Process Based Parallelism in ATLAS Software

Vakho Tsulaia
LBNL

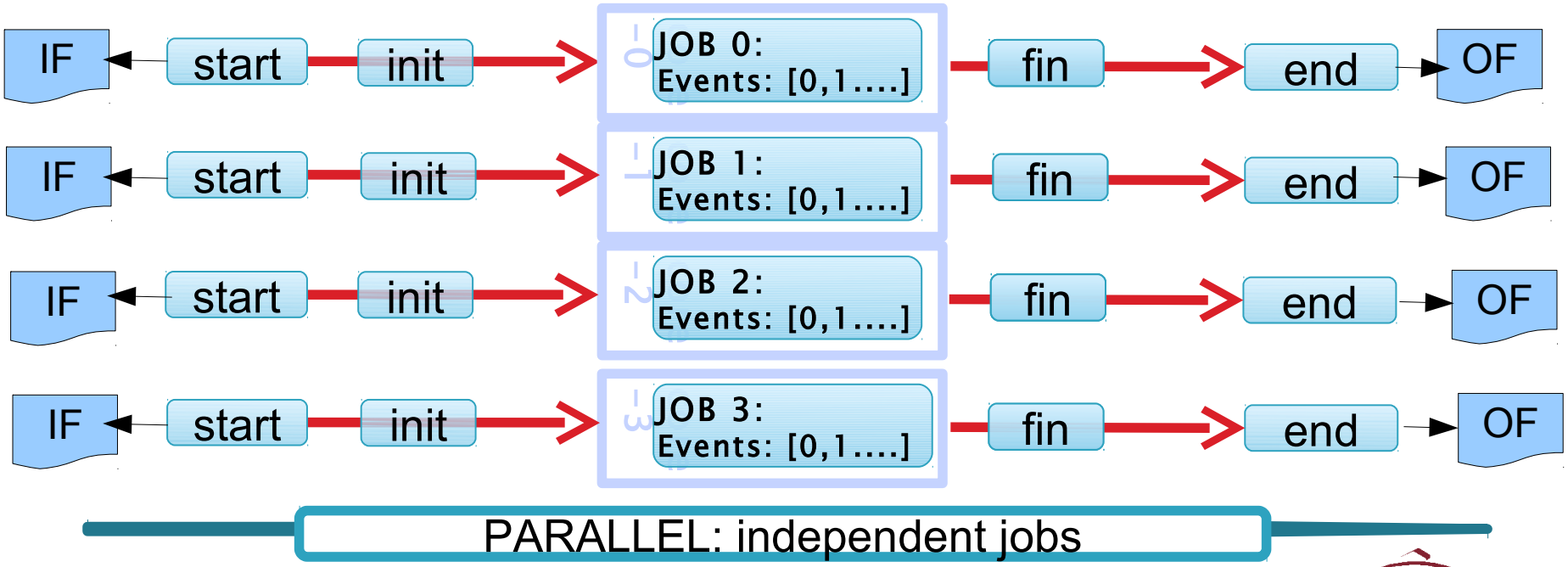
Workshop on Concurrency in the Many-Core Era
FNAL, November 21-22, 2011

Contents

- **Process based parallelism**
- **AthenaMP**
  - Architecture
  - Pros and Cons of the multi-process approach
- **Considerations for future development**
  - Flexible process steering
  - Specialized I/O worker processes
  - Inter-process communication

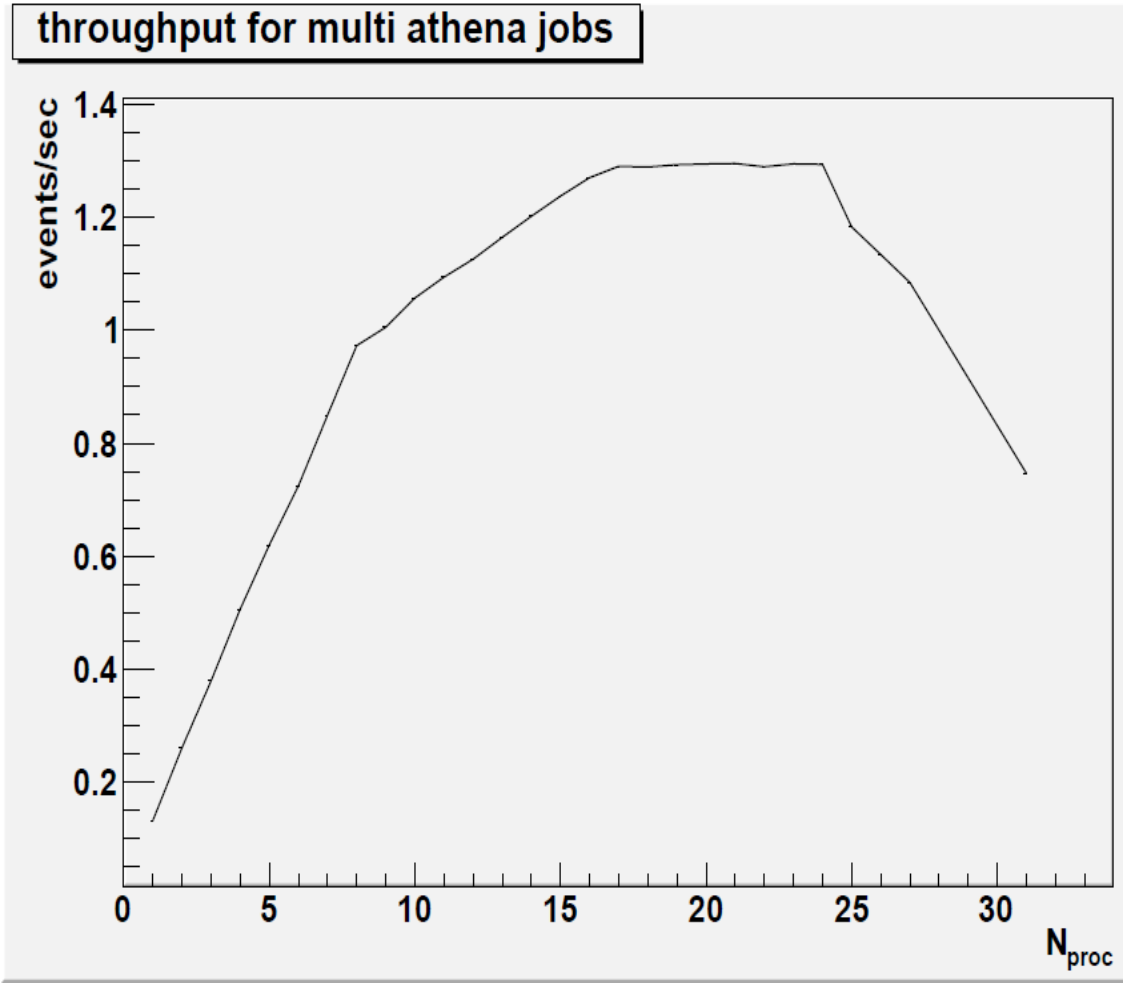
Process based parallelism

- In its simplest incarnation: **Spawn N instances of the application – Athena MJ (Multiple Jobs)**
- No code rewriting required
- We have been using this mode of operation over years on the production system



Athena MJ

- **Can scale surprisingly well (despite hitting hardware memory limits)**
- **The dedicated test run in 32 bit**



- Event throughput vs Number of individual processes
- Standard ATLAS reconstruction
- 8 Core machine, Hyper-threading, total memory 24GB
- Intel(R) Xeon(R) CPU E5530 @ 2.40GHz

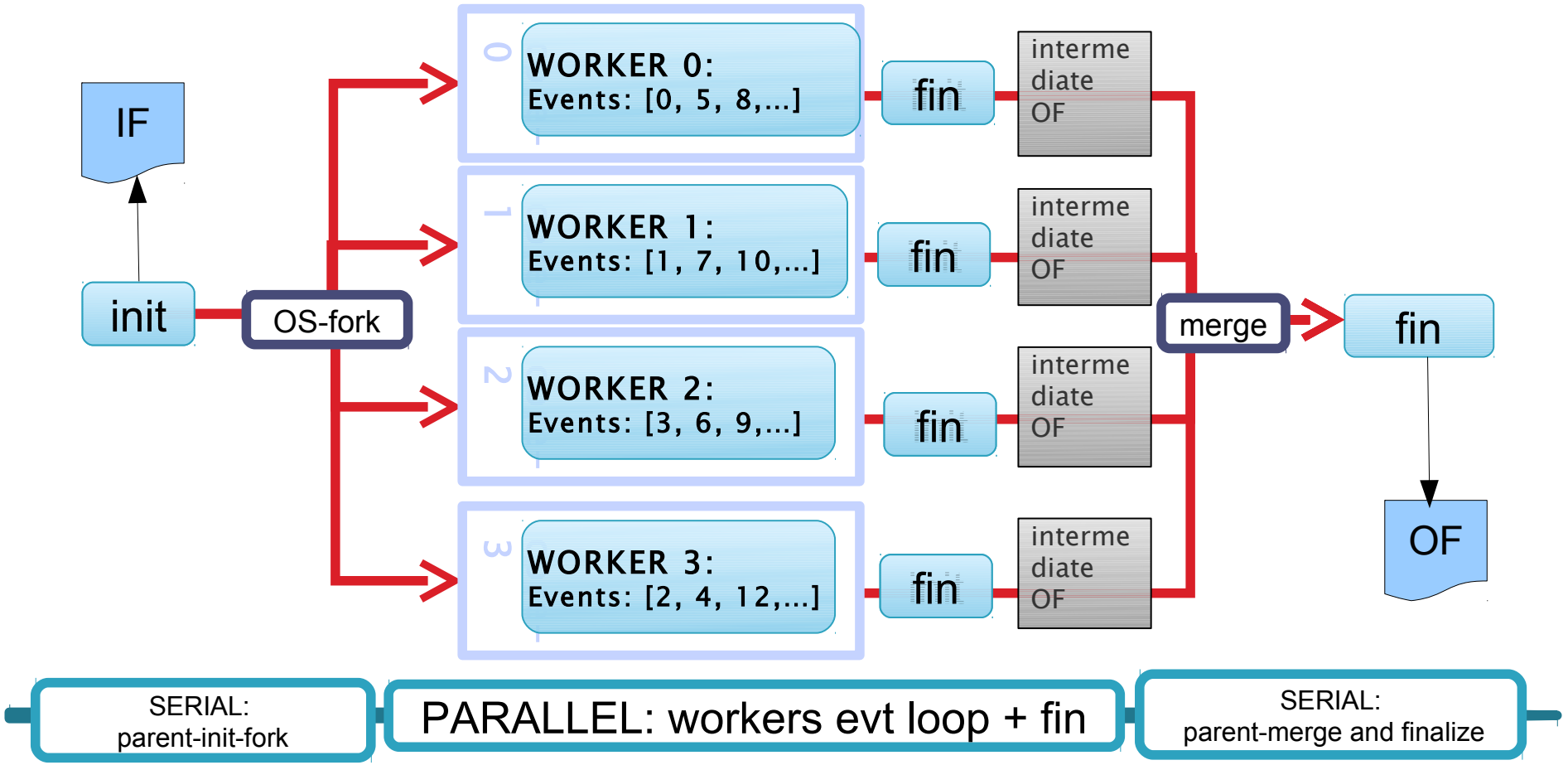
- Improvement up to **N=16**
- Degradation starts at **N=25**

Plot by Rolf Seuster

Resource crisis

- **Memory is a scarce resource for ATLAS reconstruction jobs**
  - **Example:** we can not produce the analog of the plot on previous page for 64 bit simply because that many jobs cannot run in parallel
  - An attempt to run 8 individual reconstruction jobs in parallel in 64 bit resulted to heavy swapping at very early stage of the jobs. The machine stopped responding and had to be rebooted.
- **Situation with I/O is not better either**
  - The scenario when N jobs access events in N files does not scale.
- **We need a parallel solution which allows for resource sharing**

Athena MP


IF
init
OS-fork
0
WORKER 0:
Events: [0, 5, 8,...]
1
WORKER 1:
Events: [1, 7, 10,...]
2
WORKER 2:
Events: [3, 6, 9,...]
3
WORKER 3:
Events: [2, 4, 12,...]
fin
fin
fin
fin
interme
diate
OF
interme
diate
OF
interme
diate
OF
interme
diate
OF
merge
fin
OF
SERIAL:
parent-init-fork
PARALLEL: workers evt loop + fin
SERIAL:
parent-merge and finalize

Process management

- **Athena MP uses *python multiprocessing* module**
- **MP semantics hidden inside Athena in order to avoid client changes**
  - Special MP Event Loop Manager
- **When it is time to `fork()` create Pool of worker processes**
  - Initializer function
    - **Change work directory**
    - **Reopen file descriptors**
  - Worker function
    - **Call executeRun of the wrapped Event Loop Manager**
- **Easy to use, however the simplicity comes at the cost of reduced functionality**
  - More details later in this presentation

Isolated processes

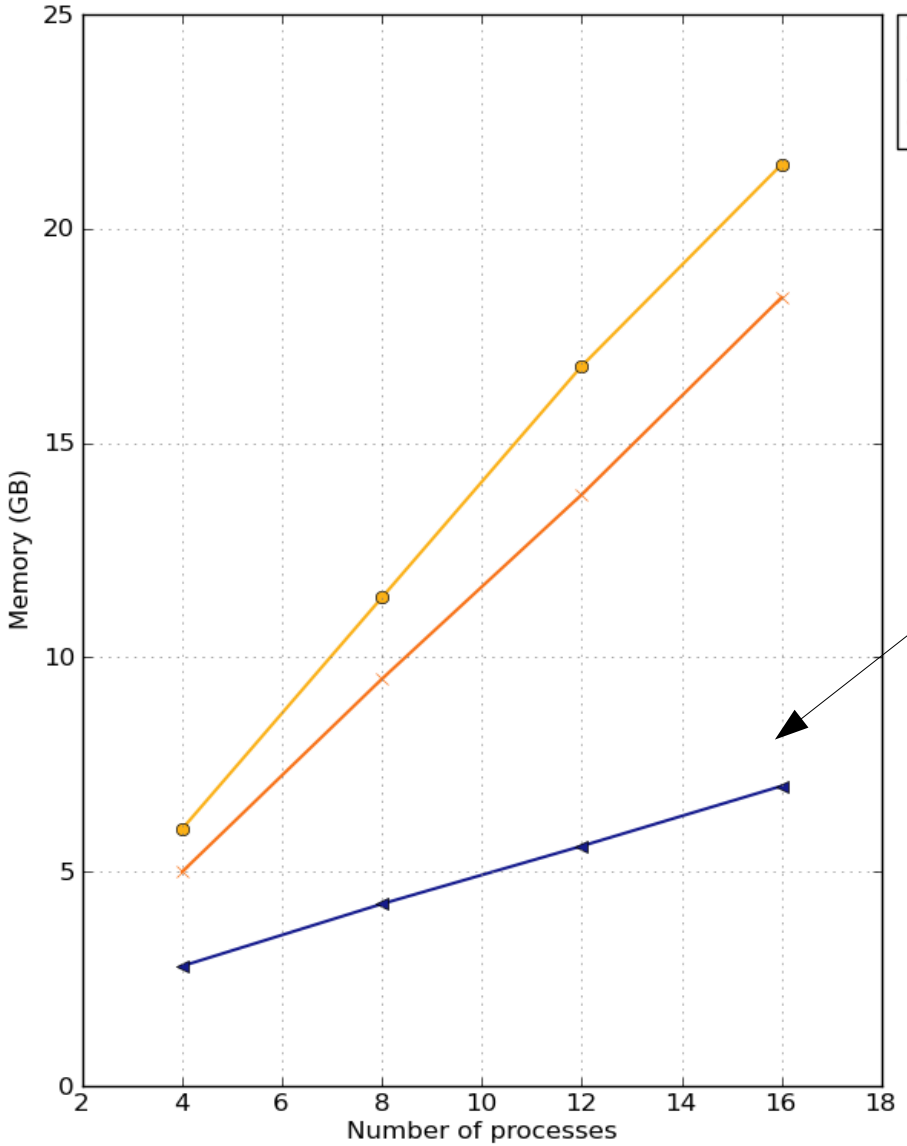
- **AthenaMP worker processes don't communicate to each other**
- **Changes were required only to few core packages**
  - To implement MP functionality and handle I/O
- **No changes are necessary in the user code**
- **In future versions of the AthenaMP workers will have to communicate**
  - But again: the IPC should be either completely isolated from the user code, or exposed to a minimal set of packages

Memory sharing

- **Memory sharing between processes comes 'for free' thanks to Copy On Write**
- **Pros**
  - If memory can be shared between processes, it will be shared
  - No need to do anything on our side to achieve that – let the OS do the work
  - No need to worry about memory access synchronization
- **Optimal strategy: `fork()` as late as possible in order to reduce overall memory footprint**

Effect of late forking

Maximal memory consumption during event loop



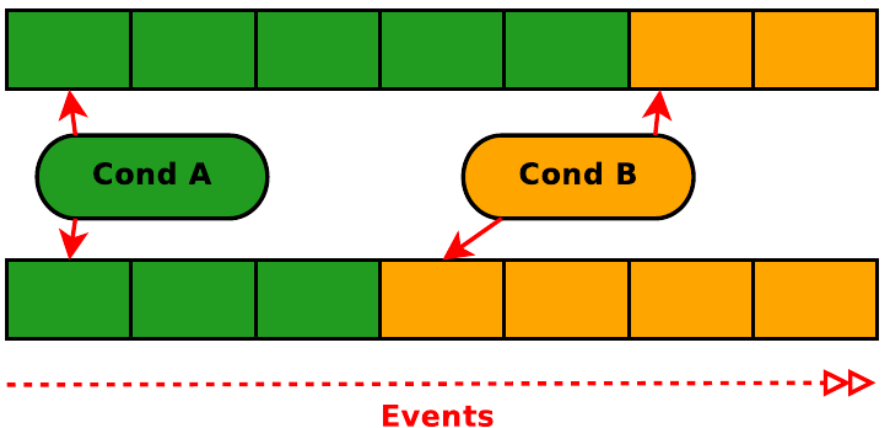
Delayed fork()
1.4GB shared

- Standalone test running standard Athena reconstruction with different number of processes
- Platform:
  - Intel Xeon L5520 @ 2.27GHz
  - 8 Cores
  - Memory 24GB
  - Hyper-threading

COW, handle with care

Unshared memory (1)

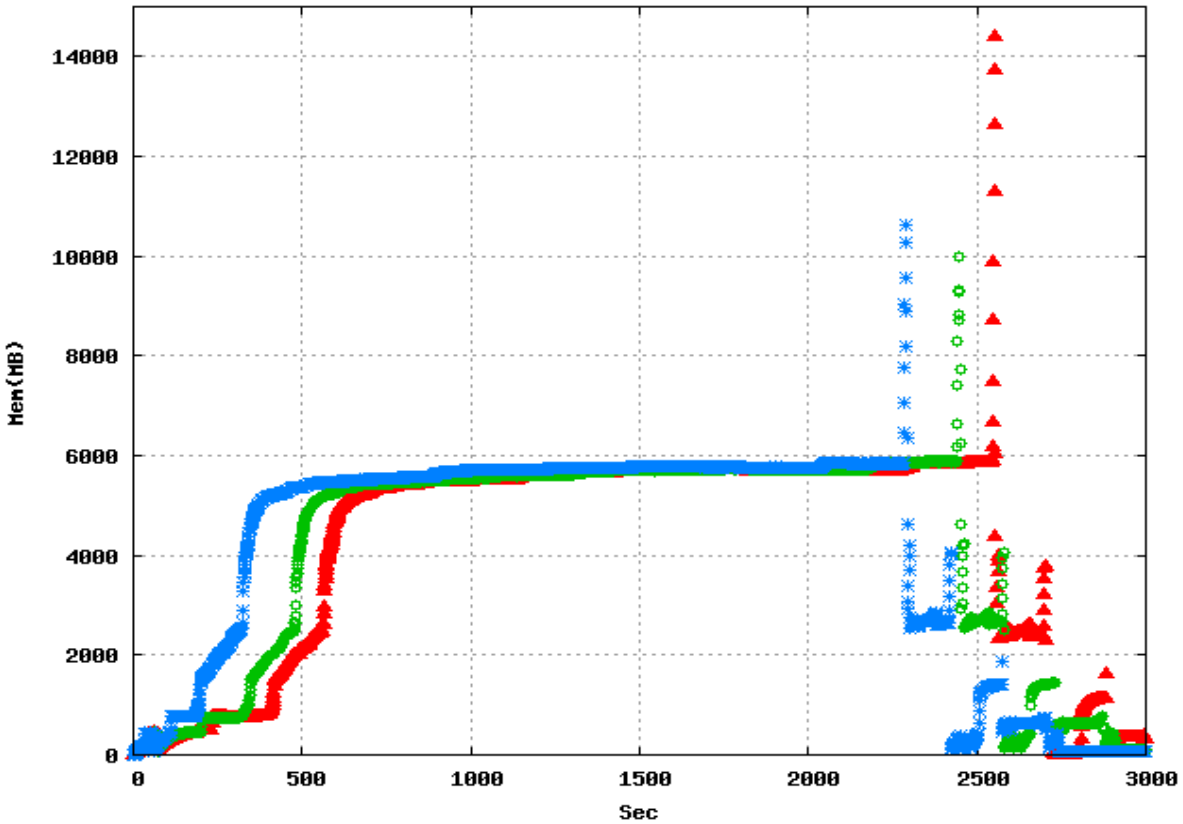
- **Once memory gets unshared during event loop it cannot be re-shared again**
- **Example**
  - Conditions change during event loop and all workers need to read new constants from the database
  - Even though they all get the same set of constants, each worker will have its private copy
  - The amount of unshared memory can become substantial
- **Possible solution/workaround: develop shared storage for conditions data**
  - No plans so far, just an idea



Unshared memory (2)

- **Spikes at `finalize`() caused by massive memory un-sharing**
- **Harmless if remain within hardware memory limits**
- **… otherwise leading to severe CPU performance penalties**

Total memory of one 8 process Athena MP 32 bit reconstruction job vs Wall Time

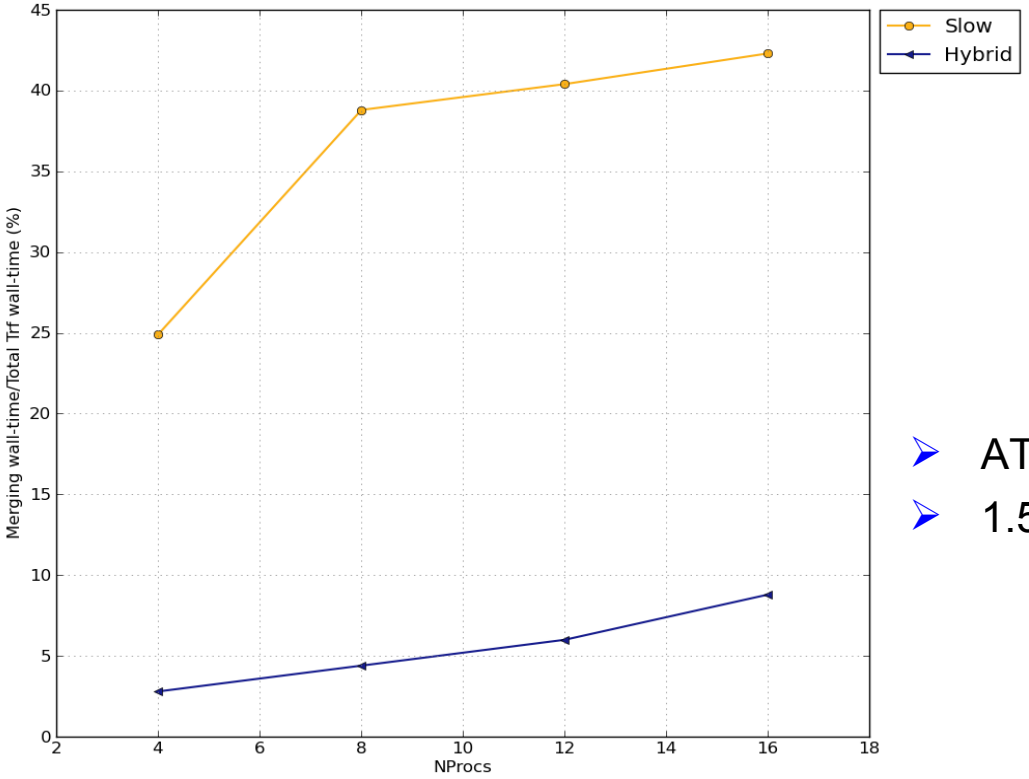


- Same job run 3 times on the same machine
- Spike sizes non reproducible (race conditions)

Output file merging

- **Output file merging is a tedious process, which has large negative impact on the overall performance of the Athena MP**
  - Most of the time is spent in merging POOL files despite of switching to the fast (hybrid) merging utility

Merging time/Total job (transform) wall time



- ATLAS reconstruction RAWtoESD
- 1.5K events

Need for parallel I/O

- **Even with the fast merger Athena MP spends a substantial fraction of time merging POOL files**
- **We also need to avoid reading events from single file by N individual processes**
- **Solution: develop specialized I/O worker processes**
  - Event source: Read data centrally from disk, deflate once, do not duplicate buffers
  - Even sink: Merge events on the fly, no merge post processing

More details in the presentation by Peter VanGemmeren later this afternoon

More on merging

- **Not only POOL files need to be merged**
- **Since recently we also started to include monitoring in our tests and this brought the issue of histogram merging into the list of AthenaMP issues**
- **We seem to have solved problems in histograms produced by individual workers**
  - The right merger is yet to be implemented into AthenaMP infrastructure
- **However the question remains open what to do with certain types of objects, for example TGraph-s**
  - No strategy for the moment

Need for flexible process steering

- **This is critical already now due to python multiprocessing shortcomings**
  - When a child process segfaults and hence does not run the Python-side cleanup the parent will hang forever.
  - Finally the parent process and all remaining zombie children have to be killed by hand
  - Makes it unsuitable for production

- **Proposal: replace python multiprocessing**
  - Move to C++ as the main implementation
  - Keep thin Python layer to allow steering from Python

Development started by Wim Lavrijsen

New steering (1)

- **Requirements**
  - "Clean" behavior on disruptive failures
    - All associated processes die (if need be)
    - No resources left behind
    - Descriptive exit codes
  - Interactive/debugging runs
    - Including the ability to attach a debugger to the faulty process
  - Finer grained driving of processes

- **Also need to address the issue of memory spikes at `finalize()`**
  - Perhaps by scheduling finalization of worker processes

New steering (2)

- **Work on standalone prototype is ongoing**
  - Process organization: use groups
    - Mother and children in separate groups. Can have multiple groups of children
    - Allows waitpid(-pgid) to retrieve all exit codes
    - Allows to suspend workers and attach debugger
    - Allows killing all workers from shell
  - Steering of workers through boost message queues
  - Automatic attachment of debugger to faulty process
  - Retrieval performance monitoring types
  - Improved handling of file descriptors on type

- **Move into AthenaMP will be somewhat disruptive**
  - AthenaMP too tightly coupled to implementation details

Passing objects between processes (1)

- **Do we have a use-case?**
  - None for the moment
  - But we'll certainly need to do that when we have I/O workers

- **Possible candidates to be passed between workers are Incidents**
  - We have implemented some prototype examples for passing file incidents between workers and for broadcasting file incidents from the I/O worker to all event workers
  - The examples are based on boost interprocess, objects are communicated via shared memory segments
  - Since file incidents contain strings we had to play around with interprocess stings, vectors

More on passing C++ objects between processes in the presentation by Roberto Vitillo later this afternoon

Passing objects between processes (2)

- **How to handle such communication between processes?**
  - Should such objects be handled synchronously?
    - Direct intervention in the event processing. Control flow problem
  - Or asynchronously by placing objects into shared memory segments and having consumer processes to check for their existence?
    - When do the client processes perform such checks?
    - How to make sure the objects are delivered to clients in time and no object gets missed?

- **We don't have a clear strategy for the moment**
  - And the absence of real use-cases does not make the situation any easier
  - We may end up defining individual strategies on case by case basis

Summary

- **Despite the relative simplicity of the idea of process based parallelism the actual implementation/validation has taken few years and is far from being over**
  - On the other hand we are now ready to embark on a large scale validation campaign with current version of AthenaMP and hand the results over to physics groups for analysis
- **A memory optimal solution is vital for switching Athena to 64 bit**
- **Move to new, C++ based, multiprocessing is probably the most critical task for the moment**
- **Introduction of specialized I/O workers will bring a fundamentally new level of complexity into AthenaMP**
  - … and for sure will keep us busy for long time

BACKUP

Efficiency: job size

- In order to compete in CPU efficiency with N single process Athena jobs (assuming that we have enough memory for those), we need to **increase Athena MP job size**
  - Run **one Athena MP job over N input files** instead of running N Athena MP jobs over single input file each