From your desktop
... to the cluster
... to the grid

Introduction

- You: hopefully have some computations running on your desktop PCs
- This module talks about making those applications run in bigger places.
- bigger places = clusters, grids
- some ideas of parallel and distributed computing from that perspective – but this is not a general parallel computing course, nor is it a general distributed computing course

brief overview of scales

what is a PC?

- the thing you have on your desktop or lap
- 1 ... 4 CPU cores (eg my laptop has 2 cores)

what is a cluster?

- Lots of PC-like machines, stuck together in a rack
- Additional pieces to make them work together
- UJ Cluster

what is a grid?

- (many different definitions)
- For now: Lots of clusters stuck together
- Additional pieces to make them work together
- Two grids especially relevant to UJ:
  - SA national grid
  - Open Science Grid

what is parallel?

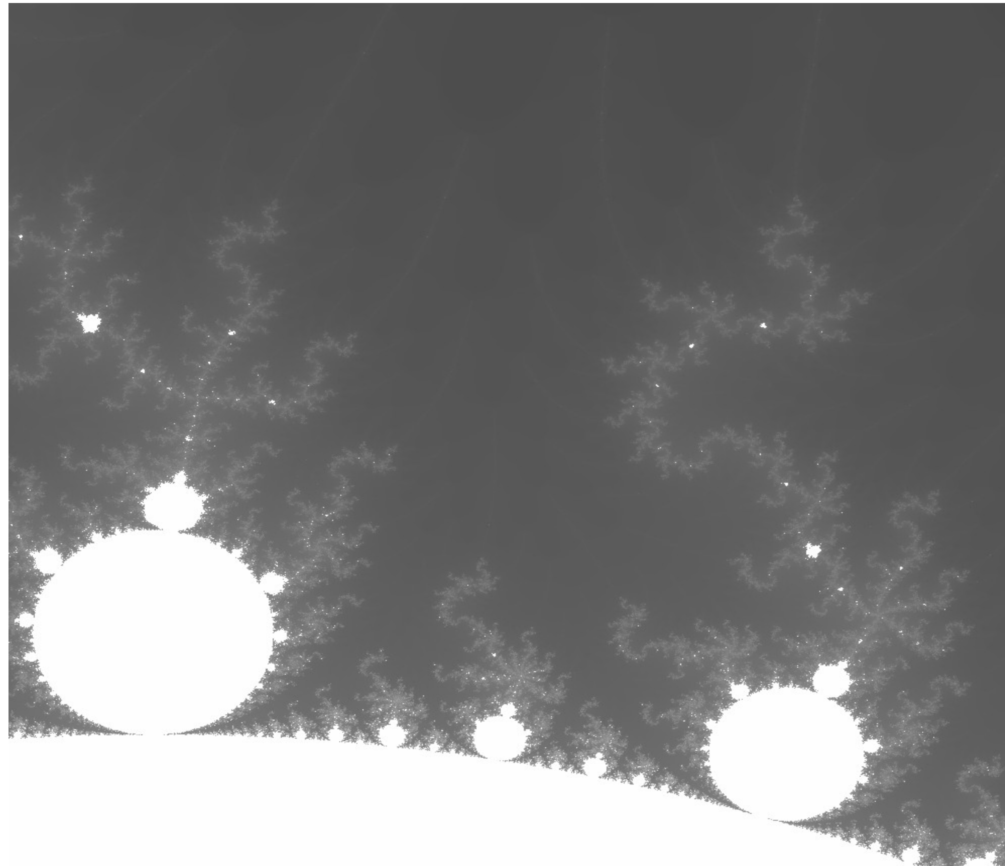
- Structuring your program so that pieces can run simultaneously
- This is how to take advantage of multiple CPU cores.

what is distributed?

- Structuring your program so that pieces can run in different places.
- Different places:
  - different nodes in a cluster
  - different sites in a grid

Example application

- Mandlebrot fractal rendering application as an example.
- Graphical rendering of a mathematical function
- You don't need to understand the maths involved
- This is “some scientific application”



mandelbrot

for x=0..1000, y=0..1000
each point (x,y) has colour determined by
function mandel(x,y);

If you don't like maths, close your eyes now

- mandel(x,y) is computed like this:
- $c=x+yi$
- iterate $z \rightarrow z^2 + c$
- shade is how many iterations before $|z|>2$
- http://en.wikipedia.org/wiki/Mandelbrot_set

A0. Mandelbrot on your desktop

we can run the following sequential pseudocode.
Easy to implement in many languages – I used C.

```
for x=0..1000
    for y=0..1000
        pixel[x][y]=mandel(x,y);
    endfor
endfor
```

baseline mandelbrot run

- implementation mandel10.c
- took 9m49s (589s) on my MacBook
- `time ./mandel10 0 0 1 0.0582 1.99965 200000  1000 1000 32000 > a.pbm`
- This measurement will be used to compare speedup for the rest of this module.

A1. Your multicore desktop

desktop multicore

- Multicore CPUs – put two CPUs on the same chip
- increasingly common – eg my laptop has two cores, cheapest mac laptop I could get
- Trivially: can run two separate sequential programs at the same time
- But what if we have one program that we want to use both cores?

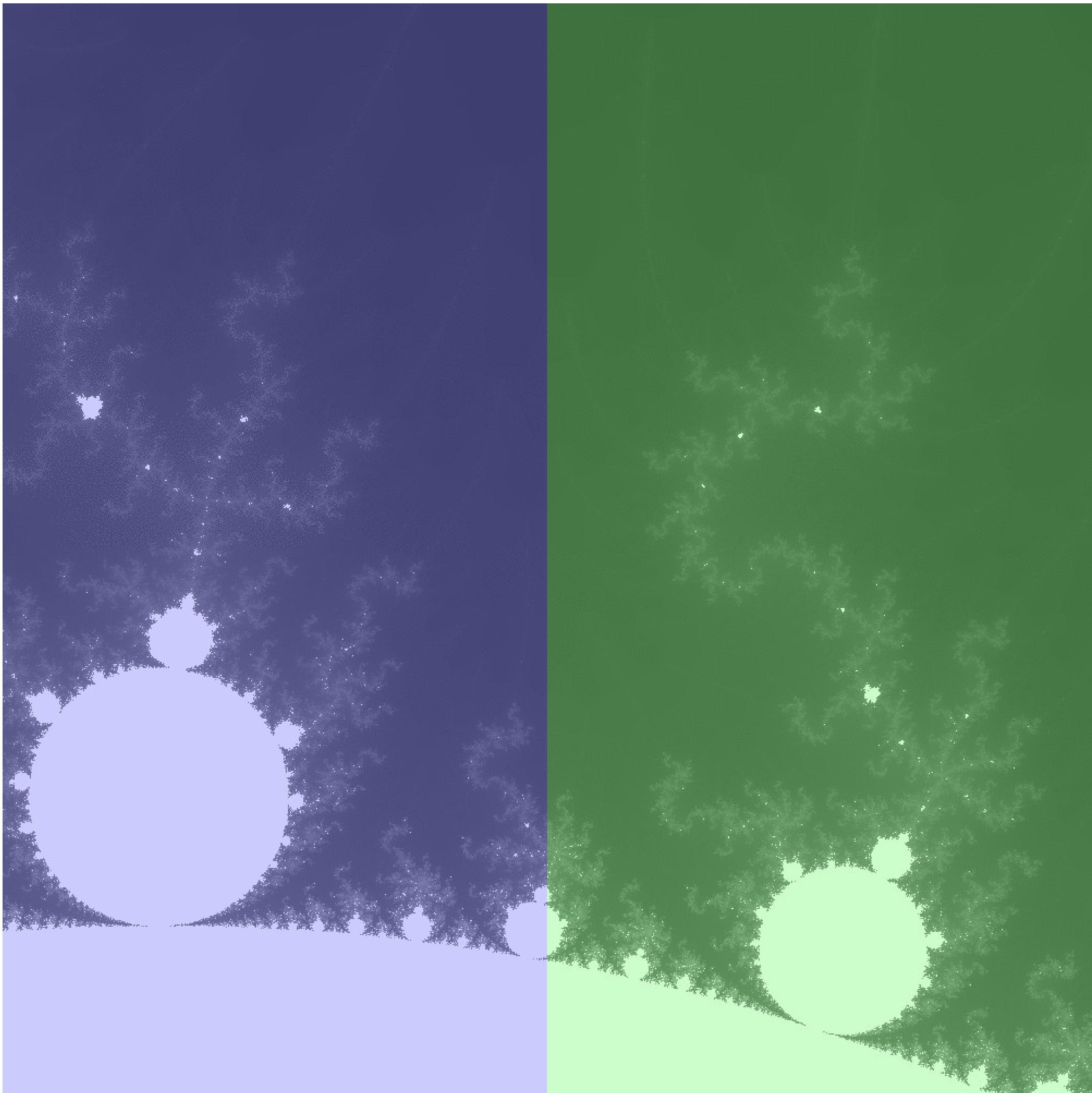
- Previous mandelbrot algorithm ran 10^6 computations in sequence.
- In the case of mandelbrot:
  - split the loops into two separate executables
  - run them independently, one on each CPU core
  - join the results when both are finished
  - hopefully faster?

parallelised mandelbrot

```
for x=0..499
    for y=0..1000
        pixelA[x][y]=mandel(x,y);
    endfor
endfor

for x=500..1000
    for y=0..1000
        pixelB[x][y]=mandel(x,y);
    endfor
endfor

pixel=combine(pixelA, pixelB)
```



parallelised mandelbrot

```
for x=0..499
    for y=0..1000
        pixelA[x][y]=mandel(x,y);
    endfor
endfor

for x=500..1000
    for y=0..1000
        pixelB[x][y]=mandel(x,y);
    endfor
endfor

pixel=combine(pixelA, pixelB)
```

timings

- Naively hope it would be twice as fast (because two CPU cores)
- In reality: duration (walltime) = 6m59s (419s)
- 589/419=1.4x speedup
- faster, but not twice as fast...
  - why? in a few slides.

Communication between parallel components

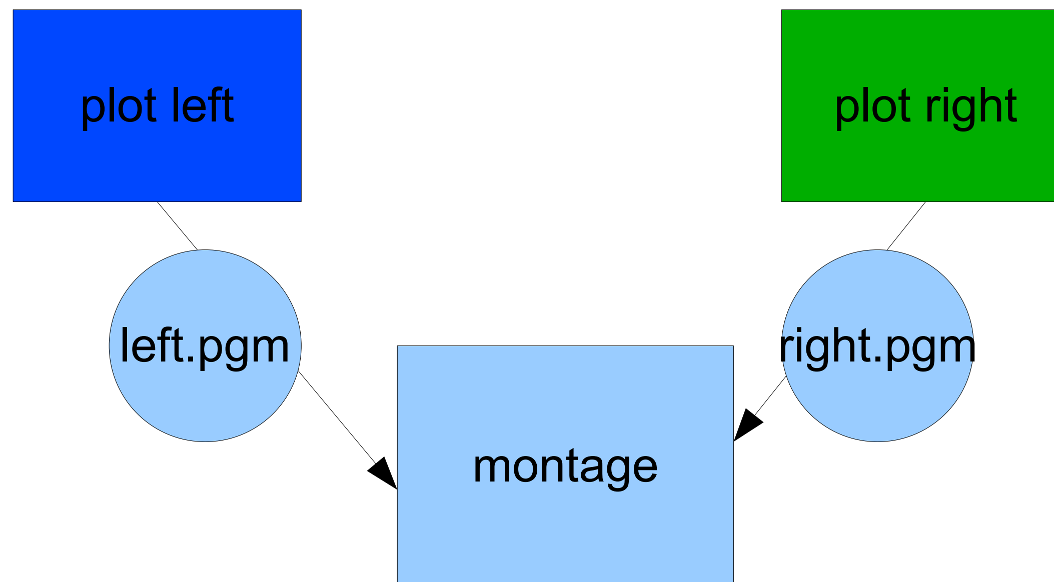
- Components running in parallel need to communicate with each other.
- In this mandelbrot example, communicate to:
  - tell code which half of the fractal to render
  - join the results together in a single picture

Loose file coupling

- Model used here is loose file coupling.
- This is not the best model for single PC multicore parallelisation, but it is flexible when moving between different scales.
- Components communicate using files and commandline parameters

mandelbrot

- mandel 0..499 > left.pgm &
- mandel 500..999 > right.pgm &
- wait
- montage left.pgm right.pgm all.pgm



```
$ cat tile-dualcore-1.sh

rm -v tile-*-*.gif
rm -v tile-*-*.pgm

for x in 0 1 ; do
 ( for y in 0 1 ; do
   ./mandel5 $x $y 2 0.0582 1.99965 200000 1000 1000 32000 > tile-
$y-$x.pgm
   convert tile-$y-$x.pgm tile-$y-$x.gif
  done ) &      # launch this iteration in the background
done

wait            # wait for all the iterations to finish

montage -tile 2x2 -geometry +0+0 tile-*-*.gif mandel.gif
```

- ./mandel5 $x $y 4 0.0582 1.99965 200000 1000 1000 32000 > tile-$y-$x.pgm
- $x and $y indicate which of 4 tiles will be rendered, tile-$y-$x.pgm is output file containing the image
- when all the tiles exist, we need to combine them together:
- montage -tile 2x2 -geometry +0+0 tile-*-*.pgm mandel.gif

timings again

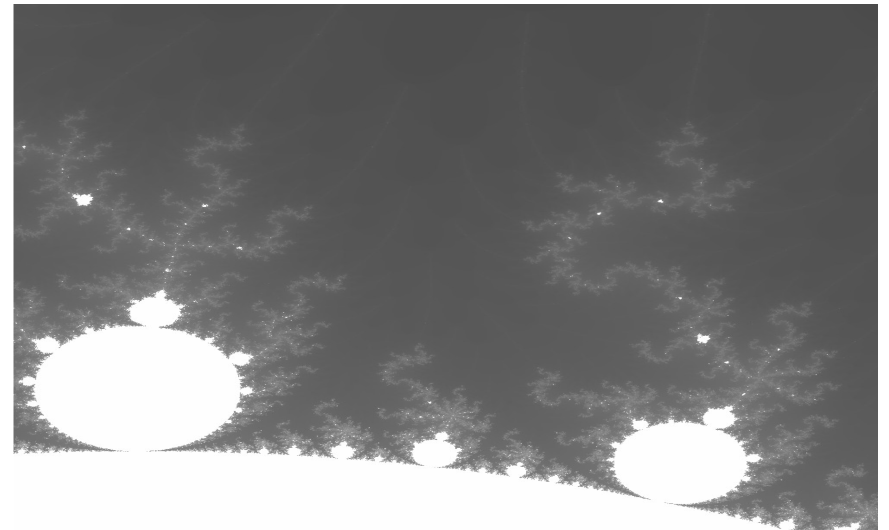
- from before: t(single) = 589s
- wall duration: 6m59s (419s) – 1.4x speedup
- Running these two tiles separately:
  - x=0 wall time: 410s
  - x=1 wall time: 172s
- max(t(0),t(1)) ~ t(wall) : 410 ~ 419 (5s extra)
- t(0) + t(1) ~ t(single) : 410+172=584 ~ 589
- limited by t(0)
- tile-dualcore-1.sh

Why are 2 chunks not enough?

- Why were 2 chunks not enough when we have 2 CPUs?
- Chunks don't all take the same amount of time – some take <1s, others take minutes.
- We don't know ahead of time how long each will take...

Time for each chunk to run, 16 chunk example

Y pos \ X pos	1	2	3	4
1	0	1	0	0
2	2	0	1	0
3	102	5	0	1
4	182	126	105	67



timings with n chunks instead of 2

- in this app we can get near to the theoretical limit of 2x fairly easily, but then doesn't get any faster.
- (plot of n vs time or n vs speedup)

n	t (s)	speedup
1	589	1
2	419	1.41
4	415	1.42
9	366	1.61
16	329	1.79
36	310	1.9
49	299	1.97
64	296	1.99
256	295	2

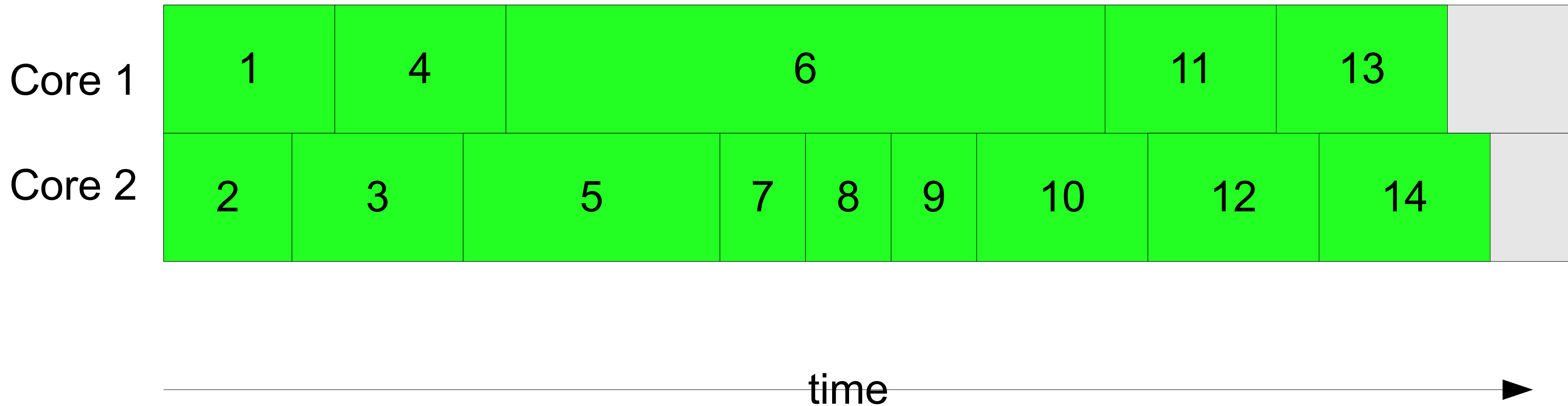
problem: different components have different timings

- in general can't tell ahead of time how long a component will take to run
  - (if you like CS, that is related to The Halting Problem)
  - (for some problems, we can estimate pretty well, though)

task farm model

- If we have n CPUs, split into n*10 tasks.
- Each CPU starts working on one task. When its finished, it takes another one.
- If a CPU gets a quick task, it will quickly finish and move onto the next
- If a CPU gets a slow task, other CPUs will handle the other tasks.
- If a new CPU becomes available, it will start performing tasks.

task farm diagram again



Even though jobs are of very different duration, we get fairly even distribution of load.

But... we need enough jobs for this to happen.

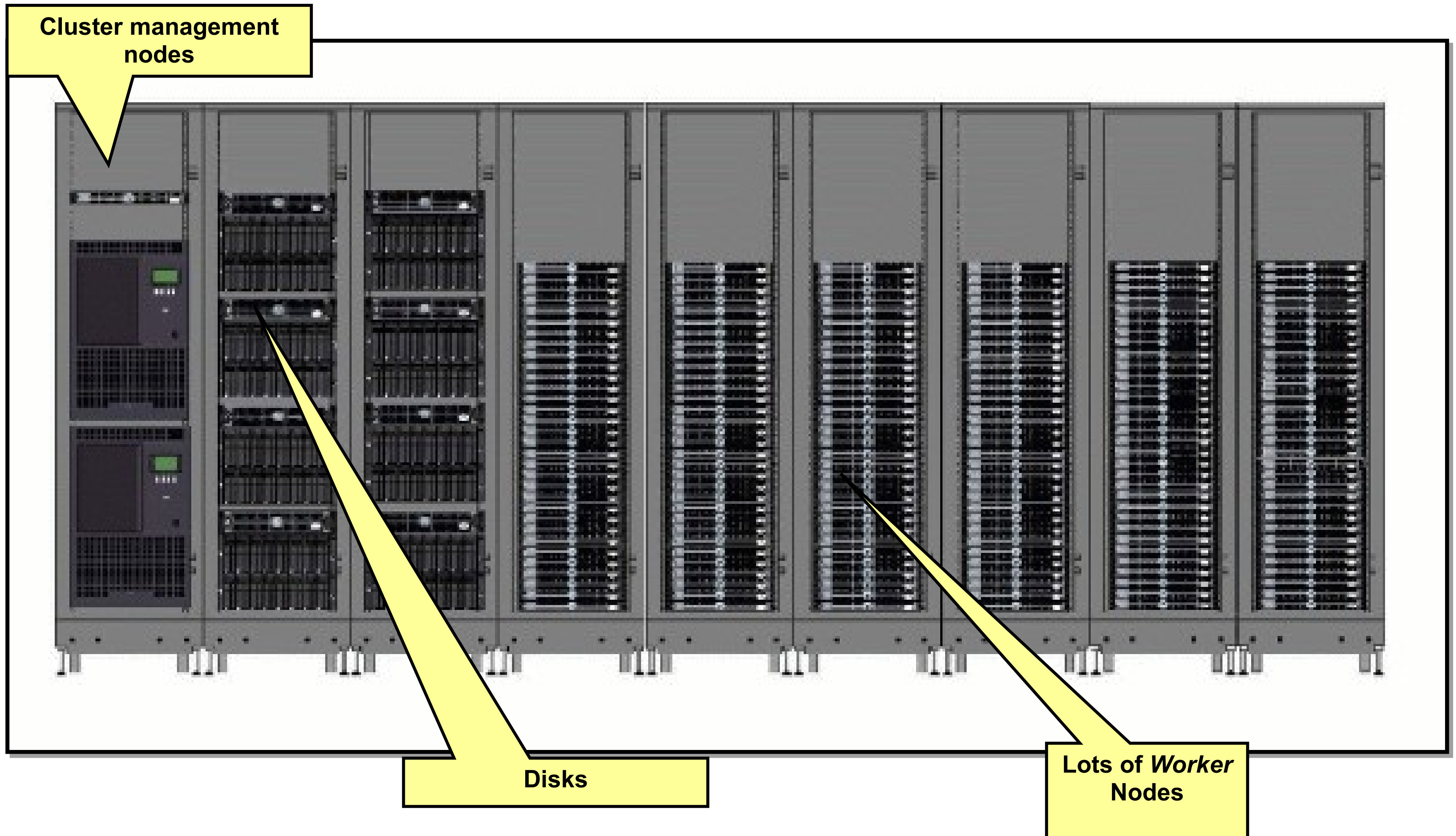
other models of computing on a multicore CPU

- Shared memory parallelism
  - one program
  - shared memory
  - rather than fork two unix processes, fork threads inside your program, with each thread able to access the same memory

B. distributing the work

- so how can we use more CPU cores than we have in one desktop machine?
- we can render different tiles of the fractal on different computers
- how?
  - we need to co-ordinate so that all the tiles get rendered, and so that we don't duplicate work
  - we need to get all the results into one place so we can assemble them into a single picture
- Look at two distributed models:
  - clusters – distributed computation between PC-like nodes in the same physical location and under same administration
  - grid – distributed computation between clusters widely separated geographically, under different administrations

C. clusters



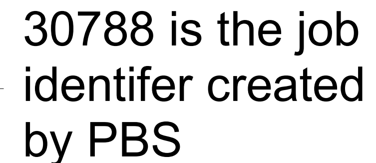
Batch queueing system / local resource manager

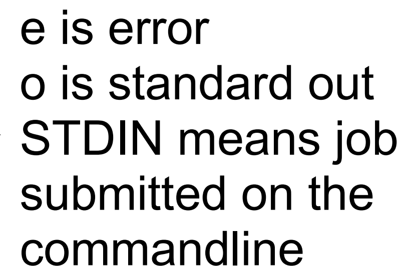
- Different people use different names for the same thing:
  - Batch queueing system
  - Local resource manager (LRM) in grid-speak
- PBS (Portable batch system) on UJ cluster
- Allocates nodes to jobs so that one job has one CPU

Submitting jobs to PBS with qsub

- qsub command submits a job to PBS

```
$ qsub
echo hello world
<CTRL-D>
30788.gridvm.grid.uj.ac.za
$ ls STDIN.*30788*
STDIN.e30787  STDIN.o30787
$ cat STDIN.o30787
hello world
```





Watching the queue with qstat

- qstat command shows jobs in the queue

```
$ qstat
Job id             Name            User            Time Use S Queue
------------------ --------------- --------------- -------- - -----
30184.gridvm       null            benc            00:39:35 R batch
30272.gridvm       null            benc            00:32:11 R batch
30736.gridvm       null            benc            00:18:00 R batch
30737.gridvm       null            benc            00:18:09 R batch
30755.gridvm       null            benc            00:18:25 R batch
30779.gridvm       null            benc            00:17:08 R batch
30780.gridvm       null            benc            00:17:06 R batch
```

job status:
Q = queued
R = running
C = completed

queue that the job was submitted to

job ID corresponds with ID from qsub

who submitted the job

Time the job has been running

shared file system

- This cluster (and many clusters, but not all) have shared file system.
- We can create a file here on any worker node, and access it from any other worker node (or the head nodes)

```
$ df -h /nfs/data
Filesystem            Size  Used Avail Use% Mounted on
gridvm:/data           385G   21G  345G   6% /nfs/data
```

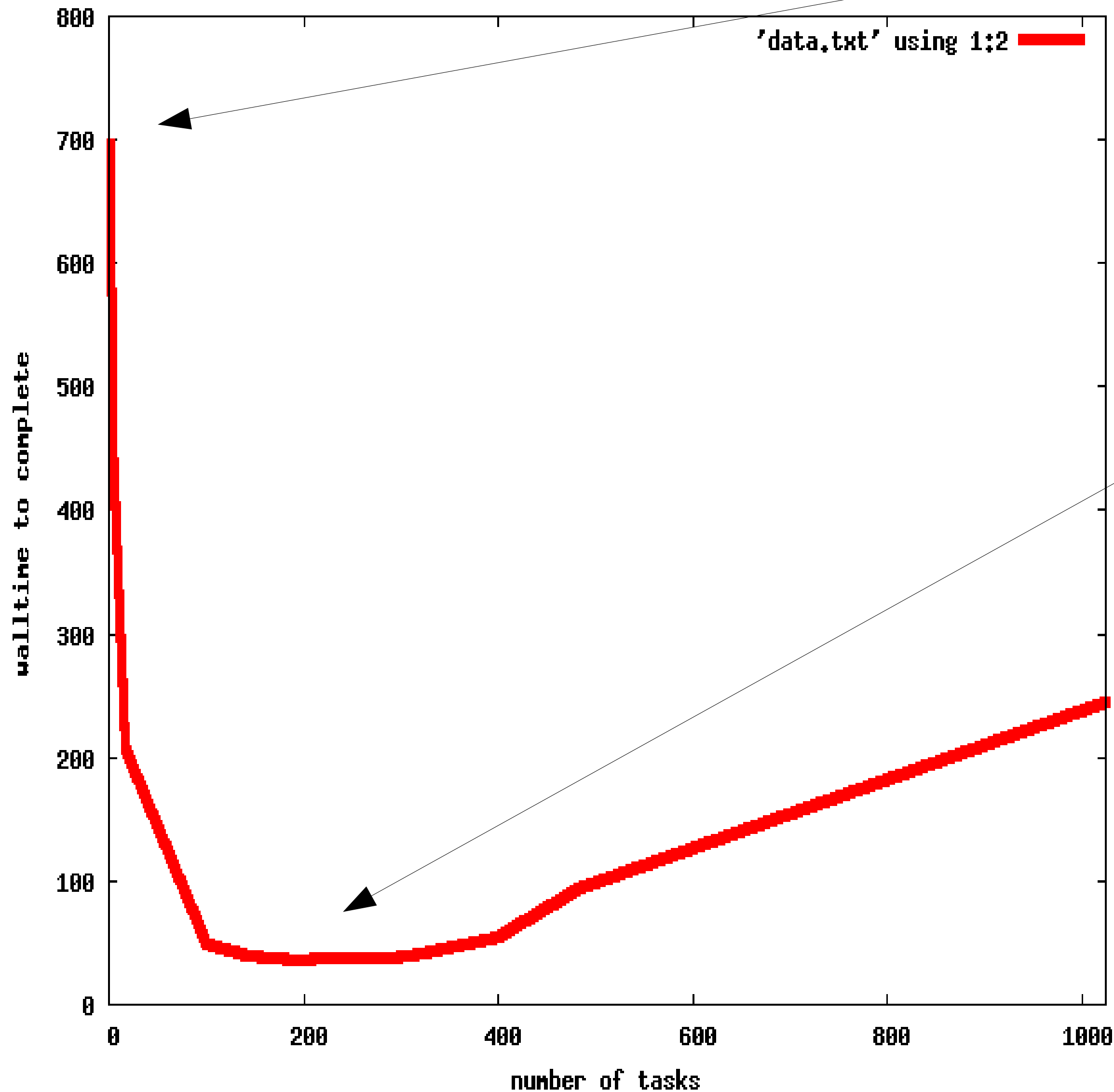
mandelbrot on the cluster

- On the UJ cluster, there are 56 CPU cores. Can we get 56x speedup?
- graphs
- here we can perhaps talk a bit about how application sizes change how much speedup we can get – eg a 10 minute mandelbrot is probably not going to get much use out of 100 cores, but a 100 hour mandelbrot probably can
- plots showing this
- extended app: animation (zoom in from full set to the two pretty pretty frames that I've found)

implementing mandelbrot on the cluster

- Divide into tiles (like in multicore case)
- Submit each tile separately using qsub
- Put output tiles on the shared file system
- When all of the tiles are finished (when qstat shows we have no jobs left) then run the montage command (qsub it or run on login node – probably better to qsub it; or run montage on my laptop using sshfs?)
- View output on laptop (can use sshfs to get cluster shared fs on my laptop too)
- TODO: colour tiles by worker node, so can see some worker nodes did only a small number of expenseive tiles and some did lots of cheap tiles

single-frame timings



- 1 job: compare speed of cluster node with speed of my laptop: 697s (actually slower than my laptop)
- we can get up to around 17x speedup, at 14x14 tiles
- not near 56x theoretical speedup
- overhead dominating
- Try a bigger app...

data for previous slide

sqrt(n)	n	t	speedup
1	1	697	0.85
2	4	458	1.29
4	16	207	2.85
10	100	48	12.27
12	144	39	15.1
13	169	38	15.5
14	196	35	16.83
15	225	37	15.92
16	256	37	15.92
17	289	37	15.92
18	324	41	14.37
20	400	54	10.91
22	484	95	6.2
32	1024	245	2.4

Mandelbrot animation

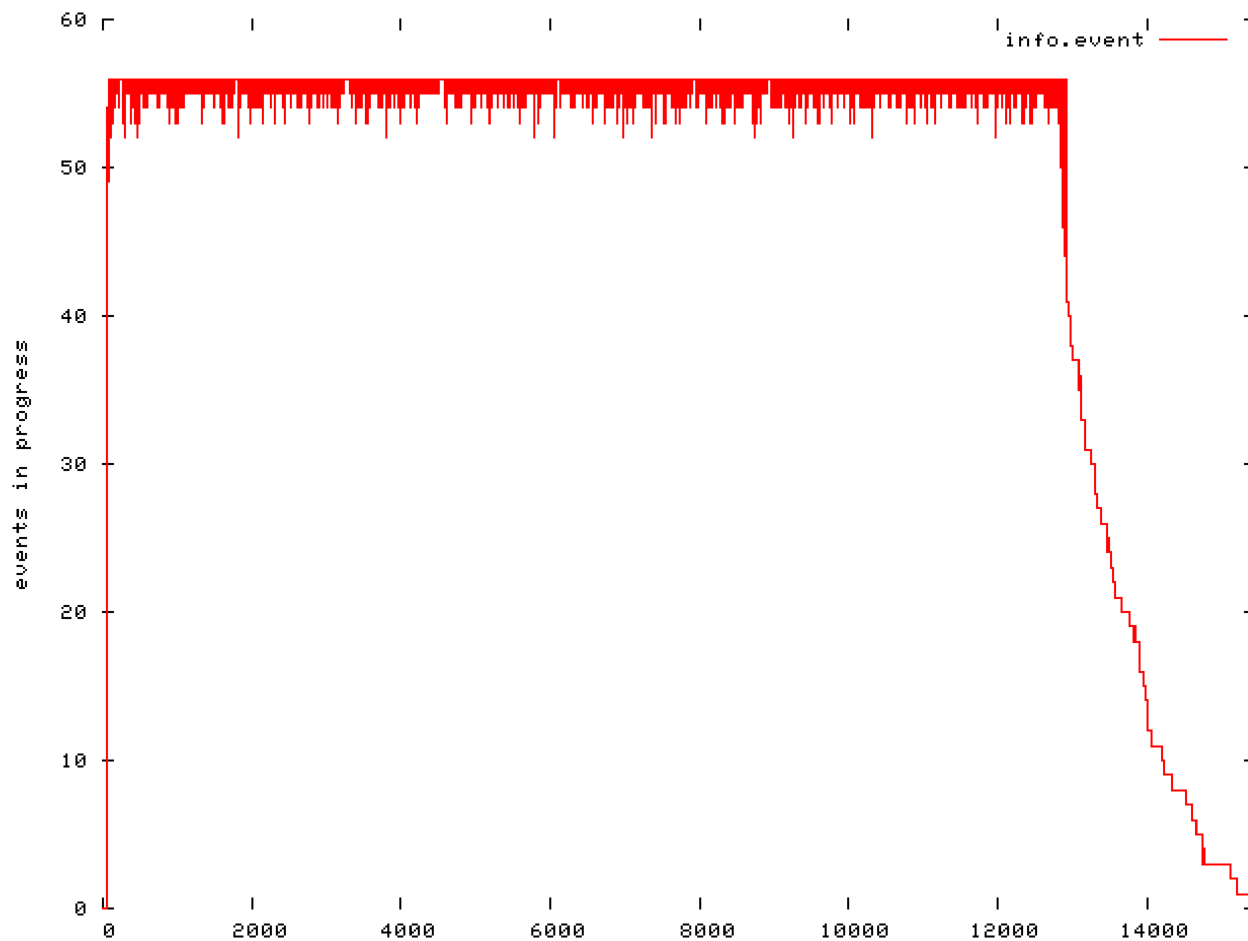
- Generate an animation of moving around the mandelbrot space
- Each frame is rendered like we rendered the static image in earlier sections.
- Stitch all the frames together in sequence to give an animation.
- multidimensional parameter sweep (more than the 2 parameters x,y earlier)

mandelbrot animation implementation

- one task = one tile of one frame
- parameters that vary are x,y,zoom (and perhaps others)
- tasks to join tiles into frames
- one final task to join frames into an animation
- same pattern as tiled rendering; common pattern: apply same operation to lots of data, then one final operation to combine results

Some animation statistics

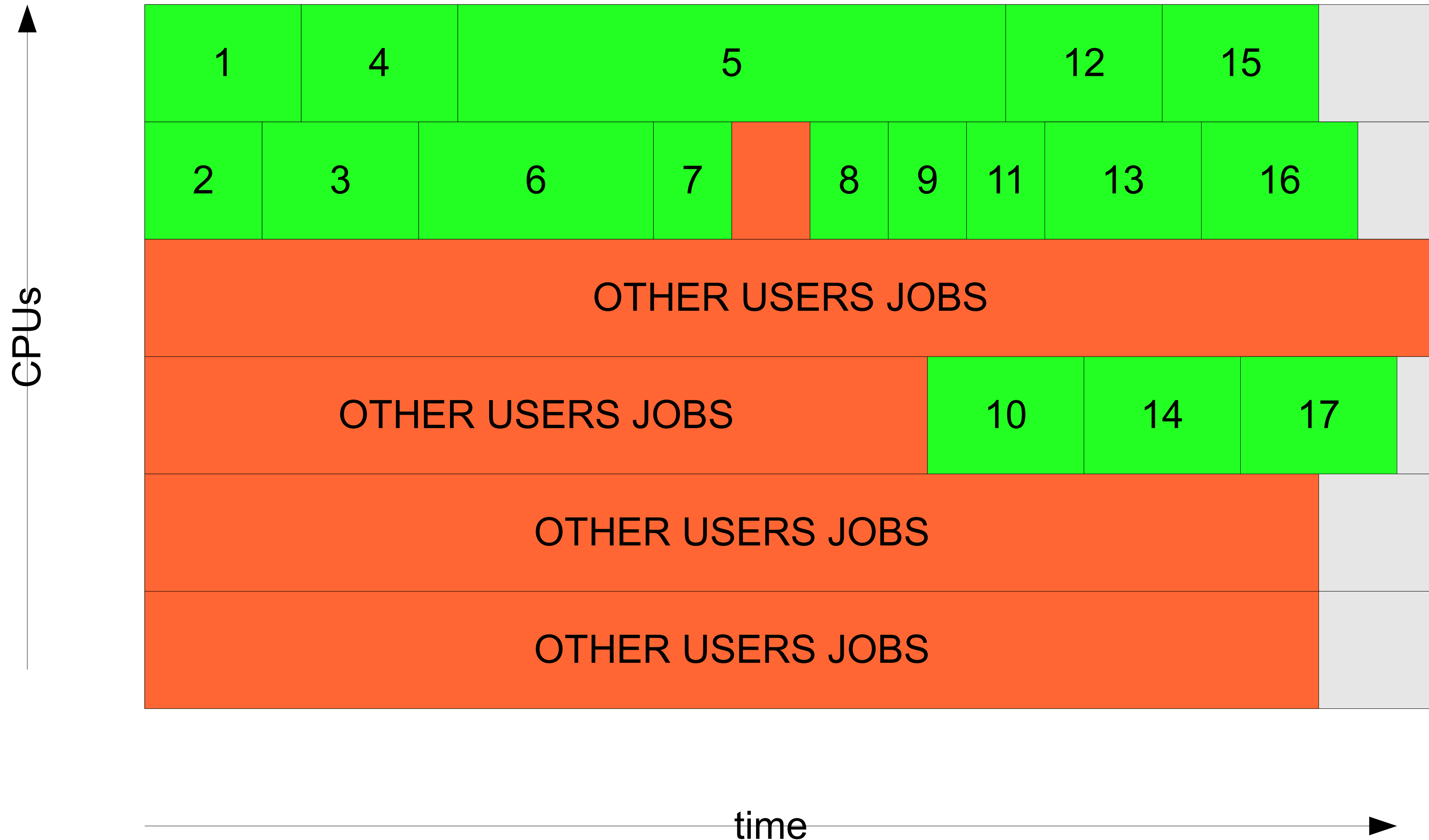
- I ran this with 840 frames – using all 56 cluster cores for most of the run, 15000s = 4.2 hours



problem: we have varying numbers of CPUs available

- start computation on cluster. only 10 CPUs available because someone else is running a big job
- so perhaps we split job into 10 components and start them running.
- now that other persons big job finishes, and 50 more CPUs are available. (chart of CPU use – time x CPUs coloured green for us, and red for them and white for empty)
- we can't use those newly freed CPUs

task farm diagram again



Other cluster programming models

- MPI – Message Passing Interface
  - very common
- PVM – Parallel Virtual Machine

change in the way of doing science: run taking a few seconds

- When a run takes a few seconds to run, we can interact with the computation very differently – explore parameter space much more rapidly, try out new algorithms, etc – 100s of times per hour, perhaps, compared to 6 per hour.
- Pretty much interactive exploration, rather than starting a run and getting distracted over the next 10 minutes.

think

- Something to think about for your applications:
- what kinds of shift can you make like this?
- example:
  - drug testing – in-silico testing of entire drug database against new diseases

D. The grid

- Introduction talked about the Open Science Grid
- Later modules will talk about that and grids in general in much more depth
- 'grid' is a nebulous term, but for this particular module:
  - a collection of clusters spread around the place, all owned by different people

Running on the grid

- In some respects similar to running on a local cluster
- But in other respects, very different
  - fault tolerance
  - application installation
  - security
  - high network latency and low network bandwidth
  - scheduling and site selection

fault tolerance

- something *will* break
- cannot avoid faults, so need to tolerate them
- one of the causes of 'brittle' systems is the assumption that probably everything will work, and that it is unusual for something to break.
- in cluster case, worker nodes can go wrong sometimes. in grid case, all cluster problems plus more – network failures and site failures. something somewhere is always broken somehow

application installation

- On cluster, because heterogeneous workers and usually a shared fs, relatively easy to install complex apps once and use from all workers
- On grid, need to do per site. Lots of effort. Approach from two directions:
  - using common packages like R, a VO software team maintains R on many sites, for many users to use. (get someone else to deploy your software)
  - write your codes using more commonly available software/libraries, and statically link (write your software so it does not need complex deployment)

security

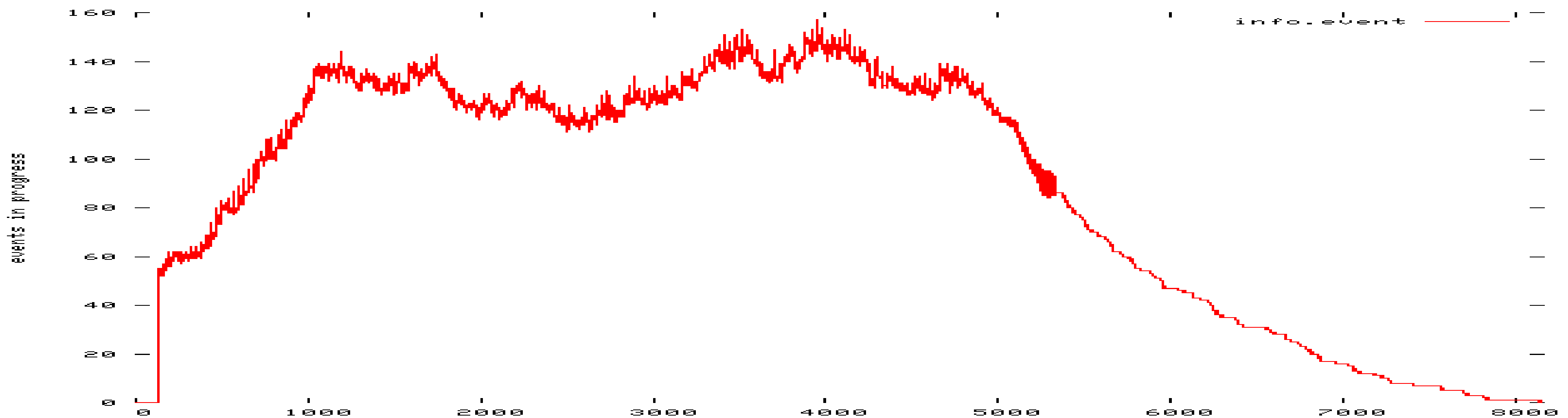
- How can I prove that I am allowed to use a resource?
- How can I stop people seeing or changing my data in transit?

network characteristics

- Within a cluster, usually have quite fast network connection between all of the components – they all sit in a rack together and have ethernet connecting them
- On grid, sites have wide area links that are typically:
  - high latency
  - low bandwidth
- This significantly changes application performance characteristics

Mandelbrot on Open Science Grid

- I ran the same 840 frame animation on both the UJ cluster and the Open Science Grid
- 5 sites (these were the ones that worked straight away without me trying very hard - ~30 were available in total)
- peak of 160 cores at once (could go much higher, but my runs were made cautiously)



Think...

- How can your apps grow from cluster scale to grid scale?

Review: scaling up

- single core PC
  - no synchronisation necessary, only one component so no connections, you own everything – no sharing. 10^0 cores. good for small apps, 10^0 CPU hours
- multicore PC
  - shared memory, shared fs, components connected closely, you own everything – no sharing. 2*10^0
- cluster
  - no shared memory, often (but not always) a shared filesystem, components connected by local area network (100mbit...10gbit). components are multicore PCs. you share access with others nearby. all nodes configured the same. 10^2..10^3 of cores. apps: 10^3 CPU hours
- grid
  - no shared fs, components are connected by wide area network (high latency, low bandwidth, expensive). you share access with large numbers of people. different sites configured very differently. 10^4 cores (or ^5? what are latest OSG/SANG/TG stats?)

- fin

def/concept

- parallel
- should point out differences between the parallel stuff here (which is fairly loosely coupled) vs MPI-style tightly coupled computing
- differences in ease of programming – high concurrency is very hard to think about. but can achieve more.

def/concept

- distributed

def/concept: metrics

- walltime
- CPU-time (CPU-hours)
  - time it takes to run on a single core
  - number of hours used on all cores added together
- speedup = w/c
- or perhaps...
- speed = walltime(cluster) / walltime(single)
  - where the single time is the best single-core algorithm, rather than the same algorithm as the cluster...

def/concept: checkpointing

- not too much depth here, but put in somewhere near fault tolerance

why no shared fs on grid?

- large scale shared fs exists (for example, AFS)
- but performance poor
- even on large clusters, it can be hard to get a decent performing shared fs
-
- (perhaps this belongs in a different slide?)