The Stata Journal (*yyyy*) *vv*, Number *ii*, pp. 1–16

Parallel: Stata module for parallel computing

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

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

Abstract. The `parallel` package allows parallel processing of tasks that are not inter-dependent. 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 currently have multiple "processors". Physically there may be multiple "cores" in one "socket", and multiple sockets in one system. There may be more logical processors if the cores use simultaneous multithreading (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 very easy to parallelize. These tasks, called "embarrassingly 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". It proceeds as follows,

1. `parallel` splits the dataset into N pieces.
2. `parallel` starts N new instances of Stata. These 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 and loads this into memory.

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

There are two considerations that limit the parallelization in practice. First, it will never be useful to use more clusters than the number of processors on the machine.

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.

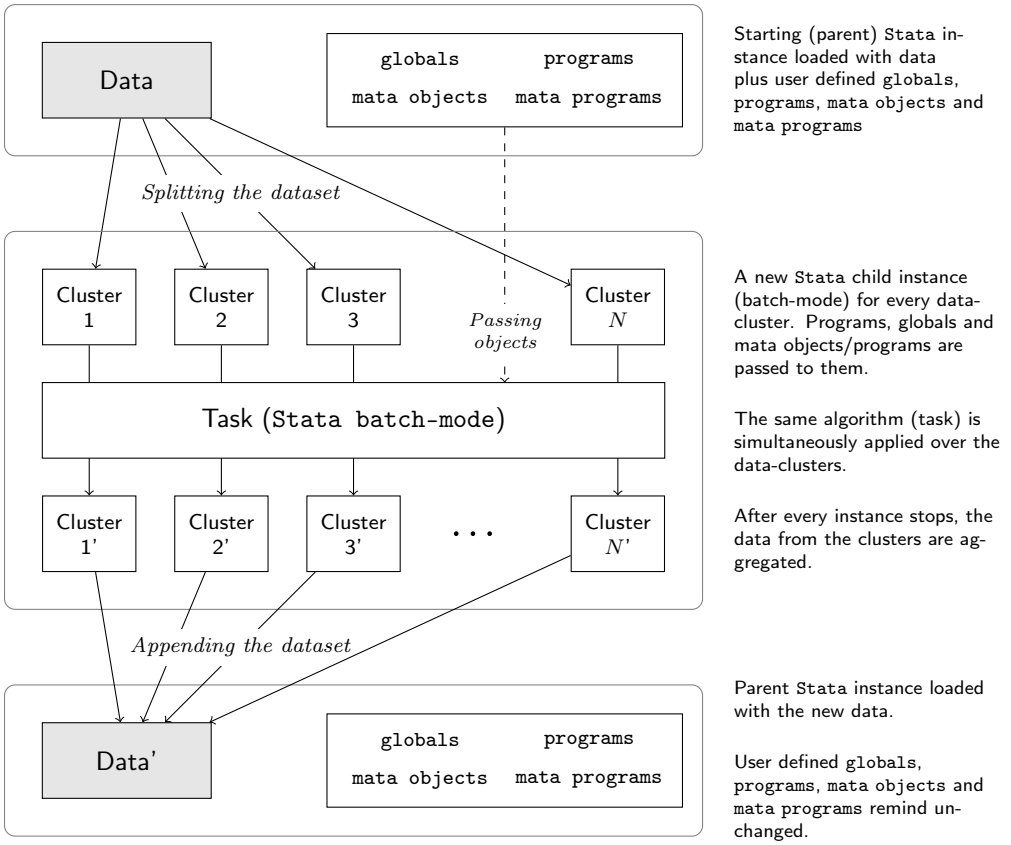

Figure 1: How `parallel` works

Second, processing a task in parallel using `parallel` uses more memory (i.e. RAM). 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 the system memory.

Some existing solution are able to take advantage of multiprocessors systems. 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 expands the set of possible parallelizations and allows this for single-threaded flavors of Stata. This module is similar to R's `parallel` package (superseding the `snow` package) and Matlab's parallel toolbox.

The document proceeds as follows. Section 2 discusses the details of execution, section 3 provides some usage examples, section 4 provides some execution speed details, 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. Diagnostic tool commands are also available.

2.1.1 Initialization

To initialize the parallel setup use the `setclusters` subcommand

parallel setclusters *#*|*default* [, 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:

- `#` - The number of clusters to use. One can also input `default` which sets it to $\max(\lfloor(\text{num processors}) \cdot 0.75\rfloor, 1)$. If there are multiple processors, the default will leave some free for other computer interactions, which should be fine for testing on a personal computer. Note that when using Stata/MP with child tasks that are automatically parallelized by Stata, care should be taken with this option and the `processors()` option for the execution so as not to inadvertently use more processors than are available.

- `force` - To prevent slowdowns there is a soft limit that restricts setting the number of clusters to be more than the number of processors on the system. Use this option to override the limit.

- `statapath(`*`stata_path`*`)` - By default, `parallel` tries to automatically identify Stata's executable path. Using this option will override this and force `parallel` to use a specific path to the executable.

- `includefile(`*`filename`*`)` - This file will be `included` in the child processes before the parallelized tasks are executed. This allows one to copy over preferences that `parallel` does not copy automatically (see section 2.3).

Use the following subcommand to determine the number of processors on a system.

parallel numprocessors

2.1.2 Parallel task execution

The following 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 [*file(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 removes those saved prior to the current execution.
- noglobal - Avoids 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 print the contents of those programs to the output window. If `parallel` is being run from inside an ado file (say `my_cmd.ado`) and will need to access auxiliary local subroutines (other programs defined in the ado), then their names must be passed in as `<main command name.local subrouting name>` (e.g. `my_cmd.aux_prog`) for them to be accessible.
- mata - If the algorithm needs to use Mata objects, this option causes each cluster to receive 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 random number generator 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 specific random seeds to be used within each cluster.
- processors(*integer*) - If running on Stata/MP, sets the number of processors each cluster should use. The default value is 0 which means to take no specific change in the child processes.
- timeout(*integer*) - If a cluster hasn't started, how much time in seconds does `parallel` wait until it assumes that there was a connection error and thus the child process (cluster) won't start. The default value is 60.

- outputopts(*namelist*) - Allows generic file-based appending. First, imagine a non-parallel setup where a program generates multiple outputs and the extra outputs are stored in files as in
  `. my_prog, output1(outputfile1.dta) output2(outputfile2.dta)`
  With `parallel` we add the option `outputopts(output1 output2)` as in
  `. parallel, outputopts(output1 output2): my_prog, output1(outputfile1.dta) output2(outputfile2.dta)`
  This causes `parallel` to run the parallel tasks with their own pair of temporary files passed in for `output1` and `output2` and then aggregates those to create `outputfile1.dta` and `outputfile2.dta`.

- deterministicoutput will eliminate displayed output that would vary depending on the machine (e.g. timers, seeds, and number of parallel clusters) so that log files can be easily compared across runs. Errors are still printed.

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

Main `parallel` subcommand options:

- by(*varlist*) - Tells the command through which observations the current dataset can be divided, avoiding splitting stories (panels) over two or more clusters. [3]

- force - When using `by()`, `parallel` checks whether the dataset is properly sorted. The `force` option skips this check.

- nodata - Tells `parallel` not to use loaded data and thus not to try splitting at the beginning or appending anything at the end.

- setparallelid(*pll_id*) - Forces `parallel` to use a 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:

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 Stata-style semantics are needed, the solution is to add `by` in the subcommand. For example,
`. parallel, by(byvar): by byvar: egen x_max = max(x)`.

- expression(*exp_list*) - An expression 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.

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 remaining 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 users 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 run. Options:

- # - specifies which cluster number of an event to display (default is 1).
- event(*pll_id*) - Specifies which event's log file 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 spent appending and cleaning.
`r(pll_t_calc)`	Time spent completing the parallel job.
`r(pll_t_setu)`	Time spent setting up (before the parallelization) and to finishing 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_STATA_PATH, PLL_CLUSTERS, USE_PROCEXEC`	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 random number generator state upon completion. Subcommands that invoke randomization functions restore the state before finishing.

Log files from the children 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 to set different tasks/actions depending on the cluster number. Additionally, the global macro `PLL_CLUSTERS` (equal to N) is available within each cluster. Note that the locals will be not available inside of programs that are called from `parallel` (in prefix or do-file setup), but will be available inside a script called from `parallel do`.

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, and the `tempname`/`tempvar` state. To start child processes with additional setting changes then one 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 steal the user interface's focus.

Results not explicitly saved in the child processes's datasets will not be available afterward (e.g. matrices, scalars, Mata objects, returns). Programs can often be modified to aggregate data in the primary dataset or using secondary files (see the *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`.

Expressions run in the child-processes that contain `_n` or `_N` will be evaluated locally to the child not the parent dataset. These expressions may therefore be different if run in `parallel` than without `parallel`.

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 one is switching the source of the installation materials (e.g. if moving 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 in parallel since parallel execution has its own overhead and the task was quite easy. The short examples that follow assume that `parallel` has been setup.

◁

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
```

When parallelized it becomes

```
. 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 from "2008_01" up to "2012_12", that 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`:

- `do(myprogram)`: execute the command `myprogram`,
- `prog(myprogram)`: `myprogram` is a user written program that needs to passed to child clusters, and
- `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, that 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
2. Once each cluster starts, for each dta file:
    a. Opens the file using [if] [in] accordingly to in and if options.

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

c. Stores the results in a temporary dta file.

3. Finally, once all the files have been processed, append all the resulting files into a single one.

◁

3.2 Parallelizing a loop

If a user has a loop where the processing in each iteration 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 = <number set by user>
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
```

3.3 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 loops (section 3.2) 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`.

◁

3.4 Parallelizing user commands

A third-party Stata package developer with easily parallelizable tasks can 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 loops or the sequential consistency example. One can put that secondary program in the original ado file (in which case use `myado.ado.subtask` form) or one can make a separate file.

3.5 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, `trace` can be turned on in the executed task or custom diagnostic information and be printed.

4 Benchmarks

In order to assess the speed-gains obtain when using `parallel`, we present what we think are the two most relevant uses of the module: bootstrapping and simulations. We compared the performance of running each routine in the following fashions on computer with at least four processors[4]: serial, parallel using two clusters, and parallel using four clusters. 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 illustrate the benefits of using `parallel`.

It is important to keep in mind that, as we will see, the lack of perfectly linear speed-

4. Tests were run using StataIC 12.1 on a Unix machine with an Intel i7-4790 CPU @ 3.60GHz with 8 processors.

gains is due to the simplicity of the problem with respect to the time that it 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* dataset 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

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

// Parallel fashion
parallel setclusters 2
parallel bs, rep($size) nodots: regress mpg weight gear foreign
parallel setclusters 4
parallel bs, rep($size) nodots: regress mpg weight gear foreign
```

For each number of repetitions (1000; 2000; 4000) we ran the problem 1000 times and recorded average computing time. The results are presented in table 4.

Problem size	Serial	2 Clusters	4 Clusters
1000	2.93s	1.62s	1.09s
	×2.69	×1.48	×1.00
2000	5.80s	3.13s	2.03s
	×2.85	×1.54	×1.00
4000	11.59s	6.27s	3.86s
	×3.01	×1.62	×1.00

Table 4: Absolute and relative computing times for each run of a basic bootstrap problem. 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 four 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 as $Y = X\beta + \varepsilon$ where $X \sim N(0,1)$, $\varepsilon \sim N(0,1)$, 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

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

// Parallel fashion
parallel setclusters 2
parallel sim, reps($size) expr(beta=r(beta)) nodots: mysim
parallel setclusters 4
parallel sim, reps($size) expr(beta=r(beta)) nodots: mysim
```

As before, for each number of simulations (1000; 2000; 4000), we ran the problem 1000 times and recorded average computing time. The results are presented in table 5.

Problem size	Serial	2 Clusters	4 Clusters
1000	2.19s	1.18s	0.73s
	×3.01	×1.62	×1.00
2000	4.36s	2.29s	1.33s
	×3.29	×1.73	×1.00
4000	8.69s	4.53s	2.55s
	×3.40	×1.77	×1.00

Table 5: Absolute and relative computing times for each run of a simple Monte Carlo exercise. 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 four clusters.

5 Discussion

5.1 Development and feedback

In case one 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 the problem has been solved already (see section 2.4). If these do not resolve the 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 multiprocessor 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. 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 AI & Research.