Time-critical gravitational wave searches

Craig Robinson
Cardiff University

Overview

Brief discussion of gravitational waves

Current model for distribution of the search – advantages and disadvantages

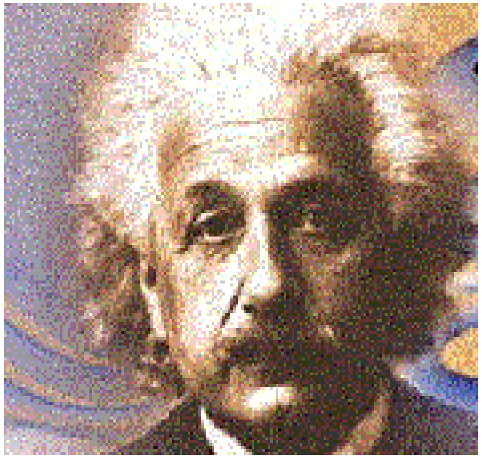
Another model – idea of a low latency search

Algorithm for load balancing the different jobs

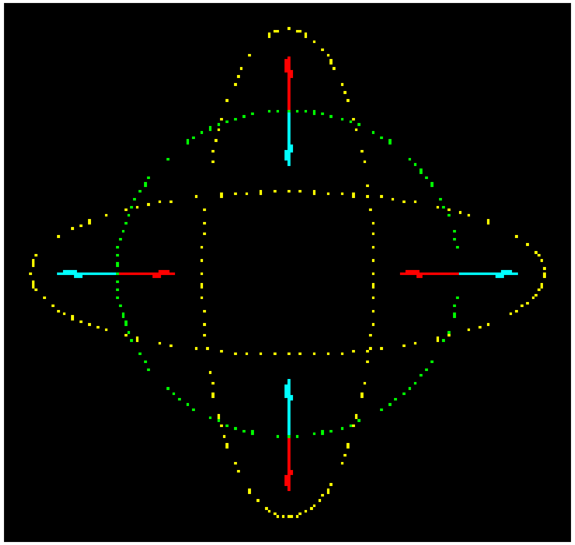
Current status

Future developments

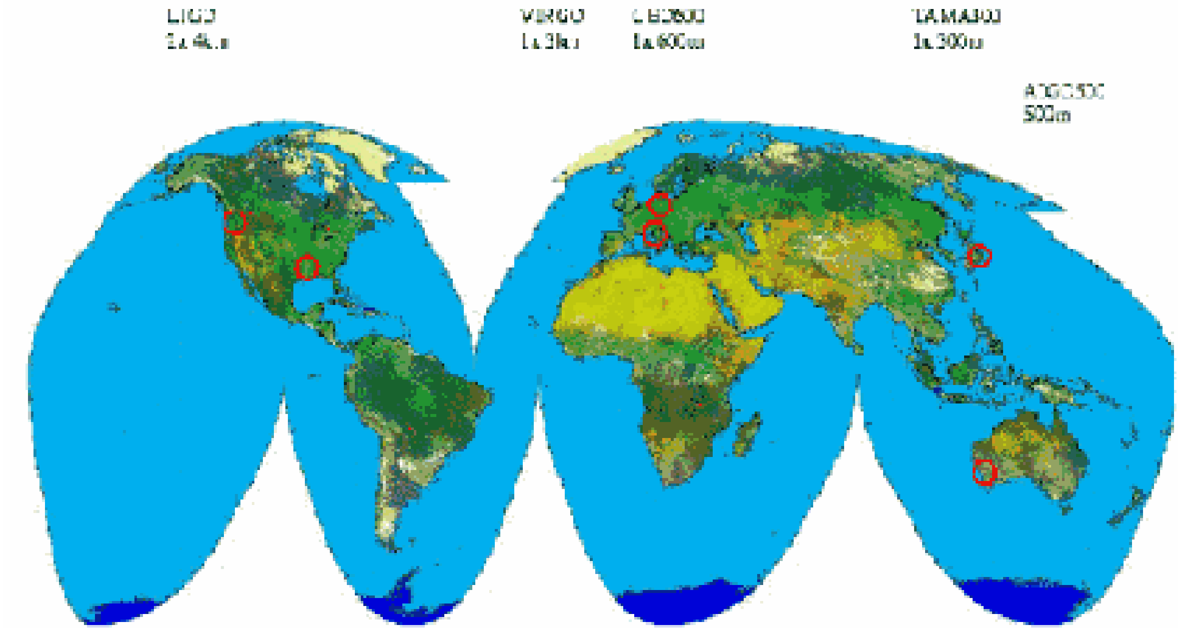
What are gravitational waves?



- Ripples in the curvature of space-time
- Predicted by General Relativity
- Produced by acceleration of massive objects
- Various potential sources
  - *Neutron stars*
  - *Inspiralling binary NS/BH*
  - *Burst sources*
  - *Stochastic background*



Detection of Gravitational Waves



Many interferometric gravitational wave detectors across the world

LIGO detectors (2x4km, 1x2km) in Livingston/Hanford USA

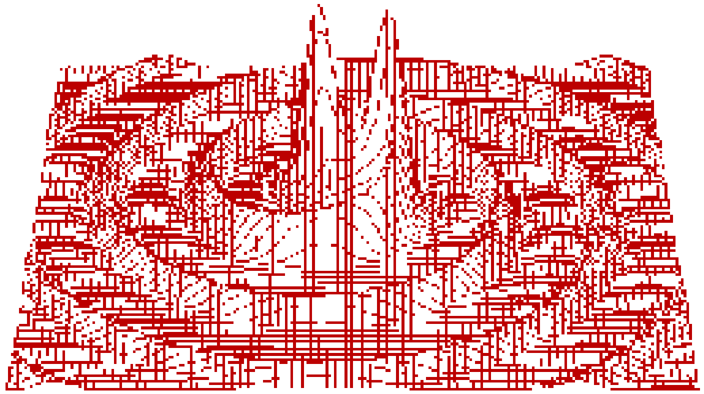
GEO detector (600m) UK/Germany collaboration

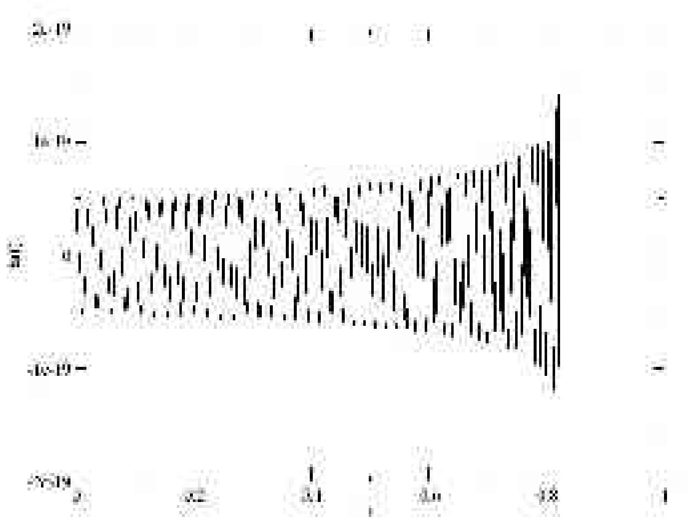
VIRGO detector (3km) France/Italy

TAMA detector (300m) Japan

Detection of Gravitational Waves

- Binary system of compact objects – one of the most promising sources
- Objects orbit each other emitting GW and lose energy
- Eventually coalesce in a final 'plunge'
- Waveform is a characteristic 'chirp'





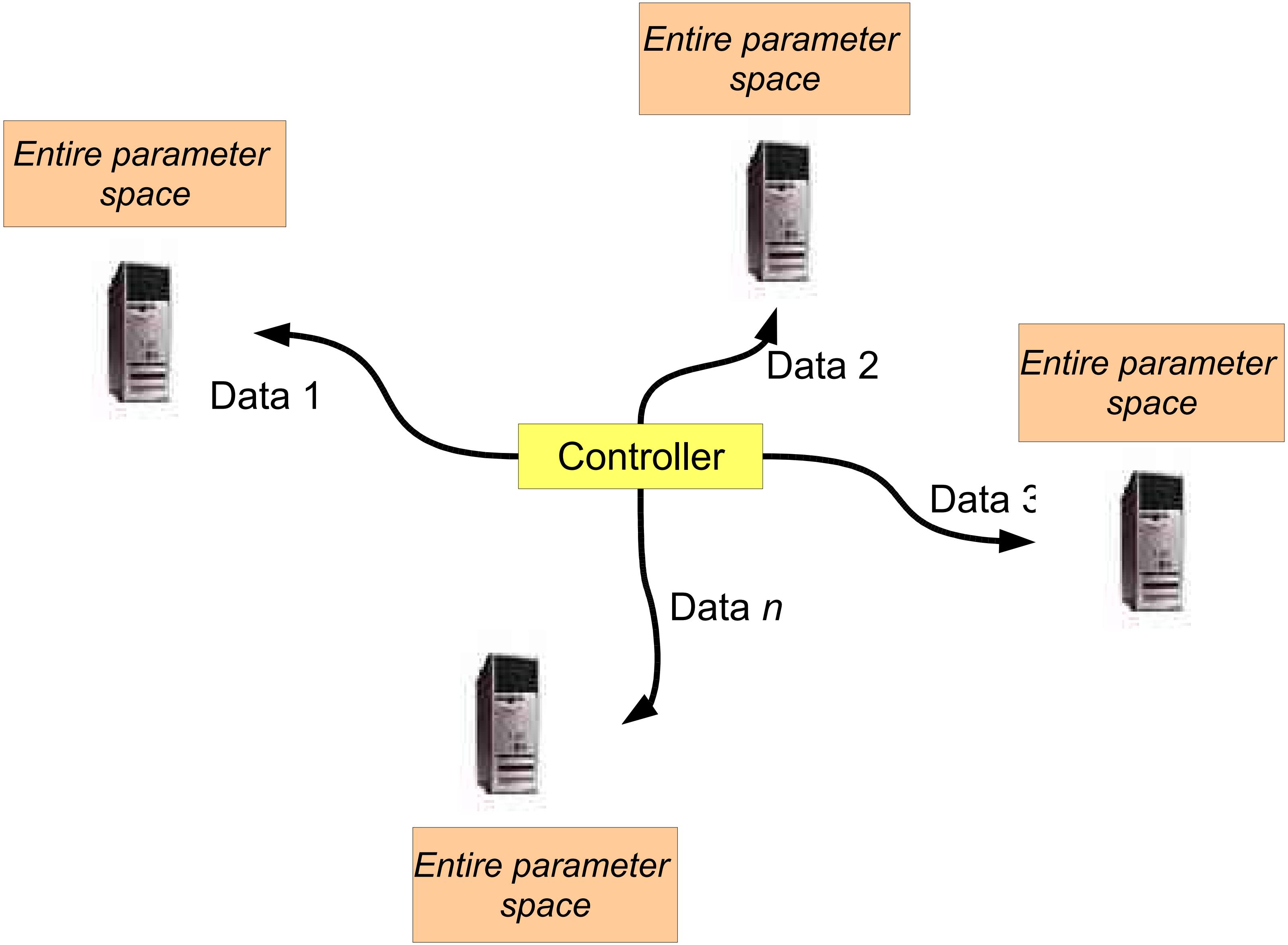
Search we wish to perform

- Search LIGO/GEO data for gravitational wave signals from inspiralling compact binary systems
- Matched filtering – correlation of data with templates defined within search space
- Non-spinning case – 4 search parameters (t_a, phi_a, m_1, m_2)
- Spinning case – 12 search parameters (as above plus spins, orientation of orbit)

Current model for distribution

- Structure mainly in use for distribution (except online search) is a data-parallel model
- Each slave node receives different chunk of data
- Each node searches the entire parameter space for its chunk of data

Illustration of distribution model



Advantages and disadvantages of this distribution model

Advantages

- Simple to achieve – start up multiple identical jobs with different data
- When a job finishes, start another with a different set of data
- Simple but *effective!*

Disadvantages

- Results not received until jobs have processed the *entire* parameter space!
- Thus, first set of results obtained after time taken for 1 node to process chunk of data
- Introduces a large latency in the analysis

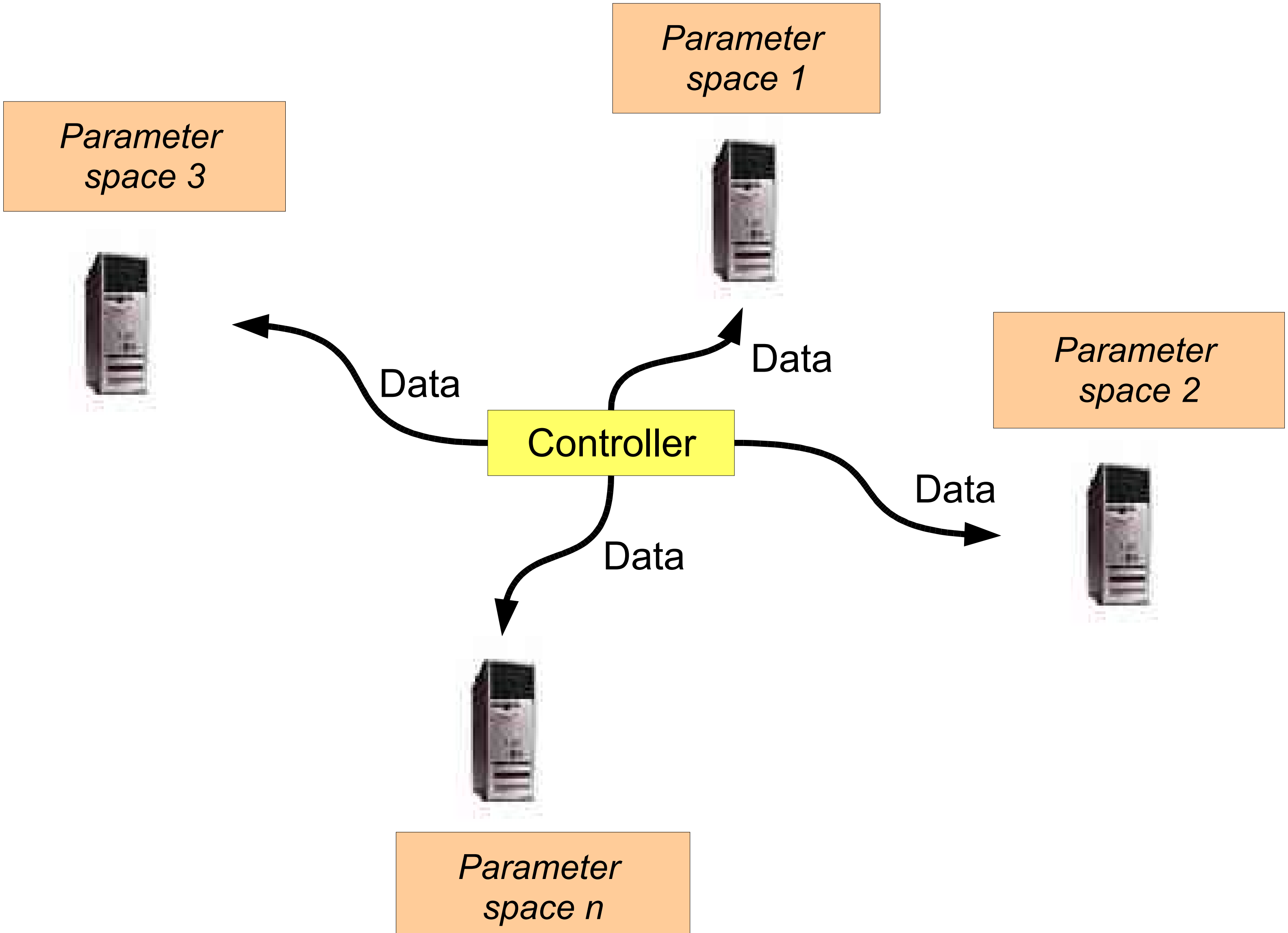
Is latency a problem?

- Not in some cases – e.g. It could get you 100 hours of results in 100 hours
- However, it could take 100 hours to get *any* results!!
- If results are needed quickly, this is not satisfactory

Another model – low latency search

- In this case jobs are distributed in a data-serial, parameter space-parallel manner
- Each node searches the same chunk of data, but a different area of the parameter space
- In this way results can be obtained in real time (provided enough nodes...)

Illustration of distribution model



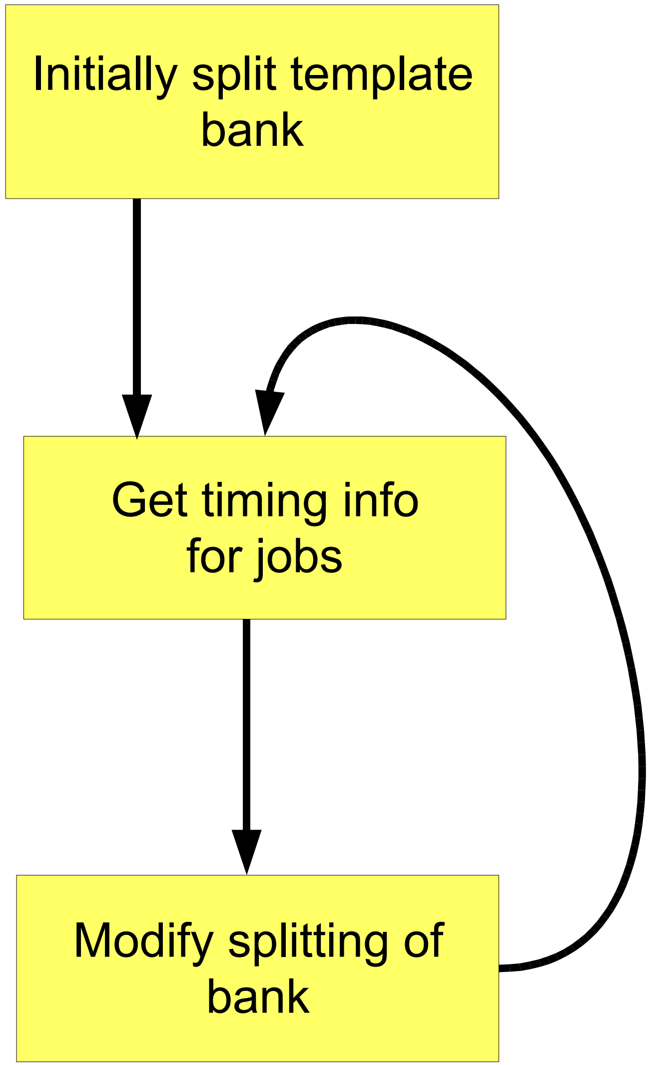
Low-latency distribution

- Each node has same data, but different areas of the parameter space – how do we split the parameter space?
- For certain searches, some areas of parameter space 'more equal' than others
- For heterogeneous resources, different nodes may perform differently
- Need a means of balancing the splitting such that each node takes about the same time to process the data

Low latency distribution

- Step-wise load-balancing algorithm
- A simple model: -
  - *Initially split the parameter space naively (i.e. each node gets same number of templates)*
  - *Use the timing information of this data chunk to determine the splitting for the next*
  - *The splitting of subsequent runs will be determined by timings obtained from previous runs*

Illustration of model



Mechanism for balancing the load

- Get timing information for the previous run for the nodes T_n
- Work out the average time per template for the node, $t_n = T_n / n_n$, where n_n is the no . of templates for node n for previous run.
- For next run, if N_{tot} is total no. of templates for distribution,

$$N_1 = \frac{N_{tot}}{1 + \frac{t_1}{t_2} + \ldots + \frac{t_1}{t_n}} \qquad N_n = n_1 \frac{t_1}{t_n}$$

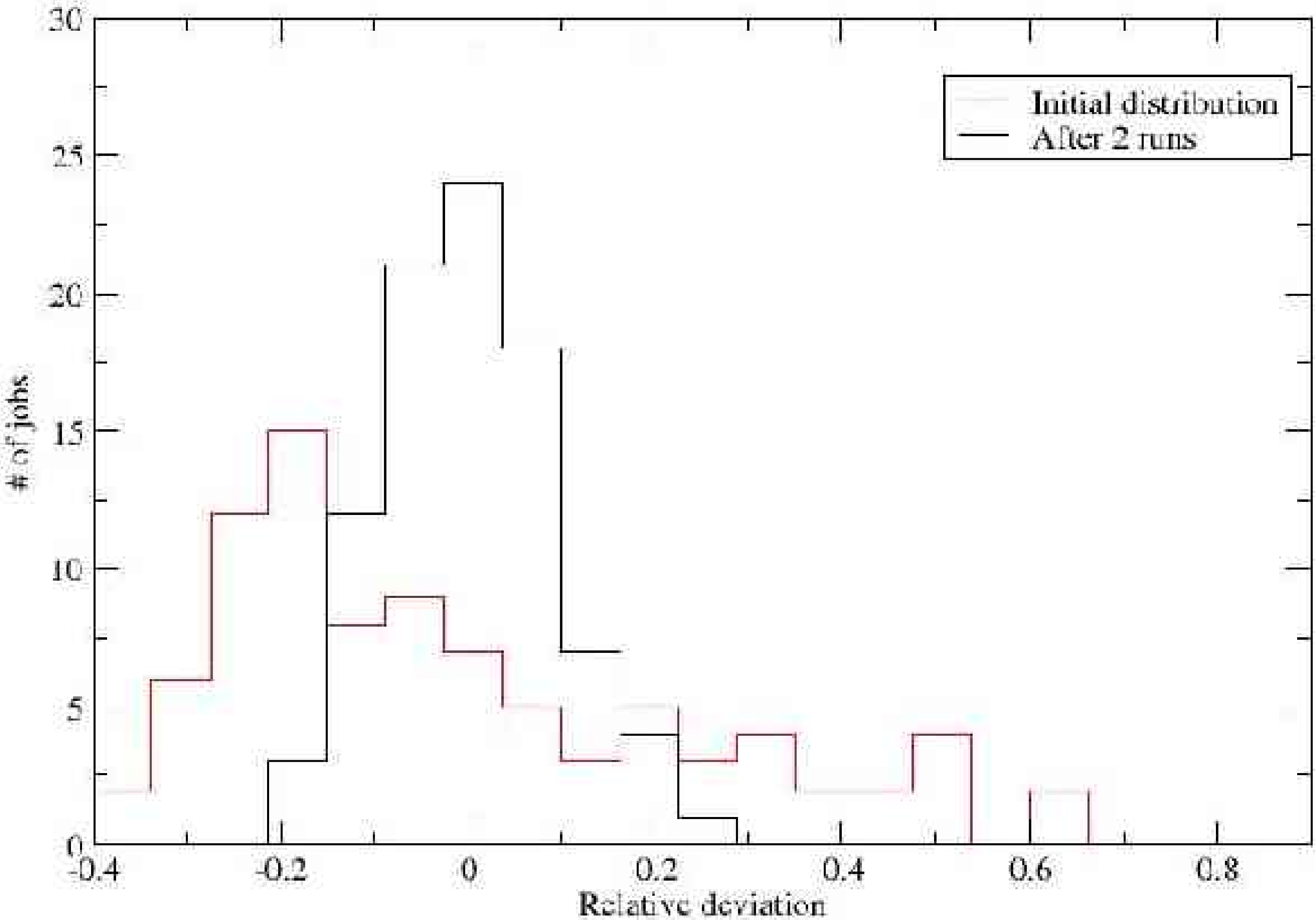
Status of implementation

- Implemented as a set of Python classes/ scripts
- Runs under Condor
- Requires no modification of LAL inspiral search codes
- In event of job failure/delay, 'march ahead regardless'

Performance test

- Inspiral search run on S3 playground data for L1 using PadeT1 templates
- Parameter space 3-20M_{sun} – around 300 templates
- Run on 30 nodes on (temperamental) explorer cluster at Cardiff

Performance test



Future development

- Improvement of the march-ahead step-wise algorithm
  - Instead of using timing info for previous run, use Gaussian weighted average of many
  - Re-implement in a more robust manner
- Development of a new dynamically load-balanced algorithm
  - Slave nodes request templates off controller when idle. When templates are all used, supply the new data
  - May require modification of inspiral code
  - How to implement?? MPI??