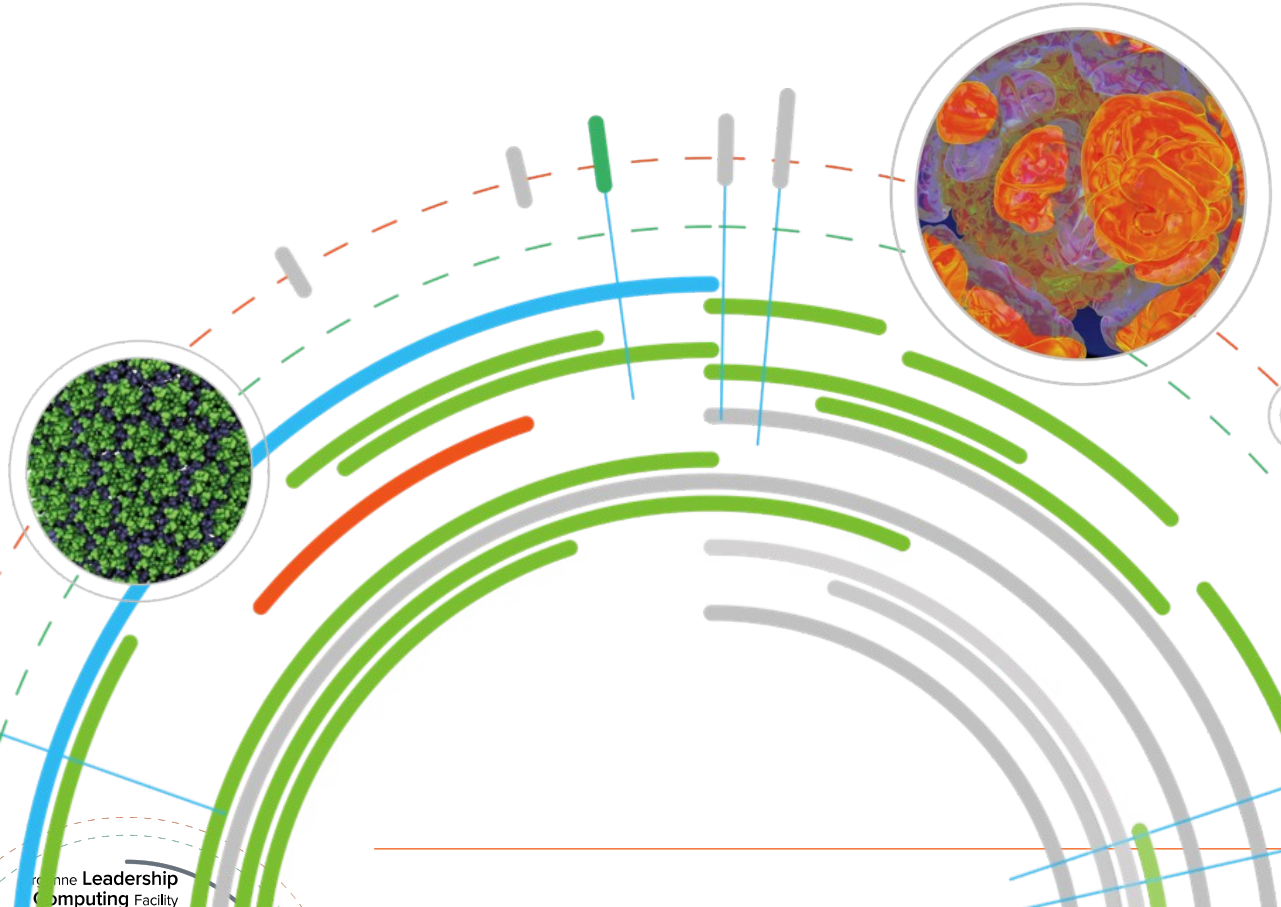
The Ghost of HPC Present



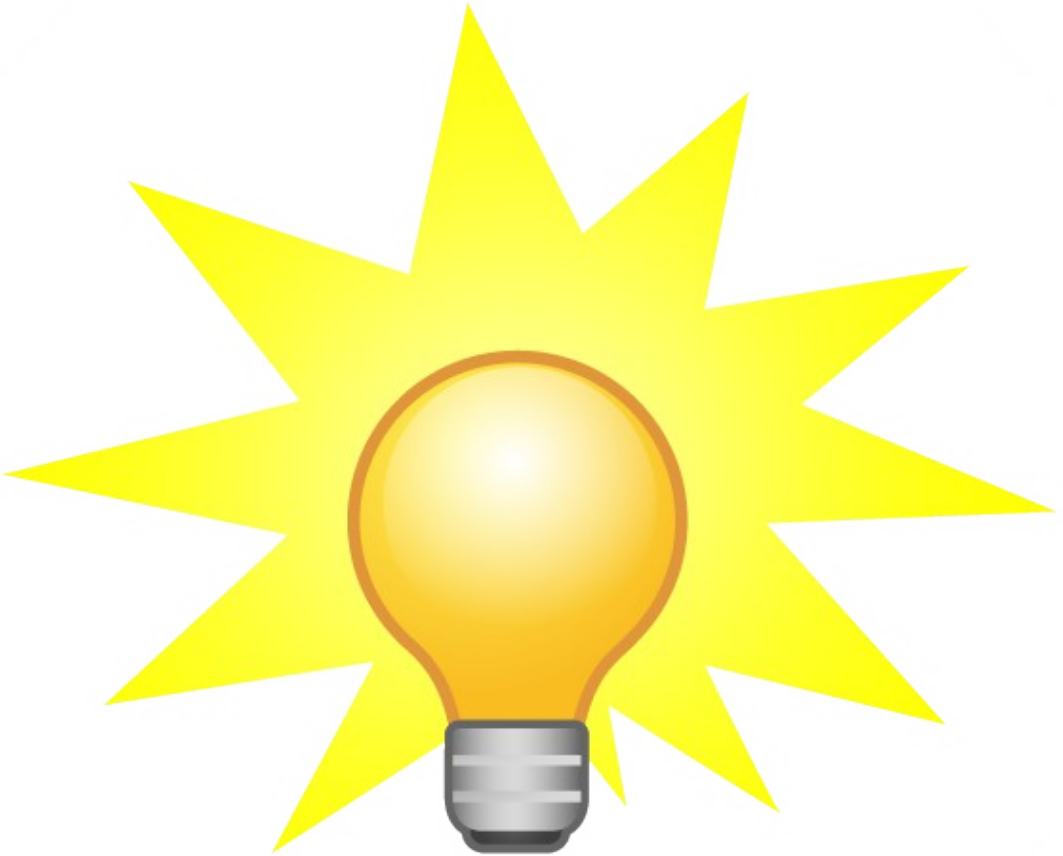
Hal Finkel
hfinkel@anl.gov
2015-07-27

Argonne **Leadership Computing** Facility

Argonne NATIONAL LABORATORY

Outline

- Why is HPC Hard?
- What can you do about it?
- Conclusions

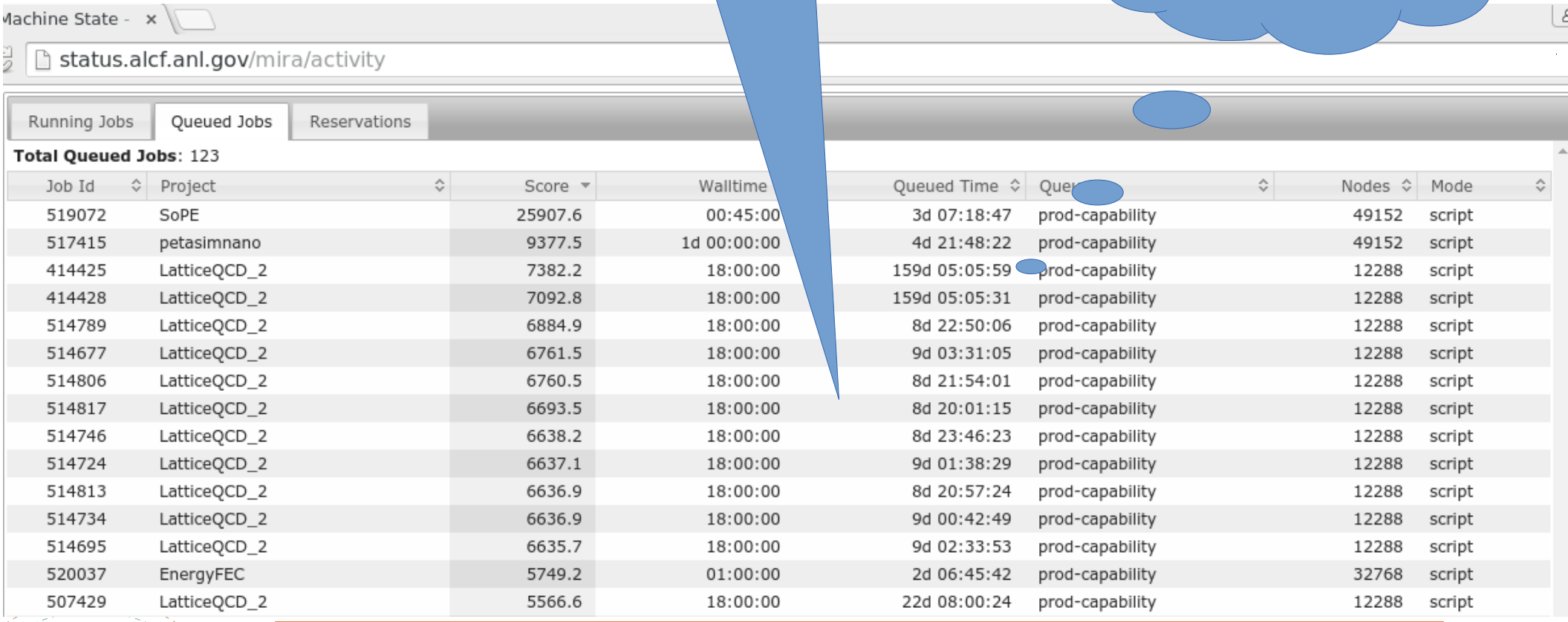


You need to share

Your job won't run right away...

These big jobs have been waiting for days

These were on hold for a while... I hope



Job Id	Project	Score	Walltime	Queued Time	Que	Nodes	Mode
519072	SoPE	25907.6	00:45:00	3d 07:18:47	prod-capability	49152	script
517415	petasimnano	9377.5	1d 00:00:00	4d 21:48:22	prod-capability	49152	script
414425	LatticeQCD_2	7382.2	18:00:00	159d 05:05:59	prod-capability	12288	script
414428	LatticeQCD_2	7092.8	18:00:00	159d 05:05:31	prod-capability	12288	script
514789	LatticeQCD_2	6884.9	18:00:00	8d 22:50:06	prod-capability	12288	script
514677	LatticeQCD_2	6761.5	18:00:00	9d 03:31:05	prod-capability	12288	script
514806	LatticeQCD_2	6760.5	18:00:00	8d 21:54:01	prod-capability	12288	script
514817	LatticeQCD_2	6693.5	18:00:00	8d 20:01:15	prod-capability	12288	script
514746	LatticeQCD_2	6638.2	18:00:00	8d 23:46:23	prod-capability	12288	script
514724	LatticeQCD_2	6637.1	18:00:00	9d 01:38:29	prod-capability	12288	script
514813	LatticeQCD_2	6636.9	18:00:00	8d 20:57:24	prod-capability	12288	script
514734	LatticeQCD_2	6636.9	18:00:00	9d 00:42:49	prod-capability	12288	script
514695	LatticeQCD_2	6635.7	18:00:00	9d 02:33:53	prod-capability	12288	script
520037	EnergyFEC	5749.2	01:00:00	2d 06:45:42	prod-capability	32768	script
507429	LatticeQCD_2	5566.6	18:00:00	22d 08:00:24	prod-capability	12288	script

You have a limited amount of time

- A large allocation on Mira (ALCF's production resource) is a few hundred million core hours
- 100 M core hours / (16 (cores per node) * 1024 (nodes per rack) * 48 (racks)) == 127 hours
- 127 hours is 5.3 days
- Running on the whole machine (48 racks) for 24 hours is the largest possible job
- Thus, with 5 jobs (plus some test runs), a 100-M-core-hour allocation could be gone

Supercomputers are not commodity machines

- Even when built from commodity parts, the configuration and scale are different
- The probability that you'll try to do something in your application that has never been tested before is high
- The system software will have bugs, and the hardware might too.
- The libraries on which your code depends might not be available.

Your jobs will fail

- The probability that a node will die, its DRAM will silently corrupt your data (including those in the storage subsystem), etc. is very low.
- However, if you spend a large fraction of your life running on large machines, you'll see these kinds of problems.

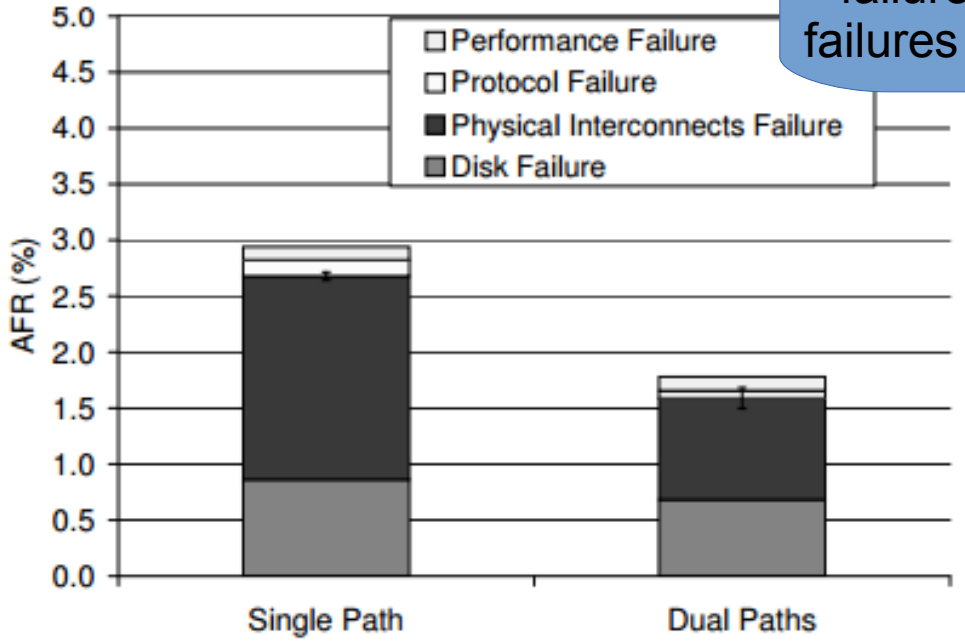
Your jobs will fail (cont.)

From the introduction of Fiala, et al. 2012:

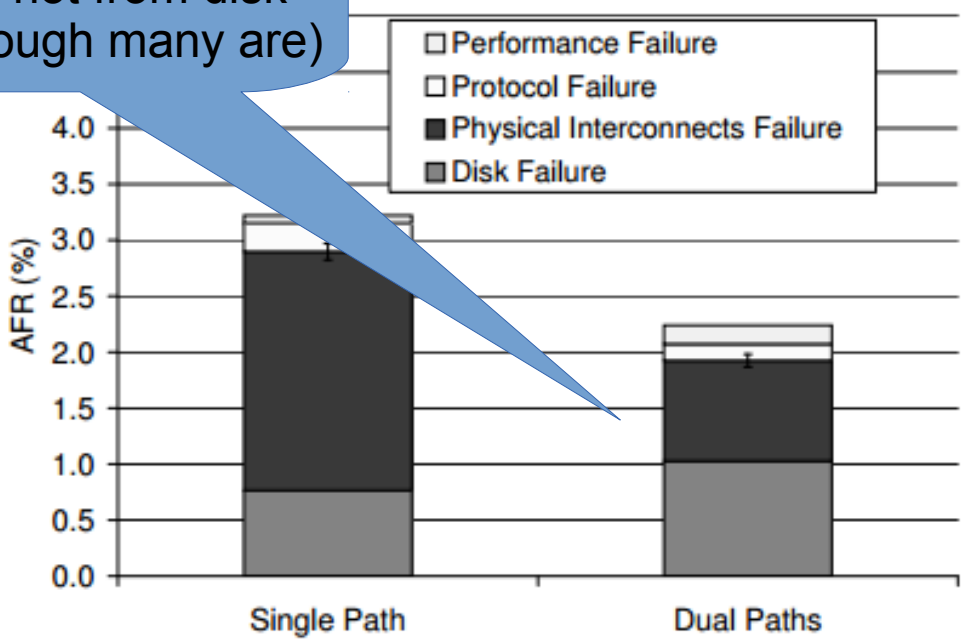
- Servers tend to crash twice per year (2-4% failure rate) (Schroeder, et al. 2009). *HPC node hardware is designed to be somewhat more reliable, but...*
- 1-5% of disk drives die per year (Pinheiro, et al. 2007). *HPC storage systems use RAID, but... (also, many hardware RAID controllers don't do proper error checking, see Krioukov, et al. 2009 – older paper, but personal experience says this is still true today)*
- DRAM errors occur in 2% of all DIMMs per year (Schroeder, et al. 2009)
- ECC alone fails to detect a significant number of failures (Hwang, et al. 2012)

Your jobs will fail (cont.)

Most storage subsystem failures are not from disk failures (although many are)



(a) Mid-range systems



(b) High-end systems

Jiang, et al. 2008

Your jobs will fail (sometimes worse)

Table 2. Estimated rates of UDEs in $\frac{\text{UDEs}}{\text{I/O}}$.

UDE Type	Estimated Rate	
	Nearline	Enterprise
Dropped I/O	$9 \cdot 10^{-13}$	$9 \cdot 10^{-14}$
Near-off Track I/O	10^{-13}	10^{-14}
Far-off Track I/O	10^{-12}	10^{-13}

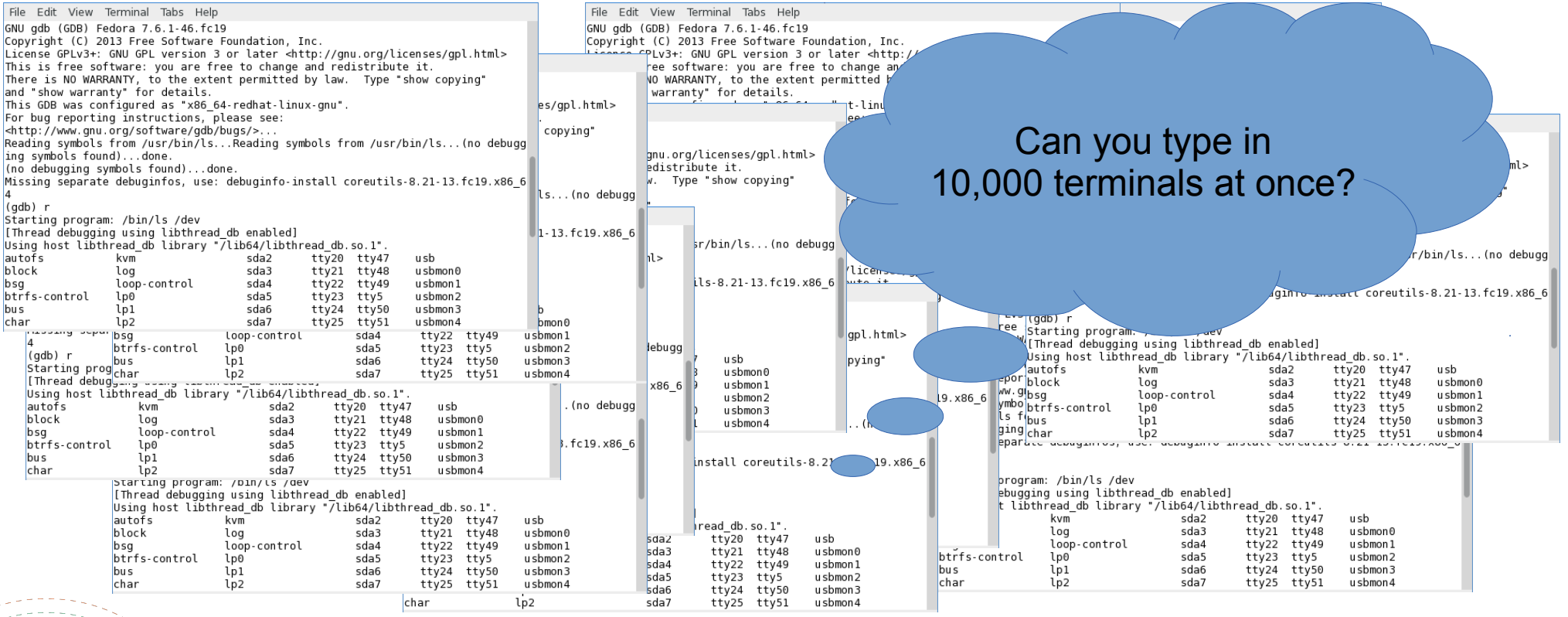
Rozier, et al. 2009 – UDE == Undetected Disk Error.
They assume 4 KB per I/O request

So you should expect to see silent data corruption once in every (even with RAID):
1/(2 x 10^-13) * 4*1024 == 2 x 10^16 bytes (20 PB)

Mira's file system is ~28 PB, so this is not an unthinkable number
(and, from personal experience, the rate is somewhat higher than that)

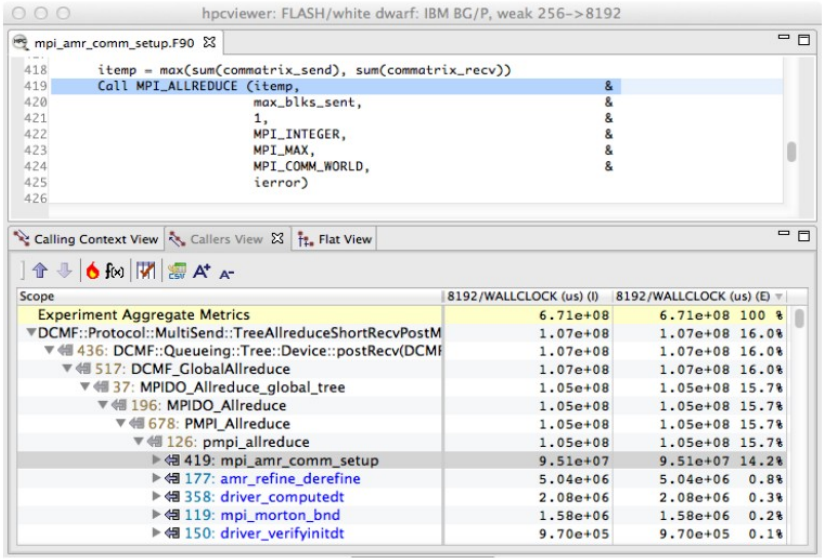
Debugging is hard

Your favorite debugger is great, but probably won't run at scale...

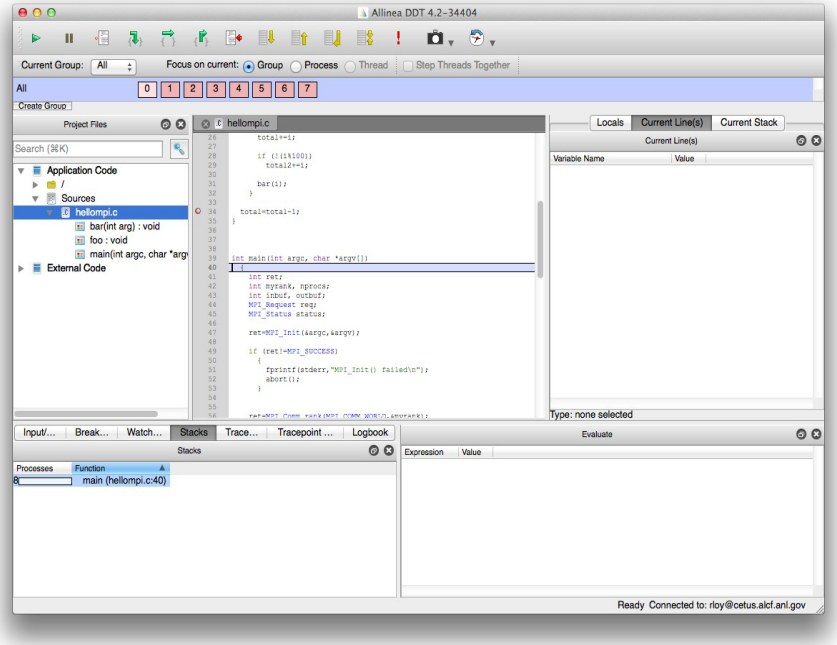


Debugging is hard (cont.)

Tools for debugging, profiling, etc. at scale are available, but they won't be what you're used to, and they might fail as well.



HPCToolkit



Allinea DDT

Everything is fast, but too slow...

- The network is fast, but likely slower than you'd like
- The same is true for the memory subsystem
- The same is true for the storage subsystem

Mira's file system can provide 240 GB/s – but you need to use the whole machine to get that rate. In addition, writing is slower than reading.

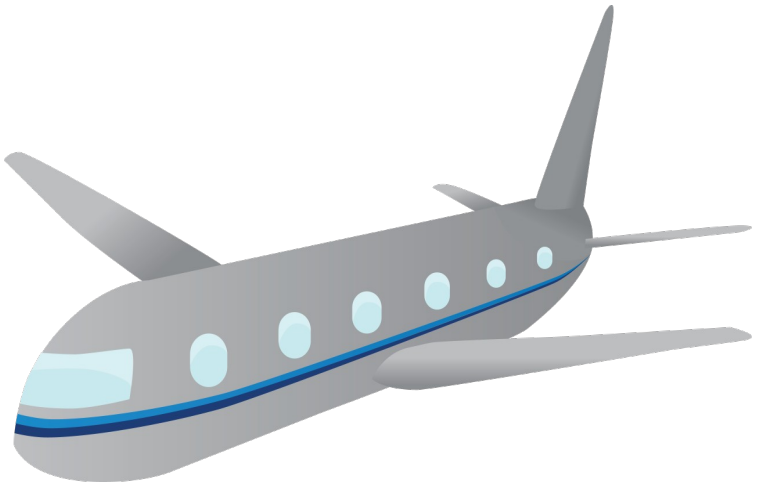
If you had the whole machine, and were the only one using the file system, then reading in enough to fill all 768 TB of DRAM would take:

(768*1024 / 240)/60 == 55 minutes

Everything is fast, but too slow... (cont.)

Also remember that:

- There is a big difference between latency and bandwidth
- I/O performance tends to be quirky (memory performance does too, to a lesser extent)
- Load balancing can be tricky



Parallelism (not just MPI any more)

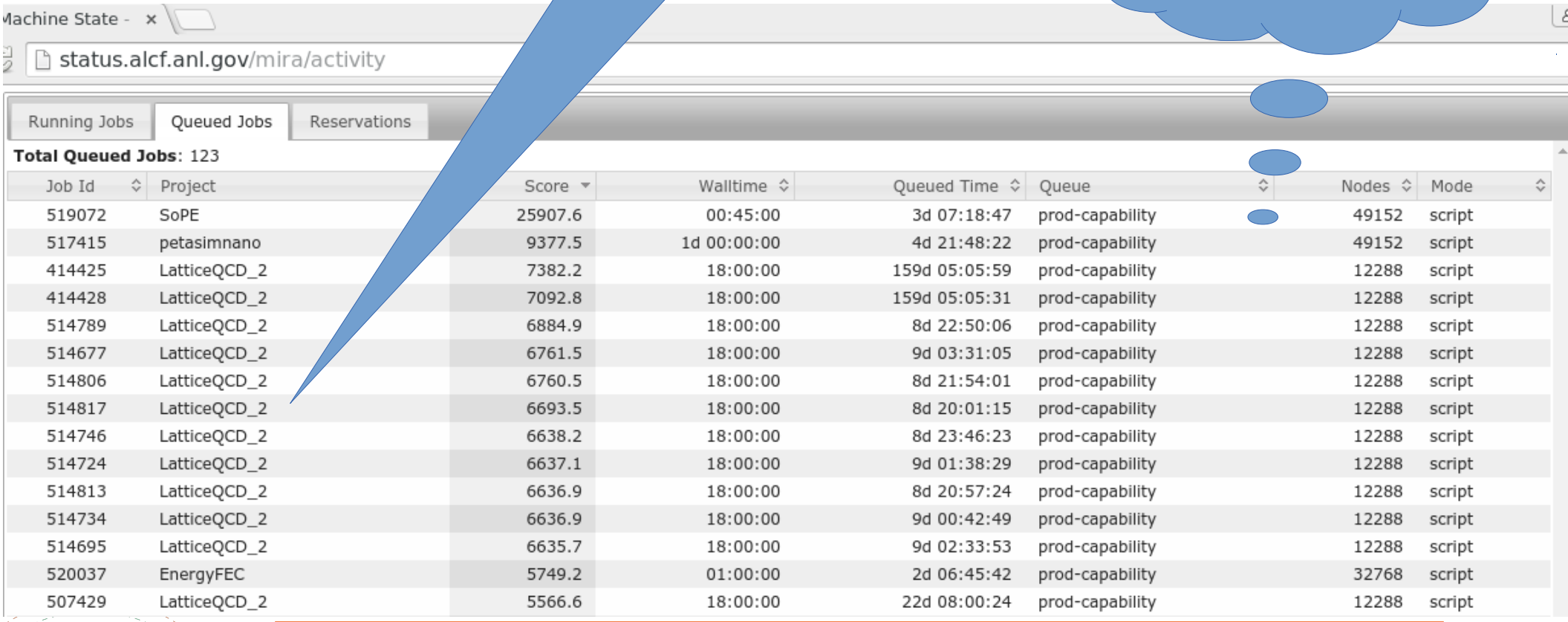
- MPI often has many O(#ranks) data structures per node (that's quadratic in the number of ranks globally), and then there are GPUs...
- So you'll end up using MPI+X, where X is OpenMP, CUDA, etc. depending on the platform
- Threads, vectorization, etc. are key, but exploiting them requires planning
- Your code might need to run on both many-core CPU systems, and CPU+GPU systems

So, now what do you do?

Experiment in parallel

This guy has the right idea

Some of these (time,size) combinations are better than others

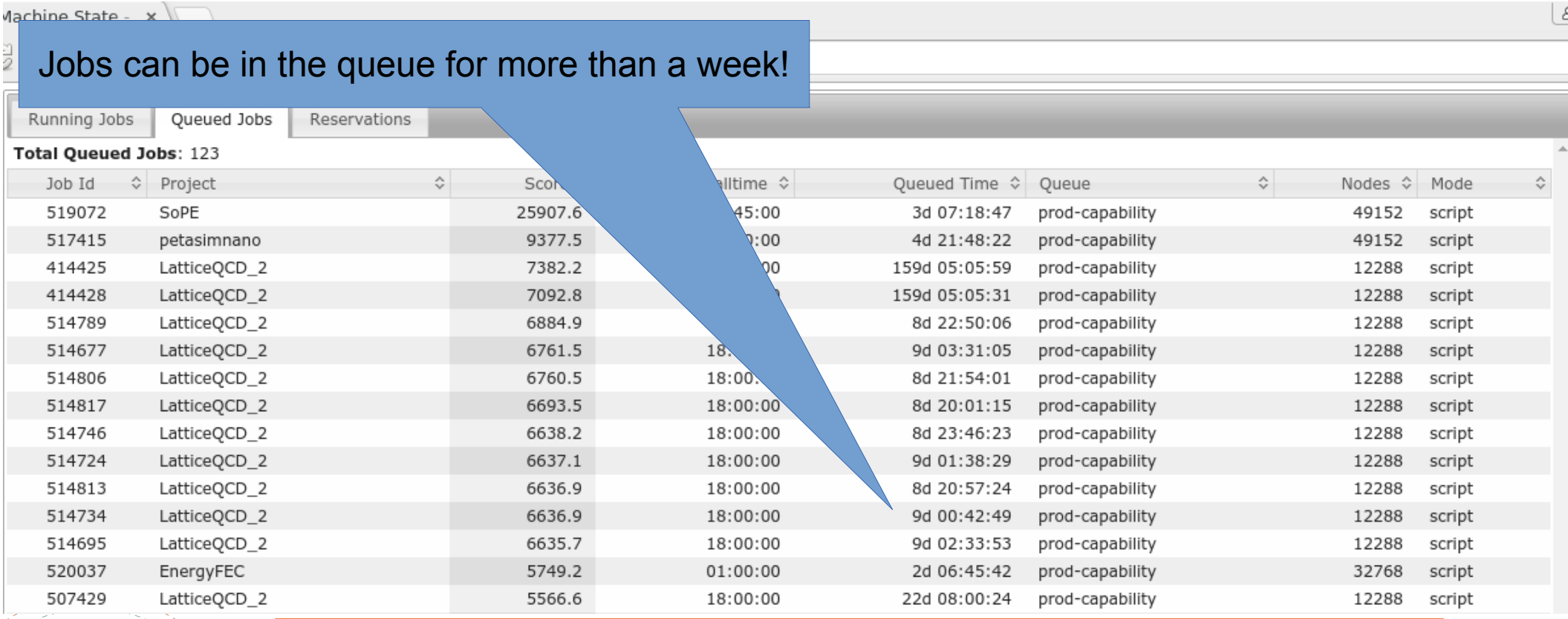


Total Queued Jobs: 123

Job Id	Project	Score	Walltime	Queued Time	Queue	Nodes	Mode
519072	SoPE	25907.6	00:45:00	3d 07:18:47	prod-capability	49152	script
517415	petasimnano	9377.5	1d 00:00:00	4d 21:48:22	prod-capability	49152	script
414425	LatticeQCD_2	7382.2	18:00:00	159d 05:05:59	prod-capability	12288	script
414428	LatticeQCD_2	7092.8	18:00:00	159d 05:05:31	prod-capability	12288	script
514789	LatticeQCD_2	6884.9	18:00:00	8d 22:50:06	prod-capability	12288	script
514677	LatticeQCD_2	6761.5	18:00:00	9d 03:31:05	prod-capability	12288	script
514806	LatticeQCD_2	6760.5	18:00:00	8d 21:54:01	prod-capability	12288	script
514817	LatticeQCD_2	6693.5	18:00:00	8d 20:01:15	prod-capability	12288	script
514746	LatticeQCD_2	6638.2	18:00:00	8d 23:46:23	prod-capability	12288	script
514724	LatticeQCD_2	6637.1	18:00:00	9d 01:38:29	prod-capability	12288	script
514813	LatticeQCD_2	6636.9	18:00:00	8d 20:57:24	prod-capability	12288	script
514734	LatticeQCD_2	6636.9	18:00:00	9d 00:42:49	prod-capability	12288	script
514695	LatticeQCD_2	6635.7	18:00:00	9d 02:33:53	prod-capability	12288	script
520037	EnergyFEC	5749.2	01:00:00	2d 06:45:42	prod-capability	32768	script
507429	LatticeQCD_2	5566.6	18:00:00	22d 08:00:24	prod-capability	12288	script

Don't wait until the last minute...

Your allocation probably ends along with many others, and many users procrastinate, don't be one of them!



Job Id	Project	Score	alltime	Queued Time	Queue	Nodes	Mode
519072	SoPE	25907.6	45:00	3d 07:18:47	prod-capability	49152	script
517415	petasimnano	9377.5	:00	4d 21:48:22	prod-capability	49152	script
414425	LatticeQCD_2	7382.2	00	159d 05:05:59	prod-capability	12288	script
414428	LatticeQCD_2	7092.8		159d 05:05:31	prod-capability	12288	script
514789	LatticeQCD_2	6884.9		8d 22:50:06	prod-capability	12288	script
514677	LatticeQCD_2	6761.5	18:	9d 03:31:05	prod-capability	12288	script
514806	LatticeQCD_2	6760.5	18:00:	8d 21:54:01	prod-capability	12288	script
514817	LatticeQCD_2	6693.5	18:00:00	8d 20:01:15	prod-capability	12288	script
514746	LatticeQCD_2	6638.2	18:00:00	8d 23:46:23	prod-capability	12288	script
514724	LatticeQCD_2	6637.1	18:00:00	9d 01:38:29	prod-capability	12288	script
514813	LatticeQCD_2	6636.9	18:00:00	8d 20:57:24	prod-capability	12288	script
514734	LatticeQCD_2	6636.9	18:00:00	9d 00:42:49	prod-capability	12288	script
514695	LatticeQCD_2	6635.7	18:00:00	9d 02:33:53	prod-capability	12288	script
520037	EnergyFEC	5749.2	01:00:00	2d 06:45:42	prod-capability	32768	script
507429	LatticeQCD_2	5566.6	18:00:00	22d 08:00:24	prod-capability	12288	script

Make your code exit!

"Begin at the beginning," the King said, very gravely, "and go on till you come to the end: then stop."
- Lewis Carroll, Alice in Wonderland

On Mira, 1 minute on the whole machine is 13,107 core hours!

- Take the time to figure out how long your code takes to run, and make it exit (don't always run your jobs until the system kills them).
- Exiting cleanly (using an exit code of 0), is often necessary for dependency chaining to work.

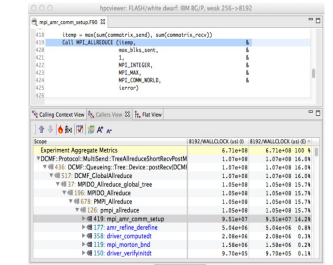
Save your work

- Save all of your configuration files for any experiments you do.
- Save your log files.
- Document how to run your code and process the results, what you've actually run, and where the data lives.

Store the run configuration in every output file
(as comments, metadata, etc.)
Seriously, just do it...

Optimize your code

- Read some documentation on the system, compiler, etc.
- Do you need strict IEEE floating point semantics? If not, turn them off.
- Profile your code
- Then optimize (using better algorithms first)



Always compile with -g (debugging symbols)
Essential for debugging if your code crashes,
but also useful for profiling, etc.

Contact support

If something is wrong, or you need help for any other reason, contact the facility's support service:

(system reservations for debugging are often possible, just ask!)

These people are not scary!

On using libraries

Using libraries written by experts is really important, but remember that if you're using something obscure, you'll “own” that dependency.

https://en.wikipedia.org/wiki/File:Two_Paths_Diverged_in_a_wood.JPG

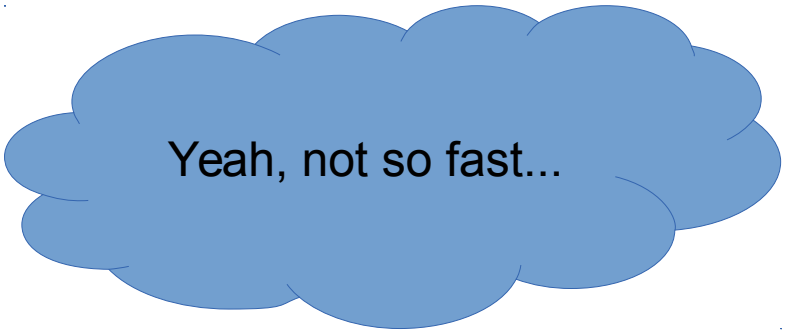
Some popular libraries:

- Trilinos
- PETSc
- HDF5
- FFTW
- BLAS/LAPACK

Programming-Language Features

- Be careful when using the latest-and-greatest programming-language features
- We're just getting C++11 and Fortran 2008 support in compilers now (not counting co-arrays)

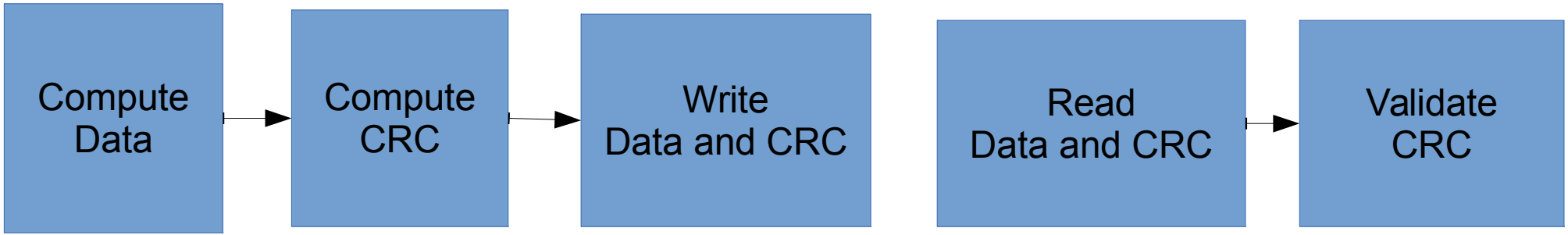
```
// C++14: new expressive power

auto size = [](const auto& m) { return m.size(); };
```



(the facilities try their best to support these things using open-source compilers, etc. but, as a user, you'll likely want the option of using the vendor's compilers)

Validation

- Have test problems, hopefully both small and large ones, and run them on the system, with the same binary you plan to use for production, before starting your production science.
- Make sure all of the data files have checksums (CRCs) so that you can validate that the data you wrote is the same as the data you read in.
- Build physical diagnostics into your simulations (conservation of energy, power spectrum calculations, etc.) and actually check them.



Save your build settings

- Make your code print out or save its configuration when it starts, and also:
  - The compilers and build flags used
  - The version control revision information for the source being built

If you don't know what version control is, learn about git.

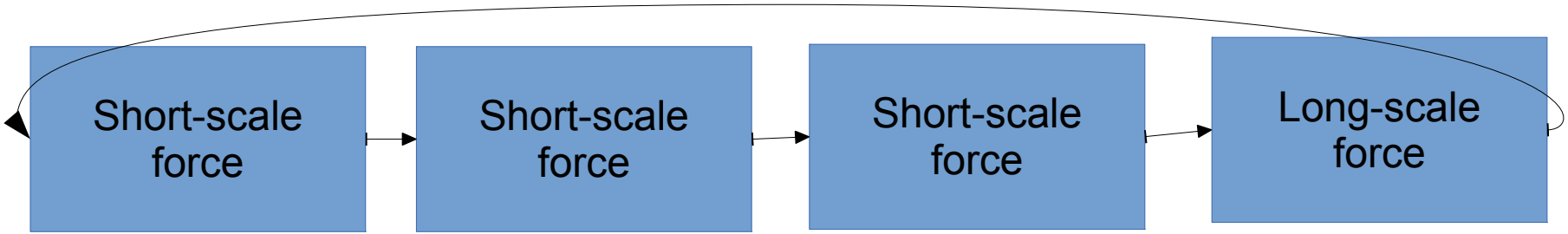
There is no general automatic way to do these things: You'll need to hack your build system. It will be worth it.

Some thoughts on testing

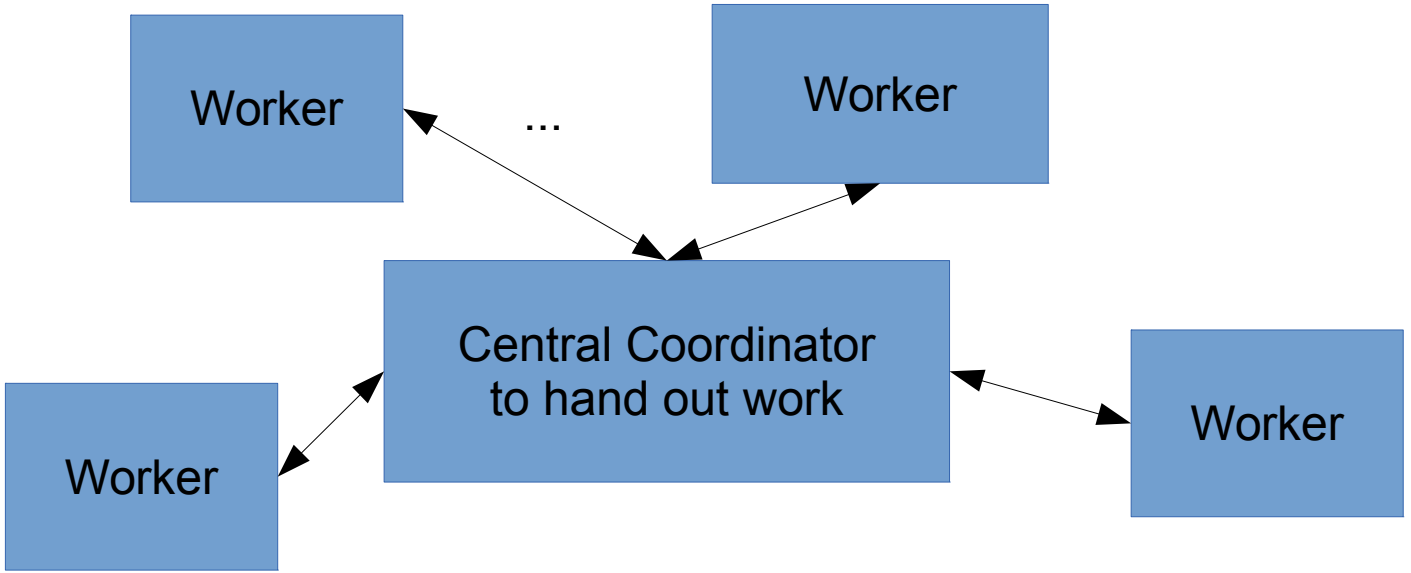
- Make as much of your code testable at small scale as possible.
- Unit testing is trendy for a good reason.
- Learn how to use Valgrind, and run your code at small scale with it.
- Add print statements in your code for anomalous situations: lots of them.
- Make sure you actually check for error return codes on routines that have them (for MPI, communication failures will kill your application by default, file I/O errors won't).

Avoid the network

- Network bandwidth, relative to FLOPS, is decreasing
- Choose, to the extent possible, communication-avoiding algorithms
- If your problem has multiple physical time/length scales, try to separate out the shorter/faster ones and keep them rank-local (local sub-cycling).
- More generally, learn about split-operator methods.



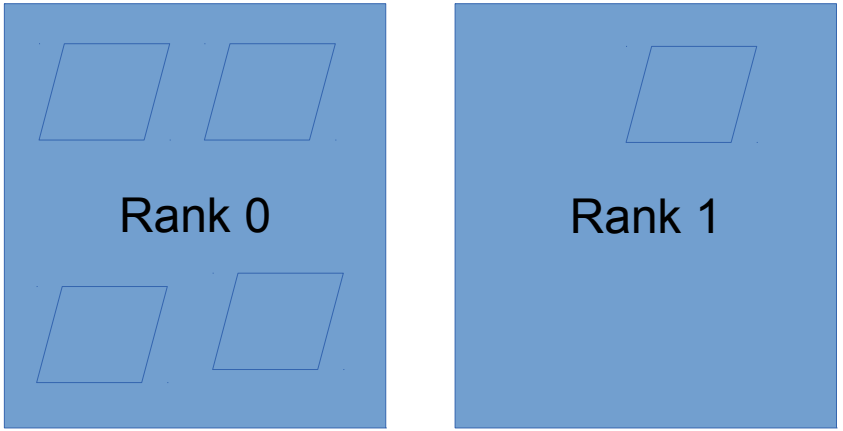
Avoid central coordinators



A scheme like this is highly unlikely to scale!

Load Balancing

- Keep "work units" being distributed between ranks as large as possible, but try hard to keep everything load balanced.
- Think about load balancing early in your application design: it is the largest impediment to scaling on large systems.



This is not good; rank 0 has much more work.

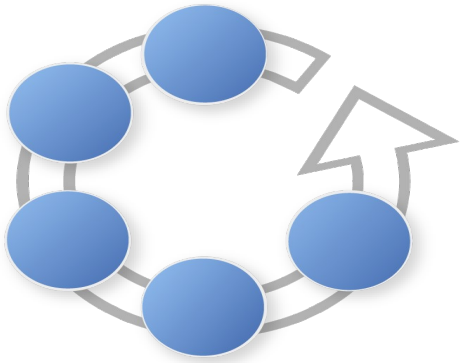
Memory Bandwidth

- Memory bandwidth will be low compared to the compute capability of the machine.
- To get the most out of the machine, you'll need to use FMA instructions (these machines were built to evaluate polynomials, many of them in parallel, so try to cast what you're doing in those terms).
- Try to do as much as possible with every data value you load, and remember that gathering data from all over memory is expensive.

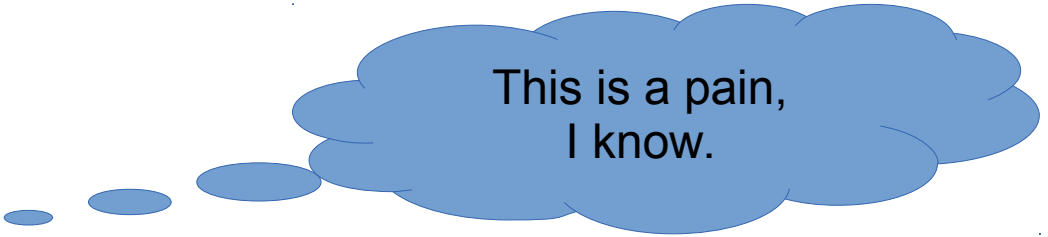
FMA = Fused Multiply Add = (a * b) + c [with no intermediate rounding]

Expensive Algorithms

- Don't dismiss seemingly-expensive algorithms without benchmarking them (higher-order solvers, forward uncertainty propagation, etc. all might have high data reuse so the extra computational expense might be “free”).
- If you're using an iterative solver, the number of iterations you use will often dominate over the expense of each iteration.

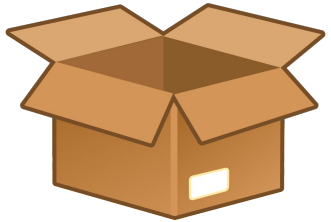


More on I/O



- Make your files “write optimized”.
- Don't use one file per rank, but don't have all ranks necessarily write to the same file either: make the number of files configurable.
- The optimal mapping between ranks and files will be system specific (ask the system experts what this is).
- There is often lock contention on blocks, files, directories, etc.
- Pre-allocate your file extents when possible.
- Use collective MPI I/O when the amount of data per rank is small (a few MB or less per rank).

More on Concurrency



- Use OpenMP, and learn about vectorization.
- In the future, you'll be able to use OpenMP for GPU programming too.
- Learn about structure of arrays, arrays of structures, and blocking to make a mixture of both.
- Don't be afraid to pick data structures that make vectorization (or, especially on GPUs, coallescable memory accesses) easier.
- You might need different data structures for different systems: try to make the science parts of your code agnostic to the data-structure implementations (also makes unit testing easier).

On OpenMP

- OpenMP is not the “one true way”, but it is popular in HPC for good reasons.
- schedule(dynamic), etc. will be your friend.
- When using OpenMP v4 with GPUs, you'll probably want: target teams distribute

```
#pragma omp parallel for schedule(dynamic)
  for(i=0; i<n; i++) {
    unpredictable_amount_of_work(i);
}
```

https://msdn.microsoft.com/en-US/library/9w1x7937.aspx

Conclusions

- HPC is hard but manageable with forethought and planning.
- Be rigorous in your testing, documentation, and archiving.
- Optimization is important, good algorithms more so.
- Go forth and learn many things!

- ALCF is supported by DOE/SC under contract DE-AC02-06CH11357