Serial farming from A to Z

Sergey Mashchenko
(SHARCNET / Compute Ontario / Compute Canada)

Outline

- Introduction
  - What is serial farming?
  - Examples of serial farming
- Simple solutions
  - Command line
  - BASH scripting
  - Array jobs
- Advanced serial farming
  - Small number of cases
  - Large number of cases
  - Using the scripts
- Word of caution

Introduction

What is serial farming?

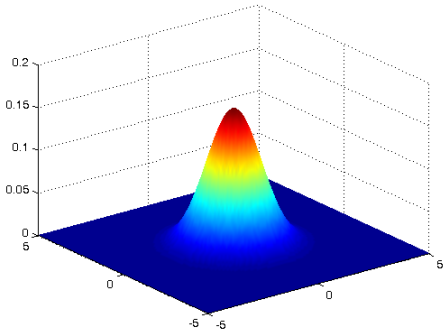
- Serial farming is running multiple instances of a serial code (serial jobs on a cluster), when the order of execution is not important (no data dependencies between the jobs).
- It can be trivially extended to cover batches of independent parallel jobs, which one might call “MPI farming”, “OpenMP farming”, “GPU farming” etc.

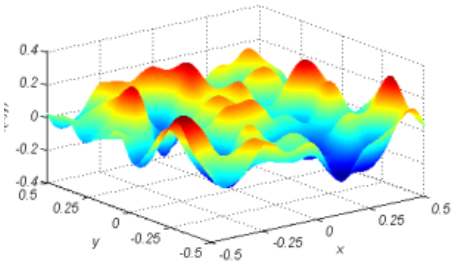
Examples of serial farming

- One common situation when serial/parallel farming is needed is when the code output depends on a few poorly constrained parameters, and the task is to either find a global solution(s), or the global maximum or minimum.
- When the number of unknown parameters is small (say <5) one can attempt a brute force approach: running separate jobs corresponding to different points on a grid in the multi-dimensional parameter space.

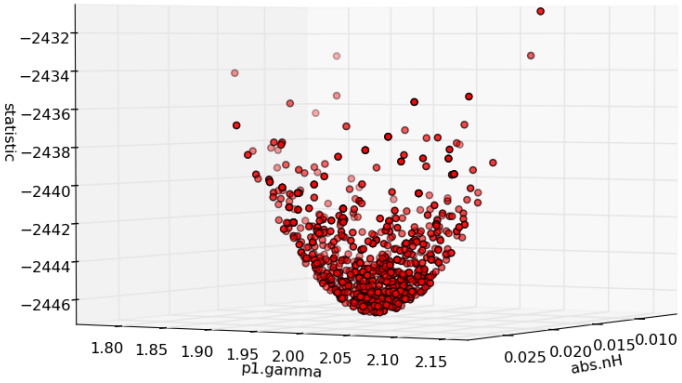
Examples of serial farming

- How many grid points in each dimension depends on the expected properties of the solution: for a uni-modal situation one can use as few as three grid points per dimension.
- In a more general (multi-modal) case number of grid points will be determined by practical considerations (maximum number of cpu hours one can spare for the project).





Examples of serial farming

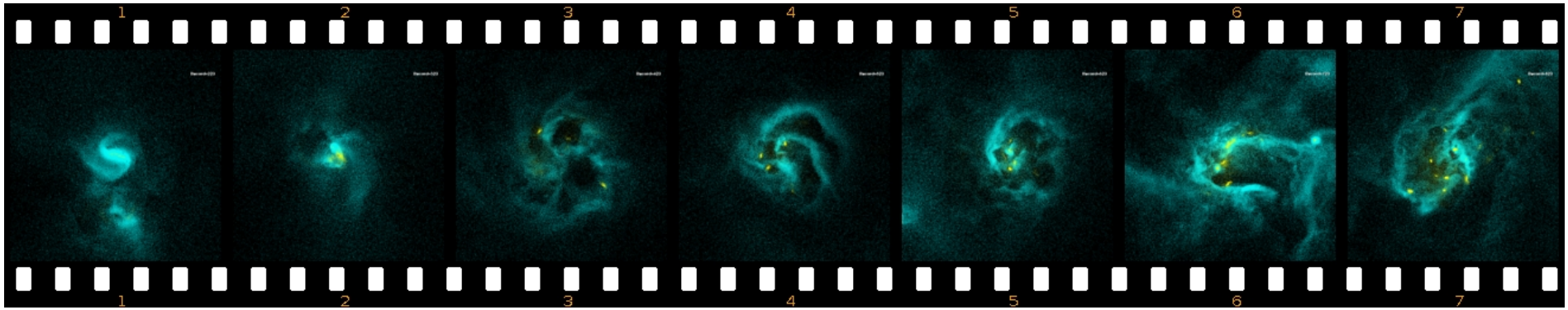


- In a larger dimensionality case (say >4) one can resort to Monte Carlo (random guessing) way of sampling the parameter space.

- In either regular grid or Monte Carlo approaches, one can run just one batch of jobs, or run one, analyze the results, and then run the next batch, where one is zooming in onto the region(s) of interest in the parameter space.

Examples of serial farming

- Another typical situation when one needs serial farming is when one has to post-process multiple “snapshots” (from a time evolution) produced by a large parallel simulation code – to get some global quantitities, or perhaps to produce a movie. These tasks do not have data dependencies, so are perfect for serial/parallel farming.



Simple solutions

Command line

- Using SHARCNET's job manipulation scripts (sqsub, sqjobs, sqkill) plus BASH commands (loops etc.).
- Required sqsub arguments (serial jobs):
  sqsub -r 1d -o out.log ./code
- We will need one optional argument, “--idfile”, (for the full options' list, execute “sqsub -h”):
  sqsub --idfile=idfile

Command line

- Here “idfile” is the name of the file which will store the jobid.
- To submit a batch of serial jobs from the command line, using BASH' “for” loop command, one can do something like this:

```
$$  for ((i=0; i<25; i++)); do sqsub -r 1d \
      -o out.${i}  ./code  ic.${i}; done
```

Command line

- Here it is assumed that 25 different initial condition files, with names ic.0 ... ic.24, are provided, and that the code is executed as "./code ic.xxx".
- It is very useful to use the "--idfile" option here; this will enable subsequent meta-job operations:

```
$$  \rm idfile; for ((i=0; i<25; i++)); do sqsub \
-r 1d -o out.${j} --idfile=id  ./code  ic.${i}; \
cat id >> idfile; done
```

Command line

- Now we have file “jobid” containing all jobid's of the jobs in the batch.
- This file can be used to do query for the batch status:
  sqjobs `cat jobid`
- And we can kill all the jobs in the batch with a single command:
  sqkill `cat jobid`

Command line

- If we want to launch one job per each initial conditions file in the current directory (say, with names ic.*), we can use a different flavor of the "for" loop command:

```
$$  for name in ic.*; do sqsub -r 1d \
      -o ${name}.log  ./code  ${name}; done
```

- Here the standard output from each job will go to files "ic.*.log". As before, one can add "--idfile" stuff to enable meta-job manipulation.

Command line

- For FORTRAN codes (which typically use standard input instead of command line arguments), the previous example can be re-written as follows:

```
$$  for name in ic.*; do sqsub -r 1d \
      -o ${name}.log  -i ${name}  ./code; done
```

- Here we use the sqsub option “-i input_file” to provide the file for the job's standard input.

Command line

- In another common scenario – when arguments to be used in serial farming are stored in one text file, one line per job, it is convenient to use BASH's “while” loop command.
- E.g. your code needs three numbers as its command line arguments. Your IC table (say “IC.dat”) might look like this:

```
10 4.5 1.1e9
20 3.7 7.5e8
...
50 4.8 1.1e9
```

Command line

- “While” loop can be used here as follows:
  $$ i=0; while read a b c; do i=$(($i+1)); \
  sqsub -r 1d -o out.${i} ./code $a $b $c; done \
  < IC.dat
- Here the standard output will go to files out.1 ... out.N (N – number of jobs). You can add “--idfile” as before.
- “read a b c” reads all the three columns from the file IC.dat to shell variables $a, $b, $c, one line at a time.

Command line

- As the most complex (and perhaps the most realistic) scenario, let's consider the case when each job needs to run in a separate sub-directory (because it creates one or more output files).
- Expanding upon the previous slide's example, we can accomplish this task as follows:

```
$$  i=0;  while read a b c; do i=$(($i+1)); \
mkdir RUN$i; cd RUN$i; sqsub -r 1d -o out \
../code $a $b $c; cd ..; done < IC.dat
```

BASH scripting

- The previous example shows that the command line approach has its limitations; it becomes too difficult to deal with a single line when the number of commands becomes too large.
- Instead, one can put all these commands in a text file – BASH script. Instead of using the command line's separator “;”, in a script one can place different commands on separate lines.

BASH scripting

- The previous example, written as a BASH script, will become

```
#!/bin/bash
i=0
while read a b c
  do
  i=$(($i+1))
  mkdir RUN$i
  cd RUN$i
  sqsub -r 1d -o out ../code $a $b $c
  cd ..
  done < IC.dat
```

BASH scripting

- Here “#!/bin/bash” tells the OS that this is a BASH script (for Python you'd put “#!/usr/bin/python”).
- The script file (say, “metajob.sh”) should be made executable:
  $$ chmod u+x metajob.sh
- Then one can run it as follows:
  $$ ./metajob.sh

BASH scripting

- BASH scripts can accept command line arguments.
- E.g. the previous script can be modified to accept two arguments: the full path to the code, and the directory where the script should run (where the file IC.dat is located).
- One would run the script like this:
  $$ ./metajob.sh /home/user/bin/code \
  /work/user/case1

BASH scripting

- The required changes to the metajob.sh file are:

```
#!/bin/bash
CODE=$1
WD=$2
cd $WD
i=0
while read a b c
  do
  i=$(($i+1))
  mkdir RUN$i
  cd RUN$i
  sqsub -r 1d -o out $CODE $a $b $c
  cd ..
  done < IC.dat
```

Array jobs

- Schedulers often provide a way to submit a whole array of jobs in a single submission step.
- These are called “job arrays” or “array jobs”, and are designed to handle serial/parallel farming scenarios.
- Typically an array job would launch N instances of a user-provided script, and initialize a special environment variable to be equal to the job “rank” within the array.

Array jobs

- The task of the user script is to read the rank variable, and use it to customize the job execution (e.g., by making the user's code read the IC.$rank file).
- To run array jobs on SHARCNET clusters one has to use directly the underlying scheduler commands (like qsub) instead of the SHARCNET wrappers sqsub, sqkill etc.

Array jobs

- For example, to submit an array job one has to add “-t” switch to qsub:

  qsub -t x-y

- Here x and y are array bounds (define the range of “ranks”).
- For more information, you can check

  http://wiki.hpc.ufl.edu/doc/Torque_Job_Arrays

Advanced serial farming

Small number of cases

- Let's call a single execution of the code in a serial/parallel farm a “case”.
- When the total number of cases, N_cases, is fairly small (say, <500) it is convenient to dedicate a separate job to each case – the way it was handled in the previous section (“Command line”).

Small number of cases

- I created a set of BASH scripts utilizing both the “one case – one job” approach (works for cases when the number of jobs is <500 or so) and “dynamic workload balancing” approach (best for larger number of cases).
- The scripts can be found on our clusters in the following directory:
  ~syam/Serial_farming/META
- The two essential scripts are “submit.run” and “single_case.run”.

Small number of cases

- “submit.run” has two obligatory command line arguments.
- The first one is the path to the cases table, containing information (command line arguments, and/or the name(s) of the input file(s)) for your code, each line corresponding to a case to be executed.
- The second argument (when used in the “one job per case” mode) should be “-1”, e.g.

```
$$ ./submit.run  /work/user/dir/cases.dat  -1
```

Small number of cases

- The other principal script, “single_case.run”, is the only script which might need customization.
- Its task is to read the corresponding line from the case table, parse it, and use these data to launch your code for this particular case.
- The version of the file provided literally executes one full line from the case table (meaning that the line should also include path to your code) in a separate subdirectory, RUNyyy (yyy being the case number).

Small number of cases

- “single_case.run”:

```
...
# ++++++++++++  This part can be customized:  +++++++++++++++++
#  $ID contains the case id from the original table
#  $COMM is the line corresponding to the case $ID in the original table, without the ID field
mkdir RUN$ID
cd RUN$ID
# Executing the command:
$COMM &> out.log
# Exit status of the code:
STATUS=$?
cd ..
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
...
```

Small number of cases

- Your “cases.dat” table can look like this:

```
/home/user/bin/code1  1.0  10  2.1
./code2 < IC.2
sleep 10m
...
```

- In other words, any executable statement which can be written on one line can go there.
- Note: “submit.run” will modify your cases table once (add line # at the beginning of each line).

Small number of cases

- What is “$STATUS” for in “single_case.run”?
  - It is a shell variable which should be equal to “0” if your case was computed correctly, and >0 otherwise.
  - It is very important: it is used by “resubmit.run” to figure out which cases failed, so they can be re-computed.
  - In the provided version of “single_case.run”, $STATUS only reflects the exit code of your program. This likely won't cover all problems.

Small number of cases

- (continued)
  - You can always change or augment $STATUS derivation in “single_case.run”.
  - E.g., if your code creates a new file (say, “out.dat”) at the very end of a case run, the existence of such a file can be used to judge if the case failed or not:

```
STATUS=$?
if test ! -s out.dat
   then
   STATUS=1
   fi
```

Small number of cases

- The “submit.run” script will generate some files in the current directory:
  - out.cluster.xxx files (one file per job): standard output from jobs;
  - status.yyy files (one file per case): files containing the status of processed cases; yyy is the case number in the original table.

Small number of cases

- Other auxiliary scripts are also provided for your convenience.
- “list.run” will list all the jobs with their current state for the serial/parallel farm (no arguments).
- “query.run” will provide a one line summary (number of queued / running / done jobs) in the farm, which is more convenient than using “list.run” when the number of jobs is large. It will also “prune” queued jobs if warranted (see the next slide).

Small number of cases

- “kill.run”: will kill all the running/queued jobs in the farm.
- “prune.run”: will only kill queued jobs.
- “Status.run” (capital “S”!) will list statuses of all processed cases.
- Finally, script “resubmit.run” is run the same way as “submit.run”, e.g.:

```
$$  ./resubmit.run  /work/user/dir/cases.dat  -1
```

Small number of cases

- “resubmit.run”:
  - will analyze all those status.* files;
  - figure out which cases failed and which never ran for whatever reason (e.g. because of the 7d runtime limit);
  - create a new case table (adding “_” at the end of the original table name), which lists only the cases which still need to be run;
  - uses “submit.run” internally to launch a new farm, for the unfinished/failed jobs.

Small number of cases

- Notes:
  - You won't be able to run “resubmit.run” until all the jobs from the original run are done or killed.
  - If some cases still fail or do not run, one can resubmit the farm again and again, with the same arguments as before:

```
$$  ./resubmit.run  /work/user/dir/cases.dat  -1
```

Small number of cases

- Previous slides described the situation when you run your farm on a single cluster.
- This is the default behaviour when the current directory doesn't have file “clusters.lst”.
- If the file “clusters.lst” is present and is not empty, the scripts switch to a “meta-cluster” mode.

Small number of cases

- Proper “clusters.lst” file should contain a list of clusters which you wish to use for you serial farm, one cluster per line. E.g.:

  kraken
  saw
  redfin

- The current (local) cluster doesn't have to be in the list.

Small number of cases

- When you both use “-1” argument (one job per case mode) **and** provide “clusters.lst” file, “submit.run” and “resubmit.run” will submit 2*N_cases jobs across all the clusters in the list.
- Only the first N_cases jobs to start will do the computations; the rest will get pruned or die instantly.

Small number of cases

- This is how it works:
  - The very first job which runs (on any cluster) will request a case to process (case #1 initially);
  - The second job to run will ask for the next available case (#2), and so on.
  - To accomplish this, certain operations need to be serialized (only one job at a time can do it):
    - Read “currently served” case number from a file.
    - Update it (add “1”), and write it back to the file.

Small number of cases

- To serialize file-related operations, you need a special binary program, “lockfile”.
  - It is included in the META package.
  - It should be placed in a directory listed in your $PATH environment variable. You can accomplish this by

```
$$ mkdir ~/bin
$$ cp lockfile ~/bin
$$ echo 'export PATH=/home/$USER/bin:$PATH' \
     >> ~/.bashrc
$$ . ~/.bashrc
```

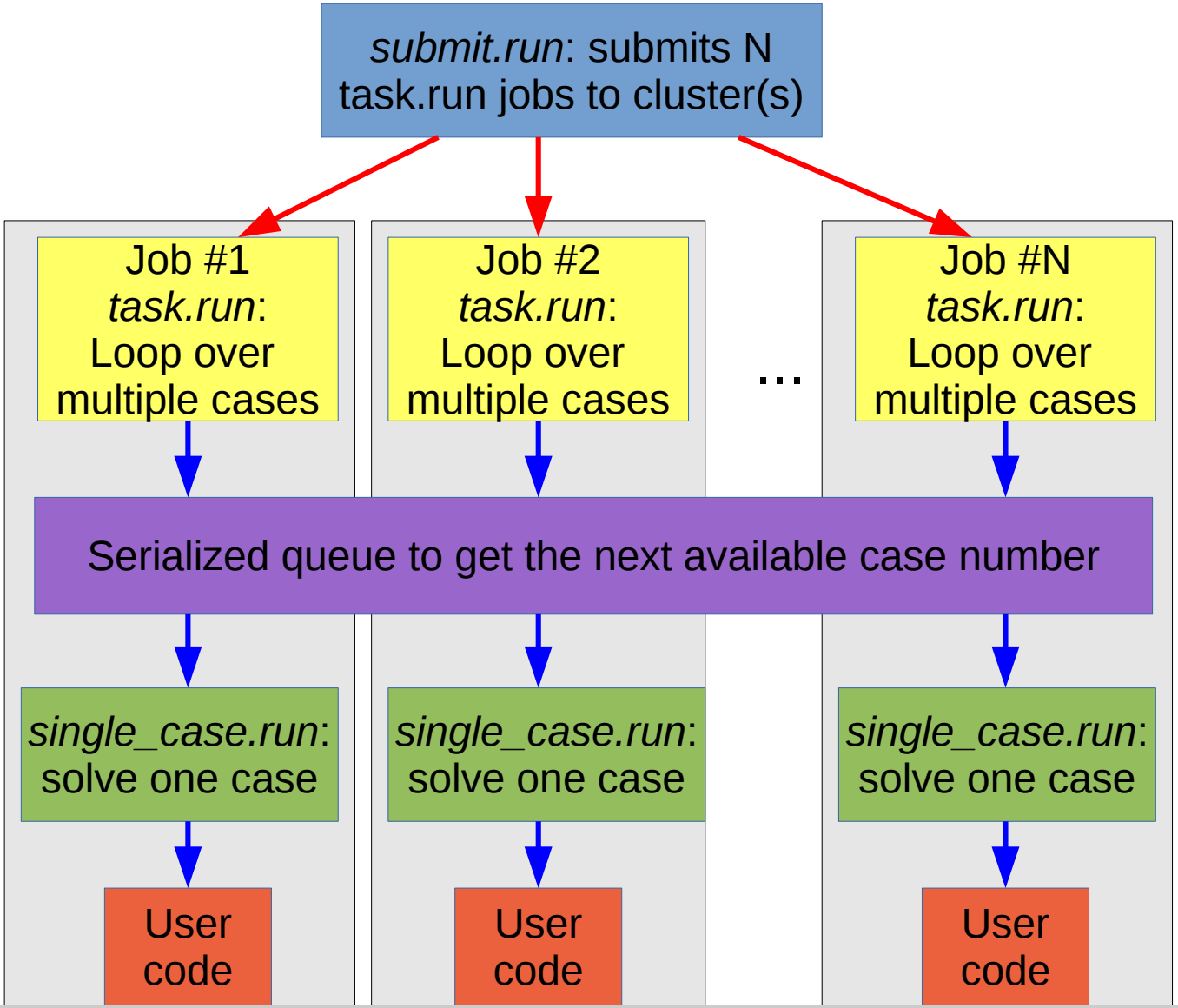
Large number of cases

- The “one job per case” works fine when the number of cases is fairly small (<500).
- When N_cases >> 500, the following problems arise:
  - There is a limit on the number of jobs submitted (for orca, it is 5000).
  - Job submission becomes very slow. (With 5000 jobs and ~2s per job submission, the submission will last ~3 hours!).
  - With very large number of cases, each case run is typically short. If one case runs for <30 min, you start wasting cpu cycles due to scheduling overheads (~30s per job).

Large number of cases

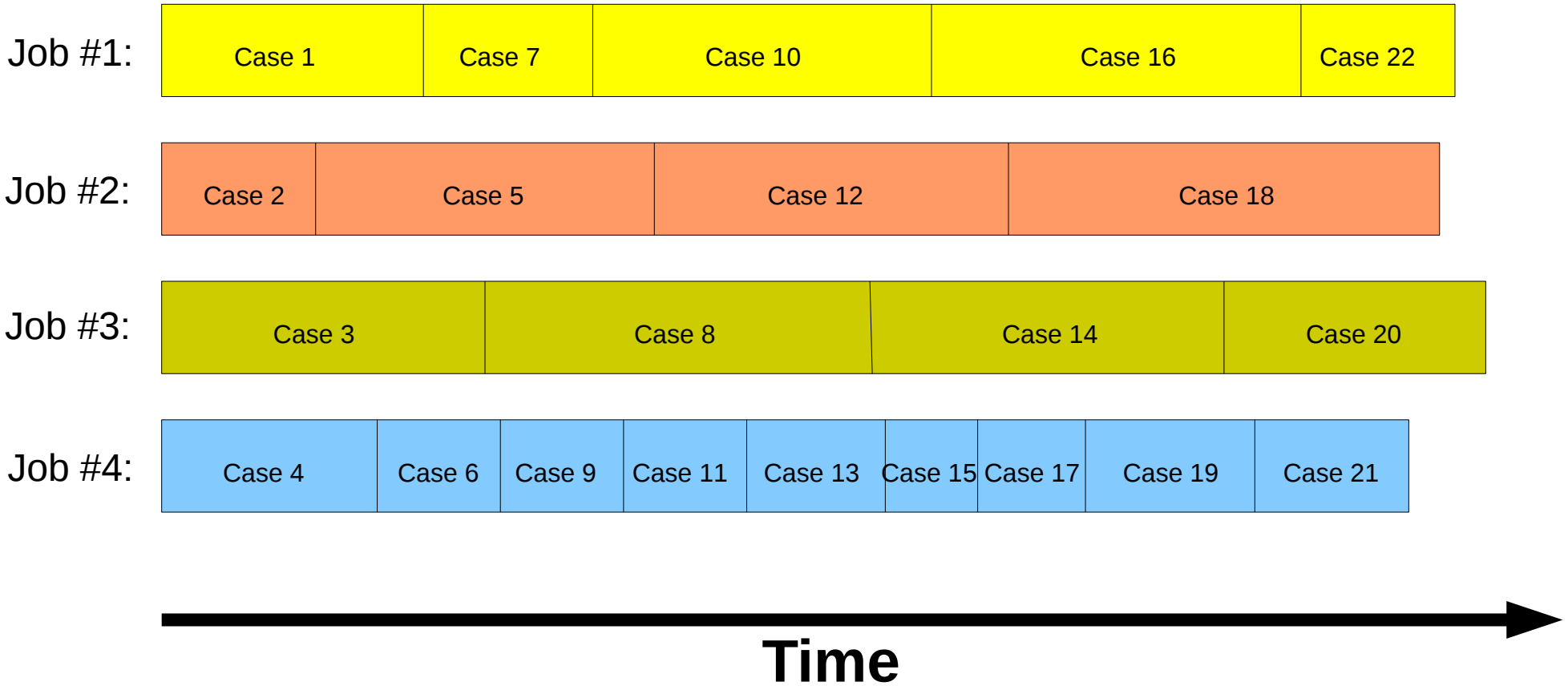
- The solution: instead of submitting a separate job for each case, one should submit a smaller number of jobs, each of which would process multiple cases.
- As cases can take different time to process, it is highly desirable to utilize a dynamic workload balancing scheme.
- Let's see how it is implemented.

Large number of cases



Large number of cases

- Dynamic workload balancing at work:



Large number of cases

- To enable the “multiple cases per job” mode (with dynamic workload balancing), the second argument to “submit.run” script should be the number of “task.run” jobs requested, e.g.:

```
$$  ./submit.run  /work/user/dir/cases.dat  32
```

  - If file “clusters.lst” is present, listing multiple clusters, one per line, the number of submitted jobs will be the above number x2, and is limited to 256*2=512 jobs.

Using the scripts

- Simple usage scenario:
  - Copy directory /home/syam/Serial_farming/META to your file space (usually in /work/$USER).
  - Put “lockfile” utility on your $PATH (see slide 45).
  - Copy your code and initial conditions files to the META directory (optional; you can use full paths instead).
  - Create a cases table (text file, one case per line).
  - (To run another farm in parallel to the first one, create another directory - say, META1 - and repeat the above steps.)

Using the scripts

- (continued)
  - Modify “single_case.run” accordingly (so it will parse all the information the code needs from a specific line in the cases table).
  - Create file “clusters.lst” if you plan to use the meta-cluster feature. **Don't put requin** (it is not compatible with the scripts).
  - Probably a good idea to initially run on a single cluster (no “clusters.lst” file), with just a few jobs, to see if everything works.

Using the scripts

- What if you need to use additional sqsub arguments (like --mpp, -q mpi, -q threaded, -n etc.)?
  - Simple: just add all those arguments at the end of "submit.run" and "resubmit.run" command line, and they will be passed to sqsub, e.g.:

    $$ ./submit.run test.dat -1 -q mpi -n 4
  - You can also override the default job runtime value (encoded to be 7 days in "submit.run"), by adding a "-r" argument.

Using the scripts

- How to do “multi-threaded farming” (OpenMP etc.)?
  - Add these sqsub arguments to “(re)submit.run”:

    -q threaded -n N
  - Here “N” is the number of cpu cores/threads to use.
  - Nothing special needs to be done inside “single_case.run”: run the code as in the serial case.

Using the scripts

- What about “MPI farming”?
  - Use these sqsub arguments with “(re)submit.run”:

    ```
    -q mpi  -n N  --nompirun
    ```

  - Inside “single_case.run”, add “mpirun” before your code, e.g.:

    ```
    mpirun  $COMM &> out.log
    ```

  - (Alternatively, you can prepend “mpirun” on each line of your cases table.)

Using the scripts

- You have a FORTRAN serial code, “fcode”; each case needs to read a separate file from standard input – say “data.xxx” (in /work/user/IC directory), where xxx goes from 1 to N_cases.
  - Place “fcode” on your $PATH (e.g., in ~/bin, make sure /home/$USER/bin is added to $PATH in .bashrc).
  - Create the cases table (inside META directory) like this:

```
fcode < /work/user/IC/data.1
fcode < /work/user/IC/data.2
...
fcode < /work/user/IC/data.N_cases
```

Using the scripts

- (continued)
  - The task of creating the table can be greatly simplified if you use a BASH loop command, e.g.:

```
$$  for ((i=1; i<=10; i++)); do echo \
      "fcode < /work/user/IC/data.$i"; done >table.dat
```

Using the scripts

- Finally, another typical FORTRAN code situation: you need to copy a file (say, /path/to/data.xxx) to each case subdirectory.
  - Your cases table can look like this:

```
/path/to/code
/path/to/code
...
```

  - Add one line to your “single_case.run”:

```
\cp /path/to/data.$ID .
$COMM &> out.log
STATUS=$?
```

Word of caution

Word of caution

- Please use common sense when using the “meta-cluster” mode of the scripts.
  - Most of the jobs you submit should be expected to run (don't submit many more jobs than you might need - “just in case”).
  - The “$Overcommit_factor” variable defined in “submit.run” (=2) was introduced to force you to be reasonably compliant with the above point.

Word of caution

- (continued)
  - After submitting a farm to multiple clusters, regularly check its state with “query.run” script.
    - First, it will keep you informed about the overall “health” of your farm, and if things don't look right you can do something about it.
    - As importantly, “query.run” script will run “prune.run” script internally when it detects that you won't need the jobs which are still queued.

Word of caution

- Make sure your code only accesses /work or /home.
- Always start with a much smaller test farm run, to make sure everything works, before submitting a large production run farm.
- Stay away from orca (when using meta-cluster mode): it is very busy because of NRAC jobs.
- Don't include requin (the scripts don't work there).

Word of caution

- How to use contributed systems?
  - Some contributed systems (like brown and redfin) apparently still accept 7 day jobs, so use them as any other cluster.
  - Many contributed systems (perhaps all in the future) will only accept short – up to 4h – jobs.
  - To make full advantage of the contributed systems,
    - add “-r 4h” argument to “(re)submit.run”, and either
    - use “-1” mode of “(re)submit.run” if your cases take between 0.5 and 4 hours to run each, or
    - use “many cases per job” mode if each case takes <0.5h.

Questions?

- You can always contact me directly (syam@sharcnet.ca) if you need any help with job farming in general, or META scripts in particular.