Jobman: Computation Job Management

Heng Sun

September 10, 2015

Contents

1 Introduction

This section gives an overview of Jobman with a simple example. You may go to section 5 directly for all Jobman specifications. You may also refer to section 6 for what each executable does in a demo. In all the demos, we "pretend" the multiplications of two integers are expensive operations, which we would parallelize.

1.1 What is Jobman

Computation Job Management (Jobman) provides a framework for multiprocess computing. It is a program to call executables according the control flow specified by the user. Each executable is run in a separate process.

Jobman is intended to be used in the following situation. A project needs to carry out a series of jobs that are performed by software written in different languages and/or supplied by third parties, according to certain program flow. The program may run in a machine with multiple processors. Writing multiprocessed (and thread safe) program is not your interest. However, invoking each individual program manually is error prone and hard to manage. This is especially the case in some scientific computing, quantitative finance, and prototyped programming. The various individual programs don't frequently communicate with each other except via persistent storage like databases and flat files, where inputs are read and outputs are written. In short, Jobman treats individual executables like Lego blocks. It lets a user to build a program from those blocks in a flexible way.

Jobman has the following features:

- Glue individual executables into a program according to a given program flow.
- Run individual executables in multiple processes.
- It supports some dynamic alterations of execution flows. Certain execution paths and user preferences can be modified when the program is running.
- It supports resumed run. The program can start from where it was left last time.
- The user can prioritize jobs when the number of jobs to be run in parallel exceeds the maximum number of parallels allowed.

In Jobman, the program flow is expressed in certain configuration files. Each collection of tasks to be performed by individual executables is called a 'job'. The basic communications between individual jobs are via text files.

In this document, we use Unix file path separator '/' for illustration. For Windows users, replace '/' with '\', all else the same.

The project is hosted at SourceForge. The distribution of this program consists of a single JAR file `jobman.jar`. To use Jobman, you need a JRE (Java Runtime Environment). But you don't need to know Java. This JAR file, together with the source code, documentations, and demos, can be downloaded there.

1.2 An Example: Demo 1

To use Jobman, you need to do two things:

- Implement basic computing blocks.
- Express the desired execution order in a Jobman configuration file.

As an example, we try to find out all prime numbers no more than 10. Assume we already know 2 and 3 are all the primes less than $\sqrt{10}$. We can use the sieve method. To use this method, we find out all multiples of 2 between 3 and 10. We also find out all multiples of 3 between 4 and 10. Then all the numbers between 2 and 10 that are not in the above multiples are the primes we are looking for.

The basic computing block is some code to list all multiples of a given numbers within a specified range. The Jobman configuration file controls how the computing blocks are executed. Each line in the configuration file has the format

```
<job type> <job name> <job specification>
```

For example, a Jobman configuration file may look like below.

```
1   loop all setup multiples primes
2   exec setup java -cp work/classes demo.Setup work/out
3   fork multiples mult2 mult3
4   exec mult2 java -cp work/classes demo.Multip 1 10 2 work/out/m2.txt
5   exec mult3 java -cp work/classes demo.Multip 1 10 3 work/out/m3.txt
6   exec primes java -cp work/classes demo.Primes 2 10 work/out
```

The line number at the beginning of each line is added to identify a line. It is not part of the configuration file. There are three job types used in the above configuration:

loop Specifies a parent job consisting of child jobs that run in a sequential order.

fork Specifies a parent job consisting of child jobs that run in parallel.

exec Specifies an executable that perform an individual computation. An `exec` job is always run in a separate process.

In the above example, line 1 specifies a job named `all` consists of three child jobs: `setup`, `multiples`, and `primes`. They are executed in a linear order. Job `setup` goes first. After it is finished, Job `multiple` is executed. Job `primes` is the last one to be run.

Line 2 specifies job `setup` is an executable, whose command to run is

```
java -cp work/classes demo.Setup work/out
```

This command is implemented in Java. It does a set-up work for later computation. It accepts one argument: the directory to clean, and to create if it does not already exist. You could use any computing language (or third party software) to implement. For example, you could write a simple shell script in Linux (or batch file in Windows). Let's call it `setup.sh`. Then line 2 would be

```
exec setup setup.sh work/out
```

An `exec` needs a way to communicate its execution status to Jobman. There can be several ways. In this demo, we assume the successful execution is confirmed by outputting a string `JOBMAN_STEP_COMPLETED`. If Jobman finds this string in the standard output of the executable, it will consider the job has been executed successfully. Otherwise, it will not continues executions of other jobs.

Line 3 specifies job `multiples` consists of two child jobs: `mult2` and `mult3`. Each child job is an executable as specified in lines 4 and 5. For example, line 4 specifies `mult2` is run by executing

```
java -cp work/classes demo.Multip 1 10 2 work/out/m2.txt
```

If you look into the Java source code, you will see it accept 4 arguments. The first and the second arguments specify the range of integers, which is between 1 and 10 in this case. The third argument specifies the common difference, i.e., the multiples of which number. In our case, it is 2. The last argument is the output file path to stored the multiples. This Java program will give result 4, 6, 8, and 10, each is written as one line into file `work/out/m2.txt`. As before, you can implement this in any computing language.

The last job `primes` appeared in line 1 is specified in line 6. It is an executable by calling command

```
java -cp work/classes demo.Primes 2 10 work/out
```

What it does is to go into directory `work/out` and to find out the two files containing multiples of 2 and 3. Exclude those multiples from numbers ranging from 2 to 10. We thus obtain all the primes between 2 and 10.

1.3 More on Configuration File

As as summary, each line in a job configuration file has three parts:

1. The first word is the job type. It always consists of four ASCII characters. For examples, `loop`, `exec`, `fork`.
2. The second word is the job name. It consists of alphanumeric characters and possible underscore "_". Other characters are not allowed.
3. The third part of the job specification is all the words after the job name. Its format depends on the job type. For an `exec` job, it is the command to be run. For a `loop` or `fork` job, it is the list of child jobs to be run.

For the convenience of managing a job configuration file, a comment line is allowed. A comment line starts with "#". Any such a line will be ignored by Jobman when the configuration file is processed. An empty line will also be ignored.

You can also define a variable for some string you would use repeatedly. The definition of a variable is given by

```
defn <variable> <value string>
```

For example, in the previous section, we always use `java -cp work/classes` to define a command running Java. Then we can define

```
defn jv java -cp work/classes
```

Use the dollar sign "$" to extract the value of a variable. For example, the job `setup` can be specified by

```
exec setup $jv demo.Setup work/out
```

When Jobman processes the configuration file, the above line will be expanded into

```
exec setup java -cp work/classes demo.Setup work/out
```

With the above understandings, the configuration file in the previous section can be written as

```
# demo_1 config
# A simple program to list all primes between 1 and 10.

defn jv java -cp work/classes
defn outdir work/out
```

```
loop all setup multiples primes
exec setup $jv demo.Setup $outdir
fork multiples mult2 mult3
# Compute nontrivial multiples of 2 between 1 and 10.
exec mult2 $jv demo.Multip 1 10 2 $outdir/m2.txt
# Compute nontrivial multiples of 3 between 1 and 10.
exec mult3 $jv demo.Multip 1 10 3 $outdir/m3.txt
# List all numbers between 2 and 10 that are not multiples of 2 or 3.
exec primes $jv demo.Primes 2 10 $outdir
```

This is the contents of the configuration file in Demo 1. Remark that the order of the lines in a configuration file does not matter.

1.4 Running Demo 1

The above example is implemented in Demo 1. To run it, you need to have a Java Runtime Environment 8 or above. Go to the unpacked directory of `jobman`. Then go to subdirectory `demo`. Execute command `jobman.bat`. Assume `java` is in your path. This command runs on both a Windows and Unix/Linux machine. (I don't have a Mac computer.)

Or equivalently, you can run the contents in file `jobman.bat` directly in a command line interface:

```
java -jar ../dist/jobman.jar
```

If `java` is not in your path, you can use the full path to `java` to replace the `java` above.

The full command is

```
java -jar ../dist/jobman.jar <job name>
```

If the job name is missing, Jobman assumes a job named `all` is defined and runs

```
java -jar ../dist/jobman.jar all
```

You can also run other jobs. For example

```
java -jar ../dist/jobman.jar setup
```

will run job `setup` only.

1.5 File Structure

Let's call the directory where Jobman runs the Jobman home directory. Under this directory, there must be a pre-existing subdirectory `cfg`, which contains a mandatory file named `job_list.txt`. The file `job_list.txt` is the job configuration file described in sections 1.2 and 1.3. This is the only mandatory input file to run Jobman. There are several optional input files to be described in later sections.

After Jobman runs, a new directory `log` will be created. If it exists before Jobman runs, it may be emptied first if the user specifies to run Jobman from start, or to be reused if the user specifies to resume the previous run.

Directory `log` is the Jobman output directory. It has three types of files

- A general log file `jobman.log` contains messages output by Jobman.
- A comma separated file `jobman.csv` contains important job state when a job changes its status.
- Directories and files particular to individual jobs. These are used by job processes to communicate with the Jobman main process.

There are two categories of jobs specified in a job configuration file. One category of jobs are nodes. They contains child jobs. For example, `loop` and `fork` are nodes. The other category of jobs are leafs. They don't contain other jobs. For example, `exec` is a leaf. The `exec` is the only job type the user of Jobman needs to implement. The implementation should produce an executable that performs certain activities according to a specific business logic.

In directory `log`, a node job has a directory. All the output information from its child job is contained in that directory. A leaf job may have some output files. For example, The success status string `JOBMAN_STEP_COMPLETED` output from an `exec` job `setup` mentioned in section 1.2 is captured by Jobman and written into file `demo/logs/all/setup.log` in the Jobman home.

In directory `demo`, there is a subdirectory `work`, which is not part of Jobman file structure. It is used by the programs that implement `exec` jobs. Directory `demo/work/out` contains the outputs of those `exec` jobs. For example, file `demo/work/out/primes.txt` contains the list of primes no more than 10, which is the computation result of Demo 1. We'll leave the details of other subdirectories to later when other demos are explained.

2 Parameterized Jobs

2.1 Loops and Forks with Parameters

Defining a `loop` or `fork` jobs requires enumeration of all its child jobs. In many situations, we need loop or fork over child jobs that are indexed with different arguments or parameters. The job types to achieve this are the following two parameterized job types.

plop Specifies a parent job consists of child jobs with different parameters, executed in the order of parameters.

pfrk Specifies a parent job consists of child jobs with different parameters, executed in parallel.

The syntax to form such a `plop` or `pfrk` job in job configuration file is as follows:

```
<plop|pfrk> <job name> <parameter job> <child job>
```

There are four words in the job specification. The first one is either `plop` or `pfrk` specifying the parameterized jobs will be run in sequence or in parallel. The second word is the job name in definition. The last word is the child job performing a task with a given parameter as an argument. The third word is the job listing parameters to be provided to the child job. It must be an `exec` job writing parameters into a text file, one per line. It has a variable `$par` as an argument. This variable will be replaced by a file path where the parameters will be written.

The child job in a parameterized job has a parameter going with it when Jobman executes it. The parameter can be accessed using `$<child job>` in the job specification. For example,

```
pfrk composites ten multip
```

specifies `composites` is a parametrized job. The parameters are provided by a leaf job `ten`. For each parameter, child job `multip` will be run. The parameter can be accessed by variable `$multip`.

The child job may be a node job containing some grandchild jobs. Each of the grandchild jobs also has the access to the parameter. In the above example, a child job of `multip` can access parameter `$multip`.

Many times the parameter job needs to produce a sequence of numbers in a range. You can write your own executable to achieve this. Jobman provides a convenience implementation using job type `rnge`. It has syntax

```
rnge <job name> <file path> <start> <end>
```

The file path is the location where the range of numbers will be written. Use `$par` to let Jobman write into the parameter file. The last two words in the above specification are the first number and the last numbers respectively. For example

```
rnge ten $par 2 10
```

produces numbers 2 to 10 and writes into the job parameter file.

2.2 Example on Parameterized Job: Demo 2

In this example, we are trying to find out all primes no larger than 100. By the sieve method, we need find out all composites, which are the numbers up to 100 that are multiples of numbers between 2 and 10. The job configuration file is given below. (Line numbers are not part of the configuration.)

```
1   defn jv java -cp work/classes
2   defn outdir work/out
3   loop all setup composites primes
4   exec setup $jv demo.Setup $outdir
5   pfrk composites ten multip
6   rnge ten $par 2 10
7   exec multip $jv demo.Multip 1 100 $multip $outdir/$multip[].txt
8   exec primes $jv demo.Primes 1 100 $outdir
```

The top job `all` consists of three child jobs, in sequence. The second child job `composites` finds all composites between 2 and 100. It writes all nontrivial multiples of 2 into file `work\out\2.txt`; all nontrivial multiples of 3 into file `work\out\3.txt`, etc. The third child job `primes`, defined on line 8, simply filters out all composites from numbers 2 to 100. It reads those files in `work\out` that are multiples into a list. All those numbers between 2 and 100 that are not in this list are written into file `primes.txt`.

The main calculation is carried out by `pfrk` job `composites` which is explained in the previous section. Line 7 specifies `multip` is an `exec` job accepting four arguments. The first two specify the range of integers. The third argument is the common factor of the multiples, which is one in the number list provided by job `ten`. The last argument is the path to the file where the composites are written. For example, when Jobman execute `multip` with parameter 5, it will call the following command

```
java -cp work/classes demo.Multip 1 100 5 work/out/5.txt
```

The result of this executable is that all nontrivial multiples of 5 between 2 and 100 will be written into file `5.txt`. The numbers are 10, 15, 20, 25, ..., 100.

A remark on how parameter variable is accessed in `$multip` and `$multip[]`. A parameter can be a space separated string, which Jobman considers as an array of strings tokenized by the space character. Each element in the string can be accessed via indexing [].

For example, if a parameter to job `myjob` is string

```
arg1 arg2 arg3
```

then `$myjob[1]` is `arg1`. So Jobman uses 1-based indexing. Both `$myjob[]` and `$myjob` would give the full parameter "`arg1 arg2 arg3`.

In the earlier example, since `multip.txt` is not a defined variable in `$multip.txt`, we should use `$multip[].txt`. In short `[]` is served as an end of variable identifier.

To run Demo 2, delete file `demo/cfg/job_list.txt`. Copy `examples/job_list_2.txt` into `demo/cfg` and rename it to `job_list.txt`. Then execute `demo/jobman.bat`. The list of primes is given in file `demo/work/out/primes.txt`. Check the directory `demo/logs`, in particular, file `demo/logs/jobman.csv`, to see how each job was run.

2.3 More Efficient Way to Find Primes: Demo 3

In the above section, we find composites between 1 and 100 by finding the multiples of all numbers between 2 and 10. This is not efficient since we only need to find the multiples of primes between 2 and 10. Demo 3 shows how this can be achieved.

Assume we are interested in finding all primes between 2 and 500. We know all primes in $(1, 2]$. Here $(a, b]$ denotes the set of all integers between a (exclusive) and b (inclusive). Then the sieve method can find all composites, and hence all primes, in $(2, 4]$. Then again the sieve method can find all composites, and hence all primes, in $(4, 16]$.

The general statement is that if we know all primes in $(1, x]$, then the sieve method gives us all primes in $(1, x^2]$ since all the composites in this interval must be a nontrivial multiple of a prime in $(1, x]$. This gives us a way to work out primes to a larger range recursively. The idea can be express in the following job configuration file. (Line numbers are not part of the file.)

```
 1  loop all setup findp
 2  exec setup $jv demo.Setup $outdir
 3  plop findp intervals sieve
 4  # Partition (1,500] into intervals (1,2], (2,4], (4,16], (16,256], (256,500].
 5  exec intervals $jv demo.Intervals 500 $par
 6  loop sieve composites allprimes
 7  pfrk composites pfac multip
 8  # Read off primes from the previous primes file.
 9  exec pfac $jv demo.ListFactors $outdir $sieve[1] $sieve[3] $par
10  exec multip $jv demo.Multip $sieve[2] $sieve[3] $multip $outdir/$sieve[1]/$multip[].txt
11  # List all prime numbers between 2 and 500.
12  loop allprimes primes merge
13  exec primes $jv demo.Primes $sieve[2] $sieve[3] $outdir/$sieve[1]
14  exec merge $jv demo.Merge $sieve[1] $outdir
15  defn jv java -cp work/classes
16  defn outdir work/out
```

As in Demo 2, to run Demo 3, we need to copy file `examples/job_list_3.txt` to `demo/cfg/job_list.txt` and execute `demo/jobman.bat`. The list of all primes are given in `demo/work/out/5/primes_all.txt`.

We'll make the following comments on this Demo:

- Parameterized job `findp` on line 3 loops over intervals in order since we must have all primes in the smaller interval to find all composites in the larger interval.
- The parameter job `intervals` for its parent `findp` is defined on line 5, implemented as `exec` job by user. Its output parameters go to location specified by `$par`, which is `all/findp/intervals.par`. In that file, each parameter contains three numbers:

  $$j, \quad 2^{2^{j-1}}, \quad 2^{2^j}$$

  for $1 \le j \le 5$ unless the last number is larger than 500. Later they are accessed in lines 9, 10, and 13, via variable `$sieve` and array indexing.
- Parameterized job `composites` on line 7 computes multiples in parallel.
- The parameter job `pfac`, defined on line 9, uses all primes in the earlier interval as parameters. In fact, its implementation is simply copying the list of earlier primes into the job parameter file. For example, my run of Demo 3 has the following line in the log file `demo/logs/jobman.log` in my computer running Windows:

```
Starting exec job \all\findp\sieve.4_16_256\composites\pfac using command:
java -cp work/classes demo.ListFactors work/out 4 256
E:\dev\jobman\demo\logs\all\findp\sieve.4_16_256\composites\pfac.par
```

The above is one line in the log file, which is broken into three for ease of reading. The executable (implemented in Java) simply copies the file `demo/out/3/all_primes.txt` to the parameter file location given by the last argument to the executable.

3 More Job Types

In practice, the five job types `exec`, `loop`, `fork`, `plop`, and `pfrk` introduced earlier should be enough to build most of program flows. This section introduces more job types which, as an example, can improve our prime number search performance by further parallelizing processes.

3.1 Job Communication Files

Each job has a string representation, which is the concatenation of the job names and the parameters of all its ancestors separated by the operating system's path separator. The job name and its parameter are separated by dot ".". All non-alphanumeric characters are replaced by the underscore character "_". For example, in my computer running Windows, in Demo 3 described in section 2.3, the job to find out all multiples of 3 between 4 and 16 is represented as

```
\all\findp\sieve.3_4_16\composites\multip.3
```

You can find this in file `demo/logs/jobman.csv` and `jobman.log`.

Process communications in Jobman are through files in the log directory. Each job can have up to three associated files, whose path relative to the Jobman log directory is its string representation affixed by the proper file extension.

Log file It contains general messages. This file has name extension `log`. The file can be written directly by an individual job, or by Jobman from the standard out of an individual executable. The job success string should be written into this file. Missing the success string indicates a failed job execution unless a configuration specifies otherwise.

Error file The error messages go to this file. It has name extension `err`. Unless specified otherwise in configuration file, nonempty error file indicates a failed job execution.

Parameter file The job parameters can be output into this file. It has file name extension `par`. This file is read by Jobman to extract into a list of string. Jobman use the strings to take actions depending on the job type.

The path to the above three files can be derived from the string representation of the job, using appropriate file extension `.log`, `.err`, or `.par`. For example, the error file for the `multip` job mentioned above is

```
demo\logs\all\findp\sieve.3_4_16\composites\multip.3.err
```

In the job specification, the path of each of the three files can be accessed using variable `$log`, `$err`, and `$par` reps. If `$log` is present in the job specification, Jobman will not catch the `stdout` into the log file. Instead, it relies on the individual executable to log.

However, the executable in a job must be a stand-alone program. An interpreted command cannot be used for the executable in an `exec` job. For example, in a Linux system, the following line in `job_list.txt`

```
exec job1 my_command > $log
```

will not redirect the outputs of `my_command` to the log file when job1 is run, as one might imagine. The reason is that the executable called by Jobman is treated as a stand-alone program, not a shell interpreted command.

If a job is a node, it also has a folder containing further information about its child jobs.

3.2 Job Dependencies

Job dependencies are specified using `depd` job type:

```
depd <parent job> <condition 1> ... <condition n> <child job>
```

where `child job` is the job to carry out the intended calculations, and where `<condition j>` specifies a job that must be completed before `child job` can start. Each condition job `<condition j>` must write the path to the job that `<child job>` depends on into its parameter file. Jobman will check the job specified in the parameter file has completed before it moves forward.

Another type of job dependency is `locks` that ensures only one job that has acquired the locks can be run. The syntax is

```
lock <parent job> <lock 1> ... <lock n> <child job>
```

The `<child job>` can be started only if it has acquired all the locks listed above. Each job `<lock j>` outputs a string, representing a lock, into its parameter file. Note that lock is simply a string, with all variables substituted. Any two jobs cannot acquire a lock at the same time.

The last type of job dependency is like program flow control:

```
case <parent job> <parameter job> <child 1> ... <child n>
```

The parameter job runs first. It writes the job name of one of the children `<child 1>`, ..., and `<child n>` to run into the parameter file. Jobman will then run the selected child job with other child jobs ignored.

Jobman contains a special leaf job type to facilitate the implementations of the above conditions and locks.

```
parm <job name> <file path> <parameter string>
```

This leaf job writes parameter string into file path. The above file path or parameter string for a `parm` job can have variables of job parameters that will be substituted by Jobman at runtime. The file path and the parameter string can be both empty. In this case, the job is a dummy job, like a place holder not doing anything.

For example, consider the following job specifications:

```
pfrk parent num child
rnge num 1 100
lock child wrt_lck write
exec write <my implementation of write>
parm wrt_lck $par mylock
```

It creates a lock since each `write` must acquire the string `mylock`. On the other hand, if we replace the last line with

```
parm wrt_lck $par mylock$child
```

Then job `/parent/child.1/write` needs to acquire lock `mylock1`; ...; Job `/parent/child.100/write` needs to acquire lock `mylock100`. The lock in this example is superfluous.

3.3 Maximizing Parellels: Demo 4

Demo 3 still has not fully utilized multiprocessing capability. When it tries to find out all multiples in $(2^{2^{j-1}}, 2^{2^j}]$, it must wait all the `multip` jobs to be finished until it moves to the next interval. If the last such job takes extra long time to complete, other processors will be idle. Can we further parallelize the job executions?

This can be achieved by indexing child jobs by a pair of integers (j, k). A child job (j, k) would compute the numbers in $(2^{2^{k-1}}, 2^{2^k}]$ that are multiples of primes in $(2^{2^{j-1}}, 2^{2^j}]$. All these child jobs can be parallelized if we require job (j, k) can be started only after (i, j) have completed for all $i < j$. We can express this requirement using `depd` job type. The job configuration file can be written as follows.

```
 1  defn jv java -cp work/classes
 2  defn outdir work/out
 3  loop all setup find_p
 4  exec setup $jv demo.Setup $outdir
 5  pfrk find_p intv_p_par intv_p
 6  exec intv_p_par $jv demo.Intervals 500 $par
 7  case intv_p p_sel p1 p2
 8  exec p_sel $jv demo.Select $intv_p[1] $par
 9  parm p1 $outdir/1/primes.txt 2
10  loop p2 multp primes
11  pfrk multp m1_par m1
12  exec m1_par $jv demo.Intervals $intv_p[2] $par
13  depd m1 prev_p m2
14  parm prev_p $par all/find_p/intv_p.$m1[1]_$m1[2]_$m1[3]
15  pfrk m2 m3_par m3
16  exec m3_par $jv demo.Copy $outdir/$m1[1]/primes.txt $par
17  exec m3 $jv demo.Multip $intv_p[2] $intv_p[3] $m3 $outdir/$intv_p[1]/$m3[].txt
18  exec primes $jv demo.Primes $intv_p[2] $intv_p[3] $outdir/$intv_p[1]
```

The logic may be a little bit twisted. But as you see in the outputs of Demo 4, it works.

4 Jobman Input Files

4.1 Configuration Files

So far we have only focused on the job configuration file `job_list.txt`. Other configuration files can also control Jobman execution flow and performance. However, none of the configuration files in this section is mandatory. Nor is any configuration variables in any of the file. If any configuration variable is not present, a default value is assumed. All the configuration files discussed below are in directory `cfg`.

File opt_dynamic.txt

This file contains user preferences that can be dynamically reloaded while Jobman is running. The file contains the following user preferences:

`parallels` The maximum number of parallel processes allowed. This number excludes the Jobman process. The default value is 4.

`sleep_time` Time in milliseconds for Jobman to sleep before it checks the status of all running jobs. The default value is 100.

`between_invokes` Time in milliseconds between the invocations of two executables. The default value is `sleep_time`.

`queue_length` The maximum queue length for each job name. A job may be placed into a queue due to limited number of processes or dependency conditions. This parameter controls the number of jobs (with the same name) constructed but not running. The default is `parallels`.

`enable_debug` Should extra debug info be dumped into the log file `jobman.log`? Any value other than `true` (case insensitive) is treated as false. The default value is `false`.

File opt_static.txt

This file contains user preferences that cannot be dynamically reloaded while Jobman is running. The change of the preference will be effective only after Jobman is restarted.

`success` The string that will be written at end of each job's log file to indicate the successful completion. It must have at least 10 characters. Make sure this string will not appear in the normal log messages not intended for job status. The default value is `JOBMAN_STEP_COMPLETED`.

`lock_file` It is the path to a lock file to prevent running two instances of this program. The default value is `jobman.lck`.

`resume_last` If the program has been run before, should we start over, or resume from where we left in the last run. Any value other than `true` (case insensitive) is treated as false. The default value is `false`. If this value if true, Jobman will check the log files for each job. If it was completely in the previous run, the job will be skipped in this run.

File job_exec.txt

This file gives configurations specific to individual `exec` jobs. Each line in the file specifies one job with format

```
<job name> <job rank> <parallels> <check log> <check err> <success string>
```

Each word since the second one in the line gives a preference. At least first two words must be present in a line. If any preference is missing, so must the preferences after it.

The first word is the `exec` job name.

The second word is an integer gives the rank of the job. By default, all `exec` jobs have rank 0. You can change the job priority here. A higher rank means higher priority. So a negative rank will have lower priority than a normal job. Once the maximum number of parallel processes has been reached, `exec` jobs are placed into the queue to be executed. Once a free process is available, the higher ranked job will be executed first. Jobs of the same rank will be executed according to the arrival order.

The third word in a line is the maximum number of parallel processes used by this job name. It is an integer. A negative value means there is no limit on number of processes used by this job name. The zero value means the limit follows the general default limit, i.e., `parallels` in `opt_dynamic.txt`.

The fourth word is either `true` or `false`. If check log is `true`, the log file will be searched for success string as the job completion indicator. Otherwise, the job log file will be ignored. Similarly, the fifth word specifies if the job error file should be checked.

The remaining of the line is the success string.

If a preference is missing in the line, the default value will be used.

File out_ignore.txt

This file specifies the messages to be ignored in error files. If a line in an error file contains a line in file `out_ignore.txt`, the line will be ignored. This means, this line will not be considered as an error message. This file is used when an executable is know to throw fake error messages (for example, into stderr).

A program that we cannot modify (e.g., from a third-party) may write non-critical messages into standard error channel. To ignore certain error messages, put error strings, one in a line, into this file. An error line containing a line from this file will not be treated as an error and hence not affect the status of the execution.

An empty job error file will be deleted by Jobman. A non-empty job error file, though possibly containing only non-critical messages, will not be deleted.

4.2 Control Files

While Jobman is running, we may want to tell Jobman to take certain action. This is achieved by placing a file of particular name into directory `cfg`.

To pause the program, create a new file `pause.txt` in `cfg`. Jobman will not move forward once this file is present. However, the executables that already started in separate processes will continue. The program will resume after the `pause.txt` file is deleted. Once the program resumes, configuration files `opt_dynamic.txt`, `job_list.txt`, and `job_exec.txt` will be re-fetched if they exist. The corresponding default values are used if a file does not exist.

To let the Jobman re-read files `opt_dynamic.txt`, `job_list.txt`, and `job_exec.txt` while keep it running, create a file `refresh.txt` in directory `cfg`. This file will be deleted by Jobman once the configuration files are re-read.

You can use the pause or refresh files to dynamically alter Jobman execution path when it is running by editing the job configuration files. However, the jobs already completed or running will not be altered and are not subject to the new configurations.

To kill the Jobman program, create a new file `kill.txt` in directory `cfg`. The program will try to kill all the running jobs and exit, without any cleanup for the individual executables.

To dump the current program state, create a new file `dump.txt` in directory `cfg`. The lists of finished jobs, running jobs, and other information will be dumped into the main log file. This may be useful for analysis.

The contents of the above four files `pause.txt`, `refresh.txt`, `kill.txt`, and `dump.txt` are not important. Only the file names matter. You can simply create an empty file to achieve the effect.

Any exception occurred in the program will cause the Jobman stop running. After the problem is addressed, you can resume the run, if `resume_last` is set to true in `opt_static.txt`.

4.3 Demo 5: Factorization

This demo gives an example on certain job types not previously covered, and on how to use configuration files.

Our goal is to find out all possible ways to write 36 as the product of two positive integers. This means find all $0 < x \leq y$ such that $xy = 36$. You may factor any positive integer by modifying the value of `testnum` in the job specifications below.

We use a brutal force method to do the job. We let y runs through from 1 to 36. And let x runs through from 1 to y. Check if xy is 36. If so, then we log the two numbers into `out/factors.txt`. Also, if $xy \geq 36$, we stop the x loop. The job configuration is given below. (Line numbers are not part of the configuration.)

```
1   defn jv java -cp work/classes
2   defn outdir work/out
```

```
 3   defn testnum 36
 4   loop all setup work
 5   exec setup $jv demo.Setup $outdir
 6   pfrk work first batch
 7   rnge first $par 1 $testnum
 8   plop batch second operation
 9   rnge second $par 1 $batch
10   case operation prod null save over
11   exec prod $jv demo.Product $batch $operation $testnum $log $par
12   parm null
13   parm over $log JOBMAN_STEP_BREAK
14   lock save save_lock dummy append
15   parm save_lock $par writelock
16   parm dummy $par $batch $operation
17   exec append $jv demo.SaveProd $batch $operation $outdir/factor.txt
```

Line 10 define a `case` job. The test job `prod` is an executable as defined on line 11. It compares 36 against the product of the two parameters from its ancestry jobs `batch` and `operation`. If 36 is larger, then it outputs string `null` into the parameter file. If they are equal, it outputs string `save`. If 36 is smaller, it outputs string `over`. The next child job to execute on line 10 is what `prod` writes into the parameter file `$par`.

Also observe that both `prod` on line 11 and `over` on line 13 will write the job success string directly into their log files. This is evidenced by including `$log` as their arguments. Jobman will not capture stdout for the two jobs.

We would like to save the results into a single file. This is done by job `append` on line 17. To avoid a possible situation when multiple processes write into the same file, we need to place a lock. That is why we define a `lock` job on line 14. For illustration, we require job `append` to obtain two locks before processing. The second one `dummy` is defined on line 16. As each `exec` job has a unique pair `$batch` and `$operation`, the lock `dummy` is superfluous.

To run Demo 5, copy files `opt_dynamic.txt`, `opt_static.txt`, and `out_ignore.txt` from `examples` into `demo/cfg`. Copy file `examples/job_list_5.txt` to `demo/cfg/job_list.txt`; and `examples/job_exec_5.txt` to `demo/cfg/job_exec.txt`.

You may want to modify the variable settings. For example, the job success string from individual executables in demos is from `demo/work/config.txt`. Any change to the string for entry `success_string` will result in a failed run. You may edit `success` in `opt_static.txt`, or the success string for each individual jobs in `job_exec.txt`, to match the outputs from individual executables.

5 Jobman Summary

5.1 Using Jobman

Jobman can be downloaded at

`http://sourceforge.net/projects/jobman/`

To use Jobman, you need to have Java Runtime Environment 8 or above installed. Please see section 1.4 for more details.

You only need the Java jar file `dist/jobman.jar` to use Jobman. Place it somewhere. Also choose your working directory. Within it, you need to have a subdirectory `cfg`. You need to write your own `job_list.txt` in that subdirectory. All the executables mentioned in that job list file must be implemented by you as you are the person knowing the business logic. Jobman just runs your executables according

to the flows you specify in the job list file. You may want to have other configuration files to have a finer control over job priorities, maximum processes, etc. You can then start running Jobman by executing

```
java -jar <path to jobman.tar> <the topmost job in your job\_list.txt>
```

To compile Jobman from source code, you need to have Java Development Kit or equivalent, for Java 1.8. The source code for Jobman and the demos is in directory `src`. An ant build script is `build.xml`.

5.2 Job Types

Jobman is driven by job configurations, which specifies how to piece various executables into a multi-processing program. In a sense, the job configuration itself is a computing language, with flow control commands. A job `task` is defined by

```
<type> <task> <specifications>
```

where `type` is one of the following job types whose syntax is summarized in the following table. The syntax specifies a line in file `job_list.txt`.

Type	Syntax of Each Line
defn	`defn <task> <contents>` Define variable `<task>` to be the string `<contents>`.
loop	`loop <task> <child_1> ... <child_n>` The task consists of looping over child jobs `<child_1>`, ..., `<child_n>` in sequence.
fork	`fork <task> <child_1> ... <child_n>` The task consists of running child jobs `<child_1>`, ..., `<child_n>` in parallel.
plop	`plop <task> <param> <child>` The task runs child job `<child>` in sequence with parameters supplied by job `<param>`. Each descendant of `<child>` can access the parameter value via variable `$<child>`.
pfrk	`pfrk <task> <param> <child>` The task runs child job `<child>` in parallel with parameters supplied by job `<param>`. Each descendant of `<child>` can access the parameter value via variable `$<child>`.
depd	`depd <task> <cond_1> ... <cond_n> <child>` The task runs child job `<child>` only if all the jobs specified by `<cond_1>`, ..., `<cond_n>` have completed.
lock	`lock <task> <cond_1> ... <cond_n> <child>` The task runs child job `<child>` only if all the locks specified by `<cond_1>`, ..., `<cond_n>` have been acquired.
case	`case <task> <select> <child_1> ... <child_n>` The task runs only one of the child jobs `<child_1>`, ..., `<child_n>` specified by job `<select>`.
exec	`exec <task> <command>` The task is an executable to run `<command>`. The command may have variables to be substituted at run time as the arguments. An `exec` job always runs in a separate process. And this is the only job type running in a separate process.
rnge	`rnge <task> <file> <start> <end>` Write integers between `<start>` and `<end>` inclusive into file `<file>`. The file path may contain variables to be substituted at run time.
parm	`parm <task> <file> <contents>` Write `<contents>` into file `<file>`. Both may contain variables to be substituted at run time.

The first type defines a variable. The last three are leafs. The other seven jobs are nodes.

A variable mentioned in the above table can be either defined by `defn`, or a parameter carried from `plop` or `pfrk` job.

5.3 Other Configurations

The following table summarizes the contents of each line in other configuration files.

File	Contents of Each Line
`opt_dynamic.txt`	`parallels` Number of parallel processes. `sleep_time` Time in milliseconds before update the status of all running jobs. `between_invokes` Time in milliseconds between the invocations of two executables. `queue_length` The maximum queue length for each job name.
`opt_static.txt`	`resume_last` Specifies if resume from where the last run left. `success` The string to indicate the successful completion of a job. `lock_file` The lock file to prevent running Jobman in parallel.
`job_exec.txt`	`<name> <rank> <parallels> <check_log> <check_err> <success>` A line customize an individual `exec` job. Refer to section 4.1 for details.
`out_ignore.txt`	String not to be treated as an error message in the error file.

5.4 Job Status

Jobman uses job status to decide a certain action to be performed for each job. Job status is also shown in `logs/jobman.csv`.

Status	Description
`STARTED`	The job has been constructed and ready to run.
`QUEUED`	A leaf job is placed into the queue since no processes are available or certain conditions (dependencies or locks) have not been satisfied. A node job has status `QUEUED` if all its child jobs are `QUEUED` or its dependencies or locks have not been cleared.
`RUNNING`	A leaf job is being executed. A node job is `RUNNING` if one of its children is.
`SUCCEED`	The job has completed successfully.
`FAILED`	The job failed.
`BREAK`	The job has completed. It further requests its closest parametrized ancestor stop starting new child jobs.
`IGNORED`	The job's status is ignored since one of its siblings requests `BREAK`.
`SKIPPED`	In a resumed run, this job is skipped since it was completed in the previous run.
`KILLED`	The job has been killed.

The last six status values are terminal status. An executable job is marked `FAILED` if any of the following is true.

- The executable return a nonzero status.
- The job err file is not empty and `job_exec.txt` does not specify to ignore the err file for the job.
- The job log file does not contain the success string and `job_exec.txt` does not specify to ignore the log file for the job.

You may let Jobman to ignore both log file and err file for an executable, so that the job is always completely successfully. You then have another job following it to check and report the status. This approach is useful when the primary computation job (the first job) is not easily customized to confirm the status right away.

5.5 Job Communications

Individual job `task` communicates to Jobman via the text files. We assume `task` is the job name in discussion within this section.

Each job has a unique string representation, which is its job path. It is the concatenation of the job names and the parameters of all its ancestors, separated by special characters. See section 3.1 for details.

	Who	Description
Log file	`exec`	Individual program writes its terminal status into `task.log`, or into stdout which is captured by Jobman into `task.log`.
	Others	Jobman writes the job status into `task.log`.
	All	The path to the log file can be accessed via `$log`.
	Jobman	If `check_log` is false in `job_exec.txt`, Jobman ignores `task.log`.
Err file	`exec`	Individual program writes errors into `task.err`. or into stderr which is captured by Jobman into `task.err`.
	Others	Never writes into the error file.
	All	The path to the err file can be accessed via `$err`.
	Jobman	If `check_err` is false in `job_exec.txt`, Jobman ignores `task.err`.
Par file	Leaf	`(plop|pfrk) task <child>` `task` writes list of parameters for `<child>` into `task.par`, one per line. Jobman runs `<child>` once for each parameter.
		`case task <child_1> ... <child_n>` `task` write the selection among child jobs into `task.par`. Jobman runs the selected child job.
		`depd <parent> <task_1> ... <task> ... <task_n> <child>` `task` writes a job path into `task.par`. Jobman runs `<child>` only after it confirms the job given by the path in `task.par` has completed.
		`lock <parent> <task_1> ... <task> ... <task_n> <child>` `task` writes a string into `task.par`. Jobman runs `<child>` only after it confirms the lock string in `task.par` has been acquired.
Return	`exec`	`task` returns the execution status to Jobman.
	Other	Not applicable.
	Jobman	Determine if `task` has failed or not by the return value of an executable.

The output of Jobman is a hierarchy of file structure in log root directory `logs`. A job maybe associated with up to three files and one directory: log file `<s>.log` for any job, parameter file `<s>.par` for a leaf job, error file `<s>.err` for a leaf job, and the directory `<s>` for a node job. Here `<s>` is the file path formed by concatenating the log root directory and the job string representation.

There are also two special files in the log root directory. File `jobman.log` records the general messages from Jobman, with more details if `enable_debug` is set `true` in `cfg/opt_dynamic.txt`. File `jobman.csv` records the status changes for each job in a tabular format. So we have an overall view on the job running progress. The following fields are reported in the csv file.

Field	Description
`seq`	The sequence number of this row in this run session.
`time`	The time when this row was logged.
`loop`	Jobman loop number. Jobman walks through the hierarchy of jobs. Starting from the topmost job, it constructs new jobs if possible and moves each constructed jobs forward. After it completes one cycle, it returns to the topmost job. Each such a cycle is a loop.
`level`	The level of the job in the row. The topmost job has level 1. The level of a child job is one more than its parent node job.
`type`	This is the job type defined in section 5.2.
`status`	This is the job status defined in section 5.4.
`path`	This is the string representation of the job.
`run`	This is the session number of this run. The first run of Jobman has session 1. If `resume_last` in `opt_static.txt` is set `true`, then each subsequent run will increase the session number. Note that the `seq` always starts from 1 for each run.

5.6 Jobman Controls

A user can monitor and control Jobman by creating a file with the specific name in directory `cfg`.

File	Description
`pause.txt`	Pause Jobman execution.
`refresh.txt`	Request Jobman to reload configuration files.
`dump.txt`	Request Jobman to dump the current state.
`kill.txt`	Stop Jobman.

5.7 Implementation Overview

Jobman is implemented in Java without other libraries. Version 1.0 requires Java 1.8.

Jobman controls program flow using the relationships between jobs. A parent job can contain child jobs. A job without a child is a leaf job. Except the convenience job types `rnge` and `parm`, A leaf job is an executable to be run in a separate process. So all the jobs to be run is like a hierarchical tree of jobs, with possible further dependencies among branches and leafs. When Jobman executes, it starts from the top most job, and walks through certain jobs in the tree. In this process, it collects all the executables (`exec` jobs) to be run. At the end of a walk, Jobman starts executing the collected leaf jobs in separate processes. In the meantime, it performs another round of walk of the tree of jobs. This process continues until all jobs are completed or some job fails.

In summary, except new processes started for `exec` jobs, Jobman is actually a single processed program. It uses of Java collections to hold job progress data, without using synchronized containers.

6 Executables Used in Demos

The executables in the demos in this document are all written in Java. These can also be implemented by other languages. This section describes the arguments, inputs, outputs, and functionalities of each executable in case you don't want to look into the demo Java code. It is always a good idea to run the demo first. Check the log files and output files to see what it is doing.

To make demos more realistic, each executable pauses running (i.e., put execution process into sleep) for some random time. The lower and upper limits of the random time can be specified in file `demo/work/config.txt`. Also the success string output from `exec` jobs can be customized there.

6.1 Copy

This executable is used in Demo 4. The command is:

```
java -cp work/class demo.Copy <from> <to>
```

It accepts two arguments `<from>` and `<to>`, both being file paths. The executable simply copy `<from>` to `<to>`.

6.2 Interval

This executable is used in Demos 3 and 4. The command is:

```
java -cp work/class demo.Interval <max> <file>
```

The executable partitions interval (1, `<max>`] into subintervals, with results written int `<file>`. Each line has three numbers: index, lower boundary, and upper boundary. Mathematically, line j is a triple of integers

$$j, \qquad 2^{2^{j-1}}, \qquad \min(2^{2^j}, M)$$

where M is the first argument `<max>`. For example, the output of

```
java -cp work/classes demo.Intervals 500 E:\dev\jobman\demo\logs\all\findp\intervals.par
```

will output the following contents into the file given by the last argument:

```
1 1 2
2 2 4
3 4 16
4 16 256
5 256 500
```

6.3 ListFactors

This executable is used in Demo 3. The command is:

```
java -cp work/class demo.ListFactors <dir> <index> <max> <file>
```

Go to the file `primes.txt` produced from the previous step. List all primes no more than the square root of `<max>` and write them into `<file>`. For example,

```
java -cp work/classes demo.ListFactors work/out 5 500 all\findp\sieve.5_256_500\composites\pfac.par
```

reads primes in file `work/out/4/primes_all.txt` and writes those primes no more than $\sqrt{500}$ into the last argument.

6.4 Merge

This executable is used in Demo 3. The command is:

```
java -cp work/class demo.Merge <step> <outdir>
```

Read all the prime files `<outdir>/<j>/primes.txt`, where `<j>` is the step number up to `<step>`, and merge the contents into file `<outdir>/<step>/all_primes.txt`.

6.5 Multip

This executable is used in Demos 1, 2, 3, and 4. The command is:

```
java -cp work/class demo.Multip <min> <max> <factor> <file>
```

Compute all nontrivial multiples of `<factor>` between `<min>` and `<max>`. Write the results into file `<file>`, one number per line.

6.6 Primes

This executable is used in Demos 1, 2, 3, and 4. The command is:

```
java -cp work/class demo.Primes <lower> <upper> <dir>
```

Assume `<dir>` already has a list of files containing composite numbers. This executable computes the primes between `<lower>` (exclusive) and `<upper>` (inclusively) by removing the integers in the those composite files. Write results into `<dir>/primes.txt`, one number per line.

6.7 Product

This executable is used in Demo 5. The command is:

```
java -cp work/class demo.Product <num1> <num2> <check> <logfile> <parfile>
```

Multiply `<num1>` and `<num2>`. Compare the product with `<check>`. Write string into file `<parfile>` depending on situation: If the product is less than `<check>`, write `null`. If they are equal, write string `save`. If the product is larger, write `over`. Upon completion, write the success string into `<logfile>`.

6.8 SaveProd

This executable is used in Demo 5. The command is:

```
java -cp work/class demo.SaveProd <num1> <num2> <file>
```

Append line

```
<prod> = <num1> X <num2>
```

into `file`, where `<prod>` is the product of two numbers.

6.9 Select

This executable is used in Demo 4. The command is:

```
java -cp work/class demo.Select <number> <file>
```

If argument `<number>` equals 1, write `p1` into `<file>`. Otherwise, write `p2` into `<file>`.

6.10 Setup

This executable is used in all demos. The command is:

```
java -cp work/class demo.Setup <dir>
```

Accept one argument as the directory path. If it does not exist, create. Otherwise, empty the directory.

7 Legal

```
Computation Job Management (Jobman)
    Managing the executions of programs like Lego blocks.
    Version 1.0
    http://sourceforge.net/projects/jobman
    Copyright @ 2006-2015 Heng Sun <sunheng at hotmail dot com>
```