HF task distribution inefficiency

Due to the way in which hf_sim distributes the stations to be calculated for the HF part of the workflow it is highly likely some processes will finish faster than others.

Due to the need for synchronization between processes the ones that finish early will idle while the others to finish before they are released.

If the total idle time is found to be high, then it would be useful to investigate alternative methods of dividing the work between processes.

One of the primary challenges for work allocation is that each station takes a different amount of time to calculate, based on its distance from the source, with alphabetically adjacent stations having a similar distance and the current method of job allocation gives each process a block of alphabetically adjacent stations, meaning each process is likely to get assigned a group of long or short running calculations.

Ideally each process would receive a similar workload, with a mixture of long and short running calculations.

Measuring process completion times

For the HF tasks run in the resumption of CyberShake v19p5 logging messages were added to each process just before they called the comm.barrier() function that would cause them to idle.

These log messages record the time the message was logged, the rank of the process creating the log and the number of stations the process computed.

Another log message was created by the master process after the comm.barrier was released.

This final message gives us the baseline for when all processes are completed, and the time between when each process makes its completion log, and this master process log gives the time the process was idling for.

In some cases this does not give a 0 second duration for the last process, however the time difference is in the order of milliseconds, and therefore negligible.

Data collection

A script was created to read all passed logs and gather the idle time of each process for each process, creating an entry in a numpy table with the aggregated results.

In cases where the final message was not found the task was ignored as the required messages were only present for HF simulations run during the v19p5 rerun.

As HF is hyper processed these idle core hours must be halved to get the number of physical/billable core hours used.

A number of simulations were restarted, resulting in them having incomplete data. Where a realisation had a duration longer than twice the average for the fault it was discarded.

Results

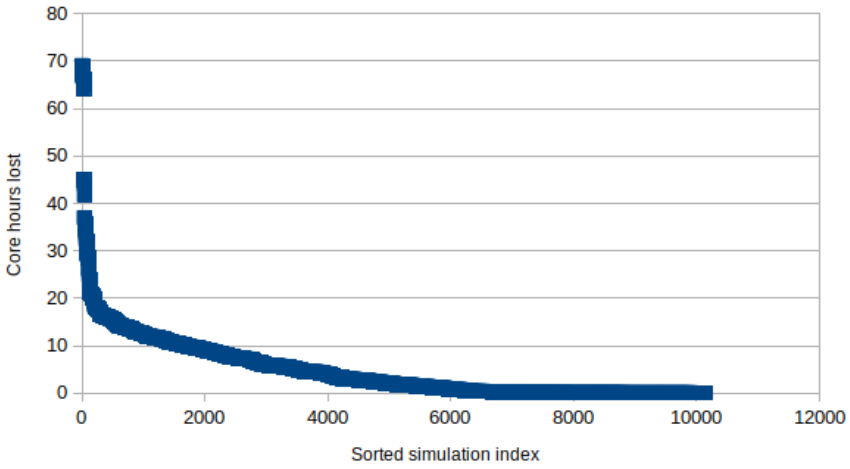
A total of 10133 realisations had complete data from a sample of 11317 realisations.

These realisations had a total of 46596.6 idle core hours to processes waiting to complete.

This gave a mean of 4.6 idle core hours per realisation, and a median value of 1.99 idle core hours.

Using this mean we can approximate that the entire run would have had 52041.2 idle core hours.

The HF tasks of cybershake v19p5 used a total of 389024 core hours, meaning that approximately 11.98% of core hours were idle.



The majority of realisations (8443) had less than 10 idle core hours, while a few (184) had more than 20.

The most idl core hourse on a single realisation was 69.1 (WairarapNich_REL13) and the least idle was 10 core seconds (Ran-02_REL05).

Case studies

Three realisations were chosen to investigate manually.

A plot of the station finish times is given, with idle time on the y axis, and station finish number on the x axis, such that the points form a decreasing line.

It should be noted that the first data point given represents the recorded start time of the simulation, given as a point of reference to indicate how much time is spent idle.

Hollyford_REL32

The Hollyford realisation was the first to be looked at.

The HF task had a run time of 34m 23s on 80 physical cores.

The first process finished after 21m 27s.

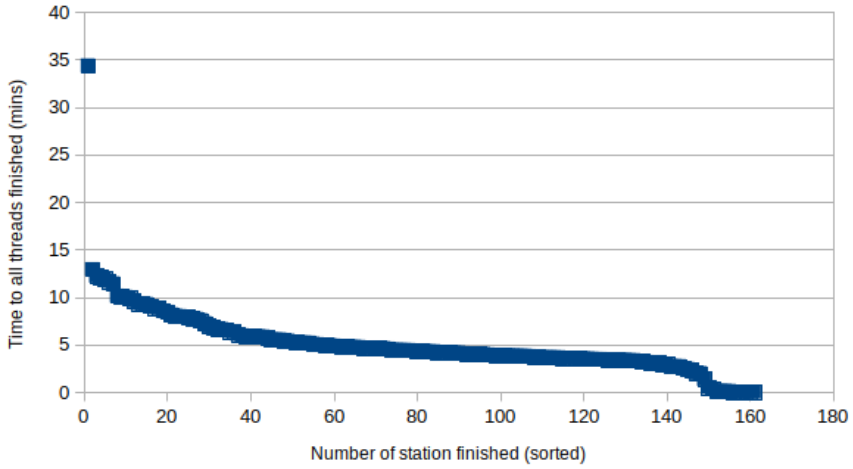
The task used a total of 45.84 core hours.

A total of 6.47 core hours were spent idling.

This means 14.1% of core hours were spent idling.

The plot of finish times indicates that a few processes finished slightly earlier than the bulk of them, and that the final group of 10 or so finished 3 minutes after the remainder.

For each minute this group of 10 ran for, 1.25 core hours were spent idling on the remaining 150 processes



Gimmerburn_REL24

The Gimmerburn realisation was chosen to be looked at for the relatively high number of core hours spent idling.

The HF task had a run time of 20m 31s on 80 physical cores.

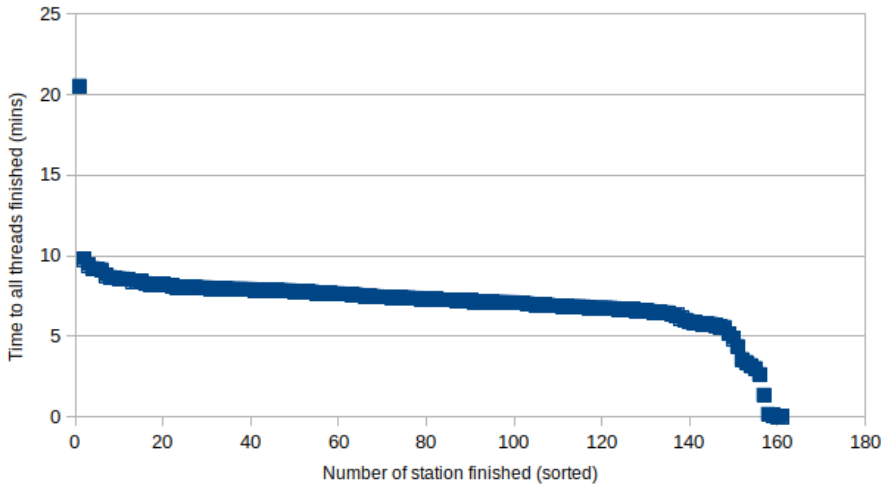
The first process finished after 10m 43s.

The task used a total of 27.36 core hours.

A total of 9.32 core hours were spent idling.

This means 34.1% of core hours were spent idling.

The plot of finish times indicates that a few processes finished slightly earlier than the bulk of them, and that the final group of 10 or so finished spread out over the last 5 minutes.
For each minute this group of 10 ran for, 1.25 core hours were spent idling on the remaining 150 processes.



WairarapNich_REL13

The WairarapNich realisation was chosen for its relatively high run time.

The HF task had a run time of 2h 5m 22s on 320 physical cores.

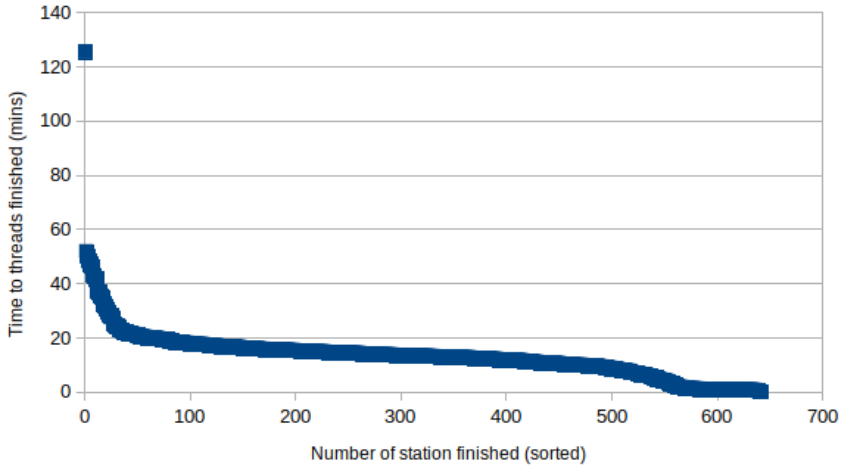
The first process finished after 1h 13m 33s.

The task used a total of 668.62 core hours.

A total of 69.1 core hours were spent idling.

This means 10.3% of core hours were spent idling.

The plot of finish times indicates that a few processes finished slightly earlier than the bulk of them, but after the first 20 or so, there was a steady drop off.



Proposed solutions

1) Leave it. (Give each process a continuous block of #Stations/#processes stations to calculate)

Pros:

- Known approximate amount of idle CH
- No dev time required

Cons:

- Could be higher loss than other station distribution methods
- Stations ordered by name, adjacently named stations likely to be physically adjacent, meaning processes are likely to be allocated a group of short or long running calculations

2) Give each process a non-continuous block of stations (Calculate if Station# % #processes == process#)

Pros:

- Simple implementation, relative to other solutions
- Taking stations not adjacently named means each process is likely to be assigned a range of short and long running stations
- As method of allocation is effectively the transpose of the original allocation method each process is likely to get a similar distribution of station computational complexity

Cons:

- Some dev time required

3) Give each process #Stations/#processes randomly chosen stations

Pros:

- Generally will give performance similar to option 2

Cons:

- Worst case scenario will give all long stations to one process increasing inefficiency

4) Use a queue or other data structure to allow processes to take another station on demand

Pros:

- Each process only takes on more work when its current queue is empty

Cons:

- Increased development complexity
- Overhead of using queue may out weight speedups gained