

1 Summary

Stata/MP¹ is the version of Stata that is programmed to take full advantage of multicore and multiprocessor computers. Stata/MP distributes many of Stata’s most computationally demanding tasks across all the cores in your computer and thereby runs faster—much faster.

In a perfect world, software would run 2 times faster on 2 cores, 3 times faster on 3 cores, and so on. Stata/MP achieves about 74% efficiency. It runs 1.6 times faster on 2 cores, 2.3 times faster on 4 cores, and 2.9 times faster on 8 cores (see figure 1). Half the commands run faster than that. The other half run slower than the median speedup, and some of those commands are not sped up at all, either because they are inherently sequential (most time-series commands) or because they have not been parallelized (graphics, *mixed*).

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 79%. Taken at the median, estimation commands run 1.7 times faster on 2 cores, 2.6 times faster on 4 cores, and 3.4 times faster on 8 cores. Stata/MP supports up to 64 cores.

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

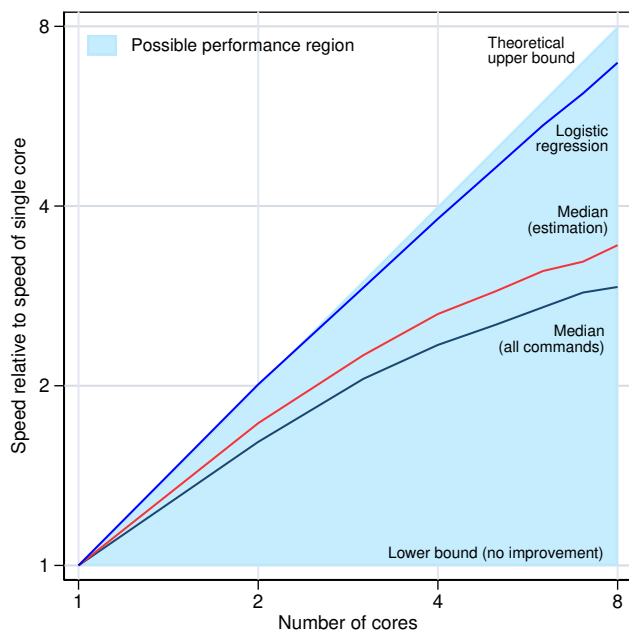


Figure 1. **Performance of Stata/MP.** Speed on multiple cores relative to speed on a single core.

1. Support for this effort was partially provided by the U.S. National Institutes of Health, National Institute on Aging grants 1R43AG019542-01A1, 2R44AG019542-02, and 5R44AG019542-03. We also thank Cornell Institute for Social and Economic Research (CISER) at Cornell University for graciously providing access to several highly parallel SMP platforms. CISER staff, 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

1	Summary	1
2	Table of contents	2
3	Introduction	3
4	Parallel computing hardware	4
5	Constructing Stata/MP	5
6	Measuring Stata/MP's performance	5
7	Performance summary	7
8	Stata/MP performance, command by command	14
9	Performance variability across computing platforms	53
10	Hyperthreading—single- and multiple-processor platforms	53
A	Performance assessment graphs for desktop computers	56
B	Performance assessment graphs for high-end servers	222
C	Command names and descriptions	296
D	Problem sizes	331
E	Commands not assessed	367
F	Mata	369
G	GLLAMM	370

3 Introduction

Stata/MP was designed to take advantage of computers with multiple cores and multiple processors by partitioning the work among the multiple cores. From the outset, Stata/MP was required to be 100% compatible with all other flavors of Stata, including Stata/SE and Stata/BE. Stata/MP was also required to run all 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 multicore and multiprocessor computers, including computers running MS Windows, Intel-based Mac OS computers, and Linux computers.

With multiple cores, one might expect to achieve the theoretical upper bound of doubling the speed by doubling the number of cores—2 cores 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) multicore/multiprocessor systems only duplicate processors and cores, not all the other system resources.

Stata/MP achieved 74% efficiency overall and 79% efficiency among estimation commands.

Speed is more important for problems that are quantified as *large* 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 cores runs half of Stata's commands at least 1.6 times faster than on a single core. With 4 cores, the same commands run at least 2.3 times faster than on a single core.

Figure 1, shown in the summary above, displays the theoretically possible performance as a shaded region. All Stata commands fall somewhere in the shaded region. Performance is measured as speed relative to speed on a single core: 1 indicates the speed on a single core; 2 means twice as fast as a single core; 4 means four times as fast as a single core; and so on. We could say the same thing in a different way: 2 means that a given problem runs in half the time required on a single core; 4 means that it runs in one-quarter the time; and so on.

The line in figure 1 for logistic regression reveals a speedup that is near the theoretical maximum. 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, so speeding them up was a priority for Stata programmers. Figure 1 also shows the median performance of Stata's estimation commands. The median estimation command runs 1.7 times faster on 2 cores and 2.6 times faster on 4 cores. Again, half the estimation commands speed up more than the median and half speed up less. Twenty-five percent of estimation commands speed up 2 times with 2 cores (the theoretical limit) and more than 3.7 times with 4 cores (this is not shown on the graph).

Figure 1 emphasizes dual-core, quad-core, and 8-core computers because those are the most common multicore platforms available. Stata/MP will work with up to 64 cores, however, and performance improvements continue to increase with more cores. For example, 25% of estimation commands run at least 6 times faster on 8-core computers, 9 times faster on 16-core computers, 12 times faster on 32-core

computers, and 15 times faster on 64-core 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

Chip makers are increasing the number of cores on a computer processor, and computer makers are increasing the number of processors in a computer. Prior to 2005, chip makers essentially doubled the speed of computer processors every 18 months, a fact known informally as Moore's law (Moore 1965). The speed improvements were achieved 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 alternatives for further speeding up processors are on the horizon, these alternatives involve dramatic changes in technology and fabrication.

The other alternative to make computers run faster is simply to give you more processors or cores.

Modern computers run faster by having multiple processors in one box or multiple processors on one chip. When multiple processors are on one chip, the chip makers call such processors cores, and the chip they reside on is called a multicore processor. Each core is itself a processor that is bundled together with other cores onto a single chip.

Regardless, when they reside together in one box, all the processors and cores share the main memory, disk drives, and other devices on the computer. Most modern computers use multicore processors. Modern servers typically use multiple processors, each having multiple cores. Whether the cores are on one processor chip or on multiple processor chips does not much matter.

Following the lead of the chip makers, we are going to count cores and talk about cores on a computer. We are also going to use the term multicore to include both a single-processor computer whose processor has multiple cores and a multiprocessor computer whose processors also may have multiple cores.

Multicore designs work exceptionally well when running different programs simultaneously, especially when programs run independently. Hence, a 4-core computer can do almost as much work as four separate computers, and none of the programs needs to be modified to recognize that it is running in a multicore environment.

Single programs can take advantage of multicore environments, too, but the programs must be modified to do so. This modification is accomplished 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 and edit a document simultaneously. This type of threading is relatively easy to implement and is even allowed on single-core 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. To speed computations, a statistical package must be able to perform simultaneous computations on the same task. This ability is referred to as symmetric multiprocessing (SMP). Stata/MP is a modified version of Stata that uses SMP to speed up its computations.

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

For a thorough discussion of parallel processing, see [Culler, Singh, and Gupta \(1999\)](#) and [Grama et al. \(2003\)](#).

5 Constructing Stata/MP

For Stata to take advantage of multicore systems, sections of its code had to be rewritten to distribute their work across cores. Stata's internal design includes key algorithms that are used in many contexts. Once those key 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. One way parallelization happens is that many statistical computations can be partitioned over observations.

Parallelizing key algorithms 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 250 sections of Stata's internal code were parallelized 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 cores (or processors). With 2 cores, that limit is twice as fast (or half the run time); with 4 cores, the limit is 4 times as fast (or one-quarter the run time); and so on. This limit 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, for example, 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 you might imagine. 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, Stata/MP partitions on panels. Whereas most time-series commands run only a little faster in the SMP environment, most panel-data commands run substantially faster.

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

Taken together, those sections of code 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 and their associated relative speeds, 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 us to make extrapolations on how problems will run when the number of cores 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 unparallelized regions of code; and Block *C*, the additional overhead required for parallelization.

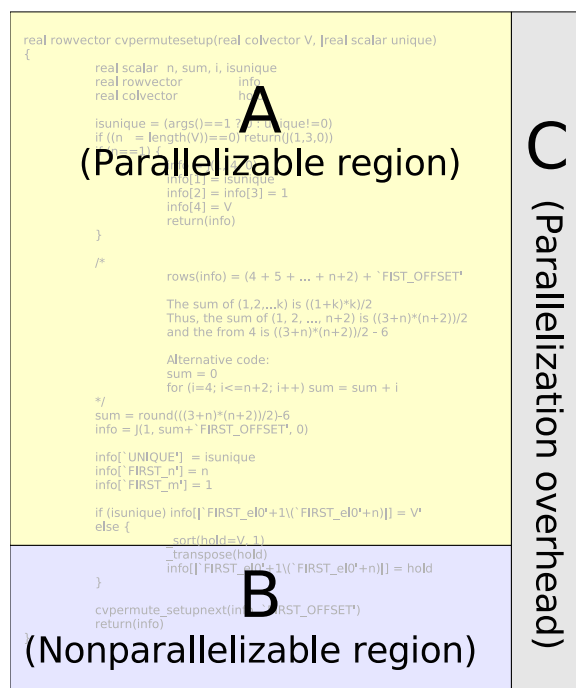


Figure 2. **Parallelization components.**

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 core. 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 cores.

If we know the percentage of time spent in A , then we have completely described the SMP performance of a command. Ignoring C , that is just $100A/(A + B)$.

We want to be more conservative, however, and account for the time required to parallelize the command. Considering C to be only the parts of the overhead that cannot be parallelized, we will refer to $100A/(A + B + C)$ as the percentage parallelized:

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

The percentage parallelized is a useful measure of how much performance will improve as cores (or processors) are added. All gains to parallelization occur because region A can be made to run on multiple cores at the same time. If we partition the region perfectly and each core runs uninterrupted, then when we double the number of cores, we halve the time used to perform A . As we add more cores, time spent in A continues to decrease. With 2 cores, it is $A/2$; with 4 cores, it is $A/4$; and with c cores, it is A/c . If we increase the number of cores without bound, A/c goes to zero. In contrast, $B + C$ is a constant time for running the command; it cannot be reduced by adding more cores. As we add cores, the run time asymptotes to $B + C$.

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

The concept of percentage parallelizable helps clarify why some commands will have less-than-perfect scaling and allows results to be extrapolated to any number of cores. We also present performance results as simple relative speeds that can be read directly from tables or graphs to find the relative speed for multiple cores or processors compared with the speed for a single core or processor.

7 Performance summary

The performance of Stata/MP has been measured on 655 Stata commands. Excluding I/O commands, these 655 commands are most of the 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 because they consume a negligible part of the time required to perform any analysis. Commands that run a target command repeatedly are not explicitly assessed, and some other commands are not timed for a variety of reasons; see appendix E. If you are searching this document for a specific command, know that we have tried to list every Stata command somewhere in the paper.

For each of the 655 commands, timings were recorded on a multicore computer where Stata/MP used 1, 2, ..., 40 cores to execute the same command. The computer contained four processors, each having 10 cores, for a total of 40 cores. 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 shortest time was recorded.

Timings have also been performed on other dual-core, quad-core, 8-core, and 16-core computers.

Although speeds relative to a single core 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 results of the timings are presented in section 8, *Stata/MP performance, command by command*, and appendix A, *Performance assessment graphs for desktop computers*.

Appendix A, *Performance assessment graphs for desktop computers*, shows graphs for each of the 655 commands. Figure 3 shows the graph of Stata's logistic regression command, `logistic`:

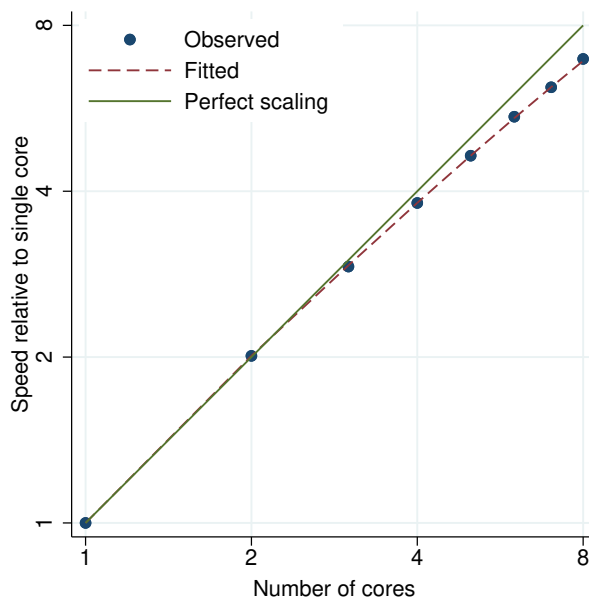


Figure 3. `logistic` performance plot.

The y axis shows speed relative to the speed on a single core. For `logistic`, the relative speeds are 2 (2 cores), 3.8 (4 cores), and 7 (8 cores). Also shown is a 45° reference line representing perfect scalability or, if you prefer, 100% parallelized: 2 times (2 cores), 4 times (4 cores), and 8 times (8 cores). `logistic` is 97% parallelized, but even so, you can see that its relative speeds are a bit below what is theoretically possible.

Stata's linear regression command, `regress`, very nearly achieves theoretical limits (see figure 4); its relative speeds increase in almost direct proportion to the number of cores.

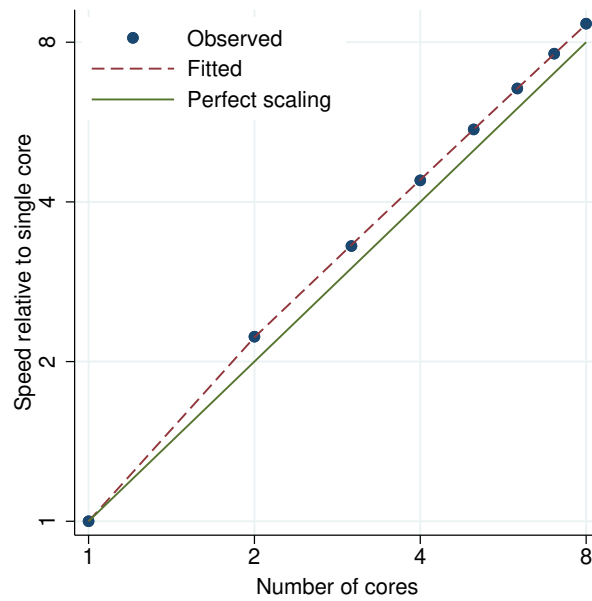


Figure 4. regress performance plot.

Figure 5 shows the graph for arima:

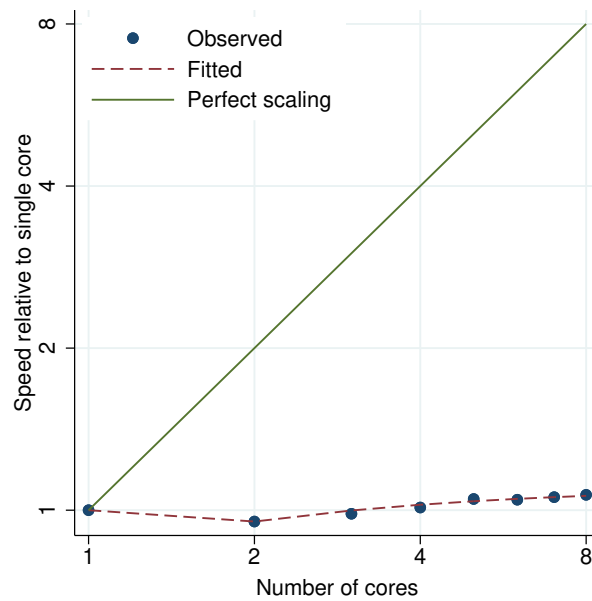


Figure 5. arima performance plot.

`arima`, a time-series command, hardly benefits from parallelization. Relative speeds are 1 (2 cores), 1 (4 cores), and 1.1 (8 cores).

Figure 6 shows the graph for Stata's command for Poisson regression with endogenous treatment effects, `etpoisson`:

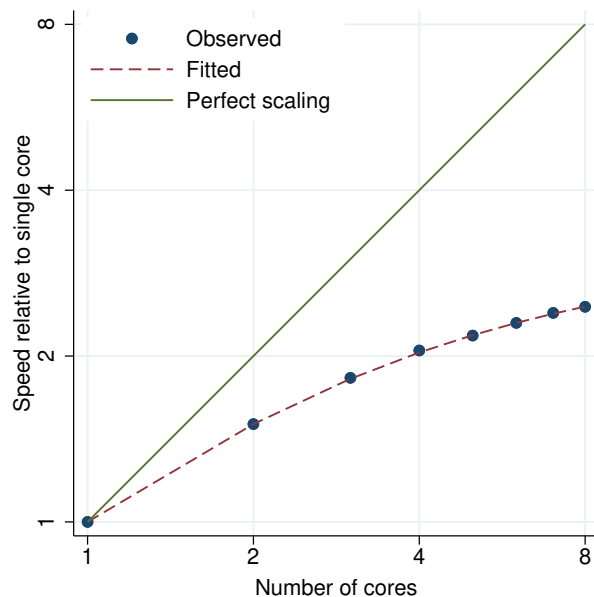


Figure 6. `etpoisson` performance plot.

Relative speeds are 1.5 (2 cores), 2 (4 cores), and 2.5 (8 cores). What is interesting about this graph is that the line flattens out as the number of cores increases. This is what happens when a command is not 100% parallelized: the relative run time approaches a horizontal asymptote that is related to the percent parallelized, which here is about 62%. Specifically, the asymptote is at $1/\{1 - (\text{percentage parallelized})/100\}$, which for `etpoisson` is about 2.6.

Finally, all 655 performance profiles can be combined into one figure, such as figure 7. The shaded area shows the region containing all possible performances. The diagonal top of the region represents perfect scaling (the maximum speed theoretically possible), while the horizontal lower boundary of the region represents no speed improvement.

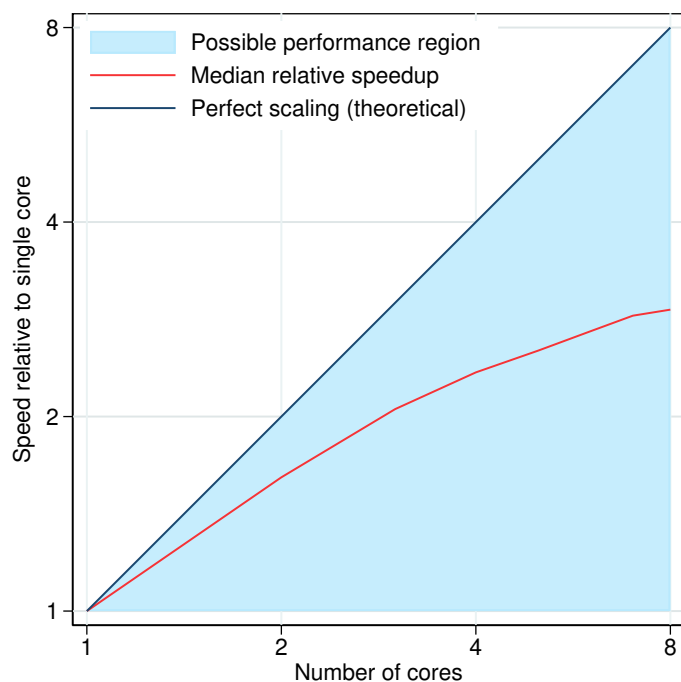


Figure 7. **Performance of Stata/MP.** Speed on multiple cores relative to speed on a single core.

Also included are the median results over all 655 commands; 327 commands have better performance gains (their curves lie above the median relative speedup line), and 327 exhibit lesser performance gains (their curves lie below the line).

Median performance for most Stata users will 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 analyses we had to run to record reliable results. Stata's command for t tests runs quickly on single or multiple cores. Meanwhile, Stata/MP development efforts focused on improving run times of commands that require substantial run times. Ergo, the median improvements are understated.

Figure 8 better illustrates the distribution of results by showing not just the median but also the quartiles. The most interesting thing about figure 8 is the first quartile (light-blue swath at the top). It shows that 25% of commands exhibit nearly perfect scaling. The worst commands among this group run about 2 times faster on 2 cores, 3.6 times faster on 4 cores, and 6.1 times faster on 8 cores.

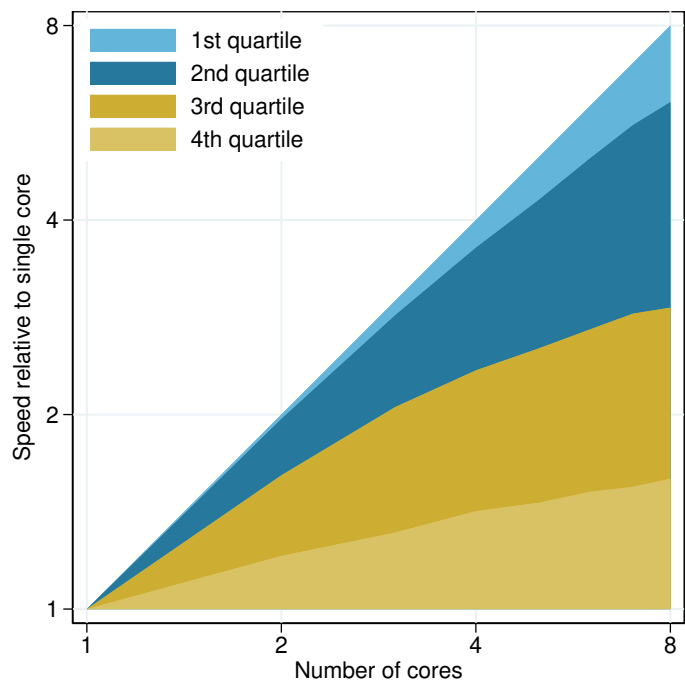


Figure 8. **Quartiles of Stata/MP performance.**
Speed on multiple cores relative to speed on a single core.

Figures 7 and 8 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 require iterative solutions.

Figure 9 summarizes the observed performance and median performance for the 417 estimation commands. These include all the estimation commands in Stata, and some commands are included more than once to include critical options, such as `vce(robust)` and `vce(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 proceeds and thus affect speed.

Compared with figure 7, figure 9 shows that the median performance for estimation commands is better than the overall median. The median relative speed for estimation commands is 1.7 times faster on 2 cores, 2.6 times faster on 4 cores, and 3.4 times faster on 8 cores. Half of all estimation commands perform even better. Figure 10 reveals that only 25% of all estimation commands run less than 1.4 times faster on 2 cores, less than 1.8 times faster on 4 cores, and less than 2 times faster on 8 cores.

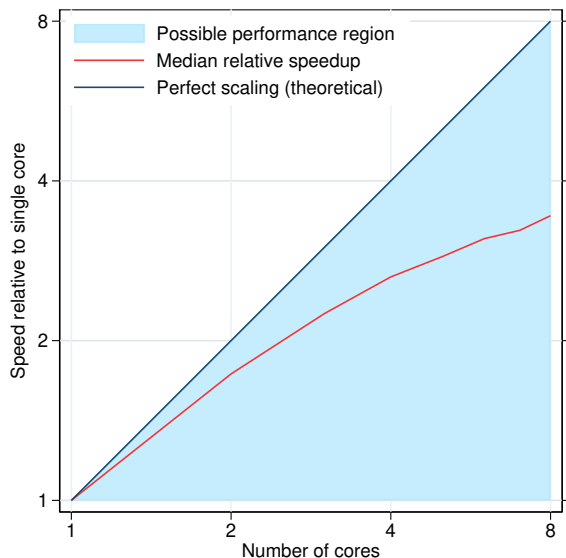


Figure 9. **Performance of Stata/MP on estimation commands.** Speed on multiple cores relative to speed on a single core.

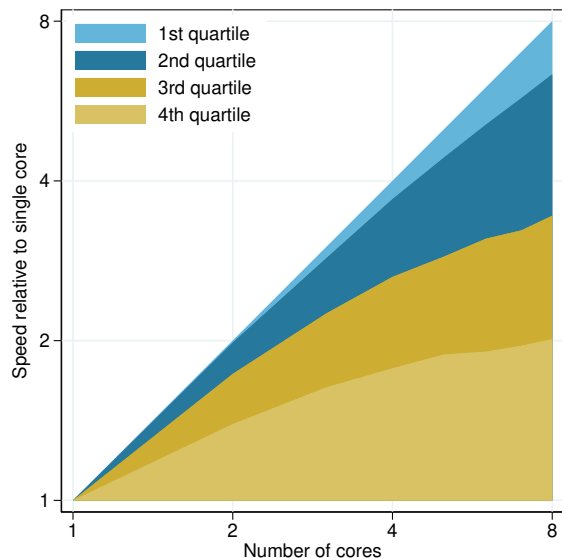


Figure 10. **Quartiles of Stata/MP performance on estimation commands.** Speed on multiple cores relative to speed on a single core.

We have emphasized results on 2, 4, and 8 cores because those are the most common desktop architectures currently available. Stata/MP supports up to 64 cores, and performance continues to improve as cores are added. Figure 11 shows the performance boundary and median for all 417 estimation commands on 1–40-core computers, and figure 12 shows their performance quartiles.

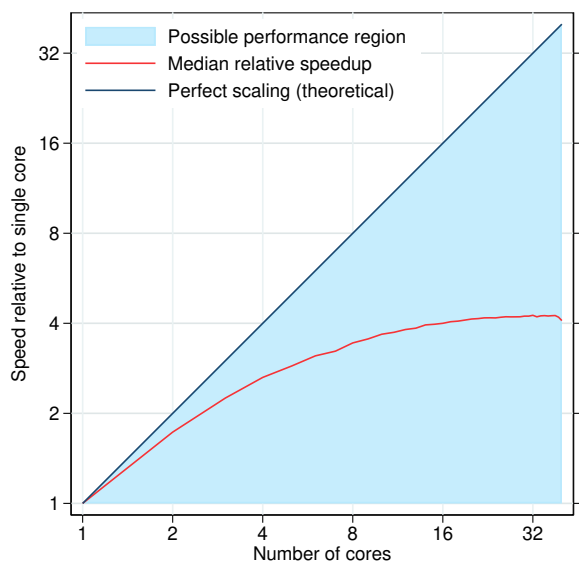


Figure 11. **Performance of Stata/MP on estimation commands (1 to 40 cores).** Speed of estimation commands on multiple cores relative to speed on a single core.

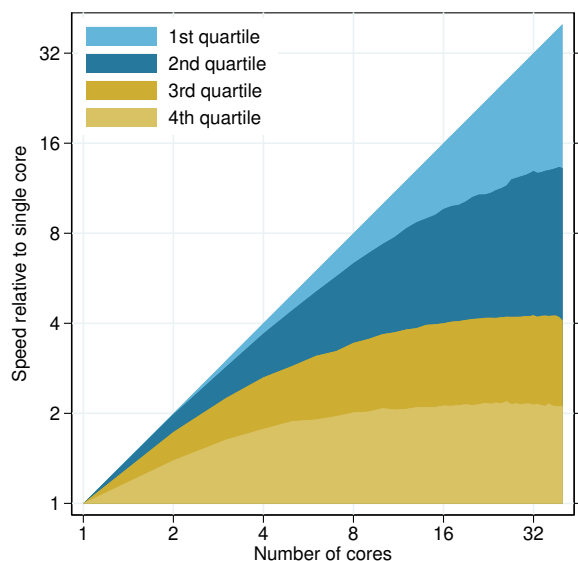


Figure 12. **Quartiles of Stata/MP performance on estimation commands (1 to 40 cores).** Speed of estimation commands on multiple cores relative to speed on a single core.

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 on large problems.

Table 1, toward the end of this section, provides relative speeds on individual commands, comparing the speed on 2, 4, 8, and 16 cores with the speed on a single core. The table also reports 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 required substantial time to solve and that were large enough to measure performance gains on 8, 16, 32, or even 64 cores. 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 cores. Longer timings also ameliorate variations in timings, such as interruptions caused by operating system processes or the memory status of the system when the command begins. Run-to-run variations are much greater for smaller problems that have shorter run times.

Timings were typically performed on commands that took 1–2 minutes to run on a single-core computer running at 2.2–3.4 GHz. For some commands, this meant that 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 slow, because of, for example, iterative or even simulated solutions. For details on the sizes of the problems, see appendix D.

Stata/MP was designed mostly to improve performance on large problems, such as those reported in appendix D. Even so, the performance on small-to-moderate problems improves surprisingly well. Using the same commands as those in appendix D, but with problems 100 to 10,000 times smaller and run times of 0.4 seconds to just over 4 seconds on a machine running at 2.2–3.4 GHz, substantial speedups were still observed. Among commands that were at least 50% parallelized, more than half exhibited greater than 90% 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 were obtained from the minimum of three runs on a 40-core computer.

Stata/MP performance was tested on many computers under MS Windows, Mac, and Linux operating systems. Although performance varies somewhat across platforms, the results from the table below can reasonably be applied to any platform.

Most users should simply look at the column reporting results for the number of cores in which they are interested. This column estimates the speed on that number of cores relative to the speed on a single core. Given a computer with a known number of cores, this column of results is the most direct measure of performance improvement.

Relative speed is easy to understand. When relative speed is 2, you could run a given problem twice in the same amount of time that you could run it once on a single-core computer. When relative speed is 4, you could run a given problem four times, and so on. Equivalently, when relative speed is 2, you could run a given problem in half the time that you could run it on a single-core computer. When relative speed is 4, you could run a given problem in one-fourth the time, and so on.

Table 1 also presents the percentage parallelized discussed in section 6. Given a set of percentage run times (relative to the run times on a single core) for at least 3 different numbers of cores, we can estimate the percentage parallelized and parallelization overhead parameters. The form of the model is particularly simple:

$$\text{percentage run time} = \alpha + \widehat{PP} \frac{1}{c} + \widehat{O} \frac{\delta_1}{c} \quad (1)$$

where c is the number of cores, δ_1 is an indicator for $c > 1$, and α is an intercept.

Our parameters of interest are directly estimated:

$$\text{percentage parallelized} = \widehat{PP} \quad (2)$$

and

$$\text{parallelization overhead} = \widehat{O} \quad (3)$$

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 \widehat{O} captures any unexpected difference in the speed when using one core. Because different computer, processor, cache, and operating system architectures respond differently in moving from 1 to 2 cores, \widehat{O} captures not only the theoretical parallelization overhead, but also anything that causes the time from the first core to differ from the time from the second.

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

The estimated percentage parallelized is also the most comparable measure across computing platforms; it is very consistent from one platform to another. Most of the differences across computing platforms are captured in \widehat{O} . Because the relative speeds are compared with the run time on a single core, 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. To learn more about any command, including worked examples, all of the Stata manuals can be access from <http://www.stata.com/features/documentation/>.

Appendix A contains performance graphs for each command using 1–8 cores. Appendix B contains graphs using 1–40 cores. The graphs plot the observed relative speed, the modeled performance using equation (1), and the perfect scalability reference line. If you are reading the PDF version of this document, you can click on the command name in table 1 to go to the page with the associated graph.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
alpha	1.6	2.8	4.2	5.6	87
ameans	1.8	2.8	3.9	4.9	84
anova (one-way)	2.0	3.7	6.5	10.2	96
anova (two-way)	2.8	5.2	8.3	11.7	95
arch	0.8	1.1	1.1	1.1	13
areg	2.3	4.1	6.2	8.1	91
areg, vce(cluster)	2.0	3.6	5.7	7.7	92
areg, vce(robust)	2.0	3.6	5.4	7.6	92
arfima	1.1	1.1	1.1	1.1	7
arima	1.0	1.0	1.1	1.0	4
bayes dsge	0.9	0.9	0.9	0.9	0
bayes dsge1	0.9	0.9	0.9	0.9	0
bayes: logit	2.1	4.0	6.9	10.8	96
bayes: poisson	1.7	2.6	3.4	4.2	82
bayes: regress	1.9	3.4	5.1	6.8	90
bayes var	0.9	0.9	0.9	0.9	0
bayesmh logit	1.3	1.4	1.5	1.5	34
bayesmh mvn	1.1	1.1	1.1	1.1	8
bayesmh mylogit	1.3	1.5	1.8	1.6	39

All values are expressed as the speed relative to the speed of a single core.

- a.* Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.
- b.* Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
bayesmh nl	0.9	1.0	1.1	1.1	14
bayesmh normal	1.2	1.2	1.3	1.3	23
bayesmh normal gibbs	1.1	1.1	1.1	1.1	7
bayesmh normal re	1.2	1.2	1.2	1.2	15
betareg, link(logit)	2.0	3.8	6.9	11.6	97
betareg, link(probit)	2.0	3.8	6.9	11.5	97
binreg	2.2	4.0	6.9	10.8	96
biplot	1.0	1.0	1.0	1.0	0
biprobit	1.9	3.5	6.7	11.7	98
biprobit (seemingly unrelated)	1.9	3.9	7.0	11.5	97
bitest	1.7	2.9	4.3	5.6	88
blogit	2.0	3.6	6.6	10.6	96
bmaregress	1.7	2.7	3.9	4.1	79
boxcox	2.1	3.9	6.9	10.8	96
bprobit	1.9	3.6	6.3	10.1	96
brier	1.4	1.4	1.6	1.7	44
bsample	1.1	1.4	1.5	1.6	42
bstat	1.7	2.9	4.4	5.5	87
by: generate	2.0	4.2	8.3	16.0	100

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
by: generate (small groups)	2.2	3.9	5.8	10.1	92
by: replace	2.9	5.8	11.5	22.9	100
by: replace (small groups)	8.0	14.9	20.8	24.2	98
ca	1.4	1.8	2.1	2.3	61
candisc	2.4	4.6	8.1	13.2	97
canon	1.7	2.9	4.7	6.8	93
cc	1.2	1.4	1.4	1.4	35
by: cc	1.0	1.0	1.0	1.0	2
centile	1.1	1.5	1.7	1.9	51
churdle linear	1.8	3.2	4.8	6.3	89
ci means	1.5	1.9	2.5	2.7	65
ci means, poisson	1.2	2.0	2.4	3.0	72
ci proportions	1.7	2.7	3.5	3.9	78
clogit (k1 to k2 matching)	1.9	3.5	6.0	9.5	95
clogit (1 to k matching)	1.5	1.9	2.3	2.6	65
cloglog	1.9	3.7	6.6	11.0	97
cluster averagelinkage	1.9	3.7	6.9	12.9	99
cluster centroidlinkage	1.8	3.6	6.9	12.8	99
cluster completelinkage	1.8	3.5	6.6	12.0	98

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>cluster generate</code>	1.0	0.9	0.9	0.9	0
<code>cluster kmeans</code>	2.3	4.5	8.6	15.1	99
<code>cluster kmedians</code>	2.3	4.6	8.9	16.2	99
<code>cluster medianlinkage</code>	1.9	3.7	7.1	12.8	99
<code>cluster singlelinkage</code>	1.0	0.9	1.0	1.0	0
<code>cluster wardslinkage</code>	1.8	3.4	6.4	11.3	98
<code>cluster waveragelinkage</code>	1.8	3.4	6.5	11.5	98
<code>cmlogit</code>	1.8	3.2	4.9	6.5	90
<code>cmmprobit</code>	1.3	1.5	1.6	1.4	15
<code>cmroprobit</code>	1.3	1.6	1.6	1.3	7
<code>cnsreg</code>	2.3	4.5	8.9	17.3	100
<code>codebook</code>	2.0	3.2	5.5	7.5	92
<code>collapse</code>	3.1	4.9	6.2	7.3	88
<code>compare</code>	1.8	2.9	3.8	5.3	86
<code>compress</code>	1.0	1.0	0.8	0.8	0
<code>contract</code>	1.2	1.3	1.5	1.5	40
<code>corr2data</code>	1.9	3.6	5.5	7.4	91
<code>correlate</code>	2.3	4.5	8.9	17.6	100
<code>corrgram</code>	1.2	1.4	1.3	1.5	37

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>count</code>	2.0	4.0	8.0	15.8	100
<code>cpoisson</code>	1.6	2.2	2.7	3.0	71
<code>cs</code>	1.1	1.2	1.3	1.3	23
<code>by: cs</code>	1.0	1.0	1.0	1.0	0
<code>ctset</code>	2.0	4.0	8.0	14.1	99
<code>cttost</code>	1.0	1.0	1.0	1.0	3
<code>cumul</code>	1.1	1.3	1.5	1.5	39
<code>cusum</code>	1.3	1.6	1.8	1.9	52
<code>datasignature</code>	1.0	1.0	1.0	1.0	3
<code>decode</code>	1.0	1.0	1.0	1.0	0
<code>demandsys aids</code>	1.0	1.0	1.0	1.0	0
<code>demandsys cdouglas</code>	1.3	1.6	1.8	1.9	50
<code>demandsys gaids</code>	1.0	0.9	0.8	1.0	0
<code>demandsys gquids</code>	1.2	1.0	1.2	1.1	15
<code>demandsys gtranslog</code>	1.0	1.0	1.0	1.0	0
<code>demandsys les</code>	1.5	2.0	2.3	1.6	56
<code>demandsys quids</code>	0.9	1.0	0.9	0.9	0
<code>demandsys translog</code>	1.0	1.0	1.0	1.0	0
<code>destring</code>	1.2	1.0	1.1	1.2	21

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>dfactor</code>	1.6	2.1	2.7	2.9	67
<code>dfgls</code>	1.2	1.3	1.4	1.4	31
<code>dfuller</code>	2.0	2.6	3.0	2.6	71
<code>didregress</code>	1.6	2.3	3.1	3.6	77
<code>discrim knn</code>	1.5	2.1	2.5	2.8	67
<code>discrim lda</code>	2.0	3.5	5.9	8.7	94
<code>discrim logistic</code>	1.8	3.4	6.4	11.5	98
<code>discrim qda</code>	1.7	2.4	3.0	3.5	75
<code>dotplot</code>	1.1	1.4	1.5	1.6	41
<code>drawnorm</code>	2.0	3.6	6.3	9.7	95
<code>drop if <i>exp</i></code>	1.0	1.0	1.0	1.0	0
<code>drop in <i>range</i></code>	1.0	1.0	1.0	1.0	0
<code>dsge</code>	0.9	0.9	0.9	0.9	0
<code>dsgenl</code>	0.9	0.9	0.9	0.9	0
<code>dslogit</code>	1.3	1.7	1.8	2.0	52
<code>dspoisson</code>	1.3	1.6	1.9	2.0	54
<code>dsregress</code>	1.4	1.6	1.8	1.8	49
<code>dstdize</code>	0.9	0.9	1.0	1.0	0
<code>dvech</code>	1.0	1.0	1.0	1.0	3

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>egen group()</code>	1.8	2.9	4.7	6.0	87
<code>by: egen mean</code>	1.1	1.4	1.5	2.6	66
<code>eivreg</code>	2.3	4.4	8.6	16.3	99
<code>encode</code>	1.4	2.3	3.1	3.9	81
<code>eregress</code>	1.7	2.5	3.2	3.6	76
<code>esize twosample</code>	1.3	1.6	1.8	1.8	43
<code>esize unpaired</code>	1.6	2.3	3.0	3.6	77
<code>eteffects (exponential), ate</code>	1.8	1.5	1.9	2.1	81
<code>eteffects (linear), ate</code>	2.0	3.5	5.6	7.6	92
<code>eteffects (linear), pomeans</code>	2.0	3.5	5.5	7.6	92
<code>eteffects (probit), ate</code>	2.0	3.5	5.5	7.4	91
<code>etpoisson</code>	1.5	2.0	2.5	2.5	62
<code>etregress, poutcomes</code>	1.8	2.9	4.2	4.5	80
<code>etregress, twostep</code>	2.0	3.8	7.1	11.9	98
<code>exlogistic</code>	1.2	1.2	1.2	1.2	18
<code>expand #</code>	0.9	0.9	0.9	0.9	0
<code>expand varname</code>	0.9	0.9	0.9	0.9	0
<code>expandcl #</code>	1.4	1.8	2.2	2.4	62
<code>expandcl varname</code>	1.3	1.7	2.0	2.2	60

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>expoisson</code>	1.4	1.4	1.4	1.4	30
<code>factor</code>	1.5	1.9	2.0	2.2	58
<code>fcast compute</code>	1.0	1.0	1.0	1.0	1
<code>fillin</code>	1.1	1.4	1.6	1.8	48
<code>fmm 2: poisson</code>	1.5	2.2	2.7	2.8	66
<code>fmm 2: regress</code>	1.4	1.9	2.2	2.1	53
<code>fmm 3: poisson</code>	1.4	1.9	2.2	2.4	58
<code>fmm 3: regress</code>	1.6	1.9	2.1	2.2	51
<code>fracreg probit</code>	2.1	4.1	7.9	14.1	98
<code>frontier</code>	2.0	4.0	7.5	13.6	99
<code>fvrevar (factors)</code>	1.6	3.3	4.9	6.3	88
<code>fvrevar (interaction)</code>	1.8	3.1	5.0	5.9	78
<code>generate (small expressions)</code>	3.5	6.7	12.2	20.2	98
<code>generate</code>	2.1	4.1	7.9	15.9	100
<code>glm, family(gamma)</code>	1.9	3.7	6.4	10.2	96
<code>glm, family(gaussian)</code>	2.1	4.0	7.1	11.8	97
<code>glm, family(igaussian)</code>	2.0	3.9	7.1	11.9	97
<code>glm, family(nbinomial)</code>	2.1	4.0	7.4	12.7	98
<code>glm, family(poisson)</code>	2.0	3.9	7.1	12.1	98

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>glogit</code>	2.2	4.4	8.5	15.8	99
<code>gmm</code>	1.1	1.2	1.2	1.2	16
<code>gmm (with derivatives)</code>	2.1	3.4	4.9	5.9	88
<code>gprobit</code>	2.2	4.4	8.2	15.1	99
<code>graph bar</code>	1.3	1.4	1.9	2.7	66
<code>graph box</code>	1.0	1.1	1.1	1.1	8
<code>graph pie</code>	1.2	1.3	1.4	1.4	29
<code>grmeanby</code>	1.3	1.5	1.6	2.9	68
<code>gsem, oprobit (CFA, 2-level)</code>	2.4	3.5	4.4	4.3	77
<code>gsem, oprobit (CFA)</code>	2.7	3.7	4.1	4.0	73
<code>gsem, logit group()</code>	1.7	2.6	3.4	3.9	78
<code>gsem, group()</code>	1.7	2.7	3.6	4.2	81
<code>gsem, ologit group()</code>	1.7	2.5	3.3	3.8	77
<code>gsem, poisson group()</code>	1.7	2.7	3.5	4.0	80
<code>gsort</code>	1.0	1.1	1.1	1.1	11
<code>hausman</code>	1.2	1.2	1.3	1.3	21
<code>hdidregress aipw</code>	1.3	1.7	1.9	2.0	51
<code>hdidregress ipw</code>	1.3	1.7	1.9	1.9	51
<code>hdidregress ra</code>	1.3	1.5	1.7	1.7	43

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>hdidregress twfe</code>	1.2	1.3	1.4	1.4	29
<code>heckman</code>	2.0	3.8	6.9	11.2	97
<code>heckman, twostep</code>	2.0	3.8	6.9	11.6	97
<code>heckoprobit</code>	2.0	3.7	6.5	10.0	96
<code>heckpoisson</code>	1.8	2.8	4.1	4.4	79
<code>heckprob</code>	2.0	4.0	7.2	11.8	97
<code>hetoprobit</code>	1.9	3.3	5.0	6.7	90
<code>hetprob</code>	1.7	3.5	5.9	8.8	94
<code>hetregress</code>	2.0	3.6	6.2	9.6	95
<code>hetregress, twostep</code>	2.1	4.0	7.4	12.1	97
<code>histogram</code>	1.3	1.6	1.8	2.0	52
<code>hotelling</code>	2.0	4.2	8.1	15.5	99
<code>icc, mixed</code>	1.2	1.4	1.6	1.7	44
<code>icc (one-way)</code>	1.4	1.7	1.9	2.0	53
<code>icc (two-way)</code>	1.1	1.3	1.4	1.7	46
<code>import delimited</code>	1.0	1.3	1.4	1.4	30
<code>intreg</code>	2.1	4.1	7.7	13.8	99
<code>ir</code>	1.1	1.2	1.3	1.3	27
<code>by: ir</code>	1.0	1.0	1.0	1.0	0

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>irf create</code>	1.3	1.6	1.7	1.9	53
<code>irt 1pl</code>	1.6	2.2	2.8	2.6	60
<code>irt 2pl</code>	1.6	2.4	2.8	2.6	59
<code>irt 3pl</code>	2.2	2.6	2.7	2.6	59
<code>irt grm</code>	1.7	2.3	2.8	2.7	62
<code>irt nrm</code>	1.5	2.1	2.5	2.4	57
<code>irt pcm</code>	1.6	2.1	2.5	2.3	55
<code>irt rsm</code>	1.6	2.1	2.4	2.2	48
<code>istdize</code>	1.0	1.0	1.0	1.0	0
<code>ivfprobit</code>	1.9	3.4	5.4	7.5	92
<code>ivpoisson cfunction</code>	2.5	4.2	6.0	7.3	89
<code>ivpoisson gmm, additive</code>	2.5	4.2	7.0	8.5	92
<code>ivpoisson gmm, multiplicative</code>	1.8	2.7	3.7	4.4	82
<code>ivprobit</code>	1.9	3.3	5.1	6.9	90
<code>ivqregress iqr</code>	0.9	1.1	1.3	1.3	25
<code>ivregress 2sls</code>	2.1	4.2	7.5	12.6	97
<code>ivregress gmm</code>	1.9	3.3	4.9	6.7	90
<code>ivregress liml</code>	2.0	3.7	6.5	10.1	96
<code>ivtobit</code>	1.7	2.3	2.9	3.2	73

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>kap</code>	1.1	1.4	1.7	1.9	53
<code>kappa</code>	2.0	3.6	6.1	8.8	93
<code>kdensity</code>	1.9	3.4	4.1	4.5	81
<code>keep if exp</code>	1.0	1.0	1.0	1.0	0
<code>keep in range</code>	1.0	1.0	1.0	1.0	0
<code>keep varlist</code>	1.0	1.0	1.0	1.0	0
<code>ksmirnov</code>	1.4	1.9	2.2	2.5	66
<code>ksmirnov, by()</code>	2.1	2.5	2.9	3.2	72
<code>ktau</code>	1.0	1.0	1.0	1.0	0
<code>kwallis</code>	2.3	2.8	3.3	3.7	76
<code>ladder</code>	1.7	2.9	4.1	4.8	85
<code>lasso linear</code>	1.3	1.6	1.7	1.7	45
<code>lasso logit</code>	1.1	1.2	1.3	1.3	24
<code>lasso poisson</code>	1.1	1.2	1.3	1.3	21
<code>gsem, lclass(C 2)</code>	1.6	2.2	2.8	3.2	73
<code>gsem, lclass(C 3)</code>	1.5	2.2	2.7	3.0	71
<code>levelsof</code>	1.2	1.7	2.2	2.2	49
<code>loadingplot</code>	0.9	0.9	1.0	1.1	5
<code>logistic</code>	2.0	3.8	7.0	11.9	97

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>logit</code>	2.0	3.8	6.9	11.8	97
<code>loneway</code>	1.1	1.3	1.4	1.6	40
<code>lowess</code>	2.0	3.4	6.8	13.2	100
<code>lpoly</code>	1.7	2.7	3.7	4.6	83
<code>ltable</code>	0.7	0.7	0.8	0.8	0
<code>makespline bspline</code>	1.5	2.2	2.7	3.0	70
<code>makespline linear</code>	1.8	3.2	4.6	5.6	86
<code>makespline piecewise</code>	1.8	3.0	4.3	5.2	83
<code>makespline rcs</code>	1.7	2.4	4.1	5.1	85
<code>manova (one-way)</code>	1.8	2.7	3.7	4.7	83
<code>manova (two-way)</code>	1.4	2.0	2.7	4.1	84
<code>margins</code>	1.8	3.4	6.1	11.1	98
<code>margins, dydx() exp()</code>	1.8	3.2	5.2	7.9	94
<code>margins, dydx()</code>	1.8	3.0	5.0	7.4	91
<code>margins, exp()</code>	1.8	3.2	5.4	8.3	94
<code>markout</code>	2.0	4.0	7.7	14.3	99
<code>marksample</code>	2.0	4.1	8.2	16.1	100
<code>marksample if exp</code>	2.0	4.0	8.0	16.0	100
<code>matrix accum</code>	2.3	4.6	9.1	18.1	100

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>matrix eigenvalues</code>	1.0	1.0	1.0	1.0	1
<code>matrix score</code>	2.1	4.2	8.4	16.6	100
<code>matrix svd</code>	1.0	1.0	1.0	1.0	0
<code>matrix syeigen</code>	1.0	1.0	1.0	1.0	1
<code>matrix syminv</code>	1.0	1.6	2.8	5.1	96
<code>mca</code>	1.0	1.0	1.1	1.1	8
<code>mcc</code>	1.1	1.1	1.2	1.2	18
<code>mds</code>	2.2	2.7	3.0	3.2	71
<code>mdslong</code>	1.9	2.3	2.5	2.6	63
<code>mean</code>	2.0	3.8	6.8	11.5	97
<code>mecloglog</code>	1.6	2.3	2.9	3.3	71
<code>median</code>	1.7	2.7	3.8	4.5	82
<code>mediate</code> (<code>,</code> <code>expmean</code>) (<code>,</code> <code>linear</code>)	1.3	1.7	1.9	1.9	51
<code>mediate</code> (<code>,</code> <code>linear</code>) (<code>,</code> <code>linear</code>)	1.4	1.9	2.2	2.3	59
<code>mediate</code> (<code>,</code> <code>logit</code>) (<code>,</code> <code>linear</code>)	1.3	1.7	2.1	2.1	54
<code>mediate</code> (<code>,</code> <code>poisson</code>) (<code>,</code> <code>linear</code>)	1.4	2.1	2.7	2.7	66
<code>mediate</code> (<code>,</code> <code>probit</code>) (<code>,</code> <code>linear</code>)	1.3	1.7	2.0	2.1	54
<code>meintreg</code>	1.8	2.8	4.0	5.0	84
<code>melogit</code>	1.7	2.5	3.3	3.7	76

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>menbreg, dispersion(constant)</code>	1.5	1.8	1.9	1.9	47
<code>menbreg, dispersion(mean)</code>	1.6	1.9	2.2	2.4	62
<code>menl</code>	1.1	1.2	1.2	1.2	14
<code>meologit</code>	1.8	2.6	3.4	4.0	78
<code>meoprobit</code>	1.7	2.6	3.5	4.1	79
<code>mepoisson</code>	1.6	2.2	2.7	2.8	67
<code>meprobit</code>	1.7	2.6	3.5	4.1	79
<code>mestreg, distribution(exp)</code>	1.6	2.3	2.9	3.2	71
<code>mestreg, distribution(weibull)</code>	1.7	2.6	3.4	4.0	80
<code>metobit</code>	1.7	2.6	3.5	4.0	79
<code>mgarch</code>	1.1	1.1	1.0	1.0	4
<code>mhodds</code>	1.1	1.1	1.4	1.5	27
<code>mhodds (adjusted)</code>	1.5	2.1	2.8	3.2	73
<code>by: mhodds</code>	1.0	1.0	1.0	1.0	4
<code>mhodds (trend)</code>	1.4	1.7	2.0	2.1	54
<code>mi estimate: logit (flong)</code>	1.4	1.8	2.1	2.5	63
<code>mi estimate: logit (flongsep)</code>	2.1	3.6	5.5	7.5	92
<code>mi estimate: logit (mlong)</code>	1.7	2.3	3.4	3.8	80
<code>mi estimate: logit (wide)</code>	1.9	3.4	5.4	7.6	93

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>mi estimate: mlogit</code>	1.9	3.5	6.2	10.0	96
<code>mi estimate: ologit</code>	1.9	3.5	5.9	9.1	95
<code>mi estimate: regress (flong)</code>	1.3	1.6	1.6	1.5	39
<code>mi estimate: regress (flongsep)</code>	1.9	3.0	3.9	4.6	83
<code>mi estimate: regress (mlong)</code>	1.5	1.8	2.1	2.2	60
<code>mi estimate: regress (wide)</code>	1.9	3.1	4.2	5.3	85
<code>mi impute chained (flong)</code>	1.2	1.4	1.6	1.6	42
<code>mi impute chained (flongsep)</code>	1.2	1.4	1.7	1.7	47
<code>mi impute chained (mlong)</code>	1.1	1.4	1.6	1.7	45
<code>mi impute chained (wide)</code>	1.3	1.6	1.8	2.0	53
<code>mi impute logit (flong)</code>	1.1	1.1	1.3	1.3	19
<code>mi impute logit (flongsep)</code>	1.5	1.9	2.2	2.3	60
<code>mi impute logit (mlong)</code>	1.2	1.4	1.7	1.8	46
<code>mi impute logit (wide)</code>	1.8	2.7	3.9	4.7	84
<code>mi impute mlogit</code>	1.6	2.1	2.6	2.9	69
<code>mi impute mono pmm</code>	1.5	2.3	2.8	2.8	65
<code>mi impute mono regress</code>	1.7	2.3	2.8	3.2	73
<code>mi impute mvn</code>	1.1	1.1	1.2	1.2	15
<code>mi impute ologit</code>	1.4	1.8	2.0	2.1	57

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
mi impute pmm	1.5	2.1	2.8	3.1	71
mi impute regress	1.3	1.7	1.8	1.9	49
misstable nested	1.3	1.6	1.8	2.0	51
misstable patterns	1.3	1.5	1.6	1.7	43
misstable summarize	1.0	1.0	1.0	1.0	0
misstable tree	1.3	1.6	1.9	2.0	53
mixed	1.2	1.1	1.1	1.2	13
mixed (crossed effects)	1.2	1.2	1.2	1.2	16
mkspline	1.9	2.8	3.6	3.3	79
mlevel	2.1	4.2	8.3	16.4	100
mlevel, nocons	2.1	4.2	8.3	16.4	100
mlmatbysum	1.9	3.6	6.7	12.2	98
mlmatsum	2.0	3.9	7.7	15.2	100
mlogit	2.0	4.0	7.7	14.8	99
mlsum	1.8	3.1	4.8	6.9	91
mlvecsum	1.9	3.8	7.4	14.0	99
mprobit	1.5	1.5	1.5	1.5	33
mswitch ar	0.8	0.8	0.7	0.8	0
mswitch dr	0.9	0.9	0.9	0.9	0

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
mvdecode	4.7	10.0	19.8	39.2	100
mvencode	2.0	3.9	7.8	15.2	100
mvreg	2.1	3.9	7.1	11.2	97
mvtest correlations	1.9	3.2	5.2	6.9	92
mvtest covariances	1.9	3.3	5.3	7.5	92
mvtest means, heterogeneous	1.5	2.1	2.4	2.9	68
mvtest means, homogeneous	1.9	2.8	3.3	3.5	79
mvtest means, lr	1.5	2.1	2.6	2.8	67
mvtest normality	0.9	0.9	0.9	0.9	0
nbreg	2.0	3.7	6.4	10.1	96
newey	1.2	1.3	1.4	1.5	34
nl	1.6	3.2	4.5	6.9	89
nlogit	1.5	2.0	2.2	2.2	55
nlsur	1.2	1.5	1.6	1.7	45
npregress kernel	0.9	1.0	0.9	0.9	0
nptrend, cuzick	1.5	2.0	2.2	2.2	56
nptrend, carmitage	1.3	1.6	1.8	2.0	54
nptrend, jterpstra	1.6	2.1	2.2	2.5	63
nptrend, linear	1.1	1.2	1.2	1.2	21

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>ologit</code>	2.0	3.8	7.3	13.0	99
<code>ologit, vce(cluster)</code>	2.0	3.6	6.2	9.7	96
<code>ologit, vce(robust)</code>	2.0	3.9	7.4	13.4	99
<code>oneway</code>	1.0	1.0	1.0	1.0	0
<code>oprobit</code>	2.0	3.9	7.3	13.5	99
<code>oprobit, vce(cluster)</code>	2.0	3.8	6.9	11.5	97
<code>oprobit, vce(robust)</code>	2.0	4.0	7.6	13.8	99
<code>orthog</code>	1.7	3.8	6.0	7.9	89
<code>pca</code>	1.5	2.1	2.7	3.1	71
<code>pcorr</code>	2.2	4.5	8.7	16.7	100
<code>pctile</code>	2.1	3.7	5.3	6.6	89
<code>pergram</code>	1.0	1.0	1.0	1.0	1
<code>pkcollapse</code>	0.8	0.9	0.9	0.9	0
<code>pkexamine</code>	1.1	1.1	1.2	1.2	18
<code>pksumm</code>	0.9	0.9	0.9	0.9	0
<code>poisson</code>	2.1	4.0	7.5	13.0	98
<code>poisson, vce(cluster)</code>	2.0	3.9	7.0	11.6	97
<code>poisson, exposure()</code>	2.1	4.1	7.6	13.2	98
<code>poisson, vce(robust)</code>	2.1	4.0	7.6	13.2	98

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
pologit	1.5	2.0	2.3	2.3	60
popoisson	1.6	1.9	2.1	2.3	60
poregress	1.4	1.7	2.0	2.1	55
pperron	1.1	1.1	1.1	1.7	46
prais	1.1	1.2	1.2	1.3	22
predict, cooks	2.0	4.0	7.9	15.8	100
predict, covratio	2.0	4.0	7.9	15.2	99
predict, dfbeta	2.1	4.0	7.6	13.3	98
predict, dfits	2.0	4.0	7.8	14.6	99
predict, e	2.0	3.8	6.4	10.3	96
predict, leverage	2.0	4.0	7.9	14.6	99
predict, pr	1.9	3.7	6.5	10.6	96
predict, residuals	2.1	4.3	8.3	16.0	100
predict, rstandard	2.0	4.0	8.1	16.0	100
predict, rstudent	2.0	4.0	8.0	15.9	100
predict, stdf	2.0	4.0	8.0	15.5	100
predict, stdp	2.0	4.0	8.0	15.9	100
predict, stdr	2.0	4.0	8.0	15.9	100
predict, welsch	2.0	4.1	8.0	15.6	100

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>predict, ystar</code>	1.9	3.5	5.9	8.2	94
<code>predictnl</code>	2.0	3.8	6.5	9.7	95
<code>probit</code>	2.3	4.4	7.9	13.1	97
<code>procrustes</code>	1.9	3.7	5.4	7.4	91
<code>proportion</code>	1.5	2.1	2.5	2.8	79
<code>prtest1</code>	1.8	3.0	4.3	6.2	90
<code>prtest2</code>	1.7	2.8	4.1	5.5	88
<code>prtest, by()</code>	1.1	1.2	1.2	1.2	21
<code>pwcorr</code>	1.7	2.5	5.4	8.3	95
<code>qreg</code>	2.5	4.1	6.1	7.9	90
<code>ranksum</code>	2.9	3.4	4.0	4.1	78
<code>ratio</code>	1.4	1.8	2.0	2.2	58
<code>ratio (exp1) (exp2)</code>	1.5	1.8	2.2	2.4	62
<code>recode</code>	1.4	1.7	2.0	2.2	59
<code>reg3</code>	2.1	3.8	6.2	8.8	94
<code>regress</code>	2.2	4.4	8.7	16.7	100
<code>regress, vce(cluster)</code>	2.0	3.3	5.0	6.8	90
<code>regress, vce(robust)</code>	2.1	4.1	7.8	14.3	99
<code>replace</code>	2.0	4.0	7.9	15.8	100

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>replace (small expressions)</code>	4.8	9.5	18.6	37.3	100
<code>rerilogistic</code>	1.8	3.3	5.4	8.2	94
<code>rerinbreg</code>	1.8	3.1	4.9	6.9	91
<code>reripoisson</code>	1.8	3.3	5.4	8.1	94
<code>reristcox</code>	1.1	1.2	1.6	1.6	42
<code>reristntcox, interval()</code>	1.7	2.6	3.0	3.3	72
<code>reristntcox, id() time() status()</code>	1.5	2.1	2.6	2.8	66
<code>reristntreg</code>	1.7	2.9	3.8	4.5	82
<code>reriststreg</code>	1.8	3.2	5.2	7.6	93
<code>reshapelong</code>	0.9	1.0	1.1	1.9	53
<code>reshapewide</code>	1.0	1.0	1.0	1.1	7
<code>robvar</code>	1.3	1.4	1.4	1.4	66
<code>rocfit</code>	1.6	2.1	2.4	2.8	66
<code>roctab</code>	1.1	1.4	1.6	1.7	43
<code>rotate</code>	1.0	1.0	1.0	1.0	2
<code>rotatemat</code>	1.0	1.0	1.0	1.0	2
<code>rreg</code>	2.2	4.1	7.3	12.0	97
<code>runttest</code>	1.9	3.1	4.5	5.5	85
<code>scobit</code>	2.0	3.7	6.8	11.6	97

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>scoreplot</code>	1.3	1.9	2.2	2.5	65
<code>screepplot</code>	1.1	1.2	1.2	1.2	16
<code>sdtest1</code>	1.4	1.8	2.1	2.0	60
<code>sdtest2</code>	1.3	1.7	1.9	2.0	55
<code>sdtest, by()</code>	1.4	1.7	1.9	2.1	56
<code>sem, method(adf) (CFA)</code>	2.0	4.0	7.8	15.0	99
<code>sem, method(ml) (CFA)</code>	1.6	2.1	2.8	3.1	72
<code>sem, method(mlmv) (CFA)</code>	1.1	1.2	1.2	1.2	15
<code>sem (SEM latent)</code>	1.6	2.2	2.8	3.1	72
<code>sem (SEM observed)</code>	1.5	2.1	2.6	2.9	70
<code>separate</code>	1.3	1.5	1.8	2.0	52
<code>sfrancia</code>	2.6	3.5	4.3	4.8	94
<code>signrank</code>	2.8	3.2	3.7	4.0	76
<code>signtest</code>	2.0	4.1	7.7	14.0	99
<code>sktest</code>	1.8	3.1	4.4	5.2	85
<code>slogit</code>	1.2	1.5	1.8	1.9	50
<code>sort</code>	1.3	1.6	2.1	2.4	62
<code>spearman</code>	2.4	2.7	2.8	3.1	69
<code>sspace</code>	1.6	2.1	2.9	3.1	72

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>stack</code>	1.4	1.8	2.0	2.3	57
<code>stci</code>	1.1	1.2	2.4	2.6	65
<code>stcox</code>	1.1	1.1	1.1	1.1	10
<code>sterreg</code>	1.1	1.1	1.1	1.1	8
<code>stgen</code>	1.8	2.5	3.2	3.6	76
<code>stintcox, interval()</code>	1.7	2.7	3.4	3.8	78
<code>stintcox, id() time() status()</code>	1.5	2.0	2.2	2.4	60
<code>stintreg, d(exponential)</code>	1.6	2.6	3.6	4.2	80
<code>stintreg, d(weibull)</code>	2.6	4.2	5.9	7.2	90
<code>stir</code>	1.3	1.5	1.7	1.8	47
<code>stmc</code>	2.8	3.3	3.8	4.0	77
<code>by: stmc</code>	2.8	3.4	3.8	4.7	81
<code>stmh</code>	1.3	1.5	1.8	1.9	50
<code>by: stmh</code>	1.1	1.3	1.5	1.6	39
<code>stptime</code>	1.1	1.2	1.3	1.3	26
<code>strate</code>	1.3	1.6	1.7	2.1	54
<code>streg, distribution(exponential)</code>	1.9	3.7	6.6	11.1	97
<code>streg, dist(exp) vce(cluster)</code>	2.0	3.9	7.3	12.6	98
<code>streg, dist(exp) frailty()</code>	2.0	3.7	6.6	10.9	97

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>streg, dist(exp) frailty() shared()</code>	2.0	3.7	6.0	9.2	95
<code>streg, dist(exp) vce(robust)</code>	2.0	3.9	7.4	13.4	99
<code>streg, distribution(ggamma)</code>	2.3	4.5	7.9	12.0	97
<code>streg, dist(ggamma) vce(cluster)</code>	2.2	4.6	8.2	12.0	97
<code>streg, dist(ggamma) vce(robust)</code>	2.1	4.4	8.0	12.6	97
<code>streg, distribution(gompertz)</code>	2.2	4.7	8.1	12.8	97
<code>streg, dist(gompertz) vce(cluster)</code>	2.0	3.9	6.7	10.0	96
<code>streg, dist(gompertz) frailty()</code>	1.9	4.2	7.3	11.2	96
<code>streg, dist(gomp) frailty() shared()</code>	2.1	4.0	6.3	8.3	93
<code>streg, dist(gompertz) vce(robust)</code>	1.9	3.9	6.9	10.4	96
<code>streg, distribution(llogistic)</code>	1.9	3.8	6.9	10.9	97
<code>streg, dist(llogistic) vce(cluster)</code>	2.1	4.2	7.6	12.7	98
<code>streg, dist(llogistic) frailty()</code>	2.0	3.8	6.8	11.3	97
<code>streg, dist(llog) frailty() shared()</code>	1.9	3.9	6.9	10.8	97
<code>streg, dist(llogistic) vce(robust)</code>	2.0	4.1	7.7	13.2	98
<code>streg, distribution(lnormal)</code>	2.0	4.0	6.7	10.0	95
<code>streg, dist(lnormal) vce(cluster)</code>	2.0	3.8	7.0	11.3	97
<code>streg, dist(lnormal) frailty()</code>	2.0	3.7	6.4	9.9	96
<code>streg, dist(lnorm) frailty() shared()</code>	2.0	3.8	5.9	8.1	93

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>streg, dist(lnormal) vce(robust)</code>	2.0	3.8	7.1	11.8	97
<code>streg, distribution(weibull)</code>	2.0	3.9	7.4	13.4	98
<code>streg, dist(weibull) vce(cluster)</code>	2.0	4.2	7.7	12.9	98
<code>streg, dist(weibull) frailty()</code>	2.2	5.1	8.7	13.1	97
<code>streg, dist(weib) frailty() shared()</code>	2.1	3.9	6.4	9.8	95
<code>streg, dist(weibull) vce(robust)</code>	2.0	4.2	7.9	13.9	98
<code>sts generate</code>	1.0	1.1	1.2	1.3	30
<code>sts graph</code>	1.1	1.3	1.3	1.3	28
<code>sts list</code>	1.1	1.3	1.3	1.4	31
<code>sts test</code>	1.1	1.3	1.4	1.5	37
<code>stset</code>	1.4	1.8	2.1	2.3	60
<code>stsplit</code>	1.0	1.1	1.2	1.2	17
<code>stsum</code>	1.1	1.7	1.8	1.7	46
<code>stteffects ipw (weibull)</code>	2.0	3.5	5.2	6.9	90
<code>stteffects ipwra (weibull)</code>	1.8	2.7	3.5	4.3	80
<code>stteffects ra (weibull)</code>	1.7	2.5	3.2	3.6	76
<code>stteffects wra (weibull)</code>	1.8	2.7	3.5	4.0	79
<code>stvary</code>	1.3	1.7	2.0	2.3	60
<code>suest</code>	2.0	3.8	7.1	12.6	98

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
summarize	2.0	4.0	8.0	15.6	100
sunflower	1.3	1.5	1.7	5.2	88
sureg	2.1	3.8	6.2	9.3	94
svar	1.6	1.9	2.1	2.1	56
svmat	1.1	1.1	1.1	1.1	6
svy brr: logit	1.4	1.9	2.3	2.6	65
svy brr: poisson	1.6	2.2	2.8	3.2	74
svy brr: regress	2.1	3.7	6.2	9.3	95
svy jackknife: logit	1.6	2.4	3.2	3.9	79
svy jackknife: poisson	1.5	2.2	2.9	3.5	77
svy jackknife: regress	2.0	3.3	5.0	6.6	90
svy linearized: logit	1.9	3.5	5.6	8.5	94
svy linearized: poisson	2.0	3.6	5.9	8.8	94
svy linearized: regress	1.9	3.2	4.7	6.0	88
swilk	3.5	4.2	5.7	6.2	89
symmetry	1.1	1.3	1.4	1.4	34
table (one-way)	1.1	1.3	1.4	1.6	39
table (two-way)	1.2	1.4	1.4	1.5	35
tabodds	1.0	1.1	1.1	1.1	8

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>tabodds (adjusted)</code>	0.7	0.8	0.9	0.9	0
<code>tabstat</code>	1.2	1.6	1.8	1.8	47
<code>tabstat, by()</code>	1.1	1.3	2.8	2.8	70
<code>tabulate (one-way)</code>	1.0	1.0	1.0	1.0	0
<code>tabulate (two-way)</code>	1.0	1.0	1.0	1.0	0
<code>teffects aipw (linear)</code>	1.9	3.4	5.5	6.6	89
<code>teffects aipw (probit)</code>	1.9	3.2	5.2	6.2	88
<code>teffects ipw (logit)</code>	2.3	4.2	7.0	10.3	94
<code>teffects ipwra (linear)</code>	1.9	3.3	5.3	6.3	88
<code>teffects ipwra (probit)</code>	1.8	3.2	5.0	6.0	87
<code>teffects nnmatch</code>	2.0	4.0	7.7	14.6	99
<code>teffects psmatch, logit</code>	1.1	1.1	1.1	1.1	11
<code>teffects ra (linear)</code>	2.0	3.6	6.1	7.9	91
<code>teffects ra (probit)</code>	1.9	3.4	5.6	7.0	90
<code>telasso (, linear) (, probit), ate</code>	1.5	1.7	1.7	2.0	45
<code>telasso (, linear) (, probit), atet</code>	1.3	1.7	1.9	1.8	49
<code>telasso (, linear) (, probit), pomeans</code>	1.4	1.8	1.8	2.1	48
<code>telasso (, logit) (, probit), ate</code>	1.7	2.8	3.5	3.9	77
<code>telasso (, logit) (, probit), atet</code>	1.7	2.3	2.9	3.3	73

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
telasso (, logit) (, probit), pomeans	1.7	2.6	3.3	3.7	75
telasso (, poisson) (, probit), ate	1.5	2.4	3.2	3.6	77
telasso (, poisson) (, probit), atet	1.6	2.4	3.2	3.7	78
telasso (, poisson) (, probit), pomeans	1.6	2.5	3.1	3.6	77
telasso (, probit) (, probit), ate	1.6	2.3	3.0	3.4	74
telasso (, probit) (, probit), atet	1.6	2.5	3.1	3.5	75
telasso (, probit) (, probit), pomeans	1.6	2.4	2.9	3.4	74
tetrachoric	1.2	1.3	1.5	1.5	40
threshold, threshvar()	1.1	1.1	1.1	1.0	1
threshold, threshvar() regionvars()	1.0	1.0	1.2	1.1	13
tnbreg	1.3	2.4	2.9	3.4	75
tobit	1.9	3.3	5.1	7.0	91
tostring	1.0	1.0	1.1	1.1	11
total	2.0	3.8	7.0	11.9	97
tpoisson	1.9	3.4	5.6	8.0	94
truncreg	1.9	3.3	5.2	7.2	92
tsfilter bk	1.1	1.1	1.2	1.2	16
tsfilter bw	1.4	1.8	2.1	2.3	61
tsfilter cf	1.1	1.1	1.1	1.2	14

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>tsfilter hp</code>	1.4	1.8	2.1	2.3	61
<code>tsrevar</code>	2.3	4.7	8.7	14.7	94
<code>tsset</code>	1.2	1.6	1.8	2.0	55
<code>tssmooth exp</code>	1.3	1.5	1.6	1.7	44
<code>tssmooth ma</code>	1.2	1.3	1.3	1.4	28
<code>ttest1</code>	1.4	1.8	2.0	2.2	59
<code>ttest2</code>	1.7	2.4	3.0	2.6	73
<code>ttest, by()</code>	1.4	1.7	2.0	2.1	57
<code>twoway fpfit</code>	1.3	2.2	4.2	4.6	85
<code>twoway lfitci</code>	1.2	1.4	1.6	1.7	43
<code>twoway mband</code>	2.3	3.5	4.4	5.0	83
<code>twoway mspline</code>	2.3	3.3	4.3	4.7	82
<code>ucm, model(rwdrift)</code>	1.2	1.3	1.2	1.3	20
<code>var</code>	1.4	1.7	1.6	4.0	80
<code>vargranger</code>	1.4	1.4	1.4	1.4	28
<code>varlmar</code>	1.4	1.8	2.0	2.8	66
<code>varnorm</code>	1.2	1.7	2.1	2.2	60
<code>varsoc</code>	1.2	1.6	1.8	2.0	55
<code>varstable</code>	1.0	1.1	1.1	1.2	16

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>vec</code>	1.2	1.4	1.6	1.7	41
<code>veclmar</code>	1.0	1.2	1.4	1.5	33
<code>vecnorm</code>	1.3	1.6	1.7	3.1	72
<code>vecrank</code>	1.0	1.3	1.5	1.6	40
<code>vecstable</code>	1.0	1.2	1.2	1.2	16
<code>vwls</code>	2.2	4.3	7.9	13.6	98
<code>wildbootstrap areg</code>	1.7	2.3	2.7	3.3	74
<code>wildbootstrap regress</code>	1.6	2.0	2.7	2.8	65
<code>wildbootstrap xtreg</code>	2.2	3.6	5.1	8.0	91
<code>wntestb</code>	1.0	1.0	1.0	1.0	0
<code>wntestq</code>	2.0	2.1	2.1	2.1	52
<code>xcorr</code>	1.1	1.1	1.1	1.1	7
<code>xpologit</code>	1.2	1.3	1.4	1.4	30
<code>xpopoisson</code>	1.1	1.3	1.4	1.4	30
<code>xporegress</code>	1.2	1.3	1.4	1.4	30
<code>xtabond</code>	1.0	1.3	1.4	1.5	38
<code>xtabond, twostep</code>	1.0	1.2	1.3	1.5	34
<code>xtcloglog, re</code>	2.1	3.9	6.6	10.3	95
<code>xtdata, be</code>	1.9	2.3	2.4	2.5	62

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>xtdata, fe</code>	2.9	4.1	4.8	5.4	84
<code>xtdata, re</code>	3.1	4.2	4.8	5.4	84
<code>xtdidregress</code>	1.6	2.4	3.0	3.7	78
<code>xtdpd</code>	1.2	1.4	1.5	1.5	37
<code>xtdpdsys</code>	1.0	1.3	1.3	1.4	30
<code>xteregress</code>	1.4	1.7	1.8	1.8	44
<code>xtfrontier</code>	2.5	4.3	6.7	9.0	93
<code>xtgee, family(gaussian) corr(ar2)</code>	1.5	1.8	2.0	2.1	55
<code>xtgee, fam(gauss) corr(unstruct)</code>	1.5	1.8	2.0	2.1	56
<code>xtcloglog, pa</code>	1.6	2.4	3.2	3.9	79
<code>xtlogit, pa</code>	1.3	1.7	1.9	2.1	58
<code>xtnbreg, pa</code>	1.6	2.2	2.8	3.2	73
<code>xtpoisson, pa</code>	1.5	2.0	2.4	2.7	67
<code>xtprobit, pa</code>	1.3	1.6	1.8	2.0	57
<code>xtreg, pa</code>	1.3	1.6	1.8	1.9	51
<code>xtgls</code>	1.3	1.5	1.7	1.9	49
<code>xthdidregress aipw</code>	1.2	1.4	1.6	1.6	41
<code>xthdidregress ipw</code>	1.2	1.4	1.5	1.6	39
<code>xthdidregress ra</code>	1.2	1.4	1.5	1.5	34

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>xthdidregress twfe</code>	1.2	1.3	1.3	1.4	28
<code>xthtaylor</code>	1.3	2.2	3.2	4.0	81
<code>xtile</code>	1.0	1.0	1.0	1.0	1
<code>xtintreg</code>	2.0	3.8	6.6	10.7	96
<code>xtivreg, be</code>	1.9	2.5	3.1	3.5	75
<code>xtivreg, fd</code>	2.2	2.6	3.1	3.3	73
<code>xtivreg, fe</code>	1.9	2.8	3.2	3.4	74
<code>xtivreg, re</code>	1.9	2.7	3.3	3.5	76
<code>xtlogit, fe</code>	1.5	2.2	2.8	3.1	72
<code>xtlogit, re</code>	2.2	3.7	5.6	7.4	91
<code>xtnlogit, fe</code>	1.0	1.0	1.1	1.1	6
<code>xtnlogit, re</code>	1.9	3.0	4.2	5.1	85
<code>xtnbreg, fe</code>	3.1	5.6	8.3	9.8	93
<code>xtnbreg, re</code>	2.9	4.8	7.0	8.9	92
<code>xtologit</code>	1.9	2.8	3.7	4.3	80
<code>xtoprobit</code>	1.9	2.9	3.9	4.6	82
<code>xtpcse</code>	1.3	1.6	1.7	1.8	46
<code>xtpoisson, fe</code>	3.0	4.9	7.3	9.2	92
<code>xtpoisson, re</code>	2.9	5.7	10.0	15.1	97

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>xtprobit, re</code>	2.1	3.5	5.9	8.8	94
<code>xtrc</code>	1.7	2.6	3.3	3.9	78
<code>xtreg, be</code>	1.5	2.0	2.3	2.5	64
<code>xtreg, fe</code>	2.0	3.4	5.5	8.2	93
<code>xtreg, fe vce(robust)</code>	1.9	3.5	5.9	8.9	95
<code>xtreg, mle</code>	1.3	1.8	3.9	4.1	81
<code>xtreg, re</code>	2.2	3.1	3.7	4.1	78
<code>xtregar, fe</code>	1.8	3.2	4.6	4.8	81
<code>xtregar, re</code>	1.7	2.5	2.8	3.0	70
<code>xtset</code>	1.2	1.4	1.5	1.6	42
<code>xtstreg, distribution(exponential)</code>	1.8	2.5	3.2	3.5	75
<code>xtstreg, distribution(weibull)</code>	1.9	2.8	3.6	4.2	80
<code>xtsum</code>	1.6	2.5	3.3	3.7	78
<code>xttab</code>	1.2	1.5	1.7	1.9	53
<code>xttobit</code>	1.9	3.9	6.4	9.3	94
<code>xtunitroot breitung</code>	1.1	1.3	1.4	1.6	39
<code>xtunitroot fisher</code>	1.0	1.1	1.1	1.1	8
<code>xtunitroot hadri</code>	1.3	1.3	1.4	1.4	29
<code>xtunitroot ht</code>	1.3	1.6	2.0	2.2	58

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 1. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
<code>xtunitroot ips</code>	1.0	1.0	1.0	1.0	3
<code>xtunitroot llc</code>	0.9	1.0	1.0	1.0	0
<code>zinb</code>	2.0	4.2	7.4	11.9	97
<code>ziologit</code>	1.9	3.2	4.7	6.0	88
<code>zioprobit</code>	1.9	3.2	5.0	6.8	91
<code>zip</code>	1.9	3.8	7.1	12.3	98
<code>_predict, xb</code>	2.1	4.3	8.3	16.4	100
<code>_rmcoll</code>	2.1	4.2	8.3	15.7	100
<code>_robust</code>	2.0	3.9	7.7	14.4	99

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Nine of the lines in table 1 represent estimation commands run on survey data. Each of these commands begins with `svy`. These are only a few of the estimation commands that support estimation on survey data, but we can make some generalizations about how the three primary methods of estimation with survey data will perform with other estimation commands. With the linearization method, prefix `svy linearized`, estimation commands will be parallelized just as well and sometimes better than they were parallelized on non-survey data. This is true because the linearization computation is itself almost 100% parallelized. When using the balanced repeated replications (BRR) method, `svy brr`, or the jackknife method, `svy jackknife`, almost all estimation commands are slightly less parallelized. The BRR and jackknife VCE computations are not themselves parallelized, but the overall estimation time is dominated by standard estimation.

More than a full page of table 1 is dedicated to performance when using multiple imputation (MI). These commands begin with the `mi` prefix. All the results in the table are from problems with five imputations. The number of imputations does not affect parallelization performance much. As with all commands, problems with more observations and covariates are better parallelized; see appendix D for the sizes of problems used to assess performance.

There are two particularly computationally intensive aspects to using MI data—creating the MI datasets (imputation) and estimation. The table reports the results for all the primary methods of imputation; these lines are prefixed with `mi impute`. It also reports the results for four representative estimators—linear regression (`regress`), logistic regression (`logit`), multinomial logistic regression (`mlogit`), and ordered logistic regression (`ologit`).

Performance on MI data is affected by the style in which the MI data are stored. Stata allows four styles for storing MI data: wide (`wide`), marginal long (`mlong`), full long (`flong`), and full long and separate (`flongsep`). Each style has advantages with regard to storage required and ease of use in some analyses.

With regard to imputation performance under Stata/MP, imputation is fastest and most parallelized when using style `flongsep`. `flongsep` is the native style in which imputations are performed. Table 1 reports performance across all MI storage styles for only the `logit` imputer; the relative performance of the styles is similar for other imputers.

Estimation is fastest and most parallelized when using storage style `wide`, although style `flongsep` is also well parallelized. Style `wide` is fastest because the overhead for managing `wide` data mostly involves simply changing the names of variables. The table reports estimation results in all four MI storage styles for only `regress` and `logit`. The relative performance is similar for other estimators.

As with estimation using survey data, MI can be applied to many more estimation commands than those listed in the table. Only some of the MI computations are themselves parallelized, so most commands are less parallelized when used with MI data, regardless of the style in which the data are stored. Computationally intensive estimation commands that involve iterative solutions, such as logistic regression, are less affected than are commands with closed-form solutions, such as linear regression.

For maximum performance using Stata/MP, set the MI storage style to `flongsep` when performing imputations and to `wide` when performing estimations. The short time you invest to convert between styles will be more than repaid in faster imputation and estimation. If you have insufficient memory to store an MI dataset in style `wide`, then continue to use `flongsep` during estimation.

When using many imputations on moderate-to-small problems, the overhead of the MI computations can dominate the time required. Such problems are less parallelized than reported in the table. Conversely, very large problems with few imputations are parallelized even more than reported in the table.

9 Performance variability across computing platforms

As discussed in sections 3 and 4, multicore/multiprocessor performance will vary across computing platforms for many reasons. Those reasons include differences in how operating systems partition tasks, how processors pipeline and partition instructions, how memory is accessed, and how onboard processor cache is handled.

Stata/MP performance has been tested on dozens of different platforms, including different processors (both Intel and AMD), different cache architectures, different operating systems (including Microsoft Windows, Mac OS X for Intel, Linux, and Oracle 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 appendixes A and B.

It is not helpful 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 each core of a processor to masquerade as two cores. The operating system and other applications see each physical core as two virtual cores and treat each just as they would any physical core. Intel achieves performance improvements primarily because main computer memory is slow compared with the processor and its onboard cache memory. When the execution thread of one process must wait for something from main memory, the thread for another process can execute. The effect is clearly not the same as having two cores, but for many applications, performance can be improved by treating a computer with a hyperthreaded processor as having twice as many cores as it actually has.

Stata/MP runs on hyperthreaded processors.

Most Stata commands are computationally intense and Stata/MP has been optimized to access main memory efficiently. For these reasons, we would not expect hyperthreading to substantially improve the performance of most commands. Our timings indicate that this is true for most Stata commands, but a few performance gains were surprisingly good.

Figure 13 presents the now familiar boundary region and median performance of Stata/MP running on a quad-core computer with hyperthreading – making for 8 virtual cores. Through the first 4 cores, performance is almost identical to what we saw in Figure 7 for a non-hyperthreaded processor. That is to say, so long as we do not exceed the number of physical cores on a system, hyperthreaded computers behave just like non-hyperthreaded computers.

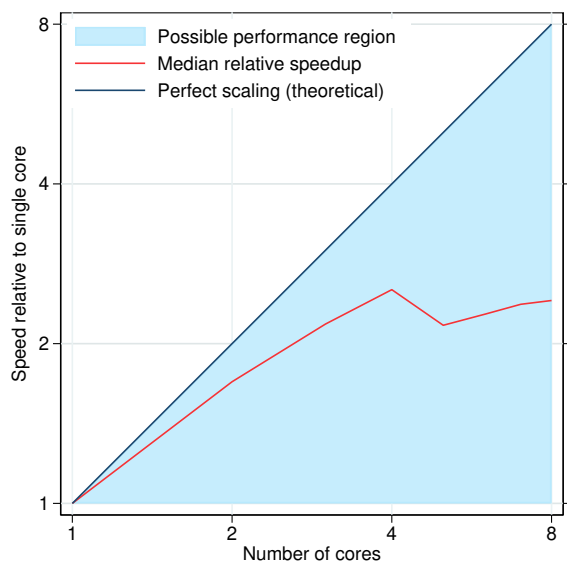


Figure 13. **Performance of Stata/MP on hyperthreaded CPUs.** Speed on multiple cores relative to speed on a single core.

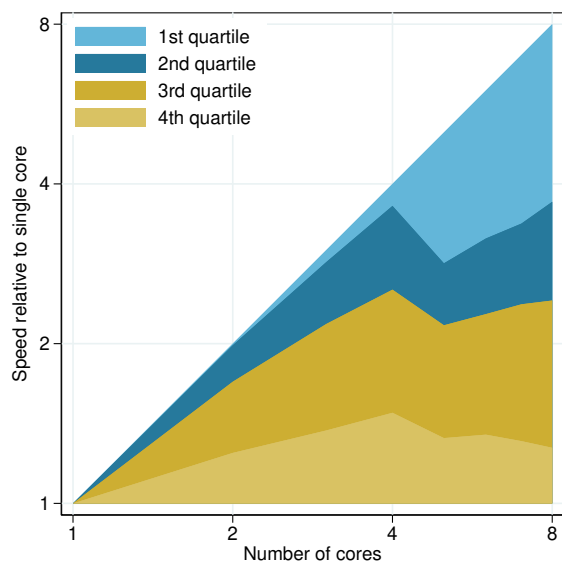


Figure 14. **Quartiles of Stata/MP Performance on hyperthreaded CPUs.** Speed on multiple cores relative speed on a single core.

Above 4 cores, median performance drops for 5 cores, one of them virtual, but improves to approaching the performance of 4 physical cores. The most interesting point beyond 4 cores is 8 cores – all of the virtual cores on the computer. The median relative speed with 8 cores is 2.4, which is slightly less than the median speed of 2.5 for the 4 physical cores.

Figure 14 presents the quartiles of command performance. The diagonal top of the light-blue region indicates that at least one command has perfect parallelization over all 8 virtual cores. Moreover, for the 25% of commands that perform best with hyperthreading their relative speed is at least 3.7 with all 8 virtual cores as compared to 3.6 with 4 physical cores — a 2% improvement. At the other end of the spectrum, for the 25% take the least advantage of hyperthreading, the performance on 8 virtual cores is worse than that of 4 physical cores.

By way of caution, Stata/MP has not been evaluated on a wide range of single-processor hyper-threaded 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 processors #
```

where # is the number of CPUs on the computer. Here we use “CPU” to mean a physical core on the computer and not a virtual core created by hyperthreading. So, we could equivalently say, where # is the number of physical cores 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 15 and 16 show the results of two commands run on an 4-processor computer, each hyper-threaded, giving the appearance of 8 virtual processors.

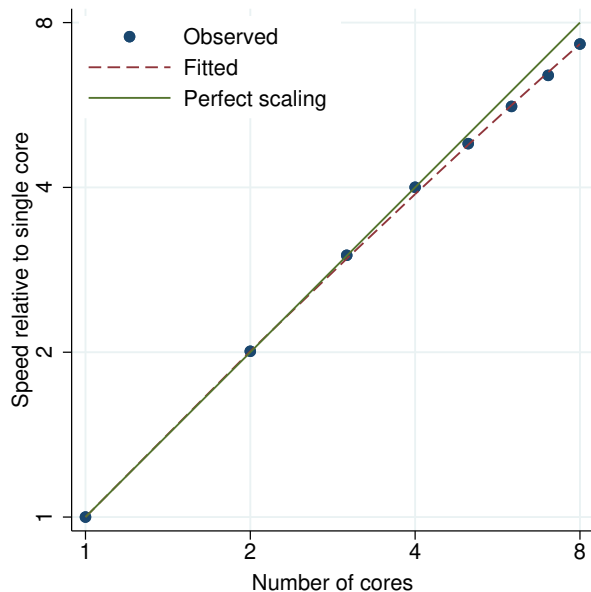


Figure 15. `predict, leverage` performance plot on computers with hyperthreaded CPUs.

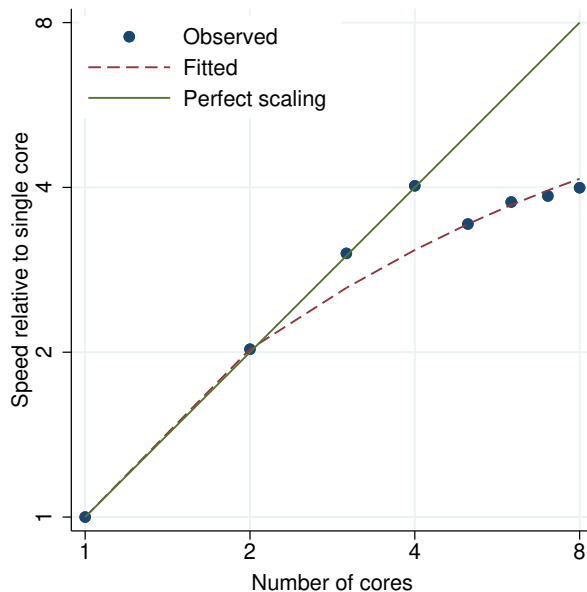


Figure 16. `regress` performance plot on computers with hyperthreaded CPUs.

The `predict, leverage` command, however, is an exception to this recommendation. This command remains nearly perfectly parallelized through all 8 processors (half of which are virtual).

Most commands do not exhibit results like this, and `regress` is an example. Beyond the number of real CPUs, performance actually degrades. This occurs because each CPU has only one FPU, and `regress`, 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.

A Performance assessment graphs for desktop computers

The performance of Stata/MP as reported in columns 2, 3, and 4 of table 1 is presented graphically below, along with the modeled performance from equation 1 and a line representing perfectly scalable performance.

Figures 17 and 18 show two typical graphs. As with table 1, performance is measured as the speed of executing the command on multiple cores relative to the speed on a single core.

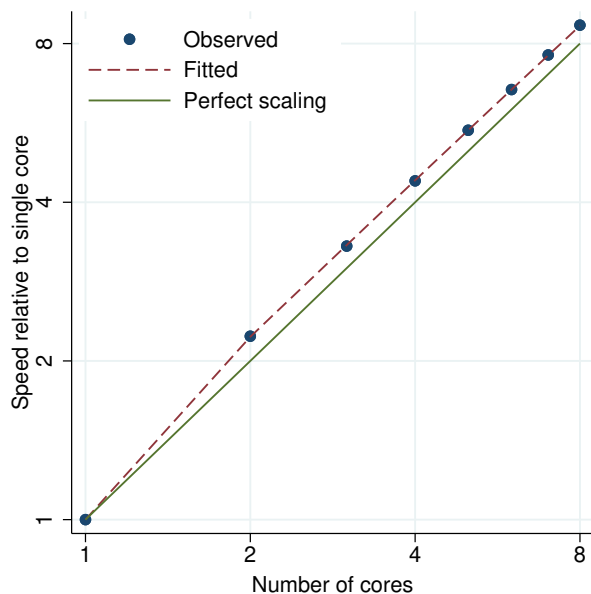


Figure 17. `regress` performance plot.

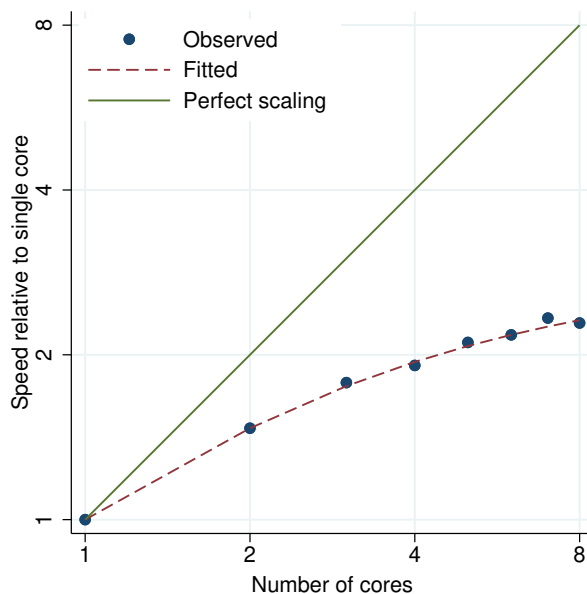


Figure 18. `clogit` (1 to k matching) performance plot.

For a perfectly scalable command, the speed doubles each time the number of cores is doubled. This type of scalability is linear when the number of cores and the relative speed are graphed on a logarithmic scale, which is the scale used in these graphs. Perfect scaling is shown on each graph as a green line that diagonally bisects the graph.

Linear regression, shown in figure 17, is nearly perfectly scalable. Both the observed values and the modeled performance are nearly on the perfect-scalability reference line. The speed is doubled each time the number of cores doubles.

As shown in figure 18, conditional logistic regression clearly performs better as the number of cores increases, but not as much better as linear regression. From table 1, we can see that `clogit` (1 to k matching) is 65% parallelized as compared with 100% for `regress`. From figure 18, we see that `clogit` run with 2 cores on a large dataset is 1.5 times faster than when run with one core; with 4 cores, this relative speed climbs to 1.9; and with 8 cores, it climbs further to 2.3.

Figure 8, from section 7, summarizes the information from all the graphs in this section by placing the observed performance for each command into one of the performance quartiles on the graph.

In a few of the graphs that follow, the observed performance exceeds the theoretical limit of perfect scaling—some of the relative speeds are above the diagonal perfect-scaling line. An example of this can be seen when the `replace` command is evaluating small expressions, as shown in figure 19.

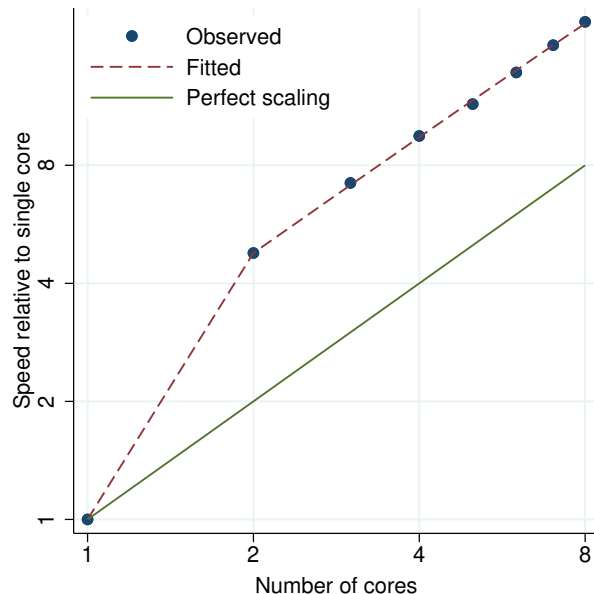


Figure 19. `replace` performance plot.

This phenomenon is nothing more than a cache effect. Cache is very high speed memory that processors use to store data and code that they use often or expect to use often. Cores run much faster when the data they need can be found in cache rather than in standard memory. The `replace` command above was able to find far more of the data it needed in cache when running on 2 or more cores than it could find when running on a single core. The model that we used to determine percentage parallelized ignored that cache effect and correctly determined that the `replace` command was just under 100% parallelized, not greater than 100%.

Observant readers will have noted that the `regress` command in figure 17 exhibited some mild cache effects. Its observed performance is slightly above perfect scaling.

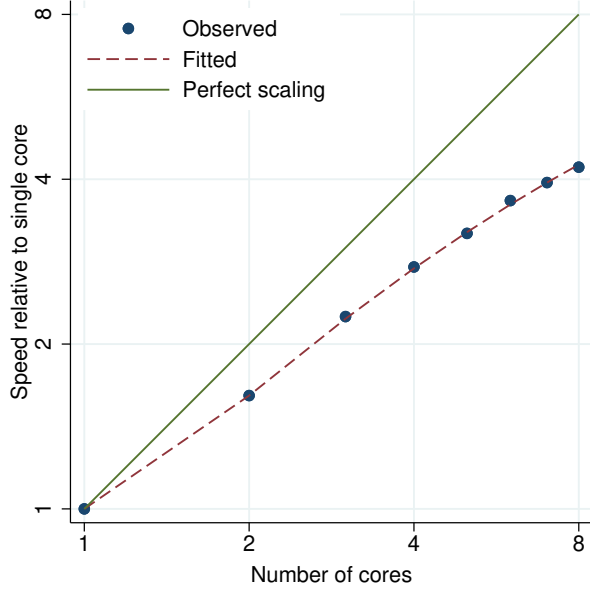


Figure 20. alpha performance plot.

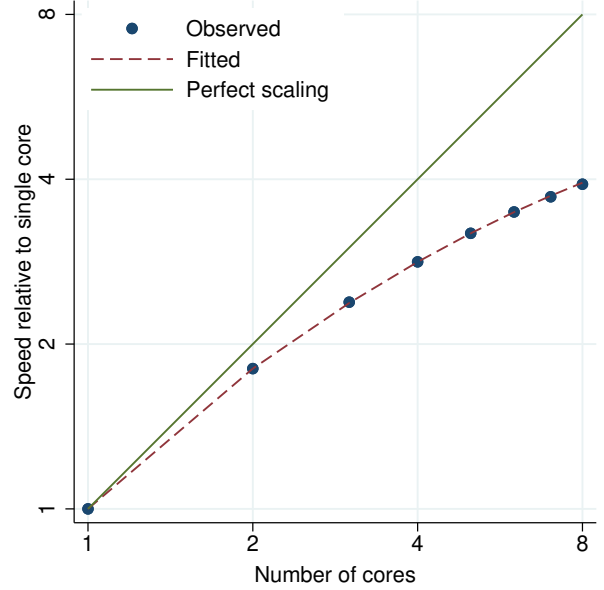


Figure 21. amean performance plot.

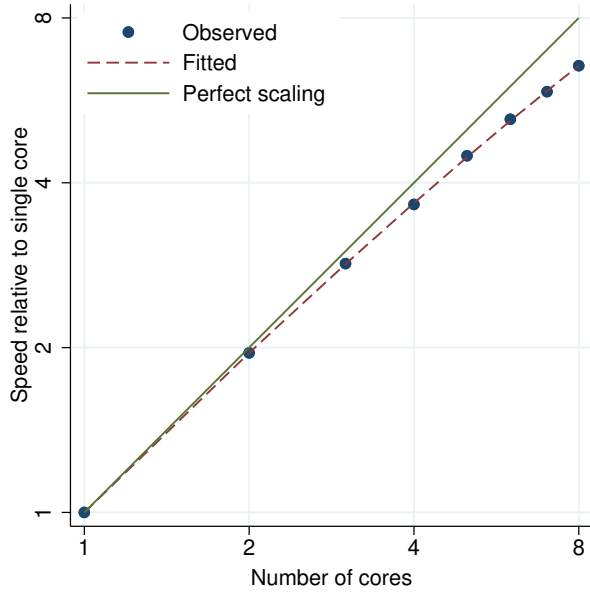


Figure 22. anova (one-way) performance plot.

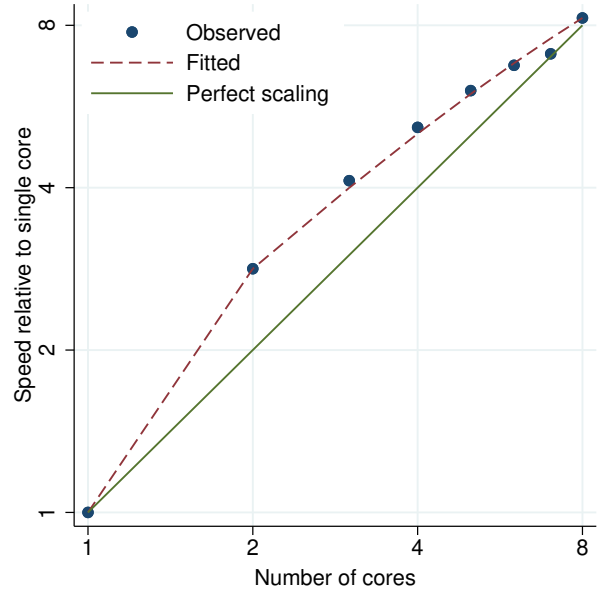


Figure 23. anova (two-way) performance plot.

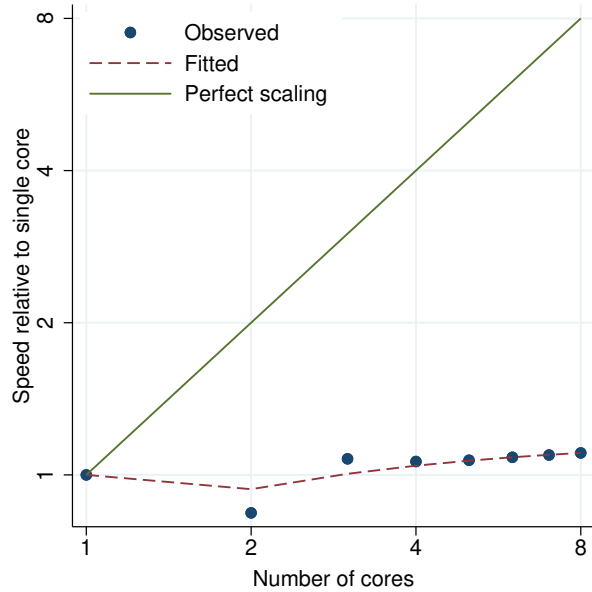


Figure 24. arch performance plot.

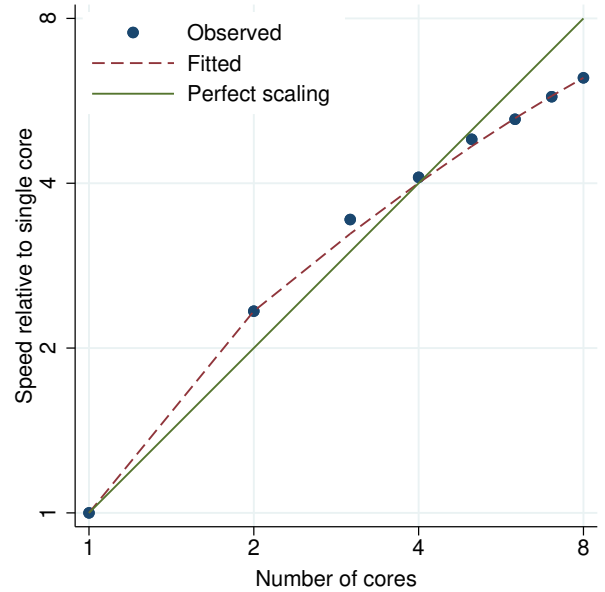


Figure 25. areg performance plot.

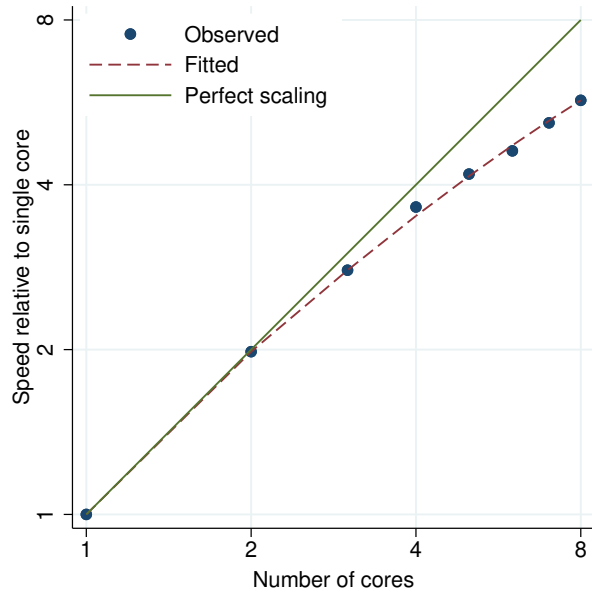


Figure 26. areg, vce(cluster) performance plot.

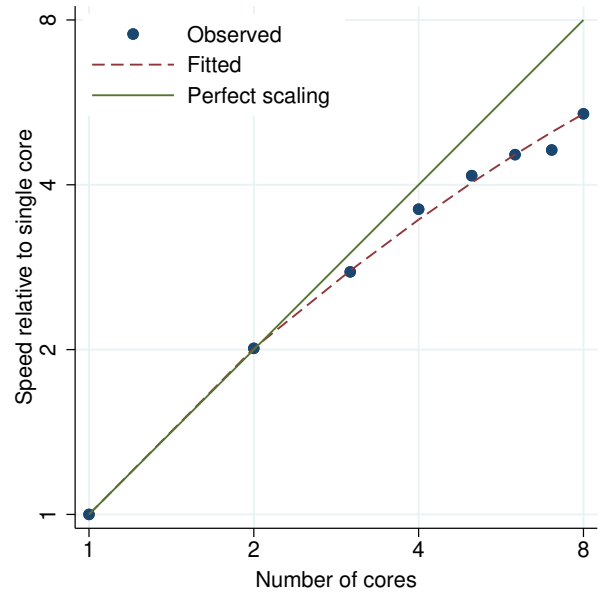


Figure 27. areg, vce(robust) performance plot.

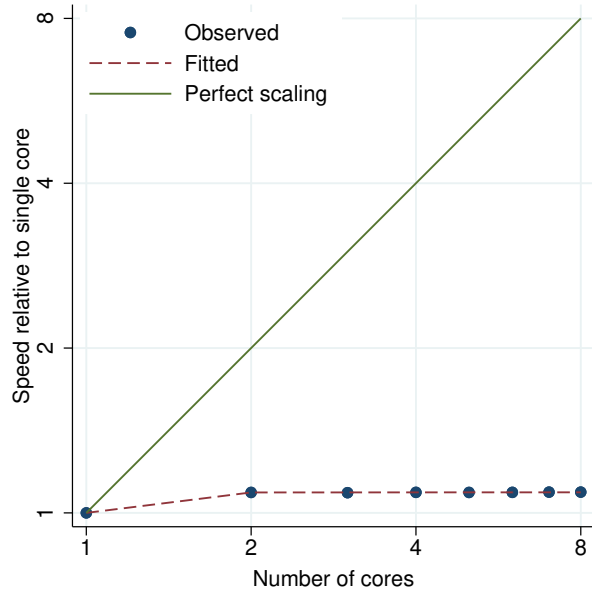


Figure 28. arfima performance plot.

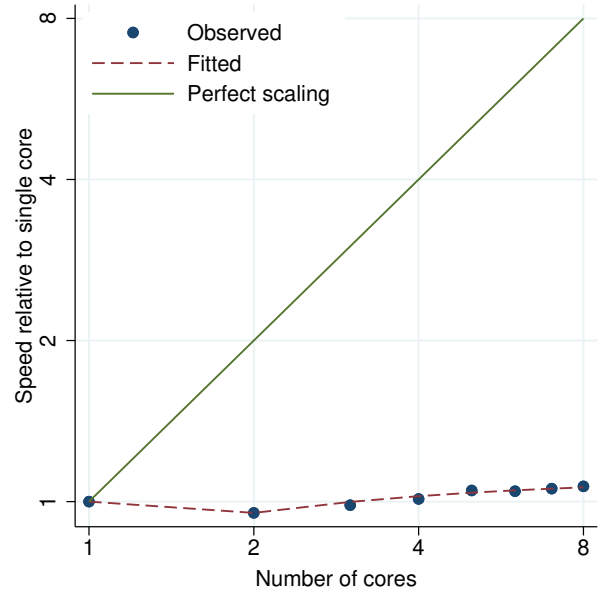


Figure 29. arima performance plot.

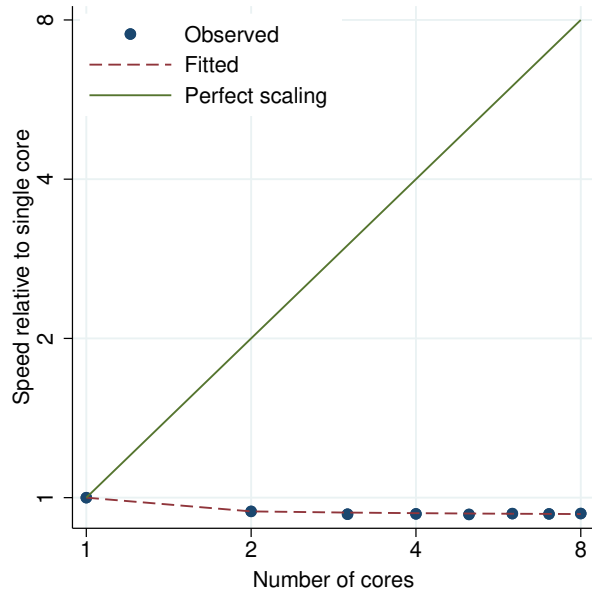


Figure 30. bayes dsge performance plot.

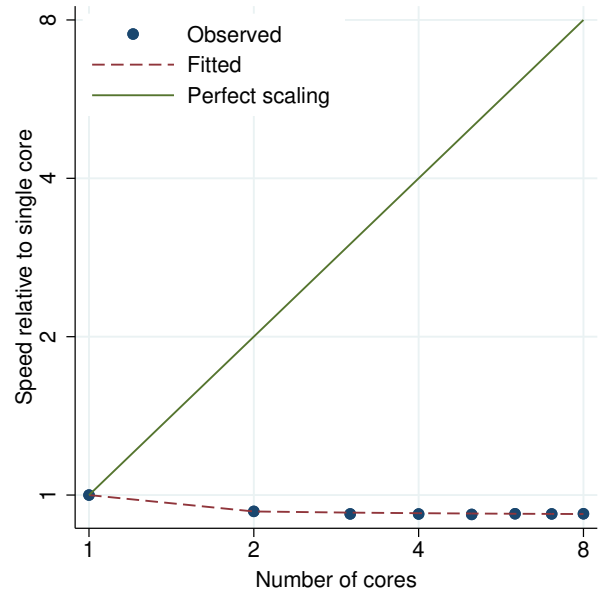


Figure 31. bayes dsge1 performance plot.

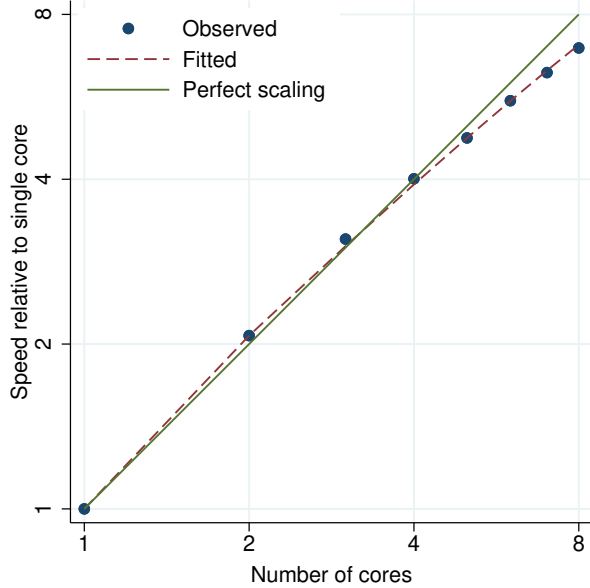


Figure 32. bayes: logit performance plot.

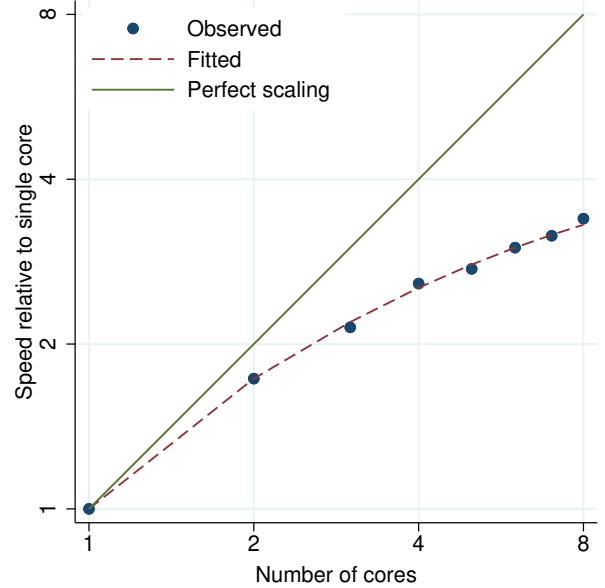


Figure 33. bayes: poisson performance plot.

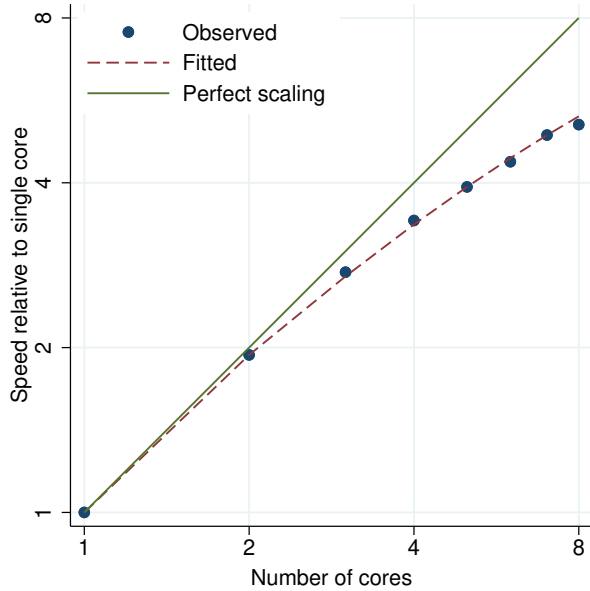


Figure 34. bayes: regress performance plot.

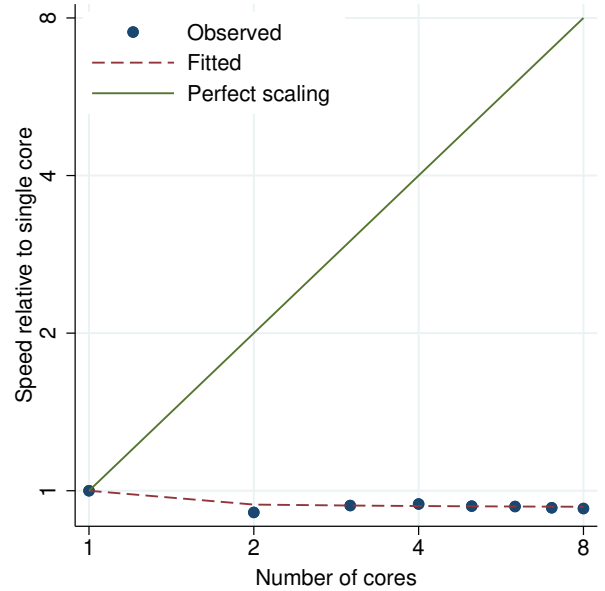


Figure 35. bayes var performance plot.

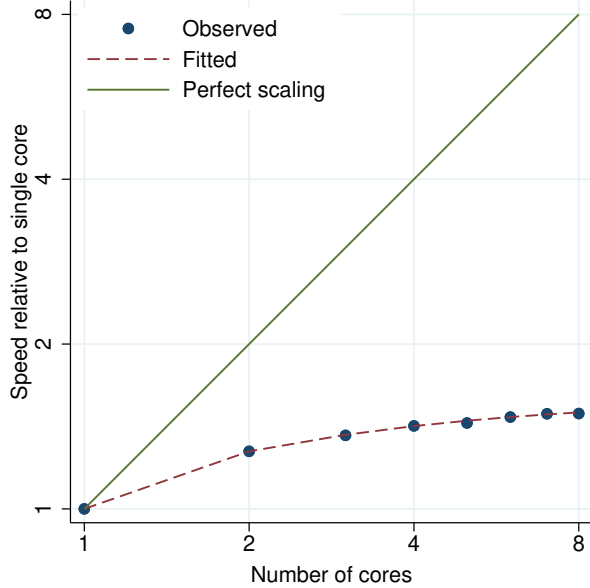


Figure 36. bayesmh logit performance plot.

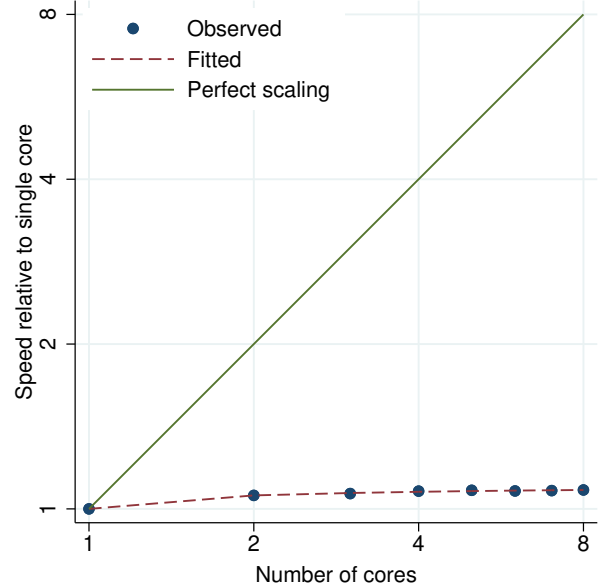


Figure 37. bayesmh mvn performance plot.

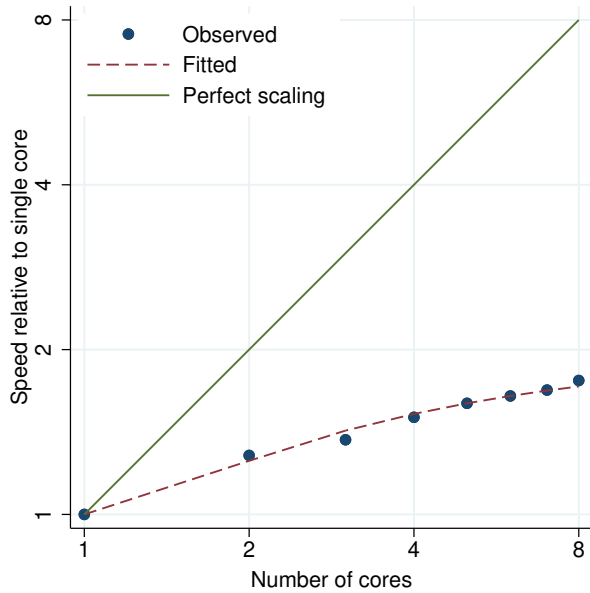


Figure 38. bayesmh mylogit performance plot.

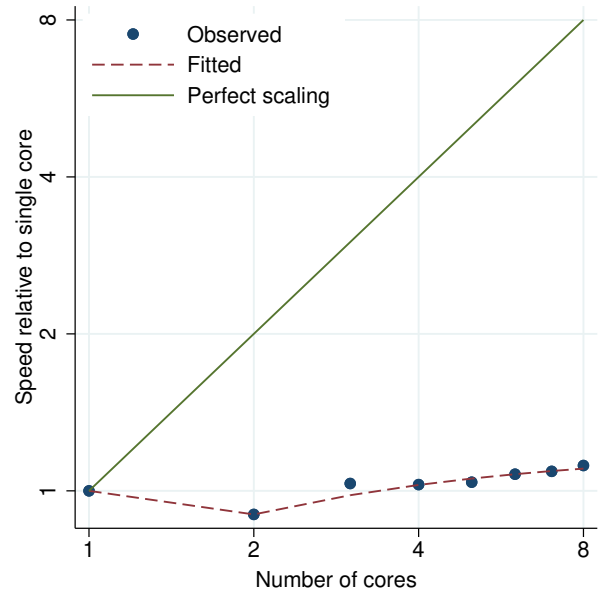


Figure 39. bayesmh nl performance plot.

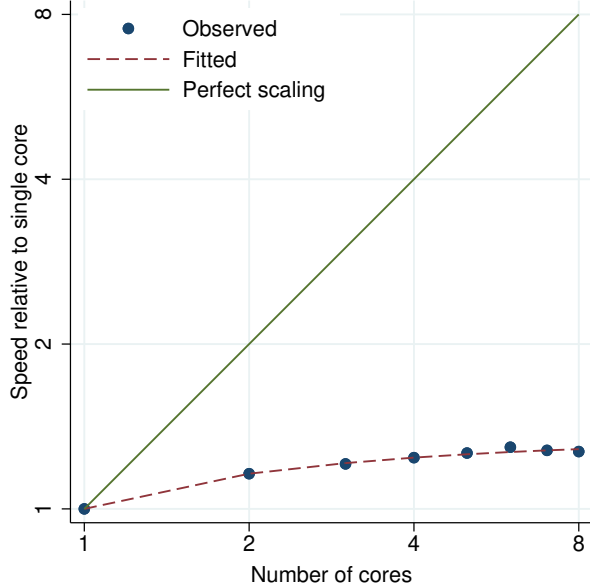


Figure 40. bayesmh normal performance plot.

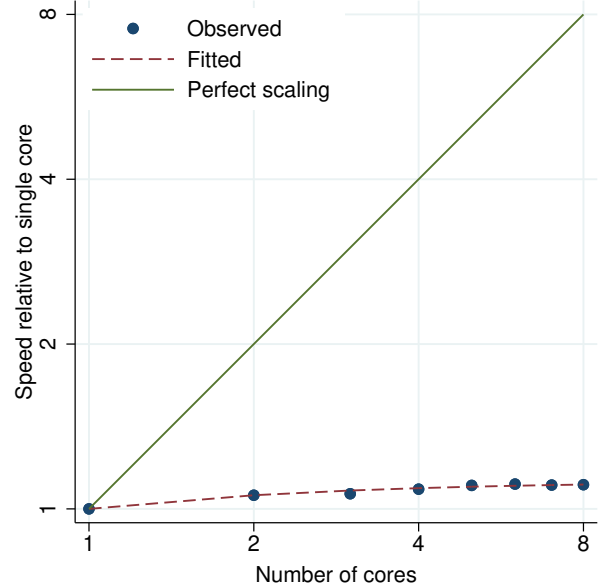


Figure 41. bayesmh normal gibbs performance plot.

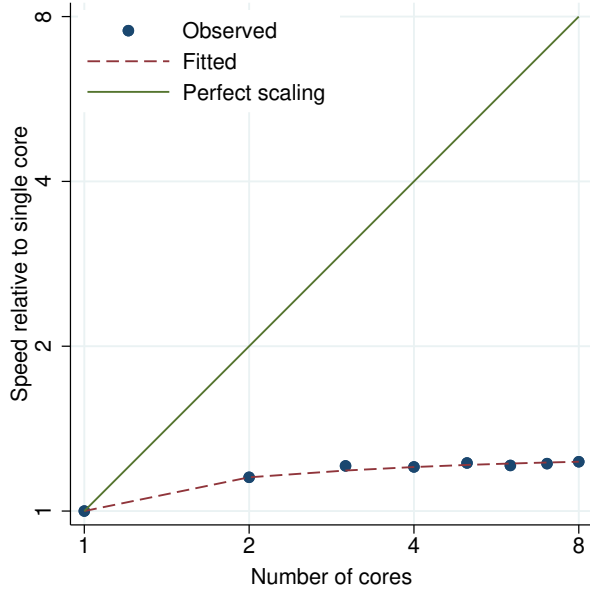


Figure 42. bayesmh normal re performance plot.

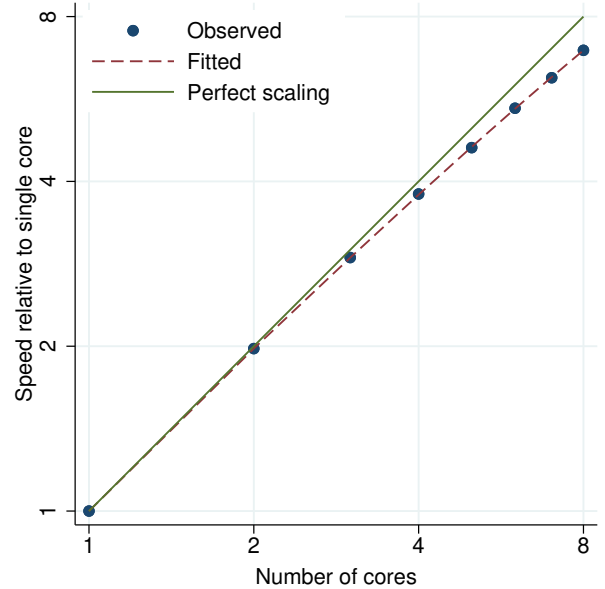


Figure 43. betareg, link(logit) performance plot.

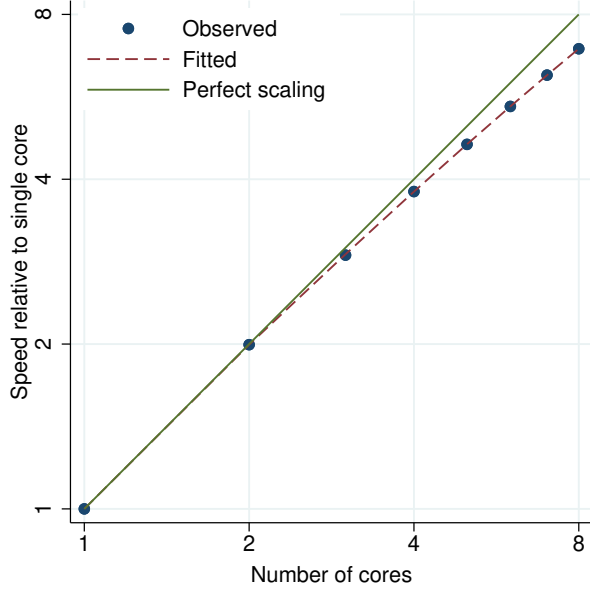


Figure 44. `betareg, link(probit)` performance plot.

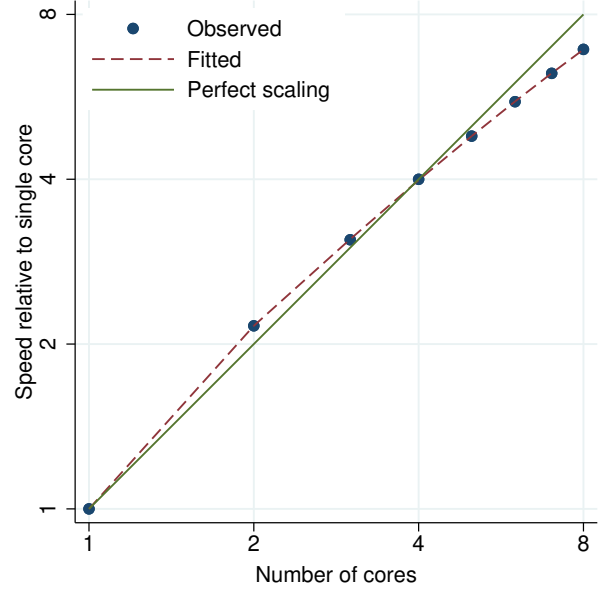


Figure 45. `binreg` performance plot.

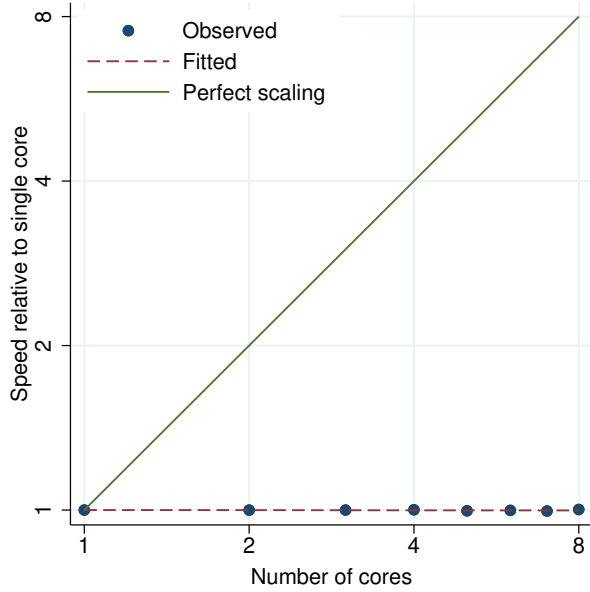


Figure 46. `biplot` performance plot.

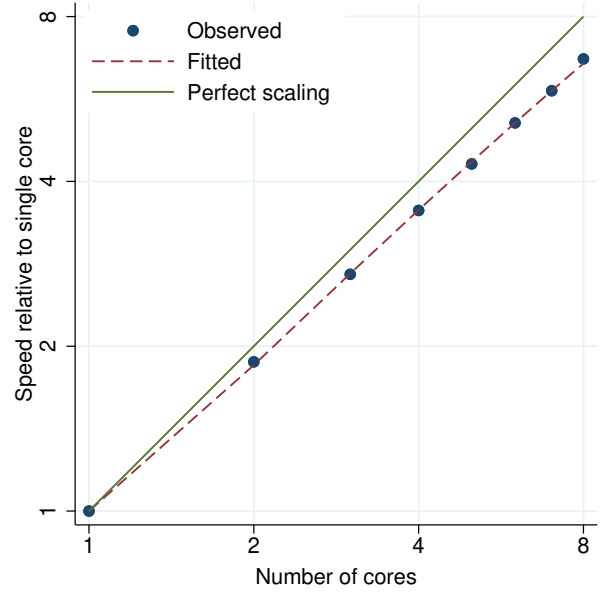


Figure 47. `biprobit` performance plot.

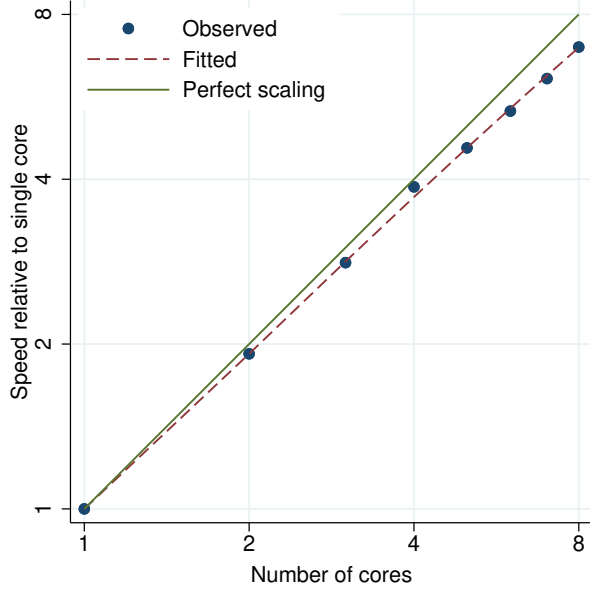


Figure 48. biprobit (seemingly unrelated) performance plot.

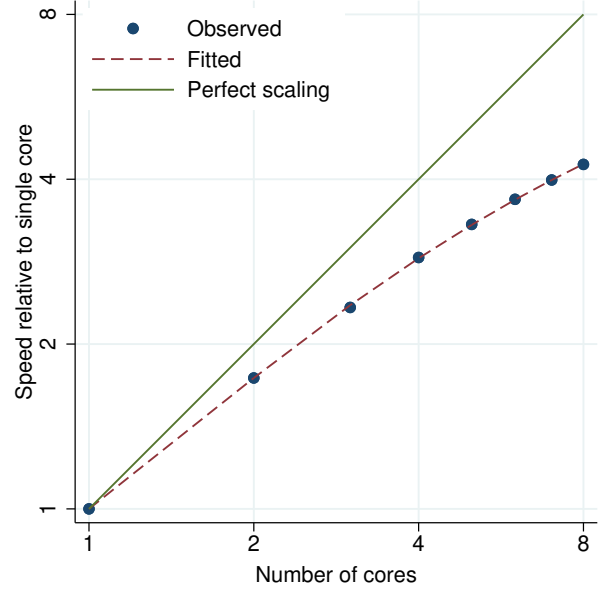


Figure 49. bitest performance plot.

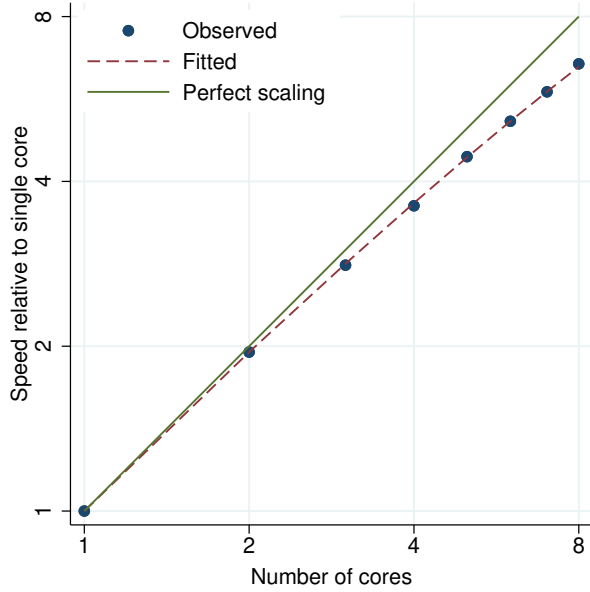


Figure 50. blogit performance plot.

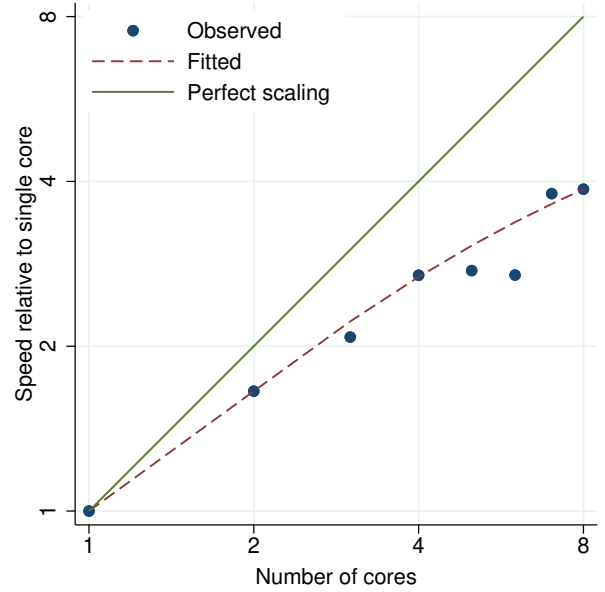


Figure 51. bmaregress performance plot.

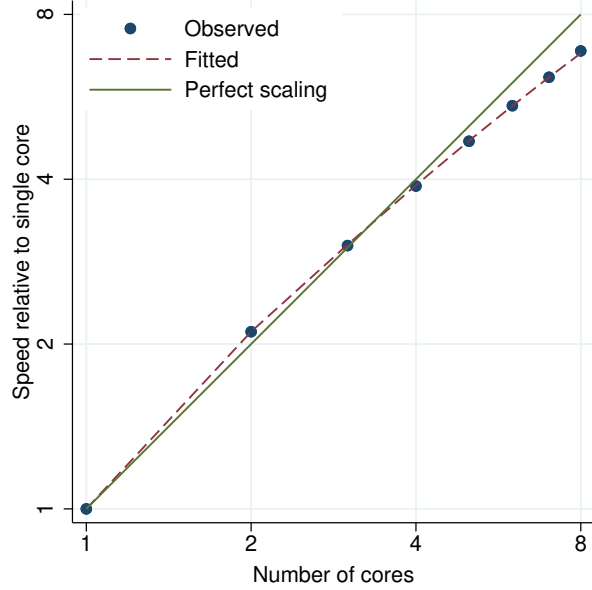


Figure 52. boxcox performance plot.

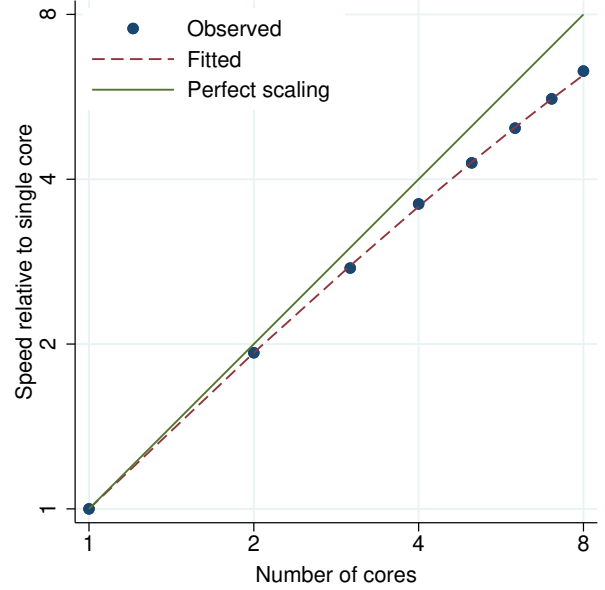


Figure 53. bprobit performance plot.

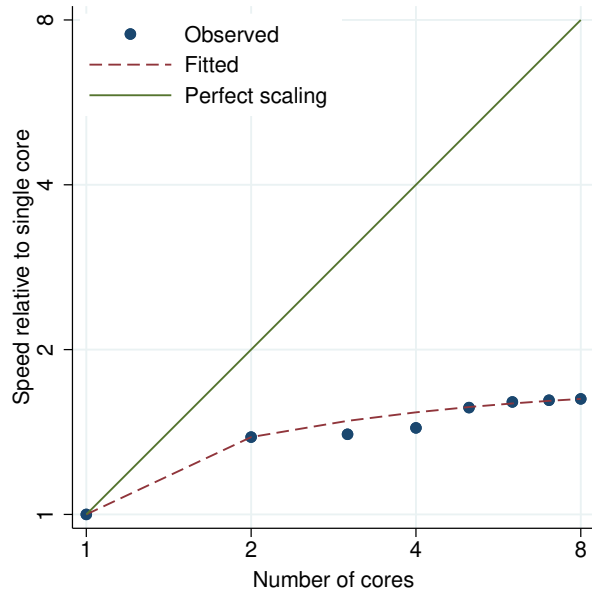


Figure 54. brier performance plot.

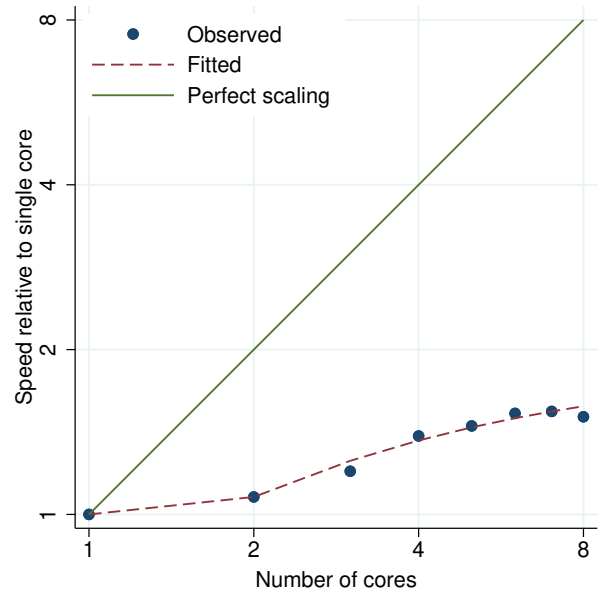


Figure 55. bsample performance plot.

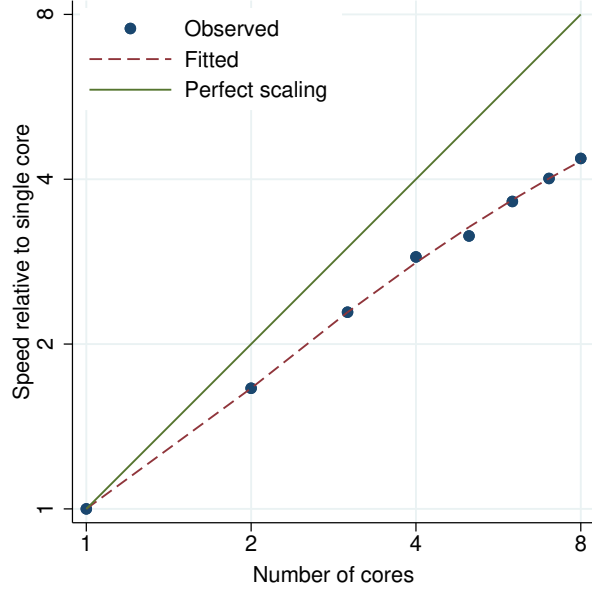


Figure 56. bstat performance plot.

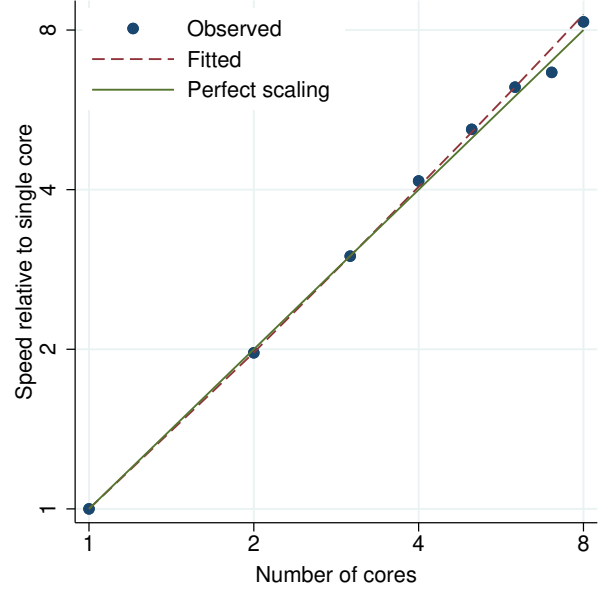


Figure 57. by: generate performance plot.

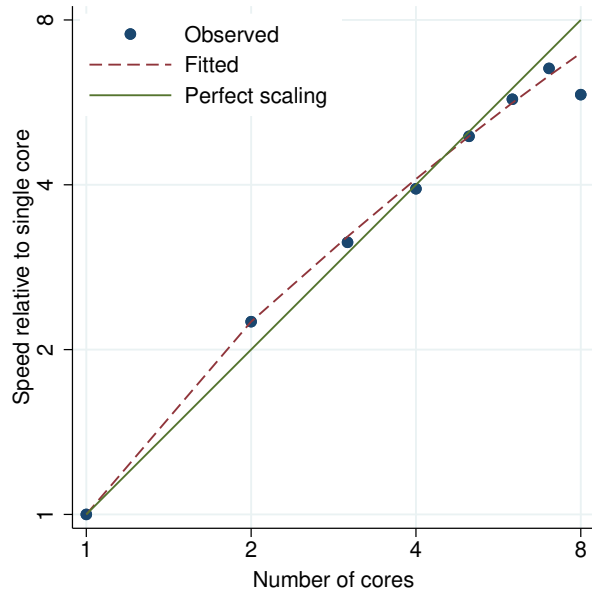


Figure 58. by: generate (small groups) performance plot.

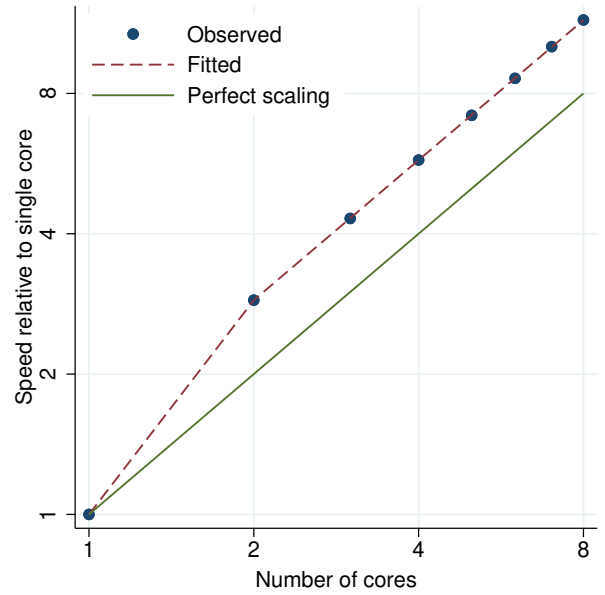


Figure 59. by: replace performance plot.

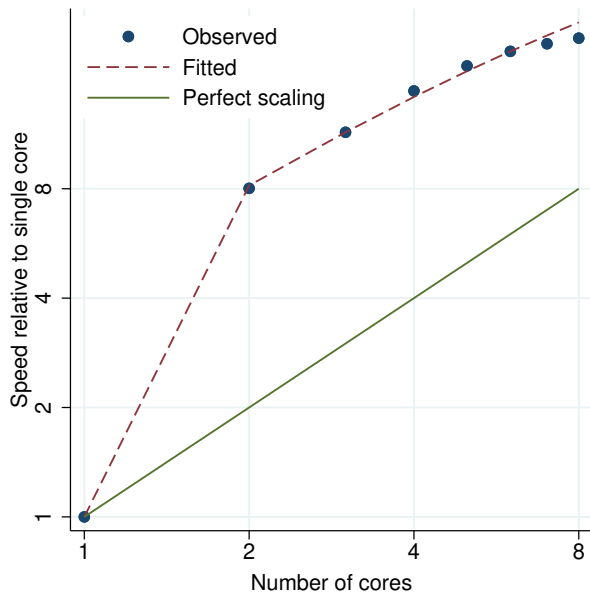


Figure 60. by: replace (small groups) performance plot.

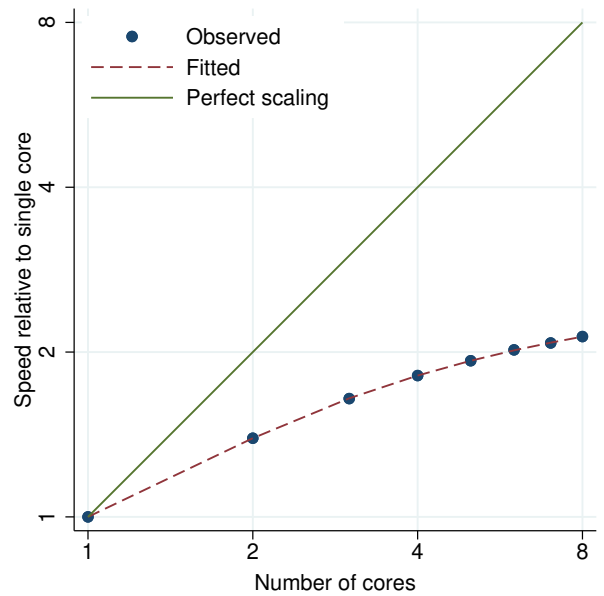


Figure 61. ca performance plot.

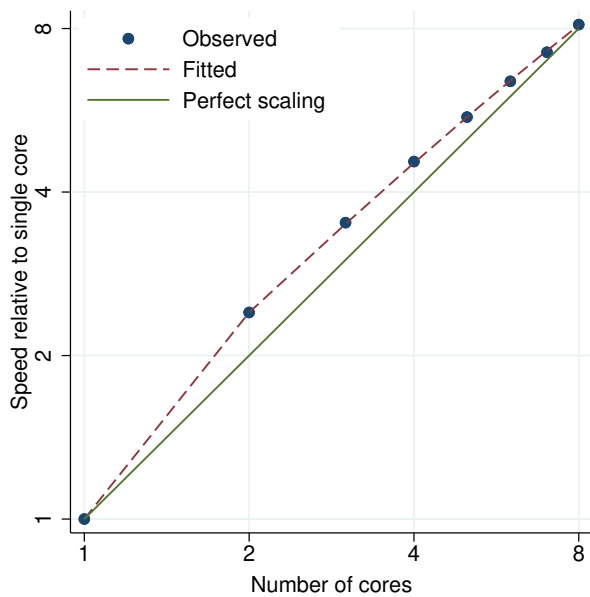


Figure 62. candisc performance plot.

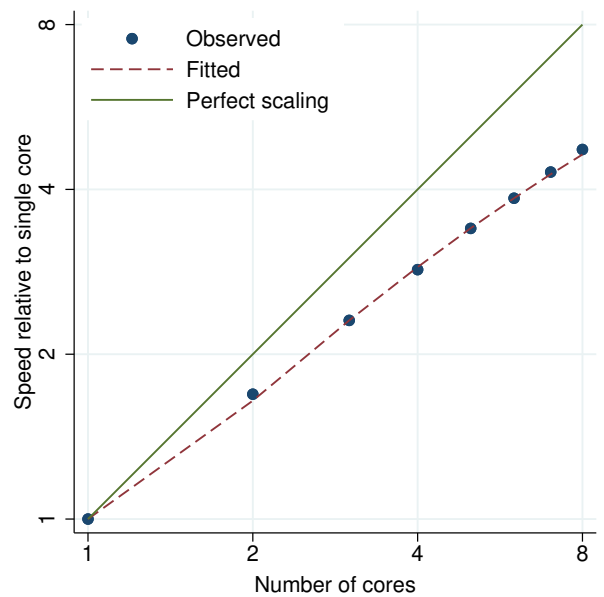


Figure 63. canon performance plot.

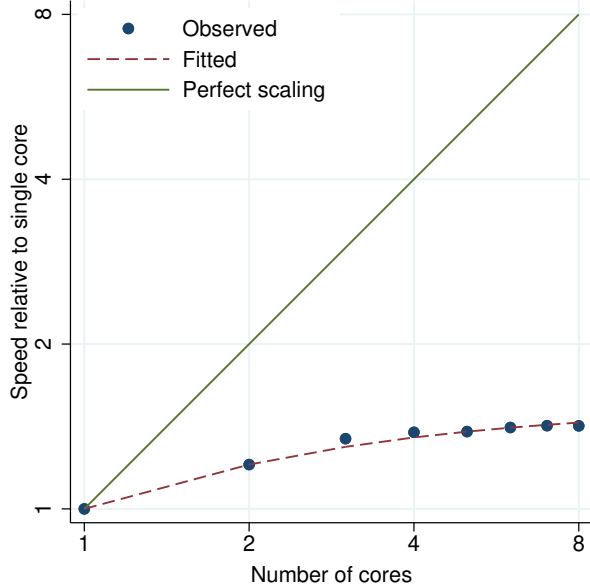


Figure 64. cc performance plot.

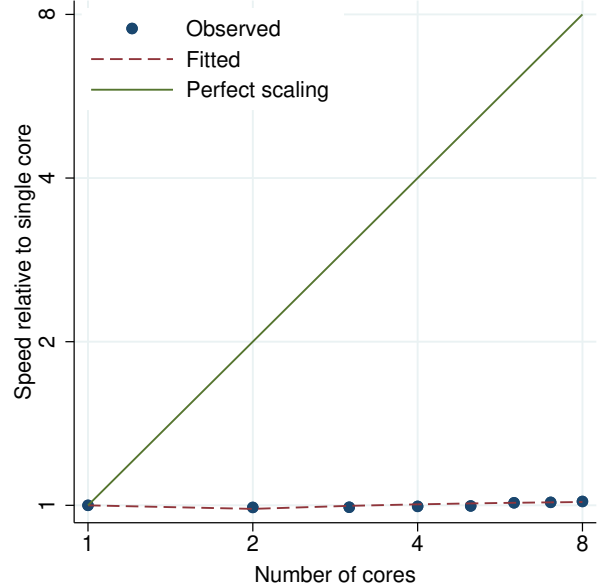


Figure 65. by: cc performance plot.

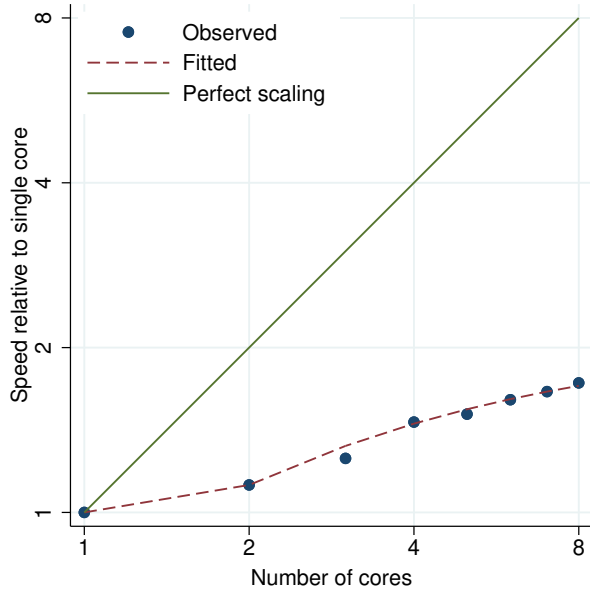


Figure 66. centile performance plot.

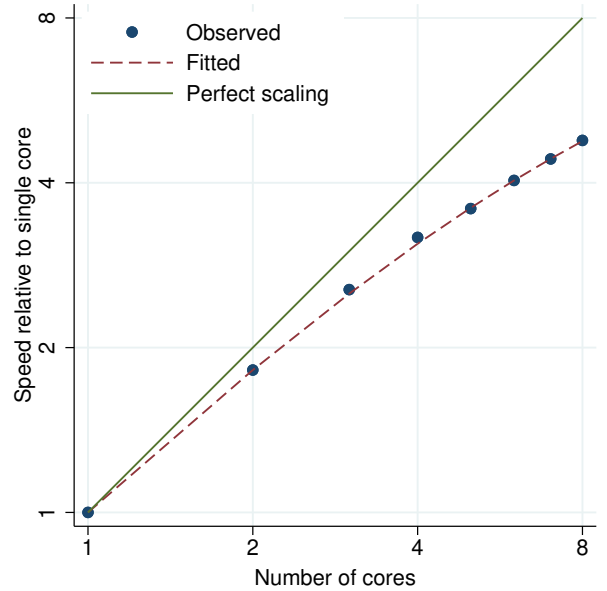


Figure 67. churdle linear performance plot.

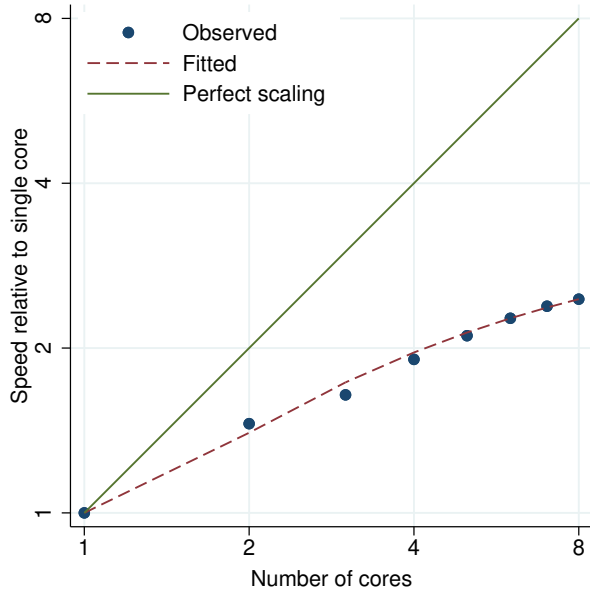


Figure 68. ci means performance plot.

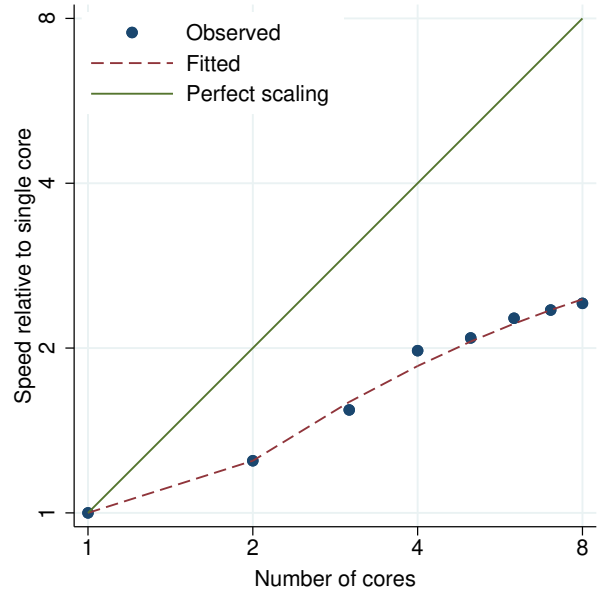


Figure 69. ci means, poisson performance plot.

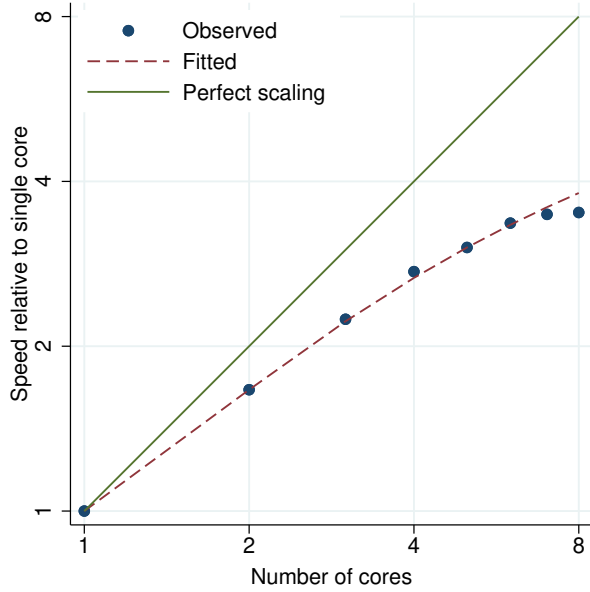


Figure 70. ci proportions performance plot.

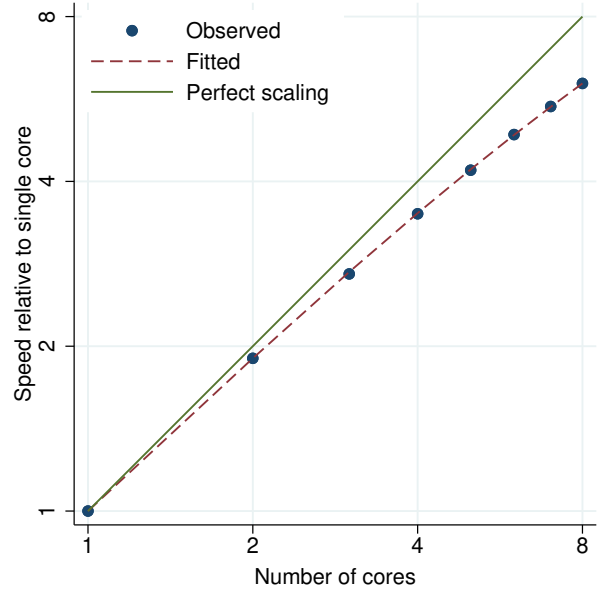


Figure 71. clogit (k1 to k2 matching) performance plot.

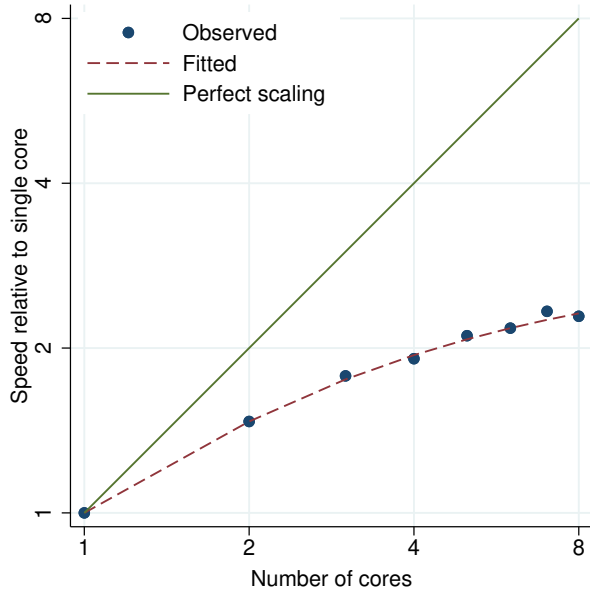


Figure 72. clogit (1 to k matching) performance plot.

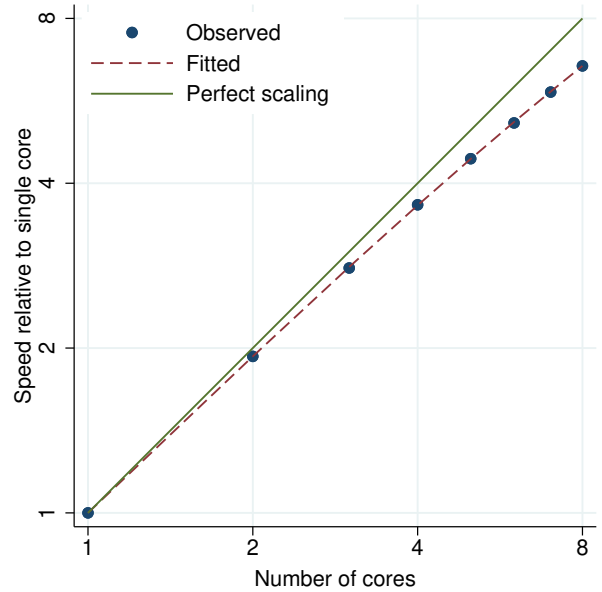


Figure 73. cloglog performance plot.

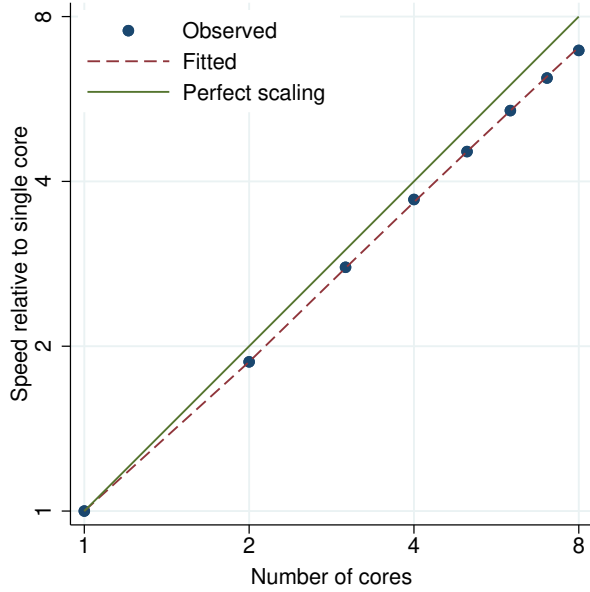


Figure 74. cluster averagelinkage performance plot.

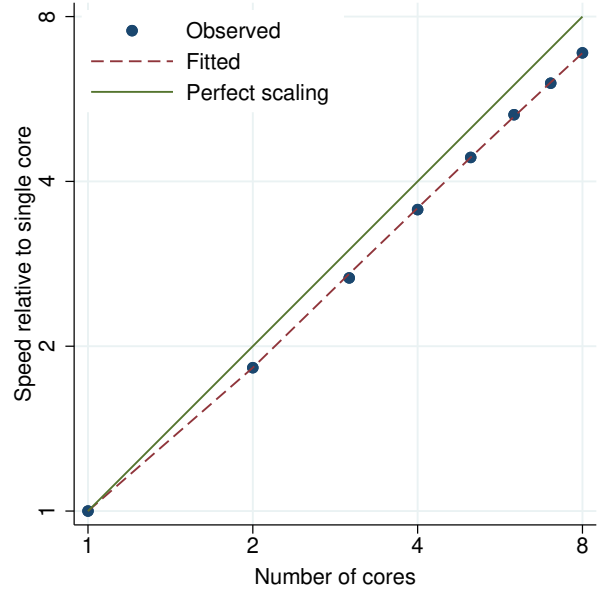


Figure 75. cluster centroidlinkage performance plot.

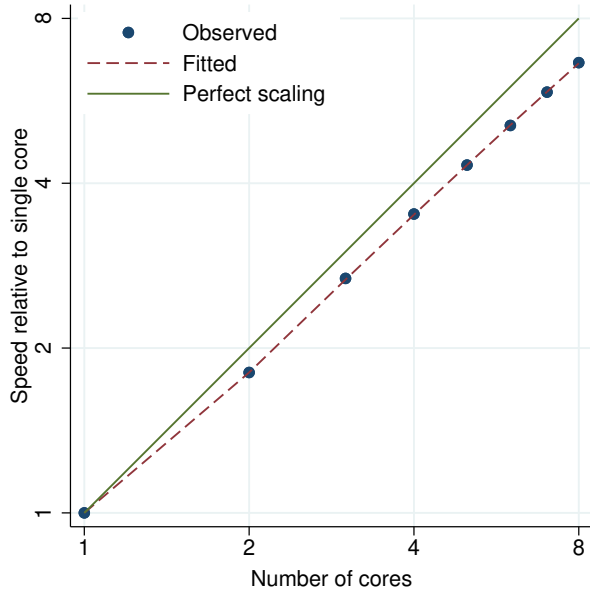


Figure 76. cluster completelinkage performance plot.

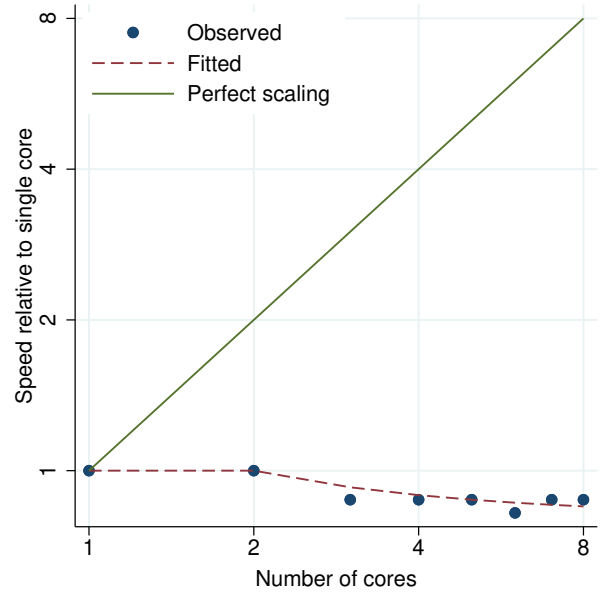


Figure 77. cluster generate performance plot.

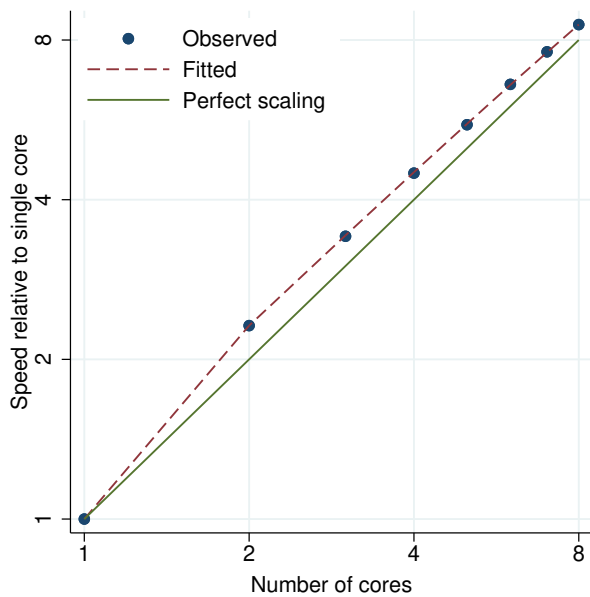


Figure 78. cluster kmeans performance plot.

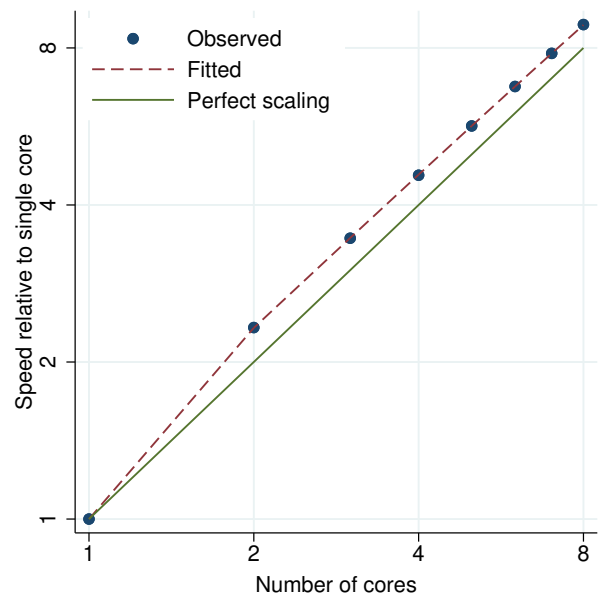


Figure 79. cluster kmedians performance plot.

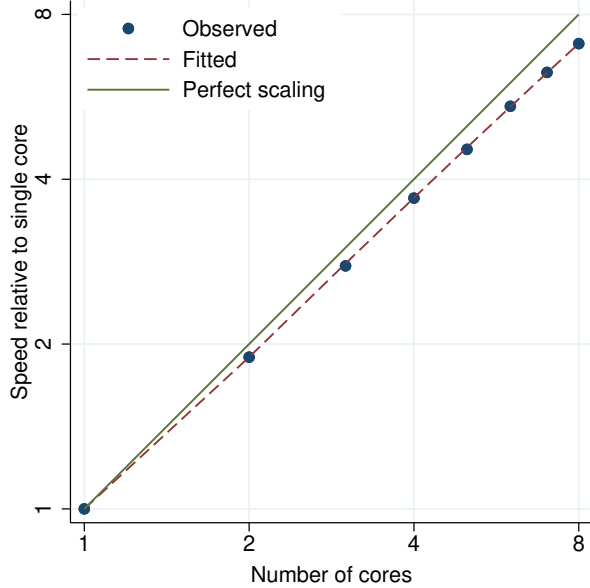


Figure 80. cluster medianlinkage performance plot.

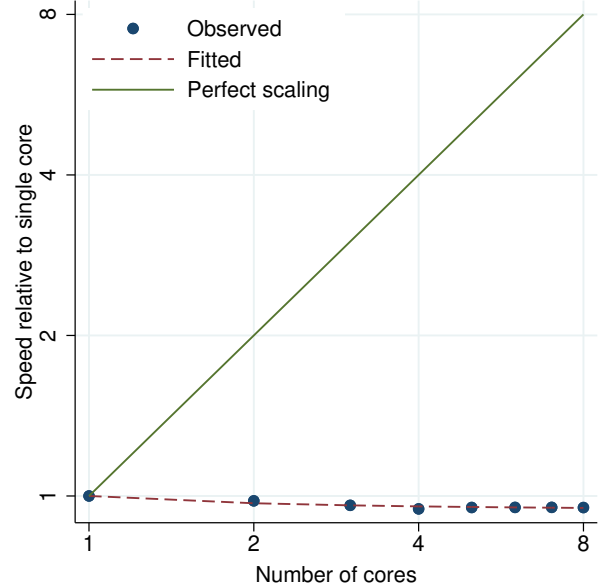


Figure 81. cluster singlelinkage performance plot.

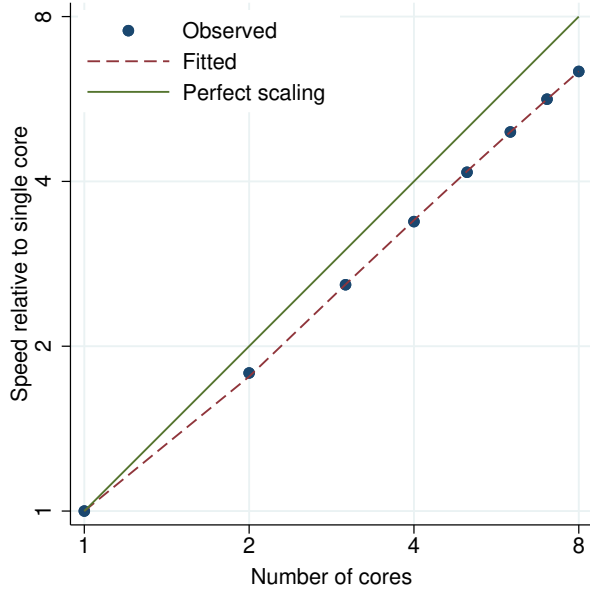


Figure 82. cluster wardslinkage performance plot.

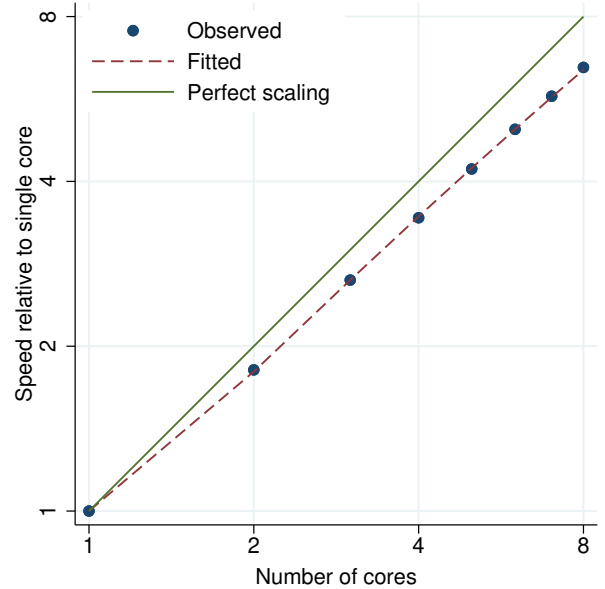


Figure 83. cluster waveragelinkage performance plot.

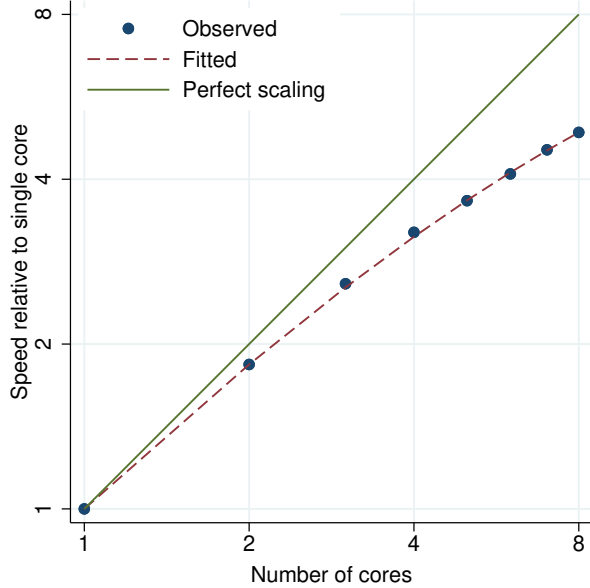


Figure 84. cmclogit performance plot.

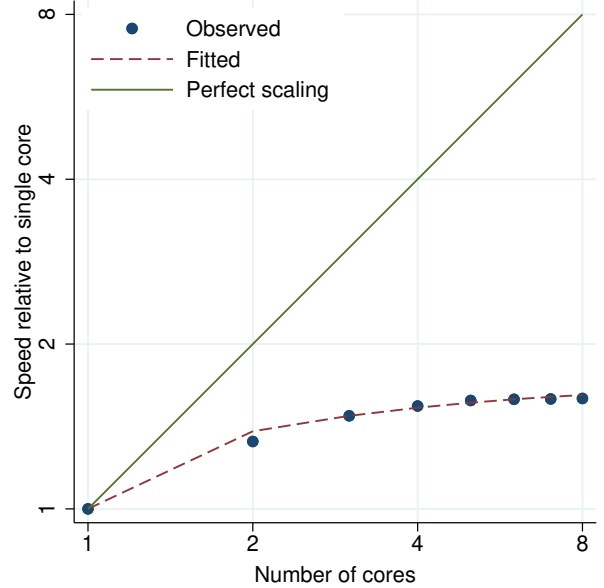


Figure 85. cmmprobit performance plot.

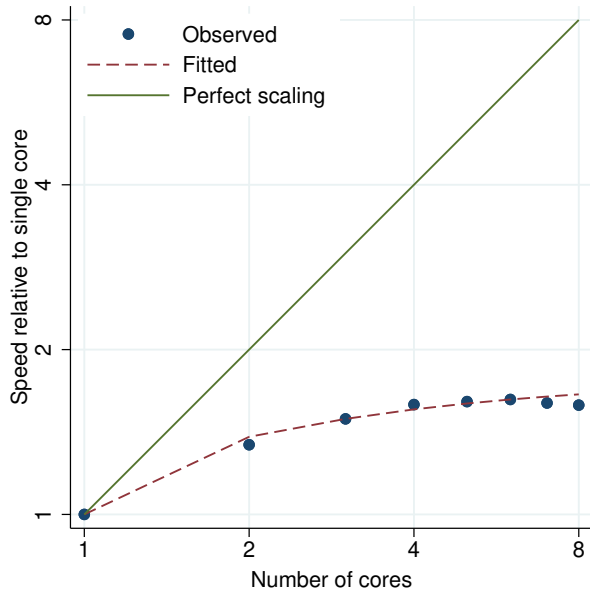


Figure 86. cmprobit performance plot.

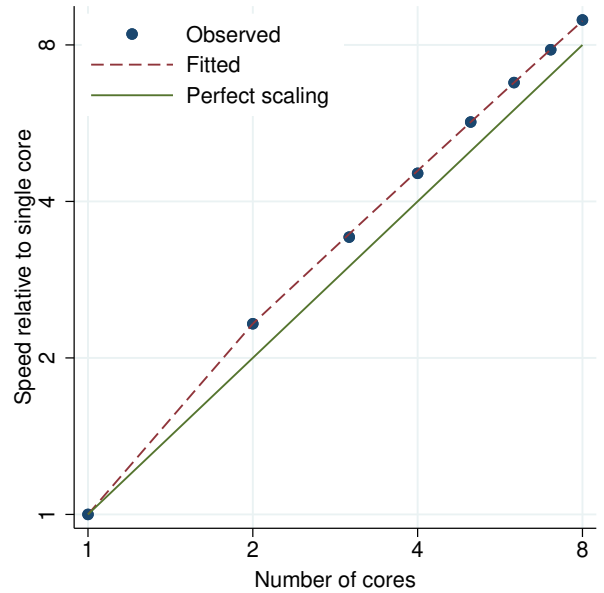


Figure 87. cnsreg performance plot.

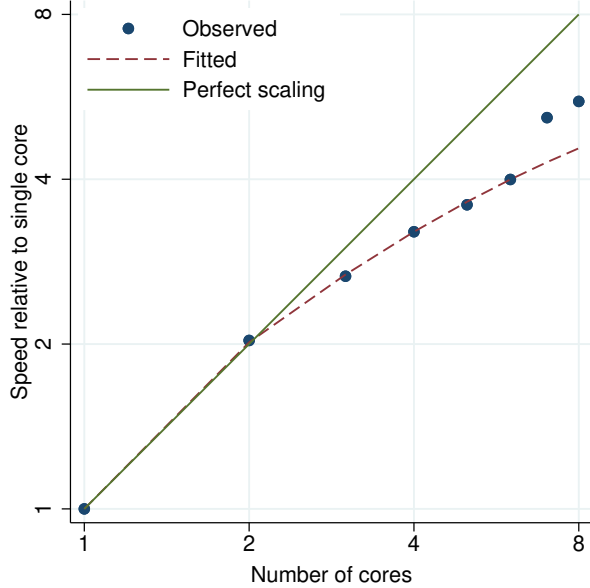


Figure 88. codebook performance plot.

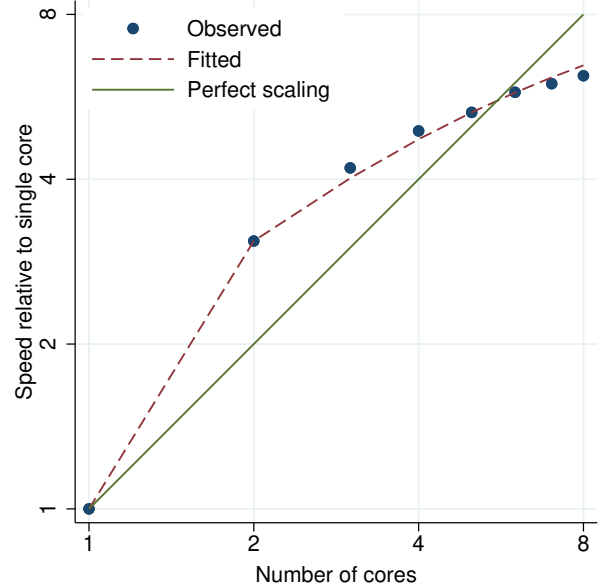


Figure 89. collapse performance plot.

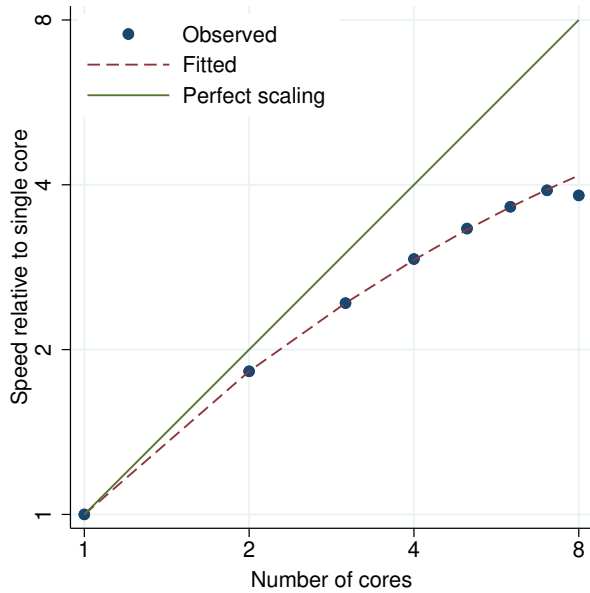


Figure 90. compare performance plot.

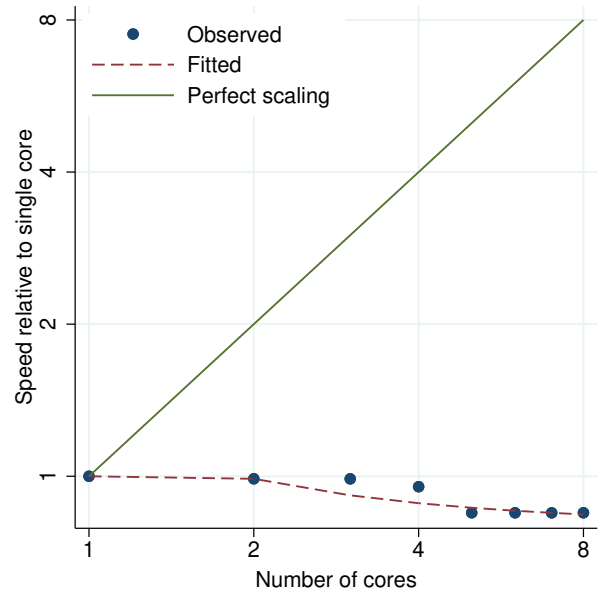


Figure 91. compress performance plot.

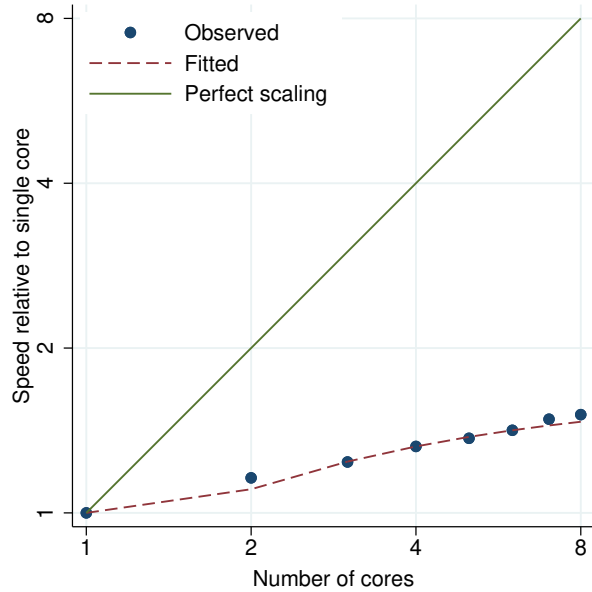


Figure 92. contract performance plot.

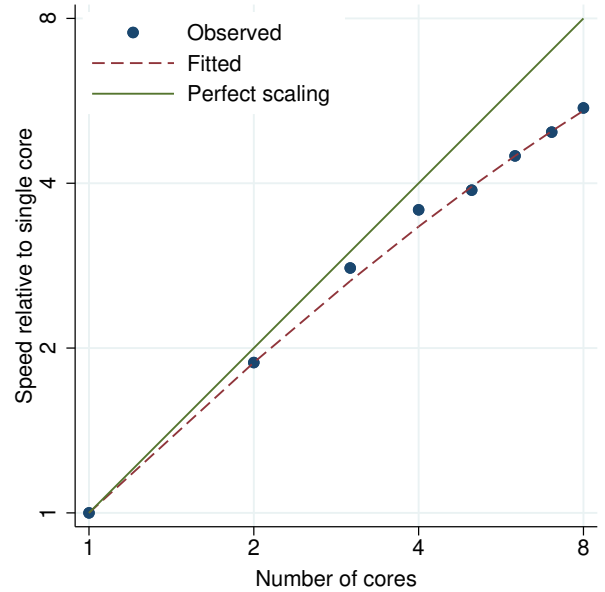


Figure 93. corr2data performance plot.

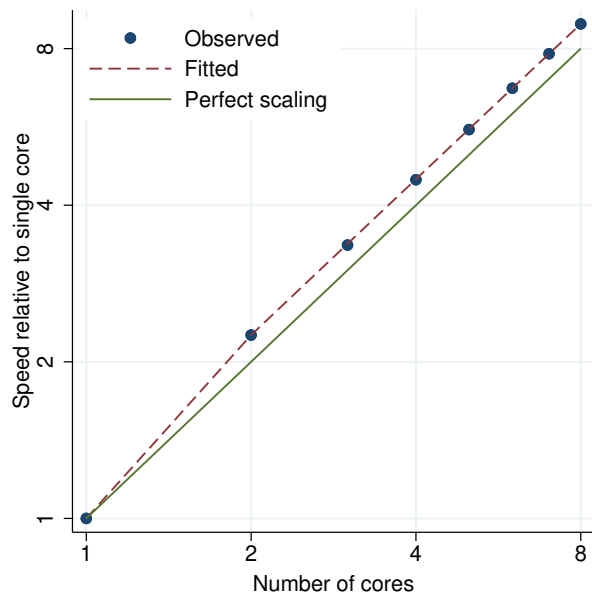


Figure 94. correlate performance plot.

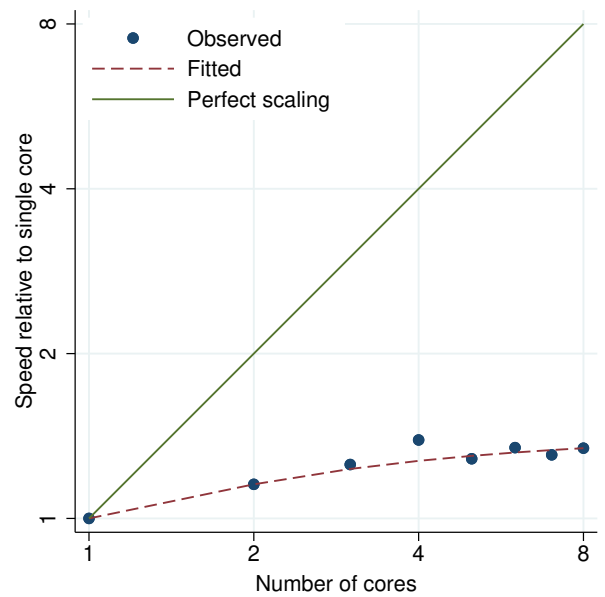


Figure 95. corrgram performance plot.

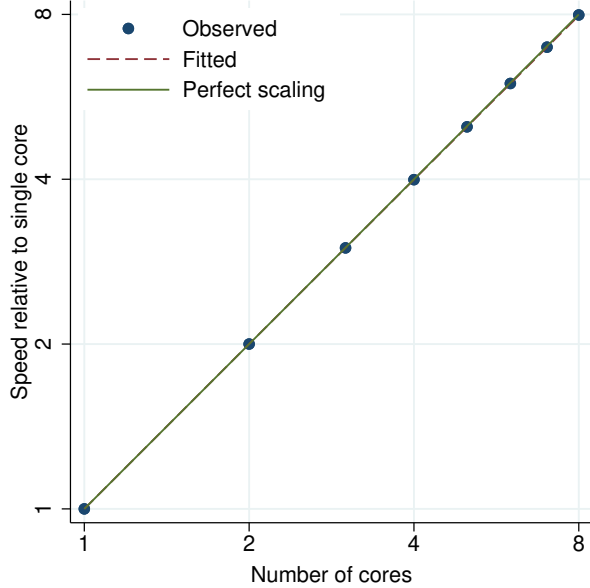


Figure 96. count performance plot.

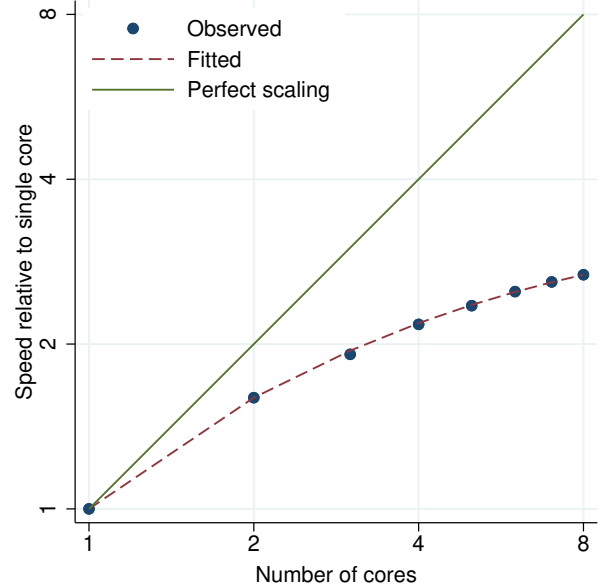


Figure 97. cpoisson performance plot.

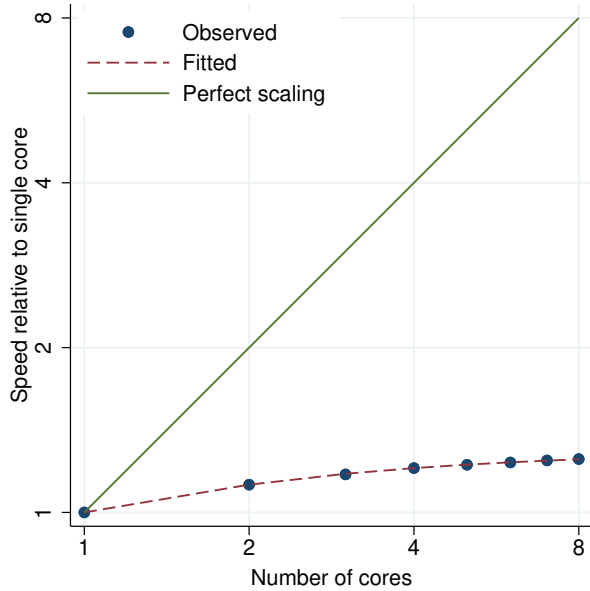


Figure 98. cs performance plot.

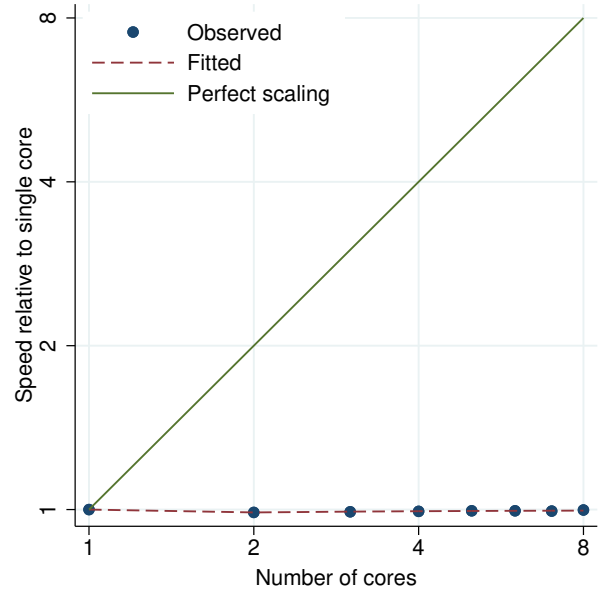


Figure 99. by: cs performance plot.

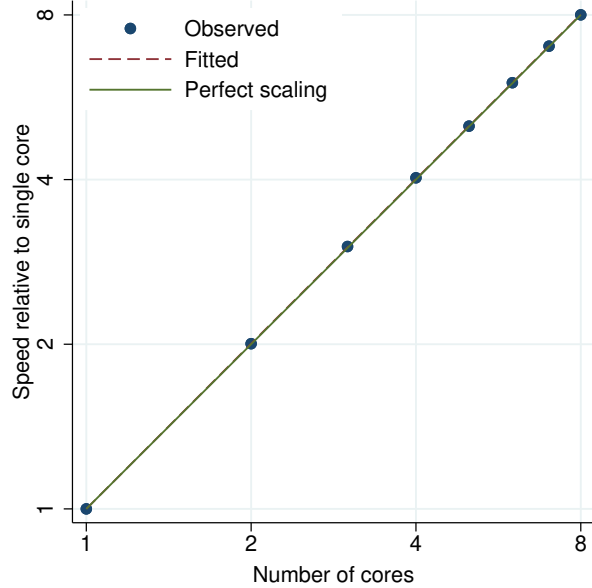


Figure 100. ctset performance plot.

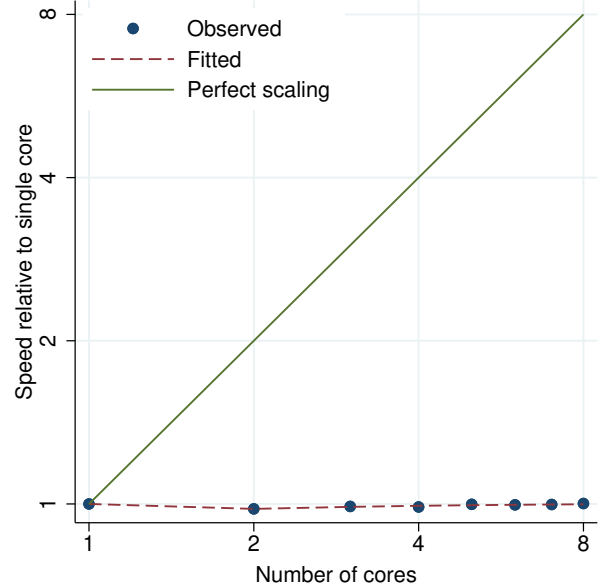


Figure 101. cttost performance plot.

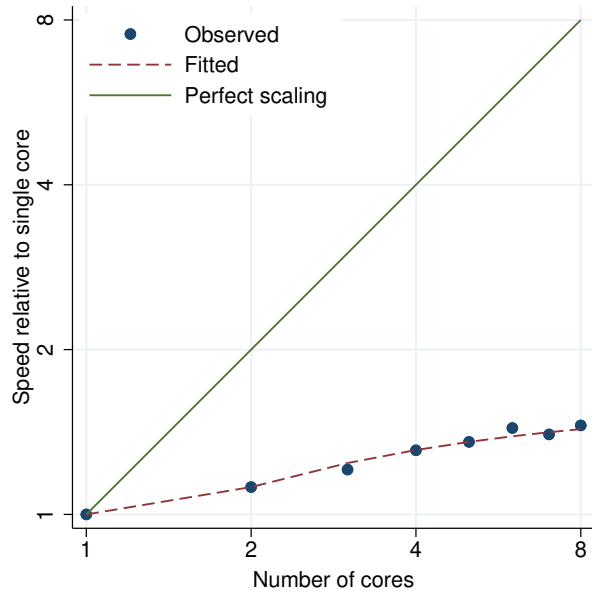


Figure 102. cumu1 performance plot.

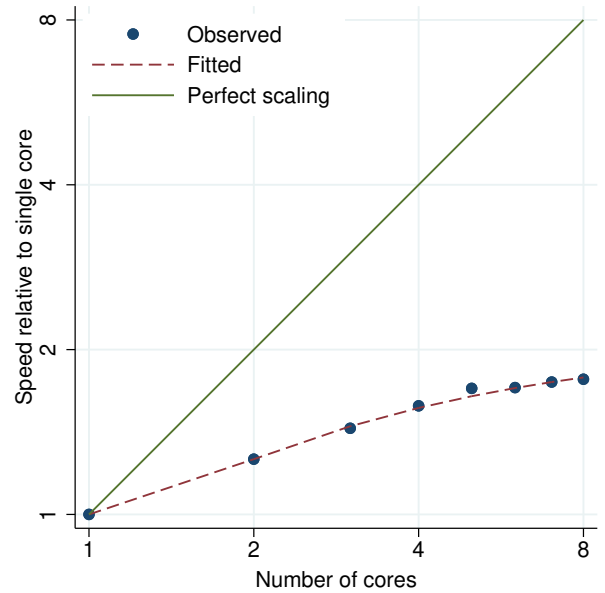


Figure 103. cusum performance plot.

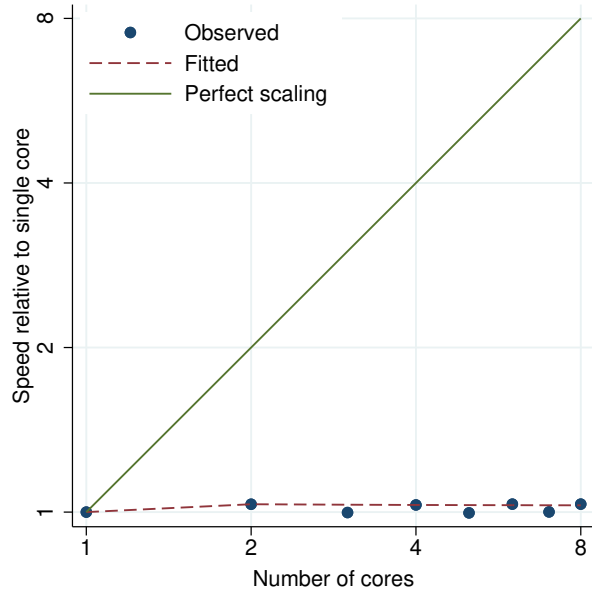


Figure 104. datasignature performance plot.

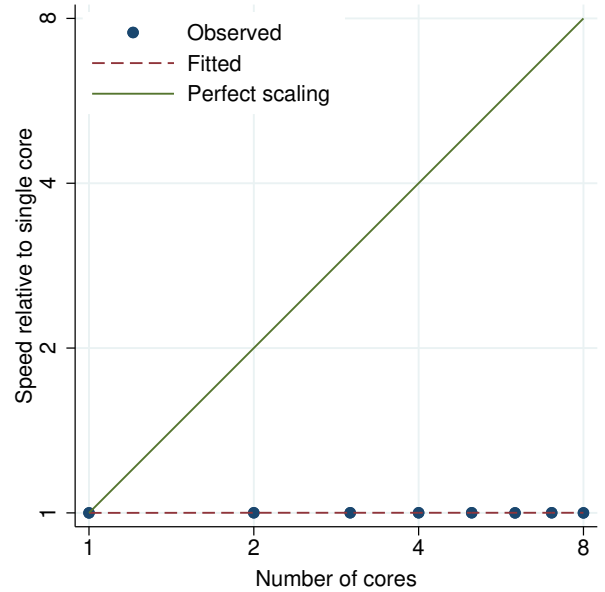


Figure 105. decode performance plot.

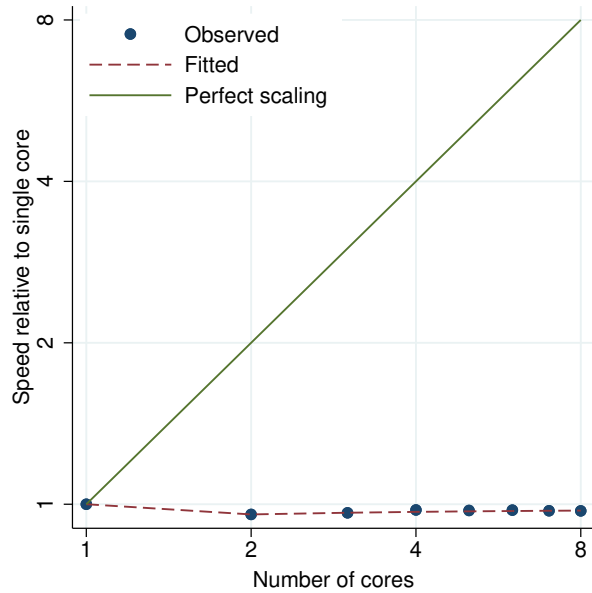


Figure 106. demandsys aids performance plot.

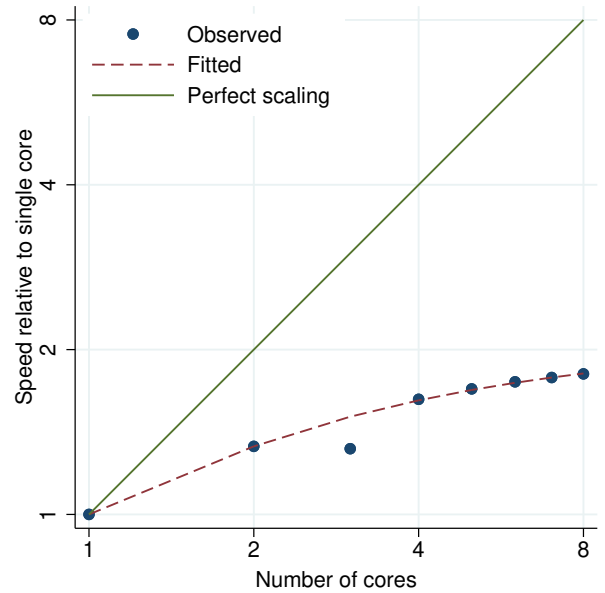


Figure 107. demandsys cdouglas performance plot.

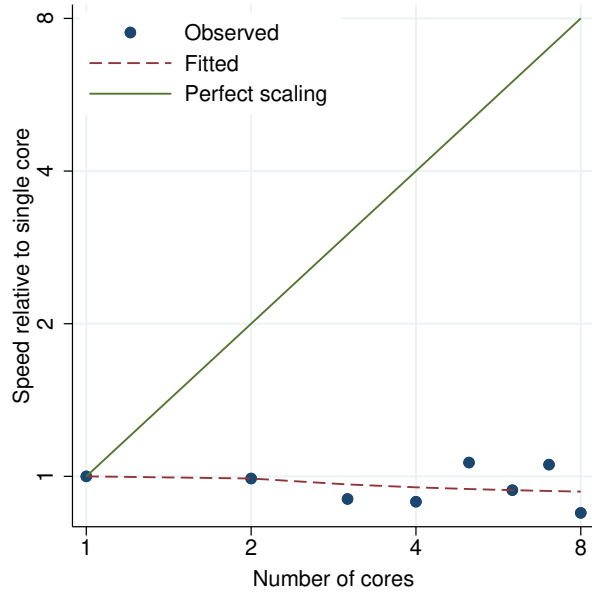


Figure 108. demandsys gaids performance plot.

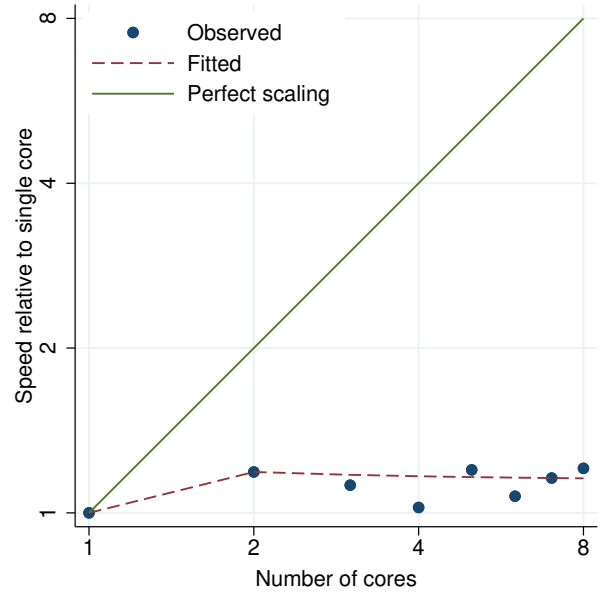


Figure 109. demandsys gquads performance plot.

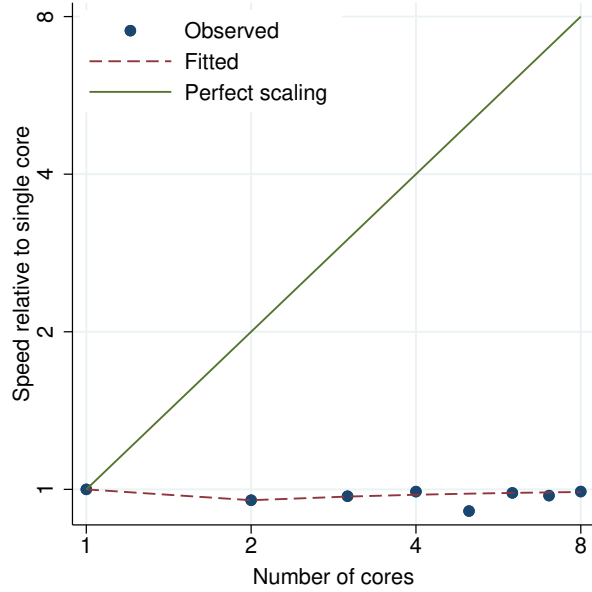


Figure 110. demandsys gtranslog performance plot.

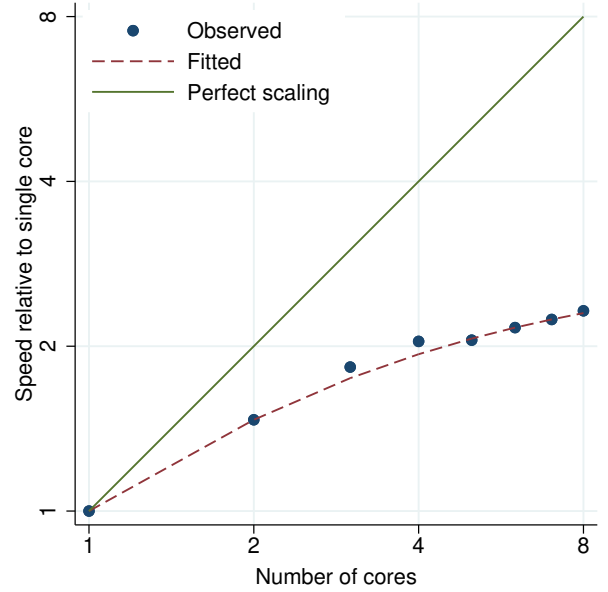


Figure 111. demandsys les performance plot.

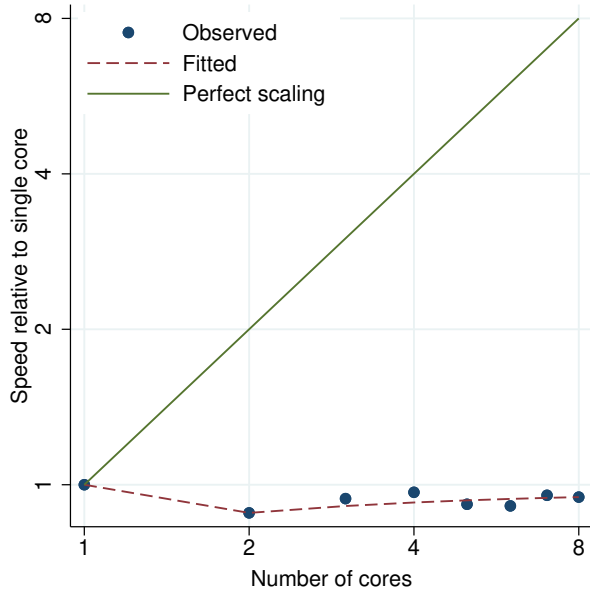


Figure 112. demandsys quajds performance plot.

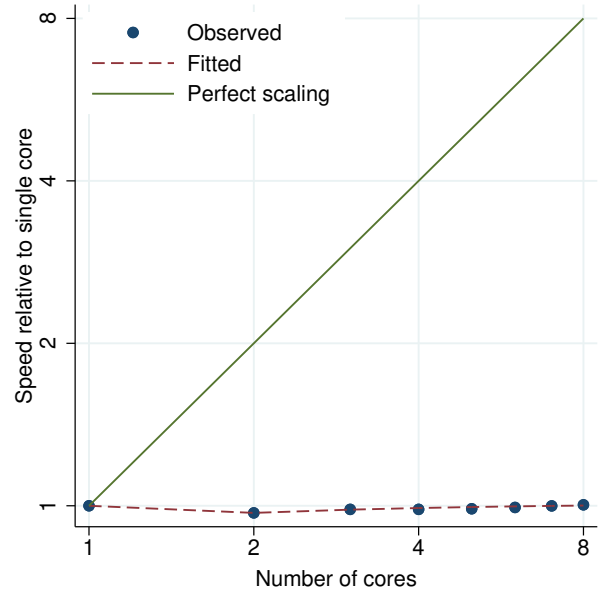


Figure 113. demandsys translog performance plot.

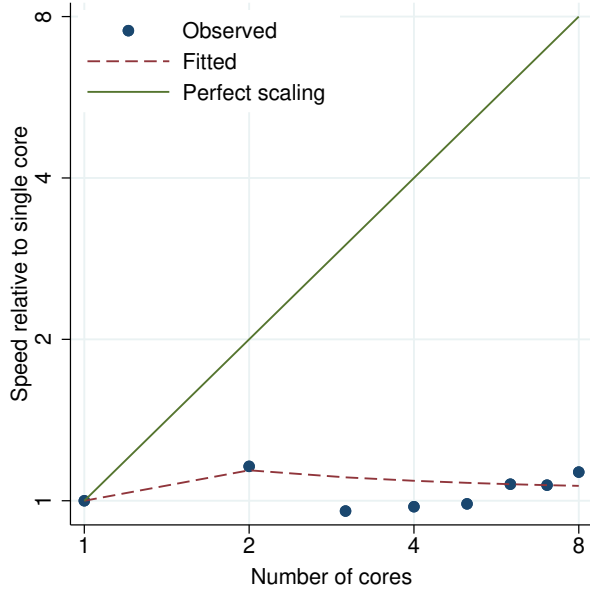


Figure 114. destring performance plot.

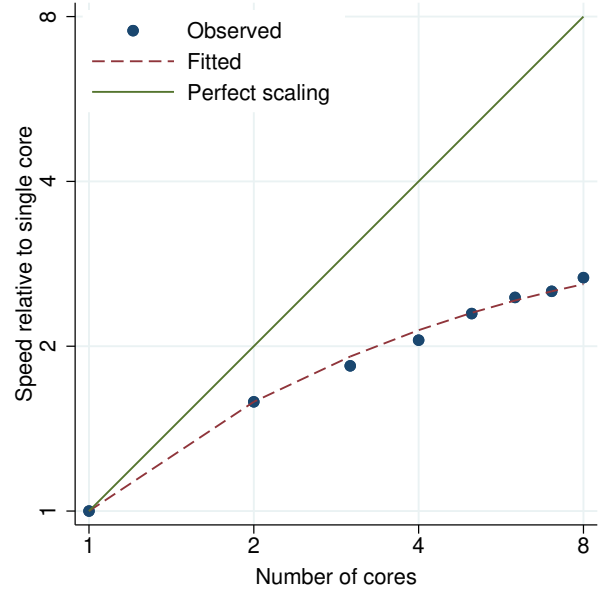


Figure 115. dfactor performance plot.

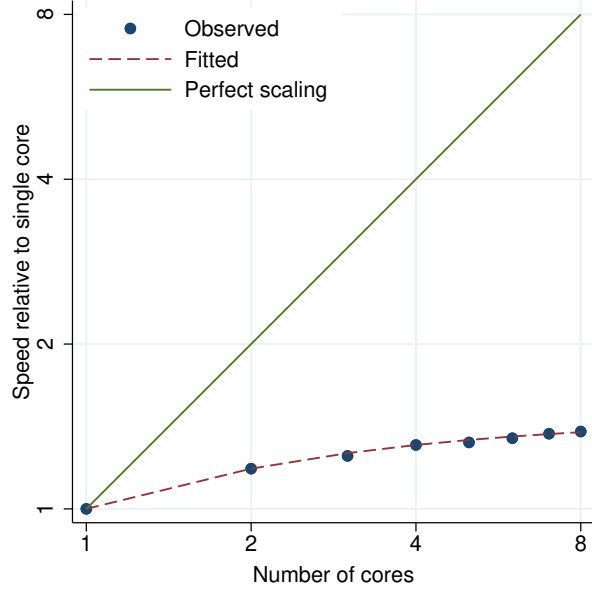


Figure 116. dfgls performance plot.

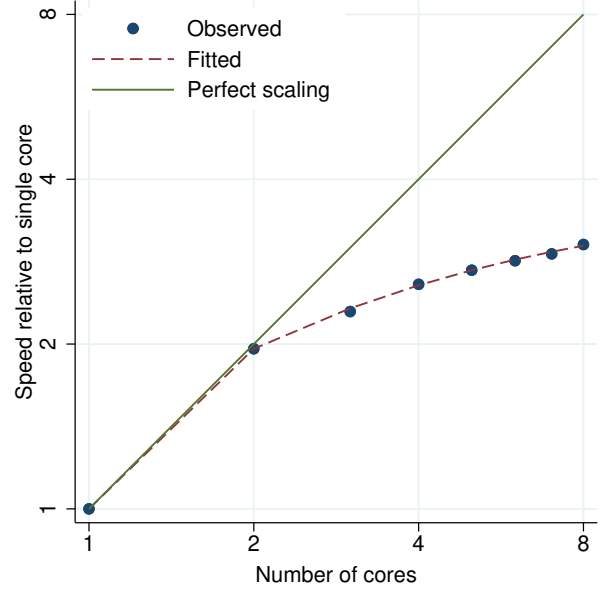


Figure 117. dfuller performance plot.

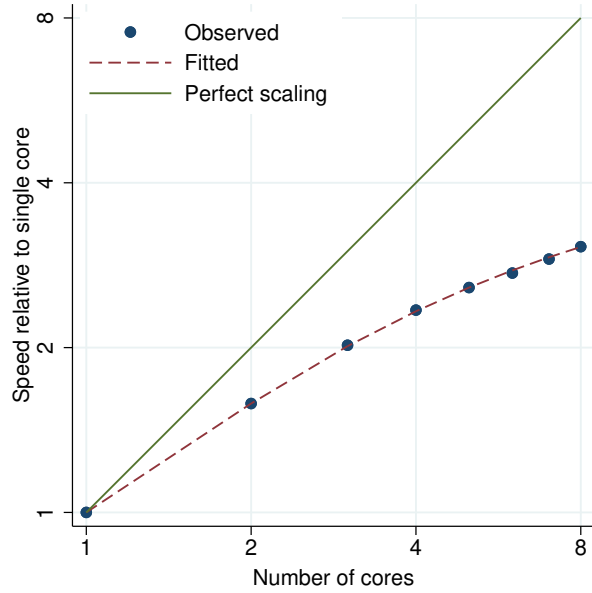


Figure 118. didregress performance plot.

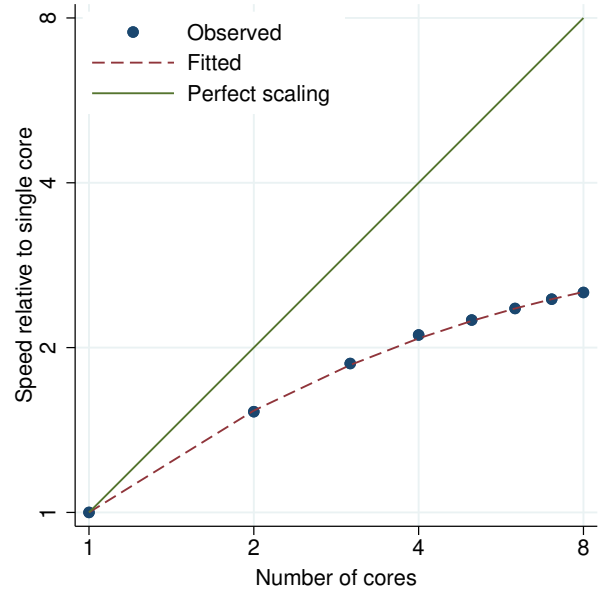


Figure 119. discrim knn performance plot.

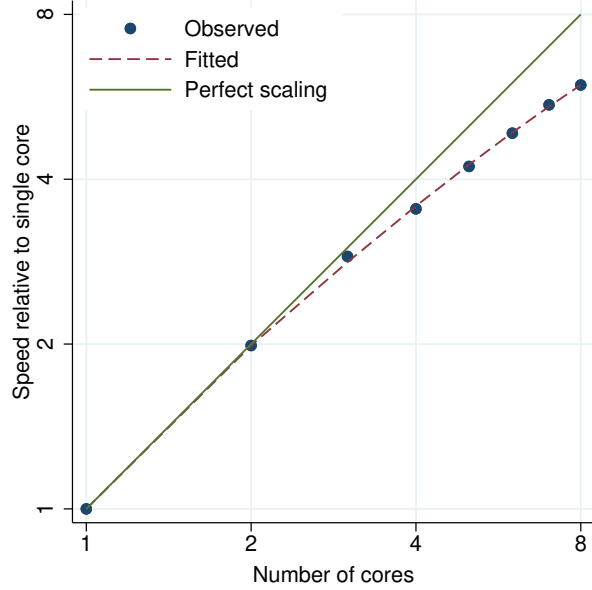


Figure 120. discrim lda performance plot.

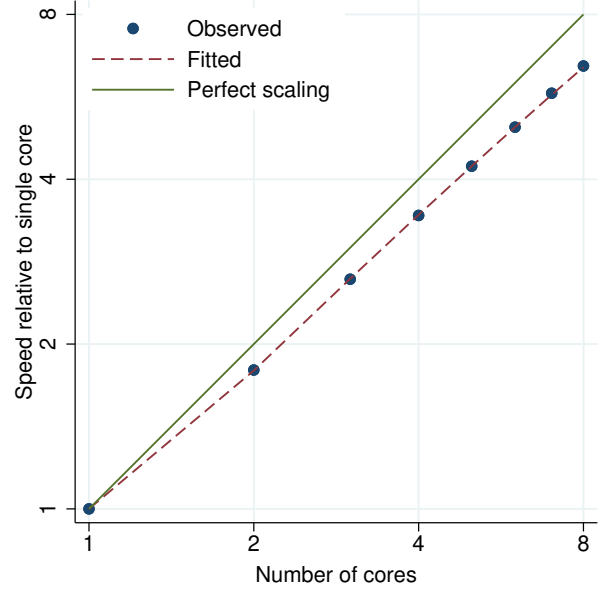


Figure 121. discrim logistic performance plot.

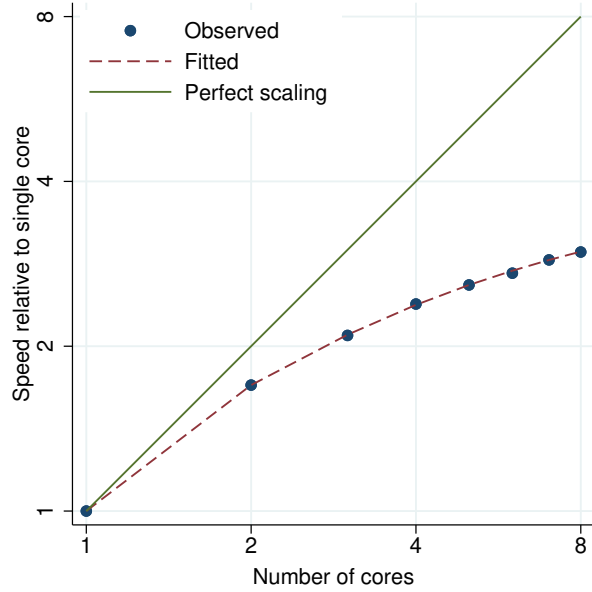


Figure 122. discrim qda performance plot.

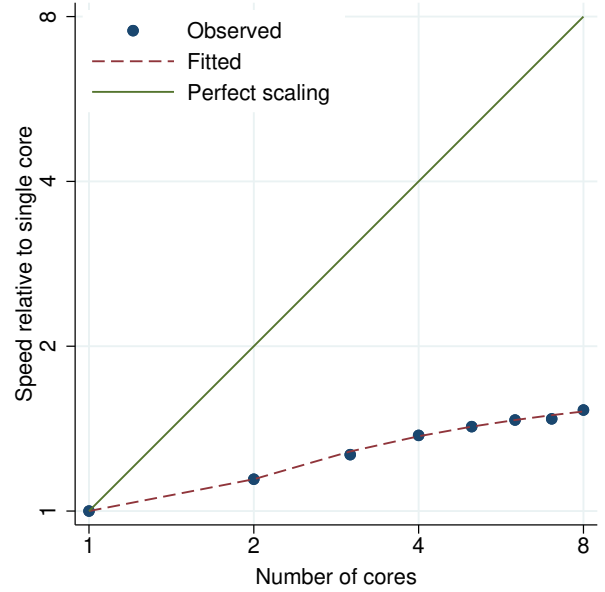


Figure 123. dotplot performance plot.

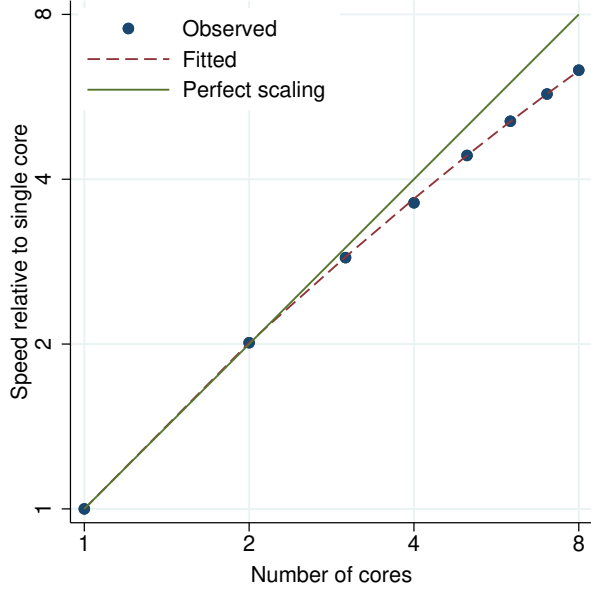


Figure 124. drawnorm performance plot.

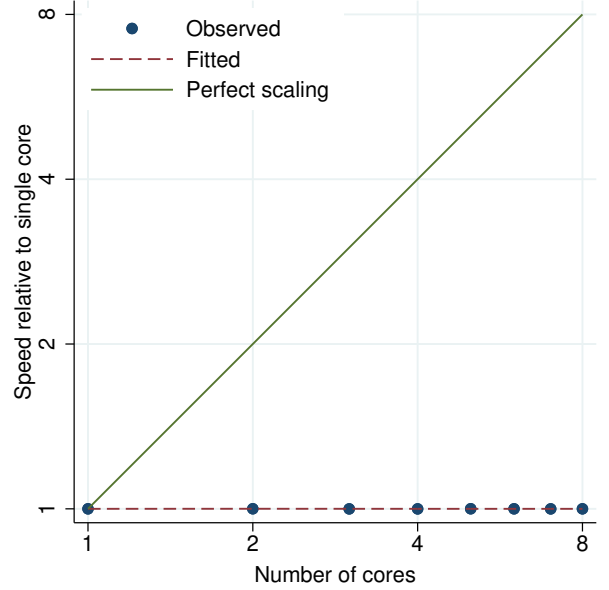


Figure 125. drop if exp performance plot.

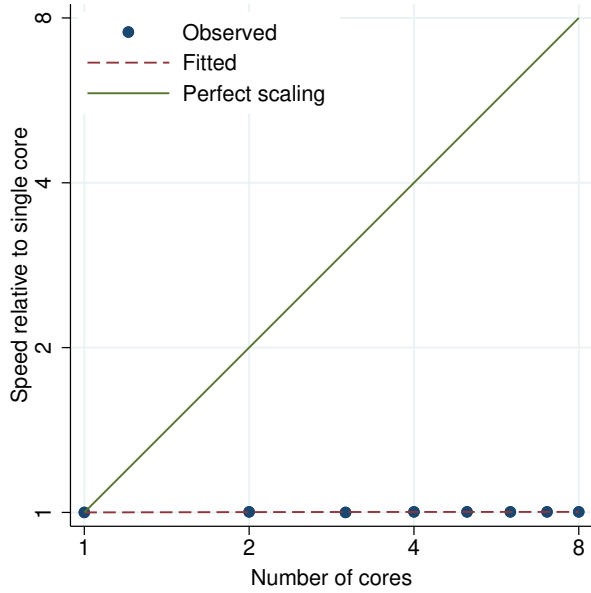


Figure 126. drop in range performance plot.

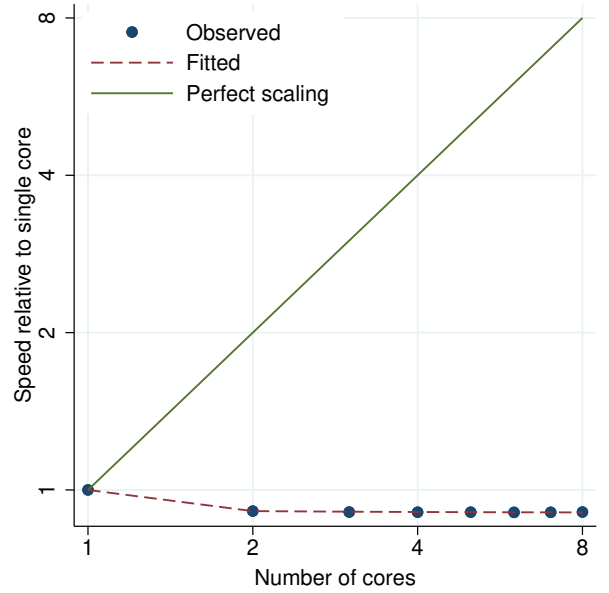


Figure 127. dsge performance plot.

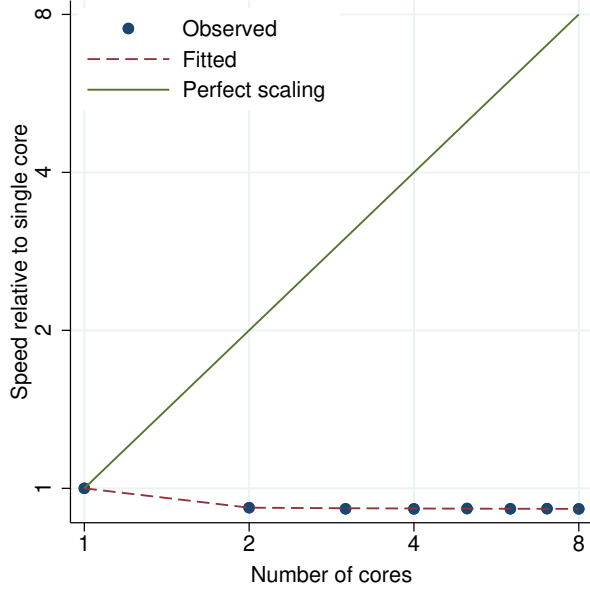


Figure 128. dsngen1 performance plot.

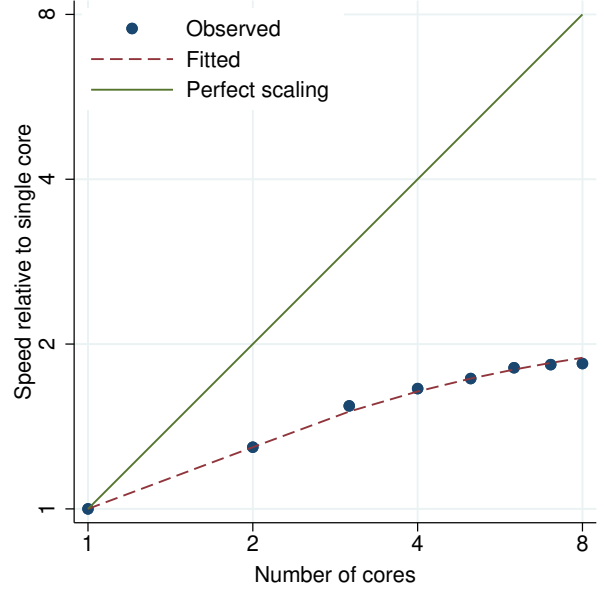


Figure 129. dslogit performance plot.

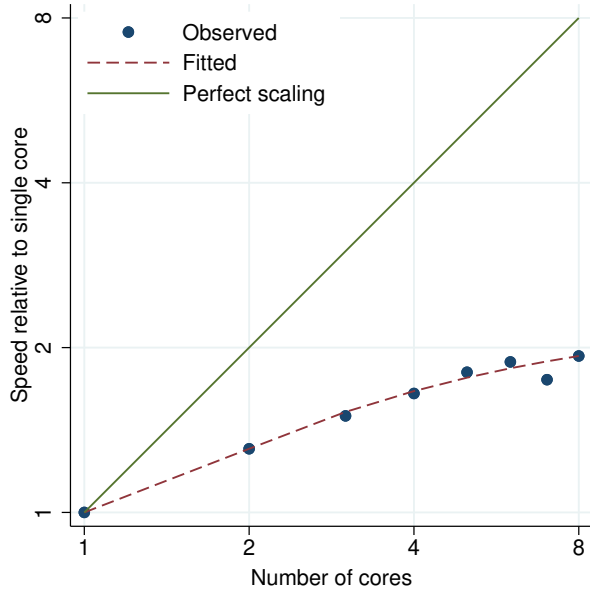


Figure 130. dspoisson performance plot.

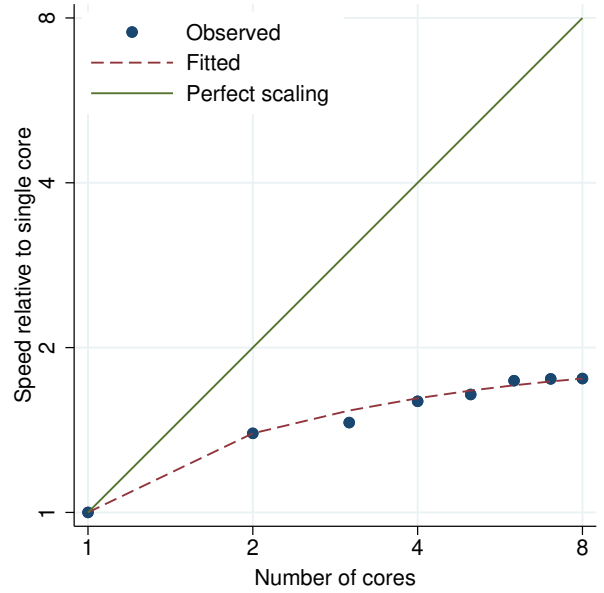


Figure 131. dsregress performance plot.

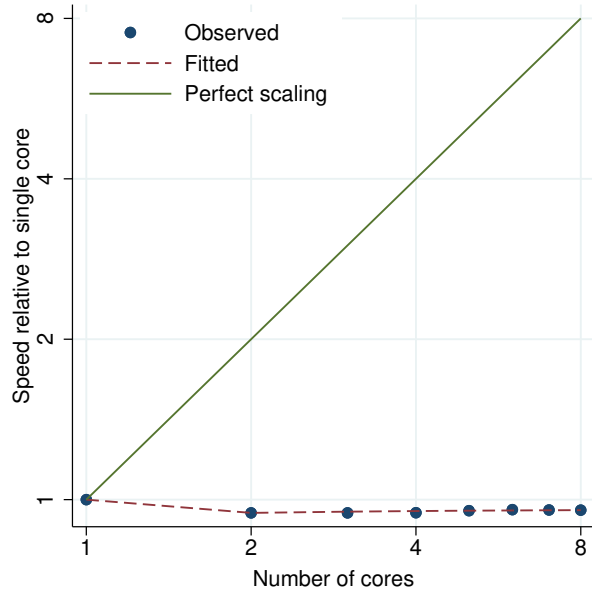


Figure 132. dstdize performance plot.

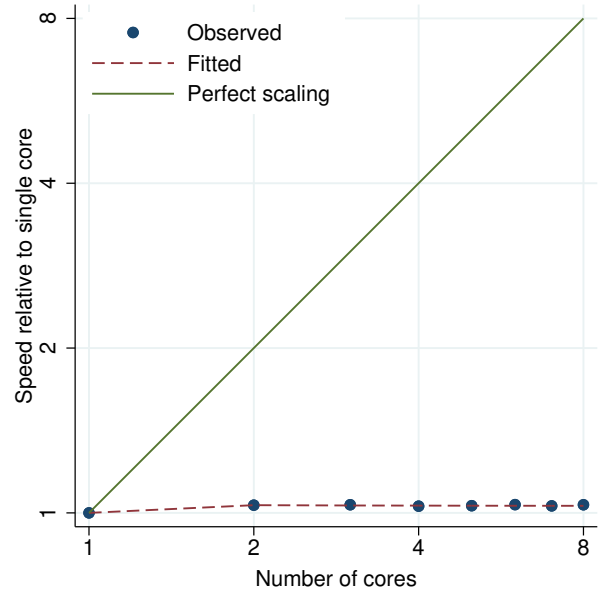


Figure 133. dvech performance plot.

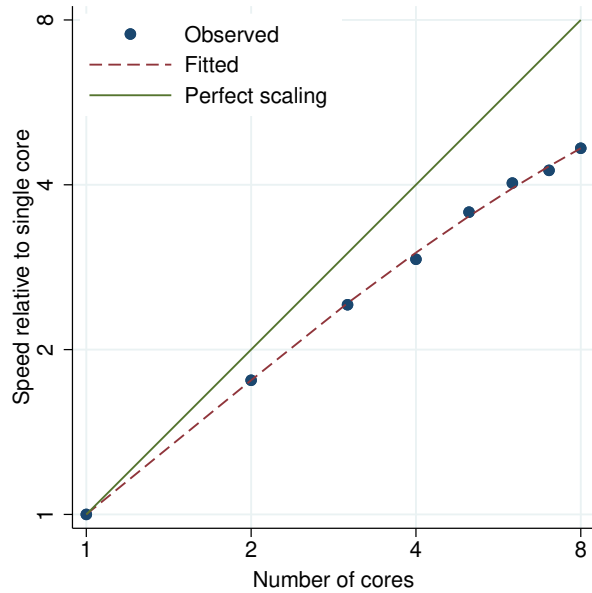


Figure 134. egen group() performance plot.

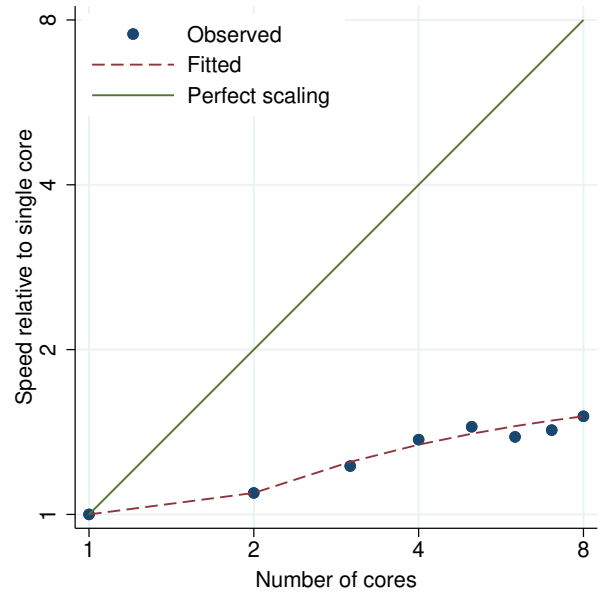


Figure 135. by: egen mean performance plot.

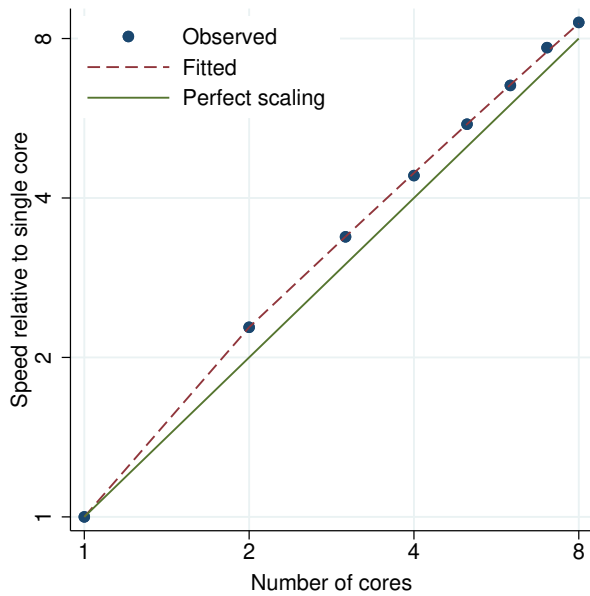


Figure 136. eivreg performance plot.

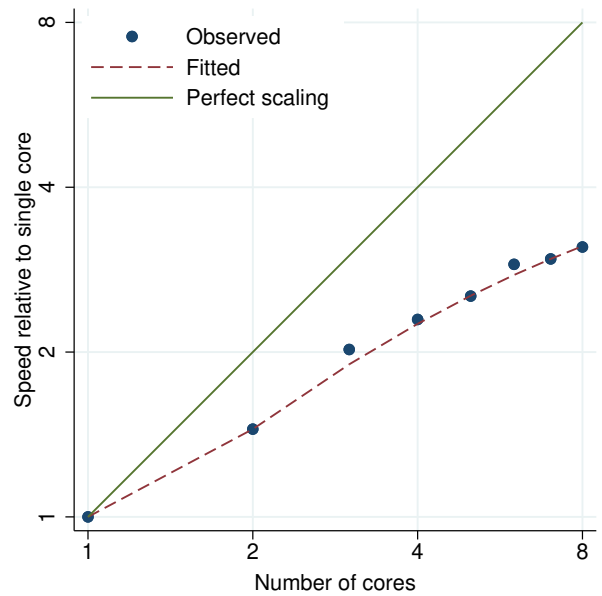


Figure 137. encode performance plot.

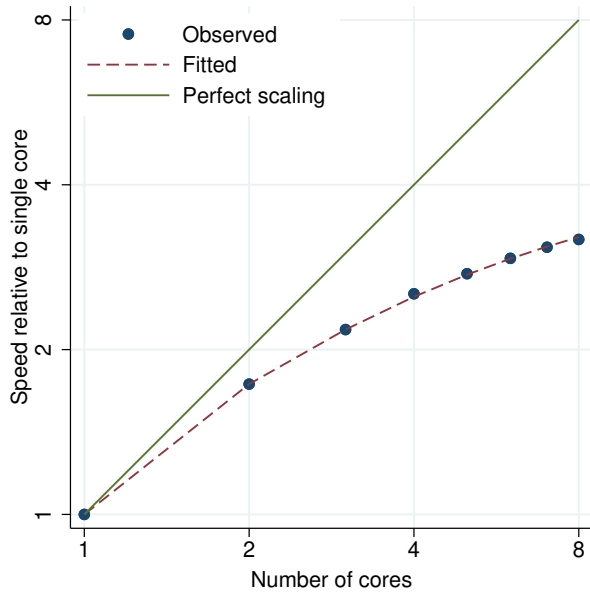


Figure 138. eregress performance plot.

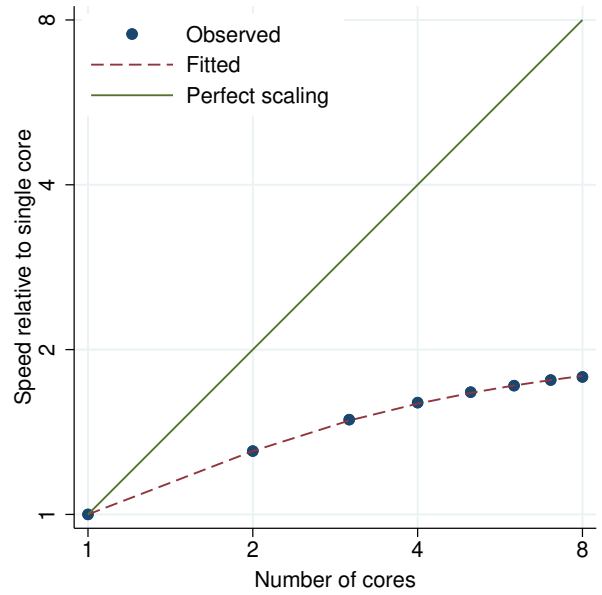


Figure 139. esize twosample performance plot.

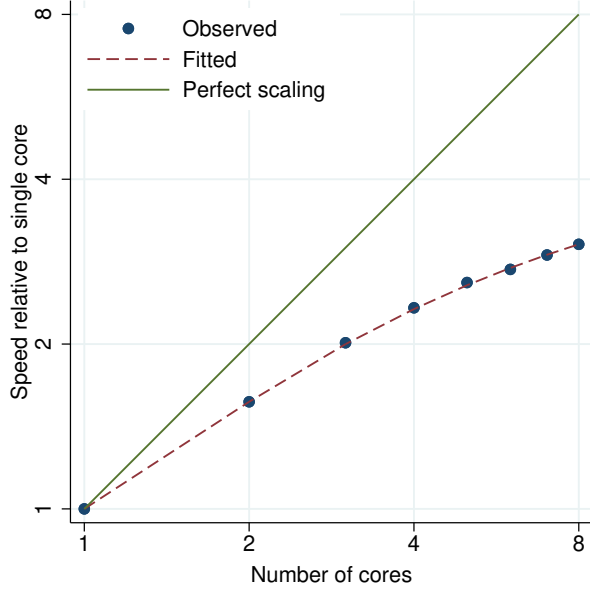


Figure 140. esize unpaired performance plot.

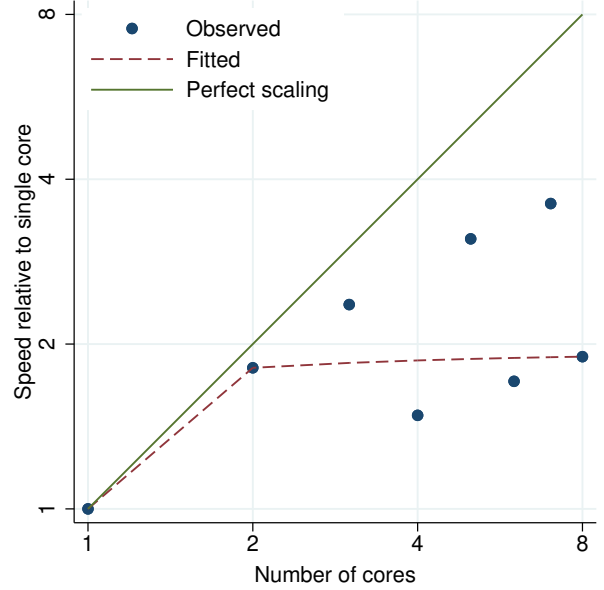


Figure 141. eteffects (exponential), ate performance plot.

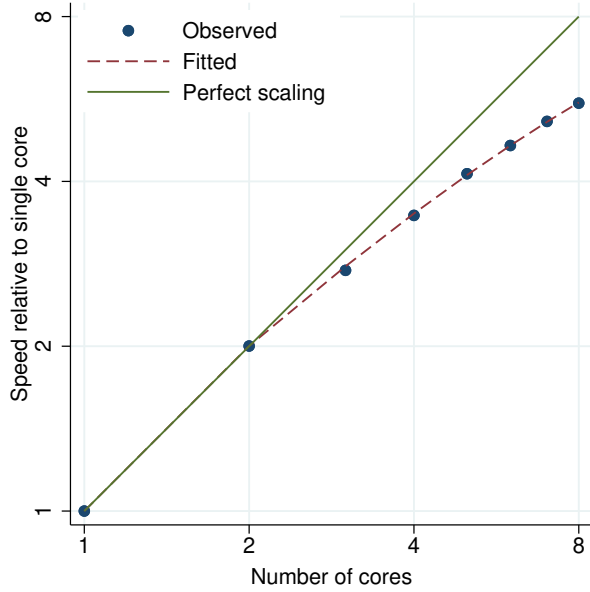


Figure 142. eteffects (linear), ate performance plot.

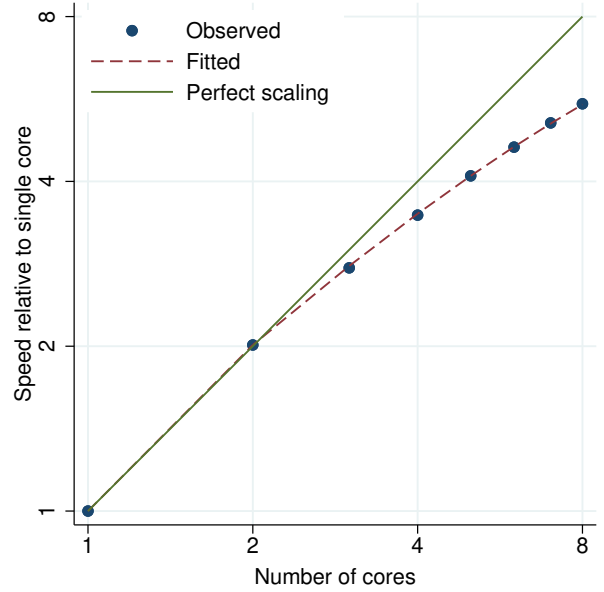


Figure 143. eteffects (linear), pomeans performance plot.

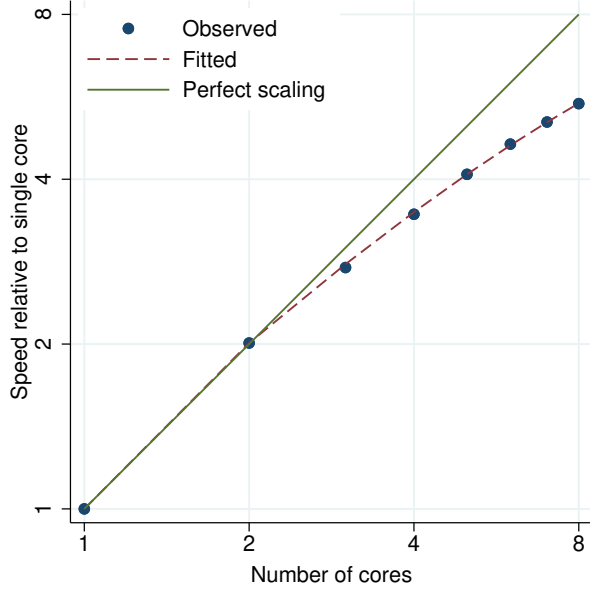


Figure 144. eteffects (probit), ate performance plot.

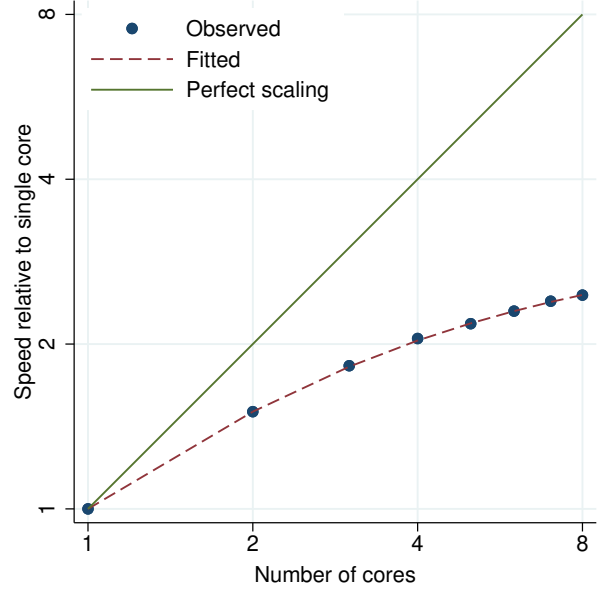


Figure 145. etpoisson performance plot.

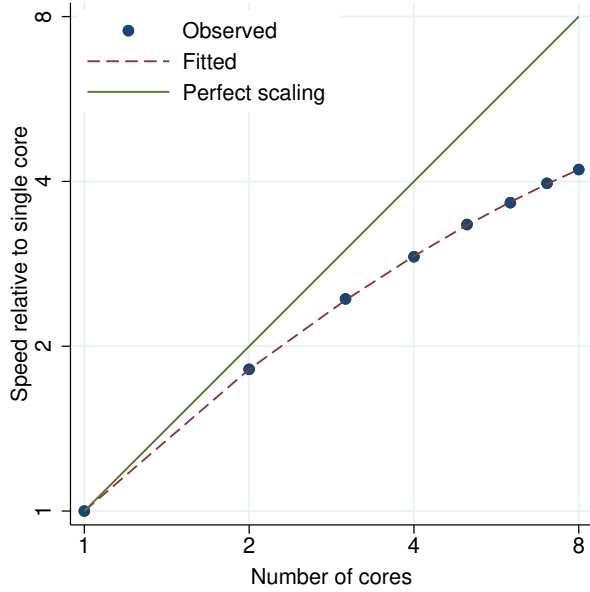


Figure 146. etregress, poutcomes performance plot.

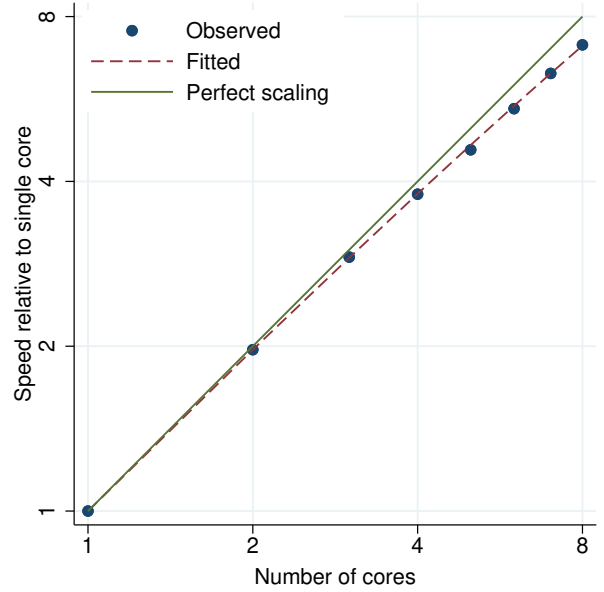


Figure 147. etregress, twostep performance plot.

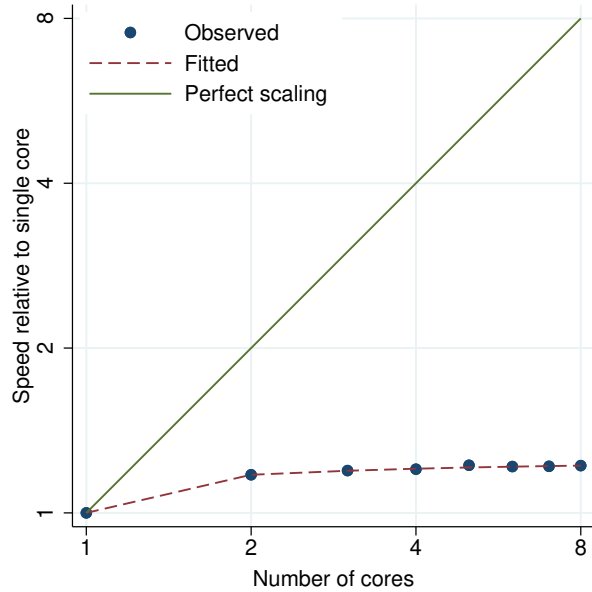


Figure 148. exlogistic performance plot.

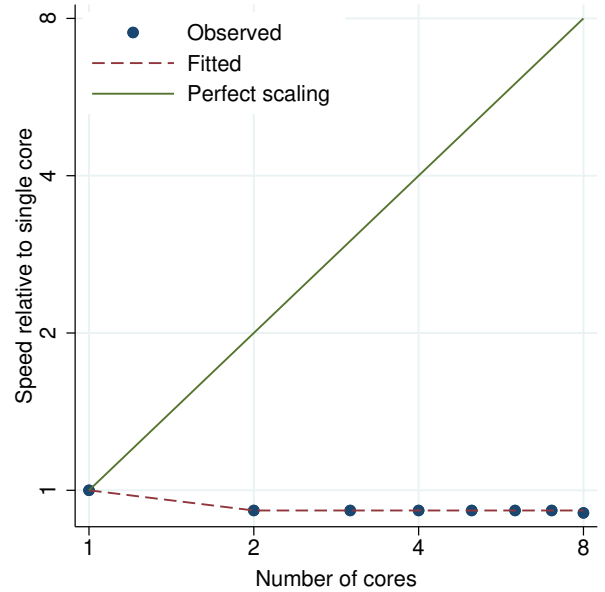


Figure 149. expand # performance plot.

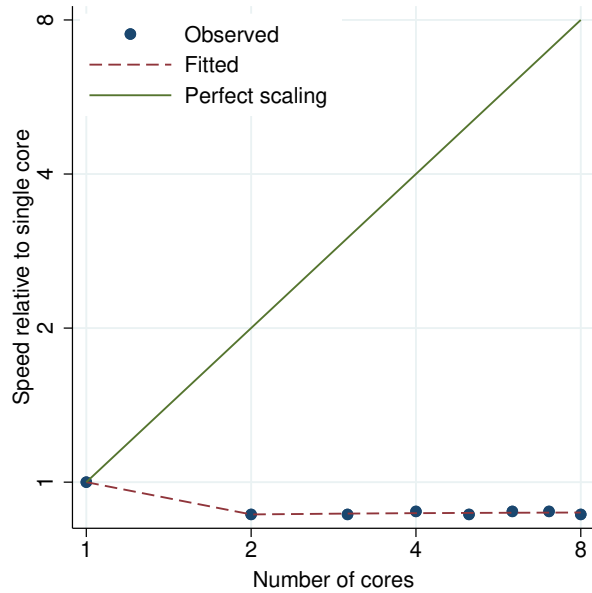


Figure 150. expand varname performance plot.

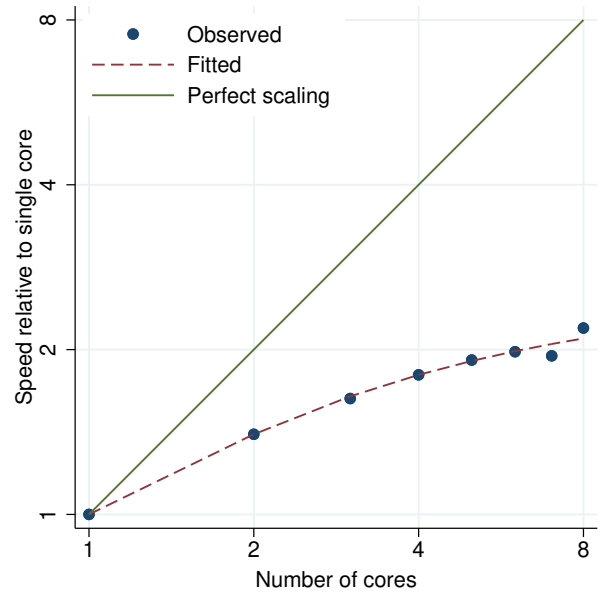


Figure 151. expandc1 # performance plot.

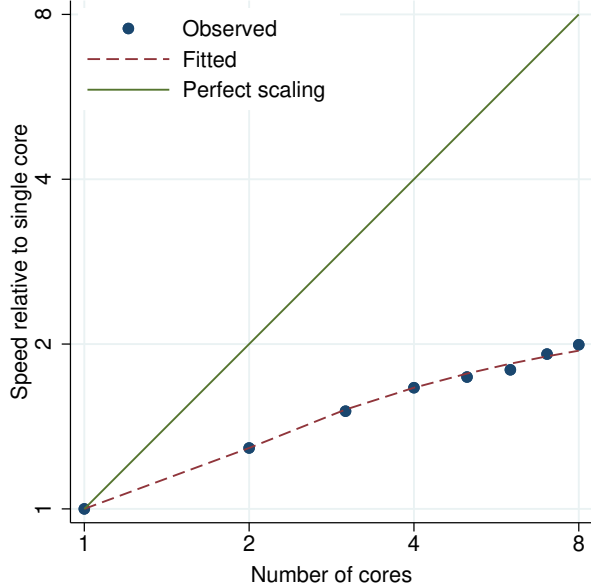


Figure 152. `expandc1 varname` performance plot.

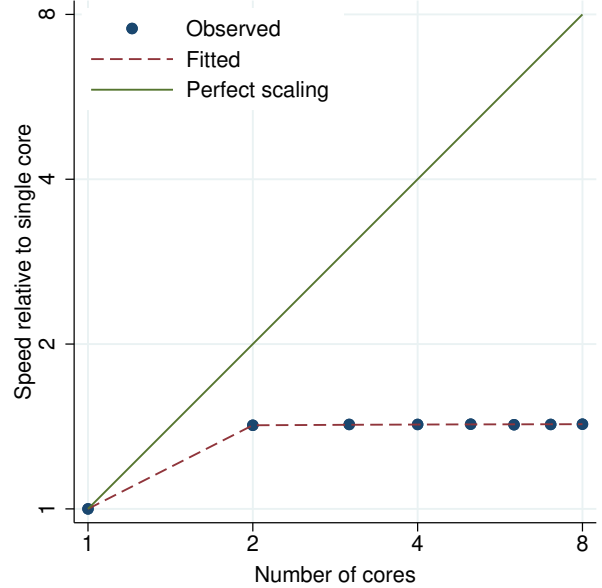


Figure 153. `expoisson` performance plot.

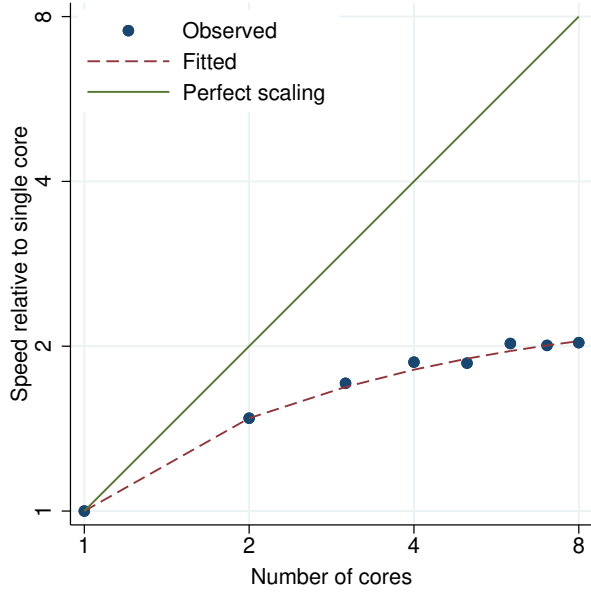


Figure 154. `factor` performance plot.

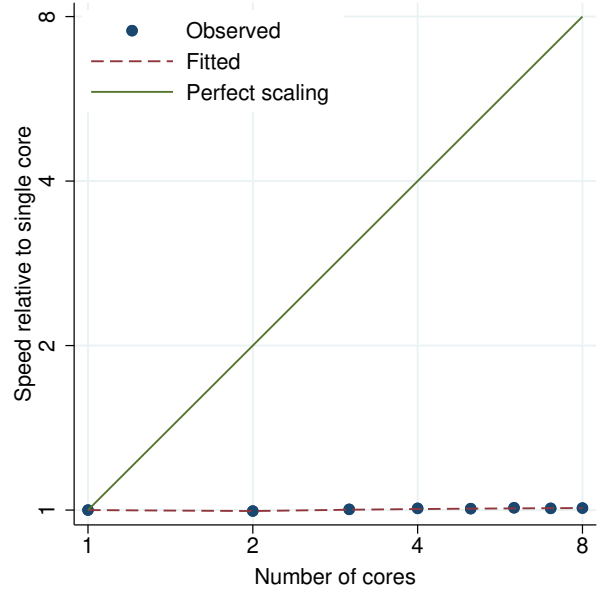


Figure 155. `fcast compute` performance plot.

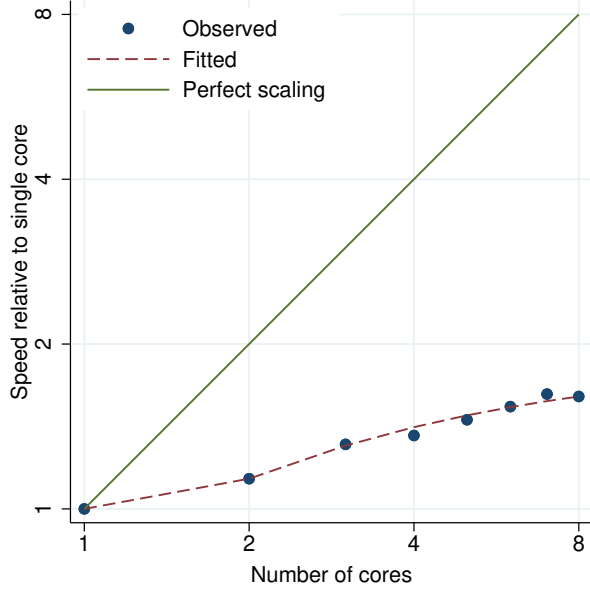


Figure 156. fillin performance plot.

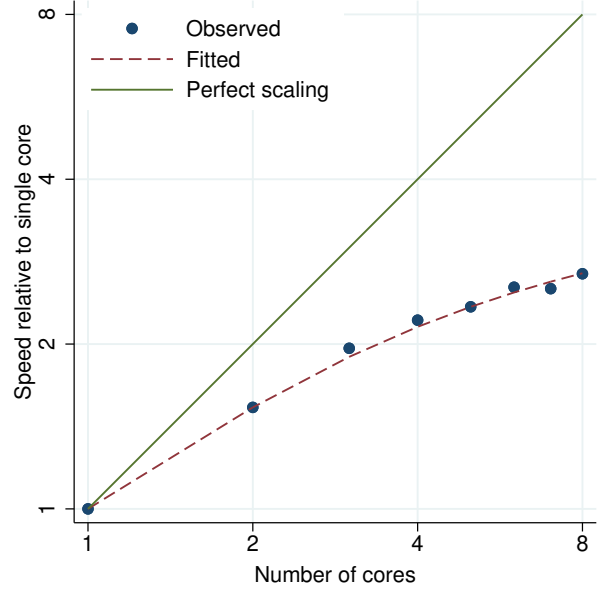


Figure 157. fmm 2: poisson performance plot.

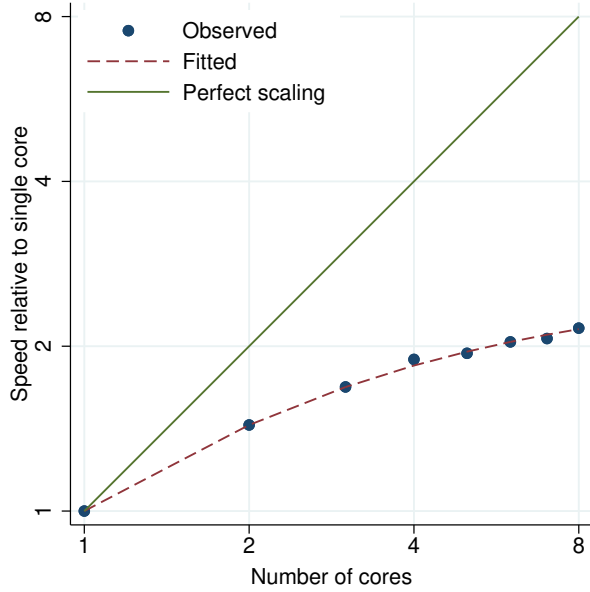


Figure 158. fmm 2: regress performance plot.

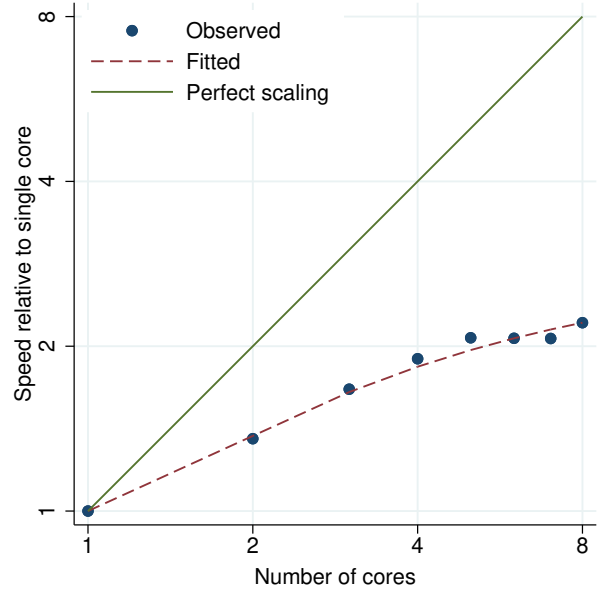


Figure 159. fmm 3: poisson performance plot.

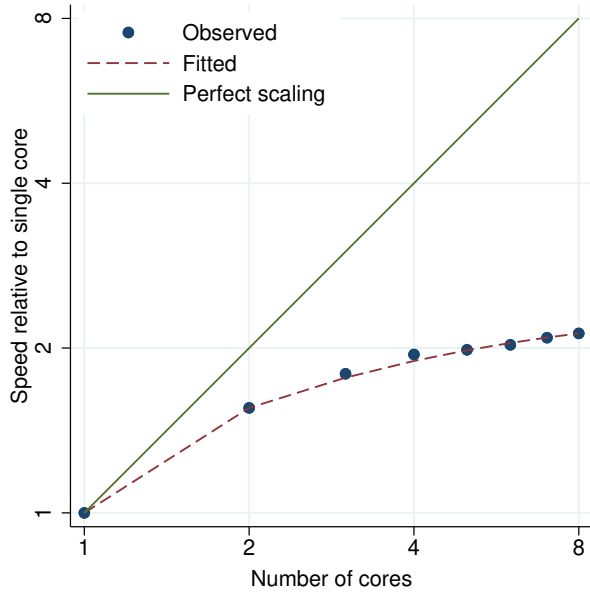


Figure 160. fmm 3: regress performance plot.

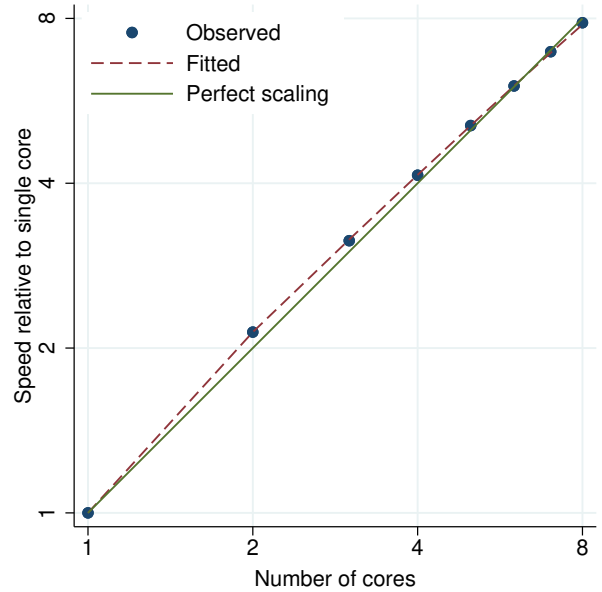


Figure 161. fracreg probit performance plot.

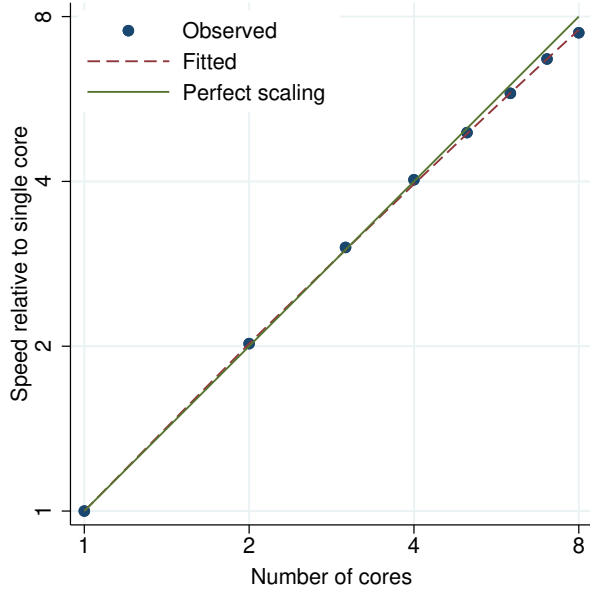


Figure 162. frontier performance plot.

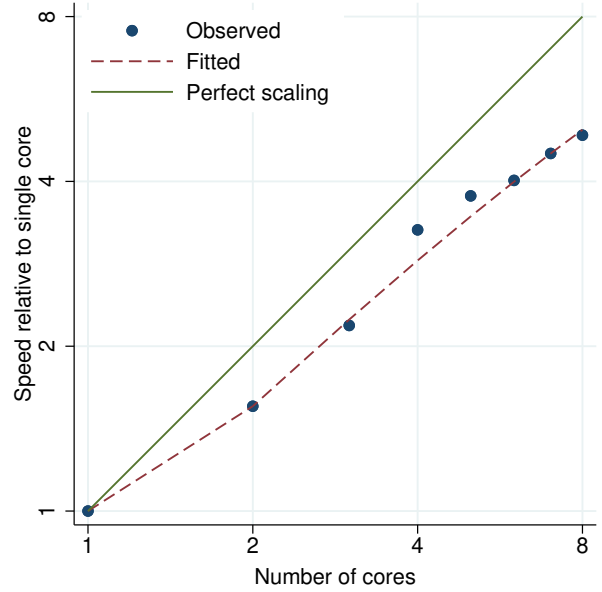


Figure 163. fvrevar (factors) performance plot.

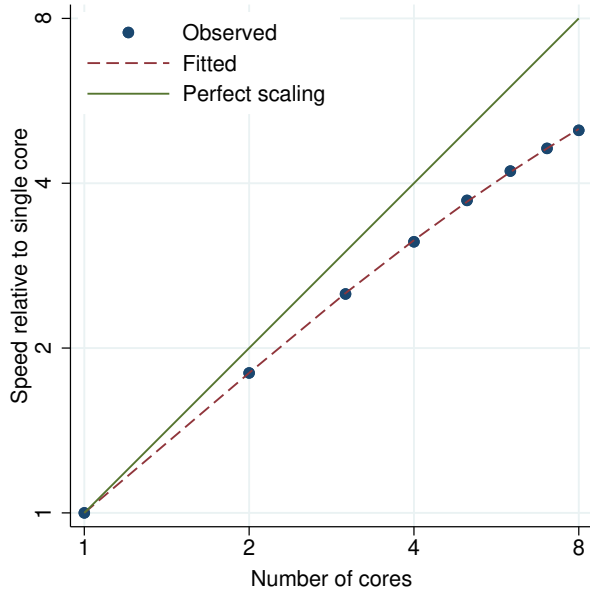


Figure 164. `fvrevvar` (interaction) performance plot.

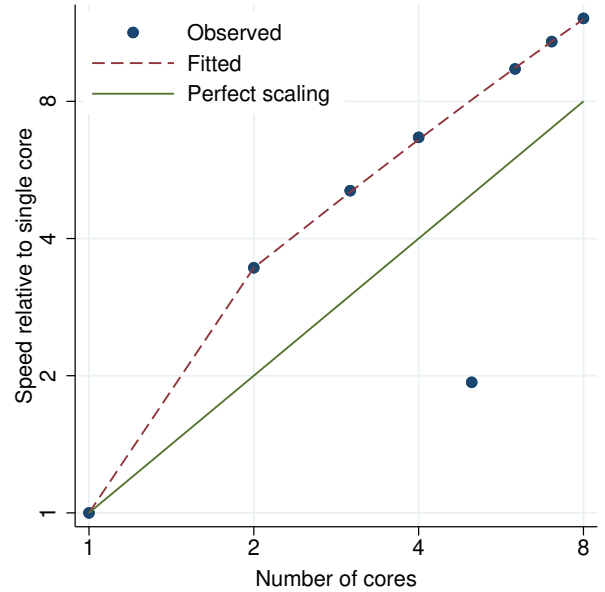


Figure 165. `generate` (small expressions) performance plot.

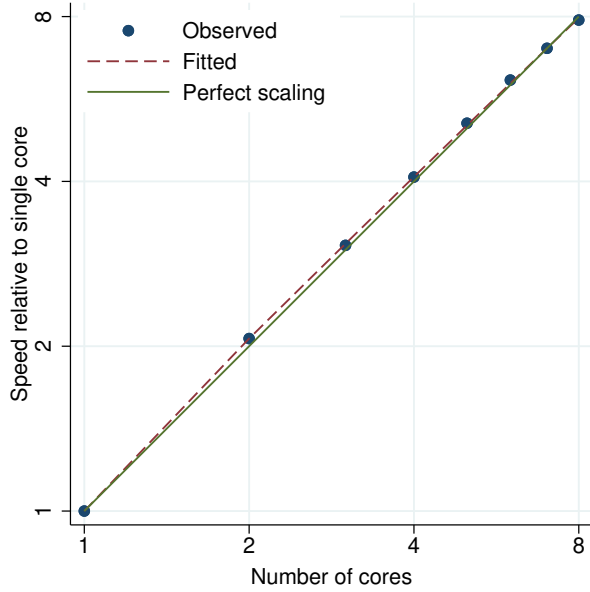


Figure 166. `generate` performance plot.

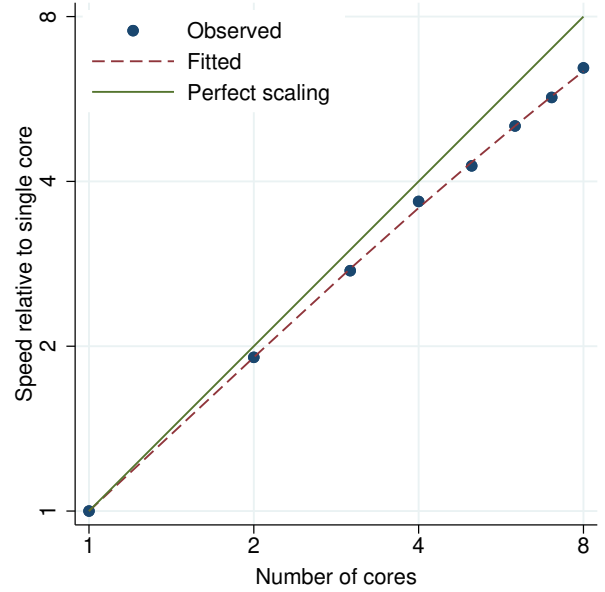


Figure 167. `glm, family(gamma)` performance plot.

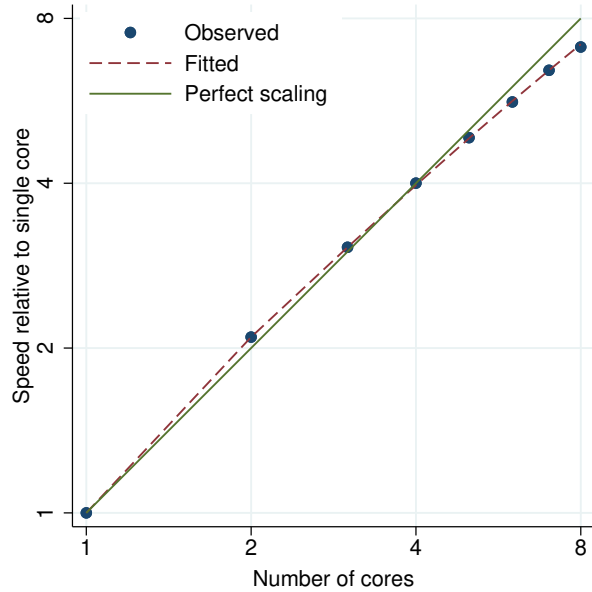


Figure 168. `glm, family(gaussian)` performance plot.

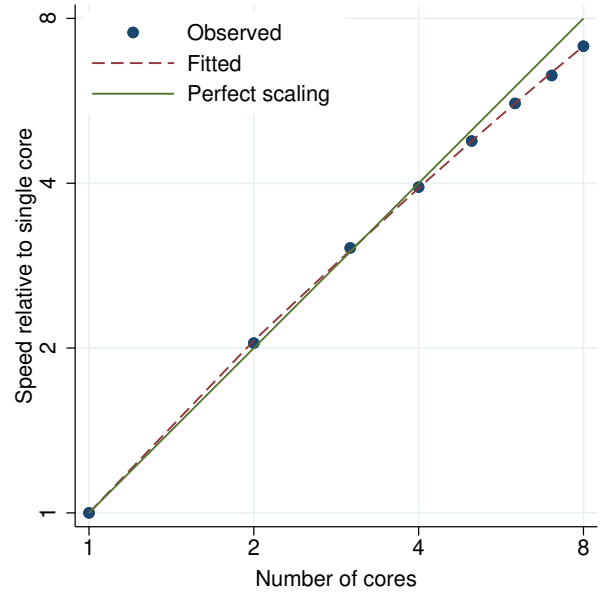


Figure 169. `glm, family(igaussian)` performance plot.

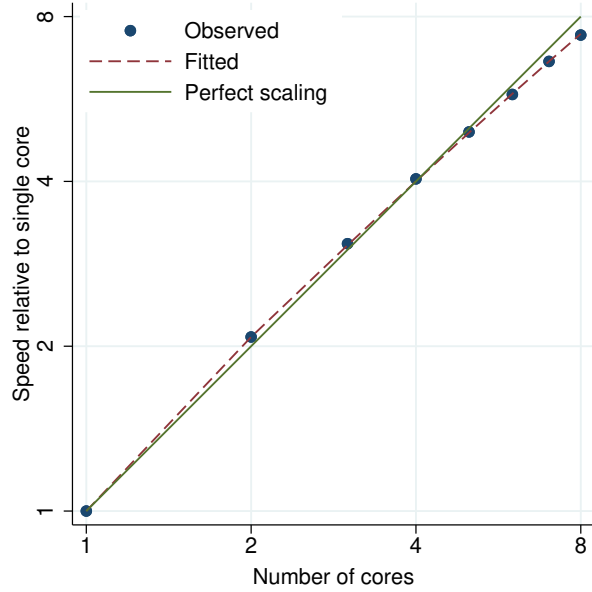


Figure 170. `glm, family(nbinomial)` performance plot.

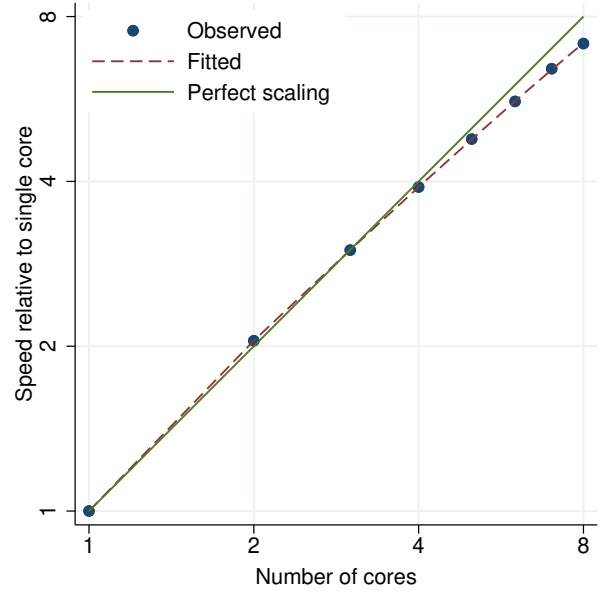


Figure 171. `glm, family(poisson)` performance plot.

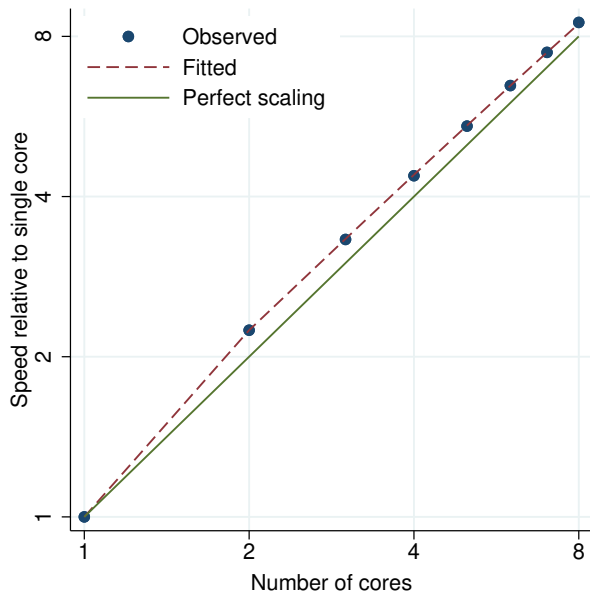


Figure 172. `glogit` performance plot.

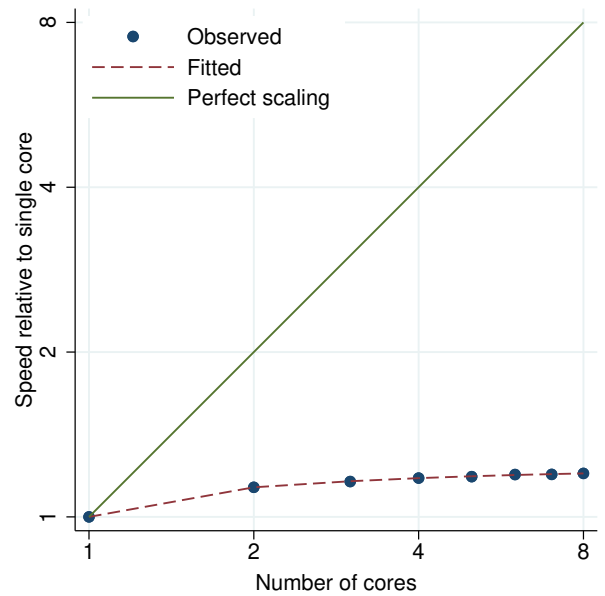


Figure 173. `gmm` performance plot.

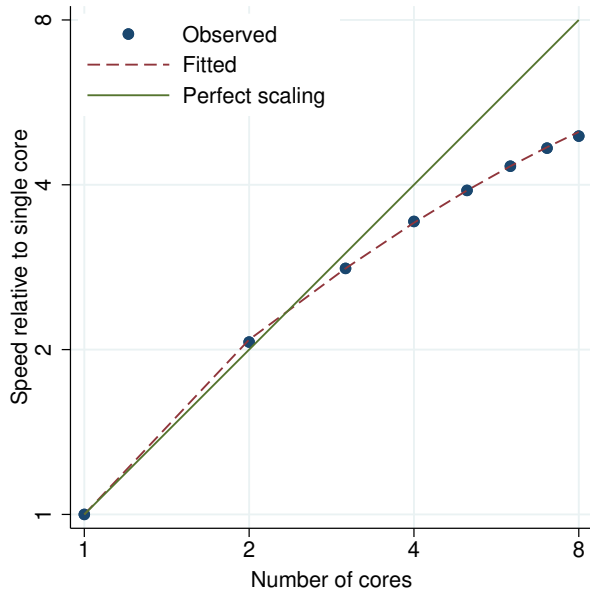


Figure 174. `gmm` (with derivatives) performance plot.

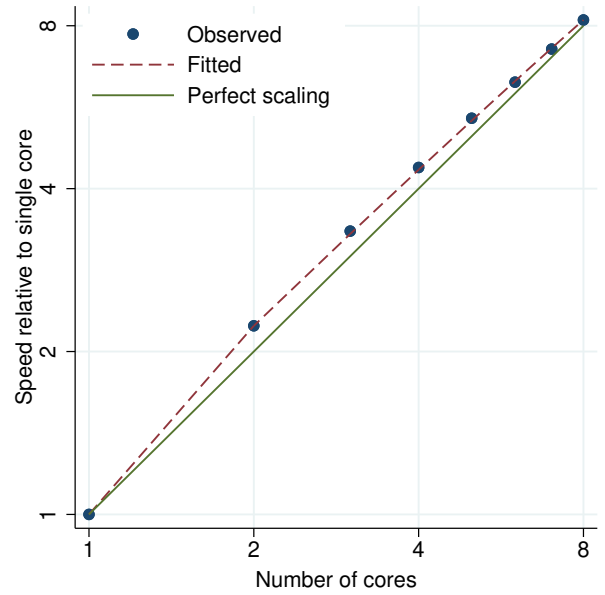


Figure 175. `gprobit` performance plot.

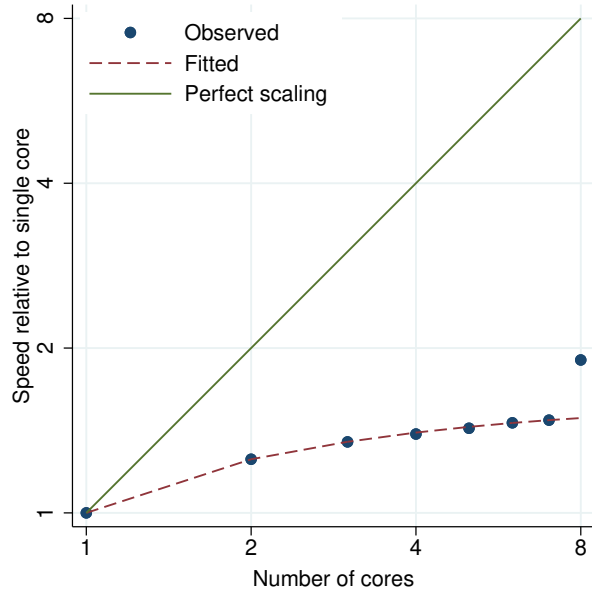


Figure 176. graph bar performance plot.

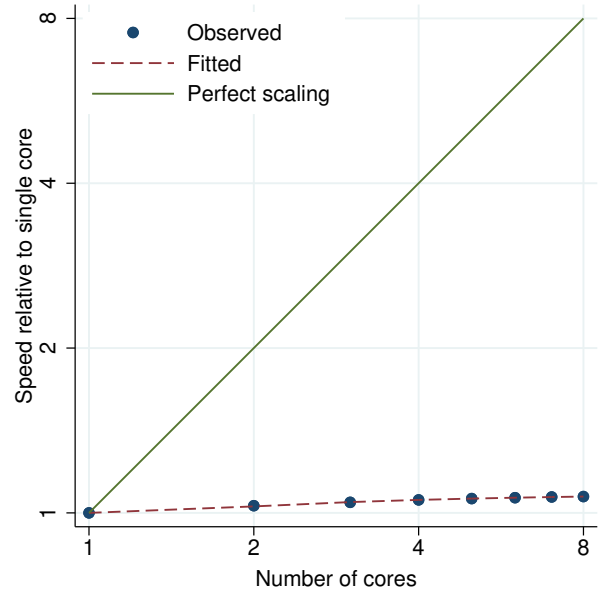


Figure 177. graph box performance plot.

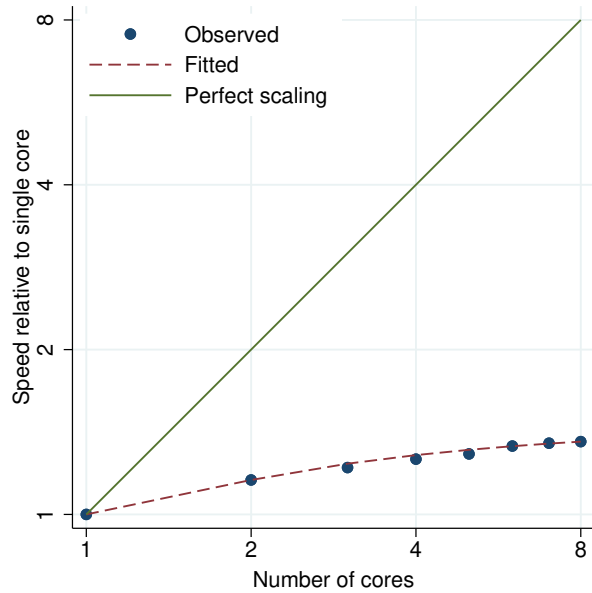


Figure 178. graph pie performance plot.

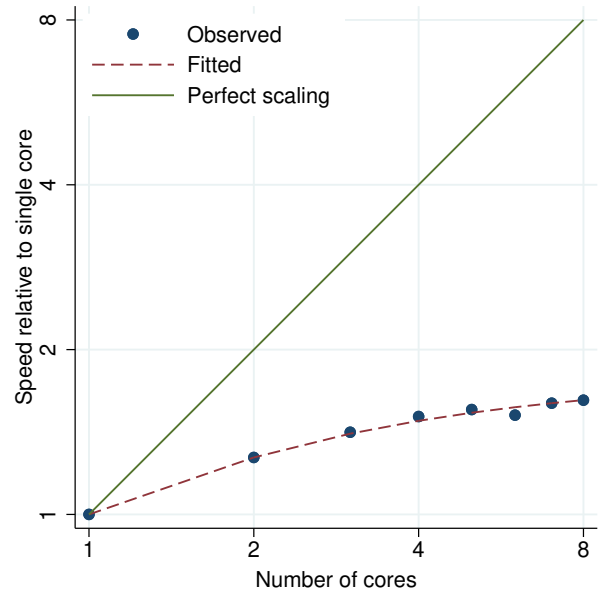


Figure 179. grmeanby performance plot.

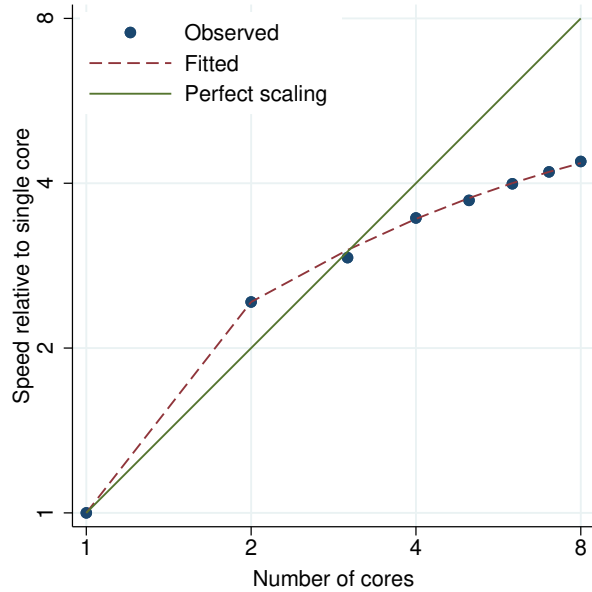


Figure 180. gsem, oprobit (CFA, 2-level) performance plot.

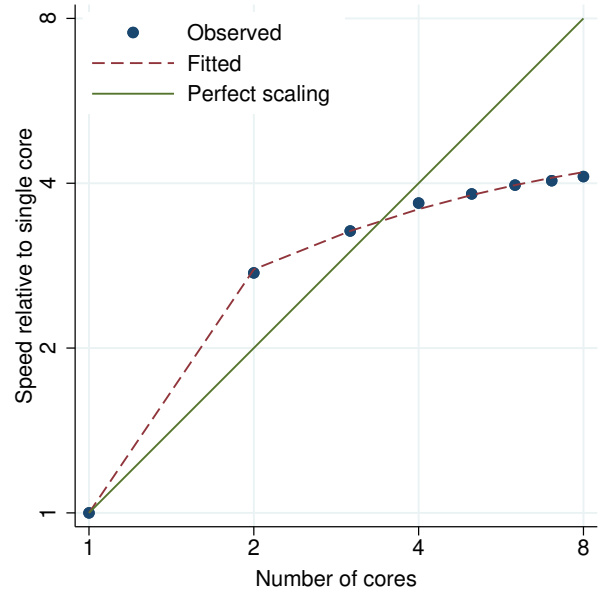


Figure 181. gsem, oprobit (CFA) performance plot.

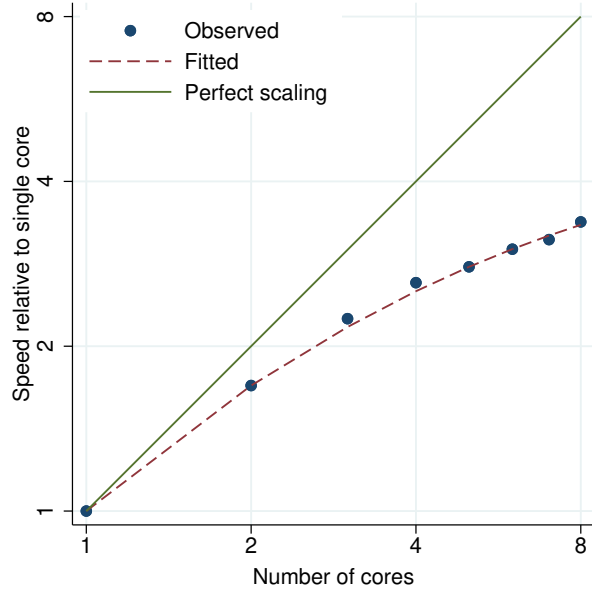


Figure 182. gsem, logit group() performance plot.

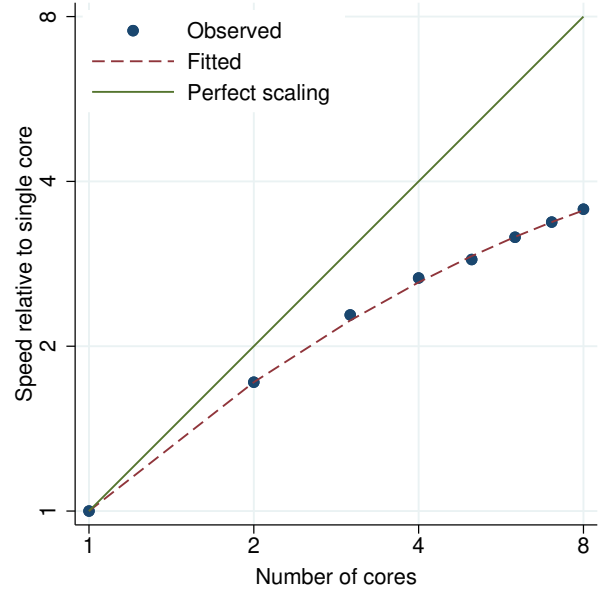


Figure 183. gsem, group() performance plot.

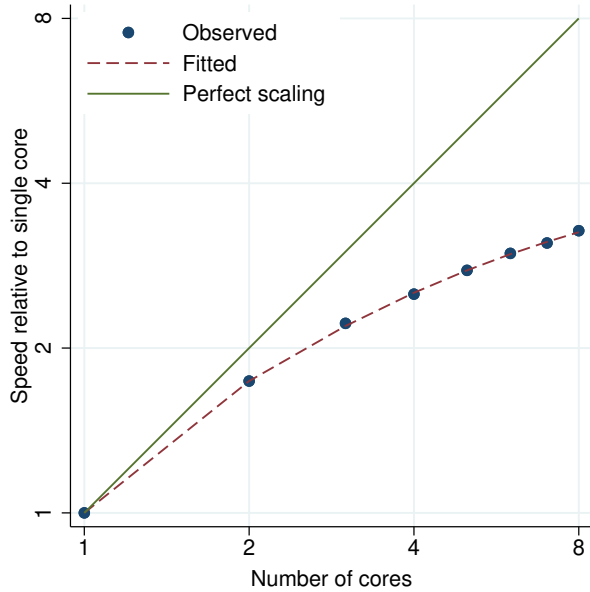


Figure 184. gsem, ologit group() performance plot.

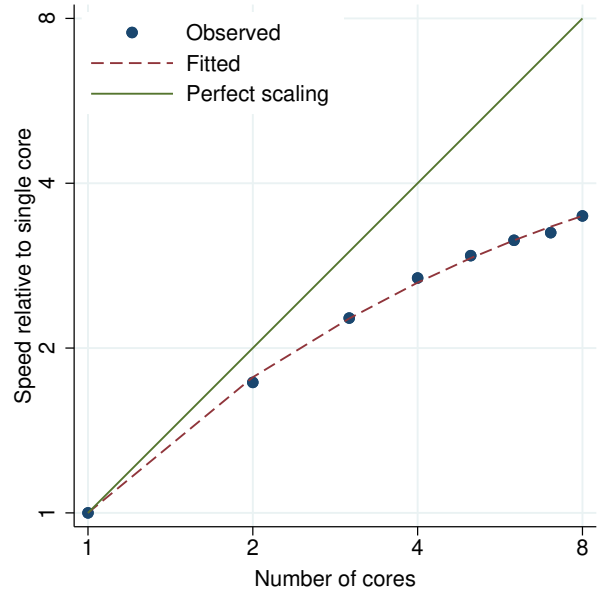


Figure 185. gsem, poisson group() performance plot.

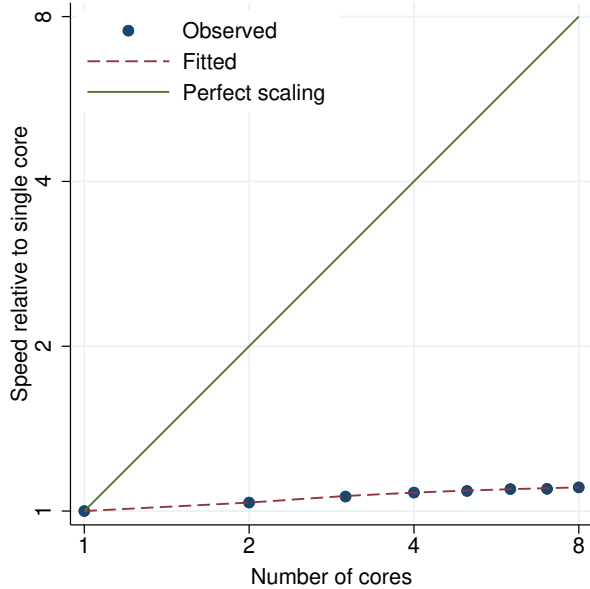


Figure 186. gsort performance plot.

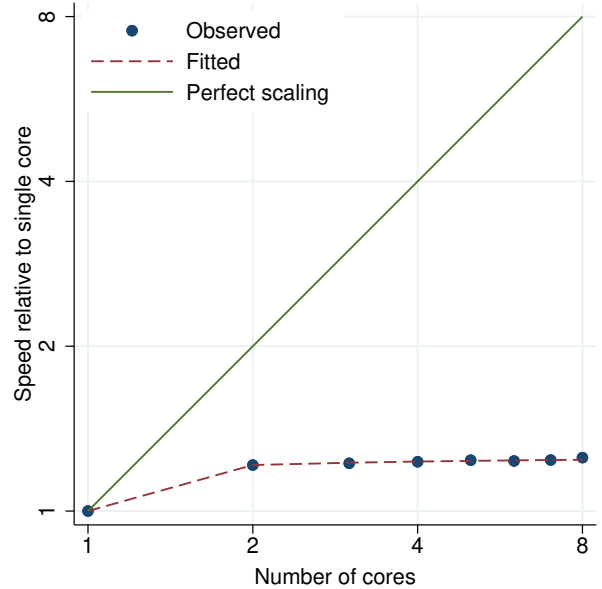


Figure 187. hausman performance plot.

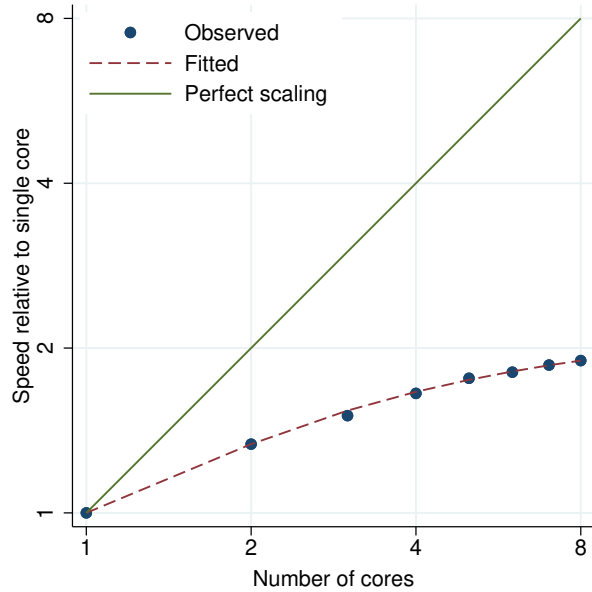


Figure 188. hdidregress aipw performance plot.

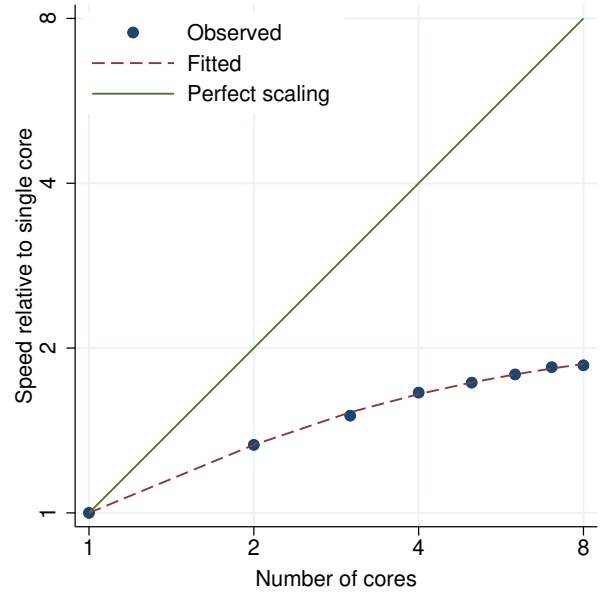


Figure 189. hdidregress ipw performance plot.

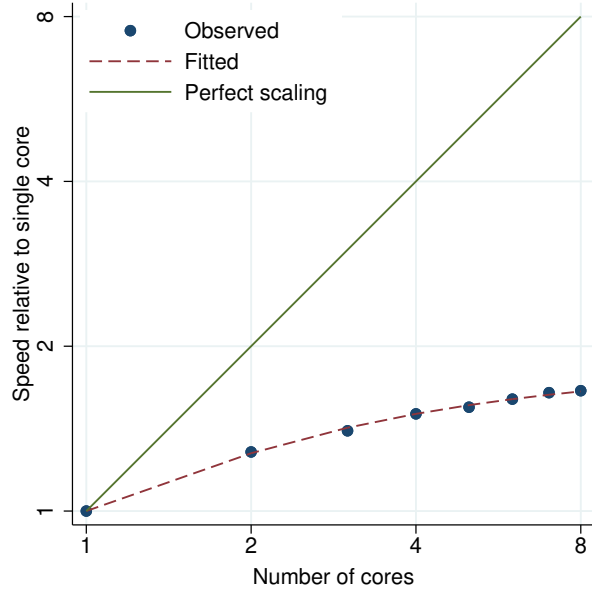


Figure 190. hdidregress ra performance plot.

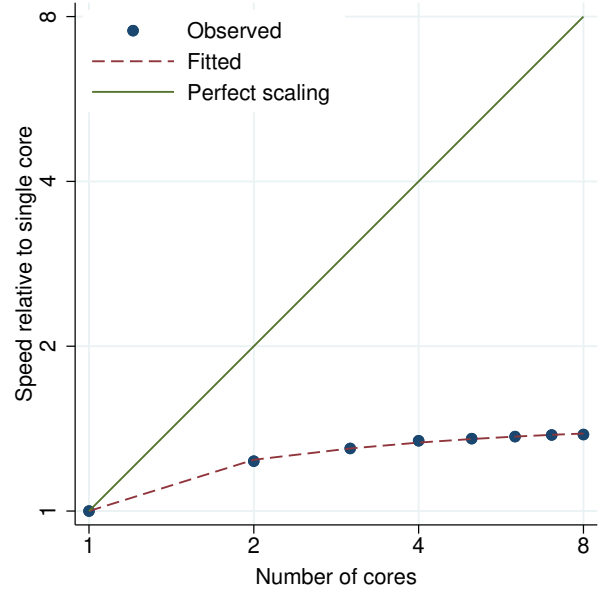


Figure 191. hdidregress twfe performance plot.

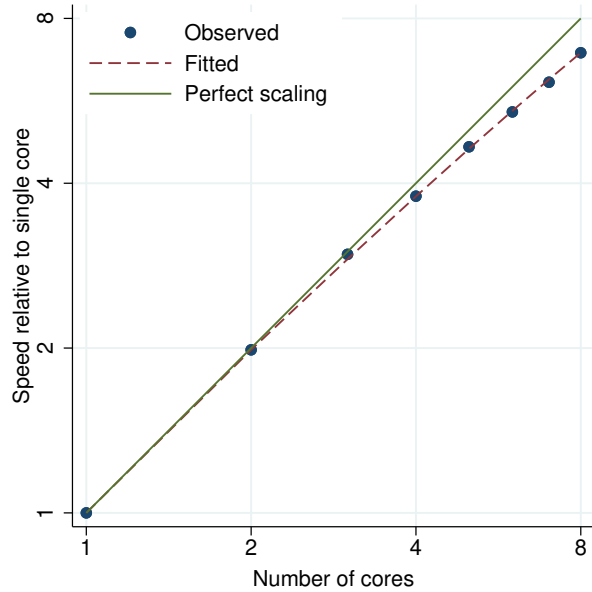


Figure 192. heckman performance plot.

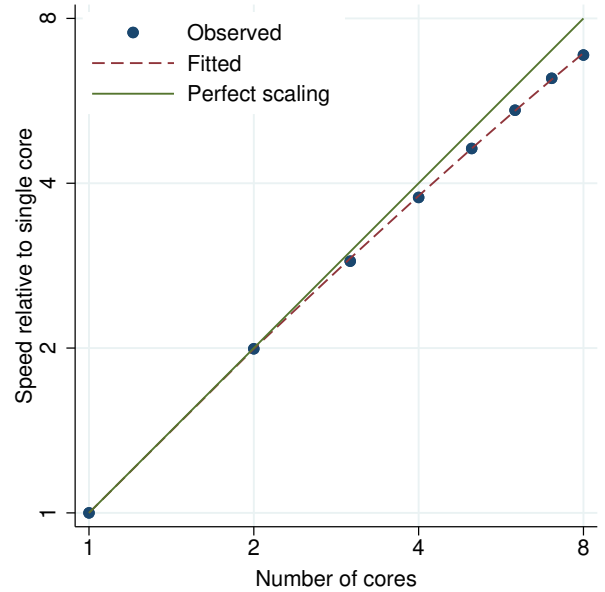


Figure 193. heckman, twostep performance plot.

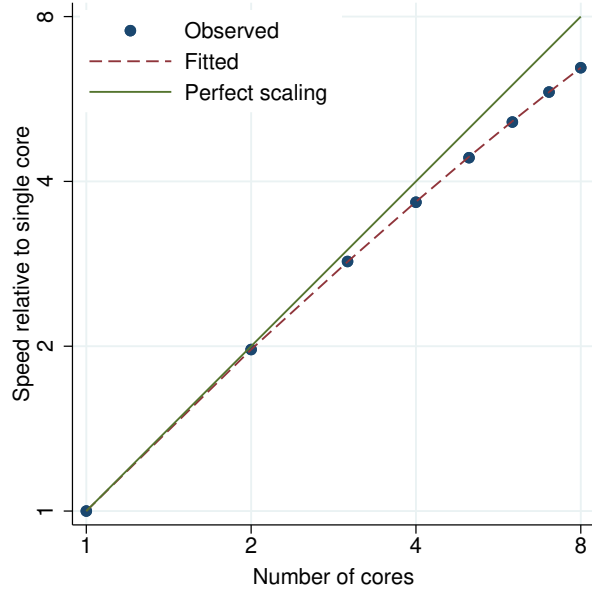


Figure 194. heckprobit performance plot.

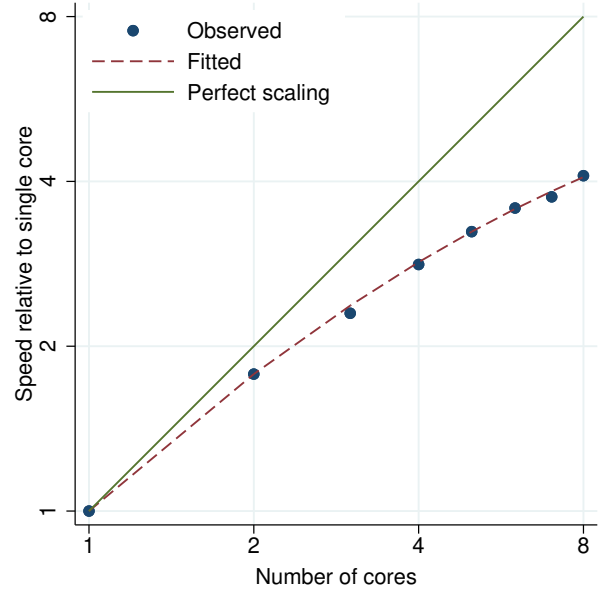


Figure 195. heckpoisson performance plot.

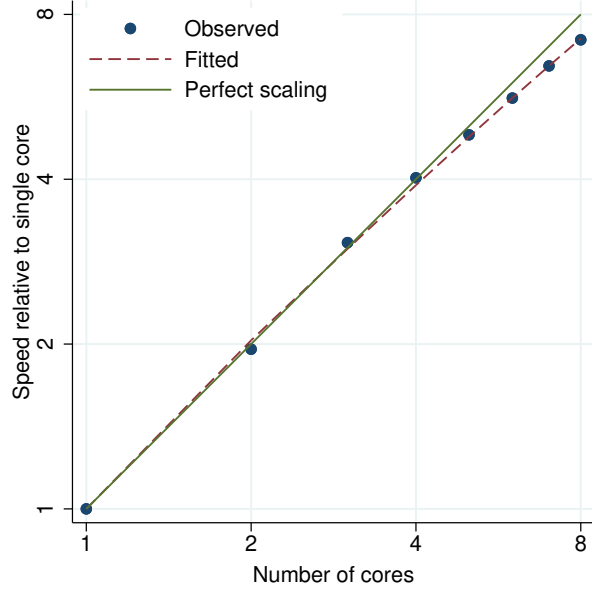


Figure 196. heckprob performance plot.

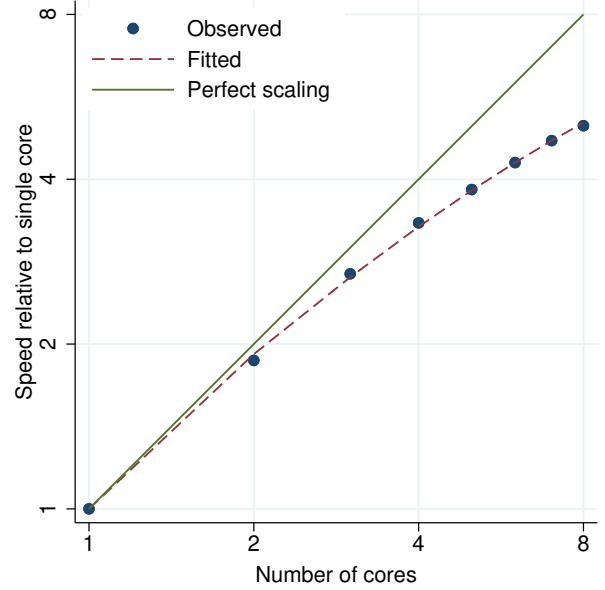


Figure 197. hetoprobit performance plot.

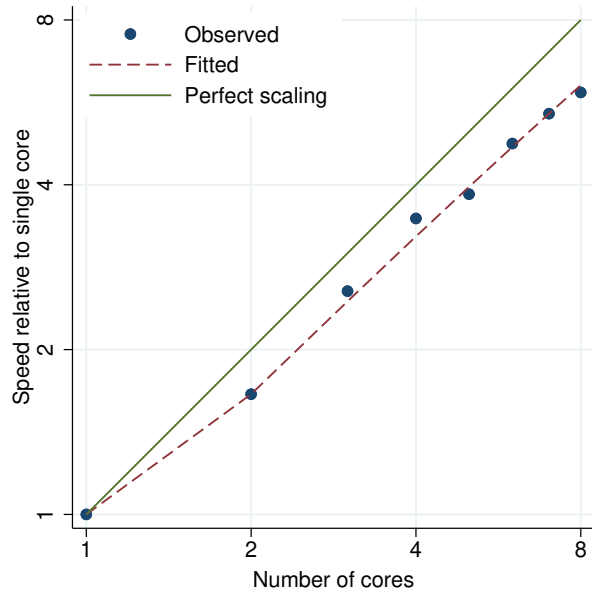


Figure 198. hetprob performance plot.

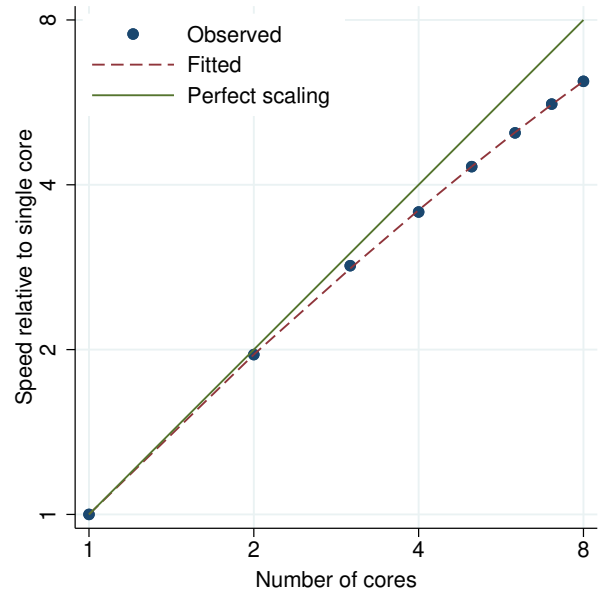


Figure 199. hetregress performance plot.

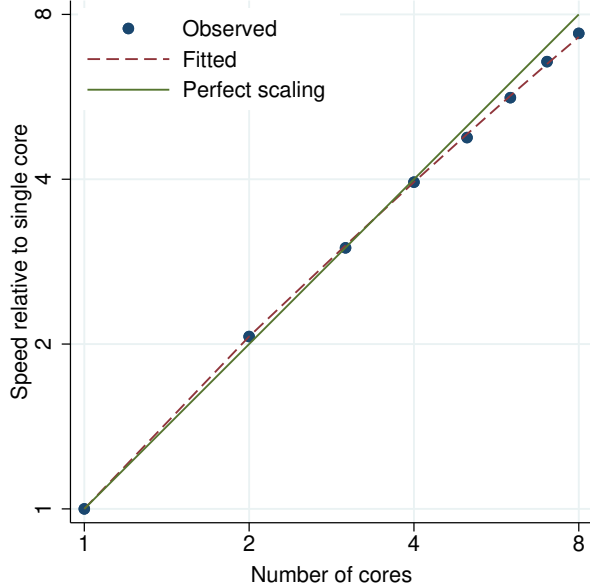


Figure 200. `hetregress`, `twostep` performance plot.

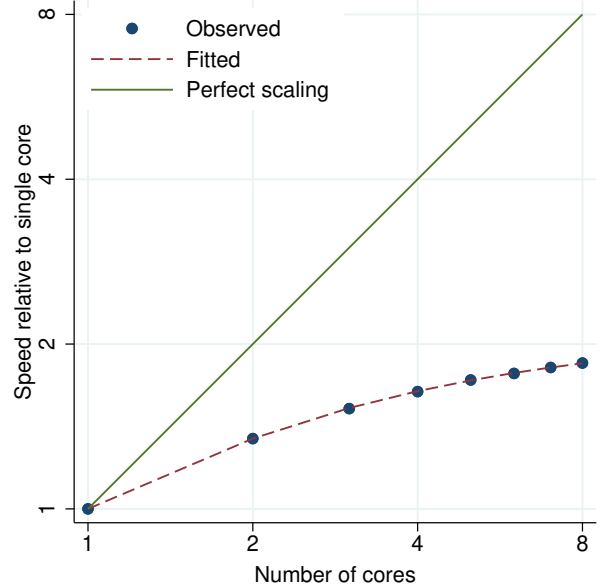


Figure 201. histogram performance plot.

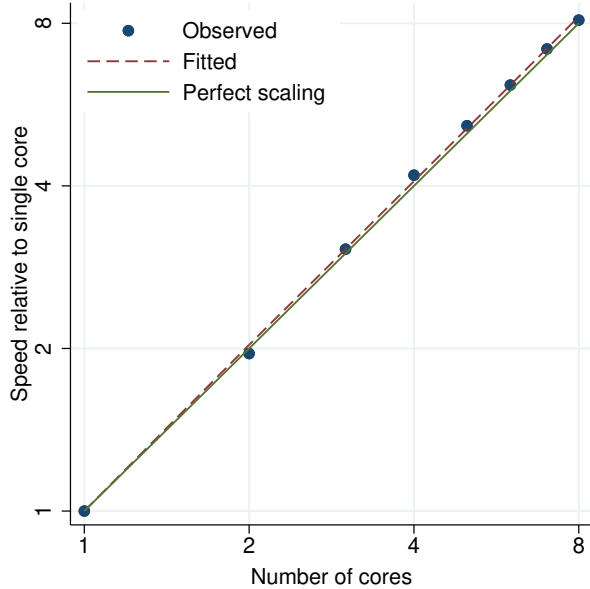


Figure 202. hotelling performance plot.

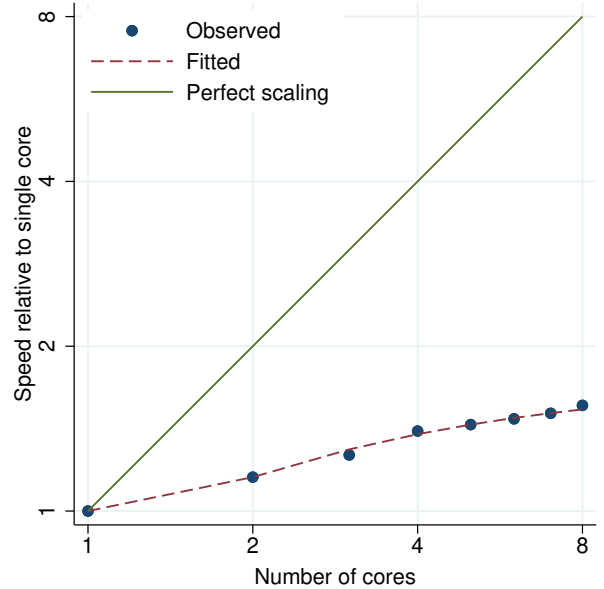


Figure 203. `icc`, `mixed` performance plot.

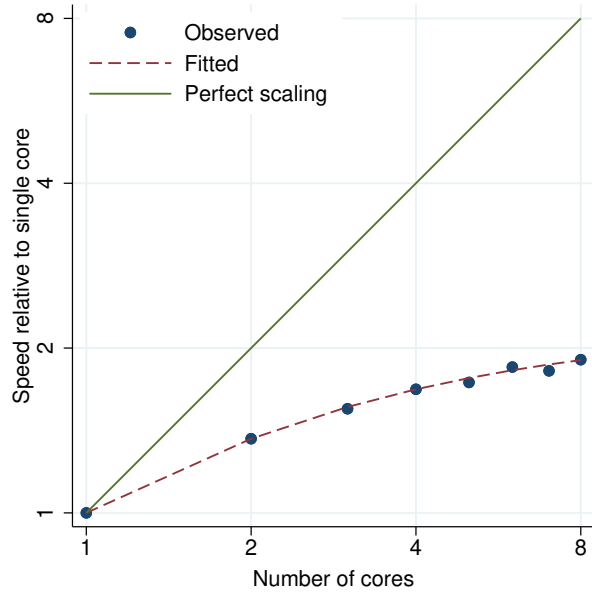


Figure 204. icc (one-way) performance plot.

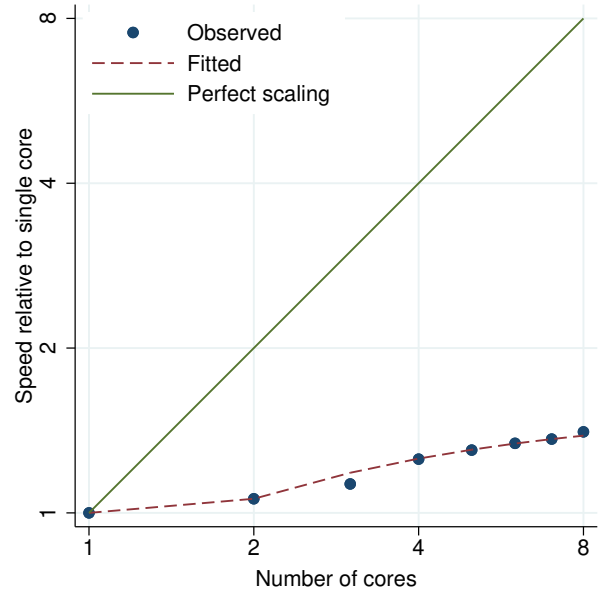


Figure 205. icc (two-way) performance plot.

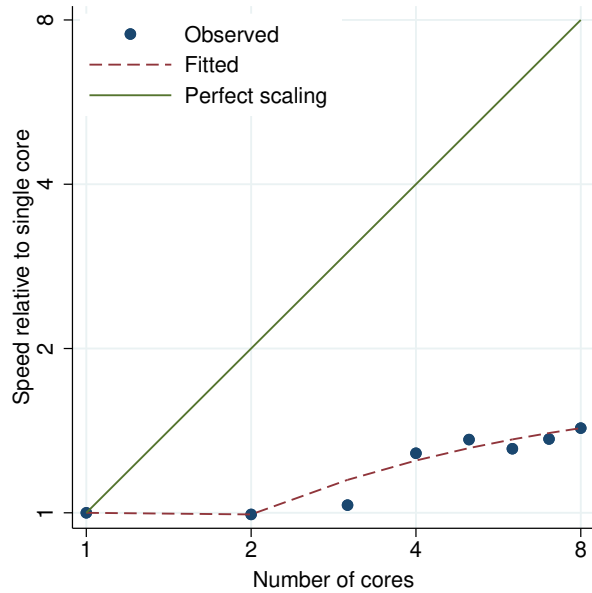


Figure 206. import delimited performance plot.

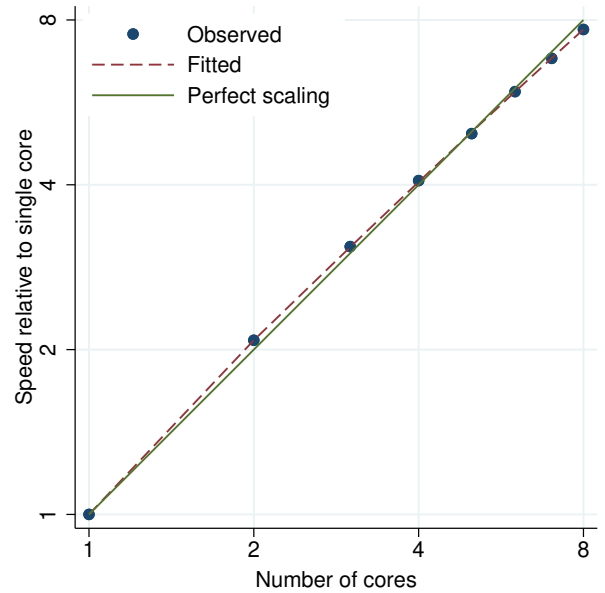


Figure 207. intreg performance plot.

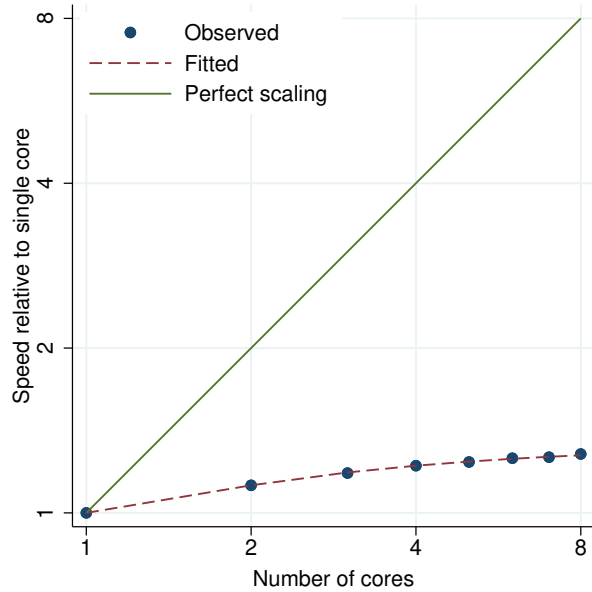


Figure 208. ir performance plot.

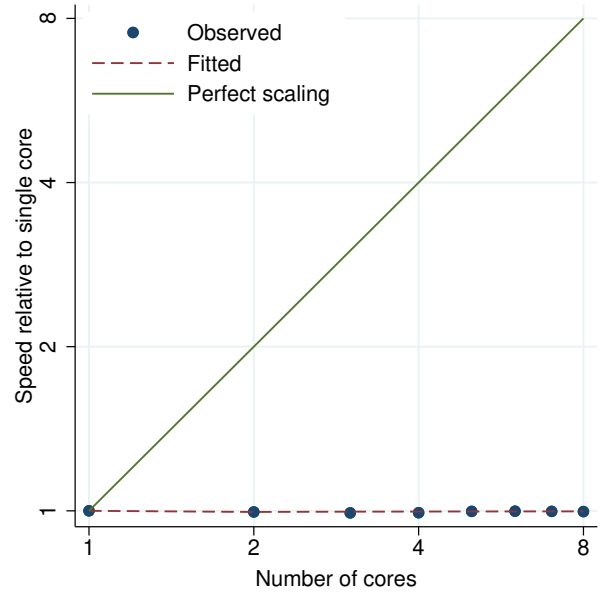


Figure 209. by: ir performance plot.

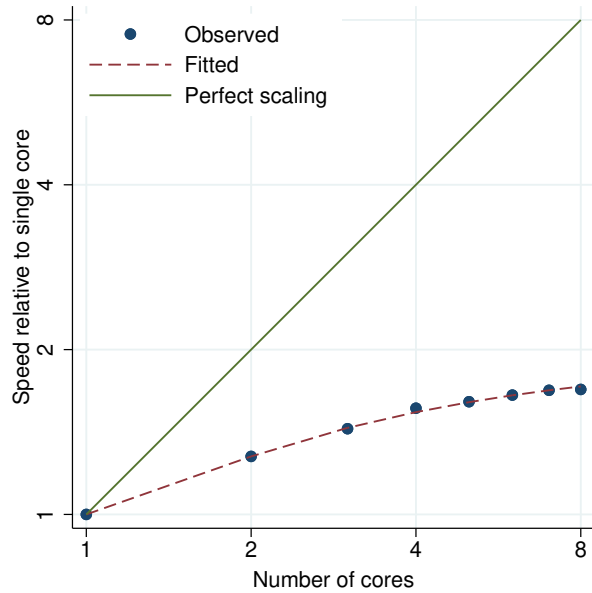


Figure 210. irf create performance plot.

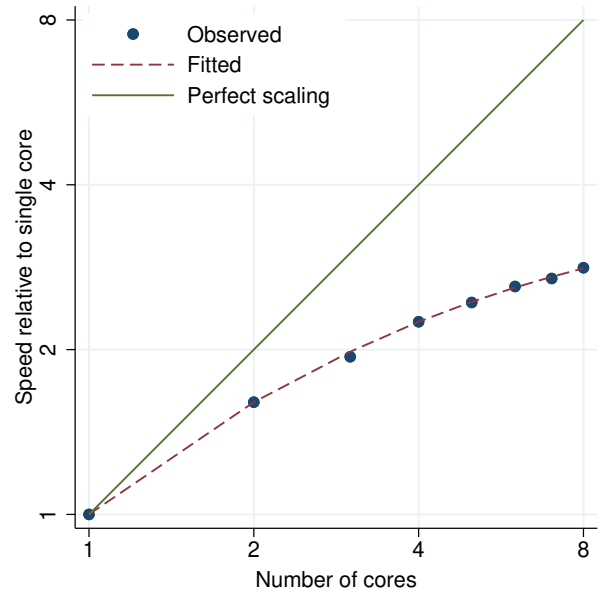


Figure 211. irt 1p1 performance plot.

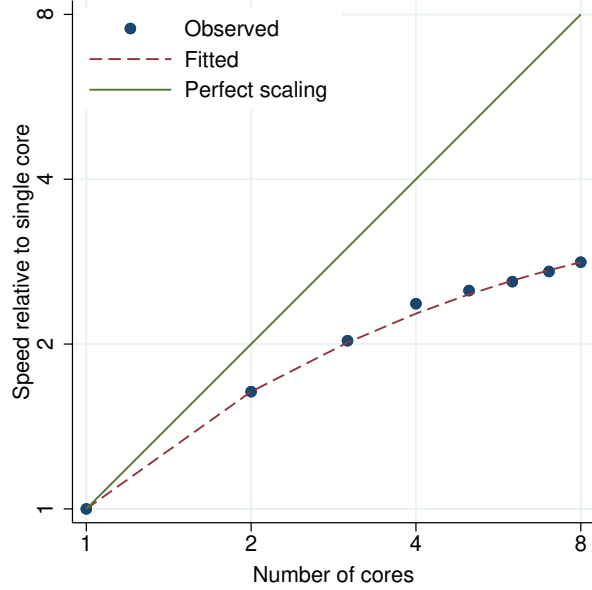


Figure 212. irt 2p1 performance plot.

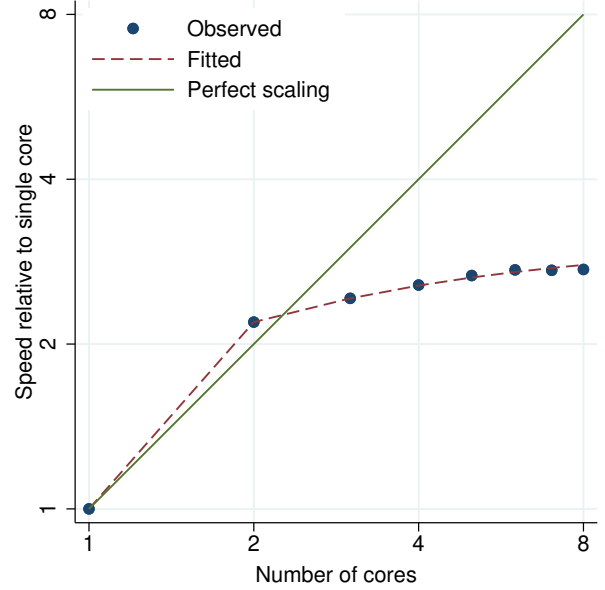


Figure 213. irt 3p1 performance plot.

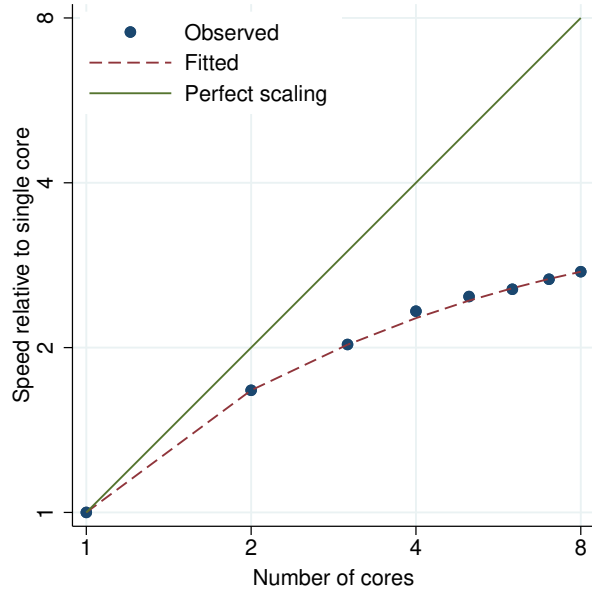


Figure 214. irt grm performance plot.

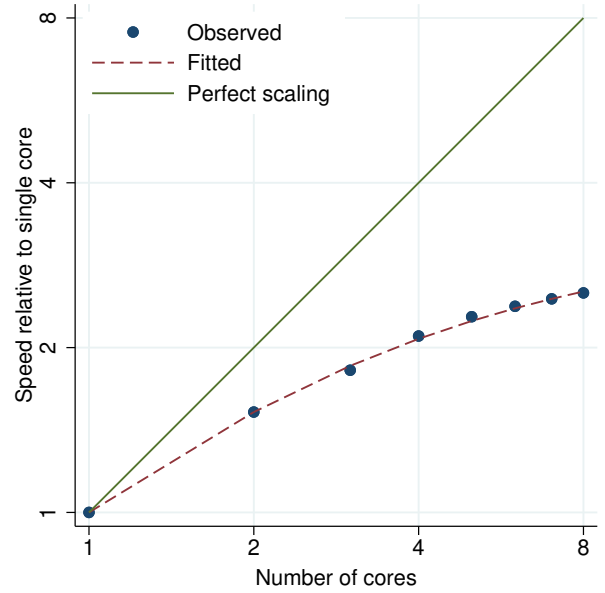


Figure 215. irt nrm performance plot.

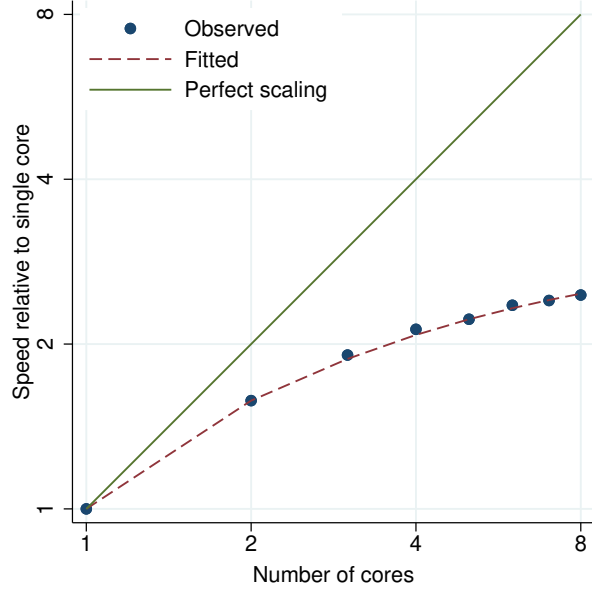


Figure 216. irt pcm performance plot.

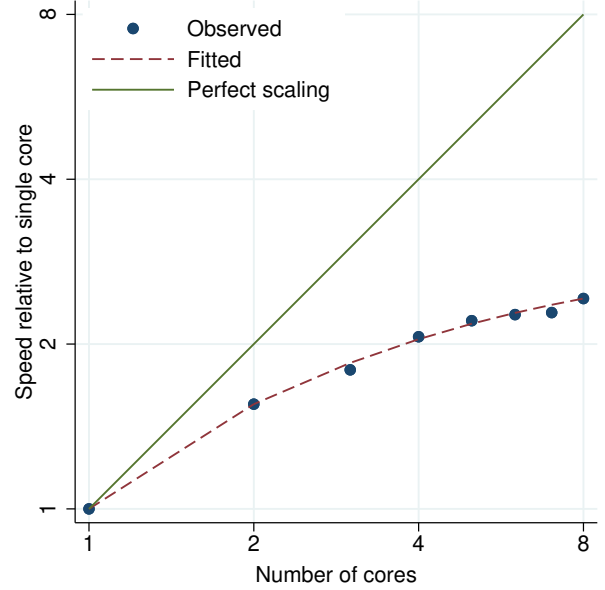


Figure 217. irt rsm performance plot.

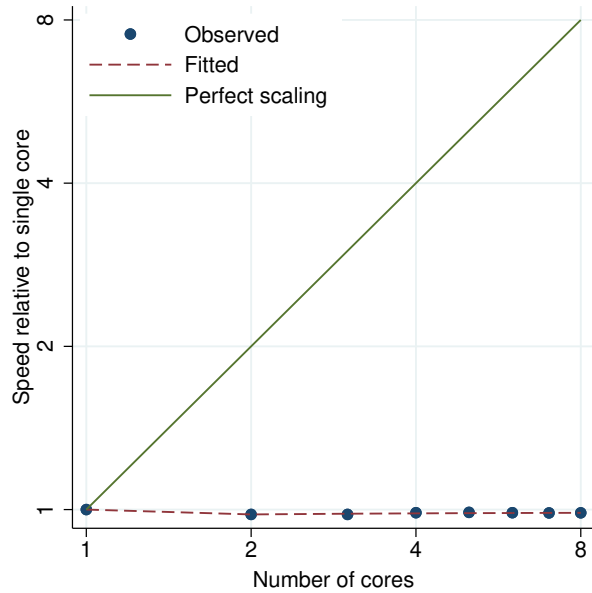


Figure 218. istsize performance plot.

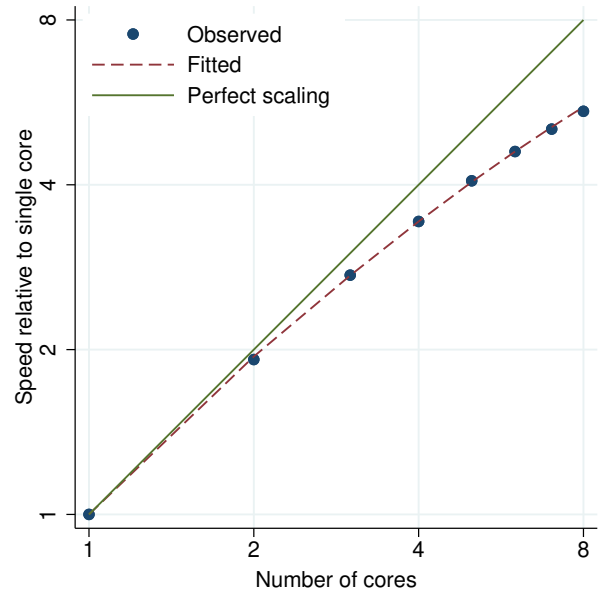


Figure 219. ivfprobit performance plot.

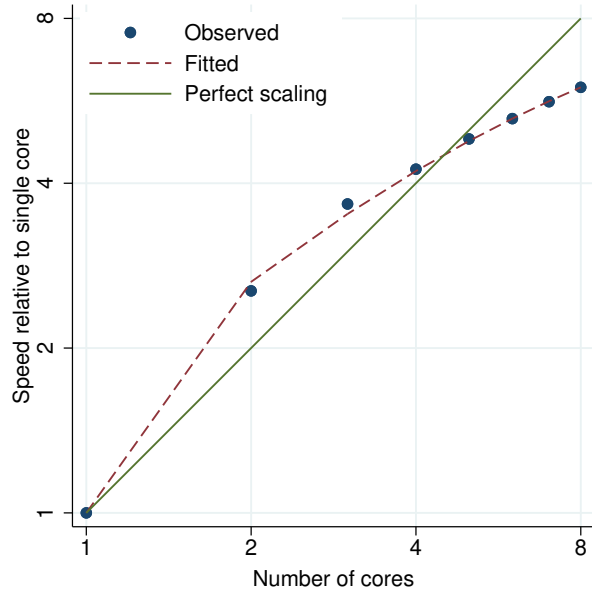


Figure 220. ivpoisson cfunction performance plot.

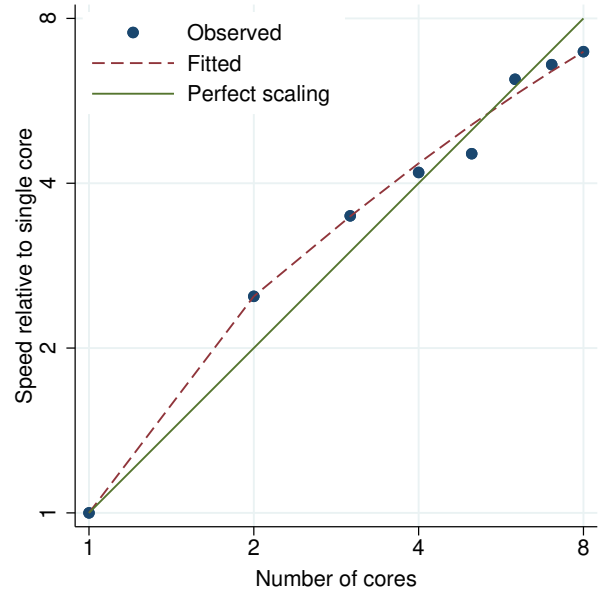


Figure 221. ivpoisson gmm, additive performance plot.

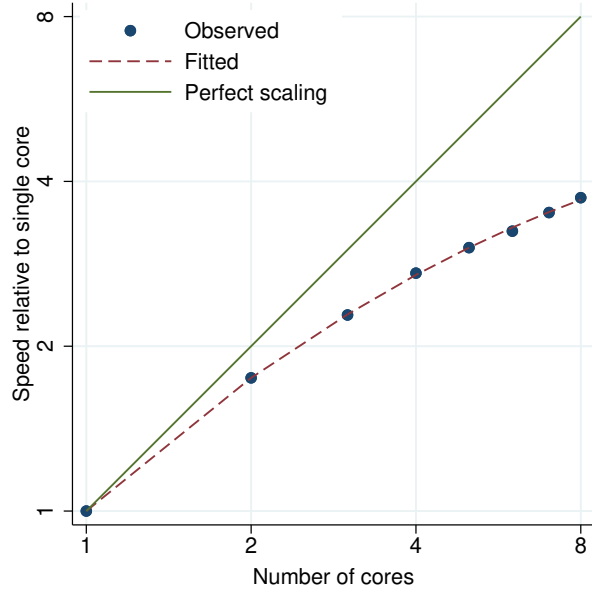


Figure 222. ivpoisson gmm, multiplicative performance plot.

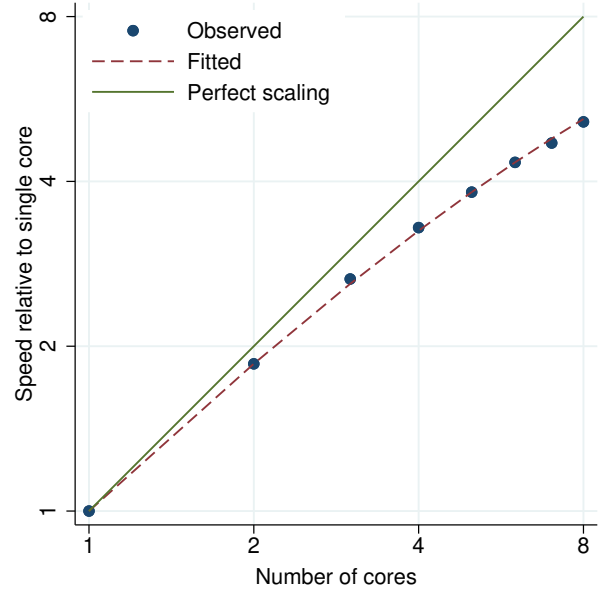


Figure 223. ivprobit performance plot.

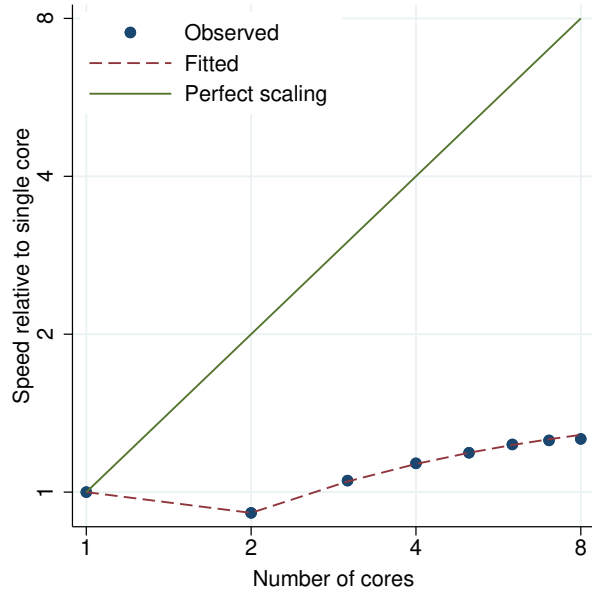


Figure 224. ivqregress iqr performance plot.

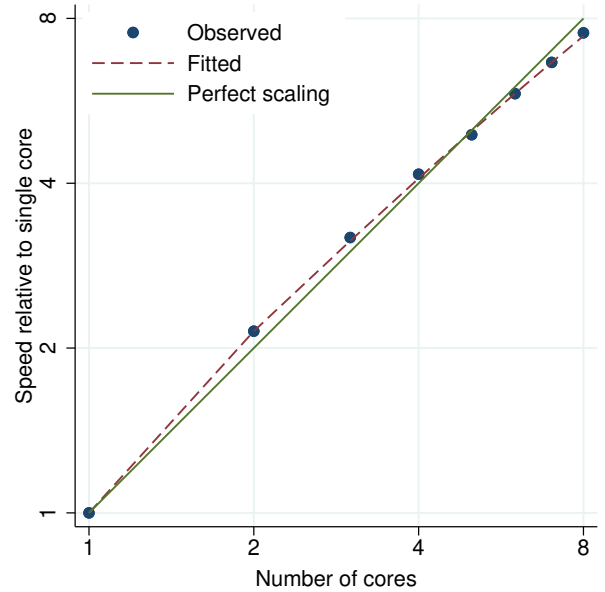


Figure 225. ivregress 2sls performance plot.

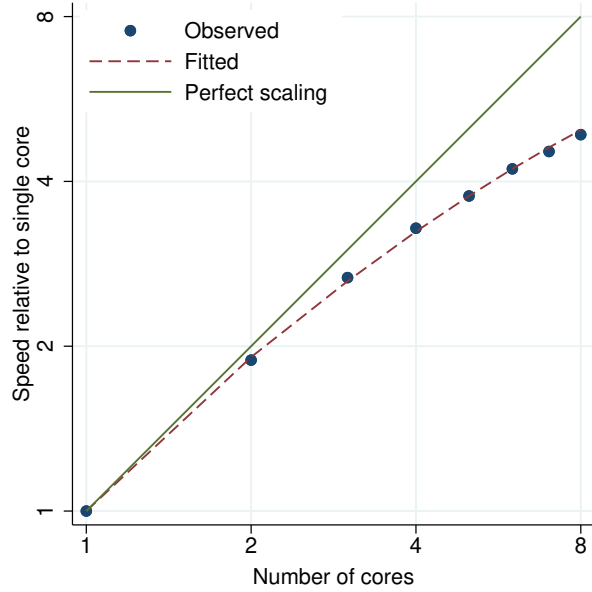


Figure 226. ivregress gmm performance plot.

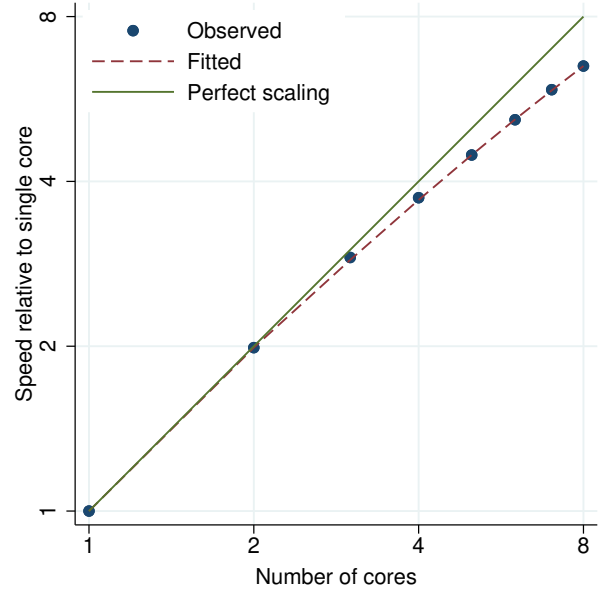


Figure 227. ivregress lml performance plot.

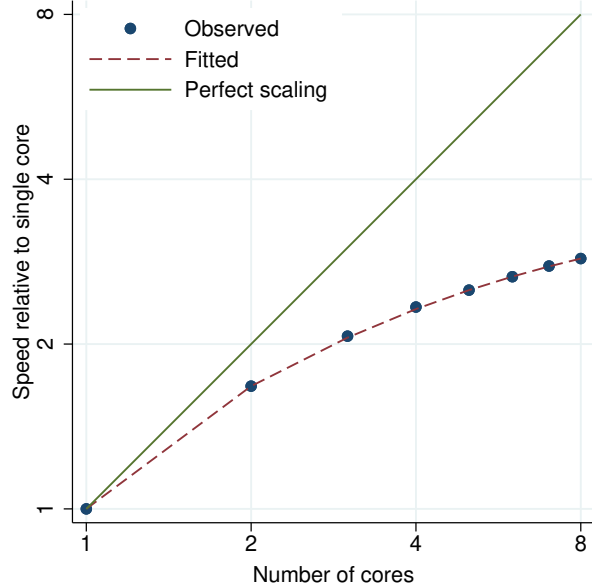


Figure 228. ivtobit performance plot.

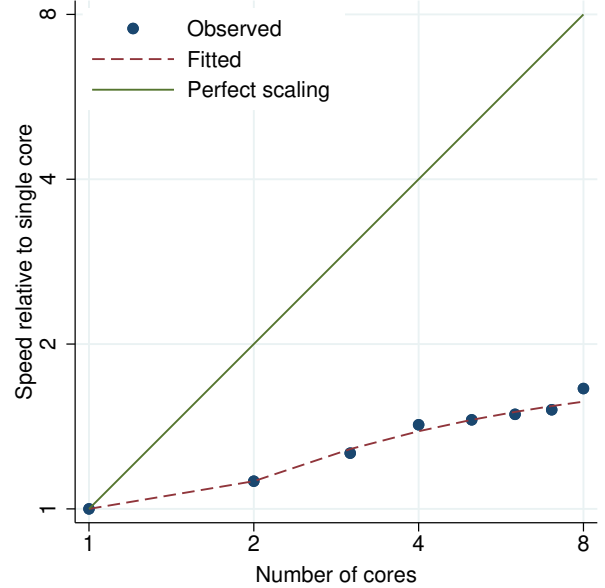


Figure 229. kap performance plot.

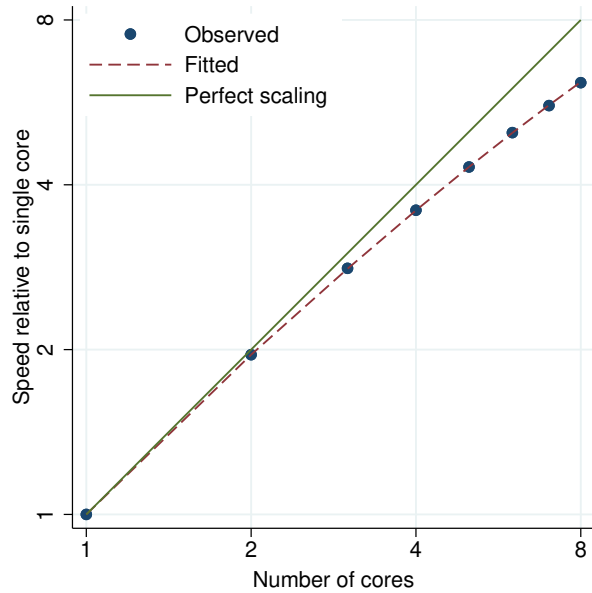


Figure 230. kappa performance plot.

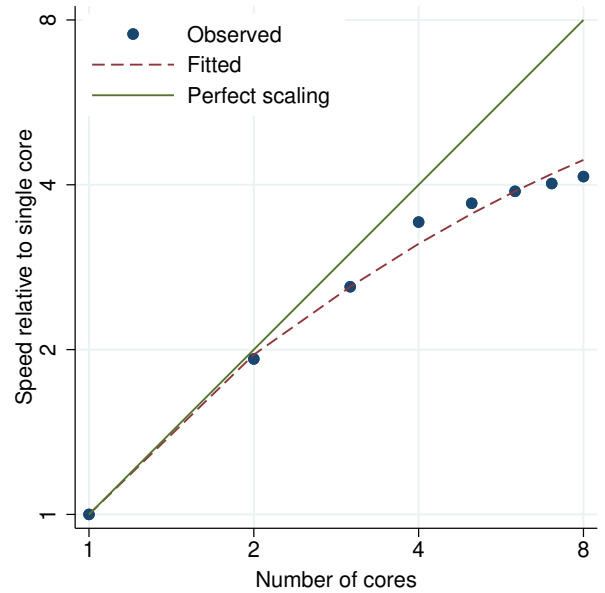


Figure 231. kdensity performance plot.

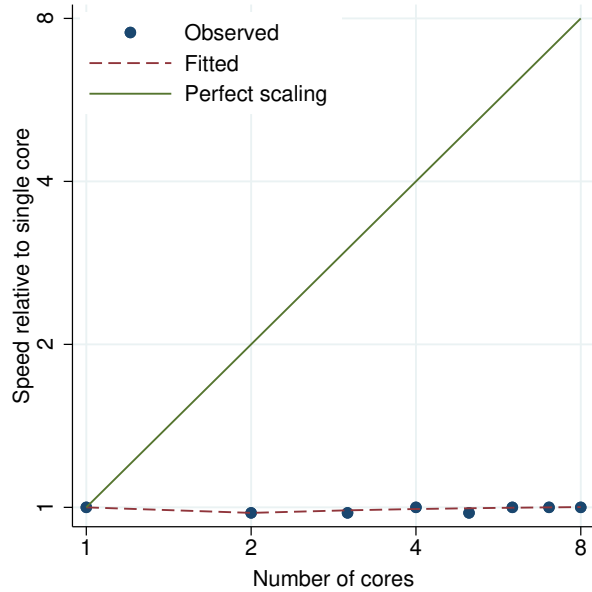


Figure 232. keep if *exp* performance plot.

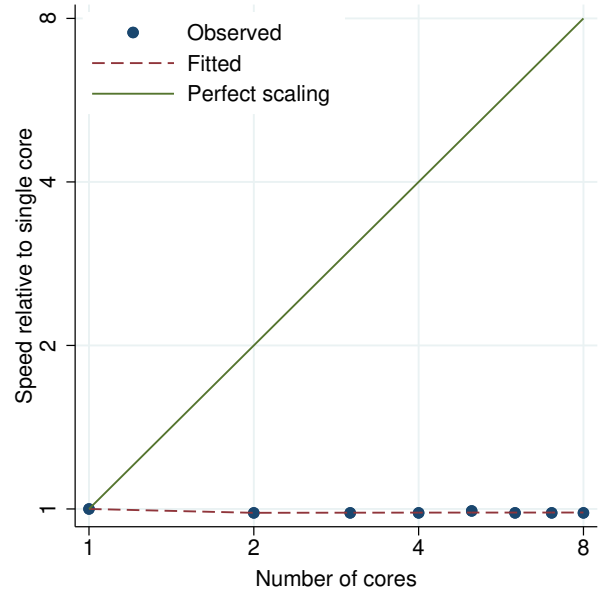


Figure 233. keep in *range* performance plot.

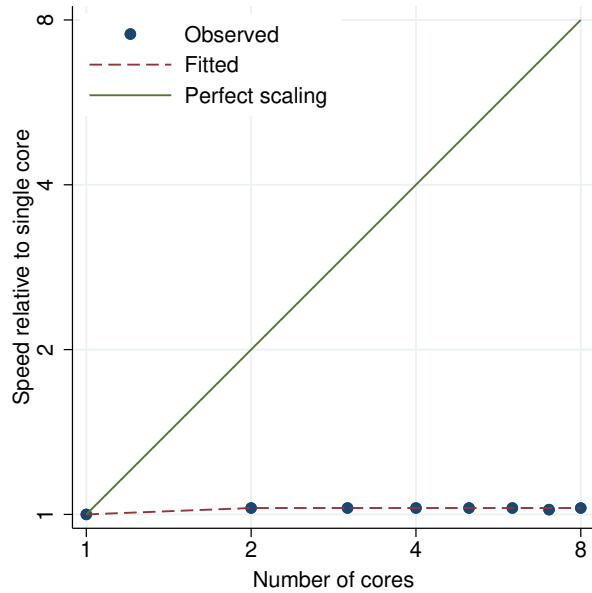


Figure 234. keep *varlist* performance plot.

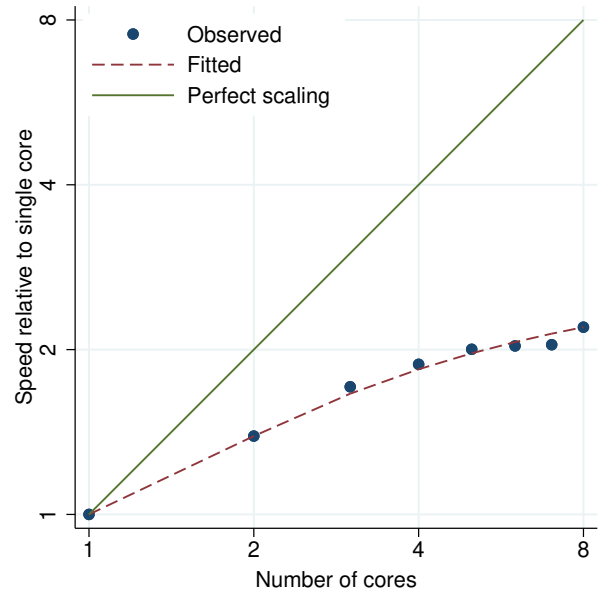


Figure 235. ksmirnov performance plot.

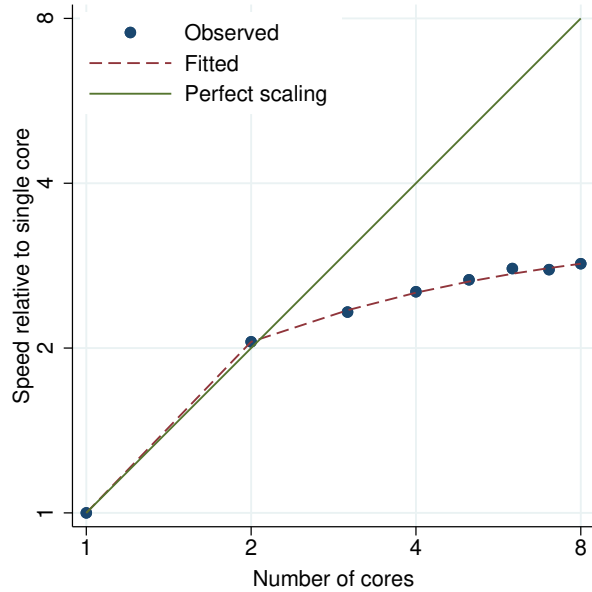


Figure 236. ksmirnov, by() performance plot.

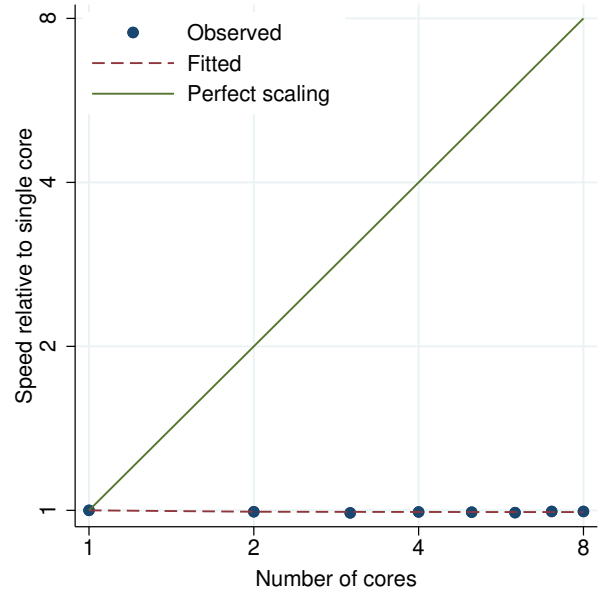


Figure 237. ktau performance plot.

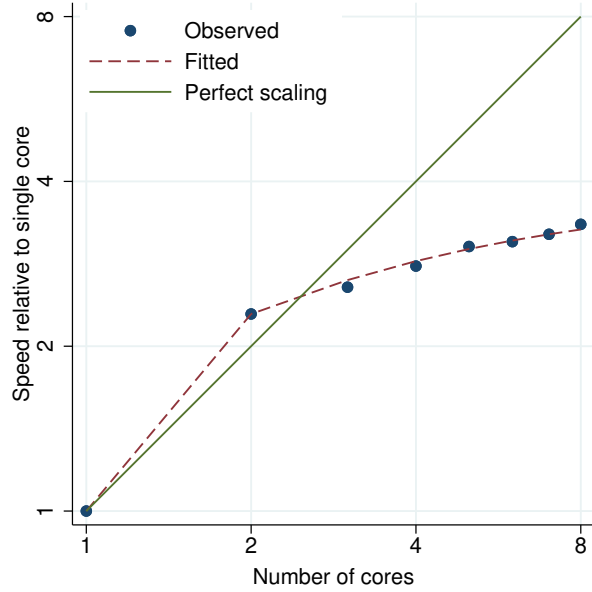


Figure 238. kwallis performance plot.

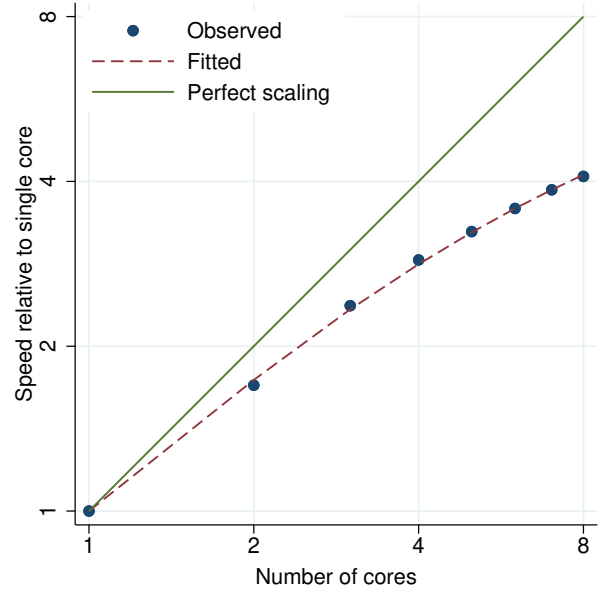


Figure 239. ladder performance plot.

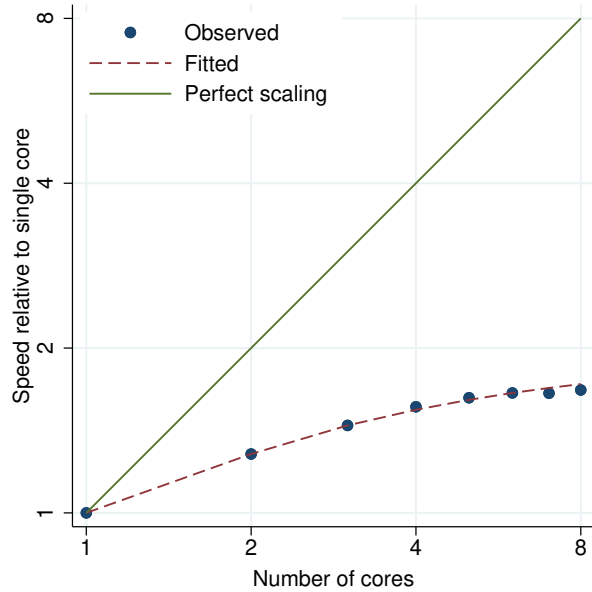


Figure 240. lasso linear performance plot.

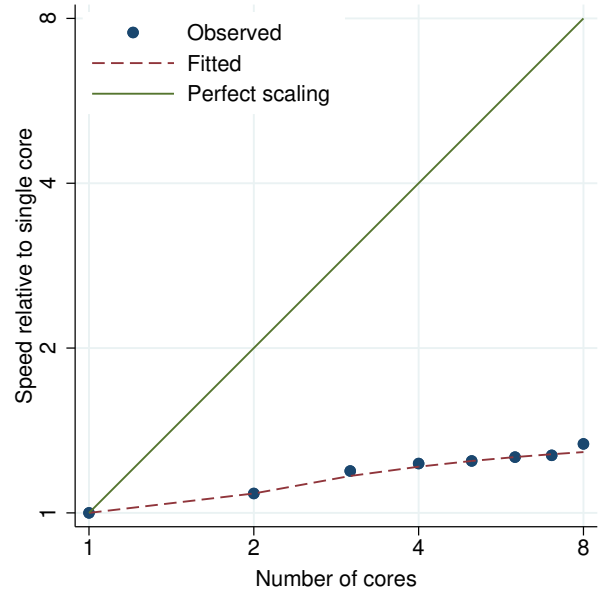


Figure 241. lasso logit performance plot.

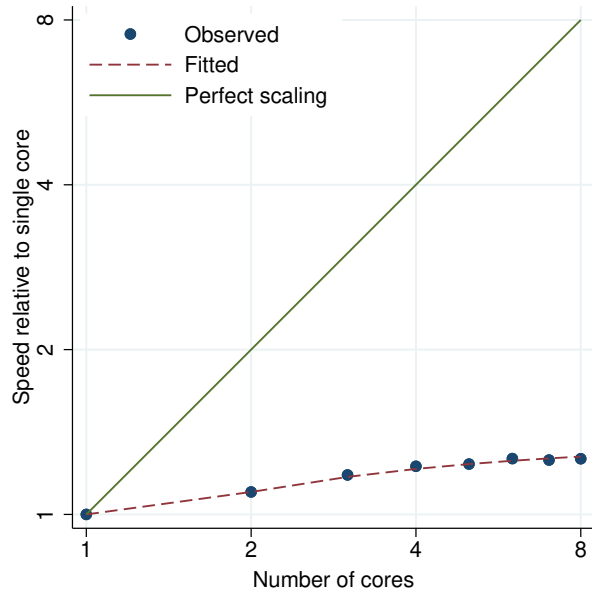


Figure 242. lasso poisson performance plot.

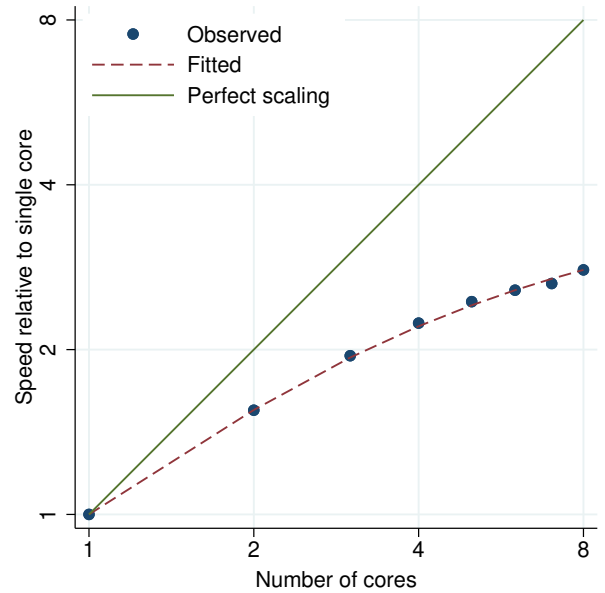


Figure 243. gsem, lclass(C 2) performance plot.

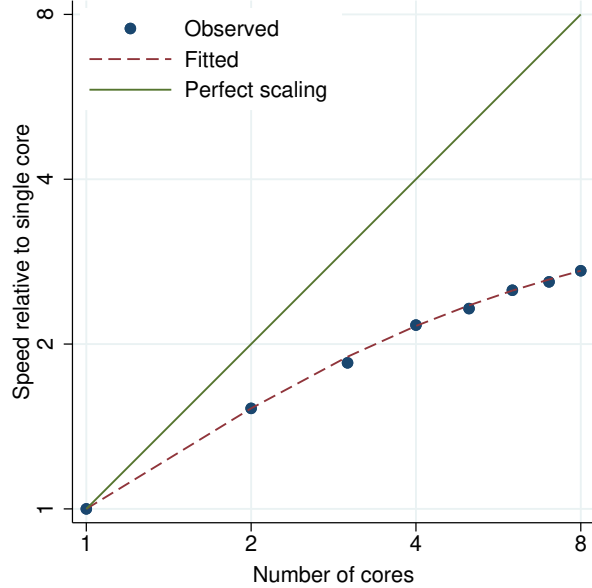


Figure 244. gsem, lclass(C 3) performance plot.

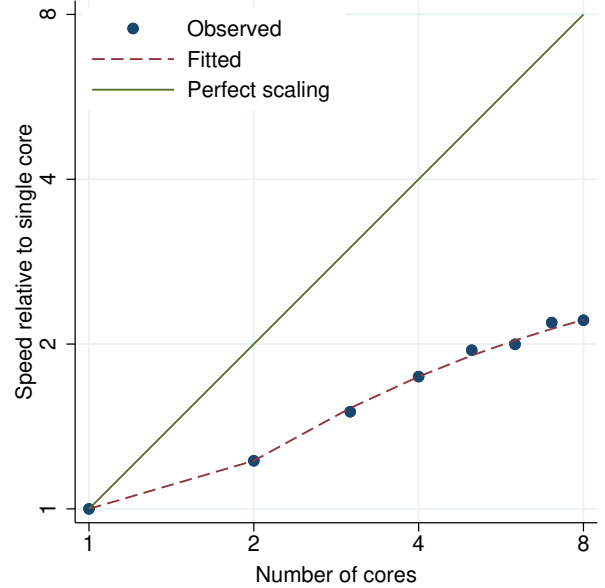


Figure 245. levelsof performance plot.

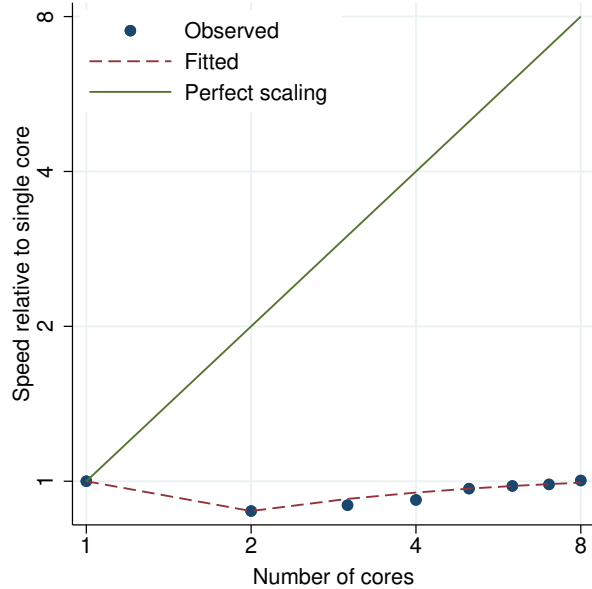


Figure 246. loadingplot performance plot.

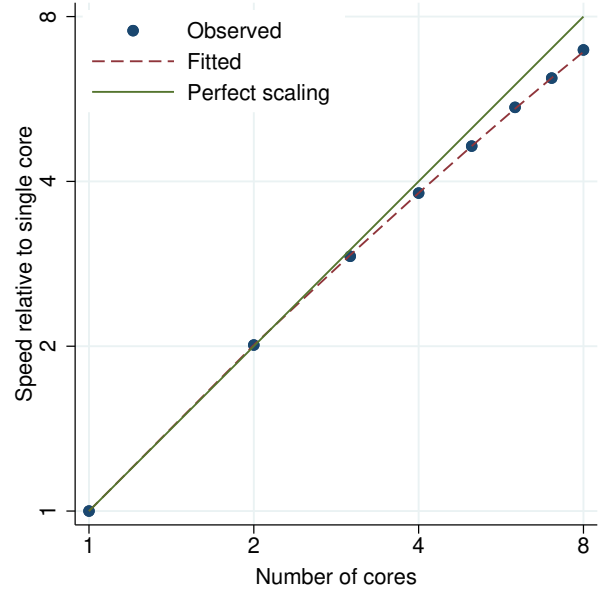


Figure 247. logistic performance plot.

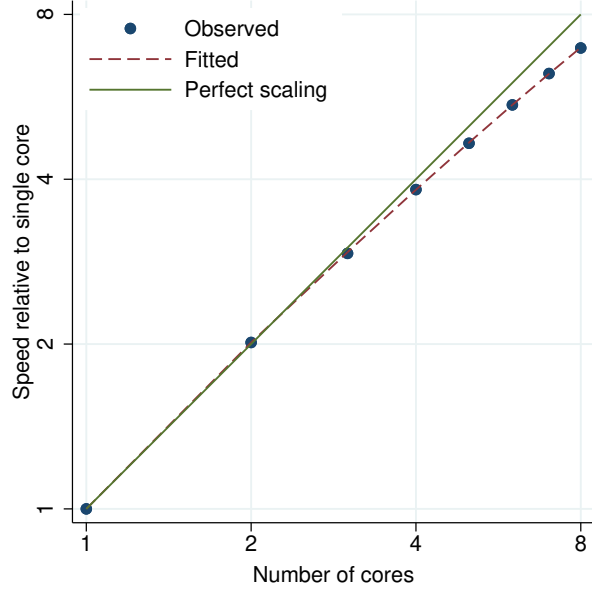


Figure 248. logit performance plot.

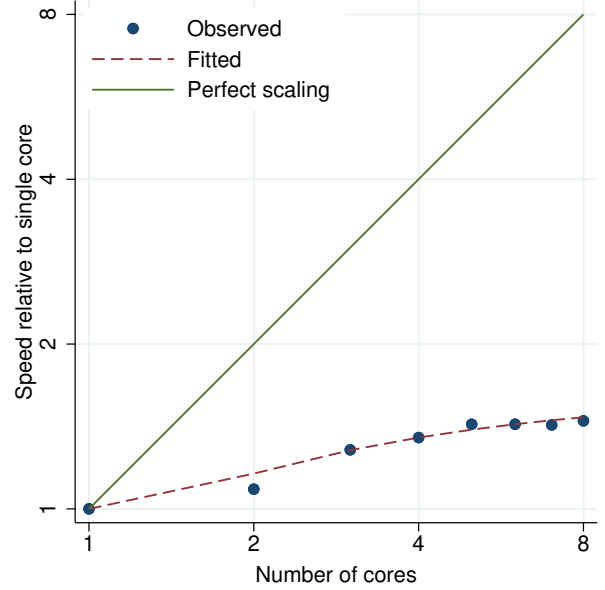


Figure 249. loneyway performance plot.

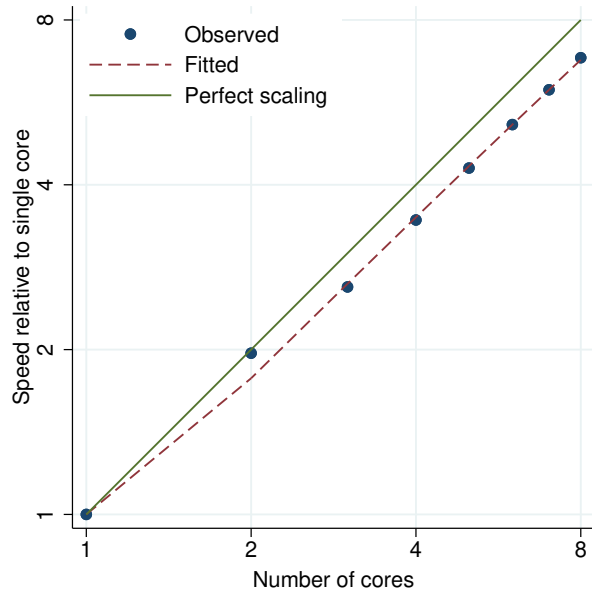


Figure 250. lowess performance plot.

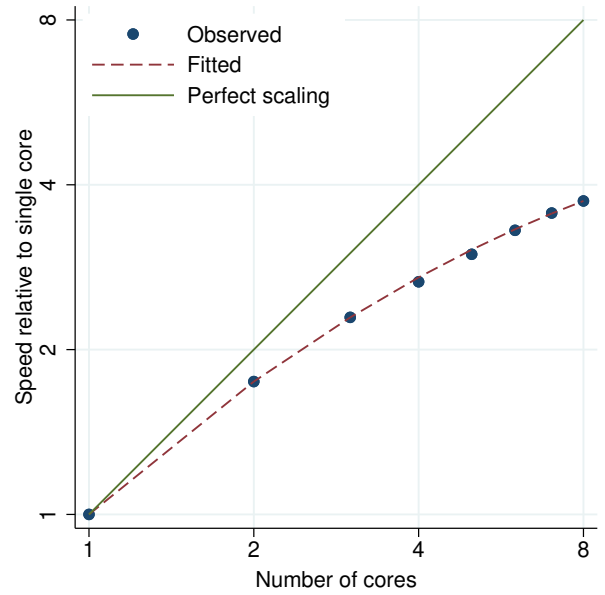


Figure 251. lpoly performance plot.

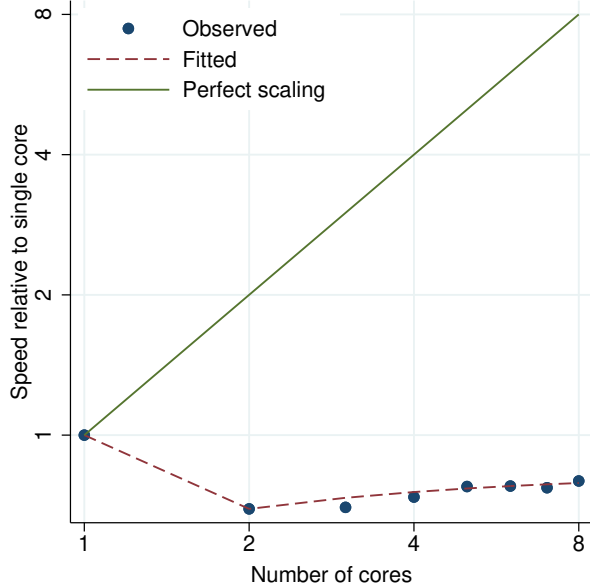


Figure 252. 1table performance plot.

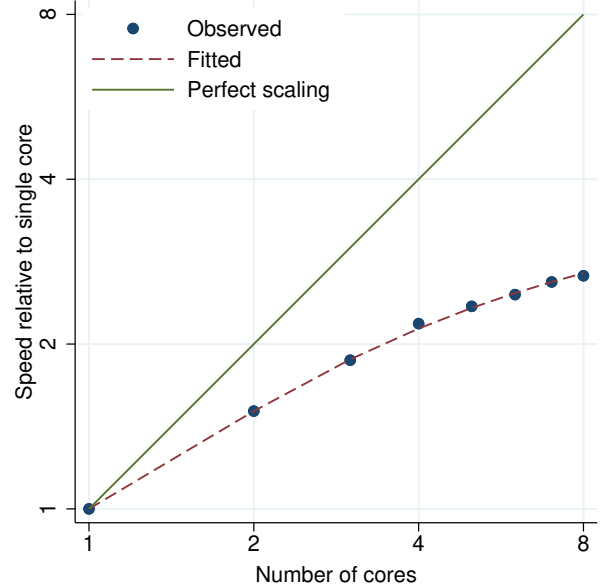


Figure 253. makespline bspline performance plot.

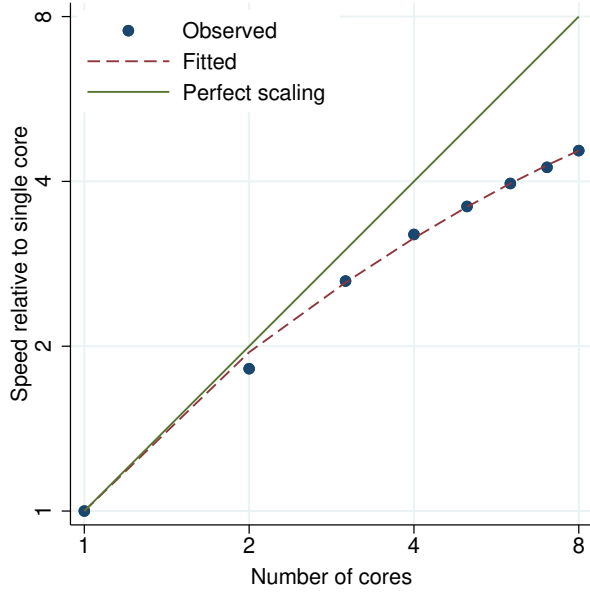


Figure 254. makespline linear performance plot.

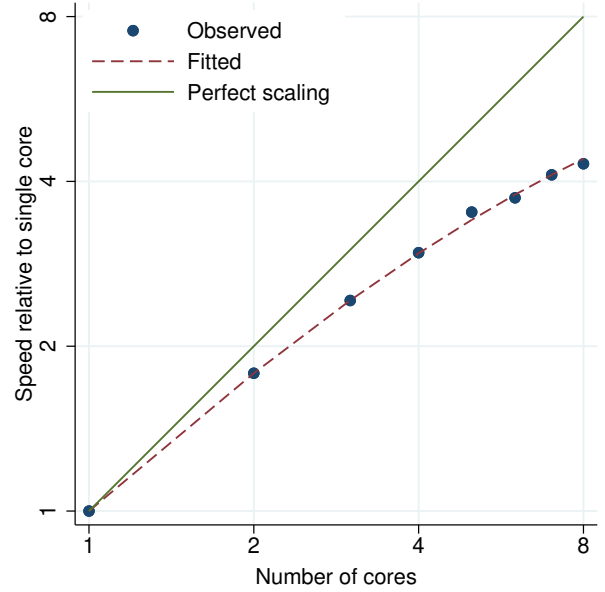


Figure 255. makespline piecewise performance plot.

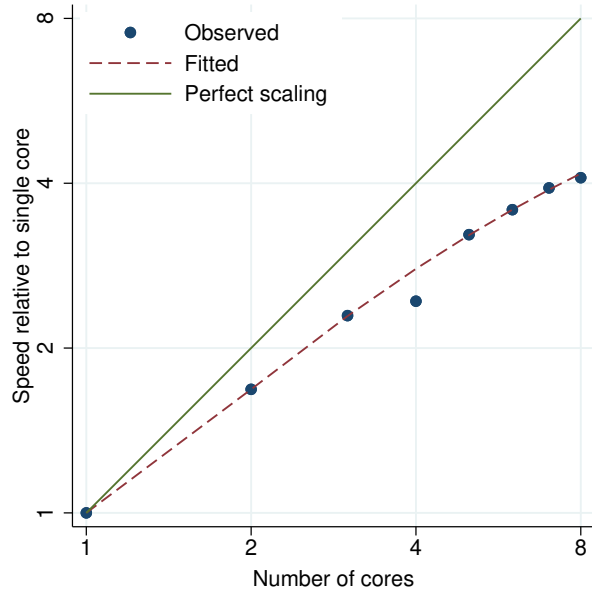


Figure 256. makespline rcs performance plot.

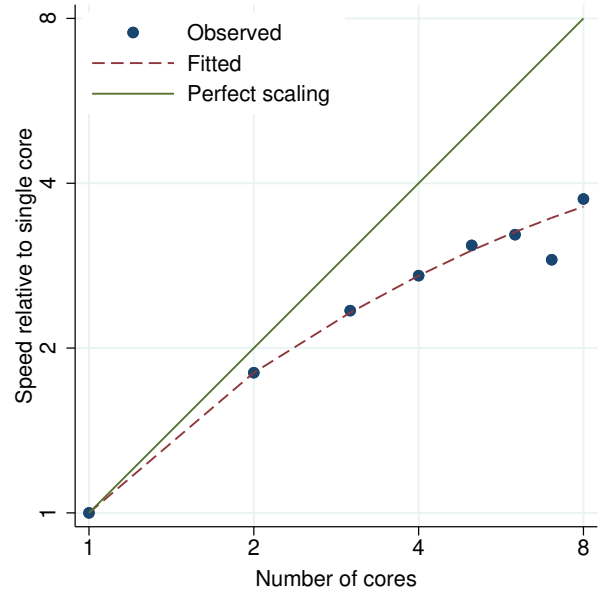


Figure 257. manova (one-way) performance plot.

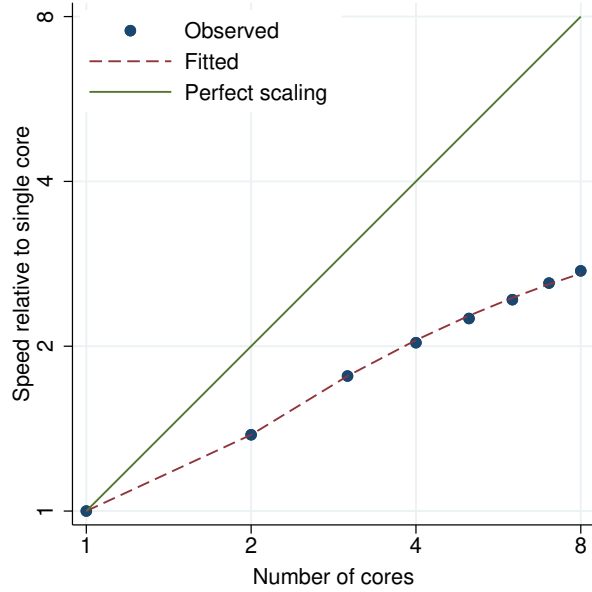


Figure 258. manova (two-way) performance plot.

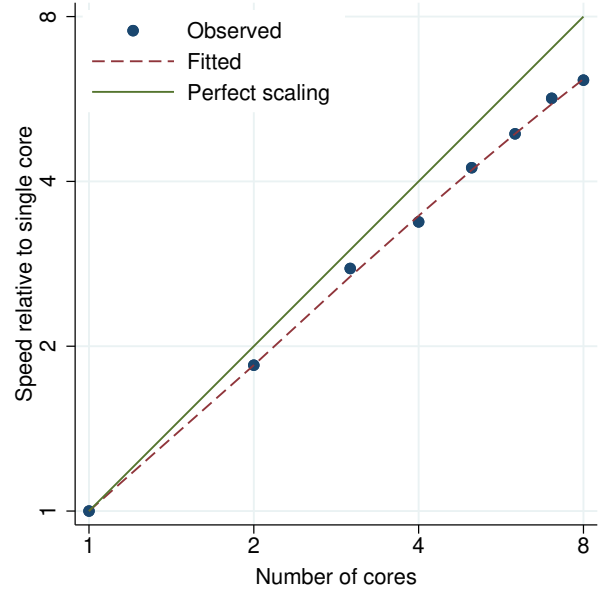


Figure 259. margins performance plot.

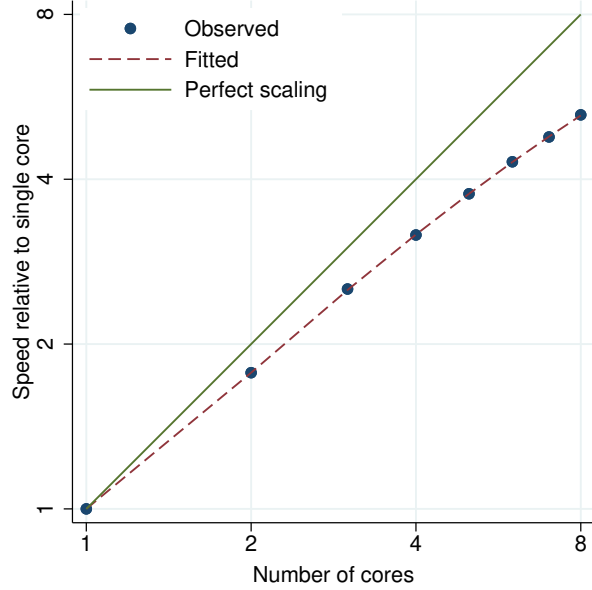


Figure 260. margins, dydx() exp() performance plot.

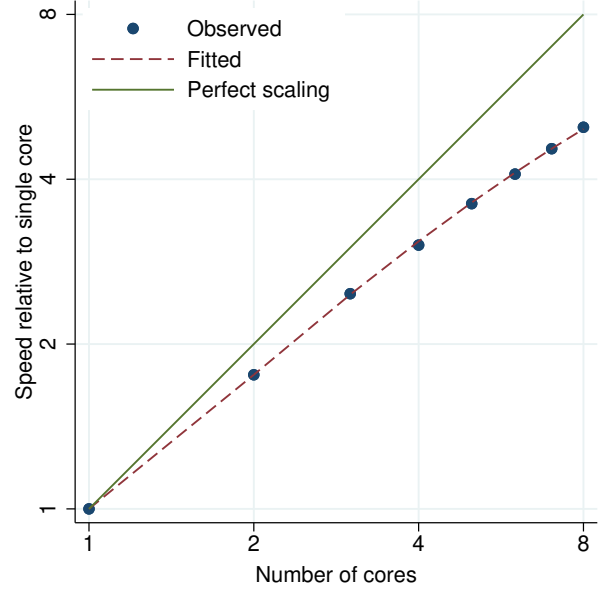


Figure 261. margins, dydx() performance plot.

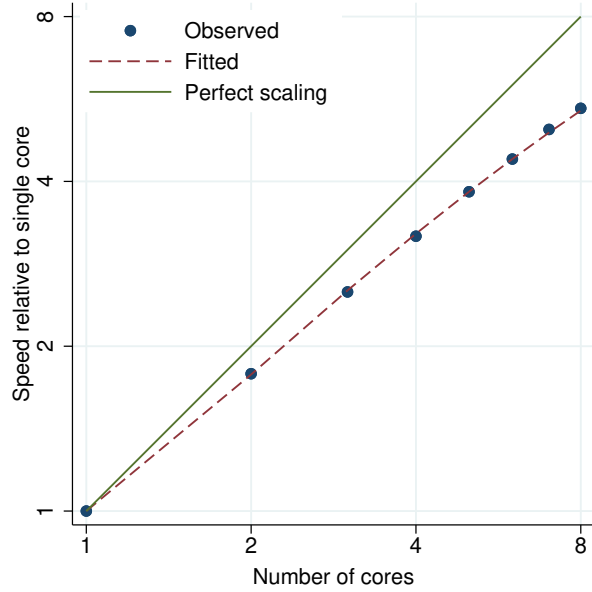


Figure 262. margins, exp() performance plot.

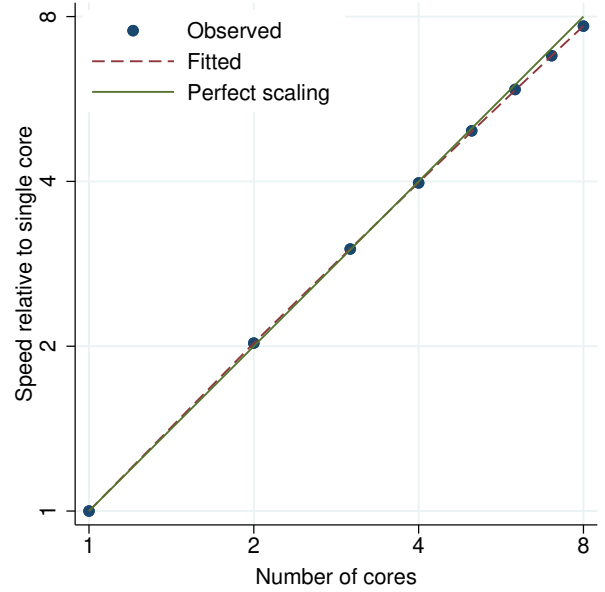


Figure 263. markout performance plot.

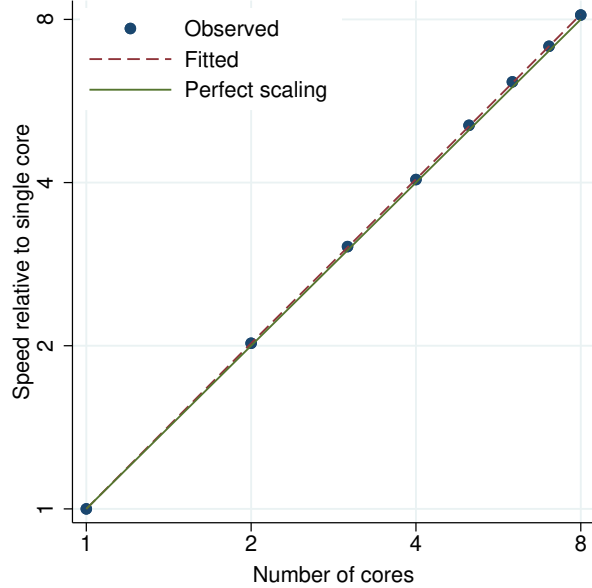


Figure 264. marksample performance plot.

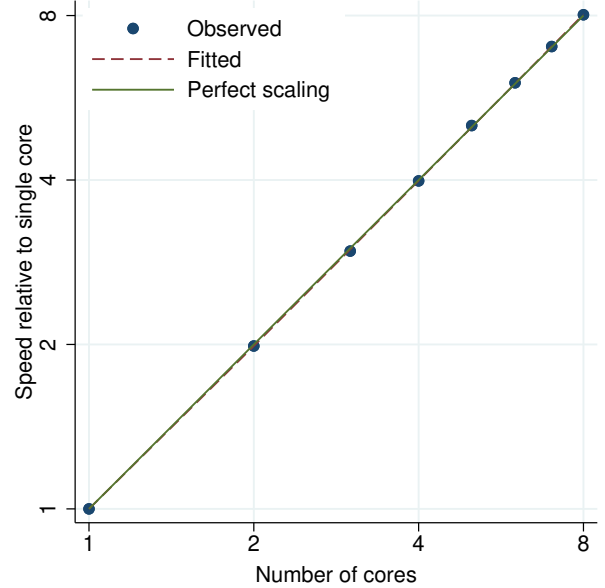


Figure 265. marksample if exp performance plot.

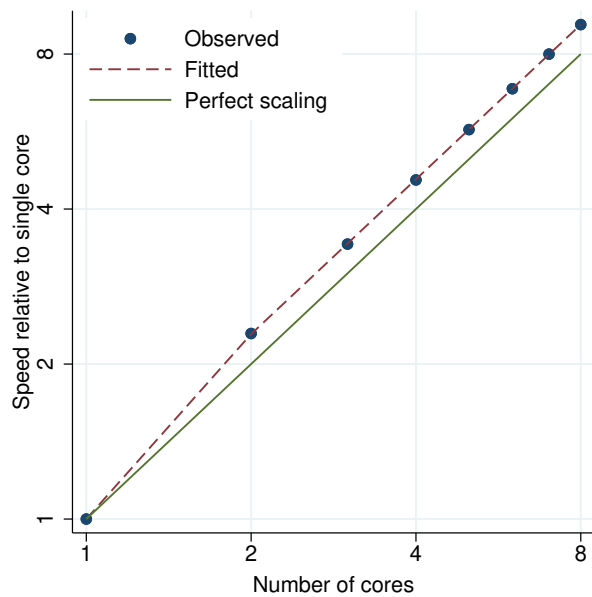


Figure 266. matrix accum performance plot.

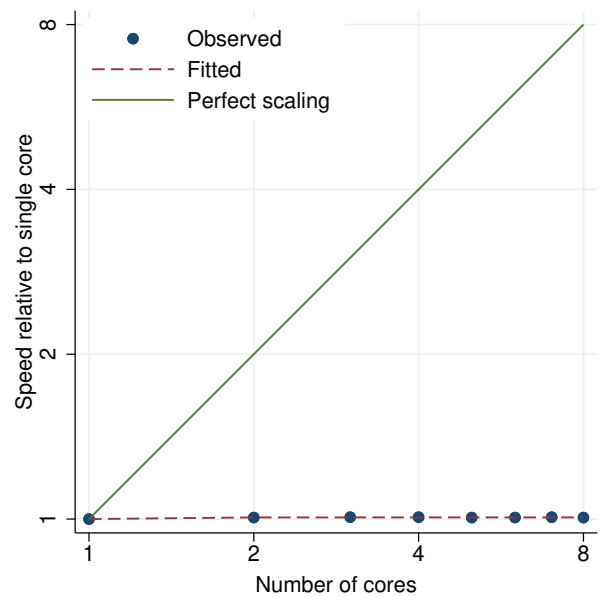


Figure 267. matrix eigenvalues performance plot.

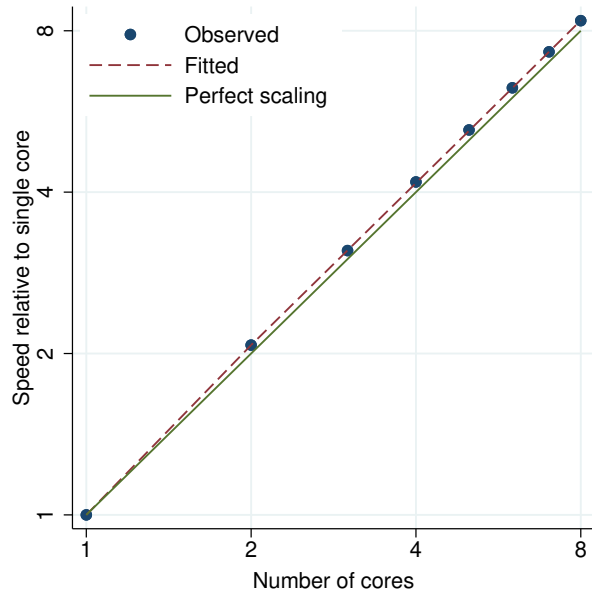


Figure 268. matrix score performance plot.

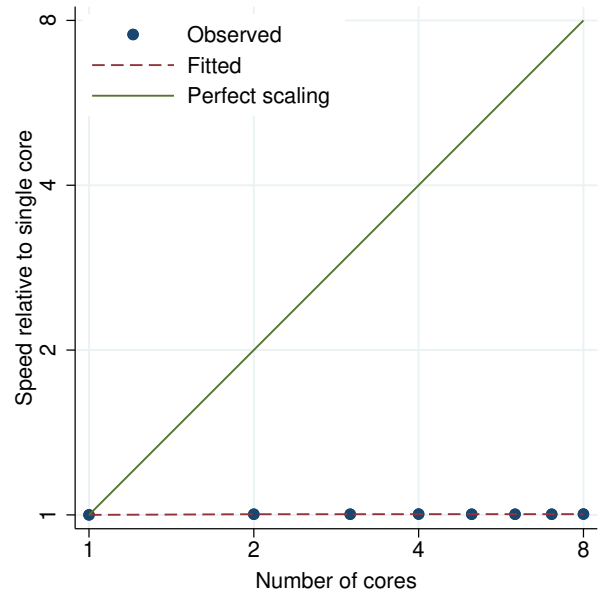


Figure 269. matrix svd performance plot.

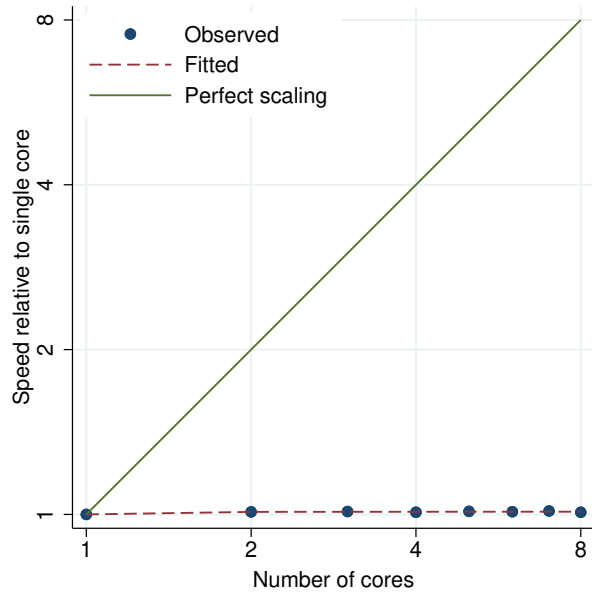


Figure 270. matrix symeigen performance plot.

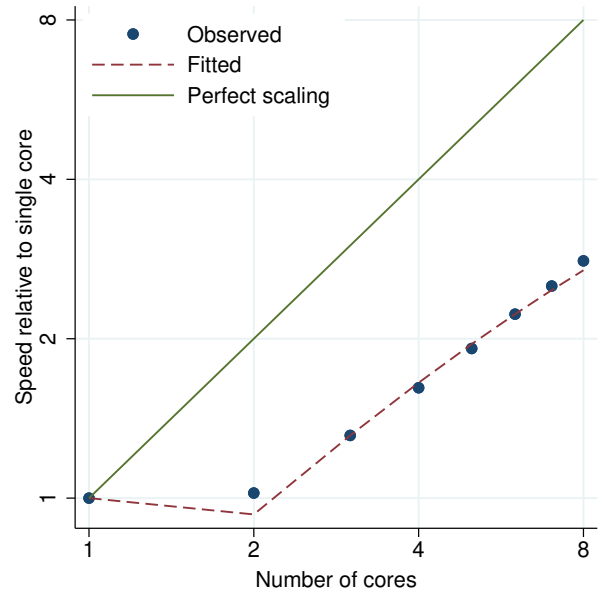


Figure 271. matrix syminv performance plot.

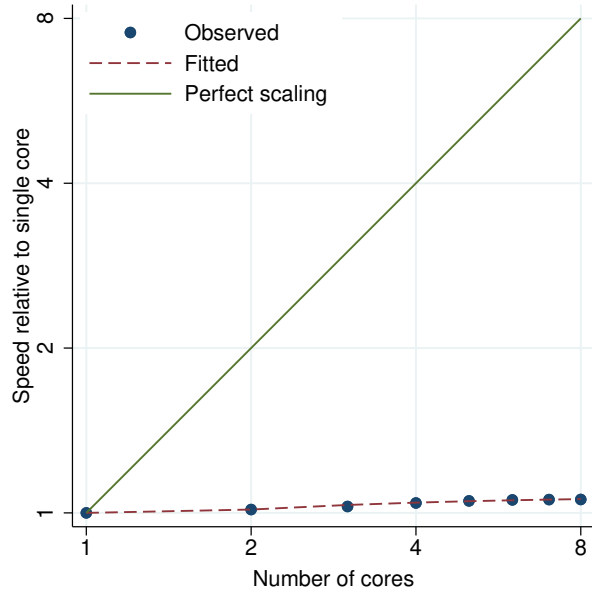


Figure 272. mca performance plot.

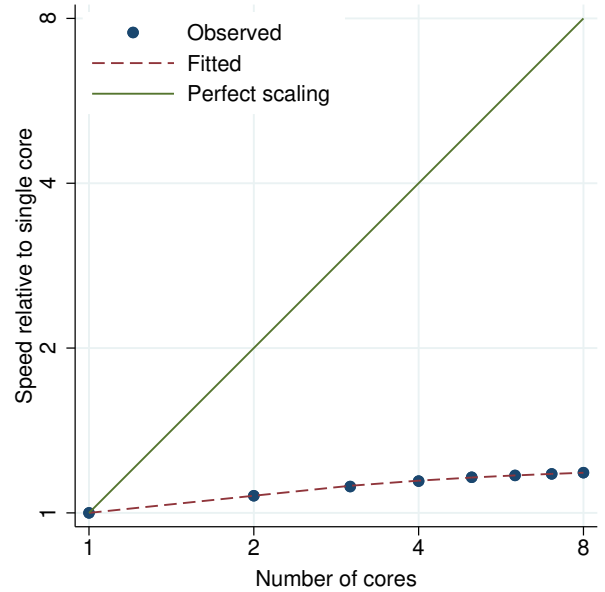


Figure 273. mcc performance plot.

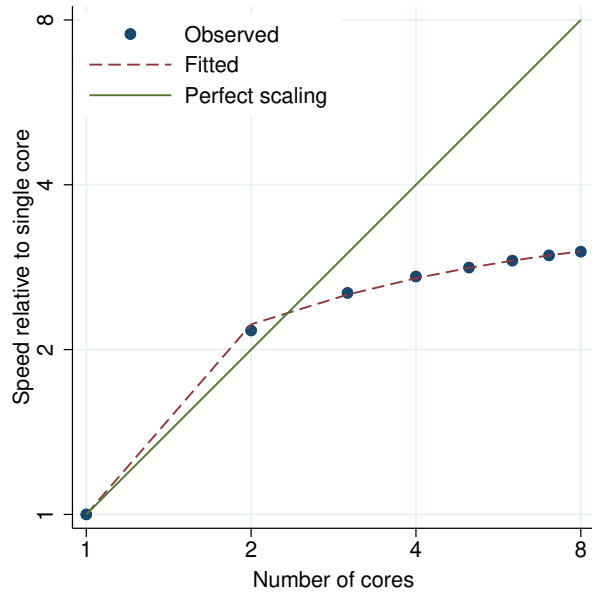


Figure 274. mds performance plot.

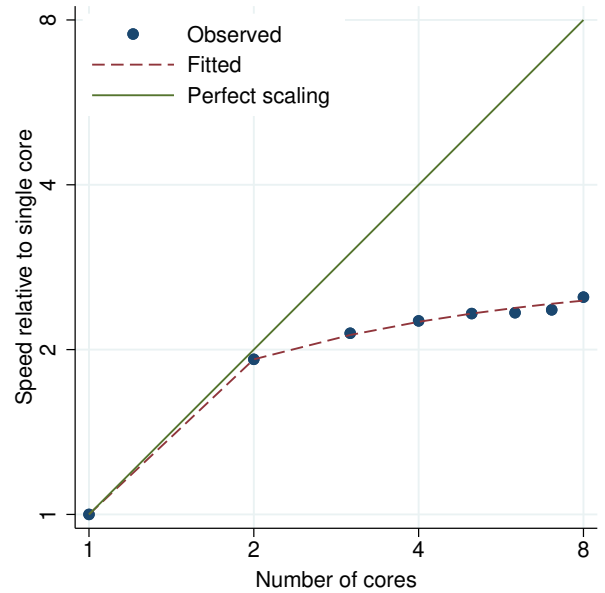


Figure 275. mdslong performance plot.

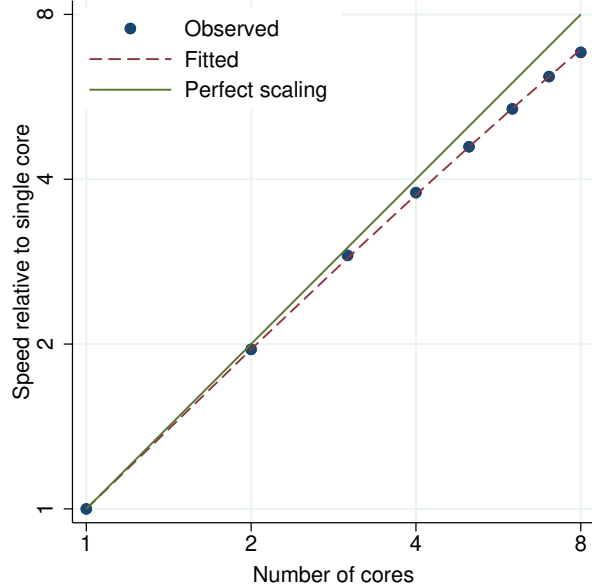


Figure 276. mean performance plot.

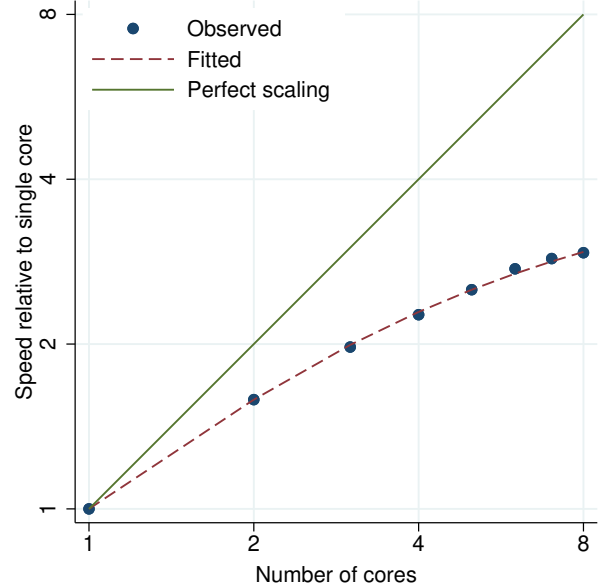


Figure 277. mecloglog performance plot.

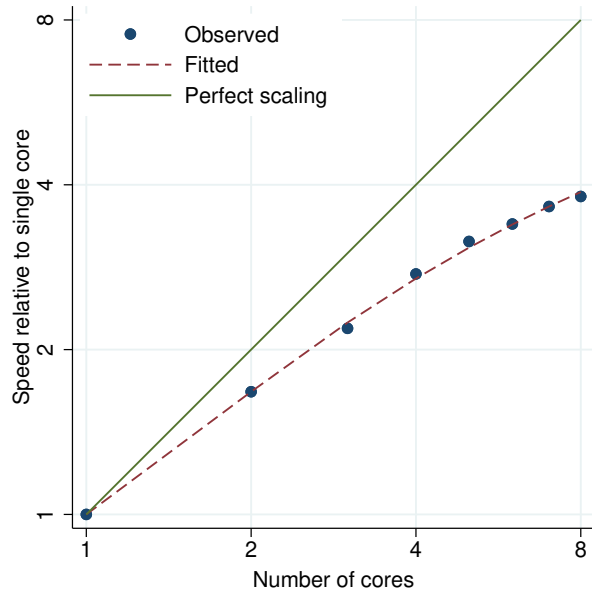


Figure 278. median performance plot.

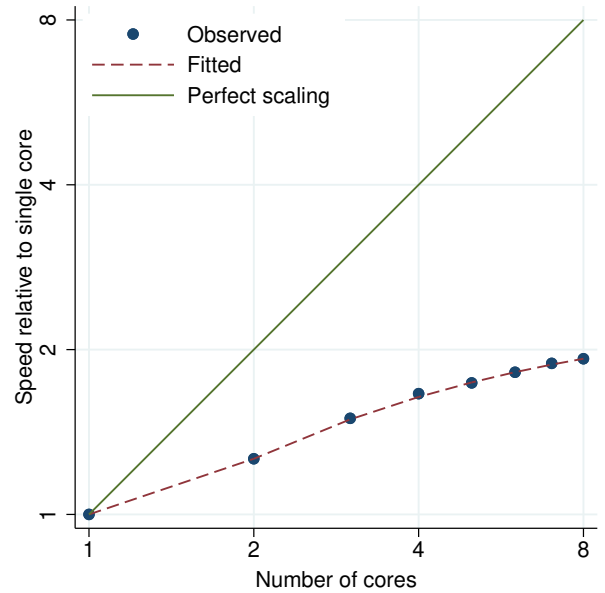


Figure 279. mediate (, expmean) (, linear) performance plot.

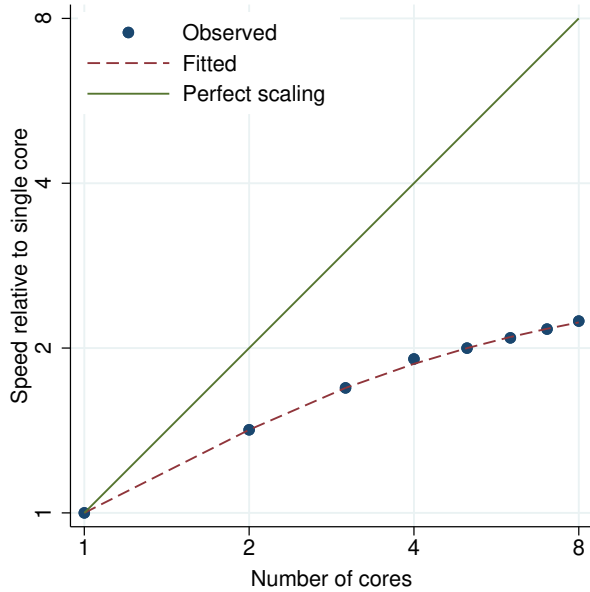


Figure 280. `mediate (, linear) (, linear)` performance plot.

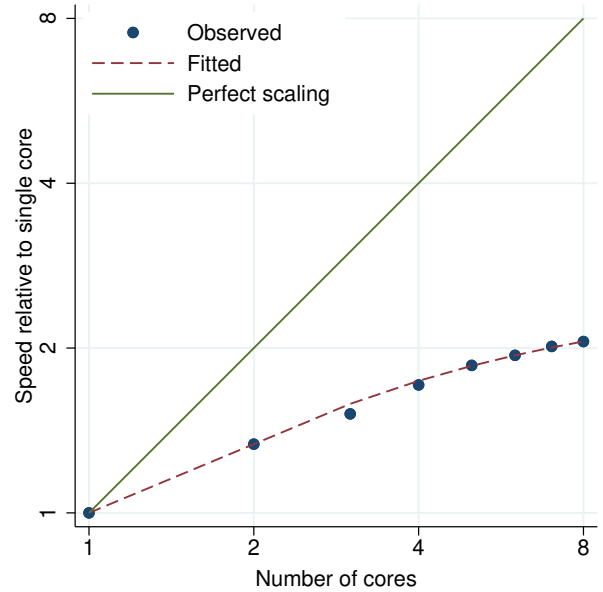


Figure 281. `mediate (, logit) (, linear)` performance plot.

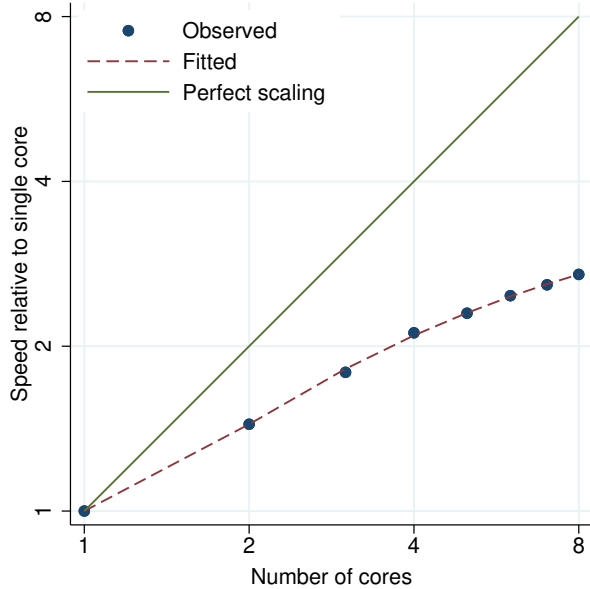


Figure 282. `mediate (, poisson) (, linear)` performance plot.

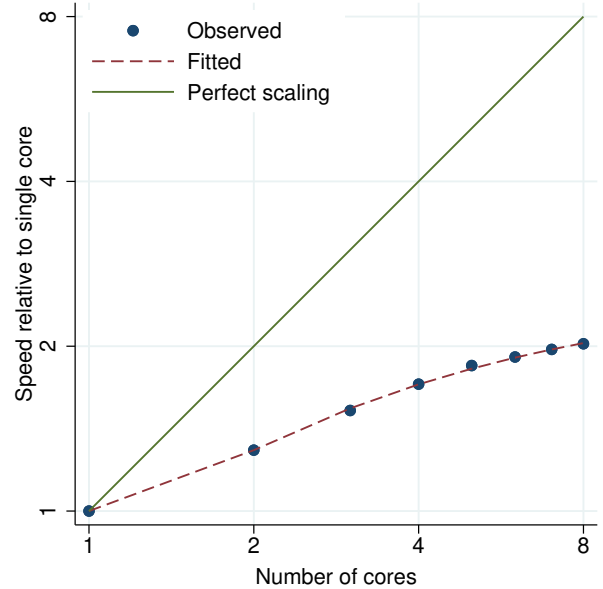


Figure 283. `mediate (, probit) (, linear)` performance plot.

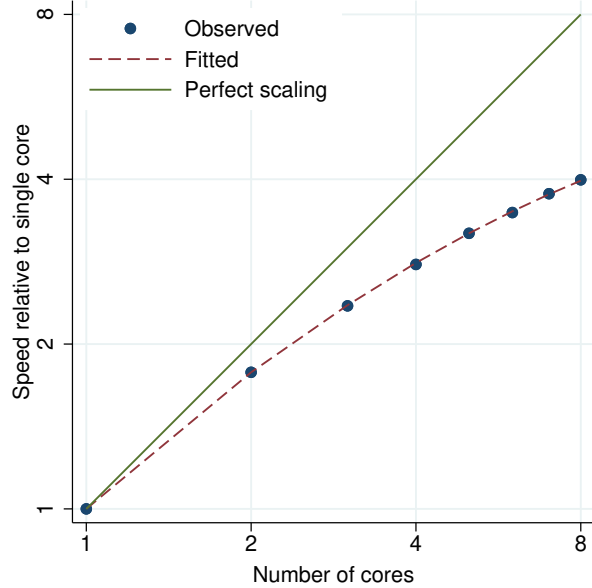


Figure 284. meintreg performance plot.

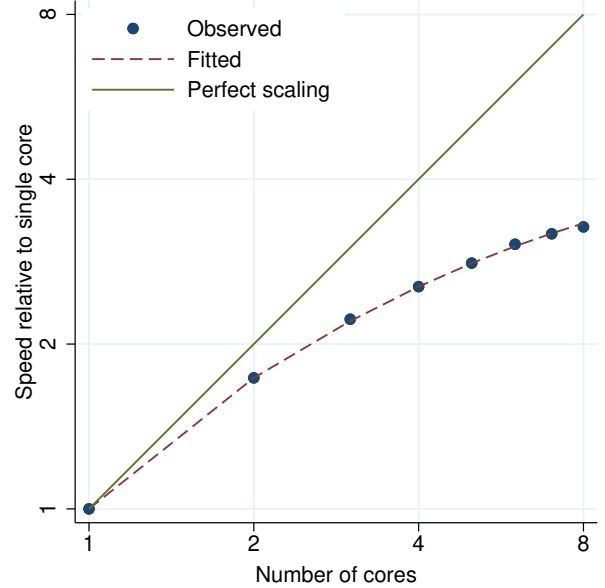


Figure 285. melogit performance plot.

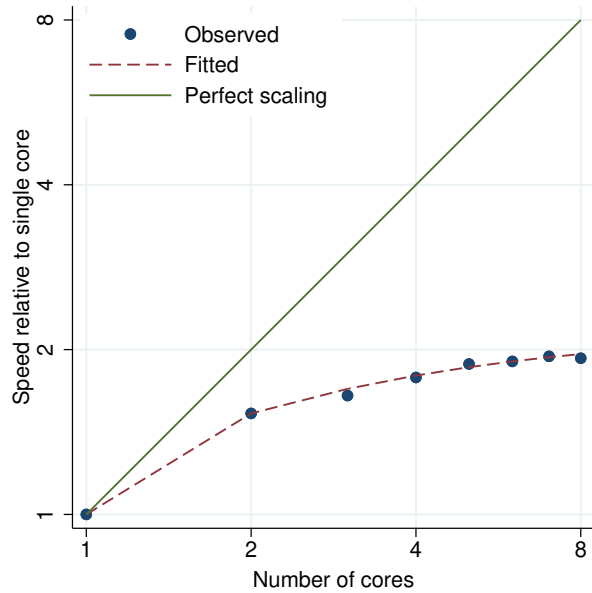


Figure 286. menbreg, dispersion(constant) performance plot.

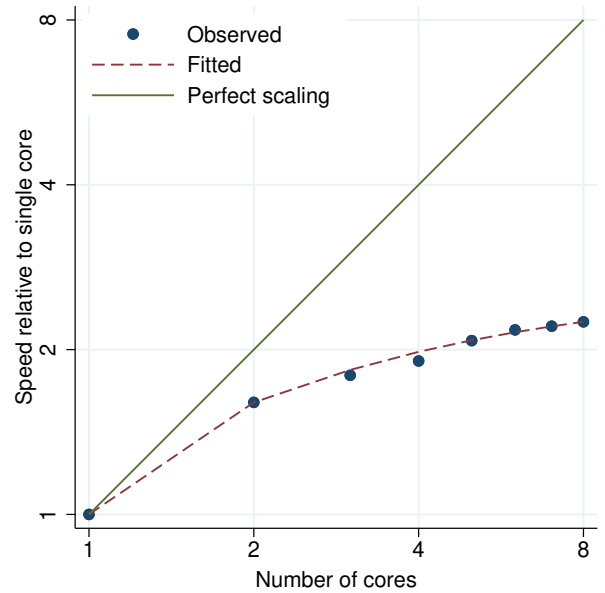


Figure 287. menbreg, dispersion(mean) performance plot.

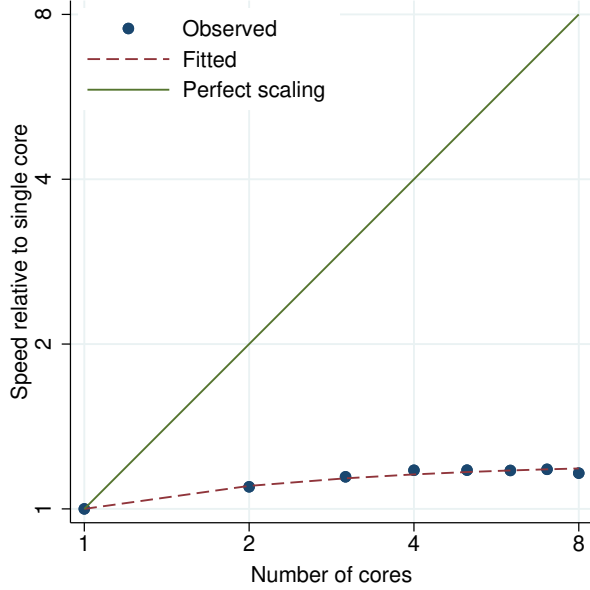


Figure 288. men1 performance plot.

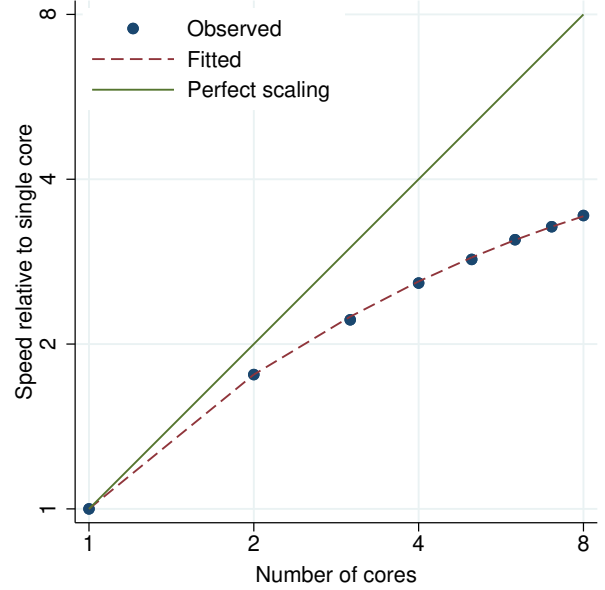


Figure 289. meologit performance plot.

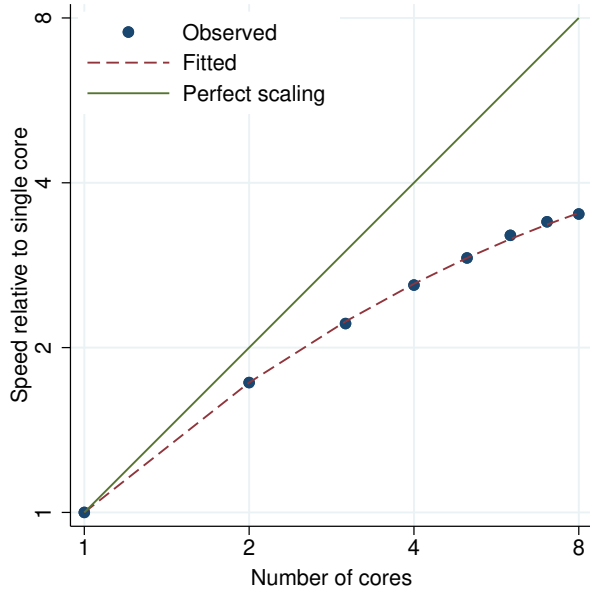


Figure 290. meoprobit performance plot.

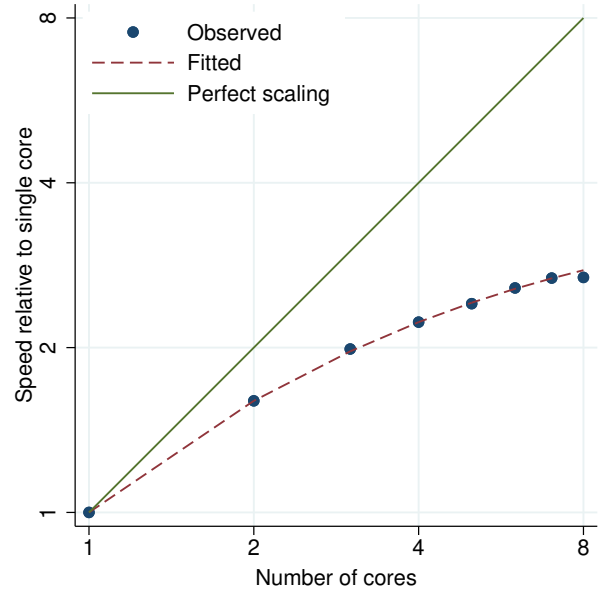


Figure 291. mepoisson performance plot.

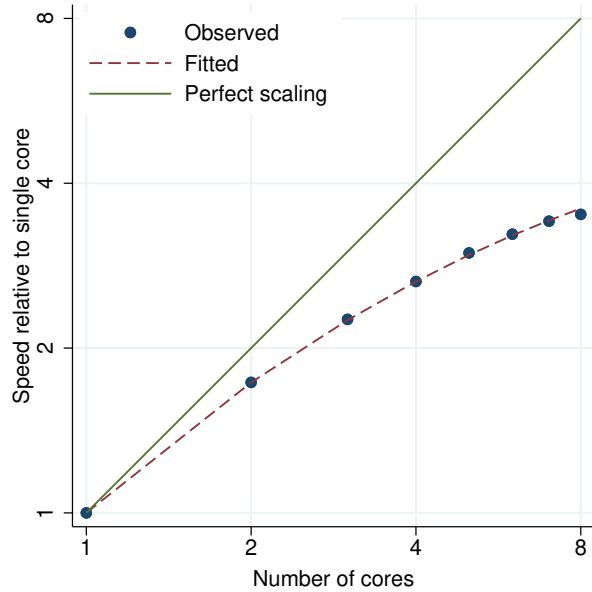


Figure 292. meprobit performance plot.

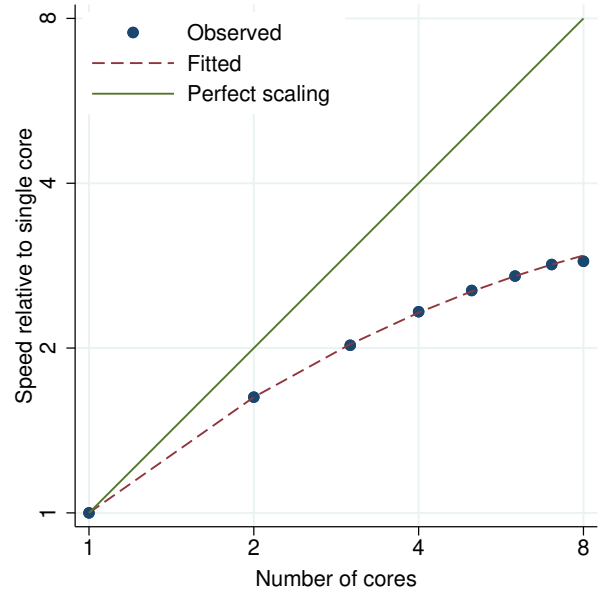


Figure 293. mestreg, distribution(exp) performance plot.

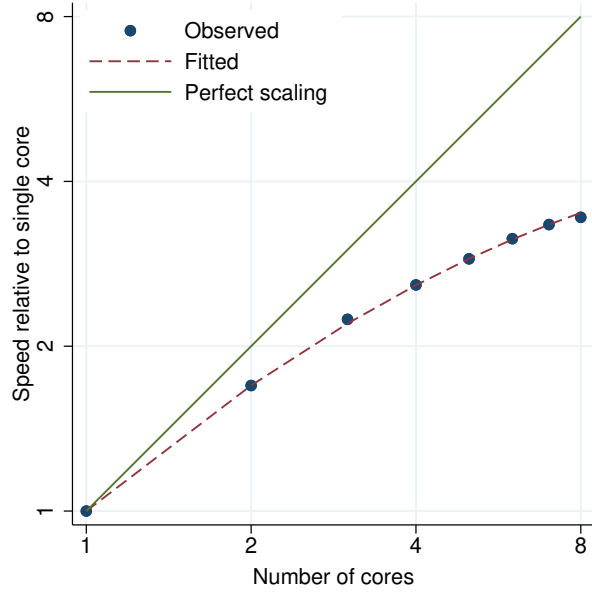


Figure 294. mestreg, distribution(weibull) performance plot.

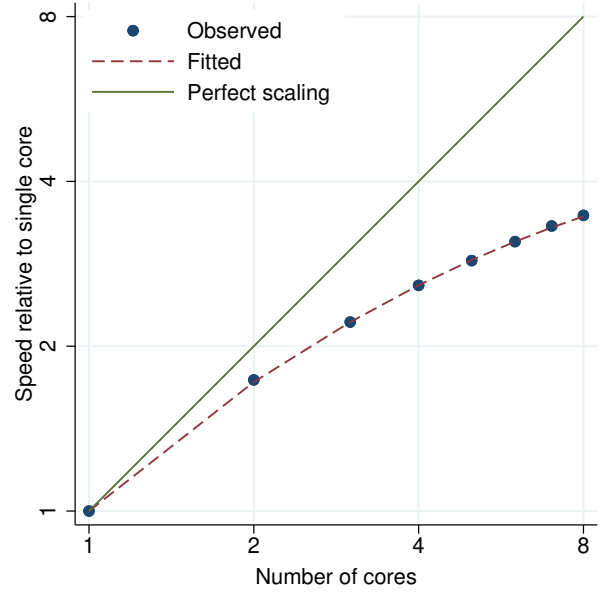


Figure 295. metobit performance plot.

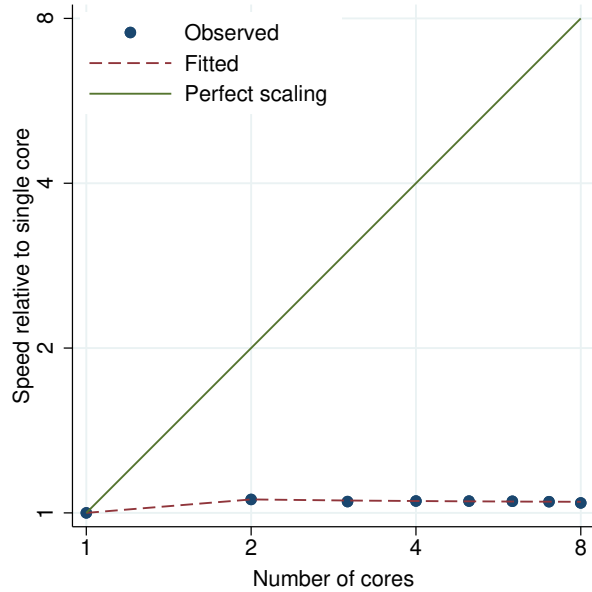


Figure 296. mgarch performance plot.

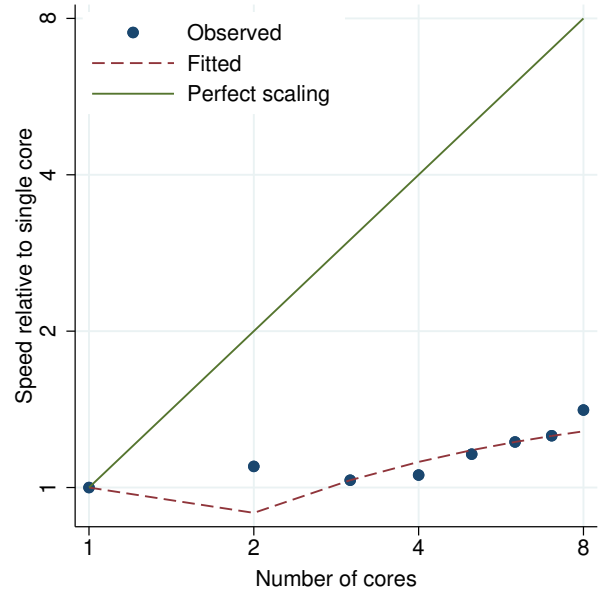


Figure 297. mhodds performance plot.

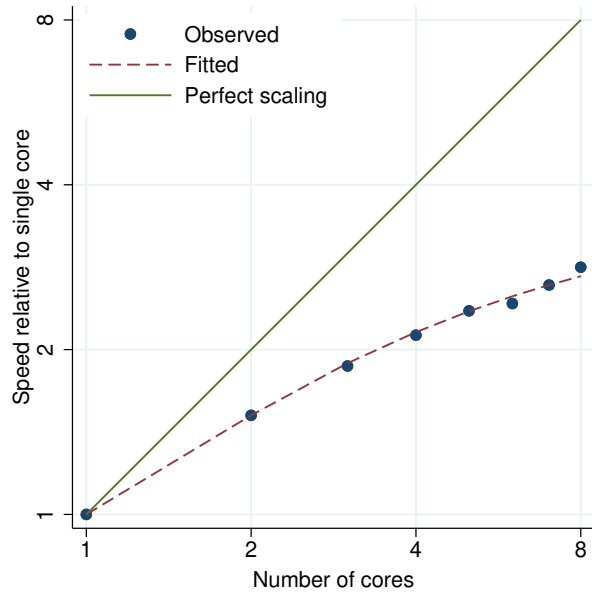


Figure 298. mhodds (adjusted) performance plot.

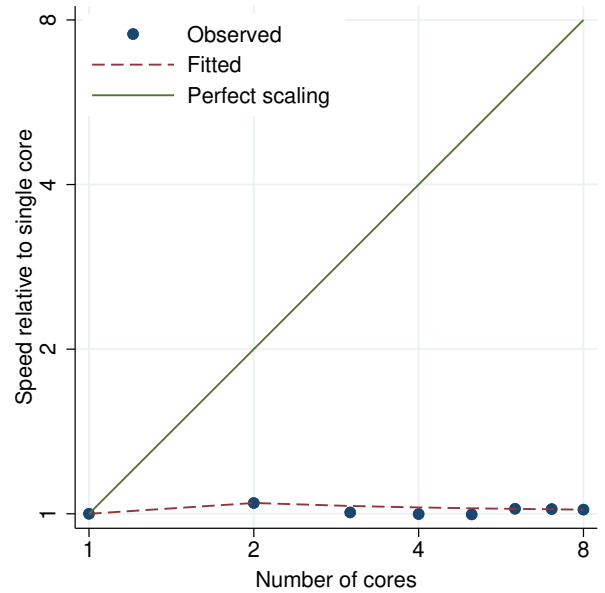


Figure 299. by: mhodds performance plot.

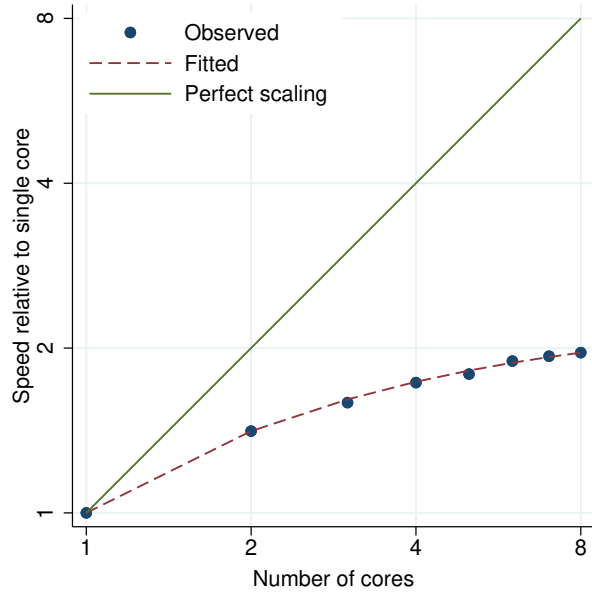


Figure 300. mhdods (trend) performance plot.

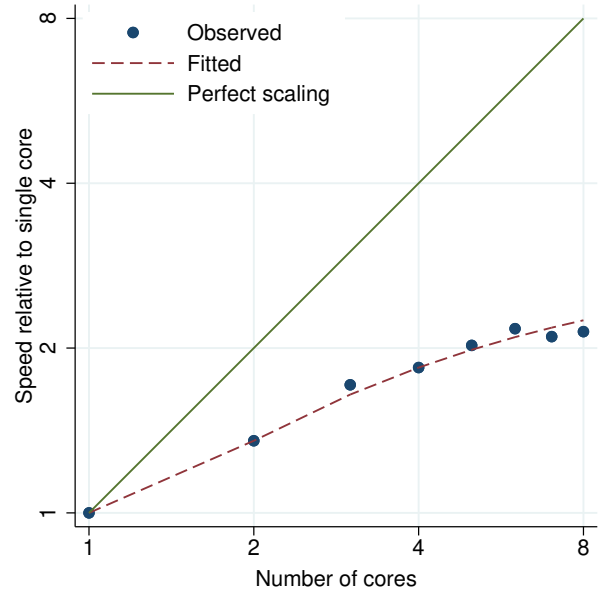


Figure 301. mi estimate: logit (flong) performance plot.

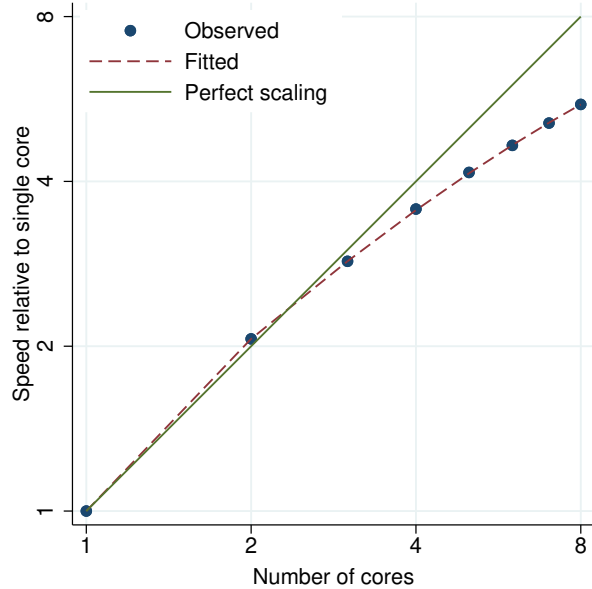


Figure 302. mi estimate: logit (flongsep) performance plot.

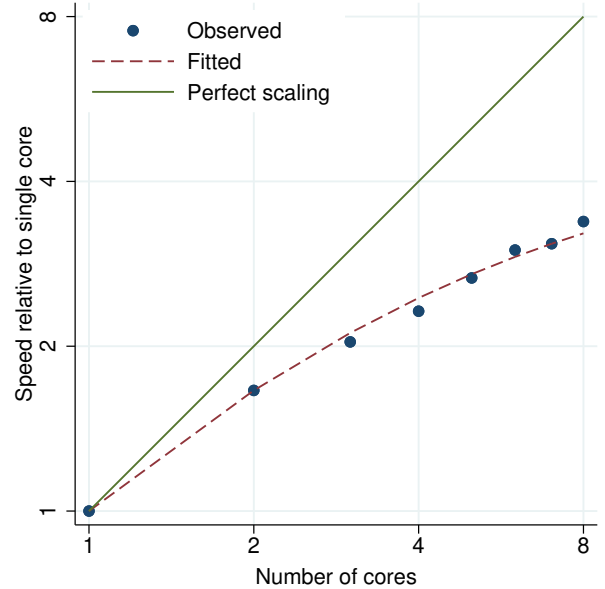


Figure 303. mi estimate: logit (mlong) performance plot.

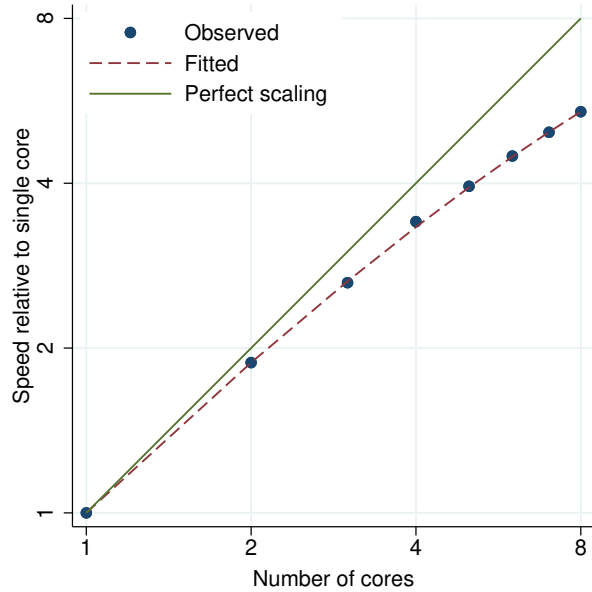


Figure 304. mi estimate: logit (wide) performance plot.

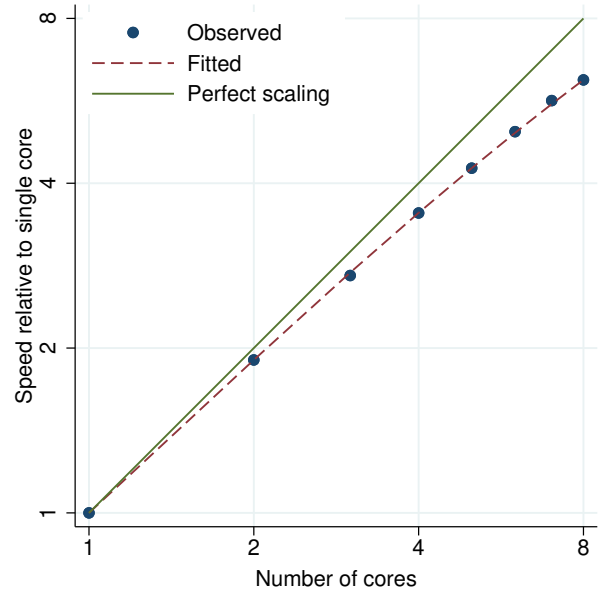


Figure 305. mi estimate: mlogit performance plot.

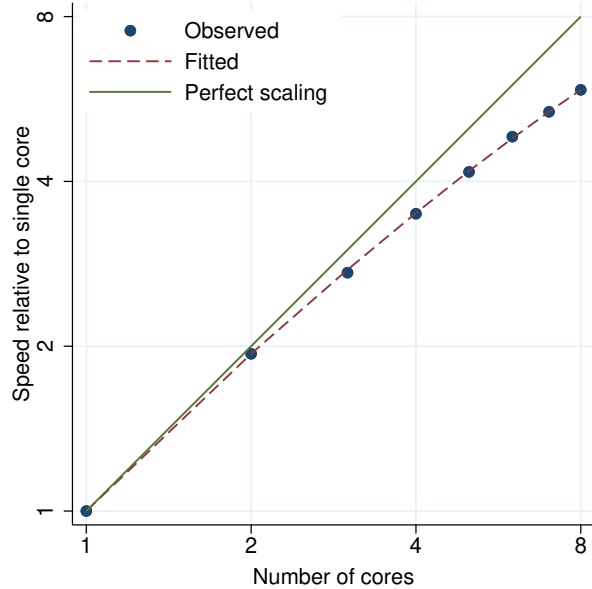


Figure 306. mi estimate: ologit performance plot.

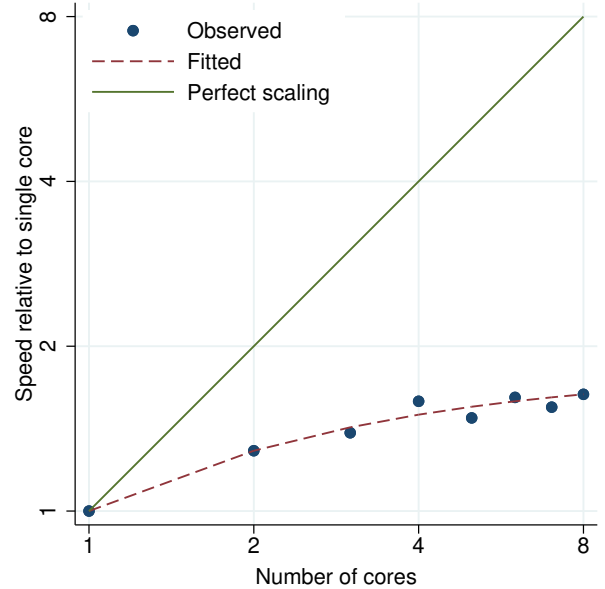


Figure 307. mi estimate: regress (flong) performance plot.

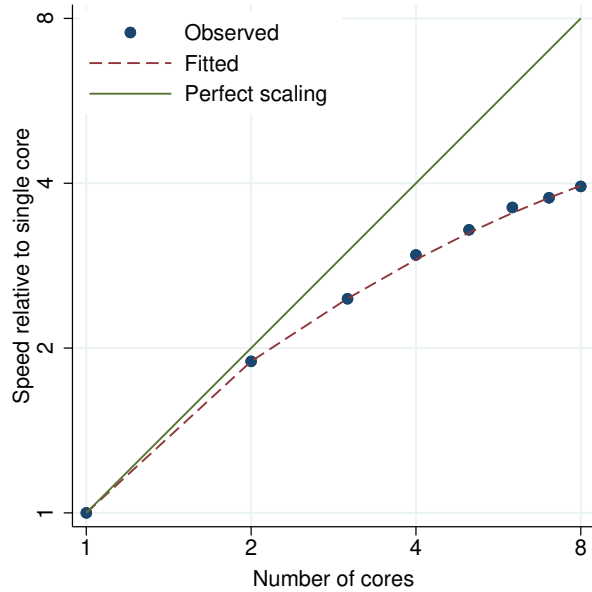


Figure 308. mi estimate: regress (flongsep) performance plot.

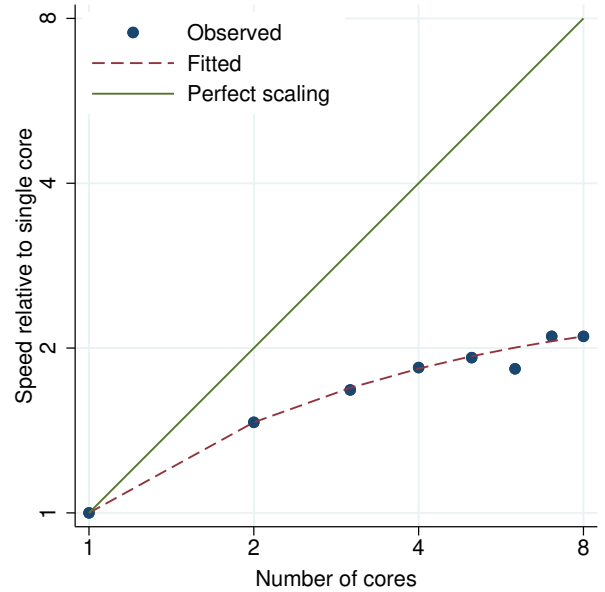


Figure 309. mi estimate: regress (mlong) performance plot.

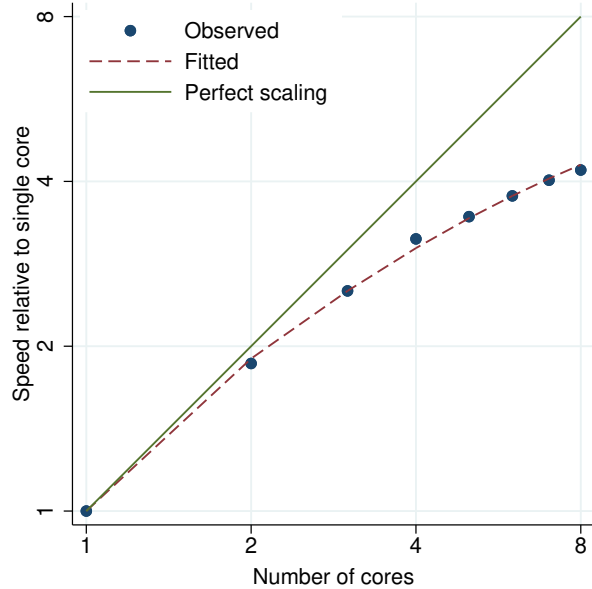


Figure 310. mi estimate: regress (wide) performance plot.

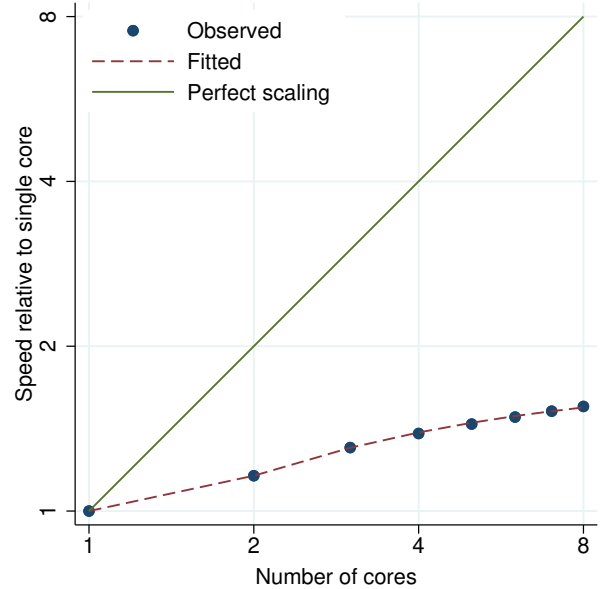


Figure 311. mi impute chained (flong) performance plot.

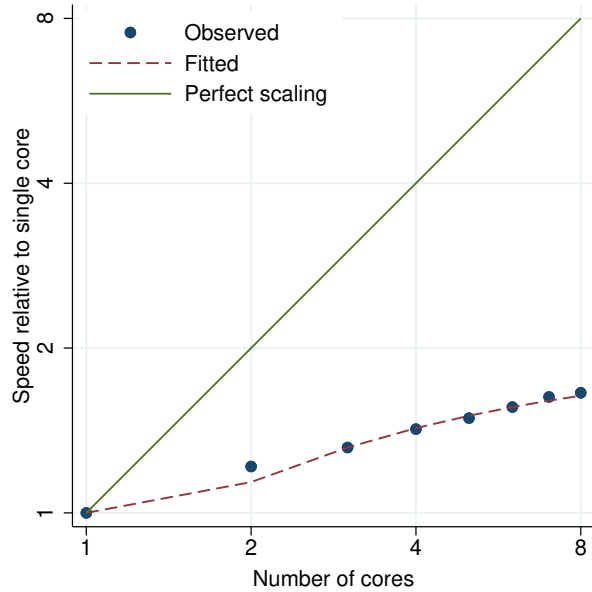


Figure 312. mi impute chained (flongsep) performance plot.

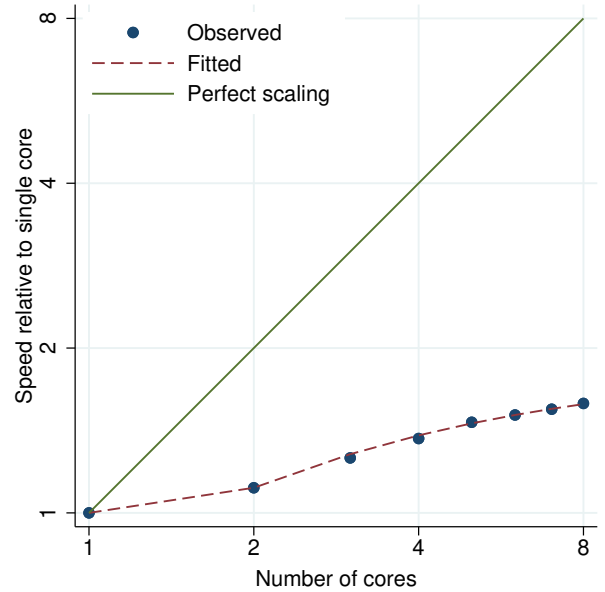


Figure 313. mi impute chained (mlong) performance plot.

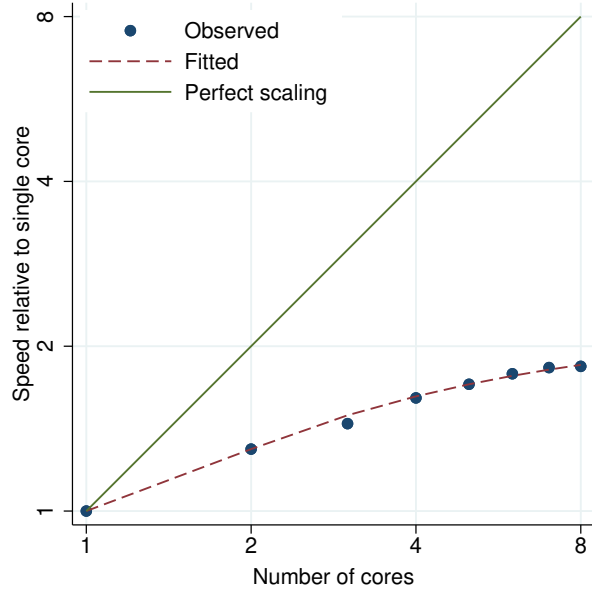


Figure 314. mi impute chained (wide) performance plot.

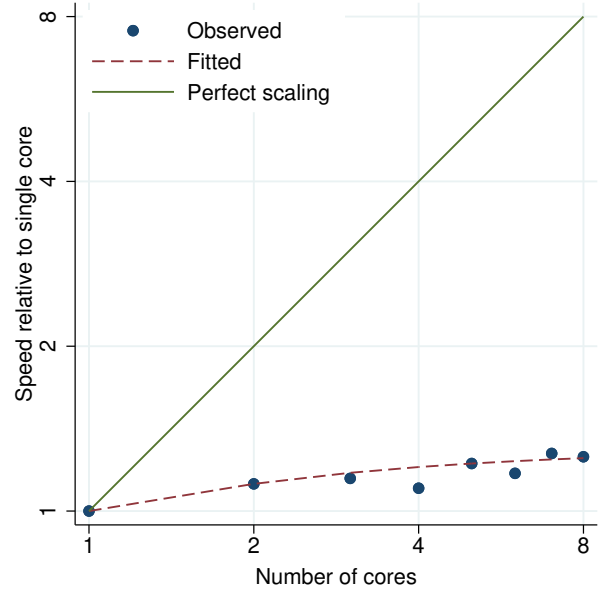


Figure 315. mi impute logit (flong) performance plot.

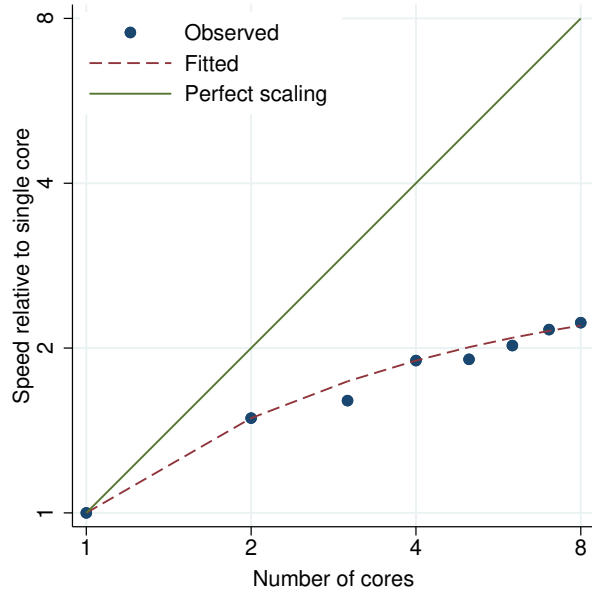


Figure 316. mi impute logit (flongsep) performance plot.

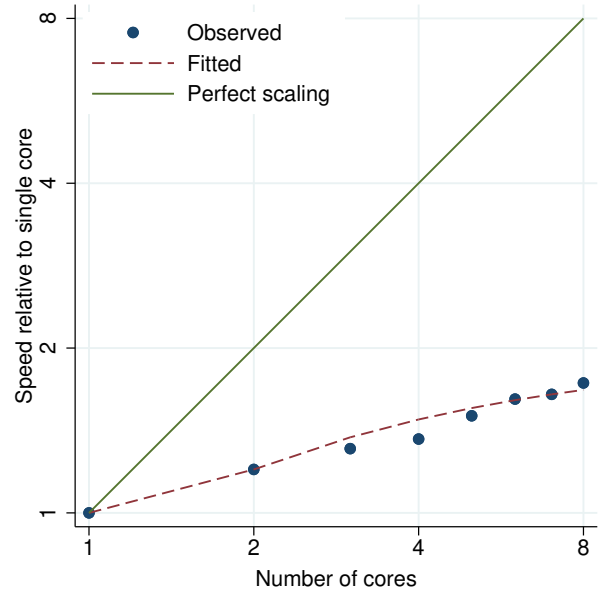


Figure 317. mi impute logit (mlong) performance plot.

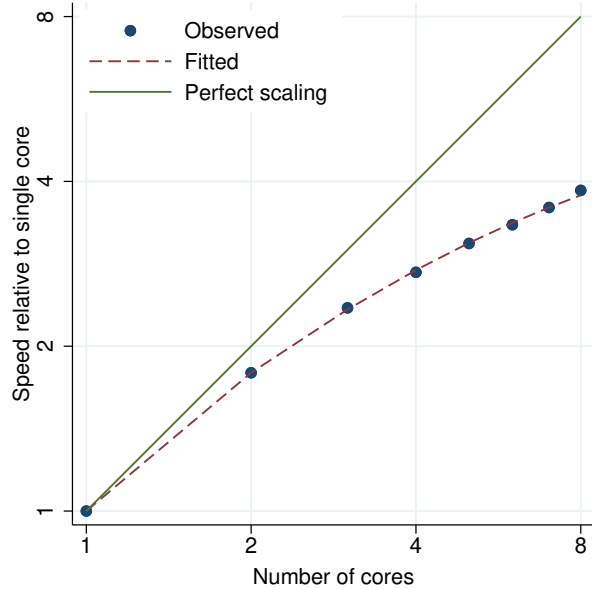


Figure 318. mi impute logit (wide) performance plot.

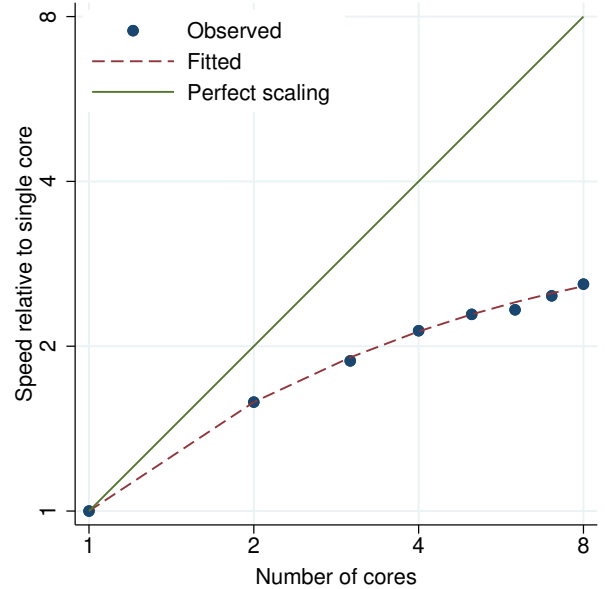


Figure 319. mi impute mlogit performance plot.

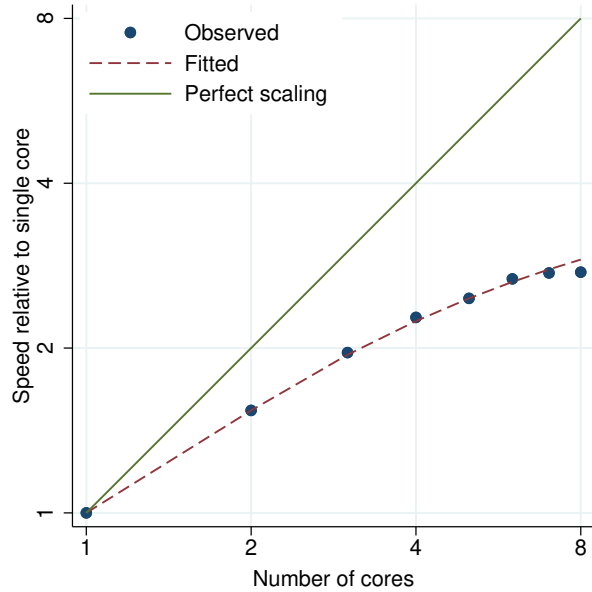


Figure 320. mi impute mono pmm performance plot.

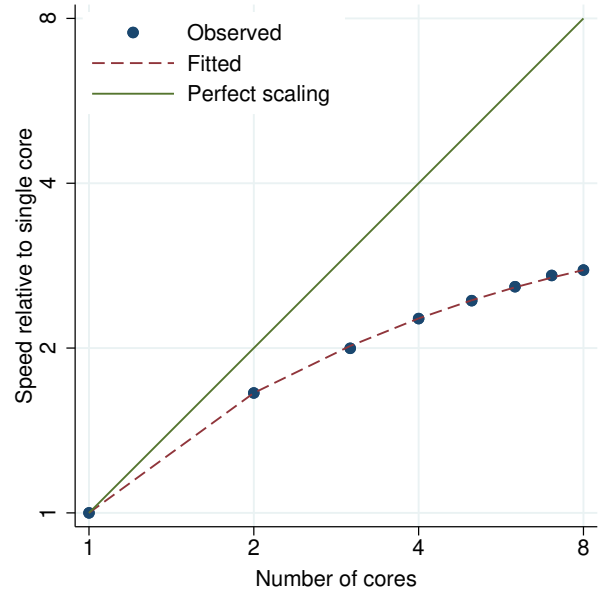


Figure 321. mi impute mono regress performance plot.

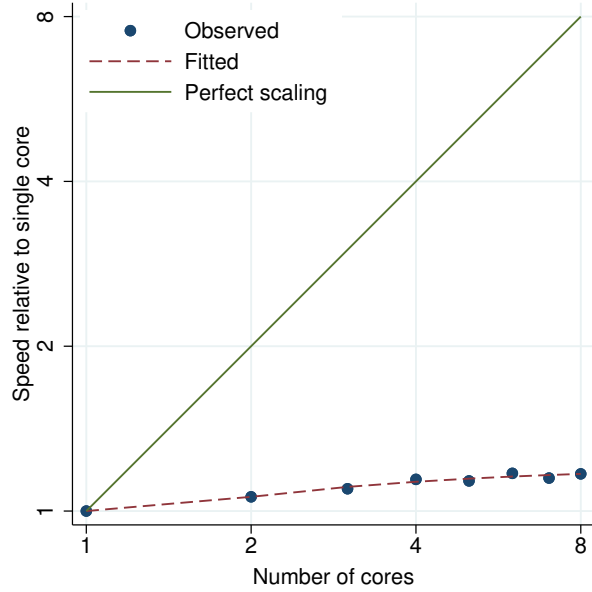


Figure 322. mi impute mvn performance plot.

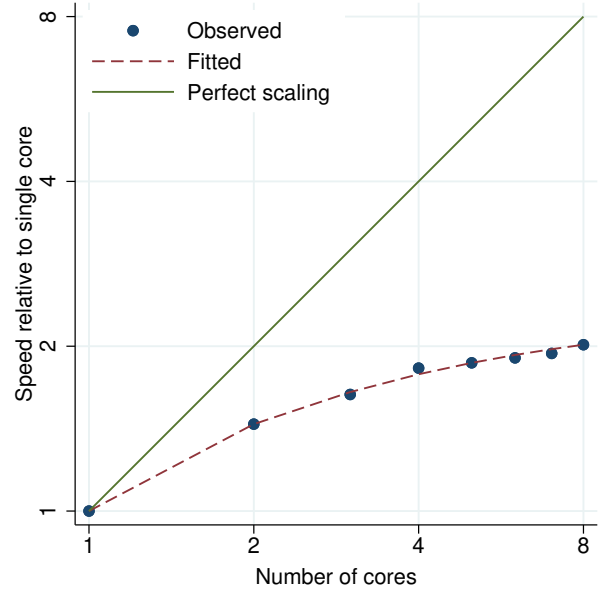


Figure 323. mi impute ologit performance plot.

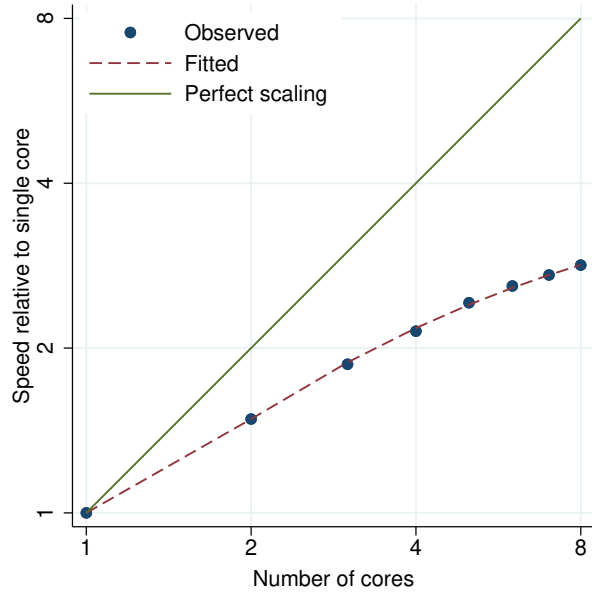


Figure 324. mi impute pmm performance plot.

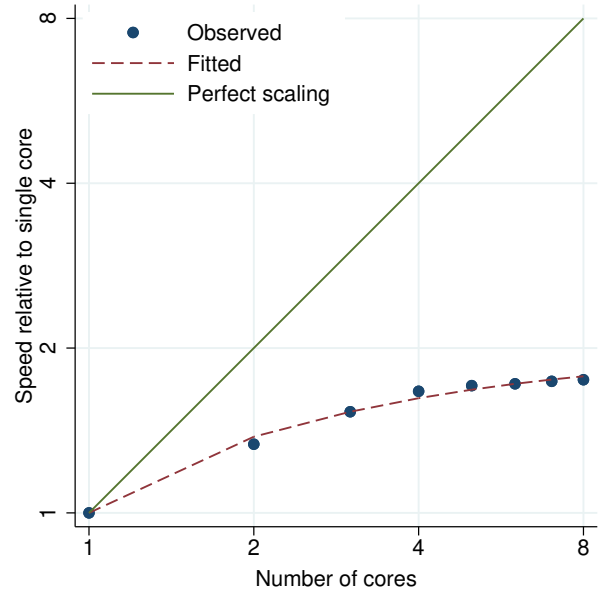


Figure 325. mi impute regress performance plot.

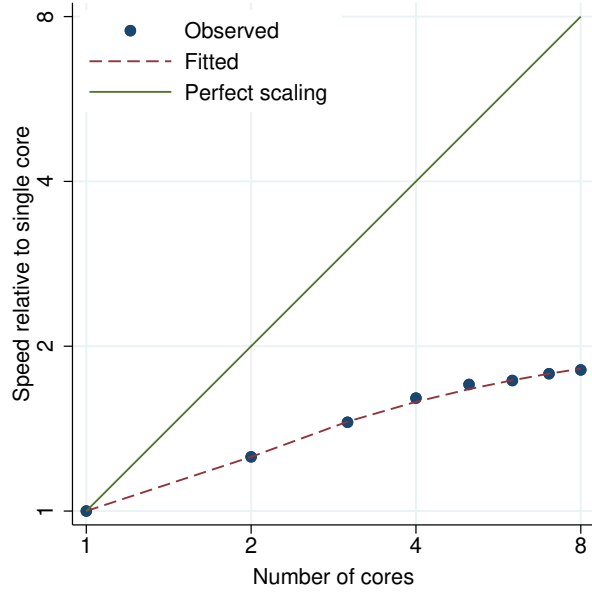


Figure 326. misstable nested performance plot.

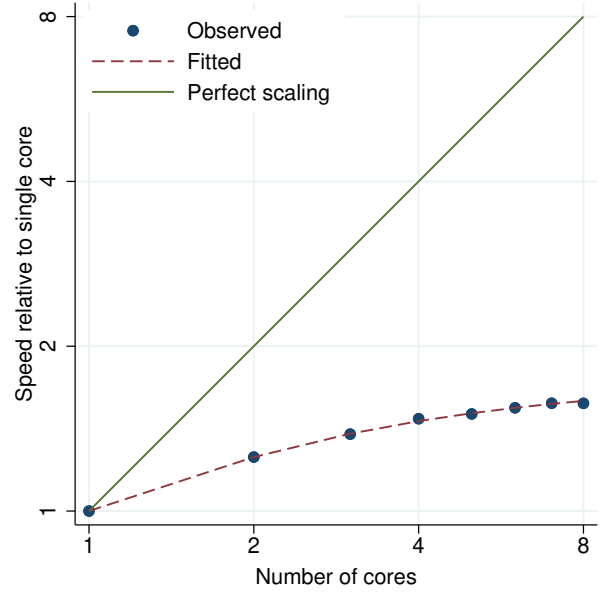


Figure 327. misstable patterns performance plot.

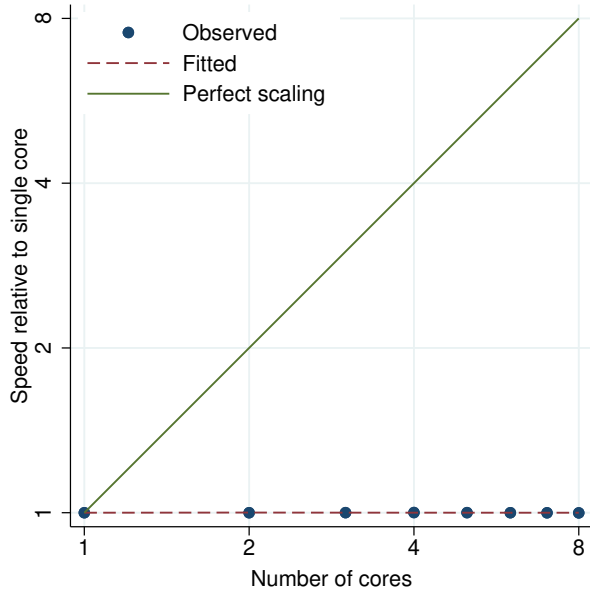


Figure 328. misstable summarize performance plot.

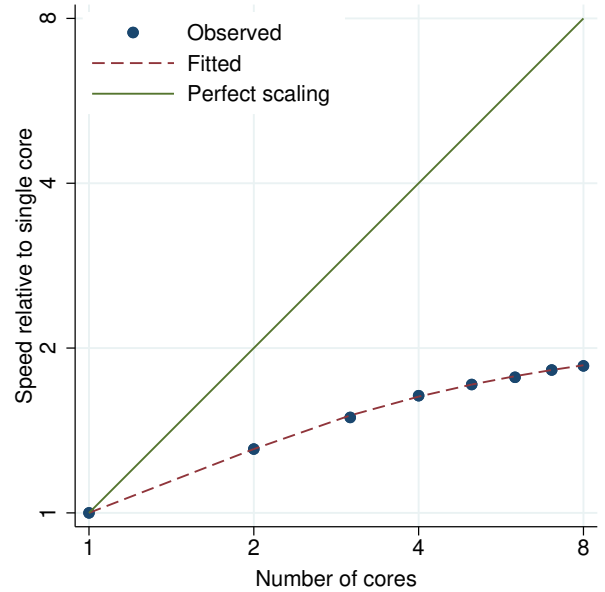


Figure 329. misstable tree performance plot.

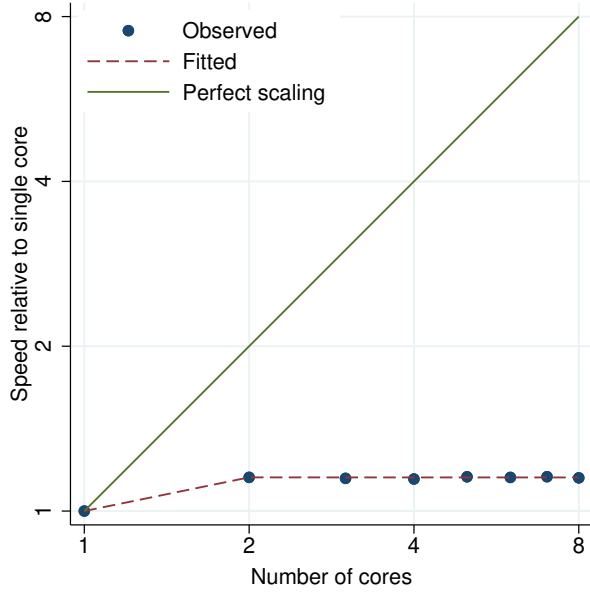


Figure 330. mixed performance plot.

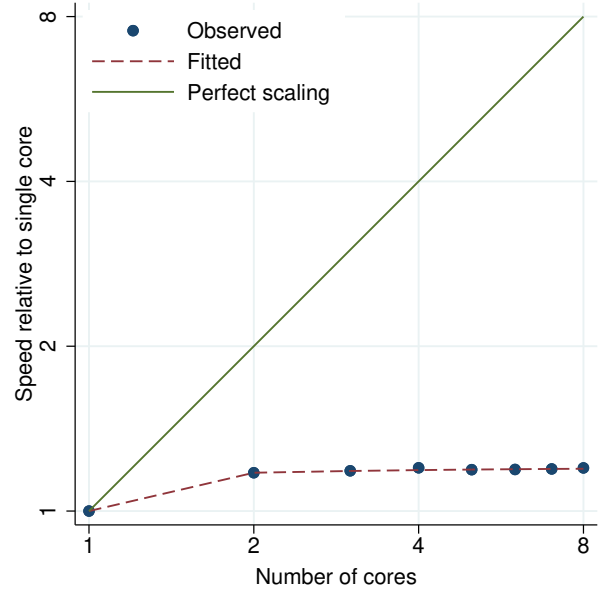


Figure 331. mixed (crossed effects) performance plot.

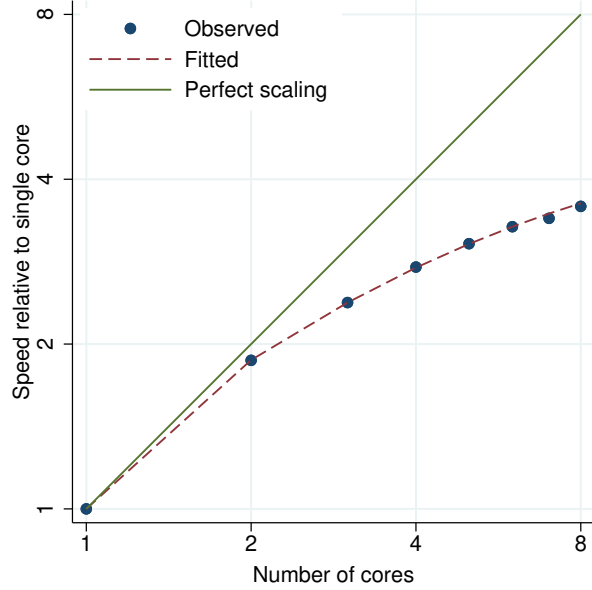


Figure 332. mkspline performance plot.

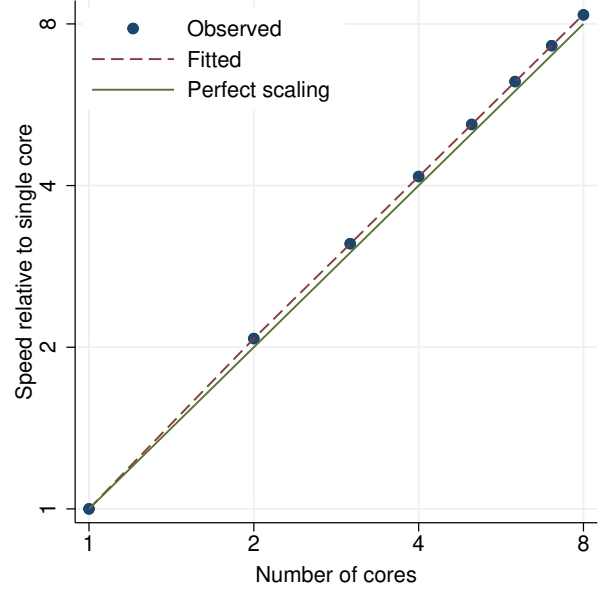


Figure 333. mlevel performance plot.

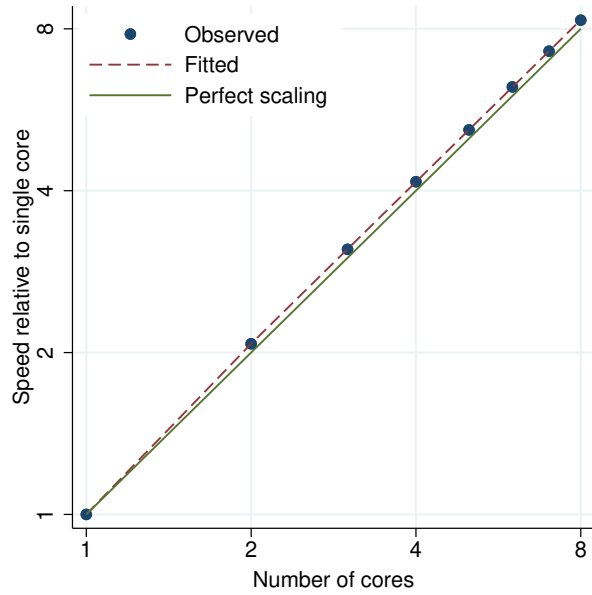


Figure 334. mlevel, nocons performance plot.

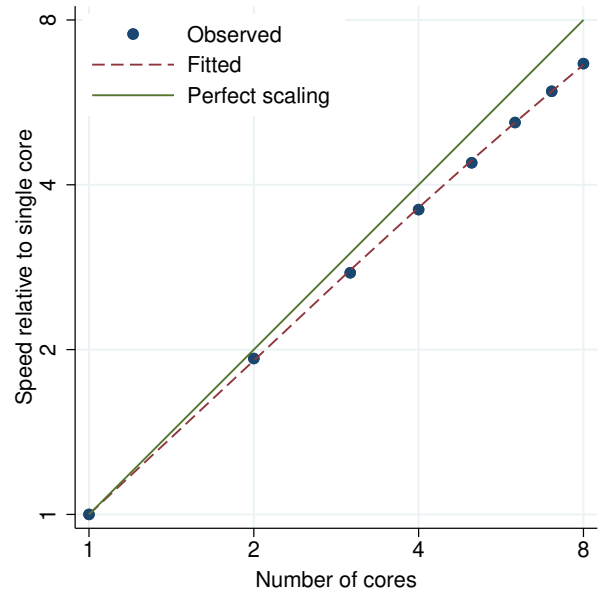


Figure 335. mlmatbysum performance plot.

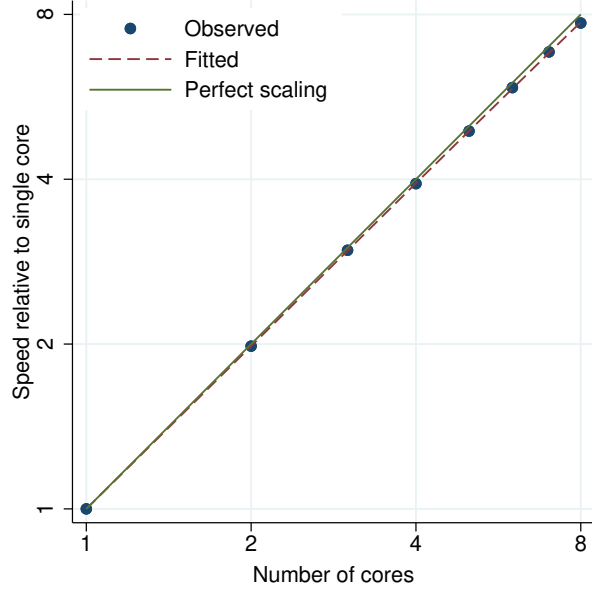


Figure 336. mpmatsum performance plot.

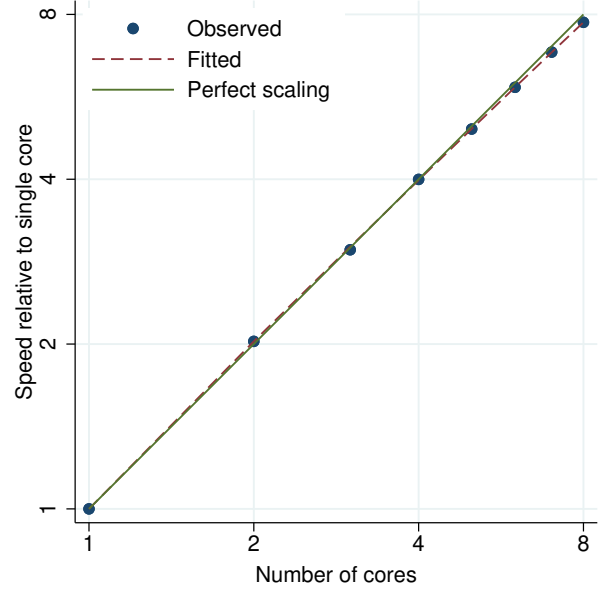


Figure 337. mlogit performance plot.

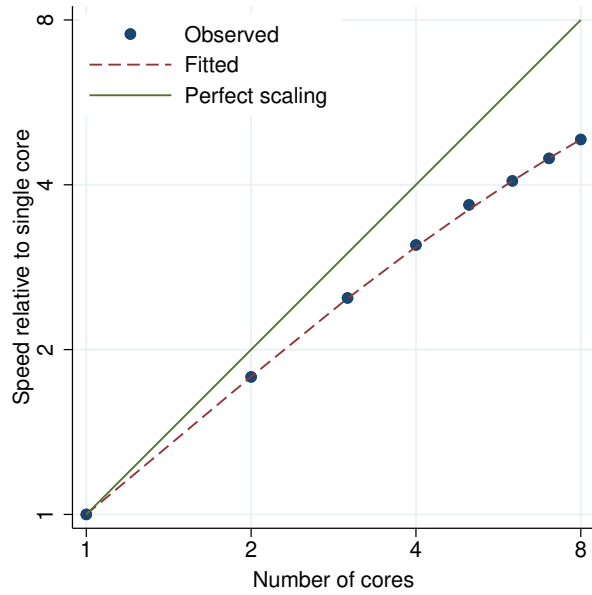


Figure 338. mlsum performance plot.

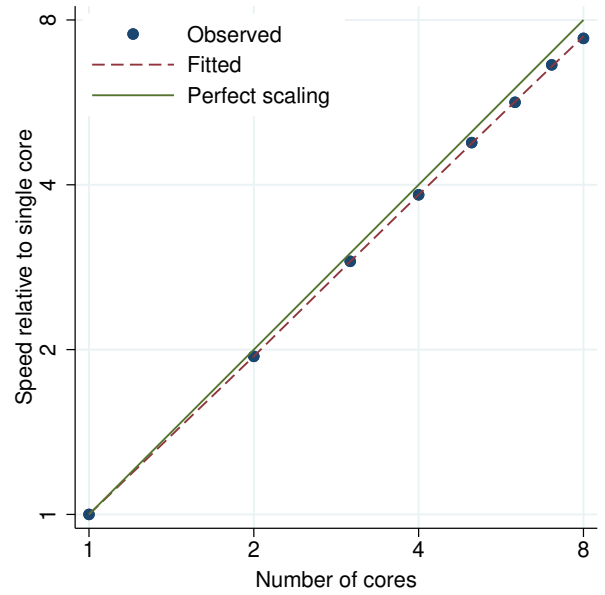


Figure 339. mlvecsum performance plot.

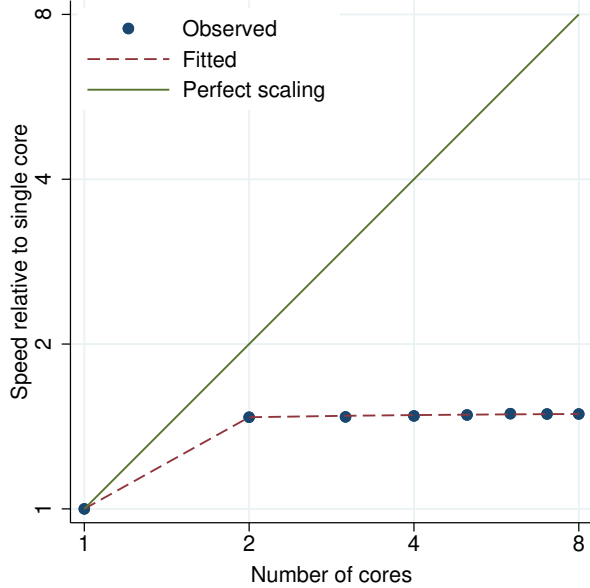


Figure 340. mprobit performance plot.

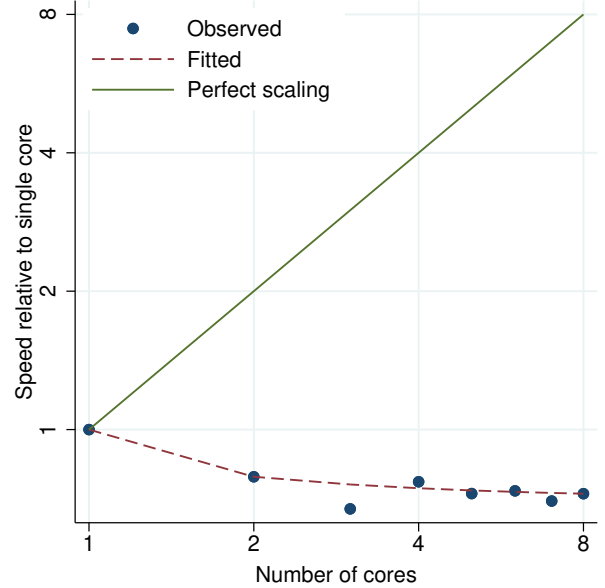


Figure 341. mswitch ar performance plot.

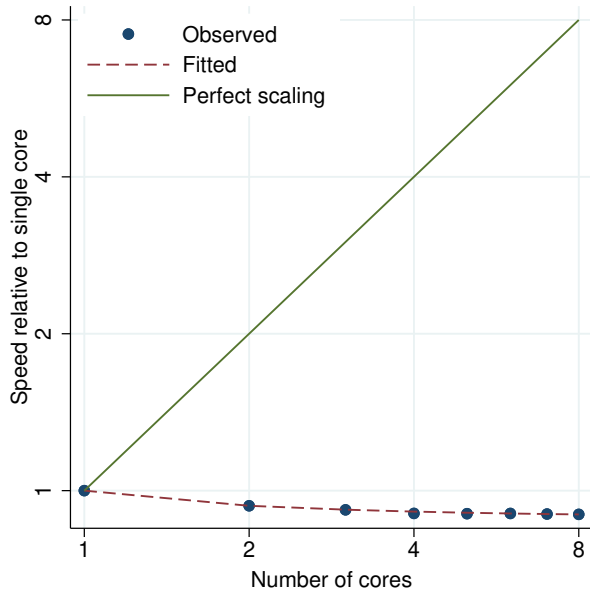


Figure 342. mswitch dr performance plot.

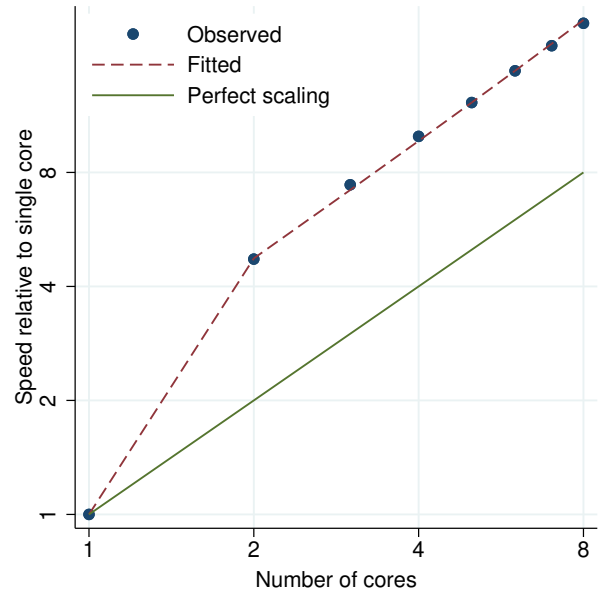


Figure 343. mvdecode performance plot.

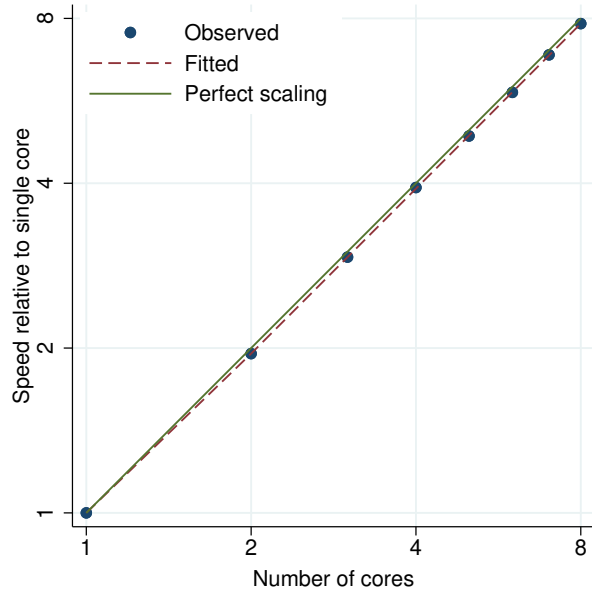


Figure 344. mvencode performance plot.

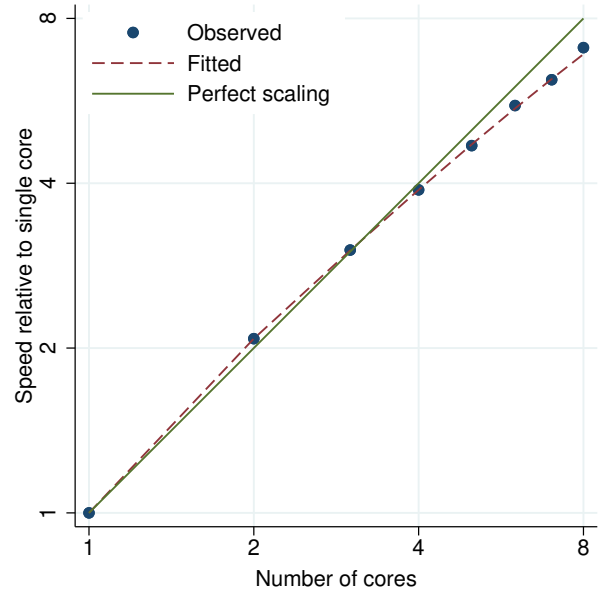


Figure 345. mvreg performance plot.

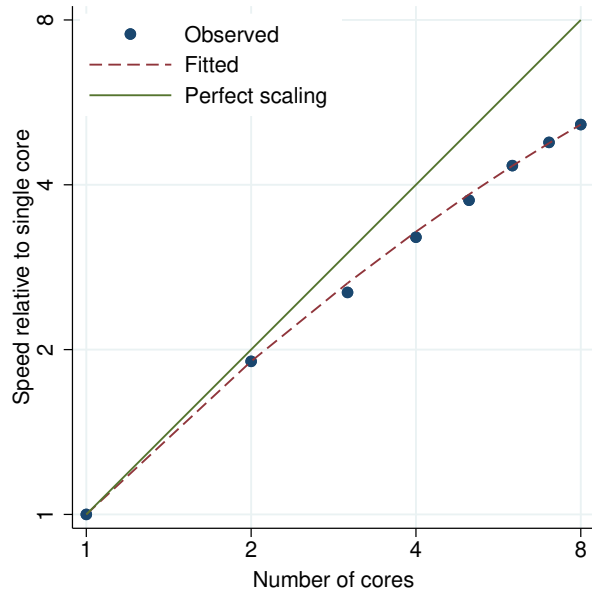


Figure 346. mvtest correlations performance plot.

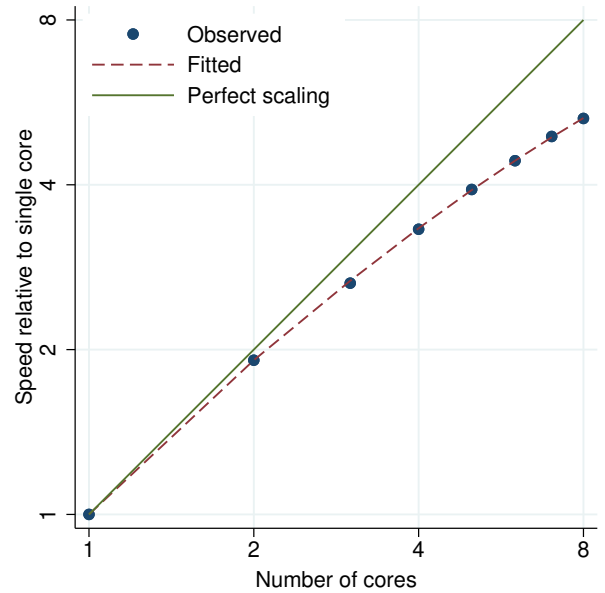


Figure 347. mvtest covariances performance plot.

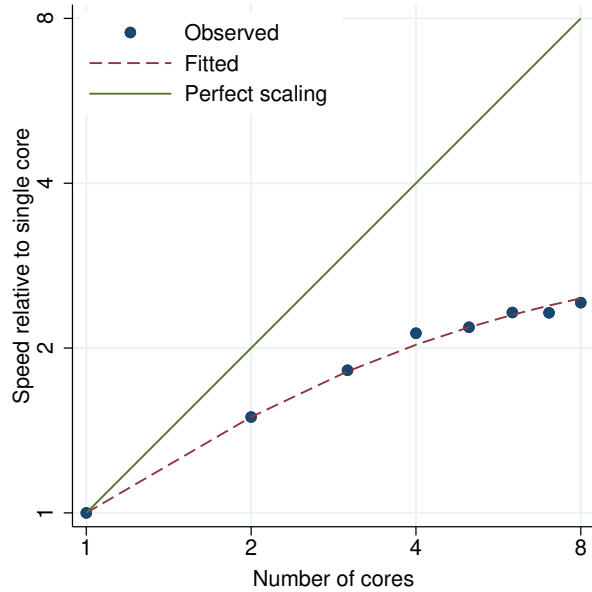


Figure 348. mvtest means, heterogeneous performance plot.

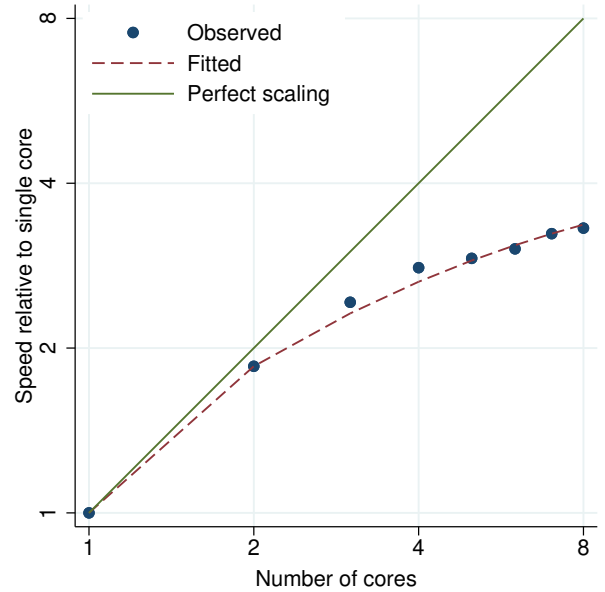


Figure 349. mvtest means, homogeneous performance plot.

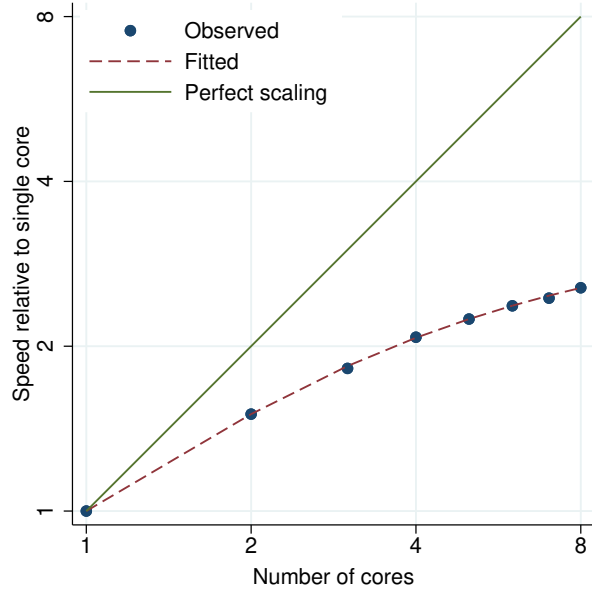


Figure 350. mvtest means, 1r performance plot.

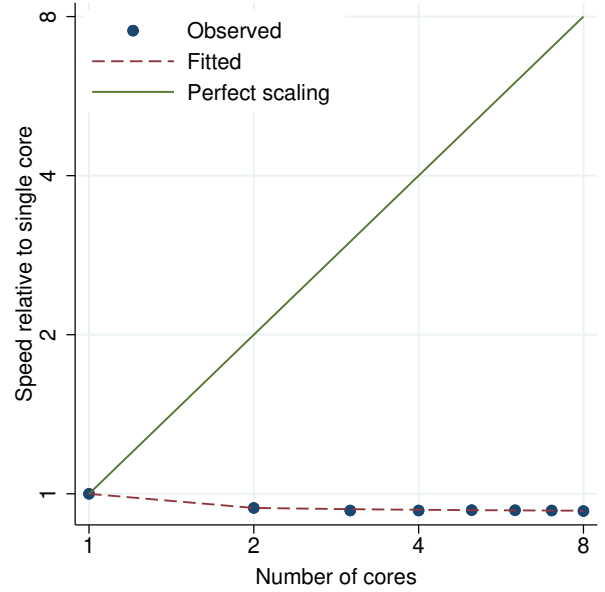


Figure 351. mvtest normality performance plot.

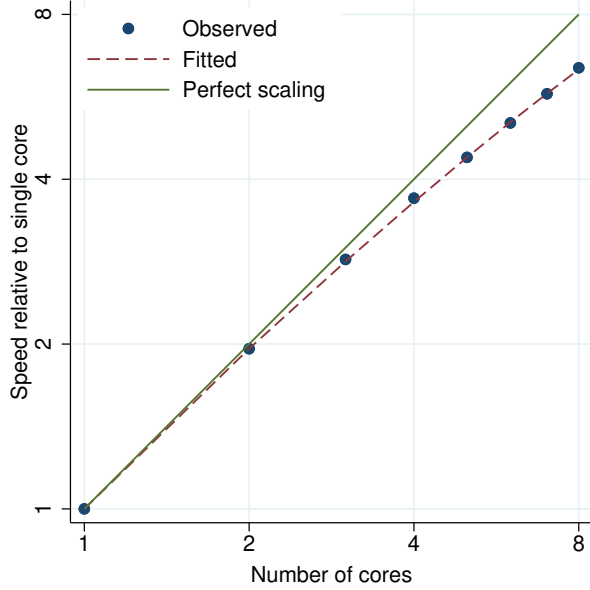


Figure 352. nbreg performance plot.

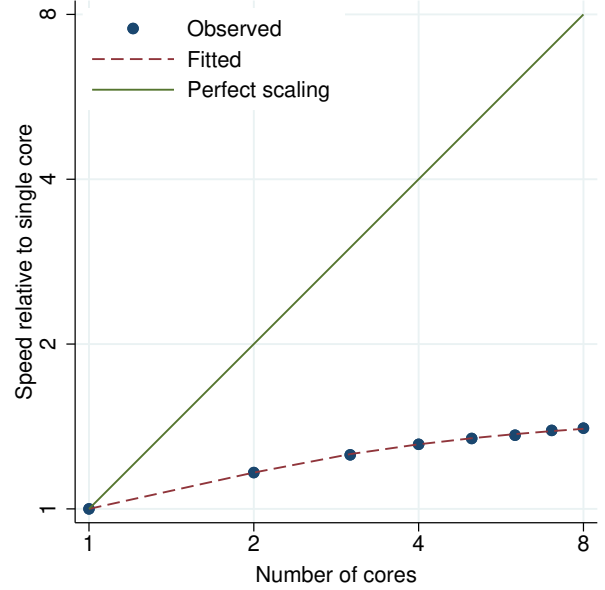


Figure 353. newey performance plot.

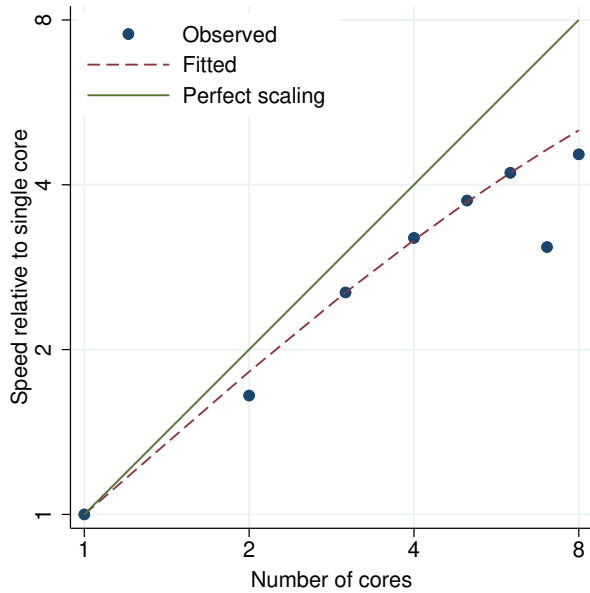


Figure 354. nl performance plot.

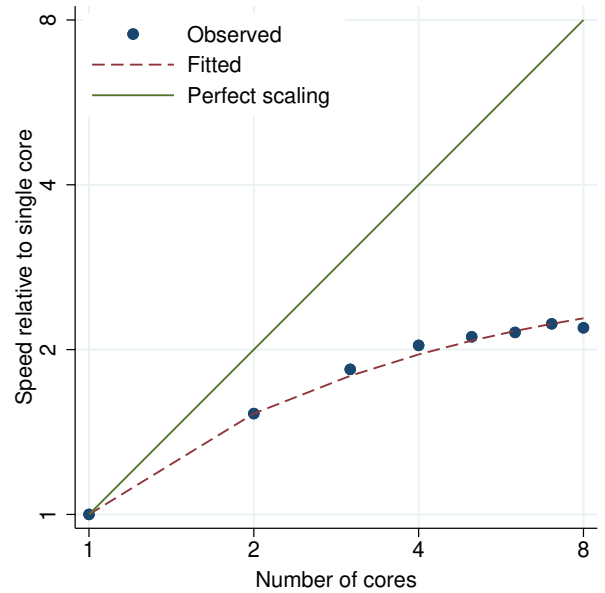


Figure 355. nlogit performance plot.

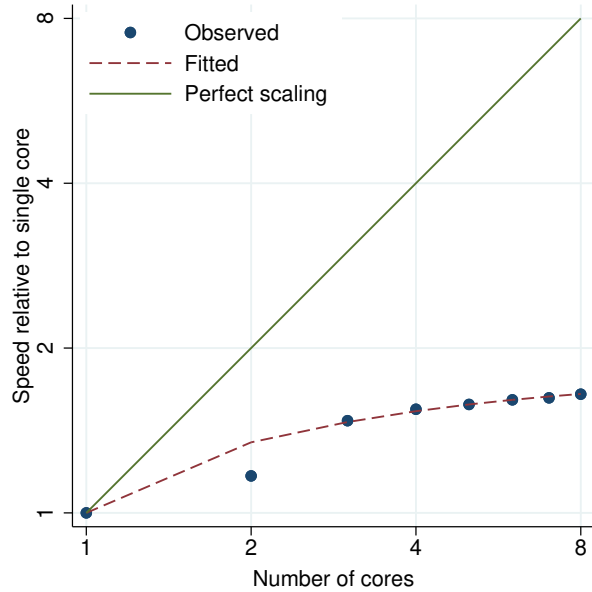


Figure 356. nlsur performance plot.

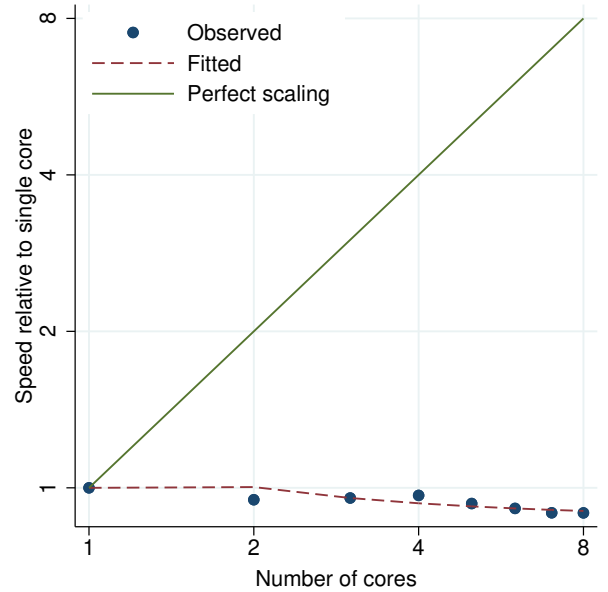


Figure 357. nprogrss kernel performance plot.

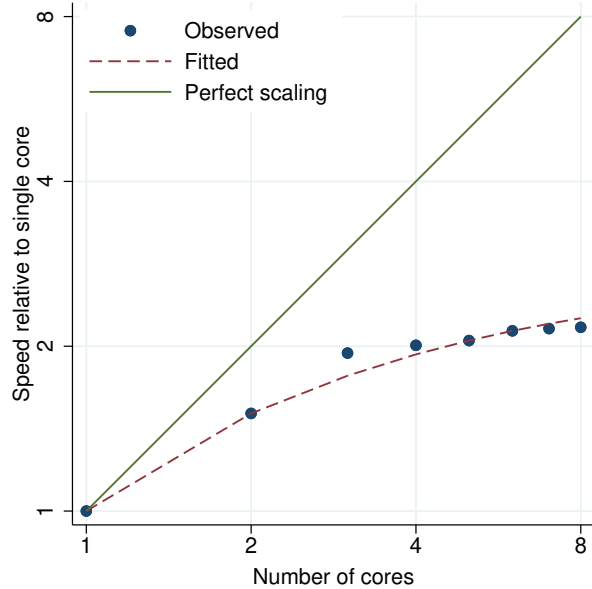


Figure 358. nptrend, cuzick performance plot.

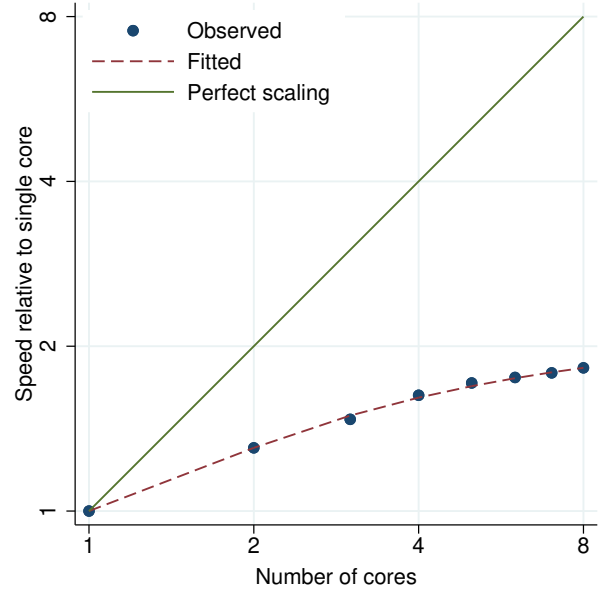


Figure 359. nptrend, carmitage performance plot.

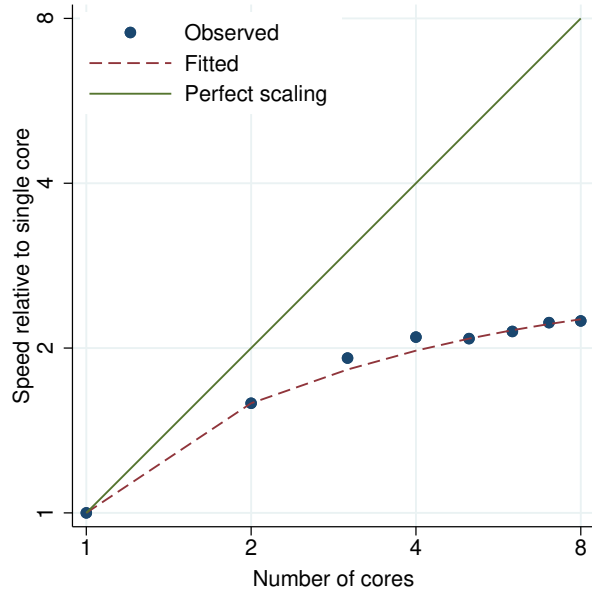


Figure 360. nptrend, jterpstra performance plot.

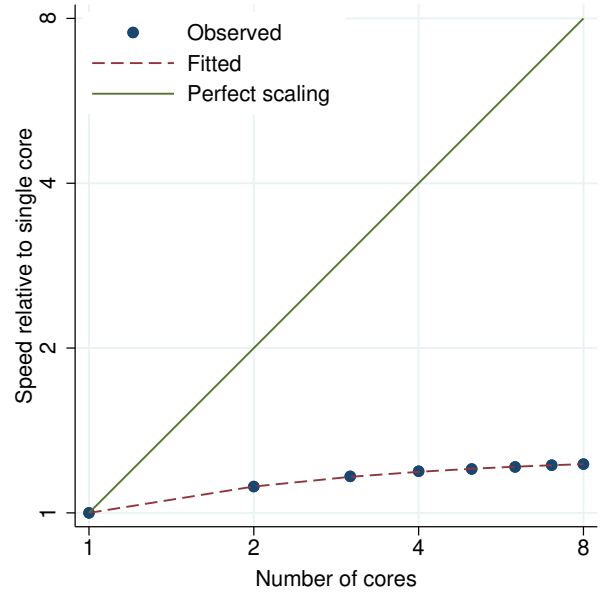


Figure 361. nptrend, linear performance plot.

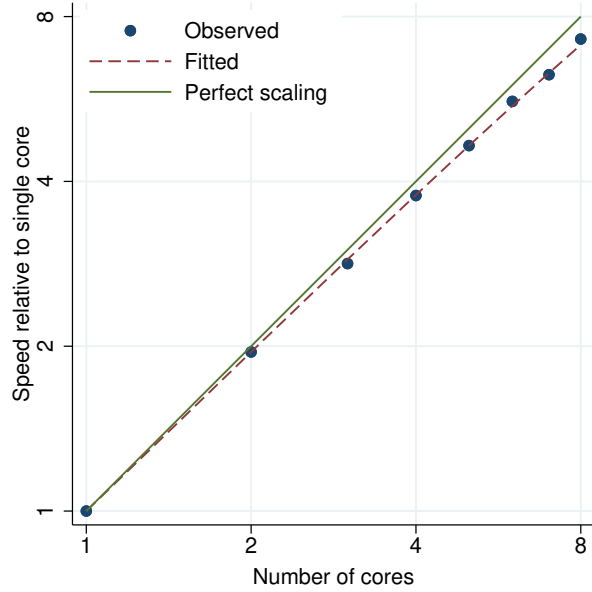


Figure 362. ologit performance plot.

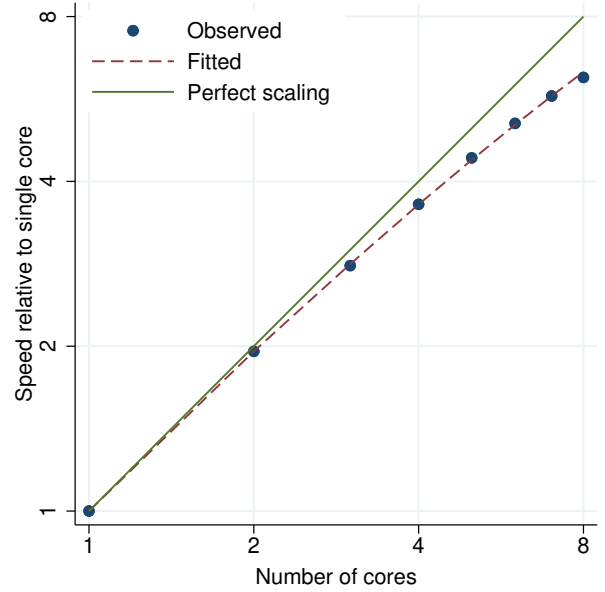


Figure 363. ologit, vce(cluster) performance plot.

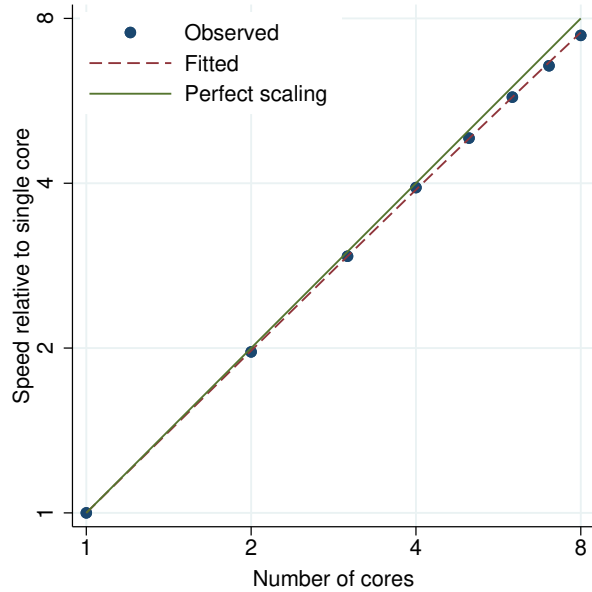


Figure 364. `ologit, vce(robust)` performance plot.

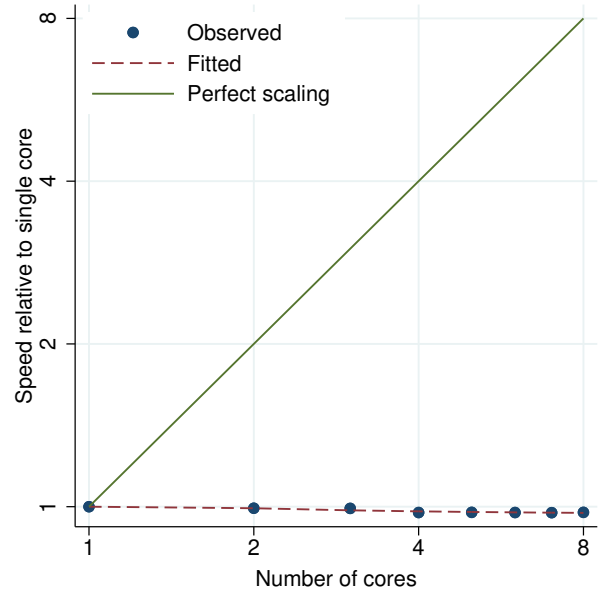


Figure 365. `oneway` performance plot.

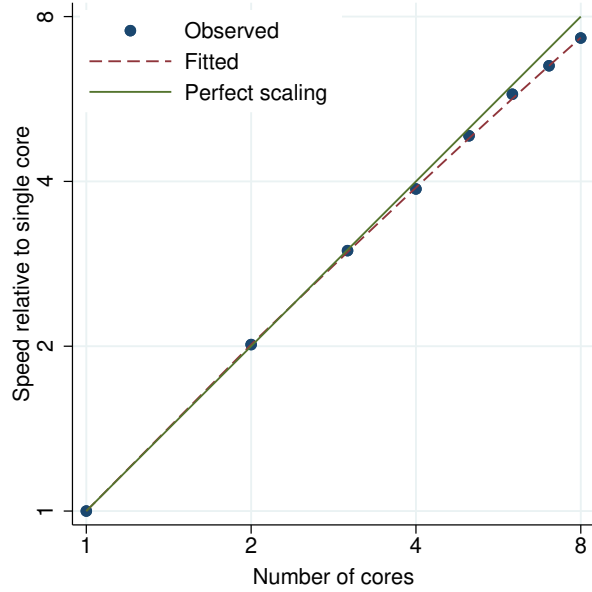


Figure 366. `oprobit` performance plot.

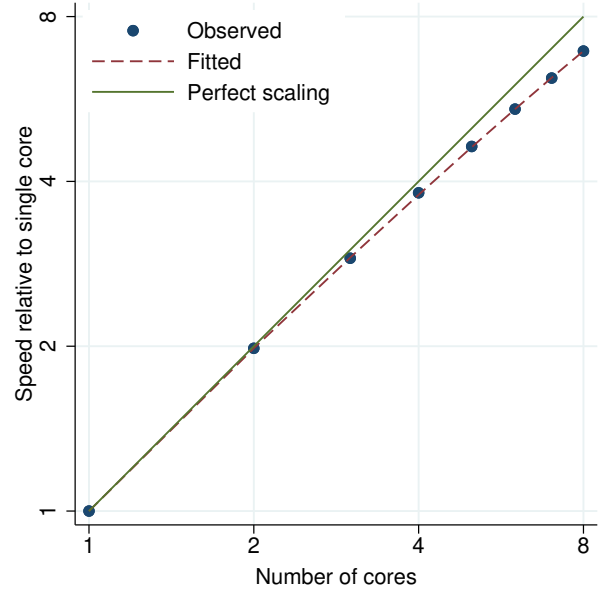


Figure 367. `oprobit, vce(cluster)` performance plot.

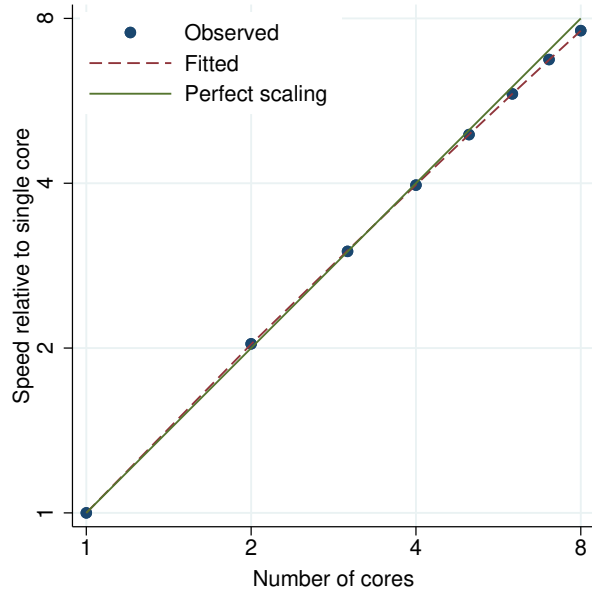


Figure 368. oprobit, vce(robust) performance plot.

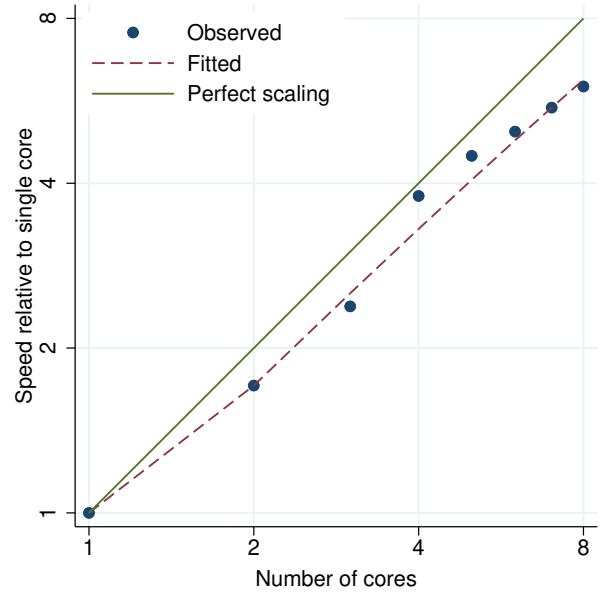


Figure 369. orthog performance plot.

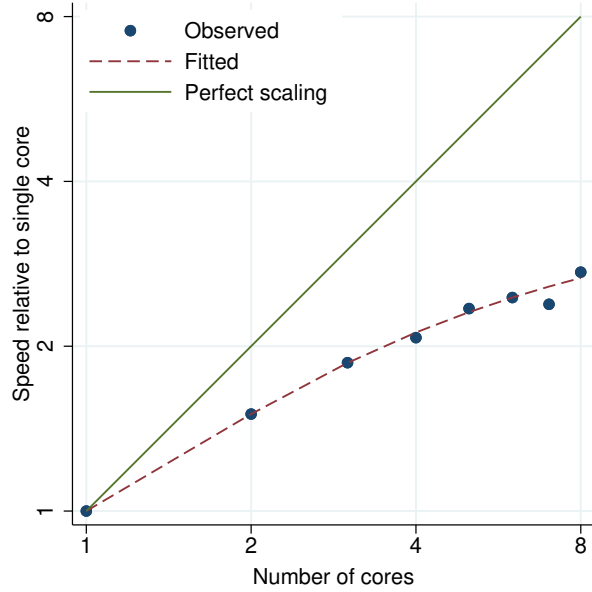


Figure 370. pca performance plot.

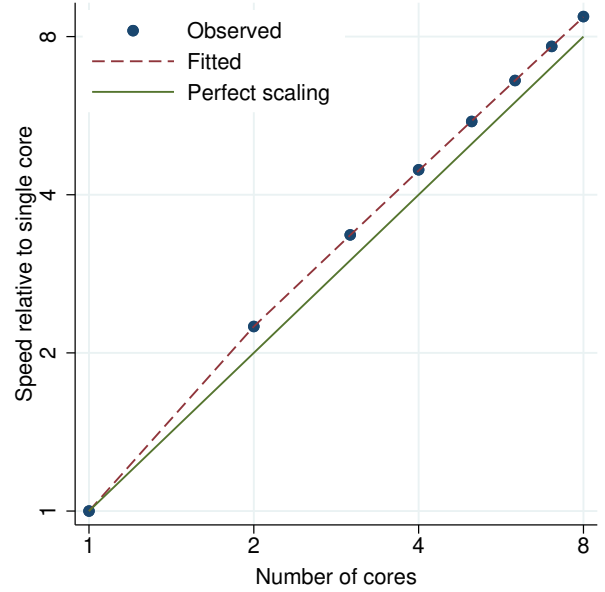


Figure 371. pcorr performance plot.

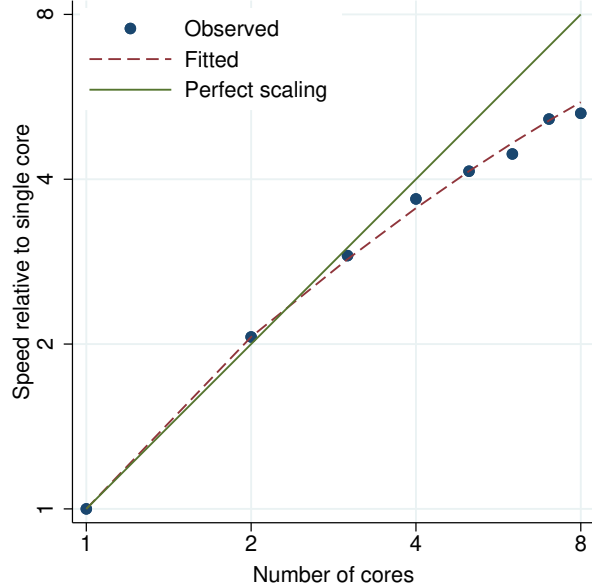


Figure 372. pctile performance plot.

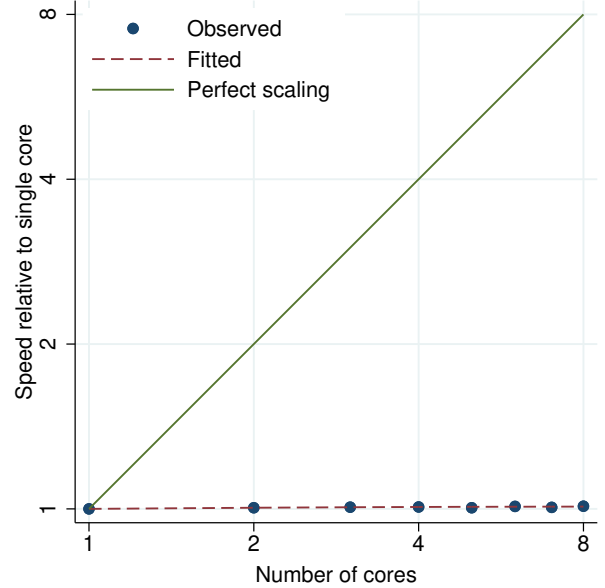


Figure 373. pergram performance plot.

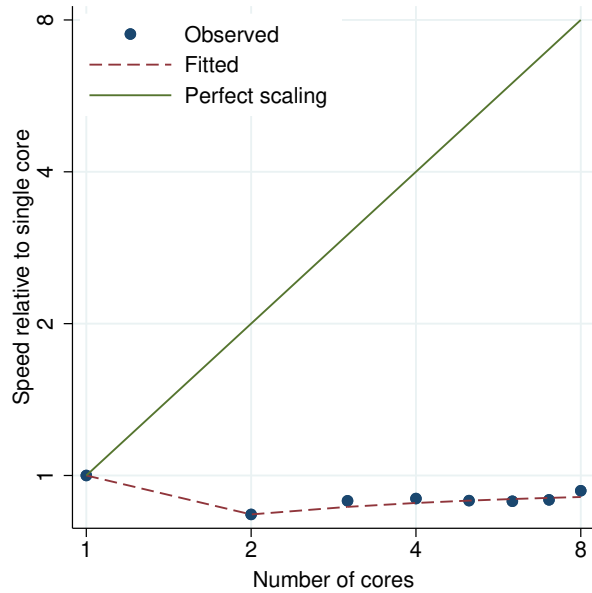


Figure 374. pkcollapse performance plot.

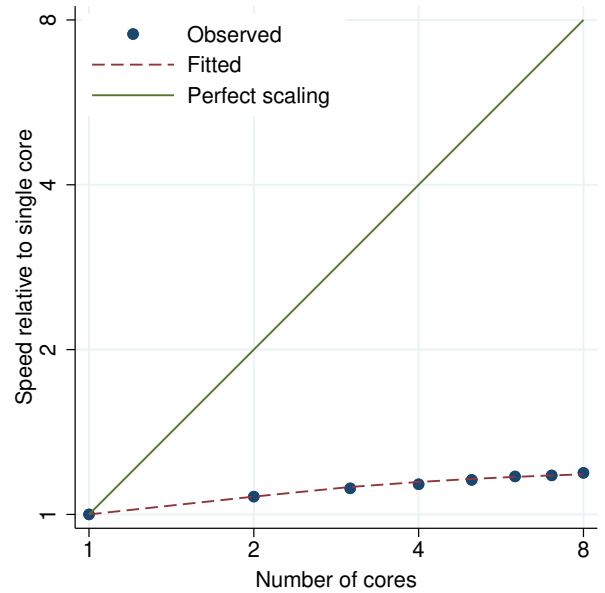


Figure 375. pkexamine performance plot.

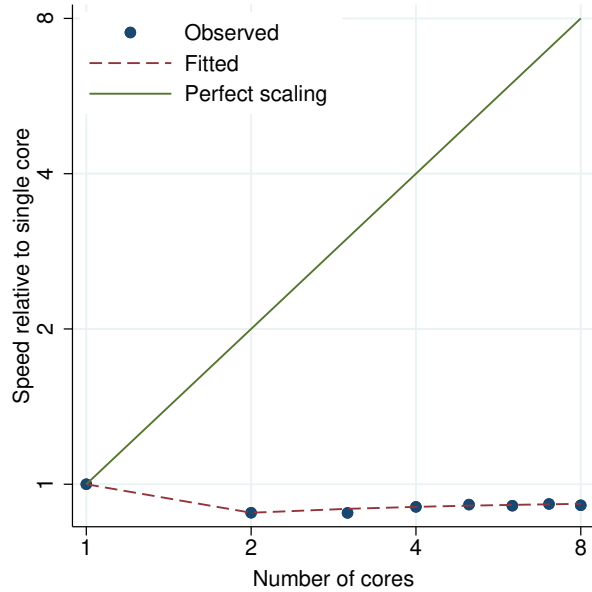


Figure 376. pksumm performance plot.

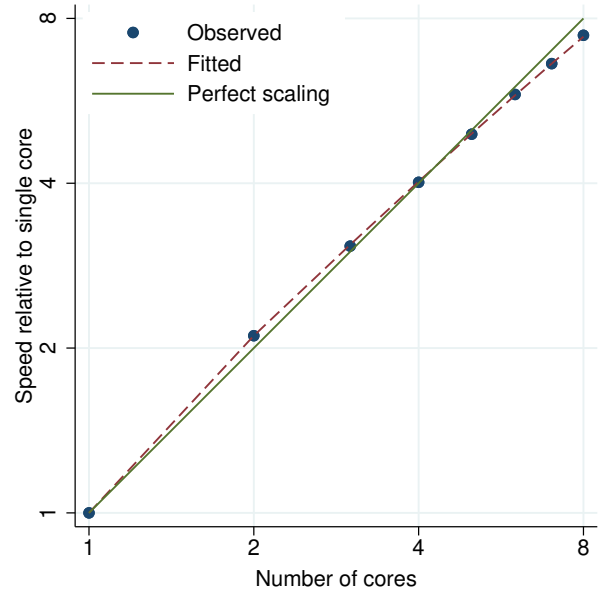


Figure 377. poisson performance plot.

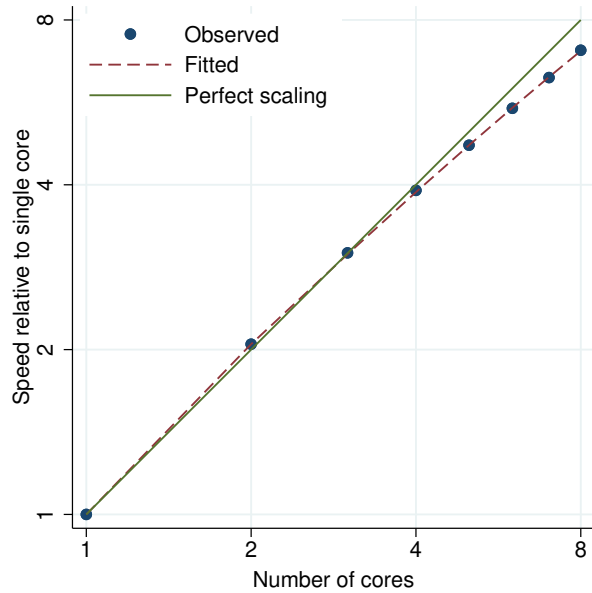


Figure 378. poisson, vce(cluster) performance plot.

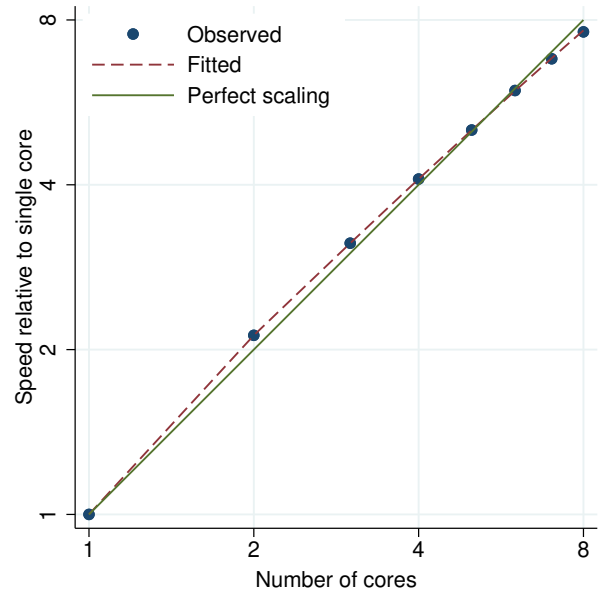


Figure 379. poisson, exposure() performance plot.

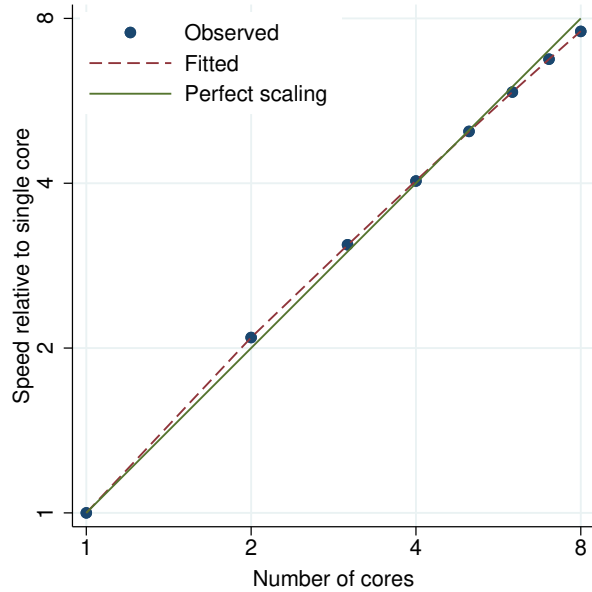


Figure 380. poisson, vce(robust) performance plot.

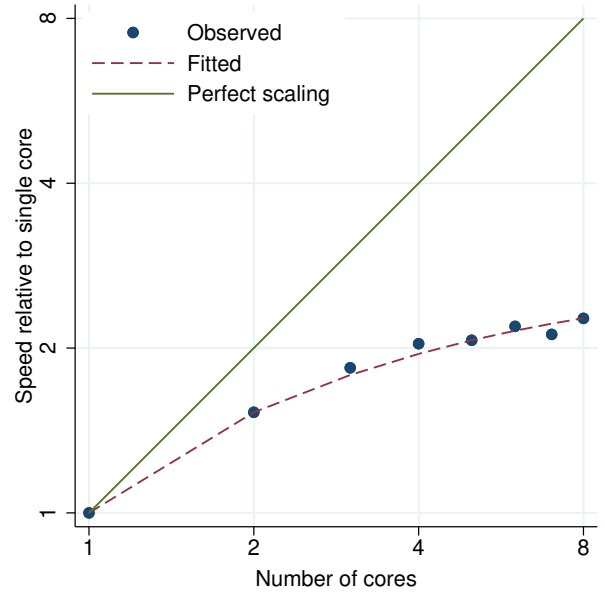


Figure 381. pologit performance plot.

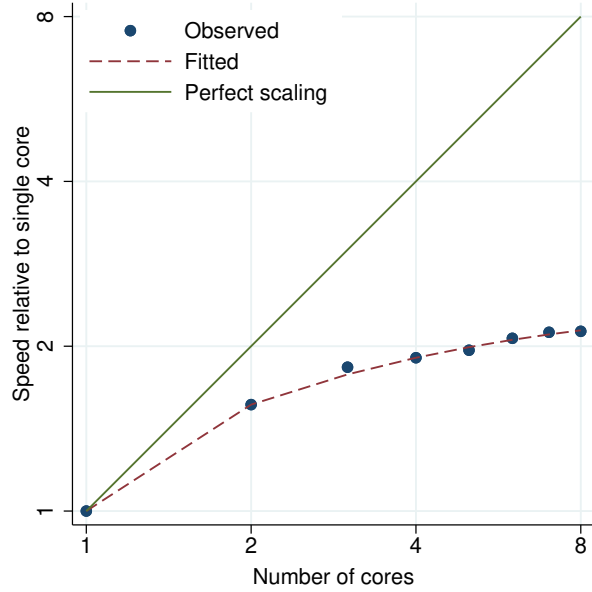


Figure 382. popoisson performance plot.

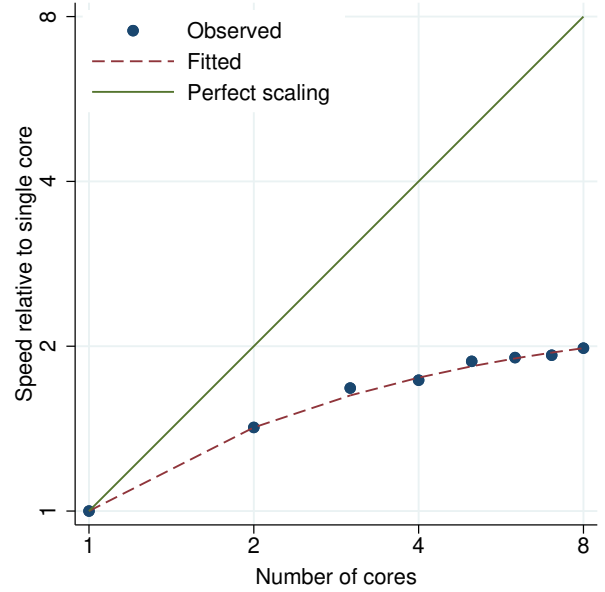


Figure 383. poregress performance plot.

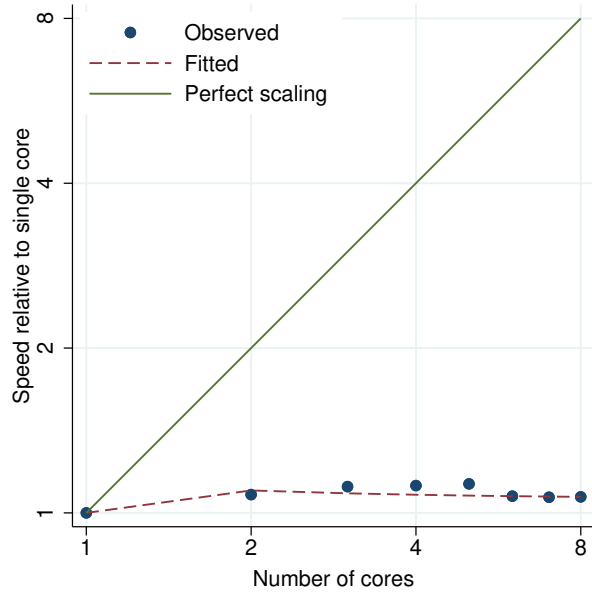


Figure 384. pperron performance plot.

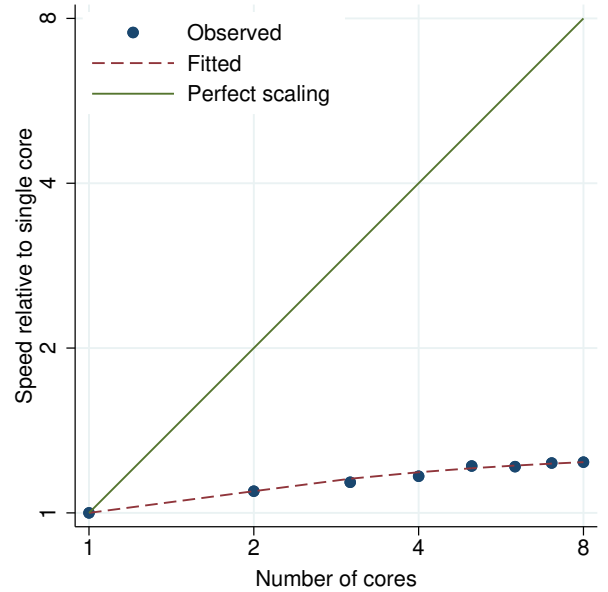


Figure 385. prais performance plot.

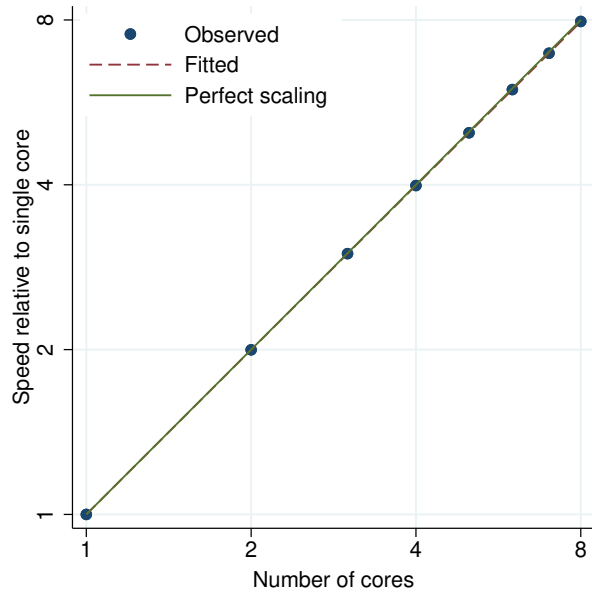


Figure 386. predict, cooksd performance plot.

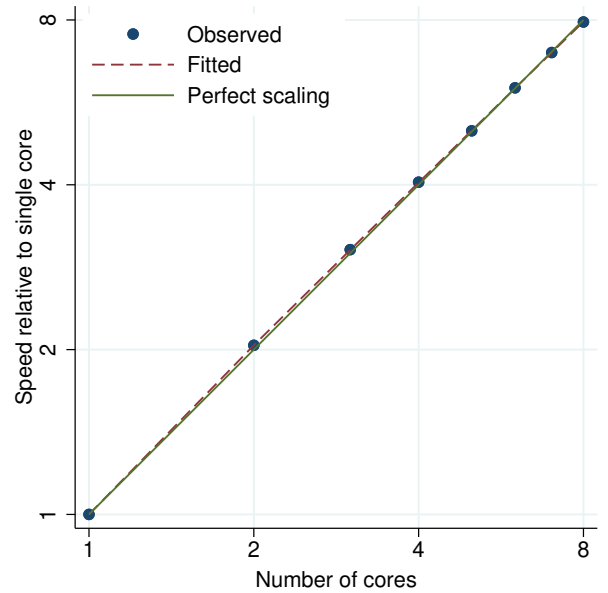


Figure 387. predict, covratio performance plot.

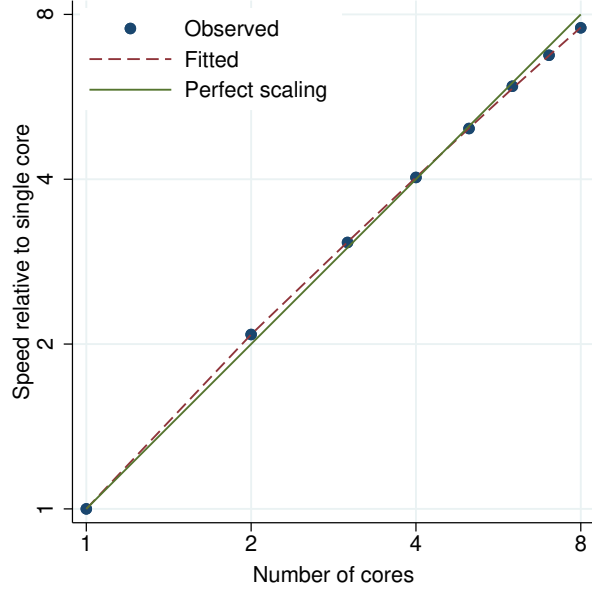


Figure 388. predict, dfbeta performance plot.

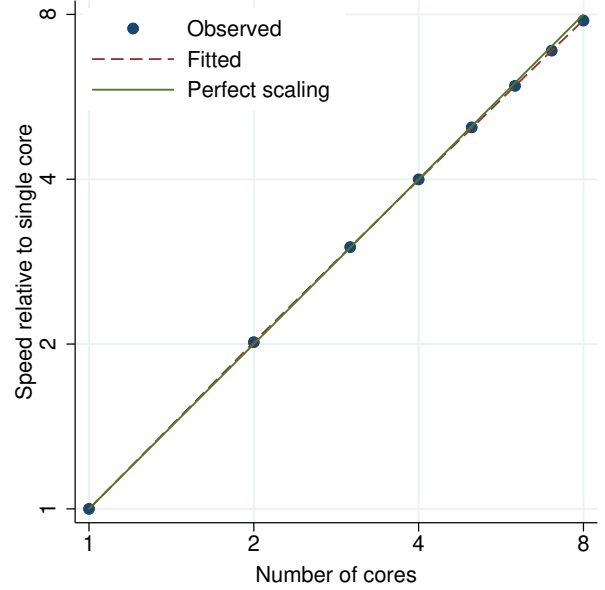


Figure 389. predict, dfits performance plot.

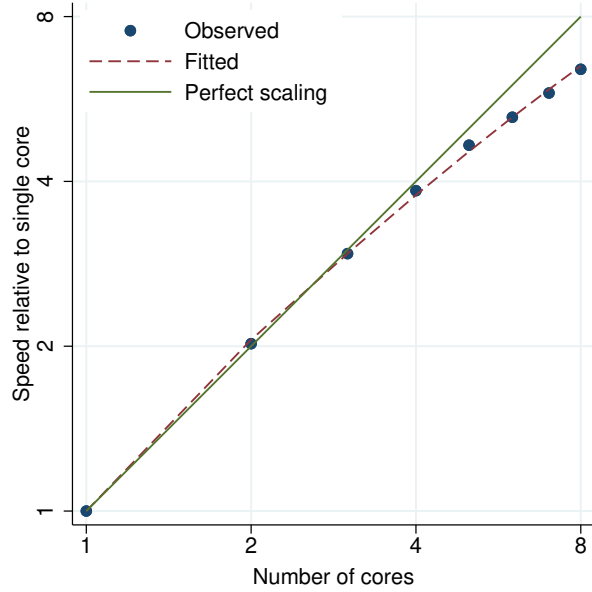


Figure 390. predict, e performance plot.

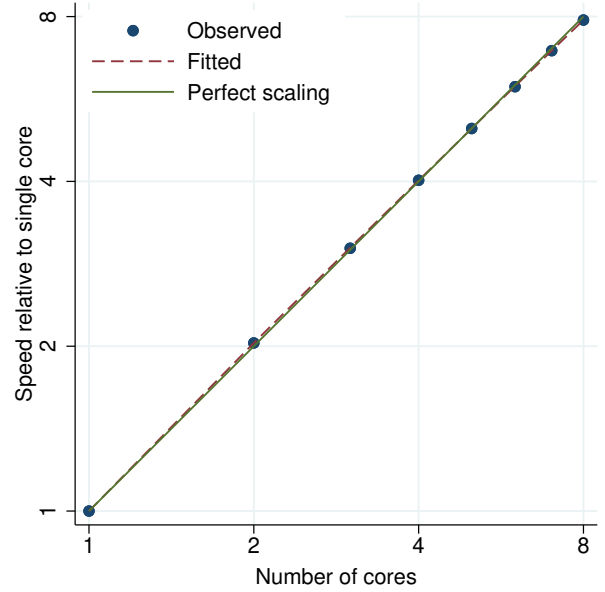


Figure 391. predict, leverage performance plot.

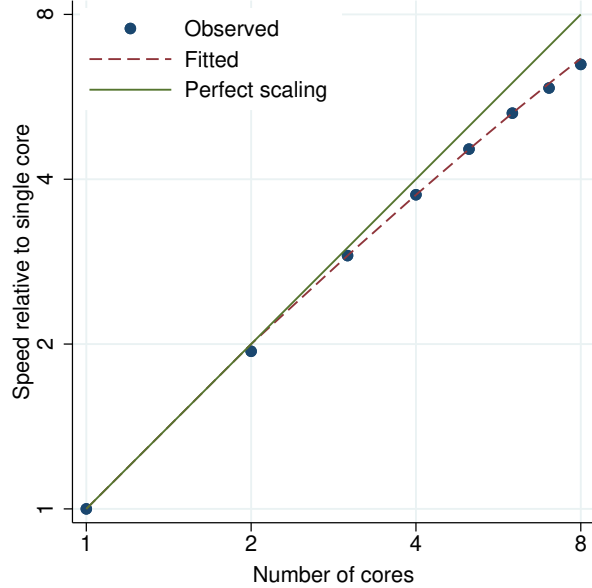


Figure 392. predict, pr performance plot.

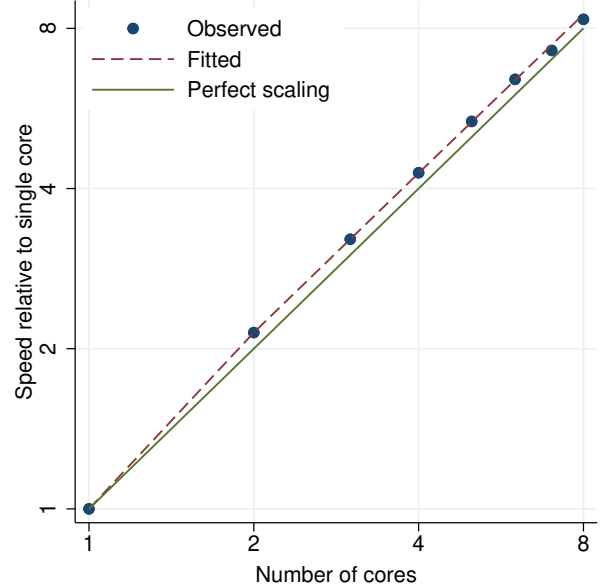


Figure 393. predict, residuals performance plot.

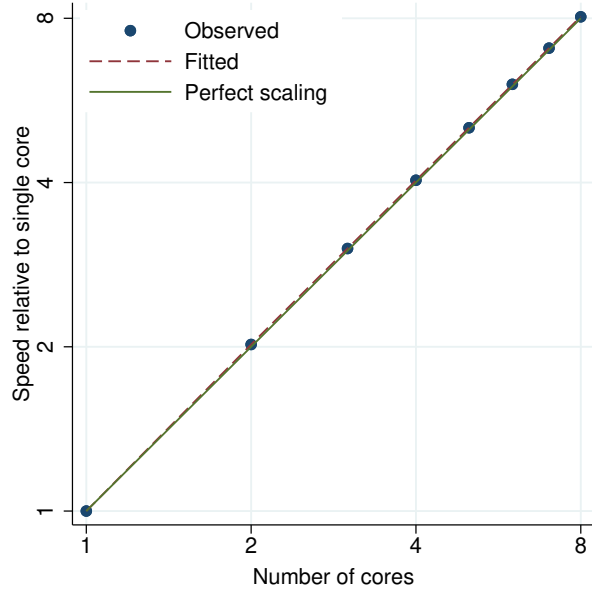


Figure 394. predict, rstandard performance plot.

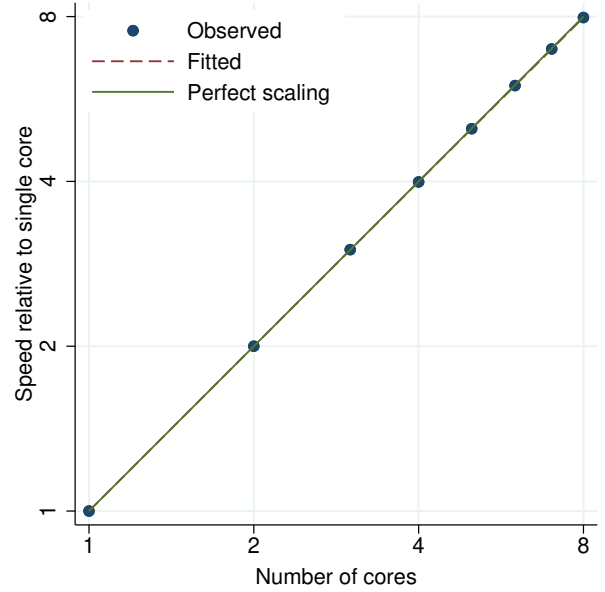


Figure 395. predict, rstudent performance plot.

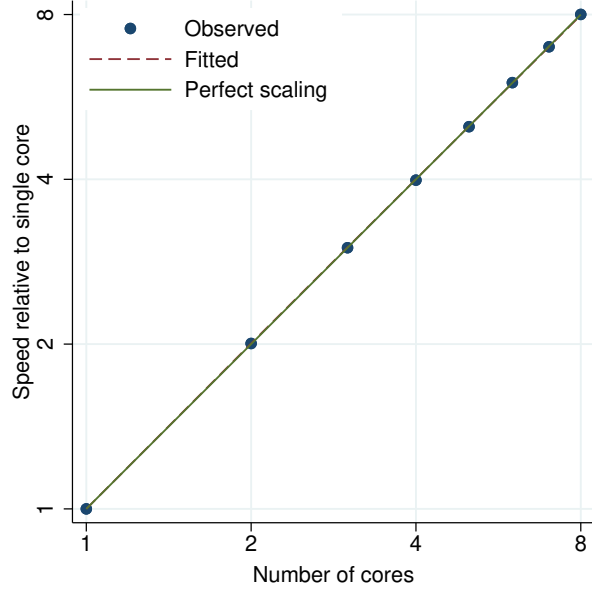


Figure 396. predict, stdf performance plot.

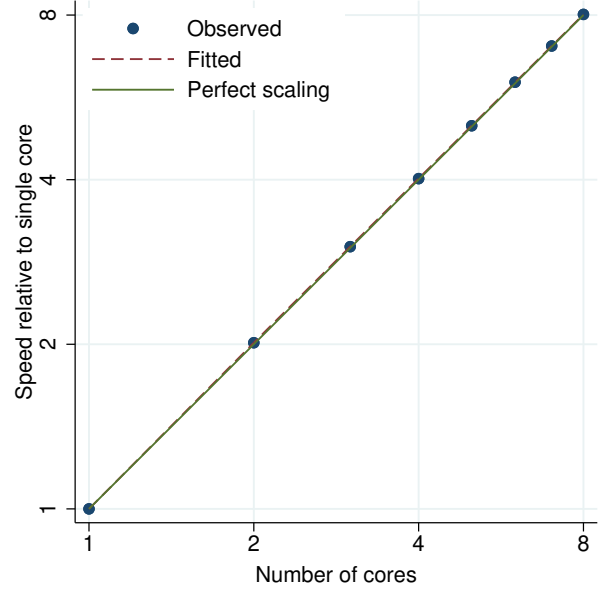


Figure 397. predict, stdp performance plot.

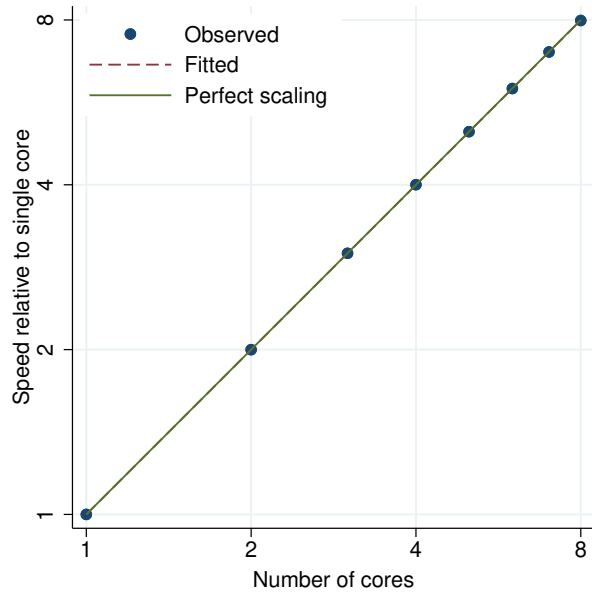


Figure 398. predict, stdr performance plot.

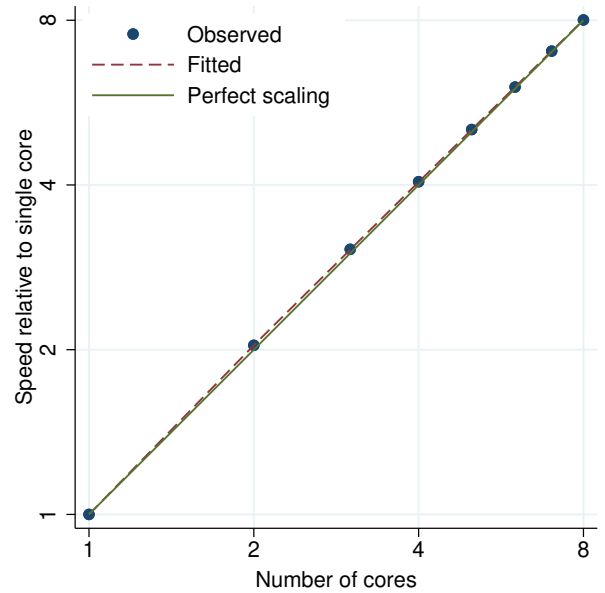


Figure 399. predict, welsch performance plot.

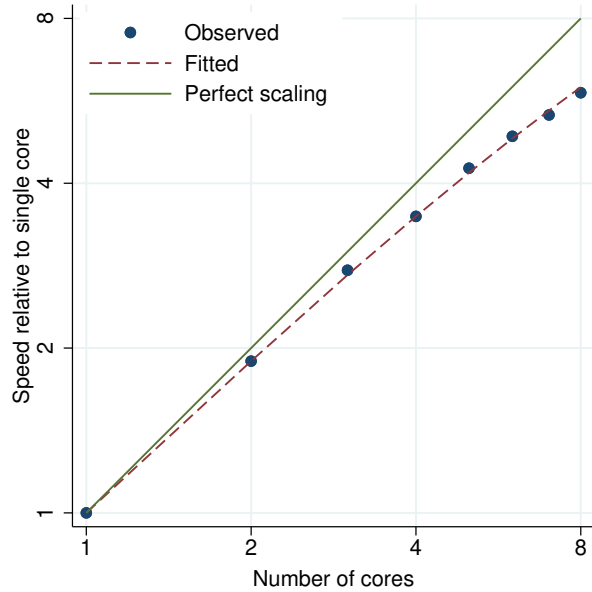


Figure 400. predict, ystar performance plot.

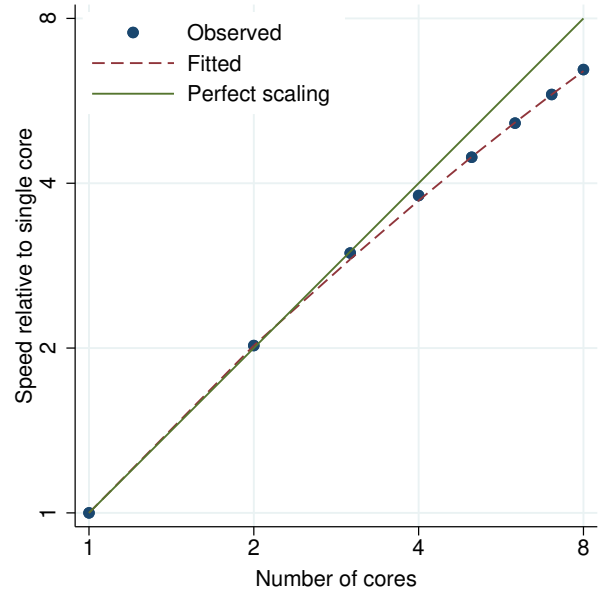


Figure 401. predictnl performance plot.

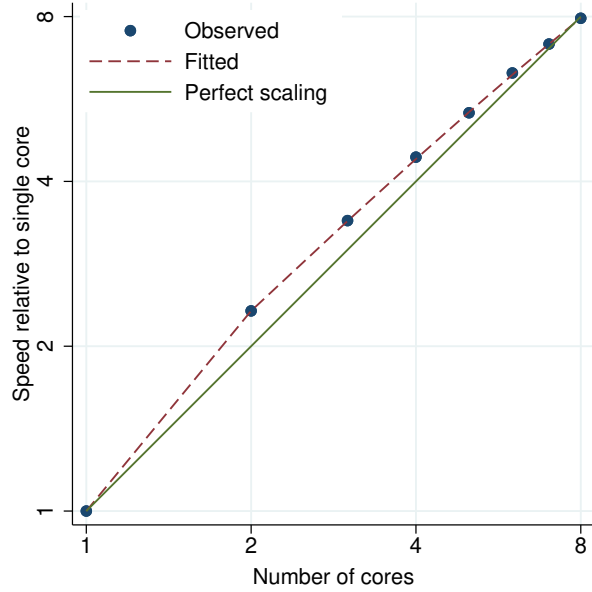


Figure 402. probit performance plot.

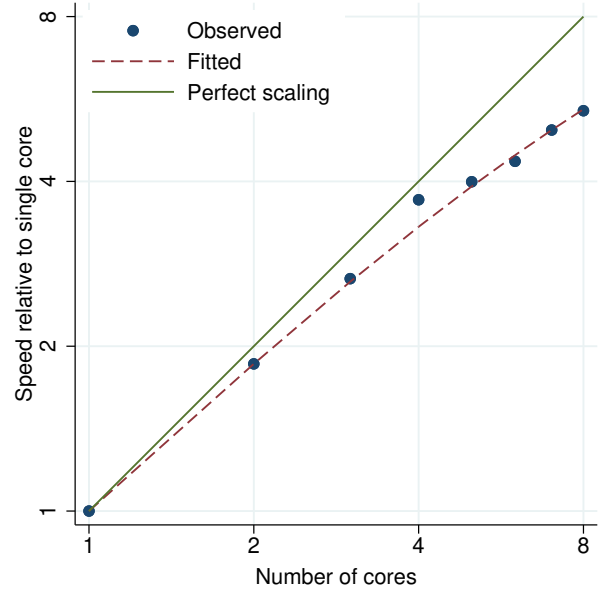


Figure 403. procrustes performance plot.

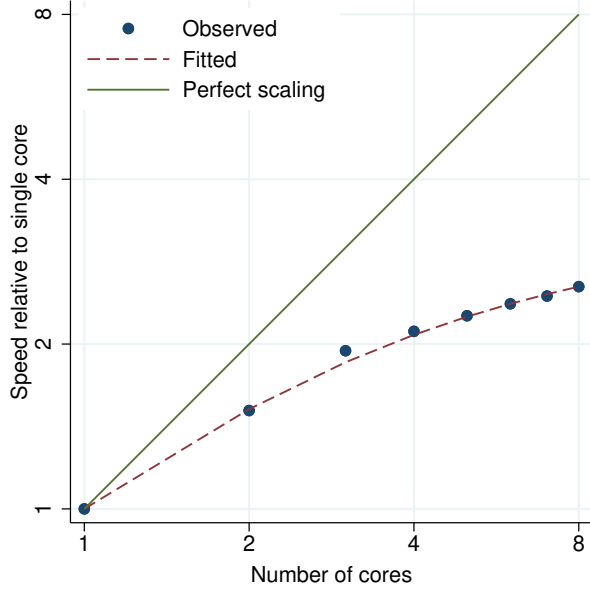


Figure 404. proportion performance plot.

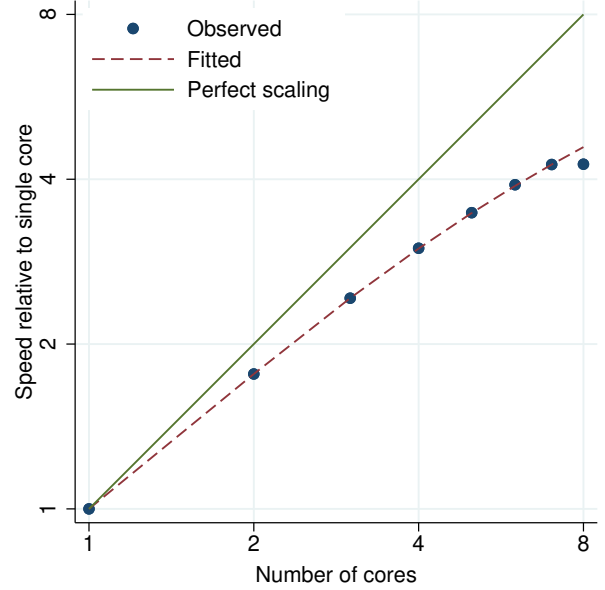


Figure 405. prtest1 performance plot.

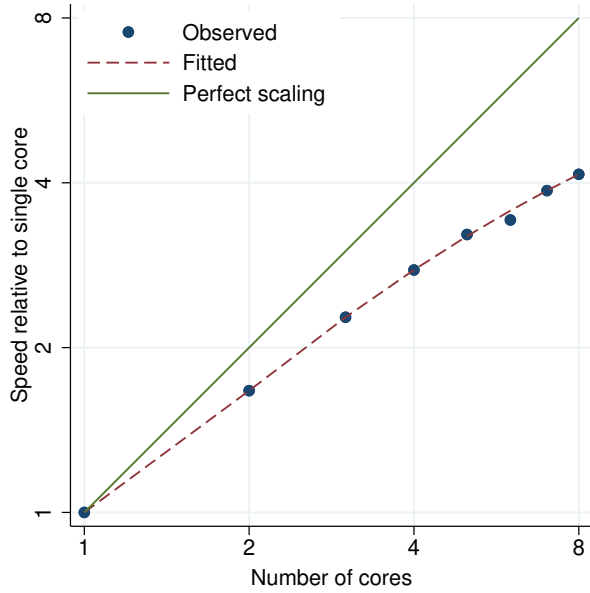


Figure 406. prtest2 performance plot.

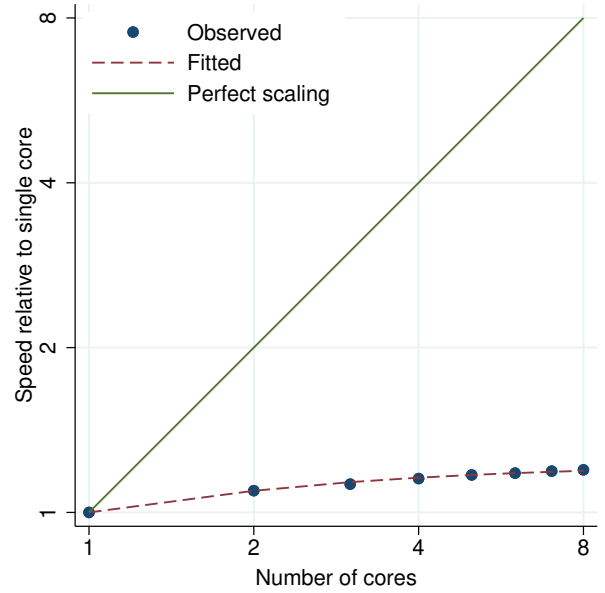


Figure 407. prtest, by() performance plot.

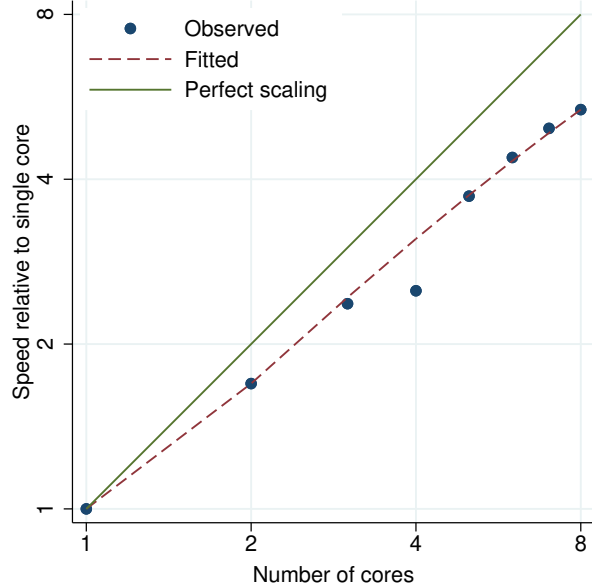


Figure 408. pwcorr performance plot.

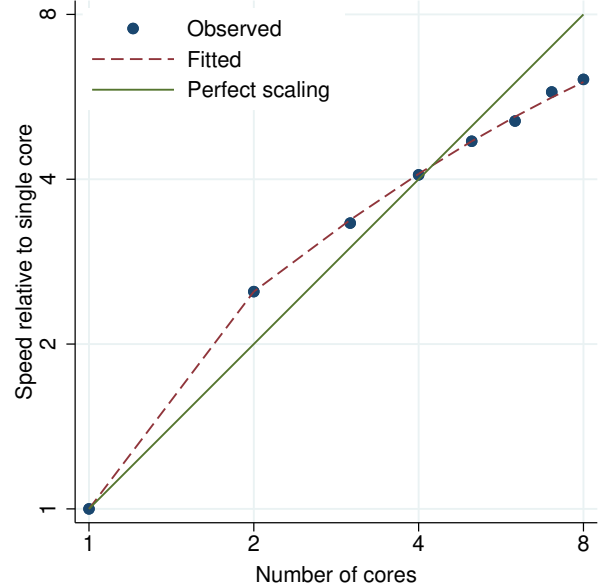


Figure 409. qreg performance plot.

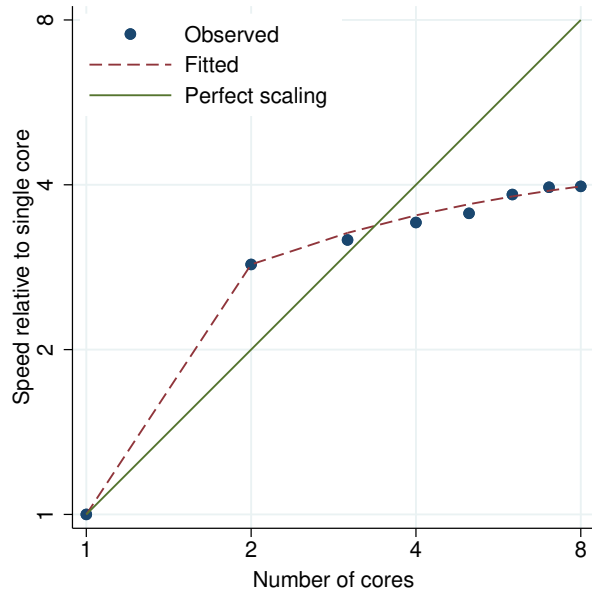


Figure 410. ranksum performance plot.

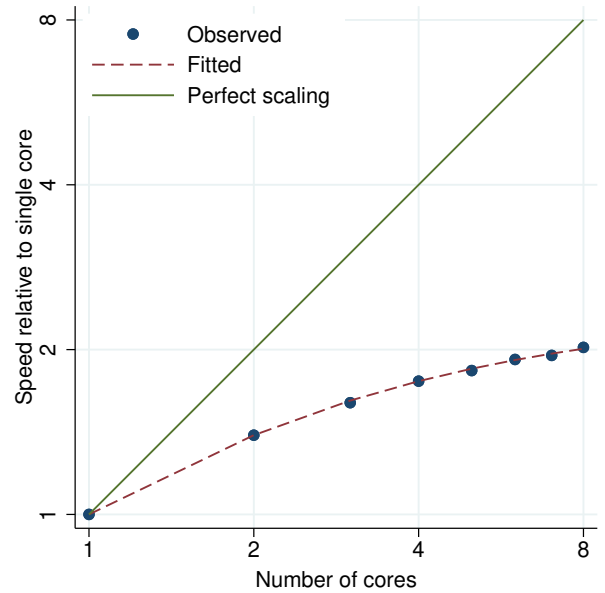


Figure 411. ratio performance plot.

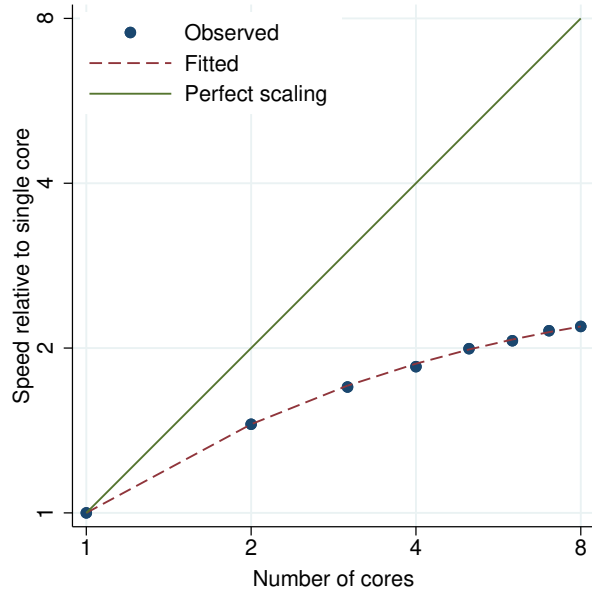


Figure 412. ratio (exp1) (exp2) performance plot.

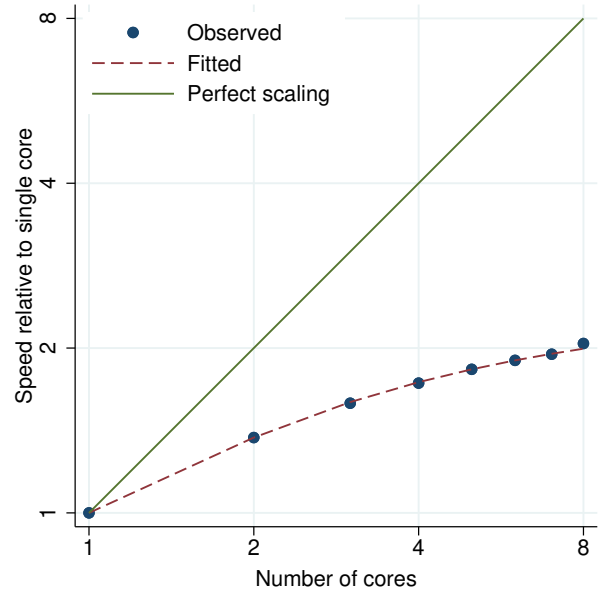


Figure 413. recode performance plot.

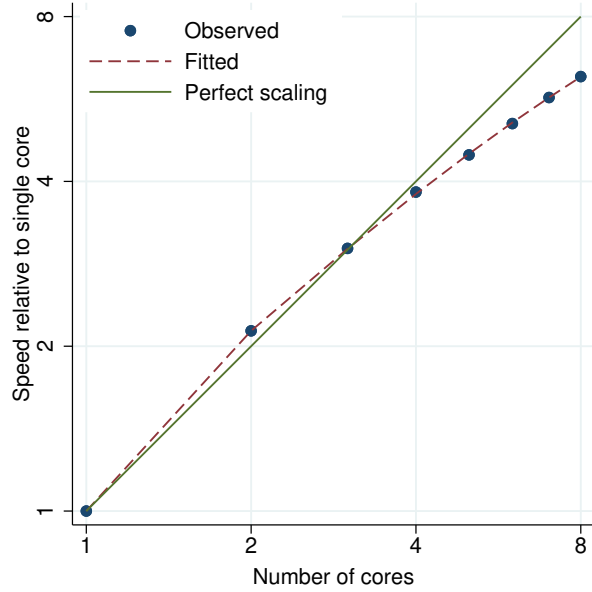


Figure 414. reg3 performance plot.

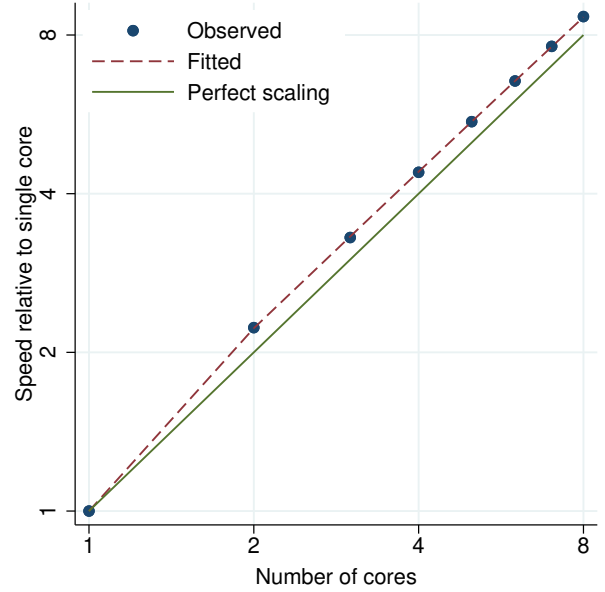


Figure 415. regress performance plot.

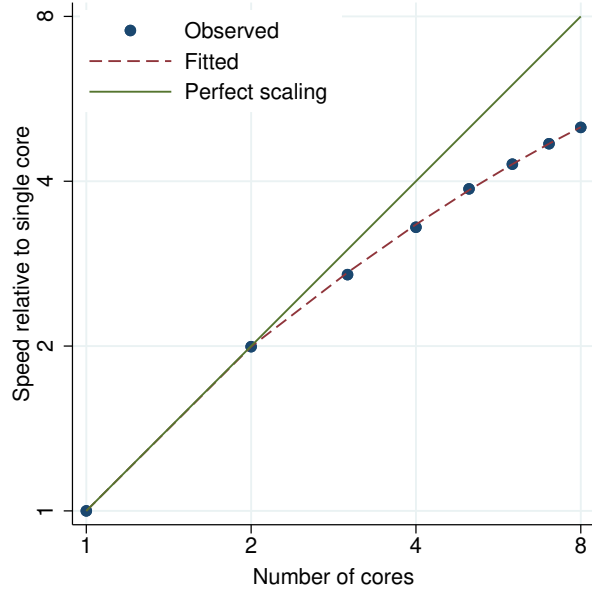


Figure 416. regress, vce(cluster) performance plot.

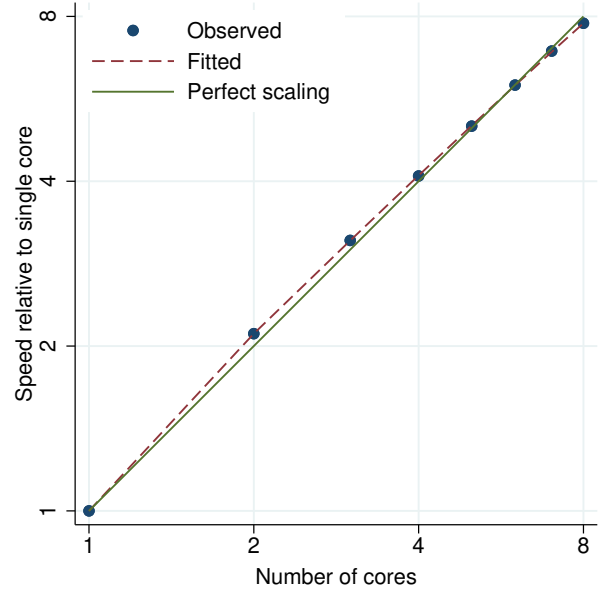


Figure 417. regress, vce(robust) performance plot.

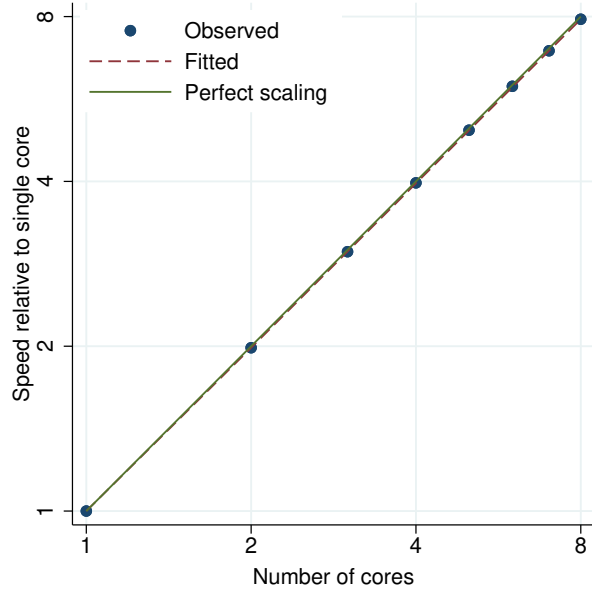


Figure 418. replace performance plot.

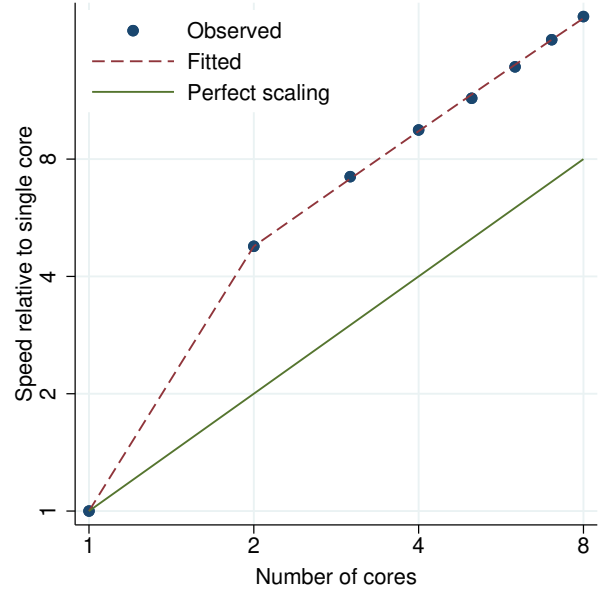


Figure 419. replace (small expressions) performance plot.

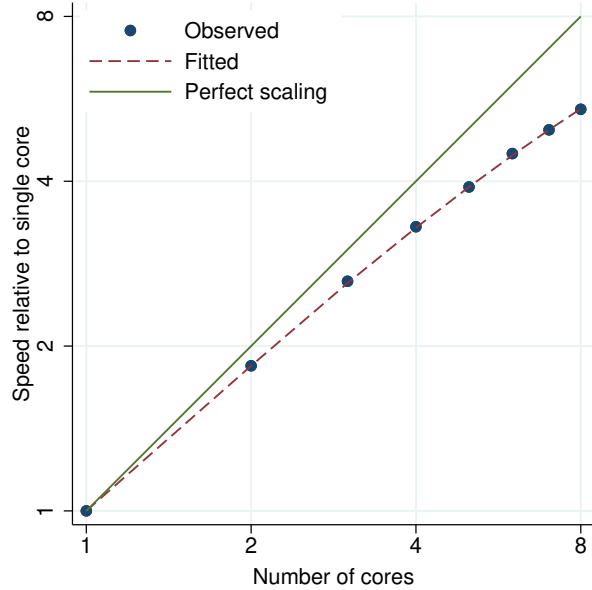


Figure 420. reri logistic performance plot.

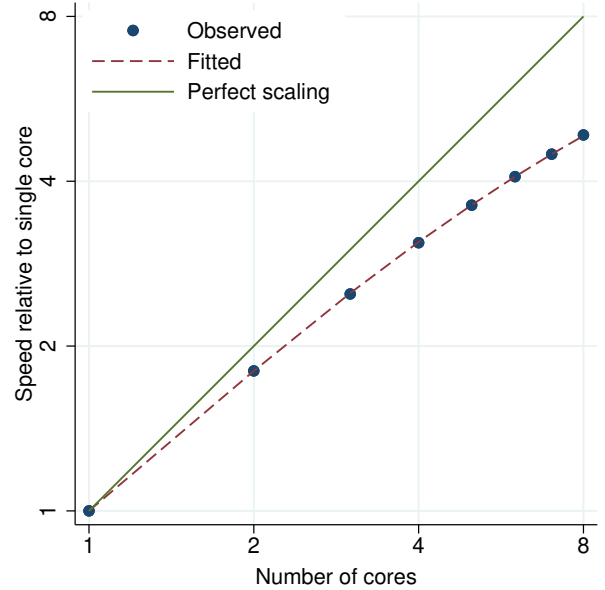


Figure 421. reri nbreg performance plot.

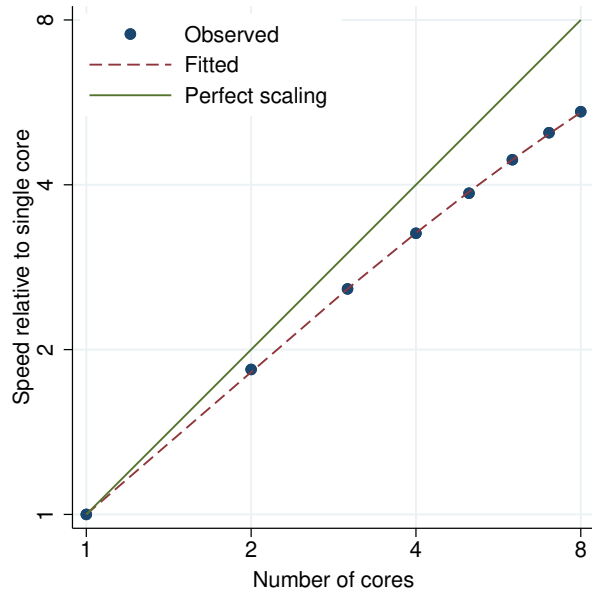


Figure 422. reri poisson performance plot.

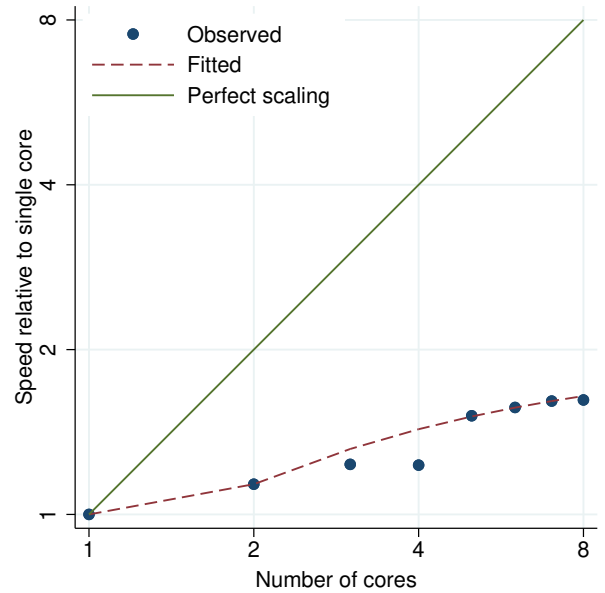


Figure 423. reri stcox performance plot.

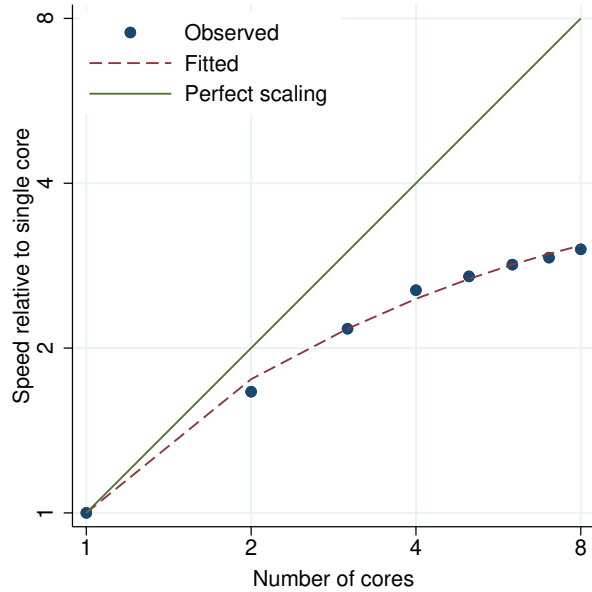


Figure 424. `reri stintcox, interval()` performance plot.

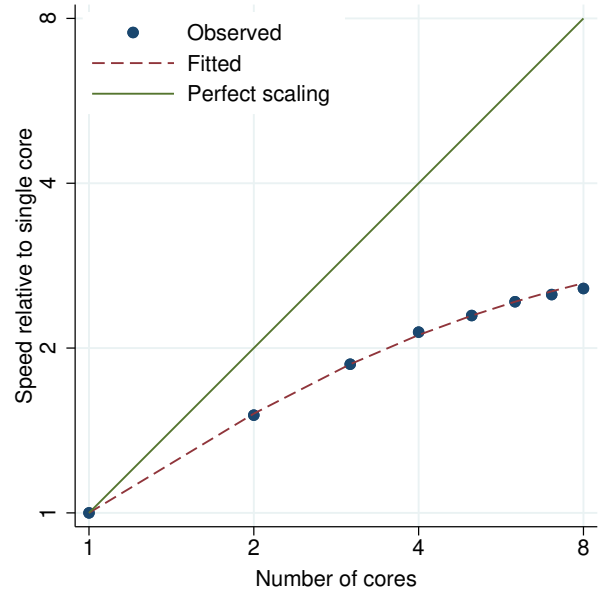


Figure 425. `reri stintcox, id() time() status()` performance plot.

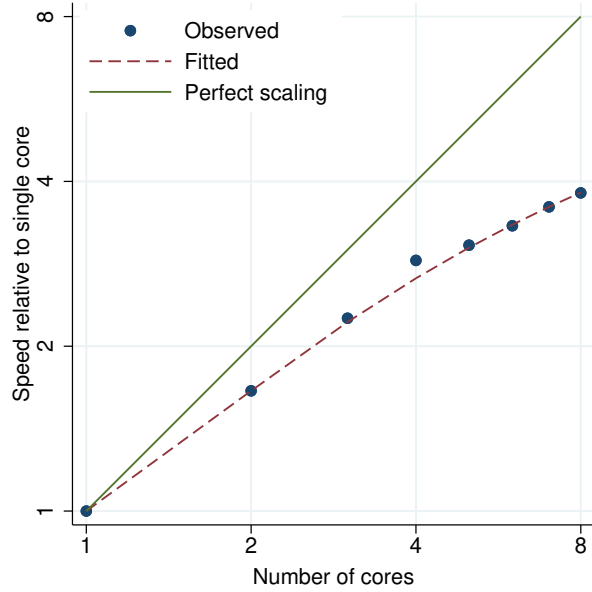


Figure 426. `reri stintreg` performance plot.

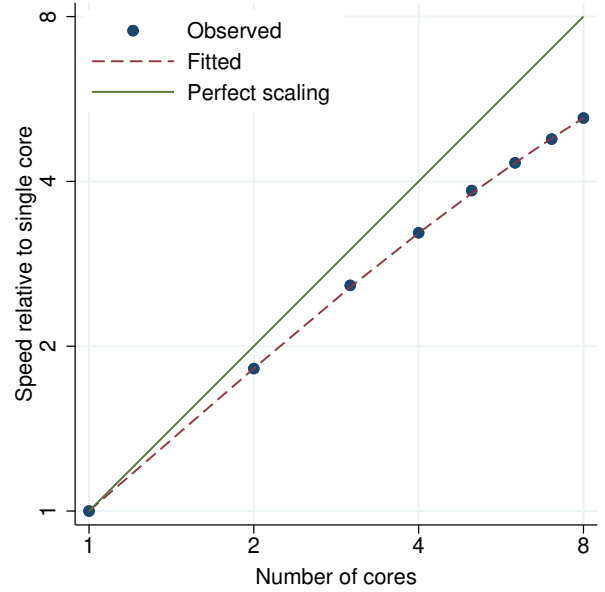


Figure 427. `reri streg` performance plot.

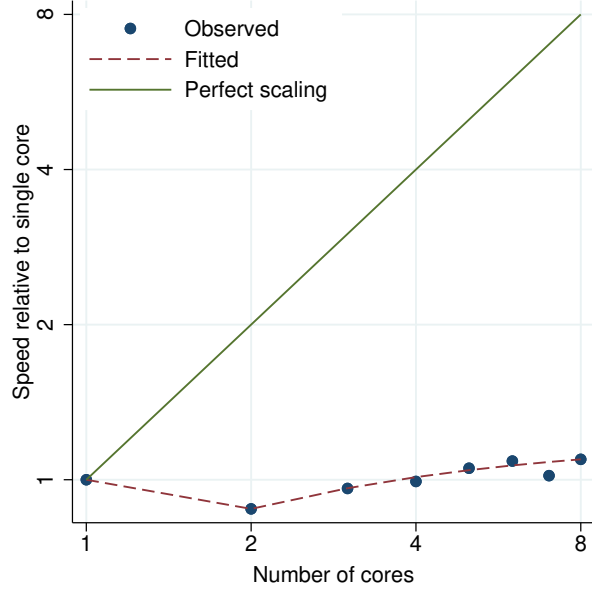


Figure 428. reshape long performance plot.

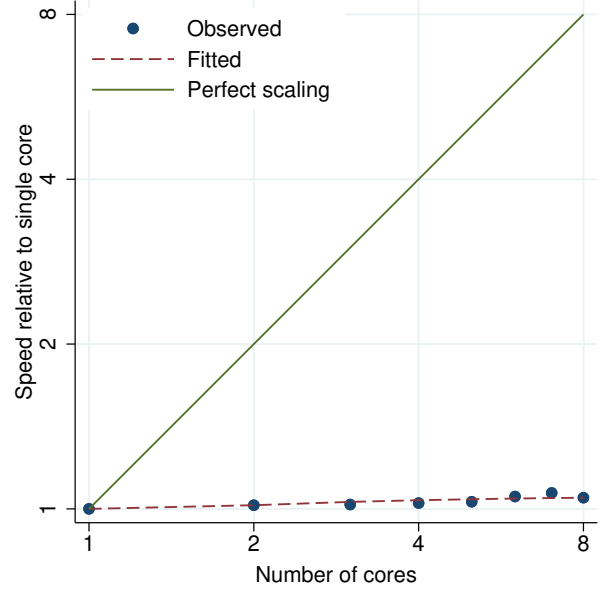


Figure 429. reshape wide performance plot.

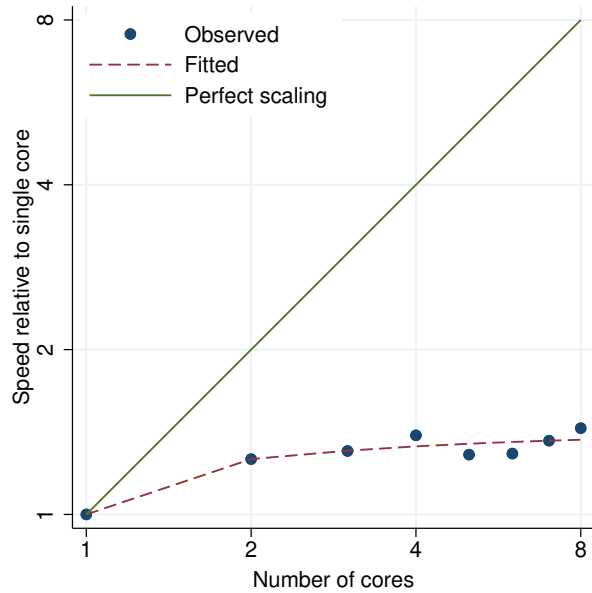


Figure 430. robvar performance plot.

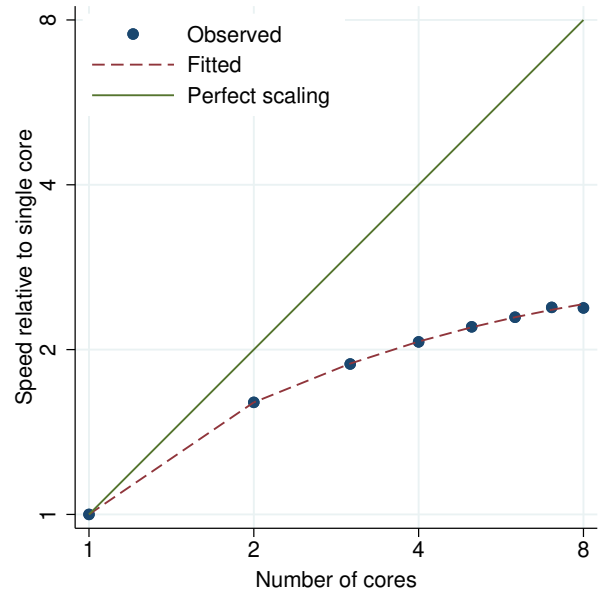


Figure 431. rocfit performance plot.

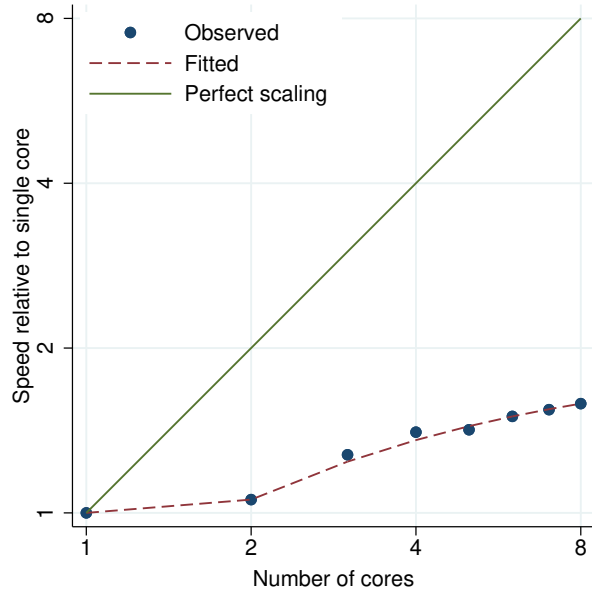


Figure 432. `roctab` performance plot.

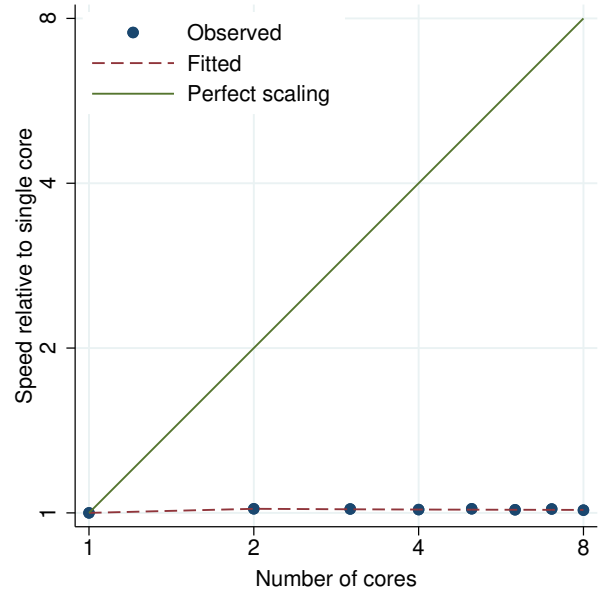


Figure 433. `rotate` performance plot.

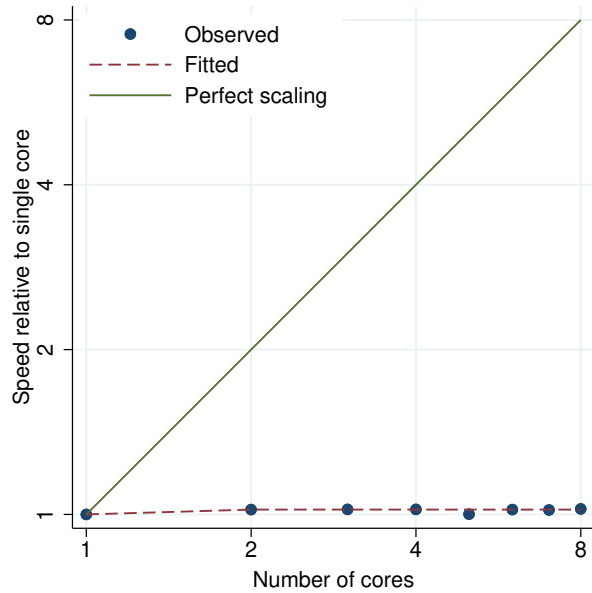


Figure 434. `rotatemat` performance plot.

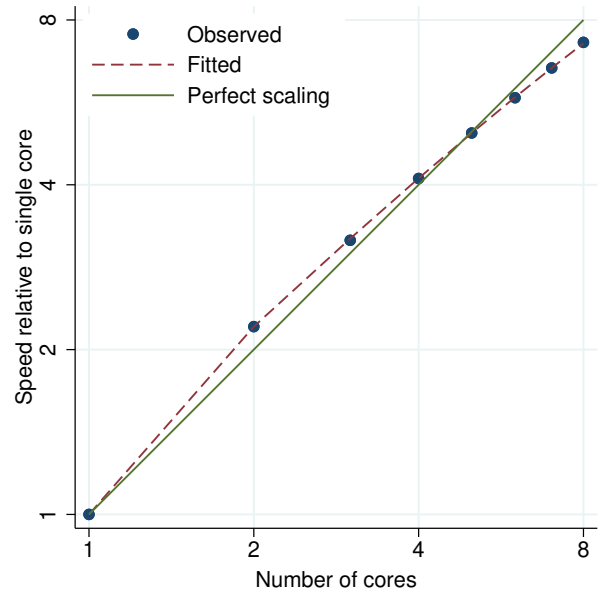


Figure 435. `rreg` performance plot.

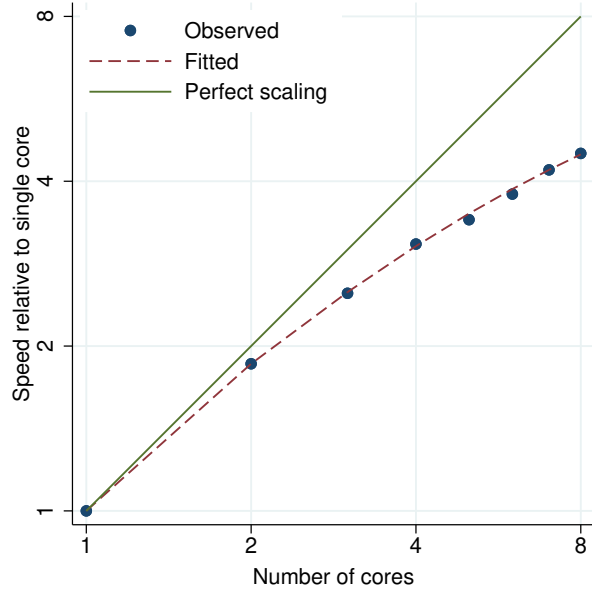


Figure 436. runtest performance plot.

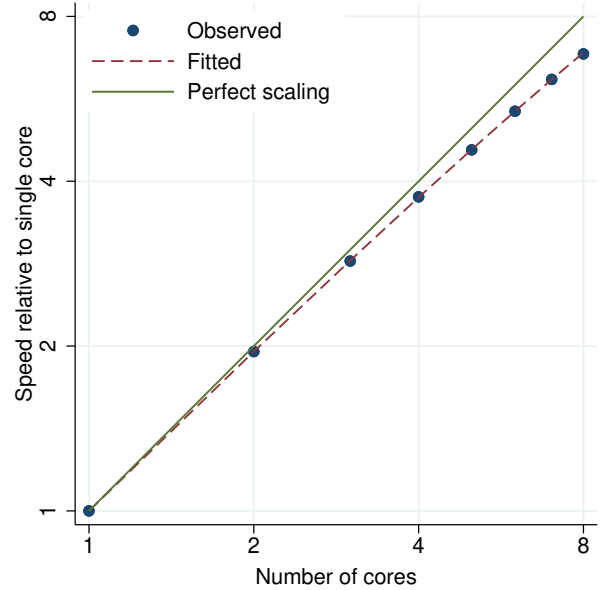


Figure 437. scobit performance plot.

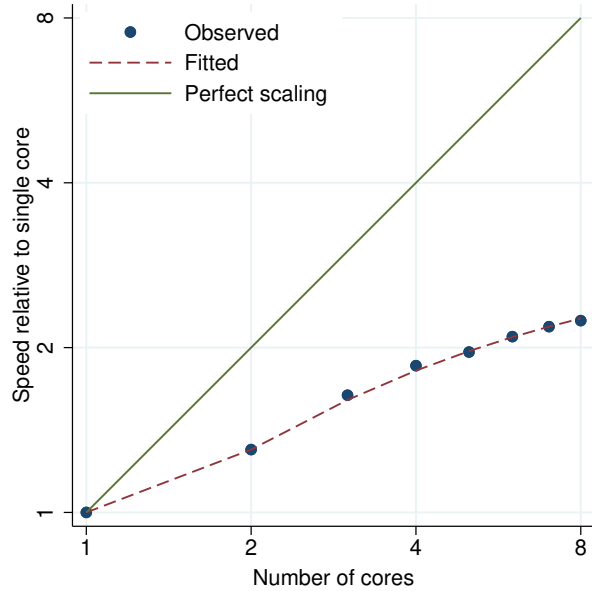


Figure 438. scoreplot performance plot.

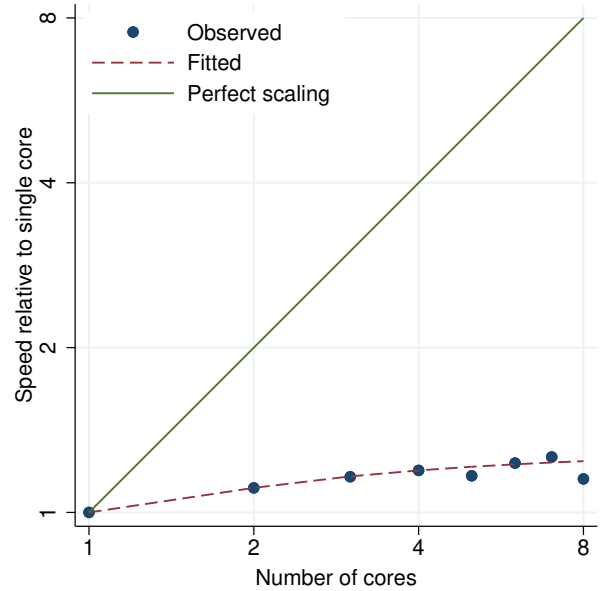


Figure 439. screeplot performance plot.

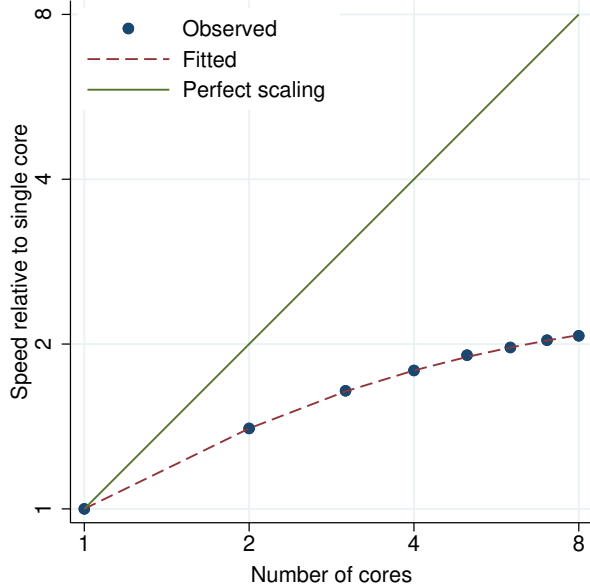


Figure 440. sdtest1 performance plot.

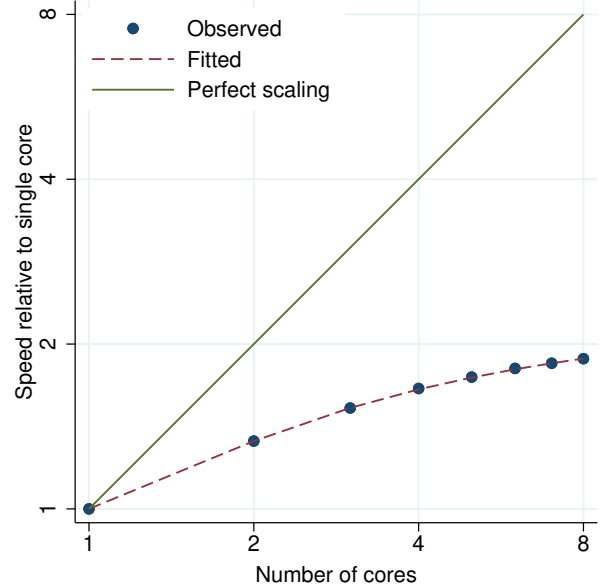


Figure 441. sdtest2 performance plot.

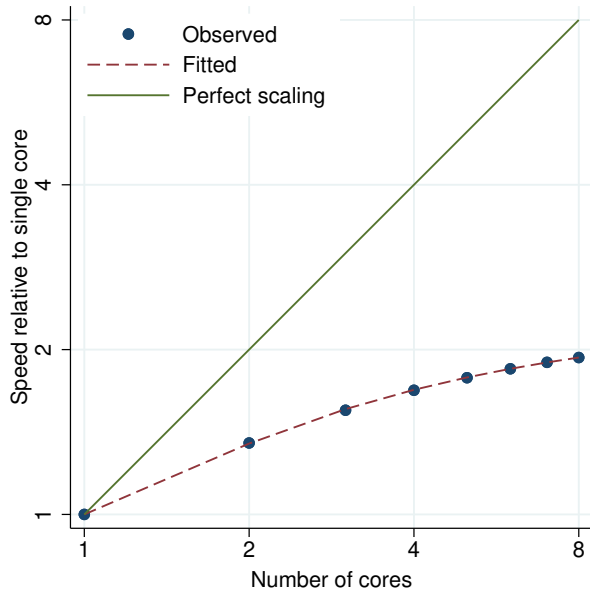


Figure 442. sdtest, by() performance plot.

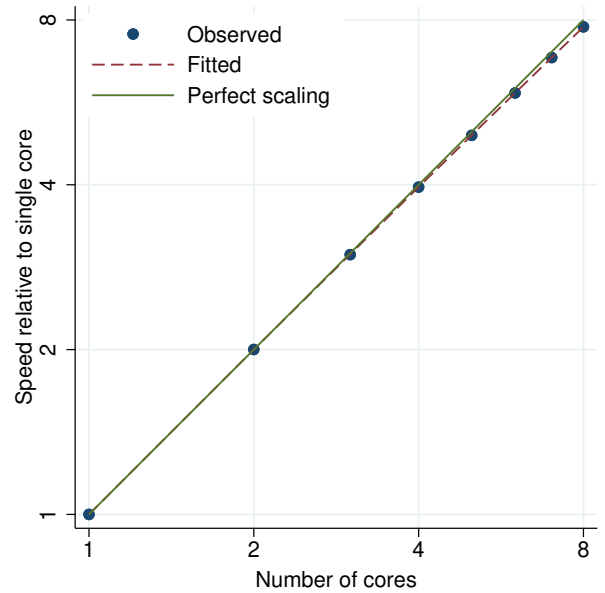


Figure 443. sem, method(adf) (CFA) performance plot.

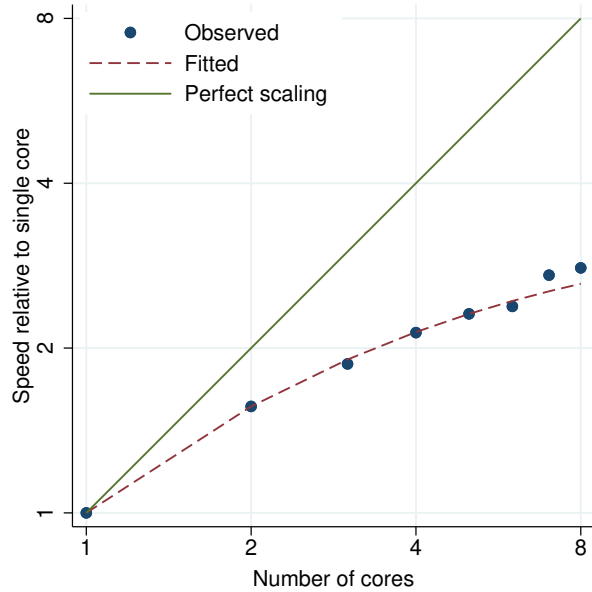


Figure 444. `sem, method(ml)` (CFA) performance plot.

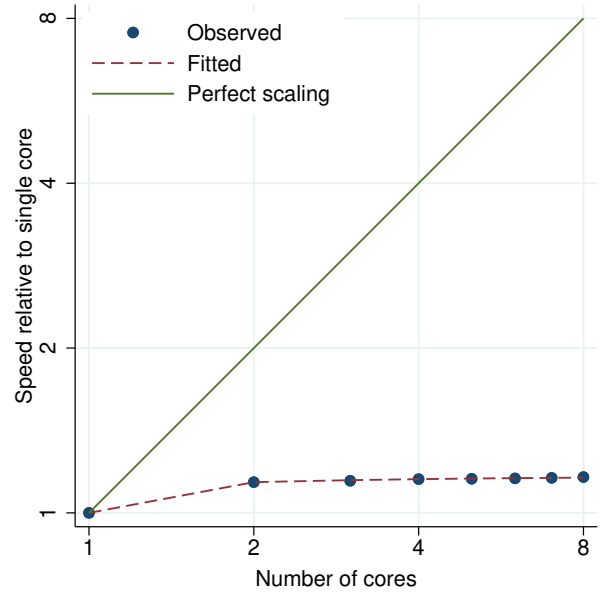


Figure 445. `sem, method(mlmv)` (CFA) performance plot.

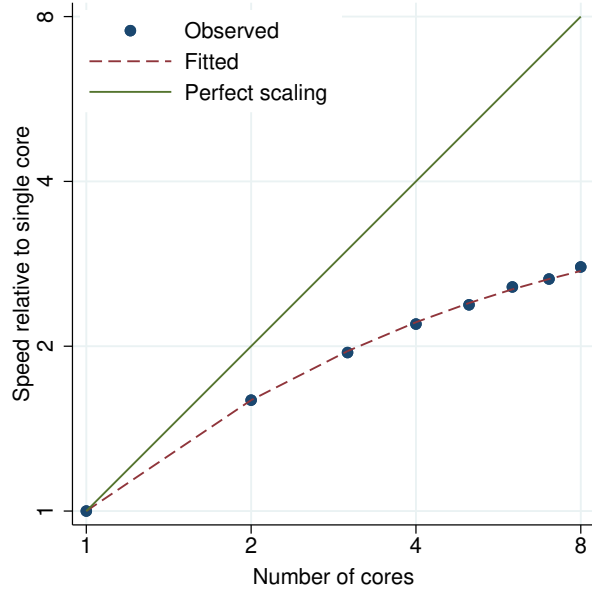


Figure 446. `sem` (SEM latent) performance plot.

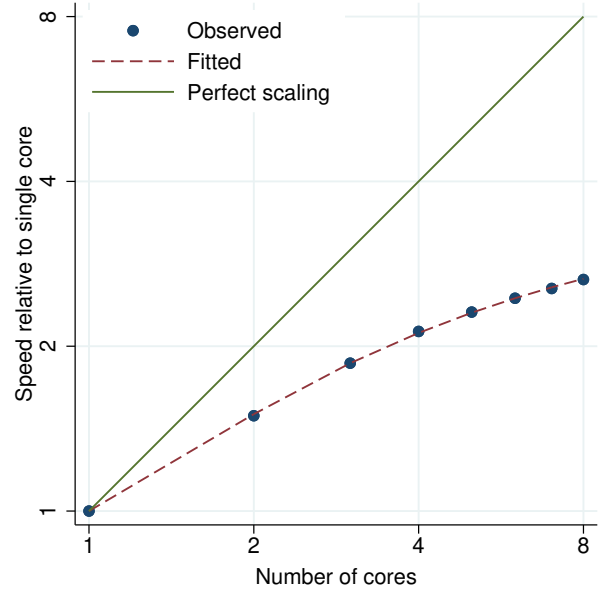


Figure 447. `sem` (SEM observed) performance plot.

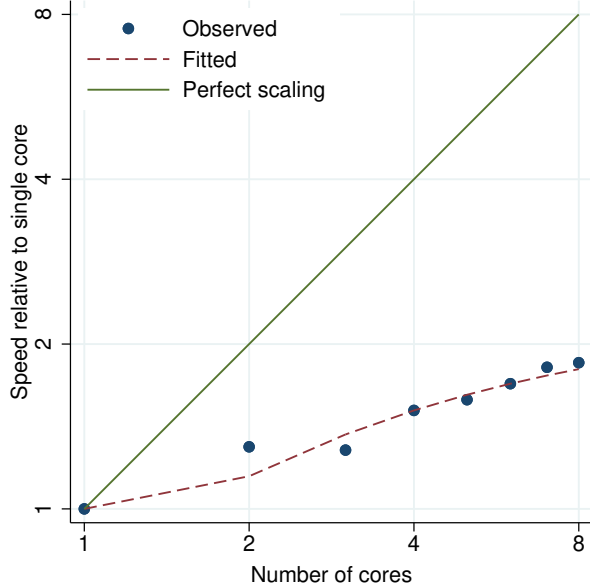


Figure 448. separate performance plot.

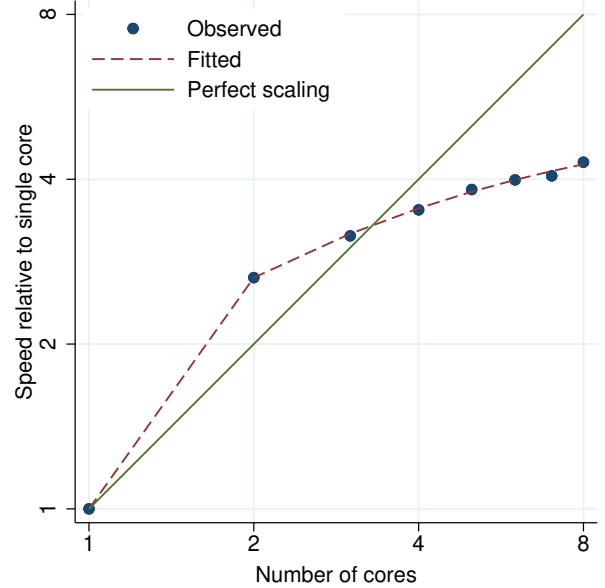


Figure 449. sfrancia performance plot.

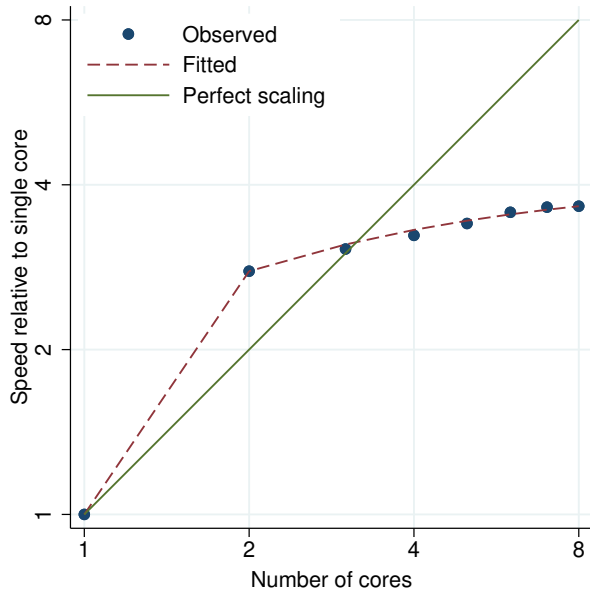


Figure 450. signrank performance plot.

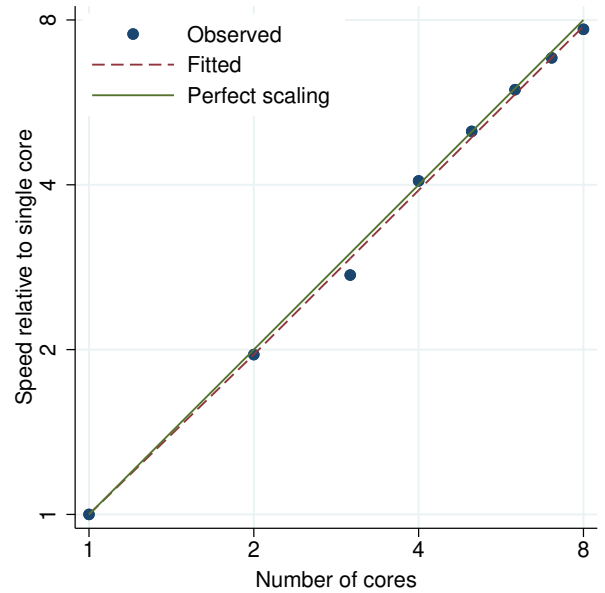


Figure 451. signtest performance plot.

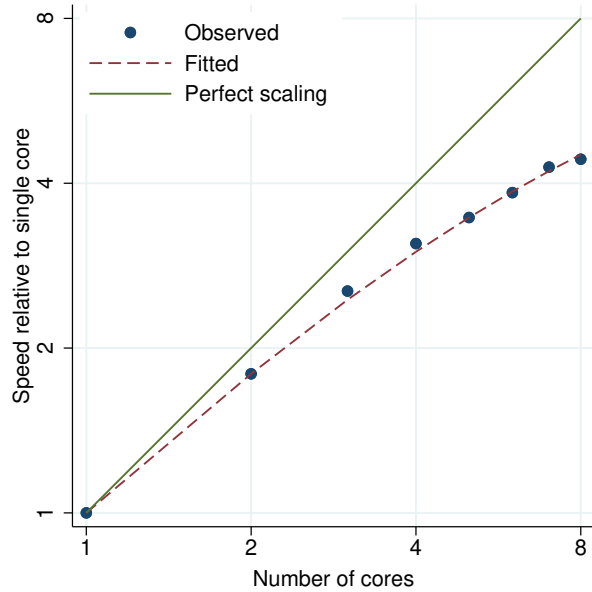


Figure 452. `sktest` performance plot.

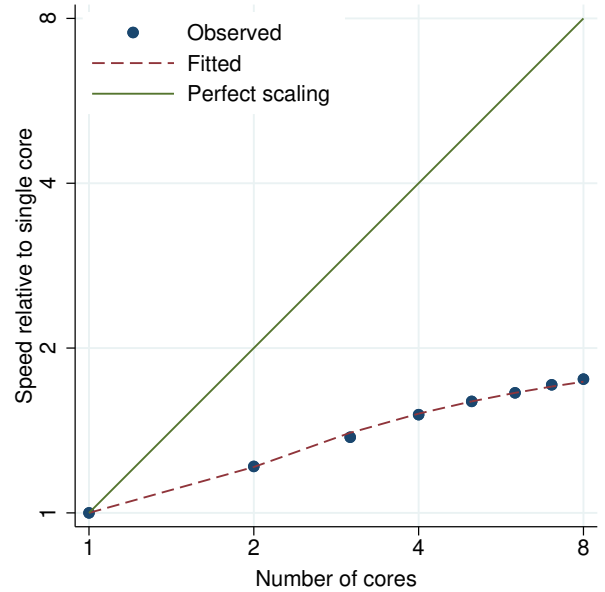


Figure 453. `slogit` performance plot.

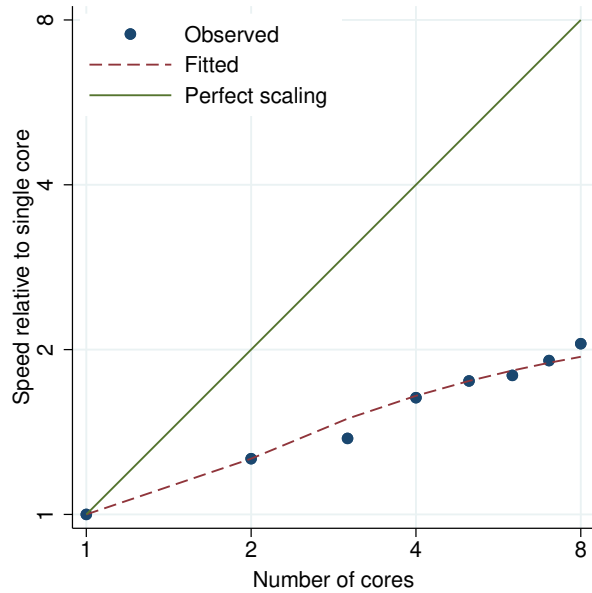


Figure 454. `sort` performance plot.

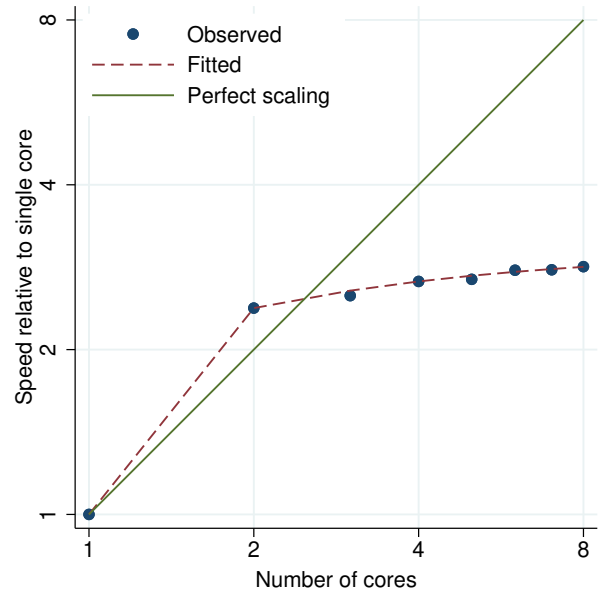


Figure 455. `spearman` performance plot.

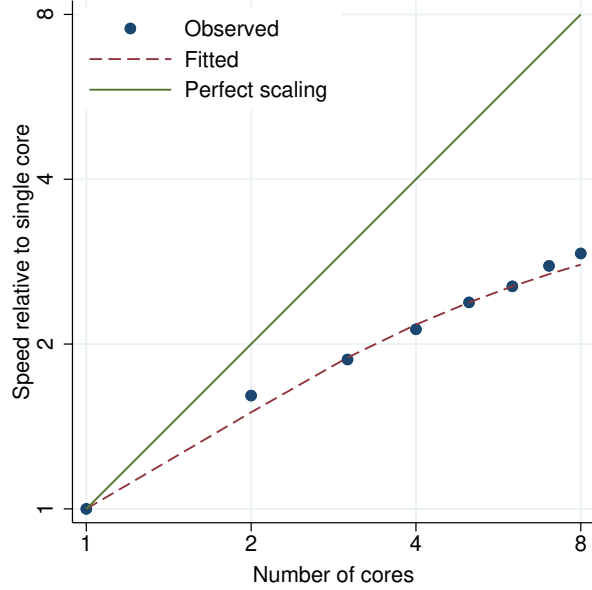


Figure 456. sspace performance plot.

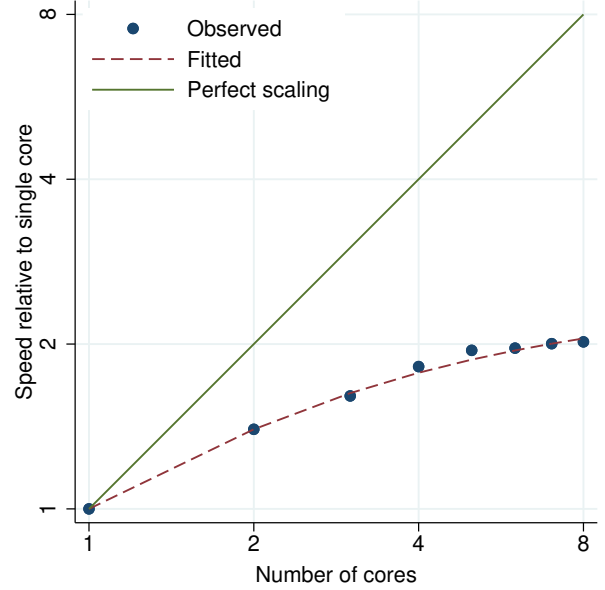


Figure 457. stack performance plot.

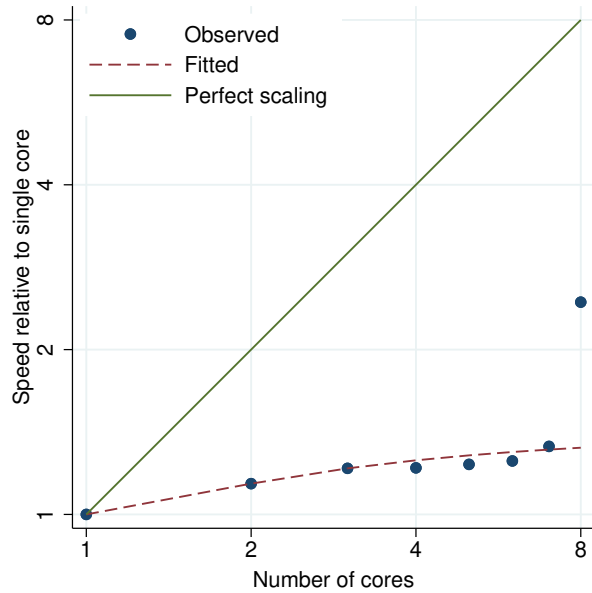


Figure 458. stci performance plot.

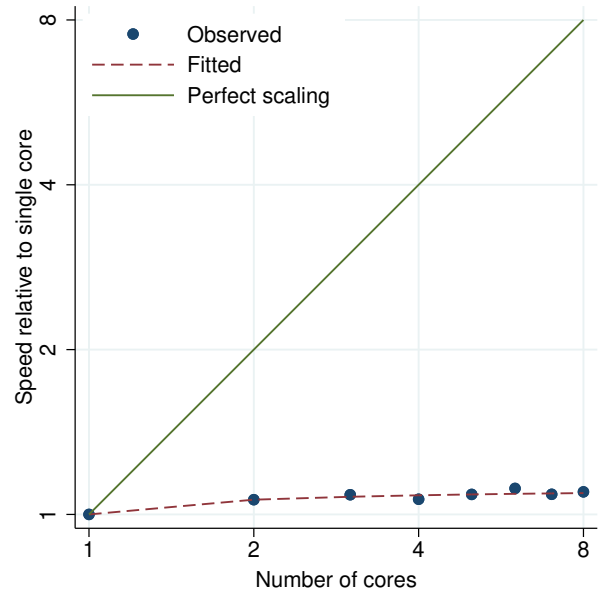


Figure 459. stcox performance plot.

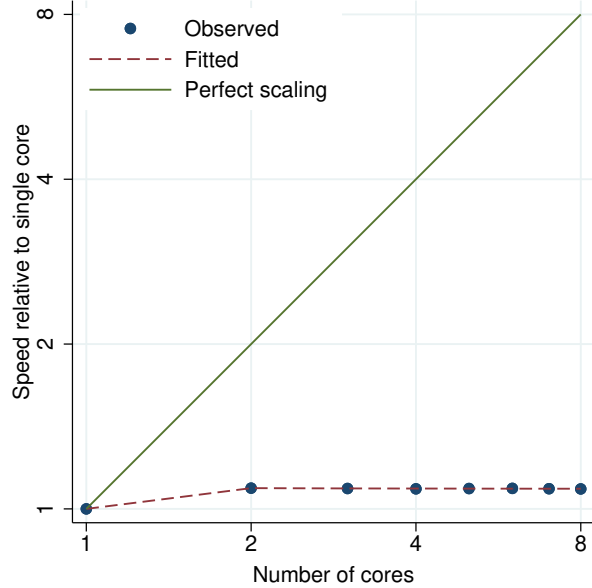


Figure 460. stcrreg performance plot.

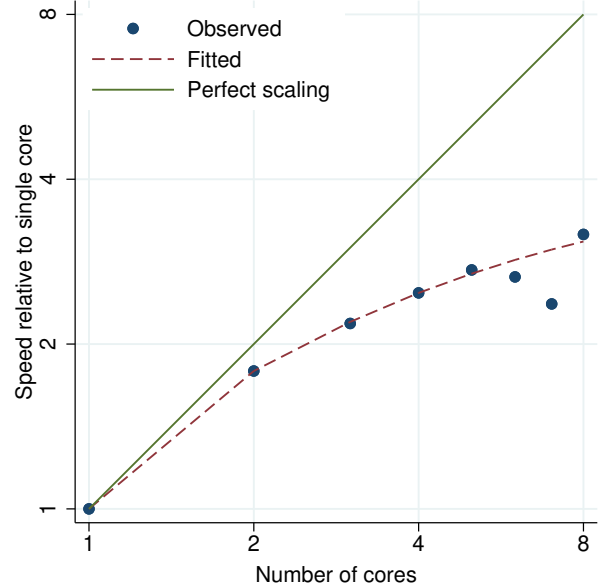


Figure 461. stgen performance plot.

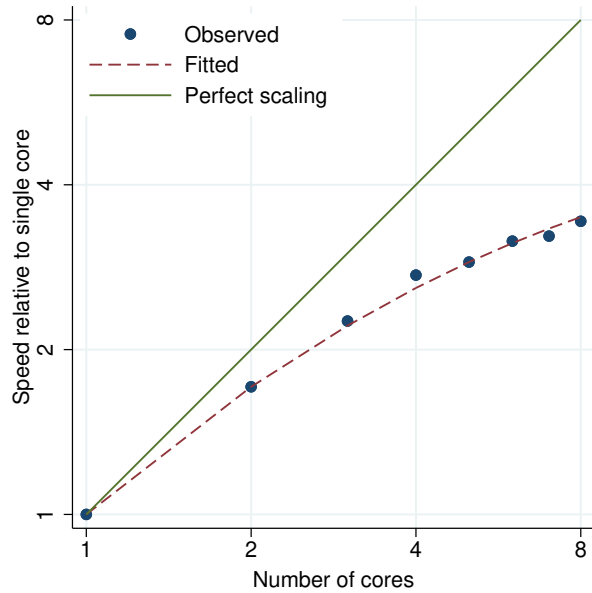


Figure 462. stintcox, interval() performance plot.

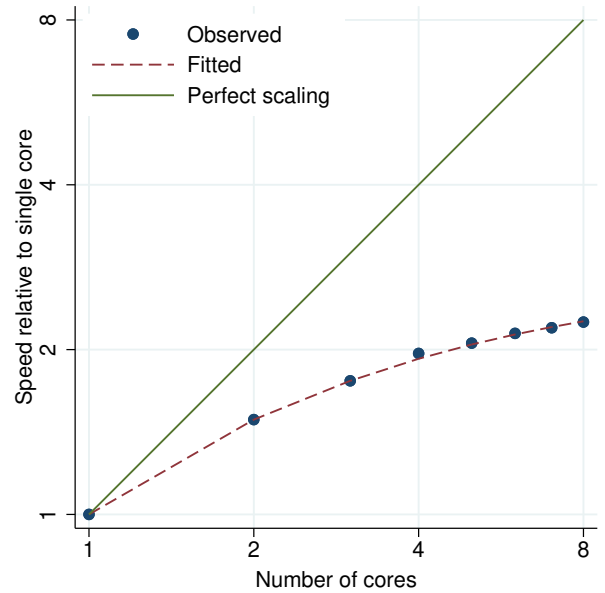


Figure 463. stintcox, id() time() status() performance plot.

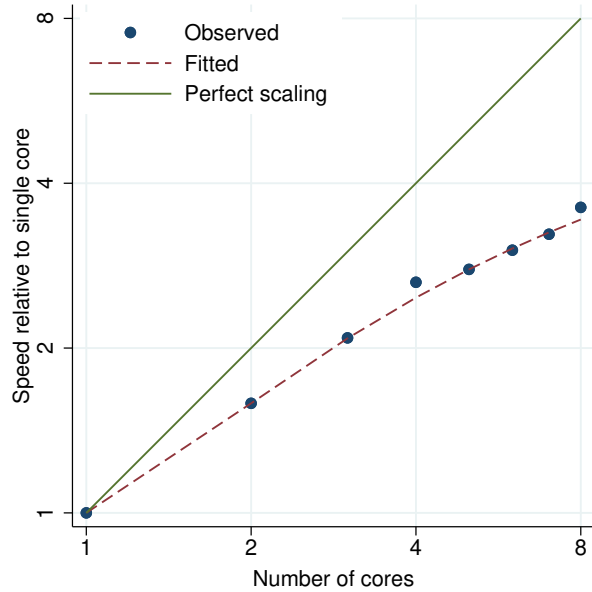


Figure 464. `stintreg, d(exponential)` performance plot.

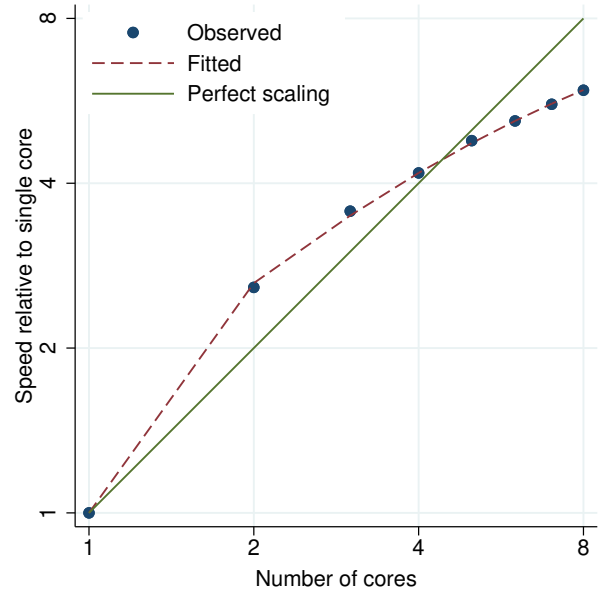


Figure 465. `stintreg, d(weibull)` performance plot.

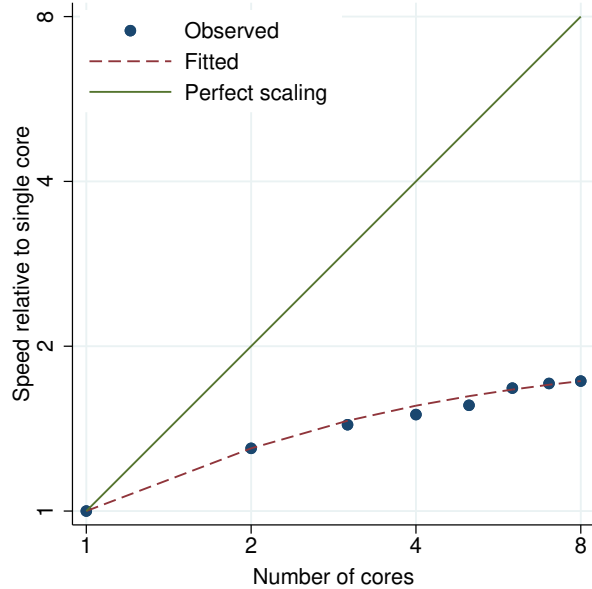


Figure 466. `stir` performance plot.

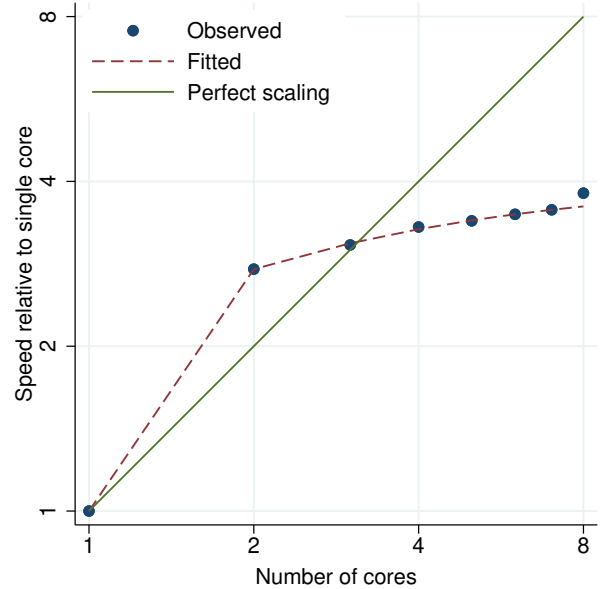


Figure 467. `stmc` performance plot.

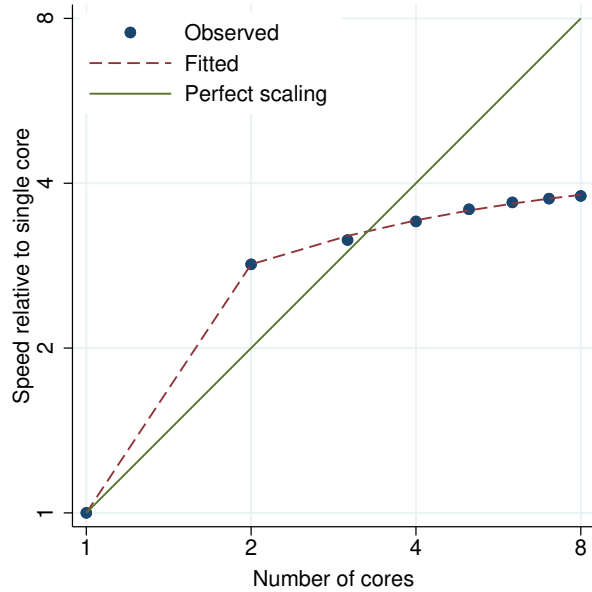


Figure 468. `by`: `stmc` performance plot.

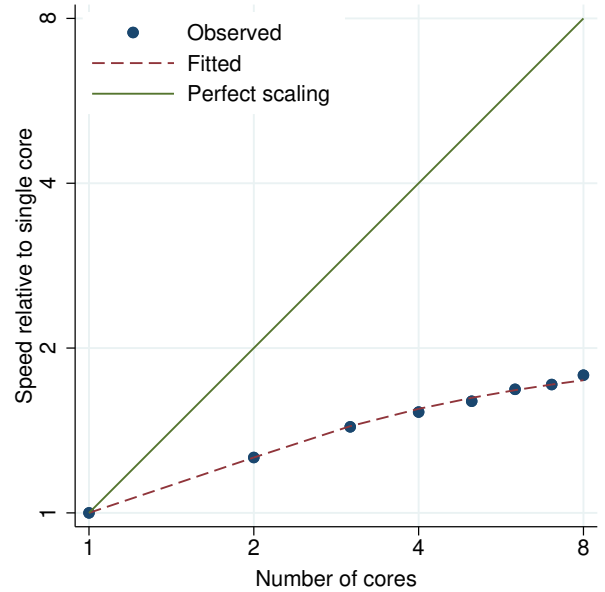


Figure 469. `stmh` performance plot.

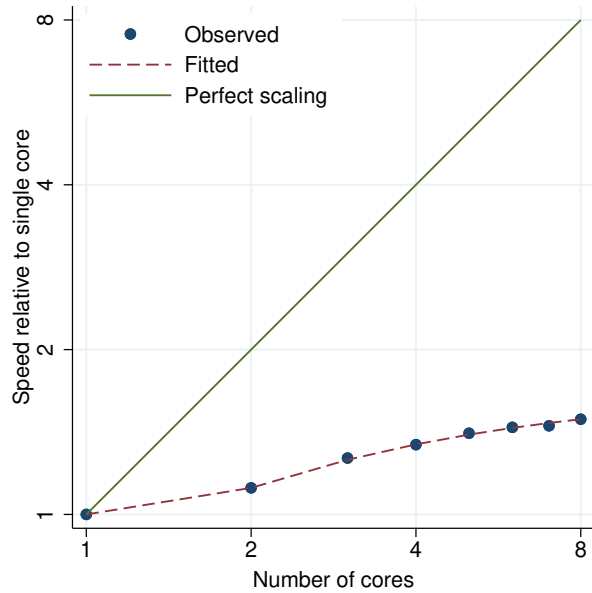


Figure 470. `by`: `stmh` performance plot.

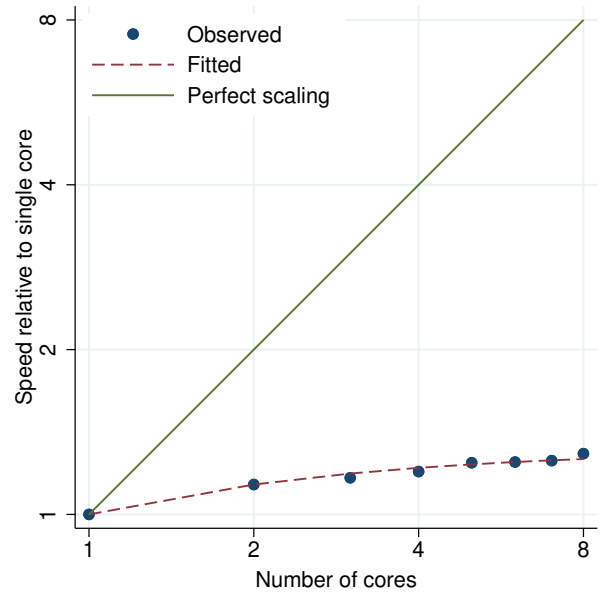


Figure 471. `stptime` performance plot.

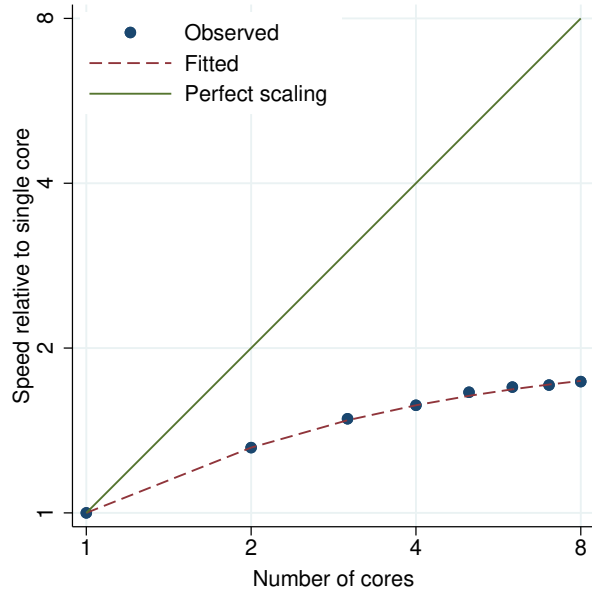


Figure 472. `strate` performance plot.

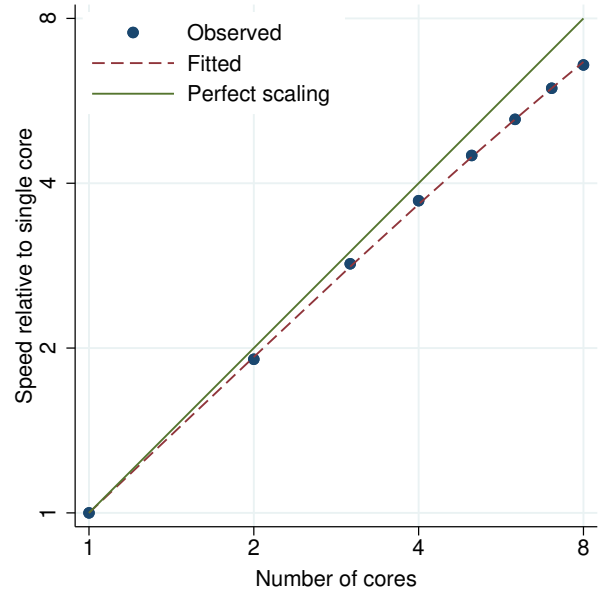


Figure 473. `streg, distribution(exponential)` performance plot.

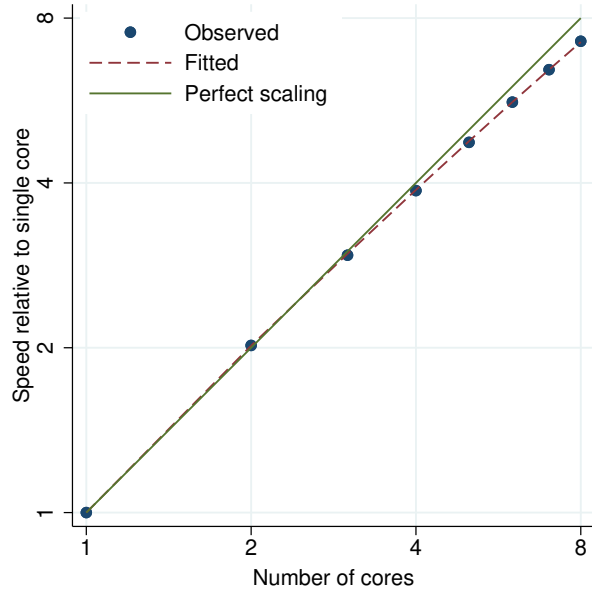


Figure 474. `streg, dist(exp) vce(cluster)` performance plot.

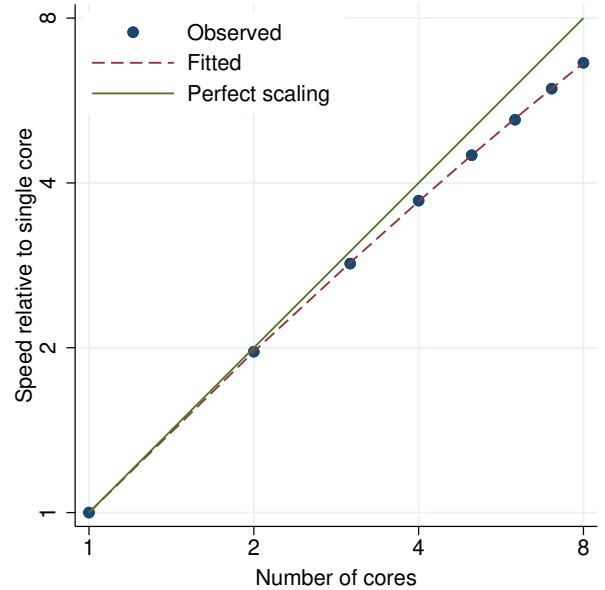


Figure 475. `streg, dist(exp) frailty()` performance plot.

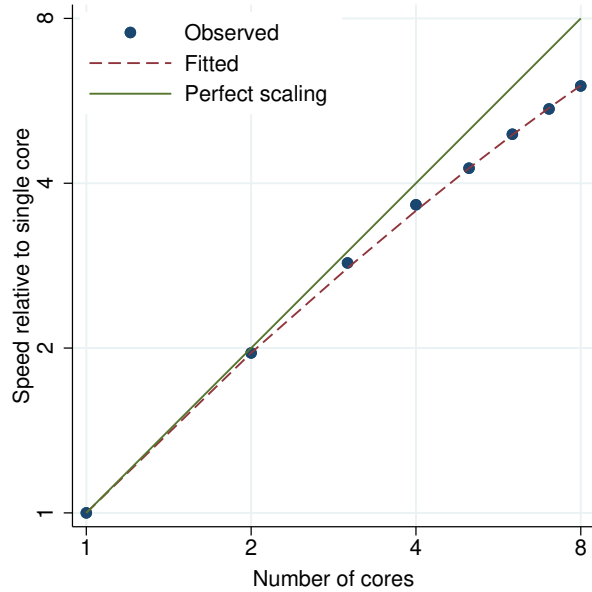


Figure 476. `streg, dist(exp) frailty() shared()` performance plot.

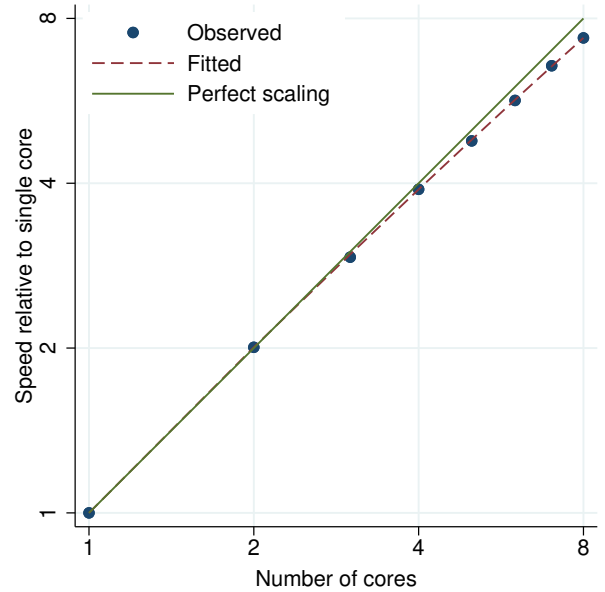


Figure 477. `streg, dist(exp) vce(robust)` performance plot.

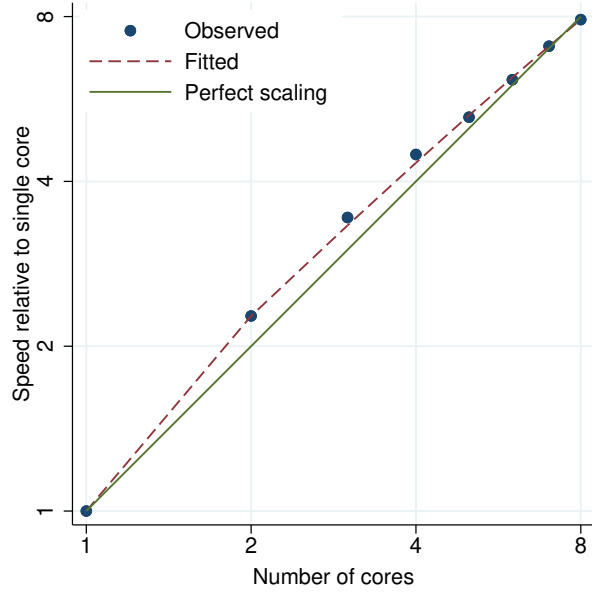


Figure 478. `streg, distribution(gamma)` performance plot.

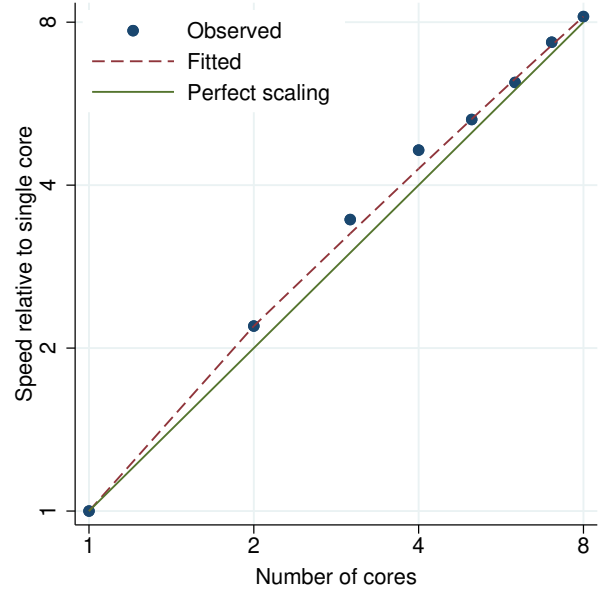


Figure 479. `streg, dist(gamma) vce(cluster)` performance plot.

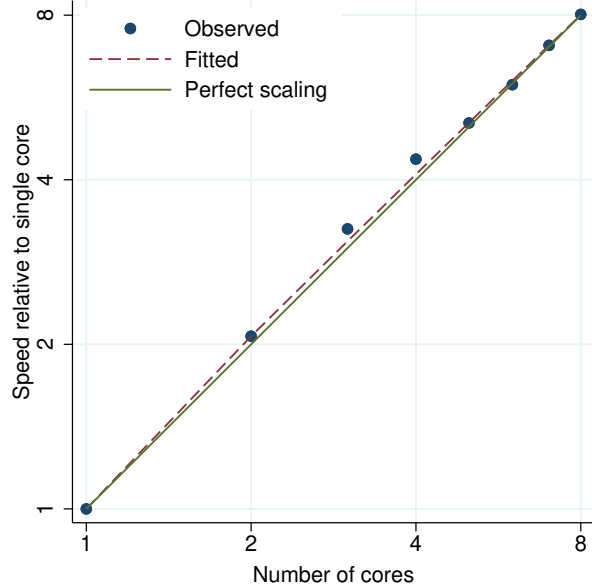


Figure 480. `streg, dist(ggamma) vce(robust)` performance plot.

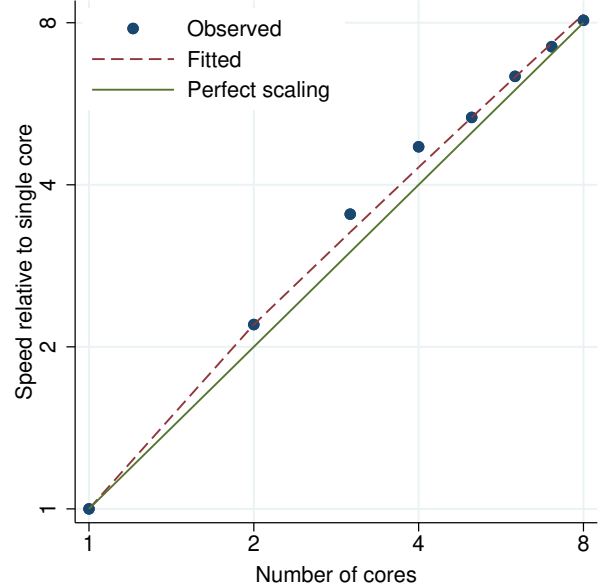


Figure 481. `streg, distribution(gompertz)` performance plot.

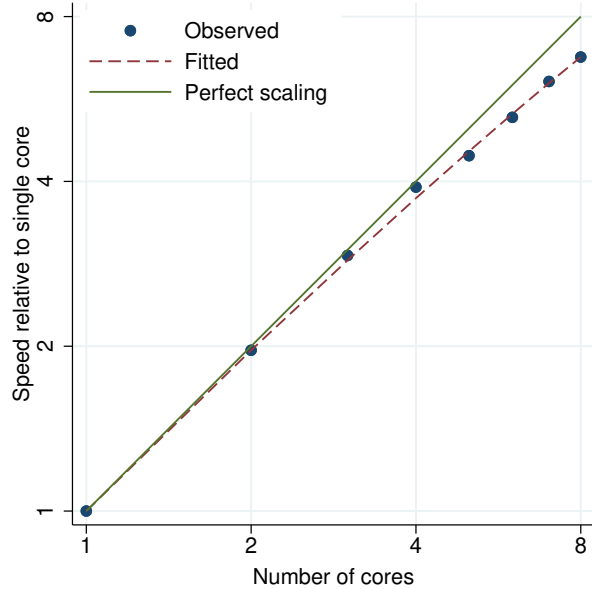


Figure 482. `streg, dist(gompertz) vce(cluster)` performance plot.

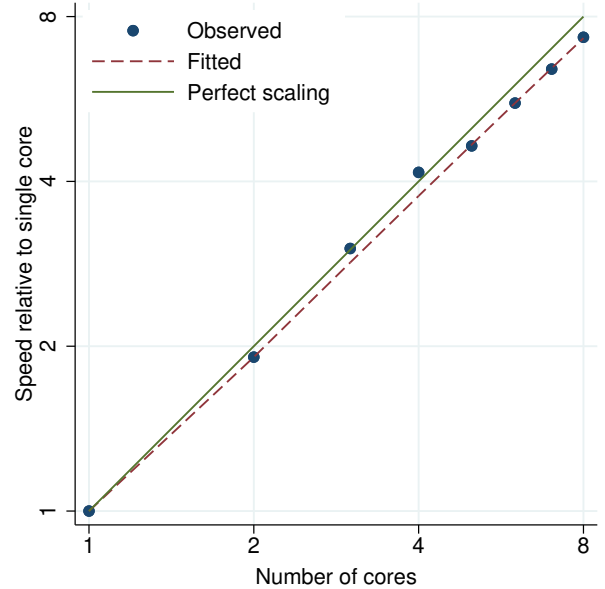


Figure 483. `streg, dist(gompertz) frailty()` performance plot.

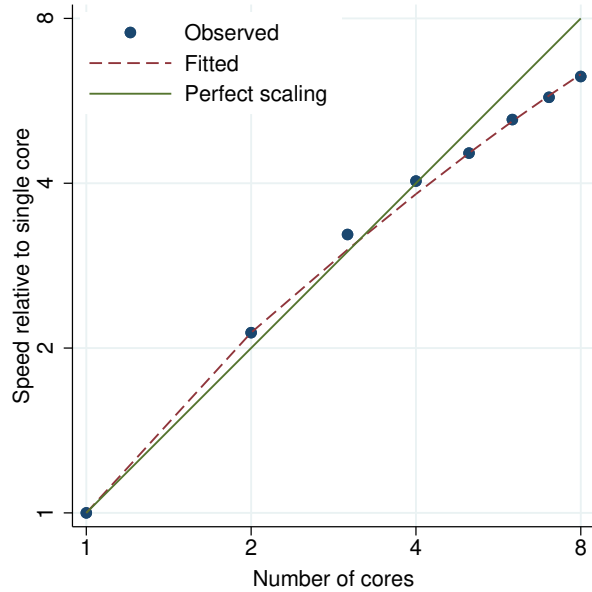


Figure 484. `streg, dist(gomp) frailty() shared()` performance plot.

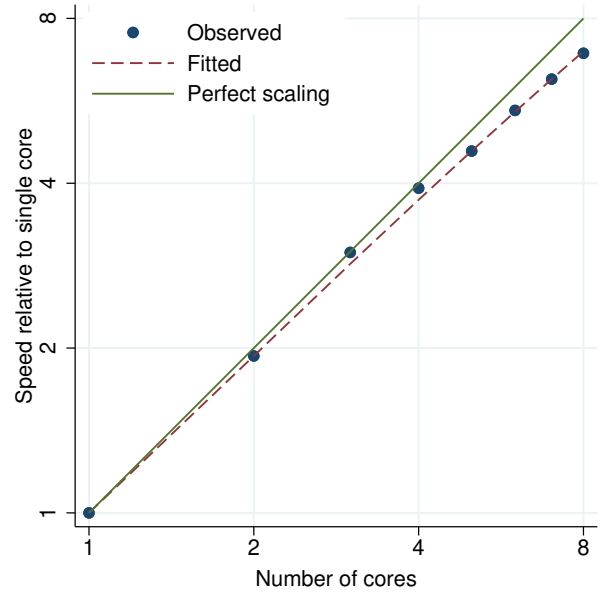


Figure 485. `streg, dist(gompertz) vce(robust)` performance plot.

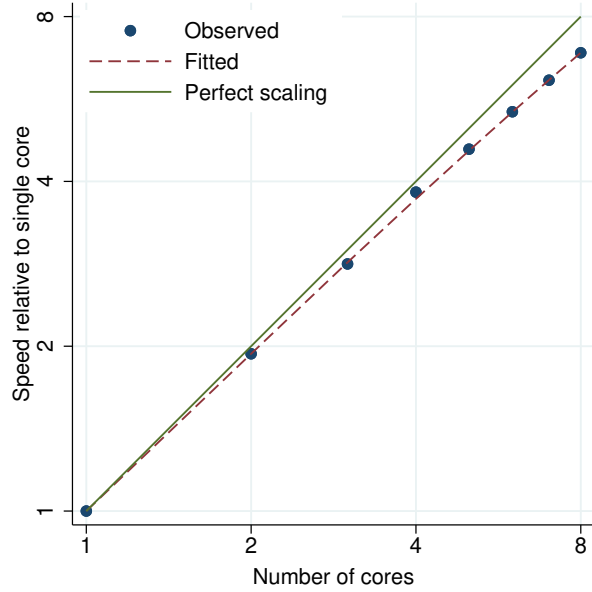


Figure 486. `streg, distribution(llogistic)` performance plot.

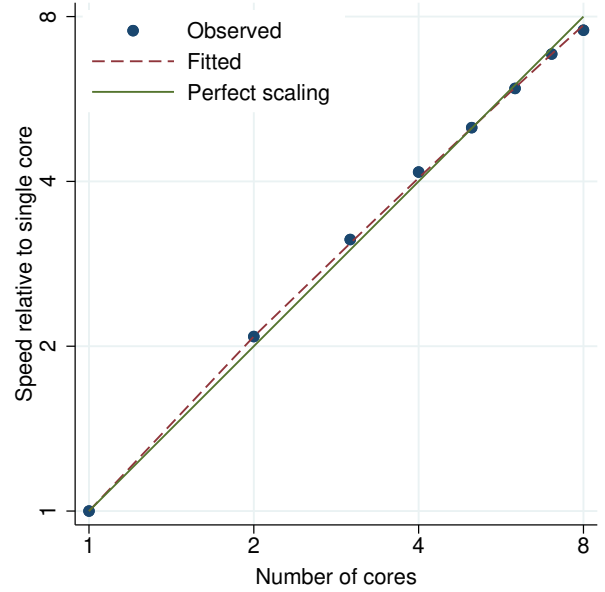


Figure 487. `streg, dist(llogistic) vce(cluster)` performance plot.

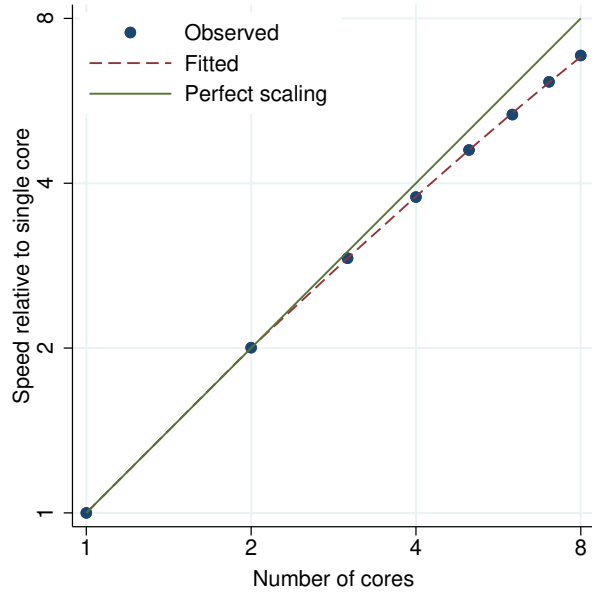


Figure 488. `streg, dist(llogistic) frailty()` performance plot.

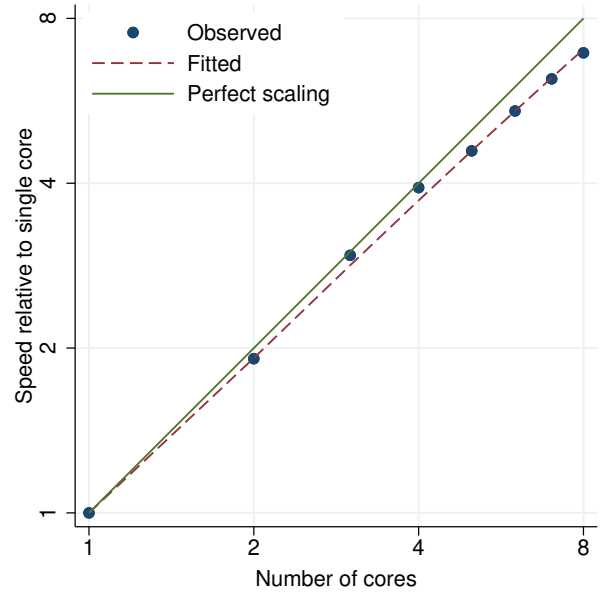


Figure 489. `streg, dist(llog) frailty() shared()` performance plot.

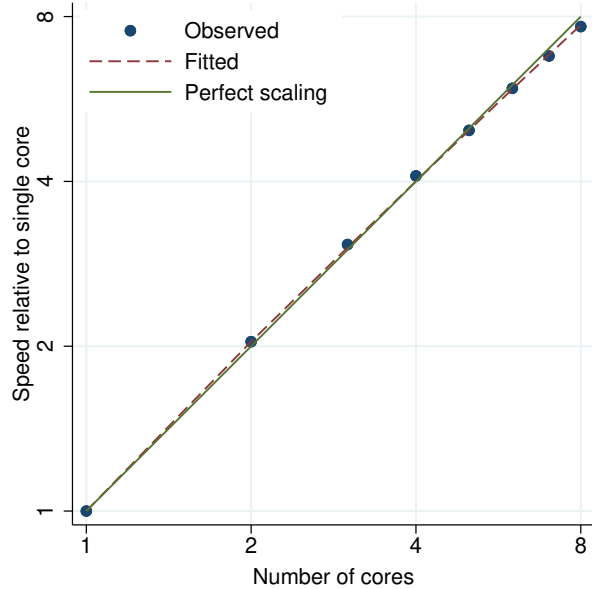


Figure 490. `streg, dist(llogistic) vce(robust)` performance plot.

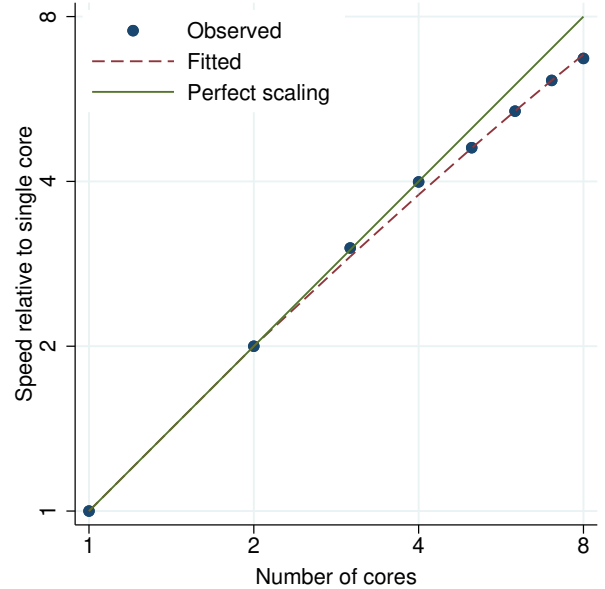


Figure 491. `streg, distribution(lnormal)` performance plot.

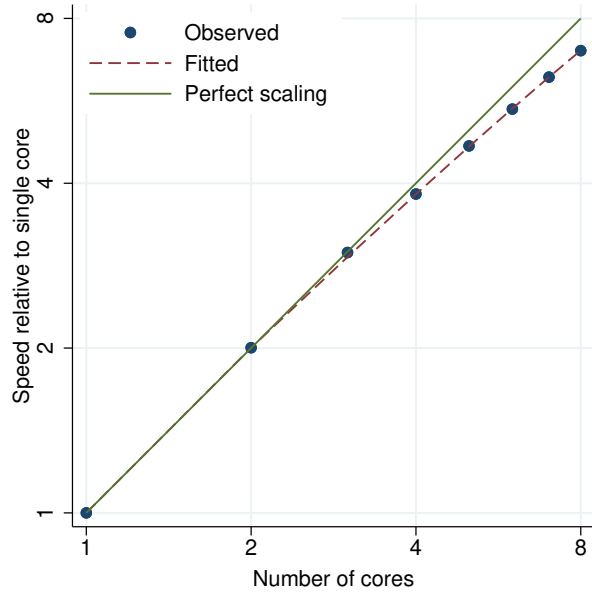


Figure 492. `streg, dist(lnormal) vce(cluster)` performance plot.

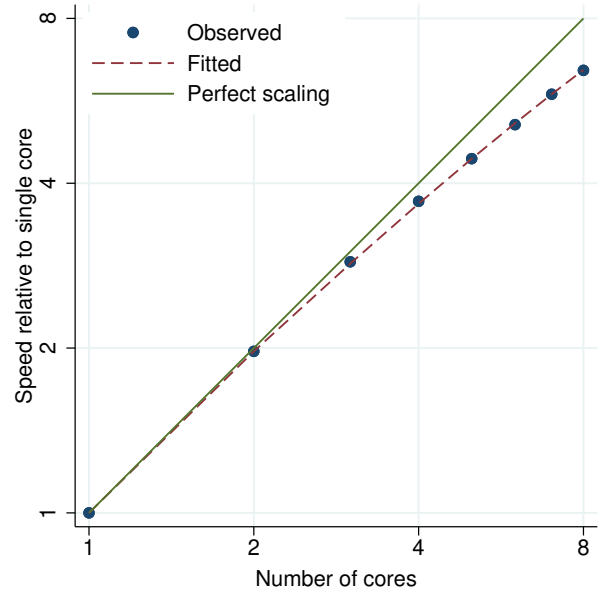


Figure 493. `streg, dist(lnormal) frailty()` performance plot.

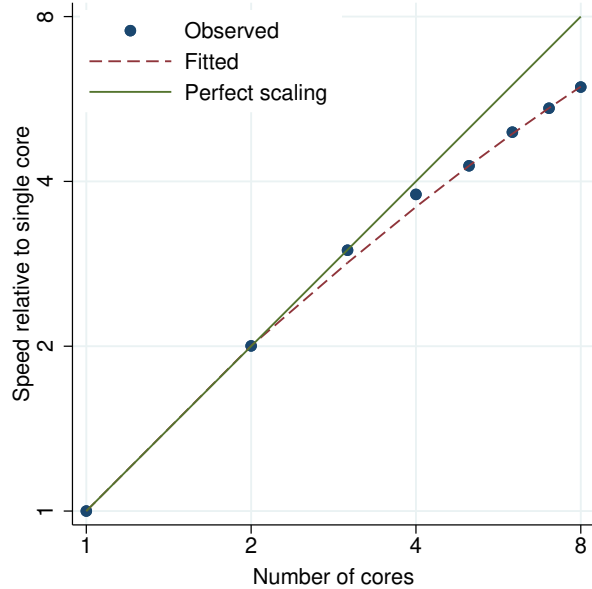


Figure 494. `streg, dist(lnorm) frailty() shared()` performance plot.

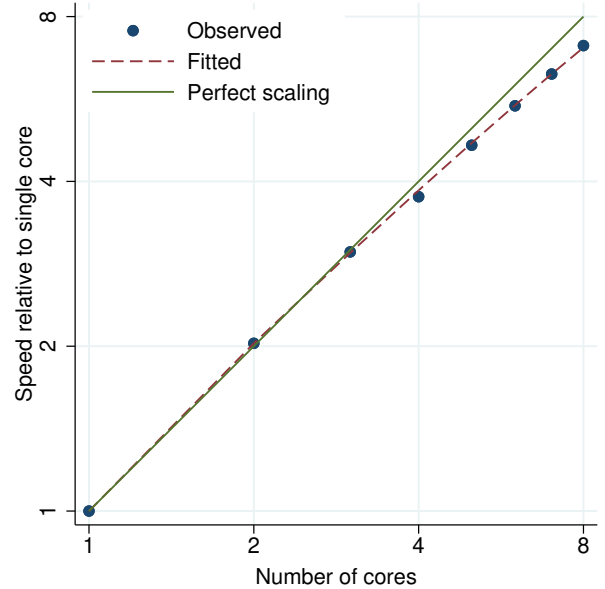


Figure 495. `streg, dist(lnormal) vce(robust)` performance plot.

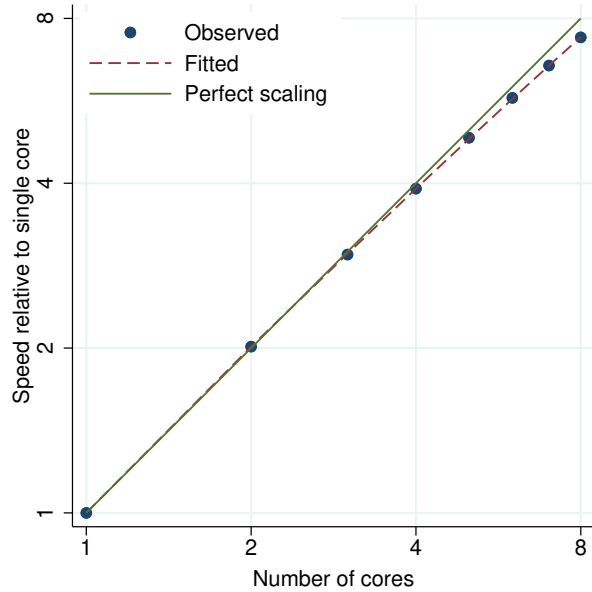


Figure 496. `streg, distribution(weibull)` performance plot.

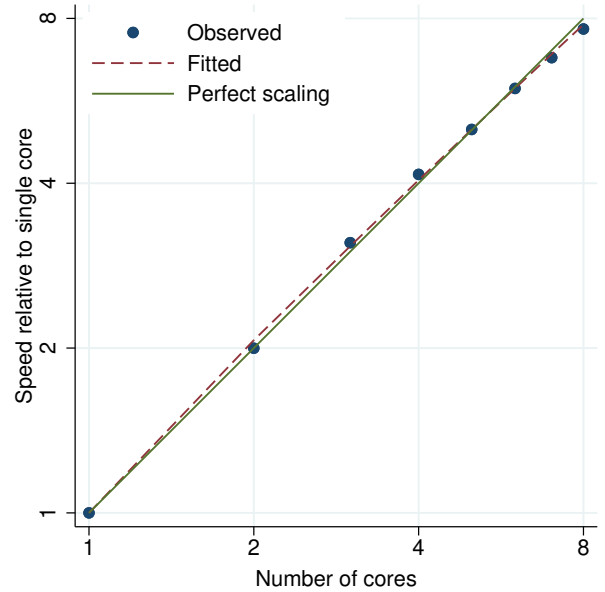


Figure 497. `streg, dist(weibull) vce(cluster)` performance plot.

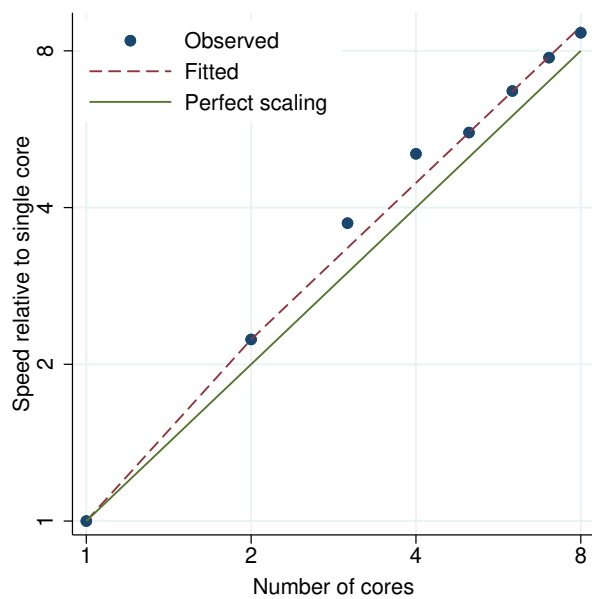


Figure 498. `streg, dist(weibull) frailty()` performance plot.

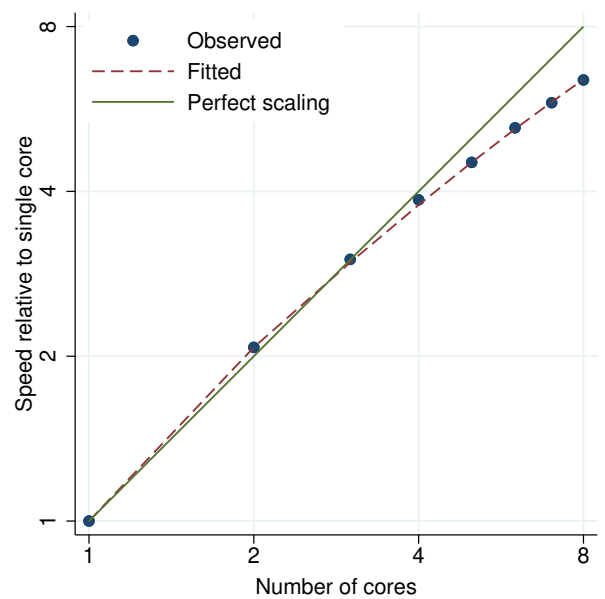


Figure 499. `streg, dist(weib) frailty() shared()` performance plot.

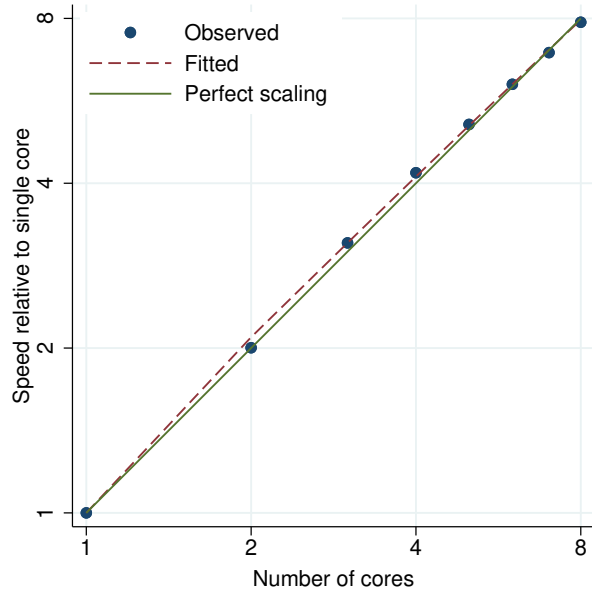


Figure 500. `streg, dist(weibull) vce(robust)` performance plot.

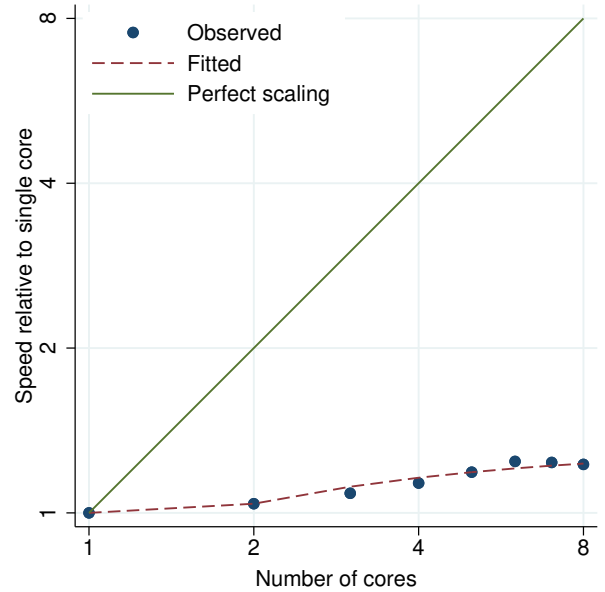


Figure 501. `sts generate` performance plot.

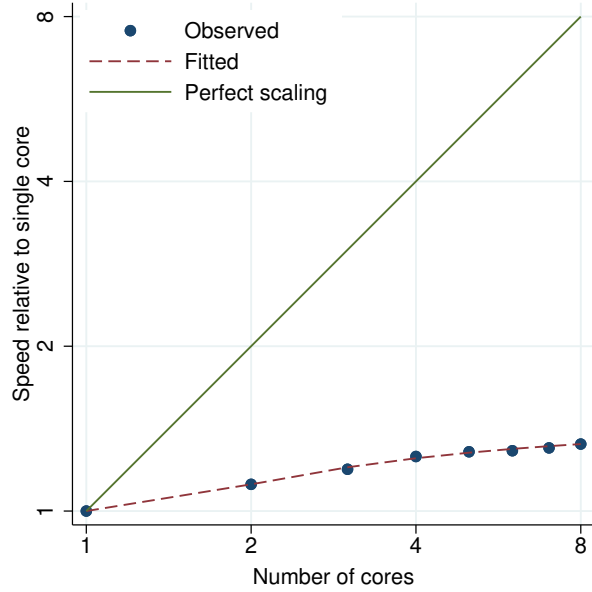


Figure 502. `sts graph` performance plot.

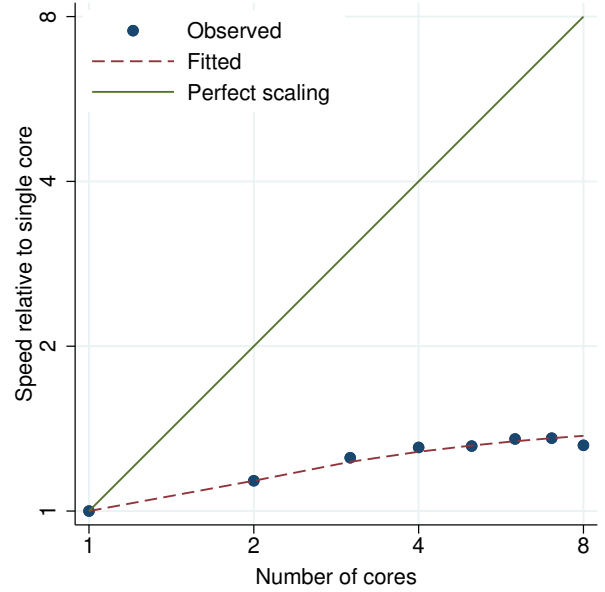


Figure 503. `sts list` performance plot.

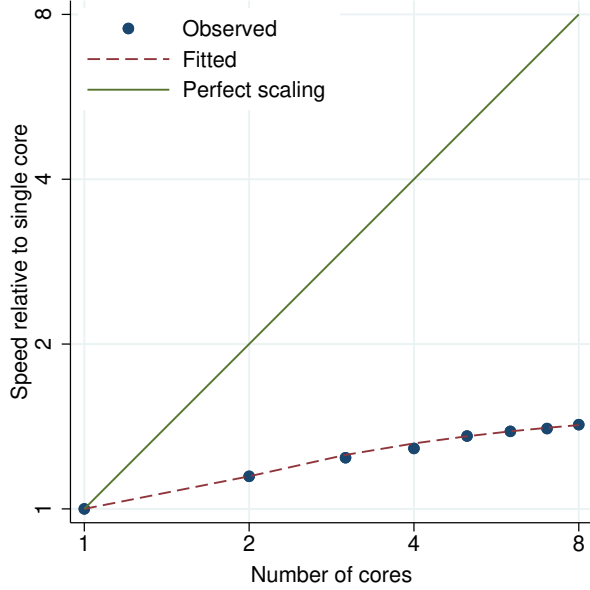


Figure 504. sts test performance plot.

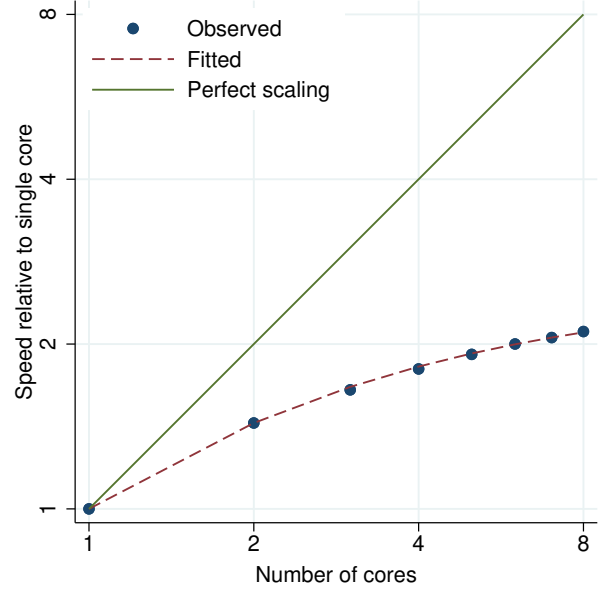


Figure 505. stset performance plot.

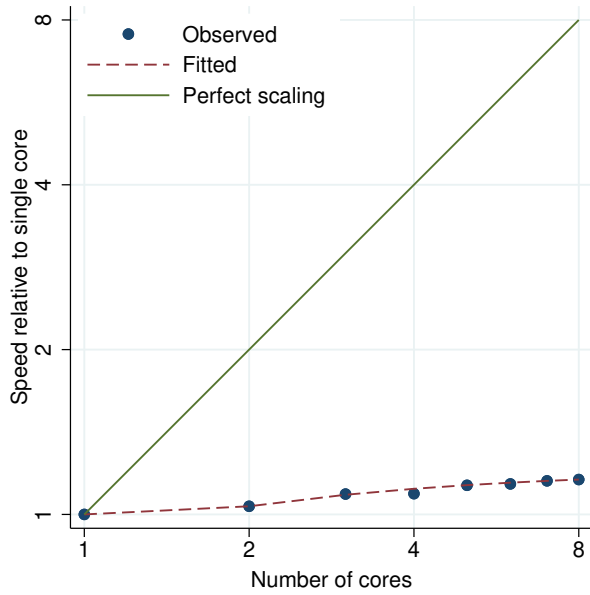


Figure 506. stsplit performance plot.

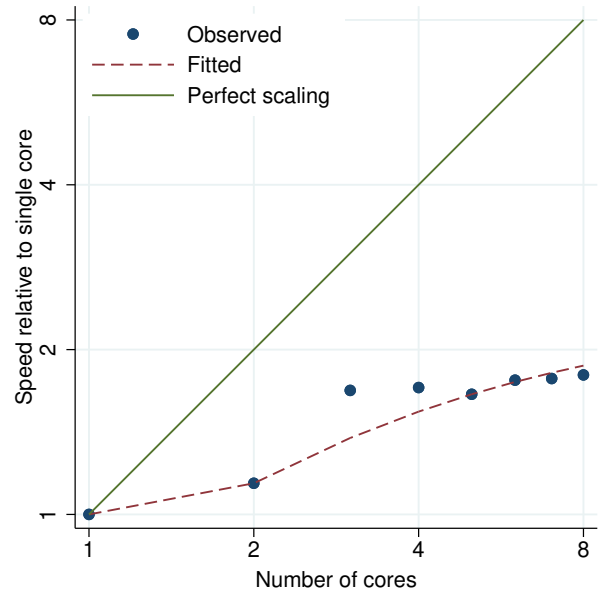


Figure 507. stsum performance plot.

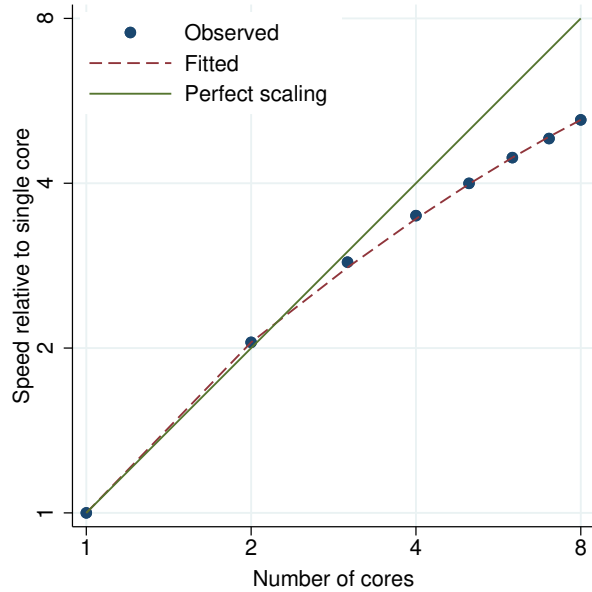


Figure 508. stteffects ipw (weibull) performance plot.

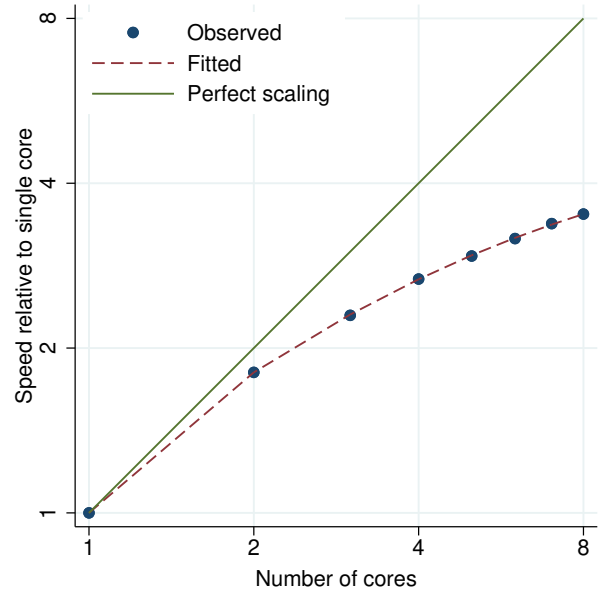


Figure 509. stteffects ipwra (weibull) performance plot.

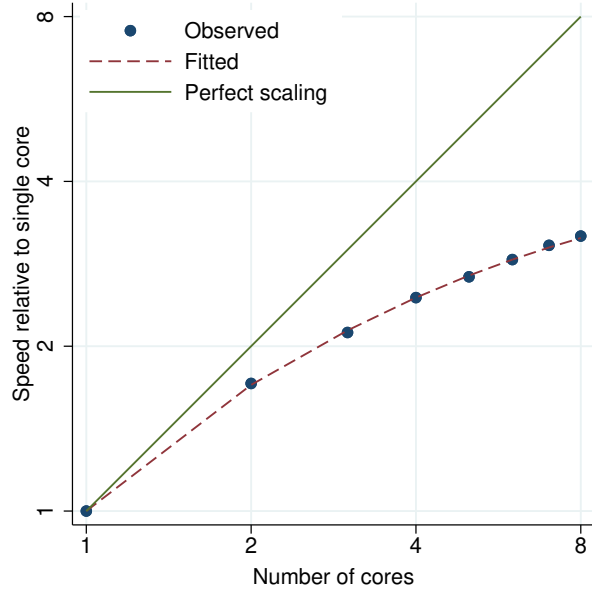


Figure 510. stteffects ra (weibull) performance plot.

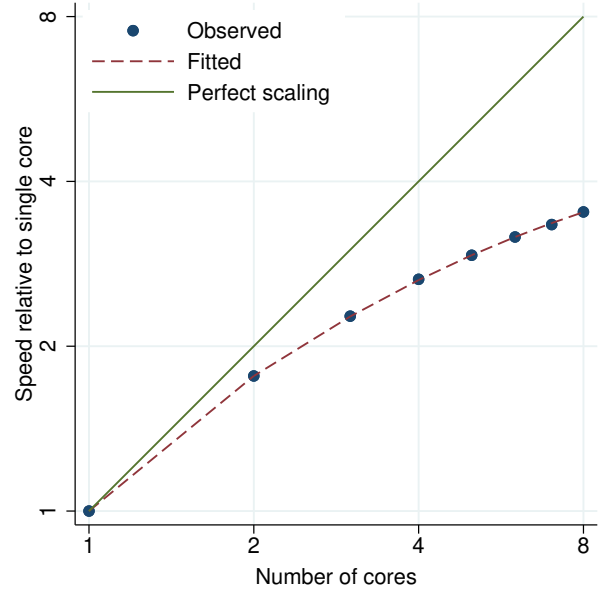


Figure 511. stteffects wra (weibull) performance plot.

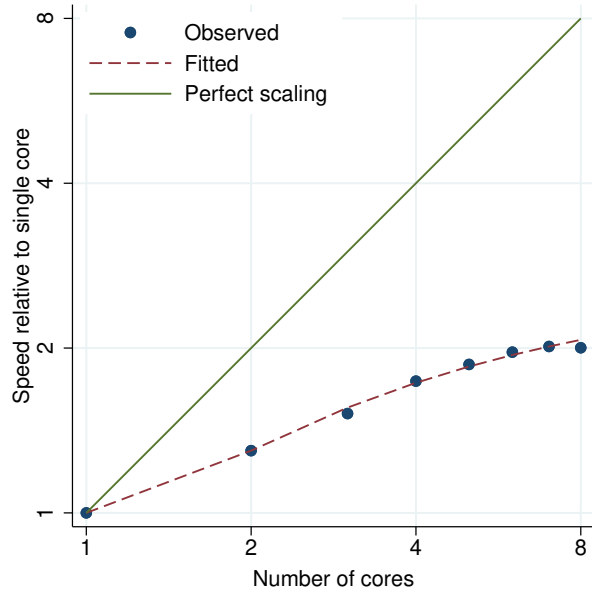


Figure 512. stvary performance plot.

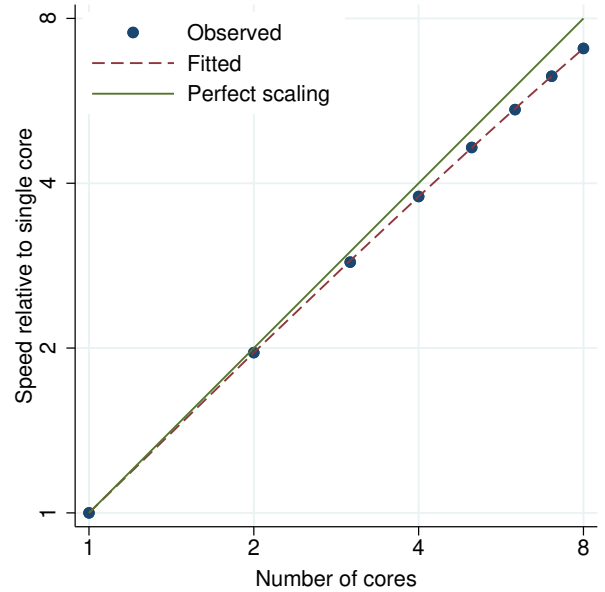


Figure 513. suest performance plot.

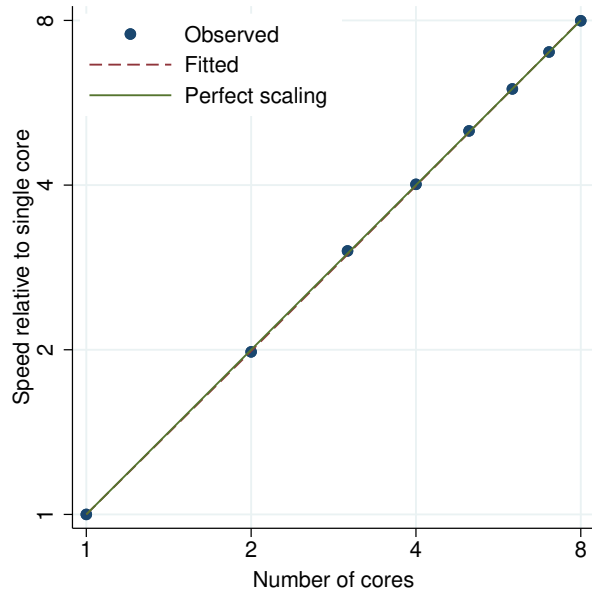


Figure 514. summarize performance plot.

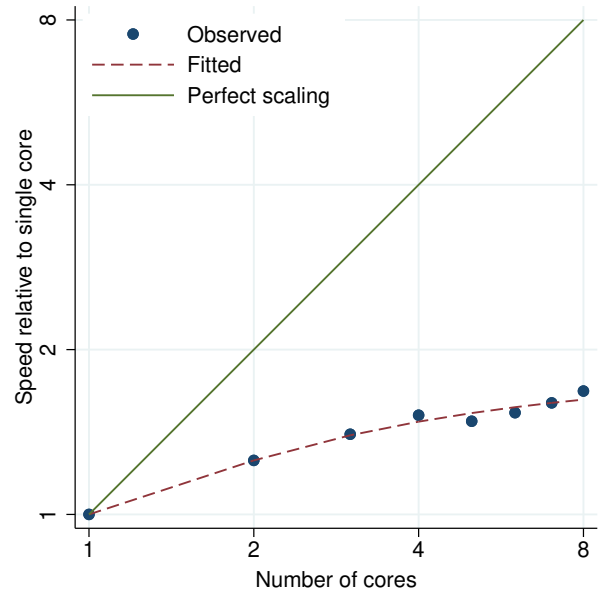


Figure 515. sunflower performance plot.

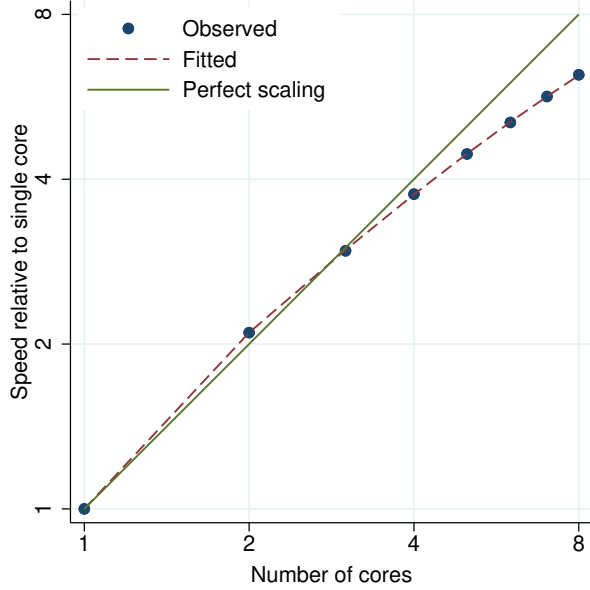


Figure 516. sureg performance plot.

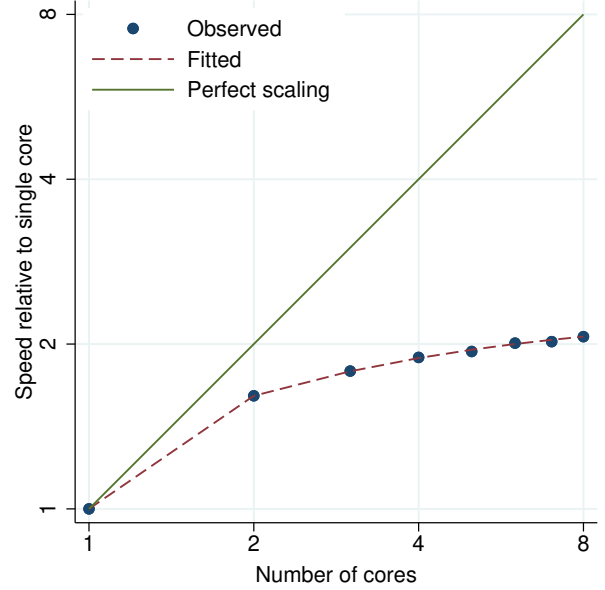


Figure 517. svar performance plot.

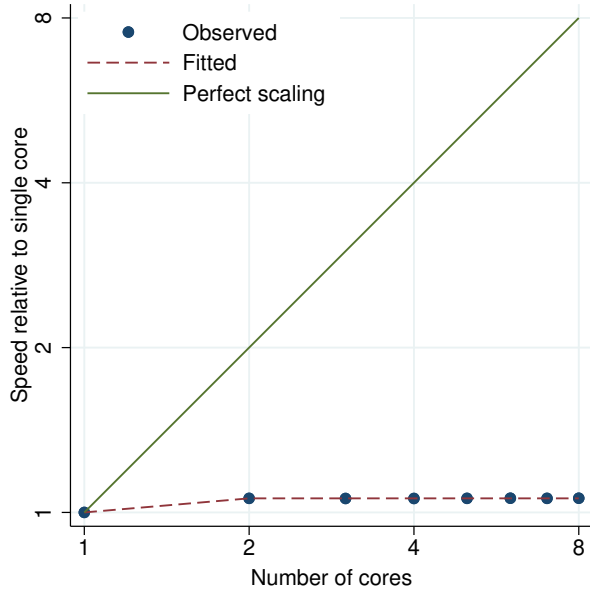


Figure 518. svmat performance plot.

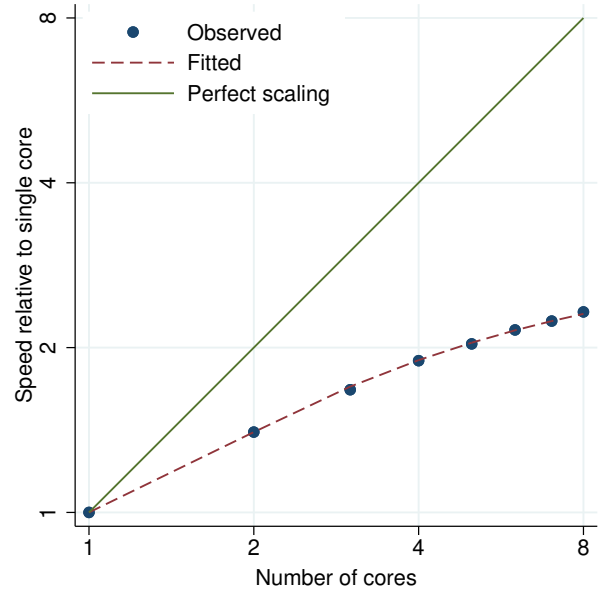


Figure 519. svy brr: logit performance plot.

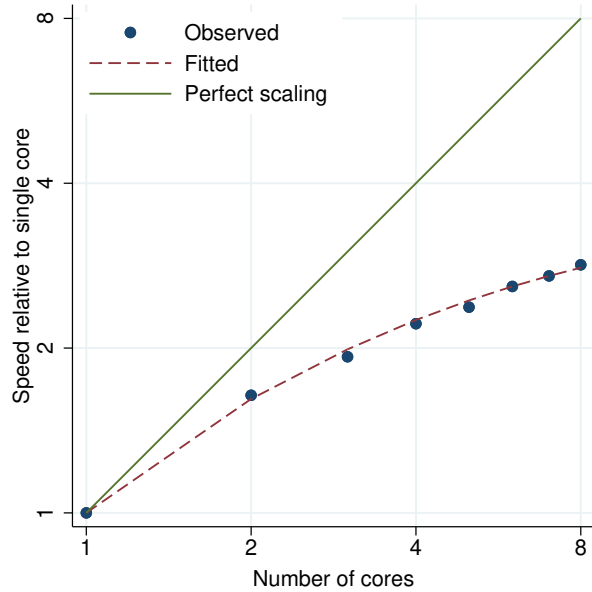


Figure 520. svy brr: poisson performance plot.

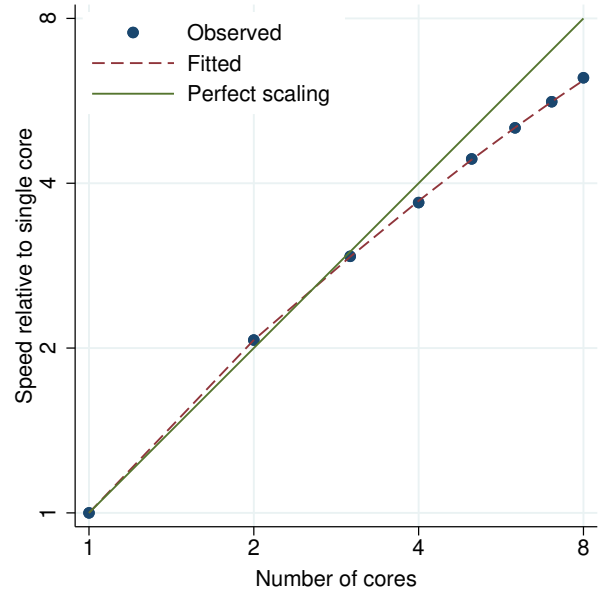


Figure 521. svy brr: regress performance plot.

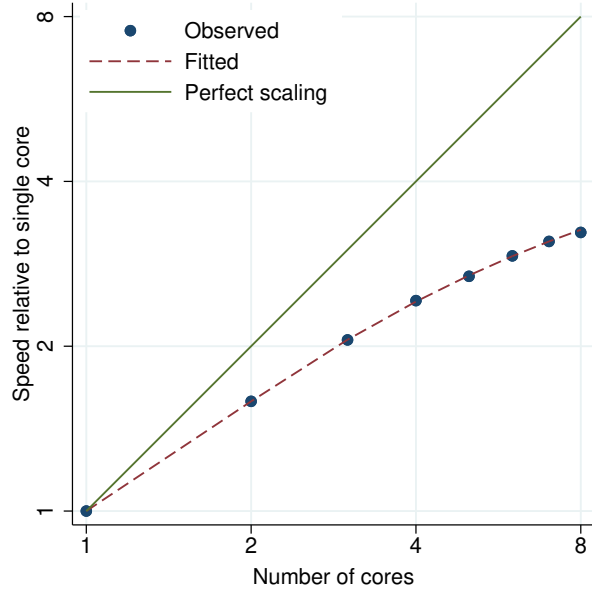


Figure 522. svy jackknife: logit performance plot.

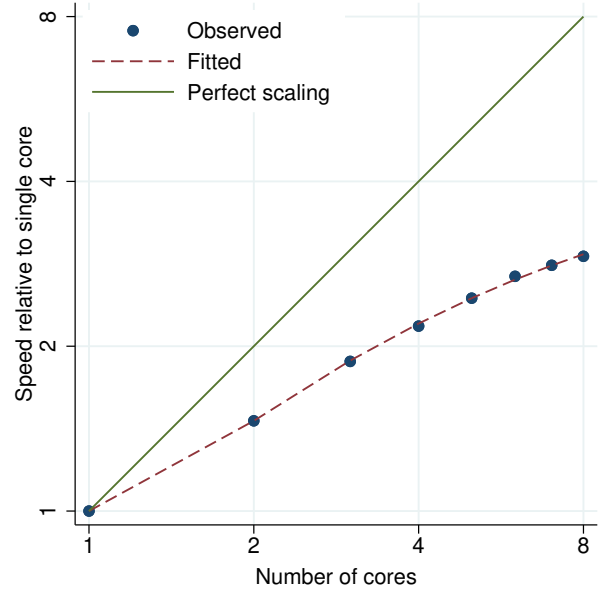


Figure 523. svy jackknife: poisson performance plot.

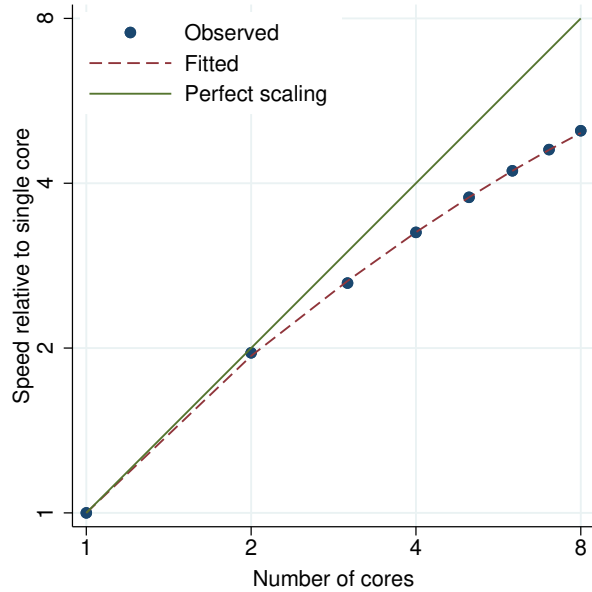


Figure 524. svy jackknife: regress performance plot.

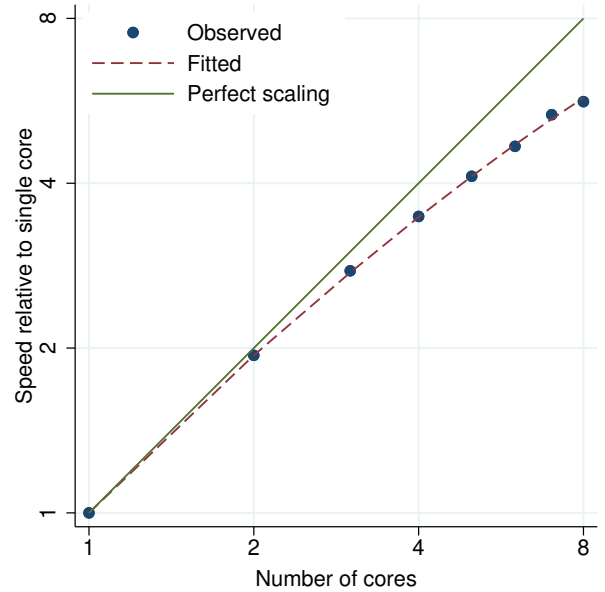


Figure 525. svy linearized: logit performance plot.

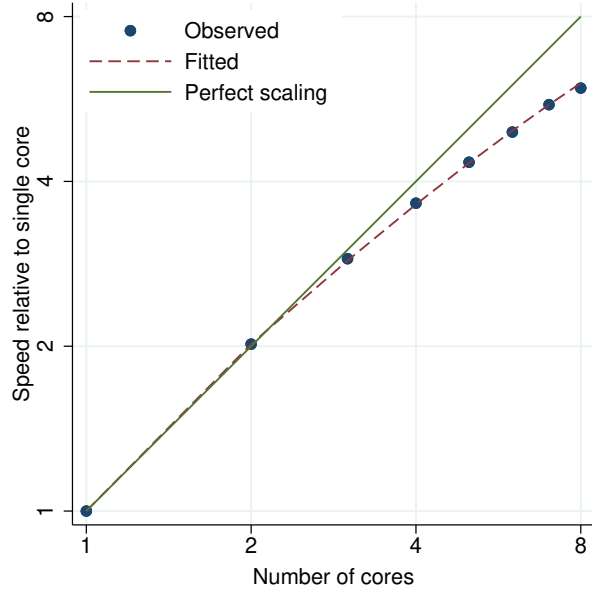


Figure 526. svy linearized: poisson performance plot.

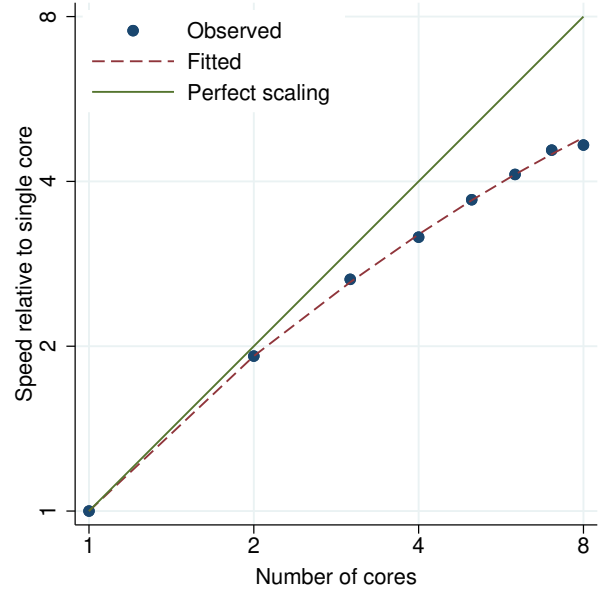


Figure 527. svy linearized: regress performance plot.

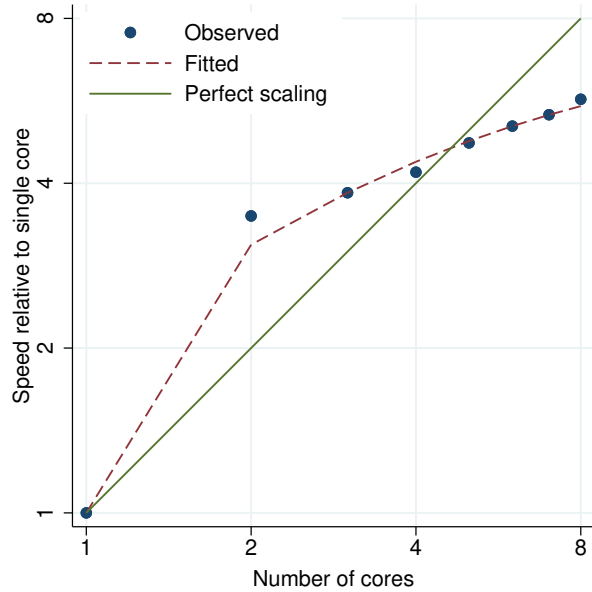


Figure 528. swilk performance plot.

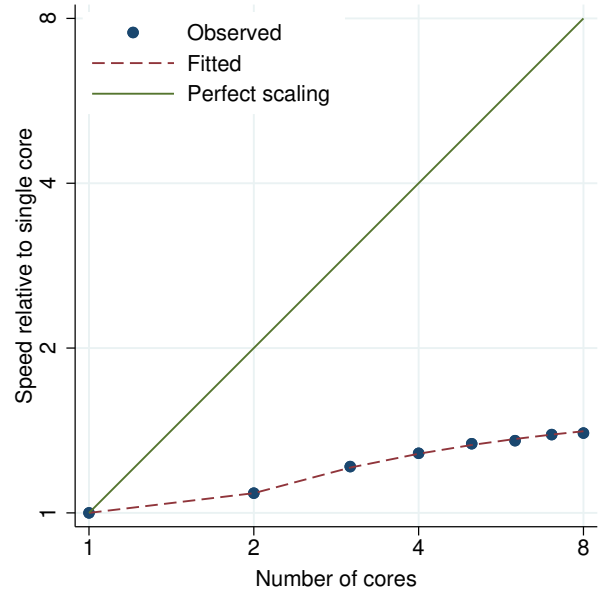


Figure 529. symmetry performance plot.

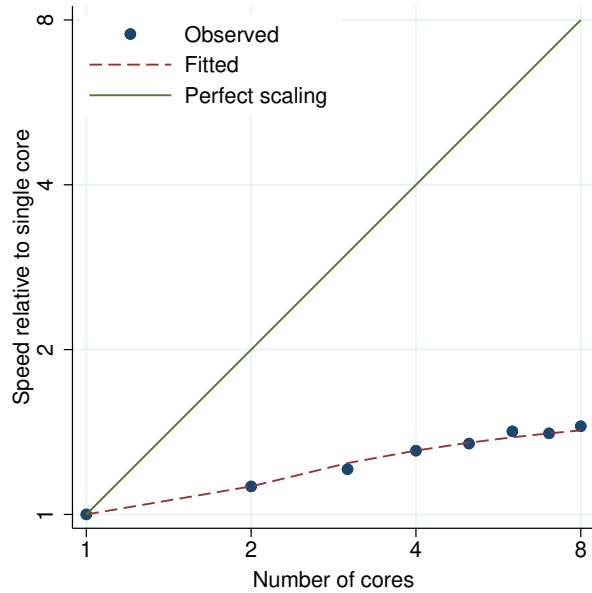


Figure 530. table (one-way) performance plot.

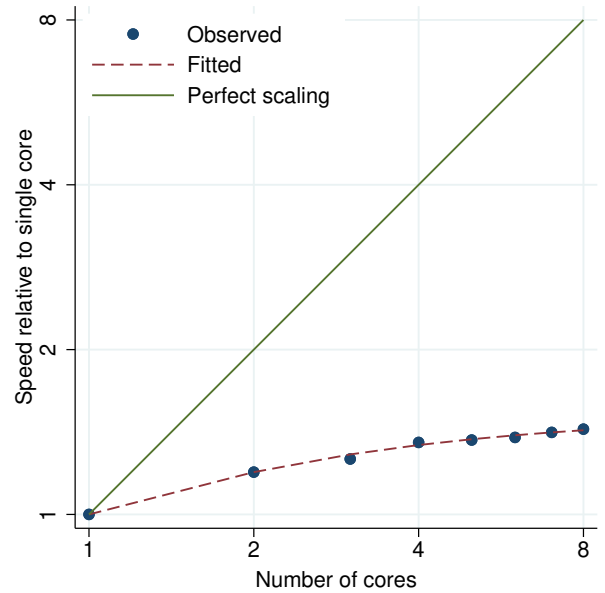


Figure 531. table (two-way) performance plot.

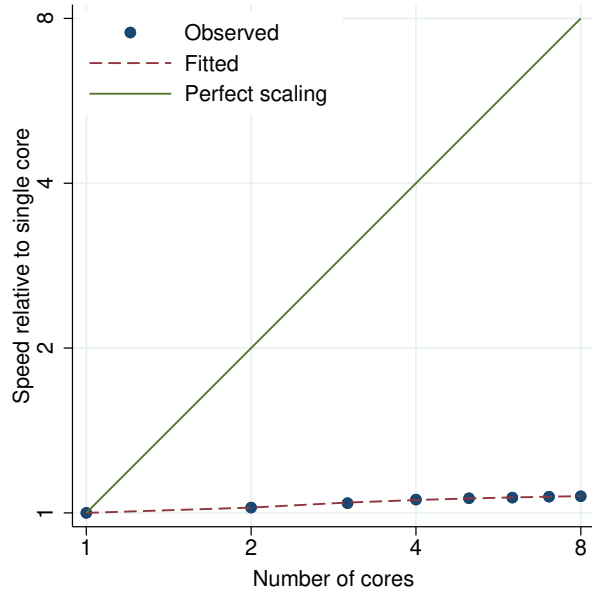


Figure 532. tabodds performance plot.

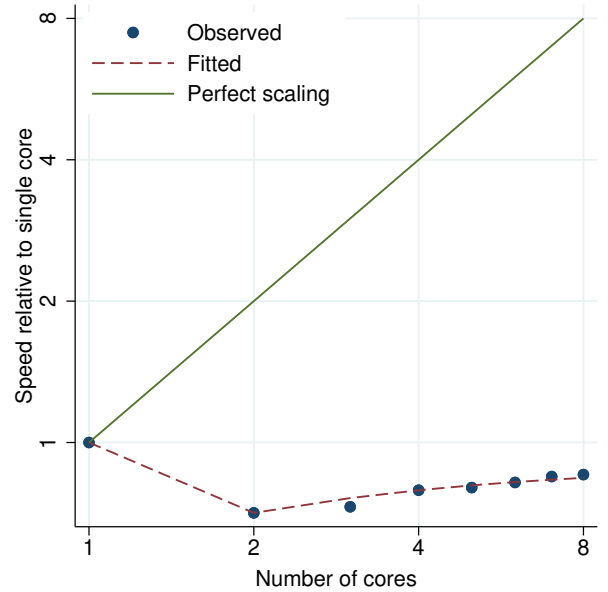


Figure 533. tabodds (adjusted) performance plot.

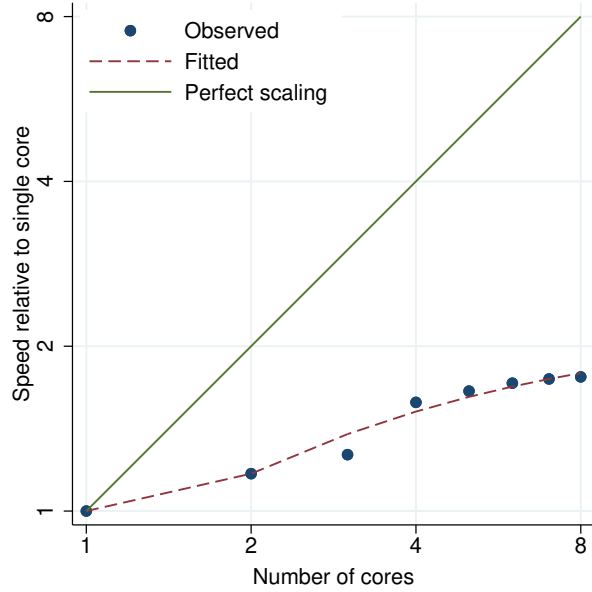


Figure 534. tabstat performance plot.

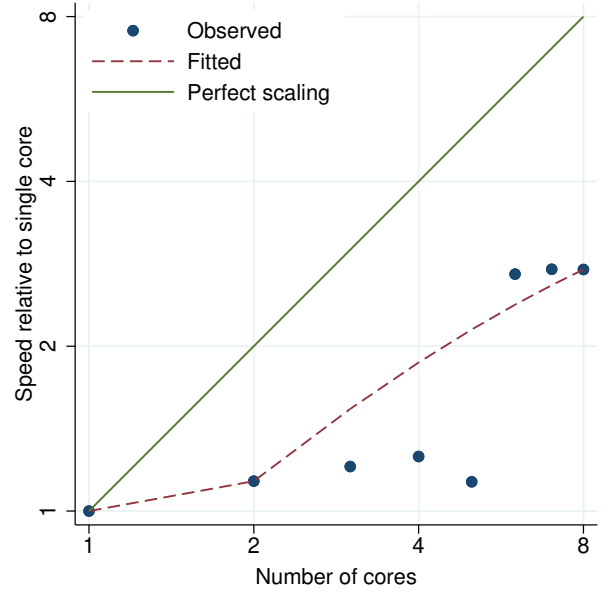


Figure 535. tabstat, by() performance plot.

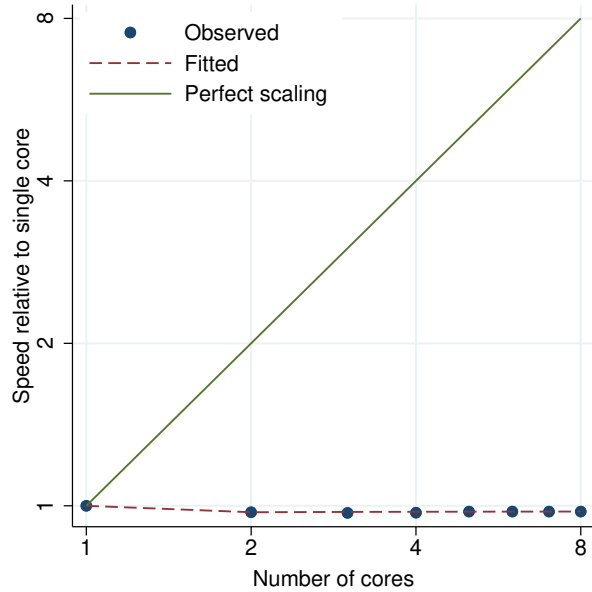


Figure 536. `tabulate` (one-way) performance plot.

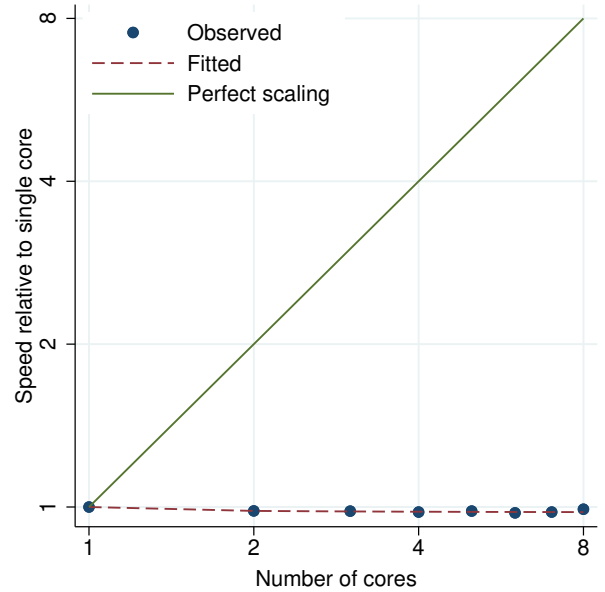


Figure 537. `tabulate` (two-way) performance plot.

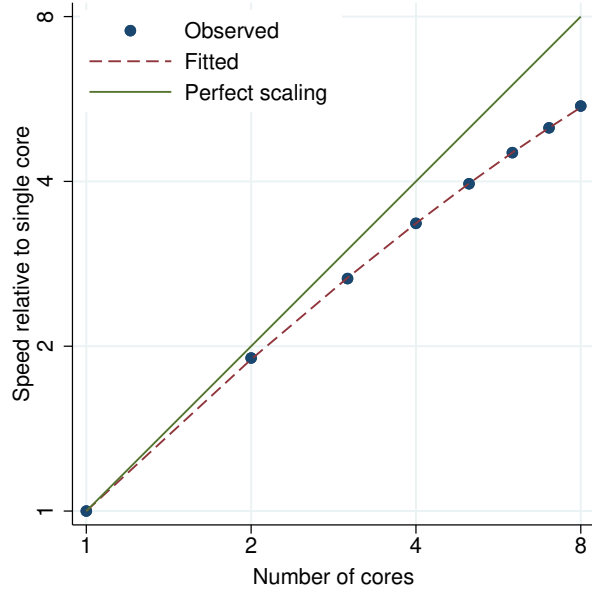


Figure 538. `teffects aipw` (linear) performance plot.

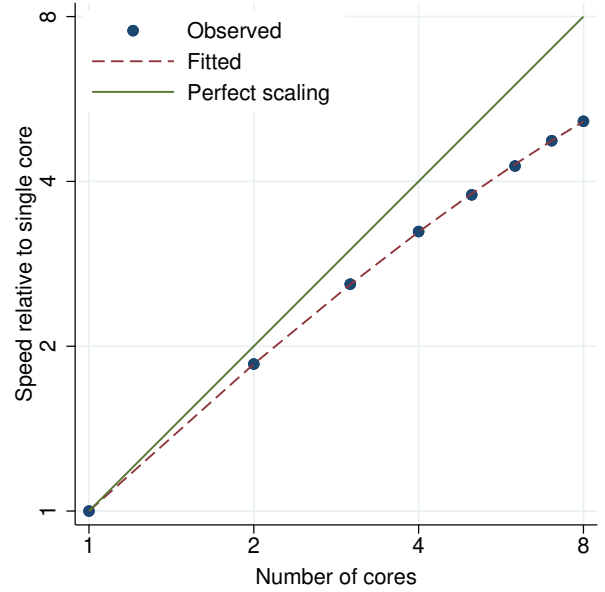


Figure 539. `teffects aipw` (probit) performance plot.

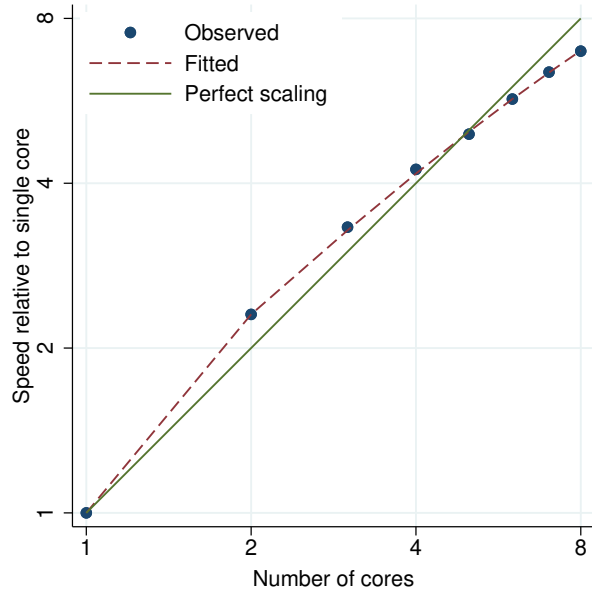


Figure 540. teffects ipw (logit) performance plot.

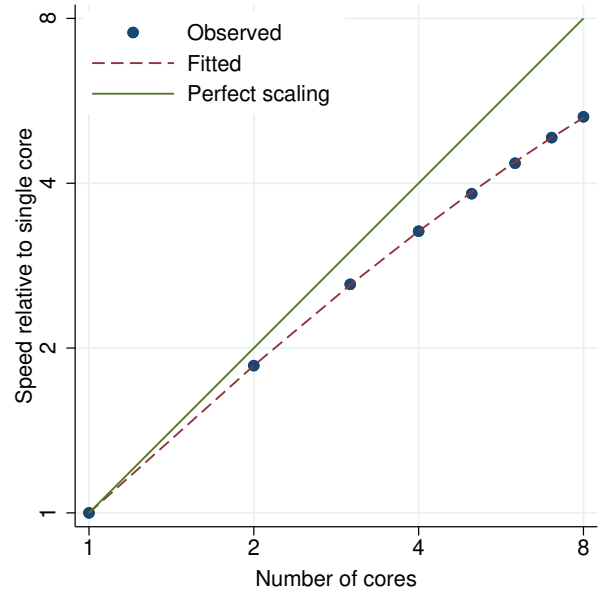


Figure 541. teffects ipwra (linear) performance plot.

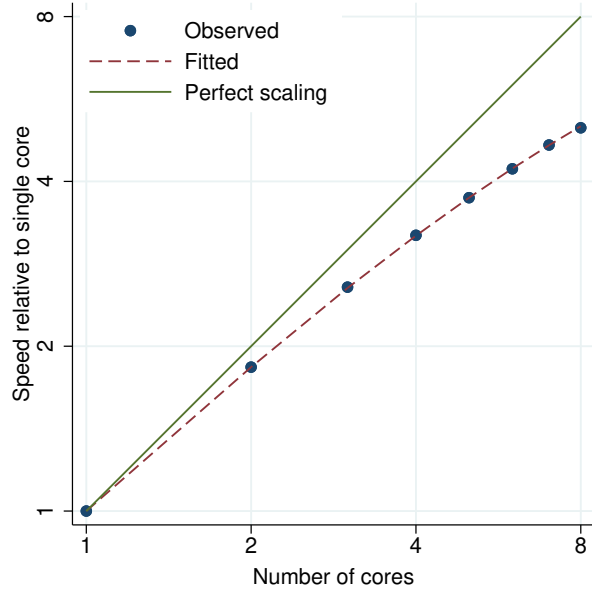


Figure 542. teffects ipwra (probit) performance plot.

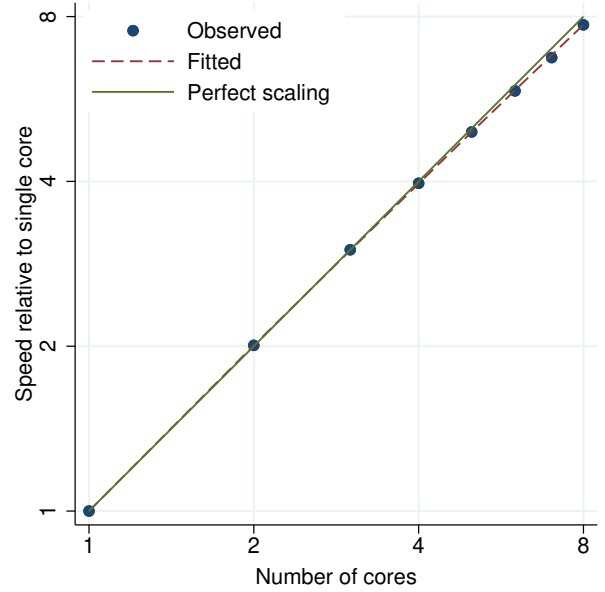


Figure 543. teffects nmatch performance plot.

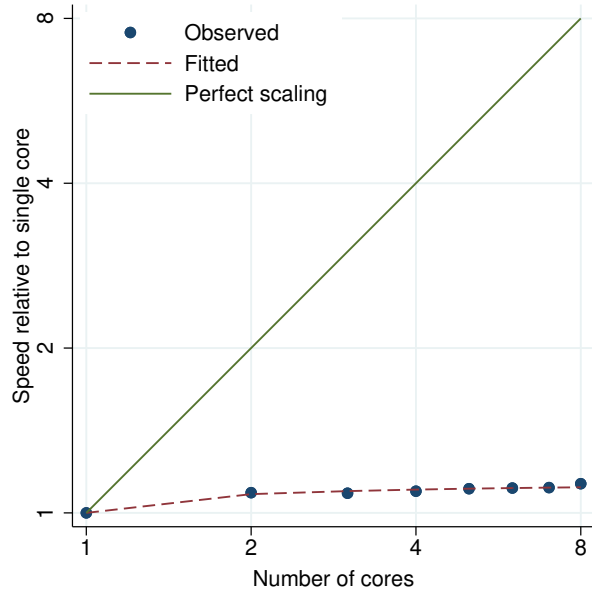


Figure 544. teffects psmatch, logit performance plot.

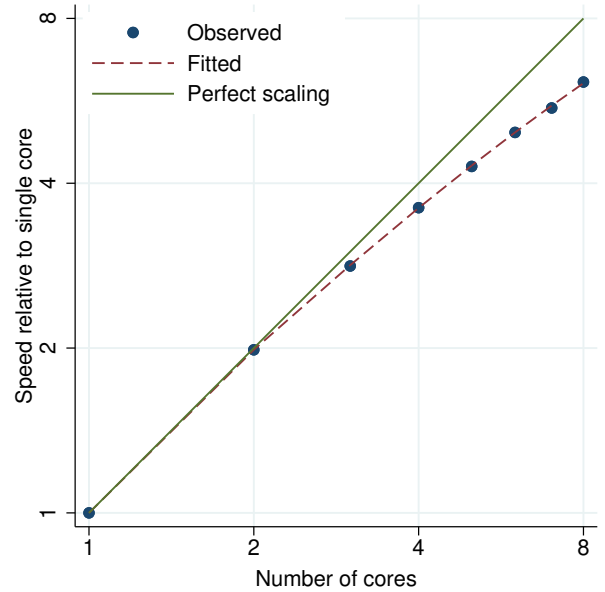


Figure 545. teffects ra (linear) performance plot.

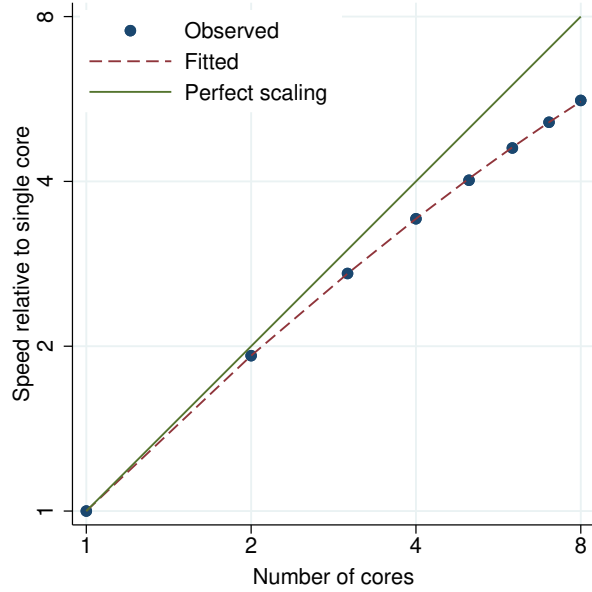


Figure 546. teffects ra (probit) performance plot.

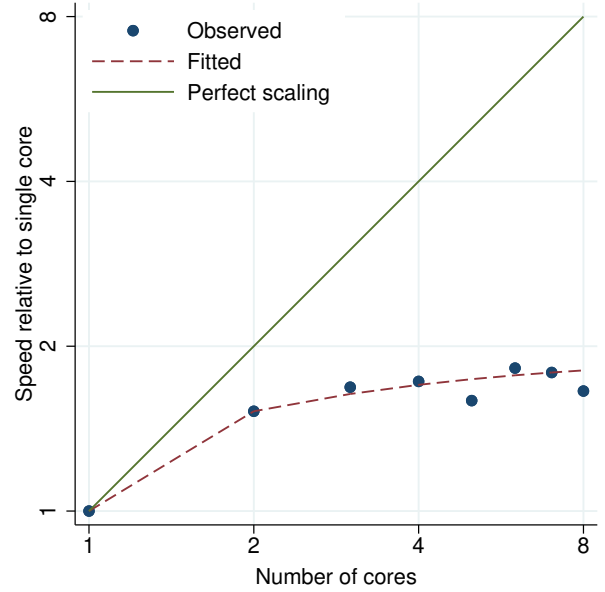


Figure 547. telasso (, linear) (, probit), ate performance plot.

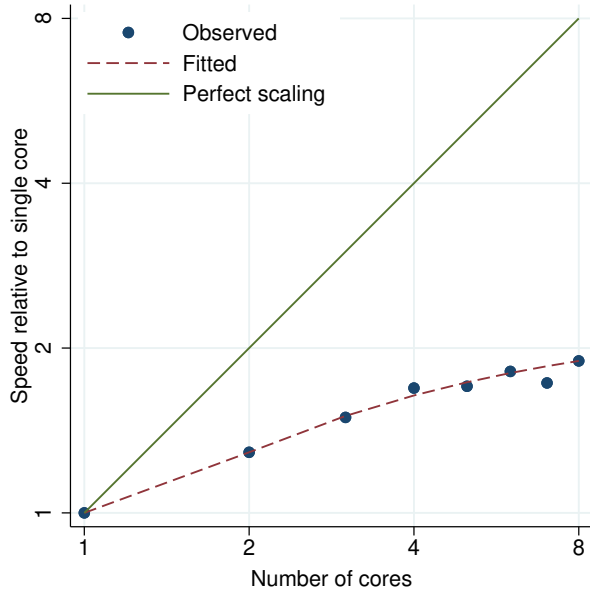


Figure 548. `telasso` (, linear) (, probit), `atet` performance plot.

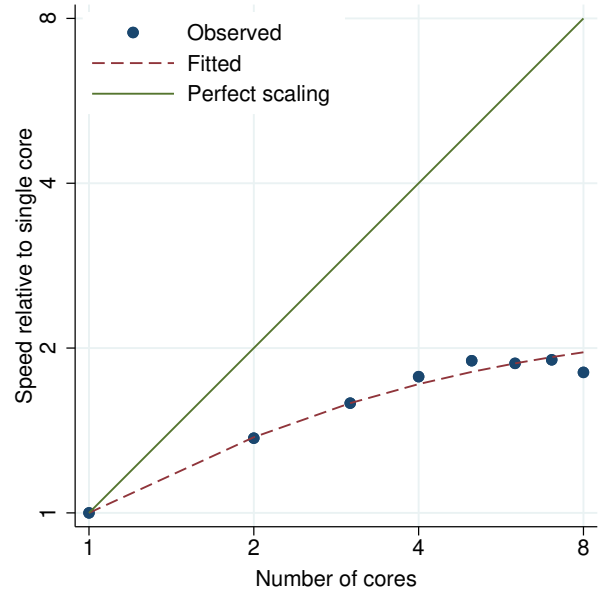


Figure 549. `telasso` (, linear) (, probit), `pomeans` performance plot.

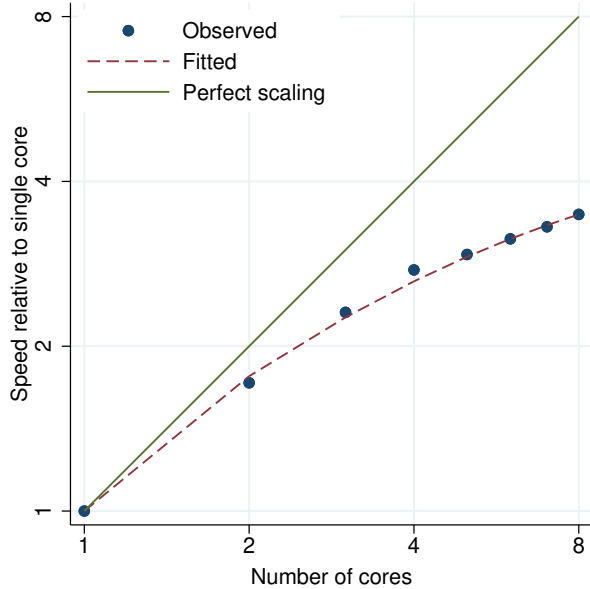


Figure 550. `telasso` (, logit) (, probit), `ate` performance plot.

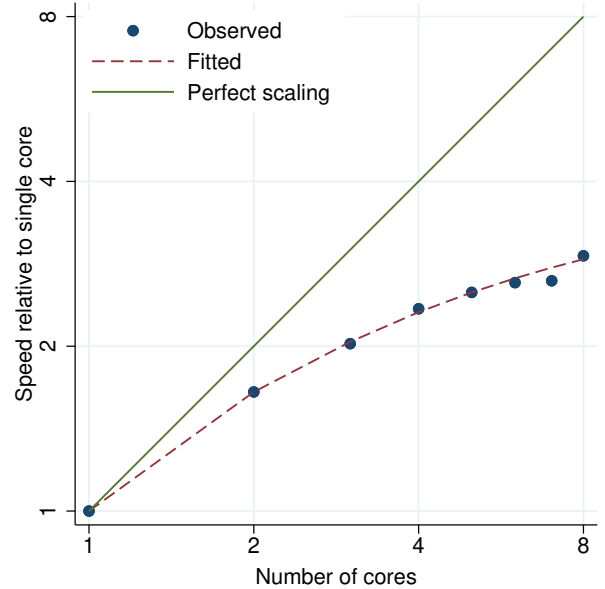


Figure 551. `telasso` (, logit) (, probit), `atet` performance plot.

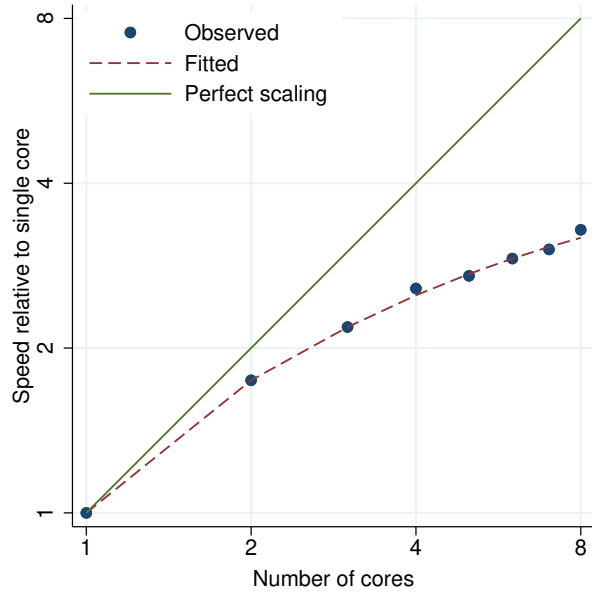


Figure 552. `telasso (, logit) (, probit), pomeans` performance plot.

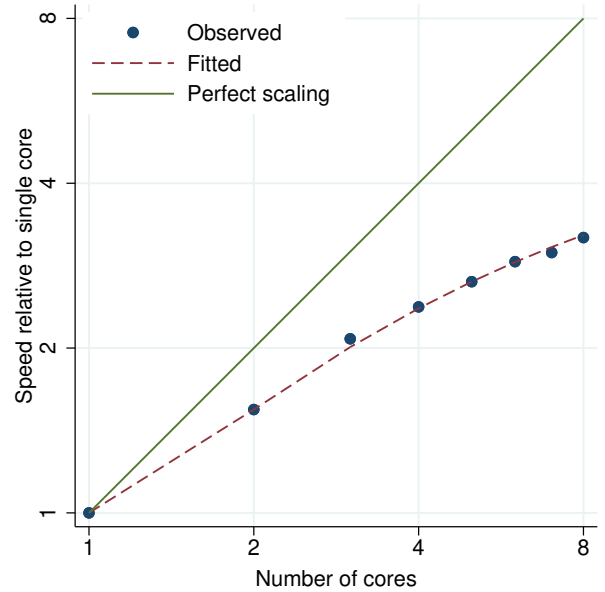


Figure 553. `telasso (, poisson) (, probit), ate` performance plot.

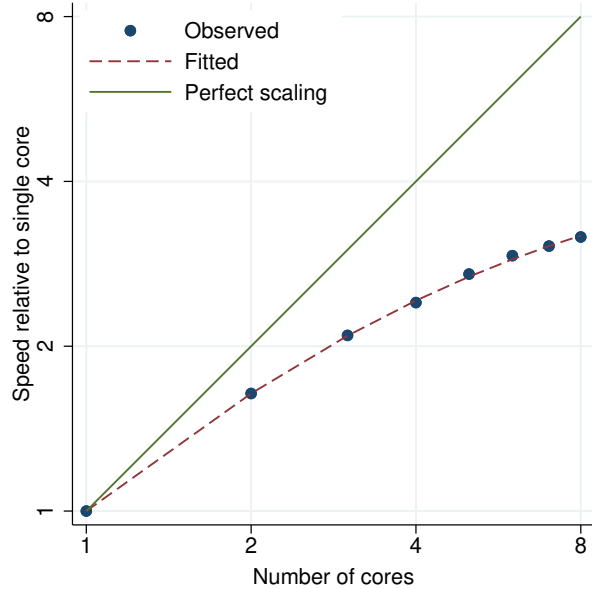


Figure 554. `telasso (, poisson) (, probit), atet` performance plot.

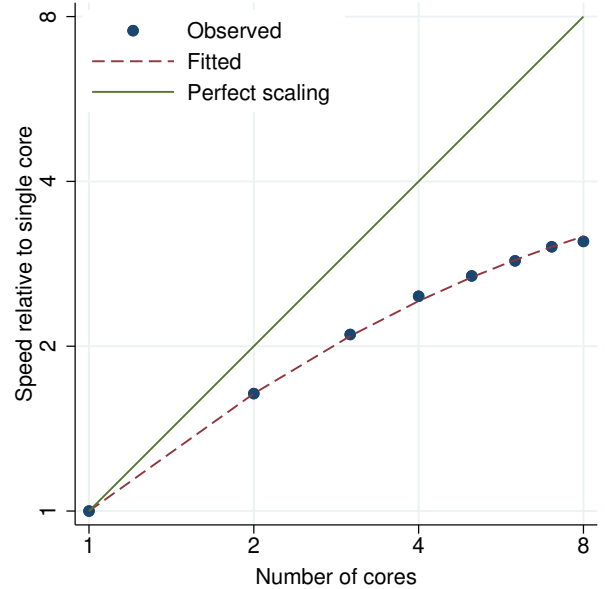


Figure 555. `telasso (, poisson) (, probit), pomeans` performance plot.

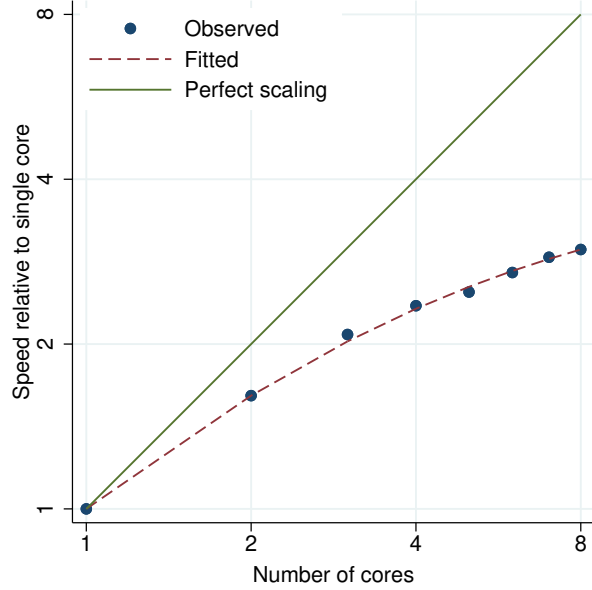


Figure 556. `telasso` (`,` `probit`) (`,` `probit`), `ate` performance plot.

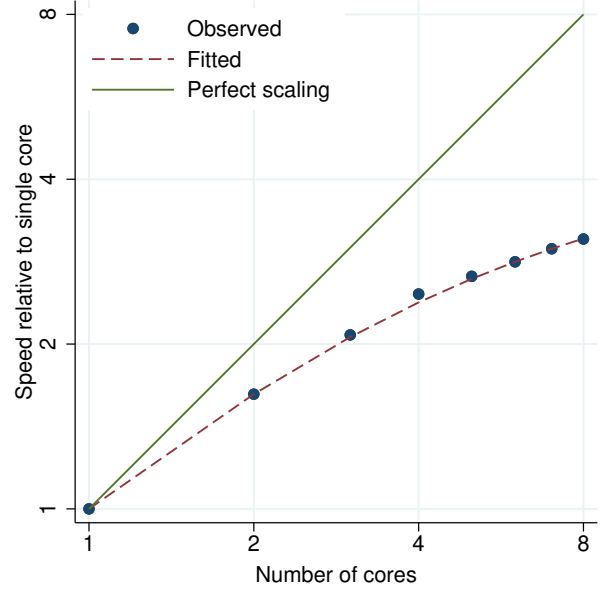


Figure 557. `telasso` (`,` `probit`) (`,` `probit`), `atet` performance plot.

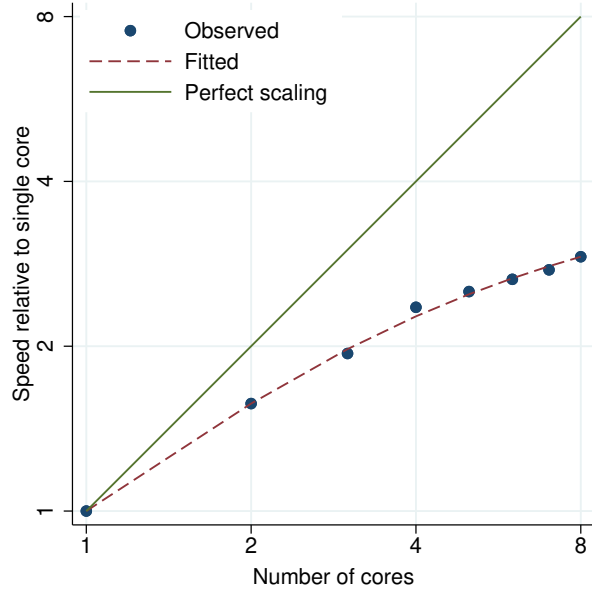


Figure 558. `telasso` (`,` `probit`) (`,` `probit`), `pomeans` performance plot.

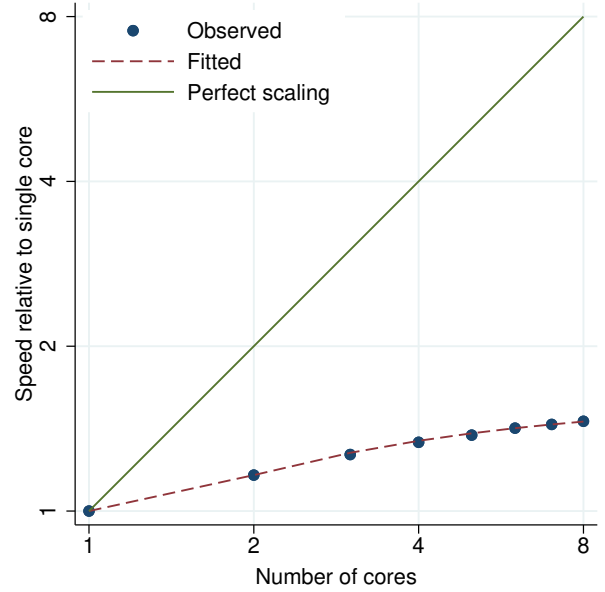


Figure 559. `tetrachoric` performance plot.

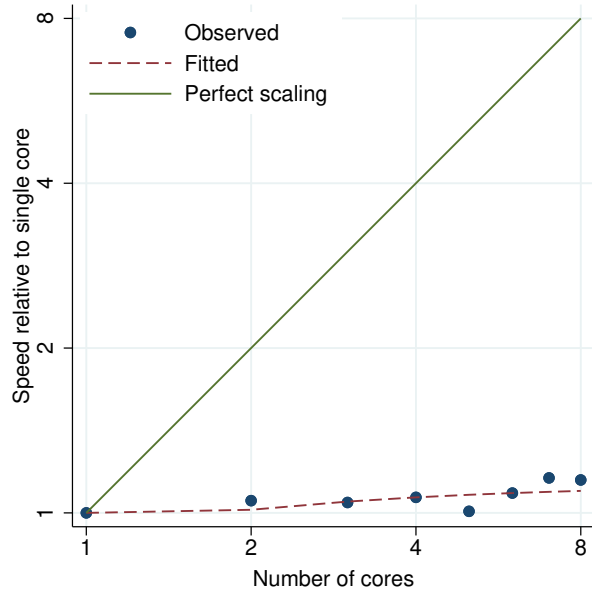


Figure 560. `threshold`, `threshvar()` performance plot.

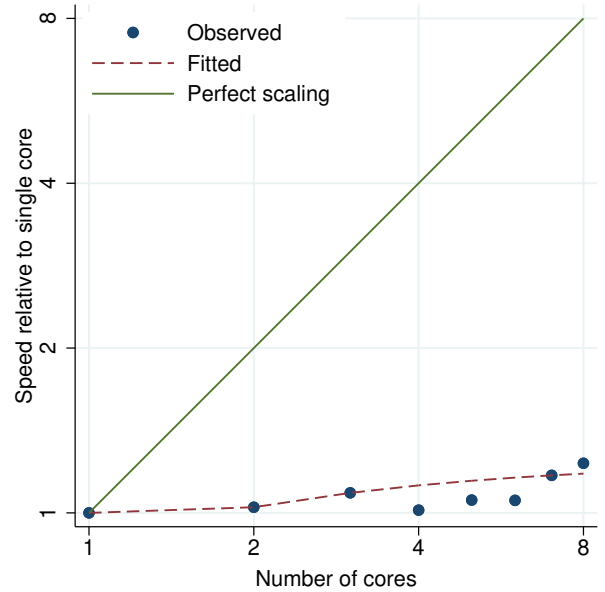


Figure 561. `threshold`, `threshvar()` `regionvars()` performance plot.

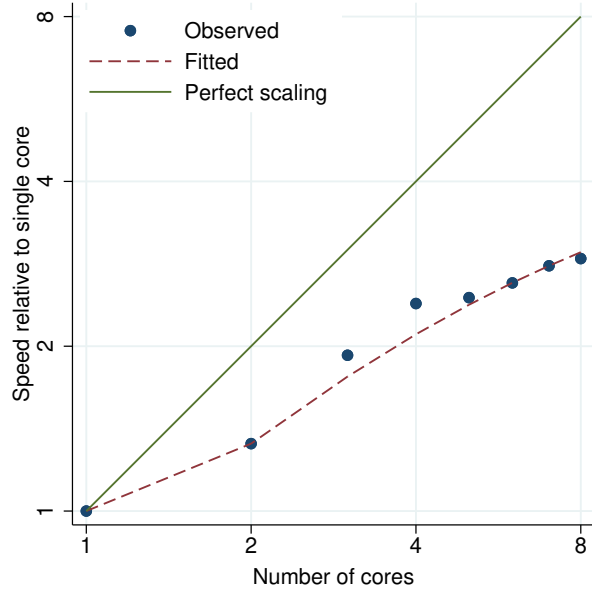


Figure 562. `tnbreg` performance plot.

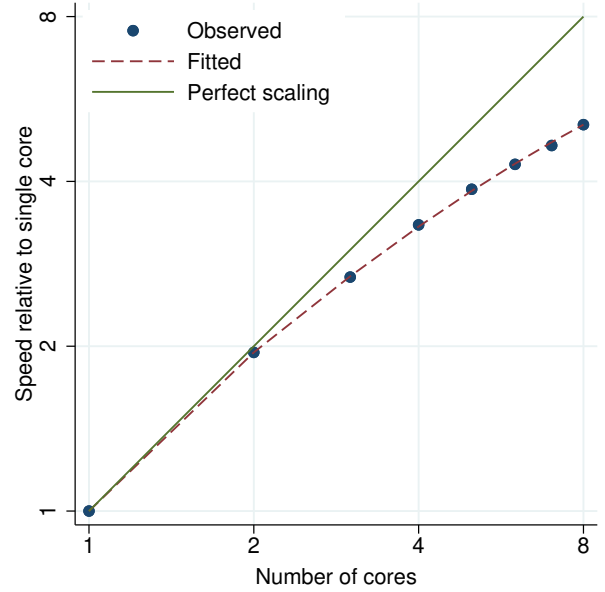


Figure 563. `tobit` performance plot.

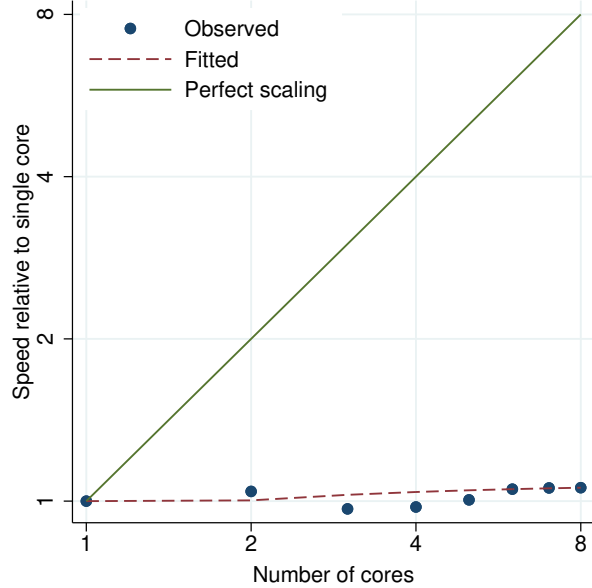


Figure 564. tostring performance plot.

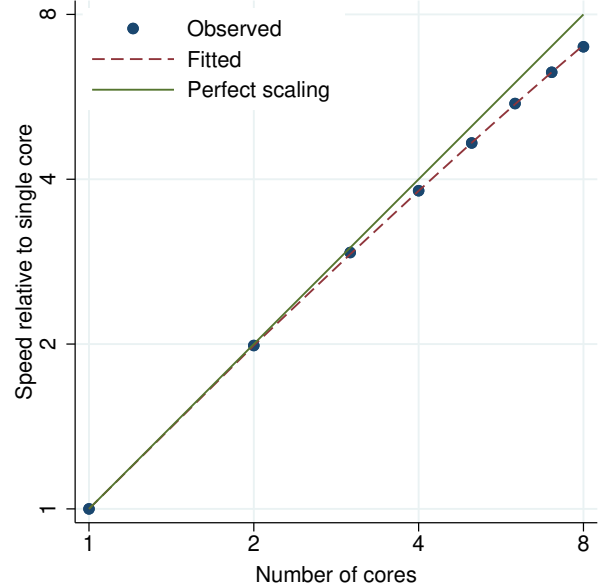


Figure 565. total performance plot.

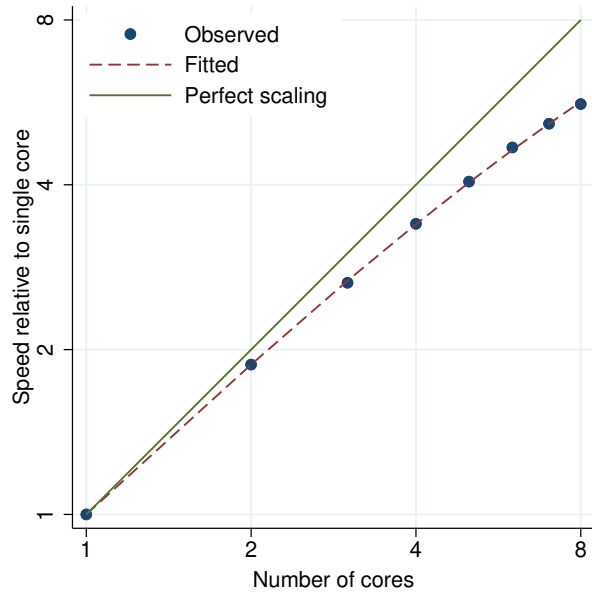


Figure 566. tpoisson performance plot.

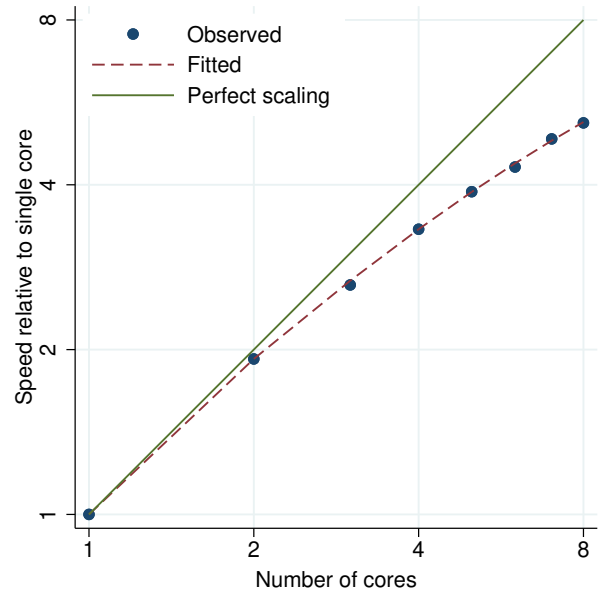


Figure 567. truncreg performance plot.

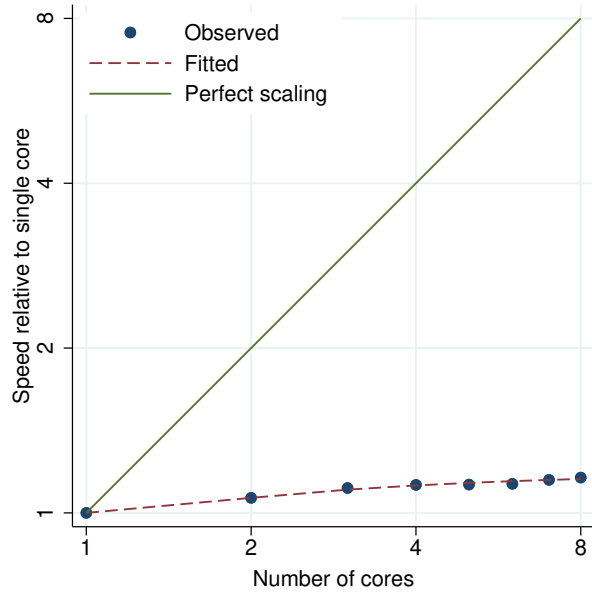


Figure 568. `tsfilter bk` performance plot.

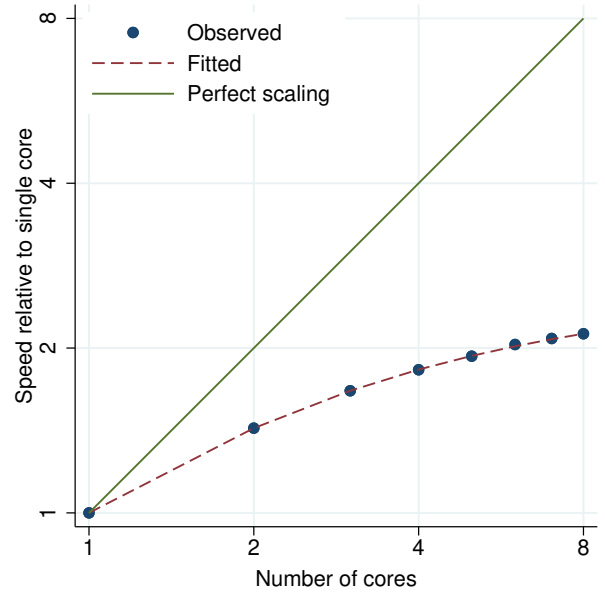


Figure 569. `tsfilter bw` performance plot.

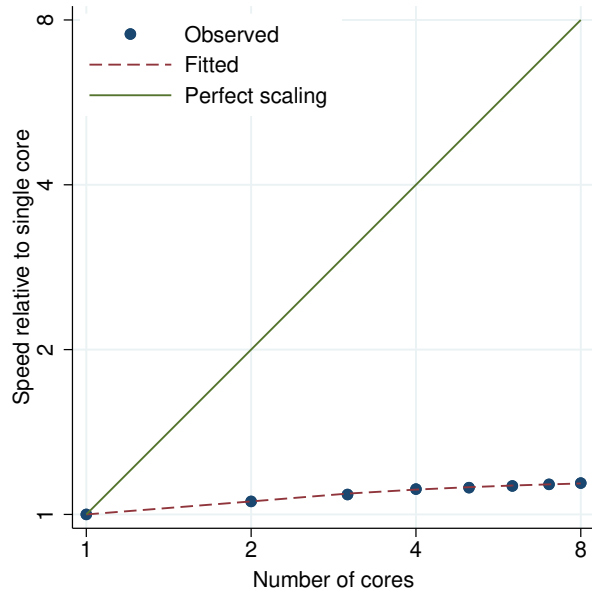


Figure 570. `tsfilter cf` performance plot.

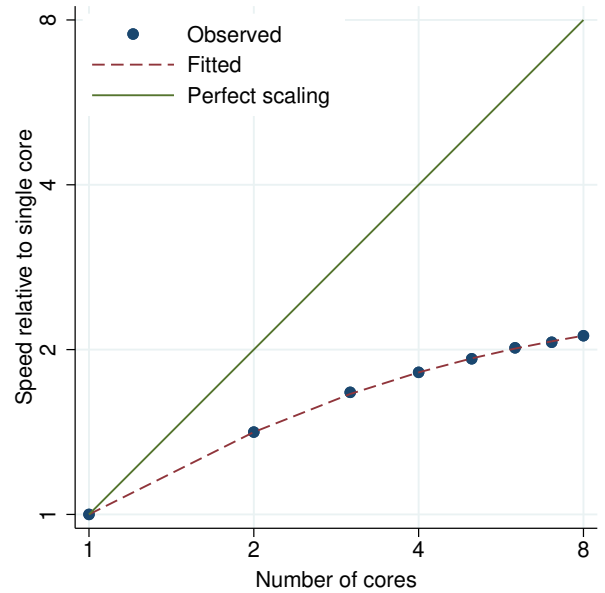


Figure 571. `tsfilter hp` performance plot.

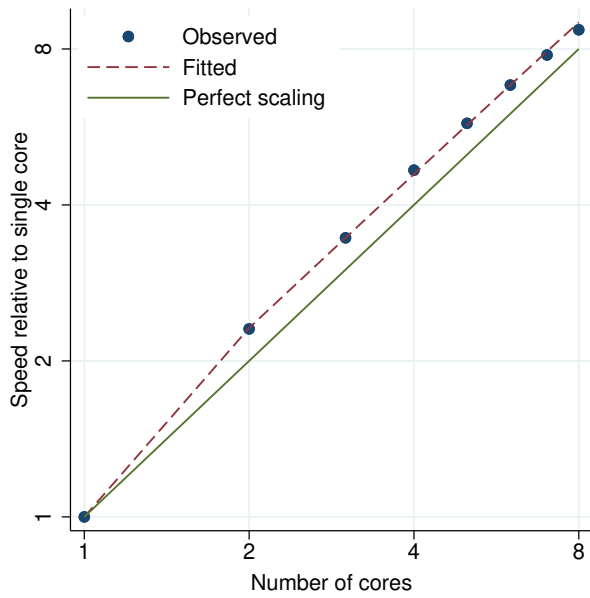


Figure 572. tsrevar performance plot.

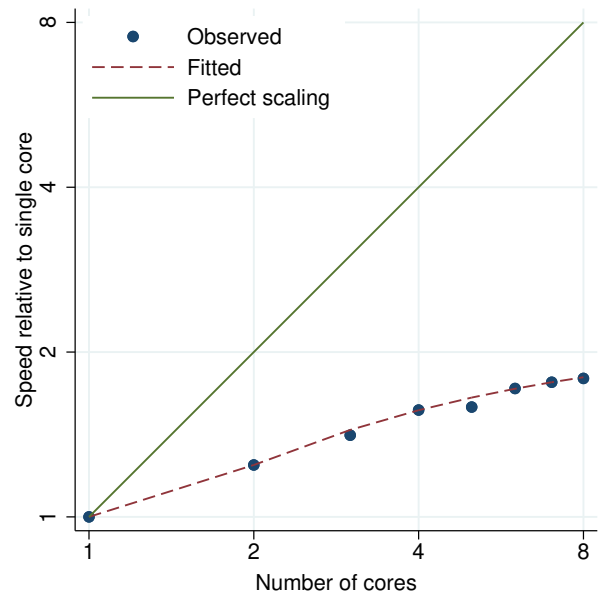


Figure 573. tsset performance plot.

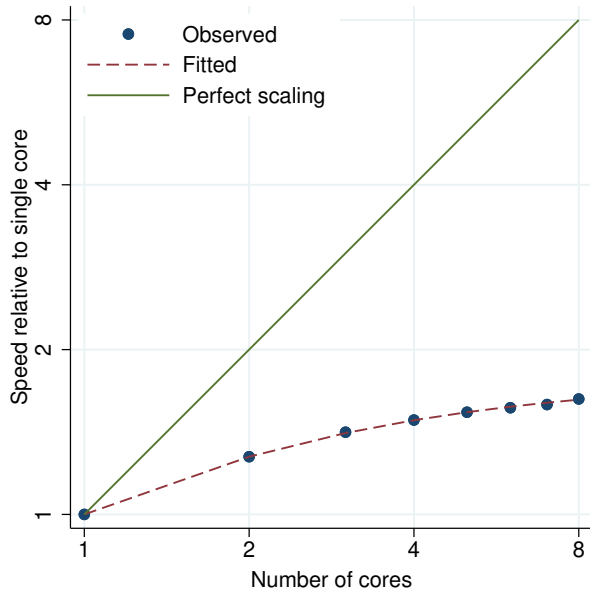


Figure 574. tssmooth exp performance plot.

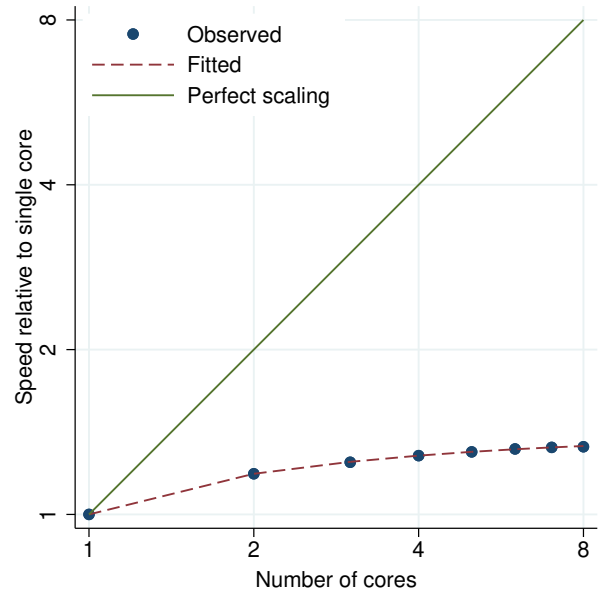


Figure 575. tssmooth ma performance plot.

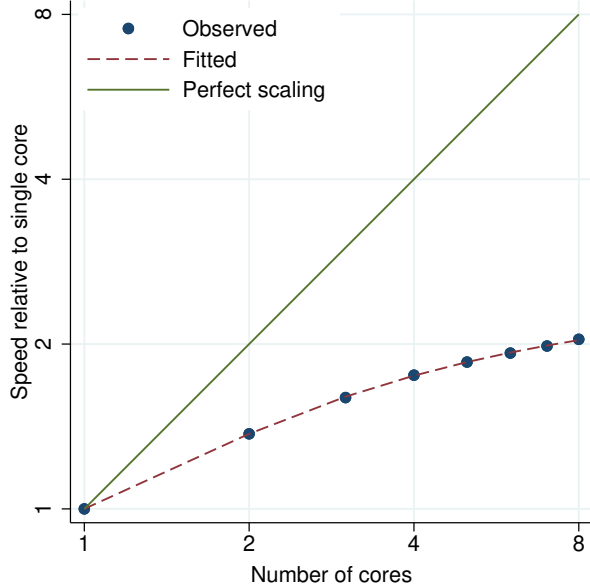


Figure 576. tttest1 performance plot.

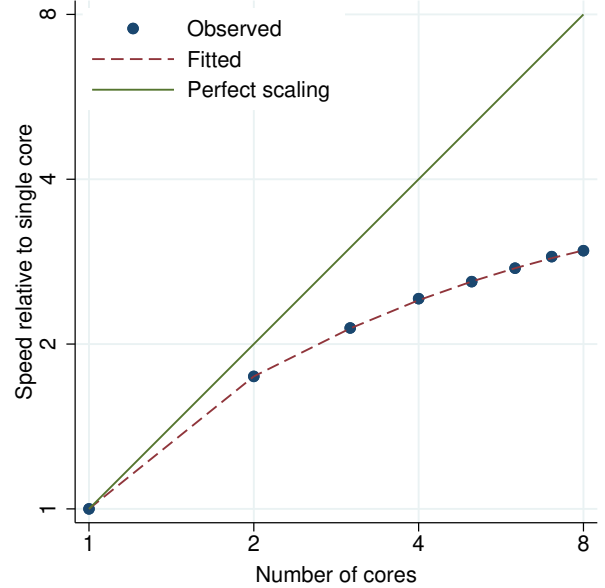


Figure 577. tttest2 performance plot.

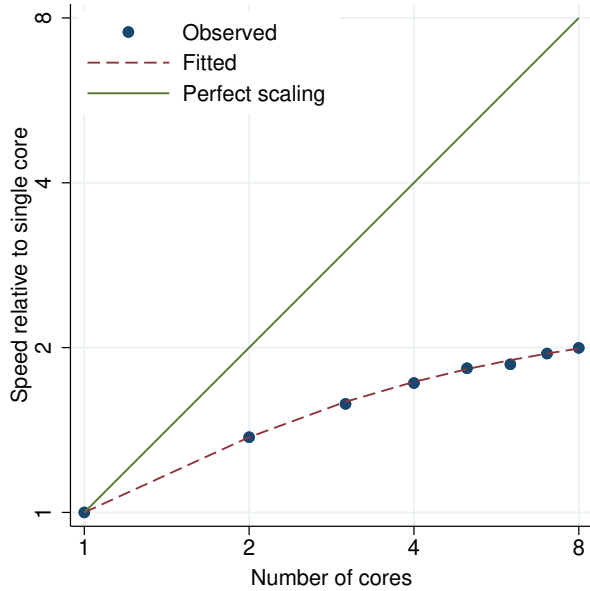


Figure 578. tttest, by() performance plot.

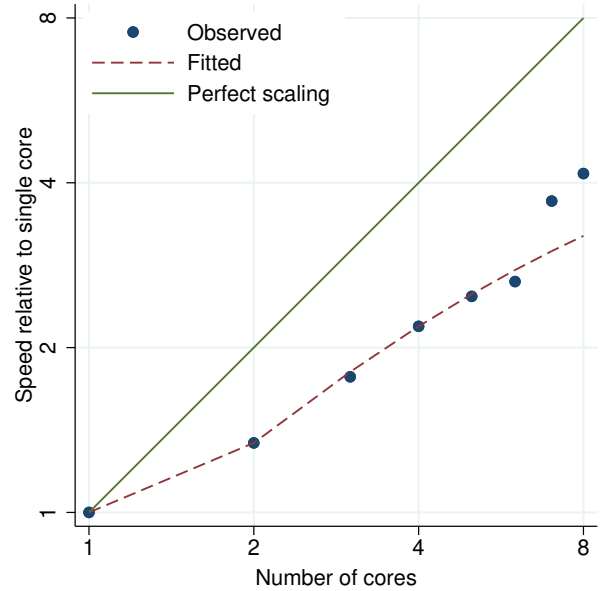


Figure 579. twoway fpfit performance plot.

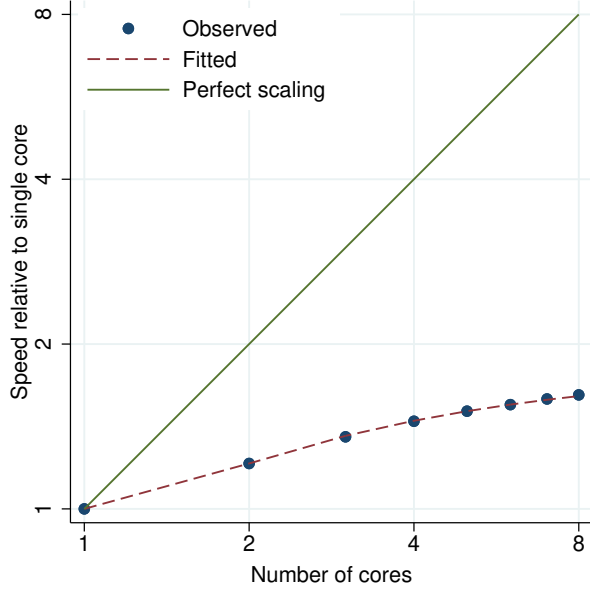


Figure 580. twoway lfitci performance plot.

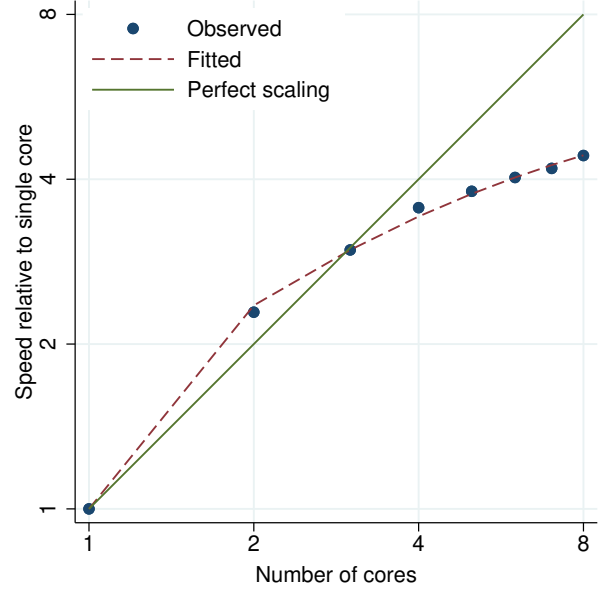


Figure 581. twoway mband performance plot.

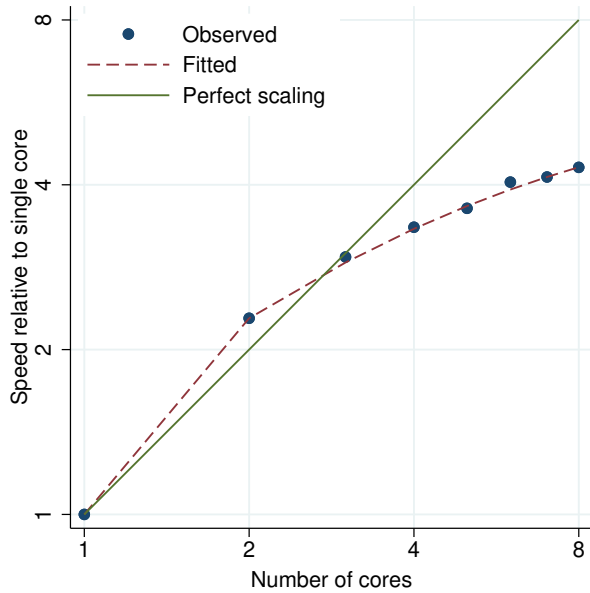


Figure 582. twoway mspline performance plot.

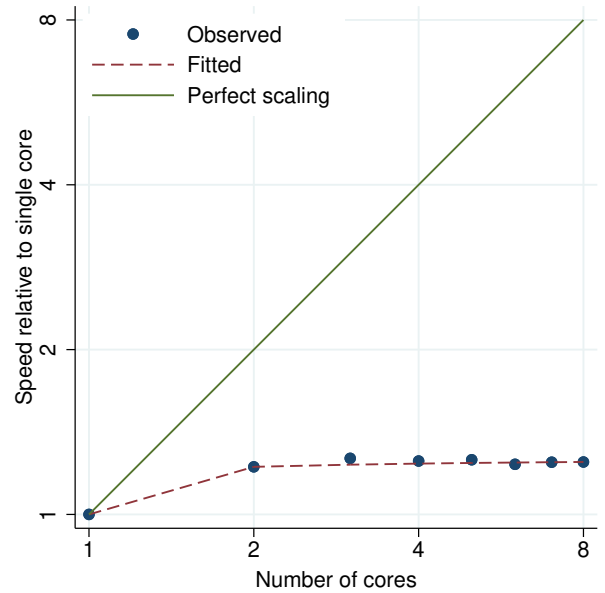


Figure 583. ucm, model(rwdrift) performance plot.

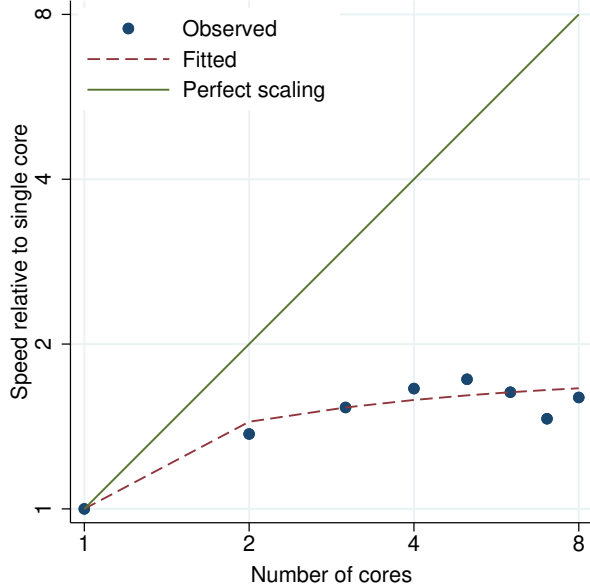


Figure 584. var performance plot.

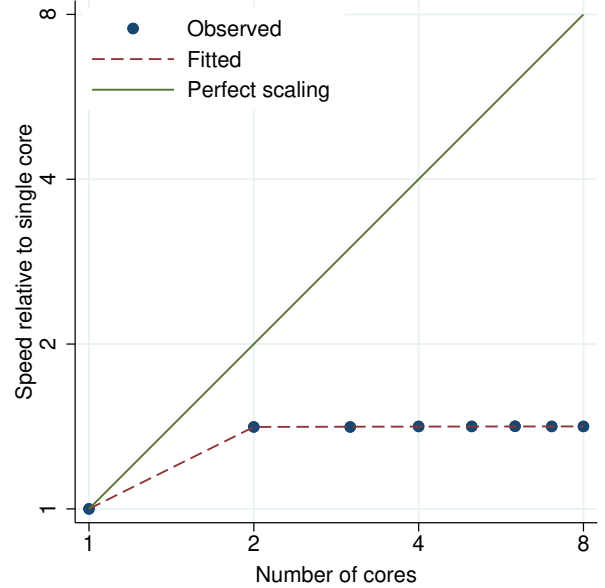


Figure 585. vargranger performance plot.

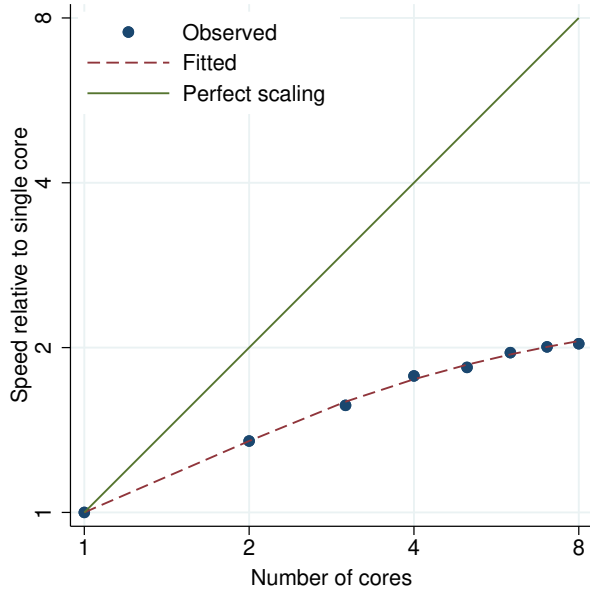


Figure 586. var1mar performance plot.

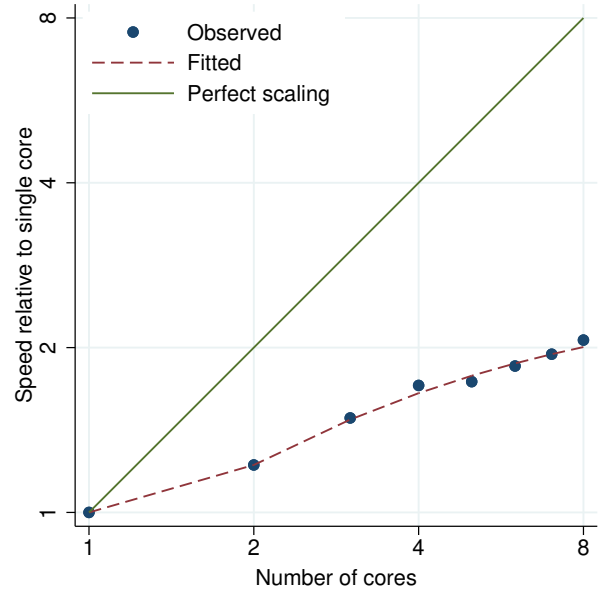


Figure 587. varnorm performance plot.

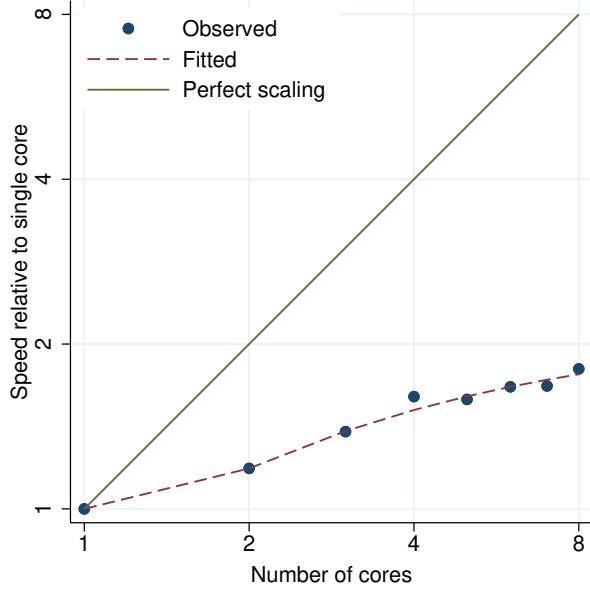


Figure 588. varsoc performance plot.

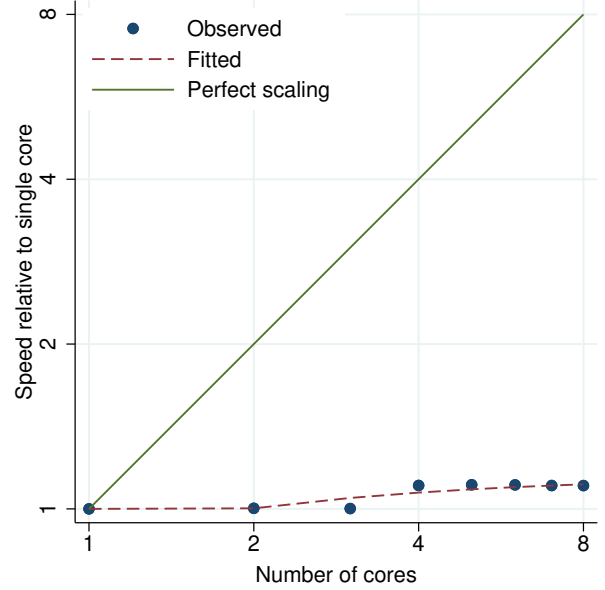


Figure 589. varstable performance plot.

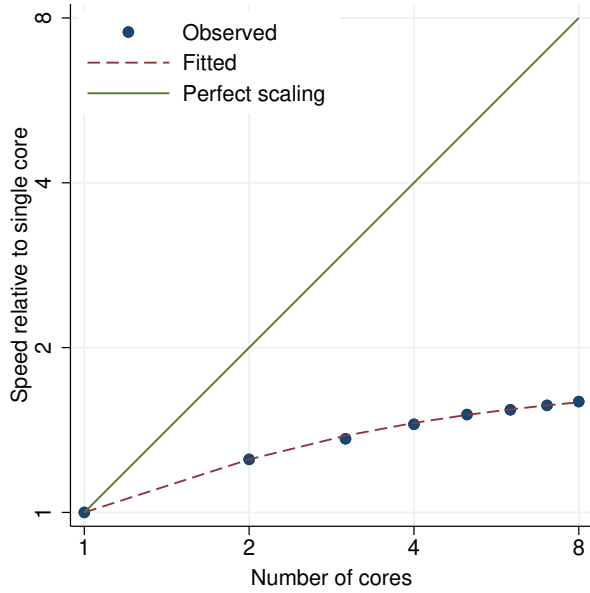


Figure 590. vec performance plot.

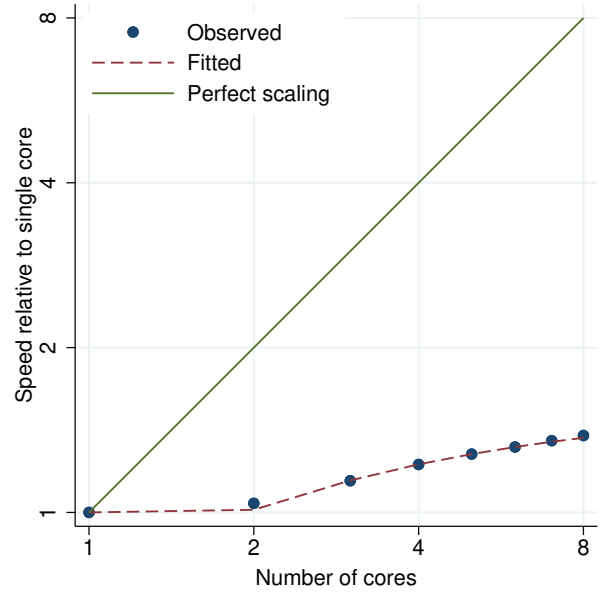


Figure 591. vec1mar performance plot.

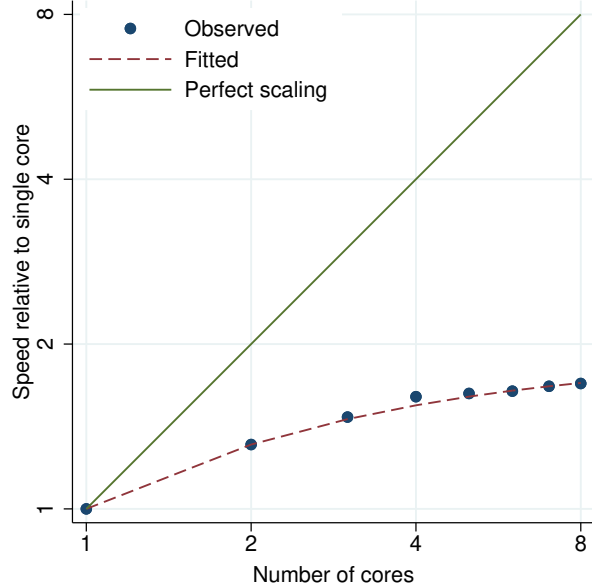


Figure 592. vecnorm performance plot.

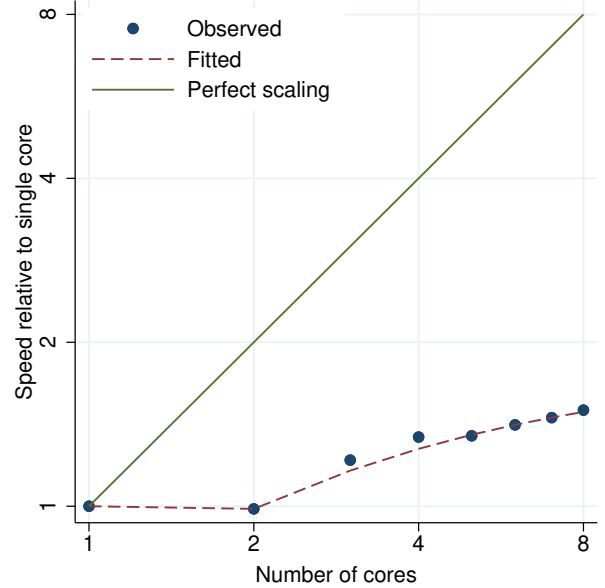


Figure 593. vecrank performance plot.

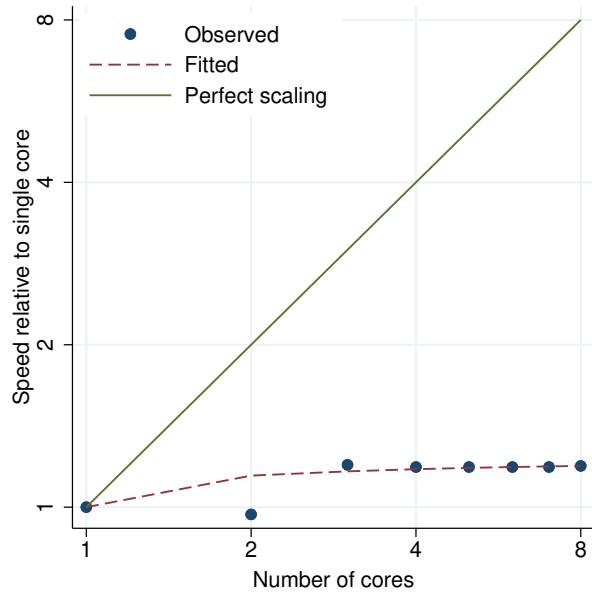


Figure 594. vecstable performance plot.

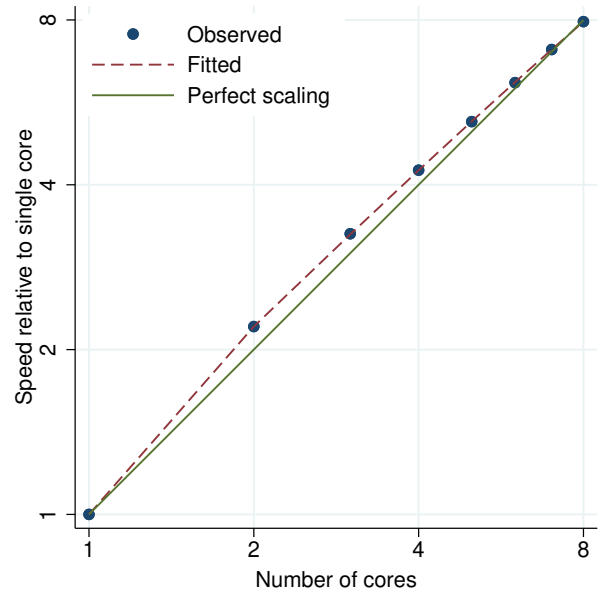


Figure 595. vwls performance plot.

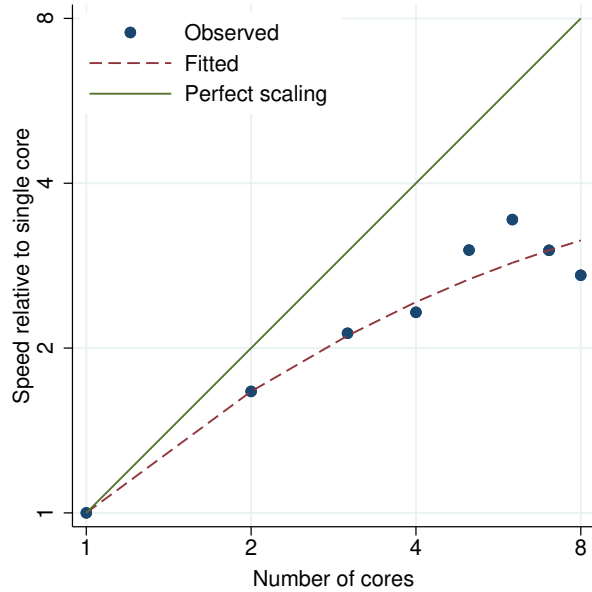


Figure 596. wildbootstrap areg performance plot.

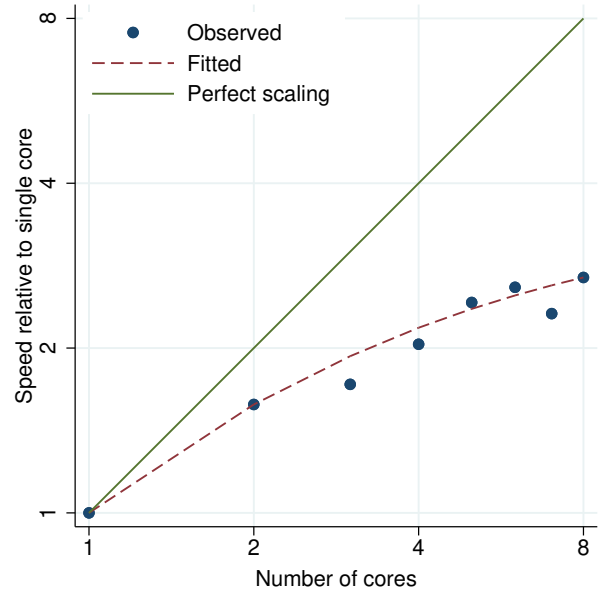


Figure 597. wildbootstrap regress performance plot.

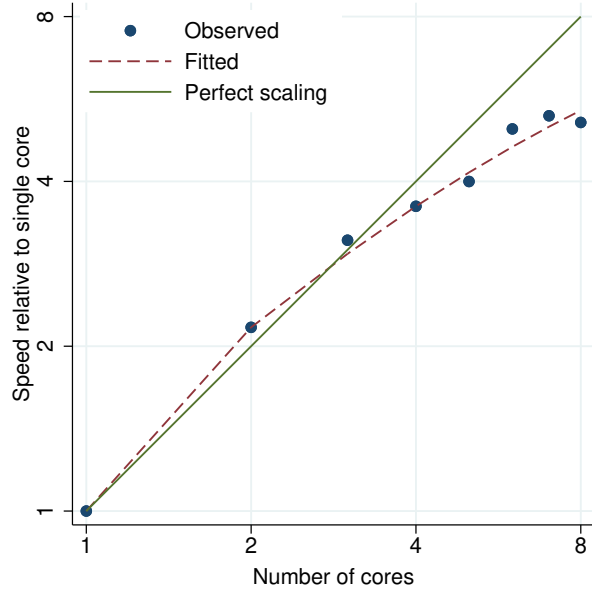


Figure 598. wildbootstrap xtreg performance plot.

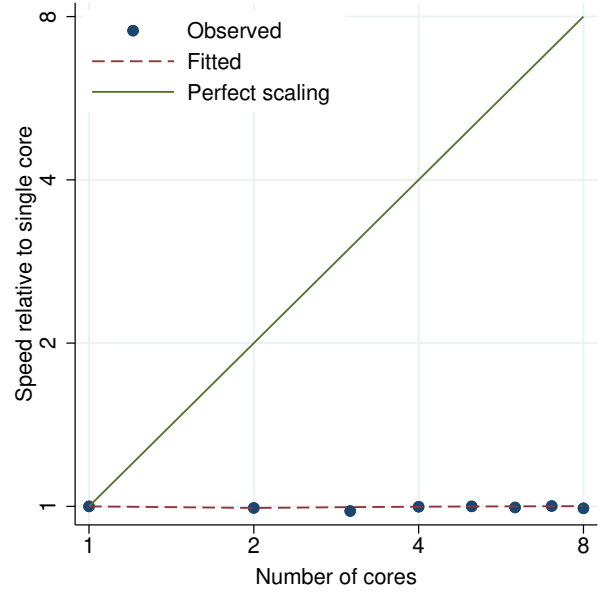


Figure 599. wntestb performance plot.

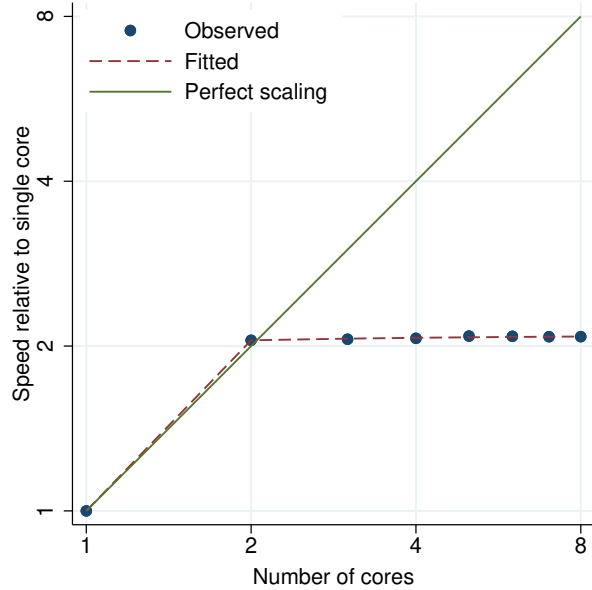


Figure 600. wntestq performance plot.

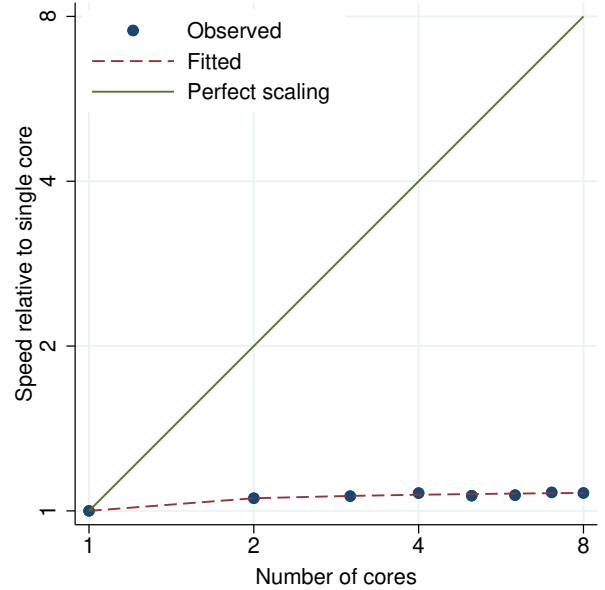


Figure 601. xcorr performance plot.

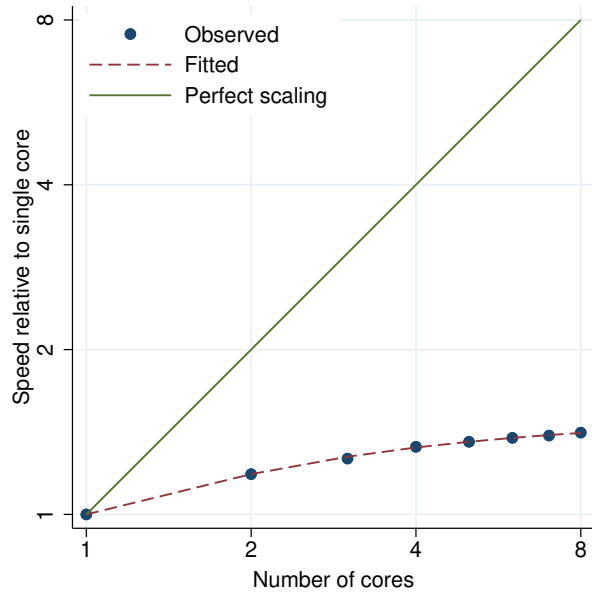


Figure 602. xplogit performance plot.

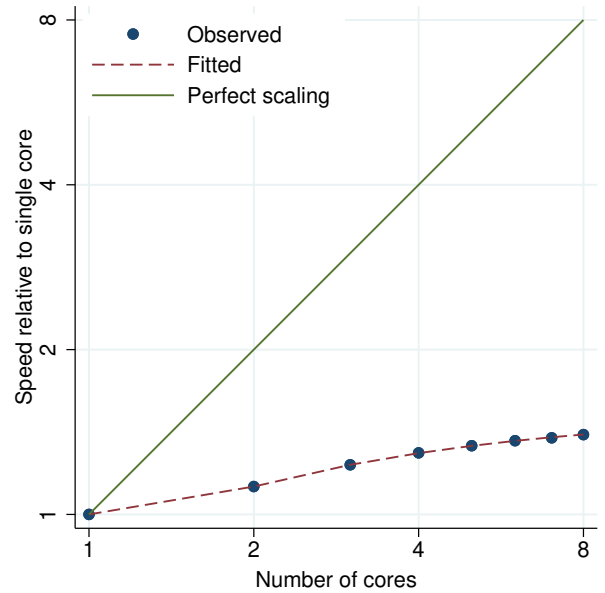


Figure 603. xpoisson performance plot.

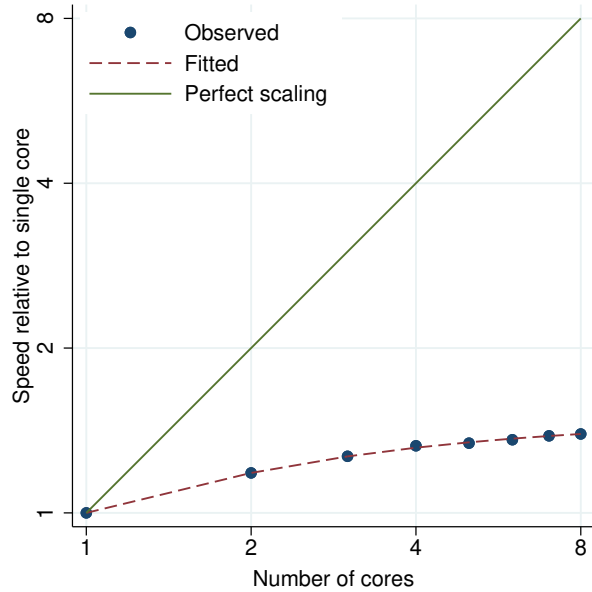


Figure 604. xppuregress performance plot.

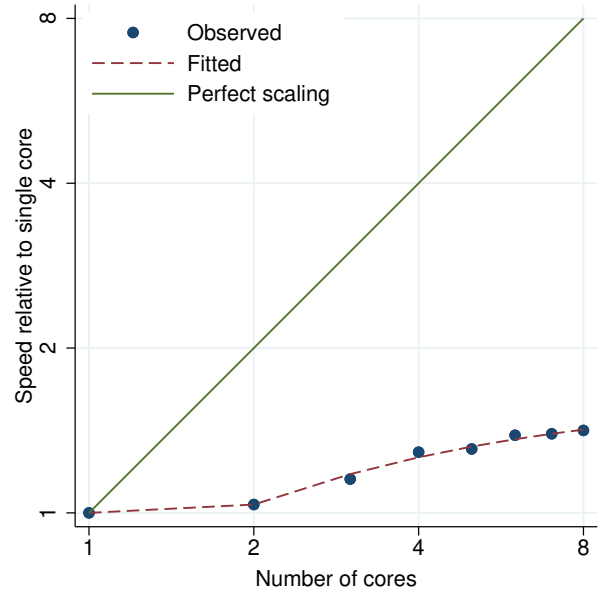


Figure 605. xtabond performance plot.

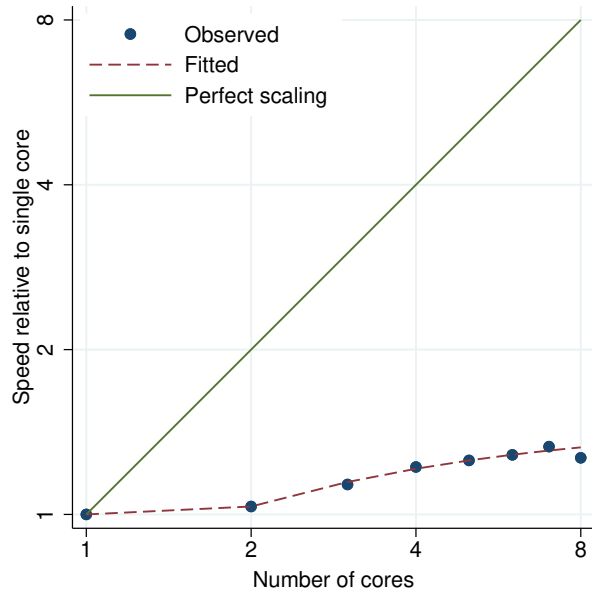


Figure 606. xtabond, twostep performance plot.

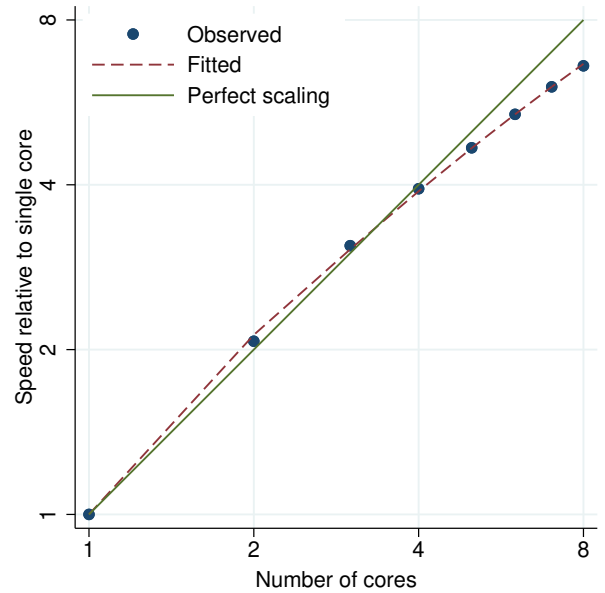


Figure 607. xtcloglog, re performance plot.

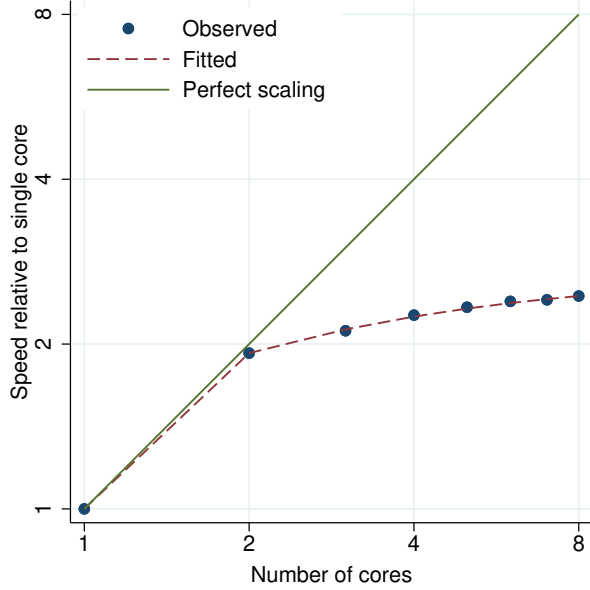


Figure 608. xtdata, be performance plot.

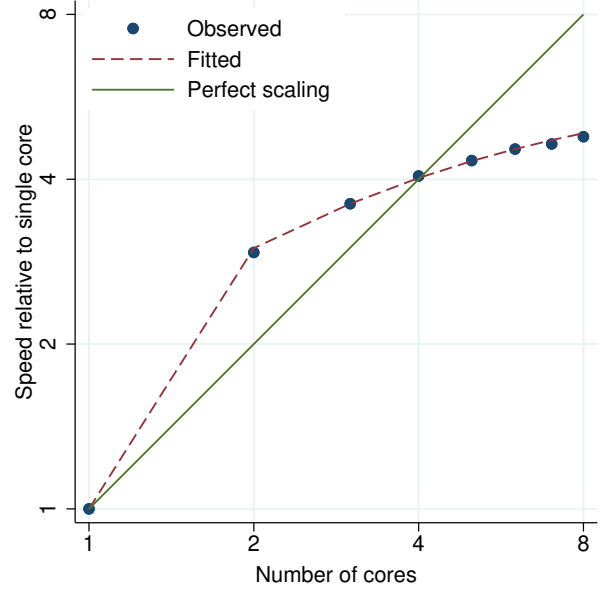


Figure 609. xtdata, fe performance plot.

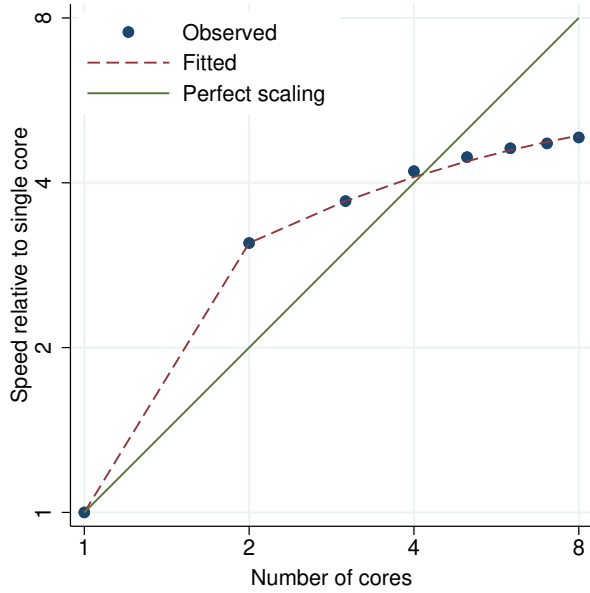


Figure 610. xtdata, re performance plot.

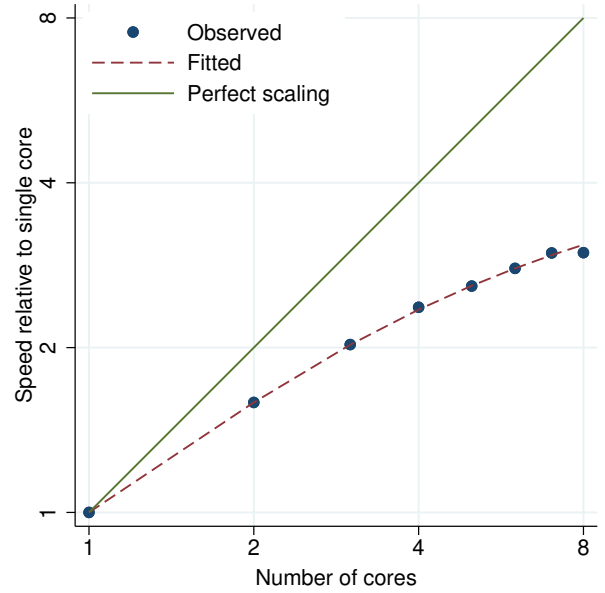


Figure 611. xtdidregress performance plot.

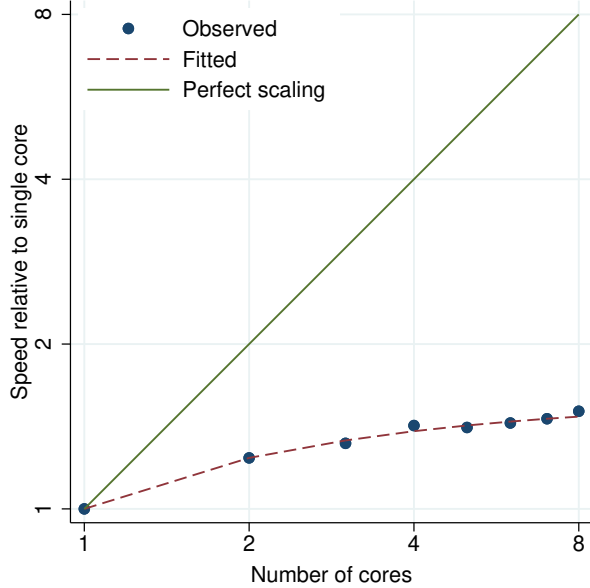


Figure 612. xtdpd performance plot.

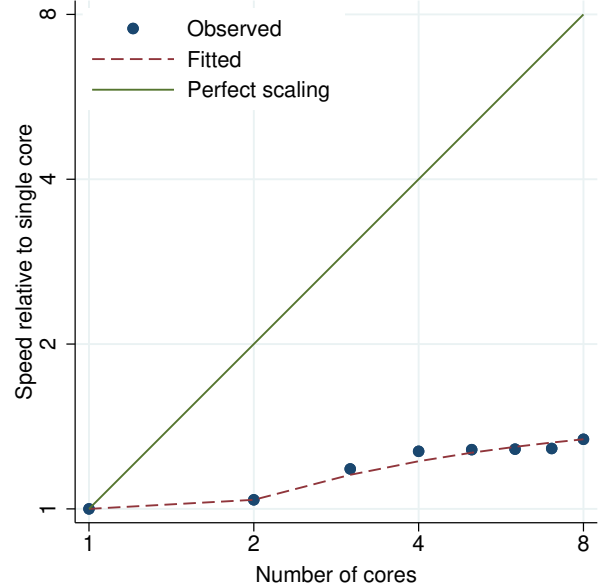


Figure 613. xtdpdsys performance plot.

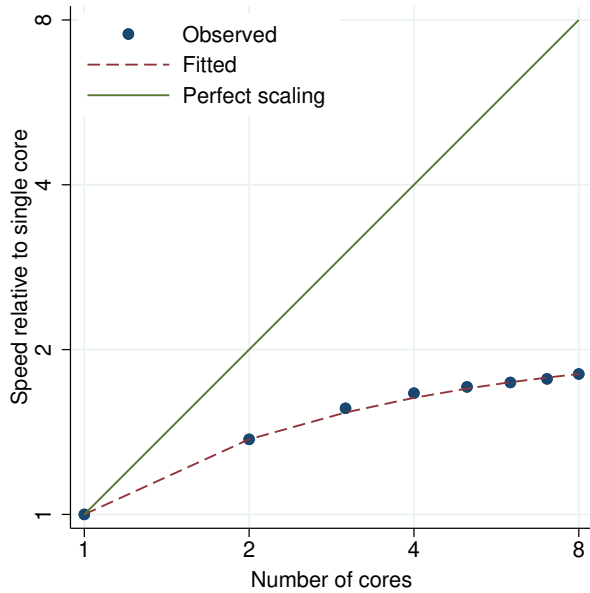


Figure 614. xtregress performance plot.

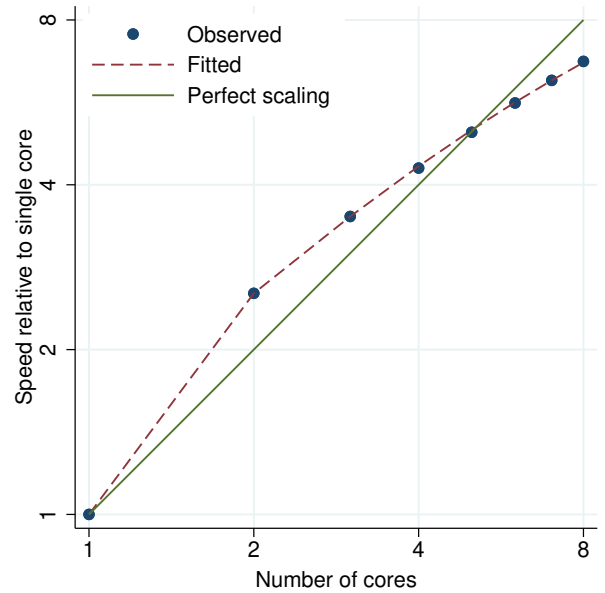


Figure 615. xtfrontier performance plot.

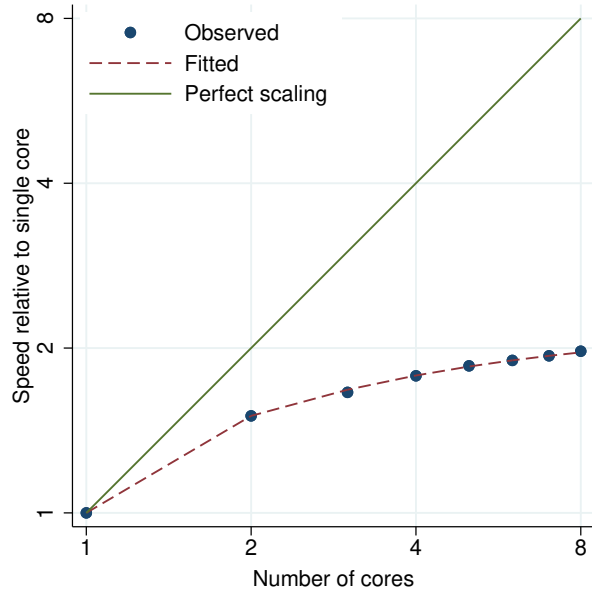


Figure 616. `xtgee, family(gaussian) corr(ar2)` performance plot.

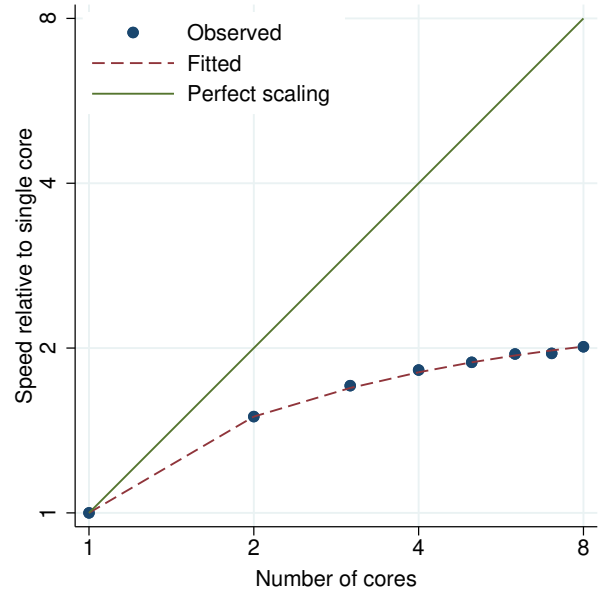


Figure 617. `xtgee, fam(gauss) corr(unstruct)` performance plot.

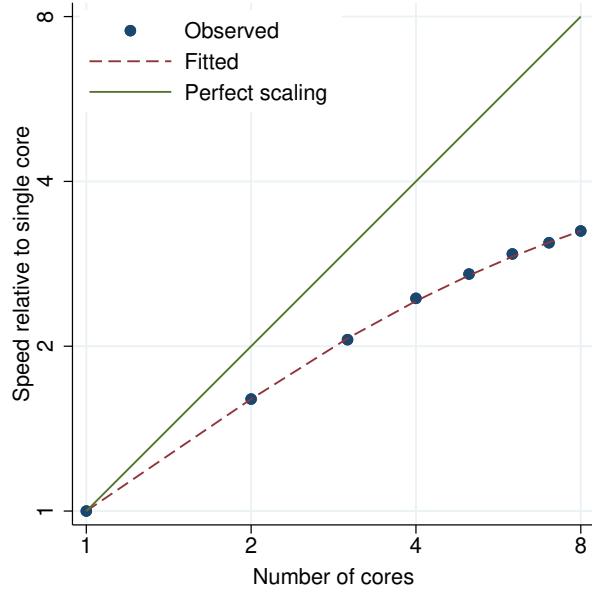


Figure 618. `xtcloglog, pa` performance plot.

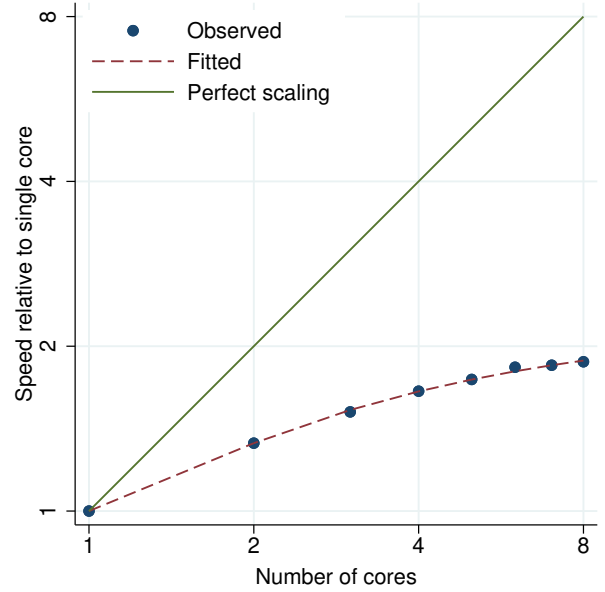


Figure 619. `xtlogit, pa` performance plot.

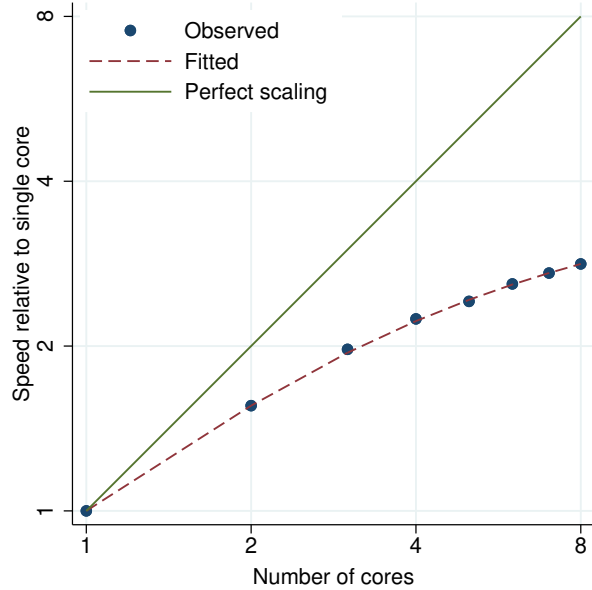


Figure 620. `xtnbreg`, pa performance plot.

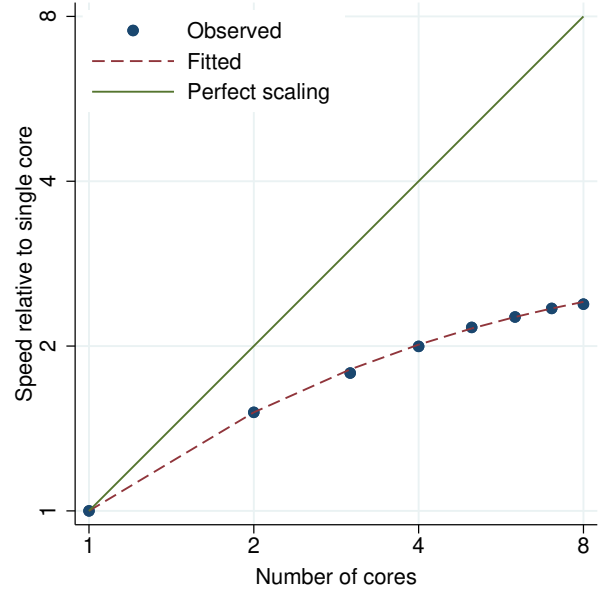


Figure 621. `xtpoisson`, pa performance plot.

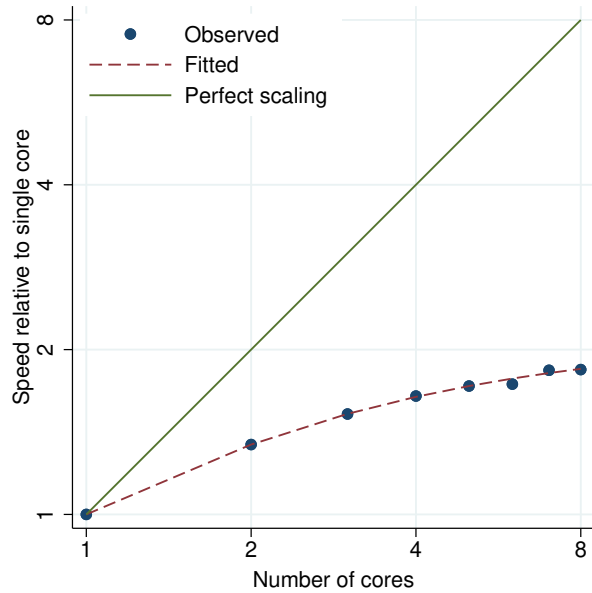


Figure 622. `xtprobit`, pa performance plot.

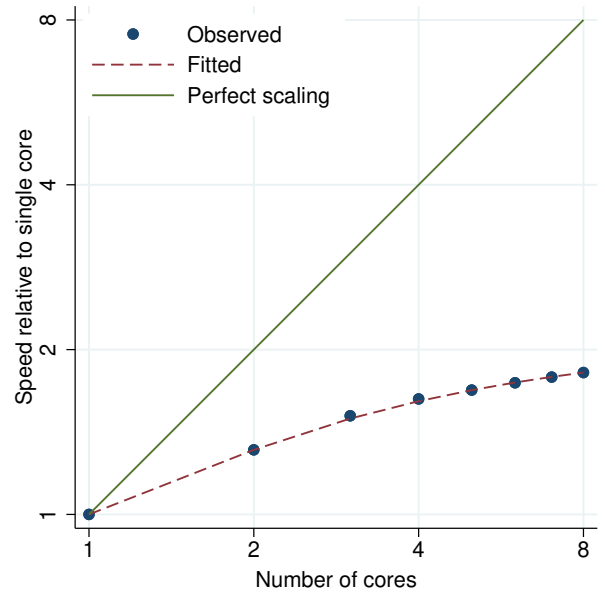


Figure 623. `xtreg`, pa performance plot.

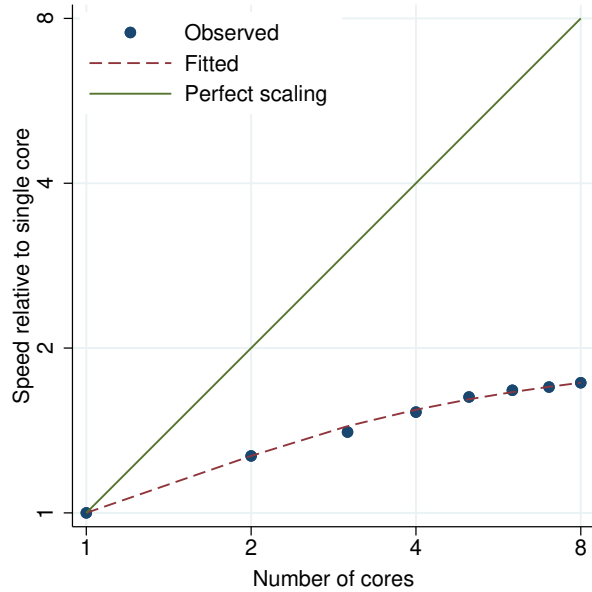


Figure 624. `xtglm` performance plot.

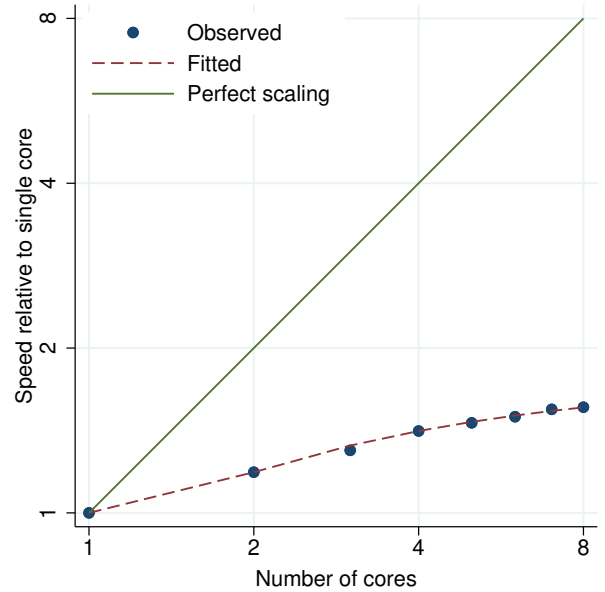


Figure 625. `xthdidregress aipw` performance plot.

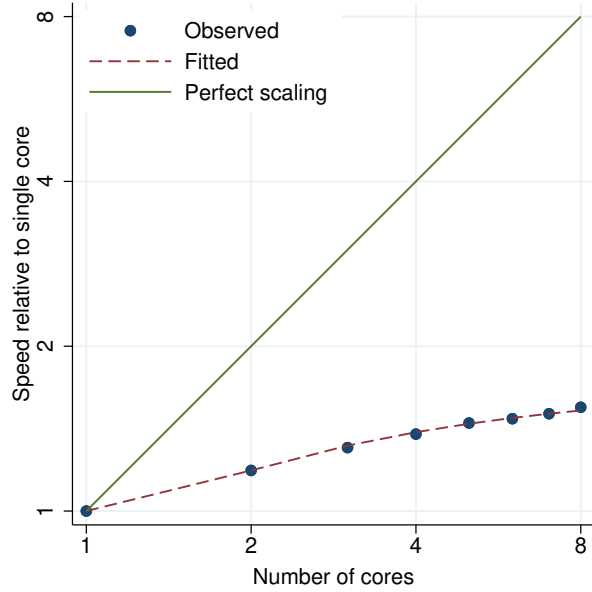


Figure 626. `xthdidregress ipw` performance plot.

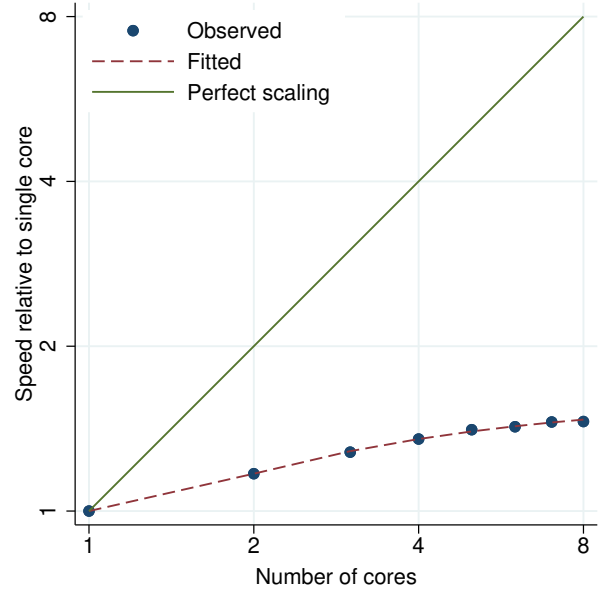


Figure 627. `xthdidregress ra` performance plot.

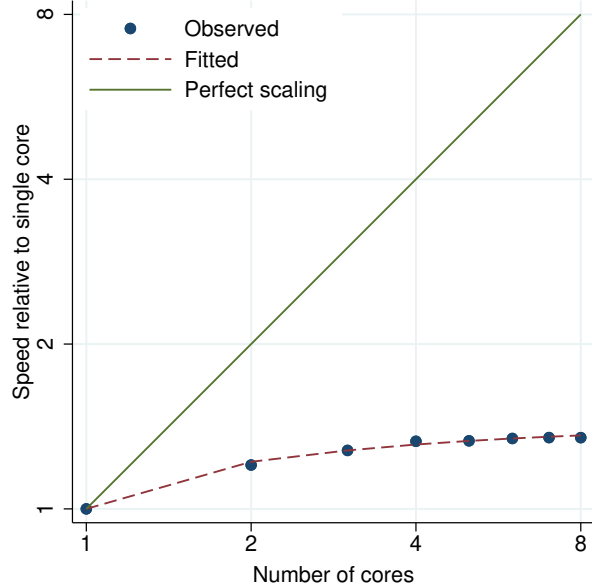


Figure 628. xthdidregress twfe performance plot.

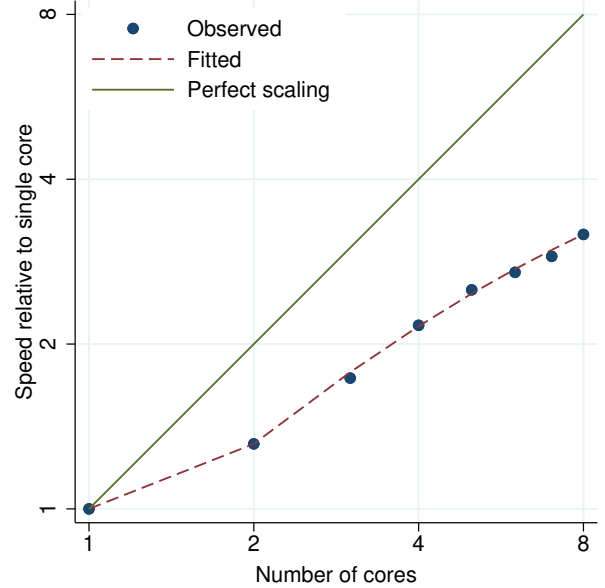


Figure 629. xthtaylor performance plot.

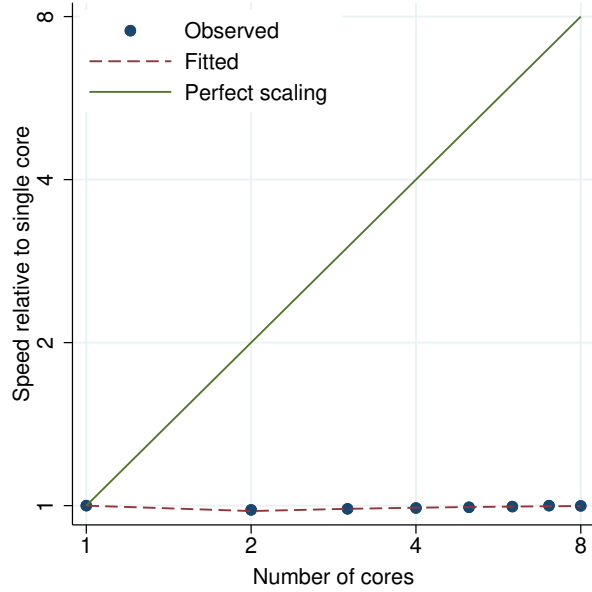


Figure 630. xtile performance plot.

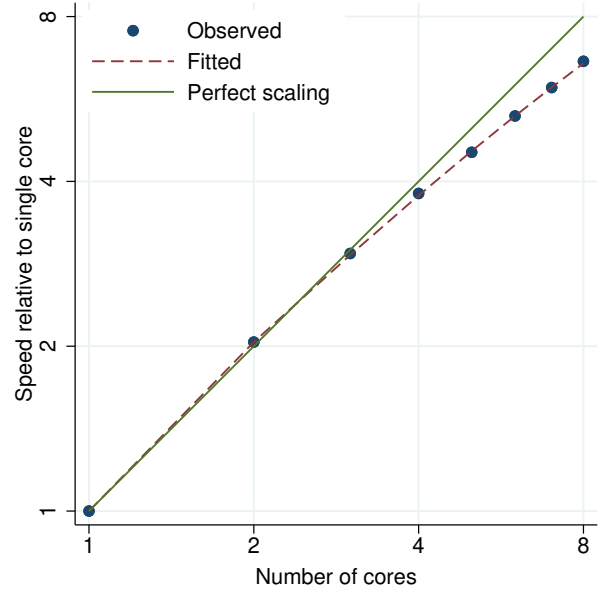


Figure 631. xtintreg performance plot.

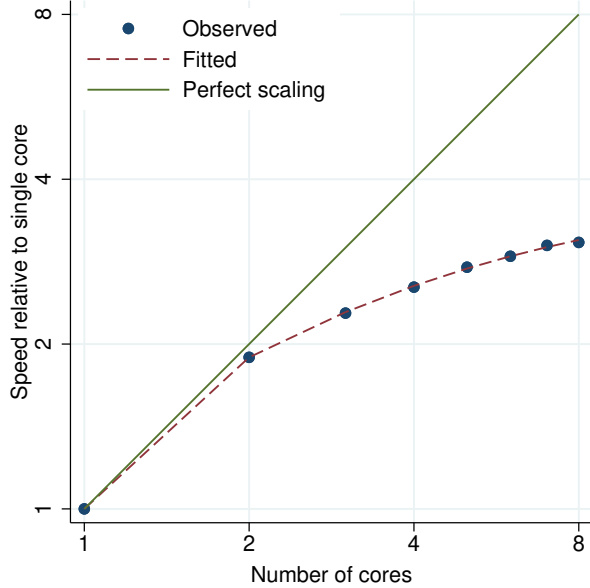


Figure 632. xtivreg, be performance plot.

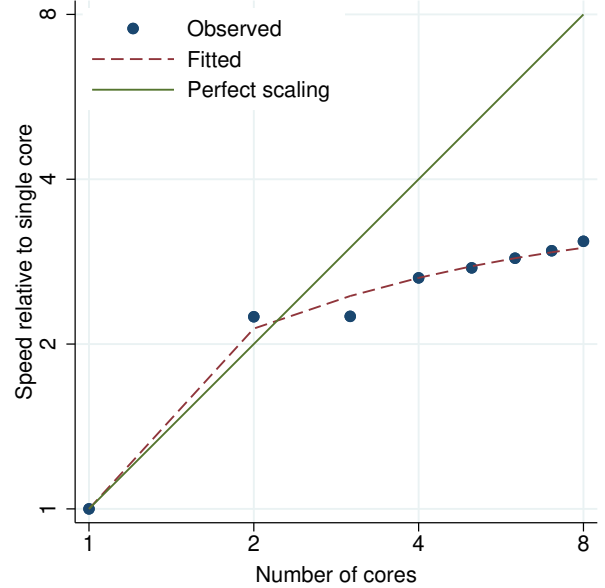


Figure 633. xtivreg, fd performance plot.

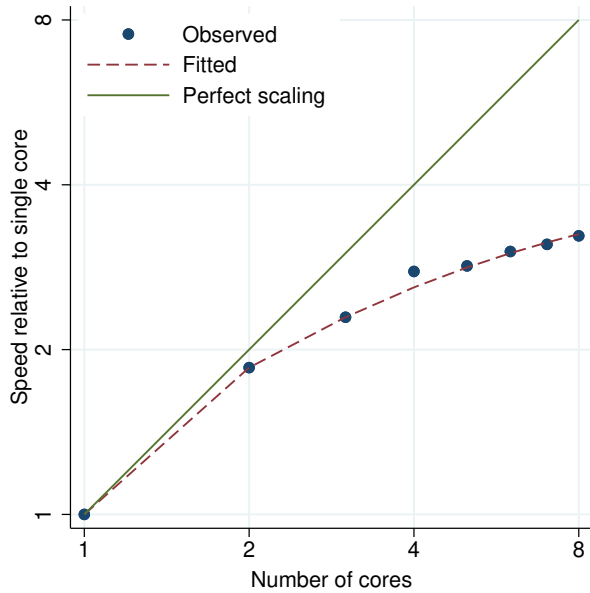


Figure 634. xtivreg, fe performance plot.

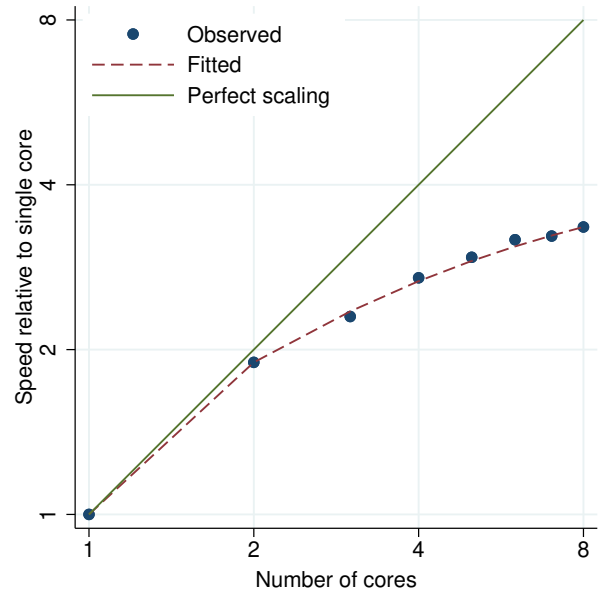


Figure 635. xtivreg, re performance plot.

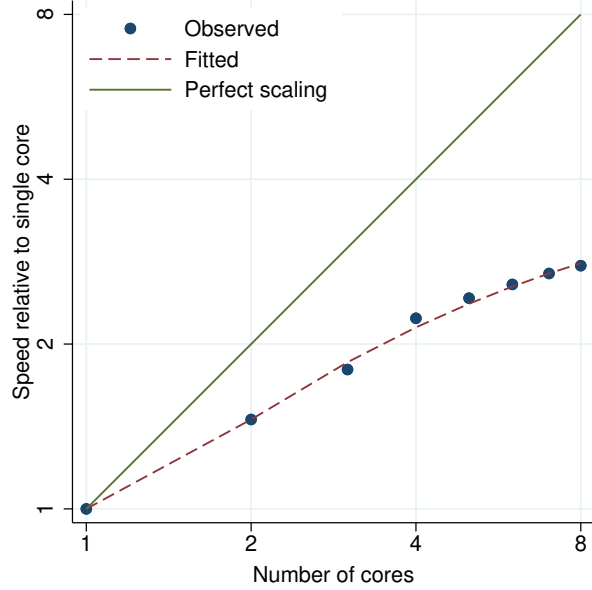


Figure 636. xtlogit, fe performance plot.

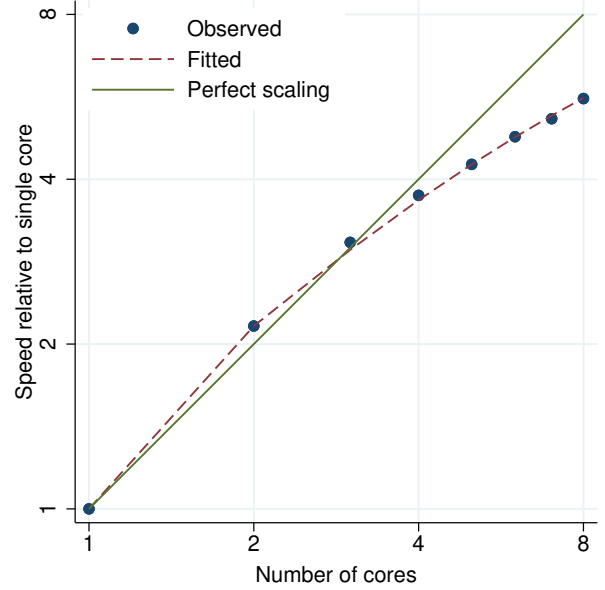


Figure 637. xtlogit, re performance plot.

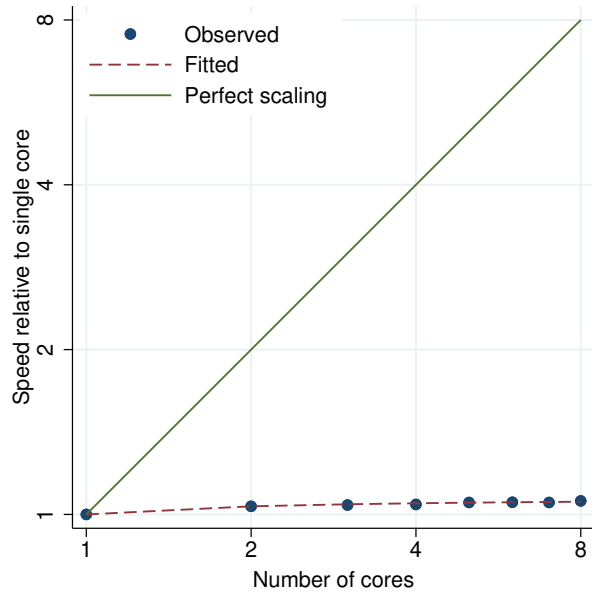


Figure 638. xtmlogit, fe performance plot.

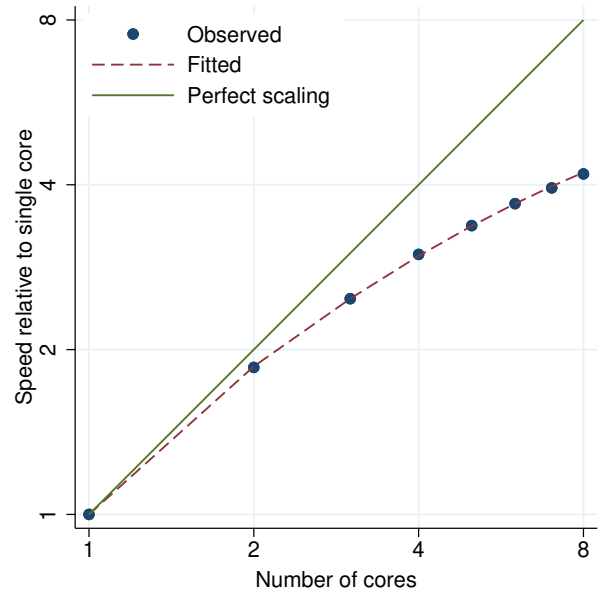


Figure 639. xtmlogit, re performance plot.

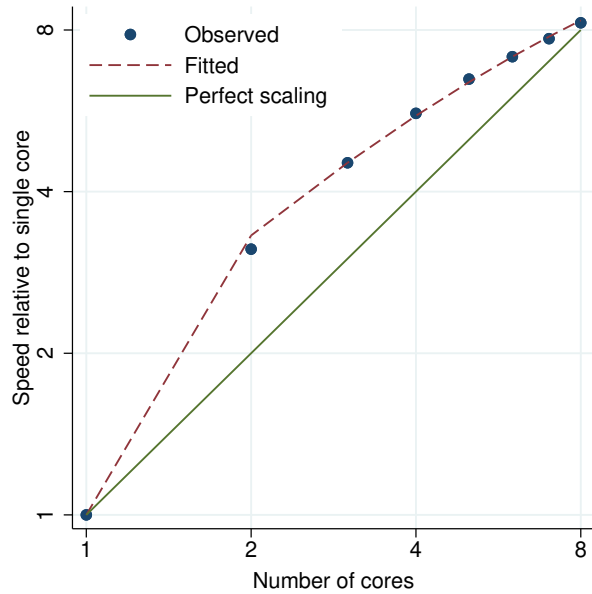


Figure 640. xtnbreg, fe performance plot.

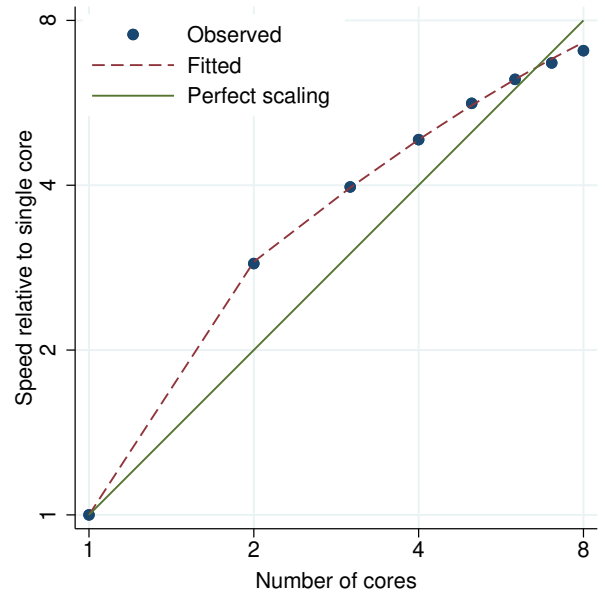


Figure 641. xtnbreg, re performance plot.

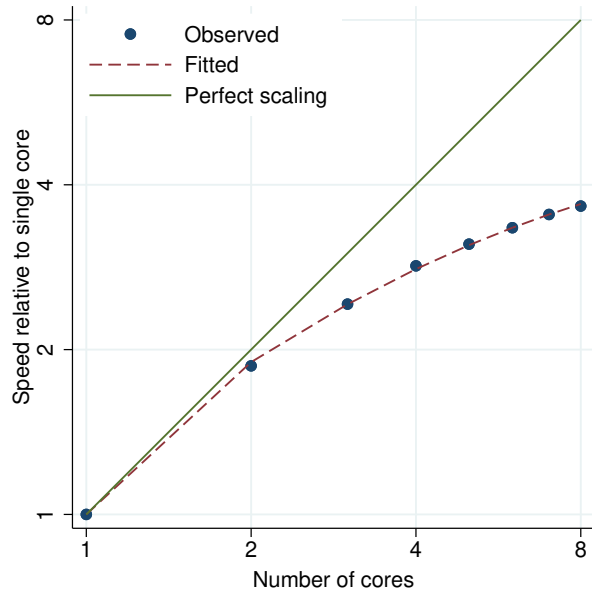


Figure 642. xtologit performance plot.

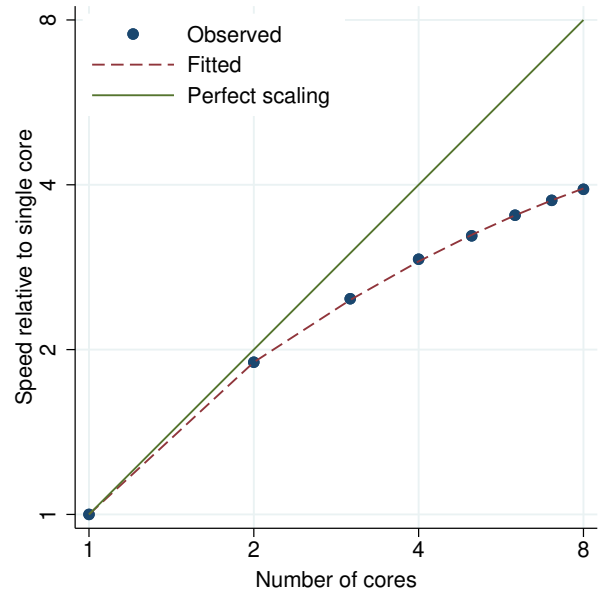


Figure 643. xtoprobit performance plot.

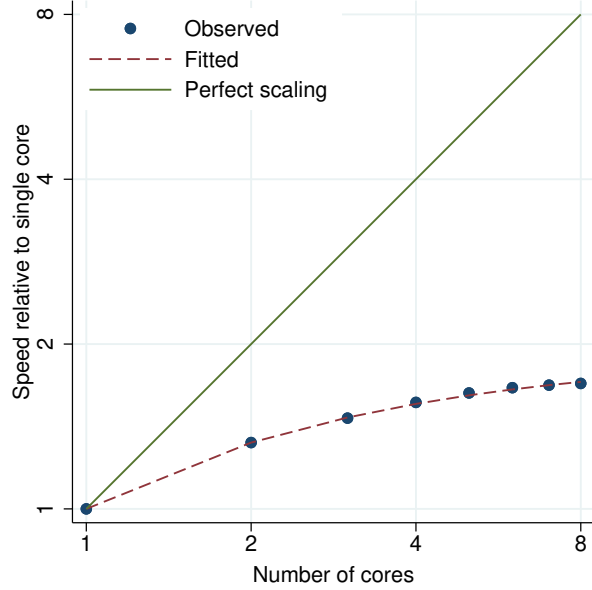


Figure 644. xtpcse performance plot.

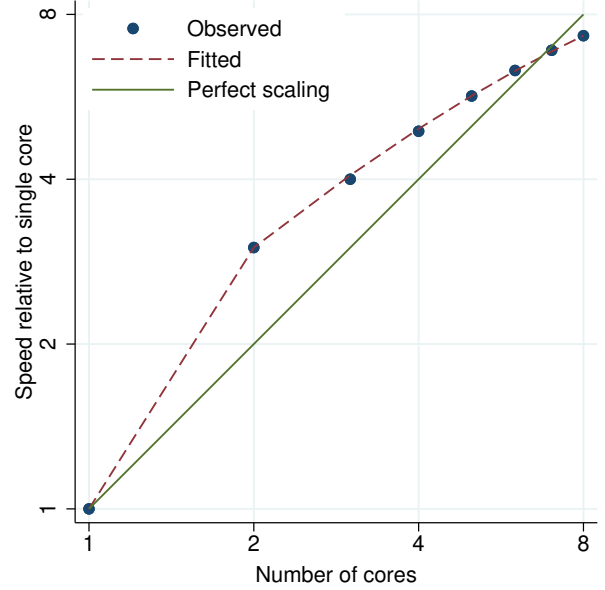


Figure 645. xtppoisson, fe performance plot.

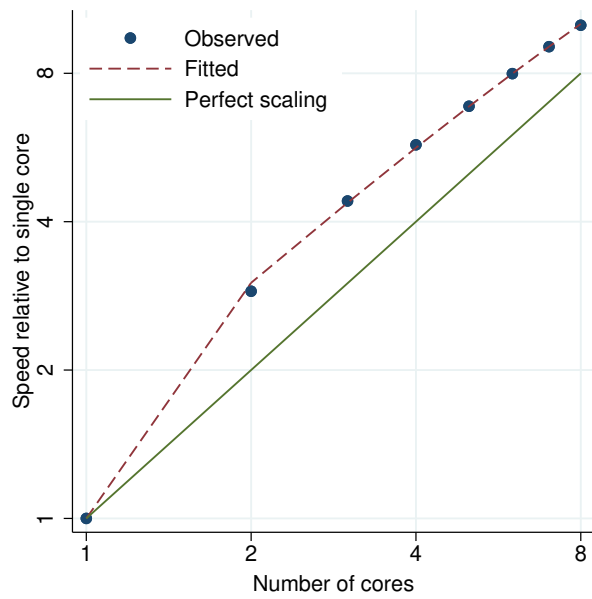


Figure 646. xtppoisson, re performance plot.

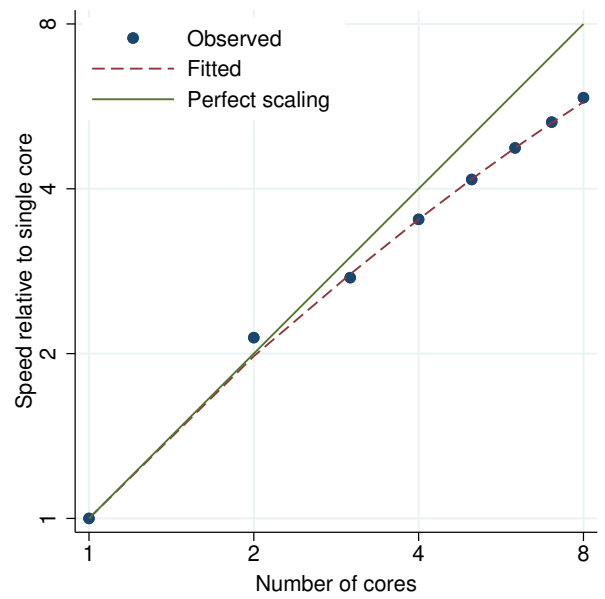


Figure 647. xtprobit, re performance plot.

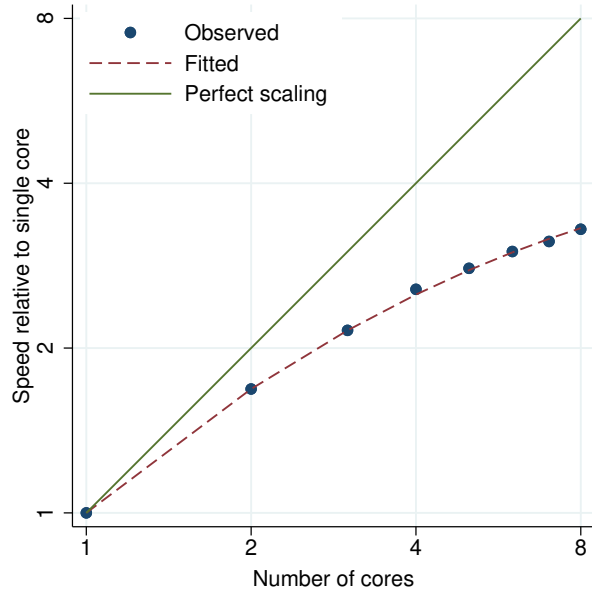


Figure 648. xtrc performance plot.

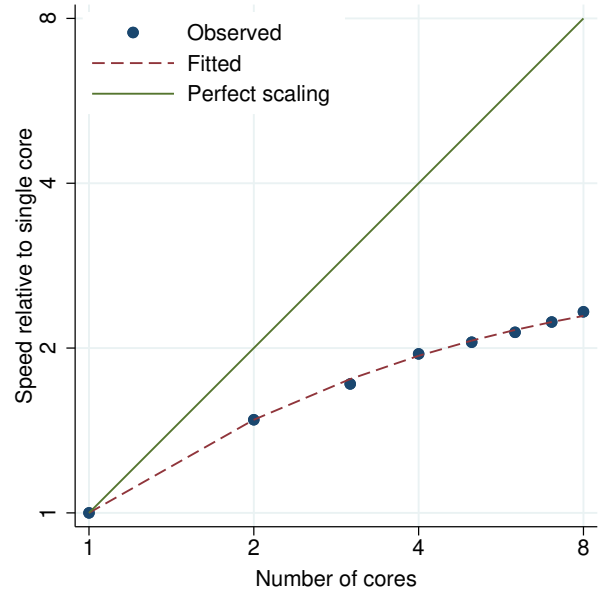


Figure 649. xtreg, be performance plot.

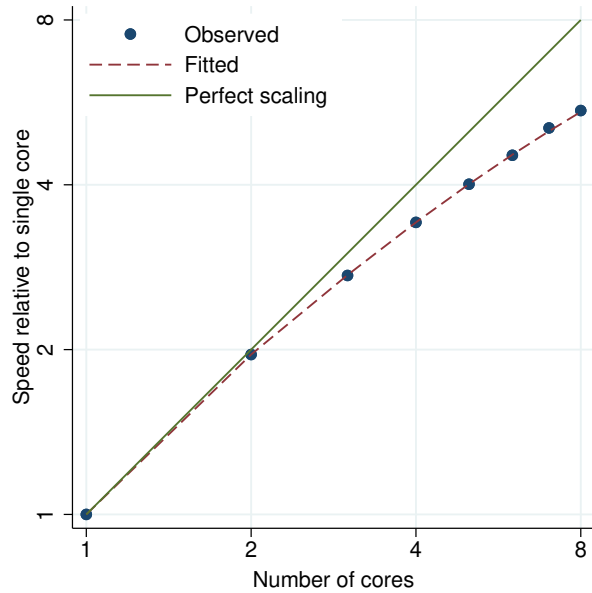


Figure 650. xtreg, fe performance plot.

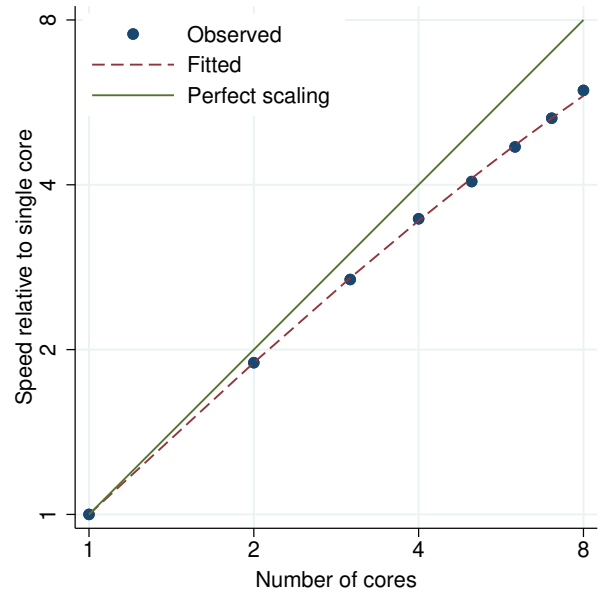


Figure 651. xtreg, fe vce(robust) performance plot.

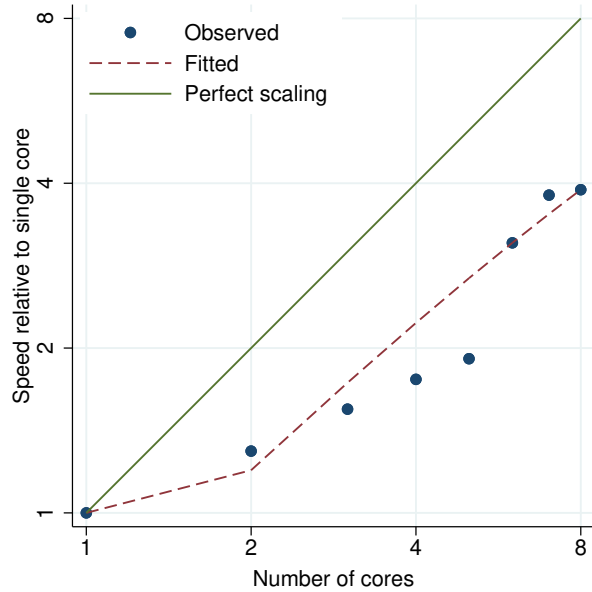


Figure 652. xtreg, mle performance plot.

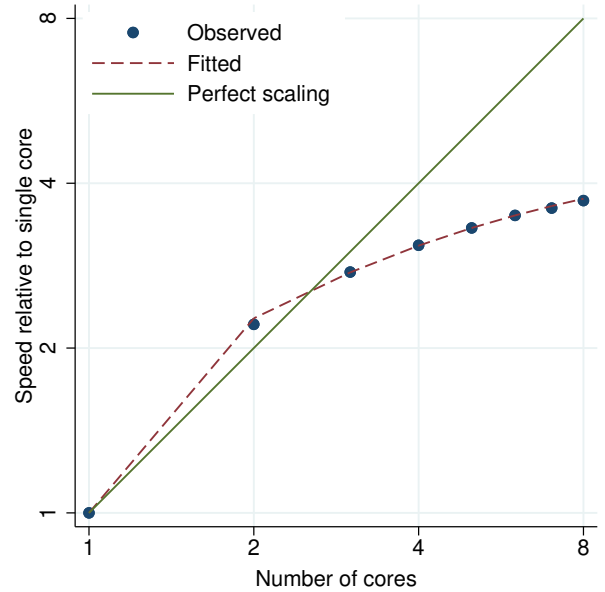


Figure 653. xtreg, re performance plot.

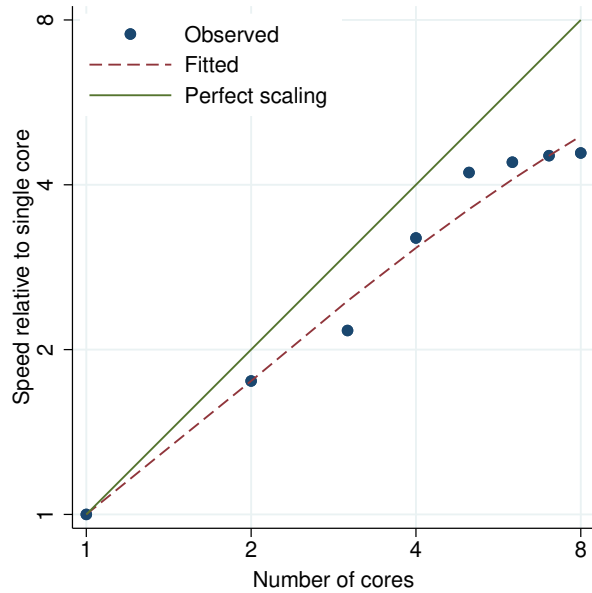


Figure 654. xtregar, fe performance plot.

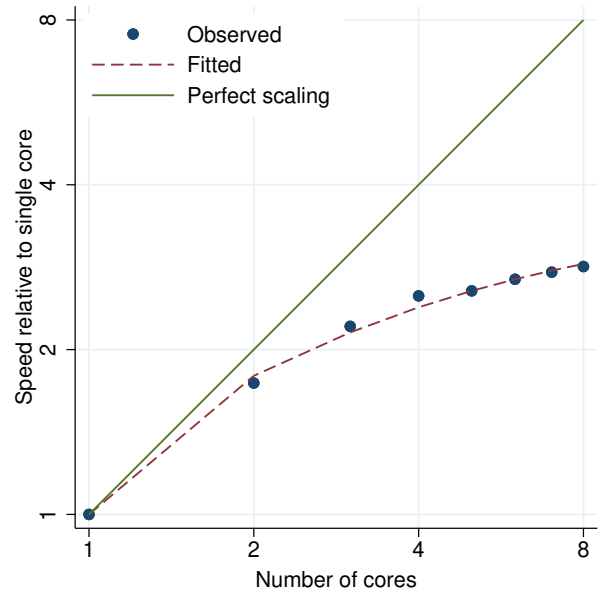


Figure 655. xtregar, re performance plot.

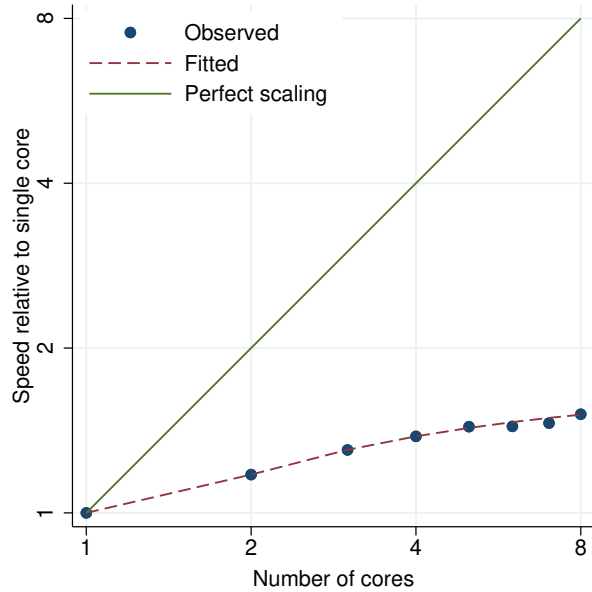


Figure 656. `xtset` performance plot.

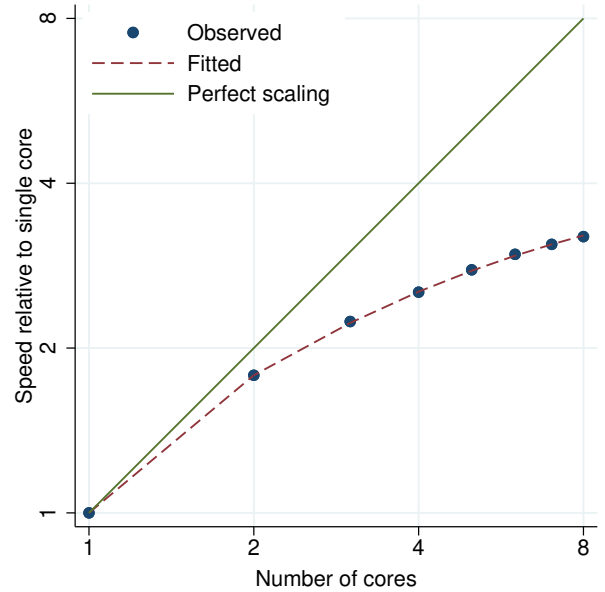


Figure 657. `xtstreg, distribution(exponential)` performance plot.

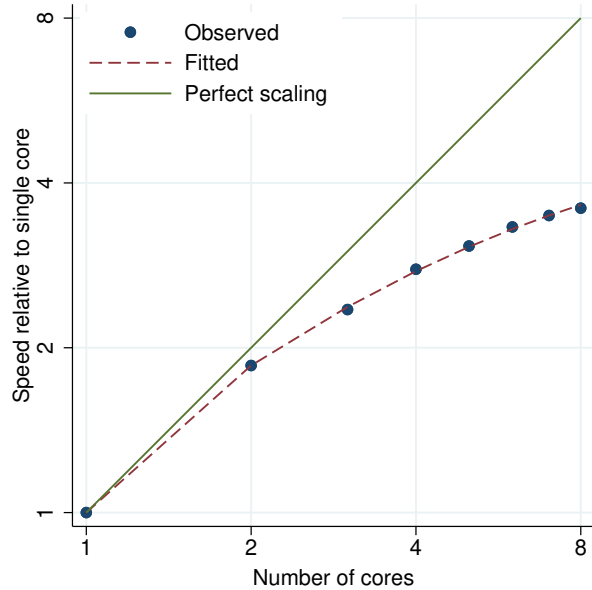


Figure 658. `xtstreg, distribution(weibull)` performance plot.

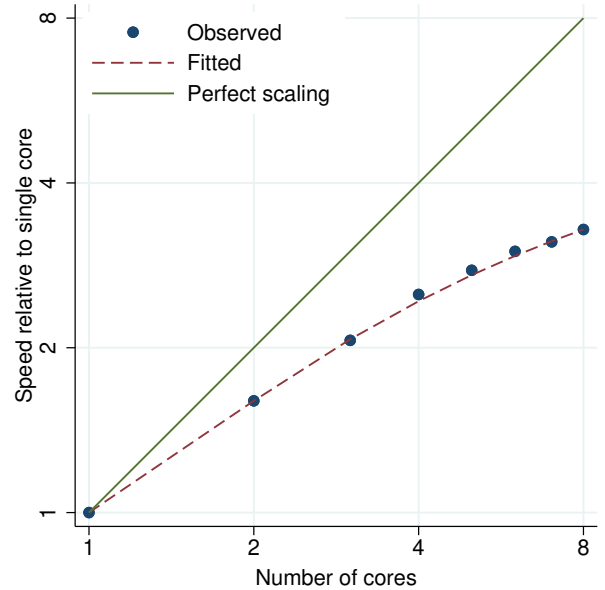


Figure 659. `xtsum` performance plot.

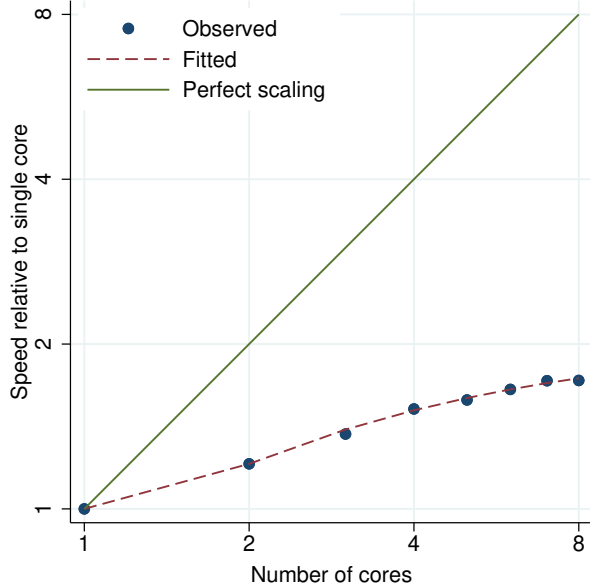


Figure 660. xttab performance plot.

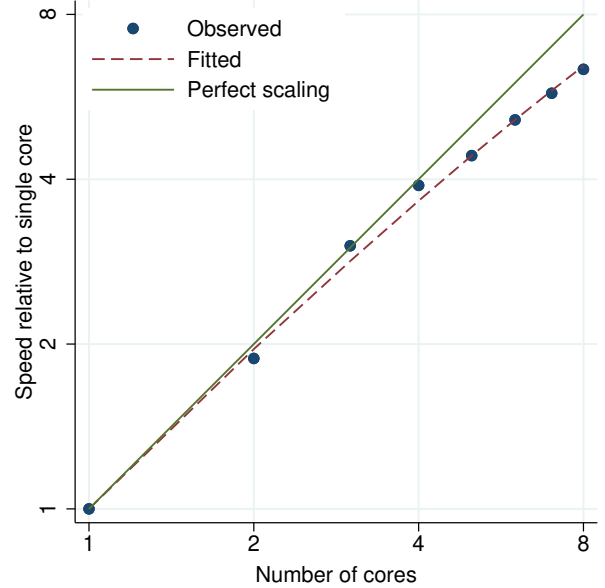


Figure 661. xttoibit performance plot.

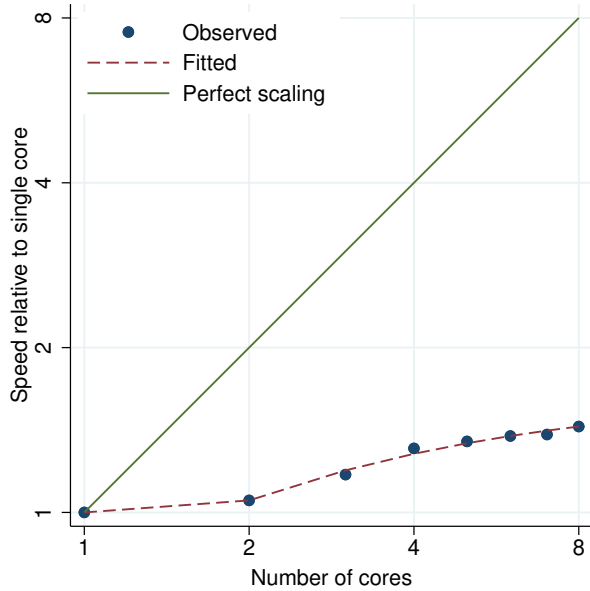


Figure 662. xtunitroot breitung performance plot.

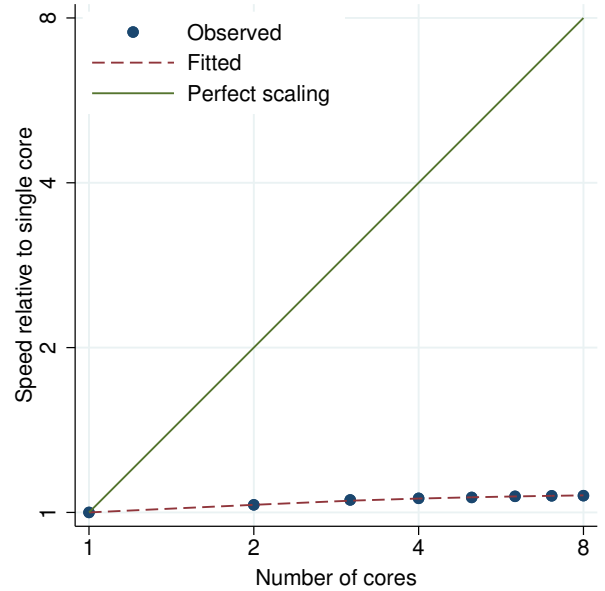


Figure 663. xtunitroot fisher performance plot.

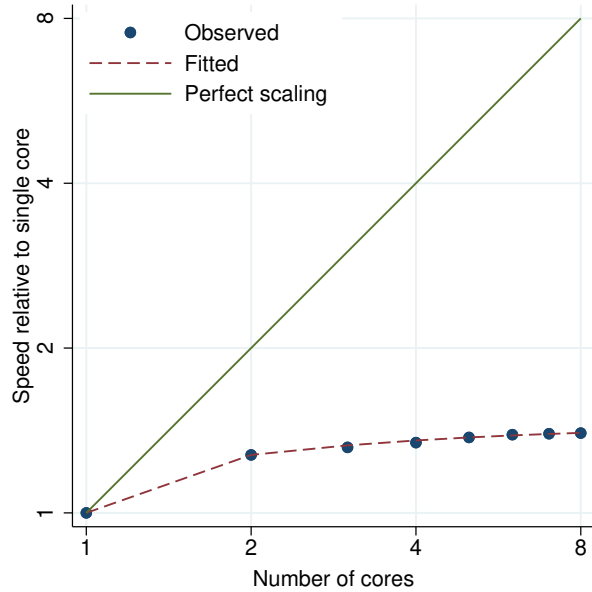


Figure 664. xtunitroot hadri performance plot.

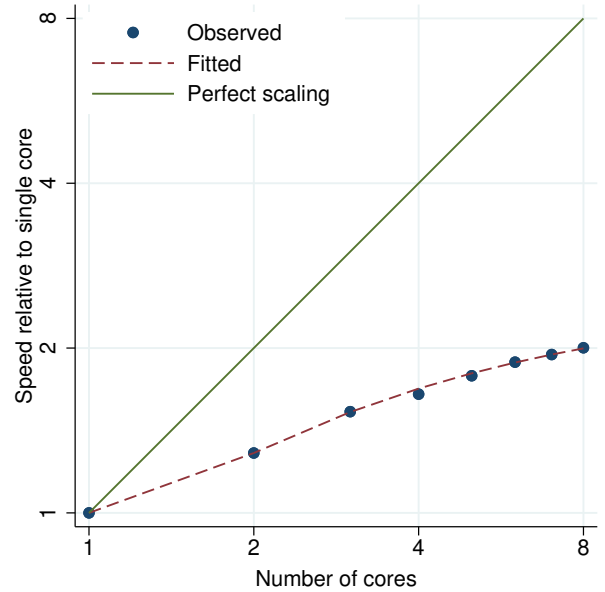


Figure 665. xtunitroot ht performance plot.

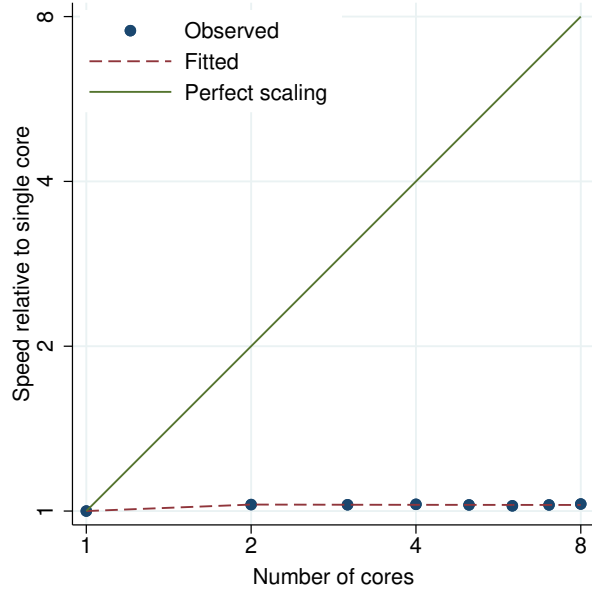


Figure 666. xtunitroot ips performance plot.

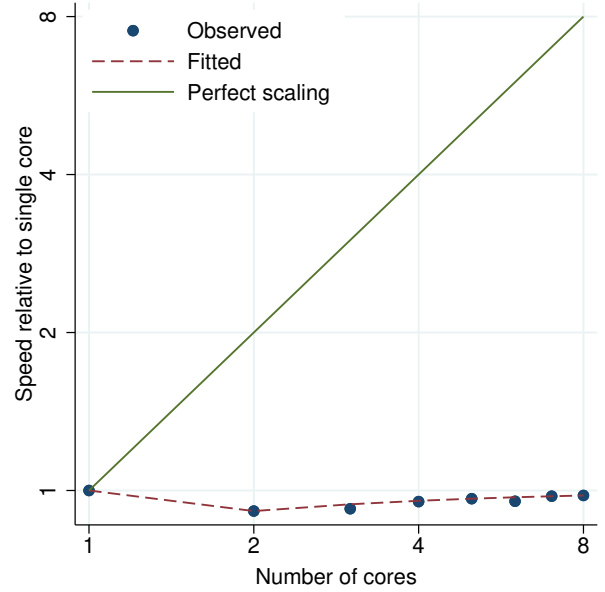


Figure 667. xtunitroot llc performance plot.

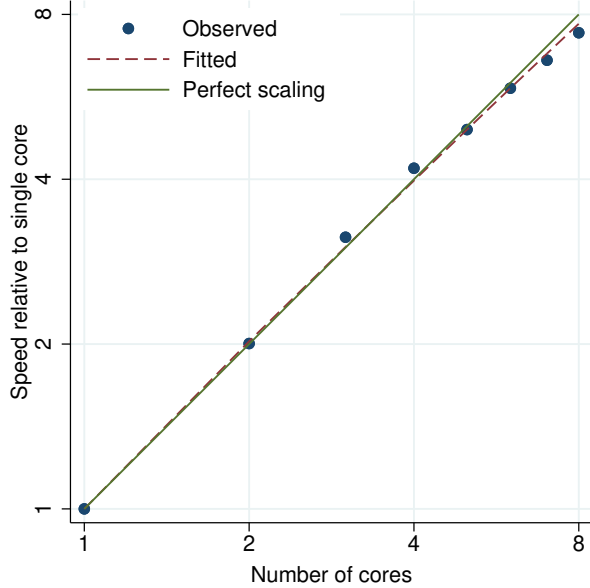


Figure 668. zinb performance plot.

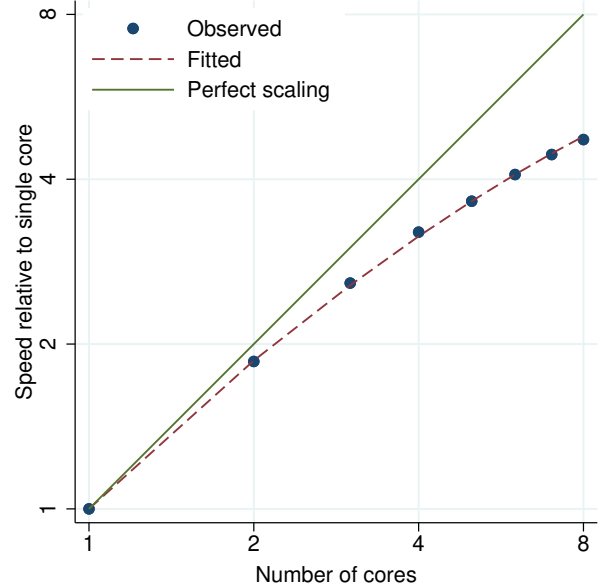


Figure 669. zilogit performance plot.

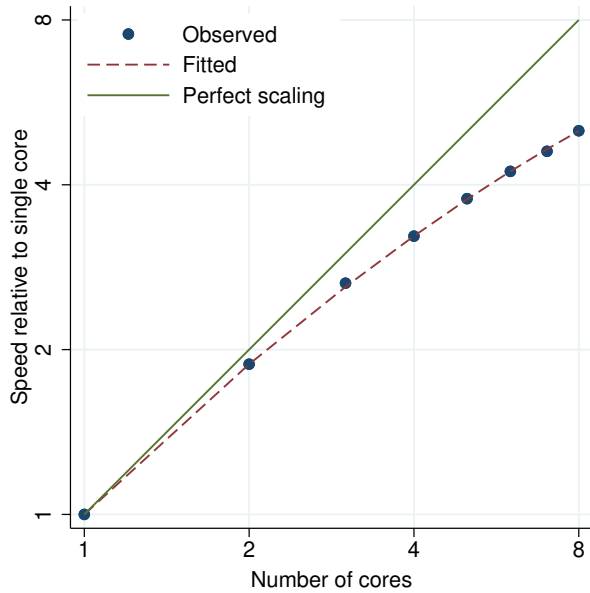


Figure 670. zioprobit performance plot.

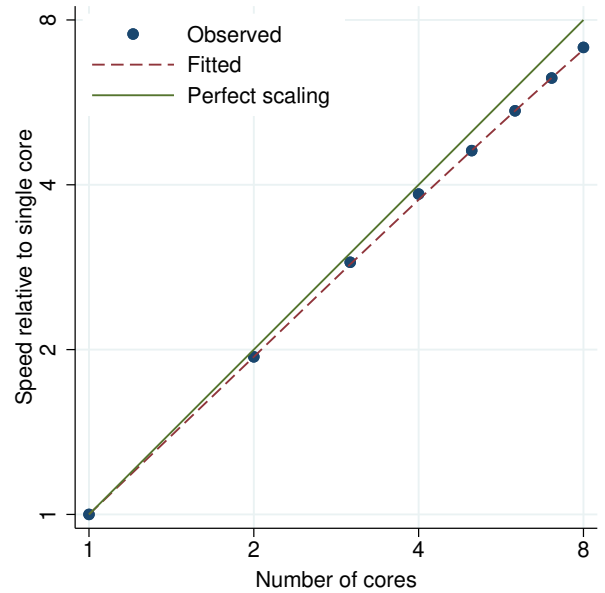


Figure 671. zip performance plot.

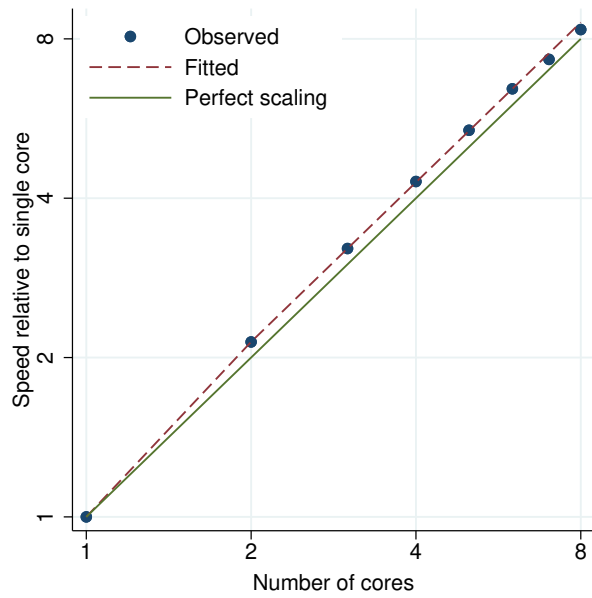


Figure 672. `_predict, xb` performance plot.

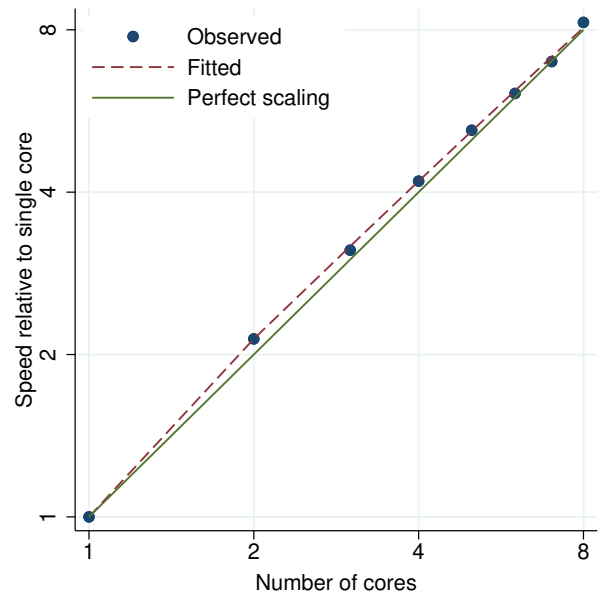


Figure 673. `_rmcoll` performance plot.

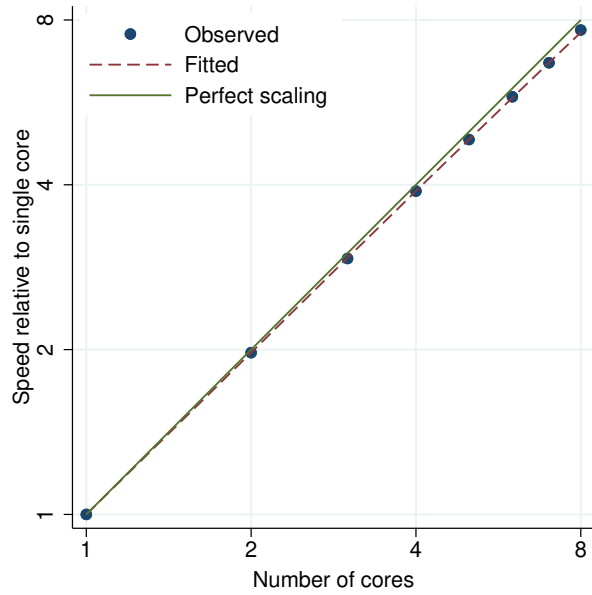


Figure 674. `_robust` performance plot.

B Performance assessment graphs for high-end servers

Performance graphs of all 655 commands running on high-end servers are presented below.

These graphs are similar to the graphs from appendix [A](#) except that here the speeds are evaluated up to 40 cores.

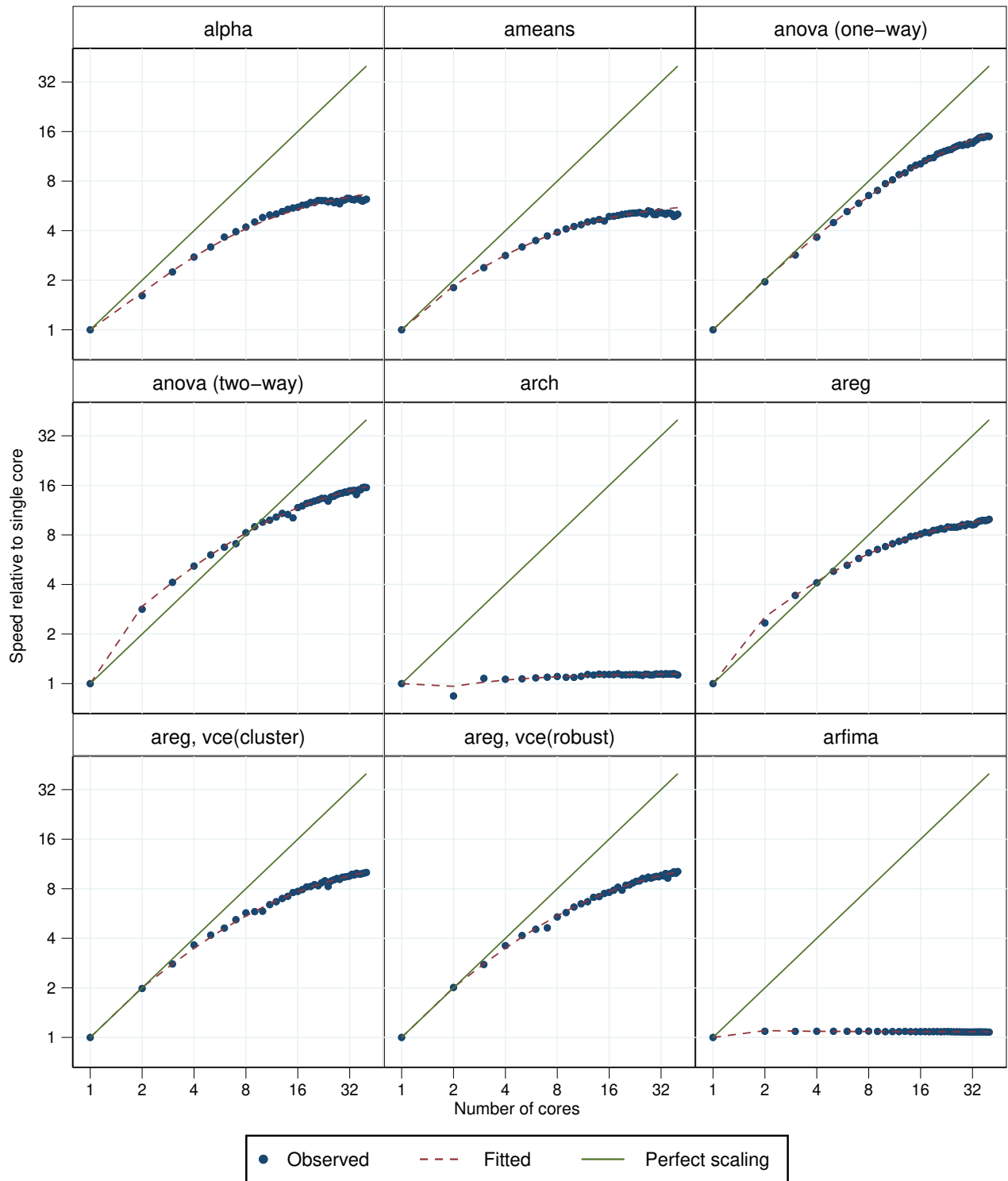


Figure 675. Parallelization performance plots.

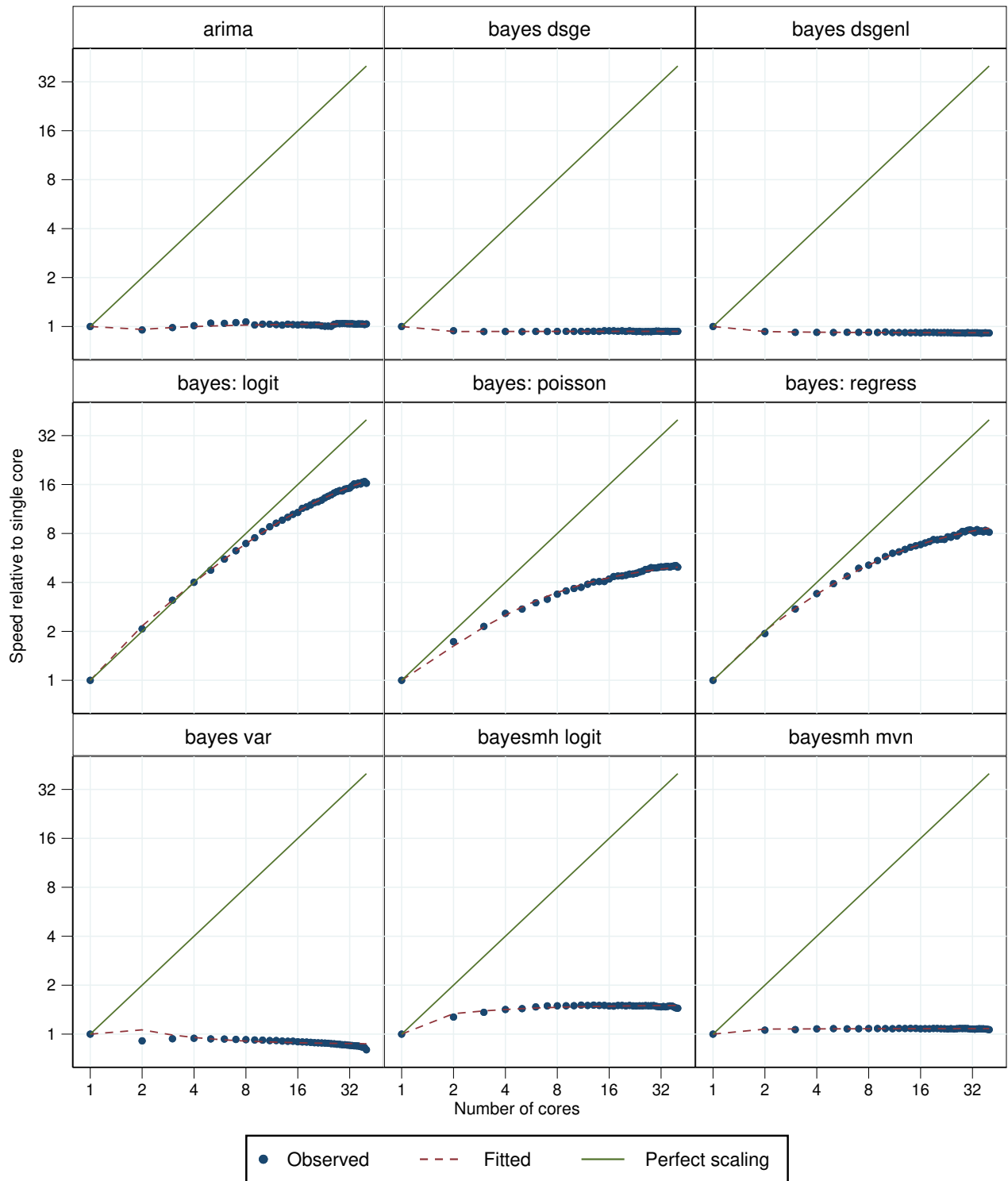


Figure 676. Parallelization performance plots.

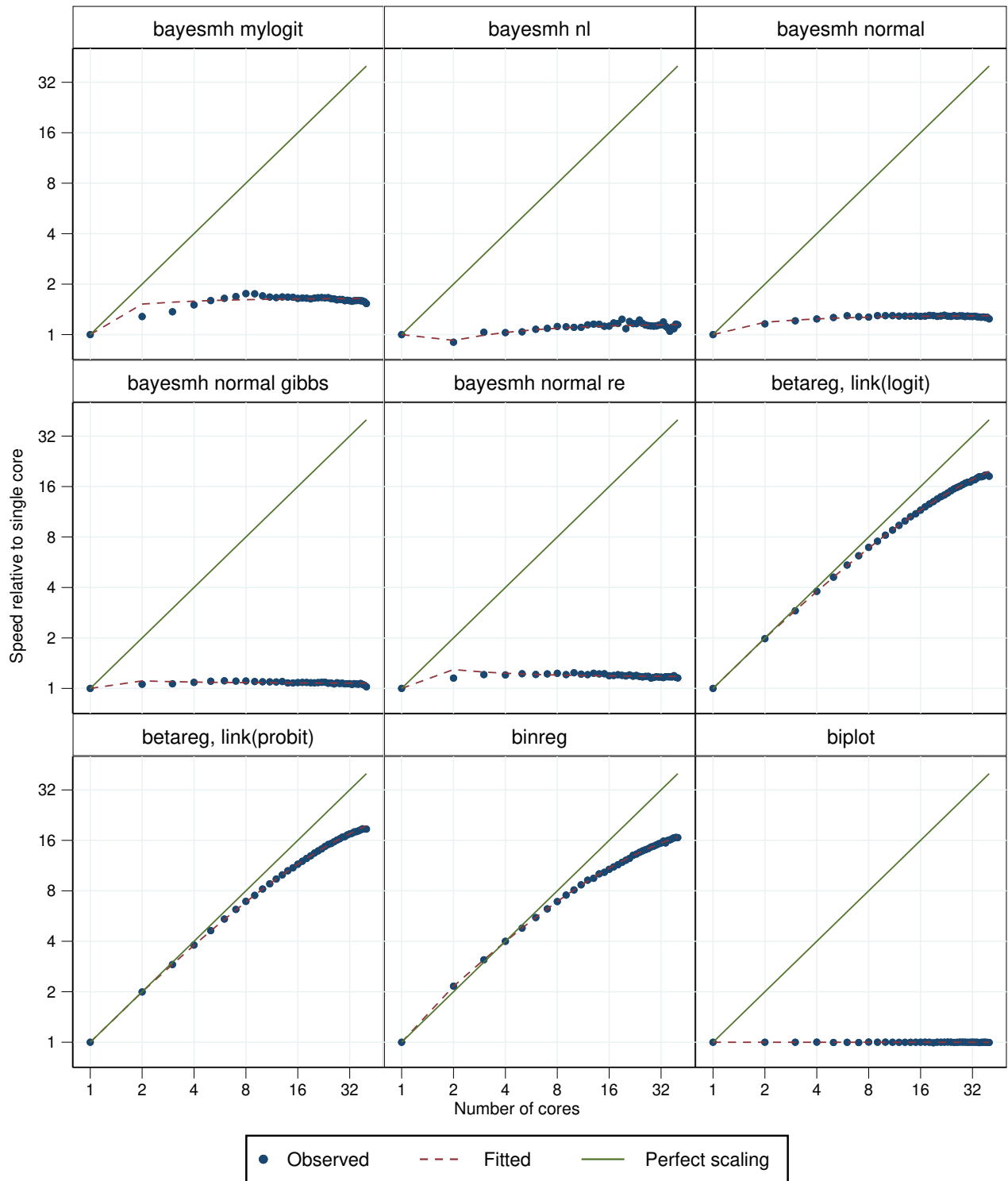


Figure 677. Parallelization performance plots.

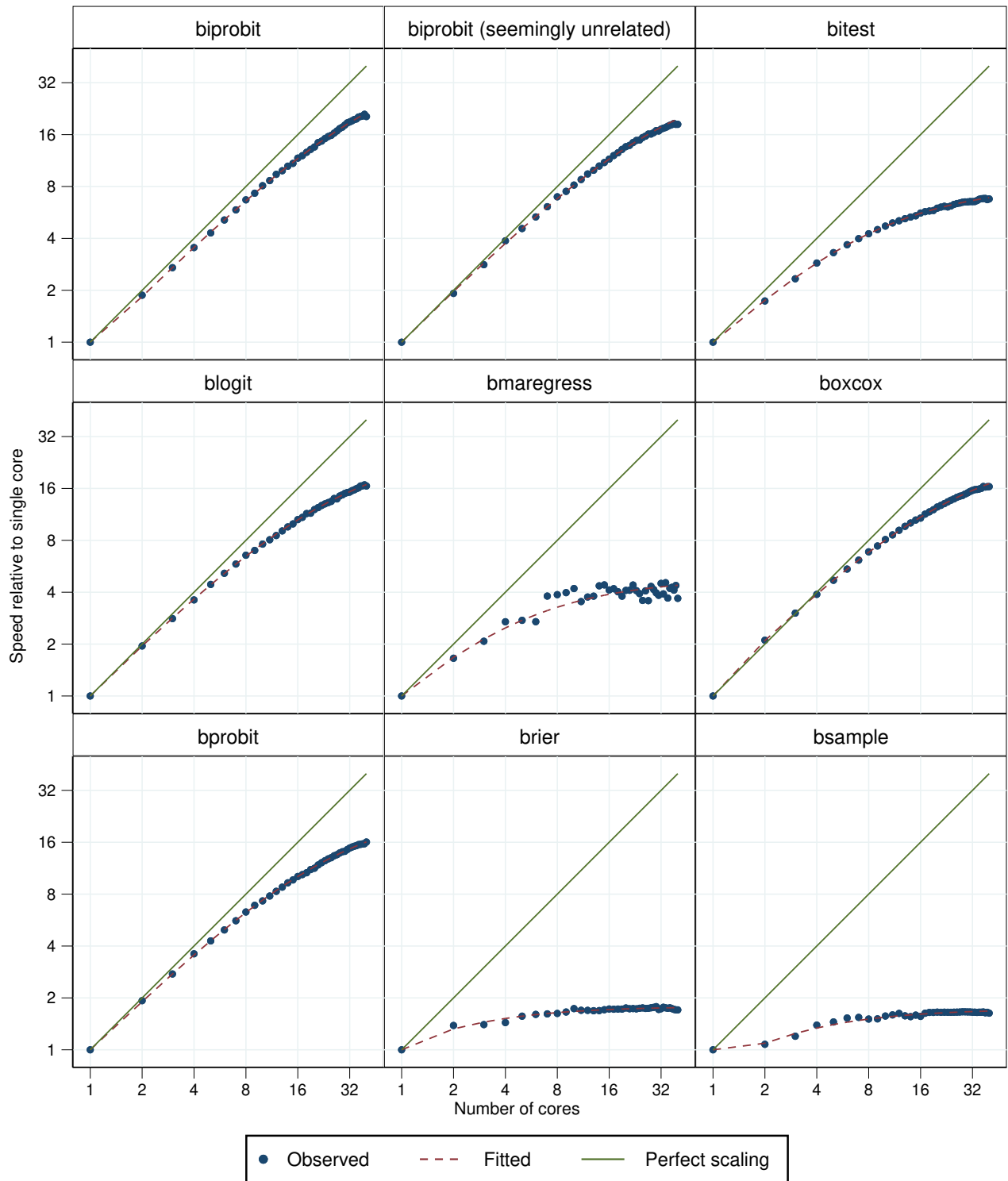


Figure 678. Parallelization performance plots.

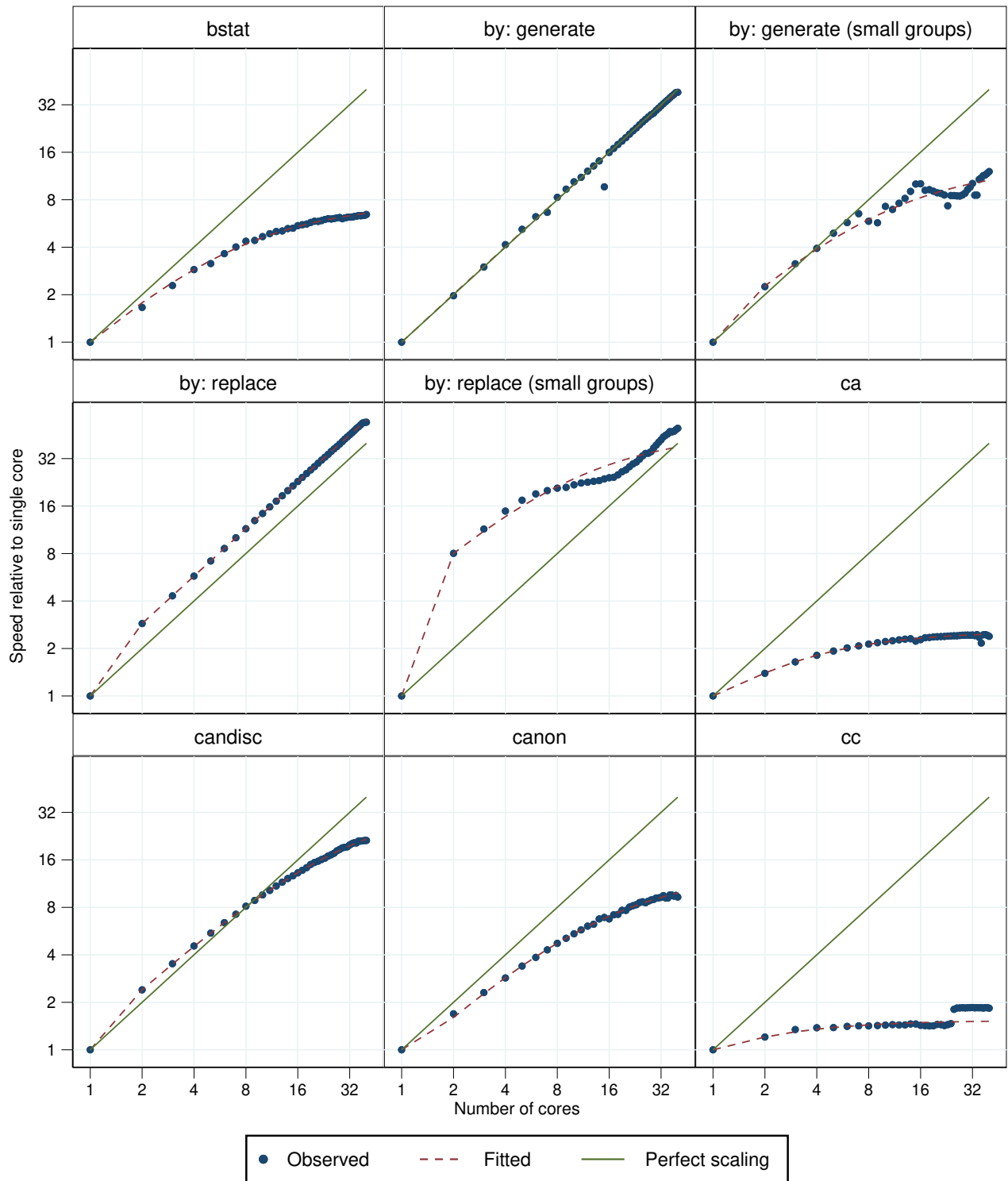


Figure 679. Parallelization performance plots.

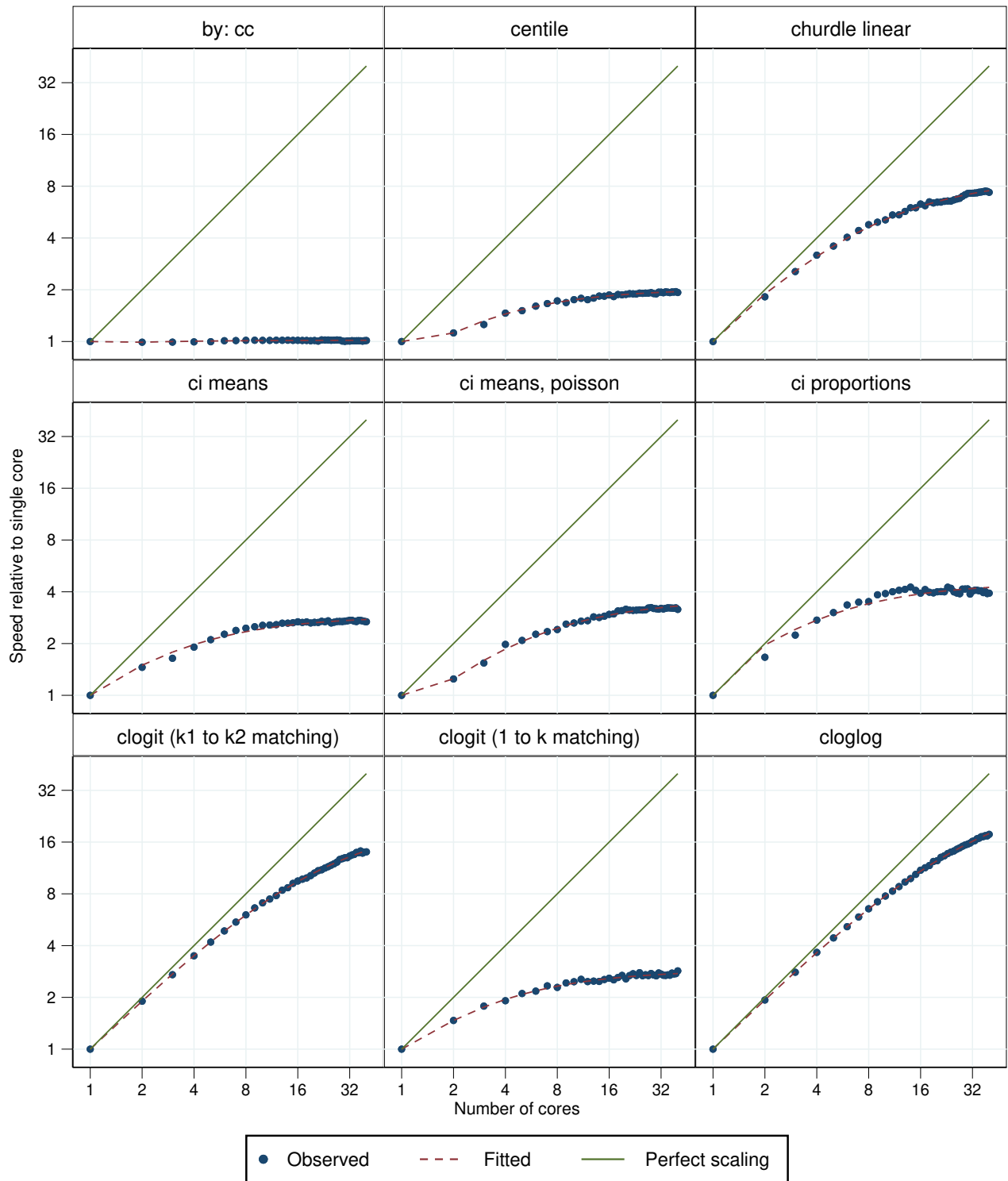


Figure 680. Parallelization performance plots.

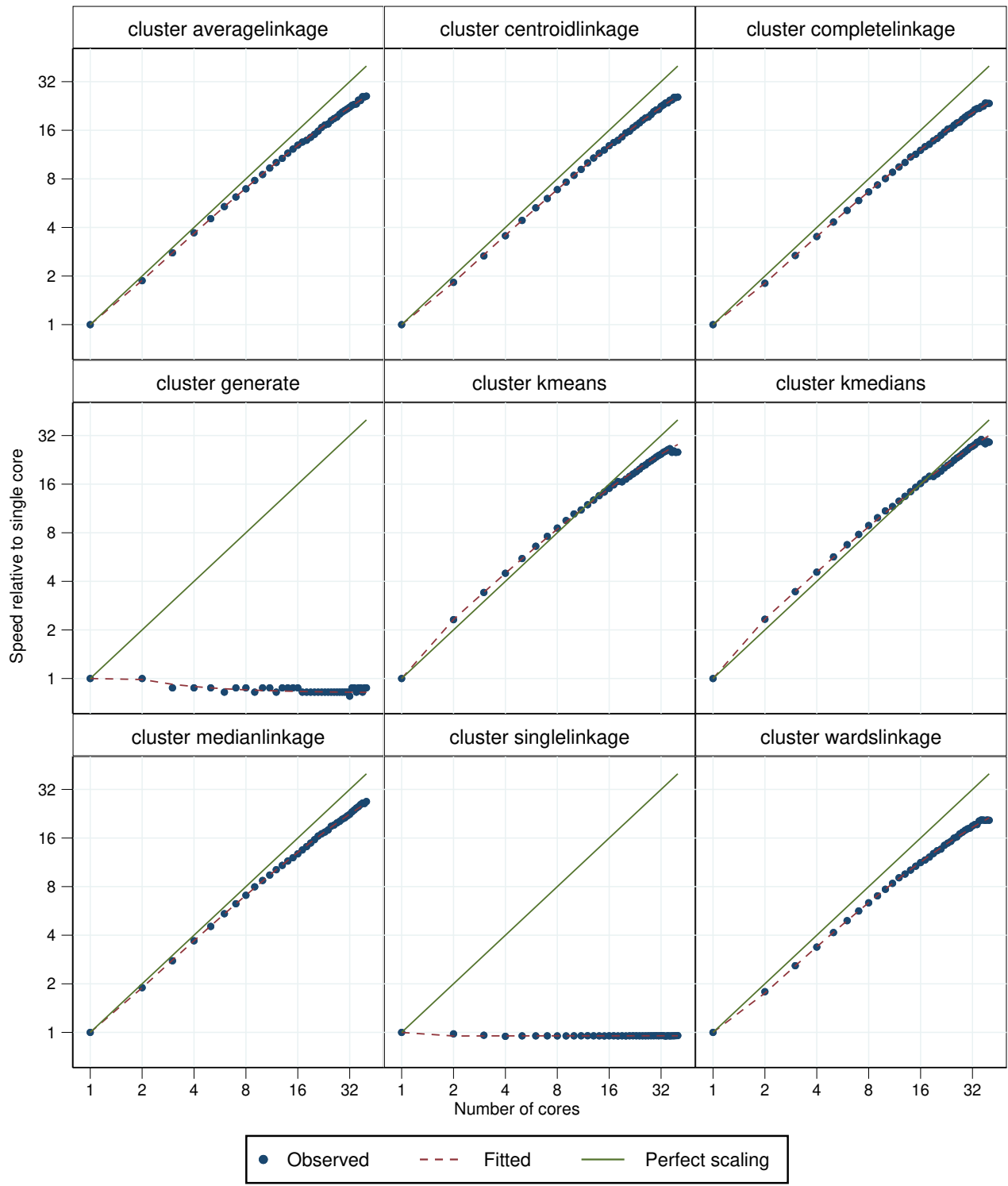


Figure 681. Parallelization performance plots.

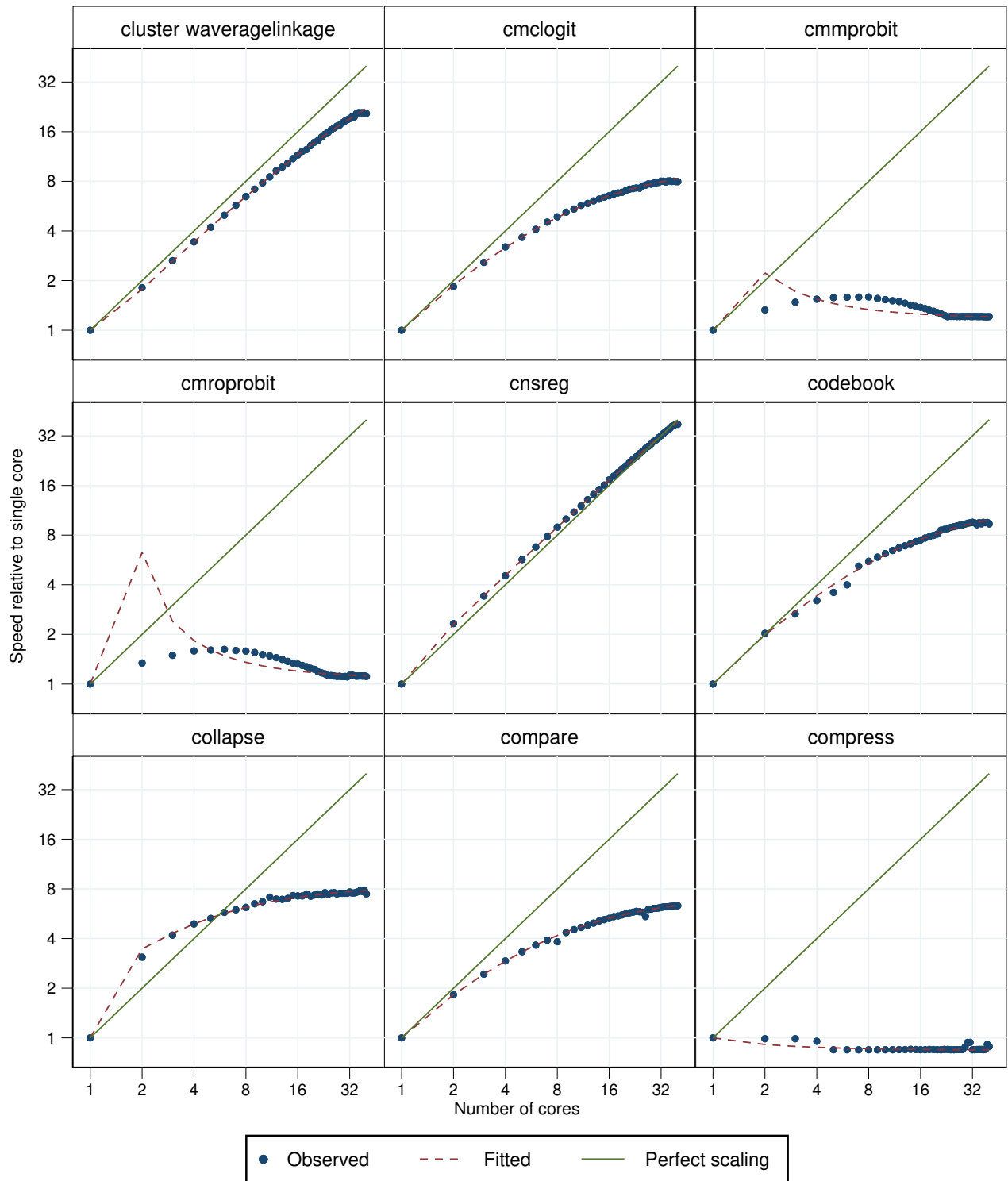


Figure 682. Parallelization performance plots.

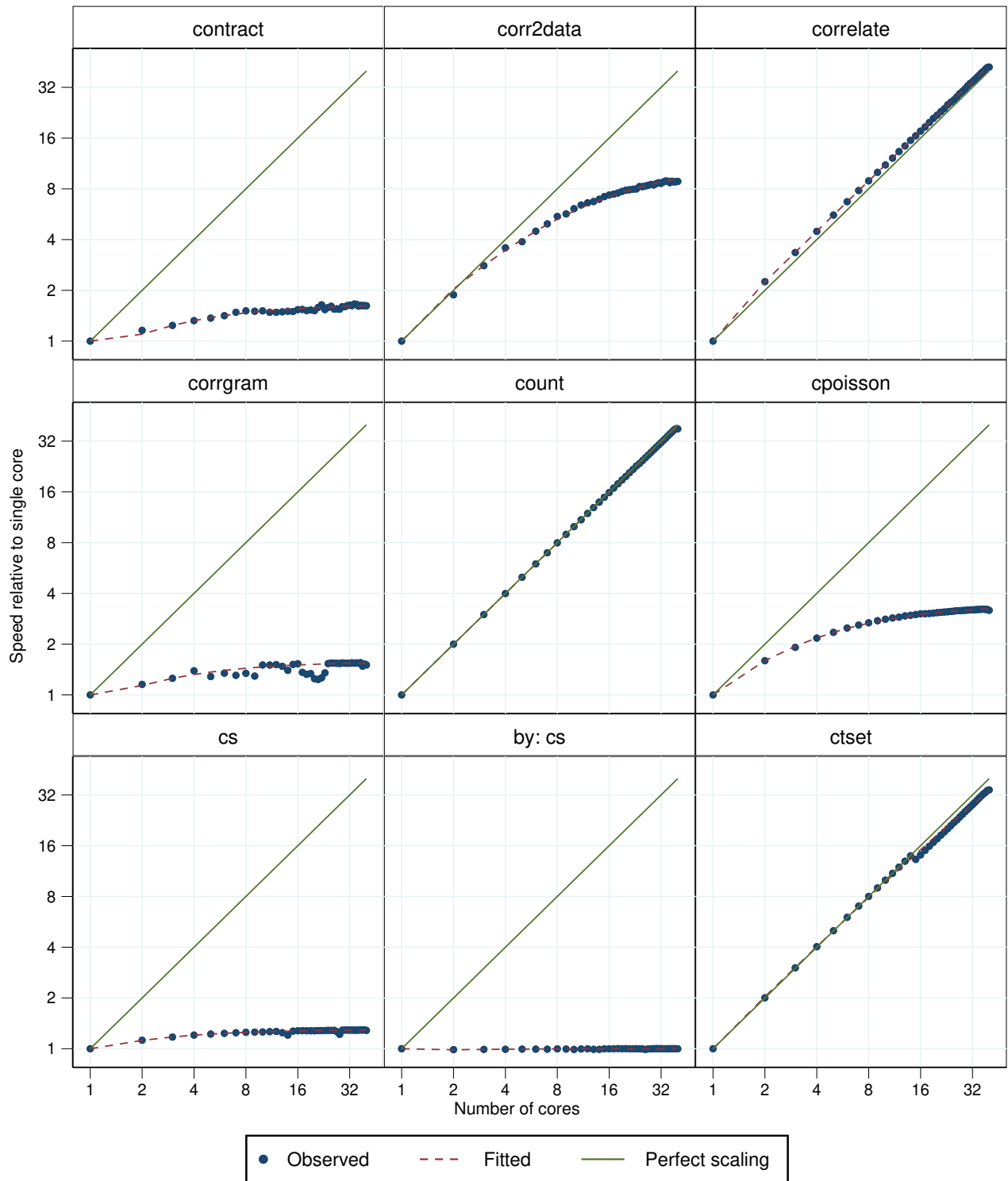


Figure 683. Parallelization performance plots.

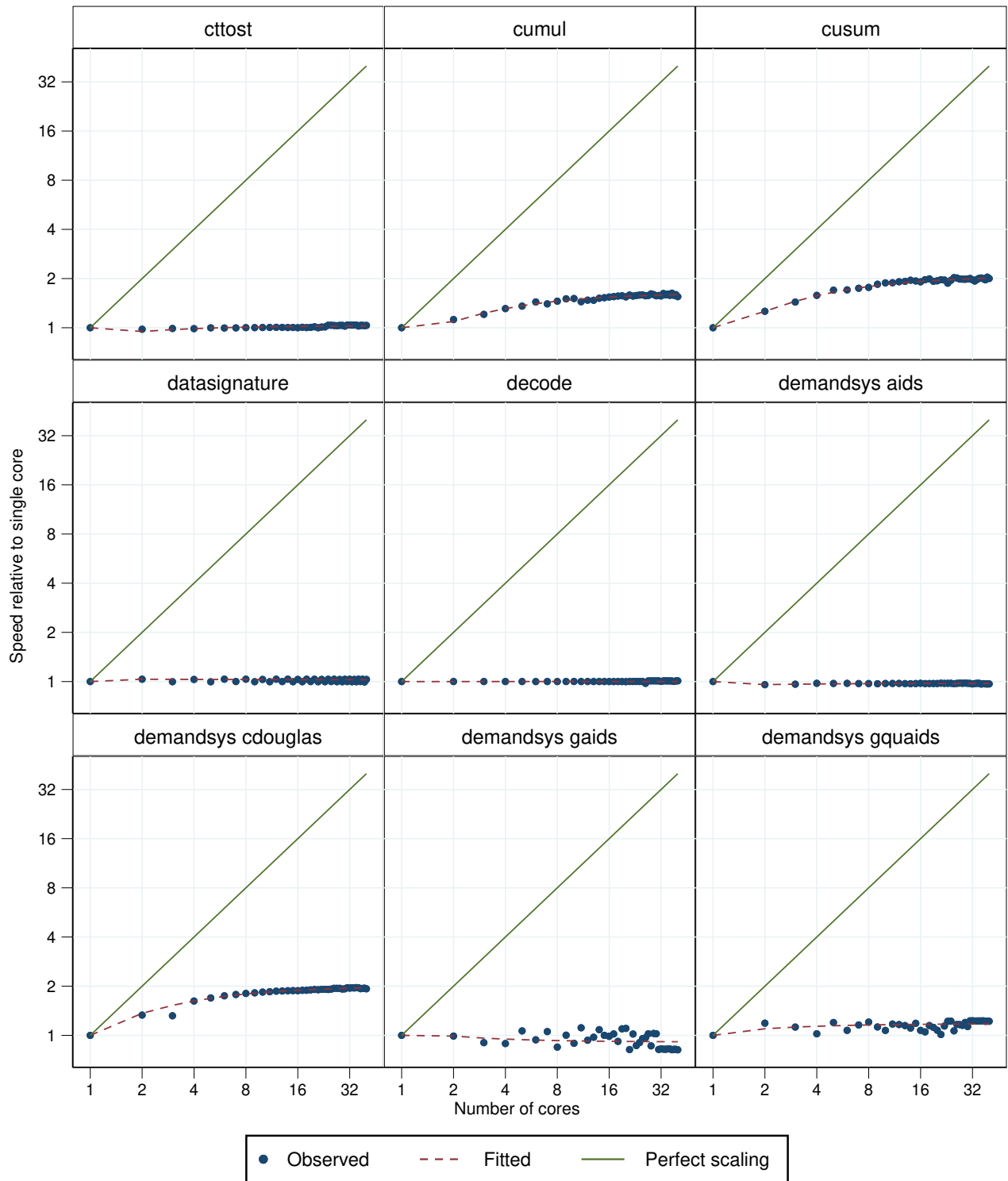


Figure 684. Parallelization performance plots.

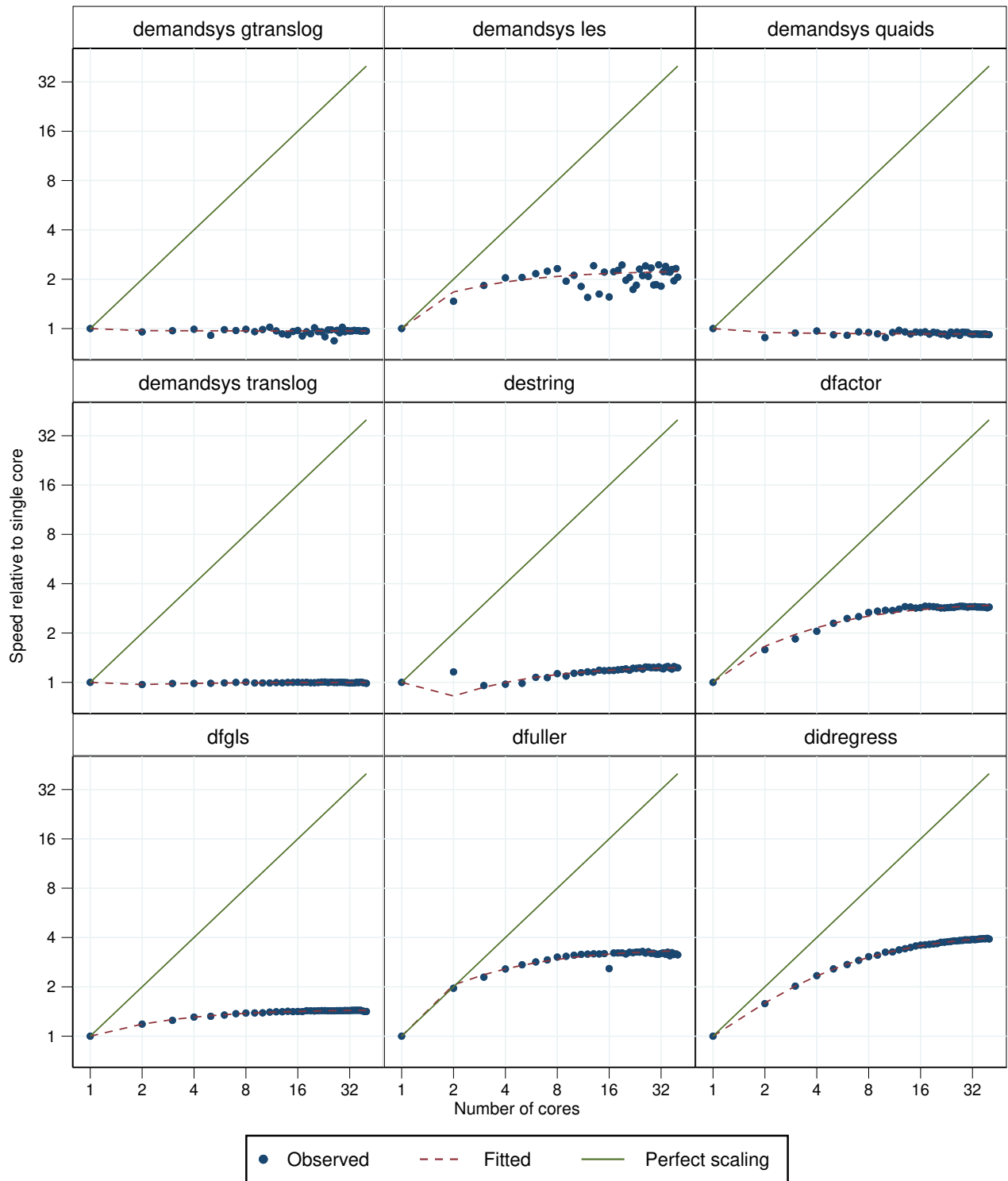


Figure 685. Parallelization performance plots.

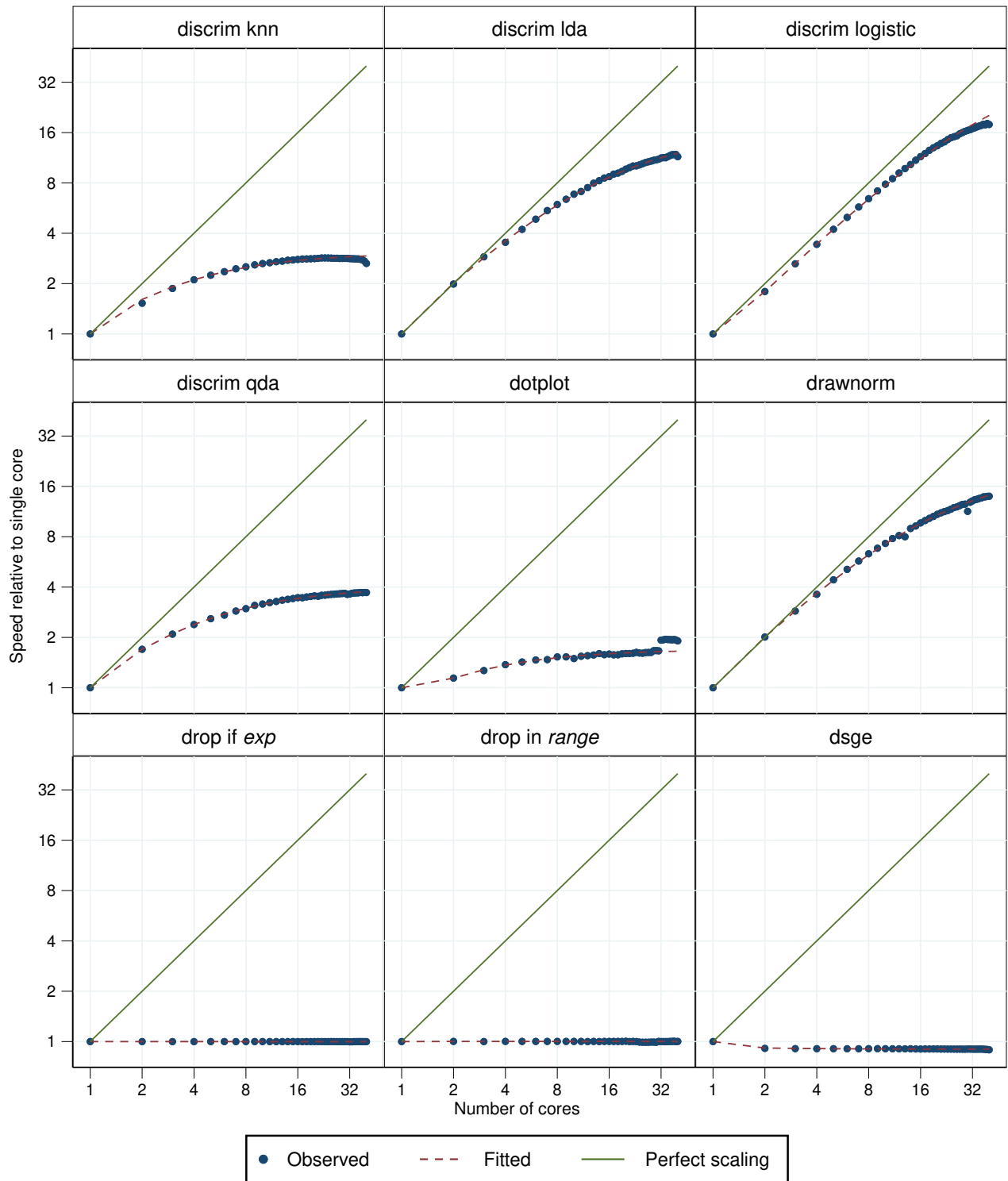


Figure 686. Parallelization performance plots.

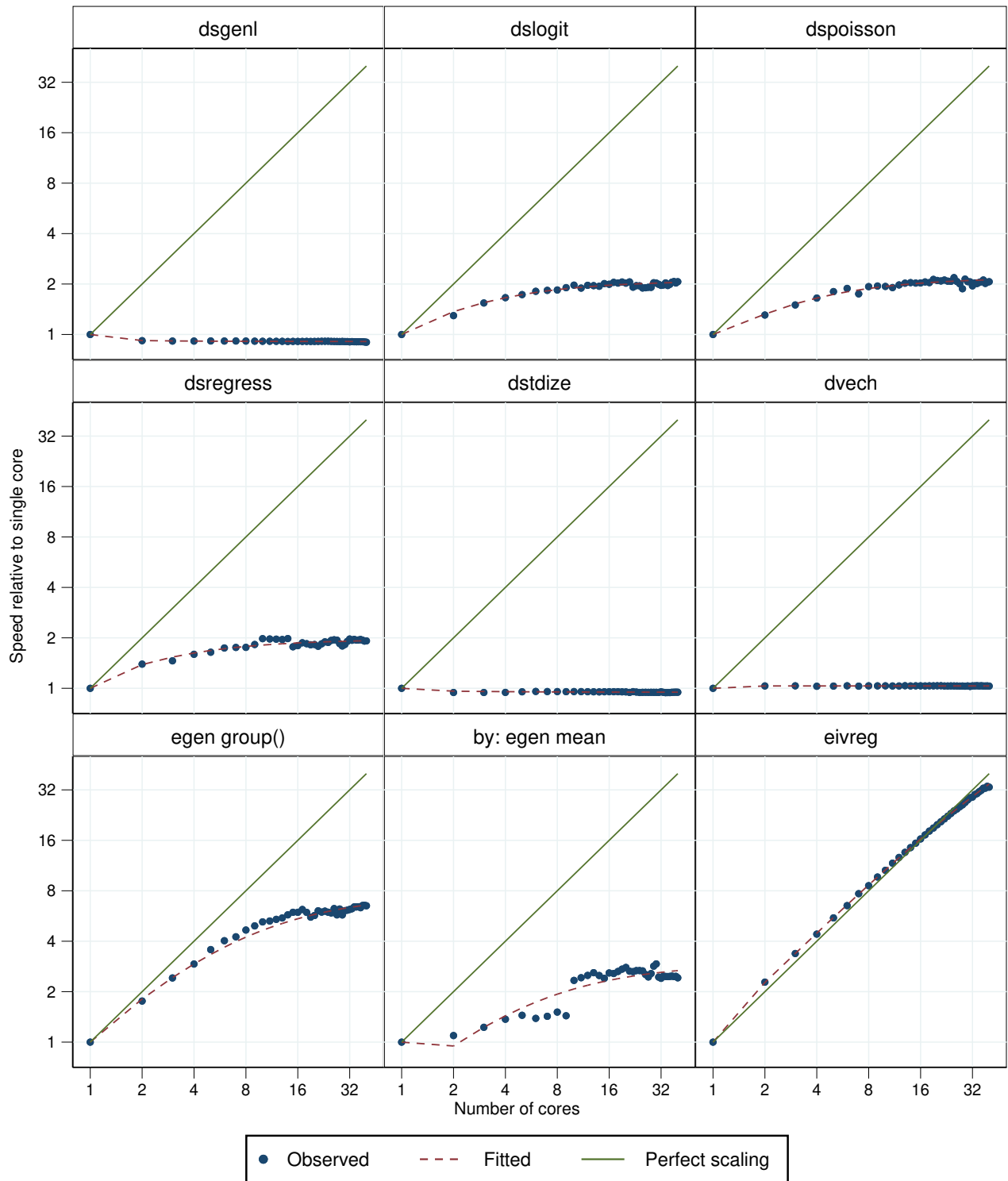


Figure 687. Parallelization performance plots.

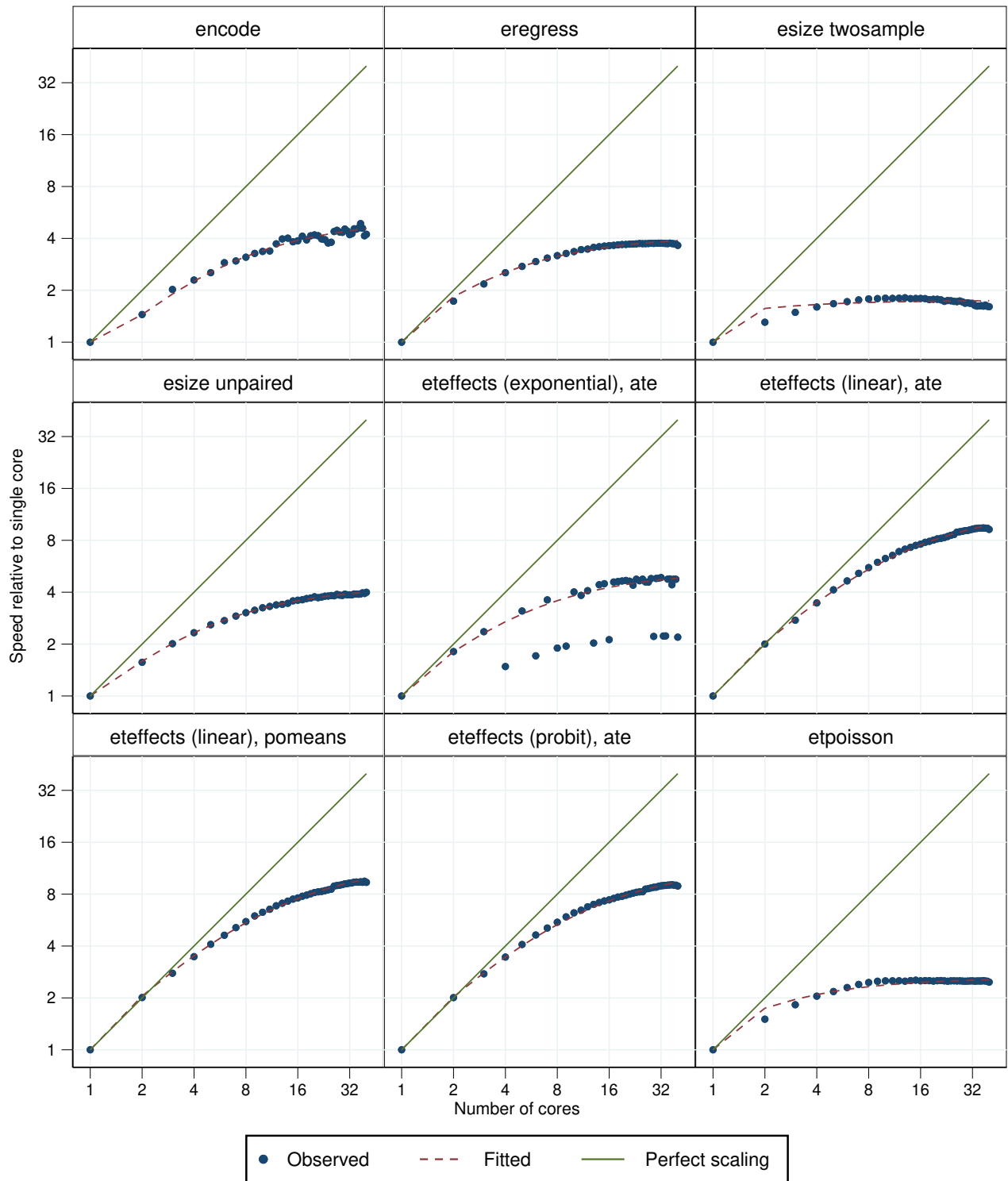


Figure 688. Parallelization performance plots.

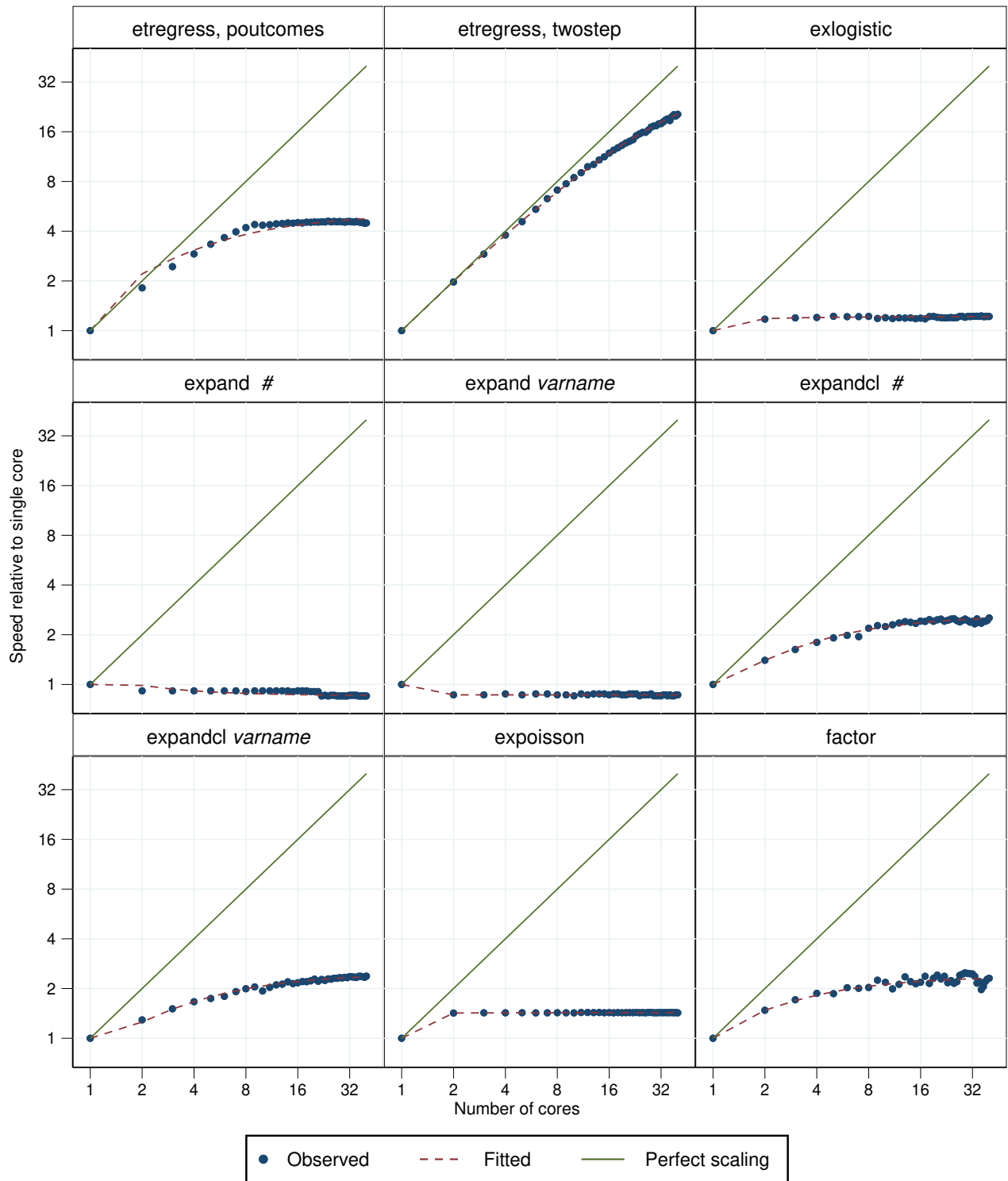


Figure 689. Parallelization performance plots.

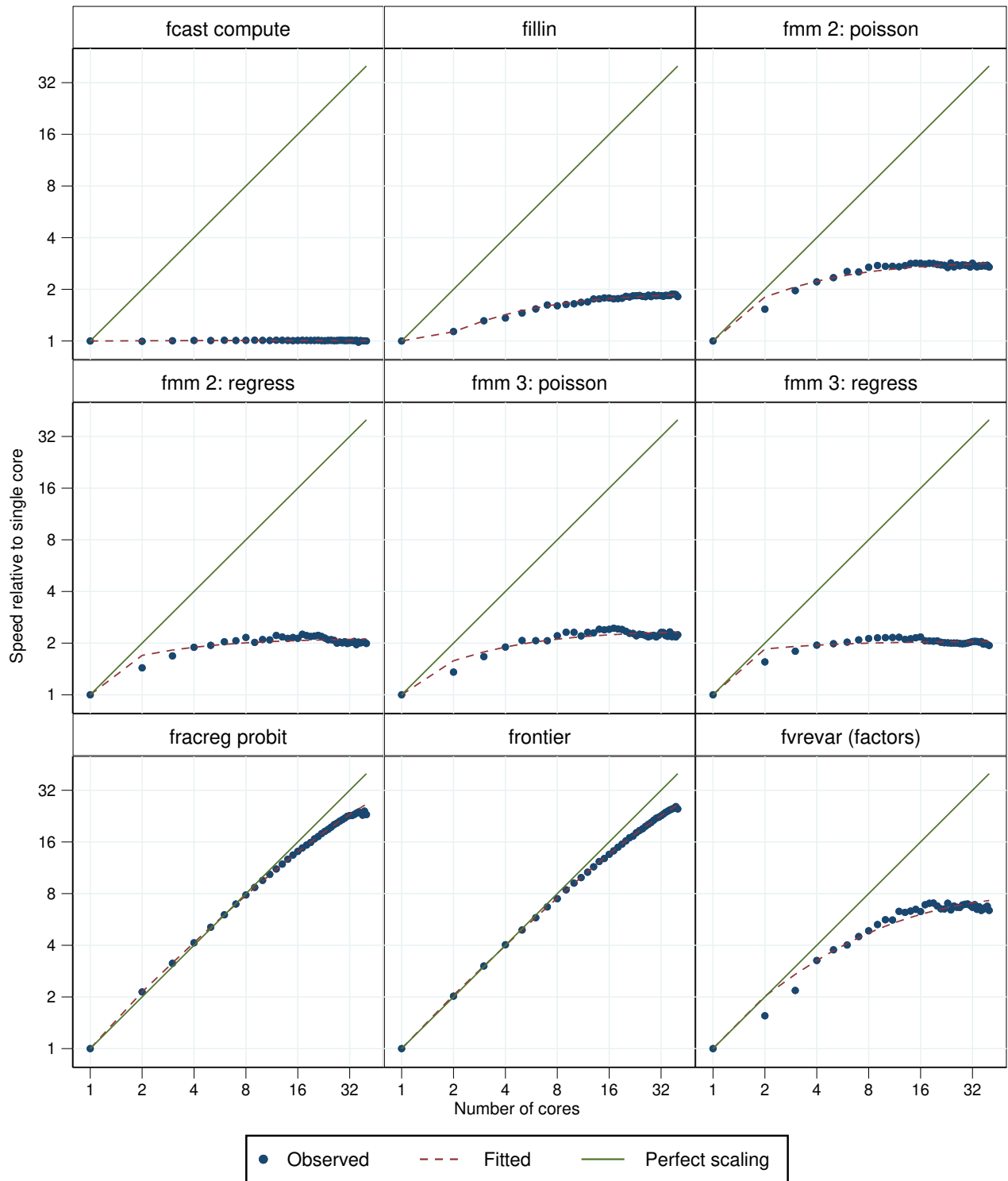


Figure 690. Parallelization performance plots.

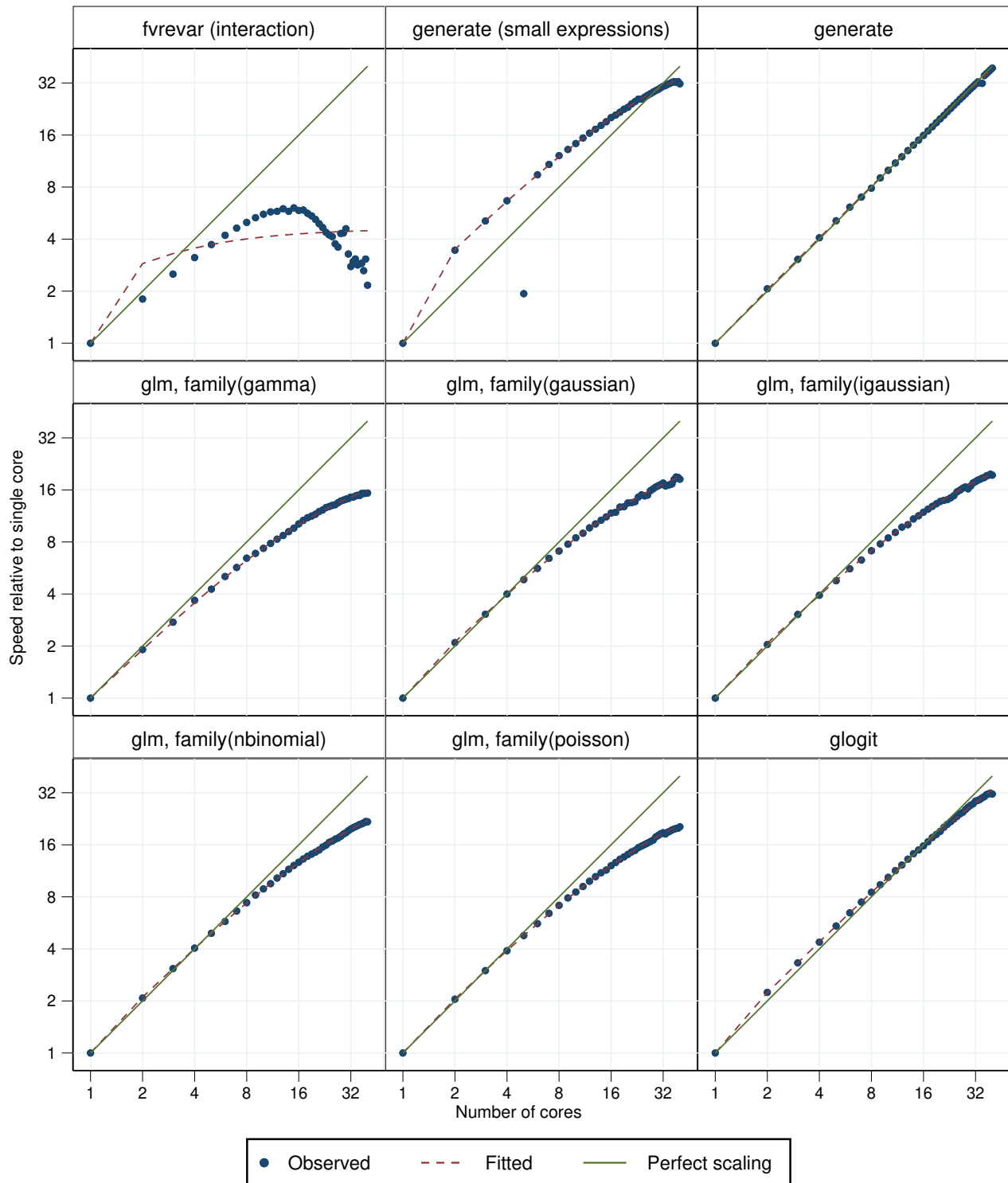


Figure 691. Parallelization performance plots.

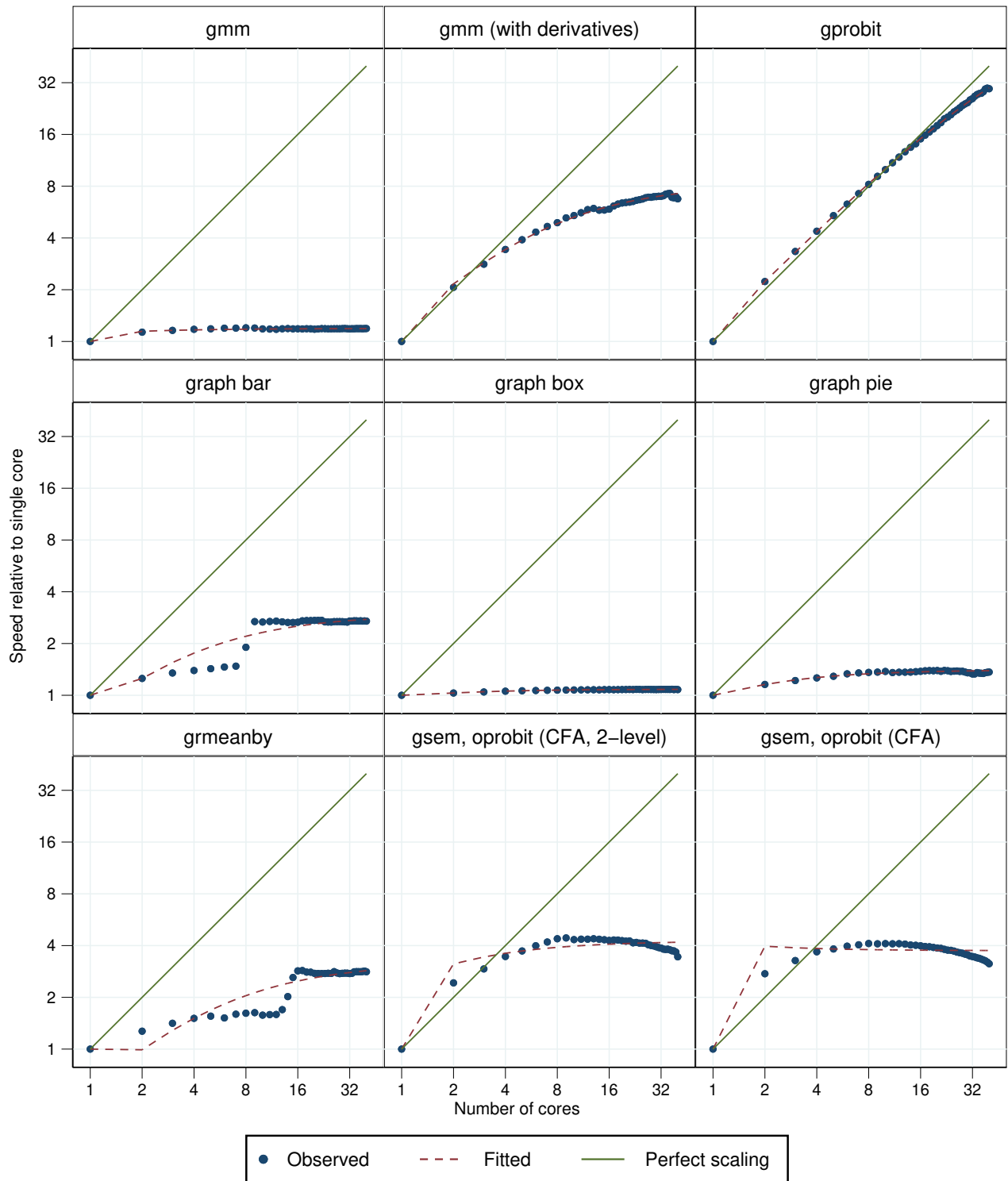


Figure 692. Parallelization performance plots.

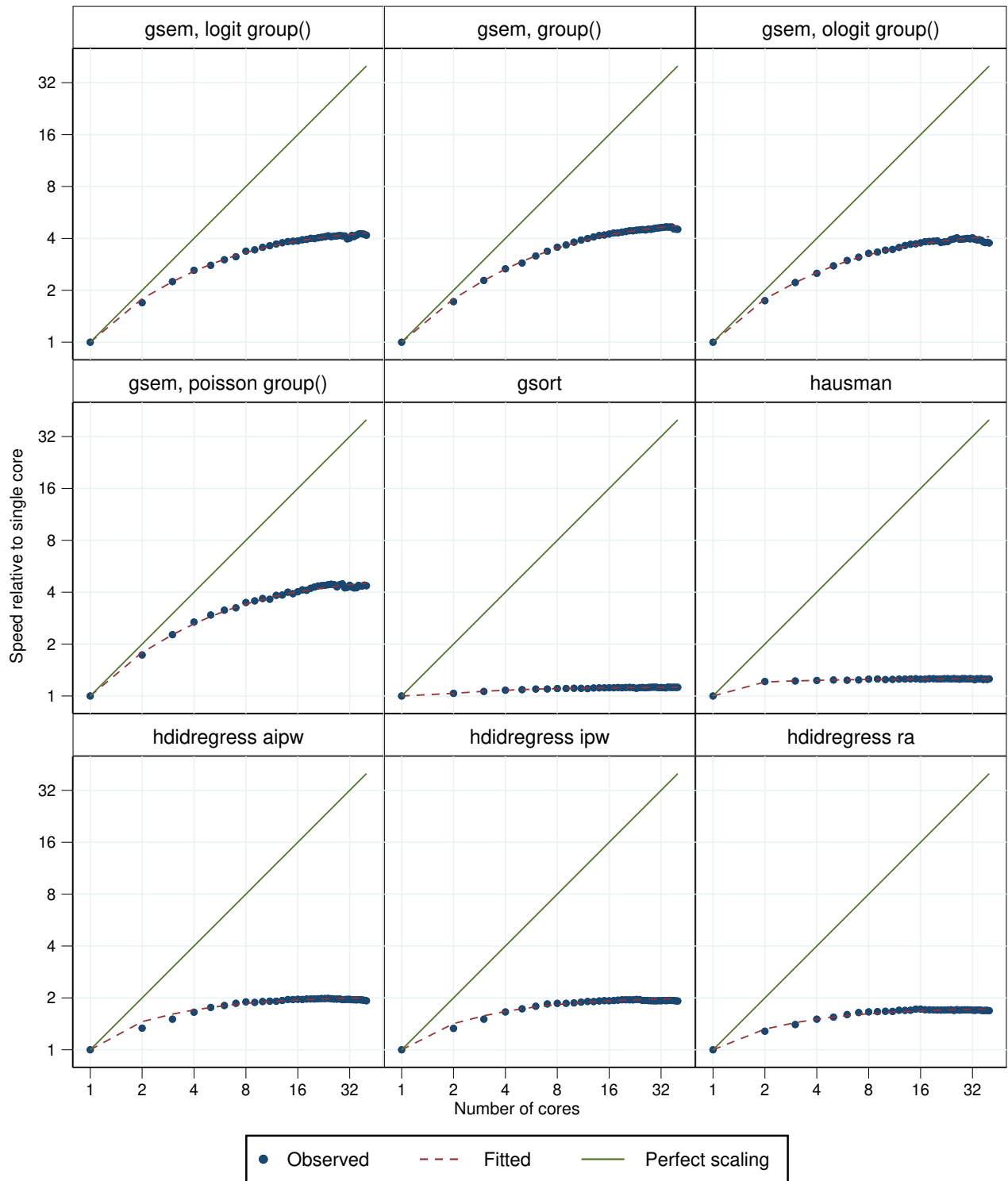


Figure 693. Parallelization performance plots.

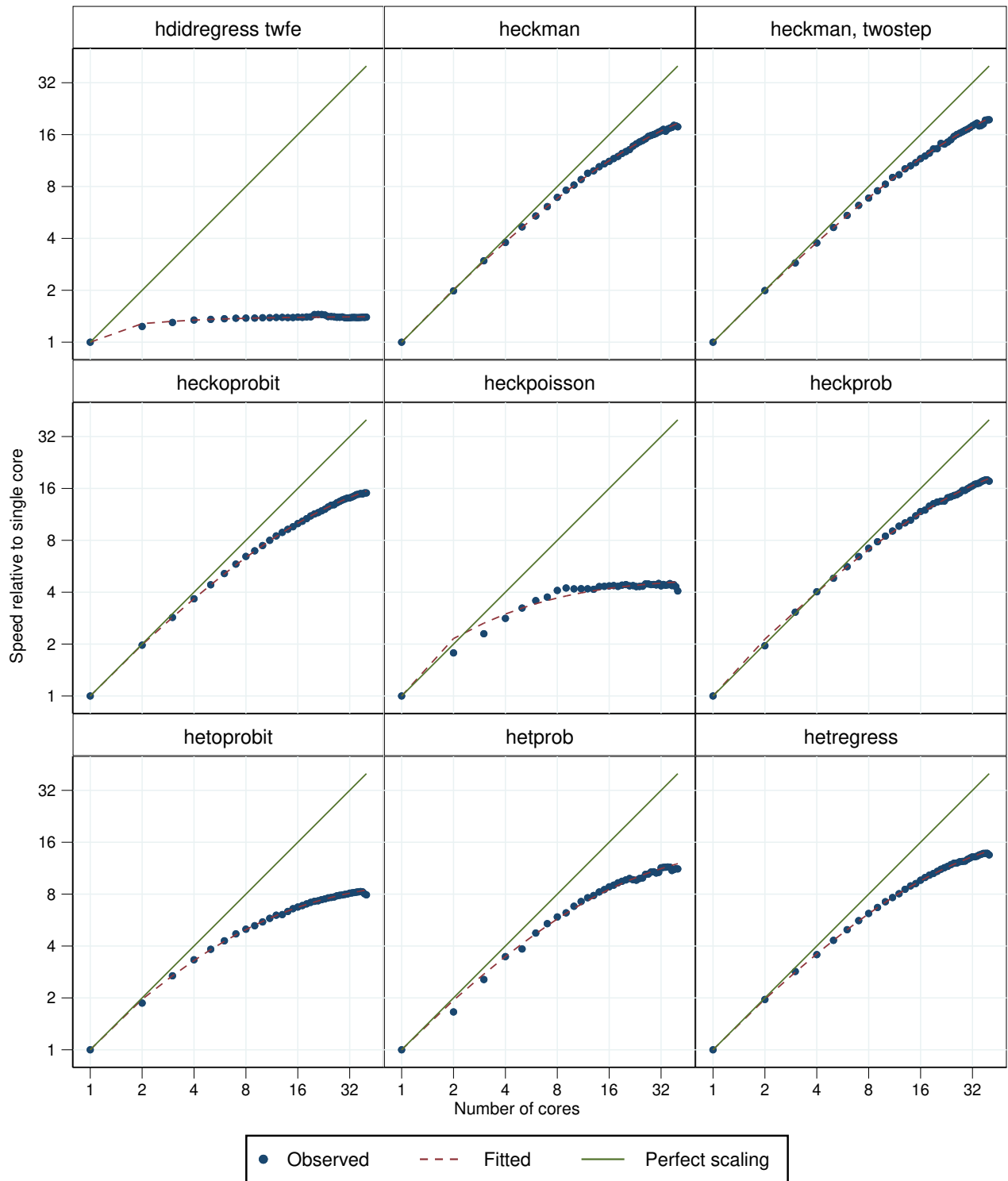


Figure 694. Parallelization performance plots.

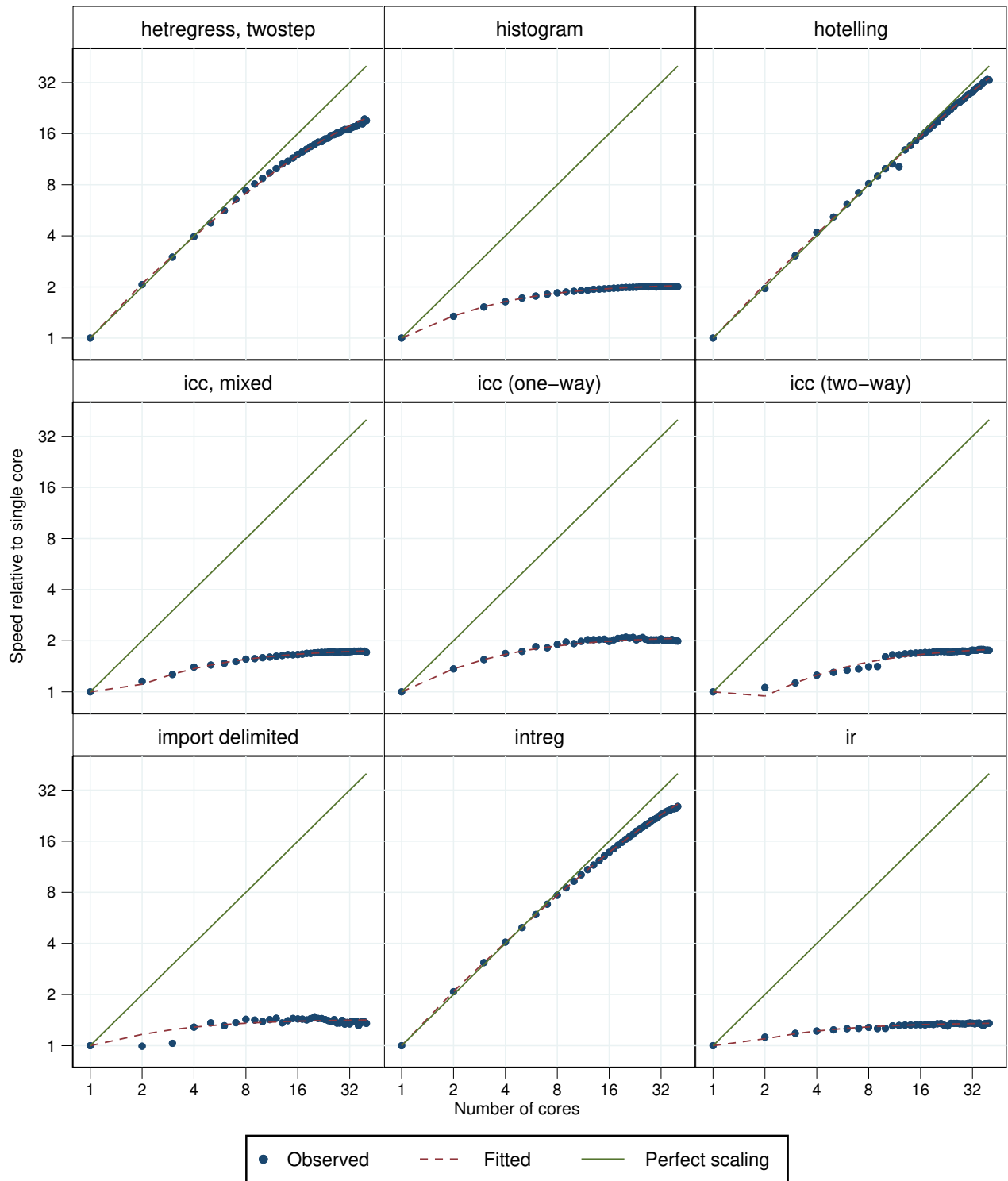


Figure 695. Parallelization performance plots.

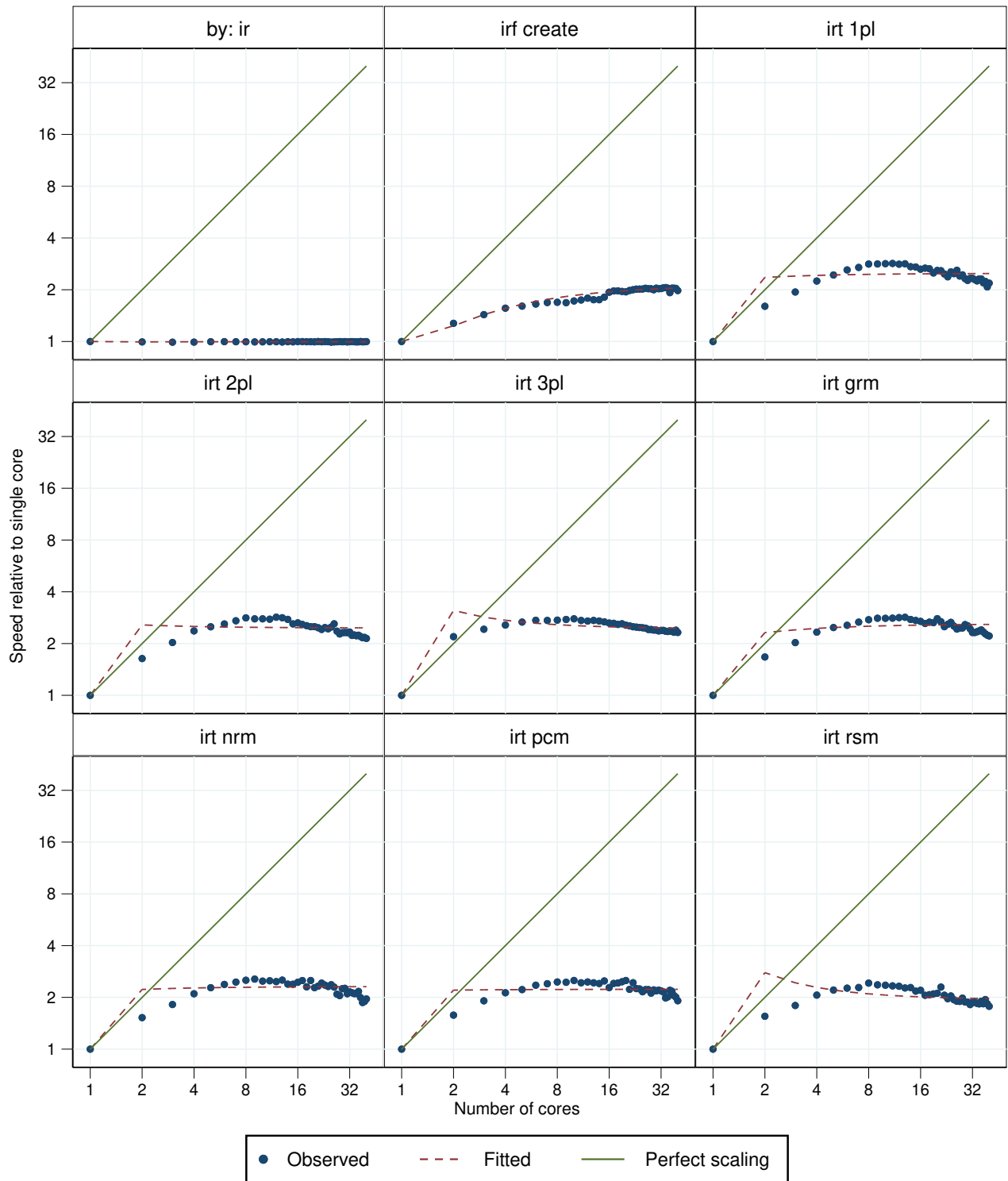


Figure 696. Parallelization performance plots.

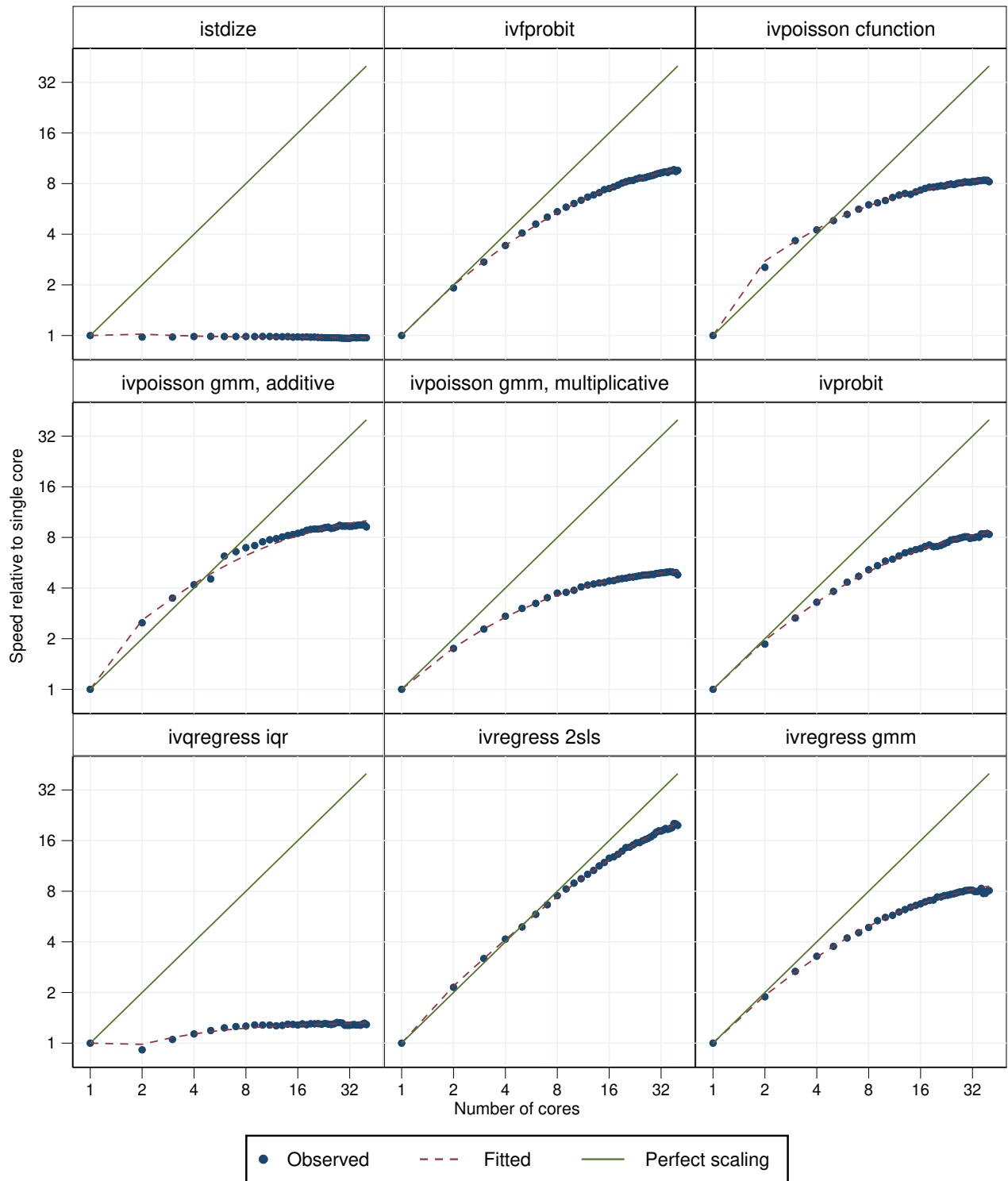


Figure 697. Parallelization performance plots.

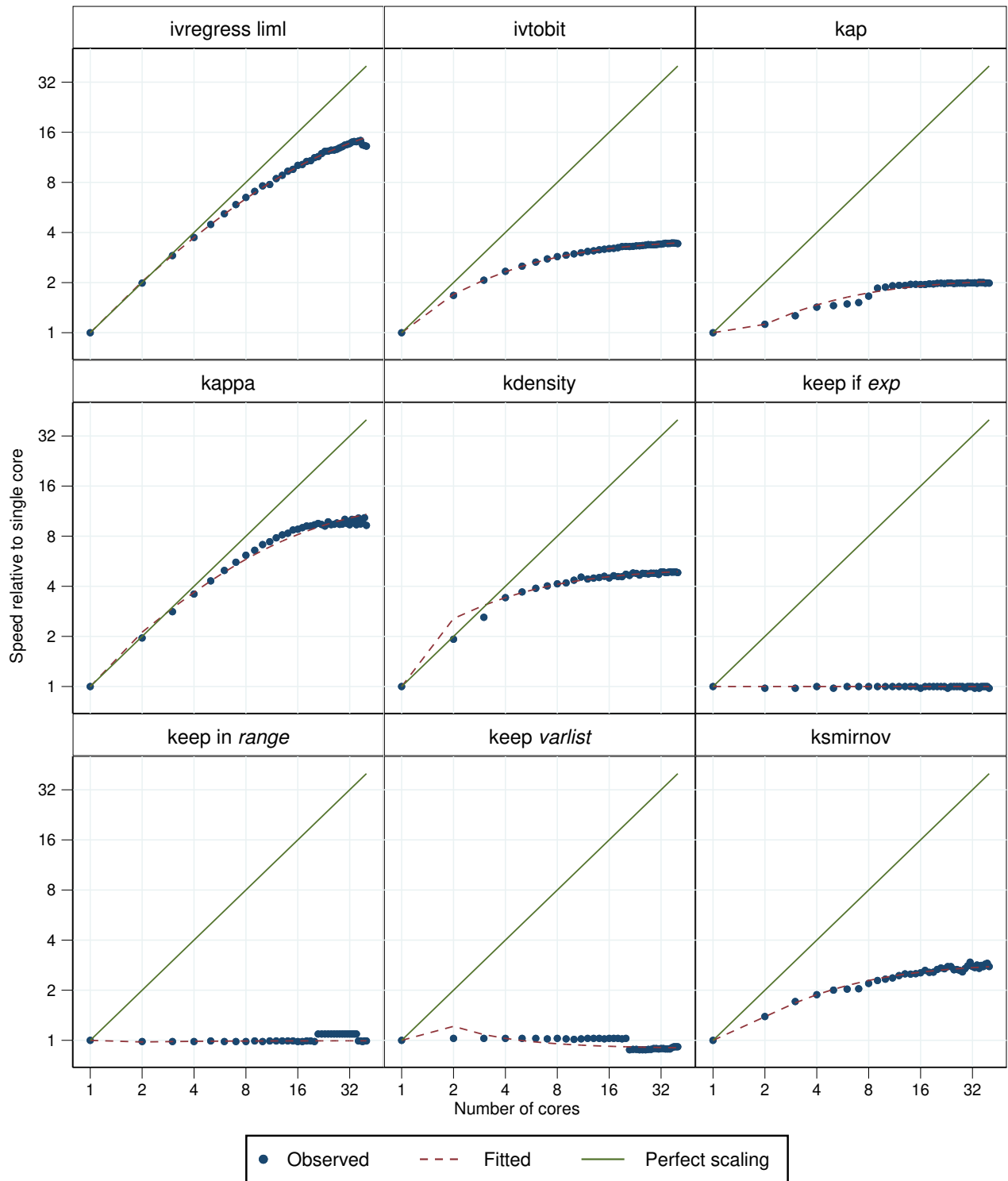


Figure 698. Parallelization performance plots.

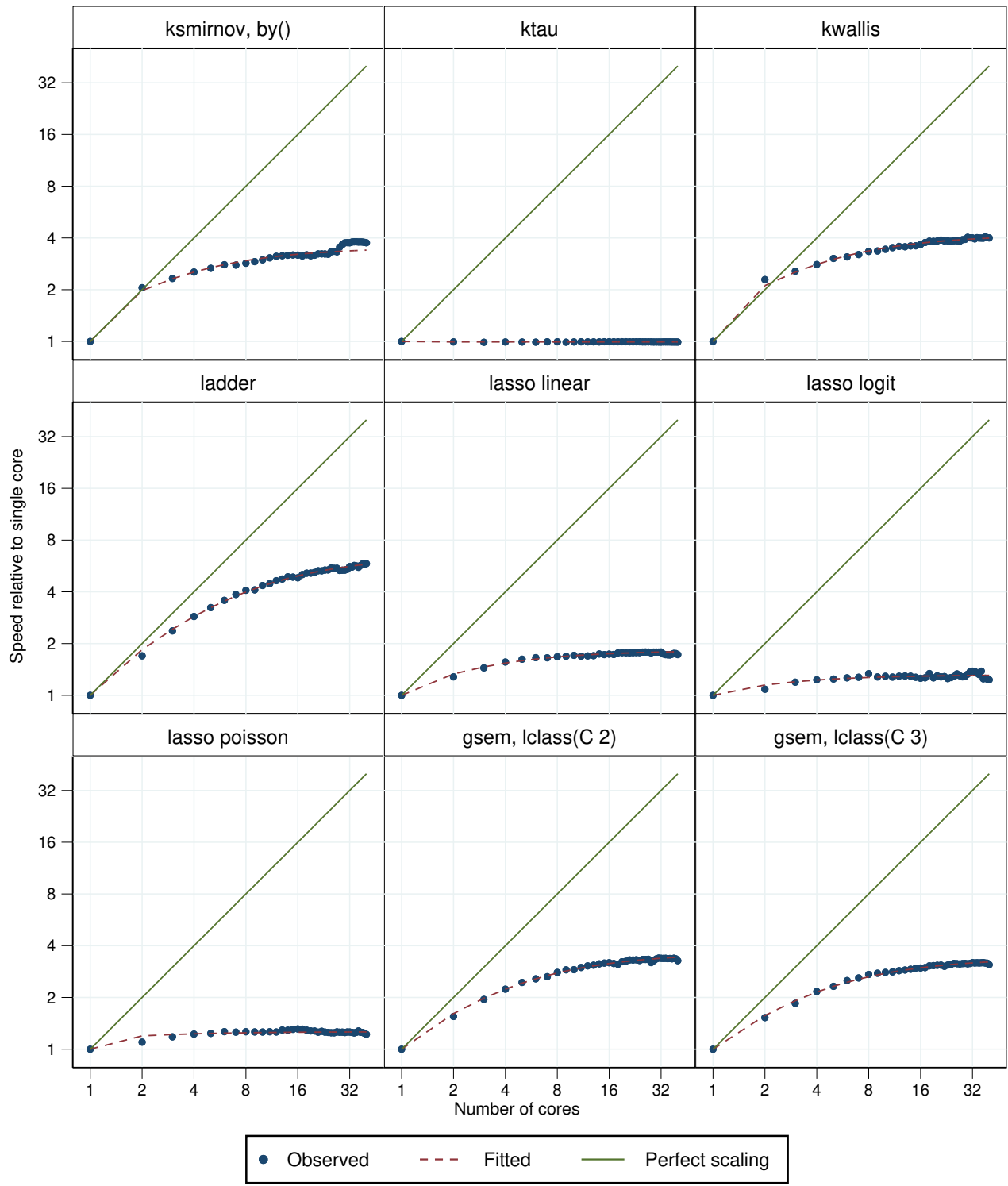


Figure 699. Parallelization performance plots.

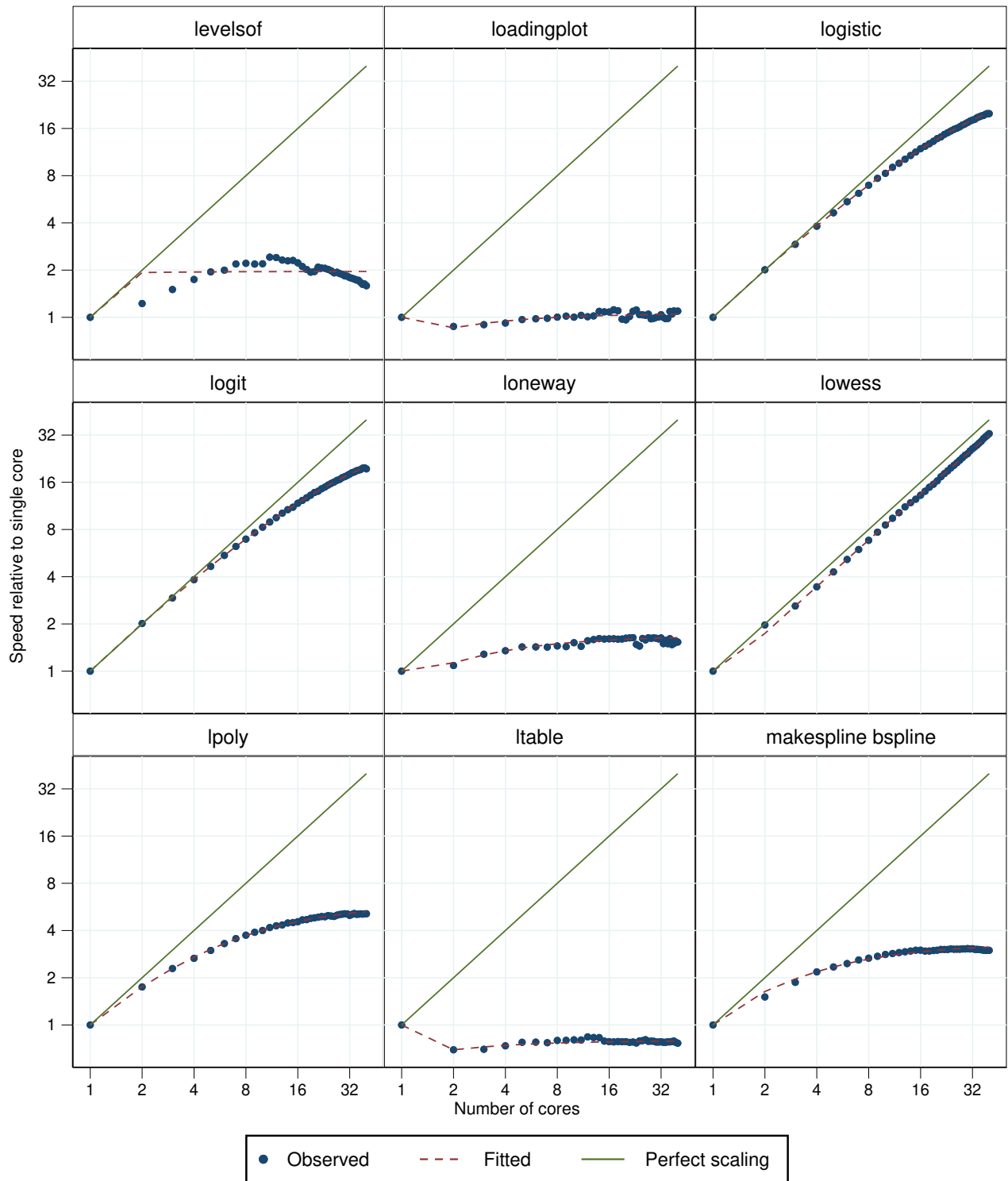


Figure 700. Parallelization performance plots.

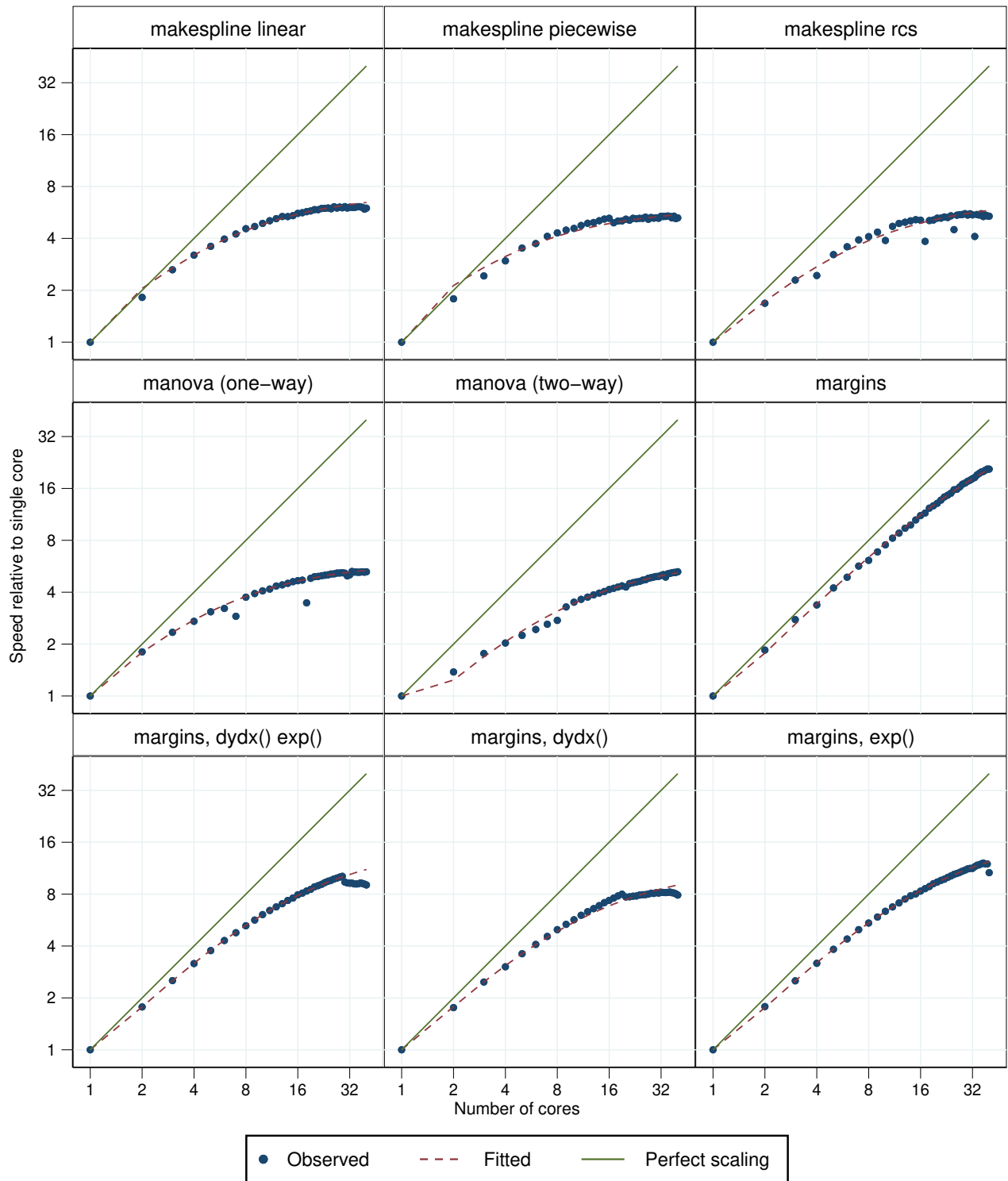


Figure 701. Parallelization performance plots.

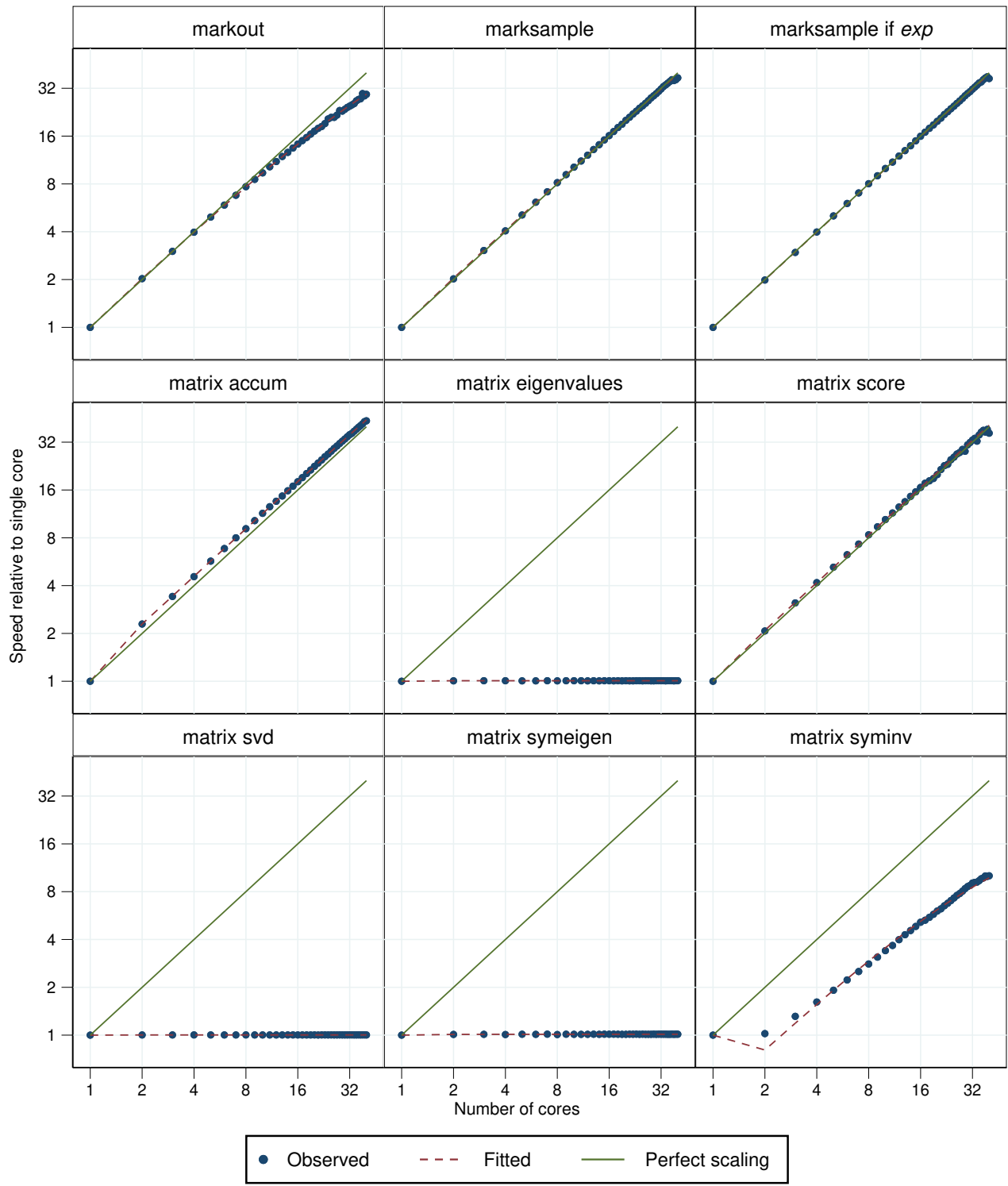


Figure 702. Parallelization performance plots.

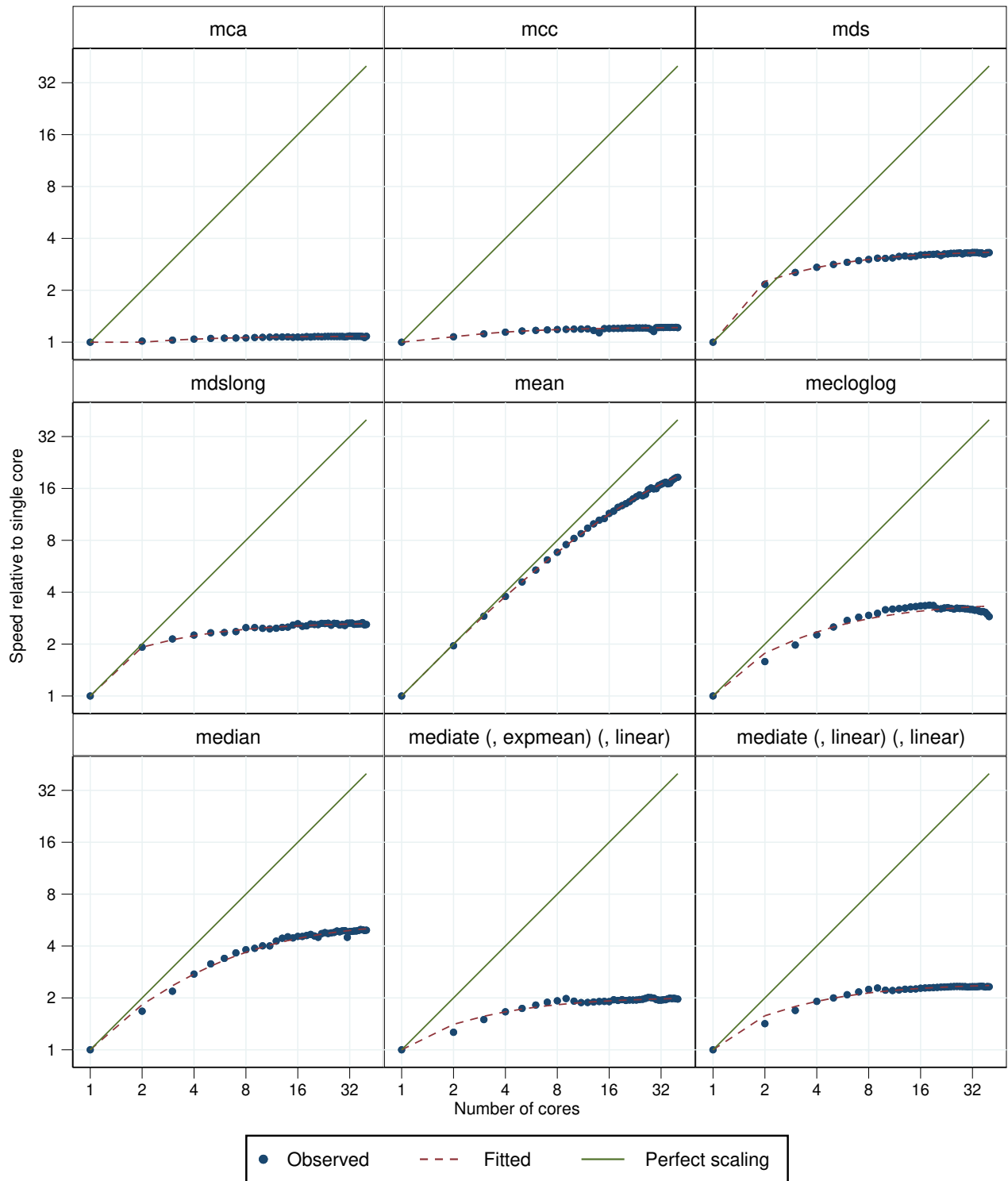


Figure 703. Parallelization performance plots.

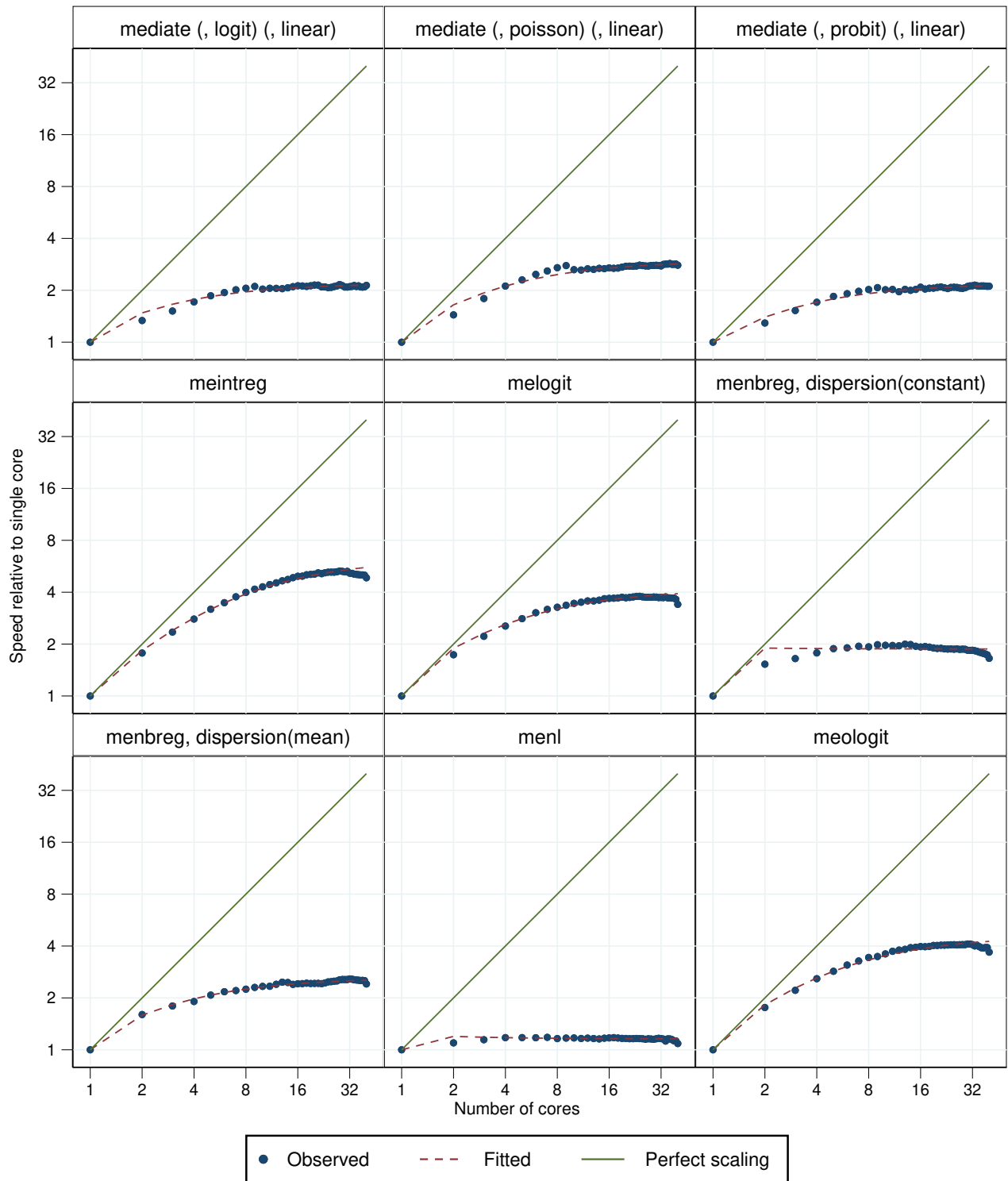


Figure 704. Parallelization performance plots.

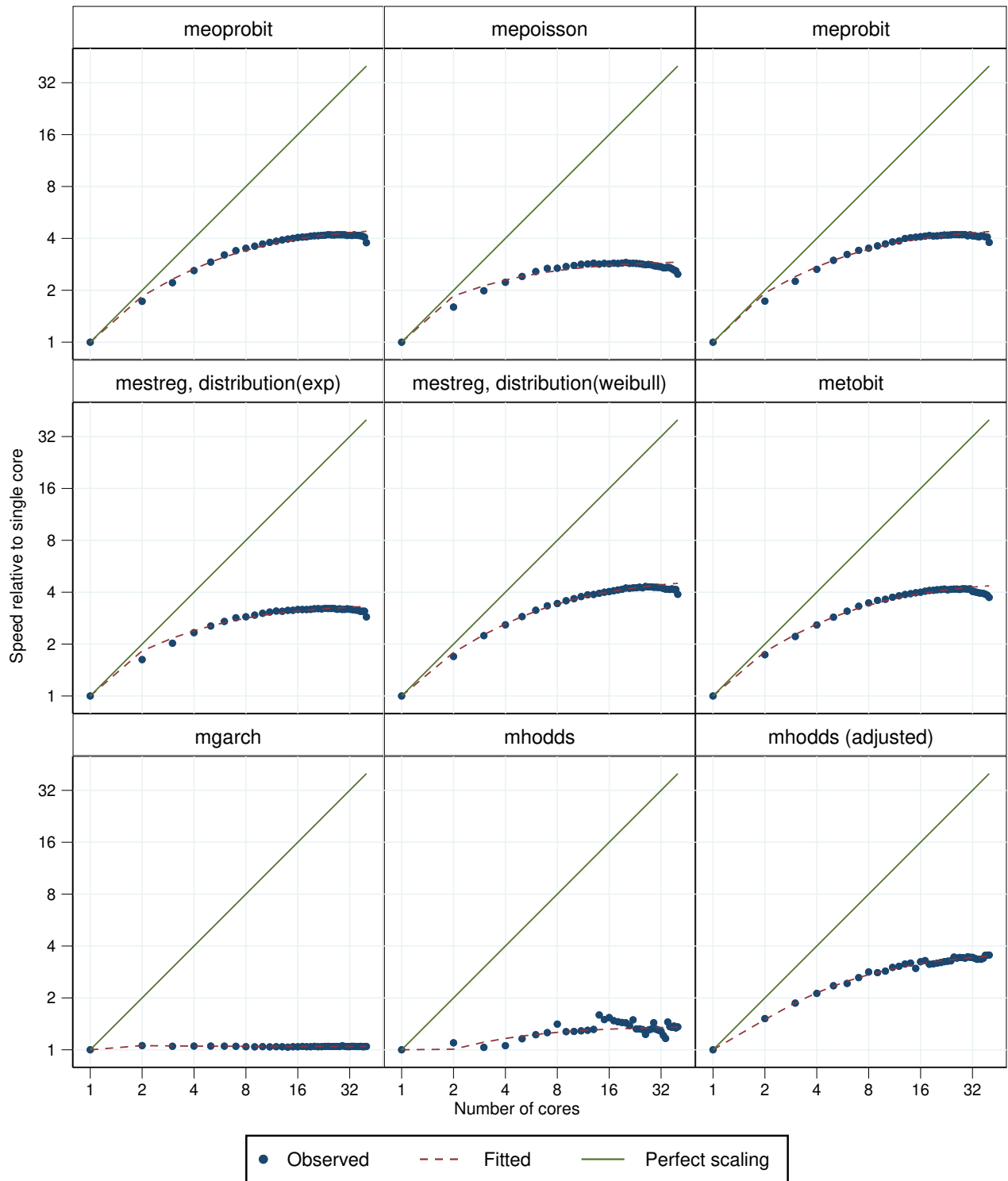


Figure 705. Parallelization performance plots.

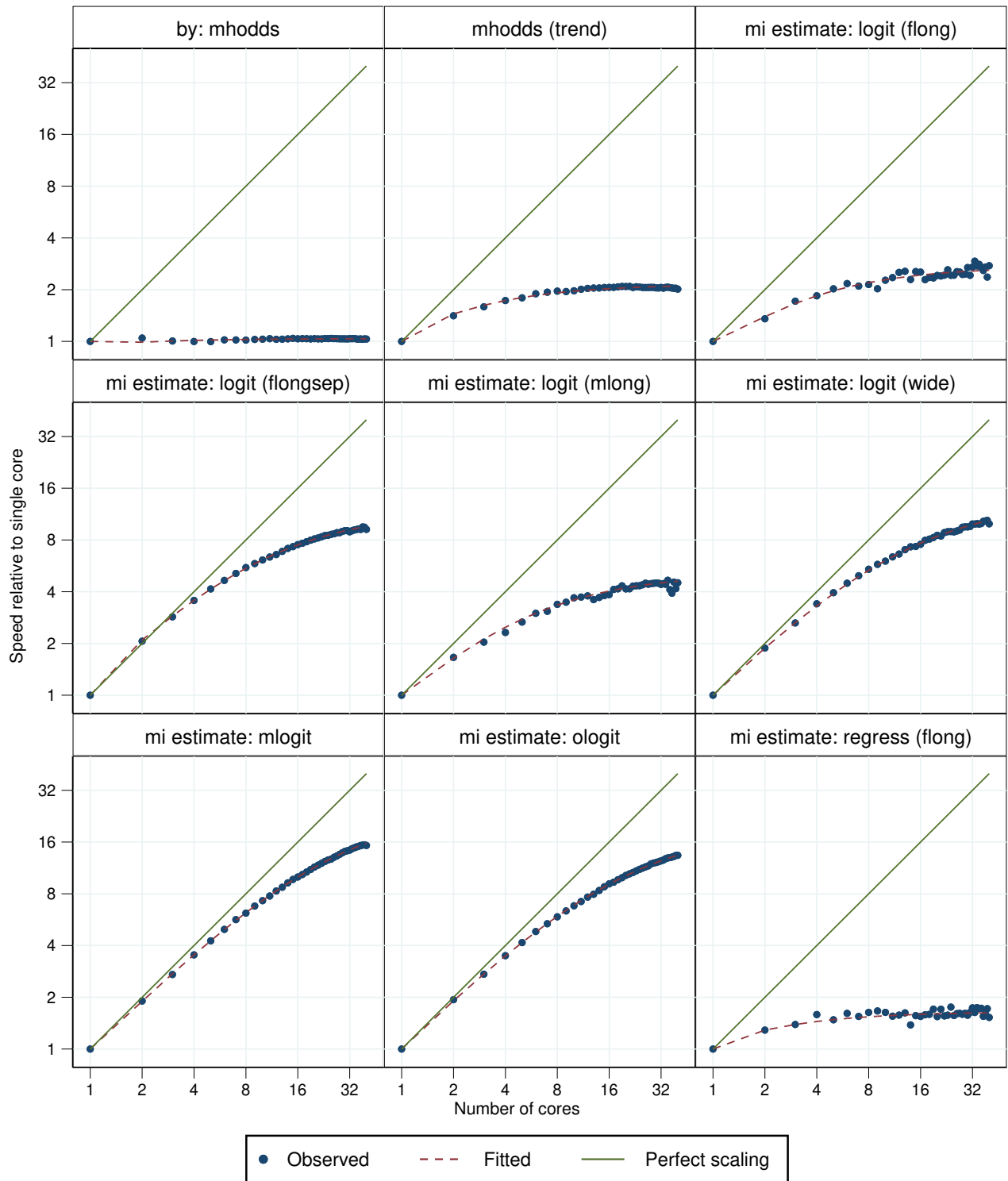


Figure 706. Parallelization performance plots.

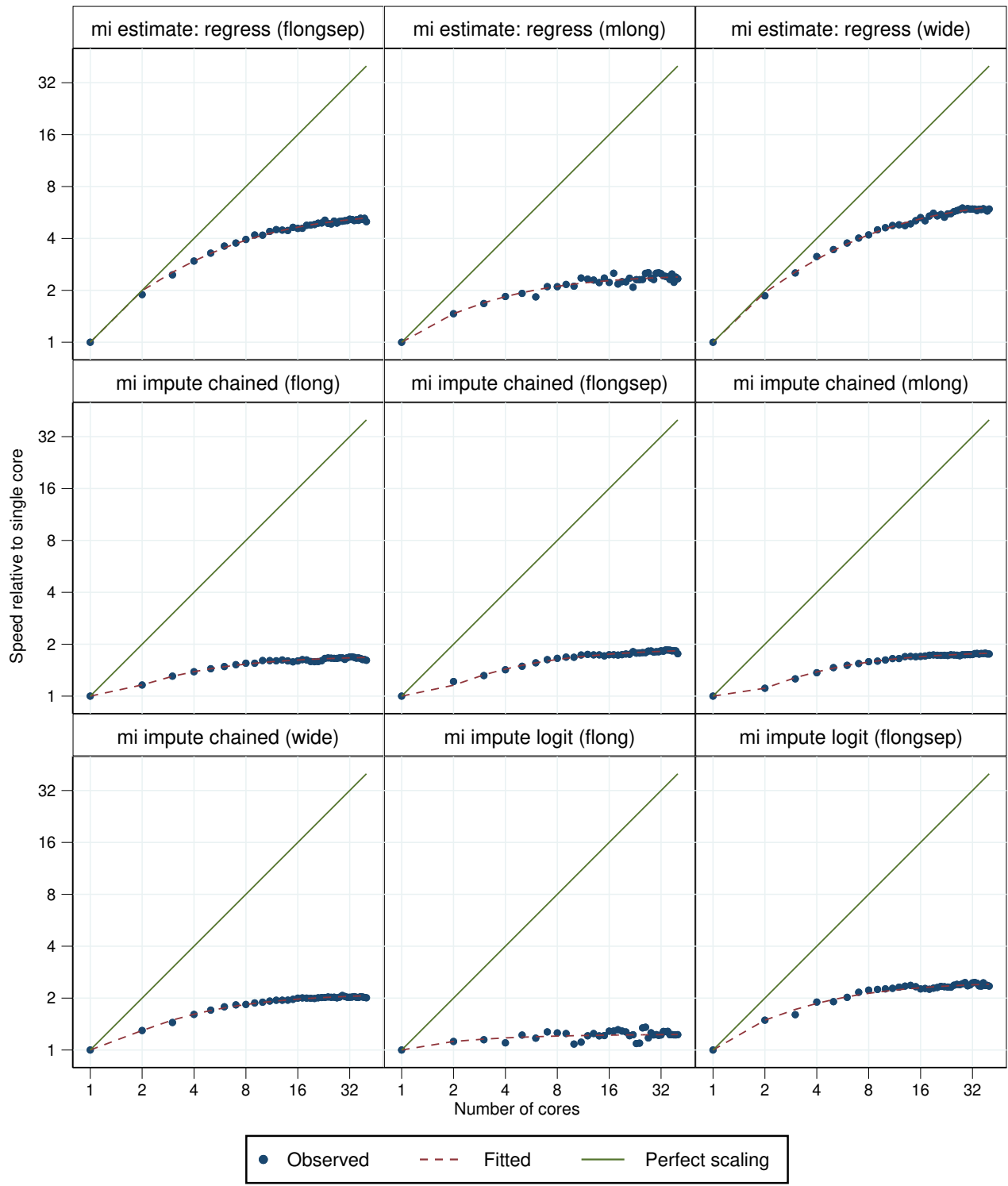


Figure 707. Parallelization performance plots.

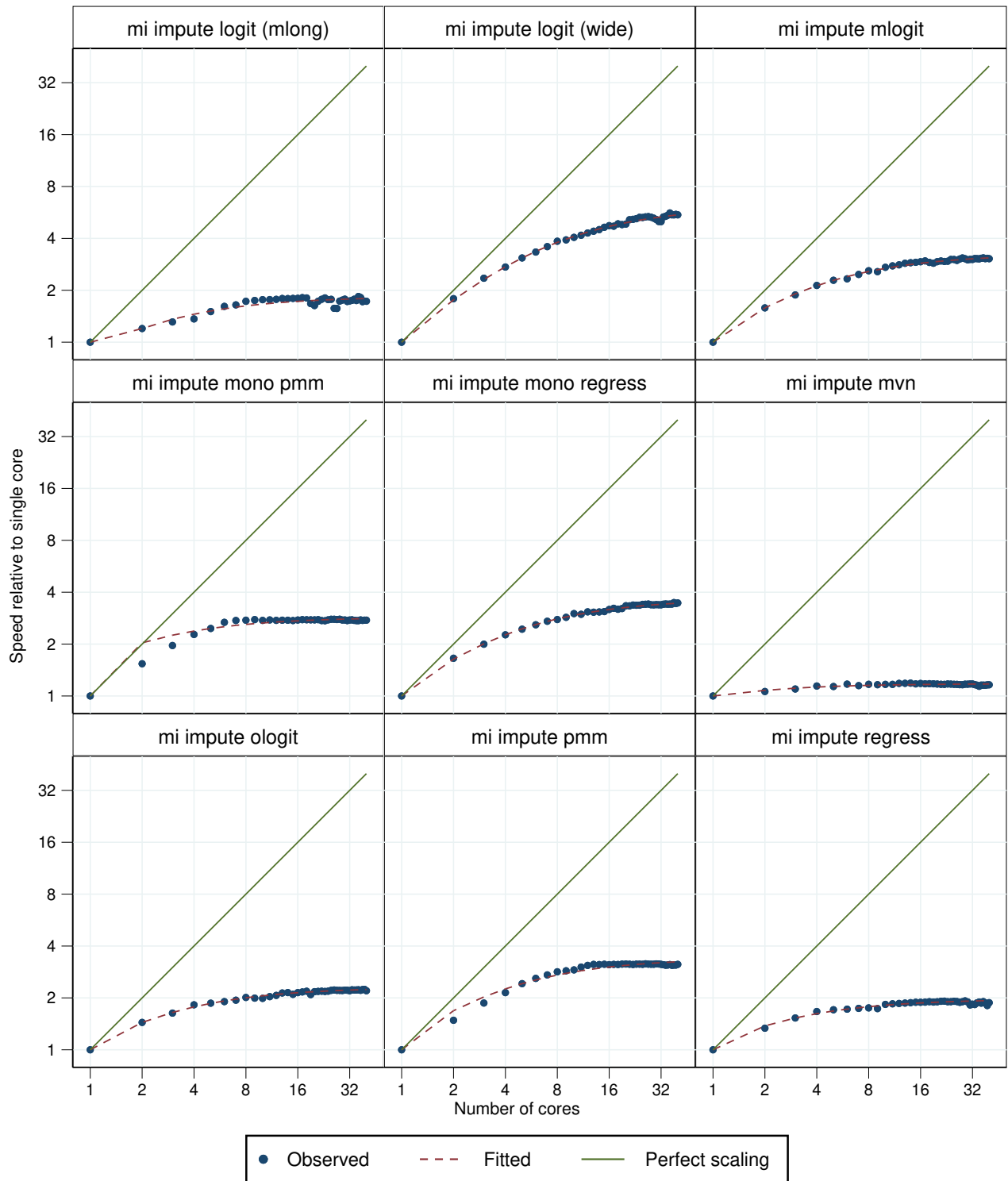


Figure 708. Parallelization performance plots.

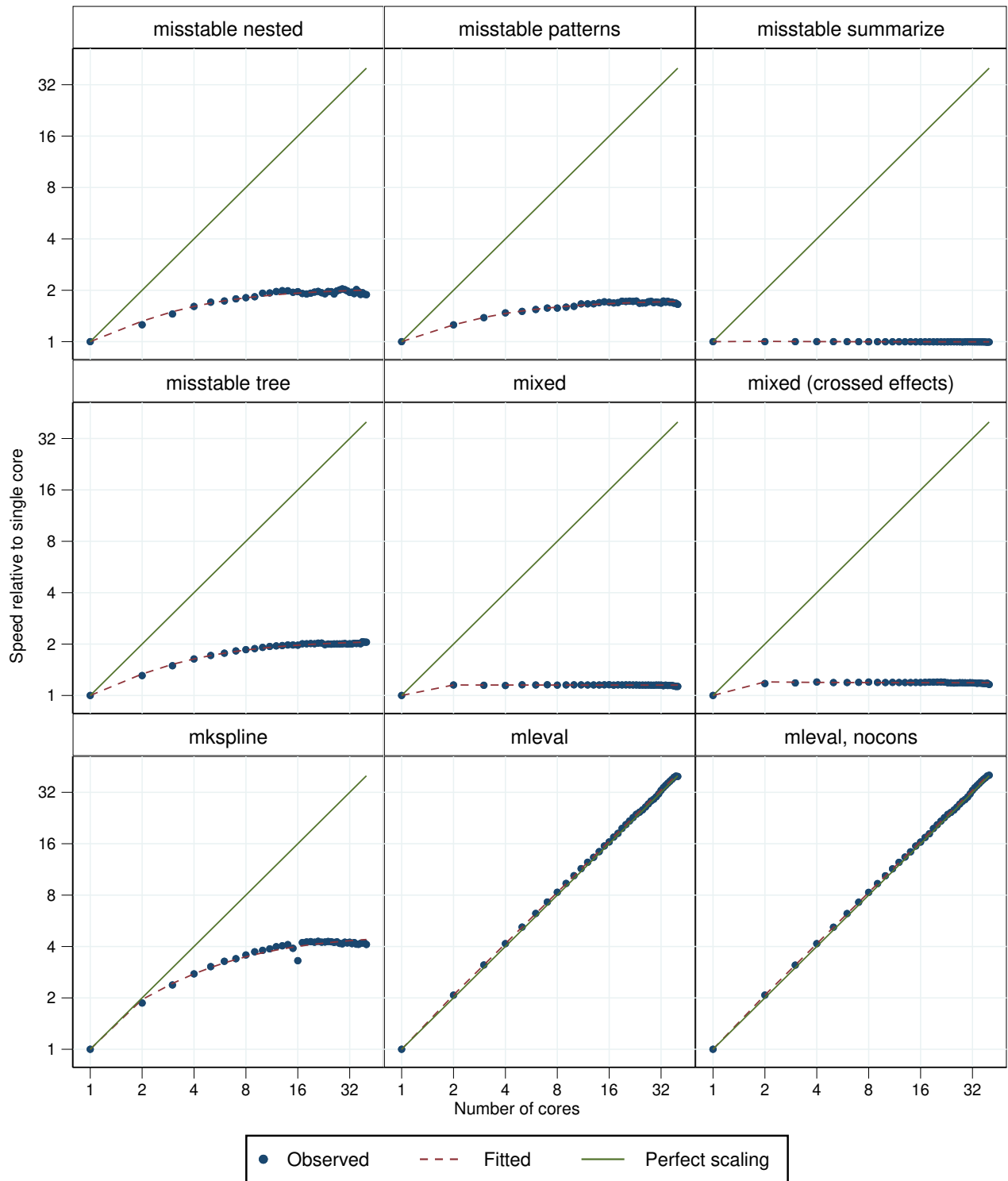


Figure 709. Parallelization performance plots.

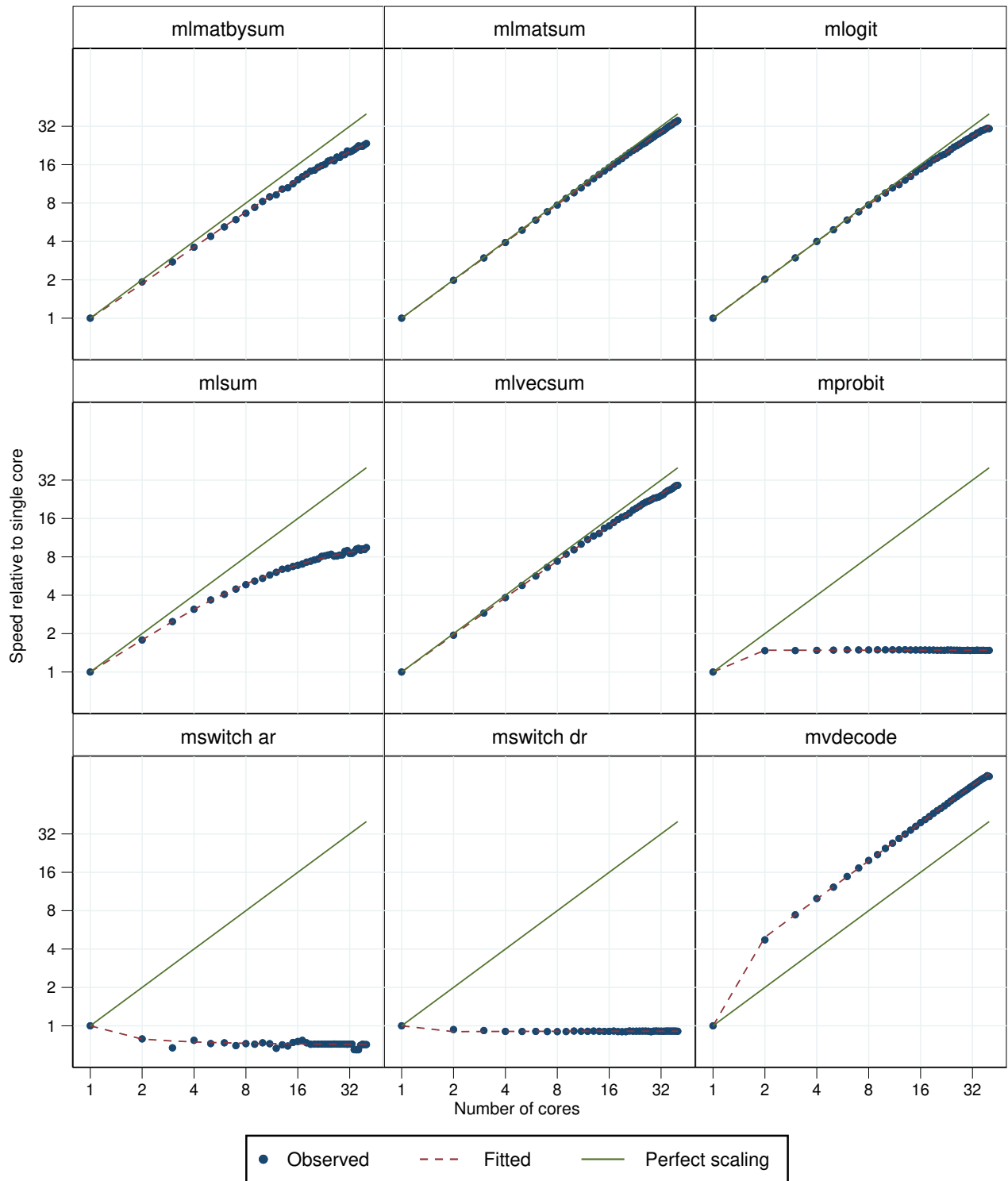


Figure 710. Parallelization performance plots.

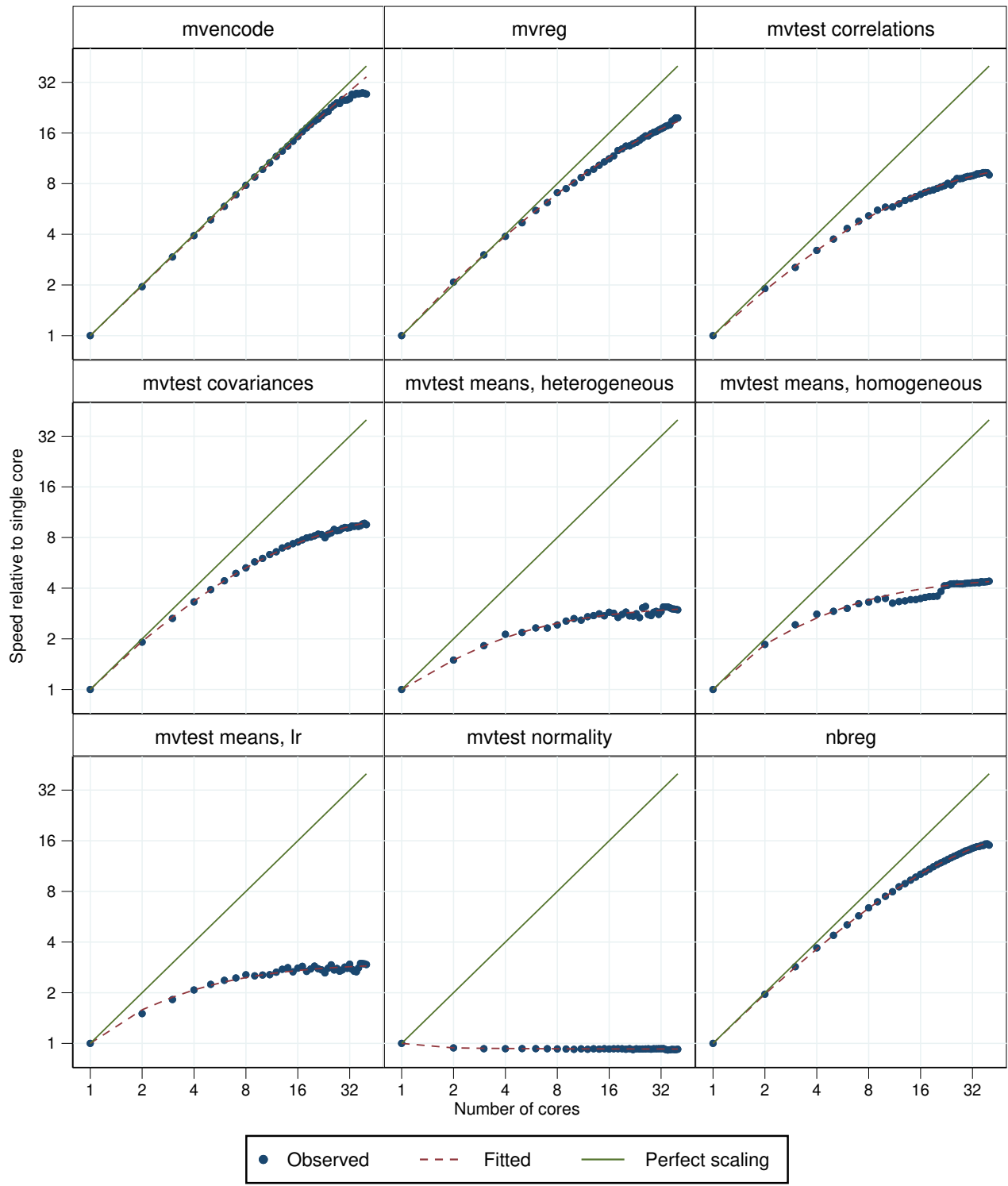


Figure 711. Parallelization performance plots.

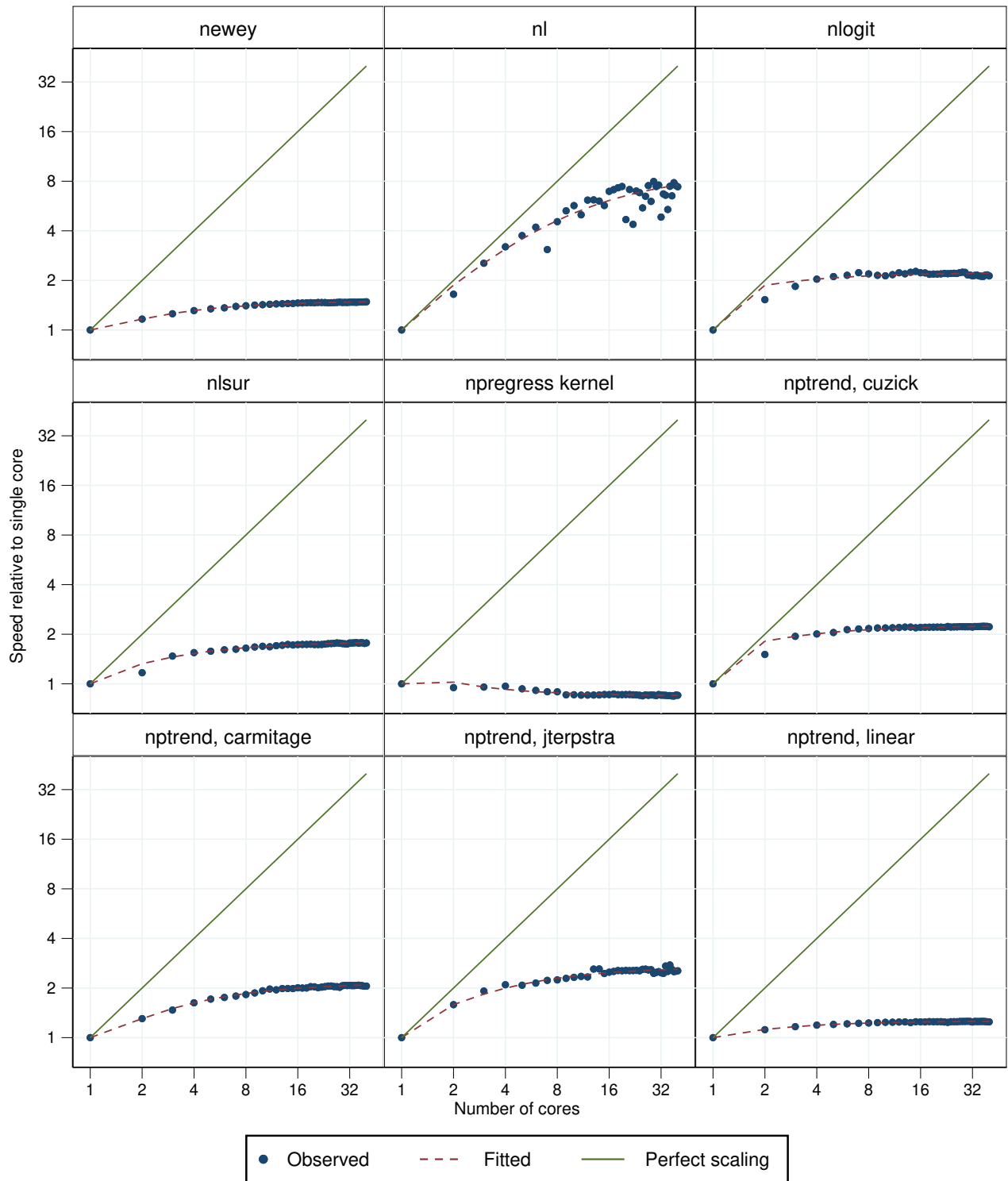


Figure 712. Parallelization performance plots.

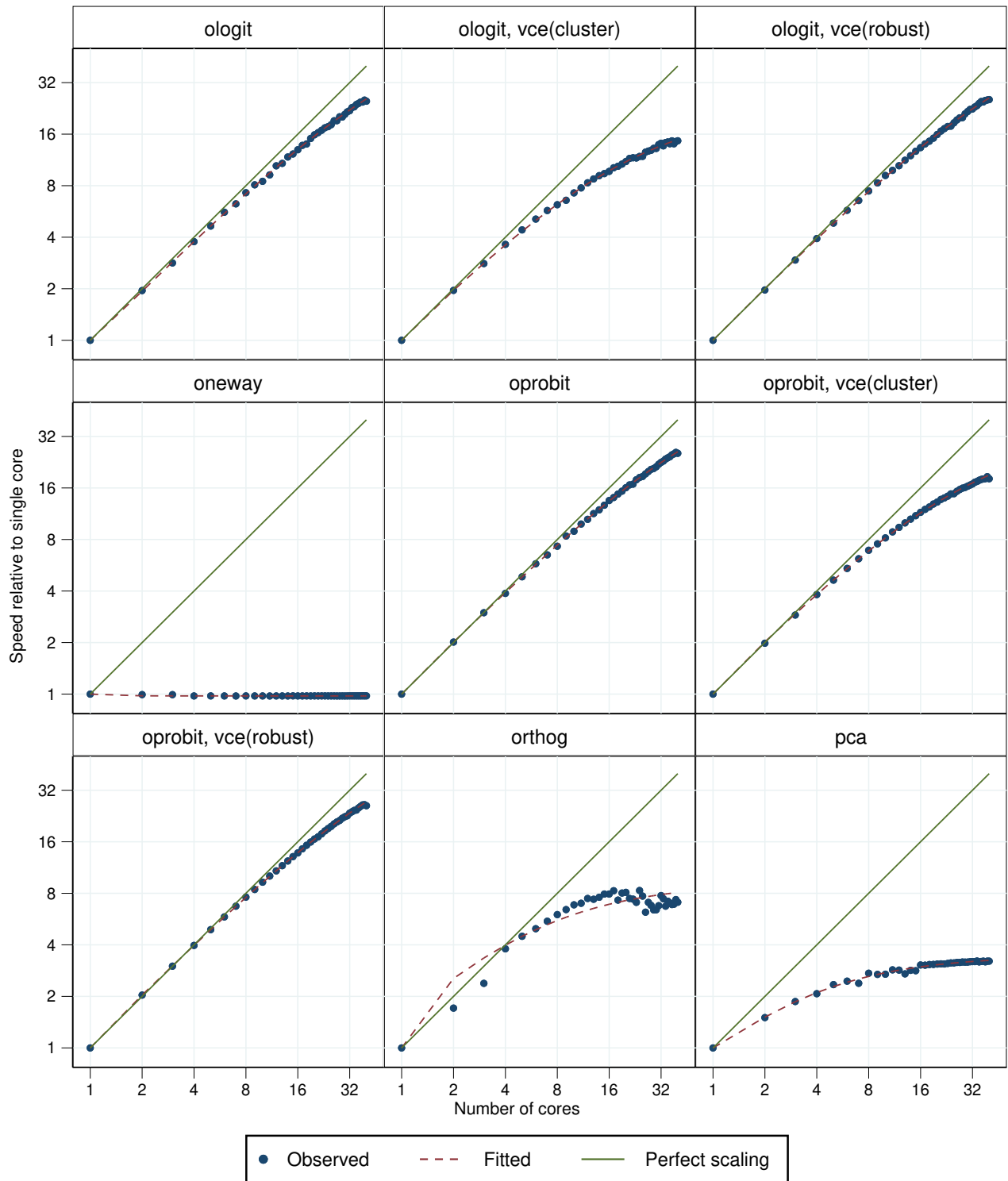


Figure 713. Parallelization performance plots.

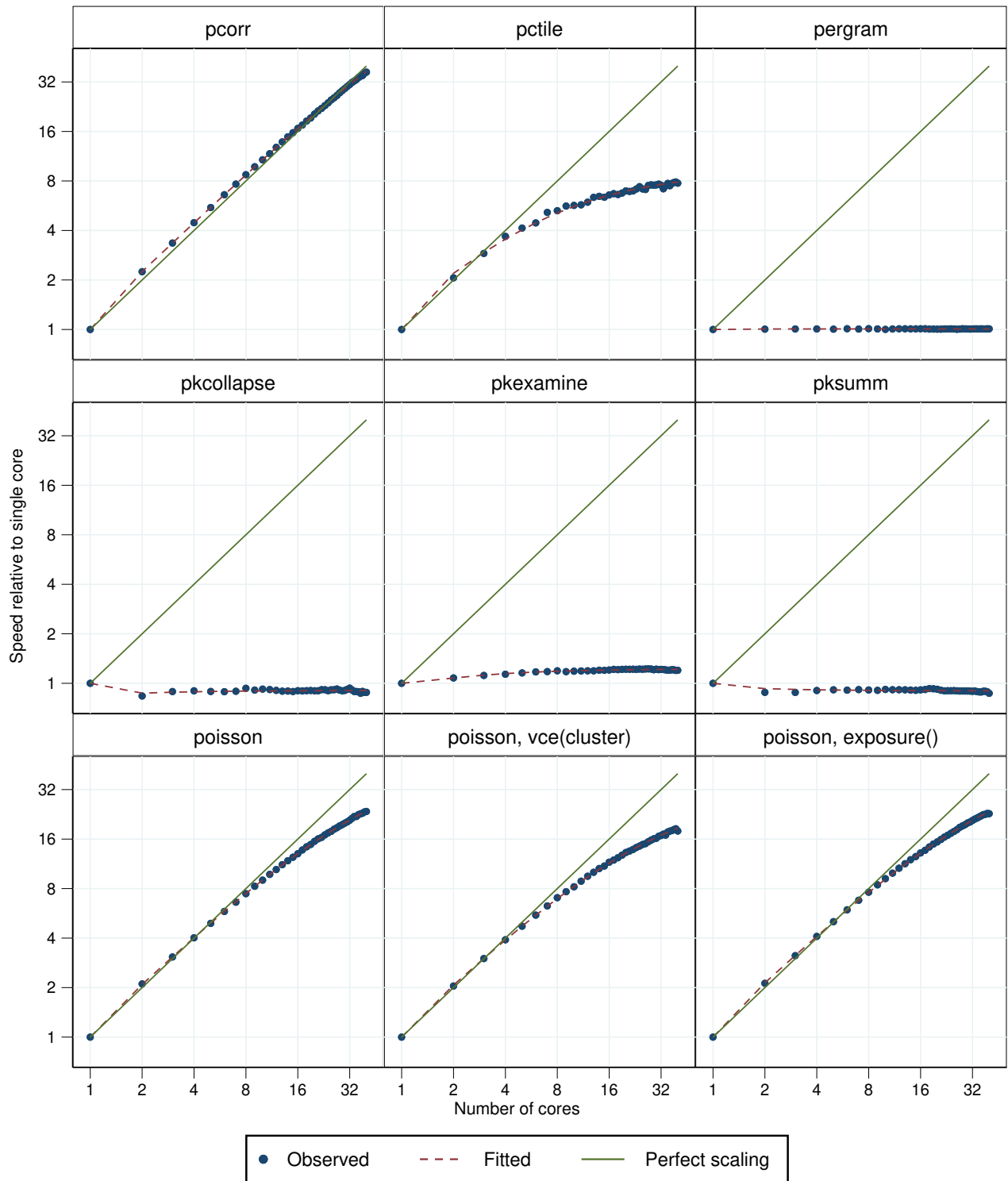


Figure 714. Parallelization performance plots.

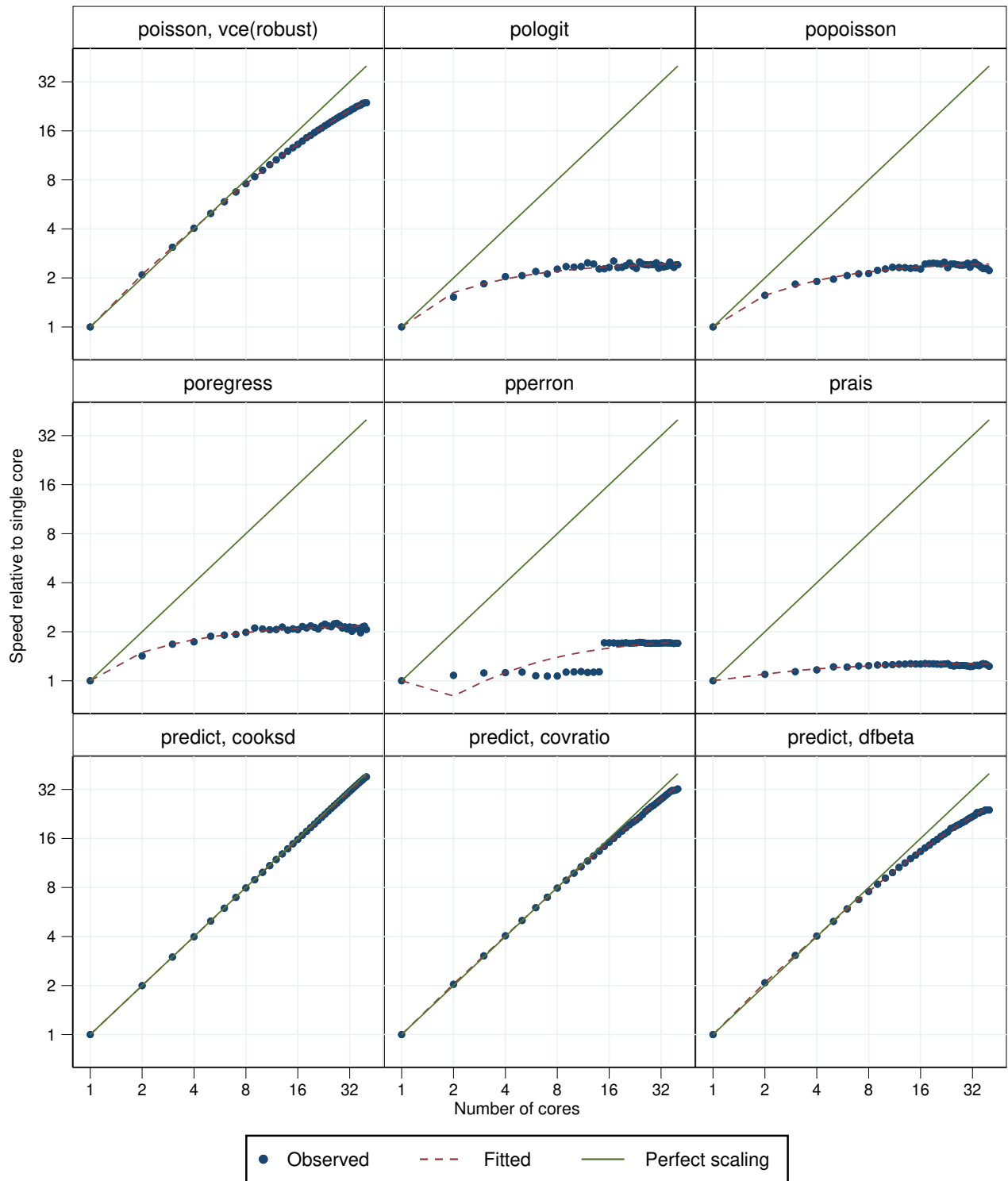


Figure 715. Parallelization performance plots.

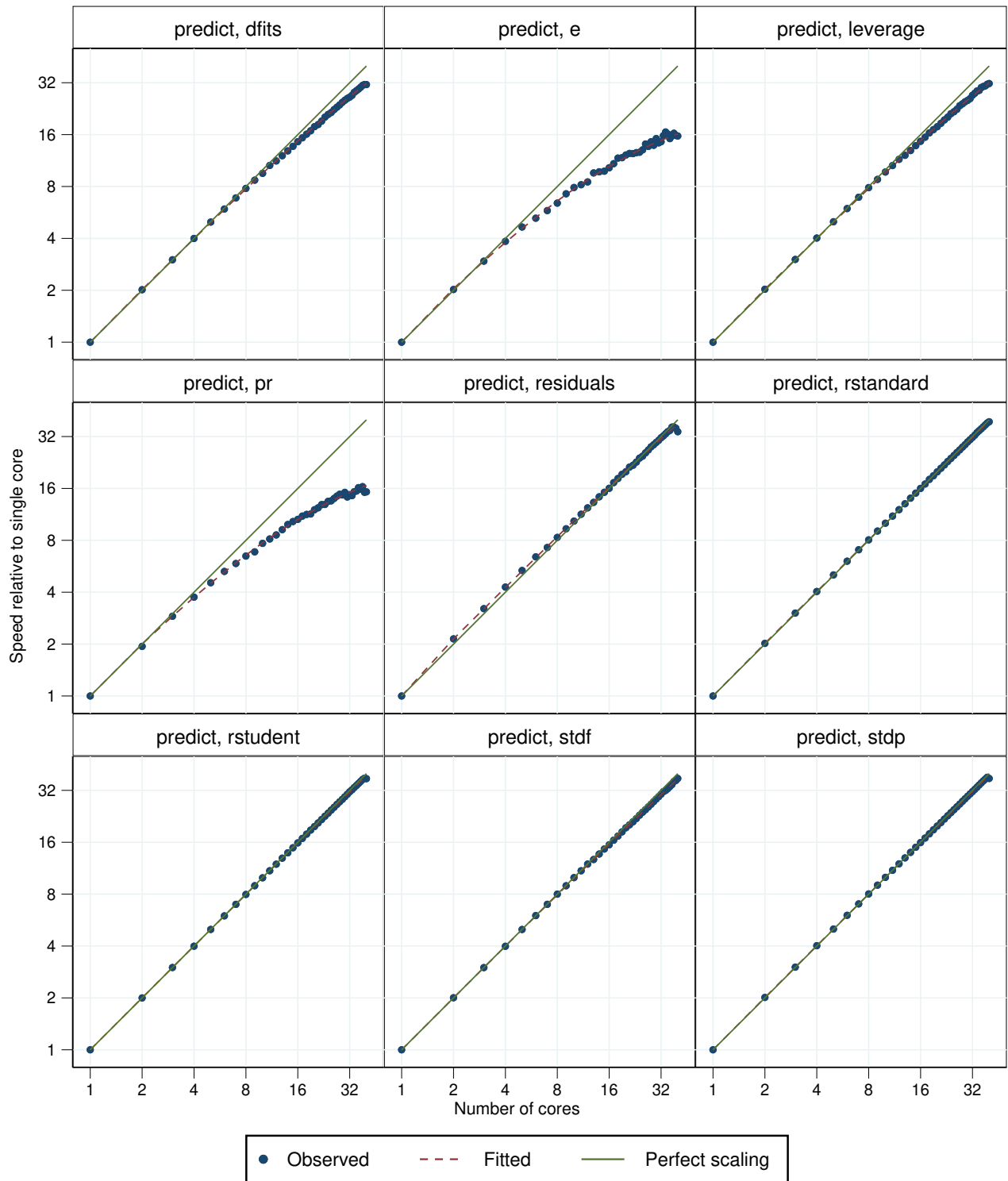


Figure 716. Parallelization performance plots.

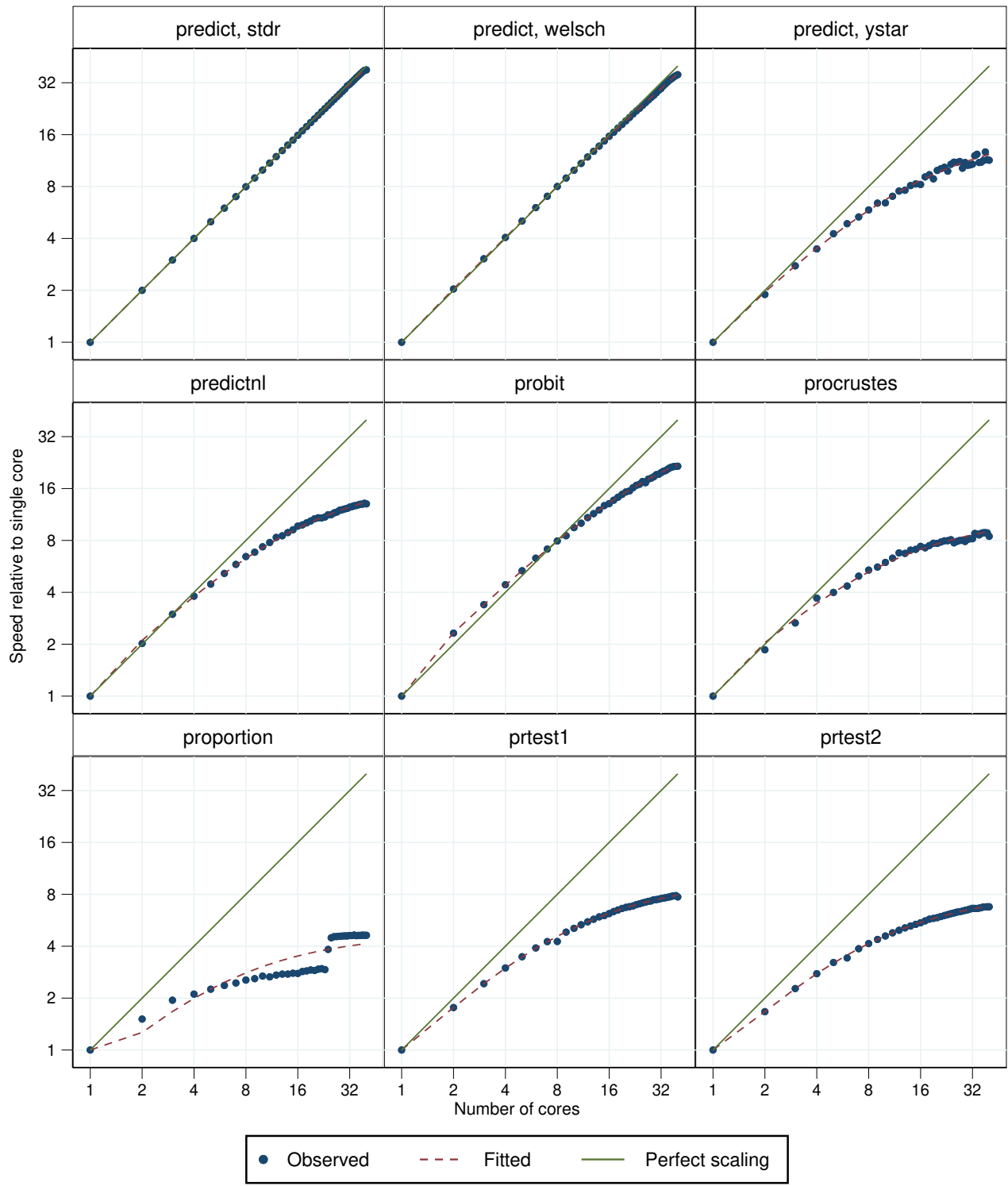


Figure 717. Parallelization performance plots.

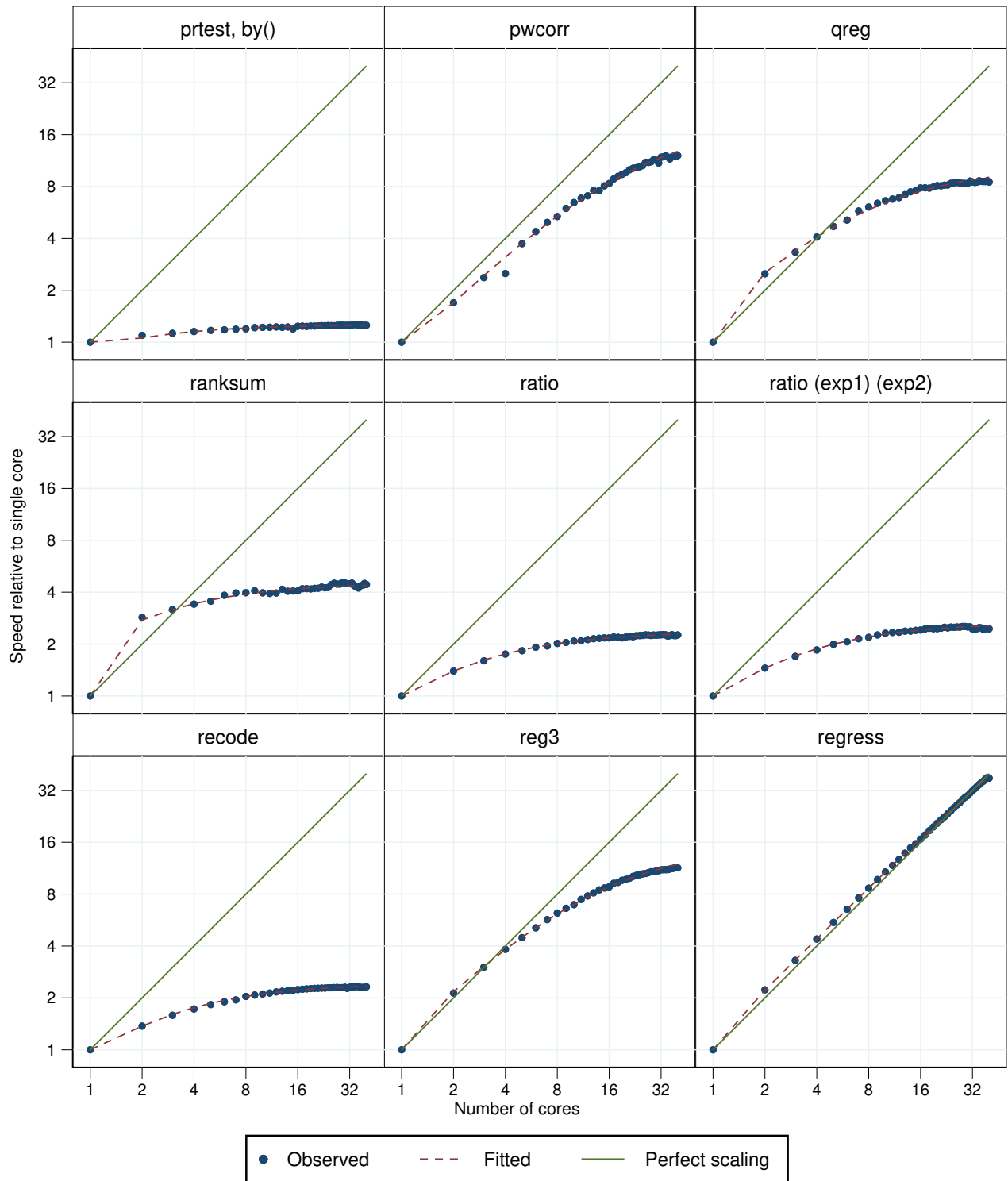


Figure 718. Parallelization performance plots.

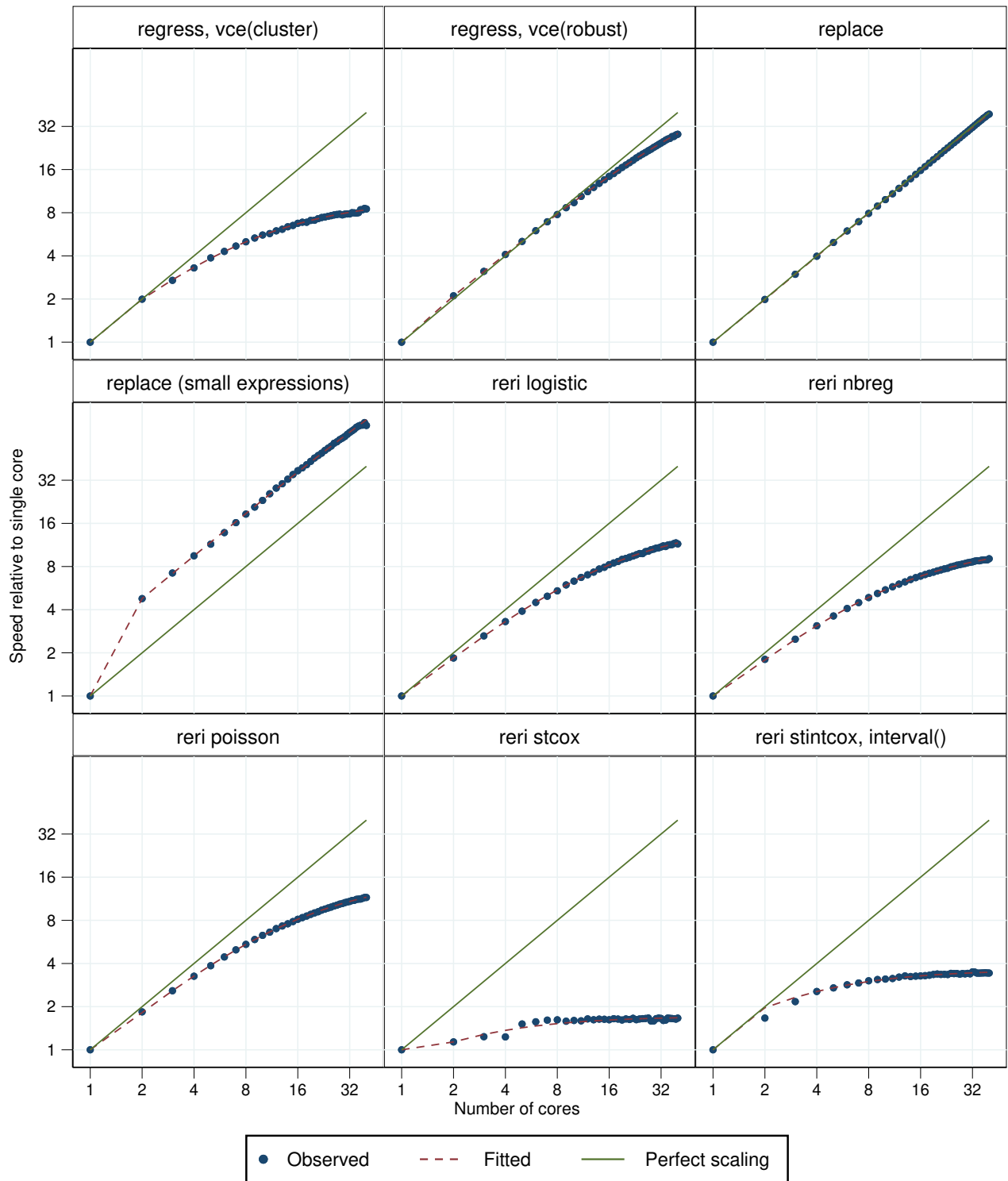


Figure 719. Parallelization performance plots.

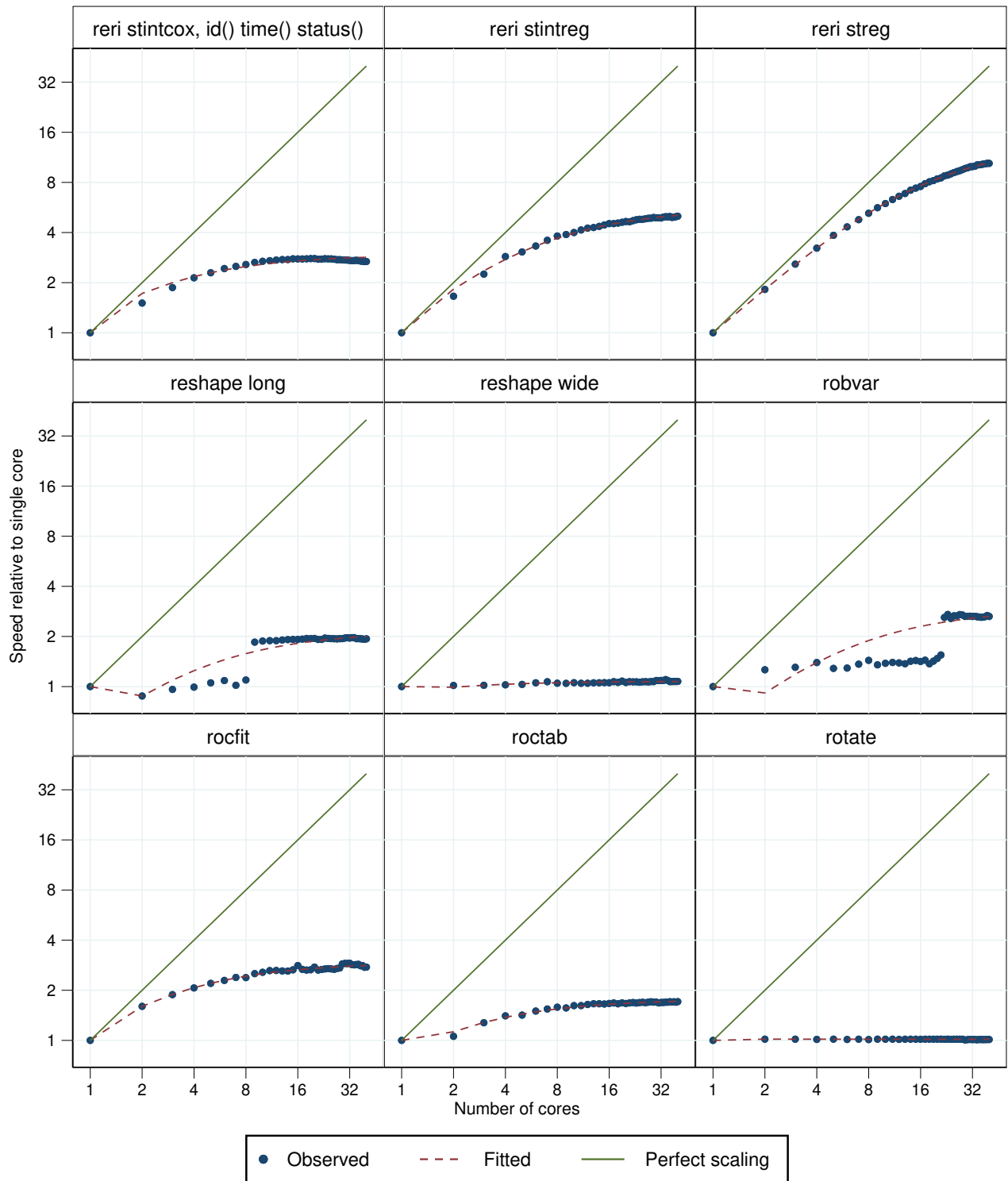


Figure 720. Parallelization performance plots.

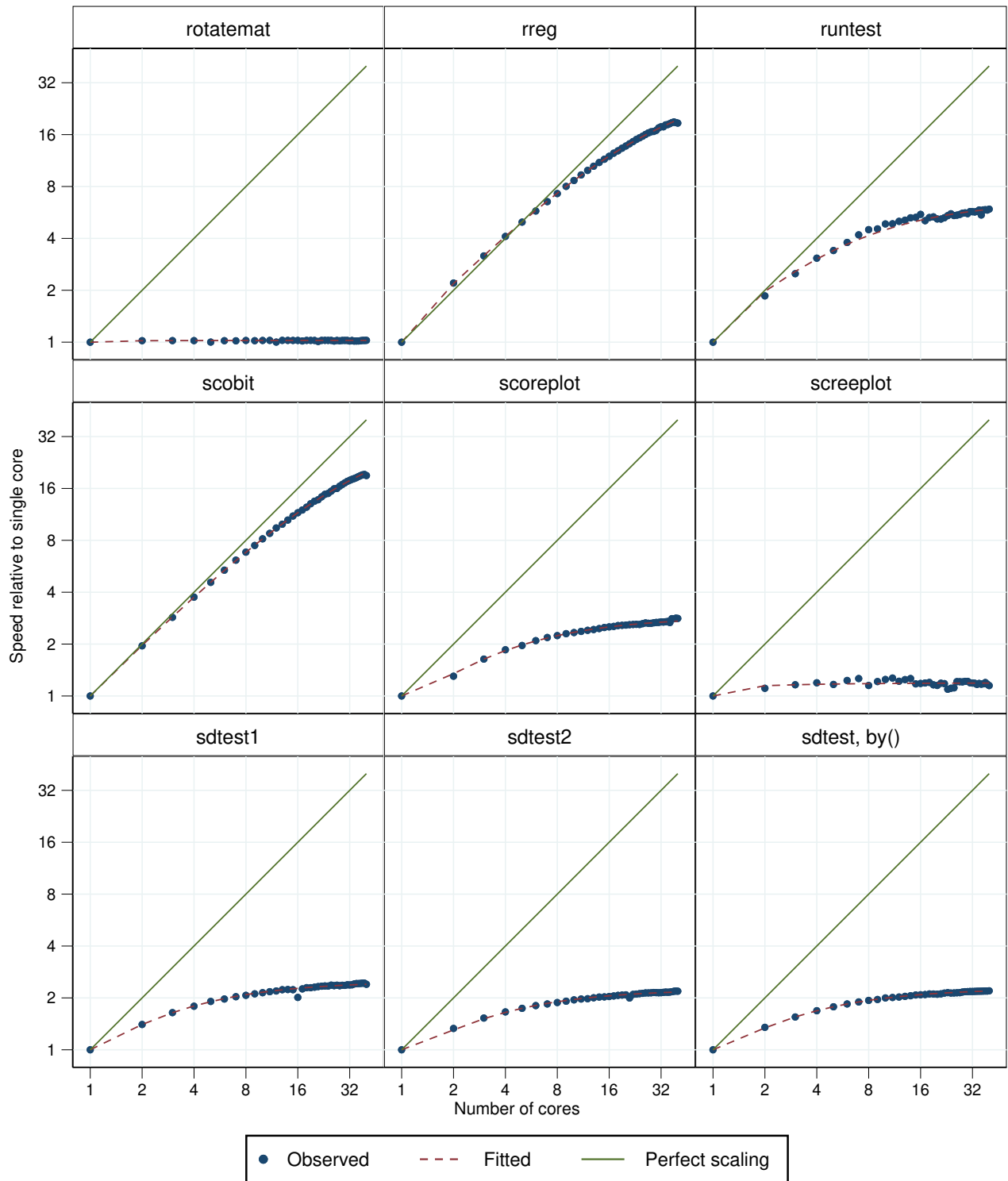


Figure 721. Parallelization performance plots.

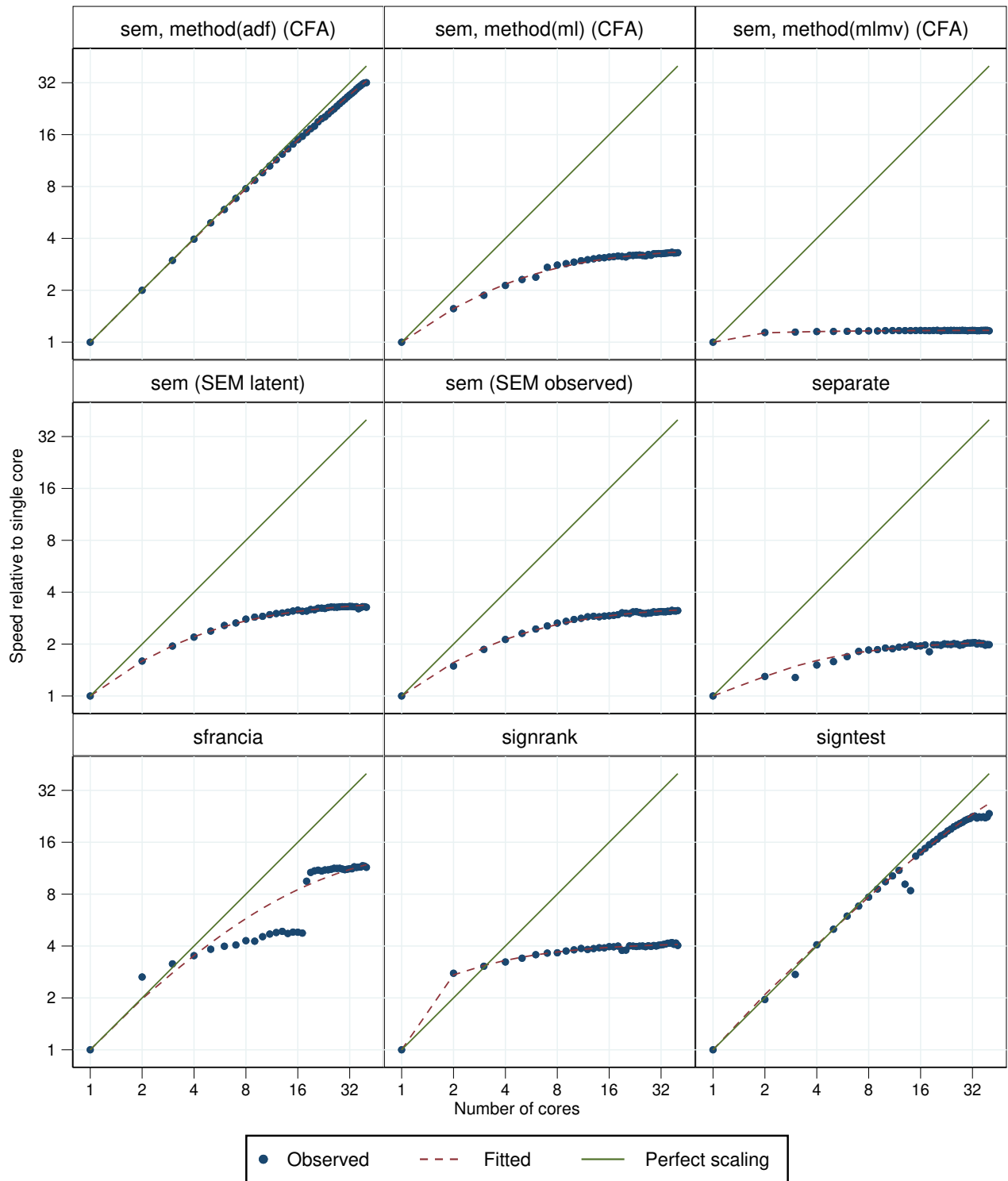


Figure 722. Parallelization performance plots.

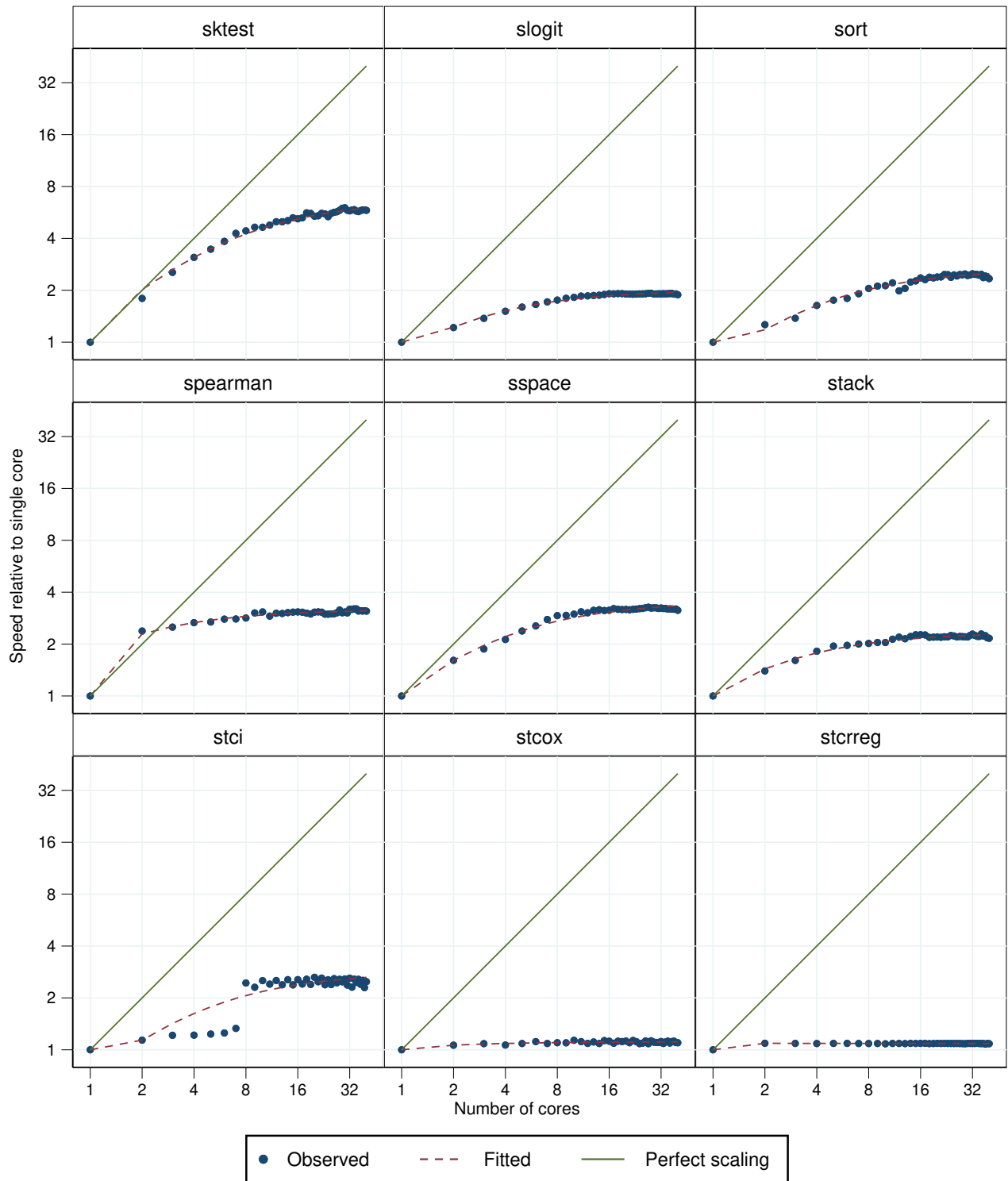


Figure 723. Parallelization performance plots.

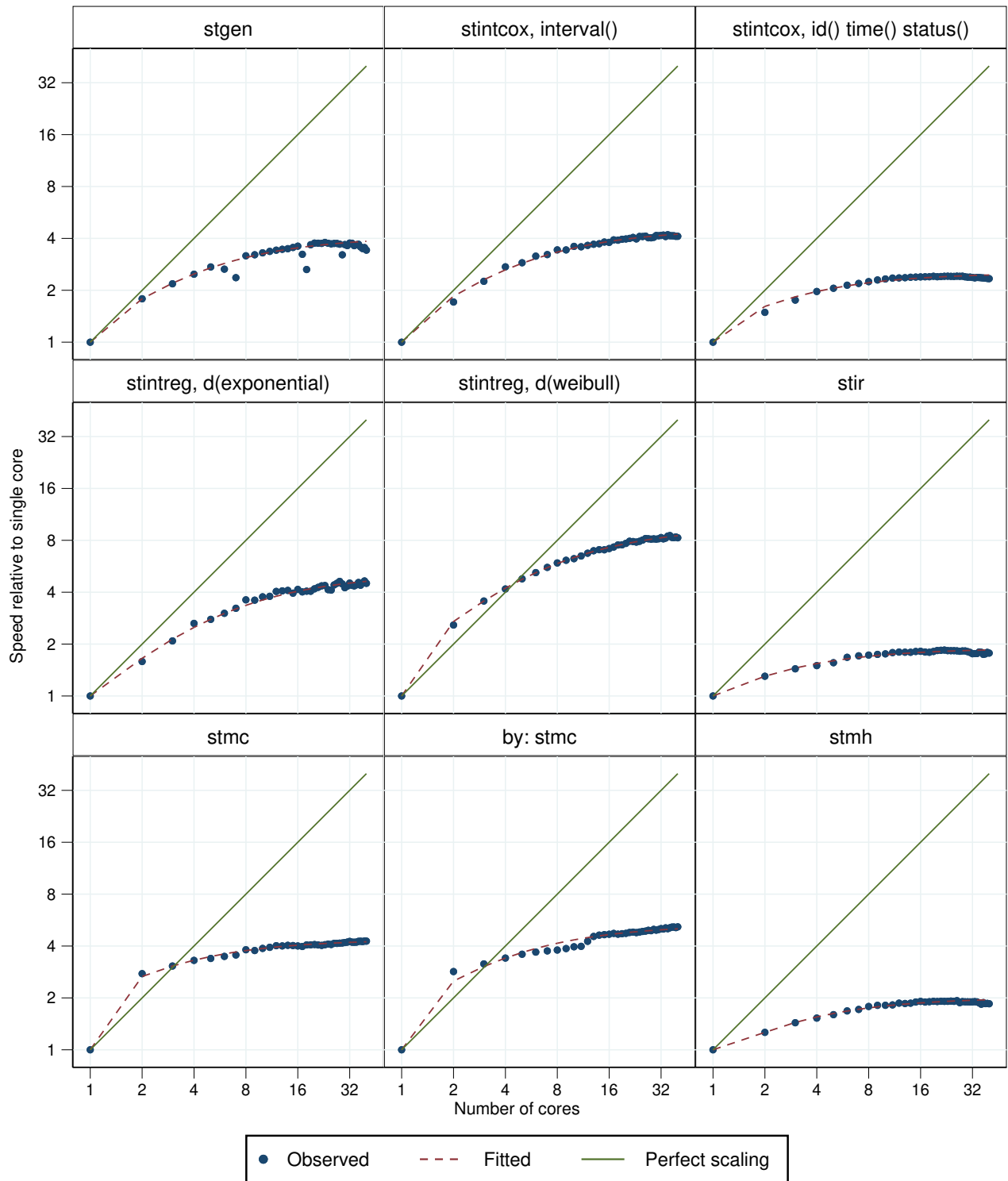


Figure 724. Parallelization performance plots.

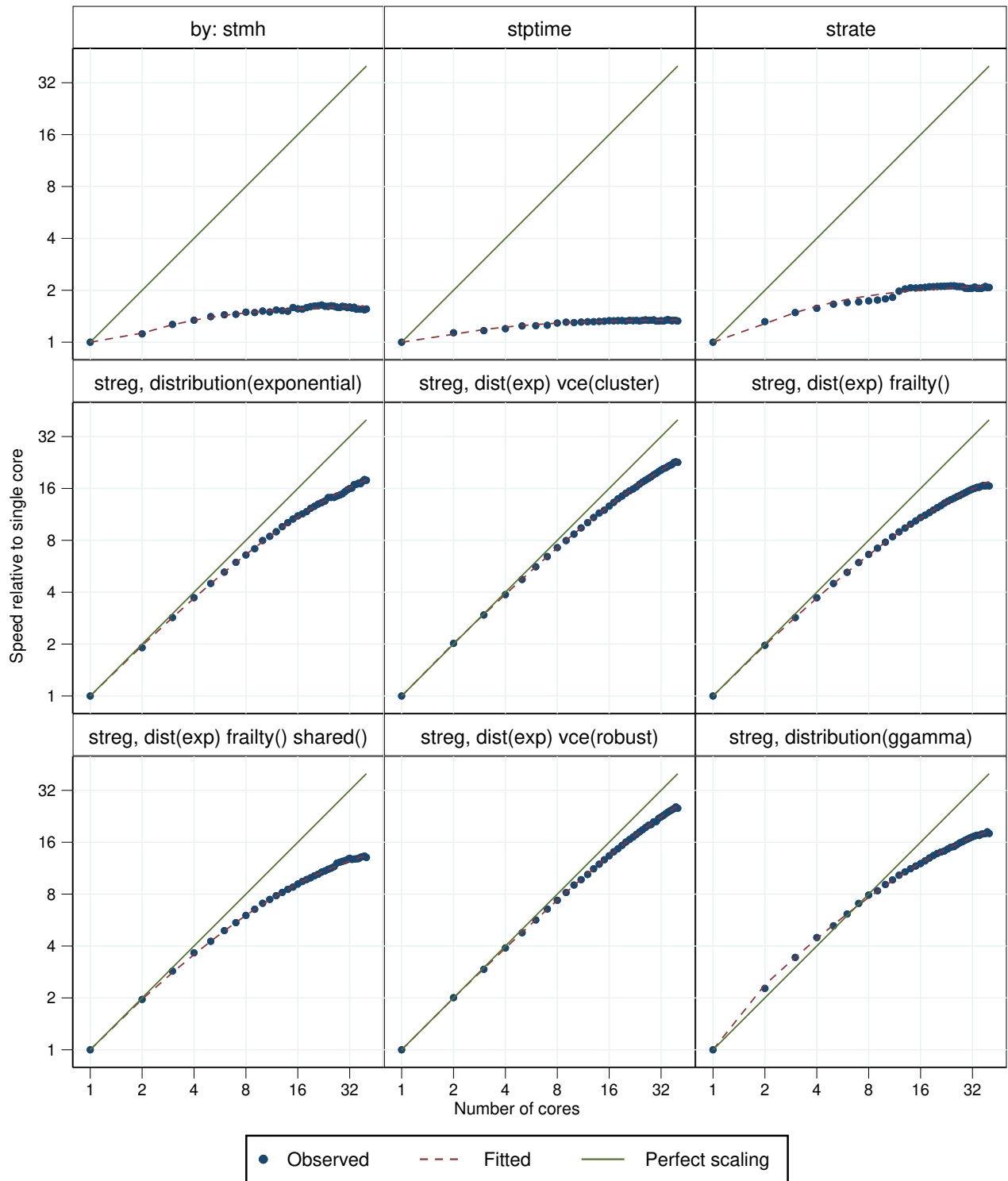


Figure 725. Parallelization performance plots.

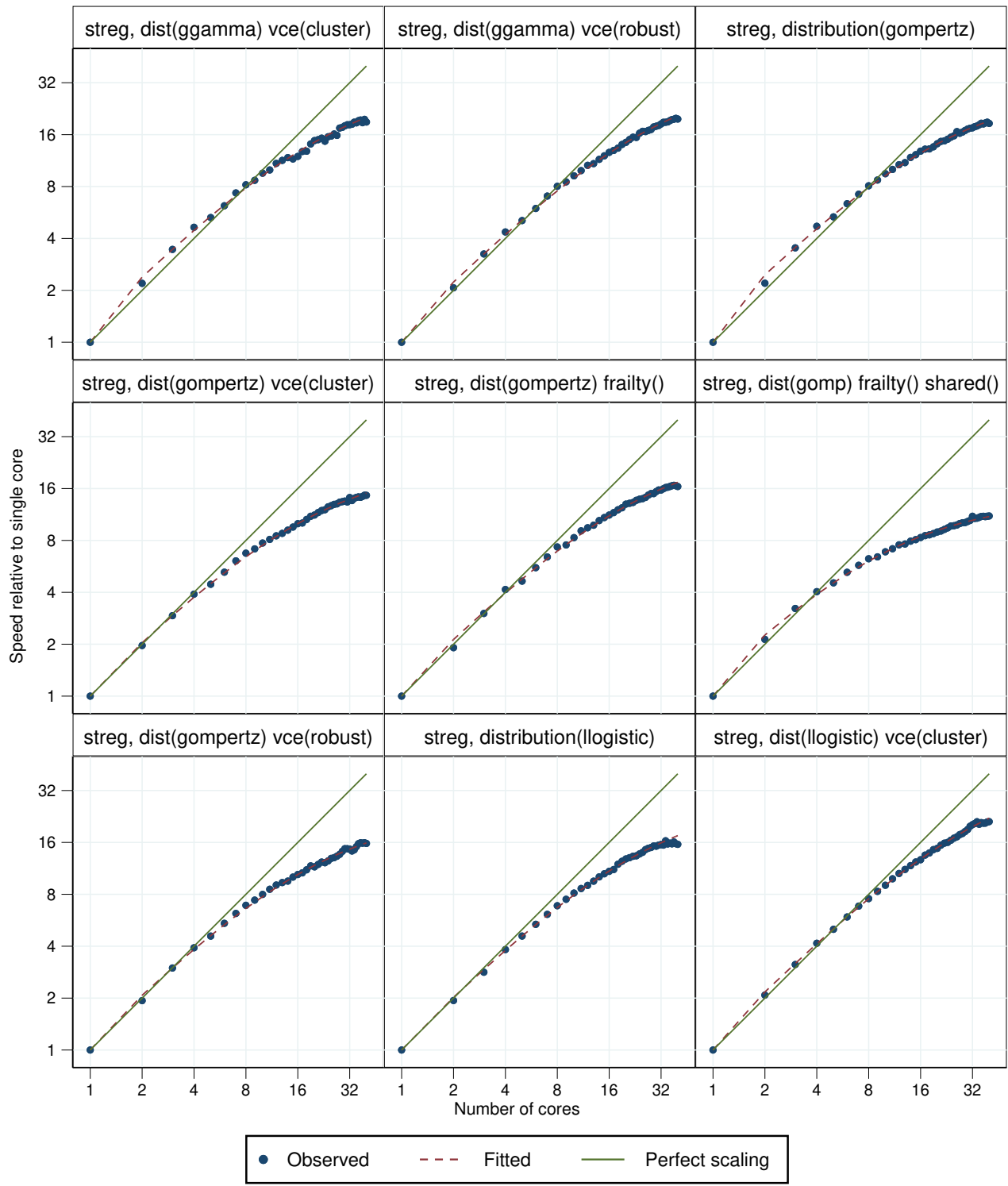


Figure 726. Parallelization performance plots.

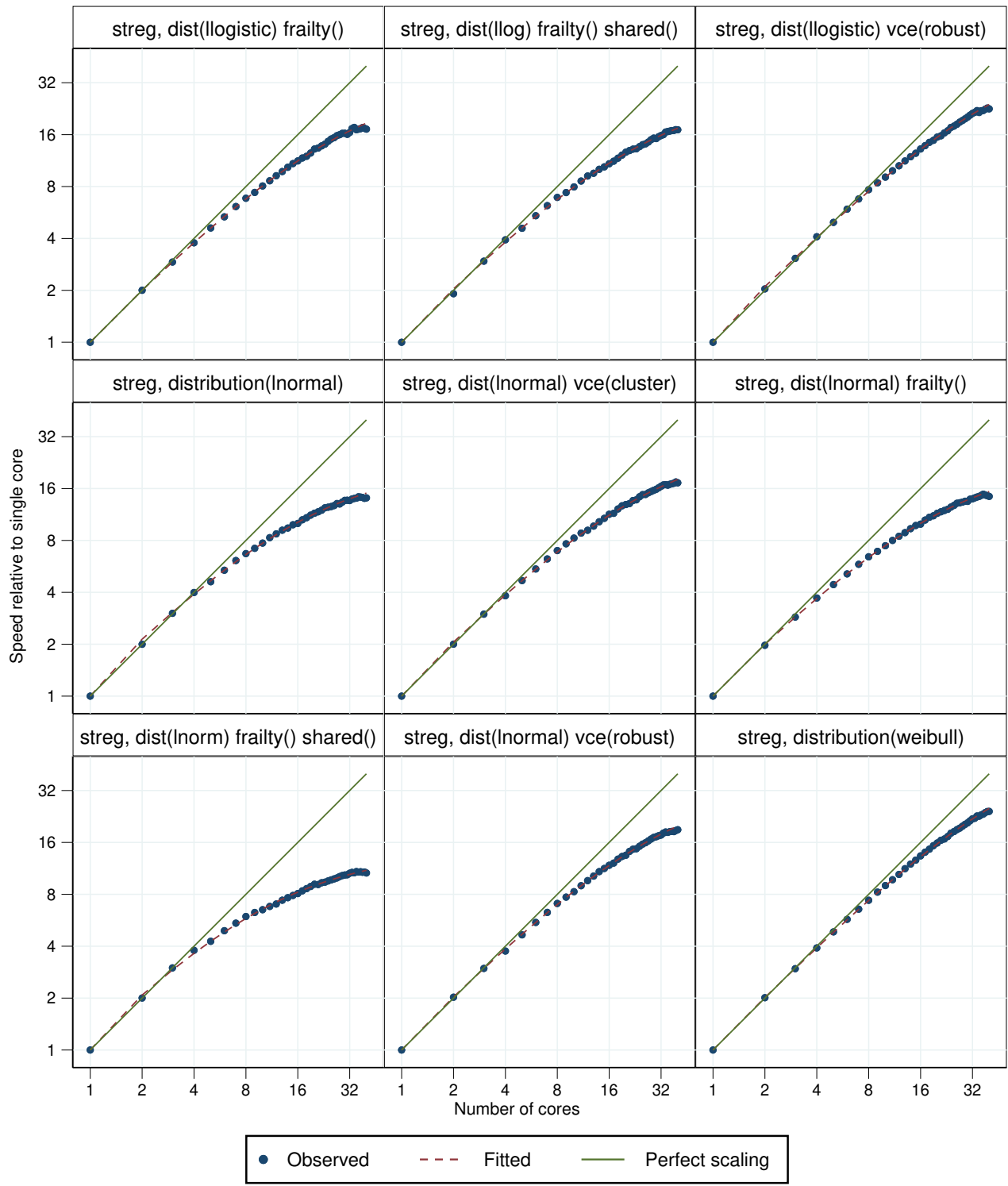


Figure 727. Parallelization performance plots.

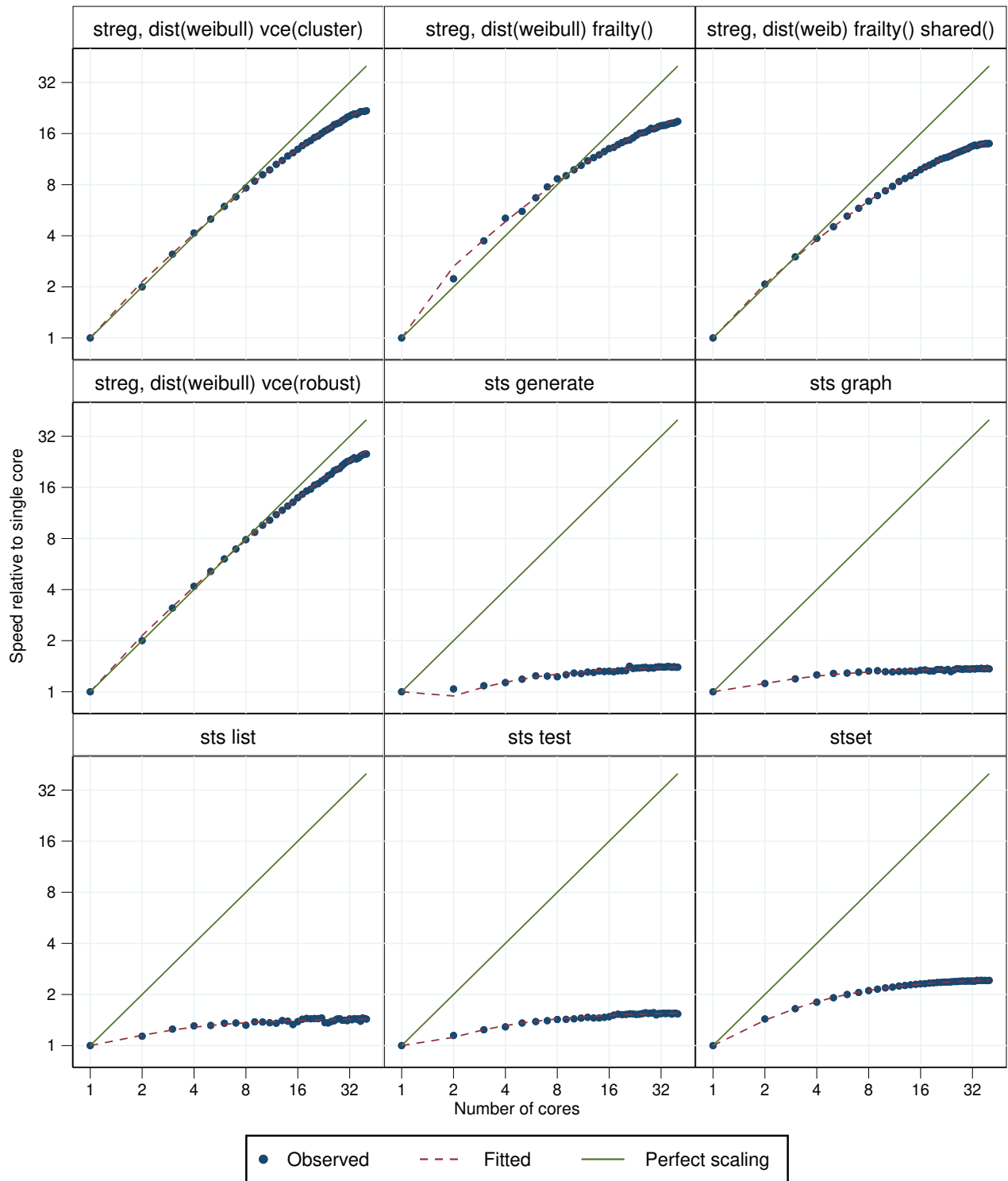


Figure 728. Parallelization performance plots.

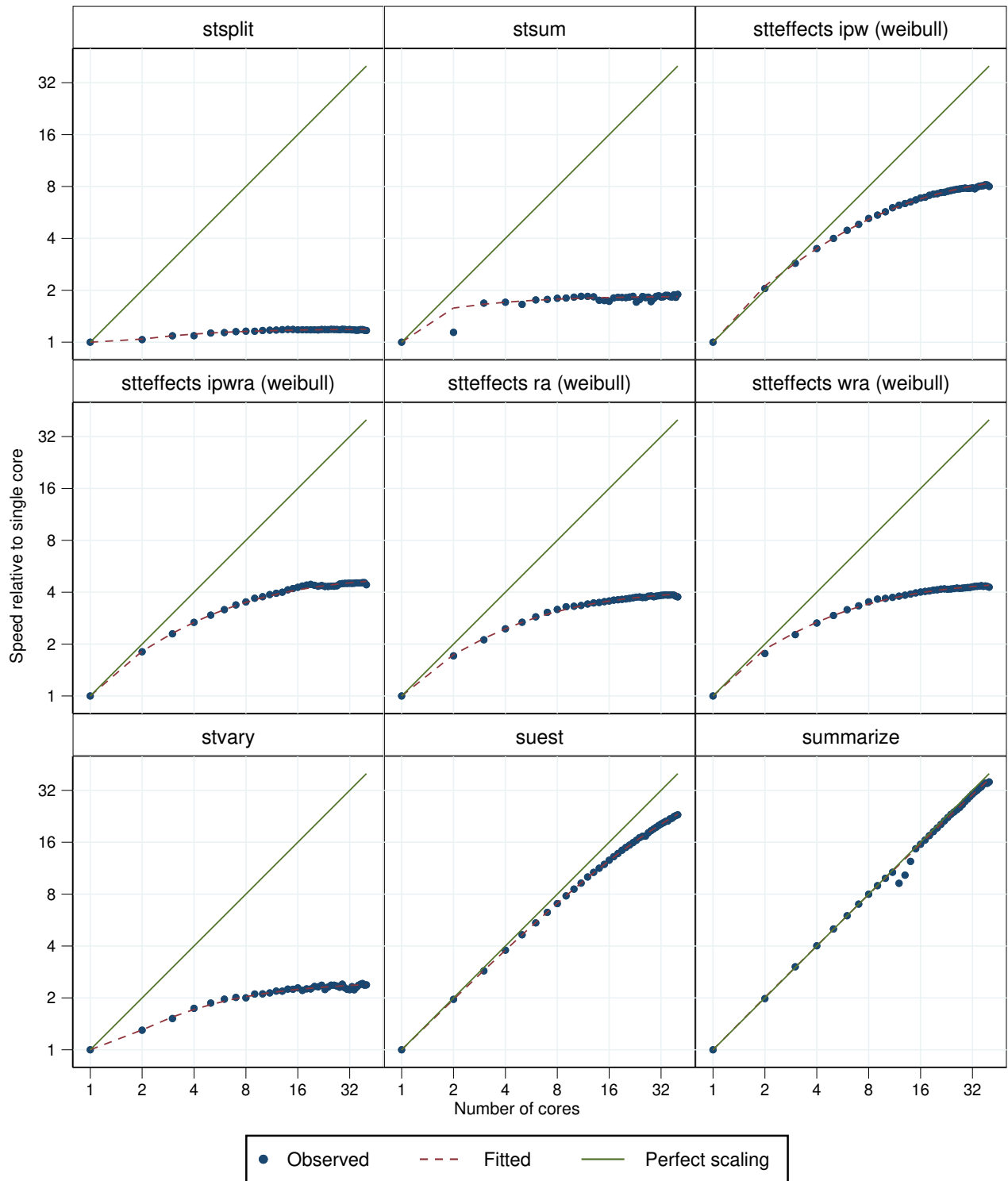


Figure 729. Parallelization performance plots.

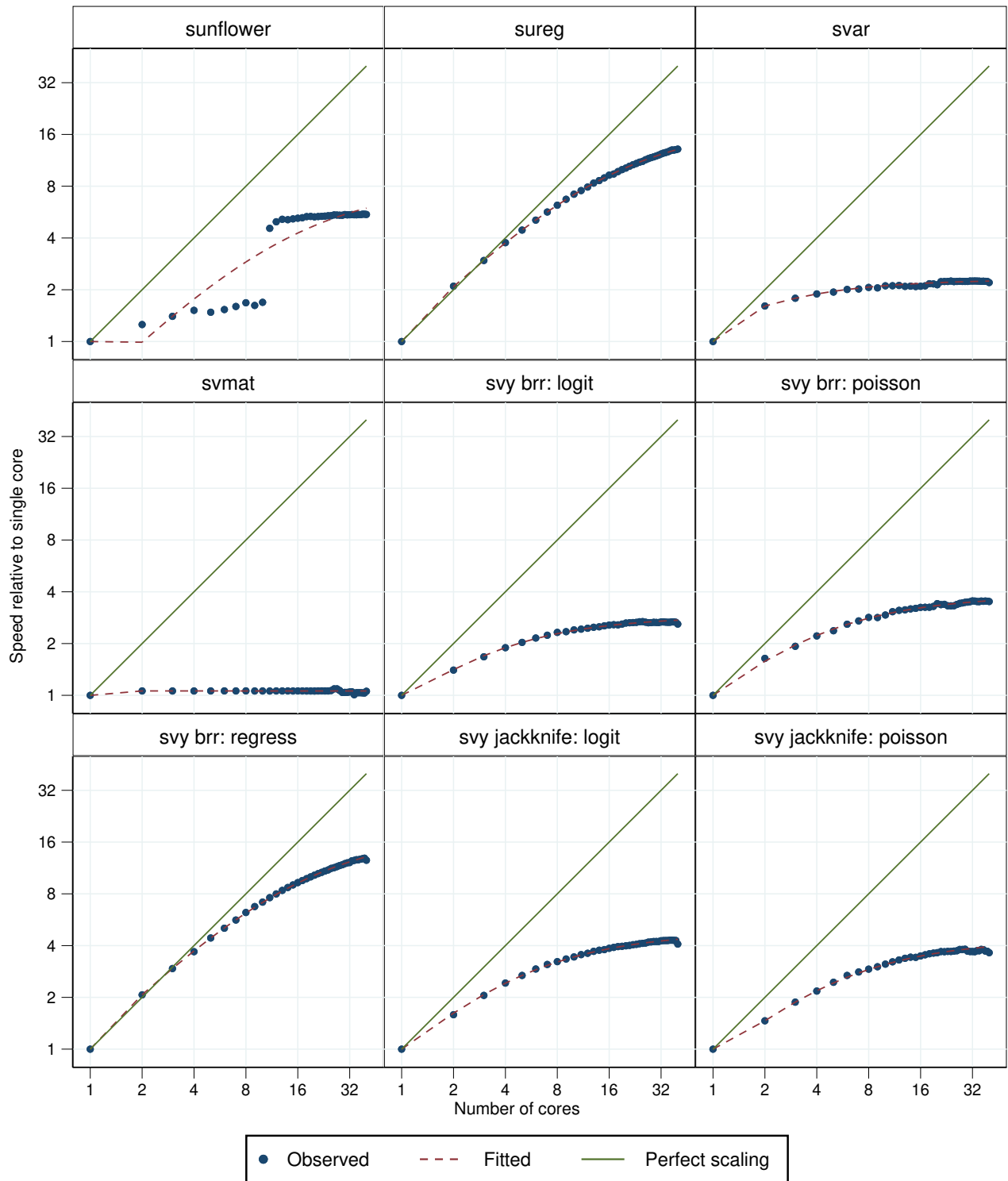


Figure 730. Parallelization performance plots.

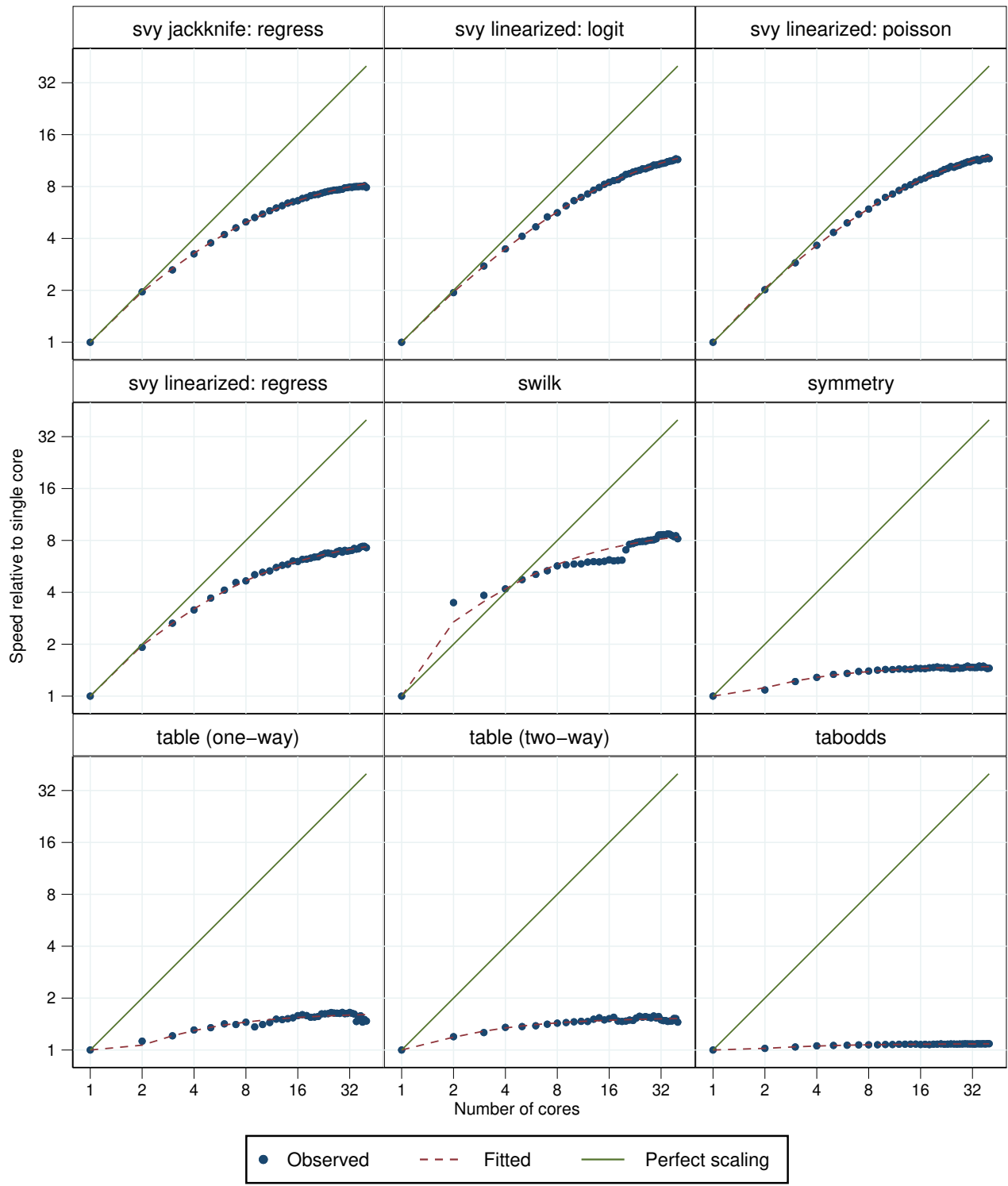


Figure 731. Parallelization performance plots.

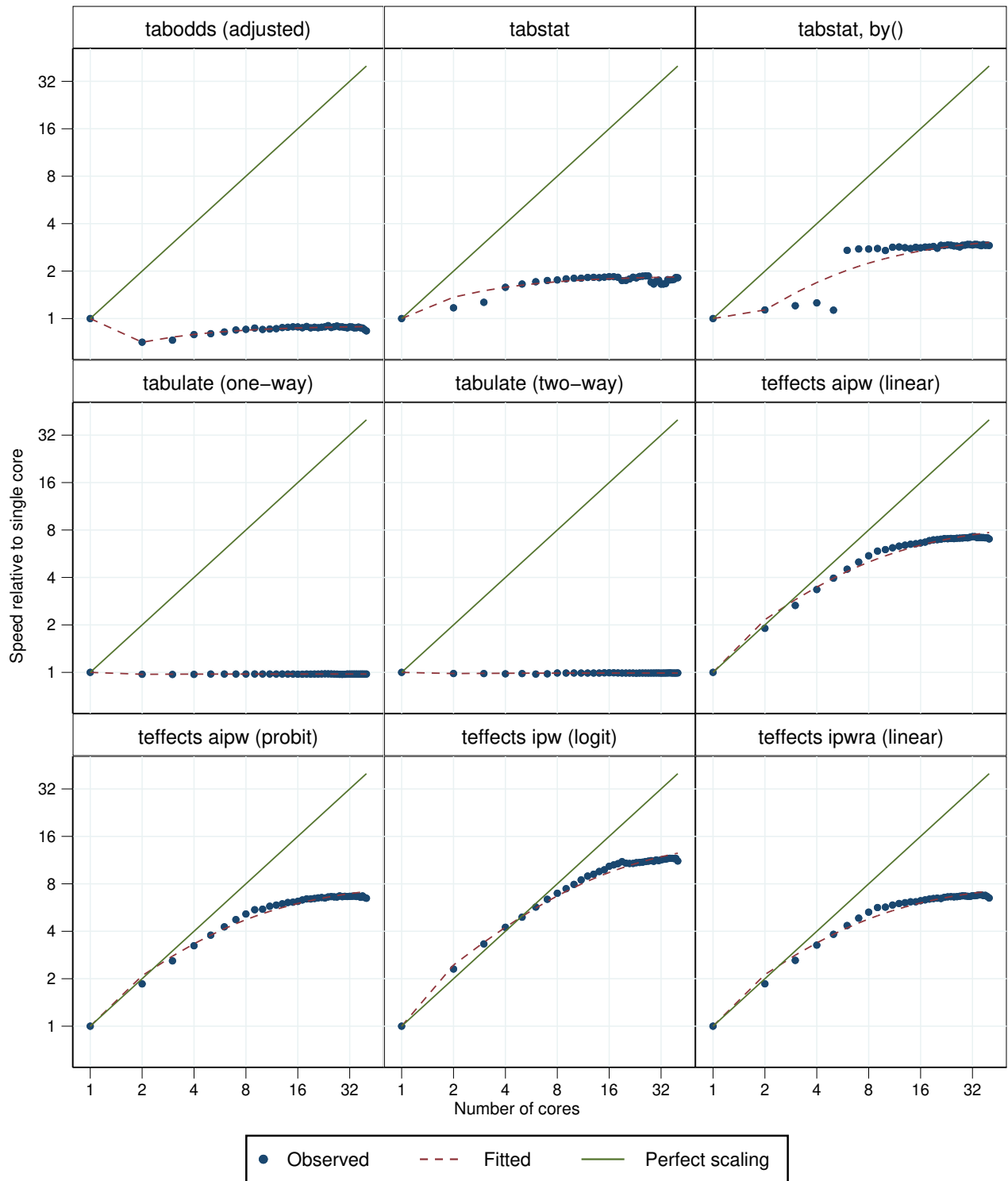


Figure 732. Parallelization performance plots.

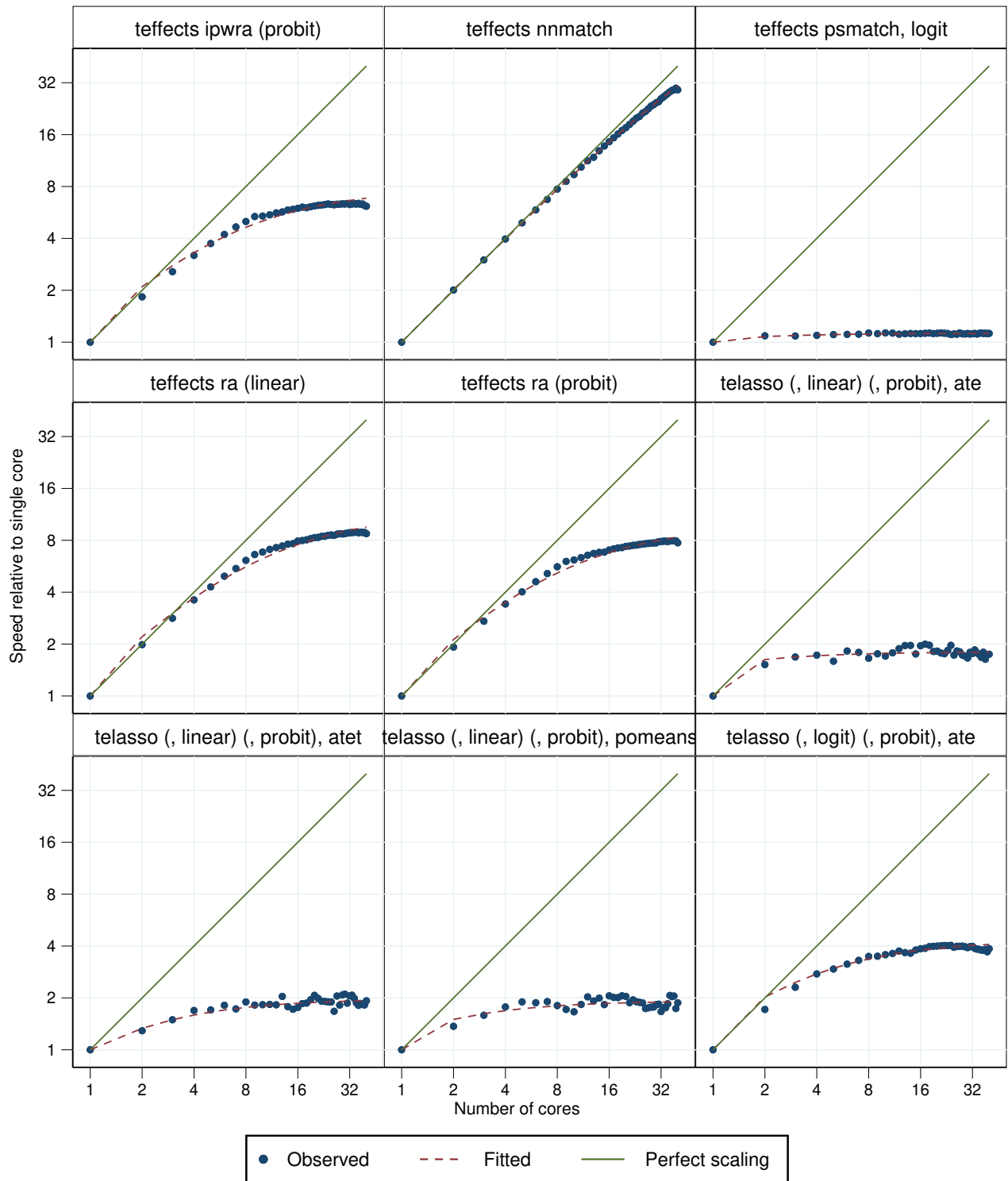


Figure 733. Parallelization performance plots.

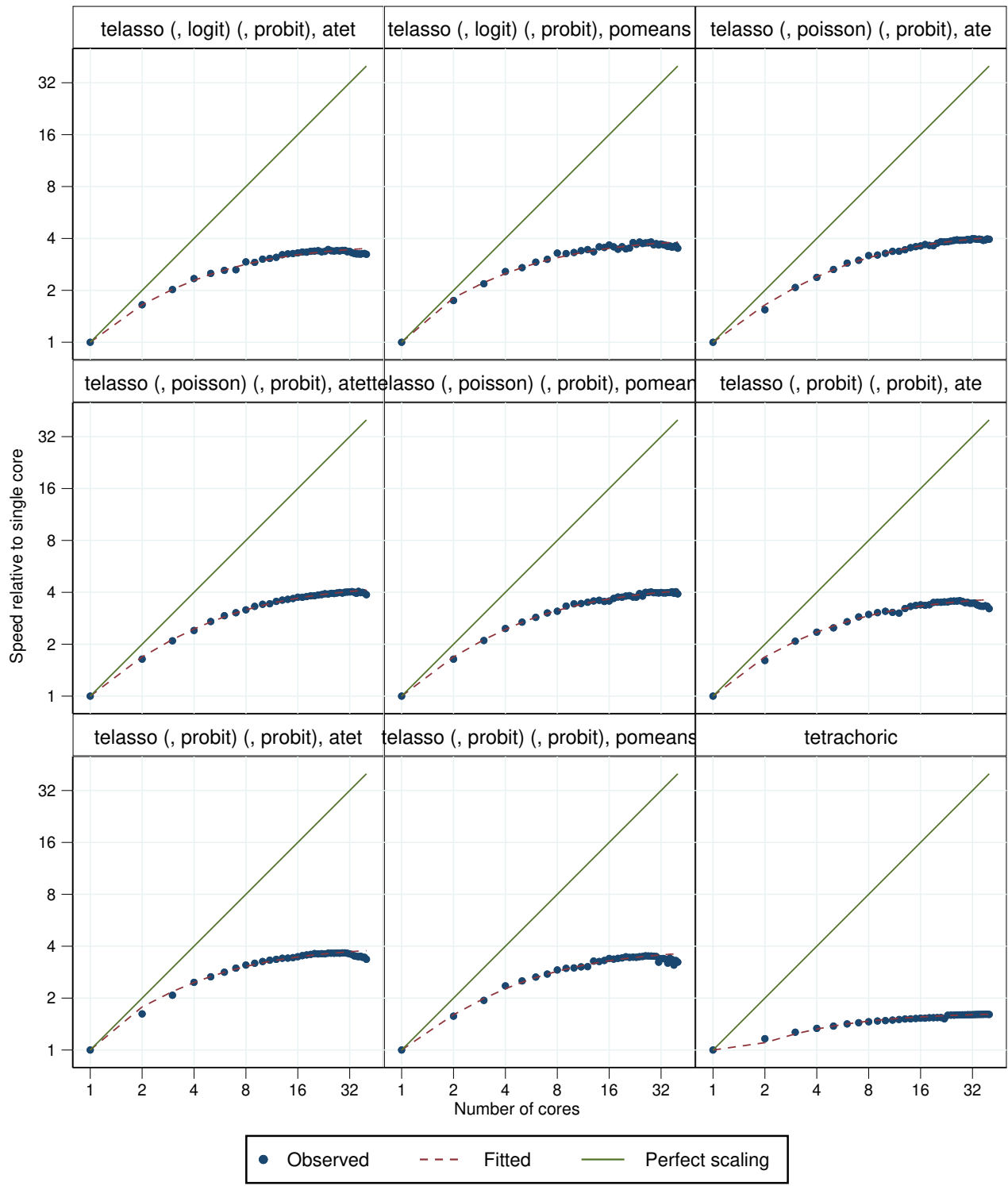


Figure 734. Parallelization performance plots.

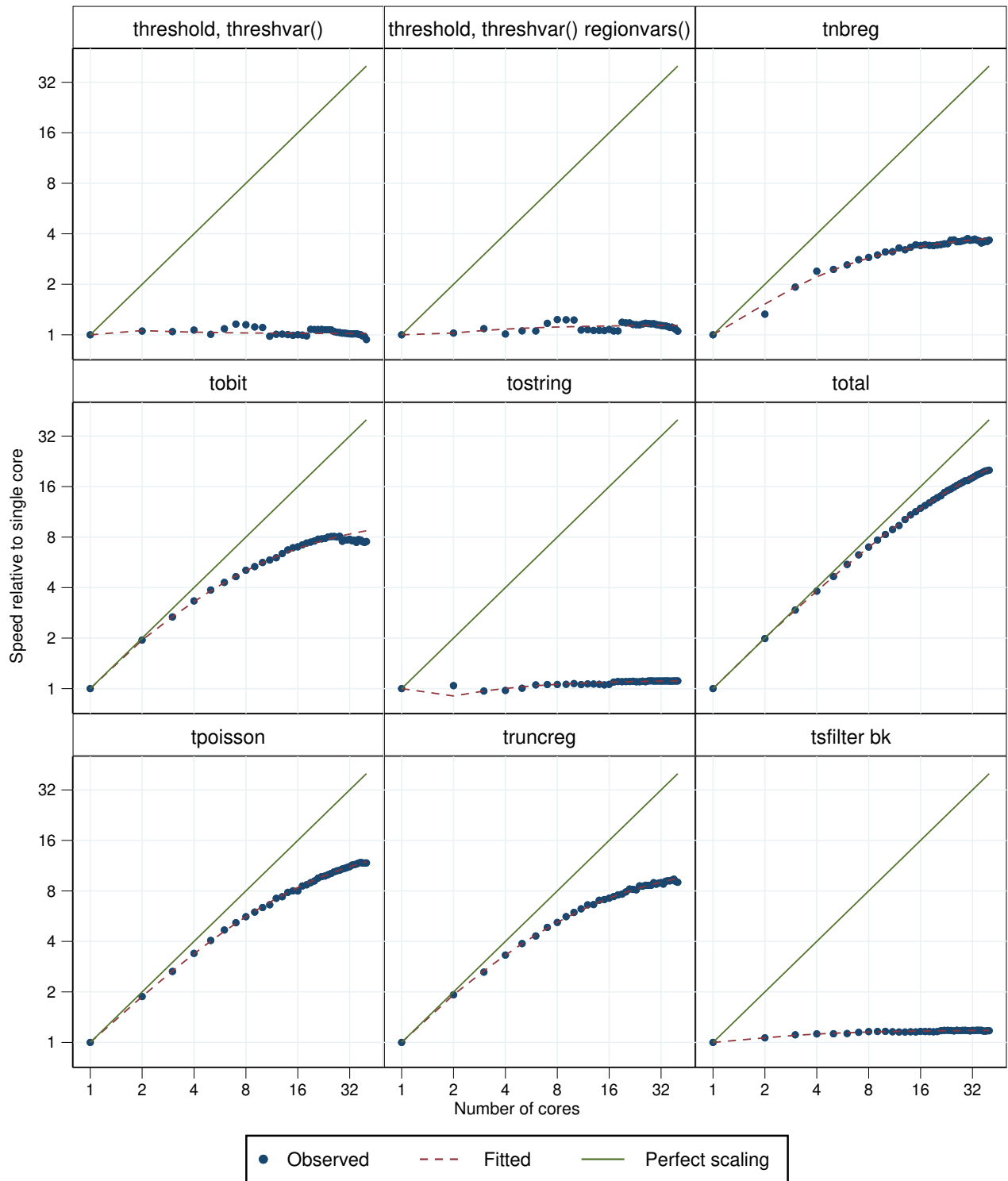


Figure 735. Parallelization performance plots.

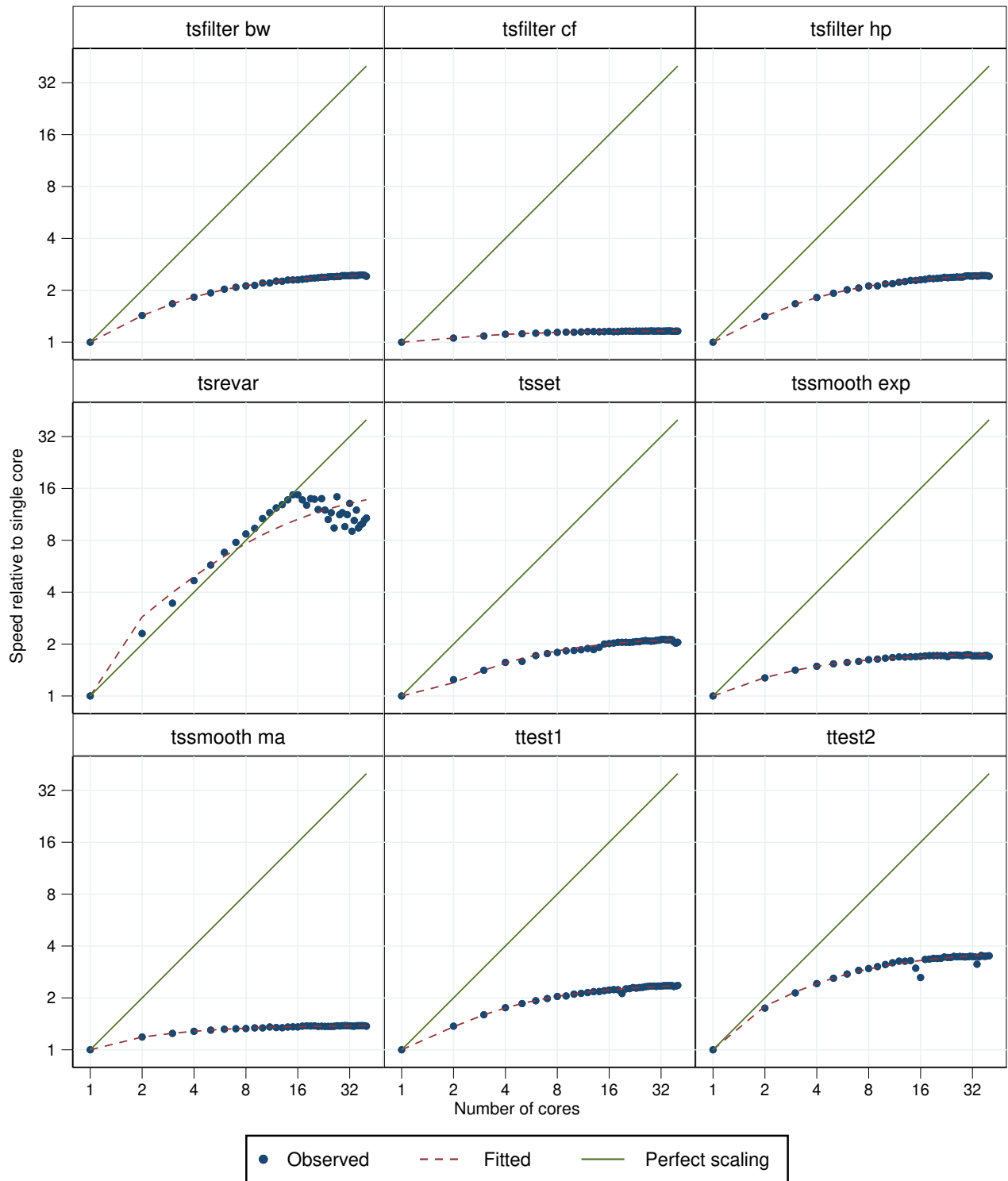


Figure 736. Parallelization performance plots.

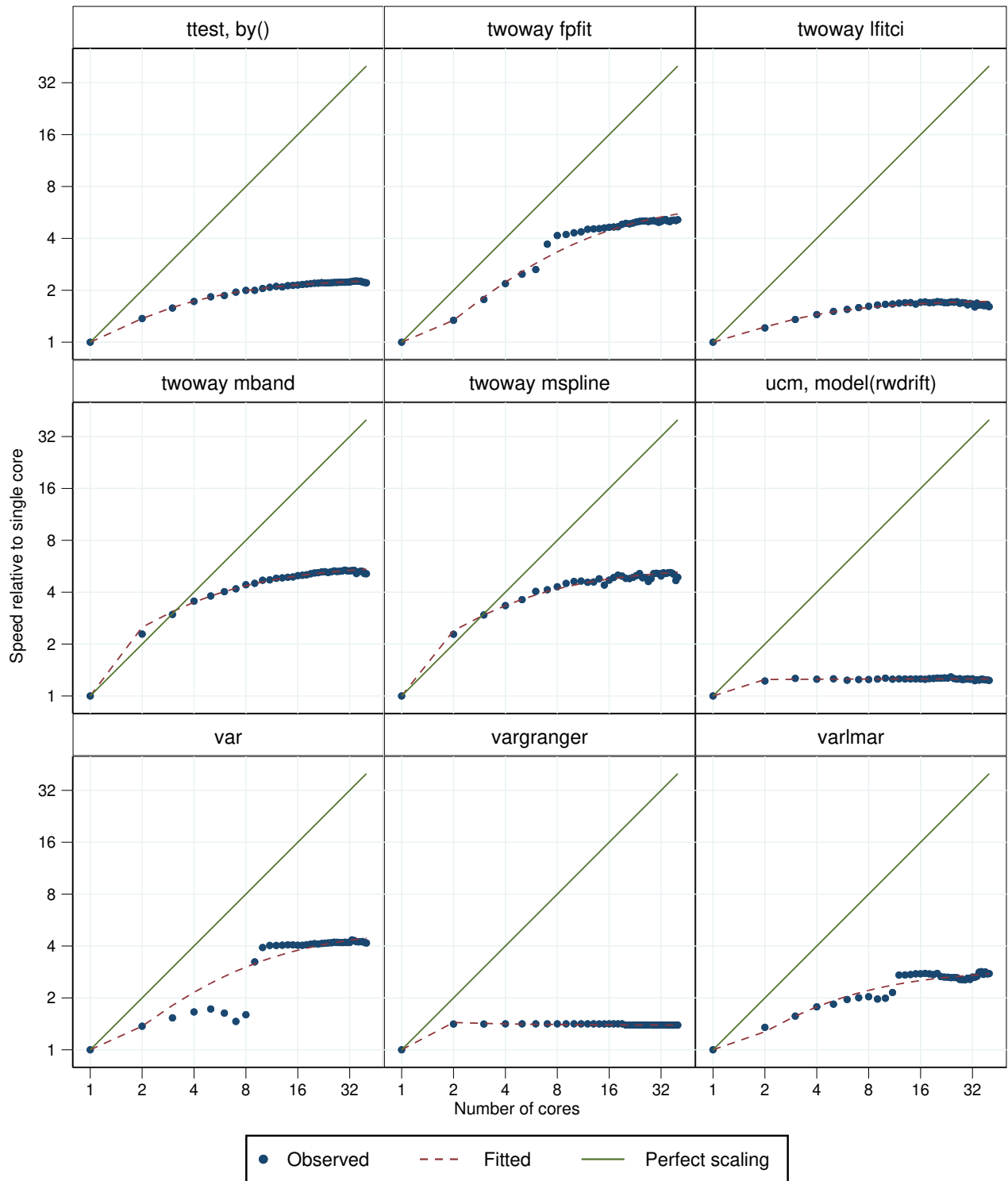


Figure 737. Parallelization performance plots.

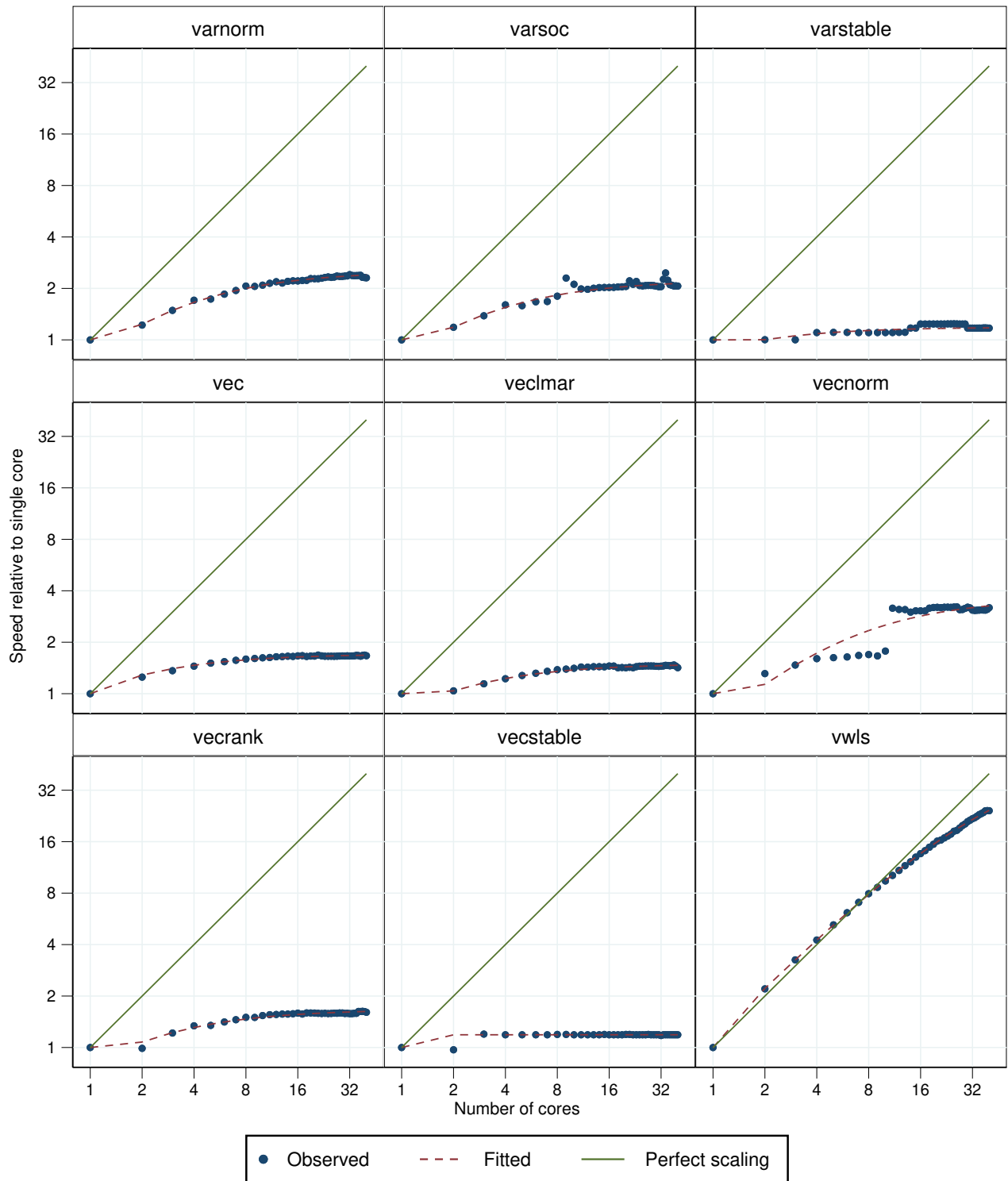


Figure 738. Parallelization performance plots.

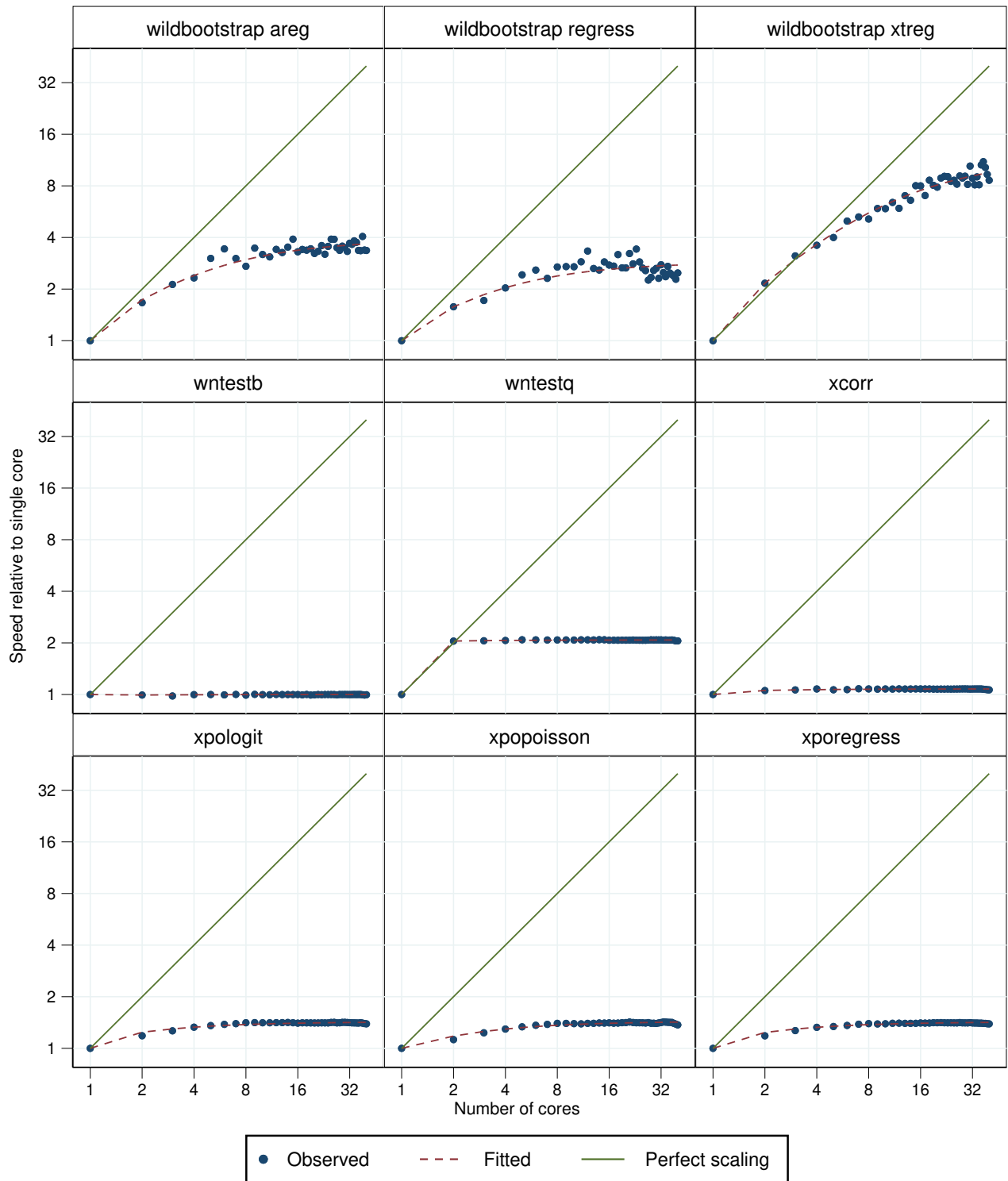


Figure 739. Parallelization performance plots.

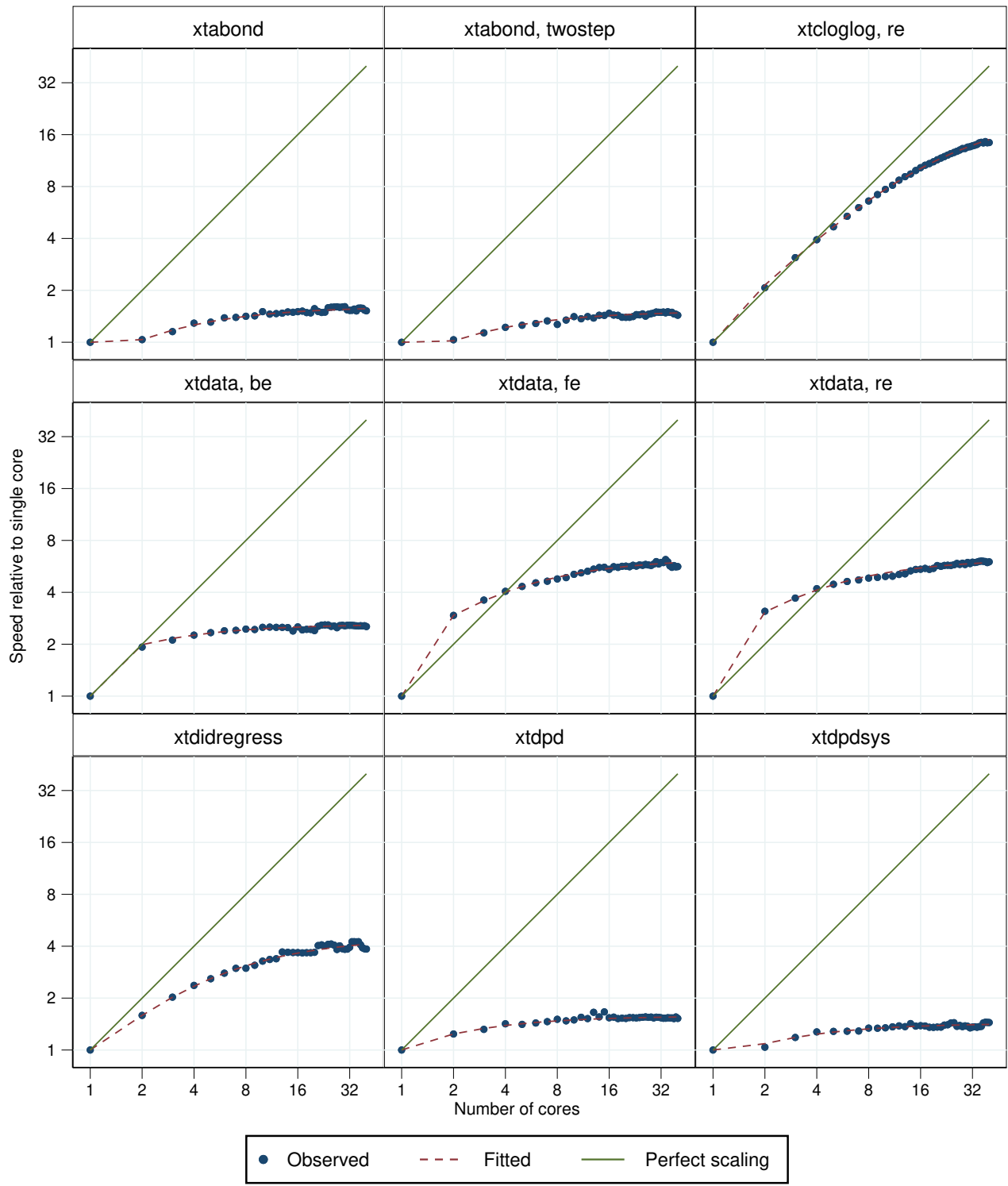


Figure 740. Parallelization performance plots.

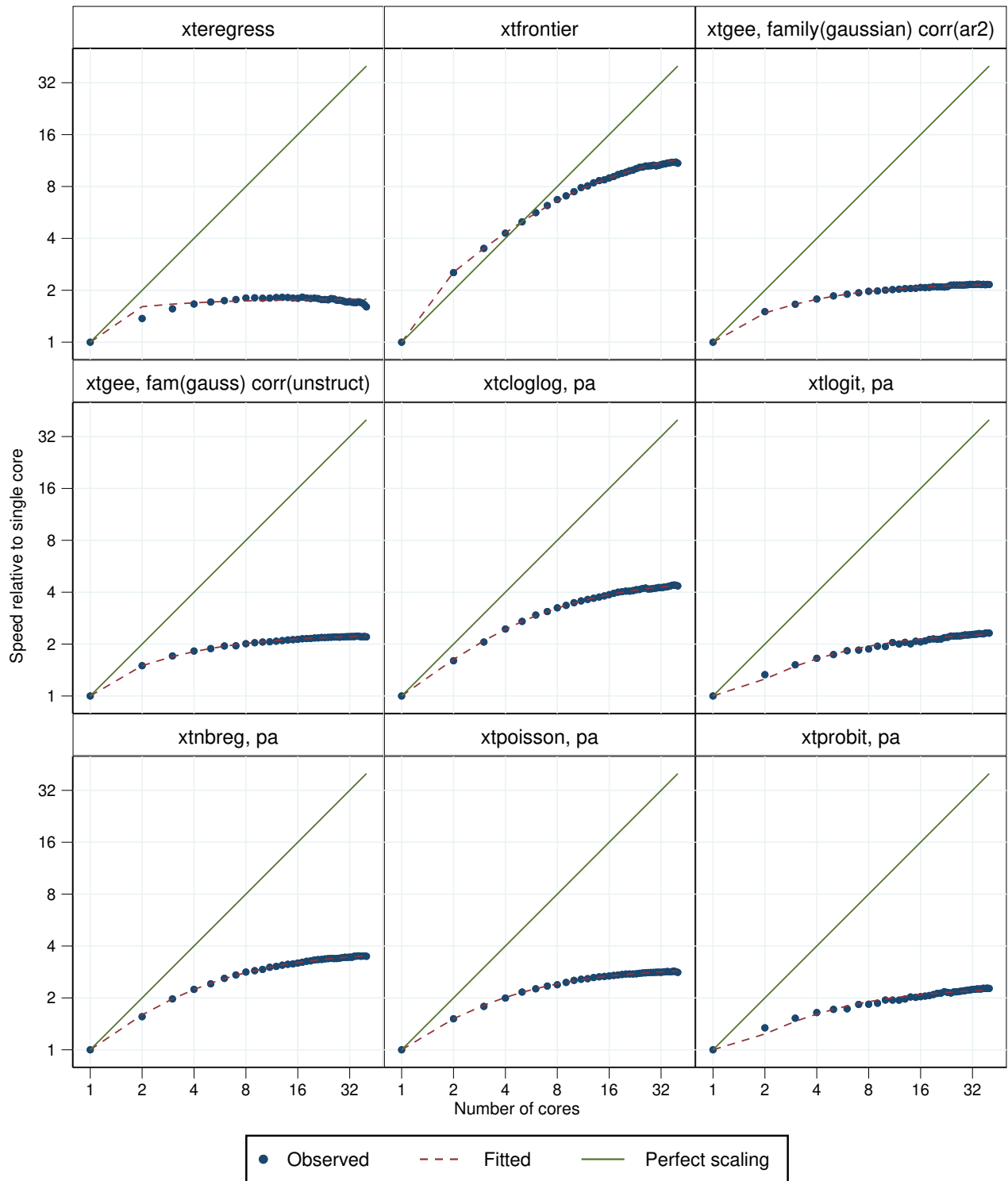


Figure 741. Parallelization performance plots.

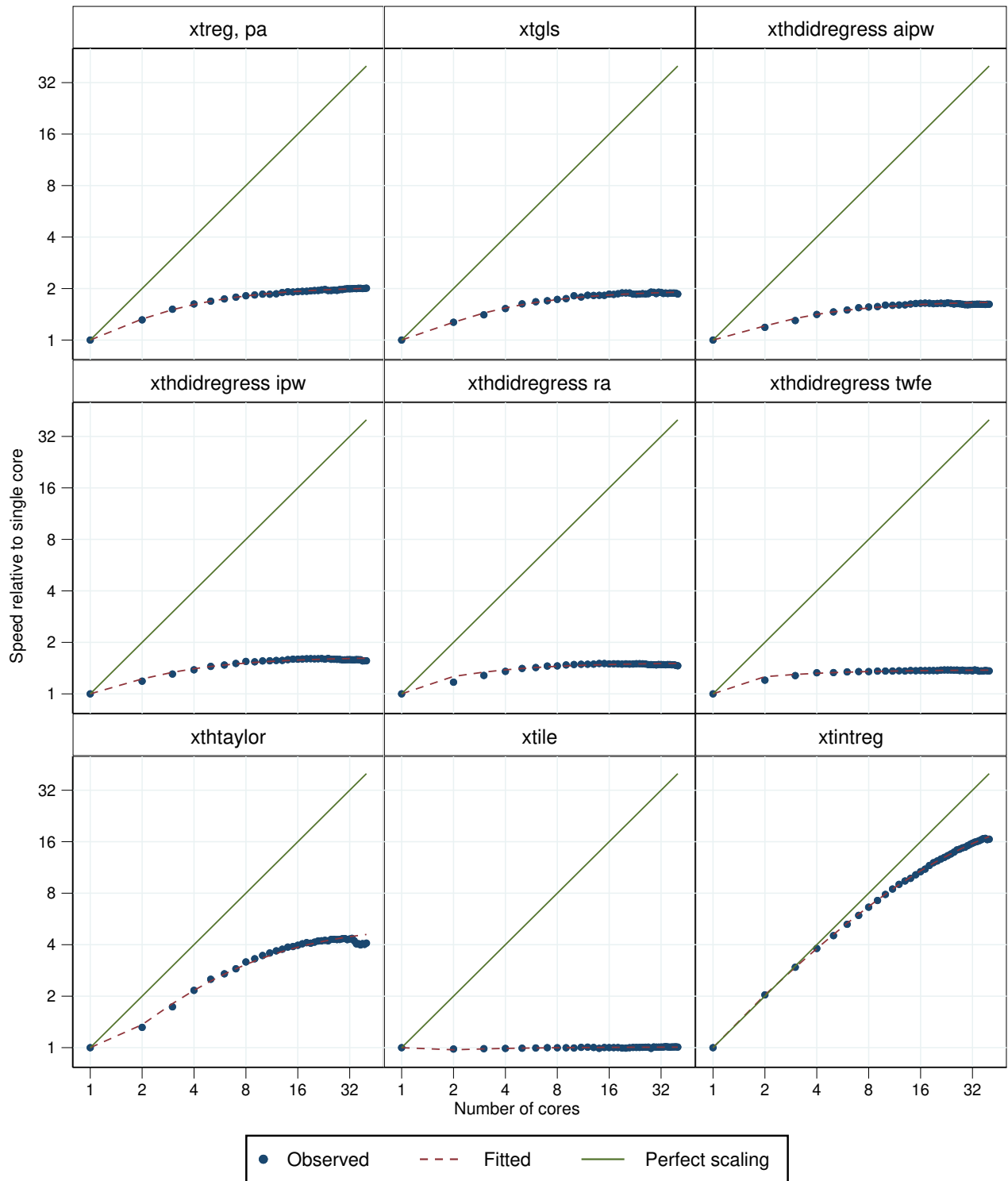


Figure 742. Parallelization performance plots.

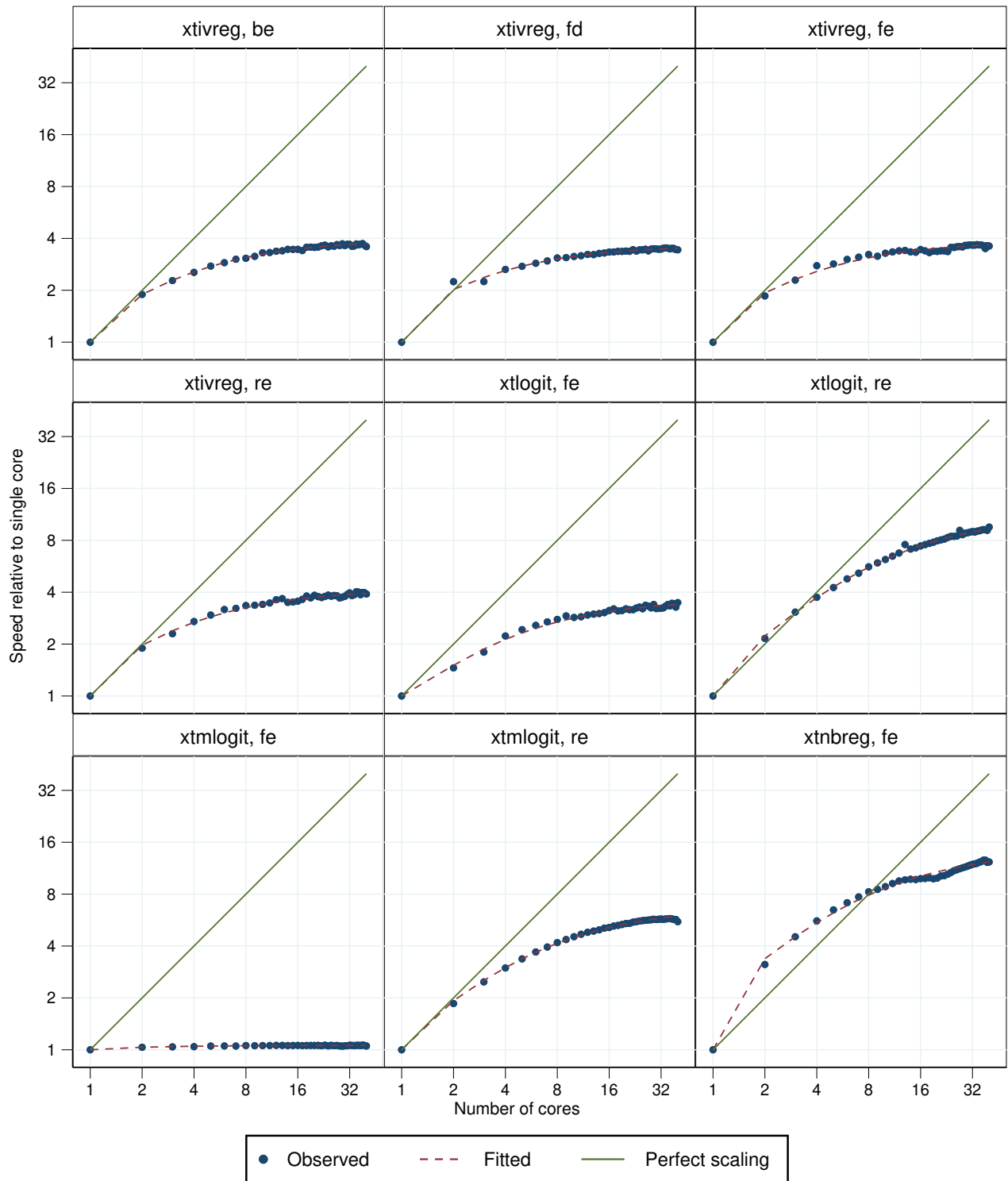


Figure 743. Parallelization performance plots.

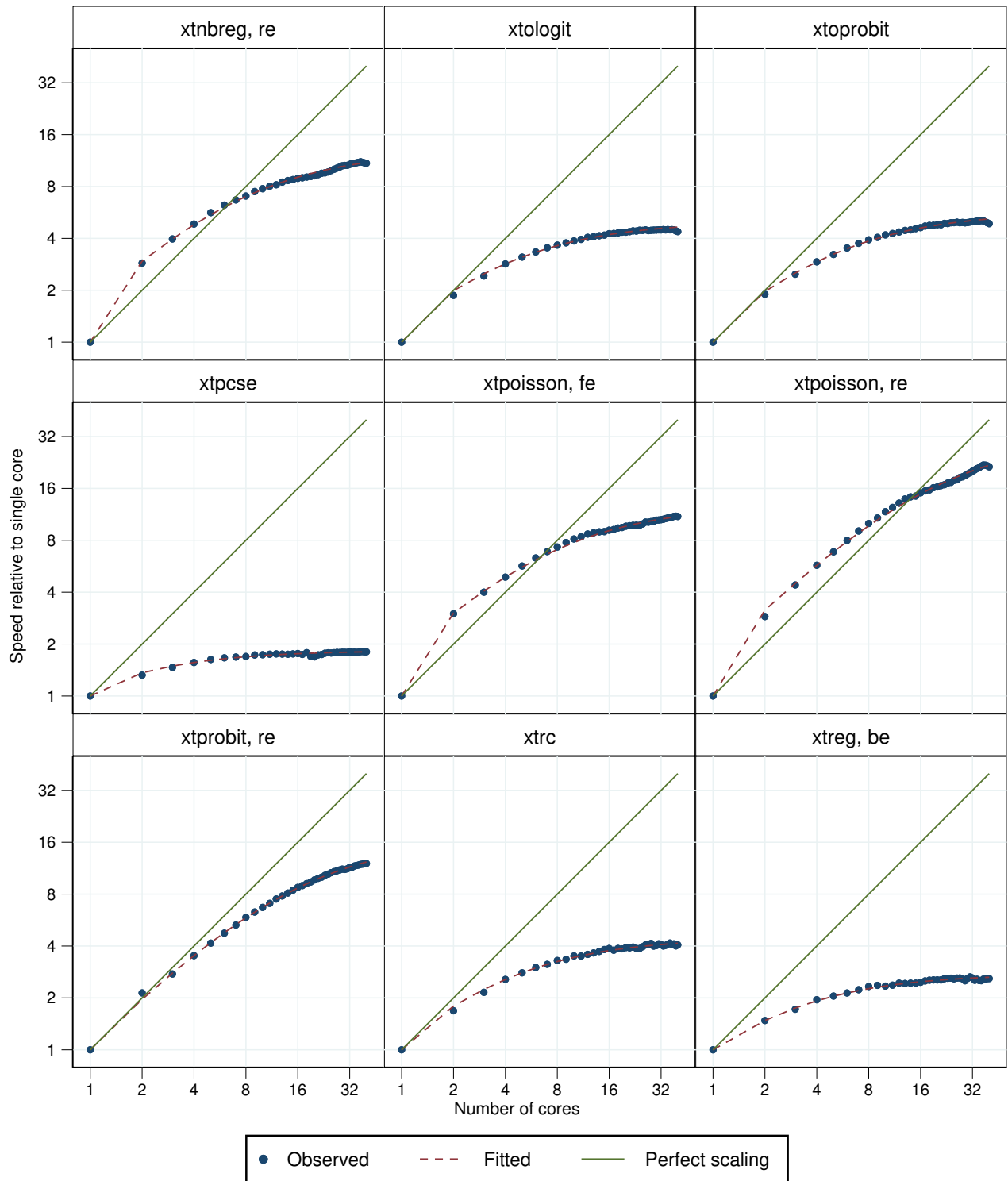


Figure 744. Parallelization performance plots.

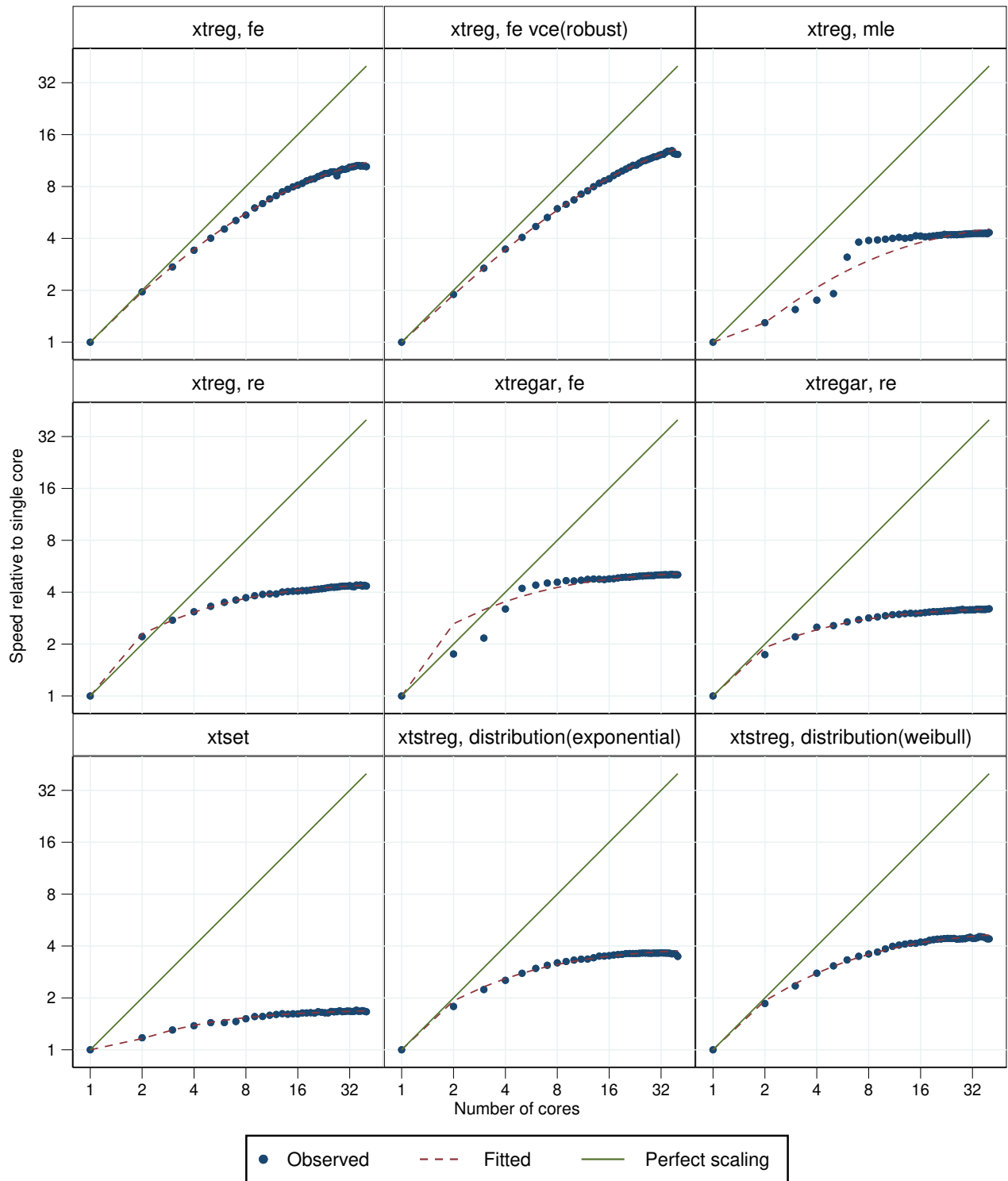


Figure 745. Parallelization performance plots.

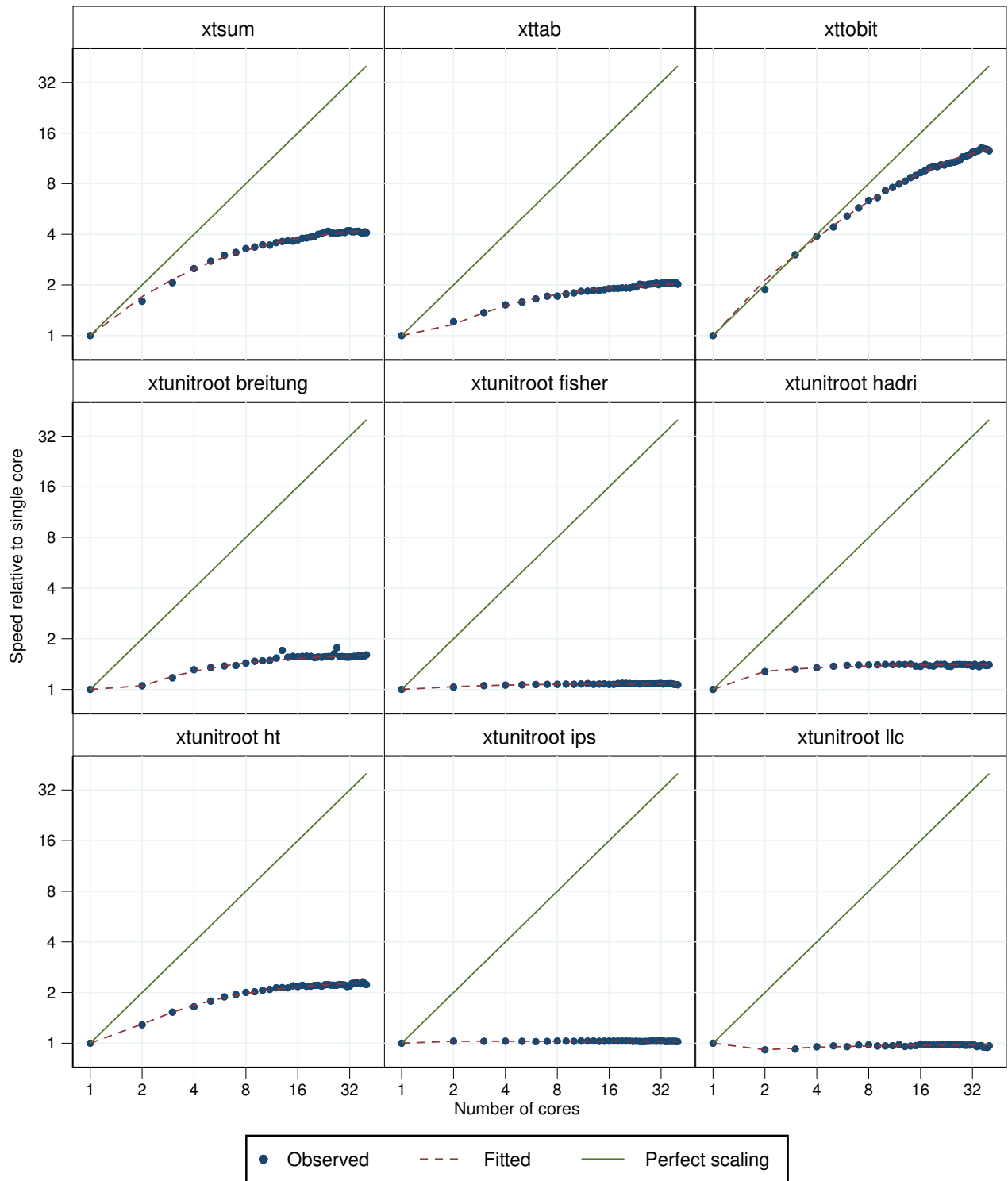


Figure 746. Parallelization performance plots.

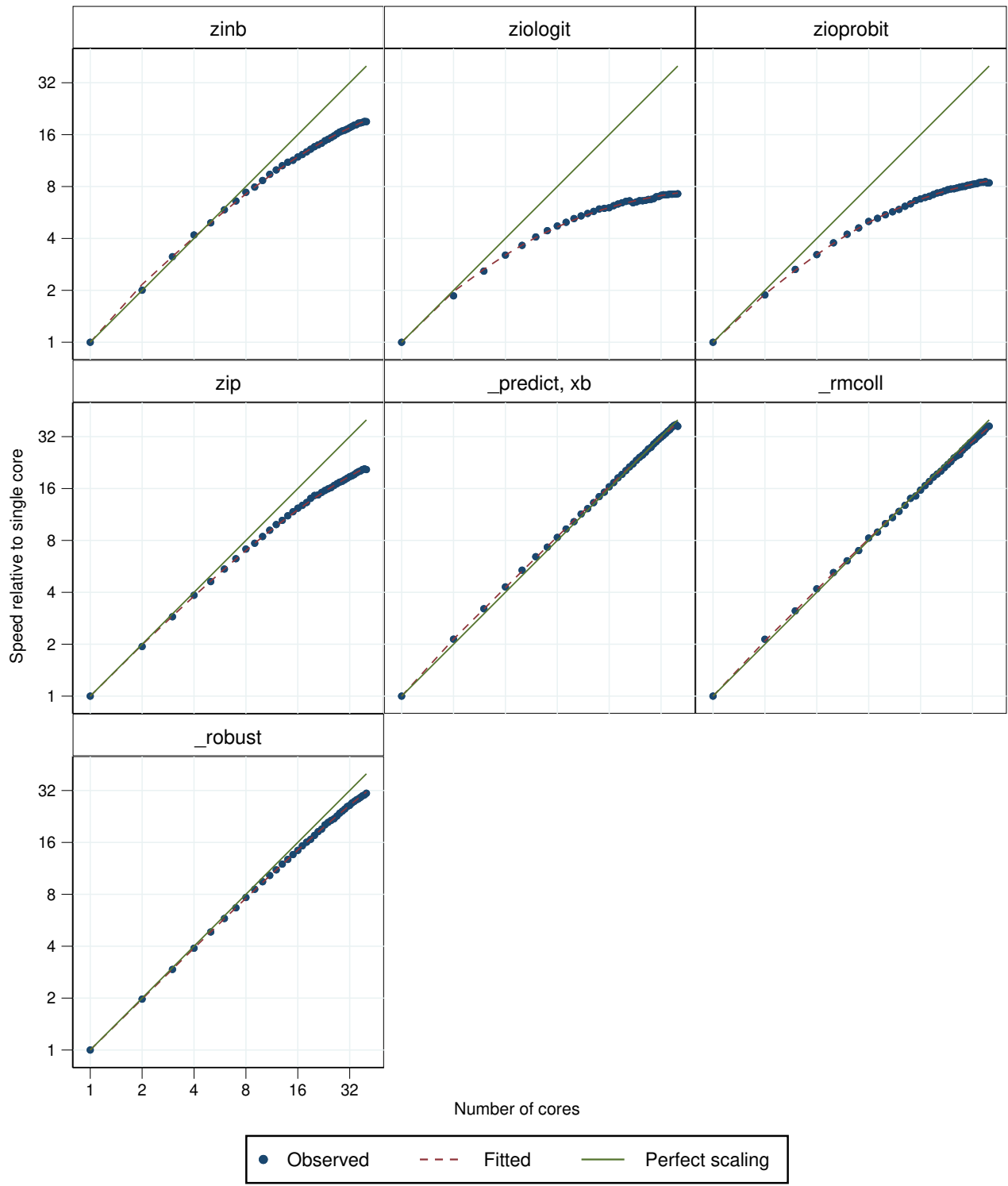


Figure 747. Parallelization performance plots.

C Command names and descriptions

Table 2. Command descriptions

Command	Description
<code>alpha</code>	Cronbach's alpha
<code>ameans</code>	Arithmetic, geometric, and harmonic means
<code>anova</code> (one-way)	Analysis of variance and covariance—one-way
<code>anova</code> (two-way)	Analysis of variance and covariance—two-way
<code>arch</code>	Autoregressive conditional heteroskedasticity (ARCH) family of estimators
<code>areg</code>	Linear regression with a large dummy-variable set
<code>areg, vce(cluster)</code>	Linear regression with a large dummy-variable set, cluster-robust standard errors
<code>areg, vce(robust)</code>	Linear regression with a large dummy-variable set, robust (Huber/White) standard errors
<code>arfima</code>	Autoregressive fractionally integrated moving-average models
<code>arima</code>	ARIMA, ARMAX, and other dynamic regression models
<code>bayes dsge</code>	Bayesian linear dynamic stochastic general equilibrium models
<code>bayes dsgenl</code>	Bayesian nonlinear dynamic stochastic general equilibrium models
<code>bayes: logit</code>	Bayesian logistic regression
<code>bayes: poisson</code>	Bayesian Poisson regression
<code>bayes: regress</code>	Bayesian linear regression
<code>bayes var</code>	Bayesian vector autoregressive models
<code>bayesmh logit</code>	Bayesian logistic regression using Metropolis-Hastings algorithm
<code>bayesmh mvn</code>	Bayesian multivariate normal regression using Metropolis-Hastings algorithm
<code>bayesmh mylogit</code>	Bayesian logistic regression using Metropolis-Hastings algorithm (custom evaluator)

Table 2. Command descriptions

Command	Description
<code>bayesmh nl</code>	Bayesian nonlinear regression using Metropolis-Hastings algorithm
<code>bayesmh normal</code>	Bayesian linear regression using Metropolis-Hastings algorithm
<code>bayesmh normal gibbs</code>	Bayesian linear regression using Gibbs sampling
<code>bayesmh normal re</code>	Bayesian linear regression with random effects using Metropolis-Hastings algorithm
<code>betareg, link(logit)</code>	Beta regression, logit link
<code>betareg, link(probit)</code>	Beta regression, probit link
<code>binreg</code>	Generalized linear models: extensions to the binomial family
<code>biplot</code>	Biplots
<code>biprobit</code>	Bivariate probit regression
<code>biprobit (seemingly unrelated)</code>	Seemingly unrelated probit regression
<code>bitest</code>	Binomial probability test
<code>blogit</code>	Logistic regression for grouped data
<code>bmaregress</code>	Bayesian model averaging for linear regression
<code>boxcox</code>	Box–Cox regression models
<code>bprobit</code>	Probit regression for grouped data
<code>brier</code>	Brier score decomposition
<code>bsample</code>	Sampling with replacement
<code>bstat</code>	Compute and report bootstrap statistics
<code>by: generate</code>	Create new variables over longitudinal/panel data

Table 2. Command descriptions

Command	Description
<code>by: generate</code> (small groups)	Create new variables over longitudinal/panel data, small panels
<code>by: replace</code>	Replace variable values over longitudinal/panel data
<code>by: replace</code> (small groups)	Replace variable values over longitudinal/panel data, small panels
<code>ca</code>	Simple correspondence analysis
<code>candisc</code>	Canonical linear discriminant analysis
<code>canon</code>	Canonical correlations
<code>cc</code>	Case-control odds ratio
<code>by: cc</code>	Case-control odds ratio over groups
<code>centile</code>	Report centile and confidence interval
<code>churdle linear</code>	Cragg hurdle regression
<code>ci means</code>	Confidence intervals for means, normal distribution
<code>ci means, poisson</code>	Confidence intervals for means, Poisson distribution
<code>ci proportions</code>	Confidence intervals for proportions
<code>clogit</code> (k1 to k2 matching)	Conditional (fixed-effects) logistic regression, k1 to k2 matching
<code>clogit</code> (1 to k matching)	Conditional (fixed-effects) logistic regression, 1 to k matching
<code>cloglog</code>	Complementary log-log regression
<code>cluster averagelinkage</code>	Hierarchical cluster analysis—average linkage
<code>cluster centroidlinkage</code>	Hierarchical cluster analysis—centroid linkage
<code>cluster completelinkage</code>	Hierarchical cluster analysis—complete linkage

Table 2. Command descriptions

Command	Description
<code>cluster generate</code>	Generate summary and grouping variables from a cluster analysis
<code>cluster kmeans</code>	Kmeans cluster analysis
<code>cluster kmedians</code>	Kmedians cluster analysis
<code>cluster medianlinkage</code>	Hierarchical cluster analysis—median linkage
<code>cluster singlelinkage</code>	Hierarchical cluster analysis—single linkage
<code>cluster wardslinkage</code>	Hierarchical cluster analysis—Ward’s linkage
<code>cluster waveragelinkage</code>	Hierarchical cluster analysis—Ward’s average linkage
<code>cmclogit</code>	Conditional logit (McFadden’s) choice model
<code>cmmprobit</code>	Multinomial probit choice model
<code>cmroprobit</code>	Rank-ordered probit choice model
<code>cnsreg</code>	Constrained linear regression
<code>codebook</code>	Describe data contents
<code>collapse</code>	Make dataset of summary datasets
<code>compare</code>	Compare two variables
<code>compress</code>	Compress data in memory
<code>contract</code>	Make dataset of frequencies and percentages
<code>corr2data</code>	Create dataset with specified correlation structure
<code>correlate</code>	Correlations (covariances) of variables or estimators
<code>corrgram</code>	Tabulate and graph autocorrelations

Table 2. Command descriptions

Command	Description
<code>count</code>	Count observations satisfying specified condition
<code>cpoisson</code>	Censored Poisson regression
<code>cs</code>	Cohort study risk-ratio
<code>by: cs</code>	Cohort study risk-ratio over groups
<code>ctset</code>	Declare data to be count-time data
<code>cttost</code>	Convert count-time data to survival-time data
<code>cumul</code>	Cumulative distribution
<code>cusum</code>	Cusum plots and tests for binary variables
<code>datasignature</code>	Determine whether data have changed
<code>decode</code>	Decode labeled numeric into string
<code>demandsys aids</code>	Almost ideal demand system estimation
<code>demandsys cdouglas</code>	Cobb-Douglas demand system estimation
<code>demandsys g aids</code>	Generalized almost ideal demand system estimation
<code>demandsys gqu aids</code>	Generalized quadratic almost ideal demand system estimation
<code>demandsys gtranslog</code>	Generalized translog demand system estimation
<code>demandsys les</code>	Linear expenditure demand system estimation
<code>demandsys qu aids</code>	Quadratic almost ideal demand system estimation
<code>demandsys translog</code>	Basic translog demand system estimation
<code>destring</code>	Convert string variables to numeric variables

Table 2. Command descriptions

Command	Description
<code>dfactor</code>	Dynamic-factor models
<code>dfgls</code>	DF-GLS unit-root test
<code>dfuller</code>	Augmented Dickey–Fuller unit-root test
<code>didregress</code>	Difference-in-differences estimation
<code>discrim knn</code>	Discriminant analysis— <i>k</i> th-nearest-neighbor
<code>discrim lda</code>	Discriminant analysis—linear
<code>discrim logistic</code>	Discriminant analysis—logistic
<code>discrim qda</code>	Discriminant analysis—quadratic
<code>dotplot</code>	Comparative scatterplots
<code>drawnorm</code>	Draw sample from multivariate normal distribution
<code>drop if <i>exp</i></code>	Eliminate observations using <code>if</code> expression
<code>drop in <i>range</i></code>	Eliminate observations using <code>in range</code>
<code>dsge</code>	Linearized dynamic stochastic general equilibrium model
<code>dsgenl</code>	Nonlinear dynamic stochastic general equilibrium model
<code>dslogit</code>	Double-selection lasso logistic regression
<code>dspoisson</code>	Double-selection lasso Poisson regression
<code>dsregress</code>	Double-selection lasso linear regression
<code>dstdize</code>	Direct and indirect standardization
<code>dvech</code>	Diagonal vech multivariate GARCH models

Table 2. Command descriptions

Command	Description
<code>egen group()</code>	Extensions to generate—create grouping variable
<code>by: egen mean</code>	Extensions to generate—create means over groups
<code>eivreg</code>	Errors-in-variables regression
<code>encode</code>	Encode string into numeric
<code>eregress</code>	Extended linear regression with endogenous covariates, treatment assignment, and sample selection
<code>esize twosample</code>	Effect size for two independent samples using groups
<code>esize unpaired</code>	Effect size for two independent samples using variables
<code>eteffects (exponential), ate</code>	Endogenous treatment-effects estimation, exponential-mean model, average treatment effect in population
<code>eteffects (linear), ate</code>	Endogenous treatment-effects estimation, linear model, average treatment effect in population
<code>eteffects (linear), pomeans</code>	Endogenous treatment-effects estimation, linear model, potential-outcome means
<code>eteffects (probit), ate</code>	Endogenous treatment-effects estimation, probit model, average treatment effect in population
<code>etpoisson</code>	Poisson regression with endogenous treatment effects
<code>etregress, poutcomes</code>	Linear regression with endogenous treatment effects, ML estimation with potential outcomes
<code>etregress, twostep</code>	Linear regression with endogenous treatment effects, two-step estimation
<code>exlogistic</code>	Exact logistic regression
<code>expand #</code>	Duplicate observations
<code>expand varname</code>	Duplicate observations using a variable
<code>expandcl #</code>	Duplicate clustered observations
<code>expandcl varname</code>	Duplicate clustered observations using a variable

Table 2. Command descriptions

Command	Description
<code>expoisson</code>	Exact Poisson regression
<code>factor</code>	Factor analysis
<code>fcast compute</code>	Dynamic forecasts after VAR or VEC estimation
<code>fillin</code>	Rectangularize dataset
<code>fmm 2: poisson</code>	Finite mixture model with two Poisson outcomes
<code>fmm 2: regress</code>	Finite mixture model with two linear outcomes
<code>fmm 3: poisson</code>	Finite mixture model with three Poisson outcomes
<code>fmm 3: regress</code>	Finite mixture model with three linear outcomes
<code>fracreg probit</code>	Fractional probit regression
<code>frontier</code>	Stochastic frontier models
<code>fvrevar (factors)</code>	Create indicators for factor variables
<code>fvrevar (interaction)</code>	Create indicators for factor variables—interactions
<code>generate (small expressions)</code>	Create or change contents of variable—small expressions
<code>generate</code>	Create or change contents of variable
<code>glm, family(gamma)</code>	Generalized linear models—gamma distribution
<code>glm, family(gaussian)</code>	Generalized linear models—Gaussian distribution
<code>glm, family(igaussian)</code>	Generalized linear models—inverse Gaussian distribution
<code>glm, family(nbinomial)</code>	Generalized linear models—negative binomial distribution
<code>glm, family(poisson)</code>	Generalized linear models—Poisson distribution

Table 2. Command descriptions

Command	Description
<code>glogit</code>	Weighted least-squares logistic regression for grouped data
<code>gmm</code>	Generalized method of moments estimation
<code>gmm (with derivatives)</code>	Generalized method of moments estimation with derivatives
<code>gprobit</code>	Weighted least-squares probit regression for grouped data
<code>graph bar</code>	Bar charts
<code>graph box</code>	Box plots
<code>graph pie</code>	Pie charts
<code>grmeanby</code>	Graph means and medians by categorical variables
<code>gsem, oprobit (CFA, 2-level)</code>	Ordered probit multilevel confirmatory factor analysis
<code>gsem, oprobit (CFA)</code>	Ordered probit confirmatory factor analysis
<code>gsem, logit group()</code>	GSEM: Logistic regression on 5 groups
<code>gsem, group()</code>	GSEM: Linear regression on 5 groups
<code>gsem, ologit group()</code>	GSEM: Ordinal logistic regression on 5 groups
<code>gsem, poisson group()</code>	GSEM: Poisson regression on 5 groups
<code>gsort</code>	Ascending and descending sort
<code>hausman</code>	Hausman specification test
<code>hdidregress aipw</code>	Heterogeneous difference in differences, augmented inverse-probability weighting
<code>hdidregress ipw</code>	Heterogeneous difference in differences, inverse-probability weighting
<code>hdidregress ra</code>	Heterogeneous difference in differences, regression adjustment

Table 2. Command descriptions

Command	Description
<code>hdidregress twfe</code>	Heterogeneous difference in differences, two-way fixed effects
<code>heckman</code>	Heckman selection model—maximum likelihood estimator
<code>heckman, twostep</code>	Heckman selection model—two-step estimator
<code>heckprobit</code>	Ordered probit model with sample selection
<code>heckpoisson</code>	Poisson regression with sample selection
<code>heckprob</code>	Probit model with selection
<code>hetprobit</code>	Heteroskedastic ordered probit regression
<code>hetprob</code>	Heteroskedastic probit model
<code>hetregress</code>	Heteroskedastic linear regression, ML estimation
<code>hetregress, twostep</code>	Heteroskedastic linear regression, two-step estimation
<code>histogram</code>	Histograms for continuous and categorical variables
<code>hotelling</code>	Hotelling's T -squared generalized means test
<code>icc, mixed</code>	Intraclass correlations for two-way mixed-effects model
<code>icc (one-way)</code>	Intraclass correlations for one-way random-effects model
<code>icc (two-way)</code>	Intraclass correlations for two-way random-effects model
<code>import delimited</code>	Import delimited text data
<code>intreg</code>	Interval regression
<code>ir</code>	Incidence-rate ratio
<code>by: ir</code>	Incidence-rate ratio over groups

Table 2. Command descriptions

Command	Description
<code>irf create</code>	Create IRFs and FEVDs after VAR and VEC estimation
<code>irt 1pl</code>	Item response theory one-parameter logistic model
<code>irt 2pl</code>	Item response theory two-parameter logistic model
<code>irt 3pl</code>	Item response theory three-parameter logistic model
<code>irt grm</code>	Item response theory graded response model
<code>irt nrm</code>	Item response theory nominal response model
<code>irt pcm</code>	Item response theory partial credit model
<code>irt rsm</code>	Item response theory rating scale model
<code>istdize</code>	Indirect standardization
<code>ivfprobit</code>	Fractional probit model with continuous endogenous covariates
<code>ivpoisson cfunction</code>	Poisson regression with endogenous regressors, control-function estimator
<code>ivpoisson gmm, additive</code>	Poisson regression with endogenous regressors, GMM with additive regression errors
<code>ivpoisson gmm, multiplicative</code>	Poisson regression with endogenous regressors, GMM multiplicative regression errors
<code>ivprobit</code>	Probit model with endogenous regressors
<code>ivqregress iqr</code>	Instrumental-variables quantile regression, inverse quantile estimator
<code>ivregress 2sls</code>	Instrumental-variables regression—two-stage least squares
<code>ivregress gmm</code>	Instrumental-variables regression—GMM
<code>ivregress liml</code>	Instrumental-variables regression—LIML
<code>ivtobit</code>	Tobit model with endogenous regressors

Table 2. Command descriptions

Command	Description
<code>kap</code>	Interrater agreement
<code>kappa</code>	Interrater agreement
<code>kdensity</code>	Univariate kernel density estimation
<code>keep if <i>exp</i></code>	Retain observations using <code>if</code> expression
<code>keep in <i>range</i></code>	Retain observations using <code>in range</code>
<code>keep <i>varlist</i></code>	Retain variables
<code>ksmirnov</code>	Kolmogorov–Smirnov equality-of-distributions test
<code>ksmirnov, by()</code>	Kolmogorov–Smirnov equality-of-distributions test over groups
<code>ktau</code>	Kendall’s rank correlation coefficients
<code>kwallis</code>	Kruskal–Wallis equality-of-populations rank test
<code>ladder</code>	Ladder of powers
<code>lasso linear</code>	Linear lasso for prediction and model selection
<code>lasso logit</code>	Logistic lasso for prediction and model selection
<code>lasso poisson</code>	Poisson lasso for prediction and model selection
<code>gsem, lclass(C 2)</code>	Latent Class Analysis, logit outcomes, 2 classes
<code>gsem, lclass(C 3)</code>	Latent Class Analysis, logit outcomes, 3 classes
<code>levelsof</code>	Levels of variable
<code>loadingplot</code>	Score and loading plots after <code>factor</code> and <code>pca</code>
<code>logistic</code>	Logistic regression, reporting odds ratios

Table 2. Command descriptions

Command	Description
<code>logit</code>	Logistic regression, reporting coefficients
<code>loneqway</code>	Large one-way ANOVA, random effects, and reliability
<code>lowess</code>	Lowess smoothing
<code>lpoly</code>	Kernel-weighted local polynomial smoothing
<code>ltable</code>	Life tables for survival data
<code>makespline bspline</code>	B-spline generation
<code>makespline linear</code>	Linear spline generation
<code>makespline piecewise</code>	Piecewise polynomial spline generation
<code>makespline rcs</code>	Restricted cubic spline generation
<code>manova (one-way)</code>	Multivariate analysis of variance and covariance, one-way
<code>manova (two-way)</code>	Multivariate analysis of variance and covariance, two-way
<code>margins</code>	Marginal means and predictive margins
<code>margins, dydx() exp()</code>	Marginal effects of an expression
<code>margins, dydx()</code>	Marginal effects
<code>margins, exp()</code>	Predictive margins of an expression
<code>markout</code>	Mark observations for exclusion
<code>marksample</code>	Mark observations for inclusion
<code>marksample if exp</code>	Mark observations for inclusion, with <code>if</code> expression
<code>matrix accum</code>	Form cross-product matrices of variables over observations

Table 2. Command descriptions

Command	Description
<code>matrix eigenvalues</code>	Eigenvalues of a matrix
<code>matrix score</code>	Inner product of matrix with variables over observations
<code>matrix svd</code>	Singular value decomposition
<code>matrix symeigen</code>	Eigenvalues of a symmetric matrix
<code>matrix syminv</code>	Inversion of a symmetric matrix
<code>mca</code>	Multiple and joint correspondence analysis
<code>mcc</code>	Matched case-control studies
<code>mds</code>	Multidimensional scaling for two-way data
<code>mdslong</code>	Multidimensional scaling of proximity data in long format
<code>mean</code>	Estimate means
<code>mecloglog</code>	Multilevel mixed-effects complimentary log-log regression
<code>median</code>	Equality tests on unmatched data
<code>mediate (, expmean) (, linear)</code>	Causal mediation analysis, exponential-mean model
<code>mediate (, linear) (, linear)</code>	Causal mediation analysis, linear model
<code>mediate (, logit) (, linear)</code>	Causal mediation analysis, logistic model
<code>mediate (, poisson) (, linear)</code>	Causal mediation analysis, Poisson model
<code>mediate (, probit) (, linear)</code>	Causal mediation analysis, probit model
<code>meintreg</code>	Multilevel mixed-effects interval regression
<code>melogit</code>	Multilevel mixed-effects logistic regression

Table 2. Command descriptions

Command	Description
<code>menbreg,</code> <code>dispersion(constant)</code>	Multilevel mixed-effects negative binomial regression, constant dispersion
<code>menbreg, dispersion(mean)</code>	Multilevel mixed-effects negative binomial regression, mean dispersion
<code>menl</code>	Nonlinear mixed-effects regression for a linear outcome
<code>meologit</code>	Multilevel mixed-effects ordered logistic regression
<code>meoprobit</code>	Multilevel mixed-effects ordered probit regression
<code>mepoisson</code>	Multilevel mixed-effects Poisson regression
<code>meprobit</code>	Multilevel mixed-effects probit regression
<code>mestreg,</code> <code>distribution(exp)</code>	Multilevel mixed-effects survival models, exponential distribution
<code>mestreg,</code> <code>distribution(weibull)</code>	Multilevel mixed-effects survival models, Weibull distribution
<code>metobit</code>	Multilevel mixed-effects Tobit regression
<code>mgarch</code>	Multivariate generalized autoregressive conditional-heteroskedasticity (MGARCH) models
<code>mhodds</code>	Ratio of odds of failure for two categories
<code>mhodds (adjusted)</code>	Ratio of odds of failure for two categories adjusting for levels
<code>by: mhodds</code>	Ratio of odds of failure for two categories over groups
<code>mhodds (trend)</code>	Ratio of odds of failure testing for trend
<code>mi estimate: logit</code> <code>(flong)</code>	Logistic regression with multiply imputed data— <code>flong</code> style data
<code>mi estimate: logit</code> <code>(flongsep)</code>	Logistic regression with multiply imputed data— <code>flongsep</code> style data
<code>mi estimate: logit</code> <code>(mlong)</code>	Logistic regression with multiply imputed data— <code>mlong</code> style data
<code>mi estimate: logit (wide)</code>	Logistic regression with multiply imputed data— <code>wide</code> style data

Table 2. Command descriptions

Command	Description
<code>mi estimate: mlogit</code>	Multinomial logistic regression with multiply imputed data
<code>mi estimate: ologit</code>	Ordered logistic regression with multiply imputed data
<code>mi estimate: regress (flong)</code>	Linear regression with multiply imputed data— <code>flong</code> style data
<code>mi estimate: regress (flongsep)</code>	Linear regression with multiply imputed data— <code>flongsep</code> style data
<code>mi estimate: regress (mlong)</code>	Linear regression with multiply imputed data— <code>mlong</code> style data
<code>mi estimate: regress (wide)</code>	Linear regression with multiply imputed data— <code>wide</code> style data
<code>mi impute chained (flong)</code>	Impute missing values using chained equations— <code>flong</code> style data
<code>mi impute chained (flongsep)</code>	Impute missing values using chained equations— <code>flongsep</code> style data
<code>mi impute chained (mlong)</code>	Impute missing values using chained equations— <code>mlong</code> style data
<code>mi impute chained (wide)</code>	Impute missing values using chained equations— <code>wide</code> style data
<code>mi impute logit (flong)</code>	Impute missing values using logistic regression— <code>flong</code> style data
<code>mi impute logit (flongsep)</code>	Impute missing values using logistic regression— <code>flongsep</code> style data
<code>mi impute logit (mlong)</code>	Impute missing values using logistic regression— <code>mlong</code> style data
<code>mi impute logit (wide)</code>	Impute missing values using logistic regression— <code>wide</code> style data
<code>mi impute mlogit</code>	Impute missing values using multinomial logistic regression
<code>mi impute mono pmm</code>	Impute missing values using monotone predictive mean matching
<code>mi impute mono regress</code>	Impute missing values using monotone linear regression
<code>mi impute mvn</code>	Impute missing values using multivariate normal
<code>mi impute ologit</code>	Impute missing values using ordinal logistic regression

Table 2. Command descriptions

Command	Description
<code>mi impute pmm</code>	Impute missing values using predictive mean matching
<code>mi impute regress</code>	
<code>misstable nested</code>	Analyze missing values—list the nesting rules
<code>misstable patterns</code>	Analyze missing values—report patterns
<code>misstable summarize</code>	Analyze missing values—report counts
<code>misstable tree</code>	Analyze missing values—present tree view
<code>mixed</code>	Multilevel mixed-effects linear regression
<code>mixed (crossed effects)</code>	Multilevel mixed-effects linear regression—crossed effects
<code>mkspline</code>	Linear spline construction
<code>mlevel</code>	Helper command for user-programmed MLEs: Evaluate likelihood of coefficient vector
<code>mlevel, nocons</code>	Helper command for user-programmed MLEs: Evaluate likelihood of coefficient vector without constant
<code>mlmatbysum</code>	Helper command for user-programmed MLEs: Compute Hessians of panel-data estimators
<code>mlmatsum</code>	Helper command for user-programmed MLEs: Compute Hessians of coefficient vector
<code>mlogit</code>	Multinomial (polytomous) logistic regression
<code>mlsum</code>	Helper command for user-programmed MLEs: Sum likelihood of coefficient vector
<code>mlvecsum</code>	Helper command for user-programmed MLEs: Compute gradients of coefficient vector
<code>mprobit</code>	Multinomial probit regression
<code>mswitch ar</code>	Markov-switching regression models, autoregression
<code>mswitch dr</code>	Markov-switching regression models, dynamic regression

Table 2. Command descriptions

Command	Description
<code>mvdecode</code>	Recode numeric values to missing
<code>mvencode</code>	Recode missing values to numeric
<code>mvreg</code>	Multivariate regression
<code>mvtest correlations</code>	Multivariate test—correlations
<code>mvtest covariances</code>	Multivariate test—covariances
<code>mvtest means,</code> <code>heterogeneous</code>	Multivariate test—means, heterogenous covariances
<code>mvtest means, homogeneous</code>	Multivariate test—means, homogeneous covariances
<code>mvtest means, lr</code>	Multivariate test—means, likelihood-ratio test
<code>mvtest normality</code>	Multivariate test—normality
<code>nbreg</code>	Negative binomial regression
<code>newey</code>	Regression with Newey–West standard errors
<code>nl</code>	Nonlinear least-squares estimation
<code>nlogit</code>	Nested logit regression
<code>nlstur</code>	Estimation of nonlinear systems of equations
<code>npregress kernel</code>	Nonparametric kernel regression
<code>nptrend, cuzick</code>	Test for trend across ordered groups, Cusick
<code>nptrend, carmitage</code>	Test for trend across ordered groups, Cochran-Armitage
<code>nptrend, jterpstra</code>	Test for trend across ordered groups, Jonckheere-Terpstra
<code>nptrend, linear</code>	Test for trend across ordered groups, linear-by-linear

Table 2. Command descriptions

Command	Description
<code>ologit</code>	Ordered logistic regression
<code>ologit, vce(cluster)</code>	Ordered logistic regression, cluster-robust standard errors
<code>ologit, vce(robust)</code>	Ordered logistic regression, robust (Huber/White) standard errors
<code>oneway</code>	One-way analysis of variance
<code>oprobit</code>	Ordered probit regression
<code>oprobit, vce(cluster)</code>	Ordered probit regression, cluster-robust standard errors
<code>oprobit, vce(robust)</code>	Ordered probit regression, robust (Huber/White) standard errors
<code>orthog</code>	Orthogonalize variables and compute orthogonal polynomials
<code>pca</code>	Principal component analysis
<code>pcorr</code>	Partial correlation coefficients
<code>pctile</code>	Create variable containing percentiles
<code>pergram</code>	Periodogram
<code>pkcollapse</code>	Generate pharmacokinetic measurement dataset
<code>pkexamine</code>	Calculate pharmacokinetic measures
<code>pksumm</code>	Summarize pharmacokinetic data
<code>poisson</code>	Poisson regression
<code>poisson, vce(cluster)</code>	Poisson regression, cluster-robust standard errors
<code>poisson, exposure()</code>	Poisson regression, with exposure
<code>poisson, vce(robust)</code>	Poisson regression, robust (Huber/White) standard errors

Table 2. Command descriptions

Command	Description
<code>pologit</code>	Partialling-out lasso logistic regression
<code>popoisson</code>	Partialling-out lasso Poisson regression
<code>poregress</code>	Partialling-out lasso linear regression
<code>pperron</code>	Phillips–Perron unit-root test
<code>prais</code>	Prais–Winsten and Cochrane–Orcutt regression
<code>predict, cooks</code>	Obtain Cook’s distance predictions after estimation
<code>predict, covratio</code>	Obtain COVRATIO predictions after estimation
<code>predict, dfbeta</code>	Obtain DFBETAs for a variable after estimation
<code>predict, dfits</code>	Obtain DFITS predictions after estimation
<code>predict, e</code>	Obtain predictions given upper and lower truncation after estimation
<code>predict, leverage</code>	Obtain leverage of observations after estimation
<code>predict, pr</code>	Obtain probability-in-range predictions after estimation
<code>predict, residuals</code>	Obtain residuals after estimation
<code>predict, rstandard</code>	Obtain standardized residuals after estimation
<code>predict, rstudent</code>	Obtain Studentized residuals after estimation
<code>predict, stdf</code>	Obtain standard errors of predictions after estimation
<code>predict, stdp</code>	Obtain standard errors of forecasts after estimation
<code>predict, stdr</code>	Obtain standard errors of residuals after estimation
<code>predict, welsch</code>	Obtain Welsch distances after estimation

Table 2. Command descriptions

Command	Description
<code>predict, ystar</code>	Obtain truncated predictions in a range after estimation
<code>predictnl</code>	Obtain nonlinear predictions, standard errors, etc., after estimation
<code>probit</code>	Probit regression
<code>procrustes</code>	Procrustes transformation
<code>proportion</code>	Estimate proportions
<code>prtest1</code>	One-sample tests of proportions
<code>prtest2</code>	Two-sample tests of proportions
<code>prtest, by()</code>	Tests of proportions computed over groups
<code>pwcorr</code>	Pairwise correlation coefficients
<code>qreg</code>	Quantile (including median) regression
<code>ranksum</code>	Equality tests on unmatched data
<code>ratio</code>	Estimate ratio with SE and CI
<code>ratio (exp1) (exp2)</code>	Estimate two ratios with SE and CI
<code>recode</code>	Recode categorical variables
<code>reg3</code>	Three-stage estimation for systems of simultaneous equations
<code>regress</code>	Linear regression
<code>regress, vce(cluster)</code>	Linear regression, cluster-robust standard errors
<code>regress, vce(robust)</code>	Linear regression, robust (Huber/White) standard errors
<code>replace</code>	Create or change contents of variable

Table 2. Command descriptions

Command	Description
<code>replace</code> (small expressions)	Create or change contents of variable, simple expression
<code>rerilogistic</code>	Relative excess risk due to interaction, logistic regression
<code>rerinbreg</code>	Relative excess risk due to interaction, negative binomial regression
<code>reripoisson</code>	Relative excess risk due to interaction, Poisson regression
<code>reristcox</code>	Relative excess risk due to interaction, Cox proportional hazards model
<code>reristintcox, interval()</code>	Relative excess risk due to interaction, interval-censored Cox proportional hazards model
<code>reristintcox, id() time() status()</code>	Relative excess risk due to interaction, multiple-record-per-subject interval-censored Cox proportional hazards model
<code>reristintreg</code>	Relative excess risk due to interaction, interval-censored parametric survival-time model
<code>reristreg</code>	Relative excess risk due to interaction, parametric survival model
<code>reshape long</code>	Convert data from wide to long
<code>reshape wide</code>	Convert data from long to wide
<code>robvar</code>	Robust tests for equality of variance
<code>rocfit</code>	Fit ROC models
<code>roctab</code>	Receiver operating characteristic (ROC) analysis
<code>rotate</code>	Orthogonal and oblique rotations after <code>factor</code> and <code>pca</code>
<code>rotatemat</code>	Orthogonal and oblique rotations of a Stata matrix
<code>rreg</code>	Robust regression
<code>runtest</code>	Test for random order
<code>scobit</code>	Skewed logistic regression

Table 2. Command descriptions

Command	Description
<code>scoreplot</code>	Score and loading plots after <code>factor</code> and <code>pca</code>
<code>screepplot</code>	Scree plot of eigenvalues
<code>sdtest1</code>	Variance-comparison test against constant
<code>sdtest2</code>	Variance-comparison test between variables
<code>sdtest, by()</code>	Variance-comparison test over groups
<code>sem, method(adf) (CFA)</code>	Confirmatory factor analysis, ADF estimation
<code>sem, method(ml) (CFA)</code>	Confirmatory factor analysis, ML estimation
<code>sem, method(mlmv) (CFA)</code>	Confirmatory factor analysis, ML estimation with missing values
<code>sem (SEM latent)</code>	Structural equations model with latent variables, ML estimation
<code>sem (SEM observed)</code>	Structural equations model on observed variables, ML estimation
<code>separate</code>	Create separate variables
<code>sfrancia</code>	Shapiro–Francia test for normality
<code>signrank</code>	Equality tests on matched data
<code>signtest</code>	Equality tests on matched data
<code>sktest</code>	Skewness and kurtosis test for normality
<code>slogit</code>	Stereotype logistic regression
<code>sort</code>	Sort data
<code>spearman</code>	Spearman’s rank correlation coefficients
<code>sspace</code>	State-space models

Table 2. Command descriptions

Command	Description
<code>stack</code>	Stack data
<code>stci</code>	Confidence intervals for means and percentiles of survival time
<code>stcox</code>	Fit Cox proportional hazards model
<code>stcrreg</code>	Competing-risks regression
<code>stgen</code>	Generate variables reflecting entire histories
<code>stintcox, interval()</code>	Cox proportional hazards model for interval-censored survival-time data
<code>stintcox, id() time() status()</code>	Multiple-record-per-subject interval-censored Cox proportional hazards model
<code>stintreg, d(exponential)</code>	Fit parametric models for interval-censored survival-time data, exponential distribution
<code>stintreg, d(weibull)</code>	Fit parametric models for interval-censored survival-time data, Weibull distribution
<code>stir</code>	Report incidence-rate comparison
<code>stmc</code>	Calculate rate ratios with the Mantel–Cox method
<code>by: stmc</code>	Calculate rate ratios with the Mantel–Cox method over groups
<code>stmh</code>	Calculate rate ratios with the Mantel–Haenszel method
<code>by: stmh</code>	Calculate rate ratios with the Mantel–Haenszel method over groups
<code>stptime</code>	Calculate person-time, incidence rates, and SMR
<code>strate</code>	Tabulate failure rates and rate ratios
<code>streg, distribution(exponential)</code>	Fit parametric survival models, exponential distribution
<code>streg, dist(exp) vce(cluster)</code>	Fit parametric survival models, exponential distribution with cluster-robust standard errors
<code>streg, dist(exp) frailty()</code>	Fit parametric survival models, exponential distribution with individual frailty

Table 2. Command descriptions

Command	Description
<code>streg, dist(exp) frailty() shared()</code>	Fit parametric survival models, exponential distribution with shared frailty
<code>streg, dist(exp) vce(robust)</code>	Fit parametric survival models, exponential distribution with robust standard errors
<code>streg, distribution(ggamma)</code>	Fit parametric survival models, generalized-gamma distribution
<code>streg, dist(ggamma) vce(cluster)</code>	Fit parametric survival models, generalized-gamma distribution with cluster-robust standard errors
<code>streg, dist(ggamma) vce(robust)</code>	Fit parametric survival models, generalized-gamma distribution with robust standard errors
<code>streg, distribution(gompertz)</code>	Fit parametric survival models, Gompertz distribution
<code>streg, dist(gompertz) vce(cluster)</code>	Fit parametric survival models, Gompertz distribution with cluster-robust standard errors
<code>streg, dist(gompertz) frailty()</code>	Fit parametric survival models, Gompertz distribution with individual frailty
<code>streg, dist(gomp) frailty() shared()</code>	Fit parametric survival models, Gompertz distribution with shared frailty
<code>streg, dist(gompertz) vce(robust)</code>	Fit parametric survival models, Gompertz distribution with robust standard errors
<code>streg, distribution(llogistic)</code>	Fit parametric survival models, log-logistic distribution
<code>streg, dist(llogistic) vce(cluster)</code>	Fit parametric survival models, log-logistic distribution with cluster-robust standard errors
<code>streg, dist(llogistic) frailty()</code>	Fit parametric survival models, log-logistic distribution with individual frailty
<code>streg, dist(llog) frailty() shared()</code>	Fit parametric survival models, log-logistic distribution with shared frailty
<code>streg, dist(llogistic) vce(robust)</code>	Fit parametric survival models, log-logistic distribution with robust standard errors
<code>streg, distribution(lnormal)</code>	Fit parametric survival models, log-normal distribution
<code>streg, dist(lnormal) vce(cluster)</code>	Fit parametric survival models, log-normal distribution with cluster-robust standard errors
<code>streg, dist(lnormal) frailty()</code>	Fit parametric survival models, log-normal distribution with individual frailty
<code>streg, dist(lnorm) frailty() shared()</code>	Fit parametric survival models, log-normal distribution with shared frailty

Table 2. Command descriptions

Command	Description
<code>streg, dist(lnormal) vce(robust)</code>	Fit parametric survival models, log-normal distribution with robust standard errors
<code>streg, distribution(weibull)</code>	Fit parametric survival models, Weibull distribution
<code>streg, dist(weibull) vce(cluster)</code>	Fit parametric survival models, Weibull distribution with cluster-robust standard errors
<code>streg, dist(weibull) frailty()</code>	Fit parametric survival models, Weibull distribution with individual frailty
<code>streg, dist(weib) frailty() shared()</code>	Fit parametric survival models, Weibull distribution with shared frailty
<code>streg, dist(weibull) vce(robust)</code>	Fit parametric survival models, Weibull distribution with robust standard errors
<code>sts generate</code>	Create new variables containing survival, hazard, and related functions
<code>sts graph</code>	Compute and graph survival, hazard, and related functions
<code>sts list</code>	Compute and list survival and related functions
<code>sts test</code>	Test the equality of the survival function across groups
<code>stset</code>	Declare data to be survival-time data
<code>stsplit</code>	Split time-span records
<code>stsum</code>	Summarize survival-time data
<code>stteffects ipw (weibull)</code>	Treatment-effects estimation for survival data, inverse-probability weighting, Weibull distribution
<code>stteffects ipwra (weibull)</code>	Treatment-effects estimation for survival data, inverse-probability weighted regression adjustment, Weibull distribution
<code>stteffects ra (weibull)</code>	Treatment-effects estimation for survival data, regression adjustment, Weibull distribution
<code>stteffects wra (weibull)</code>	Treatment-effects estimation for survival data, weighted regression adjustment, Weibull distribution
<code>stvary</code>	Report variables that vary over time
<code>suest</code>	Seemingly unrelated estimation

Table 2. Command descriptions

Command	Description
<code>summarize</code>	Summary statistics
<code>sunflower</code>	Density-distribution sunflower plots
<code>sureg</code>	Zellner's seemingly unrelated regression
<code>svar</code>	Structural vector autoregression models
<code>svmat</code>	Convert variables to matrix and vice versa
<code>svy brr: logit</code>	Logistic regression using survey data—balanced repeated replications
<code>svy brr: poisson</code>	Poisson regression using survey data—balanced repeated replications
<code>svy brr: regress</code>	Linear regression using survey data—balanced repeated replications
<code>svy jackknife: logit</code>	Logistic regression using survey data—jackknife
<code>svy jackknife: poisson</code>	Poisson regression using survey data—jackknife
<code>svy jackknife: regress</code>	Linear regression using survey data—jackknife
<code>svy linearized: logit</code>	Logistic/logit regression using survey data—linearization
<code>svy linearized: poisson</code>	Poisson regression using count survey data—linearization
<code>svy linearized: regress</code>	Linear regression using survey data—linearization
<code>swilk</code>	Shapiro–Wilk test for normality
<code>symmetry</code>	Symmetry and marginal homogeneity tests
<code>table (one-way)</code>	Table of frequencies, summaries, and command results, one-way
<code>table (two-way)</code>	Table of frequencies, summaries, and command result, stwo-way
<code>tabodds</code>	Tabulate odds of failure by category

Table 2. Command descriptions

Command	Description
<code>tabodds (adjusted)</code>	Tabulate odds of failure by category adjusting for levels
<code>tabstat</code>	Display table of summary statistics
<code>tabstat, by()</code>	Display table of summary statistics over groups
<code>tabulate (one-way)</code>	Tables of frequencies, one-way
<code>tabulate (two-way)</code>	Tables of frequencies, two-way
<code>teffects aipw (linear)</code>	Treatment-effects estimation for linear regression, augmented inverse-probability weighting
<code>teffects aipw (probit)</code>	Treatment-effects estimation for probit regression, augmented inverse-probability weighting
<code>teffects ipw (logit)</code>	Treatment-effects estimation for linear regression, inverse-probability weighting
<code>teffects ipwra (linear)</code>	Treatment-effects estimation for linear regression, inverse-probability weight regression adjustment
<code>teffects ipwra (probit)</code>	Treatment-effects estimation for probit regression, augmented inverse-probability weighted regression adjustment
<code>teffects nnmatch</code>	Treatment-effects estimation, nearest-neighbor matching
<code>teffects psmatch, logit</code>	Treatment-effects estimation, propensity-score matching
<code>teffects ra (linear)</code>	Treatment-effects estimation for linear regression, regression adjustment
<code>teffects ra (probit)</code>	Treatment-effects estimation for probit regression, regression adjustment
<code>telasso (, linear) (, probit), ate</code>	Treatment-effects estimation using lasso, linear model, average treatment effect
<code>telasso (, linear) (, probit), atet</code>	Treatment-effects estimation using lasso, linear model, average treatment effect on the treated
<code>telasso (, linear) (, probit), pomeans</code>	Treatment-effects estimation using lasso, linear model, potential-outcome means
<code>telasso (, logit) (, probit), ate</code>	Treatment-effects estimation using lasso, logistic model, average treatment effect
<code>telasso (, logit) (, probit), atet</code>	Treatment-effects estimation using lasso, logistic model, average treatment effect on the treated

Table 2. Command descriptions

Command	Description
<code>telasso (, logit) (, probit), pomeans</code>	Treatment-effects estimation using lasso, logistic model, potential-outcome means
<code>telasso (, poisson) (, probit), ate</code>	Treatment-effects estimation using lasso, Poisson model, average treatment effect
<code>telasso (, poisson) (, probit), atet</code>	Treatment-effects estimation using lasso, Poisson model, average treatment effect on the treated
<code>telasso (, poisson) (, probit), pomeans</code>	Treatment-effects estimation using lasso, Poisson model, potential-outcome means
<code>telasso (, probit) (, probit), ate</code>	Treatment-effects estimation using lasso, probit model, average treatment effect
<code>telasso (, probit) (, probit), atet</code>	Treatment-effects estimation using lasso, probit model, average treatment effect on the treated
<code>telasso (, probit) (, probit), pomeans</code>	Treatment-effects estimation using lasso, probit model, potential-outcome means
<code>tetrachoric</code>	Tetrachoric correlations for binary variables
<code>threshold, threshvar()</code>	Threshold regression, single threshold for the intercept
<code>threshold, threshvar() regionvars()</code>	Threshold regression, single threshold for the intercept and some coefficients
<code>tnbreg</code>	Truncated negative binomial regression
<code>tobit</code>	Tobit regression
<code>tostring</code>	Convert numeric variables to string variables
<code>total</code>	Estimate totals
<code>tpoisson</code>	Truncated Poisson regression
<code>truncreg</code>	Truncated regression
<code>tsfilter bk</code>	Time-series filter, Baxter-King
<code>tsfilter bw</code>	Time-series filter, Butterworth
<code>tsfilter cf</code>	Time-series filter, Christiano-Fitzgerald

Table 2. Command descriptions

Command	Description
<code>tsfilter hp</code>	Time-series filter, Hodrick-Prescott
<code>tsrevar</code>	Create time-series operated temporary variables
<code>tsset</code>	Declare a dataset to be time-series data
<code>tssmooth exp</code>	Exponential smoothing of univariate time-series data
<code>tssmooth ma</code>	Moving average smoothing of univariate time-series data
<code>ttest1</code>	Mean comparison test against constant null hypothesis
<code>ttest2</code>	Mean comparison test against between variables
<code>ttest, by()</code>	Mean comparison test against over groups
<code>twoway ffit</code>	Compute and graph fractional-polynomial fit
<code>twoway lfitci</code>	Compute and graph linear fit with confidence intervals
<code>twoway mband</code>	Compute and graph median bands
<code>twoway mspline</code>	Compute and graph spline smooth
<code>ucm, model(rwdrift)</code>	Unobserved-components model, random walk with drift
<code>var</code>	Vector autoregression models
<code>vargranger</code>	Perform pairwise Granger causality tests after <code>var</code> or <code>svar</code>
<code>varlmar</code>	Obtain LM statistics for residual autocorrelation after <code>var</code> or <code>svar</code>
<code>varnorm</code>	Test for normally distributed disturbances after <code>var</code> or <code>svar</code>
<code>varsoc</code>	Obtain lag-order selection statistics for VARs and VECMs
<code>varstable</code>	Check the stability condition of VAR or SVAR estimates

Table 2. Command descriptions

Command	Description
<code>vec</code>	Vector error-correction models
<code>veclmar</code>	Obtain LM statistics for residual autocorrelation after <code>vec</code>
<code>vecnorm</code>	Test for normally distributed disturbances after <code>vec</code>
<code>vecrank</code>	Estimate the cointegrating rank using Johansen's framework
<code>vecstable</code>	Check the stability condition of VECM estimates
<code>vwls</code>	Variance-weighted least squares
<code>wildbootstrap areg</code>	Wild cluster bootstrap, linear regression with a large dummy-variable set
<code>wildbootstrap regress</code>	Wild cluster bootstrap, linear regression
<code>wildbootstrap xtreg</code>	Wild cluster bootstrap, fixed-effects linear regression
<code>wntestb</code>	Bartlett's periodogram-based test for white noise
<code>wntestq</code>	Portmanteau (Q) test for white noise
<code>xcorr</code>	Cross-correlogram for bivariate time series
<code>xpologit</code>	Cross-fit partialling-out lasso logistic regression
<code>xpopoisson</code>	Cross-fit partialling-out lasso Poisson regression
<code>xporegress</code>	Cross-fit partialling-out lasso linear regression
<code>xtabond</code>	Arellano–Bond linear, dynamic panel-data estimation
<code>xtabond, twostep</code>	Arellano–Bond linear, dynamic panel-data estimation, two-step estimation
<code>xtcloglog, re</code>	Random-effects cloglog models
<code>xtdata, be</code>	Compute between transform of panel data

Table 2. Command descriptions

Command	Description
<code>xtdata, fe</code>	Compute within (fixed-effects) transform of panel data
<code>xtdata, re</code>	Compute random-effects transform of panel data
<code>xtddidregress</code>	Difference-in-differences estimation
<code>xtdpd</code>	Linear dynamic panel-data estimation
<code>xtdpdsys</code>	Arellano–Bover/Blundell–Bond linear dynamic panel-data estimation
<code>xteregress</code>	Extended linear regression with random effects
<code>xtfrontier</code>	Stochastic frontier models for panel data
<code>xtgee, family(gaussian) corr(ar2)</code>	GEE estimation of Gaussian panel-data model with 2-period autocorrelation
<code>xtgee, fam(gauss) corr(unstruct)</code>	GEE estimation of Gaussian panel-data model with unstructured correlation
<code>xtcloglog, pa</code>	Population-averaged cloglog models
<code>xtlogit, pa</code>	Population-averaged logit models
<code>xtnbreg, pa</code>	Population-averaged negative binomial models
<code>xtpoisson, pa</code>	Population-averaged Poisson models
<code>xtprobit, pa</code>	Population-averaged probit models
<code>xtreg, pa</code>	Population-averaged linear models
<code>xtgls</code>	Fit panel-data models using GLS
<code>xthdidregress aipw</code>	Heterogeneous difference in differences for panel data, augmented inverse-probability weighting
<code>xthdidregress ipw</code>	Heterogeneous difference in differences for panel data, inverse-probability weighting
<code>xthdidregress ra</code>	Heterogeneous difference in differences for panel data, regression adjustment

Table 2. Command descriptions

Command	Description
<code>xthdidregress twfe</code>	Heterogeneous difference in differences for panel data, two-way fixed effects
<code>xthtaylor</code>	Hausman–Taylor estimator for error-components models
<code>xtile</code>	Panel-data line plots
<code>xtintreg</code>	Random-effects interval data regression models
<code>xtivreg, be</code>	Instrumental variables and two-stage least squares for panel-data models— between effects
<code>xtivreg, fd</code>	Instrumental variables and two-stage least squares for panel-data models— first differences
<code>xtivreg, fe</code>	Instrumental variables and two-stage least squares for panel-data models— fixed effects
<code>xtivreg, re</code>	Instrumental variables and two-stage least squares for panel-data models— random effects
<code>xtlogit, fe</code>	Fixed-effects logit models
<code>xtlogit, re</code>	Random-effects logit models
<code>xtmlogit, fe</code>	Fixed-effects multinomial logit models
<code>xtmlogit, re</code>	Random-effects multinomial logit models
<code>xtnbreg, fe</code>	Fixed-effects negative binomial models
<code>xtnbreg, re</code>	Random-effects negative binomial models
<code>xtologit</code>	Random-effects ordered logistic models
<code>xtoprobit</code>	Random-effects ordered probit models
<code>xtpcse</code>	OLS or Prais–Winsten models with panel-corrected standard errors
<code>xtpoisson, fe</code>	Fixed-effects Poisson models
<code>xtpoisson, re</code>	Random-effects Poisson models

Table 2. Command descriptions

Command	Description
<code>xtprobit, re</code>	Random-effects probit models
<code>xtrc</code>	Random-coefficients regression
<code>xtreg, be</code>	Between-effects linear models
<code>xtreg, fe</code>	Fixed-effects linear models
<code>xtreg, fe vce(robust)</code>	Fixed-effects linear models, cluster-robust standard errors
<code>xtreg, mle</code>	Random-effects linear models, ML estimation
<code>xtreg, re</code>	Random-effects linear models
<code>xtregar, fe</code>	Fixed-effects linear models with an AR(1) disturbance
<code>xtregar, re</code>	Random-effects linear models with an AR(1) disturbance
<code>xtset</code>	Declare data to be panel data
<code>xtstreg, distribution(exponential)</code>	Random-effects survival models, exponential distribution
<code>xtstreg, distribution(weibull)</code>	Random-effects survival models, Weibull distribution
<code>xtsum</code>	Summarize panel data
<code>xttab</code>	Tabulate panel data
<code>xttobit</code>	Random-effects tobit models
<code>xtunitroot breitung</code>	Panel-data unit-root test—Breitung
<code>xtunitroot fisher</code>	Panel-data unit-root test—Fisher
<code>xtunitroot hadri</code>	Panel-data unit-root test—Hadri Lagrange multiplier
<code>xtunitroot ht</code>	Panel-data unit-root test—Harris–Tzavalis

Table 2. Command descriptions

Command	Description
<code>xtunitroot ips</code>	Panel-data unit-root test—Im—Pesaran—Shin
<code>xtunitroot llc</code>	Panel-data unit-root test—Levin—Lin—Chu
<code>zinb</code>	Zero-inflated negative binomial regression
<code>ziologit</code>	Zero-inflated ordered logit regression
<code>zioprobit</code>	Zero-inflated ordered probit regression
<code>zip</code>	Zero-inflated Poisson regression
<code>_predict, xb</code>	Obtain predictions, residuals, etc., after estimation programming command—option <code>xb</code>
<code>_rmcoll</code>	Remove collinear variables
<code>_robust</code>	Robust variance estimates

D Problem sizes

The following table (table 3) shows the sizes of the problems used to measure the performance gains reported in table 1. As discussed in section 9, these are intentionally large problems requiring considerable 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 several times (several iterations) for an accurate read of the timing required to run the command.

The second through fourth columns of table 3 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 a panel. Columns 3 and 4 provide more information on problem size for longitudinal panel-data problems, and the number of observations, N , is just the product of m and t . Some such problems are not really panel data but merely grouped data; in these cases, the time periods should just be considered the number of observations within group. Almost all the panel-data problems were created with balanced panels (an 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 d_1 records a miscellaneous dimension, such as number of equations for problems that involve multiple equations. Some commands have more miscellaneous dimensions; only d_1 is shown here.

The column labeled 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			k	d_1	n_{iter}
	N	m	t			
alpha	2250000			20		1
ameans	3000000			5		1
anova (one-way)	80000000			200		1
anova (two-way)	10000000			10		1
arch	80000			1		1
areg	6000000			20	30000	1
areg, vce(cluster)	2000000			20	20000	1
areg, vce(robust)	2000000			20	20000	1
arfima	1000			1		1
arima	80000			1		1
bayes dsge	1000			4		1
bayes dsgenl	1000			4		1
bayes: logit	300000			20		1
bayes: poisson	200000			20		1
bayes: regress	300000			20		1
bayes var	1000			2	5	1
bayesmh logit	10000			50		1
bayesmh mvn	20000			30	3	1
bayesmh mylogit	10000			10		1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
bayesmh nl	10000			10		1
bayesmh normal	10000			100		1
bayesmh normal gibbs	10000			10		1
bayesmh normal re		10	100	100		1
betareg, link(logit)	100000			200		1
betareg, link(probit)	100000			200		1
binreg	200000			200		1
biplot	4000			2		1
biprobit	160000			40	40	1
biprobit (seemingly unrelated)	160000			40	40	1
bitest	10000000			1	2	10
blogit	200000			200	50	1
bmaregress	300000			20		1
boxcox	100000			200		1
bprobit	200000			200	50	1
brier	150000					1
bsample	100000			100		20
bstat	1000000			10		1
by: generate		1000000	100			6

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
by: generate (small groups)		9000000	10			2
by: replace		1000000	100			6
by: replace (small groups)		9000000	10			2
ca	10000000				5	1
candisc		5	40000	150		1
canon	4000000				30	1
cc	500000					1
by: cc	100000				20	1
centile	1000000			2		1
churdle linear	200000			50	50	1
ci means	1000000			50		1
ci means, poisson	100000			50		8
ci proportions	1000000			50		1
clogit (k1 to k2 matching)		20000	10	30		1
clogit (1 to k matching)		50000	10	50		1
cloglog	200000			100		1
cluster averagelinkage	4000			200		1
cluster centroidlinkage	4000			200		1
cluster completelinkage	4000			200		1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
cluster generate	2000			200		1
cluster kmeans	50000			30		1
cluster kmedians	50000			30		1
cluster medianlinkage	5000			200		1
cluster singlelinkage	5000			5		1
cluster wardslinkage	3000			200		1
cluster waveragelinkage	3000			200		1
cmclogit	3300			100	10	1
cmmprobit		200	3	2	2	1
cmroprobit	300			2	3	1
cnsreg	1400000			200		1
codebook	150000			25		1
collapse	300000			50	100	1
compare	6000000			2		2
compress	500000			50	50	1
contract	1000000			20	100	1
corr2data	200000			50		1
correlate	3000000			200		1
corrgram	80000			1		1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations					
	N	m	t	k	d_1	n_{iter}
count	20000000					20
cpoisson	100000			100		1
cs	10000000					1
by: cs	60000				100	1
ctset	40000000					15
cttost	50000					1
cumul	1000000			2		1
cusum	1500000			1		1
datasignature	500000			300		1
decode		10000	1000			1
demandsys aids	1000			10		1
demandsys cdouglas	50000			10		1
demandsys gaids	1000			10		1
demandsys gquaid	1000			10		1
demandsys gtranslog	1000			10		1
demandsys les	25000			10		1
demandsys quaid	1000			10		1
demandsys translog	1000			10		1
destring		4000	2000			1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations					
	N	m	t	k	d_1	n_{iter}
dfactor	2000			3		1
dfgls	20000			1		1
dfuller	5000000			1		3
didregress		3000	50	25		1
discrim knn		5	1000	20		1
discrim lda		50	2000	10		1
discrim logistic		50	400	10		1
discrim qda		50	2000	10		1
dotplot	100000			10		1
drawnorm	100000			150		1
drop if <i>exp</i>	10000000			4		1
drop in <i>range</i>	10000000			4		1
dsge	10000			4		1
dsgenl	10000			4		1
dslogit	100000			40		1
dspoisson	100000			40		1
dsregress	100000			40		1
dstdize		10	150	200		1
dvech	500			2		1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
egen group()		1	800000	500		1
by: egen mean		400	10000	2		1
eivreg	1400000			200		1
encode		50	220000			1
eregress	20000			10	10	1
esize twosample	10000000					1
esize unpaired	30000000					1
eteffects (exponential), ate	20000			20		1
eteffects (linear), ate	10000			100		1
eteffects (linear), pomeans	10000			100		1
eteffects (probit), ate	10000			100		1
etpoisson	10000			10	10	1
etregress, poutcomes	10000			30	30	1
etregress, twostep	800000			50	50	1
exlogistic	100			3		1
expand #	10000			800		1
expand <i>varname</i>	100000			100	5	1
expandcl #		12000	10	100		1
expandcl <i>varname</i>		30000	10	80	5	1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
expoisson	50			20		1
factor	10000000			50		1
fcast compute	10000			2	5	1
fillin		80	1			1
fmm 2: poisson	50000			30	2	1
fmm 2: regress	50000			30	2	1
fmm 3: poisson	50000			20	3	1
fmm 3: regress	5000			20	3	1
fracreg probit	200000			200		1
frontier	400000			200		1
fvrevar (factors)	1000000			4	80	1
fvrevar (interaction)	5000000			2	8	1
generate (small expressions)	60000			4000		1
generate	5000000					1
glm, family(gamma)	700000			100		1
glm, family(gaussian)	700000			200		1
glm, family(igaussian)	500000			200		1
glm, family(nbinomial)	300000			200		1
glm, family(poisson)	300000			200		1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
<code>glogit</code>	2000000			100	50	1
<code>gmm</code>	1000			10		1
<code>gmm (with derivatives)</code>	100000			10		1
<code>gprobit</code>	3000000			100	50	1
<code>graph bar</code>	500000			10	3	1
<code>graph box</code>	200000			2	10	1
<code>graph pie</code>	2500000			10	10	1
<code>grmeanby</code>	300000			4	10	1
<code>gsem, oprobit (CFA, 2-level)</code>		1000	10	4	1	1
<code>gsem, oprobit (CFA)</code>	5000			4	1	1
<code>gsem, logit group()</code>		5	50000	40		1
<code>gsem, group()</code>		5	50000	40		1
<code>gsem, ologit group()</code>		5	50000	40		1
<code>gsem, poisson group()</code>		5	50000	40		1
<code>gsort</code>	1000000			5		1
<code>hausman</code>	200					1
<code>hdidregress aipw</code>		5000	5	25		1
<code>hdidregress ipw</code>		5000	5	25		1
<code>hdidregress ra</code>		5000	5	25		1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
hdidregress twfe		1000	5	5		1
heckman	500000			100	50	1
heckman, twostep	1000000			100	50	1
heckoprobit	100000			10	50	1
heckpoisson	10000			40	20	1
heckprob	200000			50	50	1
hetoprobit	300000			10	10	1
hetprob	300000			10	10	1
hetregress	500000			100	50	1
hetregress, twostep	1000000			50	5	1
histogram	4000000			1		1
hotelling	4000000			100		1
icc, mixed	1000000			100		1
icc (one-way)	3000000			300		1
icc (two-way)	1000000			100		1
import delimited	500000			200		1
intreg	200000			200		1
ir	10000000					1
by: ir	10000				200	1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
irf create	1000000			2	3	1
irt 1pl	40000			20		1
irt 2pl	40000			20		1
irt 3pl	40000			10		1
irt grm	20000			10		1
irt nrm	20000			10		1
irt pcm	20000			10		1
irt rsm	20000			10		1
istdize		50	100	10000		1
ivfprobit	150000			30	20	1
ivpoisson cfunction	60000			5	5	1
ivpoisson gmm, additive	80000			5	5	1
ivpoisson gmm, multiplicative	160000			5	5	1
ivprobit	150000			30	20	1
ivqregress iqr	1000			10		1
ivregress 2sls	800000			50	20	1
ivregress gmm	1500000			20	20	1
ivregress liml	2000000			20	20	1
ivtobit	150000			50	20	1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
kap	500000			2	10	4
kappa	2000000			10	20	1
kdensity	10000000					1
keep if <i>exp</i>	10000			4000		1
keep in <i>range</i>	20000			4000		1
keep <i>varlist</i>	50000			4000		1
ksmirnov	2000000					1
ksmirnov, by()	1000000					1
ktau	5000			5		1
kwallis	1500000			10		1
ladder	2000000					1
lasso linear	100000			20		1
lasso logit	20000			20		1
lasso poisson	20000			20		1
gsem, lclass(C 2)	500000			5	2	1
gsem, lclass(C 3)	50000			10	3	1
levelsof	80000000				20	1
loadingplot	2000000			60		1
logistic	300000			200		1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
logit	300000			200		1
loneway	2000000			500		1
lowess	90000			1		1
lpoly	1000000					1
ltable	50000			1		40
makespline bspline	5000000			1		1
makespline linear	10000000			1		1
makespline piecewise	5000000			1		1
makespline rcs	5000000			1		1
manova (one-way)	20000000			50	3	1
manova (two-way)	2000000			20	3	1
margins	250000			40	10	1
margins, dydx() exp()	30000			40	10	1
margins, dydx()	20000			40	10	1
margins, exp()	40000			40	10	1
markout	500000			500		1
marksample	1200000			200		1
marksample if <i>exp</i>	2300000			100		1
matrix accum	3000000			200		1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations					
	N	m	t	k	d_1	n_{iter}
<code>matrix eigenvalues</code>	500			500		1
<code>matrix score</code>	6000000			1000		1
<code>matrix svd</code>	300			300		1
<code>matrix symeigen</code>	600			600		1
<code>matrix syminv</code>	2000			2000		1
<code>mca</code>	1000000			3	5	1
<code>mcc</code>	10000000					1
<code>mds</code>	800			400		1
<code>mdslong</code>		600	1			1
<code>mean</code>	1000000			200		1
<code>mecloglog</code>		2000	10	2	1	1
<code>median</code>	8000000			5		1
<code>mediate (, expmean) (, linear)</code>	10000			20		1
<code>mediate (, linear) (, linear)</code>	10000			20		1
<code>mediate (, logit) (, linear)</code>	10000			20		1
<code>mediate (, poisson) (, linear)</code>	10000			20		1
<code>mediate (, probit) (, linear)</code>	10000			20		1
<code>meintreg</code>		1000	50	5	1	1
<code>melogit</code>		4000	10	10	1	1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations					
	N	m	t	k	d_1	n_{iter}
menbreg, dispersion(constant)		2000	5	2	1	1
menbreg, dispersion(mean)		4000	10	2	1	1
menl		1000	5	3	1	1
meologit		4000	10	5	1	1
meoprobit		4000	10	2	1	1
mepoisson		4000	10	2	1	1
meprobit		4000	10	10	1	1
mestreg, distribution(exp)		4000	10	10	1	1
mestreg, distribution(weibull)		4000	10	10	1	1
metobit		1000	50	5	1	1
mgarch	1000			3	2	1
mhodds	3000000					1
mhodds (adjusted)	400000			400		1
by: mhodds	50000				100	1
mhodds (trend)	1000000				100	1
mi estimate: logit (flong)	100000			180	20	1
mi estimate: logit (flongsep)	100000			180	20	1
mi estimate: logit (mlong)	100000			180	20	1
mi estimate: logit (wide)	70000			180	20	1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
mi estimate: mlogit	100000			100	10	1
mi estimate: ologit	120000			190	10	1
mi estimate: regress (flong)	100000			300	20	1
mi estimate: regress (flongsep)	100000			300	20	1
mi estimate: regress (mlong)	100000			300	20	1
mi estimate: regress (wide)	60000			300	20	1
mi impute chained (flong)	20000			20	20	1
mi impute chained (flongsep)	20000			20	20	1
mi impute chained (mlong)	20000			20	20	1
mi impute chained (wide)	20000			20	20	1
mi impute logit (flong)	100000			100	1	1
mi impute logit (flongsep)	100000			100	1	1
mi impute logit (mlong)	100000			100	1	1
mi impute logit (wide)	200000			100	1	1
mi impute mlogit	100000			100	1	1
mi impute mono pmm	10000			50	3	1
mi impute mono regress	40000			200	10	1
mi impute mvn	1000			10	10	1
mi impute ologit	40000			100	1	1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
mi impute pmm	20000			200	1	1
mi impute regress	40000			100	1	1
misstable nested	2000000			20		1
misstable patterns	2000000			20		1
misstable summarize	5000			10		1
misstable tree	1000000			20		1
mixed		500	10	5	5	1
mixed (crossed effects)		10	1000			1
mkspline	12000000			1		1
mleval	30000000			200		1
mleval, nocons	30000000			200		1
mlmatbysum	20000000			200	160000	1
mlmatsum	20000000			200		1
mlogit	500000			100	3	1
mlsum	4.0e+08			1		1
mlvecsum	20000000			400		1
mprobit	800			10	3	1
mswitch ar		100	100	20	5	1
mswitch dr		100	100	20	5	1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
mvdecode	500000			20	1000	1
mvencode	6000000			20	1000	1
mvreg	2000000			100	3	1
mvtest correlations		2	600000	100		1
mvtest covariances		2	600000	100		1
mvtest means, heterogeneous		2	400000	100		1
mvtest means, homogeneous		2	150000	100		1
mvtest means, lr		2	500000	100		1
mvtest normality	1000			20		1
nbreg	60000			200		1
newey	500000			5		1
nl	1500000					1
nlogit		1200	2	2	3	1
nlsur	100000			2		1
npregress kernel	1000			2		1
nptrend, cuzick	1000000			10	1000	1
nptrend, carmitage	1000000			10		1
nptrend, jterpstra	1000000			10	1000	1
nptrend, linear	1000000			10	1000	1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
ologit	700000			100	3	1
ologit, vce(cluster)	300000			100	3	1
ologit, vce(robust)	700000			100	3	1
oneway	3000000			200		1
oprobit	200000			200	3	1
oprobit, vce(cluster)	100000			200	3	1
oprobit, vce(robust)	200000			200	3	1
orthog	1000000			10		1
pca	600000			100		1
pcorr	1300000			200		1
pctile	16000000			1		1
pergram	10000			1		1
pkcollapse		100	50			1
pkexamine		1	1000000			1
pksumm		200	10			1
poisson	200000			200		1
poisson, vce(cluster)	100000			200		1
poisson, exposure()	200000			200		1
poisson, vce(robust)	200000			200		1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
pologit	100000			40		1
popoisson	100000			40		1
poregress	100000			40		1
pperron	300000			1		1
prais	1000000			5		1
predict, cooksd	600000			300		1
predict, covratio	600000			300		1
predict, dfbeta	400000			200		1
predict, dfits	600000			200		1
predict, e	3000000			1000		1
predict, leverage	1200000			200		1
predict, pr	2500000			1000		1
predict, residuals	6000000			1000		1
predict, rstandard	400000			400		1
predict, rstudent	400000			400		1
predict, stdf	1600000			200		1
predict, stdp	400000			400		1
predict, stdr	400000			400		1
predict, welsch	300000			300		1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
predict, ystar	3000000			1000		1
predictnl	60000			200		1
probit	500000			200		1
procrustes	200000			50	50	1
proportion	300000			10	5	1
prtest1	20000000			1	2	3
prtest2	20000000			2	2	2
prtest, by()	10000000			2	2	1
pwcorr	30000000			3		1
qreg	100000			20		1
ranksum	4000000			2		1
ratio	8000000					1
ratio (exp1) (exp2)	9000000					1
recode	1500000			5	5	1
reg3	90000			100	3	1
regress	3000000			180		1
regress, vce(cluster)	1500000			180		1
regress, vce(robust)	300000			180		1
replace	15000000					1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
replace (small expressions)	150000			4000		1
reri logistic	300000			200		1
reri nbreg	60000			200		1
reri poisson	200000			200		1
reri stcox	250000			10		1
reri stintcox, interval()	5000			3		1
reri stintcox, id() time() status()		5000	5	7	10	1
reri stintreg	500000			30		1
reri streg	600000			100		1
reshape long		50000	20			1
reshape wide		50000	15	5		1
robvar	200000			2		1
rocfits	100000			1	5	1
roctab	600000			1	20	1
rotate	10000			80		1
rotatemat	80			80		1
rreg	100000			200		1
runttest	6000000			1		1
scobit	120000			200		1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
scoreplot	400000			20		1
screepplot	10000000			20		1
sdtest1	24000000					3
sdtest2	12000000			2		3
sdtest, by()	9000000					2
sem, method(adf) (CFA)	150000			5	3	1
sem, method(ml) (CFA)	2500000			10	3	1
sem, method(mlmv) (CFA)	100000			4	3	1
sem (SEM latent)	10000000			4	3	1
sem (SEM observed)	5000000			20	3	1
separate	1000000			100	4	1
sfrancia	1000000			2		1
signrank	2500000			2		1
signtest	1.0e+08			2		1
sktest	6000000			2		1
slogit	20000			10	5	1
sort	9000000			10		1
spearman	400000			3		1
sspace	5000			20		1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
<code>stack</code>	500000			100		1
<code>stci</code>	200000			1		1
<code>stcox</code>	250000			10		1
<code>stcrreg</code>	2000			5		1
<code>stgen</code>	30000000			2		1
<code>stintcox, interval()</code>	5000			3		1
<code>stintcox, id() time() status()</code>		5000	5	10	10	1
<code>stintreg, d(exponential)</code>	500000			30		1
<code>stintreg, d(weibull)</code>	200000			20		1
<code>stir</code>	4500000			1	2	1
<code>stmcc</code>	900000					1
<code>by: stmcc</code>	600000				50	1
<code>stmh</code>	1500000					1
<code>by: stmh</code>	1500000				10	1
<code>stptime</code>	9000000			1	60000	1
<code>strate</code>	1000000			1	5	1
<code>streg, distribution(exponential)</code>	600000			100		1
<code>streg, dist(exp) vce(cluster)</code>	200000			200	1000	1
<code>streg, dist(exp) frailty()</code>	60000			200		1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
streg, dist(exp) frailty() shared()	200000			100	1000	1
streg, dist(exp) vce(robust)	200000			200		1
streg, distribution(ggamma)	100000			200		1
streg, dist(ggamma) vce(cluster)	200000			200	1000	1
streg, dist(ggamma) vce(robust)	200000			200		1
streg, distribution(gompertz)	200000			50		1
streg, dist(gompertz) vce(cluster)	200000			50	1000	1
streg, dist(gompertz) frailty()	200000			50		1
streg, dist(gomp) frailty() shared()	200000			10	1000	1
streg, dist(gompertz) vce(robust)	200000			50		1
streg, distribution(llogistic)	600000			100		1
streg, dist(llogistic) vce(cluster)	200000			200	1000	1
streg, dist(llogistic) frailty()	60000			200		1
streg, dist(llog) frailty() shared()	200000			100	1000	1
streg, dist(llogistic) vce(robust)	200000			200		1
streg, distribution(lnormal)	200000			100		1
streg, dist(lnormal) vce(cluster)	200000			200	1000	1
streg, dist(lnormal) frailty()	60000			200		1
streg, dist(lnorm) frailty() shared()	200000			10	1000	1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
streg, dist(lnormal) vce(robust)	200000			200		1
streg, distribution(weibull)	200000			200		1
streg, dist(weibull) vce(cluster)	200000			200	1000	1
streg, dist(weibull) frailty()	200000			50		1
streg, dist(weib) frailty() shared()	100000			100	1000	1
streg, dist(weibull) vce(robust)	200000			200		1
sts generate	1000000			1		1
sts graph	1000000			1		1
sts list	3000000			1		1
sts test	1000000			1	2	1
stset	3000000					1
stsplit	2000000				50	1
stsum	200000			1		1
stteffects ipw (weibull)	50000			50		1
stteffects ipwra (weibull)	20000			20		1
stteffects ra (weibull)	10000			50		1
stteffects wra (weibull)	10000			50		1
stvary	3000000			5		1
suest	400000			200		1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
summarize	4500000			200		1
sunflower	1000000			2		1
sureg	300000			100	2	1
svar	40000			2	10	1
svmat	3000			3000		1
svy brr: logit		128	200	20		1
svy brr: poisson		16	4000	20		1
svy brr: regress		16	6000	200		1
svy jackknife: logit		5	400	20	20	1
svy jackknife: poisson		5	300	20	20	1
svy jackknife: regress		3	3000	10	20	1
svy linearized: logit	200000			200		1
svy linearized: poisson	200000			200		1
svy linearized: regress	400000			200		1
swilk	150000			20		1
symmetry	800000			2	50	1
table (one-way)	4000000			20		1
table (two-way)	3000000			20		1
tabodds	300000				20	1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
tabodds (adjusted)	50000			10	20	1
tabstat	2000000			50		1
tabstat, by()	2000000			20		1
tabulate (one-way)	6000000			20		1
tabulate (two-way)	10000000			20		1
teffects aipw (linear)	10000			50		1
teffects aipw (probit)	10000			50		1
teffects ipw (logit)	20000			100		1
teffects ipwra (linear)	10000			50		1
teffects ipwra (probit)	10000			50		1
teffects nnmatch	20000			100		1
teffects psmatch, logit	10000			50		1
teffects ra (linear)	10000			100		1
teffects ra (probit)	10000			100		1
telasso (, linear) (, probit), ate	200000			100		1
telasso (, linear) (, probit), atet	200000			100		1
telasso (, linear) (, probit), pomeans	200000			100		1
telasso (, logit) (, probit), ate	200000			100		1
telasso (, logit) (, probit), atet	200000			100		1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
telasso (, logit) (, probit), pomeans	200000			100		1
telasso (, poisson) (, probit), ate	50000			100		1
telasso (, poisson) (, probit), atet	50000			100		1
telasso (, poisson) (, probit), pomeans	50000			100		1
telasso (, probit) (, probit), ate	200000			100		1
telasso (, probit) (, probit), atet	200000			100		1
telasso (, probit) (, probit), pomeans	200000			100		1
tetrachoric	1200000			4	2	1
threshold, threshvar()	1000			20	0	1
threshold, threshvar() regionvars()	1000			10	10	1
tnbreg	300000			10		1
tobit	300000			200		1
tostring		10000	200			1
total	600000			200		1
tpoisson	1000000			50		1
truncreg	150000			200		1
tsfilter bk	1000000			1		1
tsfilter bw	1500			1		1
tsfilter cf	1000000			1		1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
tsfilter hp	1500			1		1
tsrevar	1100000			20		1
tsset	4000000					1
tssmooth exp	1000000			1		1
tssmooth ma	1000000			1		1
ttest1	15000000			1		5
ttest2	35000000			2		1
ttest, by()	20000000					1
twoway fpfit	400000			1		1
twoway lfitci	6000000			1		1
twoway mband	3000000			1		1
twoway mspline	4000000			1		1
ucm, model(rwdrift)	5000			3		1
var	250000			2	5	1
vargranger	4000000			2	5	5
varlmar	80000			2	5	1
varnorm	300000			2	5	1
varsoc	200000			2	5	1
varstable	4000000			2	10	5

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
vec	30000			2	10	1
veclmar	50000			2	5	1
vecnorm	150000			2	5	1
vecrank	200000			2	5	1
vecstable	1000000			2	10	1
vwls	1000000			200		1
wildbootstrap areg	100000			20	1000	1
wildbootstrap regress	50000			100		1
wildbootstrap xtreg		1000	5	10		1
wntestb	10000			1		1
wntestq	400000			1		1
xcorr	400000			1		1
xpologit	10000			40		1
xpopoisson	10000			40		1
xporegress	10000			40		1
xtabond		100000	10	2		1
xtabond, twostep		100000	10	2		1
xtcloglog, re		20000	5	5		1
xtdata, be		15000	5	200		1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
<code>xtdata, fe</code>		500000	5	5		1
<code>xtdata, re</code>		300000	5	5		1
<code>xtdidregress</code>		3000	50	25		1
<code>xtdpd</code>		40000	5	5		1
<code>xtdpdsys</code>		60000	5	5		1
<code>xteregress</code>		1000	5	1	1	1
<code>xtfrontier</code>		4000	10	50		1
<code>xtgee, family(gaussian) corr(ar2)</code>		50000	5	10		1
<code>xtgee, fam(gauss) corr(unstruct)</code>		60000	5	10		1
<code>xtcloglog, pa</code>		100000	5	5		1
<code>xtlogit, pa</code>		100000	5	5		1
<code>xtnbreg, pa</code>		80000	5	5		1
<code>xtpoisson, pa</code>		30000	10	5		1
<code>xtprobit, pa</code>		60000	10	5		1
<code>xtreg, pa</code>		100000	5	10		1
<code>xtgls</code>		5	200000	5		1
<code>xthdidregress aipw</code>		5000	5	25		1
<code>xthdidregress ipw</code>		5000	5	25		1
<code>xthdidregress ra</code>		5000	5	25		1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
<code>xthdidregress twfe</code>		1000	5	5		1
<code>xthtaylor</code>		100000	10	4	4	1
<code>xtile</code>	100000					1
<code>xtintreg</code>		15000	5	5		1
<code>xtivreg, be</code>		120000	5	5	5	1
<code>xtivreg, fd</code>		80000	5	5	5	1
<code>xtivreg, fe</code>		80000	5	5	5	1
<code>xtivreg, re</code>		150000	5	5	5	1
<code>xtlogit, fe</code>		20000	10	50		1
<code>xtlogit, re</code>		40000	5	5		1
<code>xtmlogit, fe</code>		10000	5	10	3	1
<code>xtmlogit, re</code>		5000	10	10	3	1
<code>xtnbreg, fe</code>		70000	5	10		1
<code>xtnbreg, re</code>		40000	5	10		1
<code>xtologit</code>		8000	10	10	0	1
<code>xtoprobit</code>		8000	10	10	0	1
<code>xtpcse</code>		3	80000	50		1
<code>xtpoisson, fe</code>		20000	5	50		1
<code>xtpoisson, re</code>		30000	5	50		1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
xtprobit, re		20000	5	5		1
xtrc		100	10000	5		1
xtreg, be		15000	5	200		1
xtreg, fe		200000	5	100		1
xtreg, fe vce(robust)		50000	10	100		1
xtreg, mle		80000	10	5		1
xtreg, re		20000	3	200		1
xtregar, fe		100000	5	2		1
xtregar, re		90000	5	2		1
xtset		500	5000			1
xtstreg, distribution(exponential)		8000	10	10	0	1
xtstreg, distribution(weibull)		8000	10	10	0	1
xtsum		100000	10	10		1
xttab	1500000			2	50	1
xttobit		50000	5	5		1
xtunitroot breitung		200	3000			1
xtunitroot fisher		50	1000			1
xtunitroot hadri		50	1000			1
xtunitroot ht		300	2000			1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

Table 3. Problem sizes

Command	Observations			k	d_1	n_{iter}
	N	m	t			
<code>xtunitroot ips</code>		1000	20			1
<code>xtunitroot llc</code>		100	500			1
<code>zinb</code>	150000			50	50	1
<code>ziologit</code>	200000			30	30	1
<code>zioprobit</code>	200000			30	30	1
<code>zip</code>	250000			50	50	1
<code>_predict, xb</code>	5000000			1000		1
<code>_rmcoll</code>	6000000			100		1
<code>_robust</code>	3000000			200		1

N , number of observations; m , number of panels; t , number of time periods within each panel; k , number of regressors; d_1 , miscellaneous dimension; and n_{iter} , number of iterations.

E Commands not assessed

Some commands were not explicitly assessed and thus do not appear in table 1 or in the performance graphs in appendix A. These commands fall into several categories, as detailed below.

The presented results for `generate` and `replace` apply for floating point variables only. Some other uses of `generate` and `replace` are not parallelized, so their performance was not assessed. Specifically, for string (`str#` and `strL`) variables, `generate` is not parallelized. For string (`str#` and `strL`) and integer (`byte`, `int`, and `long`) variables, `replace` is not parallelized. In these cases, `generate` and `replace` will promote the variable type when the expression returns a value that does not fit in the current storage type. Type promotion is a global change to the variable and can be triggered at any observation for these commands, so parallelization is not possible.

Replication-based prefix commands, such as `bootstrap`, `fracpoly`, `jackknife`, `mfp`, `permute`, `rolling`, `simulate`, `statsby`, and `stepwise`, were not explicitly assessed. These commands run another target command repeatedly; 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 by the prefix command.

Commands that do not process data or otherwise involve lengthy computations and are therefore inherently fast are not parallelized and so their performance was not assessed. These commands include `camat`, `clear`, `clonevar`, `confirm`, `describe`, `estat`, `estimates`, `factormat`, `fvexpand`, `fvunab`, `lincom`, `nlcom`, `pcamat`, `roccomp`, `rocgold`, `sampsi`, `search`, `stpower`, `svydes`, `test`, `testnl`, `unabbrev`, and `varabbrev`.

Commands that involve file I/O or Internet access are not parallelized and so were not assessed. These include `adoupdate`, `append`, `cf`, `fdadefine`, `fdasave`, `fdause`, `filefilter`, `hsearch`, `icd9`, `icd10`, `infile`, `insheet`, `merge`, `odbc`, `outfile`, `outsheet`, `rmdir`, `save`, `search`, `snapshot`, `use`, `xmlsave`, `xmluse`, `zipfile`, and `unzipfile`.

Only a subset of prediction options were assessed. If all predictions were included, they would unduly dominate the timings. Most other predictions have performances similar to the predictions presented in table 1 and in appendix A. Two prediction-like commands whose results are not obtained from `predict` but whose timings are similar to `predict` are `fracpred` and `dfbeta`.

Some commands are partially parallelized, but their degree of parallelization is extremely variable with respect to the size and characteristics of the data. These commands were not assessed and include `bcskew0`, `lnskew0`, `fracplot`, `fracgen`, `mkmat`, `stbase`, and `stjoin`.

`ac` and `pac` are two time-series commands that are not parallelized and so their performance was not assessed.

`graph twoway` is not parallelized although a few of its plottypes that involve data management or estimation are parallelized, such as `histogram`, `lowess`, `lfit`, and `qfit`. Most statistical graphs in Stata are based on `graph twoway`. Graphs that involve data management or estimation were assessed and appear in table 1 and appendix A. Graphs that do not involve data management or estimation are not parallelized and so their performance was not assessed. These include `acprplot`, `avplot`, `avplots`, `cabiplot`, `caprojection`, `cchart`, `cluster tree`, `cprplot`, `graph twoway`, `lvr2plot`, `mdsshepard`,

`pchart`, `procoverlay`, `qbys`, `qchi`, `qnorm`, `qqplot`, `quantile`, `rchart`, `rocplot`, `rvfplot`, `rvpplot`, `shewhart`, `spikeplt`, `stcoxkm`, `stcurve`, `stphplot`, and `sympplot`.

A number of commands perform similarly to related commands that were assessed, but these commands were not themselves assessed. `bsqreg`, `iqreg`, and `sqreg` perform similarly to `qreg`. `gladder` and `qladder` perform similarly to `ladder`. `gnbreg` is similar to `nbreg`. `xttrans` is similar to `xttab`.

F Mata

Mata is Stata's optimized matrix programming language. It is fully integrated with every aspect of Stata. Some parts of Mata are parallelized and some parts are not. As with Stata, you do not need to change anything to obtain the parallelization speedups; they are automatic.

Those parts of Mata that are parallelized are fully parallelized, meaning that on large enough problems, their speedups will be close to the best theoretical speedups discussed in section 6.

The following Mata functions are parallelized: `Cofc()`, `Cofd()`, `F()`, `Fden()`, `Ftail()`, `acos()`, `arg()`, `asin()`, `atan()`, `atan2()`, `betaden()`, `binomial()`, `binomialtail()`, `binormal()`, `ceil()`, `chi2()`, `chi2den()`, `chi2tail()`, `cofC()`, `cofd()`, `comb()`, `cos()`, `cross()`, `crossdev()`, `day()`, `dgammapda()`, `dgammapdada()`, `dgammapdadx()`, `dgammapdx()`, `dgammapdxdx()`, `digamma()`, `dofC()`, `dofc()`, `dofh()`, `dofm()`, `dofq()`, `dofw()`, `dofy()`, `dow()`, `doy()`, `dunnettprob()`, `exp()`, `exponential()`, `exponentialden()`, `exponentialtail()`, `factorial()`, `floatround()`, `floor()`, `gammaden()`, `gammap()`, `gammaptail()`, `halfyear()`, `hh()`, `hhC()`, `hofd()`, `hours()`, `ibeta()`, `ibetatail()`, `invF()`, `invFtail()`, `invbinomial()`, `invbinomialtail()`, `invchi2()`, `invchi2tail()`, `invdunnettprob()`, `invexponential()`, `invexponentialtail()`, `invgammap()`, `invgammaptail()`, `invibeta()`, `invibetatail()`, `invlogistic()`, `invlogistictail()`, `invnF()`, `invnFtail()`, `invnchi2()`, `invnibeta()`, `invnormal()`, `invnt()`, `invnttail()`, `invt()`, `invttail()`, `invtukeyprob()`, `invweibull()`, `invweibullph()`, `invweibullphtail()`, `invweibullltail()`, `ln()`, `lnfactorial()`, `lngamma()`, `lnigammaden()`, `lnnormal()`, `lnnormalden()`, `logistic()`, `logisticden()`, `logistictail()`, `mdy()`, `minutes()`, `mm()`, `mmC()`, `mod()`, `mofd()`, `month()`, `msofhours()`, `msofminutes()`, `msofseconds()`, `nF()`, `nFden()`, `nFtail()`, `nbetaden()`, `nchi2()`, `nibeta()`, `normal()`, `normalden()`, `npnF()`, `npnchi2()`, `npnt()`, `nt()`, `ntden()`, `nttail()`, `qofd()`, `quadcross()`, `quadcrossdev()`, `quarter()`, `round()`, `seconds()`, `sin()`, `sqrt()`, `ss()`, `st_data()`, `t()`, `tan()`, `tden()`, `trigamma()`, `trunc()`, `ttail()`, `tukeyprob()`, `week()`, `weibull()`, `weibullden()`, `weibullph()`, `weibullphden()`, `weibullphtail()`, `weibullltail()`, `wofd()`, `year()`, `yh()`, `ym()`, `yq()`, and `yw()`.

In addition, matrix multiplication in Mata is fully parallelized, as are Mata's colon operators for performing elementwise computations. All other parts of Mata are either not parallelized or are functions of a mixture of the two.

G GLLAMM

Table 4 below shows results for a few models fit using `gllamm`. This is but a small subset of the models that `gllamm` can fit. Each command is described briefly in table 5.

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. In addition, the model 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, and Rasch models (including multidimensional marginally sufficient Rasch models).

`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 models and uses 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, sections of which have been parallelized, are written in C. 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. Many `gllamm` models are highly parallelized, some not parallelized at all, and others lie somewhere in between.

Table 4. Stata/MP performance, command by command

Command	Speed relative to a single core ^a				Percentage parallelized ^b
	Number of cores				
	2	4	8	16	
Finite mixture model	2.3	3.5	4.5	5.6	86
Item response model	1.4	2.1	2.7	3.1	74
Latent class model	1.3	1.9	2.5	2.9	71
Measurement error model	1.8	2.7	3.8	5.1	86
Rank-outcome latent class	1.8	2.6	3.2	3.8	77
MIMIC model	1.2	1.8	2.5	2.9	72
Random-effects logistic	1.4	2.0	2.7	3.2	73
RE regression	1.5	2.2	3.0	3.7	79
Two-level RE logistic	1.2	1.8	2.3	2.7	70
Random-coefficients Poisson	0.9	0.9	0.9	0.9	0
RE logistic with constant	1.6	2.3	3.2	4.1	82

All values are expressed as the speed relative to the speed of a single core.

a. Bigger is better; 2 is perfect for 2 cores, 4 is perfect for 4 cores, 8 is perfect for 8 cores, and 16 is perfect for 16 cores.

b. Bigger is better; 100 is perfect.

Table 5. Command descriptions

Command	Description
Finite mixture model	Gaussian finite mixture model with two point masses
Item response model	Two-parameter logistic item response model
Latent class model	Gaussian latent class model with two levels in the latent class
Measurement error model	Logistic regression with measurement error in a covariate
Rank-outcome latent class	Latent class model for rank outcomes
MIMIC model	Multiple-indicator, multiple-cause (MIMIC) latent variables structural equation model—ordered logistic
Random-effects logistic	Random-effects (random-intercepts) logistic regression—same as <code>xtlogit</code> , <code>re</code>
RE regression	Continuous (Gaussian distribution) model with random intercepts—same as <code>xtreg</code> , <code>re</code>
Two-level RE logistic	Logistic regression with two levels of random intercepts
Random-coefficients Poisson	Poisson count-data model with random intercepts and a random coefficient
RE logistic with constant	Random-effects (random-intercepts) logistic regression, fewer observations

The graphs below show the observed performances from table 4 in graphical form. Those graphs are followed by graphs showing performance through 40 cores.

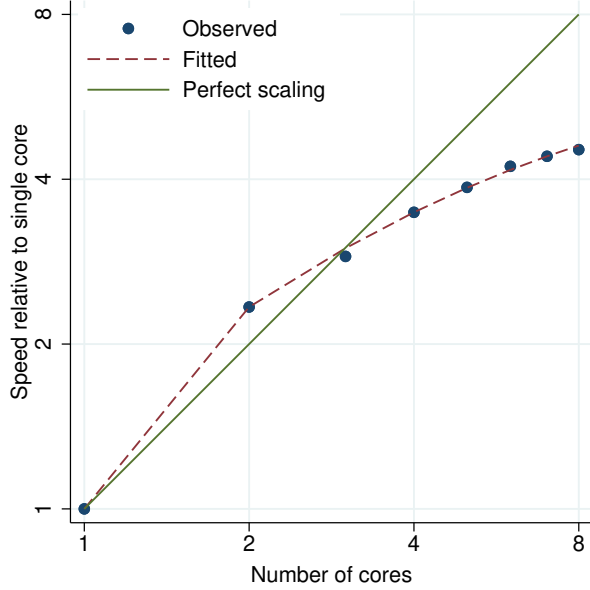


Figure 748. Finite mixture model performance plot.

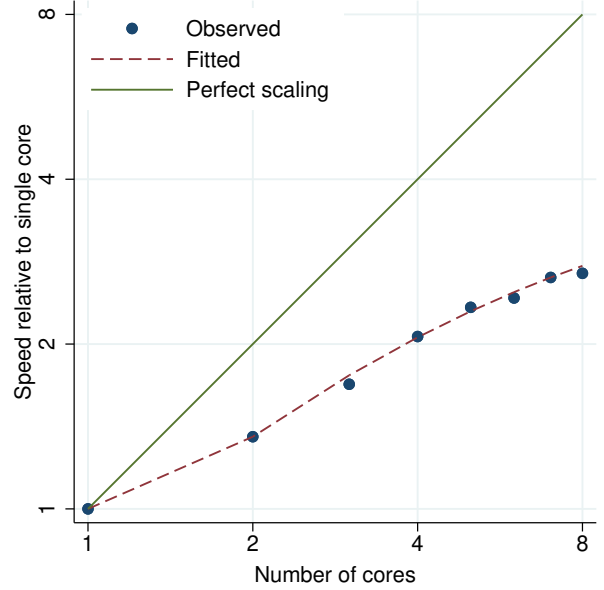


Figure 749. Item response model performance plot.

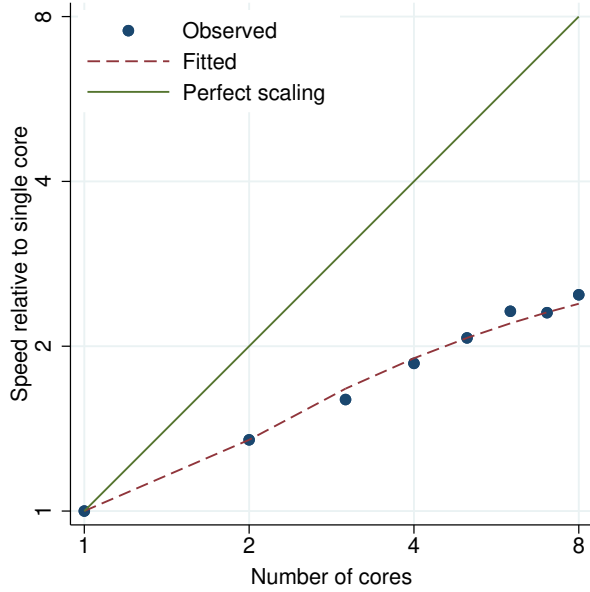


Figure 750. Latent class model performance plot.

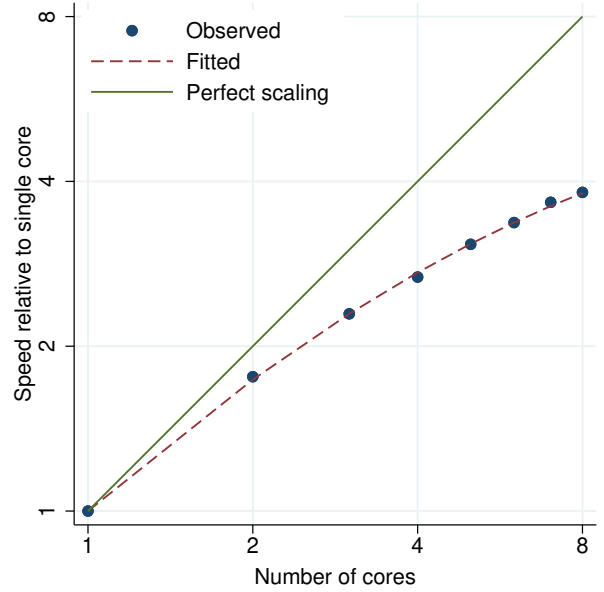


Figure 751. Measurement error model performance plot.

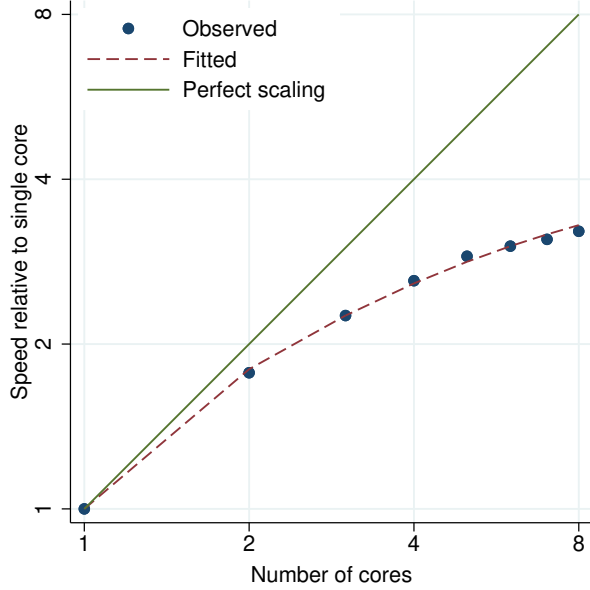


Figure 752. Rank-outcome latent class performance plot.

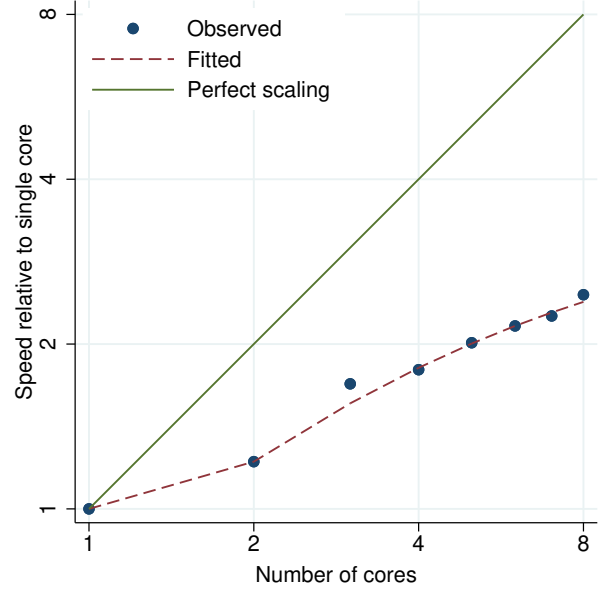


Figure 753. MIMIC model performance plot.

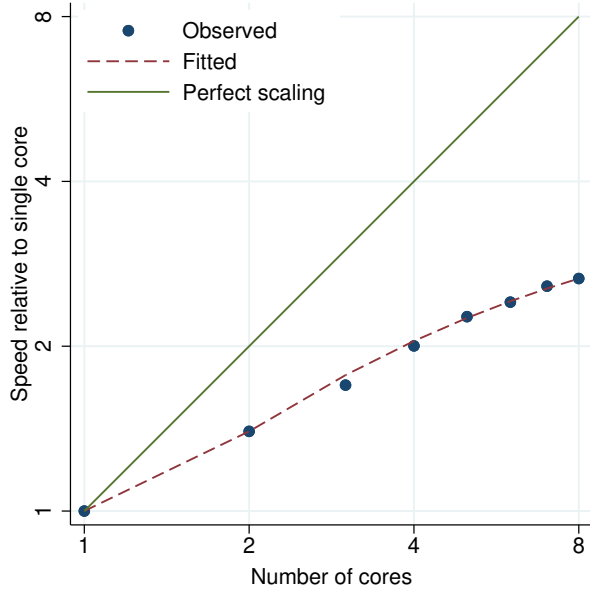


Figure 754. Random-effects logistic performance plot.

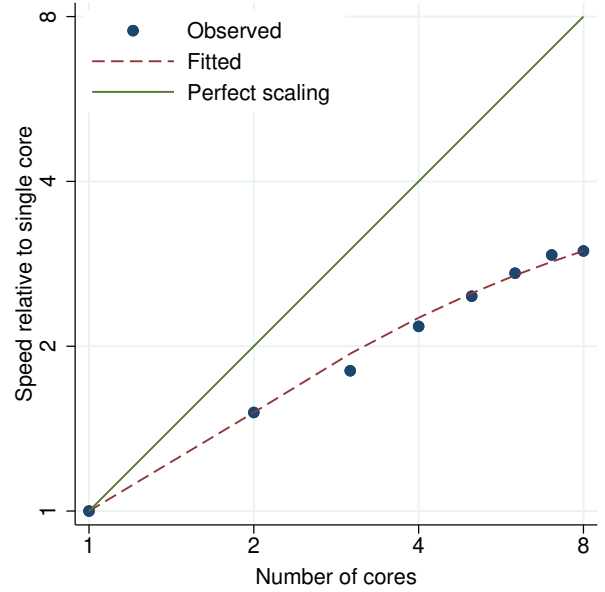


Figure 755. RE regression performance plot.

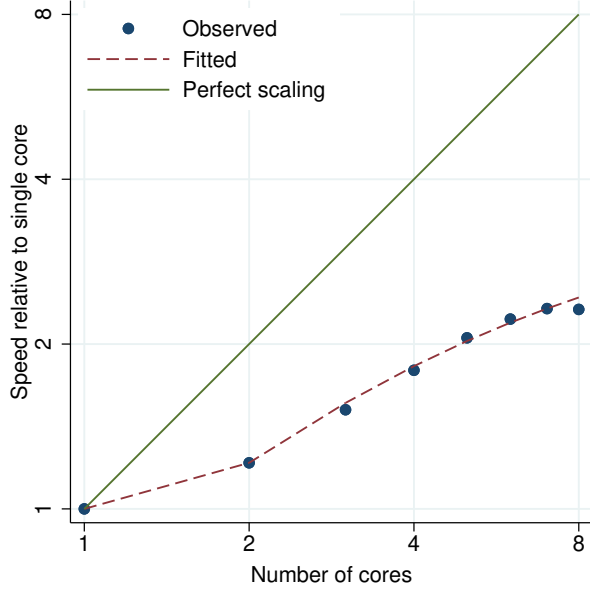


Figure 756. Two-level RE logistic performance plot.

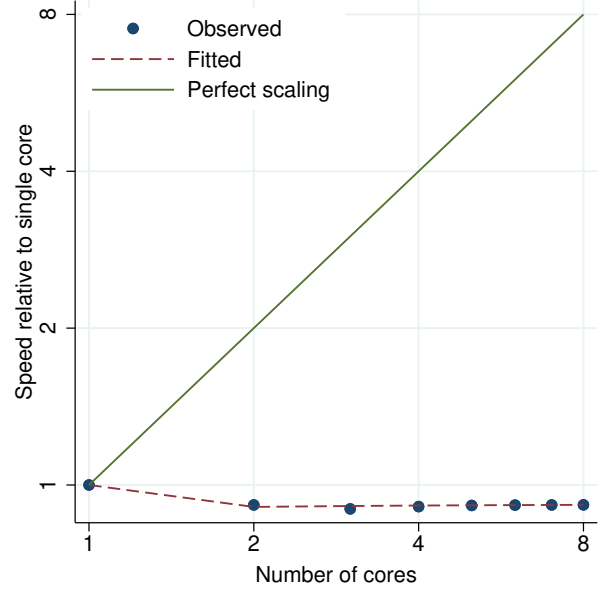


Figure 757. Random-coefficients Poisson performance plot.

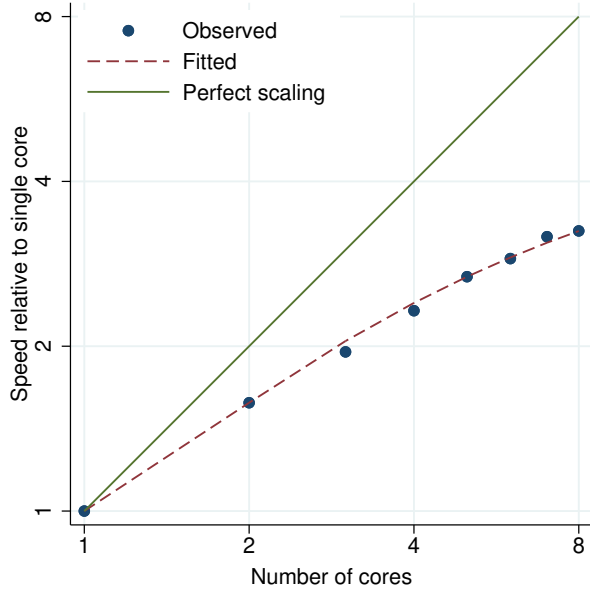


Figure 758. RE logistic with constant performance plot.

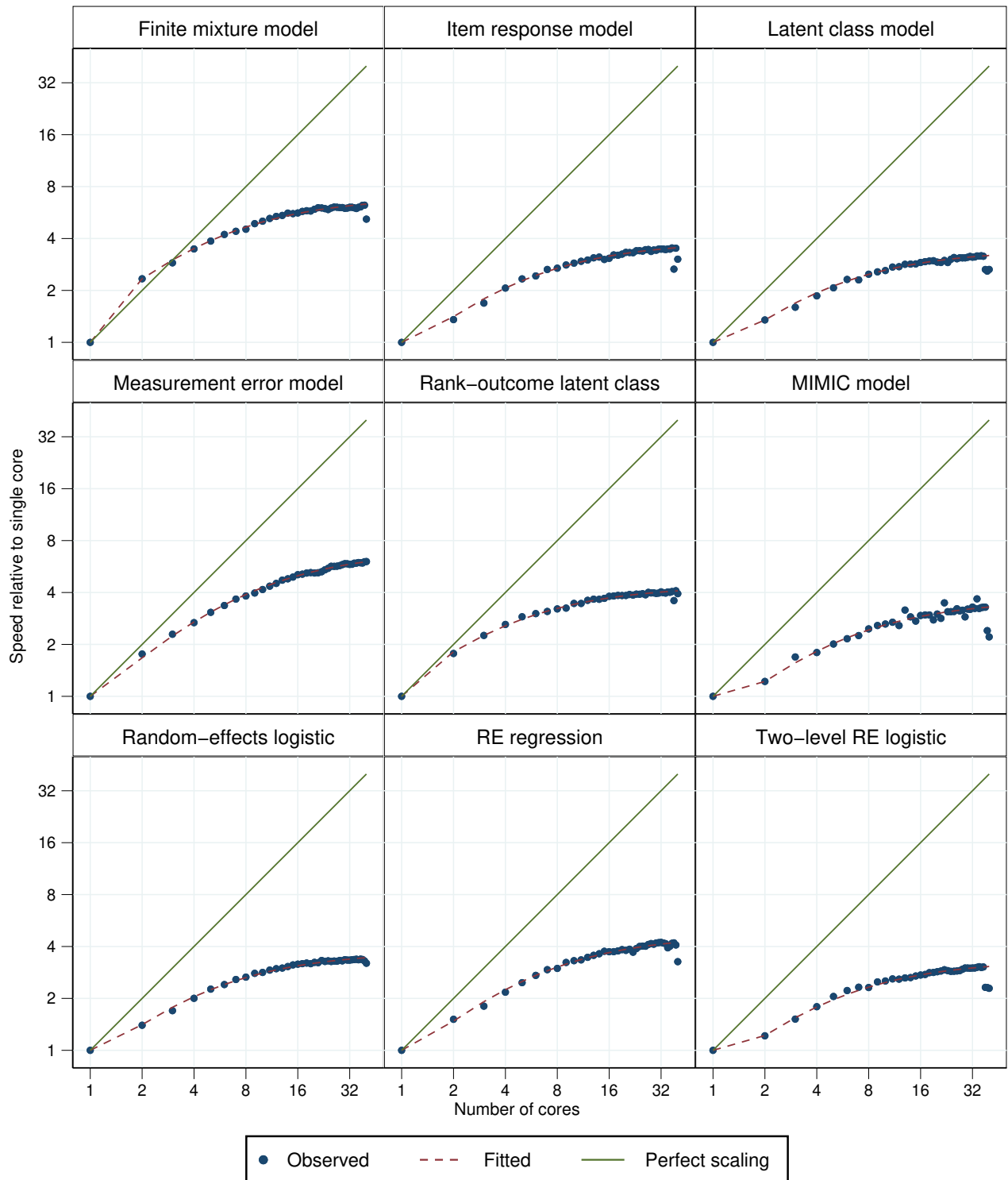


Figure 759. Parallelization performance plots.

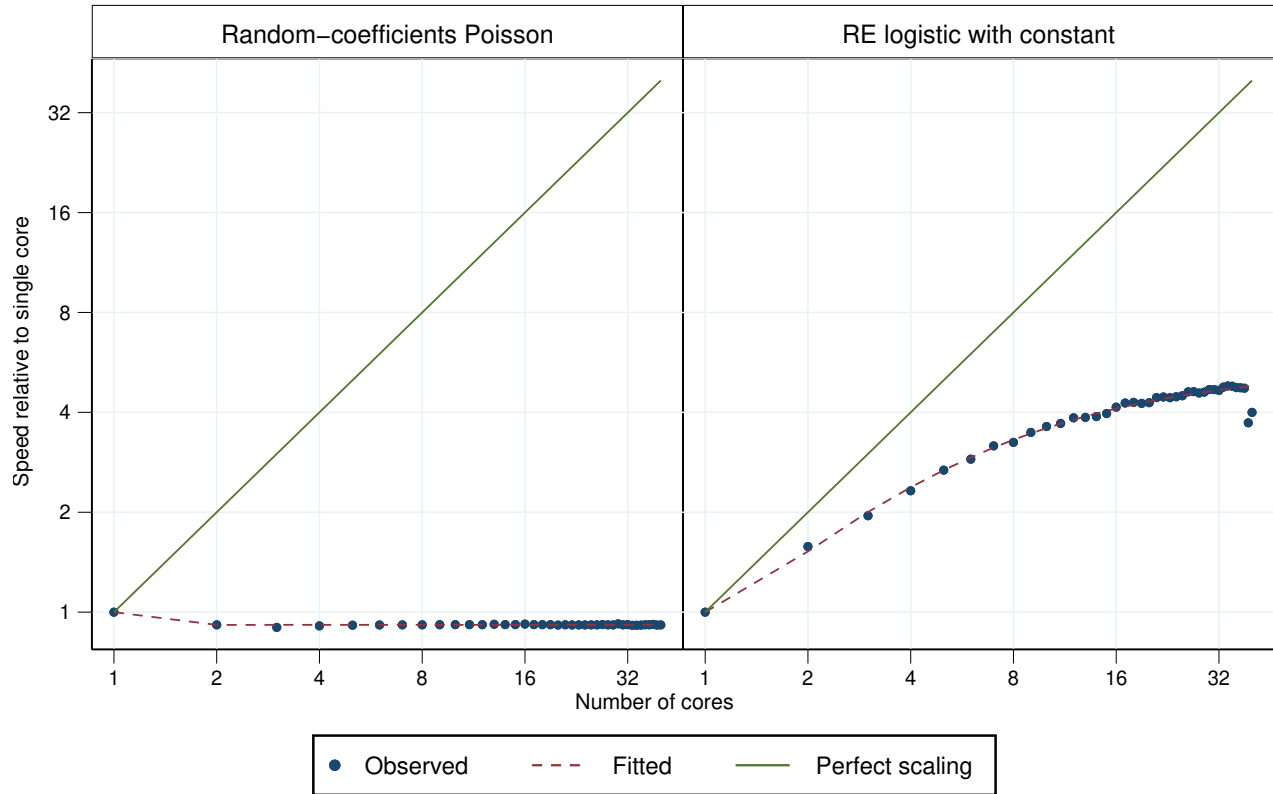


Figure 760. Parallelization performance plots.

References

- Culler, D. E., J. P. Singh, and A. Gupta. 1999. *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco: Morgan Kaufmann.
- Dagum, L., and R. Menon. 1998. OpenMP: An industry-standard API for shared memory programming. *IEEE Computational Science & Engineering* 5: 46–55.
- Grama, A., A. Gupta, G. Karypis, and V. Kumar. 2003. *Introduction to Parallel Computing*. 2nd ed. Boston: Addison–Wesley.
- Moore, G. E. 1965. Cramming more components onto integrated circuits. *Electronics* 38: 114–117.