OpenSeesMP Job Board Framework (mpjobs)

Version 0.1

Alex Baker

https://github.com/ambaker1/mpjobs

May 2, 2023

Abstract

OpenSeesMP provides basic MPI message-passing commands such as *getPID*, *getNP*, *send*, *recv*, and *barrier*, which can be used to run parametric studies in parallel. The mpjobs package provides an abstract framework for running parametric studies in series and parallel, eliminating the need to deal directly with the message passing commands. Additionally, it formalizes and organizes the parametric study in a way that supports save-state functionality, allowing the analyst to revisit a study and analyze it further.

General

The main command for the job module is *jobBoard*. All other job board commands must be called within the body of *jobBoard*. This command encapsulates an ensemble of subcommands used for creating, manipulating, running, and accessing jobs. This command can be called in series or in parallel (OpenSeesMP). If it is called in parallel, it must be called by all processes, but the body will only be evaluated on process 0. If the main process is aborted, any progress on active jobs will be lost, and will be reset the next time the job board is opened. To stop the job board without aborting active jobs, enter "stop" on the command line. This will signal to the main process to stop assigning/running new jobs, and will proceed to finish all active jobs and exit. The "-timeout" option does the same thing as entering "stop", but does so after a specified duration.

```
jobBoard <-debug> <-wipe> <-timeout $duration> $path $body
```

-wipe	Option to wipe the job board (*wipeJobs*) before loading job data.
-debug	Option to display debug information.
-timeout $duration	Option to time-out the job board analysis.
	Duration must be in format "HH:MM:SS".
$path	Folder containing job board files.
$body	Body to evaluate, with access to job board.

When *jobBoard* is called, it creates the job board folder **$path**, where job data is stored in files. The specific files created by the job board framework are listed below, where **$jobTag** is a non-negative integer:

Table 1: Job board file descriptions

Filename	Description
INPUTS	File containing Tcl list of job inputs.
SESSION	File containing system PID of process accessing job board.
STATUS	File containing status codes of all jobs.
$jobTag.tcl	Tcl script which runs job $jobTag.
$jobTag.log	Log of job $jobTag stderr channel.
$jobTag.dat	Results of job $jobTag.

Creating Jobs

The command *makeJob* creates a new job in the job board and returns the new job tag, unless if the given inputs exist, in which case it will simply return the matching job tag. Job tags start at 0 and increment each time a new job is created. The input file for a job must return a dictionary of results (or blank).

```
makeJob <$inputDir> $inputFile $var1 $value1 ...
```

$inputDir	Input directory for job. Either an absolute directory or relative to job board. Default "..", or parent directory to job board.
$inputFile	Input file for job
$var1 $value1 ...	Variable names and values to pass to the input file. Variable names "folder", "filename", and "status" are reserved.

Running Jobs

The command *runJobs* runs all jobs with status code 0 (available) in the job board in a separate instance of OpenSees, using the OpenSees executable on the same path as the current executable. If jobs are specified that have non-zero status codes, it will skip them. In series mode, it will run the jobs sequentially, and return as soon as all jobs are complete. In parallel mode, it will assign the jobs to the workers using a load-sharing scheme, and will return as soon as all jobs are assigned. In both series and parallel, if "stop" is entered to the main process stdin channel or if the timeout duration has been reached, it will not run or assign any jobs, but will simply return.

```
runJobs <$jobTags>
```

$jobTags	Jobs to run. Default "-all", for all available.

Example 1: Basic Use

Code:

```
jobBoard MyStudy {
    foreach a $aList {
        foreach b $bList {
            makeJob InputFile.tcl a $a b $b
        }
    }
    runJobs
}
```

Job Utilities

Removing all Jobs

The command *wipeJobs* removes all jobs and job data from the job board. If any jobs are active, it will wait for them to finish. This is also called by *jobBoard* if option -wipe is used.

```
wipeJobs
```

Resetting Jobs

The command *resetJobs* resets jobs to status 0, or "available". If jobs are specified that are already available, it will skip them. If active jobs are specified, it will wait for them to finish first. The intended use of this command is to re-run analyses when the analysis file changes.

```
resetJobs <$jobTags>
```

$jobTags	Jobs to reset. Default "-all", for all jobs with non-zero status.

Example 2: Re-running a parametric analysis

Code:

```
jobBoard MyStudy {
    resetJobs
    runJobs
}
```

Waiting for Jobs

The command *waitForJobs* waits for active jobs to be complete, and then updates the job statuses. This is essentially a "barrier" call for worker processes.

```
waitForJobs <$jobTags>
```

$jobTags	Jobs to wait for. Default "-all", or all active jobs.

Updating Jobs

The command *updateJobs* updates the status of jobs running in parallel. This is essentially the asynchronous version of *waitForJobs*, and is especially useful for dynamic parametric parallel studies.

```
updateJobs <$jobTags>
```

$jobTags	Jobs to update. Default "-all", or all active jobs.

Example 3: Asynchronous updating of job information

Code:

```
jobBoard MyStudy {
    foreach x {1 2 3} {
        makeJob MyFile x $x
    }
    runJobs
    # Asynchronously update job statuses and process completed jobs.
    while {[llength [getJobTags -active]] == 0} {
        updateJobs
        foreach jobTag [getJobTags -complete] {
            puts "Job $jobTag complete"
        }
        after 100; # Add a little buffer
    }
}
```

Job Information Queries

In addition to being able to create jobs and run them, the job board framework provides convenient routines to query current job board information.

Get List of Jobs

The command *getJobTags* returns either all the jobs in a job board or a subset specified by status or pattern, sorted in increasing order. If conflicting options are specified, the last options take precedence.

```
getJobTags <-all> <-available> <-active> <-complete> <-failed> <$codes>
```

`-all`	All job tags (default).
`-available`	Only available jobs (status code 0).
`-active`	Only active jobs (status code 1).
`-complete`	Only completed jobs (status code 2).
`-failed`	Only failed jobs (status code 3).
`$codes`	List of status codes.

Total Number of Jobs

The total number of created jobs can be queried with the command *getJobCount*.

```
getJobCount
```

Get Job Inputs

Job inputs can be queried with *getJobInputs*. Returns a dictionary, with keys “folder”, “filename”, and any defined input variables.

```
getJobInputs $jobTag
```

`$jobTag`	Integer job tag.

Get Job Status Code

The command *getJobStatus* returns the status of a job. Note that *waitForJobs* or *updateJobs* must be called to update active job statuses.

```
getJobStatus $jobTag
```

$jobTag	Integer job tag.

Below are the job status codes and their meanings:

0 Job is available to run.

1 Job is active.

2 Job completed successfully.

3 Job completed with error.

Get Job Results

Job results can be queried with *getJobResults*. If the job status is < 2, it will return blank. If the job status is 2, it will return a dictionary of results. If the job status is 3, it will return the option dictionary detailing the cause of the error.

```
getJobResults $jobTag
```

$jobTag	Name of job to query results.

Get Job Table

The command *getJobTable* returns a Tda table with rows corresponding to jobs, and columns corresponding to job inputs, status, and results. Column headers are as follows: "jobTag", "status", "folder", "filename", any input variables, and any output variables.

```
getJobTable <$jobTags>
```

$jobTags	Job tags to set as table keys. Default "-all" for all jobs.

Tda is available here: https://github.com/ambaker1/Tda.

Example Application

For this example, the displacement and moment of an elastic cantilever column was investigated for different geometry parameters. Note that the input variables "L" and "I" have defaults set by using the "vutil" package *default* command. Also note the definition of a result dictionary at the end of the file.

Example 4: Elastic cantilever column (Cantilever.tcl)

Code:

```
# Elastic Cantilever Column
package require tin
tin import default from vutil
model BasicBuilder -ndm 2 -ndf 3

# Variables (with defaults)
default L 10.0; # in
default I 1000.0; # in^4
set A 100.0; # in^2 (should not affect results)
set E 29000.0; # ksi

# Define nodes
node 1 0 0
node 2 $L 0
fix 1 1 1 1

# Define element
geomTransf Linear 1
element elasticBeamColumn 1 1 2 $A $E $I 1

# Add load
timeSeries Linear 1
pattern Plain 1 1 {
    load 2 0 1 0
}

# Setup analysis
constraints Plain
numberer RCM
system BandGeneral
test NormDispIncr 1.0e-8 6
algorithm Newton
integrator LoadControl 1.0
analysis Static

# Perform analysis
analyze 10

# Return results
dict set results disp [expr {double([nodeDisp 2 2])}]
dict set results moment [expr {double([localForce 1 3])}]
return $results; # This is required for the job board framework
```

For this example, the length and moment of inertia of the column were varied, and the effect on the end displacement and cantilever moment were investigated, and the results are shown in Table 2. Note the use of the "tda" package command *writeTable*.

Example 5: Simple parameter study

Code:

```
# Parameter study of cantilever column
package require tin
tin import mpjobs
tin import writeTable from tda
jobBoard -wipe CantileverStudy {
    foreach L {10 20 30} {
        foreach I {100 200 300} {
            makeJob Cantilever.tcl L $L I $I
        }
    }
    runJobs
    waitForJobs
    writeTable CantileverStudy/results.csv [getJobTable]
}
```

Table 2: Cantilever column parametric study results

jobTag	status	folder	filename	L	I	disp	moment
0	2	..	Cantilever.tcl	10	100	0.001149	-100
1	2	..	Cantilever.tcl	10	200	0.000575	-100
2	2	..	Cantilever.tcl	10	300	0.000383	-100
5	2	..	Cantilever.tcl	20	100	0.009195	-200
6	2	..	Cantilever.tcl	20	200	0.004598	-200
7	2	..	Cantilever.tcl	20	300	0.003065	-200
10	2	..	Cantilever.tcl	30	100	0.031034	-300
11	2	..	Cantilever.tcl	30	200	0.015517	-300
12	2	..	Cantilever.tcl	30	300	0.010345	-300