More advanced application techniques, using Swift

about this talk

- Talk about project that I spend most of my time working on: Swift
- Use that to introduce some more general concepts – in describing applications and in executing applications
- TODO: lots of timing data and the like in that swift advanced docbook that I made before – thoughts from that (if not apps and timing) should be absorbed

swift as a system for clustered and grid applications

- application descriptions made ad-hoc in previous stages can be described using SwiftScript
- helps address scale variation – can run same script and apps on my laptop, on a PBS cluster, and on grid
- many applications have same patterns – Swift provides these so that you don't have to implement them yourself (badly)
- many common problems that are deal with once in Swift rather than reimplemented

the language

- file data represented as variables
  - mappers
- dataflow/functional feel
- you express how your application fits together and Swift deals with the mechanics of execution
- TODO introduce enough of the language to be able to discuss the mandelanimdeploy.swift app

the language

- Previously, the code that has driven job submissions has been ad-hoc – shell scripts, or manually generated DAGs in DAGman
- Swift provides a higher level language for describing things

the runtime

- TODO list the titles of the slides later after the language

“hello world” in Swift

```
$ cat first.swift

type messagefile;

app (messagefile t) greeting() {
    echo "Hello, world!" stdout=@filename(t);
}

messagefile outfile <"hello.txt">;

outfile = greeting();

$ swift first.swift

Swift 0.8 (stripped) swift-r2448 cog-r2261

RunID: 20090406-1145-08buadea
Progress:
Progress:  Finished successfully:1

$ cat hello.txt
Hello, world!
```

“hello world” in Swift

```
type messagefile;

app (messagefile t) greeting() {
    echo "Hello, world!" stdout=@filename(t);
}

messagefile outfile <"hello.txt">;

outfile = greeting();
```

- Describe component programs
- Define 'app' procedures that invoke unix programs
- Define inputs and outputs
  - here: no inputs, one output file into which the stdout of echo will go

“hello world” in Swift

```
type messagefile;

app (messagefile t) greeting() {
    echo "Hello, world!" stdout=@filename(t);
}

messagefile outfile <"hello.txt">;

outfile = greeting();
```

- Describe data
- Define variables which are mapped to disk files
- Here: a variable called outfile, which represents the file hello.txt

“hello world” in Swift

```
type messagefile;

app (messagefile t) greeting() {
    echo "Hello, world!" stdout=@filename(t);
}

messagefile outfile <"hello.txt">;

outfile = greeting();
```

- Invoke the application
- Use programming language syntax
- Here: invoke greeting, putting the output in outfile

“hello world” in Swift

```
type messagefile;

app (messagefile t) greeting() {
    echo "Hello, world!" stdout=@filename(t);
}

messagefile outfile <"hello.txt">;

outfile = greeting();
```

- Describe data types
- Here: a simple data file
- There are strings, ints, floats
- There are more complicated structures

variables and mappings

- `messagefile outfile <"hello.txt">;`
- Files are represented as variables.
- The <...> bit maps the variable to a file
- More complicated mappings are possible
- file tile[][] <simple_mapper;suffix=".pgm">;
- A 2-dimensional array – Swift constructs the names, and puts .pgm on the end

the application execution model

```
app (messagefile t) greeting() {
    echo "Hello, world!" stdout=@filename(t);
}
```

- DO:
  - describe input files
  - describe what to run
  - describe output files
- DON'T:
  - describe where a job is to run
  - how to run a job
  - assume jobs will have access to other data

TODO the application execution model

- diagram showing stagein, execution, stageout (but not site selection)

site and transformation catalogs

- SwiftScript describes application dataflow
- Separate descriptions for:
  - execution sites – the site catalog, sites.xml
  - applications on sites – the transformation catalog, tc.data

The Site Catalog

A short name for the site

How to transfer files to and from the site

How to run jobs on the site

```
<pool handle="localhost">
  <gridftp  url="local://localhost" />
  <execution provider="local" />
  <workdirectory >/var/tmp</workdirectory>
  <profile namespace="karajan"
     key=”jobThrottle">0</profile>
</pool>
```

Where we can store temporary files on the site

Profile keys that specify assorted other settings

mandel.swift

```
type file;

int side = 8;

file tile[ ][ ] <simple_mapper;suffix=".pgm">;
file mandel <"mandelbrot.gif">;

app (file result) render(int x, int y, int side) {
  mandel x y side 0.0582 1.99965 200000 1000 1000 32000 stdout=@result;
}

app (file frame) montage(file tiles[][], int side) {
  montage "-tile" @strcat(side,"x",side) "-geometry" "+0+0" @filenames(tiles) @frame;
}

foreach x in [0:(side-1)] {
  foreach y in [0:(side-1)] {
    tile[y][x]=render(x,y, side);
  }
}

mandel=montage(tile, side);
```

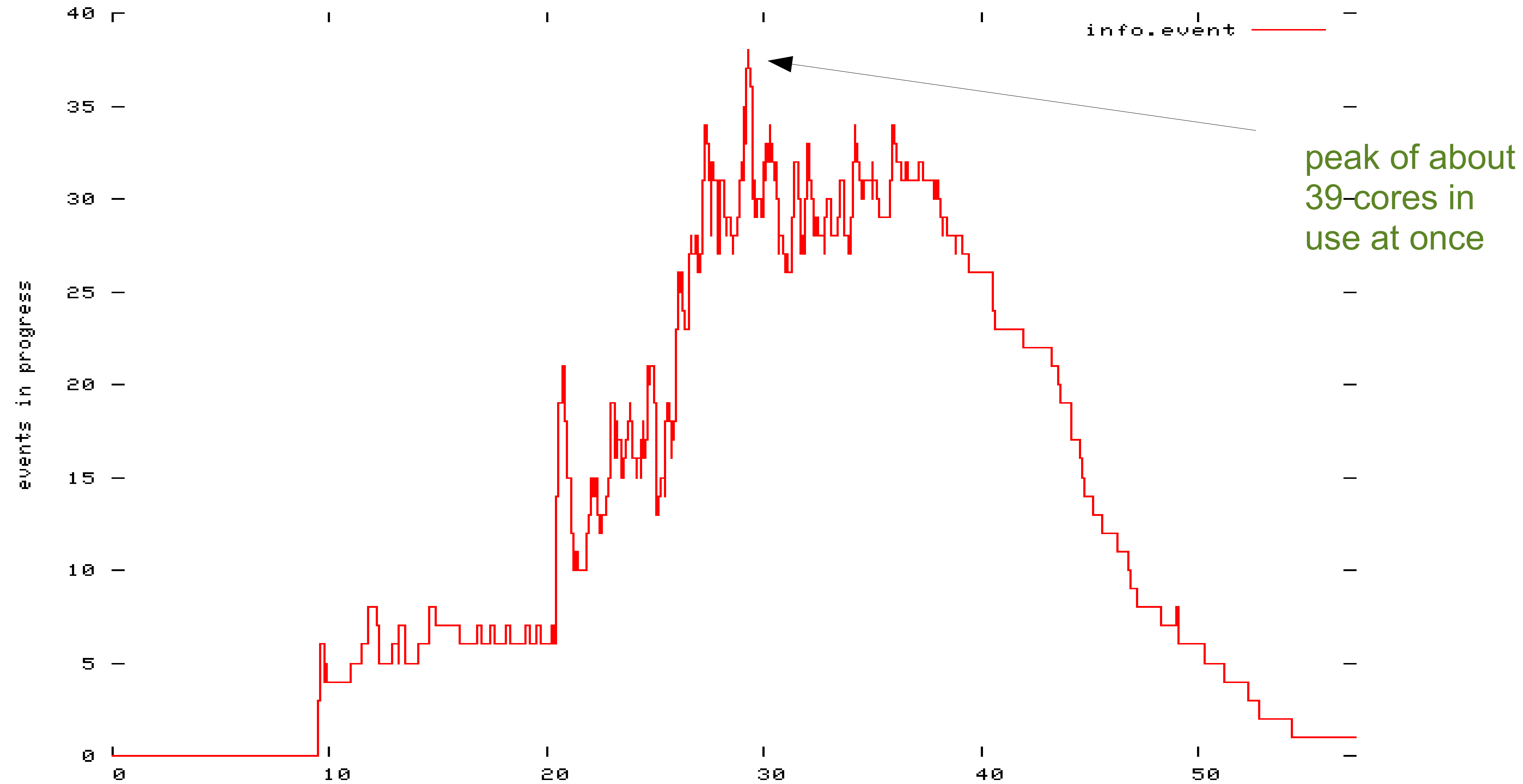
foreach

```
foreach y in [0:(side-1)] {
  tile[y][x]=render(x,y, side);
}
```

- foreach allows iteration over arrays
- easy to express a parameter sweep
- TODO if I have not introduced the term 'parameter sweep' earlier, introduce it here

Running mandel.swift on the cluster

$ swift mandel.swift



http://152.106.18.254/~benc/report-mandel-20090402-1924-2rjhm937/

mandelbrot animation using swift

- single-frame mandelbrot generation becomes a procedure
- read in frame parameters, call frame generator on each one, store results in an array
- final step to combine frames into an animation
- a third dimension to our parameter sweep (x, y, frame)

mandelbrot frame parameters

- all those apparently arbitrary constants in mandel5 command lines earlier.
- make a complex type in Swift to store these

```
type frameparameters {
    int iterations;
    int zoom;
    float yoff;
    float xoff;
}
```

mandelbrot frame parameters

- reference members of a complex type like C structs or Java objects, using a dot

```
app (file result) render(int x, int y, int side, frameparameters ap) {

  mandel x y side ap.xoff ap.yoff ap.iterations 1000 1000 ap.zoom
stdout=@result;

}
```

mandelbrot frame parameters

- read in the parameters with readData

```
file specificationFile <"framespec.data">;
frameparameters spec[] = readData(specificationFile);
```

```
$ cat framespec.data
iterations zoom yoff xoff
100 32000 1.99965  0.0582
333 32000 1.99965  0.0582
1000 32000 1.99965  0.0582
3000 35000 1.99965 0.0582
1000 38000 1.99965 0.0582
10000 41000 1.99965 0.0582
30000 45000 1.99965 0.0582
100000 49000 1.99965 0.0582
300000 53000 1.99965 0.0582
```

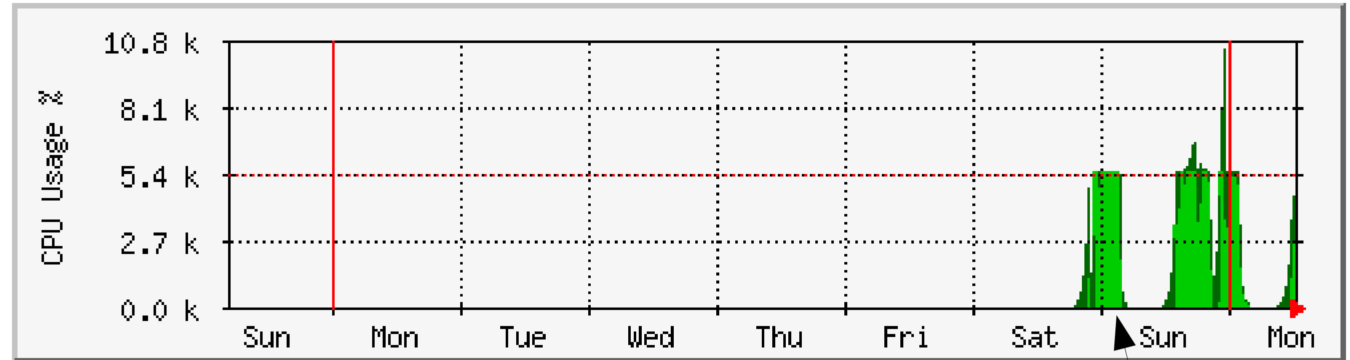
header line matching definition of frameparameters

one row per array element (frame)

Running mandelanim.swift on the UJ cluster

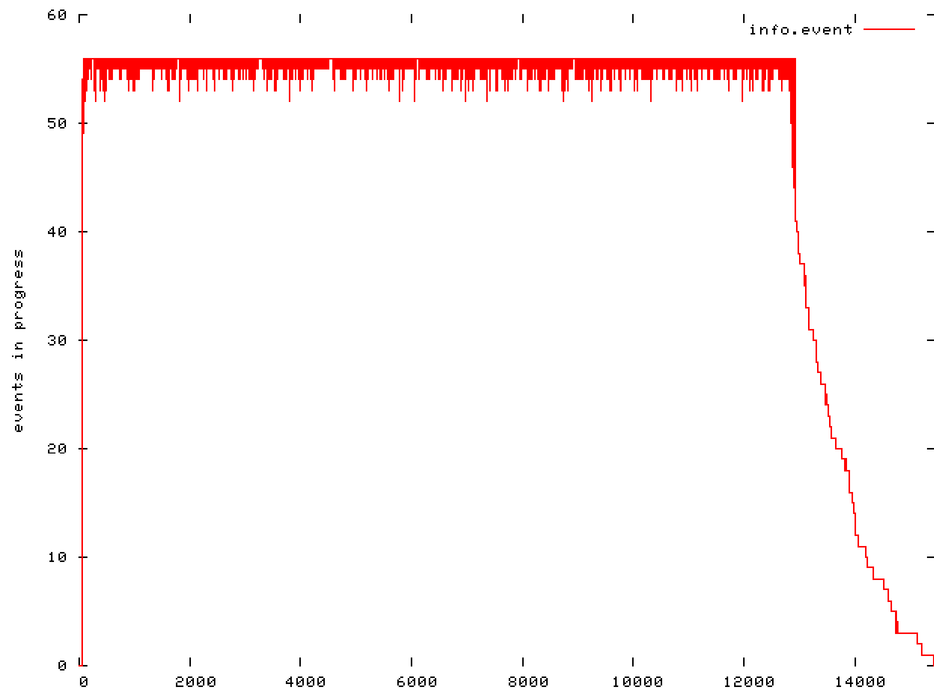
$ swift mandelanim.swift

840 frame input file

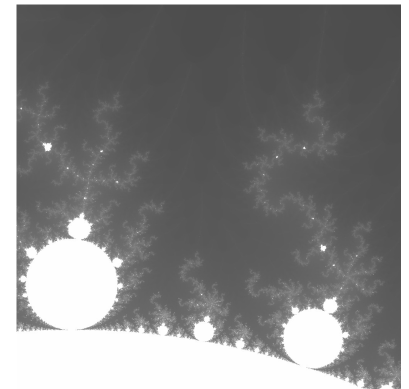


Swift log plotter

Cluster monitoring system - MRTG



this run



http://152.106.18.254/~benc/report-mandelanim-20090404-2324-zza0y5m7

Running Swift on the grid

- Moving to the grid, we get more CPUs, but more problems introduced by the very distributed nature of the system.
- These are usually not Swift specific, but Swift approaches to them give a concrete perspective.

Site selection

- We have access to a lot of sites, with differing characteristics.
  - Some are static (am I allowed to use this site?)
  - Some are dynamic (is this machine working?)
- When Swift wants to run a job, it has to pick a site to run that job on.
- We want to pick “the best” site
  - (Q: what is best?)

Load limiting

- Different sites can bear different loads.
- Important that Swift does not overload a site.
- Amount of load Swift can put on a site depends on load other things are putting on a site
- Hard to detect this value

Site scoring model

- Swift implements site selection and load limiting using a numerical score for each site.
- Number of jobs allowed on a site at once is a function of the score
- When a site is good at an operation, +score
  - successful execution of a job
- When a site is bad at an operation, -score
  - unsuccessful execution or slowness
- Feedback loop hopefully ends up at reasonable score for site, but still varies as site changes.

Site scoring model

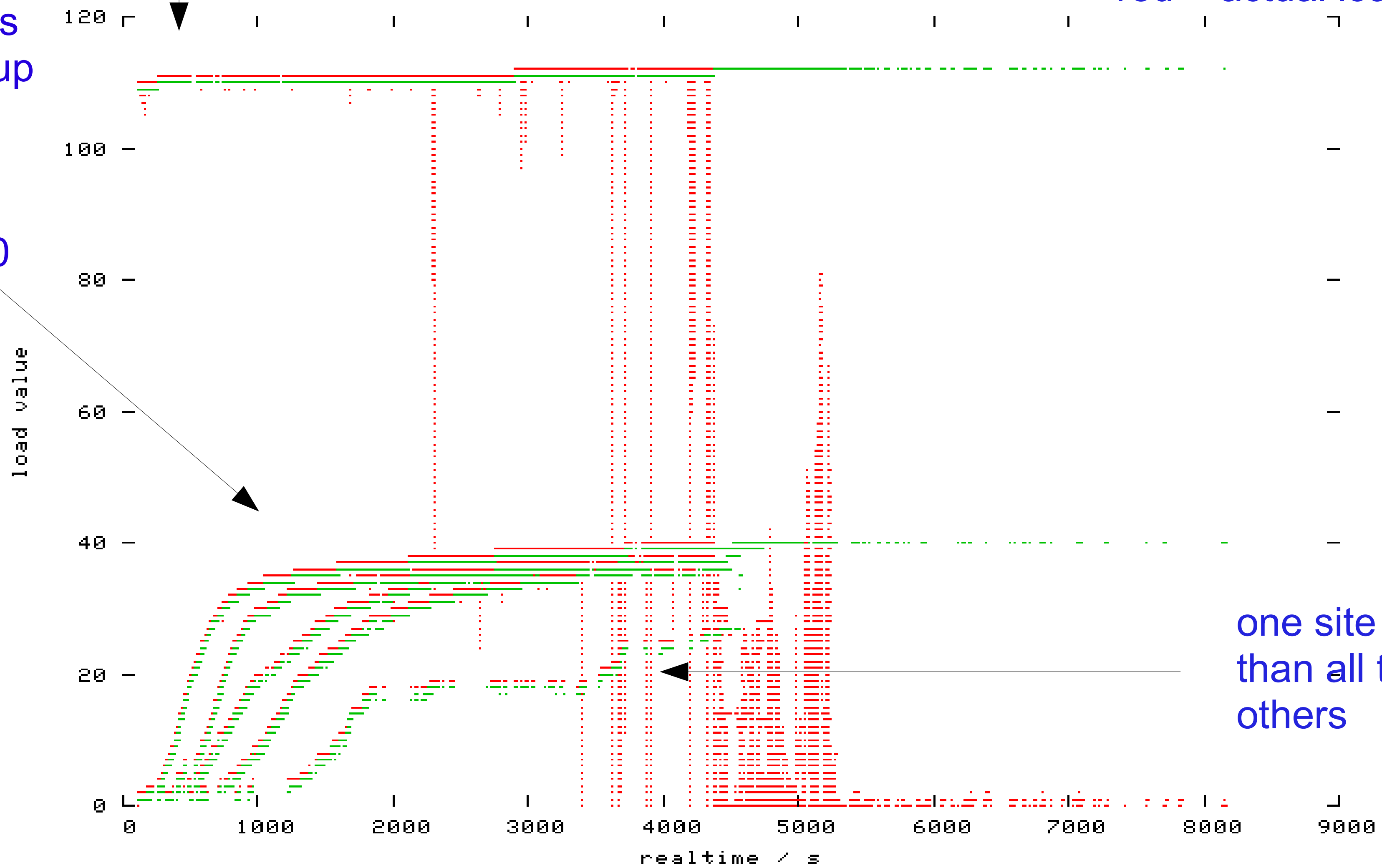
- Benefits:
  - conceptually very simple
  - deals with site changing over time
  - doesn't care about reasons for goodness or badness
- Weaknesses
  - doesn't care about reasons for goodness or badness
  - Perhaps different failures should have different responses
  - Doesn't deal with different site characteristics (CPU speeds)
  - Doesn't deal with different load characteristics (job submission load is qualitatively different from file transfer load on a site)

Site scores from a multisite run

UJ – sites.xml profile keys say to start high

green = permitted load
red = actual load

other sites ramping up from 0 to their configred max of 40 jobs



one site lamer than all the others

graceful degradation of service

- In a PC, usually it either works or it doesn't work
- In more loosely coupled, pieces can fail without the whole system failing
- In cluster, some worker nodes can crash but other worker nodes continue.
- In a grid, similar problems, more so.
- Desirable to have “graceful degradation of service” - as pieces breaks, runs becomes (for example) slower, rather than suffering catastrophic failure.

Reliability: retries, restarts and replication

- Three mechanisms in swift to help with reliability.
- retries – transient failures
- restarts – longer “manual fix” problems
- replication – when we made the wrong decision

retries

- for dealing with transient failures
  - (part of) a site breaks but other sites are available
- retries involve entire new process of site selection, stage in, execution, stageout
- swift tries 3 times by default, and then gives up and the whole run fails

restarts

- no matter how many times retry, some problems won't get fixed automatically
  - machine running Swift crashes
  - single site is taken down for a week's maintenance
  - bug in your application causes it to always fail for certain inputs
- on-disk restart log
  - logs which swift variables already computed
- lazy errors
  - a job fails retries too many times so wf will end

replication

- Some times we submit a job to a site, but we end up with other free sites that could execute the job quicker.
- Sometimes a bit quicker, sometimes much quicker (for example, UC TeraPort once had a queue that was 14 days long)
- After a certain period, submit the same job again. Like retries, but we don't cancel the first job
- Whoever reaches the front of their queue first wins!

application installation

- We need to have component programs on destination sites in order to run them
- So need to install them
- The mandel5 app is simple enough that we can stage it in like a data file.
- Other software is not so easy to install
  - needs configuring on remote site
  - has dependencies on other libraries that need installing
  - in a big VO, often rely on VO admins to install this for you

executable staging

- Map the application executable as a file, and pass it as a parameter.
- wrap everything in a shell script

mandeldeploy.swift

```
type file;
int side = 8;
file tile[][] <simple_mapper;suffix=".pgm">;
file mandel <"mandelbrot.gif">;
file apps[] <fixed_array_mapper;files="remote-mandel mandelwrapper.sh">;

app (file result) render(int x, int y, int side, file apps[]) {
  sh "mandelwrapper.sh" x y side 0.0582 1.99965 200000 1000 1000 32000
stdout=@result;
}

app (file frame) montage(file tiles[][], int side) {
  montage "-tile" @strcat(side,"x",side) "-geometry" "+0+0" @filenames(tiles) @frame;
}

foreach x in [0:(side-1)] {
  foreach y in [0:(side-1)] {
    tile[y][x]=render(x,y, side,apps);
  }
}

mandel=montage(tile, side);
```

Map the application and the wrapper script into a SwiftScript array

take the executables as a parameter to cause them to be staged in with the job

invoke the shell script using unix shell

pass the mapped apps into the render procedure

executable staging

- Other systems can do similar stuff, in less complicated fashion
- Condor: `Transfer_Executable = true`
- GRAM: `-stage` command-line option

clustering and coasters

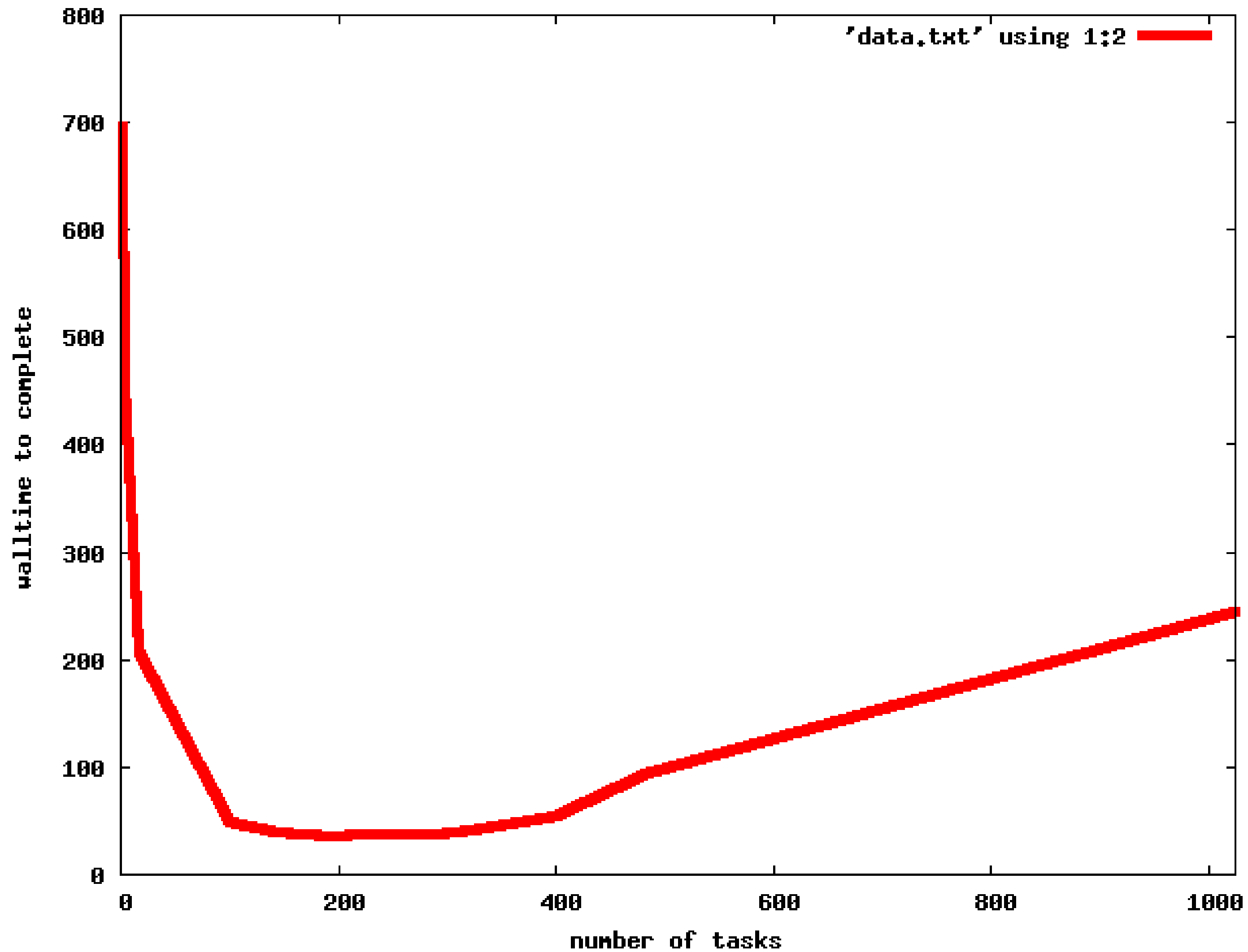
- sometimes relatively expensive to run a job
  - eg if you computation is 60s and you spend 10s submitting that job and managing overhead...
- bundle up multiple application-level (swift-level) jobs into one execution-system job
- Swift has two approaches:
  - clustering
  - coasters

job clusters

- take several jobs, submit them as a clustered job
- clustered job runs each in sequence
- reduces job load... underlying execution system sees only one job instead of all
- ... but reduces parallelism
- In swift, it can be more natural to express SwiftScript as lots of small tasks, and let Swift build job clusters.
- (this is about the 3rd use of the word cluster for different means so far)

job clusters

This is the graph from earlier lecture, when we broke up the mandelbrot app into many tasks



clustering pushes us this way

larger number of smaller tasks pushes us this way

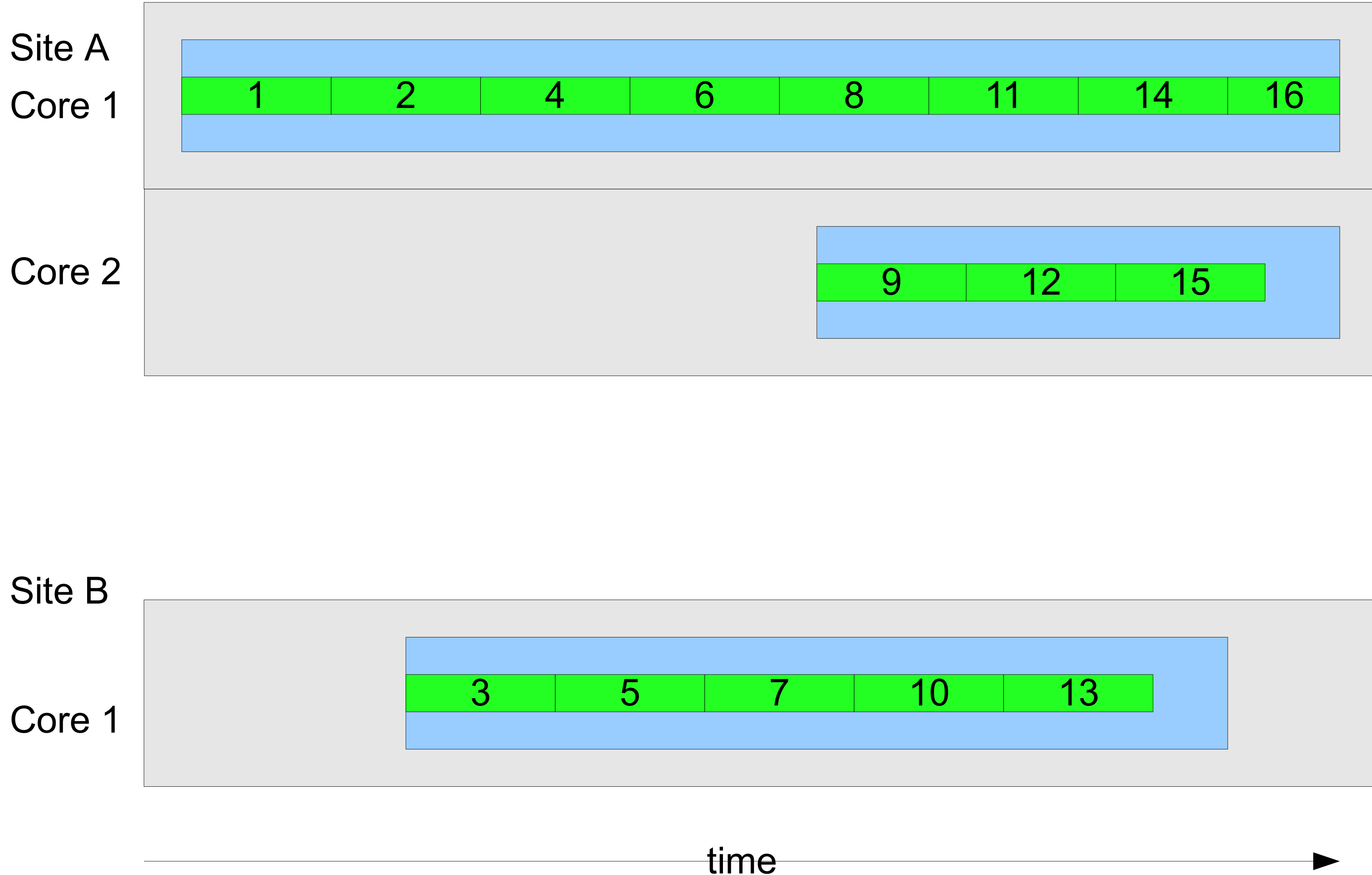
coasters

- in recent years, common model – Condor glideins, PANDA (atlas), falkon
- separate “provisioning” from “execution”
  - provisioning is getting hold of the resources to run your jobs... a bit like reservations
  - execution is running actual jobs
  - sometimes we know we have stuff to run but can't describe it accurately yet (for example, we don't know input parameters yet, or we don't know which order we want to run things)
- much more efficient “tiling” of jobs onto CPUs than using job clusters
- big job submission to own CPUs, then make smaller cheaper submissions into that allocation

coasters

- submit a worker as an execution-system job
- worker then gets jobs from Swift whenever it is free
- workers, not application jobs, go into execution-system queue – can queue workers on several sites and whichever starts running will do the work
  - cf: replication provides a different way of racing jobs to CPUs
- needs active components running on sites so less easy to get running in practice

coasters



clustering vs coasters

- TODO: graph of mandel tiles app using coasters and clusters to show how jobs are tiled onto CPUs. or maybe a hand drawn diagram? the latter is cooler but more complex to generate now... easy to draw 4-tile mandelbrot example – does it look better with coasters? I think so because 1 tile takes longer than the other 3.
-

OSG information system integration

- OSG has information systems which publish details of each grid site.
- Swift interfaces with ReSS, one of those systems
- `swift-osg-ress-site-catalog` command generates a sites.xml site catalog for an entire VO

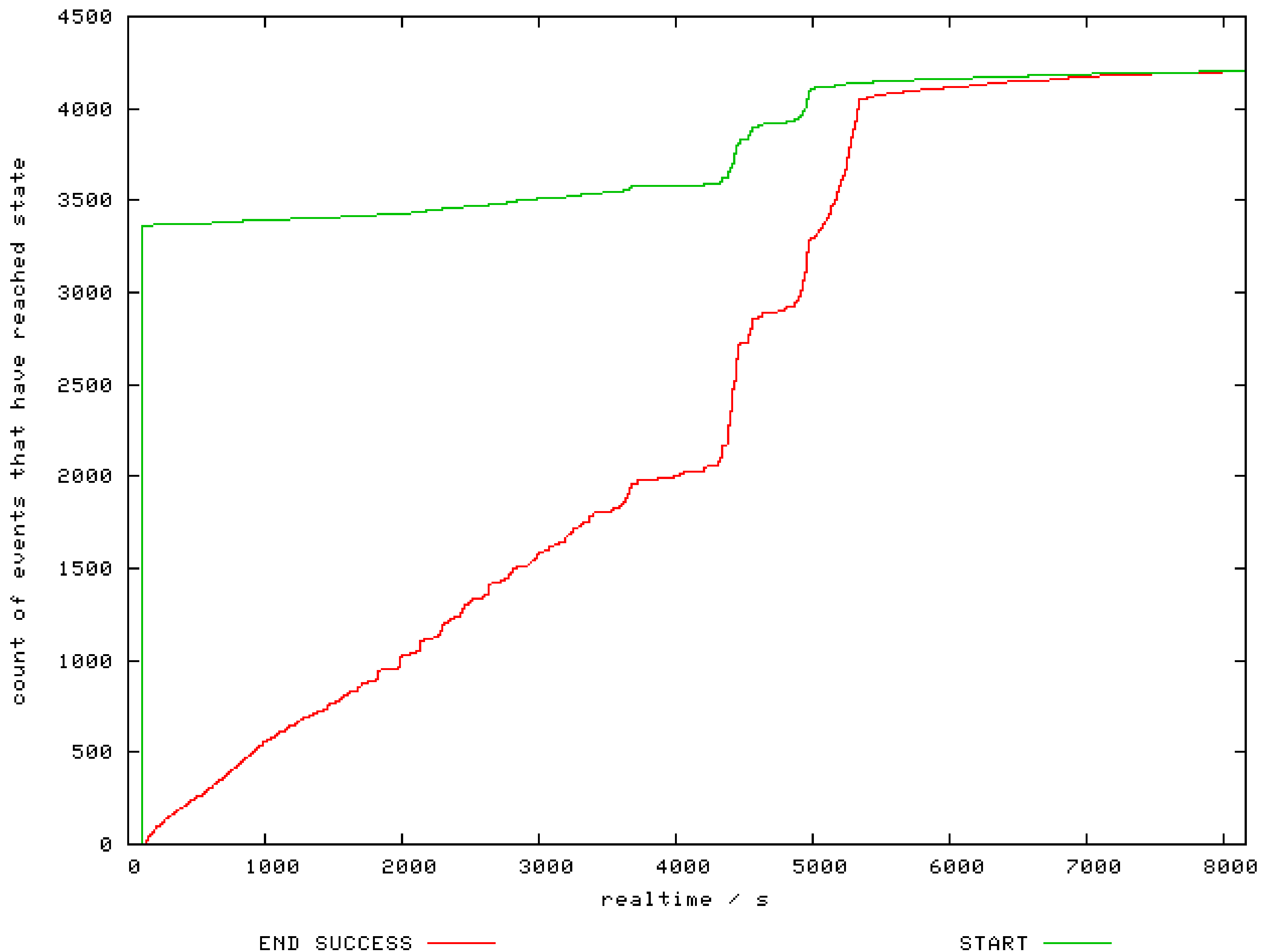
Log analysis

- Swift collects huge quantities of logs by default
- Can use `swift-plot-log` to generate graphs and statistics in webpage form
- Some graphs shown already are from this
- What information can we find from these web pages?
- Some are excuciatingly boring, some useful
- Go through multisite OSG run at http://152.106.18.254/~benc/report-mandelanimdeploy-20090406-1240-3t6hl6c4/

completion of run over time

y-axis is number of swift app procedure calls (green=calls made, red=calls completed)

1 procedure call encompasses all the retries necessary to make it eventually complete.



how many jobs did each site run?
table in execute2.html

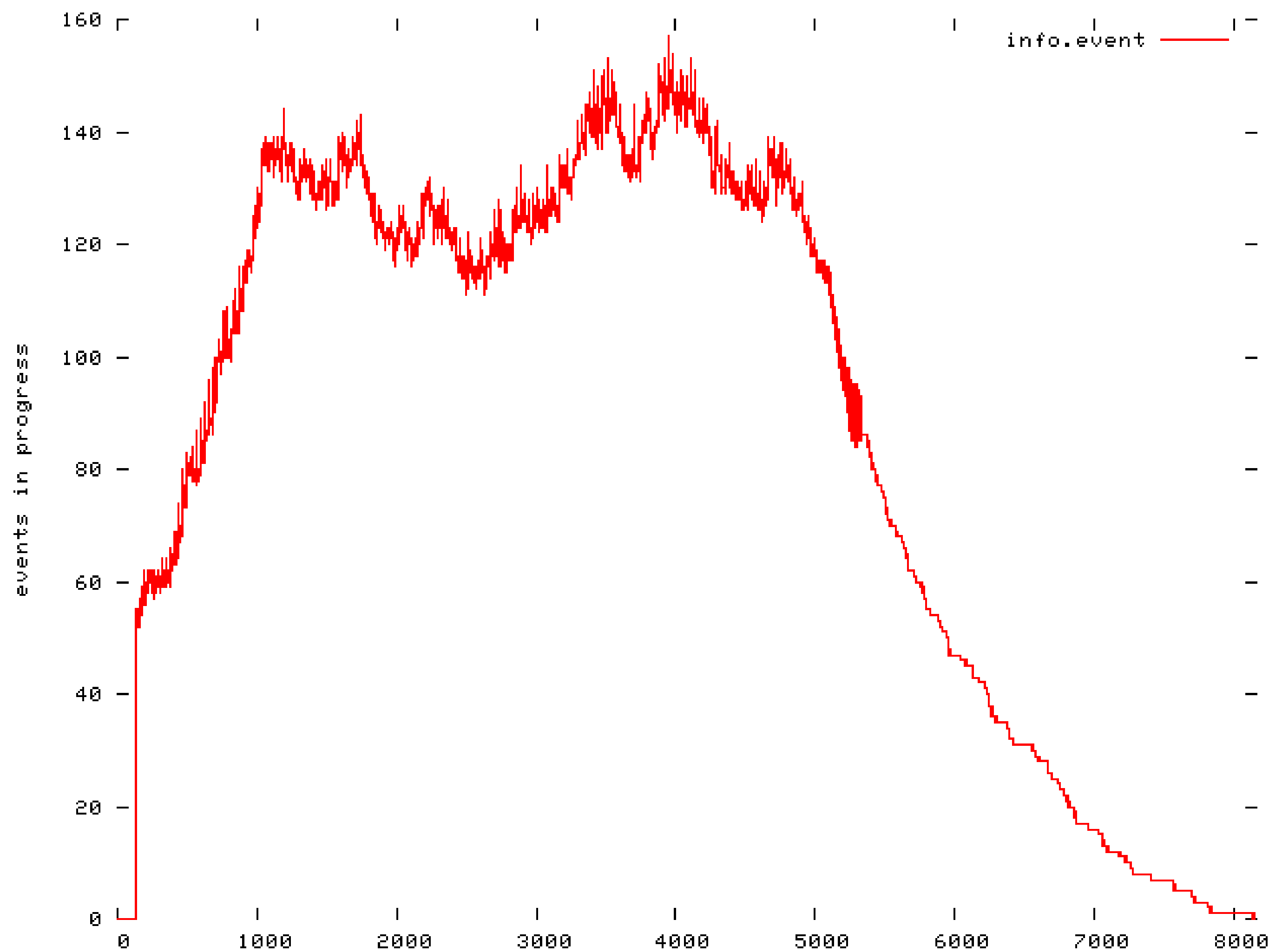
how many retries?
at bottom of execute2

worker node logs

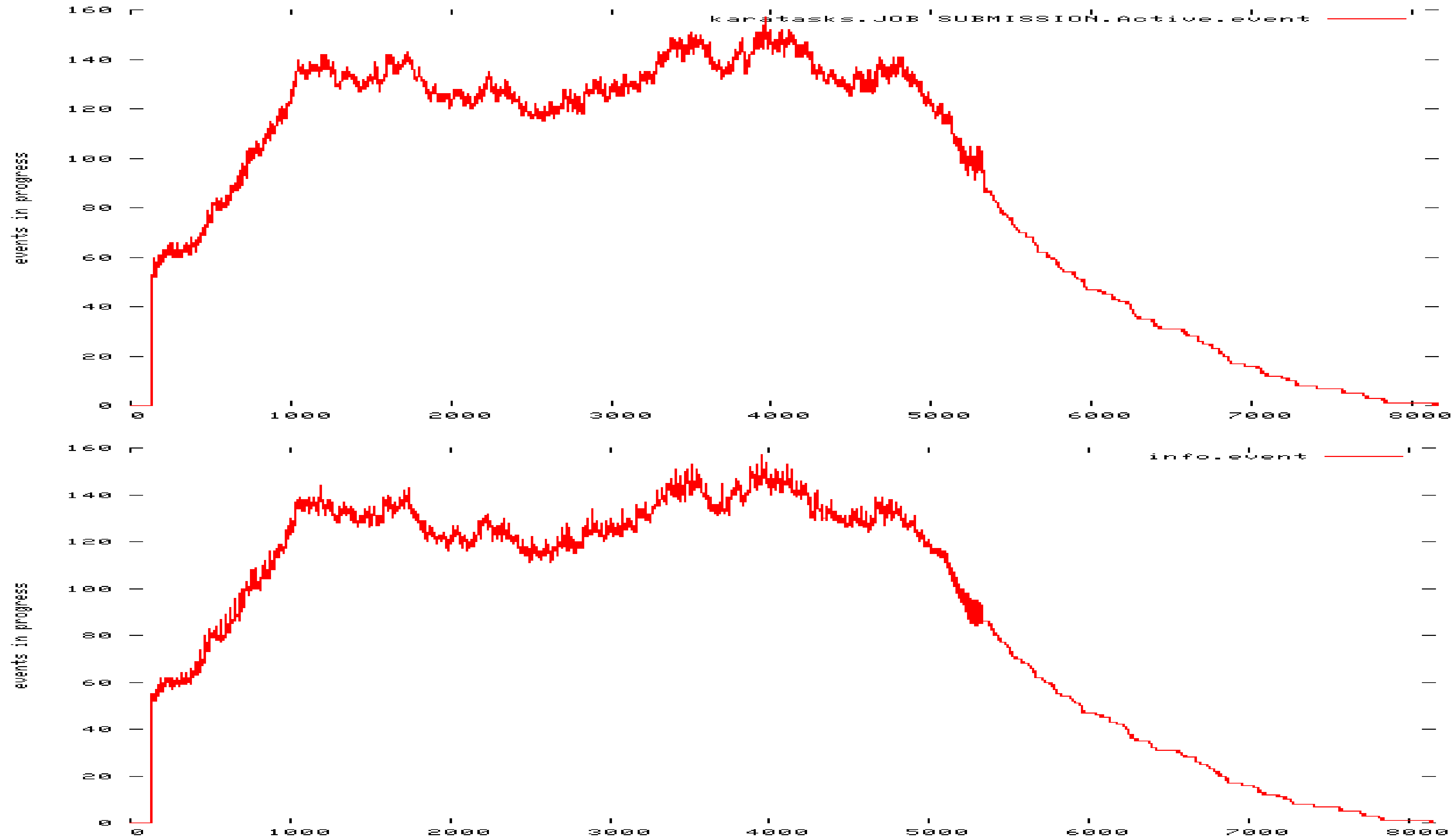
- distinct from Swift client log
- not always available because Swift must (try to) transfer it back after an execution
- most accurate log of what is happening on worker nodes

how many cores in use at once?

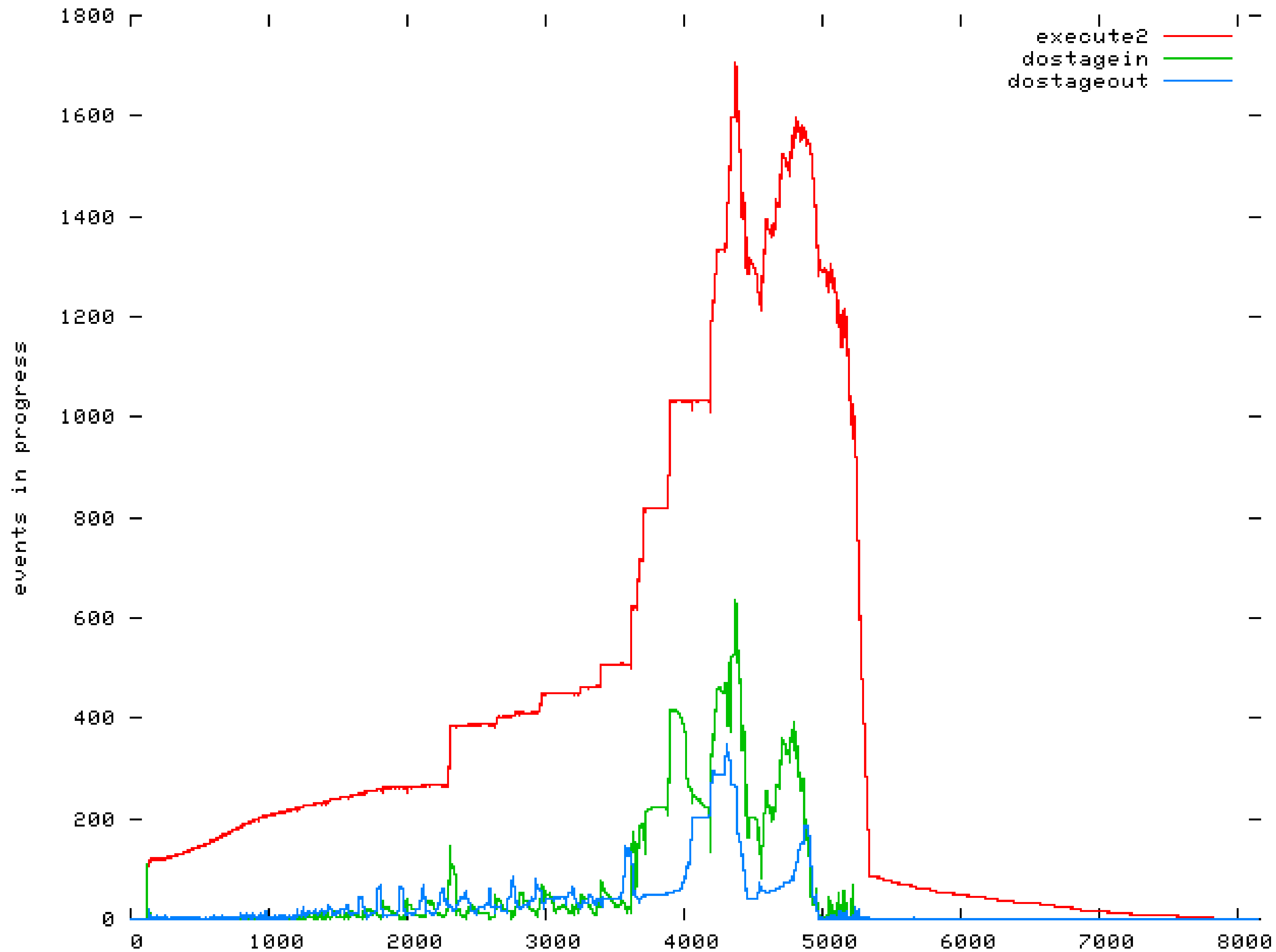
- “total concurrent wrapper+application execution on worker nodes” on info.html



how many cores at once
(two different sources)

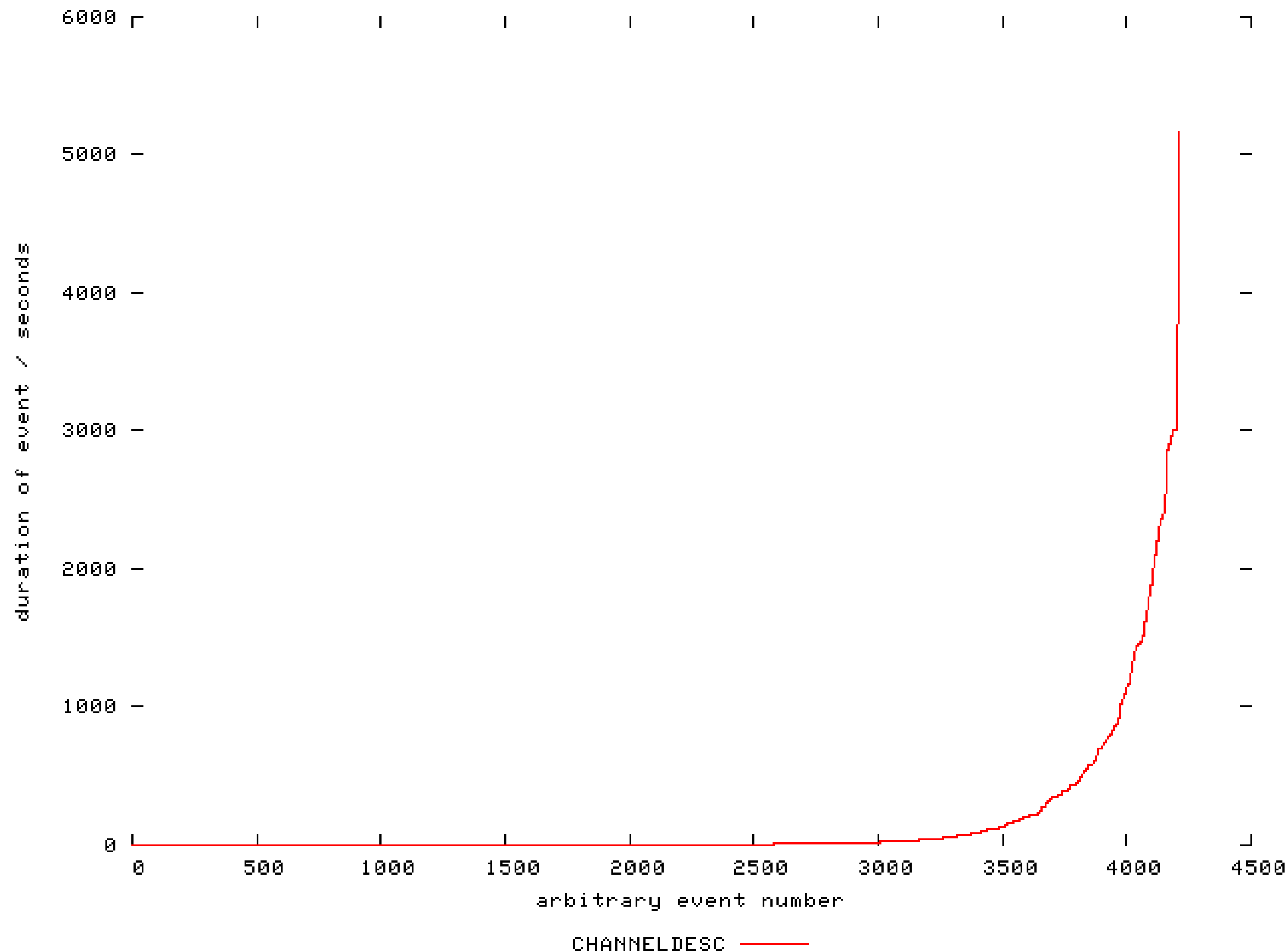


stageins, stageouts, executes



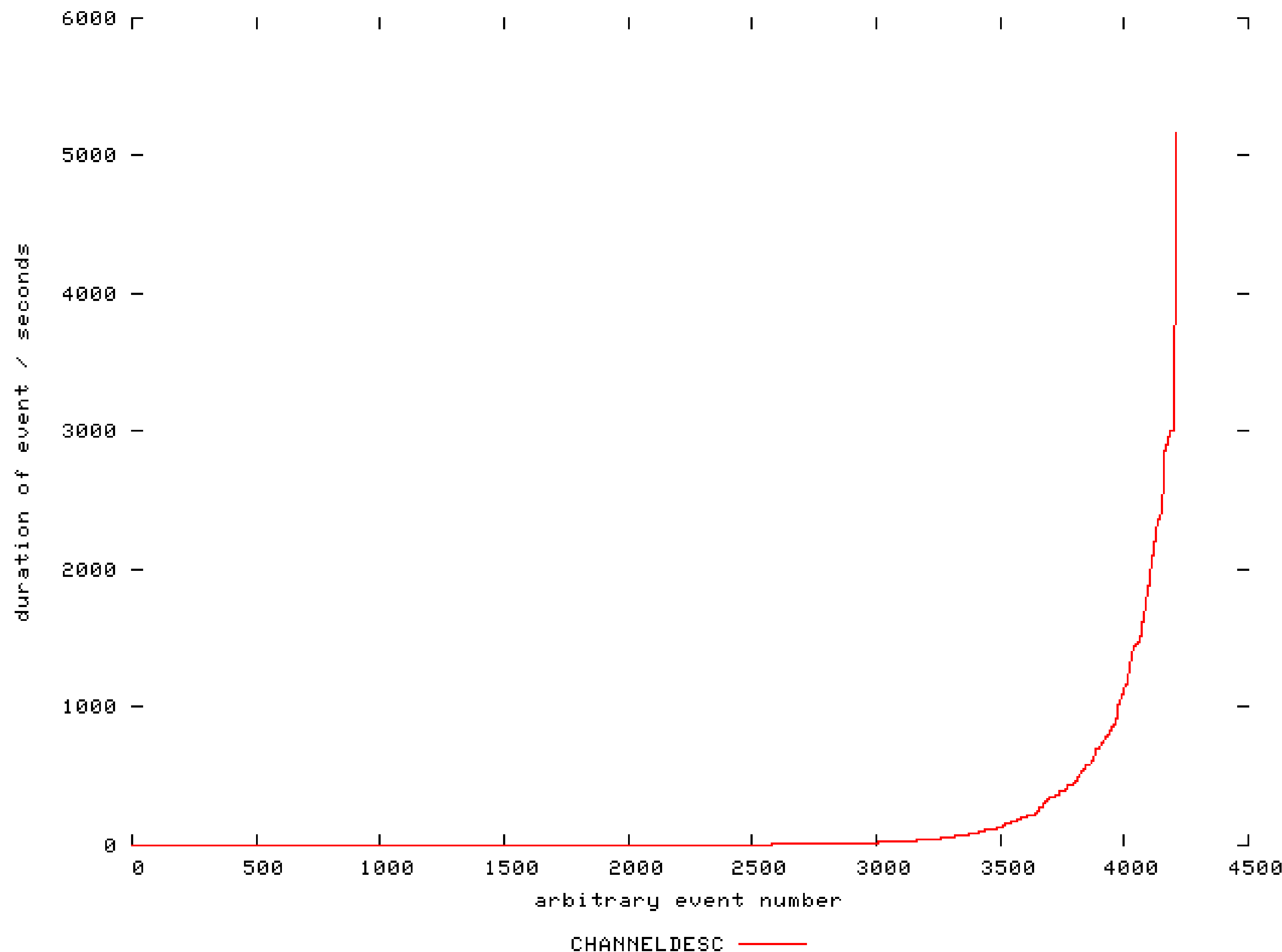
what is the distribution of individual task durations?

- on info.html, info task duration histogram



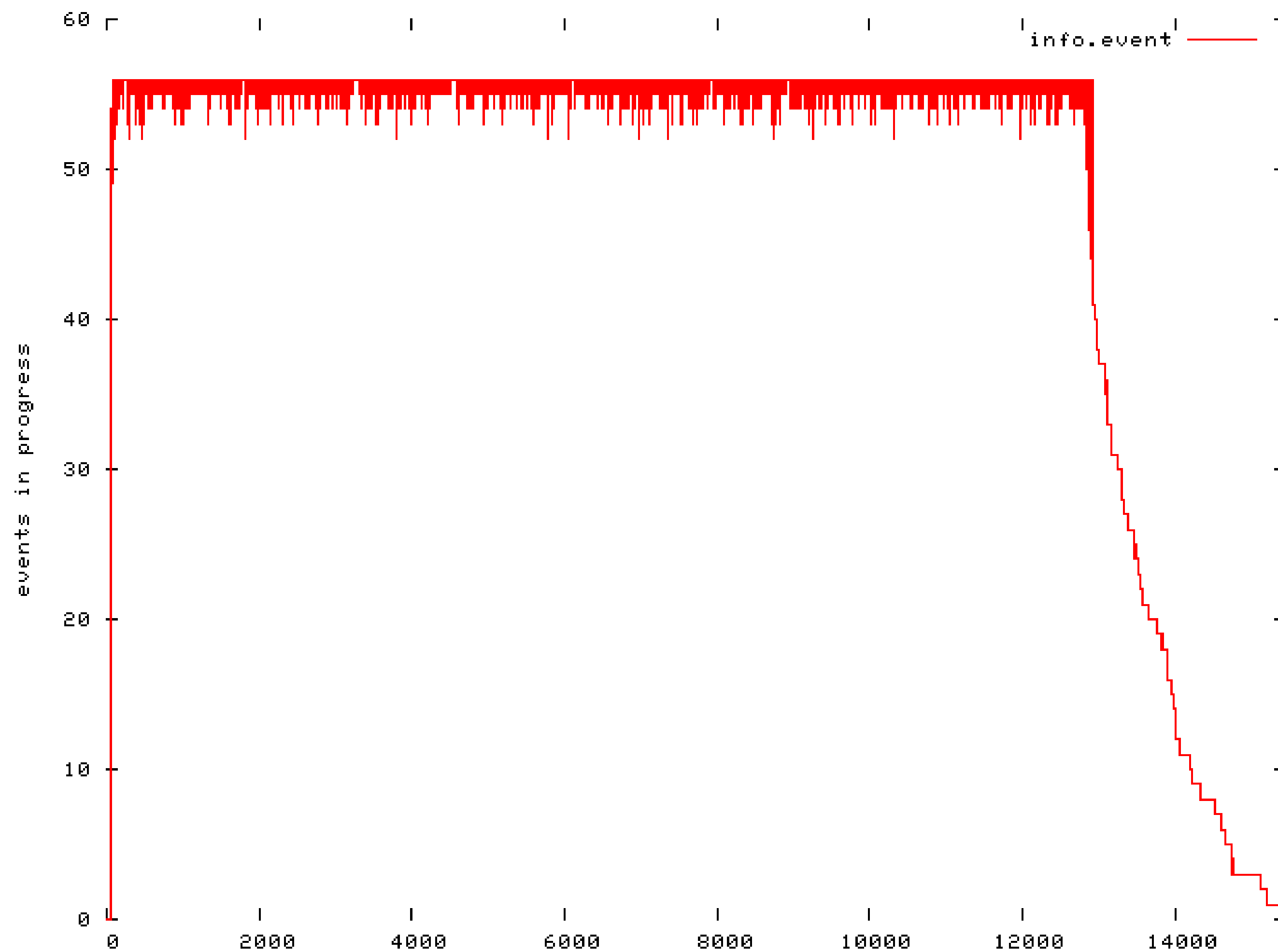
diagnosing walltime problems

- can see here that most jobs took under 1000 seconds. so maybe if we measured a few, we'd expect a maxwalltime of a few thousand to be good.
- but... some jobs take much long – 5000s (almost 2 hours)
- when I was testing I made this mistake. most of run would be OK, but some jobs would always fail, even after 10 retries...



Node-bound runs

- Here is a chart of worker node usage on the cluster – we can use all the nodes.



Could be compute-bound (100% usage) or could be bound by speed of data movement within the cluster (network/NFS speed)

total node time

- can add runtimes for all info logs
- total time spent on worker nodes
- if all the jobs are compute time, this is basically equal to CPU time
- if its much longer than you expect (by running tasks manually, separately) then maybe there is a scalability problem in the infrastructure (eg in shared filesystem)

timeouts

- maxwalltime (profile key, GRAM attribute)
- kinda like replication
- swift does maxwall*2 killing locally
- in mandel this was a problem where jobs took longer than 1h and I had stuff set to die after an hour
- (TODO this needs to move to replication?)

TODO job submission overheads

- plots of what swift sees as job submission overheads vs what the info logs see
- show as necessary overhead for jobs, encourages us to make jobs last longer, so that we incur overhead once per more compute time
- reference coasters and clustering
- mandel graph shows this quite nicely – lots of short jobs dominated by the submission waiting overhead rather than compute

More advanced topics

- provenance
- TODO language features that aren't used in mandelanim
- TODO runtime features that aren't used in mandelanim or previously discussed

Provenance information

- About Swift's prototype provenance db implementation.
- TODO provenance def from
- it is ongoing development, not a product
- some example queries
  - how did I make this data file?
  - which data files were built using data that has been computed on site X?

exercises:

- implement animated mandelbrot in swift
  - you'll need 3d arrays, to add the animation command to tc.data and declare that
  - move tiling into a procedure that outputs a single frame
  - array of frames
  - don't give people the source here – see if they can figure it out for themselves
  - run – on cluster? on osg? get execution logs and plot the results using swift-plot-log. anything interesting there?