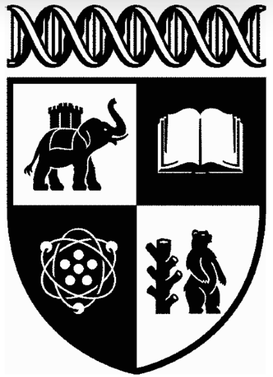
Effective Shared Computing:

Priorities, Queueing and IO

Chris Brady, Tanmoy Chakraborty, Heather Ratcliffe - SCRTP



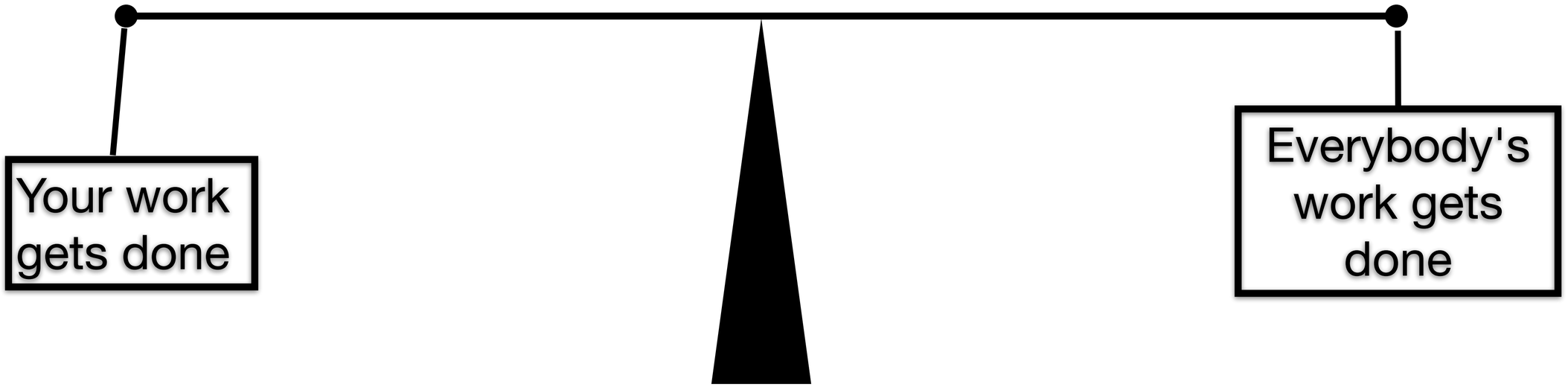
Aims

- Range of computational resources available
  - Generally, and, on average, these are over-full
  - **Aim 1 : understanding pros and cons of each option**
- To support distributed compute, need a shared storage system
  - **Aim 2 : understanding drawbacks and optimisations for working with networked storage**
- Computational work ranges from massive ensembles of independent jobs, to large "hero" simulations
  - **Aim 3 : understanding how to match job needs to available resource**

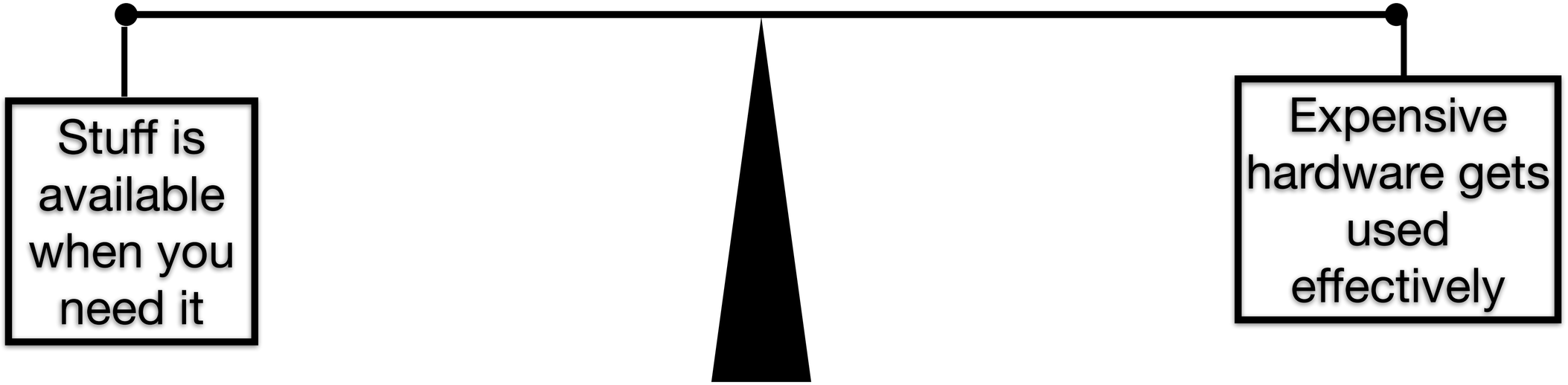
Structure

- We've got some stuff to say and present as the 'ground-work', in particular:
  - (a bit about) how to use SCRTP systems
  - what systems we have
  - how shared storage works
- Then plenty of time for questions about specific work
  - Please do ask questions as we go
  - But also, plenty of time at the end if you want to wait and see if we cover your question

The Scales of Justice



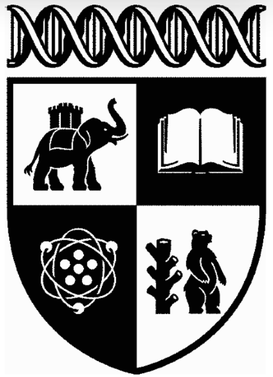
The Scales of Justice



Problems and Principles

1. Computers work best doing one thing for a "long" time
    1. They do NOT like switching tasks
2. Data works best when it is in the right place
    1. Storing data for the long term is a different issue than using it
    2. Speed of access (bandwidth+latency) costs literal money
3. Everything ends up urgent at the same time
    1. Conferences, holidays, data drops

Shared Systems



Why Queues?

- Shared systems can work many different ways, we'll focus on two
- Free-for-all: directly log on and use hardware
  - All jobs competing for the same, limited, resources
  - Oversubscription
- Queued: resource manager decides when things run
  - Needs a metric for sharing fairly
  - Requires restriction on job length (fairness occurs on this scale)
- Reserved: (worth a mention) resources are requested in advance

Oversubscription

- Oversubscription is resource contention applied to core in CPU
  - Resource contention is …. contention for resources
    - Network bandwidth (esp. with networked storage)
    - CPU cycles
    - Memory/cache space
- To avoid deadlocking, resources get passed around (packet switching, task switching, disk access/seeking)
  - *Every switch is a wasted cycle!*

Overheads

- On queued systems, *oversubscription* is avoided and we suffer instead from the *overheads* of doing so
- Resource scheduling is an overhead
  - Job setup/tear down is an overhead
  - File movement (into/out of tmp spaces etc)
  - Partially used resources (e.g. using all the memory on a node but not all the cores)

What is Available

- SCRTP provide:
  - desktops (the management and software for these) - sit in your office; connected to our storage
  - dedicated nodes - big workstations; sit in the data centre; connected to our storage; may be restricted to certain groups
  - taskfarm - shared hardware; no GPUs; do have himem nodes
  - clusters - independent, stand-alone systems; tightly connected nodes; dedicated, high performance storage

Demo 1

- A quickjob (use quickjob) on the taskfarm
  - Demo the error message running from home, but come back to this later
  - Show queue
  - Show sprio
    sprio -o "%.15i %9r %.8u %.8o %.10Y %.10A %.10F"
  - sacct: sacct --submit-line -j 5787693
  - https://docs.scrtp.warwick.ac.uk/taskfarm-pages/tf-reslimits.html

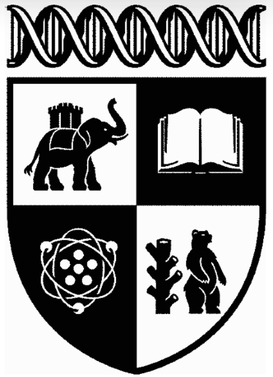
Demo 2

- A dedicated node
  - top and htop
    - Note multicore jobs appear differently depending on parallelism type
    - https://docs.scrtp.warwick.ac.uk/taskfarm-pages/tf-troubleshoot.html

Which (Batch) System

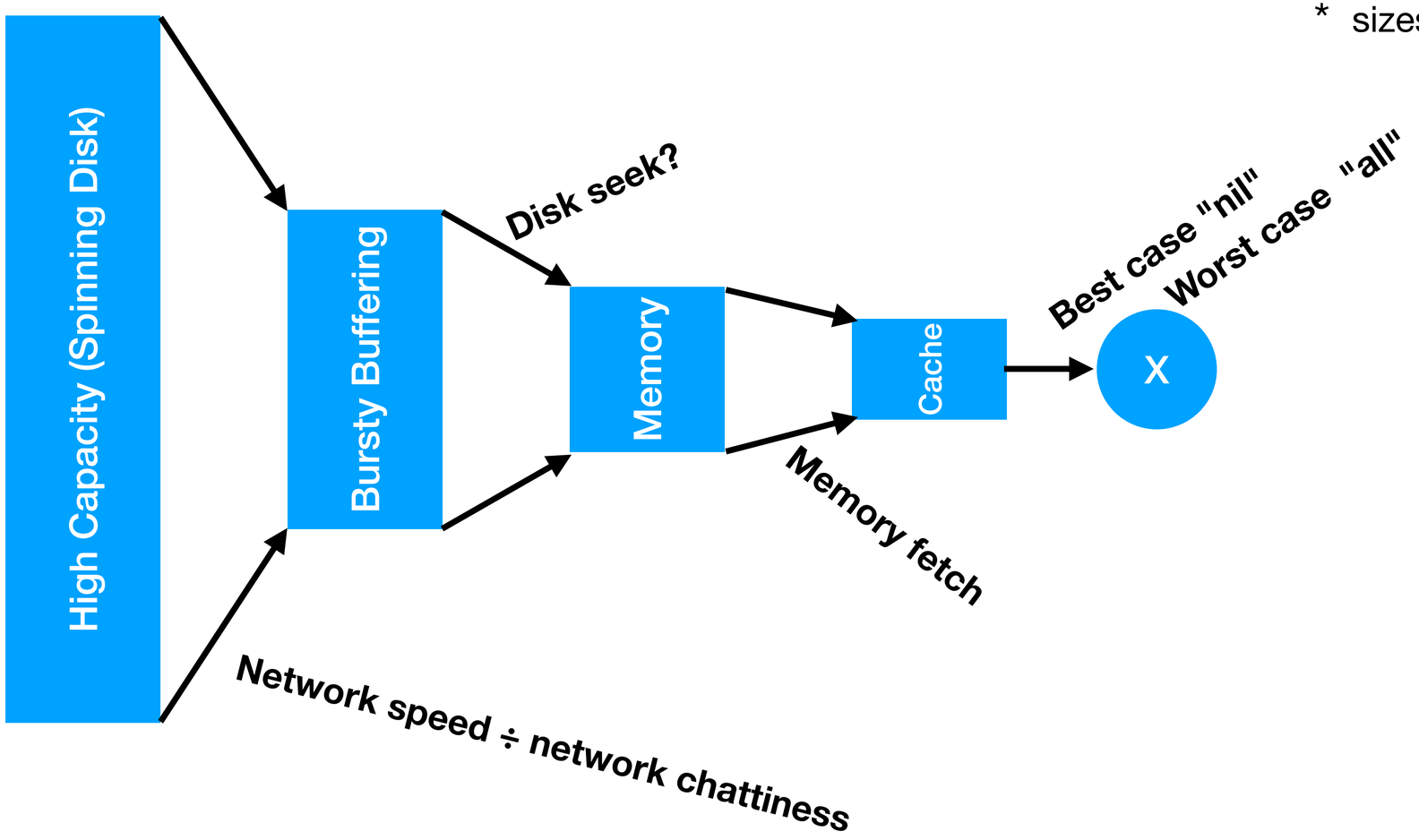
- Taskfarm is intended for farming tasks: many small jobs, not too IO intensive
- Clusters are better for heavy IO but best suited for larger jobs
- Will discuss this more later
- Packed jobs - one submission can run multiple tasks
  - On taskfarm: use this to copy-once-run-many
  - On clusters: pack multiple small core count jobs
- Job arrays - minimise scheduler overhead
- Job dependencies - one job can be made to strictly follow another

File Systems



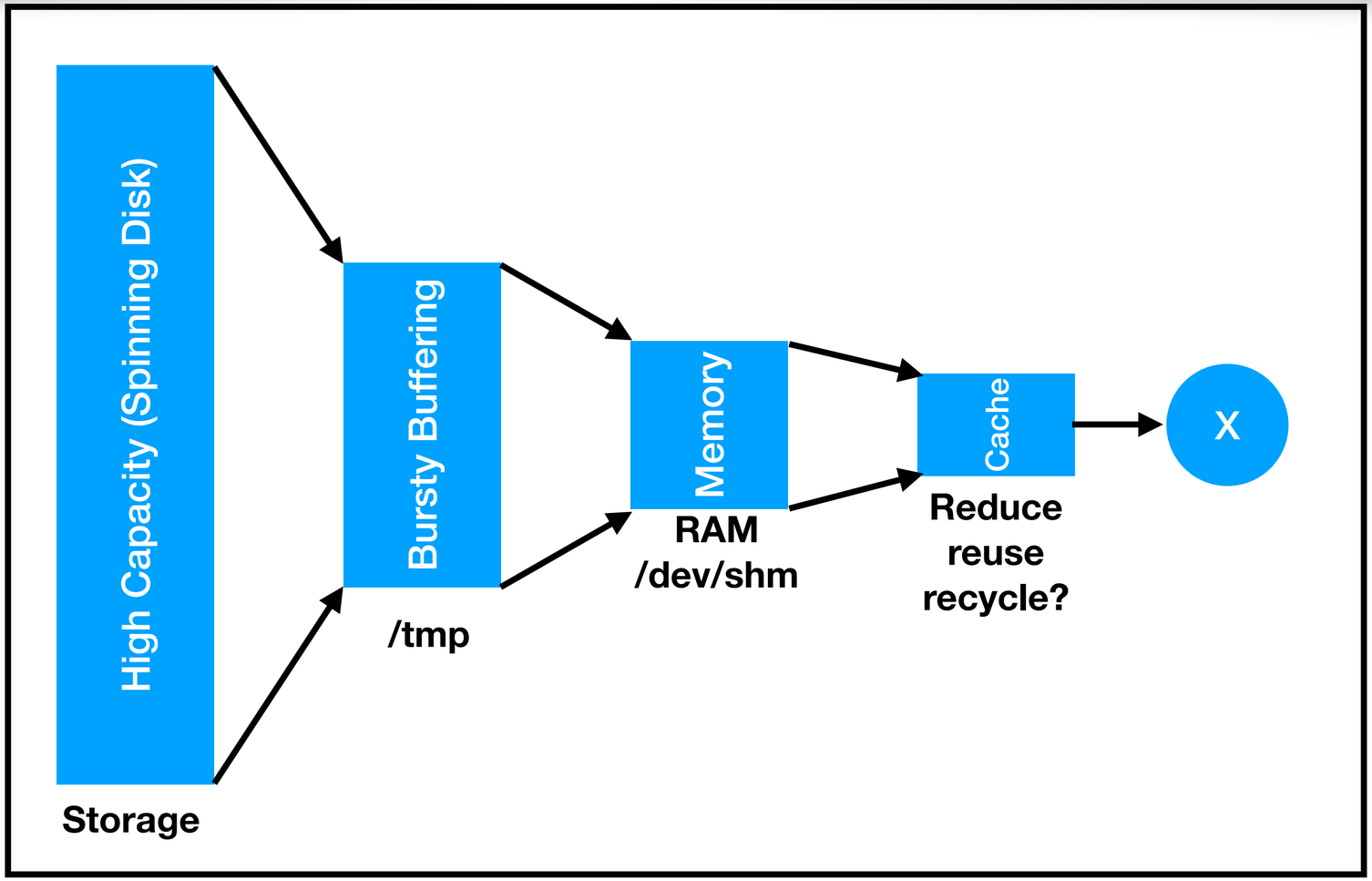
Capacity Scales

- What dictates data access speed?
- Speed is good
- So speed is expensive
- So big means slow



Workstations

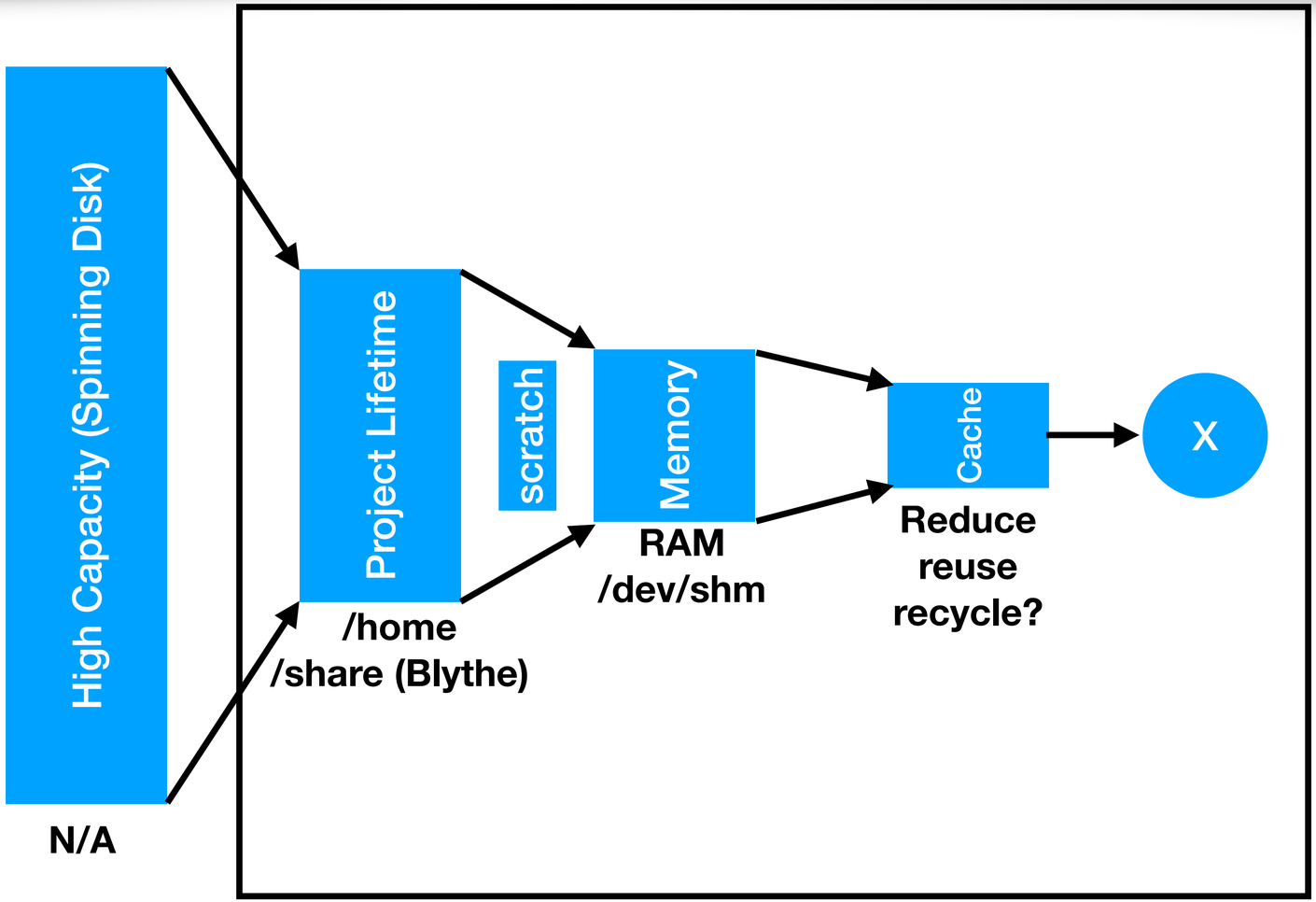
- Taskfarm AND/OR dedicated nodes
- Storage is "long term"



* sizes not to scale

Clusters

- On SCRTP clusters home and relevant storage is all high performance
- Intent is for data to leave after project is done



* sizes not to scale

Data Reduction

- First rule of efficiency - do less
  - Writing data costs, as does storing it
  - Computation which does not produce data is useless
- "Good" codes (or workflows) consider IO as fundamental part of task
  - In-code data reduction - averaging, transforming, time-snapshotting
  - Consider what could be done before going to disk
  - Can do this as part of a job script rather than build into code itself

Data Coalescence

- Example: archiving from /tmp
- https://docs.scrtp.warwick.ac.uk/taskfarm-pages/tf-slurm-pages/tf-localdisk.html

Buffering

- Second best way to be (time) efficient is to do more things at once
  - Hide the costs of 'slow' processes by running alongside
  - IO buffering is a great example
    - Occurs at many levels - the 'higher' you do it, the better you can decide
    - OS-level buffering has to be conservative, but will coalesce "somewhat"
    - Standard IO (std::cout, print etc) are buffered by default
      - May notice this because output can be missing if code crashes
    - In "user-space" mostly rely on runtime/compiler - coalesce and hope
      - whole-array ops, big writes

Buffering Demo

cd /storage/space2/phsmbb

quickjob 2

truncate --size 5G sample5G

Buffered - storage

  dd if=./sample5G of=./sample bs=1M count=5000

(Explanatory note - bs is block size, then bs*count is total bytes

UNBuffered - storage

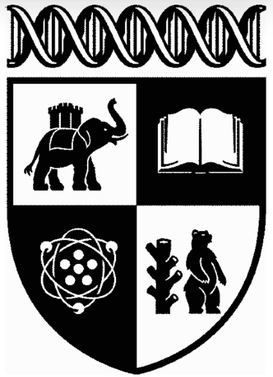
  dd if=./sample5G of=./sample bs=1M count=5000 oflag=direct

UnBuffered - /tmp

  dd if=./sample5G of=/tmp/sample bs=1M count=5000

Pre-prepared example - bs=4k

Which System?



How to Choose?

- Something to remember in all of this:
  - If it aint broke, don't fix it - but there is nothing as permanent as a temporary solution
  - So don't worry about optimal choices when:
    - You need to do literally one or two jobs
    - You have no idea what the bottlenecks will be yet
    - You don't know whether an approach even gets the right answer
    - ETC

How to Choose?

- Something to remember in all of this:
  - Other users exist
    - Their use of resources impacts yours, and vice versa
      - Directly via fairshare on clusters and taskfarm
      - Indirectly via contention on dedicated nodes and storage/network accesses
    - This means sometimes things run slower
    - Very rarely somebody else can cause your job to misbehave or crash

Dedicated Nodes

- Things to remember with dedicated nodes
  - Walltimes are not just for sharing - important updates need to be applied roughly once a month
  - Somebody else can (occasionally) do something so unkind that machine needs a reboot
  - Expect high 'up-time' in percent, but don't rely on running continually for more than a few days
  - Context switching tends to be far worse than you predict
    - Somebody else using 1 of your cores wont just double the time

Dedicated Nodes

- Things to remember with dedicated nodes
  - Storage is connected over a network
    - SCRTP don't control the (external) network - outages, slow access etc are IDG
    - NFS (Networked File System) is quite chatty - many small reads are really bad
  - Consider using /tmp space, tar-pipes, coalescing into big writes

Taskfarm

- Things to remember with taskfarm
  - You can use multiple nodes but not for tight-coupled parallelism
  - Storage connection should be good, but others using it can affect your jobs
    - Consider using /tmp or keeping files in memory

Clusters

- Things to remember with clusters
  - Big multi-node jobs will work "as well as possible"
  - Queueing times can be long around busy times
  - Data should be moved off the clusters "when project is done"
    - Note Blythe has dedicated data-transfer nodes

Wildcard : Cloud

- Cloud systems can seem like a solution
  - Queueing is replaced by cash
    - You pay more at busy times, for urgent work etc
    - You may also pay for storage and ingress/egress
  - If cloud "undercuts" us, it usually means it is being subsidised
  - Can be valuable for truly unusual hardware

Example: ensemble of 'events'

- Take an imaginary example : running a large ensemble of unconnected (or weakly connected) events
- Suppose 250 events, each taking 10 mins
  - Approx 42 hours of work on one core
  - Suppose they share _some_ but not all input data
- Compare the options and choices

Example: dedicated node

Pro	Con
Run immediately	How many at once? 1? 16?
Pick and choose which next	How to track which have run/restart on reboots etc?
Data can be 'kept' in memory	Runtime harder to predict due to contention

Example: taskfarm

Pro	Con
Can submit as a job array	Waiting in queue
At quiet times may be able to run many simultaenously	Input data has to be copied into place
Hardware costs shared with other groups	Hardware access shared with other groups

Example: cluster

Pro	Con
Submit as job array	Queue and fairshare
Better IO performance	Waiting time depends a lot on busyness
More likely to get a whole node - easier to get in-memory files	Have to copy data on and off

Ensemble: conclusion

- All options suitable
- Dedicated node will require some micromanagement
- Perhaps not worth "getting the hang" of clusters just for this?
- Taskfarm probably wins
  - Dedicated nodes can be part of taskfarm - ask for details

Example: single large 'event'

- Take an imaginary example : running a single large event
- Suppose one large "event" of around 2500 core-hours
- Imagine this completes on 32 cores in 72 hours
  - Or 64 cores in 40
  - (note 2304 -> 2560 total hours)
- Compare the options and choices

Example: dedicated node

Pro	Con
Choose number of cores	64 likely ~ entire node, highly likely to see contention
Check output freely as code runs	If IO is heavy, likely will suffer
No loss of priority for subsequent jobs	Nodes occasionally need rebooting - perhaps should checkpoint?

Example: taskfarm

Pro	Con
No contention to worry about	48 cores per node, likely to need a checkpoint-restart
Can queue several such jobs	Unlikely to perform well across nodes?
View output interactively on dedicated node?	If IO is heavy, likely will suffer

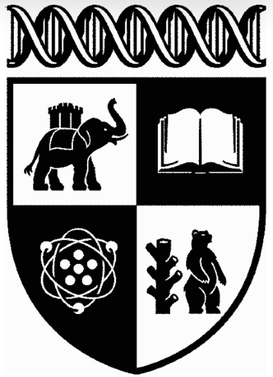
Example: cluster

Pro	Con
64 cores is comparatively easy	Code "wastes" some resource vs 32
Can run more than one like this	Might need checkpoint-restart
High IO performance	Have to copy data on and off

Big Job: conclusion

- Numbers chosen are right at the crunch between walltime and "waste"
- Dedicated node may work, but unpredictable
- Taskfarm suitable unless job is very IO heavy
  - But this is a "large" job for taskfarm
- Clusters ideal
  - If needing "many" like this, definitely worth signing up

Misc. Systems…



Multi-job Submissions

- If you have a lot of small jobs, you can just schedule them all and let slurm deal with it
- OR, there are several ways to create a "bigger" job
  - Request a larger allocation, and run several tasks within it
    - https://docs.scrtp.warwick.ac.uk/taskfarm-pages/tf-slurm-pages/tf-parsrun.html
    - Can use (GNU) parallel to perform simple task dispatch
  - Use a 'job array' - submit one job asking for many tasks - slurm script body is run for each

Multi-job Submissions

- Use a workflow manager (e.g. snakemake)
  - PLEASE be cautious - the root process of these can be heavyweight itself and thus not suitable for the head nodes
  - Can introduce a lot of overhead depending how integrated package is
  - Two ways these can work:
    - Root process submits jobs to the queue
    - Root process run inside job allocation and runs tasks directly

Checkpointing

- How do systems with walltimes want you to work?
  - Checkpointing:
    - Periodically (every hour?) write enough data to stop and restart
    - Bonus: node failures, power cuts, etc don't lose days of work
    - Downside : writing data costs time

Checkpointing

- Many bigger packages support this - just have to enable it
- Sometimes can do it "from the outside" e.g. DMTCP
  - Fragile - what if code is writing an output at the time?
- If possible do it at sensible points in a workflow
  - E.g. do these steps, write-out, restart

Checkpointing

- If implementing yourself
  - ALWAYS use A/B rolling - write a new checkpoint completely before touching the previous one
  - Consider when it is valid to 'stop' your code e.g. start of a "step"
  - Consider any parallelism - can you restart on more/fewer cores?
  - Make sure to flush and close any files in the shutdown

Shared data

- If your group has a (read only?) shared data set can ask for a shared folder
  - Read operations will usually slow down a little due to multiple accesses
  - Write operations usually lock at level of an entire file so suffer a LOT
    - Don't really want to be writing in parallel anyway, but do need to ensure files get opened read-only to avoid locking

Parallelism in One Minute

- When talking about parallel processing there are two fundamental varieties : shared memory and distributed memory
- Shared memory: memory is… shared. All processes access the same memory
  - Pro: quick and easy to access the same data
  - Con: quick and easy to clobber that data
  - In other words, you have access to all 'threads' data and have to protect against races
    - Overlapping reads and writes - bad
    - Answer depends on order of operation - also bad

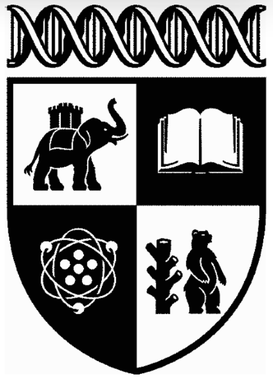
… And Two Slides

- Distributed memory is separate: individual processes have their own copy of all data. Communication is required to share it
  - Pro: works across multiple nodes
  - Con: multiple copies of any shared data
  - Con: communication is "much slower"* than direct memory access
  - *latency hiding and well designed comms mitigate this a lot
- Hybrid: mix of both. Usually shared on (some of) a single node, then distributed to allow working across them

And the Reason to Ask

- Why does this matter?
  - You start jobs a little differently
  - One works across nodes, the other doesn't
    - This also allows spreading across hardware to get more memory
  - Depends on package whether it uses the one, the other, either or both
    - Sometimes tweaking the breakdown can help performance

Conclusions



Flippant Conclusions

- Bigger is better
  - But not too big
- Don't repeat if you can re-use
  - Unless it's cheaper to create than store
- Use whichever system is most convenient
  - Until you risk impeding others