The Stata Journal (*yyyy*) *vv*, Number *ii*, pp. 1–17

Parallel: Stata module for parallel computing

George G. Vega Yon
University of Southern California
Los Angeles, CA
vegayon@usc.edu

Brian Quistorff
Microsoft Research
Redmond, WA
Brian.Quistorff@microsoft.com

Abstract. The `parallel` package allows parallel processing of tasks that can logically be separated into parallel tasks that have no dependencies between them. This allows all flavors of Stata to take advantage of multiprocessor machines. Even Stata/MP users can benefit as many user-written programs are not automatically parallelized but could be so under our framework.

Keywords: st0001, parallel computing, simulations, high performance computing

1 Parallel computing

Most computers these days have multiple "processors" such as with multiple cores per CPU or multiple threads per core (e.g. Intel's Hyper-Threading Technology). Stata currently uses only one processor except for Stata/MP with certain built-in commands[1]. Many other tasks, however, are logically easy to parallelize. These tasks, called "embarassingly parallel", are ones where there are no dependencies (or need for communication) between the parallel tasks. We provide here the package `parallel`, to parallelize these tasks.[2]

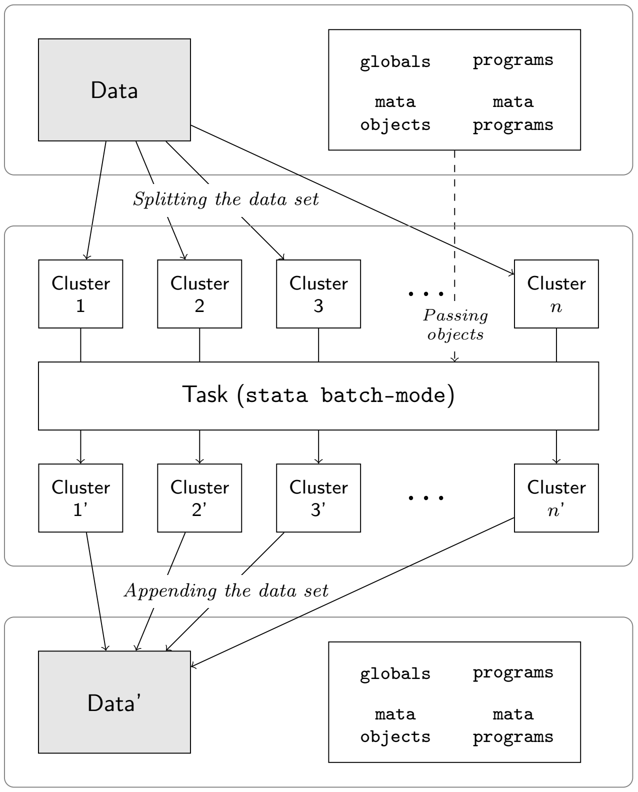
The primary process used is to invoke `parallel` with a command (or do-file) across N parallel "clusters" is as follows,

1. `parallel` splits the dataset into N pieces.
2. `parallel` starts N copies of Stata. Those are referred to as child processes while the original is the parent. In each, one of the pieces of the split dataset is loaded, the command is executed, and the resultant data is saved.
3. `parallel` waits for the child processes to finish and then aggregates the resultant datasets.

This is diagrammed in figure 1. Notice that this is a setting with "distributed" rather than shared memory between the child processes.

Stata/MP is a version where internal routines are able to take advantage of multiple processors on a machine. `parallel` allows this for generic commands which both

1. For a list of commands explicitly parallelized see the Stata/MP Performance Report (Stata Press (2010)).
2. More "fine-grained" parallelism, where tasks need to communicate frequently, could be handled but there is no direct support.



Starting (current) `stata` instance loaded with data plus user defined `globals`, `programs`, `mata objects` and `mata programs`

A new `stata` instance (batch-mode) for every data-clusters. Programs, globals and mata objects/programs are passed to them.

The same algorithm (task) is simultaneously applied over the data-clusters.

After every instance stops, the data-clusters are appended into one.

Ending (resulting) `stata` instance loaded with the new data.

User defined `globals`, `programs`, `mata objects` and `mata programs` remind unchanged.

Figure 1: How `parallel` works

expands the set of possible parallelizations and allows this for single-threaded flavors of the program.

This module is similar to R's `parallel` package (superseding the `snow` package) and Matlab's parallel toolbox.

The document follows, after this introduction, section 2 discusses the details of execution, section 3 provides some usage examples, and section 5 discusses and concludes.

2 A Stata module for parallel computing

In this section we discuss the syntax of the `parallel` subcommands, technical details of execution, and results returned from the commands.

2.1 Syntax and options

A typical program will use separate `parallel` subcommands for initialization, parallel task execution, and finally for cleanup. There are diagnostic tool commands also available.

2.1.1 Initialization

To initialize the parallel setup use the `setclusters` subcommand

parallel setclusters # [, `force` `statapath(`*path*`)` `includefile(`*filename*`)`]

The main usage of this command is to set the number of child processes to launch when parallelizing later tasks. Options:

- `force` - To prevent slowdowns on systems with few available computing resources, there is a soft limit on eight clusters. Use this option to override the limit.
- `statapath(`*stata_path*`)`- By default, `parallel` tries to automatically identify Stata's executable path. By using this option you will override this and force `parallel` to use a specific path to the executable.
- `includefile(`*filename*`)` - This file will be included before `parallel` commands are executed. The target purpose for this is to allow one to copy over preferences that `parallel` does not copy automatically (see section 2.3).

2.1.2 Parallel task execution

The follow are subcommands that execute tasks in parallel.

parallel [, `by(`*varlist*`)` `force` `nodata` `setparallelid(`*pll_ id*`)` *execution_ options*] : *command*

parallel do *dofile* [, `by(`*varlist*`)` `force` `nodata` `setparallelid(`*pll_ id*`)` *execution_ options*] parallel bs [, `expression(`*exp_ list*`)` *execution_ options bs_ options*] [: *command*]

parallel sim [, `expression(`*exp_ list*`)` *execution_ options sim_ options*] [: *command*]

parallel append [*files(s)*], do(*command/dofile*) [`in(`*in*`)` `if(`*if*`)` `expression(`*expand expression*`)` *execution_ options*]

The : (prefix) notation for `parallel` and the `do` subcommand are the main subcommands while the others are helper utilities. Their usage is shown in sections 3.

`execution_options`:

- `keep` - Keeps auxiliary files generated by `parallel`.
- `keeplast` - Keeps auxiliary files and remove those last saved during the current session.
- `noglobal` - Avoid passing current session's global macros to the clusters.
- `programs(`*`namelist`*`)` - A list of programs to be passed to each cluster. To do this, `parallel` needs to echo the contents of those programs to the output window. If `parallel` is being run from inside an ado (say `my_cmd.ado`) and you need to access local subroutines (other programs defined in the ado beside the primary `my_cmd`), then you must pass their names in this option as `my_cmd.local_subroutine_name` for them to be accessible.
- `mata` - If the algorithm needs to use mata objects, this option allows to pass to each cluster every mata object loaded in the current session (including functions). Note that when mata objects are loaded into the child processes they will have different locations and therefore pointers may no longer be accurate.
- `randtype(current|datetime|random.org)` - Tells `parallel` whether to use the current seed (default), the current datetime or random.org API to generate the seeds for each clusters.
- `seeds(`*`numlist`*`)` - With this option the user can pass an specific seed to be used within each cluster.
- `processors(`*`integer`*`)` - If running on StataMP, sets the number of processors each cluster should use. Default value is 0 (do nothing).
- `timeout(`*`integer`*`)` - If a cluster hasn't started, how much time in seconds does `parallel` has to wait until assume that there was a connection error and thus the child process (cluster) won't start. Default value is 60.
- `outputopts(`*`namelist`*`)` - allows generic file-based aggregation (appending). A sequential call such as
  `my_prog, output1(outputfile.dta)`
  can be converted to

`parallel, outputopts(output1): my_prog, output1(outputfile.dta)` `parallel` will execute each child process with its own file passed to `output1` and at the end, append them all and save it to `outputfile.dta`.

- *`dofile`*/`command` - Task to run in parallel. Note that while the prefix notation can handle parameters passed to the command, `parallel do` can not handle parameters passed to the do file.

Main `parallel` subcommand options:

- `by(`*`varlist`*`)` - Tells the command through which observations the current dataset can be divided, avoiding stories (panel) splitting over two or more clusters. [3]
- `force` - When using `by()`, `parallel` checks whether if the dataset is properly sorted. By using `force` the command skips this check.
- `nodata` - Tells `parallel` not to use loaded data and thus not to try splitting or appending anything.
- `setparallelid(`*`pll_id`*`)` - Forces `parallel` to use an specific id.

Bootstrap options

- `expression(`*`exp_list`*`)` - An expression list to be passed to the native `bootstrap` command.
- *`bs_options`* - Further options to be passed to the native `bootstrap` command, including the optional `reps()` parameter.

Simulation options:

- `expression(`*`exp_list`*`)` - An exp_list to be passed to the native `simulate` command.
- *`sim_options`* - Further options to be passed to the native `simulate` command, including the required `reps()` parameter.

Append options:

- *`file(s)`* - Explicit list of files to process.
- `expression(`*`expand expression`*`)` - Expression representing file names in the form of `"%fmts, numlist1 [, numlist2 [, ...]]"`
- `in(`*`in`*`)/if(`*`if`*`)` - Opens the file using `if` and `in` accordingly.

3. The semantics for `by` are not the same as for Stata. When Stata implements `by`, the command that is run will only see a section of the data where the `by`-variables are the same. `parallel`'s semantics are that no observations with the same `by`-values will be in different clusters. It pools together combinations when there are fewer clusters than `by`-var combinations. If you need Stata-style semantics, the solution is to add `by` in the subcommand. For example, `parallel, by(byvar): by byvar: egen x_max = max(x)`.

2.1.3 Cleanup

Log files from `parallel` execution are saved so that they can be inspected by the user. Use the `clean` subcommand to remove these and any other ancillary files that have been saved:

parallel clean [, `event(`*pll_id*`)` `all` `force`]

Options:

- `event(`*`pll_id`*`)` - Specifies which executed (and stored) event's files should be removed.
- `all` - Tells `parallel` to remove every remnant auxiliary files generated by it in the current directory.
- `force` - Forces the command to remove (apparently) in-use auxiliary files. Otherwise these will not get deleted.

2.1.4 Diagnostic tools

Additionally there are some diagnostic tools,

parallel version

This command returns the version both to the screen and programmatically.

parallel printlog [#] [, `event(`*pll_id*`)`]

parallel viewlog [#] [, `event(`*pll_id*`)`]

These commands allow you to view logs of the child processes. The initial part of the log file will be from commands generated by `parallel` for setting up the child process (loading data, global macros, settings, etc.). The final part of the log file is where the users task is. Options:

- # - specifies which cluster number of an event to display (default is 1).
- `event(`*`pll_id`*`)`- Specifies which event's log files should be displayed

2.2 Saved Results

The primary result of `parallel` is to return a transformed dataset. In addition `parallel` returns the following values:

Scalars

`r(pll_n)`	Number of parallel clusters last used.
`r(pll_t_fini)`	Time took to appending and cleaning.
`r(pll_t_calc)`	Time took to complete the parallel job.

`r(pll_t_setu)`	Time took to setup (before the parallelization) and to finish the job (after the parallelization) .
`r(pll_errs)`	Number of clusters which stopped with an error.
Global Macros	
`LAST_PLL_DIR`	A copy of `r(pll_dir)`.
`LAST_PLL_N`	A copy of `r(pll_n)`.
`LAST_PLL_ID`	A copy of `r(pll_id)`.
`PLL_LASTRNG`	Number of times that `parallel_randomid()` has been executed.
`PLL_LASTRNG`	Internal usage.

`parallel version` saves

Macros	
`r(pll_vers)`	Current version of the module.

`parallel bs` and `parallel sim` save

Scalars	
`e(pll)`	1.

2.3 Technical Details

`parallel` does not change the RNG state (even if subcommands invoke randomization functions).

Log files from the runs are stored in `c(tmpdir)` so that they can be inspected by the user. The user will likely want to delete these periodically with `parallel clean, all`.

Given N clusters, within each cluster `parallel` creates the macros `pll_id` (equal for all the clusters) and `pll_instance` (ranging 1 up to N, equaling 1 inside the first cluster and N inside the last cluster), both as global and local macros. This allows the user setting different tasks/actions depending on the cluster. Also the global macro `PLL_CLUSTERS` (equal to N) is available within each cluster. Note the locals will obviously be not available inside of programs that you call from `parallel` (in prefix or do-file setup).

When launching child Stata processes, several settings are automatically copied over. These include the `PLUS` and `PERSONAL` sysdirs, the global `S_ADO`, the mlib search index, the `tempname`/`tempvar` state. If you would like child processes to start with additional setting changes then you should use the `includefile()` option.

Child processes are managed. If the task is stopped from the parent process then all child processes will be killed directly. The parent process can recover from both errors in the child Stata program and if child Stata processes are killed by the operating system.

Child processes are launched using the shell on MacOS and Unix/Linux machines. On Windows machines a compiled plugin launches the child processes using the Win32 API. Windows uses a different system as batch-mode windows will not execute shell commands and there is no console-only version of Stata so every launched process will annoyingly flash on the screen and steel your current application's focus.

When the stata_cmd or do-file saves results, as `parallel` runs Stata in batch mode, none of the results will be kept. This is also true for matrices, scalars, mata objects, returns, or whatever other object different from data. Programs can often be modified to aggregate data in the primary dataset or using secondary files (see the {it:outputopts} options).

Although `parallel` passes-through programs, macros and mata objects, in the current version it is not capable of doing the same with matrices or scalars.

If the number of tasks to be done is less than the number of clusters, `parallel` will temporarily reduce the number of clusters. This is reported in the global macro `LAST_PLL_N`.

2.4 Installation

Stable versions of `parallel` can be installed from the SSC archives. The latest development versions can be installed install the latest version using

```
. net install parallel , ///
    from(https://raw.github.com/gvegayon/parallel/master/)
. mata mata mlib index
```

If you are switching the source of the installation materials (e.g. is you move from SSC to GitHub versions), then be sure to uninstall the program explicitly before installing the new version.

3 Examples

In this section we discuss basic usage of the commands as some common use cases.

3.1 Subcommand examples

▷ Example Prefix

A minimal example of using `parallel` is

```
. sysuse auto, clear
(1978 Automobile Data)

. parallel setclusters 2
```

```
N Clusters: 2
Stata dir:  C:\Program Files (x86)\Stata14/StataMP-64.exe

. parallel: gen price2 = price*price
--------------------------------------------------------------------------------
Parallel Computing with Stata
Clusters    : 2
pll_id      : <unique ID>
Running at  : <pwd>
Randtype    : datetime
Waiting for the clusters to finish...
cluster 0001 has exited without error...
cluster 0002 has exited without error...
--------------------------------------------------------------------------------
Enter -parallel printlog #- to checkout logfiles.
--------------------------------------------------------------------------------
```

This example illustrates that many simple tasks can be parallelized. This particular task was not executed faster because it was run in parallel since parallel execution has its own overhead and the task was quite easy.

◁

The next example shows the usage of the `do` subcommand.

▷ **Example Do-file**

Suppose that we had the existing do-file

```
//----------------- make_polynomial.do ---------//
gen price2 = price*price
gen price3 = price2*price
gen price4 = price3*price
```

We can execute it either sequentially or

```
. parallel do make_polynomial.do
```

◁

▷ **Example Bootstrap**

A simple sequential bootstrap would

```
. sysuse auto, clear
. bs: reg price c.weig##c.weigh foreign rep
```

To parallelize

```
. sysuse auto, clear
. parallel setclusters 2
. parallel bs: reg price c.weig##c.weigh foreign rep
```

◁

▷ Example Simulation

Suppose we have the following simulation program.

```
program define lnsim, rclass
  version 14
  syntax [, obs(integer 1) mu(real 0) sigma(real 1) ]
  drop _all
  set obs `obs'
  tempvar z
  gen `z' = exp(rnormal(`mu',`sigma'))
  summarize `z'
  return scalar mean = r(mean)
  return scalar Var  = r(Var)
end
```

If we were to run it sequentially we'd use

```
. simulate mean=r(mean) var=r(Var), reps(10000): lnsim, obs(100)
```

to run it parallel we could instead use a very familiar syntax

```
. parallel sim, expr(mean=r(mean) var=r(Var)) reps(10000): ///
    lnsim, obs(100)
```

◁

▷ Example Append

Imagine we have several dta files named `income.dta` stored in a set of folders ranging "2008_01" up to "2012_12", this is, a total of 60 files ordered which may look something like this:

```
2008_01/income.dta
2008_02/income.dta
2008_03/income.dta
...more files...
2010_01/income.dta
2010_02/income.dta
2010_03/income.dta
...more files...
2012_10/income.dta
2012_11/income.dta
2012_12/income.dta
```

Now, imagine that for each and every one of those files we would like to execute the following program:

```
program def myprogram
  gen female = gender == "female"
  collapse (mean) income, by(female) fast
end
```

Instead of writing a `forval`/`foreach` loop (which would be the natural solution for this situation), `parallel append` allows us to smoothly solve this with the following command.

```
. parallel append, do(myprogram) prog(myprogram) ///
        e("%g_%02.0f/income.dta, 2008/2012, 1/12")
```

Where element by element, we are telling `parallel`:

(1) `do(myprogram)`: execute the command `myprogram`,

(2) `prog(myprogram)`: `myprogram` is a user written program that needs to passed to child clusters, and

(3) `e("%g_%02.0f/income.dta, 2008/2012, 1/12")`: this should process files "2008_01/income.dta" up to "2012_12/income.dta".

Besides of the simplicity of its syntax, the advantage of using `parallel append` lies in doing so in a parallel fashion, this is, instead of processing one file at a time, `parallel` manages to process these files in groups of as many files as clusters are set. Step-by-step, what this command does is:

1. Distribute groups of files across clusters

Once each cluster starts, for each dta file

2. Opens the file using [if] [in] accordingly to in and if options.

3. Executes the command/dofile specified by the user.

3. Stores the results in a temporary dta file.

Finally, once all the files have been processed

4. Appends all the resulting files into a single one.

◁

3.2 Monte Carlo Experiment

Maybe something more fancy such as a non-parametric test statistic based on simulations? Use the `parallel sim` prefix.

3.3 Parallelizing a loop

If a user has a loop where the processing in each loop are independent of each other and the output can be aggregated easily then it is easily transformed using `parallel`.

Suppose we want to parallelize a general loop

```
forval i=1/`num_total'{
  //work for i
}
```

We can transform this so that a setup that can be done either parallel or sequential.

```
local n_proc = XXX
save currdata.dta, replace
drop _all
set obs `num_total'
gen long i = _n
if `n_proc'>1 {
  parallel setclusters `n_proc'
  parallel: parfor_task
}
else {
  parfor_task
}

//-------------parfor_task.ado-------------//
program parfor_task
  local num_task = _N
  mkmat i, matrix(tasks_i)
  use currdata.dta, clear
  forval j=1/`=_N'{
    local i = tasks_i[`j',1]
    //work for i
  }
  //put output into main data
end
```

3.4 Consistency

For many tasks we will want to ensure that there is exact consistency between multiple runs of a program. Deterministic programs virtually ensure this. With random functions, a sequential program is usually made consistent by specifying a fixed random seed at the beginning of the program. If one is always using the same number of clusters then the same can be achieved by pre-specifying the seeds with the `seeds` options.

A similar notion of sequential consistency guarantees that results do not differ between sequential and parallel operations. Again, for deterministic programs this is straight-forward to check. If the program has a random component then more care must be taken. To do this, provide the seed for each repetition. Once we do that, we can build upon the previous example about loop (section 3.3) so that the tasks are split to the child processes and show how to collect the output.

▷ **Example Sequential consistency**

Here we do it with a custom bootstrap implementation

```
set seed 1337
sysuse auto, clear
parallel setclusters 2

cap program drop do_work
program do_work
  args main_data
  local num_rep = _N
  tempname tasks pfile
  mkmat n seed, matrix(`tasks')
  qui use "`main_data'", clear
  tempfile estimates
  postfile `pfile' long(n seed) float(b_mpg) using "`estimates'"
  forval i=1/`num_rep'{
    local seedi = `tasks'[`i',2]
    set seed `seedi'
    preserve
    bsample
    qui reg price mpg
    post `pfile' (`=`tasks'[`i',1]') (`seedi') (_b[mpg])
    restore
  }
  postclose `pfile'
  use "`estimates'", clear
end

tempfile maindata
save "`maindata'"
drop _all
gen long seed = .
qui set obs 99 //number of reps
replace seed = int((-1*`c(minlong)'-1)*runiform())
gen long n=_n
local final_seed = c(seed)
parallel, program(do_work): do_work "`maindata'"
mata: rseed(st_local("final_seed"))
```

```
sort n
```

The output be the same no matter the number of clusters or if the `do_work` is run without `parallel`.

◁

3.5 Parallelizing own command

A third-party Stata package developer with easily parallelizable tasks can easily write their packages to take advantage of `parallel` if it is installed. We recommend that packages not require `parallel` as users may be on machines with limited resources. The most common example would be wanting to parallelize an existing loop, so one can follow the examples of the parallel for loop or the sequential consistency example. You can put that secondary program in your original ado file (but then you have use myado.ado.subtask) or you can make a separate file (in which case you might want to prefix the name with an underscore as it is not likely used directly by users).

3.6 Benchmarks

Processing a file in parallel using the `parallel` package will like result in more memory (ie RAM) usage than processing the file sequentially. So to some extent one is trading memory capacity for processing capacity. Therefore, there is likely to be little benefit if a sequential setup would already utilize close to all of your system memory.

3.7 Debugging

The `parallel` command will issue an error if either it or one of its child processes encounters an error. The first step towards debugging this is to look at the log files (using, e.g., `parallel viewlog`). If this does not show enough information you may want to use to turn `trace` on in your execute task or output your own diagnostic information.

4 Benchmarks

In order to assess the speed-gains obtain when using `parallel`, we present what we think are the two most relevan uses of the module: bootstrapping and simulations. We compared the performance of running each routine in: serial fashion, using 2 clusters (cores), and using 4 clusters (cores). While the tasks over which we performed the comparisons are rather simple (and not particularly time consuming since all of them took less than a minute to complete), they are useful to ilustrate the benefits of using `parallel`.

It is important to keep in mind that, as we will see, the lack of perfectly linear speed-gains is due to the simplicity of the problem with respect to the time that takes to compute it in a serial fashion. On the other hand, overall, as the problem size–number of simulations, resamples, etc.–increases, the speed-gains do become perfectly linear.

The code used to perform the benchmarks and generate the figures and tables is available in the project's website.

4.1 Bootstrapping

In this first benchmark, we use the auto.dta data base shipped with Stata. After expanding each observation 10 times–so the size of the problem increases–we perform a bootstrap of a linear regression model as follows:

```
sysuse auto, clear
expand 10

// Parallel fashion
parallel bs, rep($size) nodots: regress mpg weight gear foreign

// Serial fashion
bs, rep($size) nodots: regress mpg weight gear foreign
```

For each combination of (Serial, 2 Cluster, 4 Cluster) methods and (1000; 2000; 4000) repetitions (problem size), we ran the problem 1000 times and recorded (total) average computing time. The results are presented in Figure 2 and table 4.

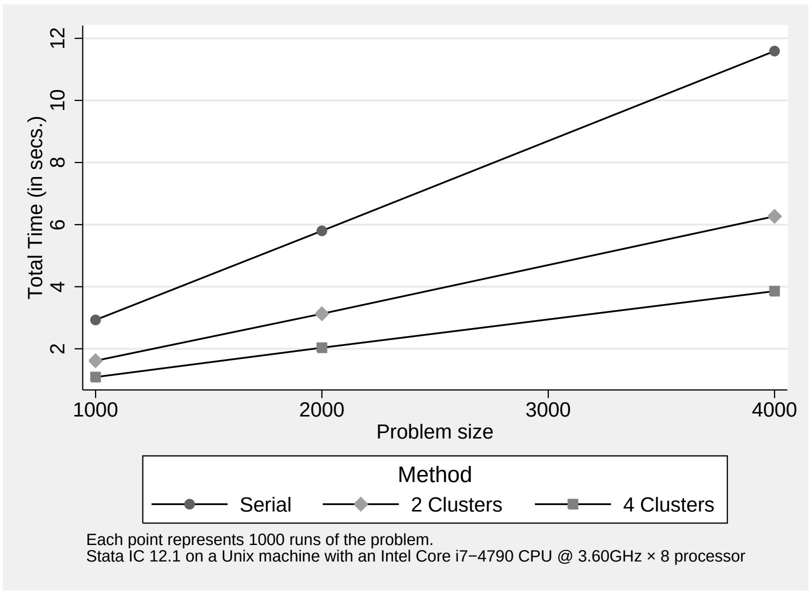


Figure 2: Bootstrap computing times by problem size (number of repetitions).

Problem size	Serial	2 Clusters	4 Clusters
1000	2.93s	1.62s	1.09s
	x2.69	x1.48	x1.00
2000	5.80s	3.13s	2.03s
	x2.85	x1.54	x1.00
4000	11.59s	6.27s	3.86s
	x3.01	x1.62	x1.00

Table 4: Absolute and relative computing times for each run of a Bootstrap. For each given problem size, the first row shows the time in seconds that each method took on average to complete the task; and the second row shows the relative time each method took to complete the task relative to using `parallel` with 4 clusters.

4.2 Simulations

In the case of Simulations, we perform a simple (and uninteresting) Monte carlo experiment which consists in two main steps: (1) Generate 1,000 observations, each with $X \sim N(0,1)$ and $Y = X\beta + \varepsilon$, where $\varepsilon \sim N(0,1)$ and $\beta = 2$, and (2) obtain the parameter estimate of β. The code used follows:

```
prog def mysim, rclass
  // Data generating process
  drop _all
  set obs 1000
  gen eps = rnormal()
  gen X   = rnormal()
  gen Y   = X*2 + eps

  // Estimation
  reg Y X
  mat def ans = e(b)
  return scalar beta = ans[1,1]
end

// Parallel fashion
parallel sim, reps($size) expr(beta=r(beta)) nodots: mysim

// Serial fashion
simulate beta=r(beta), reps($size) nodots: mysim
```

As before, for each combination of (Serial, 2 Cluster, 4 Cluster) methods and (1000; 2000; 4000) number of simulations (problem size), we ran the problem 1000 times and recorded (total) average computing time. The results are presented in Figure 3 and table 5.

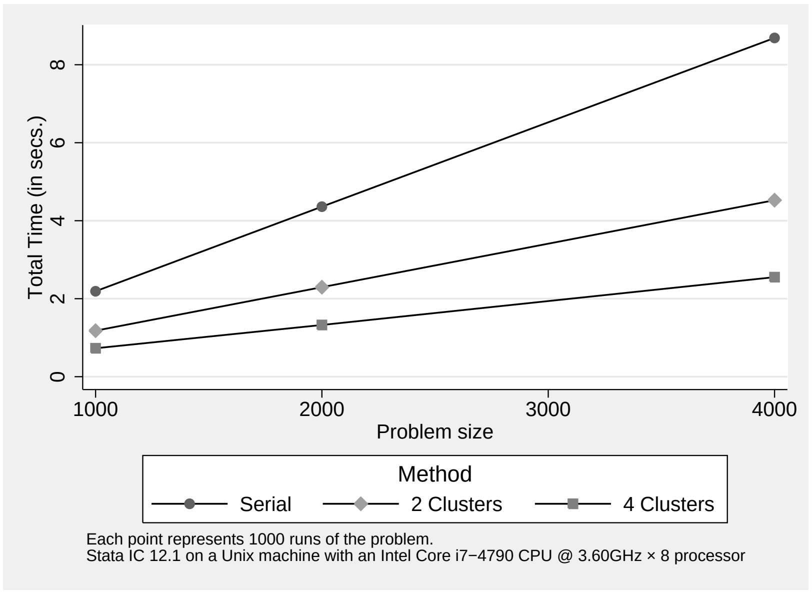


Figure 3: Simulation computing times by problem size (number of simulations).

5 Discussion

5.1 Development and feedback

In case you would like to report a bug or feature request, check first if there is an existing issue at https://github.com/gvegayon/parallel/issues. Please also try the latest development version to see if your problem has been solved already (see Section 2.4). If these do not resolve your concern, please submit an issue at the GitHub issue address so that anyone available may help to solve the issue. Include in the issue the steps to reproduce the issue and the output of the `creturn list`.

5.2 Conclusion

The `parallel` package allows users to take advantage of multicore machines for many generic tasks with a minimum of additional complexity. For tasks where the processor is the limiting factor and that are easily parallelizable, `parallel` may significantly speed up execution. We hope that this package is used not just for ad-hoc processes but can be integrated into other packages as a recommended package.

6 Reference

Stata Press. 2010. Stata/MP Performance Report. Technical report, StataCorp LP.

Problem size	Serial	2 Clusters	4 Clusters
1000	2.19s	1.18s	0.73s
	x3.01	x1.62	x1.00
2000	4.36s	2.29s	1.33s
	x3.29	x1.73	x1.00
4000	8.69s	4.53s	2.55s
	x3.40	x1.77	x1.00

Table 5: Absolute and relative computing times for each run of a Monte carlo experiment. For each given problem size, the first row shows the time in seconds that each method took on average to complete the task; and the second row shows the relative time each method took to complete the task relative to using `parallel` with 4 clusters.

http://www.stata.com/statamp/statamp.pdf.

About the authors

George G. Vega Yon is a Research Programmer at the University of Southern California.

Brian Quistorff is an Economic Researcher at Microsoft Research, Redmond.