Multi-core Aware Applications in CMS

Christopher D Jones *FNAL*
Peter Elmer *Princeton*
Liz Sexton-Kennedy *FNAL*
Chris Green
Anthony Baldocci

For the Offline and Computing Project of CMS

Overview

Why bother?

Forking: Copy on Write

Framework design

Measurements

Estimating performance of threading

Conclusion

Why Bother?

HEP processing is naturally parallelizable

We have billions of events

Each event can be processed independently

Memory is becoming a limitation

Historically GB/US$ increases at the same rate as number of transistors in a CPU

- http://www.jcmit.com/memoryprice.htm

Funding levels are not guaranteed to stay this high

- We can afford 2GB/core now but may not in the future

Opportunistic use of grid sites improves if we lower our memory requirements

- Not all grid sites have 2GB/core

Technical limitations on connecting many cores to shared system memory

- http://www.intel.com/technology/itj/2007/v11i3/3-bandwidth/7-conclusion.htm

Multi-core aware applications can improve memory sharing

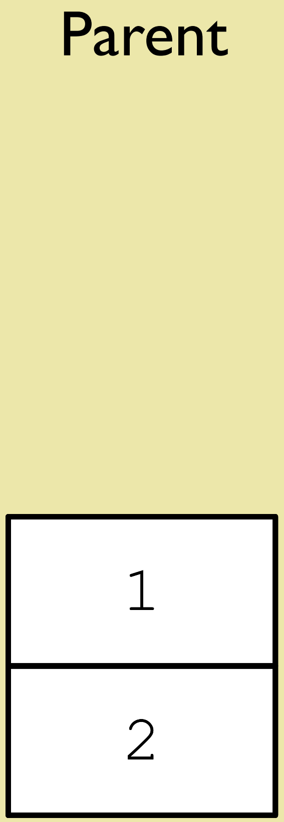
Threading

- All threads share the same address space but have to worry about concurrent usage

Forking

- Each child process gets its own address space
- Untouched memory setup by the parent is shared between the child processes

Forking: Copy on Write

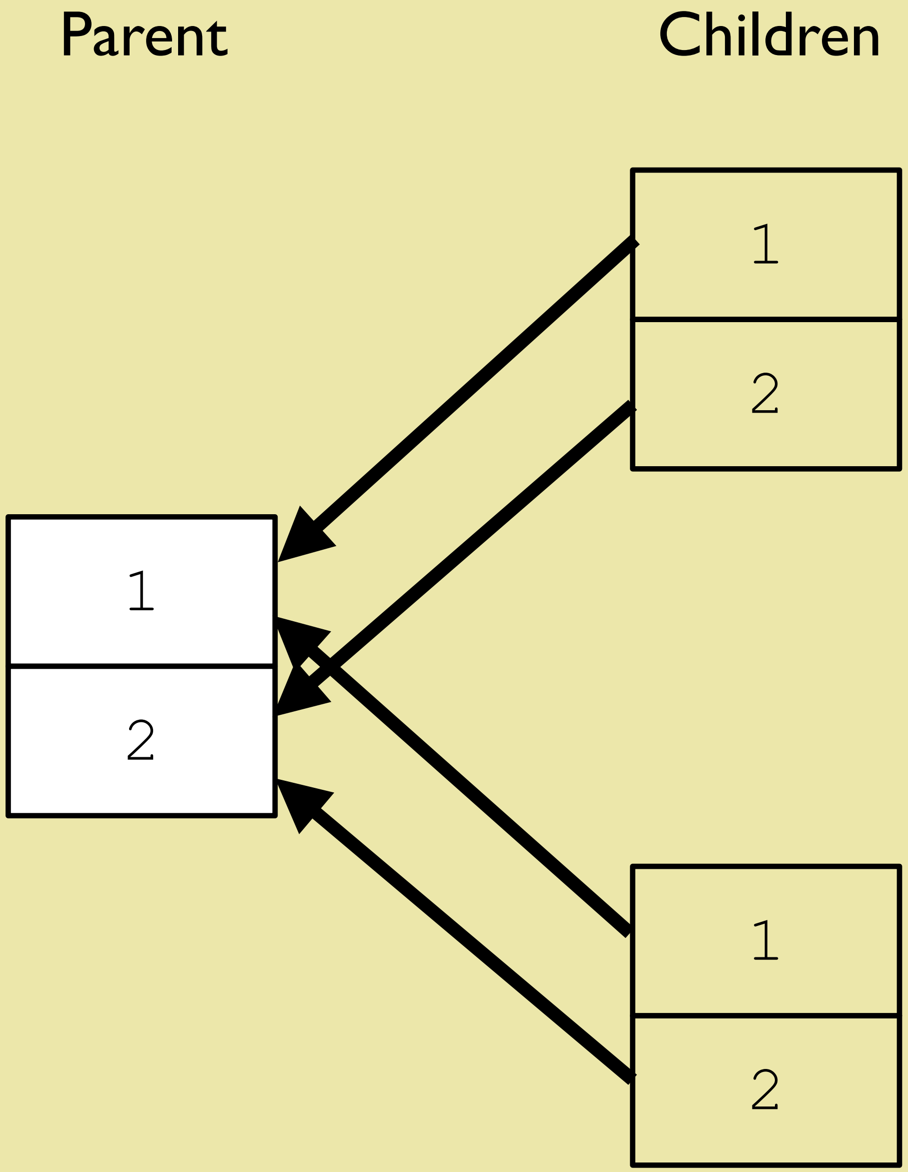


Parent

1

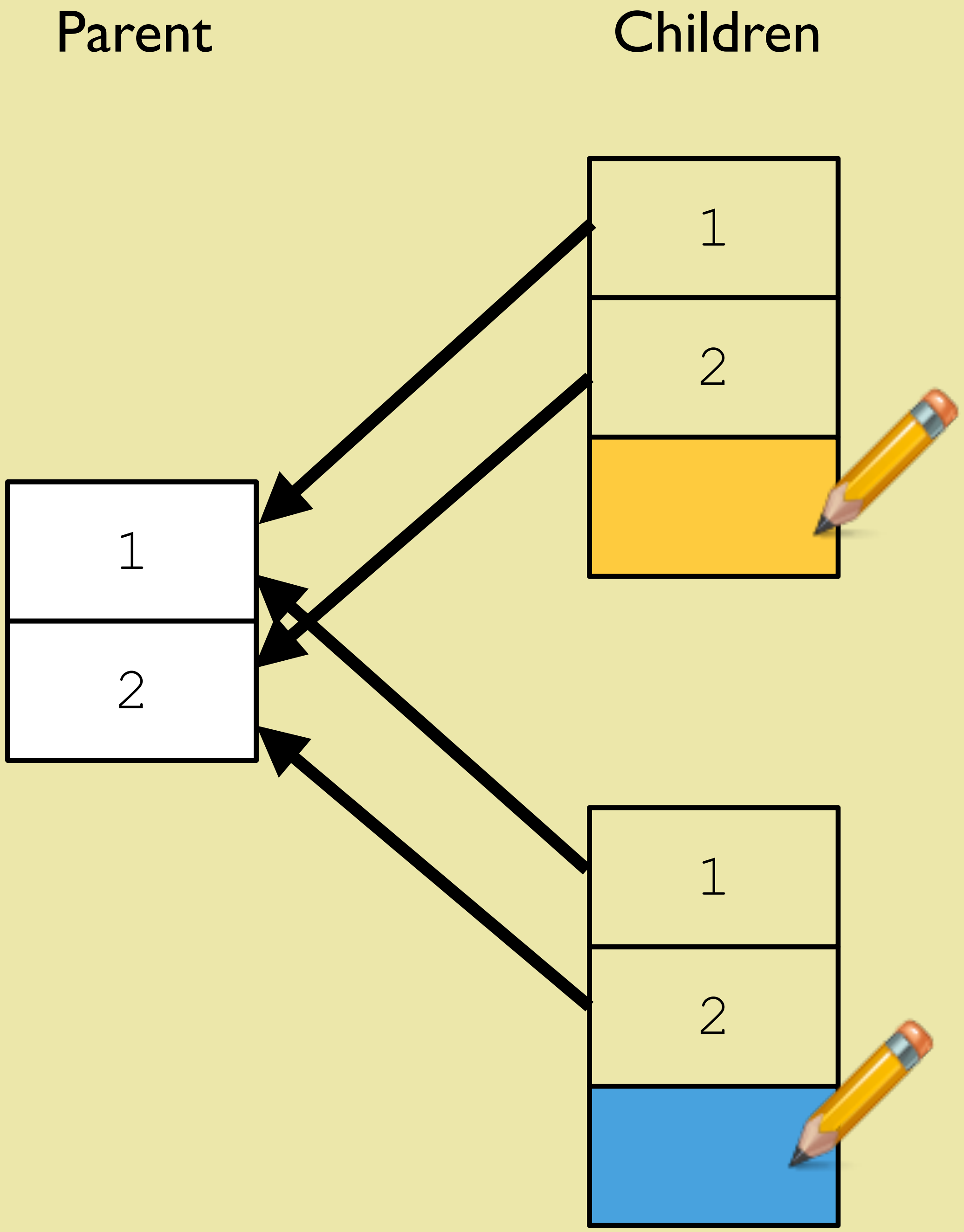
2

Forking: Copy on Write



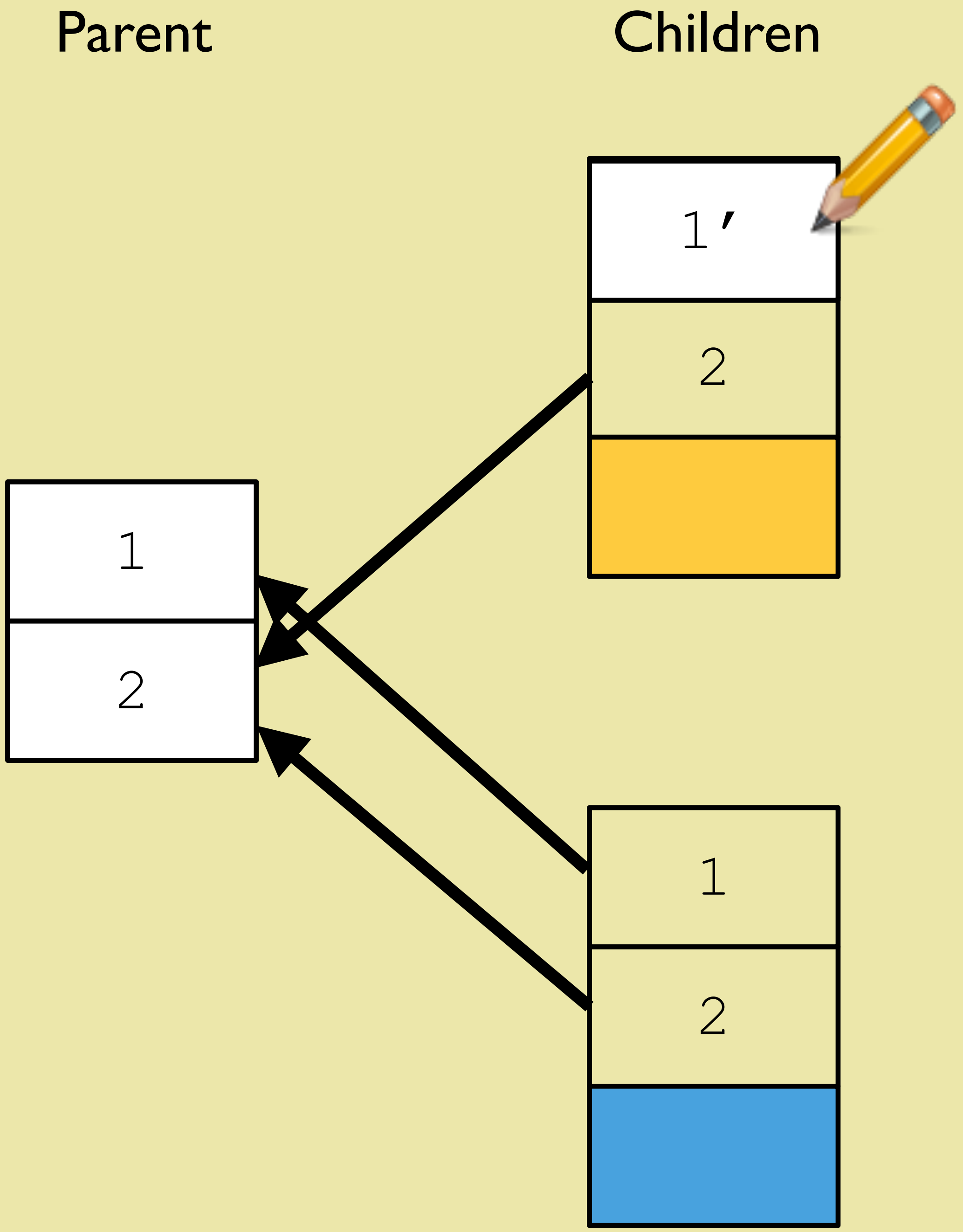
Child processes start by sharing the same memory pages as the parent

Forking: Copy on Write



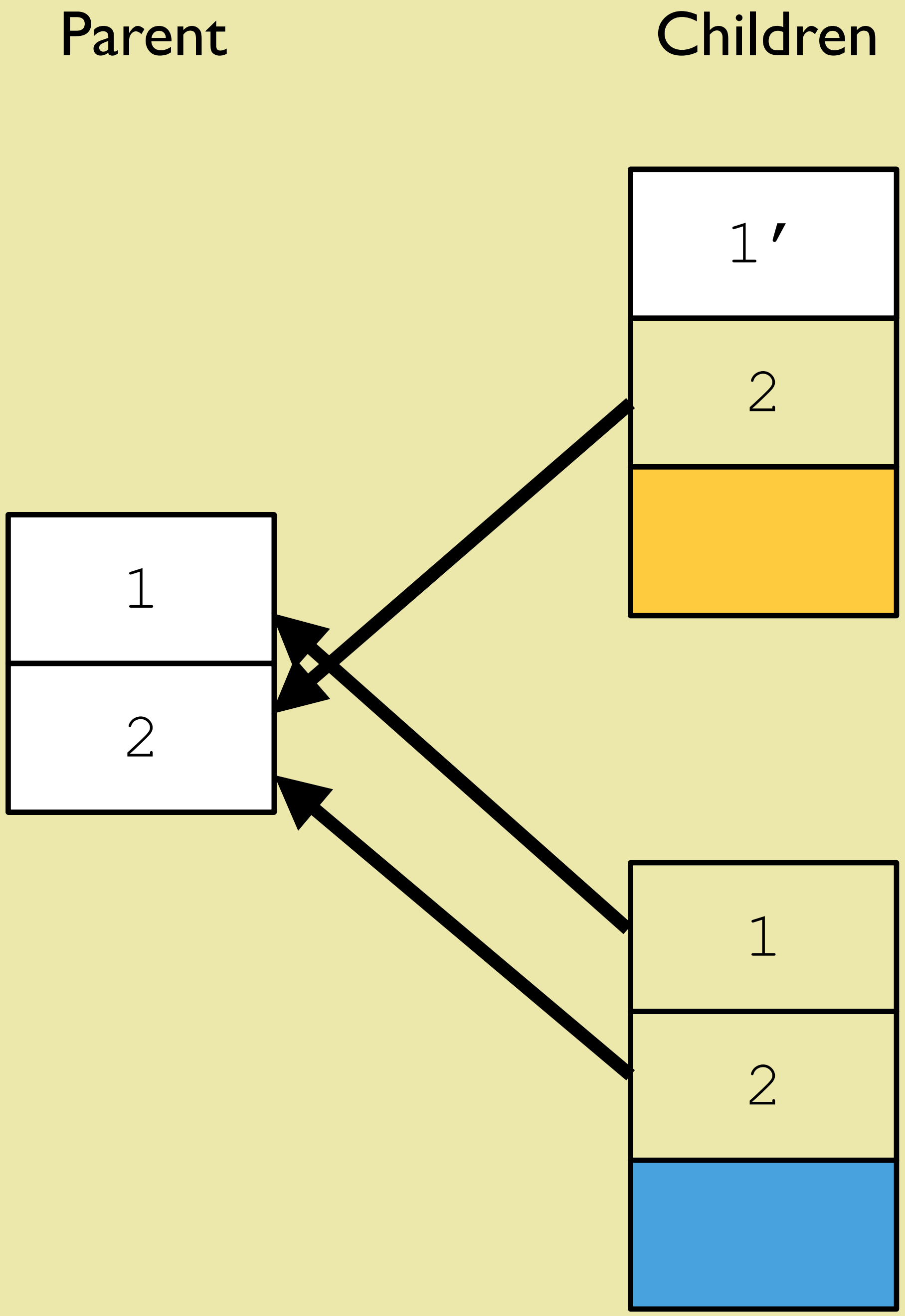
Children get their own pages when asking for new memory

Forking: Copy on Write



If a child attempts to write to shared memory, it gets its own copy

Forking: Copy on Write



Parent needs to load into memory often used, non-volatile data

Conditions, calibrations and geometry

Forking in CMS

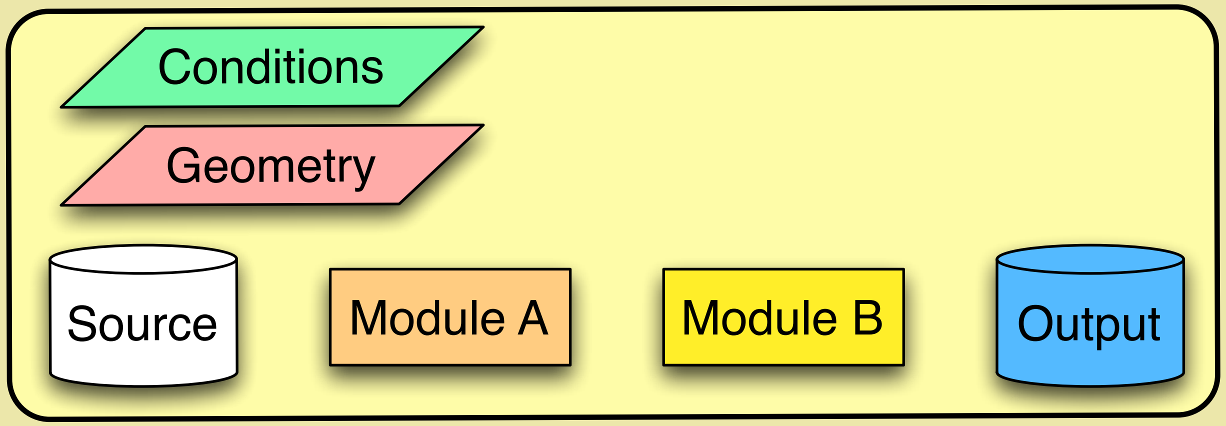
Parent

- Reads configuration and loads modules
  - Configuration says how many children and # events/child
- Opens input file and reads first run
  - modules are not called
- Pre-fetches conditions, calibrations and geometry
- Sends message to all modules that forking is going to happen
  - source closes file
- Forks

Forking in CMS

Parent

- Reads configuration and loads modules
  - Configuration says how many children and # events/child
- Opens input file and reads first run
  - modules are not called
- Pre-fetches conditions, calibrations and geometry
- Sends message to all modules that forking is going to happen
  - source closes file
- Forks



Forking in CMS

Parent

Reads configuration and loads modules

- Configuration says how many children and # events/child

Opens input file and reads first run

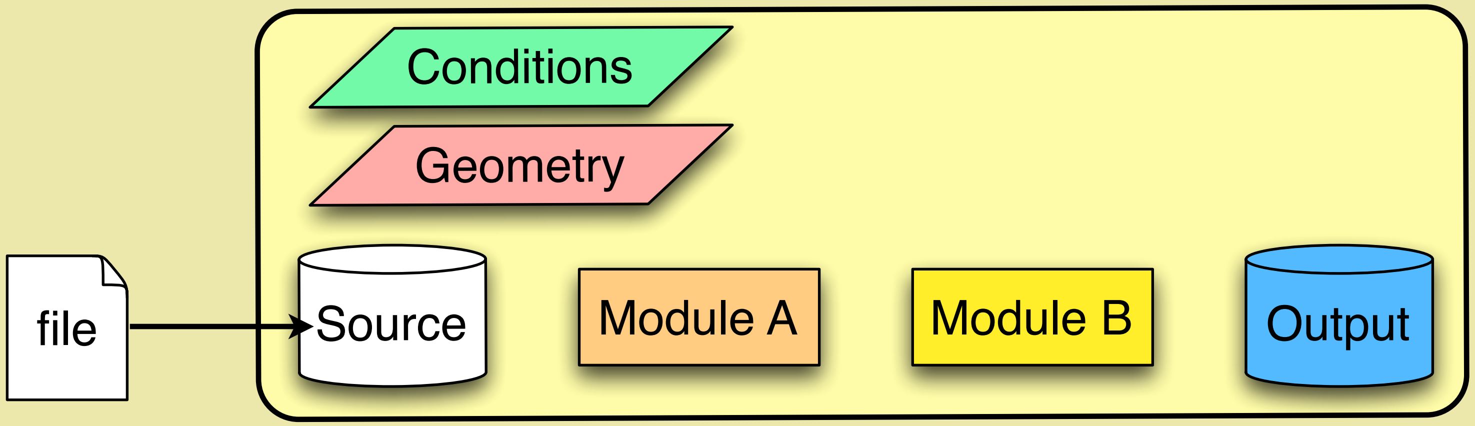
- modules are not called

Pre-fetches conditions, calibrations and geometry

Sends message to all modules that forking is going to happen

- source closes file

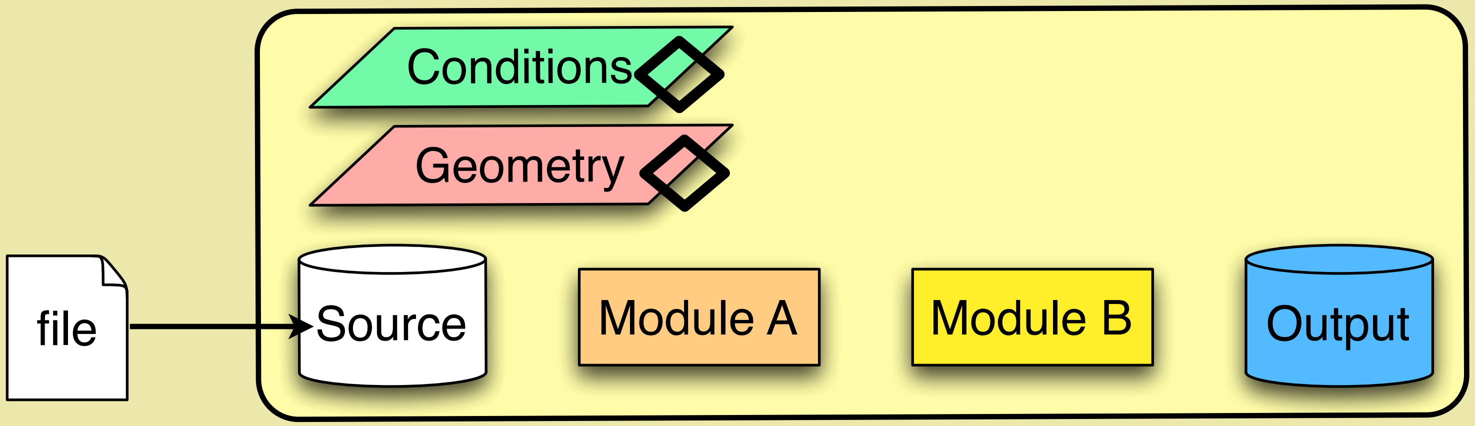
Forks



Forking in CMS

Parent

- Reads configuration and loads modules
  - Configuration says how many children and # events/child
- Opens input file and reads first run
  - modules are not called
- Pre-fetches conditions, calibrations and geometry
- Sends message to all modules that forking is going to happen
  - source closes file
- Forks



Forking in CMS

Parent

Reads configuration and loads modules

- Configuration says how many children and # events/child

Opens input file and reads first run

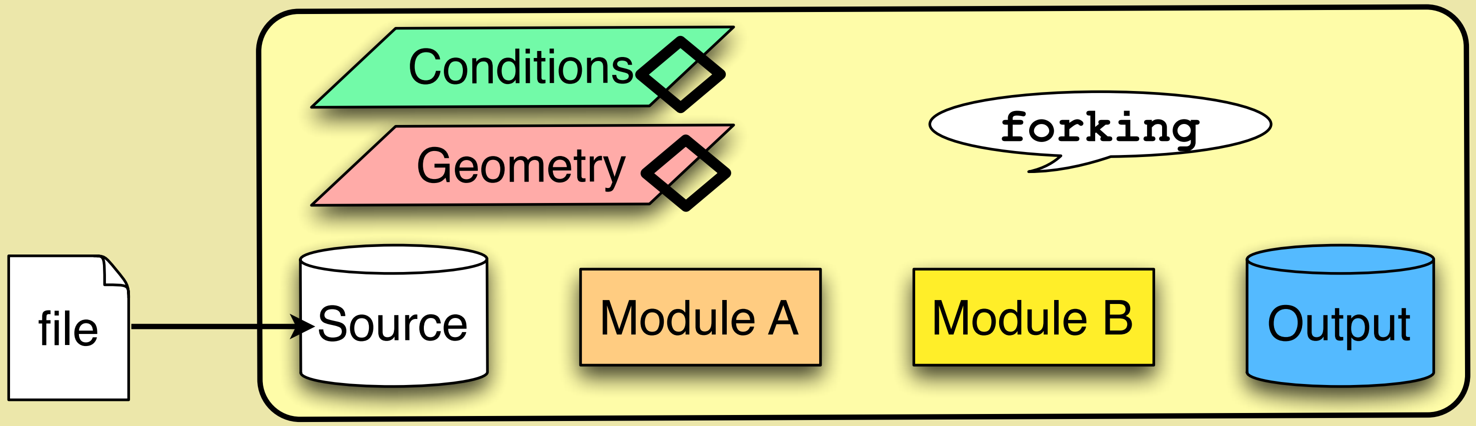
- modules are not called

Pre-fetches conditions, calibrations and geometry

Sends message to all modules that forking is going to happen

- source closes file

Forks



Forking in CMS

Parent

Reads configuration and loads modules

- Configuration says how many children and # events/child

Opens input file and reads first run

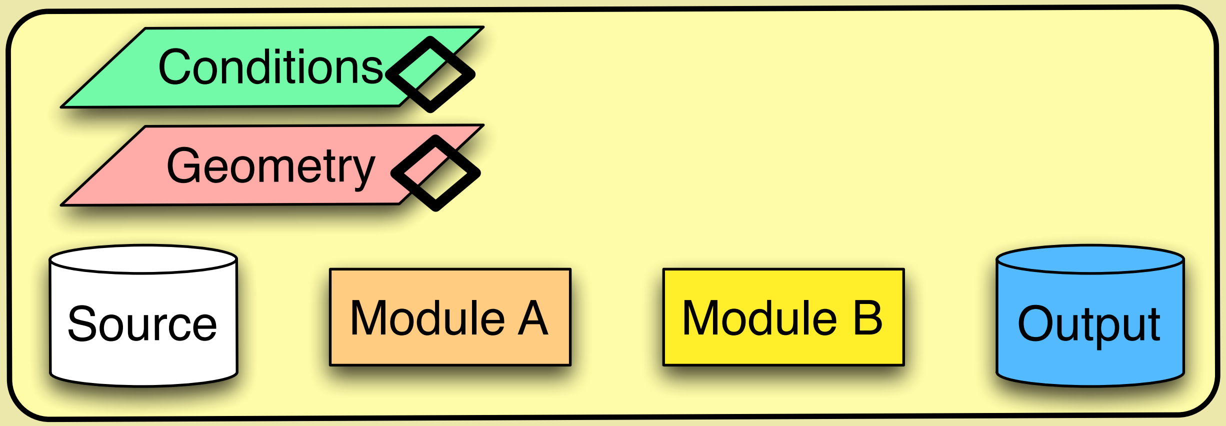
- modules are not called

Pre-fetches conditions, calibrations and geometry

Sends message to all modules that forking is going to happen

- source closes file

Forks



Forking in CMS

Parent

Reads configuration and loads modules

    Configuration says how many children and # events/child

Opens input file and reads first run

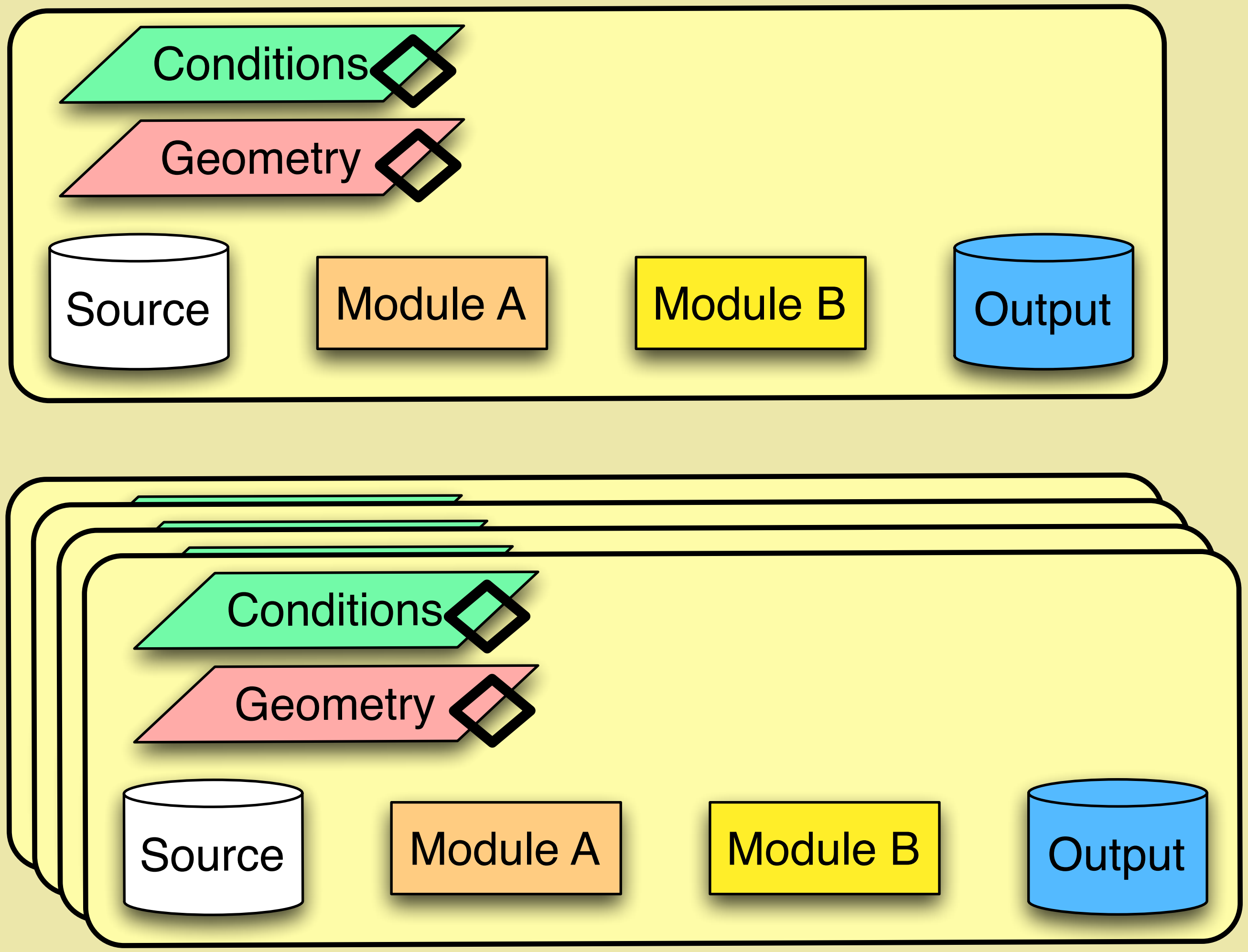
    modules are not called

Pre-fetches conditions, calibrations and geometry

Sends message to all modules that forking is going to happen

    source closes file

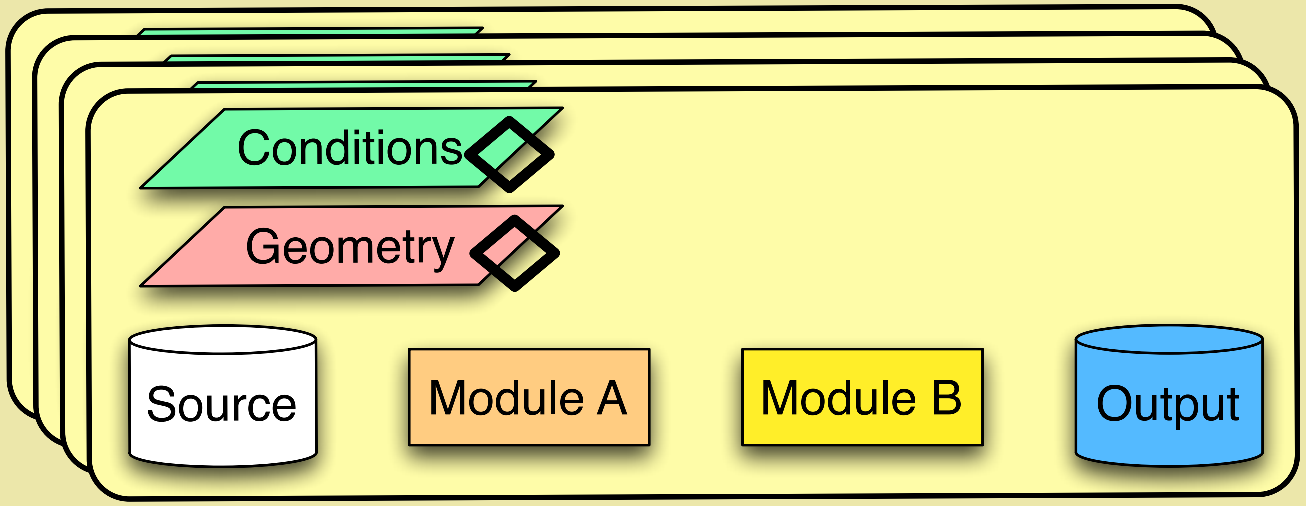
Forks



Forking in CMS (cont)

Children

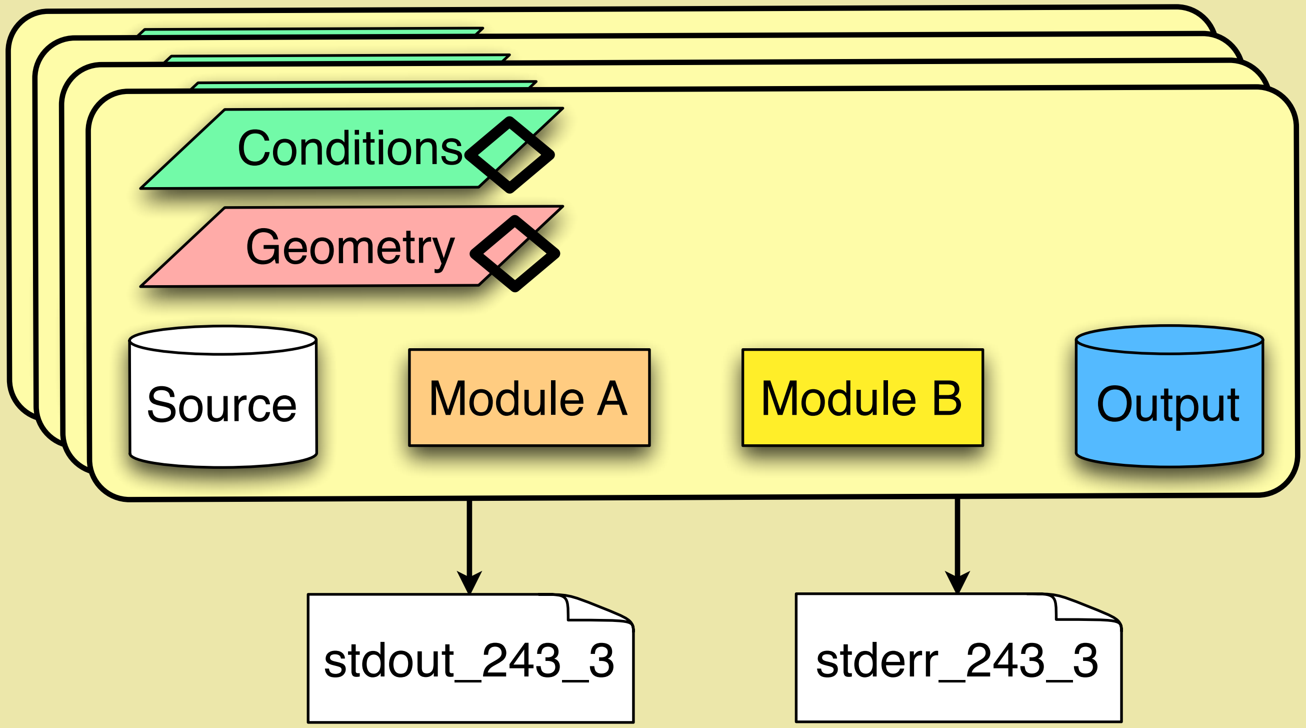
- Redirects stdout and stderr to own files whose names contain parent PID and child #
- Send messages to modules saying process is child X
  - *Output modules append child # to file names*
  - *Sources calculate their event ranges to process (no IP communication) and re-open the file*
- Process events in child's start/end range normally



Forking in CMS (cont)

Children

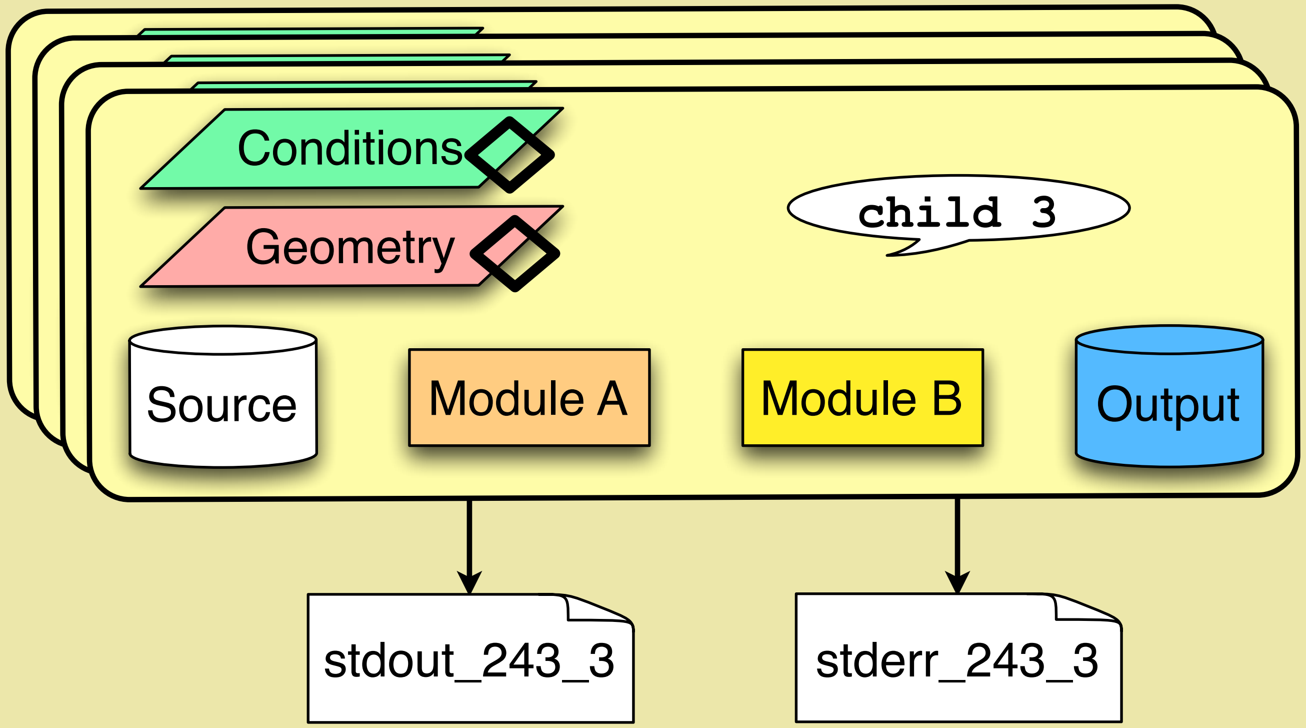
- Redirects stdout and stderr to own files whose names contain parent PID and child #
- Send messages to modules saying process is child X
  - Output modules append child # to file names
  - Sources calculate their event ranges to process (no IP communication) and re-open the file
- Process events in child's start/end range normally



Forking in CMS (cont)

Children

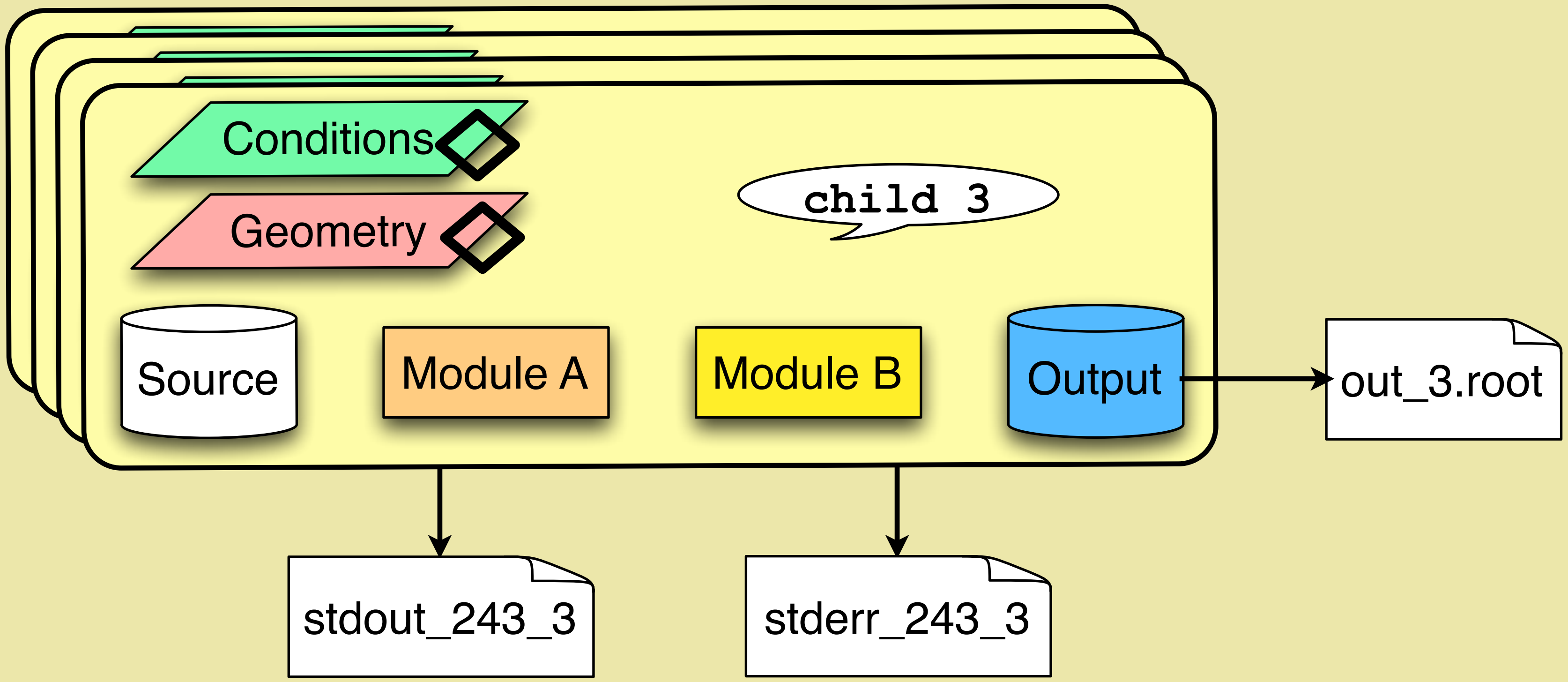
- Redirects stdout and stderr to own files whose names contain parent PID and child #
- Send messages to modules saying process is child X
  - Output modules append child # to file names
  - Sources calculate their event ranges to process (no IP communication) and re-open the file
- Process events in child's start/end range normally



Forking in CMS (cont)

Children

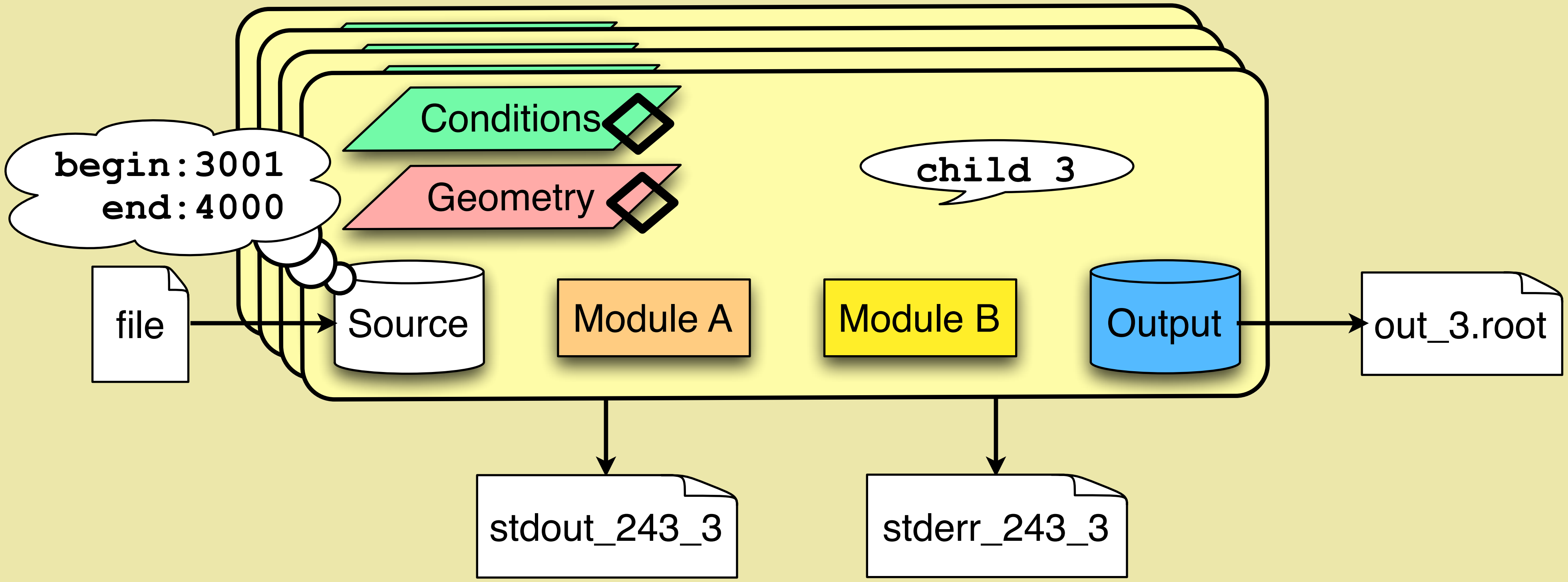
- Redirects stdout and stderr to own files whose names contain parent PID and child #
- Send messages to modules saying process is child X
  - Output modules append child # to file names
  - Sources calculate their event ranges to process (no IP communication) and re-open the file
- Process events in child's start/end range normally



Forking in CMS (cont)

Children

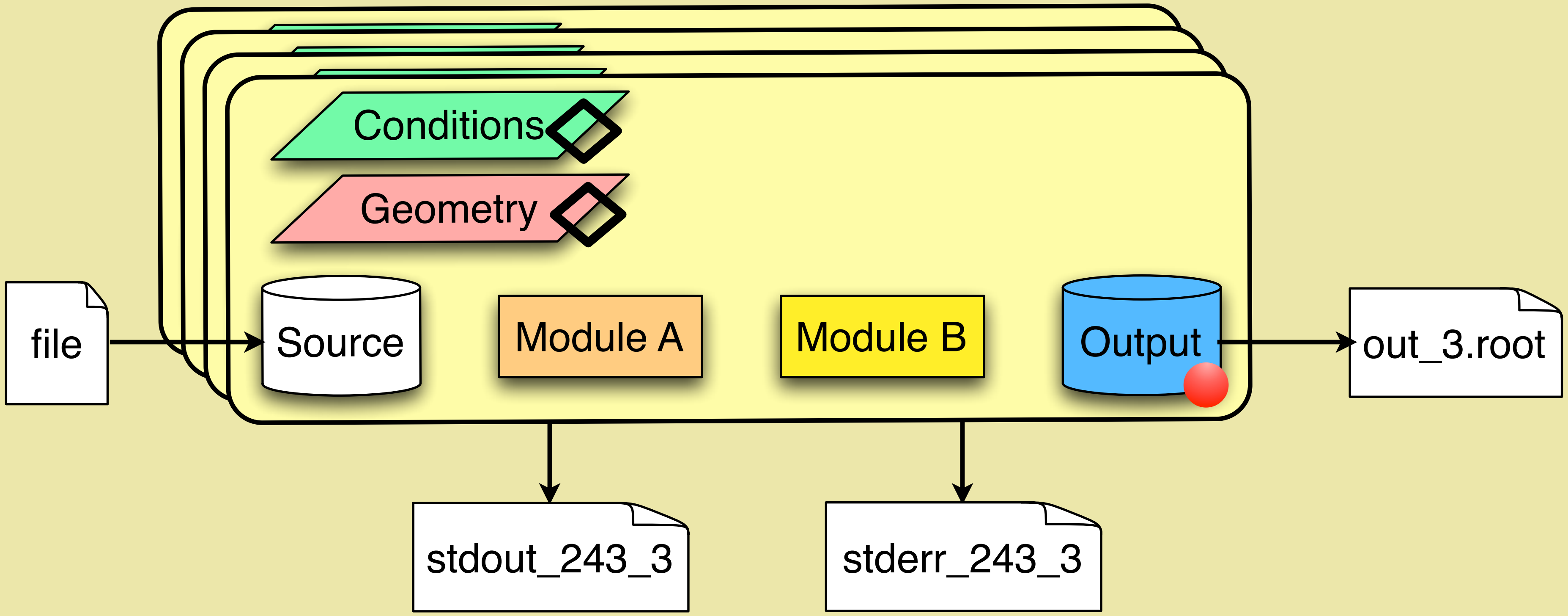
- Redirects stdout and stderr to own files whose names contain parent PID and child #
- Send messages to modules saying process is child X
  - Output modules append child # to file names
  - Sources calculate their event ranges to process (no IP communication) and re-open the file
- Process events in child's start/end range normally



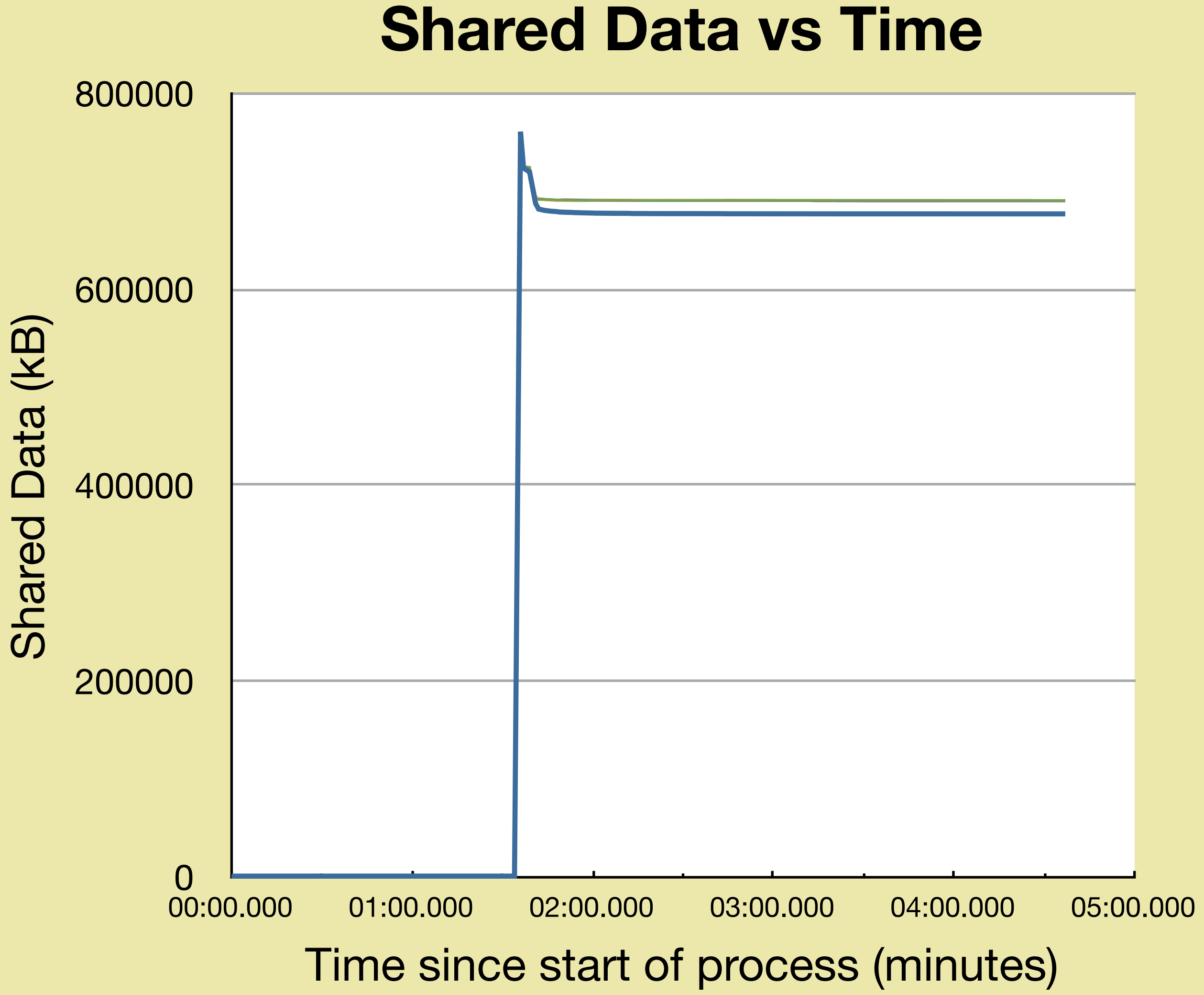
Forking in CMS (cont)

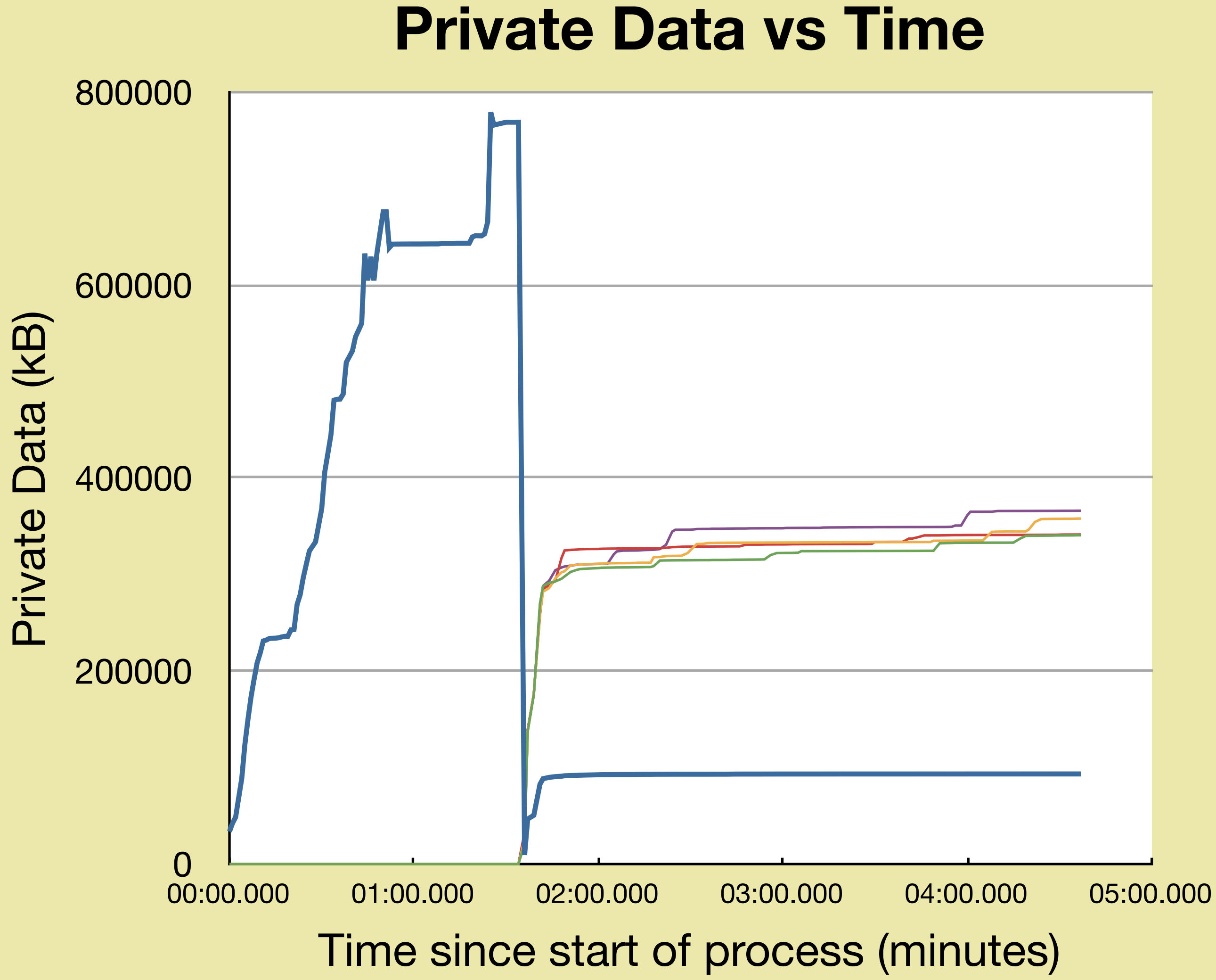
Children

- Redirects stdout and stderr to own files whose names contain parent PID and child #
- Send messages to modules saying process is child X
  - Output modules append child # to file names
  - Sources calculate their event ranges to process (no IP communication) and re-open the file
- Process events in child's start/end range normally



Memory Sharing



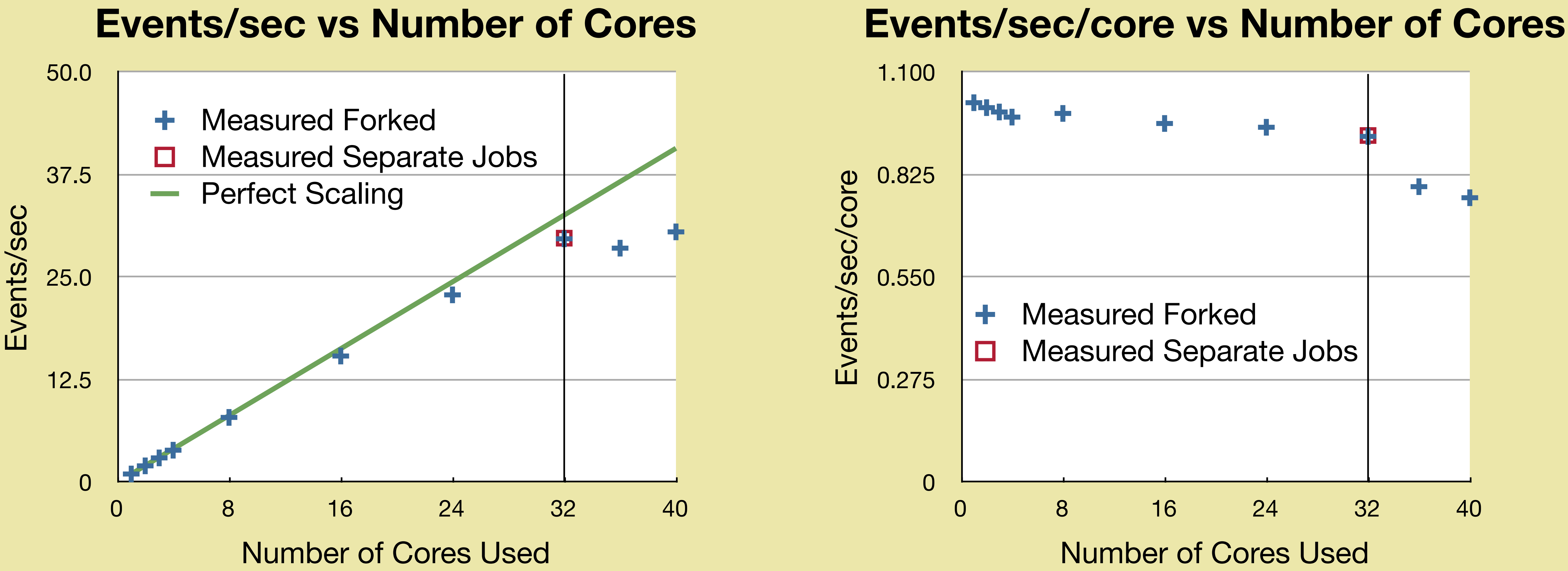


Measurements done using reconstruction with 64bit software on 4 CPU, 8 core/CPU 2GHz AMD Opteron(tm) Processor 6128

Shared memory per child: ~700MB
Private memory per child: ~375MB
Total memory used by 32 children: 13GB
Total memory used by 32 separate jobs: 34 GB

Saved 62% of memory

Throughput



Measurements done using reconstruction with 64bit software, raw data, reading and writing to local disk

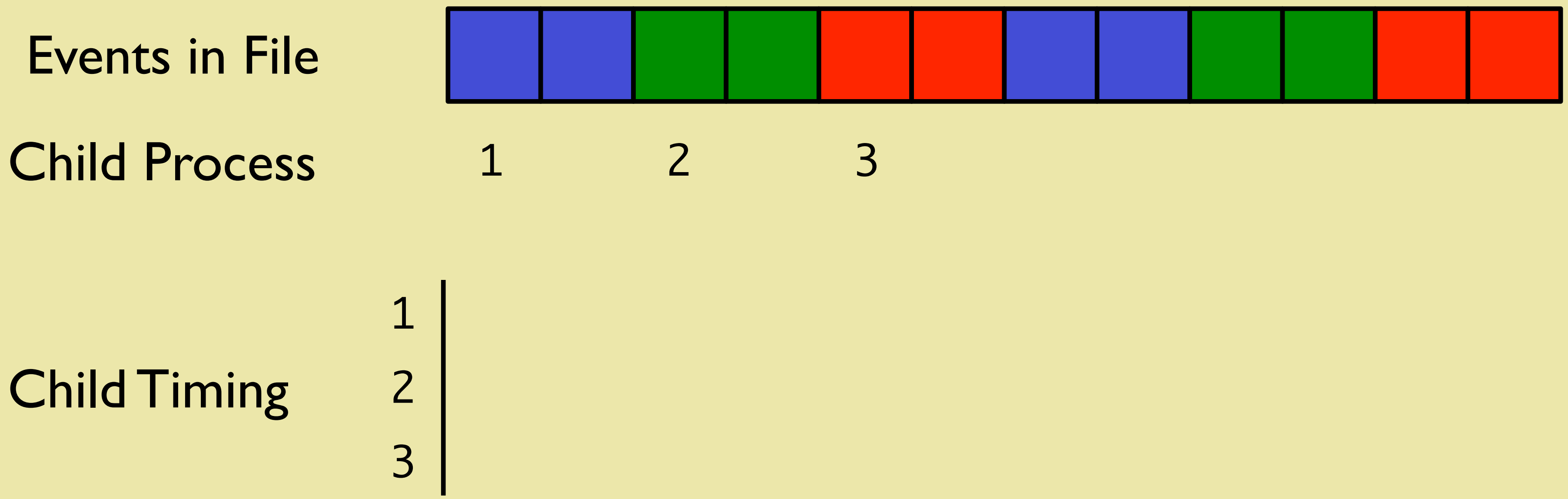
Measure total number events processed divided by the sum of the time taken by all cores

Ignores edge effects of startup and shutting down

Time Dispersion

The framework does pre-assigned round-robin event distribution with each child assigned N concurrent events to processes

E.g. with 3 children each told to do 2 concurrent events



The way events are distributed to the children affects how close in time all the children end work, i.e. dispersion

Measure dispersion by calculating **utilization**:

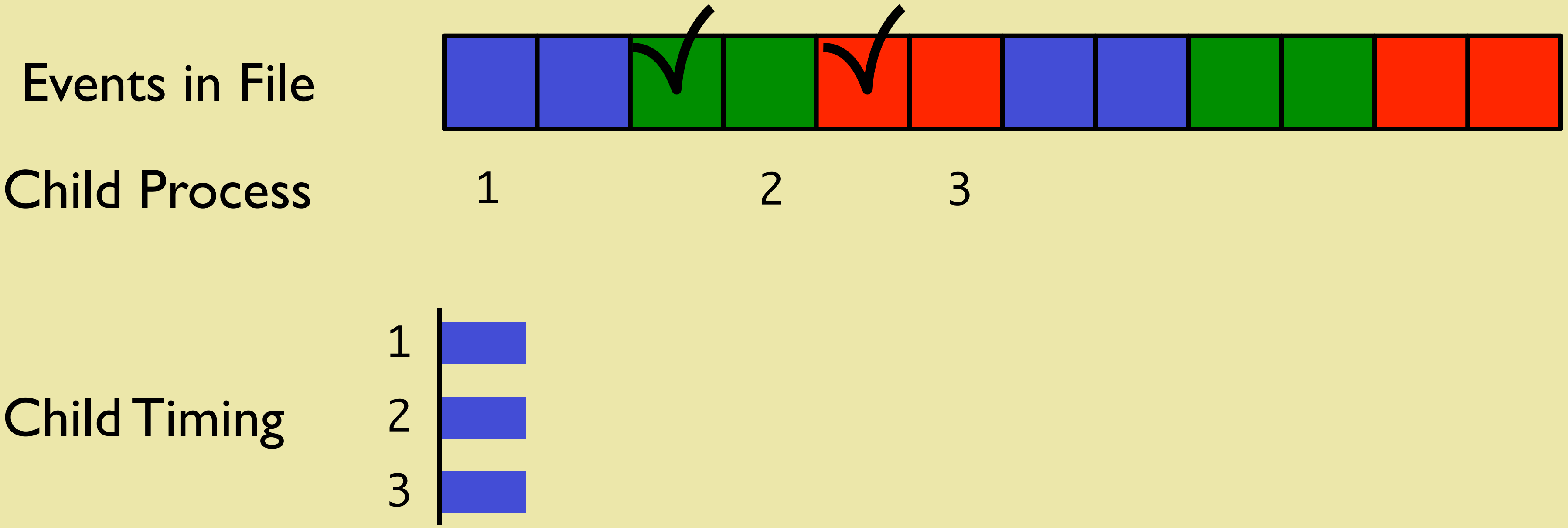
(Sum of children processing time)/(max child processing time)/(number of children)

equal to 1 when all children take the same amount of time

Time Dispersion

The framework does pre-assigned round-robin event distribution with each child assigned N concurrent events to processes

E.g. with 3 children each told to do 2 concurrent events



Events in File

Child Process

1

2

3

Child Timing

1

2

3

The way events are distributed to the children affects how close in time all the children end work, i.e. dispersion

Measure dispersion by calculating **utilization**:

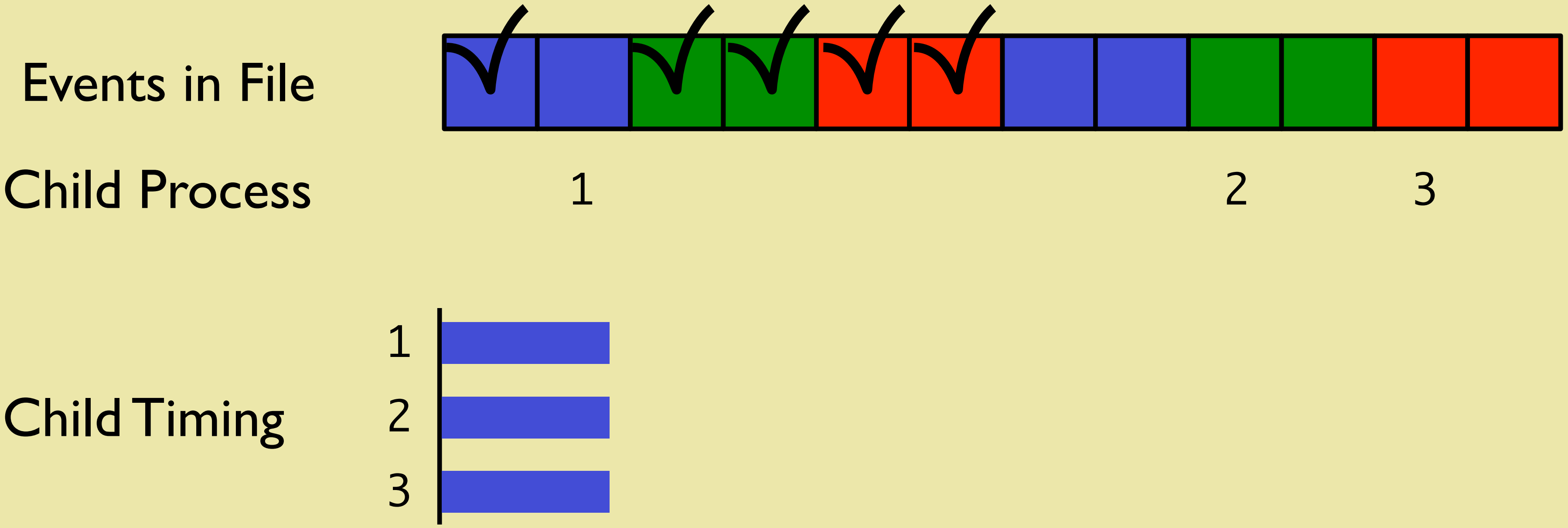
(Sum of children processing time)/(max child processing time)/(number of children)

equal to 1 when all children take the same amount of time

Time Dispersion

The framework does pre-assigned round-robin event distribution with each child assigned N concurrent events to processes

E.g. with 3 children each told to do 2 concurrent events



Events in File

Child Process

Child Timing

The way events are distributed to the children affects how close in time all the children end work, i.e. dispersion

Measure dispersion by calculating **utilization**:

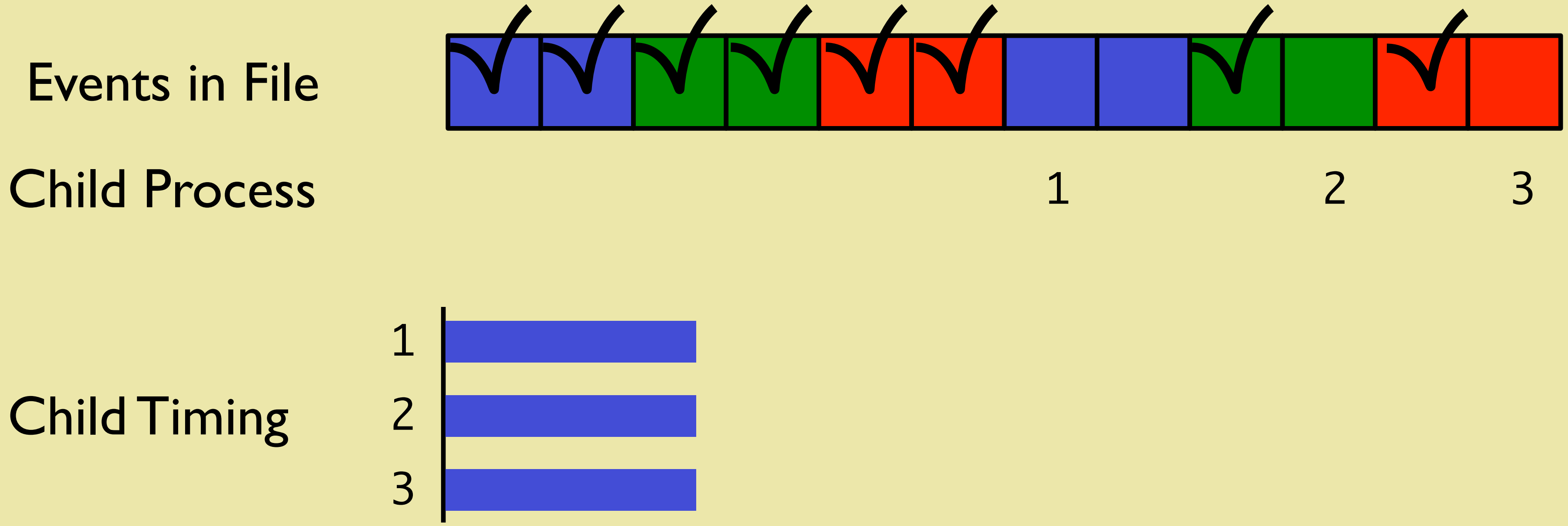
(Sum of children processing time)/(max child processing time)/(number of children)

equal to 1 when all children take the same amount of time

Time Dispersion

The framework does pre-assigned round-robin event distribution with each child assigned N concurrent events to processes

E.g. with 3 children each told to do 2 concurrent events



The way events are distributed to the children affects how close in time all the children end work, i.e. dispersion

Measure dispersion by calculating **utilization**:

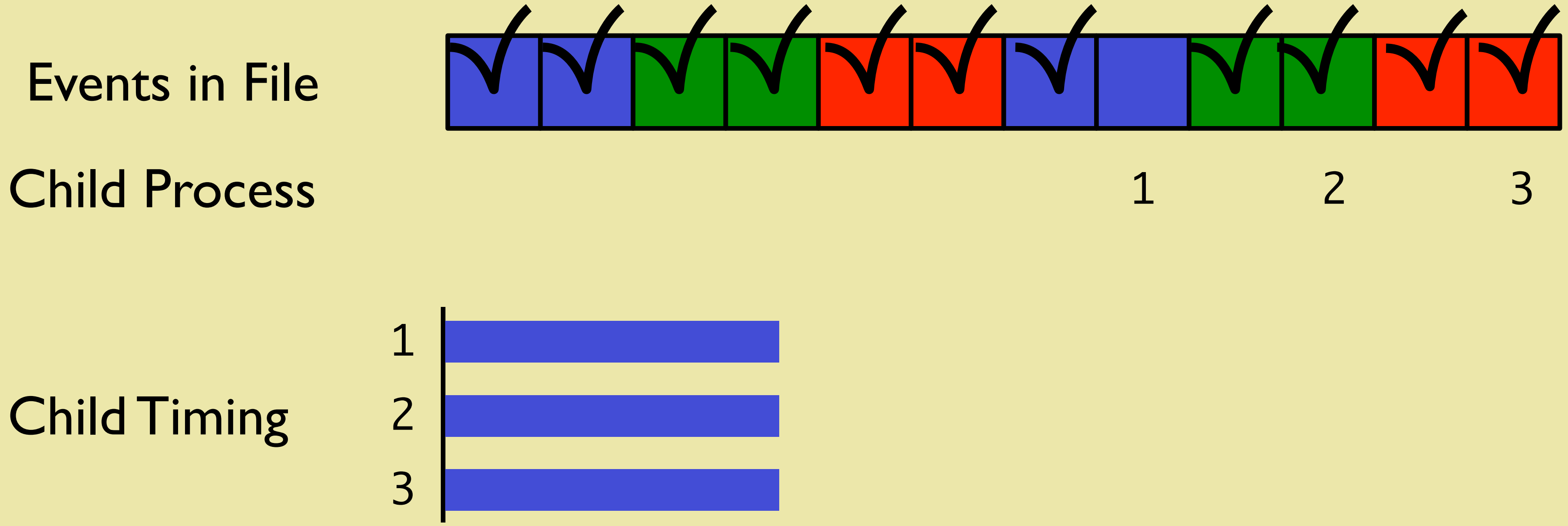
(Sum of children processing time)/(max child processing time)/(number of children)

equal to 1 when all children take the same amount of time

Time Dispersion

The framework does pre-assigned round-robin event distribution with each child assigned N concurrent events to processes

E.g. with 3 children each told to do 2 concurrent events



Events in File

Child Process

Child Timing

The way events are distributed to the children affects how close in time all the children end work, i.e. dispersion

Measure dispersion by calculating **utilization**:

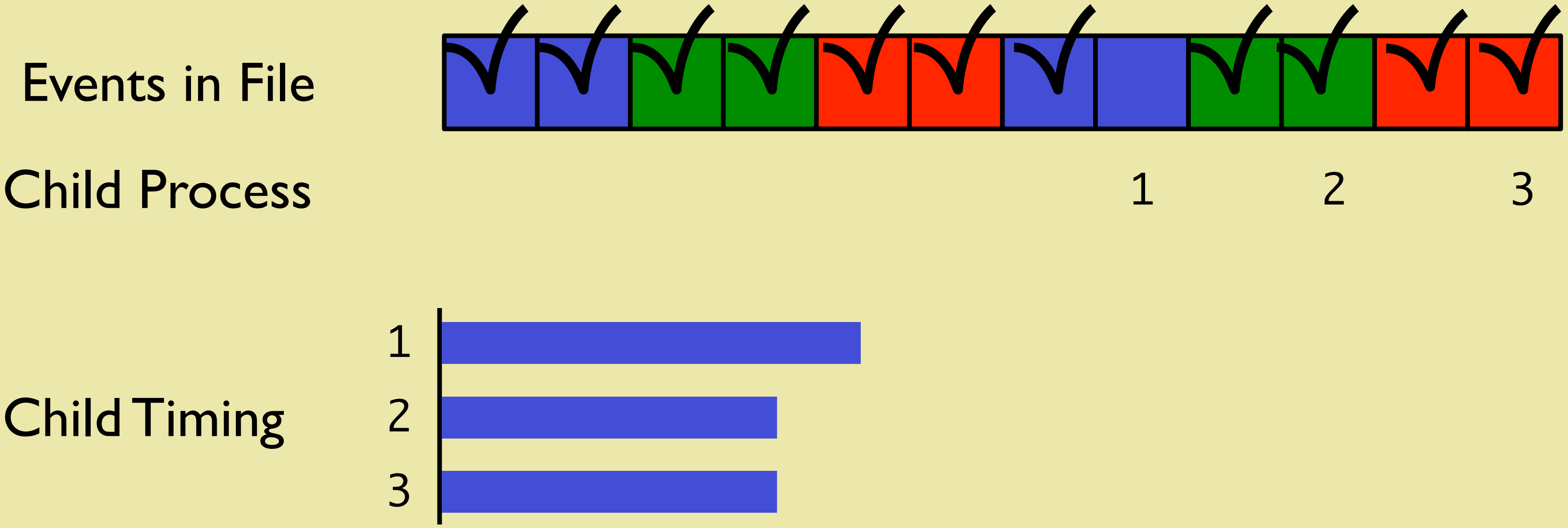
(Sum of children processing time)/(max child processing time)/(number of children)

equal to 1 when all children take the same amount of time

Time Dispersion

The framework does pre-assigned round-robin event distribution with each child assigned N concurrent events to processes

E.g. with 3 children each told to do 2 concurrent events



The way events are distributed to the children affects how close in time all the children end work, i.e. dispersion

Measure dispersion by calculating **utilization**:

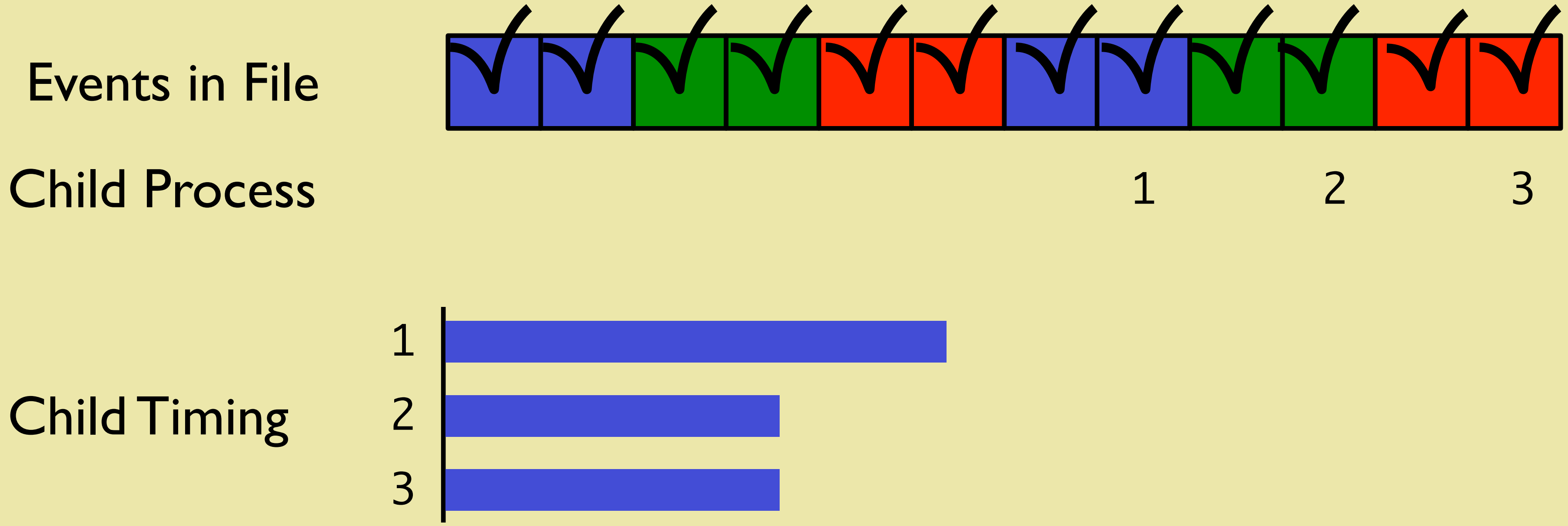
(Sum of children processing time)/(max child processing time)/(number of children)

equal to 1 when all children take the same amount of time

Time Dispersion

The framework does pre-assigned round-robin event distribution with each child assigned N concurrent events to processes

E.g. with 3 children each told to do 2 concurrent events



The way events are distributed to the children affects how close in time all the children end work, i.e. dispersion

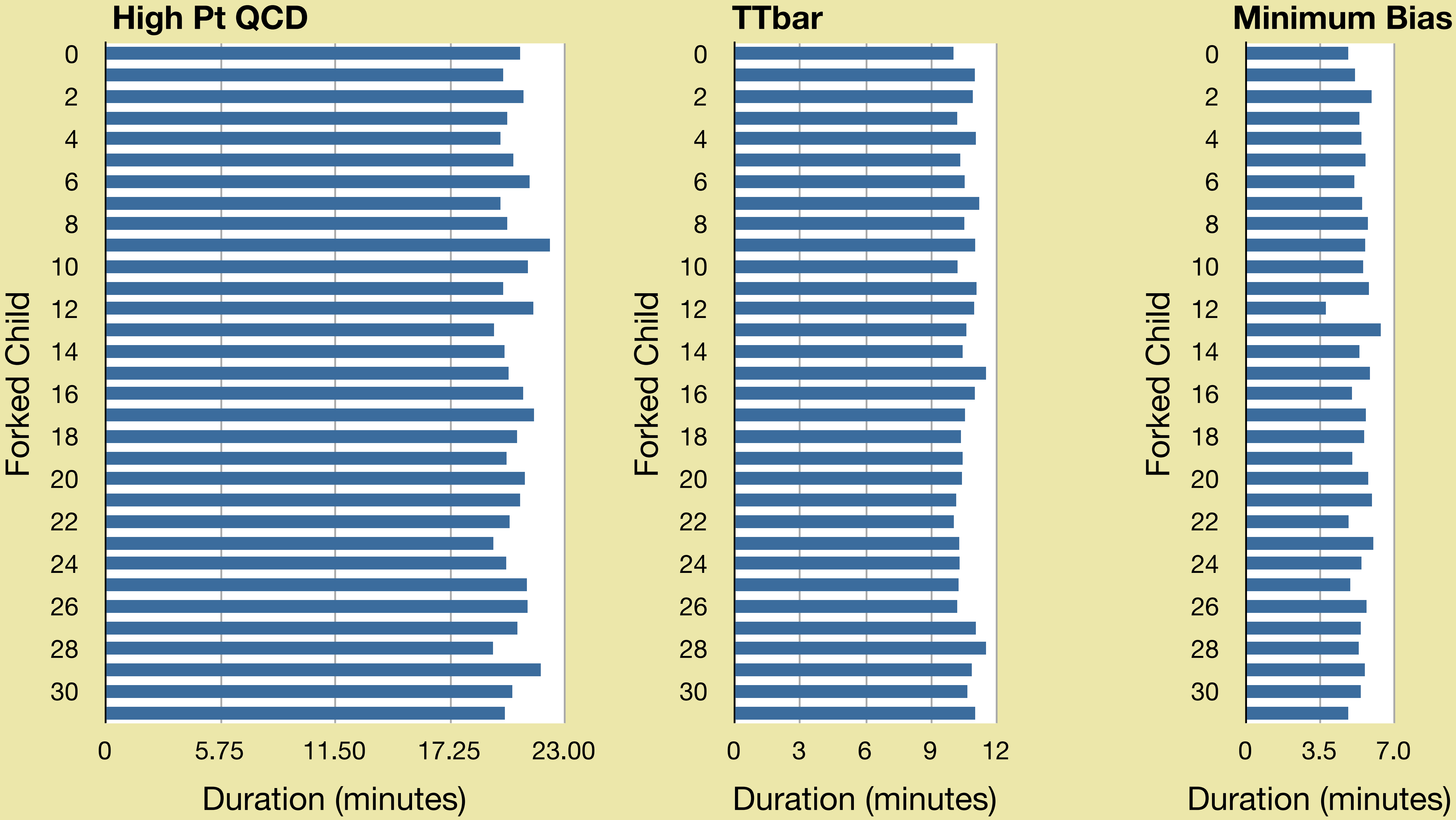
Measure dispersion by calculating **utilization**:

(Sum of children processing time)/(max child processing time)/(number of children)

equal to 1 when all children take the same amount of time

Time Dispersion MC

Duration of Forked Children For Reconstruction



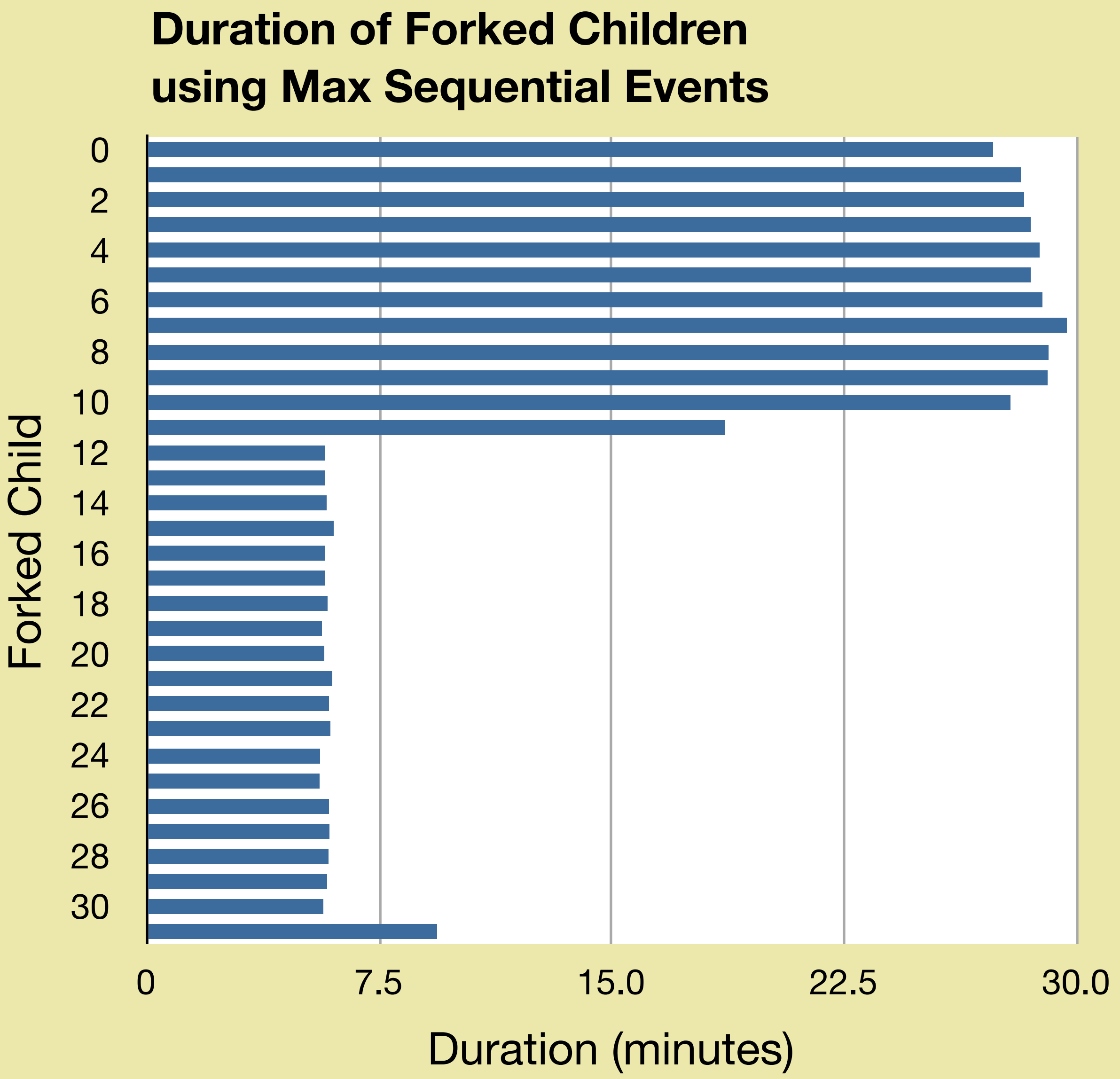
Reco MC works reasonably well with max sequential # events

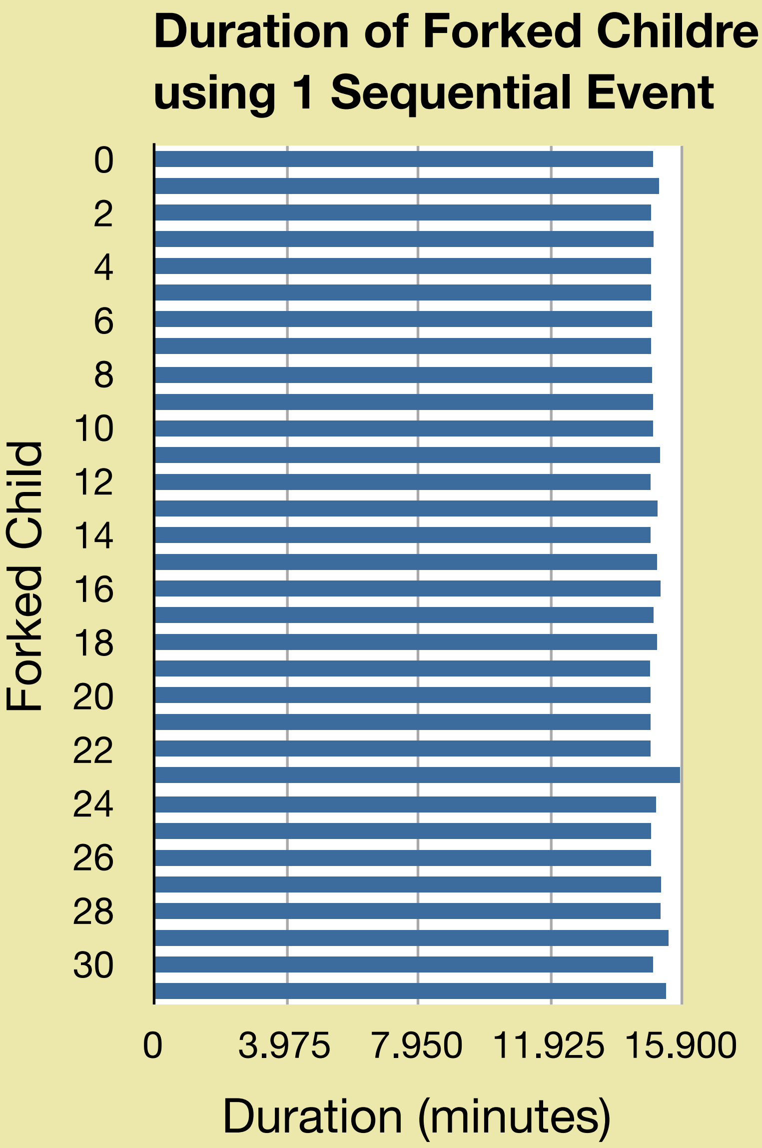
High Pt QCD utilization: 0.92

TTbar utilization: 0.92

Minimum bias utilization: 0.85

Time Dispersion





Some RAW files have ‘bigger’ events at beginning of file

The output files from the fast children are half the size of the files from the slow children

Utilization of max sequential events: 0.38

Utilization of 1 sequential event: 0.95

Event Distribution

Large *N* (concurrent events) helps with input efficiency

- ROOT has a read-ahead cache so large *N* means more cache hits
- Measurement with ***N* = max** with 32 children
  - Average total read operations/sec after startup: 6.2 ops/s
  - Average read size after startup: 600kB
- Measurement with ***N* = 1** with 32 children
  - Average total read operations/sec after startup: 156 ops/s
  - Average read size after startup: 25kB
- Minimum *N* should probably be around (cache size)/(average event size in the file)

Max size *N* (#events/#cores) aids in merging of result file

- Can merge output files in order and get back event ordering of original input file

Large *N* can lead to large dispersion if event characteristics change over the length of the input file (i.e. with time)

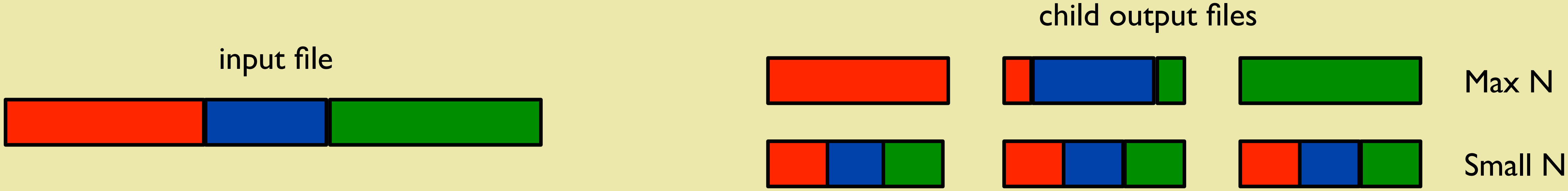
Read Performance

How reco file is merged affects input performance of analysis

- Merge step always takes the same amount of time regardless of event ordering
- Output files are just concatenated together

Reason: framework always 'plays' events from same luminosity section contiguously

- Luminosity section is created every 23 seconds during data taking
- If events from same lumi are scattered through a file, file reads will be 'random'
  - RECO with max consecutive events read-via-cache: 5900 MB
  - RECO with 1 consecutive event read-via-cache: 750 MB



NOTE: color denotes luminosity section

Optimum is to keep events from same Lumi together

- Order of events in Lumi does not matter

Planned solution:

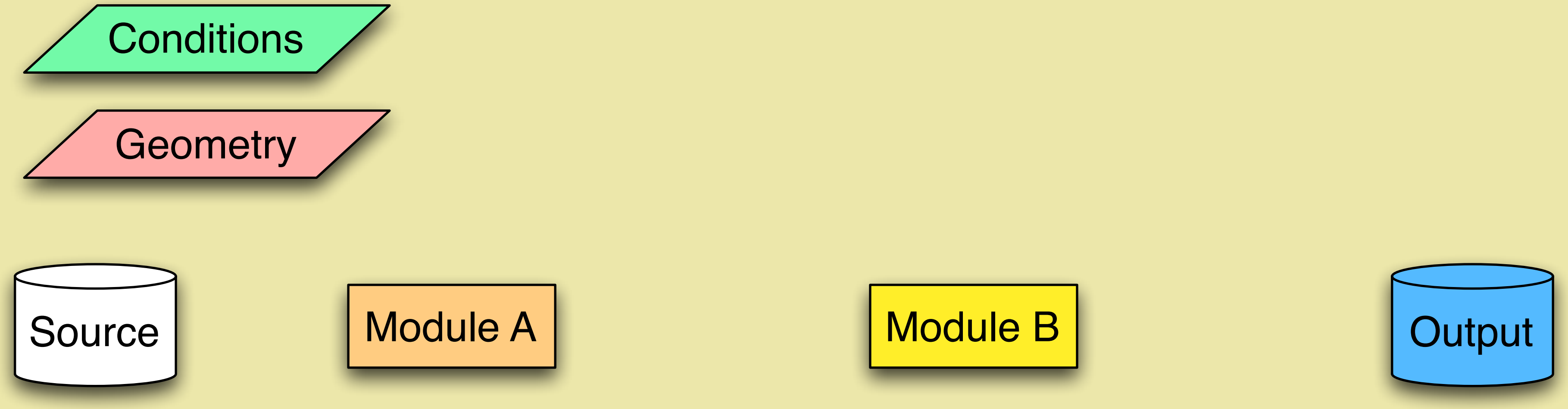
- Each child writes one temporary file per lumi section
- Merge job reads the temporary files in lumi section order

Threading

Threading an application should allow greater event throughput with nearly the same memory footprint

Simplest change would be to run modules in parallel

Assumes Module A not dependent on results from Module B

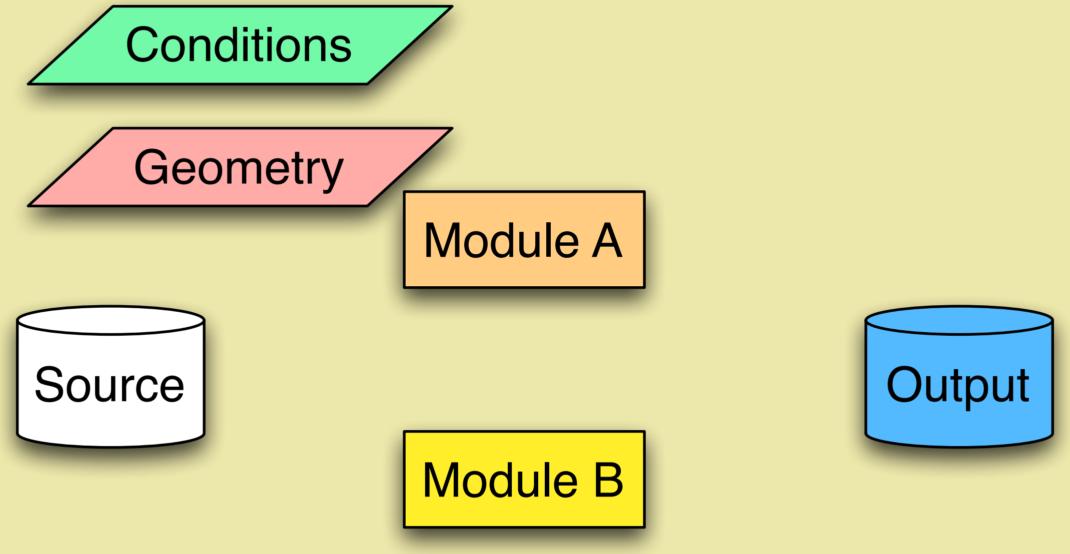


Threading

Threading an application should allow greater event throughput with nearly the same memory footprint

Simplest change would be to run modules in parallel

Assumes Module A not dependent on results from Module B



Threading Estimates

Can estimate performance benefits from threading if know

- Average time each module takes to process an event
- What modules create data used by another module
  - Already recorded by framework

Calculation

- Sort module's by start time
  - end time of last to stop dependent module
- Last module's end time is estimated parallel processing time for 1 event
- Gives # of concurrently running modules per time period

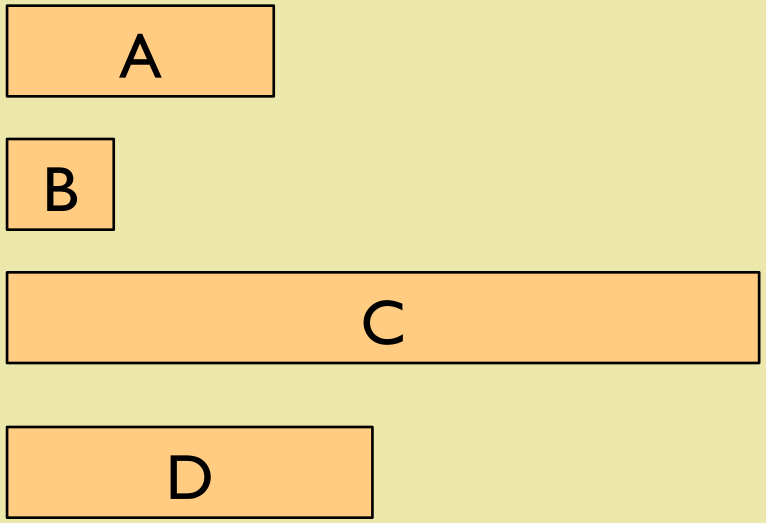
Threading Estimates

Can estimate performance benefits from threading if know

- Average time each module takes to process an event
- What modules create data used by another module
    - Already recorded by framework

Calculation

- Sort module's by start time
    - end time of last to stop dependent module
- Last module's end time is estimated parallel processing time for 1 event
- Gives # of concurrently running modules per time period



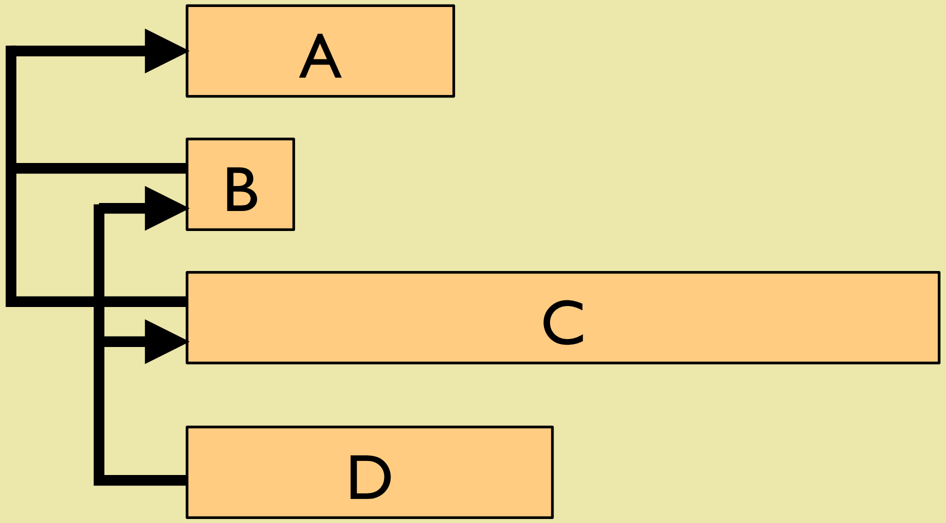
Threading Estimates

Can estimate performance benefits from threading if know

- Average time each module takes to process an event
- What modules create data used by another module
  - Already recorded by framework

Calculation

- Sort module's by start time
  - end time of last to stop dependent module
- Last module's end time is estimated parallel processing time for 1 event
- Gives # of concurrently running modules per time period



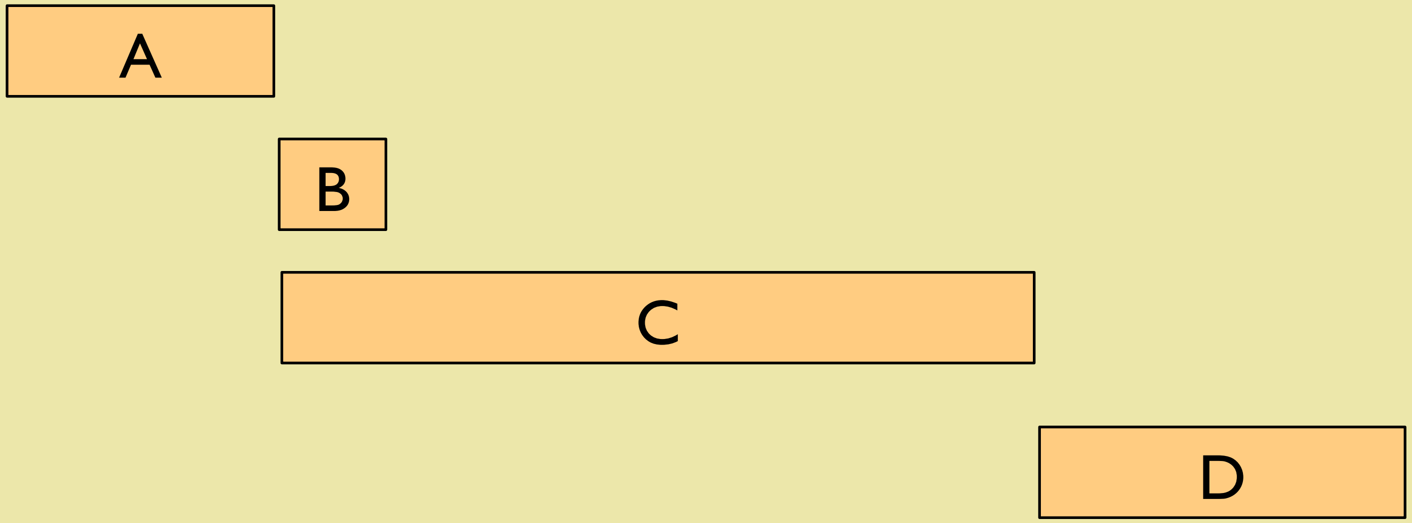
Threading Estimates

Can estimate performance benefits from threading if know

- Average time each module takes to process an event
- What modules create data used by another module
  - Already recorded by framework

Calculation

- Sort module's by start time
  - end time of last to stop dependent module
- Last module's end time is estimated parallel processing time for 1 event
- Gives # of concurrently running modules per time period



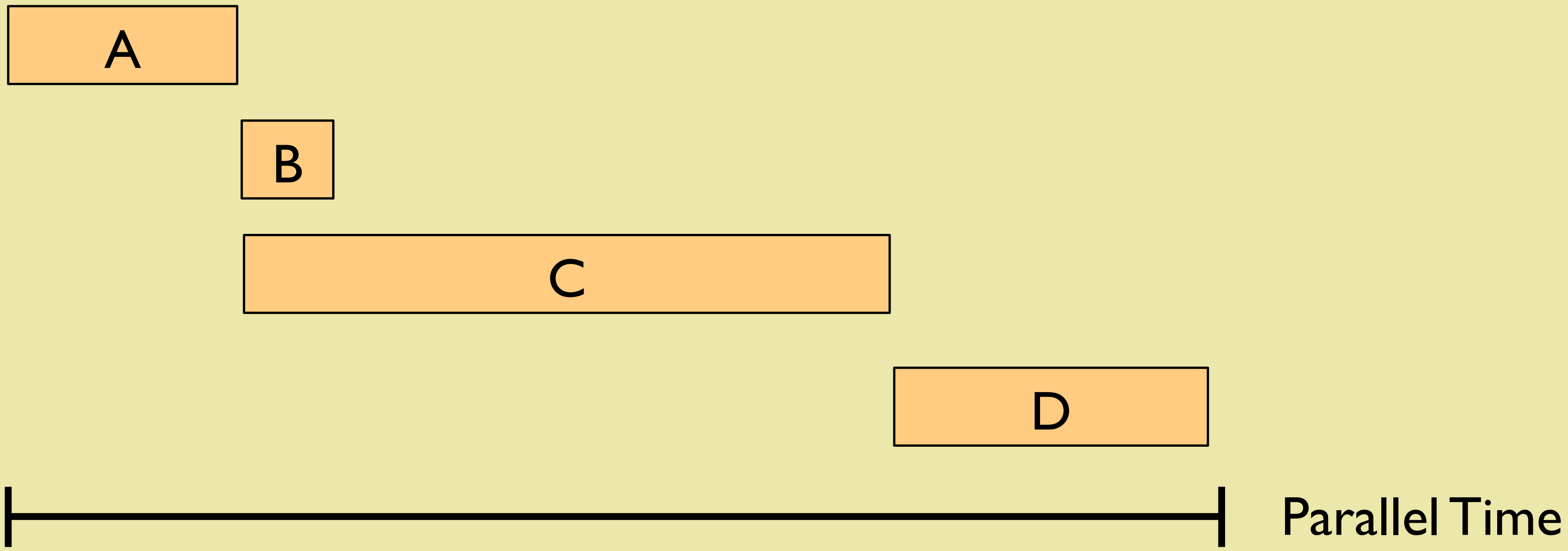
Threading Estimates

Can estimate performance benefits from threading if know

- Average time each module takes to process an event
- What modules create data used by another module
  - Already recorded by framework

Calculation

- Sort module's by start time
  - end time of last to stop dependent module
- Last module's end time is estimated parallel processing time for 1 event
- Gives # of concurrently running modules per time period



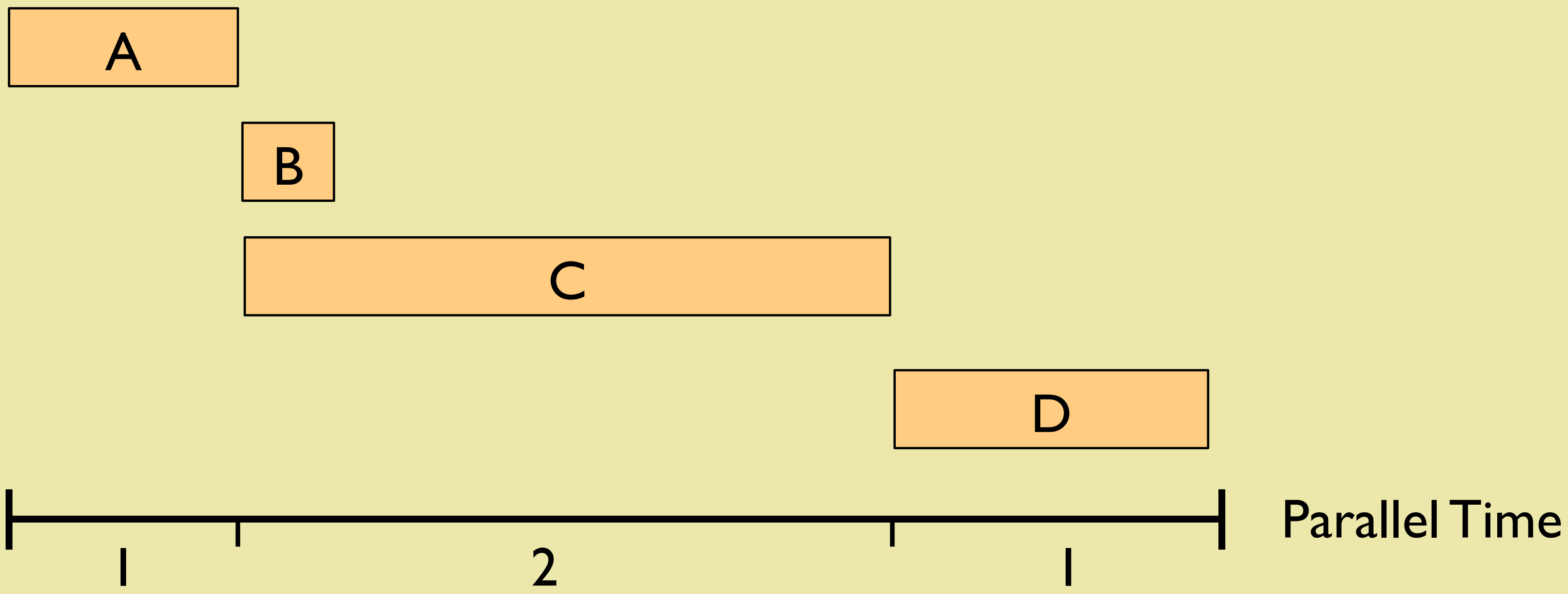
Threading Estimates

Can estimate performance benefits from threading if know

- Average time each module takes to process an event
- What modules create data used by another module
  - Already recorded by framework

Calculation

- Sort module's by start time
  - end time of last to stop dependent module
- Last module's end time is estimated parallel processing time for 1 event
- Gives # of concurrently running modules per time period



Reco Samples Used

Minimum Bias

Quickest to processes and most prevalent

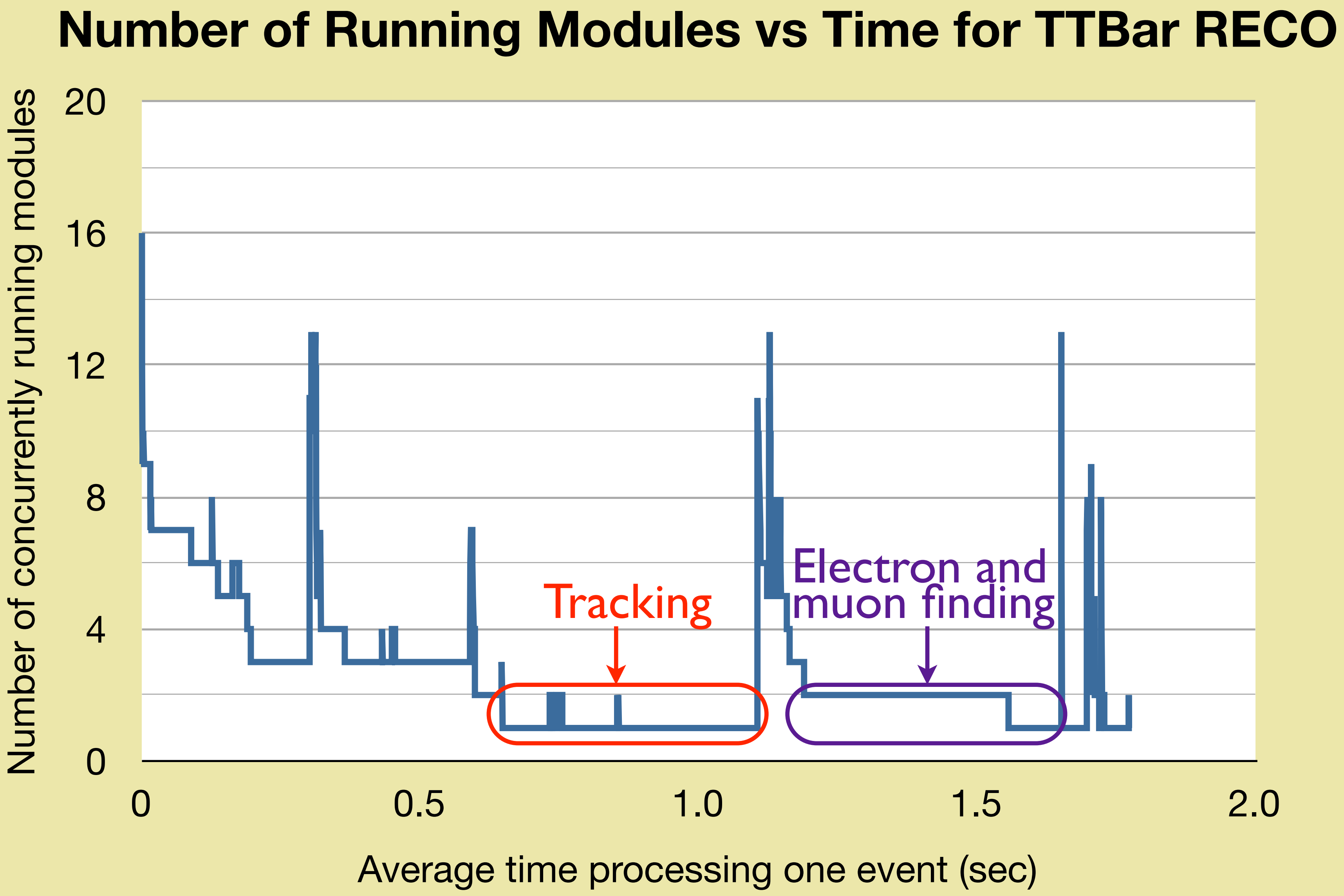
T Tbar

Middle of the road complexity

High Pt (3.0 - 3.5TeV) QCD

Most complex and slowest to process

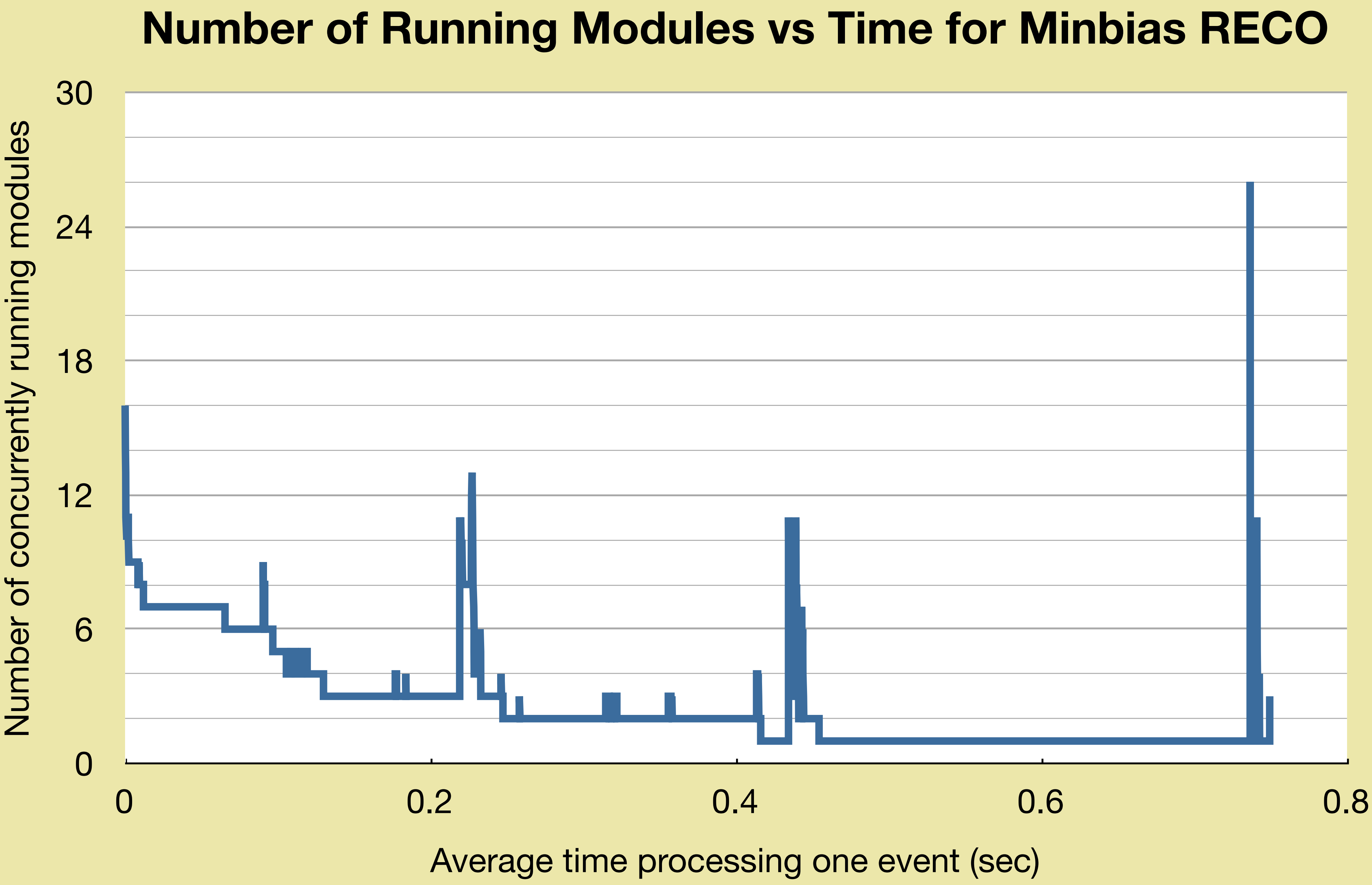
TTbar Estimates



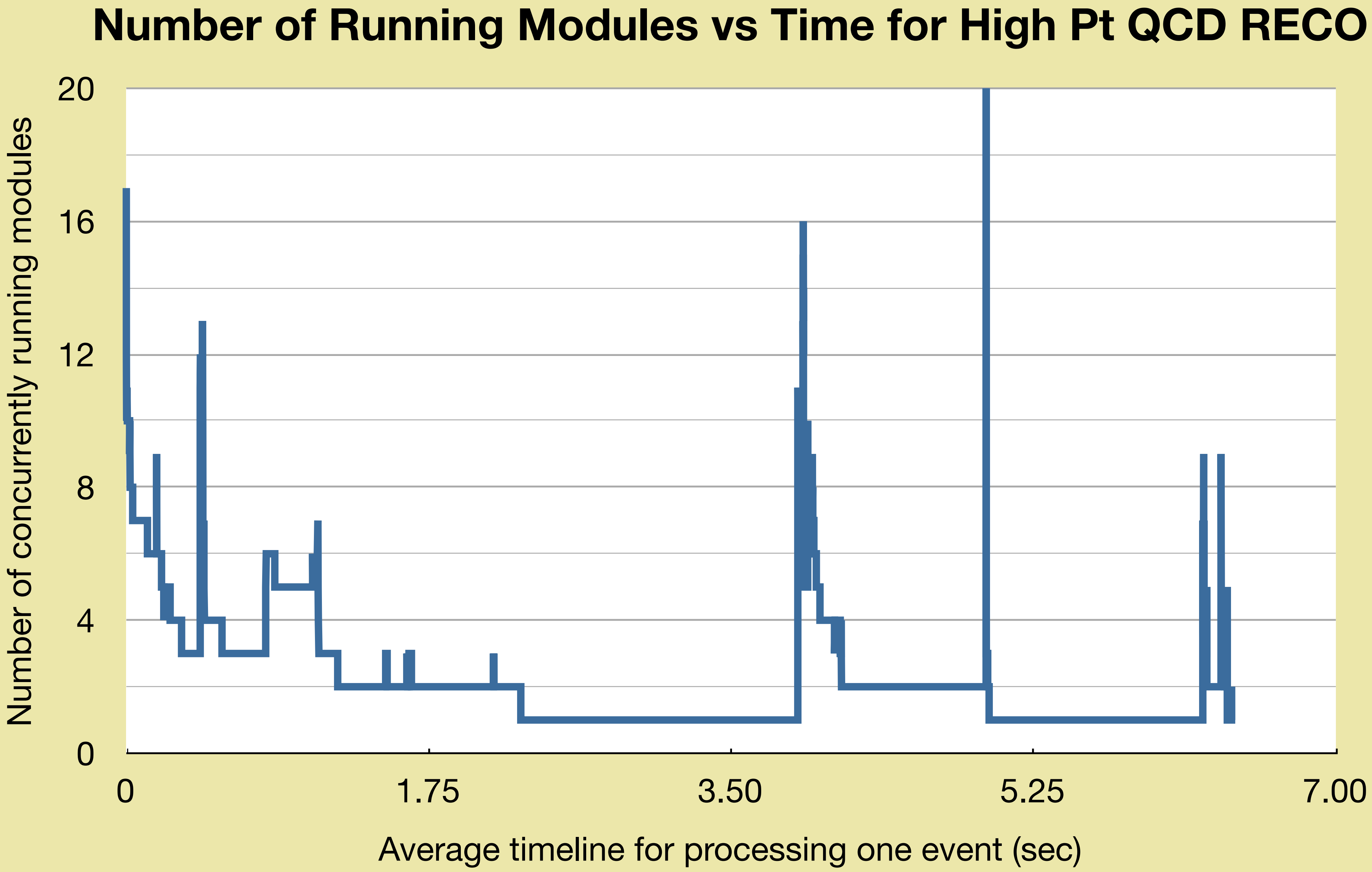
Short periods of high parallelism

Extended periods of only 1 or 2 modules running

Min Bias Estimates



High Pt Estimates



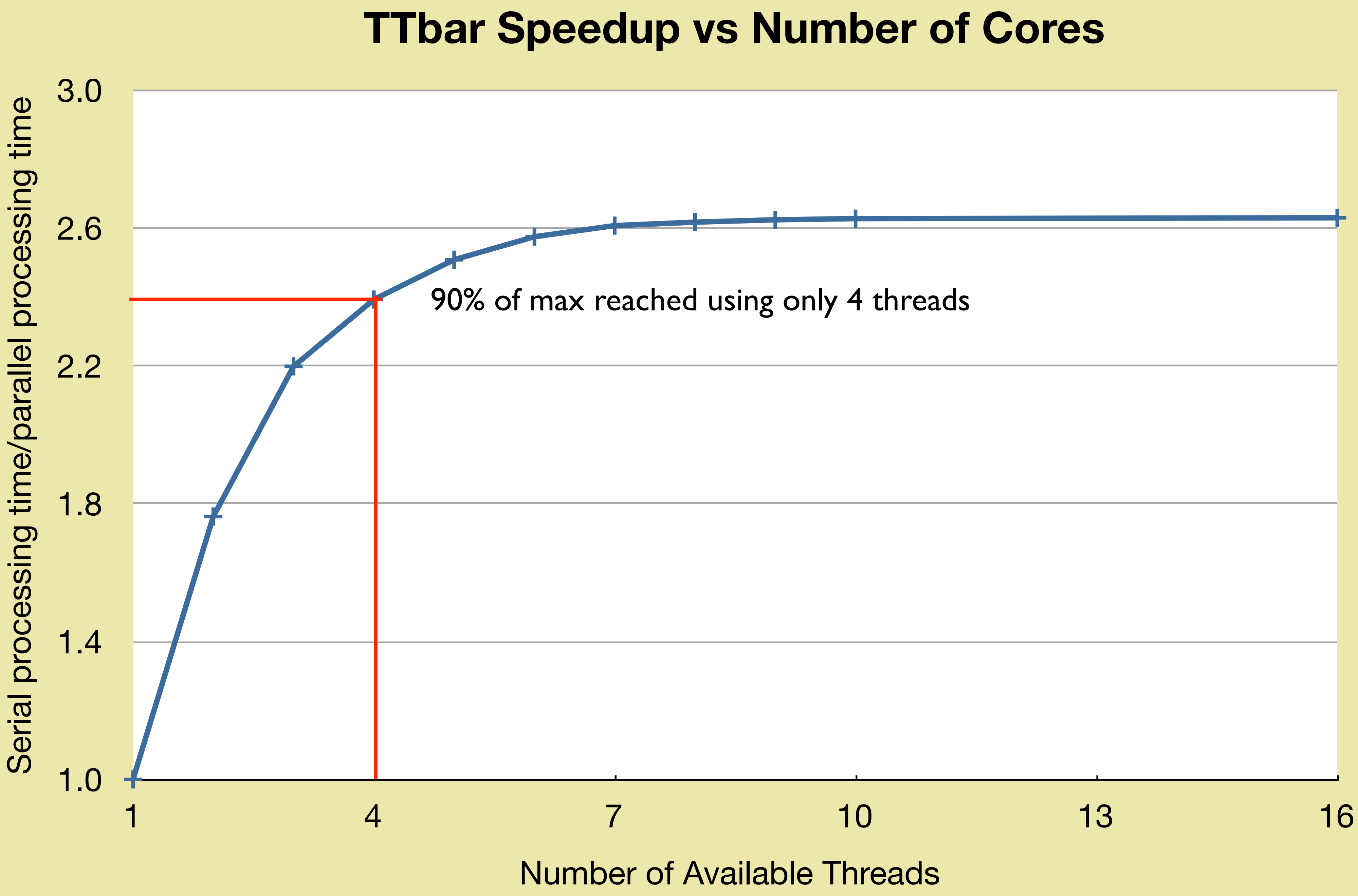
Estimate Comparison

Event Type	Max number of threads	Average number of threads
Minimum bias	26	2.64
T Tbar	16	2.62
High Pt QCD	20	2.19

Fewer Threads

Previous estimates assumed infinite number of available threads

Assume module processing time scales with #modules/#threads



Conclusion

Forking

Provides good memory sharing
- Saving 21 GB when running 32 children

The simplest event splitting per children is fine for MC but inefficient for data
- Need more analysis to determine what event splitting to use for children to get good I/O

Event order in merged file must be controlled to guarantee good analysis read performance
- Splitting reco output on luminosity section boundary should be sufficient

Threading

Future may require using cores to speed up the processing of a single event

Present decomposition of algorithms not conducive to high parallelization

Work needed to make present code thread safe is beyond the potential gains

For now and the near future, forking provides the best benefits