DESC production

NPPS

David Adams
BNL
May 6, 2022

Overview

The following are discussed here

- Introduction
- Single frame performance
- Performance for isr only
- Varying the number of concurrent processes
- Patch processing performance

Introduction

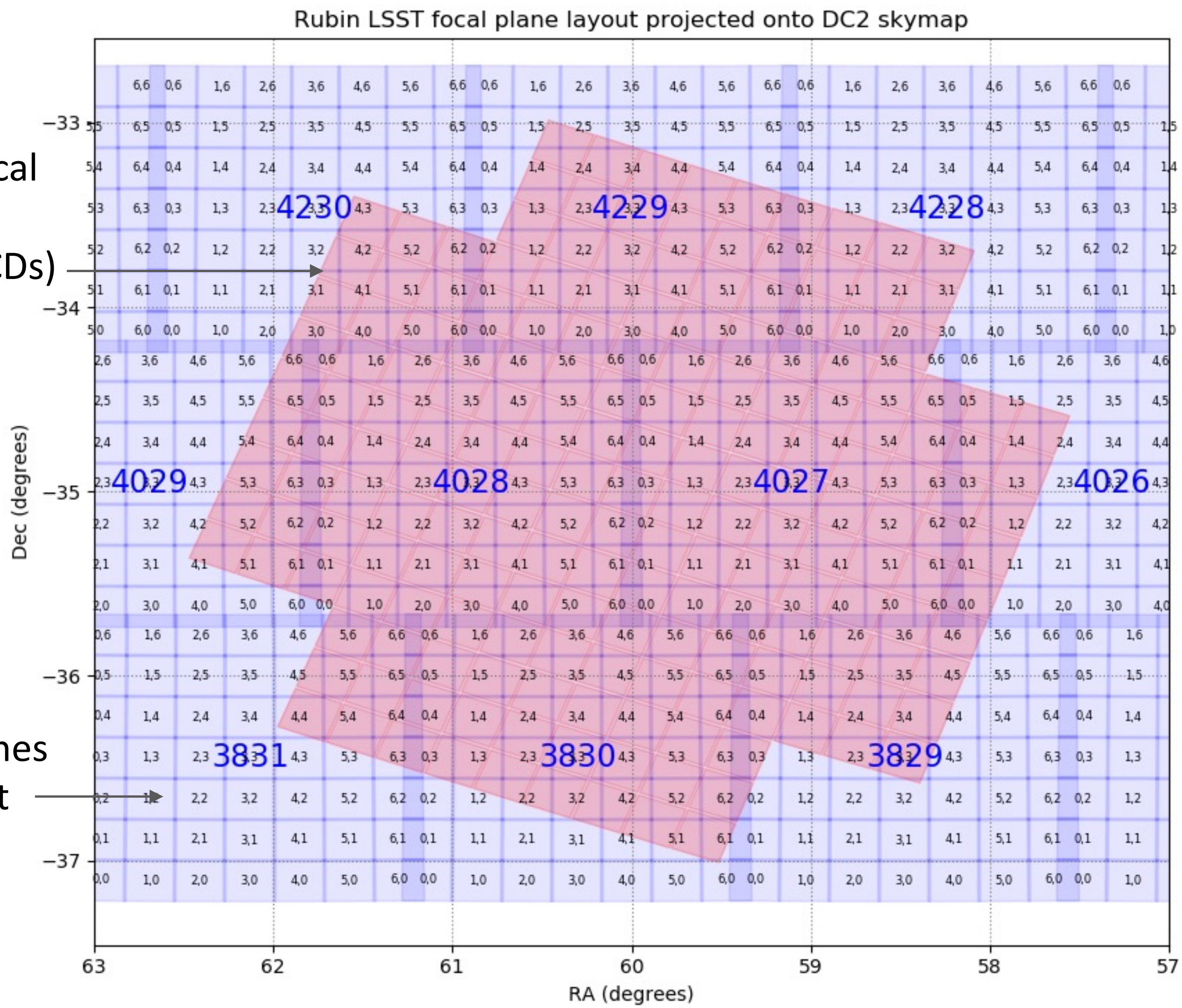
DESC production

DESC is running the LSST processing chain

- Start from raw CCD data all the way through to object catalog
  - Objects include stars, galaxies with 3D positions (angles, z)
  - Includes merging of many overlapping images
    - Wide variety of positions and orientations to cover the (southern) sky and cancel systematics
    - Each visit (2 images) includes data from 189 sensors (CCDs)
  - Sky is divided up into tracts and patches for processing
    - Following page shows focal plane overlaid on tracts and patches
    - Size of patch is about the same a CCD
    - I have been running jobs that process 10 patches
      - » Soon 1 tract = 49 patches
- When real data comes, DESC will reprocess some fraction (10%??)
  - Want to optimize use of compute resources at NERSC
  - Report code and workflow improvements to LSST

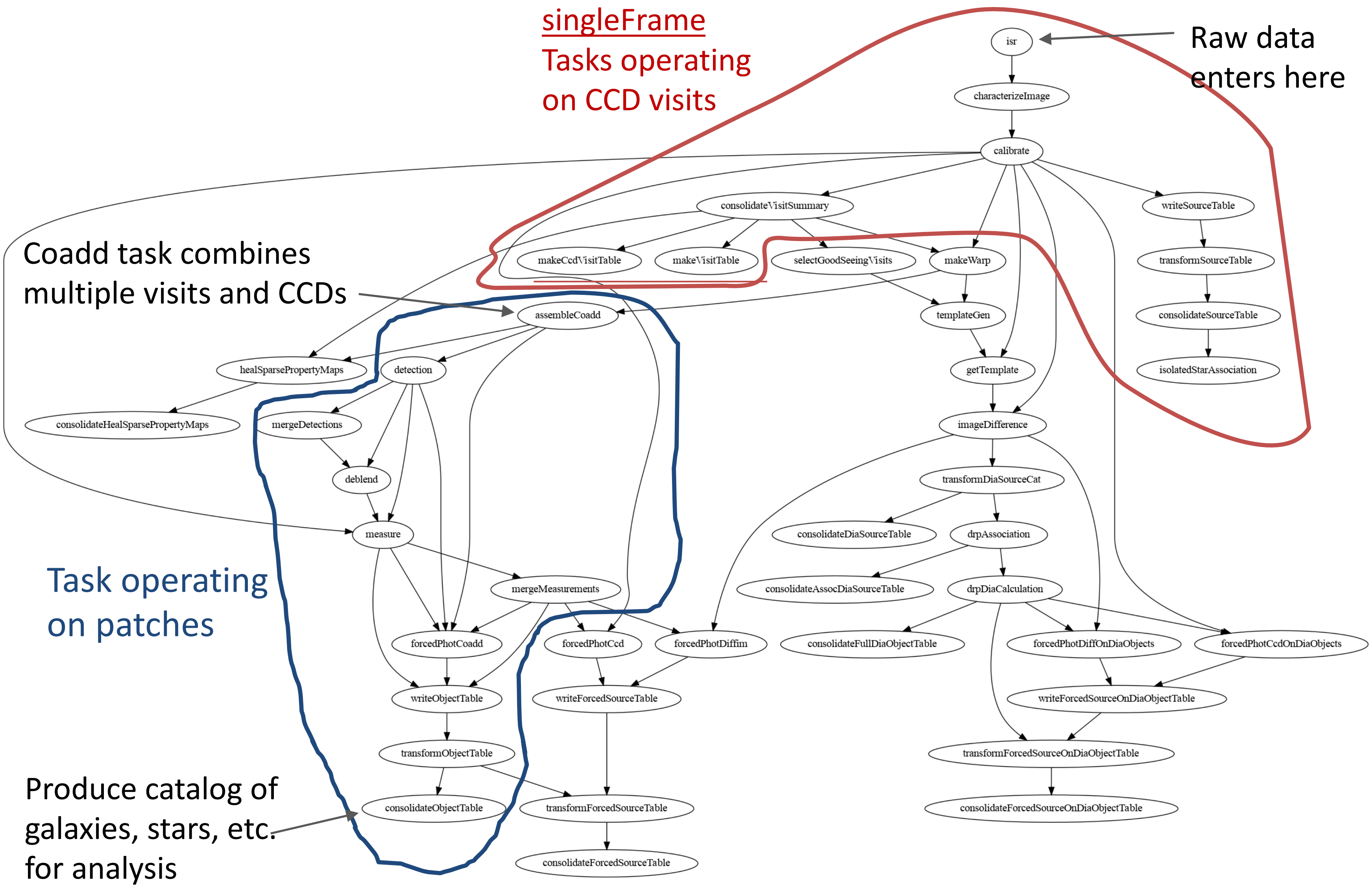
One visit (2 images)

LSST focal plane (189 CCDs)



49 patches Per tract

LSST/DESC workflow (from w_2022_10)



Scheduling

Result here are reported for NERSC Perlmutter

- Being commissioned but available for users to try out
- All runs here were done on a single node:
  - 128 CPUs (2 logical cores on each of 64 physical cores)
  - 256 GB total memory

Parsl does the scheduling

- User provides DAG specifying list of tasks and their dependencies
- User limits the number of running tasks with
  - Specified number of CPUs per task
    - Here set to 1 except for a couple test runs
  - Specified memory to allocate for each task and total available
    - These are often depend on task type and were tuned in earlier studies

Example job configuration

```
includeConfigs:
  - ${GEN3_WORKFLOW_DIR}/python/desc/gen3_workflow/etc/bps_drp_baseline.yaml
  - ${GEN3_WORKFLOW_DIR}/examples/bps_DC2-3828-y1_resources.yaml

pipelineYaml: "${OBS_LSST_DIR}/pipelines/imsim/DRP.yaml#assembleCoadd"

payload:
  inCollection: u/dladams-pm/sfp_Y1_4430_20-29_ptest39
  payloadName: sfp_Y1_4430_20-29_ptest41
  butlerConfig: /global/cfs/cdirs/lsst/production/gen3/DC2/Run2.2i/repo
  dataQuery: "skymap='DC2' and tract=4430 and patch in (20..29) and visit < 262622"

parsl_config:
  retries: 1
  monitoring: parsl.monitoring.monitoring.MonitoringHub(
           hub_address=parsl.addresses.address_by_hostname(),
           hub_port=55055,
           monitoring_debug=False,
           resource_monitoring_interval=1,
       ),
  executor: WorkQueue
  provider: Local
  nodes_per_block: 1
  worker_options: "--memory=225000"

operator: dladams-pm
commandPrepend: "time perf stat -d"

# Default task properties.
requestCpus: 1
requestMemory: 1000
```

List of tasks to run

Full workflow definition

Input data location

Output data location

Memory available for scheduling tasks in 225 GB

Prepend each task execution command with time and perfstat to collect data for the logging monitor

By default, each task is assumed to use 1 CPU and 1 GB memory. The memory limit is overridden for some task types.

Monitoring

Performance data is taken from three sources

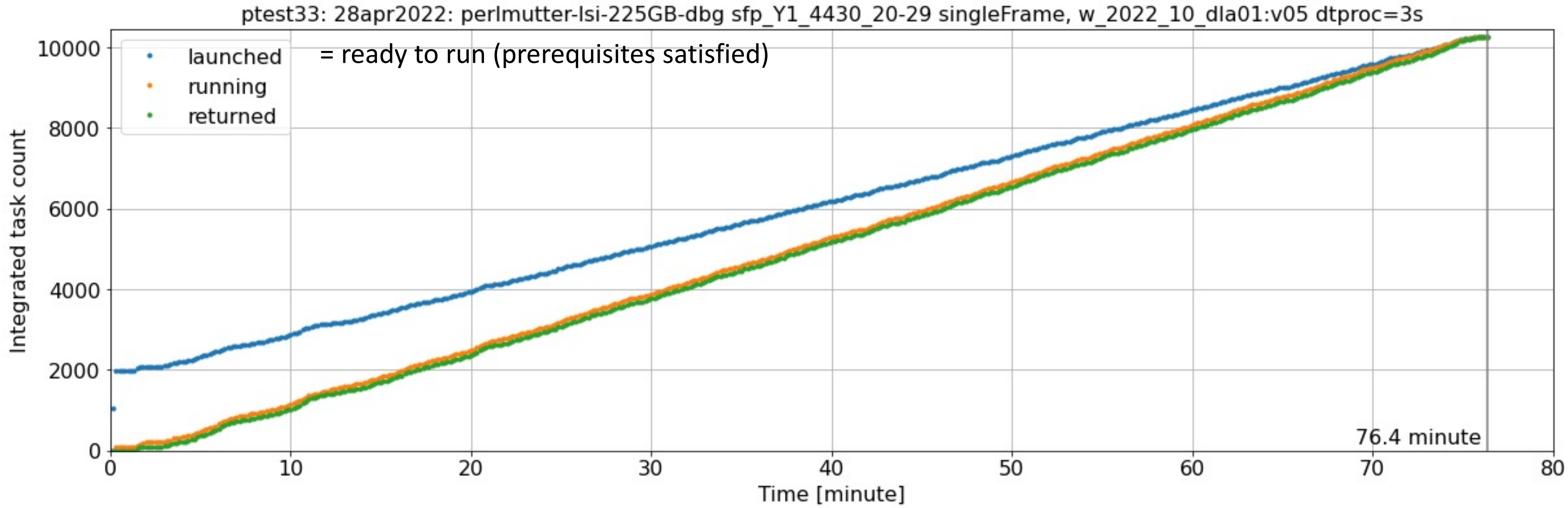
- Parsl monitoring (mysql DB) includes
  - Information about each run (I do 1 run per job)
  - Information about each task and task try (only 1 if all goes well)
  - Process monitor does regular sampling of running tasks
    - Plus a sample at the end
- System monitor
  - Regular sampling of system info
- Log monitor
  - Parse the output log of each task

Notebooks used to make the plots shown here:

- monexp – Parsl and system monitors
- perfstat – Log monitor

SingleFrame perfomance

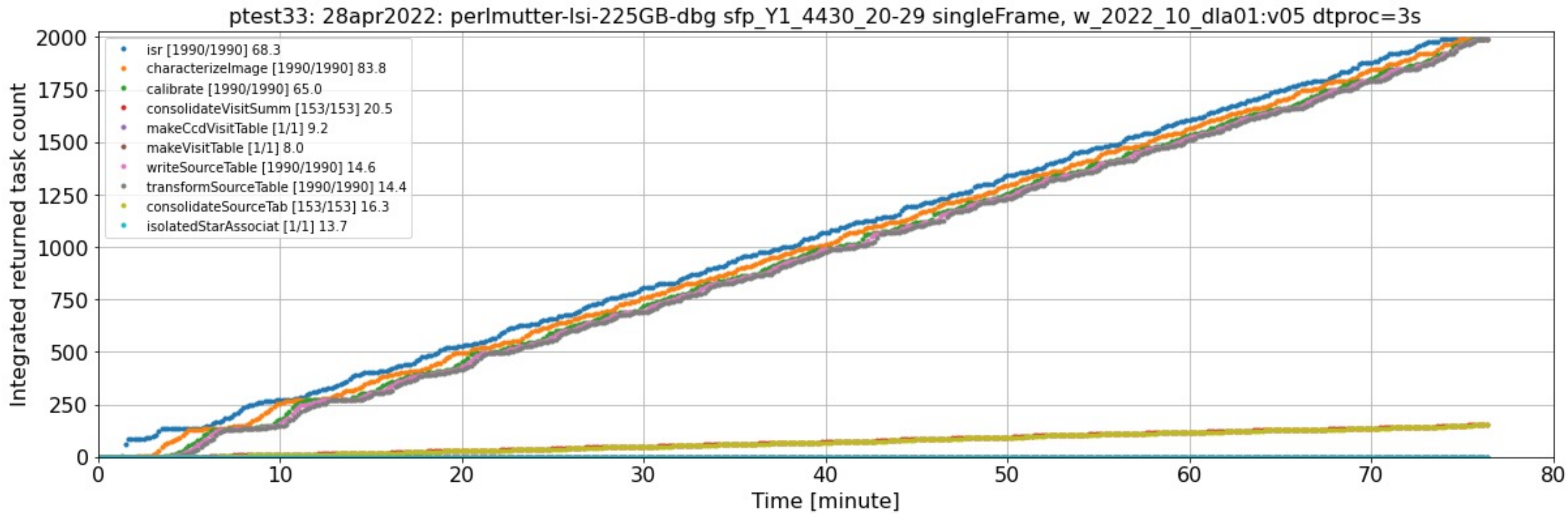
Throughput for singleFrame

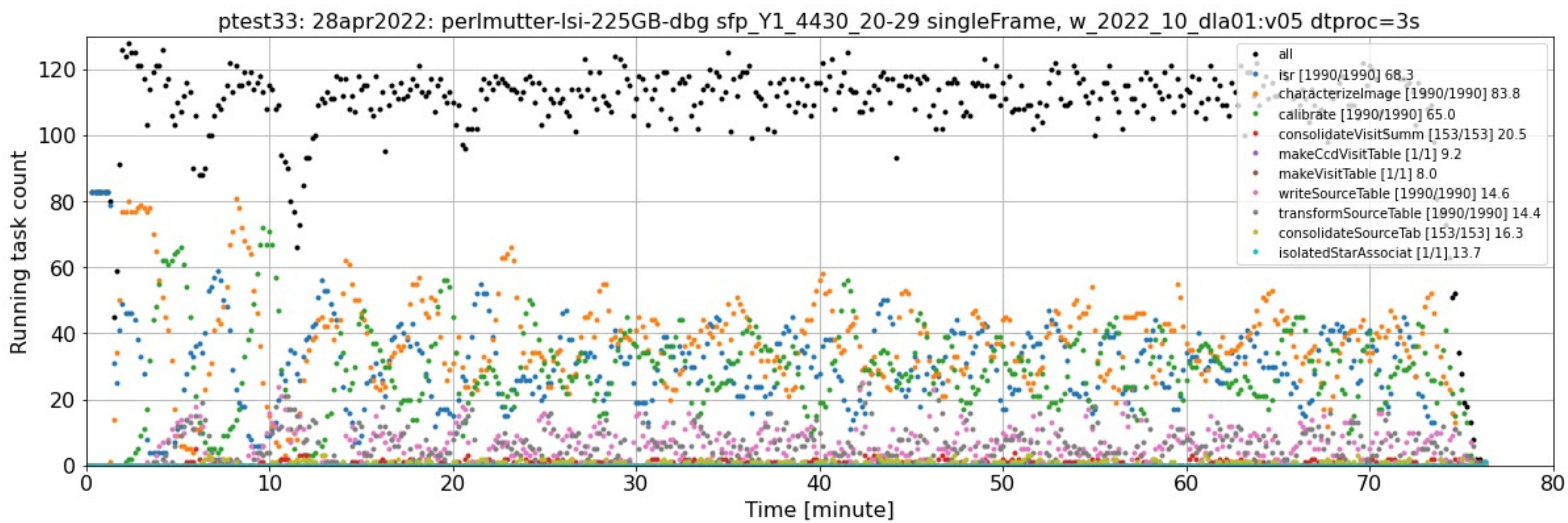


Above is an example throughput plot

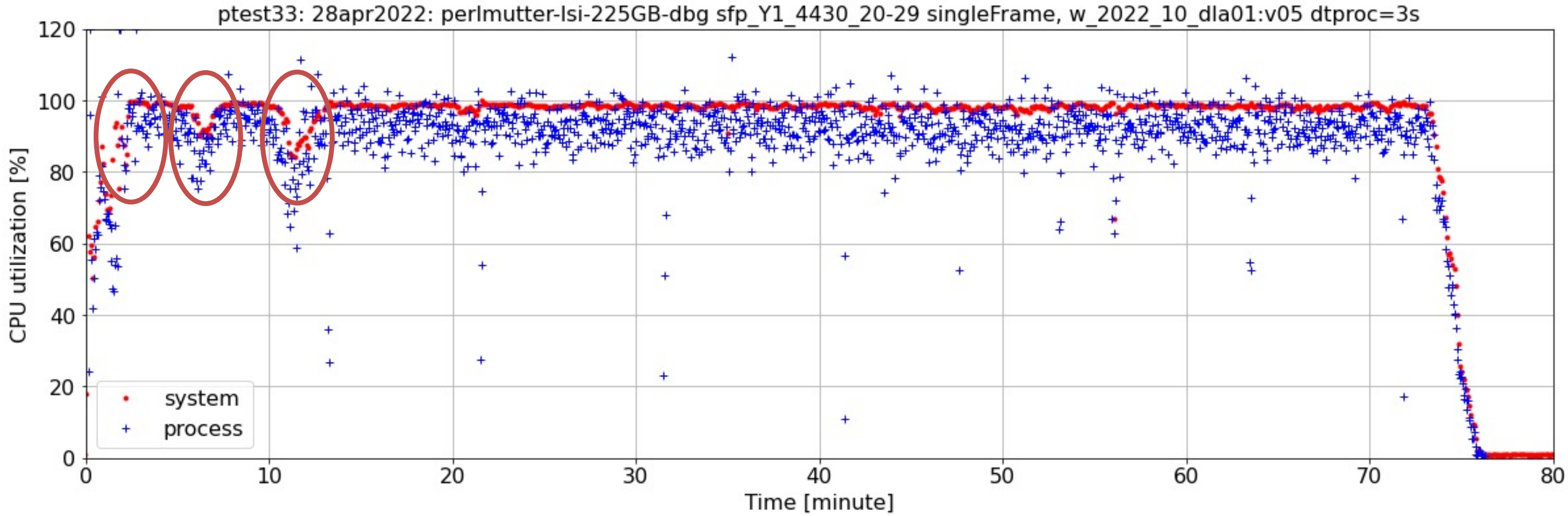
- Integrated numbers of launched, started and complete tasks vs time
  - From process monitor try table
- This is singleFrame processing for 10 patches
- Processing took 76 minutes
  - Start up time (7 min??) not included
  - Finalization (recording output in DB) not included (5 min??)

Throughput and # running tasks by task type





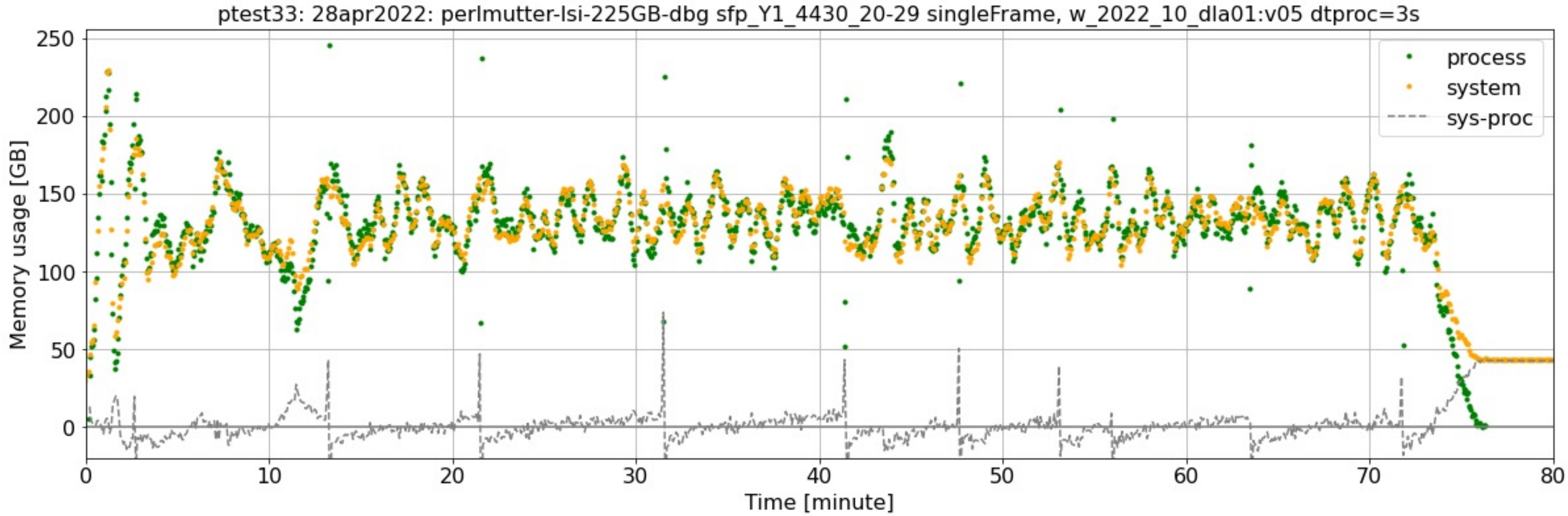
CPU utilization



Above plot shows CPU utilization vs. time

- Fraction of CPU cycles used on behalf of user
  - Red is the system monitor
  - Blue is the sum over running process from the parsl process monitor
- Some of the difference between these comes from cycles used by the scheduler and the monitor
- Generally close to 100% (good) except
  - Inevitable tail as jobs finish with none remaining to start
  - At start of processing, limited by high-memory isr jobs
  - Dips at 2, 7 and 11 minutes to be understood (more on this later)

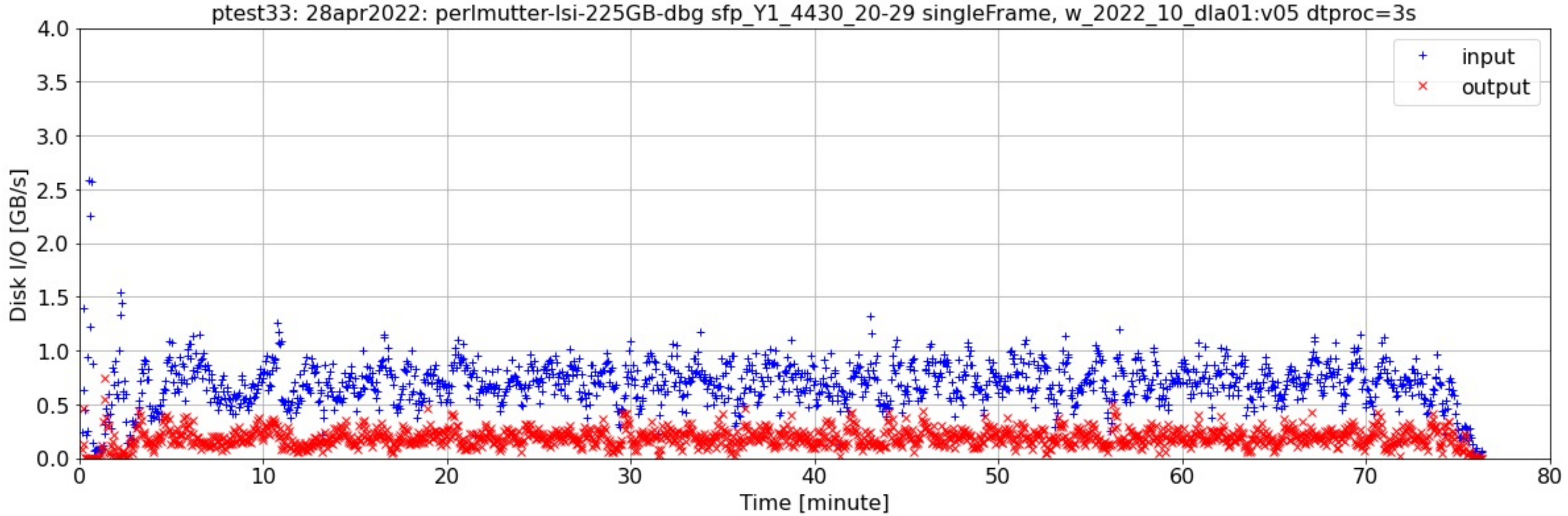
Memory usage



This plot show memory usage vs. time

- Again system report and sum of running tasks are shown
  - Gray is the difference of these
- Spikes likely due to catching extra samples in the rebinning needed for process sums. Maybe a better algorithm would help.
- We are comfortably in memory range (top of plot) except at beginning where synchronized jobs are growing together

Disk I/O

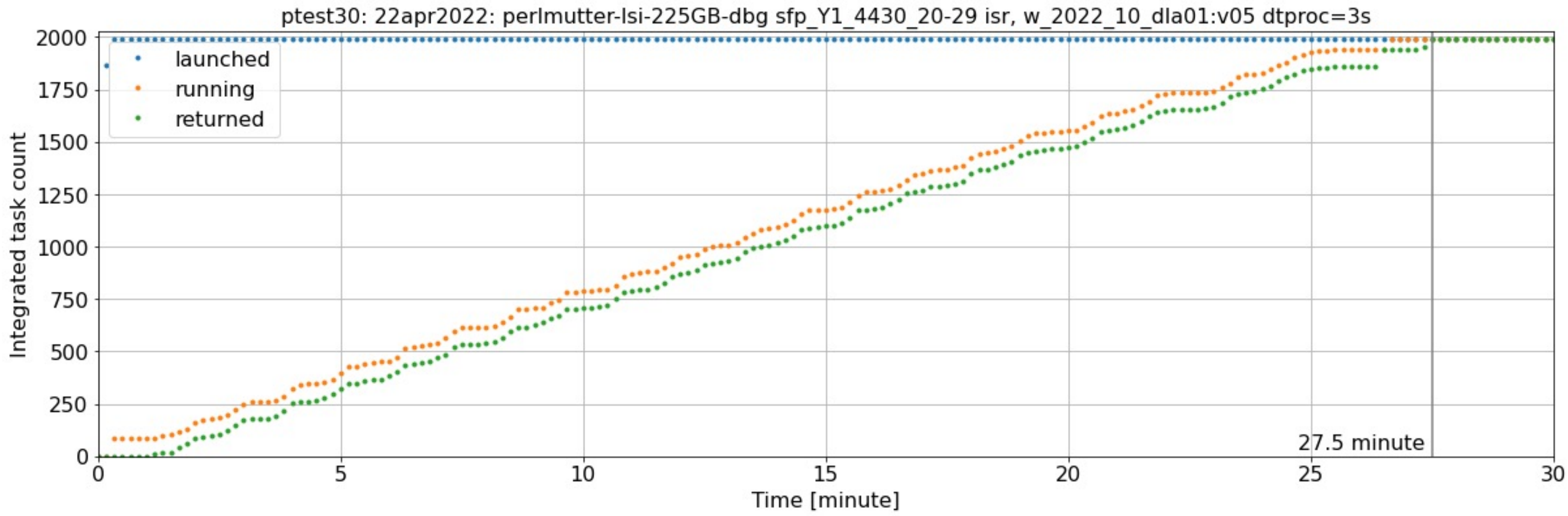


Above plot shows I/O vs time

- Process sums only
- System shows nothing
  - Because disks are network mounted?

Performance for isr (first task) only

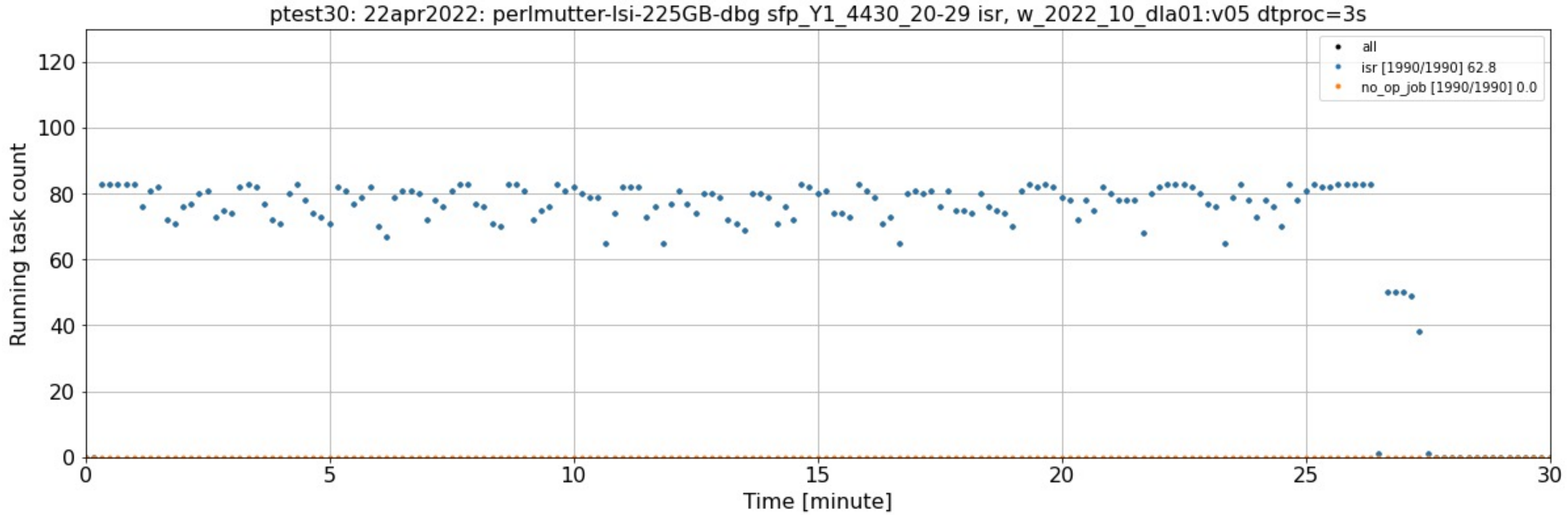
Throughput for isr only



Above plot is throughput

- Stairstep effect suggesting tasks may be stalling on a shared resource
  - Disk, input file location DB, processor speed, …

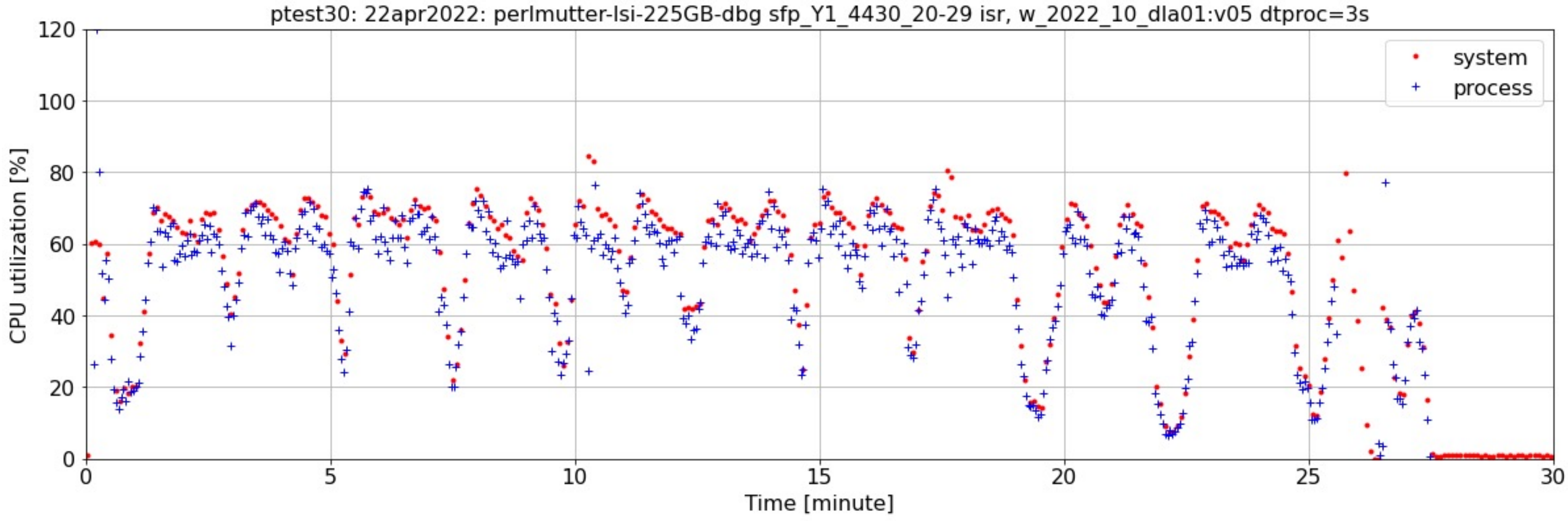
Number of running tasks



Plot show number of running tasks vs. time

- Ideal is memory limit (225 GB)/(2.7 GB/task) = 83 tasks
- Typically 10-15% below this value
    - Parsl is not keeping up?

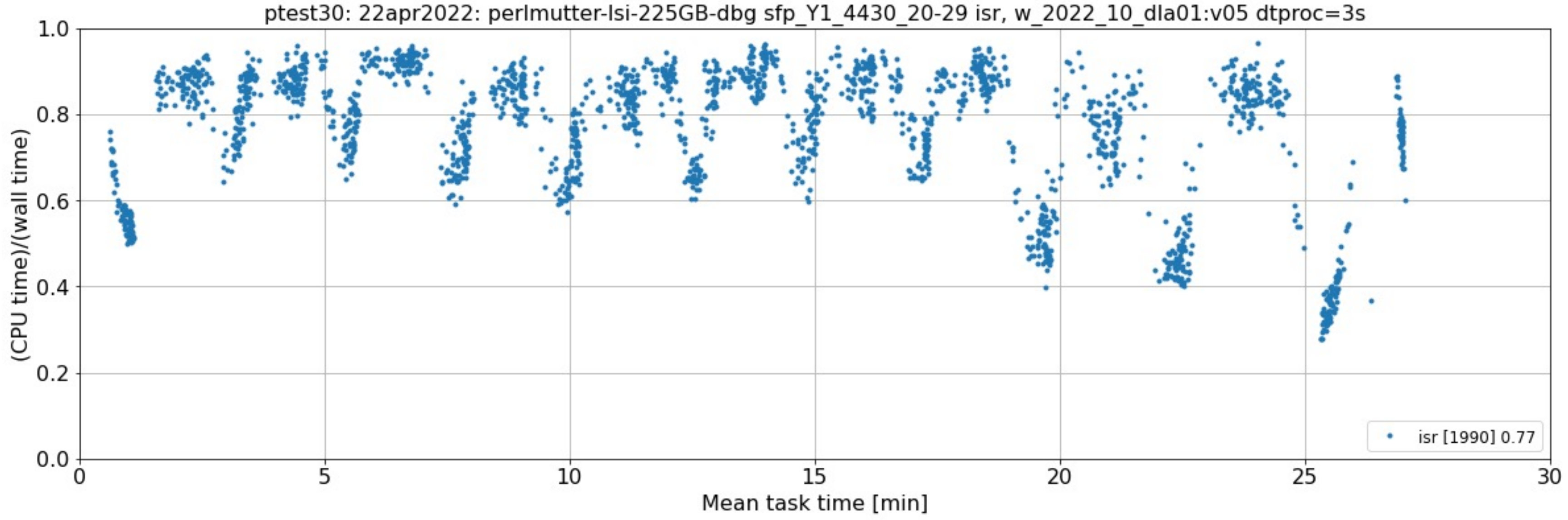
CPU utilization



Plot shows CPU utilization vs. time

- Many dips, some severe
- There are time periods when tasks are starved for a common resource

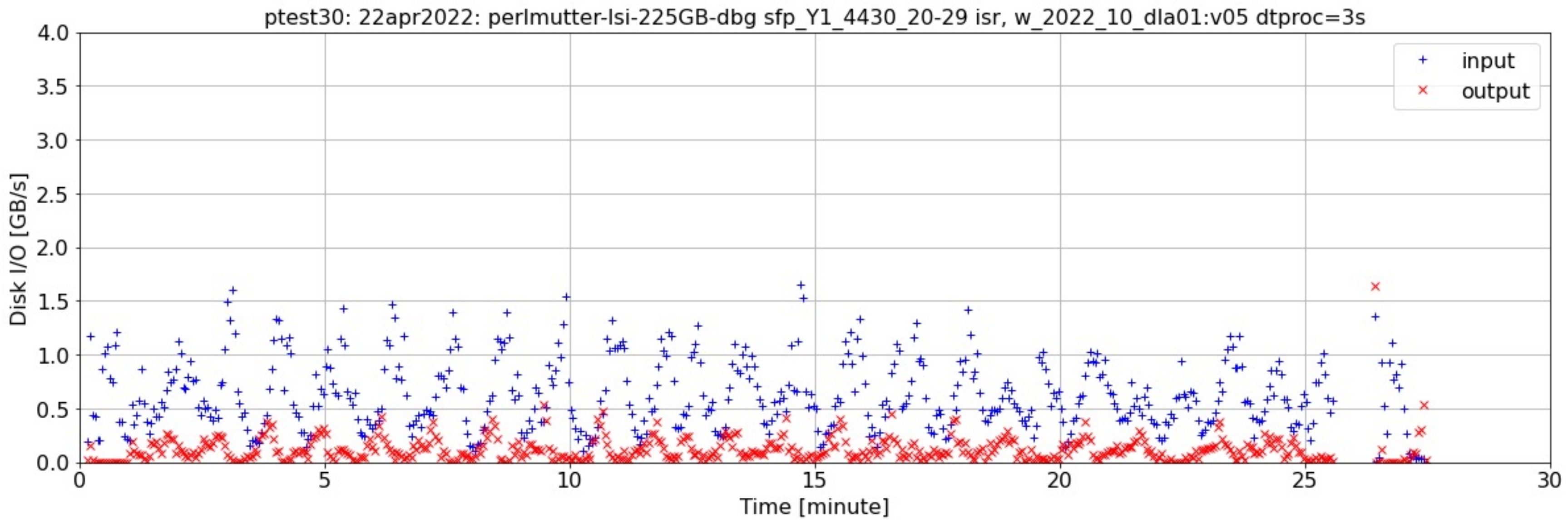
Task CPU utilization



Plot shows a different measure of CPU utilization

- Now one point for each task
  - X-axis is the mean of the process start and end times
- Data obtained from the logging monitor

Disk I/O



Plot show disk I/O

- Some correlation with preceding plots
- But hard to separate cause and effect

Varying the number of concurrent processes

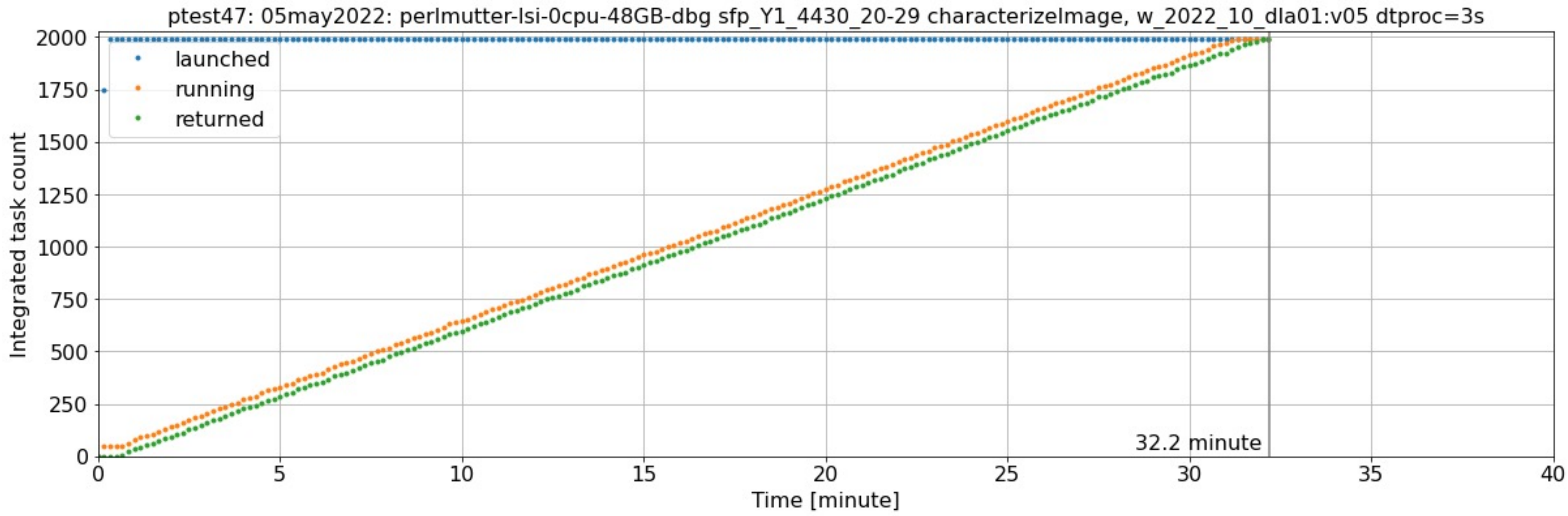
Varying the number of processes

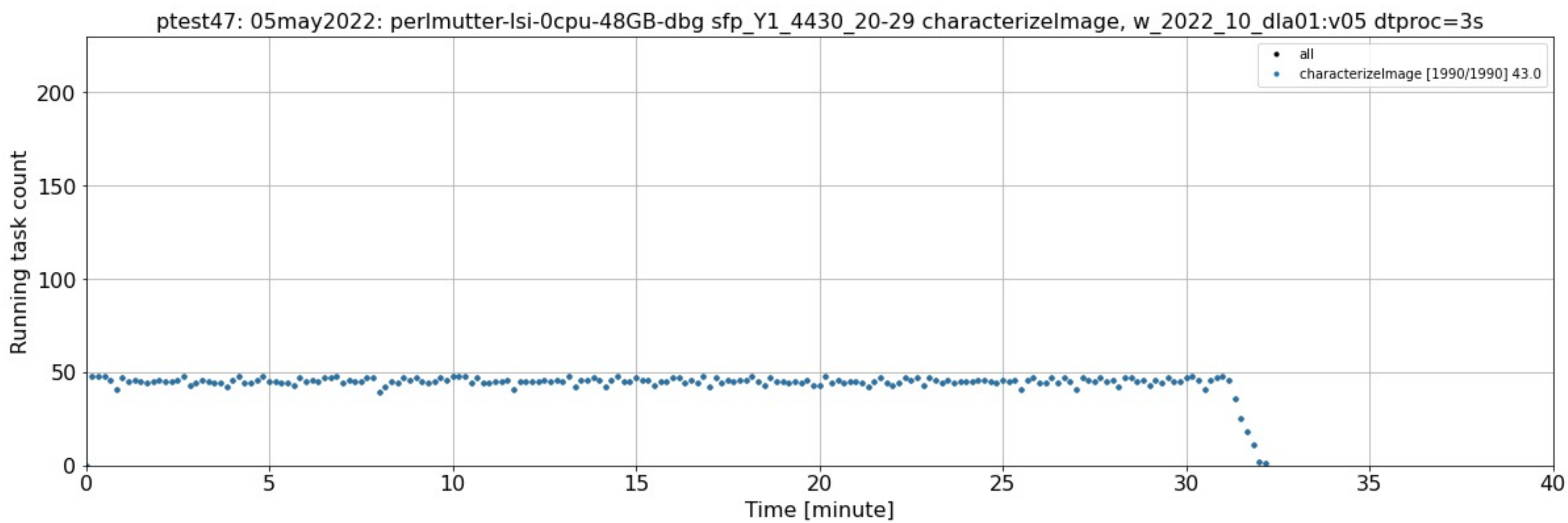
What happens if we run more or fewer concurrent processes?

Consider an example:

- One task, characterizeImage, with 10 patches
- Fix the number nominal of concurrent processes to N with
  - E.g. with: 0 CPU/task, task memory=1 GB, total memory = N GB
  - Other configs used in some cases
- Following pages show throughput (top) and actual number of running tasks as this nominal task count is increased
  - Remember we have 128/64 logical/physical cores

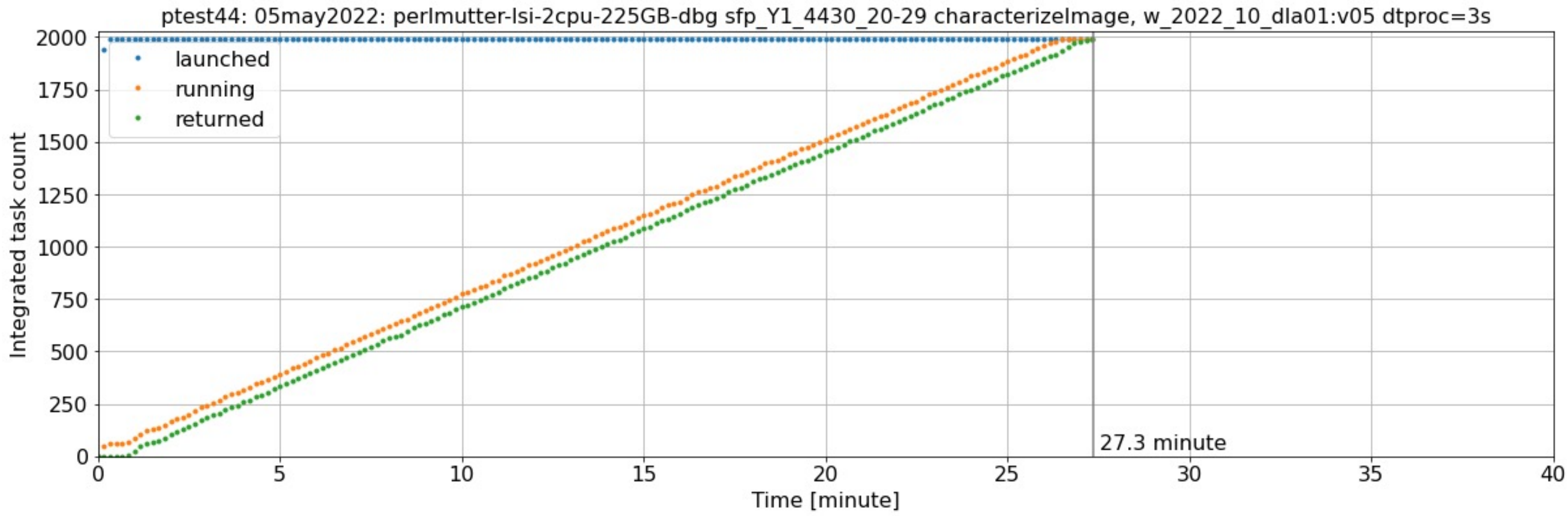
48 tasks

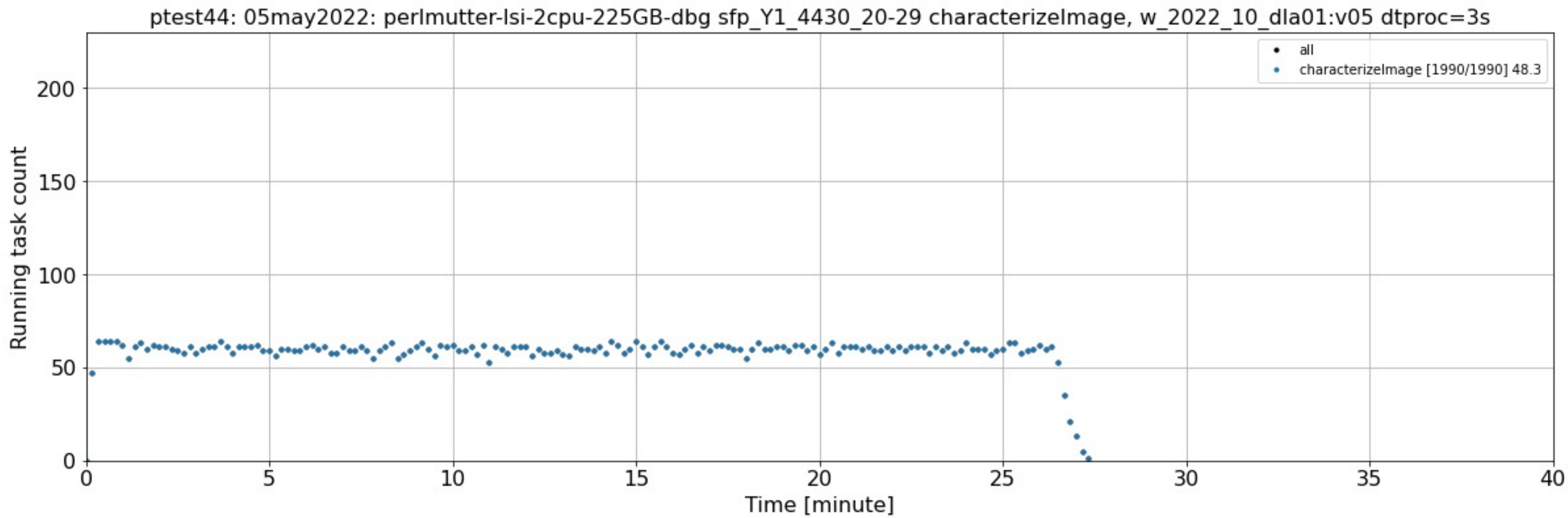

ptest47: 05may2022: perlmutter-lsi-0cpu-48GB-dbg sfp_Y1_4430_20-29 characterizeImage, w_2022_10_dla01:v05 dtproc=3s
launched
running
returned
Integrated task count
Time [minute]
32.2 minute


ptest47: 05may2022: perlmutter-lsi-0cpu-48GB-dbg sfp_Y1_4430_20-29 characterizeImage, w_2022_10_dla01:v05 dtproc=3s
all
characterizeImage [1990/1990] 43.0
Running task count
Time [minute]

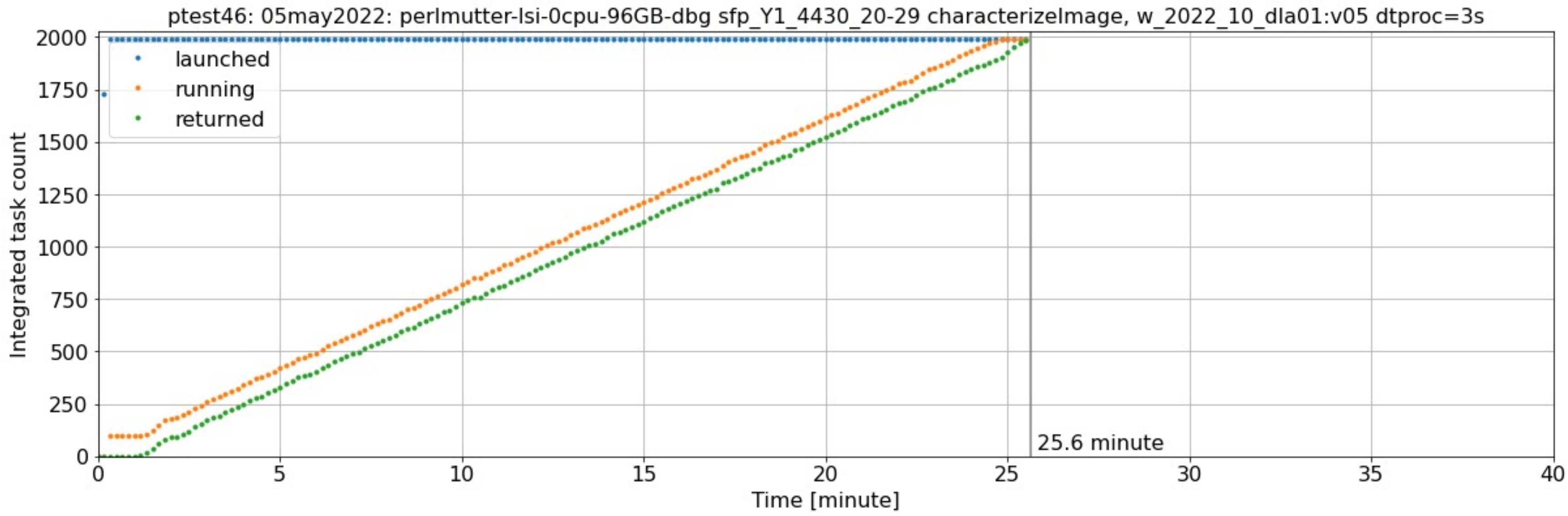
64 tasks

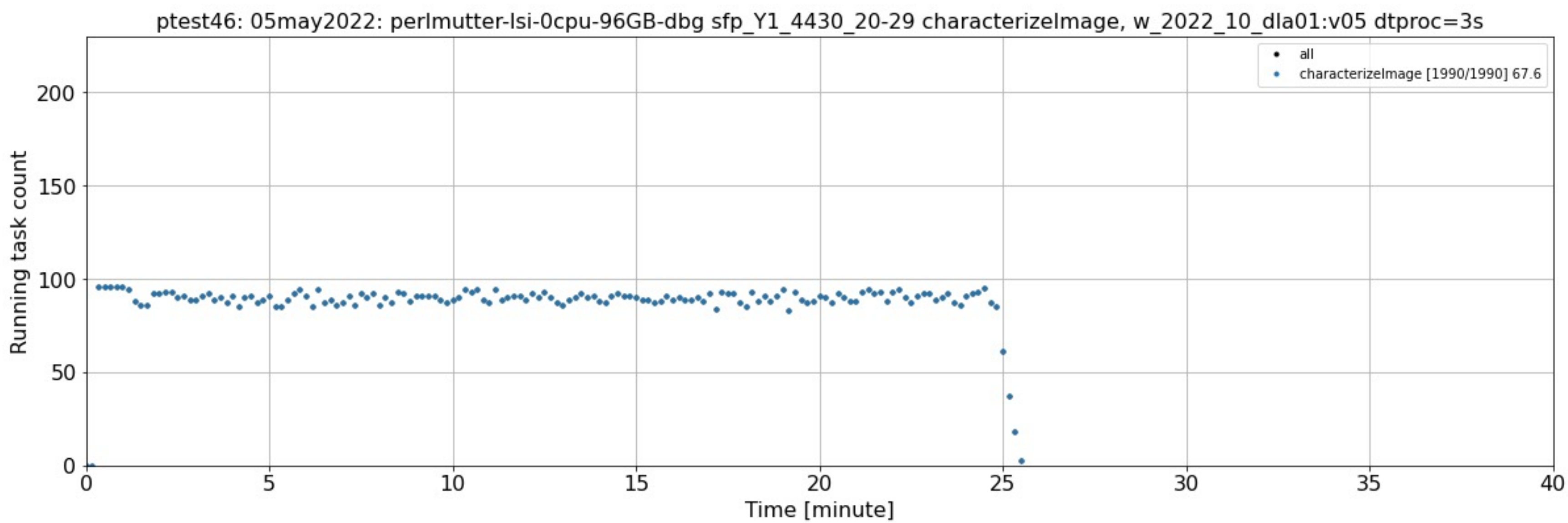
1 task/(physical core)





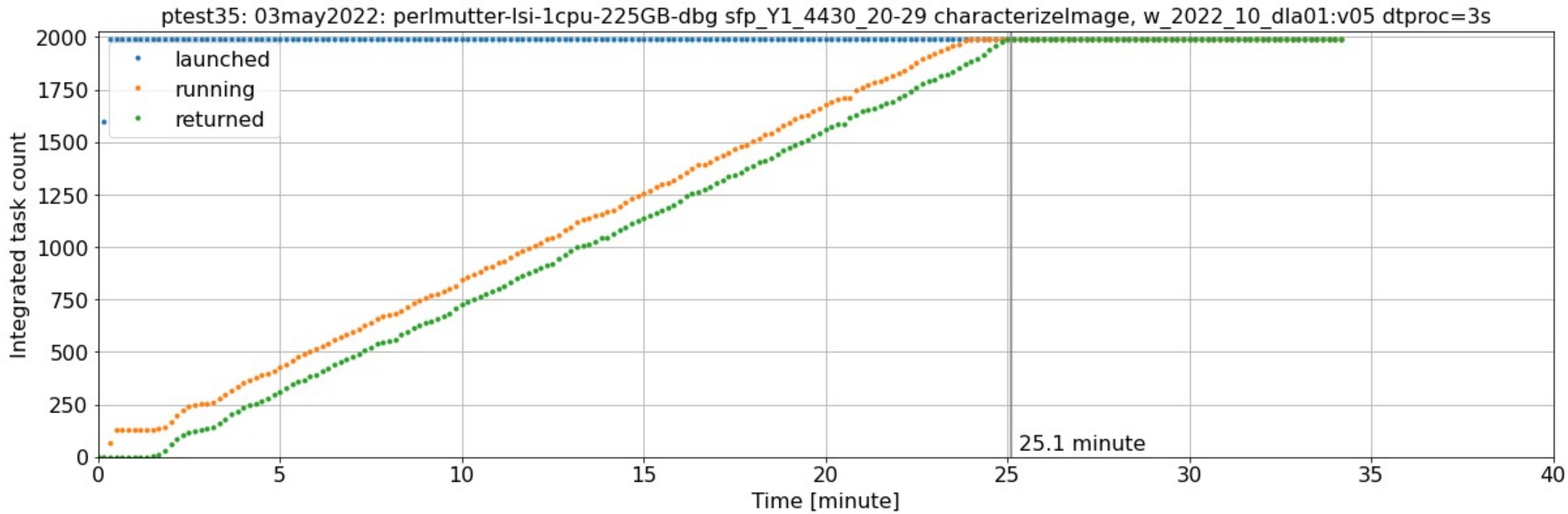
96 tasks

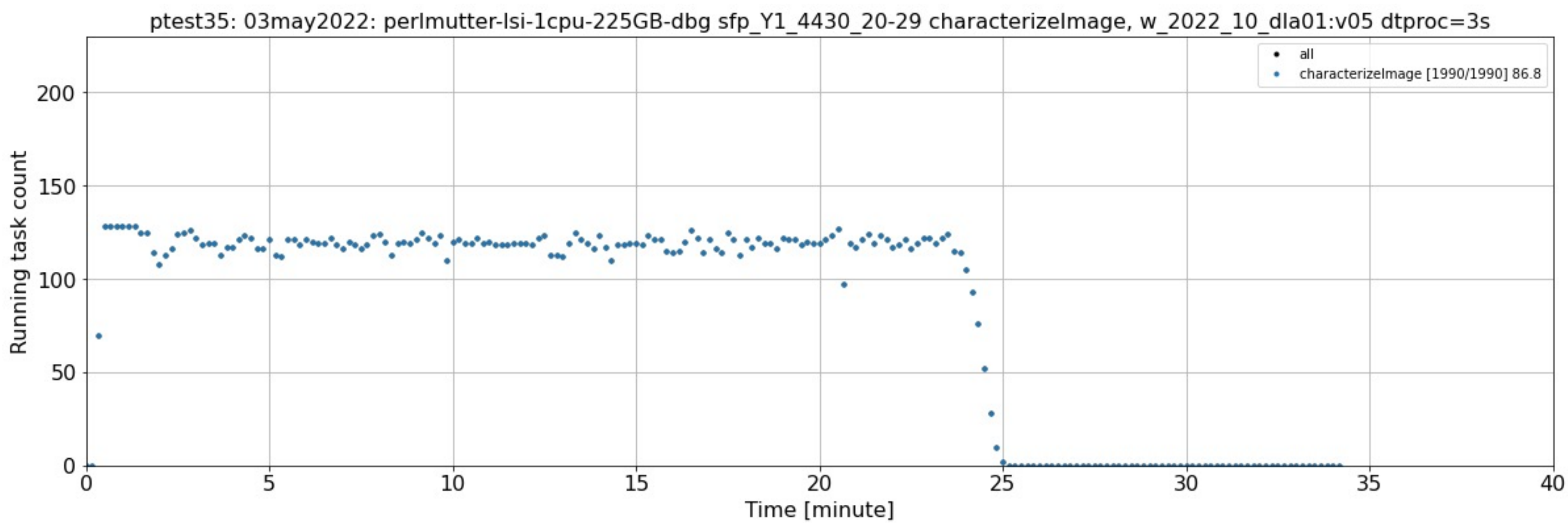

ptest46: 05may2022: perlmutter-lsi-0cpu-96GB-dbg sfp_Y1_4430_20-29 characterizeImage, w_2022_10_dla01:v05 dtproc=3s
launched
running
returned
Integrated task count
25.6 minute
Time [minute]


ptest46: 05may2022: perlmutter-lsi-0cpu-96GB-dbg sfp_Y1_4430_20-29 characterizeImage, w_2022_10_dla01:v05 dtproc=3s
all
characterizeImage [1990/1990] 67.6
Running task count
Time [minute]

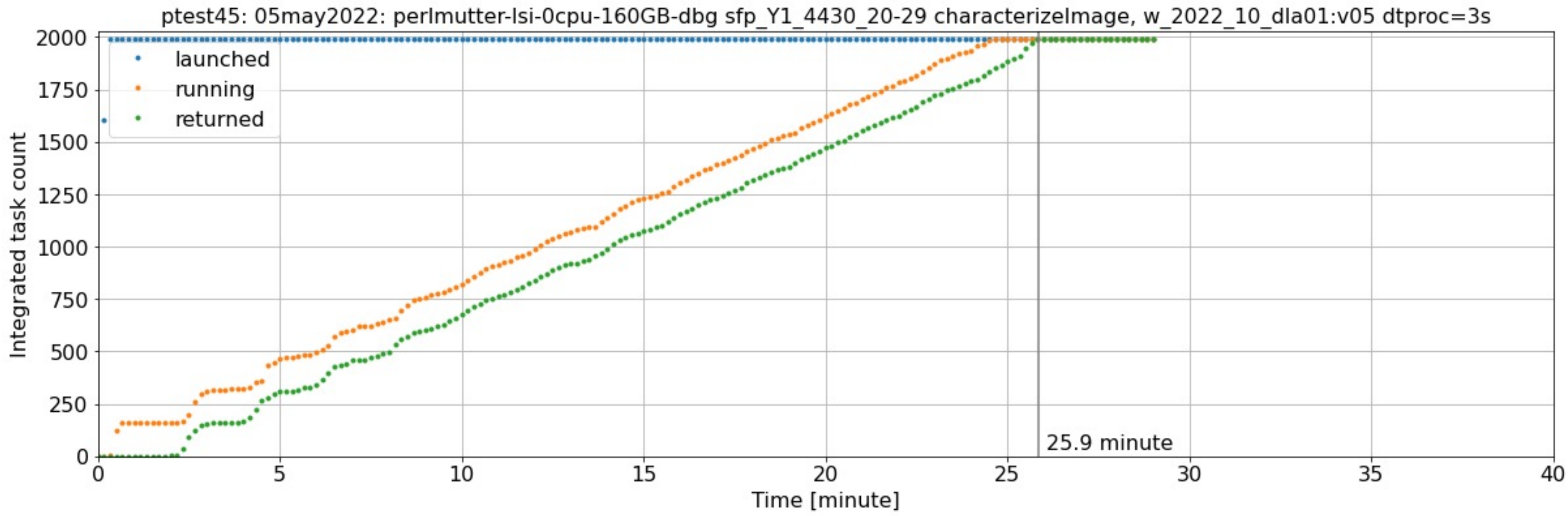
128 tasks

1 task/(physical core)

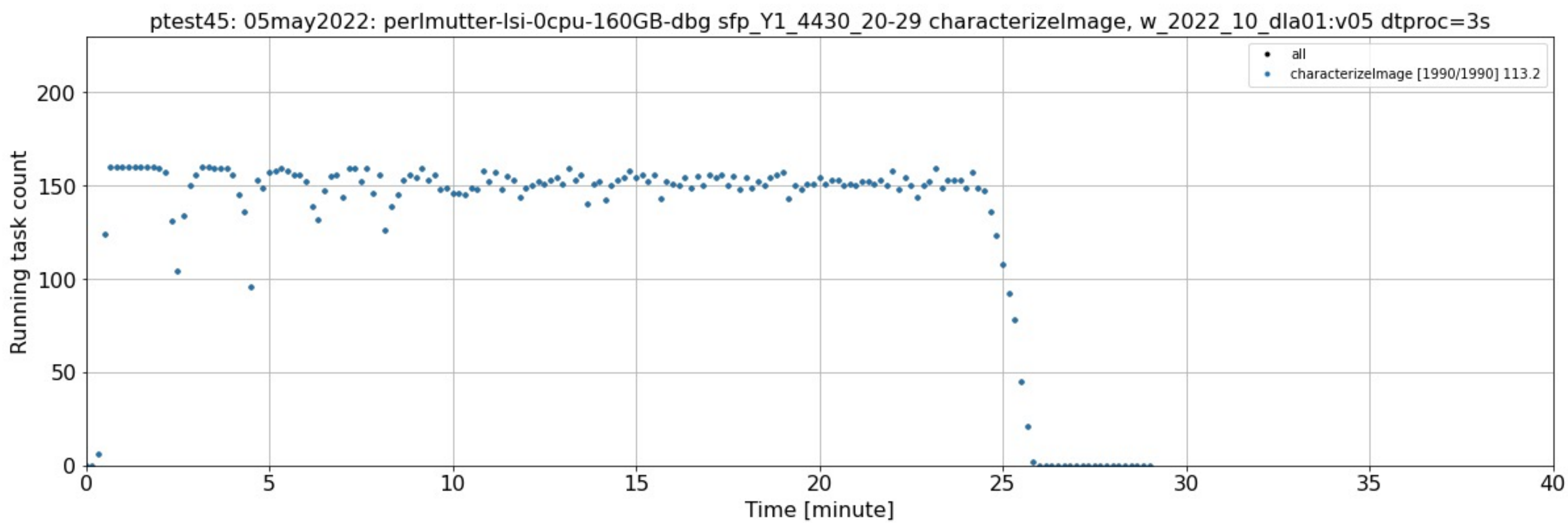




160 tasks

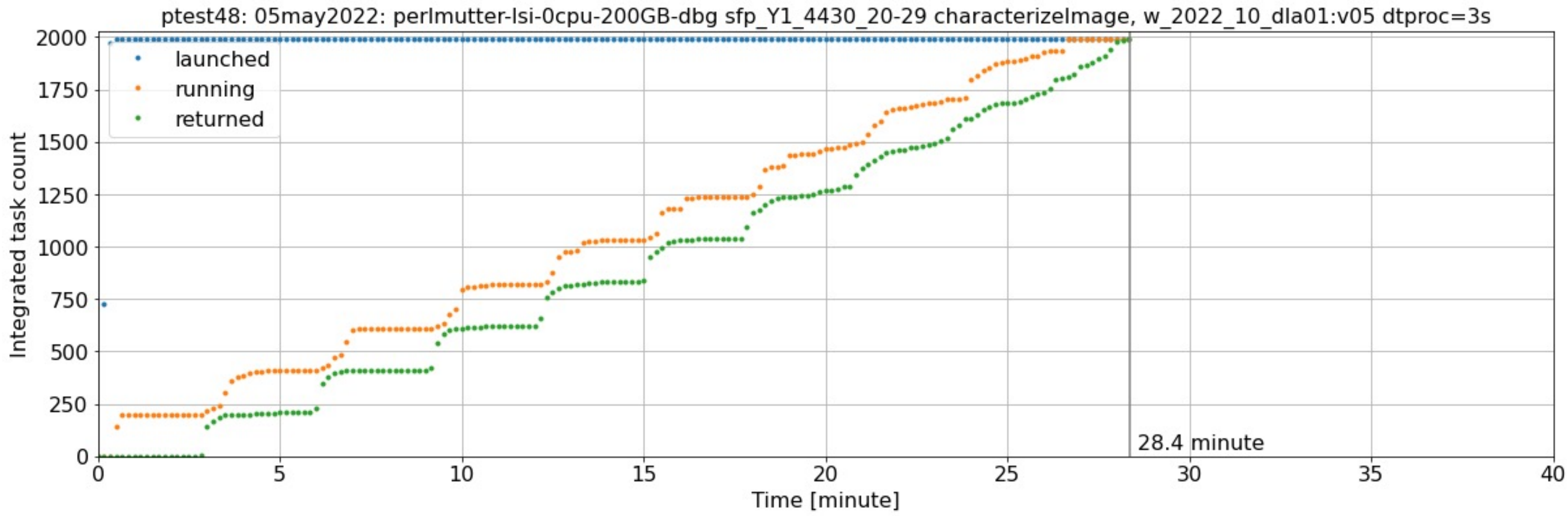


ptest45: 05may2022: perlmutter-lsi-0cpu-160GB-dbg sfp_Y1_4430_20-29 characterizeImage, w_2022_10_dla01:v05 dtproc=3s
launched
running
returned
Integrated task count
2000
1750
1500
1250
1000
750
500
250
0
25.9 minute
0
5
10
15
20
25
30
35
40
Time [minute]

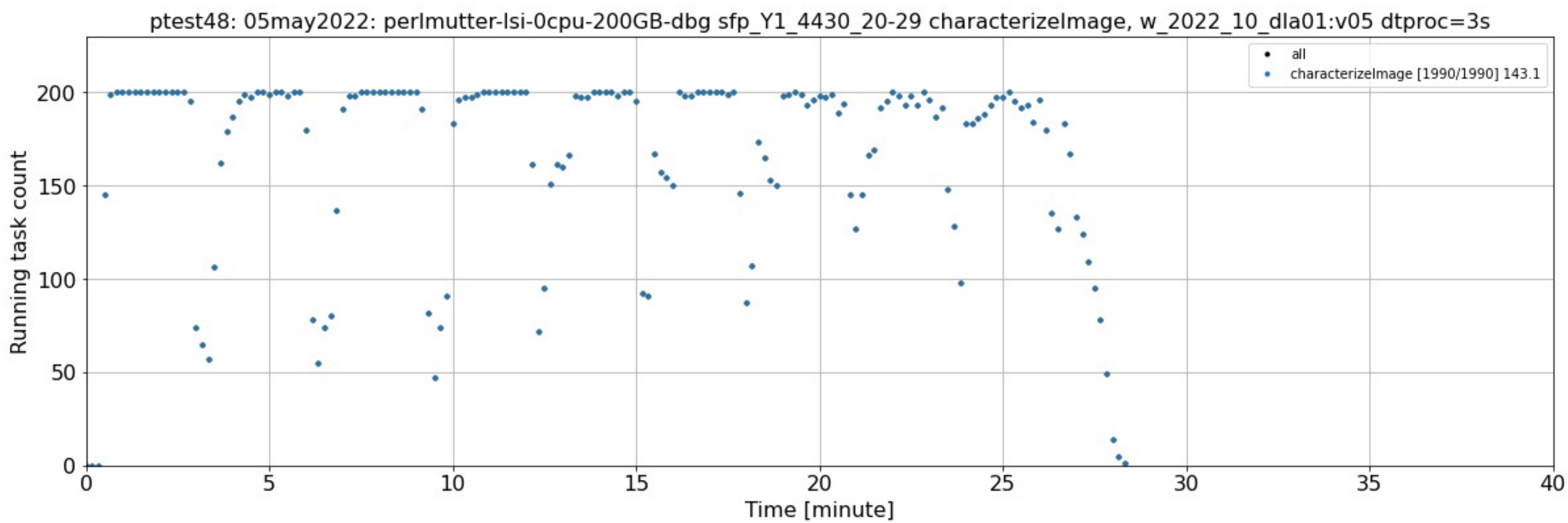


ptest45: 05may2022: perlmutter-lsi-0cpu-160GB-dbg sfp_Y1_4430_20-29 characterizeImage, w_2022_10_dla01:v05 dtproc=3s
all
characterizeImage [1990/1990] 113.2
Running task count
200
150
100
50
0
0
5
10
15
20
25
30
35
40
Time [minute]

200 tasks

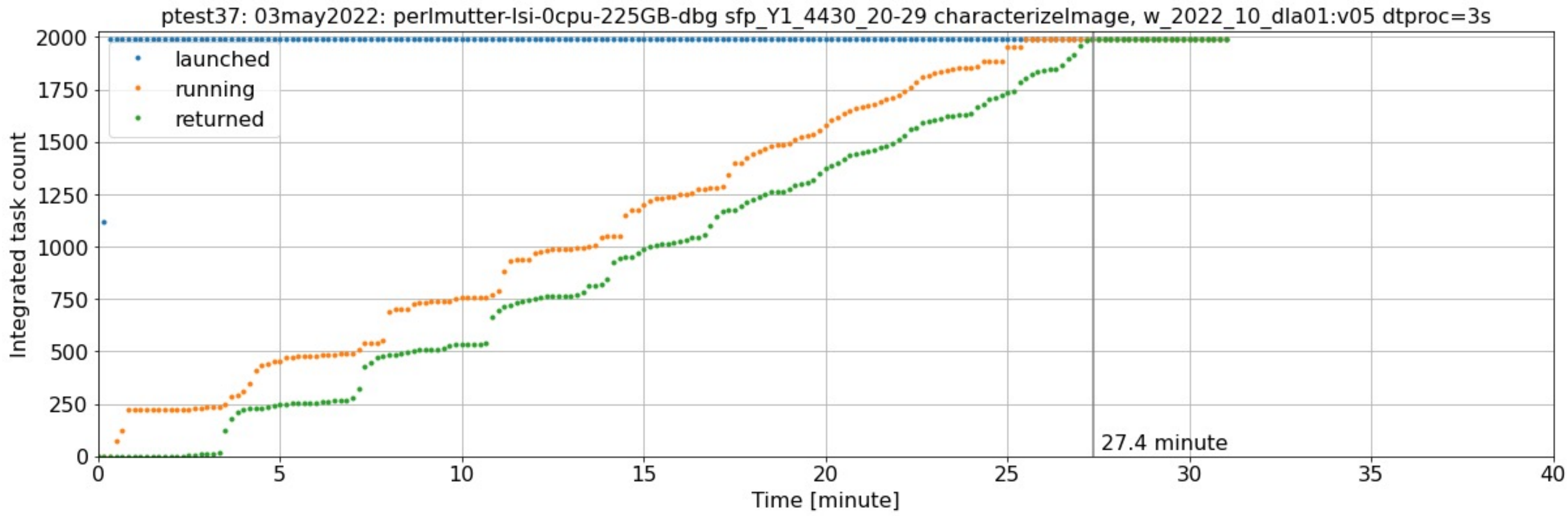


ptest48: 05may2022: perlmutter-lsi-0cpu-200GB-dbg sfp_Y1_4430_20-29 characterizeImage, w_2022_10_dla01:v05 dtproc=3s
launched
running
returned
28.4 minute
Integrated task count
Time [minute]

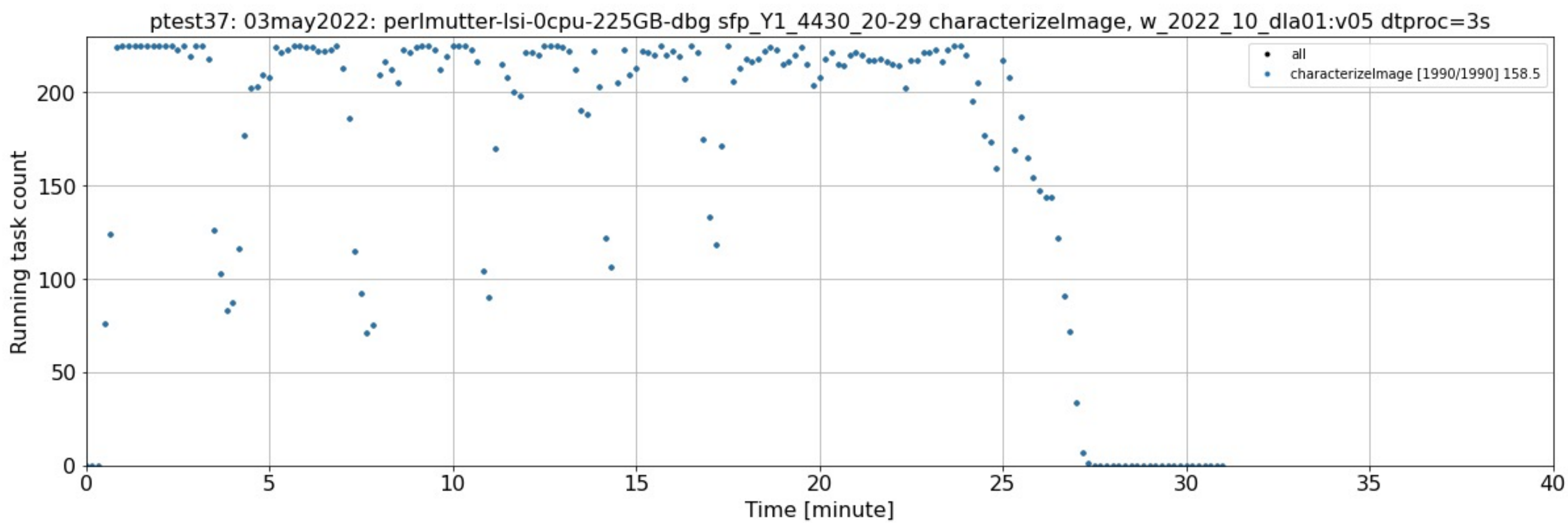


ptest48: 05may2022: perlmutter-lsi-0cpu-200GB-dbg sfp_Y1_4430_20-29 characterizeImage, w_2022_10_dla01:v05 dtproc=3s
all
characterizeImage [1990/1990] 143.1
Running task count
Time [minute]

225 tasks



ptest37: 03may2022: perlmutter-lsi-0cpu-225GB-dbg sfp_Y1_4430_20-29 characterizeImage, w_2022_10_dla01:v05 dtproc=3s
launched
running
returned
Integrated task count
27.4 minute
Time [minute]



ptest37: 03may2022: perlmutter-lsi-0cpu-225GB-dbg sfp_Y1_4430_20-29 characterizeImage, w_2022_10_dla01:v05 dtproc=3s
all
characterizeImage [1990/1990] 158.5
Running task count
Time [minute]

Comments on varying number of processes

Results

- Best performance with ~1 task/(logical core)
- But only 8% better that 1/(physical core)

What is going on

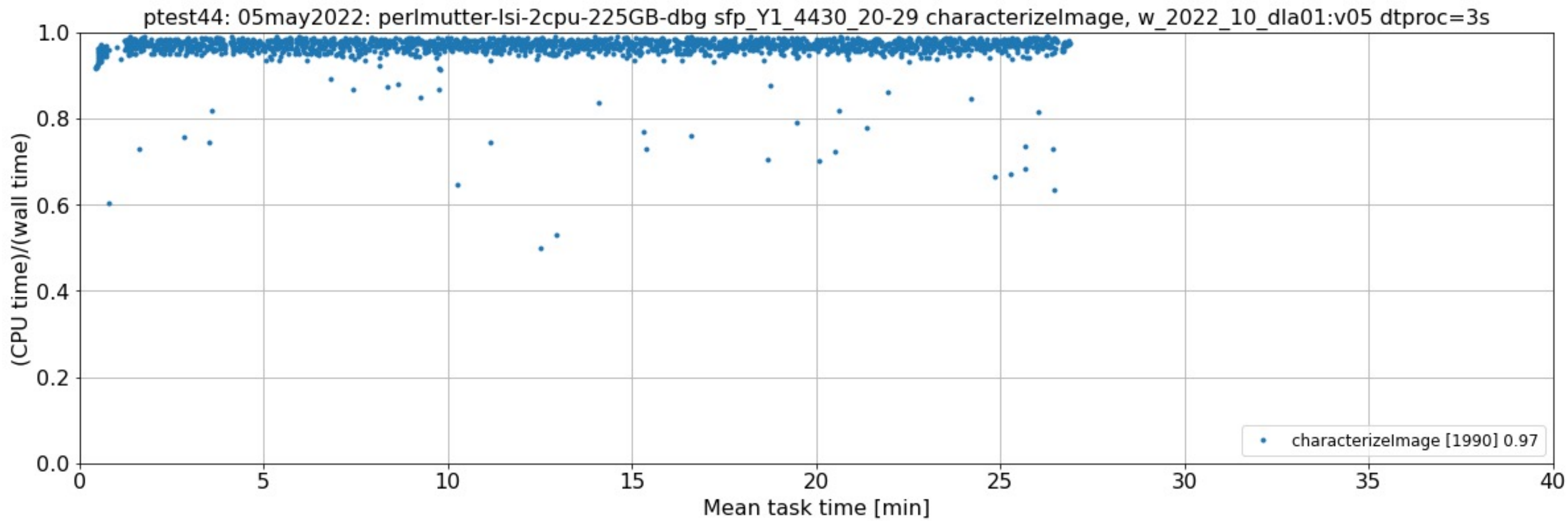
- Each physical core has
  - Two logical cores (hyperthreads)
    - Second process can execute instructions while the first is in a wait state
      - E.g. fetching data from main memory
  - Four pipes, i.e. can execute up to 4 instructions per cycle (IPC)
  - Varying clock speed
    - Slow down to save power if not used or if warm from heavy use
    - Nominal 2.8 GHz
- When multiple processes run on a core, each
  - Gets fewer cycles because OS is servicing (e.g. memory fetch) the other
  - Gets fewer IPS because the four are shared between the two hyperthreds
  - Gets fewer cycles because busier (hence warmer) processor slows done

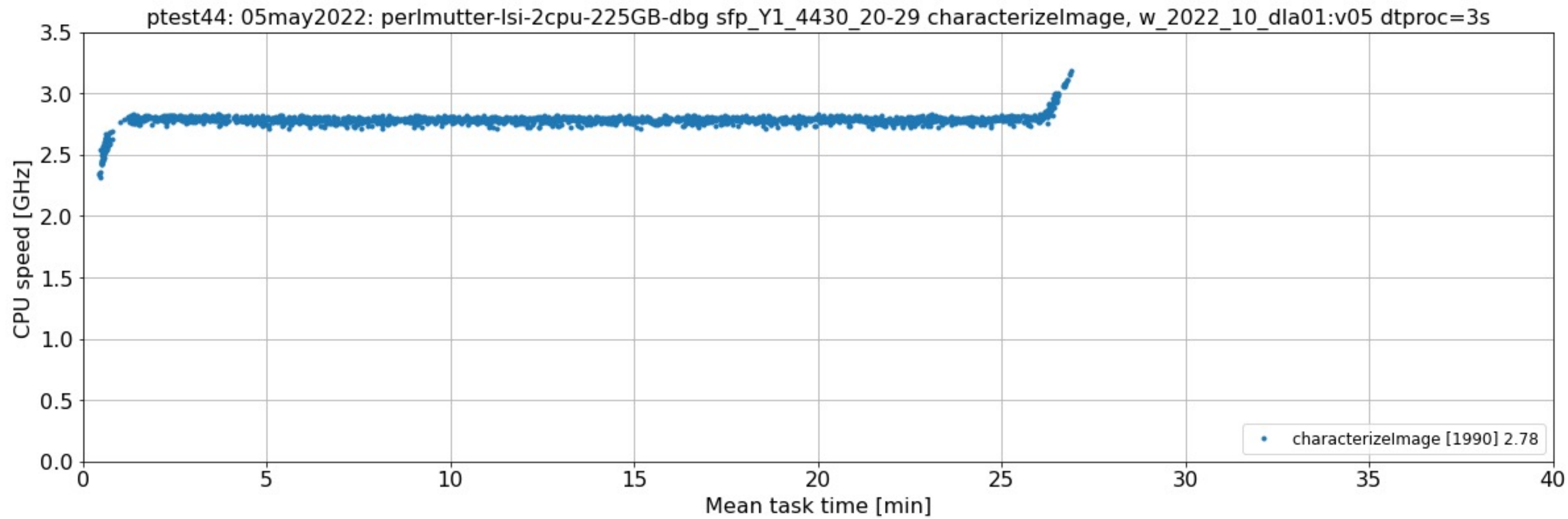
Performance plots

Following pages show plots of the preceding

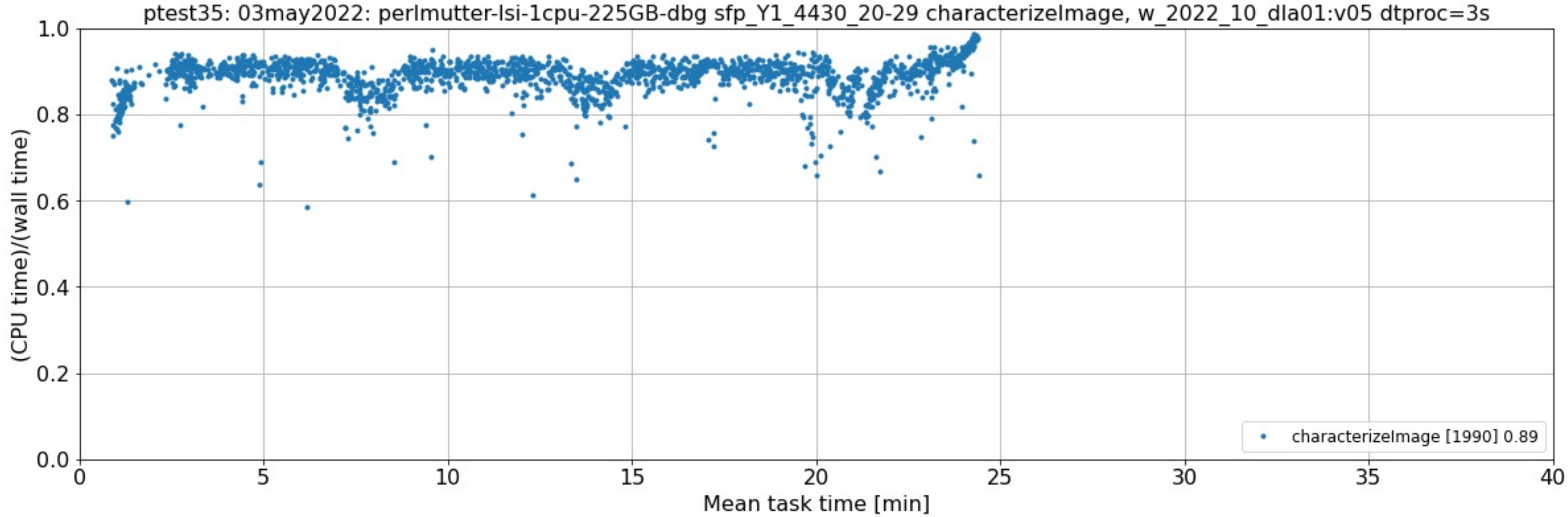
- All plots have one point per process
  - Mean value for the process is plotted
    - Not the mean/second
- For varying (nominal) number of tasks: 64, 128
  - I.e. 1 process/(physical core) and 1 process/(logical core)
  - Each on a separate page
- First pages
  - Top: CPU utilization = (active cycles)/(cycles)
    - fraction of cycles servicing the process
  - Bottom: CPU speed [cycles/(wall second)]
- Second pages
  - Top: IPC = instructions/(active cycle) used for the process
    - Two logical process share four physical pipes
  - Bottom: IPS = instructions/(wall second)
    - Product of the of the preceding three
    - Throughput is proportional to <number of processes> × IPS

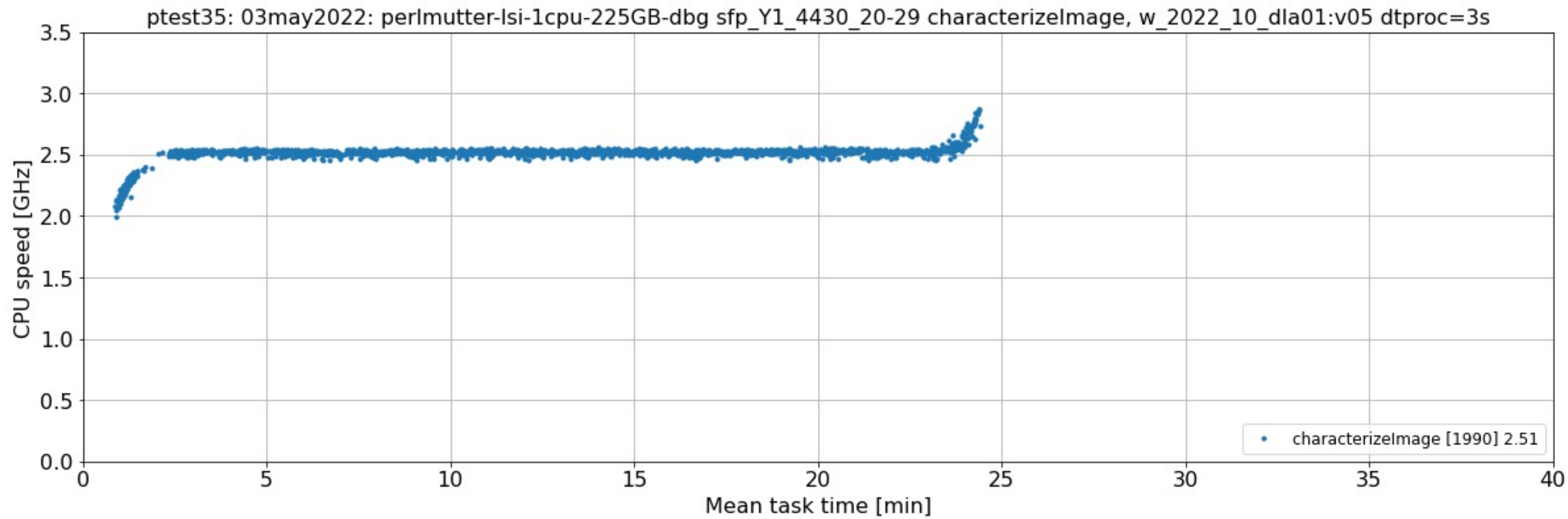
64 tasks: CPU utilization and speed



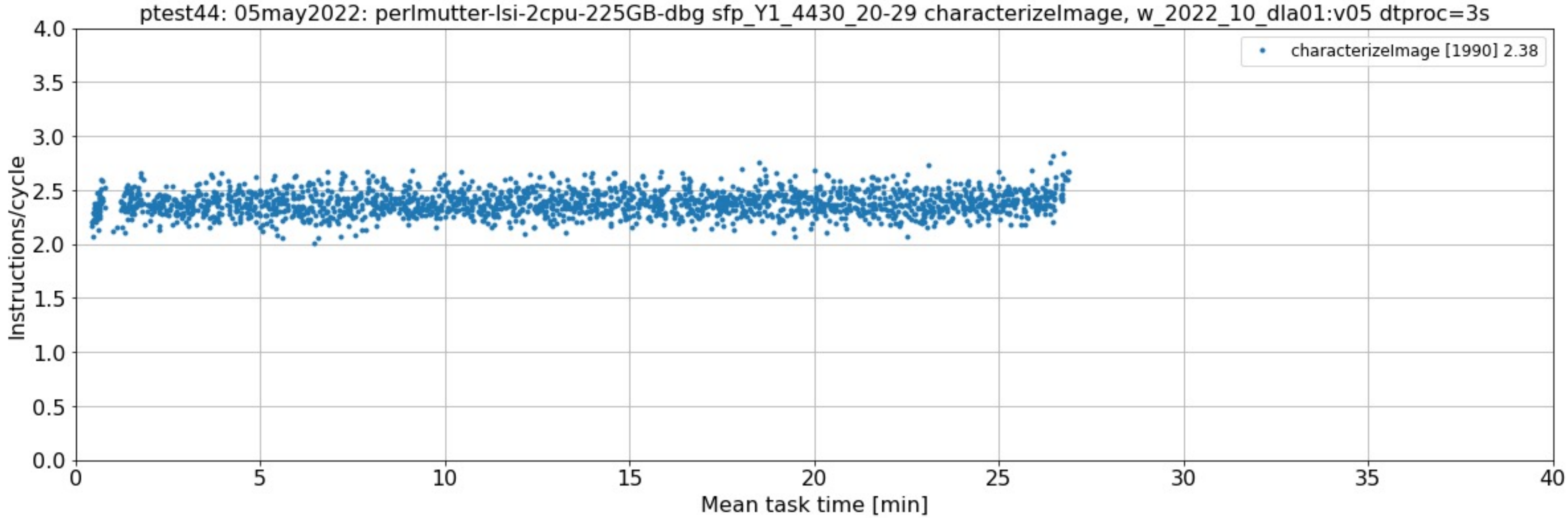


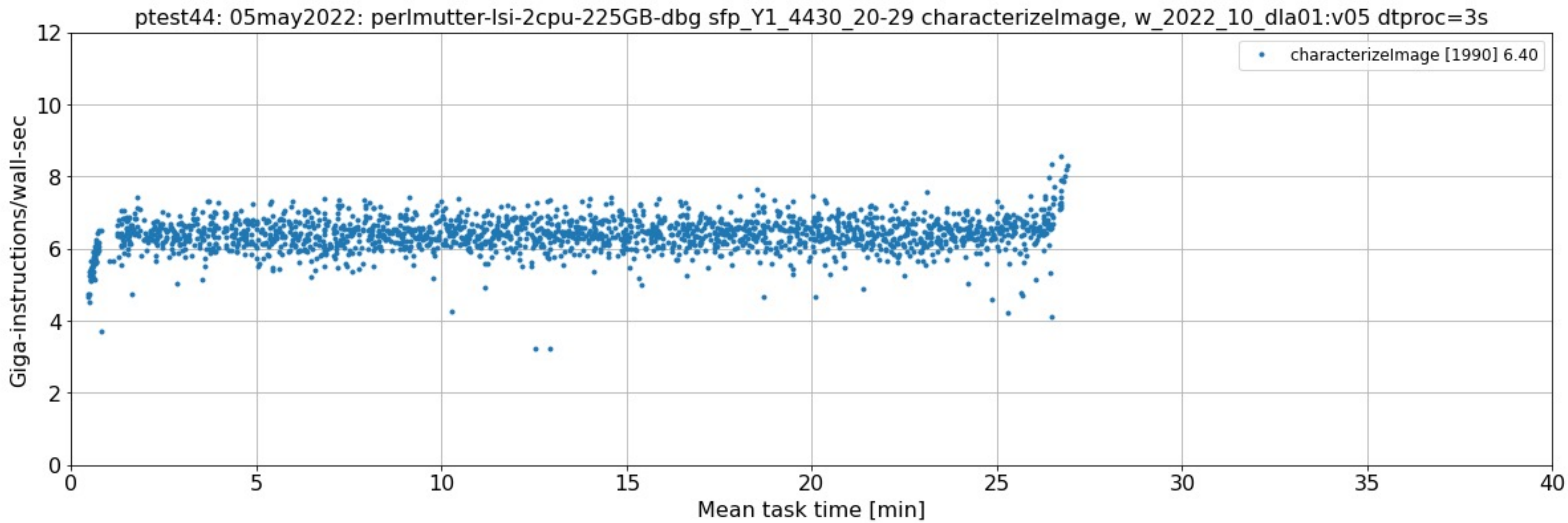
128 tasks: CPU utilization and speed



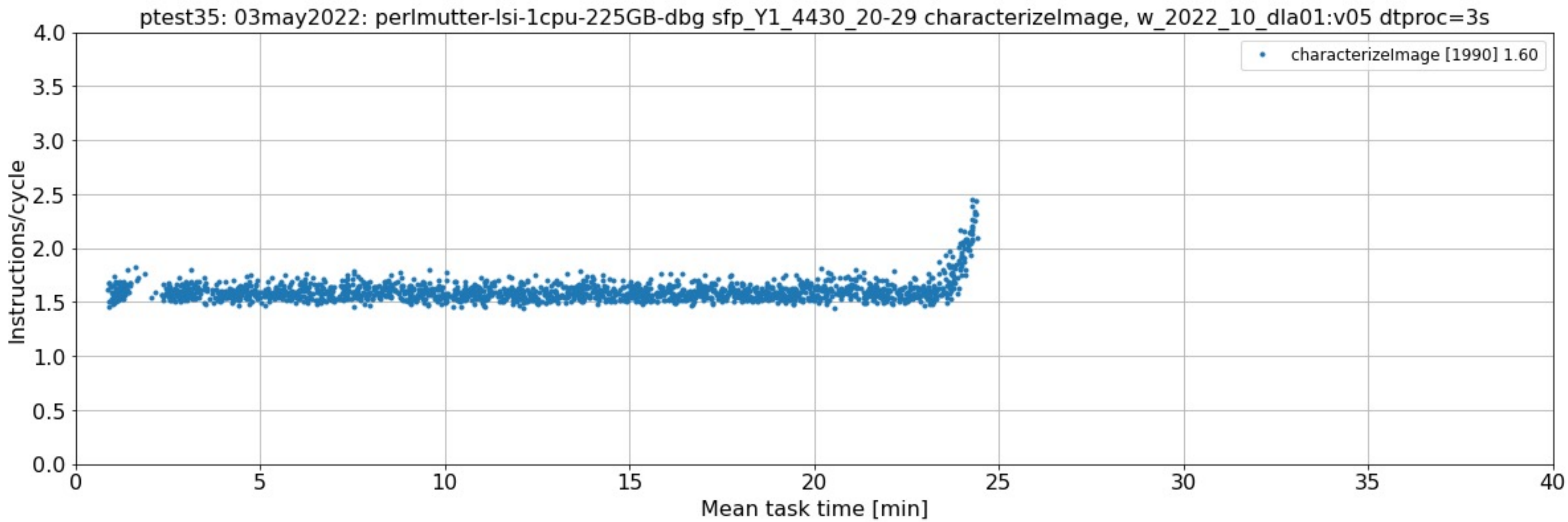


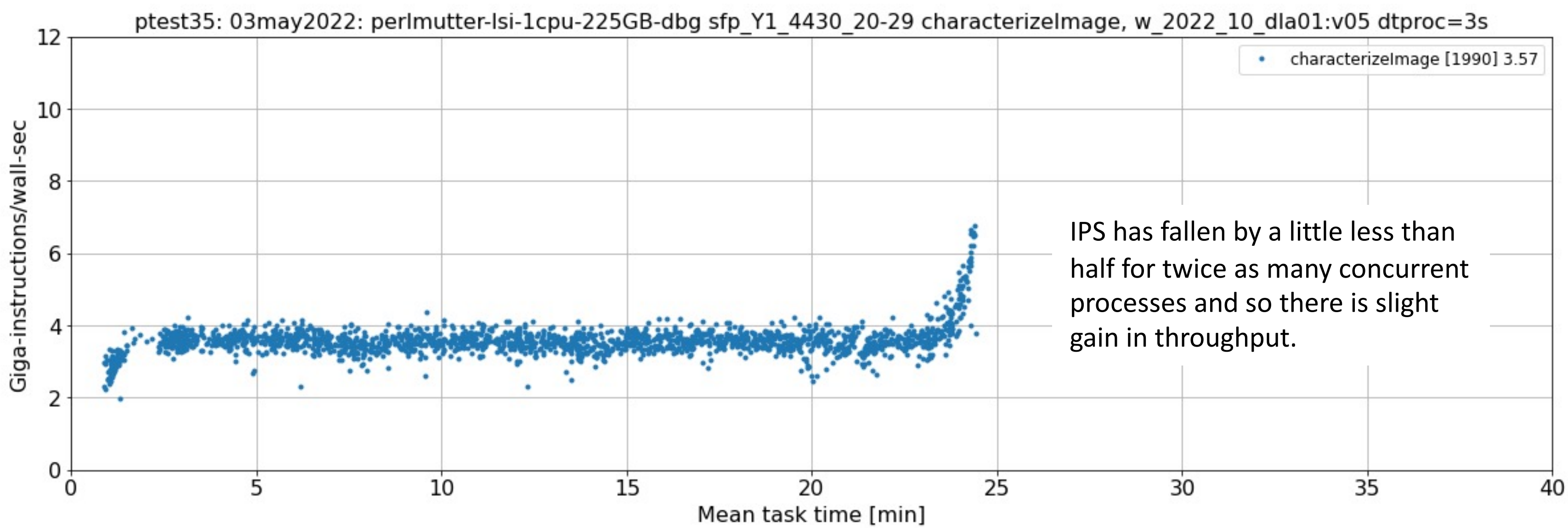
64 tasks: IPC and IPS





128 tasks: IPC and IPS

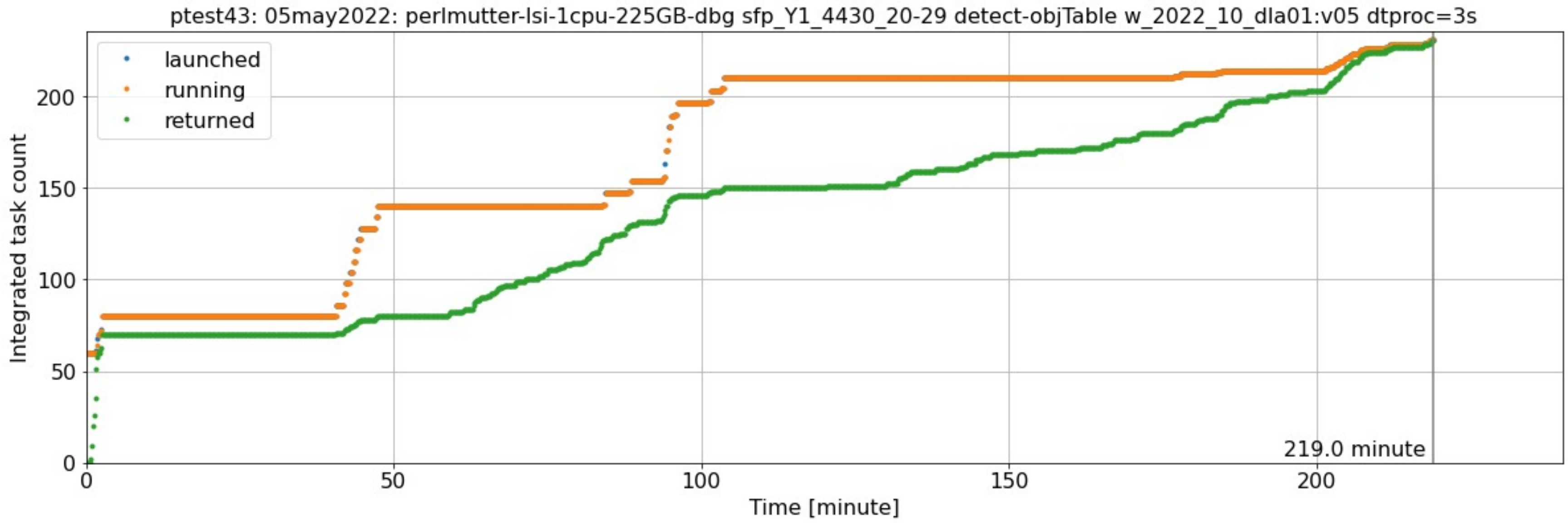




IPS has fallen by a little less than half for twice as many concurrent processes and so there is slight gain in throughput.

Patch processing performance

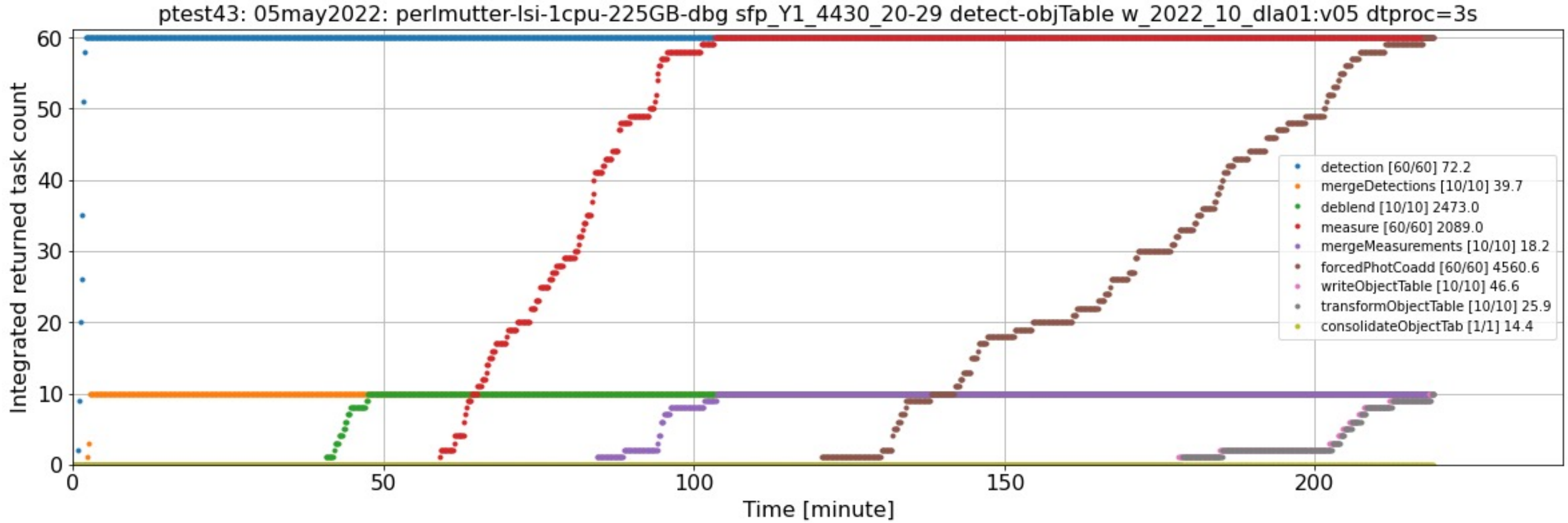
Throughput for patch processing

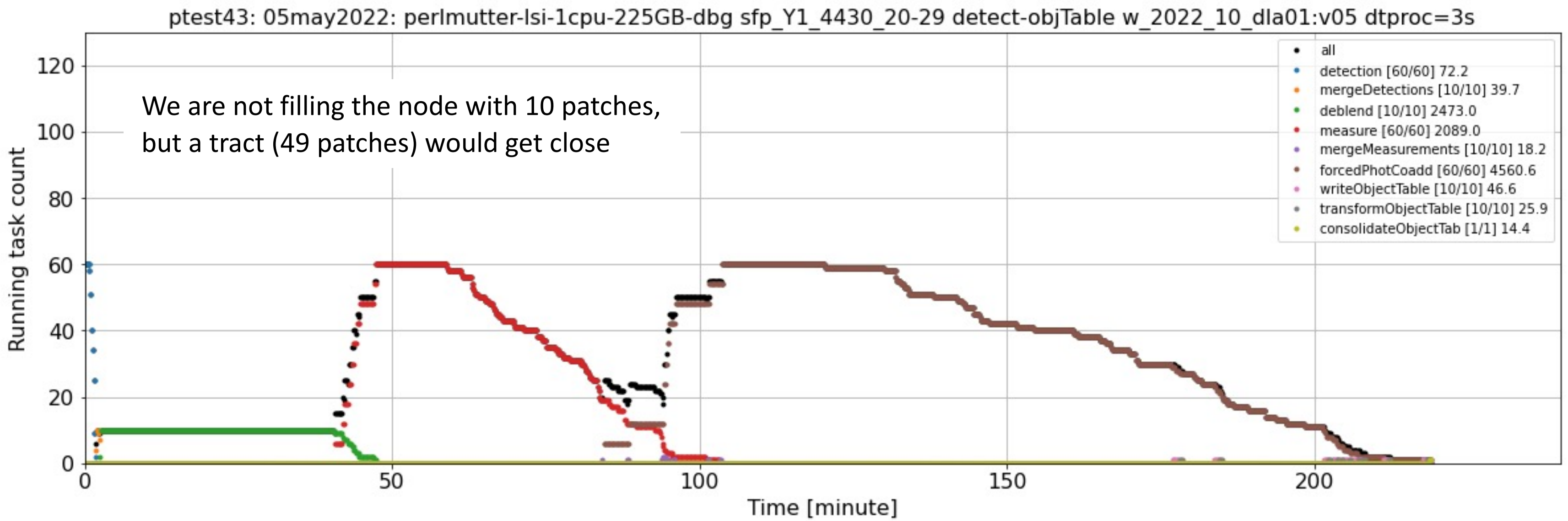


Plot show throughput for patch processing

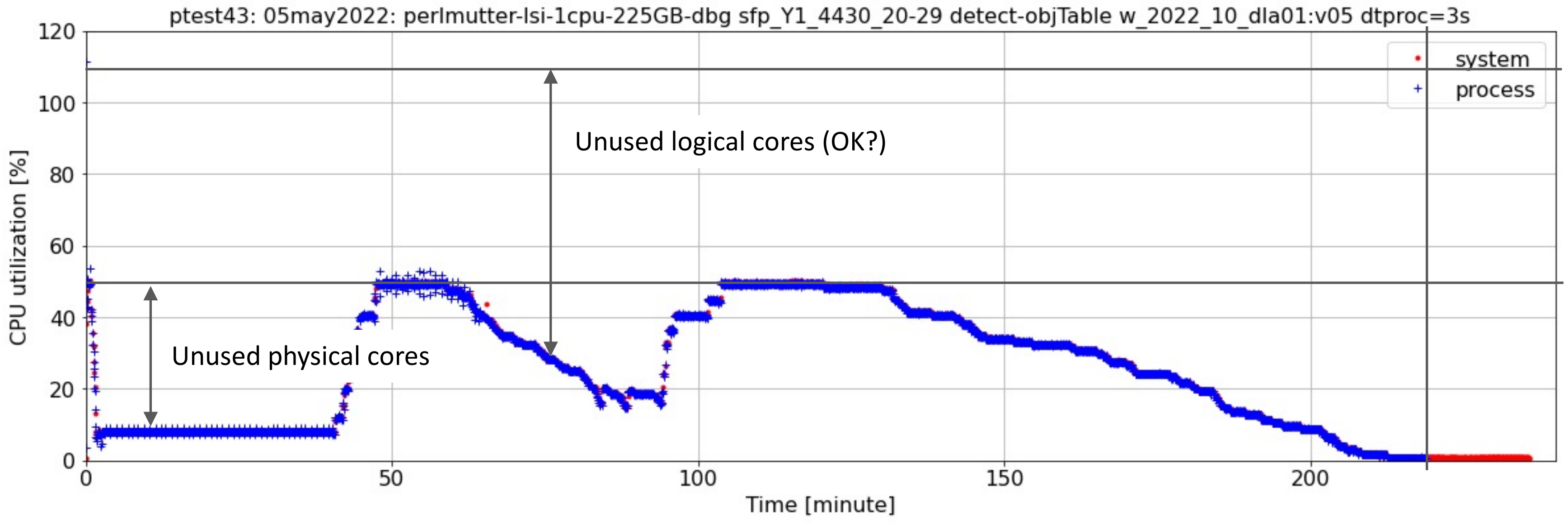
- assembleCoadd through makeObjectTable

Throughput and # running tasks by task type





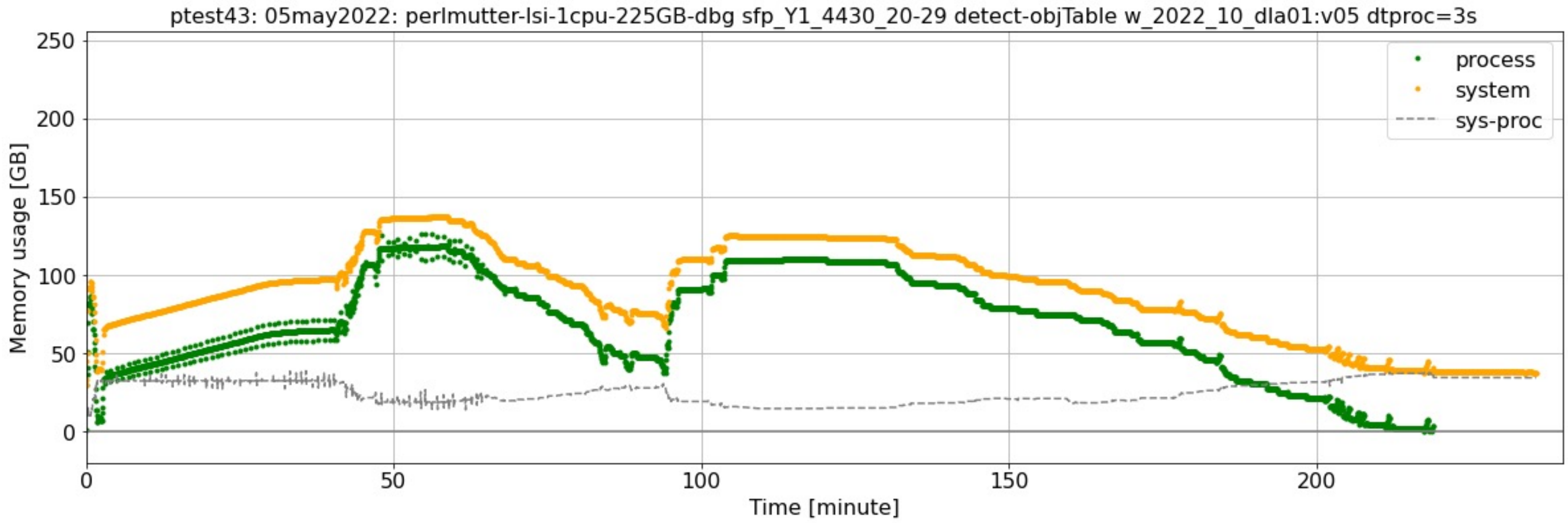
CPU utilization



Plot show CPU utilization

- Again, we need to run more than 10 patches to fill the node

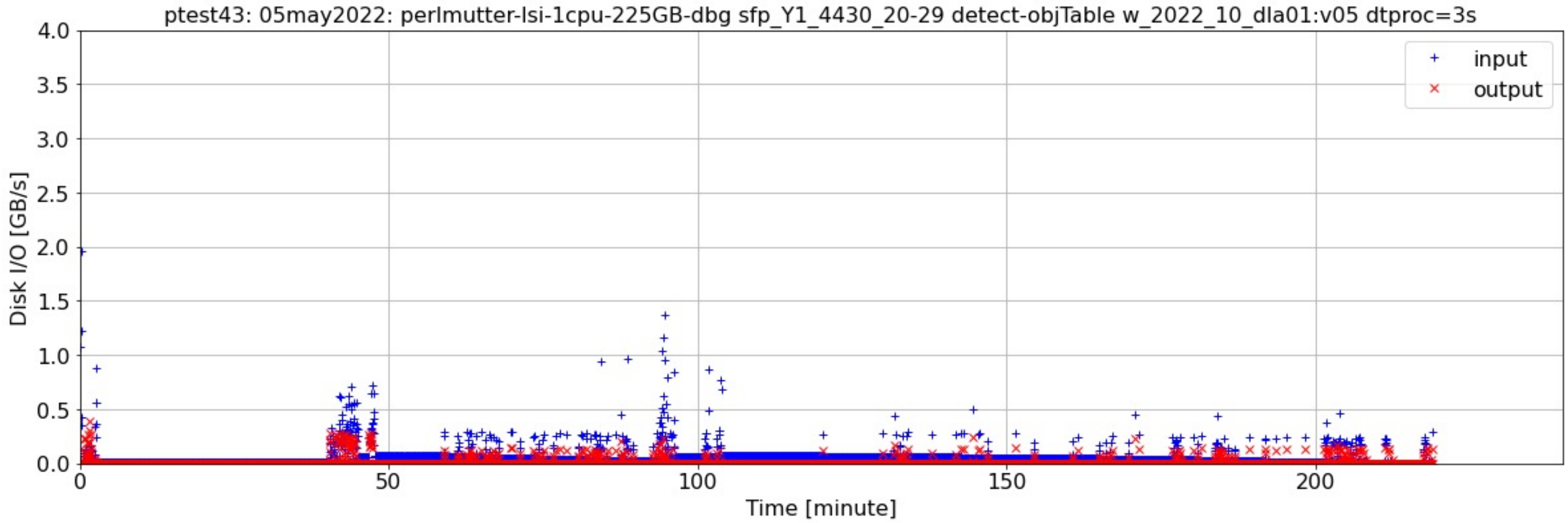
Memory usage



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

Extras

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

Assigned memory allocations

```
pipetask:
  isr:
    requestMemory: 2700
  characterizeImage:
    requestMemory: 1000
  calibrate:
    requestMemory: 1000
#  writeSourceTable:
#  transformSouceTable:
#  consolidateSourceTable:
  consolidateVisitSummary:
    requestMemory: 1000
#  selectGoodSeeingVisits:
  templateGen:
    requestMemory: 1500
  imageDifference:
    requestMemory: 3100
  forcedPhotDiffim:
    requestMemory: 2500
  skyCorrectionTask:
    requestMemory: 8000
  makeWarp:
    requestMemory: 3500
  assembleCoadd:
    requestMemory: 1600
  detection:
    requestMemory: 1300
  mergeDetections:
    requestMemory: 2000
  deblend:
    requestMemory: 7000
  measure:
    requestMemory: 1900
  mergeMeasurements:
    requestMemory: 2000
  forcedPhotCoadd:
    requestMemory: 1600
  forcedPhotCcd:
    requestMemory: 1000
#  writeObjectTable:
#  transformObjectTable:
#  consolidateVisitTable:
```