# DESC monitoring and performance

# Parsl and funcX fest 2022

David Adams

BNL

September 13, 2022

Updated Sep 16, 2022

# Overview

## Table of contents

# Performance evaluation software

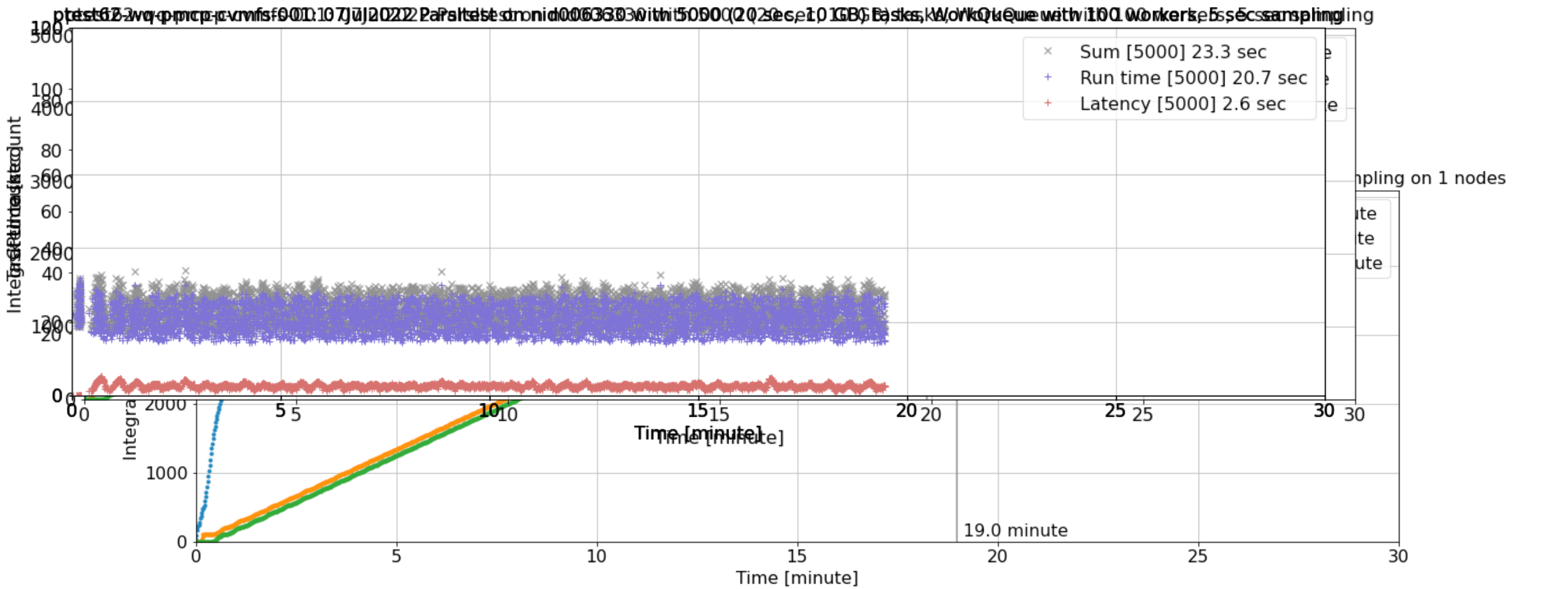
## Software developed to assess performance

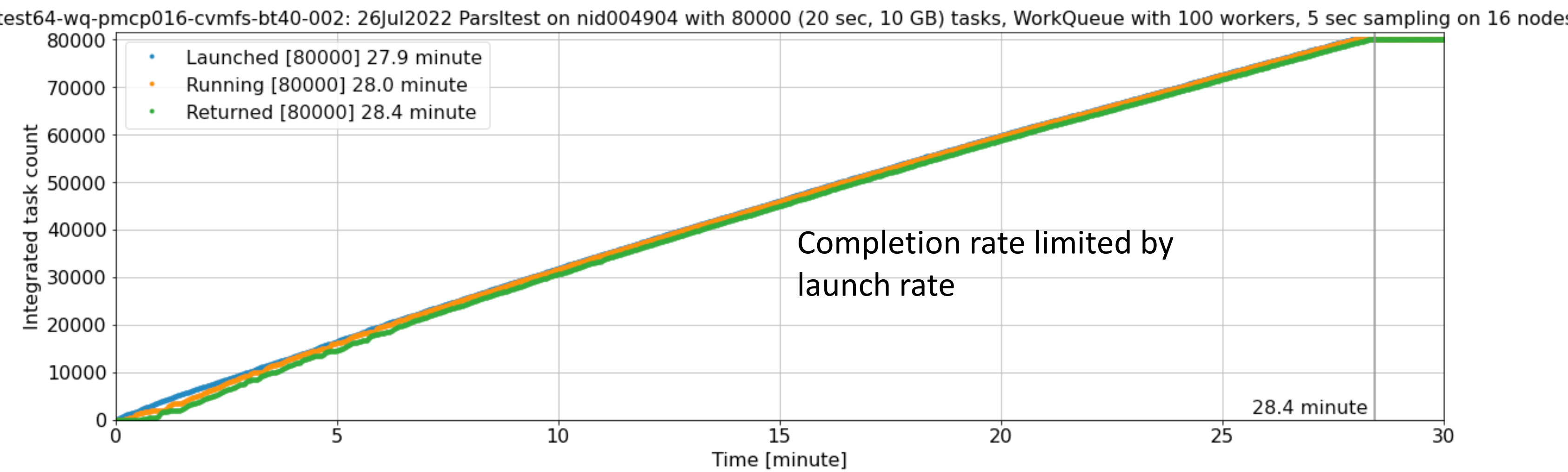
- Github package desc-wfmon
- Extract results from parsl process monitoring DB
  - Sum process data for each task to get the total CPU, memory, I/O, etc. as a function of time
  - Evaluate the latency between one task ending and the next beginning
- Parse logs from perf-stat
  - Extract per-task CPU speed and IPS (instructions per cycle)
- Add system monitoring
  - CPU utilization, memory usage, I/O collected at regular intervals
    - These can be compared these sums over processes
- CPU-intensive parsl test task
  - Each task configured to run for a specified nominal time
    - Actually for a fixed number of instructions
  - When multiple tasks are run, the nominal time is varied over a factor of two so tasks don’t run in phase
- Notebooks to generate performance plots
  - Including those shown here

# Example results

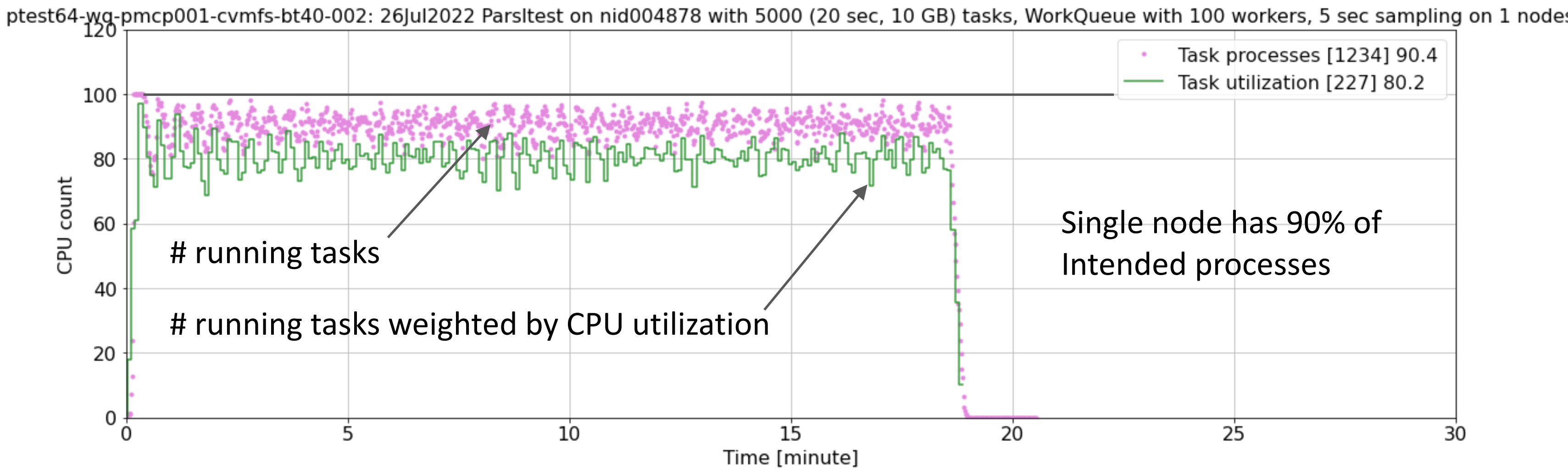
Following pages show some example monitoring plots

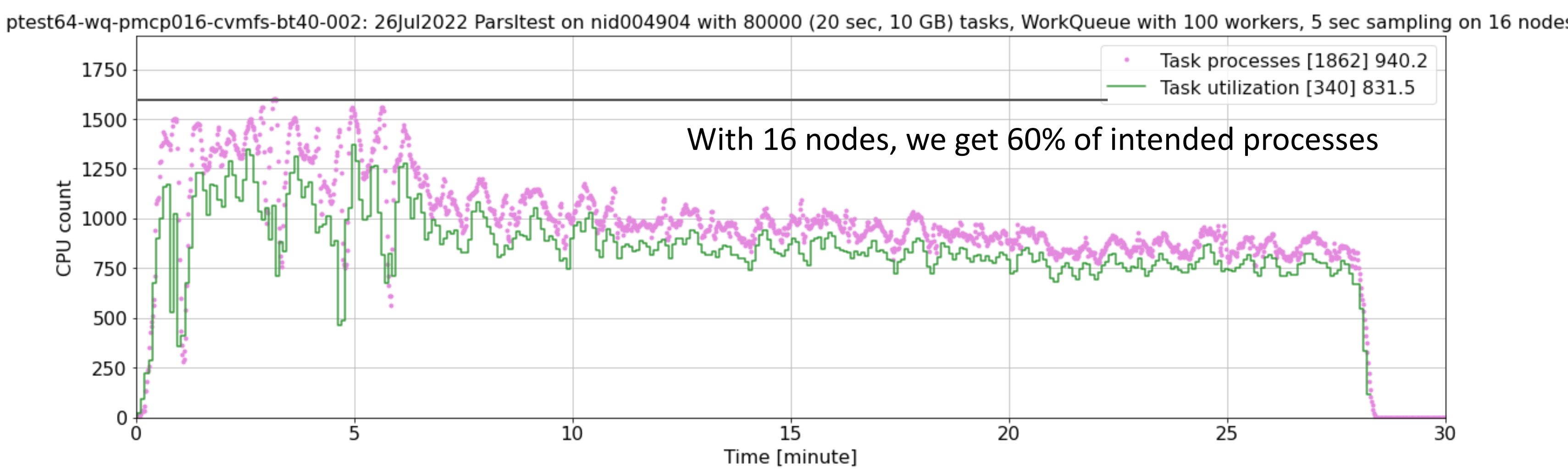
- Configuration
  - Parsl test task with an average of 20 sec/task
    - Similar time obtained with DESC single-frame tasks
  - Run on NERSC Perlmutter
    - Grants exclusive use of one or more nodes of 128/256 physical/virtual cores
  - Most of the python code is from installation on cvmfs
    - LSST release
  - WorkQueue executor with memory size/allocation to run 100 tasks/node
    - Actual number of concurrent running tasks is less when parsl doesn't keep up
- Each page shows two plots
  - Top is one node
  - Bottom is 16 nodes (so 16X as many tasks)


Sum [5000] 23.3 sec
Run time [5000] 20.7 sec
Latency [5000] 2.6 sec
19.0 minute
Time [minute]

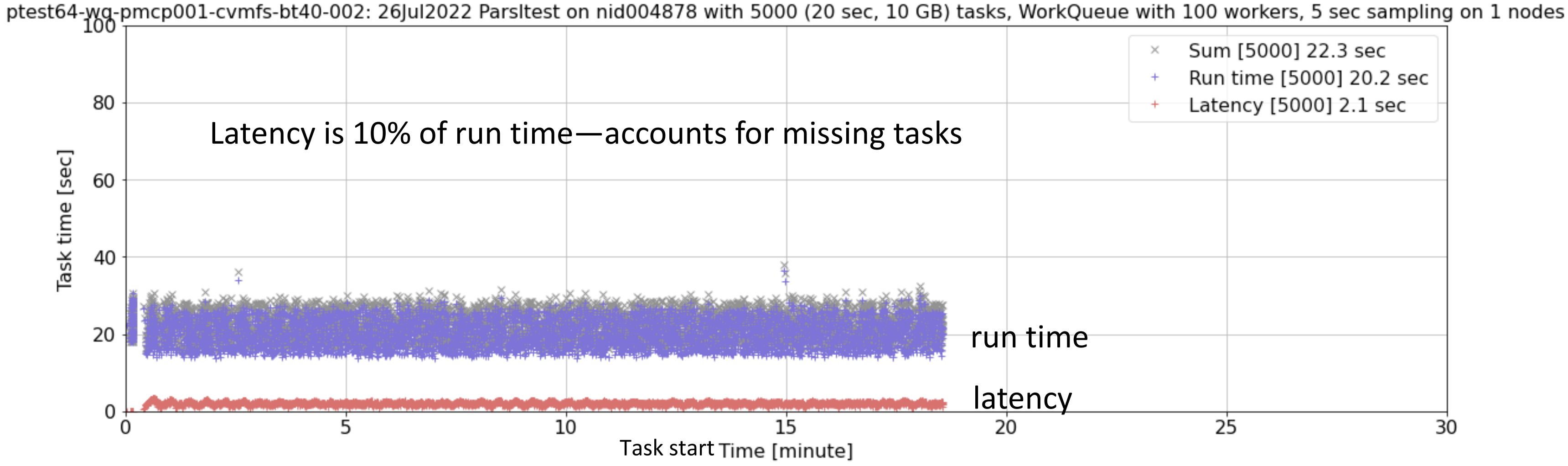

ptest64-wq-pmcp016-cvmfs-bt40-002: 26Jul2022 Parsltest on nid004904 with 80000 (20 sec, 10 GB) tasks, WorkQueue with 100 workers, 5 sec sampling on 16 nodes
Launched [80000] 27.9 minute
Running [80000] 28.0 minute
Returned [80000] 28.4 minute
Integrated task count
Completion rate limited by launch rate
28.4 minute
Time [minute]

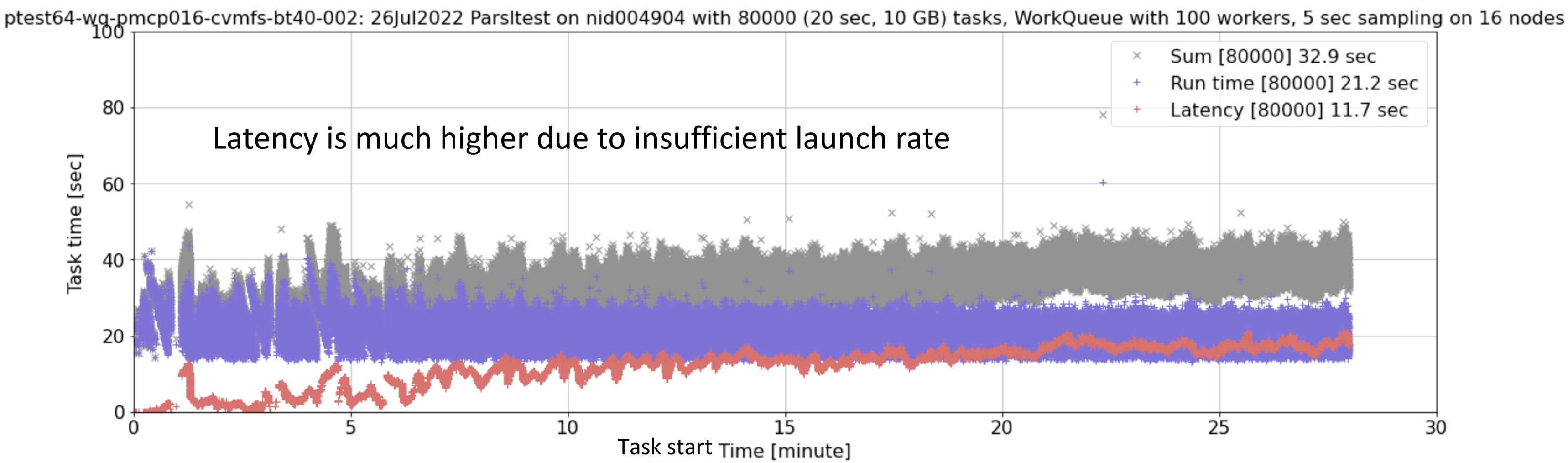
# Example plots: Processes and CPU



# Example plots: Task run time and latency



# Parsl success

Parsl has been very useful for DESC

- Enables processing of image workflows at NERSC (and other sites)
  - To date, simulated data mostly using Cori/haswell
- We will scale up in coming years
  - Larger datasets (real data!)
  - Switch to perlmutter: faster, more CPUs/node, more nodes
- Have been carrying studies to identify issues
  - Added monitoring identify bottlenecks
  - Some issues have been identified →

# Parsl issues for DESC

1. Intrinsic latency
   - This is about 2 sec, so 10% for our 20 second jobs
     - Apparently due to WorkQueue python imports
     - Better or worse if we change the file system where the code resides
2. Insufficient launch rate
   - Limit is about 1000 concurrent processes here
   - Twice as much with HighThroughput executor but still well below that required for DESC for production with one parsl instance
3. Stalls
   - There are periods where the running task count drops precipitously for 10s of seconds (not shown here)
   - Not yet understood—may be a NERSC file system issue
4. Task synchronization
   - If tasks all start together and memory increases with run time, peak is much higher than average memory and limits the # running tasks (not shown here)
5. Task variation
   - Wide range of task run times can make it difficult to optimize throughput
   - See Jim's slide
6. Slow DAG creation
   - Can be bottleneck before real processing starts

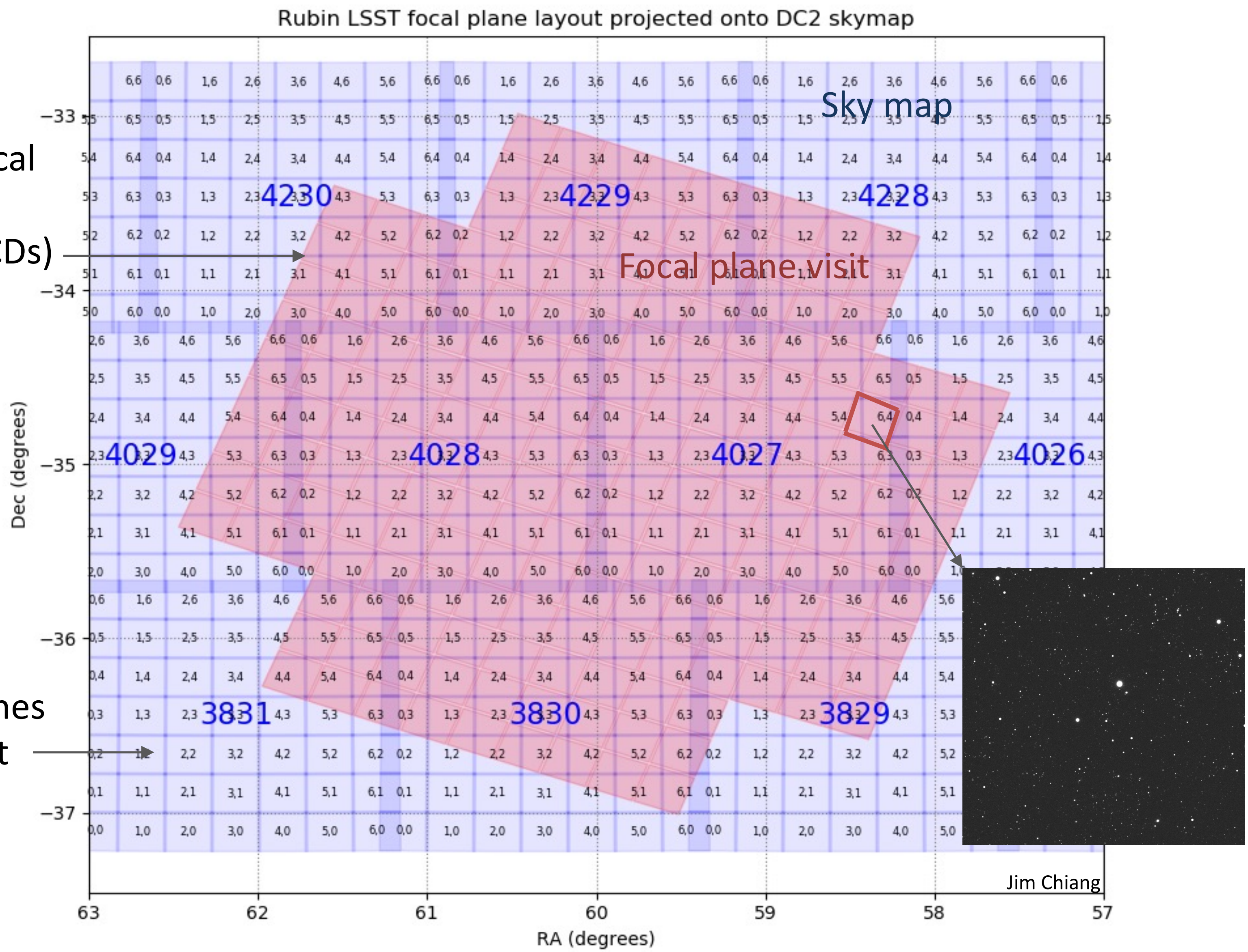
# Production model

## Production model to address these issues

- Single parsl instance will not be sufficient for DESC production
- Instead tailor the solution to our problem
  - First visit frames are processed and then patches—see following figures
- Split production into (at least) two stages
  - Single frame processing
    - Simple DAG: Each CCD in each frame is processed independently
  - Patch processing
    - Each patch is processed independently
    - First step is warping: finding the frame CCDs that overlap the patch
      - » Ensure these are processed before submitting job (instead of DAG)
- Address scaling with a hierarchical production system
  - Top level production manager (PM) provides global view of production
    - It submits jobs to job PMs
  - Each job is a frame, patch or group of either
    - Each job is an independent sub-DAG
  - Natural to have one job PM/node

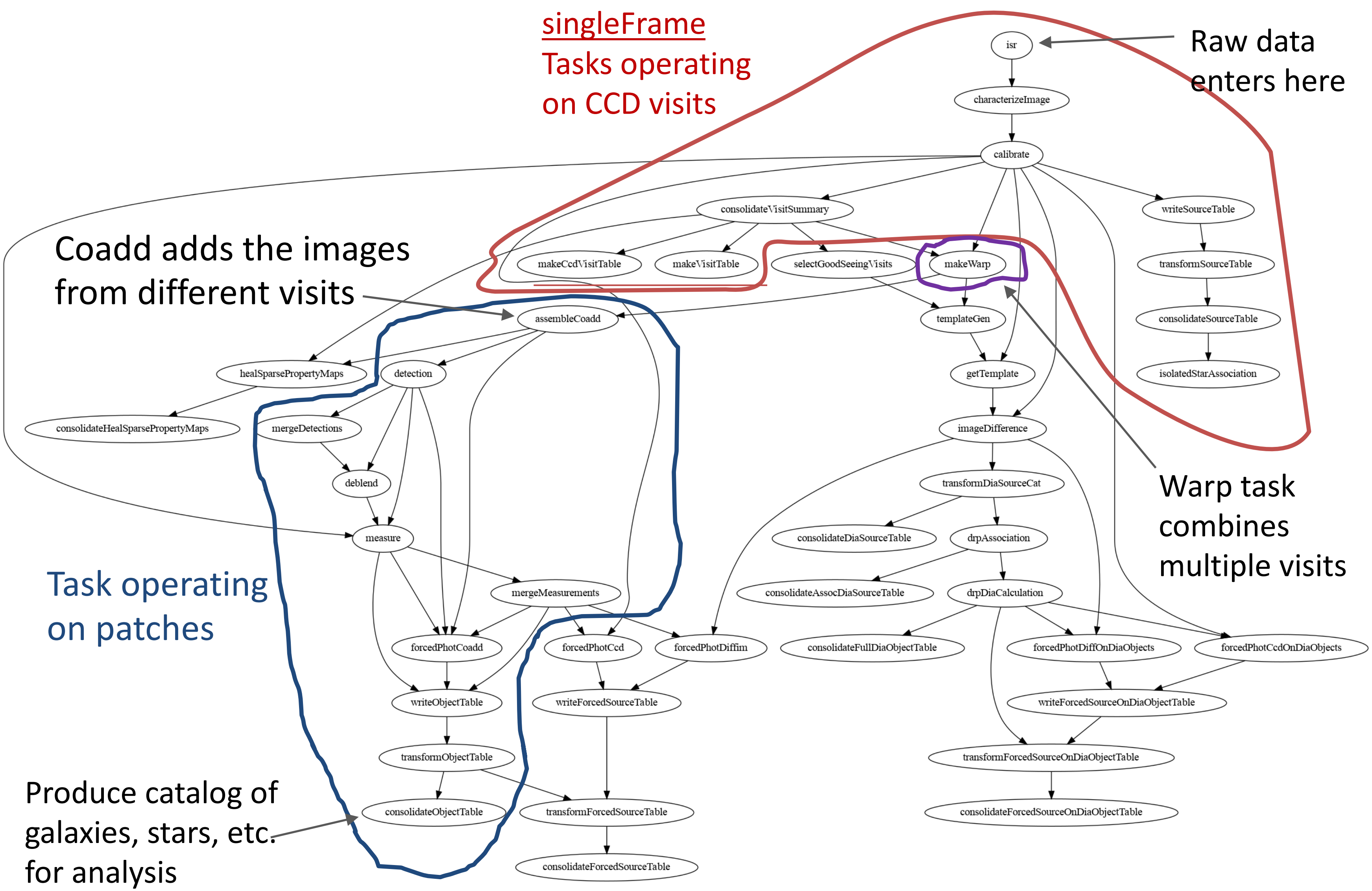
# One example visit

LSST focal plane (189 CCDs)

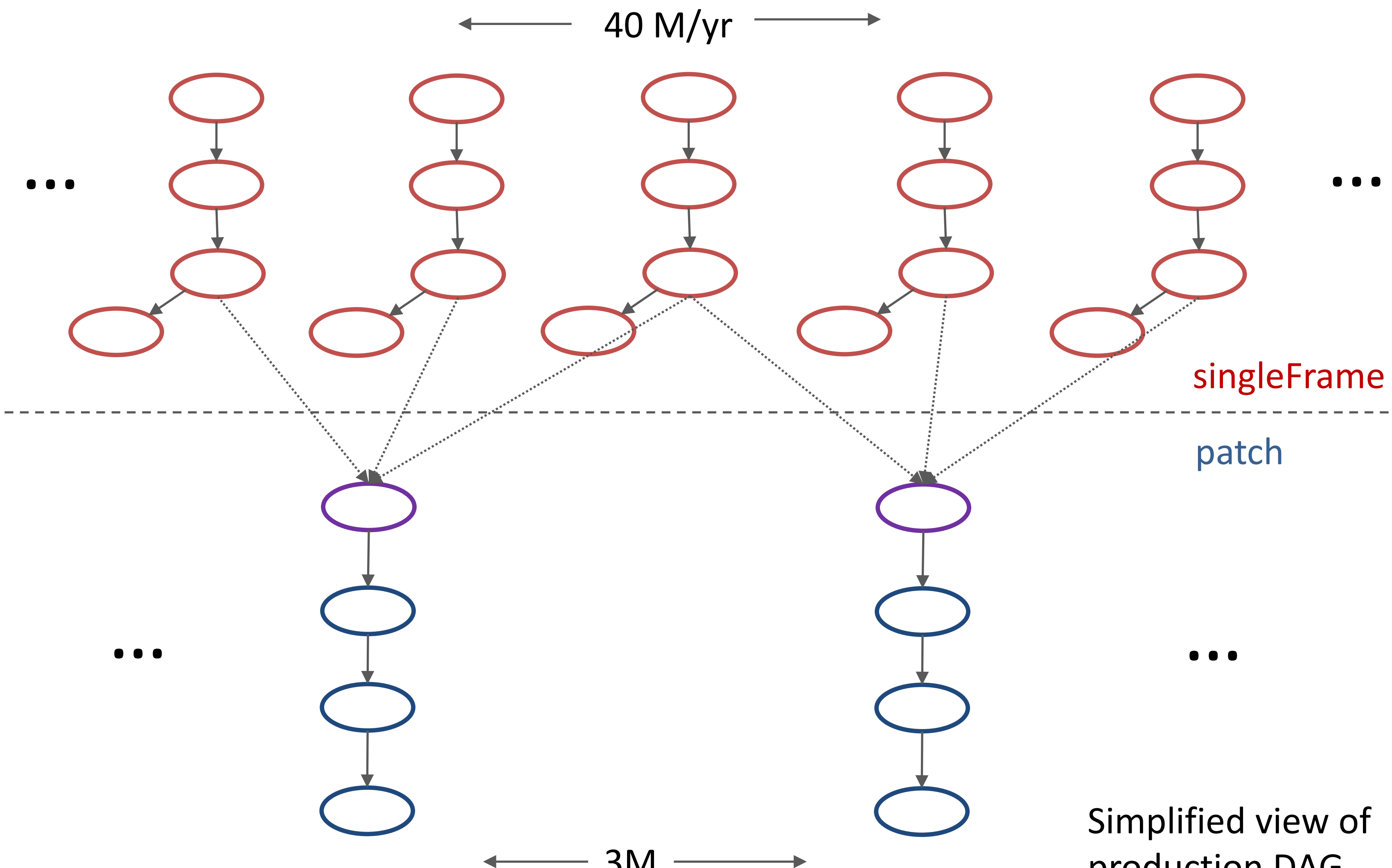



49 patches Per tract

# LSST/DESC workflow (from w_2022_10)



# DAG and sub-DAGs




Simplified view of production DAG

# Production managers

## Top and job production managers (PMs)

- Have different requirements
- Both, either (or neither) might be parsl or parsl-based

## Job PM

- Nice if  job PM might create its own DAG(s)
  - By running a user-supplied command
  - Can parsl do this?
- Like for job PM to be dynamic
  - I.e. be able to handle tasks that add sub-DAGS which are then appended to the overall PM DAG
- Then it could operate in pull mode
  - Go back to the Top PM and ask for more work as needed

## Top PM

- Should be user (i.e. human) friendly and allows user to
  - Submit new jobs
  - Resubmit failed jobs
  - Cancel running or waiting jobs
  - Monitor waiting, running and completed jobs

# Comments/conclusions

## Learning how to best process DESC images

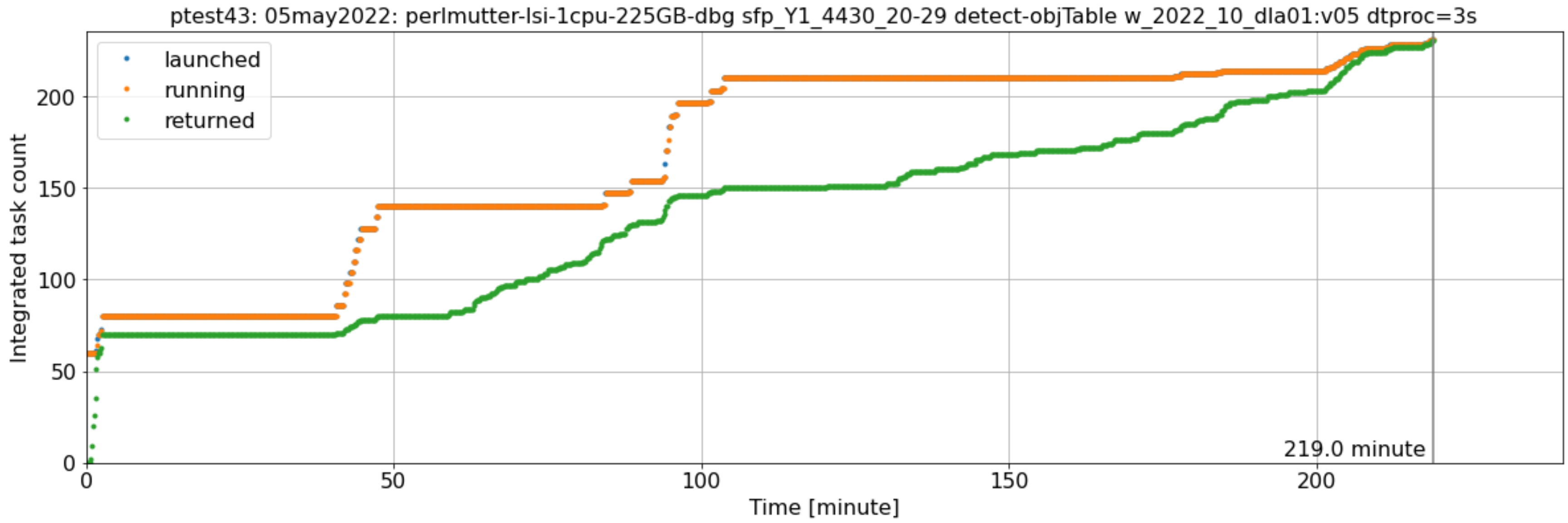
- Plan is to reprocess ~10% of LSST data
- Use NERSC Perlmutter
  - Allocation of 1000 Perlmutter CPU-only nodes
- Current baseline is to use single parsl instance to carry out processing
- But it is a challenge to fill Perlmutter nodes
  - Many DESC/LSST tasks only run for few 10s of seconds
  - Unlikely single parsl instance can efficiently run DESC production at scale
- Proposed here a hierarchical model
  - Job PM (production manager) running on each node
  - Top PM distributing jobs (groups of frames or patches) to nodes
  - Parsl at either level?
  - FuncX to communicate between them?
- Plan to continue studies
  - Demonstrate we cannot (or can) operate at scale with a single parsl executor including optimizations in the coming months
  - Demonstrate that distributed PMs addresses problems that arise

# Thank you

# Extras

# Patch processing performance

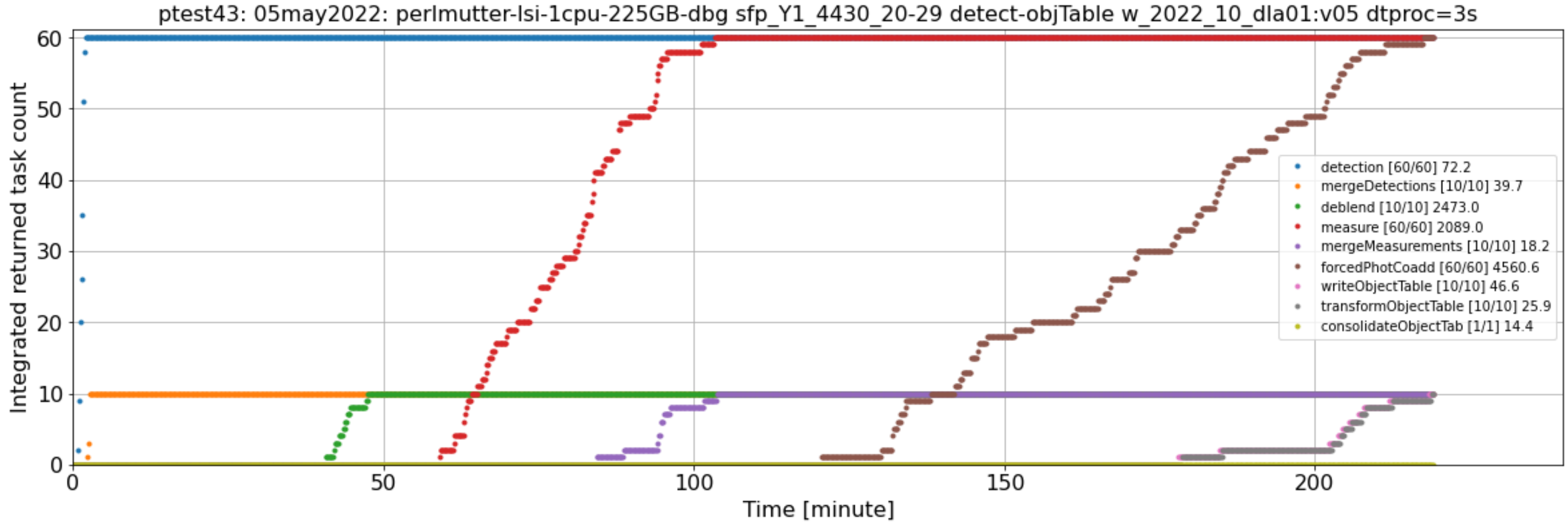
# Throughput for patch processing

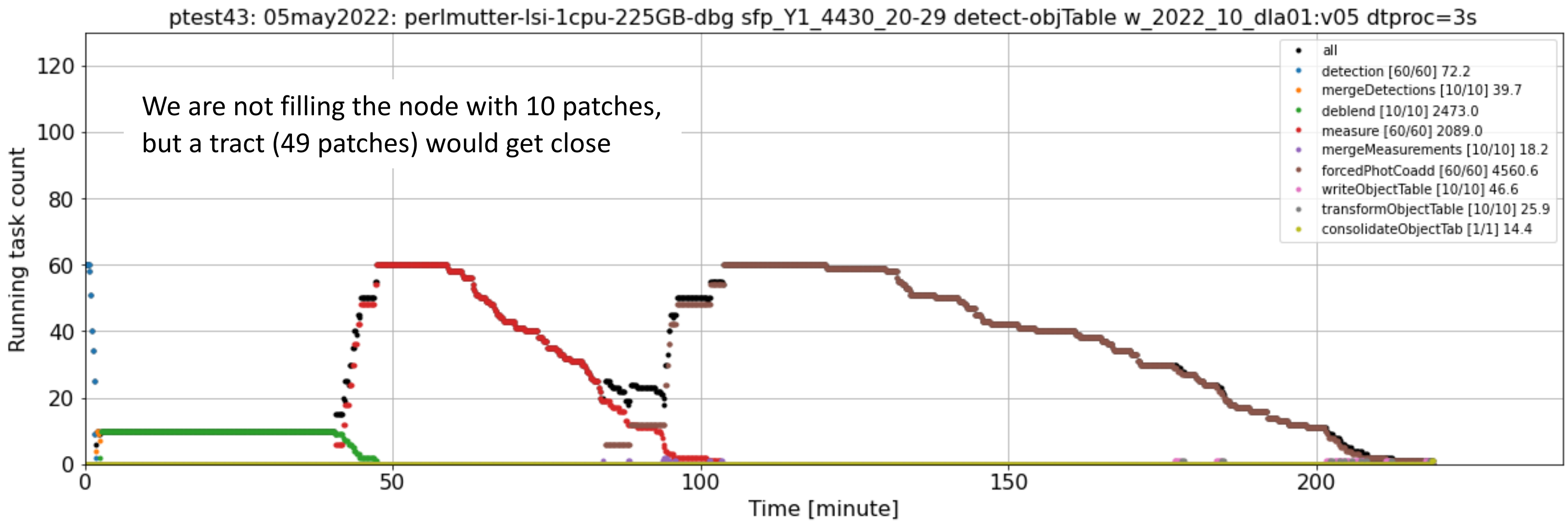



Plot show throughput for patch processing

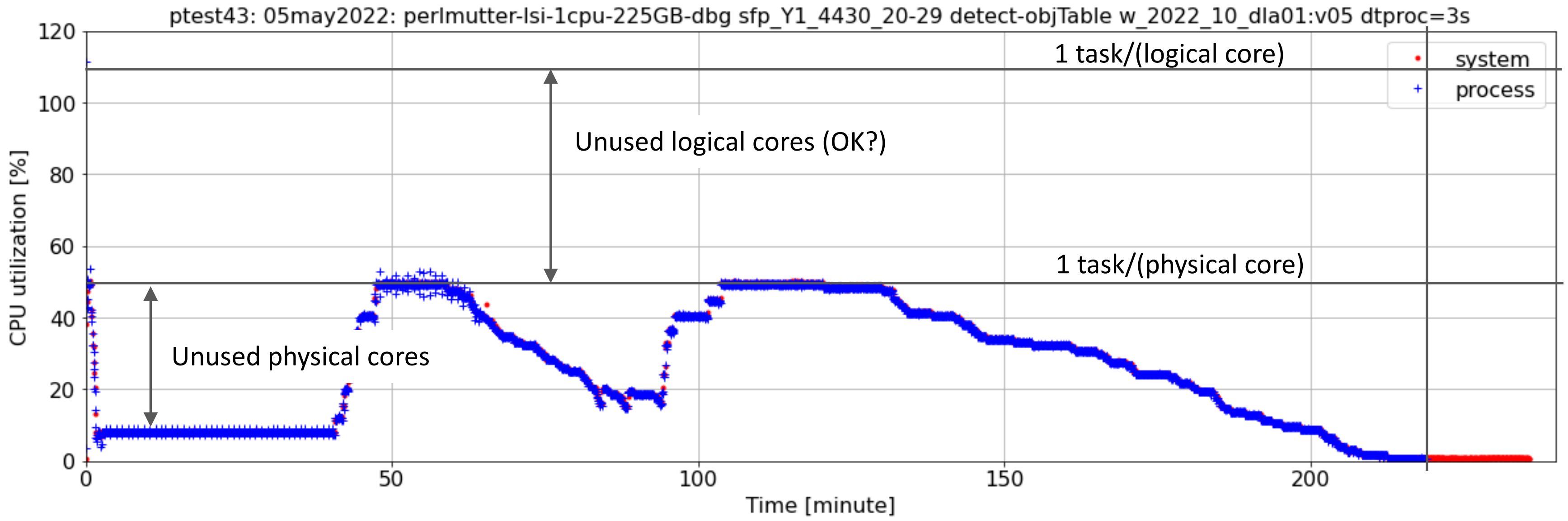
- assembleCoadd through makeObjectTable

# Throughput and # running tasks by task type



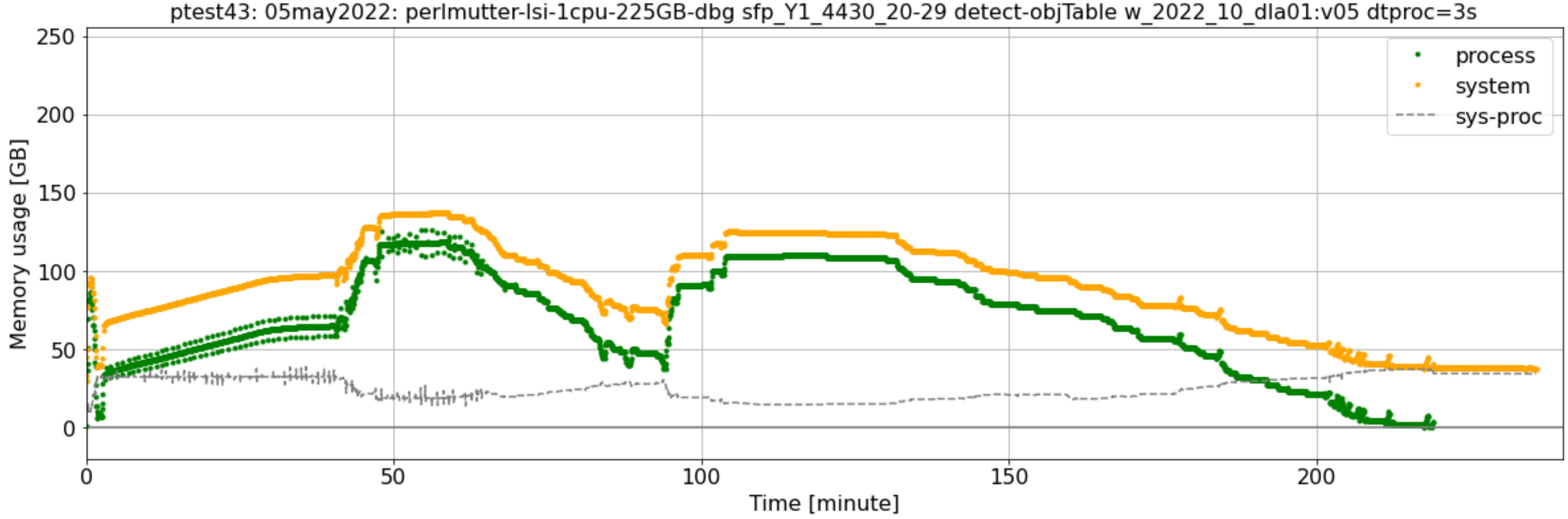
# CPU utilization




Plot show CPU utilization

- Again, we need to run more than 10 patches to fill the node
- But memory prevents this →

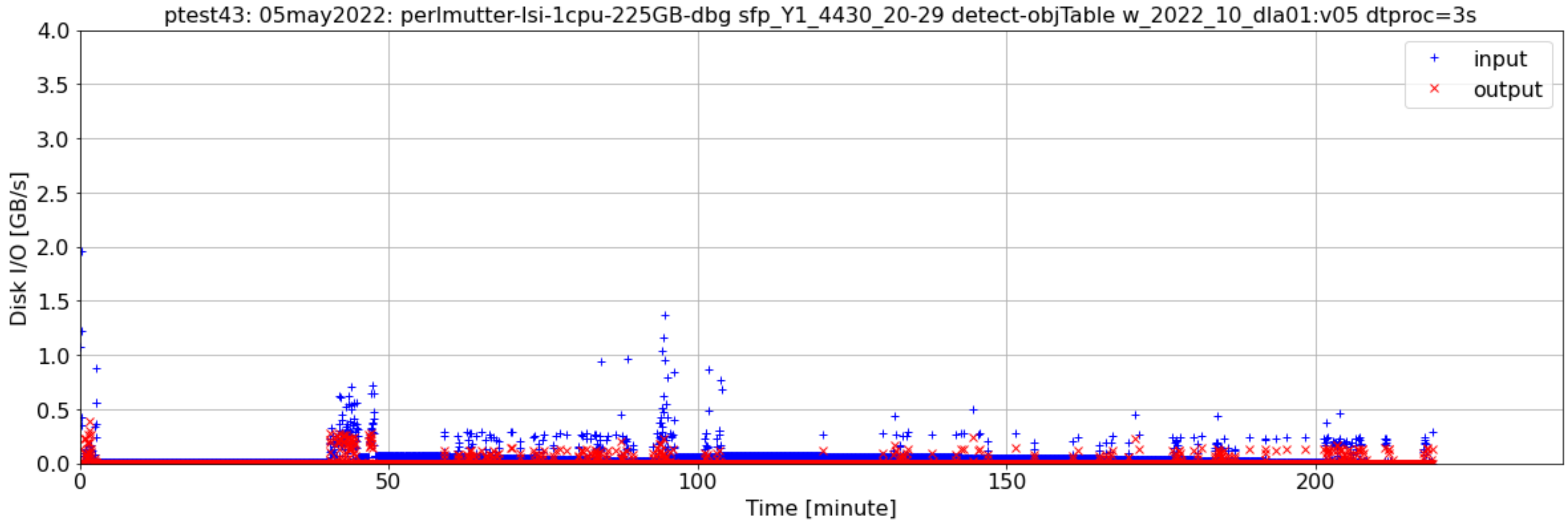
# Memory usage




Plot shows memory usage

- Why the 20-40 GB difference between system and process sum?
- Even using the lower value, we will likely be memory limited and not able to use all the physical cores

# I/O




Plot show I/O vs. time

- Just a few times when rates are higher
- We might want to stagger the patches to smooth some of this
    - Maybe fill with some of the other tasks

# Monitoring schema

# Parsl monitoring raw data

Table workflow has 1 rows and 10 columns
Column names:
  object  run_id
  object  workflow_name
  object  workflow_version
  object  time_began
  object  time_completed
  object  host
  object  user
  object  rundir
  int64  tasks_failed_count
  int64  tasks_completed_count

I do 1 run

Table task has 2158 rows and 15 columns
Column names:
  int64  task_id
  object  run_id
  object  task_depends
  object  task_func_name
  object  task_memoize
  object  task_hashsum
  object  task_inputs
  object  task_outputs
  object  task_stdin
  object  task_stdout
  object  task_stderr
  object  task_time_invoked
  object  task_time_returned
  int64  task_fail_count
  float64  task_fail_cost

with 2158 tasks (jobs)

Table try has 2158 rows and 11 columns
Column names:
  int64  try_id
  int64  task_id
  object  run_id
  object  block_id
  object  hostname
  object  task_executor
  object  task_try_time_launched
  object  task_try_time_running
  object  task_try_time_returned
  object  task_fail_history
  object  task_joins

Three try states

Table node has 0 rows and 12 columns
Column names:
  object  id
  object  run_id
  object  hostname
  object  uid
  object  block_id
  object  cpu_count
  object  total_memory
  object  active
  object  worker_count
  object  python_v
  object  timestamp
  object  last_heartbeat

Table block has 559 rows and 6 columns
Column names:
  object  run_id
  object  executor_label
  object  block_id
  object  job_id
  object  timestamp
  object  status

Table status has 10220 rows and 5 columns
Column names:
  int64  task_id
  object  task_status_name
  object  timestamp
  object  run_id
  int64  try_id

Table resource has 3229 rows and 16 columns
Column names:
  int64  try_id
  int64  task_id
  object  run_id
  object  timestamp
  float64  resource_monitoring_interval
  int64  psutil_process_pid
  float64  psutil_process_cpu_percent
  float64  psutil_process_memory_percent
  float64  psutil_process_children_count
  float64  psutil_process_time_user
  float64  psutil_process_time_system
  float64  psutil_process_memory_virtual
  float64  psutil_process_memory_resident
  float64  psutil_process_disk_read
  float64  psutil_process_disk_write
  object  psutil_process_status

This table has data for each process (task try) sampled at regular intervals

# Process level derived data

Table procsumDelta has 541 rows and 12 columns
Column names:
  float64   timestamp
    int64   nval
    int64   nproc
  float64   run_idx
  float64   procsum_memory_percent
  float64   procsum_memory_resident
  float64   procsum_memory_virtual
  float64   procsum_time_clock
  float64   procsum_time_user
  float64   procsum_time_system
  float64   procsum_disk_read
  float64   procsum_disk_write

This is derived from the resource table.
It sum contribution from all processes.

The times and disk I/O values are deltas—the contribution for each interval rather the the integral in the resource table.

Calculation is tricky and result is sometime misleading because samplings do not have the same phase for all processes and the sampling is occasionally irregular.

# System level monitoring data

System monitor sample count: 619
System monitor columns:
- time
- cpu_count
- cpu_percent
- cpu_user
- cpu_system
- cpu_idle
- cpu_iowait
- cpu_time
- mem_total
- mem_available
- mem_swapfree
- dio_readsize
- dio_writesize
- nio_readsize
- nio_writesize

All sampled at regular intervals
Every 5 sec for jobs here