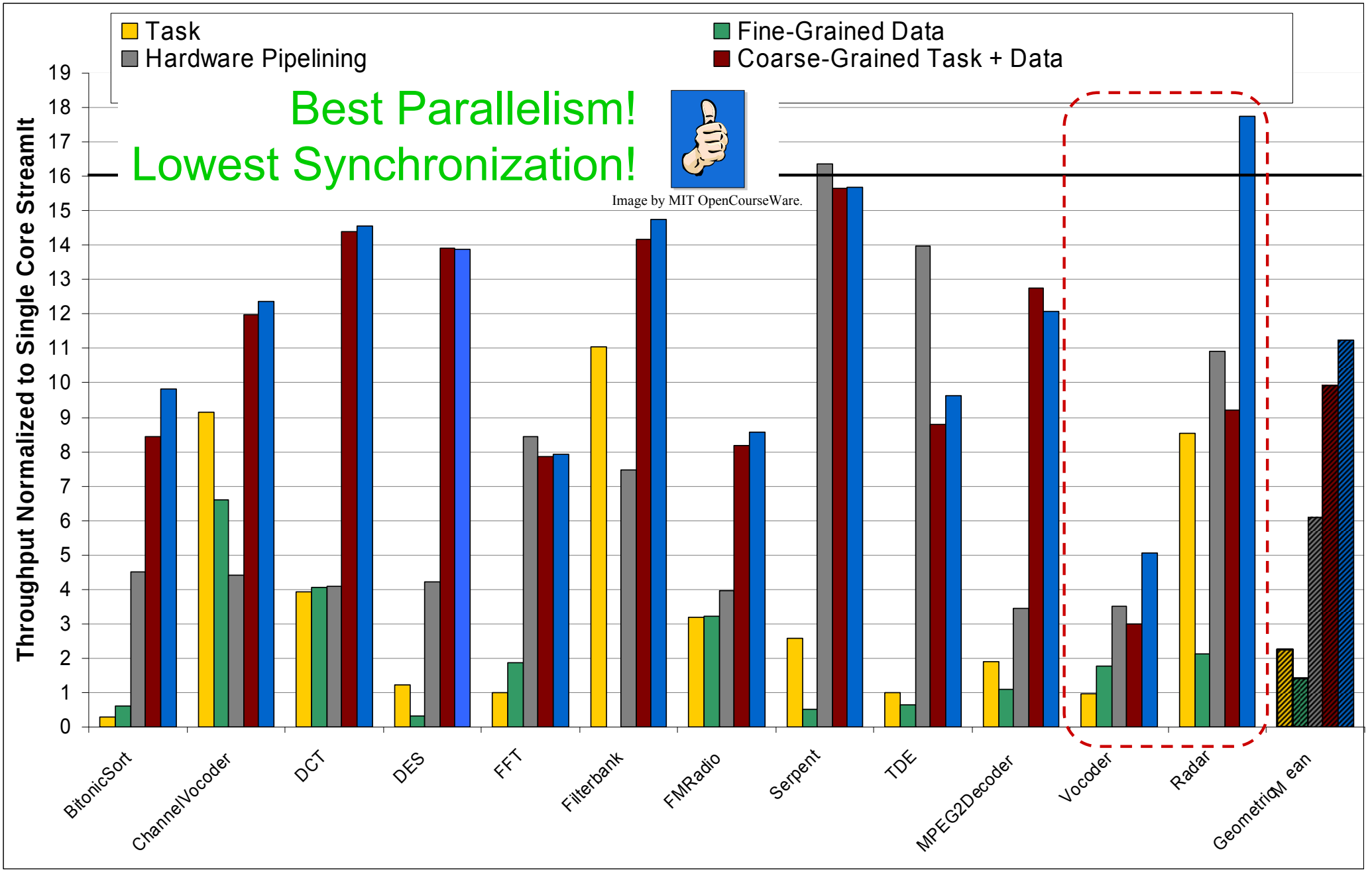
To Schedule:





Target 4 core machine

Evaluation: Coarse-Grained Task + Data + Software Pipelining



Summary

- Streaming model naturally exposes task, data, and pipeline parallelism
- This parallelism must be exploited at the correct granularity and combined correctly

	Task	Fine-Grained Data	Hardware Pipelining	Coarse-Grained Task + Data	Coarse-Grained Task + Data + Software Pipeline
Parallelism	Application Dependent	Good	Application Dependent	Good	Best
Synchronization	Application Dependent	High	Application Dependent	Low	Lowest

- Robust speedups across varied benchmark suite