1 Summary

Stata/MP[1] runs on multiprocessor computers—computers with more than one CPU.

In a perfect world, software would run twice as fast on 2 CPUs, three times as fast on 3 CPUs, and so on. Stata/MP achieves about 72% efficiency. It runs 1.4 times as fast on 2 CPUs, 1.8 times as fast on 3 CPUs, and 2 times as fast on 4 CPUs. Half the commands run faster than that, and a few achieve performance beyond what would have been considered theoretically possible (more than twice as fast on 2 CPUs, etc.) because multiple-CPU systems have greater cache size. The other half of the commands run slower than the median speedup, and some commands are not sped up at all, either because they are inherently sequential (time-series commands) or because no effort was made to parallelize them (graphics, `xtmixed`).

In terms of evaluating average performance improvement, commands that take longer to run—such as estimation commands—are of greater importance. When estimation commands are taken as a group, Stata/MP achieves an even greater efficiency of approximately 88%: estimation commands run 1.7 times as fast on 2 CPUs, 2.4 times as fast on 3 CPUs, and 2.8 times as fast on 4 CPUs. Stata/MP supports up to 32 CPUs.

This paper provides a detailed report on the performance of Stata/MP. Command-by-command performance assessments are provided in section 8.

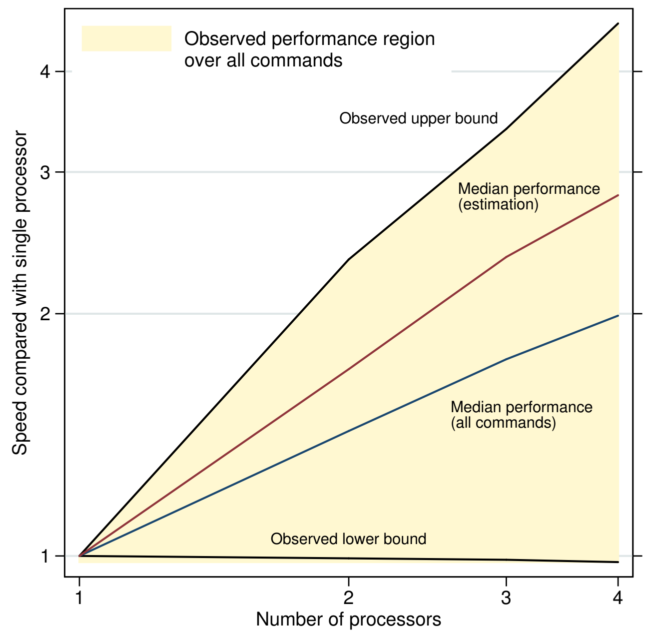

Figure 1. **Performance of Stata/MP.** Speed on multiple processors relative to speed on a single processor.

1. Support for this effort was partially provided by the National Institute on Aging, National Institutes of Health grants 1R43AG019542-01A1, 2R44AG019542-02, and 5R44AG019542-03. We also thank Cornell University, Cornell Institute for Social and Economic Research for graciously providing access to several highly parallel SMP platforms. CISER staff, and in particular John Abowd, Kim Burlingame, Janet Heslop, and Lars Vilhuber, were exceptionally helpful in scheduling time and helping with configuration. The views expressed here do not necessarily reflect those of any of the parties thanked above.

2 Table of contents

3 Introduction

Stata/MP was designed to take advantage of computers with multiple processors or with dual-core processors by partitioning the work among the multiple processors. From the outset, it was required that Stata/MP be 100% compatible with all other flavors of Stata, including Stata/SE and Intercooled Stata, and that Stata/MP run scripts, user-written programs, and analyses that run under existing Stata without any change or special action on the user's part.

Stata/MP runs on multiprocessor and dual-core computers, including computers running MS Windows (2000, XP, and later), Intel-based Apple Macintosh computers, Linux computers, and 64-bit Sun computers running Solaris.

With multiple processors, one might expect to achieve the theoretical upper bound of doubling the speed by doubling the number of processors—2 processors run twice as fast as 1, 4 run twice as fast as 2, and so on. However, there are three reasons why such perfect scalability cannot be expected: (1) some calculations have parts that cannot be partitioned into parallel processes; (2) even when there are parts that can be partitioned, determining how to partition them takes computer time; and (3) multiprocessor systems duplicate only CPUs, not all the other system resources.

Stata/MP achieved 72% efficiency overall and 88% among estimation commands.

Speed is more important for large problems, where *large* is quantified in terms of the size of the dataset or some other aspect of the problem, such as the number of covariates. On large problems, Stata/MP with 2 processors runs half of Stata's commands at least 45% faster than on a single processor. With 4 processors, the same commands run at least twice (100%) as fast as on a single processor.

Figure 1, shown in the summary above, summarizes the observed performance across all Stata commands as a shaded region. All Stata commands fall somewhere in the shaded region. Performance is measured as a percentage: 0% means zero speedup, whereas 100% means twice as fast or half the time with respect to a single processor.

Half of Stata's commands run at last 45% faster on 2 processors, and half improve less. Half run 100% faster on 4 processors, and half improve less.

The shaded region reveals that some commands improved more than would have been thought theoretically possible. This is usually due to better use of the processors' onboard cache, and such superscalability depends on the size of the problem. At the other end of the spectrum, some Stata commands experience no speedup at all. This is because their calculations are inherently sequential or because no effort was made to partition the work into parallel processes.

In typical use, Stata's estimation commands consume the bulk of the time required to perform analyses, and therefore speeding them up was a priority. Figure 1 also shows the median performance of Stata's estimation commands.

The median estimation command runs 70% faster on 2 processors and 180% faster on 4 processors. Again, half of the estimation commands speed up more, and half, less. Not shown on the graph is that 25% of estimation commands speed up more than 90% with 2 processors and more than 245% with 4 processors.

Figure 1 emphasizes 2- and 4-processor computers because those are the most common multiprocessor platforms available to users. Stata/MP will work with up to 32 processors, however, and performance improvements continue with more processors. For example, 25% of estimation commands run at least 600% faster on 8-processor computers, 900% faster on 16-processor computers, and 1300% faster on 32-processor computers.

For assessments of performance gains of individual Stata commands, see section 8. See appendices A and B for results reported in graphical form.

4 Parallel computing hardware

There is a movement toward making computers that have two or even more CPUs. Until recently, chip makers essentially have doubled the speed of processors every 18 months, a fact known informally as Moore's law (Moore 1965). This has been done by making components smaller, hence reducing electrical resistance, and by placing more transistors on a processor. Chip makers, however, are reaching the physical limits of what can be achieved through reduced size and increased complexity using existing technology. Although there are alternatives on the horizon for further speeding up processors, they involve dramatic changes in technology and fabrication.

The other solution to making computers run faster is simply to give you more of them.

One way is to put multiple processors in one box, with each processor sharing the main memory, disk drives, and other devices on the computer. The multiple processors can be on different chips or together on one chip. When the multiple processors are on one chip, they are called multicore CPUs.

Multicore or multiprocessor makes no difference: both are multiple-processor systems. These designs work exceptionally well when running different programs simultaneously, especially when programs run independently. Hence a 4-processor computer can do as much work as 4 separate computers, and none of the programs needs to be modified to recognize that they are running in a multiprocessor environment.

Single programs can take advantage of multiprocessor environments, too, but they must be modified to do so. This is done by allowing different parts of the program to run simultaneously in what are called separate execution threads. For example, a word processor might allow you to print a document and edit simultaneously. This type of threading is relatively easy to implement and is even allowed on single-processor computers to make programs more convenient.

This type of threading adds convenience but does not address the issue of speeding the computations in a statistical package. What is required there is the ability to perform computations at the same time on the same task. This is typically referred to as symmetric multiprocessing (SMP).

Stata/MP is a modified version of Stata for running in the SMP environment.

There is another type of parallel processing that involves using multiple computers over a network. This is known as cluster computing or distributed computing. Such methods require problems that admit large-grain parallelization. Although such methods can be of interest in the computation of statistical results, Stata/MP does not address such parallel architectures.

For a thorough discussion of parallel computing, see Culler, Singh, and Gupta (1999) and Grama, Kumar, Gupta, and Karypis (2003).

5 Constructing Stata/MP

For Stata to take advantage of SMP systems, sections of its code had to be rewritten to distribute their work across processors. Stata's internal design includes a few core algorithms that are used in many contexts. Those core algorithms were rewritten. The benefits then spread themselves across Stata. Statistical computations lend themselves especially well to parallelization because observations are usually independent, and independent pieces can be calculated separately. That is, statistical computations can often be partitioned over observations.

This resulted in a little more than half the observed performance gains.

The remaining gains were achieved by modifying individual routines for important Stata commands and including custom code to parallelize them.

In all, approximately 387 sections of Stata's internal code were modified.

This parallelization was performed using the Open/MP API for developing SMP applications (see Dagum and Menon 1998).

6 Measuring Stata/MP's performance

There is a theoretical limit to how much the performance of a program or command can be improved with multiple processors. With 2 that limit is twice as fast (or half the run time), with 4 processors it is 4 times as fast (or one-quarter the run time), and so on. This is called linear or perfect scaling.

Furthermore, not all algorithms or sections of code can be made to run in parallel. Some computations, or parts thereof, are inherently single threaded, e.g., a formula that depends on prior values of itself such as the autoregressive process:

$$y_t = \phi + \rho y_{t-1}$$

Statistical calculations are often more parallelizable than one imagines. For instance, many inherently sequential computations can be parallelized when performed on longitudinal (panel) data because the dependencies that made the problem inherently sequential are broken at panel boundaries. Rather than partitioning on observations, one partitions on panels. Stata/MP does this. Whereas most time-series commands run only a little faster in the SMP environment, most panel-data commands run substantially faster.

There can also be sections of code that are simply not worth the effort of parallelization because they take so little time to run or parallelization would be technically difficult. Either way, the effort simply is not worth the benefit.

Taken together, these are the nonparallelized region. Some authors refer to the parallelizable regions and the parallelized regions—the first referring to what could be parallelized and the second to what was actually parallelized—and even focus on the ratio between the two. We will focus on run times, however, and draw no distinction between parallelizable and parallelized.

How much of a calculation has been parallelized is measurable, and measuring it is useful because it allows one to make extrapolations on how problems will run when the number of processors varies.

Figure 2 presents a stylized view of the component run times associated with a command that has been parallelized. Block A represents the time spent in parallelized regions of code; Block B, the unparallelizable (or just unparallelized) regions of code; and Block C, the additional overhead required for parallelization.

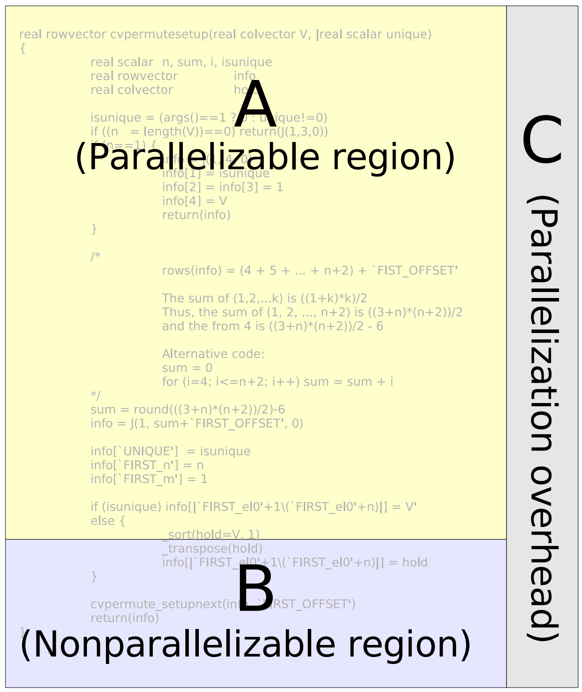

Figure 2. **Parallelization components**.

Let each letter represent an amount of time consumed in running a particular command on a particular dataset. Then $A+B$ is the run time of the command when using a single processor. If we parallelize the command, however, there is an additional time, C, associated with the overhead of partitioning the problem and coalescing the results from the processors.

We will refer to $100A/(A+B)$ as the percentage parallelized:

$$\text{percentage parallelized} = \frac{100A}{A+B}$$

The percentage parallelized is a useful measure of how much performance will improve as processors are added. All gains to parallelization occur because region A can be made to run on multiple processors in parallel. If we partition the region perfectly and each processor runs uninterrupted, when we double the number of processors, we halve the time to perform A while the time required to perform B and C remains unchanged. $B+C$ is a constant time for running the command that cannot be reduced by adding more processors.

Said differently, the total time to execute a process on a single processor is $A+B$. The total time to execute the process on p processors is $A/p+B+C$. We can rewrite that as $(P/100p)*(A+B)+B+C$. Thus P/p is an approximation of how much the command's run time decreases, ignoring the fixed cost of $B+C$.

We can also measure the parallelization overhead in terms of the overall run time on a single processor:

$$\text{parallelization overhead} = \frac{100C}{A+B}$$

Theoretically, there is no good reason why the denominators in both expressions could not be $(A+B+C)$. Operationally, however, there is a great deal of difference in how processors, caches, computer architectures, and operating systems handle the division of labor between the first processor and all the remaining processors. For this reason C, and thus parallelization overhead, varies from one computing platform to another, whereas A and B, and thereby percentage parallelizable are comparable across platforms.

We are also ignoring another contribution to run time. Sometimes there is overhead associated with each processor, rather than, or in addition to, an overall parallelization overhead. Because of the methods used to build Stata/MP, this overhead is extremely small and it affects only four commands, and even on those commands the effect is small.

Understanding percentage parallelizable and parallelization overhead clarifies why some commands will have less than perfect scaling and allows results to be extrapolated to more processors. We also present performance results as simple relative run times that can be read directly from tables or graphs to find the run time for multiple processors compared with the run time for a single processor.

7 Performance summary

The performance of Stata/MP has been measured on all 332 Stata commands that take any appreciable time to run. Commands such as `display`—which writes output to the Results window—or `local`—which sets the value of a program macro—are not considered. Such commands consume a negligible part of the time required to perform any analysis.

Other commands that were not explicitly assessed include replication-based commands such as **`bootstrap`**, **`jackknife`**, **`permute`**, **`simulate`**, and **`statsby`**; as well as other prefix commands. These commands run another target command repeatedly, and to the extent the target command's performance is improved for a particular problem size, a similar improvement will be obtained when it is run repeatedly.

For each of the 332 commands, timings were recorded on a multiprocessor computer where Stata/MP used 1, 2, 3, and 4 processors to execute the same command. All these timings were from the same installation of Stata/MP on the same computer. To reduce the impact of interruptions by the operating system, the timings were repeated three times and the median time reported. Timings have also been performed on other 2-processor and dual-core computers, 4-processor computers, 8-processor computers, and 16-processor computers. Although timings relative to a single processor do vary among tested platforms, they are generally comparable, and the results presented are indicative of what can be expected across a spectrum of platforms. The timings are presented in section 8, *Stata/MP performance, command by command*, and appendix A, *Performance assessment graphs*.

Appendix A, *Performance assessment graphs*, shows graphs for each of the 332 commands. Here is the graph of Stata's linear regression command, `regress`:

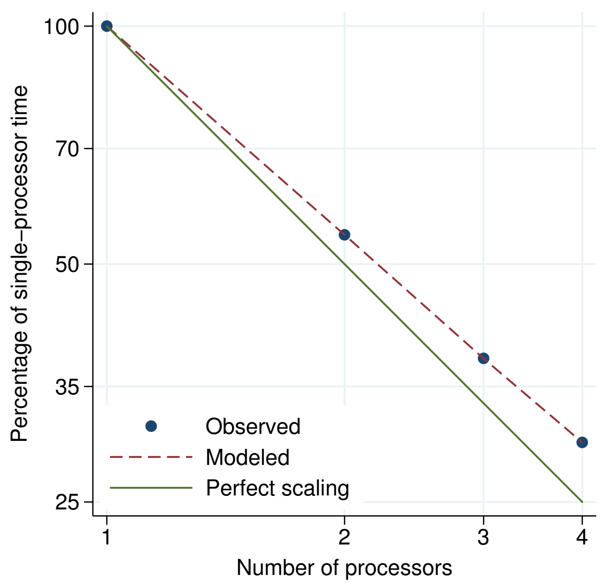

Figure 3. `regress` performance plot.

The y-axis shows run times relative to the run times on single processors. For `regress`, the relative run times are 54% (2 processors), 38% (3 processors), and 30% (4 processors). Also shown is a 45° reference line reflecting perfect scalability or, if you prefer, 100% parallized: 50% (2 processors), 33% (3 processors), and 25% (4 processors). `regess` very nearly achives theoretical limits; it has run times that decline very nearly in proportion to the number of processors.

Here is the graph for `arima`:

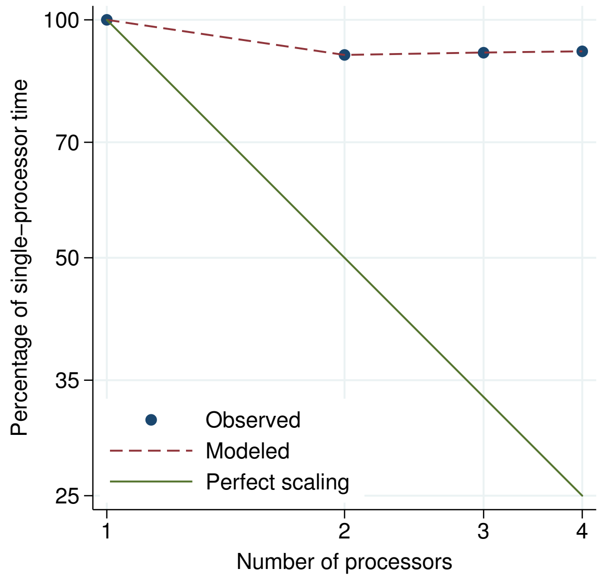

Figure 4. `arima` performance plot.

`arima`, a time-series command, hardly benefits from parallelization. Run times fall to 90% (2 processors), 91% (3 processors), and 91% (4 processors). Run times actually increased just a little at one point. Remember, these are empirical timings (3 measurements, median reported). In any case, `arima` shows little gain from parallelization.

Here is the graph for Stata's regression with random effects, `xtreg`:

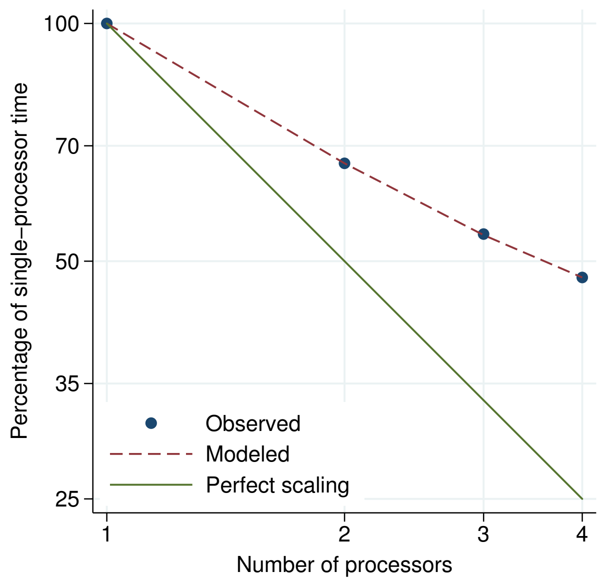

Figure 5. `xtreg, re` performance plot.

Run times fall to 74% (2 processors), 63% (3 processors), and 58% (4 processors). What is interesting about this graph is the flattening out as the number of processors increases. This is what happens when a command is not 100% parallized: the relative run time approaches a horizontal asymptote that is the percent not parallelized, which here is about 61%.

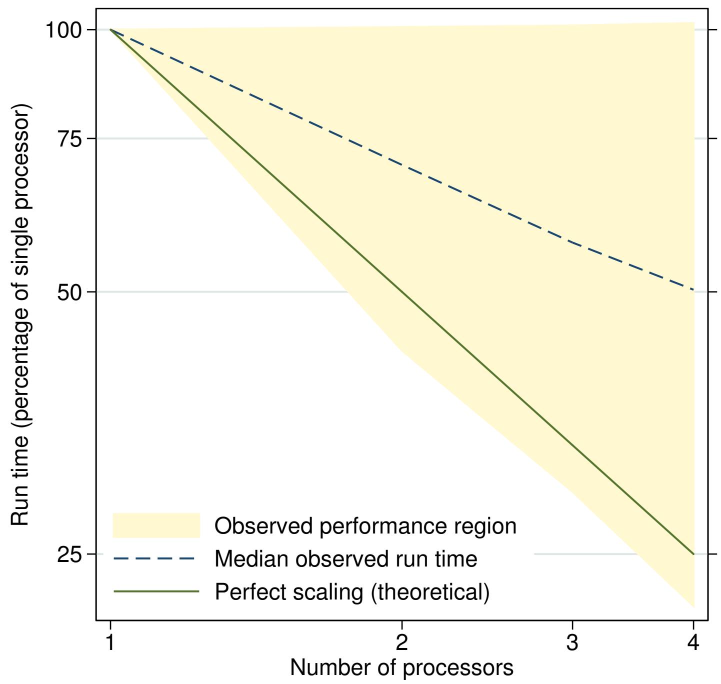

Figure 6. **Performance of Stata/MP.** Run times on multiple processors relative to a single processor.

Finally, all 332 figures can be combined into one figure, such as figure 6. The shaded area shows the region containing the 332 individual results. The lower boundary of the area extends a little below the 45° line. This means that at least one command exhibited better-than-perfect scaling. Such superscalability is due to cache effects.

Also included are the median results over all 332 commands; 166 commands have better performance gains (their curves lie below the line), and 166 exhibit lesser performance gains (their curves lie above the line).

Median performance across users will probably be better than median performance across commands as we calculated it. To be able to measure performance, we had to choose large problems even when, for a particular command, large problems are rarely run. For instance, few users would run analyses that spend as much time running *t* tests as did those we had to run to record reliable results. Stata's *t* test command runs quickly on single or multiple processors. Meanwhile, Stata/MP development efforts were focused on improving run times of commands that require substantial run times. Ergo, the median improvements are understated.

By the way, figure 6 is logically the same as the graph shown in figure 1 of the Summary. Just the units are different.

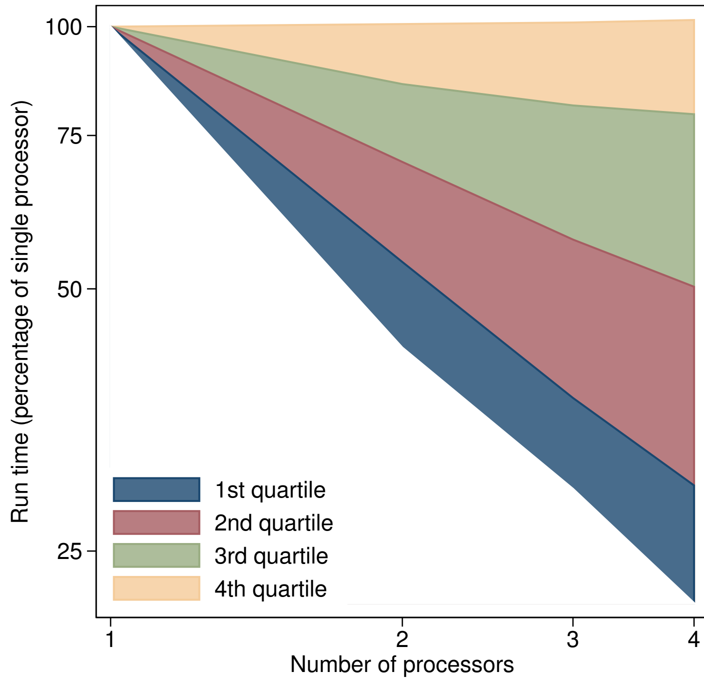

Figure 7. **Quartiles of Stata/MP performance**. Run times on multiple processors relative to a single processor.

Figure 7 better shows the distribution of results by showing not just the median but the quartiles. The most interesting thing about figure 7 is the first quartile (blue swath at the bottom). It shows that 25% of commands exhibit nearly perfect scaling. The worst of this group run in 54% of the time on two processors and 30% of the time on four processors.

Figures 6 and 7 present results for all commands, whereas the time required by most analyses is dominated by execution of estimation commands. Estimation commands tend to be the most computationally intensive, particularly those that required iterative solutions.

Figure 8 summarizes the observed performance and median performance for the 130 estimation commands. These include all the estimation commands in Stata, and some commands are included more than once to include critical options, such as `robust` and `cluster` for robust standard errors and correlation within groups. The options themselves are not important; what is important is that these options (and a few others like them) substantively affect how the calculation is made and thus run times.

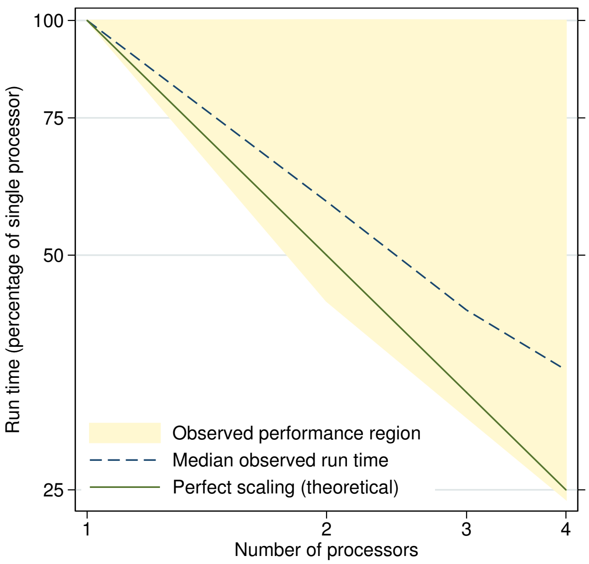

Figure 8. **Performance of Stata/MP on estimation commands.** Run times on multiple processors relative to a single processor.

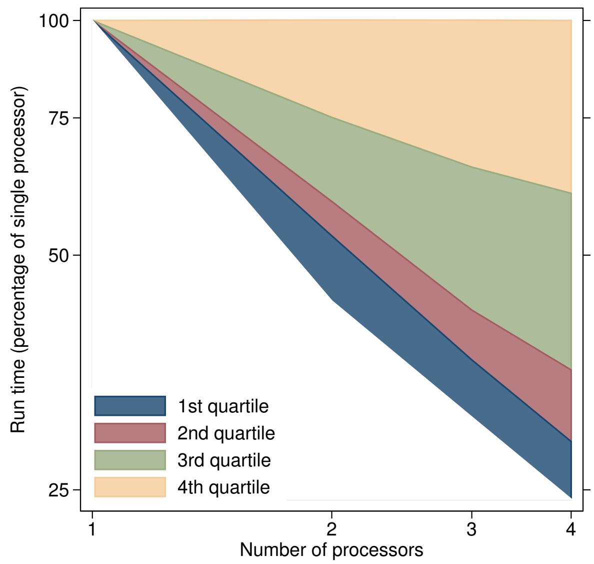

Figure 9. **Quartiles of Stata/MP performance on estimation commands**. Run times on multiple processors relative to a single processor.

Compared with figure 6, note that the median performance for estimators is better than the overall median. The median run time for estimators is less than 60% (2 processors) and 35% (4 processors). Half of all estimators perform even better and thus approach perfect scaling. Figure 9 reveals that only 25% of all estimators run in more than three-fourths of the time (2 processors) and 60% of the time (4 processors).

We have emphasized results on two and four processors, because that is the most common architecture currently available to users. Stata/MP supports up to 32 processors, however, and performance continues to improve as processors are added. Figure 10 shows the performance boundary and median for 115 common commands on a 16-processor computer, and figure 11 shows the performance quartiles.

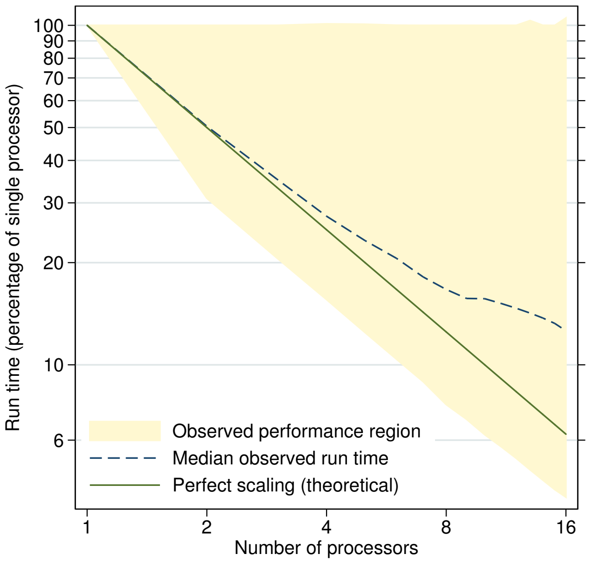

Figure 10. **Performance of Stata/MP on 1 to 16 processors.** Run times on multiple processors relative to a single processor.

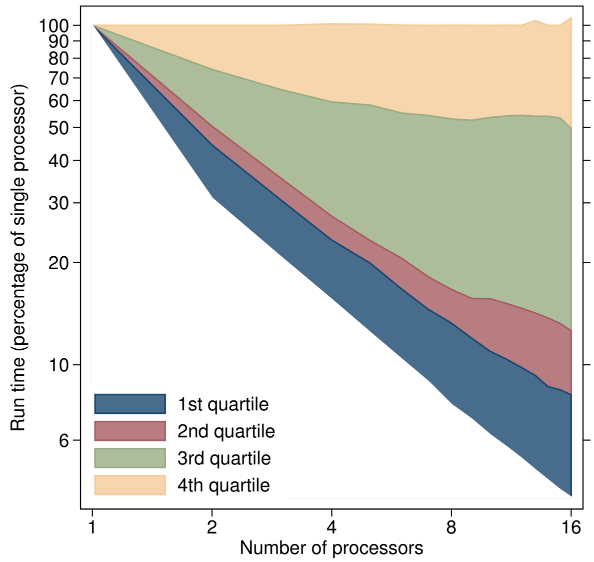

Figure 11. **Quartiles of Stata/MP Performance on 1 to 16 processors**. Run times on multiple processors relative to a single processor.

Despite a slight, temporary flattening of growth at 8 processors, performance continues to improve—and to improve at generally the same rate through 16 processors. If commands exhibited perfect scaling with 4 processors, they continued to exhibit it through 16 processors. This supports extrapolating results from 4 processors to more.

8 Stata/MP performance, command by command

The performance summaries from the prior section provide an overall sense of the performance of Stata/MP but will not reflect the experience of most users. Few users perform all the commands in Stata, and no users perform them with equal frequency. Most users will be interested in a subset of commands and often in only a few commands that they use regularly and on large problems.

The table at the end of this section provides timings on individual commands, comparing the run time on 2, 4, and 8 processors with the run time on a single processor. It also provides an estimate of the degree to which each command is parallelized.

All commands were run on moderately large to very large problems. The goal was to measure performance on problems that require substantial time to solve and that were large enough to measure performance gains on 8, 16, or even 32 processors. For commands that are parallelized, such problems have a larger parallelizable region (A) relative to the unparallelizable region (B) and are thus more amenable to parallelization, particularly when run on many processors. Longer timings also ameliorate variations in timings, such as interruptions for operating system processes or the memory status of the system when the command begins. Substantial variation occurs when run times are short.

Timings were typically performed on commands that took between 8 and 15 seconds to run on a single-processor computer running at 2.8 to 3.2 GHz. For some commands, this meant the problems used extremely large numbers of observations or covariates, because some commands are inherently fast. For others, the problems were smaller because the commands are inherently slower, due for example to iterative or even simulated solutions. For details on the size of the problems, see appendix D.

Stata/MP was designed to improve performance on large problems, such as those reported in appendix D. Even so, the performance improved surprisingly well on small to moderate problems. Using the same commands as those in appendix D, but with problems 100 to 1,000 times smaller (run times of two-fifths to just over 1 second on a 2.8- to 3.2-GHz machines), substantial speedups were still observed. Among commands that were at least 50% parallelized, more than half exhibited better than 80% of the speedup exhibited on the larger problems. These are typical results. Run times for smaller problems vary more from computer to computer because small problems are more sensitive to the architecture of the computer, processor, and operating system.

All values, except the columns for 8 and 16 processors, were obtained from the median of three runs on an 4-processor computer. The columns for 8 and 16 processors were obtained from a single run—not three—using a 16-processor computer in 8-processor and 16-processor modes. All commands were not tested on the 16-processor computer. When timings were not performed for 8- and 16-processors, expected values were extrapolated from the results on the 4-processor computer.

Stata/MP performance has been tested on many computers under MS Windows, Macintosh, Linux, and Solaris operating systems. Although performance varies somewhat across platforms, the results from the table below can reasonably be applied to any of the platforms. The percentage run times are rounded to integer percentages; when comparing across platforms it would be reasonable to round them to the nearest 5th, or occasionally 10th, percentage.

Most users should simply look at the column reporting results for the number of processors in which they are interested. This column estimates the run time on that number of processors as a percentage of the run time on a single processor. Given a computer with a known number of processors, this is the most direct measure of performance improvement.

The table also presents the percentage parallelized discussed in section 6. Given a set of percentage run times for at least 3 processors, we can estimate the percentage parallelized and parallelization overhead parameters from the run times. The form of the model is particularly simple,

$$\text{percentage run time} = \frac{\widehat{A}}{p} + \widehat{C}\delta_1 + \widehat{B} \tag{1}$$

where p is the number of processors and δ_1 is an indicator for $p > 1$.

As defined in section 6, we then have

$$\text{percentage parallelized} = \frac{100\widehat{A}}{\widehat{A} + \widehat{B}} \tag{2}$$

and

$$\text{parallelization overhead} = \frac{100\widehat{C}}{\widehat{A}+\widehat{B}} \tag{3}$$

Equation 1 is estimated by median regression (`qreg`) using Stata. Median regression is used in preference to ordinary least squares (OLS) because occasionally a timing will be far too large because of interruptions from the operating system. Such effects are ignored in median regression.

The estimated value for parallelization overhead is particularly sensitive to the computing platform, and so we do not report it here. Note from equation 1 that it captures any unexpected difference in the speed using one processor. Because different computer, processor, cache, and operating system architectures respond differently in moving from 1 to 2 processors, $\widehat{B}$ captures not only the theoretical parallelization overhead B but also anything that causes the time from the first processor to differ from the second.

Percentage parallelized is the most concrete measure of how a command responds to more processors. For most commands, the run time in this percentage of the code falls by half for each doubling of the number of processors.

The estimated percentage parallelized is also the most comparable measure across computing platforms; it is nearly constant from one platform to another. Most of the differences across computing platforms are captured in $\widehat{B}$, which does not enter in the formula for percentage parallelized. Because the simpler percentage run times are compared with the run time on a single processor, they necessarily include the parallelization overhead and are thus not quite as comparable across machines.

Each line in the table represents a command run on a particular problem. The command column shows the Stata command name and relevant options. For those unfamiliar with Stata syntax, appendix C provides short descriptions of what each command does. For those without access to the Stata manuals and wanting still more information on a command, go to http://www.stata.com/capabilities/ and enter the command name in the search section at the bottom of the page.

A few of the results for `cluster` commands produced overly optimistic projections for 8- and 16-processor performance. For 16 processors, these projections are left blank; for 8 they are reported but are likely too optimistic. The estimated percent parallelized is also over 100% for several of these commands. These results will be updated when the cluster commands have been tested on 8- and 16-processor computers.

Appendix A contains performance graphs for each command using 1, 2, 3, and 4 processors. Appendix B contains graphs using 1 through 16 processors. The graphs plot the observed percentage time, the modeled performance using equation 1, and the perfect scalability reference line. If you are reading the PDF version of the document, clicking on the command name in the table will take you to the page with the associated graph.

Table 1. Stata/MP performance, command by command

command	Run time as percentage of single processor time[a]				Percentage parallelized[b]
	Number of processors				
	2	4	8	16	
adjust	85	78	74	72	28
alpha	65	40	*28*	*22*	86
ameans	69	45	*32*	*26*	83
anova (oneway)	71	47	34	26	85
anova (twoway)	69	47	37	30	77
arch	83	75	*72*	*70*	30
areg	84	77	*73*	*71*	29
arima	90	91	*92*	*92*	
asmprobit	67	50	48	51	40
binreg	55	32	*20*	*15*	91
biplot	100	100	*100*	*100*	0
biprobit	55	31	19	14	93
biprobit (seemingly unrelated)	53	29	17	11	95
bitest	63	42	*31*	*26*	80
blogit	58	39	*30*	*25*	78
boxcox	54	30	*18*	*12*	94
bprobit	56	35	*25*	*20*	85
brier	79	63	*55*	*51*	57
bsample	91	84	*81*	*79*	25
by: generate	51	26	*14*	*8*	98
by: generate (small groups)	52	27	*14*	*8*	99
by: replace	51	26	*14*	*8*	99
by: replace (small groups)	51	26	*14*	*7*	99
ca	85	83	82	95	8
canon	60	42	26	18	90

All values are expressed as a percentage of the time required on a single processor. Slanted values are extrapolated from 4 processors.

a. Smaller is better; 50 is perfect for 2 processors, 25 is perfect for 4 processors, and 12.5 is perfect for 8 processors.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

command	Run time as percentage of single processor time[a] — Number of processors: 2	4	8	16	Percentage parallelized[b]
centile	99	98	*97*	*97*	4
ci	77	61	*53*	*49*	60
ci, binomial	65	44	*33*	*28*	79
ci, poisson	61	34	*21*	*15*	93
clogit (k1 to k2 matching)	71	55	*47*	*43*	62
clogit (1 to k matching)	63	45	33	31	75
cloglog	54	30	17	11	95
cluster averagelinkage	58	26	*10*		105
cluster centroidlinkage	57	22	*4*		110
cluster completelinkage	59	27	*11*		104
cluster generate	85	74	*69*	*66*	40
cluster kmeans	26	16	*11*	*8*	88
cluster kmedians	44	33	*28*	*26*	65
cluster medianlinkage	57	22	*4*		111
cluster singlelinkage	97	97	*97*	*97*	0
cluster wardslinkage	59	27	*11*		104
cluster waveragelinkage	58	26	*10*		105
cnreg	54	30	*17*	*11*	95
cnsreg	51	27	*15*	*9*	97
collapse	85	71	*64*	*60*	49
compare	63	39	*27*	*21*	86
compress	100	100	*100*	*100*	0
contract	93	90	*88*	*87*	14
correlate	51	27	13	7	99
corrgram	82	71	*66*	*63*	42

All values are expressed as a percentage of the time required on a single processor. Slanted values are extrapolated from 4 processors.

a. Smaller is better; 50 is perfect for 2 processors, 25 is perfect for 4 processors, and 12.5 is perfect for 8 processors.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

command	Run time as percentage of single processor time[a] — Number of processors: 2	4	8	16	Percentage parallelized[b]
count	51	26	13	7	100
ctset	54	27	*14*	*8*	99
cttost	70	51	*41*	*36*	71
cumul	95	95	*94*	*94*	3
cusum	91	81	*76*	*73*	37
dfgls	91	85	*83*	*81*	22
dfuller	74	61	*54*	*51*	53
dotplot	89	84	*81*	*80*	22
dstdize	99	99	*99*	*99*	1
eivreg	50	28	15	15	98
factor	65	46	32	25	85
fcast compute	99	98	*98*	*98*	2
fracpoly	77	59	*50*	*45*	64
frontier	54	29	16	12	97
gen (small expressions)	43	23	15	12	88
generate	50	25	13	7	100
glm, family(gamma)	59	35	23	18	88
glm, family(gaussian)	60	37	*26*	*21*	85
glm, family(igaussian)	55	32	19	13	93
glm, family(nbinomial)	58	32	21	16	89
glm, family(poisson)	59	33	23	17	88
glogit	48	27	*17*	*11*	93
gprobit	46	24	*13*	*8*	97
graph bar	92	87	*85*	*84*	19
graph box	85	77	*72*	*70*	35

All values are expressed as a percentage of the time required on a single processor. Slanted values are extrapolated from 4 processors.

a. Smaller is better; 50 is perfect for 2 processors, 25 is perfect for 4 processors, and 12.5 is perfect for 8 processors.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

command	Run time as percentage of single processor time[a]				Percentage parallelized[b]
	Number of processors				
	2	4	8	16	
graph pie	85	76	*71*	*69*	36
grmeanby	89	80	*75*	*73*	34
hausman	75	74	66	65	27
heckman	55	32	19	13	93
heckman, twostep	52	28	15	10	97
heckprob	54	31	20	15	89
hetprob	54	28	16	10	96
histogram	78	63	*56*	*52*	54
hotelling	52	23	11	6	100
impute	84	73	*68*	*65*	41
intreg	53	28	16	11	95
irf create	54	45	66	*38*	50
ivprobit	52	28	19	15	88
ivprobit, cluster()	51	29	*17*	*12*	94
ivprobit, robust	52	30	*19*	*13*	92
ivreg	55	31	17	11	97
ivtobit	53	30	20	17	88
kap	93	90	*89*	*88*	12
kappa	60	34	*21*	*14*	93
kdensity	56	34	*23*	*17*	88
ksmirnov	82	68	*62*	*58*	50
ksmirnov, by()	88	81	*77*	*75*	29
ktau	100	99	*99*	*99*	1
kwallis	93	88	*85*	*84*	20
ladder	66	46	*35*	*30*	77

All values are expressed as a percentage of the time required on a single processor. Slanted values are extrapolated from 4 processors.

a. Smaller is better; 50 is perfect for 2 processors, 25 is perfect for 4 processors, and 12.5 is perfect for 8 processors.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

command	Run time as percentage of single processor time[a] — Number of processors: 2	4	8	16	Percentage parallelized[b]
levelsof	99	98	*98*	*98*	3
loadingplot	79	66	*60*	*57*	49
logistic	50	27	*16*	*10*	95
logit	50	27	15	9	97
loneway	84	72	*66*	*63*	43
lowess	46	27	*17*	*12*	92
ltable	85	79	*76*	*75*	25
manova (oneway)	94	93	92	102	4
manova (twoway)	77	71	67	66	30
markout	54	30	16	9	98
marksample	53	27	14	7	99
marksample if exp	53	27	14	7	99
matrix accum	52	26	13	7	100
matrix eigenvalues	100	100	*100*	*100*	0
matrix score	51	27	14	7	99
matrix svd	100	100	*101*	*101*	
matrix symeigen	100	100	*100*	*100*	0
matrix syminv	75	44	23	13	98
mds	47	41	39	38	34
mdslong	52	47	*45*	*44*	31
mean	98	96	96	96	3
median	67	46	*36*	*31*	76
mfp	55	33	*22*	*17*	89
mfx	84	76	71	72	28
mkmat	100	100	*100*	*100*	0

All values are expressed as a percentage of the time required on a single processor. Slanted values are extrapolated from 4 processors.

a. Smaller is better; 50 is perfect for 2 processors, 25 is perfect for 4 processors, and 12.5 is perfect for 8 processors.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

command	Run time as percentage of single processor time[a] — Number of processors: 2	4	8	16	Percentage parallelized[b]
mkspline	71	46	*33*	*27*	83
mleval	52	26	13	7	100
mleval, nocons	52	26	13	7	100
mlmatbysum	53	33	*23*	*18*	86
mlmatsum	53	28	14	8	99
mlogit	52	27	14	9	99
mlsum	65	40	27	18	88
mlvecsum	55	28	15	8	98
mprobit	99	99	99	99	0
mvreg	63	44	30	23	85
nbreg	55	31	19	14	91
newey	88	80	*76*	*74*	31
nl	83	72	*67*	*64*	41
nlogit	65	46	*37*	*32*	73
nptrend	95	92	*91*	*90*	11
ologit	48	27	14	8	97
oneway	100	100	*100*	*100*	0
oprobit	48	26	14	8	98
orthog	60	35	*22*	*16*	92
pca	71	54	41	42	64
pcorr	51	28	15	9	98
pctile	64	45	*35*	*30*	75
pergram	100	100	*100*	*100*	0
pkcollapse	90	81	*76*	*73*	35
pkexamine	101	102	*102*	*103*	

All values are expressed as a percentage of the time required on a single processor. Slanted values are extrapolated from 4 processors.

a. Smaller is better; 50 is perfect for 2 processors, 25 is perfect for 4 processors, and 12.5 is perfect for 8 processors.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

	Run time as percentage of single processor time[a]				
	Number of processors				Percentage
command	2	4	8	16	parallelized[b]
pksumm	94	88	*85*	*84*	21
poisson	54	30	16	10	96
pperron	94	93	*93*	*92*	4
prais	86	81	*79*	*77*	22
predict, cooksd	50	26	*14*	*8*	98
predict, covratio	50	26	*14*	*8*	98
predict, dfbeta	51	27	14	8	98
predict, dfits	50	26	*14*	*8*	98
predict, e	53	34	20	13	92
predict, leverage	50	26	13	7	100
predict, pr	53	29	*17*	*11*	95
predict, residuals	53	27	14	8	98
predict, rstandard	51	26	*13*	*7*	99
predict, rstudent	51	26	*13*	*7*	99
predict, stdf	50	25	13	7	100
predict, stdp	50	26	13	7	100
predict, stdr	50	27	*15*	*9*	97
predict, welsch	50	26	*13*	*7*	99
predict, ystar	53	28	19	15	91
predictnl	59	38	30	24	79
probit	53	28	16	9	97
procrustes	71	51	43	38	65
proportion	74	60	*53*	*50*	55
prtest1	60	38	*27*	*22*	85
prtest2	64	44	*33*	*28*	78

All values are expressed as a percentage of the time required on a single processor. Slanted values are extrapolated from 4 processors.

a. Smaller is better; 50 is perfect for 2 processors, 25 is perfect for 4 processors, and 12.5 is perfect for 8 processors.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

command	Run time as percentage of single processor time[a]				Percentage parallelized[b]
	Number of processors				
	2	4	8	16	
prtest, by()	86	77	*72*	*70*	35
qreg	62	41	*30*	*24*	82
ranksum	77	58	*48*	*43*	67
ratio	89	82	*79*	*77*	26
ratio (exp1) (exp2)	89	83	*79*	*78*	25
recode	93	70	*59*	*53*	65
reg3	55	32	19	14	92
regress	54	30	16	9	97
regress, cluster()	70	52	*43*	*38*	69
regress, robust	79	66	*60*	*57*	49
replace	50	25	*13*	*7*	100
replace (small expression)	50	29	*18*	*12*	92
reshape long	96	95	*95*	*94*	4
reshape wide	101	100	*100*	*99*	2
robvar	83	78	*75*	*74*	21
rocfit	65	44	*33*	*28*	79
roctab	71	53	*44*	*39*	67
rotatemat	98	98	*98*	*98*	1
rreg	51	28	*17*	*11*	94
runtest	65	46	*36*	*31*	75
scobit	54	28	16	10	96
scoreplot	99	99	*98*	*98*	3
screeplot	92	87	*84*	*83*	19
sdtest1	76	61	*53*	*49*	57
sdtest2	77	61	*54*	*50*	57

All values are expressed as a percentage of the time required on a single processor. Slanted values are extrapolated from 4 processors.

a. Smaller is better; 50 is perfect for 2 processors, 25 is perfect for 4 processors, and 12.5 is perfect for 8 processors.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

command	Run time as percentage of single processor time[a] — Number of processors: 2	4	8	16	Percentage parallelized[b]
sdtest, by()	75	60	*52*	*49*	58
sfrancia	80	68	*61*	*58*	48
signrank	81	61	*52*	*47*	65
signtest	59	32	*19*	*12*	95
sktest	63	43	34	28	76
slogit	57	41	34	31	69
sort	72	53	43	27	75
spearman	75	54	49	45	58
stack	94	91	*89*	*88*	13
stbase	93	86	*82*	*81*	26
stci	93	90	*88*	*87*	14
stcox	97	95	*94*	*94*	7
stcurve, hazard (after stcox)	95	93	*92*	*92*	7
stcurve, hazard (after streg)	97	93	*91*	*90*	15
stgen	75	55	*45*	*40*	70
stir	73	56	*48*	*44*	63
stptime	98	91	*88*	*86*	24
strate	71	62	*58*	*55*	40
streg, distribution(exponential)	54	30	17	11	95
streg, dist(exp) cluster()	59	36	*24*	*18*	88
streg, dist(exp) frailty()	53	29	18	13	92
streg, dist(exp) frailty() shared()	53	30	*18*	*12*	94
streg, dist(exp) robust	57	34	*23*	*18*	88
streg, distribution(gamma)	50	27	*15*	*9*	96
streg, distribution(lnormal)	52	27	19	15	88

All values are expressed as a percentage of the time required on a single processor. Slanted values are extrapolated from 4 processors.

a. Smaller is better; 50 is perfect for 2 processors, 25 is perfect for 4 processors, and 12.5 is perfect for 8 processors.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

command	Run time as percentage of single processor time[a] — Number of processors: 2	4	8	16	Percentage parallelized[b]
streg, distribution(weibull)	58	33	*21*	*15*	92
streg, dist(weibull) frailty()	49	26	16	11	93
streg, dist(weibull) frailty() shared()	52	30	*19*	*13*	93
sts generate	93	90	*89*	*89*	10
sts graph	90	87	*85*	*85*	13
sts list	87	83	*81*	*81*	16
sts test	83	80	*79*	*78*	14
stset	67	45	*34*	*29*	79
stsplit	91	83	*79*	*77*	31
stsum	95	90	*88*	*86*	19
stvary	74	51	*39*	*34*	76
summarize	60	31	*16*	*9*	98
sunflower	69	52	*43*	*38*	68
sureg	55	33	20	14	92
svar	44	39	*36*	*35*	37
svmat	98	99	*99*	*99*	
svy: logit	75	60	*53*	*49*	57
svy: poisson	69	51	*41*	*36*	70
svy: regress	80	67	*60*	*56*	50
swilk	77	61	*52*	*48*	61
symmetry	91	84	*80*	*78*	27
table (oneway)	86	70	*62*	*59*	53
table (twoway)	83	65	*57*	*52*	60
tabstat	80	67	*61*	*58*	48
tabstat, by()	94	88	*85*	*84*	21

All values are expressed as a percentage of the time required on a single processor. Slanted values are extrapolated from 4 processors.

a. Smaller is better; 50 is perfect for 2 processors, 25 is perfect for 4 processors, and 12.5 is perfect for 8 processors.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

command	Run time as percentage of single processor time[a]				Percentage parallelized[b]
	Number of processors				
	2	4	8	16	
tabulate (oneway)	100	100	*100*	*100*	0
tabulate (twoway)	100	100	*100*	*100*	0
tetrachoric	95	93	*92*	*91*	10
tobit	53	28	*16*	*10*	96
total	96	94	*93*	*92*	9
treatreg	53	29	18	14	90
treatreg, twostep	53	28	16	9	97
truncreg	52	29	18	13	92
tsset	94	90	*89*	*88*	14
tssmooth exp	85	76	*71*	*69*	35
tssmooth ma	89	83	*80*	*78*	25
ttest1	77	61	*53*	*49*	58
ttest2	74	55	*45*	*40*	69
ttest, by()	75	60	*53*	*49*	57
twoway fpfit	70	50	*40*	*36*	72
twoway lfitci	100	100	*99*	*99*	1
twoway mband	62	48	*42*	*38*	61
twoway mspline	62	48	*41*	*37*	63
var	81	69	64	62	16
vargranger	100	100	*100*	*100*	1
varlmar	77	63	*56*	*52*	54
varnorm	77	64	*58*	*55*	49
varsoc	81	69	*63*	*59*	46
varstable	100	100	*100*	*100*	0
vec	79	67	63	65	

All values are expressed as a percentage of the time required on a single processor. Slanted values are extrapolated from 4 processors.

a. Smaller is better; 50 is perfect for 2 processors, 25 is perfect for 4 processors, and 12.5 is perfect for 8 processors.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

command	Run time as percentage of single processor time[a]				Percentage parallelized[b]
	Number of processors				
	2	4	8	16	
veclmar	80	68	*63*	*60*	45
vecnorm	81	69	*63*	*60*	46
vecrank	82	69	*63*	*60*	46
vecstable	100	100	*100*	*100*	
vwls	58	34	*22*	*16*	90
wntestb	100	100	*100*	*100*	0
wntestq	93	91	*89*	*89*	10
xcorr	94	91	*90*	*90*	9
xtabond	92	87	*85*	*83*	20
xtabond, twostep	92	87	*85*	*84*	18
xtcloglog, re	48	26	*15*	*9*	96
xtdata, be	80	69	*63*	*60*	44
xtdata, fe	73	57	*49*	*45*	60
xtdata, re	75	60	*53*	*49*	56
xtfrontier	54	33	*23*	*17*	87
xtgee, family(gaussian) corr(ar2)	81	71	*65*	*63*	42
xtgee, fam(gauss) corr(unstruct)	82	71	*66*	*63*	42
xtcloglog, pa	66	45	*35*	*30*	77
xtlogit, pa	75	61	57	55	43
xtnbreg, pa	68	49	43	43	52
xtpoisson, pa	74	58	54	52	45
xtprobit, pa	75	60	58	56	38
xtreg, pa	81	70	68	66	32
xtgls	76	60	52	49	53
xthtaylor	65	43	*32*	*27*	80

All values are expressed as a percentage of the time required on a single processor. Slanted values are extrapolated from 4 processors.

a. Smaller is better; 50 is perfect for 2 processors, 25 is perfect for 4 processors, and 12.5 is perfect for 8 processors.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

command	Run time as percentage of single processor time[a] Number of processors 2	4	8	16	Percentage parallelized[b]
xtintreg	49	26	*14*	*8*	98
xtivreg, be	69	50	*41*	*36*	70
xtivreg, re	67	48	*38*	*34*	72
xtlogit, fe	63	45	33	27	78
xtlogit, re	47	27	*17*	*12*	92
xtmixed	99	99	99	99	0
xtmixed (crossed effects)	100	100	100	100	1
xtnbreg, fe	59	32	*19*	*13*	94
xtnbreg, re	51	27	*15*	*9*	97
xtpcse	92	87	85	86	10
xtpcse, corr(ar1)	100	100	*100*	*100*	0
xtpcse, corr(psar1)	95	92	*91*	*90*	11
xtpoisson, fe	63	40	*29*	*23*	83
xtpoisson, re	60	36	*24*	*18*	89
xtprobit, re	50	27	*15*	*9*	96
xtrc	71	58	*52*	*49*	53
xtreg, be	75	61	*54*	*51*	54
xtreg, fe	73	57	*49*	*45*	61
xtreg, mle	84	76	72	72	27
xtreg, re	74	58	*50*	*45*	61
xtregar, fe	74	59	*51*	*47*	59
xtregar, re	73	56	*48*	*44*	62
xtsum	66	47	*38*	*33*	73
xttab	90	84	*81*	*80*	23
xttobit	50	28	*16*	*11*	95

All values are expressed as a percentage of the time required on a single processor. Slanted values are extrapolated from 4 processors.

a. Smaller is better; 50 is perfect for 2 processors, 25 is perfect for 4 processors, and 12.5 is perfect for 8 processors.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

command	Run time as percentage of single processor time[a] — Number of processors: 2	4	8	16	Percentage parallelized[b]
zinb	52	28	16	11	94
zip	53	29	17	11	95
ztnb	53	29	18	13	91
ztp	56	31	19	15	94
_predict, xb	53	27	15	9	98
_rmcoll	53	29	15	8	99
_robust	100	100	*100*	*100*	0

All values are expressed as a percentage of the time required on a single processor. Slanted values are extrapolated from 4 processors.

a. Smaller is better; 50 is perfect for 2 processors, 25 is perfect for 4 processors, and 12.5 is perfect for 8 processors.

b. Bigger is better; 100 is perfect.

9 Performance variability across computing platforms

As discussed in sections 2 and 3, there are many reasons why multiprocessor performance will vary across computing platforms. Those reasons include differences in how operating systems partition tasks, how CPUs pipeline and partition instructions, how memory is accessed, and how onboard CPU cache is handled. Any of these reasons may cause performance to vary across platforms.

Stata/MP performance has been tested on dozens of different platforms, including different CPU chips (both Intel and AMD), different cache architectures, different operating systems (including Microsoft Windows, Mac OS X for Intel, Linux, and Sun Solaris), and different architectures for accessing memory. Despite the possibility for varying performance, the results from all these tests support the results presented in section 8 and appendices A and B.

To quantify this assertion, consider the benchmarks on which section 8 and appendix A are based. Among all the tests run on Stata/MP, this set of timings was collected on 10 architectures other than the one used for the table in section 8, including several MS Windows configurations, several Linux configurations, a Macintosh platform, a Sun platform, both AMD and Intel chips, and machines with anywhere from 2 to 16 processors. We can compare the run time for two processors as a percentage of the time for one processor (column 2 in table 1) across all these platforms. With 332 commands timed across 10 architectures, this makes for 3,320 timings. Well over 50% of those timings fell within ± 3 percentage points of those reported in column 2 of table 1, and 75% of all timings fell within ± 5 percentage points. At the tail of the distribution, just under 10% of timings fell outside ± 10 percentage points of column 2, and 5% fell outside ± 13 percentage points. Considered differently, the command performance gains from 1 to 2 processors were correlated above .9 for all 55 pairings of the 11 platforms. The reported results generalize well across computing platforms.

It is not beneficial to break these results down by platform. There were no conclusive patterns among operating systems, CPUs, or other platform characteristics.

10 Hyperthreading—single- and multiple-processor platforms

Hyperthreading is an Intel technology for allowing a CPU with a single core to masquerade as a dual-processor or dual-core CPU. The operating system and other applications see the CPU as having two processors and treat it just as they would a two-processor system. Intel achieves performance improvements primarily because main computer memory is slow compared with the processor and its onboard cache memory. When the thread of execution of one virtual process must wait for something from main memory, the thread for the other virtual process can execute. The effect is clearly not the same as having two processors, but for many applications, performance can be improved by treating a computer with a hyperthreaded CPU as a multiple-processor computer.

Stata/MP runs on hyperthreaded CPUs. Our setup recommendations and the implied performance gains depend on whether the computer has a single hyperthreaded CPU or multiple hyperthreaded CPUs.

Most Stata commands are computationally intense, and because hyperthreaded CPUs contain only a single floating-point coprocessor, gains were expected to be small for computers with a single hyperthreaded CPU. In reality, performance gains on such platforms was surprisingly good, though not near those of truly parallel computers.

To ease comparison with true multiprocessor or dual-core computers, figure 12 presents the now familiar boundary region and median performance. With only 2 processors (and the 2nd virtual), we are interested in the left and right ends of the graph. The performance boundary over all commands still includes both perfectly parallelized and completely nonparallelized. The median of all commands shows only about half the performance gain that would be expected of two real processors, but even so, half the commands run in less than 85% of the time required by a single, nonhyperthreaded processor.

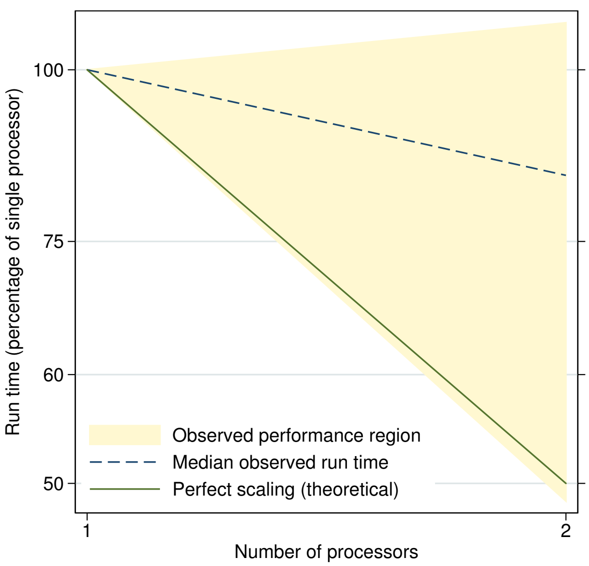

Figure 12. **Performance of Stata/MP on hyperthreaded processors.** Run times on multiple processors relative to a single processor.

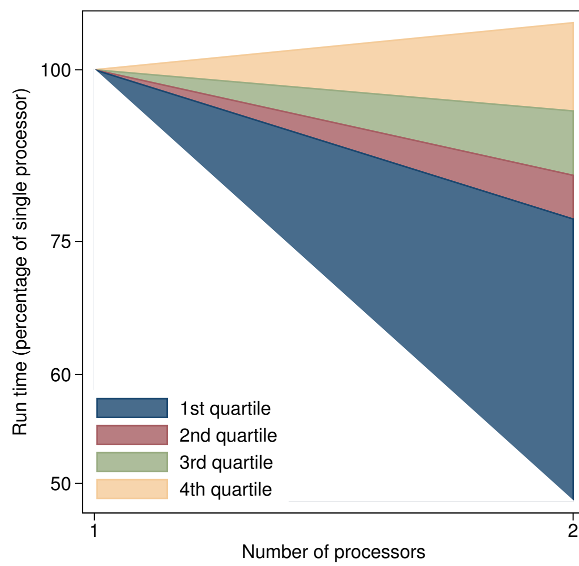

Figure 13. **Quartiles of Stata/MP Performance on hyperthreaded processors**. Run times on multiple processors relative to a single processor.

Figure 13 presents the quartiles of command performance. The most parallelized 25% of commands run in less than 79% of their time on a single, nonhyperthreaded processor, but there is less difference between the best 25% and the best 50% than on true multiprocessor.

This raises the question of which commands performed so well on a single hyperthreaded processor. The commands that ran in less that 79% of the time of a single, nonhyperthreaded processor were `canon`, `cluster kmeans`, `cluster kmedians`, `ctset`, `dfuller`, `generate`, `generate` (small expressions), `irf create`, `kdensity`, `markout`, `mds`, `mds long`, `pctile`, most predictions, `replace`, `runtest`, `summarize`, `svar`, `twoway mspline`, `xtcloglog, re`, `xtlogit, re`, and `xttobit`.

By way of caution, Stata/MP has not been evaluated on a wide range of single-processor hyperthreaded computers, and these results should therefore be considered provisional.

On multiprocessor computers where each CPU is hyperthreaded, the current recommendation is to set Stata/MP to use the number of real CPUs, not the number of virtual processors. Under such architectures, each CPU appears to Stata/MP and the operating system as two processors, and Stata/MP would by default try to use all the (virtual) processors. On these computers, users should type

```
. set procs_use #
```

where # is the number of CPUs on the computer.

This can be done either interactively or placed in Stata's `profile.do` startup script.

Current experience indicates that setting the number of processors to be used above the number of real CPUs on the computer leads to contention for the floating-point unit (FPU), which can make commands run slower when trying to use virtual processors.

Figures 14 and 15 show the results of two commands run on an 8-processor computer, each hyperthreaded, giving the appearance of 16 virtual processors.

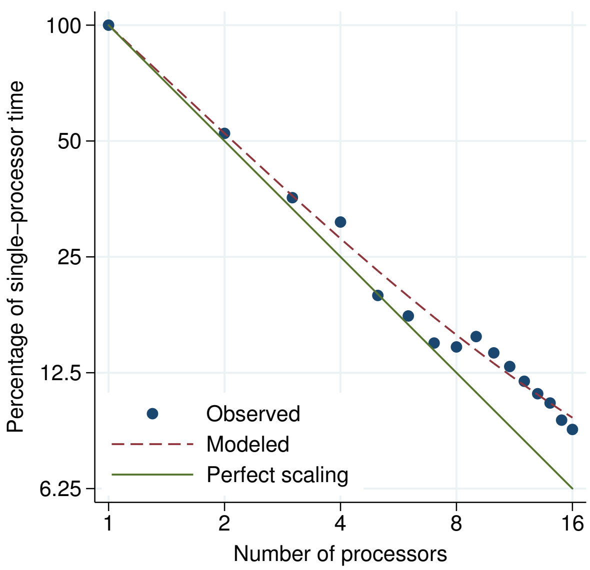

Figure 14. `by: replace` performance plot on computers with multiple hyperthreaded processors.

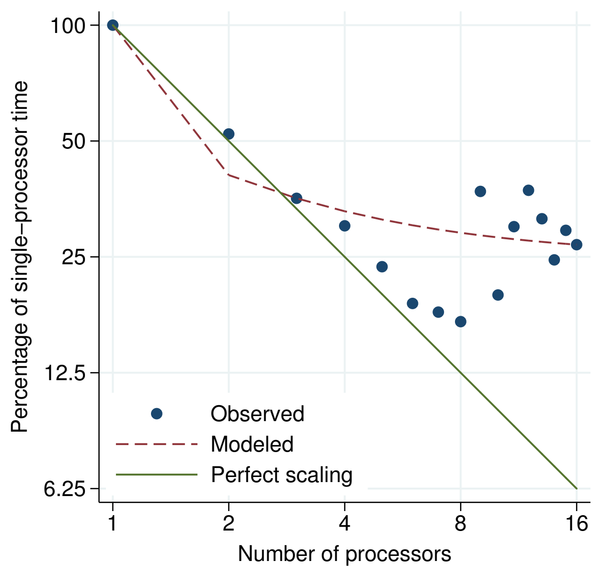

Figure 15. `weibull` performance plot on computers with multiple hyperthreaded processors.

The `by: replace` command, however, is an exception to this recommendation. Aside from a small uptick in going from 8 to 9 processors, the problem remains nearly perfectly parallelized through all 16 processors (half of which are virtual).

Most commands do not exhibit results like this, and `weibull` is an example. Beyond the number of real CPUs, performance actually degrades. This occurs because each CPU has only one FPU, and `weibull`, along with most Stata commands, requires many floating-point computations. The computations are dominated by access to the FPU, and the virtual processors must contend for access to this single FPU.

Consider this recommendation provisional until results have been obtained on more architectures that use multiple hyperthreaded CPUs.

A Performance assessment graphs

Below, the performance of Stata/MP as reported in columns 2 and 3 of the tables in section 8 is presented graphically along with the modeled performance from equation 1 and a line representing perfectly scalable performance.

Figures A.1 and A.2 show two typical graphs. As with the table in section 8, the performance is measured as the time required to execute the command as a percentage of the time required by a single processor.

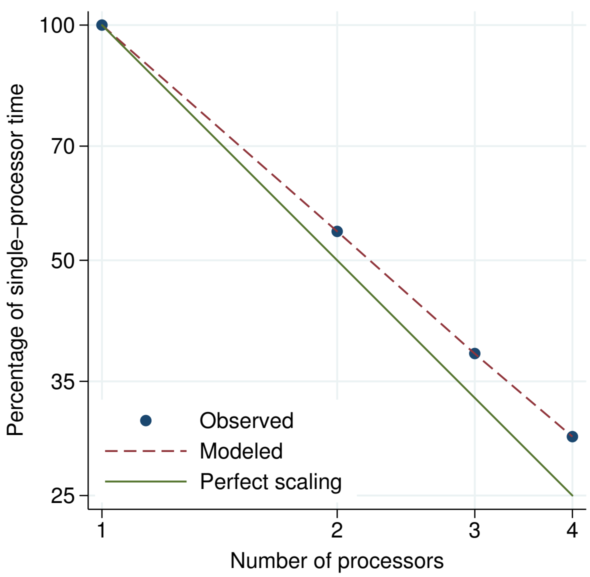

Figure A.1. `regress` performance plot.

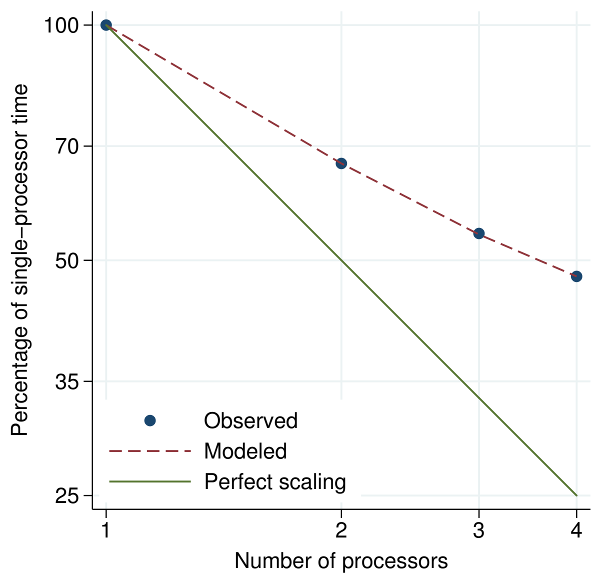

Figure A.2. `xtreg, re` performance plot.

For a perfectly scalable command, the percentage time will be halved each time the number of processors is doubled. This type of scalability is linear when the number of processors and percentage time are graphed on a logarithmic scale, and that is the scale used in these graphs. Perfect scaling is shown on the graph as a dashed green line that diagonally bisects the graph.

Linear regression, figure A.1, is nearly perfectly scalable. Both the observed values and the modeled performance are barely above the reference line. The run time is nearly halved each time the number of processors is doubled.

As shown in figure A.2, regression with random effects (random intercepts) clearly performs better as the number of processors is increased, but not as much as linear regression. From table 1, we can see that `xtreg, re` is 61% parallelized as compared with 97% for linear regression. From the graph, we see that with 2 processors `xtreg` run on a large dataset runs in just under 75% of the time of one processor, and with 4 processors this falls to just under half the time.

Figure 7 from section 7 summarizes the information from all these graphs by placing the observed performance for each command into one of the performance quartiles on the graph.

There are three or four graphs in what follows where the modeled line turns down, suggesting that at some point, increasing the number of processors accelerates the improvements in run times and would ultimately result in run times' being negative. This is nothing more than poor model fit to few observations.

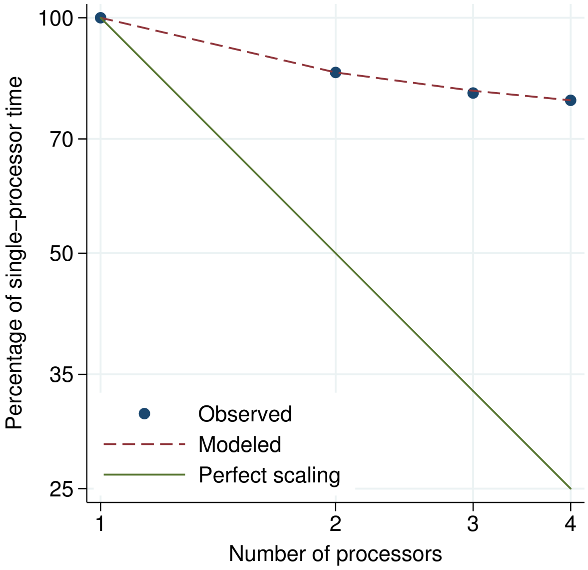

Figure A.3. `adjust` performance plot.

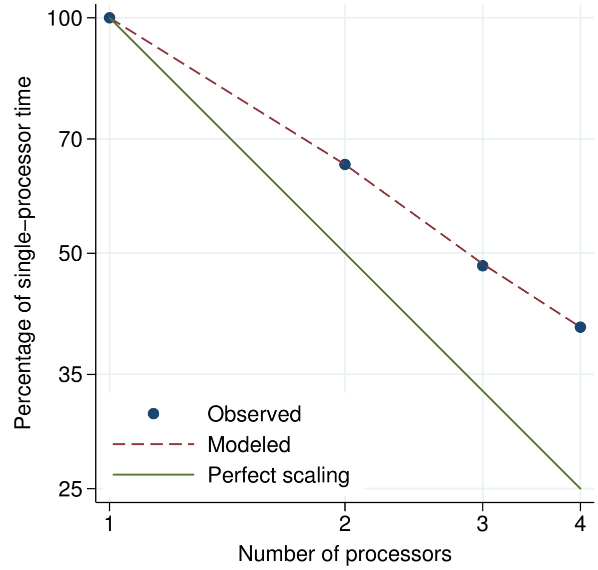

Figure A.4. `alpha` performance plot.

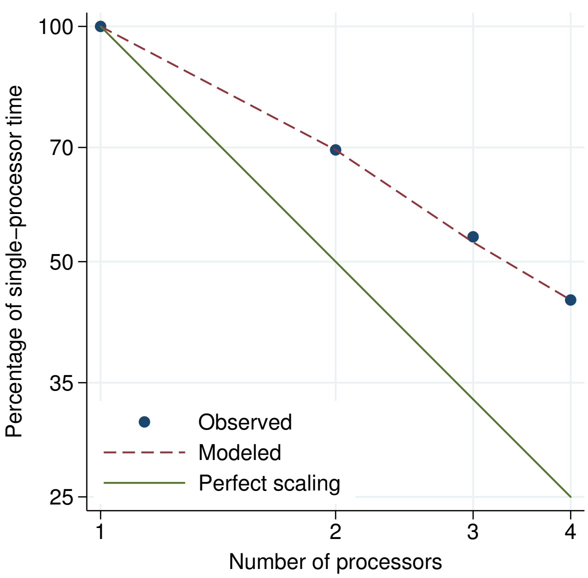

Figure A.5. `ameans` performance plot.

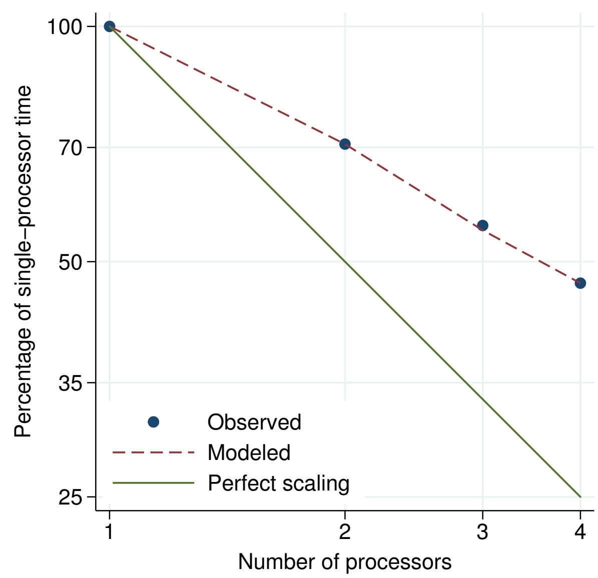

Figure A.6. `anova (oneway)` performance plot.

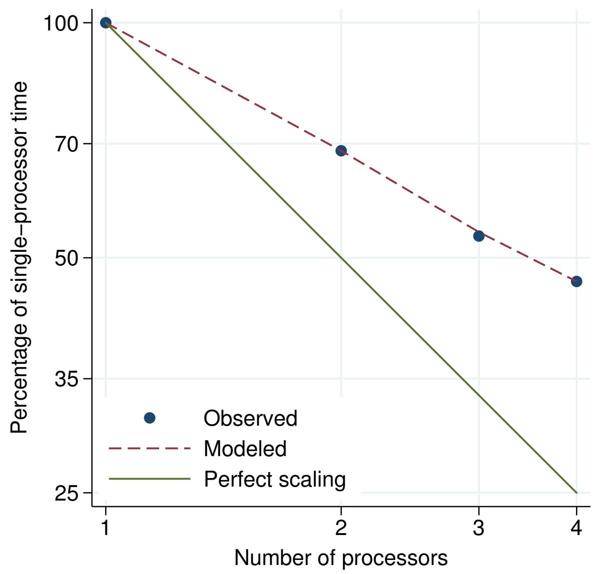

Figure A.7. `anova (twoway)` performance plot.

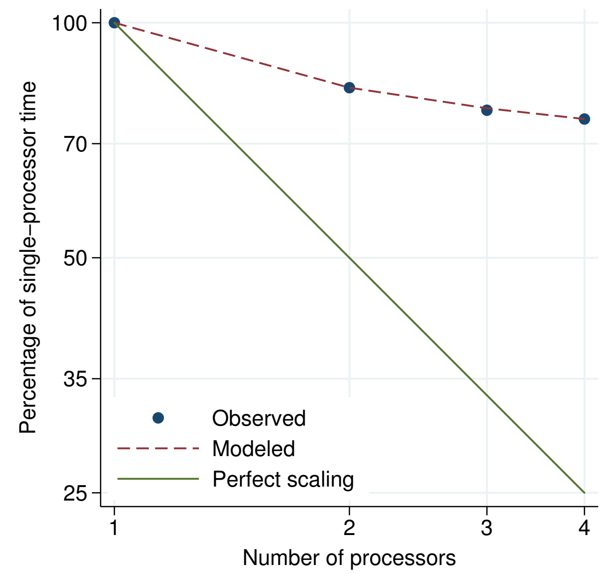

Figure A.8. `arch` performance plot.

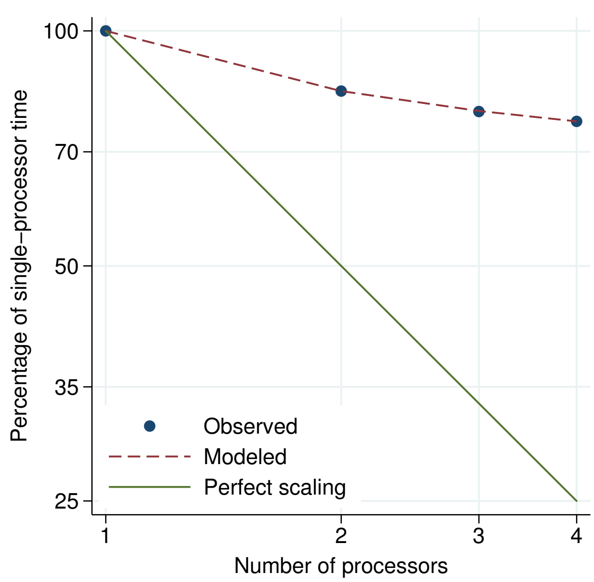

Figure A.9. `areg` performance plot.

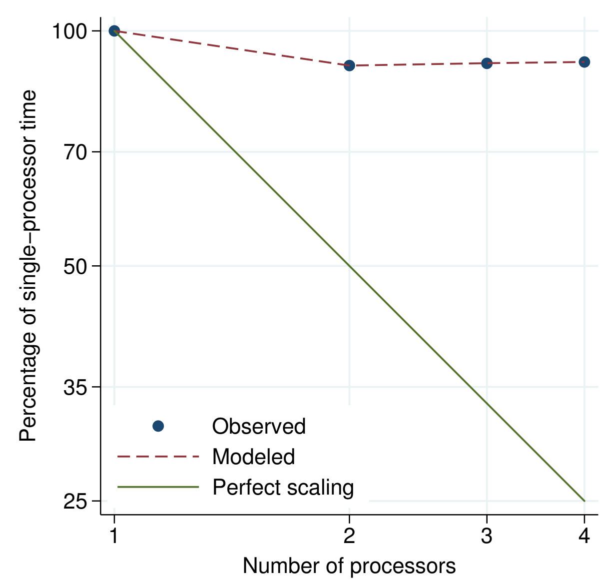

Figure A.10. `arima` performance plot.

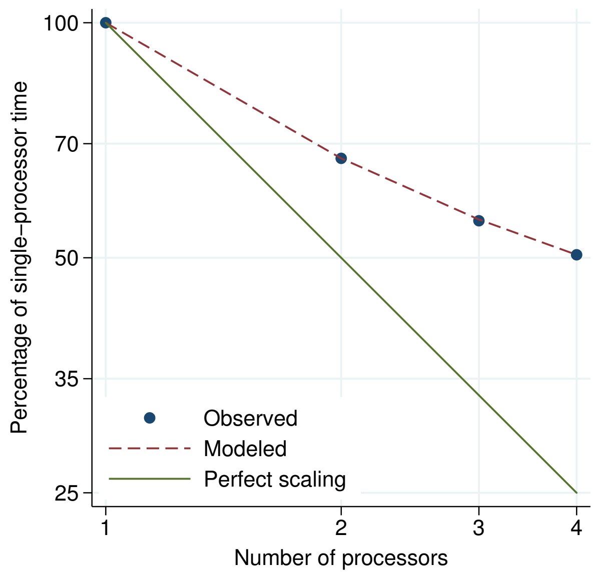

Figure A.11. `asmprobit` performance plot.

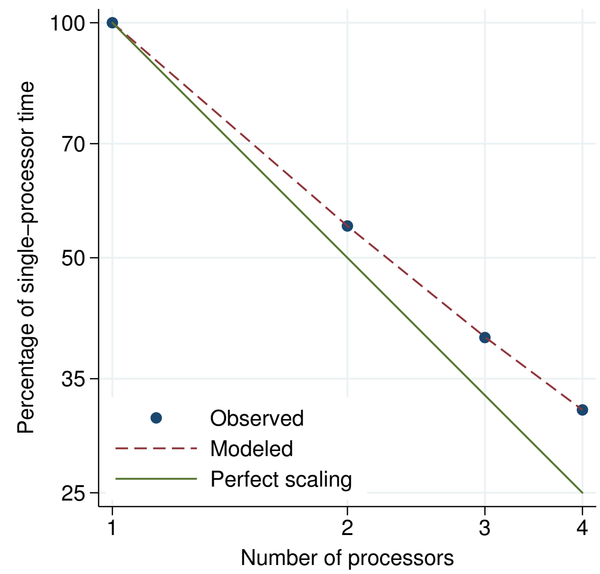

Figure A.12. `binreg` performance plot.

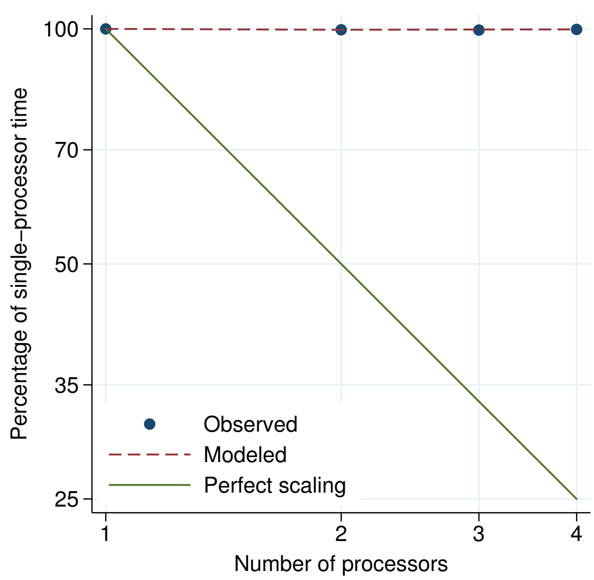

Figure A.13. `biplot` performance plot.

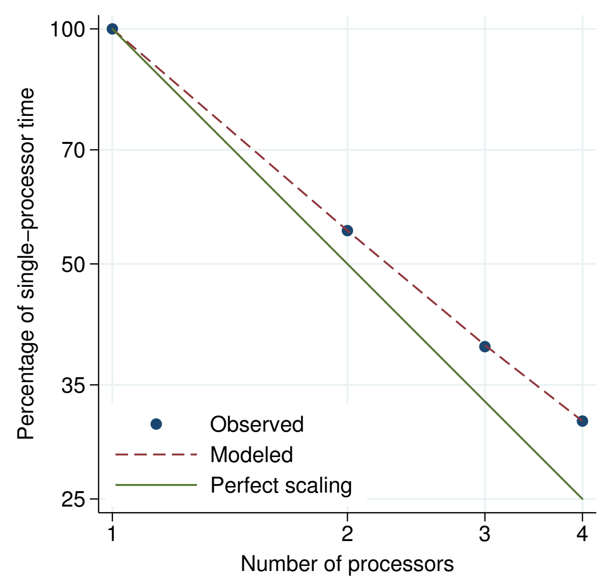

Figure A.14. `biprobit` performance plot.

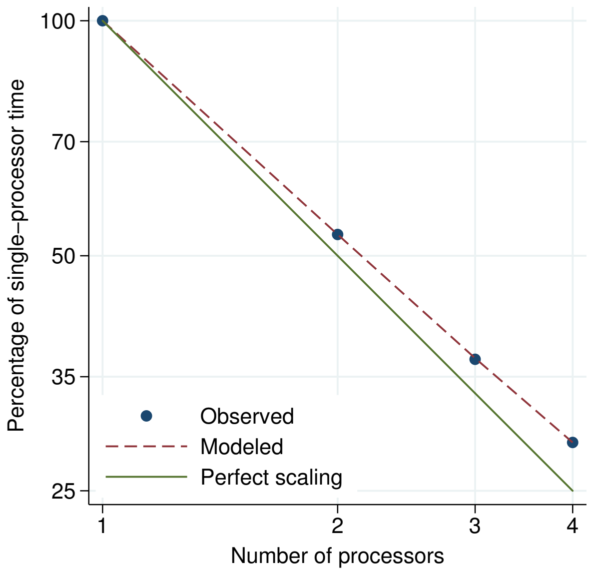

Figure A.15. `biprobit (seemingly unrelated)` performance plot.

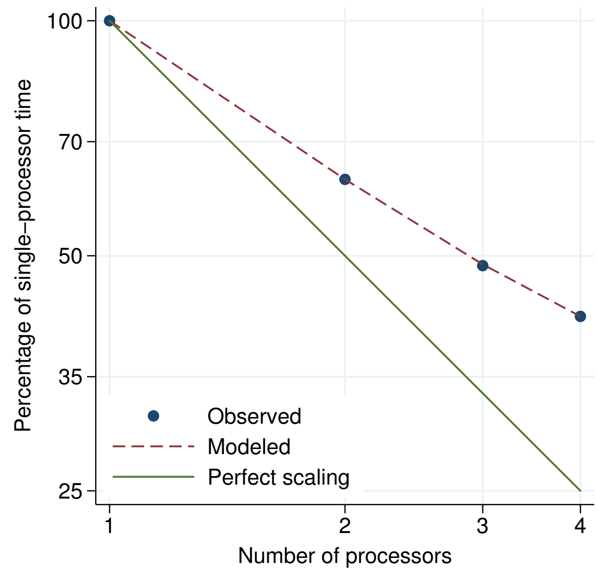

Figure A.16. `bitest` performance plot.

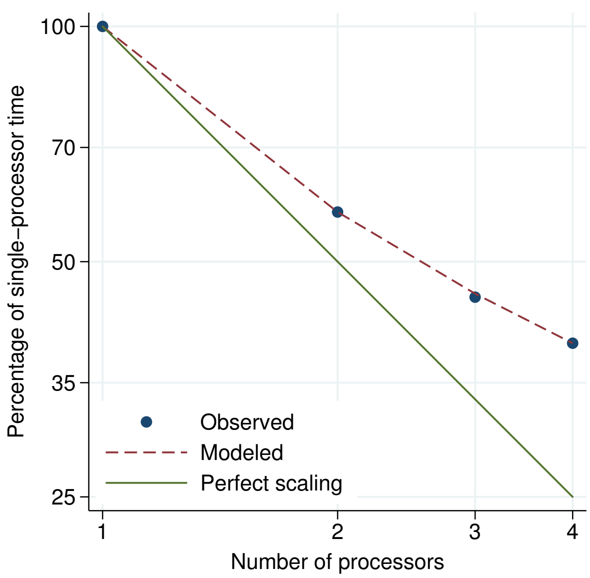

Figure A.17. `blogit` performance plot.

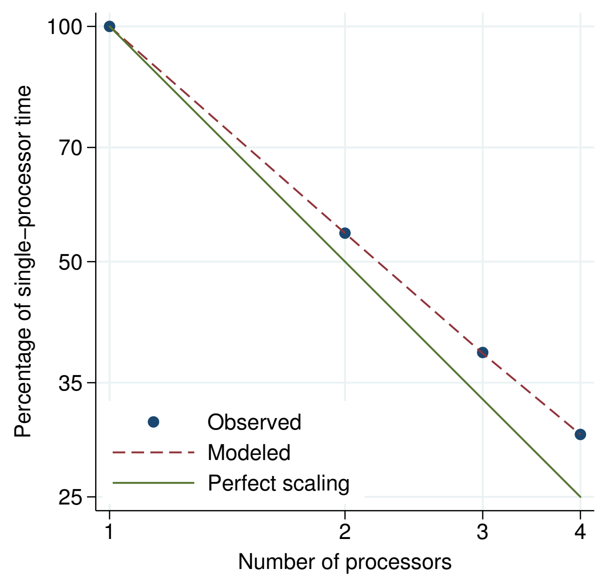

Figure A.18. `boxcox` performance plot.

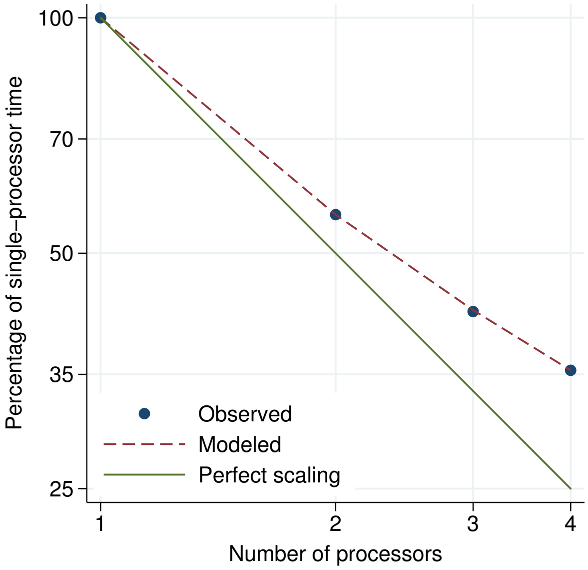

Figure A.19. `bprobit` performance plot.

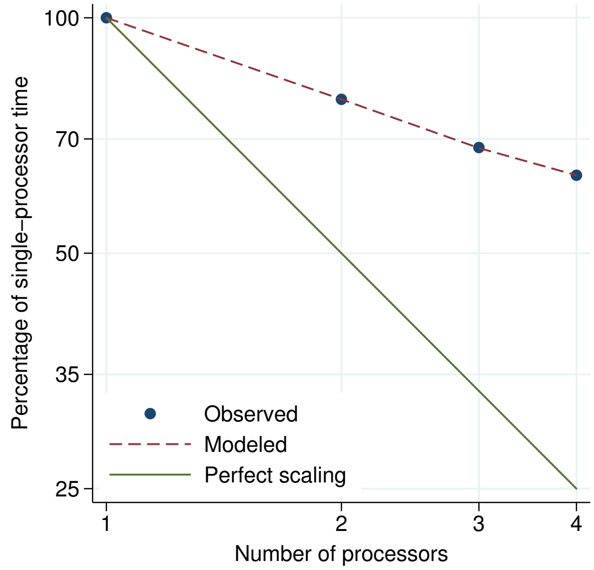

Figure A.20. `brier` performance plot.

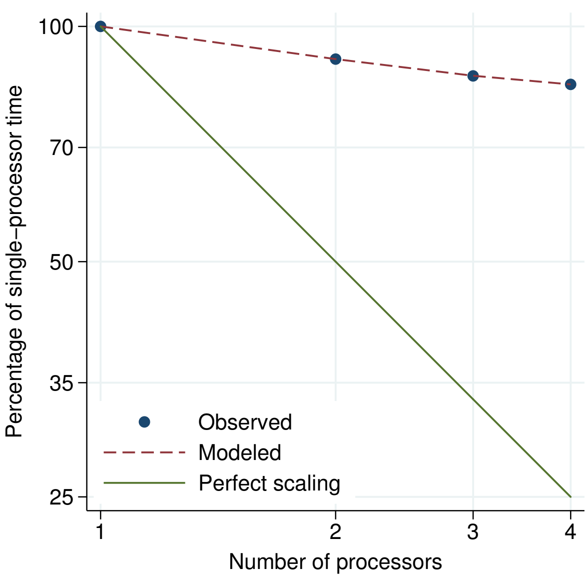

Figure A.21. `bsample` performance plot.

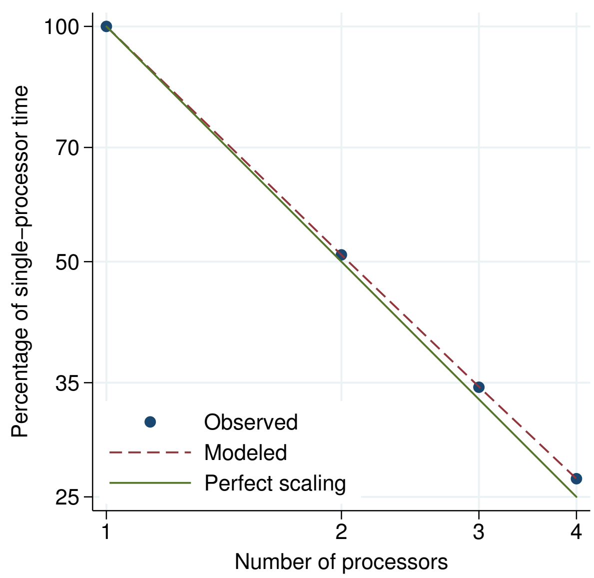

Figure A.22. `by: generate` performance plot.

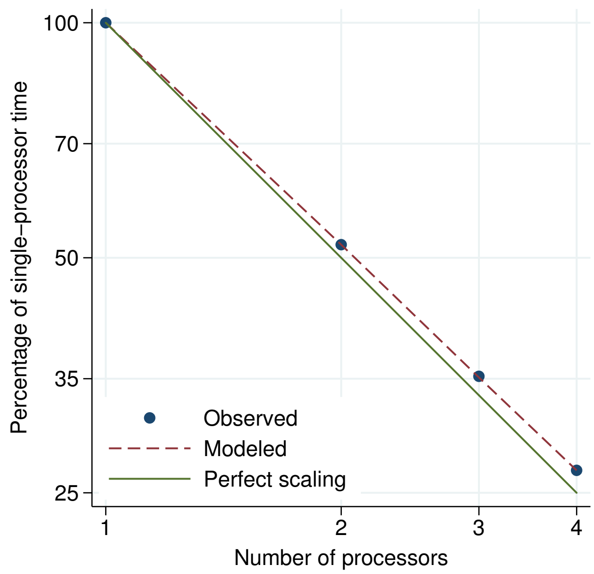

Figure A.23. `by: generate (small groups)` performance plot.

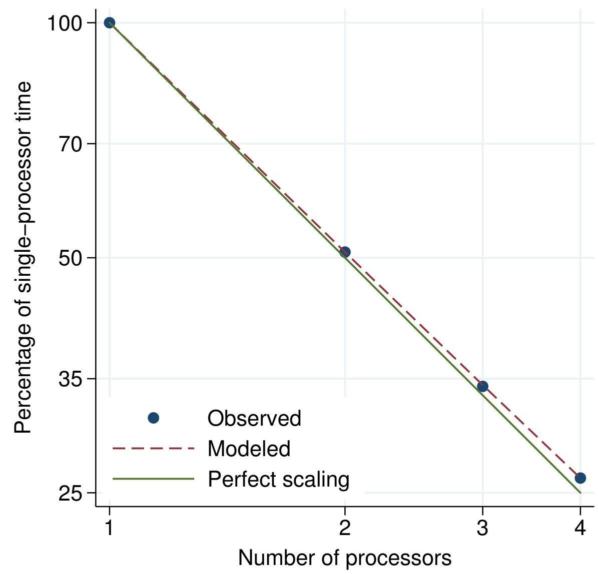

Figure A.24. `by: replace` performance plot.

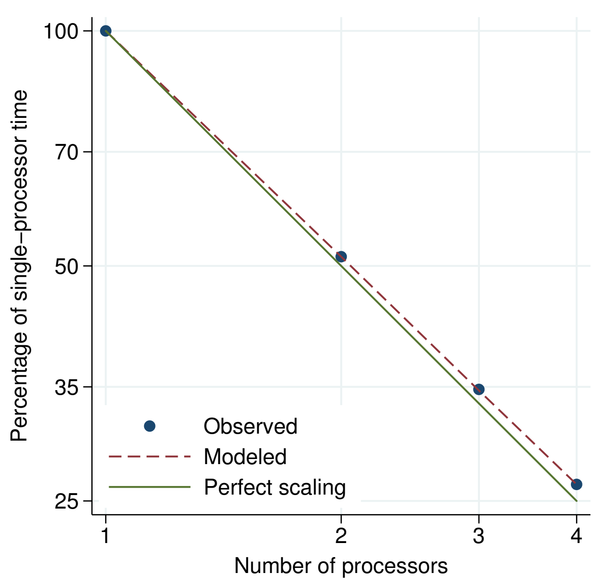

Figure A.25. `by: replace (small groups)` performance plot.

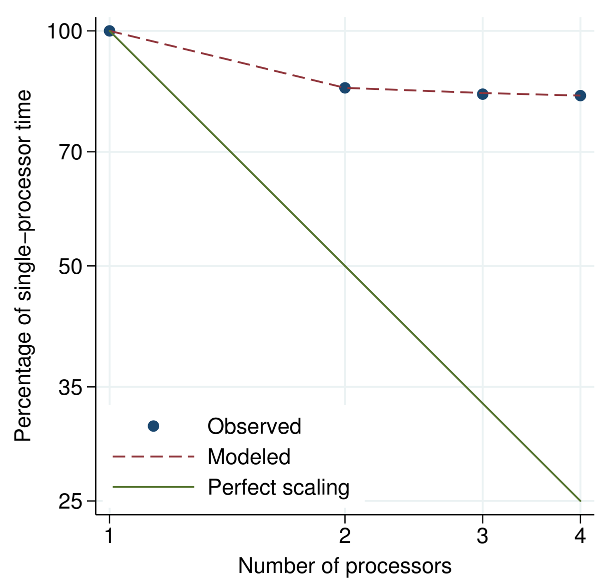

Figure A.26. `ca` performance plot.

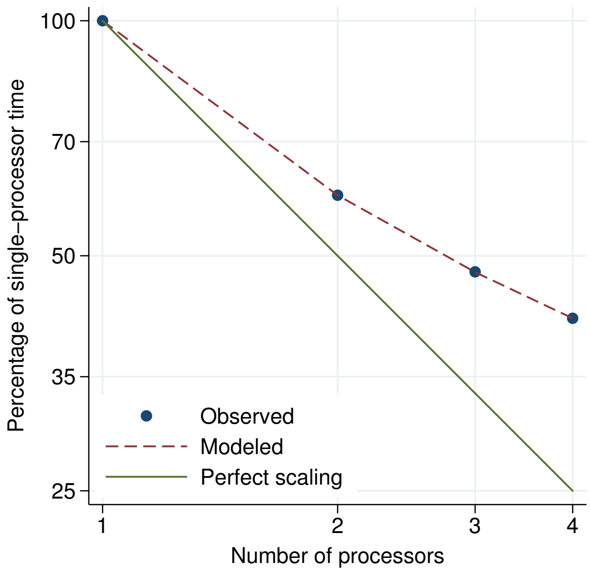

Figure A.27. `canon` performance plot.

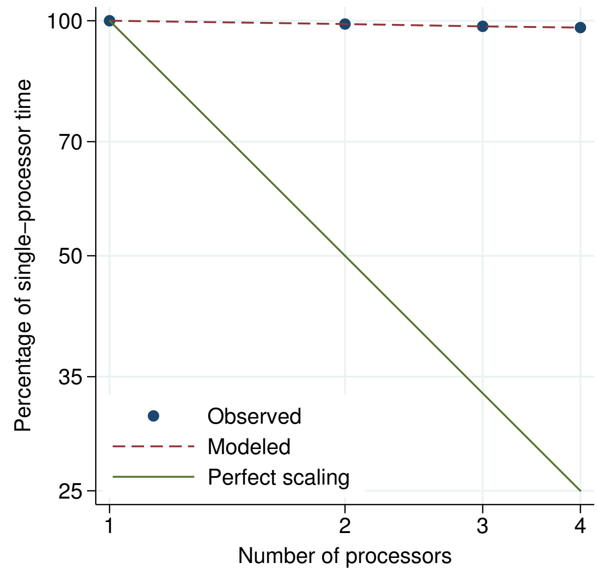

Figure A.28. `centile` performance plot.

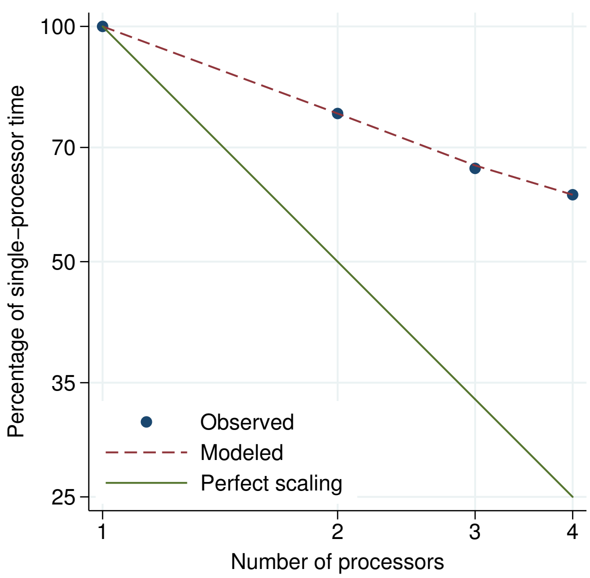

Figure A.29. `ci` performance plot.

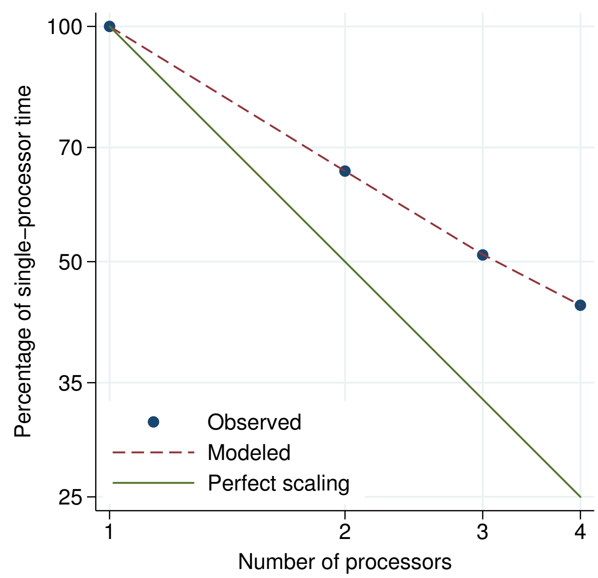

Figure A.30. `ci, binomial` performance plot.

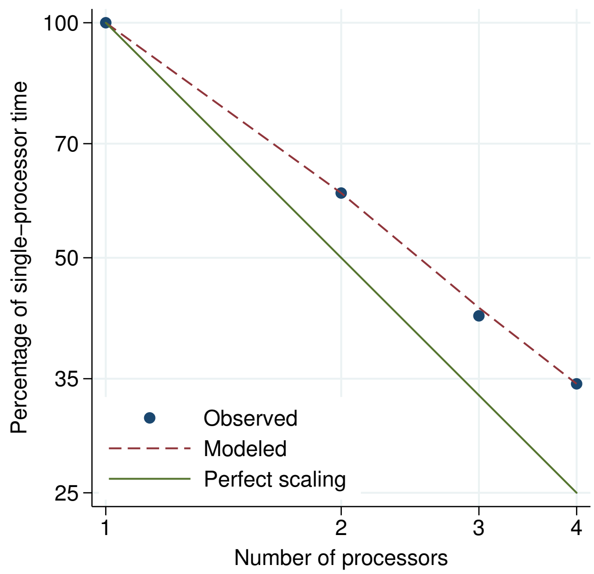

Figure A.31. `ci, poisson` performance plot.

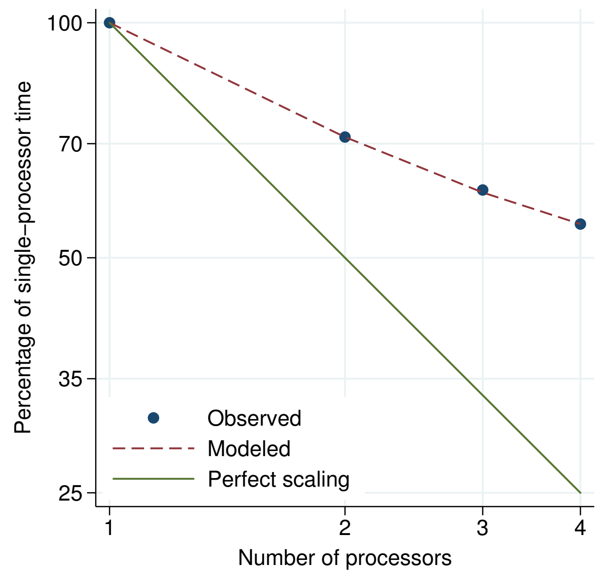

Figure A.32. `clogit (k1 to k2 matching)` performance plot.

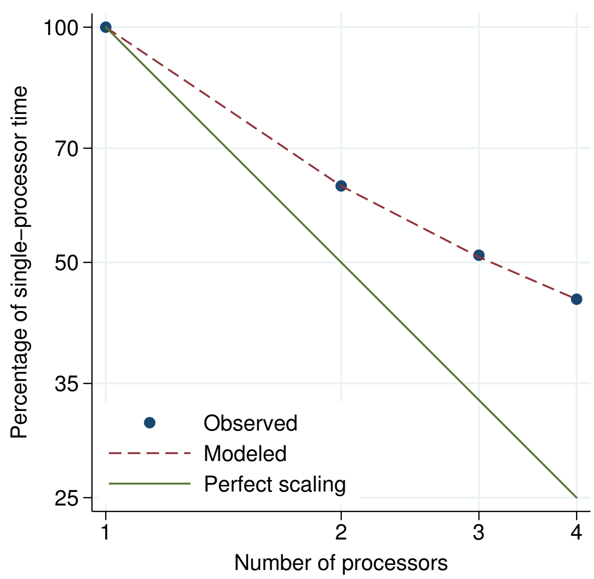

Figure A.33. `clogit (1 to k matching)` performance plot.

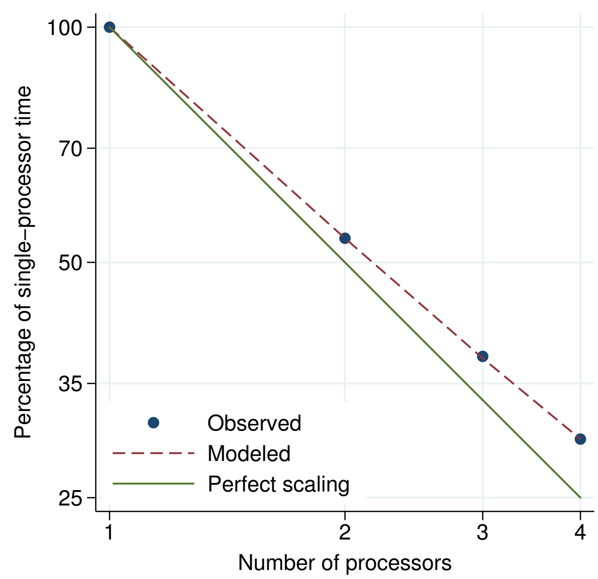

Figure A.34. `cloglog` performance plot.

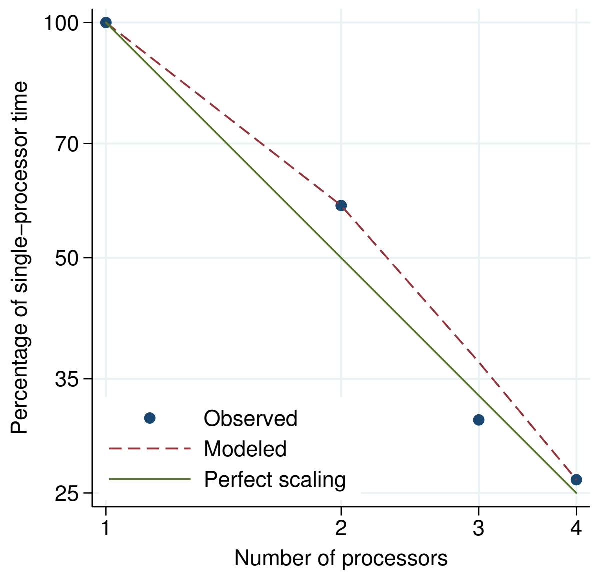

Figure A.35. `cluster averagelinkage` performance plot.

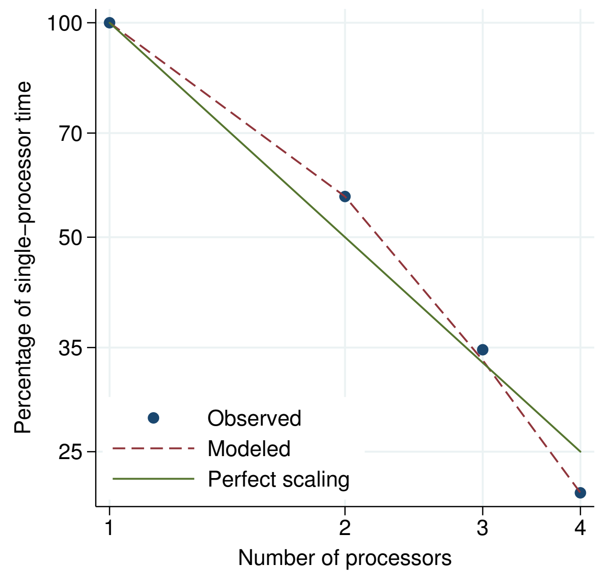

Figure A.36. `cluster centroidlinkage` performance plot.

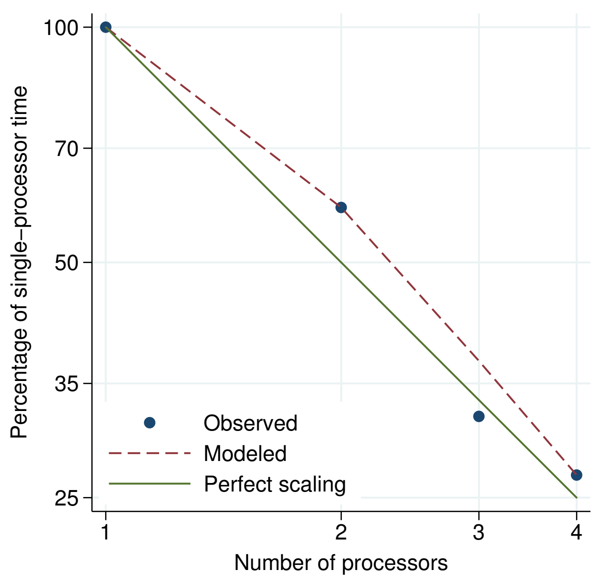

Figure A.37. `cluster completelinkage` performance plot.

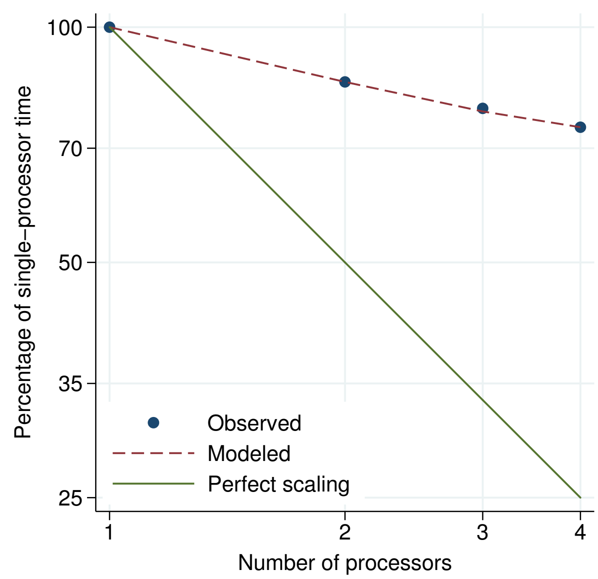

Figure A.38. `cluster generate` performance plot.

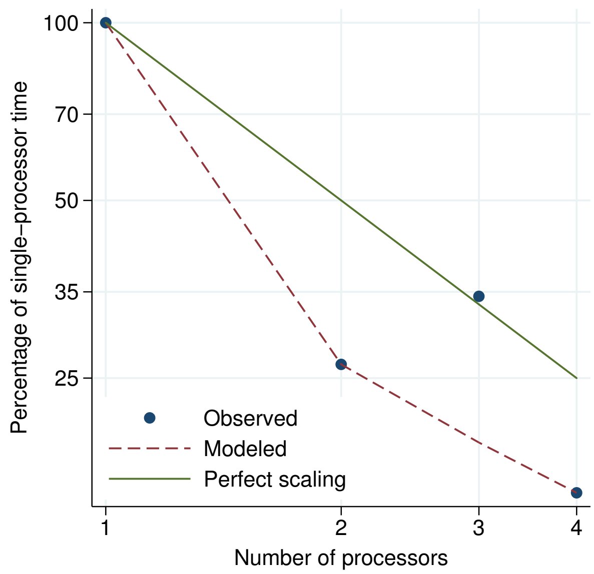

Figure A.39. `cluster kmeans` performance plot.

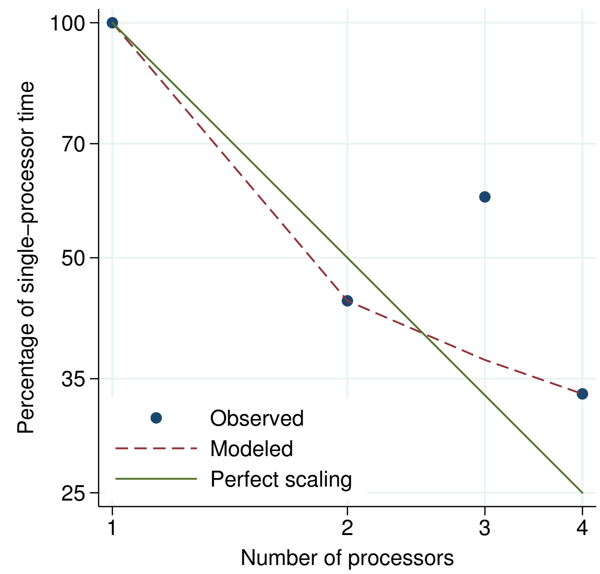

Figure A.40. `cluster kmedians` performance plot.

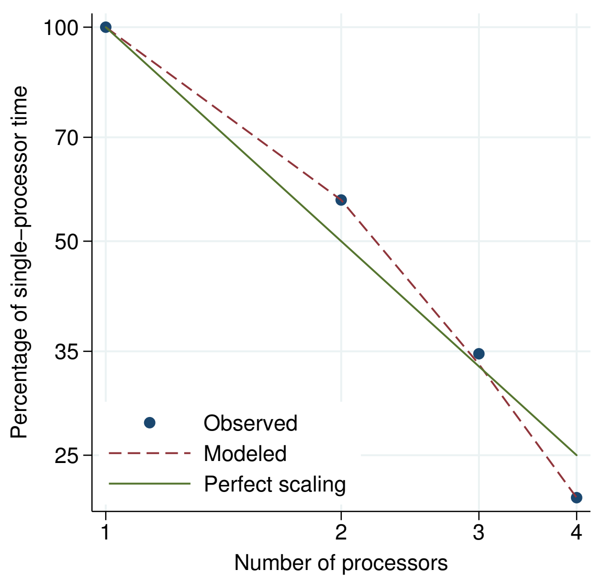

Figure A.41. `cluster medianlinkage` performance plot.

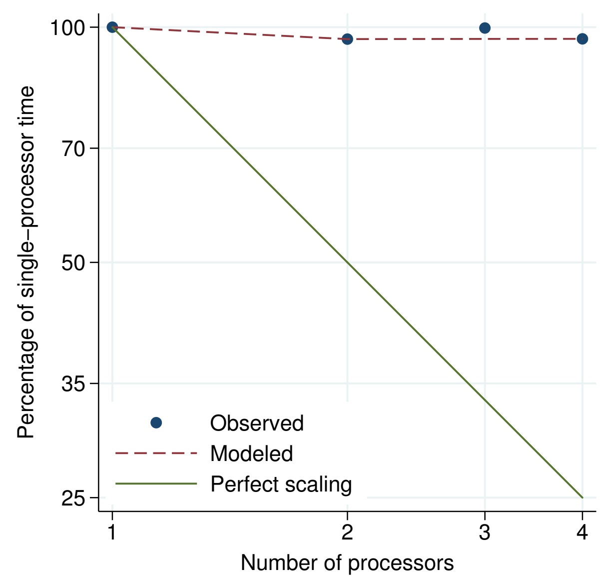

Figure A.42. `cluster singlelinkage` performance plot.

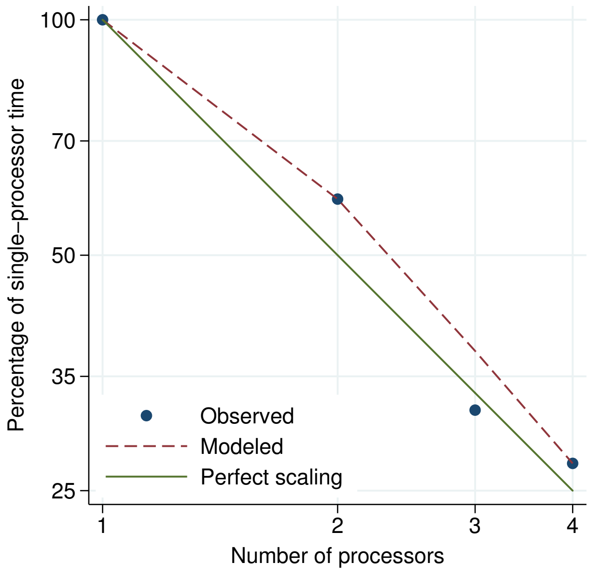

Figure A.43. `cluster wardslinkage` performance plot.

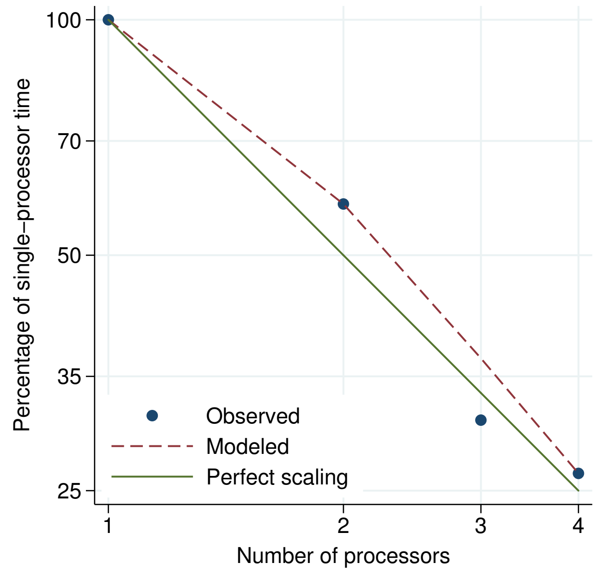

Figure A.44. `cluster waveragelinkage` performance plot.

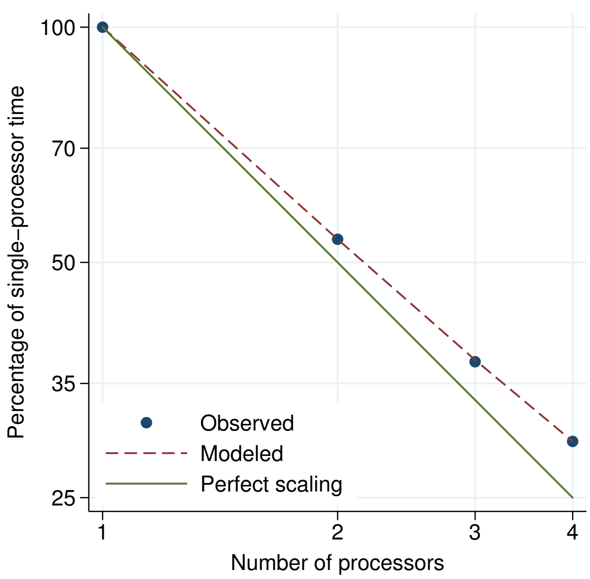

Figure A.45. `cnreg` performance plot.

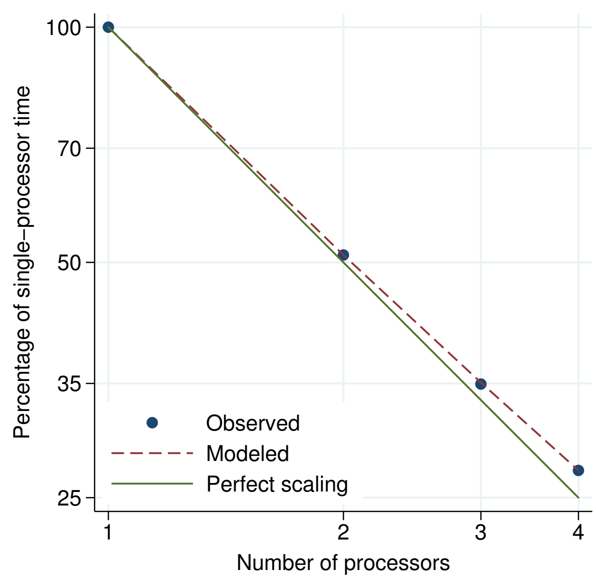

Figure A.46. `cnsreg` performance plot.

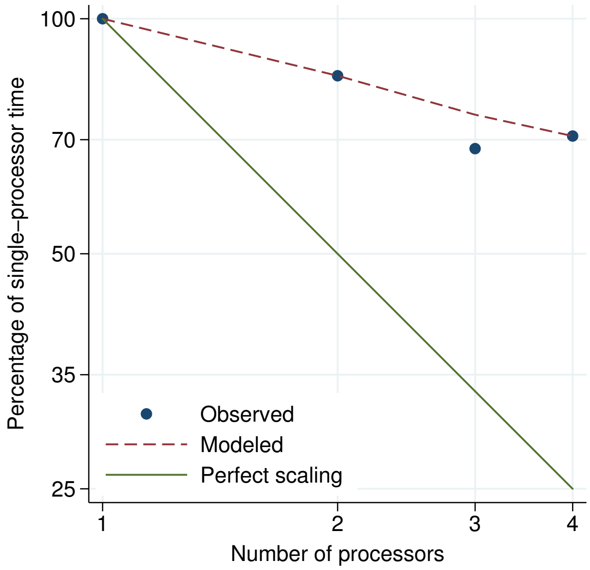

Figure A.47. `collapse` performance plot.

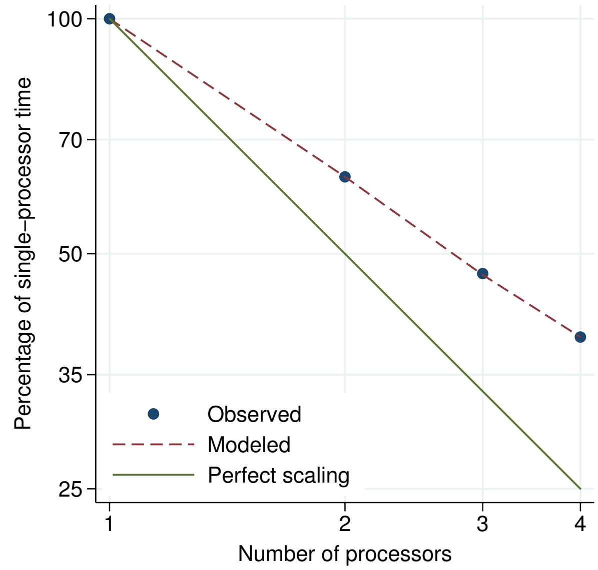

Figure A.48. `compare` performance plot.

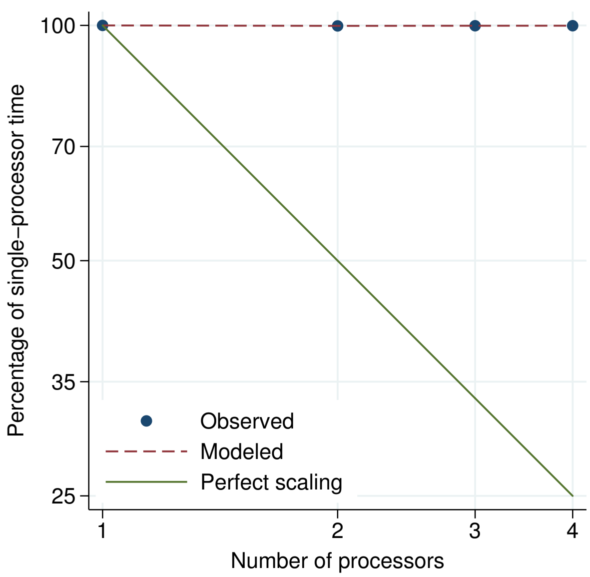

Figure A.49. `compress` performance plot.

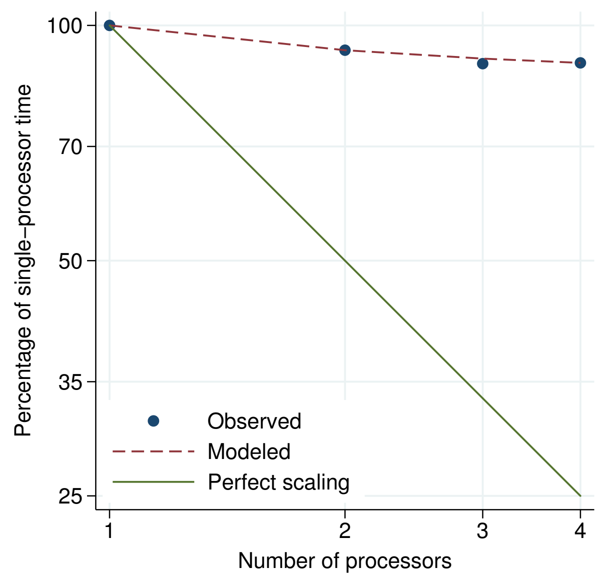

Figure A.50. `contract` performance plot.

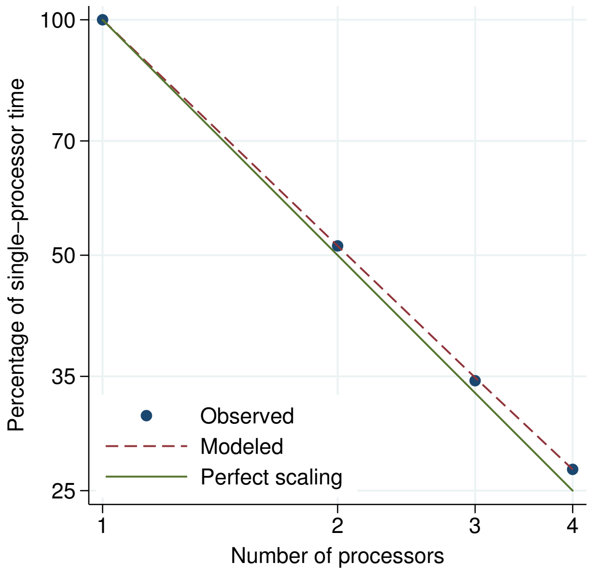

Figure A.51. `correlate` performance plot.

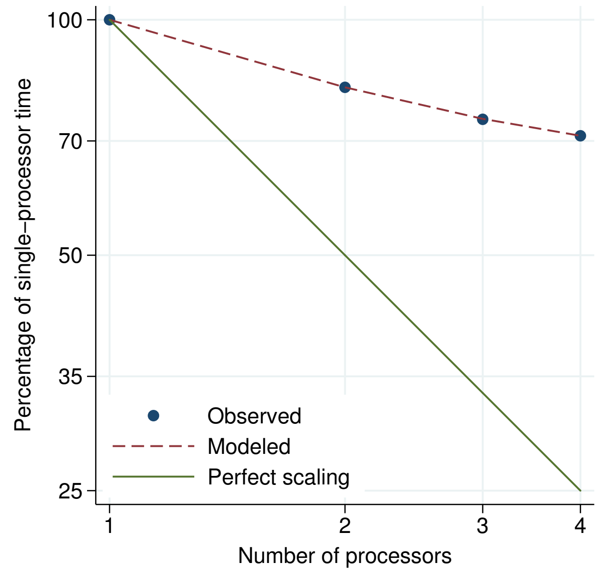

Figure A.52. `corrgram` performance plot.

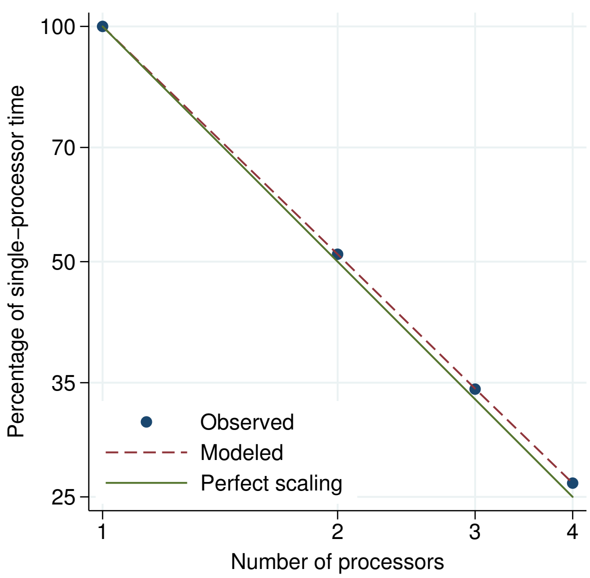

Figure A.53. `count` performance plot.

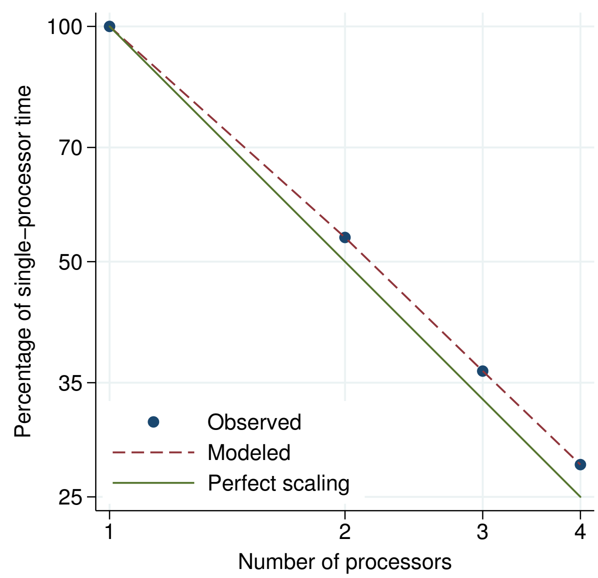

Figure A.54. `ctset` performance plot.

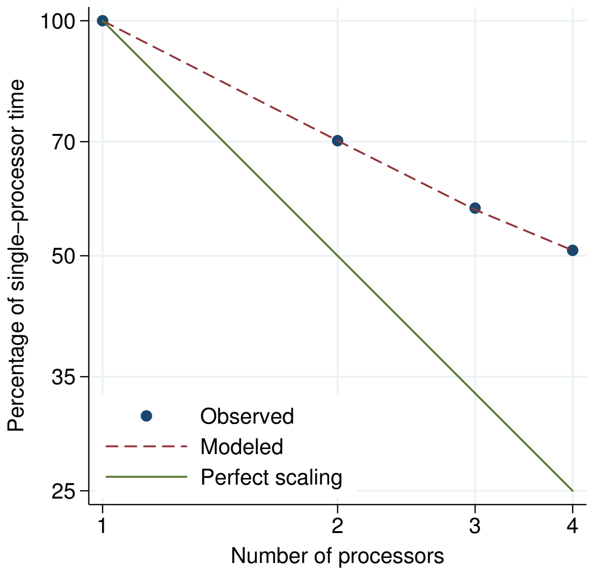

Figure A.55. `cttost` performance plot.

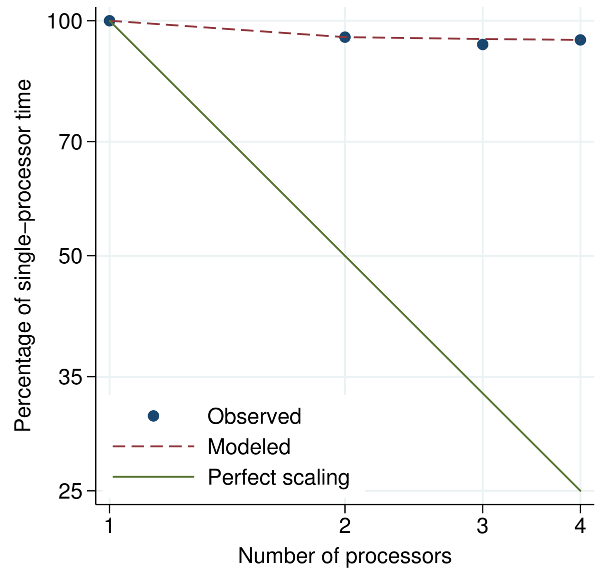

Figure A.56. `cumul` performance plot.

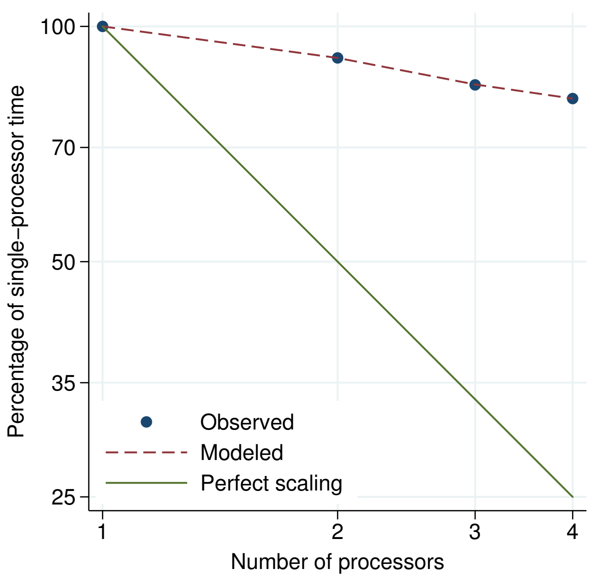

Figure A.57. `cusum` performance plot.

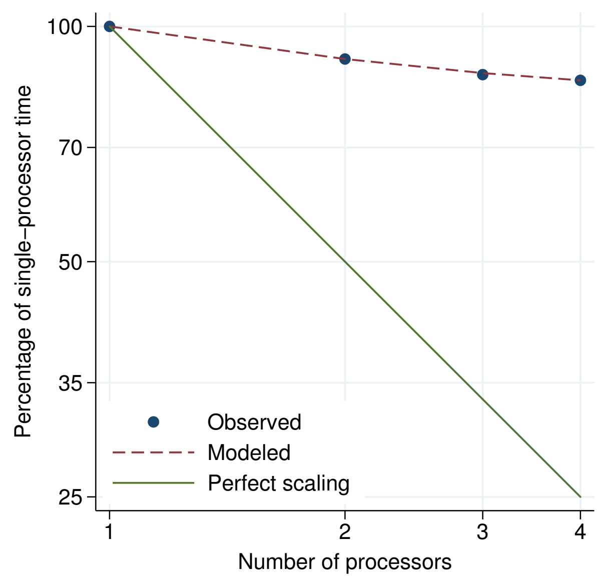

Figure A.58. `dfgls` performance plot.

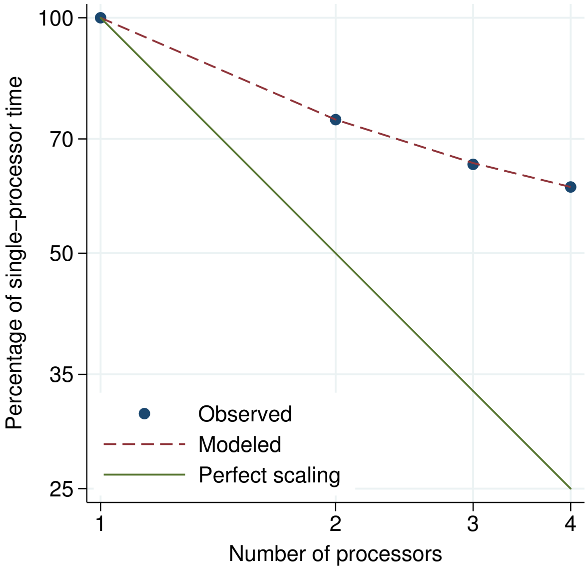

Figure A.59. `dfuller` performance plot.

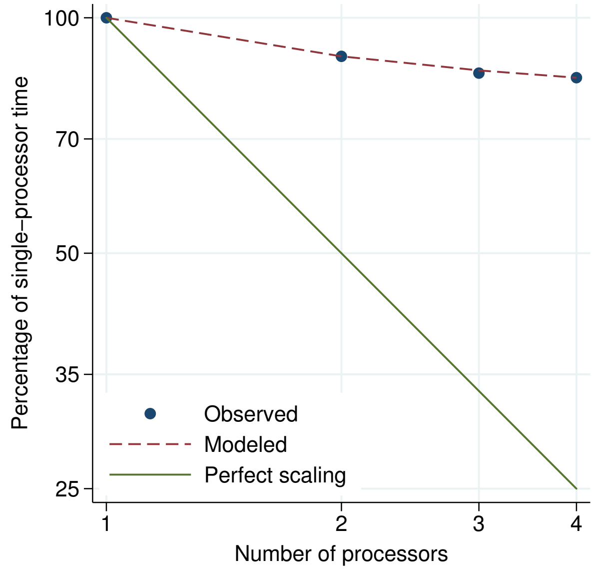

Figure A.60. `dotplot` performance plot.

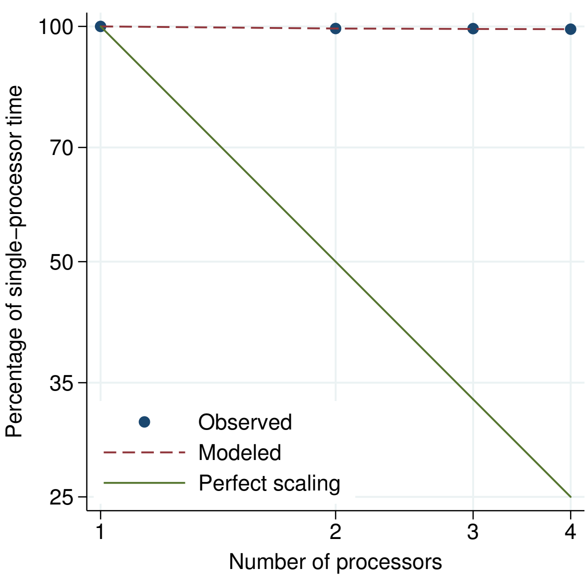

Figure A.61. `dstdize` performance plot.

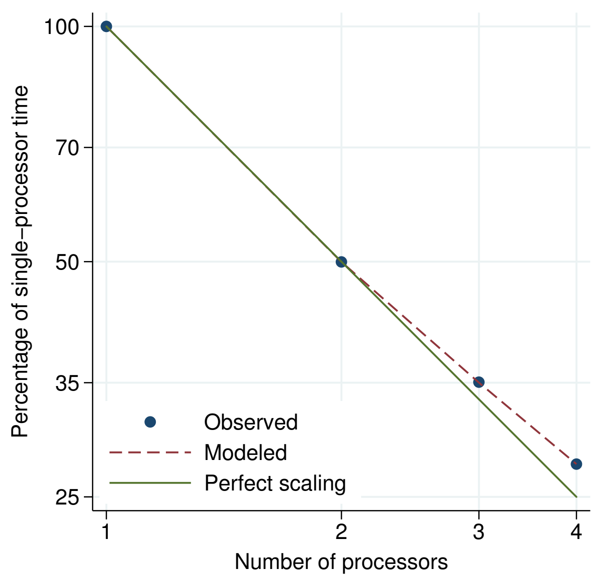

Figure A.62. `eivreg` performance plot.

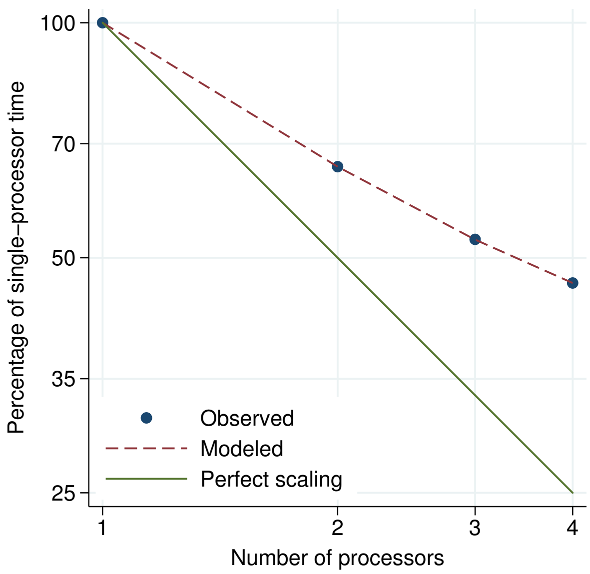

Figure A.63. `factor` performance plot.

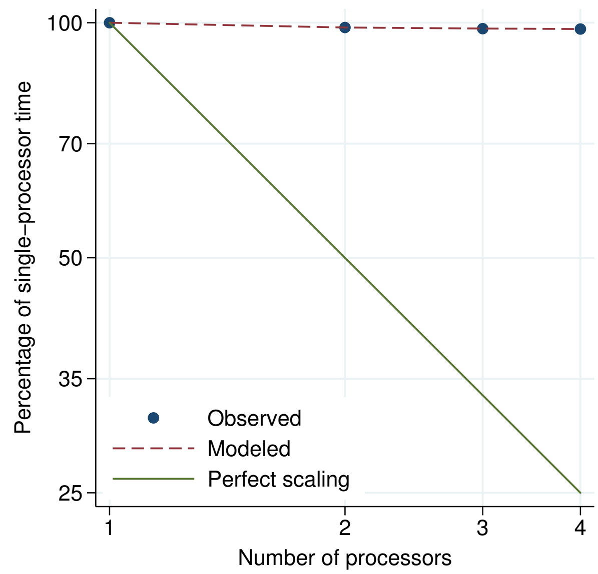

Figure A.64. `fcast compute` performance plot.

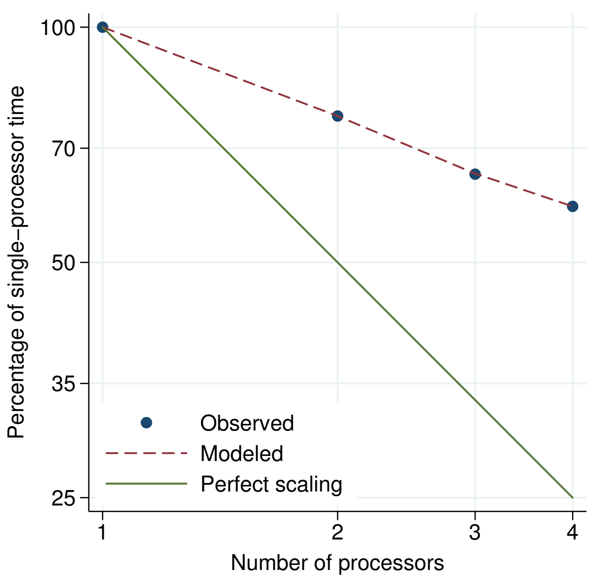

Figure A.65. `fracpoly` performance plot.

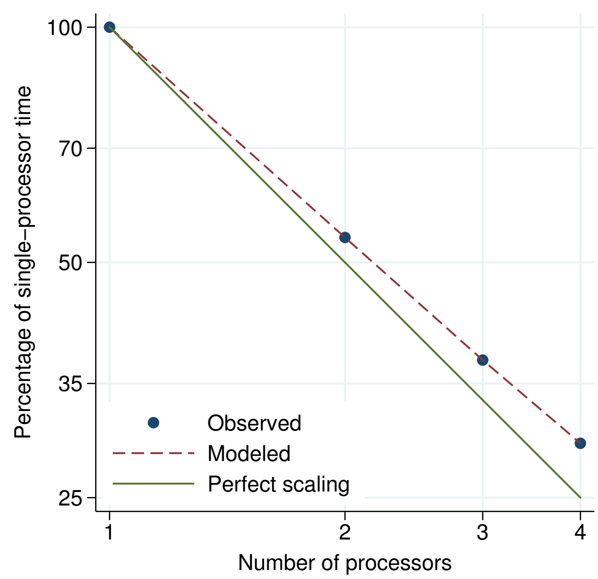

Figure A.66. `frontier` performance plot.

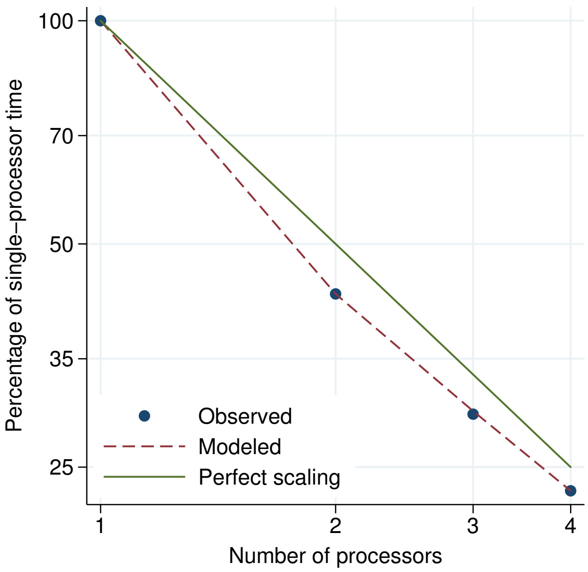

Figure A.67. `gen (small expressions)` performance plot.

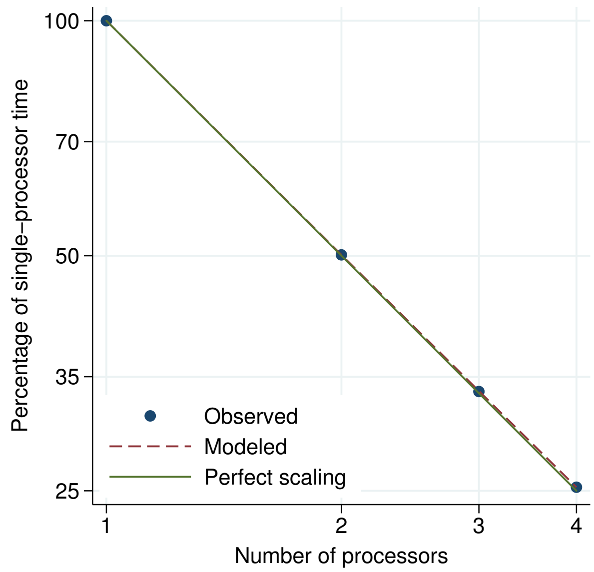

Figure A.68. `generate` performance plot.

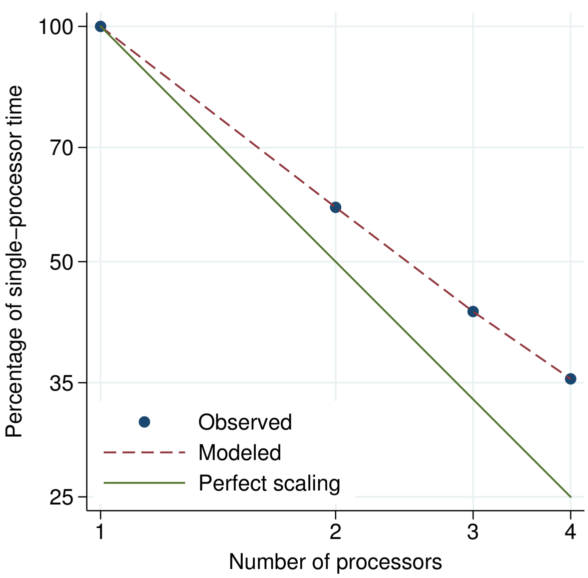

Figure A.69. `glm, family(gamma)` performance plot.

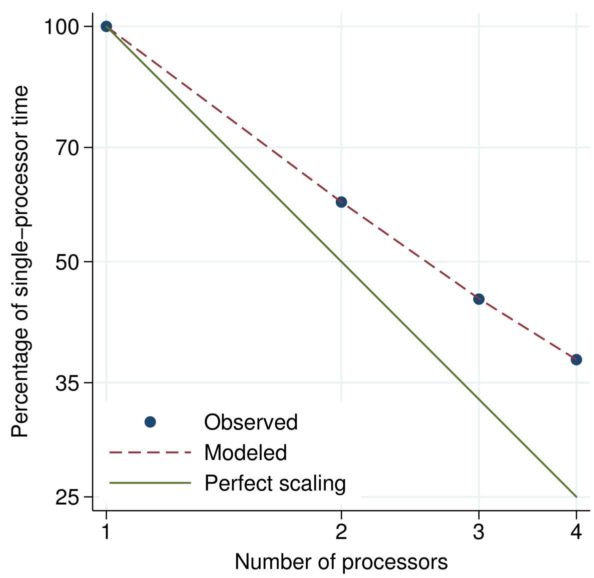

Figure A.70. `glm, family(gaussian)` performance plot.

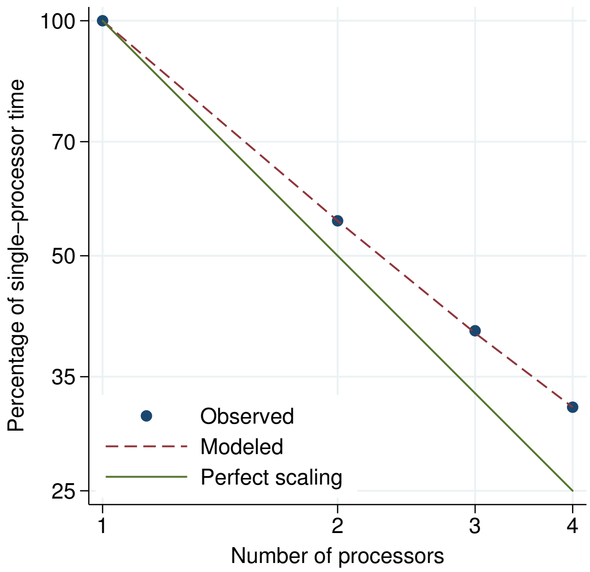

Figure A.71. `glm, family(igaussian)` performance plot.

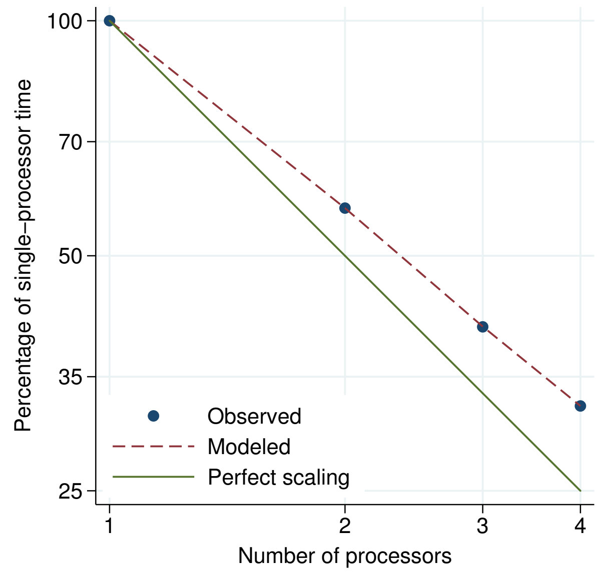

Figure A.72. `glm, family(nbinomial)` performance plot.

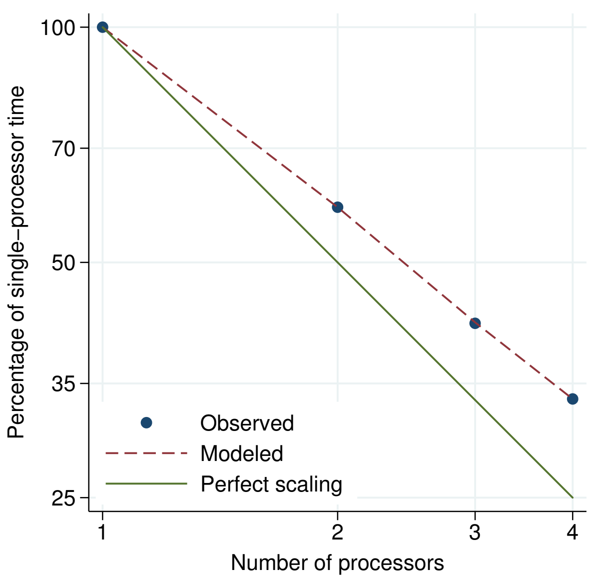

Figure A.73. `glm, family(poisson)` performance plot.

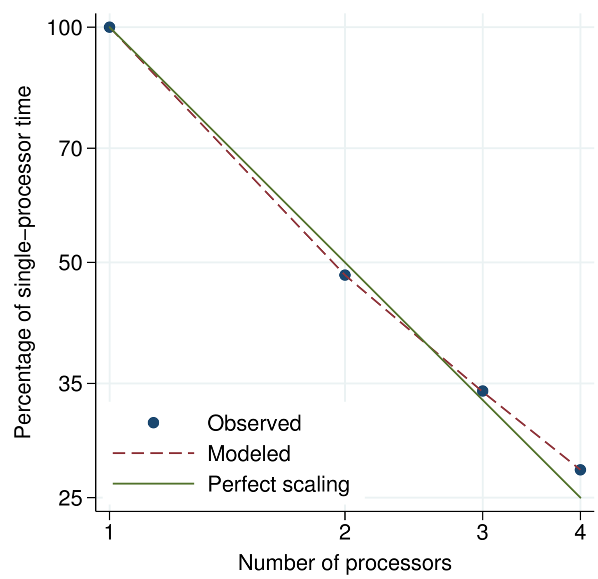

Figure A.74. `glogit` performance plot.

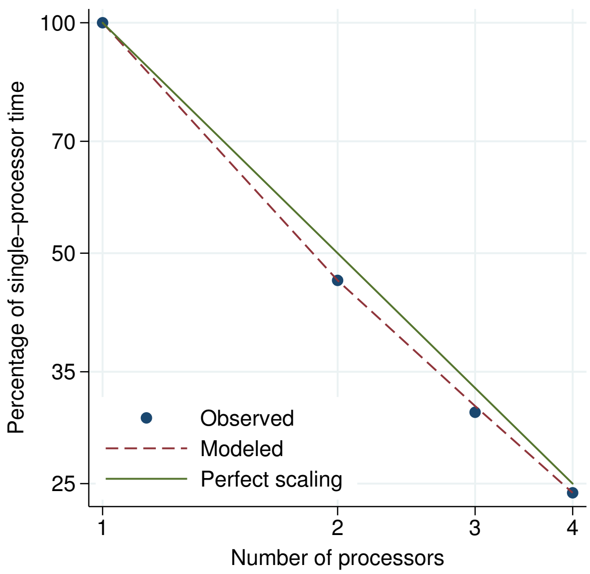

Figure A.75. `gprobit` performance plot.

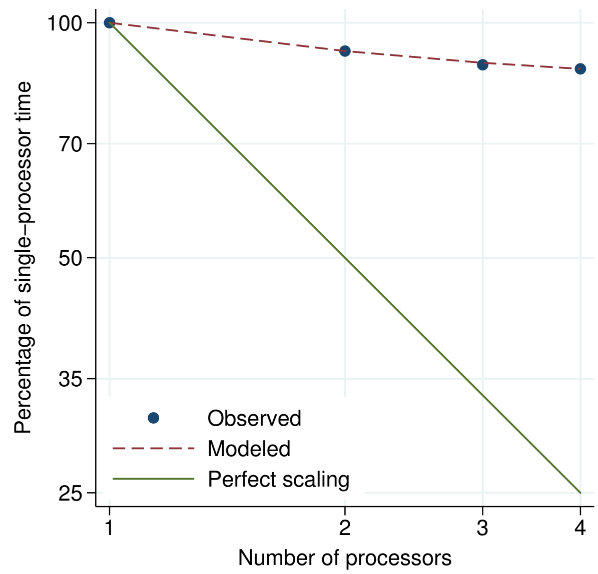

Figure A.76. `graph bar` performance plot.

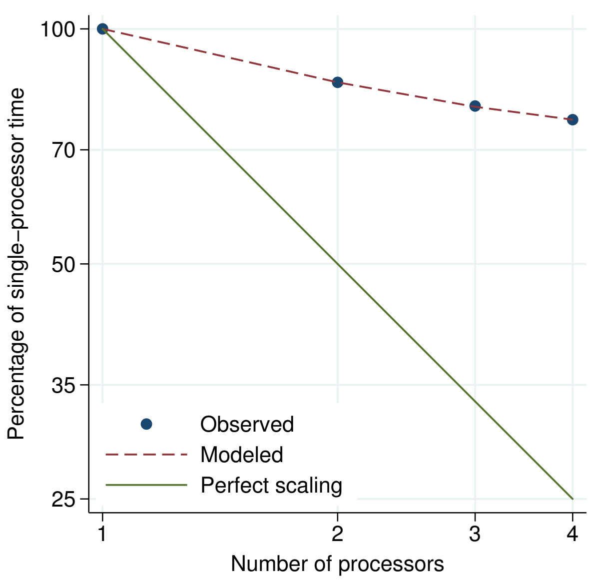

Figure A.77. `graph box` performance plot.

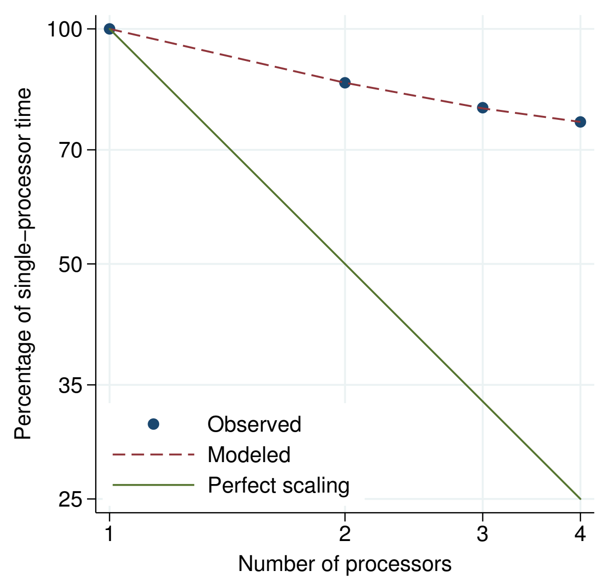

Figure A.78. `graph pie` performance plot.

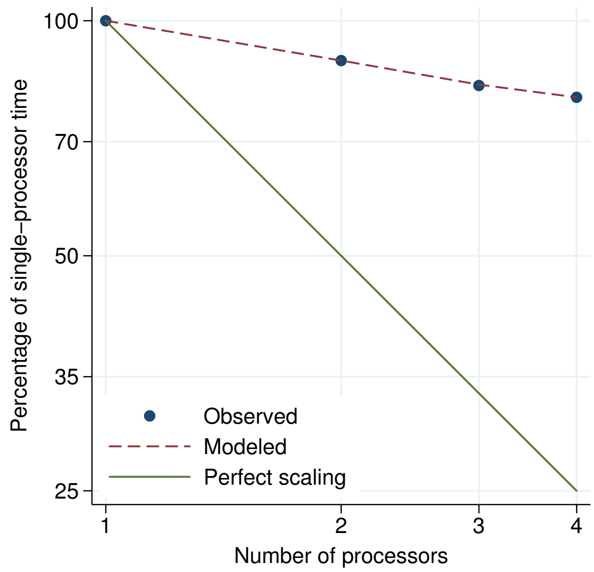

Figure A.79. `grmeanby` performance plot.

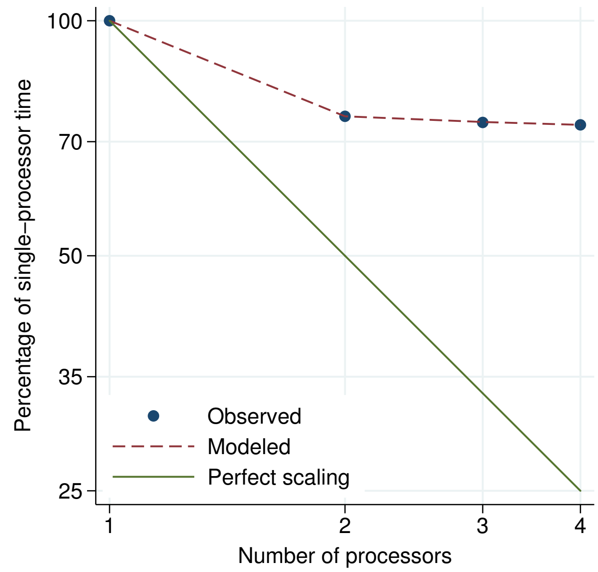

Figure A.80. `hausman` performance plot.

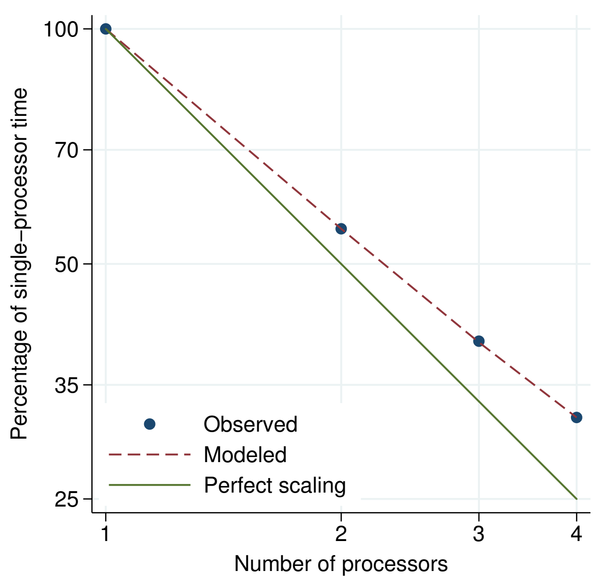

Figure A.81. `heckman` performance plot.

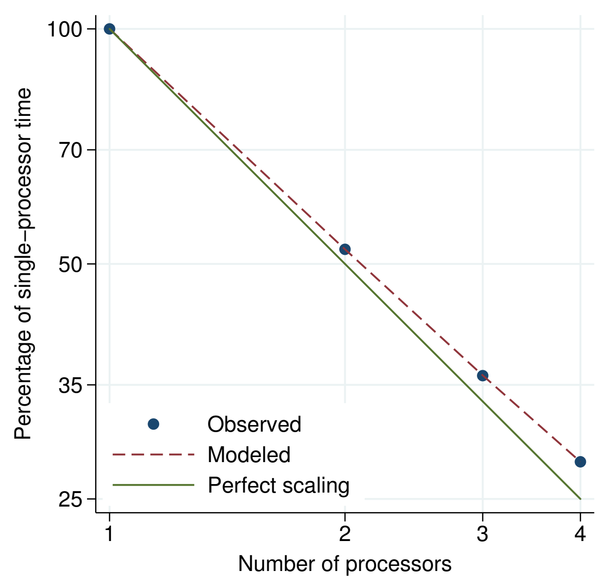

Figure A.82. `heckman, twostep` performance plot.

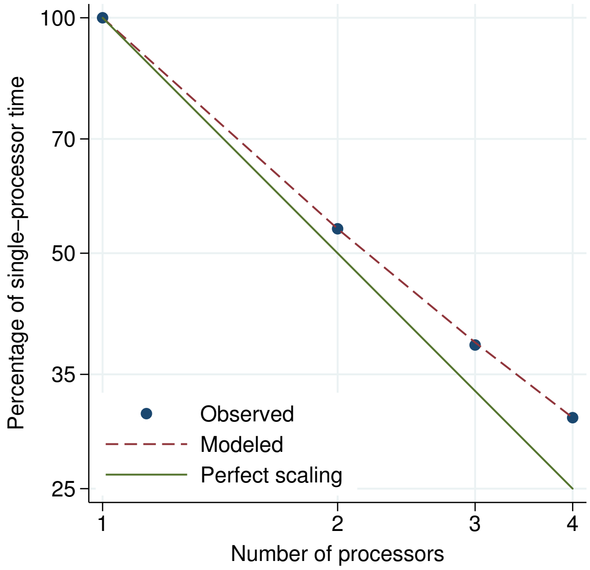

Figure A.83. `heckprob` performance plot.

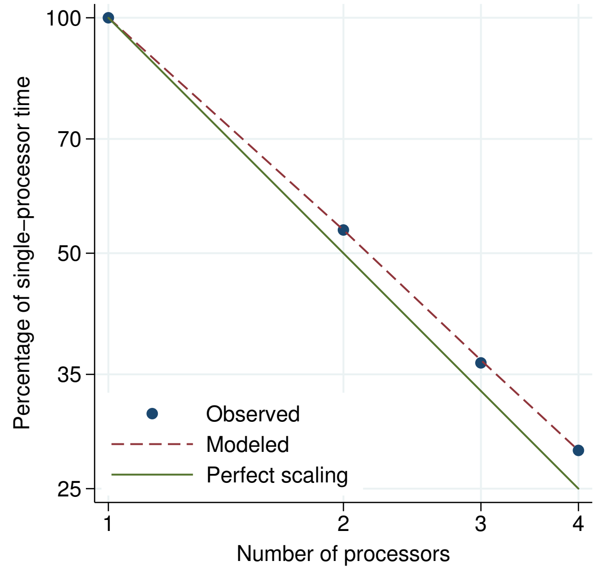

Figure A.84. `hetprob` performance plot.

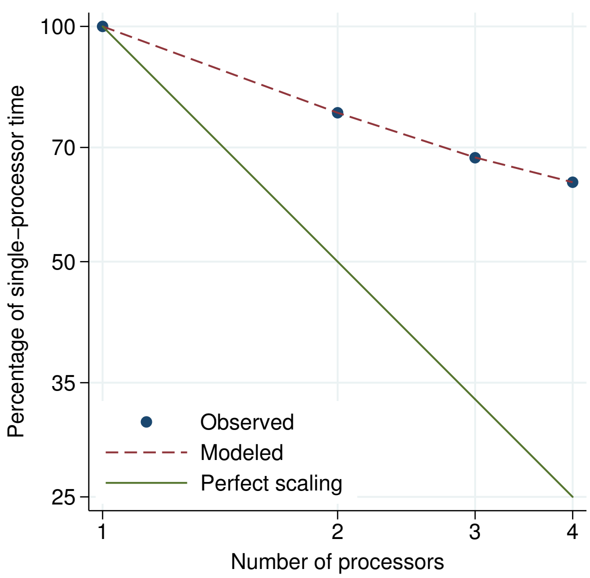

Figure A.85. `histogram` performance plot.

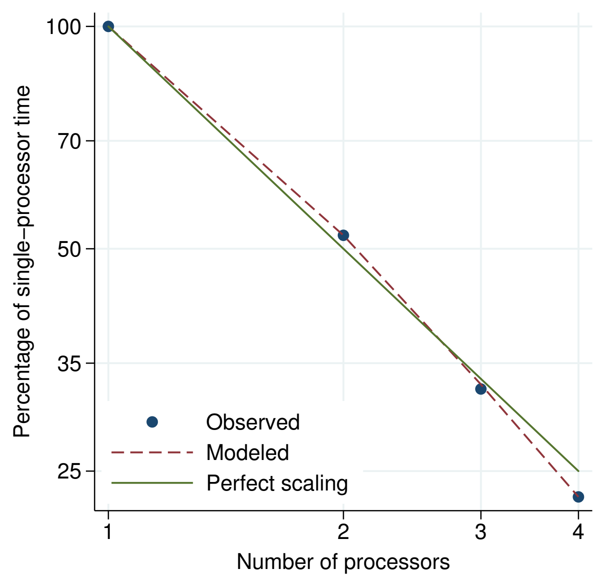

Figure A.86. `hotelling` performance plot.

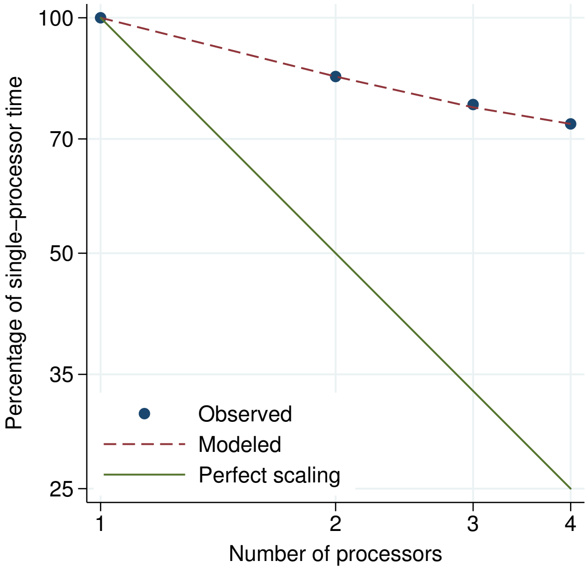

Figure A.87. `impute` performance plot.

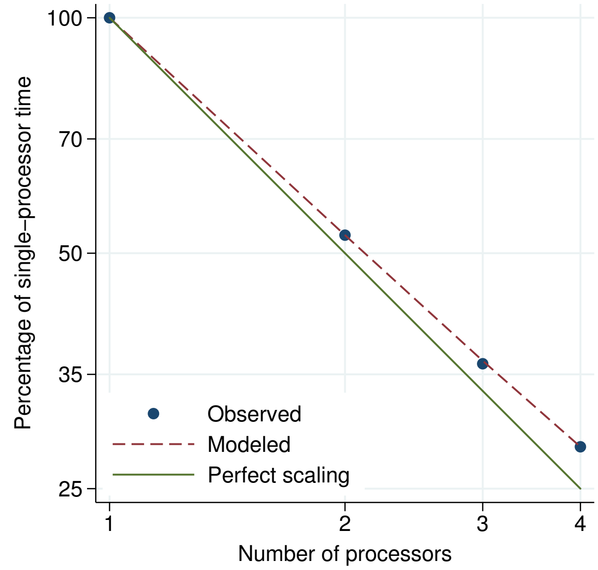

Figure A.88. `intreg` performance plot.

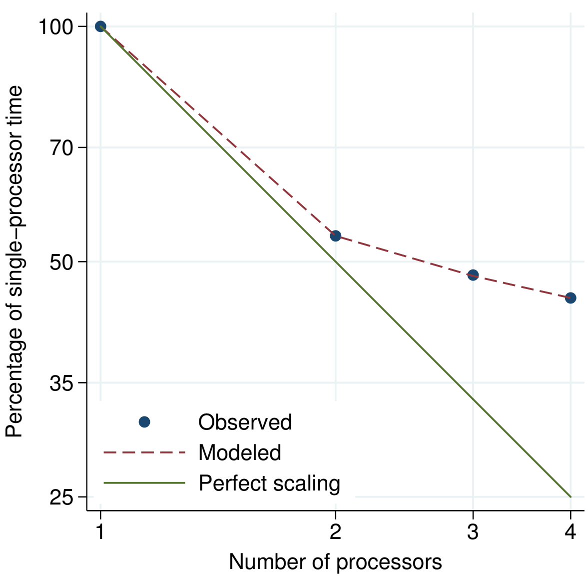

Figure A.89. `irf create` performance plot.

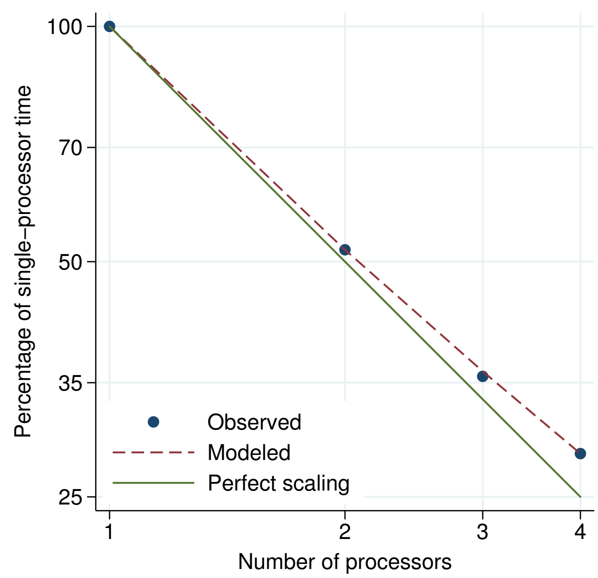

Figure A.90. `ivprobit` performance plot.

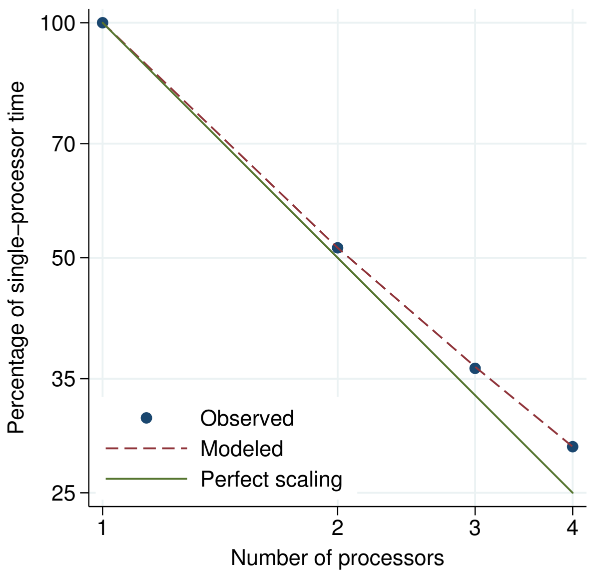

Figure A.91. `ivprobit, cluster()` performance plot.

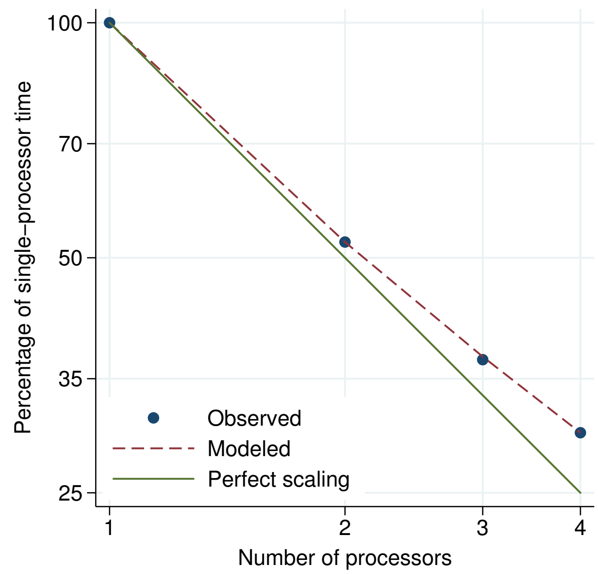

Figure A.92. `ivprobit, robust` performance plot.

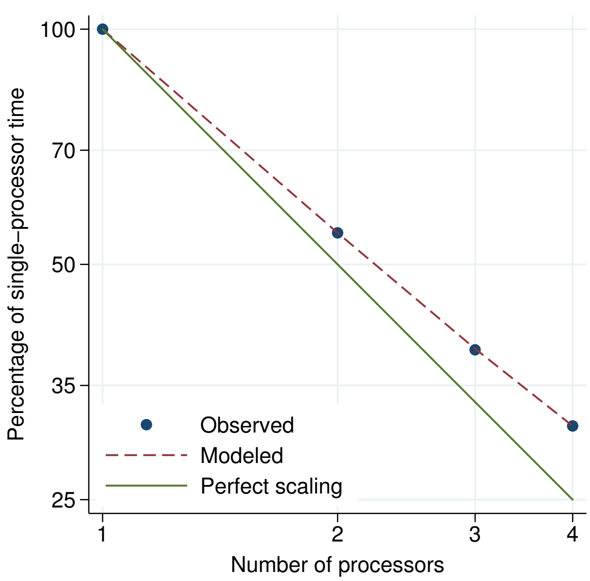

Figure A.93. `ivreg` performance plot.

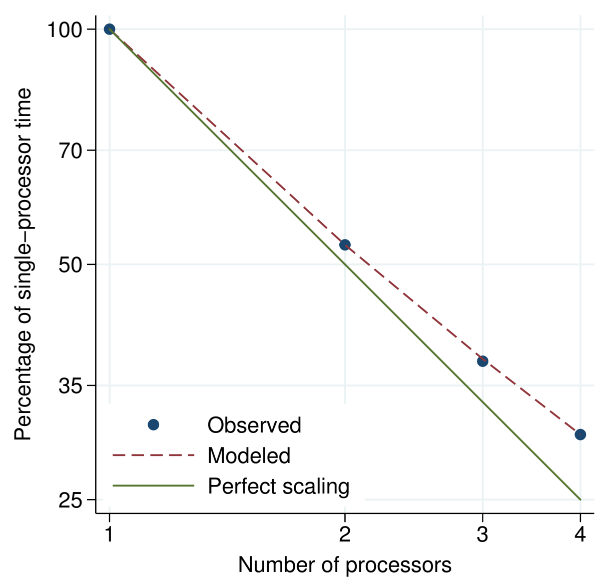

Figure A.94. `ivtobit` performance plot.

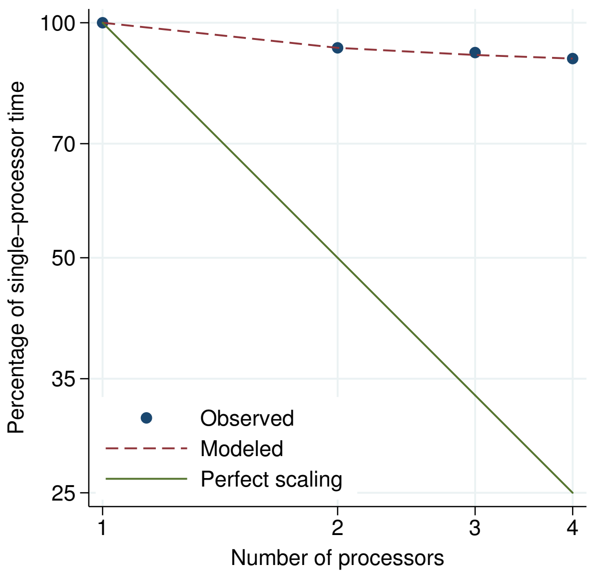

Figure A.95. `kap` performance plot.

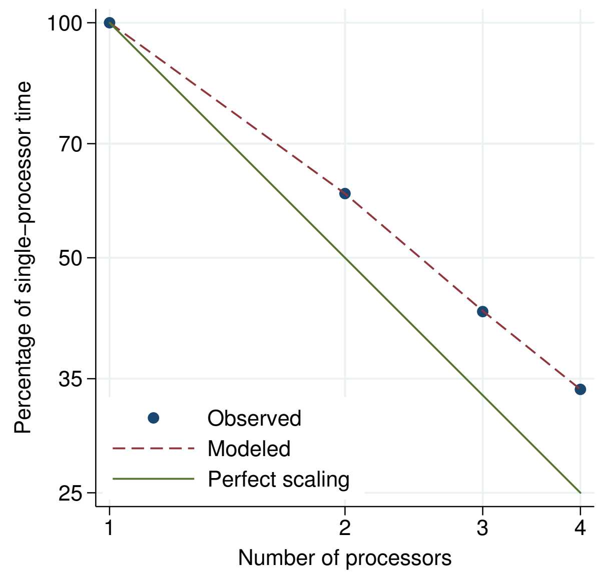

Figure A.96. `kappa` performance plot.

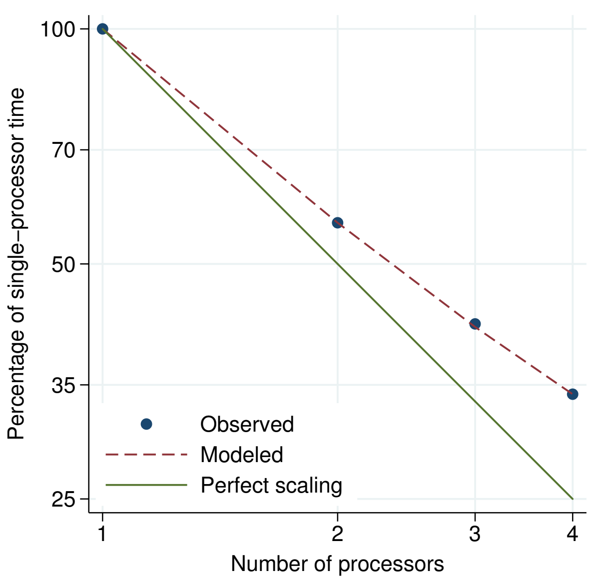

Figure A.97. `kdensity` performance plot.

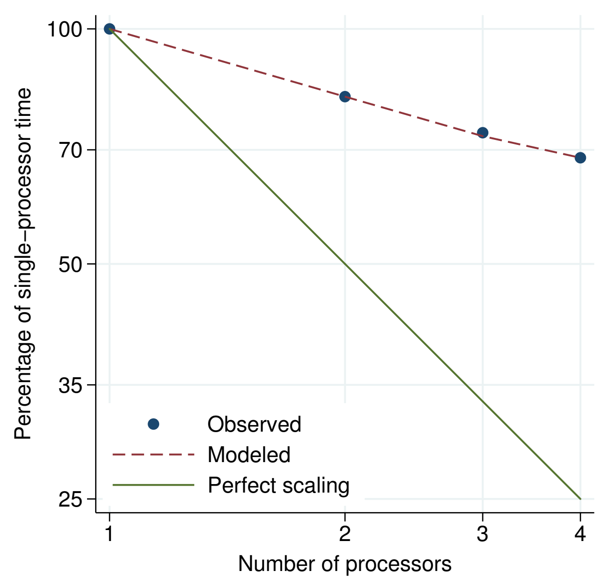

Figure A.98. `ksmirnov` performance plot.

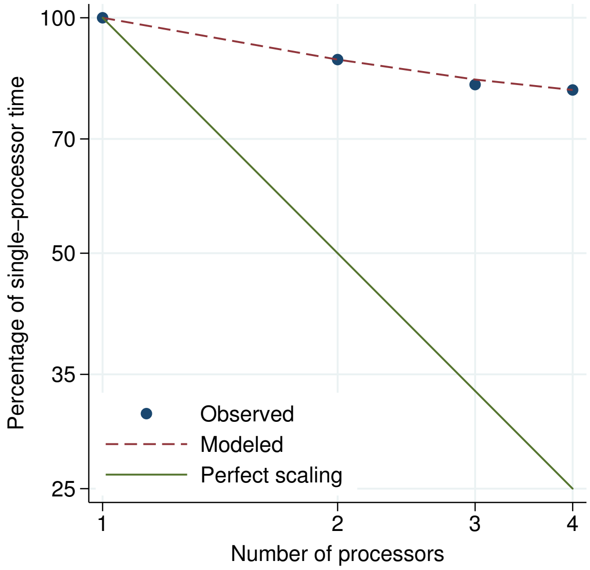

Figure A.99. `ksmirnov, by()` performance plot.

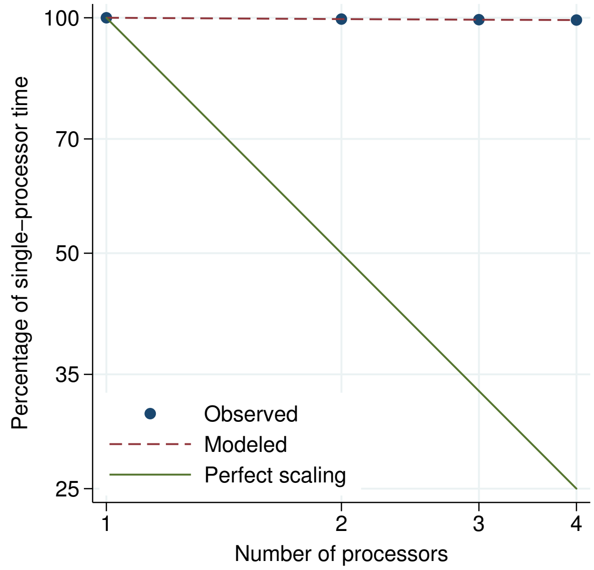

Figure A.100. `ktau` performance plot.

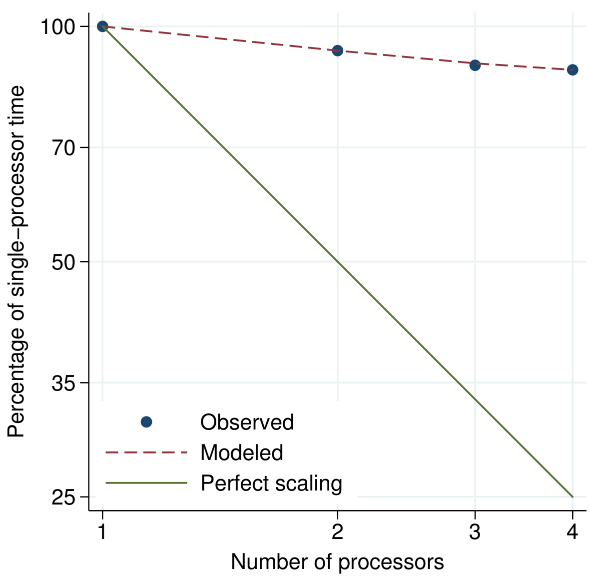

Figure A.101. `kwallis` performance plot.

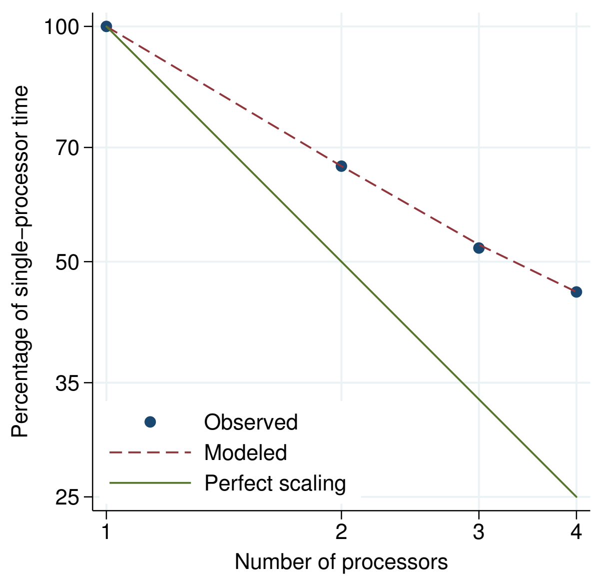

Figure A.102. `ladder` performance plot.

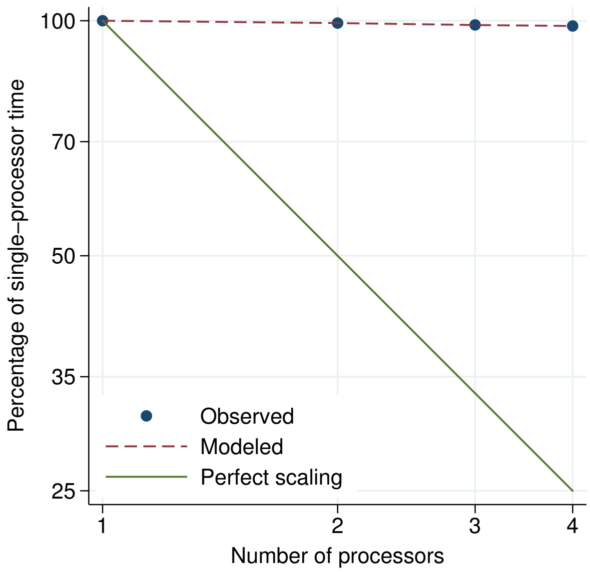

Figure A.103. `levelsof` performance plot.

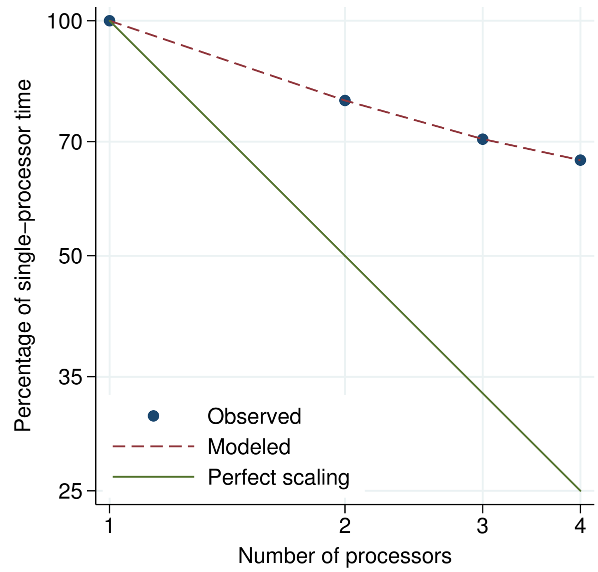

Figure A.104. `loadingplot` performance plot.

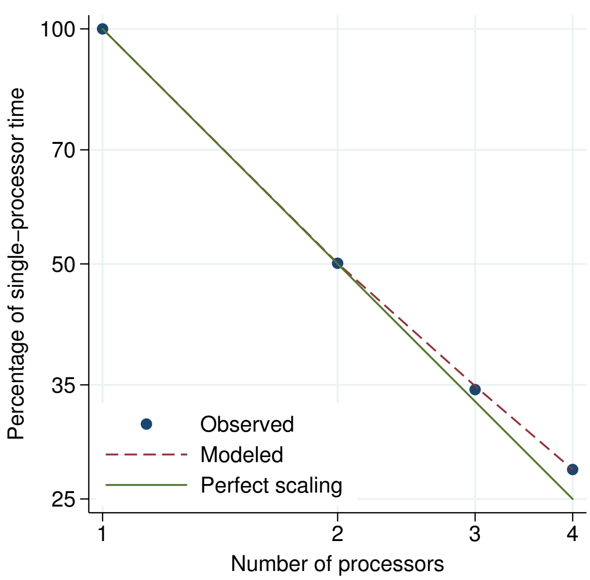

Figure A.105. `logistic` performance plot.

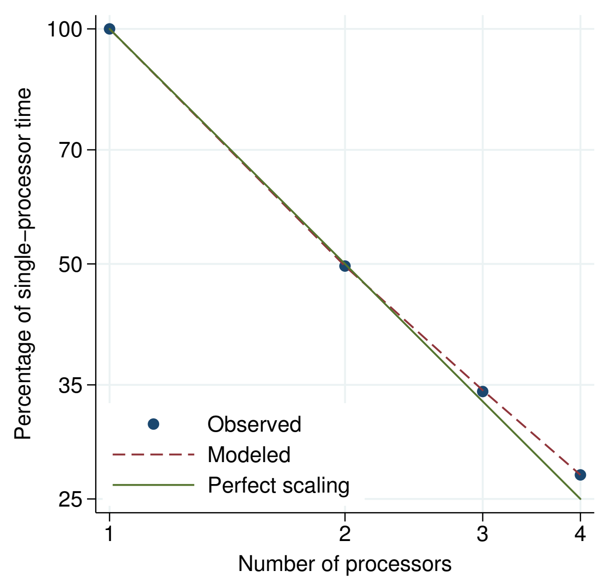

Figure A.106. `logit` performance plot.

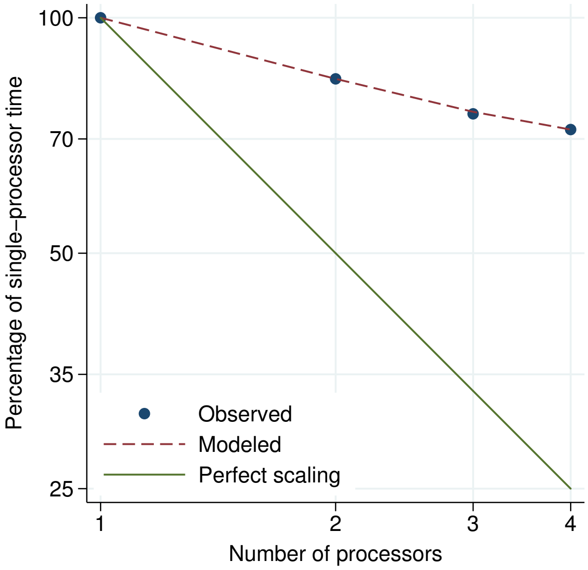

Figure A.107. `loneway` performance plot.

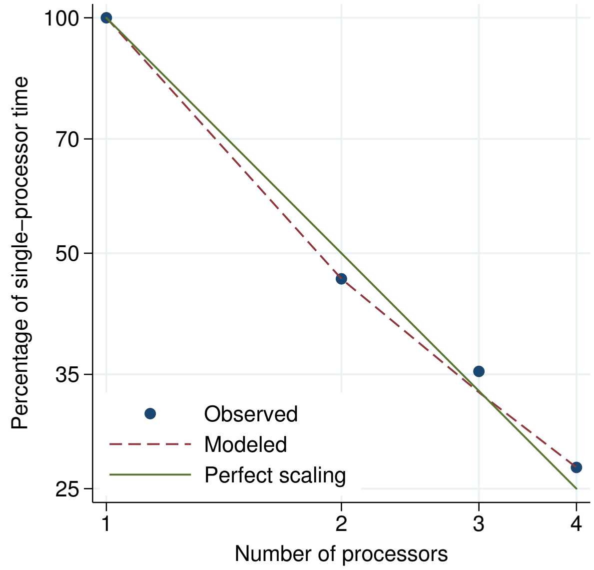

Figure A.108. `lowess` performance plot.

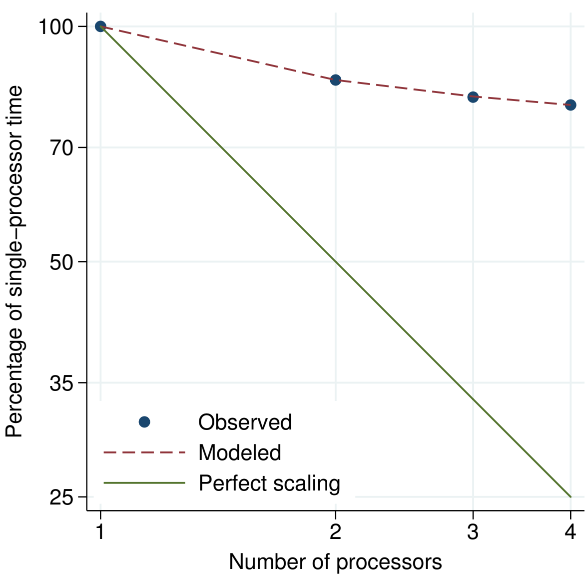

Figure A.109. `ltable` performance plot.

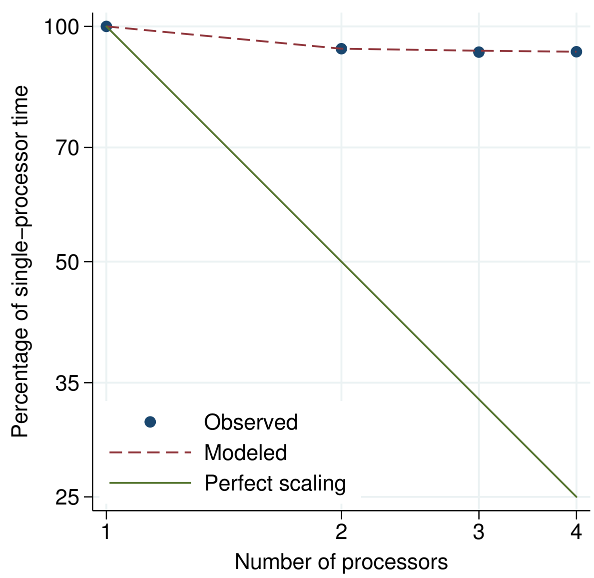

Figure A.110. `manova (oneway)` performance plot.

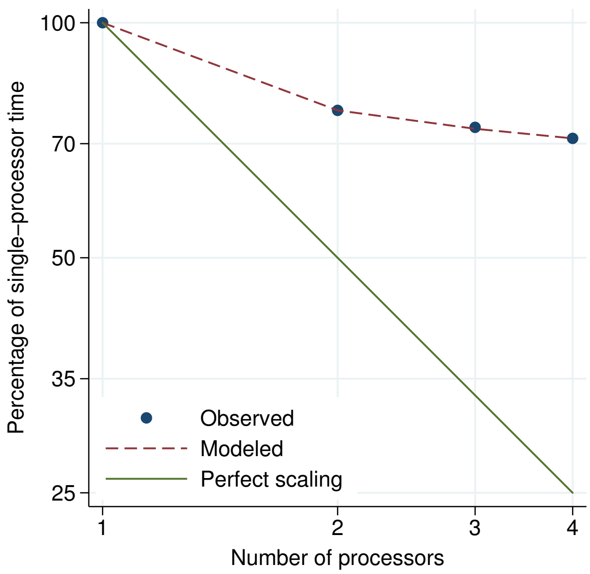

Figure A.111. `manova (twoway)` performance plot.

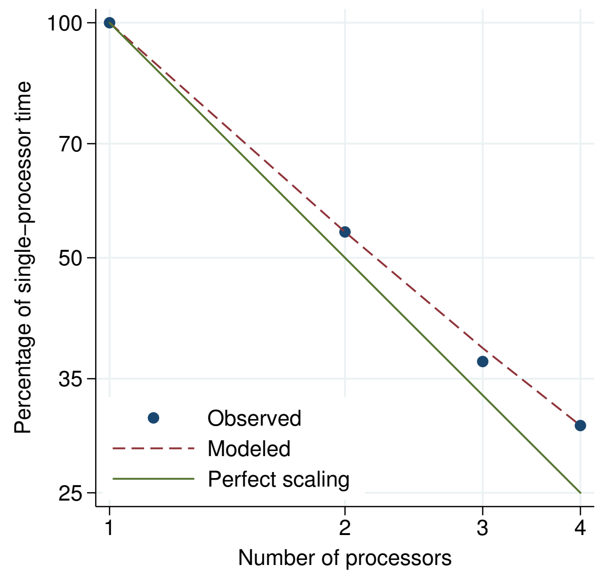

Figure A.112. `markout` performance plot.

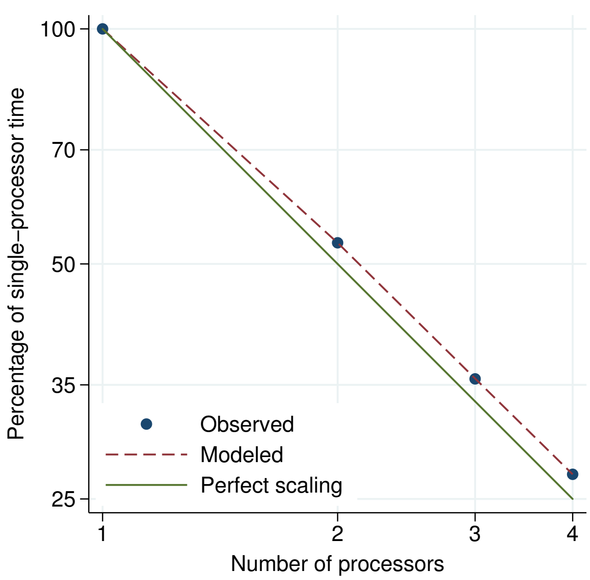

Figure A.113. `marksample` performance plot.

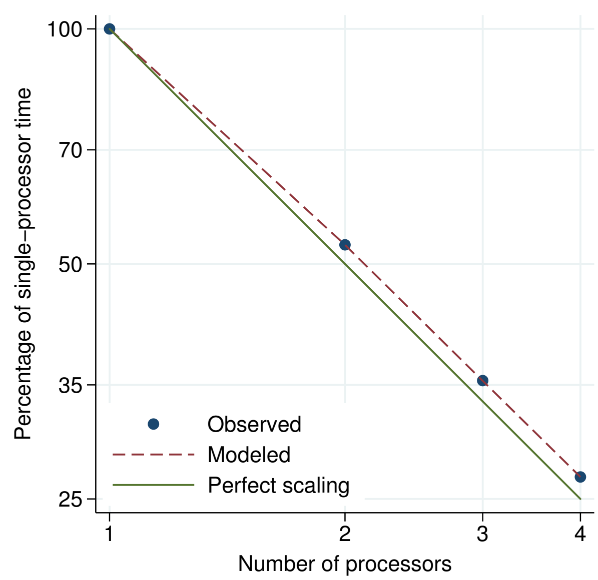

Figure A.114. `marksample if exp` performance plot.

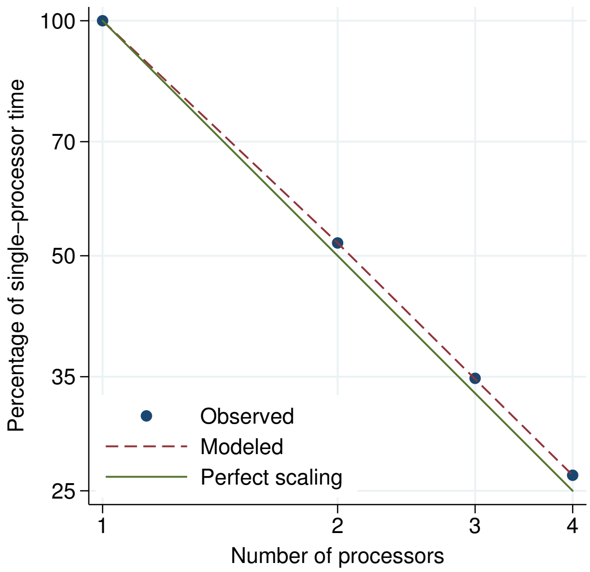

Figure A.115. `matrix accum` performance plot.

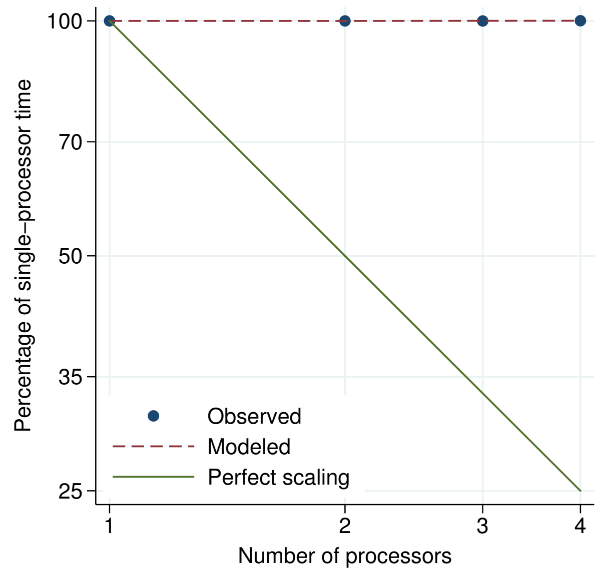

Figure A.116. `matrix eigenvalues` performance plot.

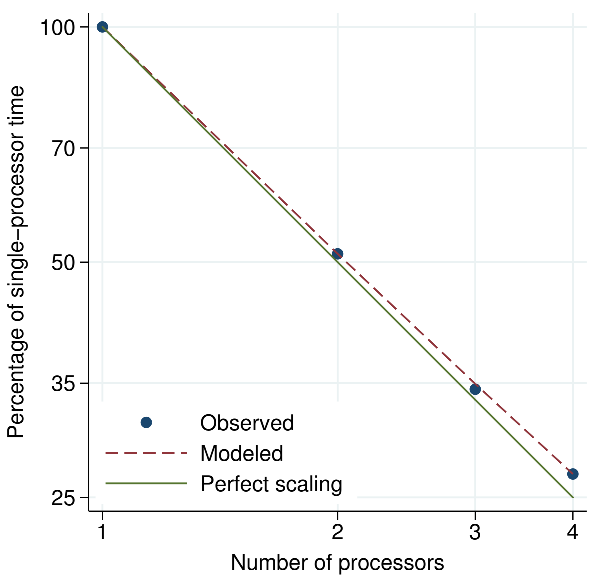

Figure A.117. `matrix score` performance plot.

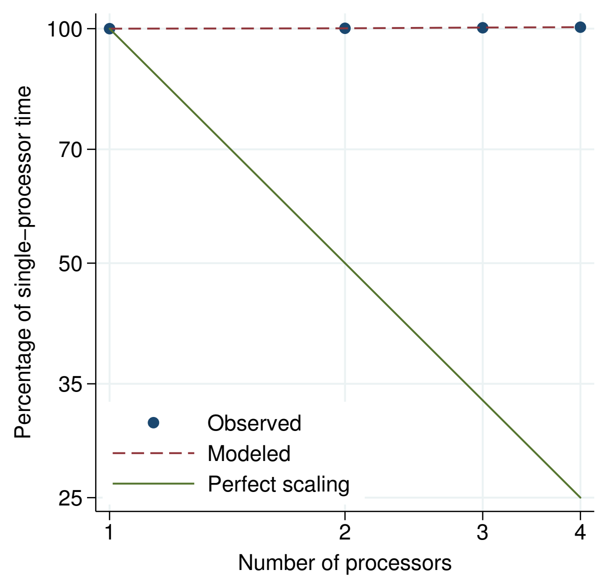

Figure A.118. `matrix svd` performance plot.

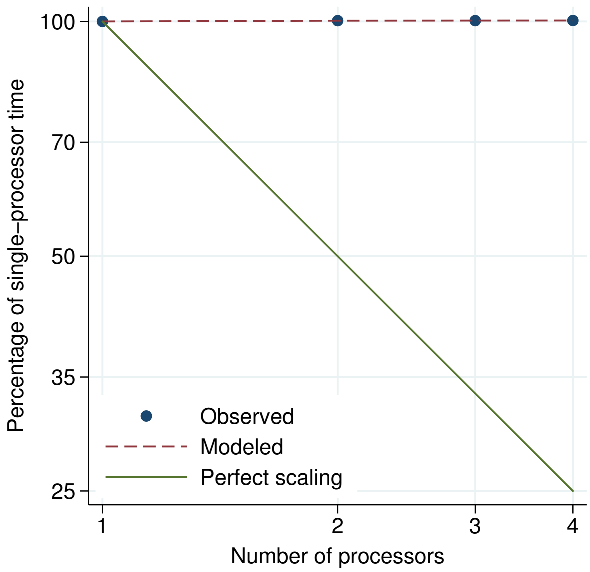

Figure A.119. `matrix symeigen` performance plot.

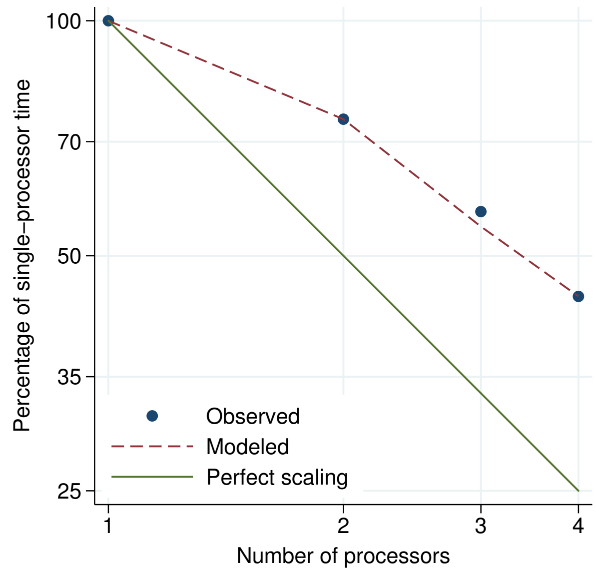

Figure A.120. `matrix syminv` performance plot.

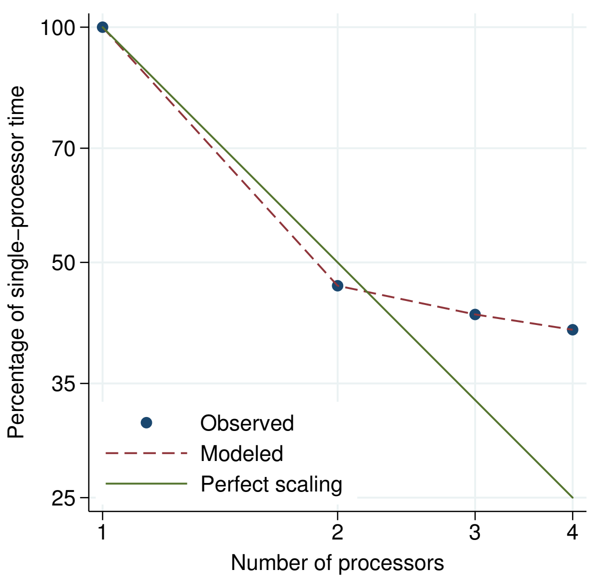

Figure A.121. `mds` performance plot.

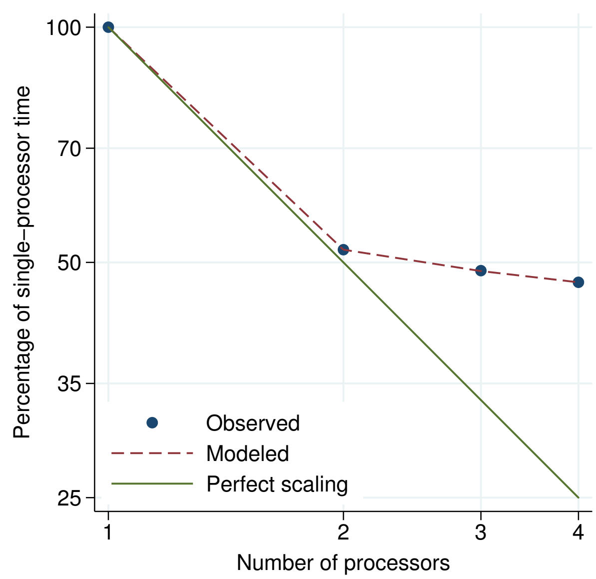

Figure A.122. `mdslong` performance plot.

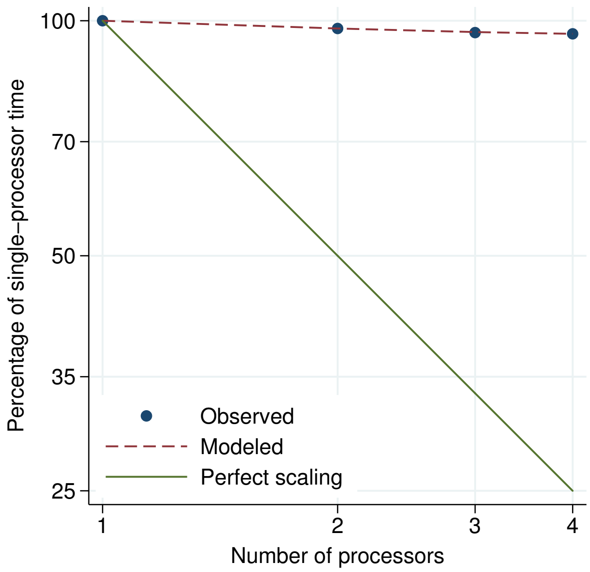

Figure A.123. `mean` performance plot.

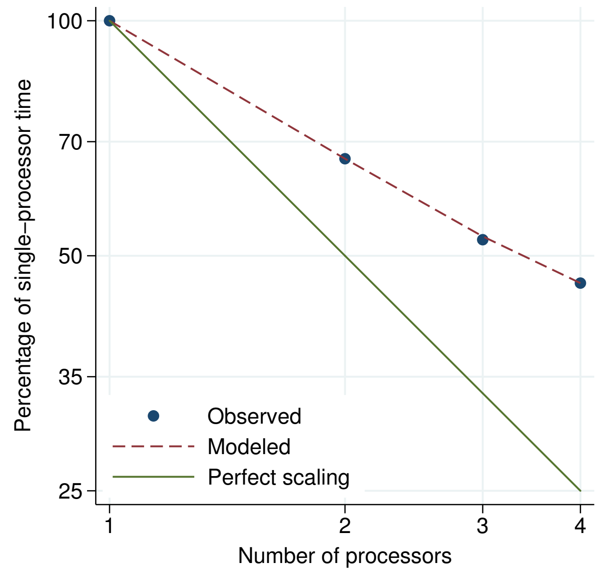

Figure A.124. `median` performance plot.

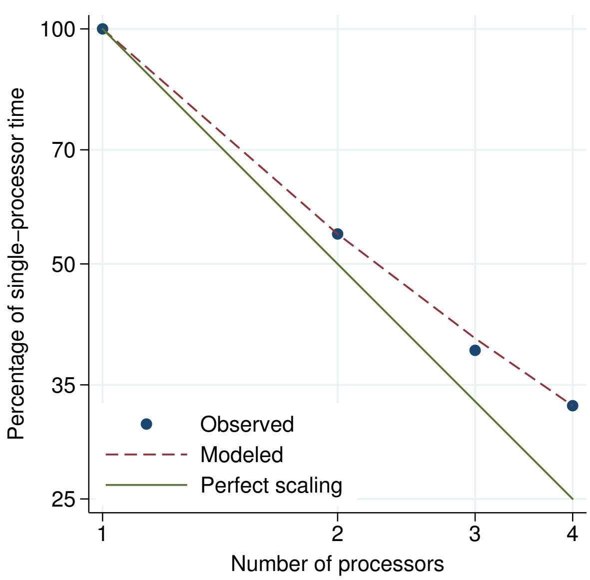

Figure A.125. `mfp` performance plot.

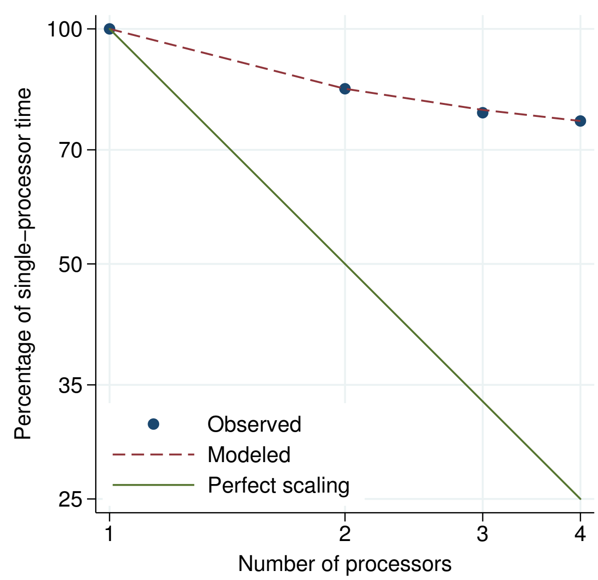

Figure A.126. `mfx` performance plot.

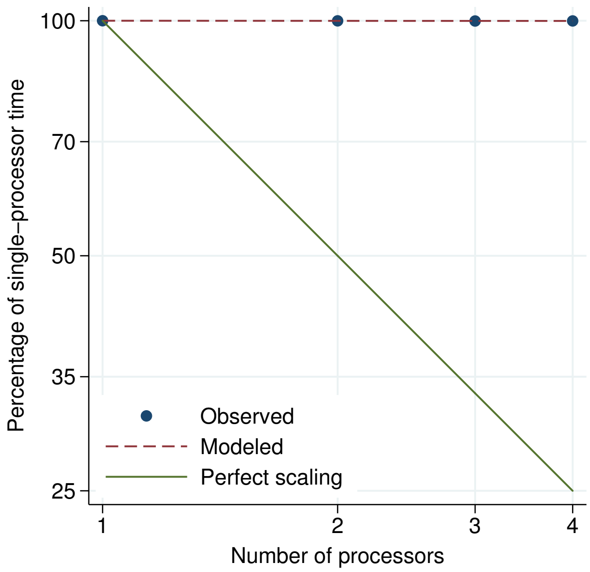

Figure A.127. `mkmat` performance plot.

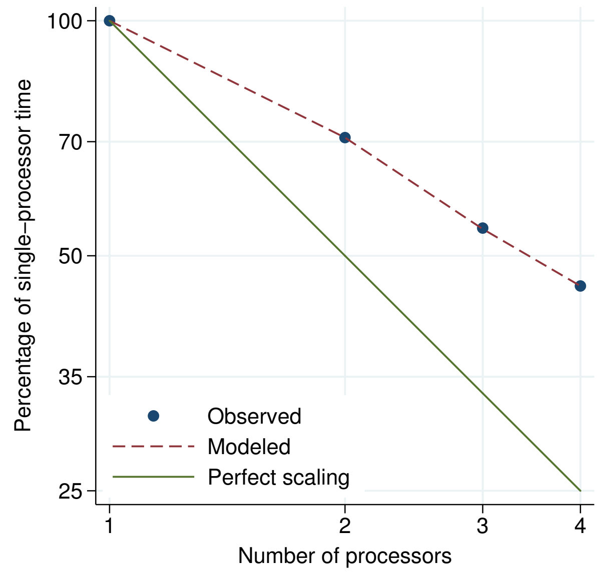

Figure A.128. `mkspline` performance plot.

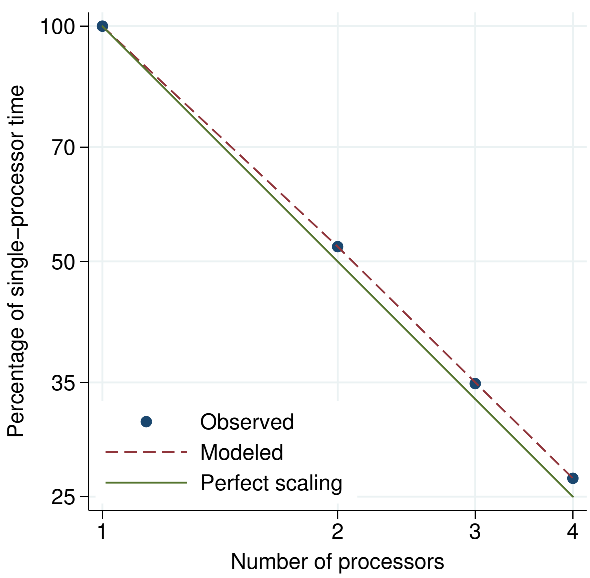

Figure A.129. `mleval` performance plot.

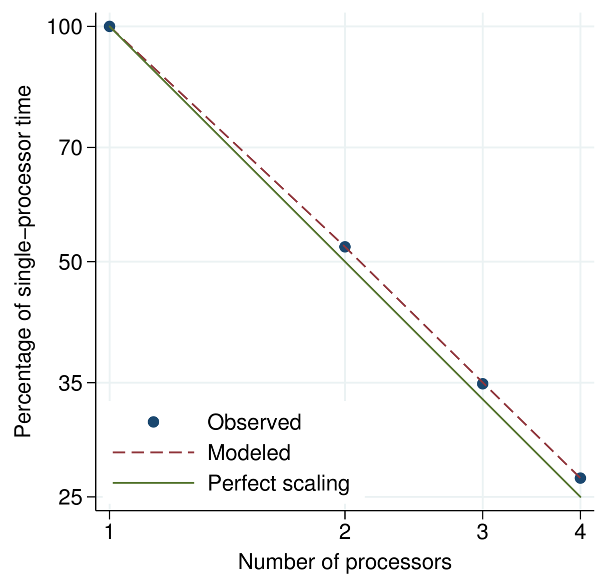

Figure A.130. `mleval, nocons` performance plot.

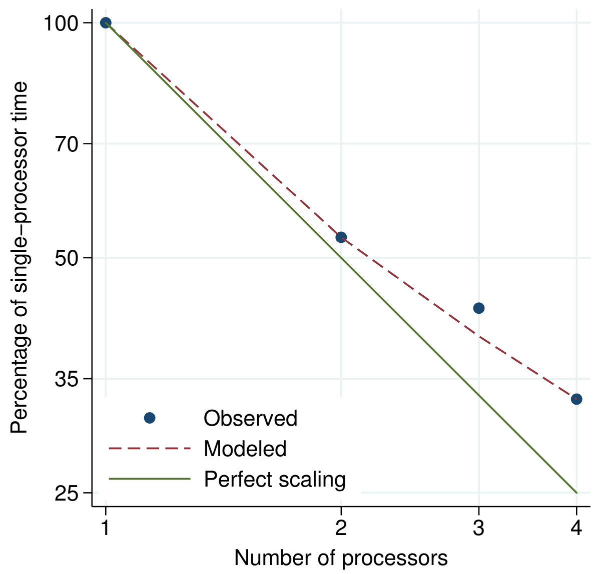

Figure A.131. `mlmatbysum` performance plot.

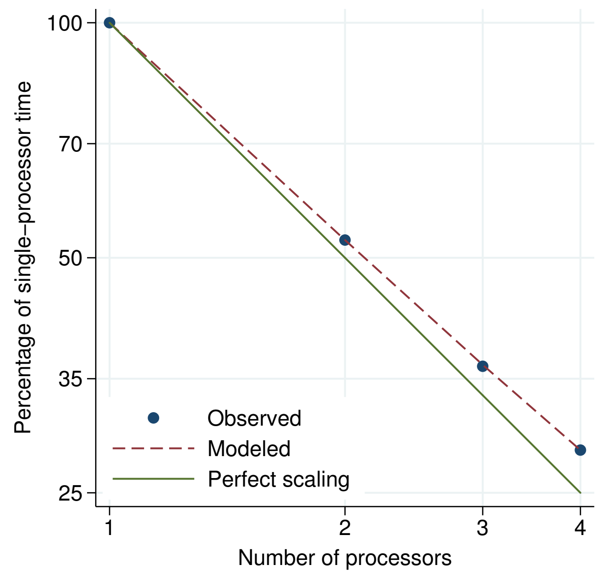

Figure A.132. `mlmatsum` performance plot.

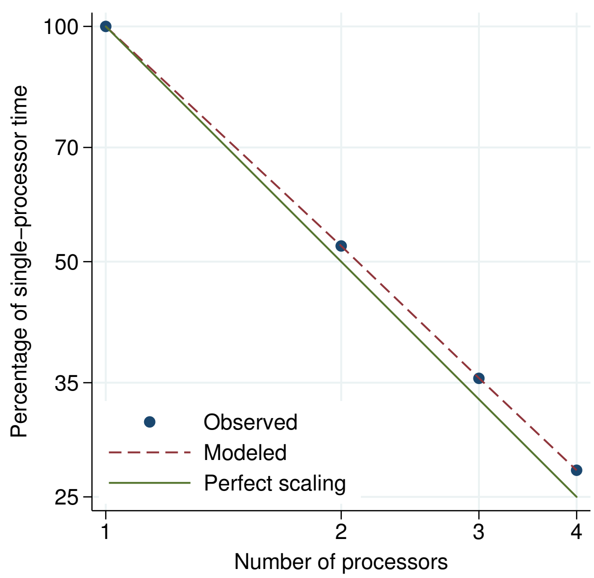

Figure A.133. `mlogit` performance plot.

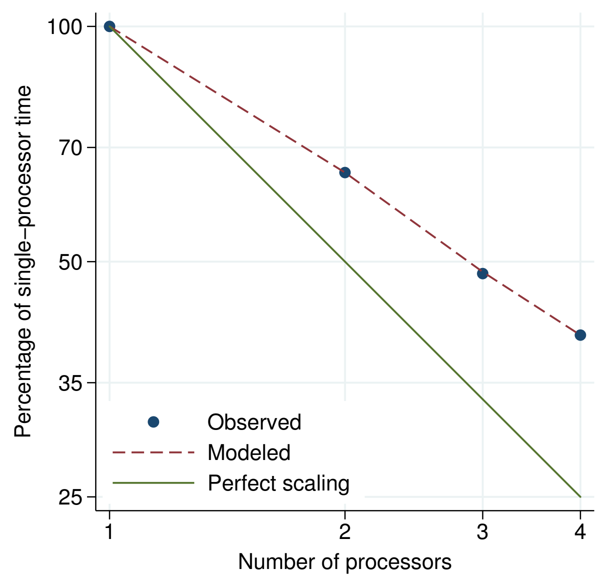

Figure A.134. `mlsum` performance plot.

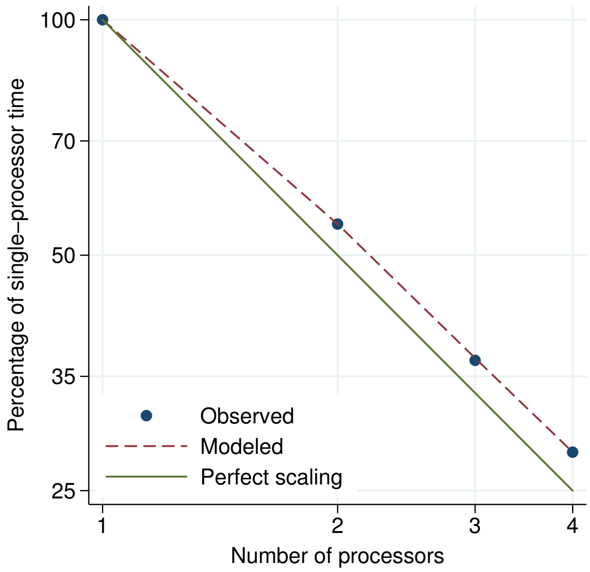

Figure A.135. `mlvecsum` performance plot.

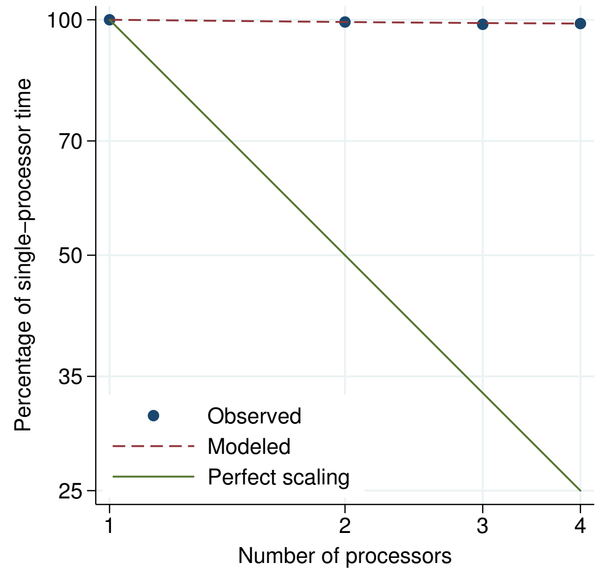

Figure A.136. `mprobit` performance plot.

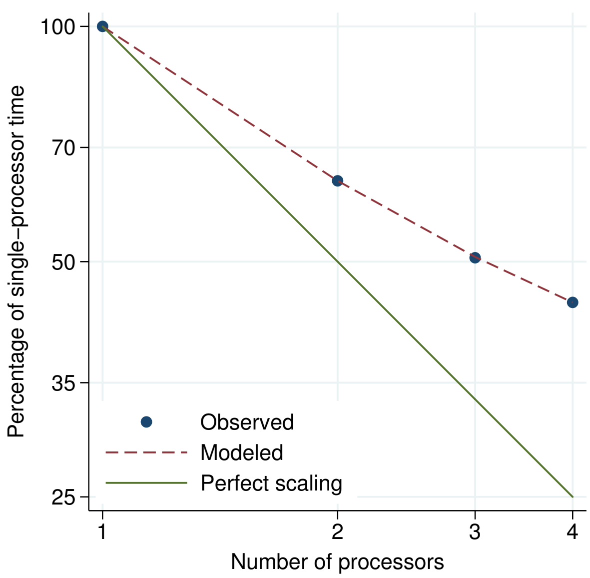

Figure A.137. `mvreg` performance plot.

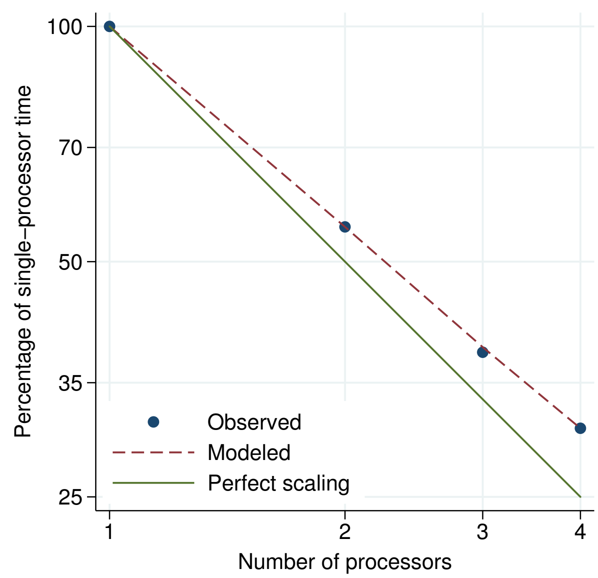

Figure A.138. `nbreg` performance plot.

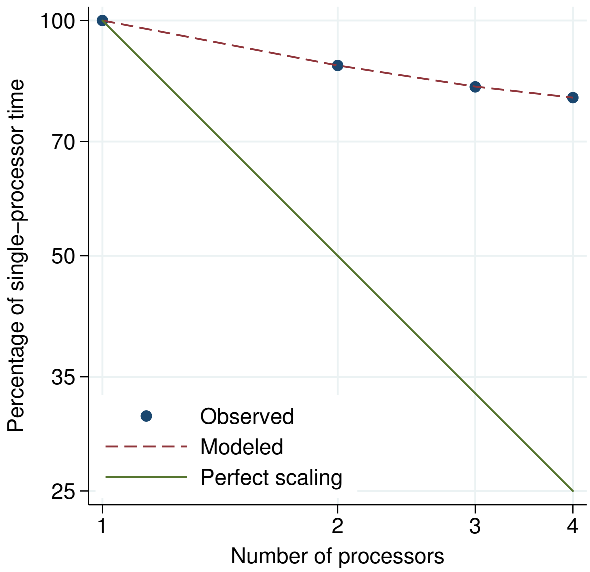

Figure A.139. `newey` performance plot.

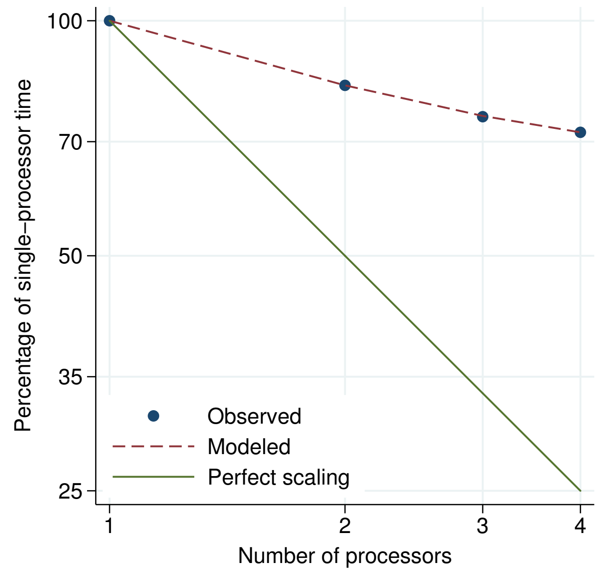

Figure A.140. `nl` performance plot.

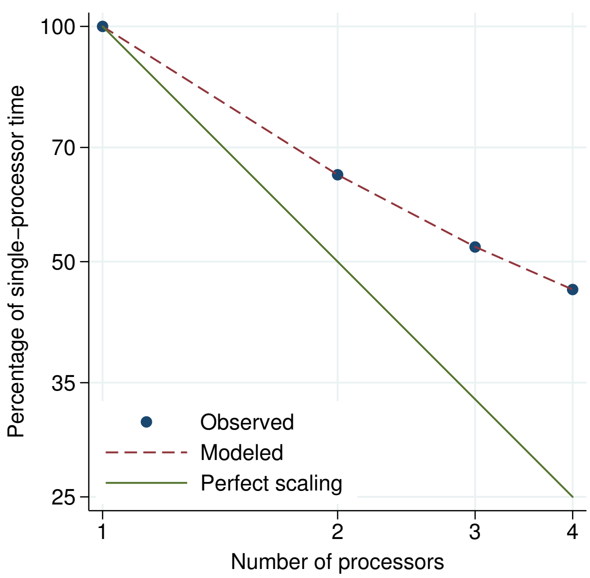

Figure A.141. `nlogit` performance plot.

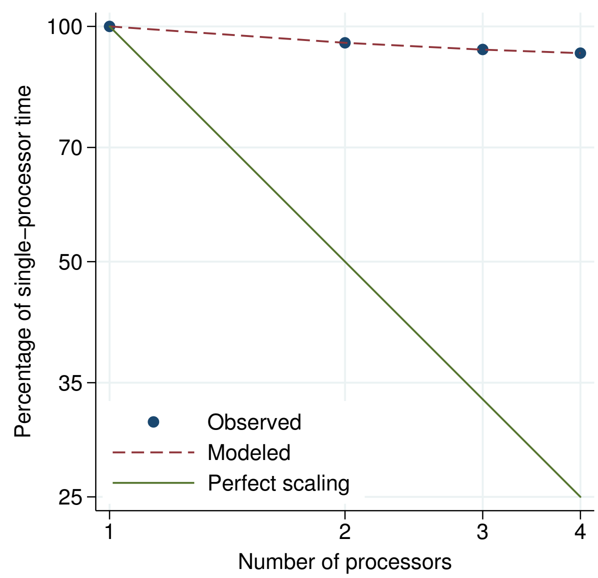

Figure A.142. `nptrend` performance plot.

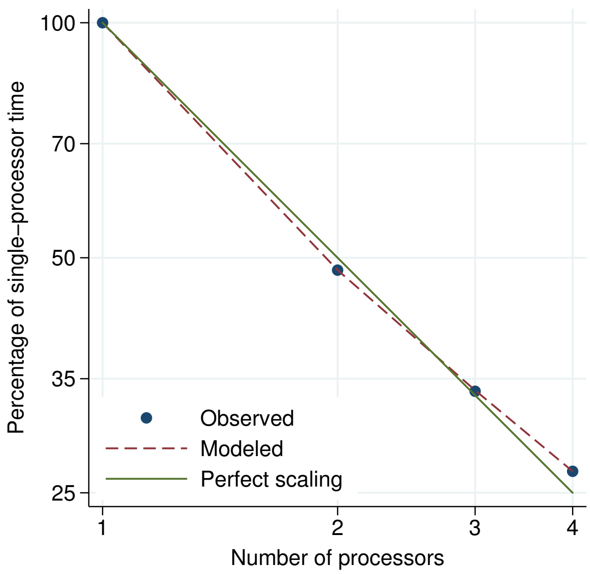

Figure A.143. `ologit` performance plot.

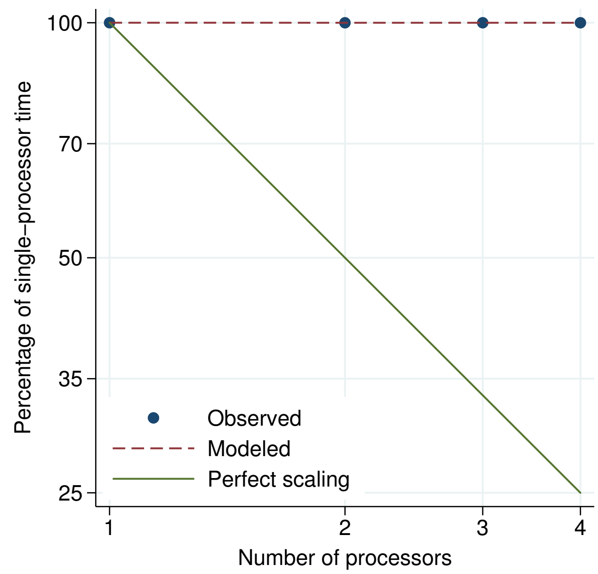

Figure A.144. `oneway` performance plot.

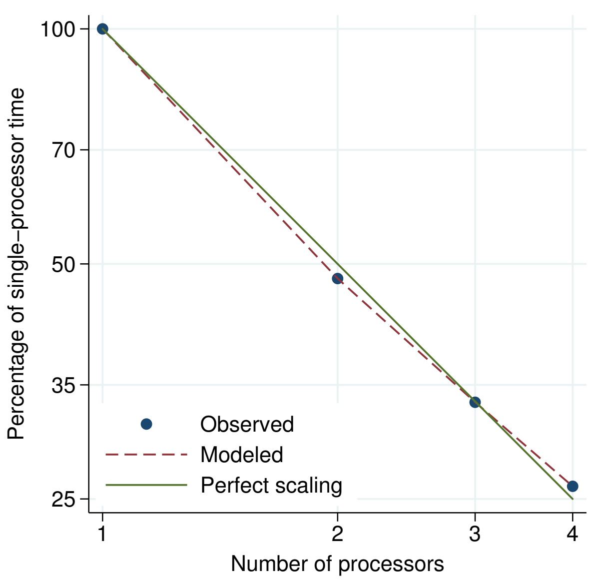

Figure A.145. `oprobit` performance plot.

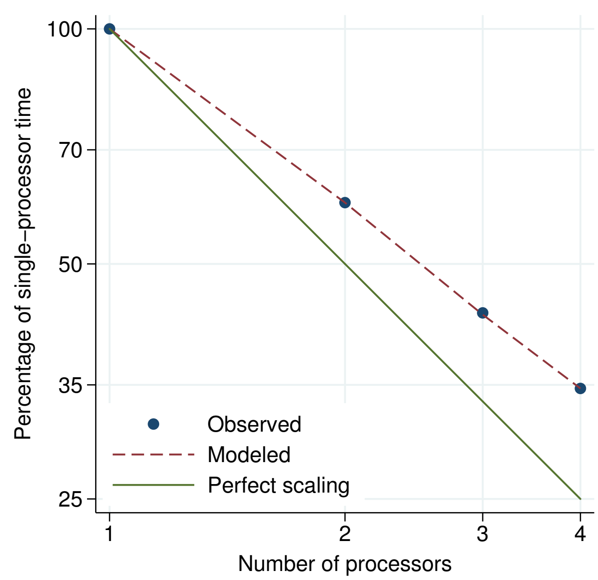

Figure A.146. `orthog` performance plot.

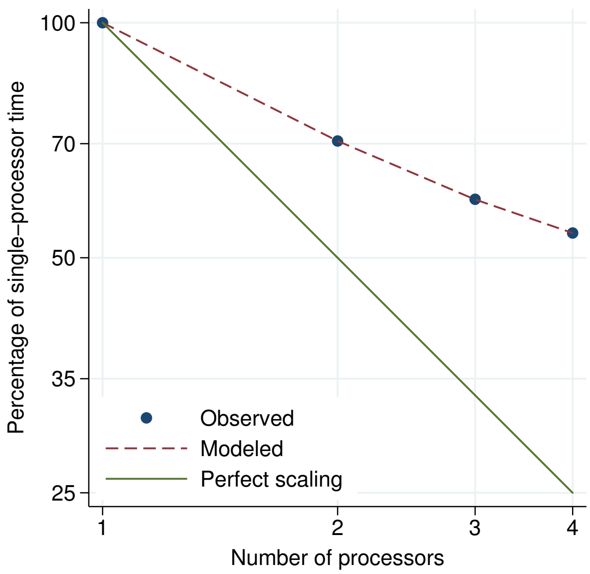

Figure A.147. `pca` performance plot.

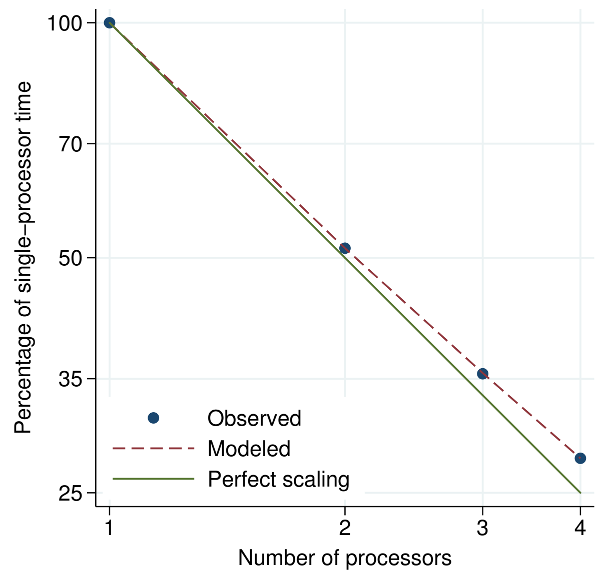

Figure A.148. `pcorr` performance plot.

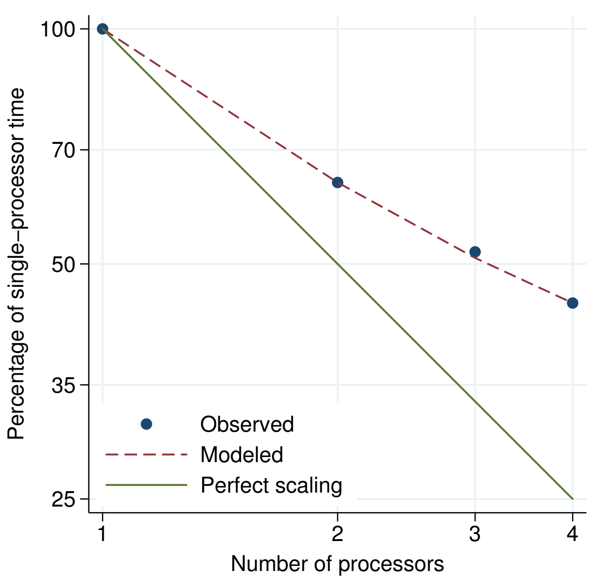

Figure A.149. `pctile` performance plot.

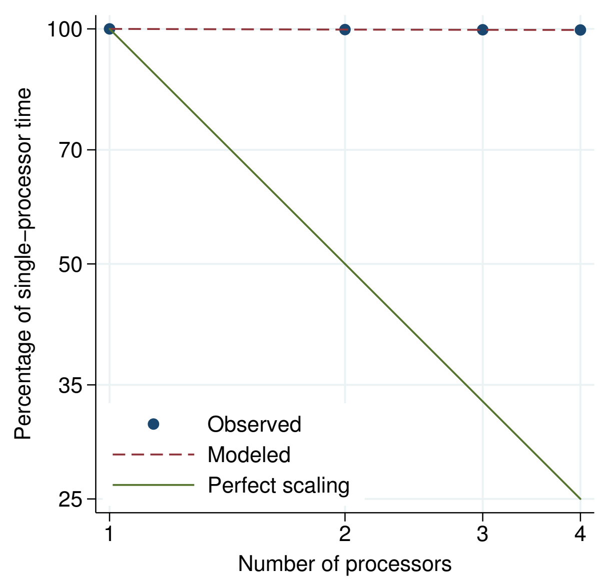

Figure A.150. `pergram` performance plot.

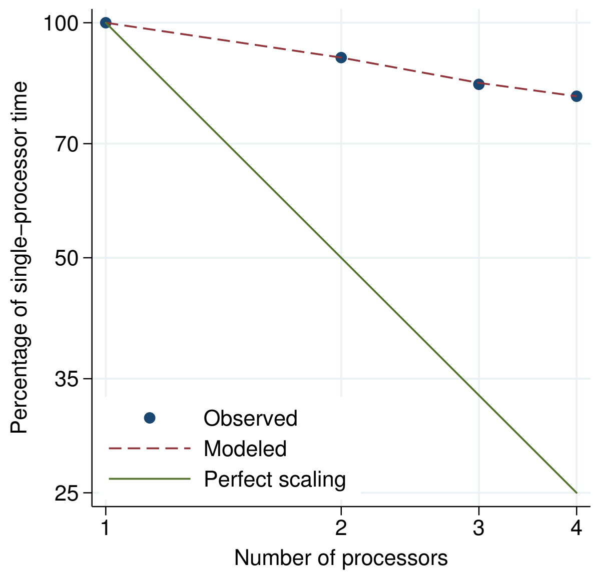

Figure A.151. `pkcollapse` performance plot.

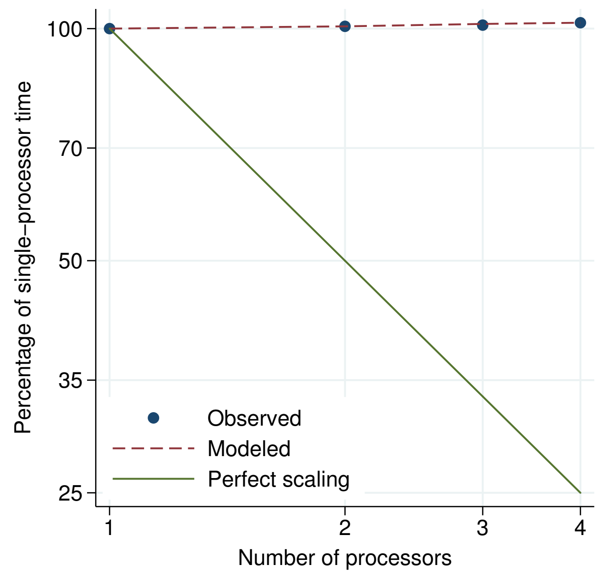

Figure A.152. `pkexamine` performance plot.

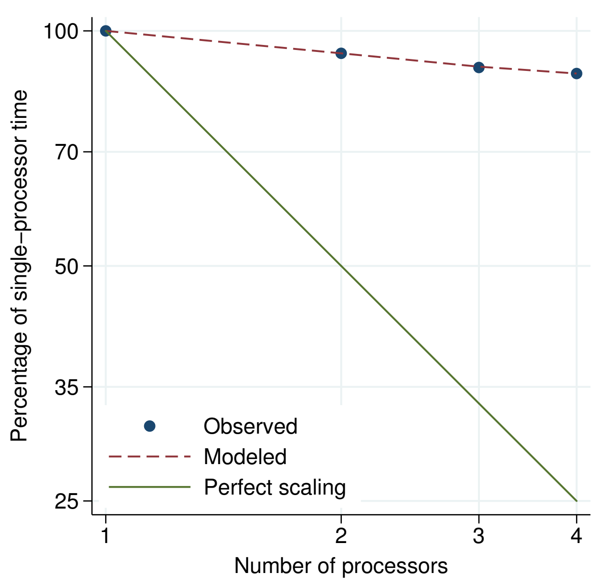

Figure A.153. `pksumm` performance plot.

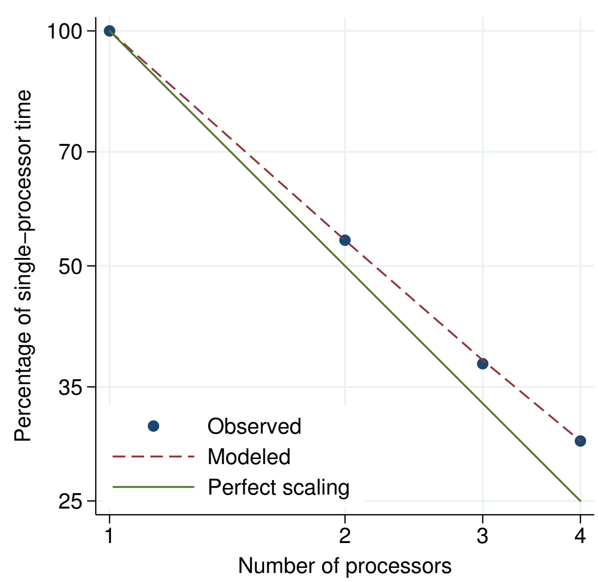

Figure A.154. `poisson` performance plot.

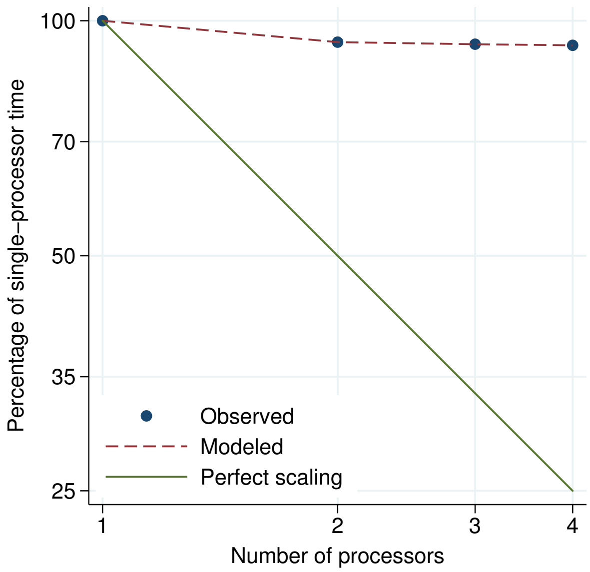

Figure A.155. `pperron` performance plot.

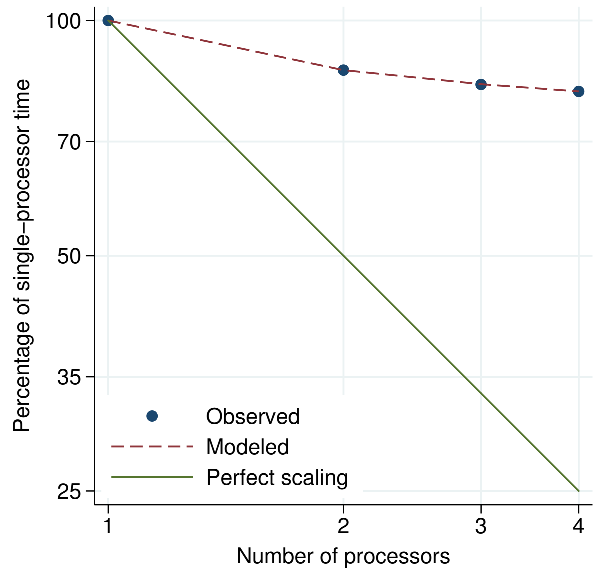

Figure A.156. `prais` performance plot.

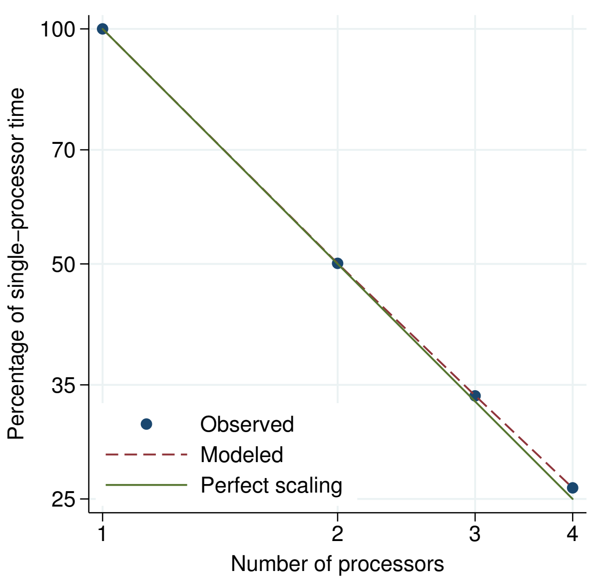

Figure A.157. `predict, cooksd` performance plot.

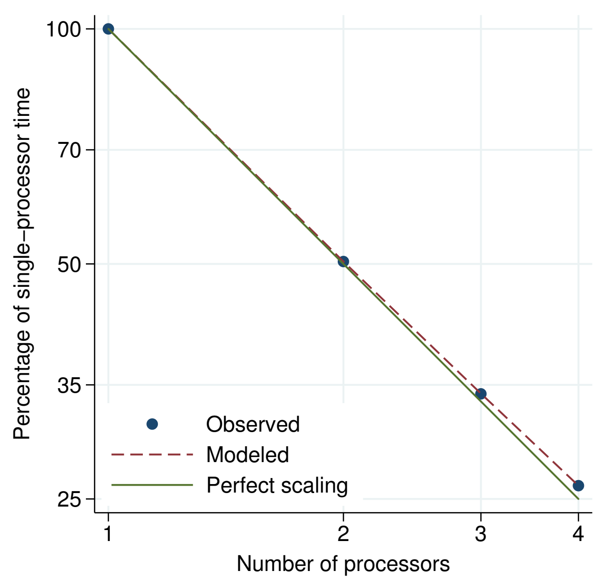

Figure A.158. `predict, covratio` performance plot.

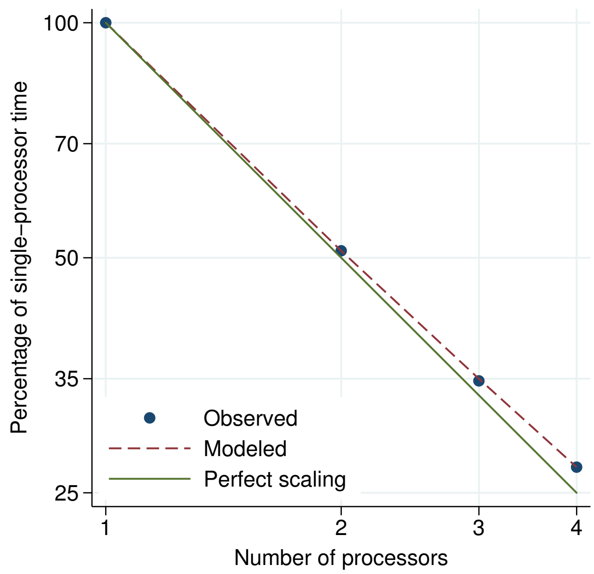

Figure A.159. `predict, dfbeta` performance plot.

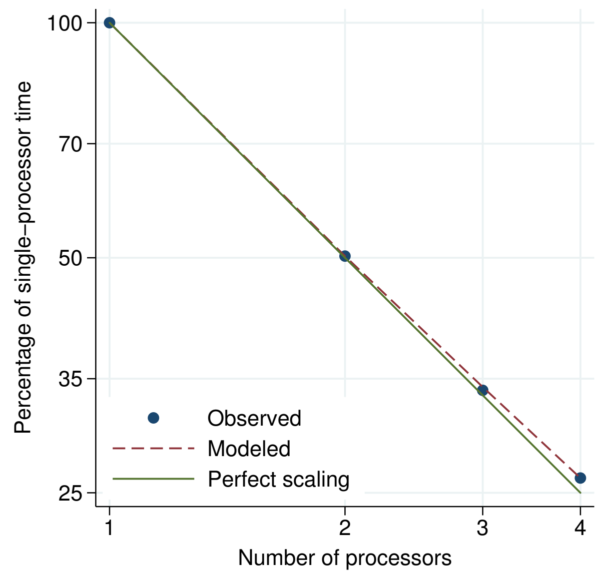

Figure A.160. `predict, dfits` performance plot.

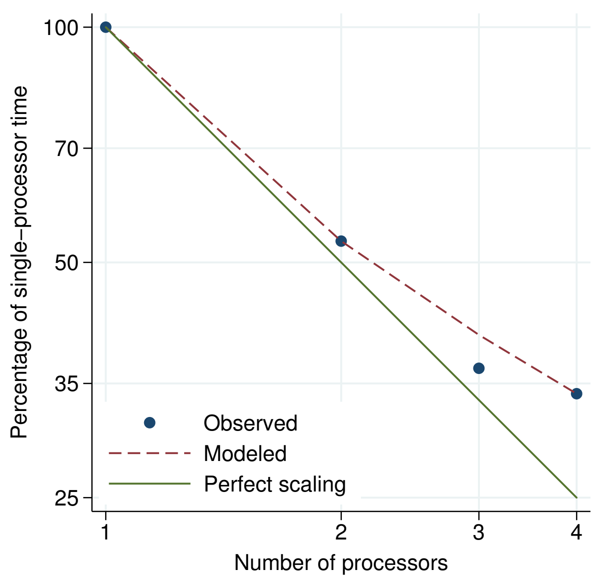

Figure A.161. `predict, e` performance plot.

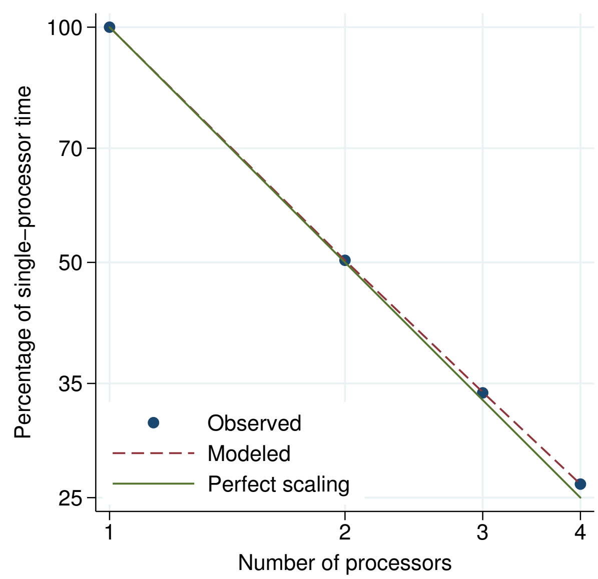

Figure A.162. `predict, leverage` performance plot.

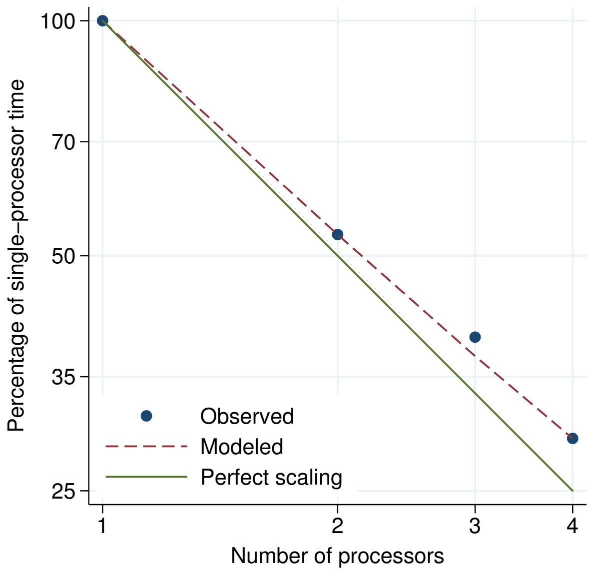

Figure A.163. `predict, pr` performance plot.

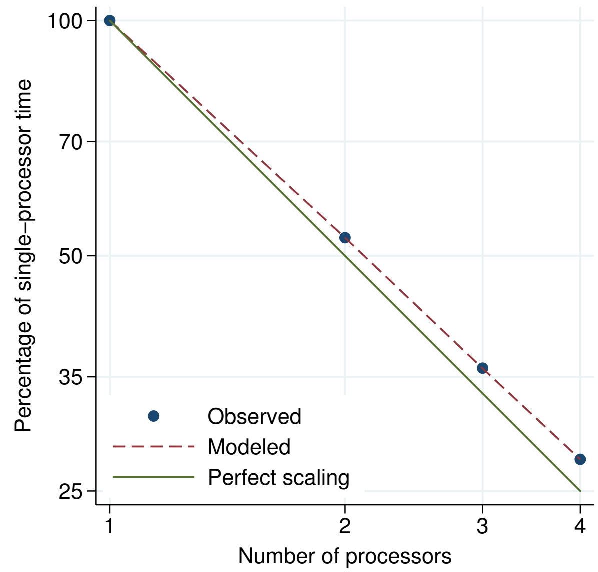

Figure A.164. `predict, residuals` performance plot.

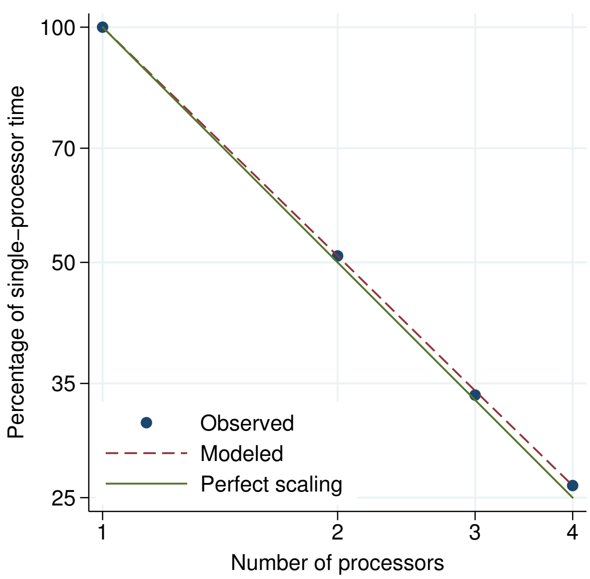

Figure A.165. `predict, rstandard` performance plot.

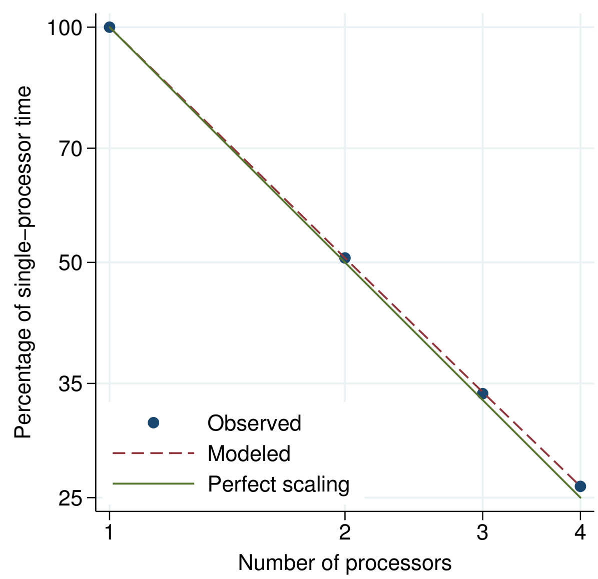

Figure A.166. `predict, rstudent` performance plot.

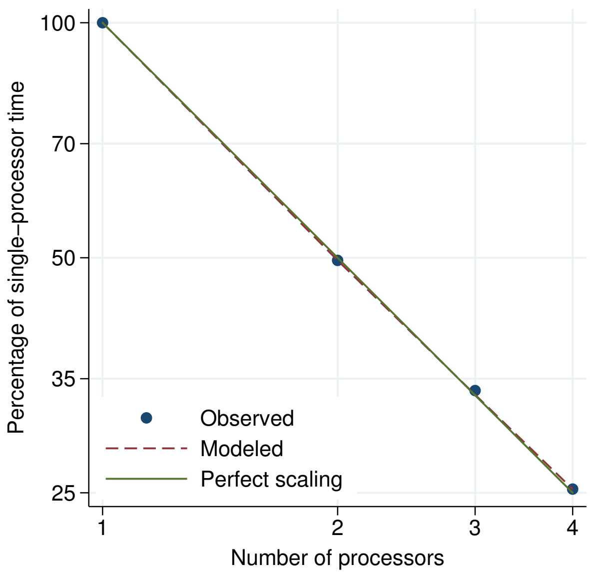

Figure A.167. `predict, stdf` performance plot.

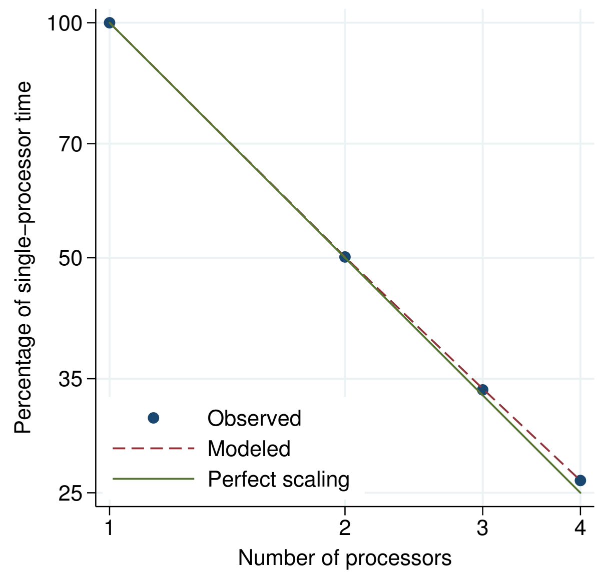

Figure A.168. `predict, stdp` performance plot.

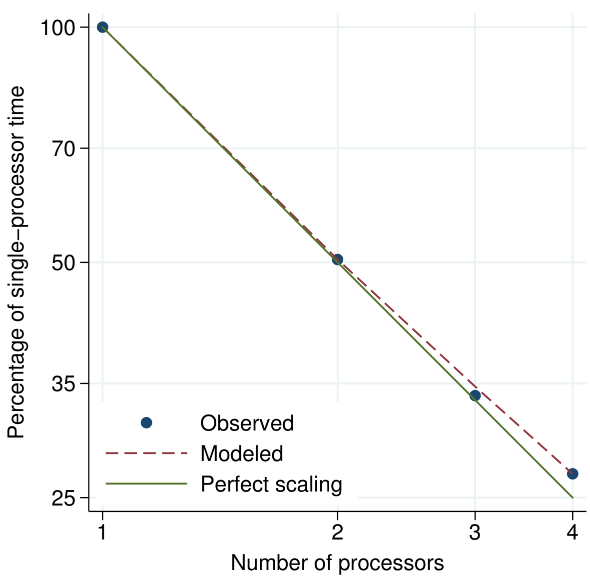

Figure A.169. `predict, stdr` performance plot.

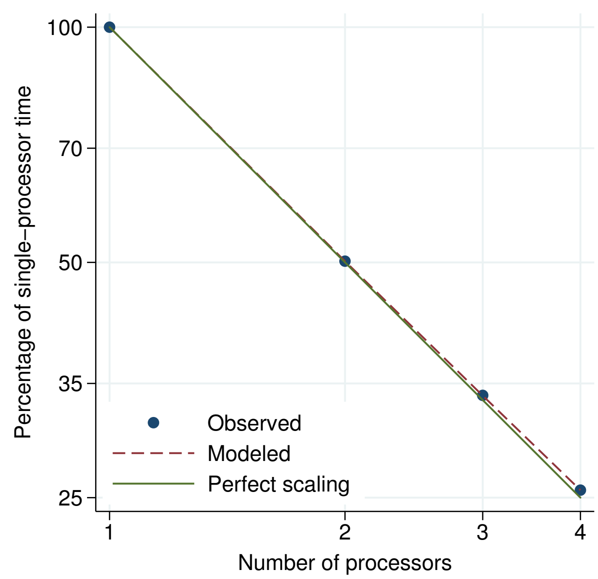

Figure A.170. `predict, welsch` performance plot.

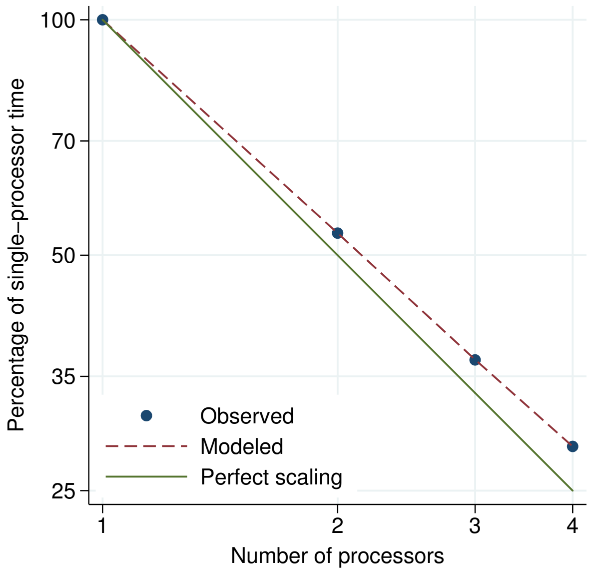

Figure A.171. `predict, ystar` performance plot.

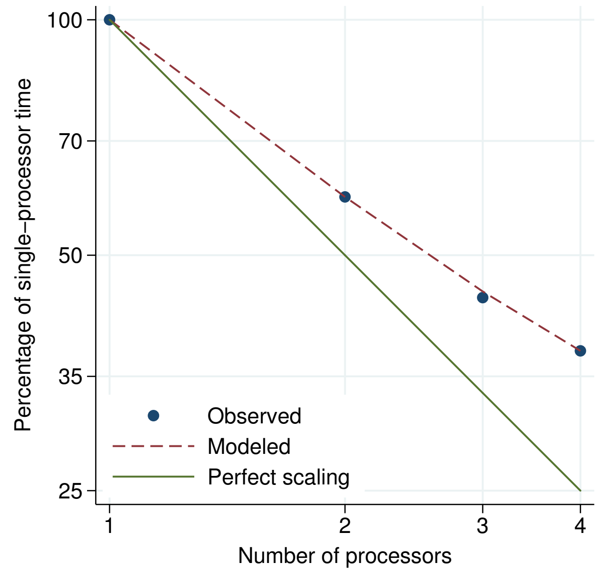

Figure A.172. `predictnl` performance plot.

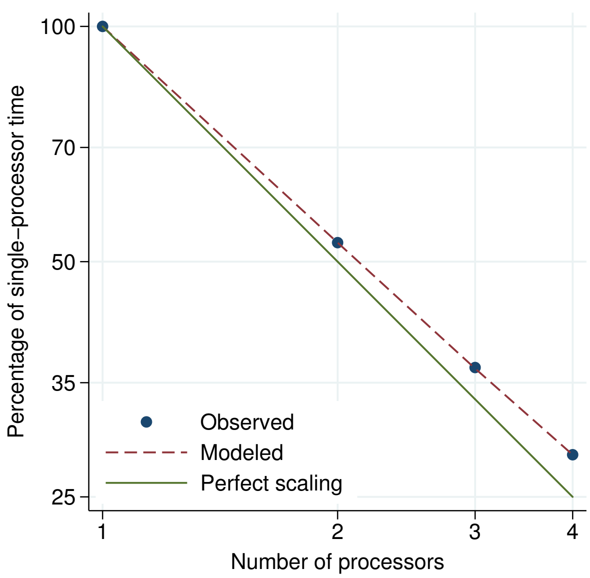

Figure A.173. `probit` performance plot.

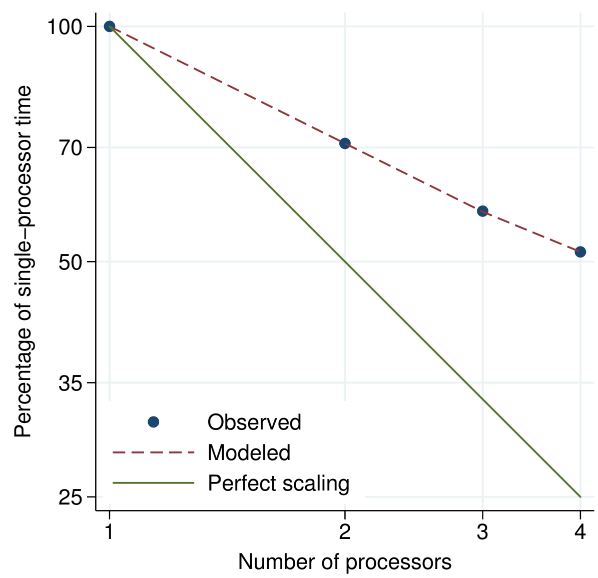

Figure A.174. `procrustes` performance plot.

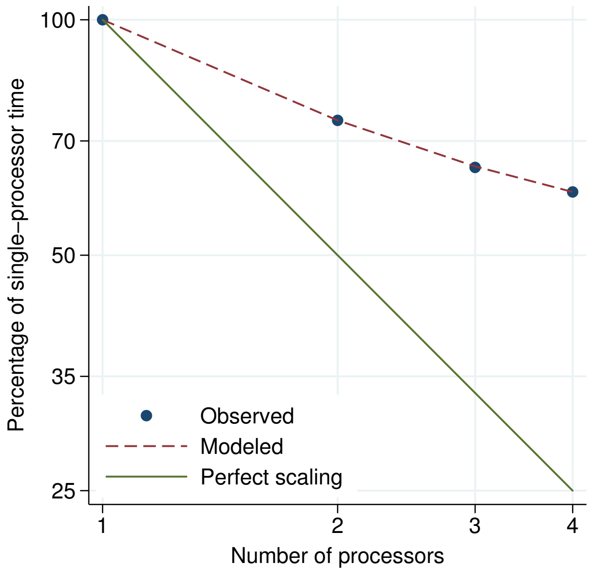

Figure A.175. `proportion` performance plot.

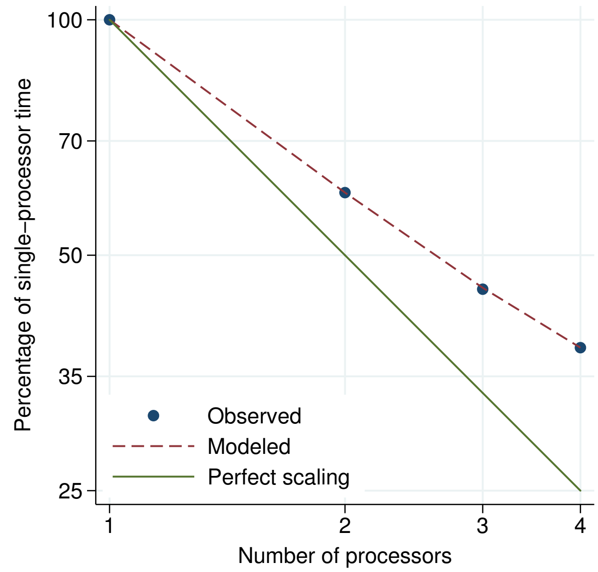

Figure A.176. `prtest1` performance plot.

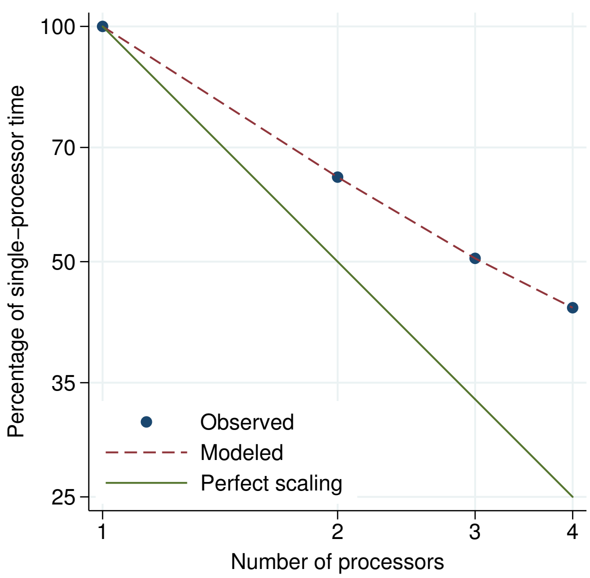

Figure A.177. `prtest2` performance plot.

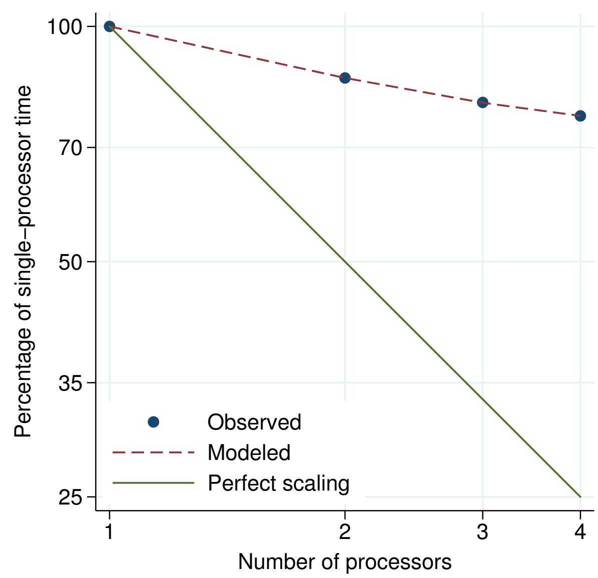

Figure A.178. `prtest, by()` performance plot.

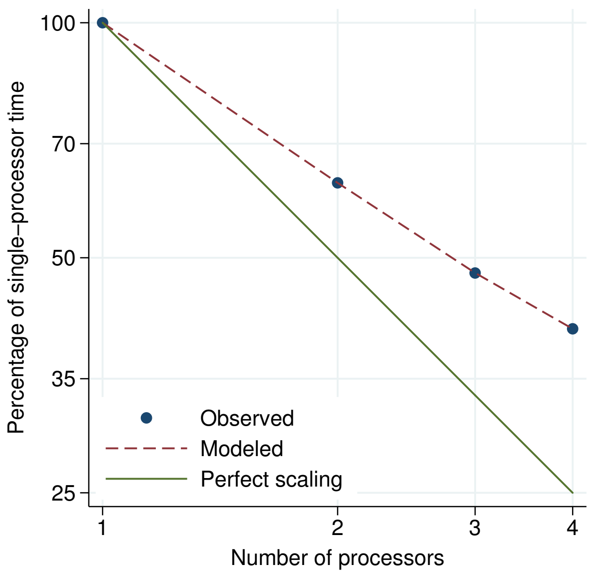

Figure A.179. `qreg` performance plot.

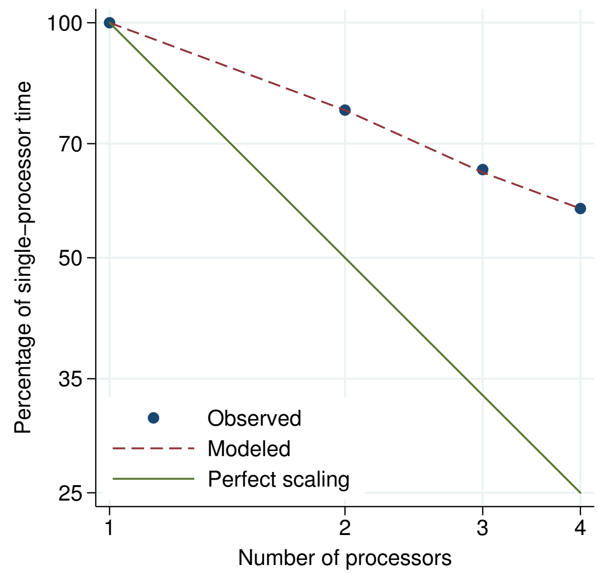

Figure A.180. `ranksum` performance plot.

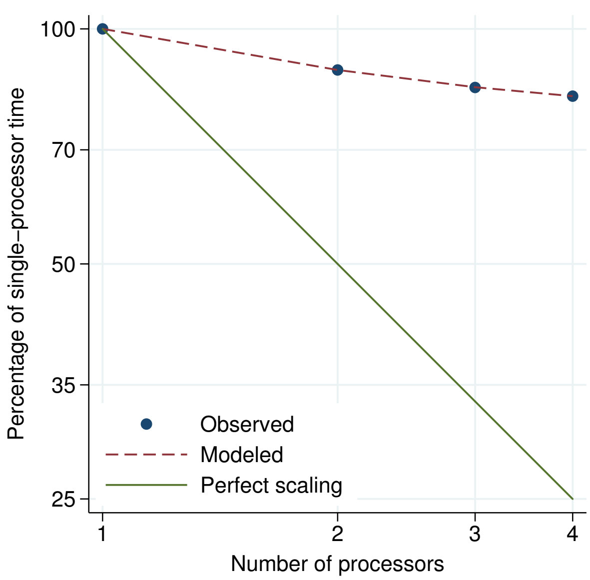

Figure A.181. `ratio` performance plot.

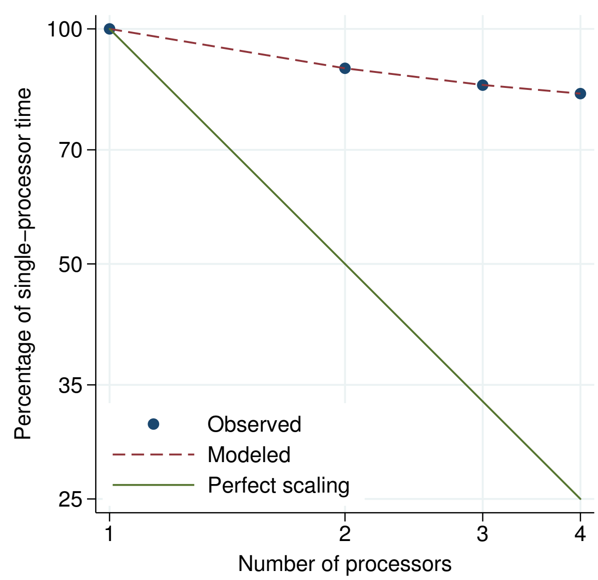

Figure A.182. `ratio (exp1) (exp2)` performance plot.

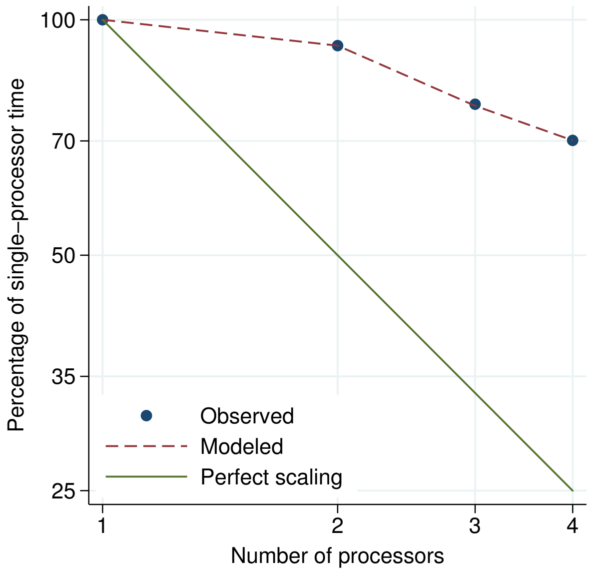

Figure A.183. `recode` performance plot.

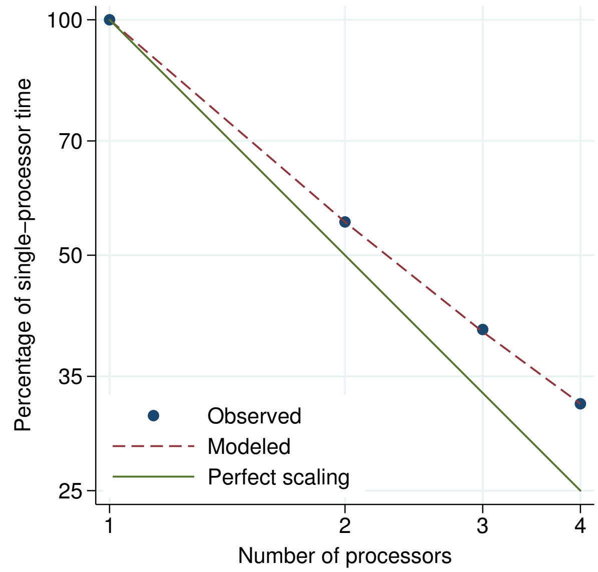

Figure A.184. `reg3` performance plot.

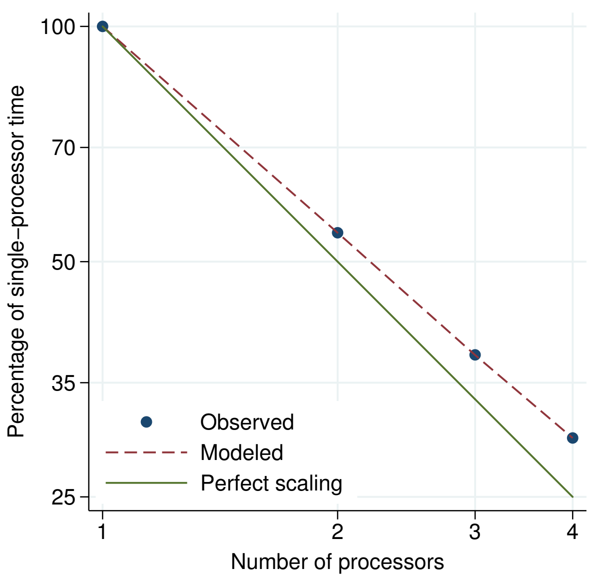

Figure A.185. `regress` performance plot.

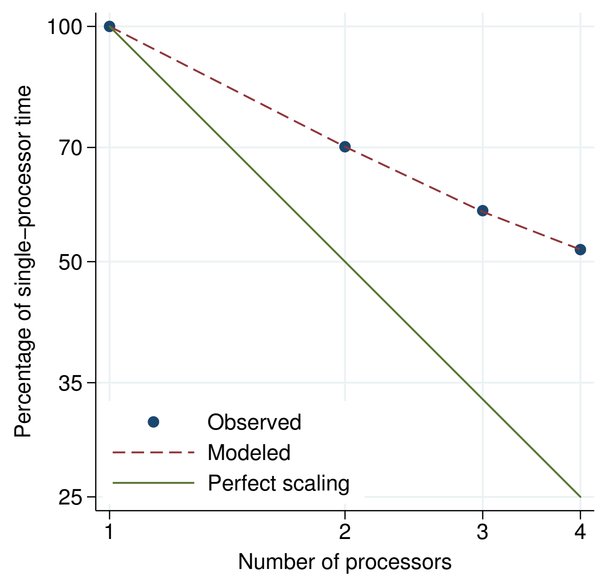

Figure A.186. `regress, cluster()` performance plot.

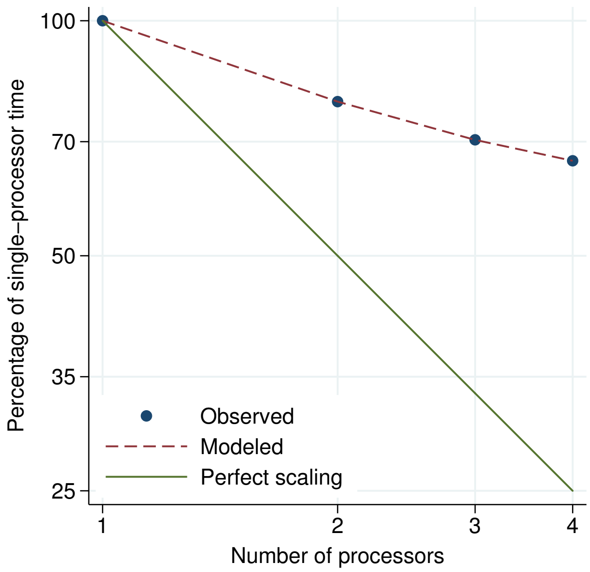

Figure A.187. `regress, robust` performance plot.

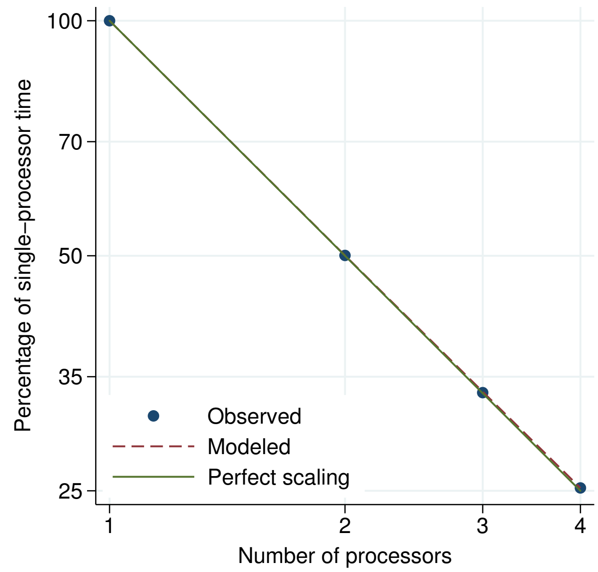

Figure A.188. `replace` performance plot.

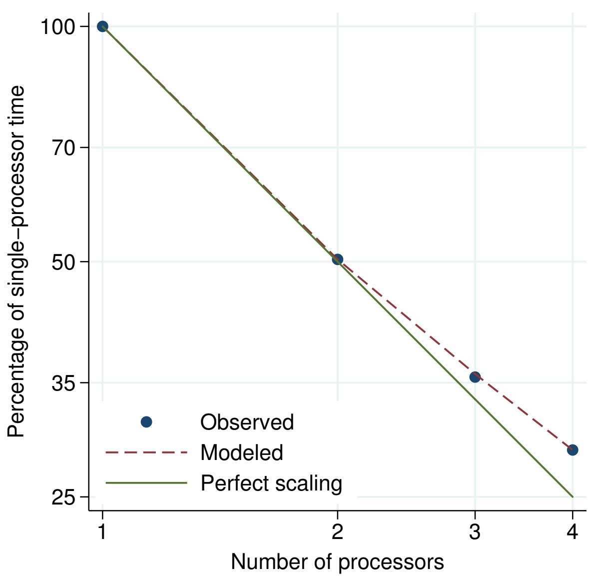

Figure A.189. `replace (small expression)` performance plot.

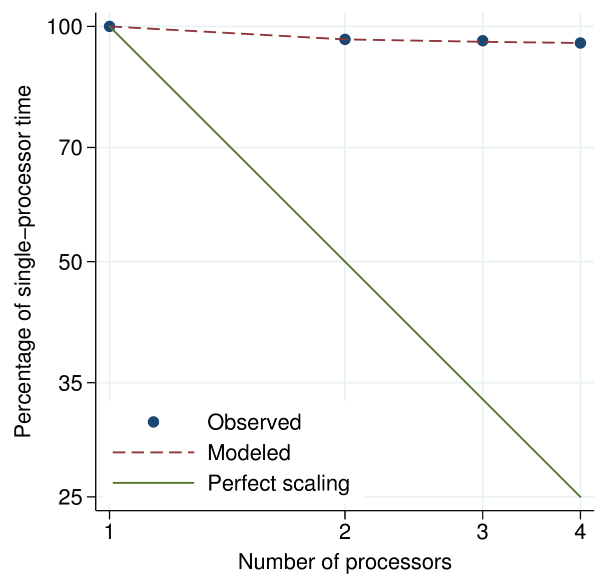

Figure A.190. `reshape long` performance plot.

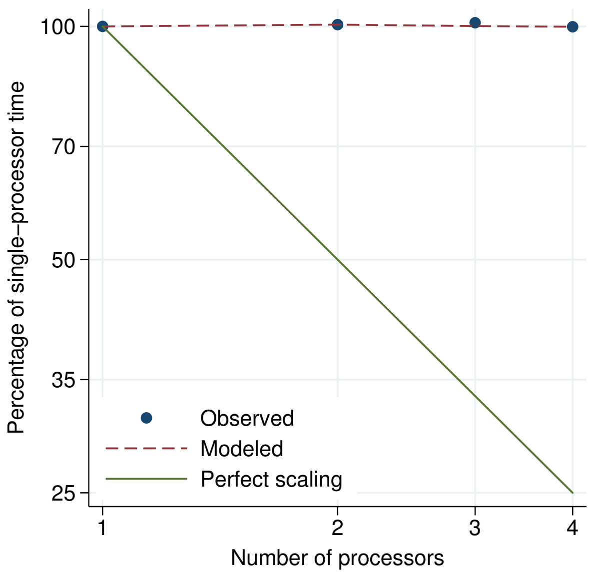

Figure A.191. `reshape wide` performance plot.

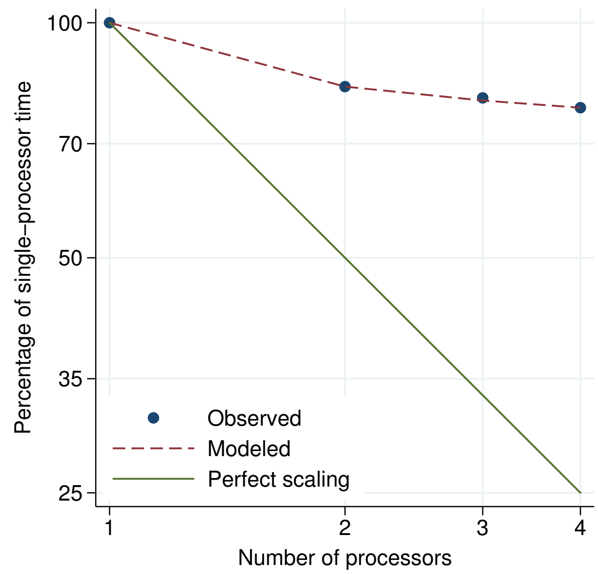

Figure A.192. `robvar` performance plot.

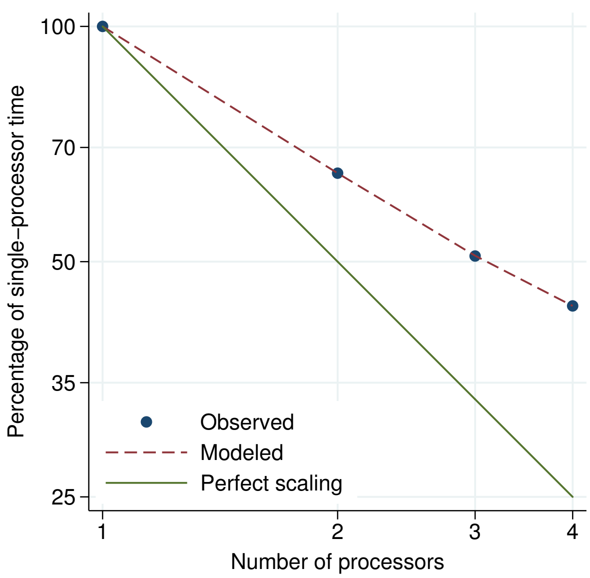

Figure A.193. `rocfit` performance plot.

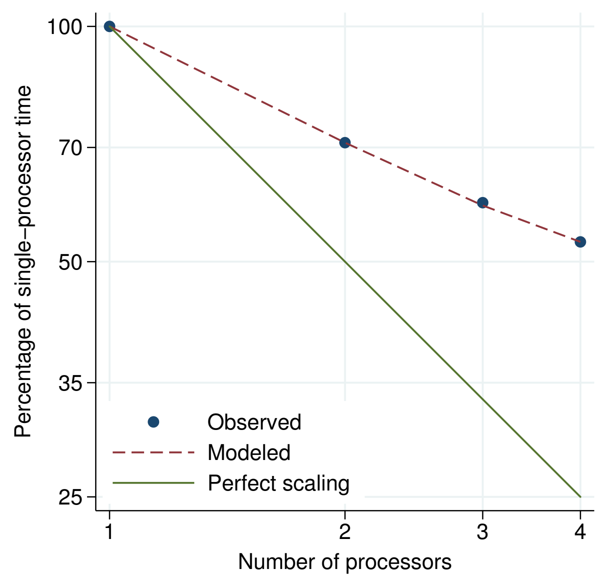

Figure A.194. `roctab` performance plot.

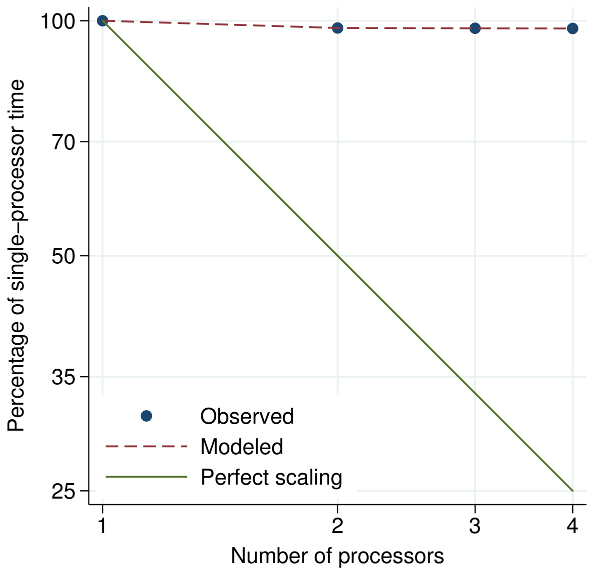

Figure A.195. `rotatemat` performance plot.

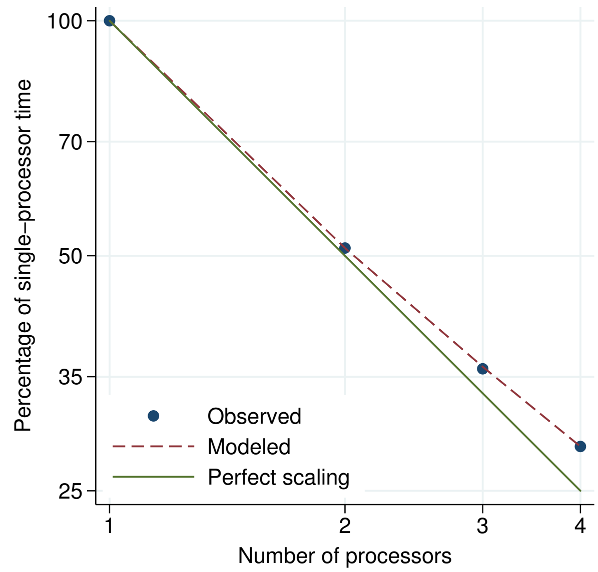

Figure A.196. `rreg` performance plot.

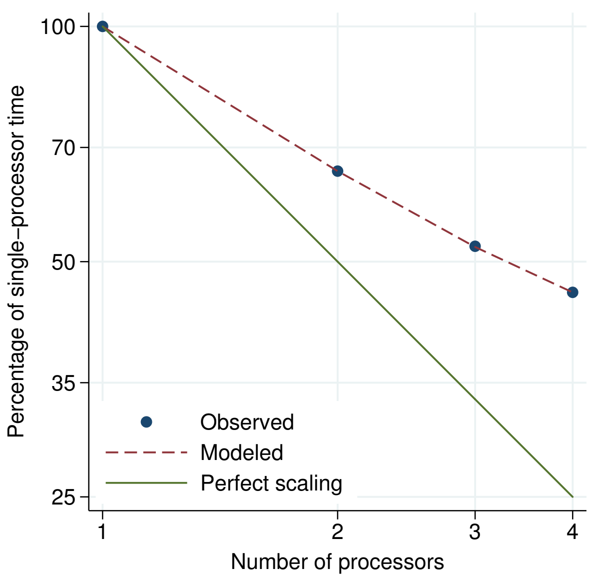

Figure A.197. `runtest` performance plot.

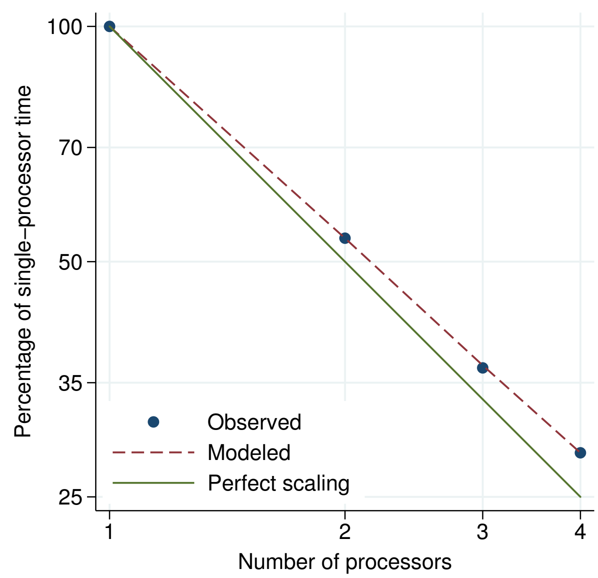

Figure A.198. `scobit` performance plot.

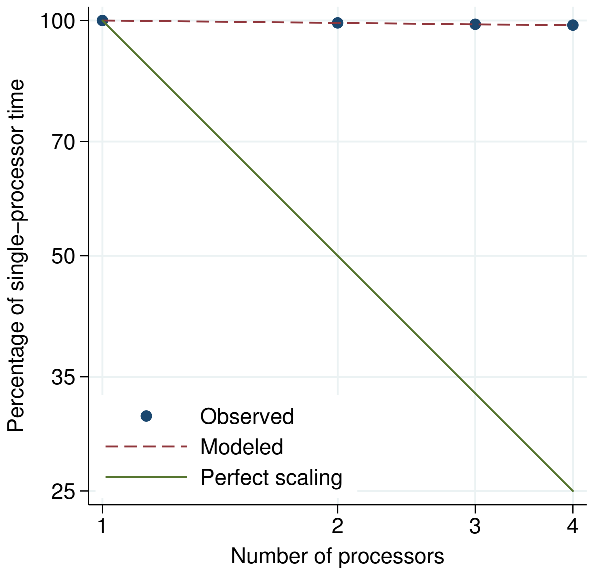

Figure A.199. `scoreplot` performance plot.

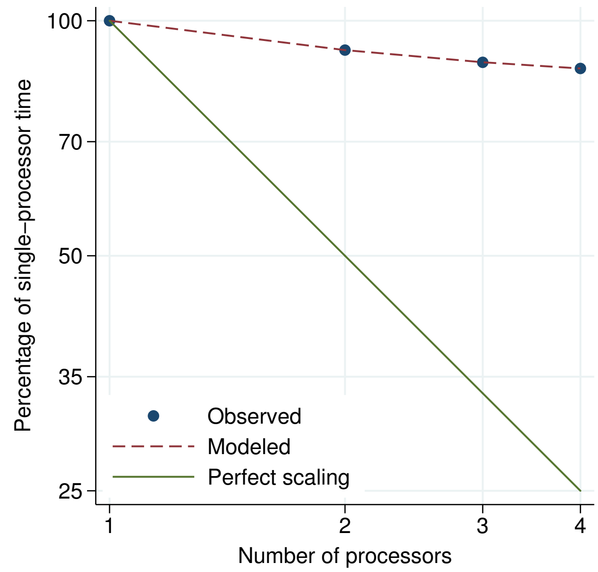

Figure A.200. `screeplot` performance plot.

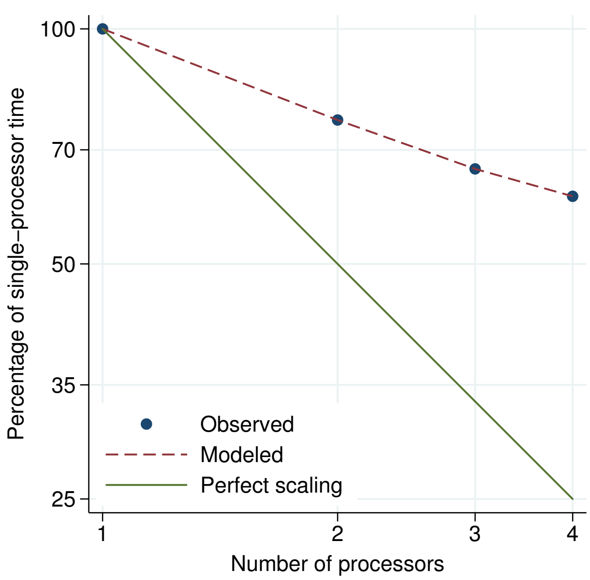

Figure A.201. `sdtest1` performance plot.

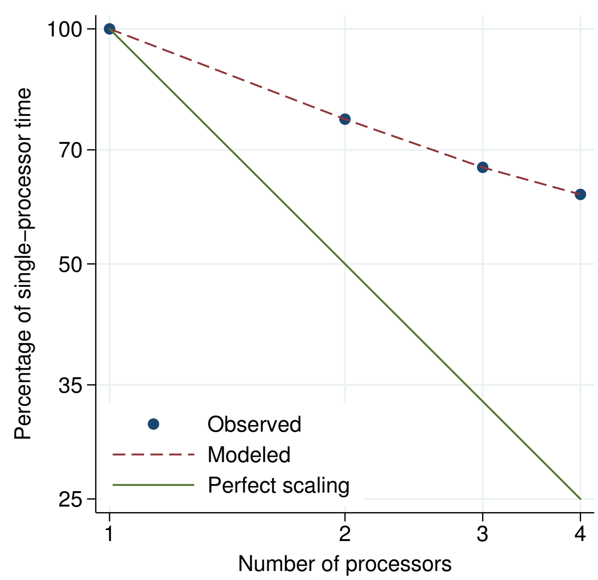

Figure A.202. `sdtest2` performance plot.

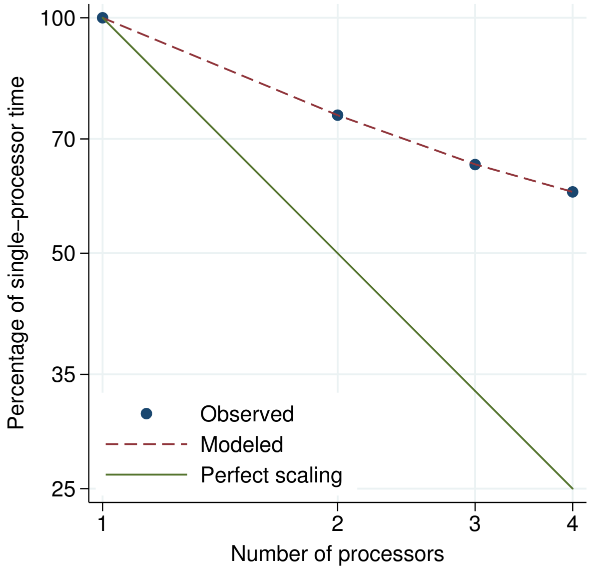

Figure A.203. `sdtest, by()` performance plot.

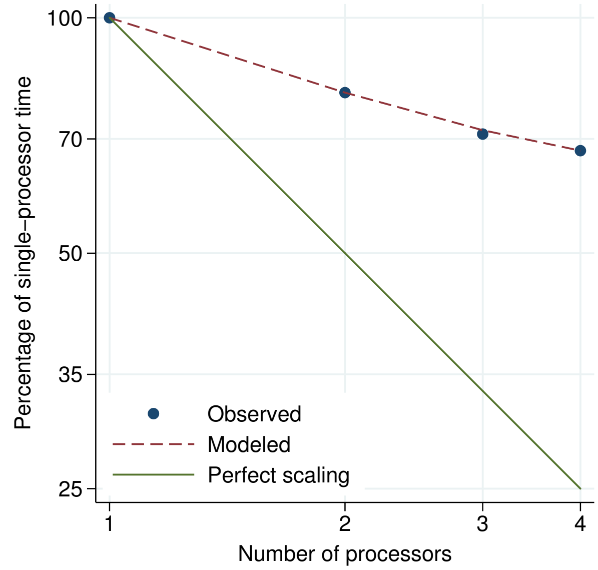

Figure A.204. `sfrancia` performance plot.

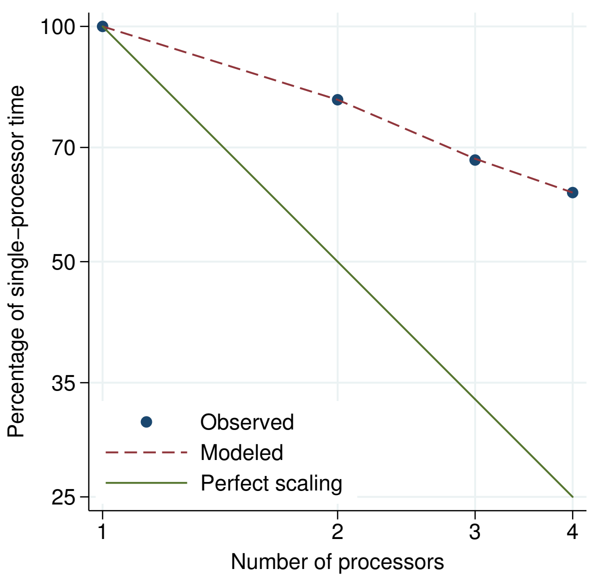

Figure A.205. `signrank` performance plot.

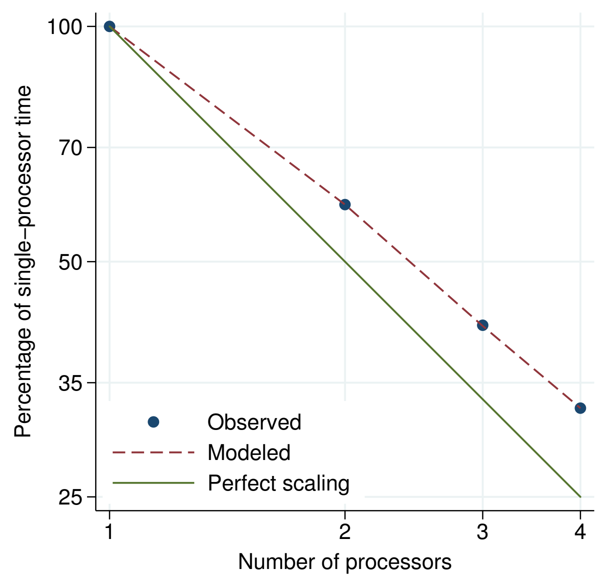

Figure A.206. `signtest` performance plot.

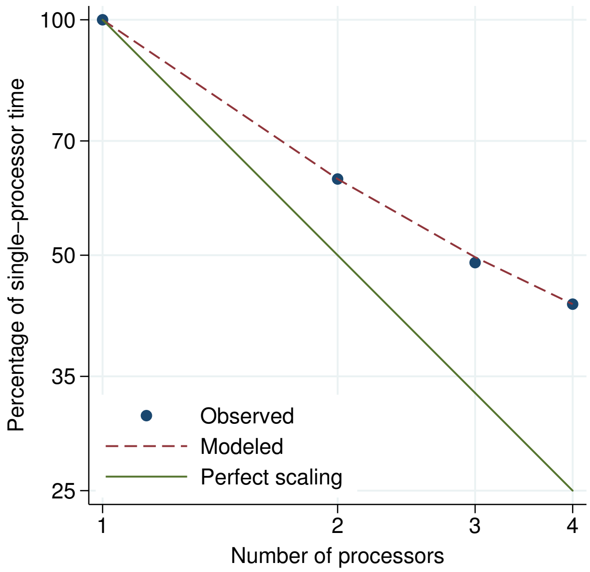

Figure A.207. `sktest` performance plot.

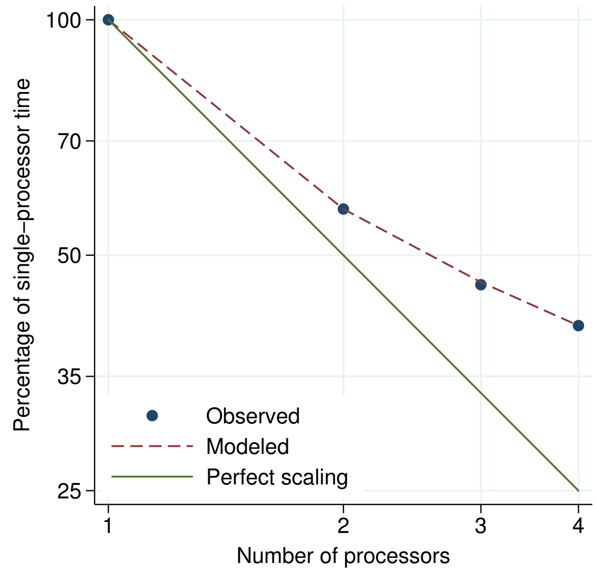

Figure A.208. `slogit` performance plot.

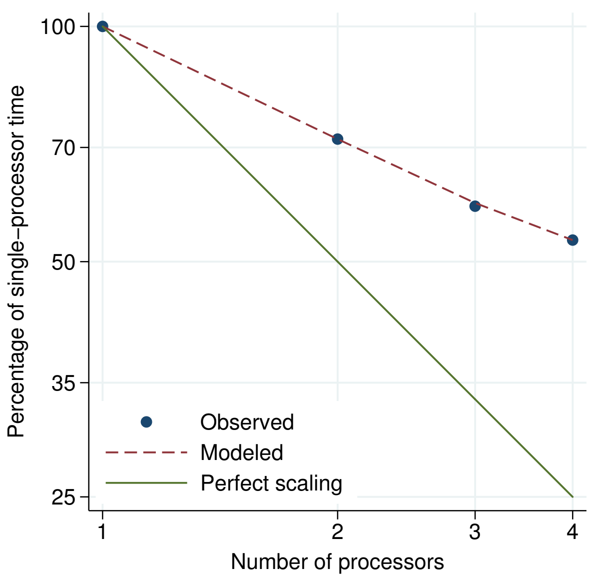

Figure A.209. `sort` performance plot.

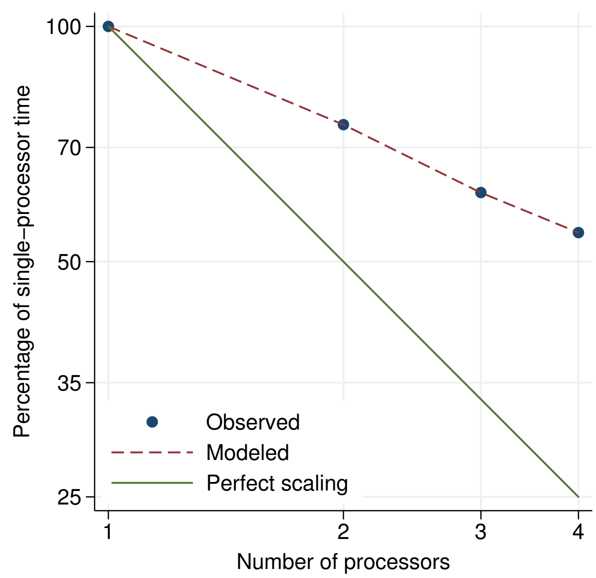

Figure A.210. `spearman` performance plot.

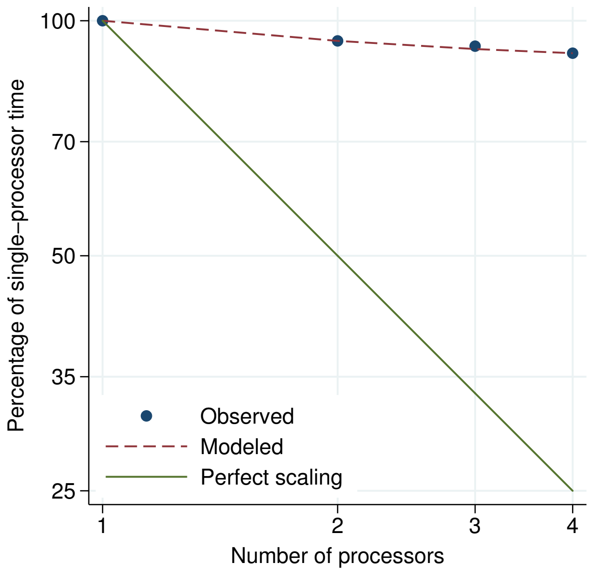

Figure A.211. `stack` performance plot.

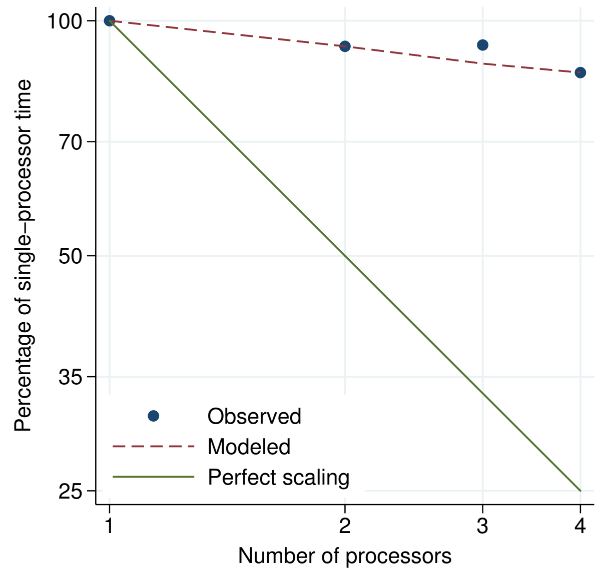

Figure A.212. `stbase` performance plot.

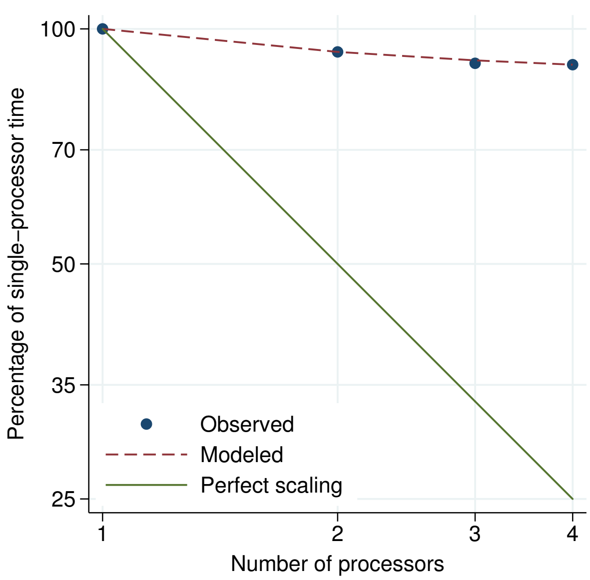

Figure A.213. `stci` performance plot.

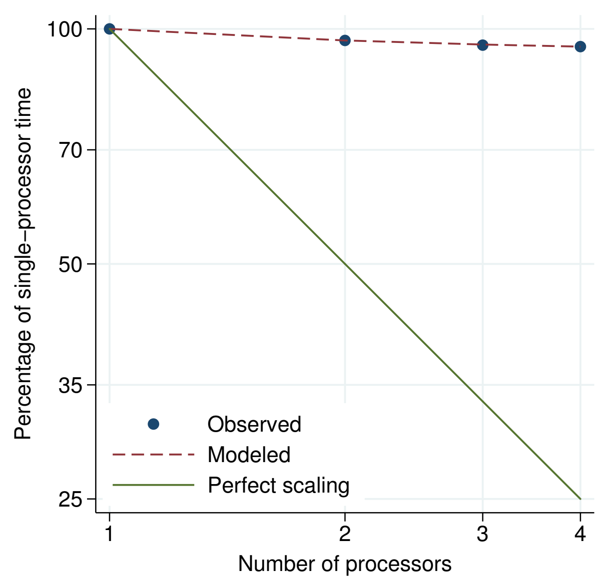

Figure A.214. `stcox` performance plot.

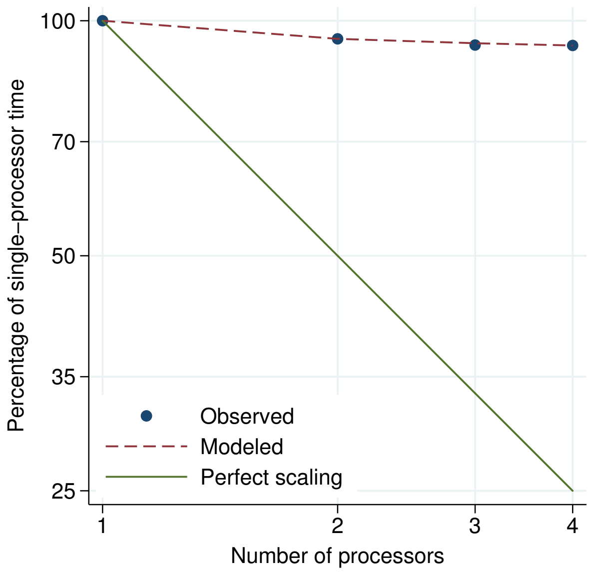

Figure A.215. `stcurve, hazard (after stcox)` performance plot.

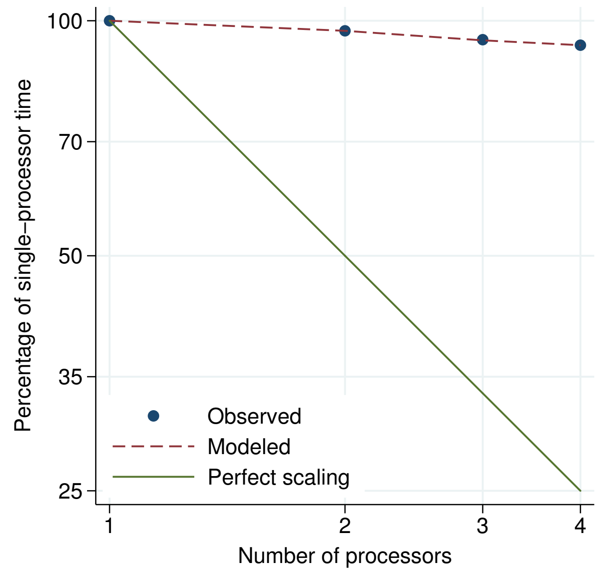

Figure A.216. `stcurve, hazard (after streg)` performance plot.

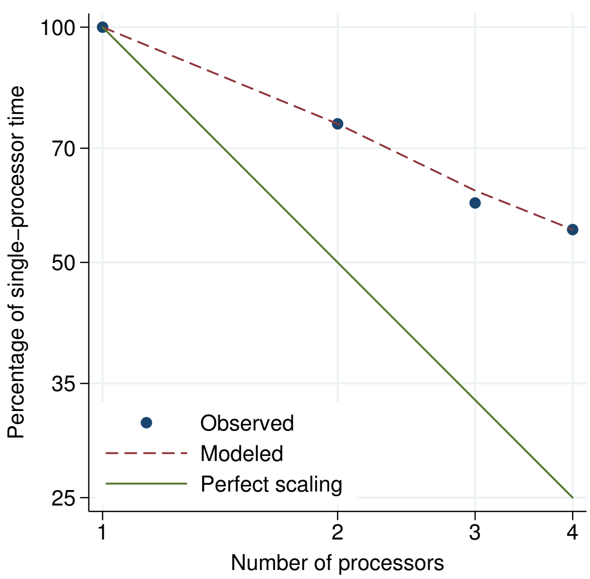

Figure A.217. `stgen` performance plot.

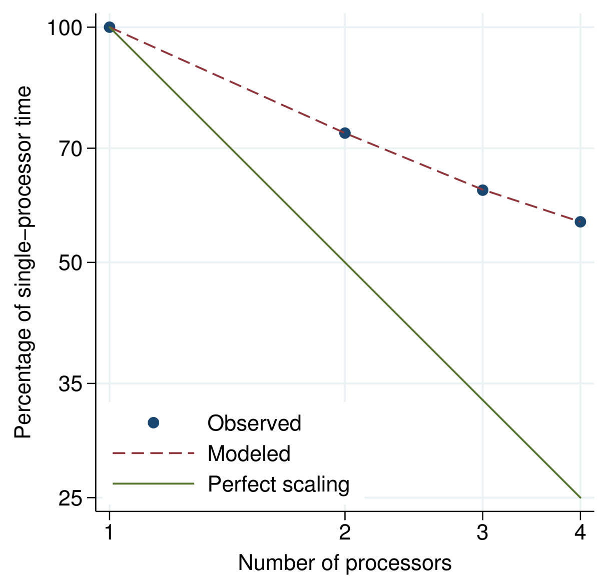

Figure A.218. `stir` performance plot.

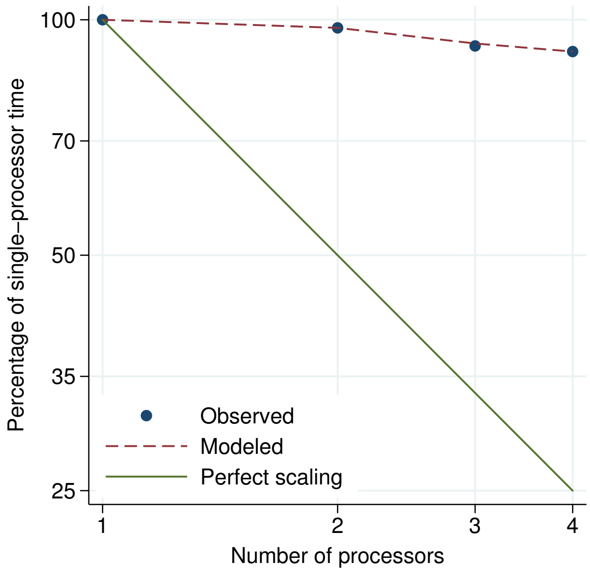

Figure A.219. `stptime` performance plot.

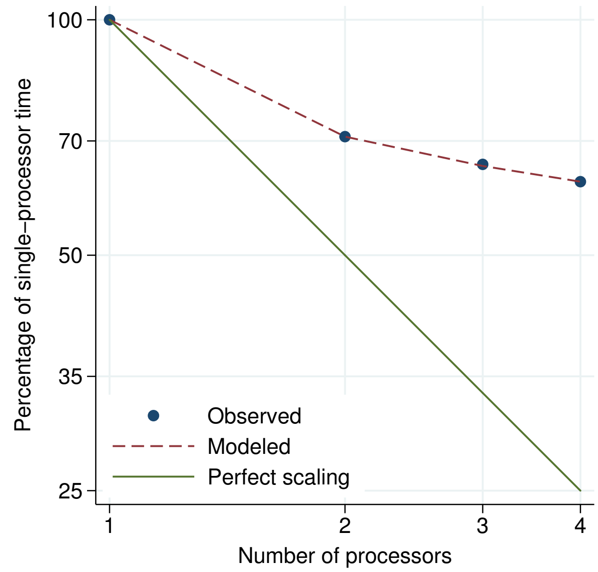

Figure A.220. `strate` performance plot.

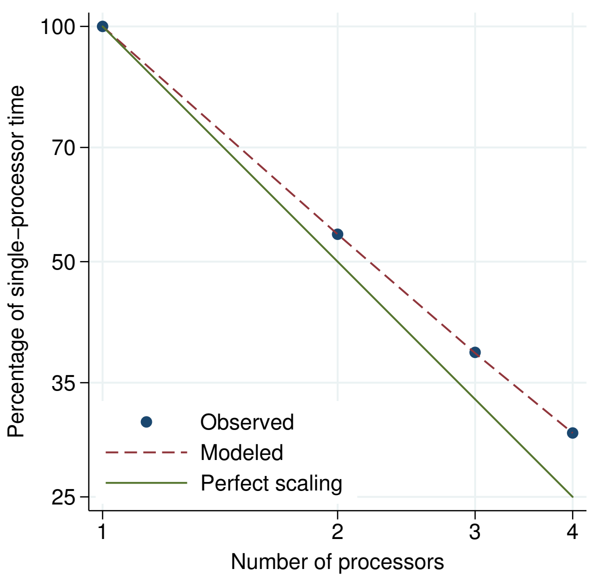

Figure A.221. `streg, distribution(exponential)` performance plot.

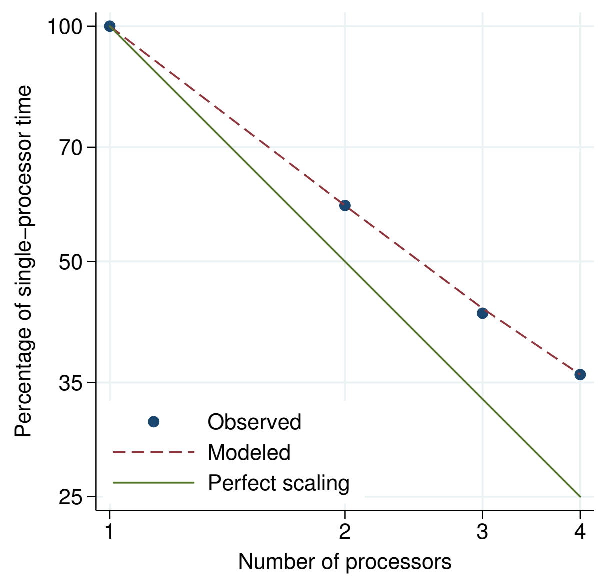

Figure A.222. `streg, dist(exp) cluster()` performance plot.

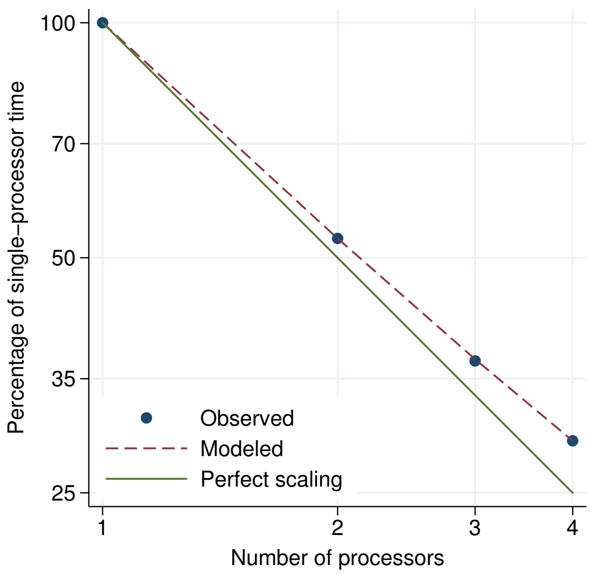

Figure A.223. `streg, dist(exp) frailty()` performance plot.

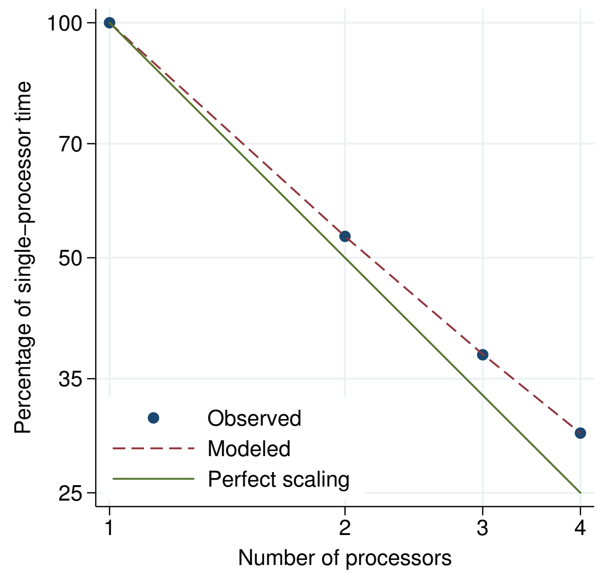

Figure A.224. `streg, dist(exp) frailty() shared()` performance plot.

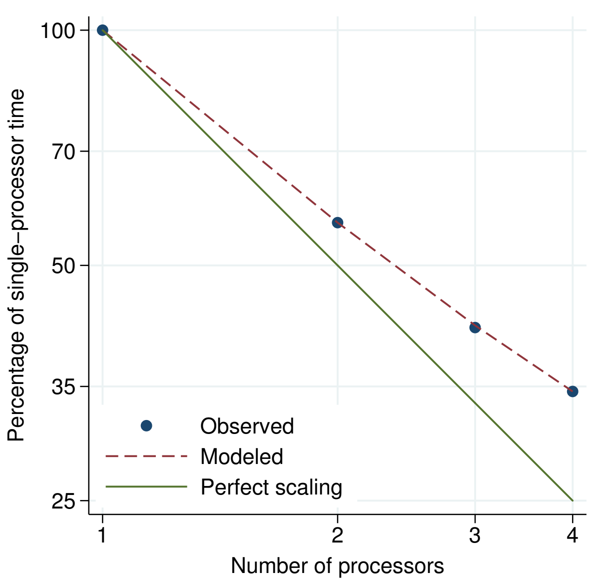

Figure A.225. `streg, dist(exp) robust` performance plot.

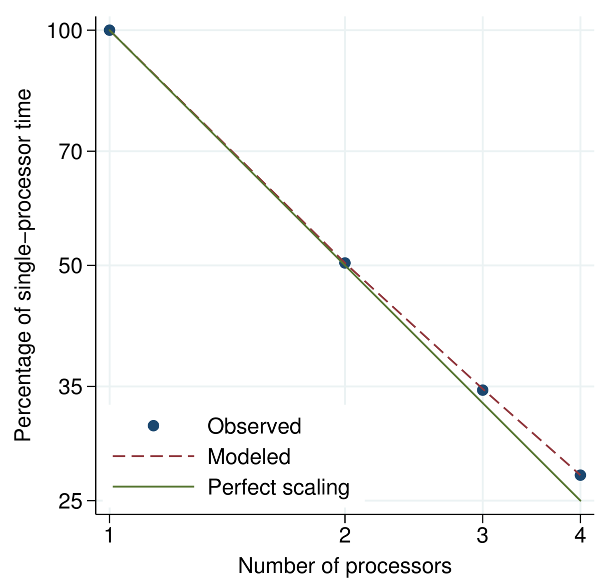

Figure A.226. `streg, distribution(gamma)` performance plot.

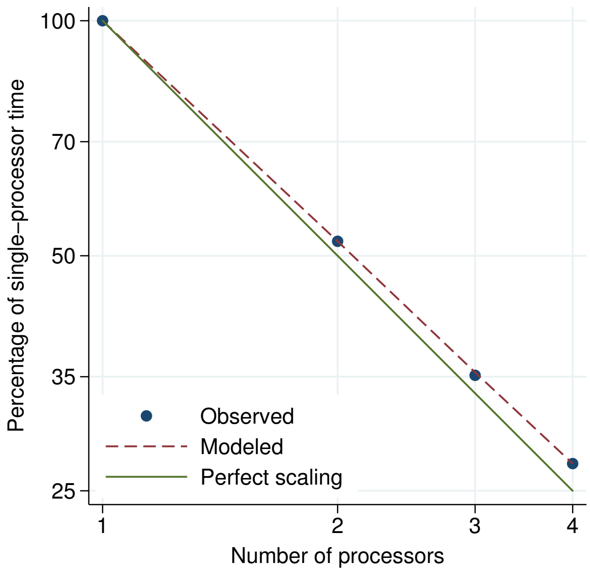

Figure A.227. `streg, distribution(lnormal)` performance plot.

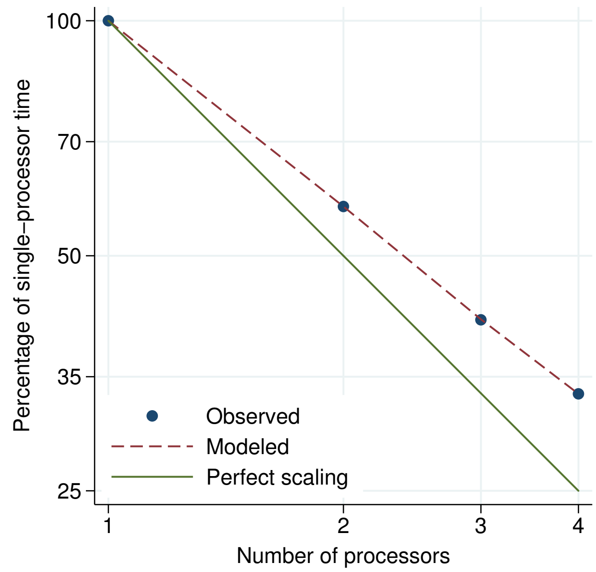

Figure A.228. `streg, distribution(weibull)` performance plot.

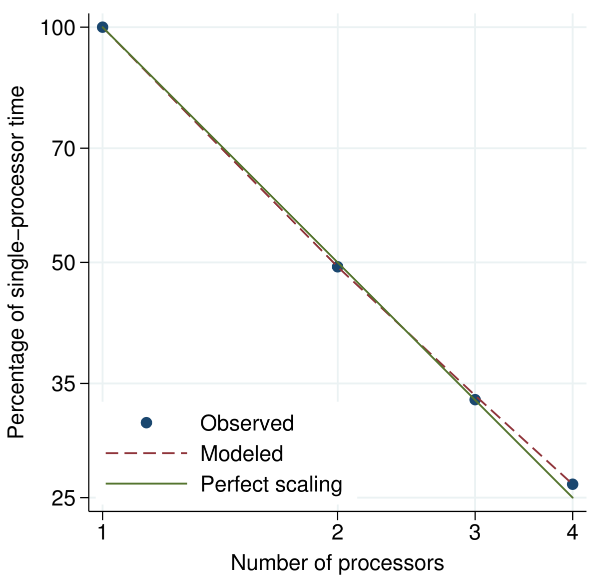

Figure A.229. `streg, dist(weibull) frailty()` performance plot.

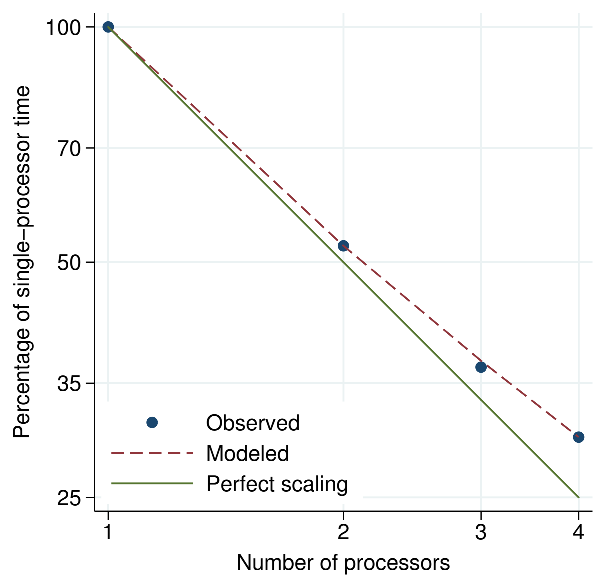

Figure A.230. `streg, dist(weibull) frailty() shared()` performance plot.

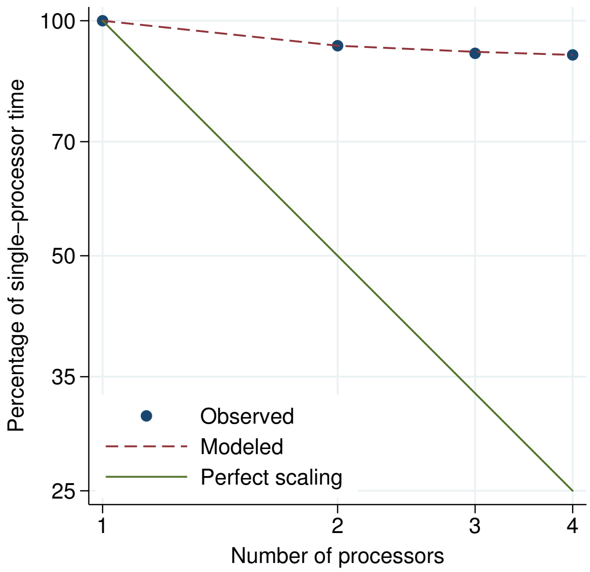

Figure A.231. `sts generate` performance plot.

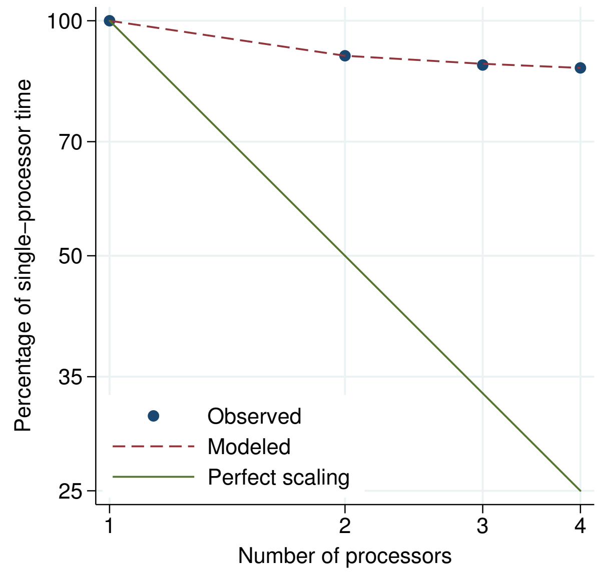

Figure A.232. `sts graph` performance plot.

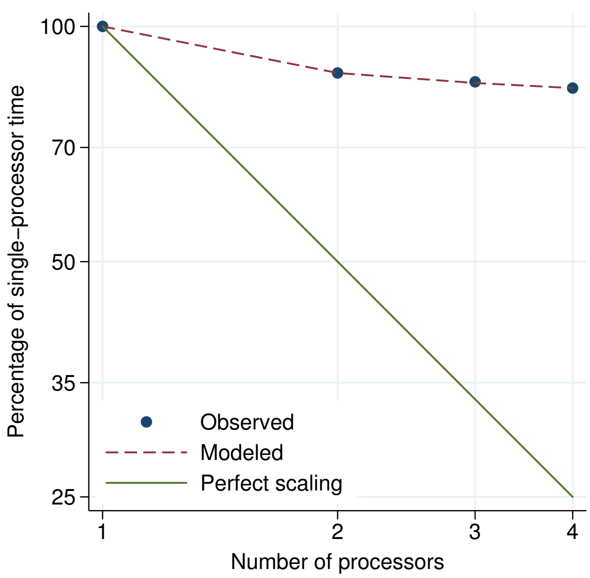

Figure A.233. `sts list` performance plot.

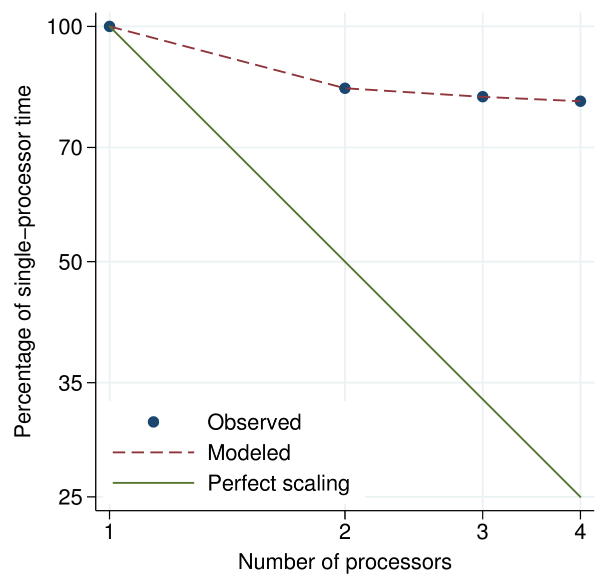

Figure A.234. `sts test` performance plot.

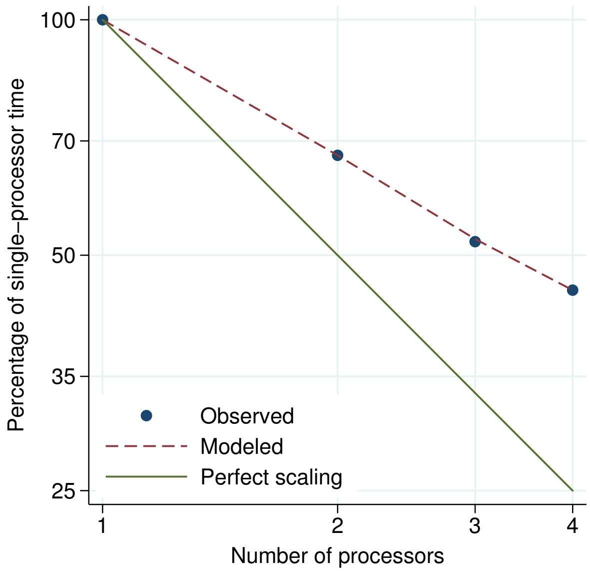

Figure A.235. `stset` performance plot.

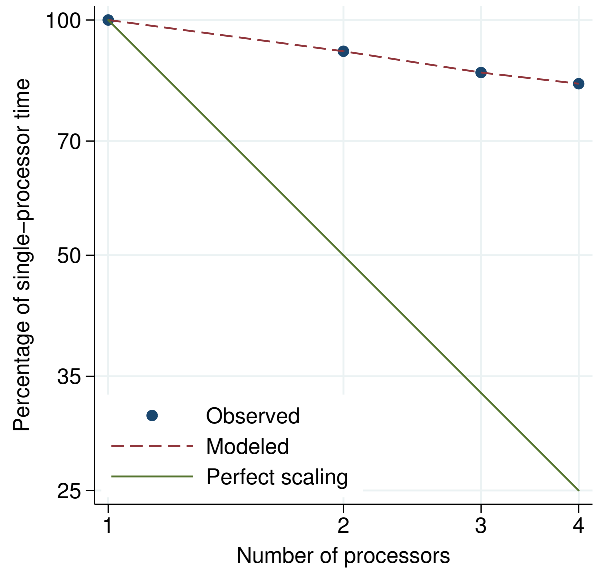

Figure A.236. `stsplit` performance plot.

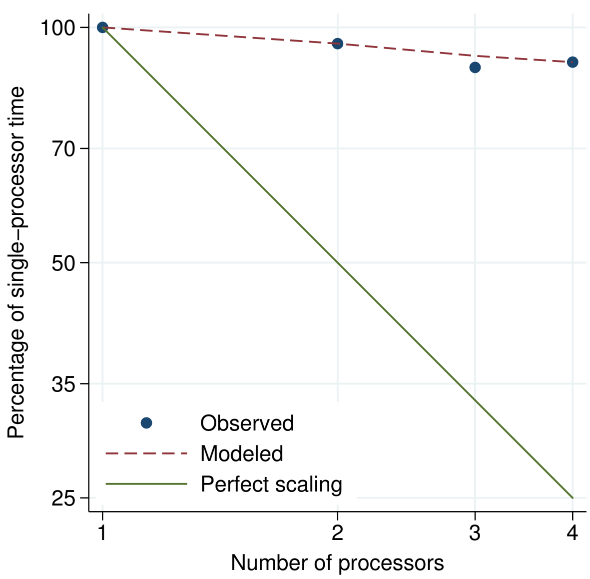

Figure A.237. `stsum` performance plot.

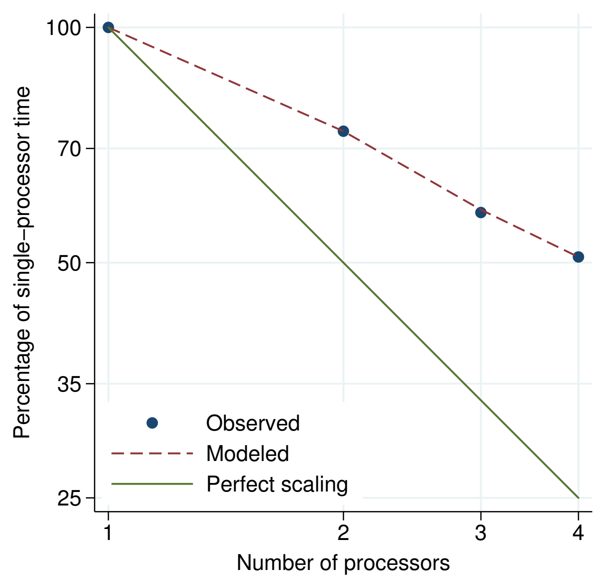

Figure A.238. `stvary` performance plot.

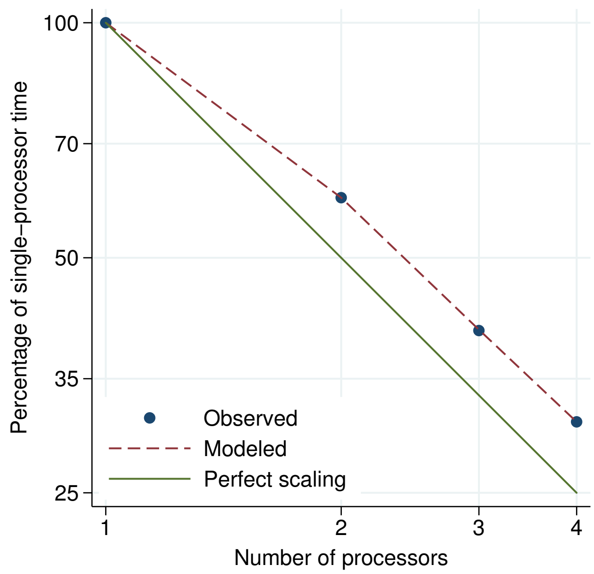

Figure A.239. `summarize` performance plot.

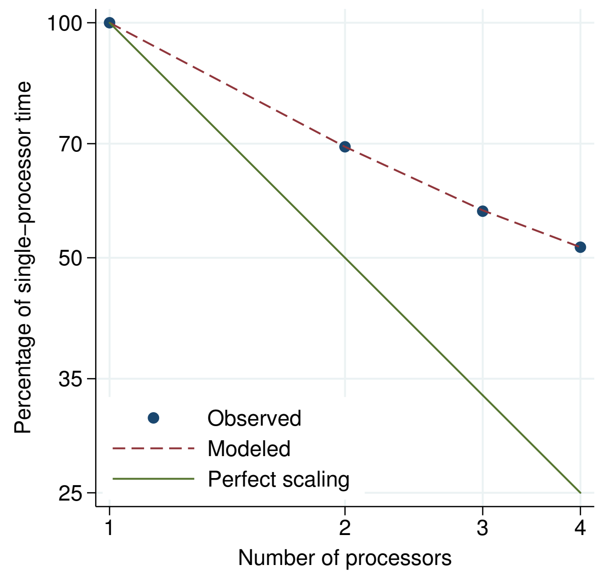

Figure A.240. `sunflower` performance plot.

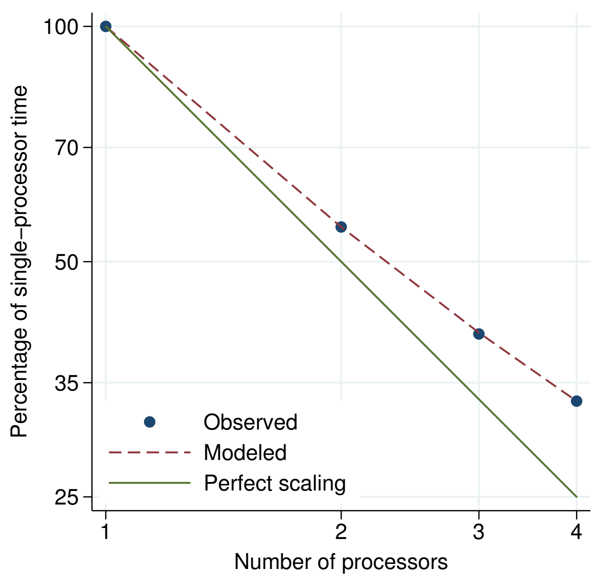

Figure A.241. `sureg` performance plot.

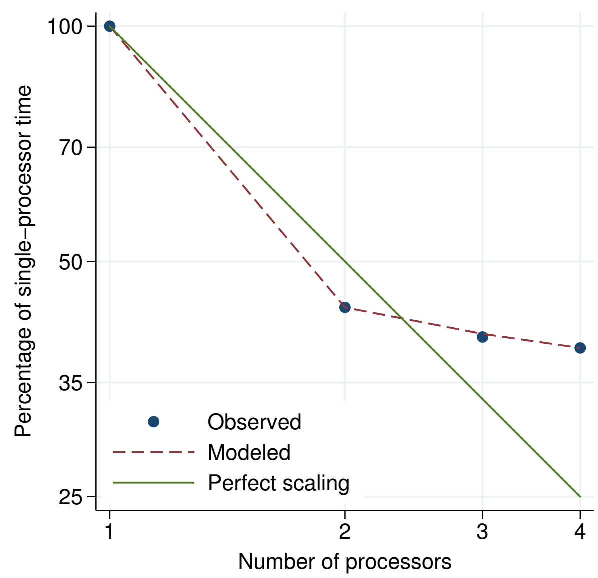

Figure A.242. `svar` performance plot.

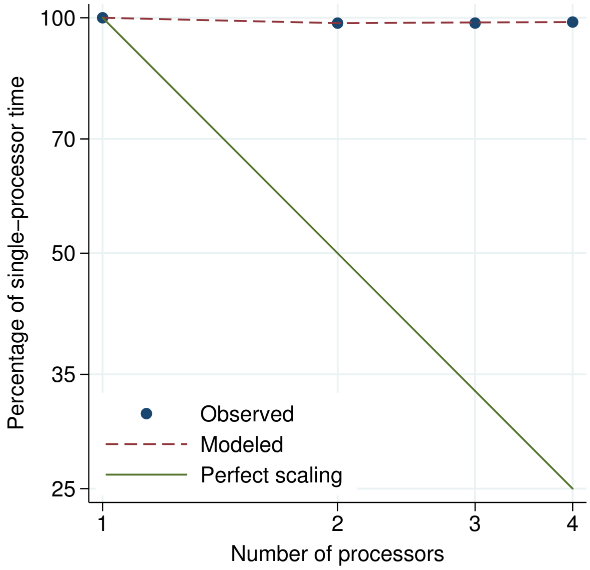

Figure A.243. `svmat` performance plot.

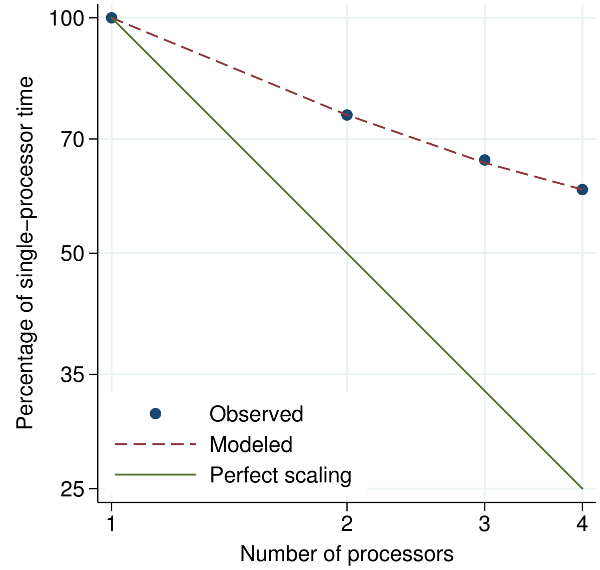

Figure A.244. `svy: logit` performance plot.

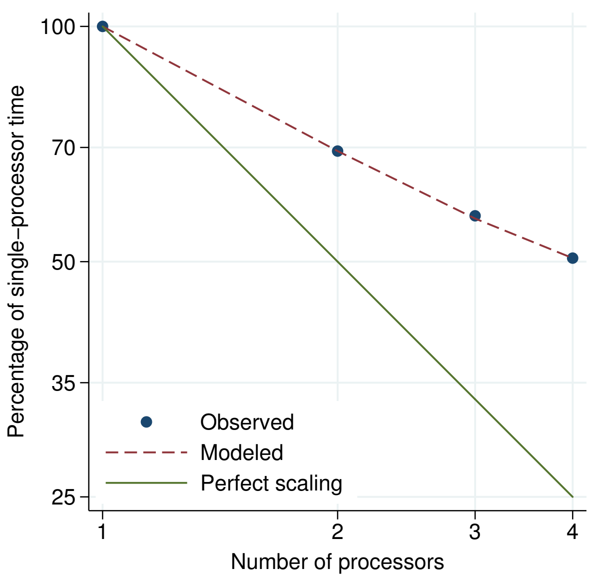

Figure A.245. `svy: poisson` performance plot.

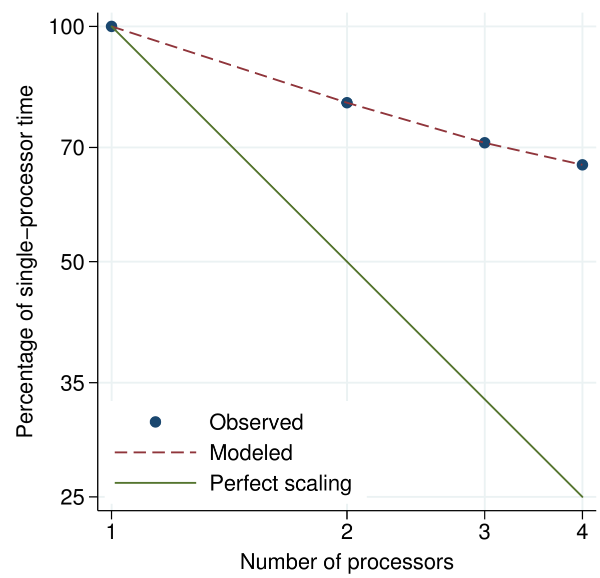

Figure A.246. `svy: regress` performance plot.

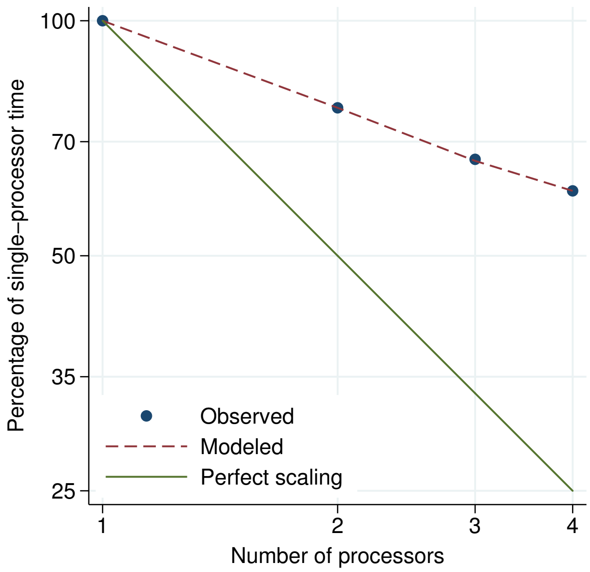

Figure A.247. `swilk` performance plot.

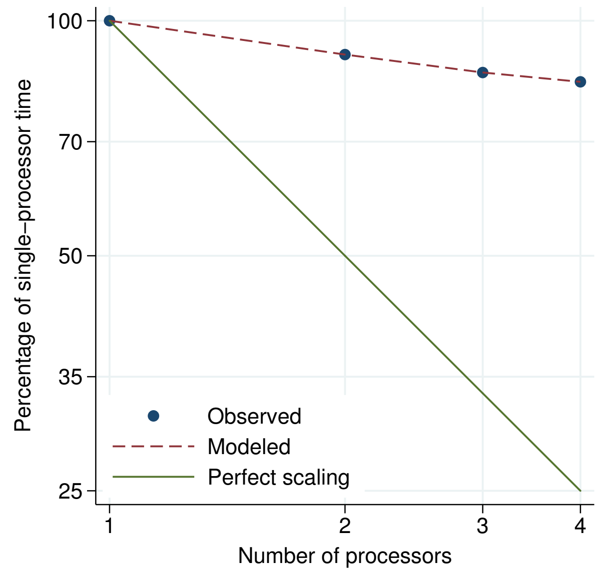

Figure A.248. `symmetry` performance plot.

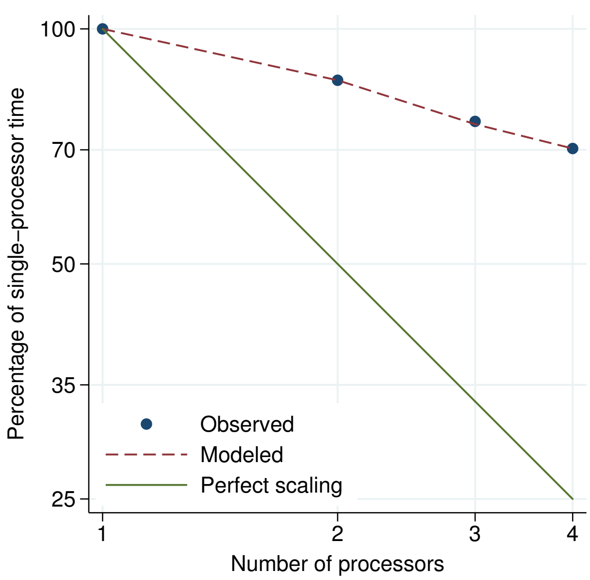

Figure A.249. `table (oneway)` performance plot.

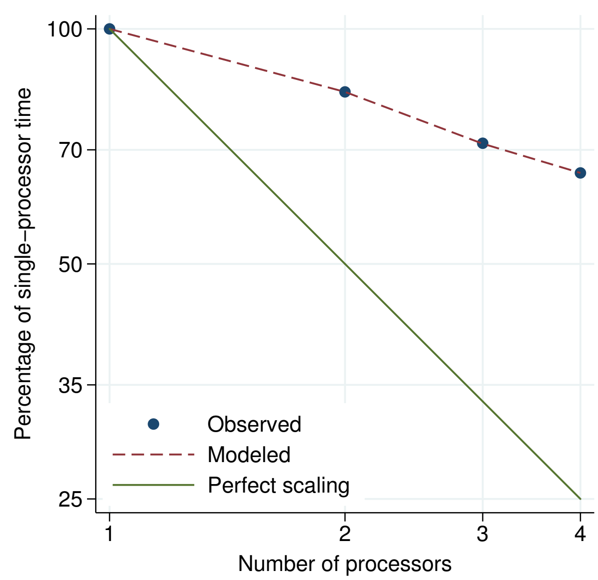

Figure A.250. `table (twoway)` performance plot.

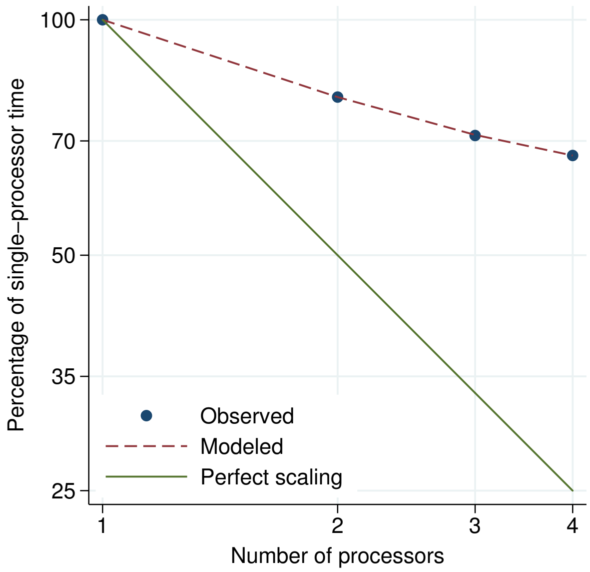

Figure A.251. `tabstat` performance plot.

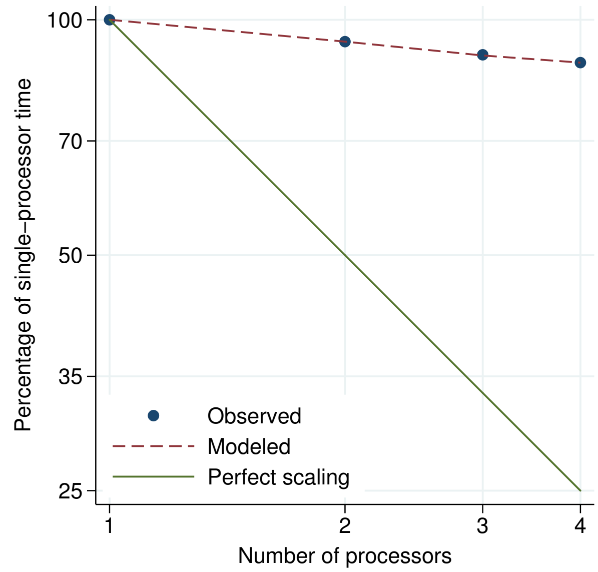

Figure A.252. `tabstat, by()` performance plot.

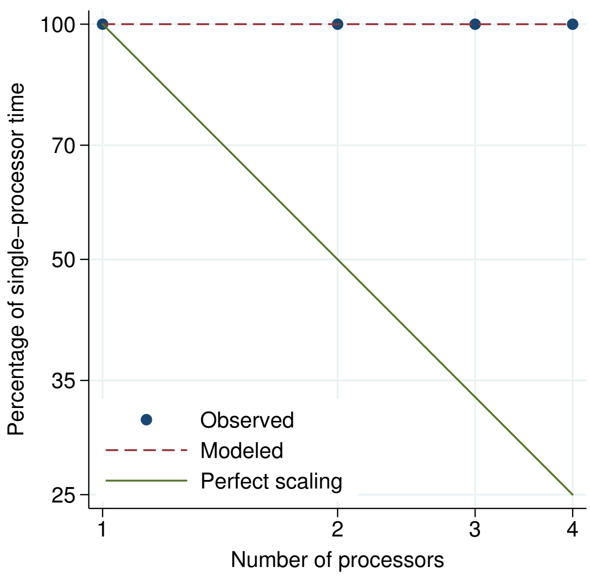

Figure A.253. `tabulate (oneway)` performance plot.

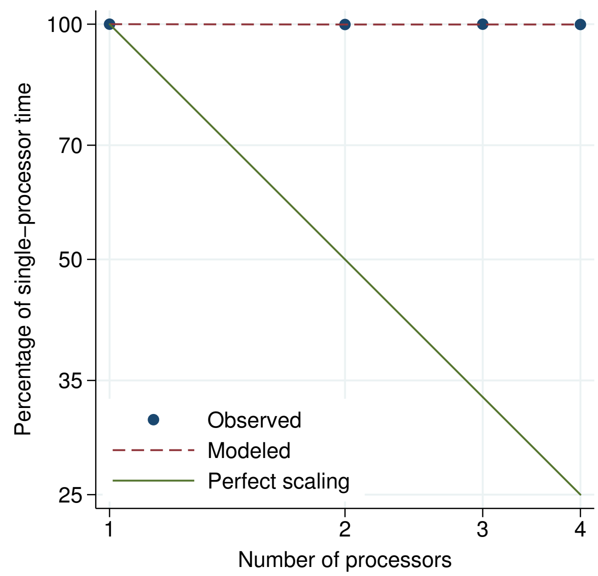

Figure A.254. `tabulate (twoway)` performance plot.

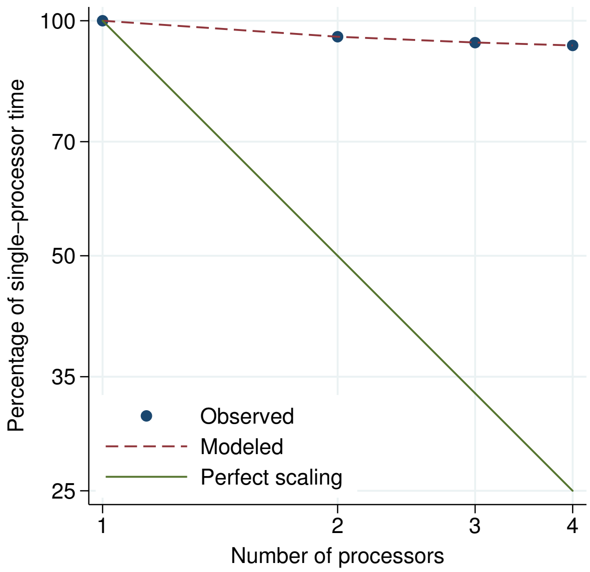

Figure A.255. `tetrachoric` performance plot.

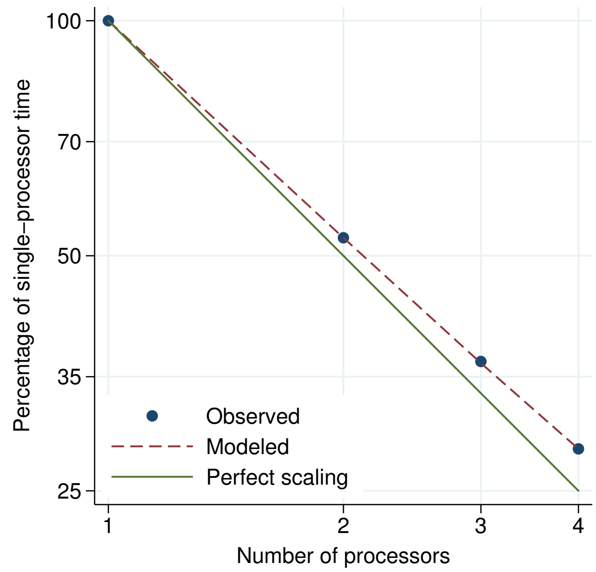

Figure A.256. `tobit` performance plot.

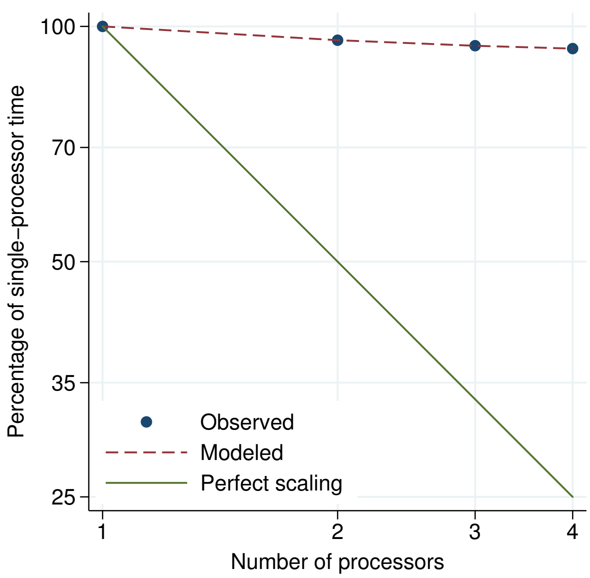

Figure A.257. `total` performance plot.

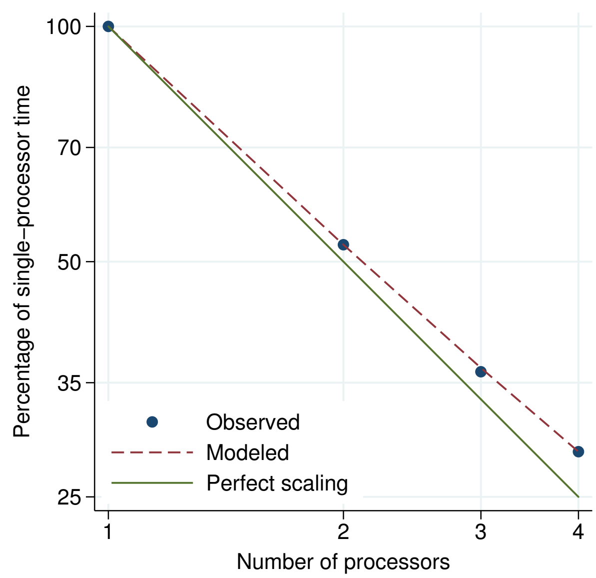

Figure A.258. `treatreg` performance plot.

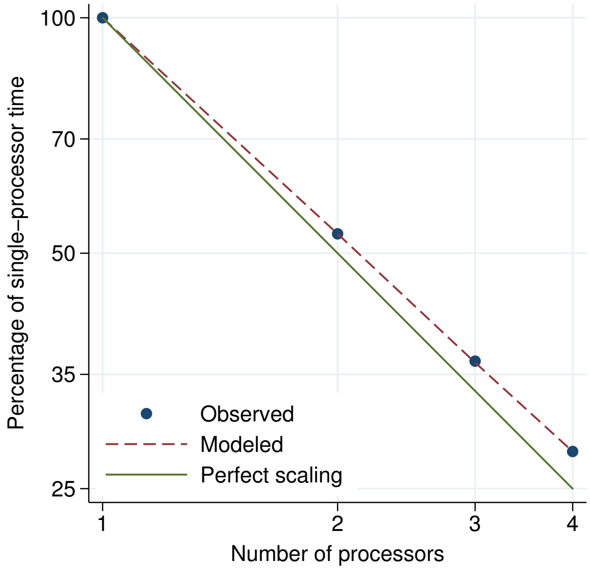

Figure A.259. `treatreg, twostep` performance plot.

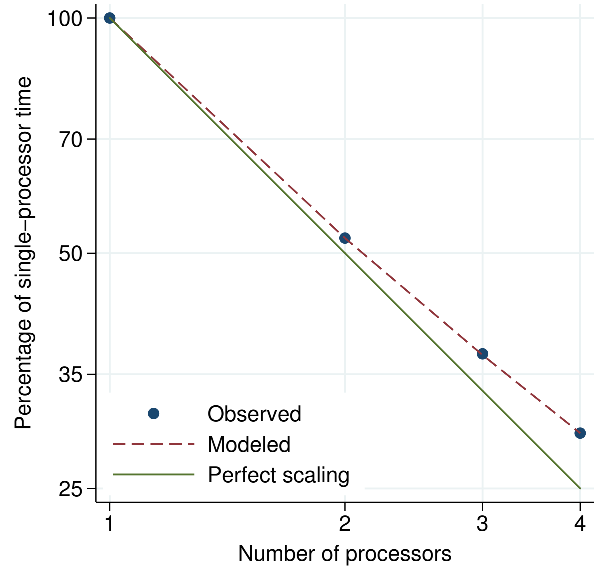

Figure A.260. `truncreg` performance plot.

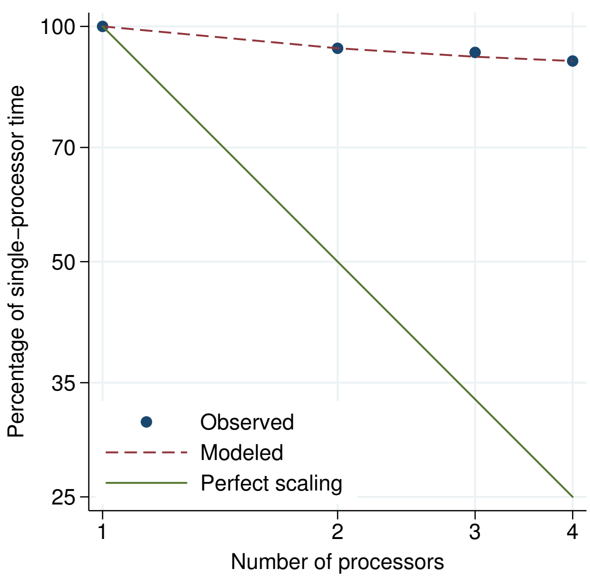

Figure A.261. `tsset` performance plot.

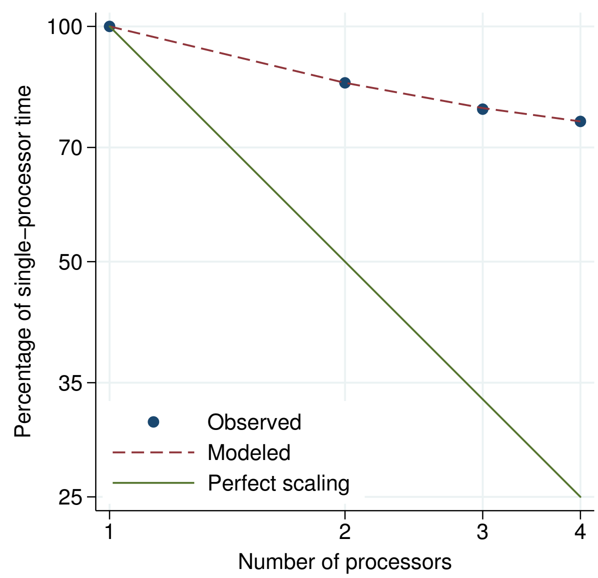

Figure A.262. `tssmooth exp` performance plot.

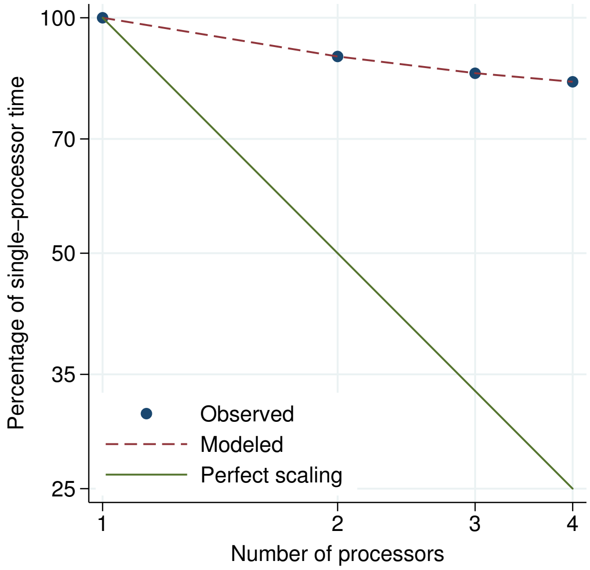

Figure A.263. `tssmooth ma` performance plot.

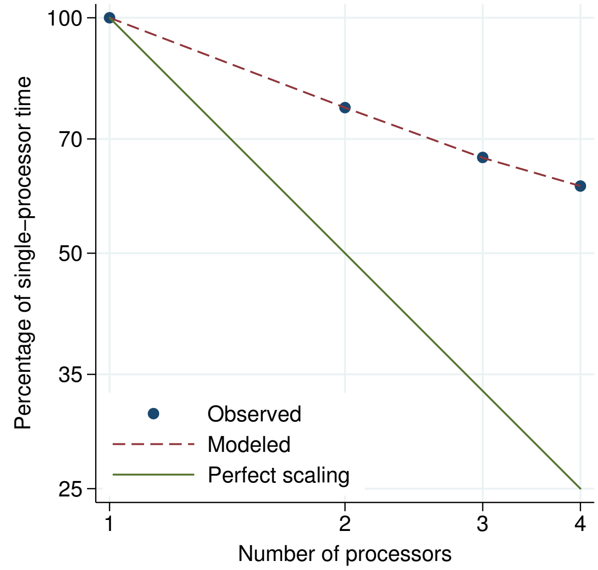

Figure A.264. `ttest1` performance plot.

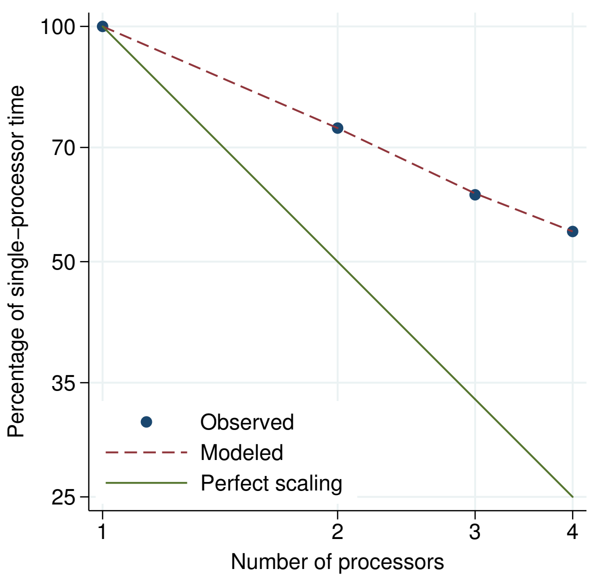

Figure A.265. `ttest2` performance plot.

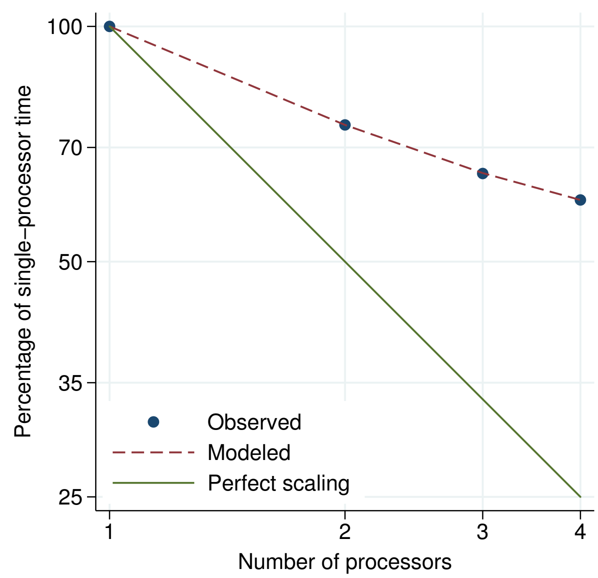

Figure A.266. `ttest, by()` performance plot.

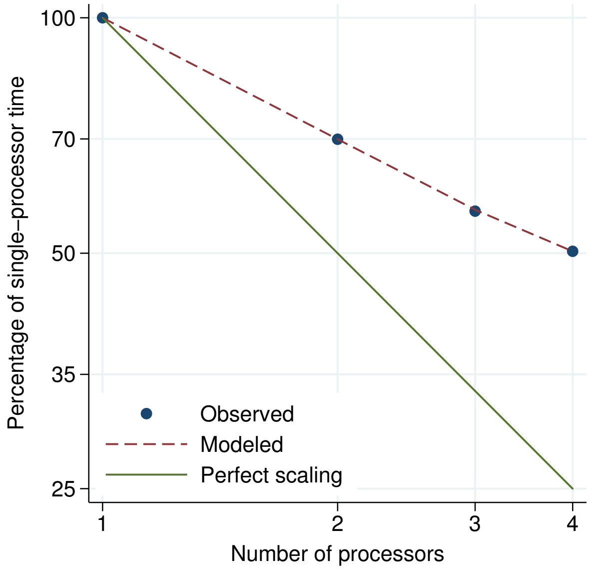

Figure A.267. `twoway fpfit` performance plot.

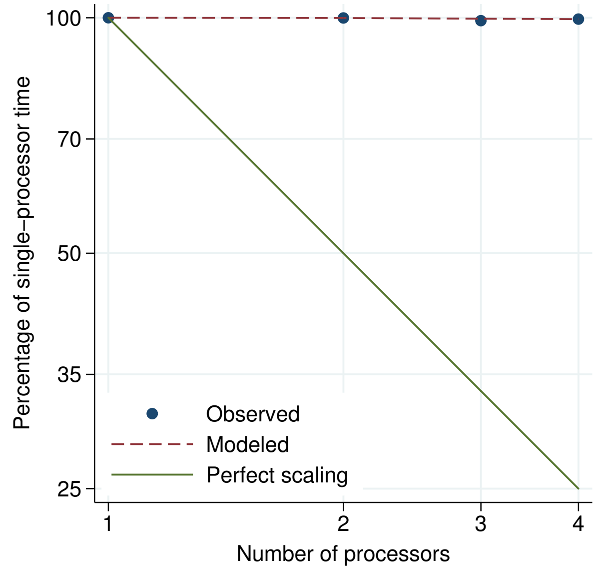

Figure A.268. `twoway lfitci` performance plot.

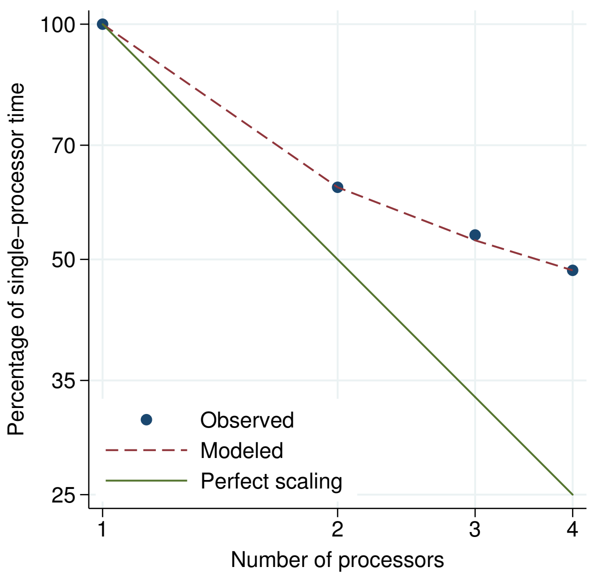

Figure A.269. `twoway mband` performance plot.

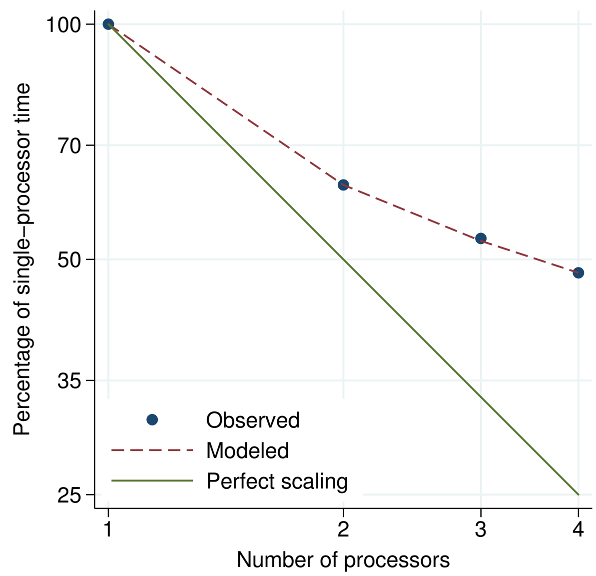

Figure A.270. `twoway mspline` performance plot.

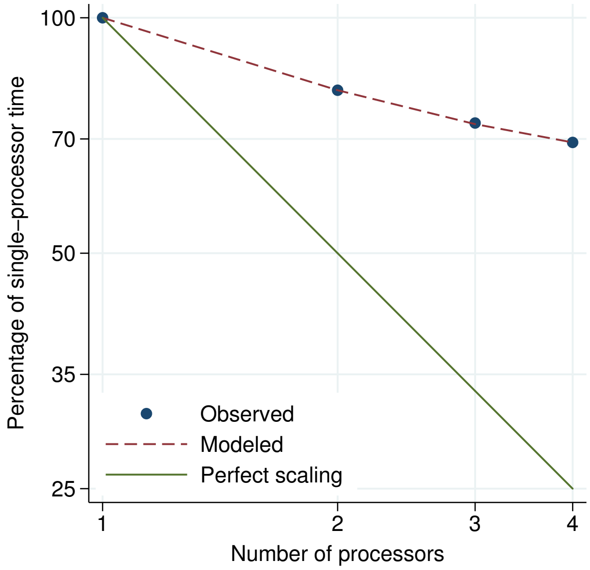

Figure A.271. `var` performance plot.

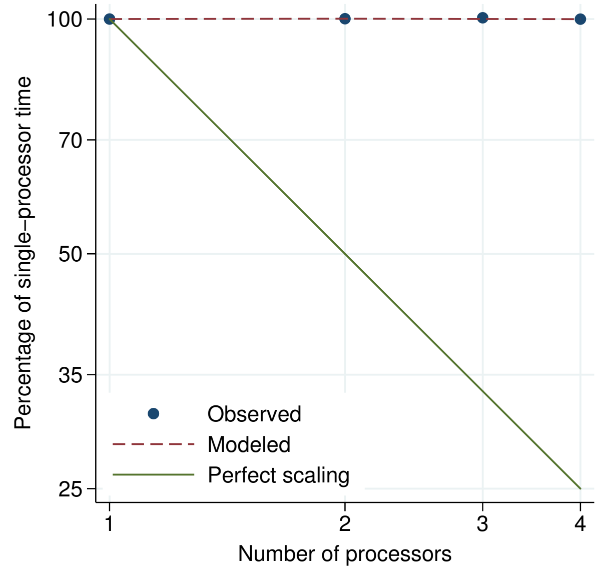

Figure A.272. `vargranger` performance plot.

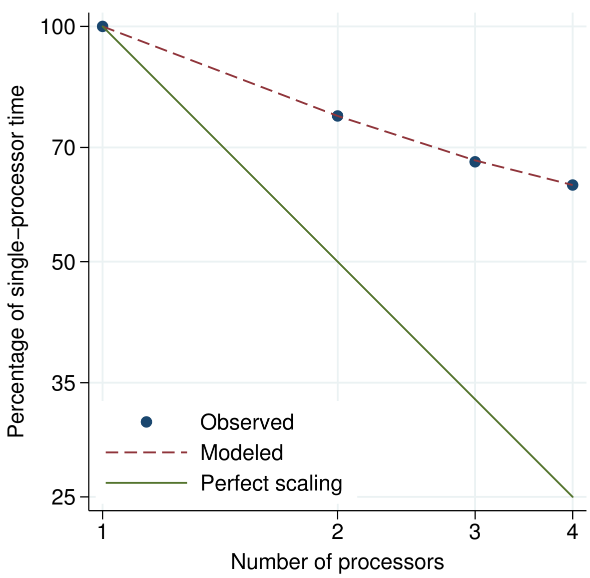

Figure A.273. `varlmar` performance plot.

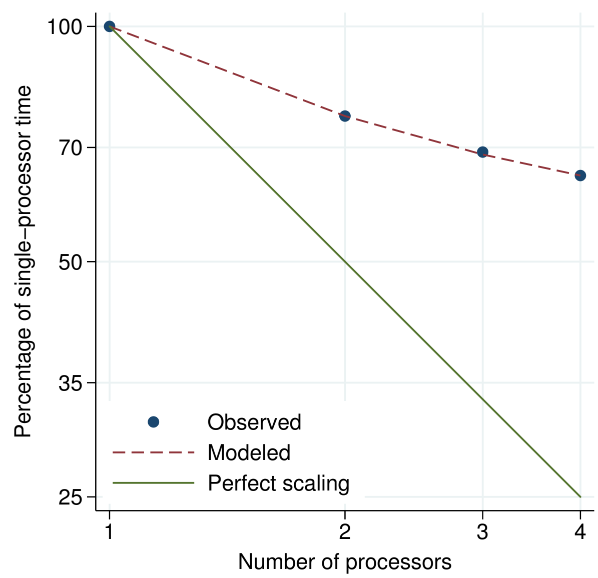

Figure A.274. `varnorm` performance plot.

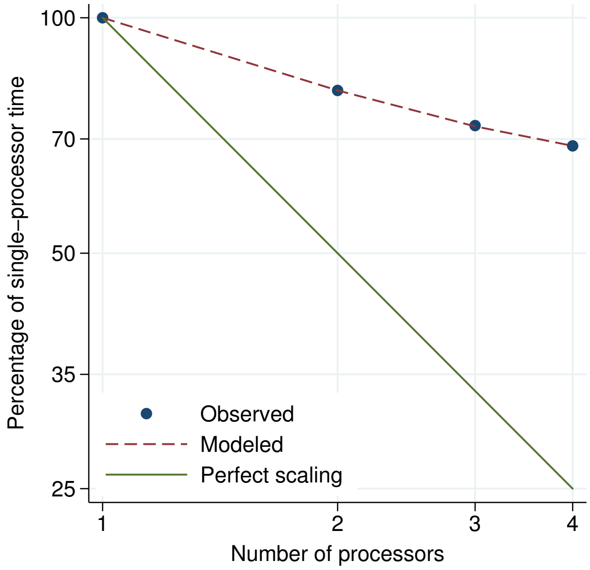

Figure A.275. `varsoc` performance plot.

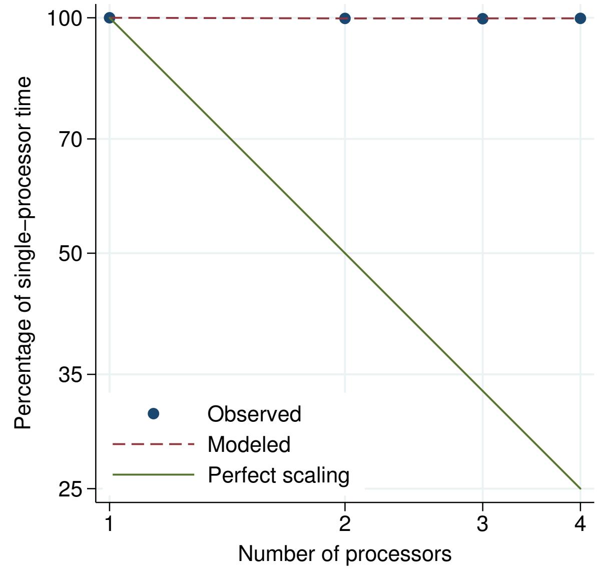

Figure A.276. `varstable` performance plot.

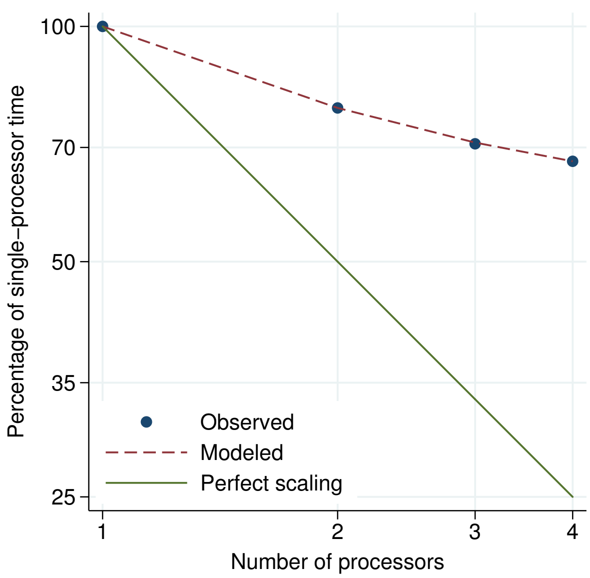

Figure A.277. `vec` performance plot.

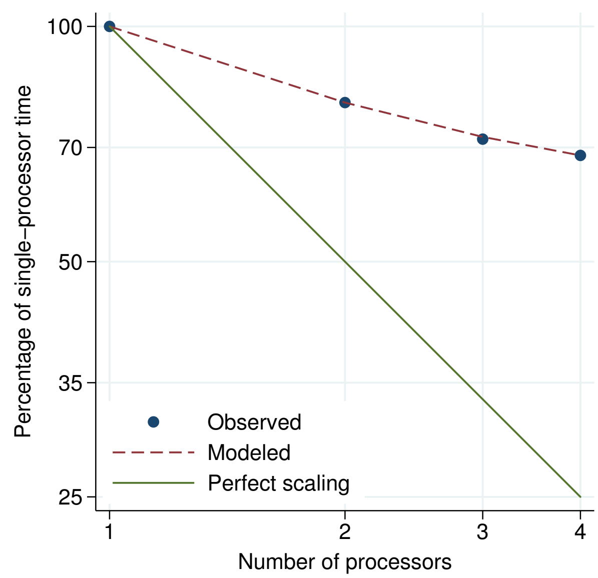

Figure A.278. `veclmar` performance plot.

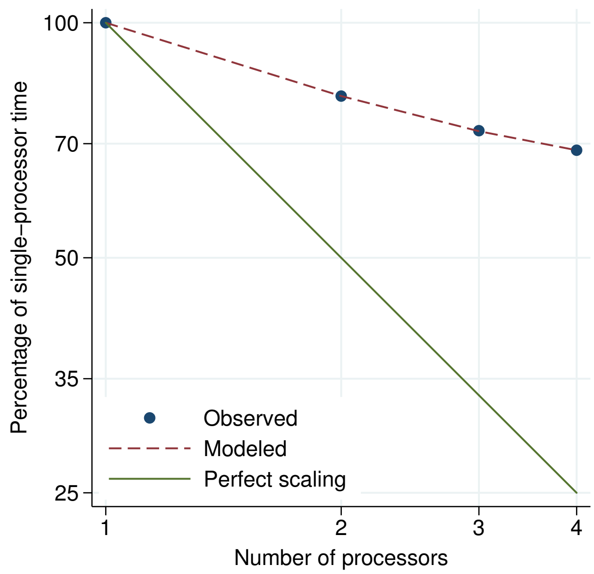

Figure A.279. `vecnorm` performance plot.

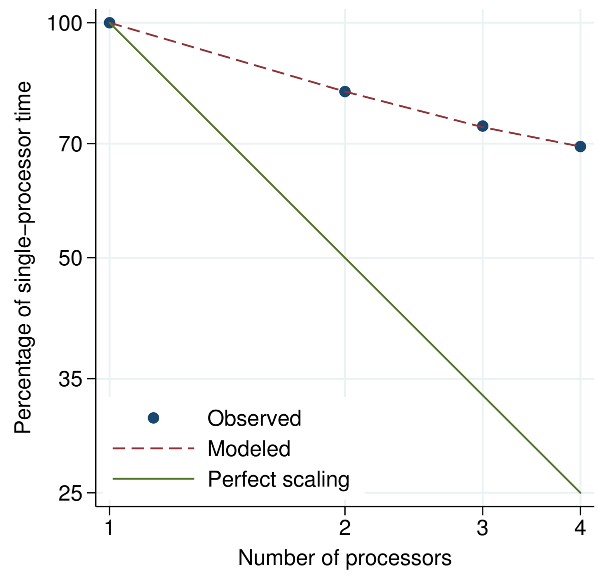

Figure A.280. `vecrank` performance plot.

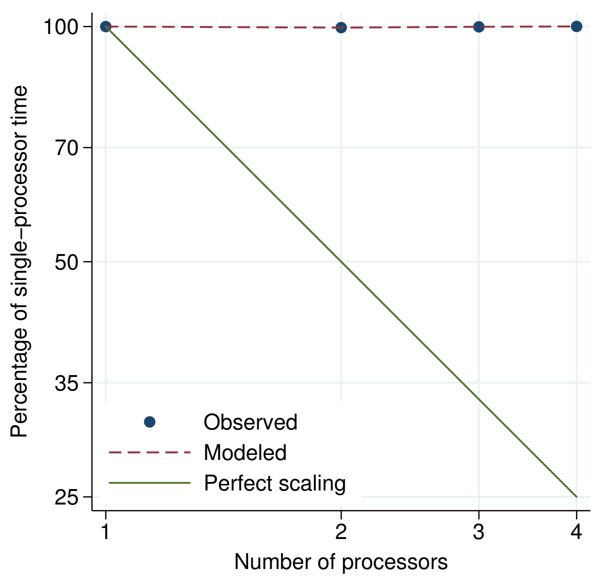

Figure A.281. `vecstable` performance plot.

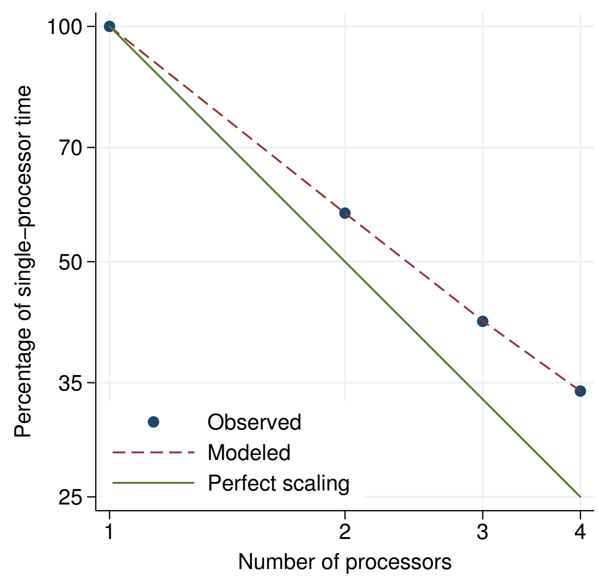

Figure A.282. `vwls` performance plot.

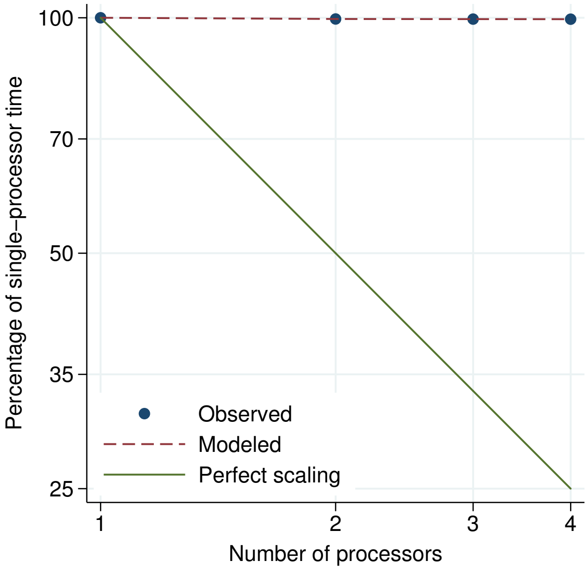

Figure A.283. `wntestb` performance plot.

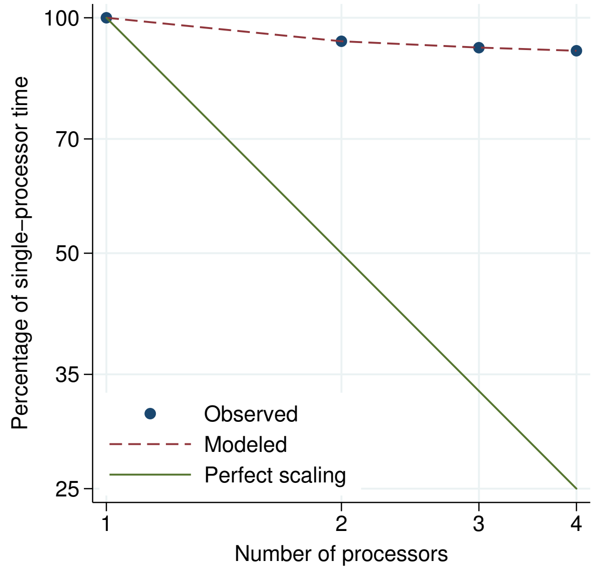

Figure A.284. `wntestq` performance plot.

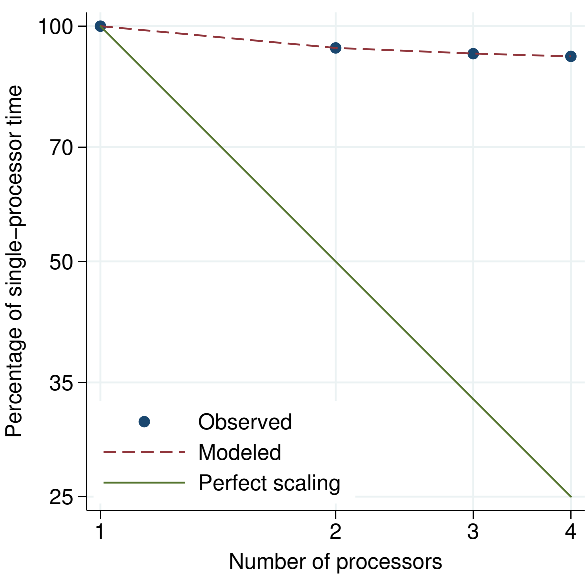

Figure A.285. `xcorr` performance plot.

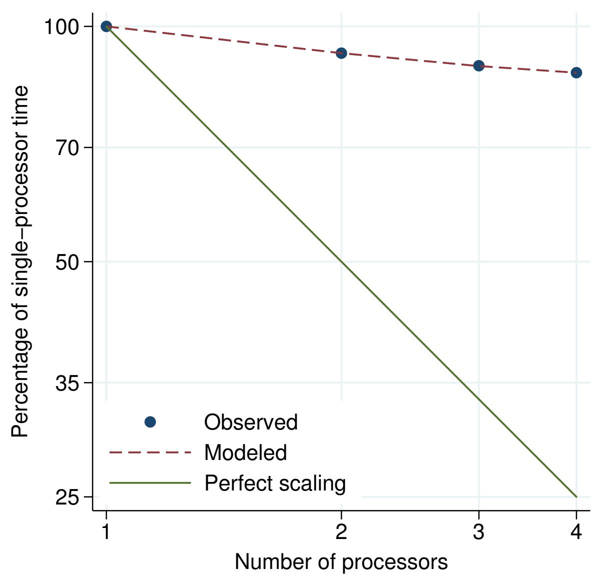

Figure A.286. `xtabond` performance plot.

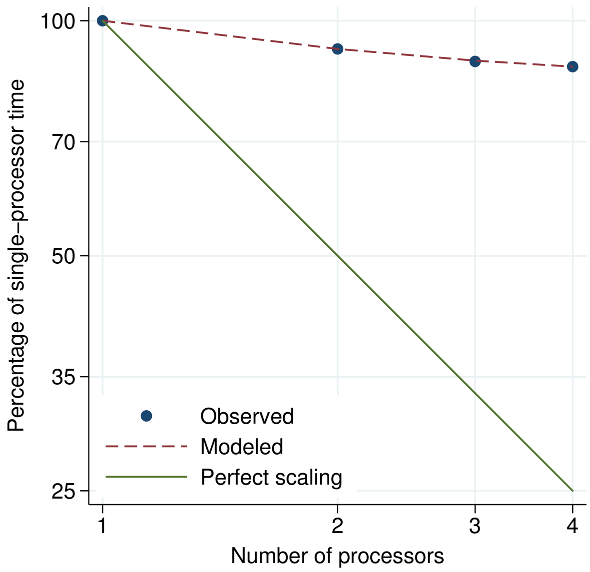

Figure A.287. `xtabond, twostep` performance plot.

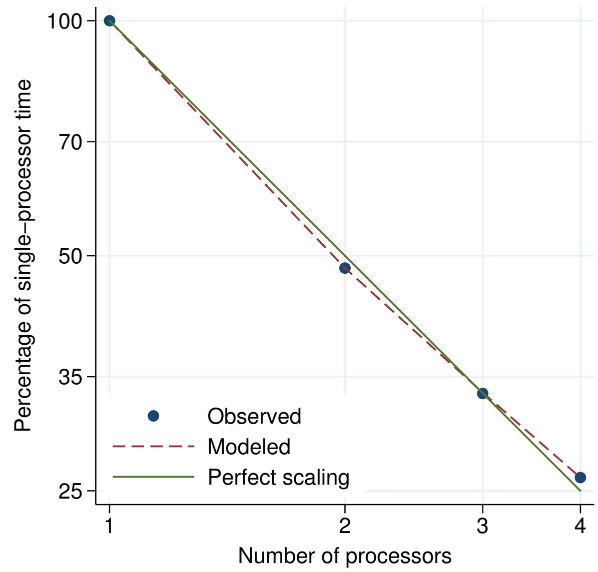

Figure A.288. `xtcloglog, re` performance plot.

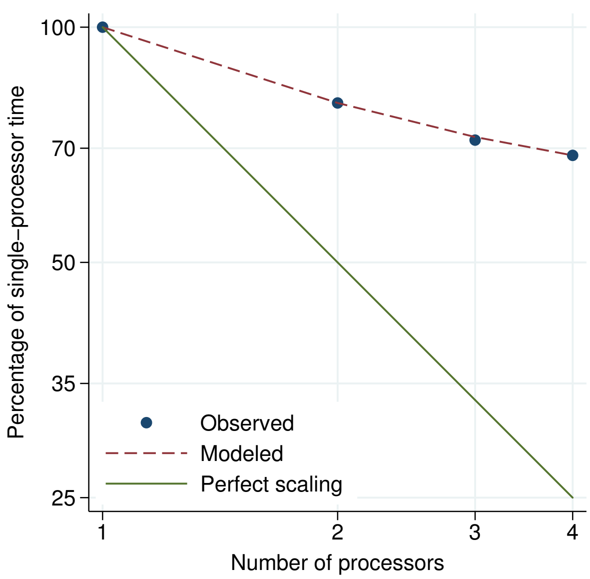

Figure A.289. `xtdata, be` performance plot.

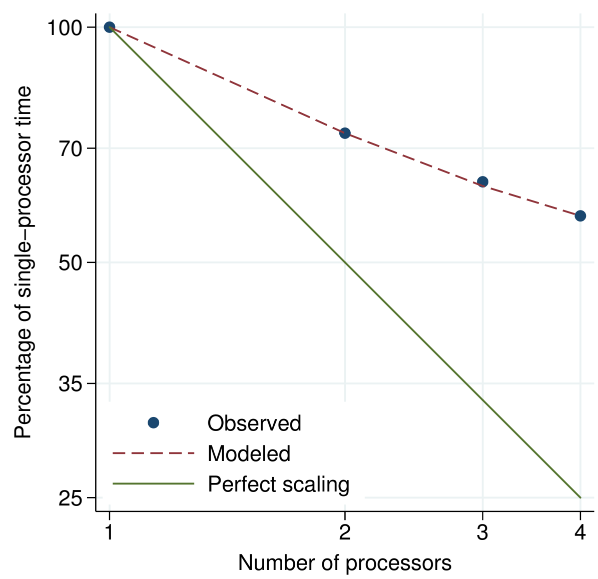

Figure A.290. `xtdata, fe` performance plot.

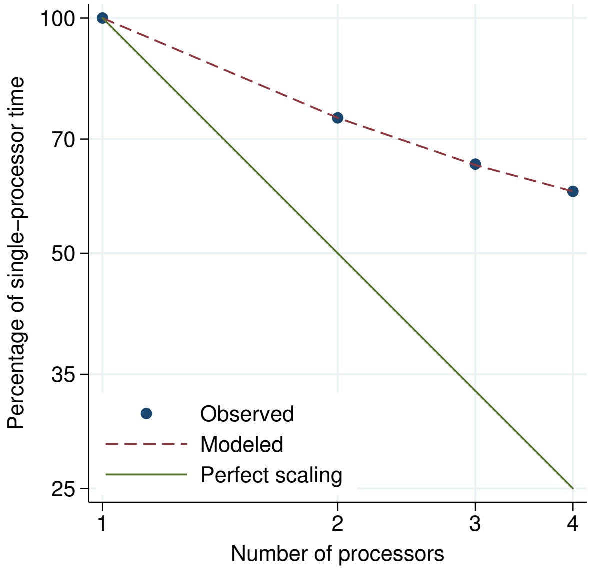

Figure A.291. `xtdata, re` performance plot.

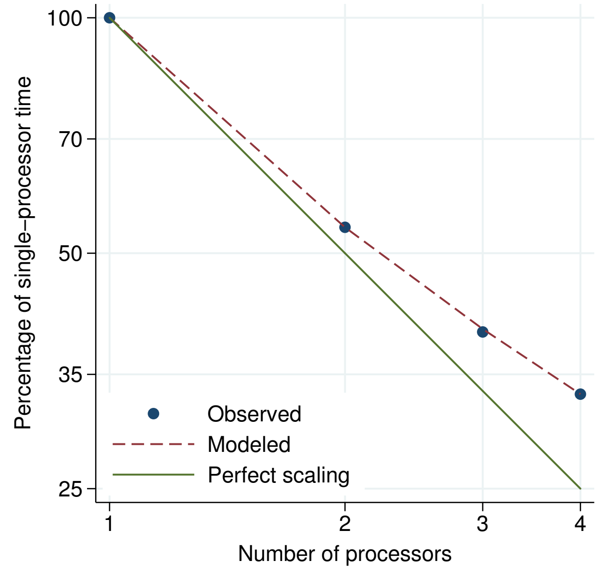

Figure A.292. `xtfrontier` performance plot.

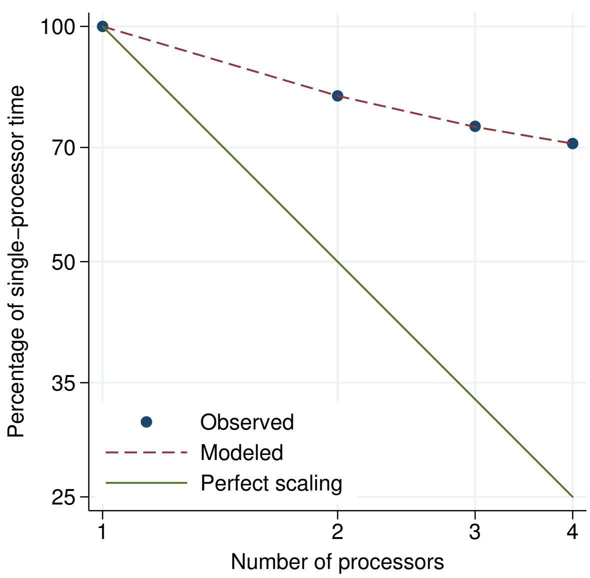

Figure A.293. `xtgee, family(gaussian) corr(ar2)` performance plot.

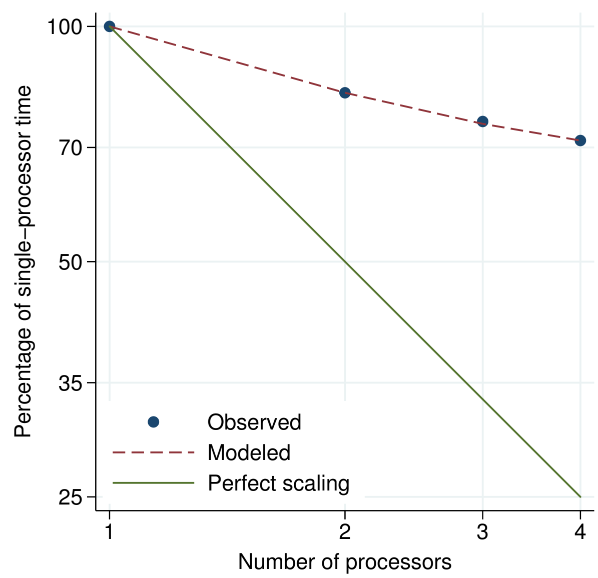

Figure A.294. `xtgee, fam(gauss) corr(unstruct)` performance plot.

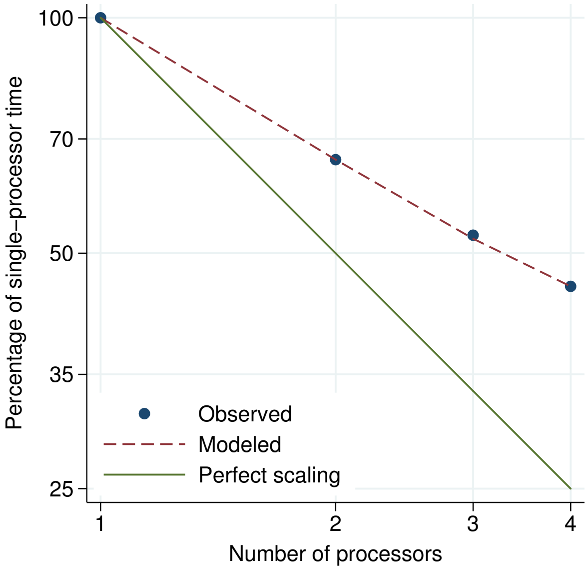

Figure A.295. `xtcloglog, pa` performance plot.

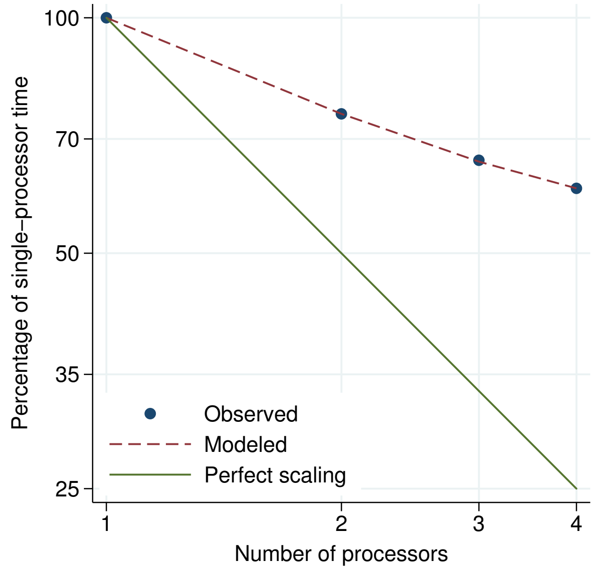

Figure A.296. `xtlogit, pa` performance plot.

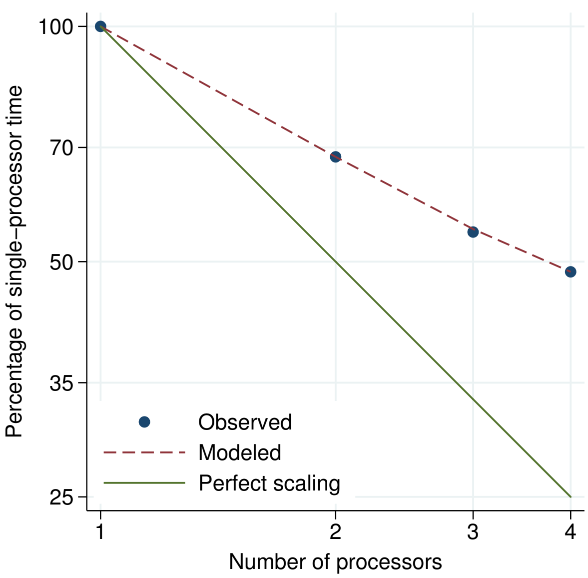

Figure A.297. `xtnbreg, pa` performance plot.

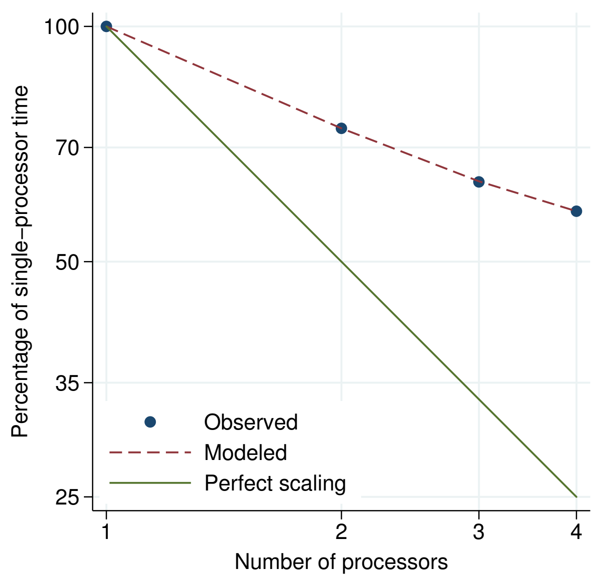

Figure A.298. `xtpoisson, pa` performance plot.

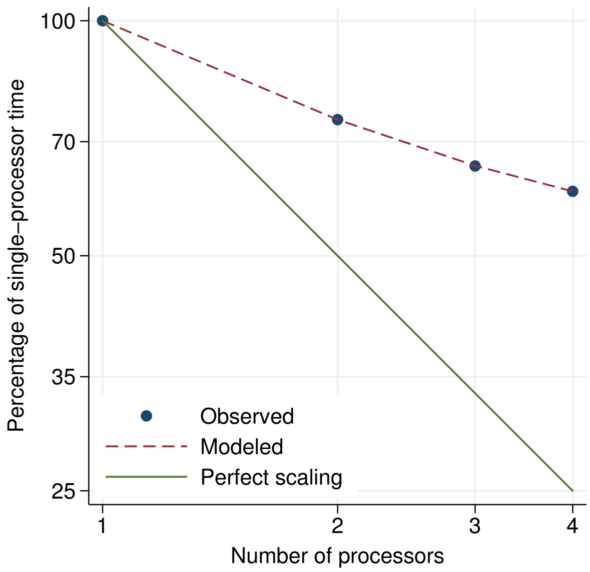

Figure A.299. `xtprobit, pa` performance plot.

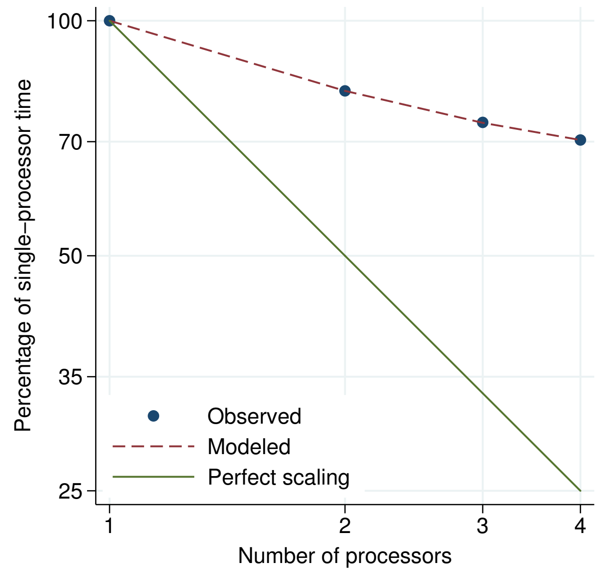

Figure A.300. `xtreg, pa` performance plot.

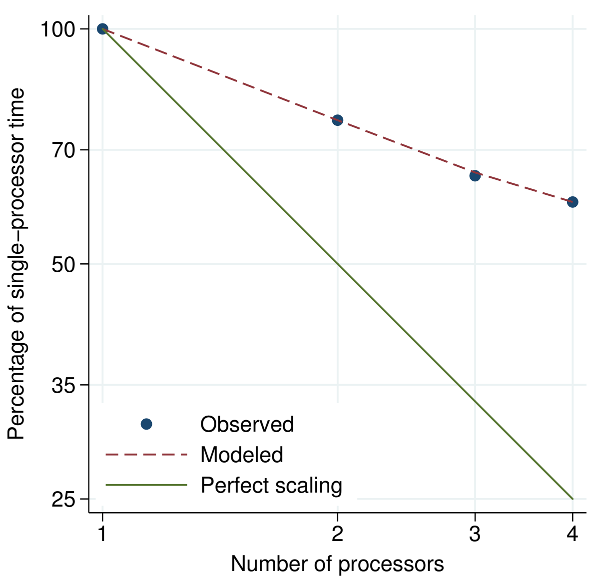

Figure A.301. `xtgls` performance plot.

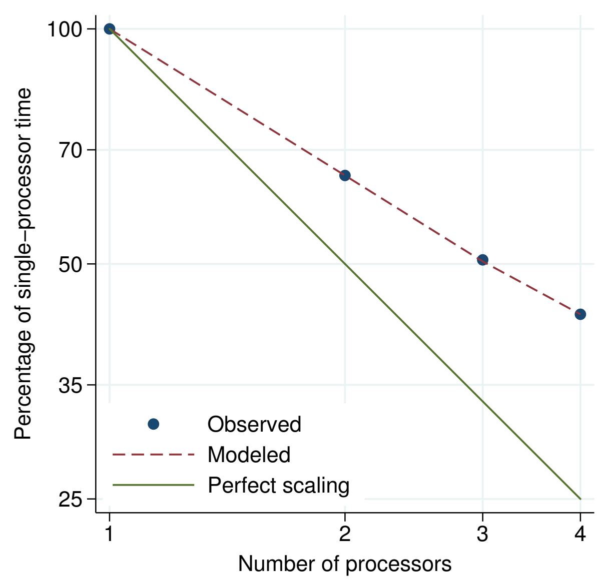

Figure A.302. `xthtaylor` performance plot.

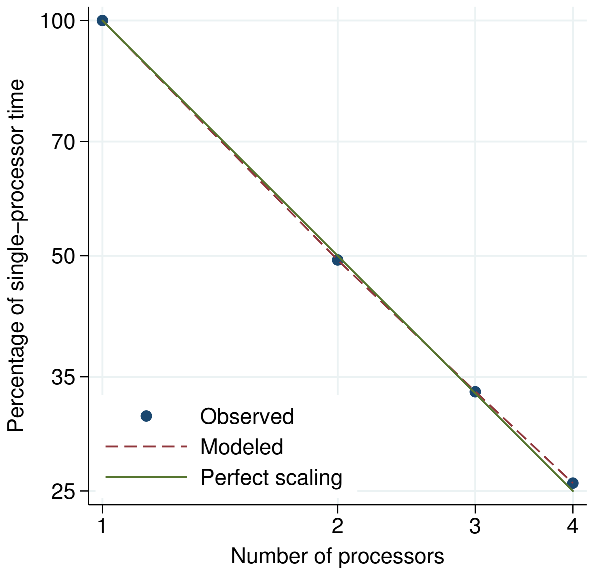

Figure A.303. `xtintreg` performance plot.

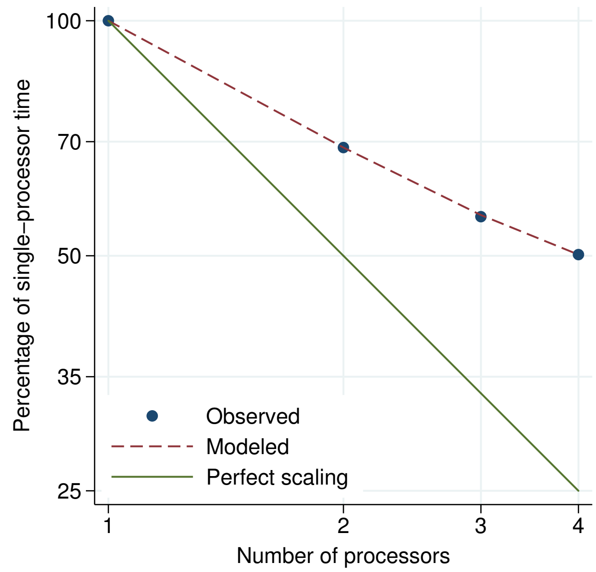

Figure A.304. `xtivreg, be` performance plot.

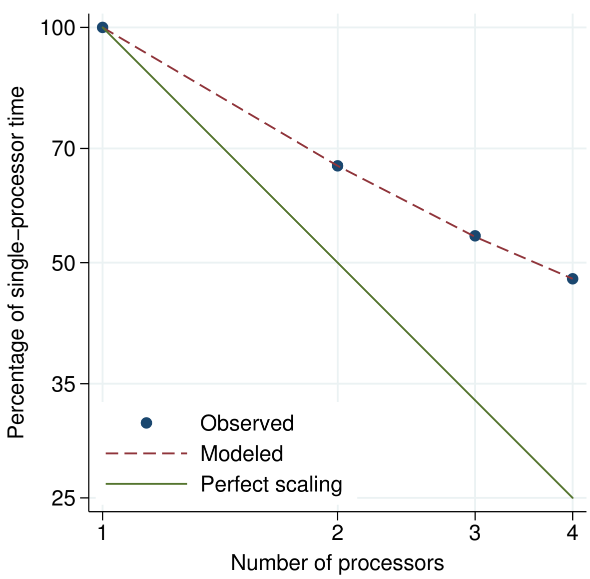

Figure A.305. `xtivreg, re` performance plot.

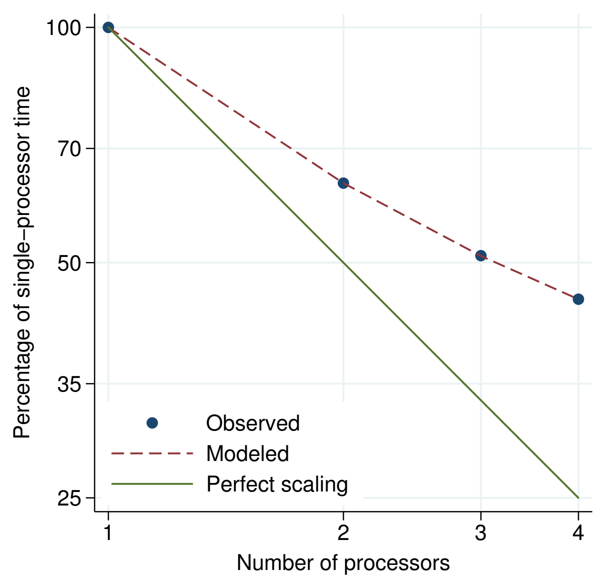

Figure A.306. `xtlogit, fe` performance plot.

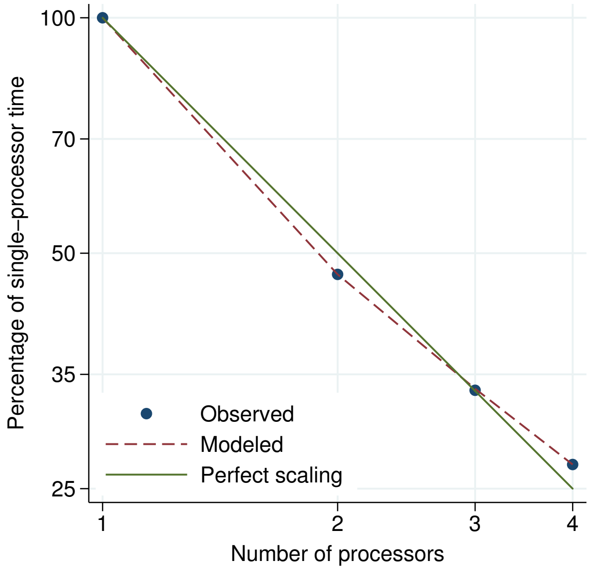

Figure A.307. `xtlogit, re` performance plot.

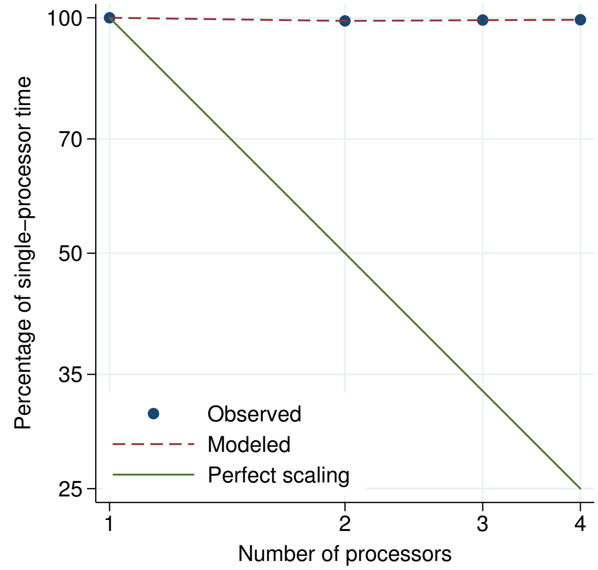

Figure A.308. `xtmixed` performance plot.

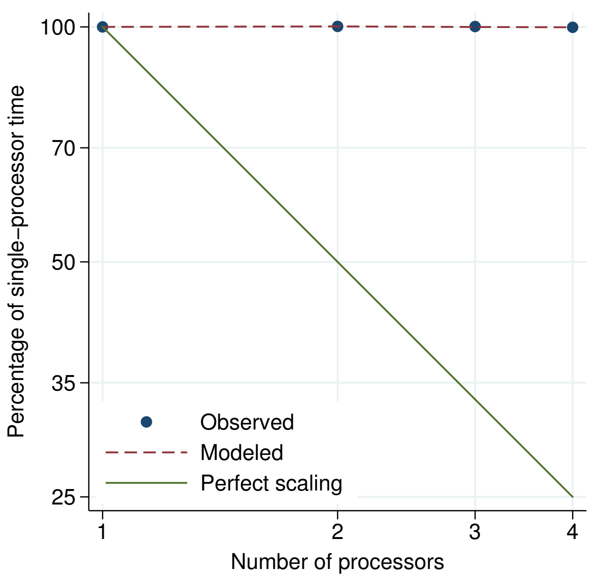

Figure A.309. `xtmixed (crossed effects)` performance plot.

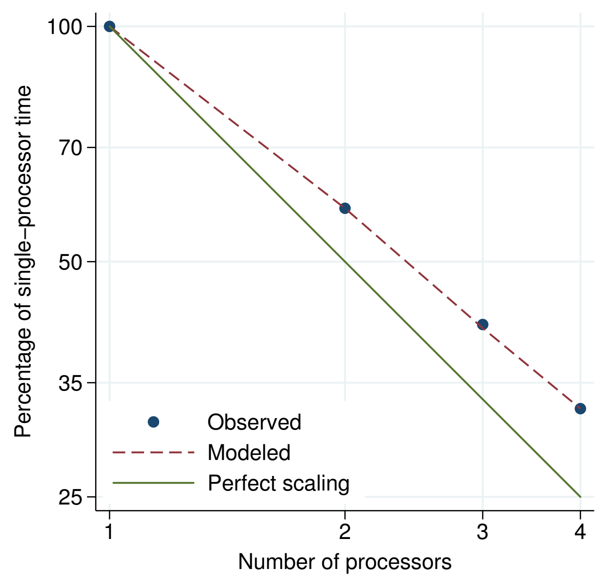

Figure A.310. `xtnbreg, fe` performance plot.

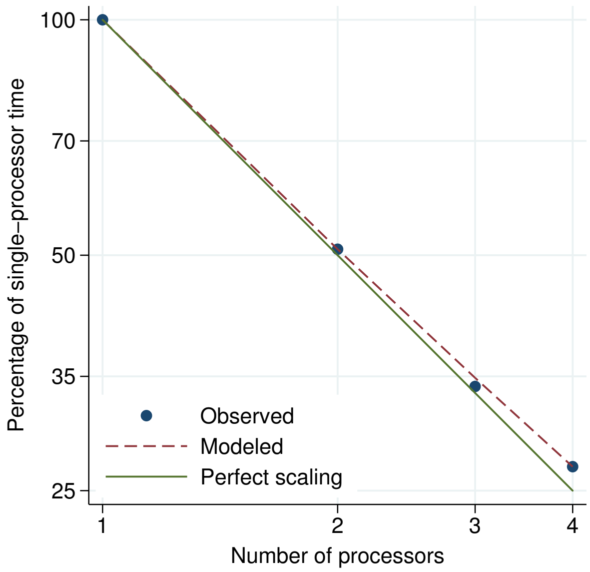

Figure A.311. `xtnbreg, re` performance plot.

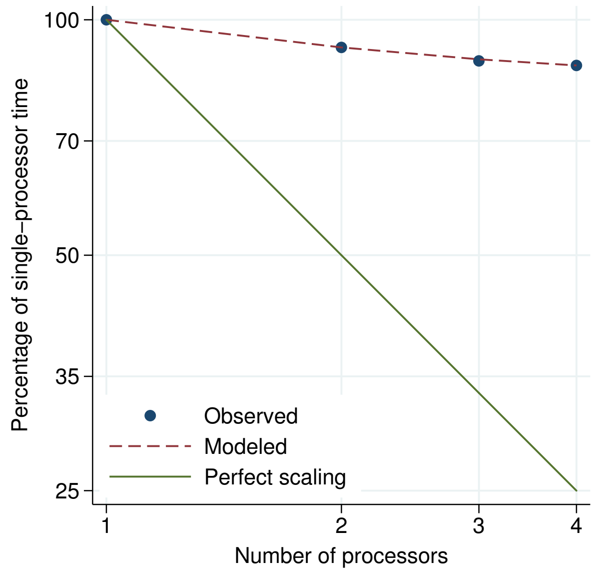

Figure A.312. `xtpcse` performance plot.

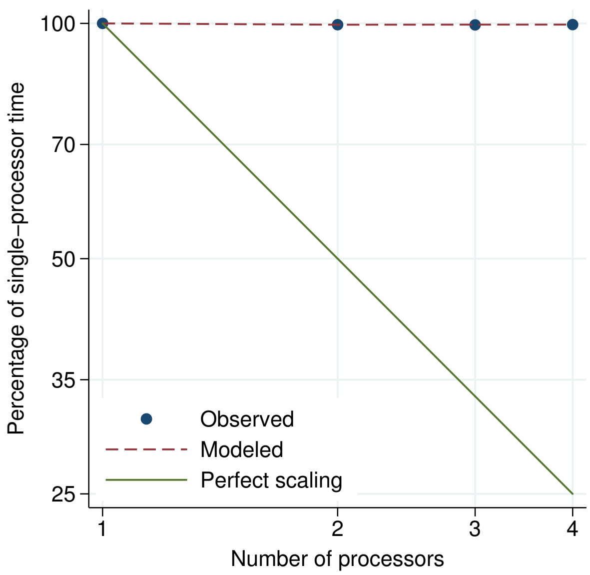

Figure A.313. `xtpcse, corr(ar1)` performance plot.

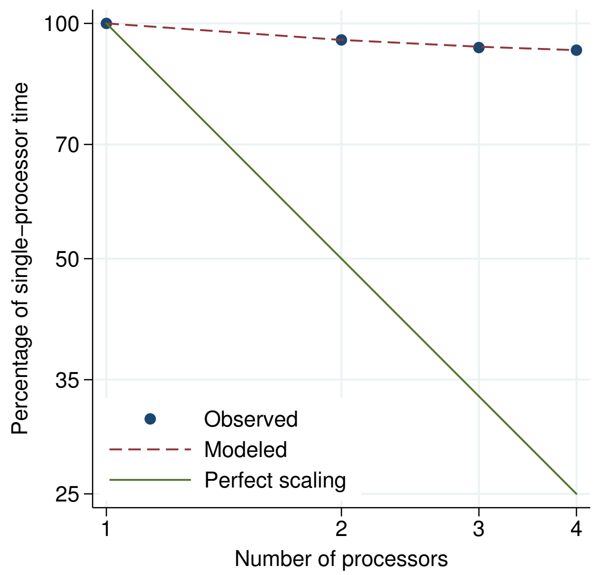

Figure A.314. `xtpcse, corr(psar1)` performance plot.

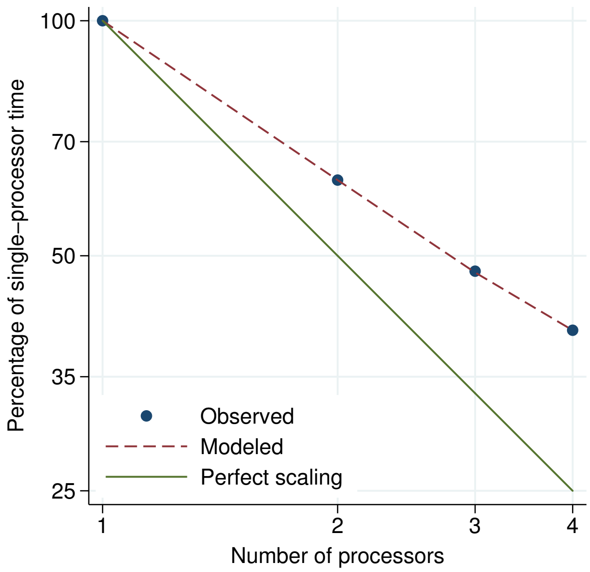

Figure A.315. `xtpoisson, fe` performance plot.

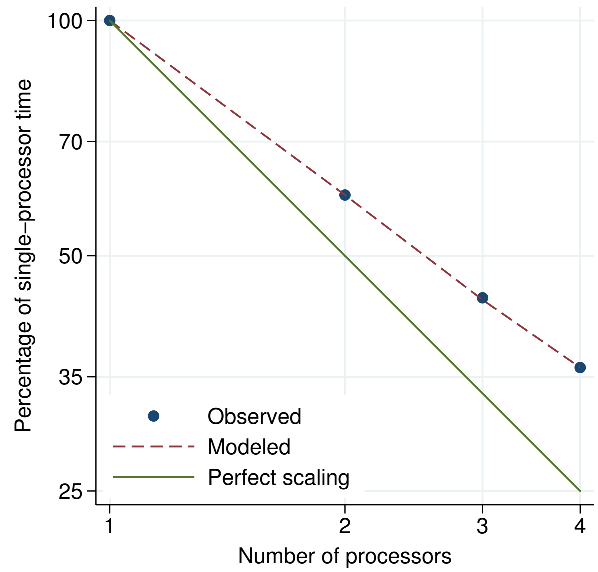

Figure A.316. `xtpoisson, re` performance plot.

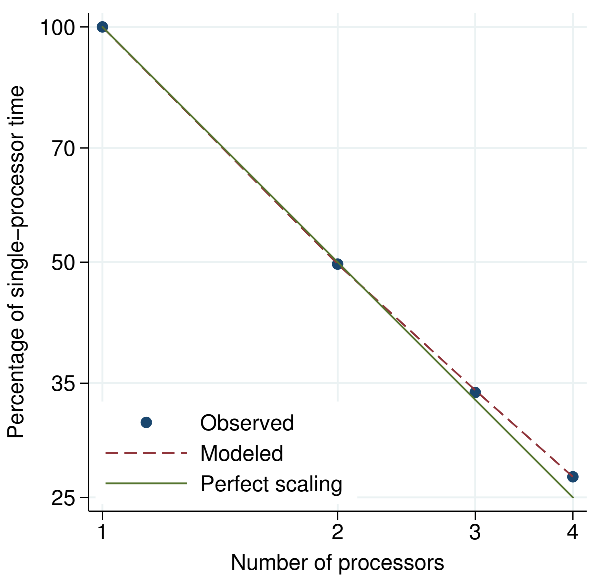

Figure A.317. `xtprobit, re` performance plot.

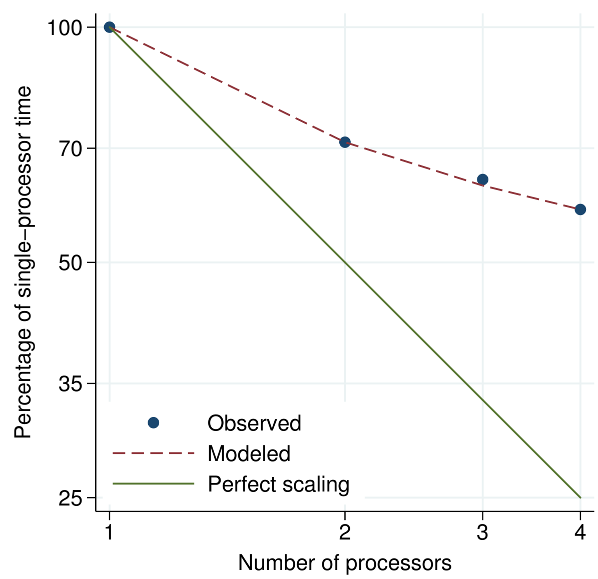

Figure A.318. `xtrc` performance plot.

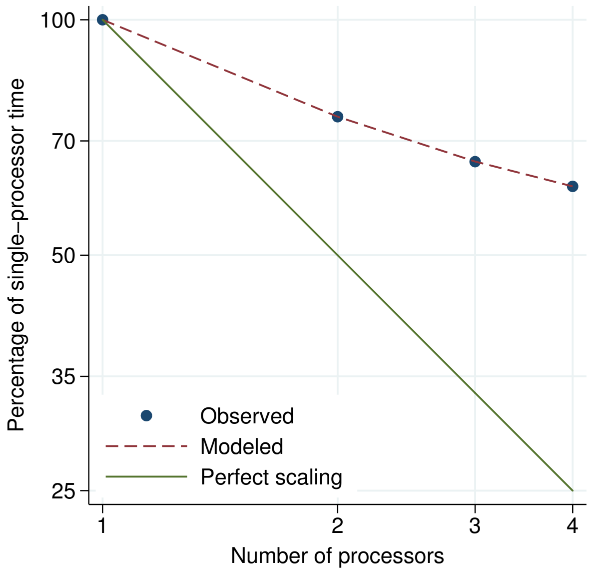

Figure A.319. `xtreg, be` performance plot.

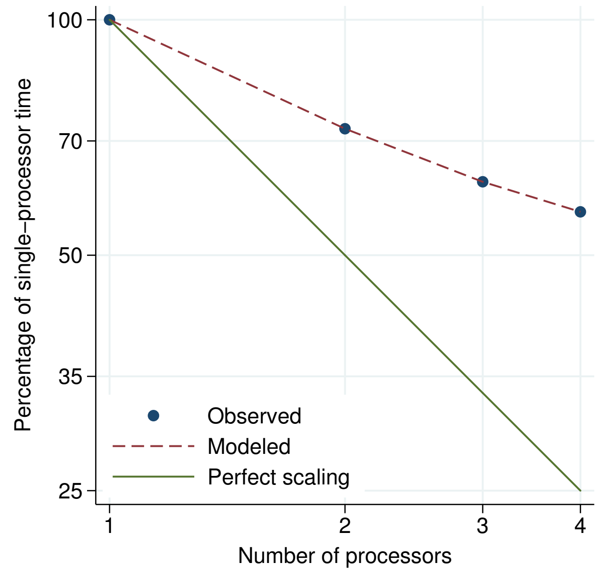

Figure A.320. `xtreg, fe` performance plot.

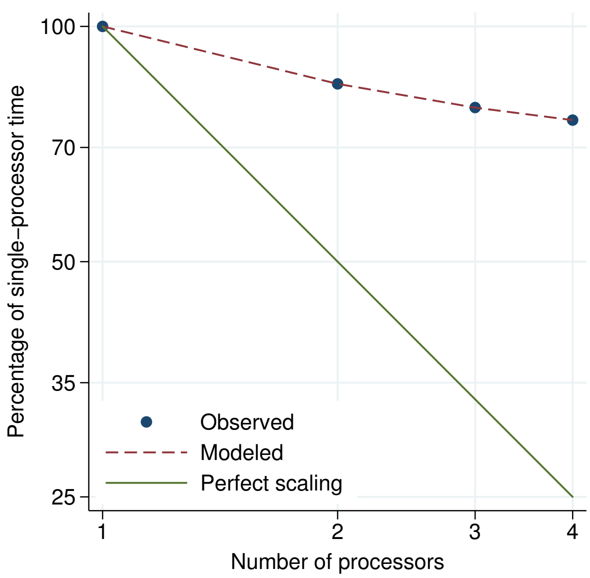

Figure A.321. `xtreg, mle` performance plot.

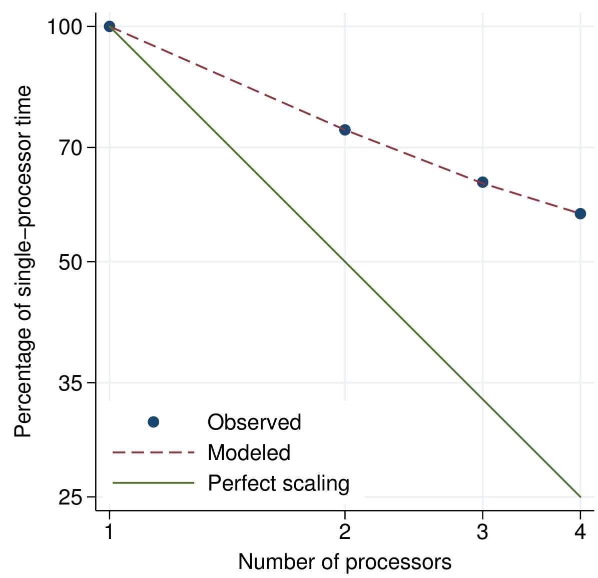

Figure A.322. `xtreg, re` performance plot.

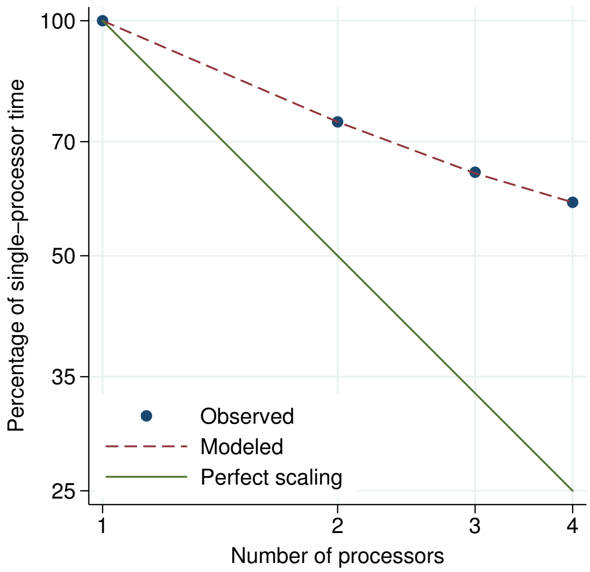

Figure A.323. `xtregar, fe` performance plot.

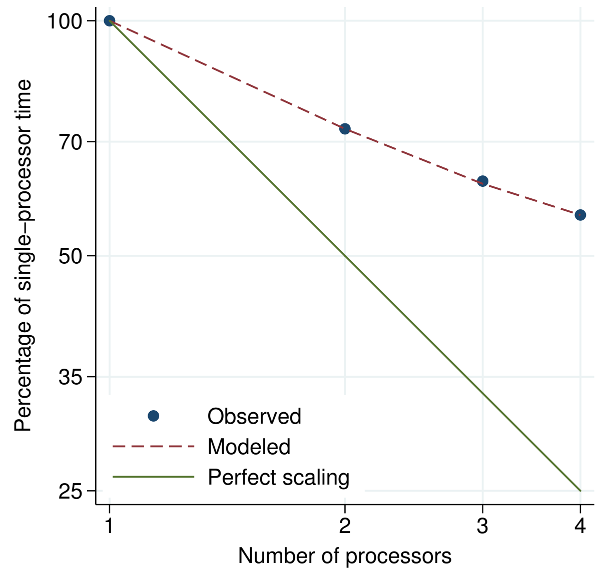

Figure A.324. `xtregar, re` performance plot.

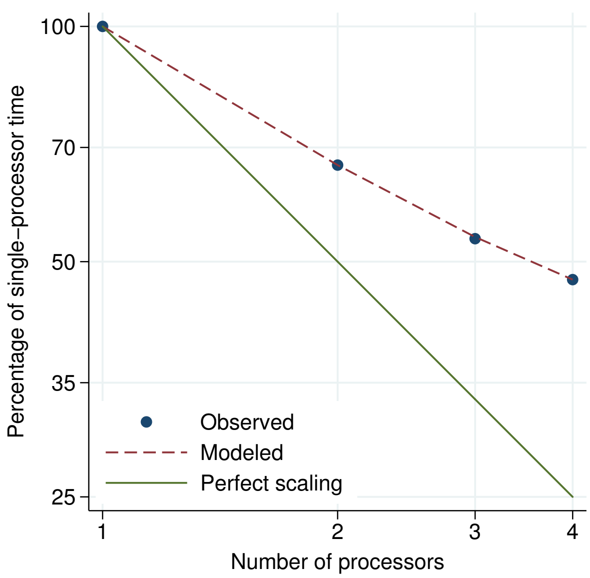

Figure A.325. `xtsum` performance plot.

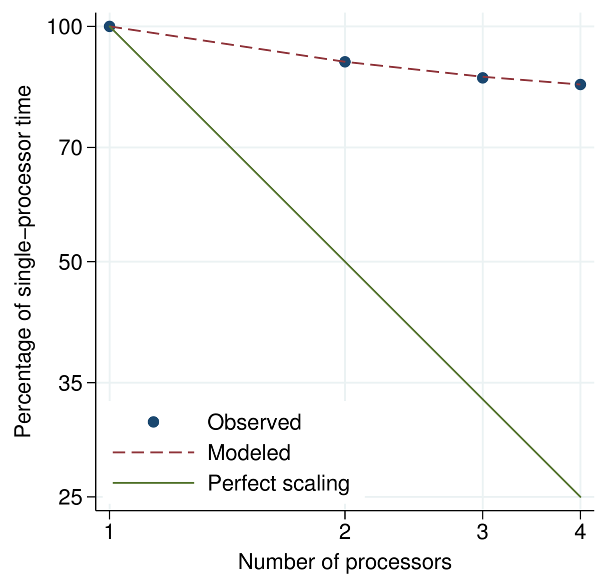

Figure A.326. `xttab` performance plot.

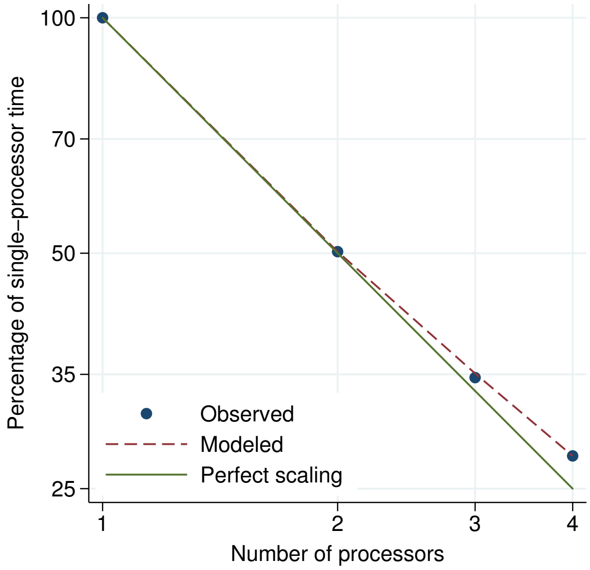

Figure A.327. `xttobit` performance plot.

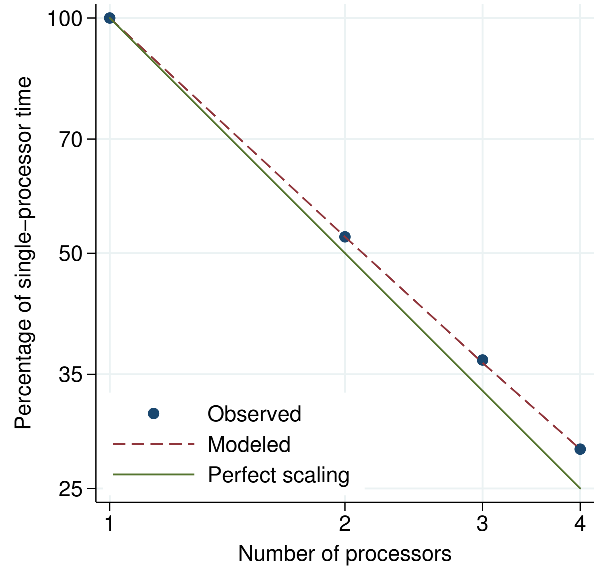

Figure A.328. `zinb` performance plot.

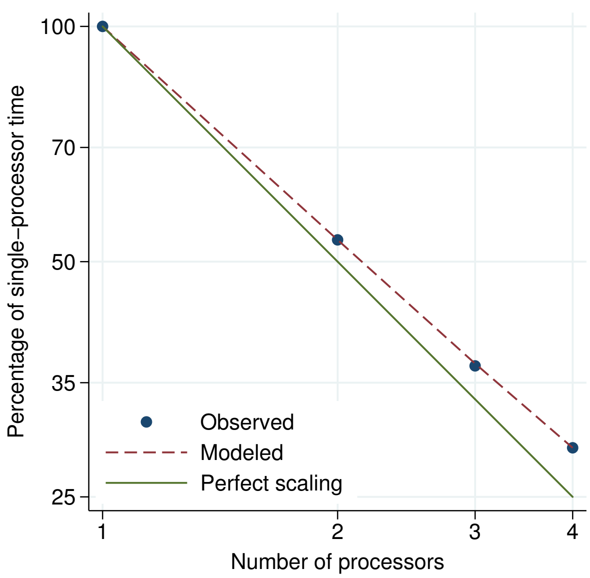

Figure A.329. `zip` performance plot.

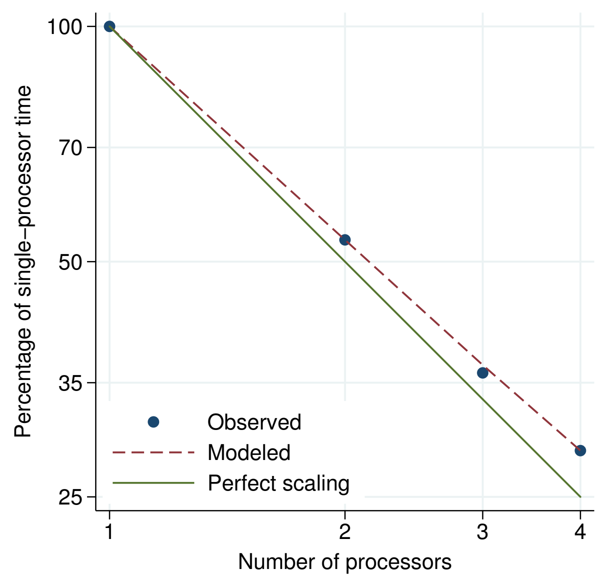

Figure A.330. `ztnb` performance plot.

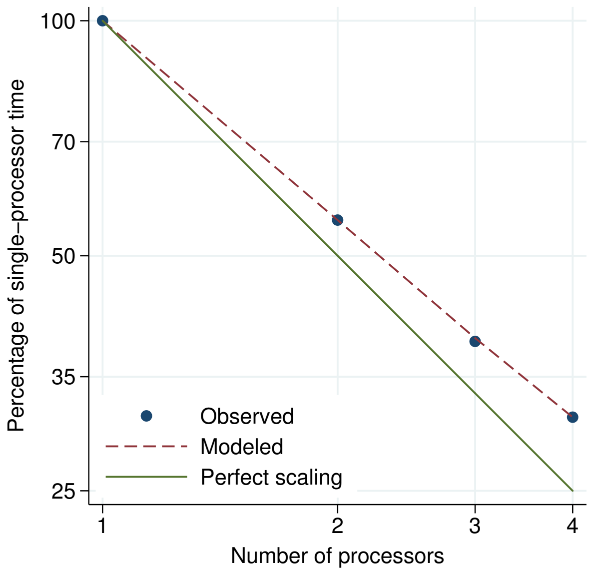

Figure A.331. `ztp` performance plot.

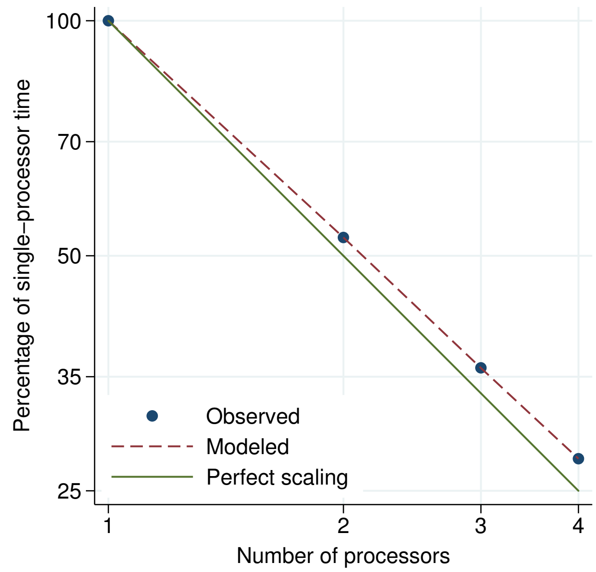

Figure A.332. `_predict, xb` performance plot.

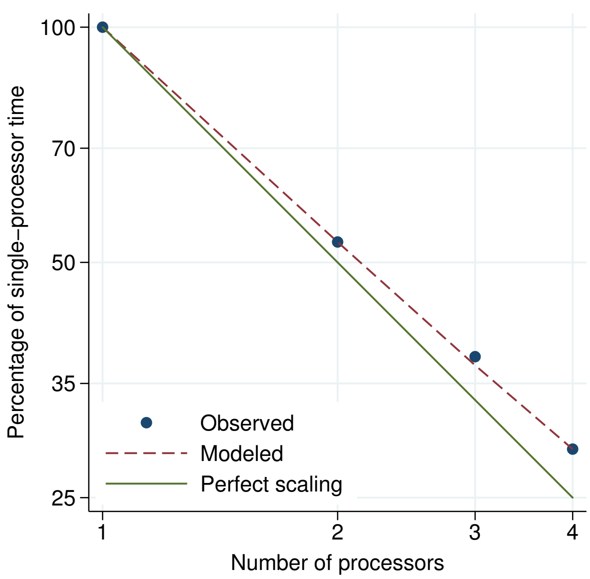

Figure A.333. `_rmcoll` performance plot.

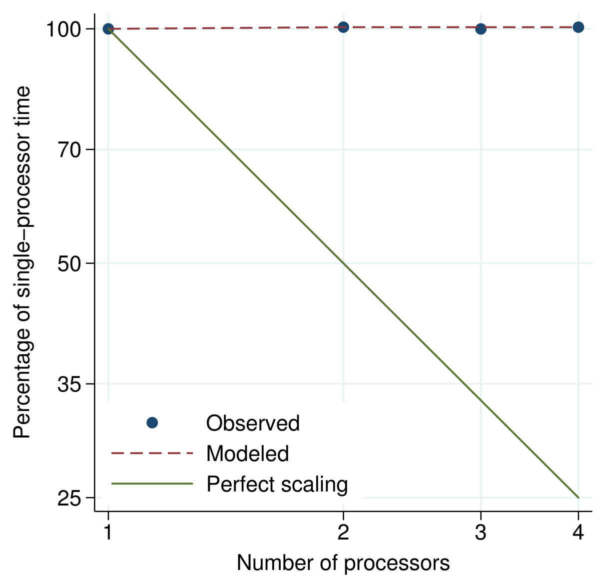

Figure A.334. `_robust` performance plot.

B Performance assessment graphs for 16 processors

Actual and projected performance graphs of all 332 commands are presented below. Of these commands, 115 have been timed on a 16-processor computer and the remaining graphs show extrapolated performance based on measurements from a 4-processor computer and equation 1. The close agreement between the 4-processor and 16-processor computers when estimating the 115 equations common to both suggest that these extrapolations will be reliable.

In all cases, observed timings are shown, and so it is easy to tell the extrapolated graphs from the rest.

As with the performance table, a few of the results for `cluster` commands produced overly optimistic projections for 8- and 16-processor performance. The graphs will be updated when the cluster commands have been tested on 8- and 16-processor computers.

These graphs take the same form as those from appendix A.

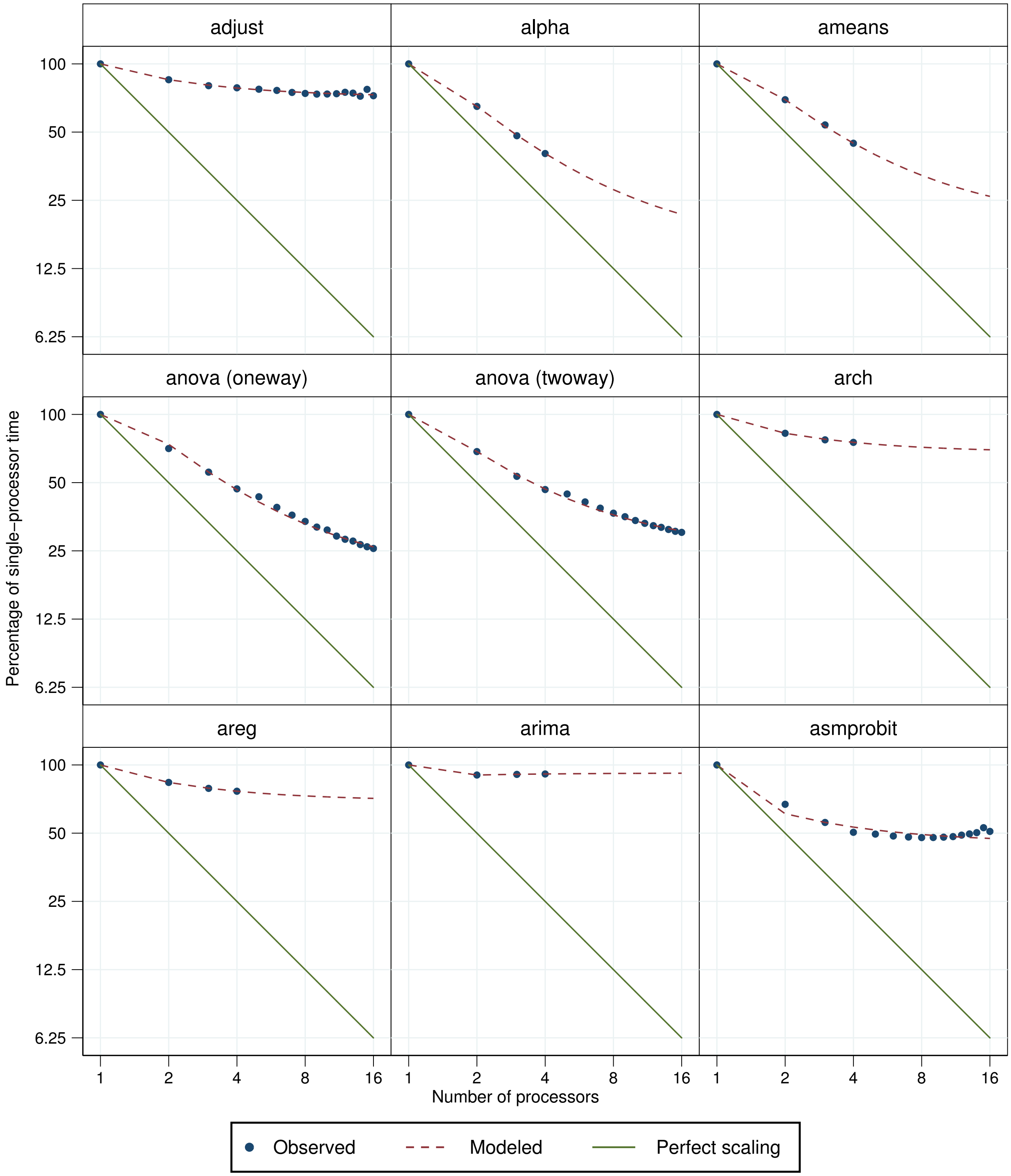

Figure B.1. **Parallelization performance plots.**

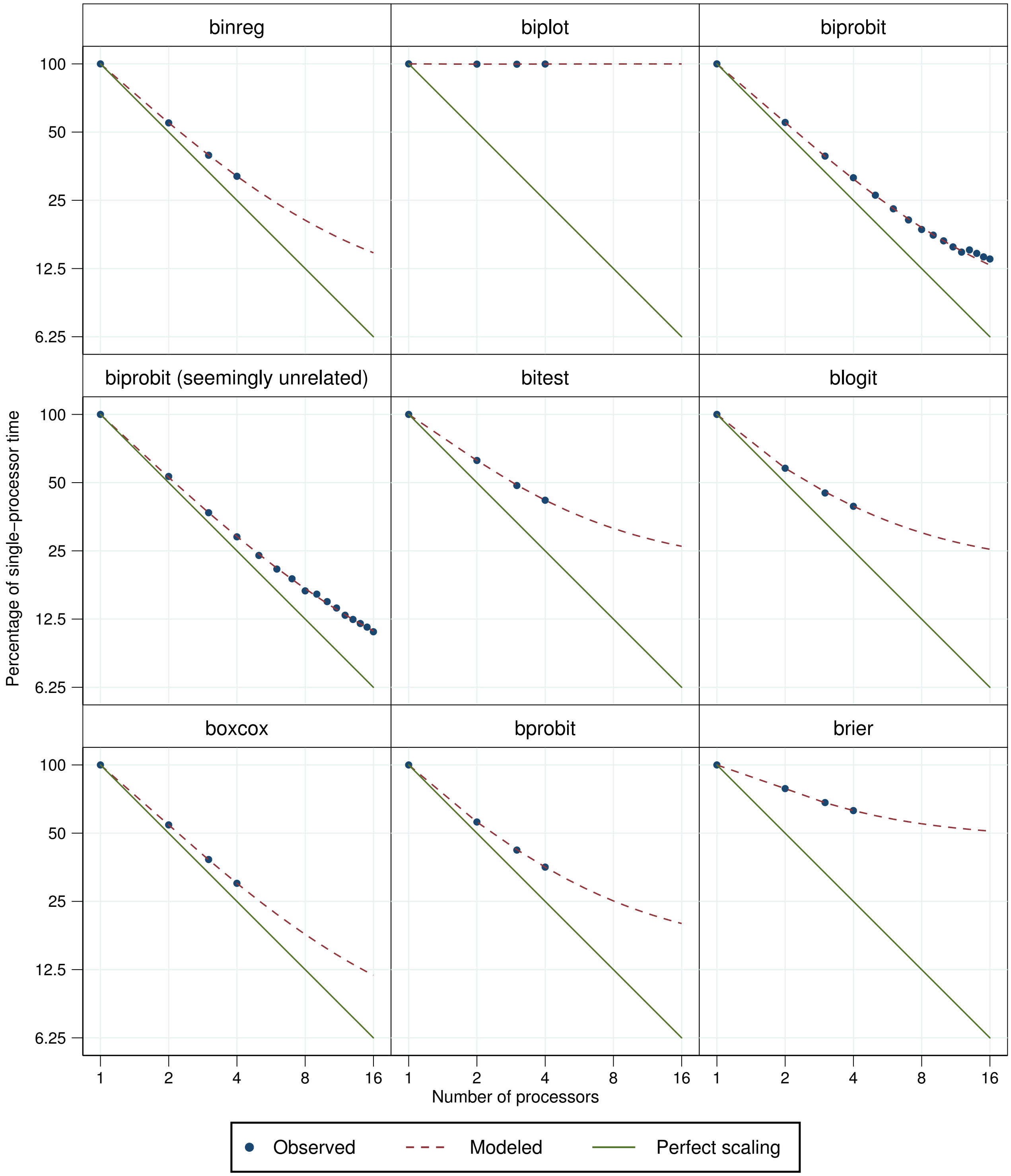

Figure B.2. **Parallelization performance plots.**

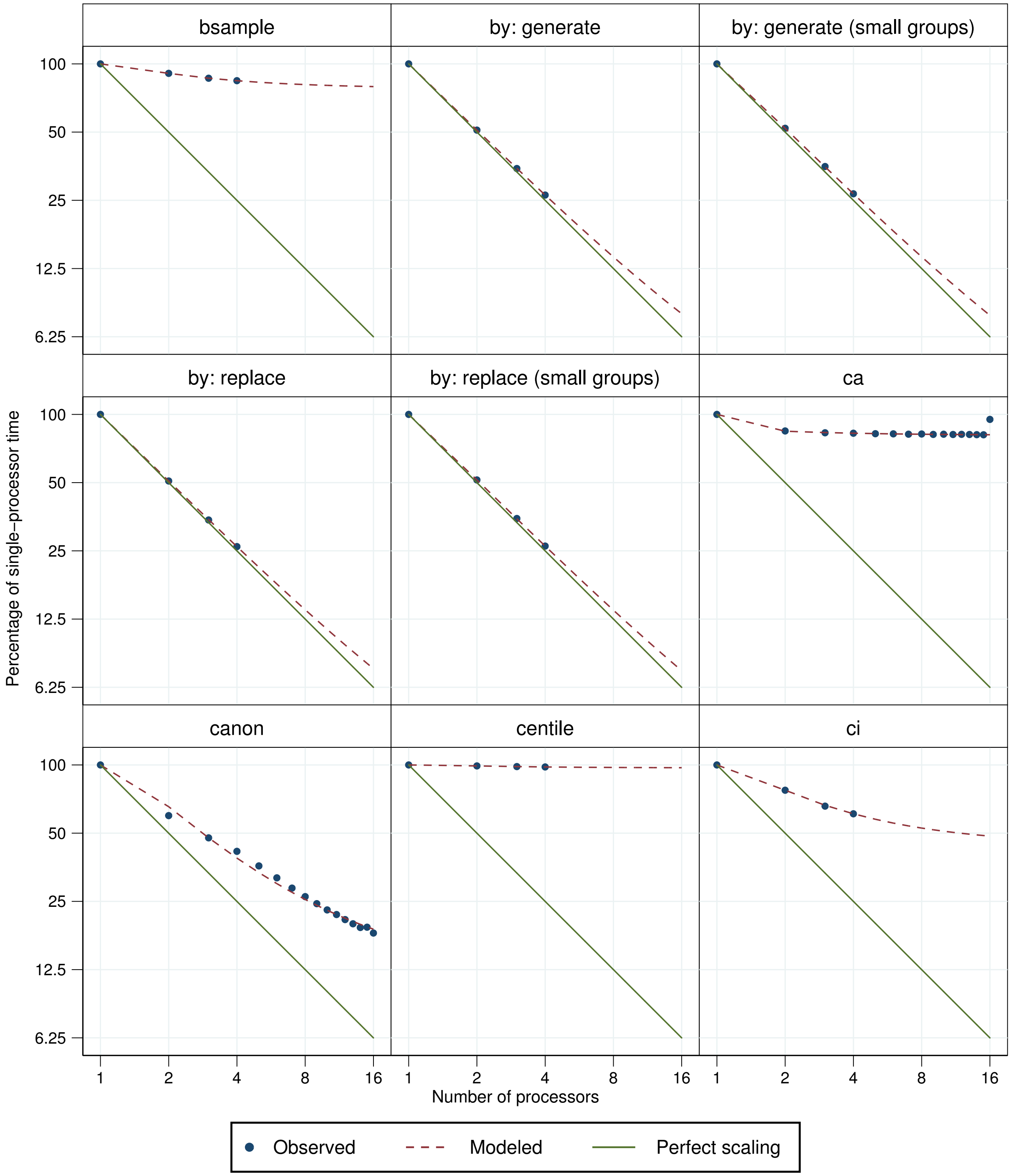

Figure B.3. **Parallelization performance plots.**

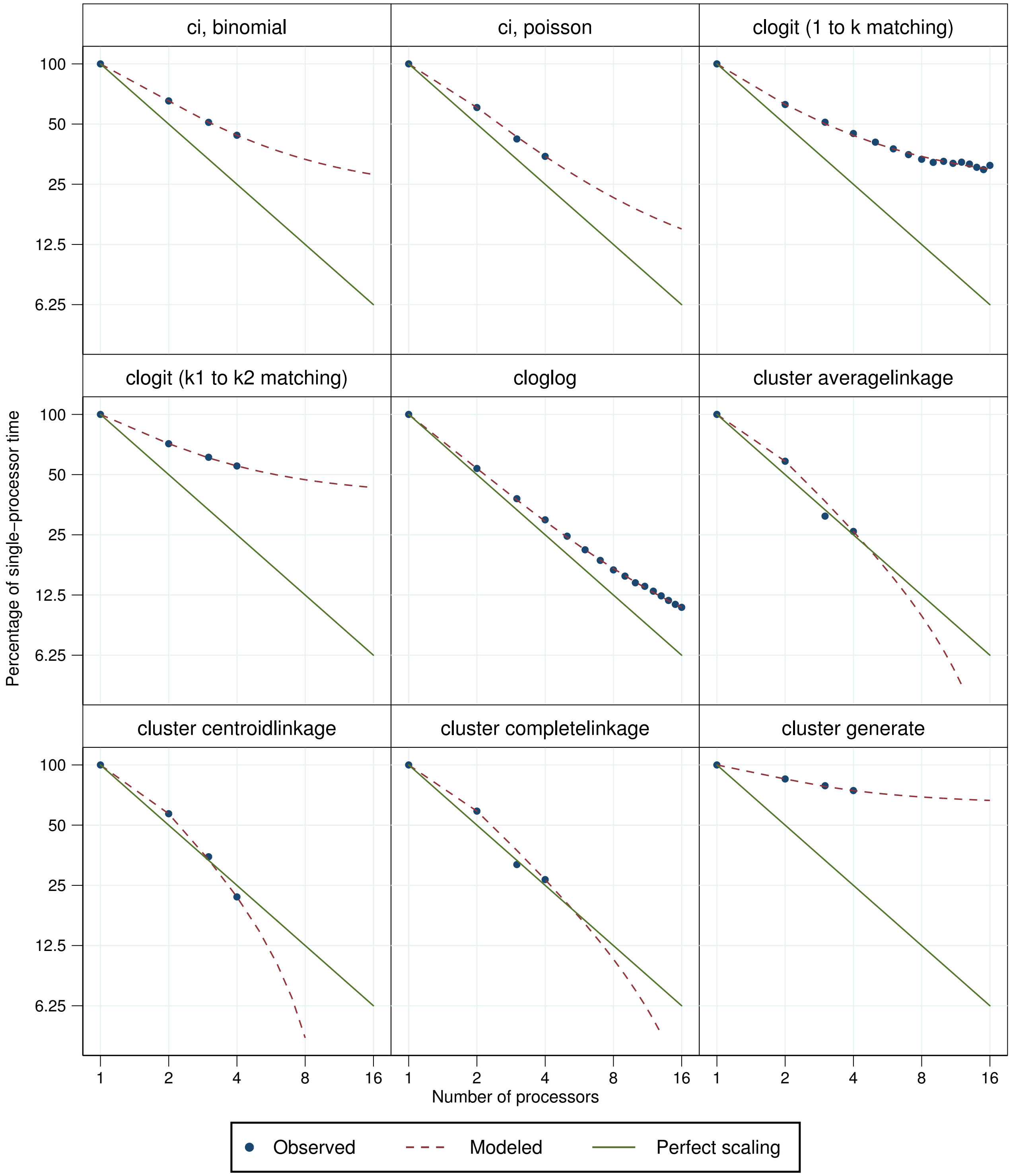

Figure B.4. **Parallelization performance plots.**

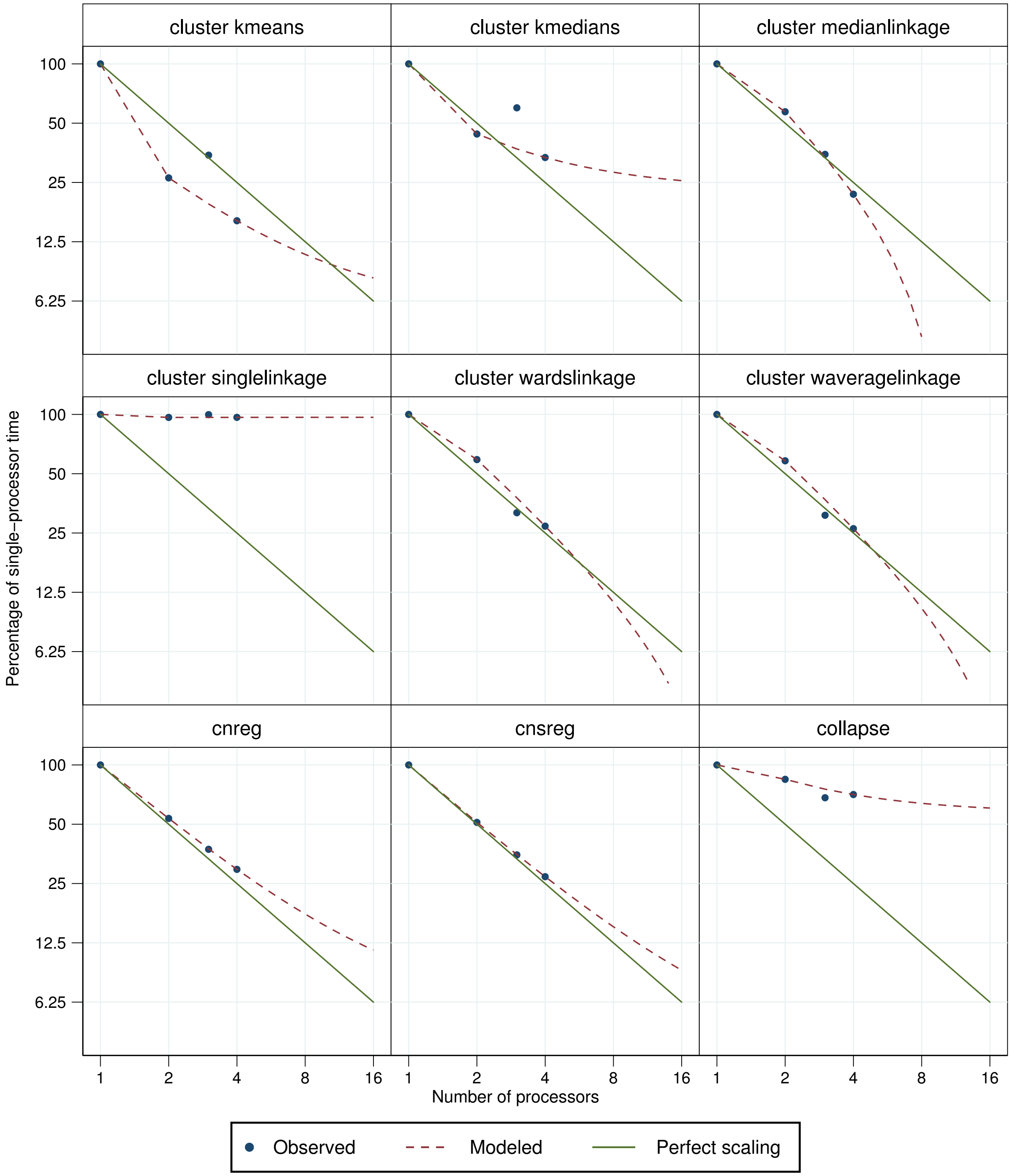

Figure B.5. **Parallelization performance plots.**

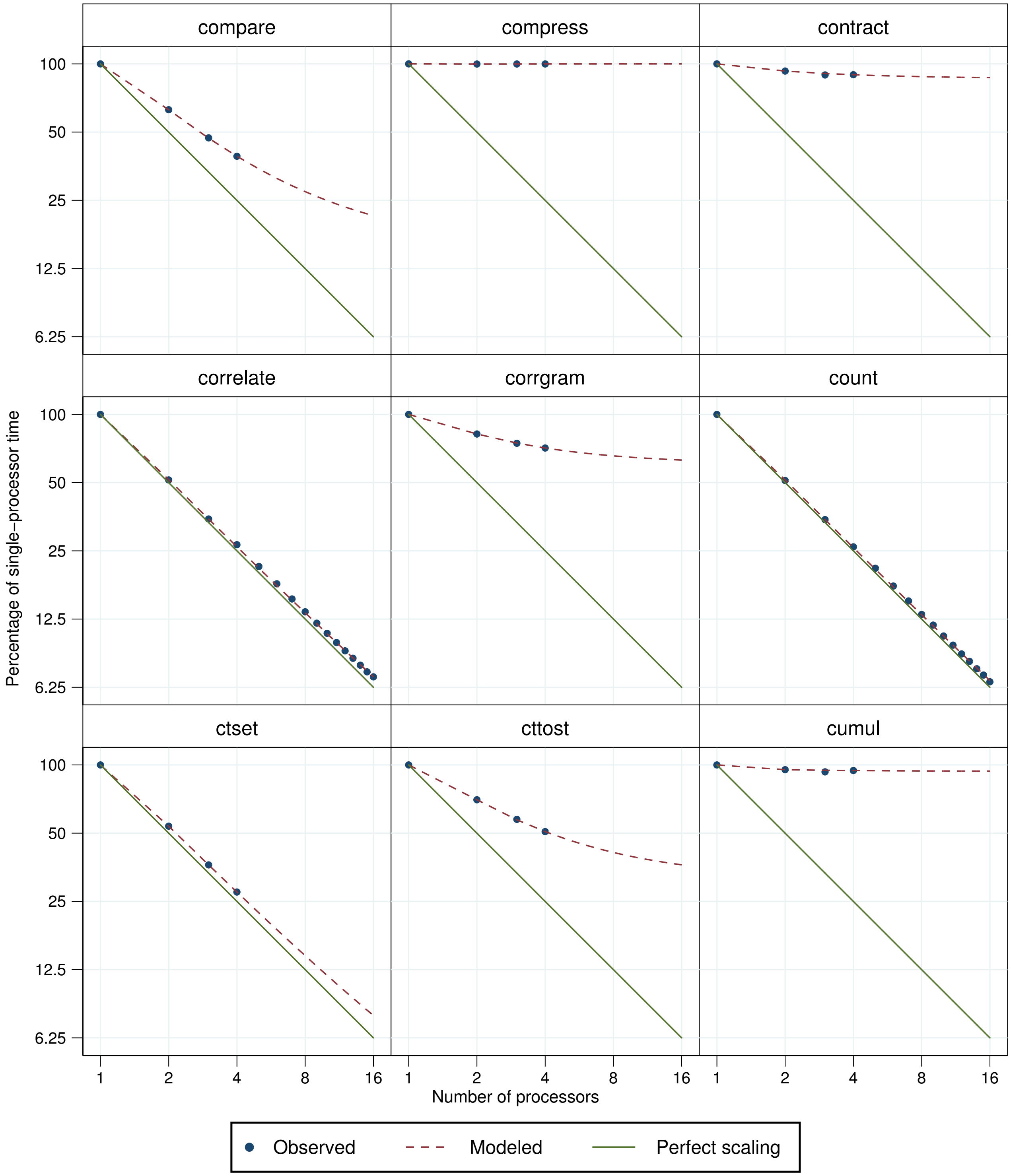

Figure B.6. **Parallelization performance plots.**

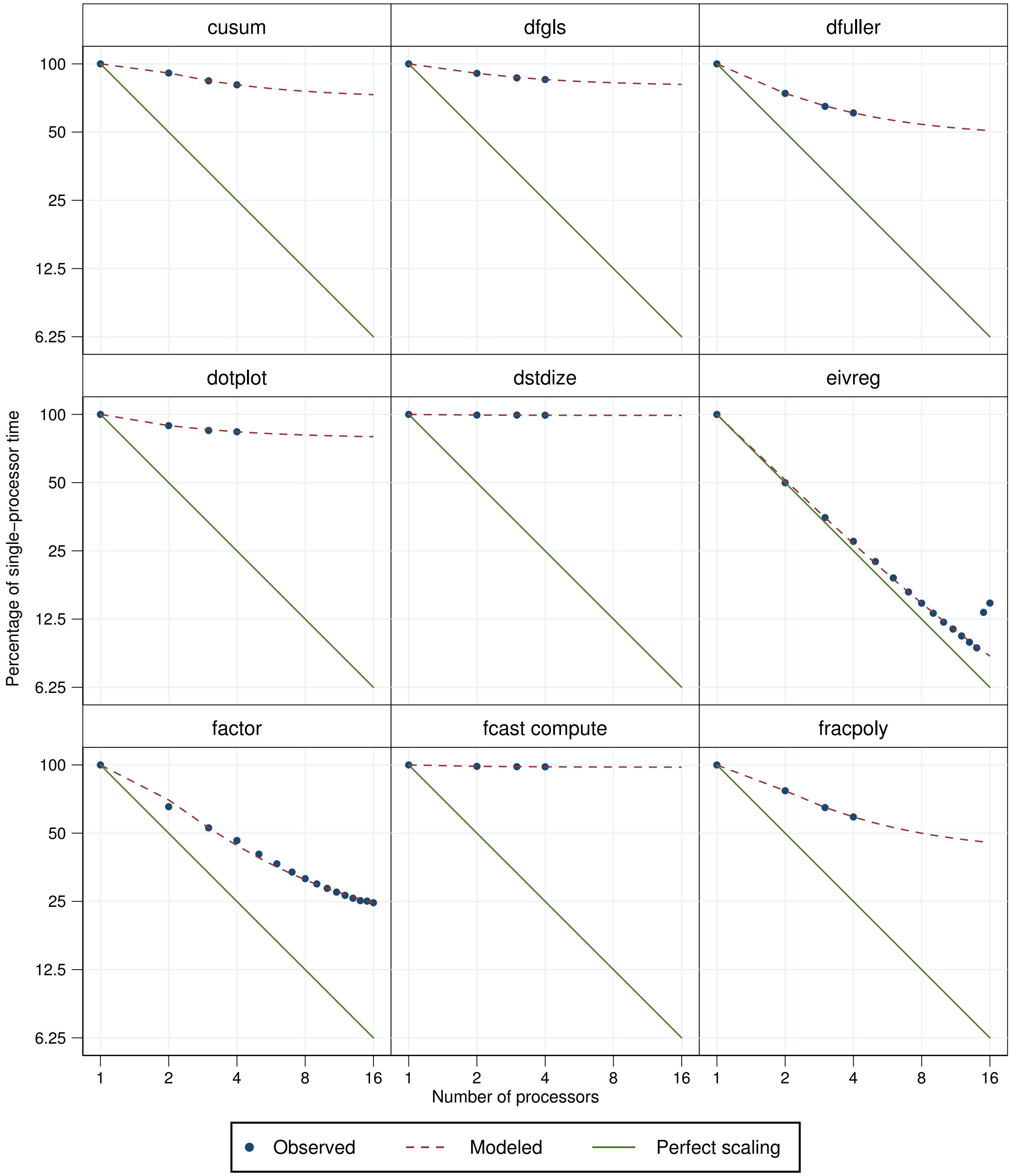

Figure B.7. **Parallelization performance plots.**

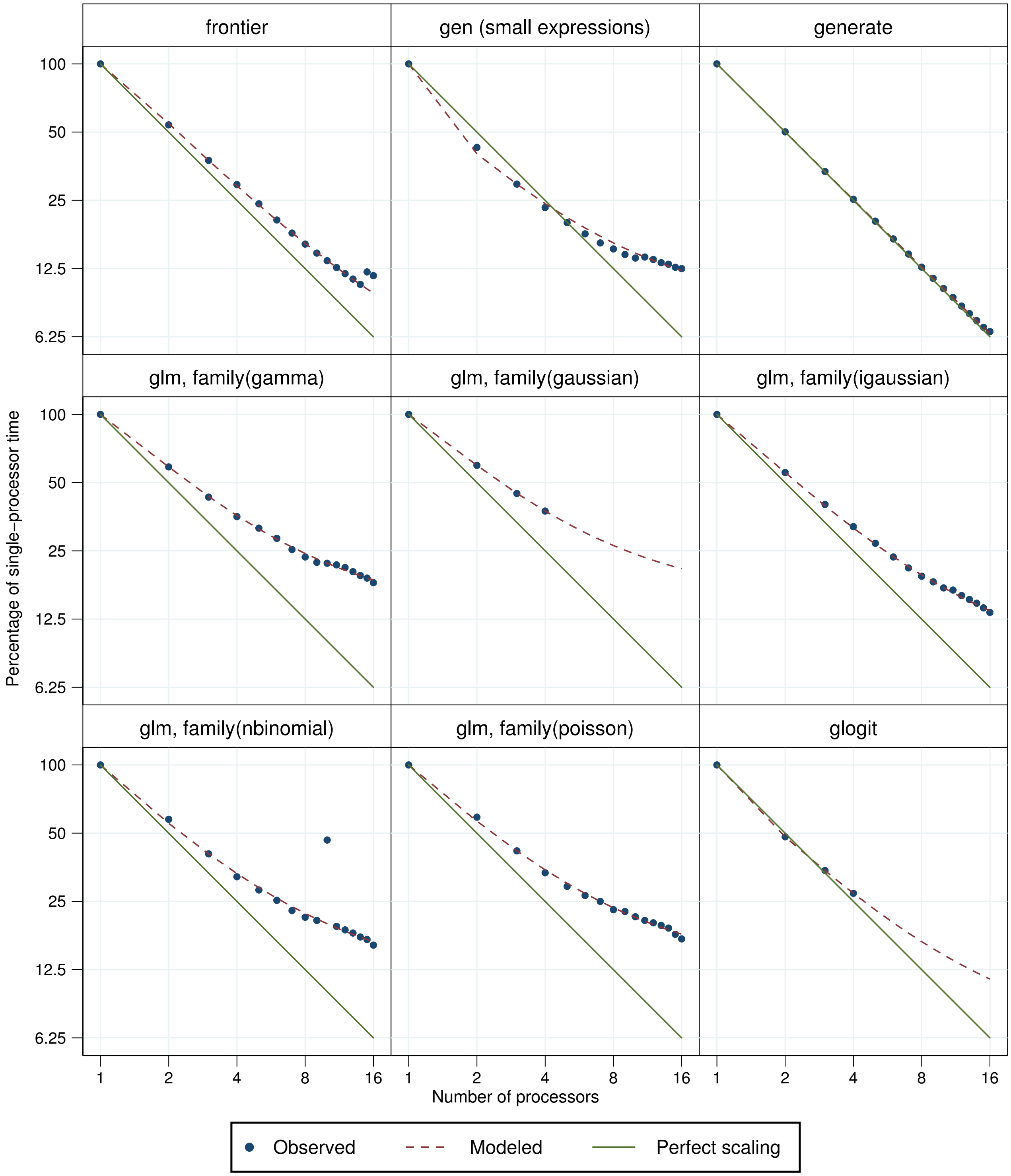

Figure B.8. **Parallelization performance plots.**

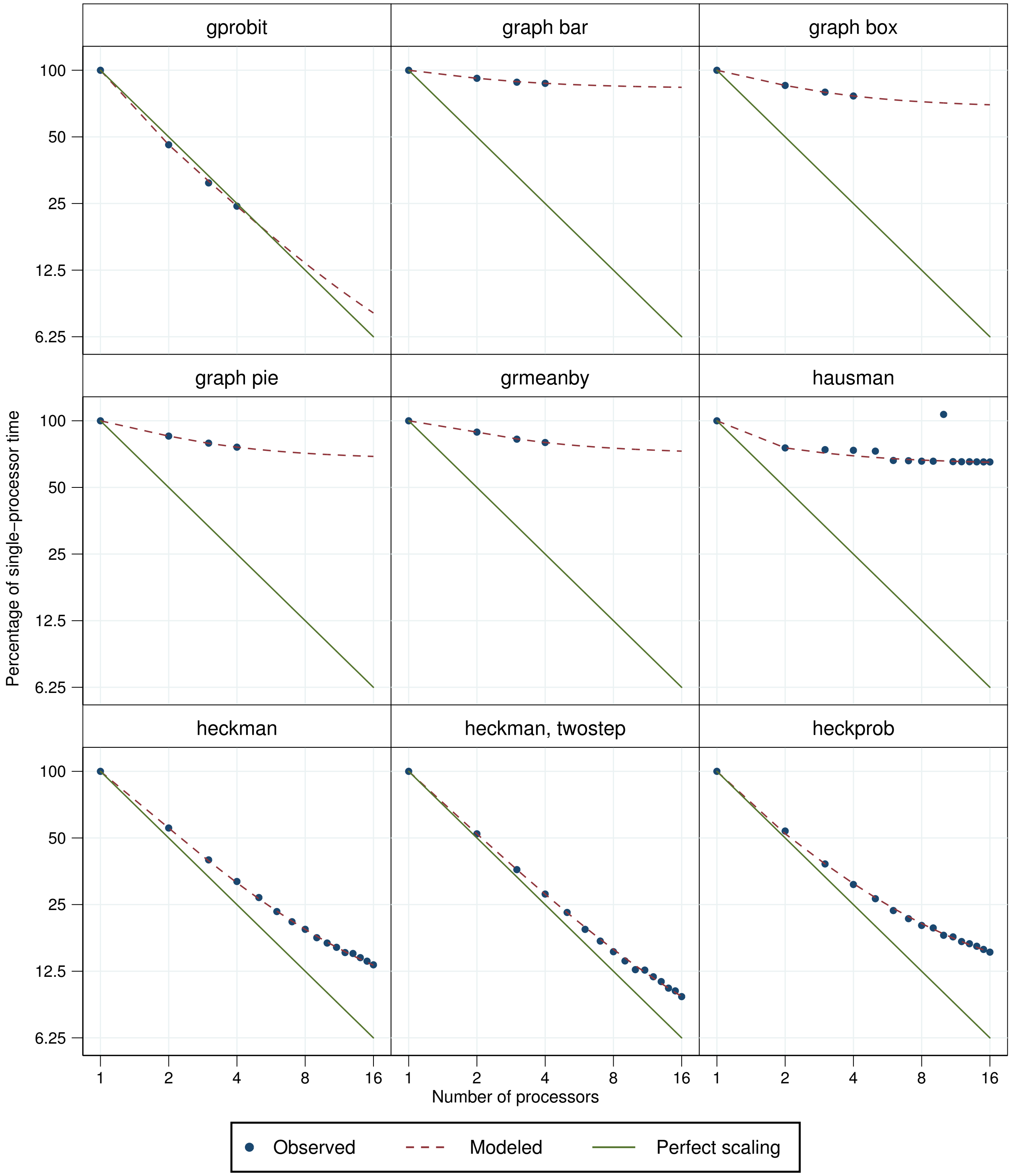

Figure B.9. **Parallelization performance plots.**

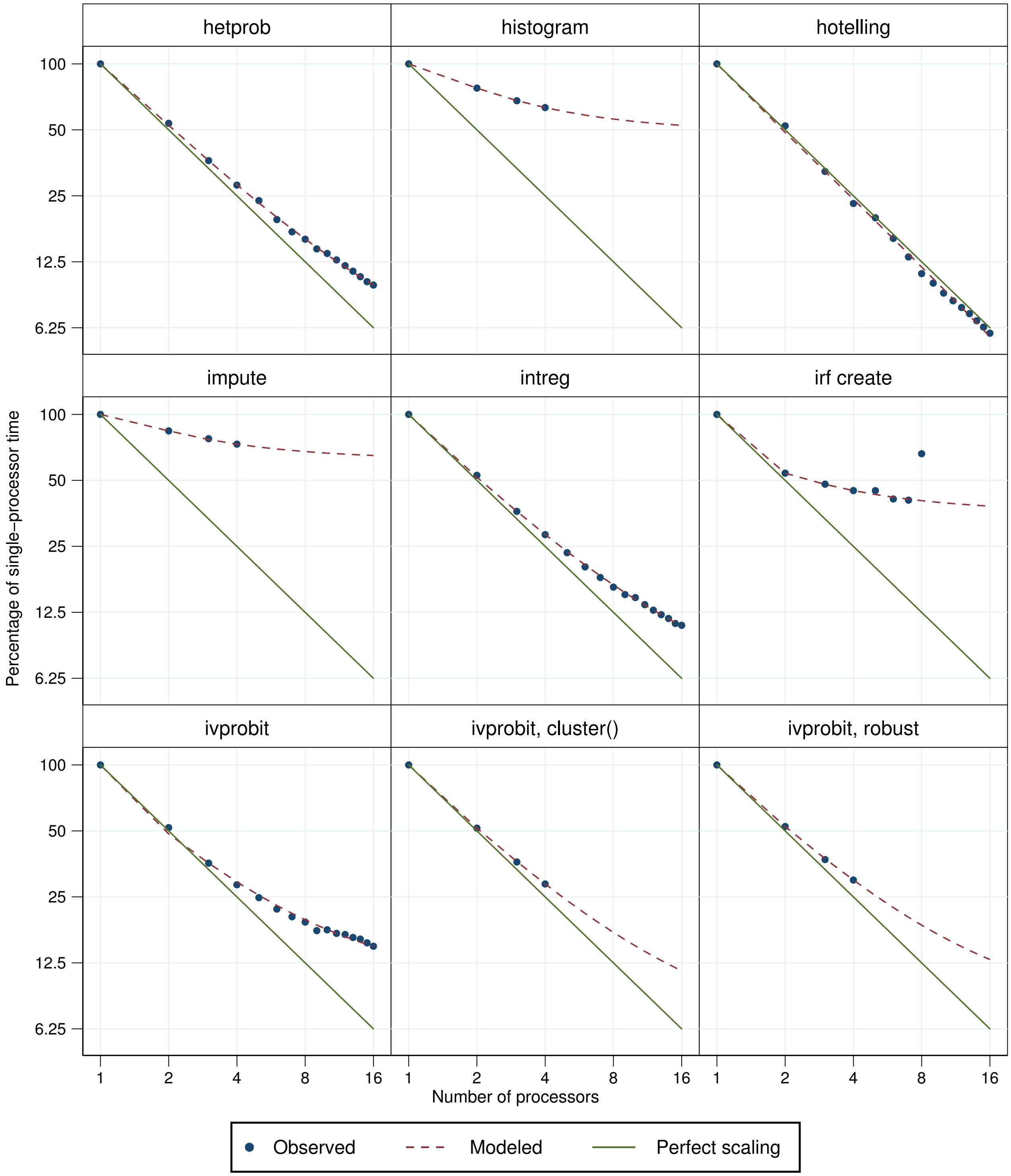

Figure B.10. **Parallelization performance plots.**

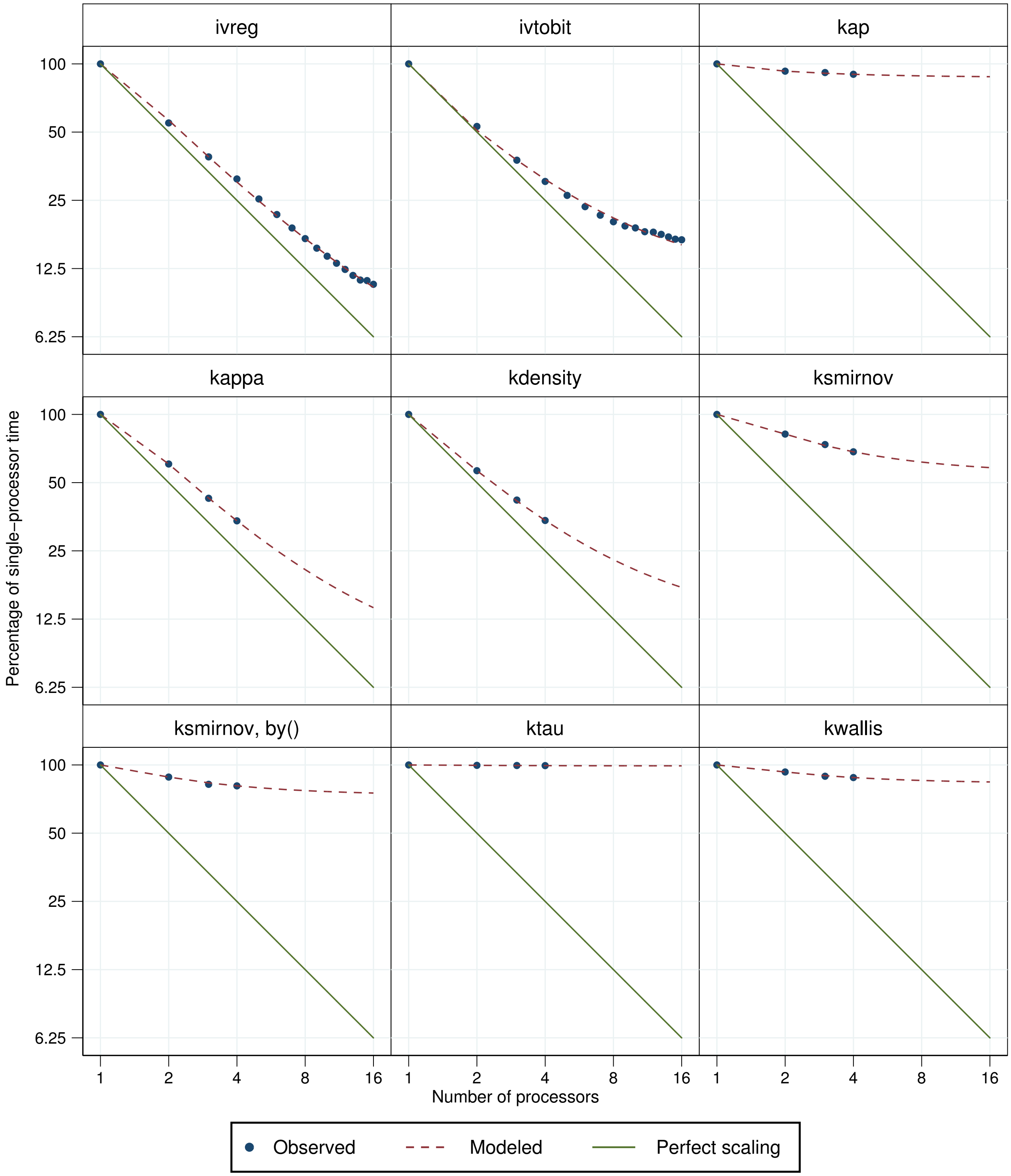

Figure B.11. **Parallelization performance plots.**

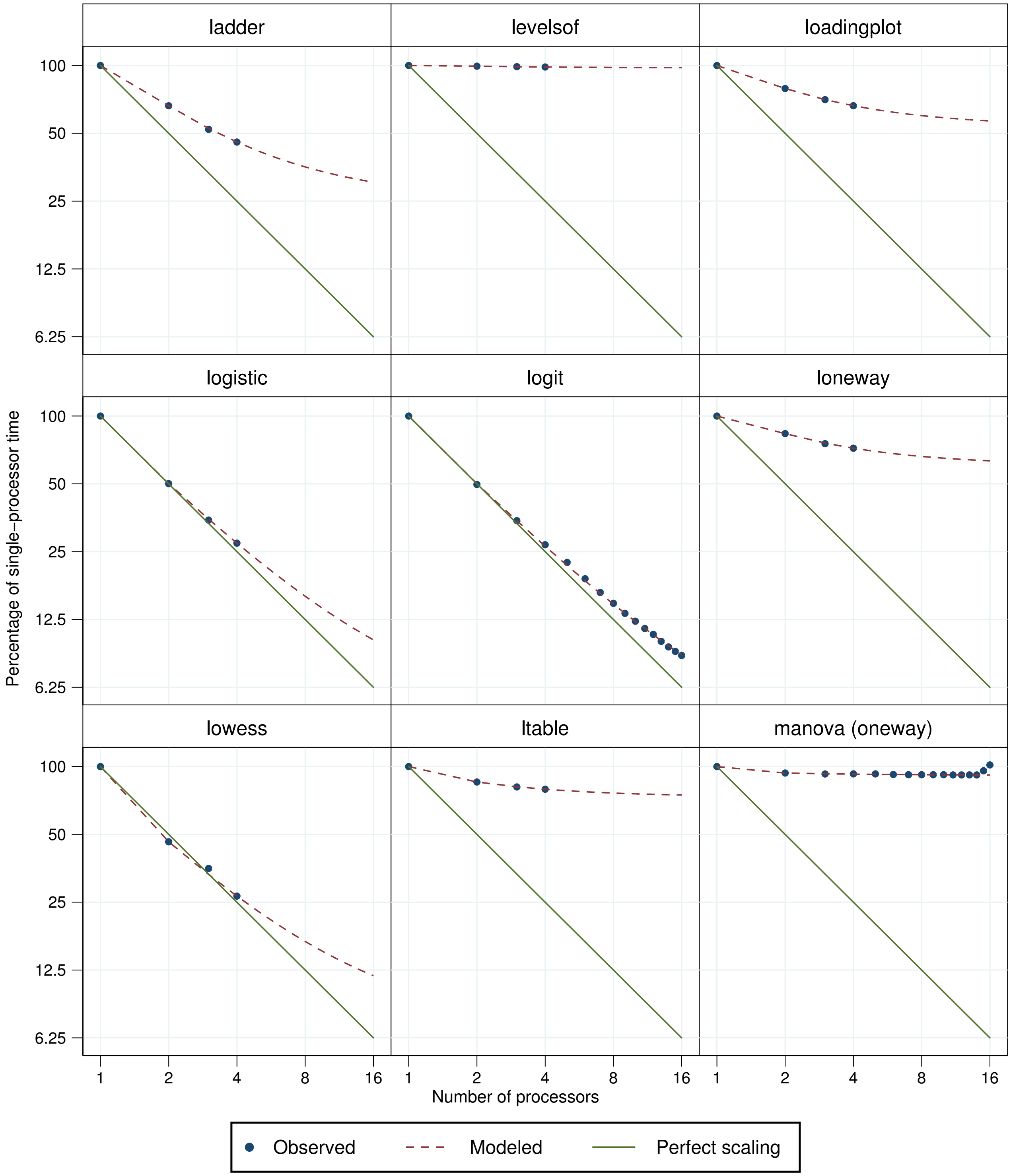

Figure B.12. **Parallelization performance plots.**

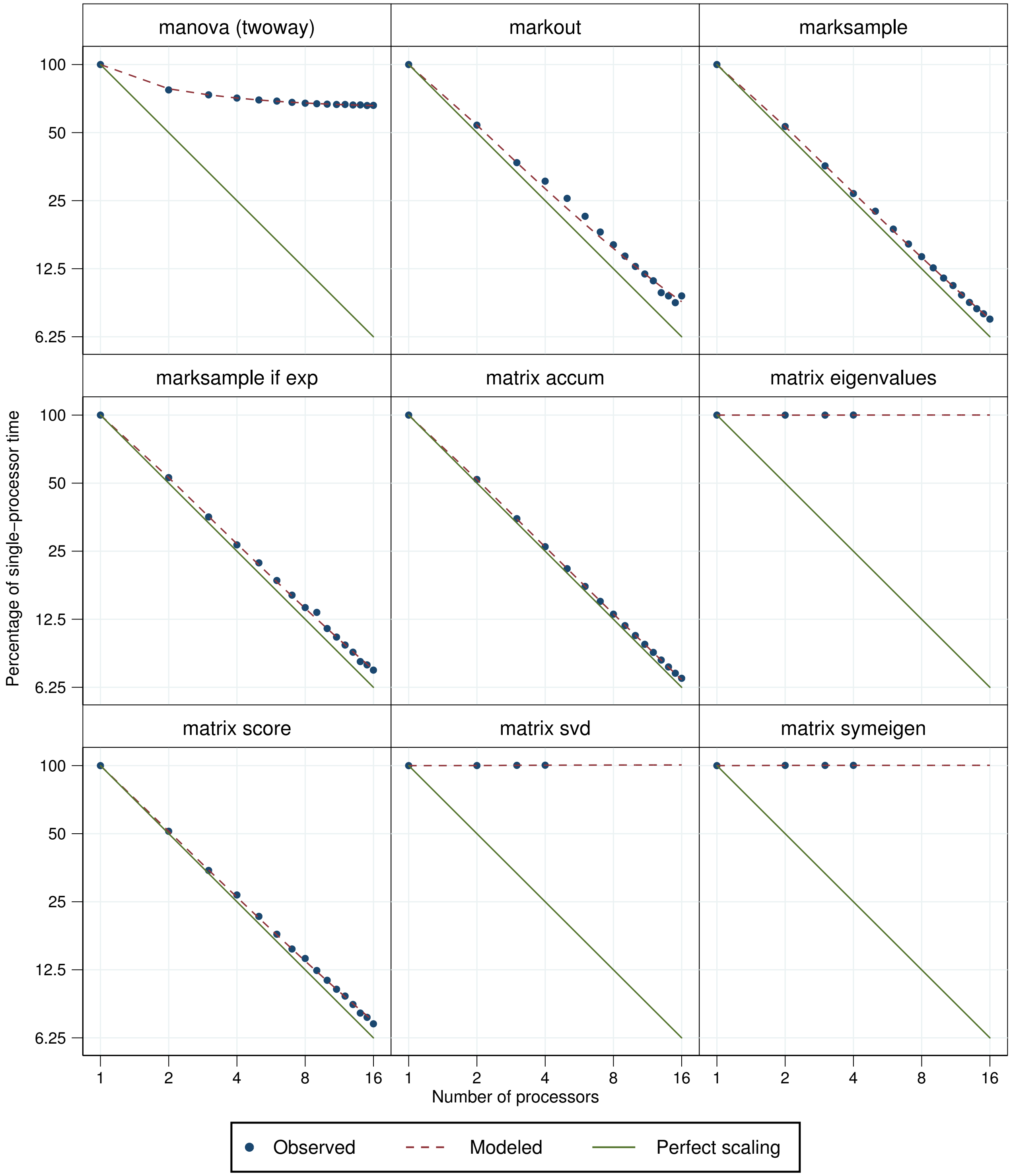

Figure B.13. **Parallelization performance plots.**

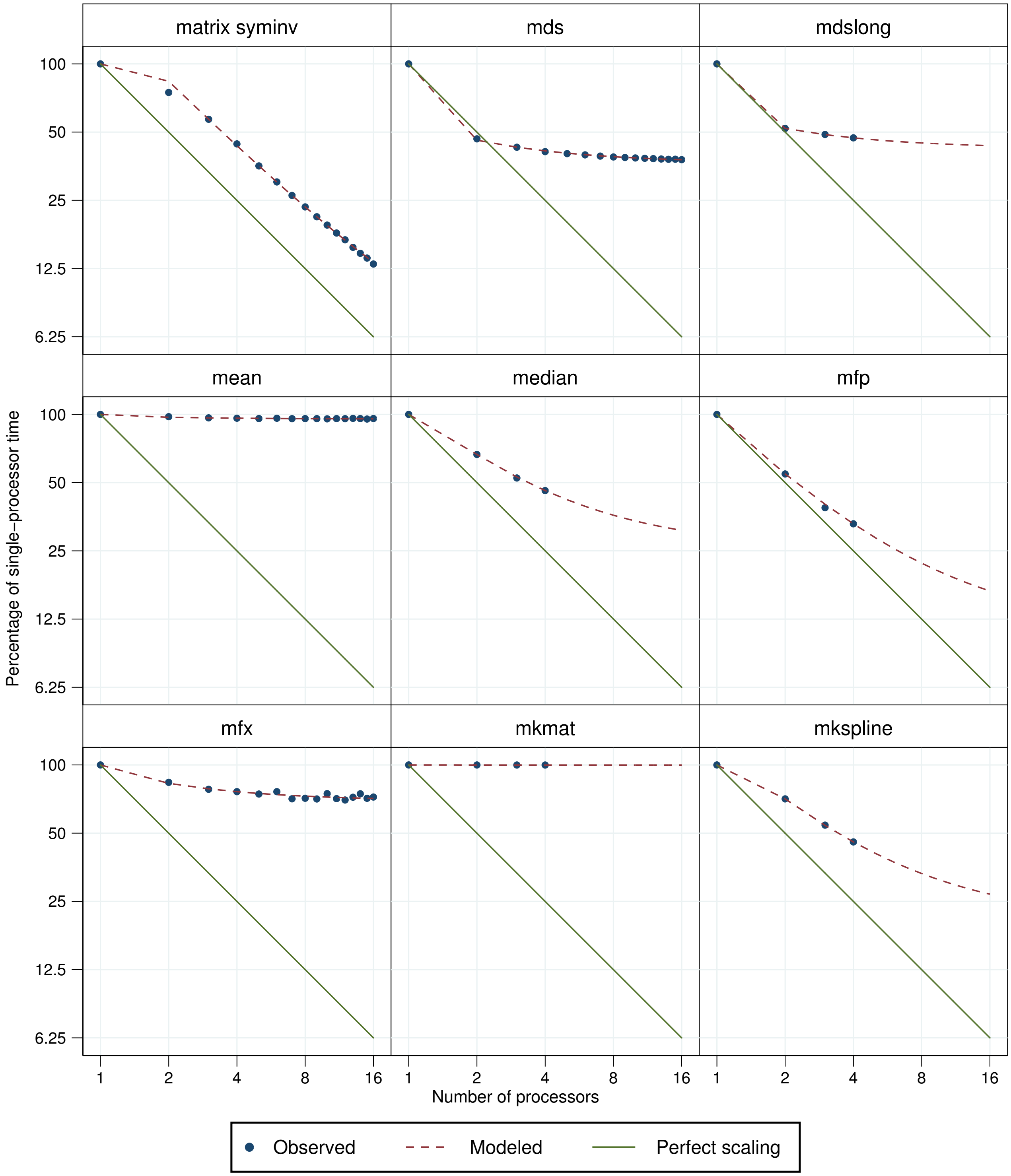

Figure B.14. **Parallelization performance plots.**

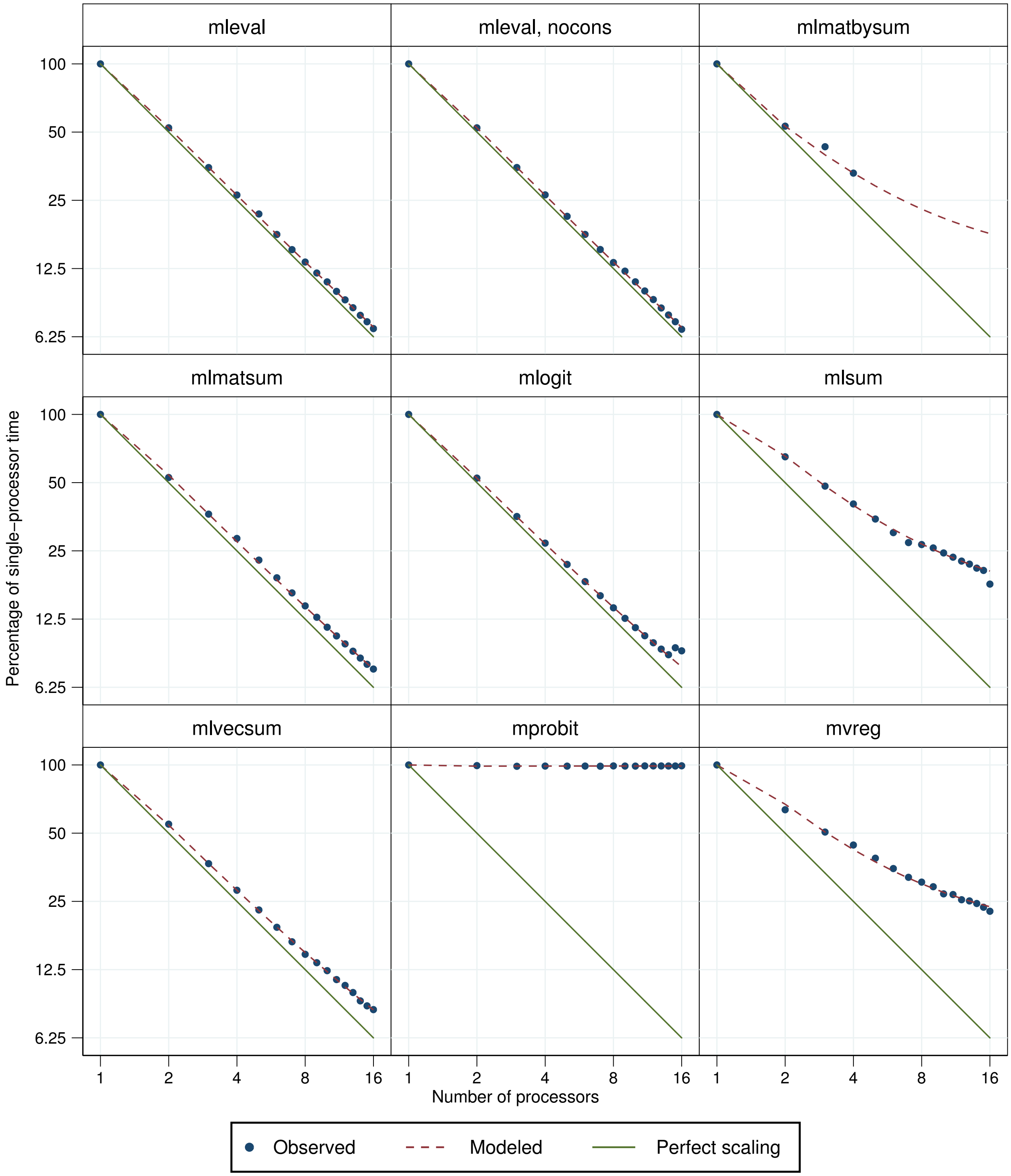

Figure B.15. **Parallelization performance plots.**

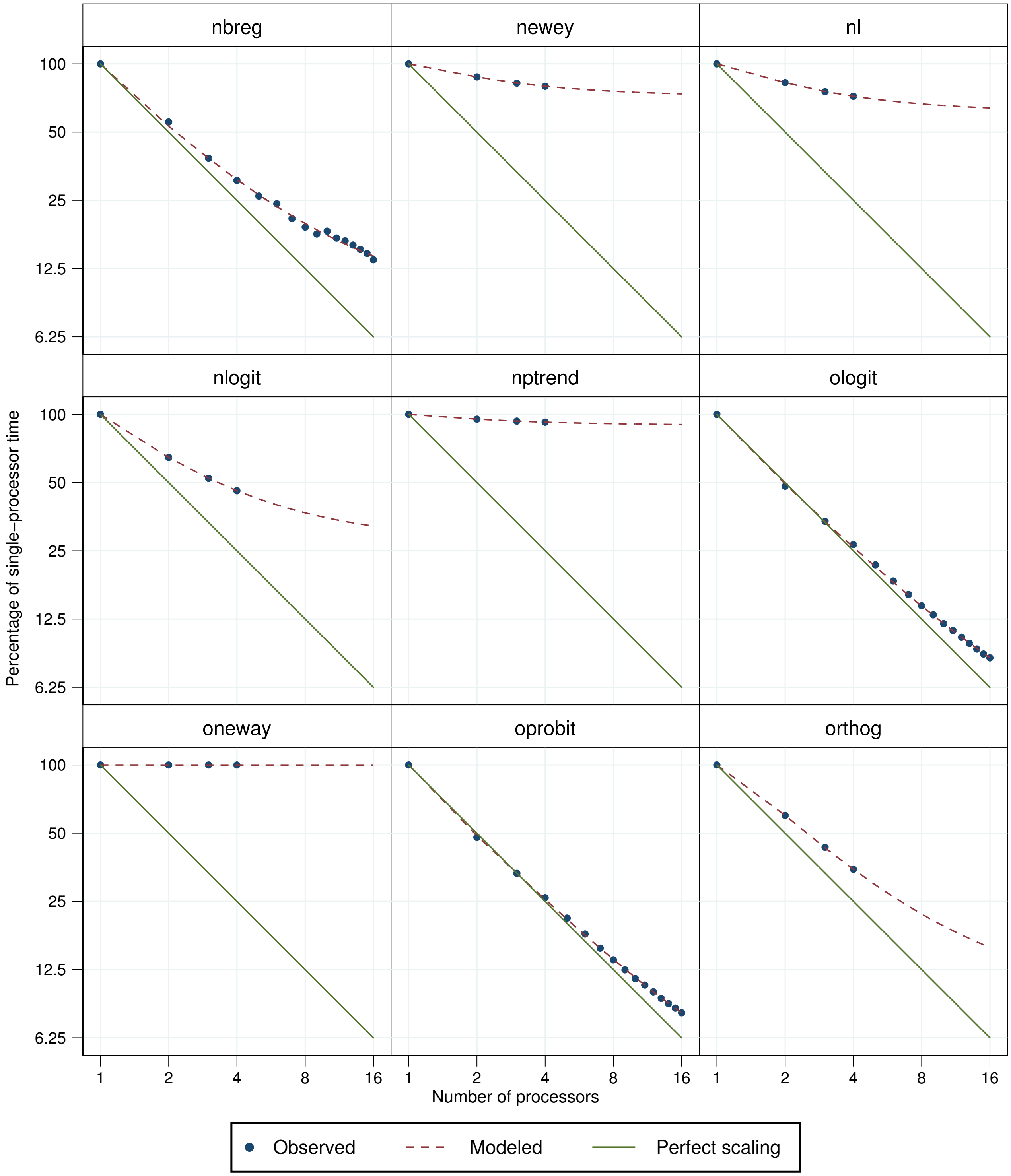

Figure B.16. **Parallelization performance plots.**

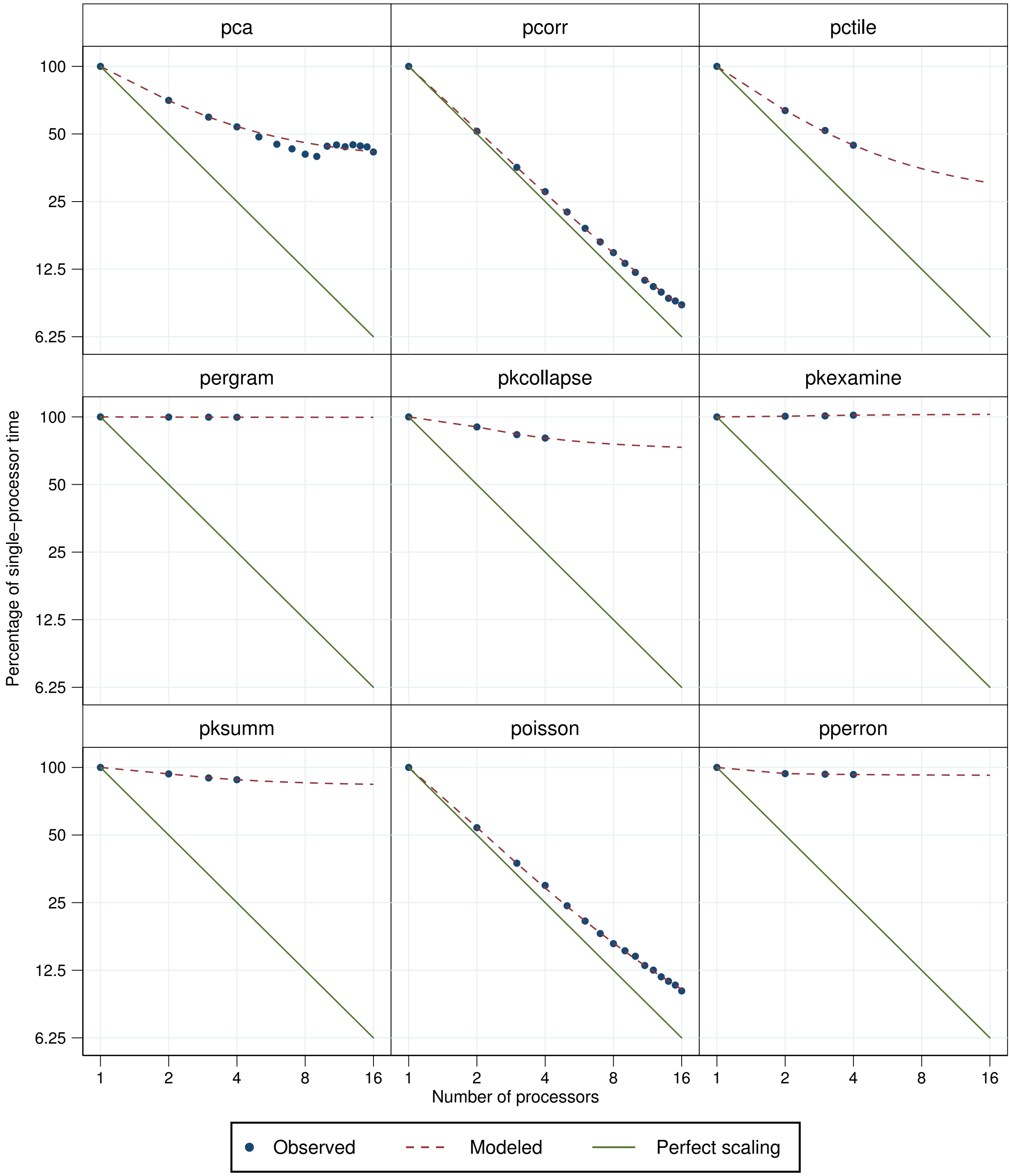

Figure B.17. **Parallelization performance plots.**

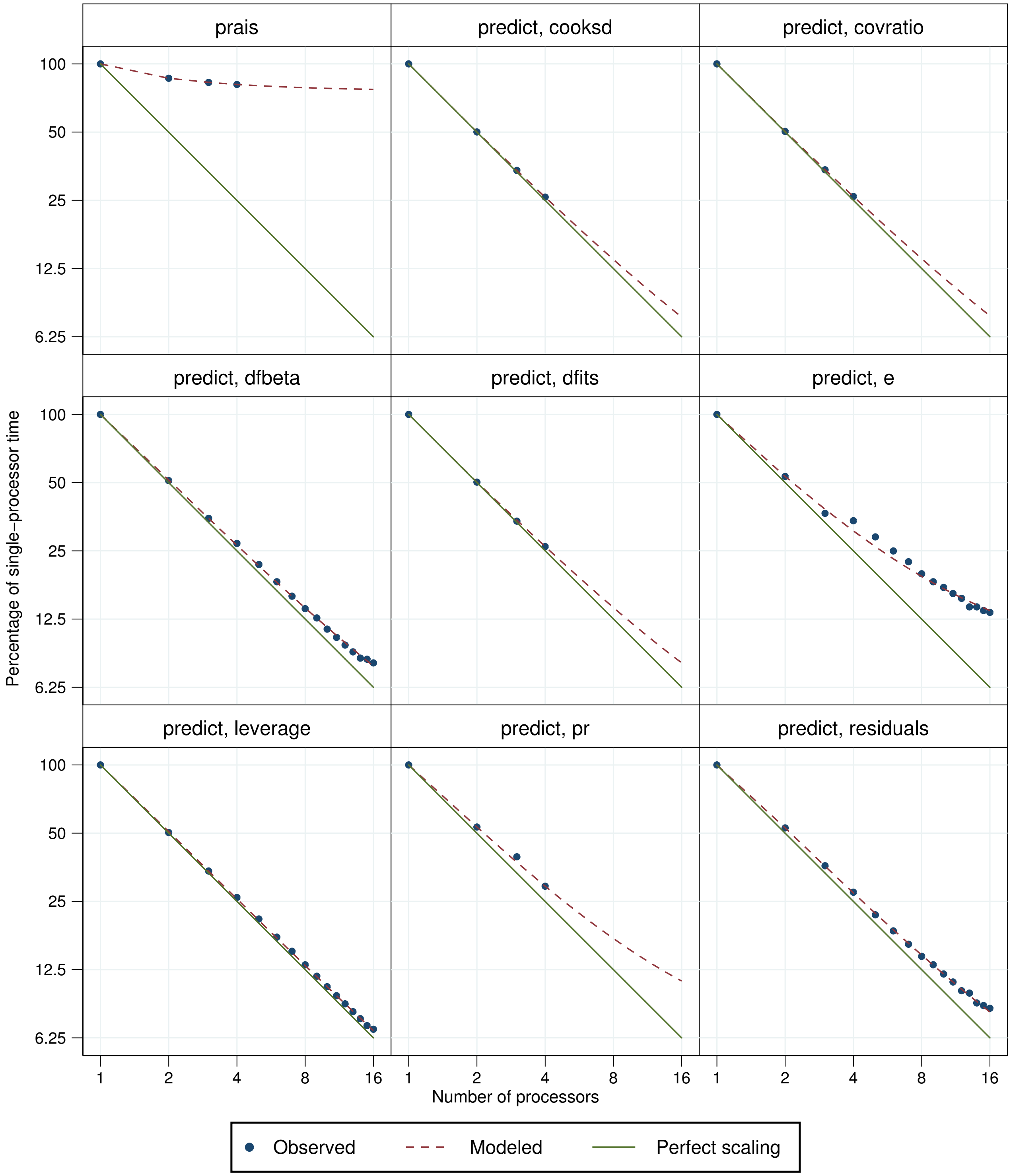

Figure B.18. **Parallelization performance plots.**

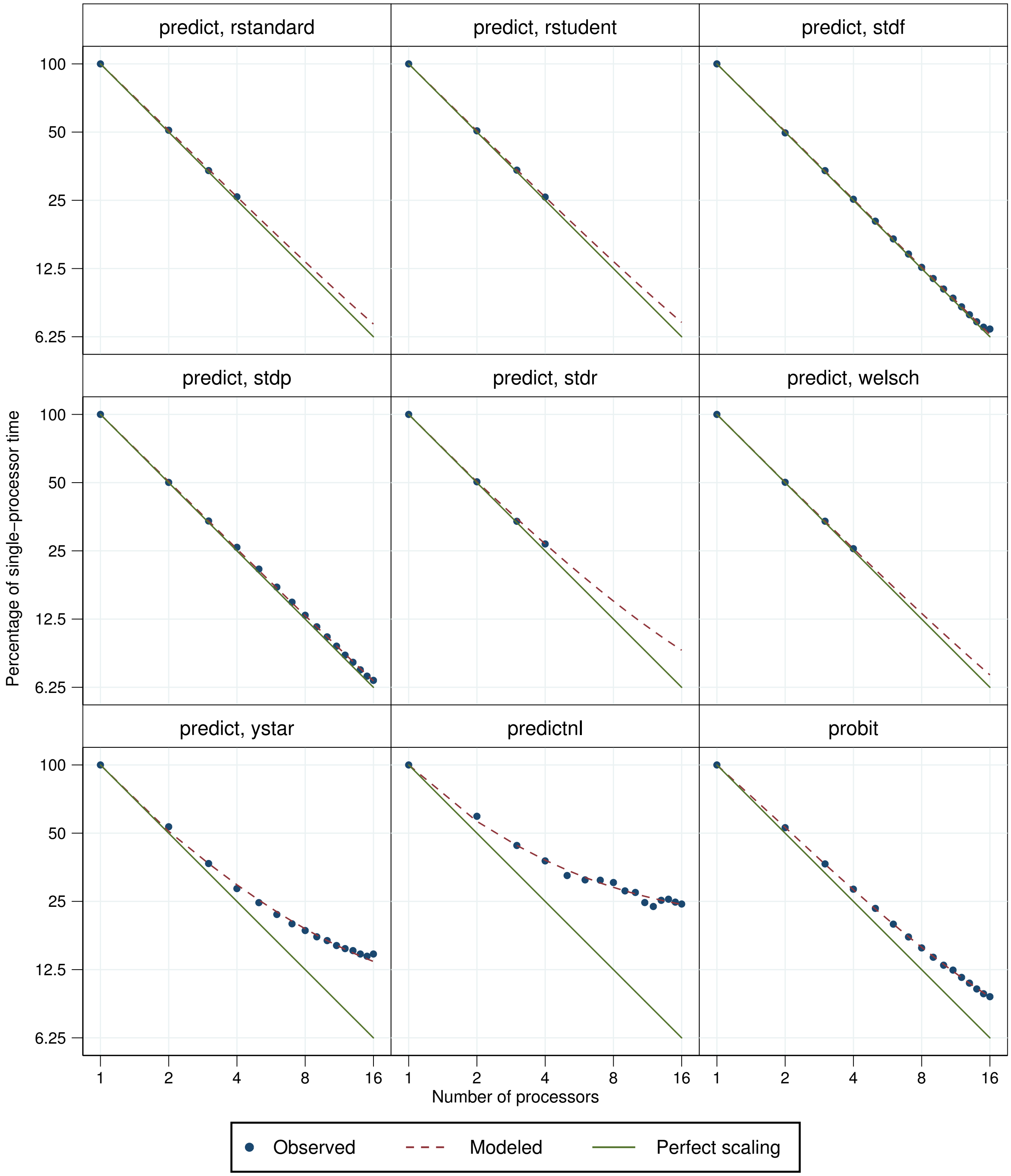

Figure B.19. **Parallelization performance plots.**

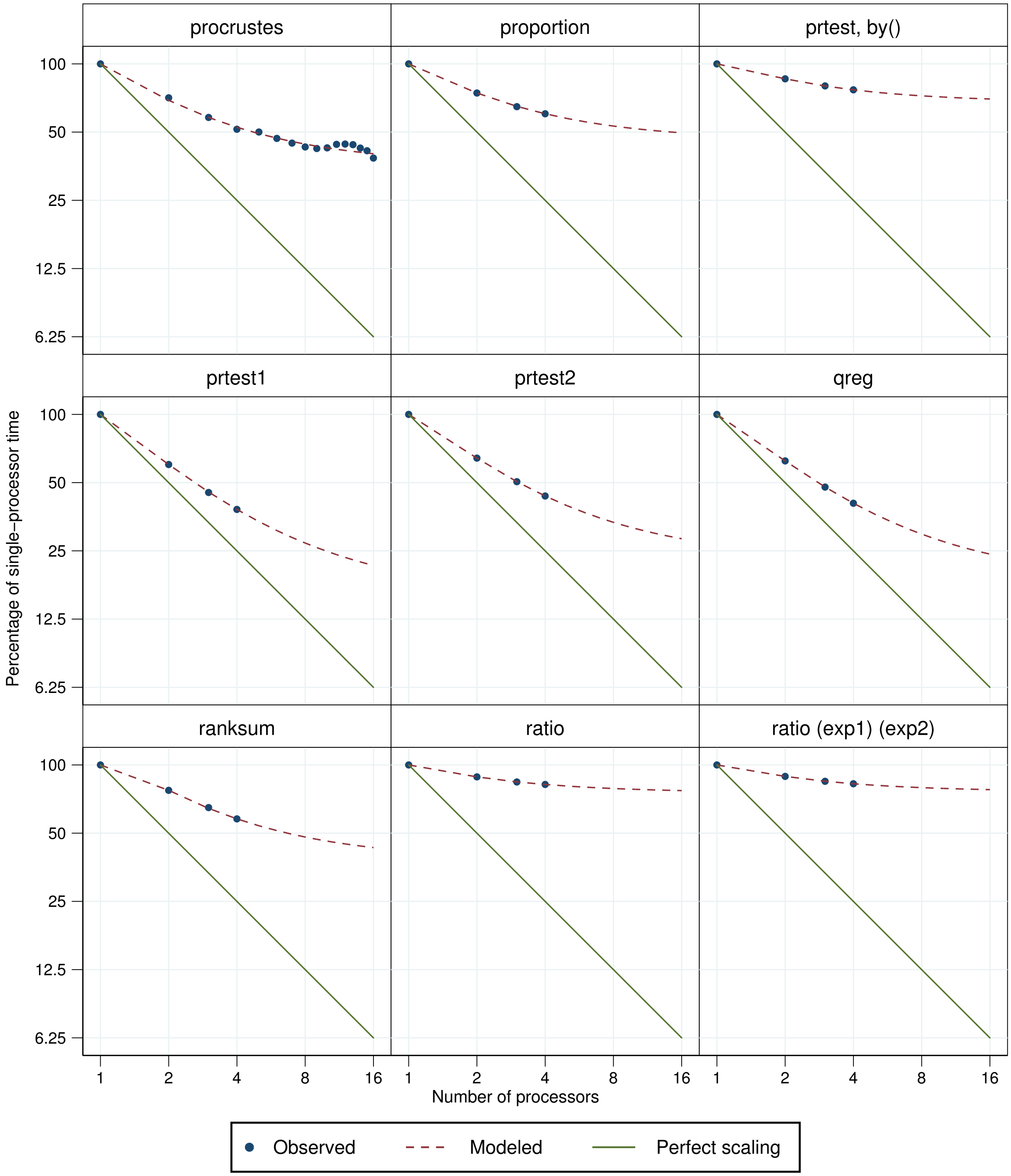

Figure B.20. **Parallelization performance plots.**

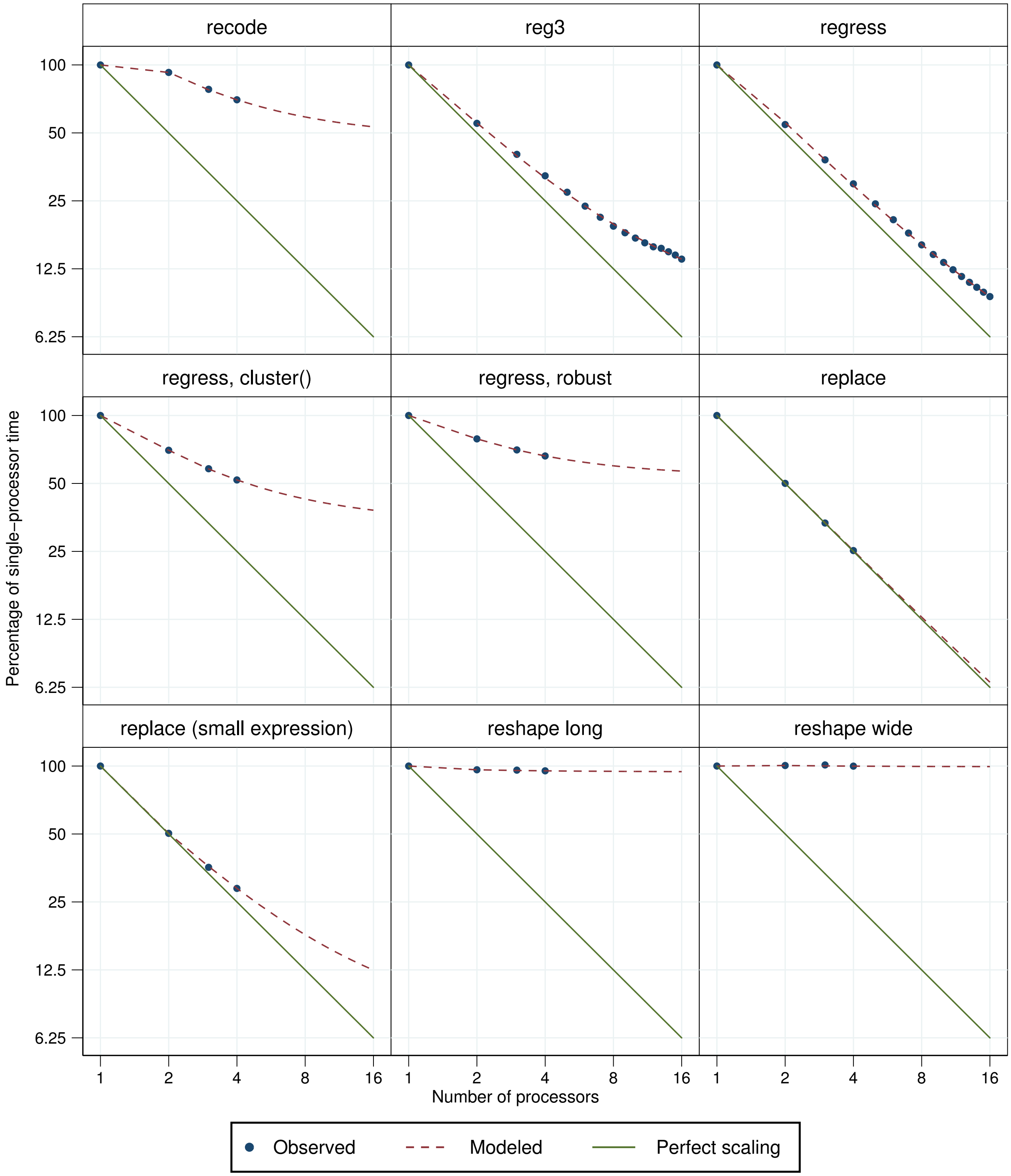

Figure B.21. **Parallelization performance plots.**

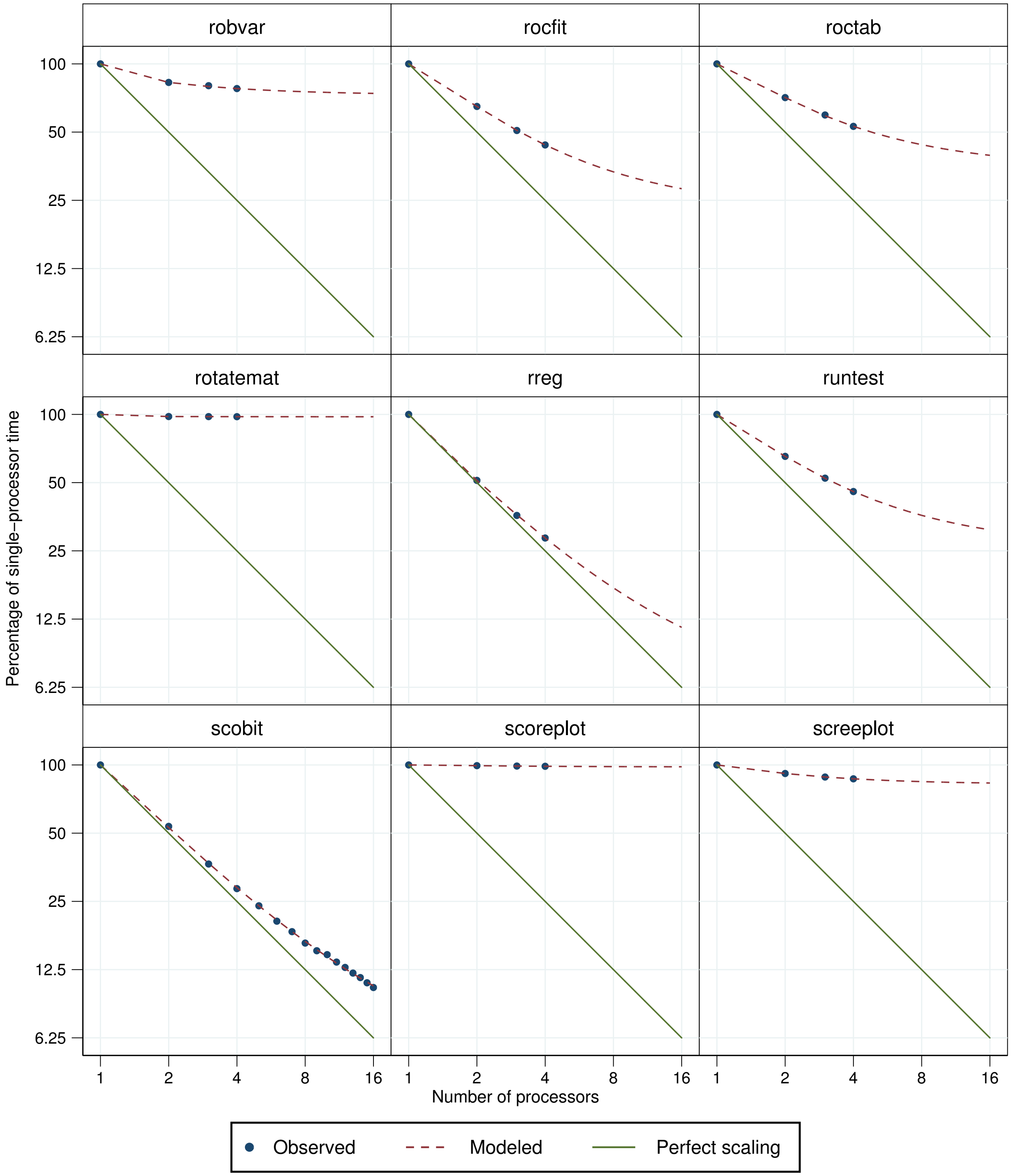

Figure B.22. **Parallelization performance plots.**

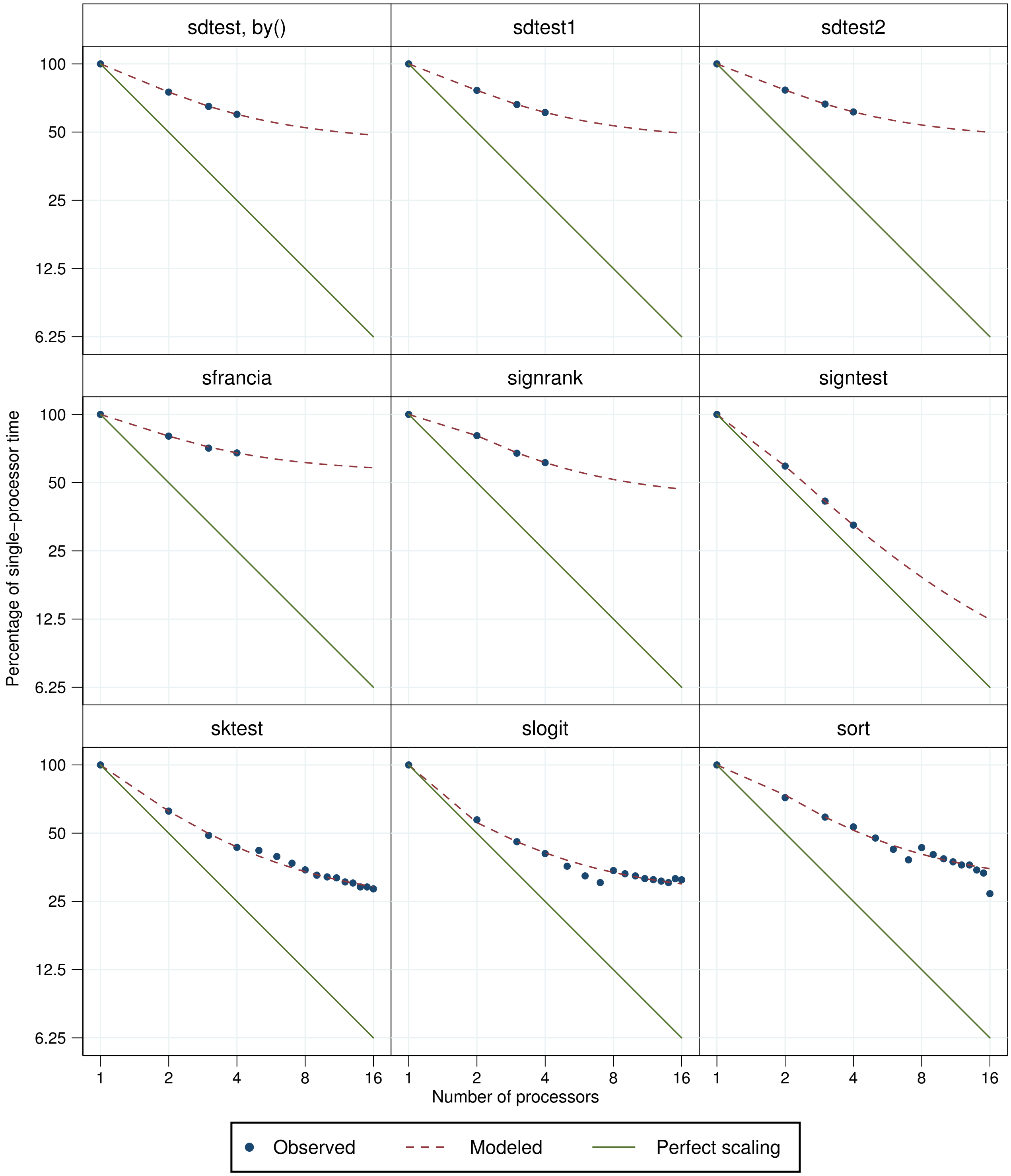

Figure B.23. **Parallelization performance plots.**

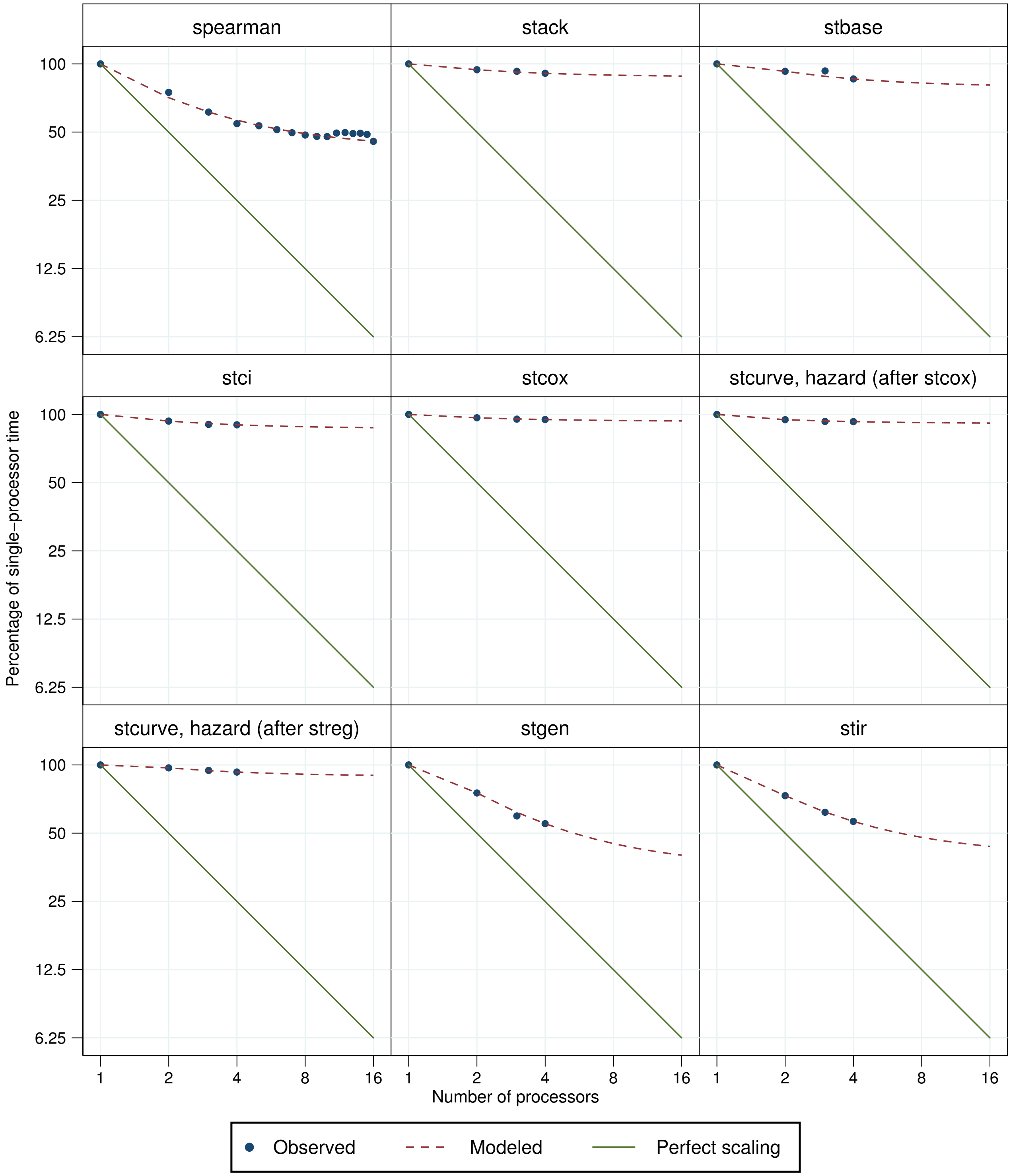

Figure B.24. **Parallelization performance plots.**

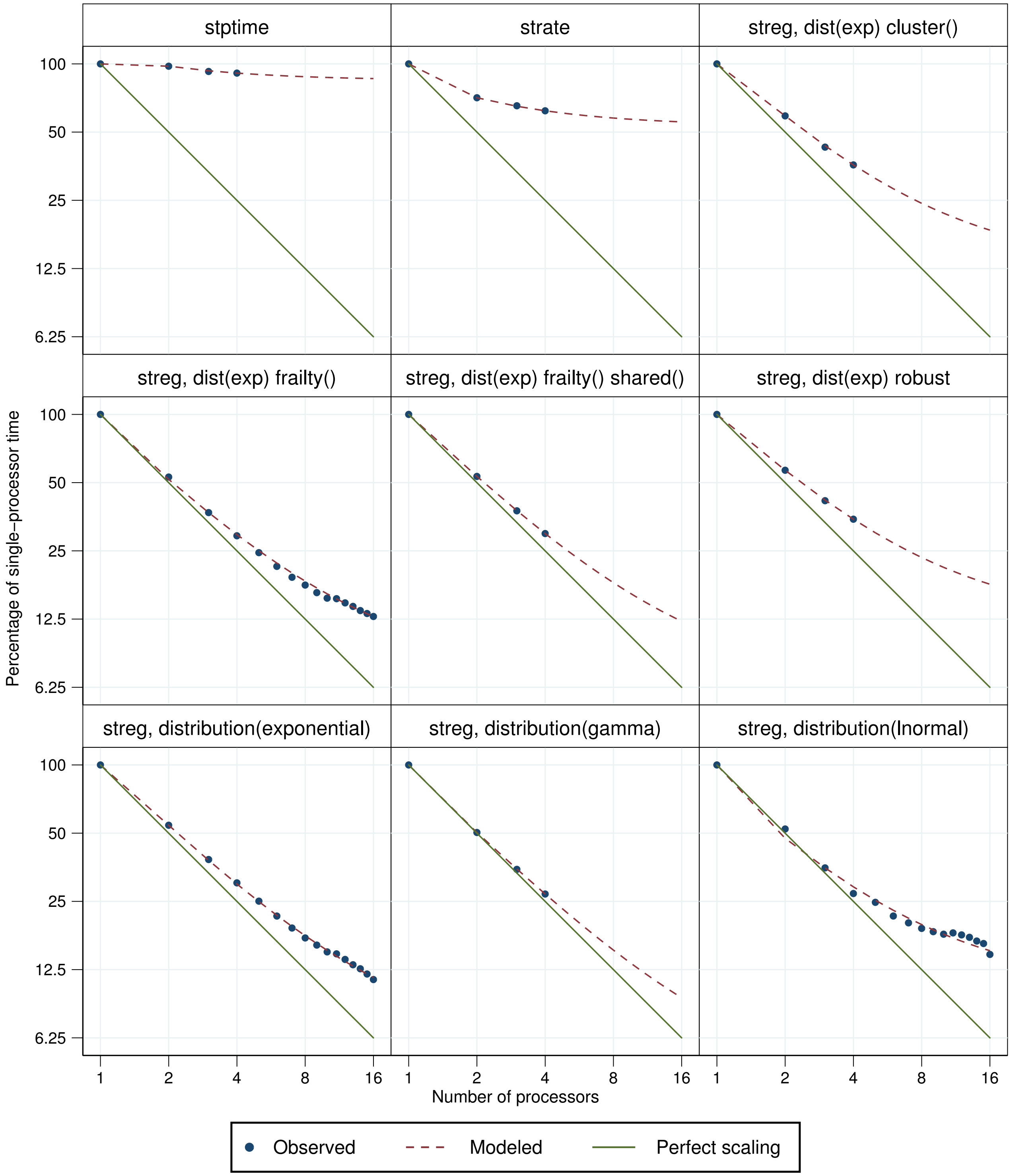

Figure B.25. **Parallelization performance plots.**

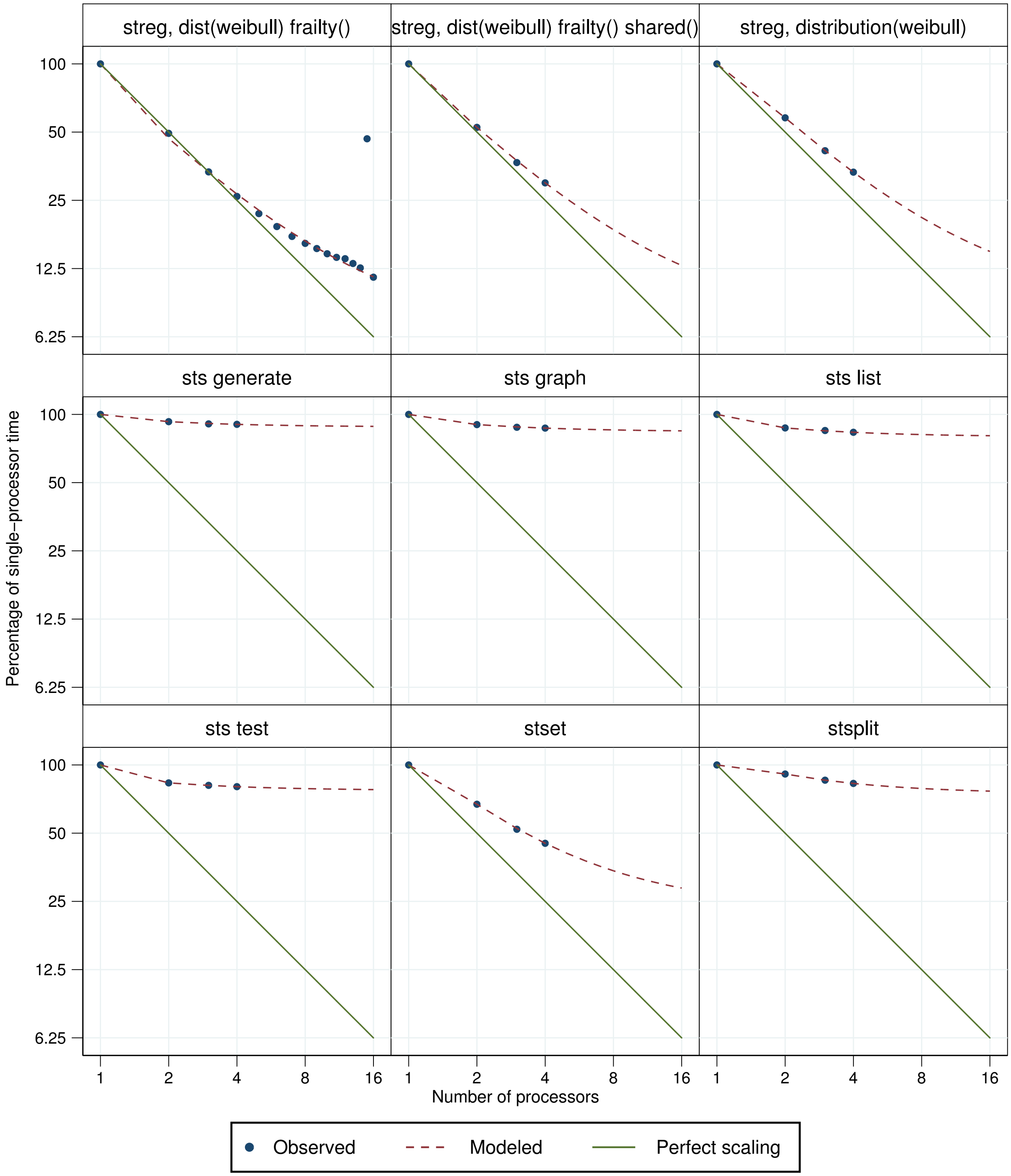

Figure B.26. **Parallelization performance plots.**

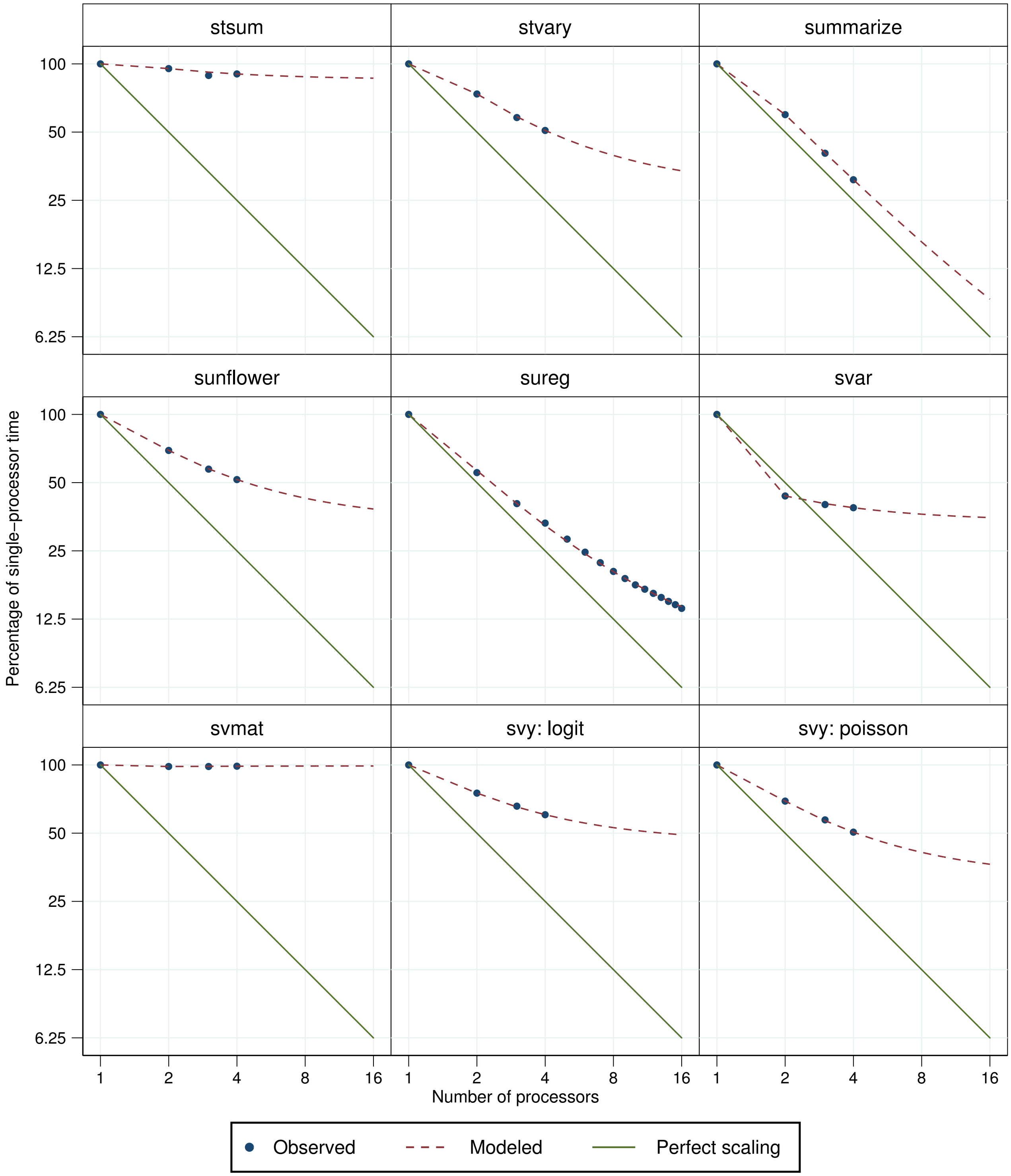

Figure B.27. **Parallelization performance plots.**

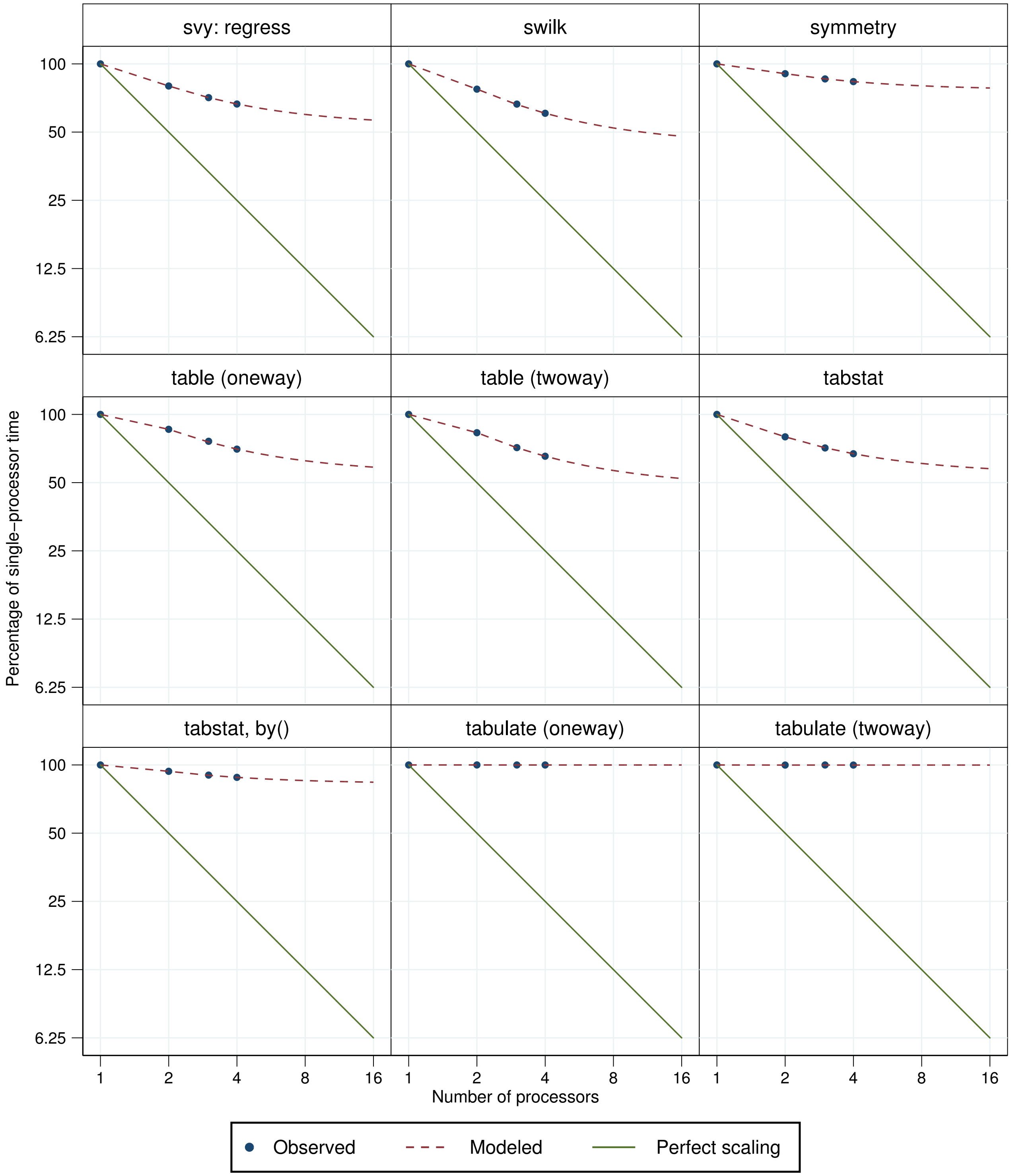

Figure B.28. **Parallelization performance plots.**

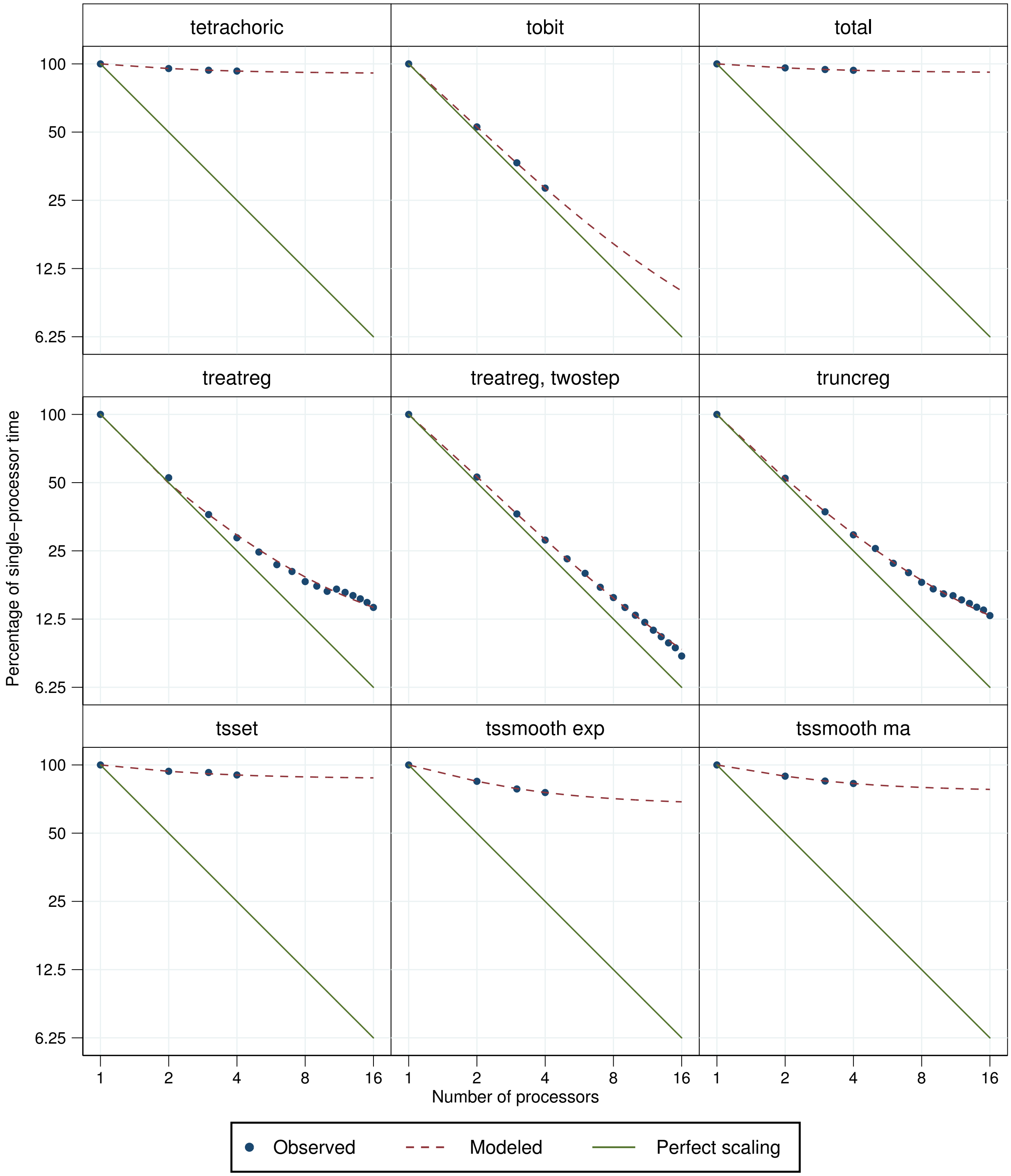

Figure B.29. **Parallelization performance plots.**

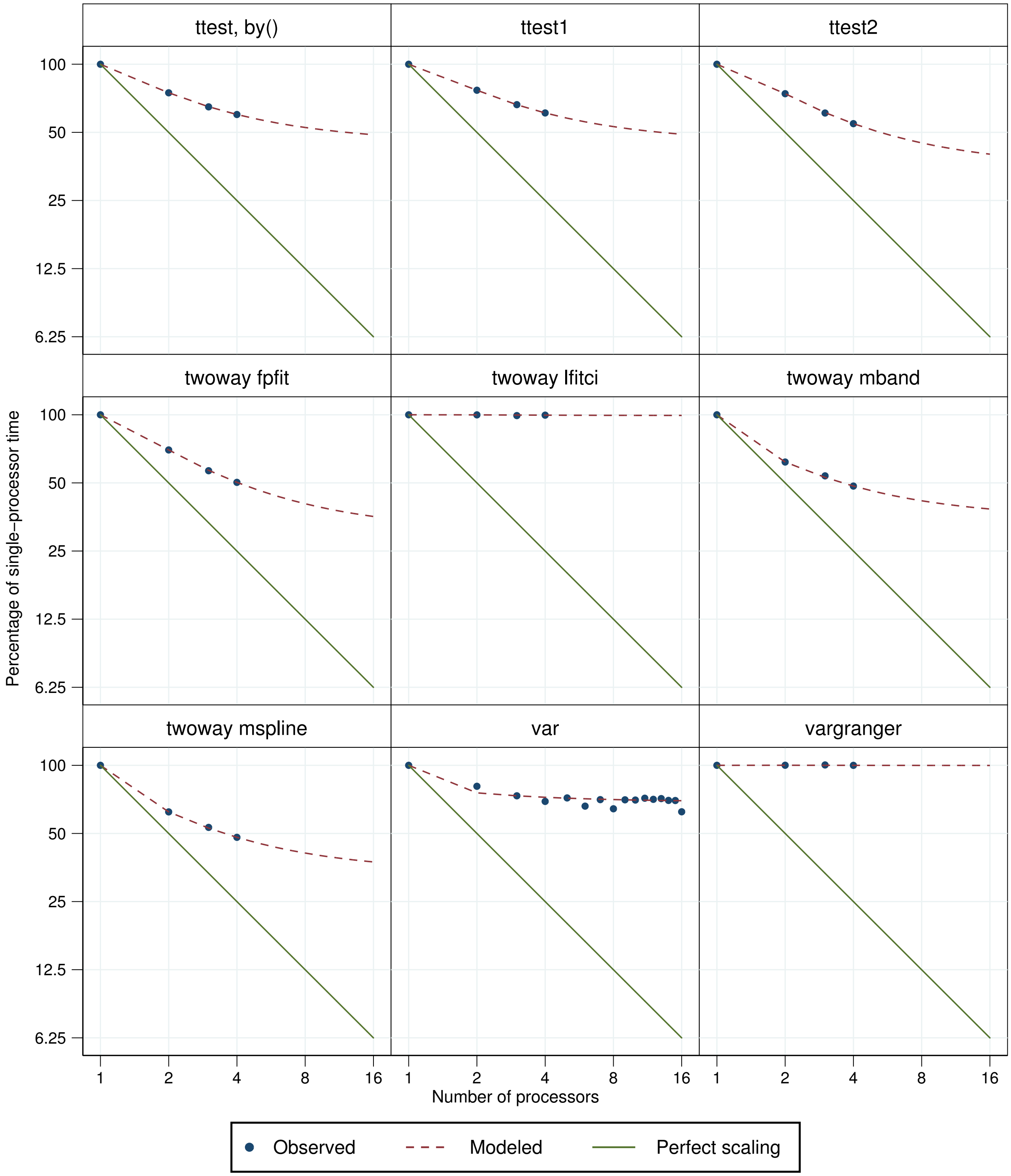

Figure B.30. **Parallelization performance plots.**

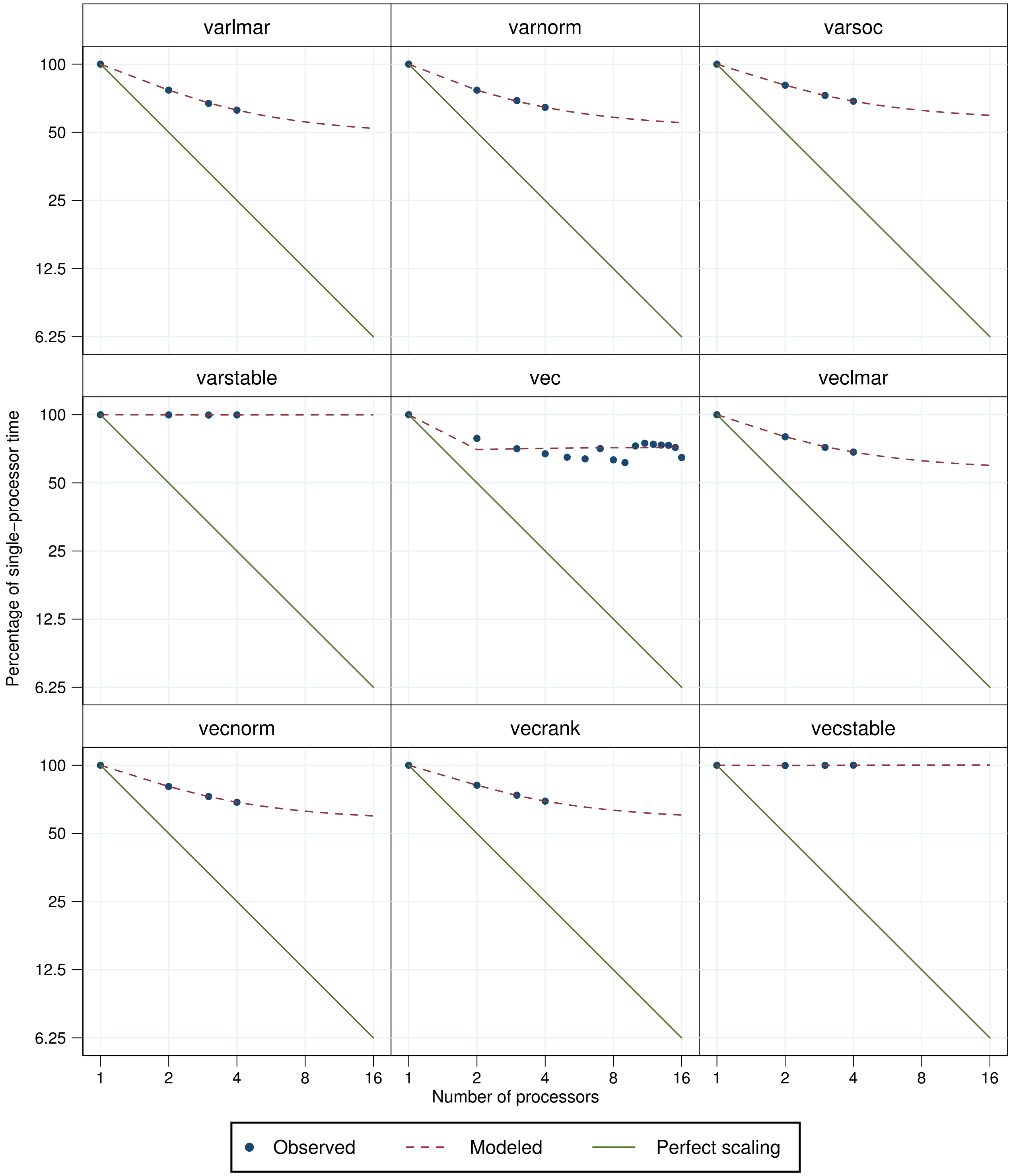

Figure B.31. **Parallelization performance plots.**

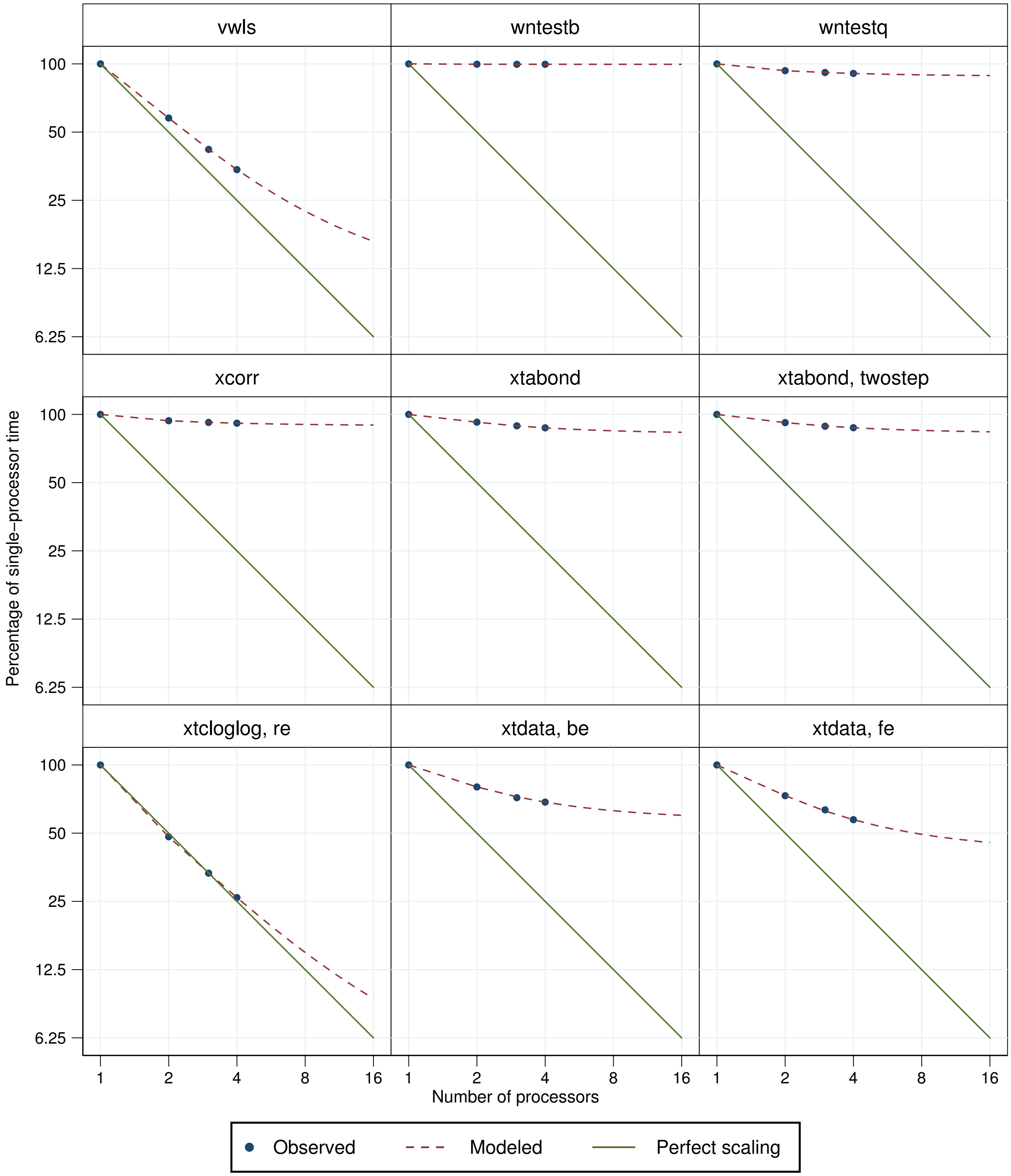

Figure B.32. **Parallelization performance plots.**

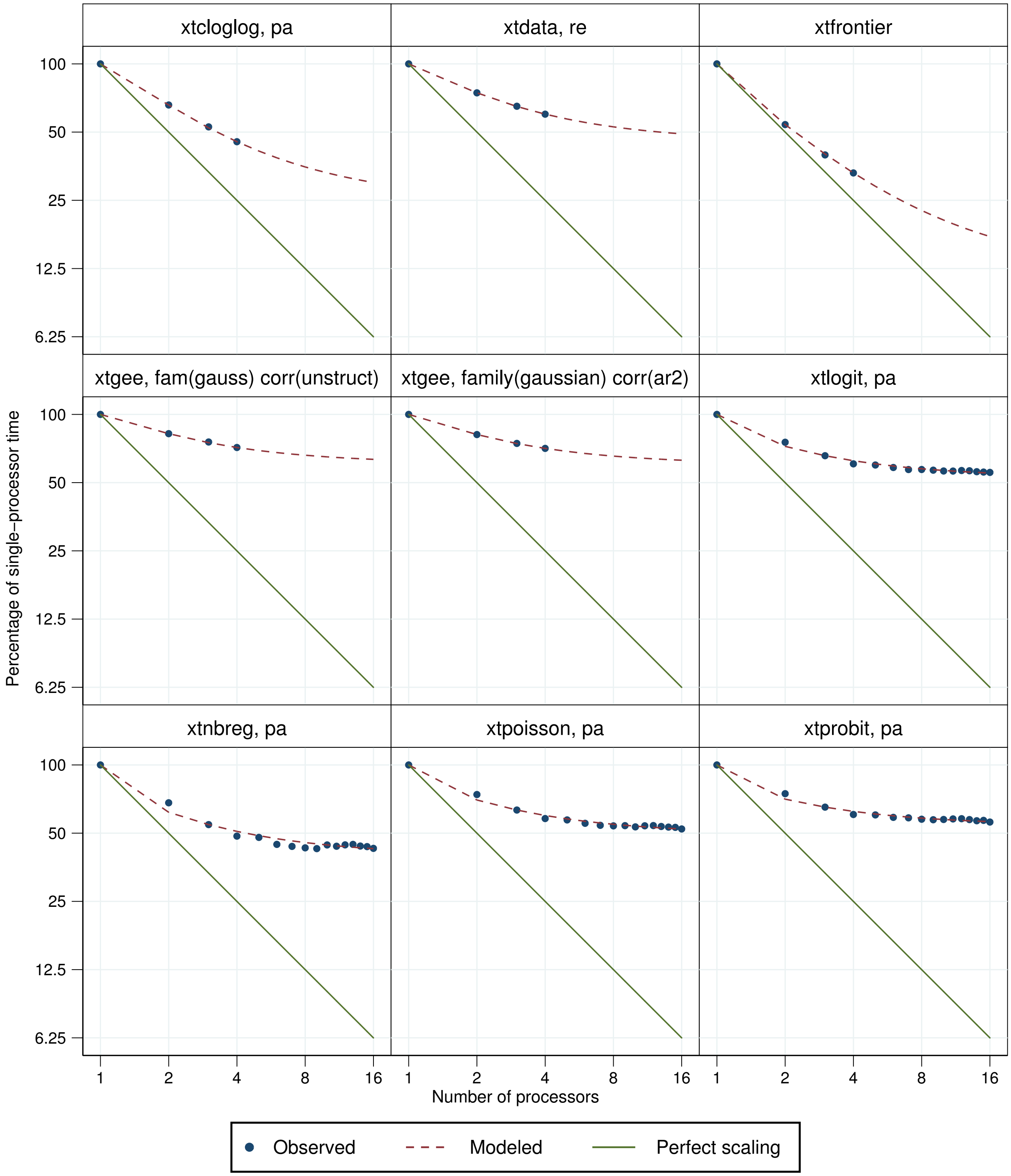

Figure B.33. **Parallelization performance plots.**

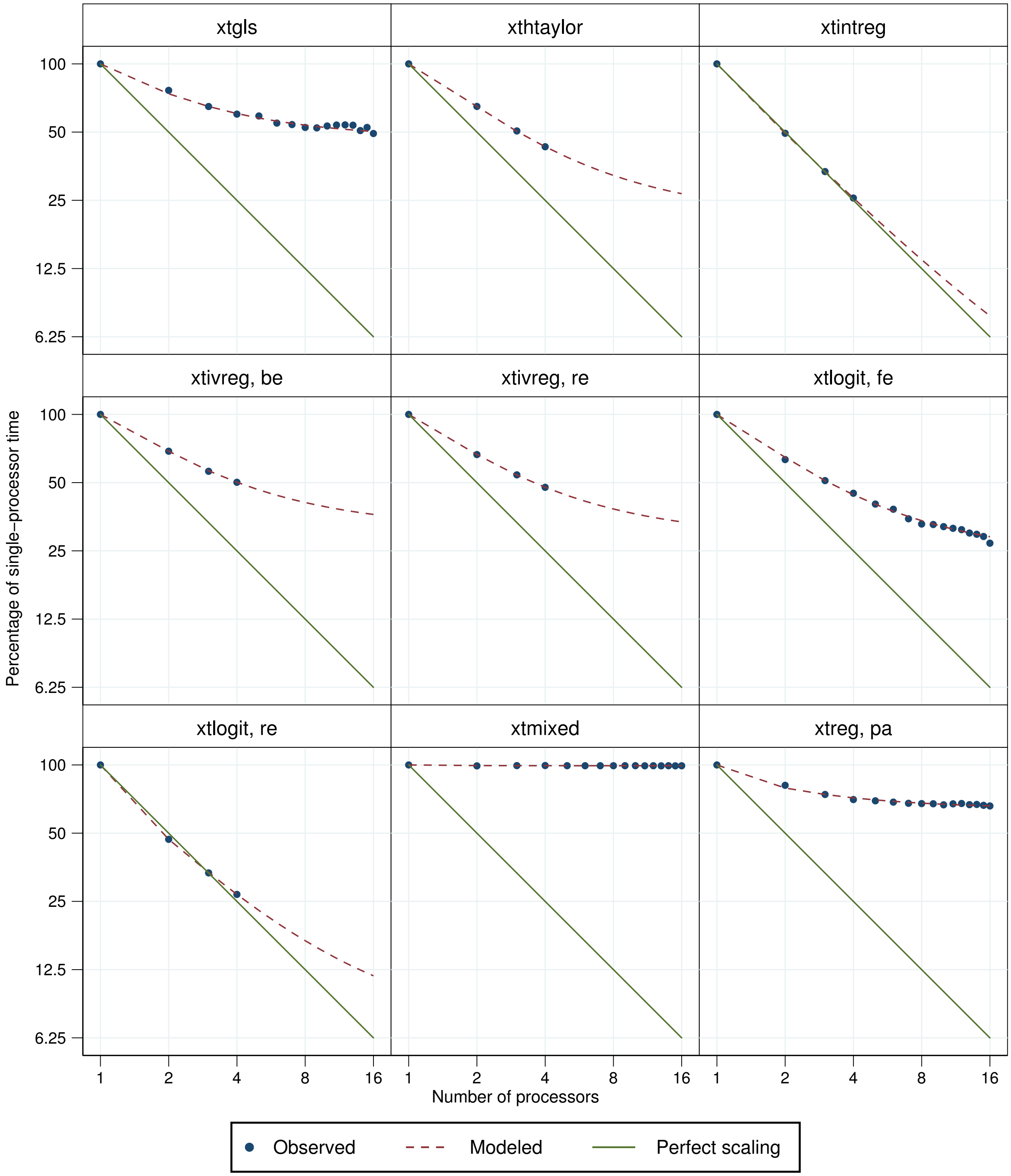

Figure B.34. **Parallelization performance plots.**

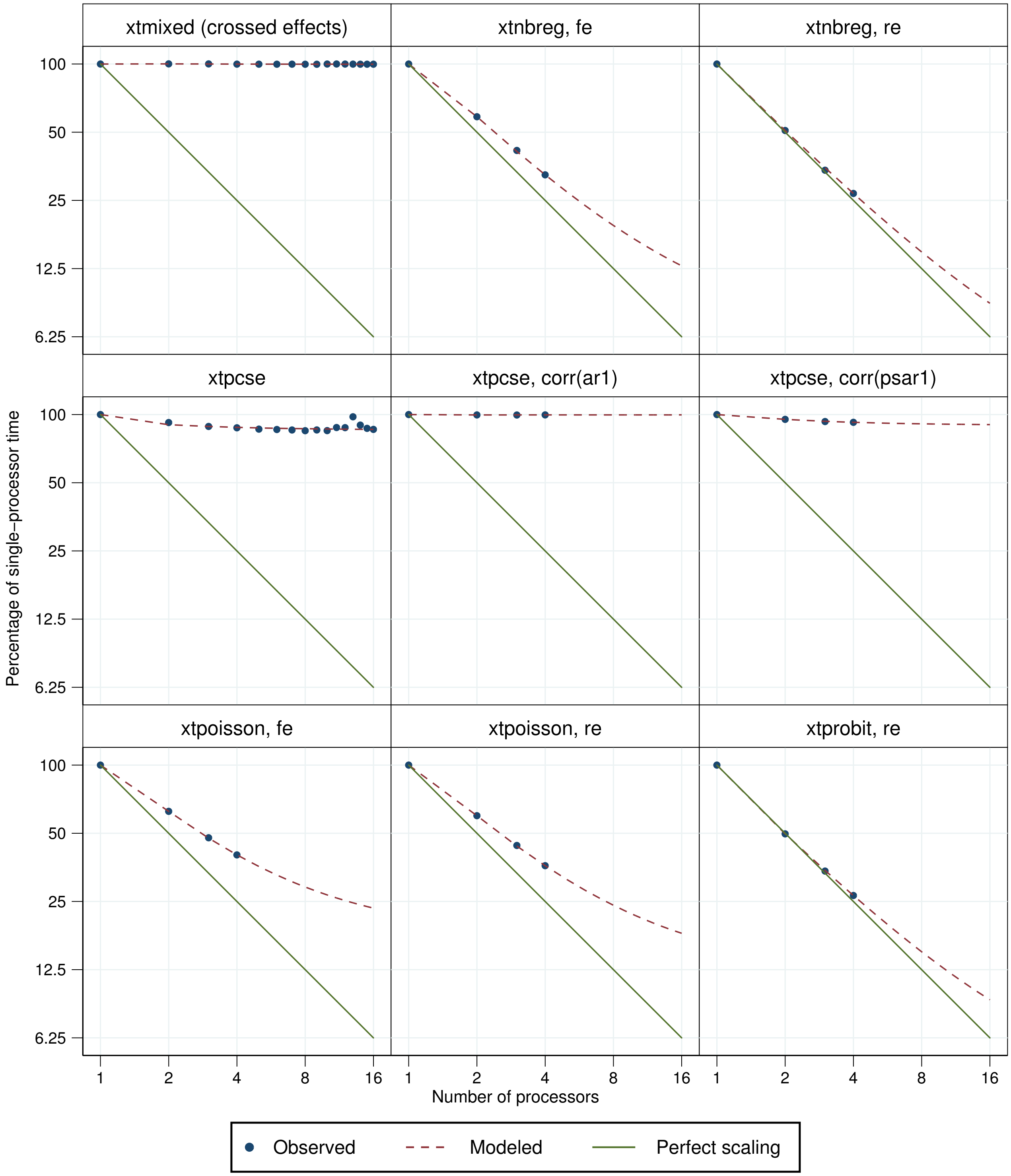

Figure B.35. **Parallelization performance plots.**

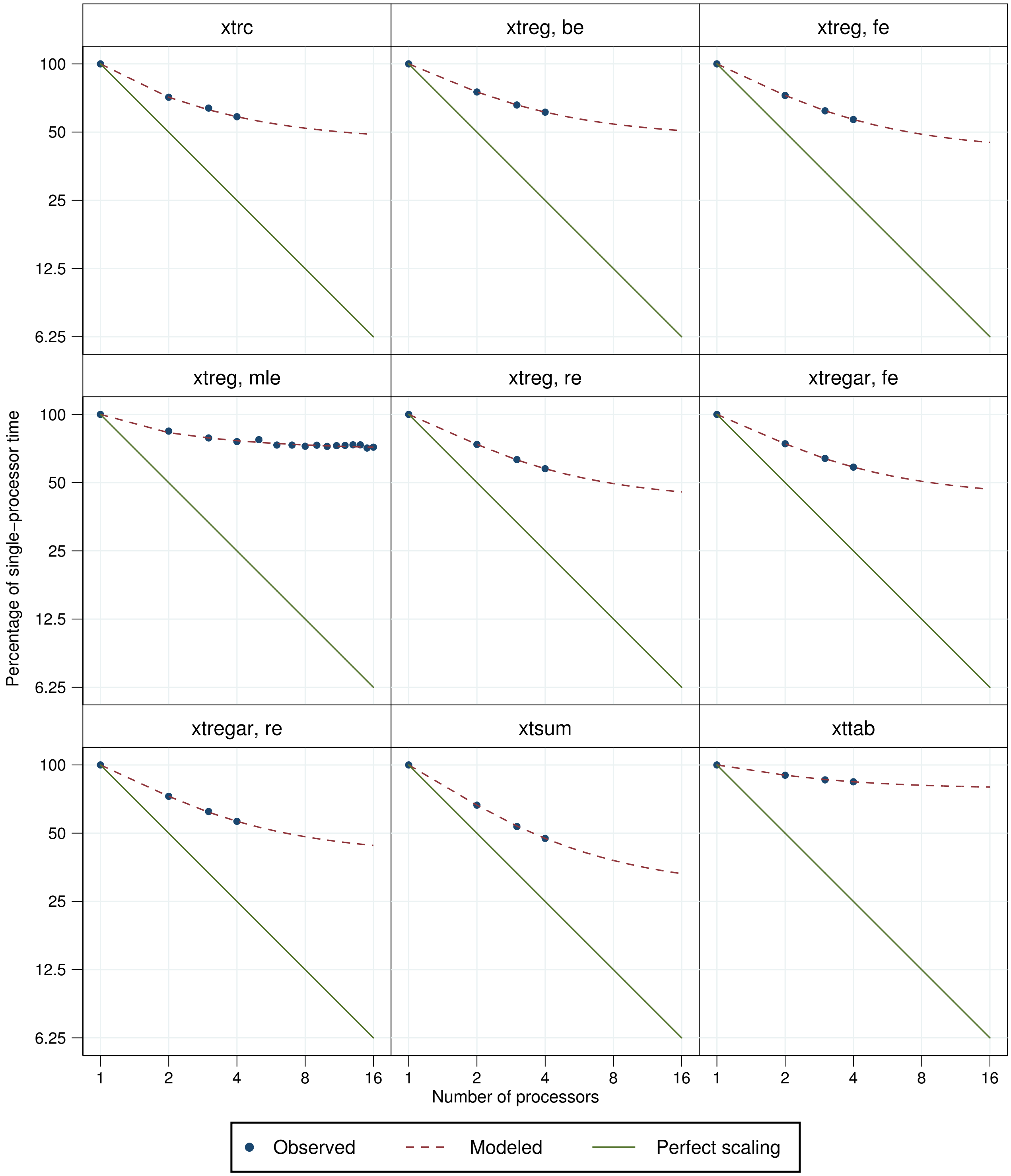

Figure B.36. **Parallelization performance plots.**

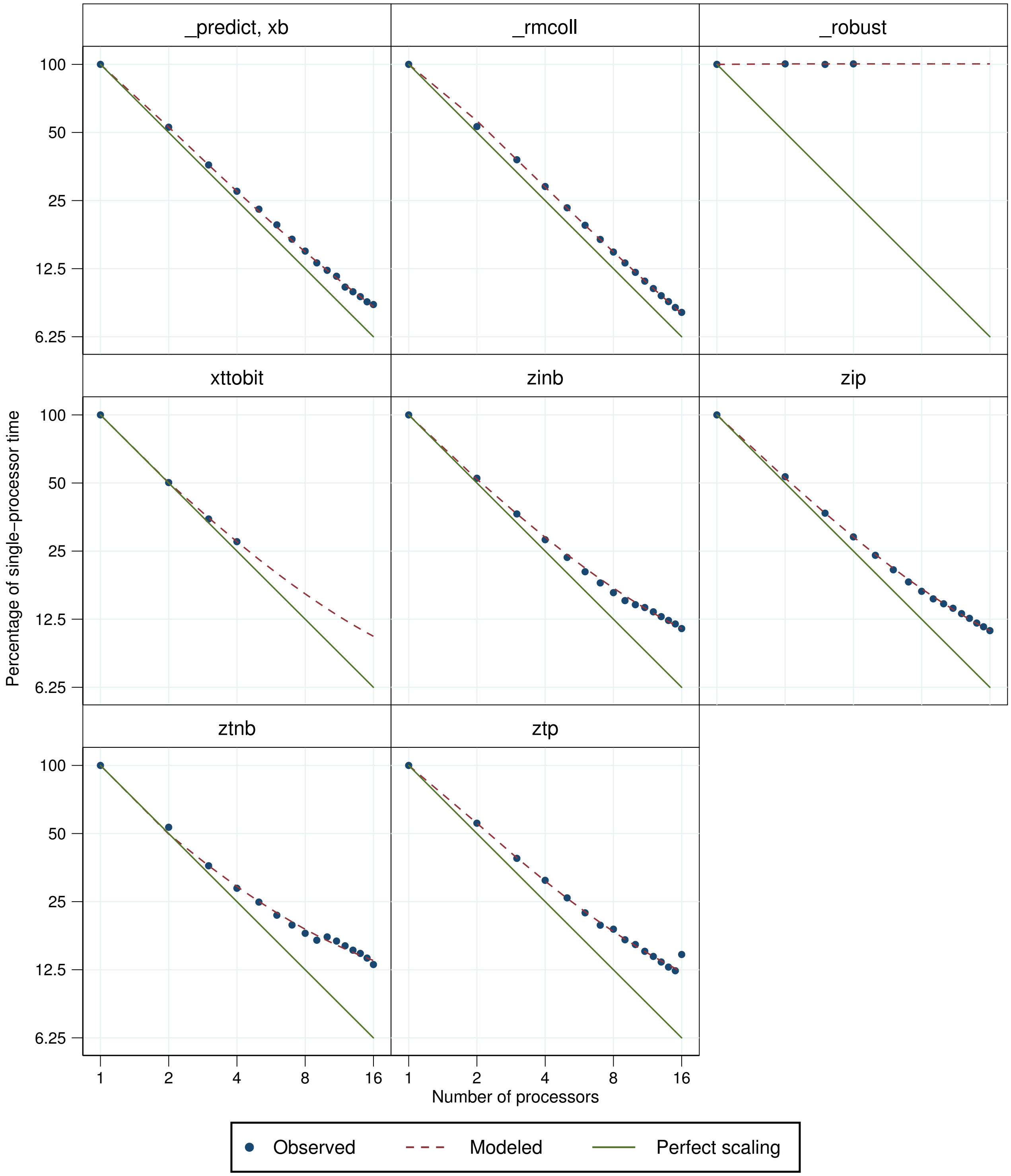

Figure B.37. **Parallelization performance plots.**

C Command names and descriptions

Table 2. Command descriptions

command	description
adjust	Tables of adjusted means and proportions
alpha	Cronbach's alpha
ameans	Arithmetic, geometric, and harmonic means
anova (oneway)	Analysis of variance and covariance—one-way
anova (twoway)	Analysis of variance and covariance—two-way
arch	Autoregressive conditional heteroskedasticity (ARCH) family of estimators
areg	Linear regression with a large dummy-variable set
arima	ARIMA, ARMAX, and other dynamic regression models
asmprobit	Maximum simulated-likelihood alternative-specific multinomial probit models
binreg	Generalized linear models: extensions to the binomial family
biplot	Biplots
biprobit	Bivariate probit regression
biprobit (seemingly unrelated)	Seemingly unrelated probit regression
bitest	Binomial probability test
blogit	Logistic regression for grouped data
boxcox	Box–Cox regression models
bprobit	Probit regression for grouped data
brier	Brier score decomposition
bsample	Sampling with replacement
by: generate	Create new variables over longitudinal/panel data
by: generate (small groups)	Create new variables over longitudinal/panel data, small panels
by: replace	Replace variable values over longitudinal/panel data
by: replace (small groups)	Replace variable values over longitudinal/panel data, small panels
ca	Simple correspondence analysis
canon	Canonical correlations

Table 2. Command descriptions

command	description
`centile`	Report centile and confidence interval
`ci`	Normal-based confidence intervals
`ci, binomial`	Binomial confidence intervals for proportions
`ci, poisson`	Poisson confidence intervals for counts
`clogit (k1 to k2 matching)`	Conditional (fixed-effects) logistic regression, k1 to k2 matching
`clogit (1 to k matching)`	Conditional (fixed-effects) logistic regression, 1 to k matching
`cloglog`	Complementary log-log regression
`cluster averagelinkage`	Hierarchical cluster analysis—average linkage
`cluster centroidlinkage`	Hierarchical cluster analysis—centroid linkage
`cluster completelinkage`	Hierarchical cluster analysis—complete linkage
`cluster generate`	Generate summary and grouping variables from a cluster analysis
`cluster kmeans`	Kmeans cluster analysis
`cluster kmedians`	Kmedians cluster analysis
`cluster medianlinkage`	Hierarchical cluster analysis—median linkage
`cluster singlelinkage`	Hierarchical cluster analysis—single linkage
`cluster wardslinkage`	Hierarchical cluster analysis—Ward's linkage
`cluster waveragelinkage`	Hierarchical cluster analysis—Ward's average linkage
`cnreg`	Censored-normal regression
`cnsreg`	Constrained linear regression
`collapse`	Make dataset of summary datasets
`compare`	Compare two variables
`compress`	Compress data in memory
`contract`	Make dataset of frequencies and percentages
`correlate`	Correlations (covariances) of variables or estimators
`corrgram`	Tabulate and graph autocorrelations

Table 2. Command descriptions

command	description
count	Count observations satisfying specified condition
ctset	Declare data to be count-time data
cttost	Convert count-time data to survival-time data
cumul	Cumulative distribution
cusum	Cusum plots and tests for binary variables
dfgls	DF-GLS unit-root test
dfuller	Augmented Dickey–Fuller unit-root test
dotplot	Comparative scatterplots
dstdize	Direct and indirect standardization
eivreg	Errors-in-variables regression
factor	Factor analysis
fcast compute	Dynamic forecasts after VAR or VEC estimation
fracpoly	Fractional polynomial regression
frontier	Stochastic frontier models
gen (small expressions)	Create or change contents of variable—small expressions
generate	Create or change contents of variable
glm, family(gamma)	Generalized linear models—gamma distribution
glm, family(gaussian)	Generalized linear models—Gaussian distribution
glm, family(igaussian)	Generalized linear models—inverse Gaussian distribution
glm, family(nbinomial)	Generalized linear models—negative binomial distribution
glm, family(poisson)	Generalized linear models—Poisson distribution
glogit	Weighted least-squares logistic regression for grouped data
gprobit	Weighted least-squares probit regression for grouped data
graph bar	Bar charts
graph box	Box plots

Table 2. Command descriptions

command	description
`graph pie`	Pie charts
`grmeanby`	Graph means and medians by categorical variables
`hausman`	Hausman specification test
`heckman`	Heckman selection model—maximum likelihood estimator
`heckman, twostep`	Heckman selection model—two-step estimator
`heckprob`	Probit model with selection
`hetprob`	Heteroskedastic probit model
`histogram`	Histograms for continuous and categorical variables
`hotelling`	Hotelling's T-squared generalized means test
`impute`	Fill in missing values
`intreg`	Interval regression
`irf create`	Create IRFs and FEVDs after VAR and VEC estimation
`ivprobit`	Probit model with endogenous regressors
`ivprobit, cluster()`	Probit model with endogenous regressors, cluster-robust standard errors
`ivprobit, robust`	Probit model with endogenous regressors, robust (Huber/White) standard errors
`ivreg`	Instrumental variables (two-stage least-squares) regression
`ivtobit`	Tobit model with endogenous regressors
`kap`	Interrater agreement
`kappa`	Interrater agreement
`kdensity`	Univariate kernel density estimation
`ksmirnov`	Kolmogorov–Smirnov equality-of-distributions test
`ksmirnov, by()`	Kolmogorov–Smirnov equality-of-distributions test over groups
`ktau`	Kendall's rank correlation coefficients
`kwallis`	Kruskal–Wallis equality-of-populations rank test
`ladder`	Ladder of powers

Table 2. Command descriptions

command	description
levelsof	Levels of variable
loadingplot	Score and loading plots after factor and pca
logistic	Logistic regression, reporting odds ratios
logit	Logistic regression, reporting coefficients
loneway	Large one-way ANOVA, random effects, and reliability
lowess	Lowess smoothing
ltable	Life tables for survival data
manova (oneway)	Multivariate analysis of variance and covariance, one-way
manova (twoway)	Multivariate analysis of variance and covariance, two-way
markout	Mark observations for exclusion
marksample	Mark observations for inclusion
marksample if exp	Mark observations for inclusion, with if qualifier
matrix accum	Form cross-product matrices of variables over observations
matrix eigenvalues	Eigenvalues of a matrix
matrix score	Inner product of matrix with variables over observations
matrix svd	Singular value decomposition
matrix symeigen	Eigenvalues of a symmetric matrix
matrix syminv	Inversion of a symmetric matrix
mds	Multidimensional scaling for two-way data
mdslong	Multidimensional scaling of proximity data in long format
mean	Estimate means
median	Equality tests on unmatched data
mfp	Multivariable fractional polynomial models
mfx	Obtain marginal effects or elasticities after estimation
mkmat	Convert variables to matrix and vice versa

Table 2. Command descriptions

command	description
mkspline	Linear spline construction
mleval	Helper commands for user-programmed MLEs, evaluates likelihood of coefficient vector
mleval, nocons	Helper commands for user-programmed MLEs, evaluates likelihood of coefficient vector without constant
mlmatbysum	Helper commands for user-programmed MLEs, computes Hessians of panel-data estimators
mlmatsum	Helper commands for user-programmed MLEs, computes Hessians of coefficient vector
mlogit	Multinomial (polytomous) logistic regression
mlsum	Helper commands for user-programmed MLEs, sums likelihood of coefficient vector
mlvecsum	Helper commands for user-programmed MLEs, computes gradients of coefficient vector
mprobit	Multinomial probit regression
mvreg	Multivariate regression
nbreg	Negative binomial regression
newey	Regression with Newey–West standard errors
nl	Nonlinear least-squares estimation
nlogit	Nested logit regression
nptrend	Test for trend across ordered groups
ologit	Ordered logistic regression
oneway	One-way analysis of variance
oprobit	Ordered probit regression
orthog	Orthogonalize variables and compute orthogonal polynomials
pca	Principal component analysis
pcorr	Partial correlation coefficients
pctile	Create variable containing percentiles
pergram	Periodogram
pkcollapse	Generate pharmacokinetic measurement dataset
pkexamine	Calculate pharmacokinetic measures

Table 2. Command descriptions

command	description
`pksumm`	Summarize pharmacokinetic data
`poisson`	Poisson regression
`pperron`	Phillips–Perron unit-root test
`prais`	Prais–Winsten and Cochrane–Orcutt regression
`predict, cooksd`	Obtain Cook's distance predictions after estimation
`predict, covratio`	Obtain COVRATIO predictions after estimation
`predict, dfbeta`	Obtain DFBETAs for a variable after estimation
`predict, dfits`	Obtain DFITS predictions after estimation
`predict, e`	Obtain predictions given upper and lower truncation after estimation
`predict, leverage`	Obtain leverage of observations after estimation
`predict, pr`	Obtain probability-in-range predictions after estimation
`predict, residuals`	Obtain residuals after estimation
`predict, rstandard`	Obtain standardized residuals after estimation
`predict, rstudent`	Obtain studentized residuals after estimation
`predict, stdf`	Obtain standard errors of predictions after estimation
`predict, stdp`	Obtain standard errors of forecasts after estimation
`predict, stdr`	Obtain standard errors of residuals after estimation
`predict, welsch`	Obtain Welsch distances after estimation
`predict, ystar`	Obtain truncated predictions in a range after estimation
`predictnl`	Obtain nonlinear predictions, standard errors, etc., after estimation
`probit`	Probit regression
`procrustes`	Procrustes transformation
`proportion`	Estimate proportions
`prtest1`	One-sample tests of proportions
`prtest2`	Two-sample tests of proportions

Table 2. Command descriptions

command	description
`prtest, by()`	Tests of proportions computed over groups
`qreg`	Quantile (including median) regression
`ranksum`	Equality tests on unmatched data
`ratio`	Estimate ratio with SE and CI
`ratio (exp1) (exp2)`	Estimate two ratios with SE and CI
`recode`	Recode categorical variables
`reg3`	Three-stage estimation for systems of simultaneous equations
`regress`	Linear regression
`regress, cluster()`	Linear regression, cluster-robust standard errors
`regress, robust`	Linear regression, robust (Huber/White) standard errors
`replace`	Create or change contents of variable
`replace (small expression)`	Create or change contents of variable, simple expression
`reshape long`	Convert data from wide to long
`reshape wide`	Convert data from long to wide
`robvar`	Robust tests for equality of variance
`rocfit`	Fit ROC models
`roctab`	Receiver-Operating-Characteristic (ROC) analysis
`rotatemat`	Orthogonal and oblique rotations of a Stata matrix
`rreg`	Robust regression
`runtest`	Test for random order
`scobit`	Skewed logistic regression
`scoreplot`	Score and loading plots after cmd:factor and cmd:pca
`screeplot`	Scree plot of eigenvalues
`sdtest1`	Variance-comparison test against constant
`sdtest2`	Variance-comparison test between variables

Table 2. Command descriptions

command	description
sdtest, by()	Variance-comparison test over groups
sfrancia	Shapiro–Francia test for normality
signrank	Equality tests on matched data
signtest	Equality tests on matched data
sktest	Skewness and kurtosis test for normality
slogit	Stereotype logistic regression
sort	Sort data
spearman	Spearman's rank correlation coefficients
stack	Stack data
stbase	Form baseline dataset
stci	Confidence intervals for means and percentiles of survival time
stcox	Fit Cox proportional hazards model
stcurve, hazard (after stcox)	Compute and plot hazard after Cox proportional hazards estimation
stcurve, hazard (after streg)	Compute and plot hazard after survival estimation using exponential model
stgen	Generate variables reflecting entire histories
stir	Report incidence-rate comparison
stptime	Calculate person-time, incidence rates, and SMR
strate	Tabulate failure rates and rate ratios
streg, distribution(exponential)	Fit parametric survival models, exponential distribution
streg, dist(exp) cluster()	Fit parametric survival models, exponential distribution with cluster-robust standard errors
streg, dist(exp) frailty()	Fit parametric survival models, exponential distribution with individual frailty
streg, dist(exp) frailty() shared()	Fit parametric survival models, exponential distribution with shared frailty
streg, dist(exp) robust	Fit parametric survival models, exponential distribution with robust standard errors
streg, distribution(gamma)	Fit parametric survival models, gamma distribution
streg, distribution(lnormal)	Fit parametric survival models, log-normal distribution

Table 2. Command descriptions

command	description
`streg, distribution(weibull)`	Fit parametric survival models, Weibull distribution
`streg, dist(weibull) frailty()`	Fit parametric survival models, Weibull distribution with individual frailty
`streg, dist(weibull) frailty() shared()`	Fit parametric survival models, Weibull distribution with shared frailty
`sts generate`	Create new variables containing survival, hazard, and related functions
`sts graph`	Compute and graph survival, hazard, and related functions
`sts list`	Compute and list survival and related functions
`sts test`	Test the equality of the survival function across groups
`stset`	Declare data to be survival-time data
`stsplit`	Split time-span records
`stsum`	Summarize survival-time data
`stvary`	Report variables that vary over time
`summarize`	Summary statistics
`sunflower`	Density-distribution sunflower plots
`sureg`	Zellner's seemingly unrelated regression
`svar`	Structural vector autoregression models
`svmat`	Convert variables to matrix and vice versa
`svy: logit`	Logistic/logit regression using survey data
`svy: poisson`	Poisson regression using count survey data
`svy: regress`	Linear regression using survey data
`swilk`	Shapiro–Wilk test for normality
`symmetry`	Symmetry and marginal homogeneity tests
`table (oneway)`	Table of summary statistics, one-way
`table (twoway)`	Table of summary statistics, two-way
`tabstat`	Display table of summary statistics
`tabstat, by()`	Display table of summary statistics over groups

Table 2. Command descriptions

command	description
tabulate (oneway)	Tables of frequencies, one-way
tabulate (twoway)	Tables of frequencies, two-way
tetrachoric	Tetrachoric correlations for binary variables
tobit	Tobit regression
total	Estimate totals
treatreg	Treatment-effects model, ML estimation
treatreg, twostep	Treatment-effects model, two-step estimation
truncreg	Truncated regression
tsset	Declare a dataset to be time-series data
tssmooth exp	Exponential smoothing of univariate time-series data
tssmooth ma	Moving average smoothing of univariate time-series data
ttest1	Mean comparison test against constant null hypothesis
ttest2	Mean comparison test against between variables
ttest, by()	Mean comparison test against over groups
twoway fpfit	Compute and graph fractional-polynomial fit
twoway lfitci	Compute and graph linear fit with confidence intervals
twoway mband	Compute and graph median bands
twoway mspline	Compute and graph spline smooth
var	Vector autoregression models
vargranger	Perform pairwise Granger causality tests after var or svar
varlmar	Obtain LM statistics for residual autocorrelation after var or svar
varnorm	Test for normally distributed disturbances after var or svar
varsoc	Obtain lag-order selection statistics for VARs and VECMs
varstable	Check the stability condition of VAR or SVAR estimates
vec	Vector error-correction models

Table 2. Command descriptions

command	description
veclmar	Obtain LM statistics for residual autocorrelation after vec
vecnorm	Test for normally distributed disturbances after vec
vecrank	Estimate the cointegrating rank using Johansen's framework
vecstable	Check the stability condition of VECM estimates
vwls	Variance-weighted least squares
wntestb	Bartlett's periodogram-based test for white noise
wntestq	Portmanteau (Q) test for white noise
xcorr	Cross-correlogram for bivariate time series
xtabond	Arellano–Bond linear, dynamic panel-data estimation
xtabond, twostep	Arellano–Bond linear, dynamic panel-data estimation, two-step estimation
xtcloglog, re	Random-effects cloglog models
xtdata, be	Compute between transform of panel data
xtdata, fe	Compute within (fixed-effects) transform of panel data
xtdata, re	Compute random-effects transform of panel data
xtfrontier	Stochastic frontier models for panel data
xtgee, family(gaussian) corr(ar2)	GEE estimation of Gaussian panel-data model with 2-period autocorrelation
xtgee, fam(gauss) corr(unstruct)	GEE estimation of Gaussian panel-data model with unstructured correlation
xtcloglog, pa	Population-averaged cloglog models
xtlogit, pa	Population-averaged logit models
xtnbreg, pa	Population-averaged negative binomial models
xtpoisson, pa	Population-averaged Poisson models
xtprobit, pa	Population-averaged probit models
xtreg, pa	Population-averaged linear model
xtgls	Fit panel-data models using GLS
xthtaylor	Hausman–Taylor estimator for error-components models

Table 2. Command descriptions

command	description
`xtintreg`	Random-effects interval data regression models
`xtivreg, be`	Instrumental variables and two-stage least squares for panel-data models—between effects
`xtivreg, re`	Instrumental variables and two-stage least squares for panel-data models—random effects
`xtlogit, fe`	Fixed-effects logit models
`xtlogit, re`	Random-effects logit models
`xtmixed`	Multilevel mixed-effects linear regression
`xtmixed (crossed effects)`	Multilevel mixed-effects linear regression—crossed effects
`xtnbreg, fe`	Fixed-effects negative binomial models
`xtnbreg, re`	Random-effects negative binomial models
`xtpcse`	OLS or Prais–Winsten models with panel-corrected standard errors
`xtpcse, corr(ar1)`	Prais–Winsten models with panel-corrected standard errors
`xtpcse, corr(psar1)`	Prais–Winsten models with panel-corrected standard errors—panel-specific autocorrelation
`xtpoisson, fe`	Fixed-effects Poisson models
`xtpoisson, re`	Random-effects Poisson models
`xtprobit, re`	Random-effects probit models
`xtrc`	Random-coefficients regression
`xtreg, be`	Between-effects linear models
`xtreg, fe`	Fixed-effects linear models
`xtreg, mle`	Random-effects linear models, ML estimation
`xtreg, re`	Random-effects linear models
`xtregar, fe`	Fixed-effects linear models with an AR(1) disturbance
`xtregar, re`	Random-effects linear models with an AR(1) disturbance
`xtsum`	Summarize xt data
`xttab`	Tabulate xt data
`xttobit`	Random-effects tobit models

Table 2. Command descriptions

command	description
`zinb`	Zero-inflated negative binomial regression
`zip`	Zero-inflated Poisson regression
`ztnb`	Zero-truncated negative binomial regression
`ztp`	Zero-truncated Poisson regression
`_predict, xb`	Obtain predictions, residuals, etc., after estimation programming command—option `xb`
`_rmcoll`	Remove collinear variables
`_robust`	Robust variance estimates

D Problem sizes

The following table shows the size of the problems used to measure the performance gains reported in table 1. As discussed in section 9, these are intentionally large problems requiring some time to run. If a command was so fast that a sufficiently large problem would have required too much memory to be run on a variety of computers, then a smaller problem was run, and the problem was run several times (iterations) for the timing.

The second though fourth columns of the table record the number of observations for the problem, either as a simple number of observations N or as a number of panels m and a number of time periods t within panel. The latter provide more information on problem size for longitudinal panel-data problems, and the number of observation is just the product of m and t. Some such problems are not really panel data but merely grouped data, and the time periods should just be considered the number of observations within group. Almost all the panel-data problems were created with balanced panels (equal number of observations within panel). Rarely would unbalanced panels affect the performance gains of Stata/MP.

The column labeled k records the number of covariates in the problem, or, for matrix commands, the row and column dimensions of the matrix.

The column labeled n_{eq} records the number of equations for problems that involve multiple equations.

The column n_{iter} records the number of times the command was run on the problem to generate a single timing.

Table 3. Problem sizes

command	Observations N	Observations m	Observations t	k	n_{eq}	n_{iter}
`adjust`	20000			100		1
`alpha`	500000			20		1
`ameans`	300000			20		1
`anova (oneway)`	80000			500		1
`anova (twoway)`	300000			26		1
`arch`	20000			5		1
`areg`	30000			200	100	1
`arima`	2000			15		1
`asmprobit`		200	3	2	2	1
`binreg`	50000			100		1
`biplot`	4000			2		1
`biprobit`	20000			40	40	1
`biprobit (seemingly unrelated)`	40000			40	40	1
`bitest`	3000000			1	2	10
`blogit`	10000			20	50	50
`boxcox`	30000			50		1
`bprobit`	10000			20	50	50
`brier`	150000					5
`bsample`	100000			100		20
`by: generate`		5000	500			6
`by: generate (small groups)`		250000	10			6
`by: replace`		5000	500			6
`by: replace (small groups)`		250000	10			6
`ca`	50000				250	1
`canon`	1000000				30	1

N, number of observations; m, number of panels; t, number of time periods within each panel; k, number of regressors; n_{eq}, number of equations; and n_{iter}, number of iterations.

Table 3. Problem sizes

command	Observations N	m	t	k	n_{eq}	n_{iter}
`centile`	10000			300		1
`ci`	500000			50		1
`ci, binomial`	500000			50		1
`ci, poisson`	100000			20		8
`clogit (k1 to k2 matching)`		20000	10	3		1
`clogit (1 to k matching)`		15000	5	50		1
`cloglog`	40000			100		1
`cluster averagelinkage`	1200			5		1
`cluster centroidlinkage`	1200			5		1
`cluster completelinkage`	1200			5		1
`cluster generate`	1000			5		1
`cluster kmeans`	10000			10		1
`cluster kmedians`	10000			5		1
`cluster medianlinkage`	1200			5		1
`cluster singlelinkage`	5000			5		1
`cluster wardslinkage`	1200			5		1
`cluster waveragelinkage`	1200			5		1
`cnreg`	500000			20		1
`cnsreg`	500000			100		1
`collapse`	300000			50	3	1
`compare`	500000			2		10
`compress`	500000			50	50	1
`contract`	1000000			20	100	1
`correlate`	200000			200		1
`corrgram`	40000			1		1

N, number of observations; m, number of panels; t, number of time periods within each panel; k, number of regressors; n_{eq}, number of equations; and n_{iter}, number of iterations.

Table 3. Problem sizes

command	Observations N	m	t	k	n_{eq}	n_{iter}
`count`	5000000					1
`ctset`	1000000					50
`cttost`	500000					1
`cumul`	1000000			2		1
`cusum`	500000			1		1
`dfgls`	2000			1		40
`dfuller`	500000			1		10
`dotplot`	100000			10		1
`dstdize`		150	50	1000		1
`eivreg`	160000			200		1
`factor`	120000			200		1
`fcast compute`	30000			2	5	1
`fracpoly`	500000			10		1
`frontier`	30000			200		1
`gen (small expressions)`	15000			4000		1
`generate`	50000					25
`glm, family(gamma)`	200000			50		1
`glm, family(gaussian)`	400000			50		1
`glm, family(igaussian)`	100000			100		1
`glm, family(nbinomial)`	100000			50		1
`glm, family(poisson)`	100000			50		1
`glogit`	10000			20	50	100
`gprobit`	20000			40	50	50
`graph bar`	500000			10	3	1
`graph box`	200000			2	10	1

N, number of observations; m, number of panels; t, number of time periods within each panel; k, number of regressors; n_{eq}, number of equations; and n_{iter}, number of iterations.

Table 3. Problem sizes

	Observations					
command	N	m	t	k	n_{eq}	n_{iter}
`graph pie`	2500000			10	10	1
`grmeanby`	300000			4	10	1
`hausman`	200					1
`heckman`	50000			100	50	1
`heckman, twostep`	100000			100	50	1
`heckprob`	20000			50	50	1
`hetprob`	40000			50	20	1
`histogram`	3000000			1		1
`hotelling`	150000			100		1
`impute`	400000			30		1
`intreg`	50000			50		1
`irf create`	100000			2	8	1
`ivprobit`	15000			30	20	1
`ivprobit, cluster()`	15000			30	20	1
`ivprobit, robust`	15000			30	20	1
`ivreg`	80000			200	100	1
`ivtobit`	10000			50	20	1
`kap`	500000			2	10	4
`kappa`	400000			10	20	1
`kdensity`	1000000					1
`ksmirnov`	1000000					1
`ksmirnov, by()`	2000000					1
`ktau`	5000			5		1
`kwallis`	800000			10		1
`ladder`	400000					1

N, number of observations; m, number of panels; t, number of time periods within each panel; k, number of regressors; n_{eq}, number of equations; and n_{iter}, number of iterations.

Table 3. Problem sizes

command	Observations N	m	t	k	n_{eq}	n_{iter}
`levelsof`	200000			500		10
`loadingplot`	400000			60		4
`logistic`	100000			100		1
`logit`	100000			100		1
`loneway`	500000			500		4
`lowess`	10000			1		1
`ltable`	50000			1		40
`manova (oneway)`	1000000			200	20	1
`manova (twoway)`	1000000			20	10	1
`markout`	100000			500		1
`marksample`	100000			500		1
`marksample if exp`	100000			500		1
`matrix accum`	100000			300		1
`matrix eigenvalues`	1000			1000		1
`matrix score`	100000			1000		1
`matrix svd`	400			400		1
`matrix symeigen`	800			800		1
`matrix syminv`	1300			1300		1
`mds`	400			400		1
`mdslong`		400	1			1
`mean`	100000			100		1
`median`	100000			5		40
`mfp`	30000			5		1
`mfx`	40000			200		1
`mkmat`	600			600		1

N, number of observations; m, number of panels; t, number of time periods within each panel; k, number of regressors; n_{eq}, number of equations; and n_{iter}, number of iterations.

Table 3. Problem sizes

command	Observations N	m	t	k	n_{eq}	n_{iter}
`mkspline`	4000000			1		1
`mleval`	200000			200		10
`mleval, nocons`	200000			200		10
`mlmatbysum`	200000			200	10	10
`mlmatsum`	60000			400		1
`mlogit`	10000			100	3	1
`mlsum`	1000000			1		100
`mlvecsum`	300000			200		10
`mprobit`	800			10	3	1
`mvreg`	400000			100	4	1
`nbreg`	60000			30		1
`newey`	500000			5		1
`nl`	300000					1
`nlogit`		1200	2	5	3	1
`nptrend`	300000			10		1
`ologit`	70000			100	3	1
`oneway`	1000000			300		30
`oprobit`	70000			100	3	1
`orthog`	200000			10		1
`pca`	300000			100		1
`pcorr`	150000			200		1
`pctile`	2000000			1		1
`pergram`	15000			1		1
`pkcollapse`		100	50			1
`pkexamine`		1	25			300

N, number of observations; m, number of panels; t, number of time periods within each panel; k, number of regressors; n_{eq}, number of equations; and n_{iter}, number of iterations.

Table 3. Problem sizes

command	Observations N	m	t	k	n_{eq}	n_{iter}
`pksumm`		200	10			1
`poisson`	80000			80		1
`pperron`	500000			1		1
`prais`	500000			5		1
`predict, cooksd`	30000			300		1
`predict, covratio`	30000			300		1
`predict, dfbeta`	30000			200		1
`predict, dfits`	30000			200		1
`predict, e`	20000			1000		1
`predict, leverage`	80000			200		1
`predict, pr`	20000			1000		1
`predict, residuals`	20000			1000		1
`predict, rstandard`	20000			400		1
`predict, rstudent`	20000			400		1
`predict, stdf`	20000			400		1
`predict, stdp`	20000			400		1
`predict, stdr`	20000			400		1
`predict, welsch`	20000			300		1
`predict, ystar`	20000			1000		1
`predictnl`	40000			30		1
`probit`	100000			100		1
`procrustes`	200000			20	20	1
`proportion`	100000			10	5	1
`prtest1`	1000000			1	2	20
`prtest2`	1000000			2	2	15

N, number of observations; m, number of panels; t, number of time periods within each panel; k, number of regressors; n_{eq}, number of equations; and n_{iter}, number of iterations.

Table 3. Problem sizes

command	Observations N	m	t	k	n_{eq}	n_{iter}
`prtest, by()`	1000000			2	2	15
`qreg`	100000			20		1
`ranksum`	2000000			2		1
`ratio`	1000000					8
`ratio (exp1) (exp2)`	1000000					5
`recode`	500000			5	5	1
`reg3`	30000			50	3	1
`regress`	40000			400		1
`regress, cluster()`	40000			400		1
`regress, robust`	40000			400		1
`replace`	50000					25
`replace (small expression)`	10000			4000		1
`reshape long`		100000	20			1
`reshape wide`		100000	15	5		1
`robvar`	300000			2		1
`rocfit`	3000			1	5	8
`roctab`	500000			1	80	1
`rotatemat`	100			100		1
`rreg`	50000			100		1
`runtest`	1000000			1		2
`scobit`	80000			50		1
`scoreplot`	400000			20		1
`screeplot`	400000			20		20
`sdtest1`	1500000					30
`sdtest2`	1500000			2		15

N, number of observations; m, number of panels; t, number of time periods within each panel; k, number of regressors; n_{eq}, number of equations; and n_{iter}, number of iterations.

Table 3. Problem sizes

command	Observations N	m	t	k	n_{eq}	n_{iter}
`sdtest, by()`	1500000					8
`sfrancia`	300000			10		1
`signrank`	2000000			2		1
`signtest`	2000000			2		20
`sktest`	2000000			2		1
`slogit`	5000			50	5	1
`sort`	500000			100	50	1
`spearman`	200000			3		1
`stack`	4000			4000		1
`stbase`	100000			200		1
`stci`	20000			1		12
`stcox`	50000			10		10
`stcurve, hazard (after stcox)`	100000			2		1
`stcurve, hazard (after streg)`	100000			2		1
`stgen`	1000000			2		1
`stir`	1000000			1	2	5
`stptime`	50000			1	500	200
`strate`	1000000			1	20	1
`streg, distribution(exponential)`	40000			100		1
`streg, dist(exp) cluster()`	15000			100	30	1
`streg, dist(exp) frailty()`	15000			100		1
`streg, dist(exp) frailty() shared()`	15000			100	30	1
`streg, dist(exp) robust`	40000			100		1
`streg, distribution(gamma)`	10000			2		1
`streg, distribution(lnormal)`	40000			30		1

N, number of observations; m, number of panels; t, number of time periods within each panel; k, number of regressors; n_{eq}, number of equations; and n_{iter}, number of iterations.

Table 3. Problem sizes

command	Observations					
	N	m	t	k	n_{eq}	n_{iter}
`streg, distribution(weibull)`	200000			30		1
`streg, dist(weibull) frailty()`	20000			50		1
`streg, dist(weibull) frailty() shared()`	10000			100	30	1
`sts generate`	1000000			1		1
`sts graph`	1000000			1		1
`sts list`	100000			1		10
`sts test`	1000000			1	2	1
`stset`	2000000					1
`stsplit`	1000000				50	1
`stsum`	500000			1		1
`stvary`	3000000			5		1
`summarize`	400000			100		1
`sunflower`	1000000			2		1
`sureg`	100000			50	2	1
`svar`	20000			2	10	1
`svmat`	3000			3000		1
`svy: logit`	500000			10		1
`svy: poisson`	200000			10		1
`svy: regress`	500000			10		1
`swilk`	150000			20		1
`symmetry`	800000			2	50	1
`table (oneway)`	2000000			20		2
`table (twoway)`	3000000			20		1
`tabstat`	1000000			1		50
`tabstat, by()`	200000			20		10

N, number of observations; m, number of panels; t, number of time periods within each panel; k, number of regressors; n_{eq}, number of equations; and n_{iter}, number of iterations.

Table 3. Problem sizes

command	Observations N	m	t	k	n_{eq}	n_{iter}
`tabulate (oneway)`	2000000			20		30
`tabulate (twoway)`	3000000			20		10
`tetrachoric`	200000			20	2	1
`tobit`	200000			50		1
`total`	400000			50		1
`treatreg`	30000			30	30	1
`treatreg, twostep`	150000			50	50	1
`truncreg`	30000			100		1
`tsset`	4000000					1
`tssmooth exp`	1000000			1		1
`tssmooth ma`	1000000			1		1
`ttest1`	1000000			1		50
`ttest2`	1000000			2		10
`ttest, by()`	1000000					10
`twoway fpfit`	200000			1		1
`twoway lfitci`	7500			1		1
`twoway mband`	1000000			1		1
`twoway mspline`	1000000			1		1
`var`	250000			2	5	1
`vargranger`	1000000			2	5	40
`varlmar`	80000			2	5	1
`varnorm`	300000			2	5	1
`varsoc`	150000			2	5	1
`varstable`	100000			2	10	15
`vec`	30000			2	10	1

N, number of observations; m, number of panels; t, number of time periods within each panel; k, number of regressors; n_{eq}, number of equations; and n_{iter}, number of iterations.

Table 3. Problem sizes

command	Observations N	m	t	k	n_{eq}	n_{iter}
veclmar	50000			2	5	1
vecnorm	150000			2	5	1
vecrank	200000			2	5	1
vecstable	100000			2	10	120
vwls	1000000			40		1
wntestb	15000			1		1
wntestq	400000			1		1
xcorr	400000			1		1
xtabond		10000	10	2		1
xtabond, twostep		15000	10	2		1
xtcloglog, re		4000	5	5		1
xtdata, be		100000	10	5		1
xtdata, fe		200000	5	5		1
xtdata, re		200000	5	5		1
xtfrontier		2000	10	5		1
xtgee, family(gaussian) corr(ar2)		50000	5	10		1
xtgee, fam(gauss) corr(unstruct)		50000	5	10		1
xtcloglog, pa		30000	5	5		1
xtlogit, pa		40000	5	5		1
xtnbreg, pa		15000	5	5		1
xtpoisson, pa		20000	10	5		1
xtprobit, pa		30000	10	5		1
xtreg, pa		50000	5	10		1
xtgls		5	100000	5		1
xthtaylor		40000	10	4	4	1

N, number of observations; m, number of panels; t, number of time periods within each panel; k, number of regressors; n_{eq}, number of equations; and n_{iter}, number of iterations.

Table 3. Problem sizes

command	Observations					
	N	m	t	k	n_{eq}	n_{iter}
`xtintreg`		1000	5	5		1
`xtivreg, be`		80000	5	5	5	1
`xtivreg, re`		30000	10	5	5	1
`xtlogit, fe`		6000	10	50		1
`xtlogit, re`		4000	10	5		1
`xtmixed`		500	10	5	5	1
`xtmixed (crossed effects)`		10	1500			1
`xtnbreg, fe`		20000	5	5		1
`xtnbreg, re`		7500	5	5		1
`xtpcse`		5	100000	5		1
`xtpcse, corr(ar1)`		50	500	5		1
`xtpcse, corr(psar1)`		20	5000	5		1
`xtpoisson, fe`		75000	5	5		1
`xtpoisson, re`		40000	5	5		1
`xtprobit, re`		2000	5	5		1
`xtrc`		100	2000	5		1
`xtreg, be`		150000	10	5		1
`xtreg, fe`		80000	10	5		1
`xtreg, mle`		80000	10	5		1
`xtreg, re`		50000	10	5		1
`xtregar, fe`		10000	20	5		1
`xtregar, re`		10000	20	5		1
`xtsum`		50000	10	10		1
`xttab`	1500000			2	50	1
`xttobit`		5000	5	5		1

N, number of observations; m, number of panels; t, number of time periods within each panel; k, number of regressors; n_{eq}, number of equations; and n_{iter}, number of iterations.

Table 3. Problem sizes

command	Observations N	m	t	k	n_{eq}	n_{iter}
`zinb`	15000			50	50	1
`zip`	25000			50	50	1
`ztnb`	50000			10		1
`ztp`	150000			50		1
`_predict, xb`	20000			1000		1
`_rmcoll`	50000			400		1
`_robust`	200000			200		1

N, number of observations; m, number of panels; t, number of time periods within each panel; k, number of regressors; n_{eq}, number of equations; and n_{iter}, number of iterations.

E GLLAMM

The table below shows results for a few models fitted using gllamm. This is but a small subset of the models that gllamm can fit. Each command is described briefly in the ensuing table.

The user-written command gllamm (generalized linear latent and mixed models) adds to Stata the ability to fit multilevel, mixed, or hierarchical regression models that have continuous, count, binary, or ordinal dependent variables, and it may have latent (unobserved) variables, endogenous covariates, and random coefficients or intercepts at any level. Among the many models that gllamm can fit, some important special cases include generalized linear mixed models, multilevel regression models, factor models, item response models, structural equation models, latent-class models, generalized linear models with covariate measurement error, endogenous switching and sample selection models, Rasch models (including multidimensional marginally sufficient Rasch models). All these models can be fitted with continuous, count, binary, or ordinal dependent variables or latent variables. gllamm's authors, Sophia Rabe-Hesketh with contributions from Anders Skrondal and Andrew Pickles, maintain a web site—http://www.gllamm.org/—with complete documentation (140 pages), tutorials, worked examples, wrapper commands to ease estimation of special models, dates of upcoming courses on gllamm, and references (often with links) to more than 150 papers published on using gllamm to fit models.

gllamm uses full maximum likelihood to estimate the parameters of these models and Gauss–Hermite quadrature or adaptive quadrature to evaluate the integrals of the likelihood. This common computation engine is one reason gllamm is so flexible and can fit so many models. It is, however, exceedingly computationally intensive, with the effect that gllamm can require substantial time to fit models. gllamm users are interested in seeing it run faster.

gllamm uses many Stata commands that have been parallelized, and some of gllamm's algorithms are written in C, sections of which have been parallelized. Even so, gllamm incorporates many algorithms, and these algorithms are triggered differently when fitting different models. It is difficult to say anything definitive about performance gains for gllamm when run under Stata/MP. Some gllamm models are highly parallelized, some not parallelized at all, and others are in between.

Table 4. Stata/MP performance, command by command

command	Run time as percentage of single processor time[a]				Percentage parallelized[b]
	Number of processors				
	2	4	8	16	
MIMIC model	67	50	*42*	*37*	67
Random-effects logistic	55	36	*27*	*22*	81
RE regression	47	29	*20*	*15*	87
Random-coefficients regression	56	40	*32*	*28*	74
Two-level RE logistic	60	42	*33*	*28*	75
Random-coefficients Poisson	98	97	*97*	*97*	2
RE logistic with constant	62	44	*35*	*30*	74

All values are expressed as a percentage of the time required on a single processor. Slanted values are extrapolated from 4 processors.

a. Smaller is better; 50 is perfect for 2 processors, 25 is perfect for 4 processors, and 12.5 is perfect for 8 processors.

b. Bigger is better; 100 is perfect.

Table 5. Command descriptions

command	description
`MIMIC model`	Multiple-equation, multiple-cause (MIMIC) latent variables structural equation model—ordered logistic
`Random-effects logistic`	Random-effects (random-intercepts) logistic regresion—same as `xtlogit, re`
`RE regression`	Continuous (Gaussian distribution) model with random intercepts—same as `xtreg, re`
`Random-coefficients regression`	Continuous (Gaussian distribution) model with random coefficients and intercepts
`Two-level RE logistic`	Logistic regression with two levels of random intercepts
`Random-coefficients Poisson`	Poisson count-data model with random intercepts and two random coefficients
`RE logistic with constant`	Logistic regression with random intercepts and fixed-effects constant

The graphs below show the observed performances from table 4 in graphical form. Those graphs are followed by graphs projecting performance through 16 processors.

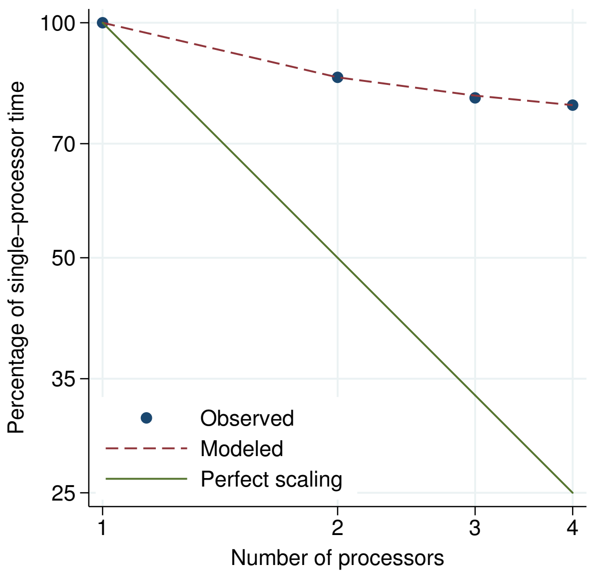

Figure E.1. `MIMIC model` performance plot.

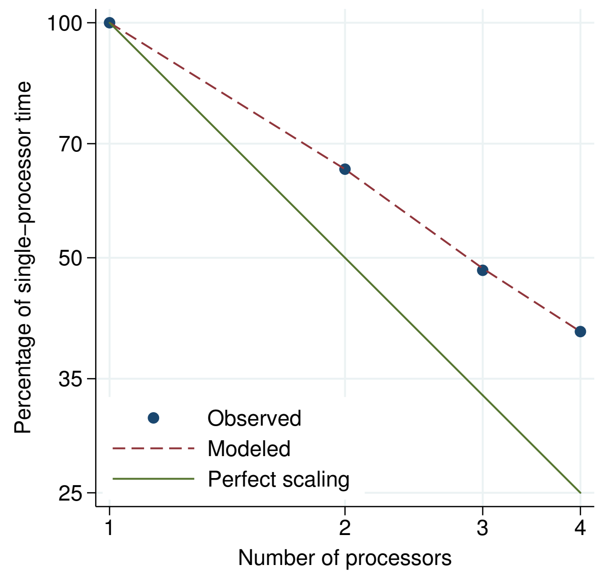

Figure E.2. `Random-effects logistic` performance plot.

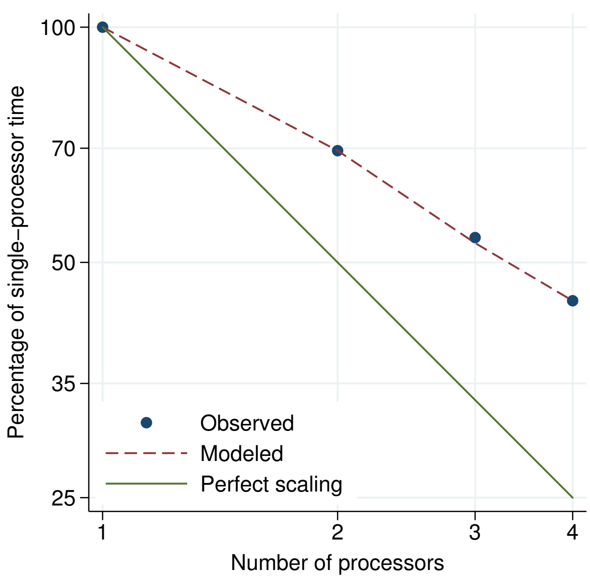

Figure E.3. `RE regression` performance plot.

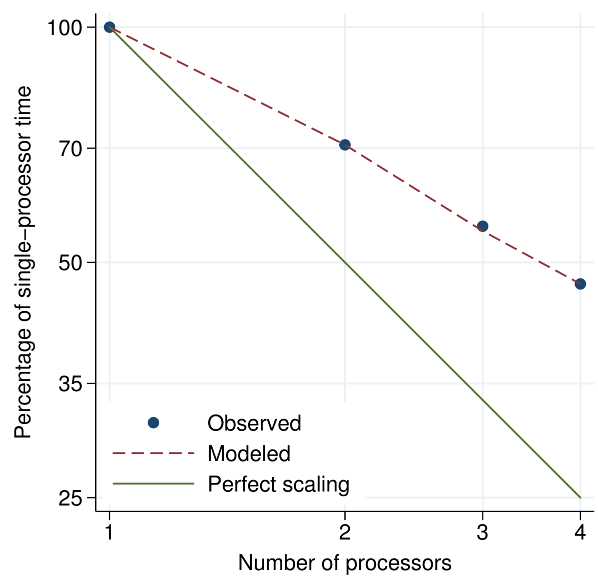

Figure E.4. `Random-coefficients regression` performance plot.

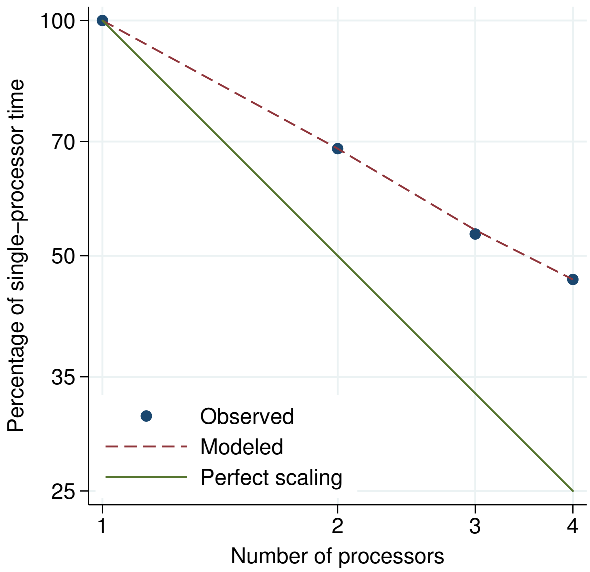

Figure E.5. `Two-level RE logistic` performance plot.

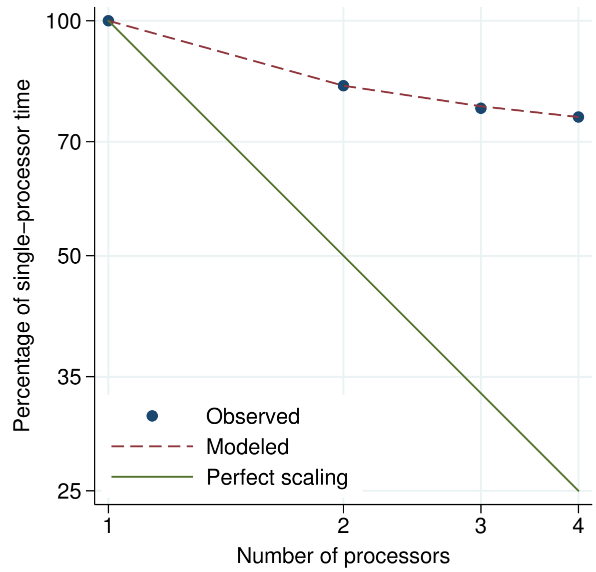

Figure E.6. `Random-coefficients Poisson` performance plot.

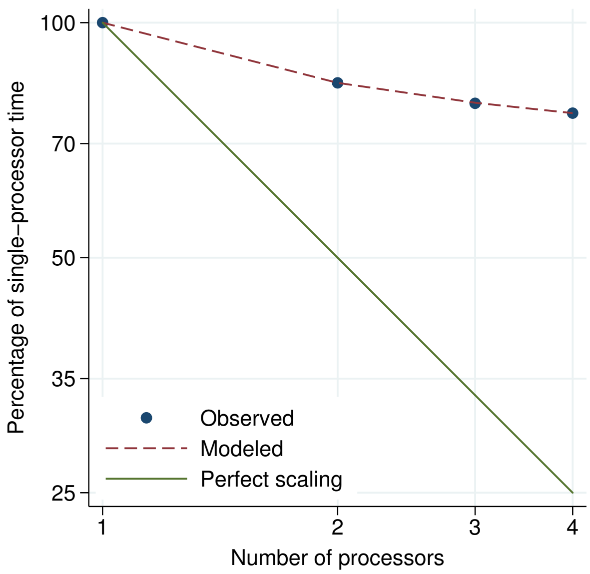

Figure E.7. `RE logistic with constant` performance plot.

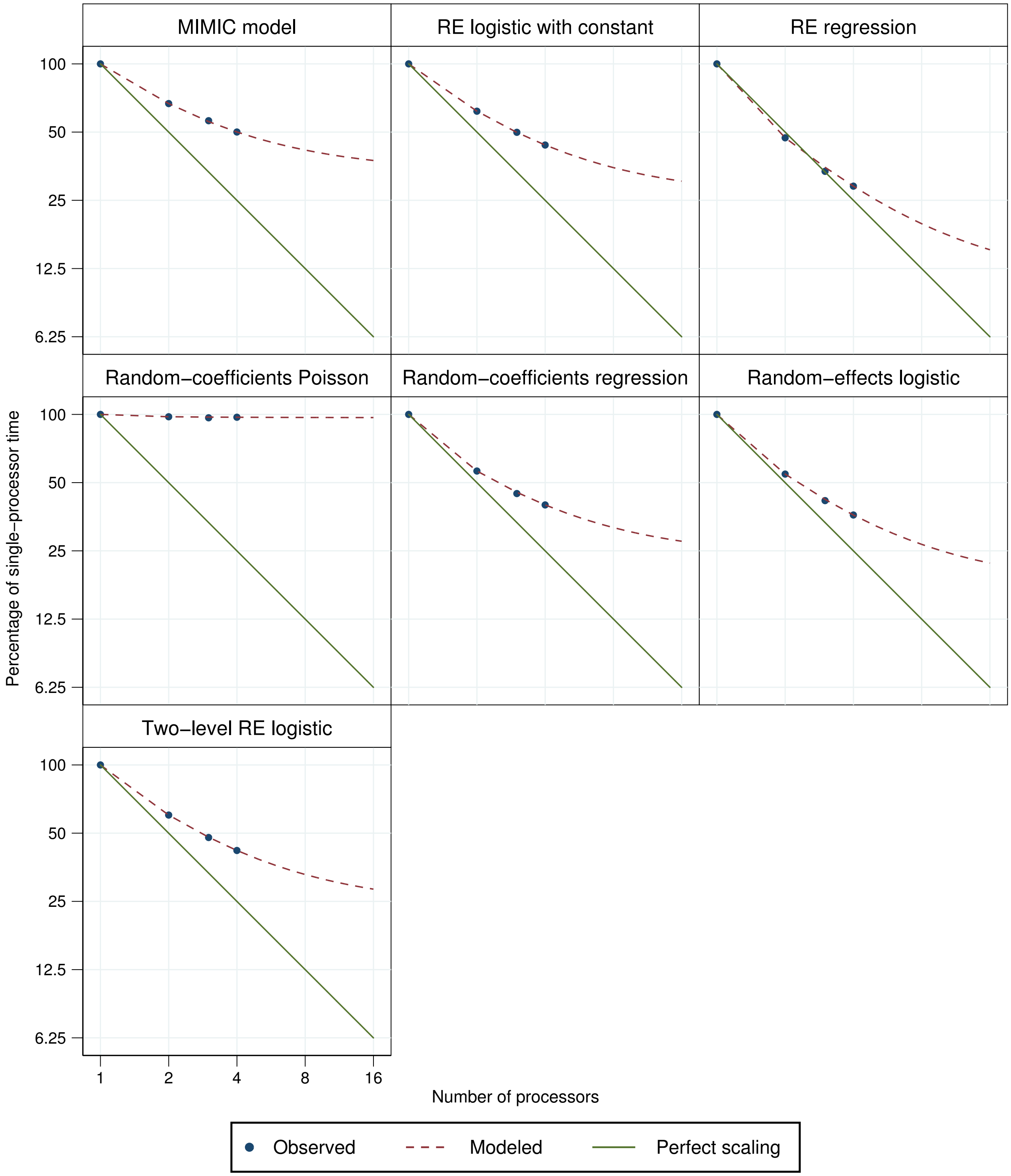

Figure E.8. **Parallelization performance plots.**

References

Culler, D. E., J. P. Singh, and A. Gupta. 1999. *Parallel Computer Architecture*. San Francisco: Morgan Kaufmann Publishers.

Dagum, L., and R. Menon. 1998. OpenMP: An industry-standard API for shared memory programming. *IEEE Computational Science & Engineering* 5: 46–55.

Grama, A., V. Kumar, A. Gupta, and G. Karypis. 2003. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. 2nd ed. Boston: Addison–Wesley.

Moore, G. E. 1965. Cramming more components onto integrated circuits. *Electronics* 38: 114–117.