## 1 Summary

Stata/MP[1] is the version of Stata that is programmed to take full advantage of multicore and multiprocessor computers. It is exactly like Stata/SE in all ways except that it 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 75% efficiency. It runs 1.7 times faster on 2 cores, 2.4 times faster on 4 cores, and 3.1 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 82%. Taken at the median, estimation commands run 1.8 times faster on 2 cores, 2.9 times faster on 4 cores, and 4.1 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.

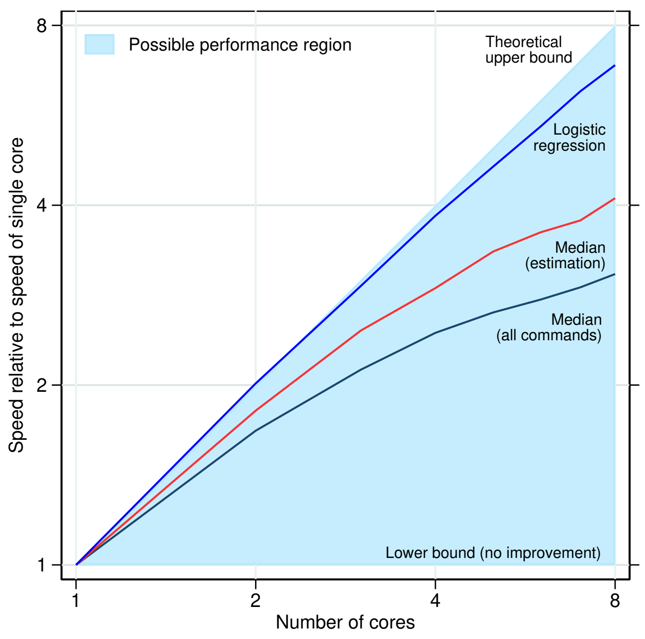


Figure 1. **Performance of Stata/MP.** Speed on multiple cores relative to speed on a single core.

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

## 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 75% efficiency overall and 82% 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 1.7 times faster than on a single core. With 4 cores, the same commands run at least 2.4 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 1.8 times faster on 2 cores and 2.9 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 3.8 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, 10 times faster on 16-core computers, 14 times faster on

32-core computers, and 18 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:

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

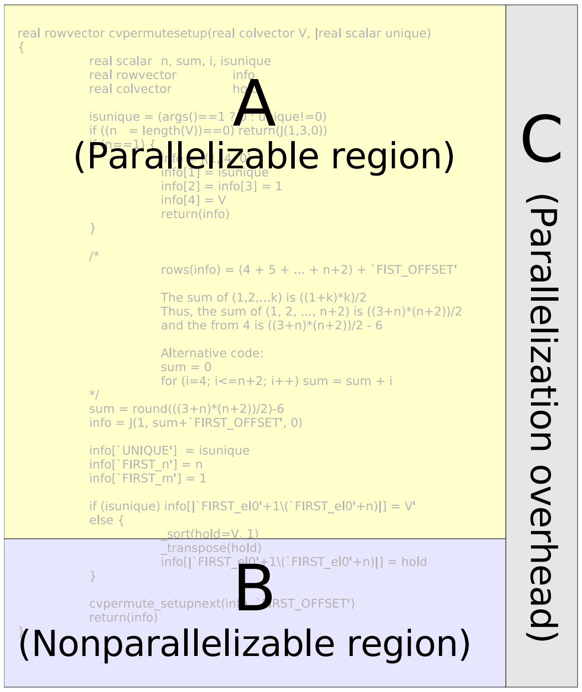
Statistical calculations are often more parallelizable than you might imagine. For instance, many inherently sequential computations can be parallelized when performed on longitudinal (panel) data because the dependencies that made the problem inherently sequential are broken at panel boundaries. Rather 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.



```
real rowvector cvpermutesetup(real colvector V, |real scalar unique)
{
        real scalar  n, sum, i, isunique
        real rowvector          info
        real colvector          hold

        isunique = (args()==1 ? 0 : unique!=0)
        if ((n   = length(V))==0) return(J(1,3,0))
        if (n==1) {
                info = J(1,4,0)
                info[1] = isunique
                info[2] = info[3] = 1
                info[4] = V
                return(info)
        }

        /*
                rows(info) = (4 + 5 + ... + n+2) + `FIST_OFFSET'

                The sum of (1,2,...k) is ((1+k)*k)/2
                Thus, the sum of (1, 2, ..., n+2) is ((3+n)*(n+2))/2
                and the from 4 is ((3+n)*(n+2))/2 - 6

                Alternative code:
                sum = 0
                for (i=4; i<=n+2; i++) sum = sum + i
        */
        sum = round(((3+n)*(n+2))/2)-6
        info = J(1, sum+`FIRST_OFFSET', 0)

        info[`UNIQUE']  = isunique
        info[`FIRST_n'] = n
        info[`FIRST_m'] = 1

        if (isunique) info[|`FIRST_el0'+1\(`FIRST_el0'+n)|] = V'
        else {
                _sort(hold=V, 1)
                _transpose(hold)
                info[|`FIRST_el0'+1\(`FIRST_el0'+n)|] = hold
        }

        cvpermute_setupnext(info, `FIRST_OFFSET')
        return(info)
}
```

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 614 Stata commands. Excluding I/O commands, these 614 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 614 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 614 commands. Figure 3 shows the graph of Stata's logistic regression command, `logistic`:

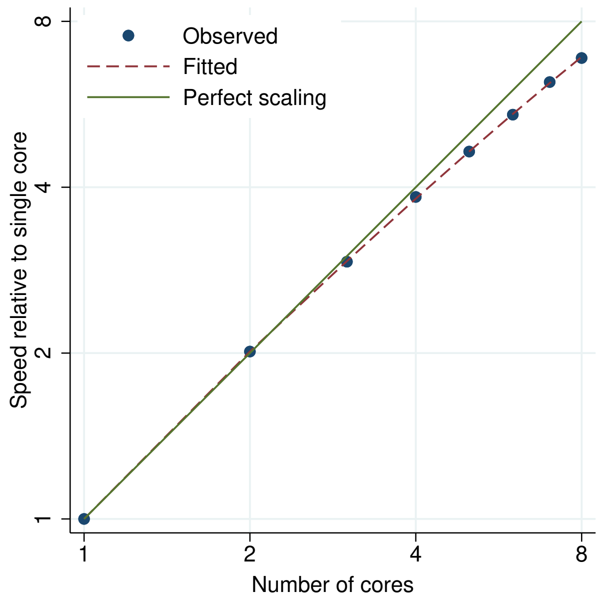


Figure 3. `logistic` performance plot.

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 6.9 (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.

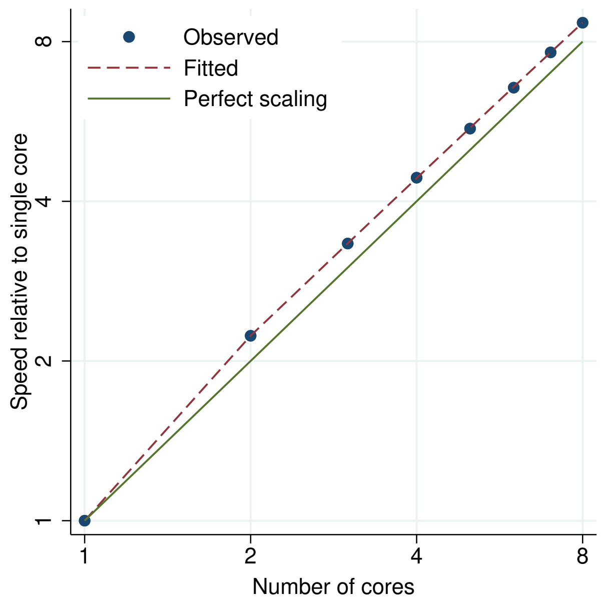


Figure 4. `regress` performance plot.

Figure 5 shows the graph for `arima`:

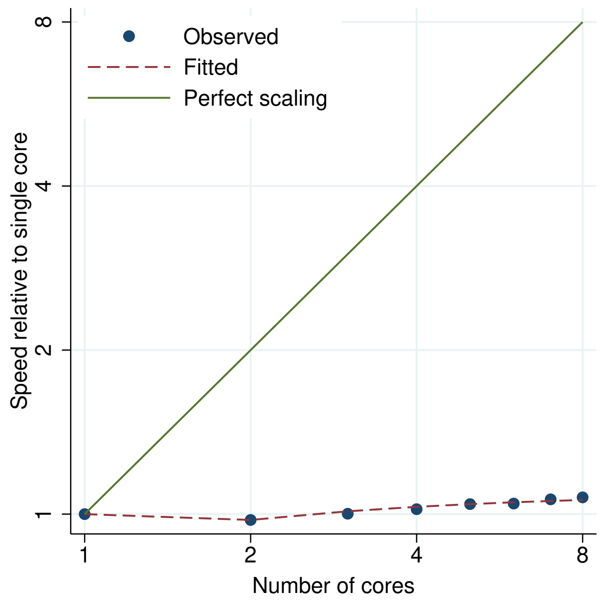


Figure 5. `arima` performance plot.

`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`:

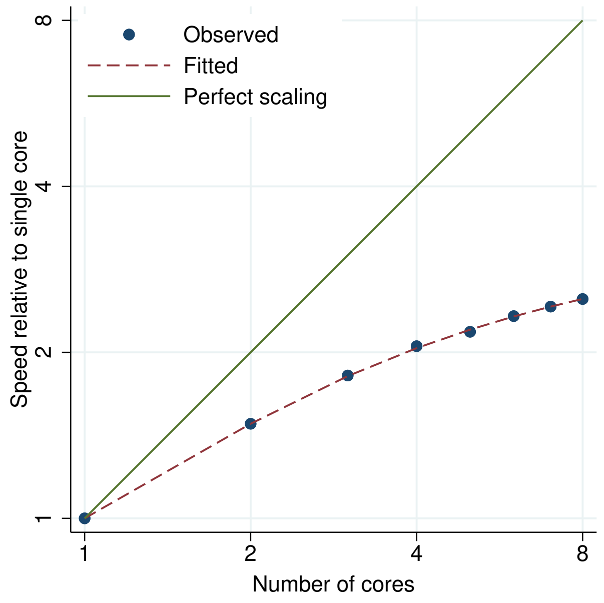


Figure 6. `etpoisson` performance plot.

Relative speeds are 1.5 (2 cores), 2.1 (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 614 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.

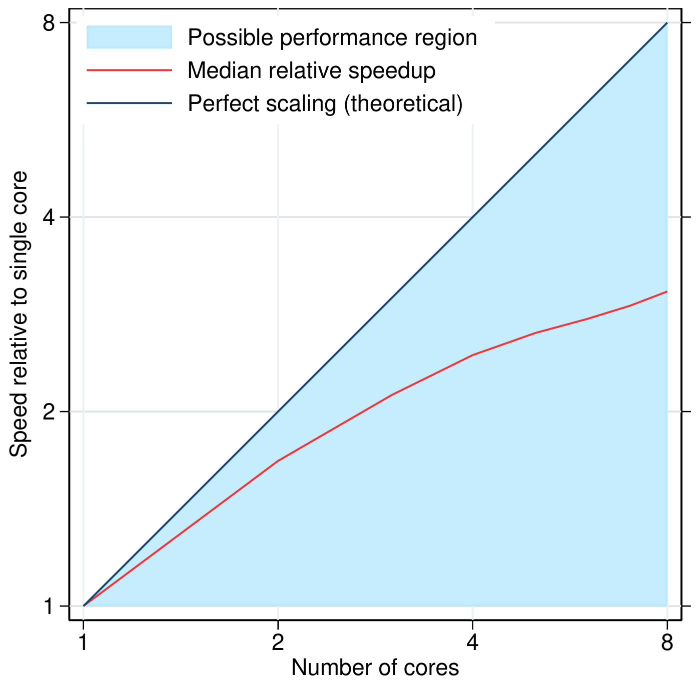


Figure 7. **Performance of Stata/MP.** Speed on multiple cores relative to speed on a single core.

Also included are the median results over all 614 commands; 307 commands have better performance gains (their curves lie above the median relative speedup line), and 307 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, 3.7 times faster on 4 cores, and 6.3 times faster on 8 cores.

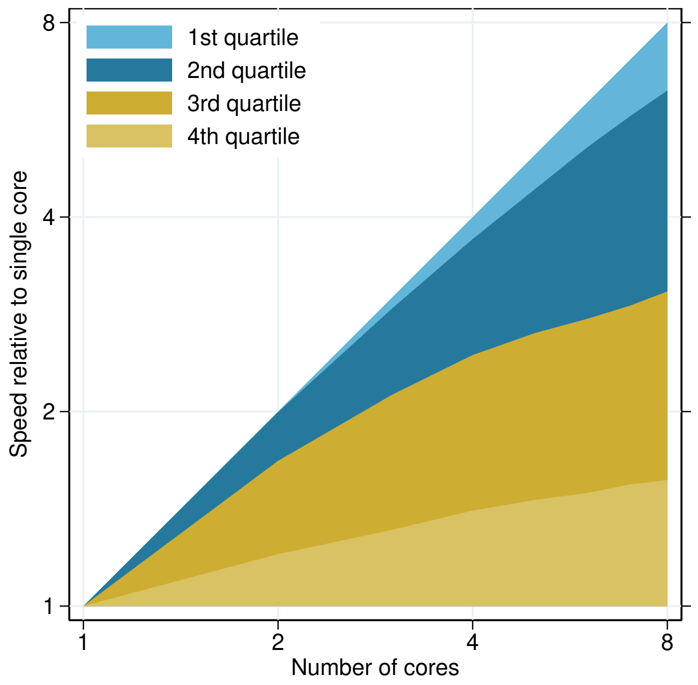


Figure 8. **Quartiles of Stata/MP performance**. Speed on multiple cores relative to speed on a single core.

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 349 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 1.8 times faster on 2 cores, 2.9 times faster on 4 cores, and 4.1 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 1.5 times faster on 2 cores, less than 2 times faster on 4 cores, and less than 2.3 times faster on 8 cores.

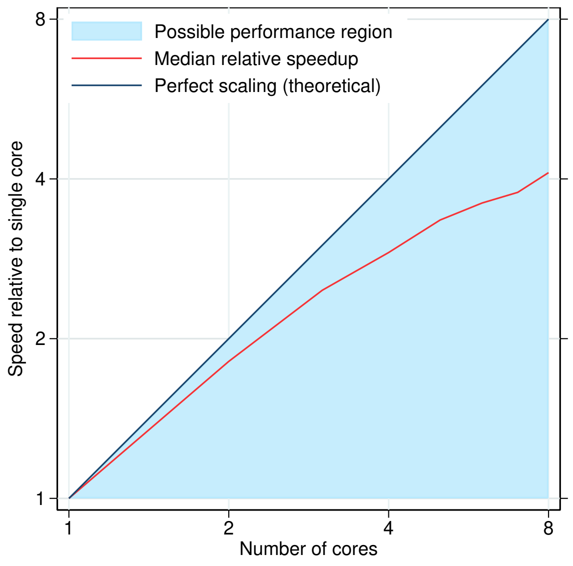


Figure 9. **Performance of Stata/MP on estimation commands.** Speed on multiple cores relative to speed on a single core.

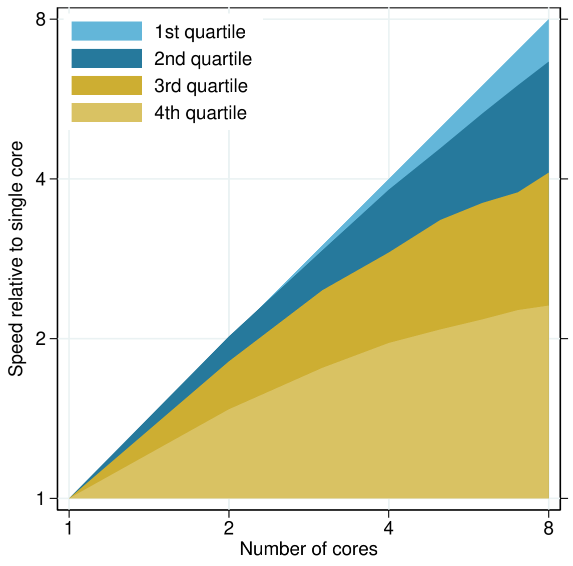


Figure 10. **Quartiles of Stata/MP performance on estimation commands**. Speed on multiple cores relative to speed on a single core.

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 349 estimation commands on 1–40-core computers, and figure 12 shows their performance quartiles.

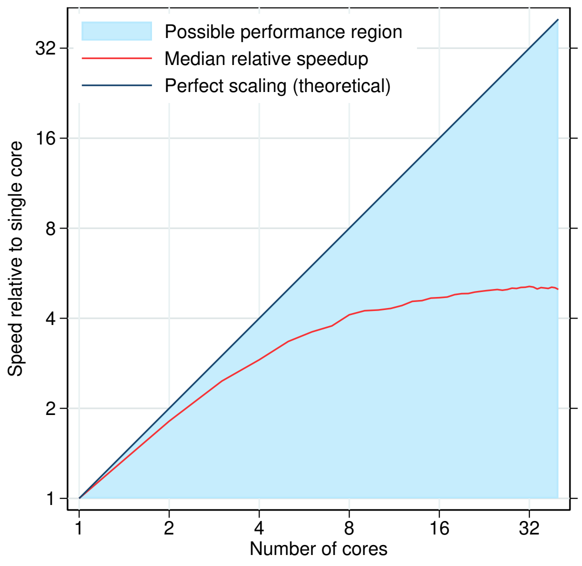


Figure 11. **Performance of Stata/MP on estimation commands (1 to 40 cores).** Speed of estimation commands on multiple cores relative to speed on a single core.

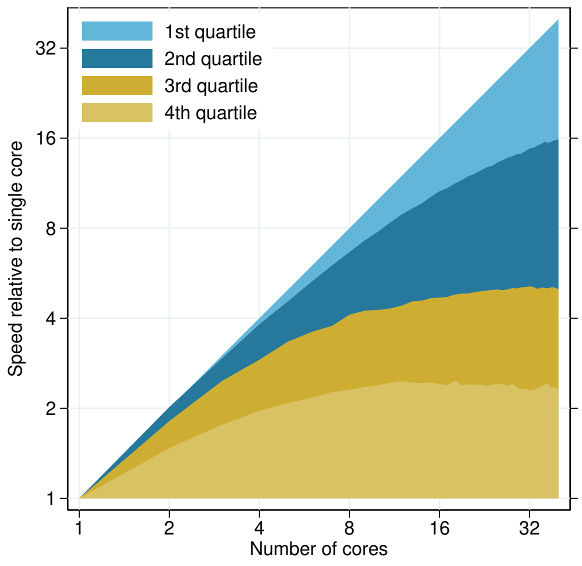


Figure 12. **Quartiles of Stata/MP performance on estimation commands (1 to 40 cores).** Speed of estimation commands on multiple cores relative to speed on a single core.

## 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:

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

where $c$ is the number of cores, $\delta_1$ is an indicator for $c > 1$, and $\alpha$ is an intercept.

Our parameters of interest are directly estimated:

$$\text{percentage parallelized} = \widehat{PP} \tag{2}$$

and

$$\text{parallelization overhead} = \widehat{O} \tag{3}$$

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

The estimated value for parallelization overhead is particularly sensitive to the computing platform, and so we do not report it here. Note from equation (1) that $\widehat{O}$ 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 | 1.7 | 2.8 | 4.2 | 5.5 | 87 |
| ameans | 1.8 | 2.8 | 3.9 | 4.6 | 84 |
| anova (one-way) | 1.9 | 3.6 | 6.4 | 10.2 | 96 |
| anova (two-way) | 2.5 | 4.5 | 7.2 | 10.1 | 94 |
| arch | 0.8 | 1.0 | 1.1 | 1.1 | 11 |
| areg | 2.3 | 4.1 | 6.2 | 8.1 | 91 |
| areg, vce(cluster) | 1.9 | 3.5 | 5.4 | 7.5 | 92 |
| areg, vce(robust) | 2.1 | 3.7 | 5.8 | 8.3 | 93 |
| arfima | 1.1 | 1.1 | 1.1 | 1.1 | 6 |
| arima | 1.0 | 1.0 | 1.1 | 1.0 | 3 |
| bayes dsge | 0.9 | 0.9 | 0.9 | 0.9 | 0 |
| bayes dsgenl | 0.9 | 0.9 | 0.9 | 0.9 | 0 |
| bayes: logit | 2.1 | 3.9 | 6.8 | 10.6 | 96 |
| bayes: poisson | 1.7 | 2.5 | 3.4 | 4.1 | 82 |
| bayes: regress | 1.9 | 3.3 | 5.0 | 6.6 | 90 |
| bayes var | 0.9 | 0.9 | 0.9 | 0.9 | 0 |
| bayesmh logit | 1.3 | 1.4 | 1.5 | 1.5 | 32 |
| bayesmh mvn | 1.1 | 1.1 | 1.1 | 1.1 | 8 |
| bayesmh mylogit | 1.2 | 1.4 | 1.6 | 1.6 | 38 |
| bayesmh nl | 1.0 | 1.0 | 1.2 | 1.2 | 15 |
| bayesmh normal | 1.1 | 1.2 | 1.3 | 1.3 | 24 |
| bayesmh normal gibbs | 1.0 | 1.0 | 1.0 | 1.0 | 0 |
| bayesmh normal re | 1.1 | 1.2 | 1.2 | 1.2 | 14 |
| betareg, link(logit) | 2.0 | 3.8 | 7.0 | 11.8 | 97 |
| betareg, link(probit) | 2.0 | 3.8 | 6.9 | 11.7 | 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

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| binreg | 2.2 | 4.0 | 6.9 | 11.0 | 96 |
| biplot | 1.0 | 1.0 | 1.0 | 1.0 | 0 |
| biprobit | 1.9 | 3.6 | 6.7 | 11.6 | 98 |
| biprobit (seemingly unrelated) | 1.9 | 3.7 | 6.8 | 11.1 | 97 |
| bitest | 1.7 | 2.8 | 4.1 | 5.3 | 87 |
| blogit | 2.0 | 3.7 | 6.5 | 10.4 | 96 |
| boxcox | 2.1 | 4.0 | 6.9 | 11.0 | 96 |
| bprobit | 1.9 | 3.6 | 6.2 | 9.9 | 96 |
| brier | 1.3 | 1.4 | 1.6 | 1.7 | 43 |
| bsample | 1.0 | 1.4 | 1.5 | 1.5 | 37 |
| bstat | 1.8 | 3.1 | 4.4 | 5.6 | 87 |
| by: generate | 2.0 | 3.5 | 6.2 | 16.5 | 100 |
| by: generate (small groups) | 2.4 | 4.1 | 5.4 | 6.0 | 89 |
| by: replace | 2.7 | 5.3 | 10.6 | 21.0 | 100 |
| by: replace (small groups) | 8.0 | 13.3 | 17.9 | 26.8 | 98 |
| ca | 1.4 | 1.8 | 2.1 | 2.2 | 61 |
| candisc | 2.4 | 4.6 | 8.2 | 13.6 | 97 |
| canon | 1.7 | 2.9 | 4.6 | 7.0 | 92 |
| cc | 1.1 | 1.2 | 1.3 | 1.2 | 26 |
| by: cc | 1.0 | 1.0 | 1.0 | 1.0 | 2 |
| centile | 1.1 | 1.4 | 1.7 | 1.9 | 51 |
| churdle linear | 1.8 | 3.2 | 4.7 | 6.0 | 88 |
| ci means | 1.8 | 2.6 | 3.1 | 3.4 | 74 |
| ci means, poisson | 1.4 | 2.1 | 2.8 | 3.5 | 77 |
| ci proportions | 1.8 | 2.5 | 3.2 | 3.6 | 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

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| clogit (k1 to k2 matching) | 1.9 | 3.5 | 5.9 | 9.3 | 95 |
| clogit (1 to k matching) | 1.5 | 1.9 | 2.3 | 2.5 | 65 |
| cloglog | 1.9 | 3.7 | 6.5 | 10.7 | 97 |
| cluster averagelinkage | 1.9 | 3.7 | 6.8 | 12.7 | 99 |
| cluster centroidlinkage | 1.8 | 3.6 | 6.9 | 12.9 | 99 |
| cluster completelinkage | 1.8 | 3.6 | 6.7 | 12.2 | 99 |
| cluster generate | 0.8 | 0.8 | 0.8 | 0.8 | 0 |
| cluster kmeans | 2.3 | 4.6 | 8.5 | 15.6 | 99 |
| cluster kmedians | 2.3 | 4.6 | 8.7 | 16.4 | 99 |
| cluster medianlinkage | 1.9 | 3.7 | 7.0 | 12.7 | 99 |
| cluster singlelinkage | 1.0 | 1.0 | 1.0 | 1.0 | 0 |
| cluster wardslinkage | 1.8 | 3.4 | 6.3 | 11.3 | 98 |
| cluster waveragelinkage | 1.8 | 3.4 | 6.3 | 11.2 | 98 |
| cmclogit | 1.9 | 3.2 | 4.8 | 6.6 | 90 |
| cmmprobit | 1.3 | 1.5 | 1.8 | 1.9 | 47 |
| cmroprobit | 1.5 | 1.8 | 2.0 | 2.0 | 52 |
| cnsreg | 2.3 | 4.6 | 9.1 | 17.5 | 100 |
| codebook | 1.8 | 2.9 | 4.5 | 5.8 | 89 |
| collapse | 3.3 | 5.0 | 6.4 | 7.2 | 88 |
| compare | 1.8 | 2.9 | 4.1 | 5.2 | 86 |
| compress | 0.9 | 1.0 | 0.9 | 1.1 | 6 |
| contract | 1.1 | 1.2 | 1.2 | 1.2 | 21 |
| corr2data | 1.9 | 3.5 | 5.3 | 6.9 | 90 |
| correlate | 2.3 | 4.6 | 9.1 | 17.9 | 100 |
| corrgram | 1.2 | 1.5 | 1.5 | 1.4 | 36 |

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

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| count | 2.0 | 3.9 | 7.8 | 15.6 | 100 |
| cpoisson | 1.6 | 2.2 | 2.7 | 3.0 | 71 |
| cs | 1.1 | 1.2 | 1.3 | 1.3 | 24 |
| by: cs | 1.0 | 1.0 | 1.0 | 1.0 | 0 |
| ctset | 2.0 | 4.0 | 8.0 | 16.0 | 100 |
| cttost | 1.0 | 1.0 | 1.0 | 1.0 | 2 |
| cumul | 1.1 | 1.3 | 1.5 | 1.7 | 40 |
| cusum | 1.3 | 1.6 | 1.8 | 2.0 | 52 |
| datasignature | 1.0 | 1.0 | 1.0 | 1.0 | 2 |
| decode | 1.0 | 1.0 | 1.0 | 1.0 | 0 |
| destring | 1.0 | 1.0 | 1.1 | 1.2 | 23 |
| dfactor | 1.6 | 2.0 | 2.5 | 2.5 | 59 |
| dfgls | 1.2 | 1.3 | 1.4 | 1.4 | 31 |
| dfuller | 2.0 | 2.6 | 3.1 | 3.3 | 72 |
| didregress | 1.3 | 1.6 | 1.9 | 2.0 | 54 |
| discrim knn | 1.5 | 2.2 | 2.6 | 2.9 | 68 |
| discrim lda | 2.2 | 4.0 | 6.7 | 9.8 | 95 |
| discrim logistic | 1.8 | 3.5 | 6.6 | 11.8 | 98 |
| discrim qda | 1.6 | 2.3 | 2.8 | 3.2 | 73 |
| dotplot | 1.1 | 1.4 | 1.5 | 1.6 | 41 |
| drawnorm | 2.0 | 3.6 | 6.1 | 9.2 | 95 |
| drop if *exp* | 1.0 | 1.0 | 1.0 | 1.0 | 0 |
| drop in *range* | 1.0 | 1.0 | 1.0 | 1.0 | 0 |
| dsge | 0.9 | 0.9 | 0.9 | 0.9 | 0 |
| dsgenl | 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 | |
| dslogit | 1.4 | 1.8 | 2.0 | 2.2 | 58 |
| dspoisson | 1.4 | 1.8 | 2.1 | 2.2 | 59 |
| dsregress | 1.3 | 1.8 | 2.0 | 2.0 | 55 |
| dstdize | 1.0 | 1.0 | 1.0 | 1.0 | 0 |
| dvech | 1.0 | 1.0 | 1.0 | 1.0 | 0 |
| egen group() | 1.8 | 3.0 | 4.1 | 4.8 | 84 |
| by: egen mean | 1.1 | 1.2 | 1.4 | 2.4 | 63 |
| eivreg | 2.3 | 4.5 | 8.7 | 16.2 | 99 |
| encode | 1.5 | 2.6 | 3.4 | 3.6 | 79 |
| eregress | 1.8 | 2.5 | 3.2 | 3.6 | 76 |
| esize twosample | 1.4 | 1.7 | 2.0 | 2.2 | 59 |
| esize unpaired | 1.6 | 2.3 | 3.0 | 3.5 | 76 |
| eteffects (exponential), ate | 1.9 | 1.6 | 1.9 | 4.8 | 82 |
| eteffects (linear), ate | 2.0 | 3.5 | 5.7 | 7.7 | 92 |
| eteffects (linear), pomeans | 2.1 | 3.6 | 5.7 | 7.9 | 92 |
| eteffects (probit), ate | 2.0 | 3.5 | 5.6 | 7.5 | 91 |
| etpoisson | 1.5 | 2.1 | 2.5 | 2.5 | 62 |
| etregress, poutcomes | 1.8 | 2.9 | 4.3 | 4.7 | 82 |
| etregress, twostep | 2.0 | 3.8 | 7.0 | 12.0 | 97 |
| exlogistic | 1.0 | 1.0 | 1.0 | 1.0 | 4 |
| expand # | 0.9 | 0.9 | 0.9 | 0.9 | 0 |
| expand *varname* | 0.9 | 0.9 | 0.9 | 0.9 | 0 |
| expandcl # | 1.3 | 1.7 | 2.1 | 2.3 | 62 |
| expandcl *varname* | 1.3 | 1.6 | 1.9 | 2.2 | 60 |
| expoisson | 1.3 | 1.3 | 1.3 | 1.3 | 23 |

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

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| factor | 1.4 | 2.0 | 2.3 | 2.5 | 63 |
| fcast compute | 1.0 | 1.0 | 1.0 | 1.0 | 1 |
| fillin | 1.1 | 1.3 | 1.6 | 1.8 | 48 |
| fmm 2: poisson | 1.7 | 2.4 | 2.9 | 3.2 | 70 |
| fmm 2: regress | 1.5 | 2.0 | 2.2 | 2.3 | 55 |
| fmm 3: poisson | 1.5 | 2.0 | 2.3 | 2.5 | 62 |
| fmm 3: regress | 1.5 | 1.9 | 2.1 | 2.2 | 53 |
| fracreg probit | 2.1 | 4.1 | 7.8 | 14.2 | 99 |
| frontier | 2.1 | 4.1 | 7.7 | 13.6 | 99 |
| fvrevar (factors) | 1.7 | 4.0 | 5.7 | 7.0 | 94 |
| fvrevar (interaction) | 1.8 | 3.2 | 4.9 | 6.0 | 79 |
| generate (small expressions) | 3.3 | 6.3 | 11.3 | 19.1 | 98 |
| generate | 2.0 | 4.0 | 8.0 | 15.8 | 100 |
| glm, family(gamma) | 2.0 | 3.8 | 6.6 | 10.6 | 96 |
| glm, family(gaussian) | 2.1 | 3.9 | 7.1 | 11.6 | 97 |
| glm, family(igaussian) | 2.1 | 3.9 | 7.2 | 11.9 | 97 |
| glm, family(nbinomial) | 2.1 | 4.0 | 7.4 | 12.4 | 98 |
| glm, family(poisson) | 2.0 | 3.9 | 7.1 | 12.1 | 98 |
| glogit | 2.2 | 4.3 | 8.2 | 15.3 | 99 |
| gmm | 1.1 | 1.1 | 1.2 | 1.2 | 13 |
| gmm (with derivatives) | 2.1 | 3.6 | 5.0 | 6.6 | 90 |
| gprobit | 2.2 | 4.3 | 8.1 | 15.0 | 99 |
| graph bar | 1.1 | 1.2 | 1.2 | 1.2 | 20 |
| graph box | 1.0 | 1.1 | 1.1 | 1.1 | 11 |
| graph pie | 1.2 | 1.3 | 1.3 | 1.4 | 28 |

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

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| grmeanby | 1.2 | 1.5 | 1.6 | 2.6 | 68 |
| gsem, oprobit (CFA, 2-level) | 2.3 | 3.3 | 4.2 | 4.1 | 75 |
| gsem, oprobit (CFA) | 2.5 | 3.5 | 3.8 | 3.7 | 71 |
| gsem, logit group() | 1.8 | 2.6 | 3.4 | 3.6 | 78 |
| gsem, group() | 1.7 | 2.6 | 3.5 | 4.2 | 80 |
| gsem, ologit group() | 1.7 | 2.5 | 3.2 | 3.7 | 77 |
| gsem, poisson group() | 1.7 | 2.5 | 3.3 | 3.8 | 78 |
| gsort | 1.0 | 1.1 | 1.1 | 1.1 | 10 |
| hausman | 1.2 | 1.2 | 1.3 | 1.2 | 21 |
| heckman | 1.9 | 3.8 | 6.6 | 10.9 | 97 |
| heckman, twostep | 2.0 | 3.9 | 7.0 | 11.4 | 97 |
| heckoprobit | 2.0 | 3.6 | 6.4 | 10.0 | 96 |
| heckpoisson | 1.8 | 2.8 | 4.1 | 4.4 | 79 |
| heckprob | 1.9 | 3.7 | 6.6 | 10.5 | 96 |
| hetoprobit | 1.9 | 3.3 | 5.0 | 6.7 | 90 |
| hetprob | 1.6 | 3.4 | 5.9 | 8.6 | 94 |
| hetregress | 1.9 | 3.6 | 6.1 | 9.5 | 95 |
| hetregress, twostep | 2.1 | 4.0 | 7.3 | 12.2 | 97 |
| histogram | 1.3 | 1.6 | 1.9 | 2.0 | 52 |
| hotelling | 2.1 | 4.3 | 8.3 | 15.9 | 99 |
| icc, mixed | 1.2 | 1.4 | 1.6 | 1.7 | 44 |
| icc (one-way) | 1.3 | 1.7 | 1.9 | 2.0 | 53 |
| icc (two-way) | 1.0 | 1.3 | 1.4 | 1.7 | 46 |
| import delimited | 1.3 | 1.9 | 1.7 | 1.7 | 45 |
| intreg | 2.1 | 4.2 | 7.8 | 14.0 | 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

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| ir | 1.1 | 1.2 | 1.3 | 1.3 | 27 |
| by: ir | 1.0 | 1.0 | 1.0 | 1.0 | 0 |
| irf create | 1.3 | 1.5 | 1.6 | 1.9 | 50 |
| irt 1pl | 1.5 | 2.3 | 2.7 | 2.6 | 57 |
| irt 2pl | 1.6 | 2.3 | 2.7 | 2.5 | 58 |
| irt 3pl | 2.1 | 2.4 | 2.6 | 2.4 | 57 |
| irt grm | 1.6 | 2.3 | 2.7 | 2.6 | 61 |
| irt nrm | 1.5 | 2.1 | 2.4 | 2.3 | 54 |
| irt pcm | 1.5 | 2.1 | 2.4 | 2.4 | 55 |
| irt rsm | 1.5 | 2.1 | 2.4 | 2.2 | 50 |
| istdize | 1.0 | 1.0 | 1.0 | 1.0 | 0 |
| ivpoisson cfunction | 2.4 | 4.4 | 6.2 | 7.5 | 90 |
| ivpoisson gmm, additive | 2.7 | 4.8 | 8.1 | 10.0 | 92 |
| ivpoisson gmm, multiplicative | 1.8 | 2.9 | 3.7 | 4.6 | 83 |
| ivprobit | 1.9 | 3.3 | 5.0 | 6.8 | 90 |
| ivregress 2sls | 2.1 | 4.2 | 7.3 | 11.9 | 97 |
| ivregress gmm | 2.0 | 3.4 | 5.1 | 6.8 | 90 |
| ivregress liml | 2.0 | 3.8 | 6.5 | 9.3 | 95 |
| ivtobit | 1.7 | 2.3 | 2.7 | 3.0 | 71 |
| kap | 1.1 | 1.3 | 1.4 | 1.8 | 49 |
| kappa | 2.0 | 3.6 | 6.1 | 8.9 | 93 |
| kdensity | 1.9 | 3.3 | 4.0 | 4.5 | 81 |
| keep if *exp* | 1.0 | 1.0 | 1.0 | 1.0 | 1 |
| keep in *range* | 1.0 | 1.0 | 1.0 | 1.0 | 0 |
| keep *varlist* | 1.0 | 1.0 | 1.0 | 1.0 | 1 |

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

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| ksmirnov | 1.4 | 1.8 | 2.2 | 2.5 | 65 |
| ksmirnov, by() | 2.2 | 2.6 | 3.0 | 3.2 | 73 |
| ktau | 1.0 | 1.0 | 1.0 | 1.0 | 0 |
| kwallis | 2.4 | 2.9 | 3.5 | 3.8 | 76 |
| ladder | 1.8 | 2.8 | 4.1 | 5.1 | 85 |
| lasso linear | 1.3 | 1.5 | 1.6 | 1.7 | 43 |
| lasso logit | 1.1 | 1.2 | 1.3 | 1.3 | 23 |
| lasso poisson | 1.1 | 1.2 | 1.3 | 1.4 | 21 |
| gsem, lclass(C 2) | 1.5 | 2.3 | 2.9 | 3.2 | 73 |
| gsem, lclass(C 3) | 1.5 | 2.2 | 2.7 | 3.1 | 71 |
| levelsof | 1.1 | 1.1 | 1.1 | 1.1 | 13 |
| loadingplot | 1.0 | 1.0 | 1.1 | 1.0 | 7 |
| logistic | 2.0 | 3.8 | 6.9 | 11.6 | 97 |
| logit | 2.0 | 3.8 | 6.9 | 11.4 | 97 |
| loneway | 1.2 | 1.4 | 1.5 | 1.6 | 39 |
| lowess | 2.0 | 3.5 | 6.9 | 13.8 | 100 |
| lpoly | 1.7 | 2.6 | 3.7 | 4.5 | 83 |
| ltable | 0.7 | 0.7 | 0.7 | 0.7 | 0 |
| manova (one-way) | 1.8 | 2.7 | 3.7 | 4.6 | 83 |
| manova (two-way) | 1.4 | 2.0 | 2.7 | 3.6 | 81 |
| margins | 1.9 | 3.4 | 6.1 | 11.2 | 98 |
| margins, dydx() exp() | 1.8 | 3.3 | 5.3 | 7.9 | 93 |
| margins, dydx() | 1.8 | 3.1 | 5.1 | 7.3 | 91 |
| margins, exp() | 1.8 | 3.2 | 5.5 | 8.4 | 94 |
| markout | 2.1 | 4.0 | 7.7 | 14.0 | 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

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| marksample | 2.0 | 3.9 | 8.2 | 16.2 | 99 |
| marksample if *exp* | 2.0 | 4.0 | 8.2 | 16.3 | 99 |
| matrix accum | 2.3 | 4.6 | 9.1 | 18.2 | 100 |
| matrix eigenvalues | 1.0 | 1.0 | 1.0 | 1.0 | 1 |
| matrix score | 2.1 | 4.2 | 8.5 | 16.3 | 100 |
| matrix svd | 1.0 | 1.0 | 1.0 | 1.0 | 1 |
| matrix symeigen | 1.0 | 1.0 | 1.0 | 1.0 | 0 |
| matrix syminv | 1.2 | 1.9 | 3.2 | 5.8 | 96 |
| mca | 1.0 | 1.0 | 1.0 | 1.1 | 6 |
| mcc | 1.1 | 1.1 | 1.2 | 1.2 | 18 |
| mds | 2.2 | 2.7 | 3.0 | 3.1 | 70 |
| mdslong | 1.8 | 2.1 | 2.3 | 2.4 | 60 |
| mean | 2.0 | 3.7 | 6.7 | 11.3 | 97 |
| mecloglog | 1.6 | 2.3 | 3.0 | 3.4 | 72 |
| median | 1.7 | 3.0 | 4.0 | 4.8 | 84 |
| meintreg | 1.8 | 2.8 | 4.0 | 4.9 | 84 |
| melogit | 1.7 | 2.5 | 3.1 | 3.6 | 75 |
| menbreg, dispersion(constant) | 1.6 | 1.8 | 2.0 | 2.0 | 46 |
| menbreg, dispersion(mean) | 1.6 | 1.9 | 2.3 | 2.5 | 63 |
| menl | 1.0 | 1.0 | 0.9 | 0.9 | 0 |
| meologit | 1.7 | 2.6 | 3.3 | 3.9 | 77 |
| meoprobit | 1.7 | 2.7 | 3.5 | 4.0 | 78 |
| mepoisson | 1.5 | 2.1 | 2.5 | 2.7 | 64 |
| meprobit | 1.7 | 2.6 | 3.4 | 3.9 | 78 |
| mestreg, distribution(exp) | 1.6 | 2.4 | 2.9 | 3.2 | 71 |

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

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| mestreg, distribution(weibull) | 1.7 | 2.6 | 3.4 | 4.0 | 79 |
| metobit | 1.7 | 2.6 | 3.4 | 4.0 | 78 |
| mgarch | 1.0 | 1.0 | 1.0 | 1.0 | 1 |
| mhodds | 1.0 | 1.3 | 1.4 | 1.5 | 34 |
| mhodds (adjusted) | 1.6 | 2.4 | 3.0 | 3.5 | 76 |
| by: mhodds | 1.0 | 1.0 | 1.0 | 1.0 | 6 |
| mhodds (trend) | 1.4 | 1.7 | 1.9 | 2.0 | 51 |
| mi estimate: logit (flong) | 1.4 | 1.9 | 2.0 | 2.4 | 62 |
| mi estimate: logit (flongsep) | 2.1 | 3.6 | 5.5 | 7.4 | 91 |
| mi estimate: logit (mlong) | 1.7 | 2.4 | 3.2 | 4.0 | 79 |
| mi estimate: logit (wide) | 1.9 | 3.3 | 5.3 | 7.5 | 92 |
| mi estimate: mlogit | 1.9 | 3.6 | 6.2 | 10.0 | 96 |
| mi estimate: ologit | 1.9 | 3.5 | 5.9 | 8.9 | 95 |
| mi estimate: regress (flong) | 1.3 | 1.3 | 1.4 | 1.5 | 32 |
| mi estimate: regress (flongsep) | 1.9 | 2.9 | 3.9 | 4.7 | 83 |
| mi estimate: regress (mlong) | 1.5 | 1.8 | 2.0 | 2.3 | 59 |
| mi estimate: regress (wide) | 1.9 | 3.0 | 4.5 | 5.5 | 86 |
| mi impute chained (flong) | 1.2 | 1.4 | 1.5 | 1.6 | 39 |
| mi impute chained (flongsep) | 1.2 | 1.4 | 1.6 | 1.7 | 45 |
| mi impute chained (mlong) | 1.1 | 1.4 | 1.6 | 1.7 | 44 |
| mi impute chained (wide) | 1.4 | 1.6 | 1.9 | 2.0 | 54 |
| mi impute logit (flong) | 1.2 | 1.2 | 1.4 | 1.2 | 25 |
| mi impute logit (flongsep) | 1.4 | 1.9 | 2.1 | 2.2 | 58 |
| mi impute logit (mlong) | 1.3 | 1.4 | 1.6 | 1.6 | 40 |
| mi impute logit (wide) | 1.8 | 2.7 | 3.7 | 4.4 | 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 1. Stata/MP performance, command by command

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| mi impute mlogit | 1.5 | 2.1 | 2.6 | 2.9 | 69 |
| mi impute mono pmm | 1.5 | 2.3 | 2.7 | 2.8 | 65 |
| mi impute mono regress | 1.6 | 2.2 | 2.7 | 3.2 | 72 |
| mi impute mvn | 1.1 | 1.1 | 1.1 | 1.2 | 14 |
| mi impute ologit | 1.4 | 1.7 | 2.0 | 2.1 | 54 |
| mi impute pmm | 1.5 | 2.2 | 2.8 | 3.1 | 70 |
| mi impute regress | 1.3 | 1.6 | 1.7 | 1.8 | 46 |
| misstable nested | 1.4 | 1.7 | 1.9 | 2.1 | 55 |
| misstable patterns | 1.2 | 1.5 | 1.6 | 1.7 | 43 |
| misstable summarize | 1.0 | 1.0 | 1.0 | 1.0 | 0 |
| misstable tree | 1.3 | 1.7 | 1.9 | 2.0 | 54 |
| mixed | 1.1 | 1.1 | 1.1 | 1.1 | 9 |
| mixed (crossed effects) | 1.2 | 1.2 | 1.2 | 1.2 | 13 |
| mkspline | 1.8 | 2.7 | 3.5 | 4.1 | 79 |
| mleval | 2.0 | 4.0 | 8.1 | 16.1 | 100 |
| mleval, nocons | 2.0 | 4.0 | 8.1 | 16.2 | 100 |
| mlmatbysum | 1.9 | 3.5 | 6.5 | 11.6 | 98 |
| mlmatsum | 2.0 | 3.9 | 7.7 | 15.1 | 100 |
| mlogit | 2.0 | 4.0 | 7.8 | 14.8 | 99 |
| mlsum | 1.7 | 3.0 | 4.8 | 7.0 | 92 |
| mlvecsum | 1.9 | 3.9 | 7.4 | 13.9 | 99 |
| mprobit | 1.4 | 1.4 | 1.4 | 1.4 | 29 |
| mswitch ar | 1.1 | 1.2 | 1.2 | 1.2 | 17 |
| mswitch dr | 0.9 | 0.9 | 0.9 | 0.9 | 0 |
| mvdecode | 5.5 | 11.5 | 23.0 | 45.5 | 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

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| mvencode | 2.0 | 4.1 | 8.1 | 16.0 | 100 |
| mvreg | 2.1 | 4.0 | 7.3 | 11.6 | 97 |
| mvtest correlations | 1.9 | 3.3 | 4.8 | 7.2 | 91 |
| mvtest covariances | 1.9 | 3.2 | 5.3 | 7.3 | 92 |
| mvtest means, heterogeneous | 1.5 | 2.1 | 2.5 | 2.9 | 69 |
| mvtest means, homogeneous | 1.9 | 3.0 | 3.7 | 4.5 | 81 |
| mvtest means, lr | 1.5 | 2.1 | 2.4 | 2.8 | 68 |
| mvtest normality | 0.9 | 0.9 | 0.9 | 0.9 | 0 |
| nbreg | 1.9 | 3.7 | 6.4 | 10.1 | 96 |
| newey | 1.2 | 1.3 | 1.4 | 1.4 | 32 |
| nl | 1.7 | 3.2 | 4.6 | 7.1 | 90 |
| nlogit | 1.6 | 1.9 | 2.1 | 2.1 | 55 |
| nlsur | 1.2 | 1.4 | 1.5 | 1.6 | 38 |
| npregress kernel | 1.0 | 1.0 | 1.0 | 1.0 | 1 |
| nptrend | 1.5 | 1.5 | 1.6 | 1.6 | 40 |
| nptrend_carmitage | 1.3 | 1.5 | 1.6 | 1.8 | 45 |
| nptrend_jterpstra | 1.6 | 2.1 | 2.4 | 2.6 | 64 |
| nptrend_linear | 1.1 | 1.2 | 1.2 | 1.2 | 21 |
| ologit | 2.0 | 3.7 | 7.0 | 12.6 | 99 |
| ologit, vce(cluster) | 2.0 | 3.6 | 6.2 | 9.5 | 96 |
| ologit, vce(robust) | 2.0 | 3.8 | 7.3 | 13.1 | 99 |
| oneway | 1.0 | 1.0 | 1.0 | 1.0 | 0 |
| oprobit | 2.0 | 3.9 | 7.3 | 13.3 | 98 |
| oprobit, vce(cluster) | 2.0 | 3.8 | 6.9 | 11.3 | 97 |
| oprobit, vce(robust) | 2.0 | 4.0 | 7.6 | 13.8 | 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.

Table 1. Stata/MP performance, command by command

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| orthog | 1.8 | 3.9 | 6.1 | 8.0 | 90 |
| pca | 1.7 | 2.3 | 2.9 | 3.0 | 72 |
| pcorr | 2.3 | 4.5 | 8.9 | 17.0 | 99 |
| pctile | 1.8 | 3.0 | 4.6 | 5.7 | 87 |
| pergram | 1.0 | 1.0 | 1.0 | 1.0 | 0 |
| pkcollapse | 0.8 | 0.9 | 0.9 | 0.9 | 0 |
| pkexamine | 1.1 | 1.2 | 1.2 | 1.2 | 19 |
| pksumm | 0.9 | 0.9 | 0.9 | 0.9 | 0 |
| poisson | 2.1 | 4.0 | 7.5 | 13.1 | 98 |
| poisson, vce(cluster) | 2.0 | 3.9 | 7.0 | 11.4 | 97 |
| poisson, exposure() | 2.1 | 4.1 | 7.6 | 13.2 | 98 |
| poisson, vce(robust) | 2.1 | 4.0 | 7.5 | 13.3 | 98 |
| pologit | 1.4 | 1.8 | 2.3 | 2.2 | 60 |
| popoisson | 1.4 | 1.9 | 2.2 | 2.3 | 57 |
| poregress | 1.3 | 1.7 | 2.0 | 2.0 | 55 |
| pperron | 1.1 | 1.1 | 1.1 | 1.8 | 48 |
| prais | 1.1 | 1.2 | 1.2 | 1.2 | 18 |
| predict, cooksd | 2.0 | 4.0 | 7.9 | 15.8 | 100 |
| predict, covratio | 2.0 | 4.0 | 7.8 | 15.0 | 99 |
| predict, dfbeta | 2.1 | 4.1 | 7.7 | 13.5 | 98 |
| predict, dfits | 2.0 | 4.0 | 7.8 | 14.8 | 99 |
| predict, e | 2.0 | 3.7 | 6.5 | 10.2 | 96 |
| predict, leverage | 2.0 | 4.0 | 7.8 | 15.1 | 99 |
| predict, pr | 2.0 | 3.7 | 6.6 | 10.6 | 96 |
| predict, residuals | 2.1 | 4.2 | 8.3 | 16.4 | 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

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| predict, rstandard | 2.0 | 4.1 | 8.1 | 16.1 | 100 |
| predict, rstudent | 2.0 | 4.0 | 8.0 | 15.9 | 100 |
| predict, stdf | 2.0 | 4.0 | 8.0 | 15.9 | 100 |
| predict, stdp | 2.0 | 4.0 | 8.1 | 16.1 | 100 |
| predict, stdr | 2.0 | 4.0 | 8.0 | 15.9 | 100 |
| predict, welsch | 2.1 | 4.1 | 8.1 | 15.9 | 100 |
| predict, ystar | 1.9 | 3.5 | 5.7 | 8.5 | 94 |
| predictnl | 1.9 | 3.6 | 6.0 | 8.9 | 94 |
| probit | 2.2 | 4.0 | 7.0 | 12.1 | 97 |
| procrustes | 2.0 | 4.2 | 6.2 | 8.4 | 92 |
| proportion | 1.5 | 2.0 | 2.7 | 3.0 | 80 |
| prtest1 | 1.7 | 2.9 | 4.2 | 5.9 | 89 |
| prtest2 | 1.7 | 2.7 | 4.1 | 5.2 | 87 |
| prtest, by() | 1.1 | 1.2 | 1.2 | 1.2 | 22 |
| pwcorr | 1.8 | 3.1 | 5.2 | 7.9 | 94 |
| qreg | 1.6 | 4.5 | 6.6 | 8.4 | 91 |
| ranksum | 3.0 | 3.5 | 3.9 | 4.2 | 79 |
| ratio | 1.4 | 1.7 | 2.0 | 2.1 | 57 |
| ratio (exp1) (exp2) | 1.4 | 1.8 | 2.2 | 2.3 | 61 |
| recode | 1.4 | 1.8 | 2.1 | 2.3 | 59 |
| reg3 | 2.1 | 3.8 | 6.3 | 9.0 | 94 |
| regress | 2.2 | 4.4 | 8.7 | 16.9 | 100 |
| regress, vce(cluster) | 2.0 | 3.4 | 5.2 | 6.8 | 91 |
| regress, vce(robust) | 2.1 | 4.1 | 7.8 | 14.3 | 99 |
| replace | 2.0 | 4.0 | 8.0 | 15.9 | 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

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| replace (small expressions) | 4.3 | 8.7 | 17.1 | 32.9 | 100 |
| reshape long | 1.0 | 1.1 | 1.8 | 1.9 | 51 |
| reshape wide | 1.0 | 1.0 | 1.1 | 1.1 | 7 |
| robvar | 1.4 | 1.4 | 1.4 | 1.5 | 69 |
| rocfit | 1.7 | 2.2 | 2.8 | 3.0 | 71 |
| roctab | 0.9 | 1.3 | 1.4 | 1.5 | 36 |
| rotate | 1.0 | 1.0 | 1.0 | 1.0 | 2 |
| rotatemat | 1.0 | 1.0 | 1.0 | 1.0 | 2 |
| rreg | 2.2 | 4.1 | 7.4 | 12.1 | 97 |
| runtest | 1.7 | 2.7 | 4.0 | 5.1 | 86 |
| scobit | 1.9 | 3.7 | 6.8 | 11.6 | 97 |
| scoreplot | 1.9 | 2.6 | 3.2 | 3.5 | 74 |
| screeplot | 1.1 | 1.1 | 1.3 | 1.3 | 21 |
| sdtest1 | 1.4 | 1.8 | 2.0 | 1.8 | 59 |
| sdtest2 | 1.3 | 1.6 | 1.8 | 2.0 | 55 |
| sdtest, by() | 1.3 | 1.7 | 1.9 | 2.0 | 56 |
| sem, method(adf) (CFA) | 2.0 | 4.0 | 7.7 | 14.8 | 99 |
| sem, method(ml) (CFA) | 1.6 | 2.1 | 2.6 | 2.9 | 70 |
| sem, method(mlmv) (CFA) | 1.1 | 1.1 | 1.1 | 1.1 | 13 |
| sem (SEM latent) | 1.6 | 2.2 | 2.7 | 3.1 | 71 |
| sem (SEM observed) | 1.5 | 2.2 | 2.6 | 3.0 | 70 |
| separate | 1.4 | 1.8 | 1.9 | 2.0 | 53 |
| sfrancia | 2.5 | 3.4 | 4.3 | 4.8 | 94 |
| signrank | 2.9 | 3.4 | 3.9 | 4.0 | 77 |
| signtest | 2.0 | 4.0 | 7.7 | 13.8 | 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

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| sktest | 2.1 | 3.7 | 5.2 | 6.4 | 88 |
| slogit | 1.2 | 1.5 | 1.8 | 1.9 | 50 |
| sort | 1.2 | 1.7 | 1.9 | 2.3 | 61 |
| spearman | 3.8 | 4.6 | 5.2 | 6.3 | 85 |
| sspace | 1.6 | 2.0 | 2.5 | 2.5 | 59 |
| stack | 1.4 | 2.0 | 2.3 | 2.4 | 61 |
| stci | 1.1 | 1.3 | 2.6 | 2.7 | 67 |
| stcox | 1.0 | 1.1 | 1.1 | 1.1 | 12 |
| stcrreg | 1.0 | 1.0 | 1.0 | 1.0 | 3 |
| stgen | 1.7 | 2.4 | 3.1 | 3.5 | 75 |
| stintcox | 1.2 | 1.3 | 1.4 | 1.4 | 30 |
| stintreg, d(exponential) | 1.7 | 3.0 | 4.0 | 4.6 | 83 |
| stintreg, d(weibull) | 2.6 | 4.1 | 5.9 | 7.2 | 90 |
| stir | 1.3 | 1.5 | 1.7 | 2.1 | 55 |
| stmc | 3.0 | 3.4 | 3.8 | 4.2 | 78 |
| by: stmc | 2.9 | 3.5 | 4.0 | 5.0 | 82 |
| stmh | 1.2 | 1.6 | 1.8 | 1.9 | 50 |
| by: stmh | 1.1 | 1.4 | 1.5 | 1.6 | 39 |
| stptime | 1.1 | 1.2 | 1.3 | 1.3 | 26 |
| strate | 1.3 | 1.5 | 1.7 | 2.0 | 54 |
| streg, distribution(exponential) | 1.9 | 3.7 | 6.6 | 11.0 | 97 |
| streg, dist(exp) vce(cluster) | 2.0 | 3.9 | 7.2 | 12.8 | 98 |
| streg, dist(exp) frailty() | 2.0 | 3.8 | 6.6 | 10.7 | 97 |
| streg, dist(exp) frailty() shared() | 2.0 | 3.8 | 6.3 | 9.6 | 95 |
| streg, dist(exp) vce(robust) | 2.0 | 3.9 | 7.4 | 13.5 | 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.

Table 1. Stata/MP performance, command by command

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| streg, distribution(ggamma) | 2.3 | 4.5 | 7.8 | 12.0 | 97 |
| streg, dist(ggamma) vce(cluster) | 2.1 | 4.6 | 8.1 | 11.8 | 97 |
| streg, dist(ggamma) vce(robust) | 2.2 | 4.7 | 8.4 | 13.2 | 97 |
| streg, distribution(gompertz) | 2.3 | 4.6 | 8.1 | 13.0 | 97 |
| streg, dist(gompertz) vce(cluster) | 2.0 | 4.1 | 7.0 | 10.8 | 96 |
| streg, dist(gompertz) frailty() | 1.9 | 4.2 | 7.3 | 11.2 | 96 |
| streg, dist(gomp) frailty() shared() | 2.3 | 4.4 | 7.0 | 9.4 | 93 |
| streg, dist(gompertz) vce(robust) | 2.1 | 4.3 | 7.2 | 11.7 | 96 |
| streg, distribution(llogistic) | 1.9 | 3.9 | 6.9 | 11.4 | 97 |
| streg, dist(llogistic) vce(cluster) | 2.1 | 4.2 | 7.7 | 13.1 | 98 |
| streg, dist(llogistic) frailty() | 2.0 | 3.8 | 6.9 | 11.4 | 97 |
| streg, dist(llog) frailty() shared() | 2.0 | 4.2 | 7.5 | 12.0 | 97 |
| streg, dist(llogistic) vce(robust) | 2.1 | 4.1 | 7.7 | 13.5 | 98 |
| streg, distribution(lnormal) | 2.1 | 4.1 | 7.0 | 10.5 | 96 |
| streg, dist(lnormal) vce(cluster) | 2.0 | 3.9 | 6.9 | 11.4 | 97 |
| streg, dist(lnormal) frailty() | 2.0 | 3.7 | 6.4 | 10.2 | 96 |
| streg, dist(lnorm) frailty() shared() | 1.9 | 3.8 | 5.8 | 7.8 | 92 |
| streg, dist(lnormal) vce(robust) | 2.1 | 4.0 | 7.3 | 12.2 | 97 |
| streg, distribution(weibull) | 2.0 | 3.9 | 7.3 | 13.1 | 98 |
| streg, dist(weibull) vce(cluster) | 2.1 | 4.0 | 7.5 | 13.0 | 98 |
| streg, dist(weibull) frailty() | 2.2 | 4.9 | 8.5 | 12.7 | 96 |
| streg, dist(weib) frailty() shared() | 2.1 | 3.9 | 6.6 | 10.0 | 95 |
| streg, dist(weibull) vce(robust) | 2.1 | 4.1 | 7.7 | 13.8 | 99 |
| sts generate | 1.0 | 1.2 | 1.3 | 1.3 | 30 |
| sts graph | 1.1 | 1.2 | 1.3 | 1.3 | 27 |

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 | |
| sts list | 1.1 | 1.3 | 1.3 | 1.4 | 30 |
| sts test | 1.2 | 1.3 | 1.4 | 1.5 | 37 |
| stset | 1.4 | 1.8 | 2.1 | 2.3 | 60 |
| stsplit | 1.0 | 1.1 | 1.2 | 1.2 | 17 |
| stsum | 0.9 | 1.3 | 1.8 | 1.9 | 51 |
| stteffects ipw (weibull) | 2.0 | 3.4 | 5.2 | 6.8 | 90 |
| stteffects ipwra (weibull) | 1.8 | 2.7 | 3.6 | 4.3 | 81 |
| stteffects ra (weibull) | 1.7 | 2.5 | 3.2 | 3.6 | 76 |
| stteffects wra (weibull) | 1.8 | 2.7 | 3.5 | 4.0 | 79 |
| stvary | 1.3 | 1.7 | 2.1 | 2.3 | 60 |
| suest | 2.0 | 3.8 | 7.1 | 12.5 | 98 |
| summarize | 2.3 | 4.8 | 9.5 | 18.8 | 100 |
| sunflower | 1.2 | 1.5 | 1.7 | 5.4 | 88 |
| sureg | 2.1 | 3.8 | 6.4 | 9.5 | 95 |
| svar | 1.6 | 1.8 | 2.0 | 2.0 | 55 |
| svmat | 1.1 | 1.1 | 1.1 | 1.1 | 5 |
| svy brr: logit | 1.4 | 1.9 | 2.3 | 2.6 | 64 |
| svy brr: poisson | 1.6 | 2.2 | 2.8 | 3.3 | 73 |
| svy brr: regress | 2.1 | 3.8 | 6.2 | 9.2 | 94 |
| svy jackknife: logit | 1.8 | 2.7 | 3.6 | 4.2 | 80 |
| svy jackknife: poisson | 1.5 | 2.2 | 2.9 | 3.4 | 76 |
| svy jackknife: regress | 2.0 | 3.4 | 5.1 | 6.9 | 90 |
| svy linearized: logit | 2.0 | 3.5 | 5.8 | 8.8 | 94 |
| svy linearized: poisson | 2.0 | 3.7 | 6.0 | 9.2 | 94 |
| svy linearized: regress | 1.9 | 3.2 | 5.0 | 6.6 | 90 |

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 |
|---|---|---|---|---|---|
| | Number of cores | | | | |
| | 2 | 4 | 8 | 16 | parallelized[b] |
| swilk | 3.6 | 4.4 | 5.8 | 6.5 | 90 |
| symmetry | 1.1 | 1.3 | 1.4 | 1.4 | 32 |
| table (one-way) | 1.1 | 1.2 | 1.4 | 1.5 | 33 |
| table (two-way) | 1.1 | 1.2 | 1.3 | 1.3 | 23 |
| tabodds | 1.0 | 1.0 | 1.1 | 1.1 | 8 |
| tabodds (adjusted) | 0.7 | 0.8 | 0.8 | 0.9 | 0 |
| tabstat | 1.7 | 2.1 | 2.3 | 2.5 | 61 |
| tabstat, by() | 1.1 | 1.3 | 2.8 | 2.9 | 70 |
| tabulate (one-way) | 1.0 | 1.0 | 1.0 | 1.0 | 0 |
| tabulate (two-way) | 1.0 | 1.0 | 1.0 | 1.0 | 0 |
| teffects aipw (linear) | 1.9 | 3.4 | 5.5 | 6.7 | 89 |
| teffects aipw (probit) | 1.9 | 3.3 | 5.3 | 6.4 | 88 |
| teffects ipw (logit) | 2.1 | 4.2 | 6.9 | 10.2 | 93 |
| teffects ipwra (linear) | 1.9 | 3.3 | 5.4 | 6.4 | 88 |
| teffects ipwra (probit) | 1.9 | 3.3 | 5.1 | 6.1 | 87 |
| teffects nnmatch | 2.0 | 3.9 | 7.7 | 14.5 | 99 |
| teffects psmatch, logit | 1.1 | 1.1 | 1.1 | 1.1 | 10 |
| teffects ra (linear) | 2.0 | 3.7 | 6.3 | 8.1 | 92 |
| teffects ra (probit) | 2.0 | 3.6 | 5.9 | 7.5 | 90 |
| telasso (, linear) (, probit), ate | 1.2 | 1.4 | 1.6 | 1.6 | 45 |
| telasso (, linear) (, probit), atet | 1.3 | 1.5 | 1.8 | 1.8 | 47 |
| telasso (, linear) (, probit), pomeans | 1.2 | 1.5 | 1.6 | 1.6 | 44 |
| telasso (, logit) (, probit), ate | 1.6 | 2.4 | 2.9 | 3.3 | 71 |
| telasso (, logit) (, probit), atet | 1.7 | 2.5 | 3.1 | 3.4 | 74 |
| telasso (, logit) (, probit), pomeans | 1.7 | 2.5 | 3.2 | 3.6 | 75 |

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

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| telasso (, poisson) (, probit), ate | 1.6 | 2.3 | 3.0 | 3.5 | 76 |
| telasso (, poisson) (, probit), atet | 1.5 | 2.4 | 3.0 | 3.6 | 76 |
| telasso (, poisson) (, probit), pomeans | 1.5 | 2.3 | 3.0 | 3.5 | 76 |
| telasso (, probit) (, probit), ate | 1.8 | 2.7 | 3.4 | 3.7 | 76 |
| telasso (, probit) (, probit), atet | 1.7 | 2.5 | 3.1 | 3.2 | 73 |
| telasso (, probit) (, probit), pomeans | 1.7 | 2.6 | 3.4 | 3.9 | 77 |
| tetrachoric | 1.2 | 1.4 | 1.5 | 1.5 | 39 |
| threshold, threshvar() | 0.9 | 0.9 | 1.1 | 0.9 | 0 |
| threshold, threshvar() regionvars() | 1.0 | 1.0 | 1.2 | 1.1 | 11 |
| tnbreg | 1.3 | 2.4 | 3.1 | 3.5 | 76 |
| tobit | 1.9 | 3.3 | 5.0 | 7.0 | 90 |
| tostring | 1.0 | 1.0 | 1.0 | 1.1 | 12 |
| total | 2.0 | 3.8 | 6.9 | 11.6 | 97 |
| tpoisson | 1.9 | 3.5 | 5.8 | 8.4 | 94 |
| truncreg | 1.9 | 3.3 | 5.3 | 7.3 | 92 |
| tsfilter bk | 1.0 | 1.1 | 1.1 | 1.1 | 13 |
| tsfilter bw | 1.4 | 1.8 | 2.1 | 2.3 | 60 |
| tsfilter cf | 1.0 | 1.1 | 1.1 | 1.1 | 14 |
| tsfilter hp | 1.4 | 1.8 | 2.1 | 2.3 | 60 |
| tsrevar | 2.4 | 4.6 | 8.3 | 14.1 | 96 |
| tsset | 1.2 | 1.4 | 1.7 | 1.8 | 51 |
| tssmooth exp | 1.2 | 1.5 | 1.6 | 1.7 | 43 |
| tssmooth ma | 1.1 | 1.2 | 1.3 | 1.3 | 26 |
| ttest1 | 1.3 | 1.7 | 2.0 | 2.2 | 57 |
| ttest2 | 1.7 | 2.4 | 2.9 | 2.6 | 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

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| ttest, by() | 1.4 | 1.7 | 2.0 | 2.1 | 56 |
| twoway fpfit | 1.5 | 2.4 | 3.8 | 5.1 | 85 |
| twoway lfitci | 1.2 | 1.4 | 1.6 | 1.3 | 42 |
| twoway mband | 2.5 | 3.7 | 4.8 | 5.6 | 84 |
| twoway mspline | 2.3 | 3.5 | 4.2 | 4.7 | 82 |
| ucm, model(rwdrift) | 1.2 | 1.3 | 1.3 | 1.3 | 20 |
| var | 1.5 | 3.2 | 3.6 | 3.9 | 75 |
| vargranger | 1.1 | 2.8 | 2.8 | 2.8 | 65 |
| varlmar | 1.3 | 1.7 | 1.8 | 2.5 | 61 |
| varnorm | 1.2 | 1.8 | 2.1 | 2.2 | 59 |
| varsoc | 1.2 | 1.5 | 1.8 | 2.1 | 55 |
| varstable | 1.0 | 1.0 | 1.3 | 1.3 | 27 |
| vec | 1.2 | 1.4 | 1.5 | 1.6 | 40 |
| veclmar | 1.1 | 1.2 | 1.4 | 1.4 | 32 |
| vecnorm | 1.2 | 1.5 | 1.6 | 3.2 | 73 |
| vecrank | 1.0 | 1.4 | 1.5 | 1.6 | 39 |
| vecstable | 1.1 | 1.1 | 1.1 | 1.1 | 7 |
| vwls | 2.2 | 4.3 | 8.0 | 13.8 | 98 |
| wntestb | 1.0 | 1.0 | 1.0 | 1.0 | 0 |
| wntestq | 2.1 | 2.2 | 2.2 | 2.2 | 54 |
| xcorr | 1.0 | 1.1 | 1.1 | 1.1 | 5 |
| xpologit | 1.2 | 1.3 | 1.4 | 1.4 | 29 |
| xpopoisson | 1.1 | 1.3 | 1.4 | 1.4 | 30 |
| xporegress | 1.2 | 1.3 | 1.4 | 1.4 | 29 |
| xtabond | 1.0 | 1.2 | 1.4 | 1.4 | 32 |

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

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| xtabond, twostep | 1.0 | 1.2 | 1.3 | 1.4 | 32 |
| xtcloglog, re | 2.0 | 3.9 | 6.6 | 9.9 | 95 |
| xtdata, be | 2.1 | 2.5 | 2.6 | 2.7 | 65 |
| xtdata, fe | 3.2 | 4.2 | 5.1 | 5.5 | 84 |
| xtdata, re | 3.2 | 3.9 | 5.0 | 5.3 | 84 |
| xtdidregress | 1.4 | 2.0 | 2.5 | 2.6 | 65 |
| xtdpd | 1.2 | 1.4 | 1.6 | 1.6 | 40 |
| xtdpdsys | 1.1 | 1.3 | 1.4 | 1.4 | 32 |
| xteregress | 1.4 | 1.7 | 1.8 | 1.8 | 44 |
| xtfrontier | 2.7 | 4.6 | 7.2 | 9.7 | 93 |
| xtgee, family(gaussian) corr(ar2) | 1.5 | 1.7 | 1.9 | 1.9 | 51 |
| xtgee, fam(gauss) corr(unstruct) | 1.5 | 1.7 | 1.9 | 2.0 | 52 |
| xtcloglog, pa | 1.6 | 2.4 | 3.3 | 4.0 | 80 |
| xtlogit, pa | 1.4 | 1.7 | 1.9 | 2.1 | 57 |
| xtnbreg, pa | 1.6 | 2.2 | 2.8 | 3.2 | 73 |
| xtpoisson, pa | 1.5 | 2.0 | 2.5 | 2.7 | 67 |
| xtprobit, pa | 1.3 | 1.5 | 1.9 | 2.1 | 57 |
| xtreg, pa | 1.3 | 1.6 | 1.8 | 1.9 | 51 |
| xtgls | 1.4 | 1.7 | 1.9 | 2.1 | 54 |
| xthtaylor | 1.3 | 2.3 | 3.2 | 3.9 | 80 |
| xtile | 1.0 | 1.0 | 1.1 | 1.0 | 5 |
| xtintreg | 2.0 | 3.8 | 6.7 | 10.9 | 96 |
| xtivreg, be | 1.9 | 2.6 | 3.1 | 3.4 | 74 |
| xtivreg, fd | 2.4 | 2.7 | 3.1 | 3.4 | 73 |
| xtivreg, fe | 2.0 | 2.7 | 3.3 | 3.6 | 75 |

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

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| xtivreg, re | 1.9 | 2.8 | 3.3 | 3.7 | 75 |
| xtlogit, fe | 1.5 | 2.2 | 2.7 | 3.0 | 71 |
| xtlogit, re | 2.1 | 3.8 | 5.6 | 7.5 | 91 |
| xtmlogit, fe | 1.0 | 1.0 | 1.0 | 1.0 | 4 |
| xtmlogit, re | 1.8 | 3.0 | 4.2 | 5.1 | 85 |
| xtnbreg, fe | 3.3 | 5.7 | 8.1 | 10.0 | 93 |
| xtnbreg, re | 3.0 | 4.9 | 7.2 | 9.0 | 92 |
| xtologit | 1.8 | 2.8 | 3.6 | 4.1 | 79 |
| xtoprobit | 1.8 | 2.8 | 3.8 | 4.5 | 82 |
| xtpcse | 1.3 | 1.5 | 1.6 | 1.6 | 41 |
| xtpoisson, fe | 3.1 | 4.9 | 7.3 | 9.8 | 93 |
| xtpoisson, re | 3.1 | 5.8 | 10.3 | 15.9 | 97 |
| xtprobit, re | 2.1 | 3.7 | 6.0 | 8.8 | 94 |
| xtrc | 1.6 | 2.4 | 3.0 | 3.4 | 74 |
| xtreg, be | 1.5 | 2.0 | 2.2 | 2.4 | 63 |
| xtreg, fe | 2.0 | 3.5 | 5.6 | 8.1 | 93 |
| xtreg, fe vce(robust) | 2.0 | 3.6 | 6.0 | 9.1 | 95 |
| xtreg, mle | 1.3 | 1.7 | 3.6 | 3.8 | 79 |
| xtreg, re | 2.3 | 3.1 | 3.6 | 4.0 | 78 |
| xtregar, fe | 1.7 | 2.6 | 4.2 | 4.3 | 80 |
| xtregar, re | 1.8 | 2.4 | 2.8 | 2.9 | 68 |
| xtset | 1.1 | 1.2 | 1.3 | 1.3 | 25 |
| xtstreg, distribution(exponential) | 1.7 | 2.5 | 3.1 | 3.4 | 74 |
| xtstreg, distribution(weibull) | 1.8 | 2.7 | 3.6 | 4.2 | 80 |
| xtsum | 1.7 | 2.5 | 3.3 | 4.1 | 78 |

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

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| xttab | 1.3 | 1.6 | 1.8 | 1.9 | 54 |
| xttobit | 1.9 | 3.9 | 6.3 | 9.2 | 94 |
| xtunitroot breitung | 0.9 | 1.1 | 1.1 | 1.2 | 19 |
| xtunitroot fisher | 1.0 | 1.0 | 1.1 | 1.1 | 6 |
| xtunitroot hadri | 1.3 | 1.3 | 1.3 | 1.3 | 25 |
| xtunitroot ht | 1.2 | 1.4 | 1.5 | 1.7 | 43 |
| xtunitroot ips | 1.0 | 1.0 | 1.0 | 1.0 | 2 |
| xtunitroot llc | 1.0 | 1.0 | 1.0 | 1.0 | 1 |
| zinb | 2.0 | 4.1 | 7.4 | 11.8 | 97 |
| ziologit | 1.8 | 3.1 | 4.6 | 5.9 | 88 |
| zioprobit | 1.9 | 3.3 | 5.1 | 6.9 | 90 |
| zip | 1.9 | 3.9 | 7.1 | 12.4 | 98 |
| _predict, xb | 2.1 | 4.2 | 8.3 | 16.5 | 100 |
| _rmcoll | 2.1 | 4.2 | 8.0 | 16.0 | 100 |
| _robust | 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 surpisingly 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.

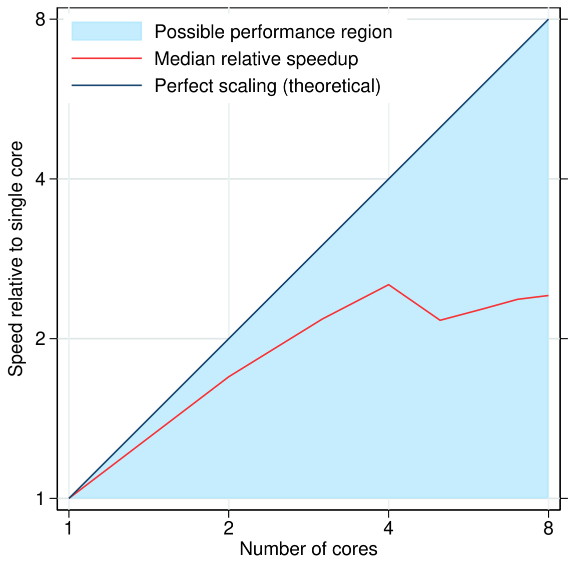


Figure 13. **Performance of Stata/MP on hyperthreaded CPUs.** Speed on multiple cores relative to speed on a single core.

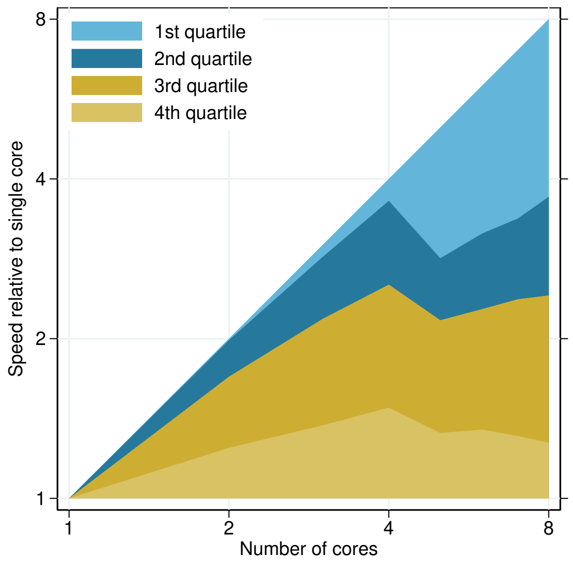


Figure 14. **Quartiles of Stata/MP Performance on hyperthreaded CPUs.** Speed on multiple cores relative speed on a single core.

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 hypertheading, 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 hyperthreaded computers, and these results should therefore be considered provisional.

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

```
. set 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 hyperthreaded, giving the appearance of 8 virtual processors.

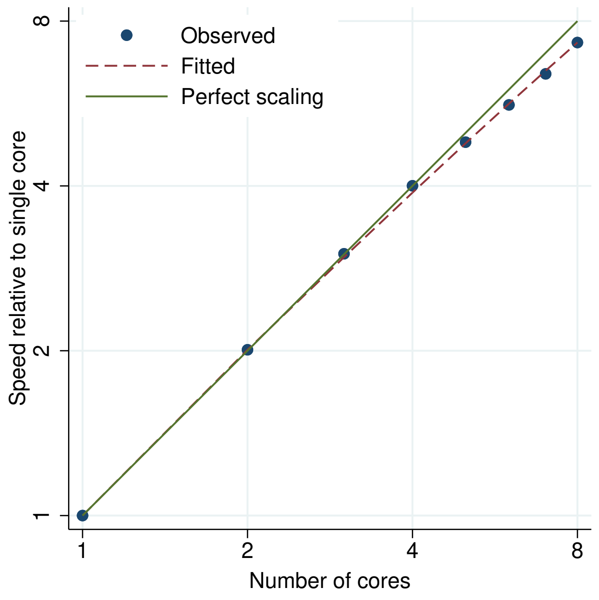


Figure 15. `predict, leverage` performance plot on computers with hyperthreaded CPUs.

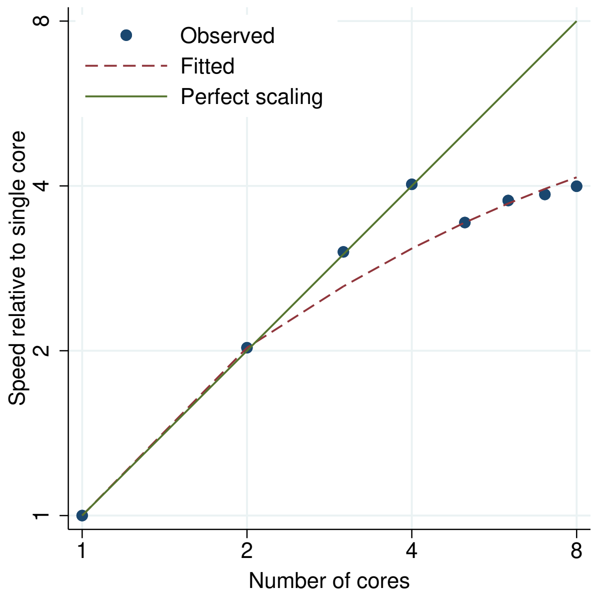


Figure 16. `regress` performance plot on computers with hyperthreaded CPUs.

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.

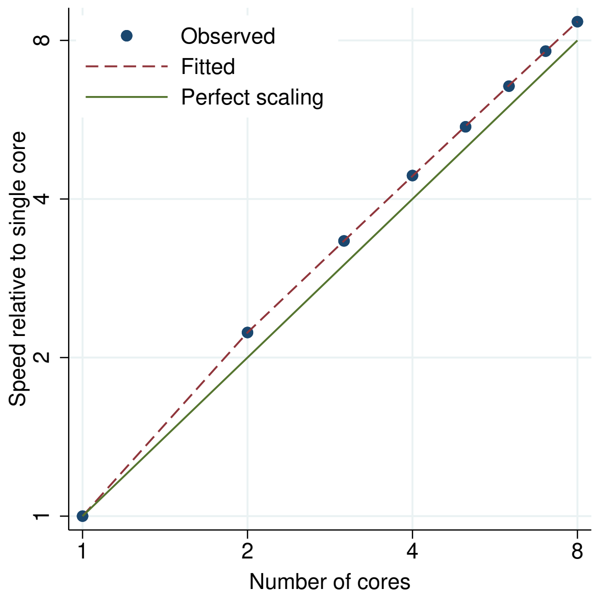


Figure 17. `regress` performance plot.

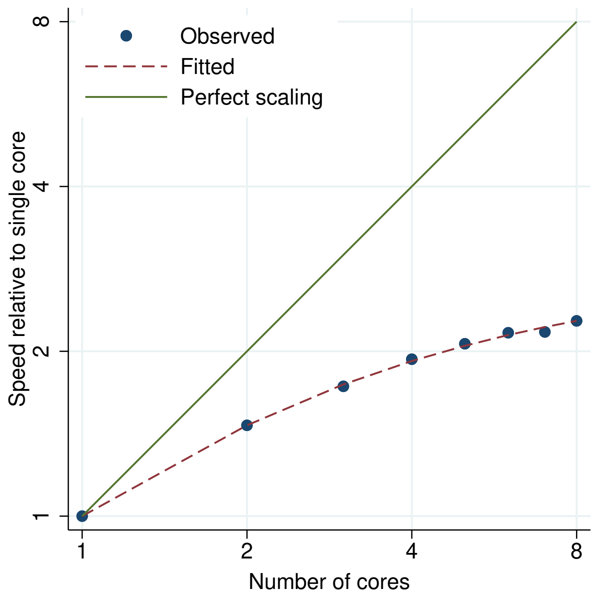


Figure 18. `clogit` (1 to k matching) performance plot.

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.

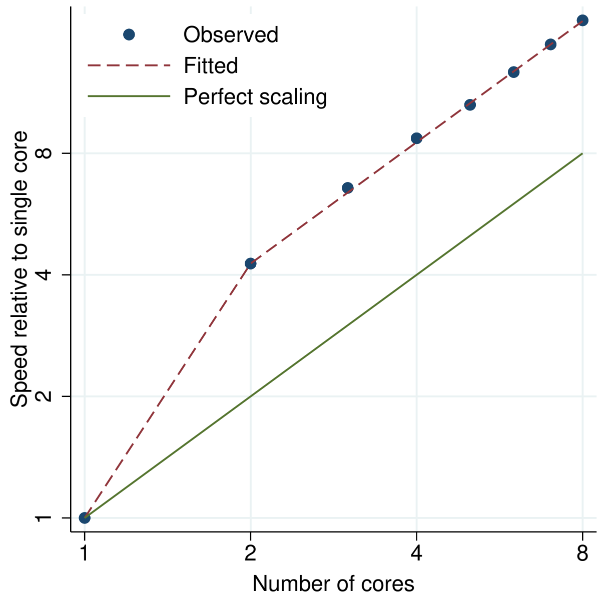


Figure 19. `replace` performance plot.

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.

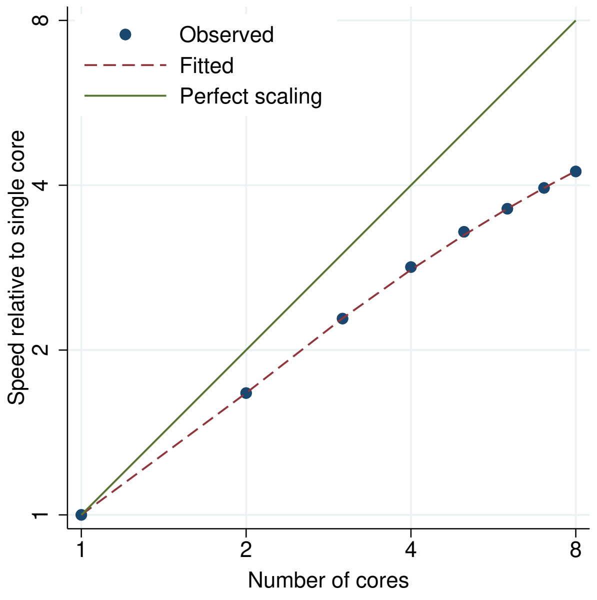


Figure 20. `alpha` performance plot.

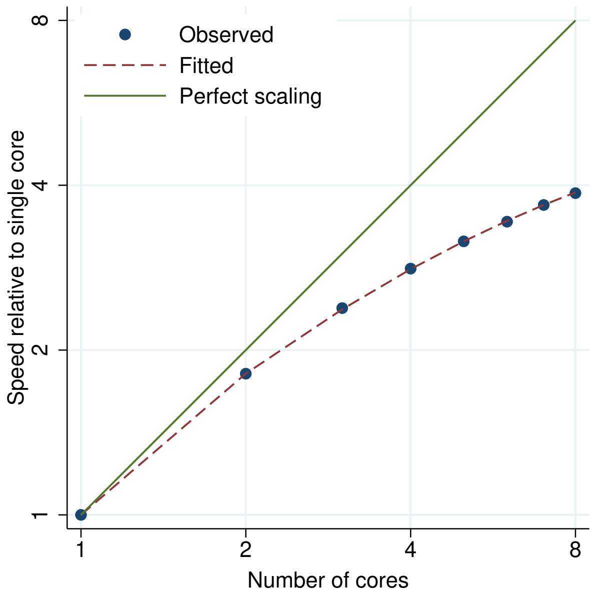


Figure 21. `ameans` performance plot.

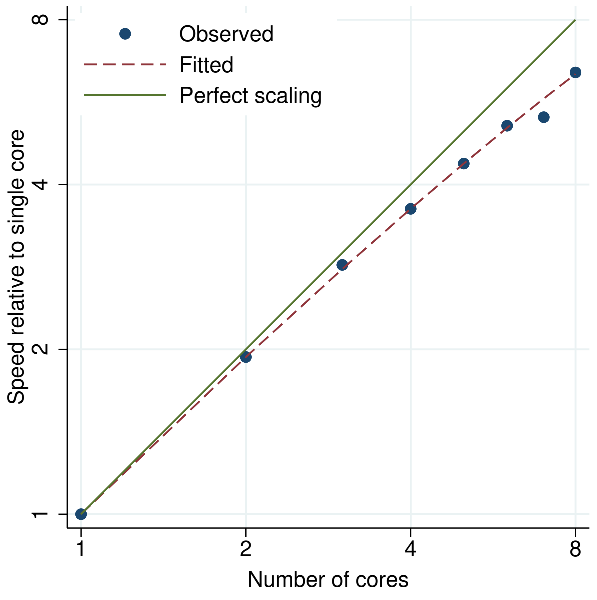


Figure 22. `anova` (one-way) performance plot.

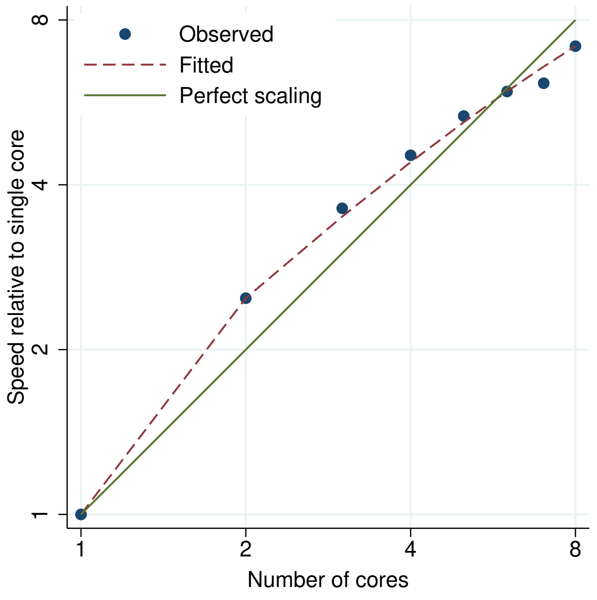


Figure 23. `anova` (two-way) performance plot.

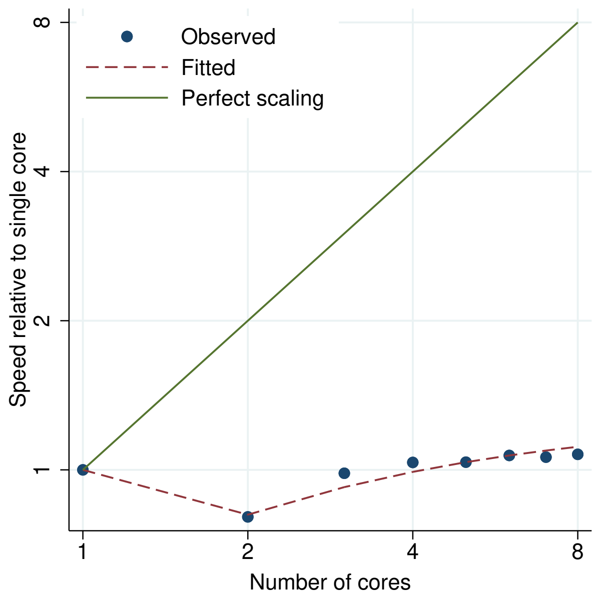


Figure 24. `arch` performance plot.

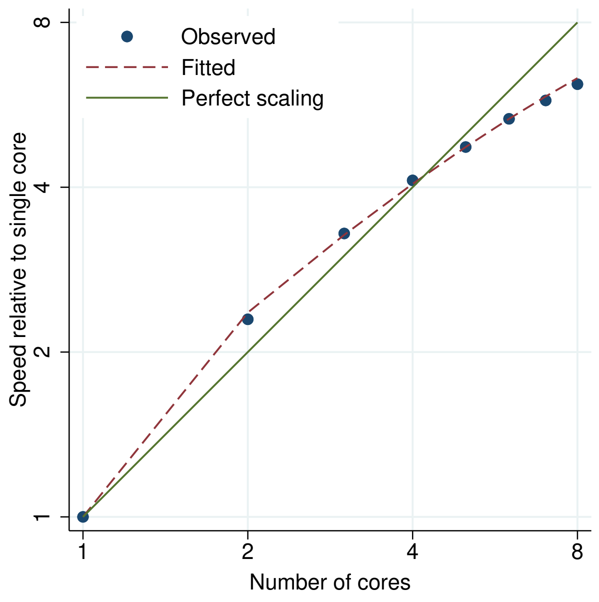


Figure 25. `areg` performance plot.

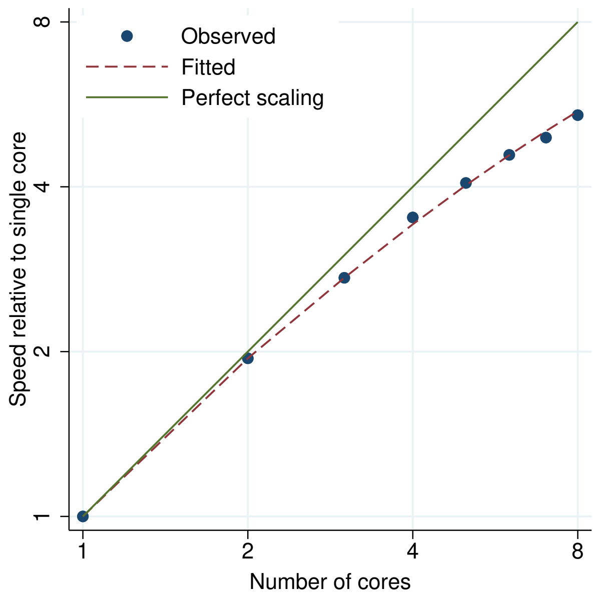


Figure 26. `areg, vce(cluster)` performance plot.

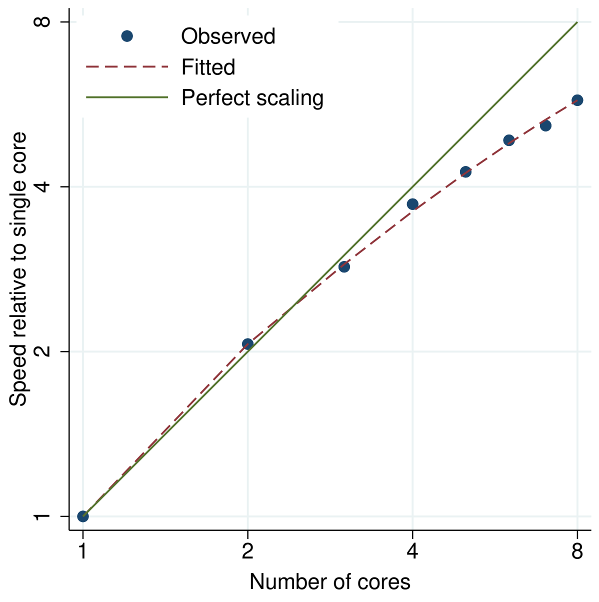


Figure 27. `areg, vce(robust)` performance plot.

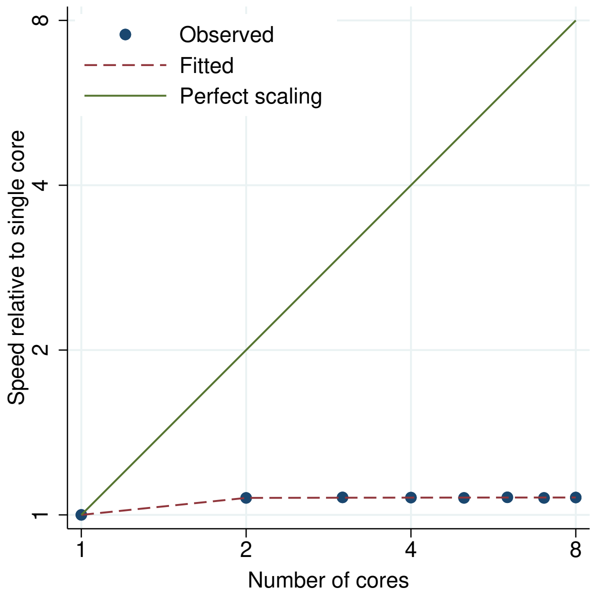


Figure 28. `arfima` performance plot.

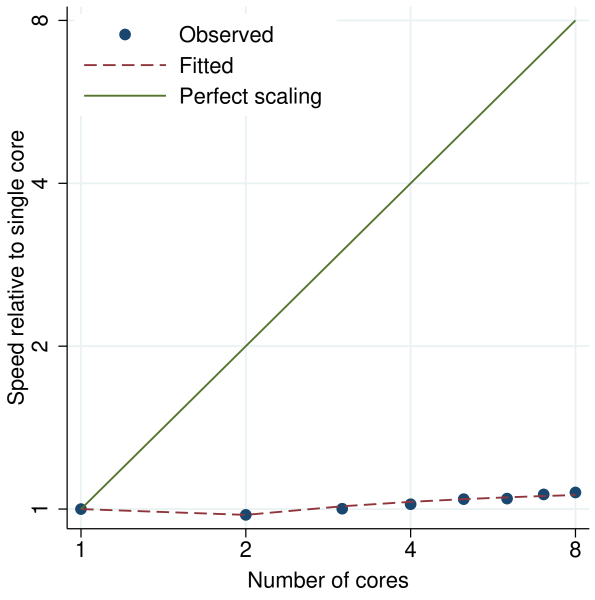


Figure 29. `arima` performance plot.

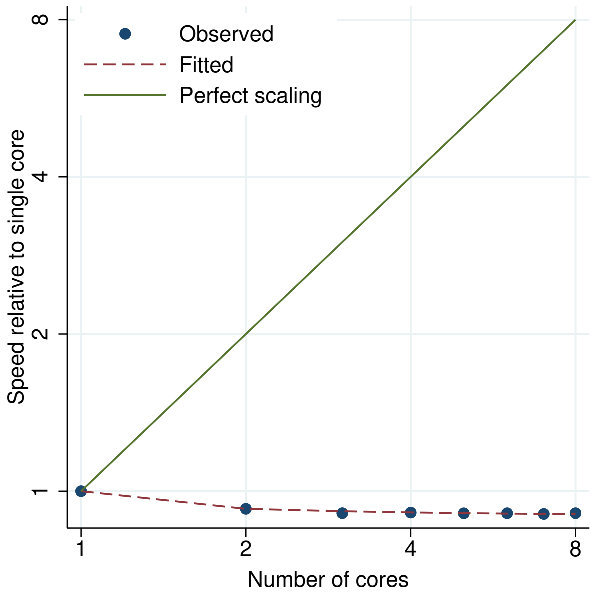


Figure 30. `bayes dsge` performance plot.

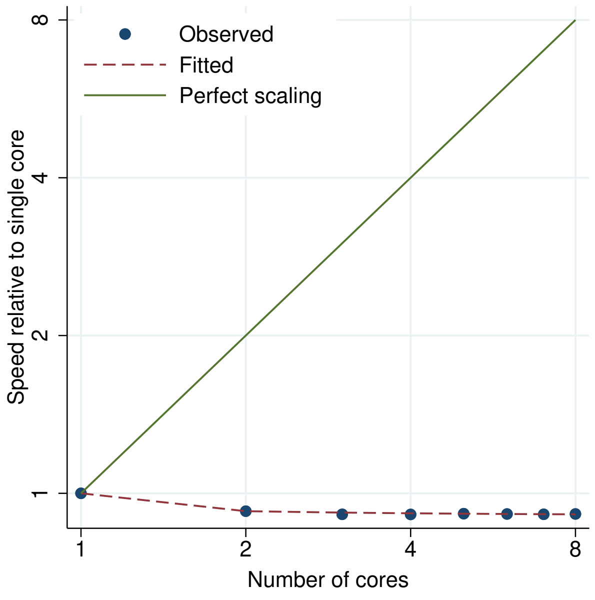


Figure 31. `bayes dsgenl` performance plot.

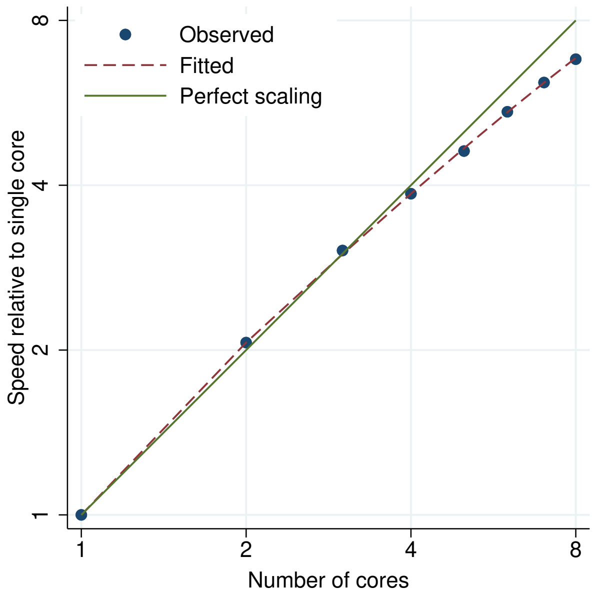


Figure 32. `bayes: logit` performance plot.

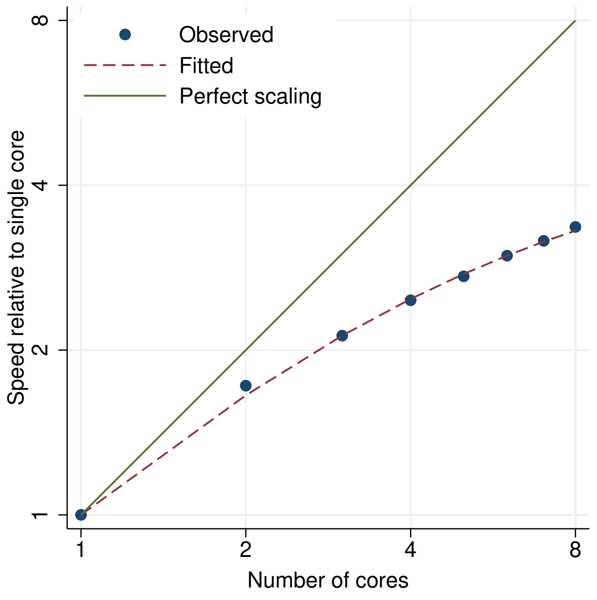


Figure 33. `bayes: poisson` performance plot.

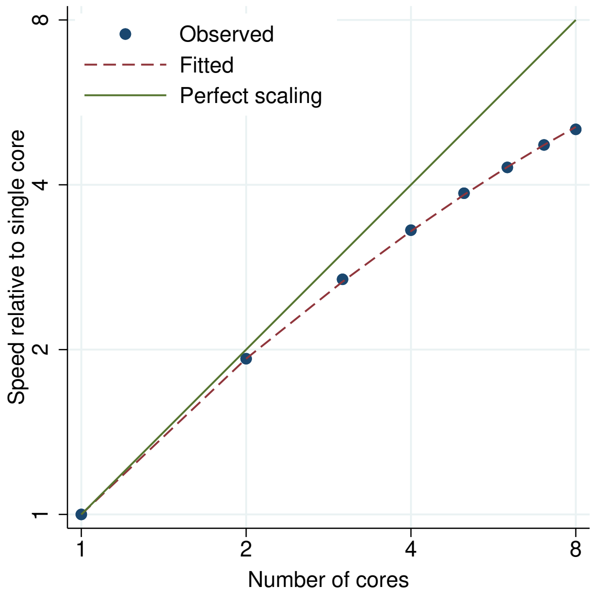


Figure 34. `bayes: regress` performance plot.

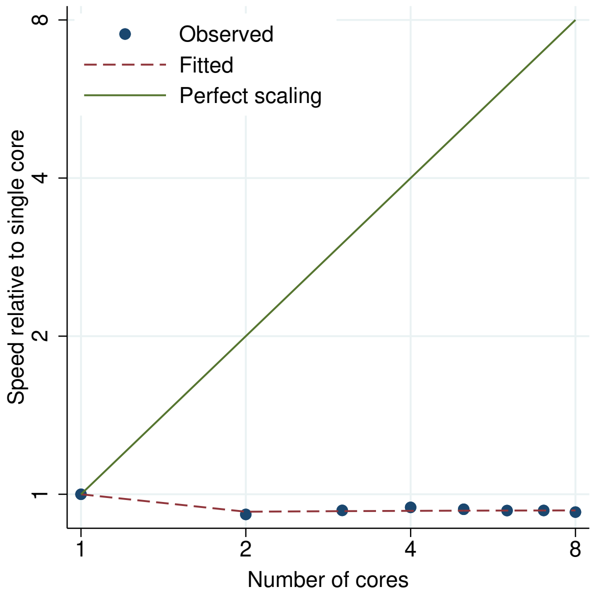


Figure 35. `bayes var` performance plot.

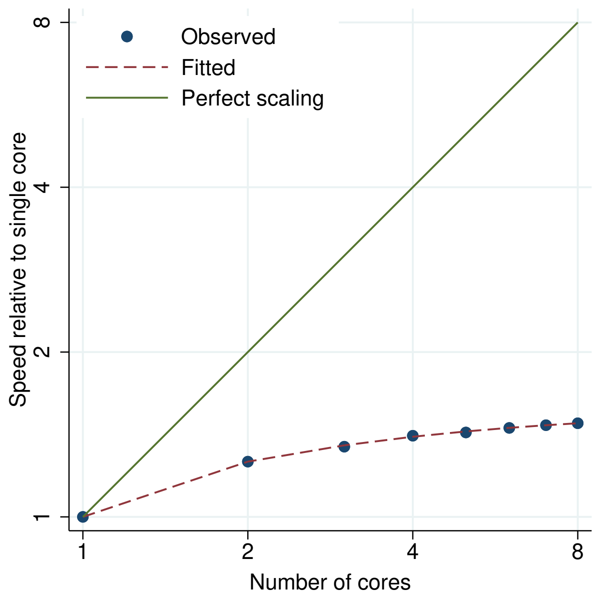


Figure 36. `bayesmh logit` performance plot.

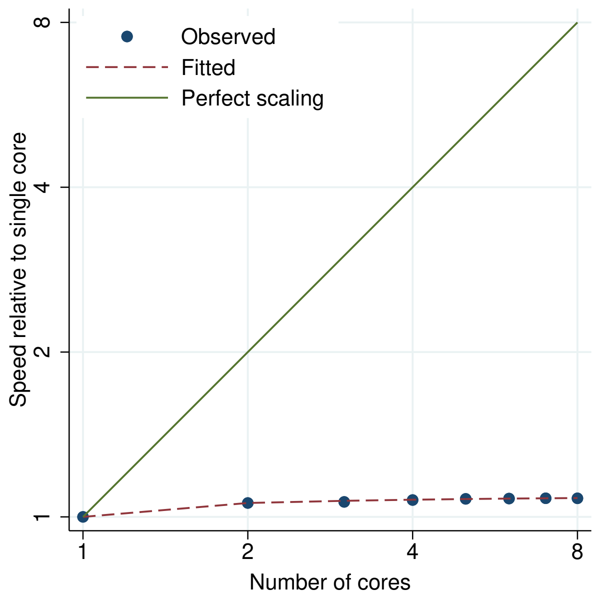


Figure 37. `bayesmh mvn` performance plot.

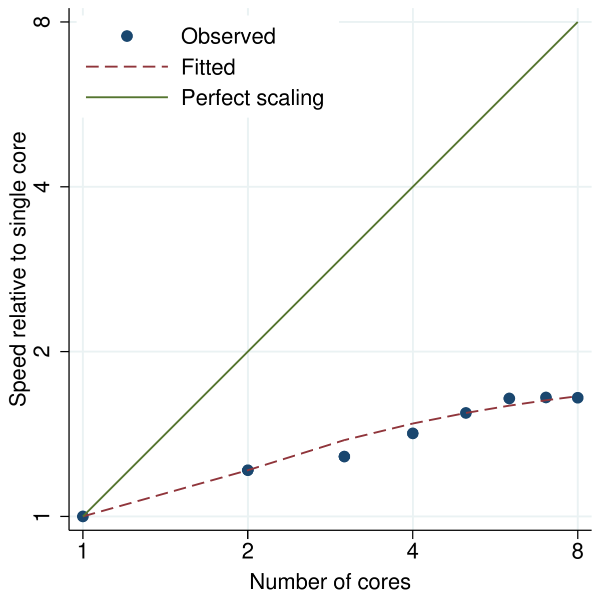


Figure 38. `bayesmh mylogit` performance plot.

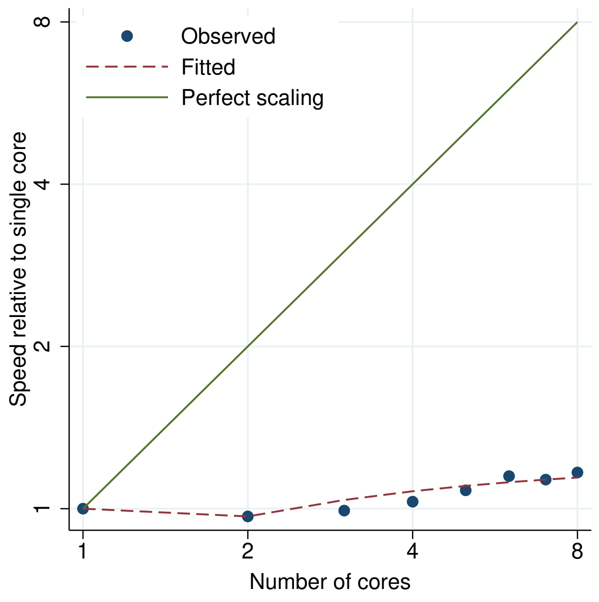


Figure 39. `bayesmh nl` performance plot.

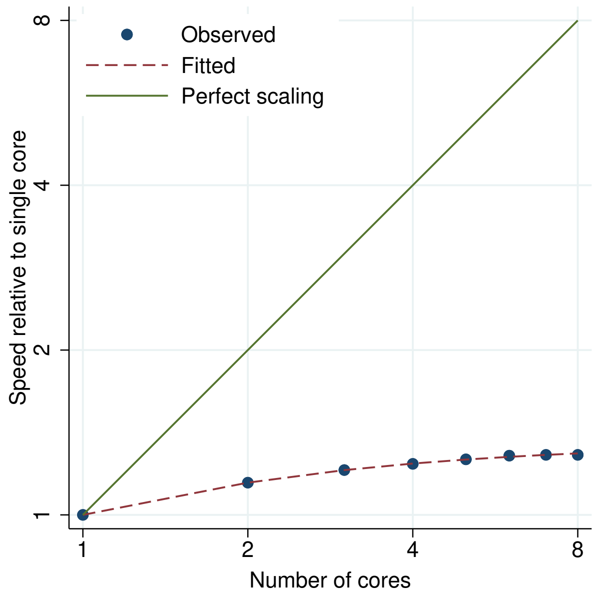


Figure 40. `bayesmh normal` performance plot.

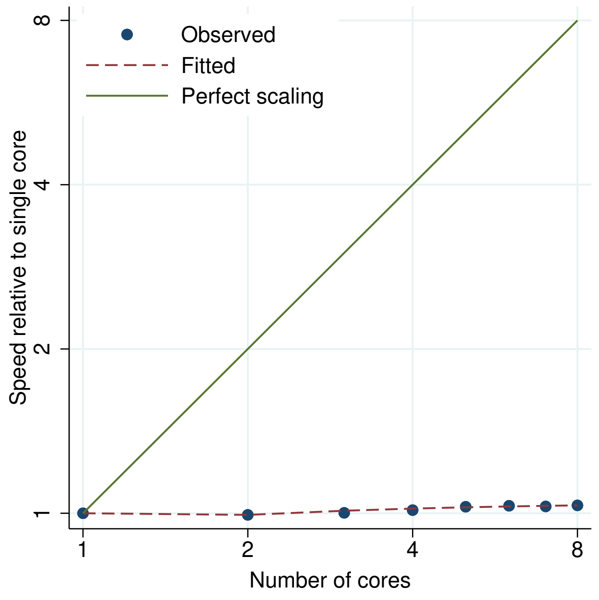


Figure 41. `bayesmh normal gibbs` performance plot.

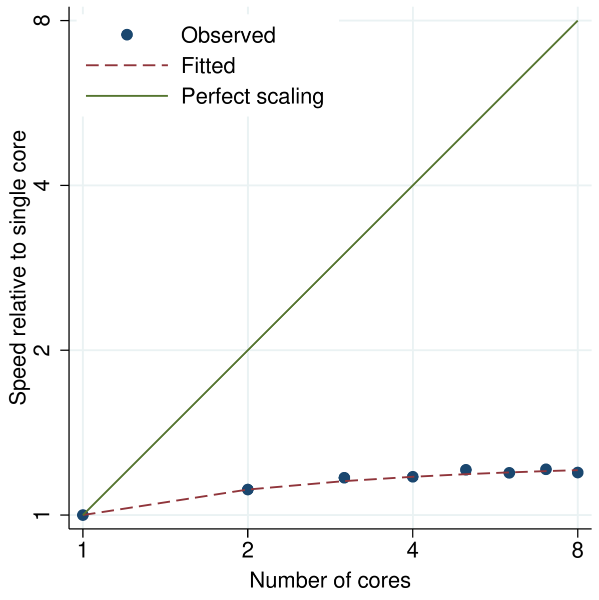


Figure 42. `bayesmh normal re` performance plot.

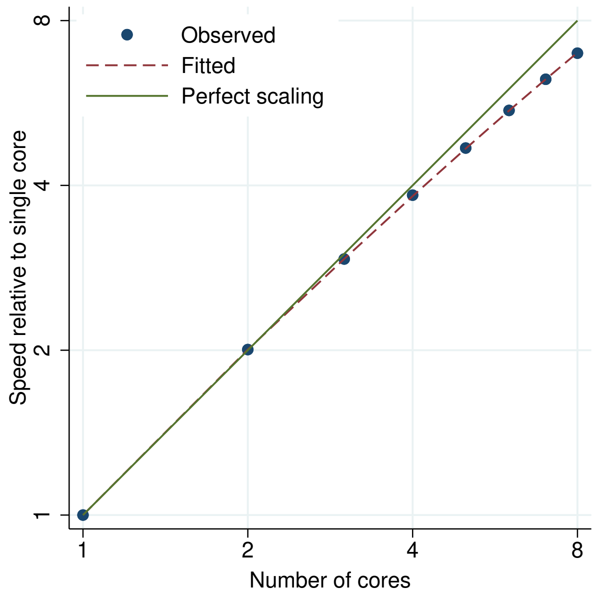


Figure 43. `betareg, link(logit)` performance plot.

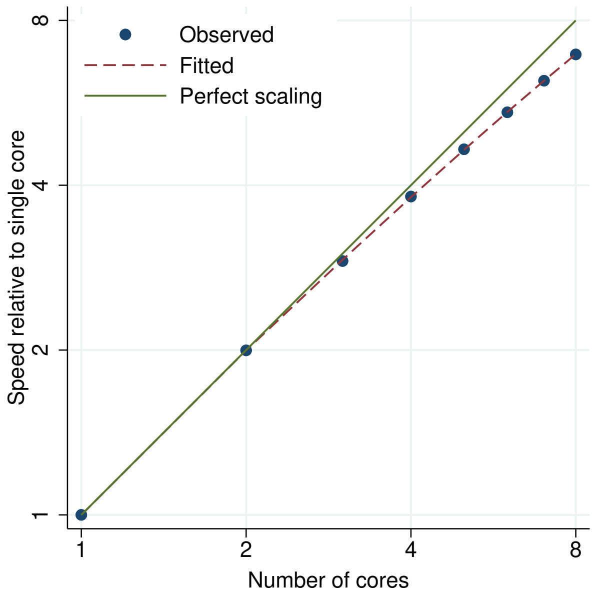


Figure 44. `betareg, link(probit)` performance plot.

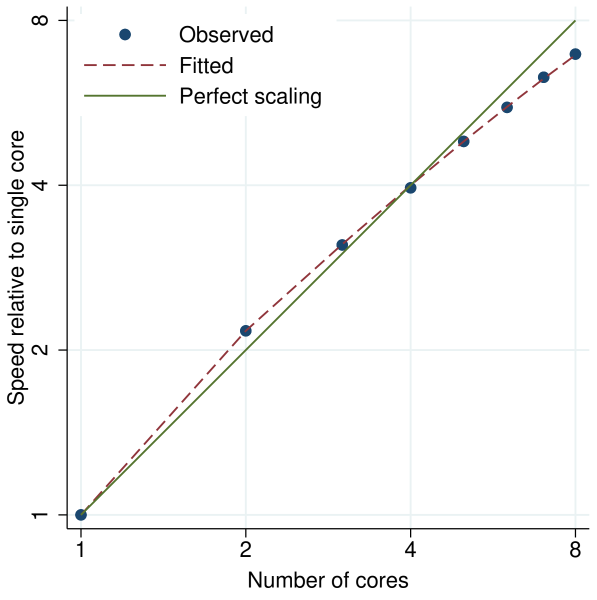


Figure 45. `binreg` performance plot.

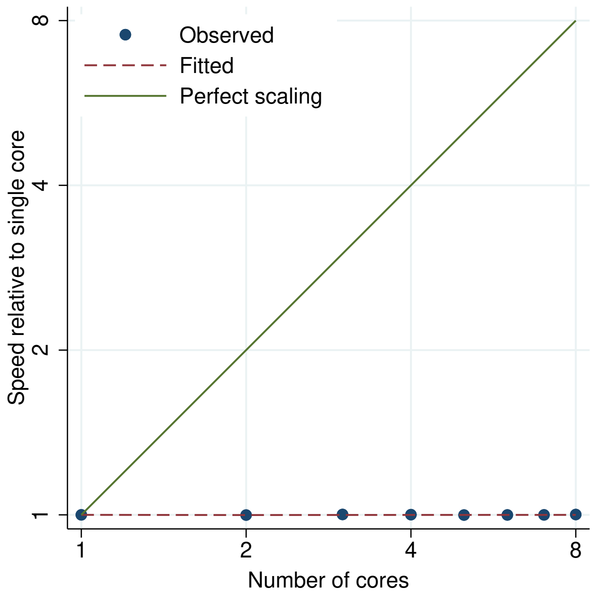


Figure 46. `biplot` performance plot.

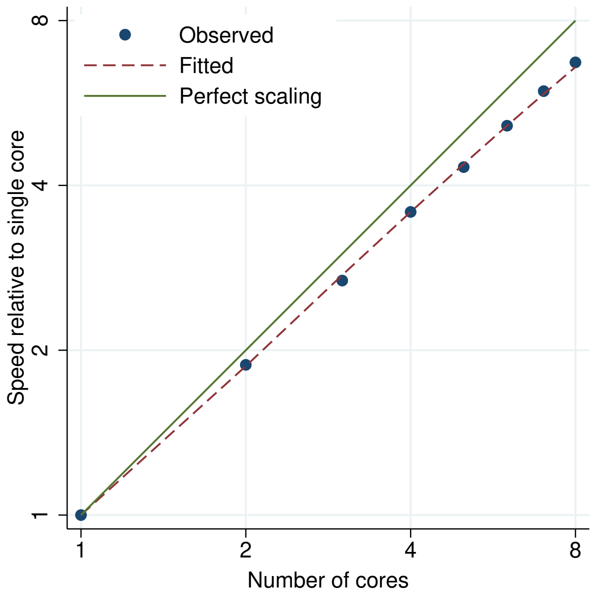


Figure 47. `biprobit` performance plot.

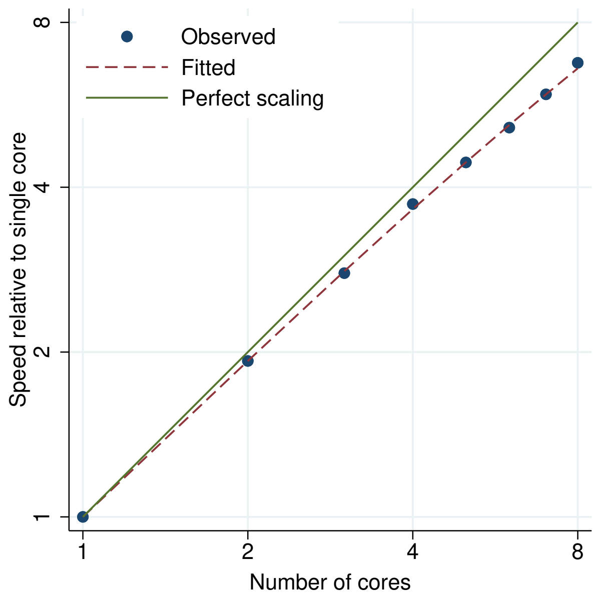


Figure 48. `biprobit` (seemingly unrelated) performance plot.

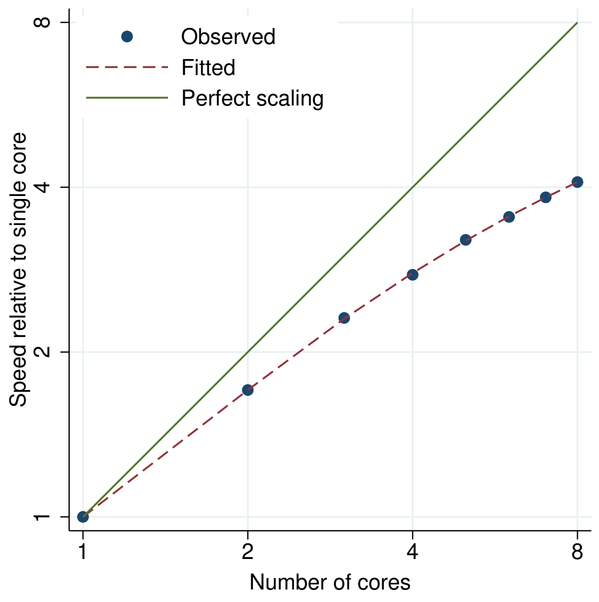


Figure 49. `bitest` performance plot.

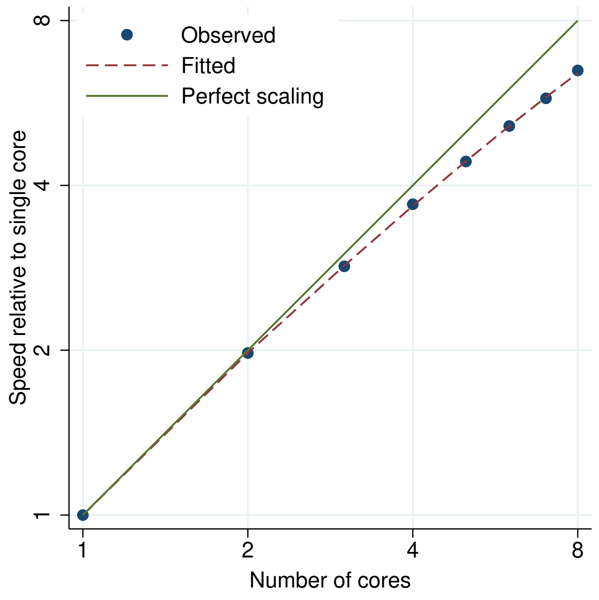


Figure 50. `blogit` performance plot.

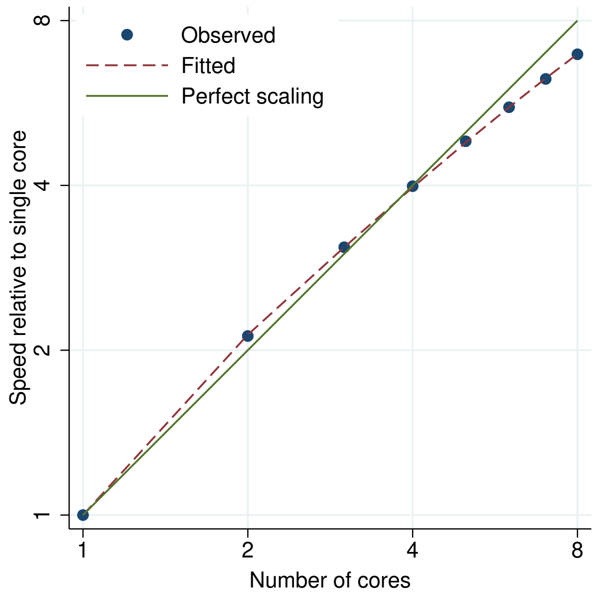


Figure 51. `boxcox` performance plot.

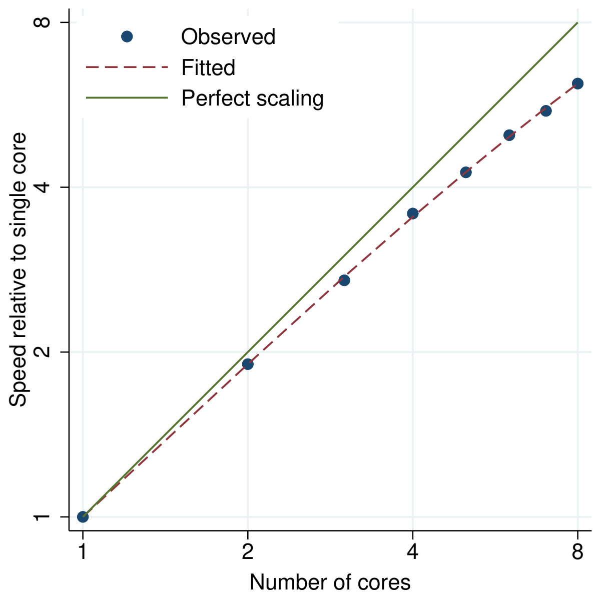


Figure 52. `bprobit` performance plot.

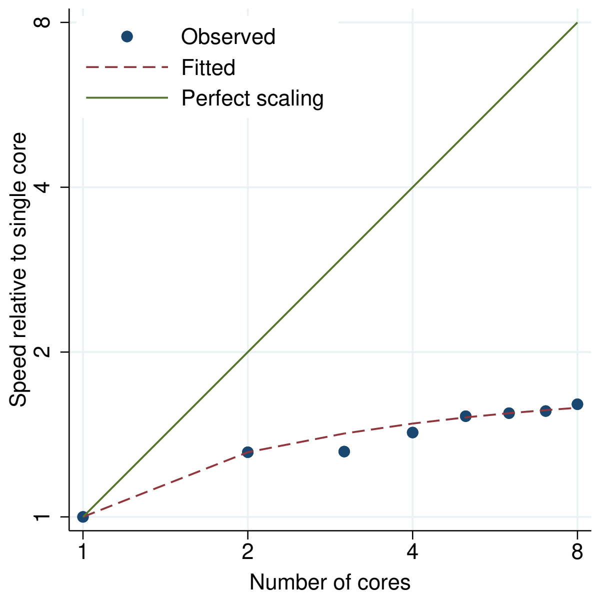


Figure 53. `brier` performance plot.

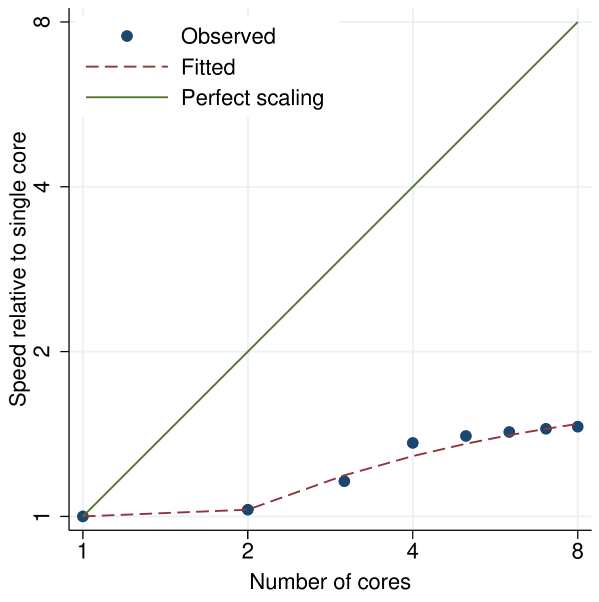


Figure 54. `bsample` performance plot.

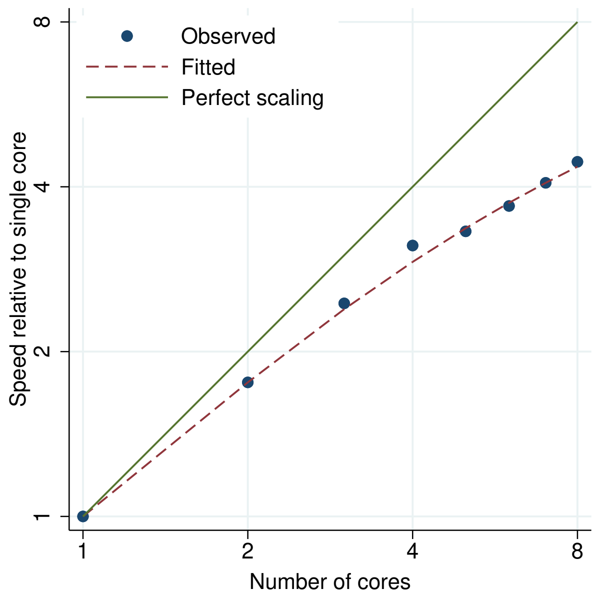


Figure 55. `bstat` performance plot.

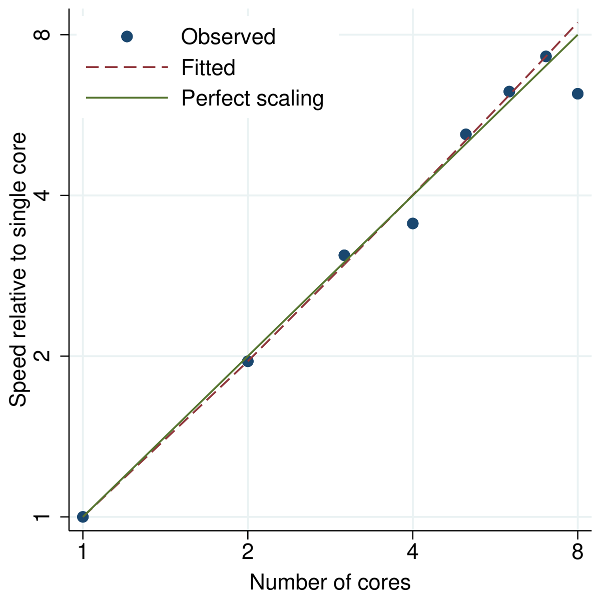


Figure 56. `by: generate` performance plot.

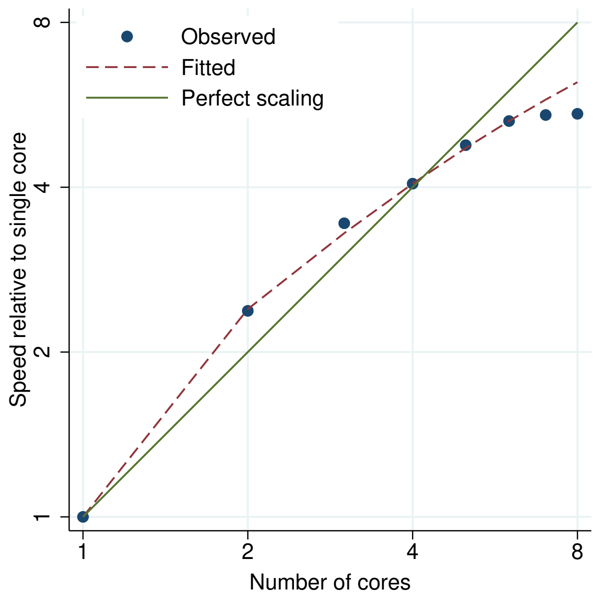


Figure 57. `by: generate` (small groups) performance plot.

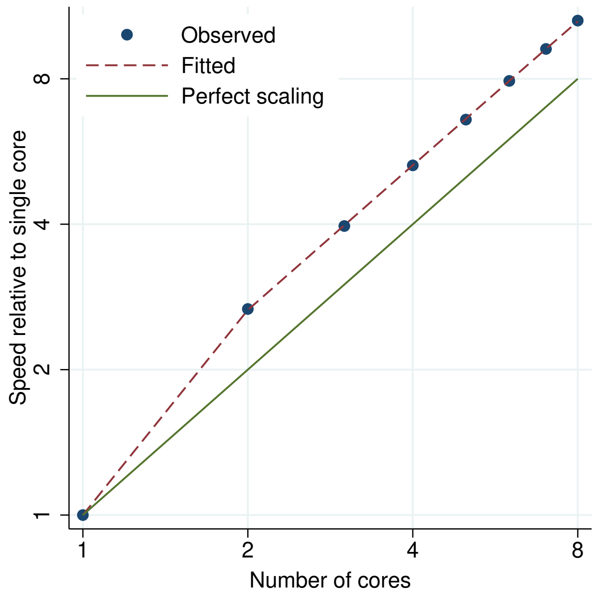


Figure 58. `by: replace` performance plot.

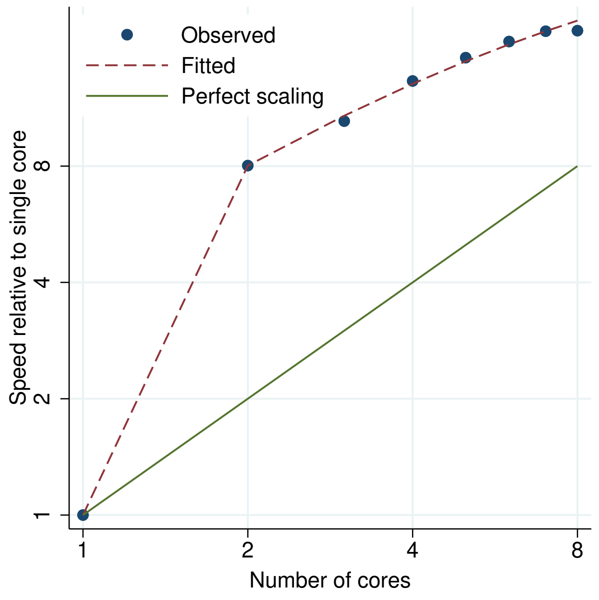


Figure 59. `by: replace` (small groups) performance plot.

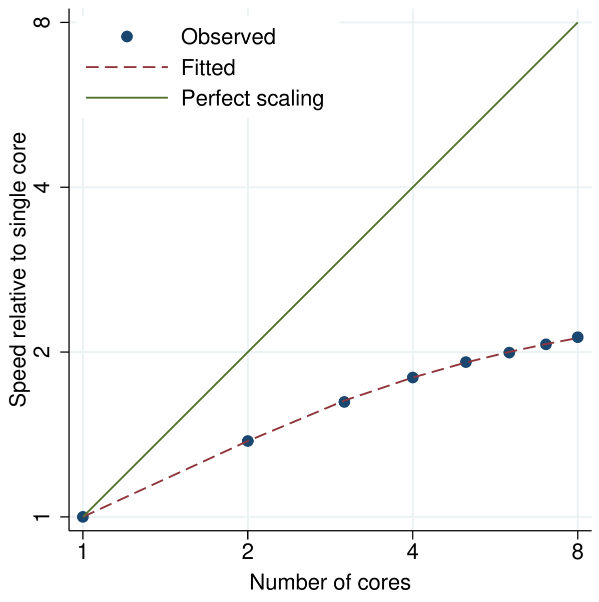


Figure 60. `ca` performance plot.

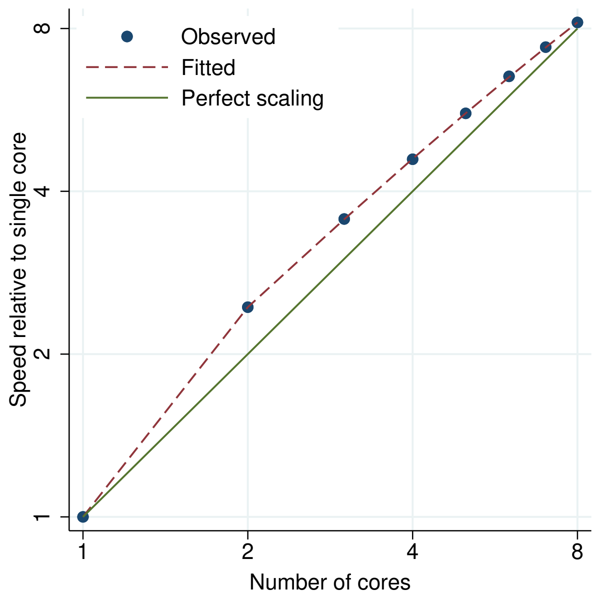


Figure 61. `candisc` performance plot.

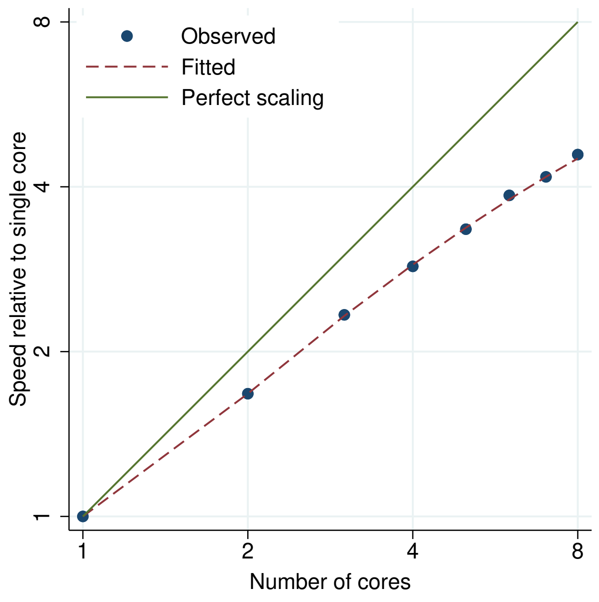


Figure 62. `canon` performance plot.

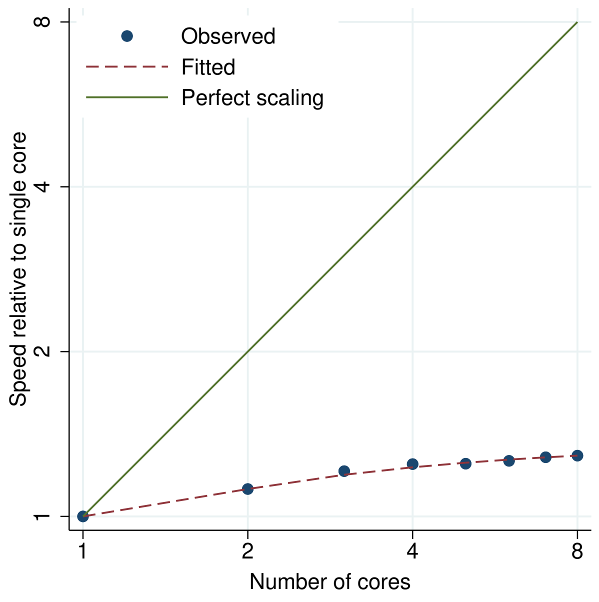


Figure 63. `cc` performance plot.

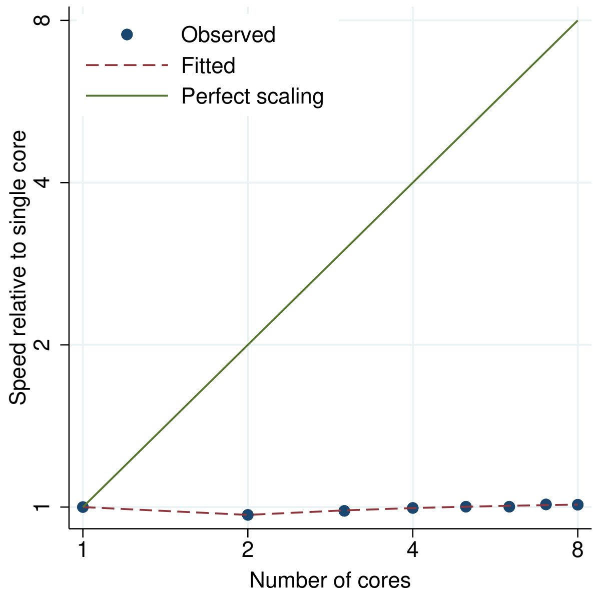


Figure 64. `by: cc` performance plot.

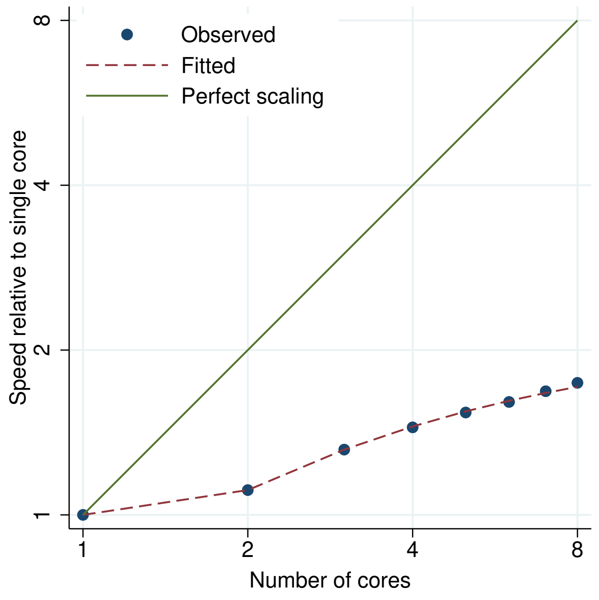


Figure 65. `centile` performance plot.

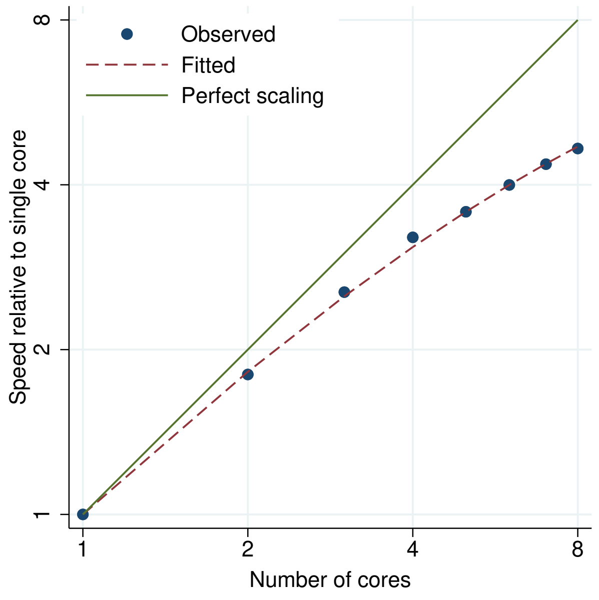


Figure 66. `churdle linear` performance plot.

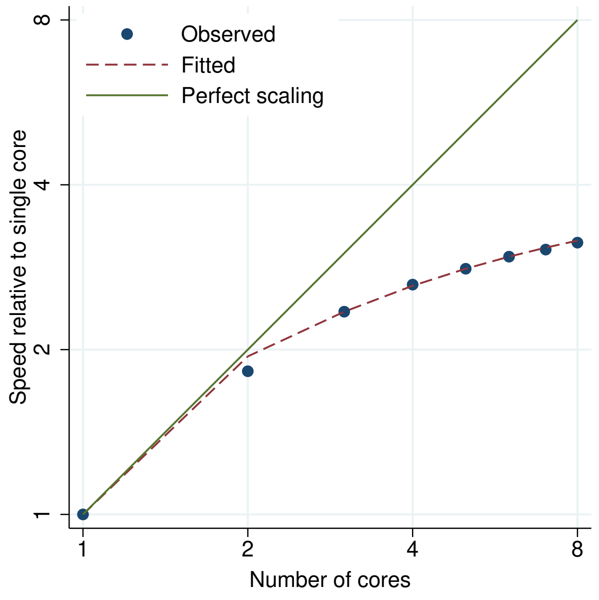


Figure 67. `ci means` performance plot.

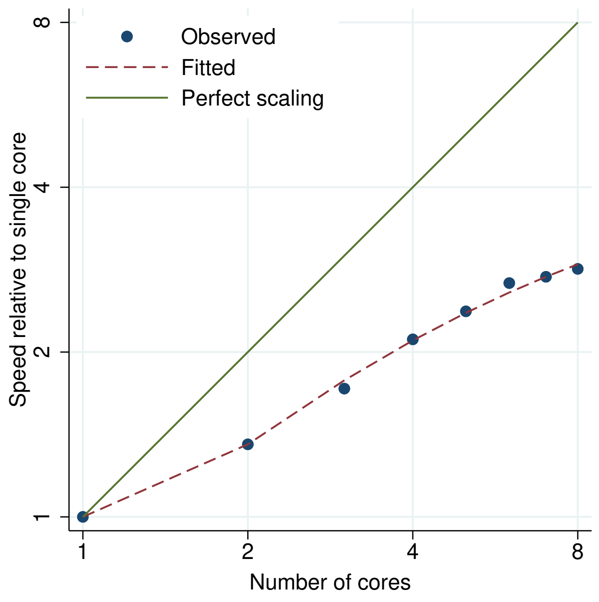


Figure 68. `ci means, poisson` performance plot.

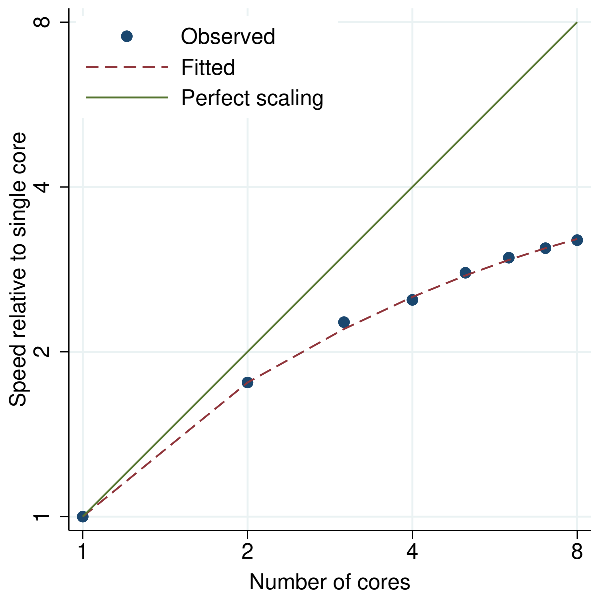


Figure 69. `ci proportions` performance plot.

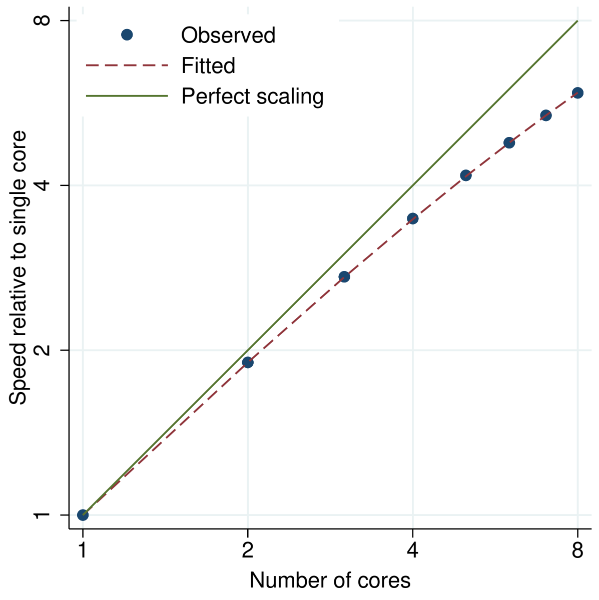


Figure 70. `clogit` (k1 to k2 matching) performance plot.

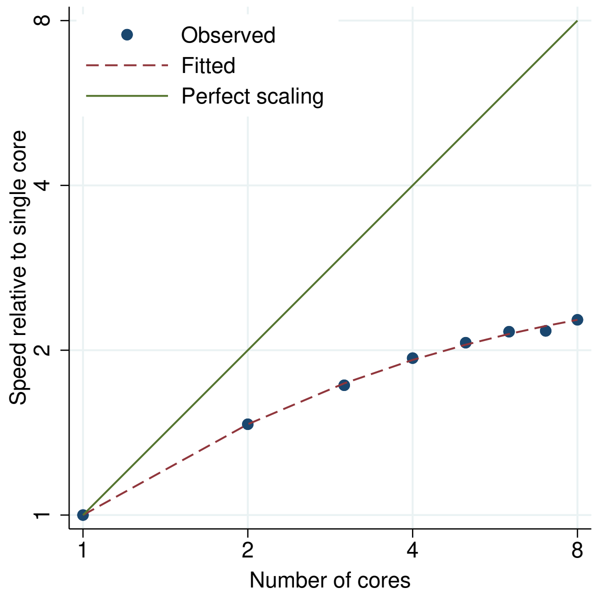


Figure 71. `clogit` (1 to k matching) performance plot.

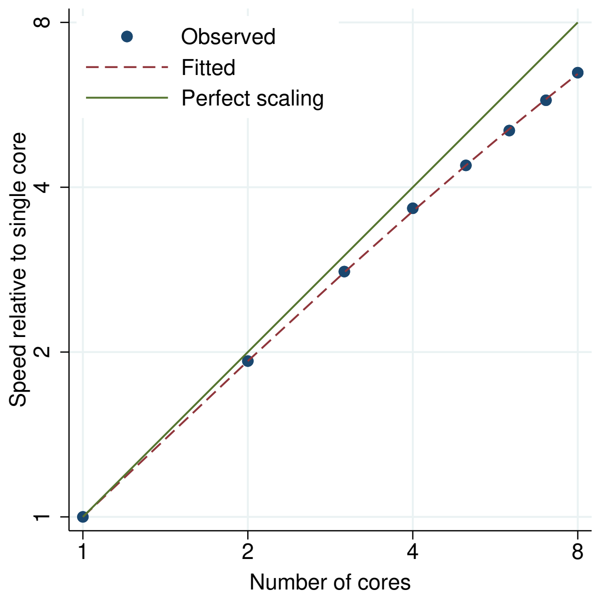


Figure 72. `cloglog` performance plot.

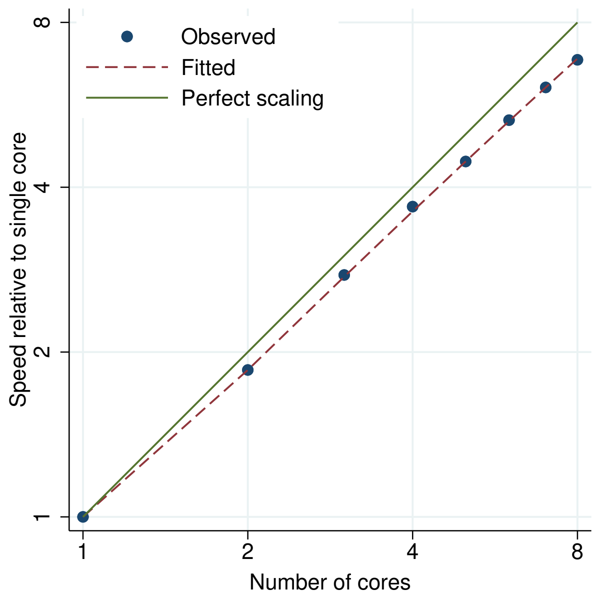


Figure 73. `cluster averagelinkage` performance plot.

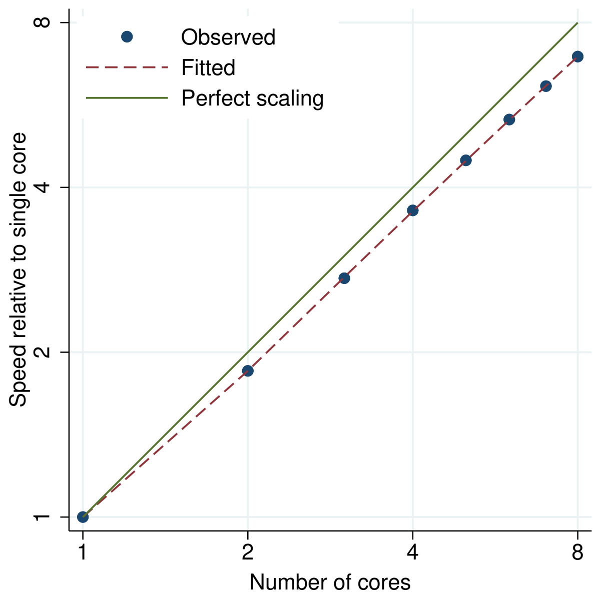


Figure 74. `cluster centroidlinkage` performance plot.

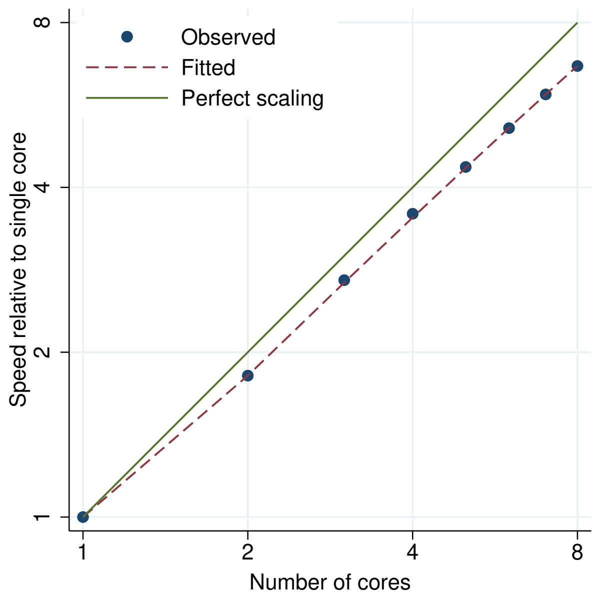


Figure 75. `cluster completelinkage` performance plot.

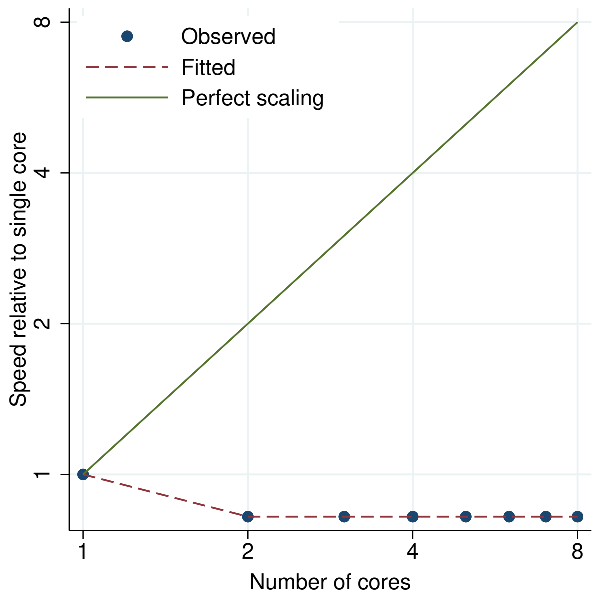


Figure 76. `cluster generate` performance plot.

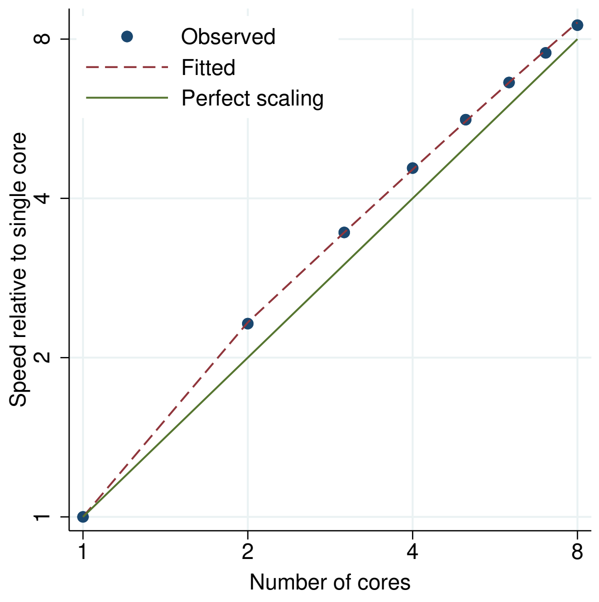


Figure 77. `cluster kmeans` performance plot.

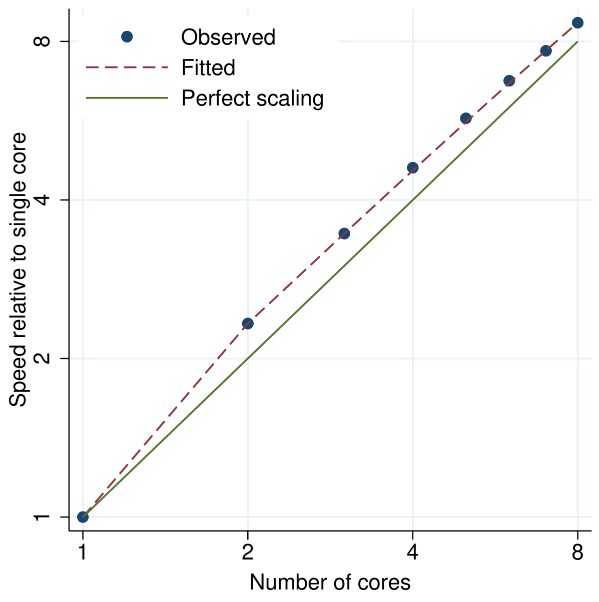


Figure 78. `cluster kmedians` performance plot.

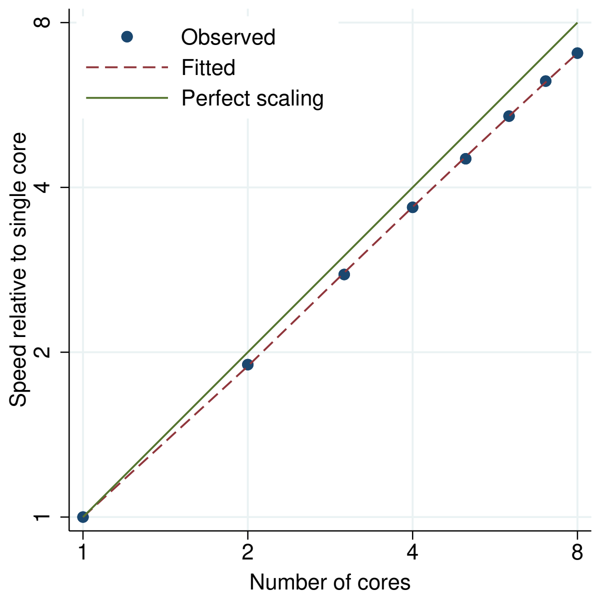


Figure 79. `cluster medianlinkage` performance plot.

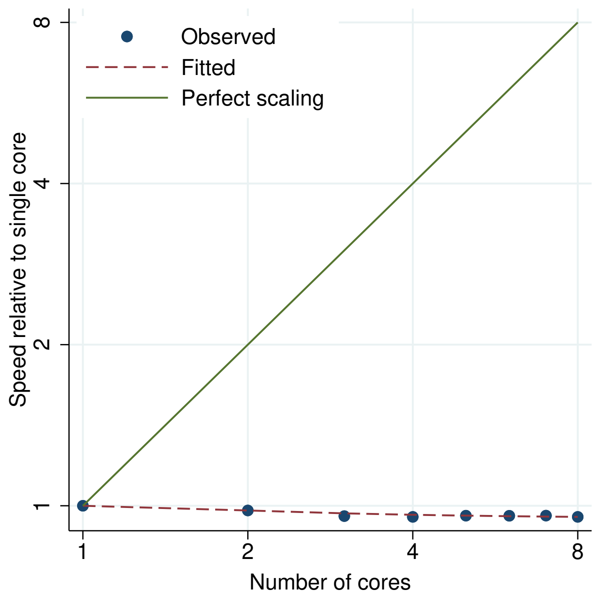


Figure 80. `cluster singlelinkage` performance plot.

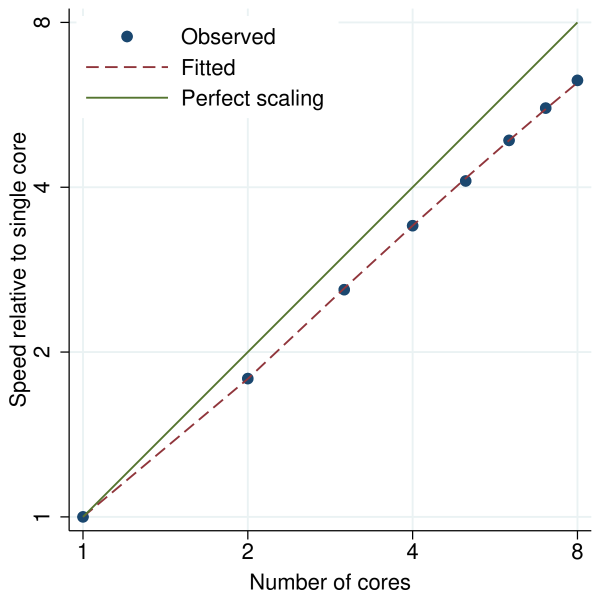


Figure 81. `cluster wardslinkage` performance plot.

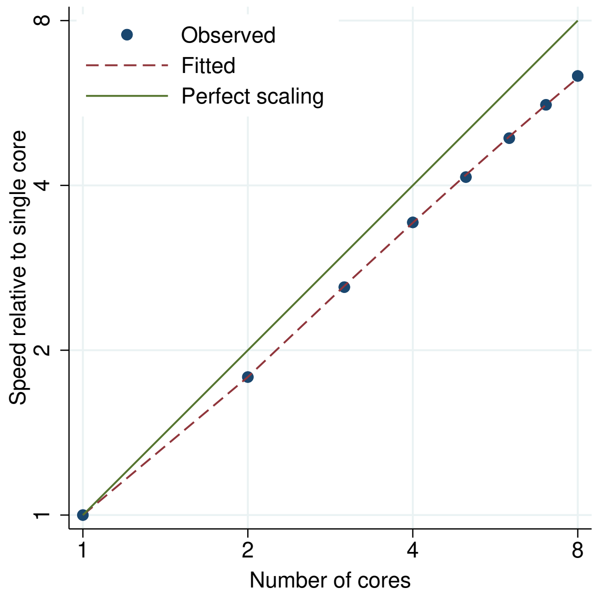


Figure 82. `cluster waveragelinkage` performance plot.

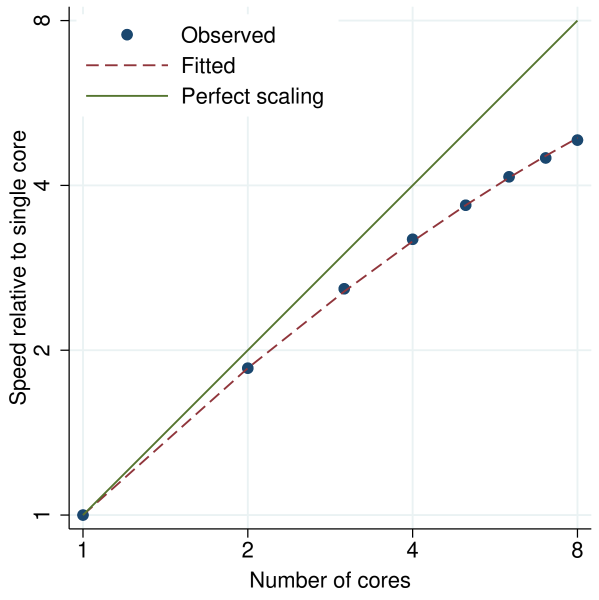


Figure 83. `cmclogit` performance plot.

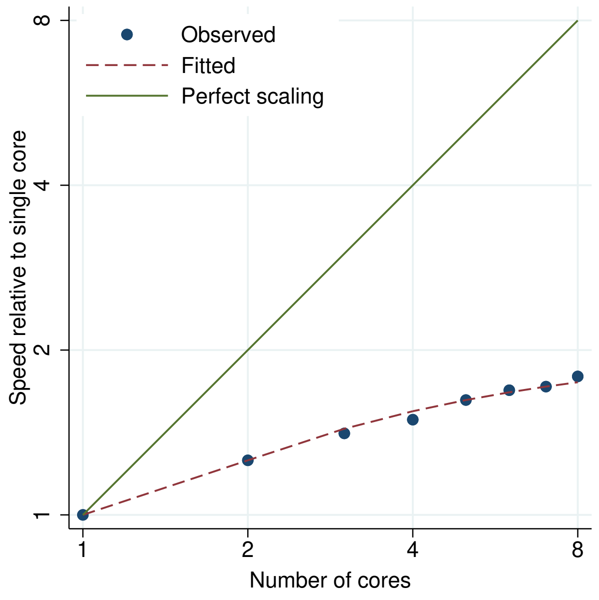


Figure 84. `cmmprobit` performance plot.

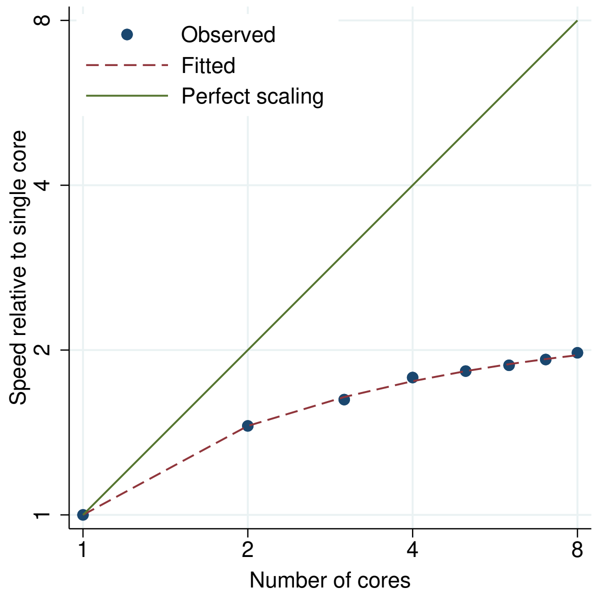


Figure 85. `cmroprobit` performance plot.

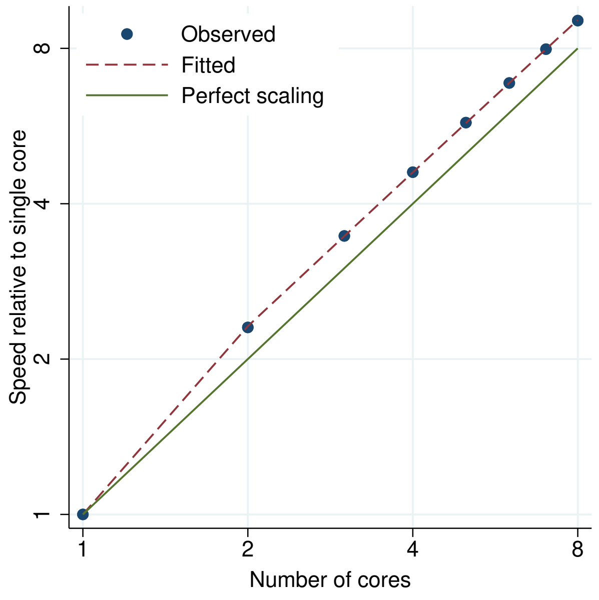


Figure 86. `cnsreg` performance plot.

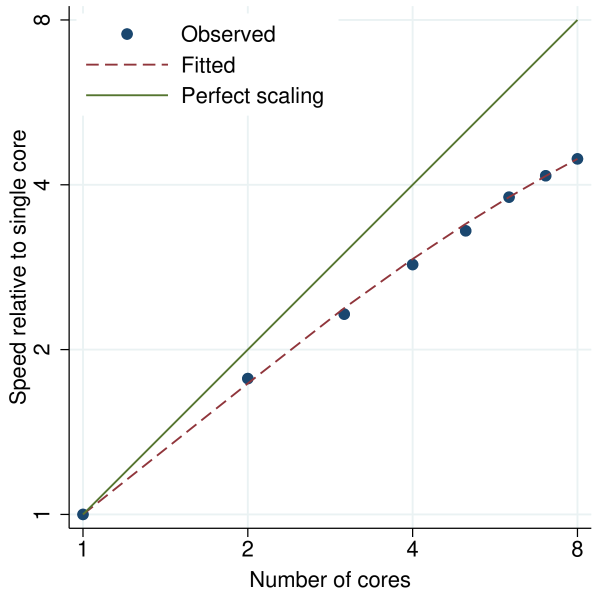


Figure 87. `codebook` performance plot.

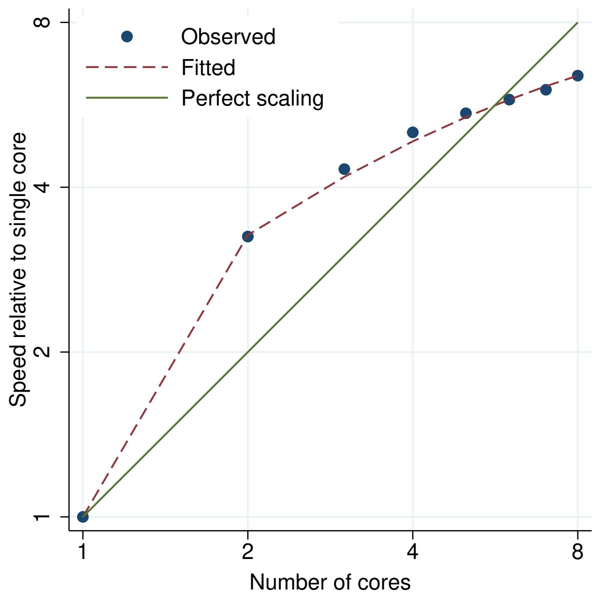


Figure 88. `collapse` performance plot.

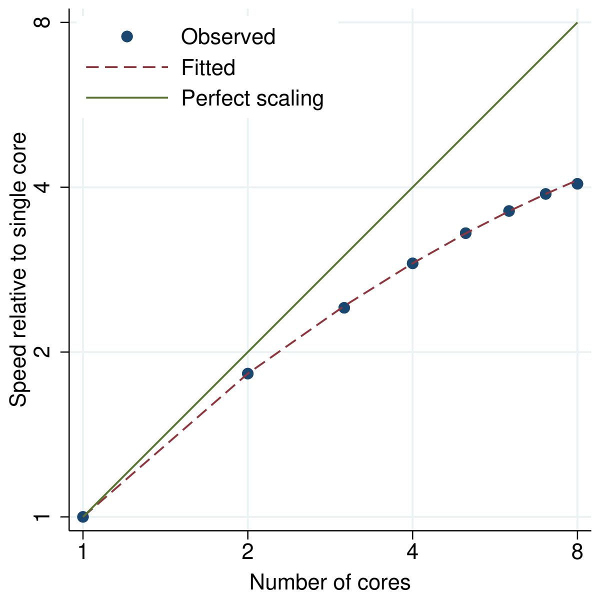


Figure 89. `compare` performance plot.

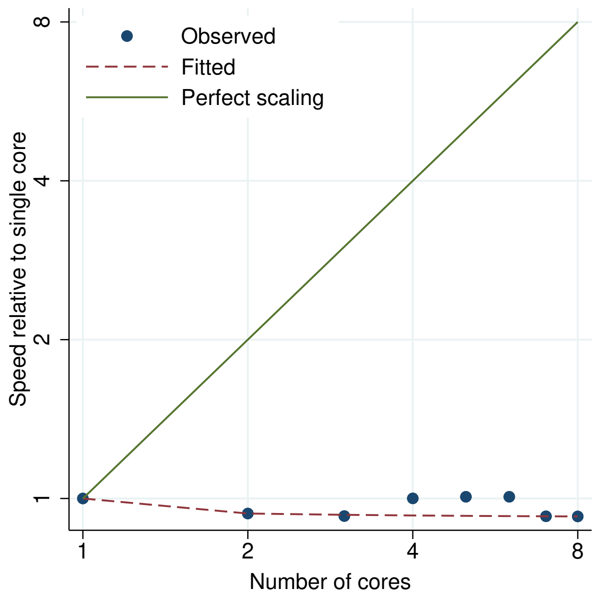


Figure 90. `compress` performance plot.

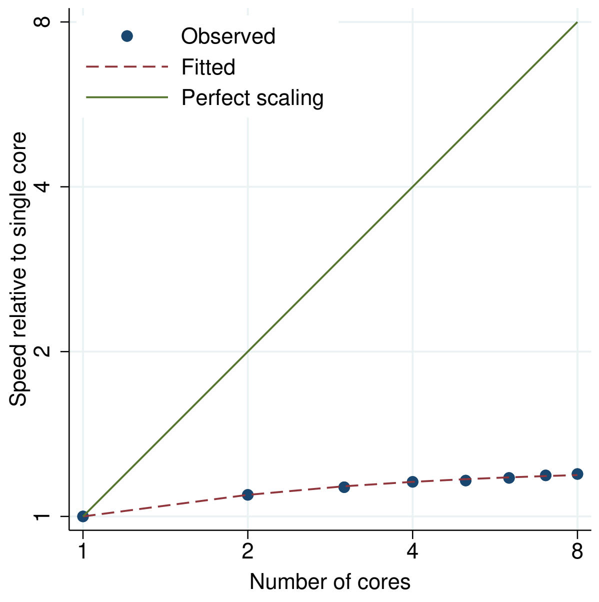


Figure 91. `contract` performance plot.

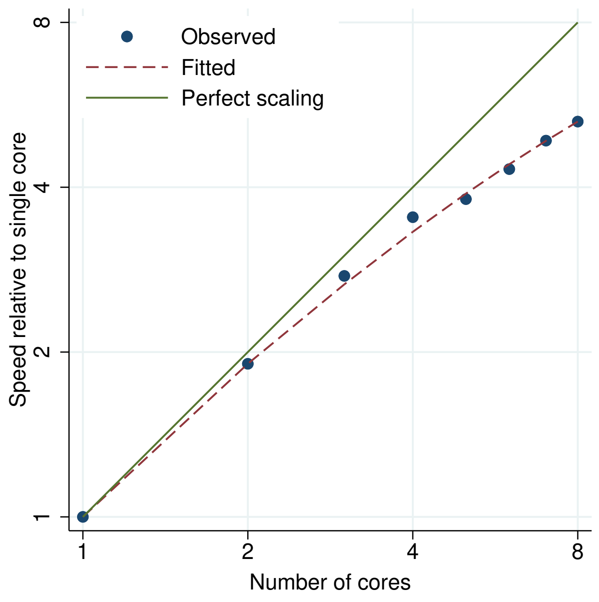


Figure 92. `corr2data` performance plot.

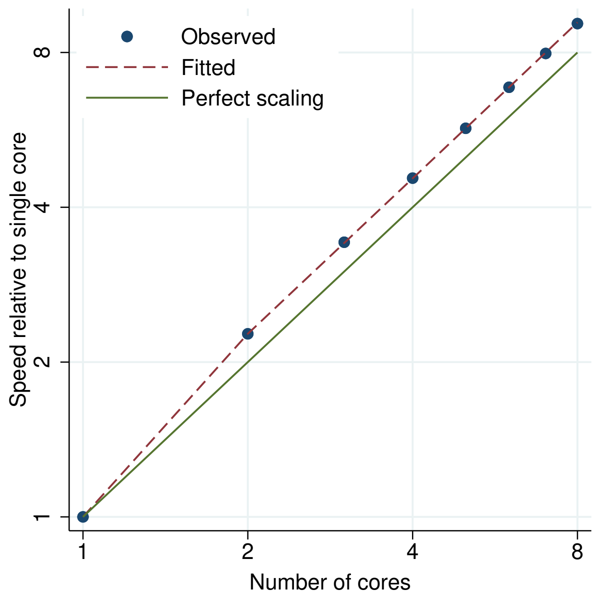


Figure 93. `correlate` performance plot.

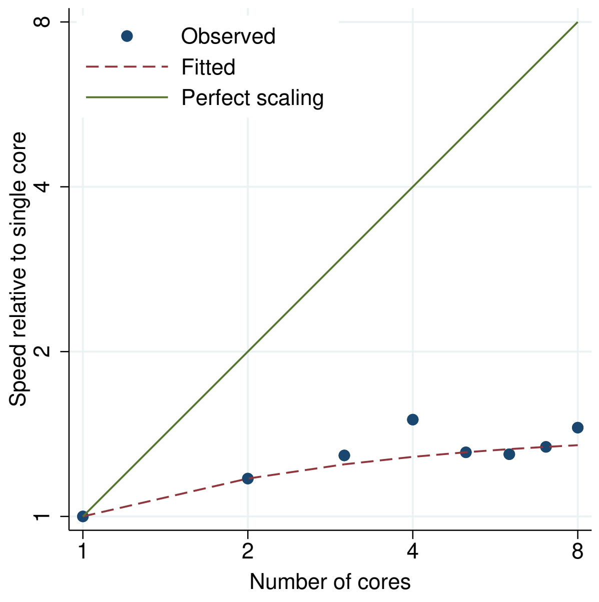


Figure 94. `corrgram` performance plot.

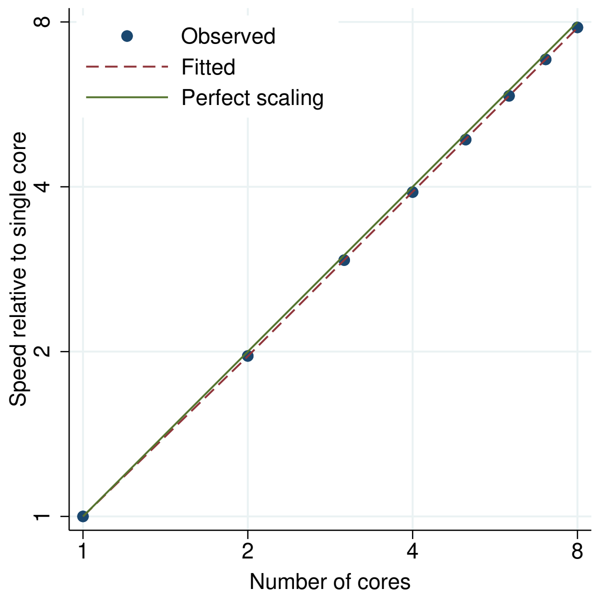


Figure 95. `count` performance plot.

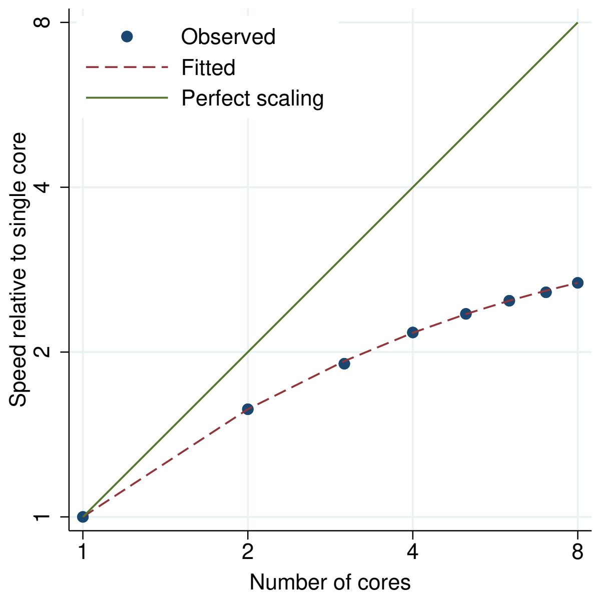


Figure 96. `cpoisson` performance plot.

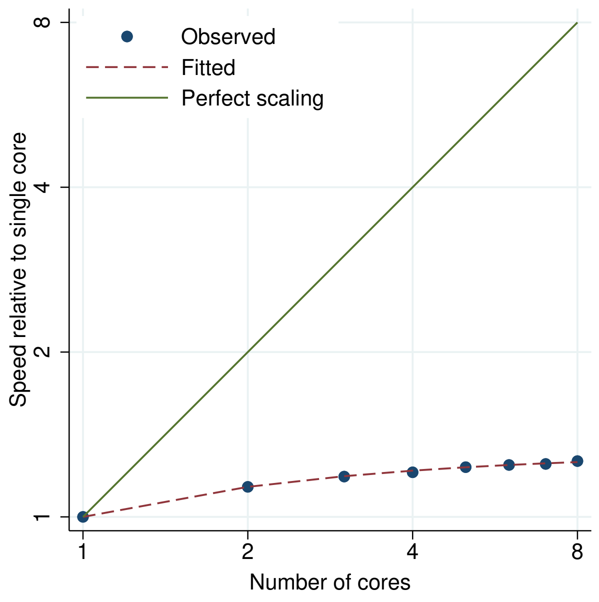


Figure 97. `cs` performance plot.

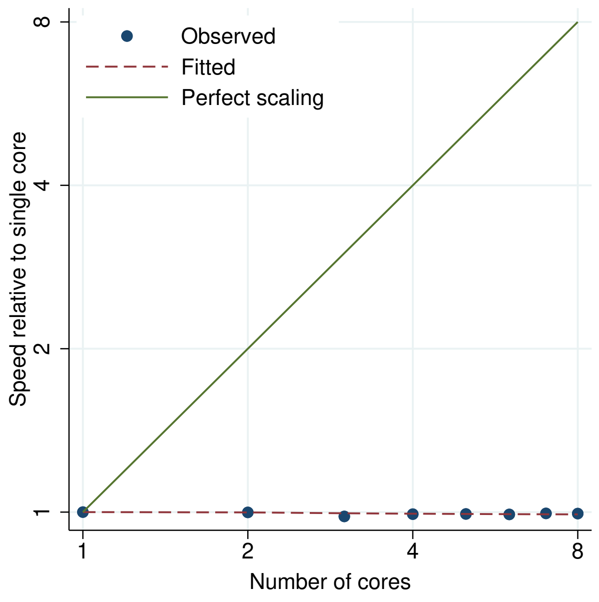


Figure 98. `by: cs` performance plot.

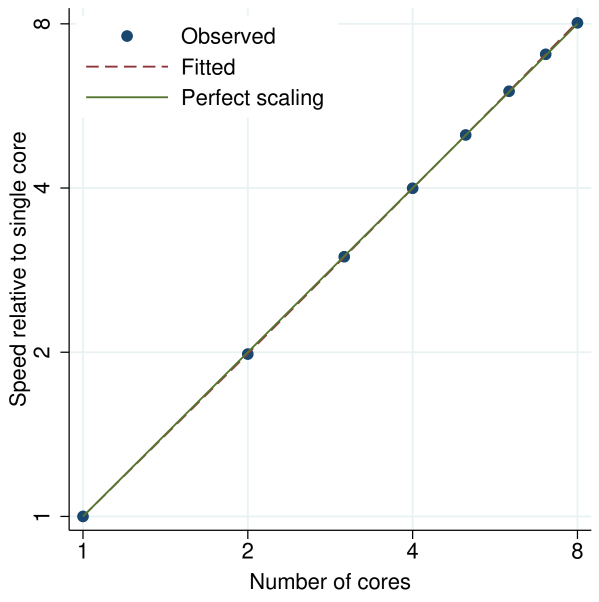


Figure 99. `ctset` performance plot.

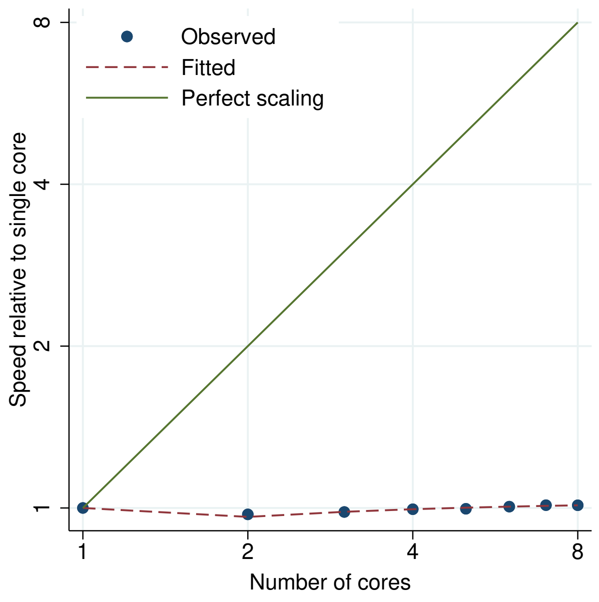


Figure 100. `cttost` performance plot.

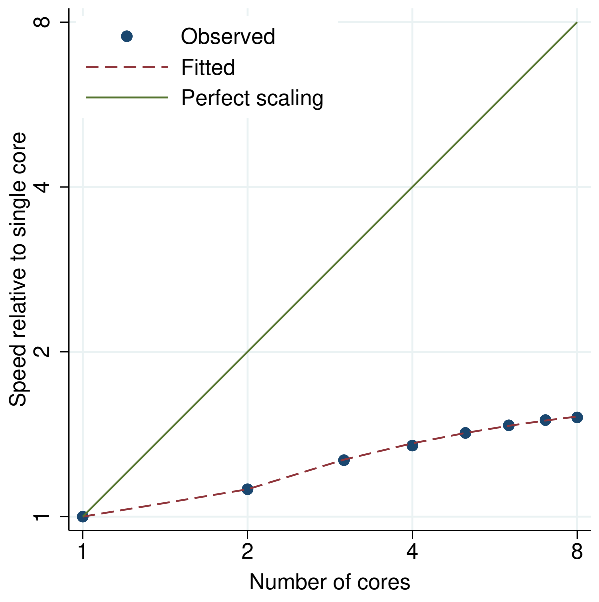


Figure 101. `cumul` performance plot.

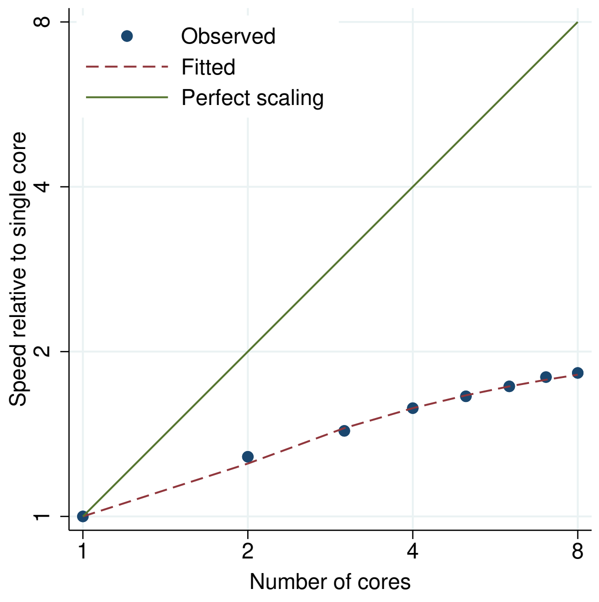


Figure 102. `cusum` performance plot.

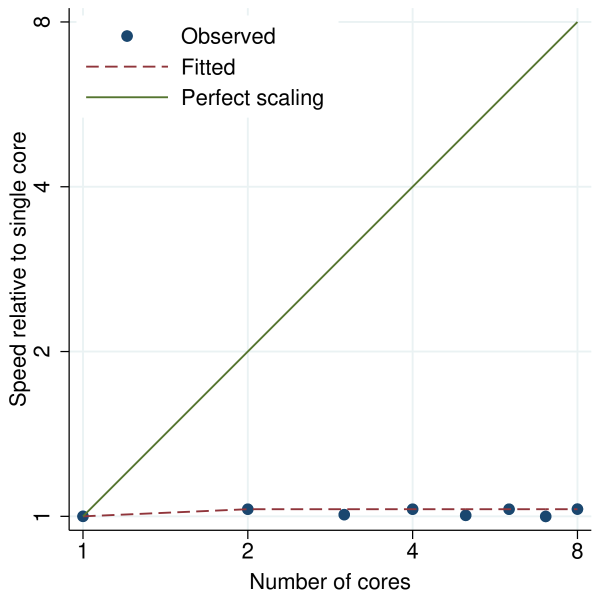


Figure 103. `datasignature` performance plot.

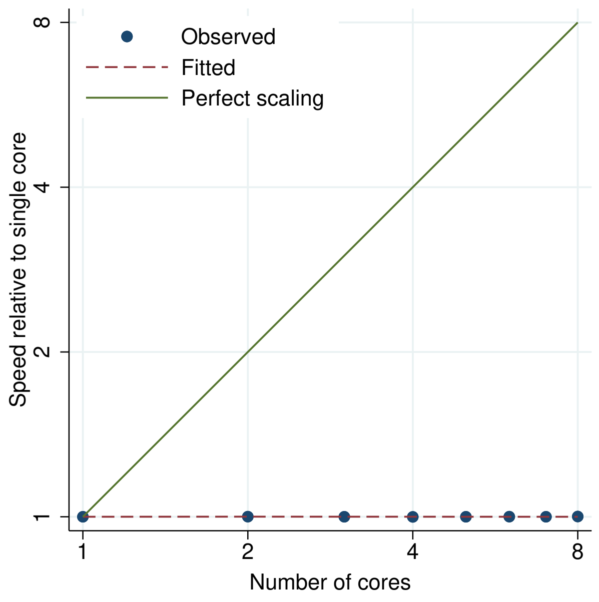


Figure 104. `decode` performance plot.

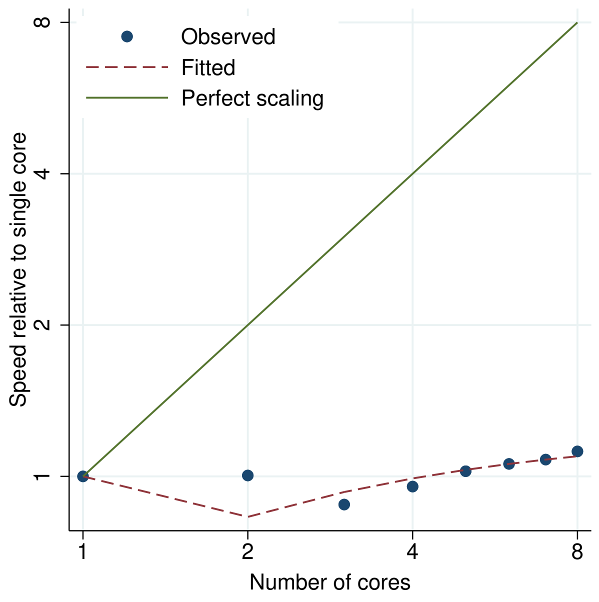


Figure 105. `destring` performance plot.

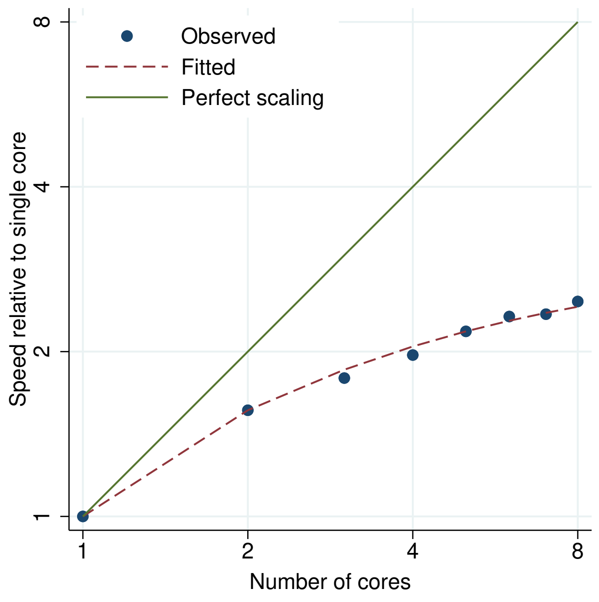


Figure 106. `dfactor` performance plot.

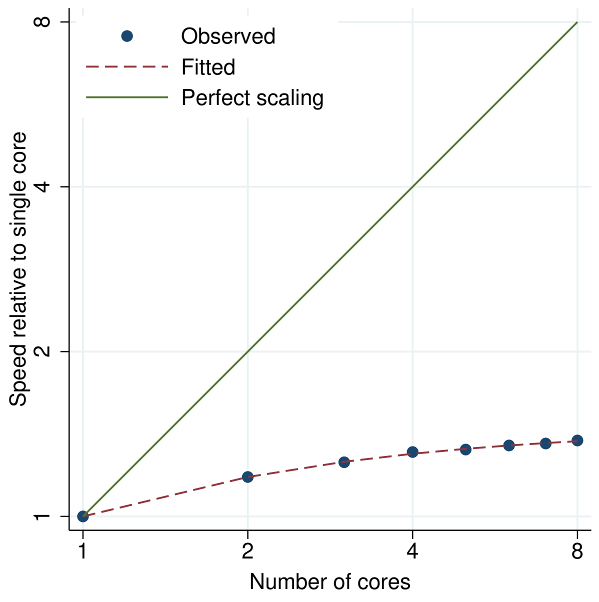


Figure 107. `dfgls` performance plot.

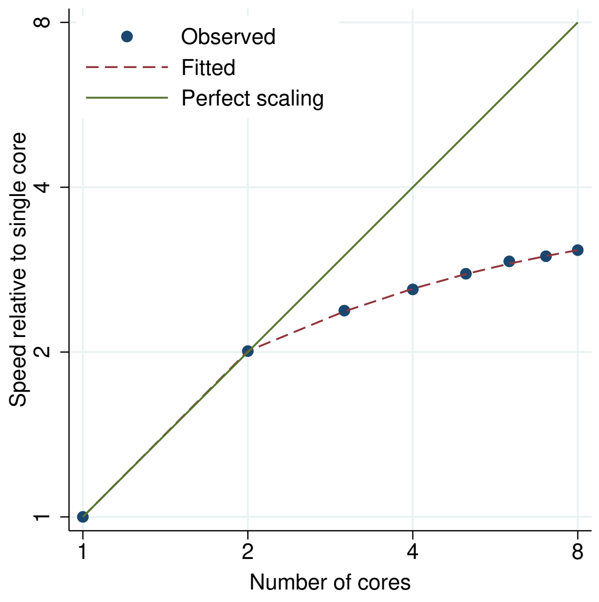


Figure 108. `dfuller` performance plot.

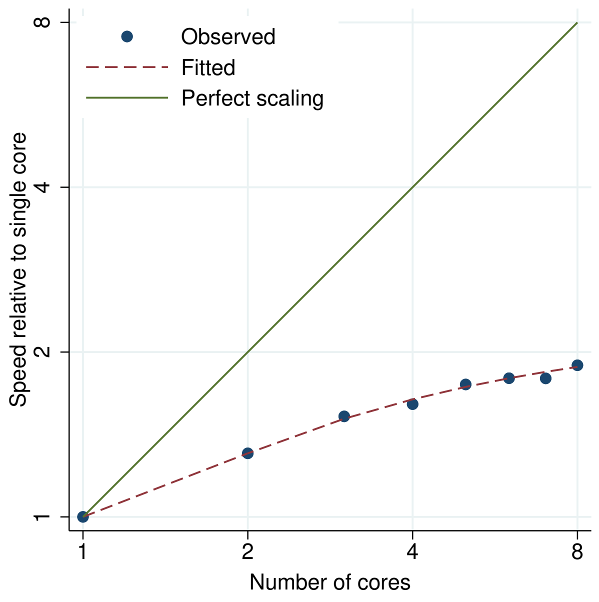


Figure 109. `didregress` performance plot.

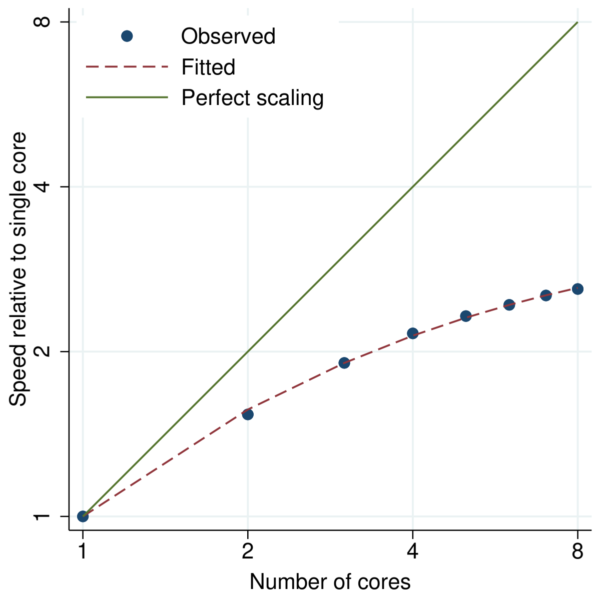


Figure 110. `discrim knn` performance plot.

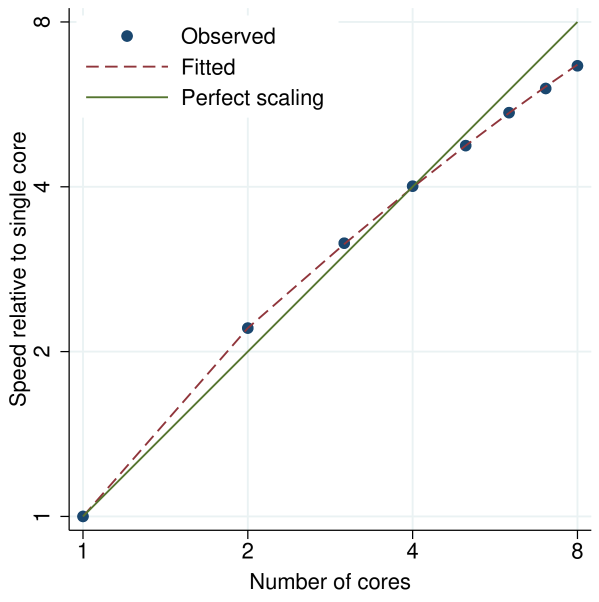


Figure 111. `discrim lda` performance plot.

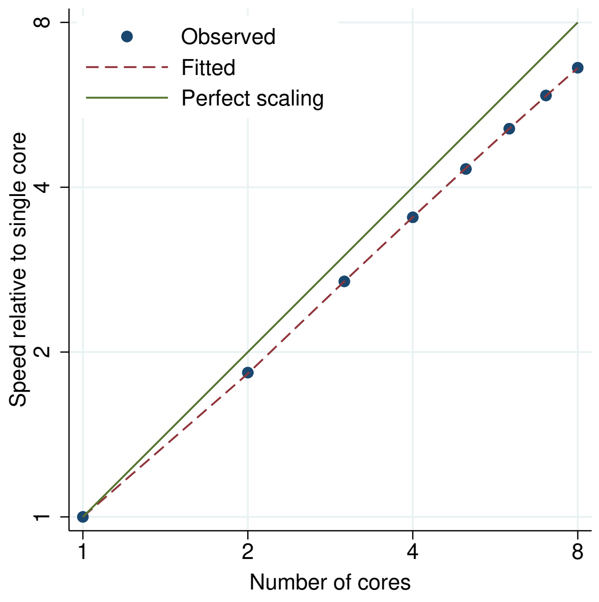


Figure 112. `discrim logistic` performance plot.

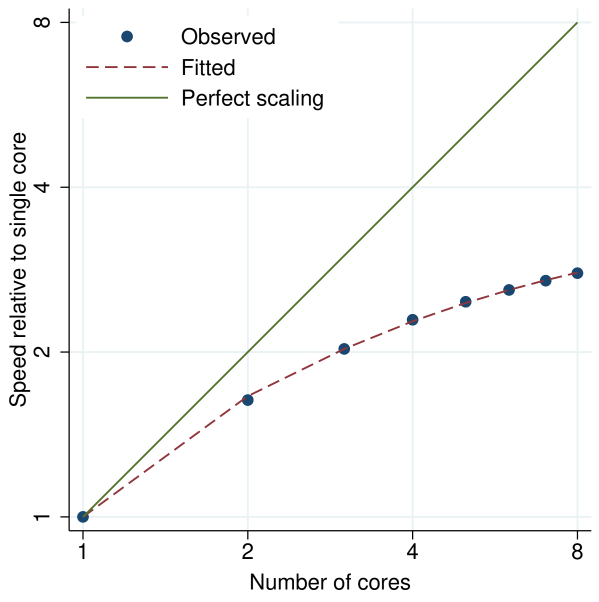


Figure 113. `discrim qda` performance plot.

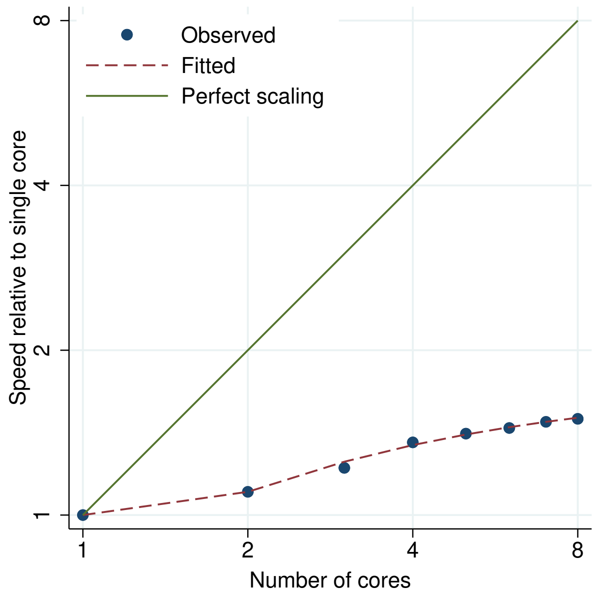


Figure 114. `dotplot` performance plot.

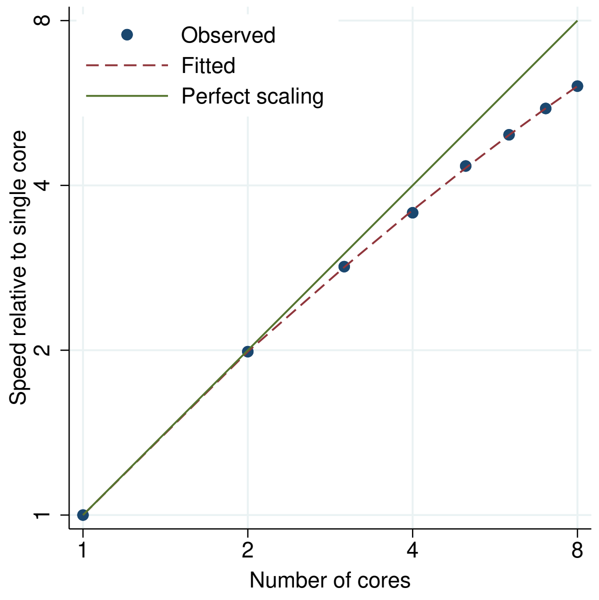


Figure 115. `drawnorm` performance plot.

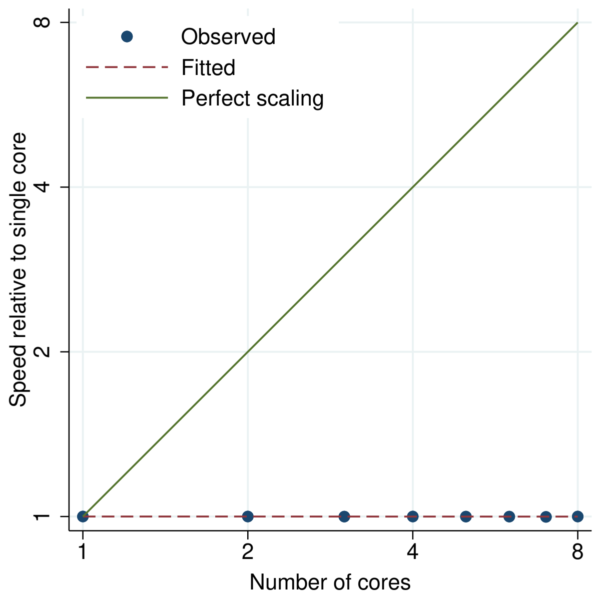


Figure 116. `drop if` *exp* performance plot.

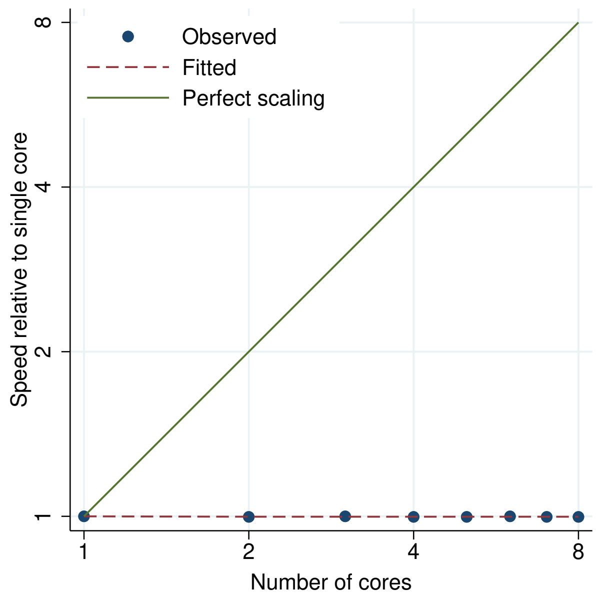


Figure 117. `drop in` *range* performance plot.

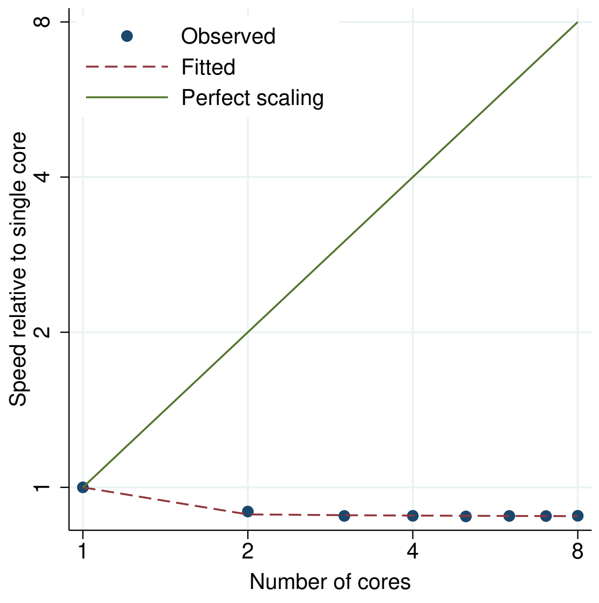


Figure 118. `dsge` performance plot.

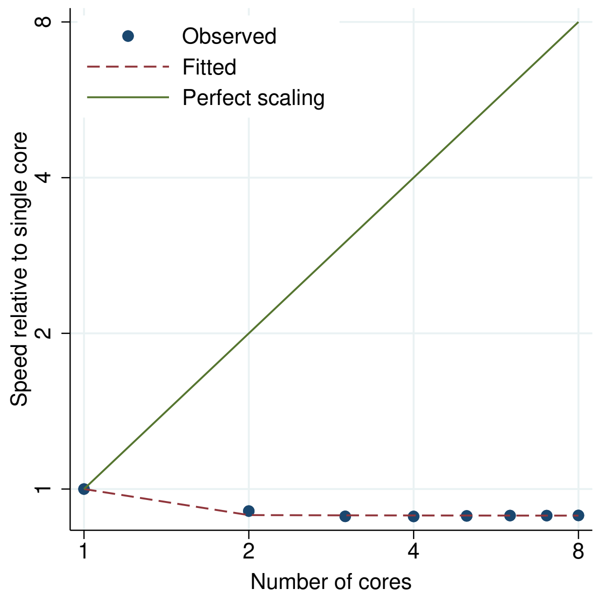


Figure 119. `dsgenl` performance plot.

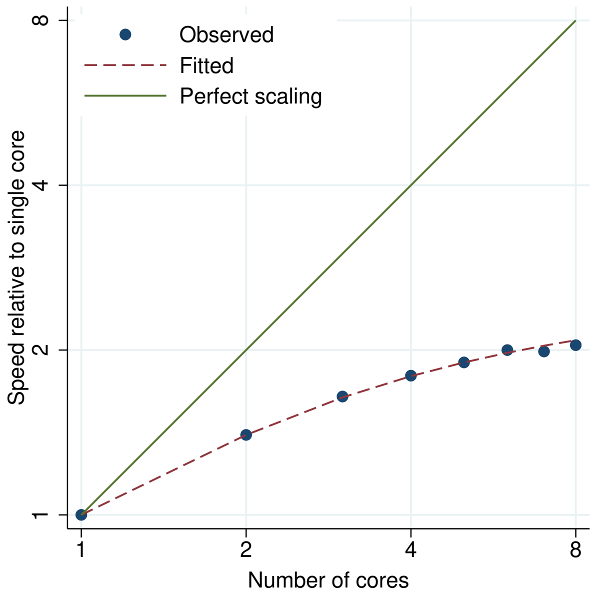


Figure 120. `dslogit` performance plot.

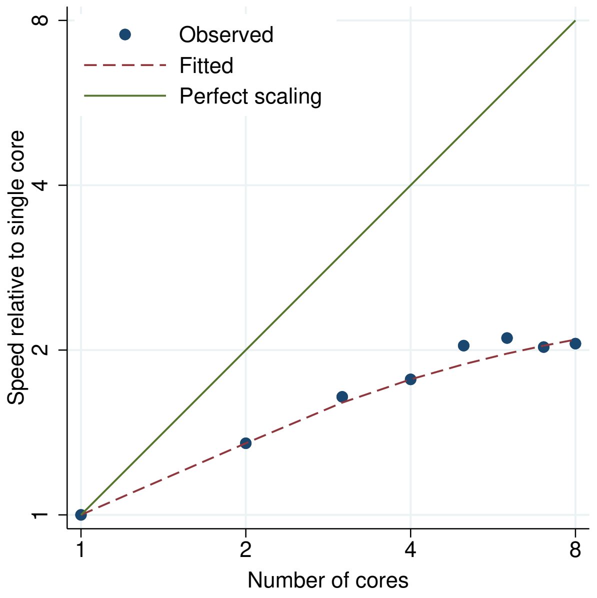


Figure 121. `dspoisson` performance plot.

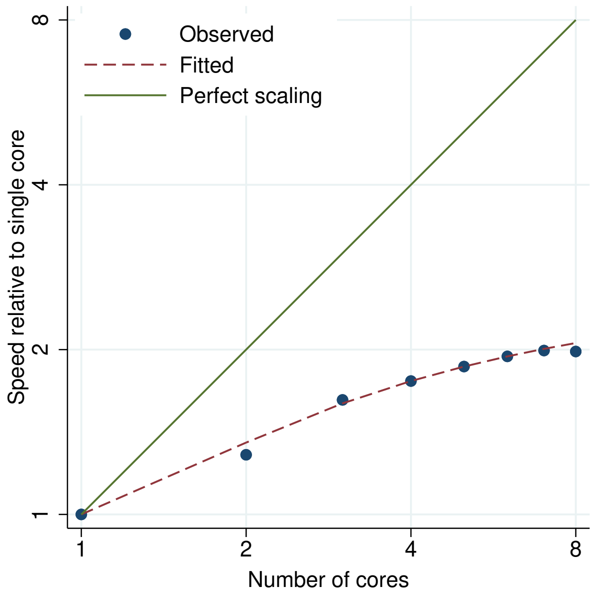


Figure 122. `dsregress` performance plot.

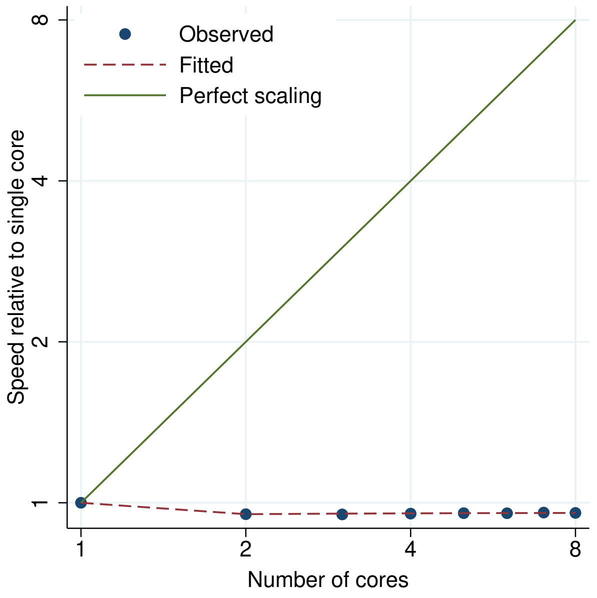


Figure 123. `dstdize` performance plot.

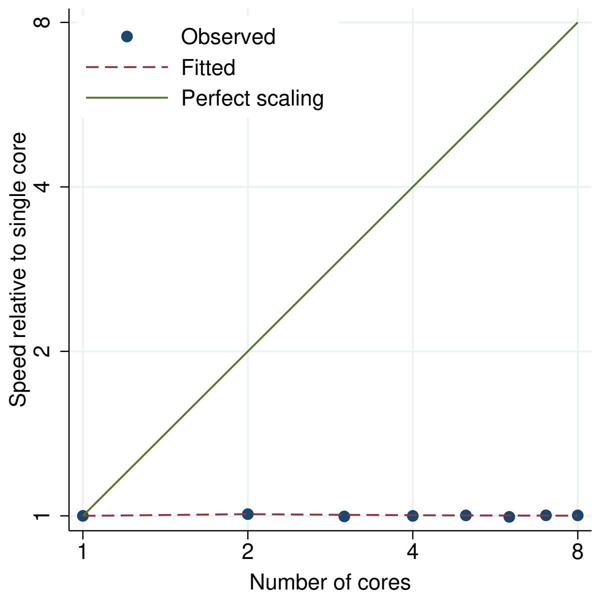


Figure 124. `dvech` performance plot.

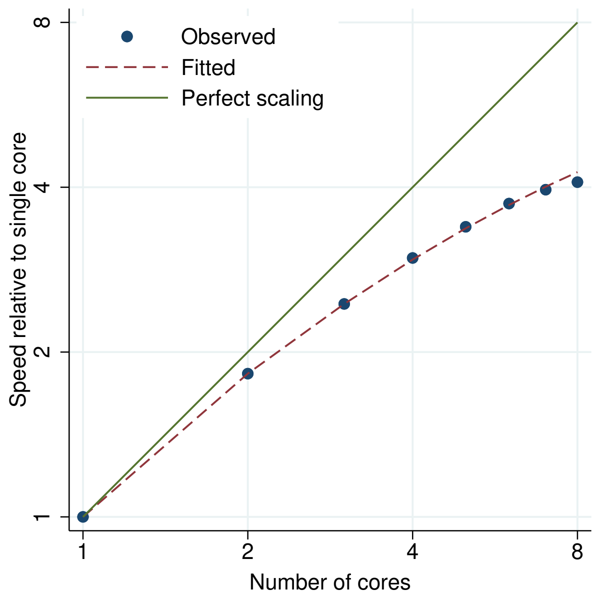


Figure 125. `egen group()` performance plot.

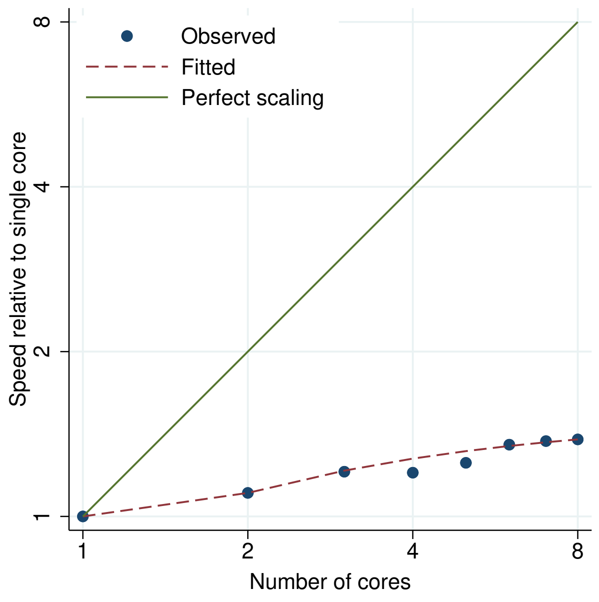


Figure 126. `by: egen mean` performance plot.

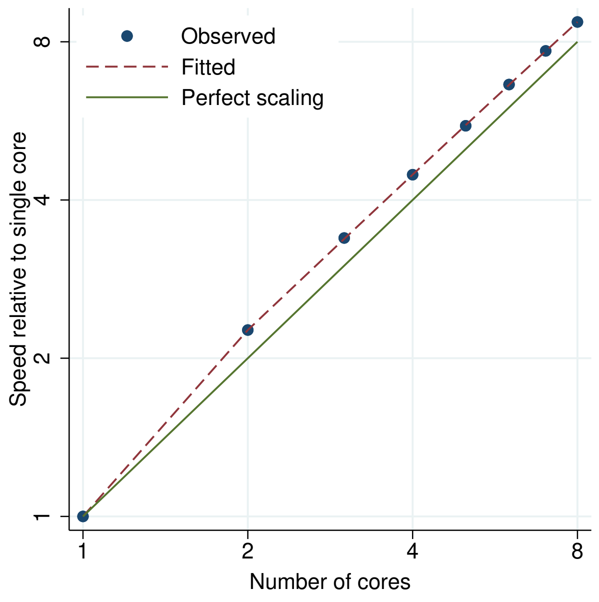


Figure 127. `eivreg` performance plot.

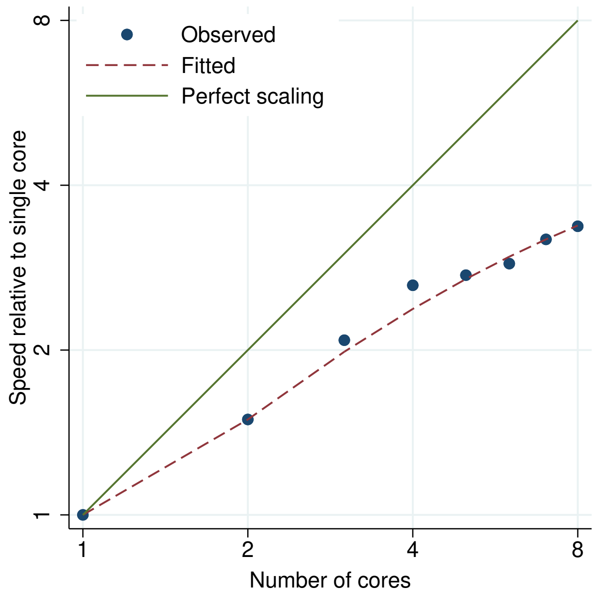


Figure 128. `encode` performance plot.

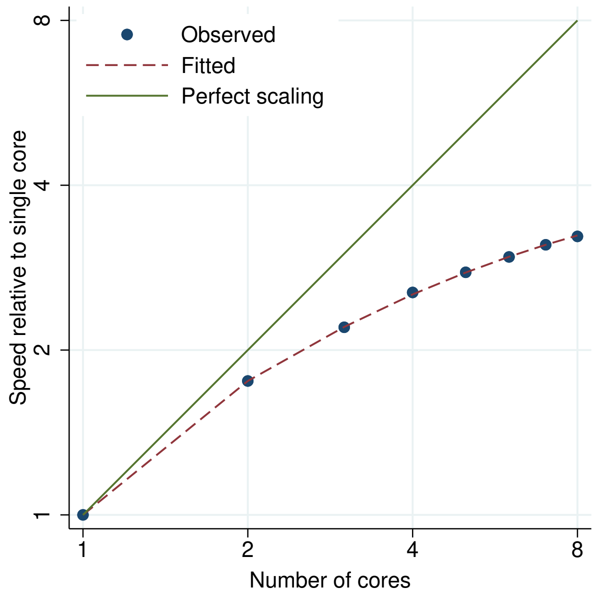


Figure 129. `eregress` performance plot.

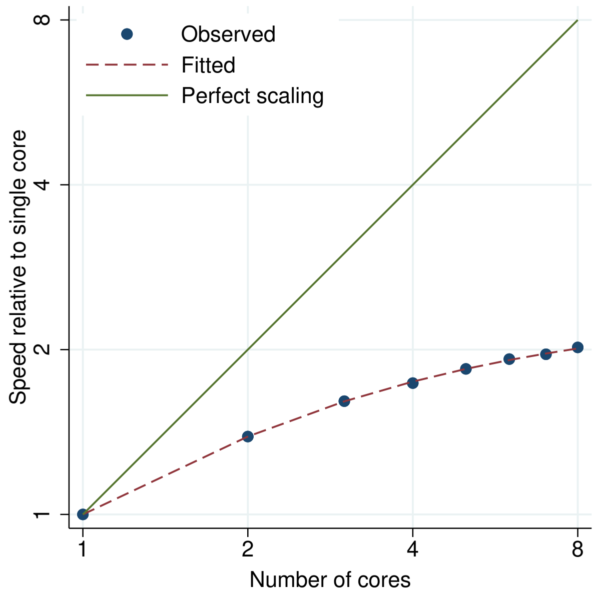


Figure 130. `esize twosample` performance plot.

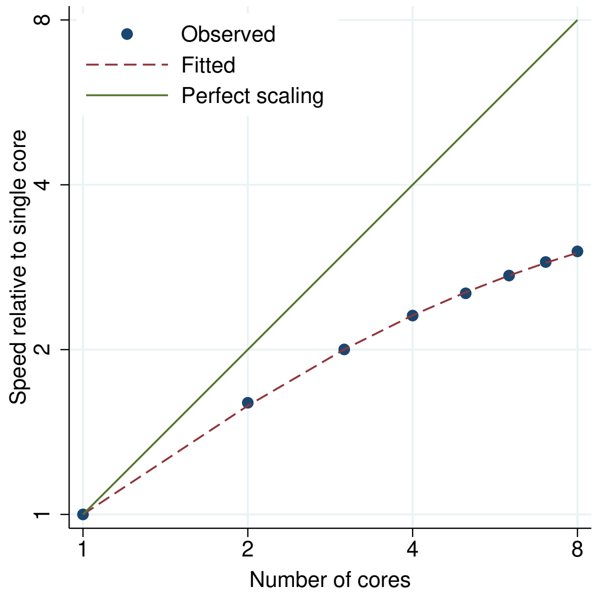


Figure 131. `esize unpaired` performance plot.

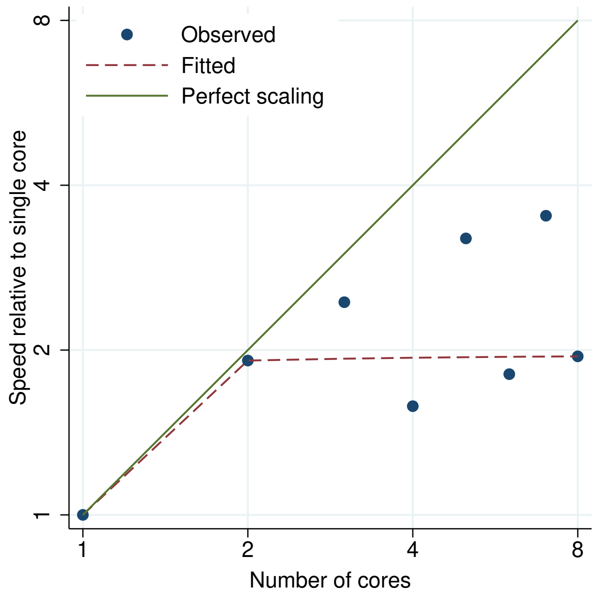


Figure 132. `eteffects (exponential), ate` performance plot.

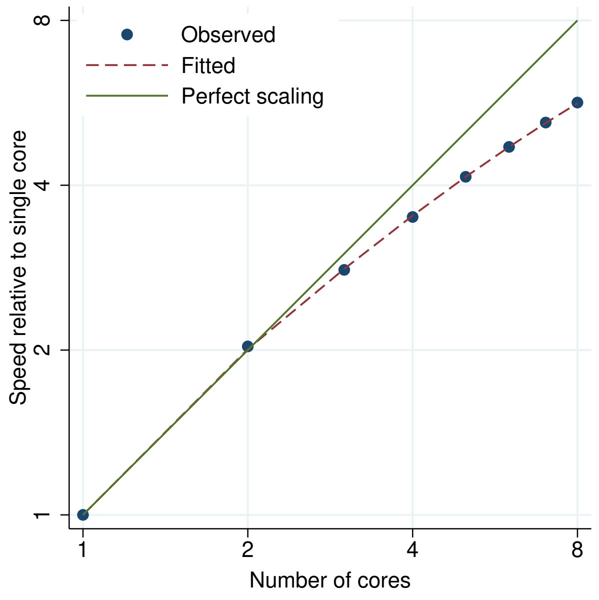


Figure 133. `eteffects (linear), ate` performance plot.

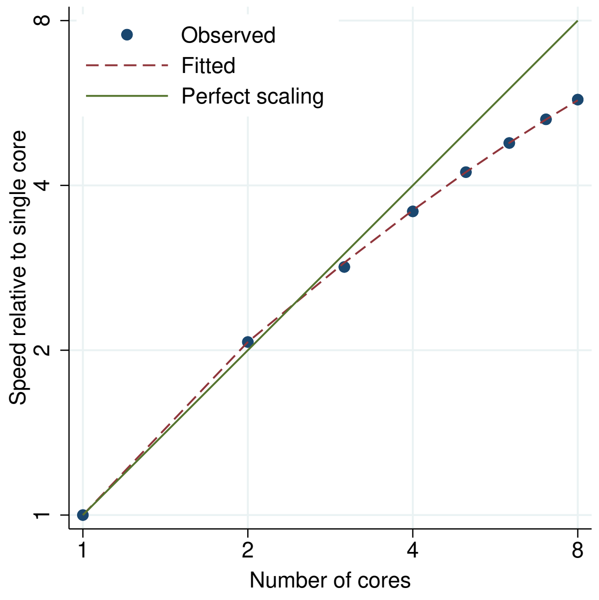


Figure 134. `eteffects (linear), pomeans` performance plot.

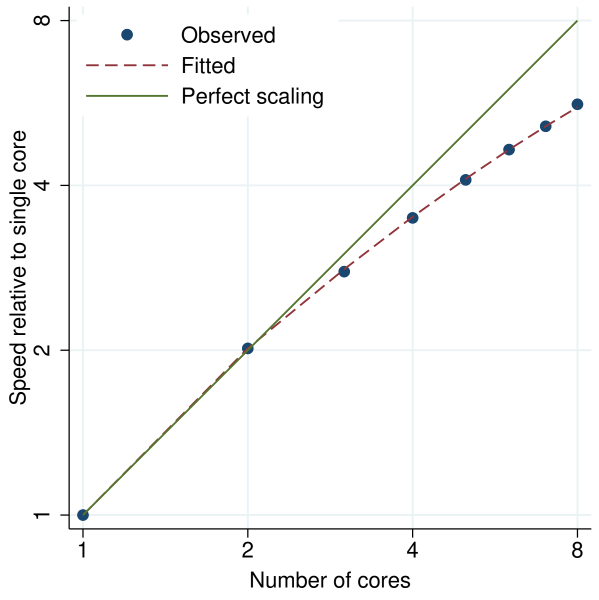


Figure 135. `eteffects (probit), ate` performance plot.

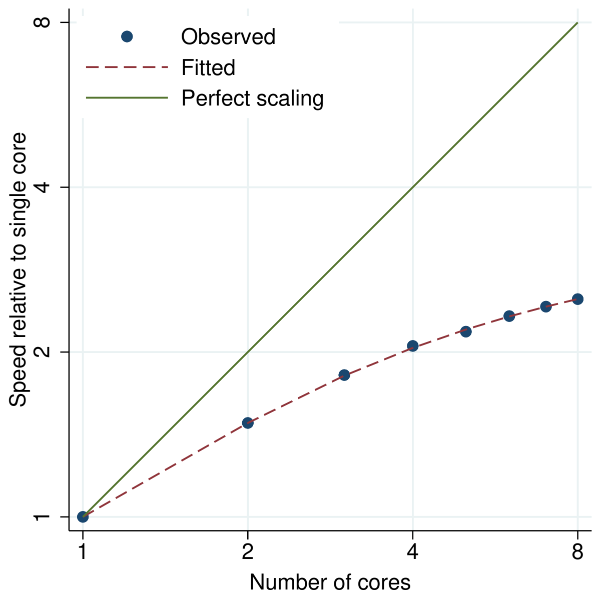


Figure 136. `etpoisson` performance plot.

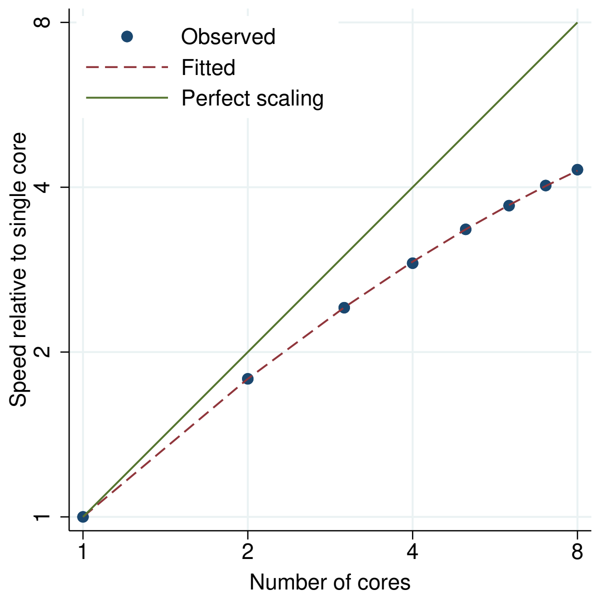


Figure 137. `etregress, poutcomes` performance plot.

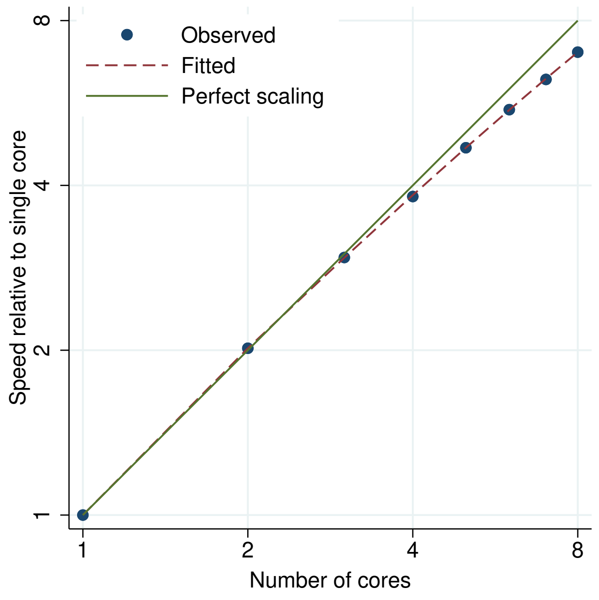


Figure 138. `etregress, twostep` performance plot.

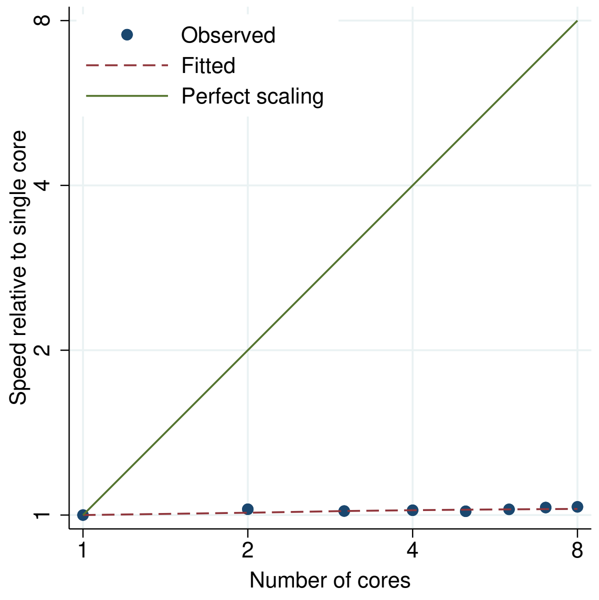


Figure 139. `exlogistic` performance plot.

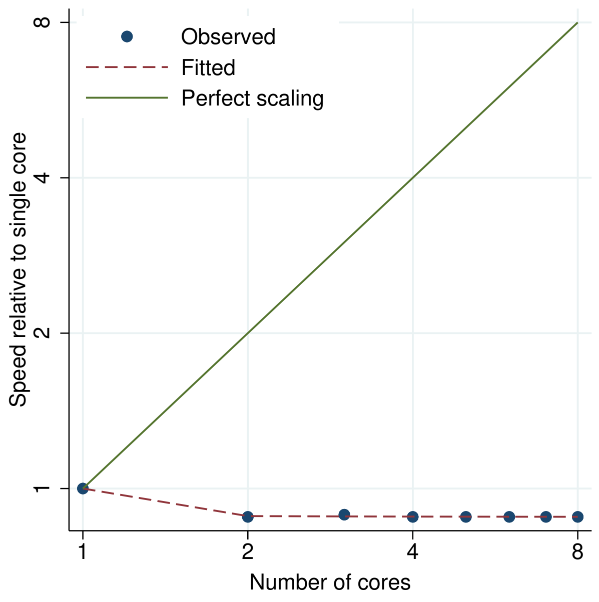


Figure 140. `expand` # performance plot.

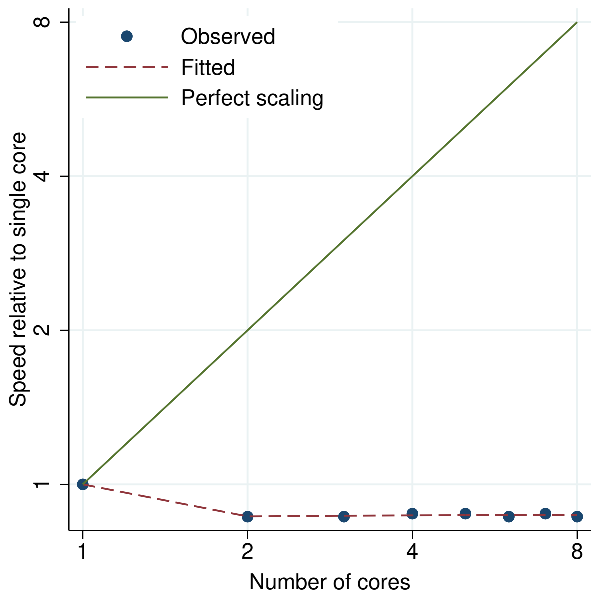


Figure 141. `expand` *varname* performance plot.

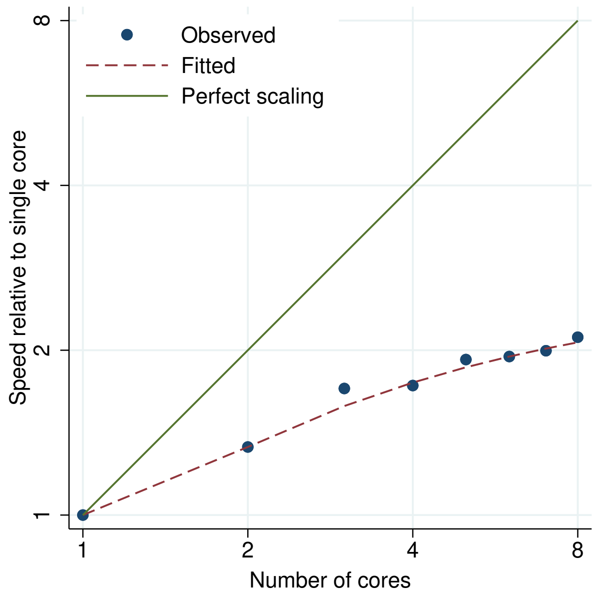


Figure 142. `expandcl` # performance plot.

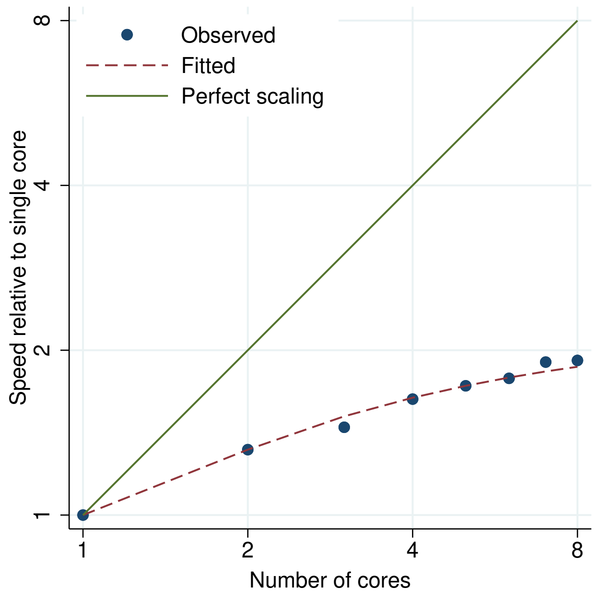


Figure 143. `expandcl` *varname* performance plot.

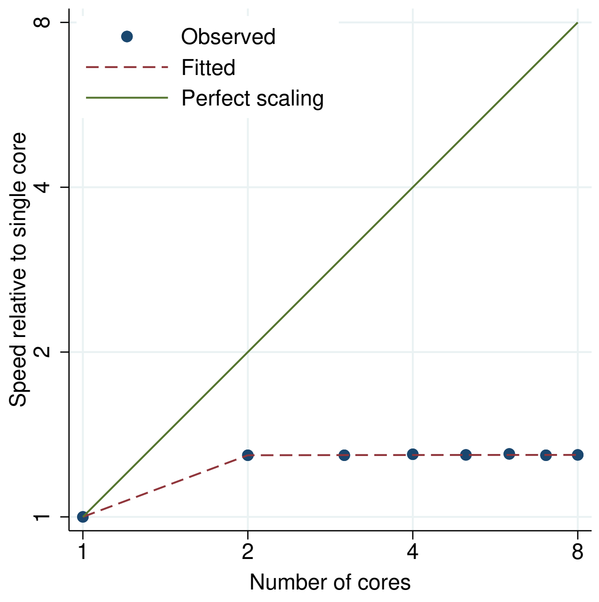


Figure 144. `expoisson` performance plot.

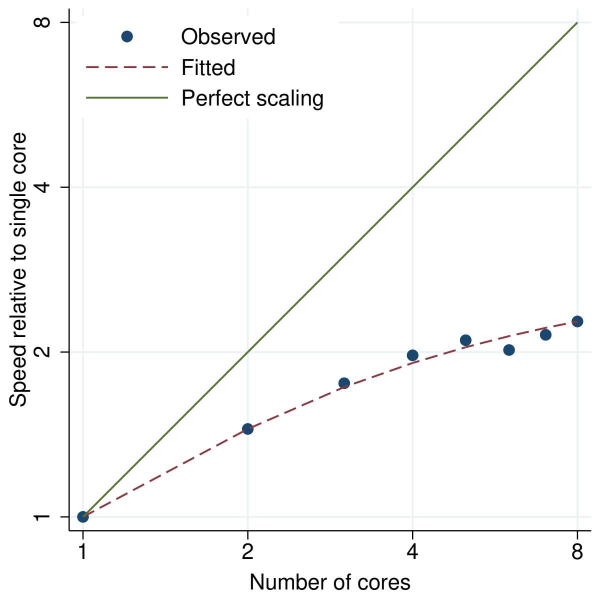


Figure 145. `factor` performance plot.

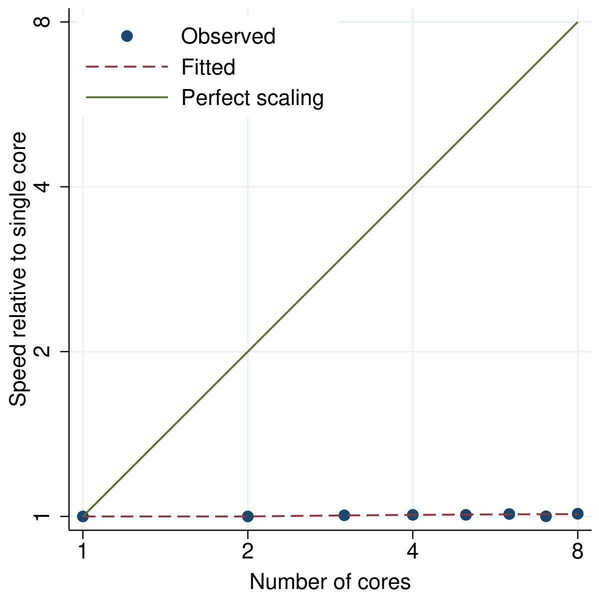


Figure 146. `fcast compute` performance plot.

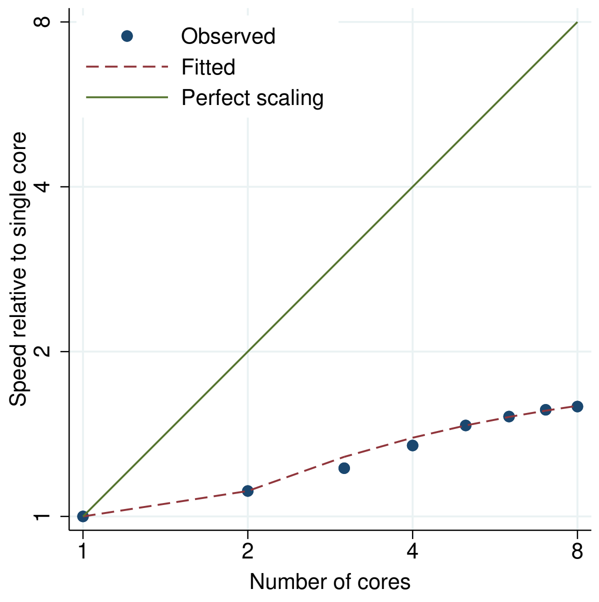


Figure 147. `fillin` performance plot.

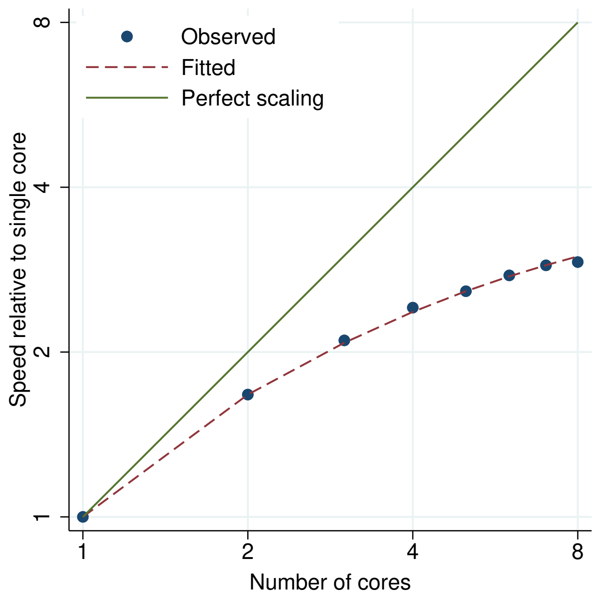


Figure 148. `fmm 2: poisson` performance plot.

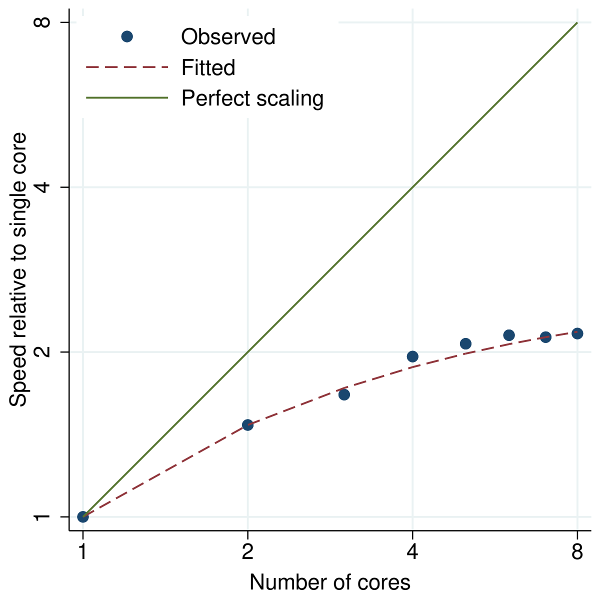


Figure 149. `fmm 2: regress` performance plot.

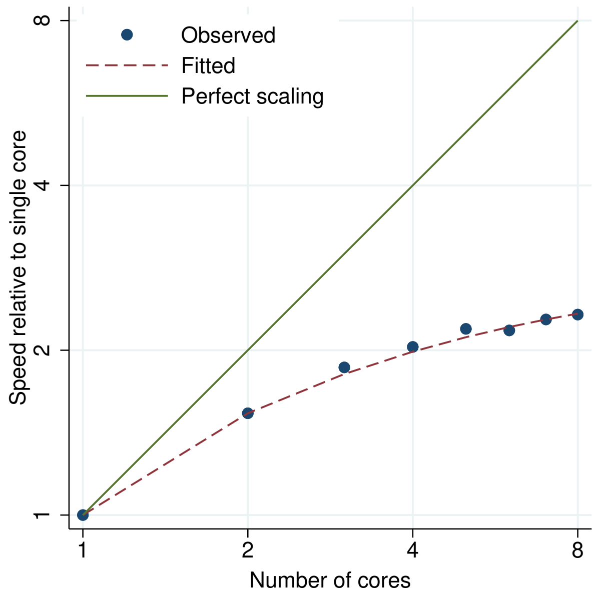


Figure 150. `fmm 3: poisson` performance plot.

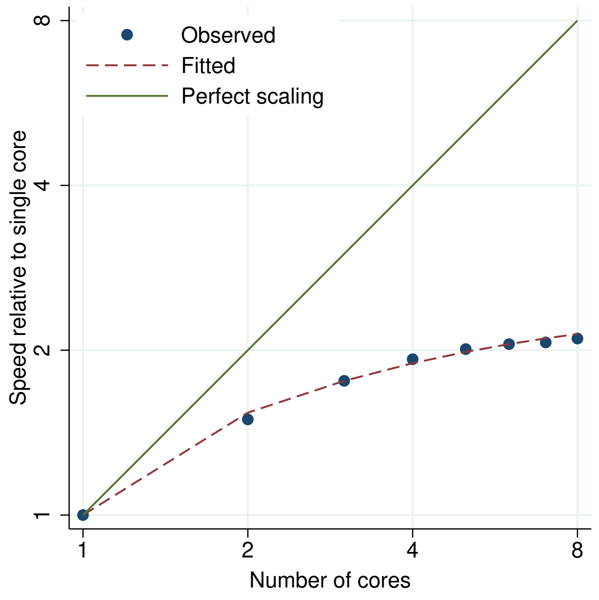


Figure 151. `fmm 3: regress` performance plot.

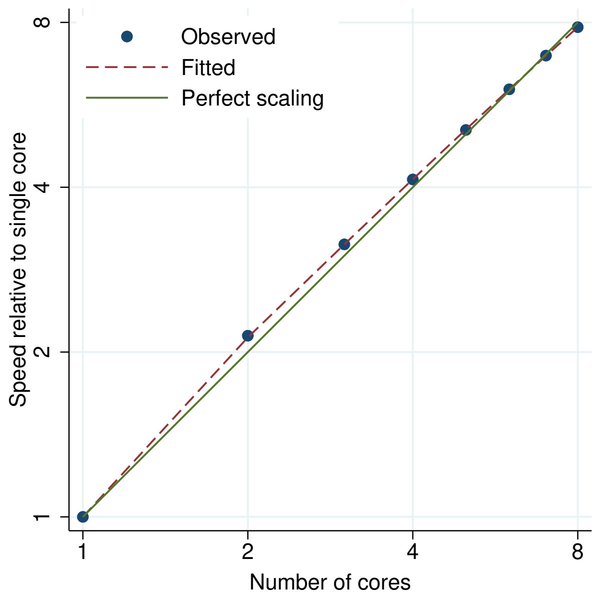


Figure 152. `fracreg probit` performance plot.

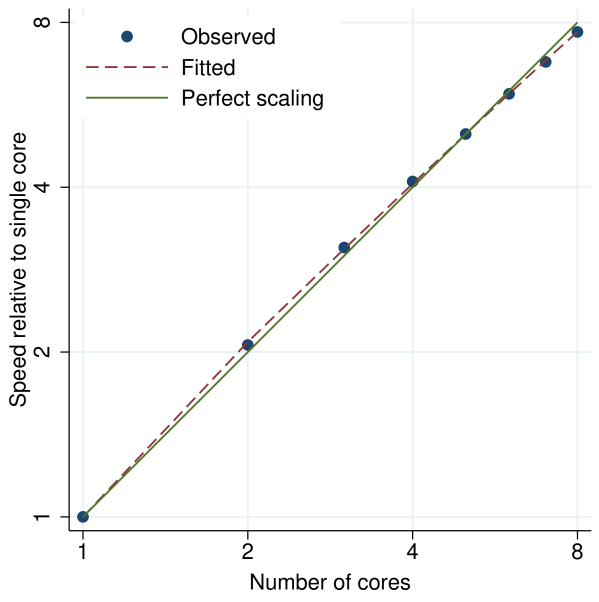


Figure 153. `frontier` performance plot.

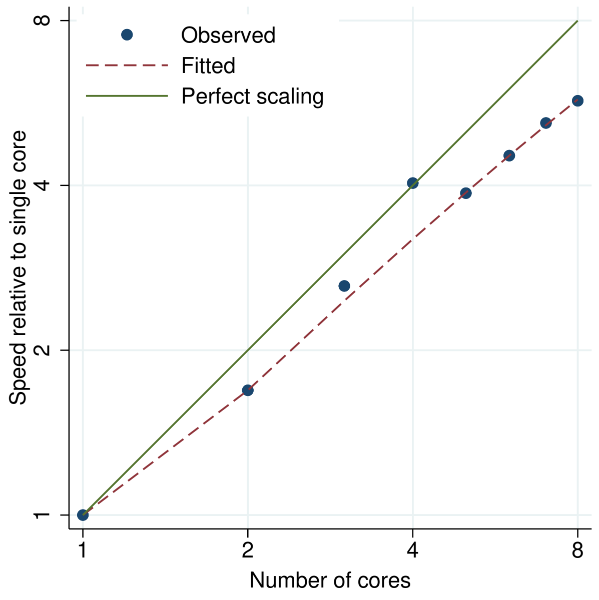


Figure 154. `fvrevar` (factors) performance plot.

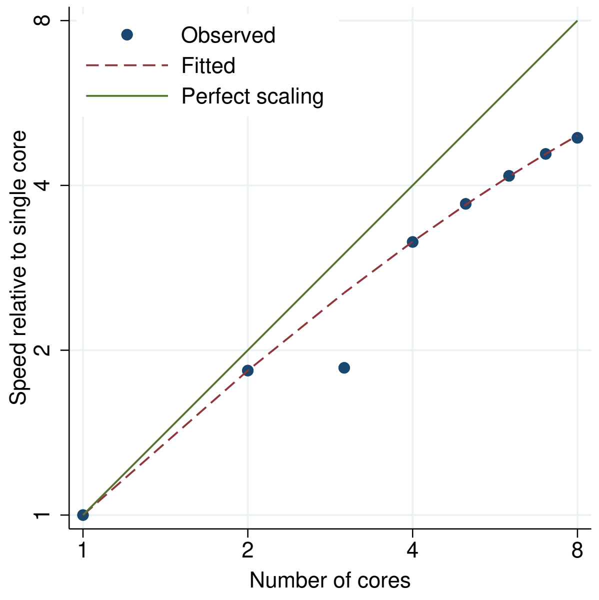


Figure 155. `fvrevar` (interaction) performance plot.

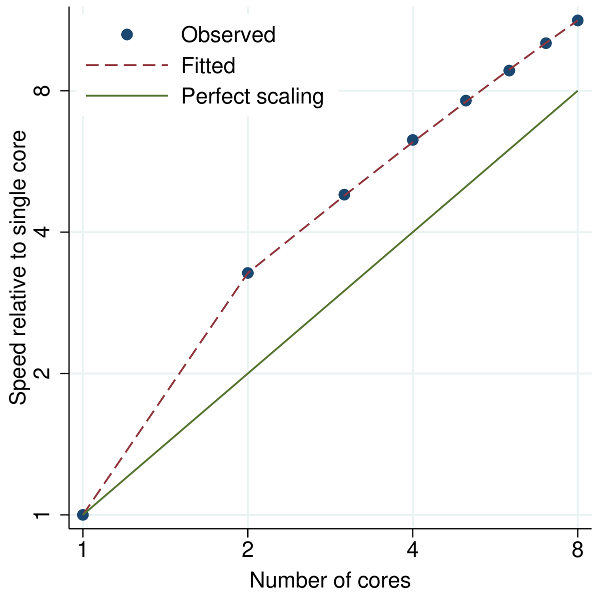


Figure 156. `generate` (small expressions) performance plot.

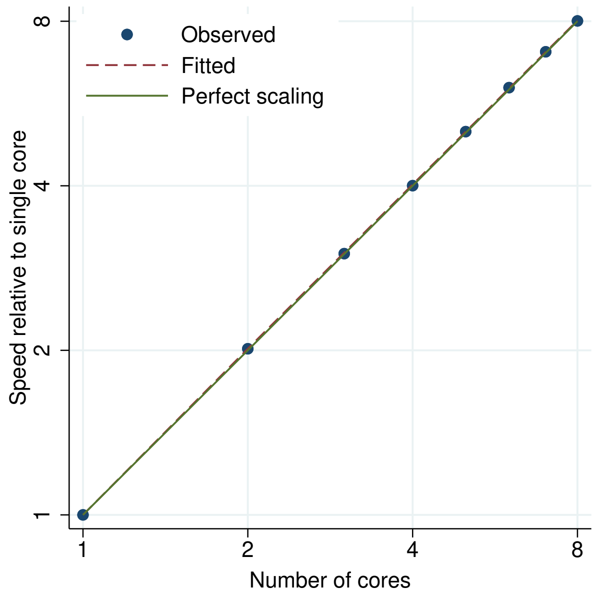


Figure 157. `generate` performance plot.

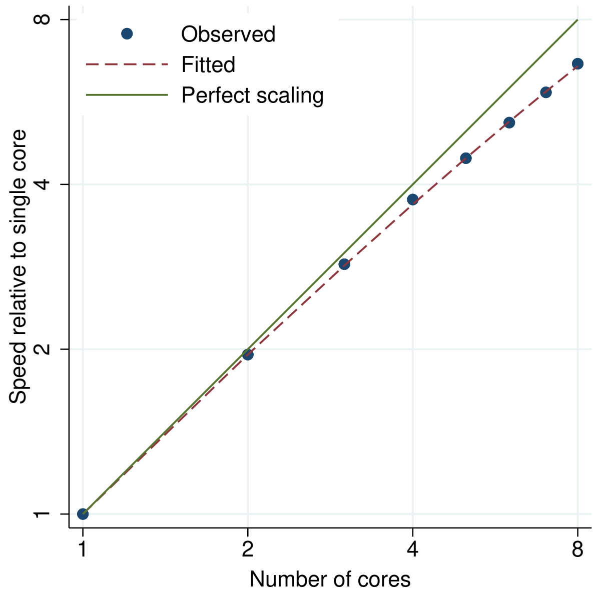


Figure 158. `glm, family(gamma)` performance plot.

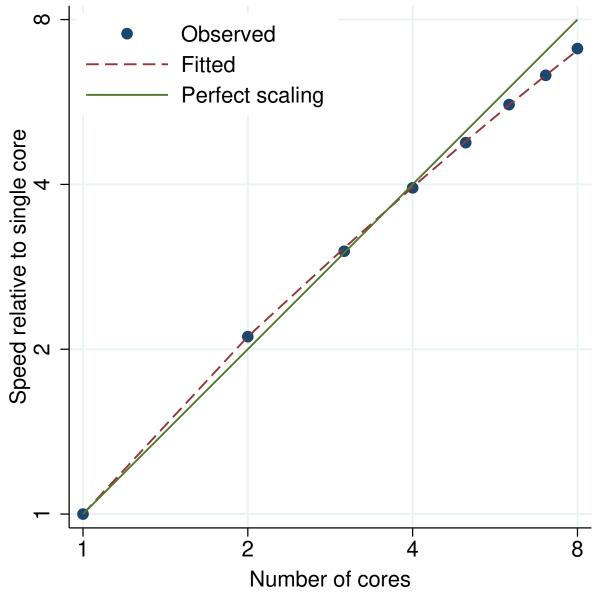


Figure 159. `glm, family(gaussian)` performance plot.

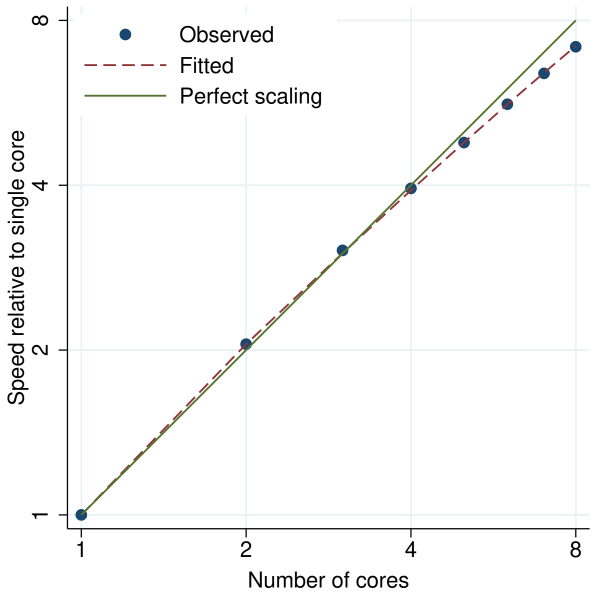


Figure 160. `glm, family(igaussian)` performance plot.

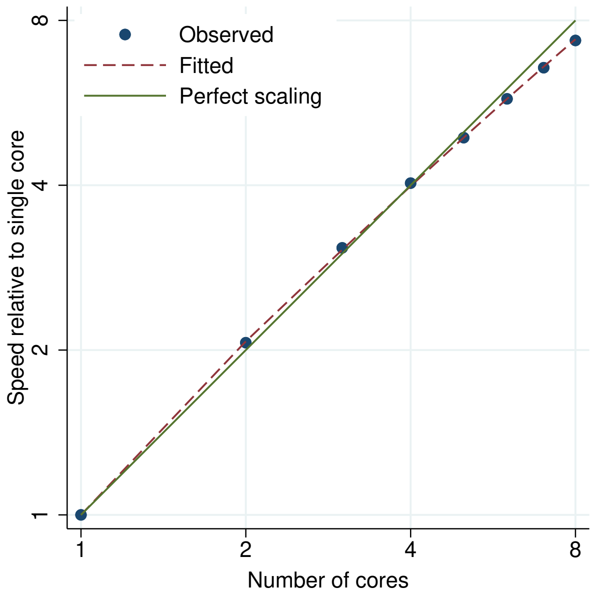


Figure 161. `glm, family(nbinomial)` performance plot.

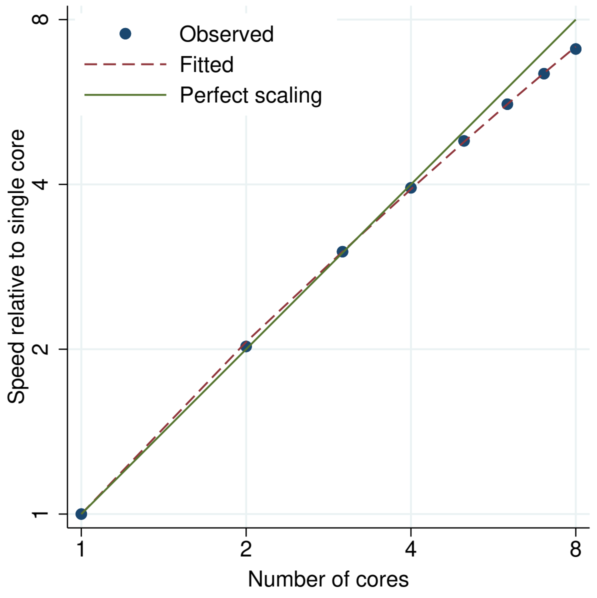


Figure 162. `glm, family(poisson)` performance plot.

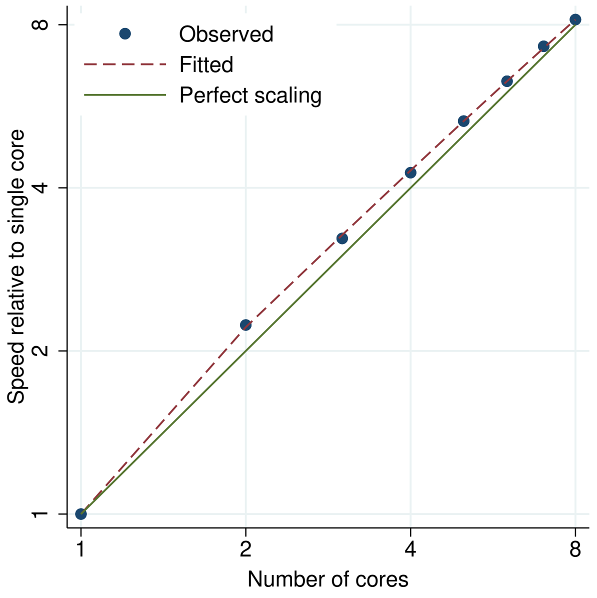


Figure 163. `glogit` performance plot.

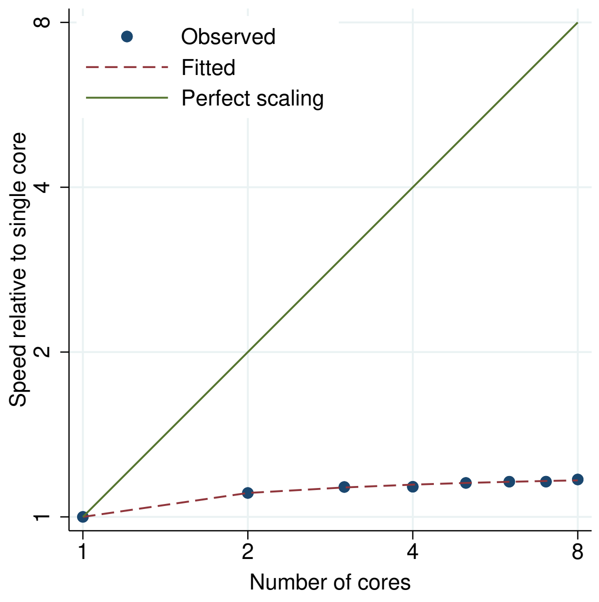


Figure 164. `gmm` performance plot.

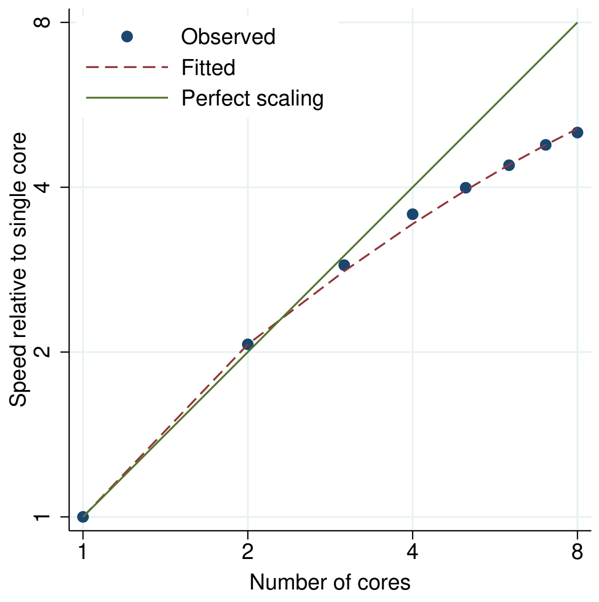


Figure 165. `gmm` (with derivatives) performance plot.

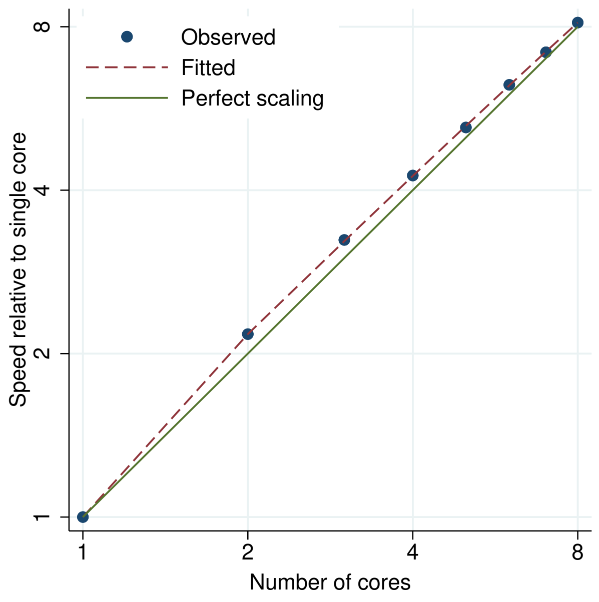


Figure 166. `gprobit` performance plot.

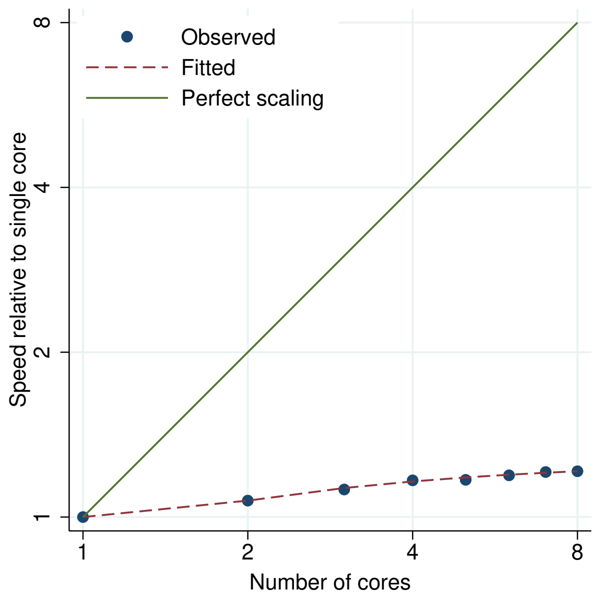


Figure 167. `graph bar` performance plot.

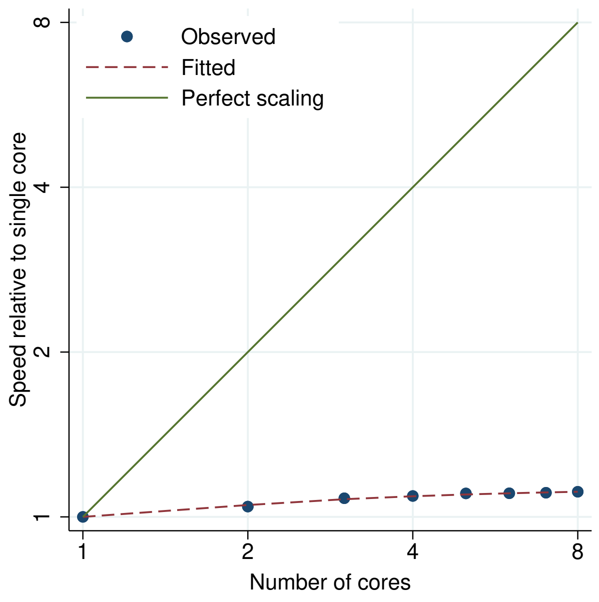


Figure 168. `graph box` performance plot.

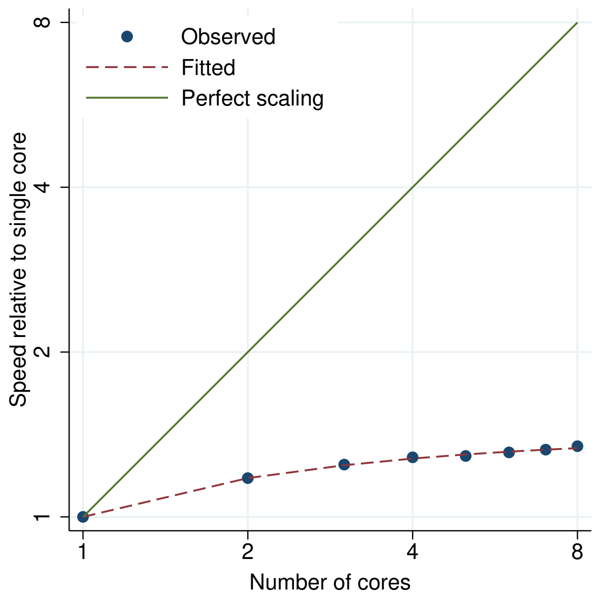


Figure 169. `graph pie` performance plot.

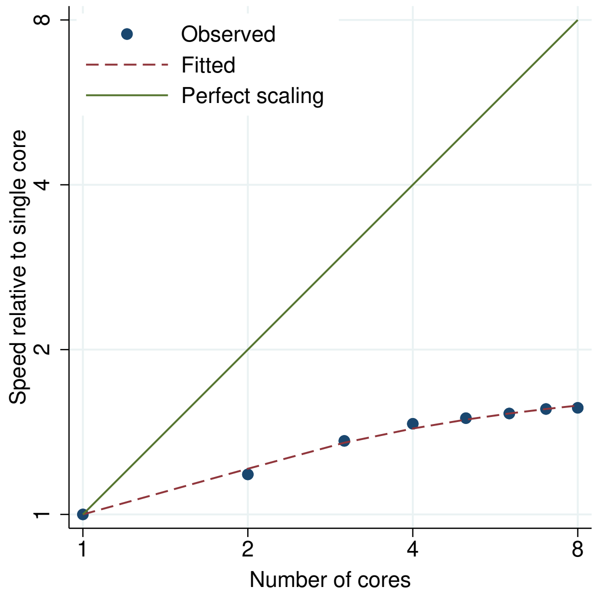


Figure 170. `grmeanby` performance plot.

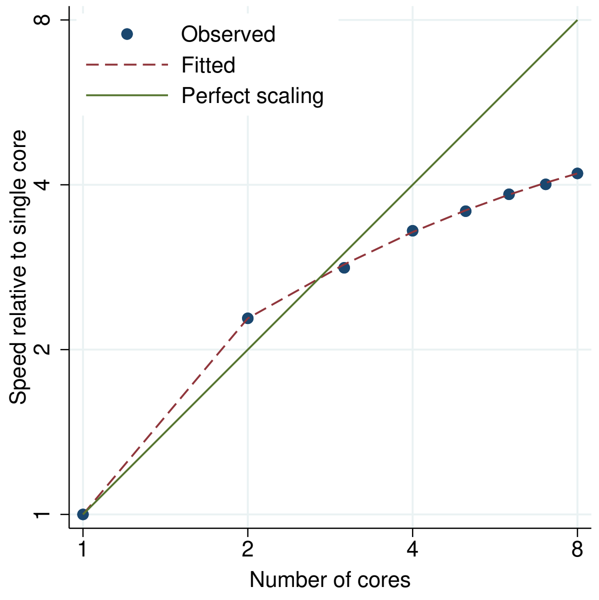


Figure 171. `gsem, oprobit (CFA, 2-level)` performance plot.

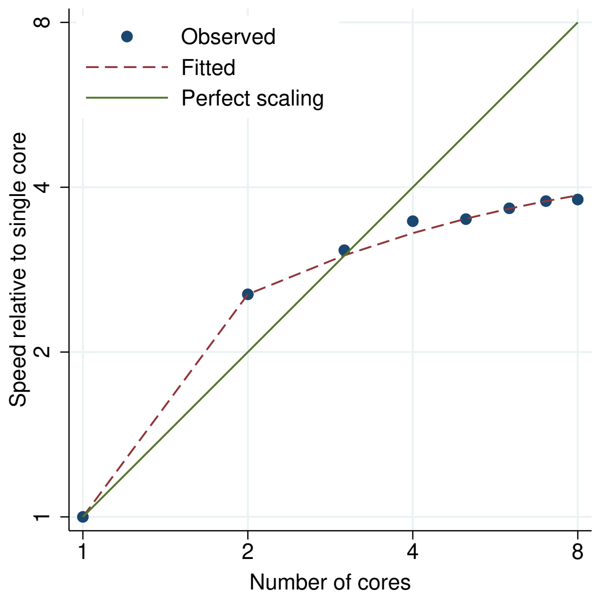


Figure 172. `gsem, oprobit (CFA)` performance plot.

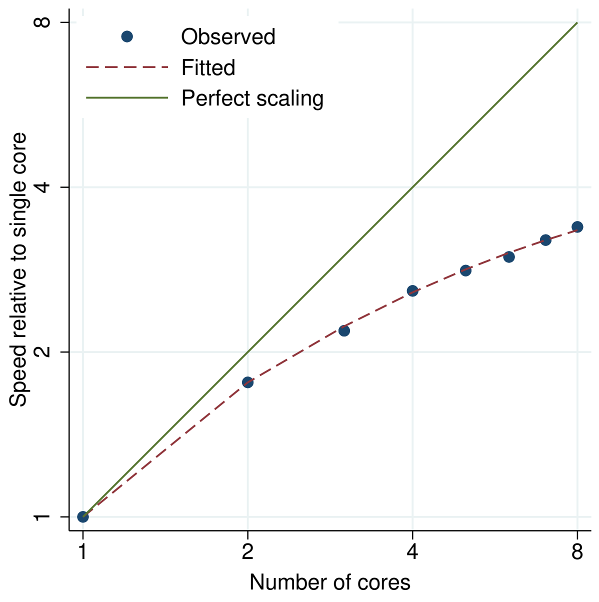


Figure 173. `gsem, logit group()` performance plot.

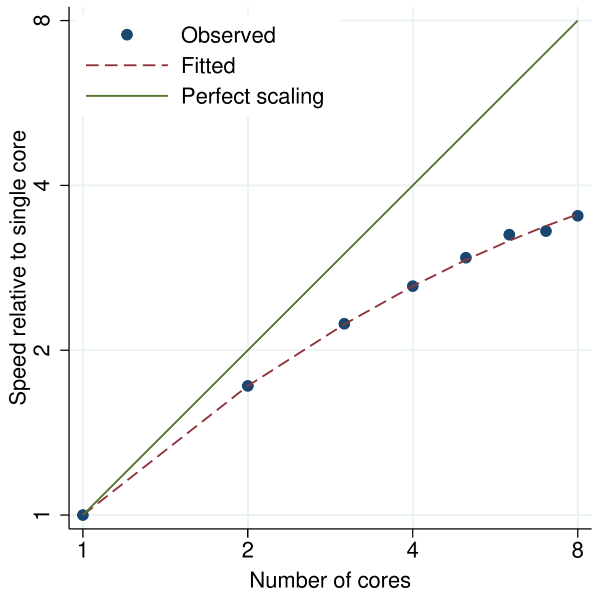


Figure 174. `gsem, group()` performance plot.

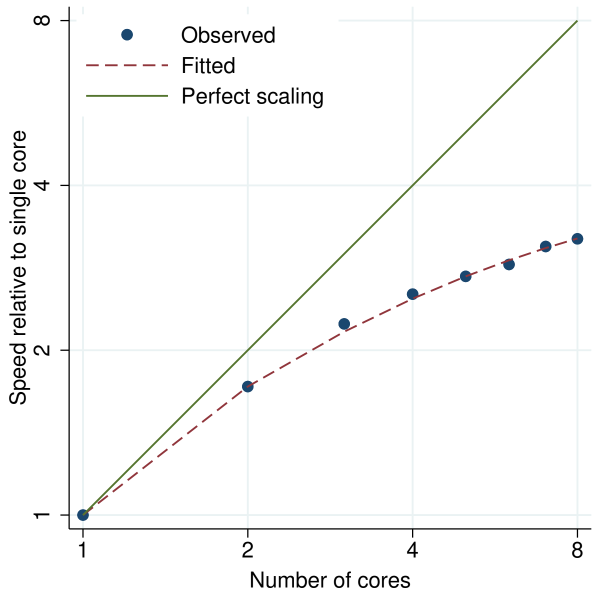


Figure 175. `gsem, ologit group()` performance plot.

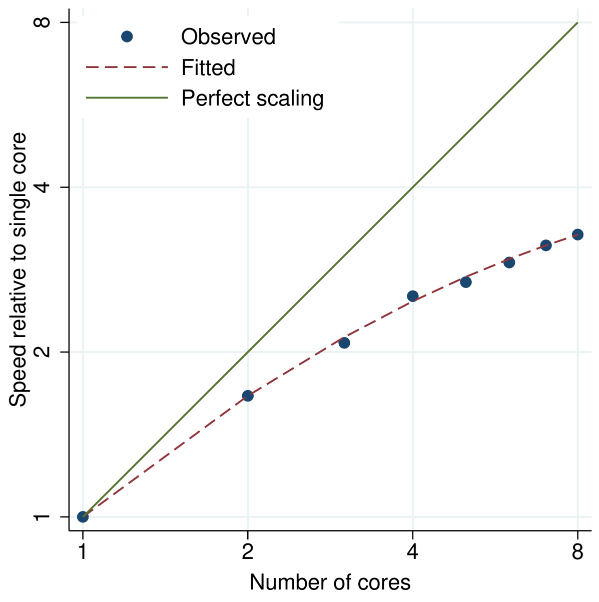


Figure 176. `gsem, poisson group()` performance plot.

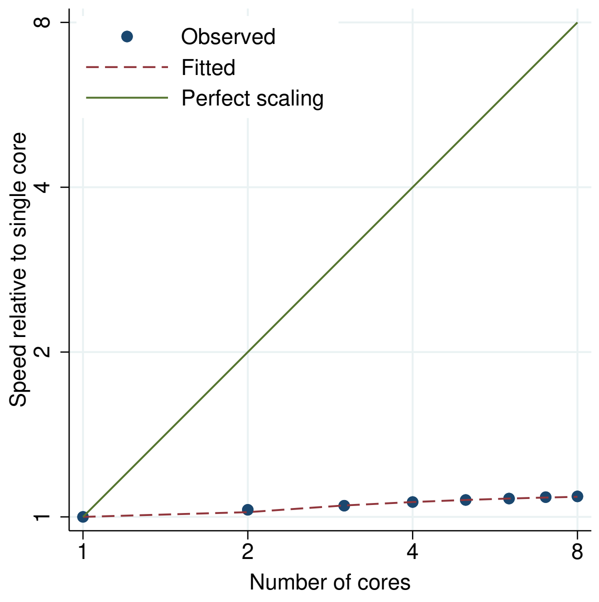


Figure 177. `gsort` performance plot.

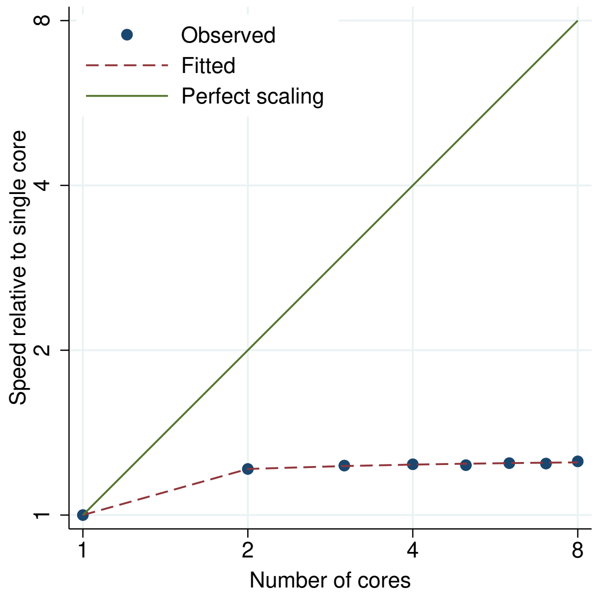


Figure 178. `hausman` performance plot.

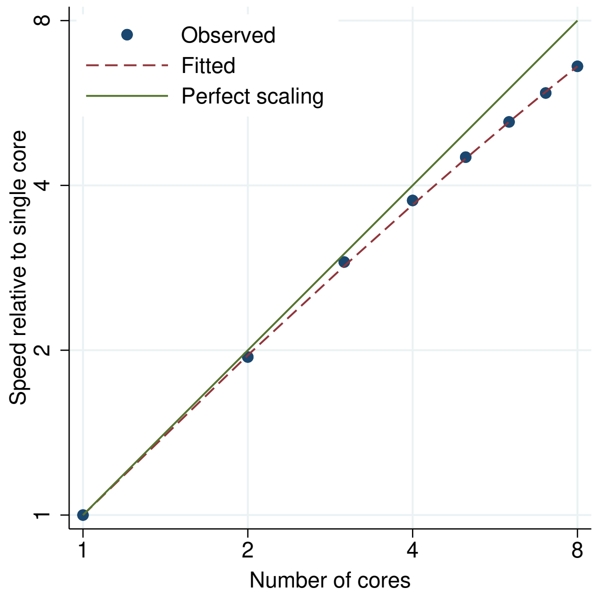


Figure 179. `heckman` performance plot.

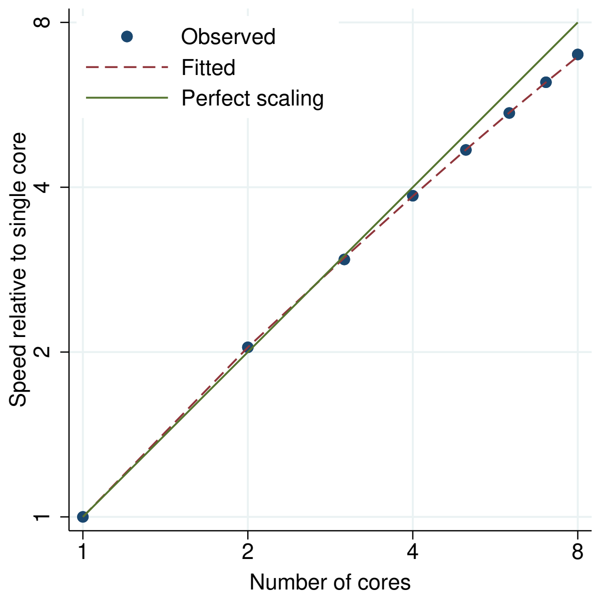


Figure 180. `heckman, twostep` performance plot.

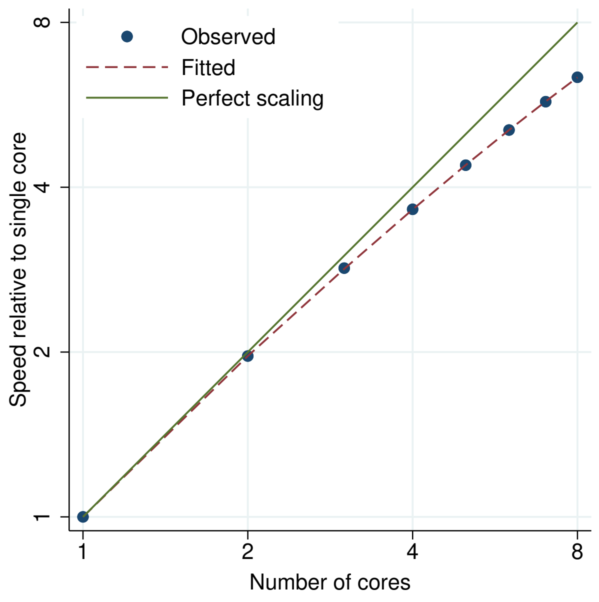


Figure 181. `heckoprobit` performance plot.

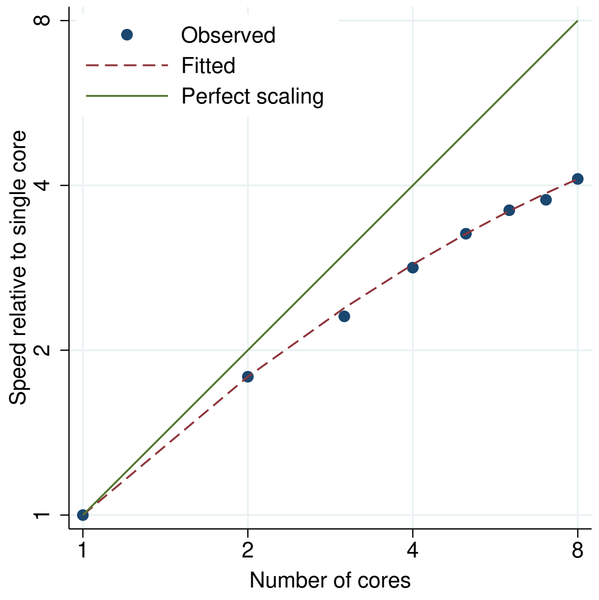


Figure 182. `heckpoisson` performance plot.

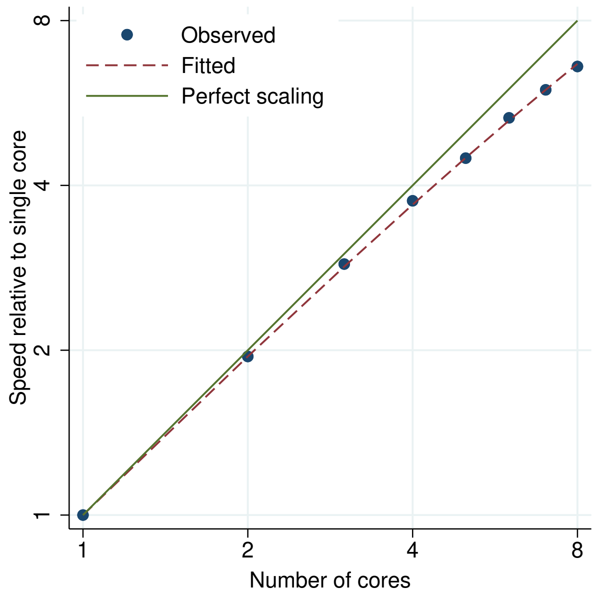


Figure 183. `heckprob` performance plot.

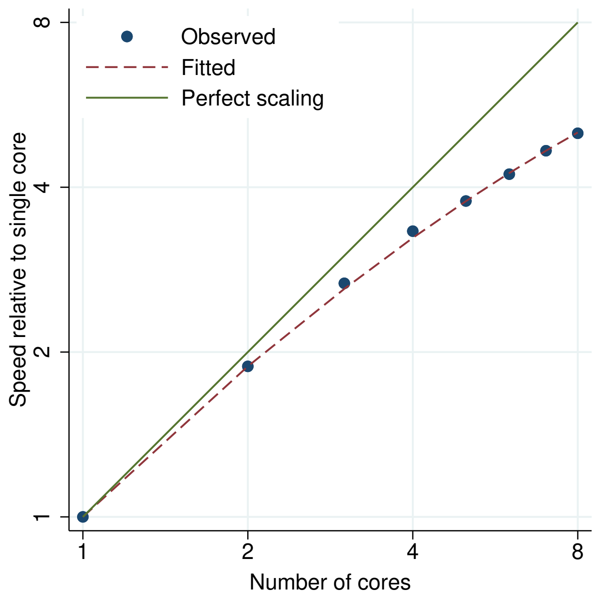


Figure 184. `hetoprobit` performance plot.

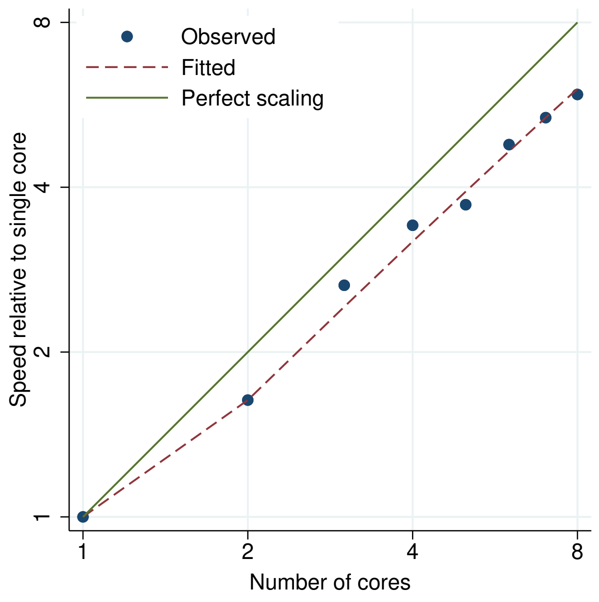


Figure 185. `hetprob` performance plot.

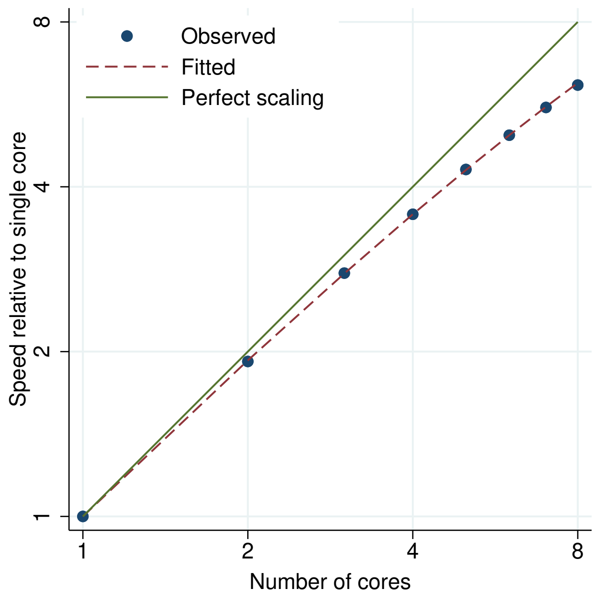


Figure 186. `hetregress` performance plot.

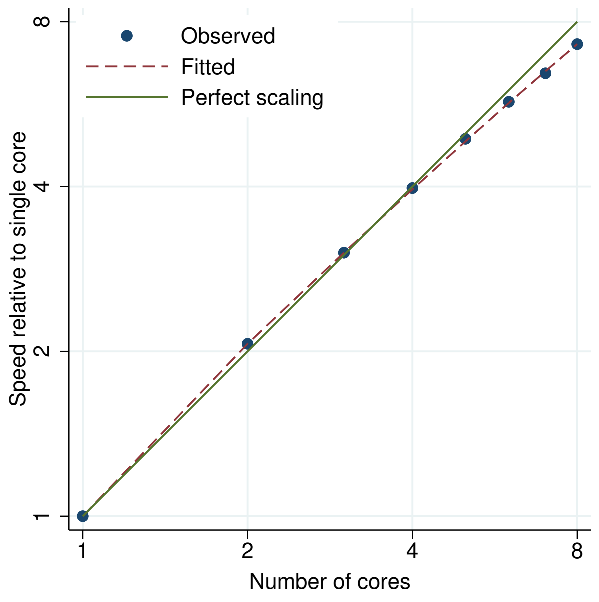


Figure 187. `hetregress, twostep` performance plot.

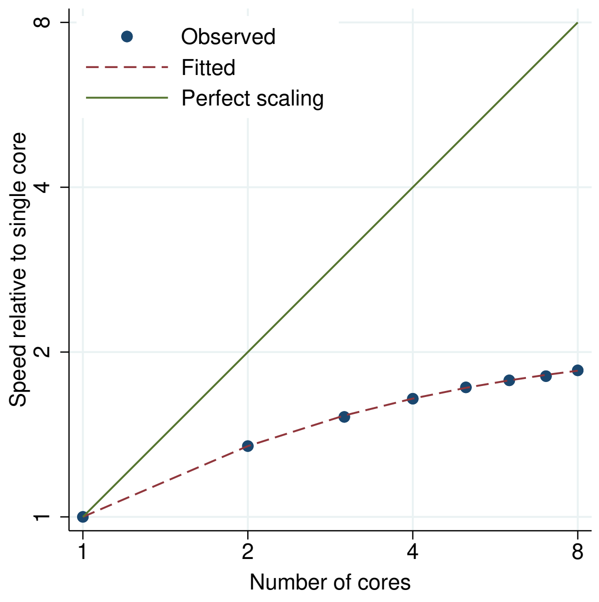


Figure 188. `histogram` performance plot.

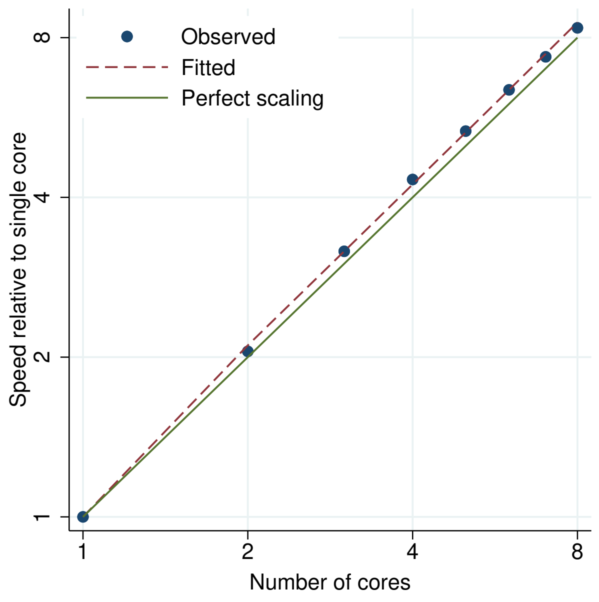


Figure 189. `hotelling` performance plot.

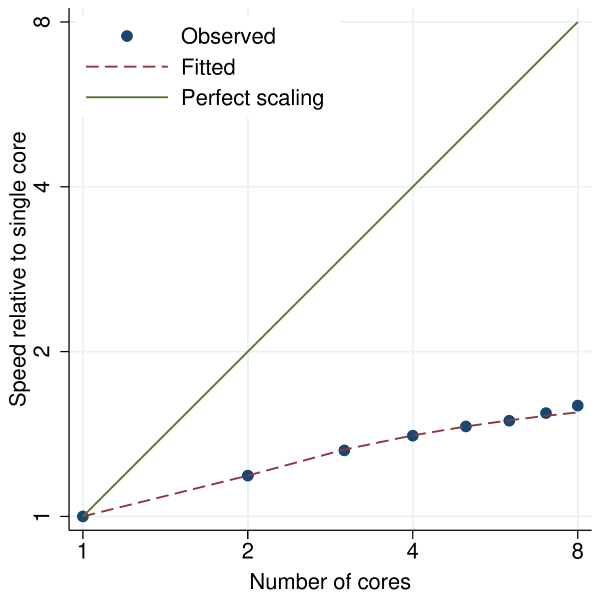


Figure 190. `icc, mixed` performance plot.

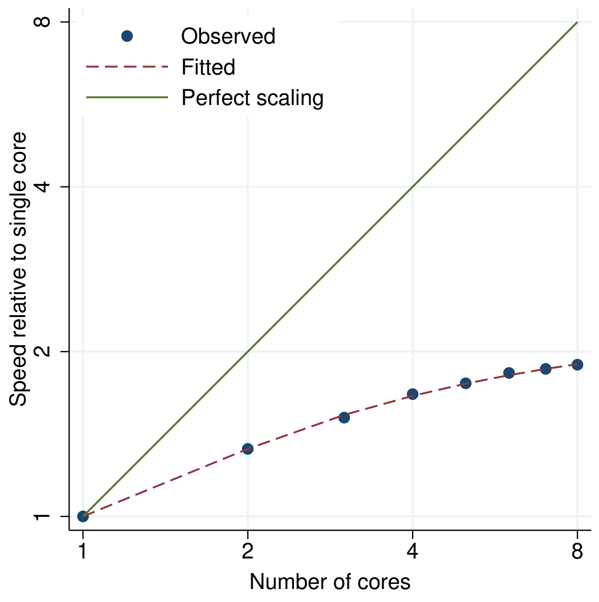


Figure 191. `icc` (one-way) performance plot.

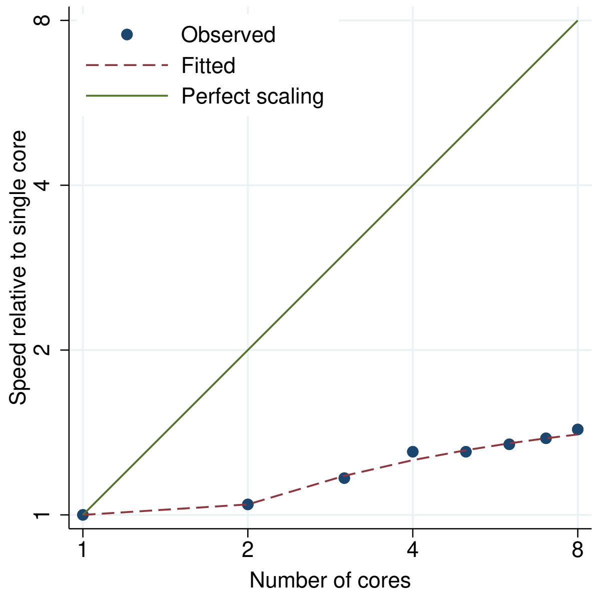


Figure 192. `icc` (two-way) performance plot.

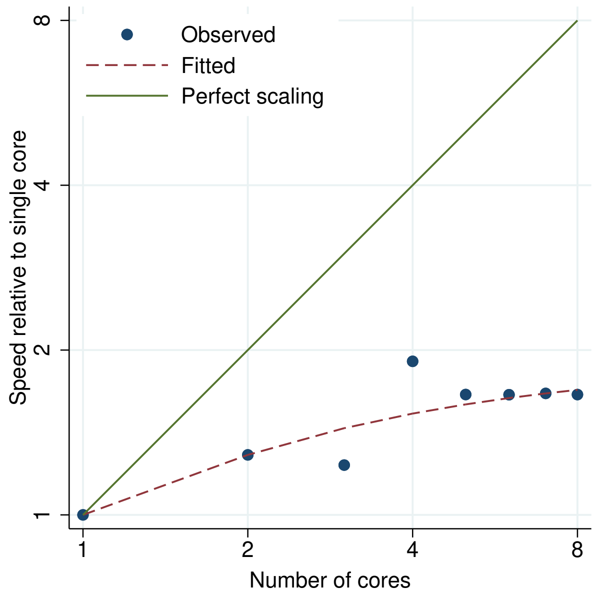


Figure 193. `import delimited` performance plot.

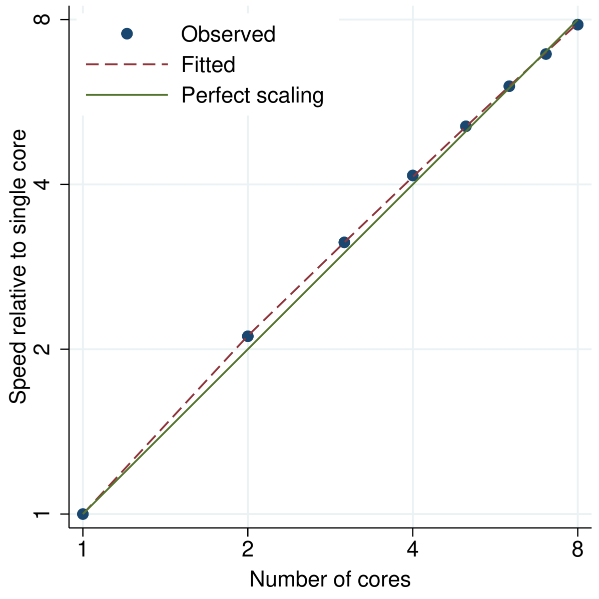


Figure 194. `intreg` performance plot.

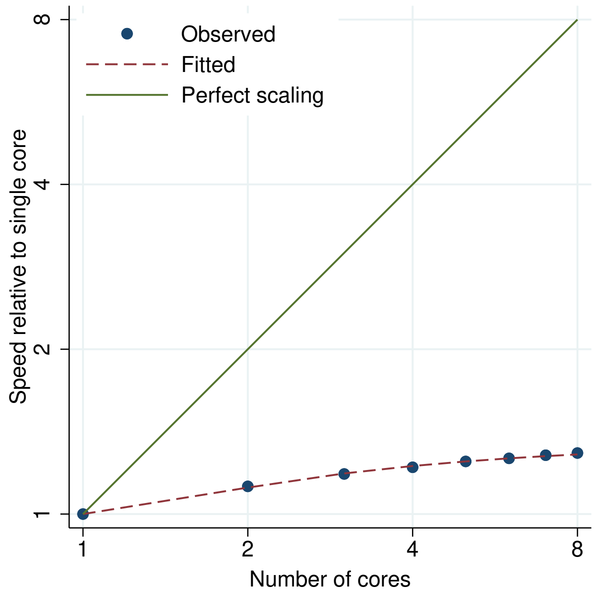


Figure 195. `ir` performance plot.

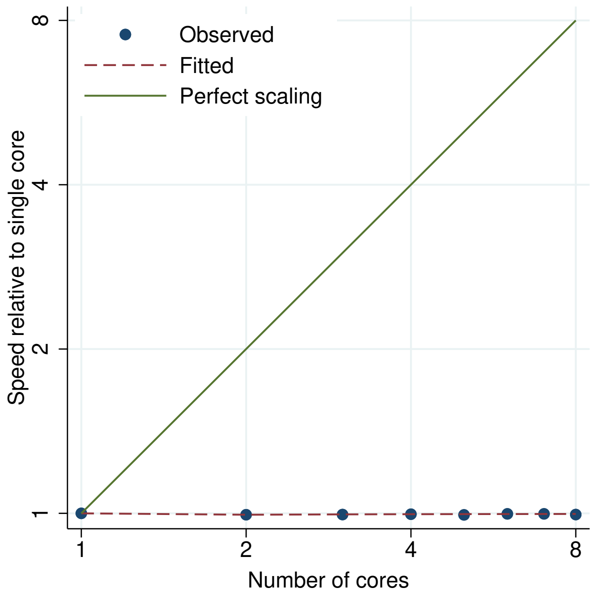


Figure 196. `by: ir` performance plot.

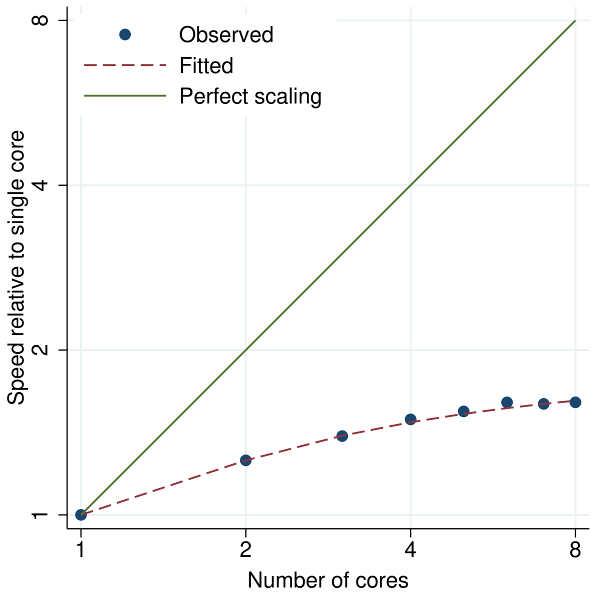


Figure 197. `irf create` performance plot.

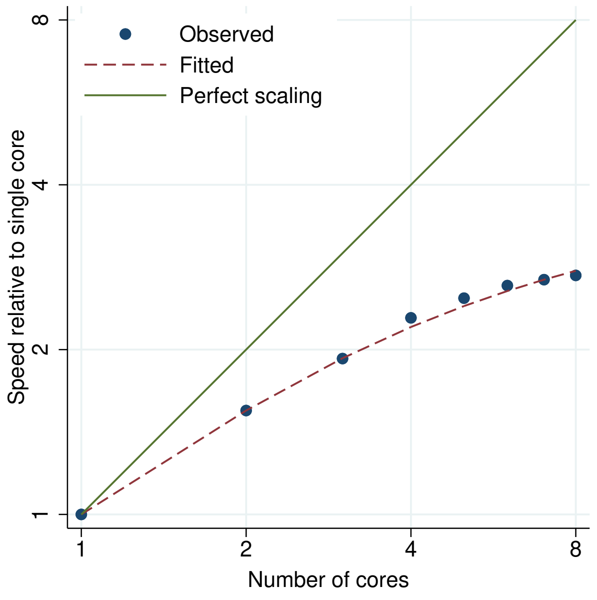


Figure 198. `irt 1pl` performance plot.

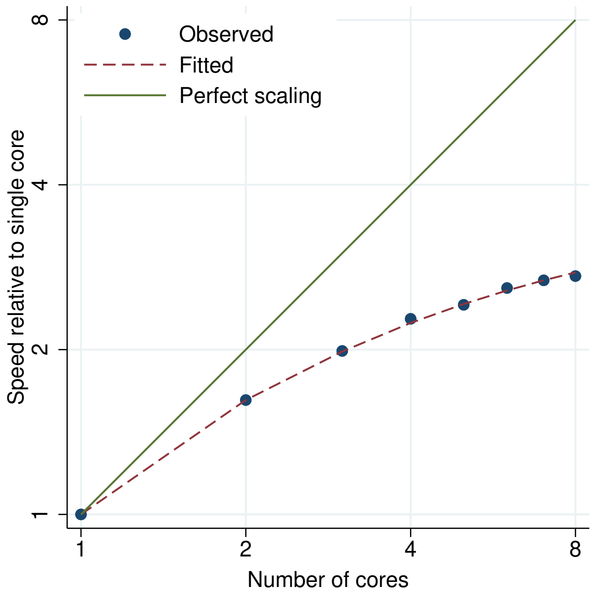


Figure 199. `irt 2pl` performance plot.

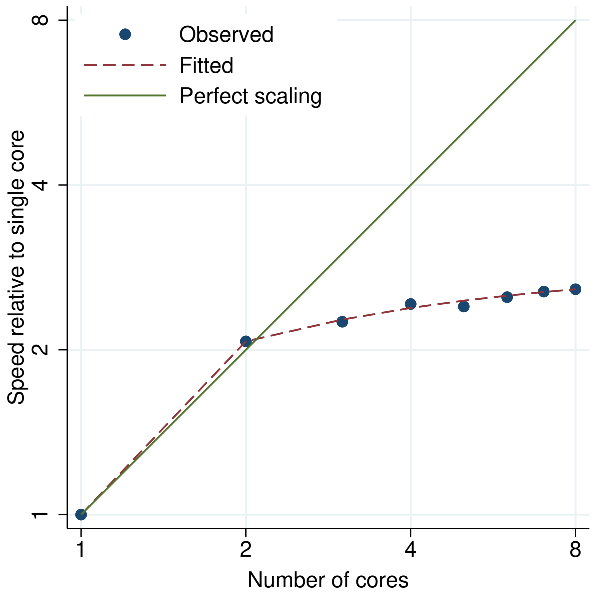


Figure 200. `irt 3pl` performance plot.

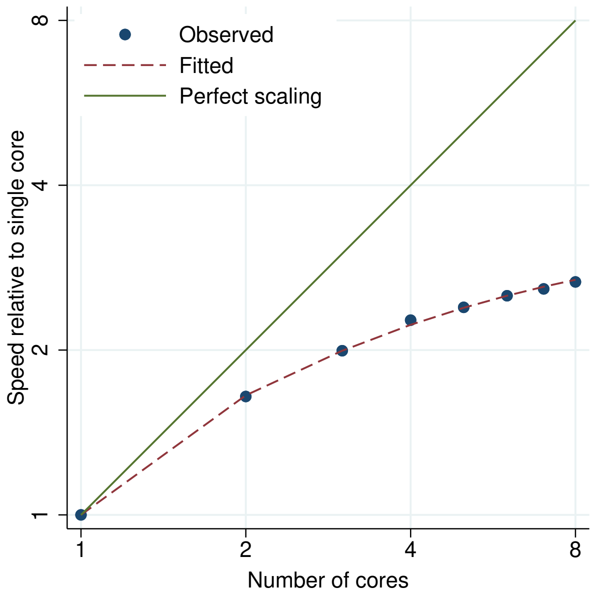


Figure 201. `irt grm` performance plot.

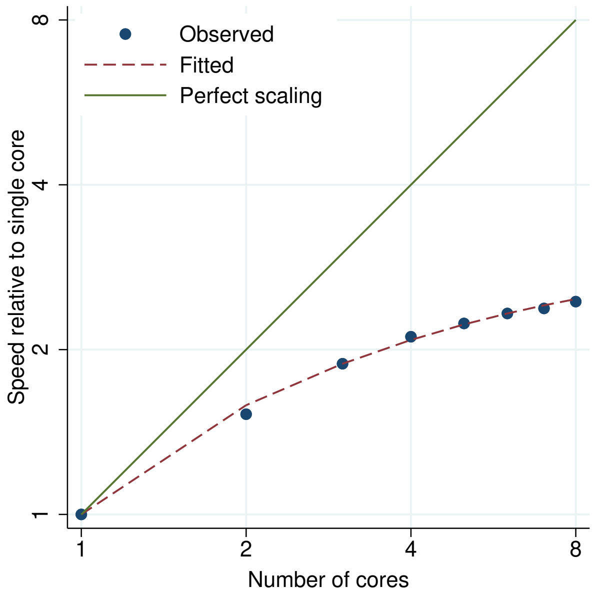


Figure 202. `irt nrm` performance plot.

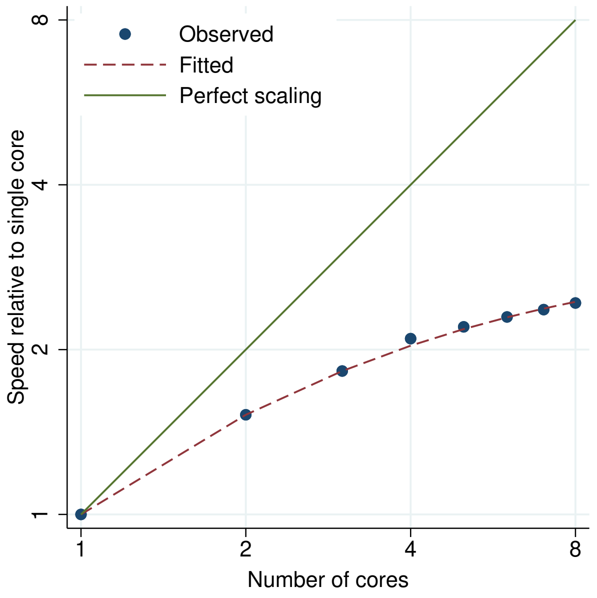


Figure 203. `irt pcm` performance plot.

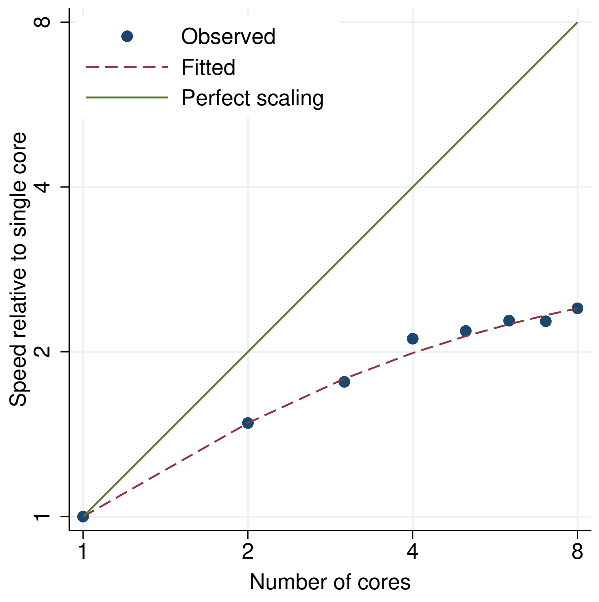


Figure 204. `irt rsm` performance plot.

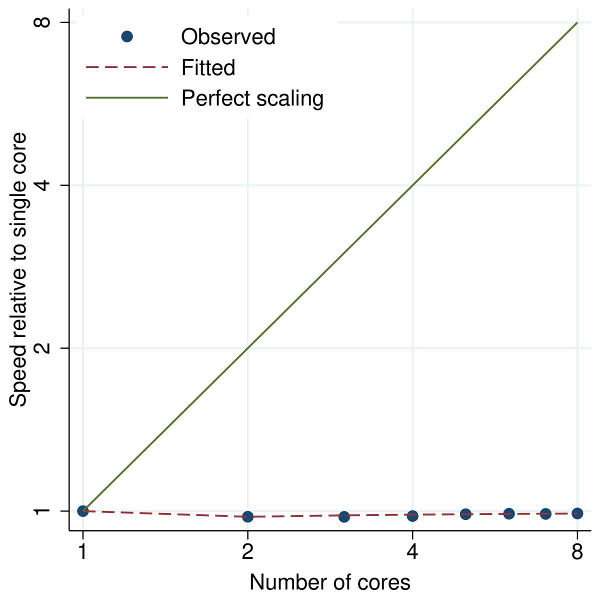


Figure 205. `istdize` performance plot.

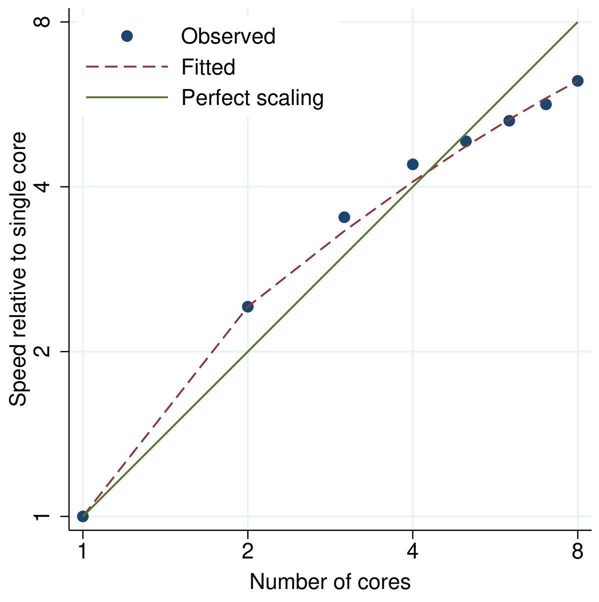


Figure 206. `ivpoisson cfunction` performance plot.

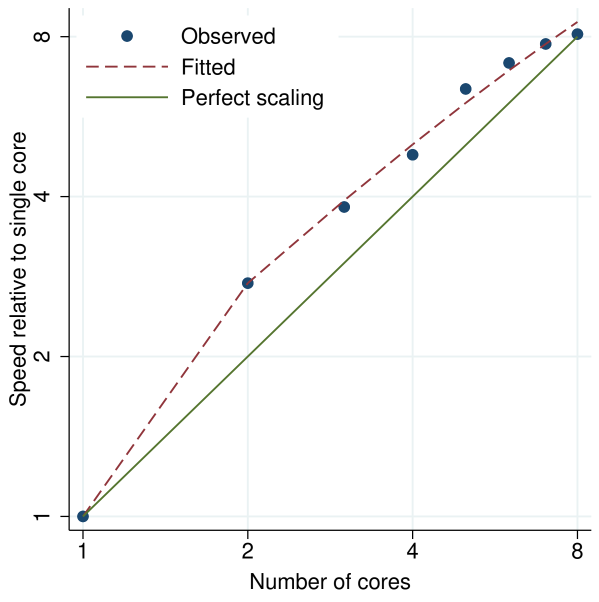


Figure 207. `ivpoisson gmm, additive` performance plot.

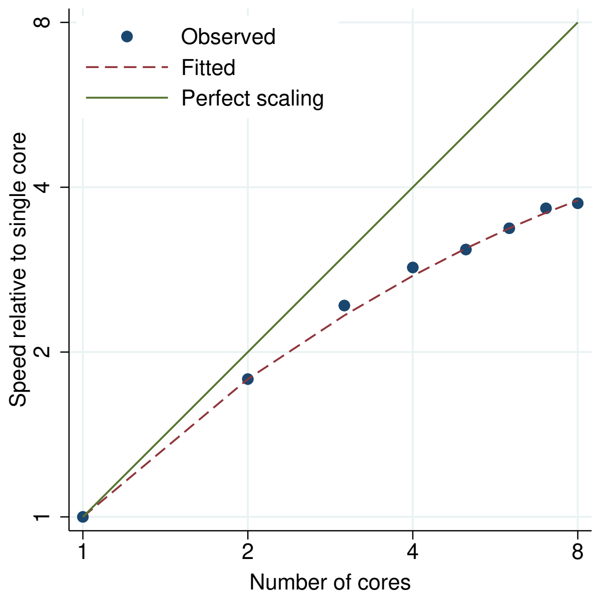


Figure 208. `ivpoisson gmm, multiplicative` performance plot.

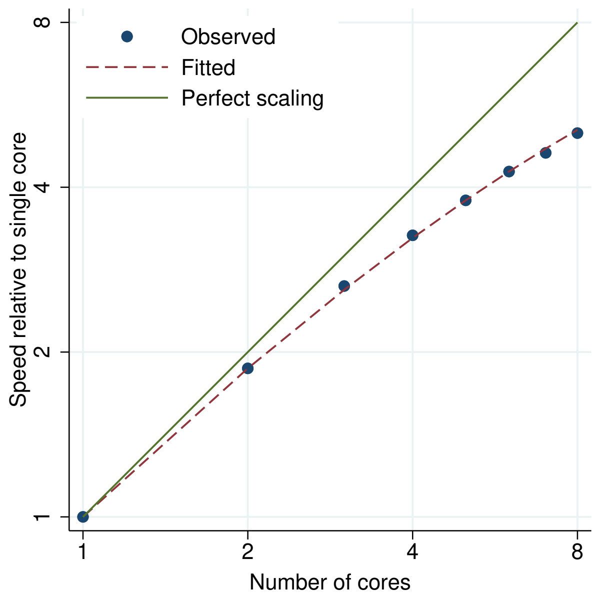


Figure 209. `ivprobit` performance plot.

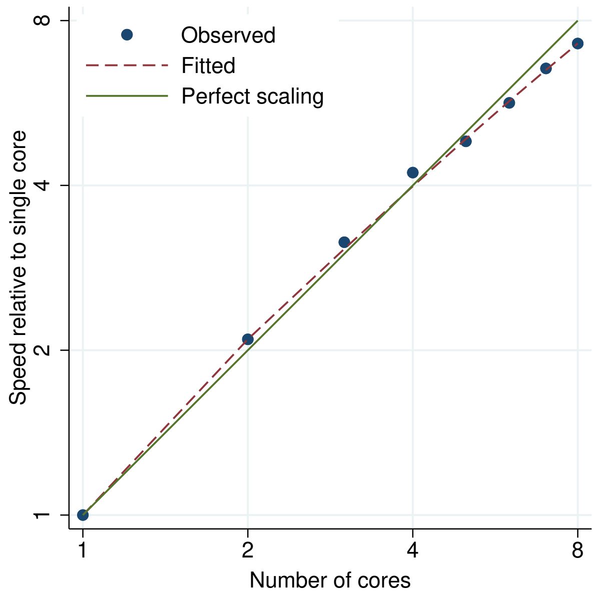


Figure 210. `ivregress 2sls` performance plot.

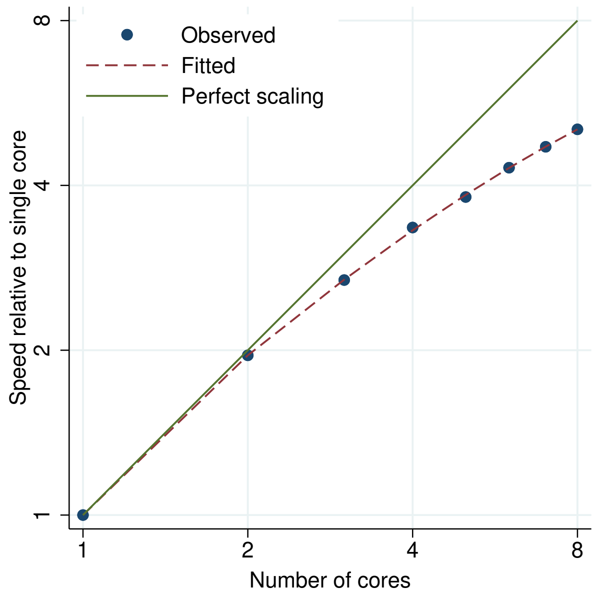


Figure 211. `ivregress gmm` performance plot.

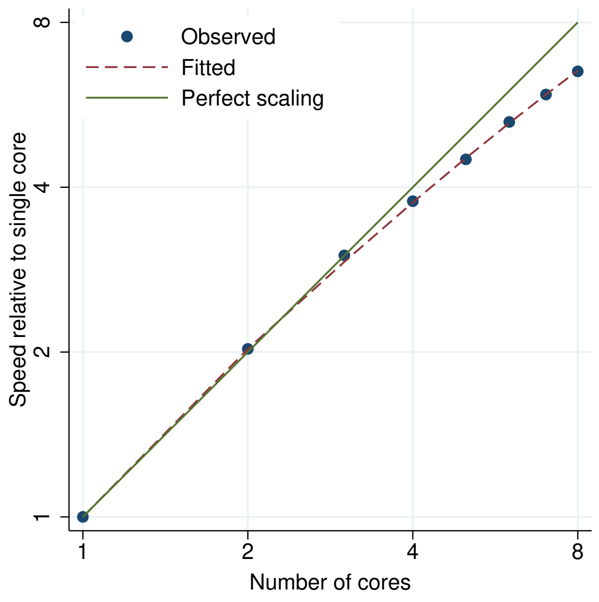


Figure 212. `ivregress liml` performance plot.

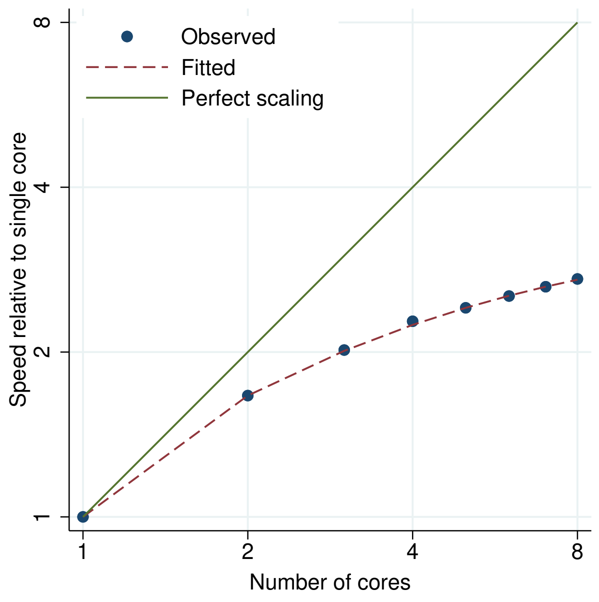


Figure 213. `ivtobit` performance plot.

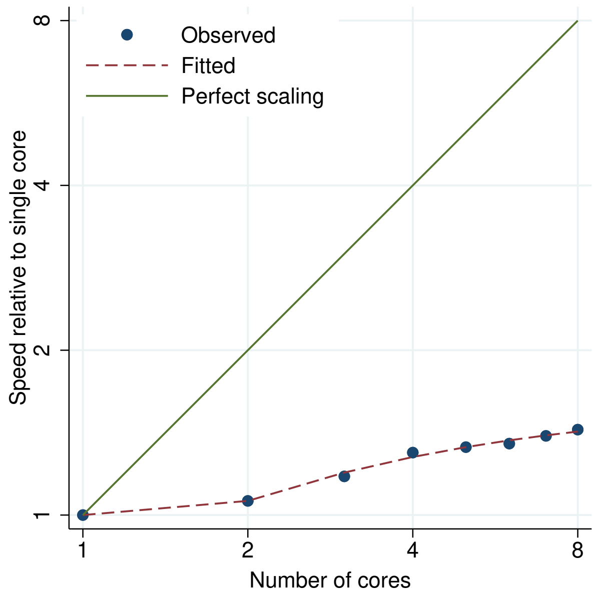


Figure 214. `kap` performance plot.

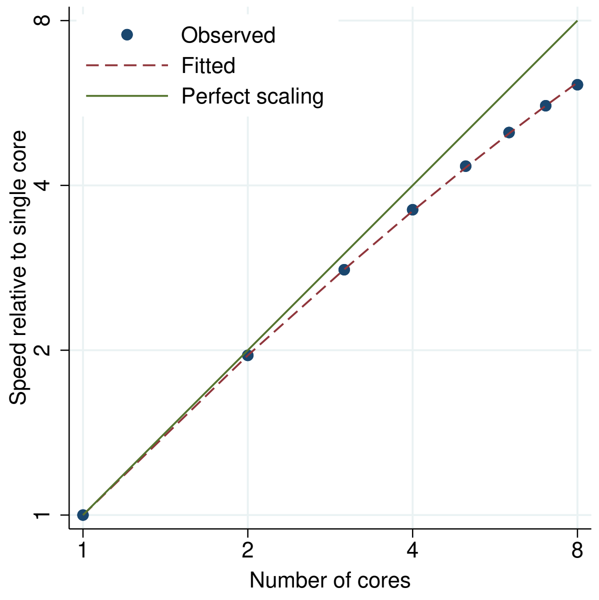


Figure 215. `kappa` performance plot.

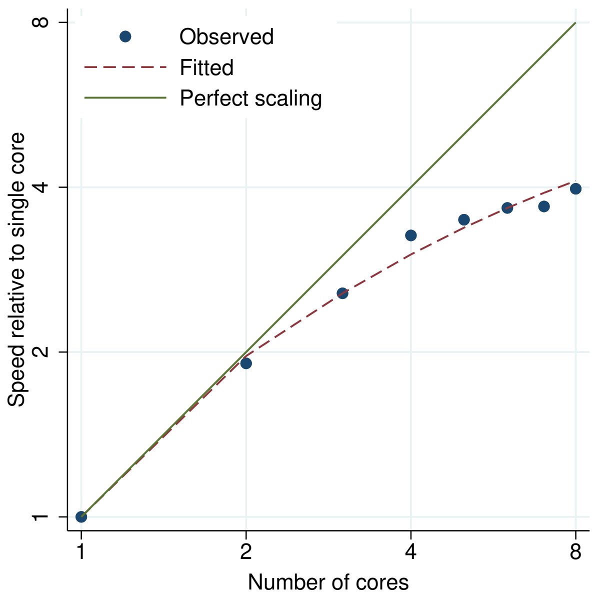


Figure 216. `kdensity` performance plot.

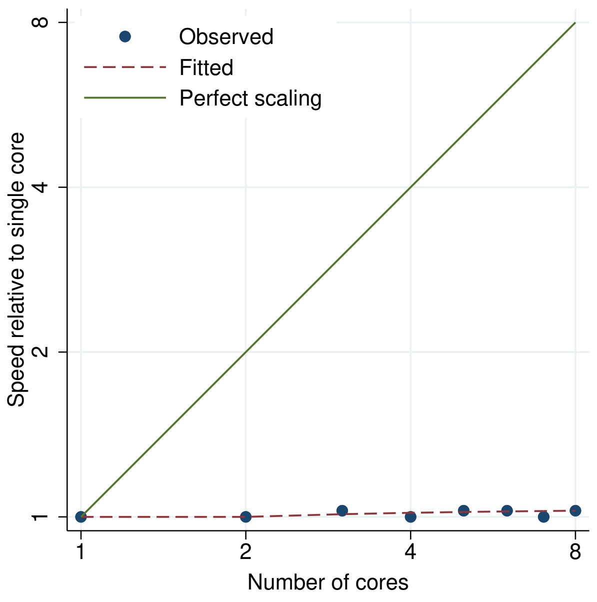


Figure 217. `keep if` *exp* performance plot.

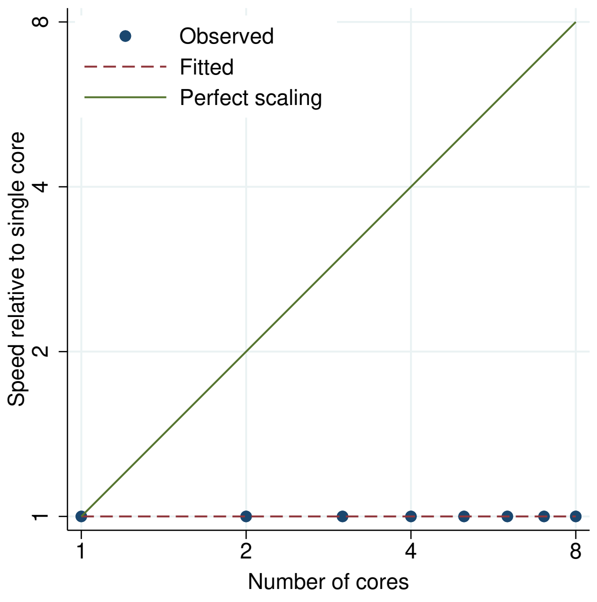


Figure 218. `keep in` *range* performance plot.

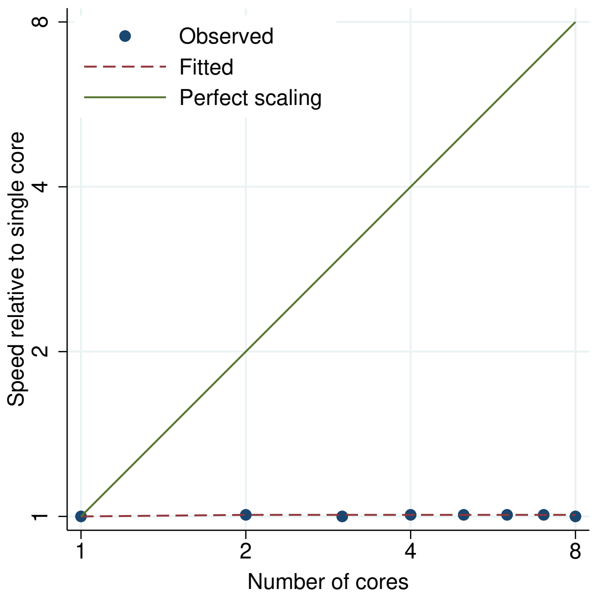


Figure 219. `keep` *varlist* performance plot.

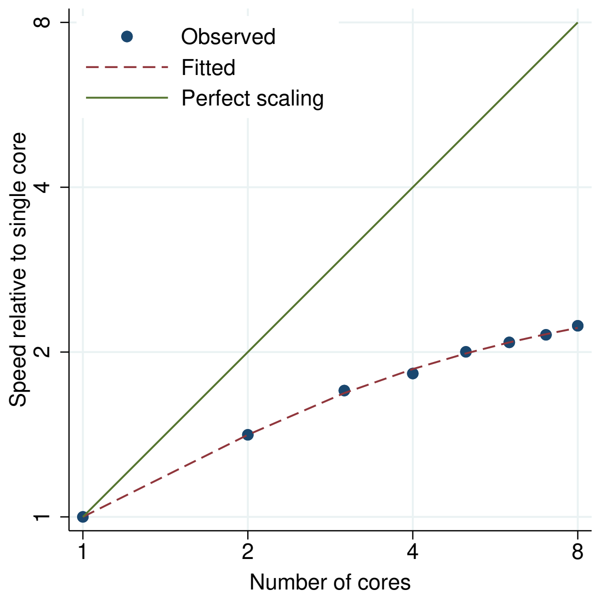


Figure 220. `ksmirnov` performance plot.

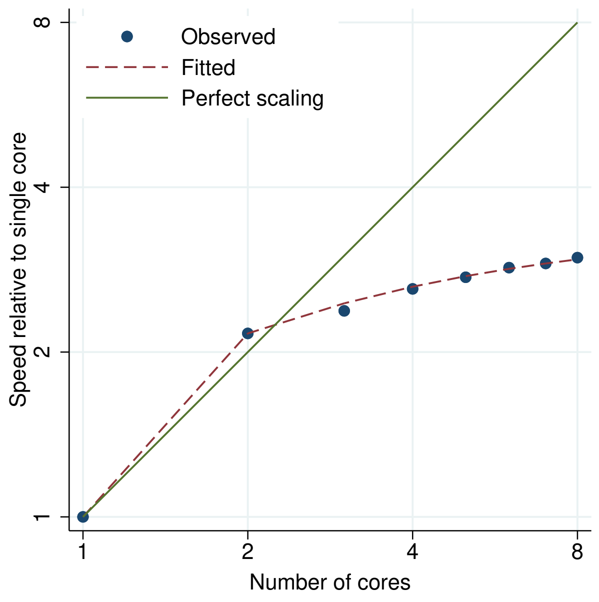


Figure 221. `ksmirnov, by()` performance plot.

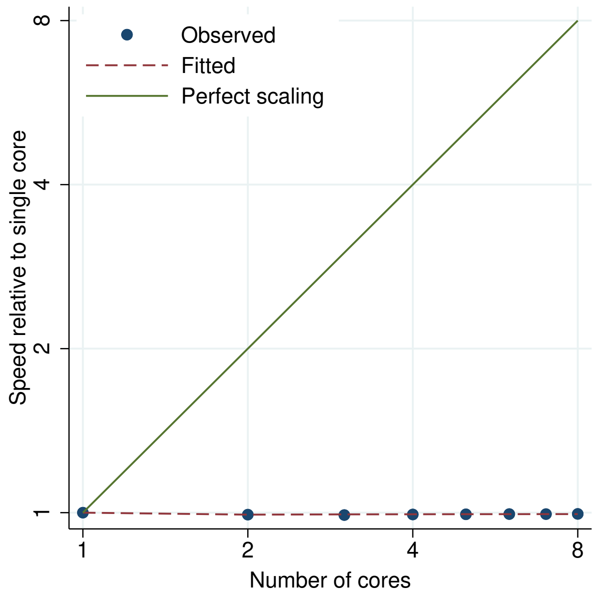


Figure 222. `ktau` performance plot.

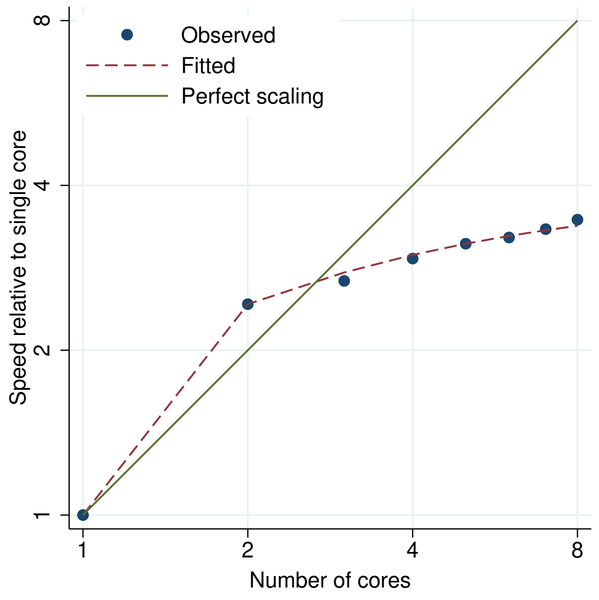


Figure 223. `kwallis` performance plot.

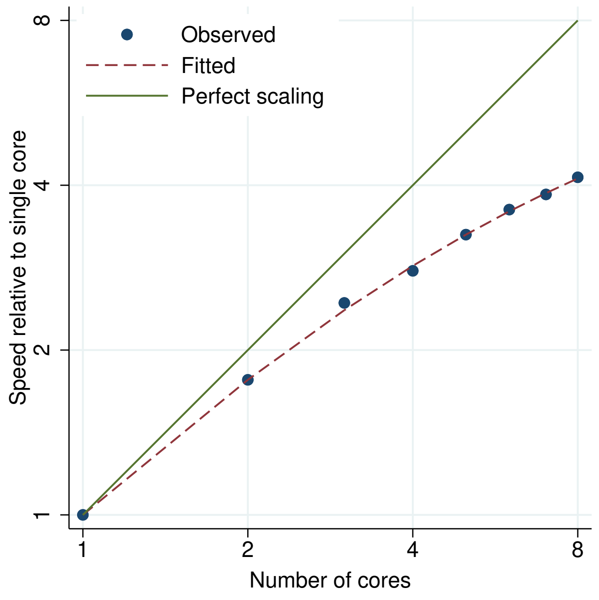


Figure 224. `ladder` performance plot.

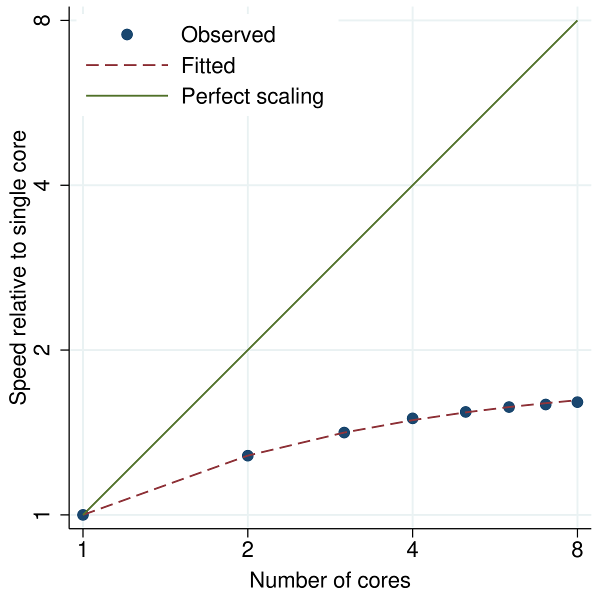


Figure 225. `lasso linear` performance plot.

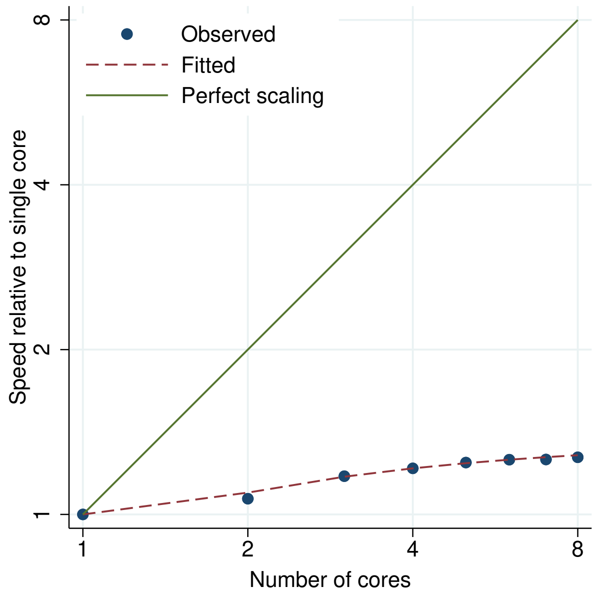


Figure 226. `lasso logit` performance plot.

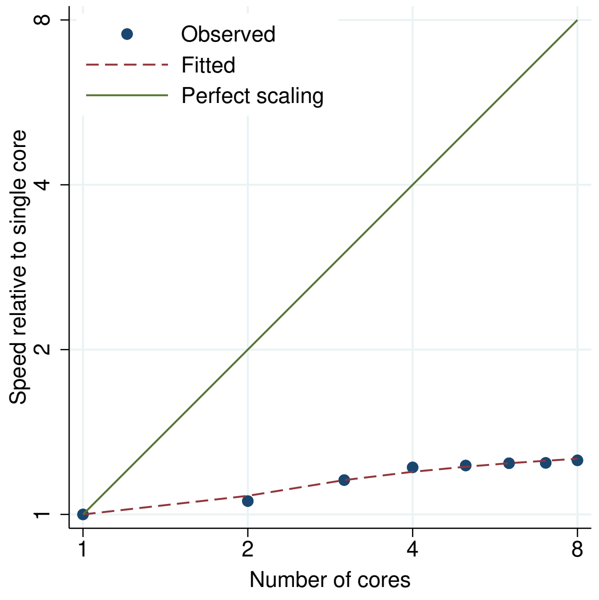


Figure 227. `lasso poisson` performance plot.

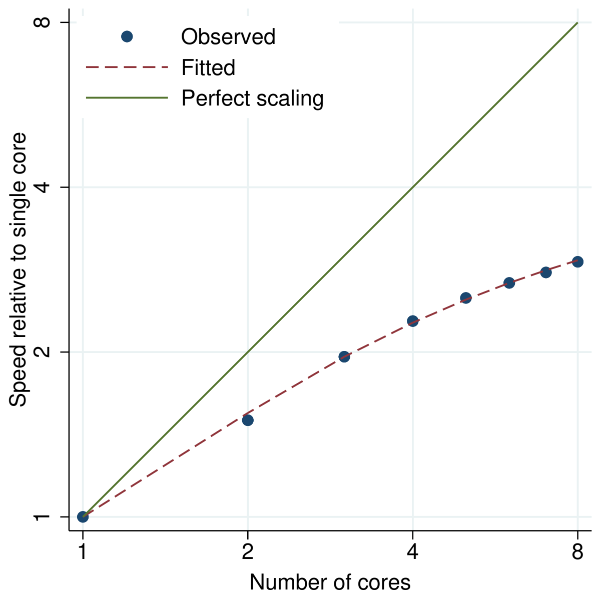


Figure 228. `gsem, lclass(C 2)` performance plot.

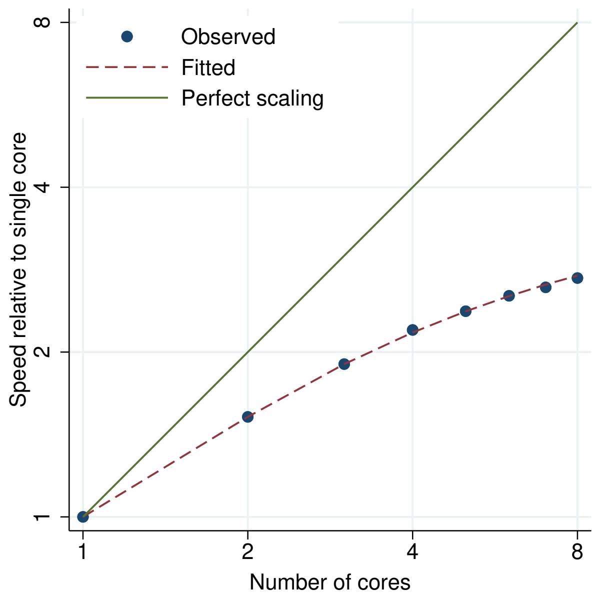


Figure 229. `gsem, lclass(C 3)` performance plot.

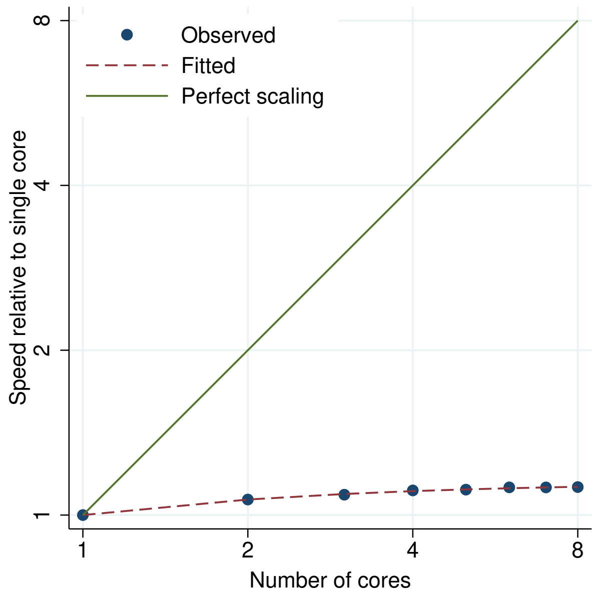


Figure 230. `levelsof` performance plot.

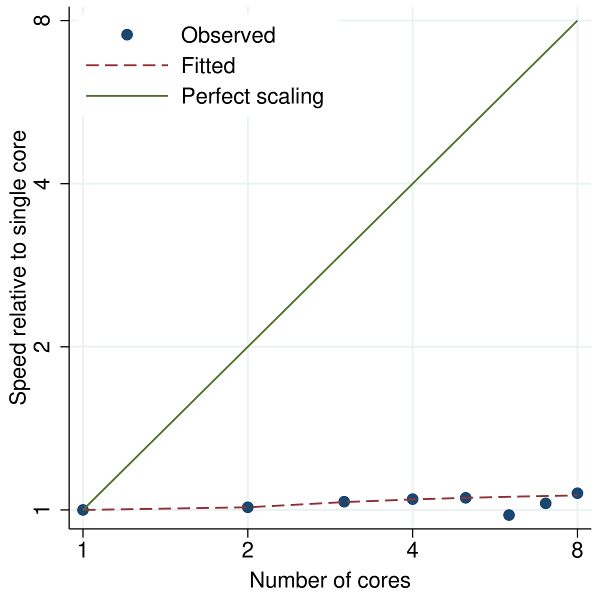


Figure 231. `loadingplot` performance plot.

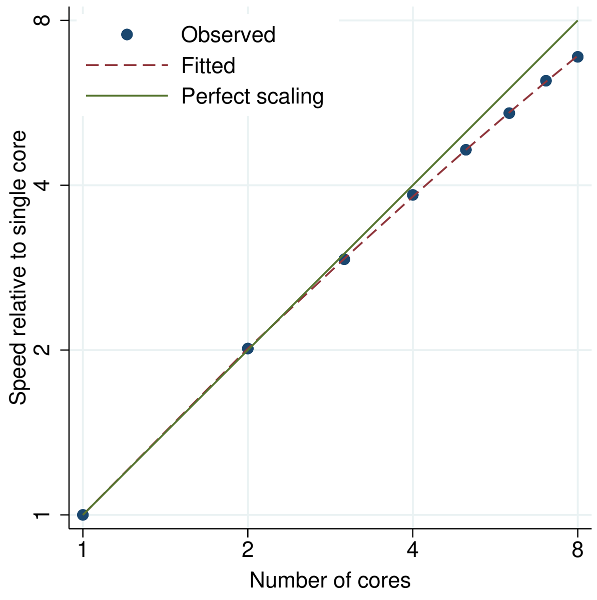


Figure 232. `logistic` performance plot.

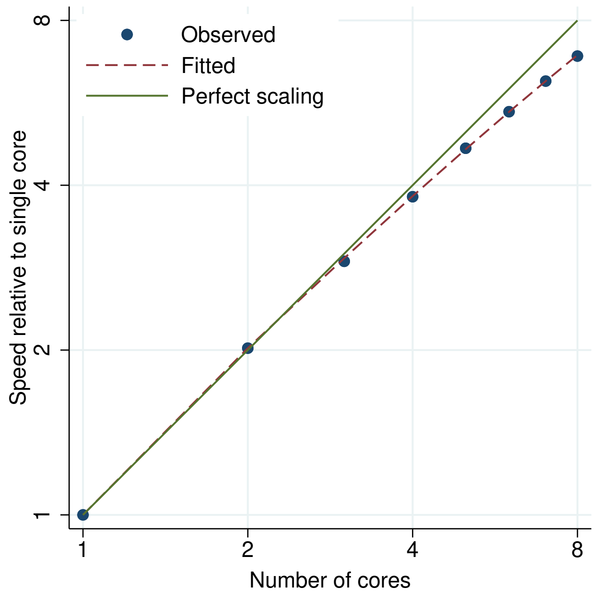


Figure 233. `logit` performance plot.

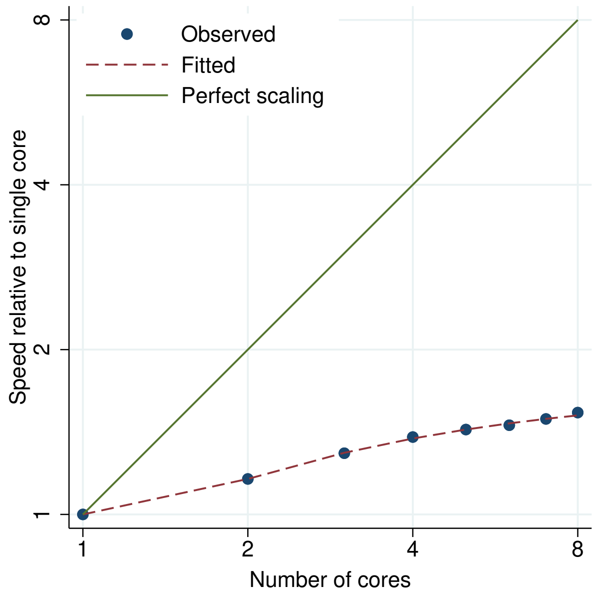


Figure 234. `loneway` performance plot.

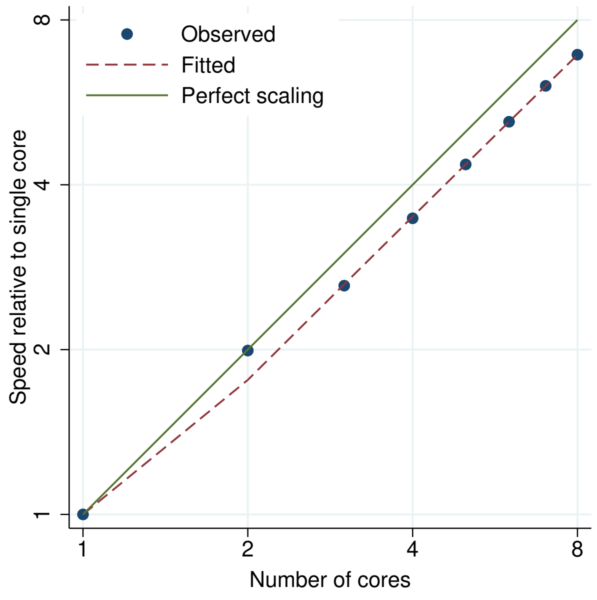


Figure 235. `lowess` performance plot.

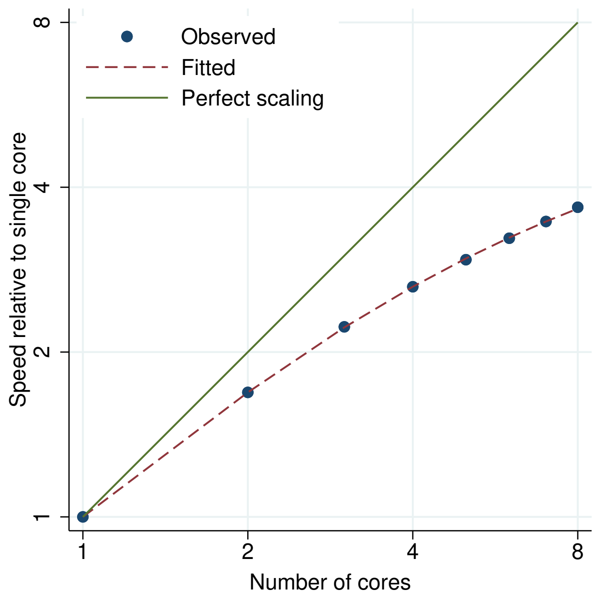


Figure 236. `lpoly` performance plot.

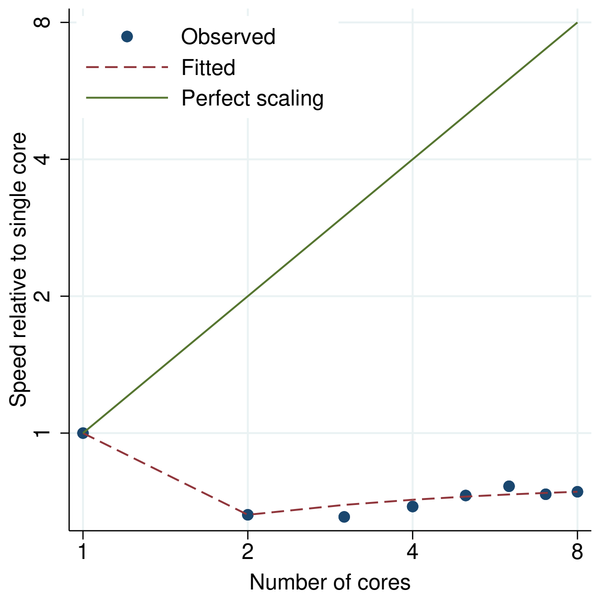


Figure 237. `ltable` performance plot.

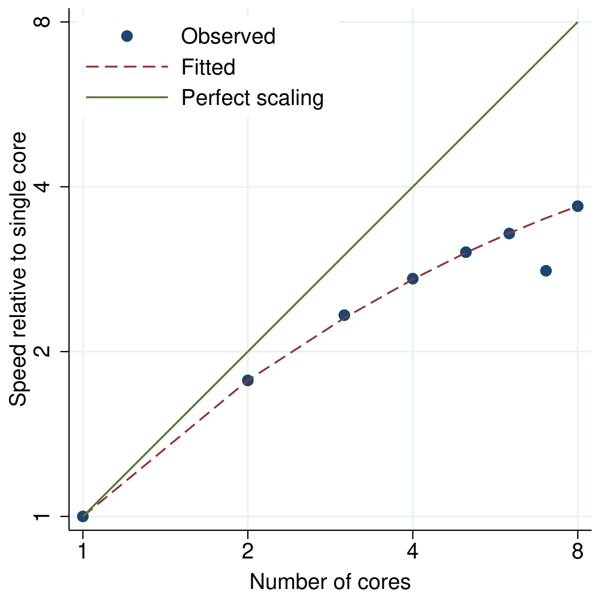


Figure 238. `manova` (one-way) performance plot.

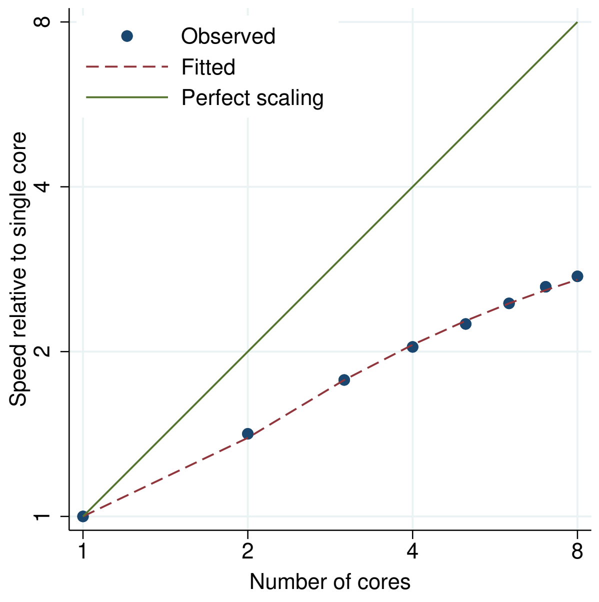


Figure 239. `manova` (two-way) performance plot.

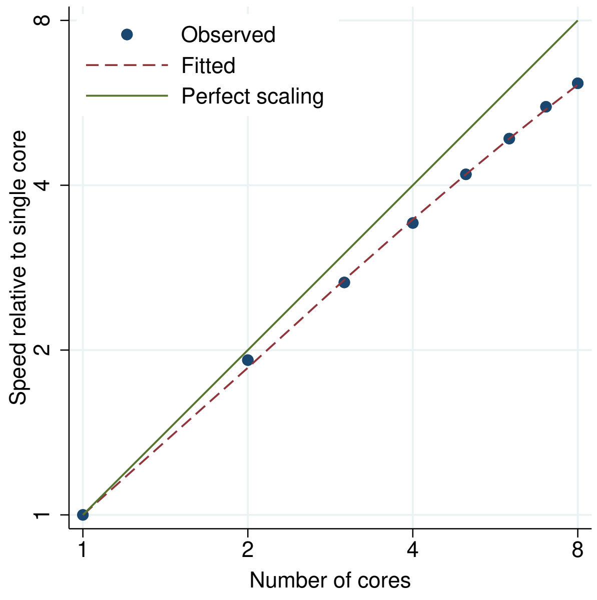


Figure 240. `margins` performance plot.

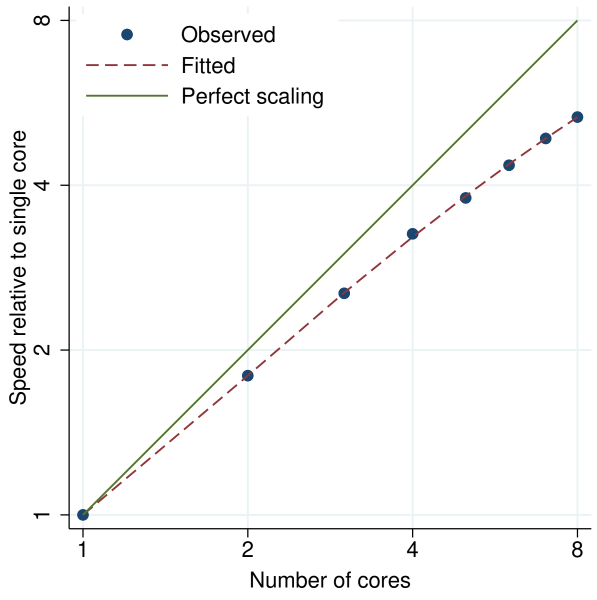


Figure 241. `margins, dydx() exp()` performance plot.

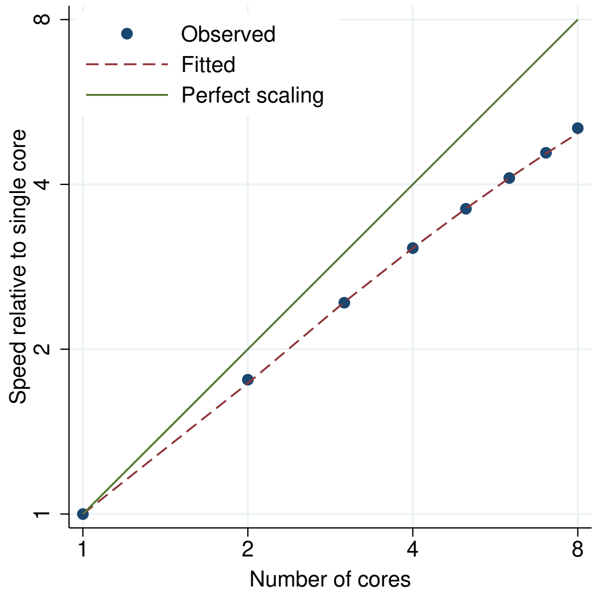


Figure 242. `margins, dydx()` performance plot.

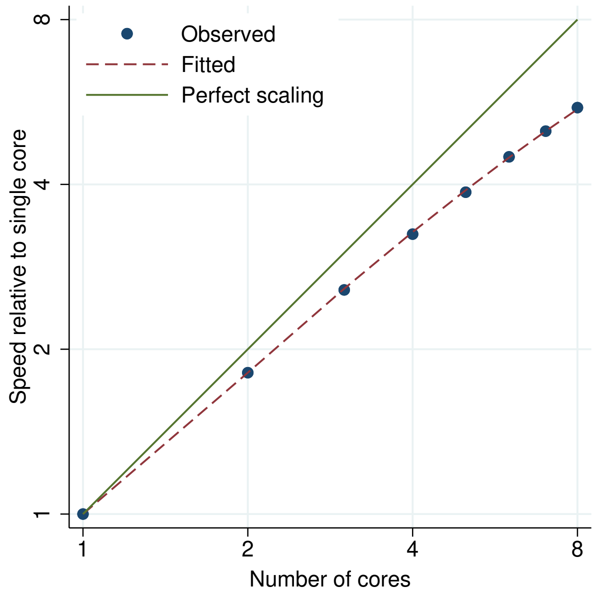


Figure 243. `margins, exp()` performance plot.

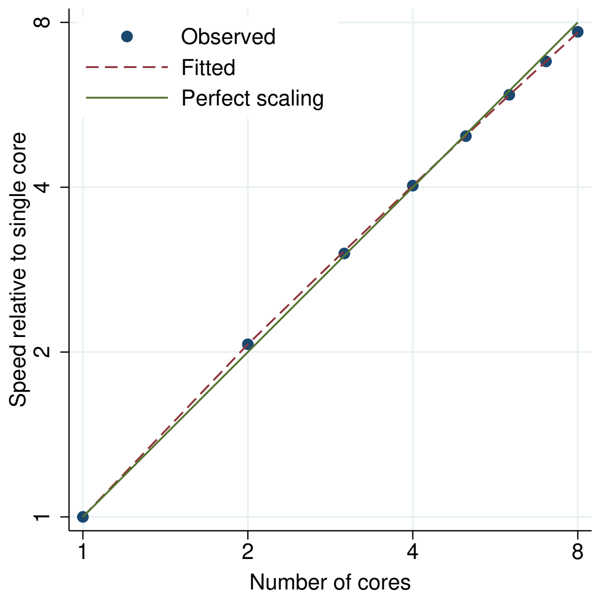


Figure 244. `markout` performance plot.

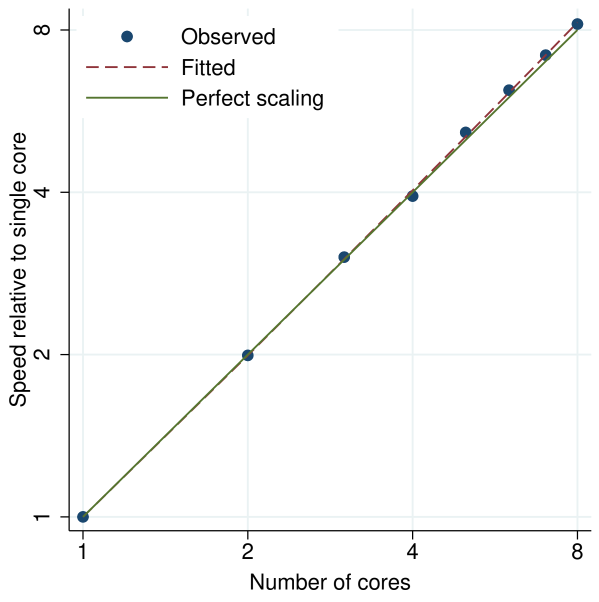


Figure 245. `marksample` performance plot.

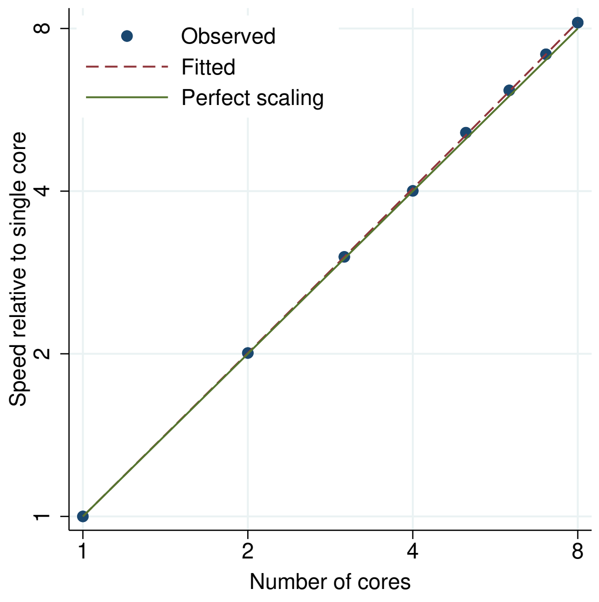


Figure 246. `marksample if` *exp* performance plot.

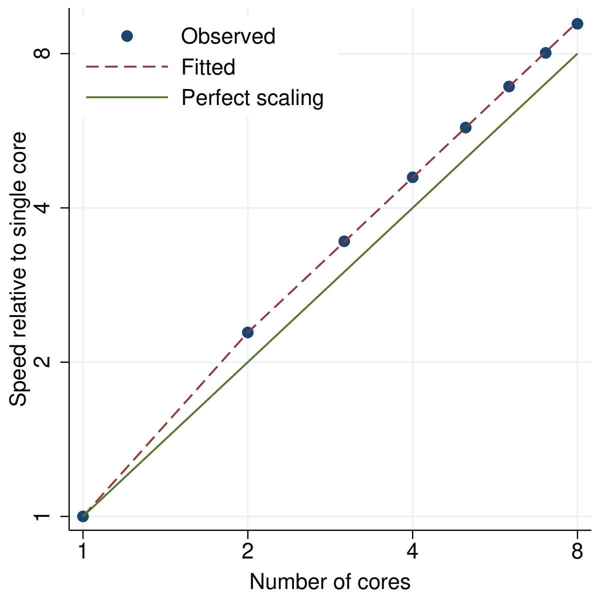


Figure 247. `matrix accum` performance plot.

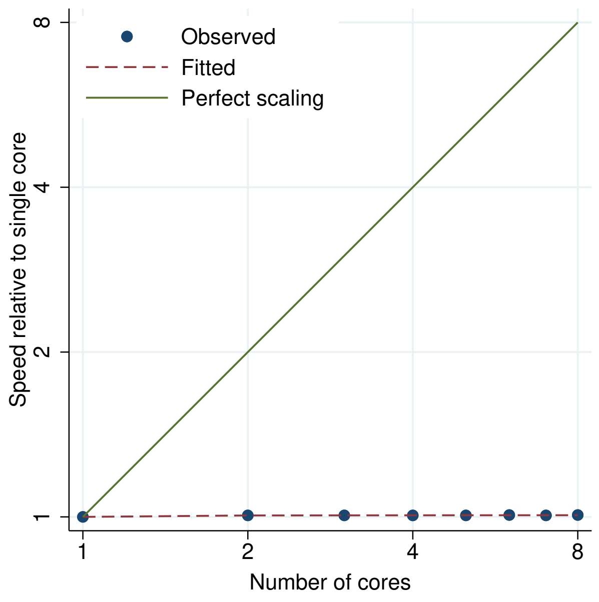


Figure 248. `matrix eigenvalues` performance plot.

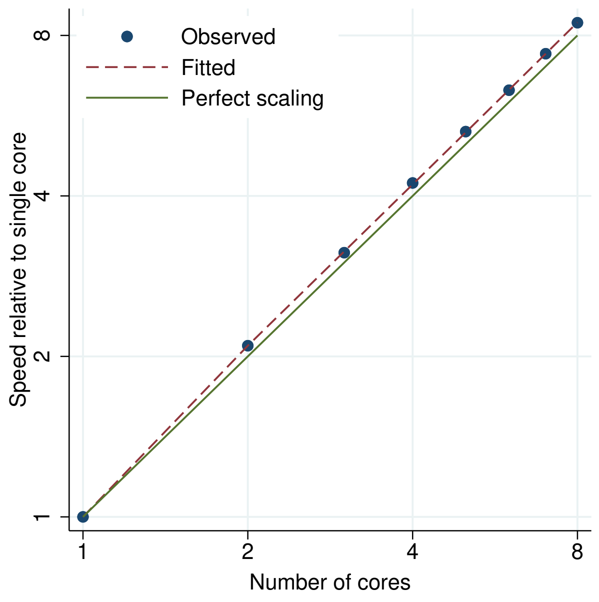


Figure 249. `matrix score` performance plot.

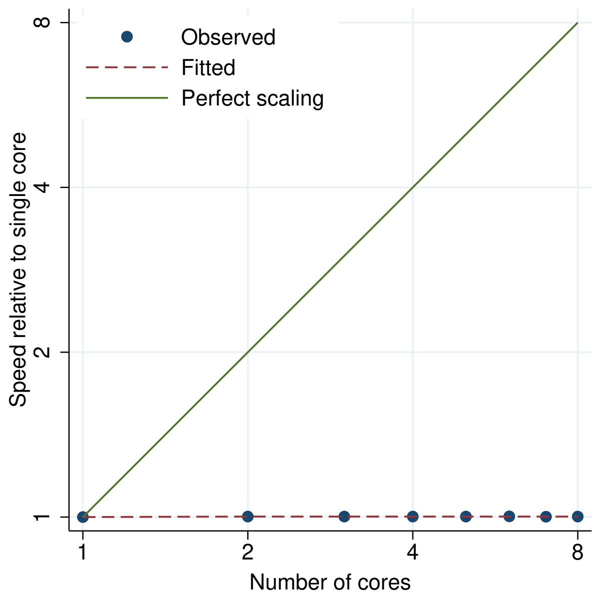


Figure 250. `matrix svd` performance plot.

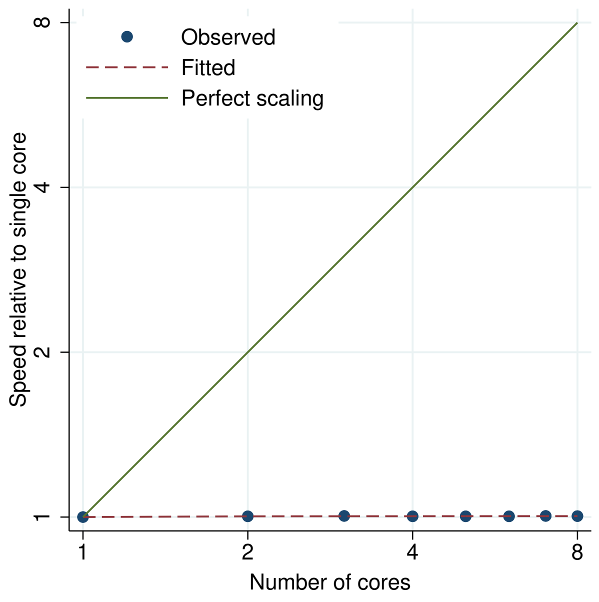


Figure 251. `matrix symeigen` performance plot.

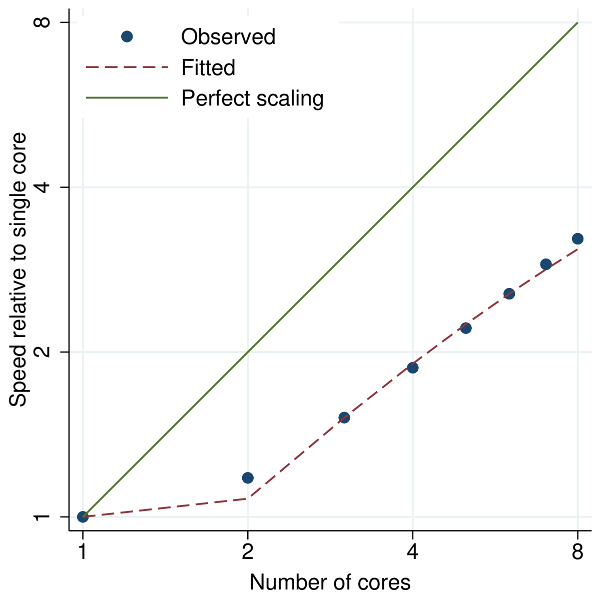


Figure 252. `matrix syminv` performance plot.

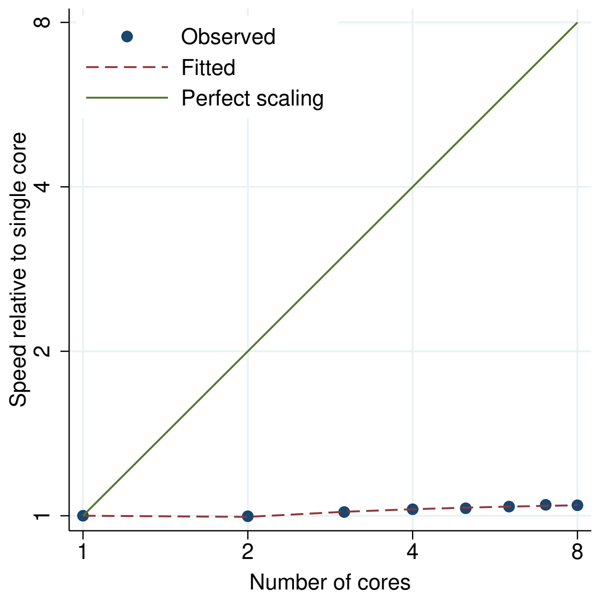


Figure 253. `mca` performance plot.

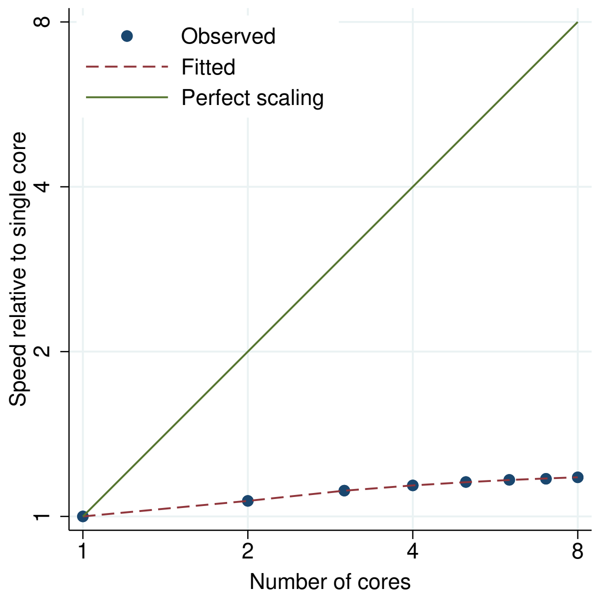


Figure 254. `mcc` performance plot.

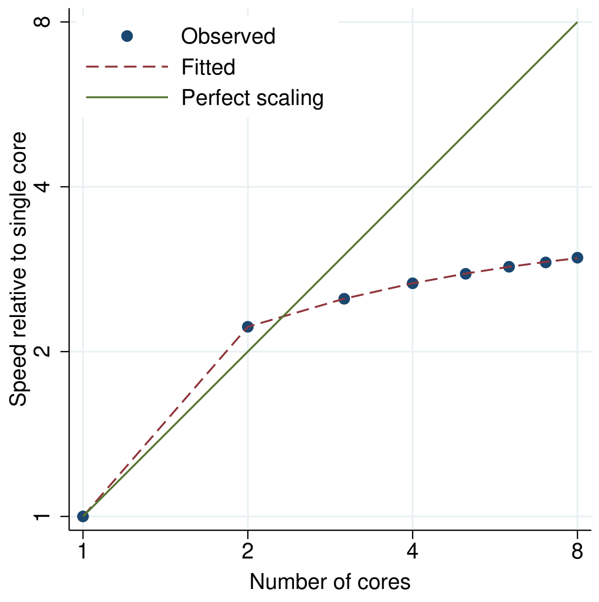


Figure 255. `mds` performance plot.

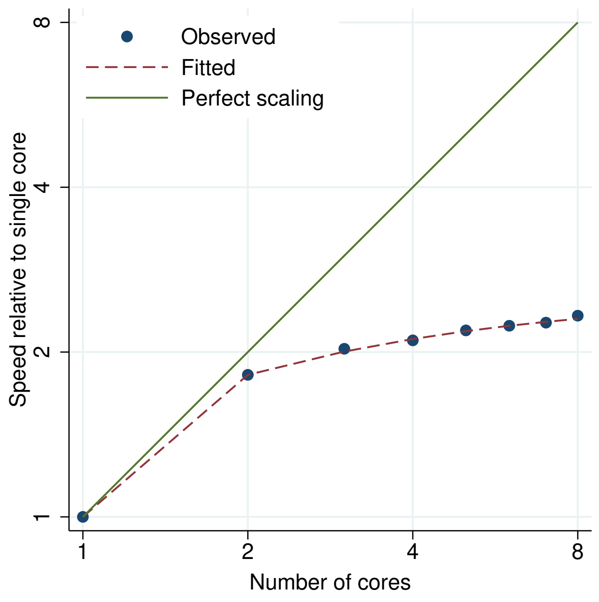


Figure 256. `mdslong` performance plot.

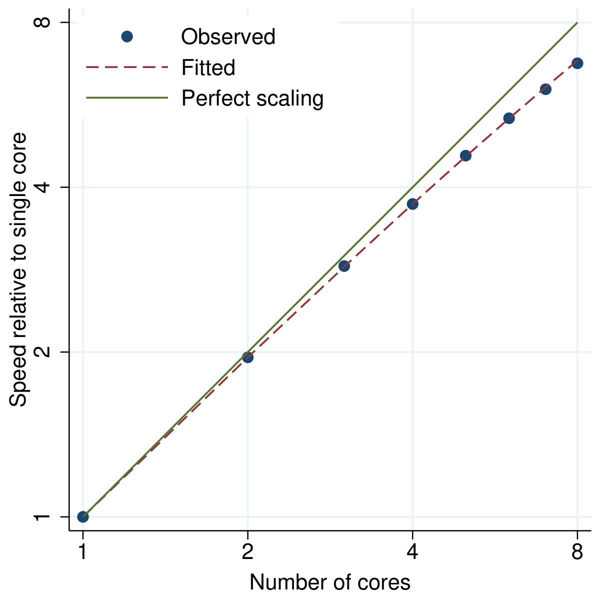


Figure 257. `mean` performance plot.

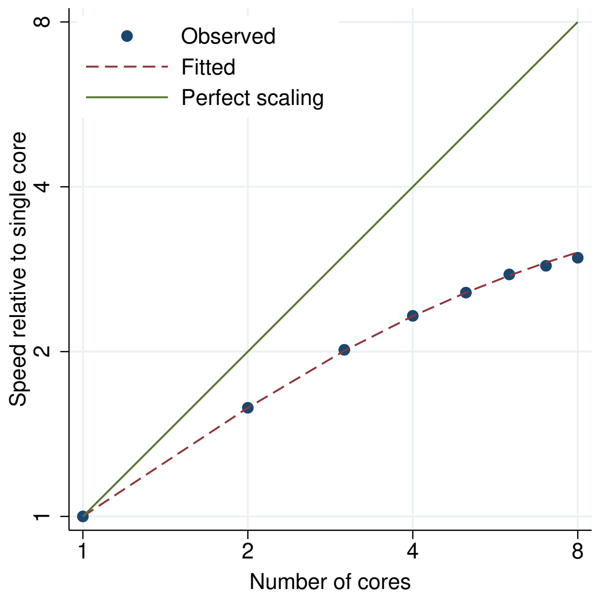


Figure 258. `mecloglog` performance plot.

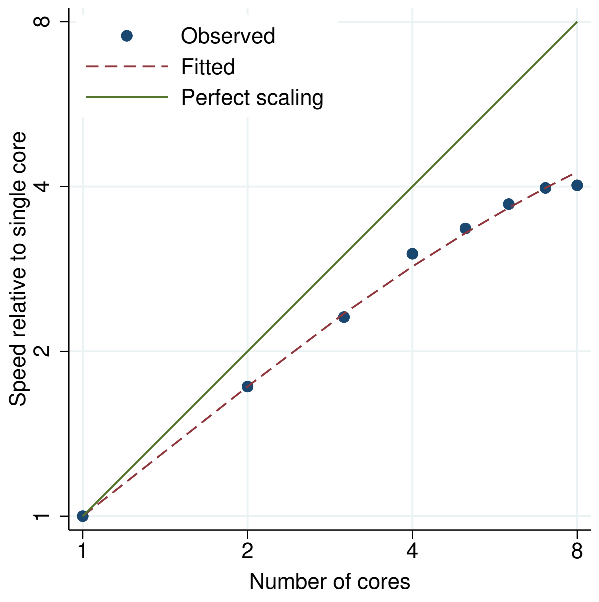


Figure 259. `median` performance plot.

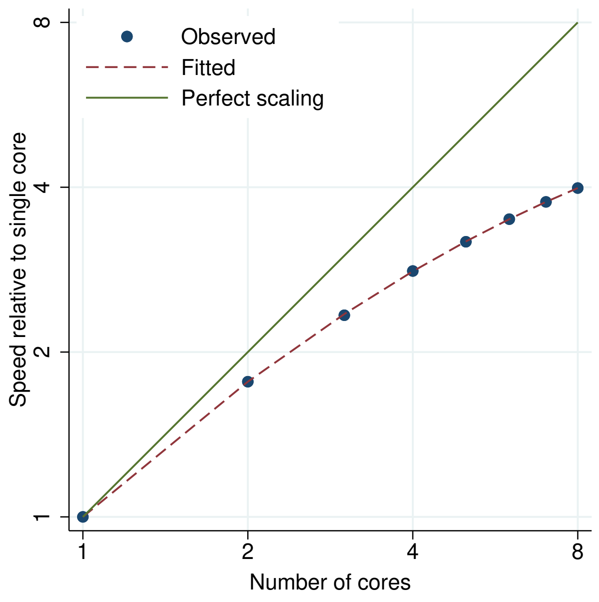


Figure 260. `meintreg` performance plot.

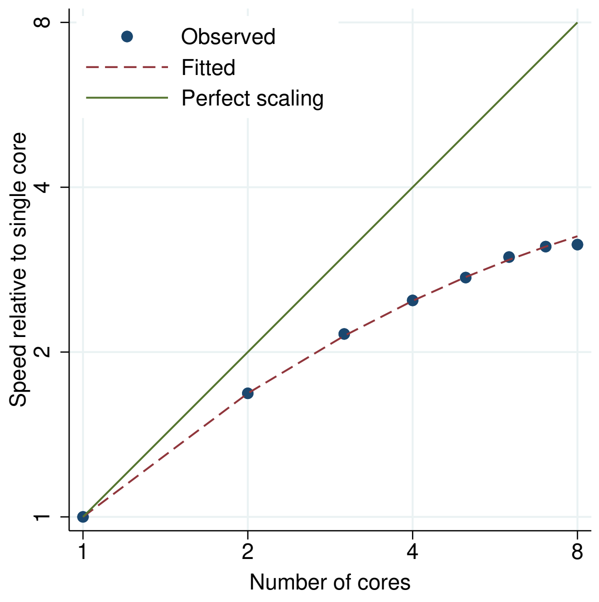


Figure 261. `melogit` performance plot.

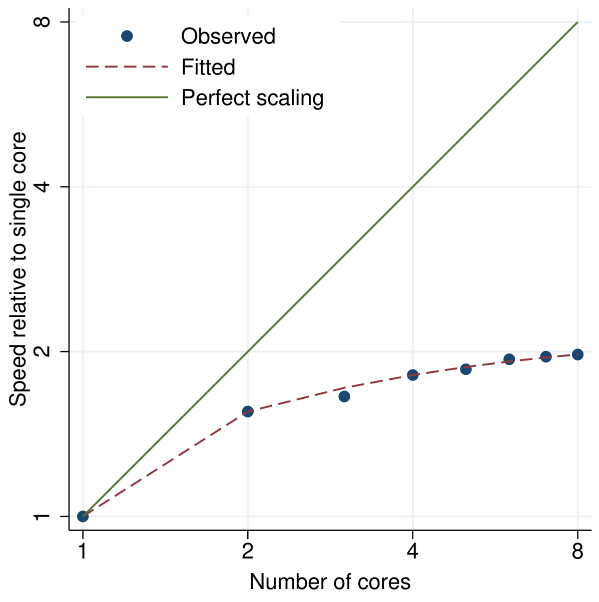


Figure 262. `menbreg, dispersion(constant)` performance plot.

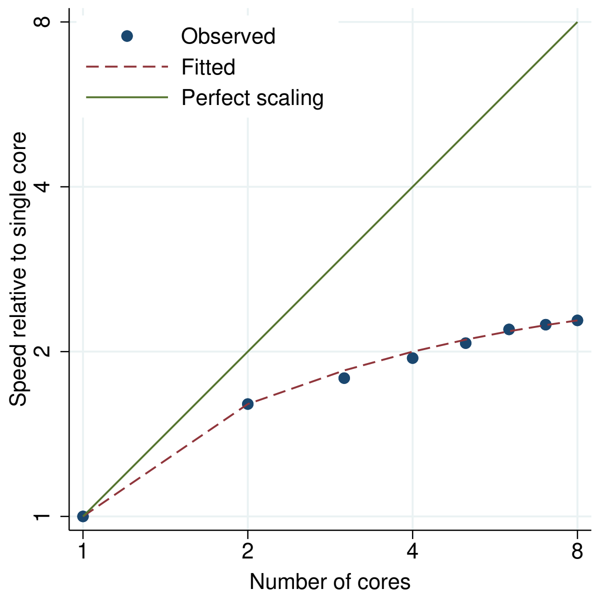


Figure 263. `menbreg, dispersion(mean)` performance plot.

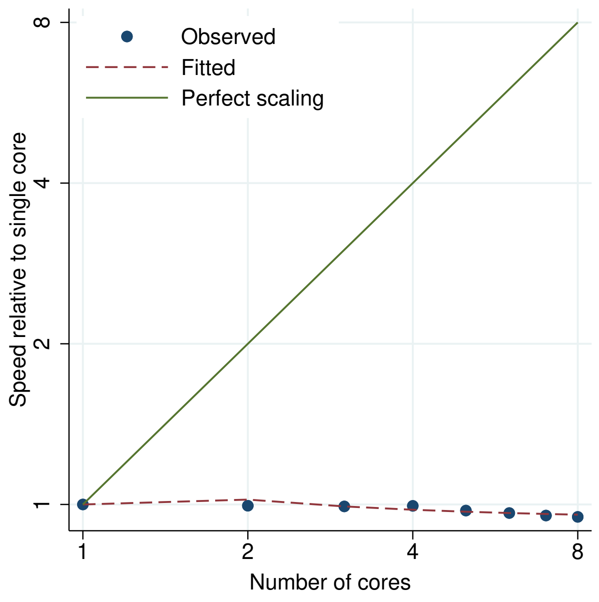


Figure 264. `menl` performance plot.

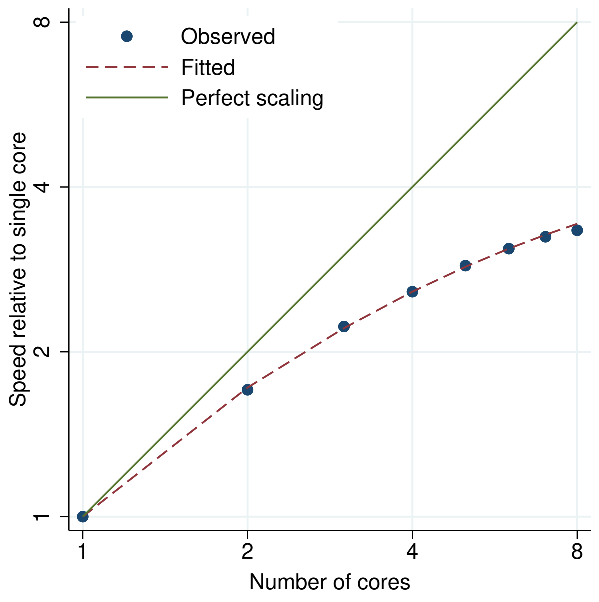


Figure 265. `meologit` performance plot.

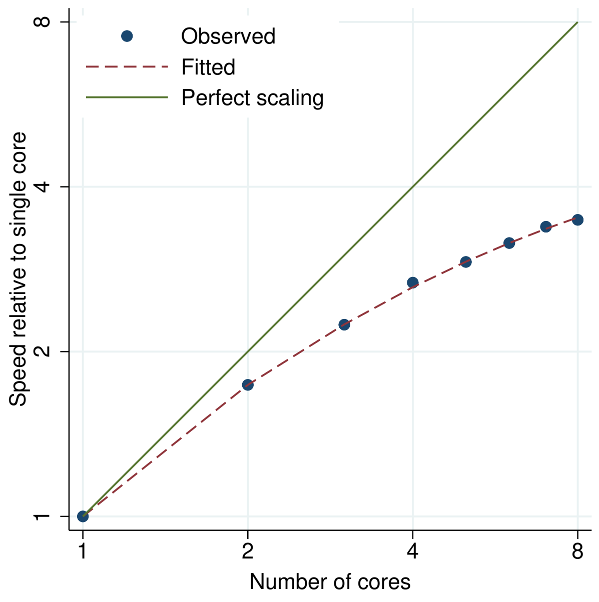


Figure 266. `meoprobit` performance plot.

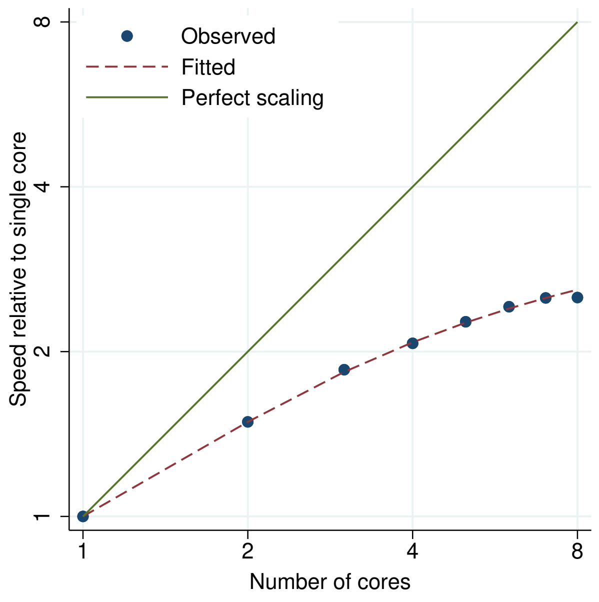


Figure 267. `mepoisson` performance plot.

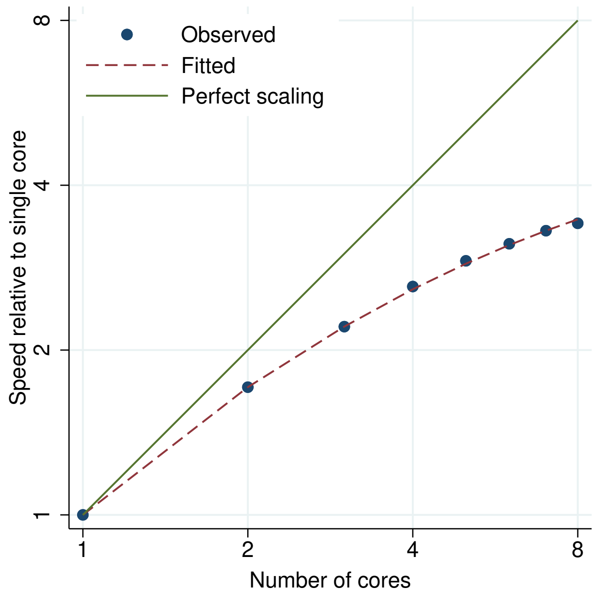


Figure 268. `meprobit` performance plot.

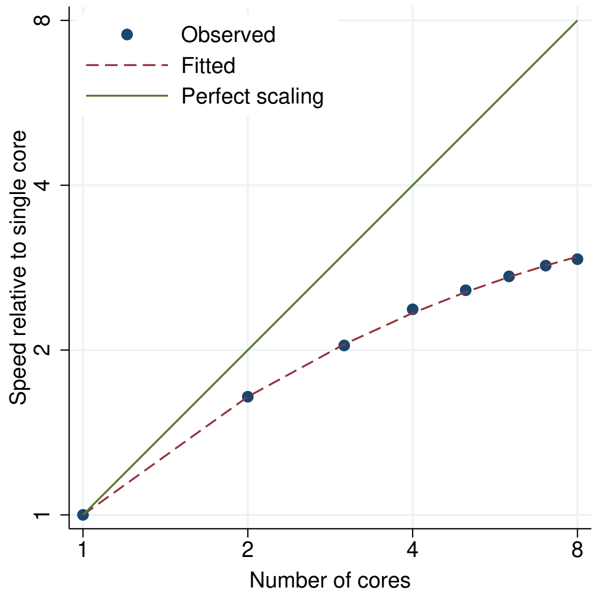


Figure 269. `mestreg, distribution(exp)` performance plot.

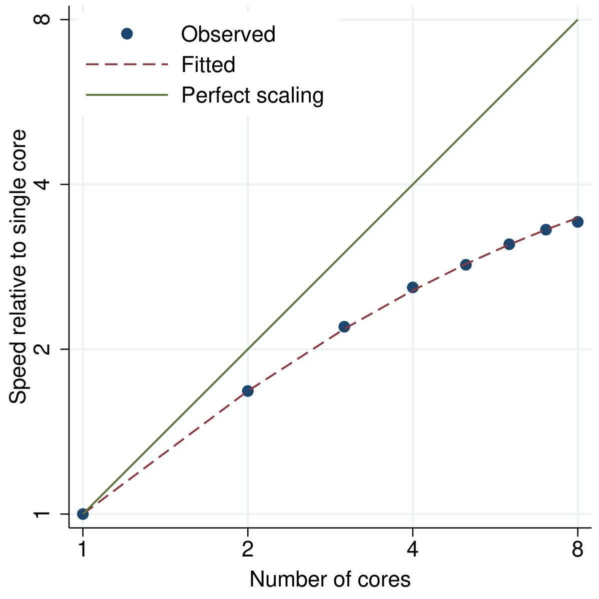


Figure 270. `mestreg, distribution(weibull)` performance plot.

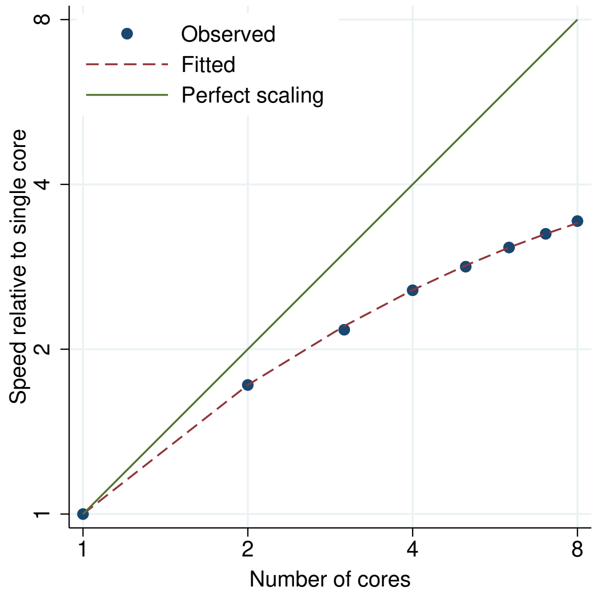


Figure 271. `metobit` performance plot.

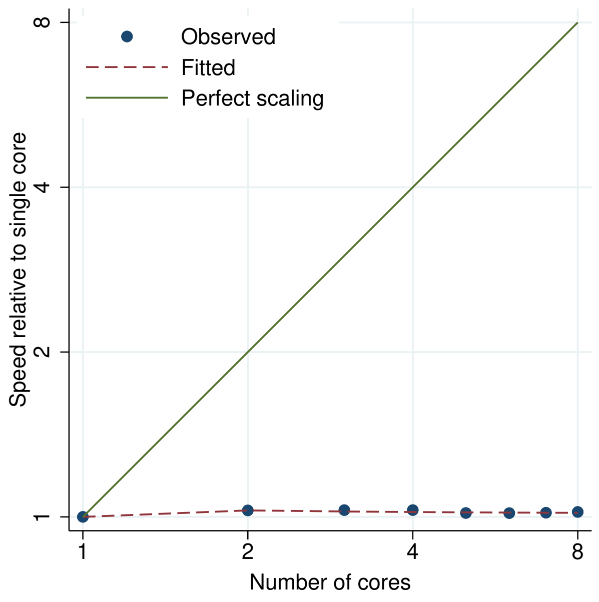


Figure 272. `mgarch` performance plot.

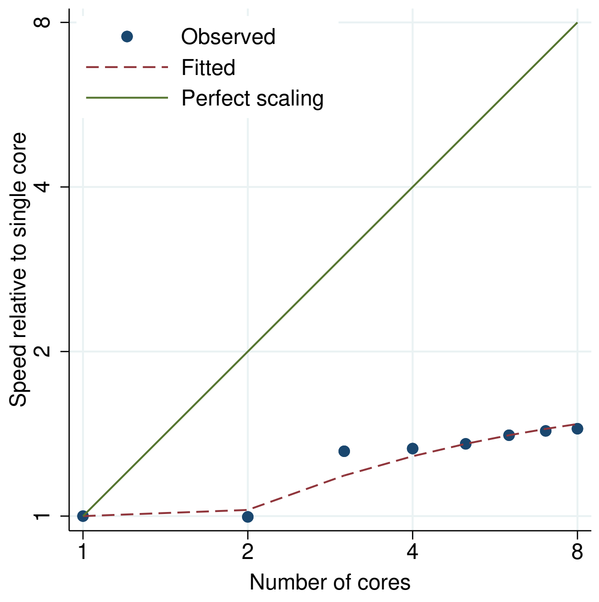


Figure 273. `mhodds` performance plot.

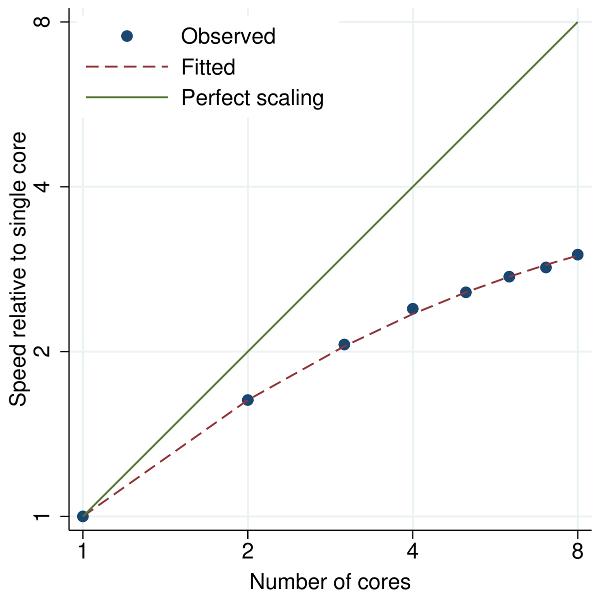


Figure 274. `mhodds` (adjusted) performance plot.

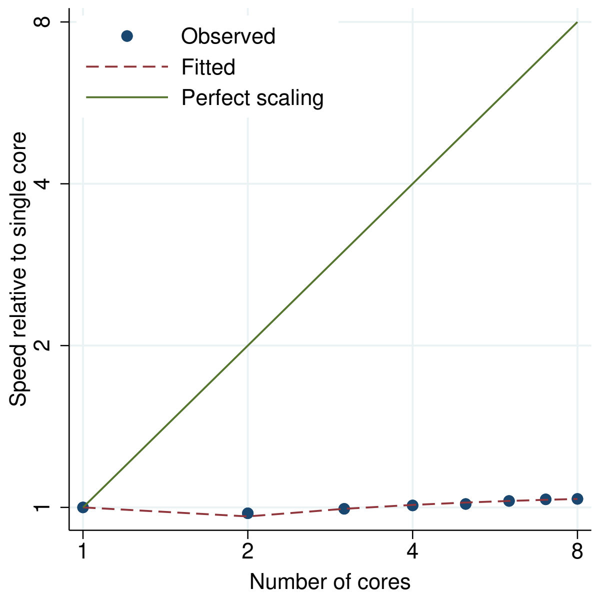


Figure 275. `by: mhodds` performance plot.

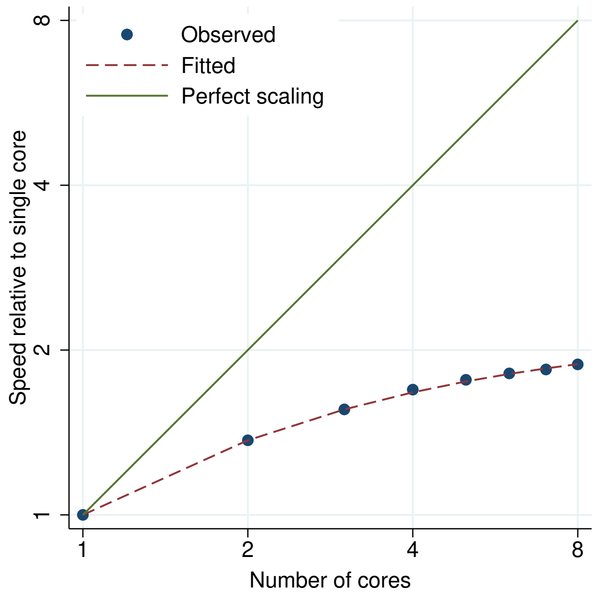


Figure 276. `mhodds` (trend) performance plot.

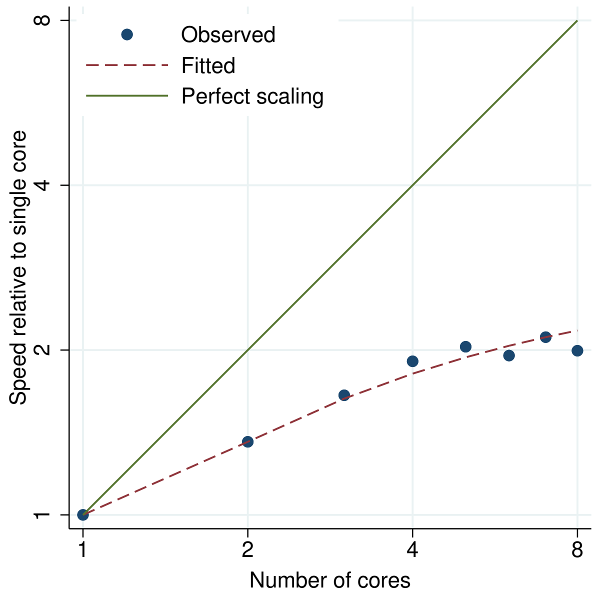


Figure 277. `mi estimate: logit (flong)` performance plot.

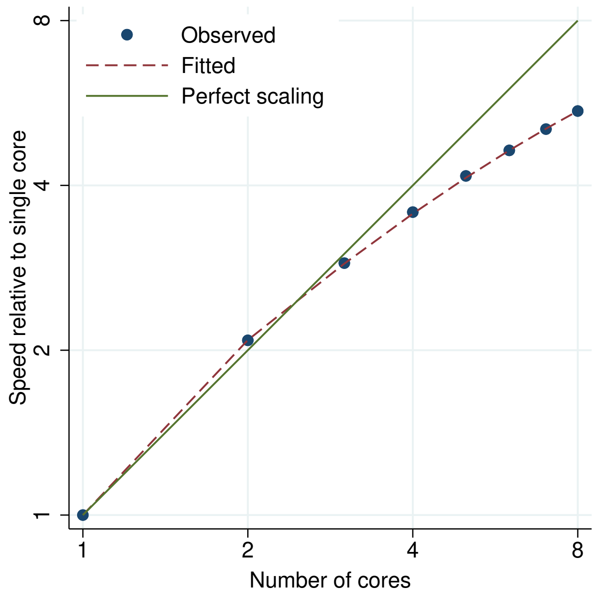


Figure 278. `mi estimate: logit (flongsep)` performance plot.

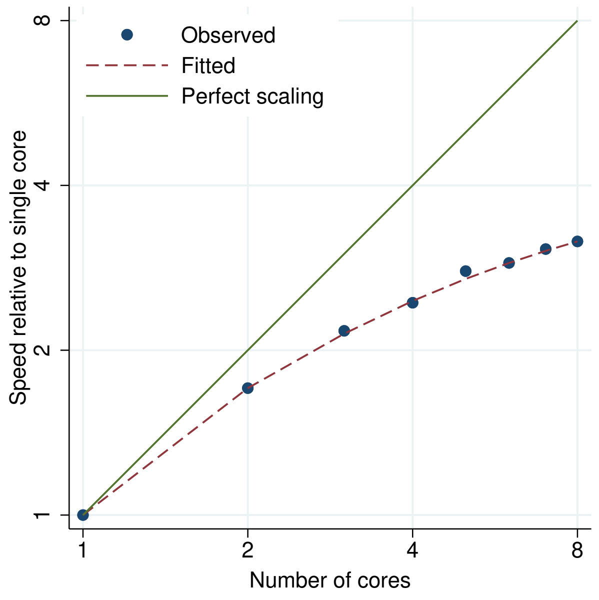


Figure 279. `mi estimate: logit (mlong)` performance plot.

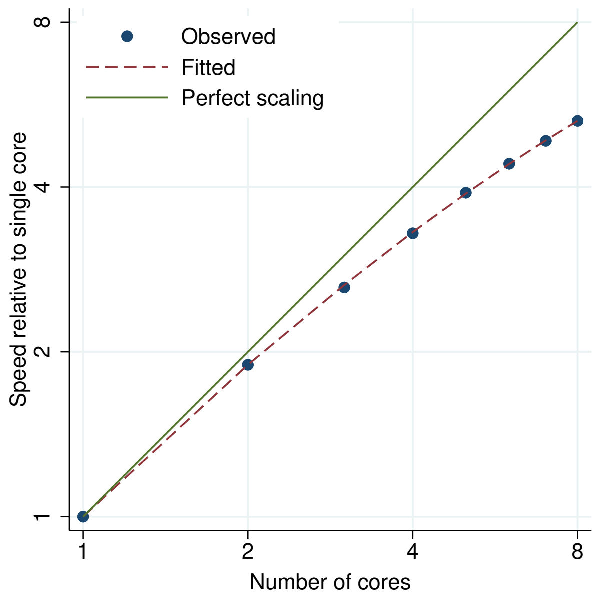


Figure 280. `mi estimate: logit (wide)` performance plot.

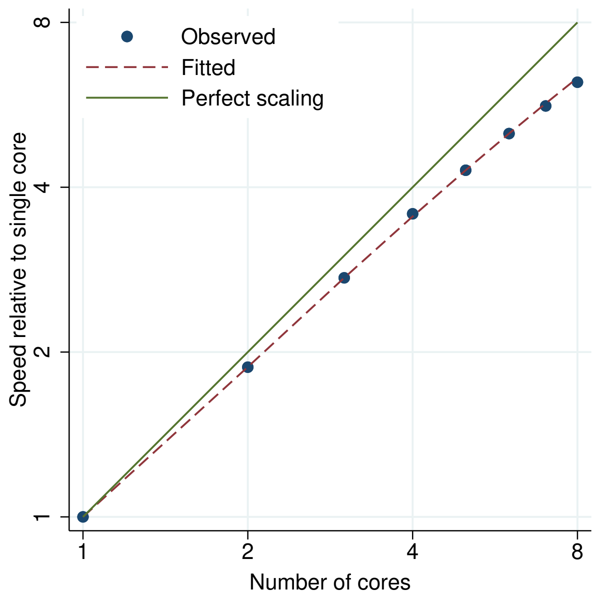


Figure 281. `mi estimate: mlogit` performance plot.

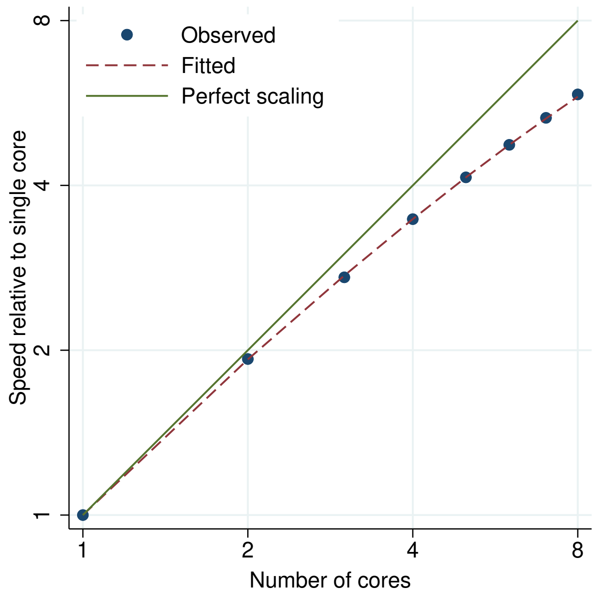


Figure 282. `mi estimate: ologit` performance plot.

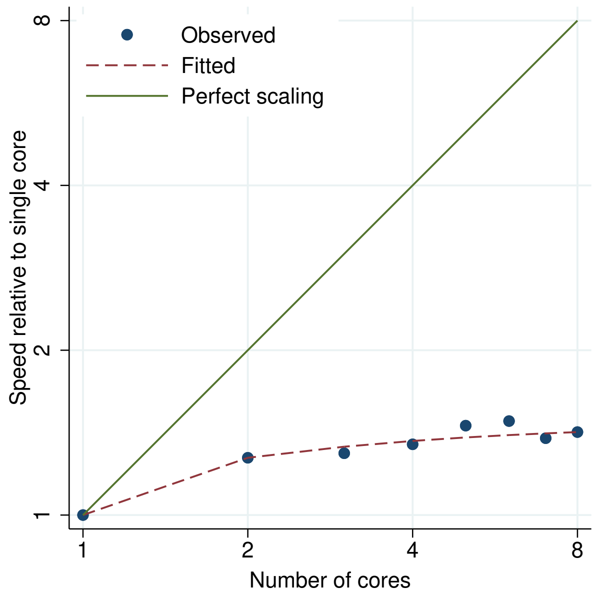


Figure 283. `mi estimate: regress (flong)` performance plot.

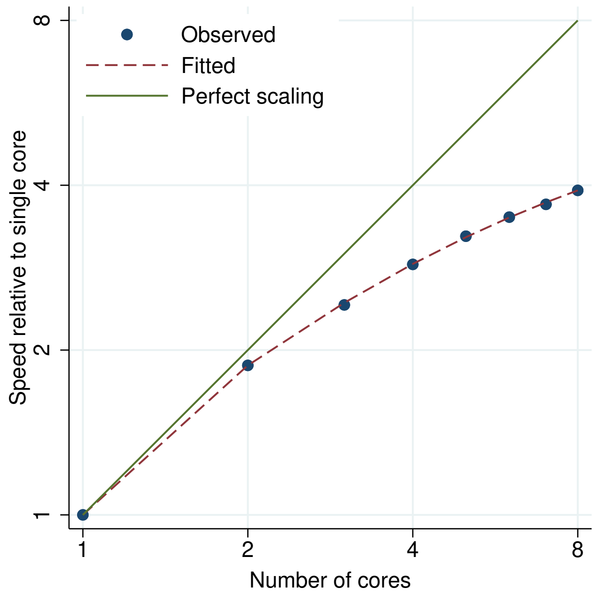


Figure 284. `mi estimate: regress (flongsep)` performance plot.

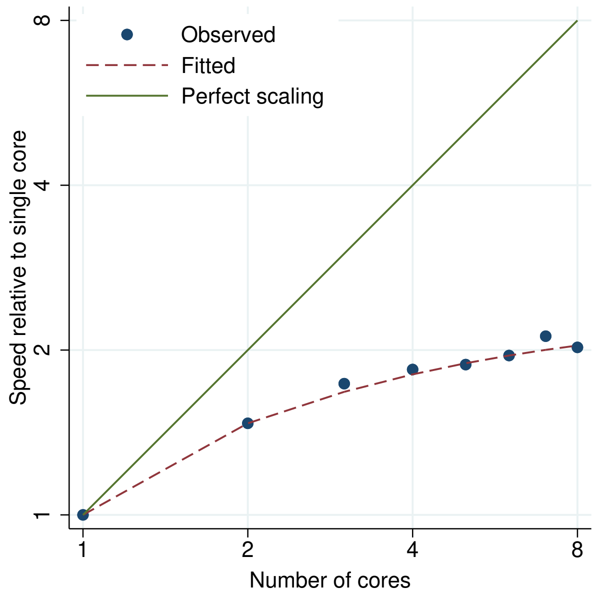


Figure 285. `mi estimate: regress (mlong)` performance plot.

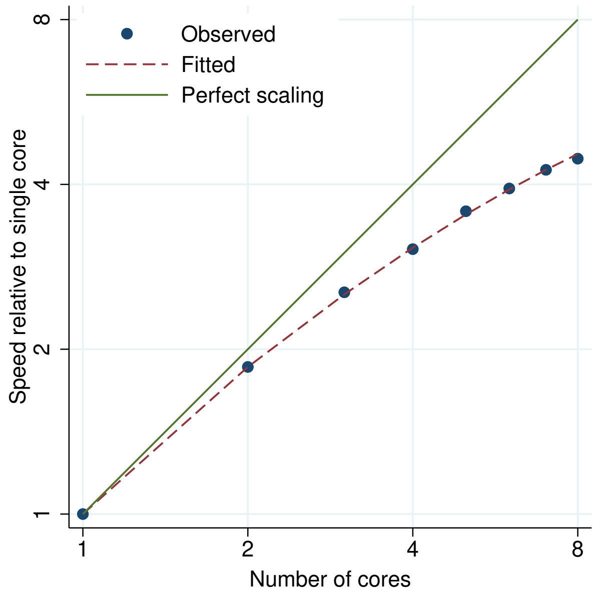


Figure 286. `mi estimate: regress (wide)` performance plot.

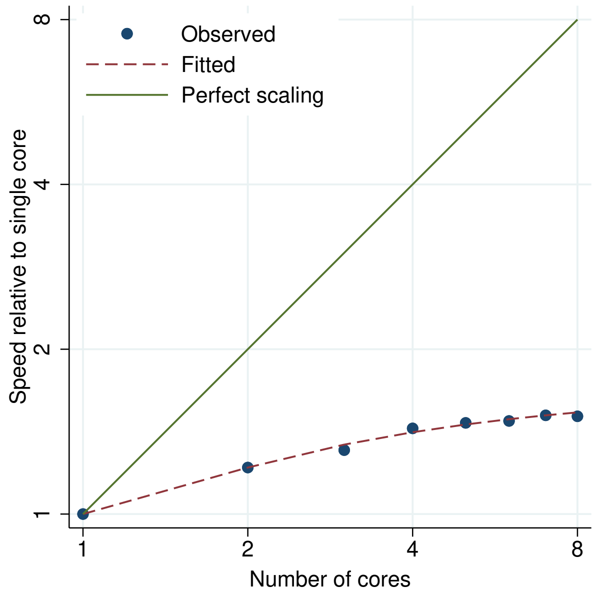


Figure 287. `mi impute chained (flong)` performance plot.

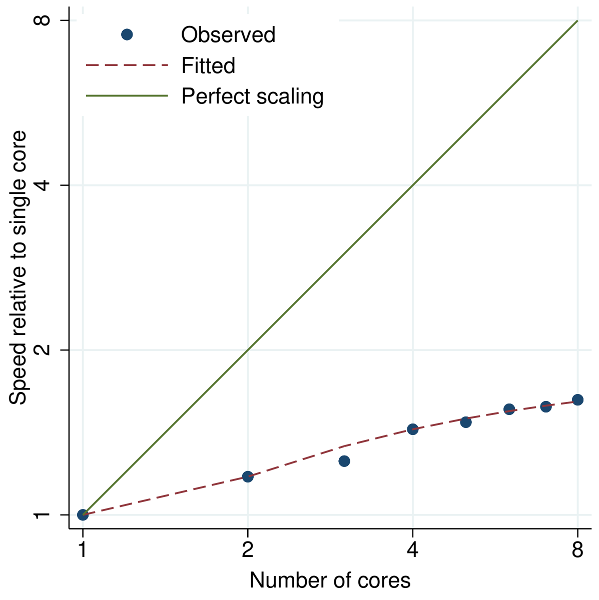


Figure 288. `mi impute chained (flongsep)` performance plot.

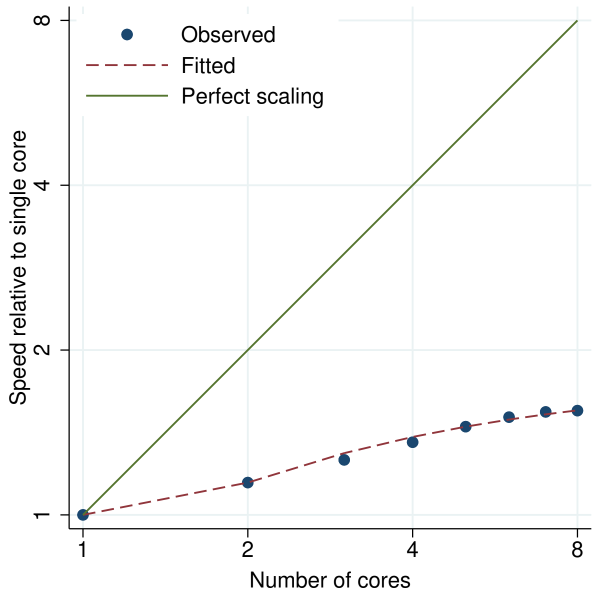


Figure 289. `mi impute chained (mlong)` performance plot.

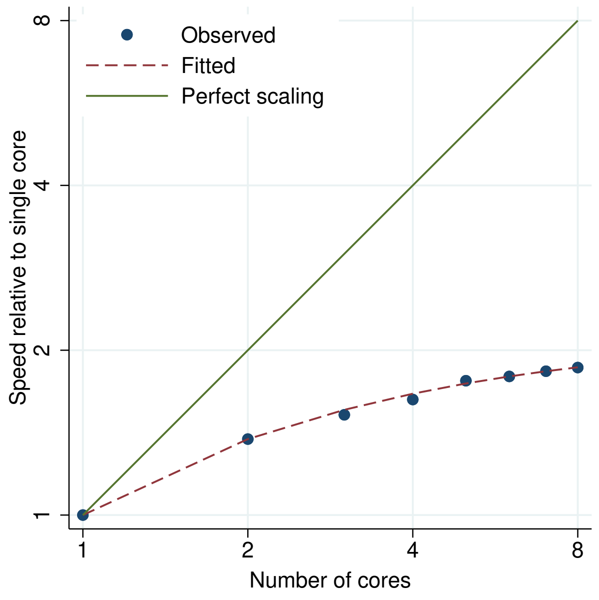


Figure 290. `mi impute chained (wide)` performance plot.

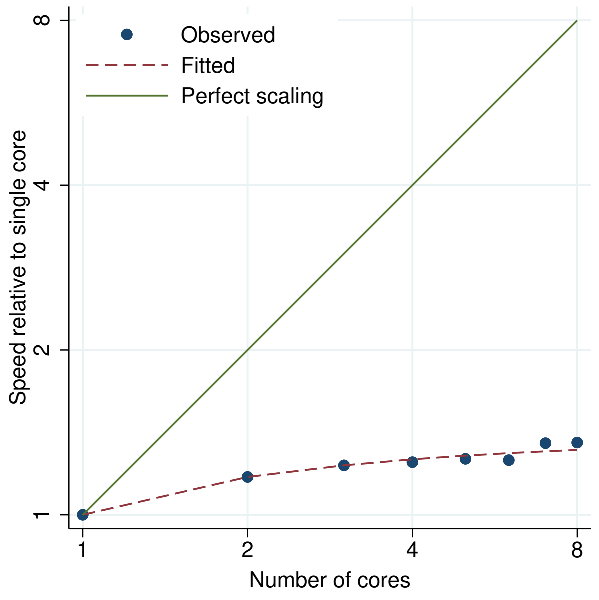


Figure 291. `mi impute logit (flong)` performance plot.

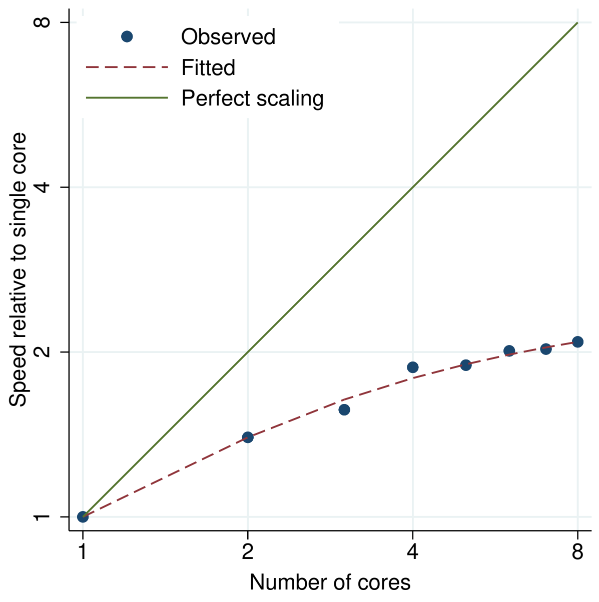


Figure 292. `mi impute logit (flongsep)` performance plot.

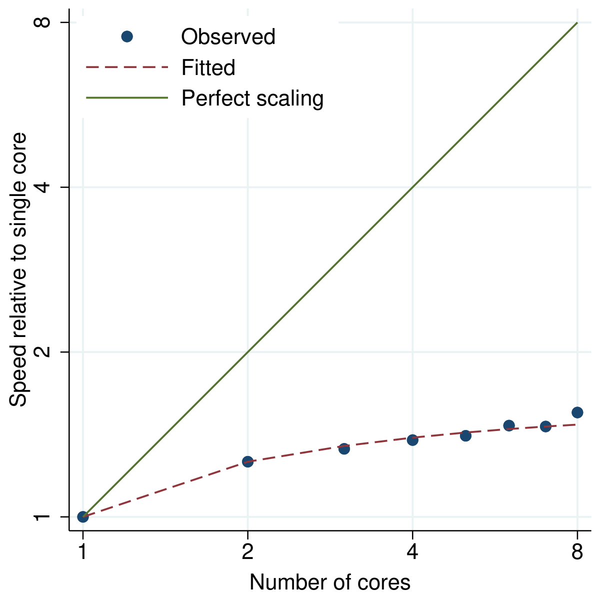


Figure 293. `mi impute logit (mlong)` performance plot.

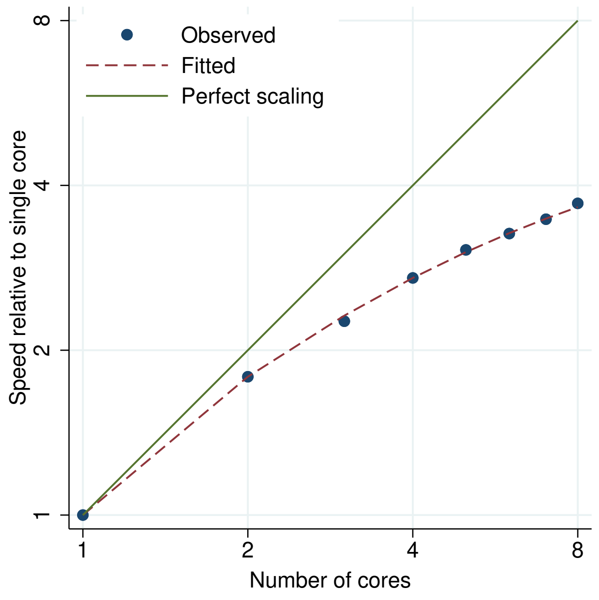


Figure 294. `mi impute logit (wide)` performance plot.

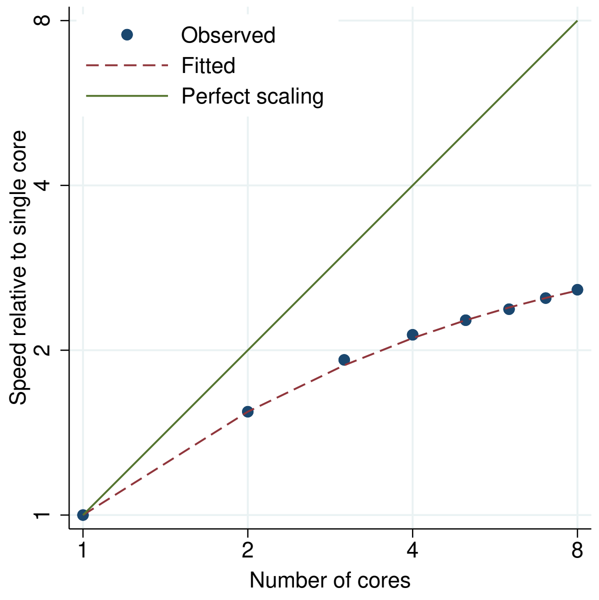


Figure 295. `mi impute mlogit` performance plot.

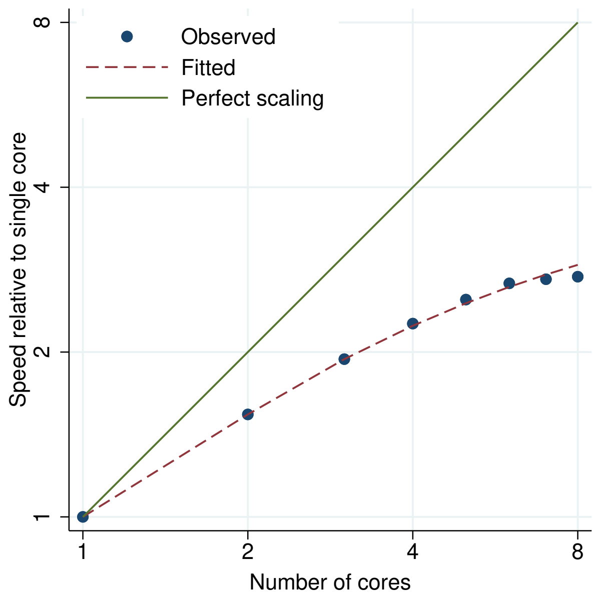


Figure 296. `mi impute mono pmm` performance plot.

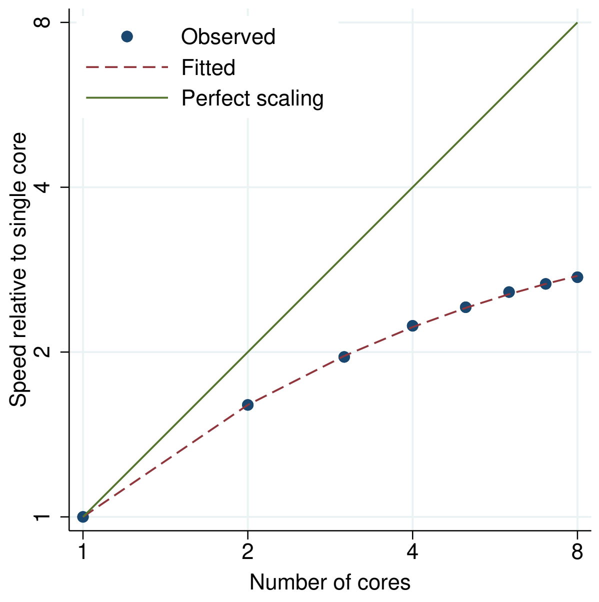


Figure 297. `mi impute mono regress` performance plot.

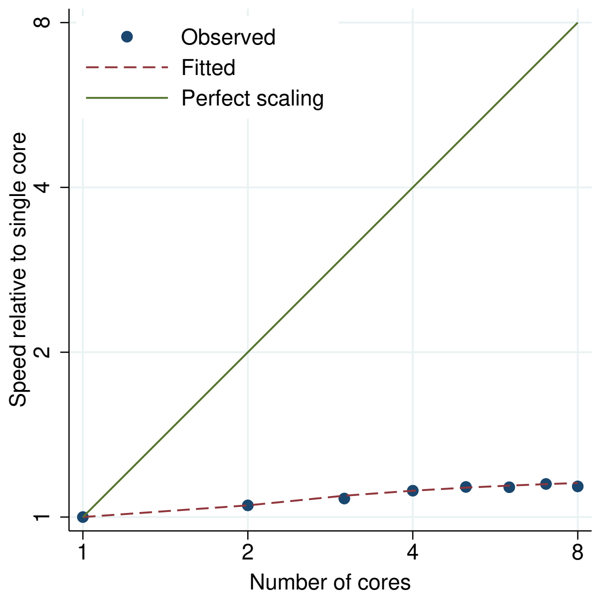


Figure 298. `mi impute mvn` performance plot.

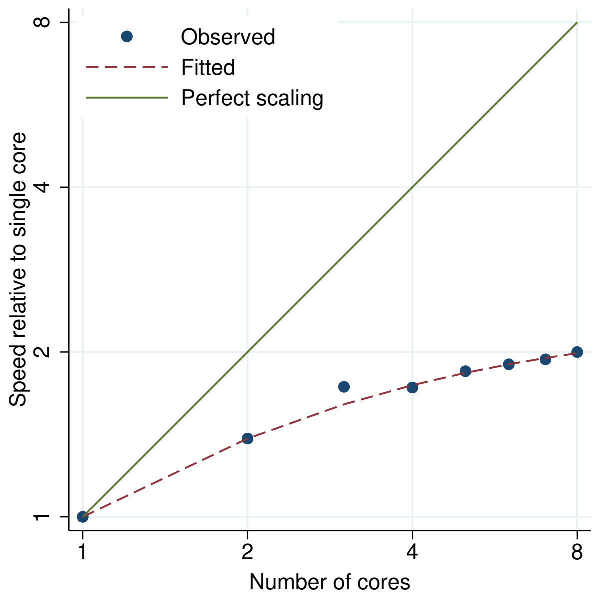


Figure 299. `mi impute ologit` performance plot.

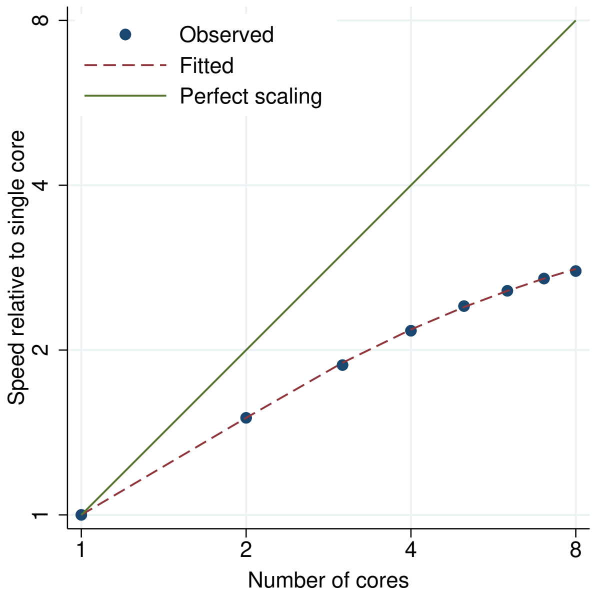


Figure 300. `mi impute pmm` performance plot.

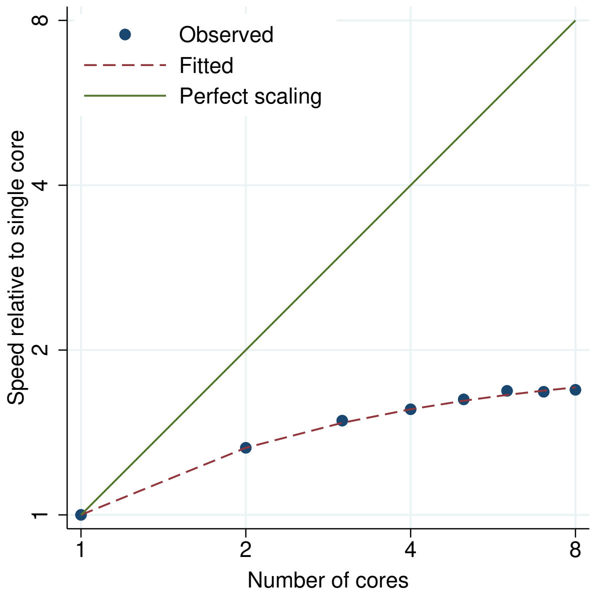


Figure 301. `mi impute regress` performance plot.

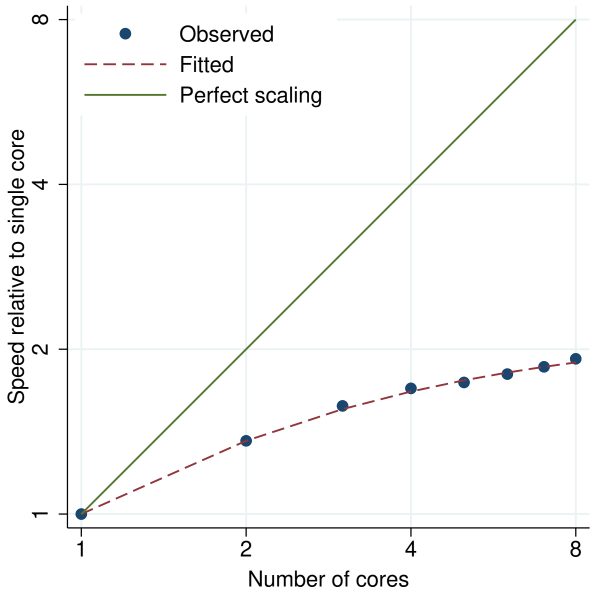


Figure 302. `misstable nested` performance plot.

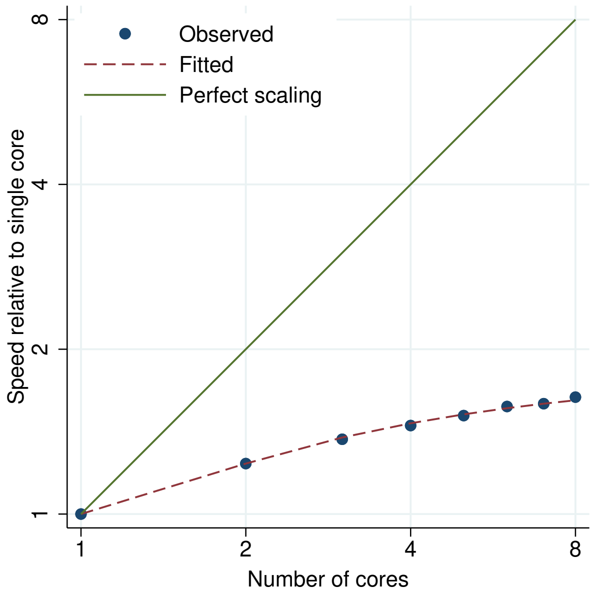


Figure 303. `misstable patterns` performance plot.

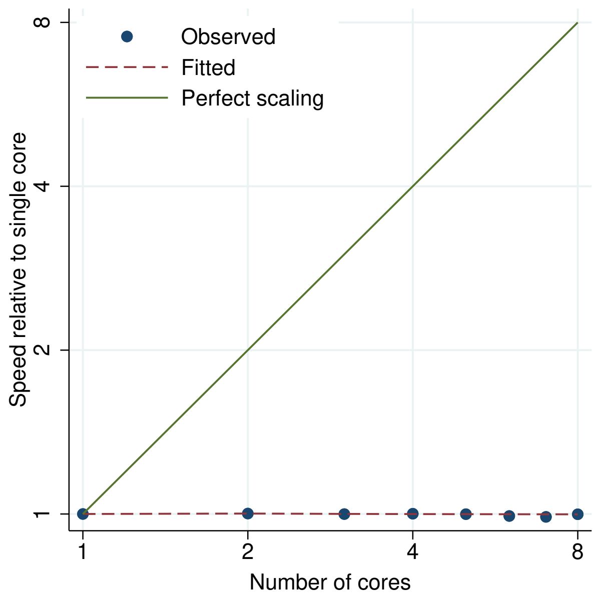


Figure 304. `misstable summarize` performance plot.

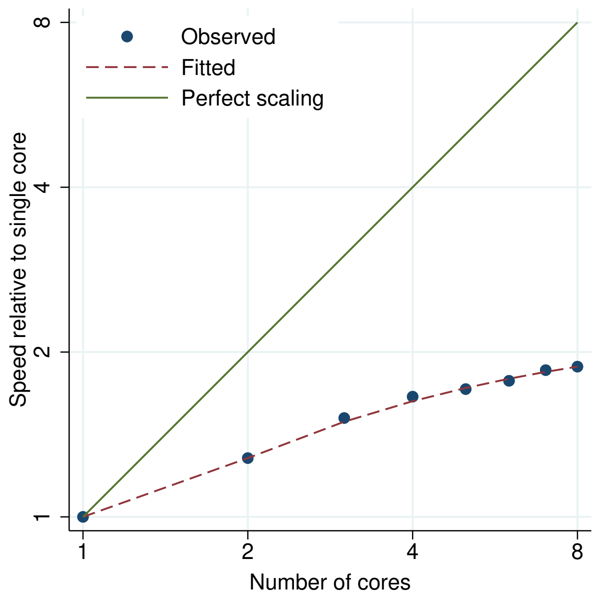


Figure 305. `misstable tree` performance plot.

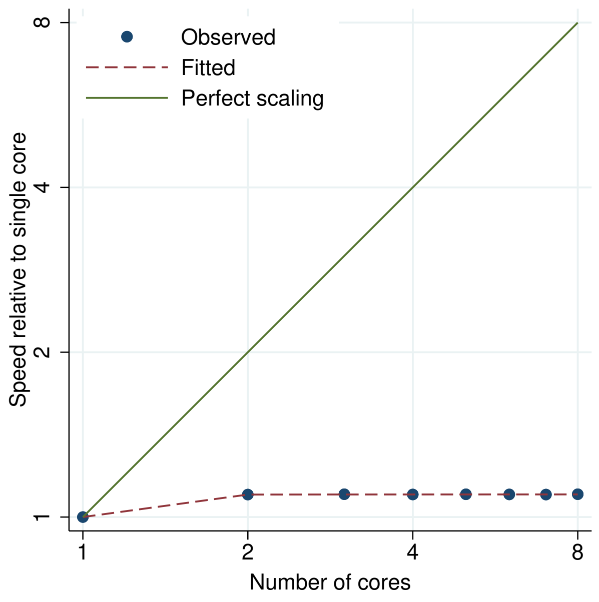


Figure 306. `mixed` performance plot.

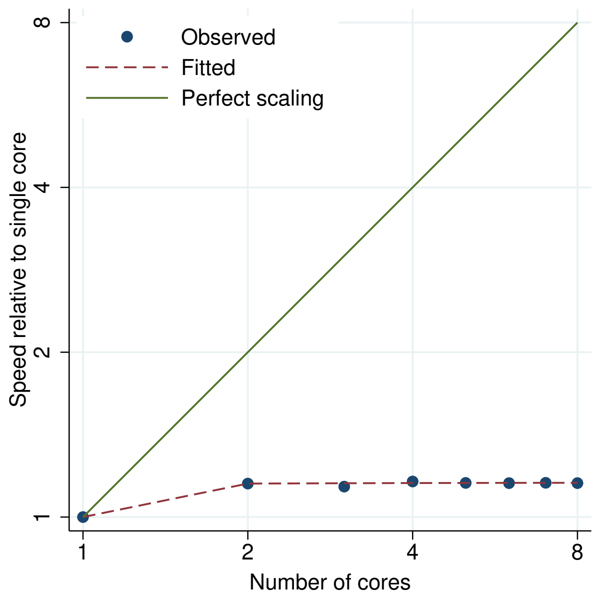


Figure 307. `mixed` (crossed effects) performance plot.

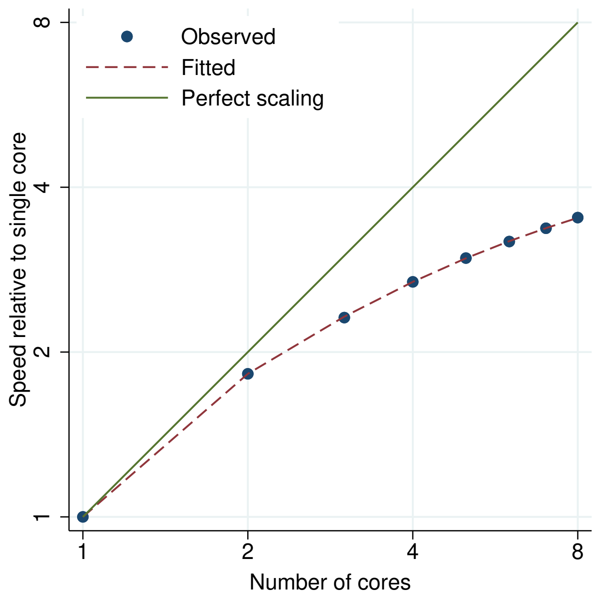


Figure 308. `mkspline` performance plot.

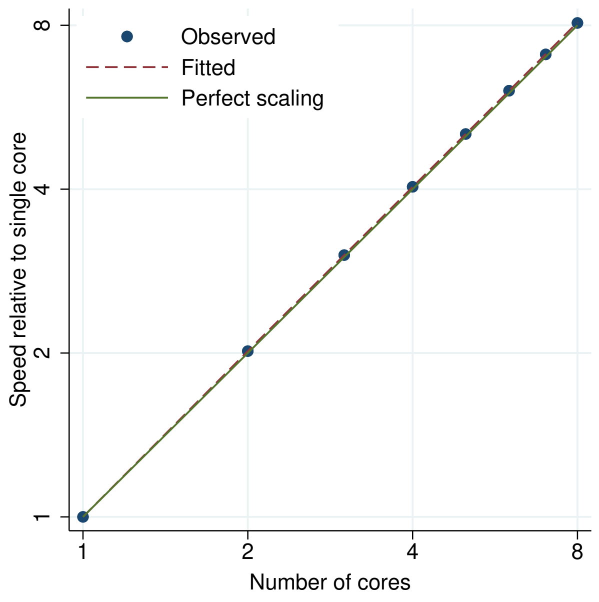


Figure 309. `mleval` performance plot.

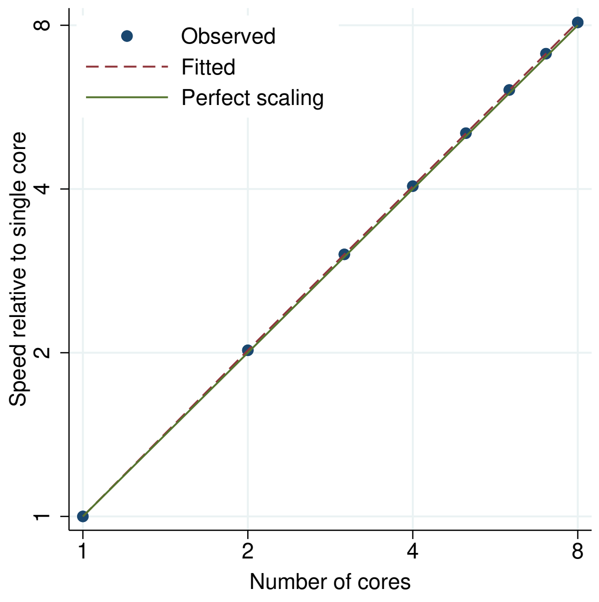


Figure 310. `mleval, nocons` performance plot.

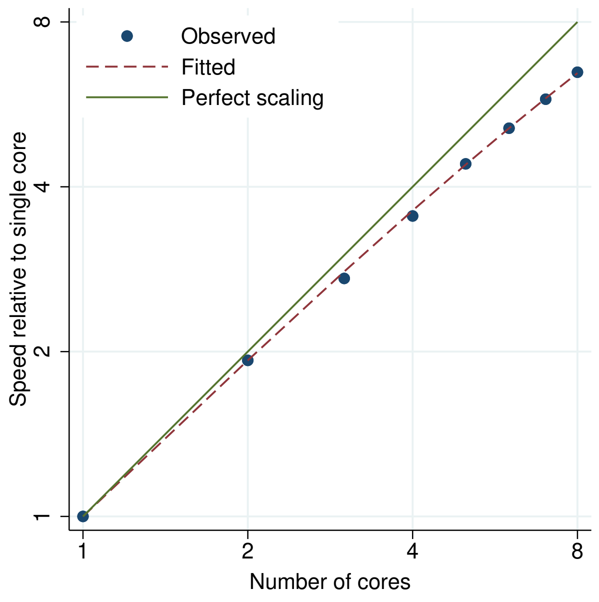


Figure 311. `mlmatbysum` performance plot.

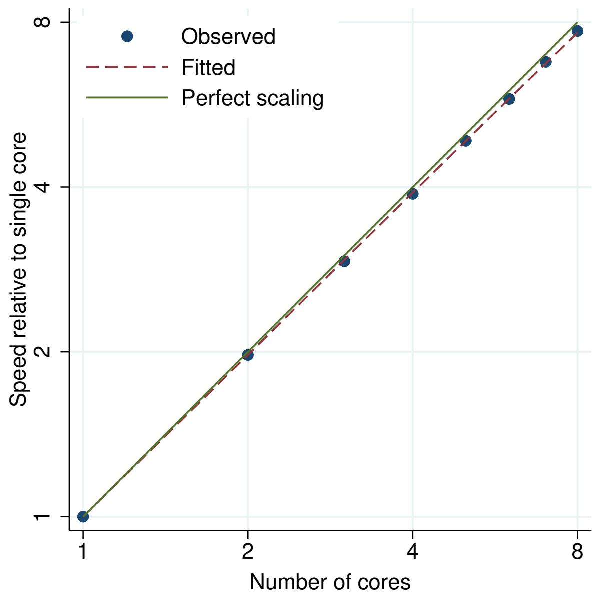


Figure 312. `mlmatsum` performance plot.

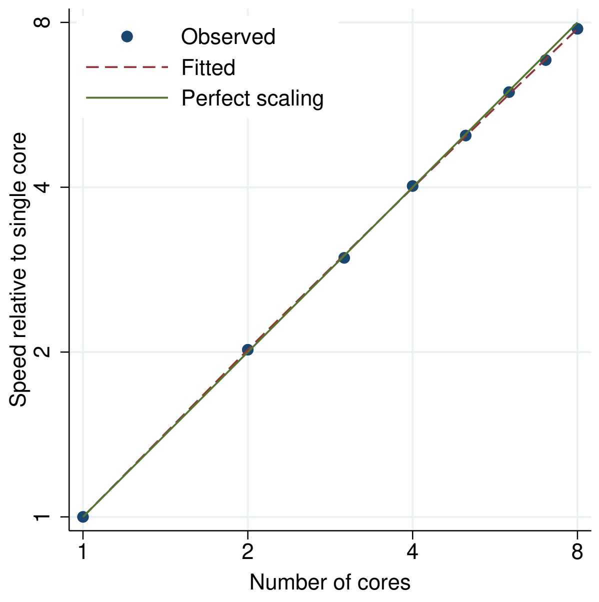


Figure 313. `mlogit` performance plot.

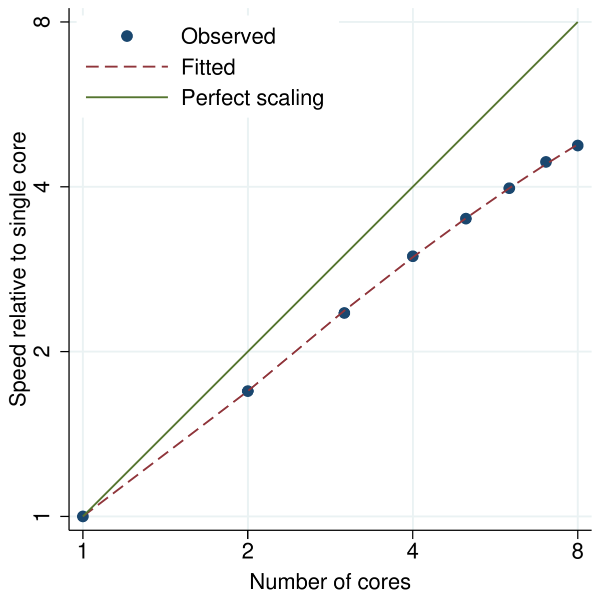


Figure 314. `mlsum` performance plot.

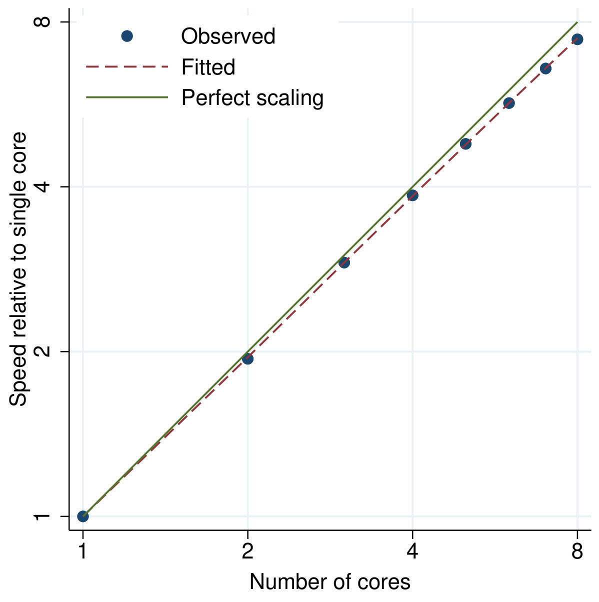


Figure 315. `mlvecsum` performance plot.

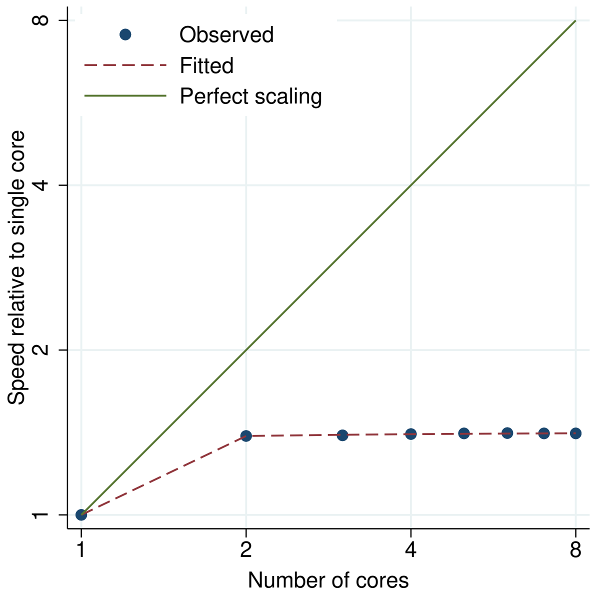


Figure 316. `mprobit` performance plot.

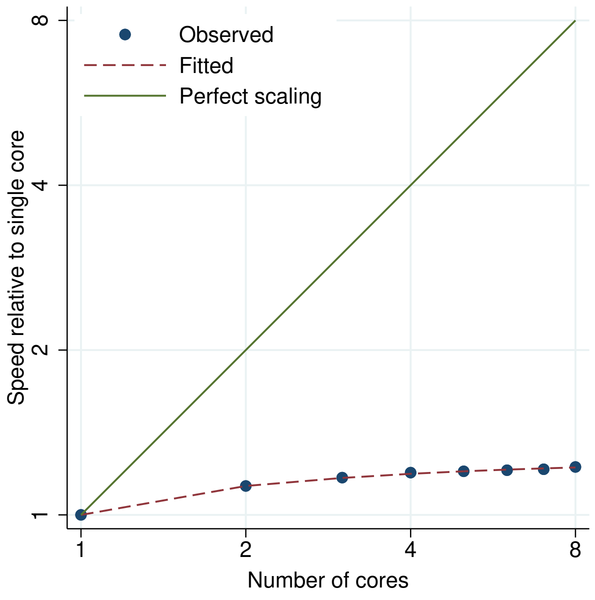


Figure 317. `mswitch ar` performance plot.

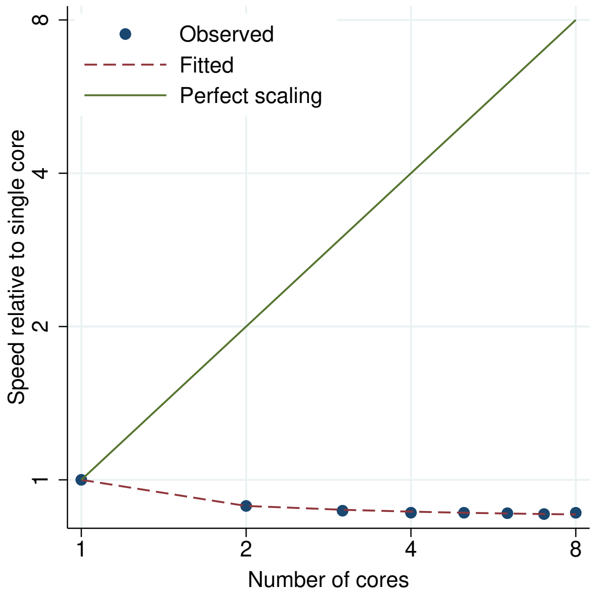


Figure 318. `mswitch dr` performance plot.

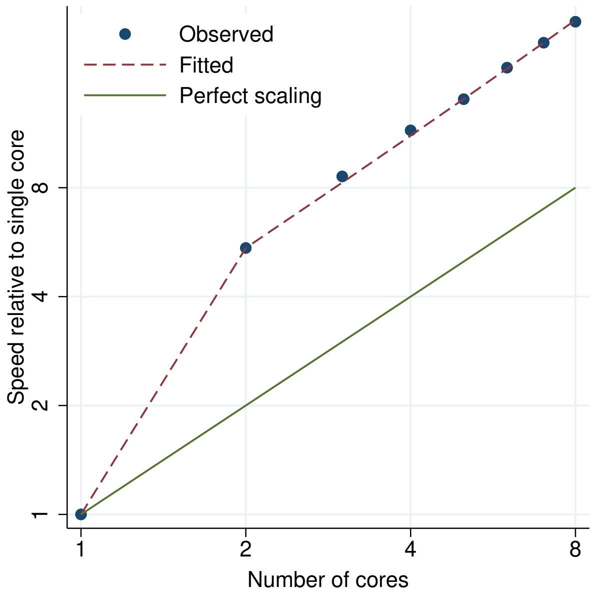


Figure 319. `mvdecode` performance plot.

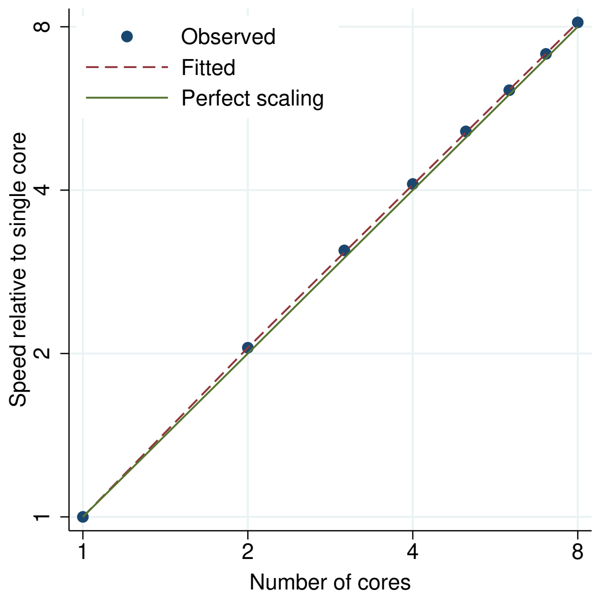


Figure 320. `mvencode` performance plot.

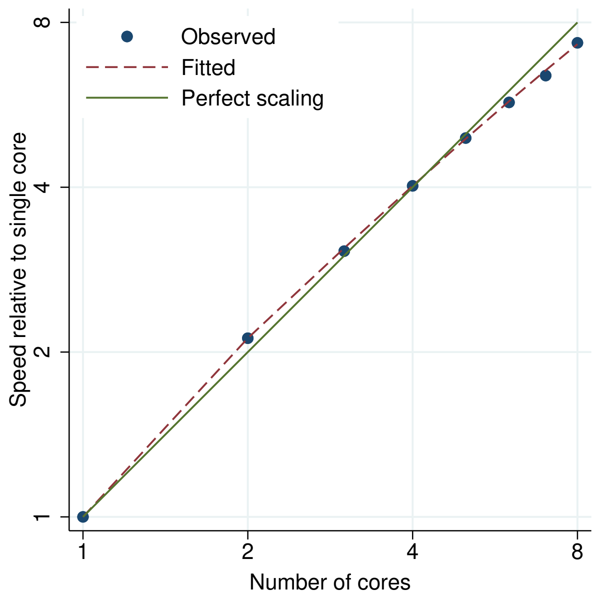


Figure 321. `mvreg` performance plot.

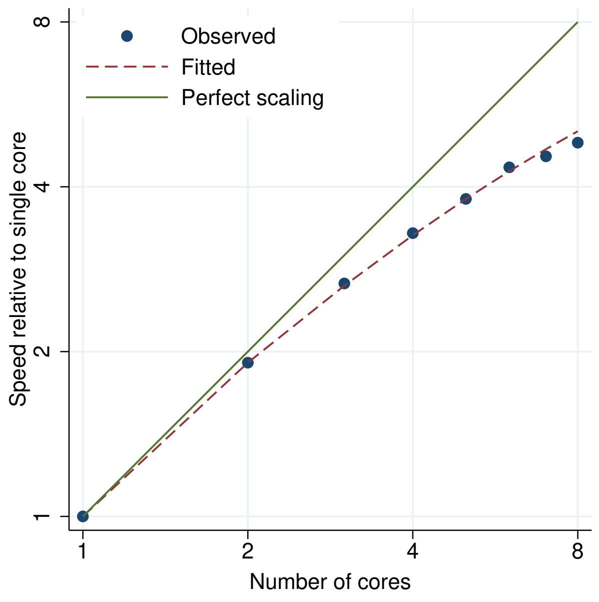


Figure 322. `mvtest correlations` performance plot.

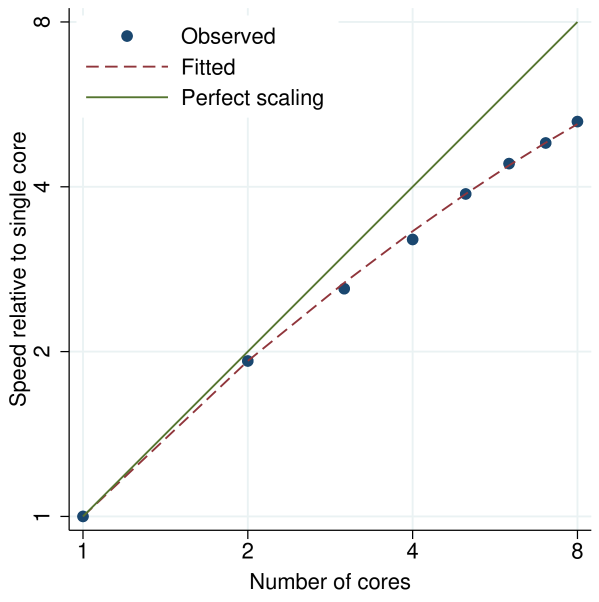


Figure 323. `mvtest covariances` performance plot.

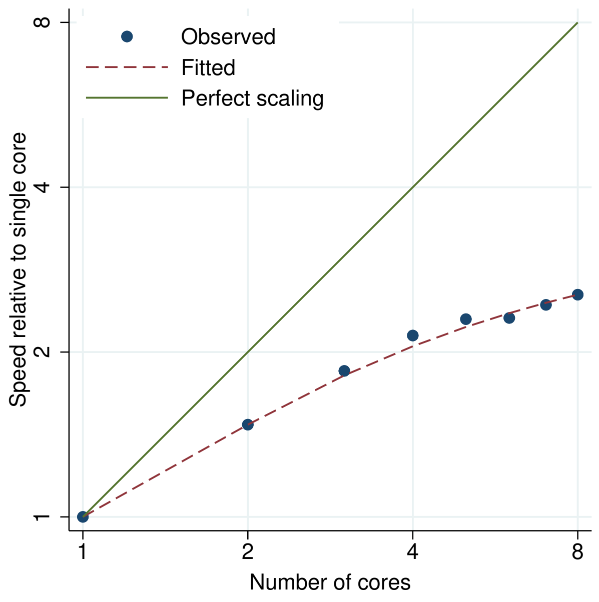


Figure 324. `mvtest means, heterogeneous` performance plot.

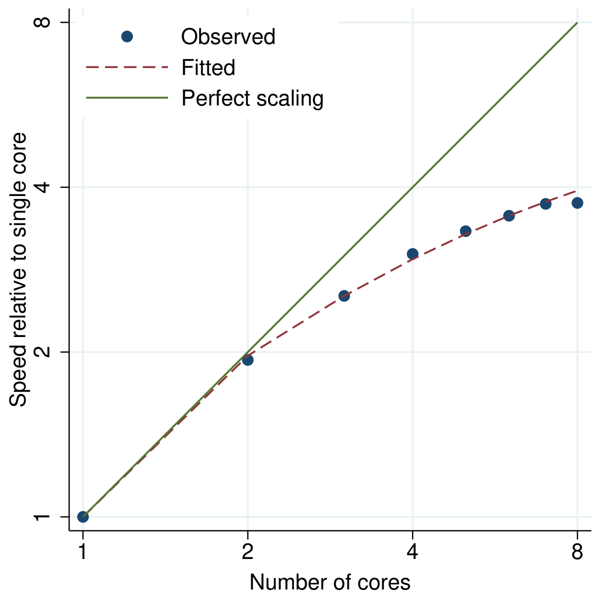


Figure 325. `mvtest means, homogeneous` performance plot.

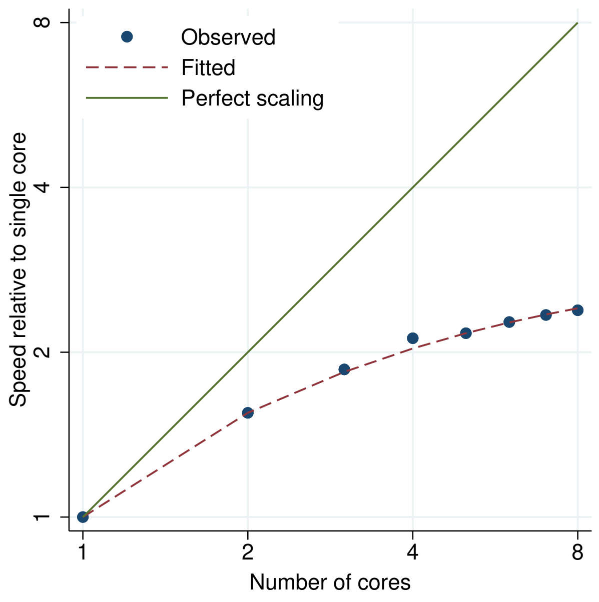


Figure 326. `mvtest means, lr` performance plot.

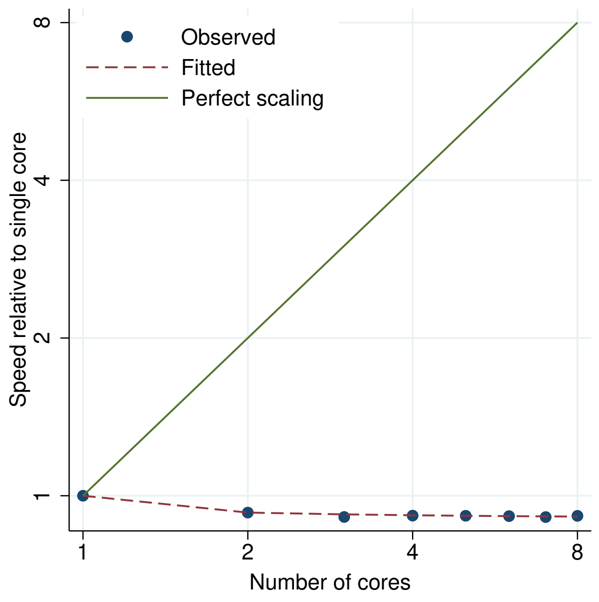


Figure 327. `mvtest normality` performance plot.

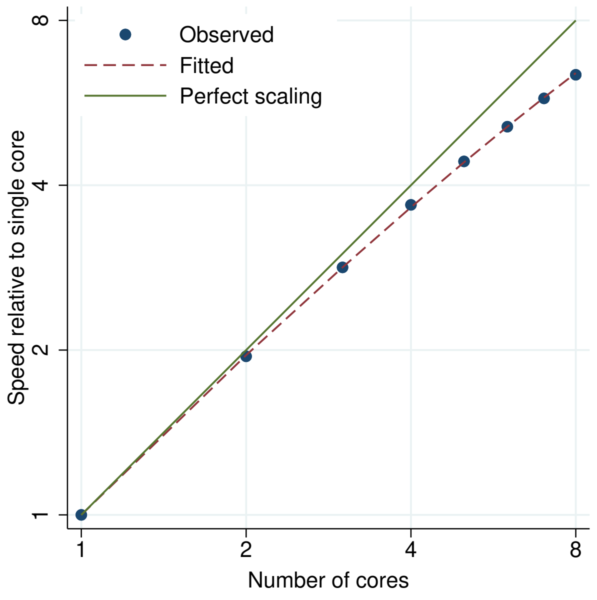


Figure 328. `nbreg` performance plot.

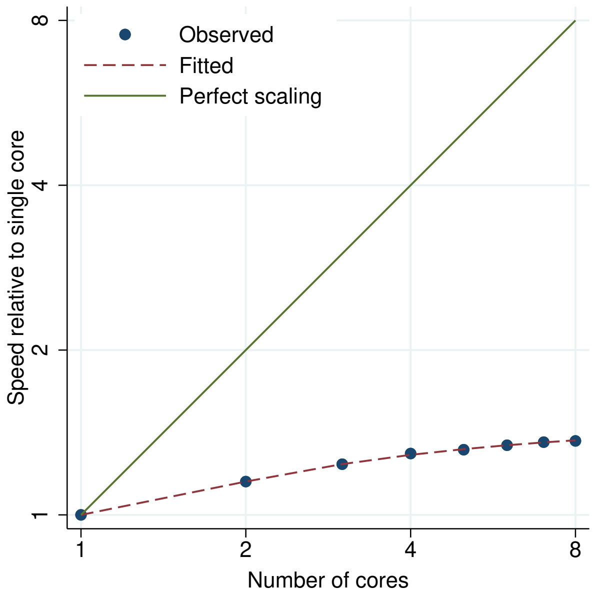


Figure 329. `newey` performance plot.

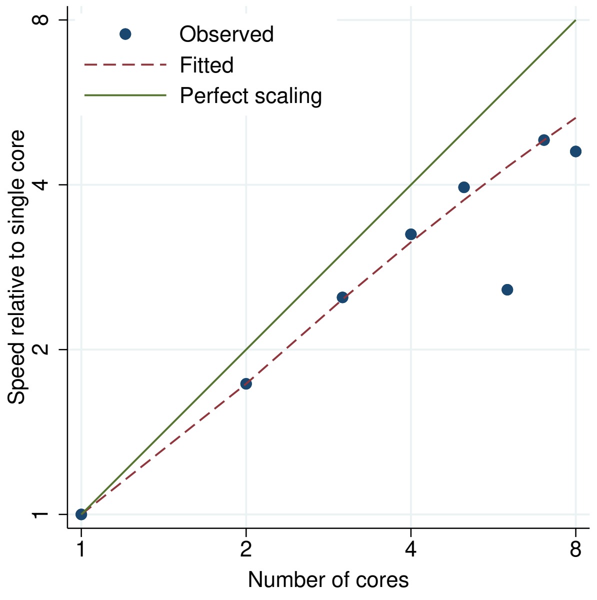


Figure 330. `nl` performance plot.

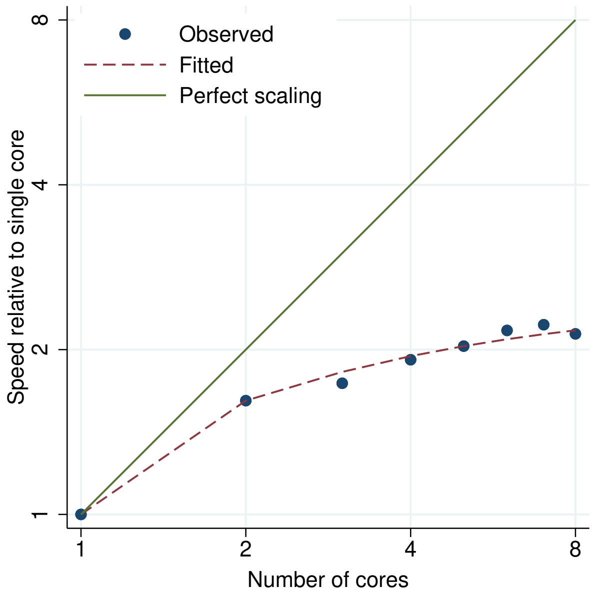


Figure 331. `nlogit` performance plot.

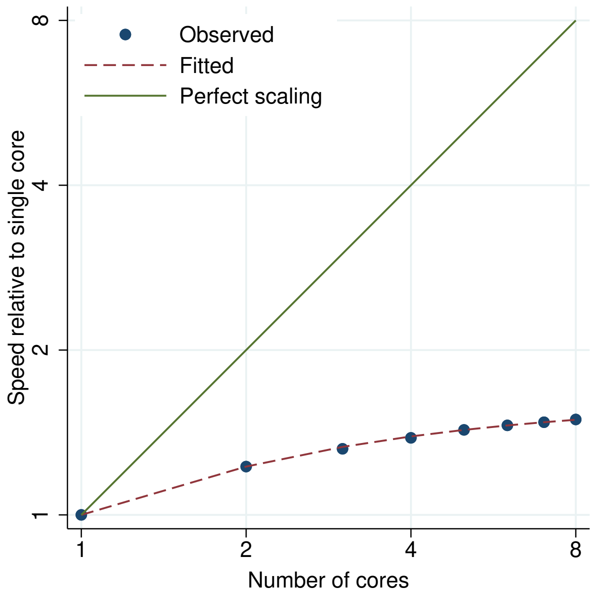


Figure 332. `nlsur` performance plot.

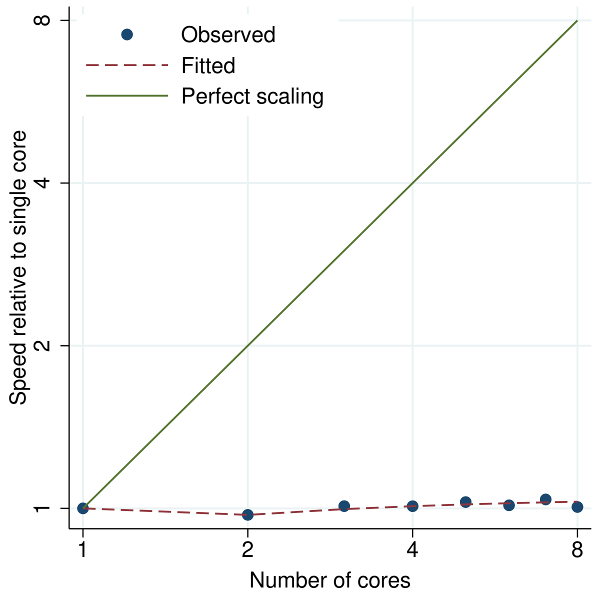


Figure 333. `npregress kernel` performance plot.

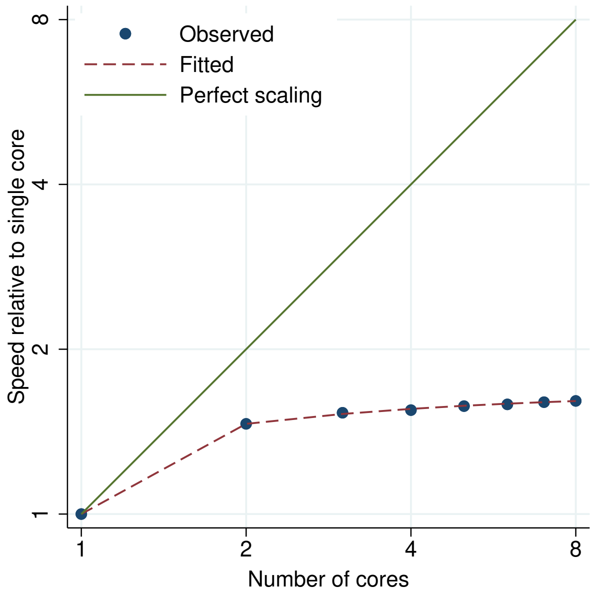


Figure 334. `nptrend` performance plot.

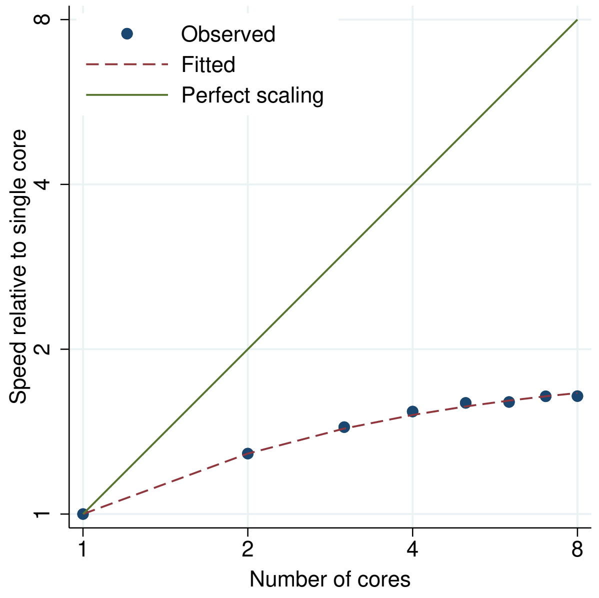


Figure 335. `nptrend_carmitage` performance plot.

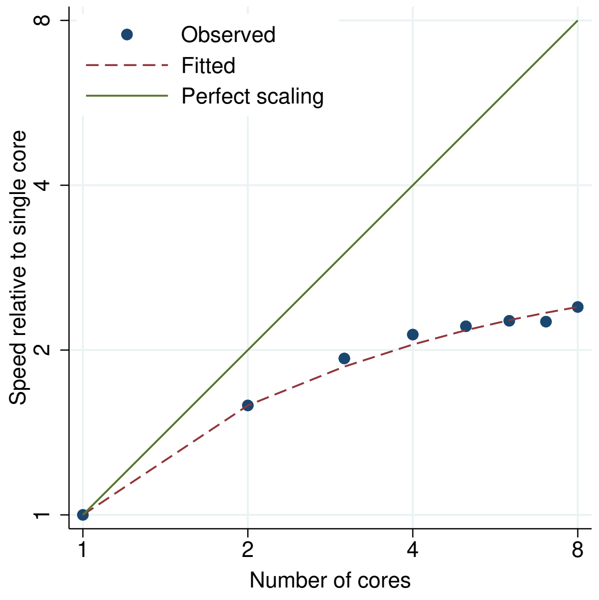


Figure 336. `nptrend_jterpstra` performance plot.

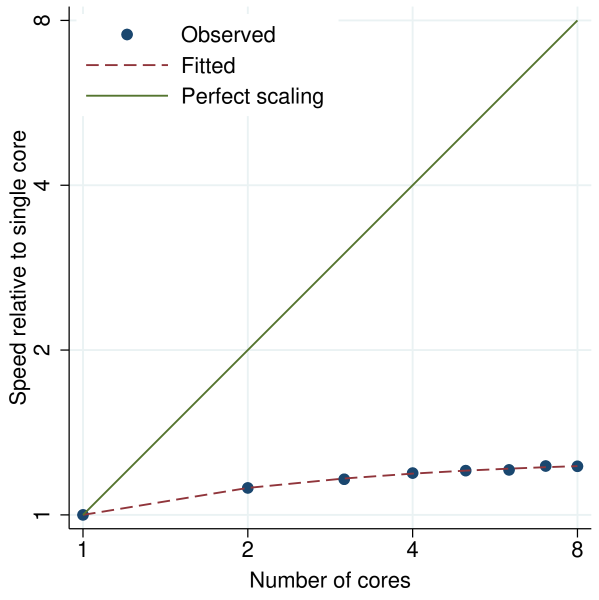


Figure 337. `nptrend_linear` performance plot.

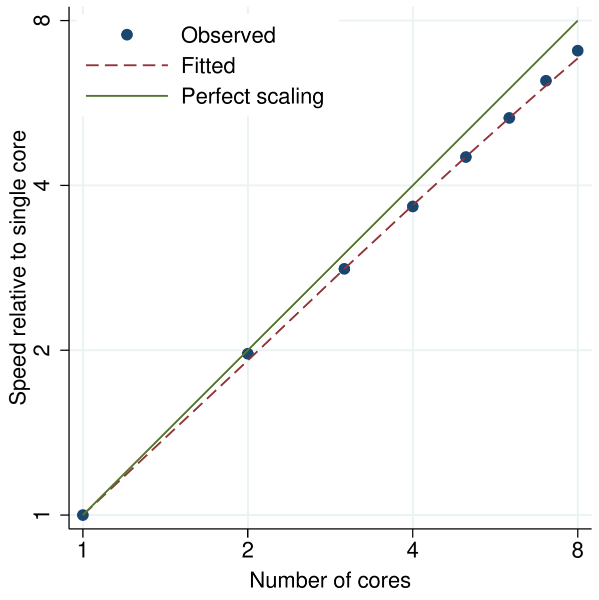


Figure 338. `ologit` performance plot.

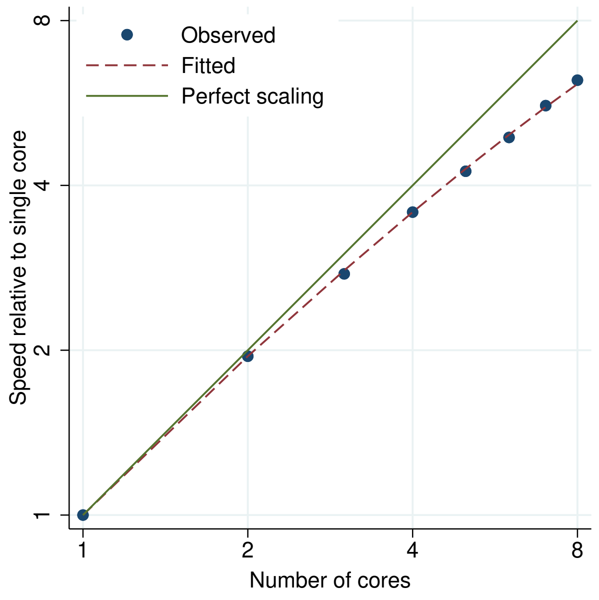


Figure 339. `ologit, vce(cluster)` performance plot.

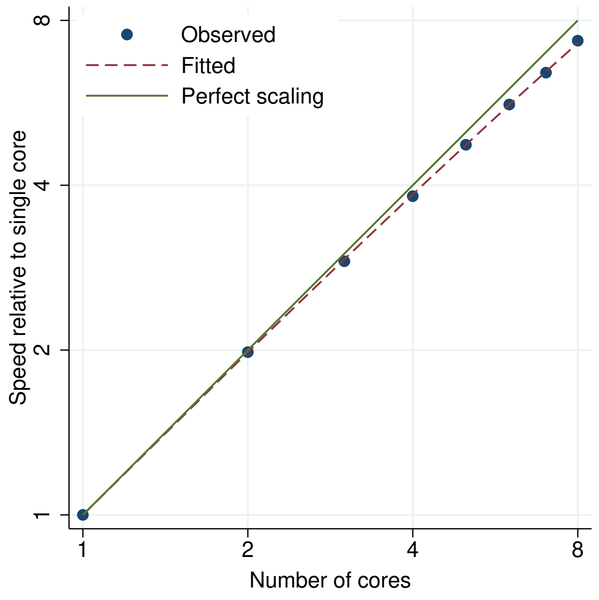


Figure 340. `ologit, vce(robust)` performance plot.

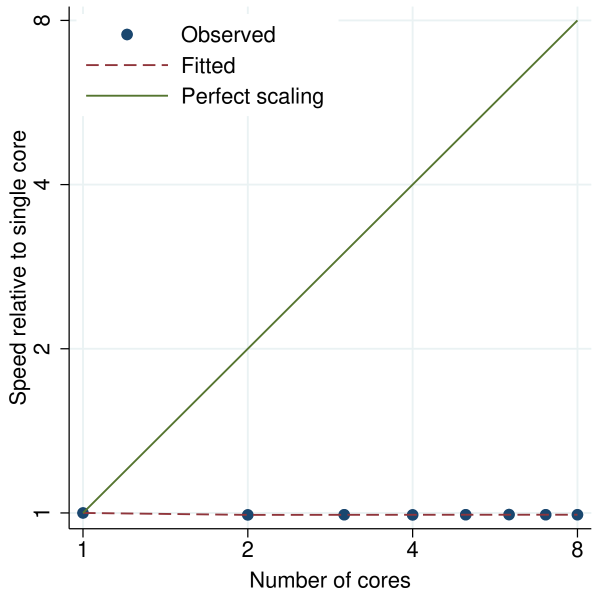


Figure 341. `oneway` performance plot.

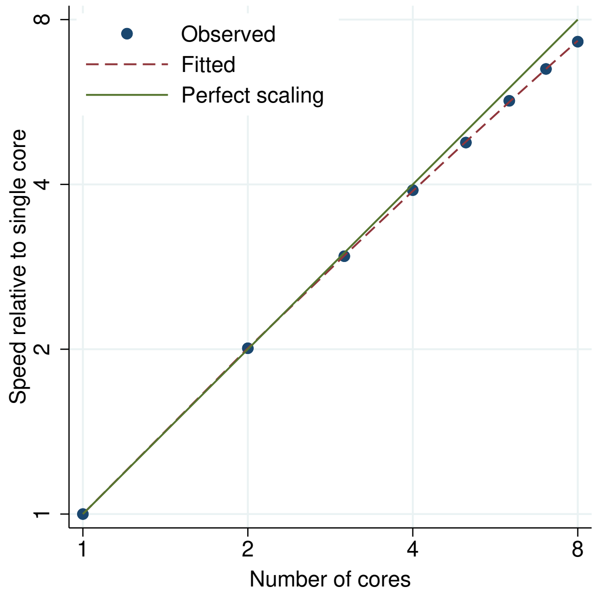


Figure 342. `oprobit` performance plot.

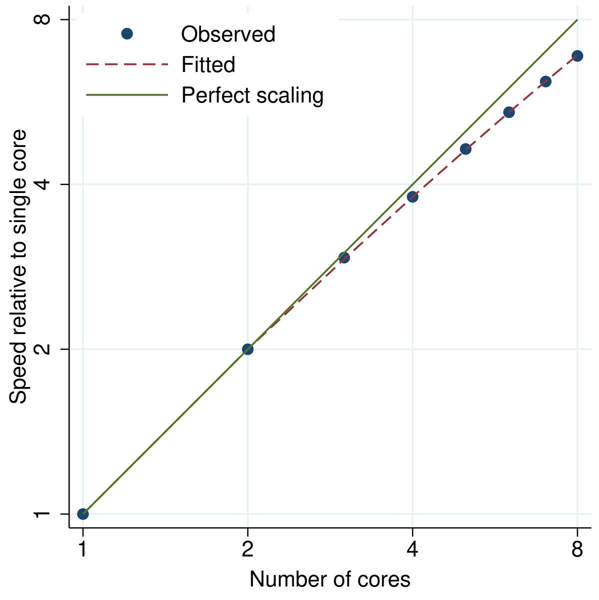


Figure 343. `oprobit, vce(cluster)` performance plot.

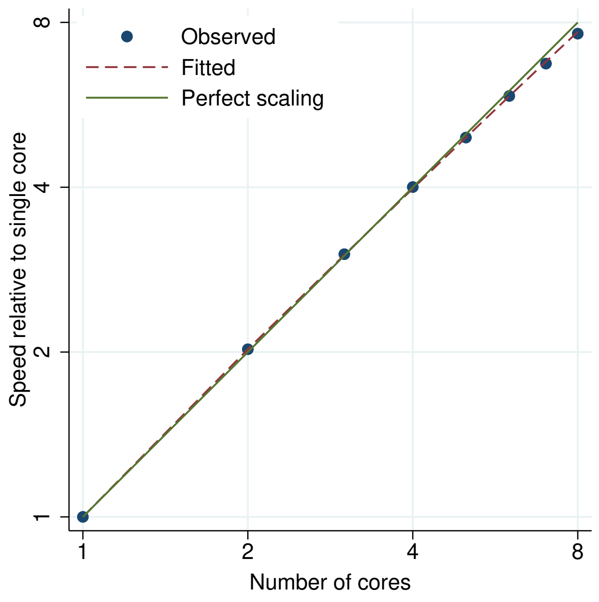


Figure 344. `oprobit, vce(robust)` performance plot.

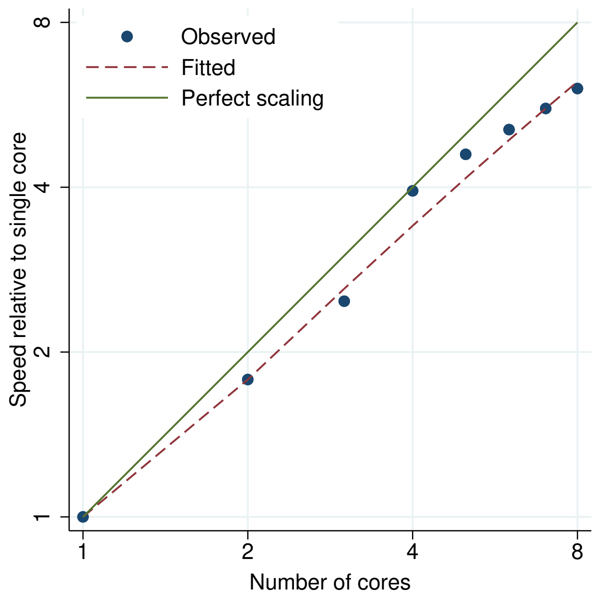


Figure 345. `orthog` performance plot.

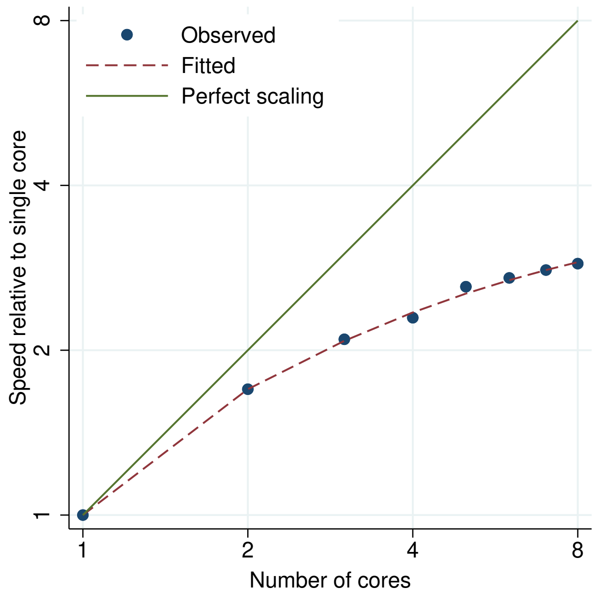


Figure 346. `pca` performance plot.

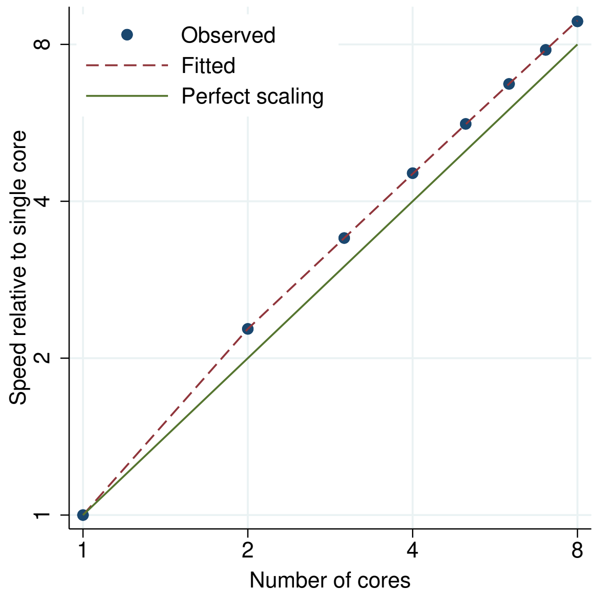


Figure 347. `pcorr` performance plot.

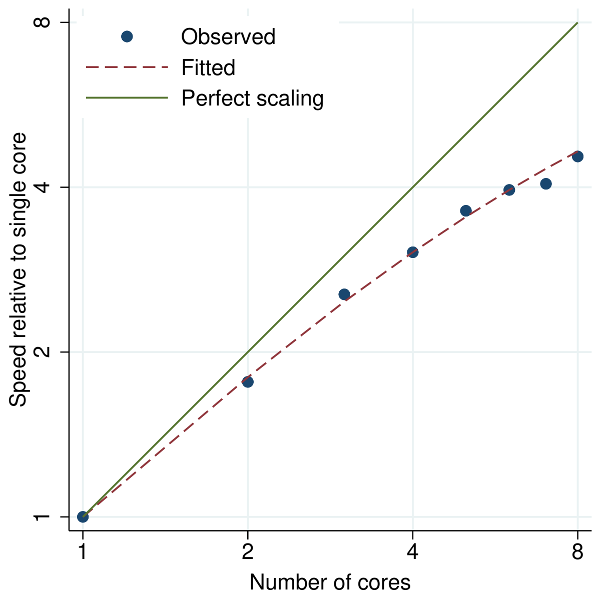


Figure 348. `pctile` performance plot.

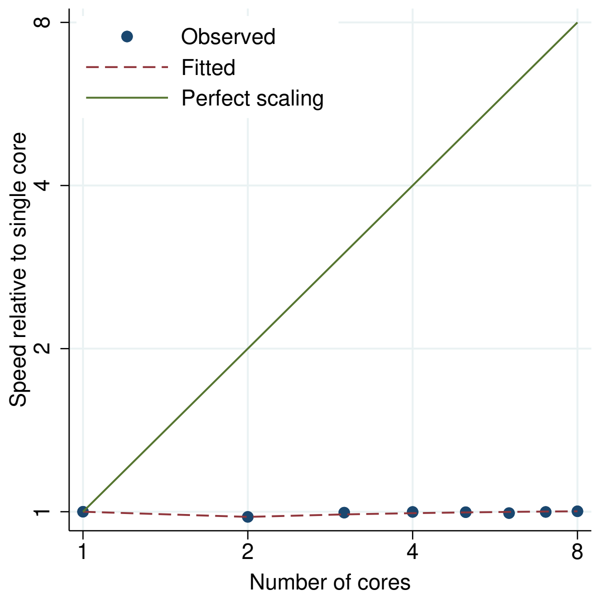


Figure 349. `pergram` performance plot.

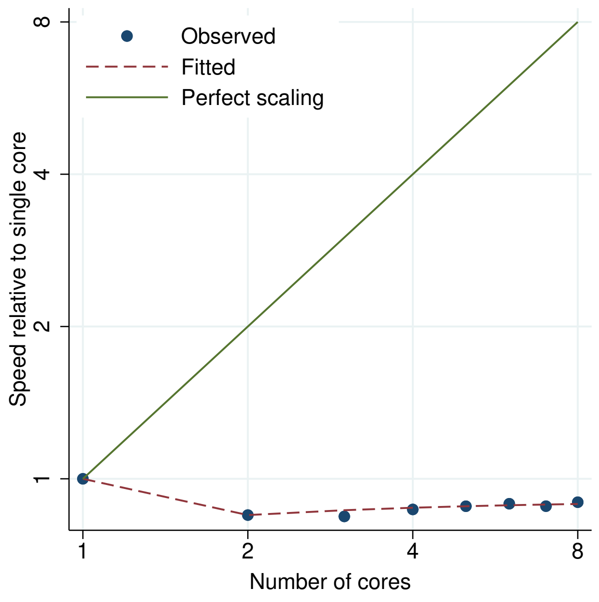


Figure 350. `pkcollapse` performance plot.

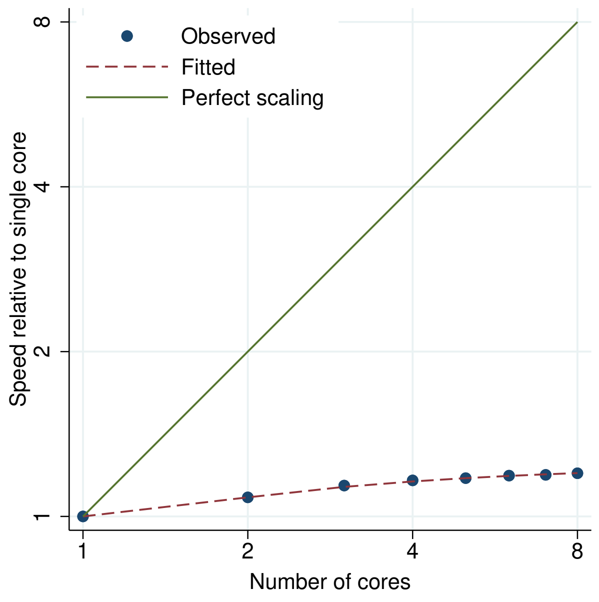


Figure 351. `pkexamine` performance plot.

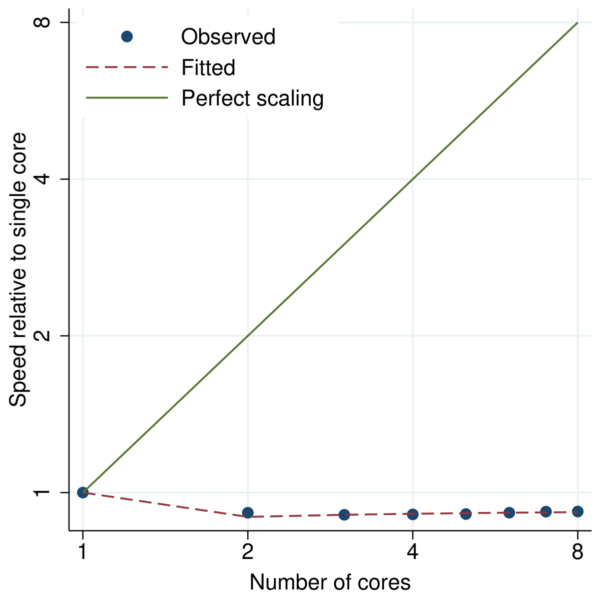


Figure 352. `pksumm` performance plot.

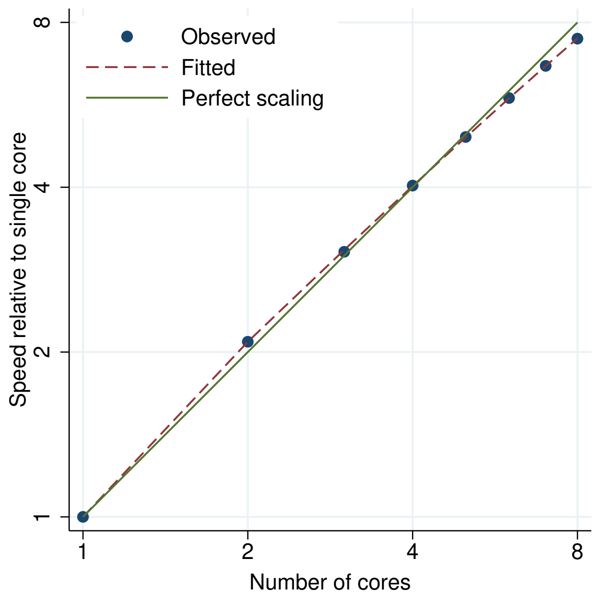


Figure 353. `poisson` performance plot.

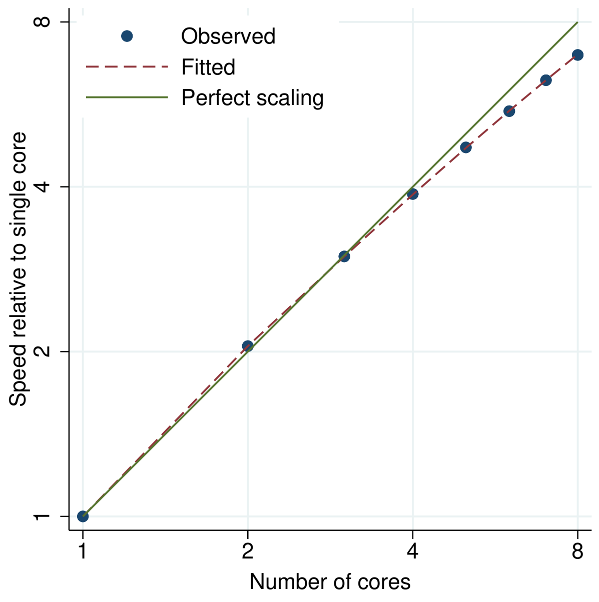


Figure 354. `poisson, vce(cluster)` performance plot.

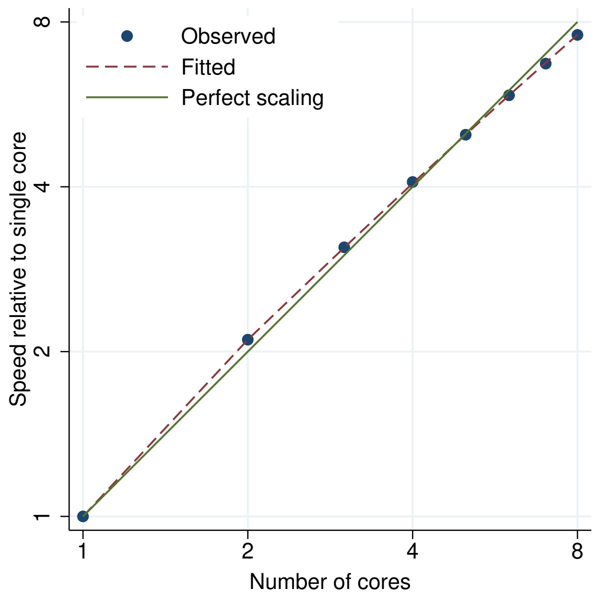


Figure 355. `poisson, exposure()` performance plot.

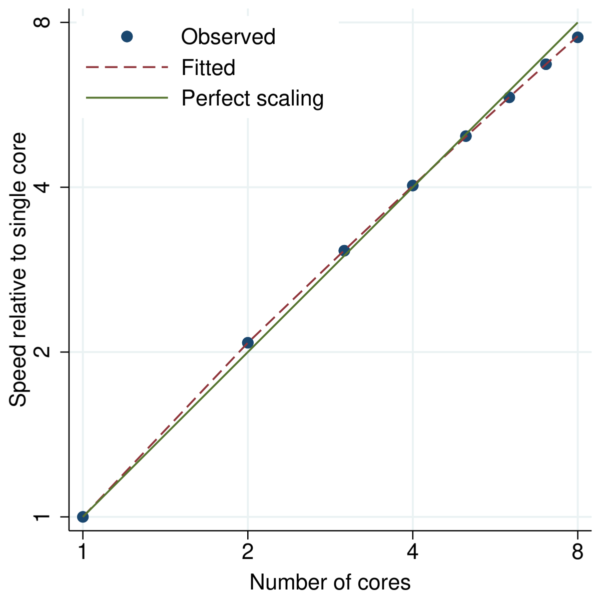


Figure 356. `poisson, vce(robust)` performance plot.

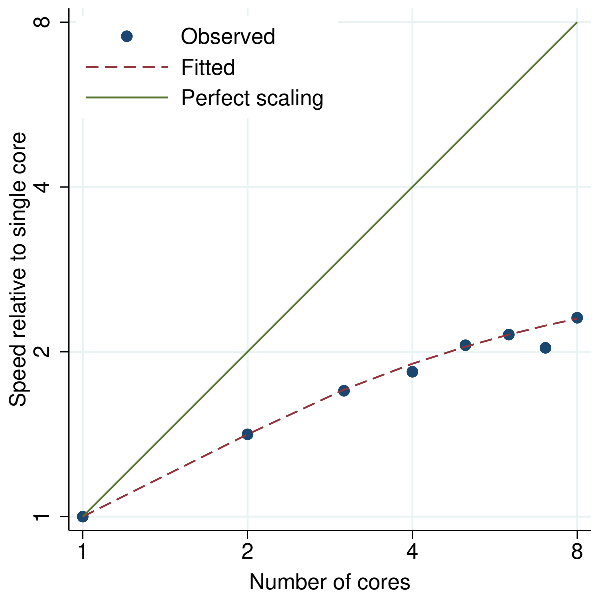


Figure 357. `pologit` performance plot.

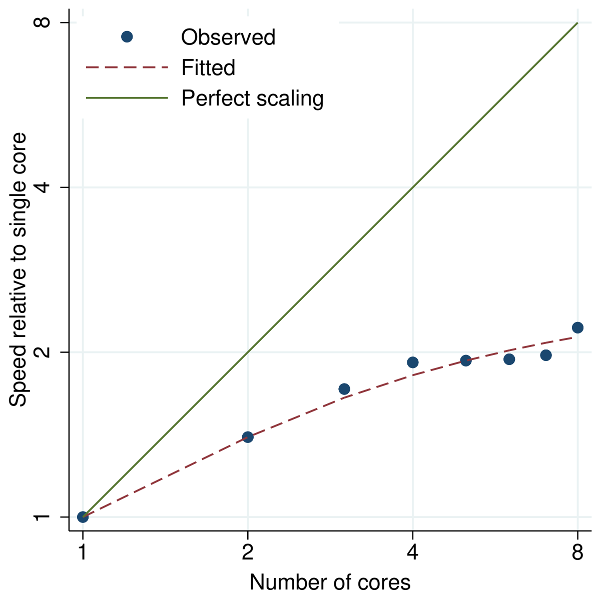


Figure 358. `popoisson` performance plot.

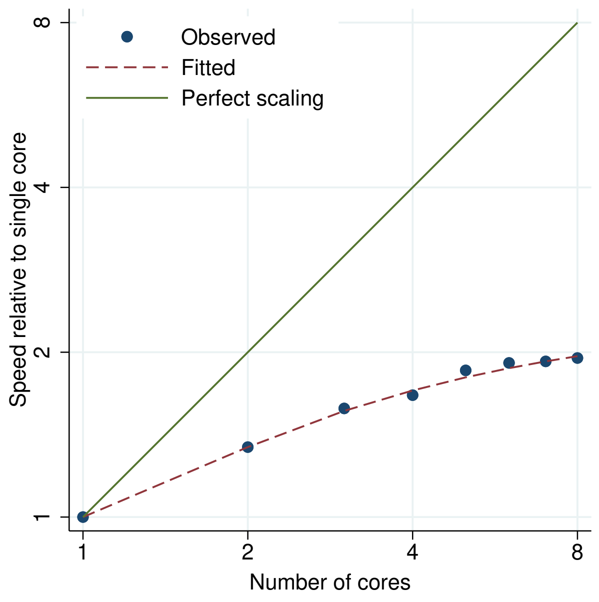


Figure 359. `poregress` performance plot.

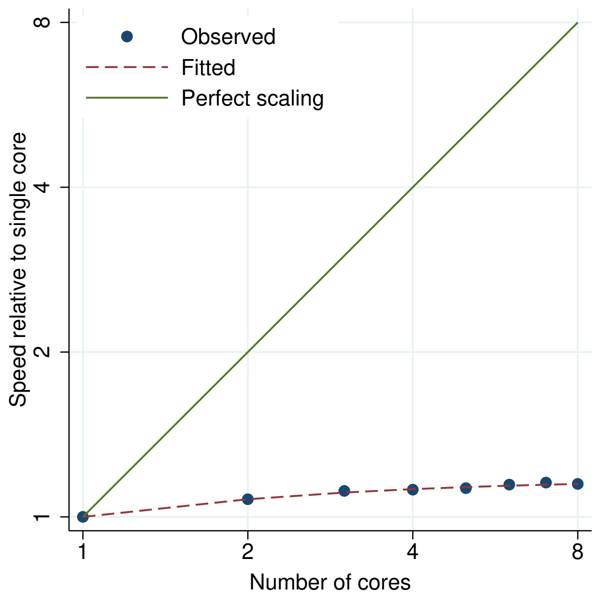


Figure 360. `pperron` performance plot.

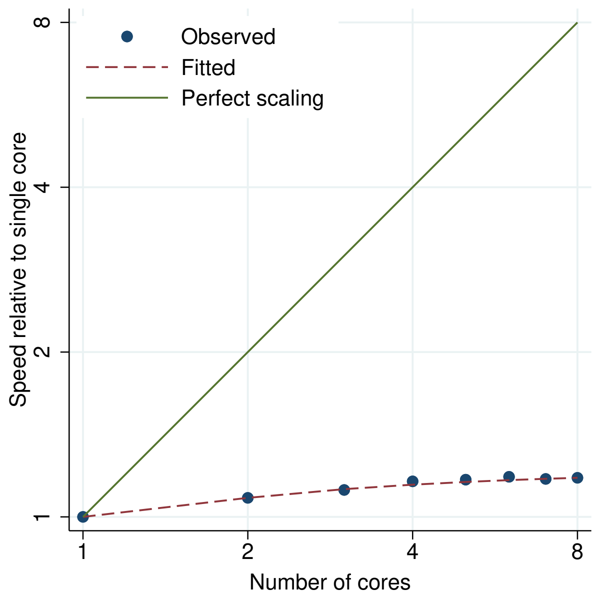


Figure 361. `prais` performance plot.

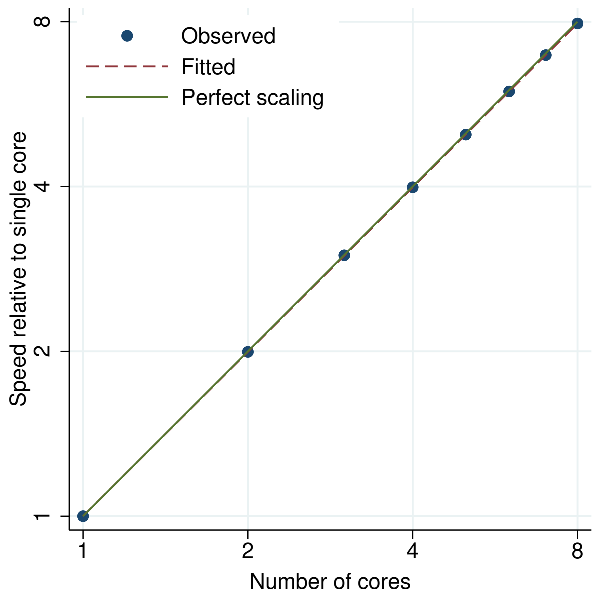


Figure 362. `predict, cooksd` performance plot.

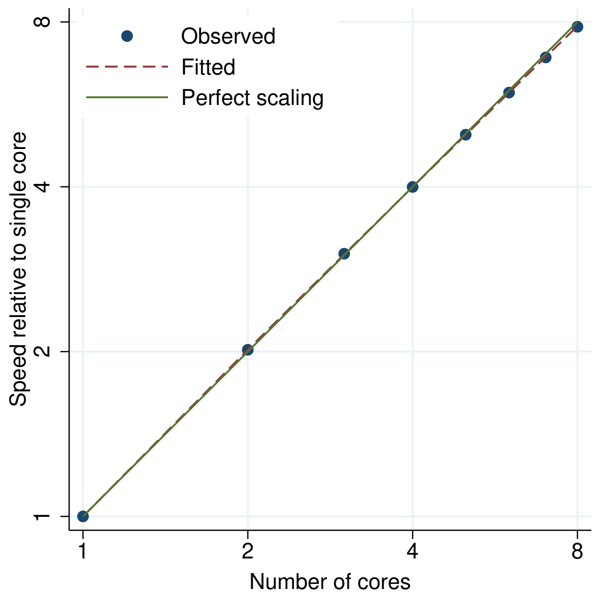


Figure 363. `predict, covratio` performance plot.

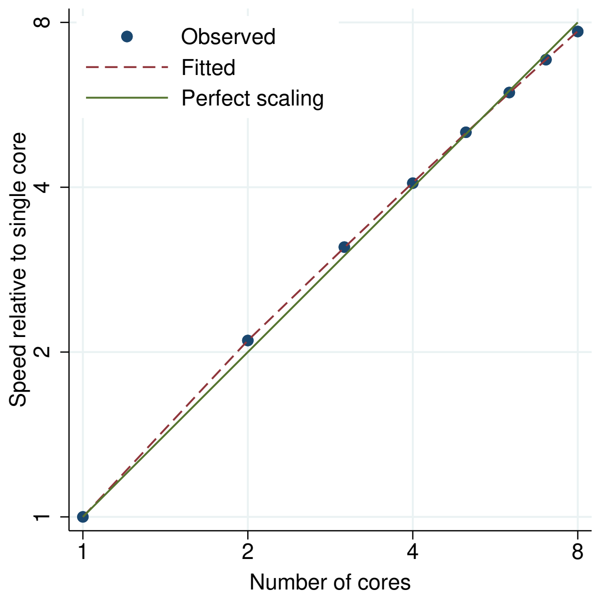


Figure 364. `predict, dfbeta` performance plot.

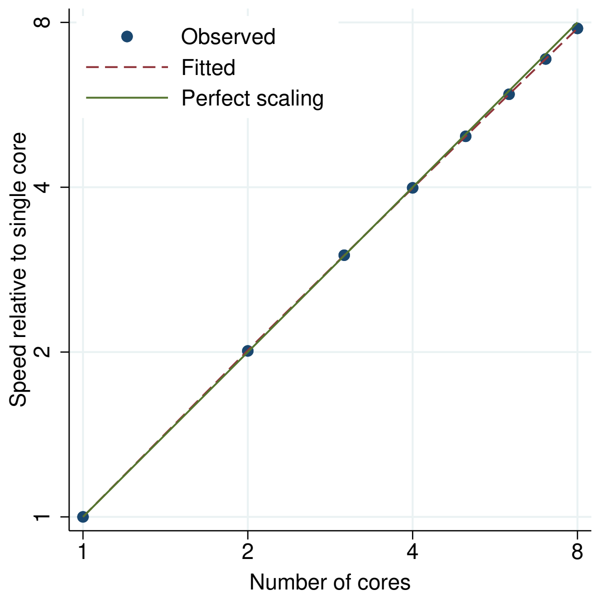


Figure 365. `predict, dfits` performance plot.

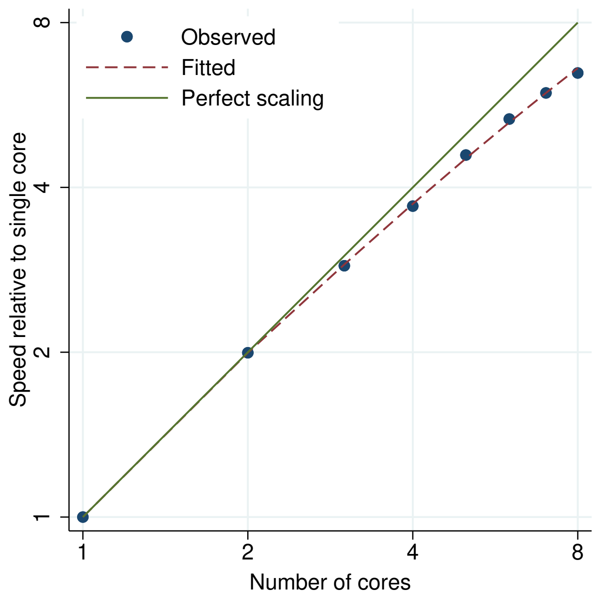


Figure 366. `predict, e` performance plot.

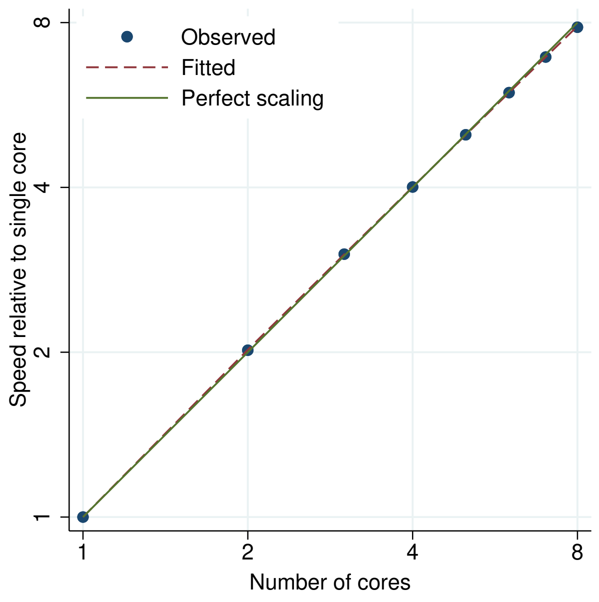


Figure 367. `predict, leverage` performance plot.

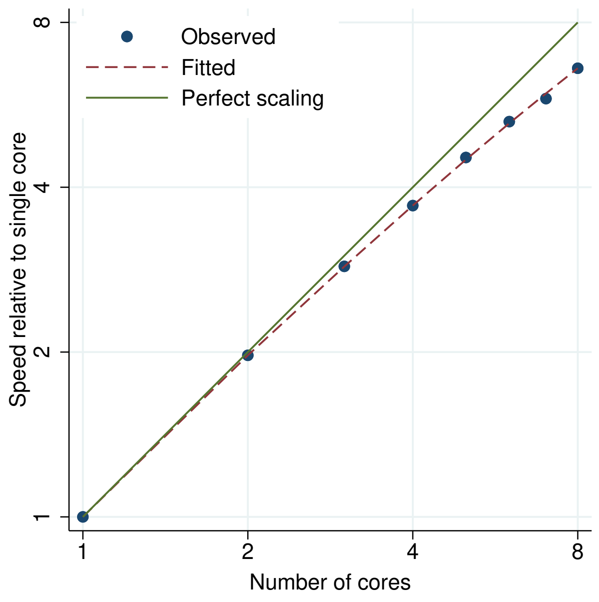


Figure 368. `predict, pr` performance plot.

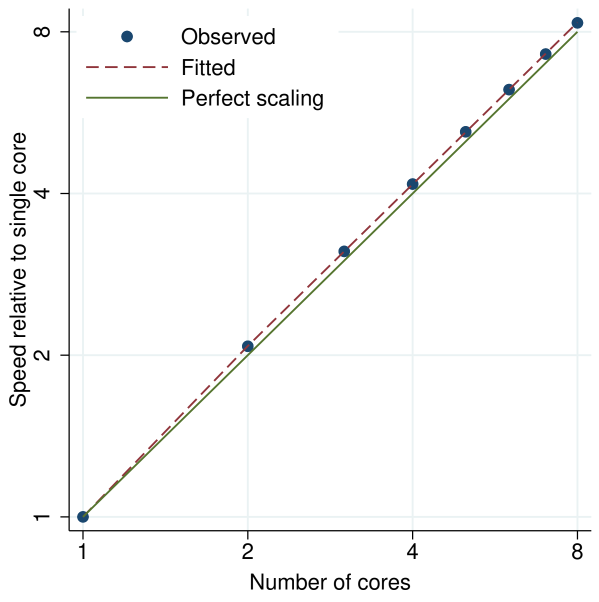


Figure 369. `predict, residuals` performance plot.

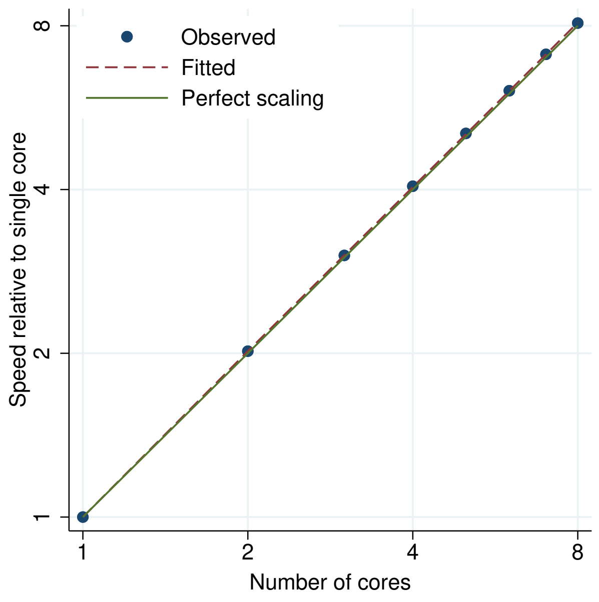


Figure 370. `predict, rstandard` performance plot.

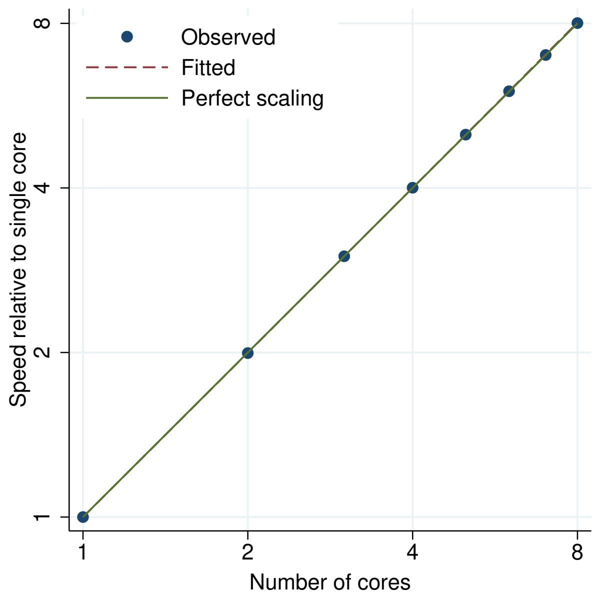


Figure 371. `predict, rstudent` performance plot.

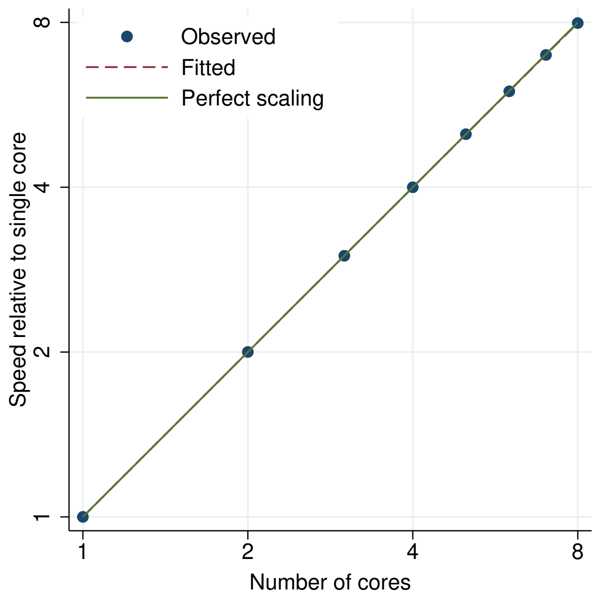


Figure 372. `predict, stdf` performance plot.

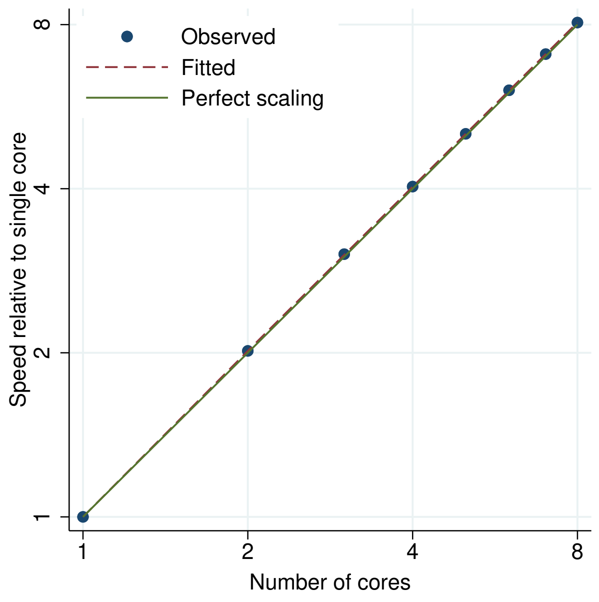


Figure 373. `predict, stdp` performance plot.

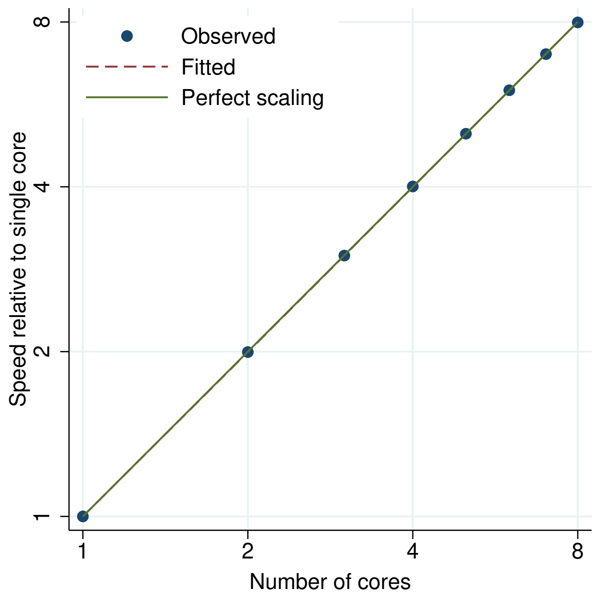


Figure 374. `predict, stdr` performance plot.

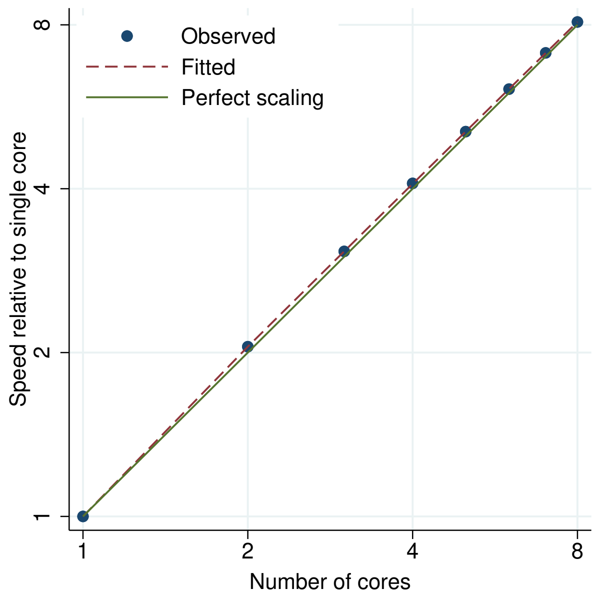


Figure 375. `predict, welsch` performance plot.

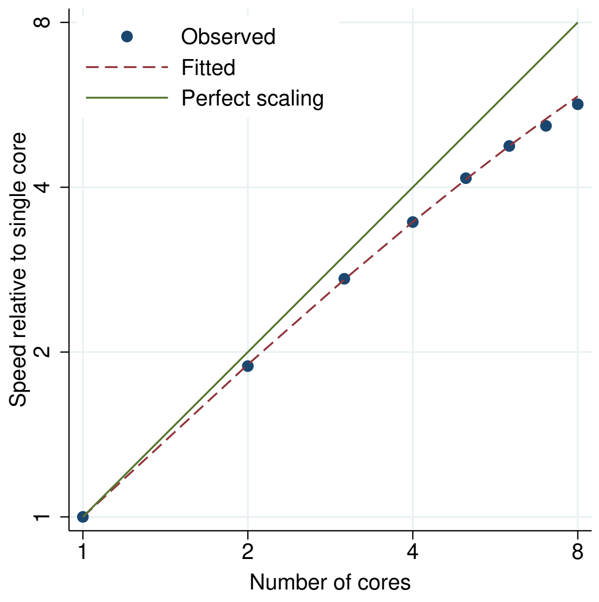


Figure 376. `predict, ystar` performance plot.

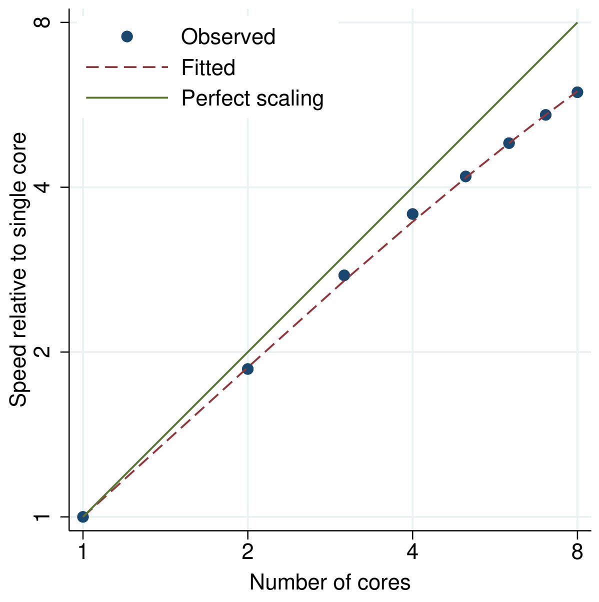


Figure 377. `predictnl` performance plot.

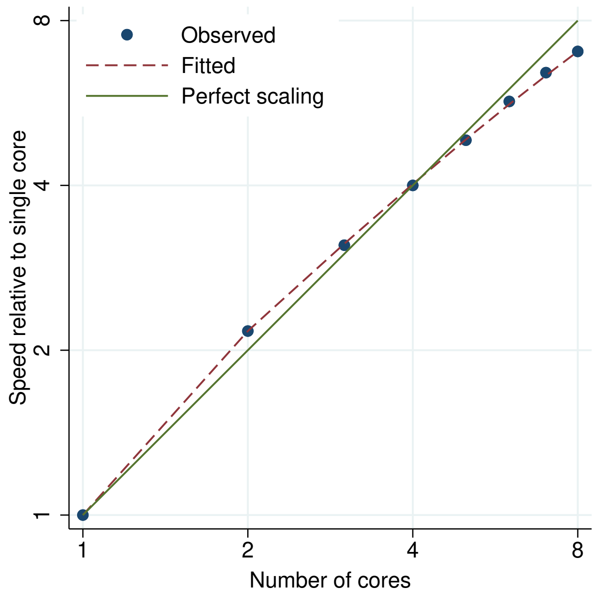


Figure 378. `probit` performance plot.

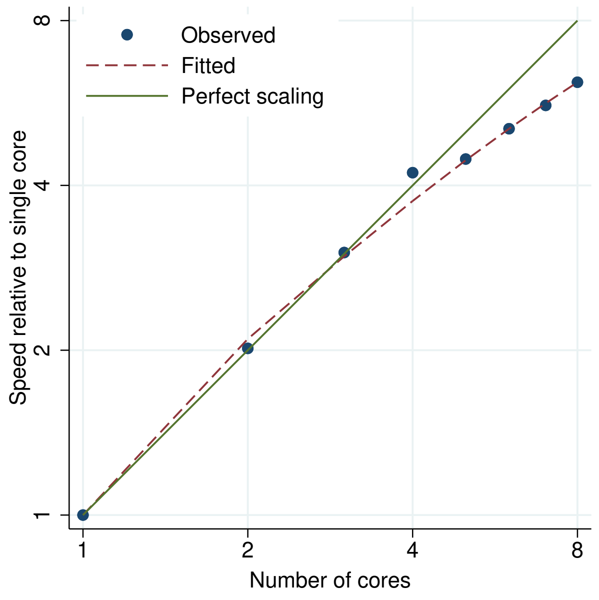


Figure 379. `procrustes` performance plot.

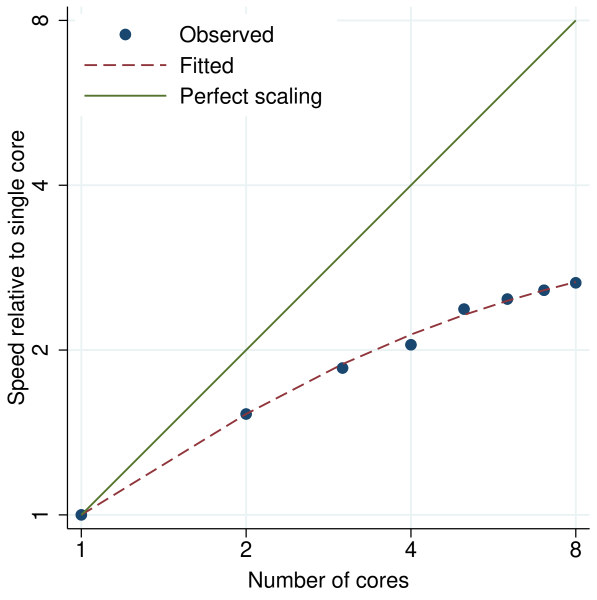


Figure 380. `proportion` performance plot.

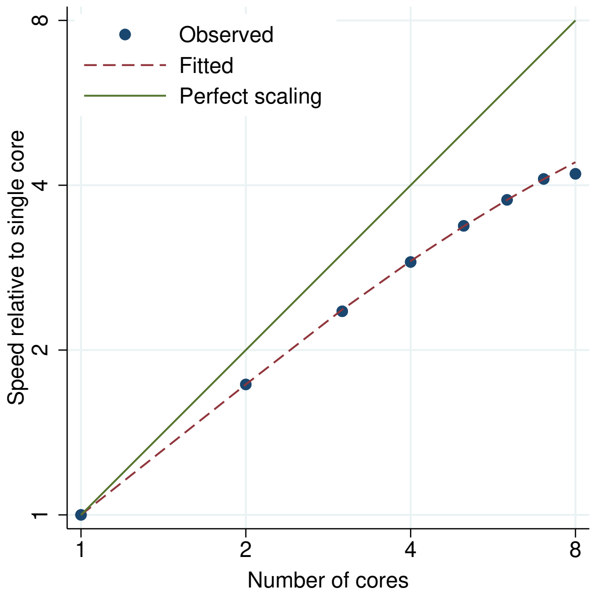


Figure 381. `prtest1` performance plot.

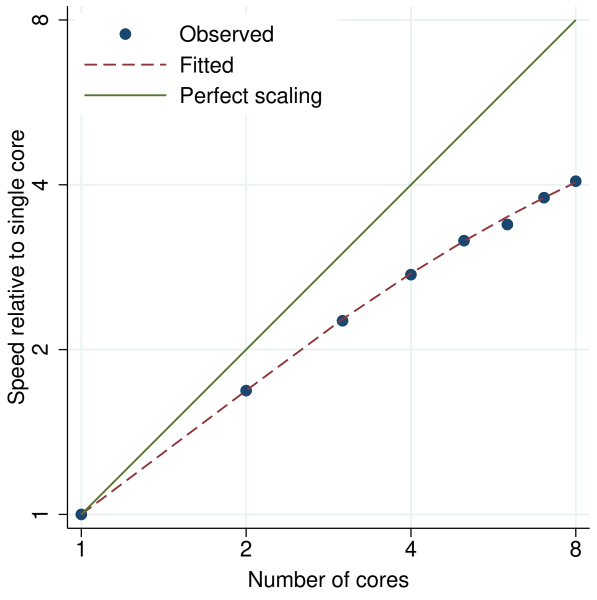


Figure 382. `prtest2` performance plot.

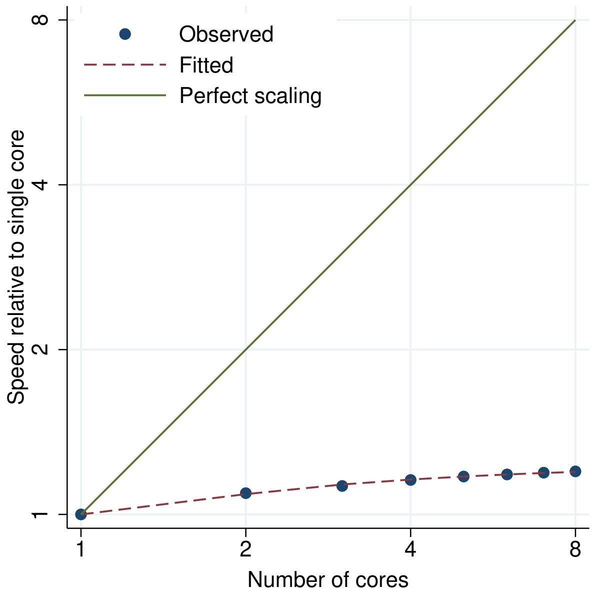


Figure 383. `prtest, by()` performance plot.

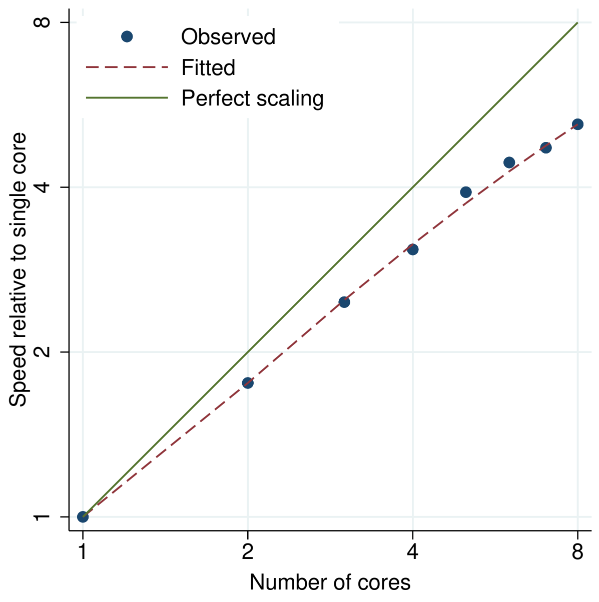


Figure 384. `pwcorr` performance plot.

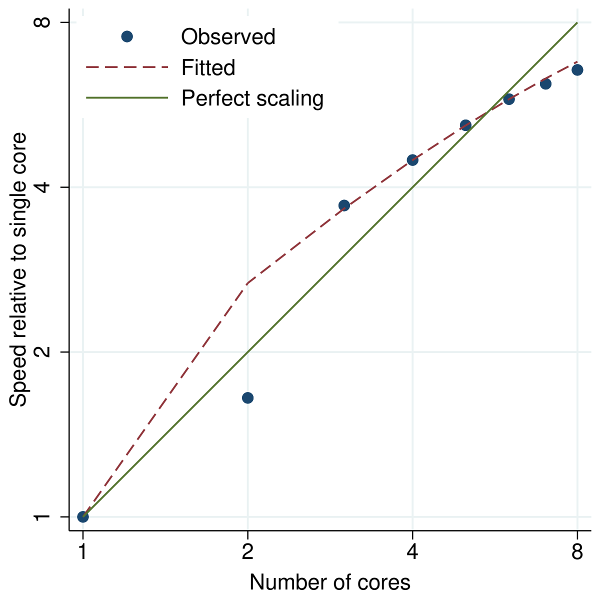


Figure 385. `qreg` performance plot.

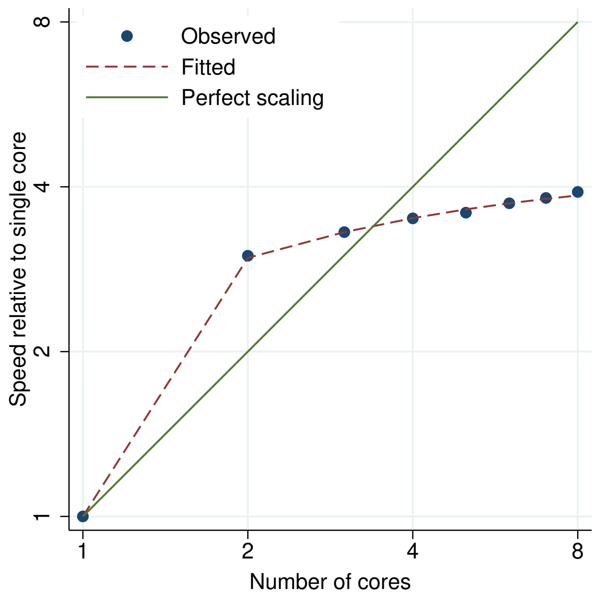


Figure 386. `ranksum` performance plot.

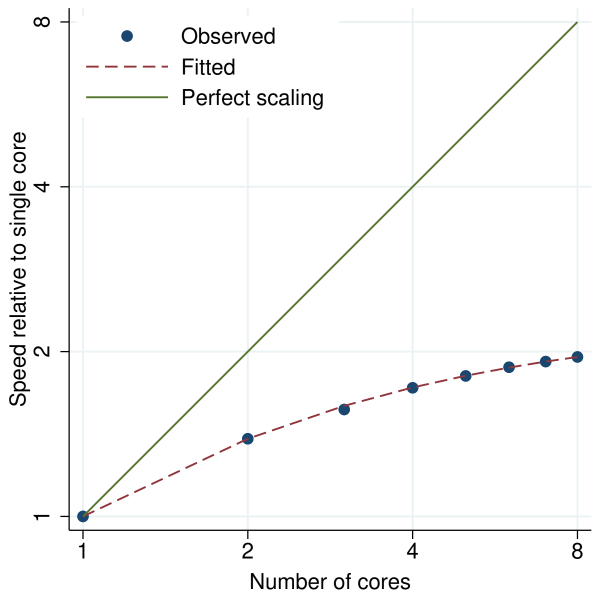


Figure 387. `ratio` performance plot.

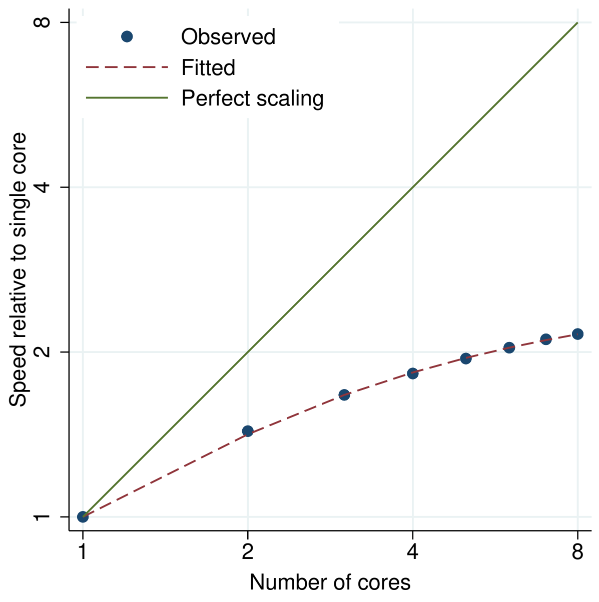


Figure 388. `ratio (exp1) (exp2)` performance plot.

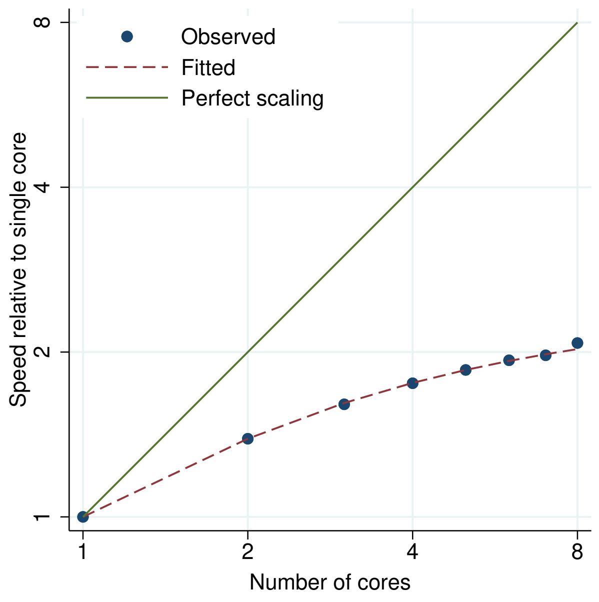


Figure 389. `recode` performance plot.

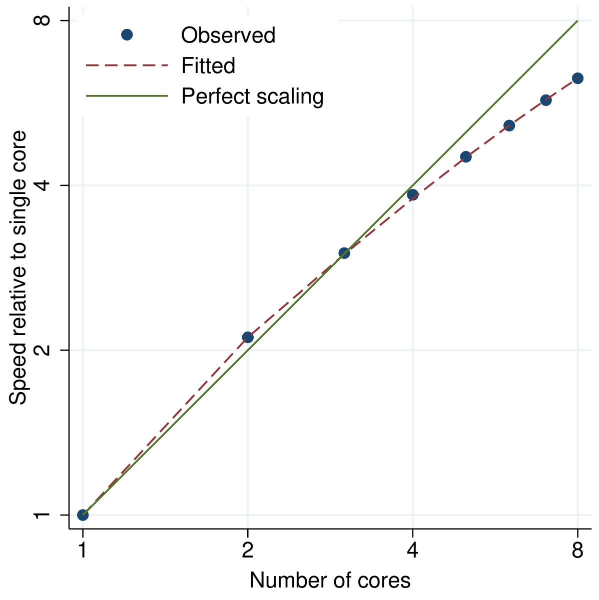


Figure 390. `reg3` performance plot.

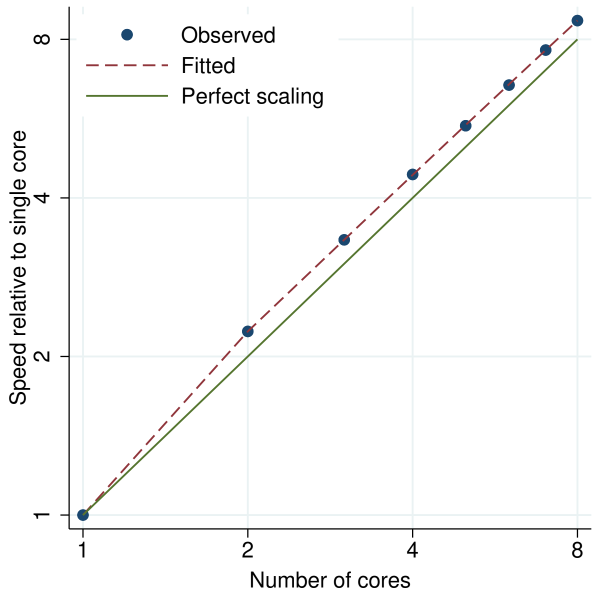


Figure 391. `regress` performance plot.

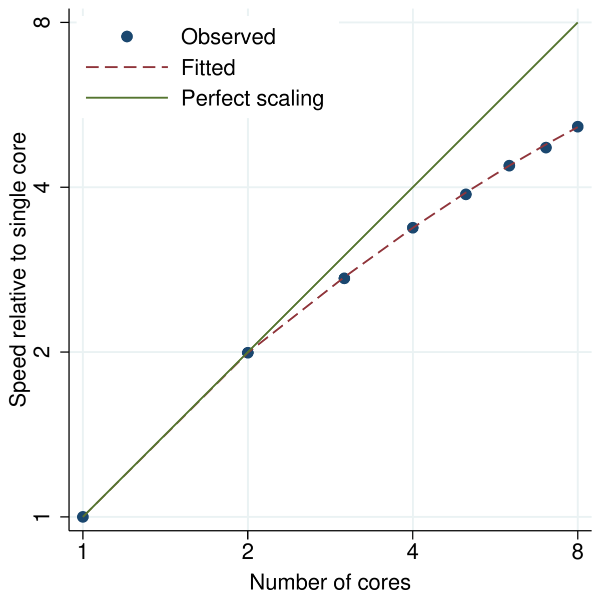


Figure 392. `regress, vce(cluster)` performance plot.

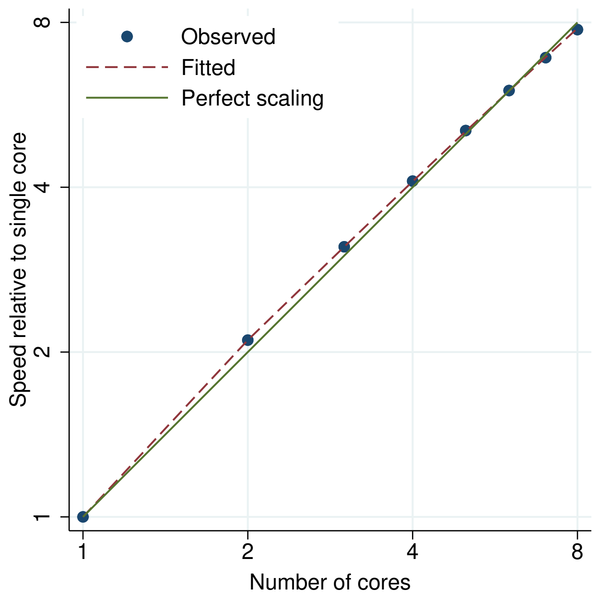


Figure 393. `regress, vce(robust)` performance plot.

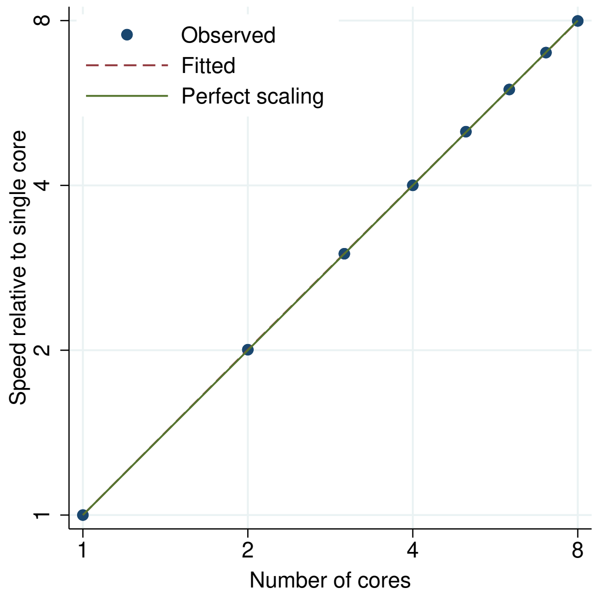


Figure 394. `replace` performance plot.

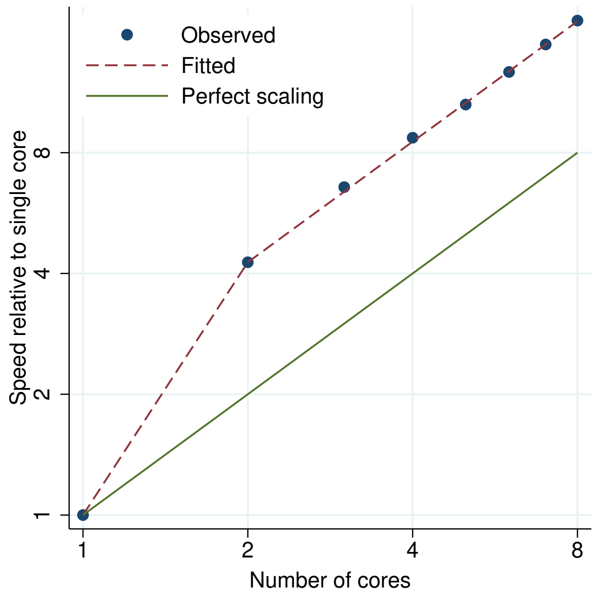


Figure 395. `replace` (small expressions) performance plot.

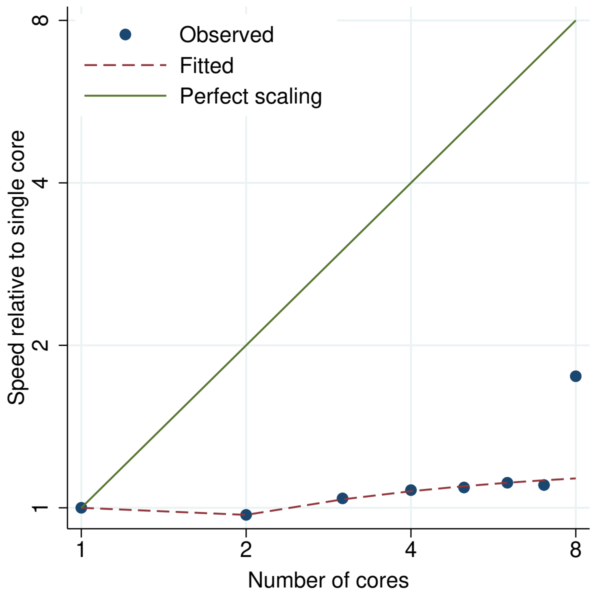


Figure 396. `reshape long` performance plot.

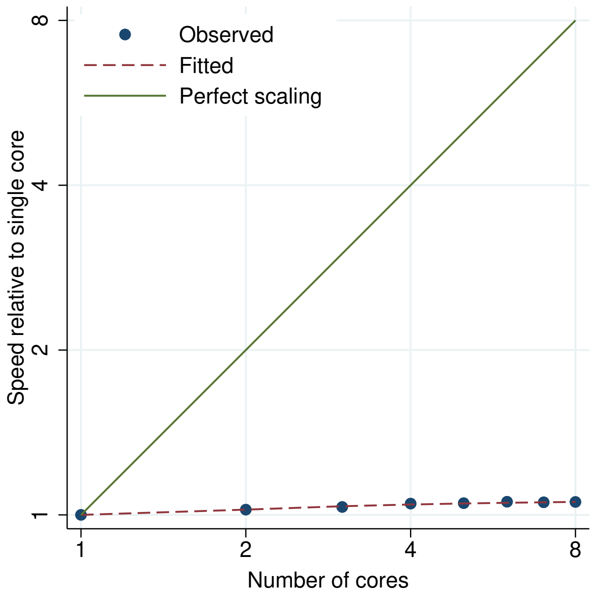


Figure 397. `reshape wide` performance plot.

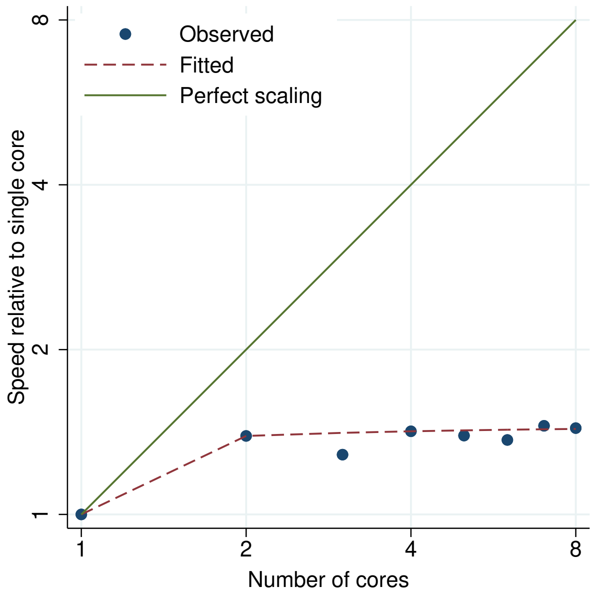


Figure 398. `robvar` performance plot.

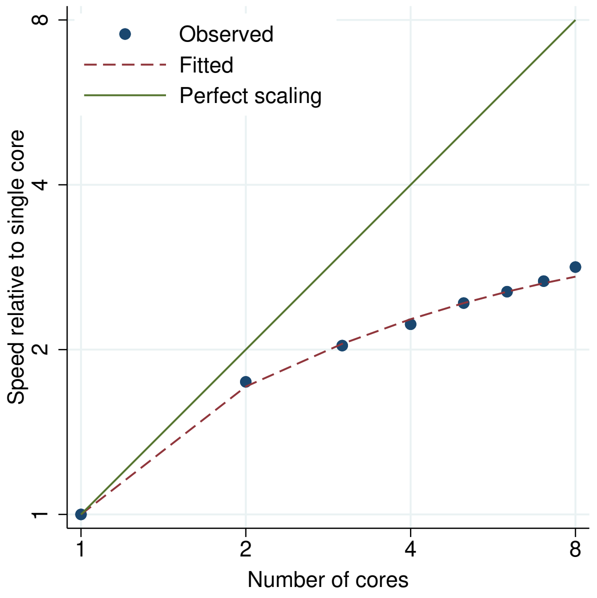


Figure 399. `rocfit` performance plot.

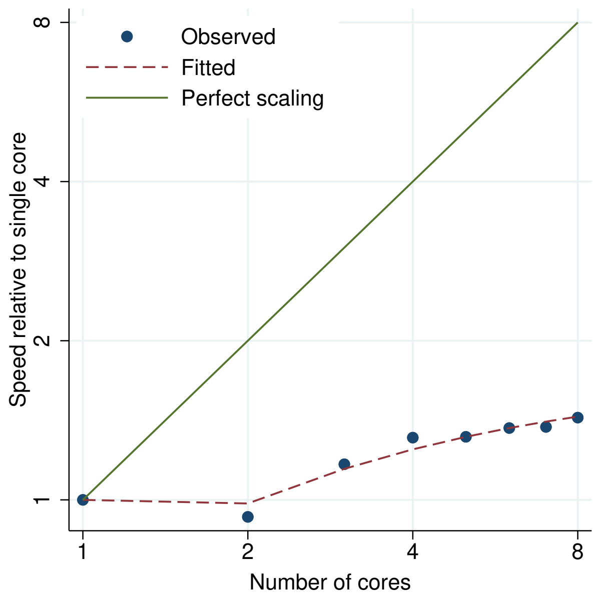


Figure 400. `roctab` performance plot.

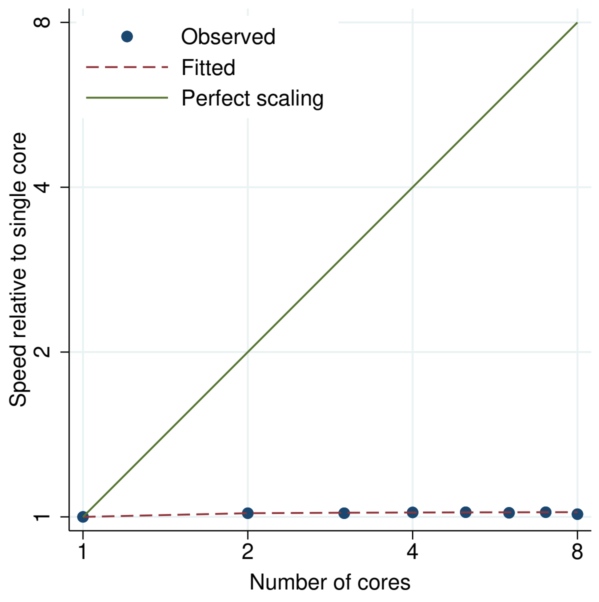


Figure 401. `rotate` performance plot.

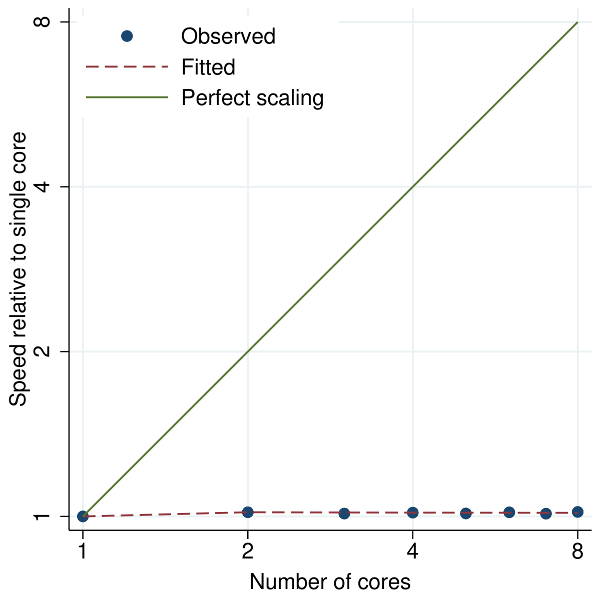


Figure 402. `rotatemat` performance plot.

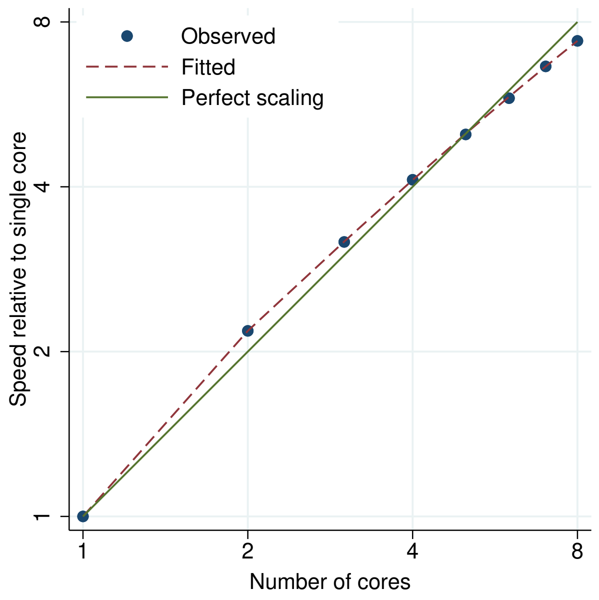


Figure 403. `rreg` performance plot.

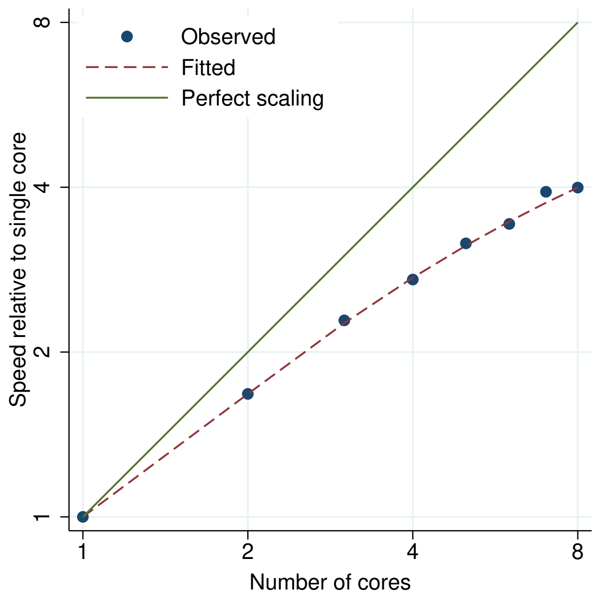


Figure 404. `runtest` performance plot.

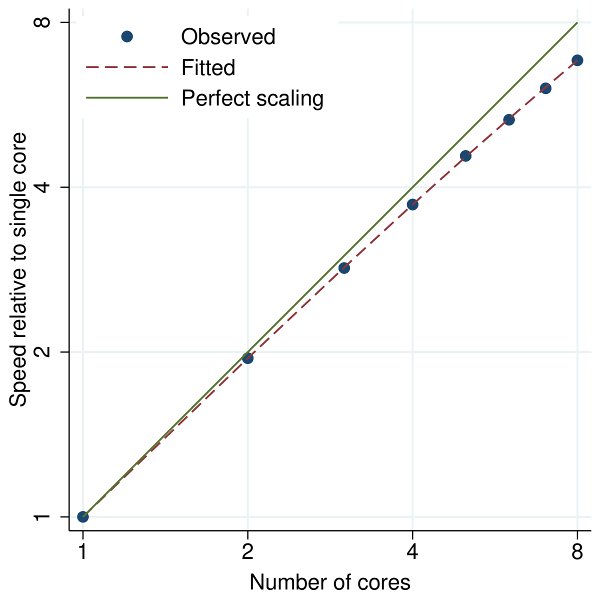


Figure 405. `scobit` performance plot.

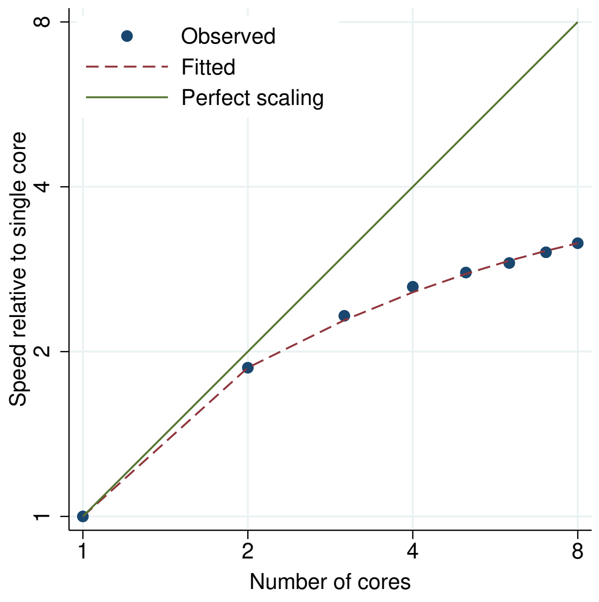


Figure 406. `scoreplot` performance plot.

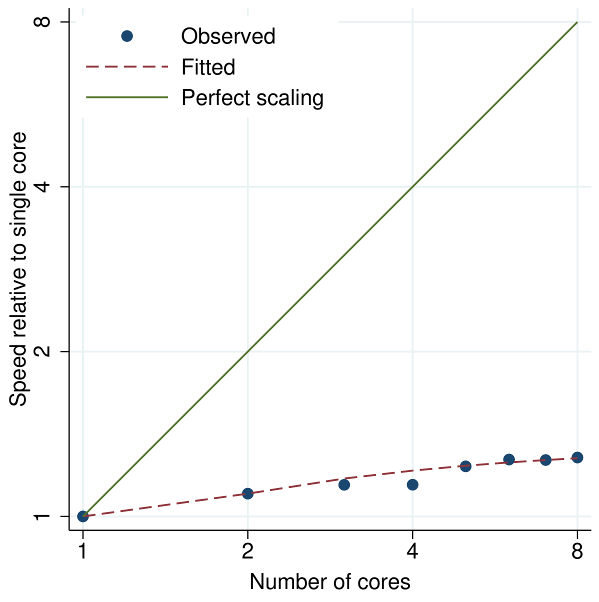


Figure 407. `screeplot` performance plot.

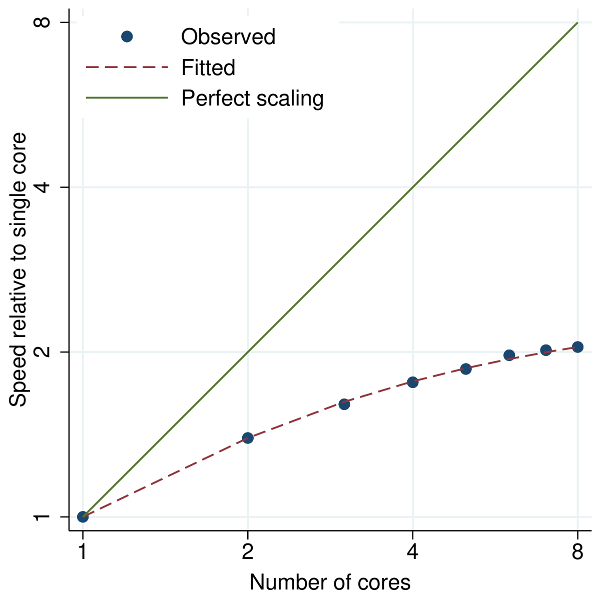


Figure 408. `sdtest1` performance plot.

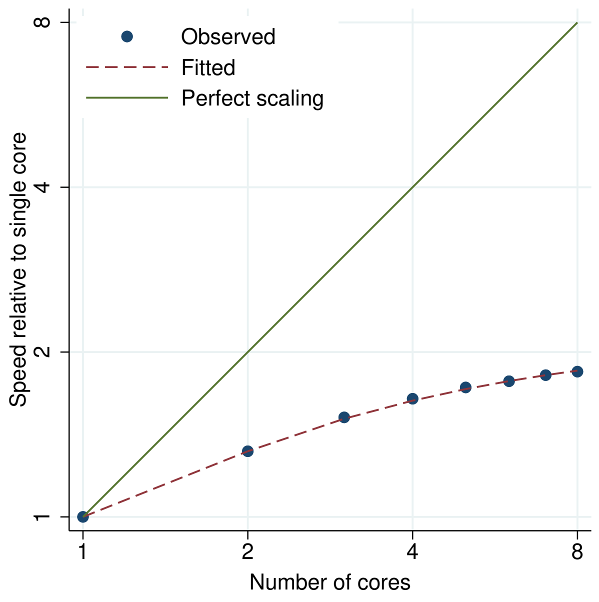


Figure 409. `sdtest2` performance plot.

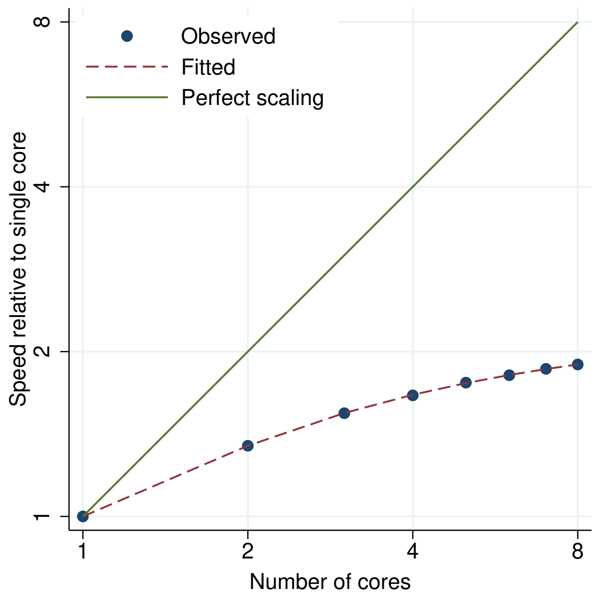


Figure 410. `sdtest, by()` performance plot.

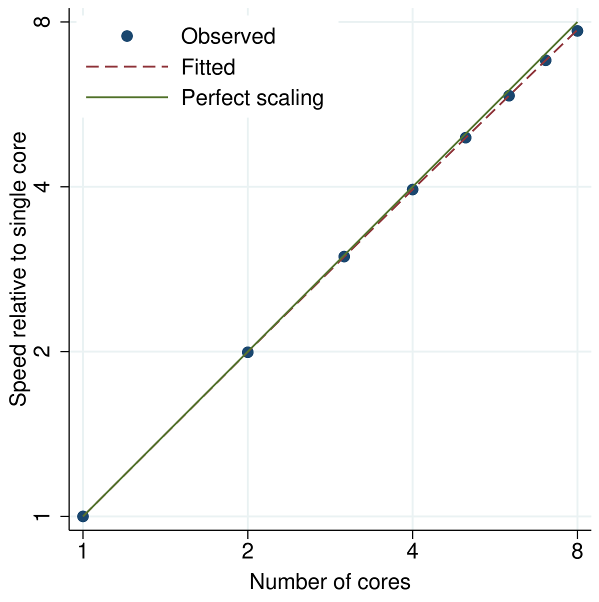


Figure 411. `sem, method(adf) (CFA)` performance plot.

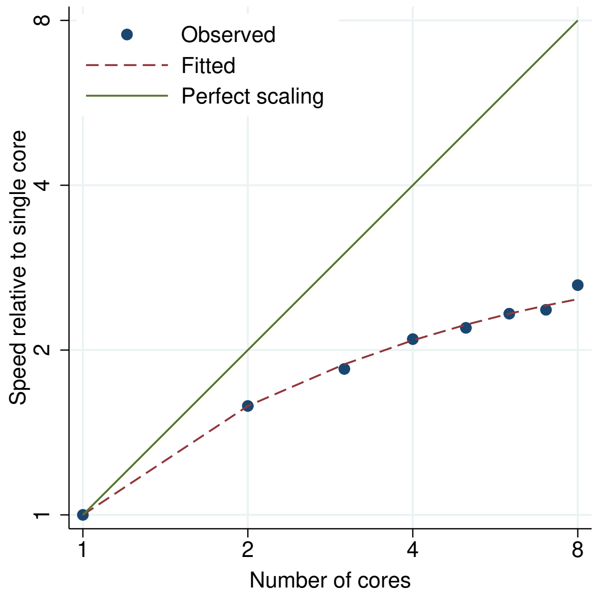


Figure 412. `sem, method(ml) (CFA)` performance plot.

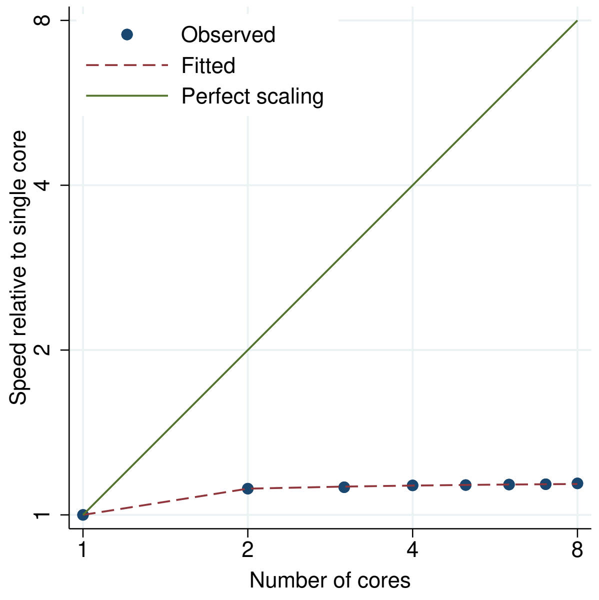


Figure 413. `sem, method(mlmv) (CFA)` performance plot.

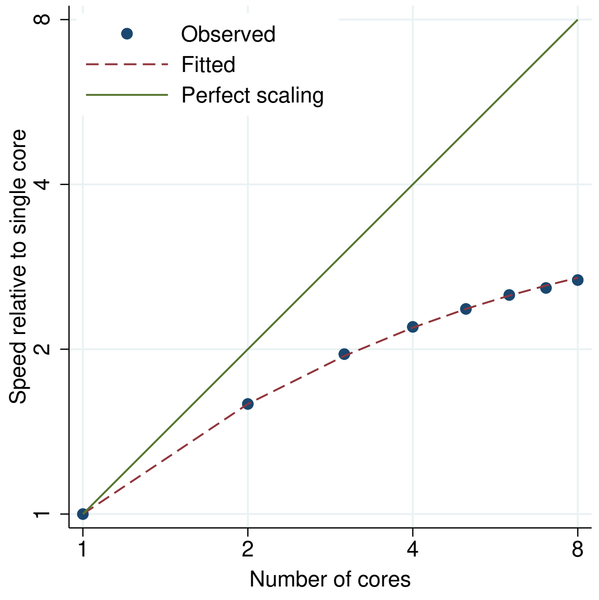


Figure 414. `sem (SEM latent)` performance plot.

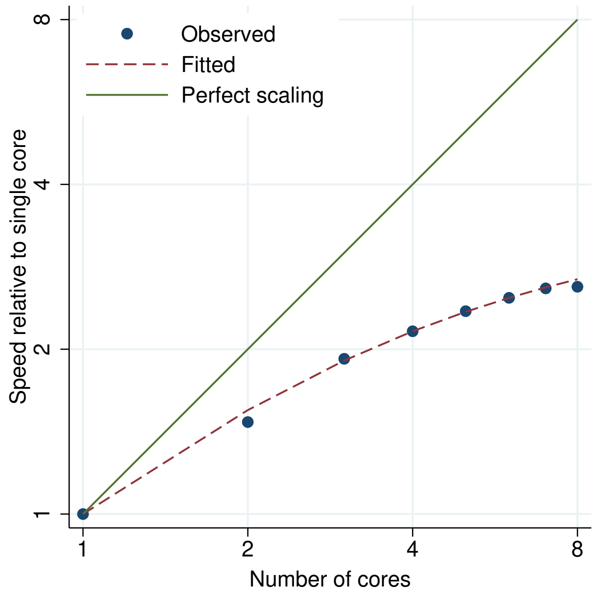


Figure 415. `sem (SEM observed)` performance plot.

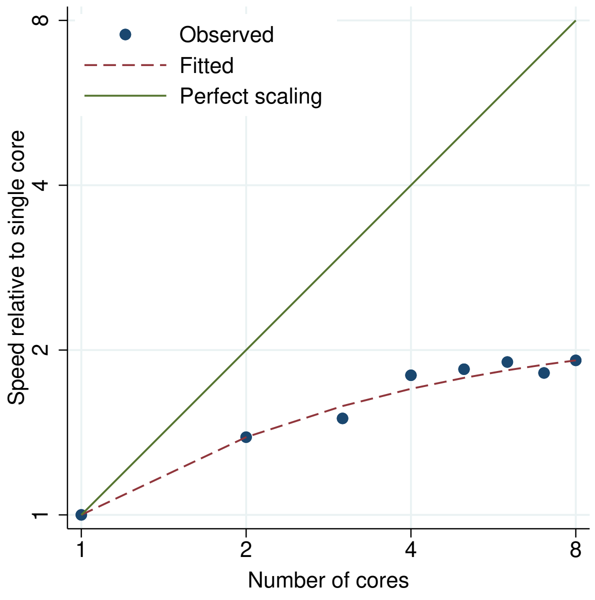


Figure 416. `separate` performance plot.

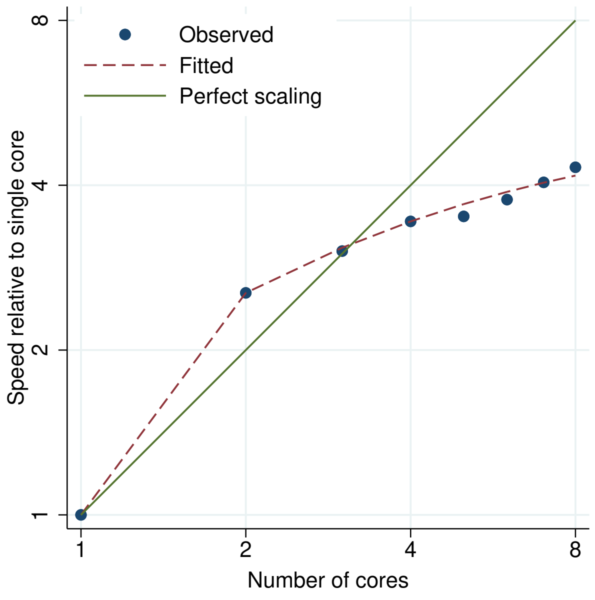


Figure 417. `sfrancia` performance plot.

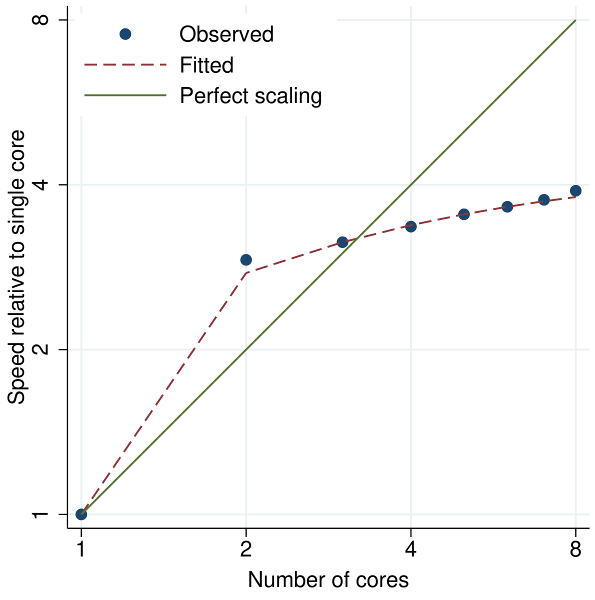


Figure 418. `signrank` performance plot.

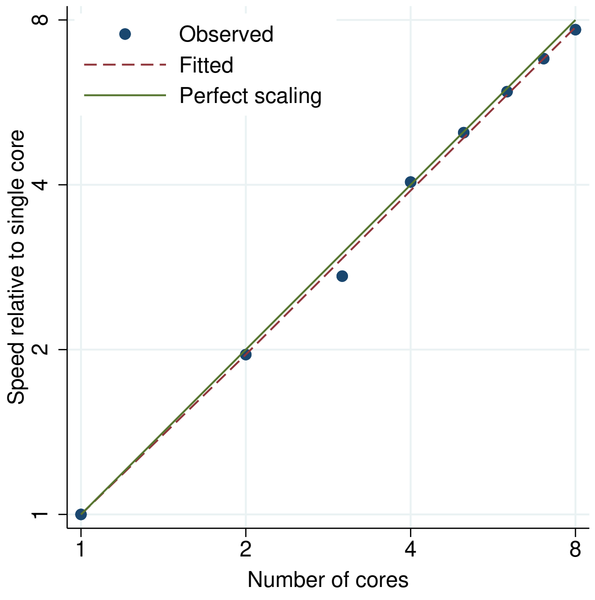


Figure 419. `signtest` performance plot.

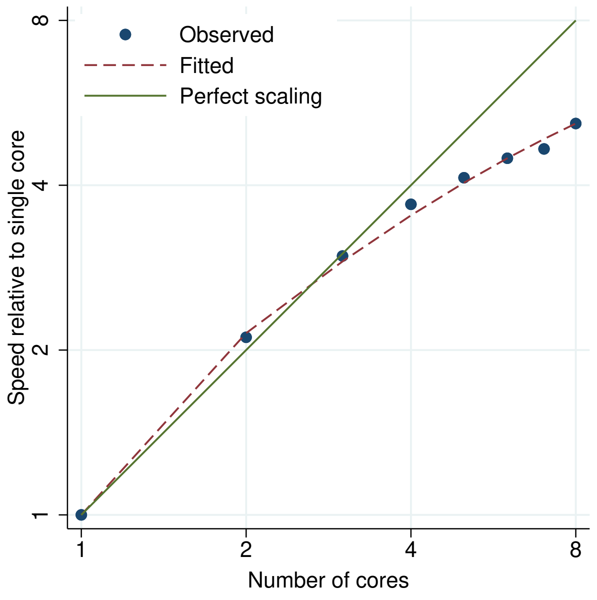


Figure 420. `sktest` performance plot.

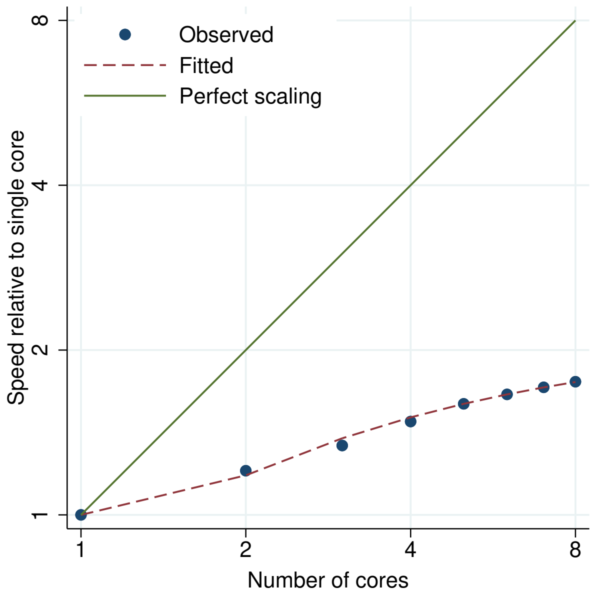


Figure 421. `slogit` performance plot.

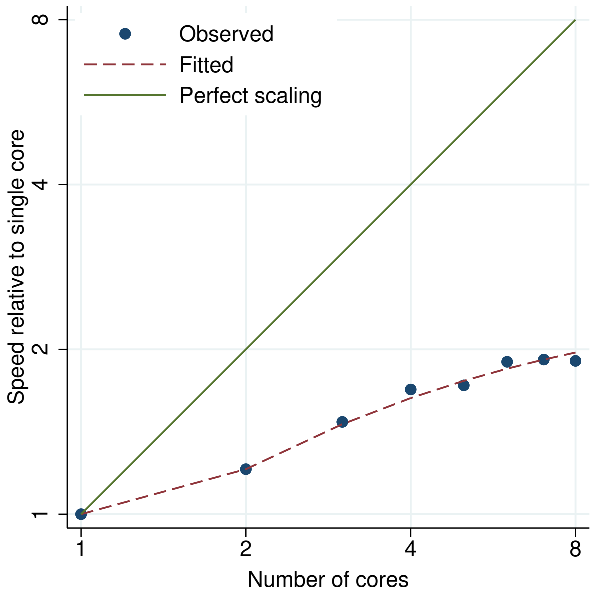


Figure 422. `sort` performance plot.

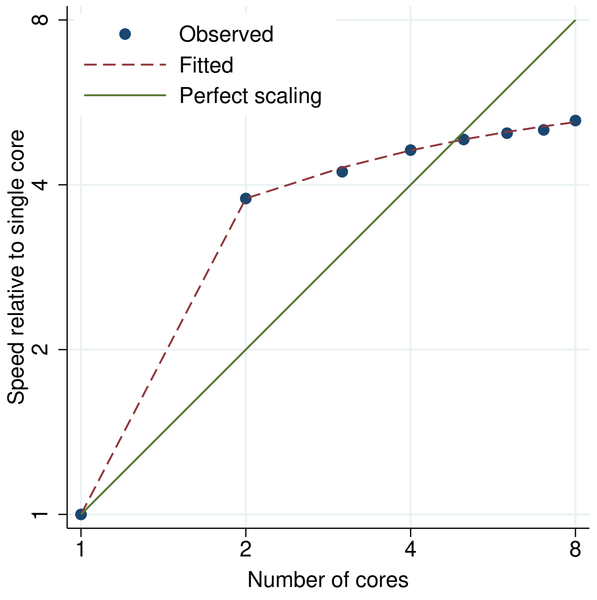


Figure 423. `spearman` performance plot.

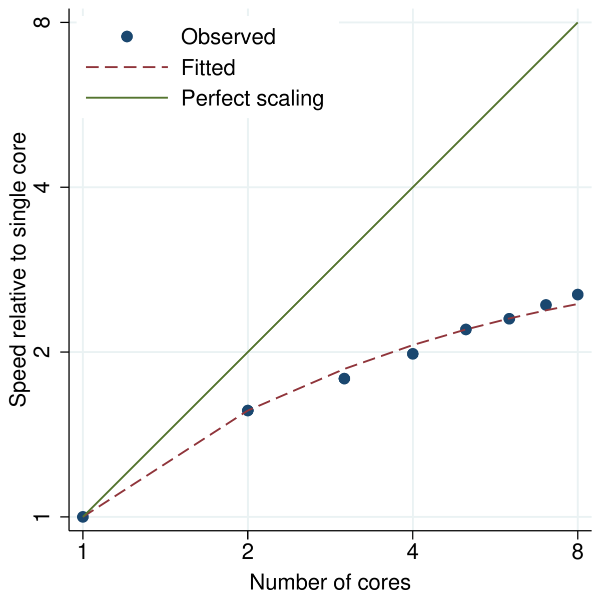


Figure 424. `sspace` performance plot.

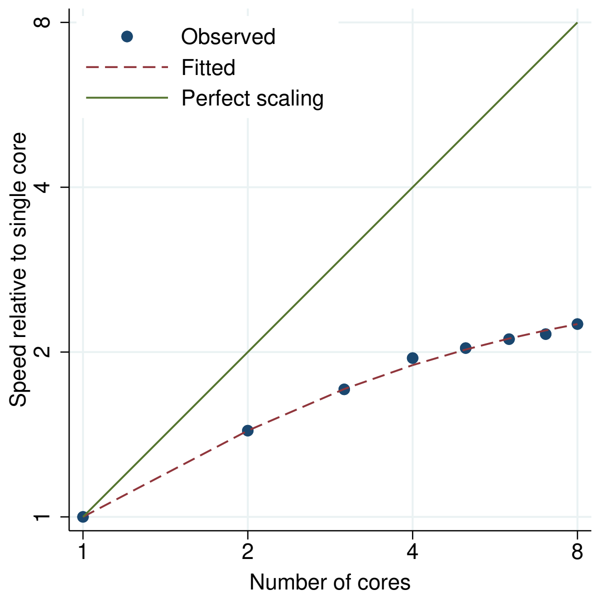


Figure 425. `stack` performance plot.

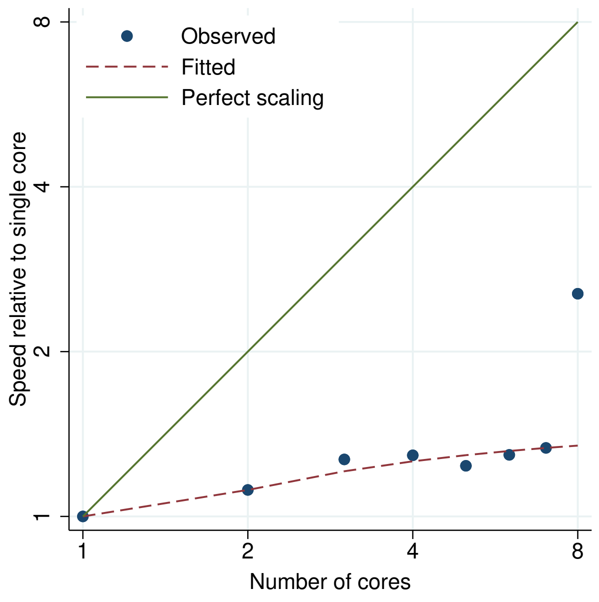


Figure 426. `stci` performance plot.

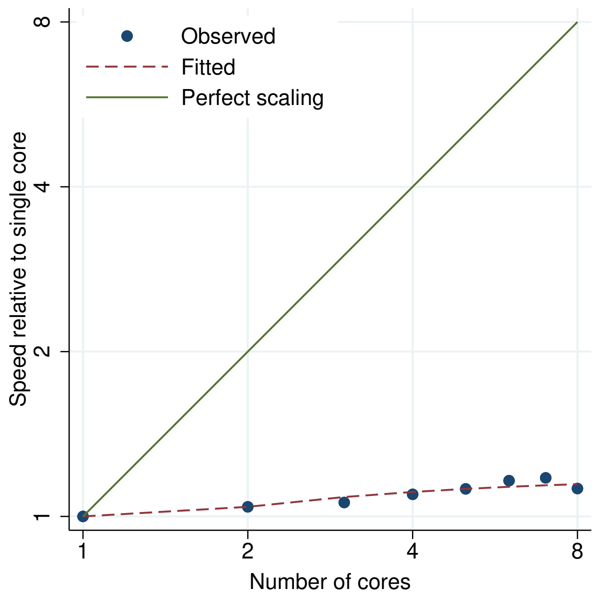


Figure 427. `stcox` performance plot.

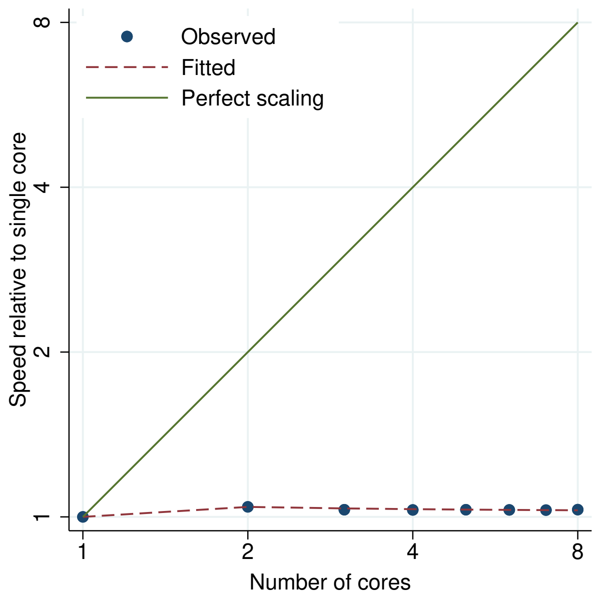


Figure 428. `stcrreg` performance plot.

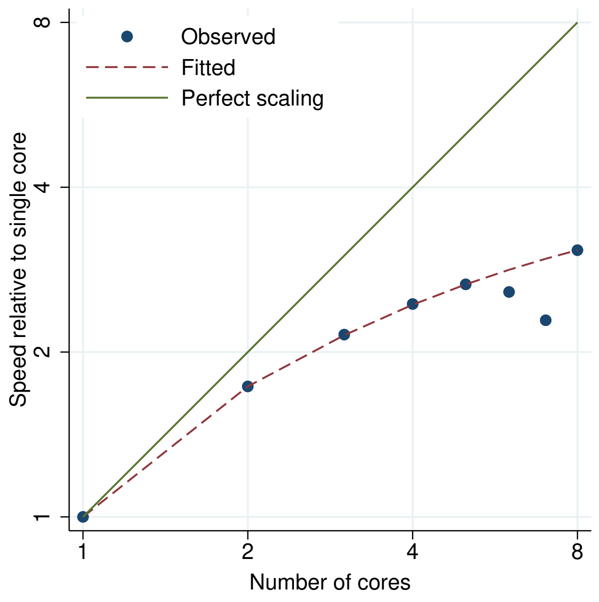


Figure 429. `stgen` performance plot.

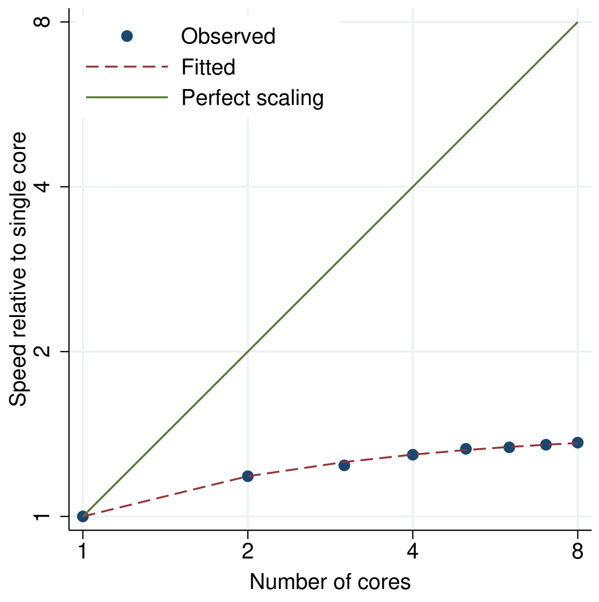


Figure 430. `stintcox` performance plot.

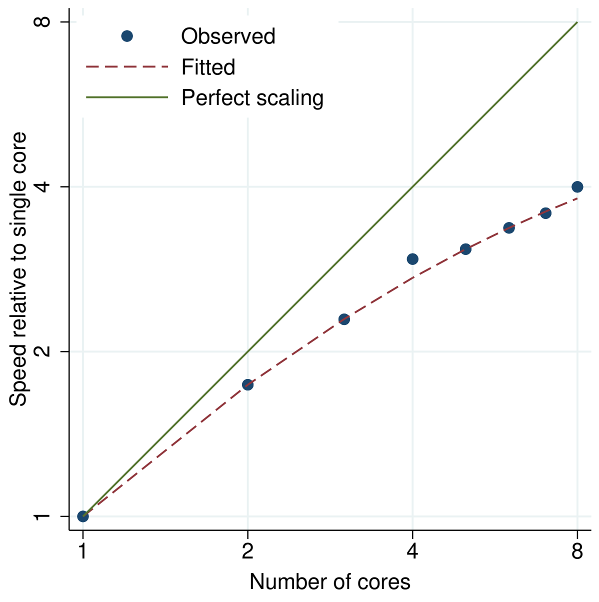


Figure 431. `stintreg, d(exponential)` performance plot.

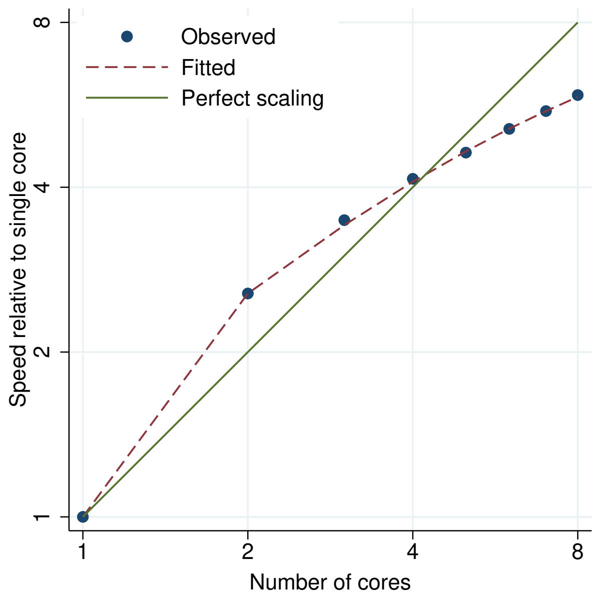


Figure 432. `stintreg, d(weibull)` performance plot.

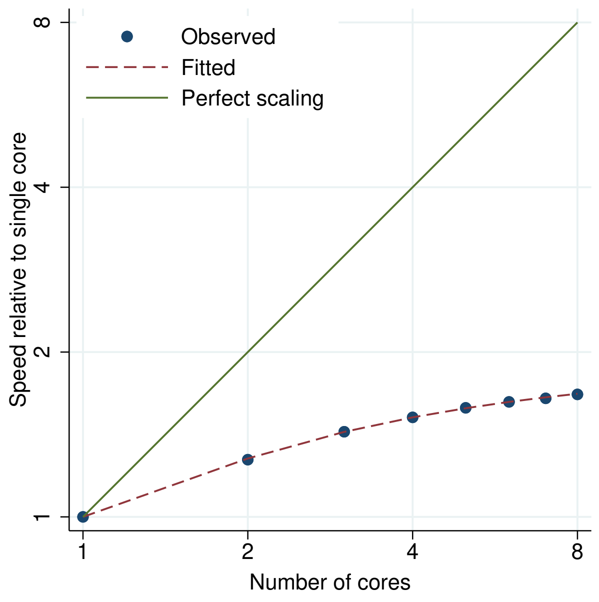


Figure 433. `stir` performance plot.

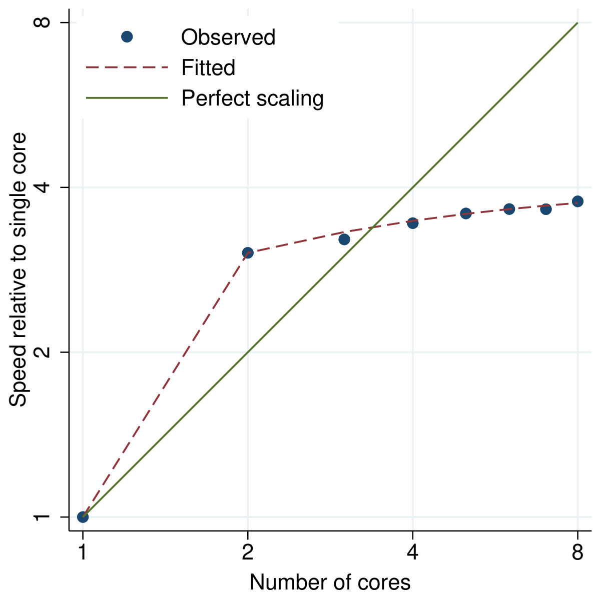


Figure 434. `stmc` performance plot.

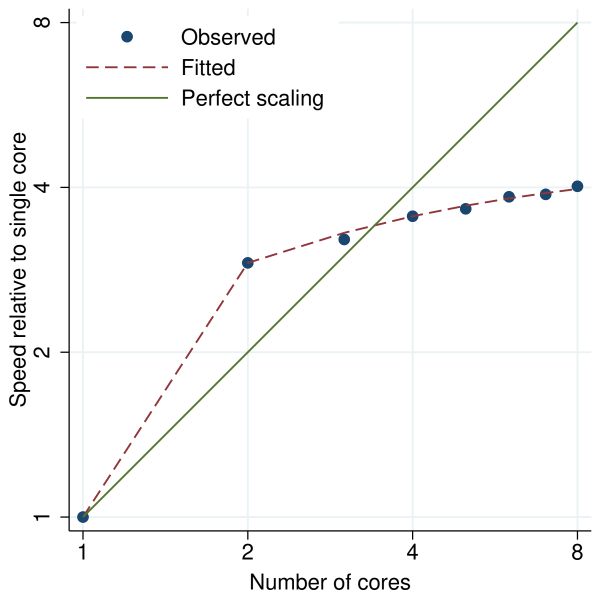


Figure 435. `by: stmc` performance plot.

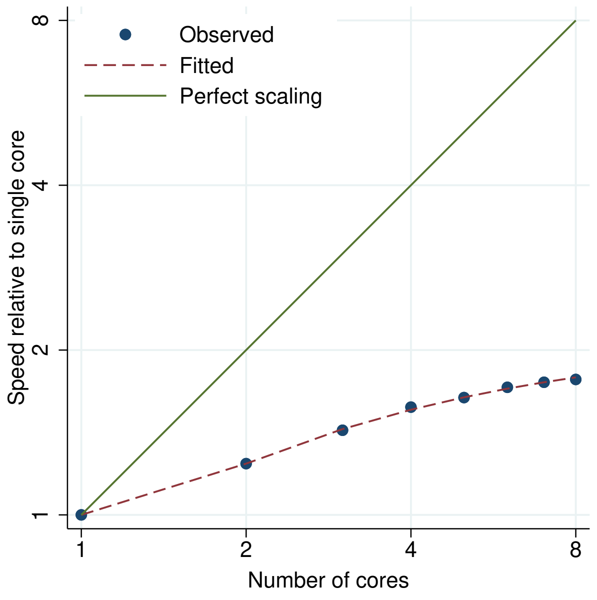


Figure 436. `stmh` performance plot.

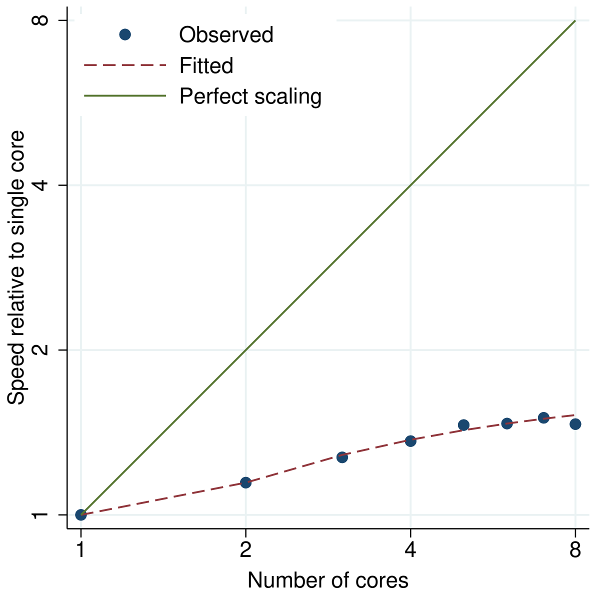


Figure 437. `by: stmh` performance plot.

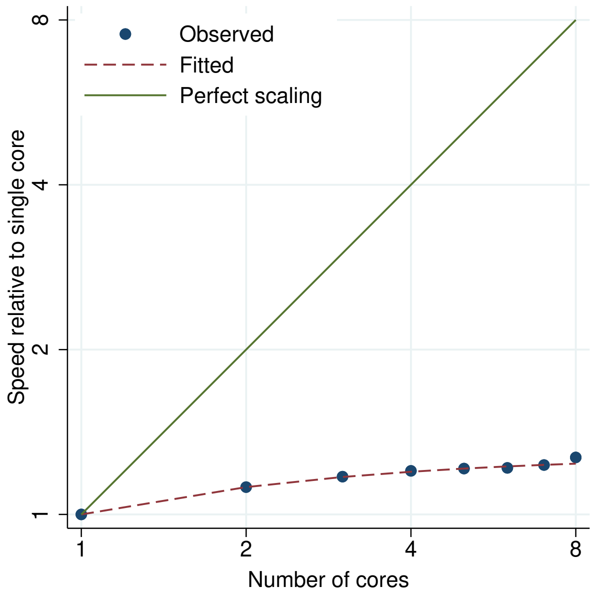


Figure 438. `stptime` performance plot.

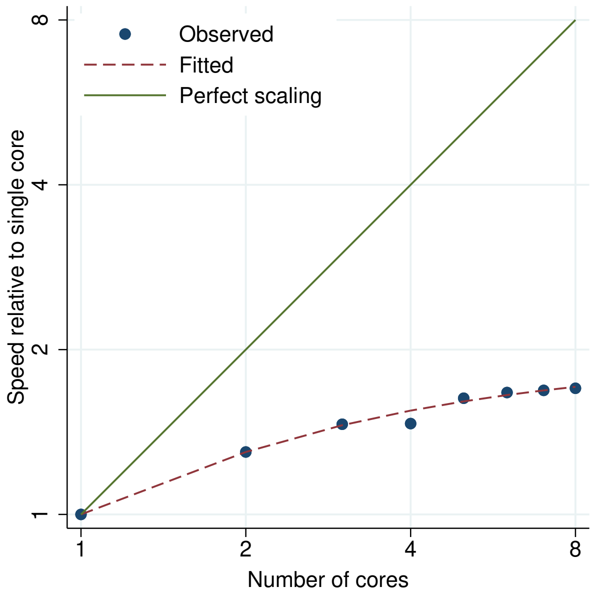


Figure 439. `strate` performance plot.

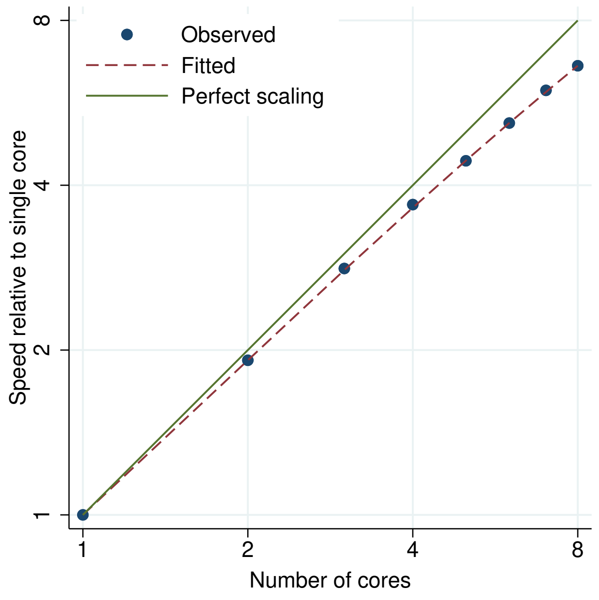


Figure 440. `streg, distribution(exponential)` performance plot.

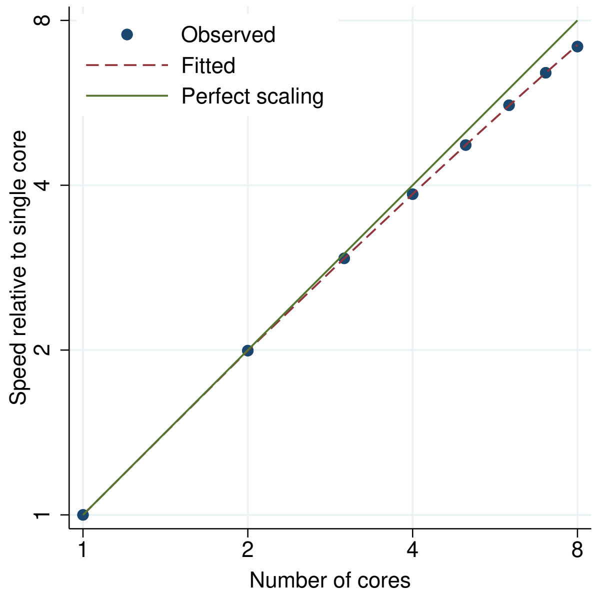


Figure 441. `streg, dist(exp) vce(cluster)` performance plot.

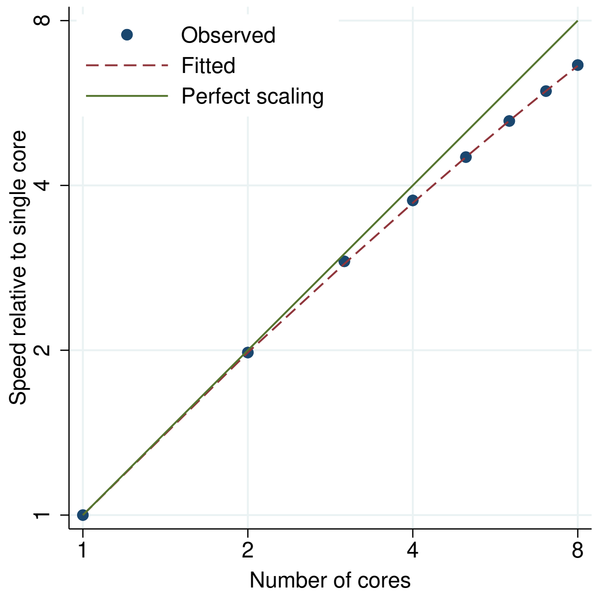


Figure 442. `streg, dist(exp) frailty()` performance plot.

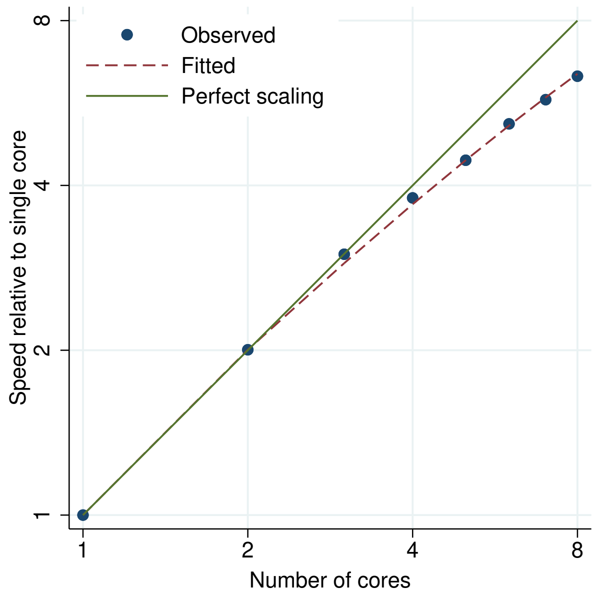


Figure 443. `streg, dist(exp) frailty() shared()` performance plot.

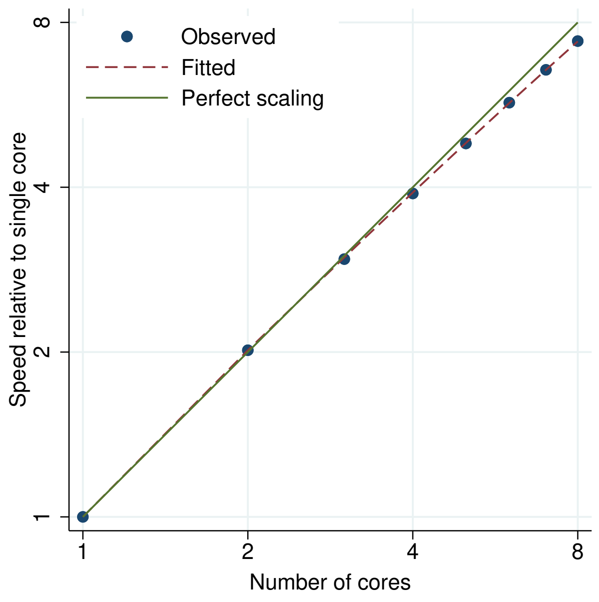


Figure 444. `streg, dist(exp) vce(robust)` performance plot.

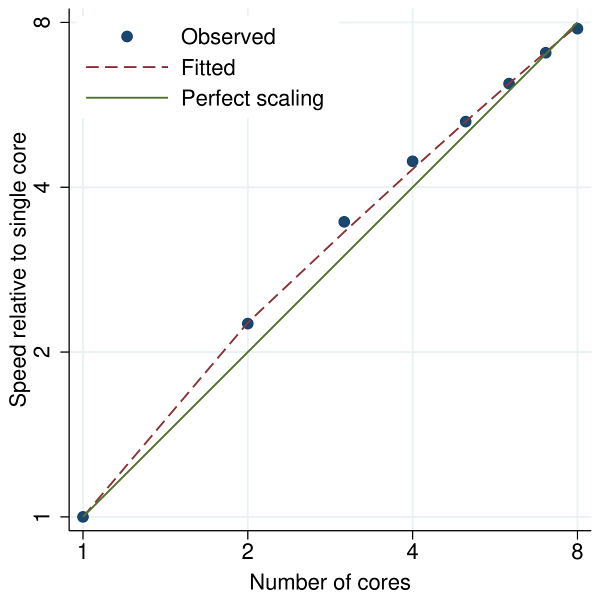


Figure 445. `streg, distribution(ggamma)` performance plot.

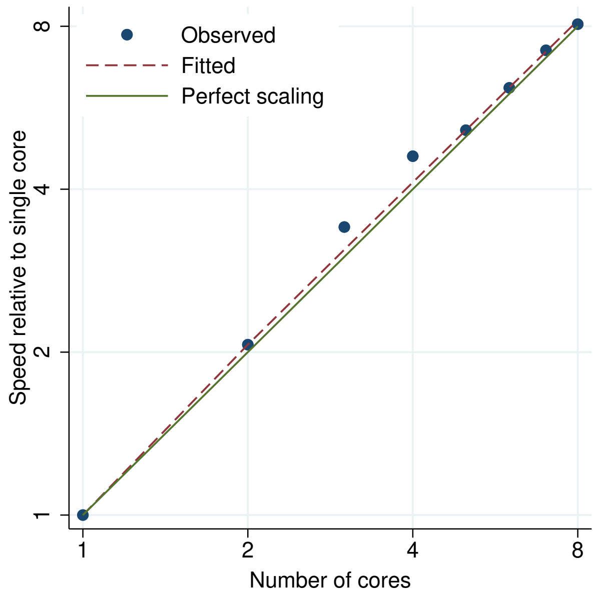


Figure 446. `streg, dist(ggamma) vce(cluster)` performance plot.

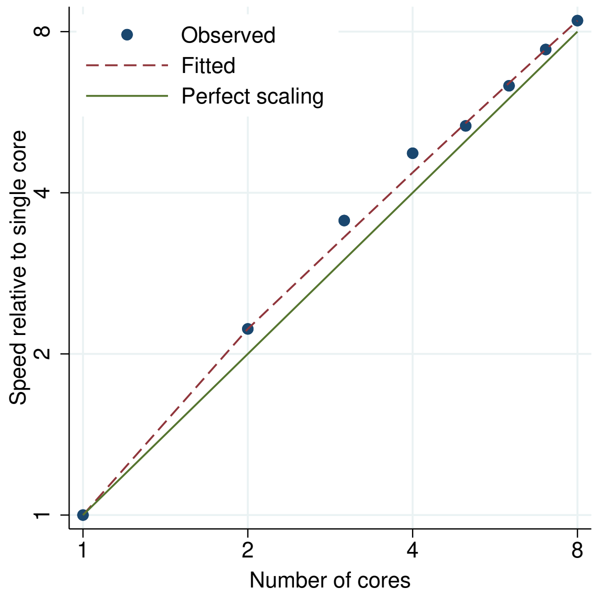


Figure 447. `streg, dist(ggamma) vce(robust)` performance plot.

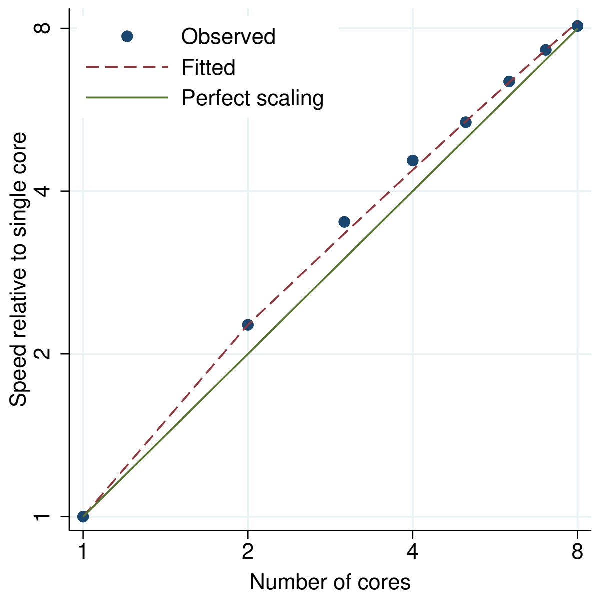


Figure 448. `streg, distribution(gompertz)` performance plot.

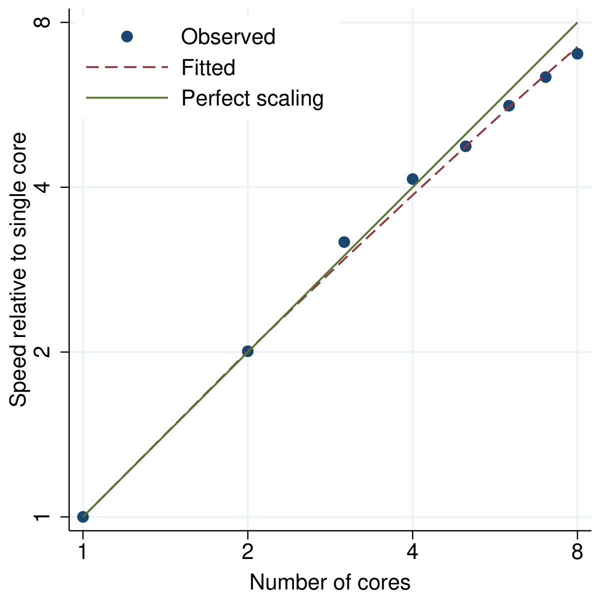


Figure 449. `streg, dist(gompertz) vce(cluster)` performance plot.

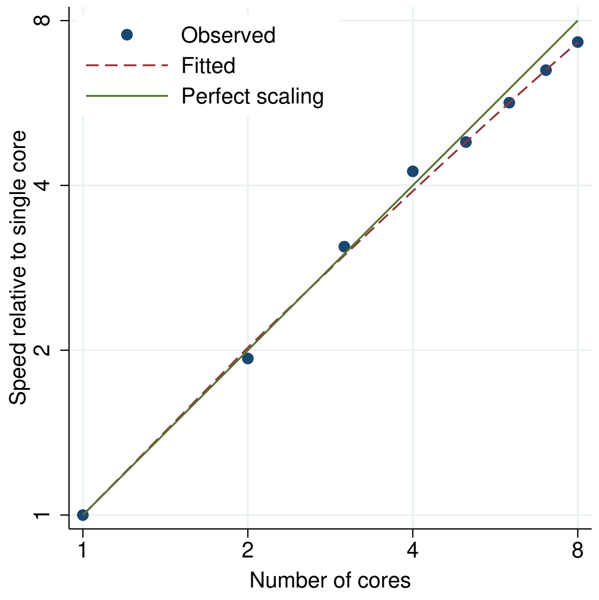


Figure 450. `streg, dist(gompertz) frailty()` performance plot.

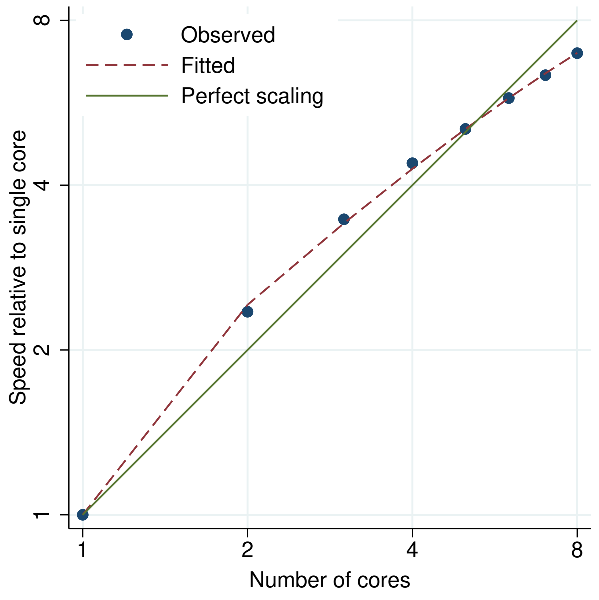


Figure 451. `streg, dist(gomp) frailty() shared()` performance plot.

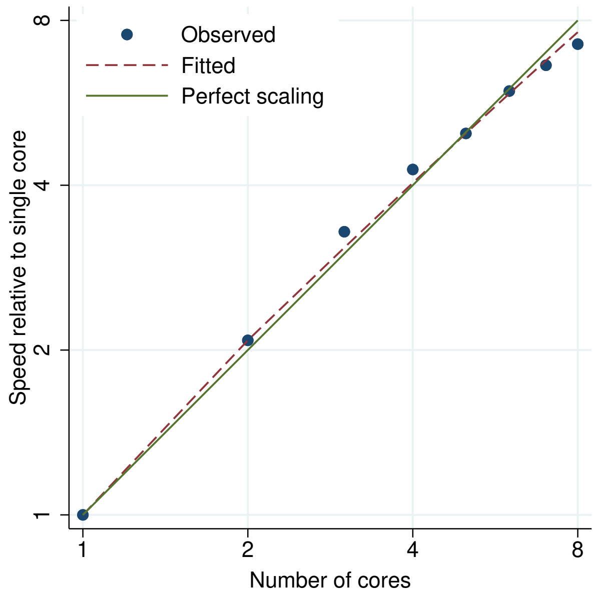


Figure 452. `streg, dist(gompertz) vce(robust)` performance plot.

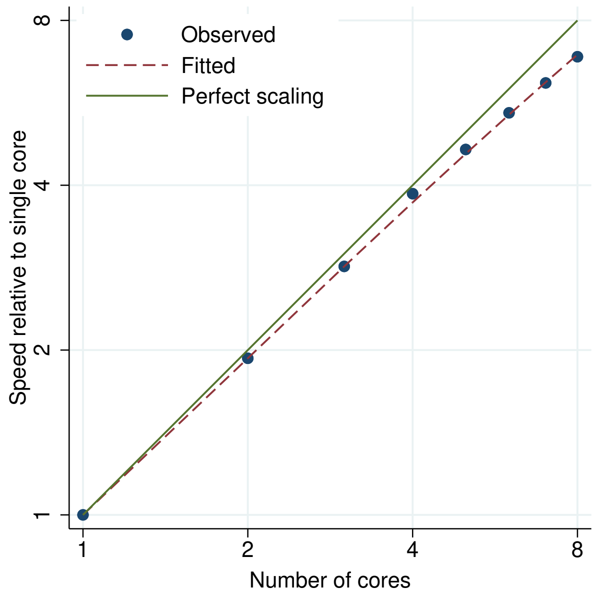


Figure 453. `streg, distribution(llogistic)` performance plot.

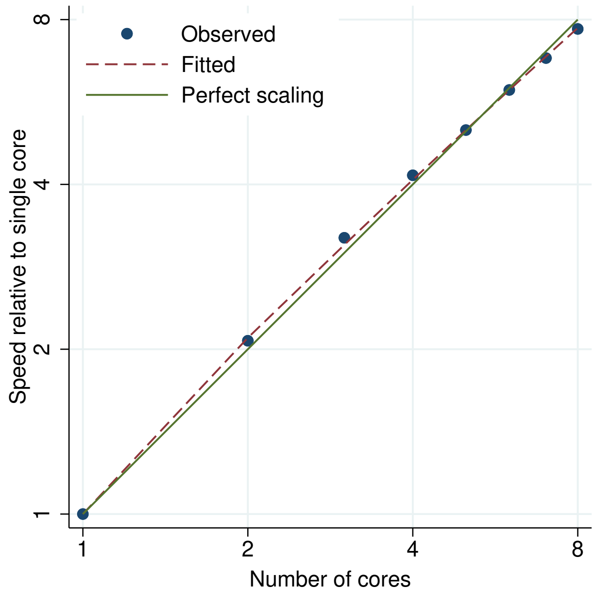


Figure 454. `streg, dist(llogistic) vce(cluster)` performance plot.

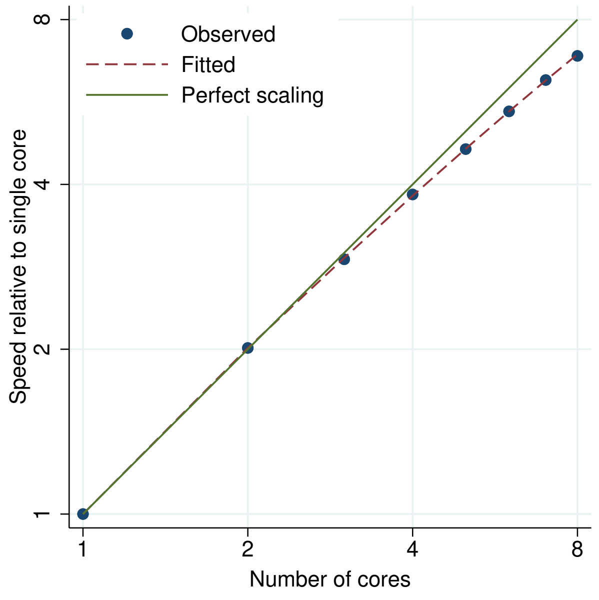


Figure 455. `streg, dist(llogistic) frailty()` performance plot.

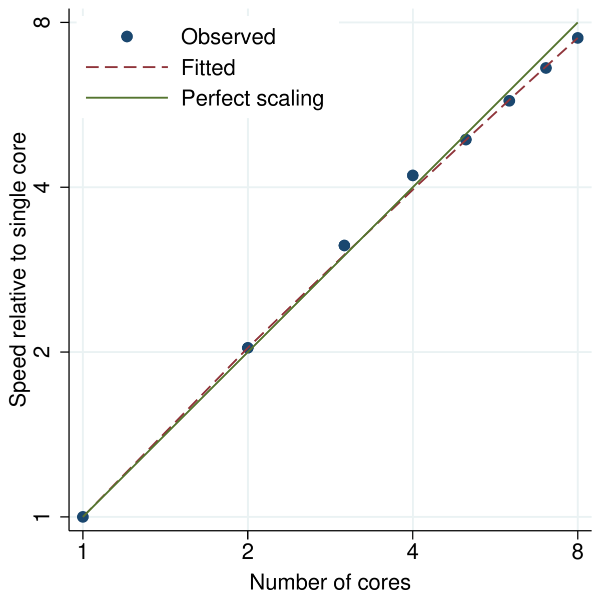


Figure 456. `streg, dist(llog) frailty() shared()` performance plot.

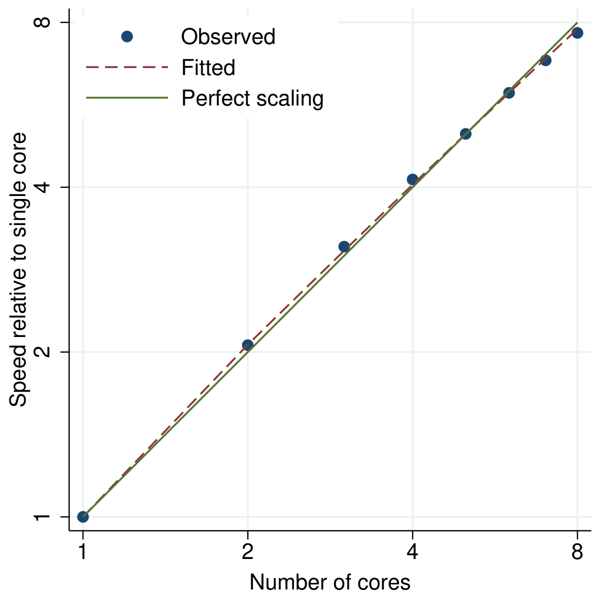


Figure 457. `streg, dist(llogistic) vce(robust)` performance plot.

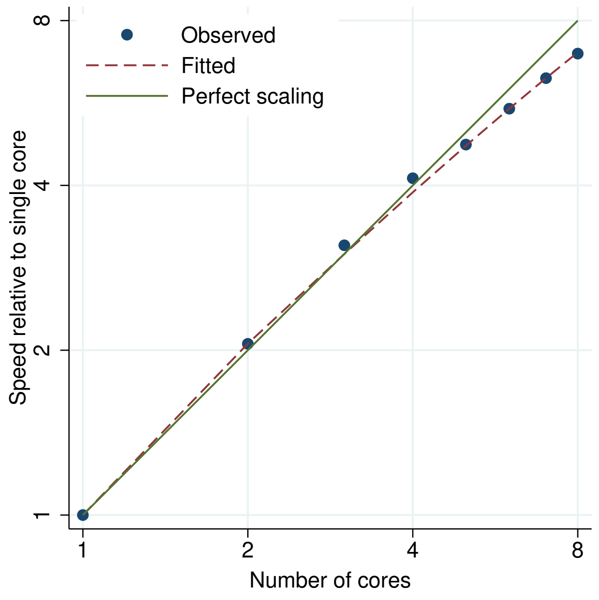


Figure 458. `streg, distribution(lnormal)` performance plot.

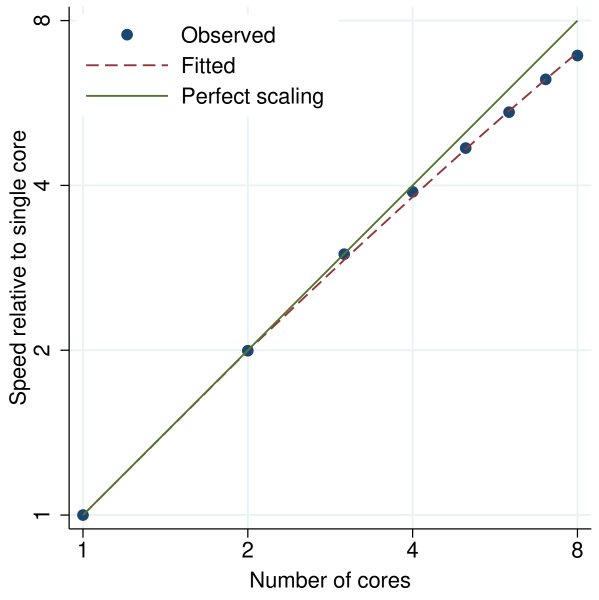


Figure 459. `streg, dist(lnormal) vce(cluster)` performance plot.

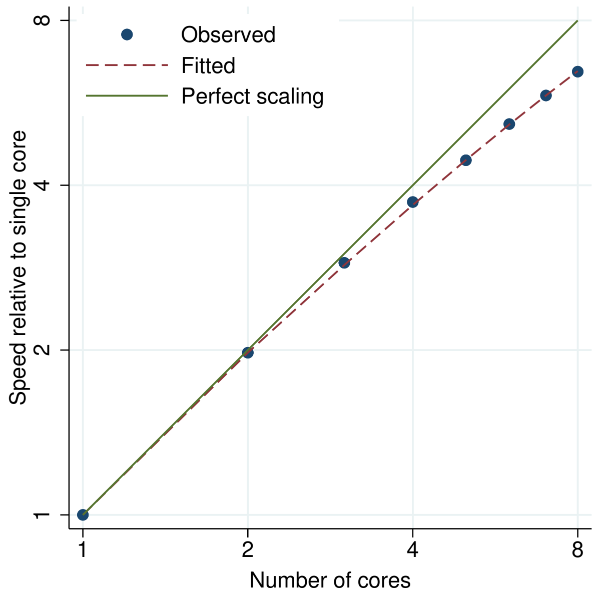


Figure 460. `streg, dist(lnormal) frailty()` performance plot.

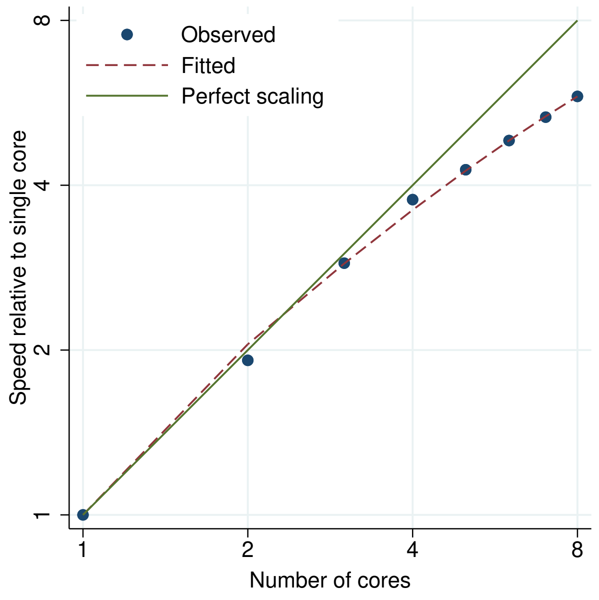


Figure 461. `streg, dist(lnorm) frailty() shared()` performance plot.

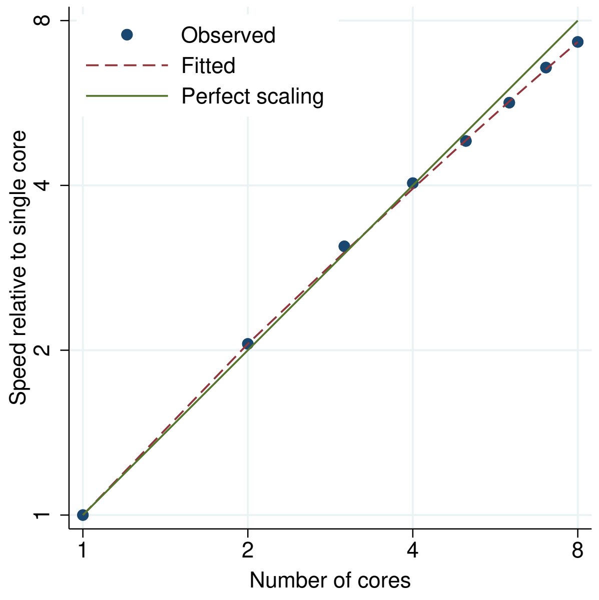


Figure 462. `streg, dist(lnormal) vce(robust)` performance plot.

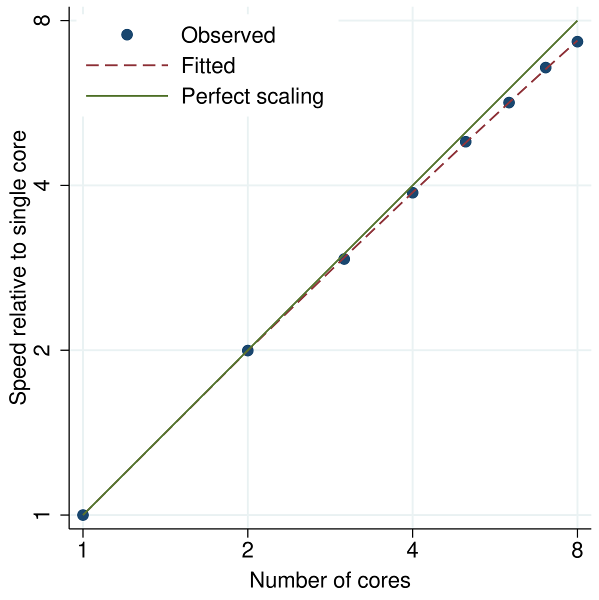


Figure 463. `streg, distribution(weibull)` performance plot.

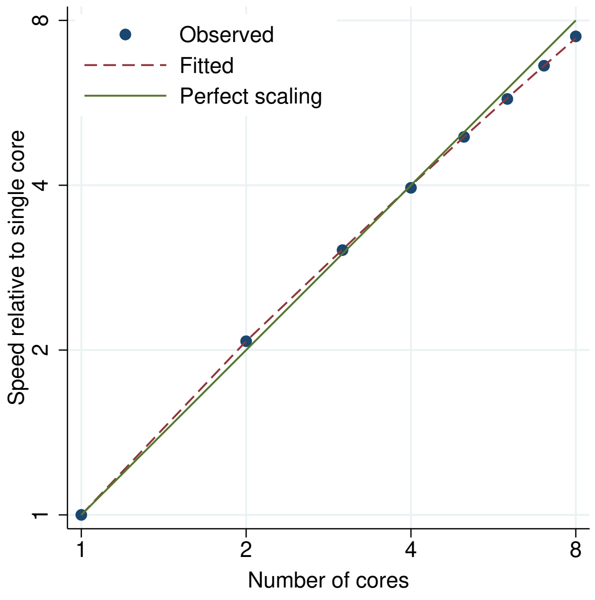


Figure 464. `streg, dist(weibull) vce(cluster)` performance plot.

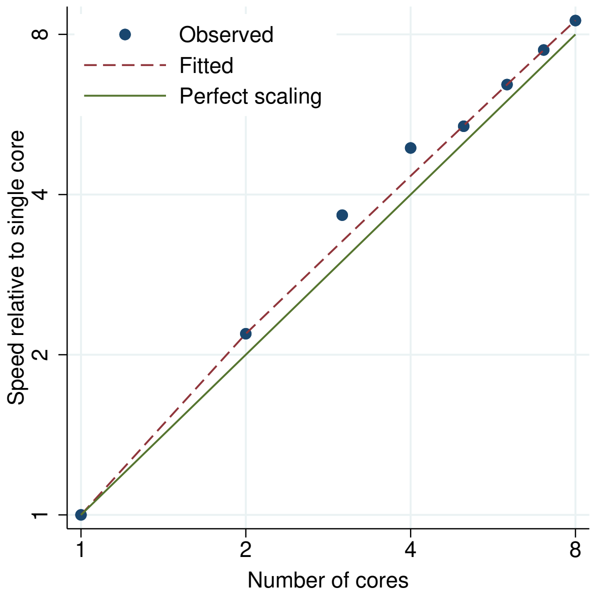


Figure 465. `streg, dist(weibull) frailty()` performance plot.

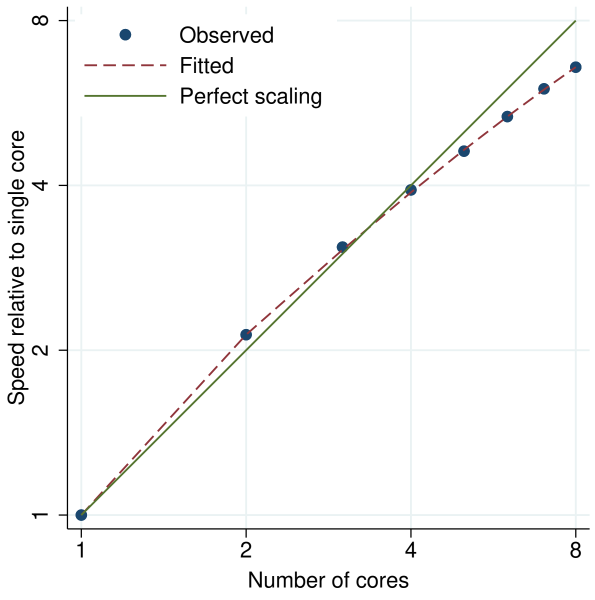


Figure 466. `streg, dist(weib) frailty() shared()` performance plot.

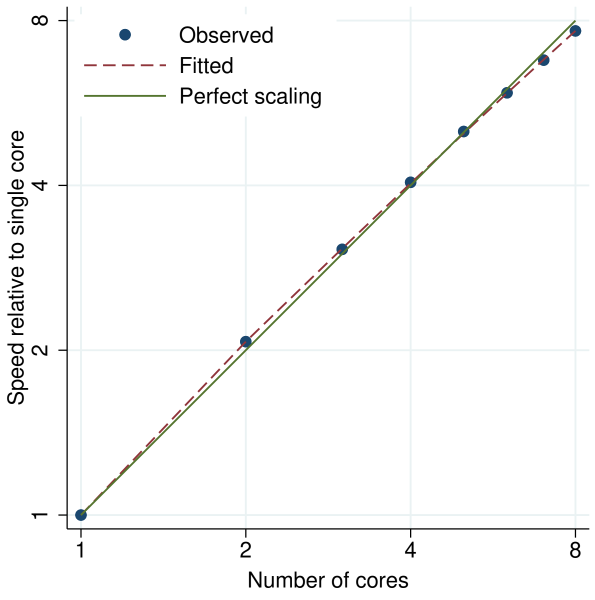


Figure 467. `streg, dist(weibull) vce(robust)` performance plot.

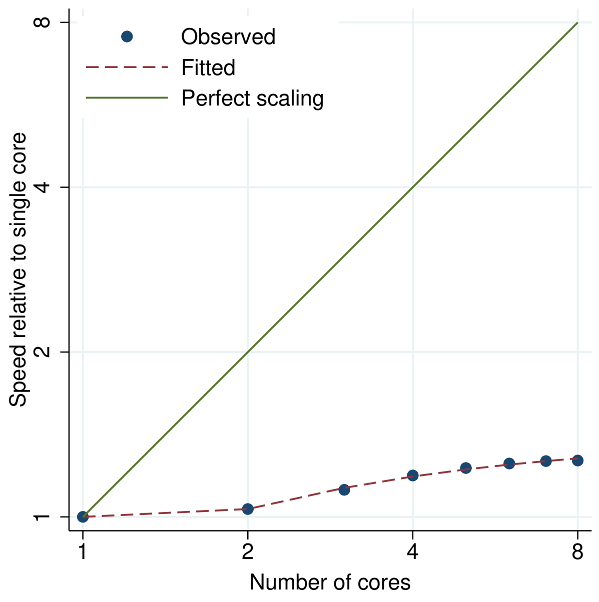


Figure 468. `sts generate` performance plot.

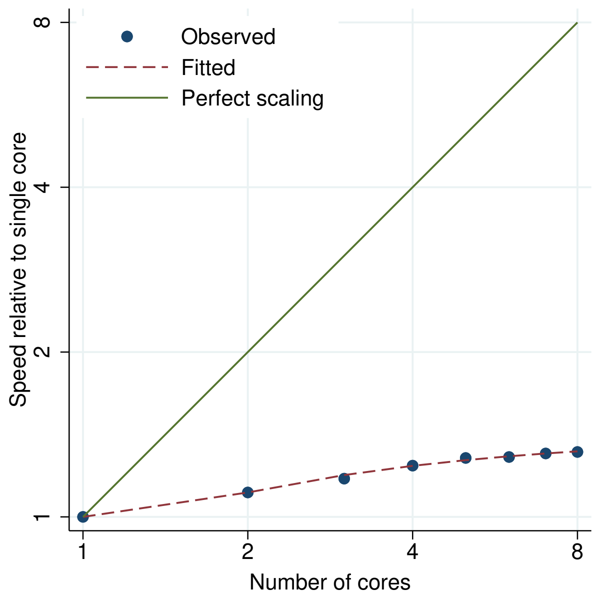


Figure 469. `sts graph` performance plot.

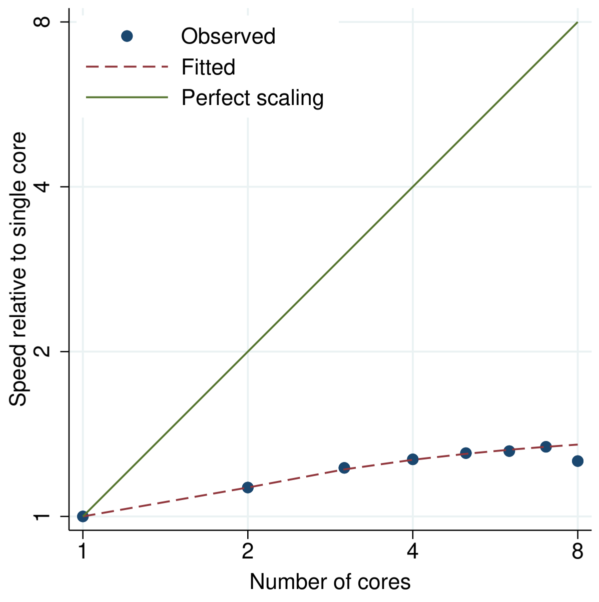


Figure 470. `sts list` performance plot.

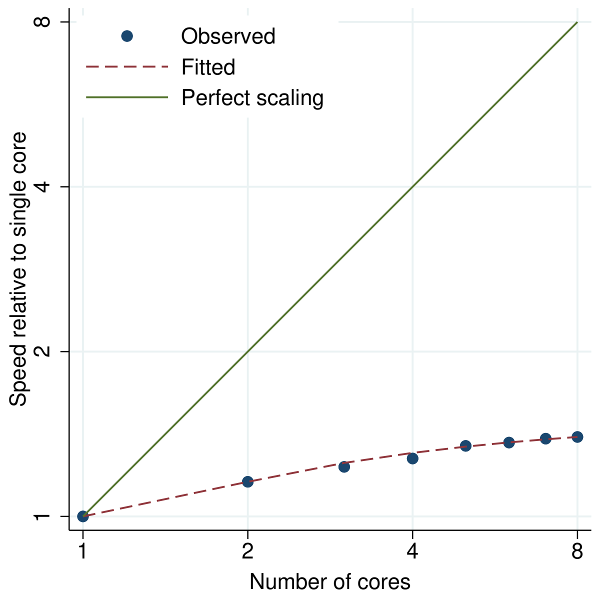


Figure 471. `sts test` performance plot.

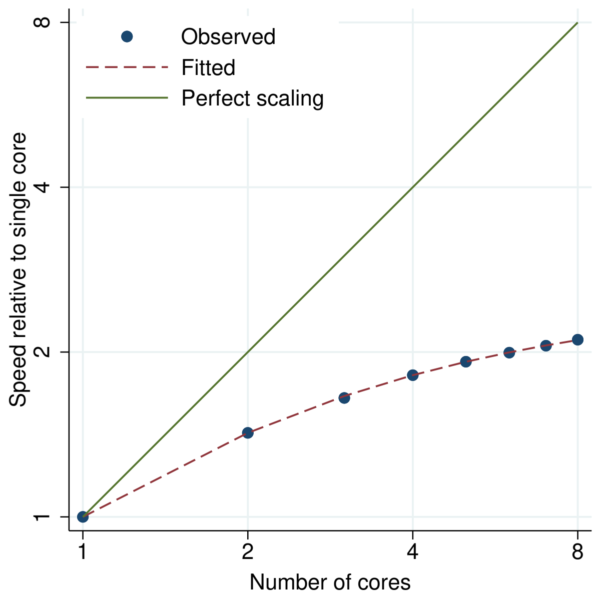


Figure 472. `stset` performance plot.

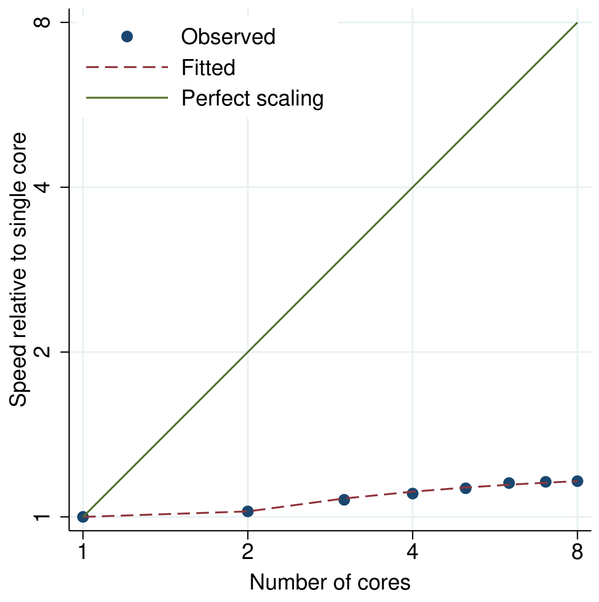


Figure 473. `stsplit` performance plot.

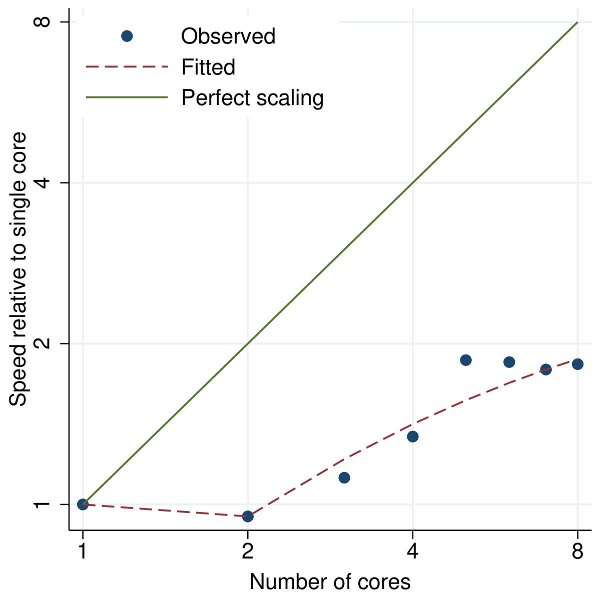


Figure 474. `stsum` performance plot.

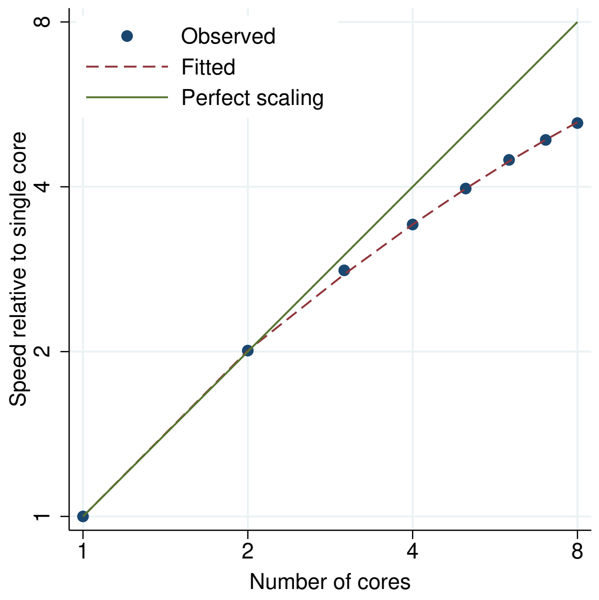


Figure 475. `stteffects ipw (weibull)` performance plot.

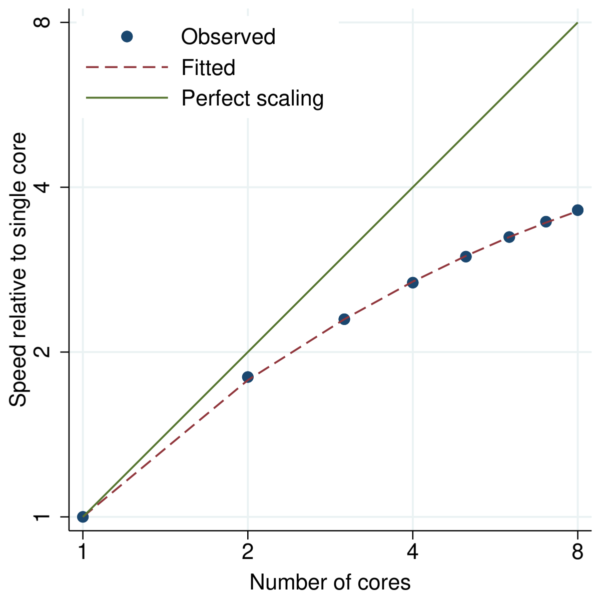


Figure 476. `stteffects ipwra (weibull)` performance plot.

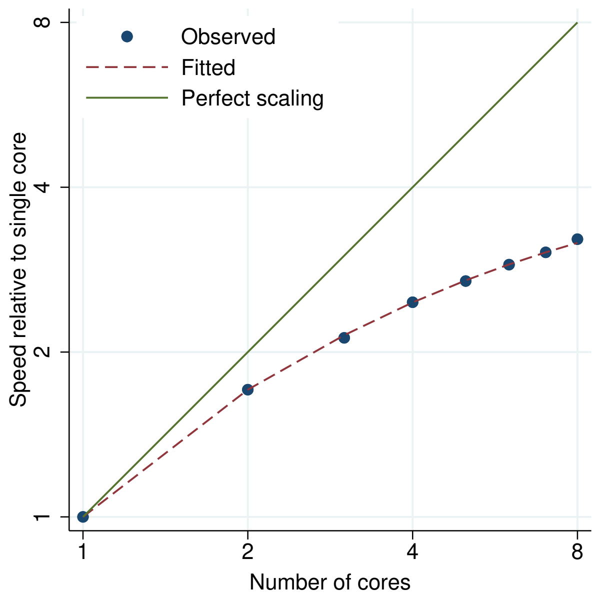


Figure 477. `stteffects ra (weibull)` performance plot.

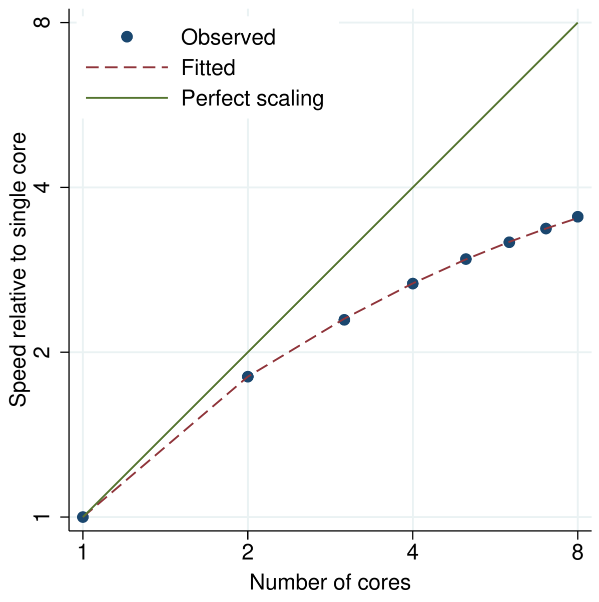


Figure 478. `stteffects wra (weibull)` performance plot.

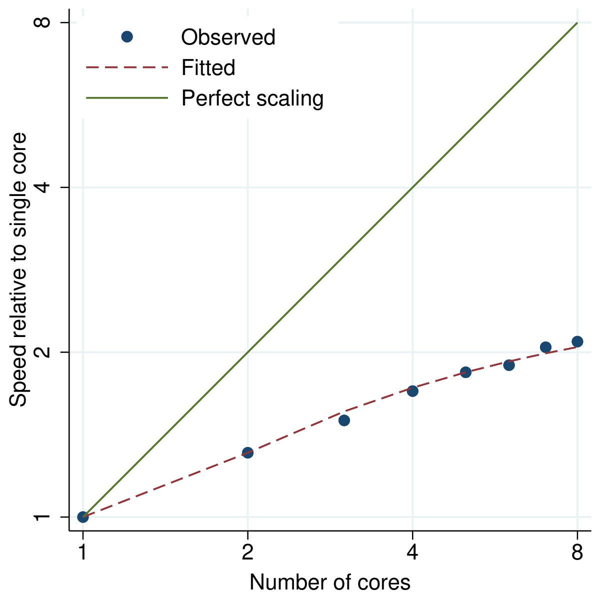


Figure 479. `stvary` performance plot.

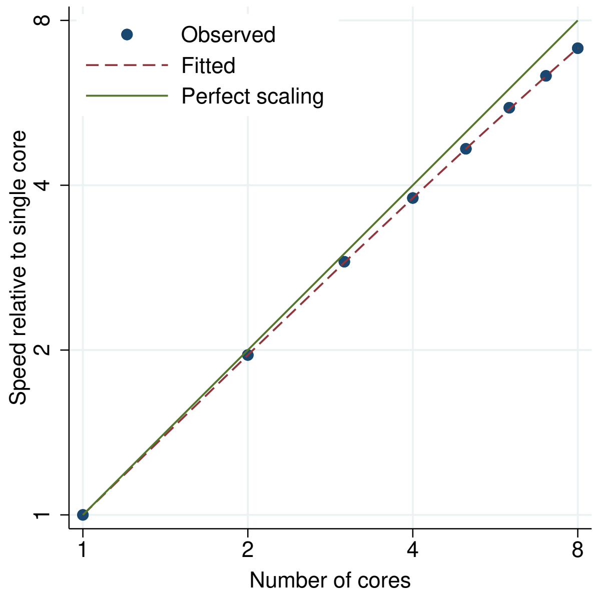


Figure 480. `suest` performance plot.

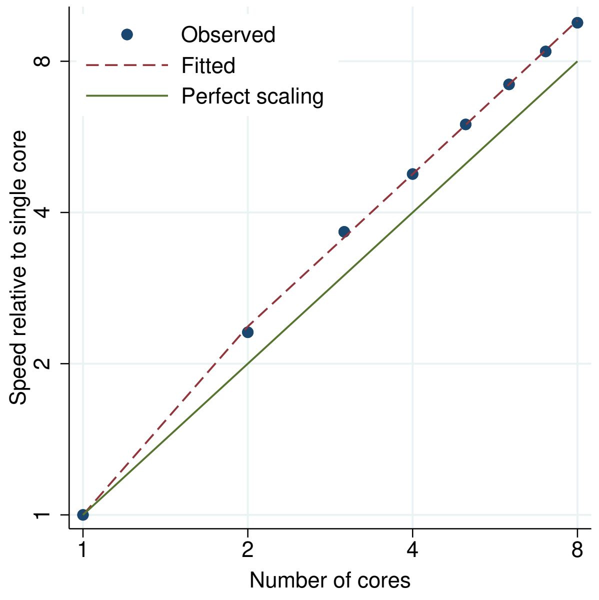


Figure 481. `summarize` performance plot.

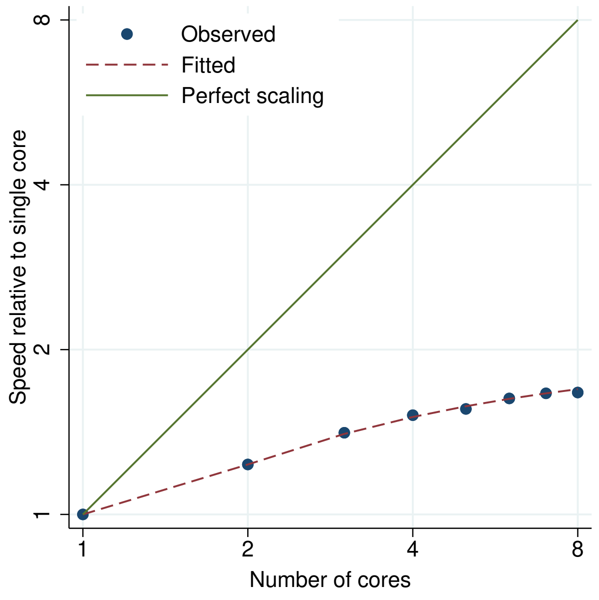


Figure 482. `sunflower` performance plot.

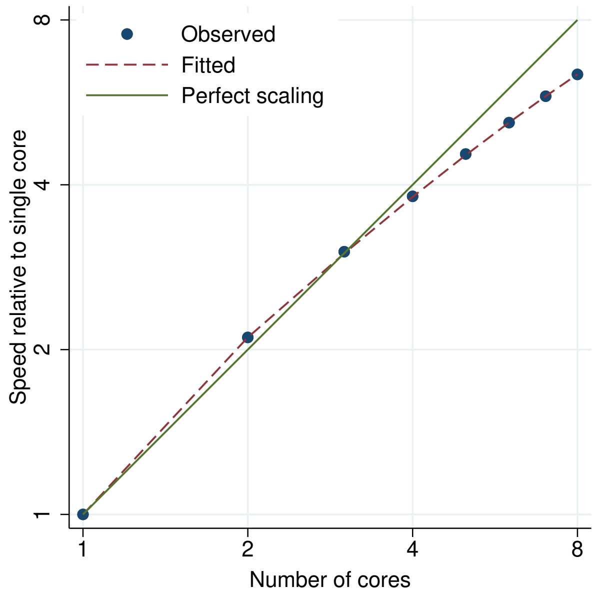


Figure 483. `sureg` performance plot.

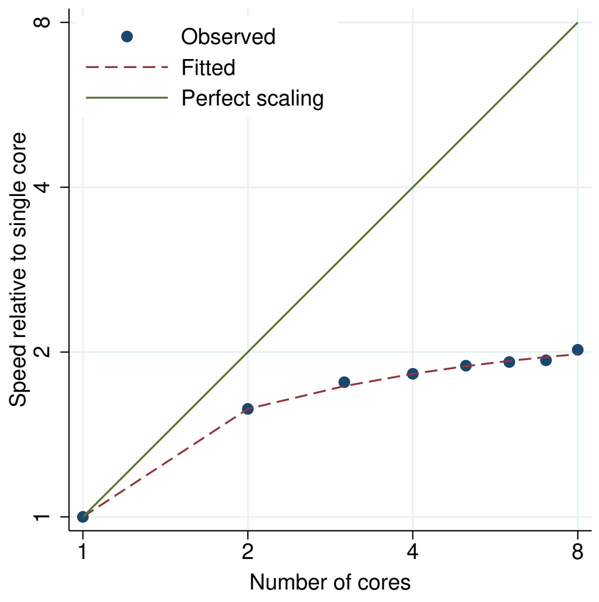


Figure 484. `svar` performance plot.

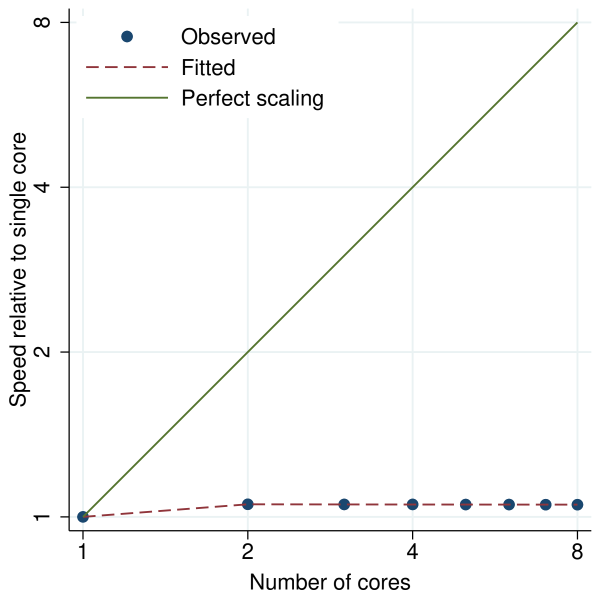


Figure 485. `svmat` performance plot.

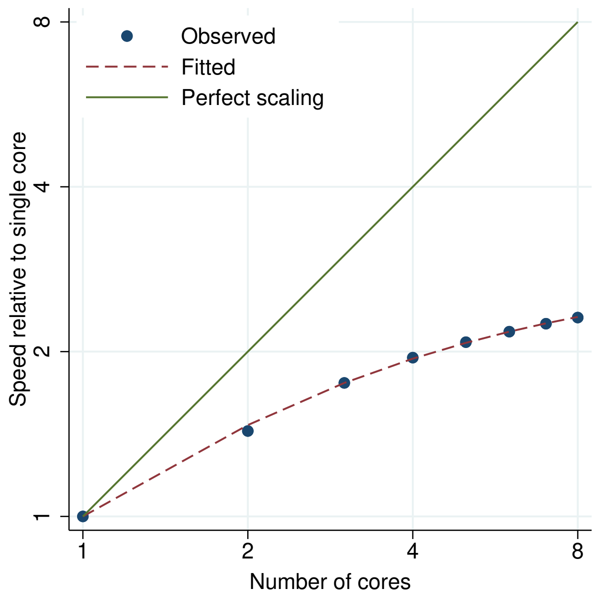


Figure 486. `svy brr: logit` performance plot.

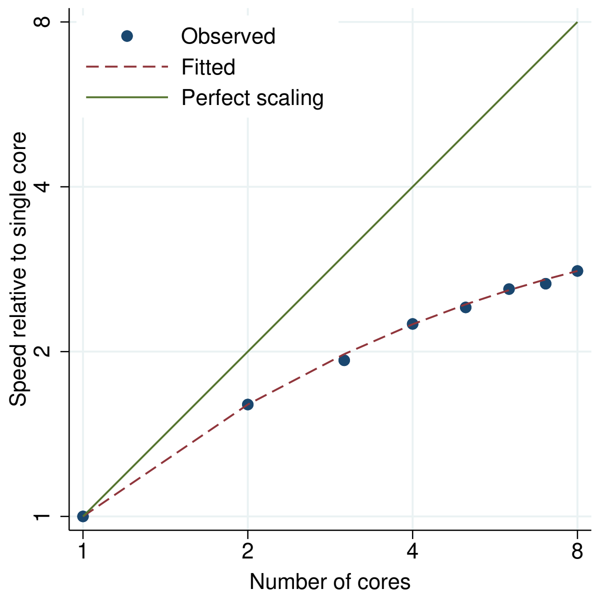


Figure 487. `svy brr: poisson` performance plot.

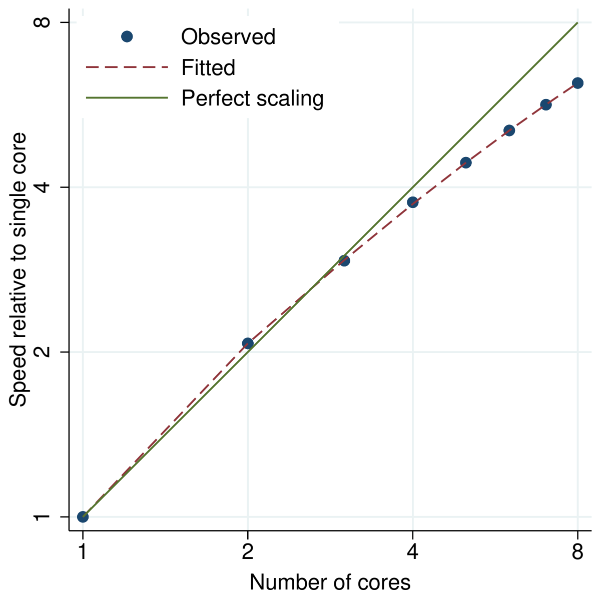


Figure 488. `svy brr: regress` performance plot.

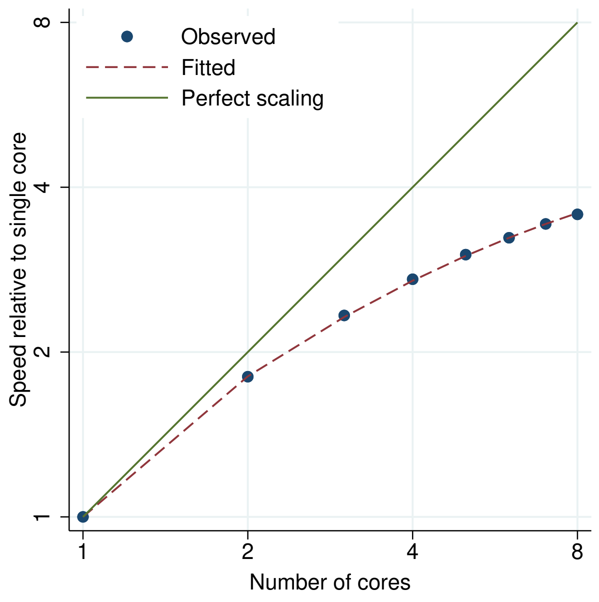


Figure 489. `svy jackknife: logit` performance plot.

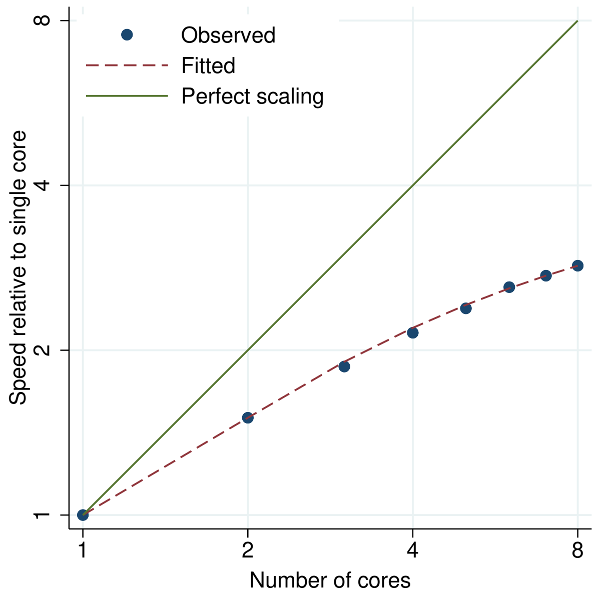


Figure 490. `svy jackknife: poisson` performance plot.

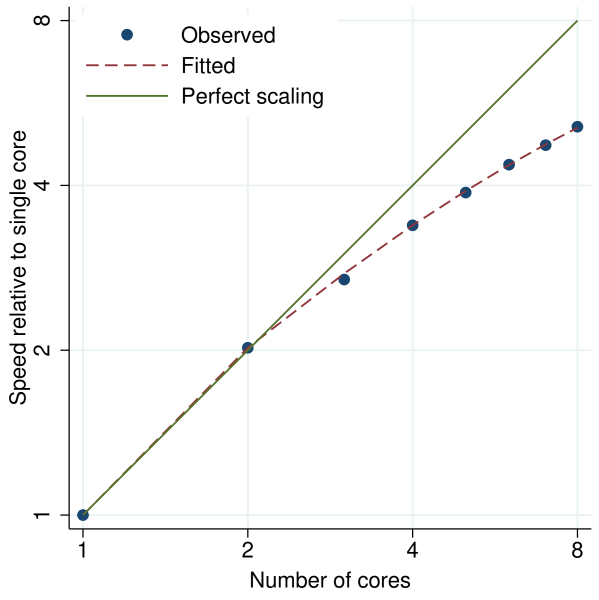


Figure 491. `svy jackknife: regress` performance plot.

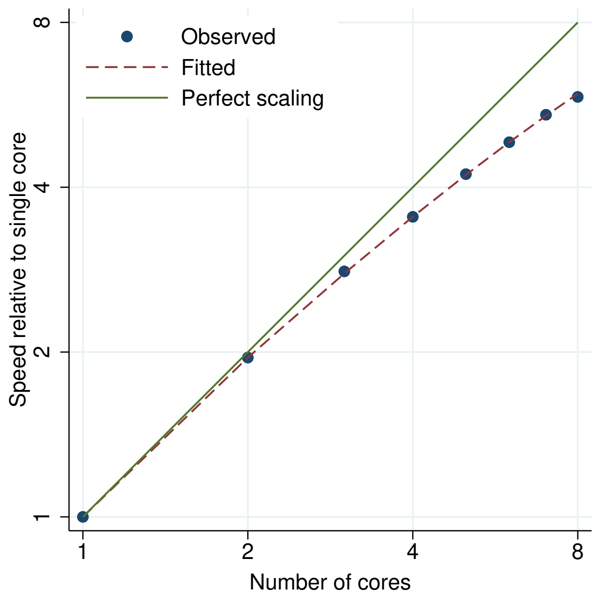


Figure 492. `svy linearized: logit` performance plot.

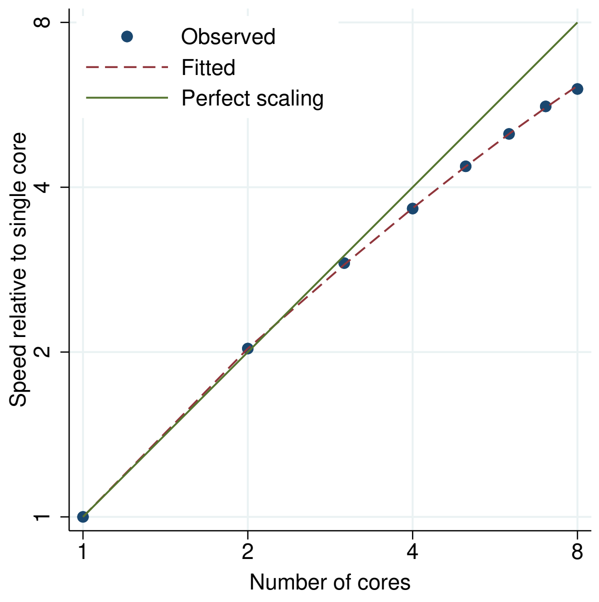


Figure 493. `svy linearized: poisson` performance plot.

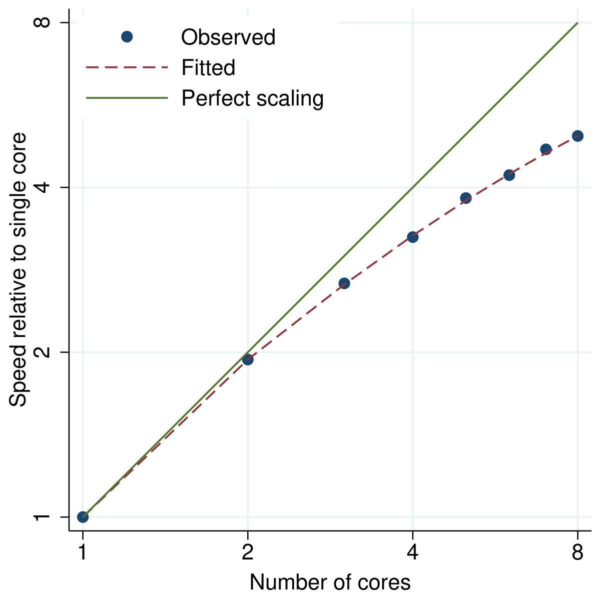


Figure 494. `svy linearized: regress` performance plot.

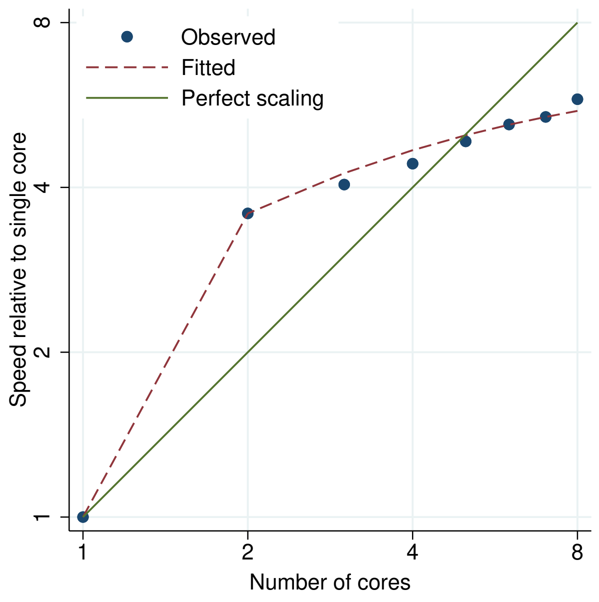


Figure 495. `swilk` performance plot.

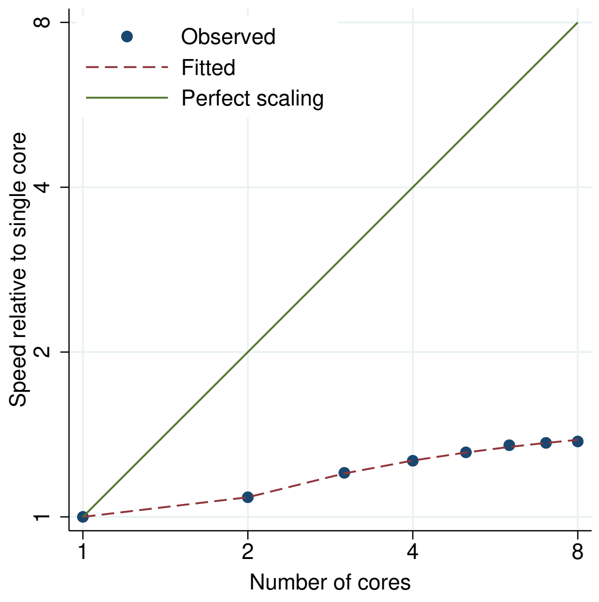


Figure 496. `symmetry` performance plot.

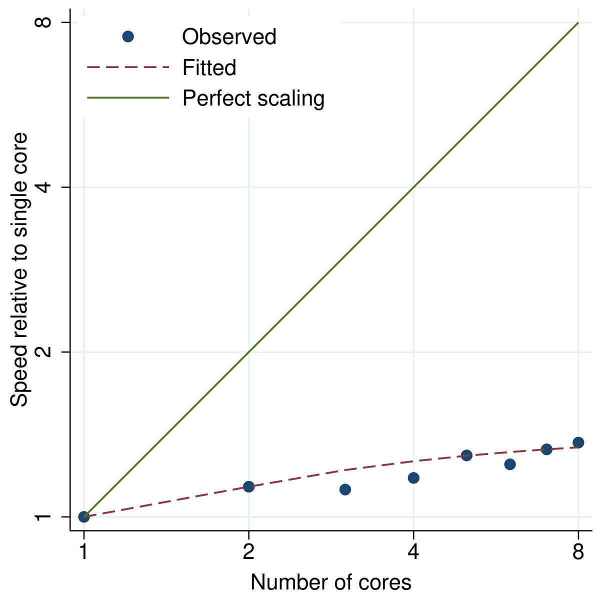


Figure 497. `table` (one-way) performance plot.

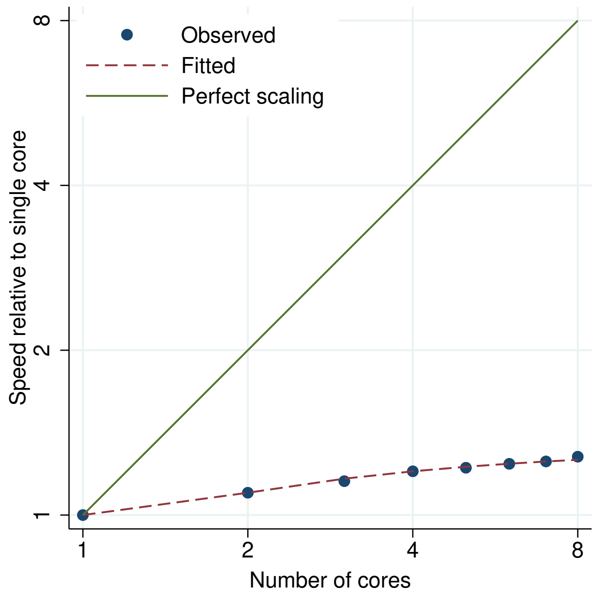


Figure 498. `table` (two-way) performance plot.

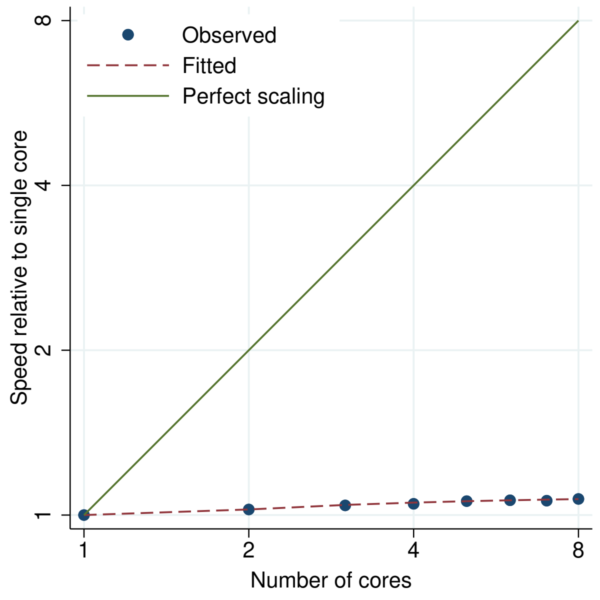


Figure 499. `tabodds` performance plot.

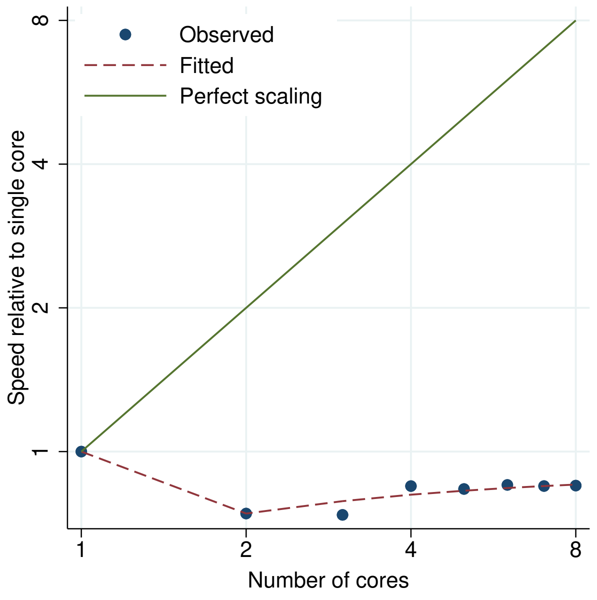


Figure 500. `tabodds` (adjusted) performance plot.

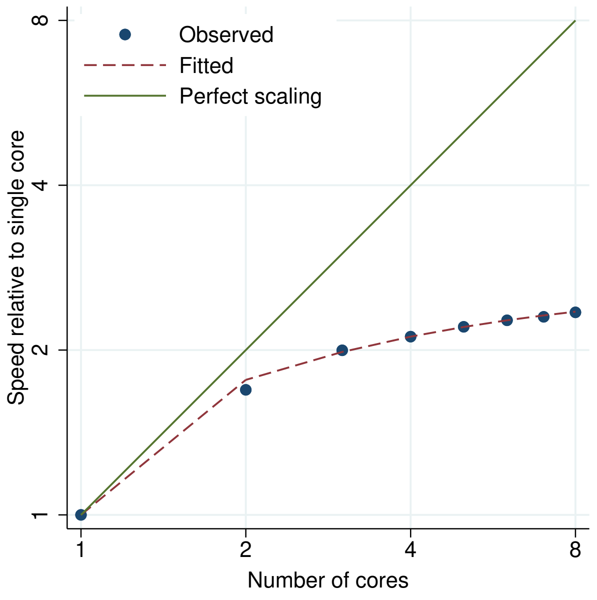


Figure 501. `tabstat` performance plot.

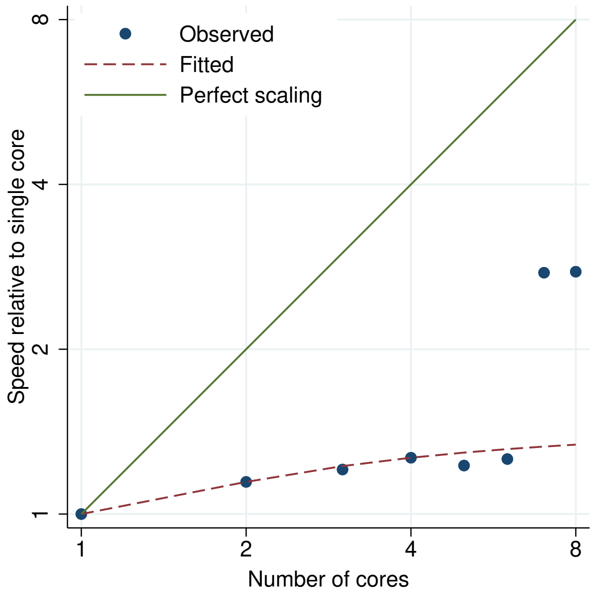


Figure 502. `tabstat, by()` performance plot.

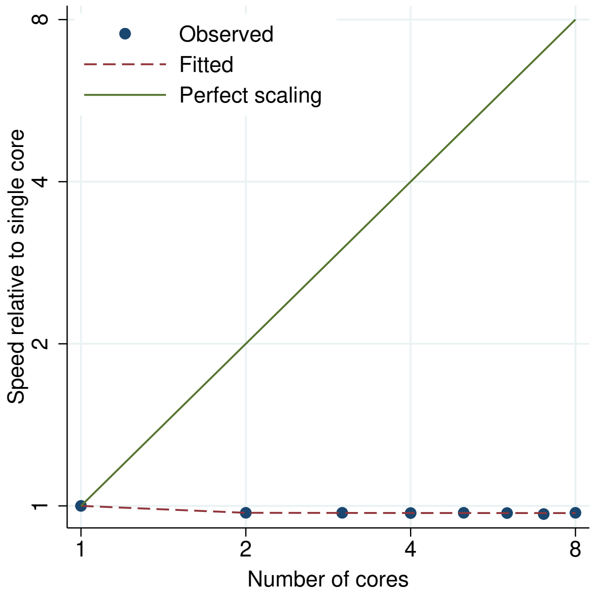


Figure 503. `tabulate` (one-way) performance plot.

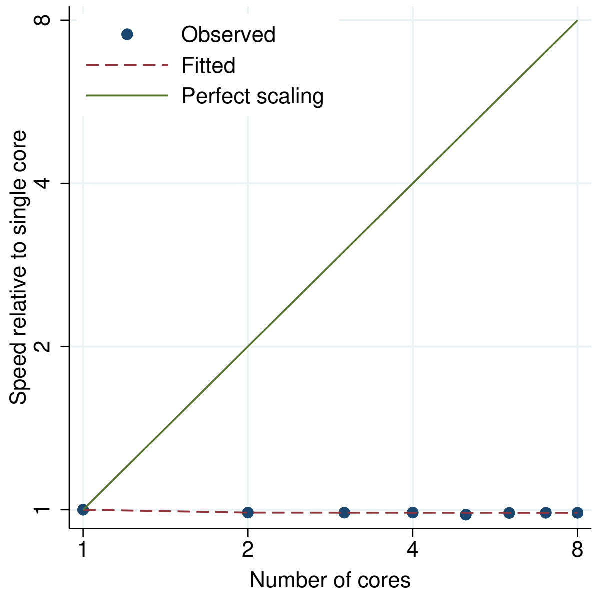


Figure 504. `tabulate` (two-way) performance plot.

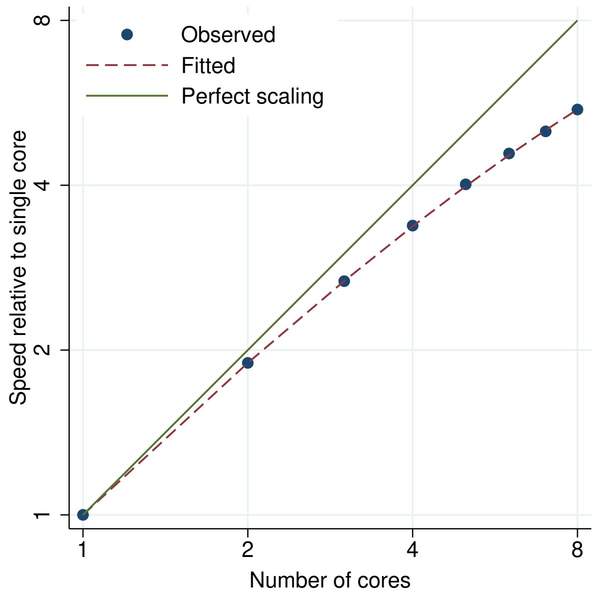


Figure 505. `teffects aipw (linear)` performance plot.

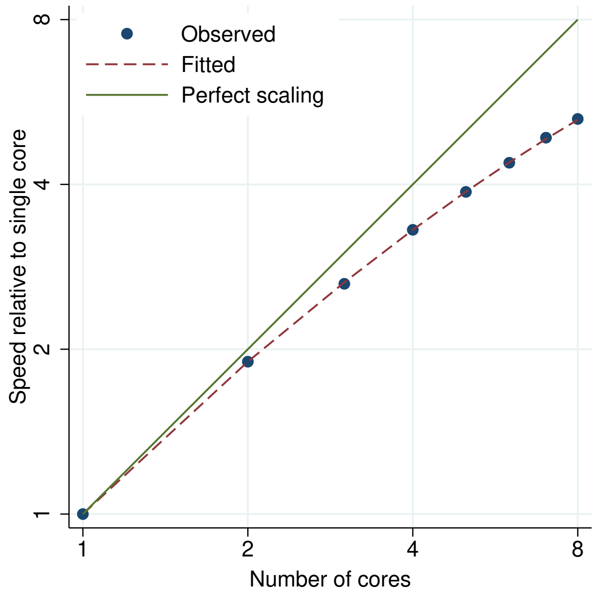


Figure 506. `teffects aipw (probit)` performance plot.

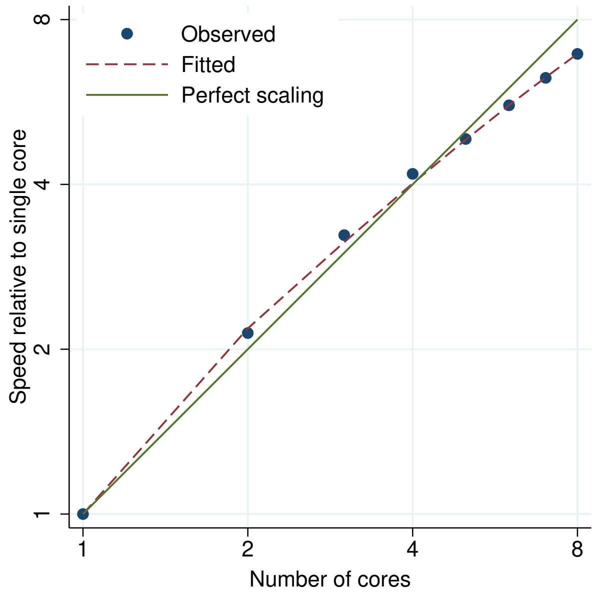


Figure 507. `teffects ipw (logit)` performance plot.

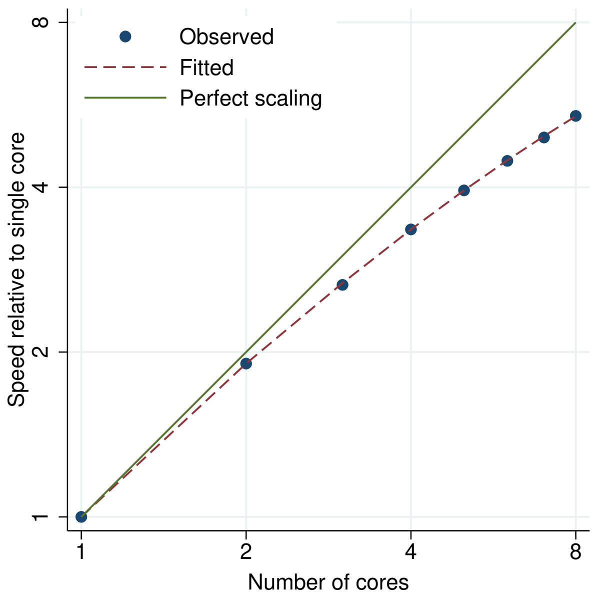


Figure 508. `teffects ipwra (linear)` performance plot.

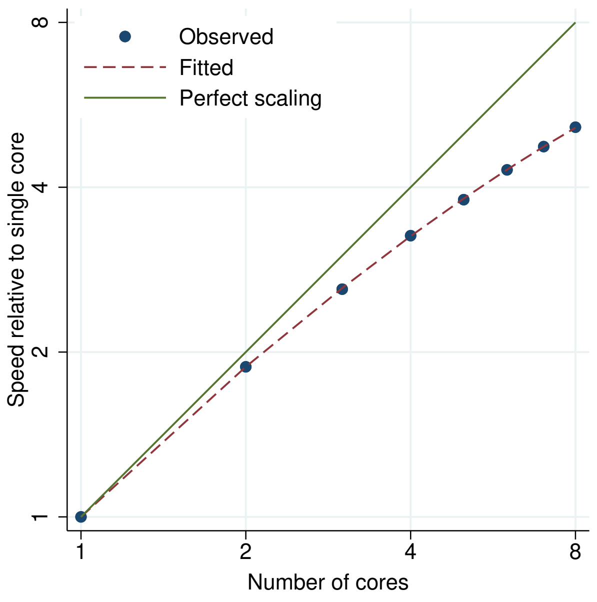


Figure 509. `teffects ipwra (probit)` performance plot.

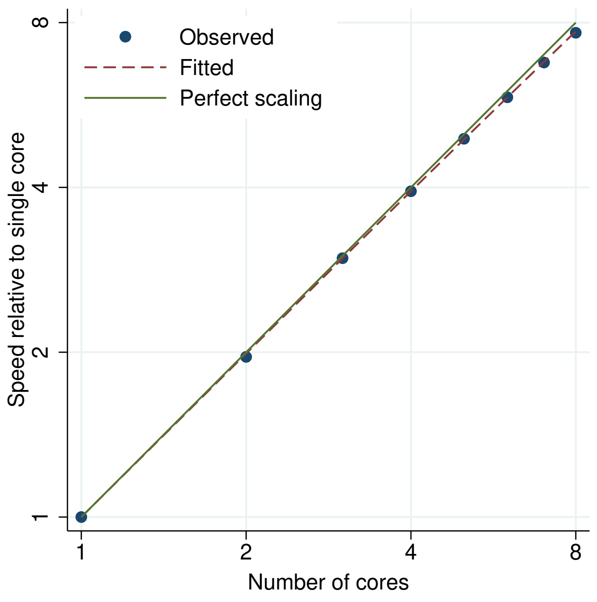


Figure 510. `teffects nnmatch` performance plot.

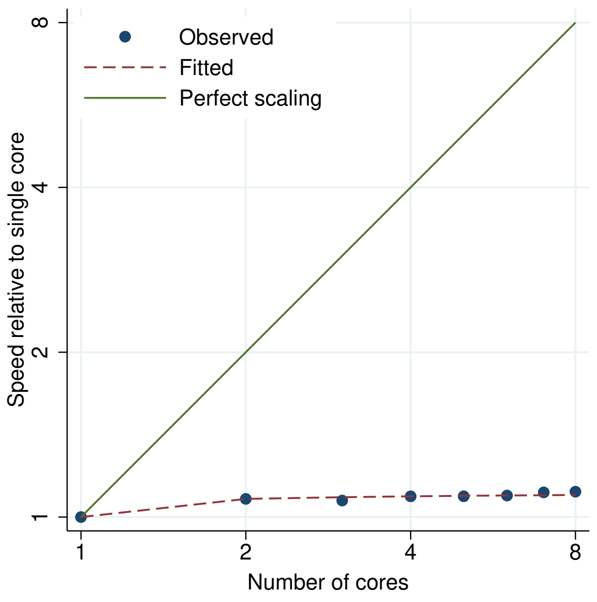


Figure 511. `teffects psmatch, logit` performance plot.

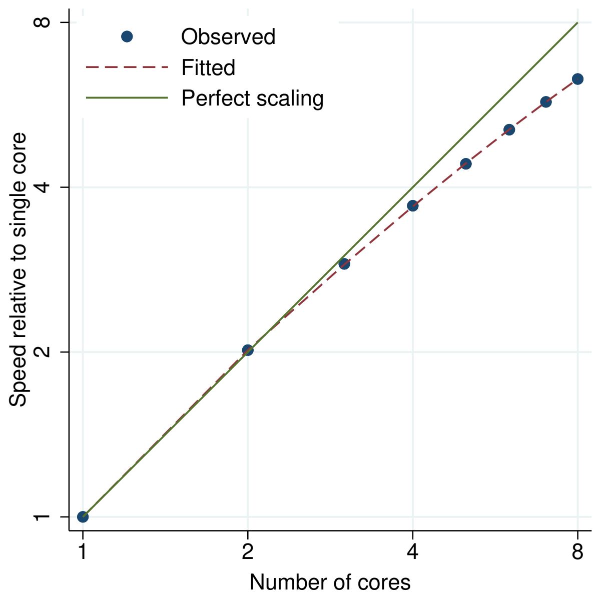


Figure 512. `teffects ra (linear)` performance plot.

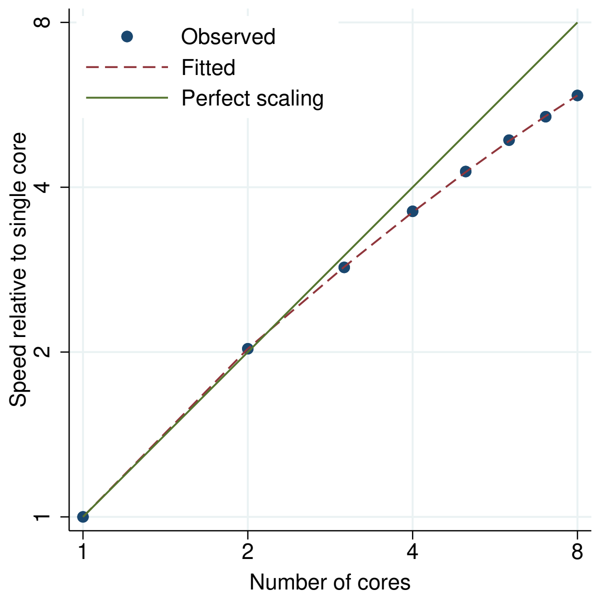


Figure 513. `teffects ra (probit)` performance plot.

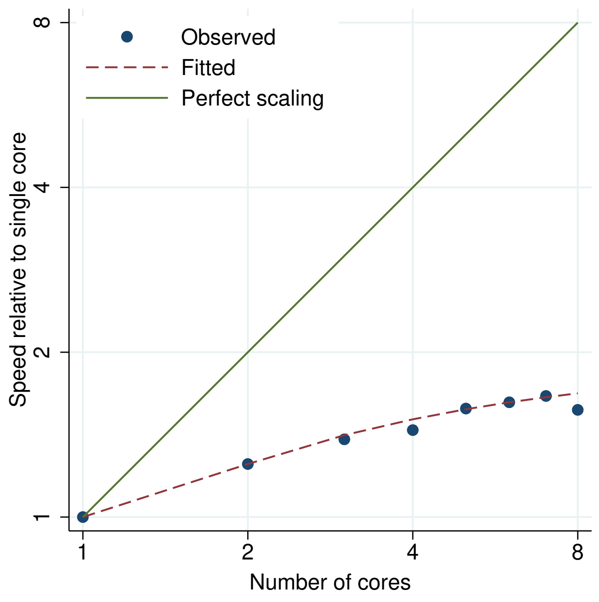


Figure 514. `telasso (, linear) (, probit), ate` performance plot.

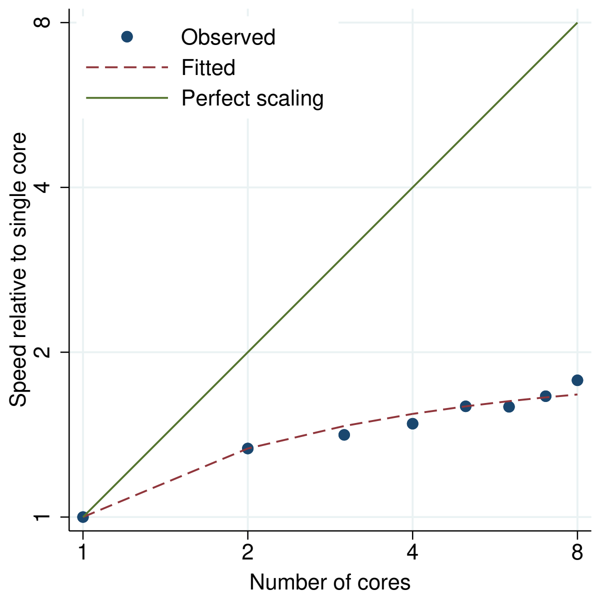


Figure 515. `telasso (, linear) (, probit), atet` performance plot.

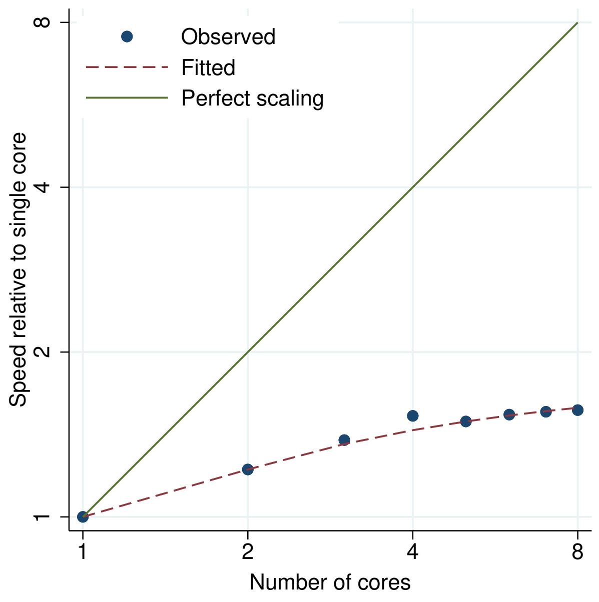


Figure 516. `telasso (, linear) (, probit), pomeans` performance plot.

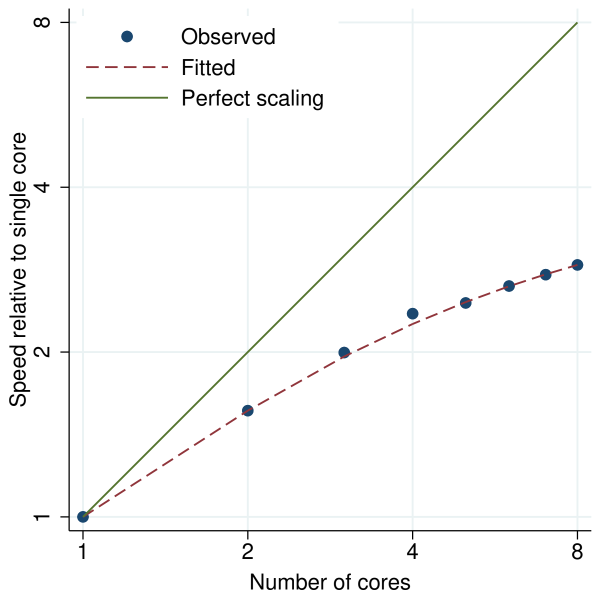


Figure 517. `telasso (, logit) (, probit), ate` performance plot.

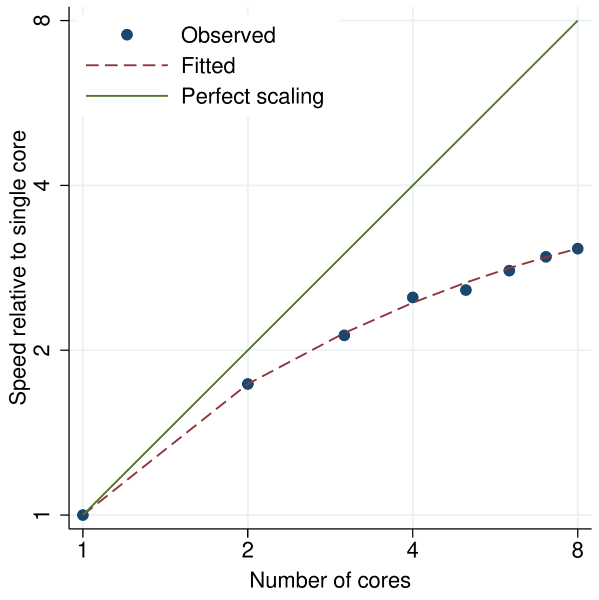


Figure 518. `telasso (, logit) (, probit), atet` performance plot.

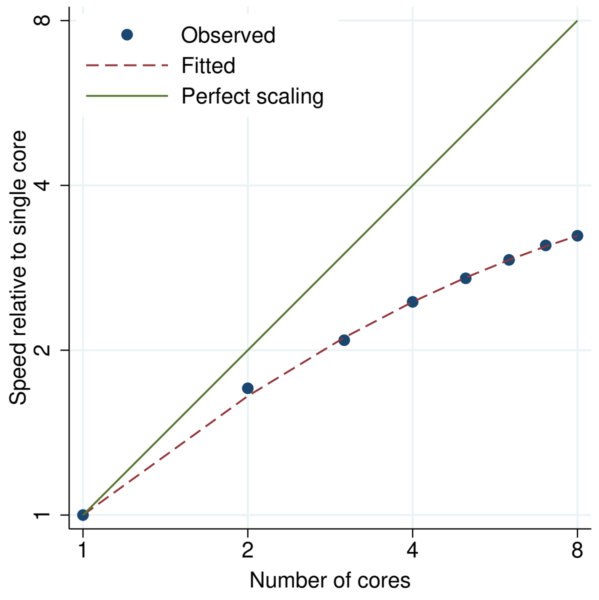


Figure 519. `telasso (, logit) (, probit), pomeans` performance plot.

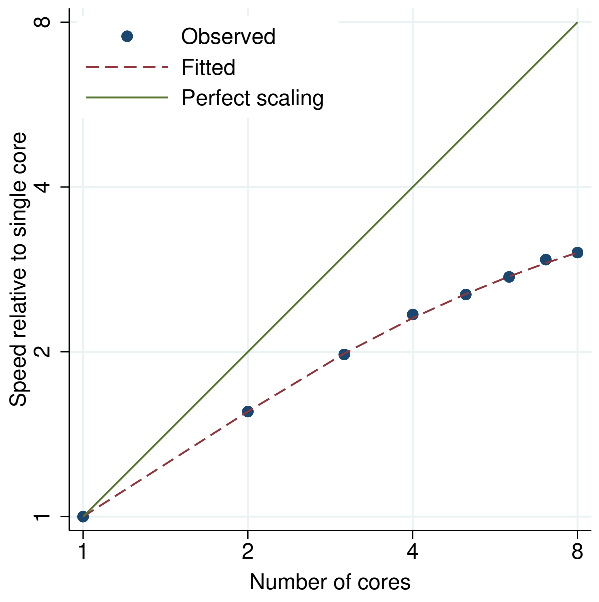


Figure 520. `telasso (, poisson) (, probit), ate` performance plot.

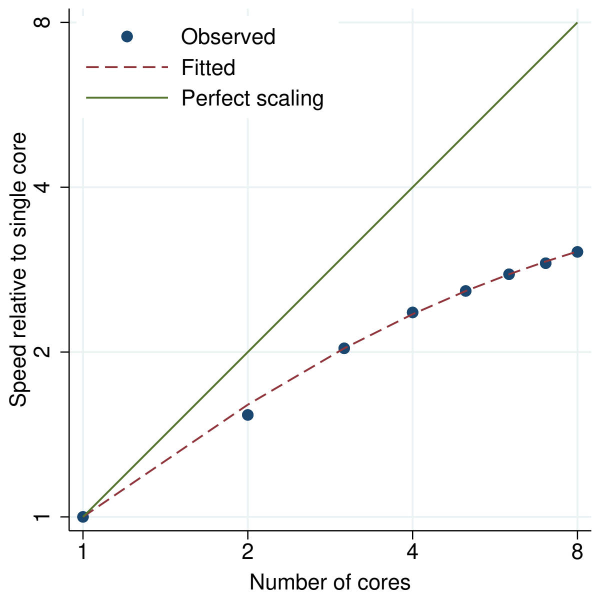


Figure 521. `telasso (, poisson) (, probit), atet` performance plot.

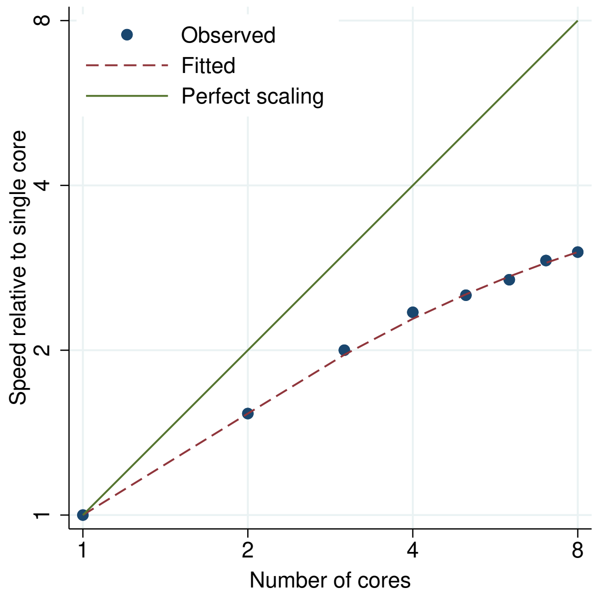


Figure 522. `telasso (, poisson) (, probit), pomeans` performance plot.

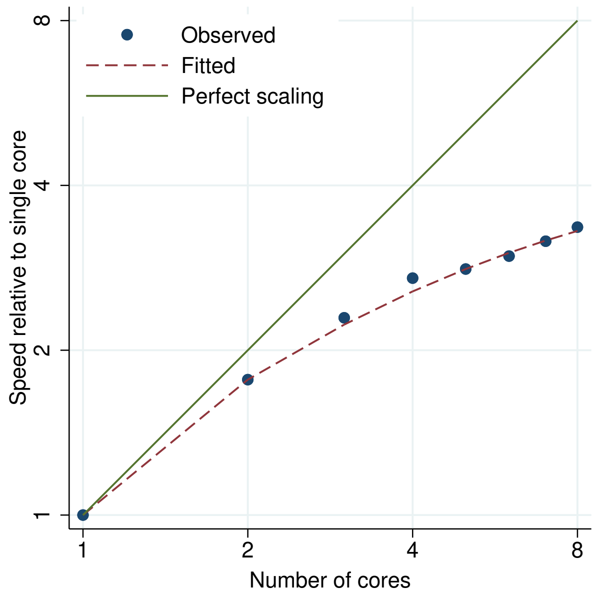


Figure 523. `telasso (, probit) (, probit), ate` performance plot.

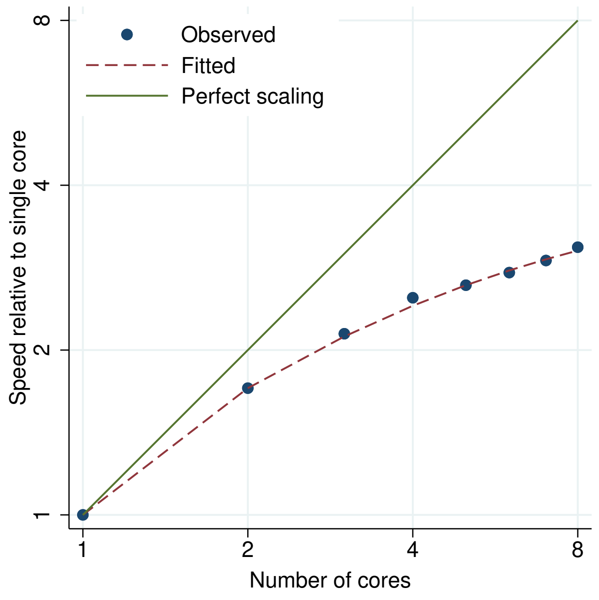


Figure 524. `telasso (, probit) (, probit), atet` performance plot.

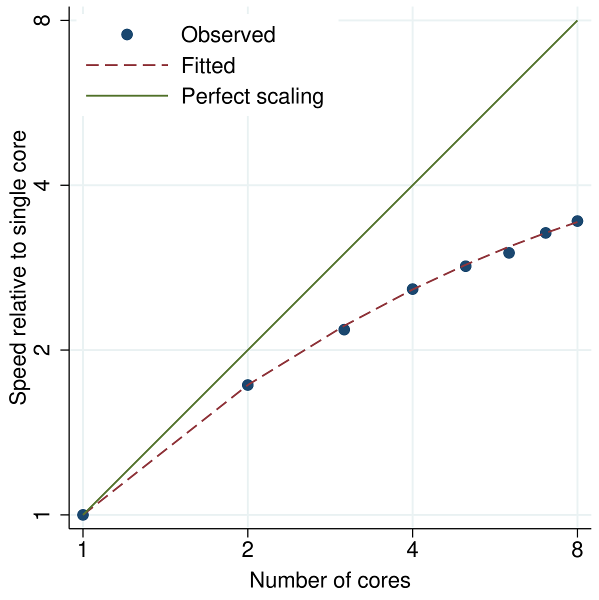


Figure 525. `telasso (, probit) (, probit), pomeans` performance plot.

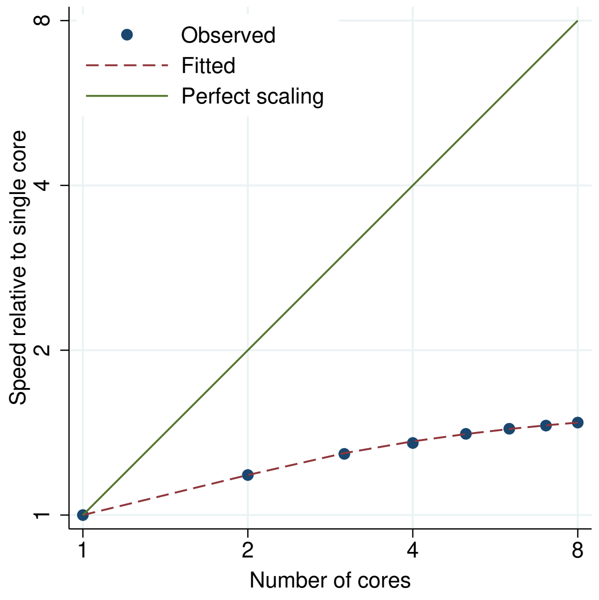


Figure 526. `tetrachoric` performance plot.

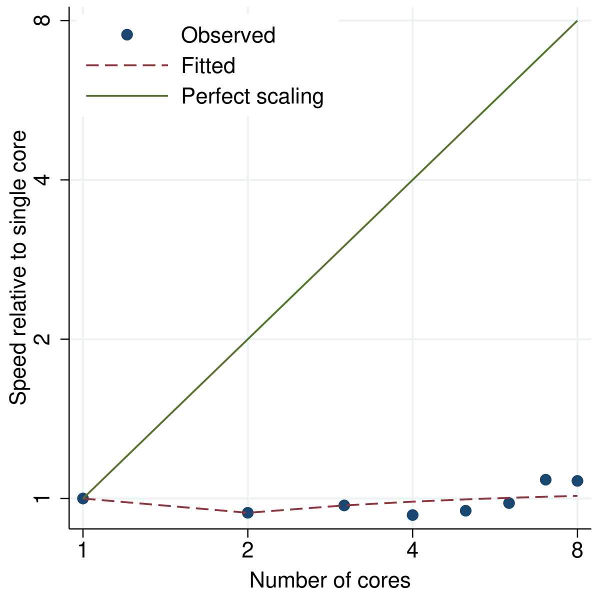


Figure 527. `threshold, threshvar()` performance plot.

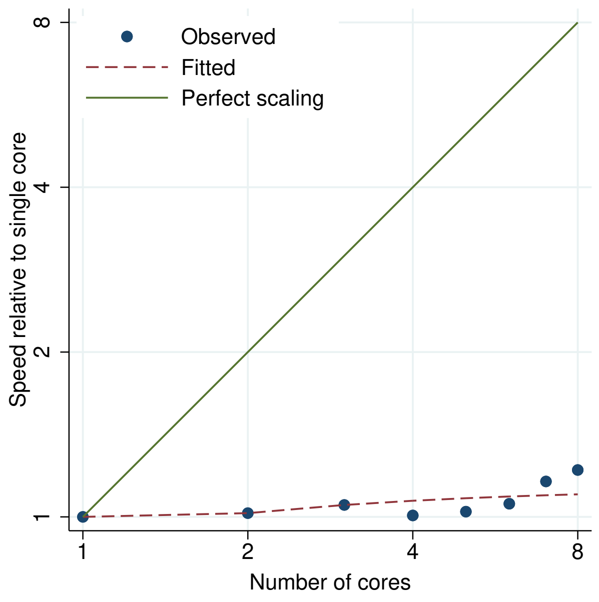


Figure 528. `threshold, threshvar() regionvars()` performance plot.

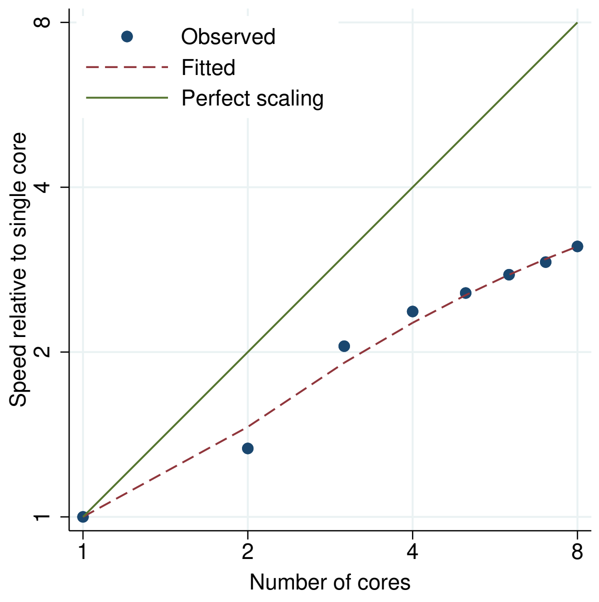


Figure 529. `tnbreg` performance plot.

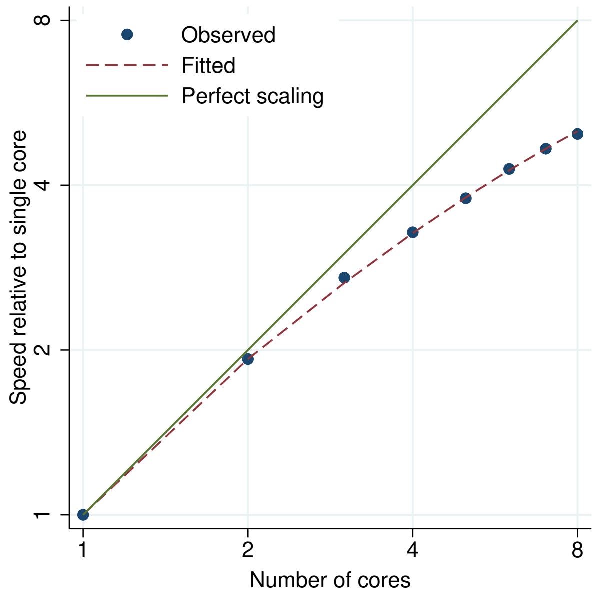


Figure 530. `tobit` performance plot.

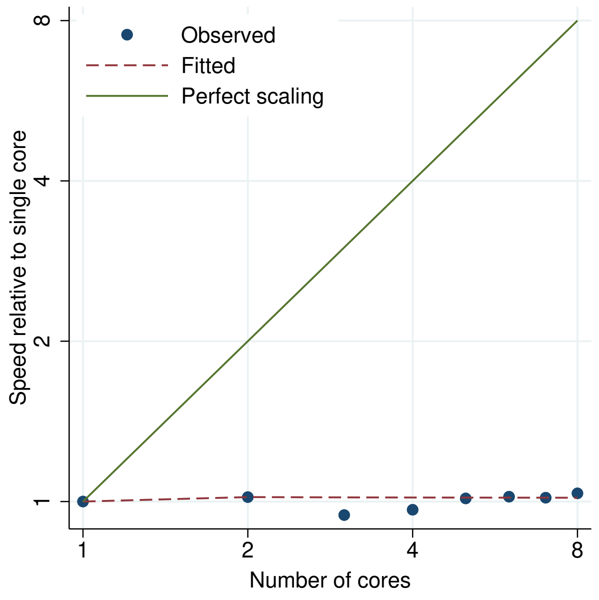


Figure 531. `tostring` performance plot.

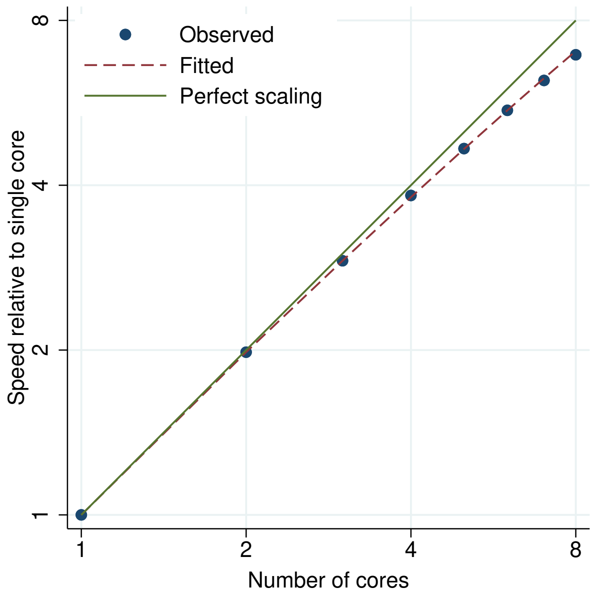


Figure 532. `total` performance plot.

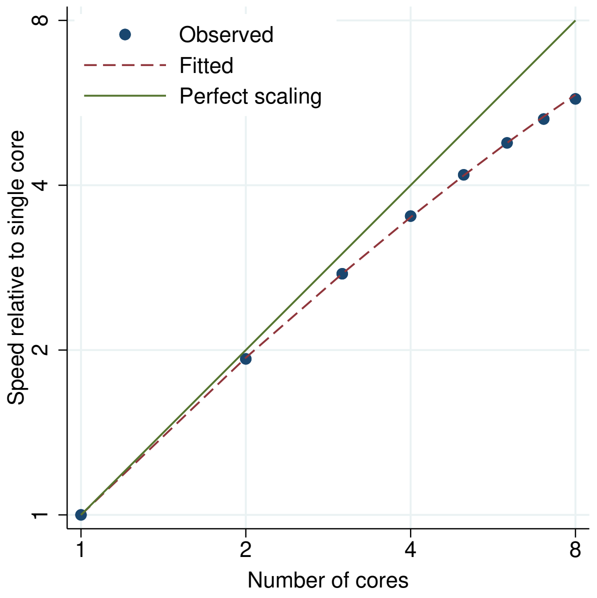


Figure 533. `tpoisson` performance plot.

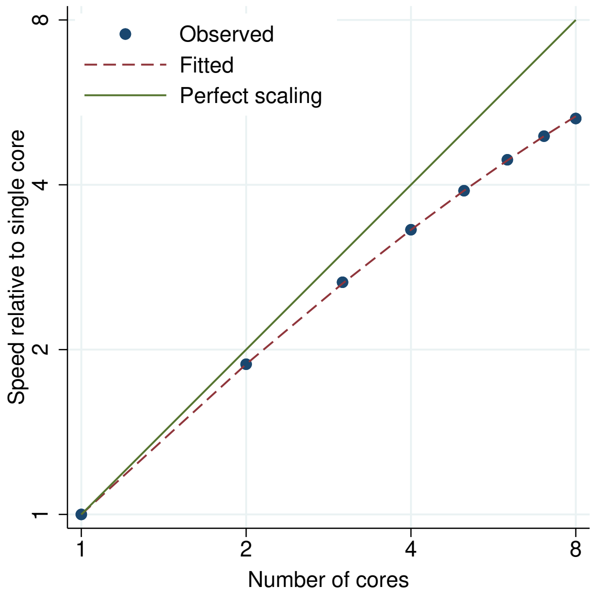


Figure 534. `truncreg` performance plot.

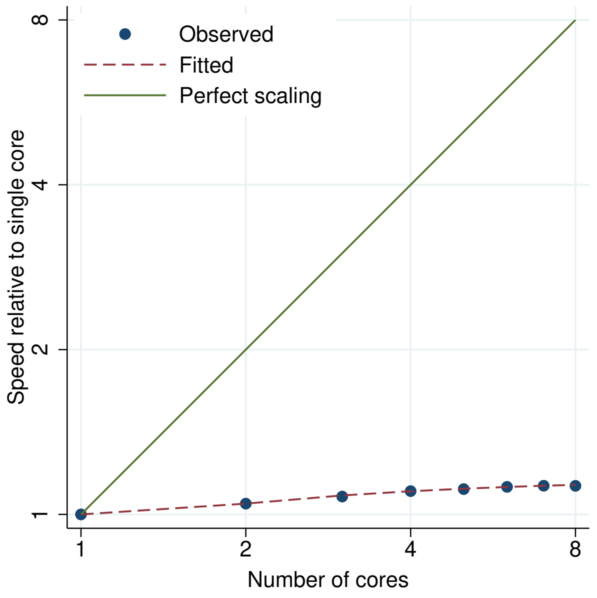


Figure 535. `tsfilter bk` performance plot.

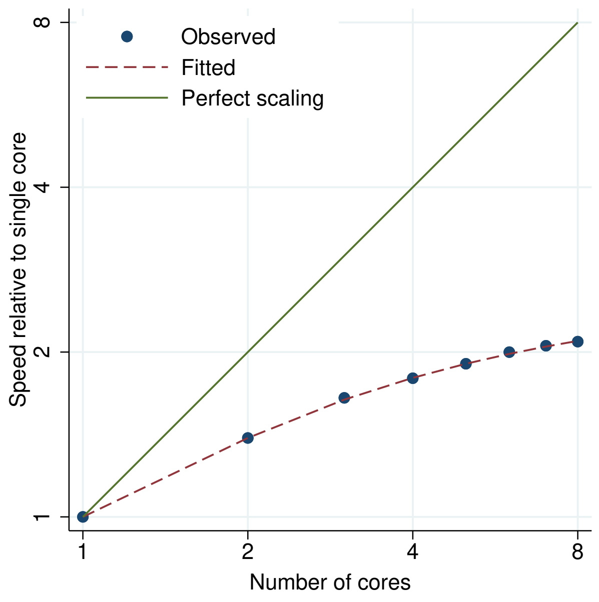


Figure 536. `tsfilter bw` performance plot.

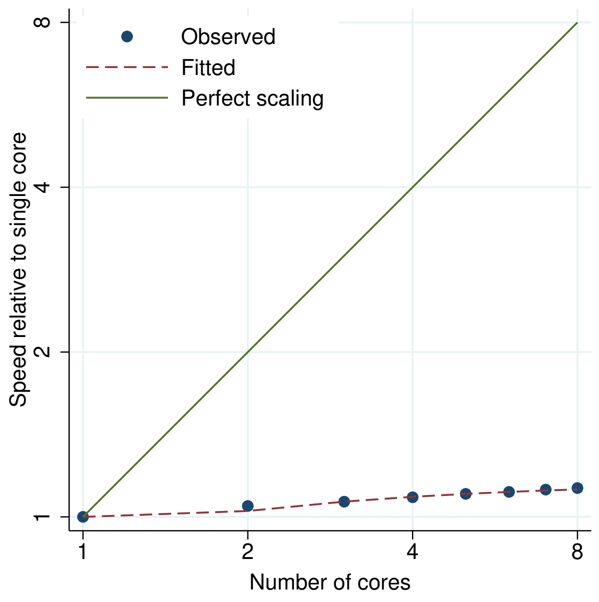


Figure 537. `tsfilter cf` performance plot.

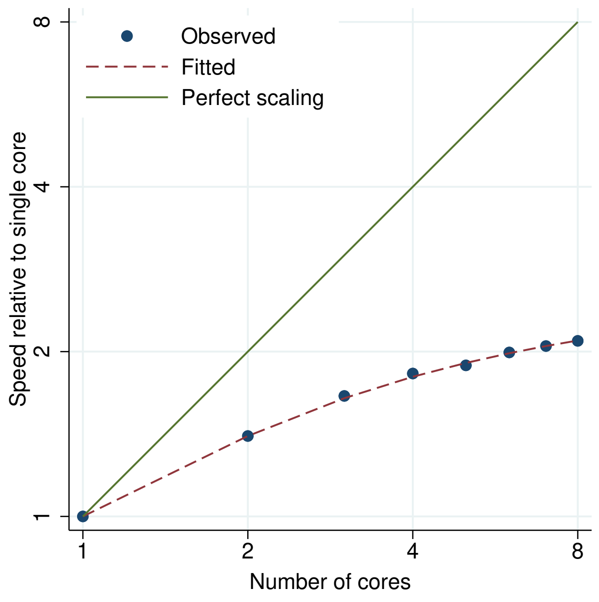


Figure 538. `tsfilter hp` performance plot.

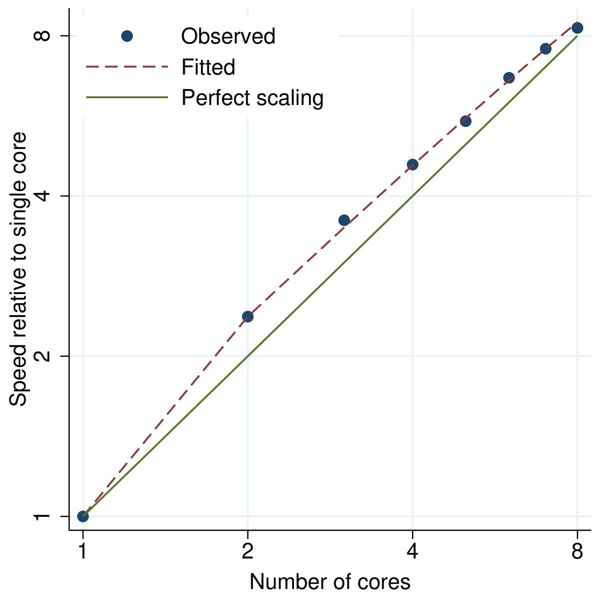


Figure 539. `tsrevar` performance plot.

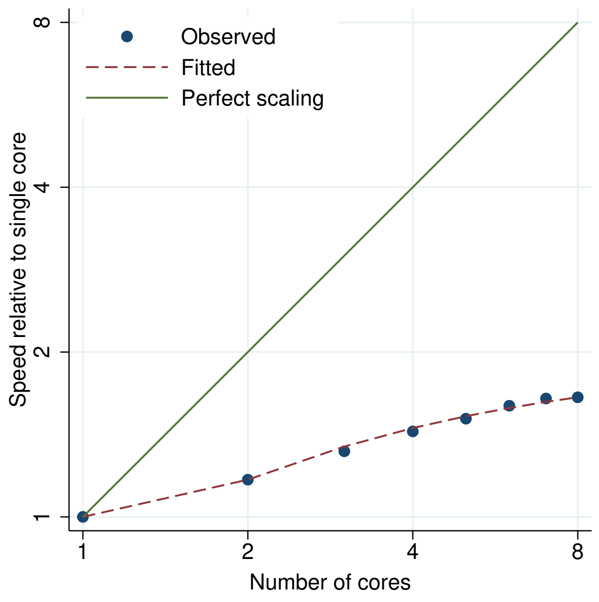


Figure 540. `tsset` performance plot.

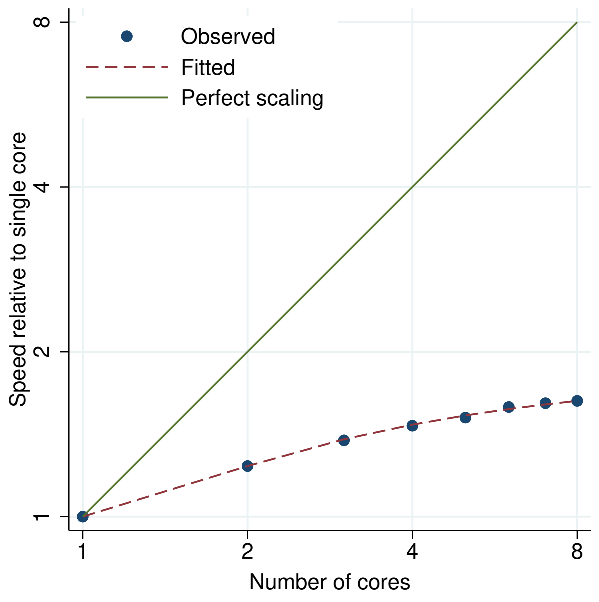


Figure 541. `tssmooth exp` performance plot.

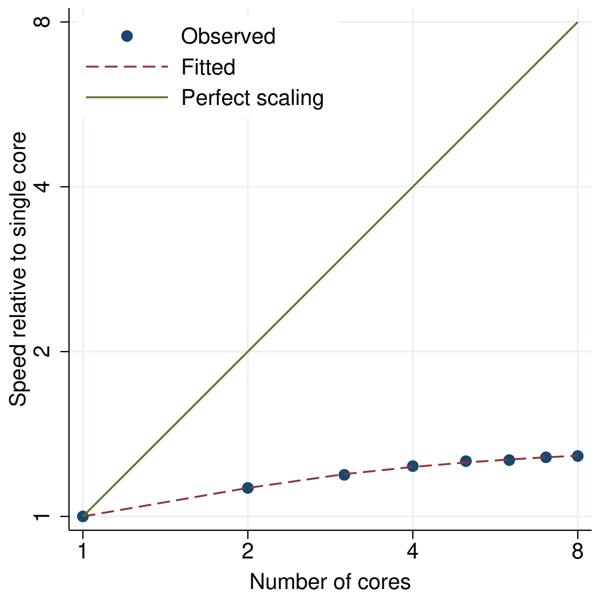


Figure 542. `tssmooth ma` performance plot.

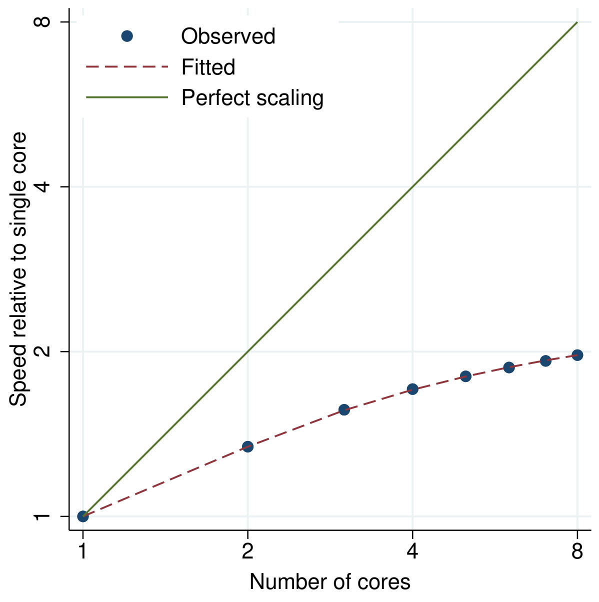


Figure 543. `ttest1` performance plot.

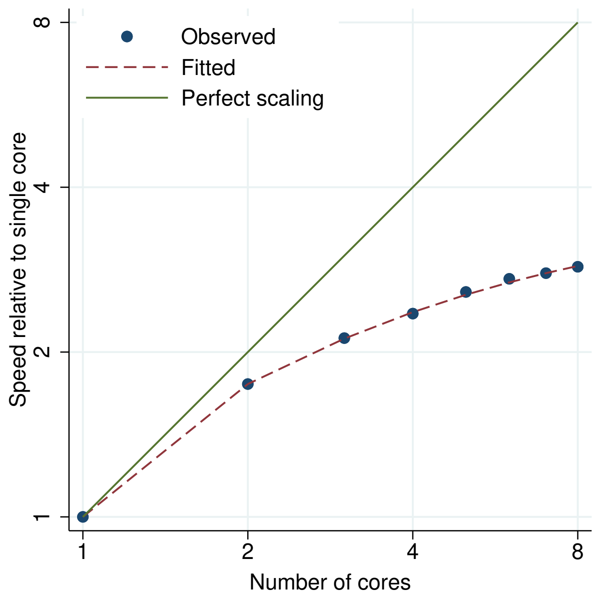


Figure 544. `ttest2` performance plot.

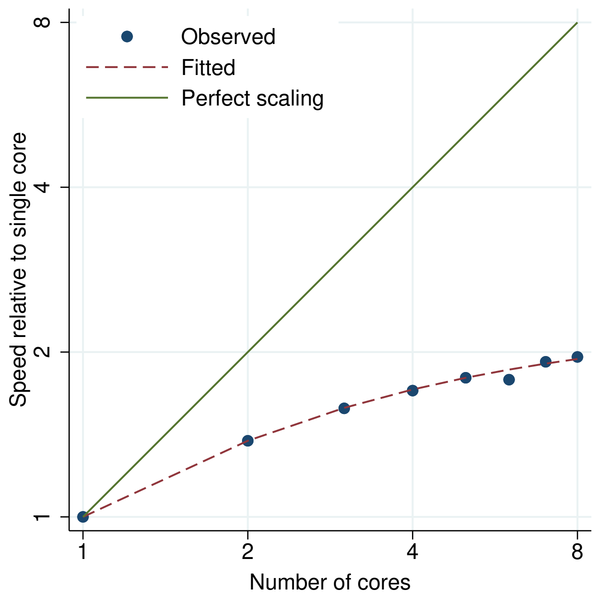


Figure 545. `ttest, by()` performance plot.

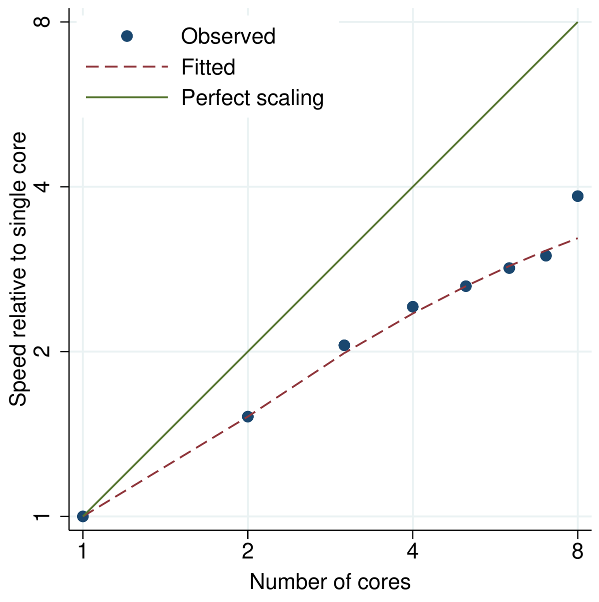


Figure 546. `twoway fpfit` performance plot.

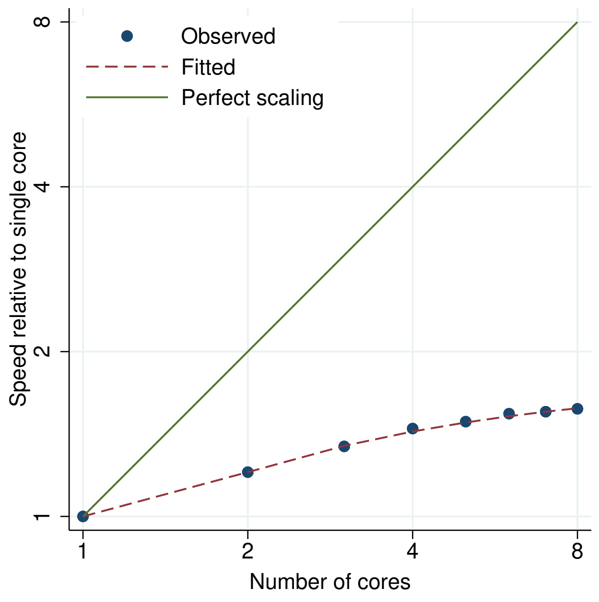


Figure 547. `twoway lfitci` performance plot.

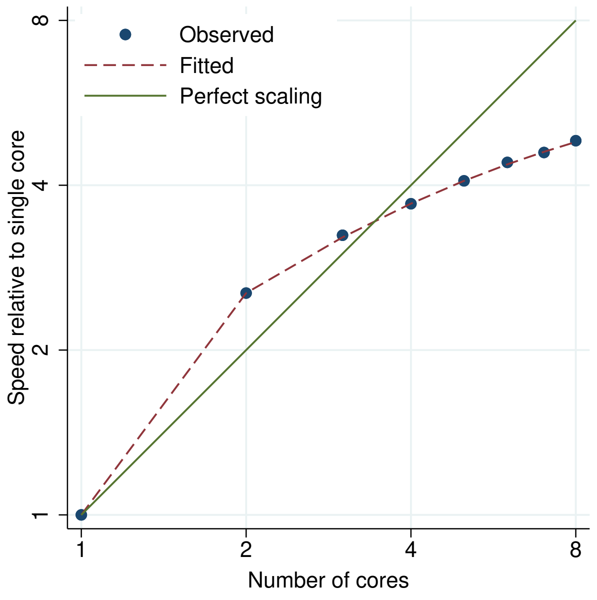


Figure 548. `twoway mband` performance plot.

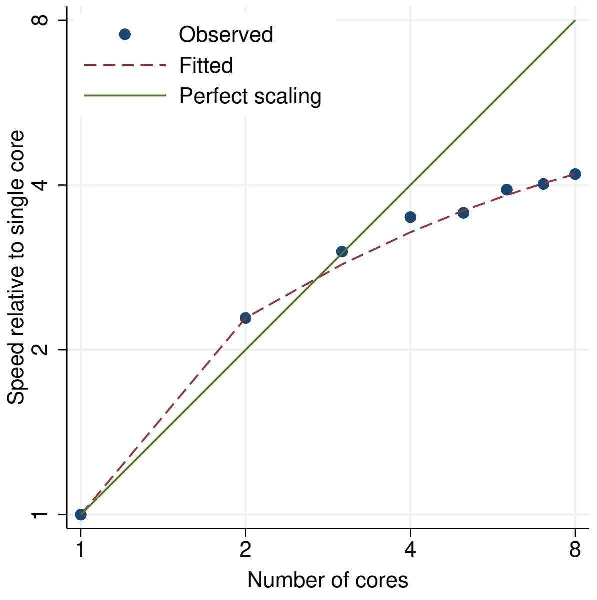


Figure 549. `twoway mspline` performance plot.

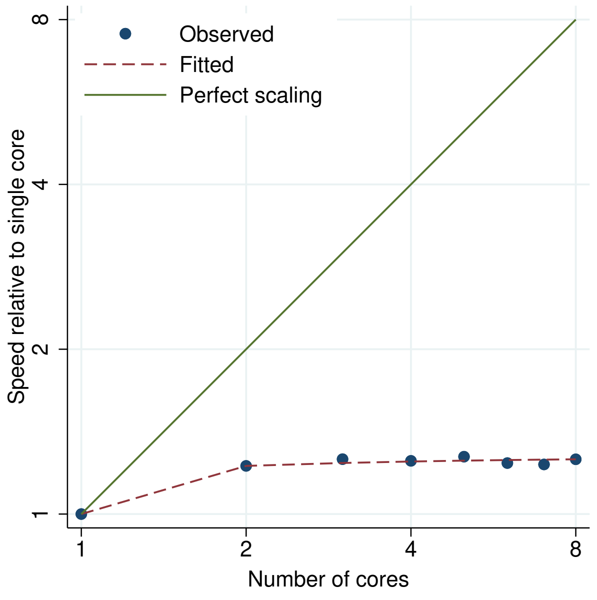


Figure 550. `ucm, model(rwdrift)` performance plot.

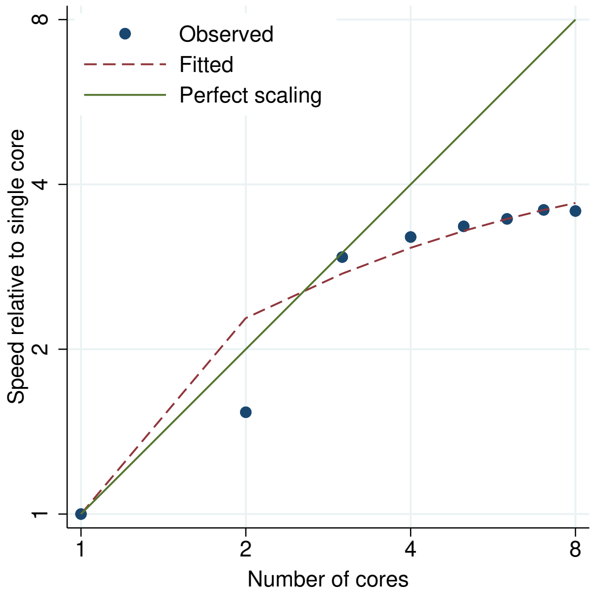


Figure 551. `var` performance plot.

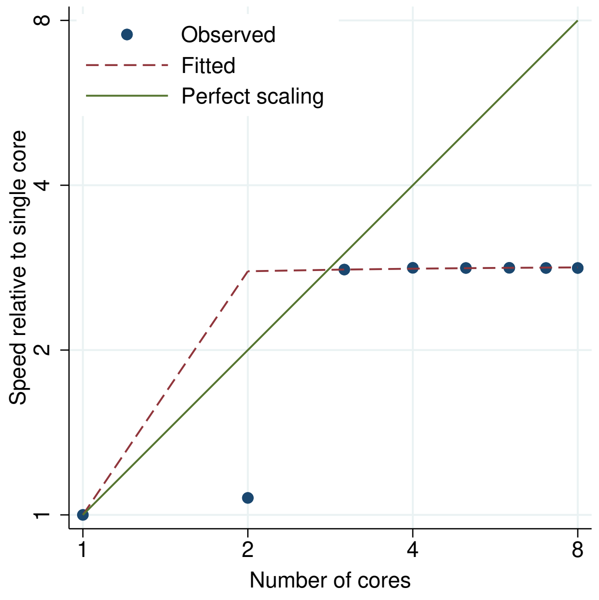


Figure 552. `vargranger` performance plot.

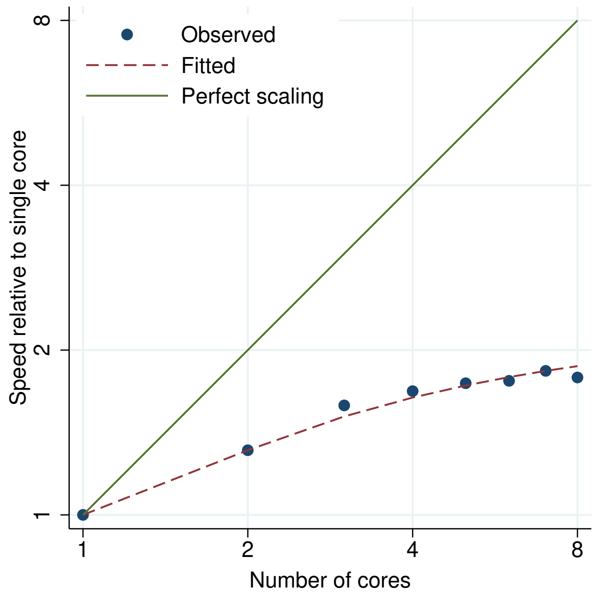


Figure 553. `varlmar` performance plot.

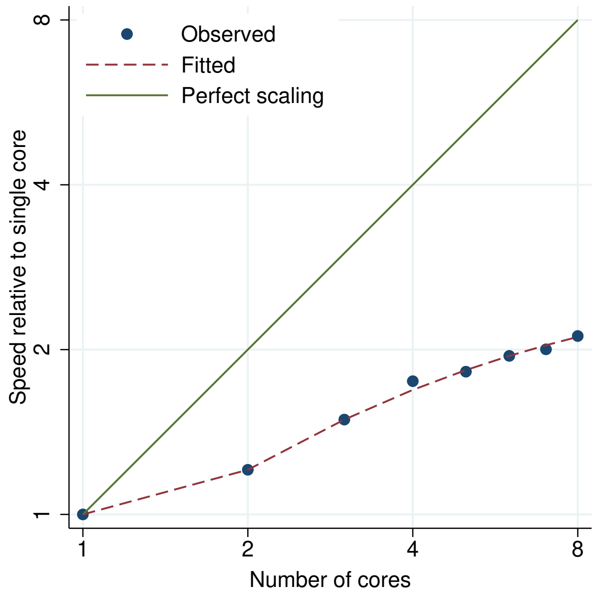


Figure 554. `varnorm` performance plot.

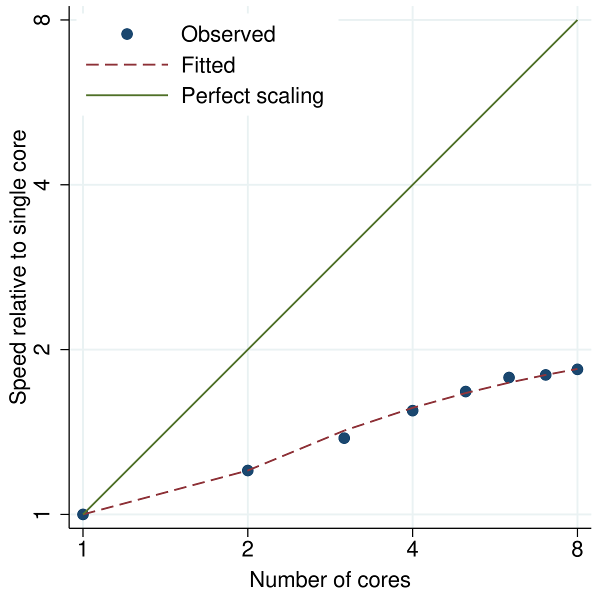


Figure 555. `varsoc` performance plot.

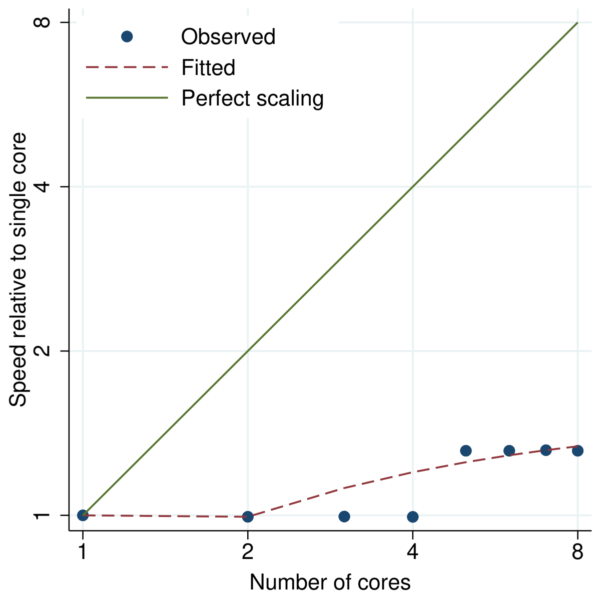


Figure 556. `varstable` performance plot.

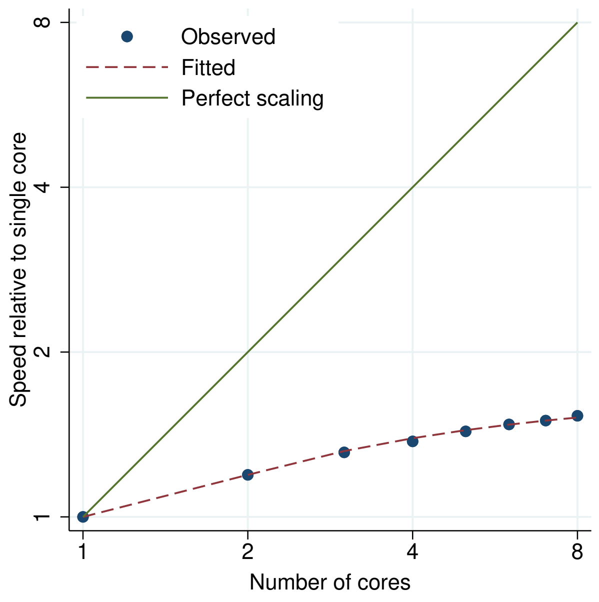


Figure 557. `vec` performance plot.

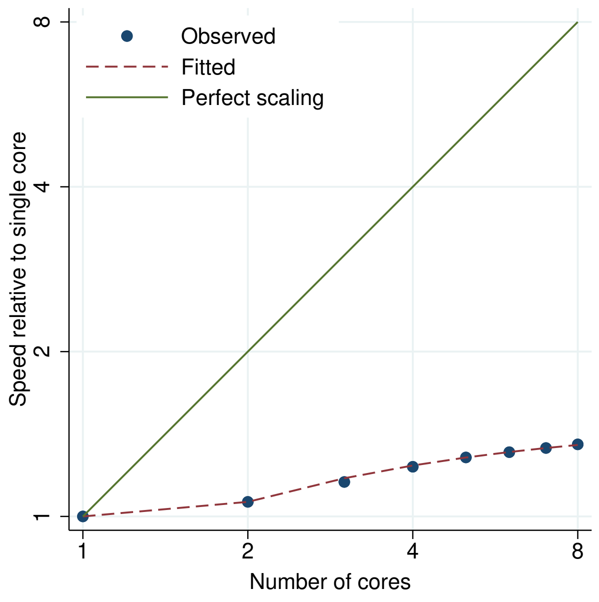


Figure 558. `veclmar` performance plot.

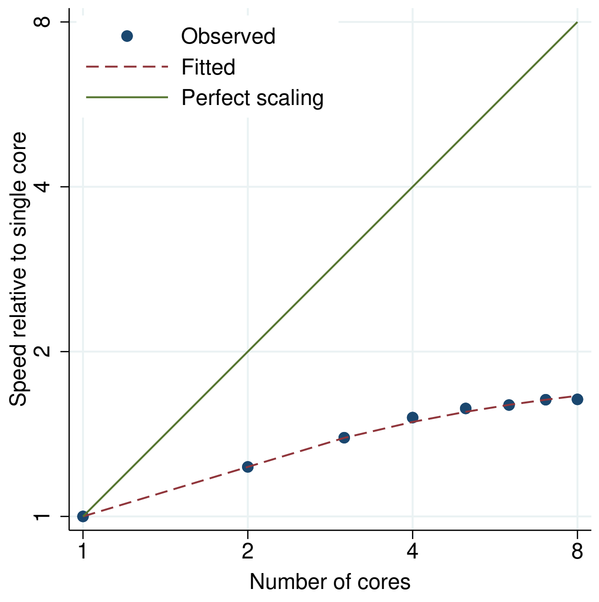


Figure 559. `vecnorm` performance plot.

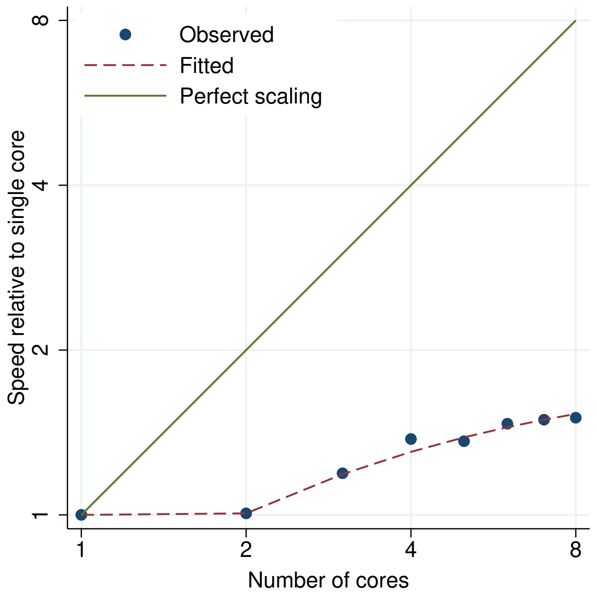


Figure 560. `vecrank` performance plot.

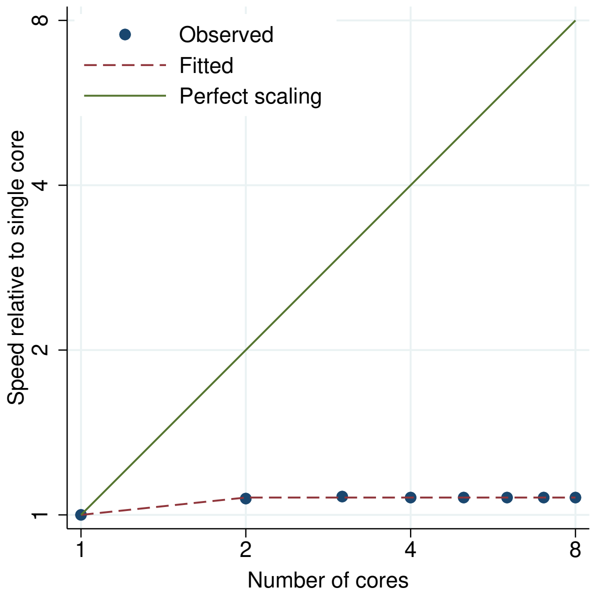


Figure 561. `vecstable` performance plot.

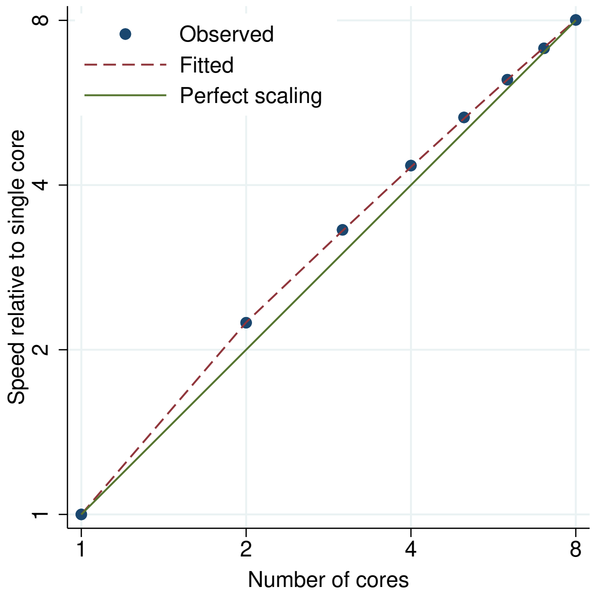


Figure 562. `vwls` performance plot.

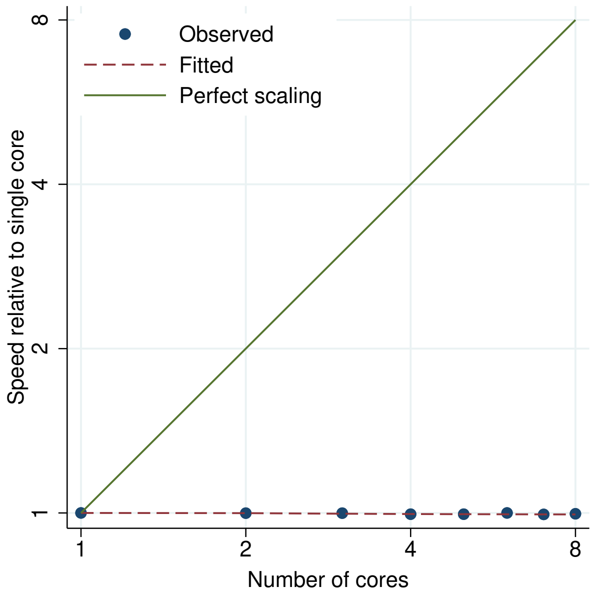


Figure 563. `wntestb` performance plot.

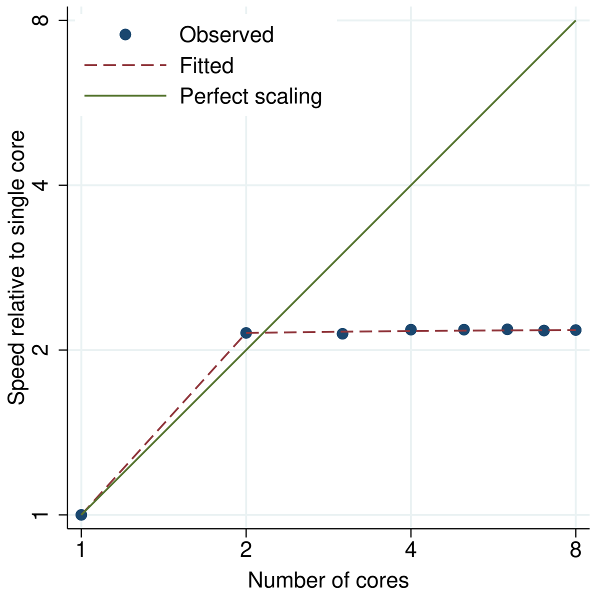


Figure 564. `wntestq` performance plot.

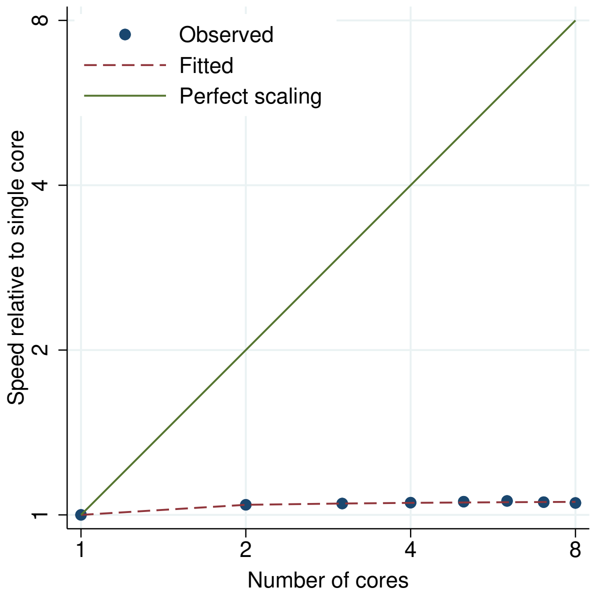


Figure 565. `xcorr` performance plot.

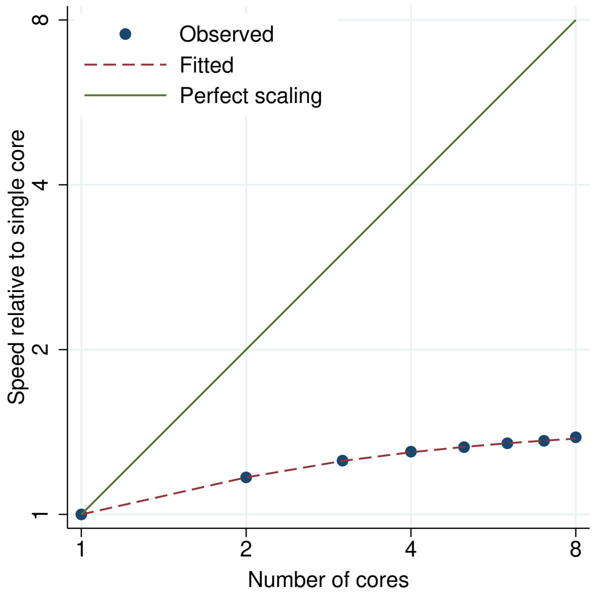


Figure 566. `xpologit` performance plot.

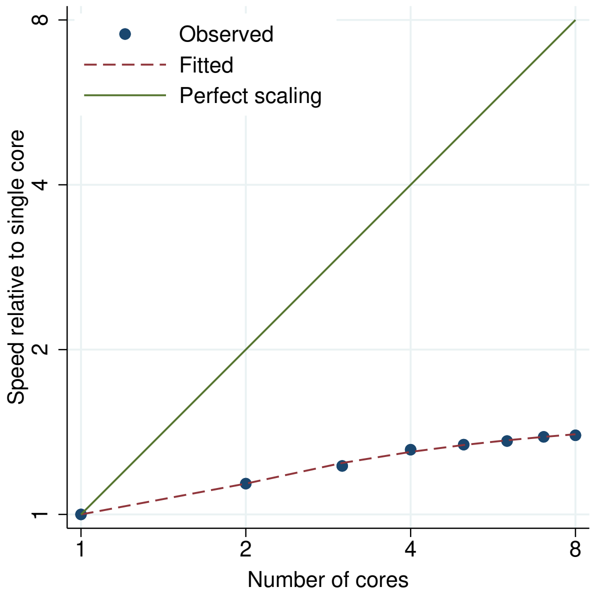


Figure 567. `xpopoisson` performance plot.

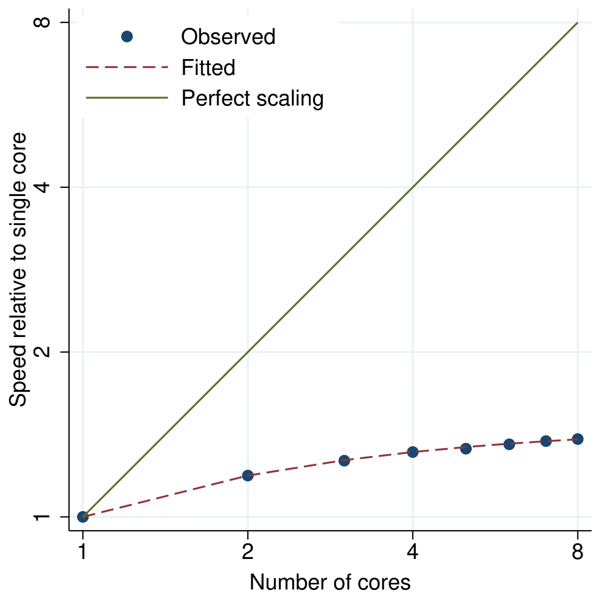


Figure 568. `xporegress` performance plot.

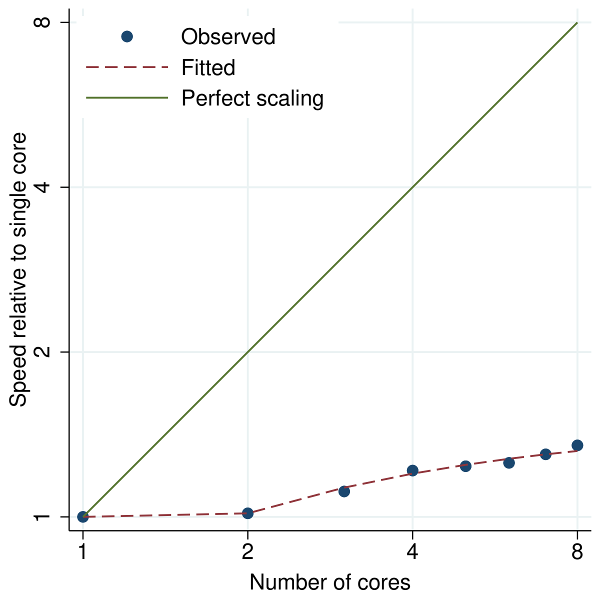


Figure 569. `xtabond` performance plot.

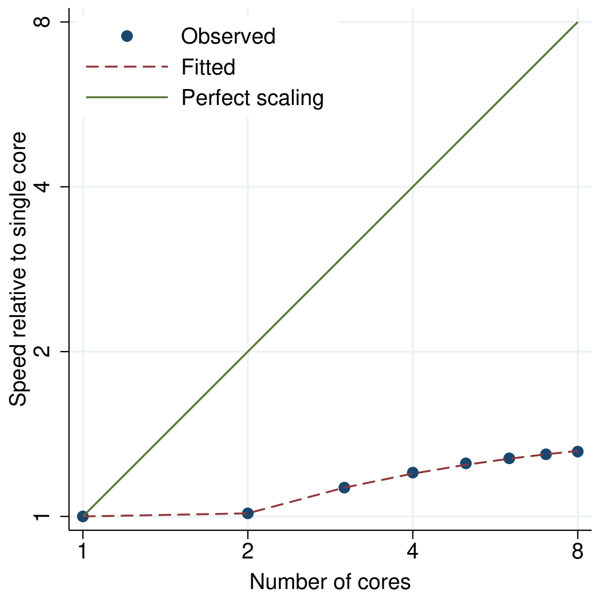


Figure 570. `xtabond, twostep` performance plot.

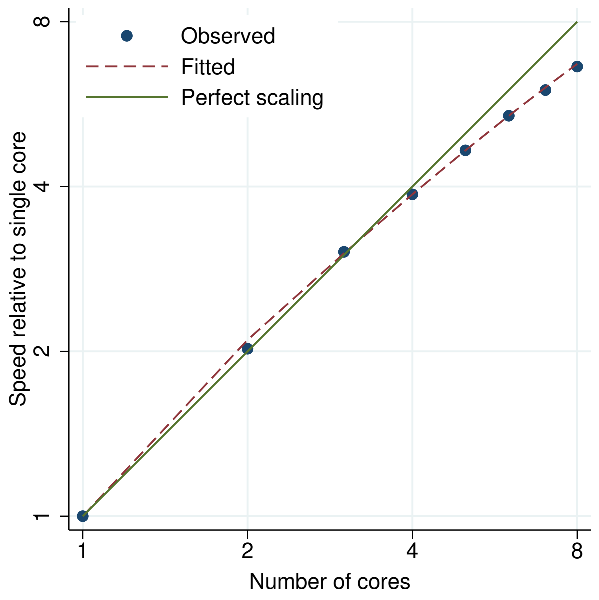


Figure 571. `xtcloglog, re` performance plot.

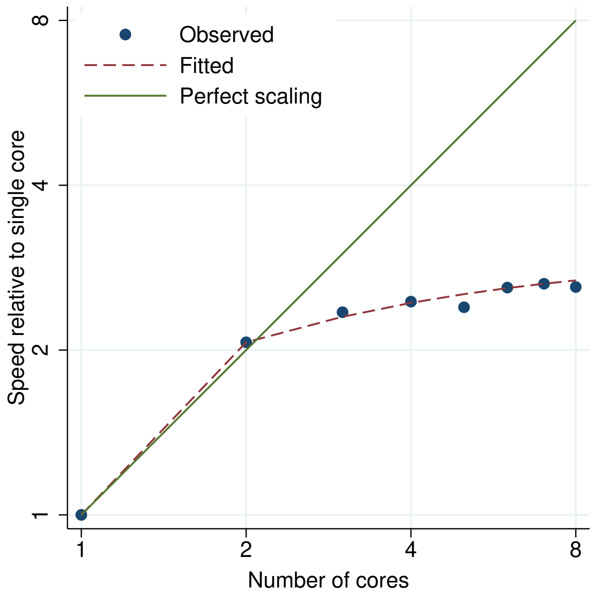


Figure 572. `xtdata, be` performance plot.

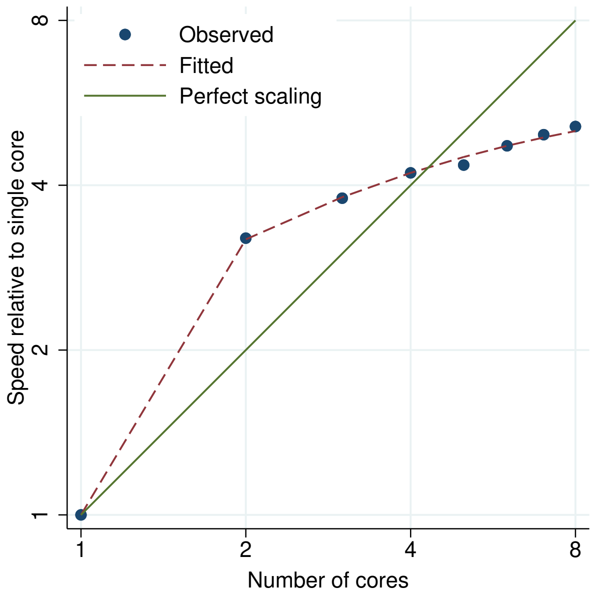


Figure 573. `xtdata, fe` performance plot.

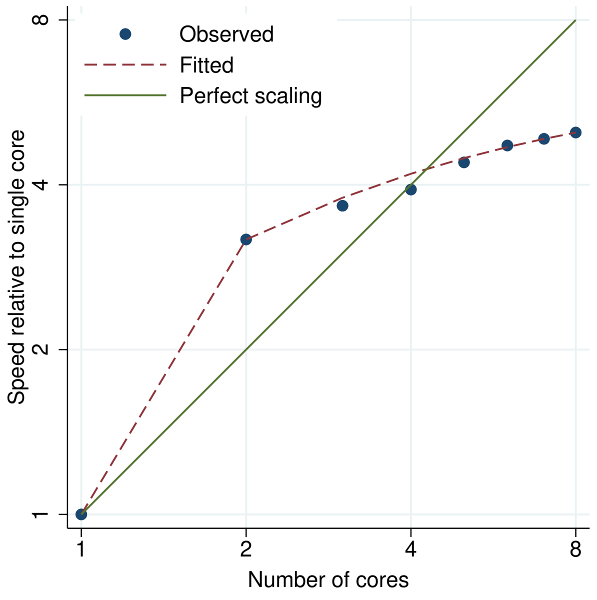


Figure 574. `xtdata, re` performance plot.

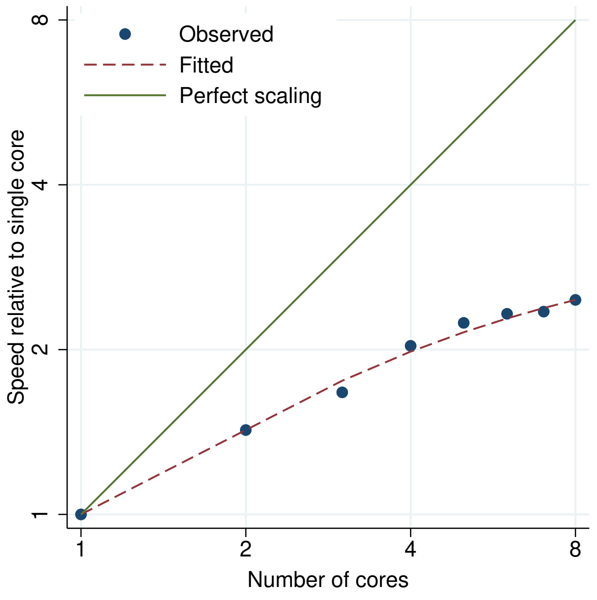


Figure 575. `xtdidregress` performance plot.

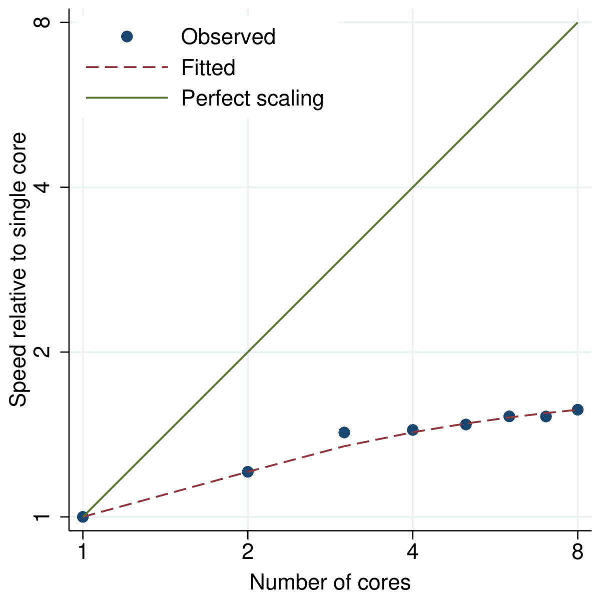


Figure 576. `xtdpd` performance plot.

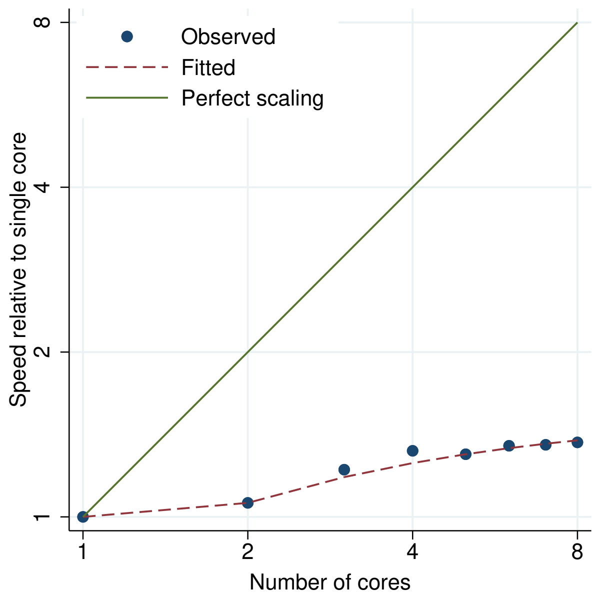


Figure 577. `xtdpdsys` performance plot.

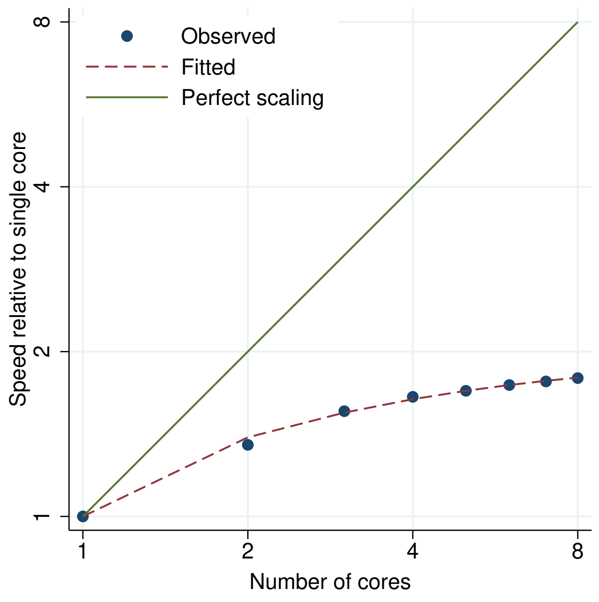


Figure 578. `xteregress` performance plot.

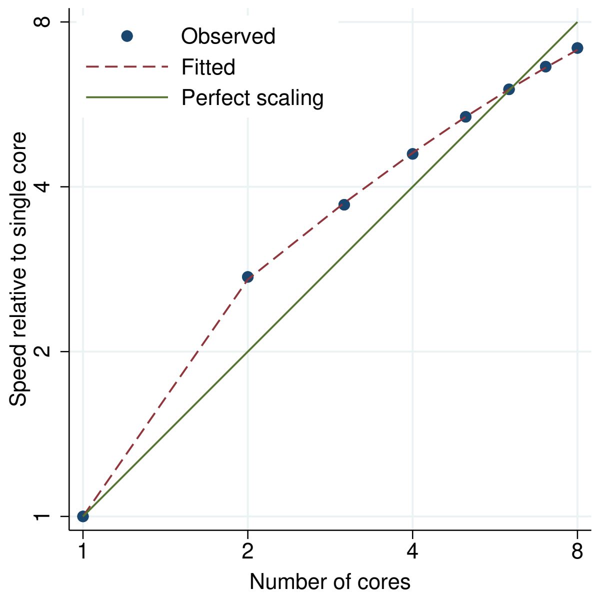


Figure 579. `xtfrontier` performance plot.

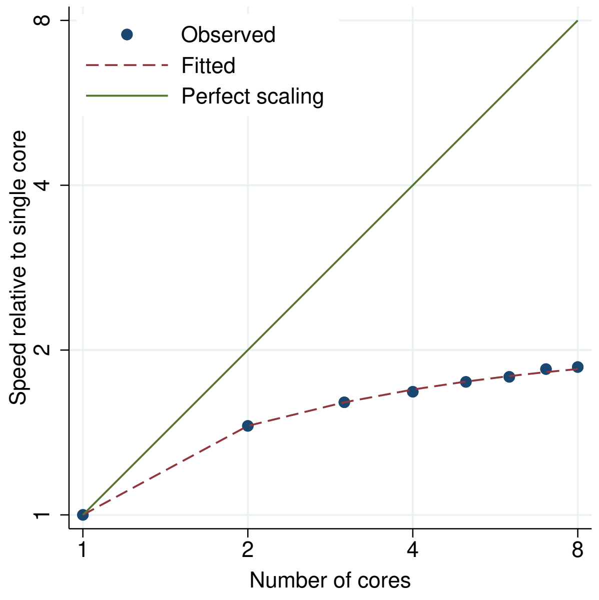


Figure 580. `xtgee, family(gaussian) corr(ar2)` performance plot.

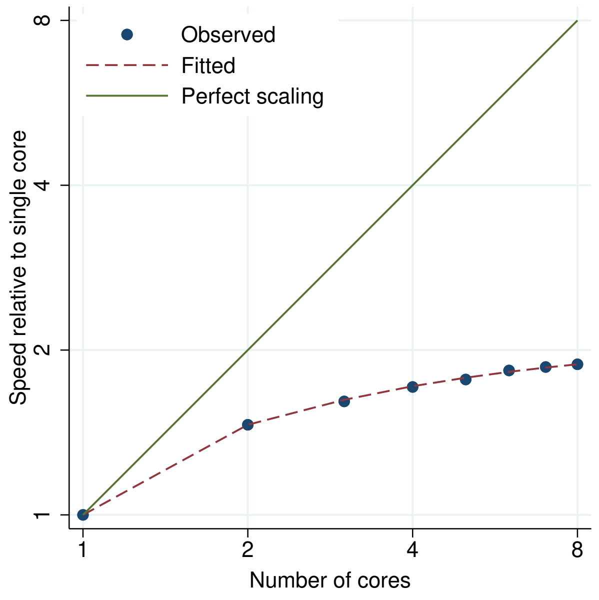


Figure 581. `xtgee, fam(gauss) corr(unstruct)` performance plot.

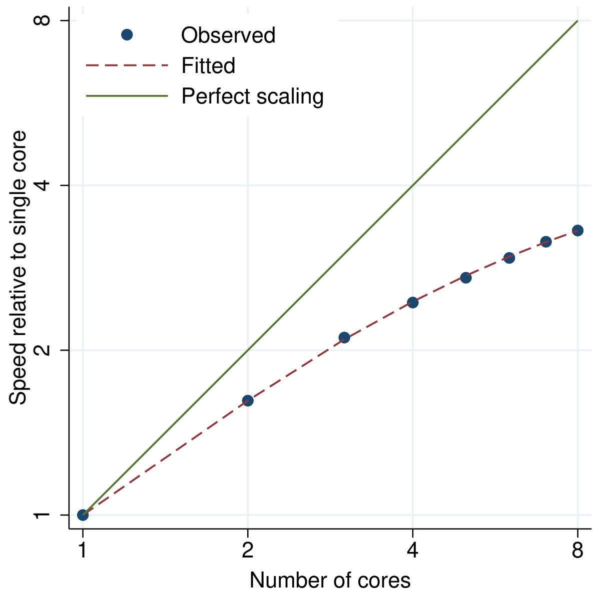


Figure 582. `xtcloglog, pa` performance plot.

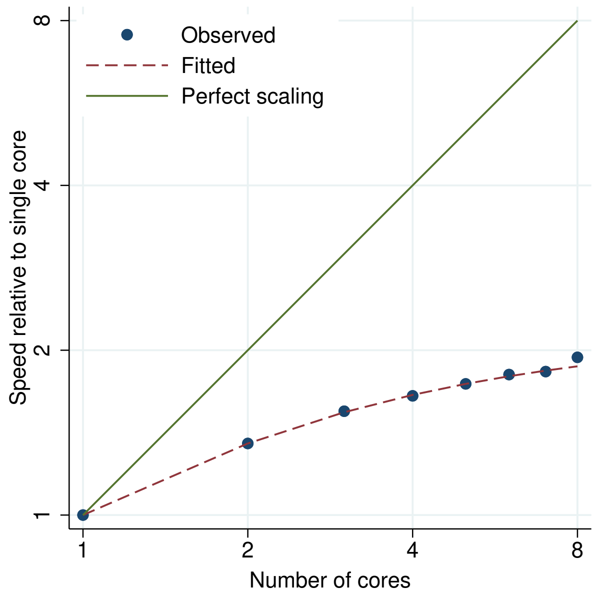


Figure 583. `xtlogit, pa` performance plot.

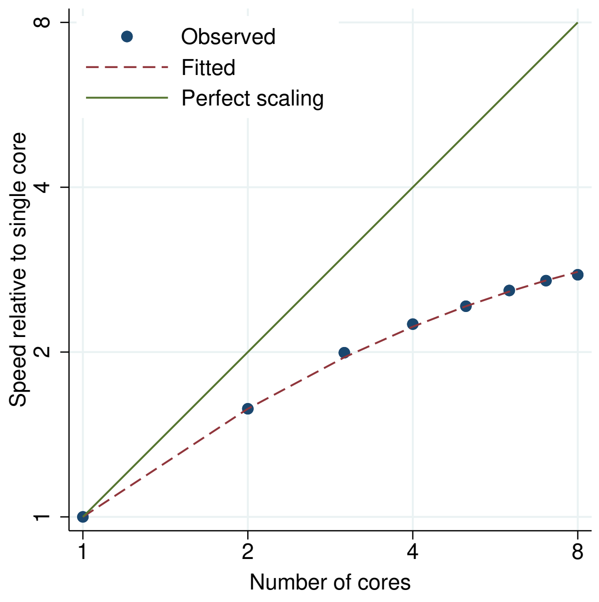


Figure 584. `xtnbreg, pa` performance plot.

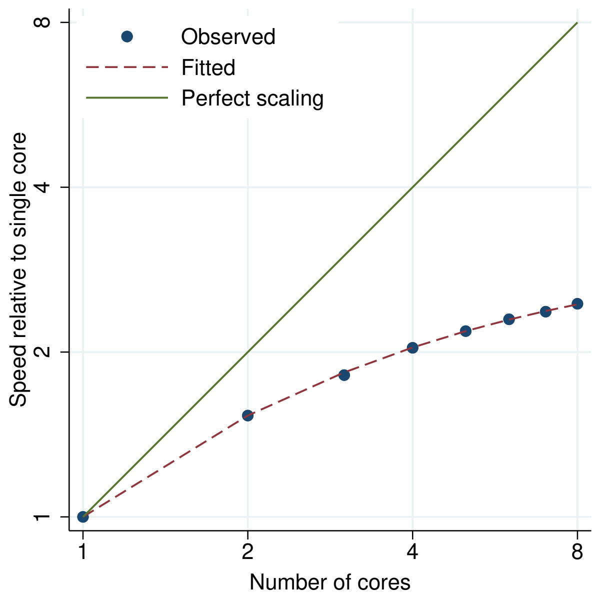


Figure 585. `xtpoisson, pa` performance plot.

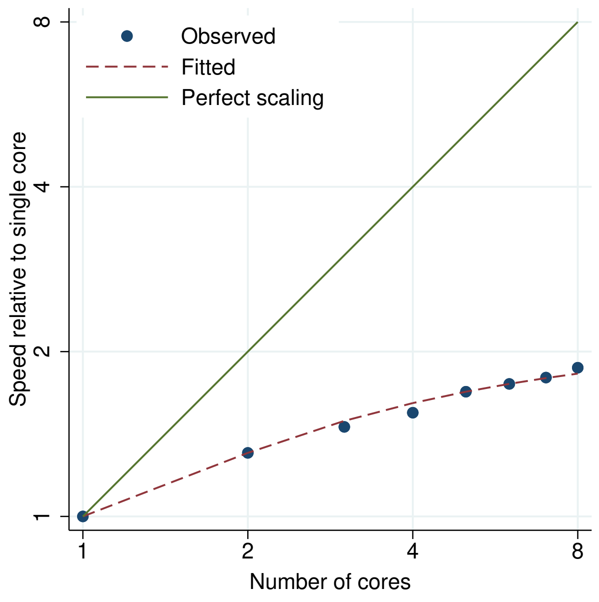


Figure 586. `xtprobit, pa` performance plot.

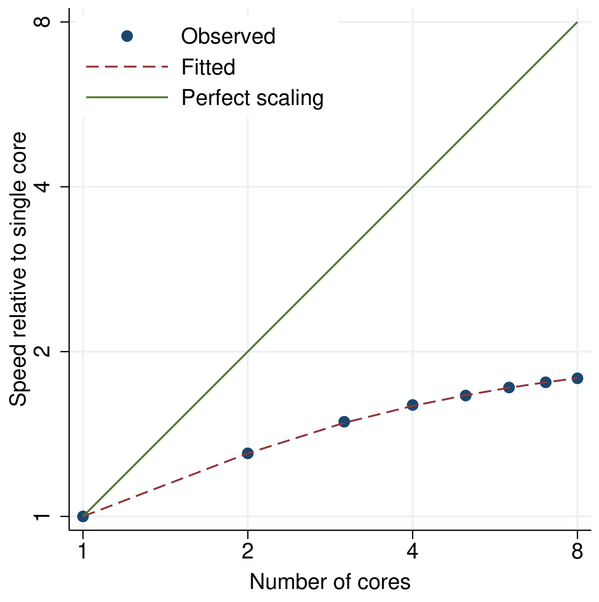


Figure 587. `xtreg, pa` performance plot.

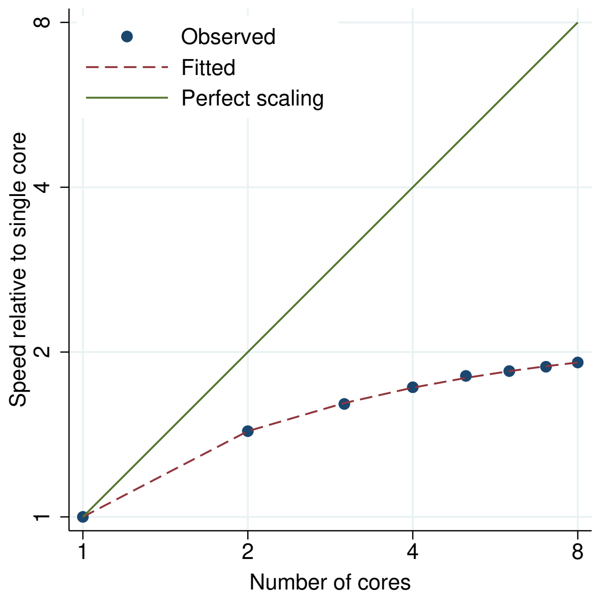


Figure 588. `xtgls` performance plot.

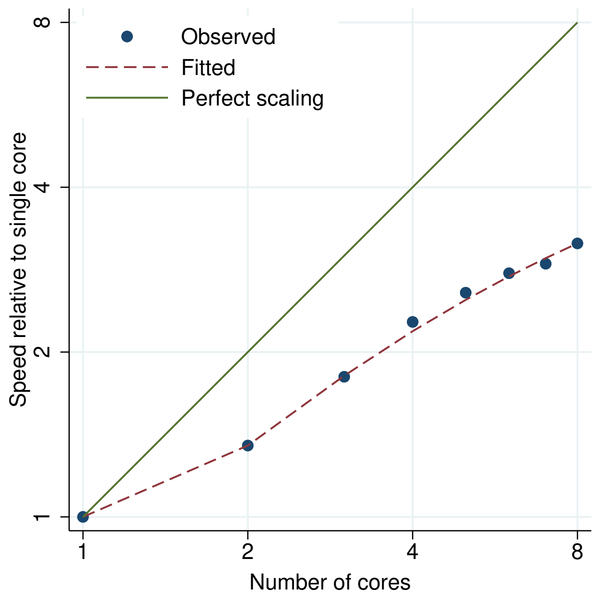


Figure 589. `xthtaylor` performance plot.

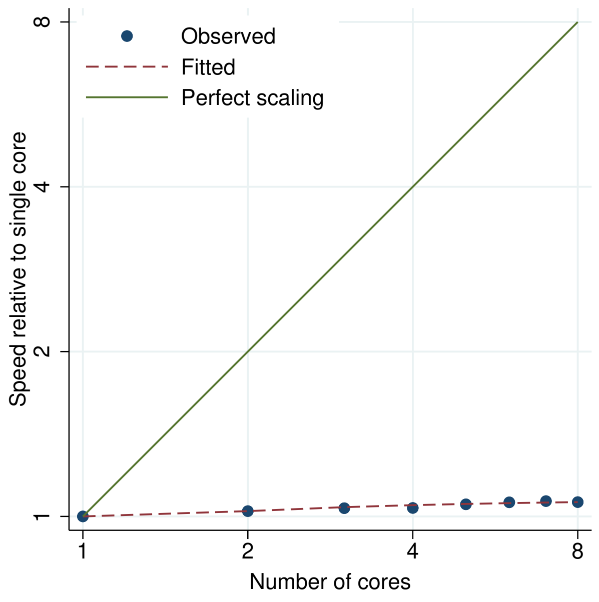


Figure 590. `xtile` performance plot.

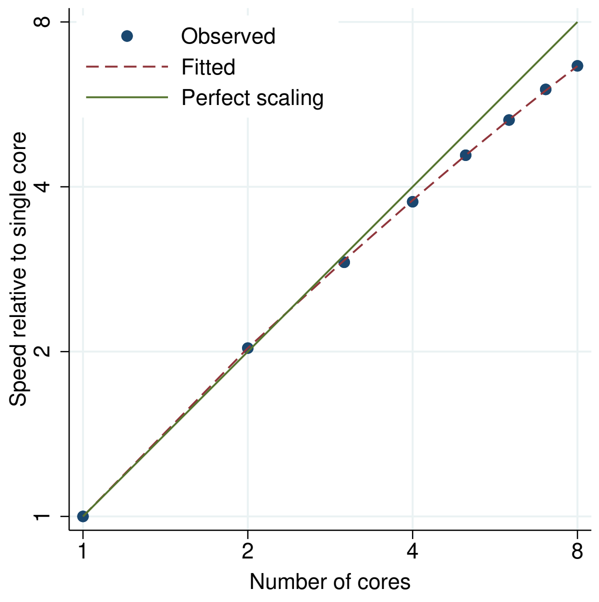


Figure 591. `xtintreg` performance plot.

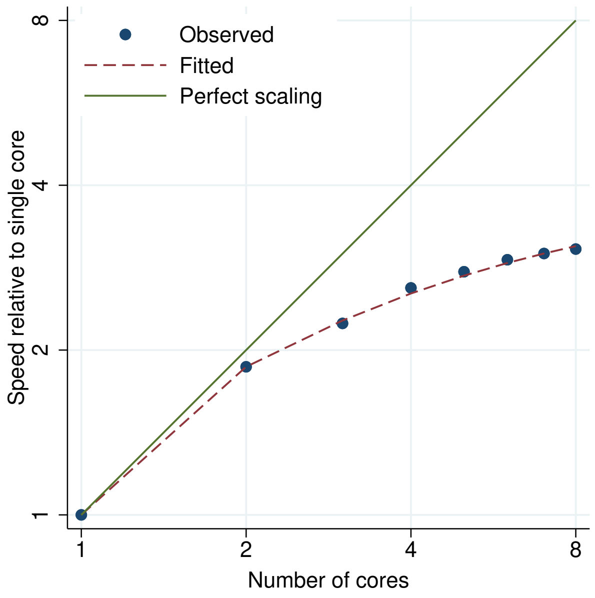


Figure 592. `xtivreg, be` performance plot.

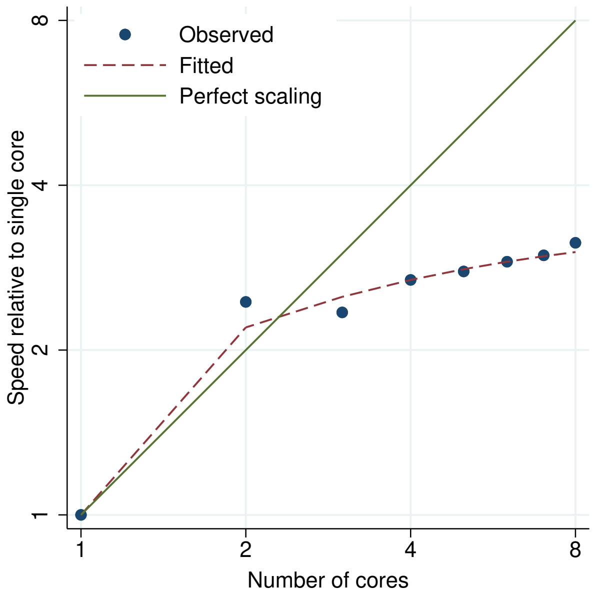


Figure 593. `xtivreg, fd` performance plot.

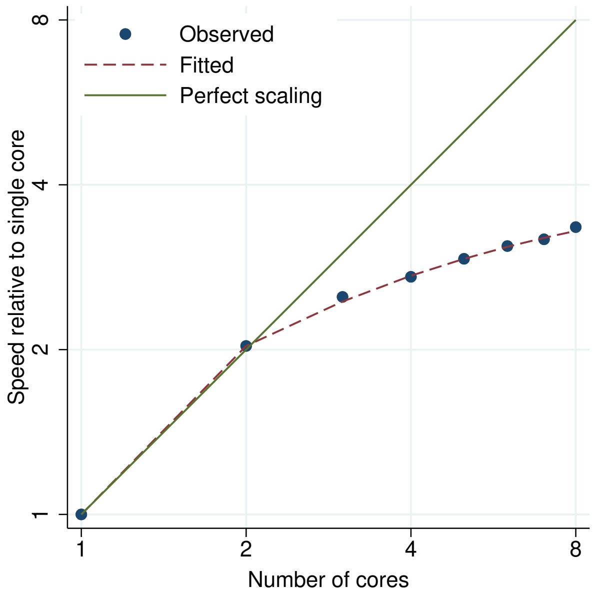


Figure 594. `xtivreg, fe` performance plot.

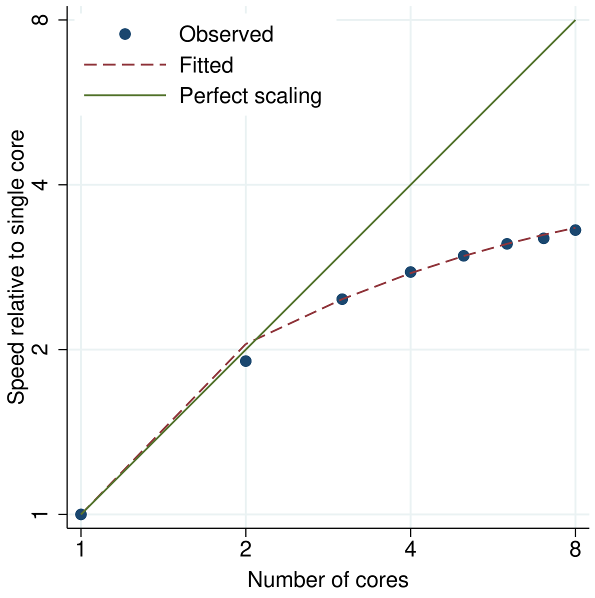


Figure 595. `xtivreg, re` performance plot.

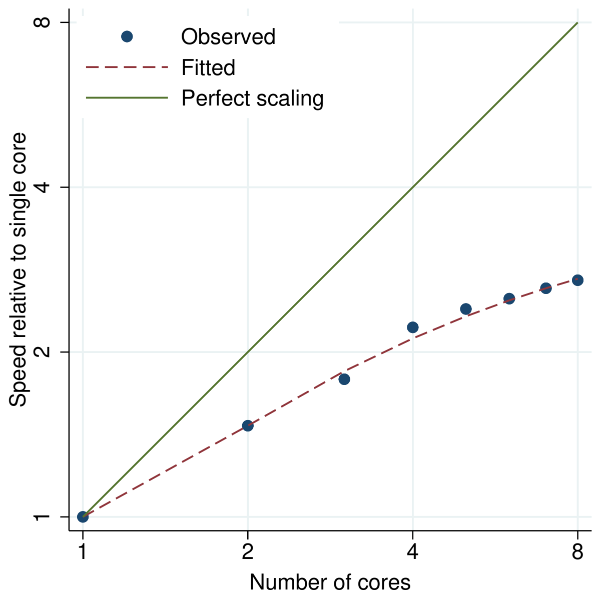


Figure 596. `xtlogit, fe` performance plot.

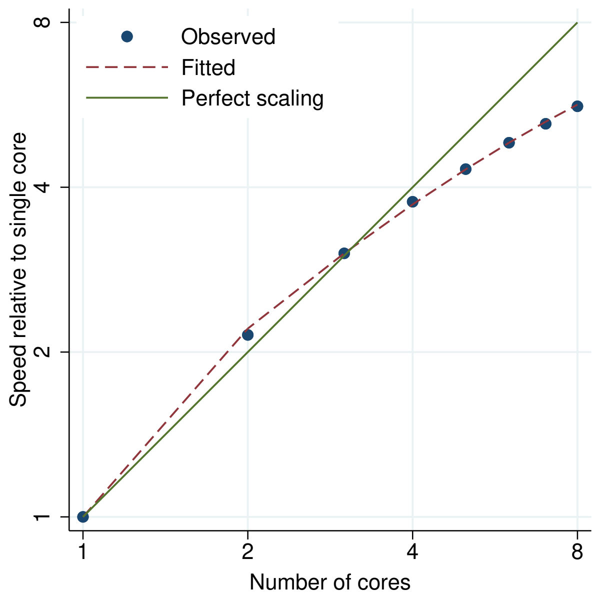


Figure 597. `xtlogit, re` performance plot.

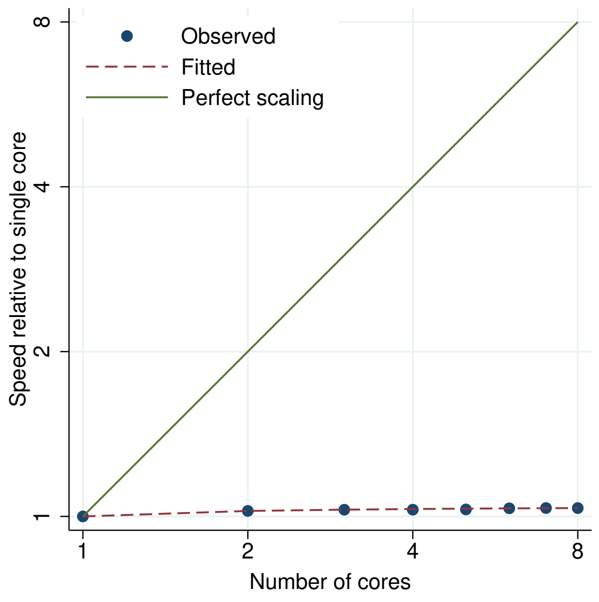


Figure 598. `xtmlogit, fe` performance plot.

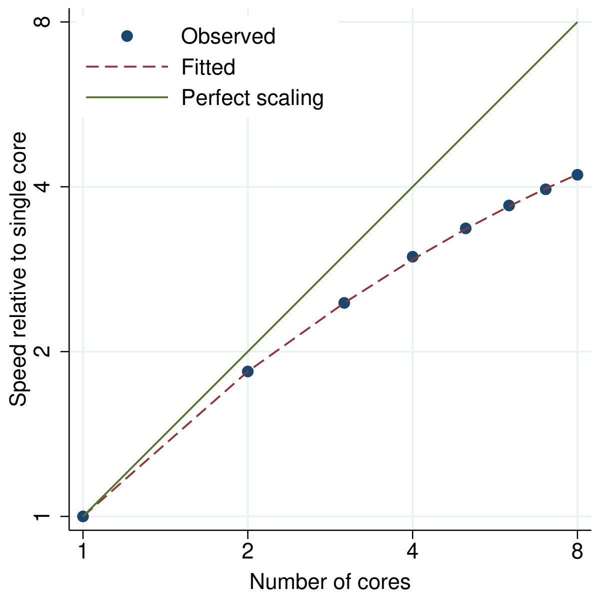


Figure 599. `xtmlogit, re` performance plot.

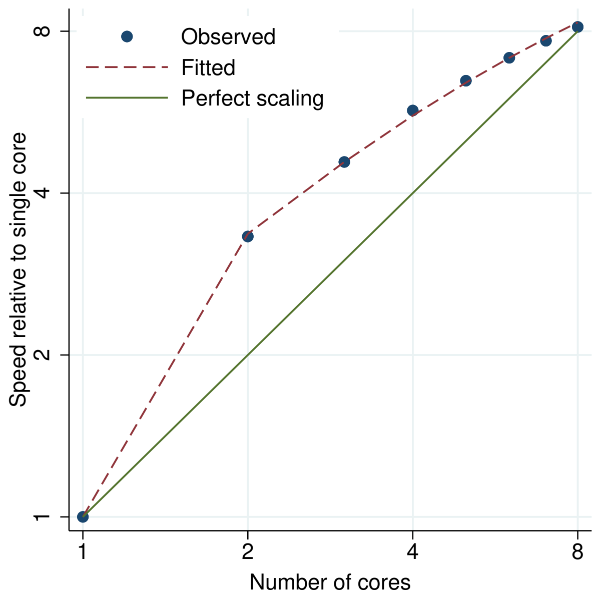


Figure 600. `xtnbreg, fe` performance plot.

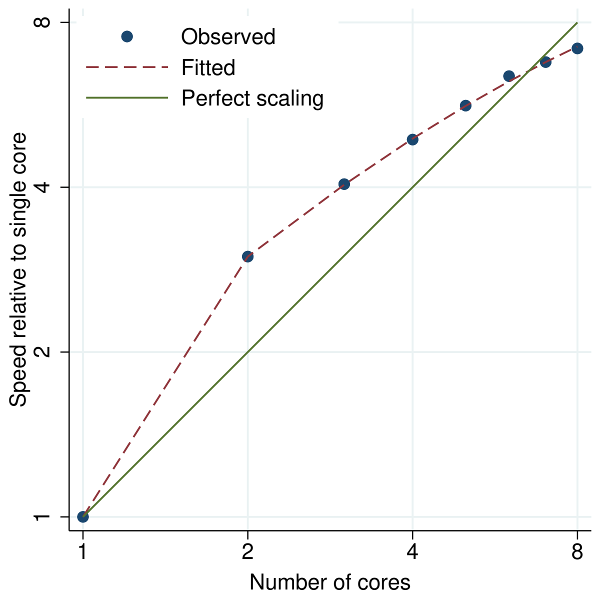


Figure 601. `xtnbreg, re` performance plot.

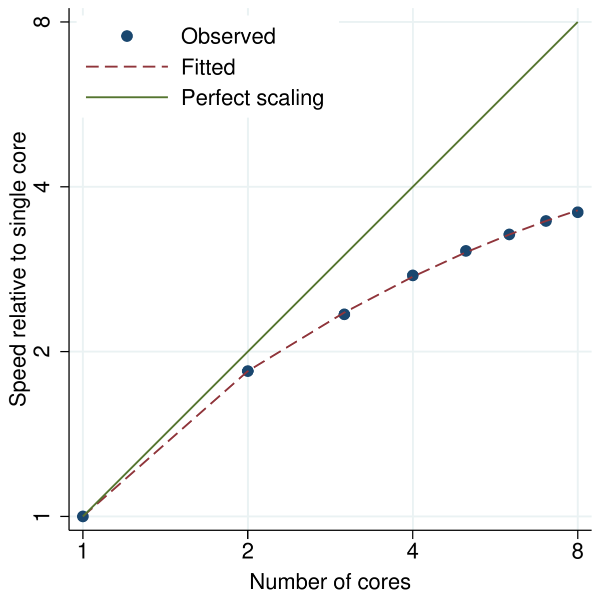


Figure 602. `xtologit` performance plot.

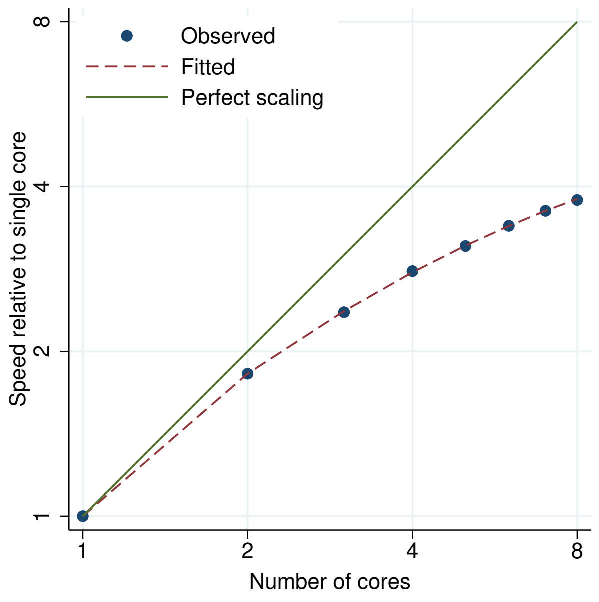


Figure 603. `xtoprobit` performance plot.

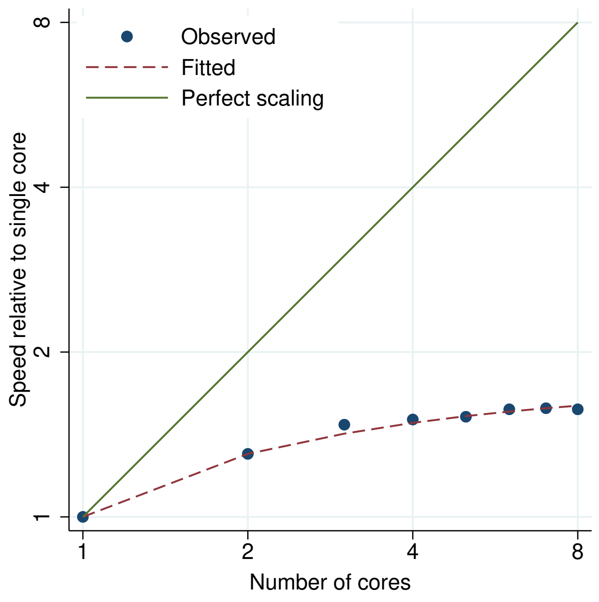


Figure 604. `xtpcse` performance plot.

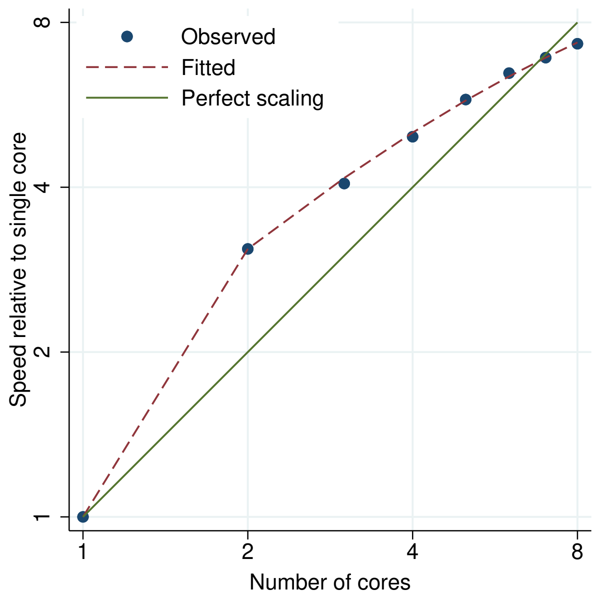


Figure 605. `xtpoisson, fe` performance plot.

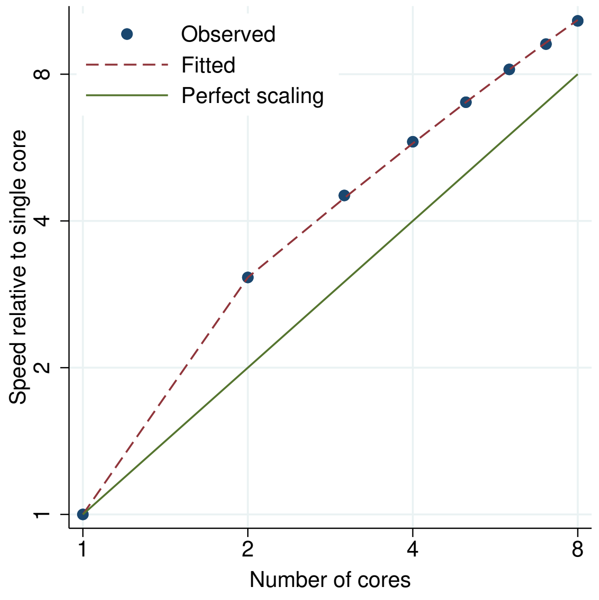


Figure 606. `xtpoisson, re` performance plot.

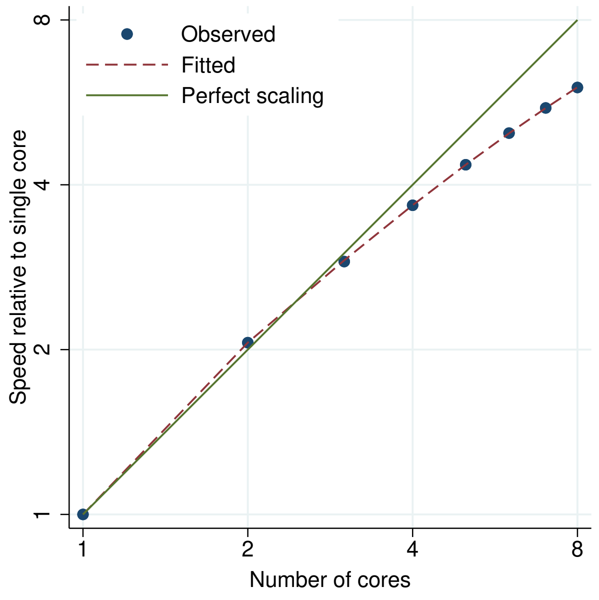


Figure 607. `xtprobit, re` performance plot.

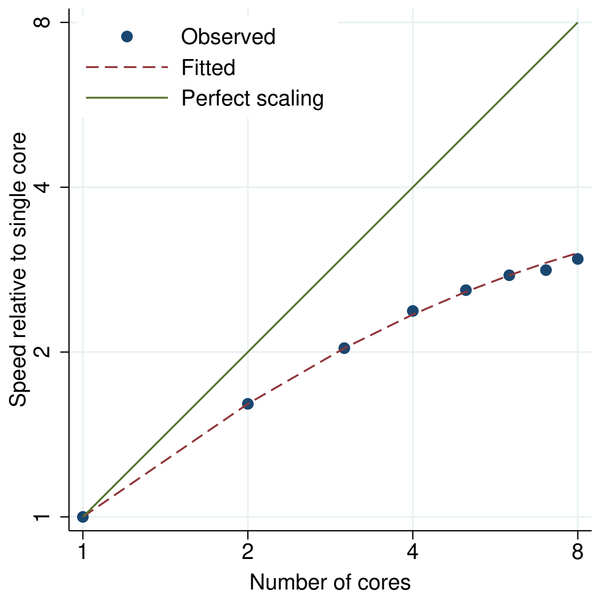


Figure 608. `xtrc` performance plot.

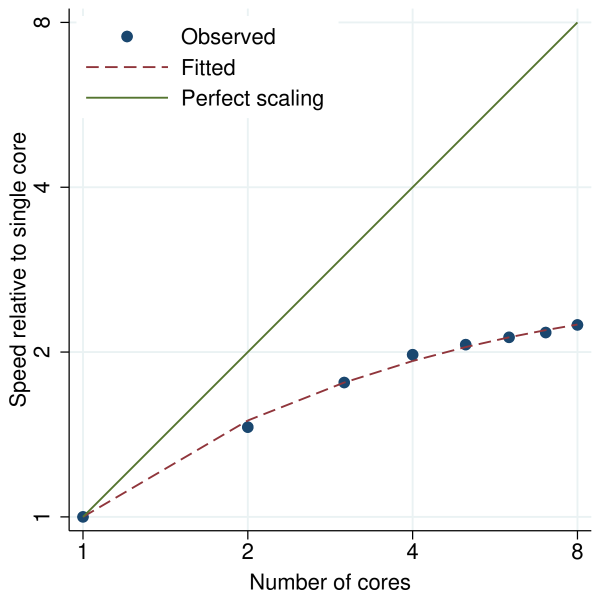


Figure 609. `xtreg, be` performance plot.

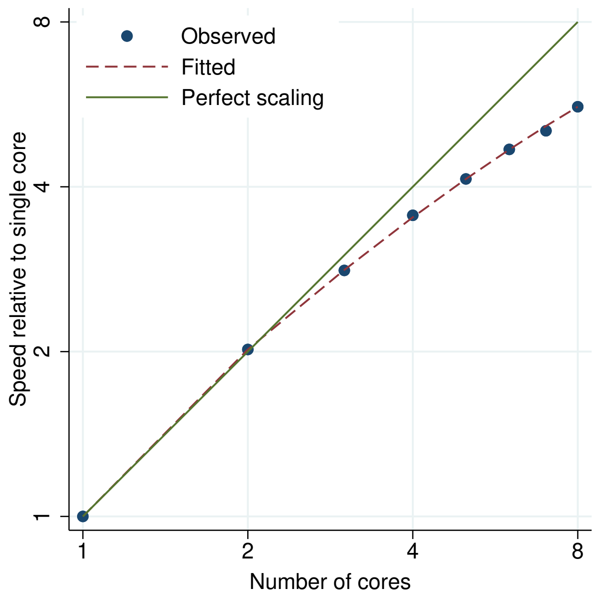


Figure 610. `xtreg, fe` performance plot.

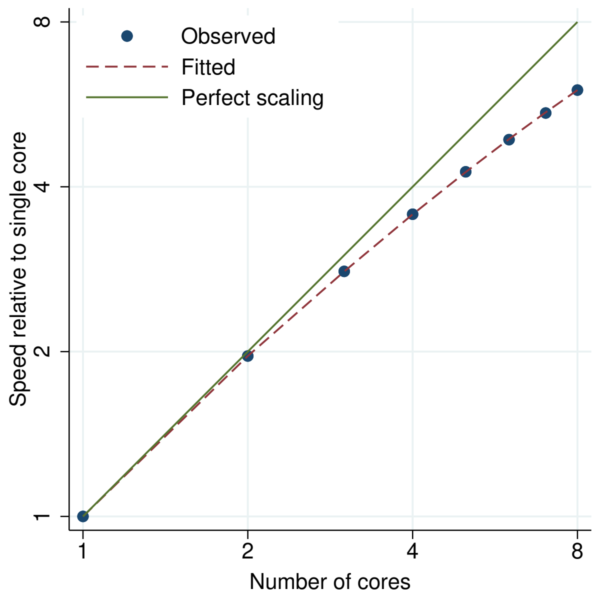


Figure 611. `xtreg, fe vce(robust)` performance plot.

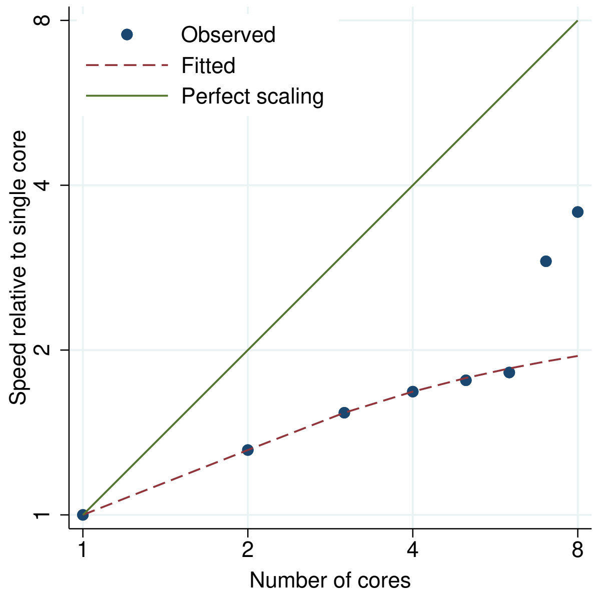


Figure 612. `xtreg, mle` performance plot.

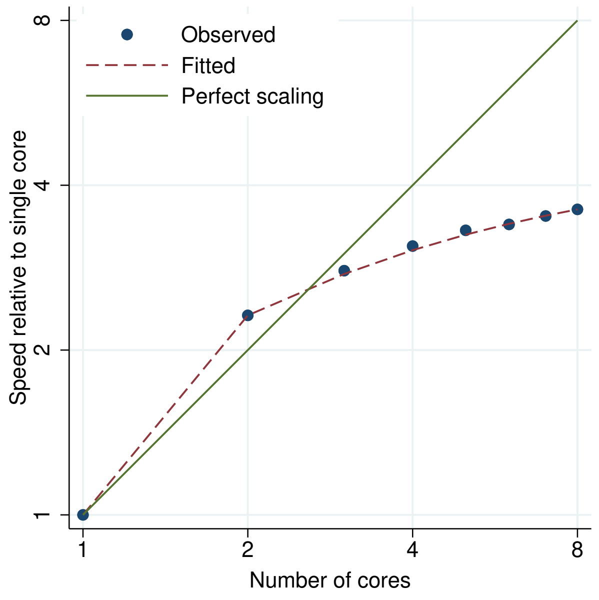


Figure 613. `xtreg, re` performance plot.

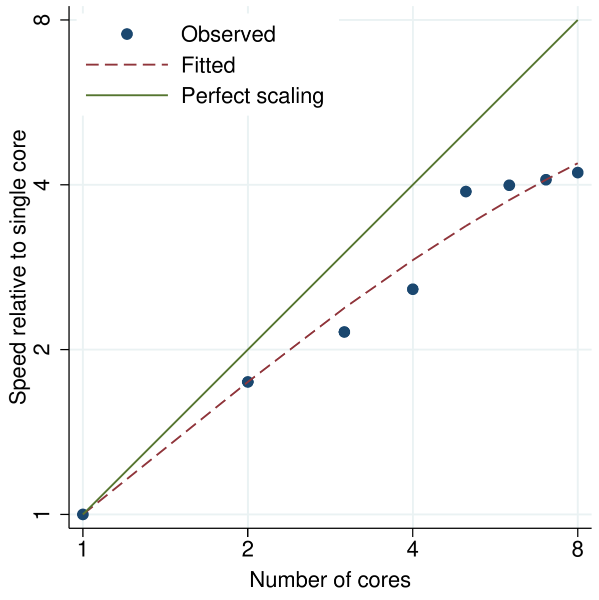


Figure 614. `xtregar, fe` performance plot.

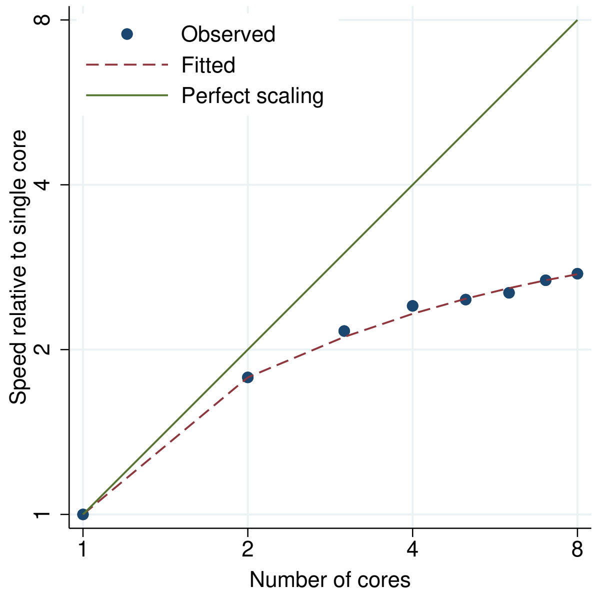


Figure 615. `xtregar, re` performance plot.

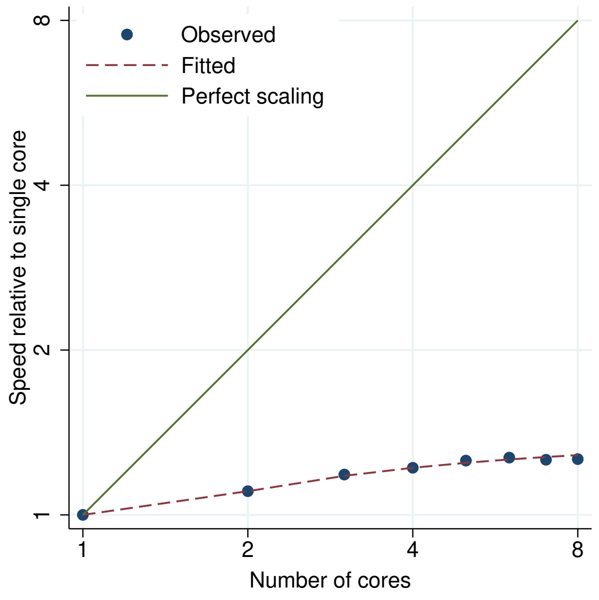


Figure 616. `xtset` performance plot.

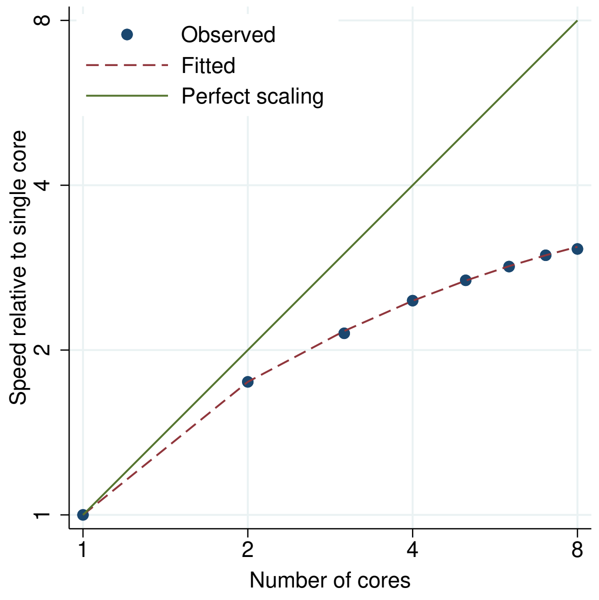


Figure 617. `xtstreg, distribution(exponential)` performance plot.

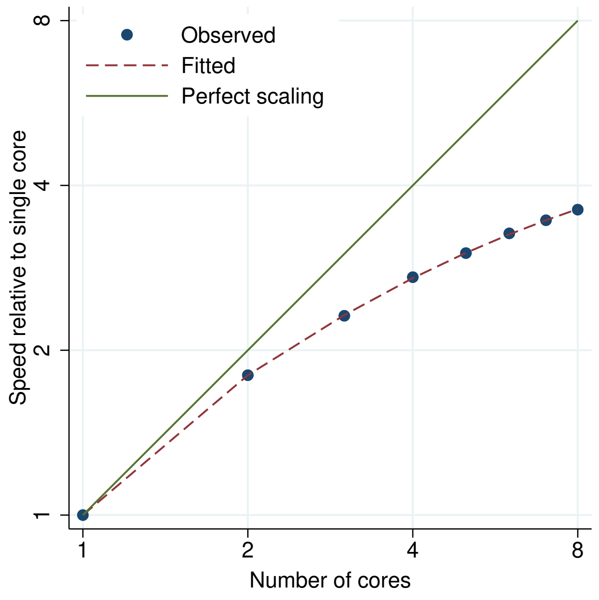


Figure 618. `xtstreg, distribution(weibull)` performance plot.

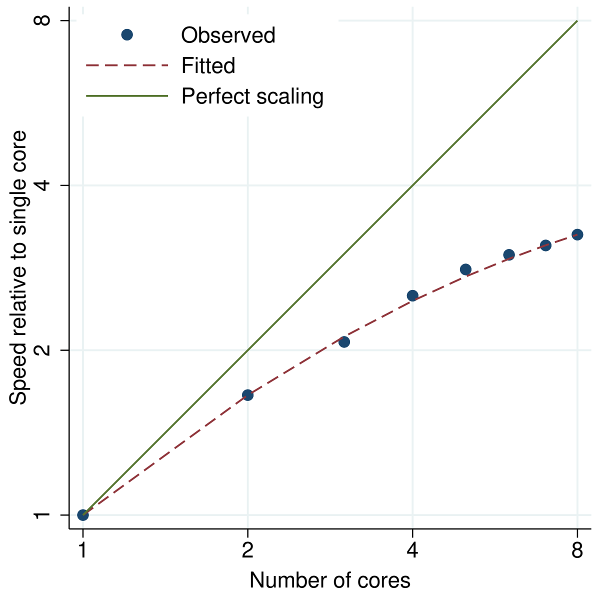


Figure 619. `xtsum` performance plot.

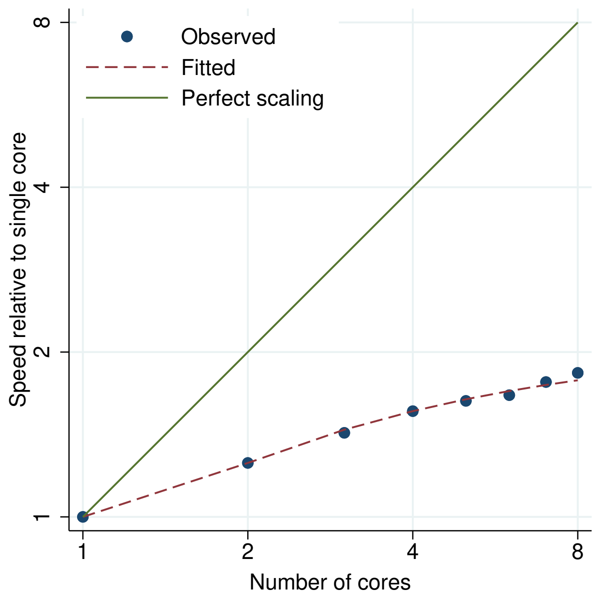


Figure 620. `xttab` performance plot.

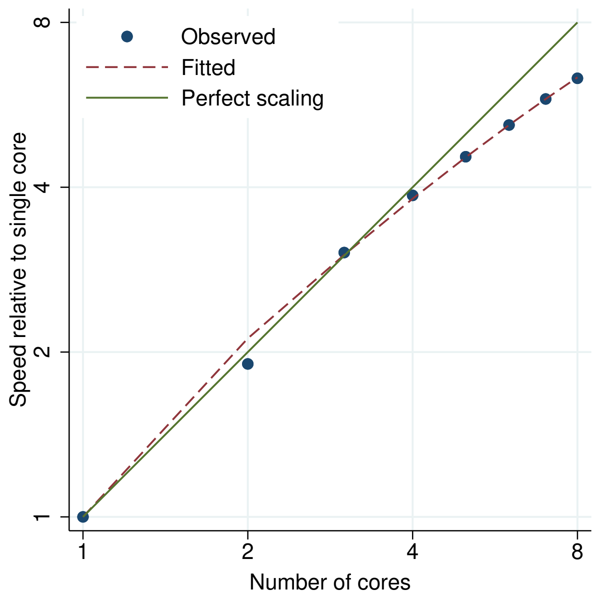


Figure 621. `xttobit` performance plot.

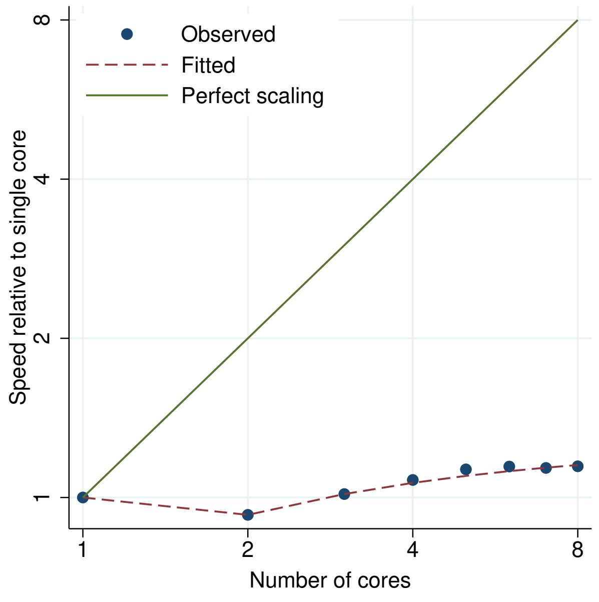


Figure 622. `xtunitroot breitung` performance plot.

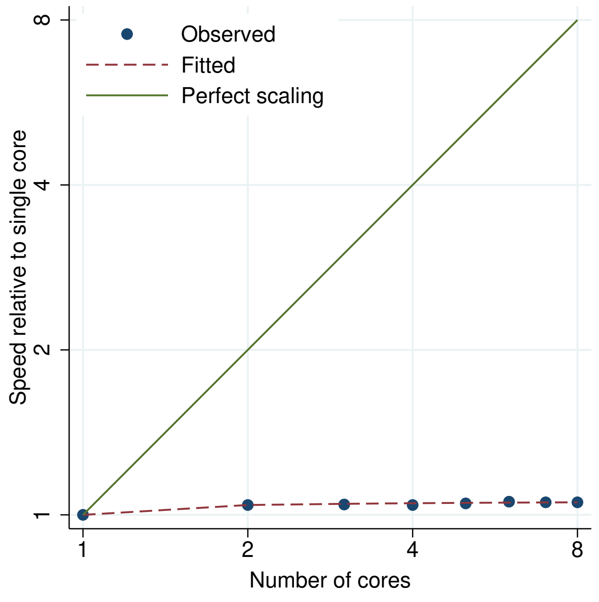


Figure 623. `xtunitroot fisher` performance plot.

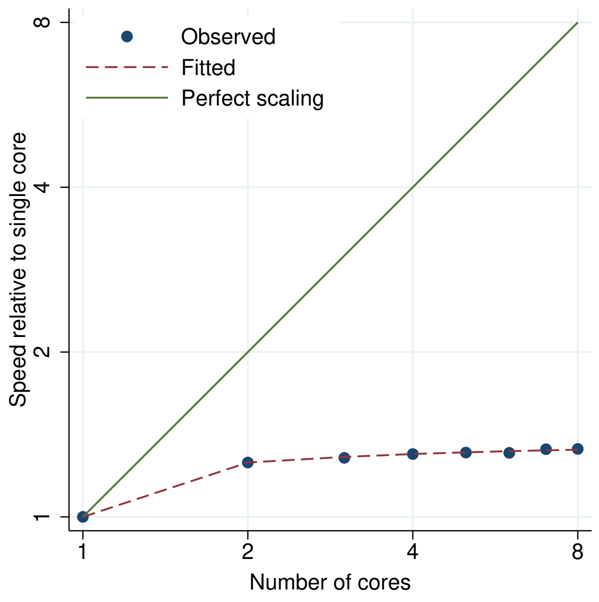


Figure 624. `xtunitroot hadri` performance plot.

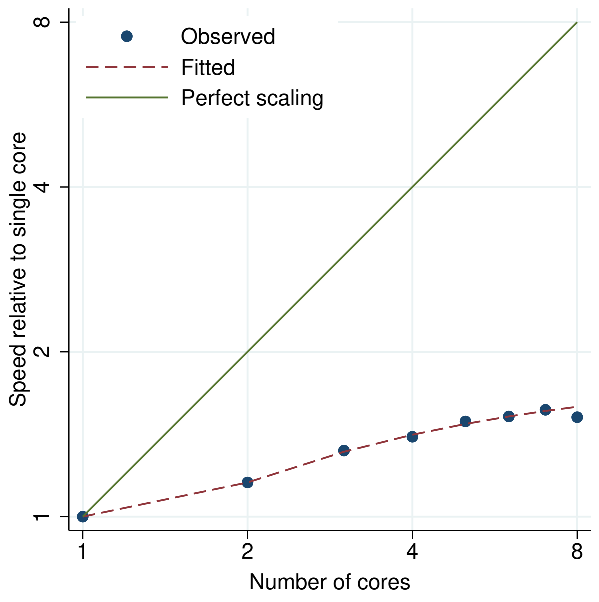


Figure 625. `xtunitroot ht` performance plot.

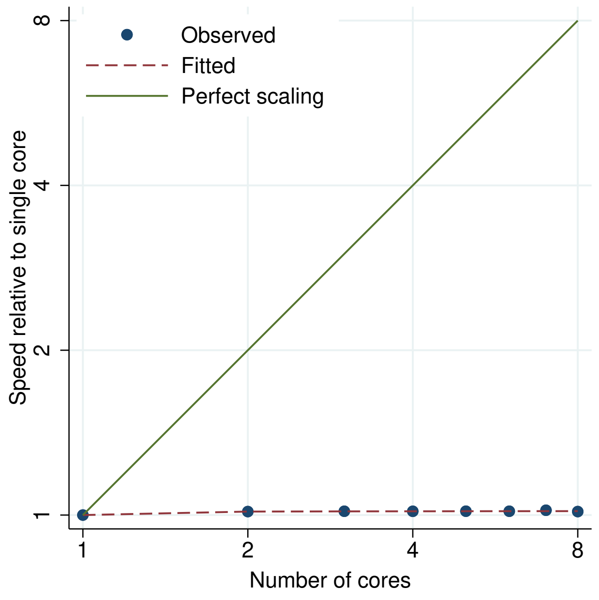


Figure 626. `xtunitroot ips` performance plot.

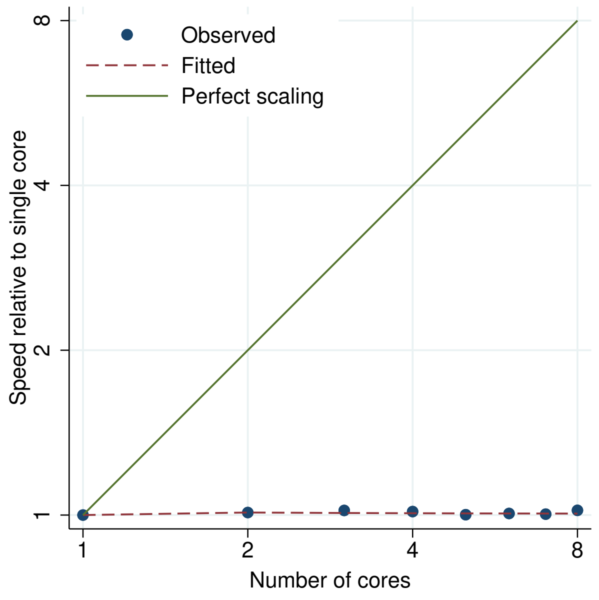


Figure 627. `xtunitroot llc` performance plot.

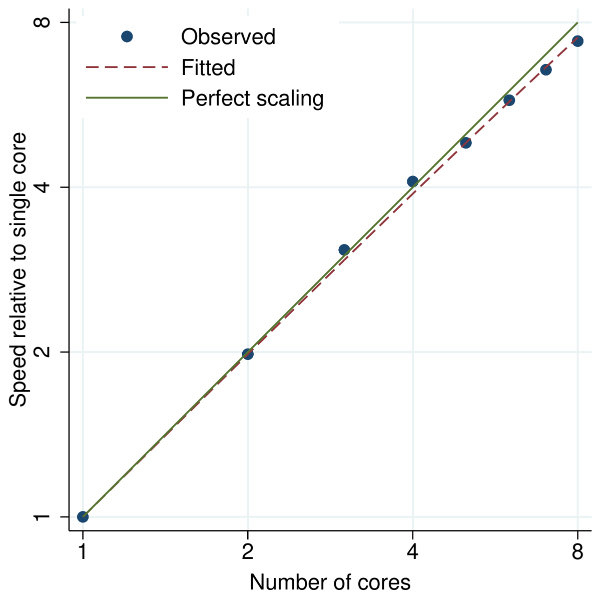


Figure 628. `zinb` performance plot.

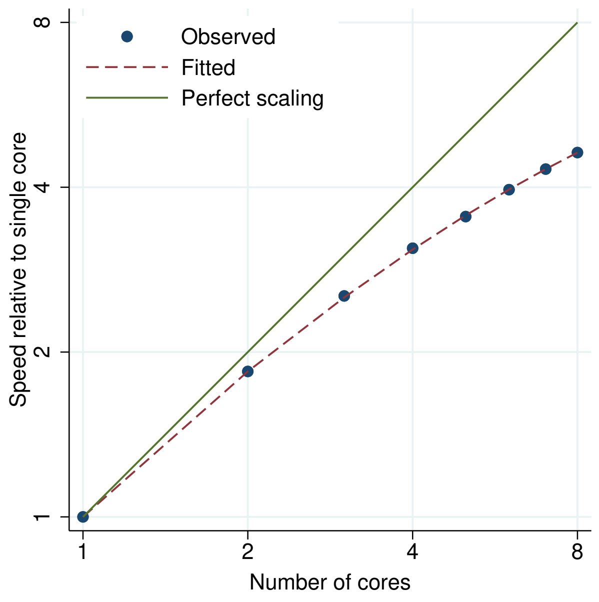


Figure 629. `ziologit` performance plot.

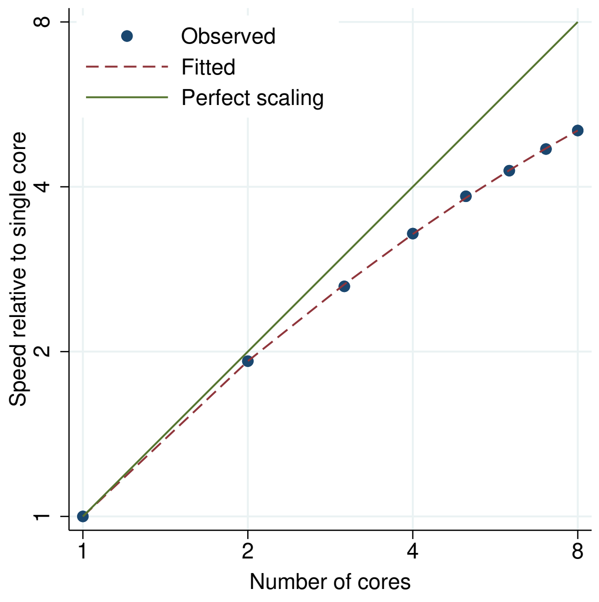


Figure 630. `zioprobit` performance plot.

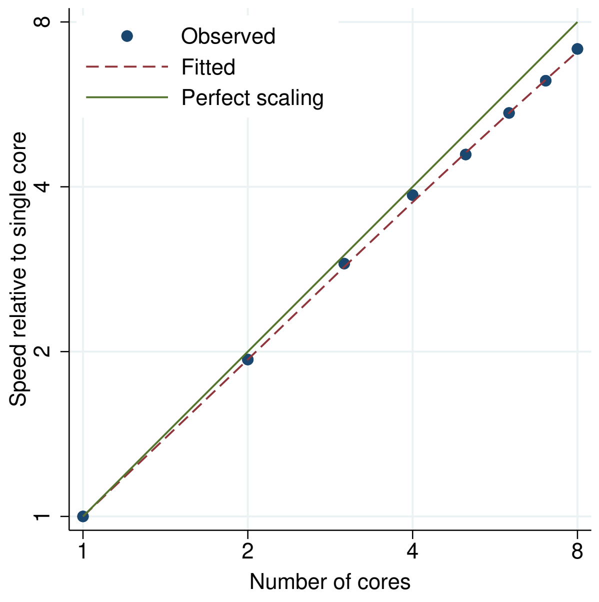


Figure 631. `zip` performance plot.

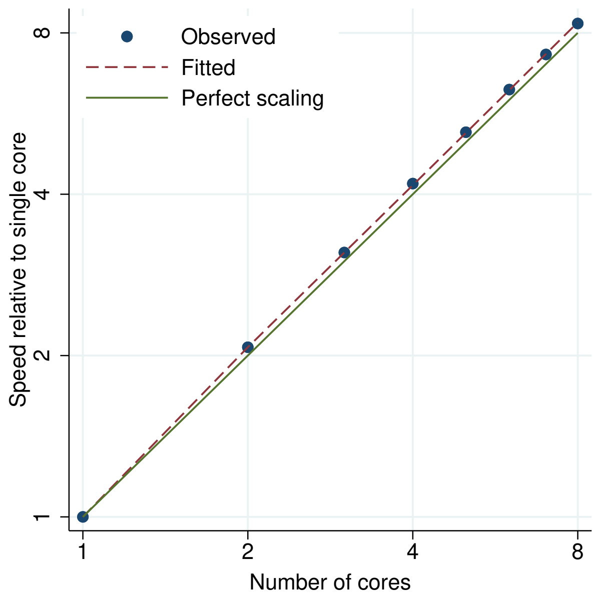


Figure 632. `_predict, xb` performance plot.

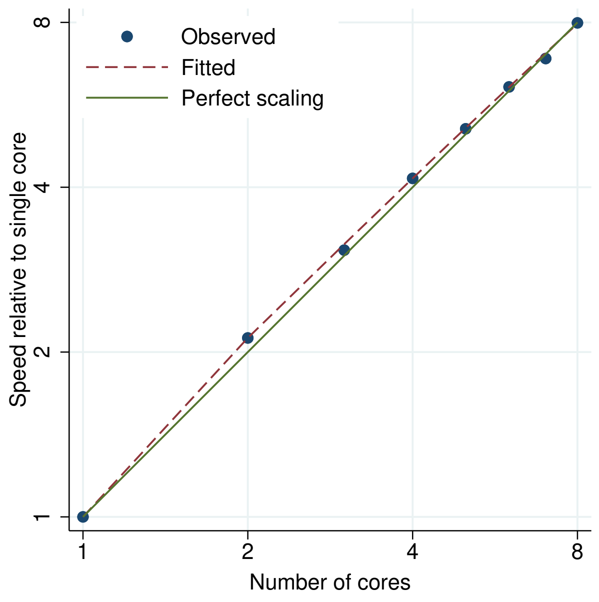


Figure 633. `_rmcoll` performance plot.

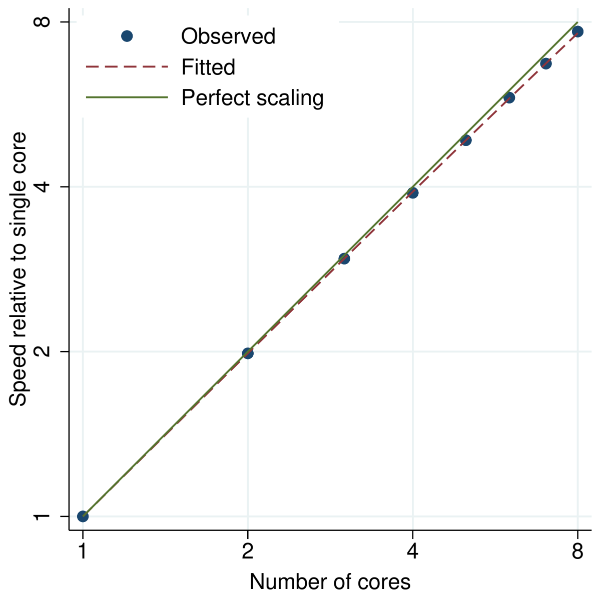


Figure 634. `_robust` performance plot.

## B Performance assessment graphs for high-end servers

Performance graphs of all 614 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.

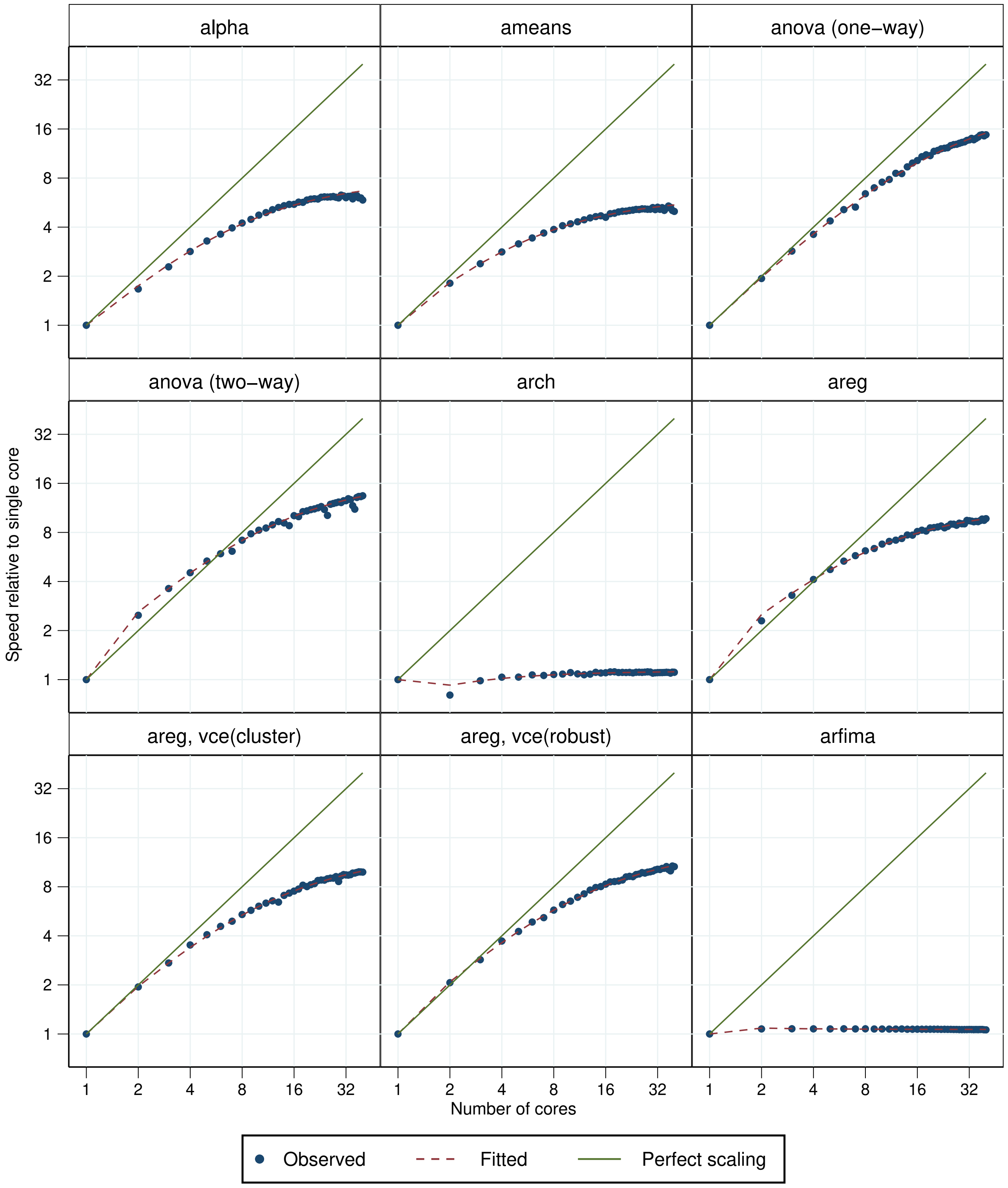


Figure 635. **Parallelization performance plots.**

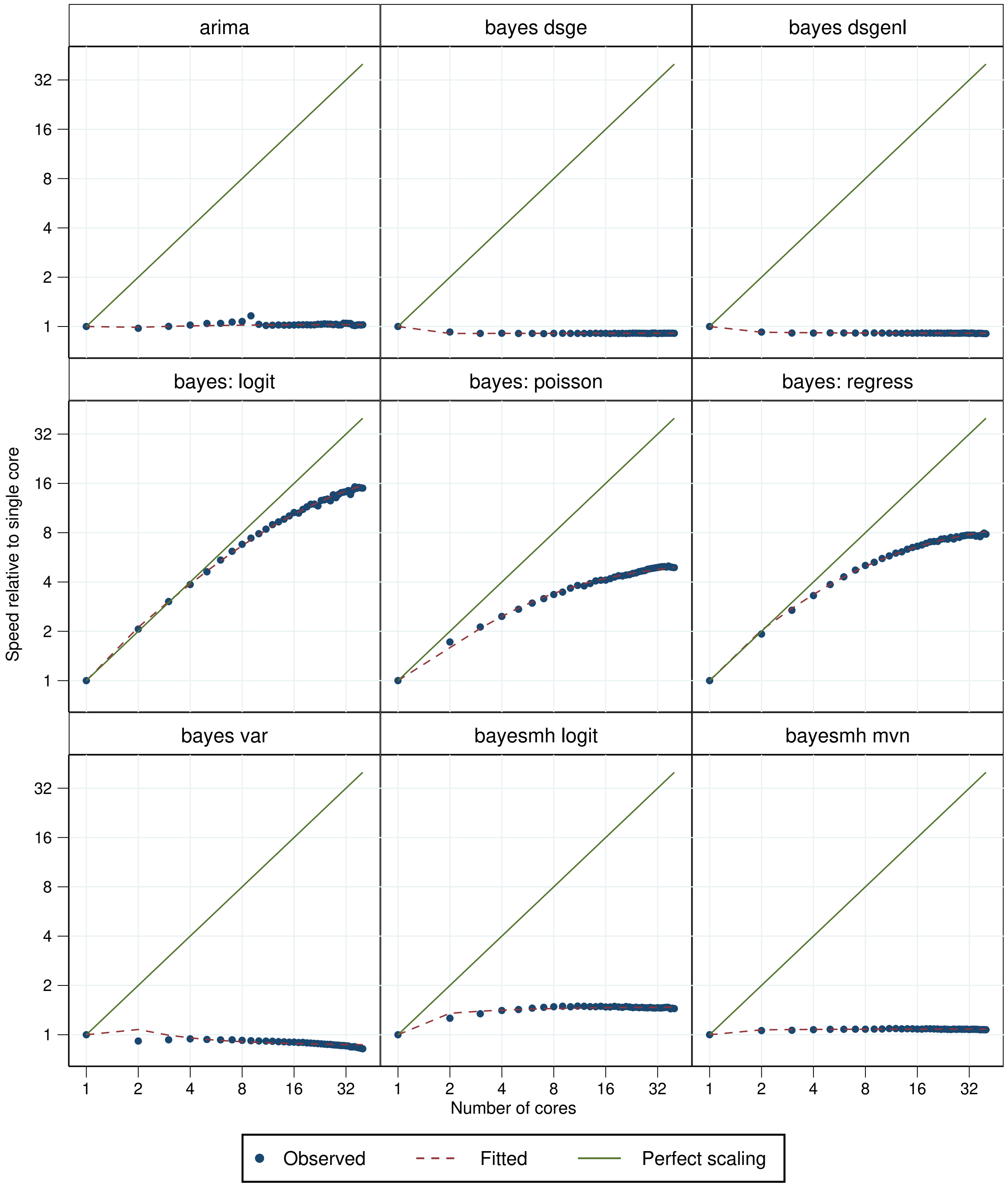


Figure 636. **Parallelization performance plots.**

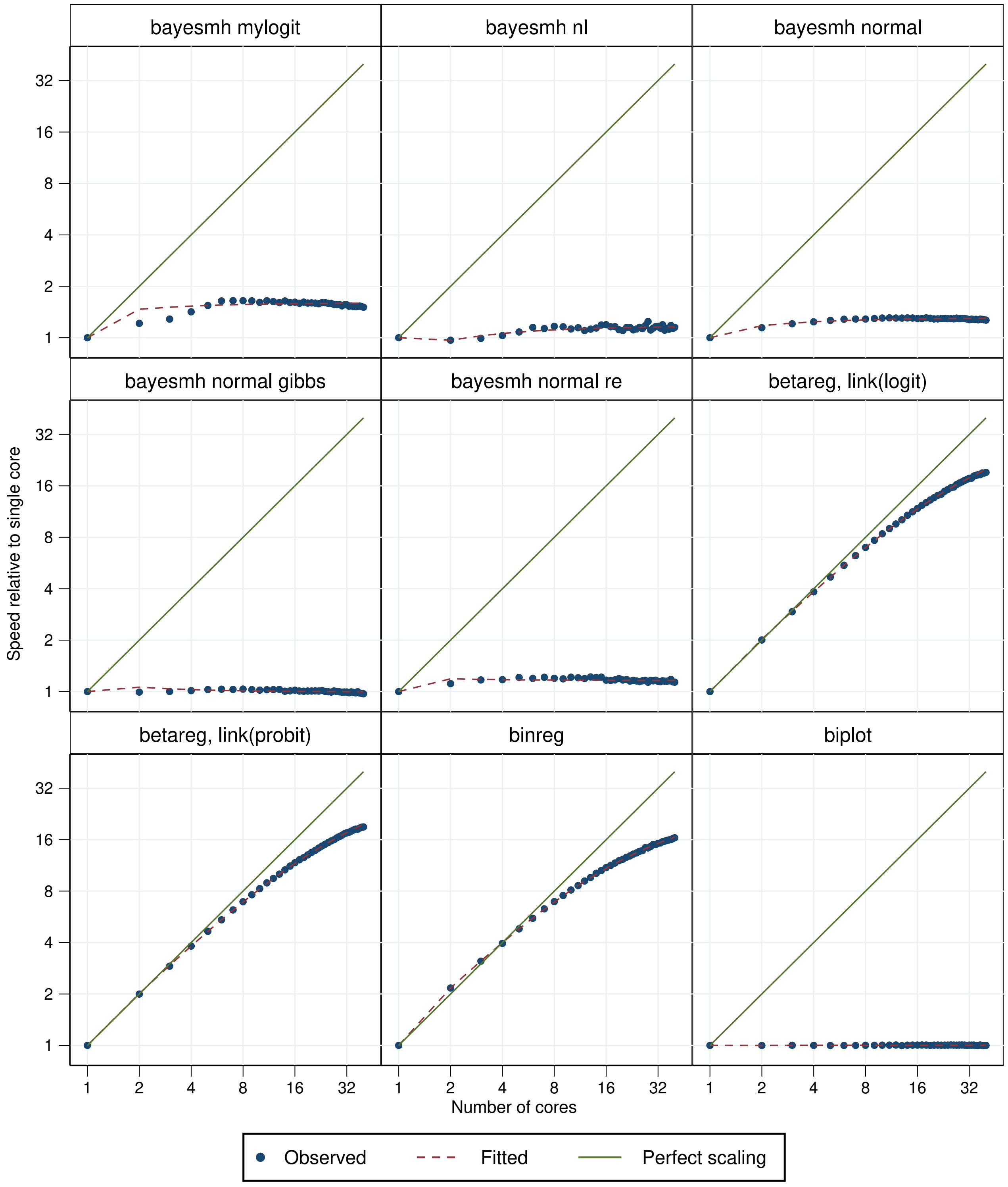


Figure 637. **Parallelization performance plots.**

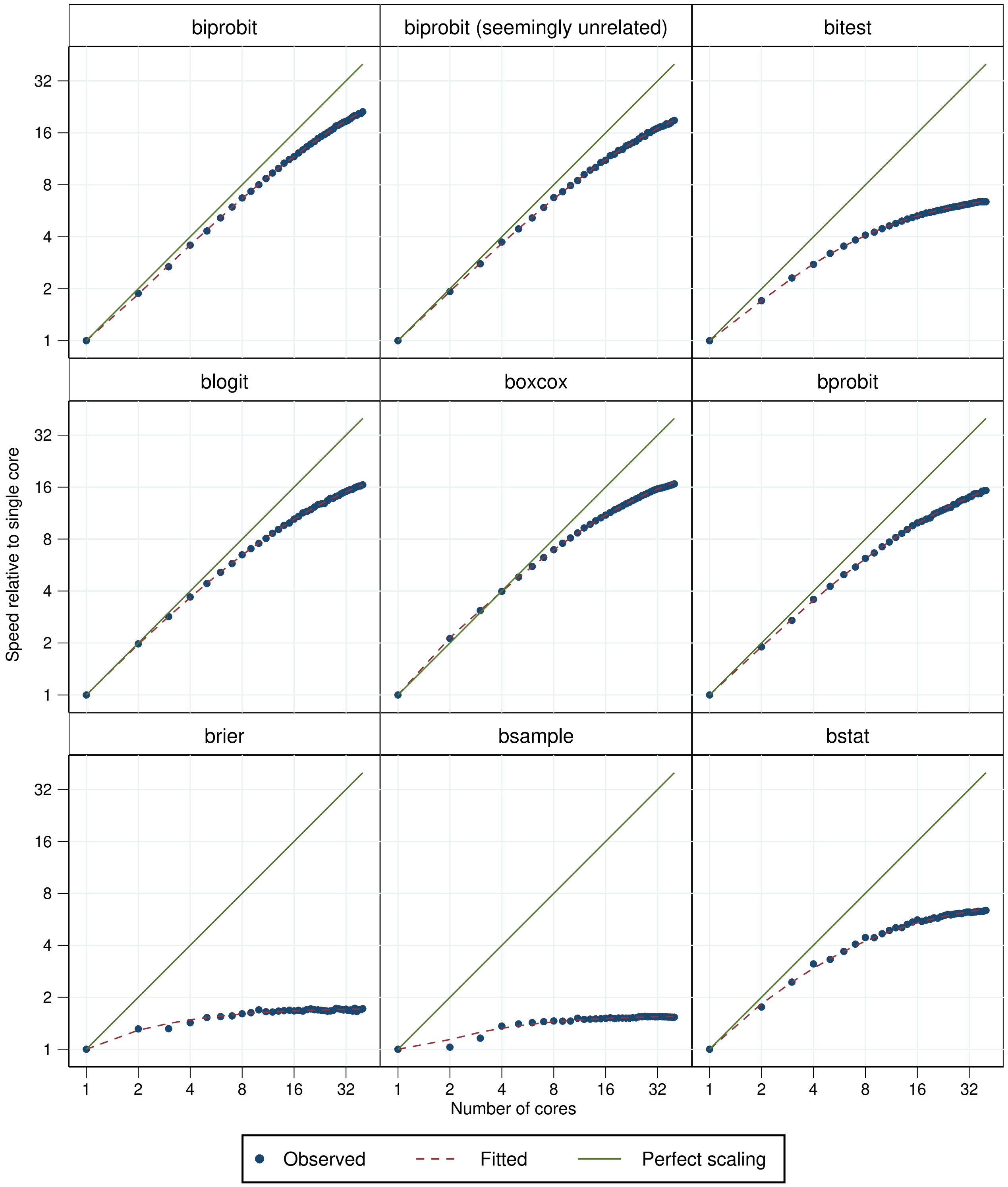


Figure 638. **Parallelization performance plots.**

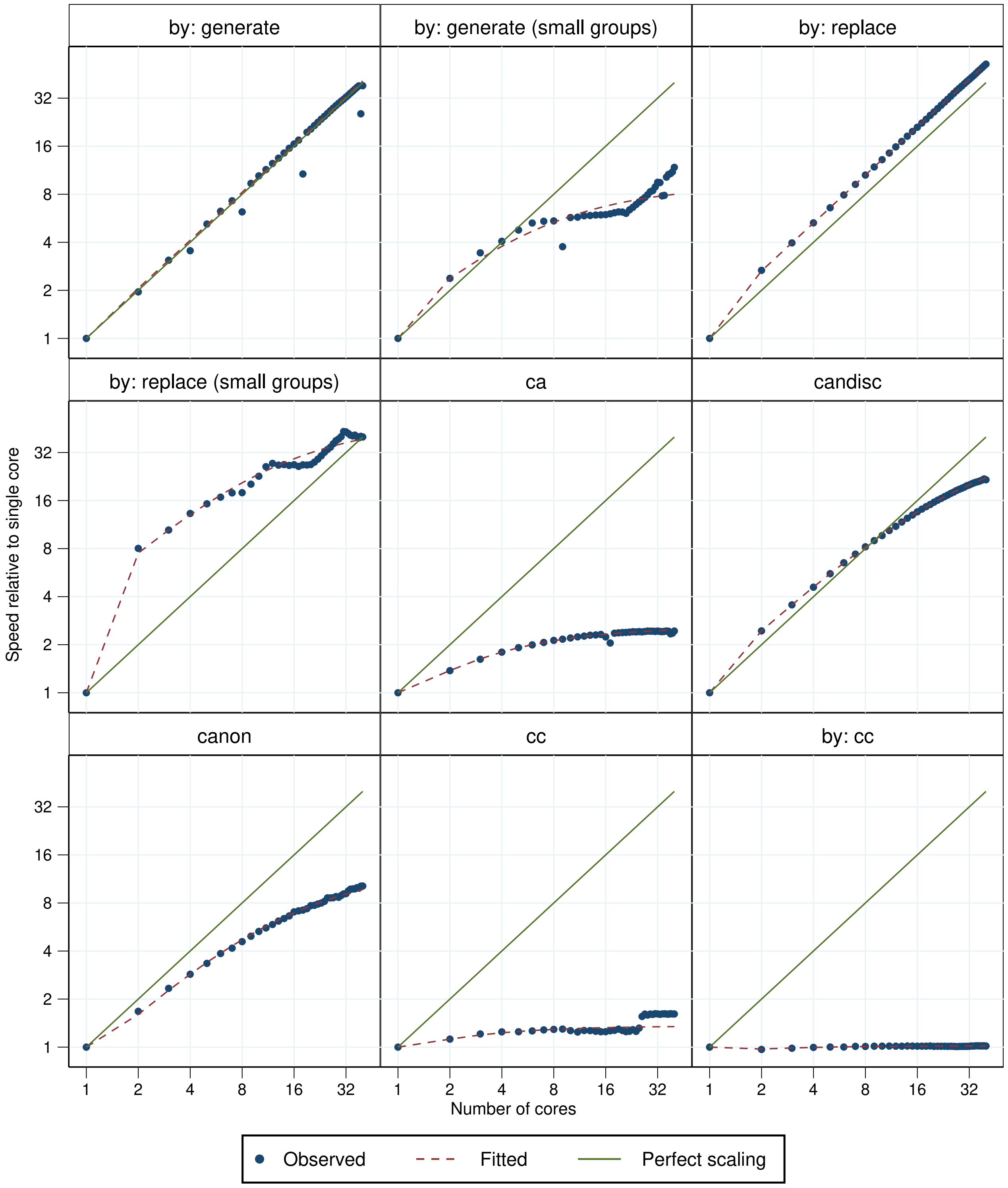


Figure 639. **Parallelization performance plots.**

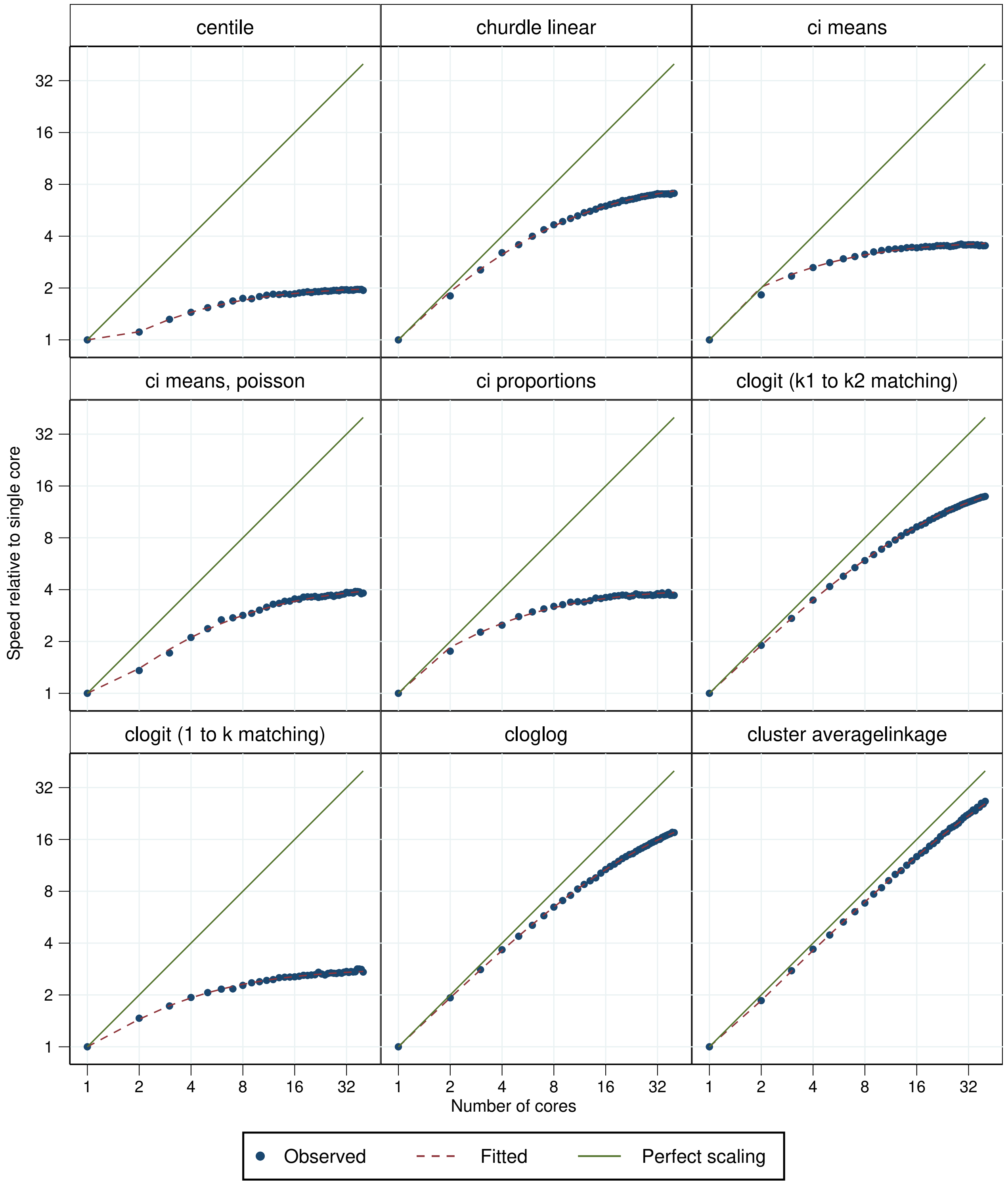


Figure 640. **Parallelization performance plots.**

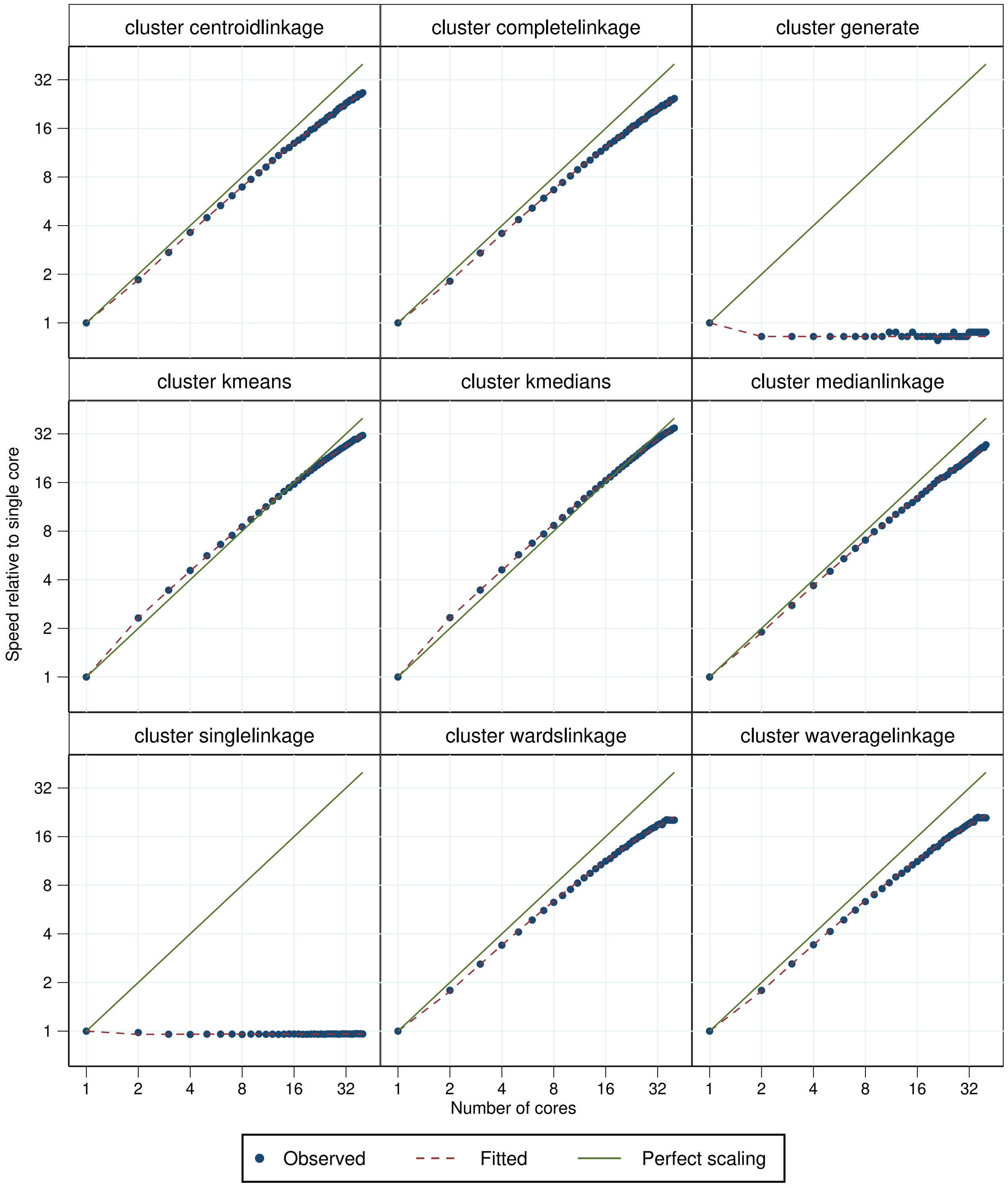


Figure 641. **Parallelization performance plots.**

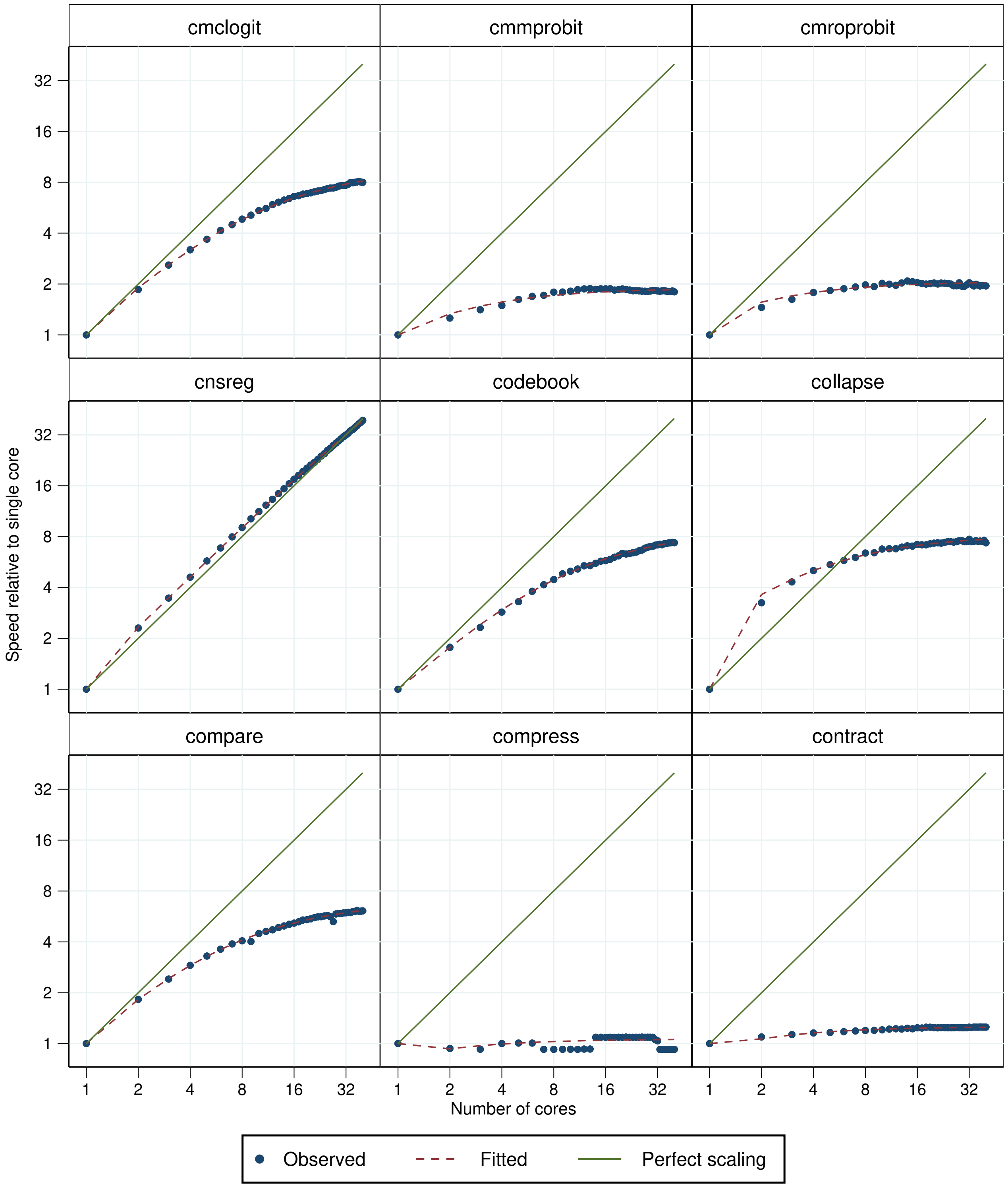


Figure 642. **Parallelization performance plots.**

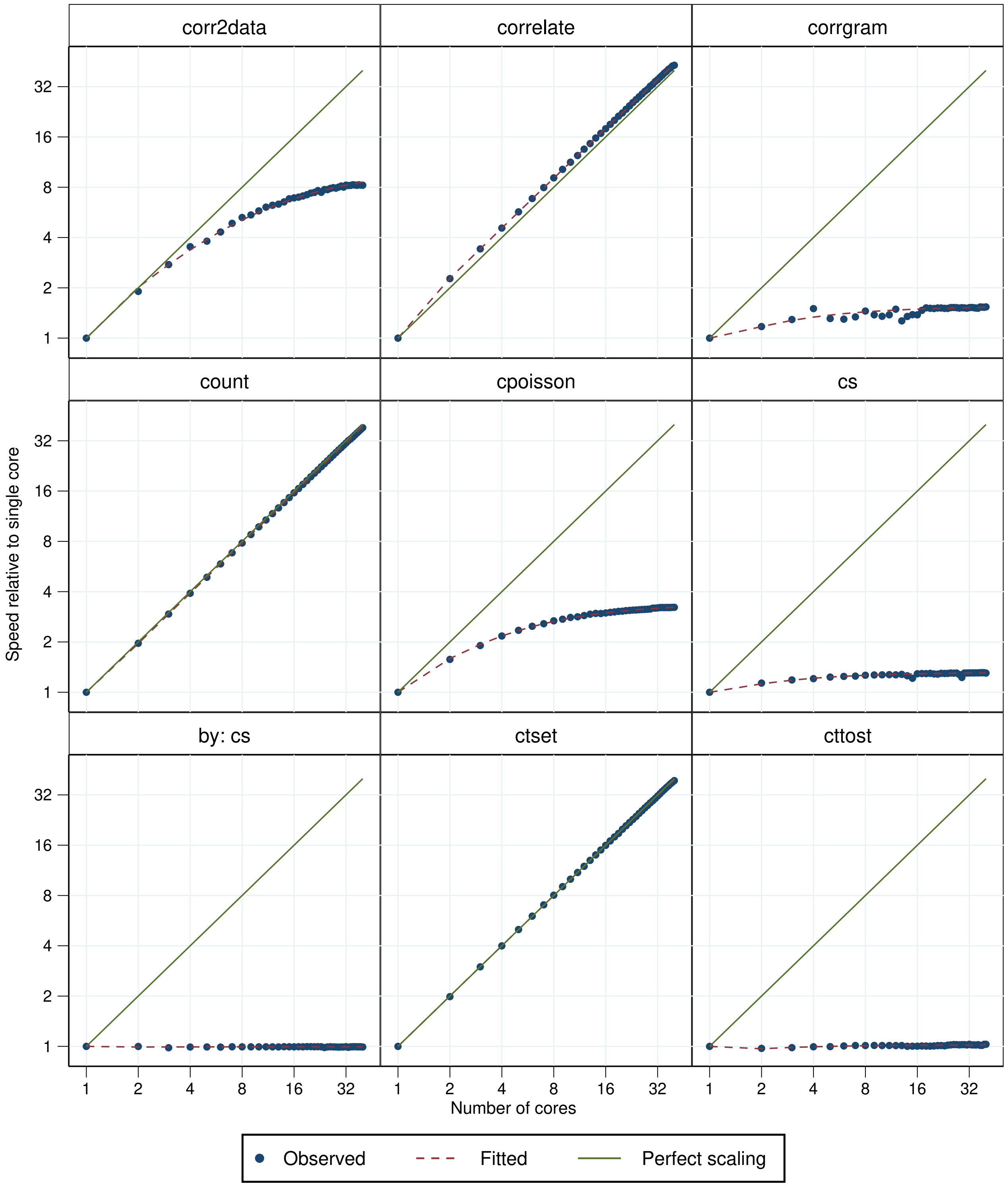


Figure 643. **Parallelization performance plots.**

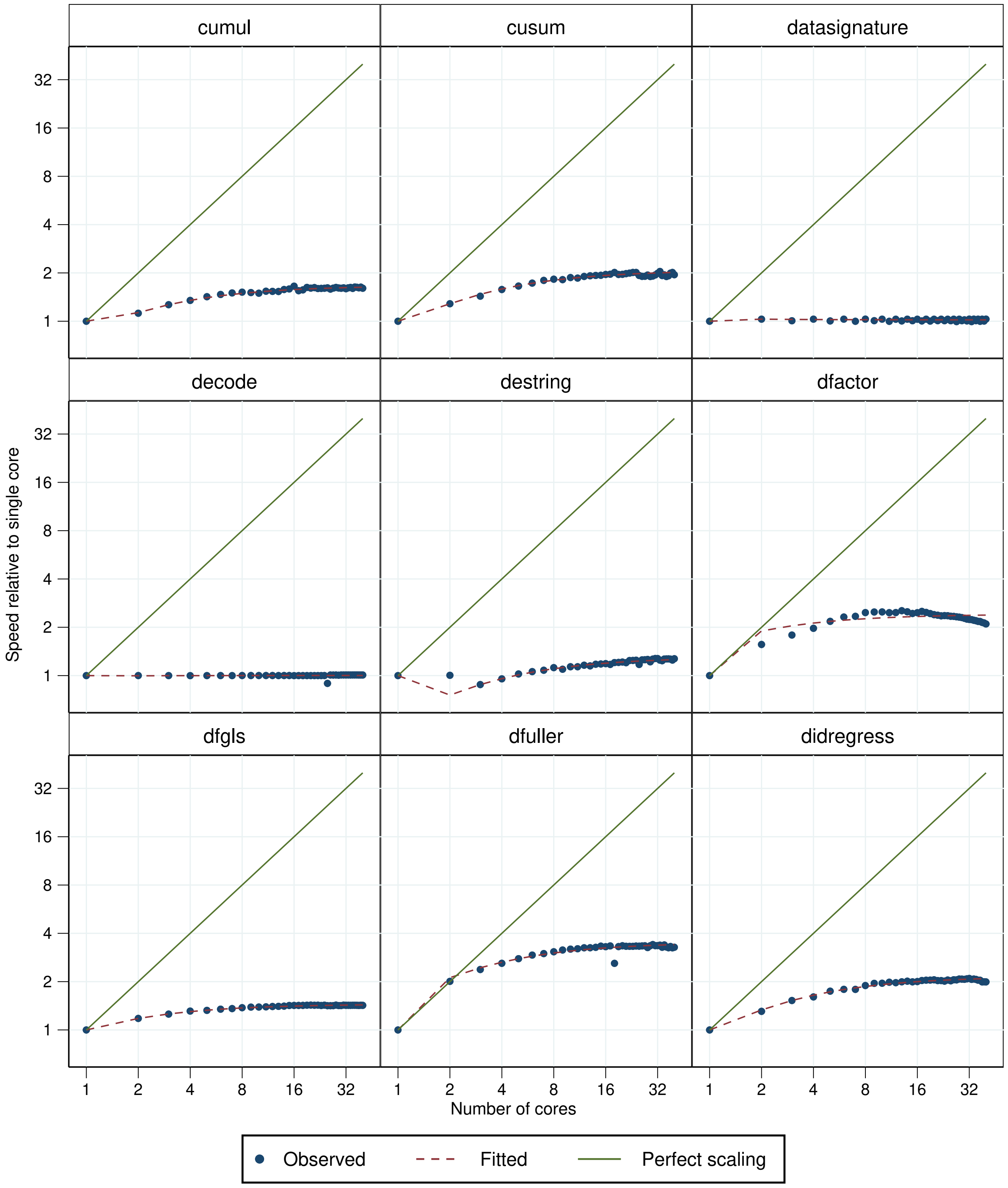


Figure 644. **Parallelization performance plots.**

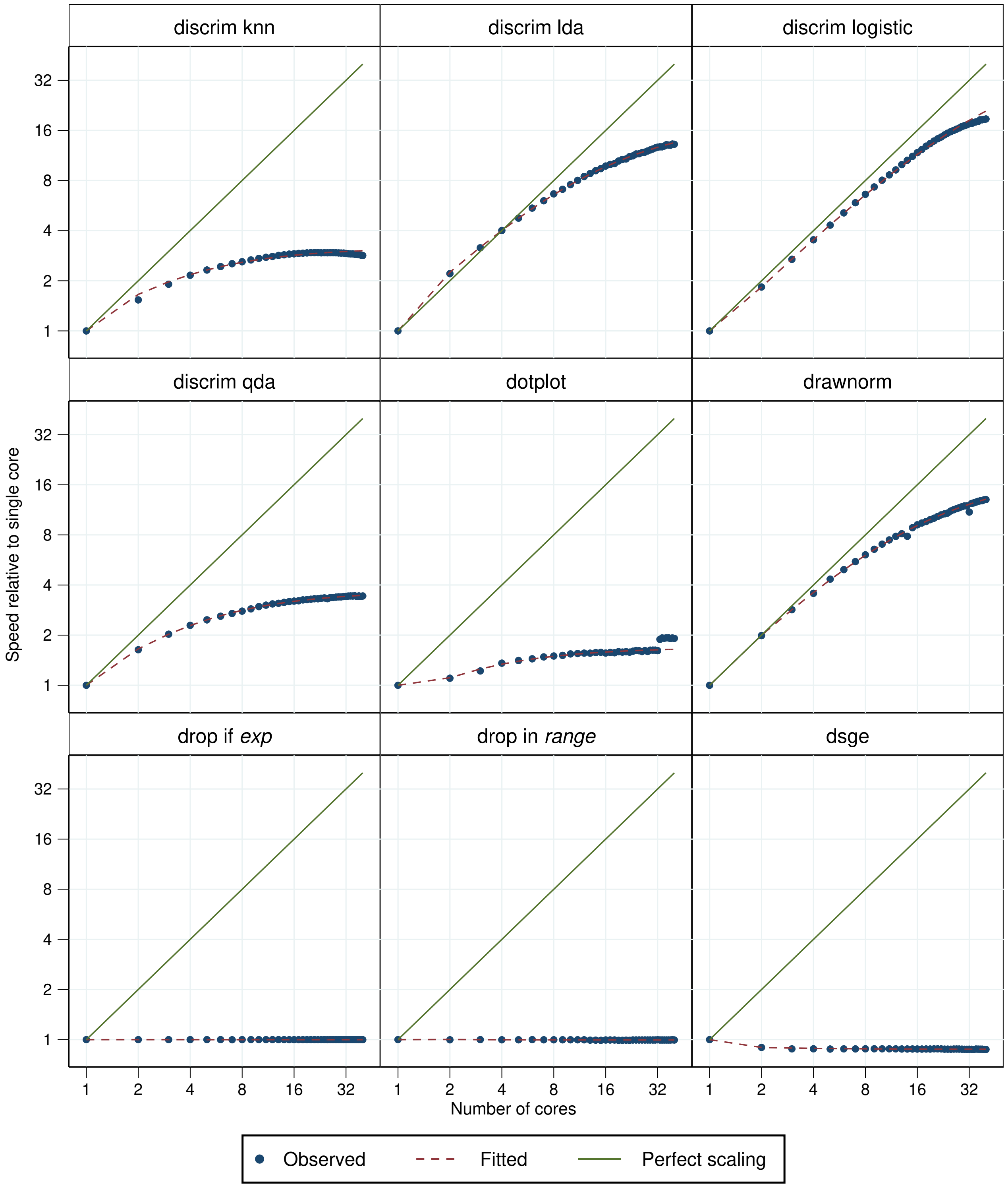


Figure 645. **Parallelization performance plots.**

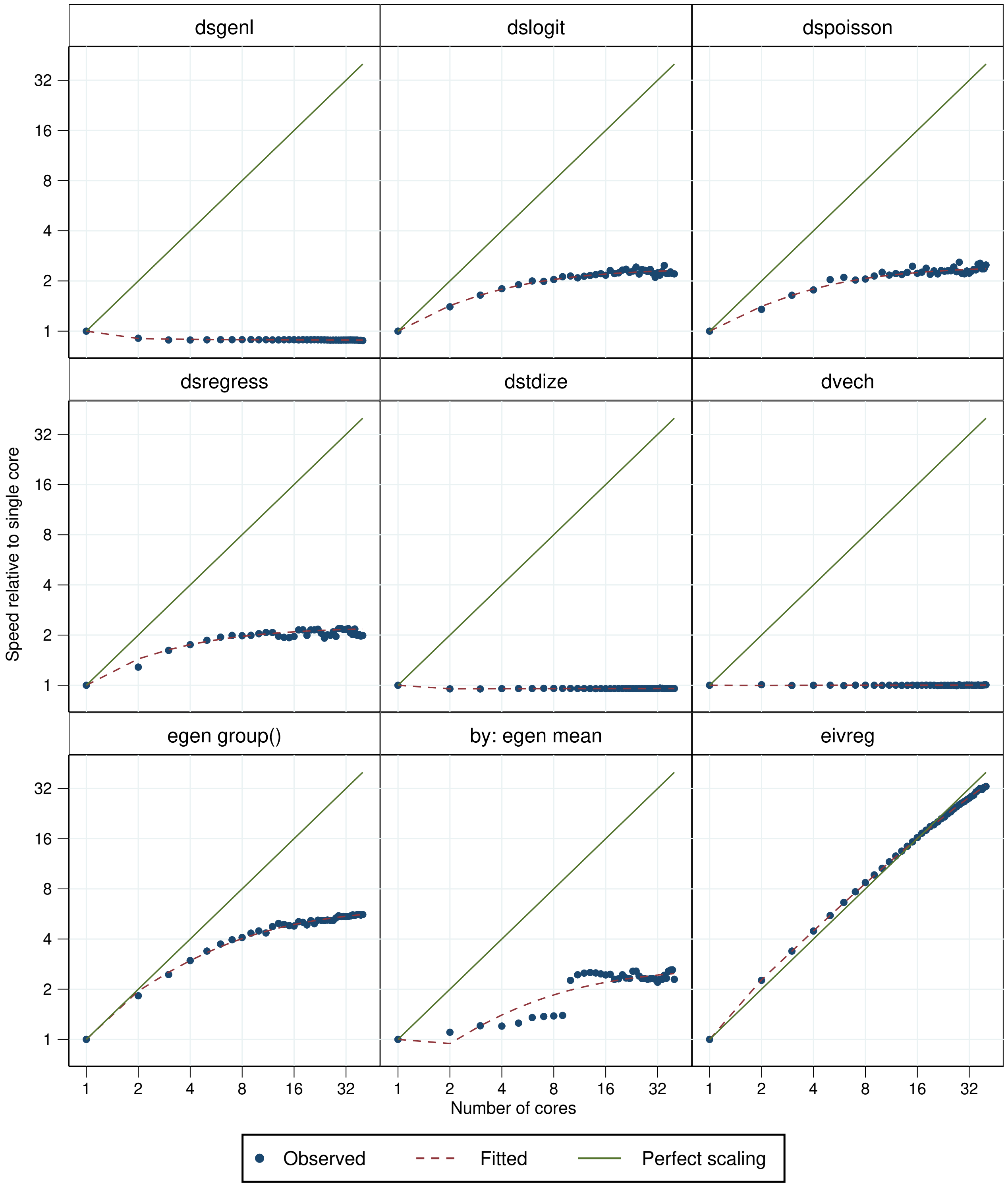


Figure 646. **Parallelization performance plots.**

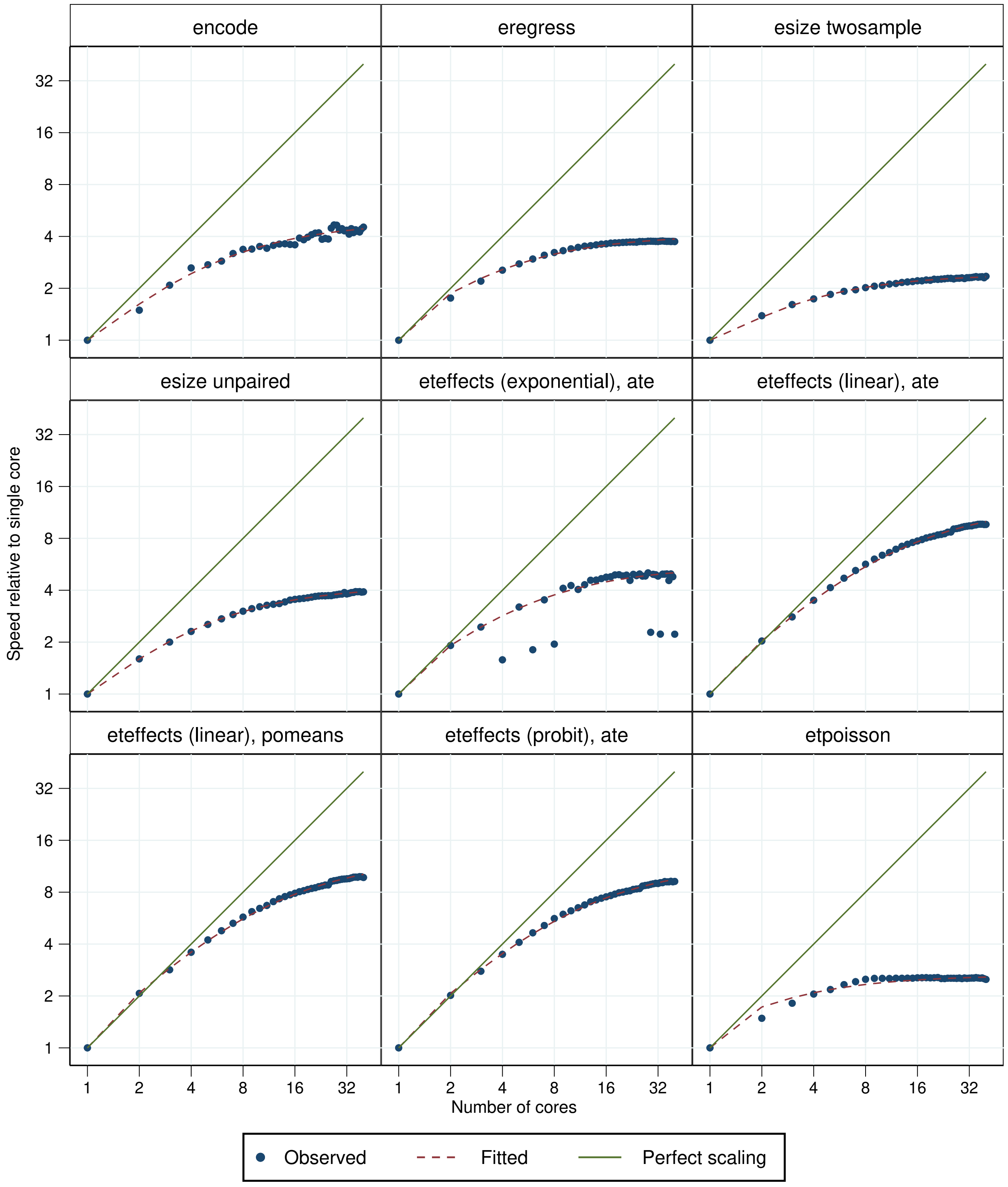


Figure 647. **Parallelization performance plots.**

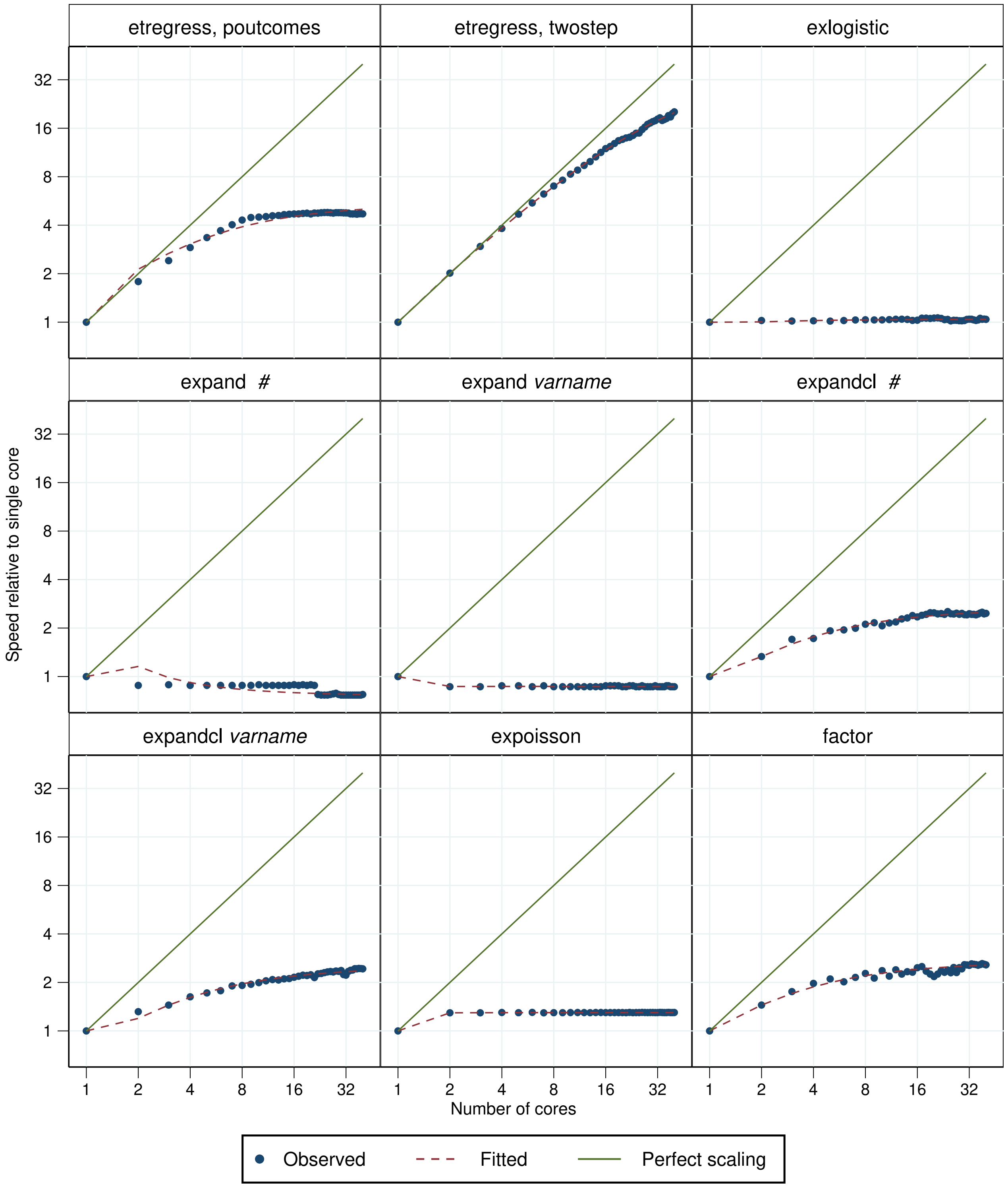


Figure 648. **Parallelization performance plots.**

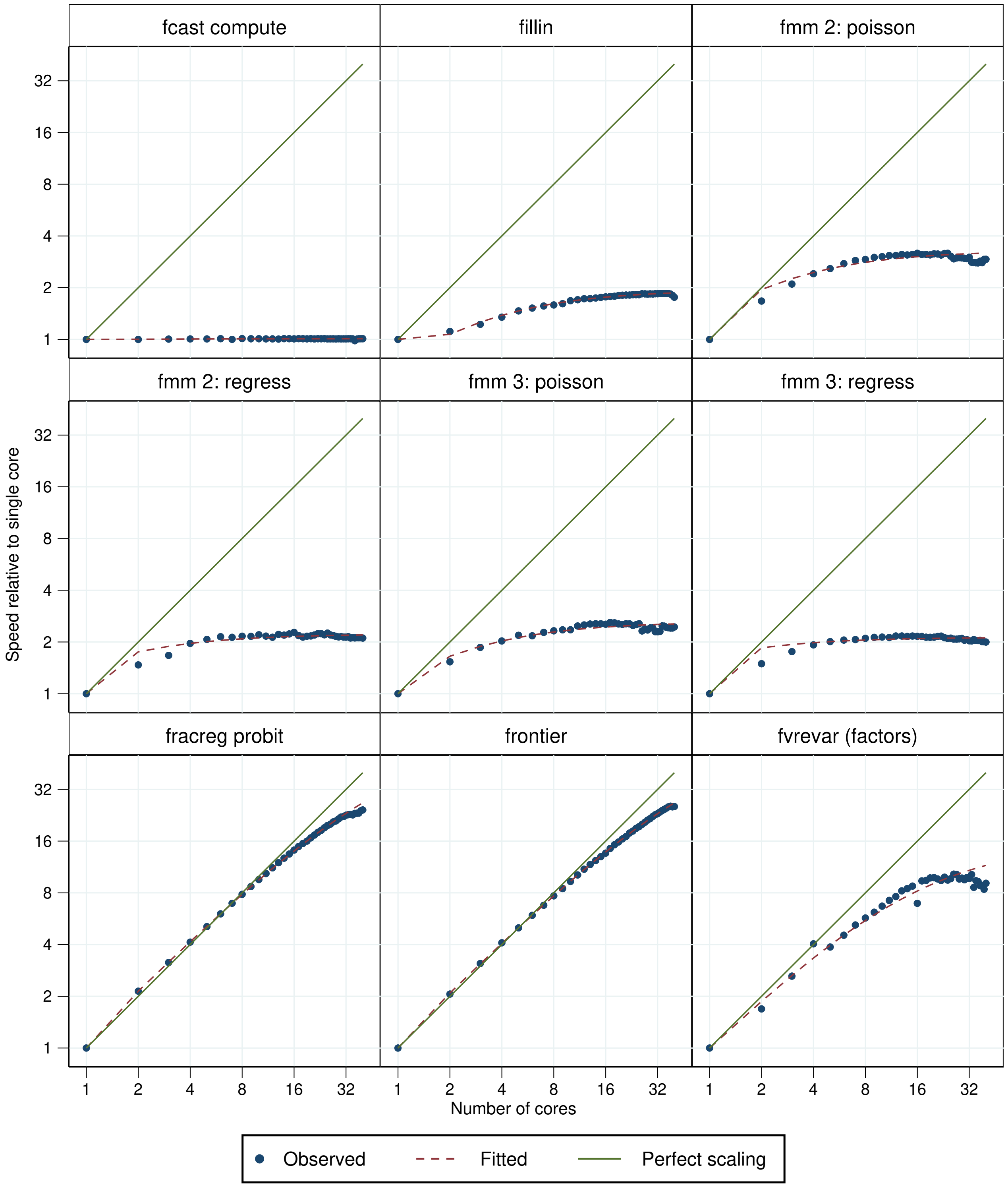


Figure 649. **Parallelization performance plots.**

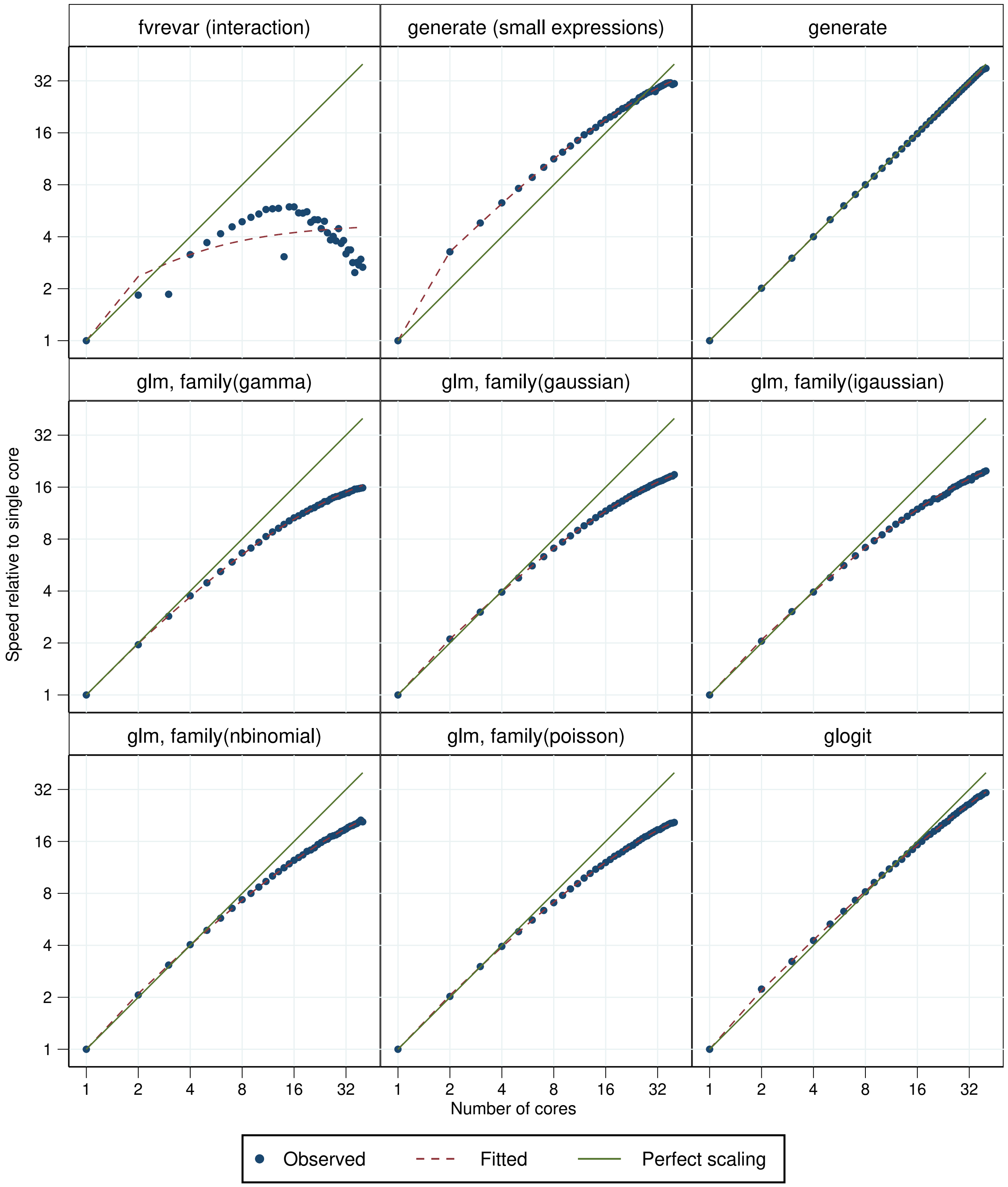


Figure 650. **Parallelization performance plots.**

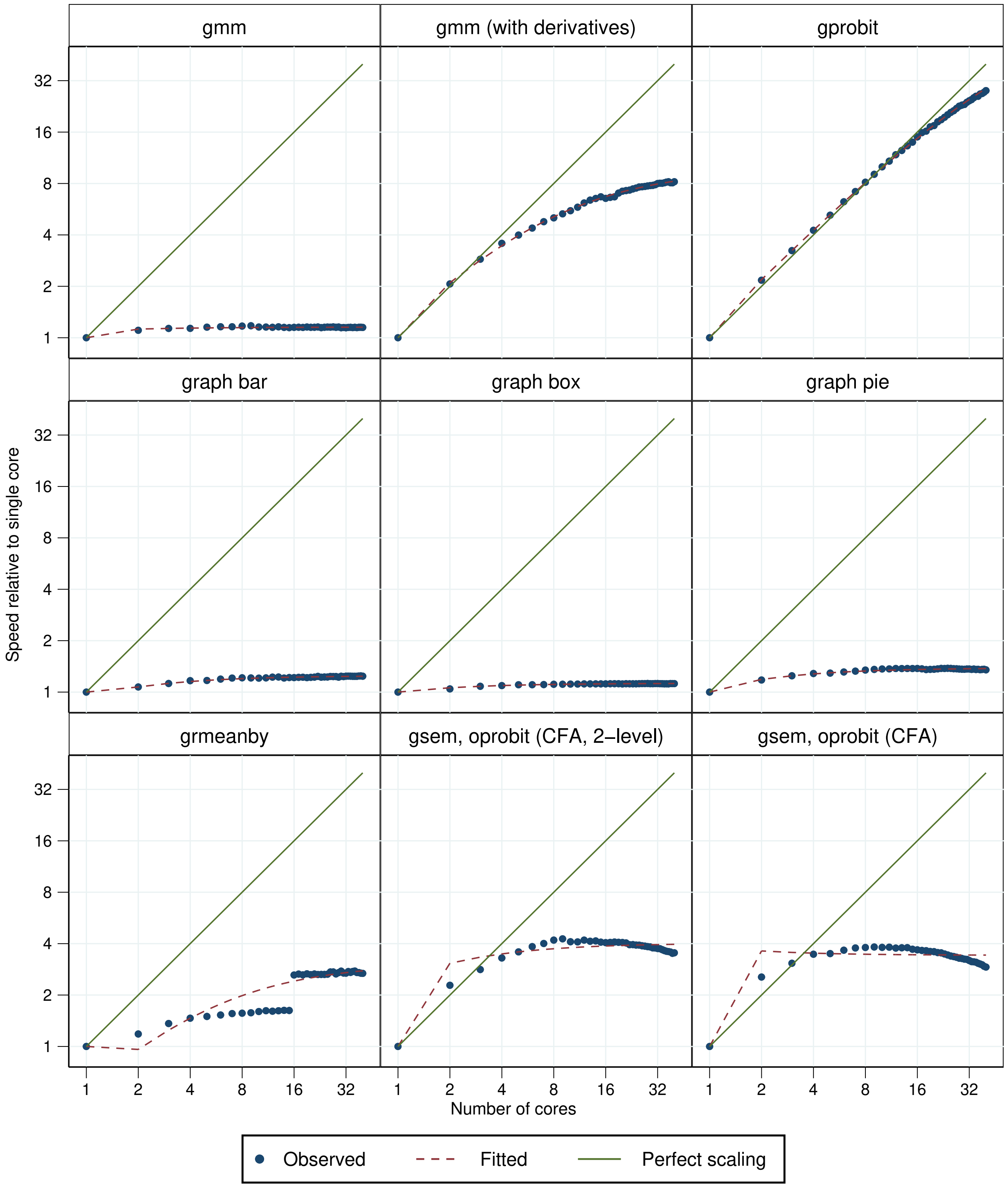


Figure 651. **Parallelization performance plots.**

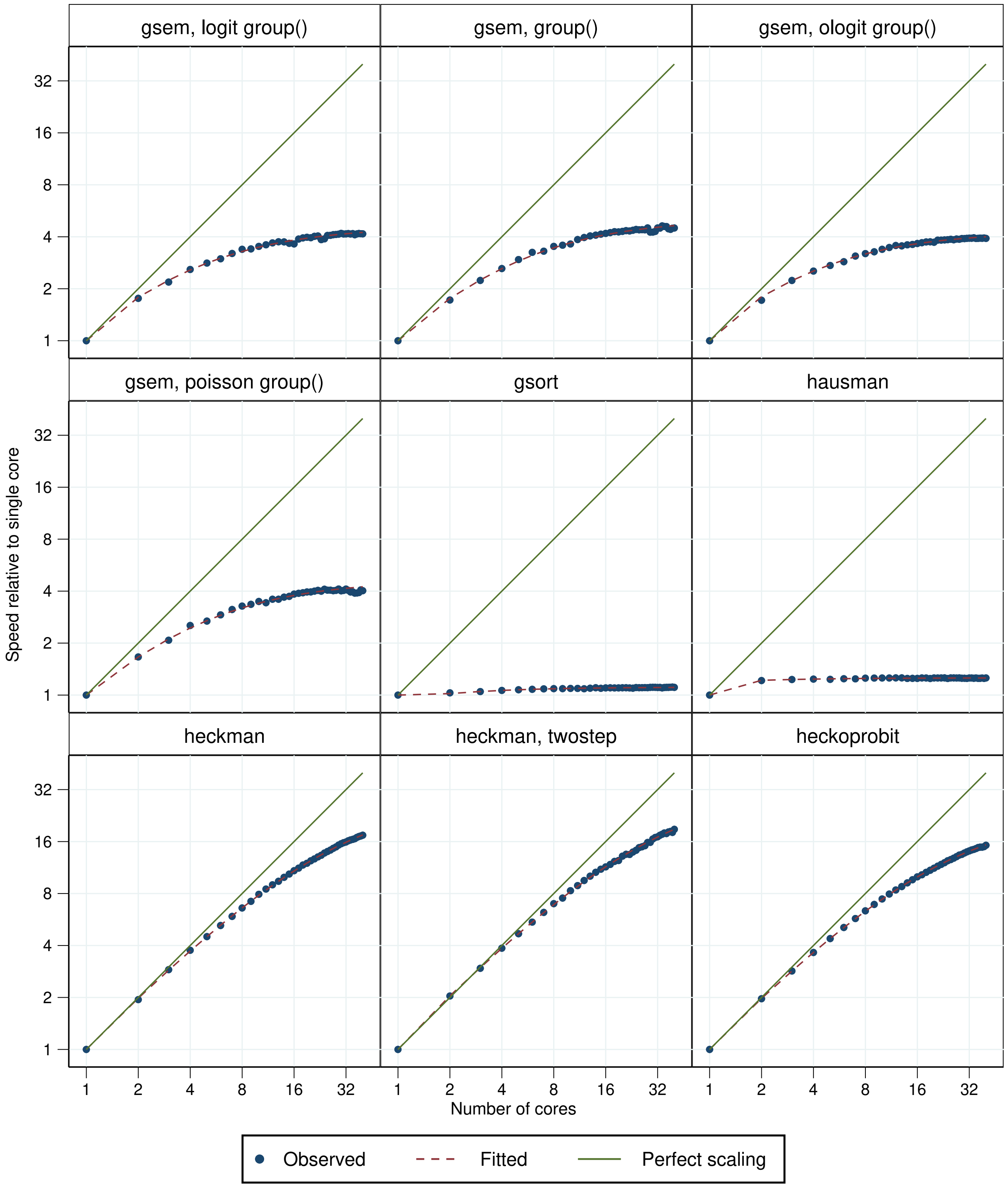


Figure 652. **Parallelization performance plots.**

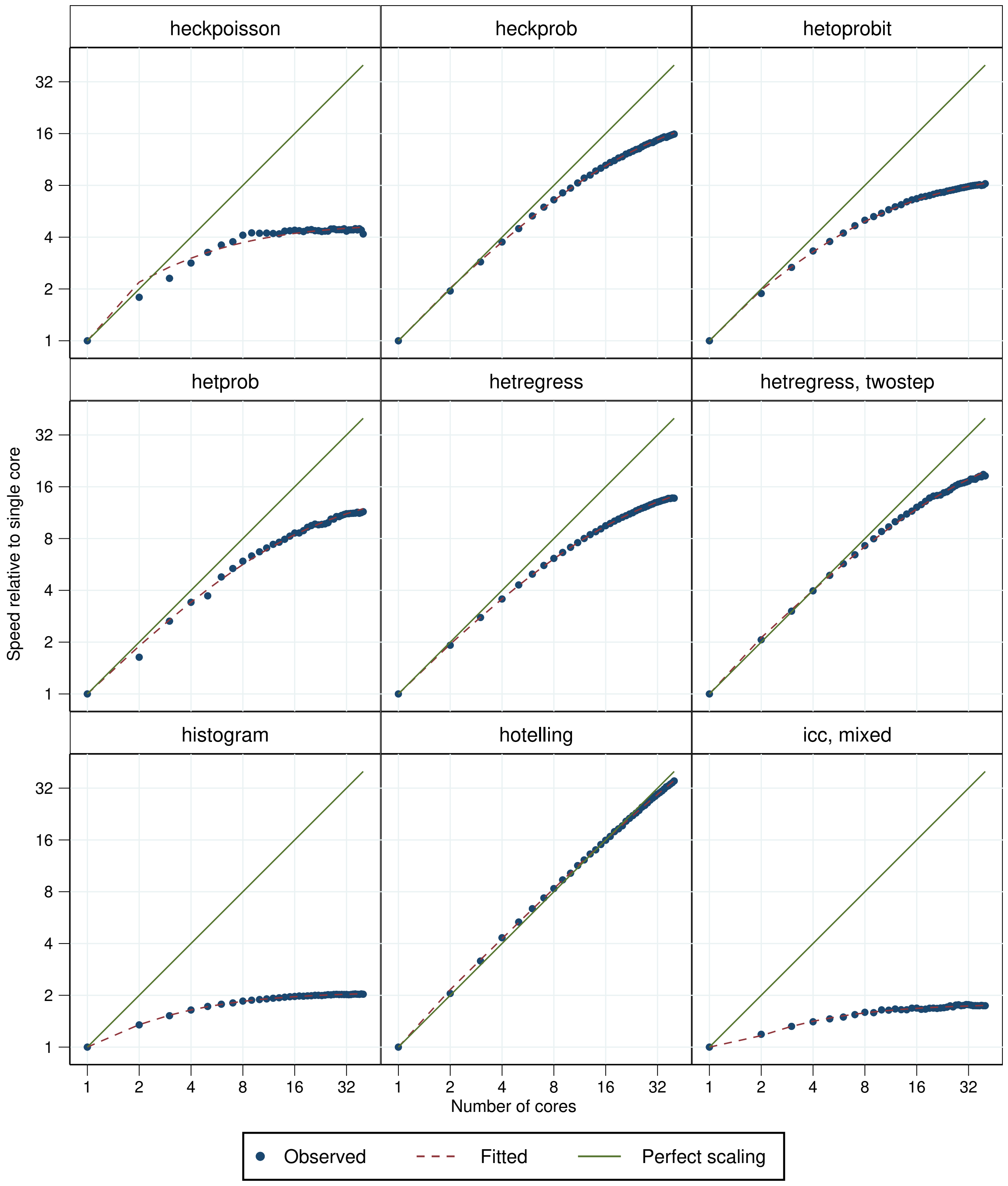


Figure 653. **Parallelization performance plots.**

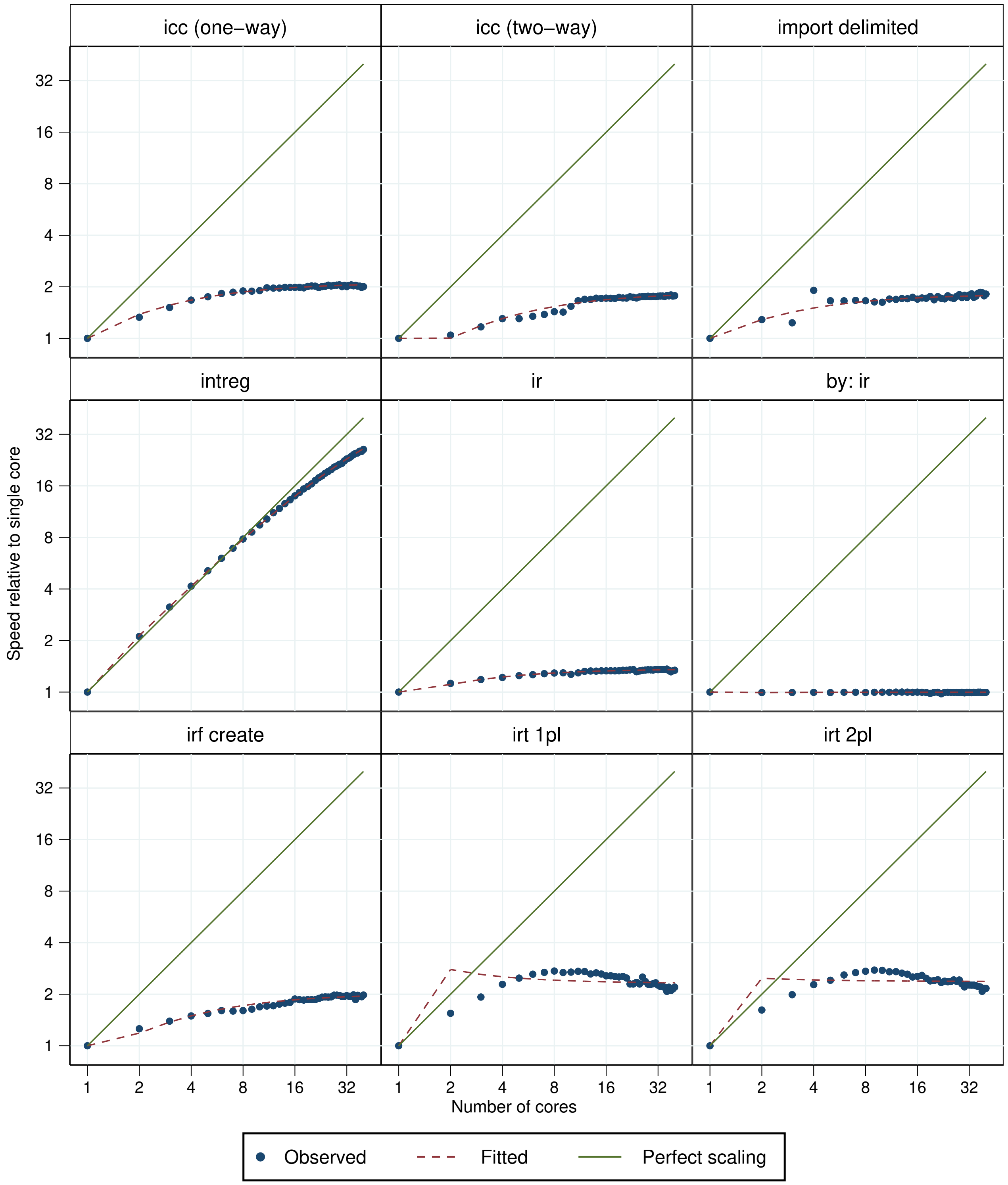


Figure 654. **Parallelization performance plots.**

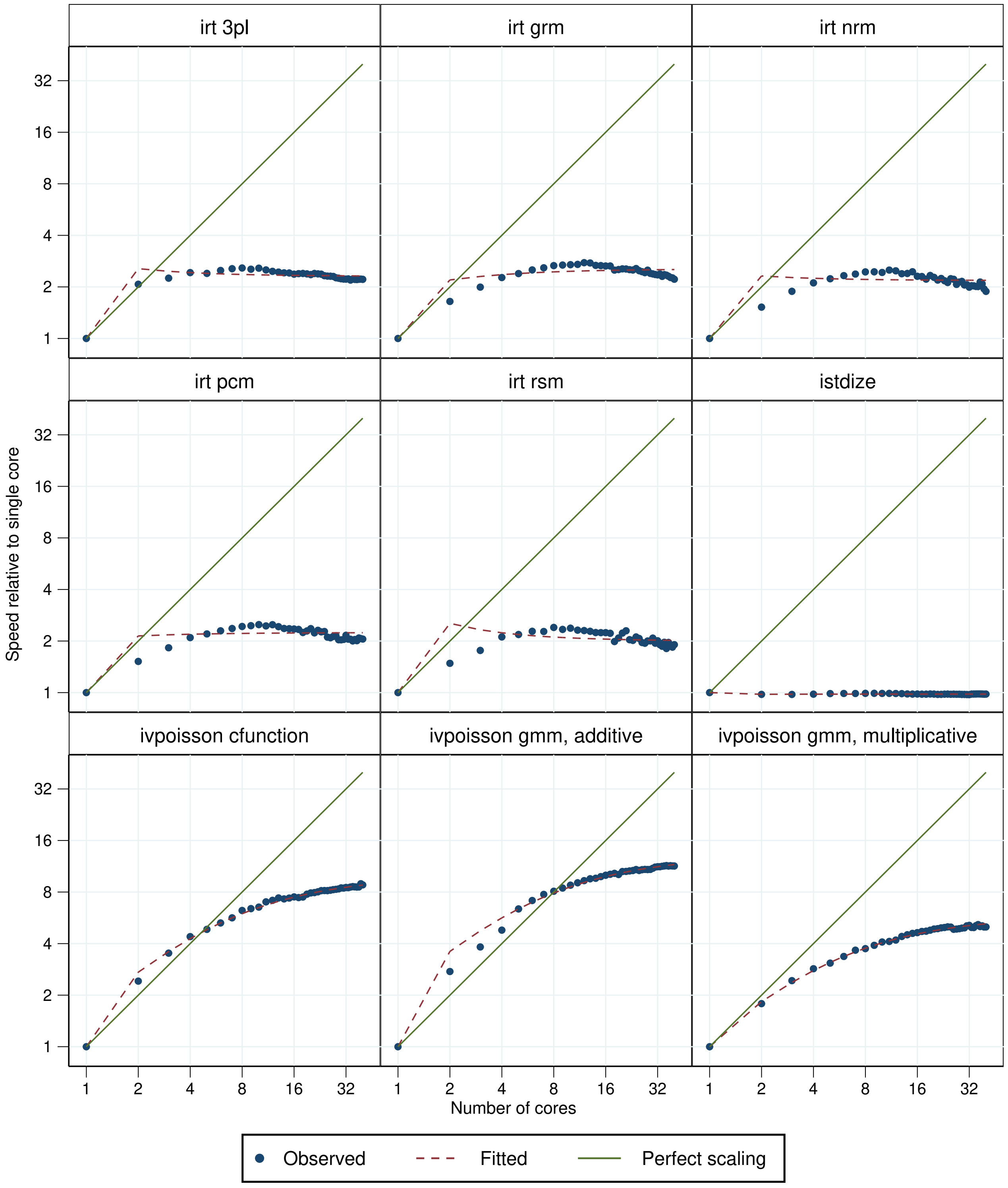


Figure 655. **Parallelization performance plots.**

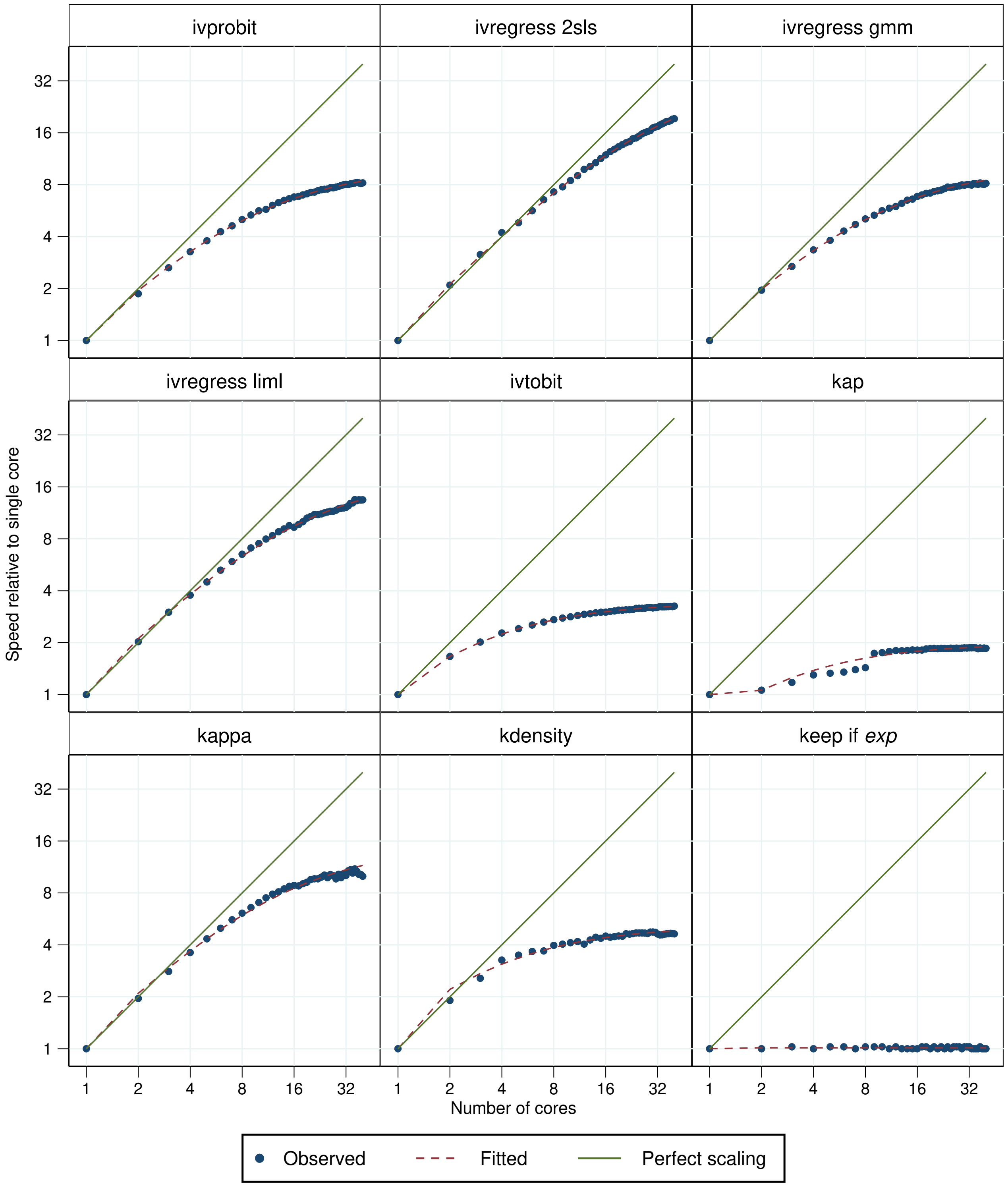


Figure 656. **Parallelization performance plots.**

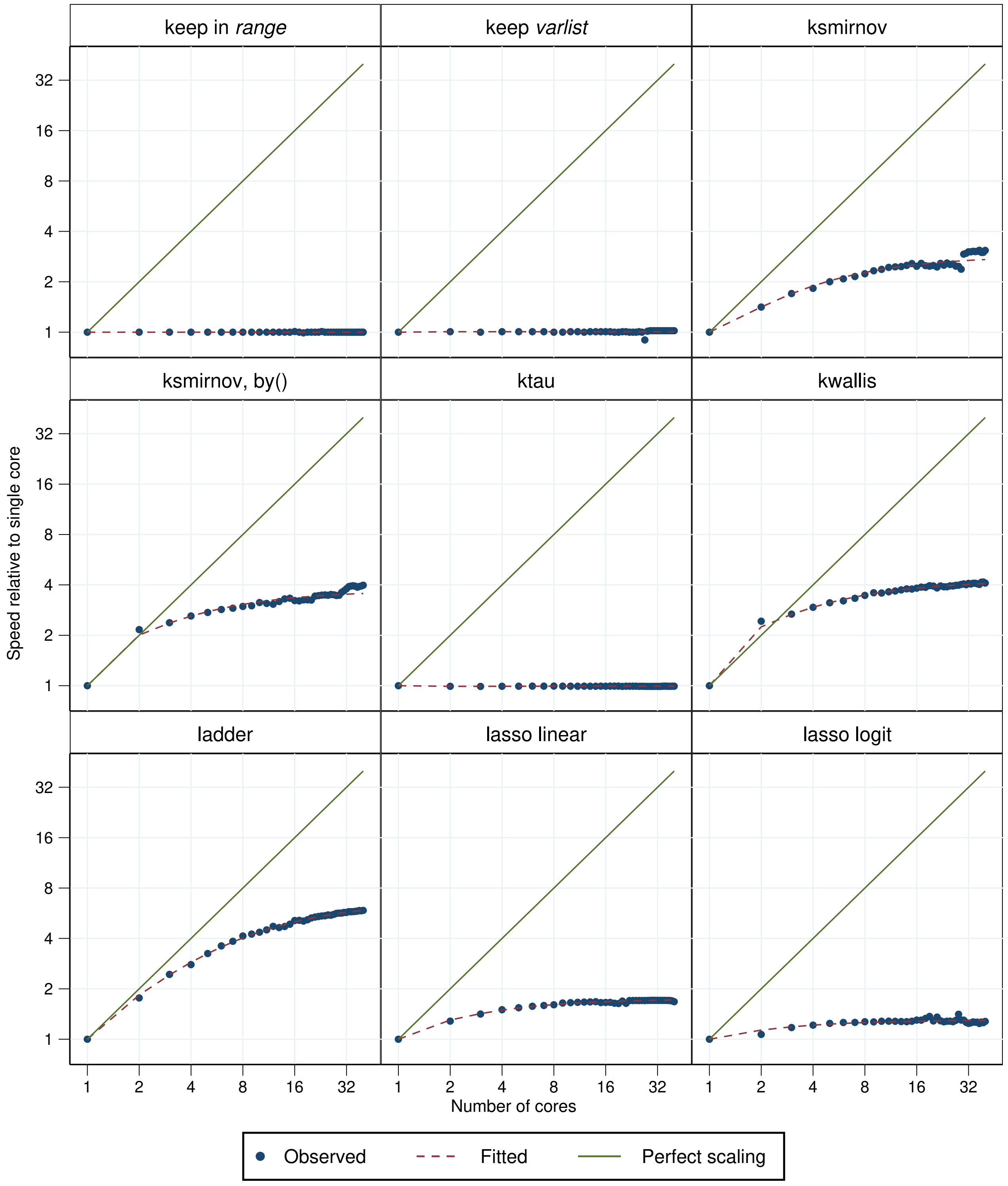


Figure 657. **Parallelization performance plots.**

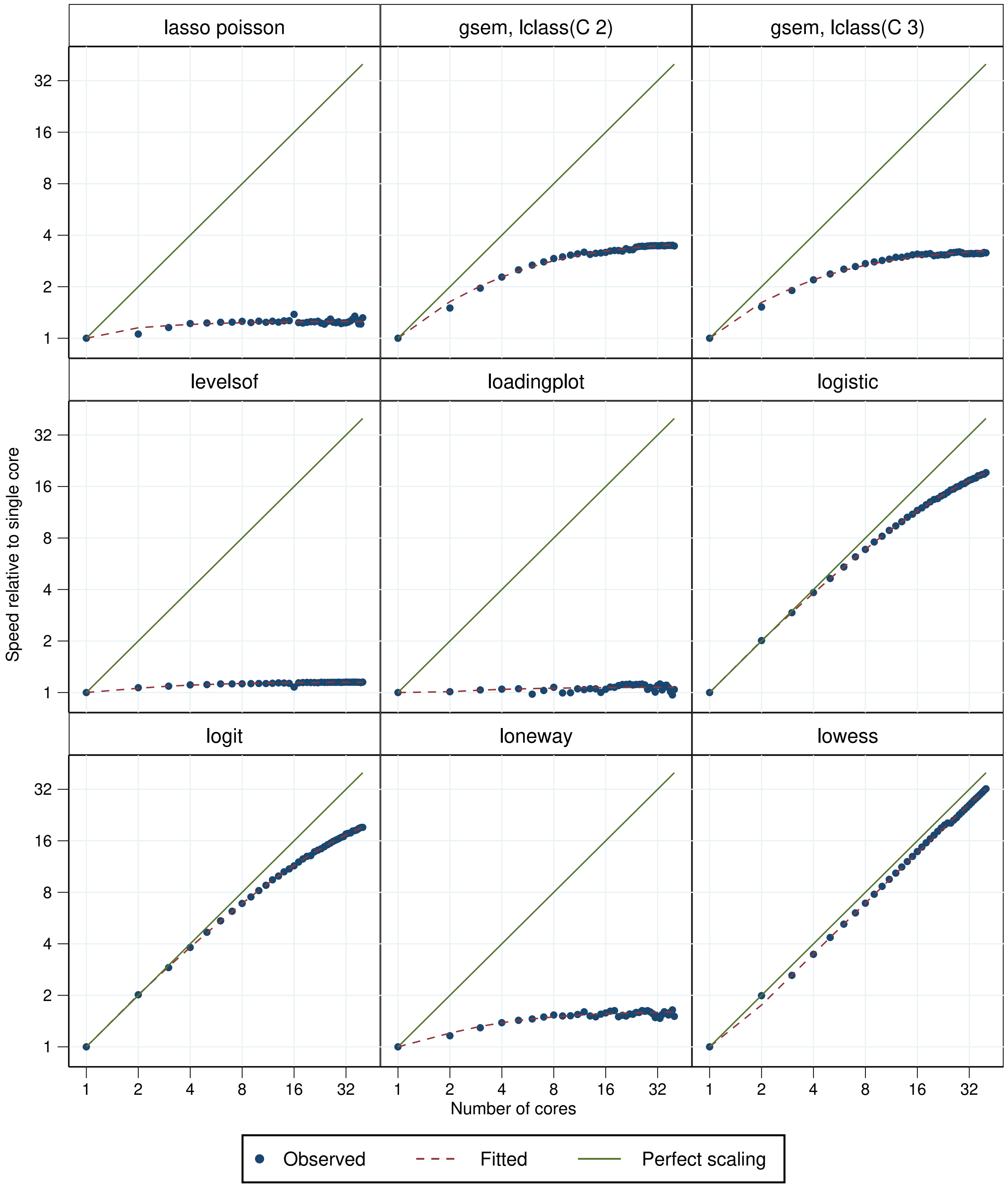


Figure 658. **Parallelization performance plots.**

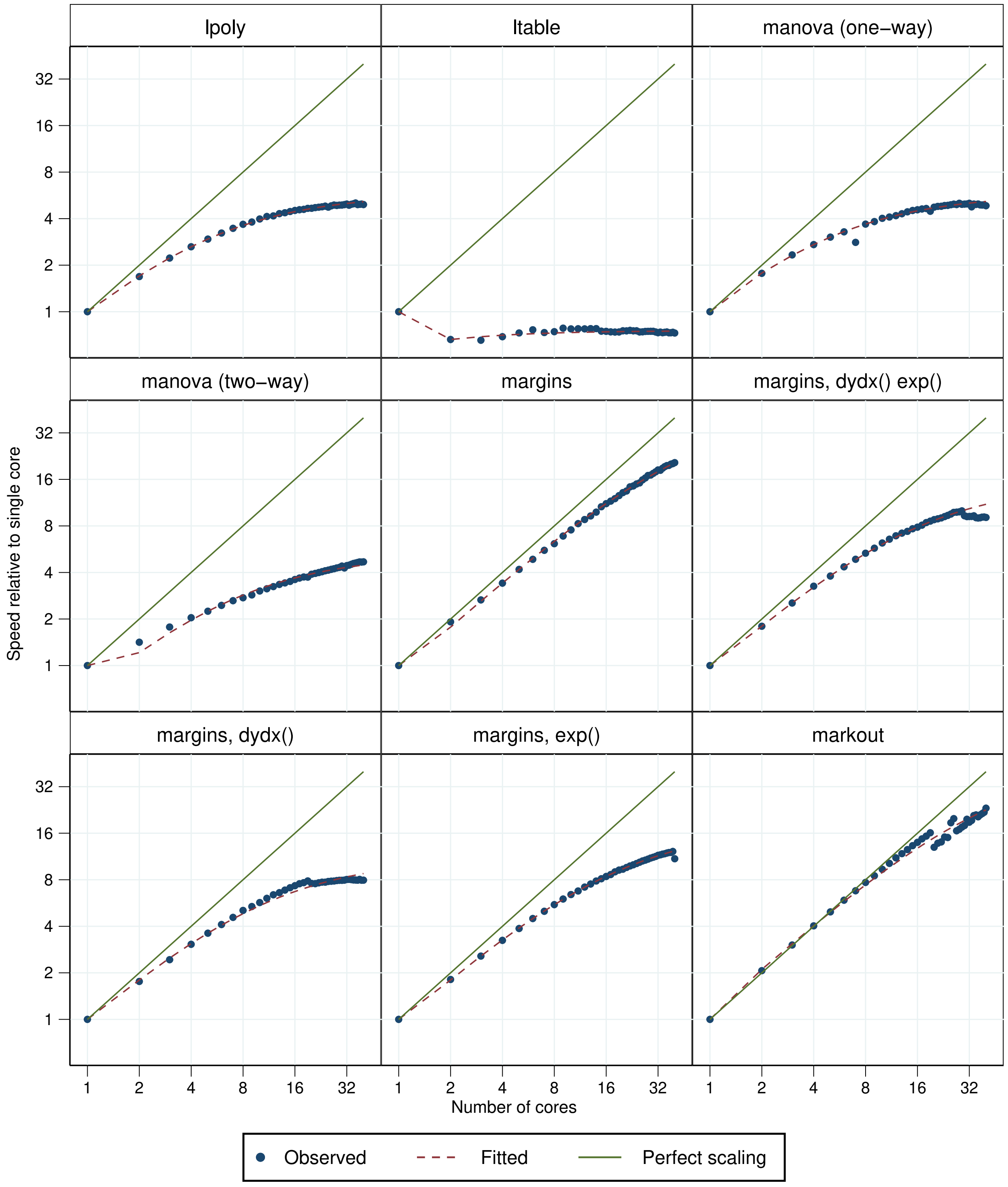


Figure 659. **Parallelization performance plots.**

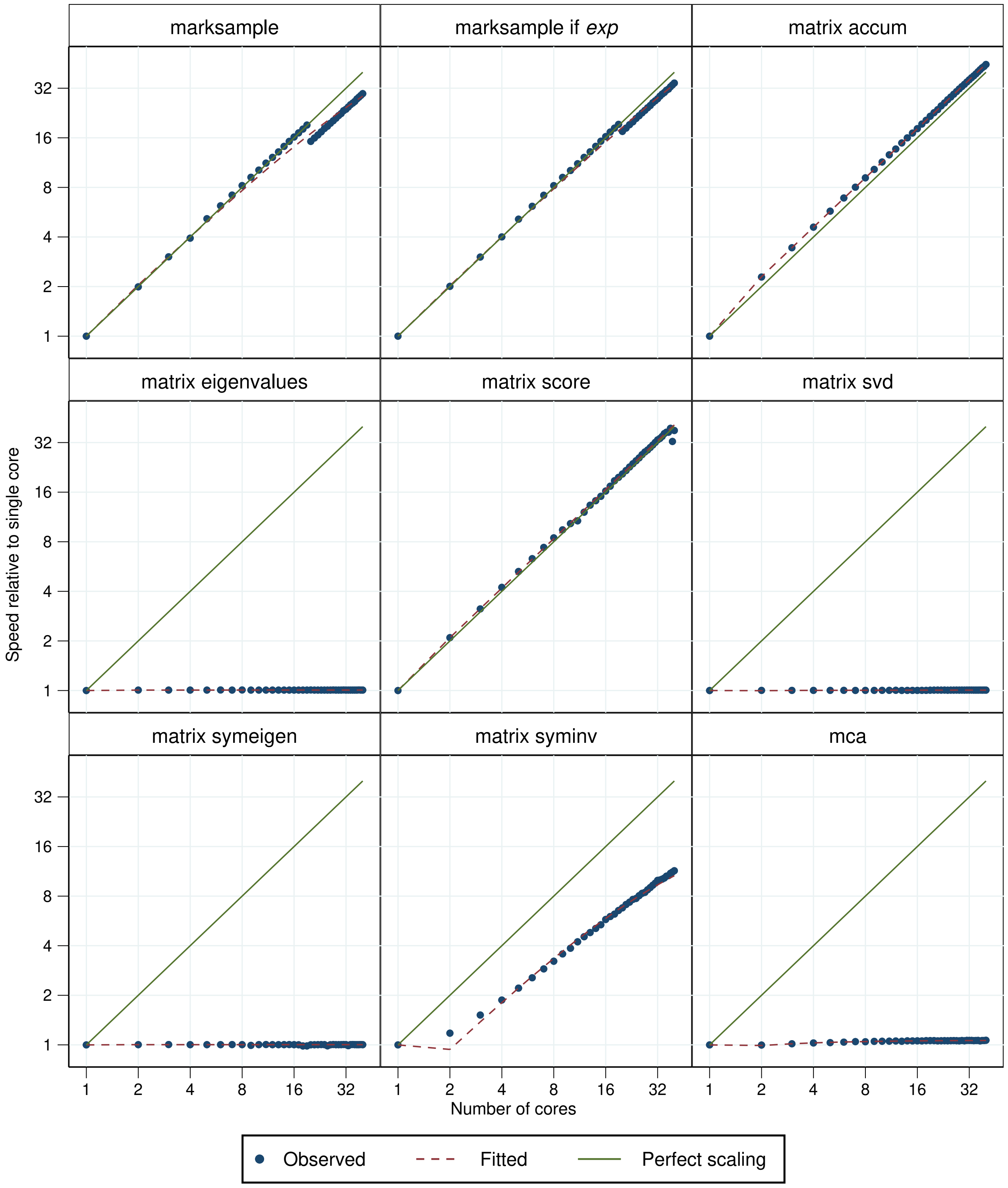


Figure 660. **Parallelization performance plots.**

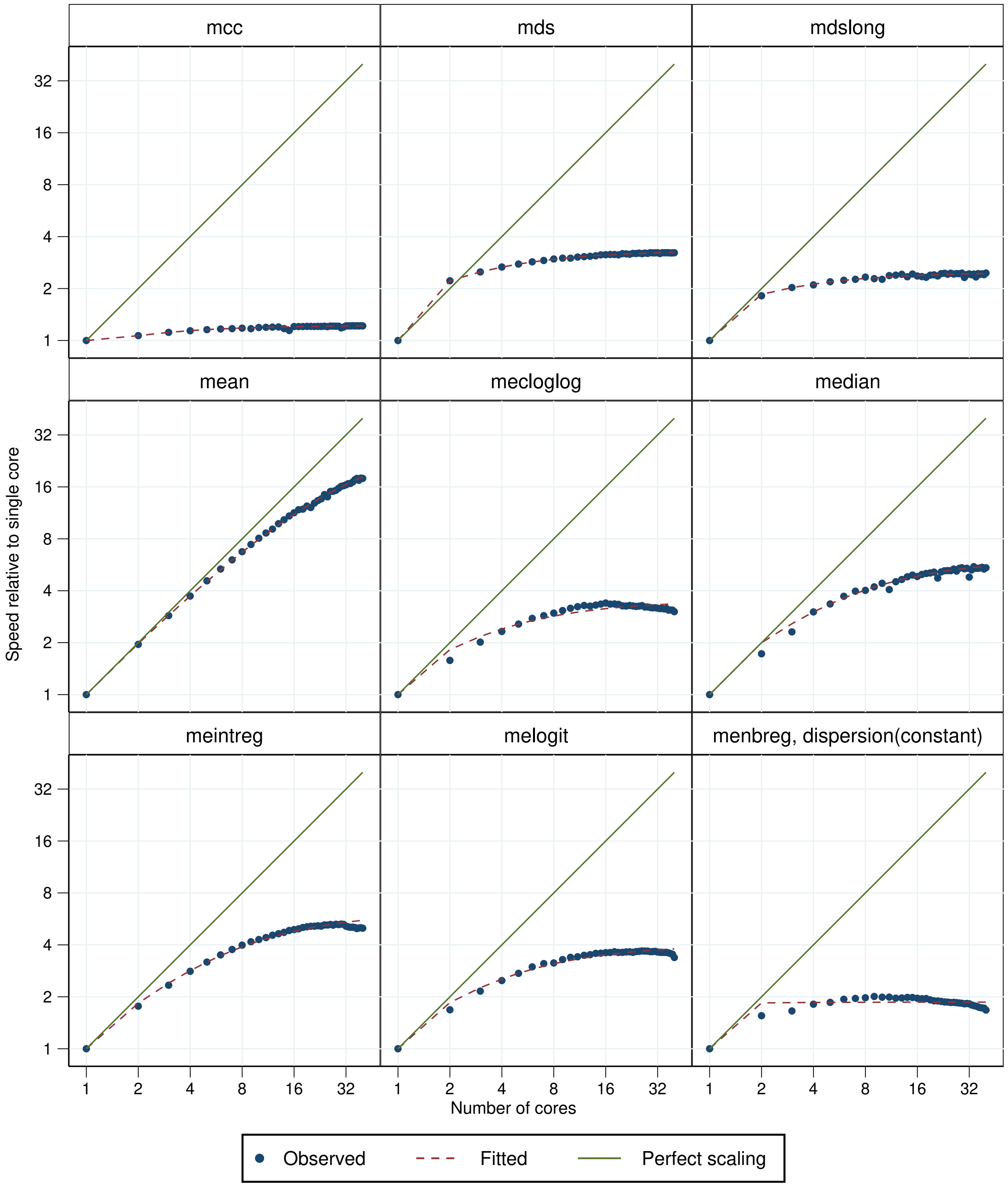


Figure 661. **Parallelization performance plots.**

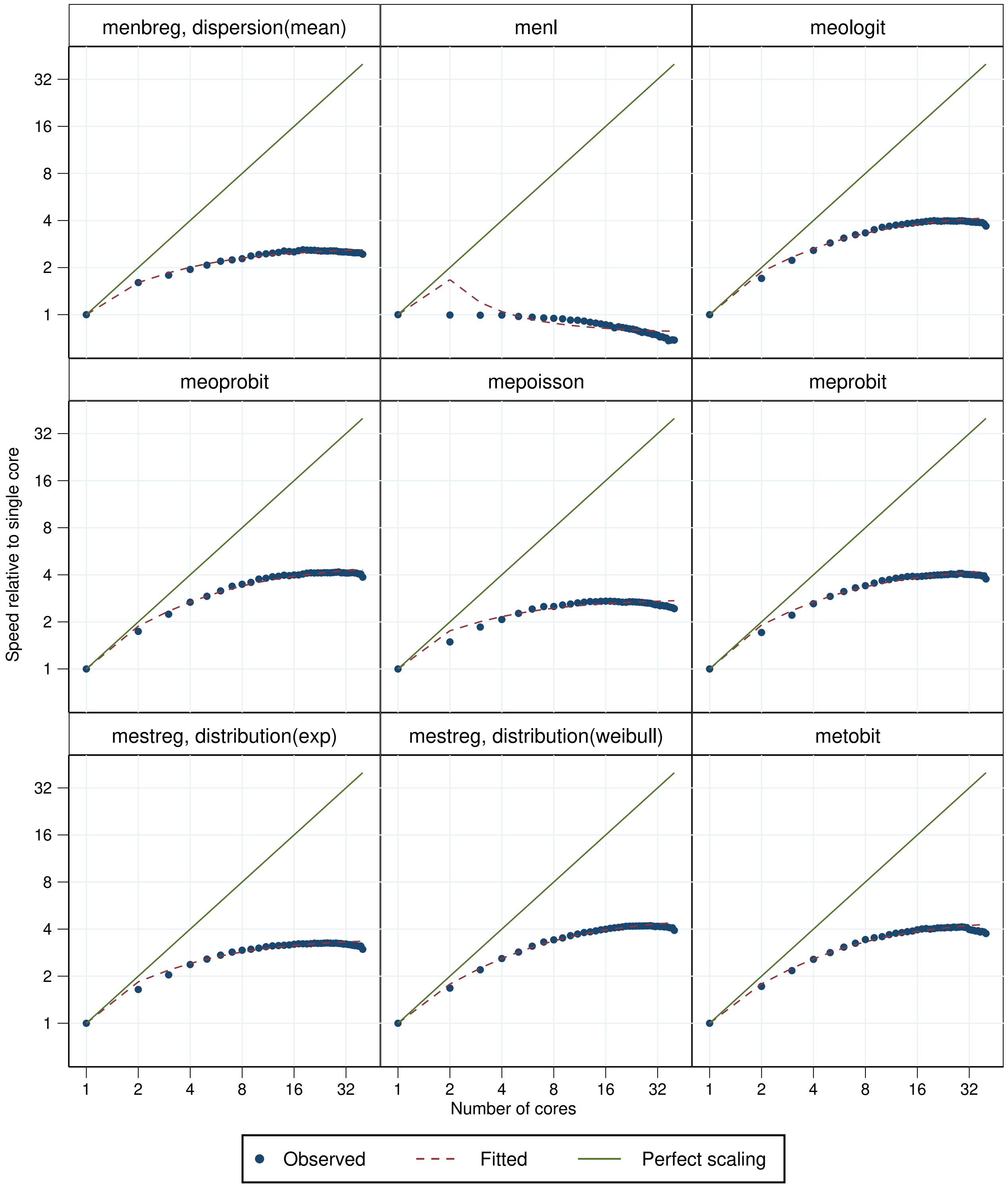


Figure 662. **Parallelization performance plots.**

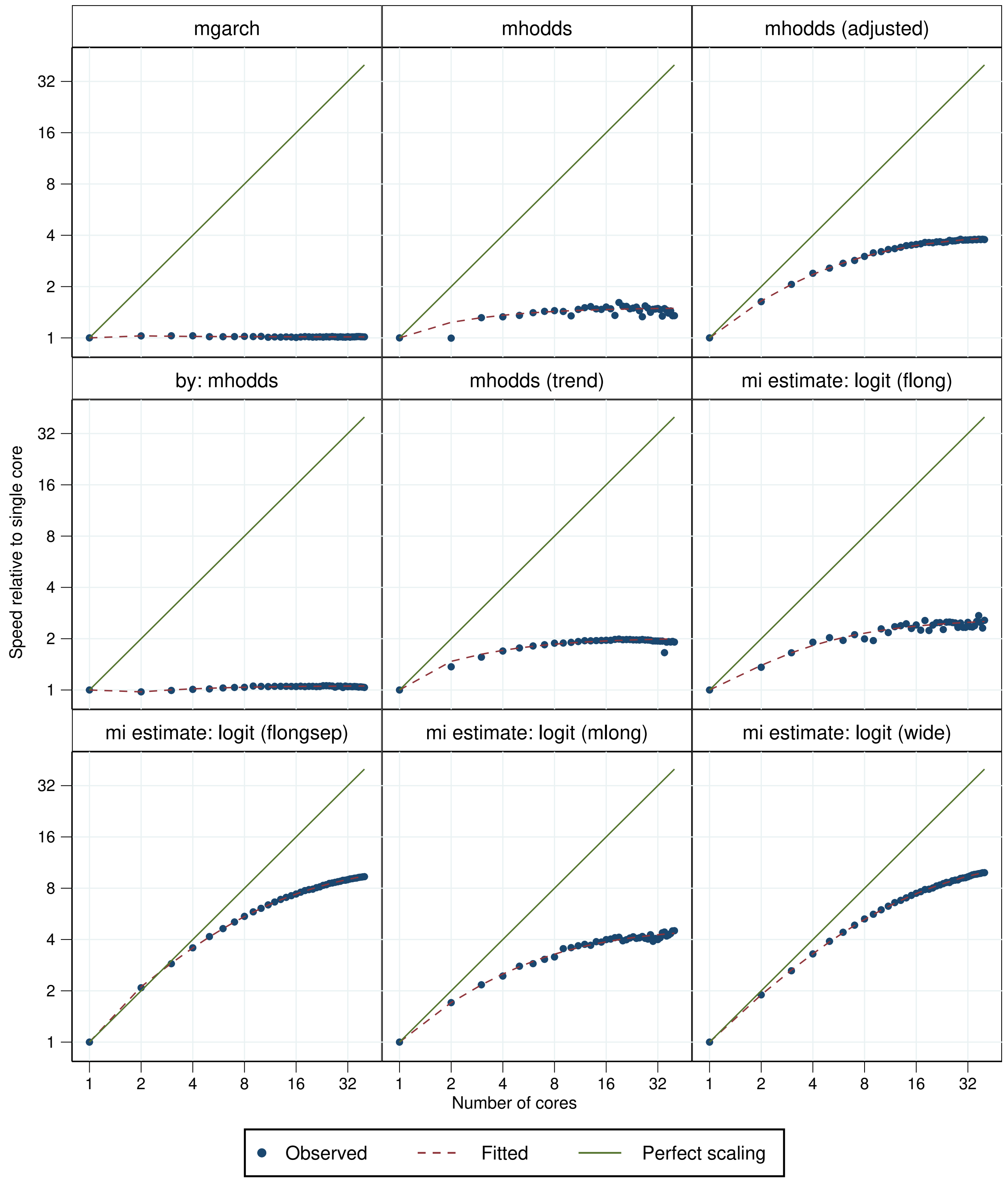


Figure 663. **Parallelization performance plots.**

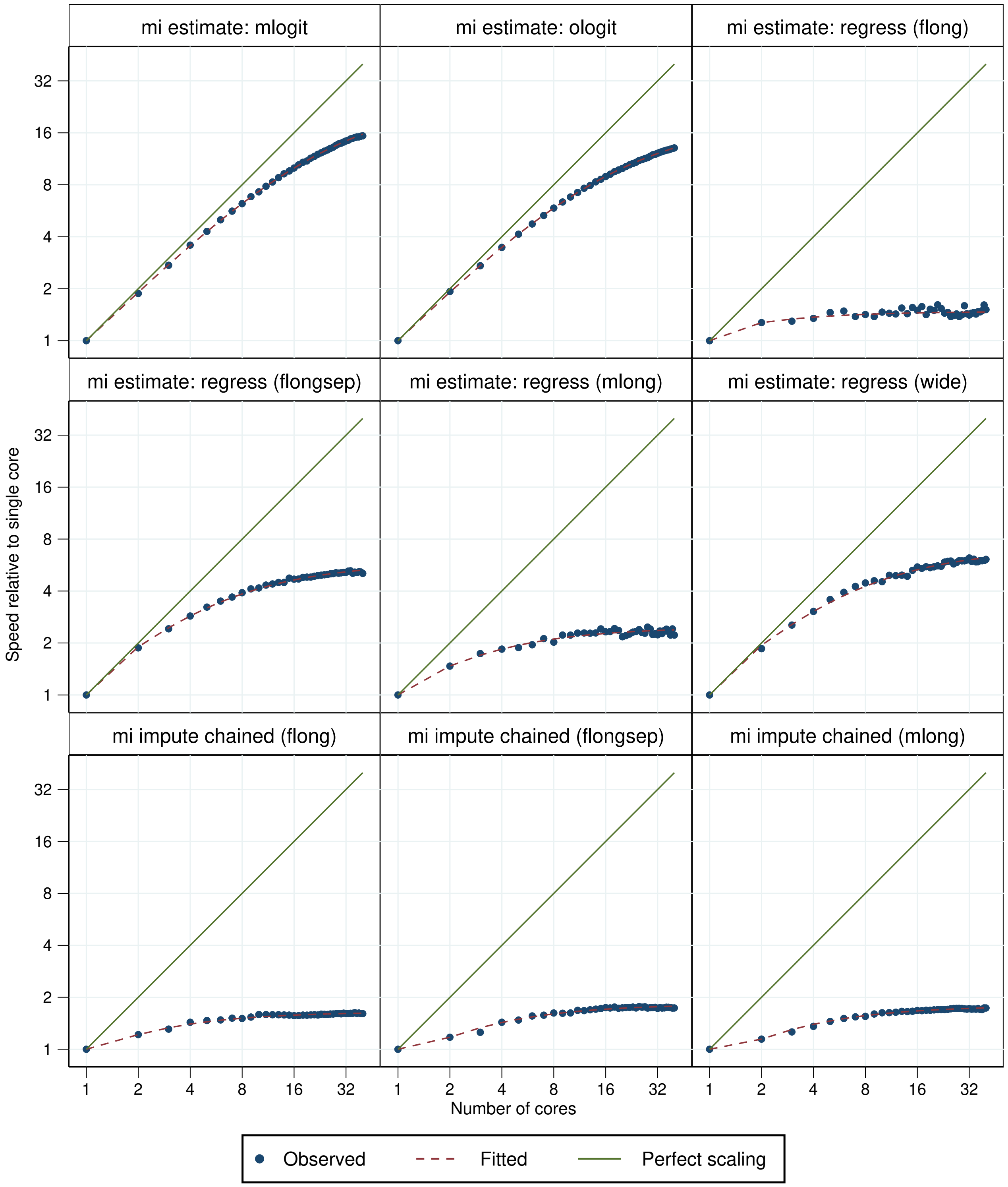


Figure 664. **Parallelization performance plots.**

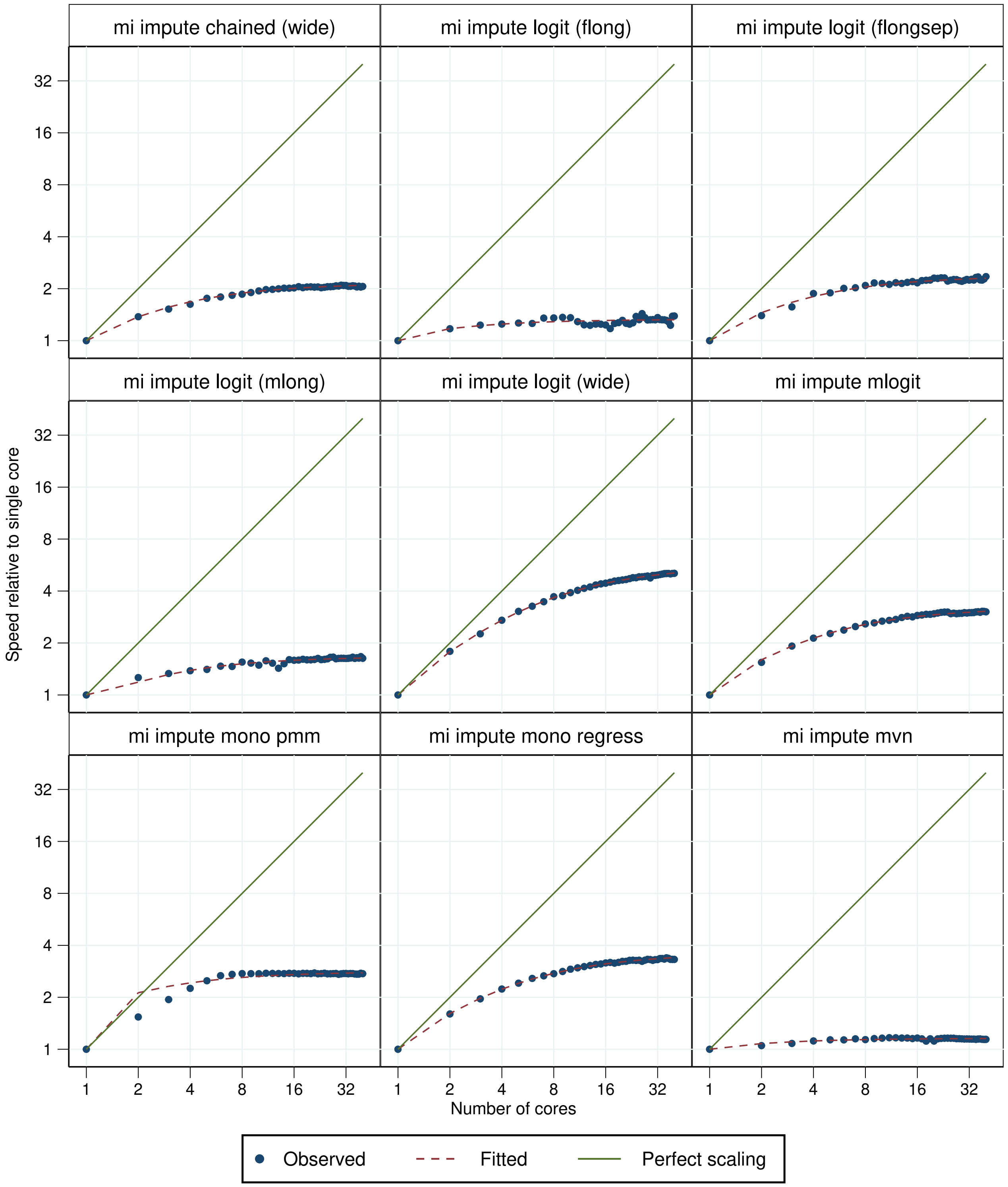


Figure 665. **Parallelization performance plots.**

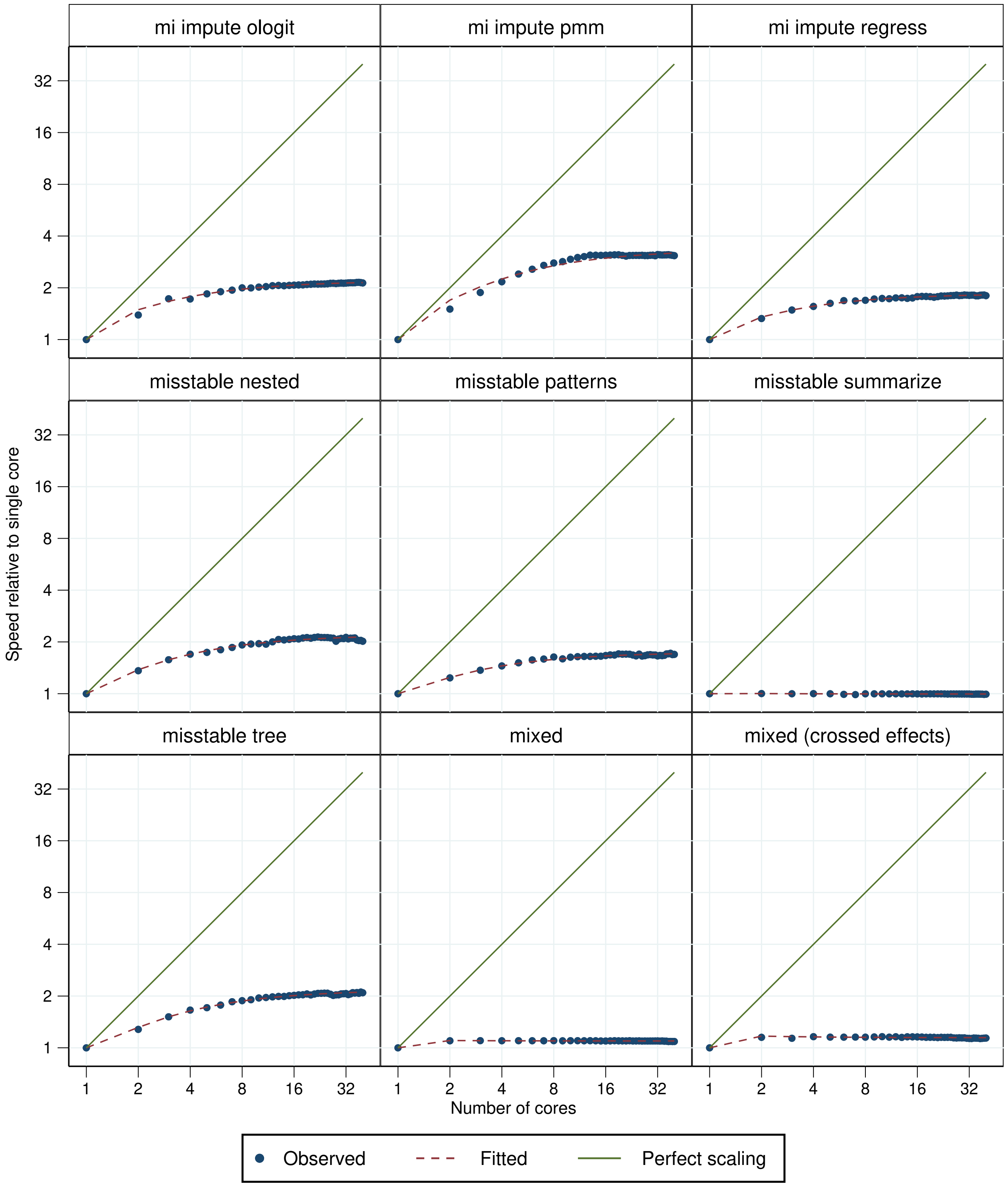


Figure 666. **Parallelization performance plots.**

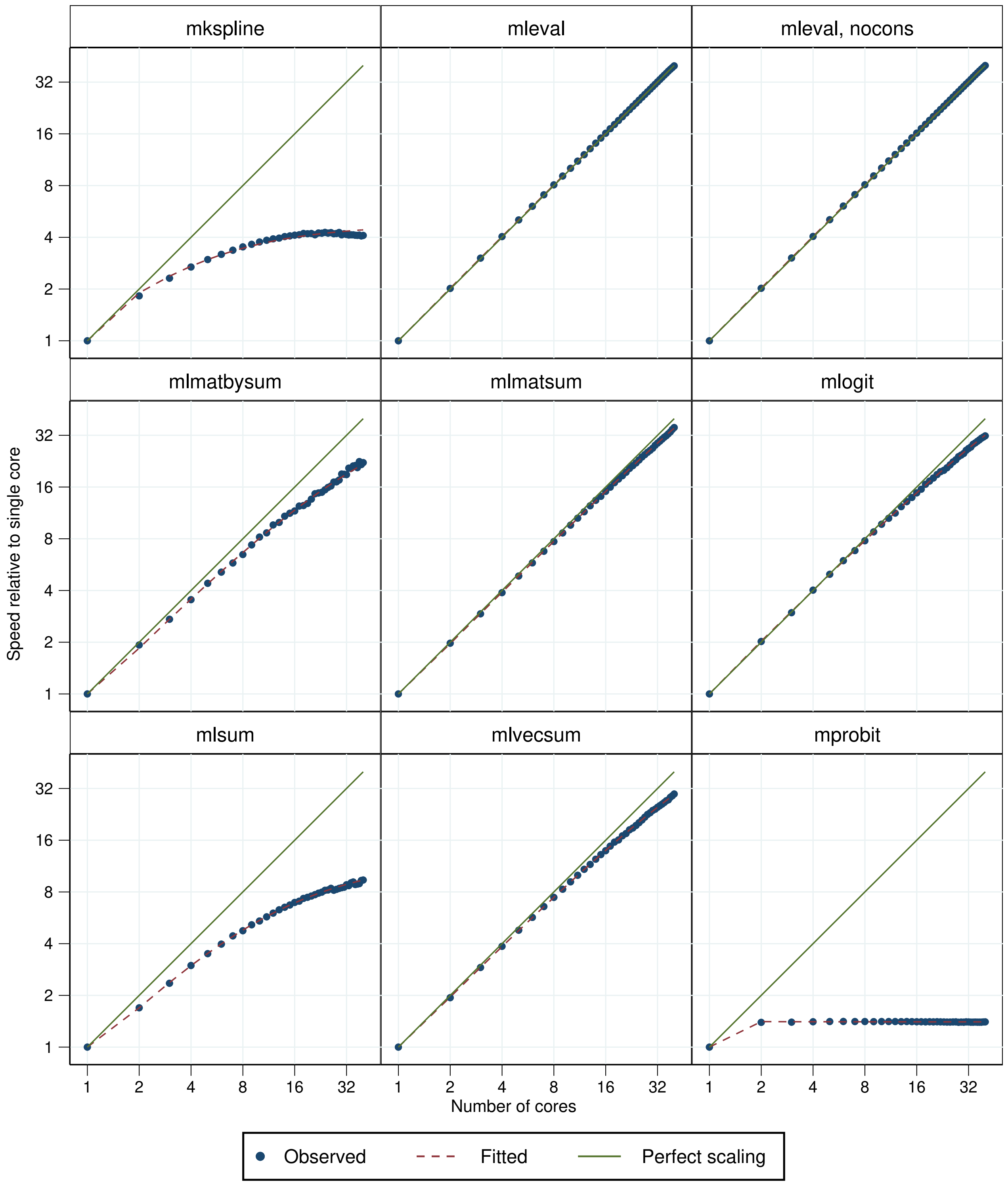


Figure 667. **Parallelization performance plots.**

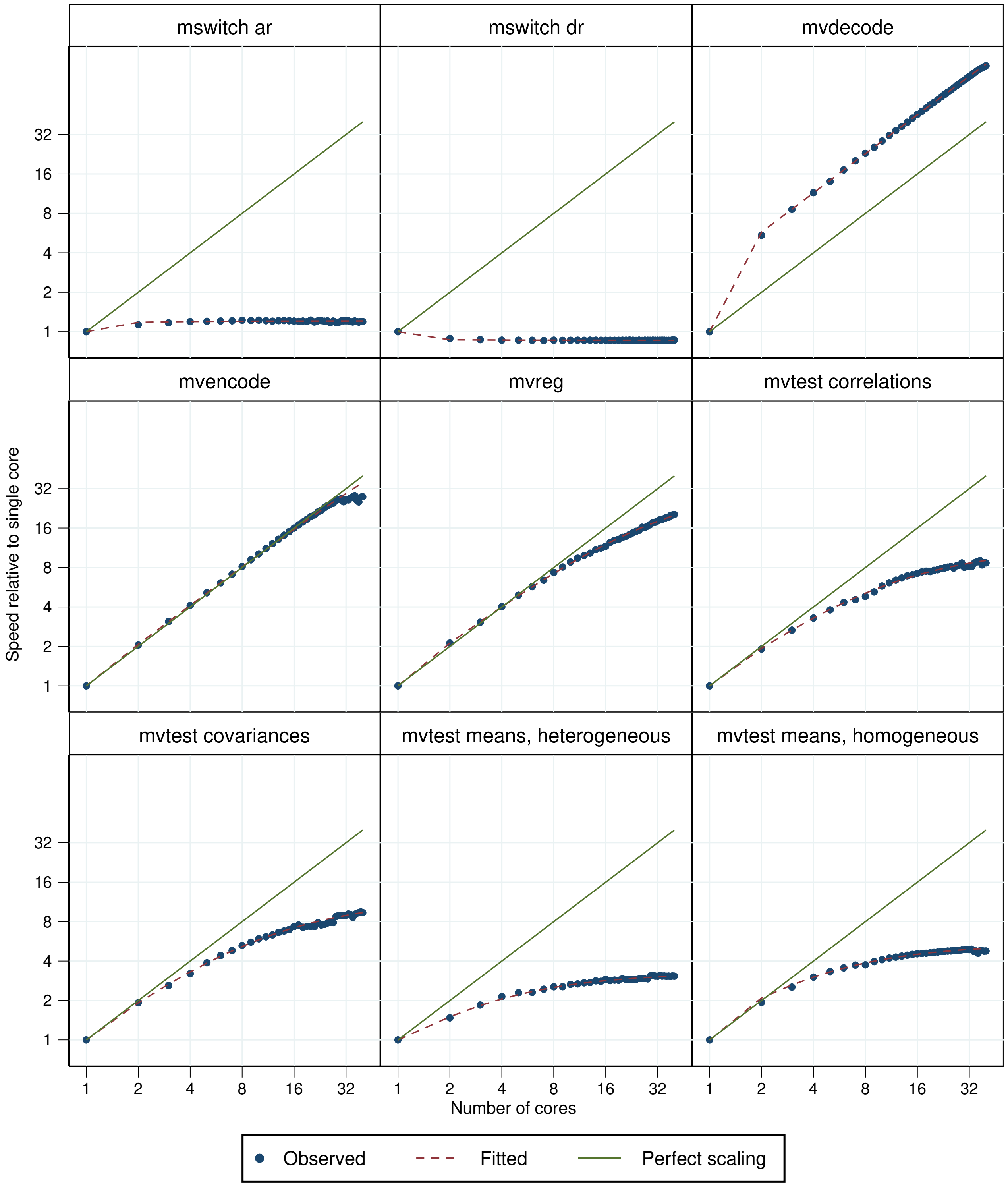


Figure 668. **Parallelization performance plots.**

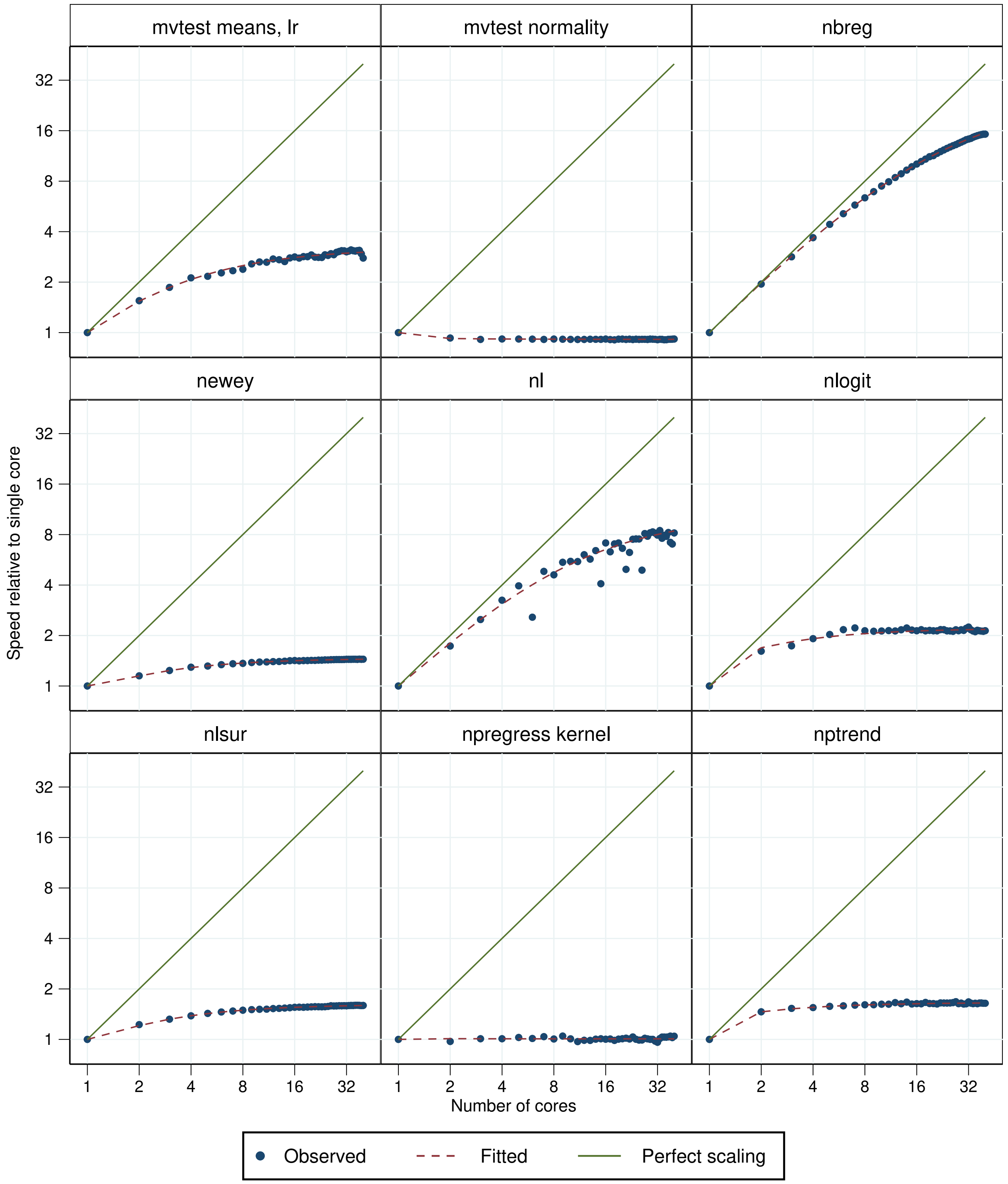


Figure 669. **Parallelization performance plots.**

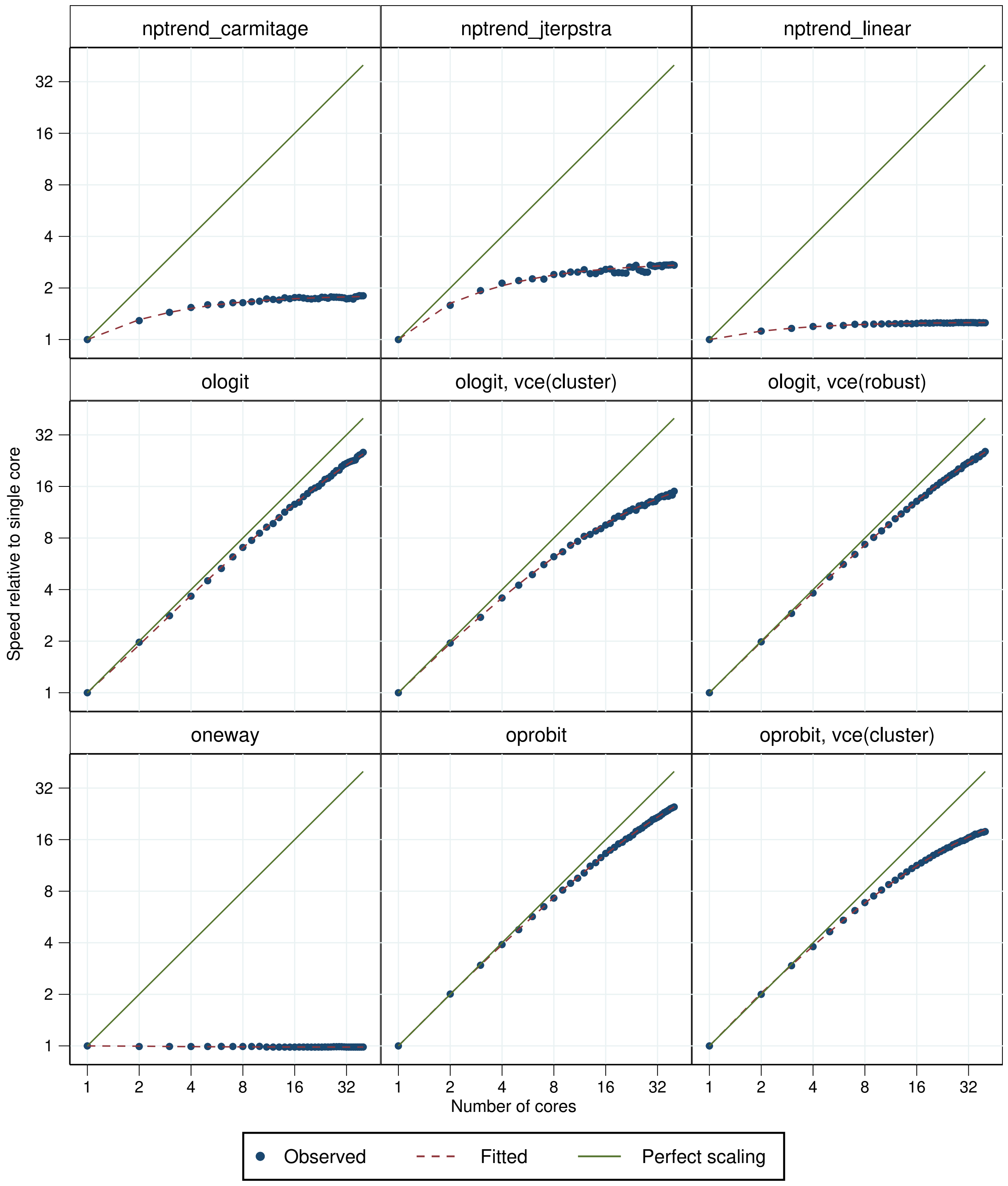


Figure 670. **Parallelization performance plots.**

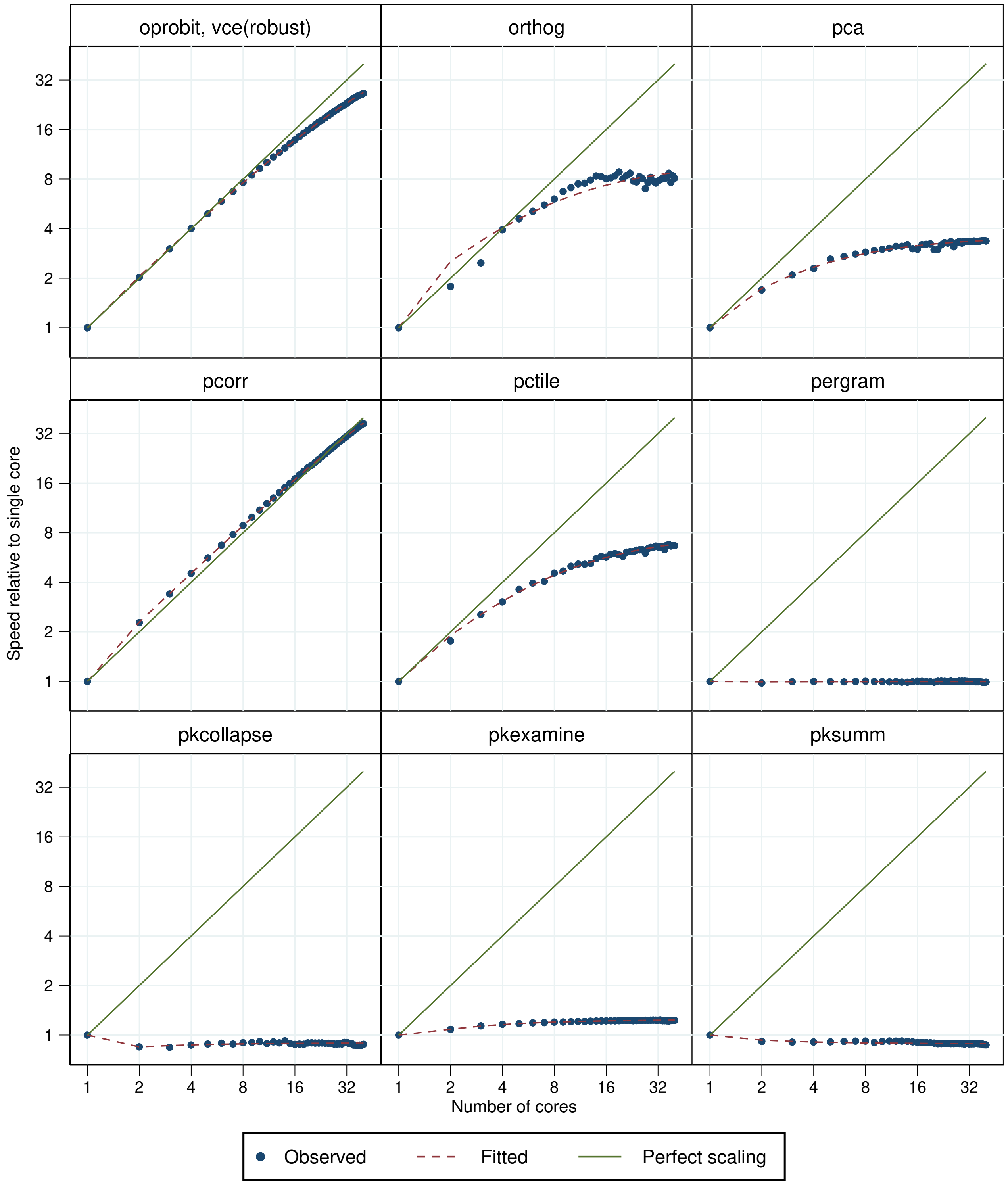


Figure 671. **Parallelization performance plots.**

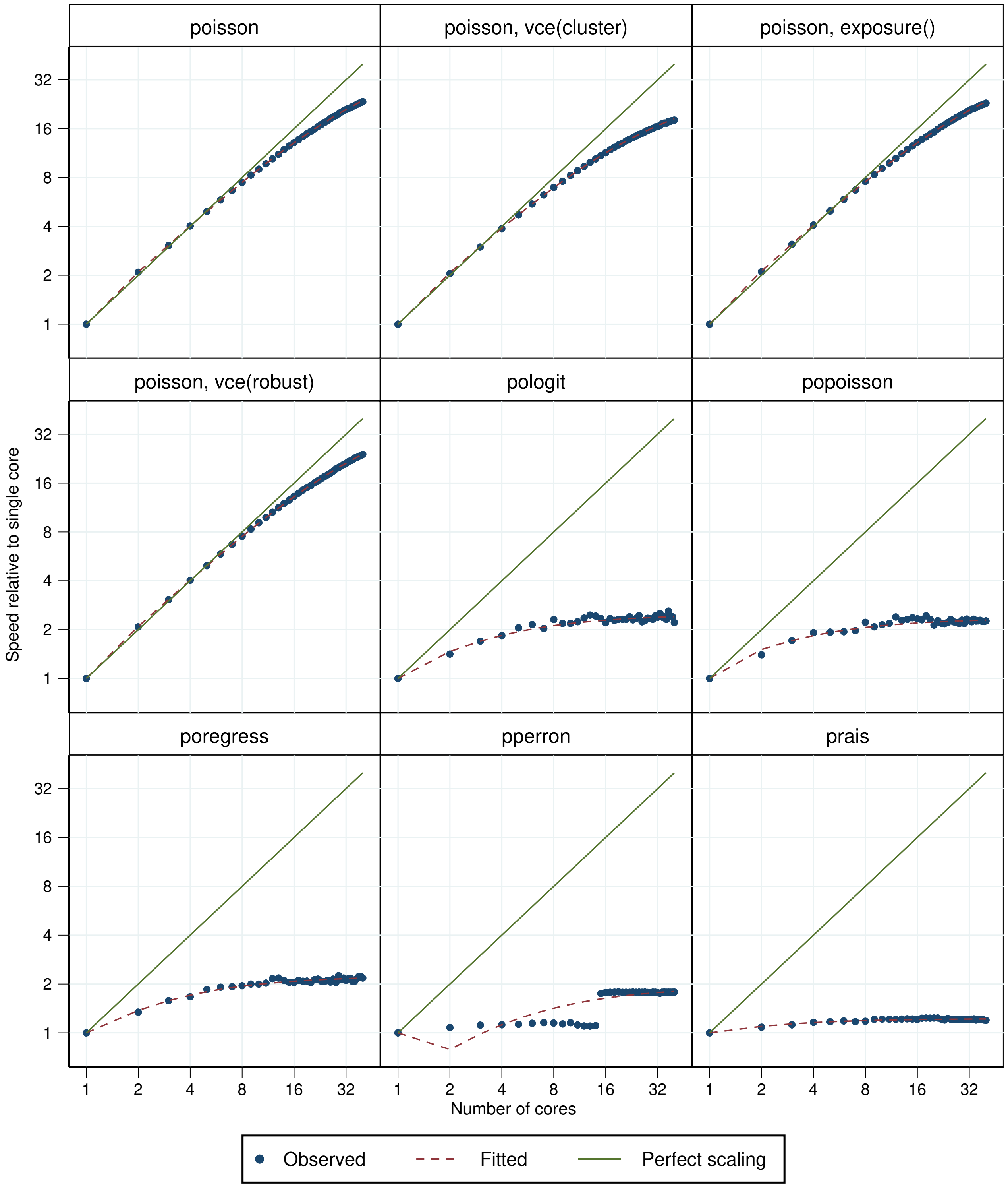


Figure 672. **Parallelization performance plots.**

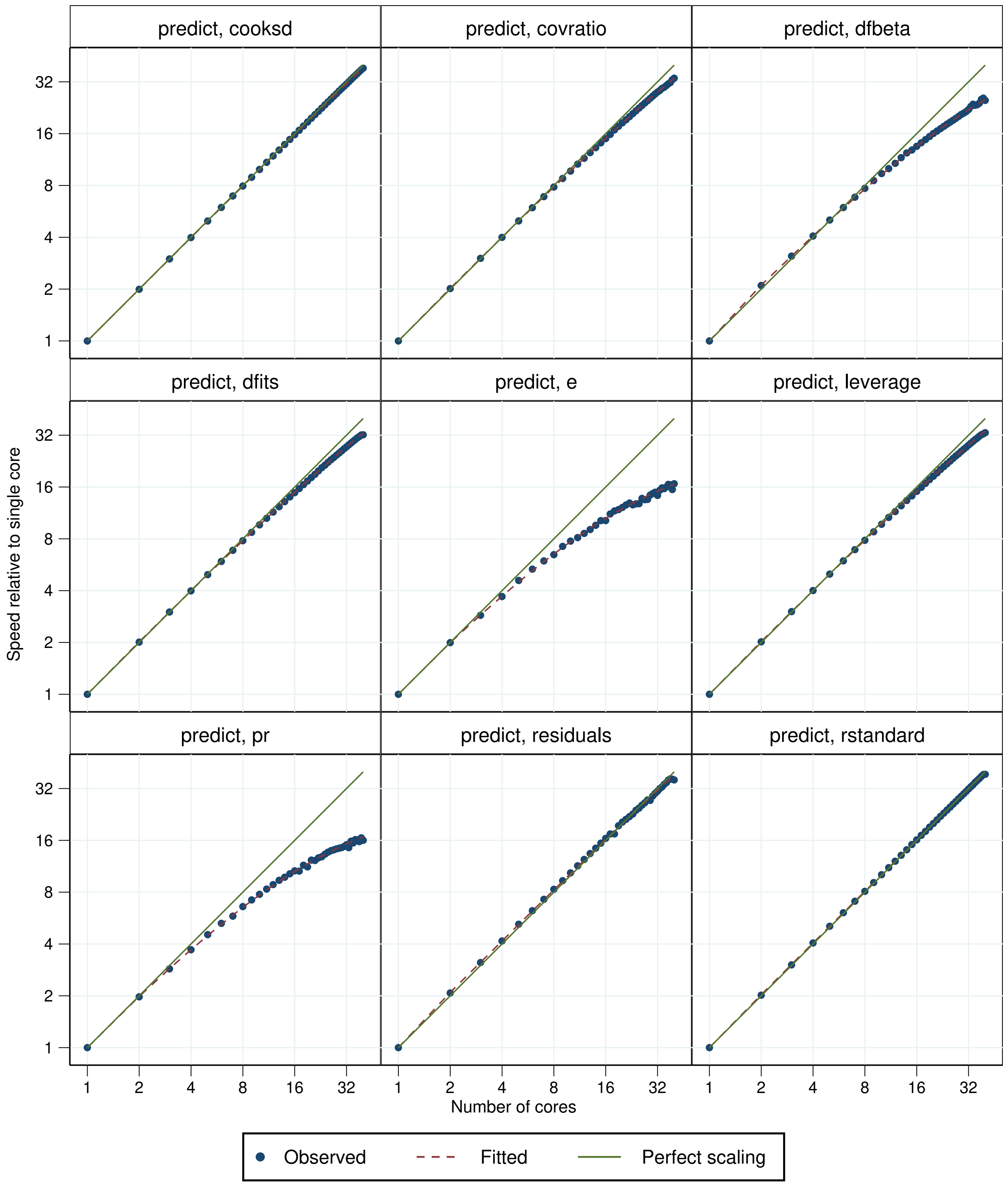


Figure 673. **Parallelization performance plots.**

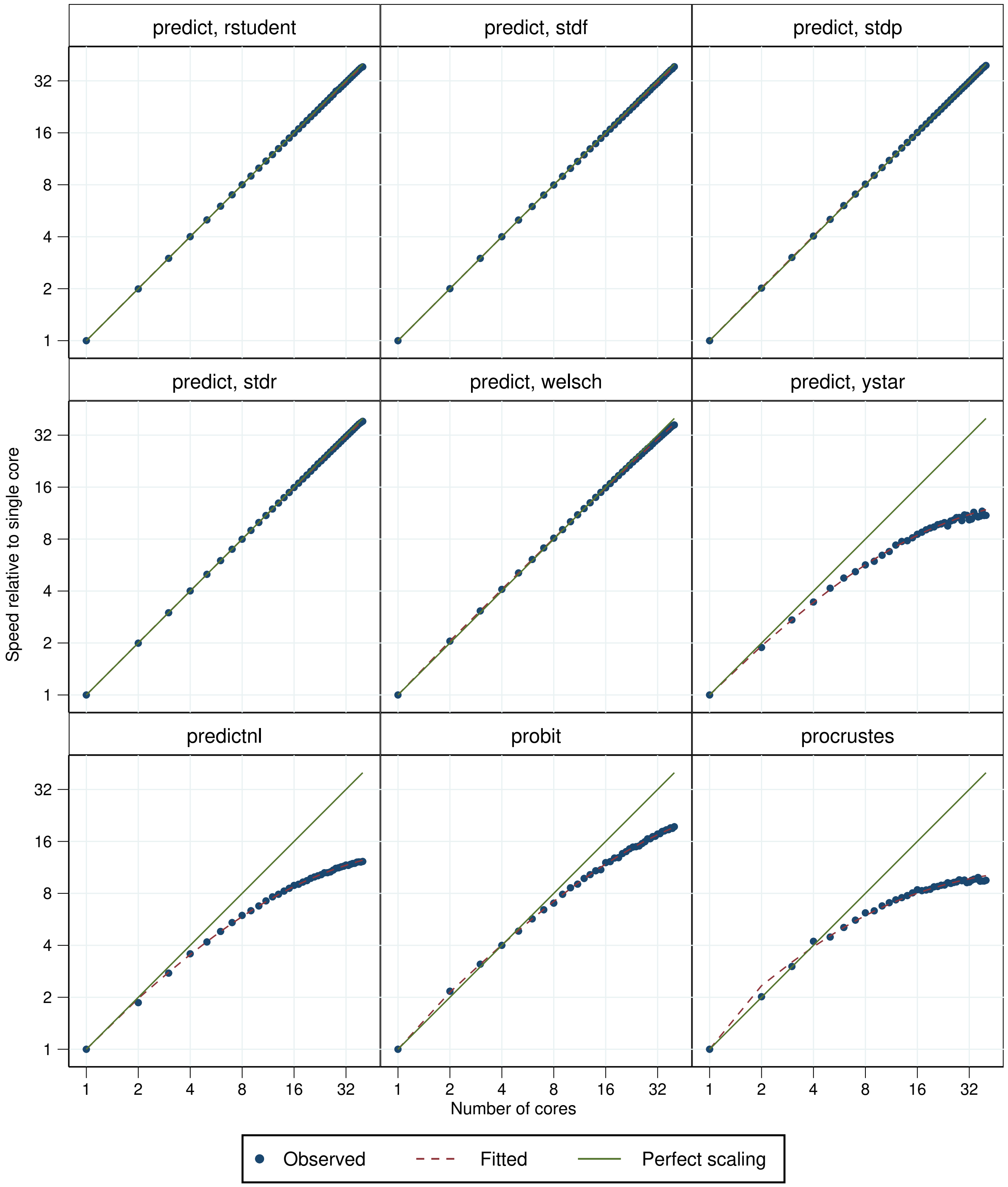


Figure 674. **Parallelization performance plots.**

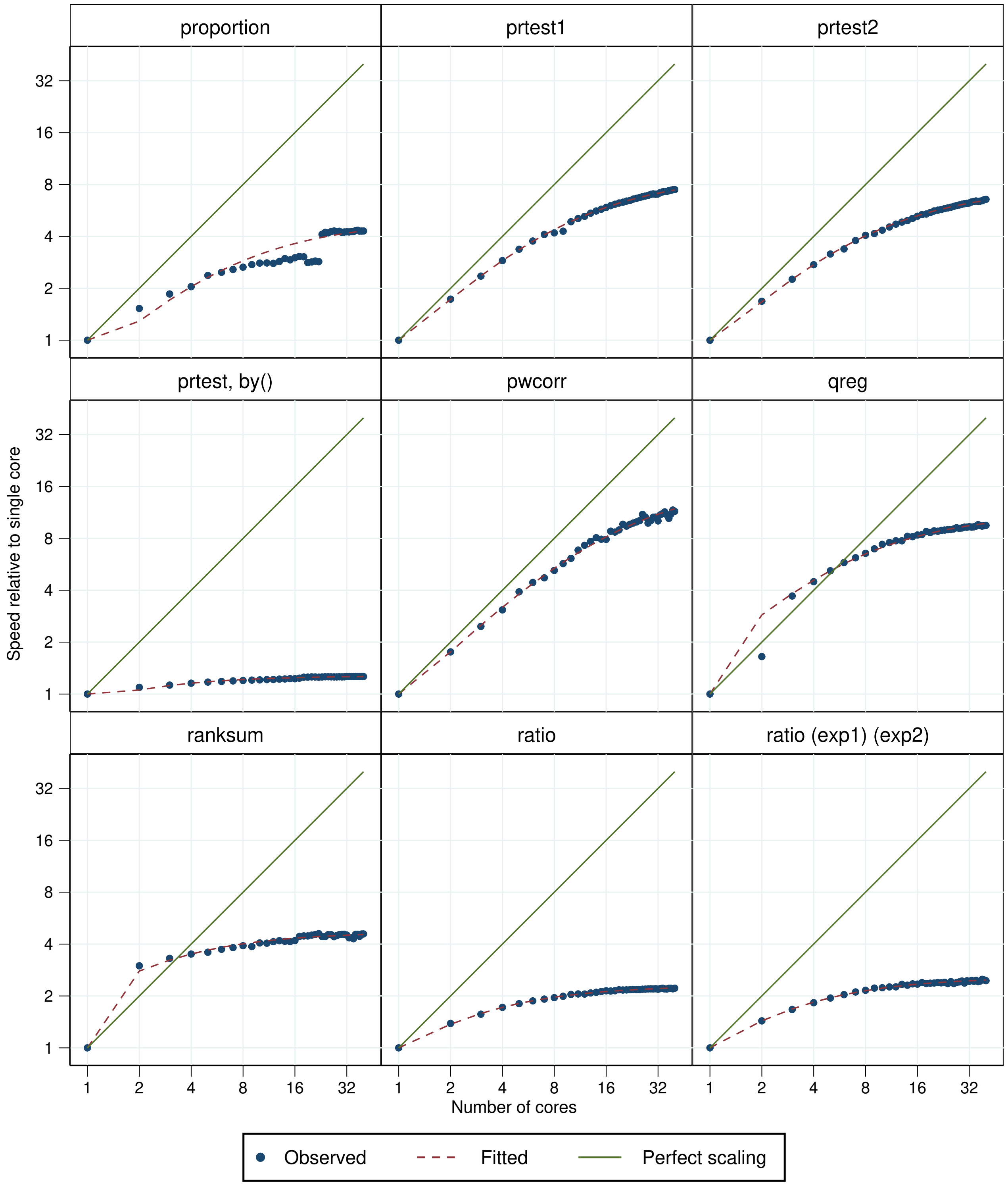


Figure 675. **Parallelization performance plots.**

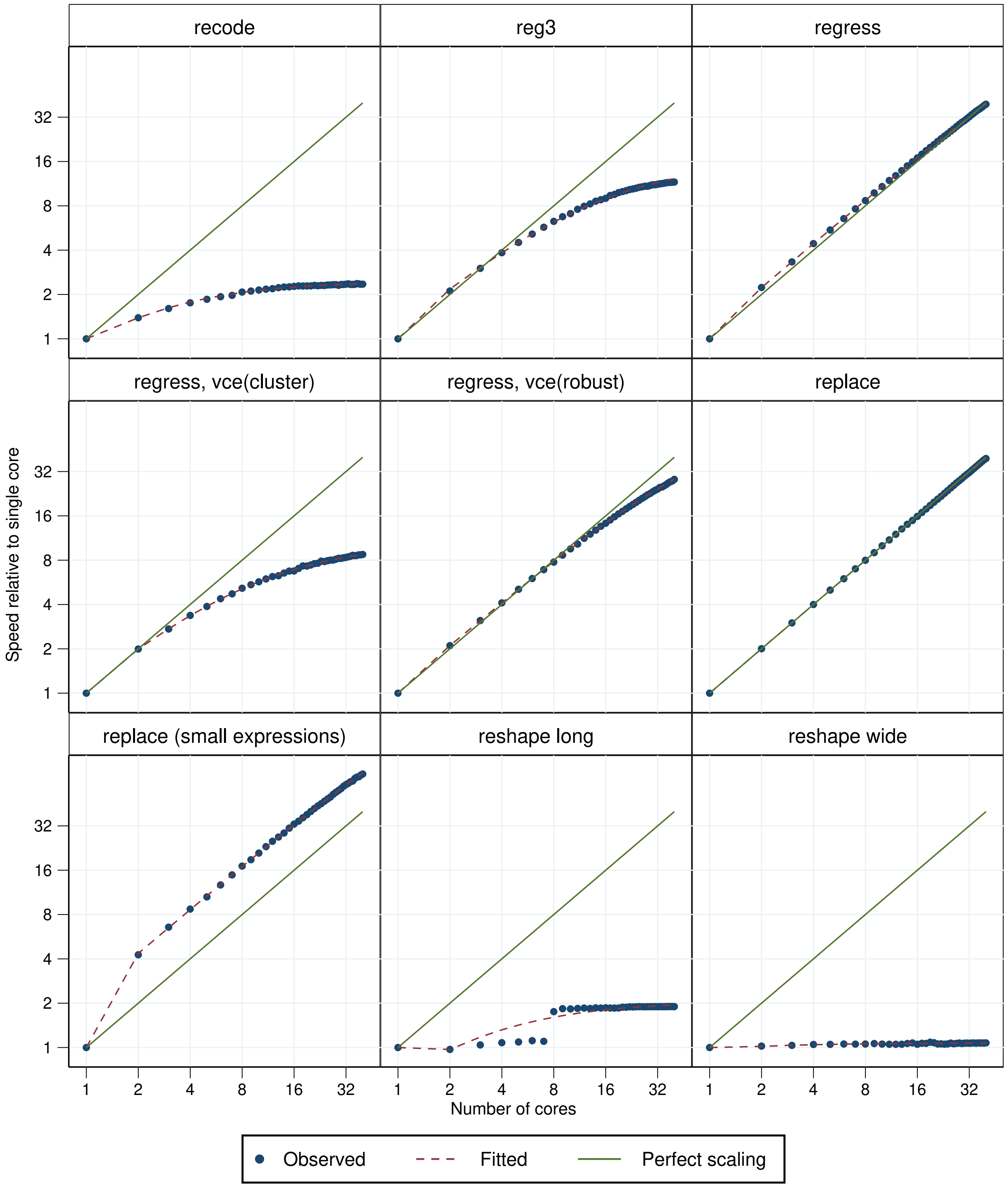


Figure 676. **Parallelization performance plots.**

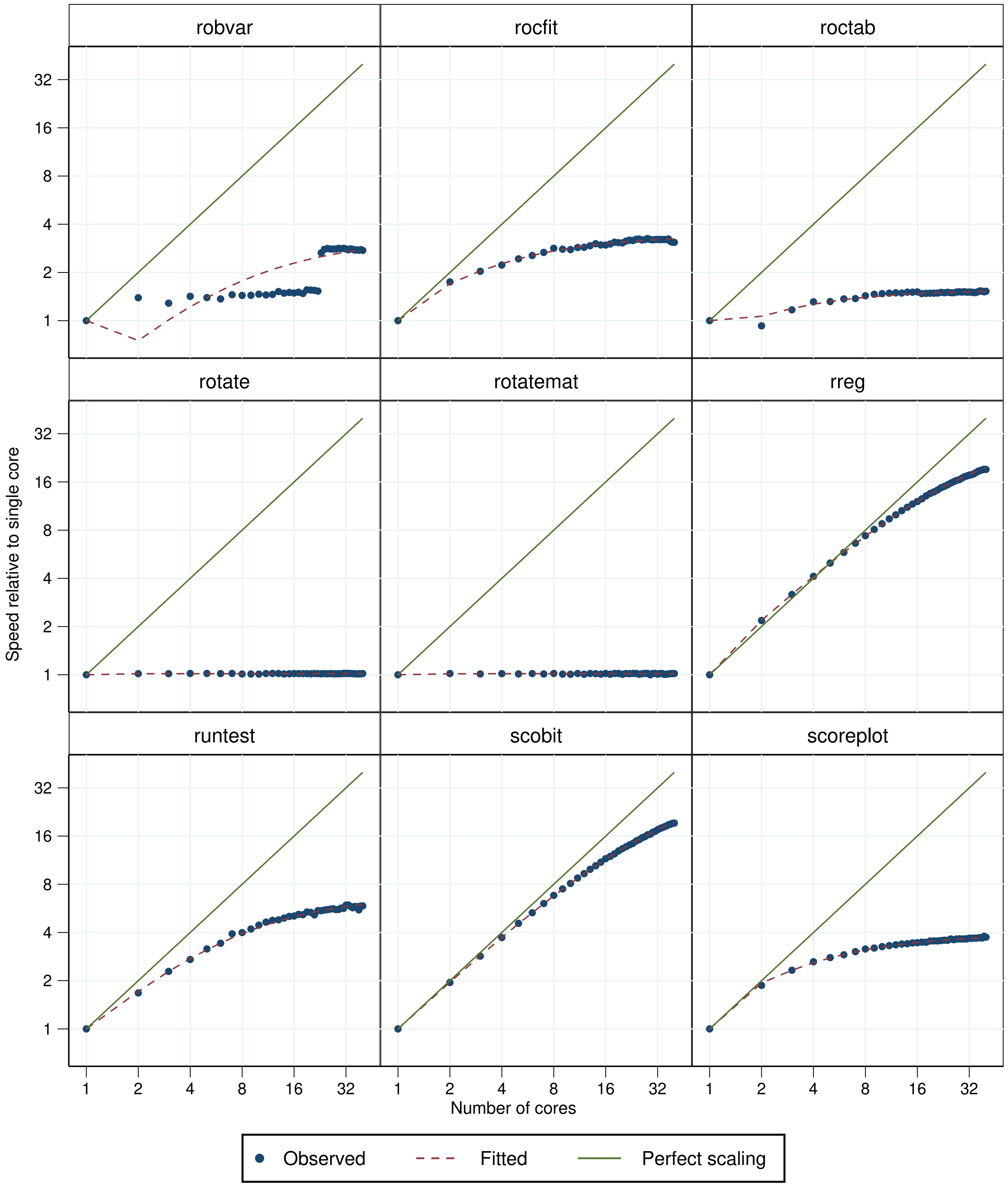


Figure 677. **Parallelization performance plots.**

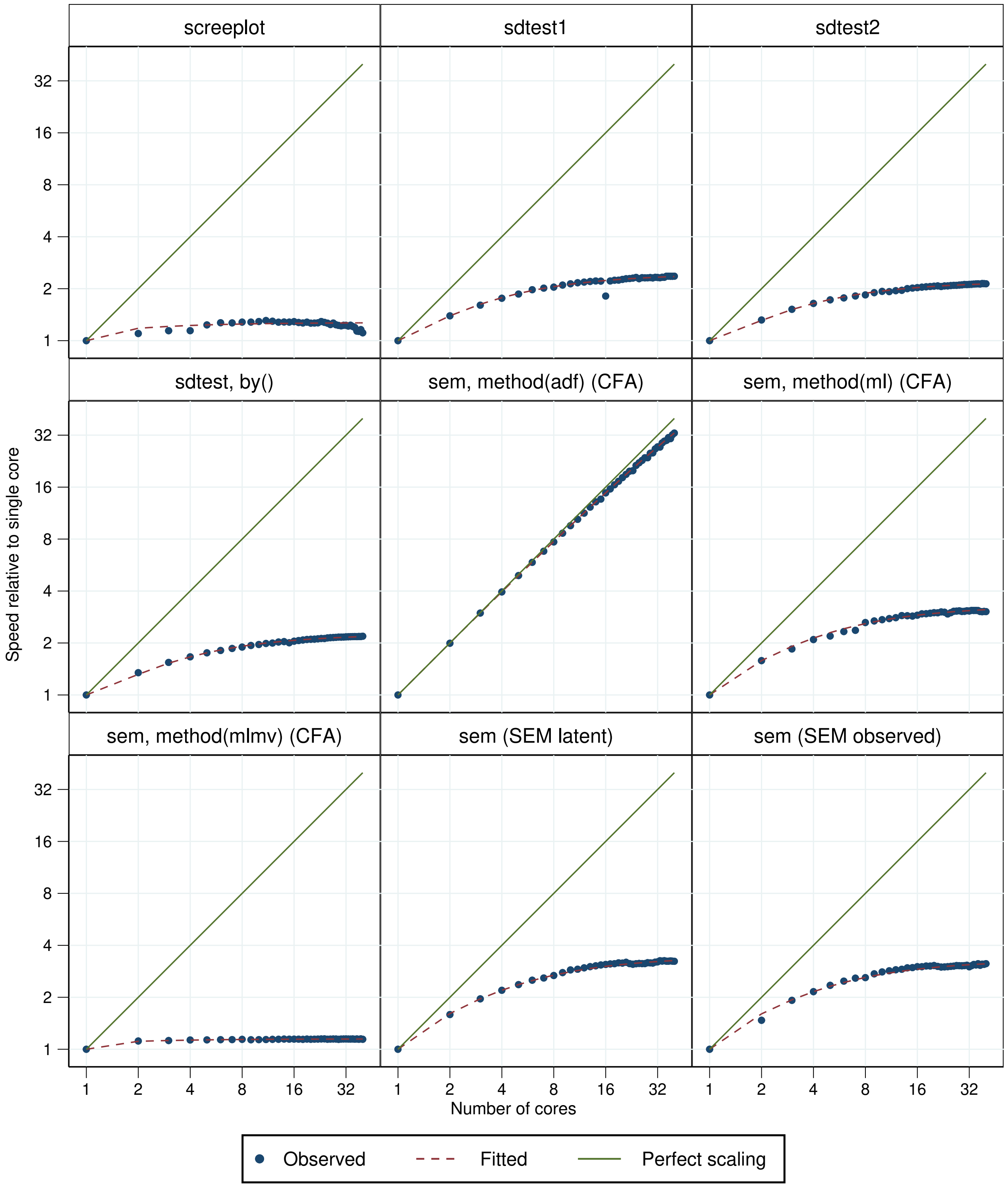


Figure 678. **Parallelization performance plots.**

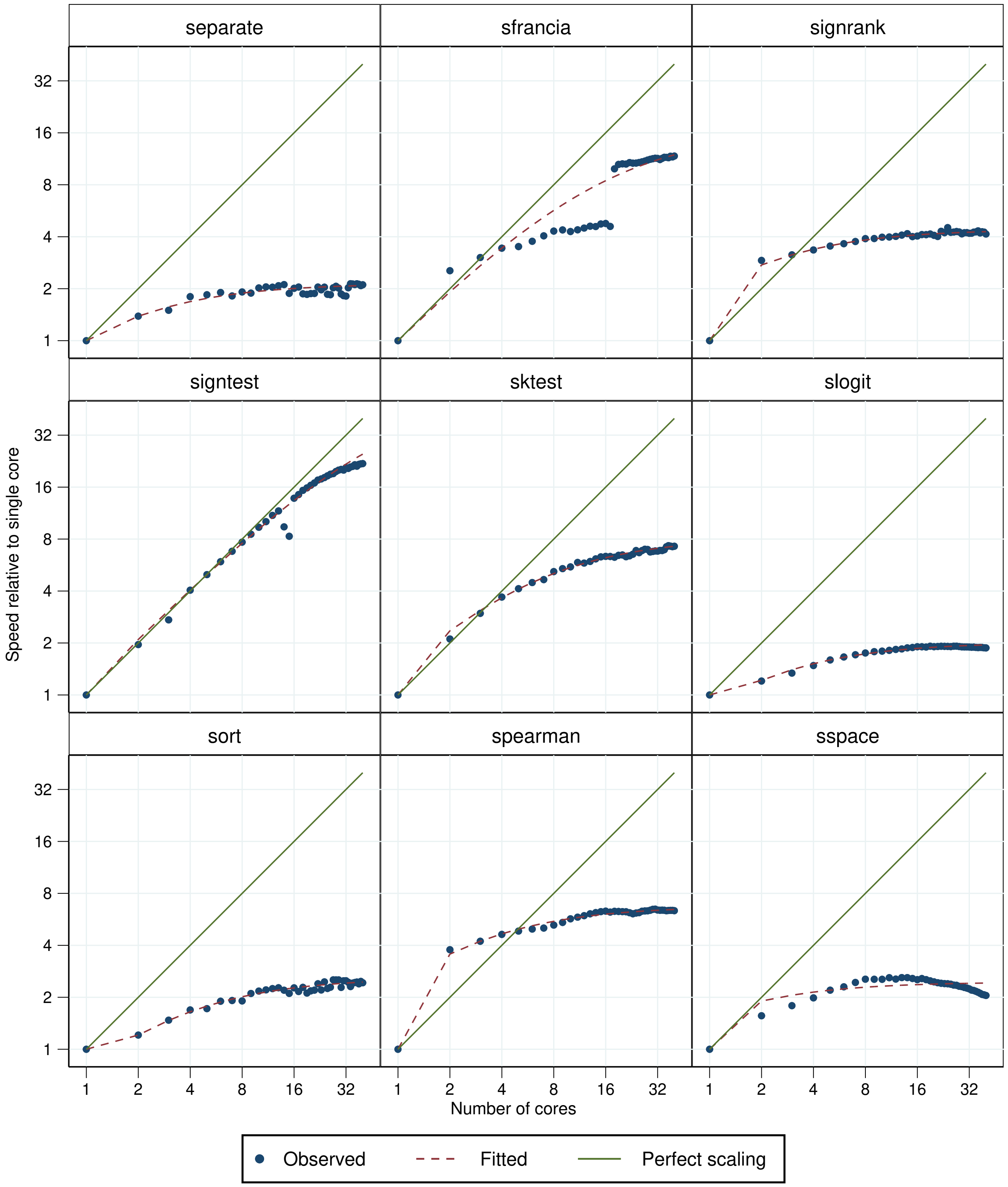


Figure 679. **Parallelization performance plots.**

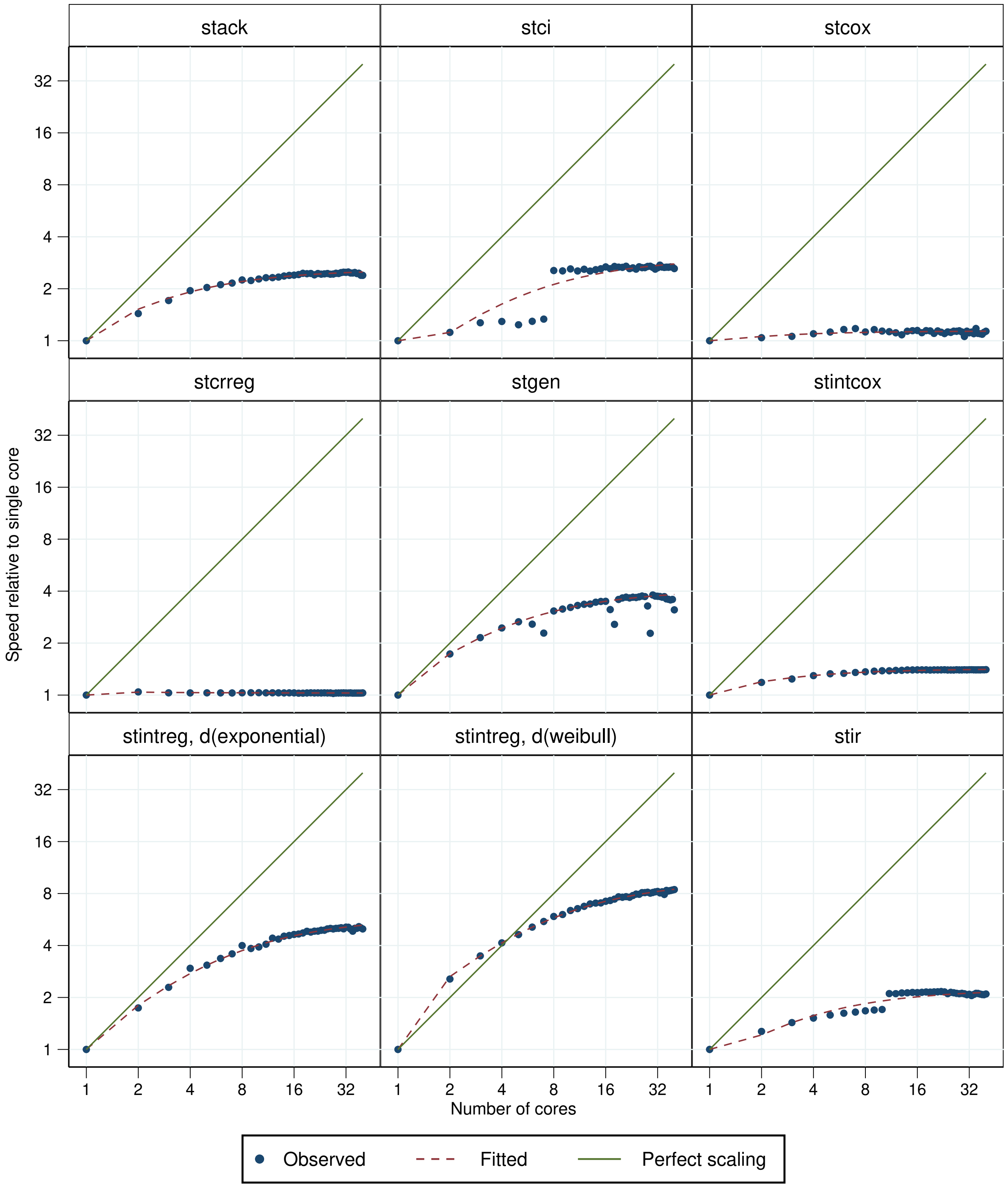


Figure 680. **Parallelization performance plots.**

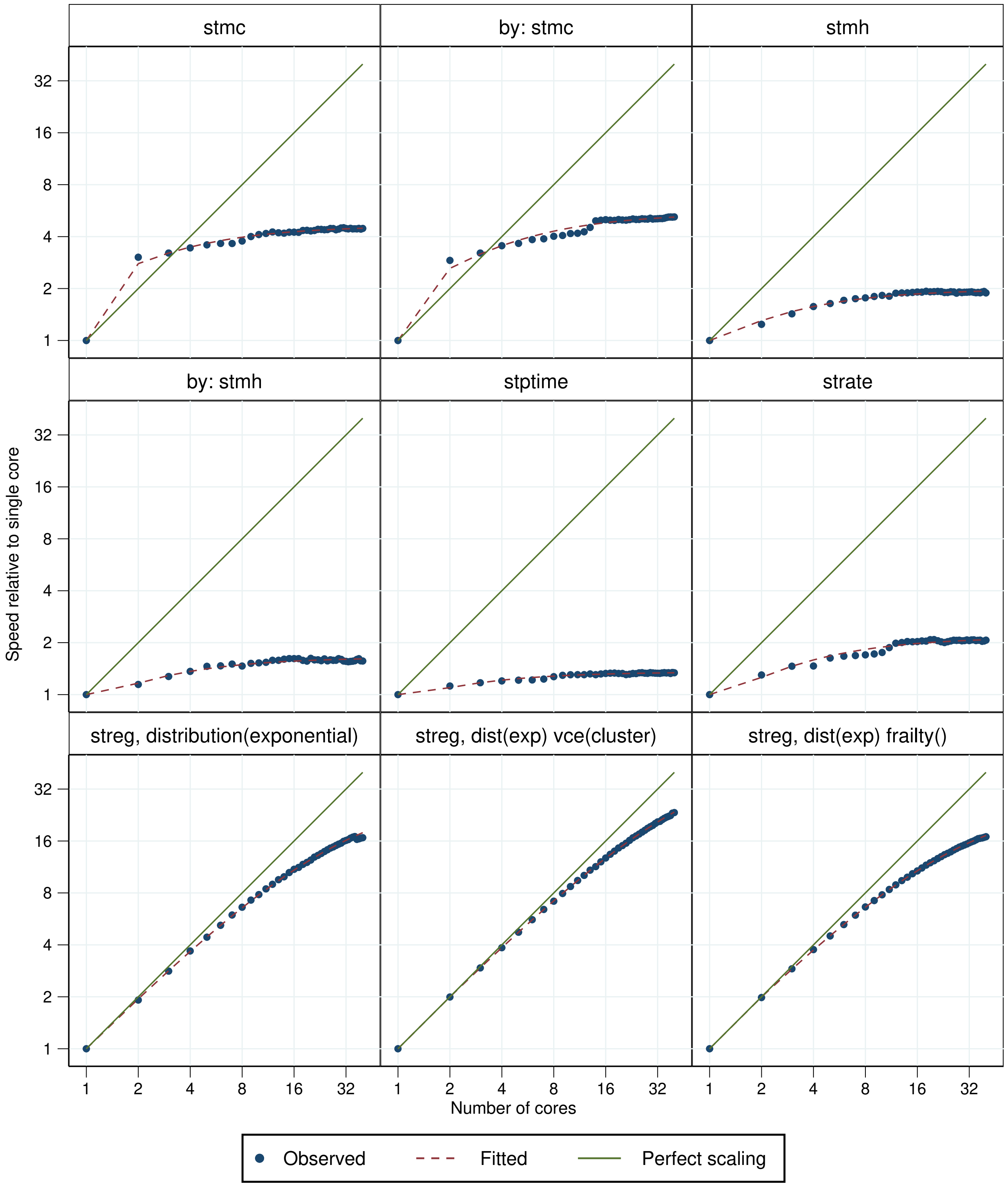


Figure 681. **Parallelization performance plots.**

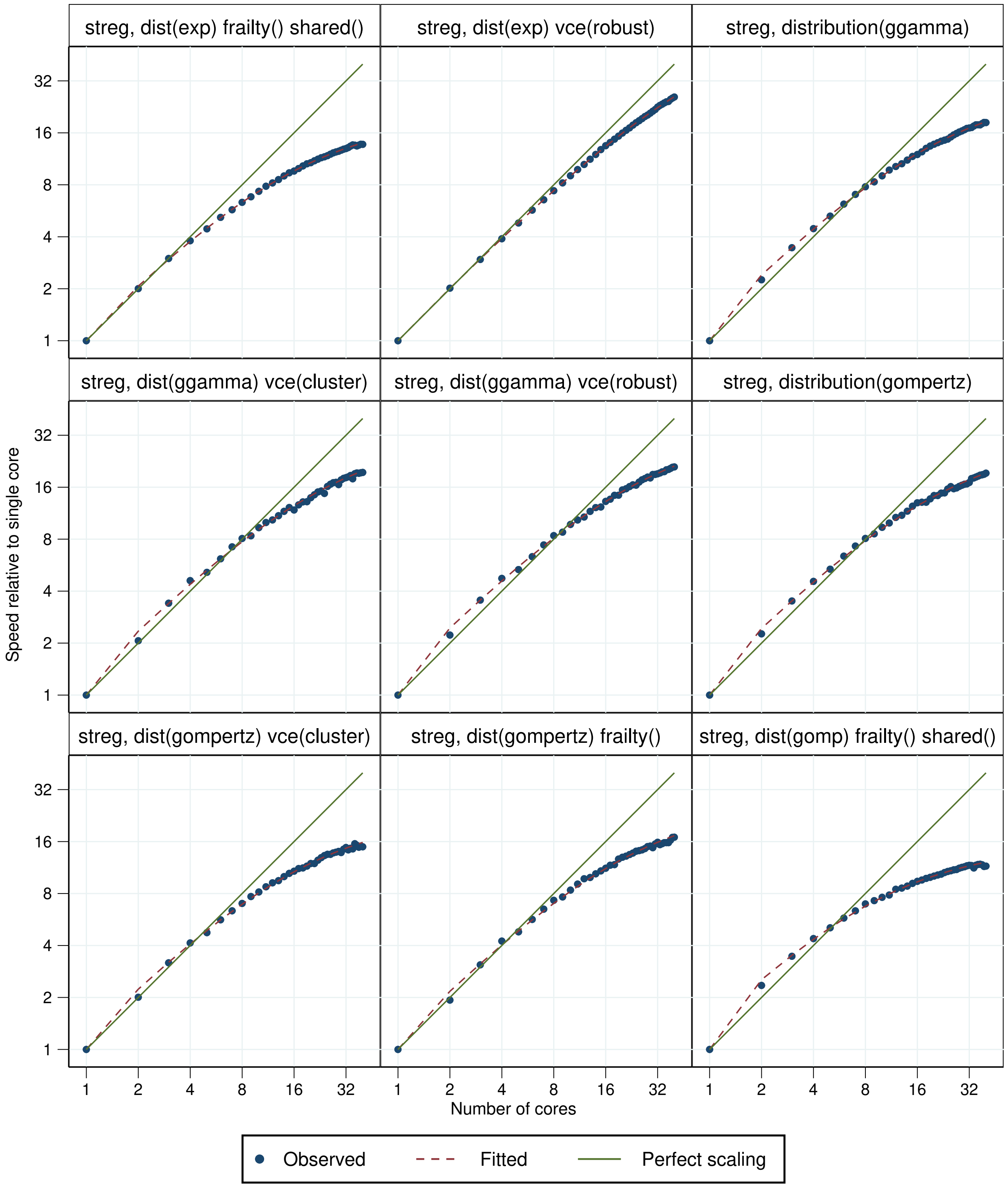


Figure 682. **Parallelization performance plots.**

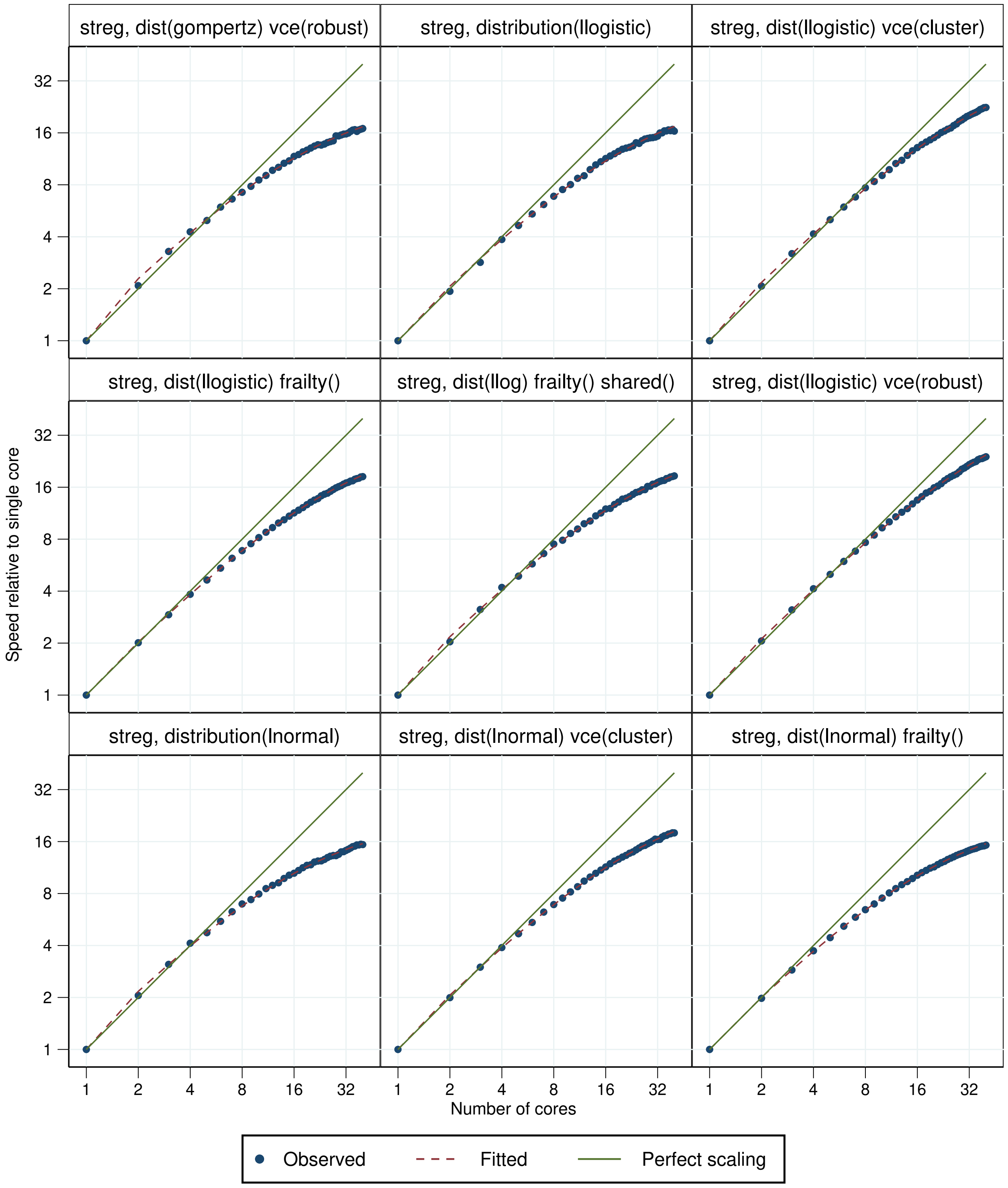


Figure 683. **Parallelization performance plots.**

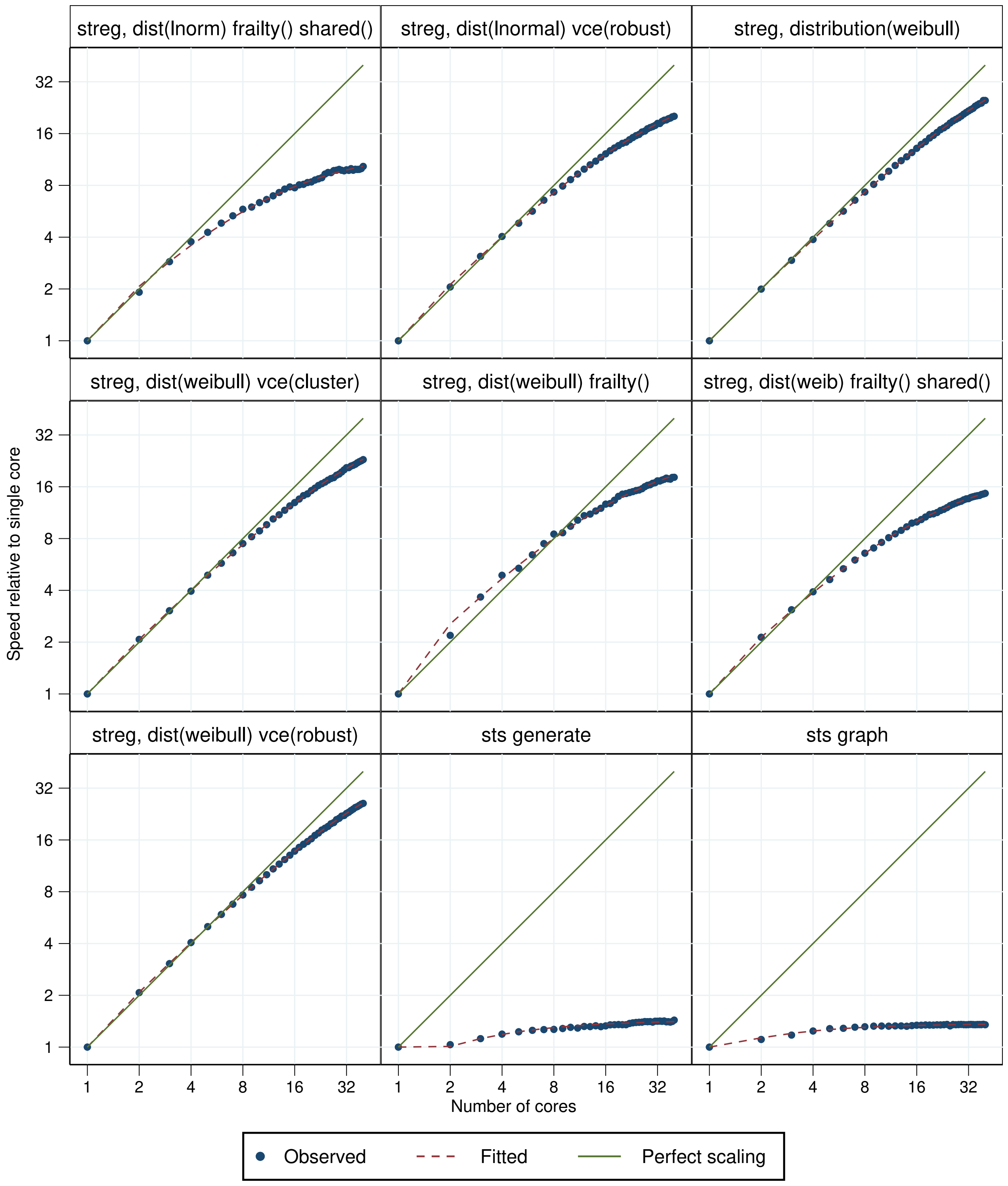


Figure 684. **Parallelization performance plots.**

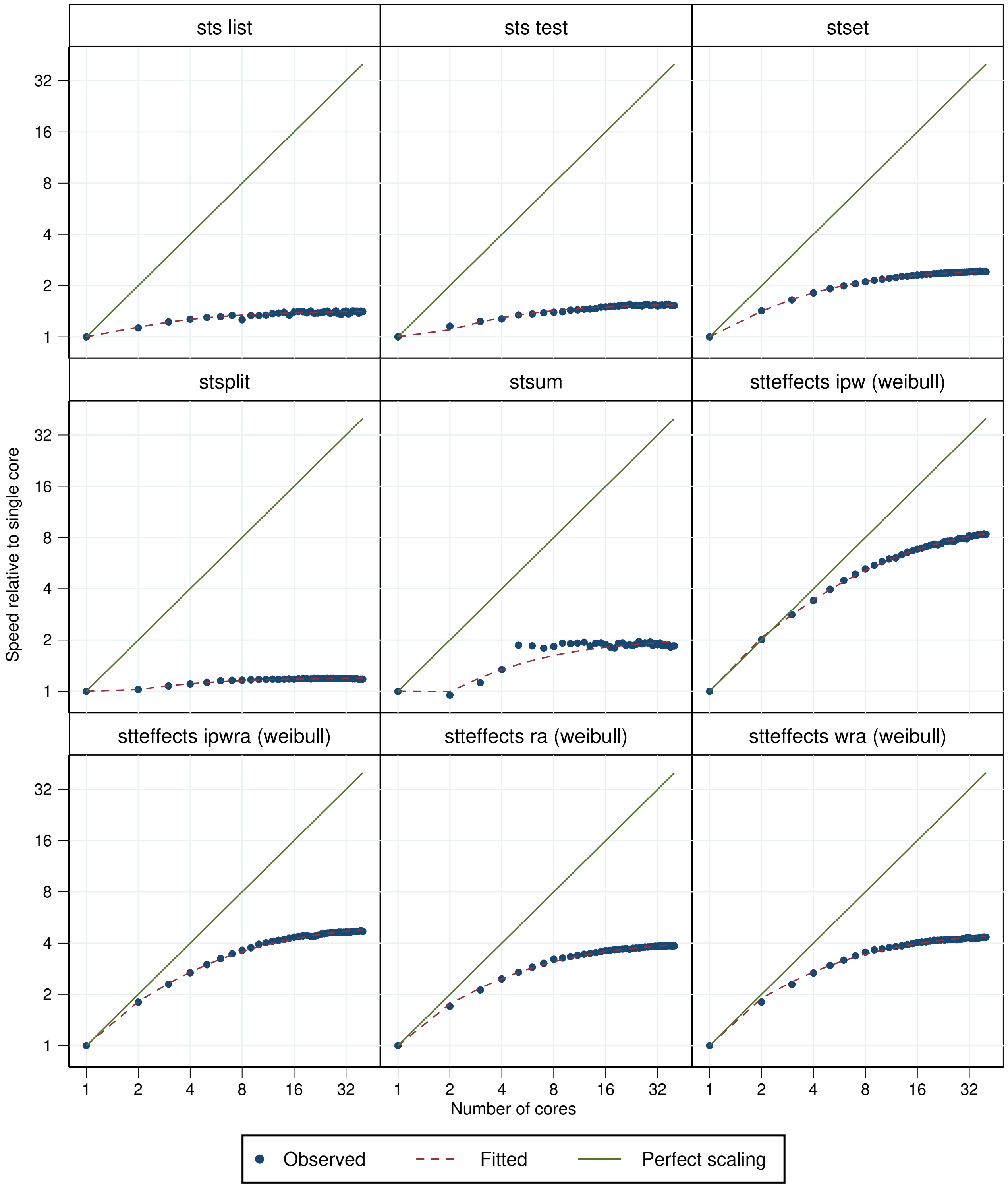


Figure 685. **Parallelization performance plots.**

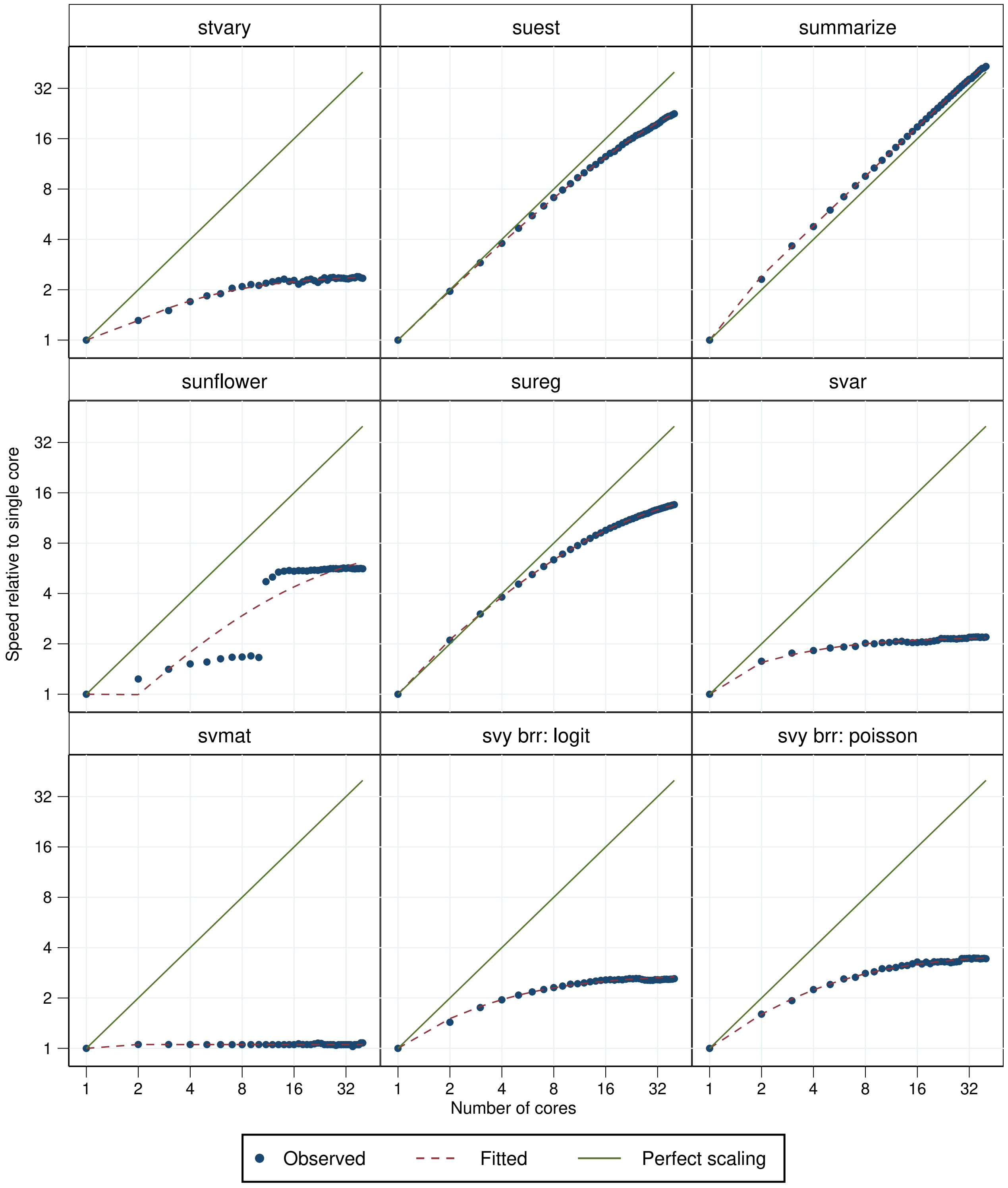


Figure 686. **Parallelization performance plots.**

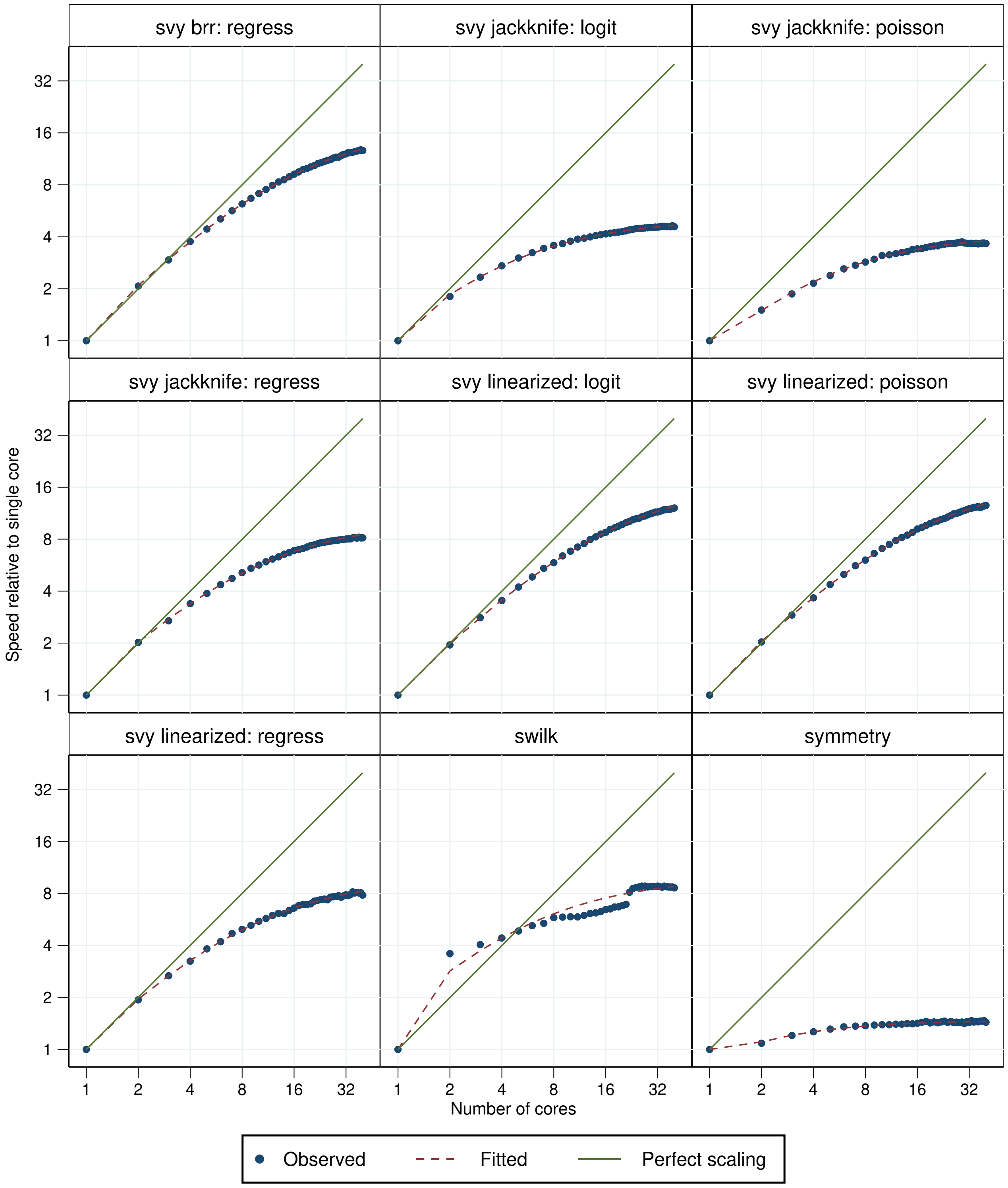


Figure 687. **Parallelization performance plots.**

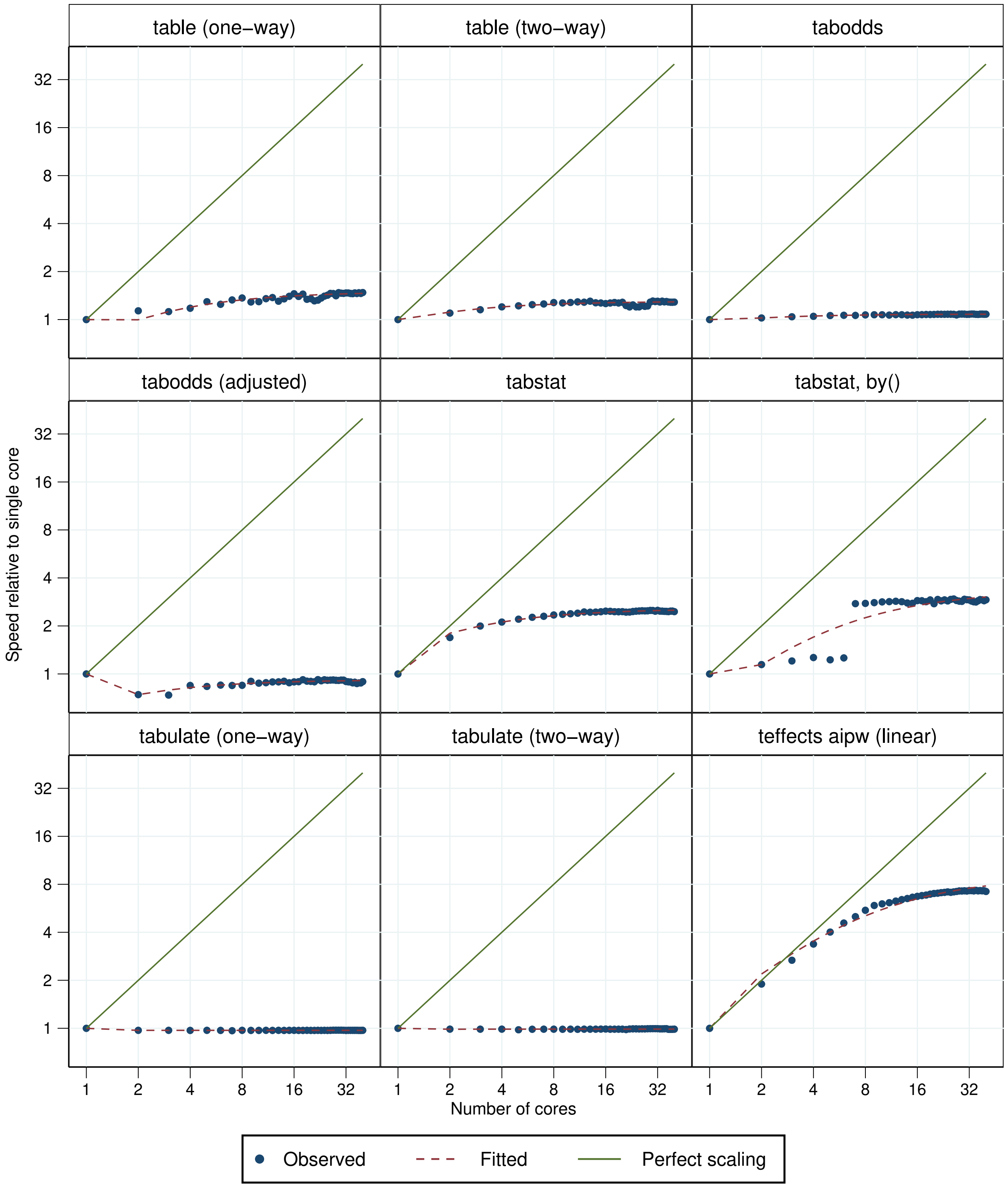


Figure 688. **Parallelization performance plots.**

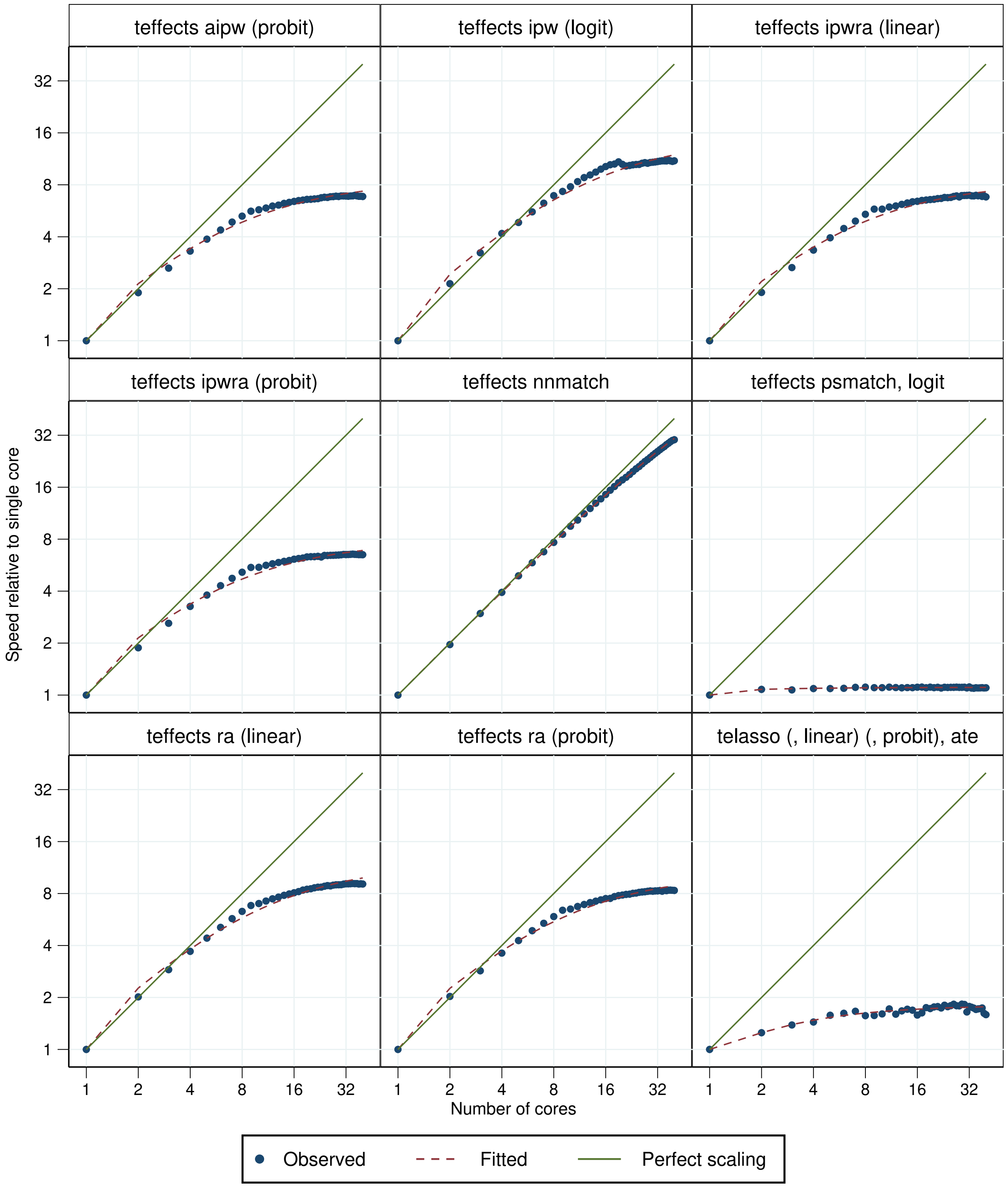


Figure 689. **Parallelization performance plots.**

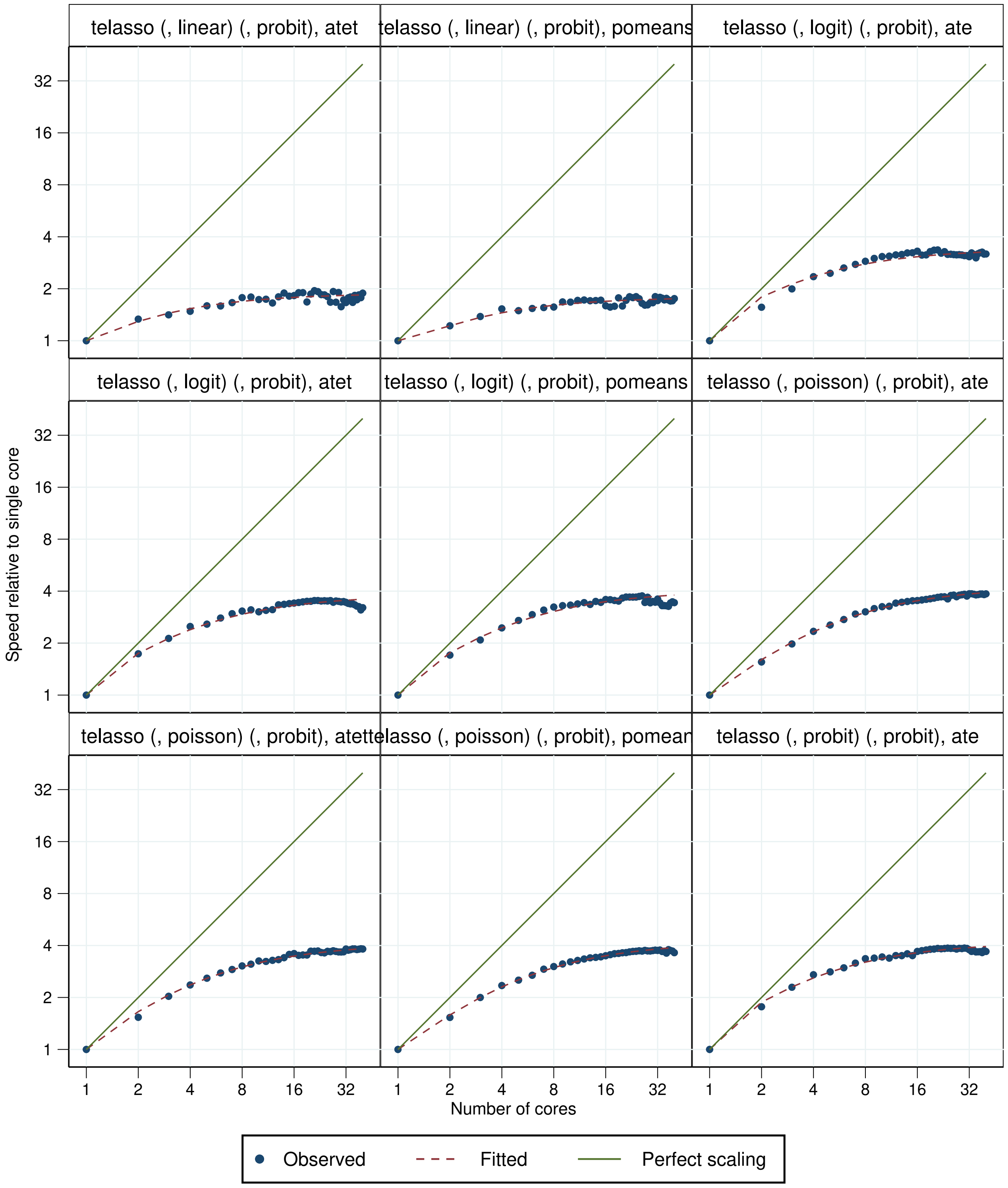


Figure 690. **Parallelization performance plots.**

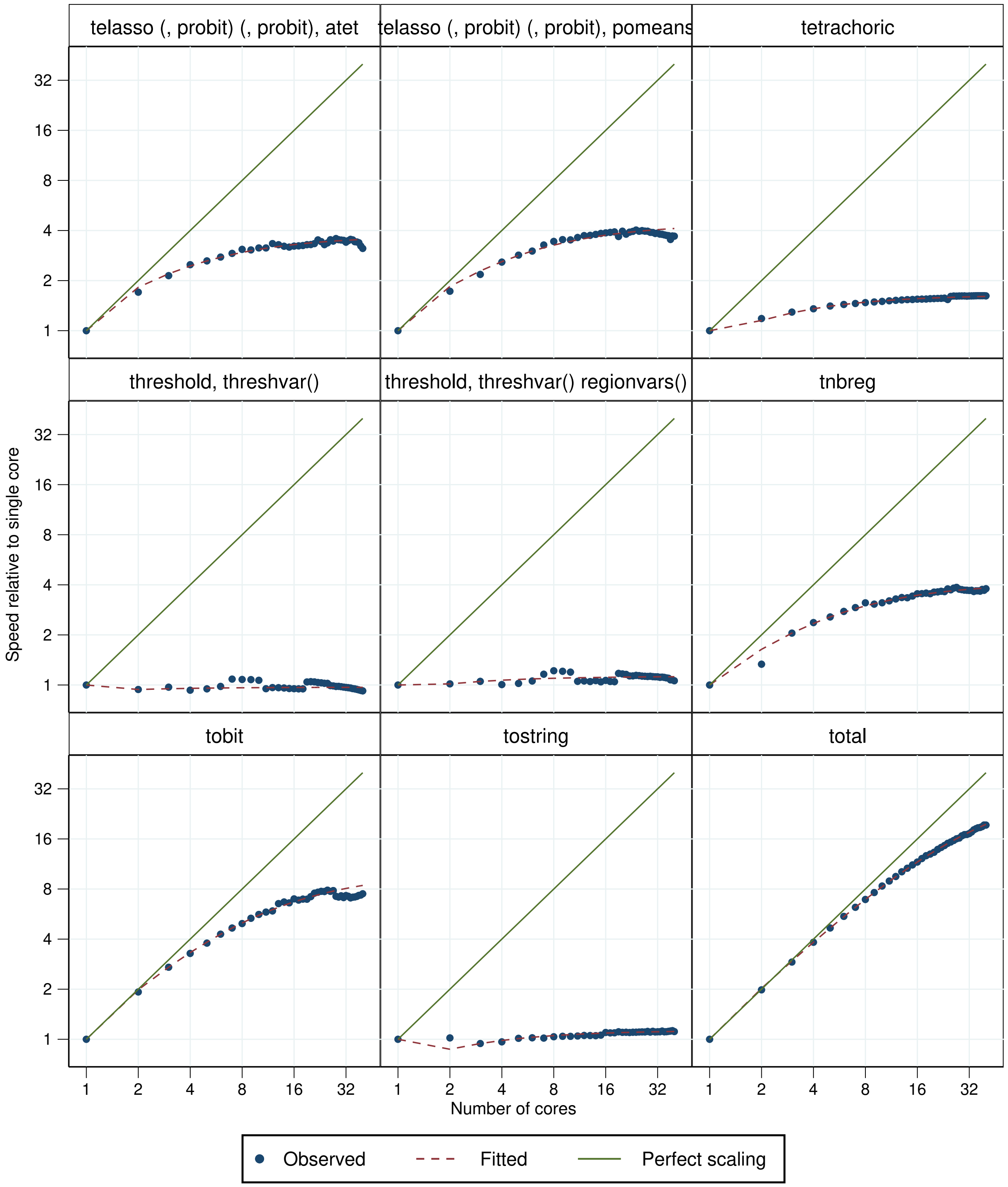


Figure 691. **Parallelization performance plots.**

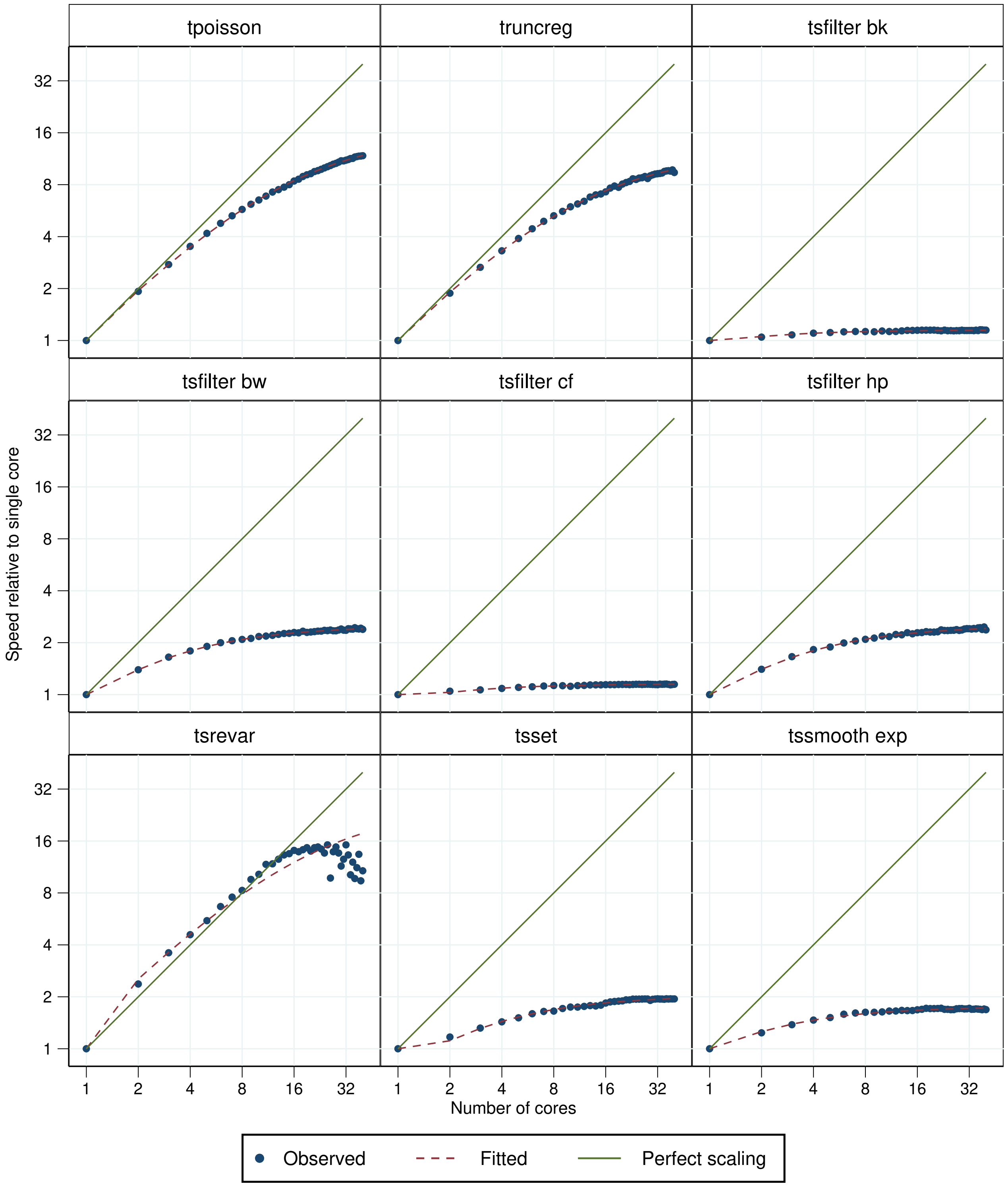


Figure 692. **Parallelization performance plots.**

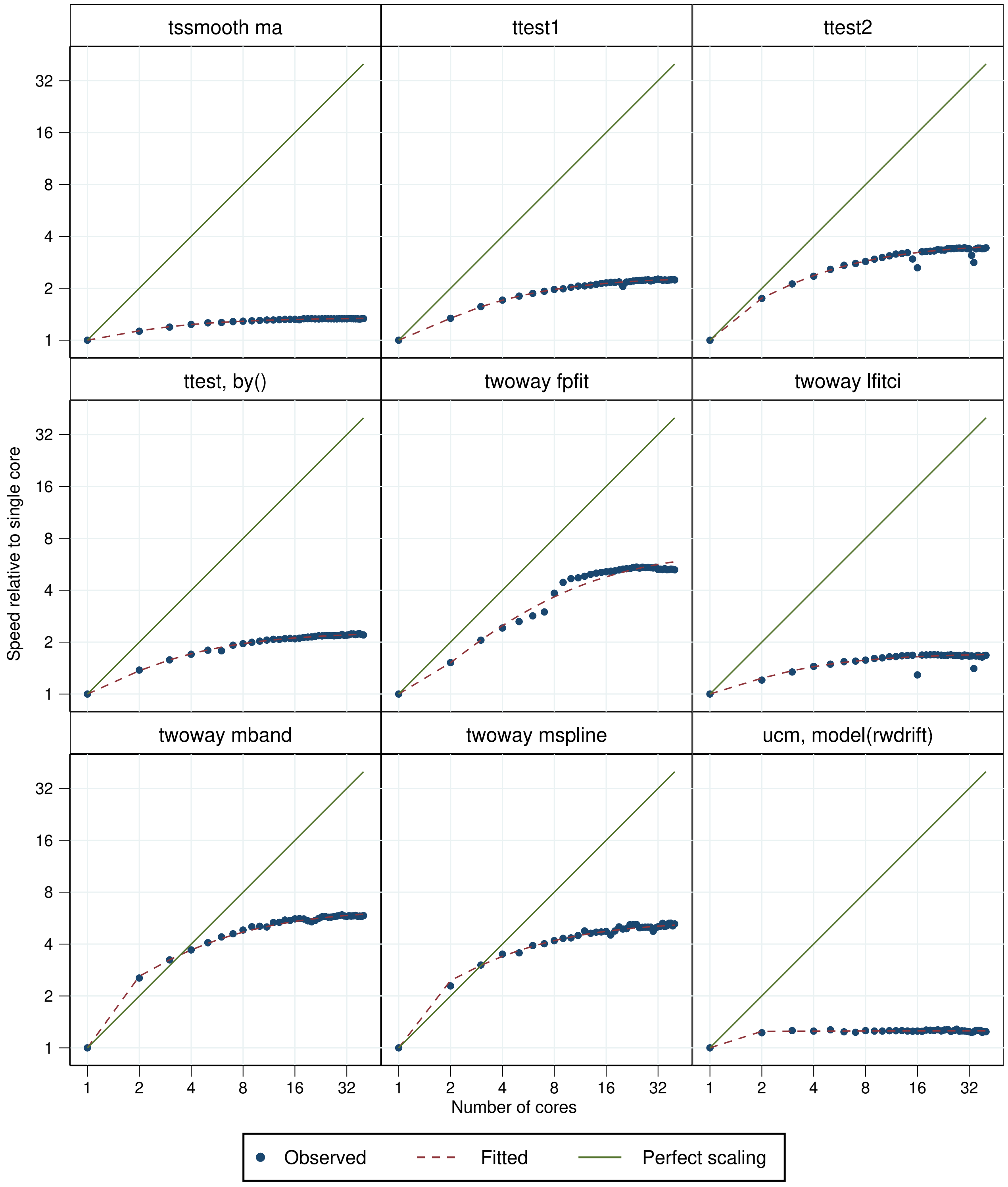


Figure 693. **Parallelization performance plots.**

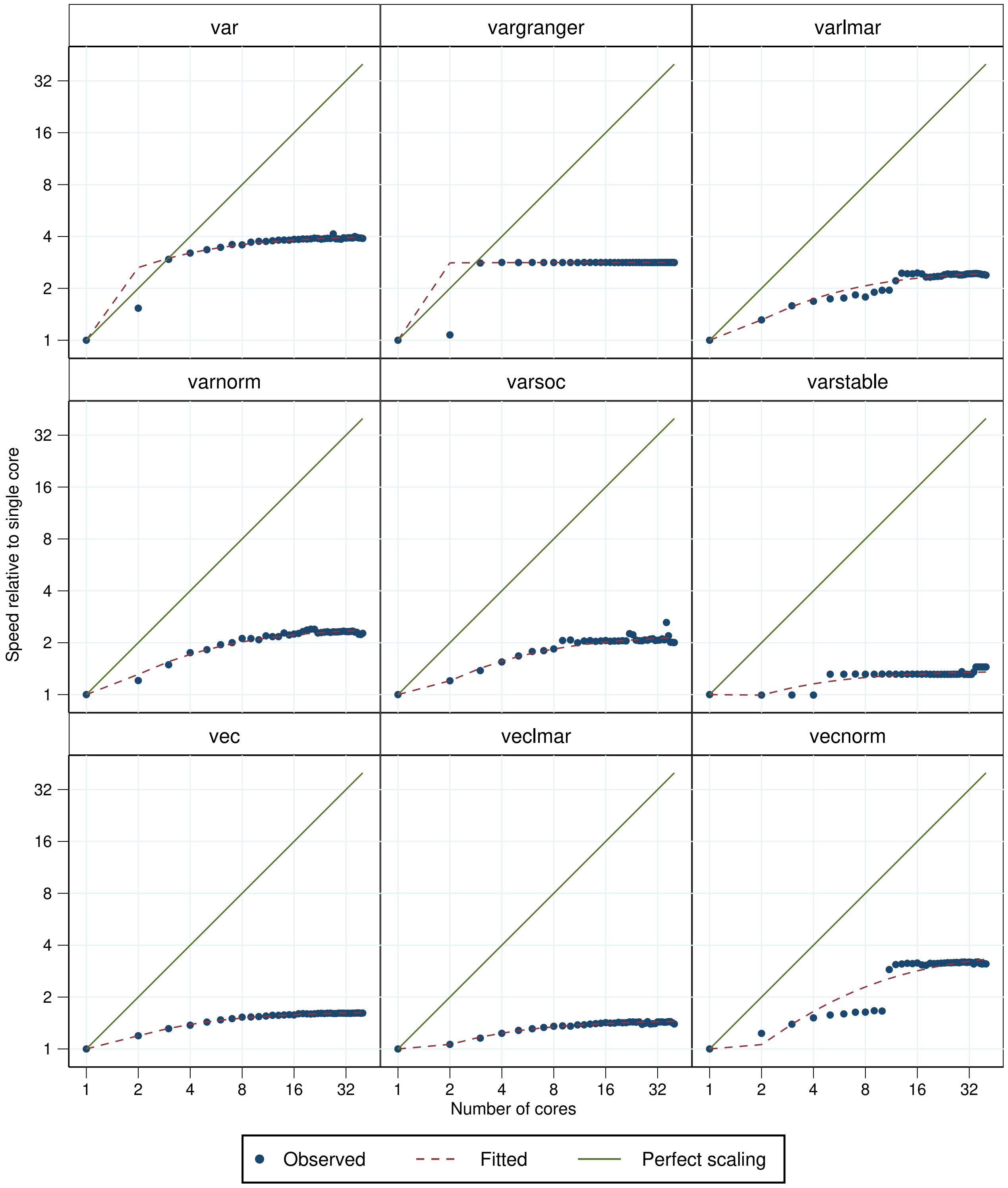


Figure 694. **Parallelization performance plots.**

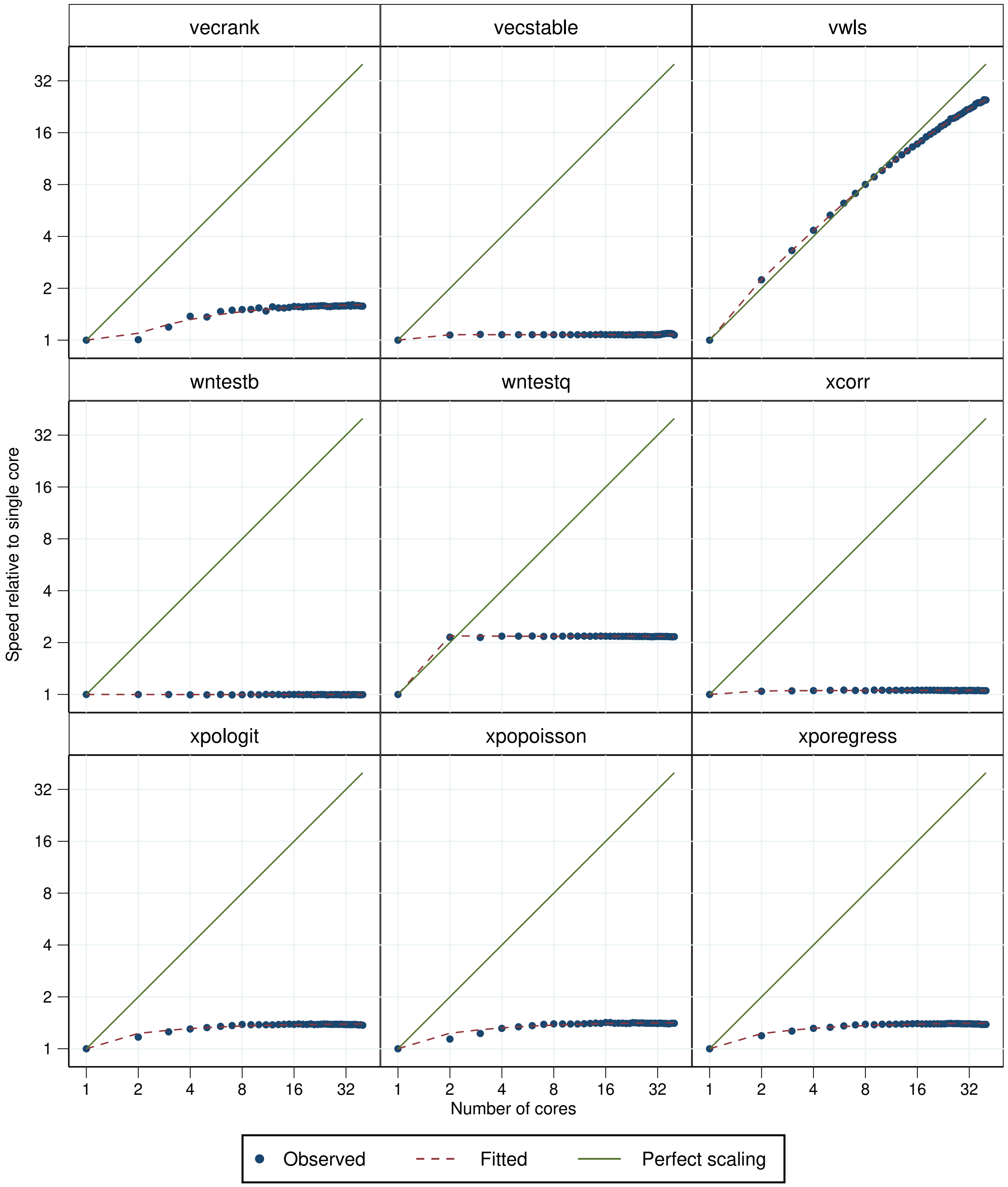


Figure 695. **Parallelization performance plots.**

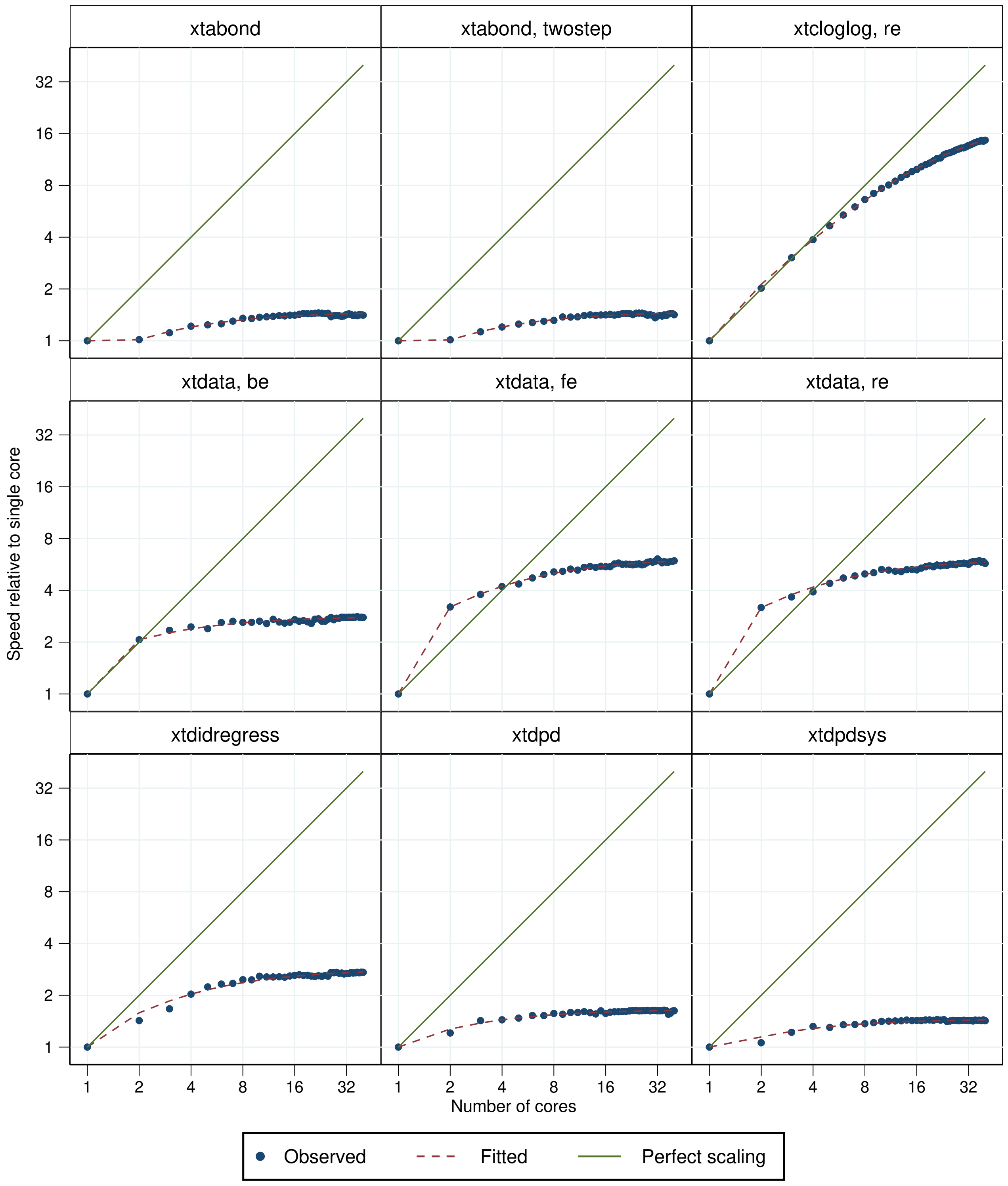


Figure 696. **Parallelization performance plots.**

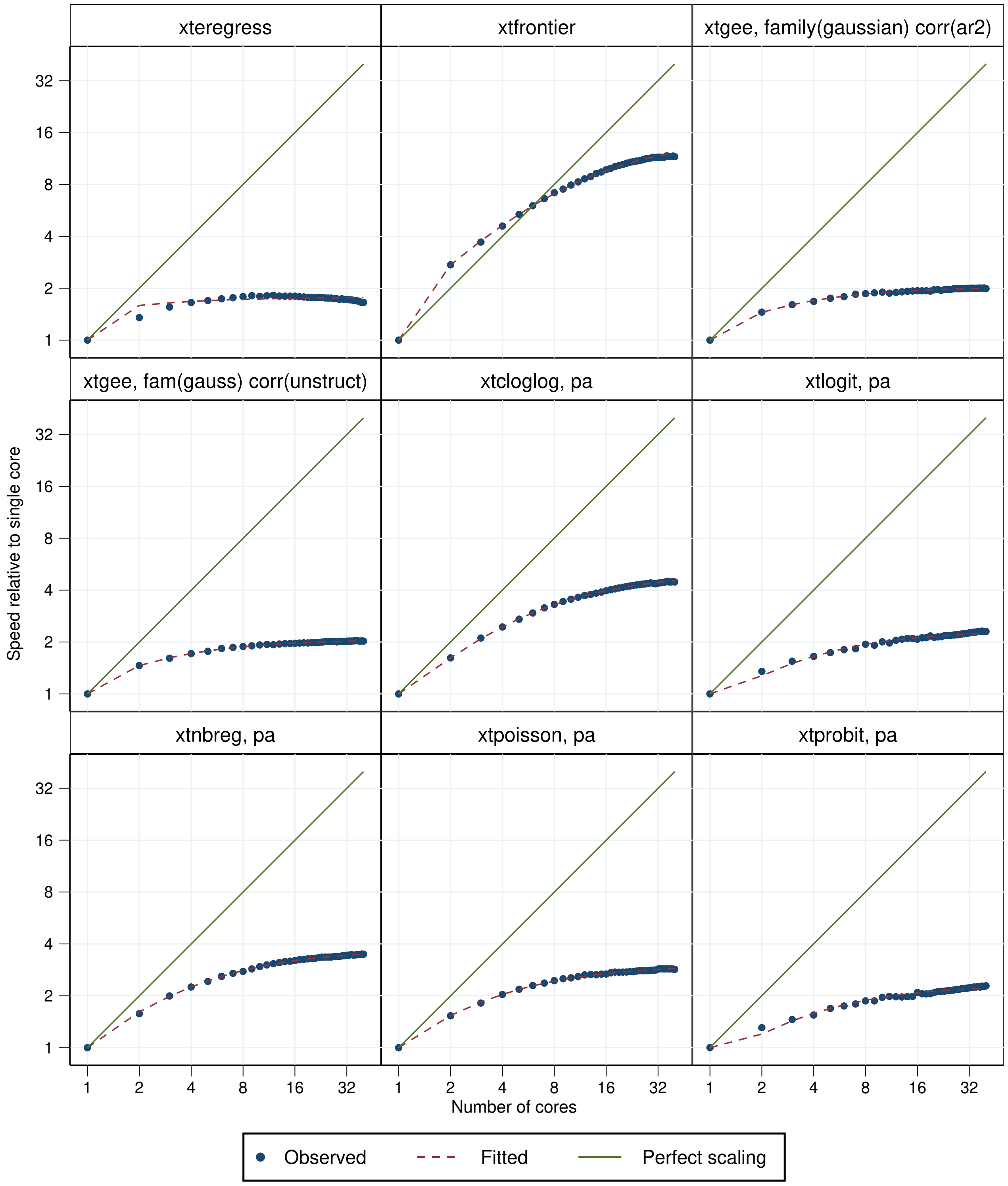


Figure 697. **Parallelization performance plots.**

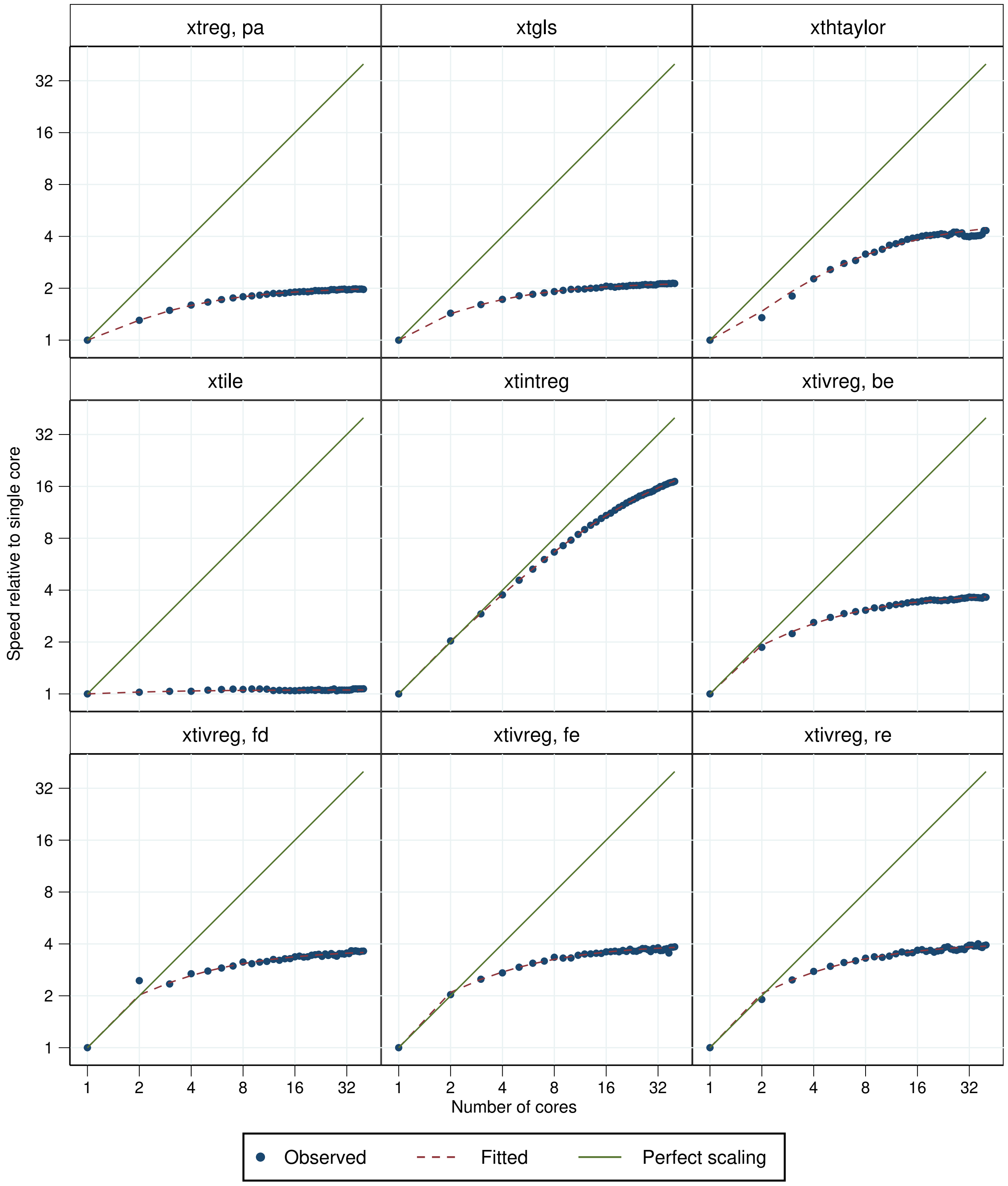


Figure 698. **Parallelization performance plots.**

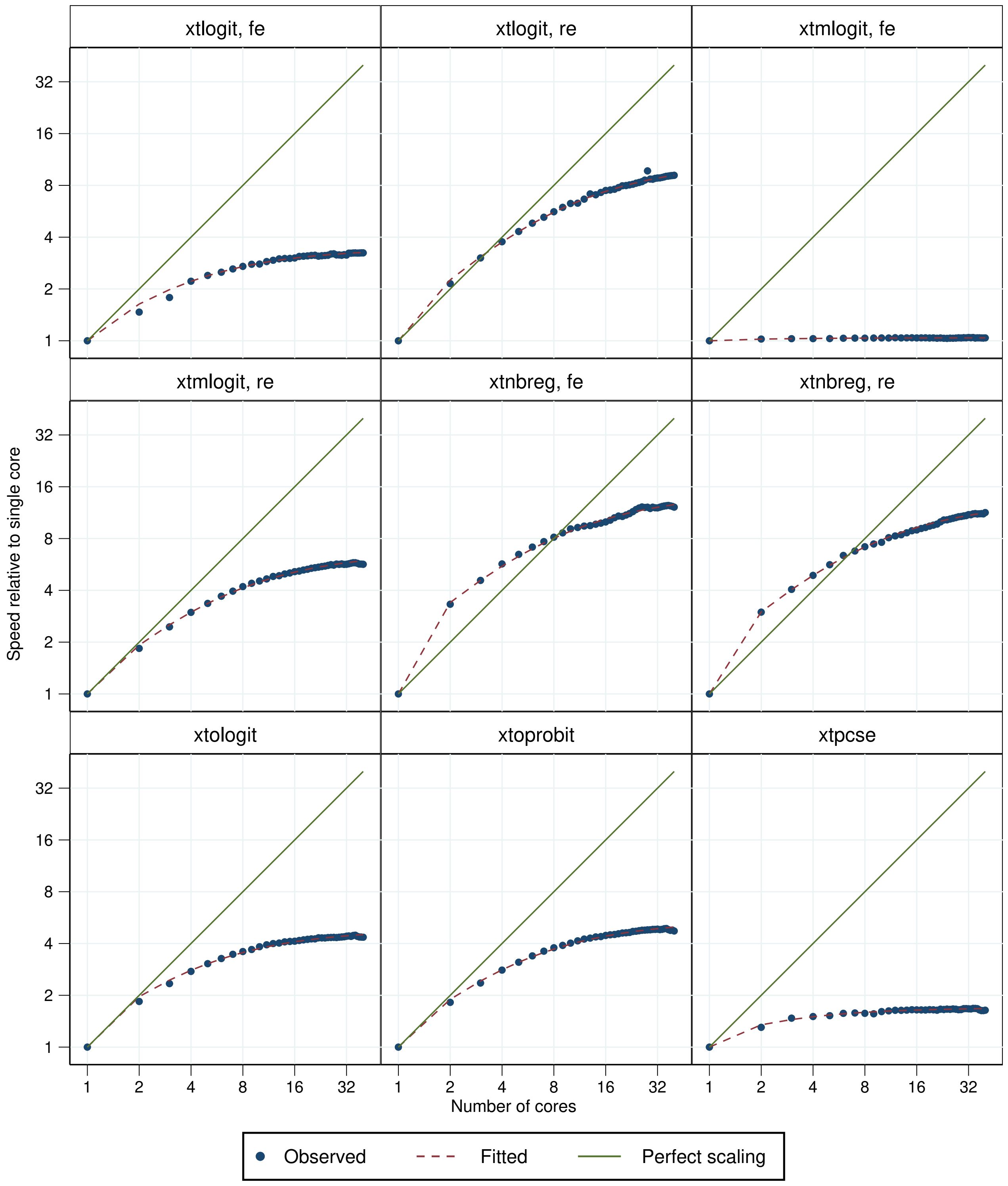


Figure 699. **Parallelization performance plots.**

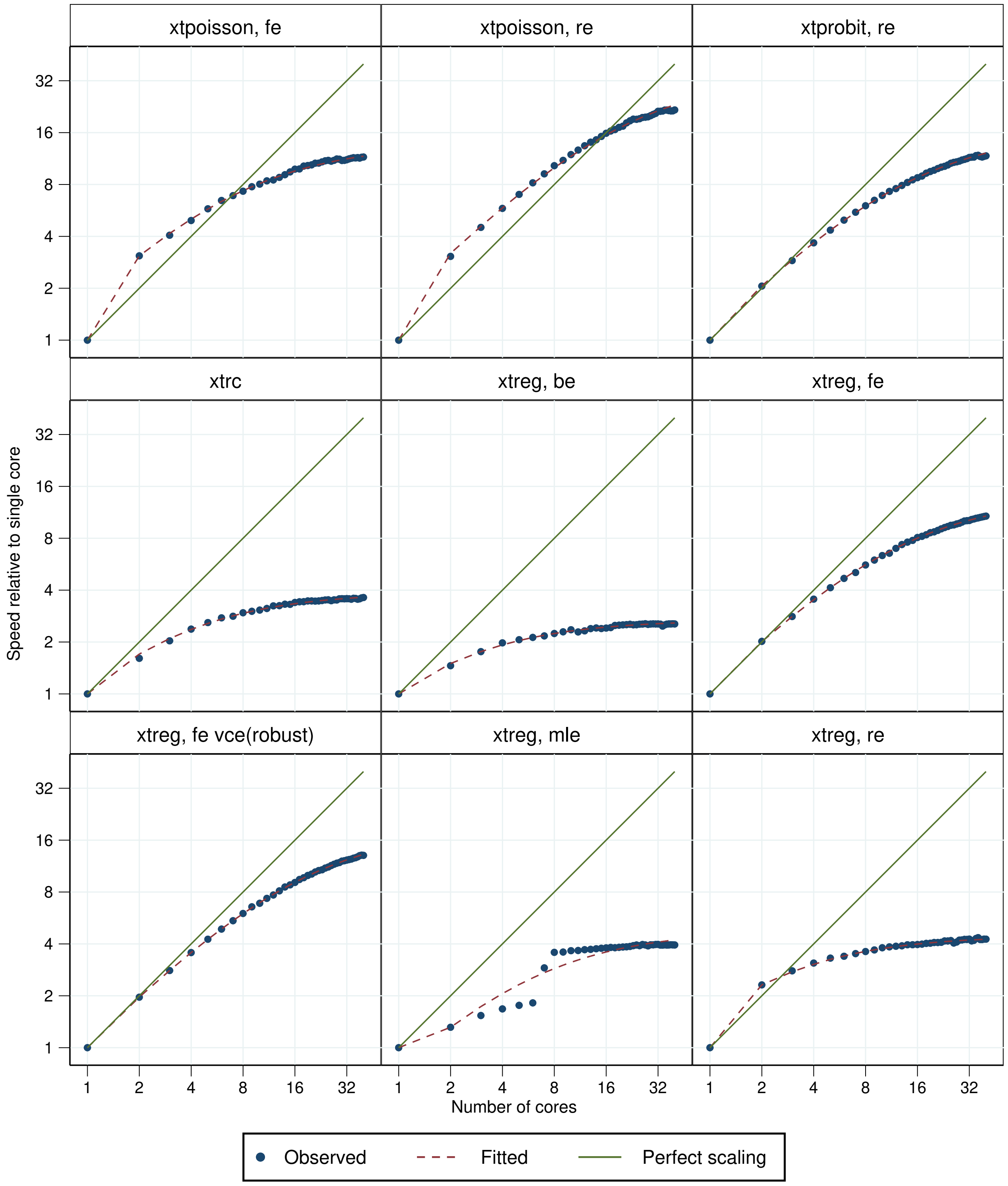


Figure 700. **Parallelization performance plots.**

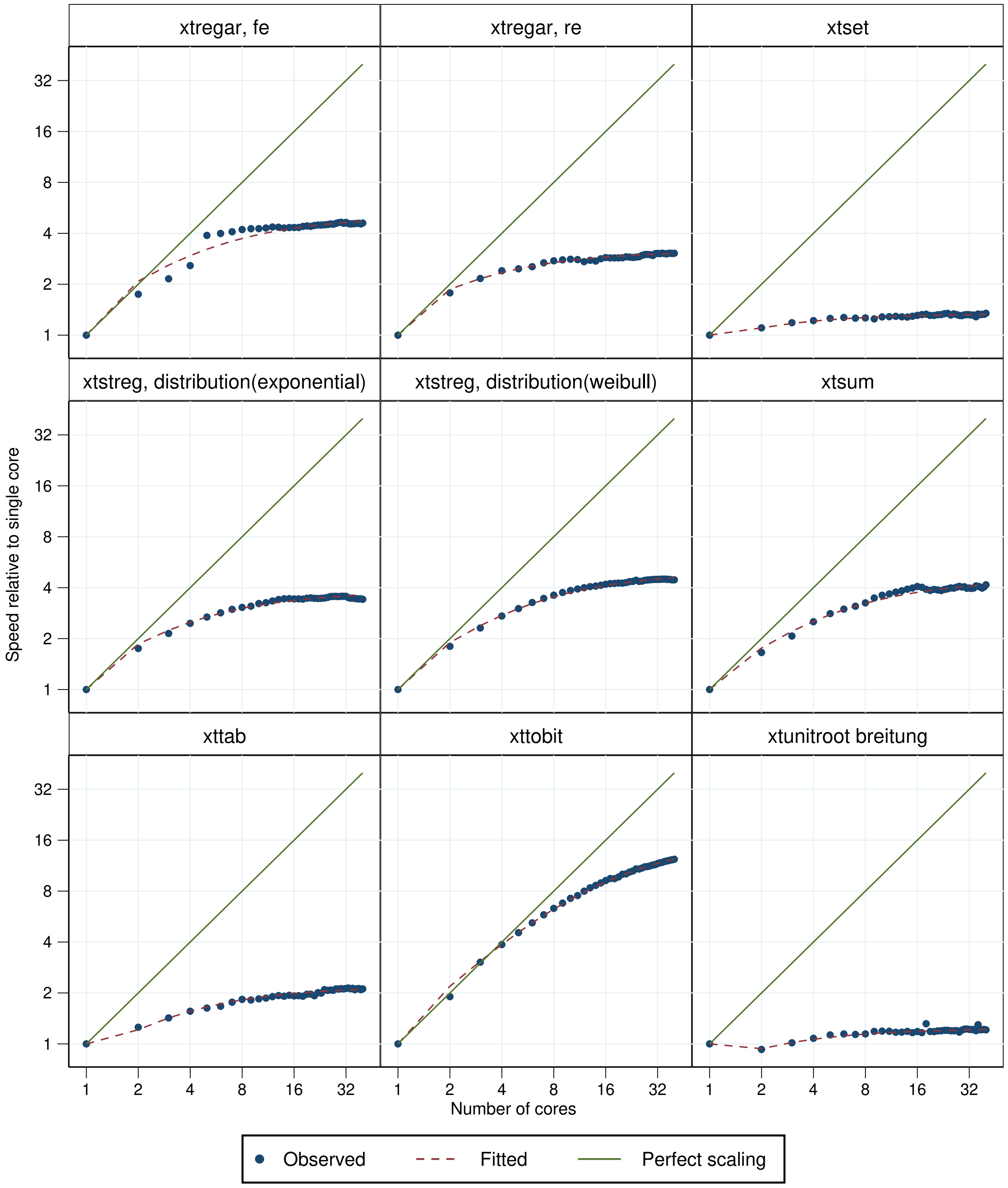


Figure 701. **Parallelization performance plots.**

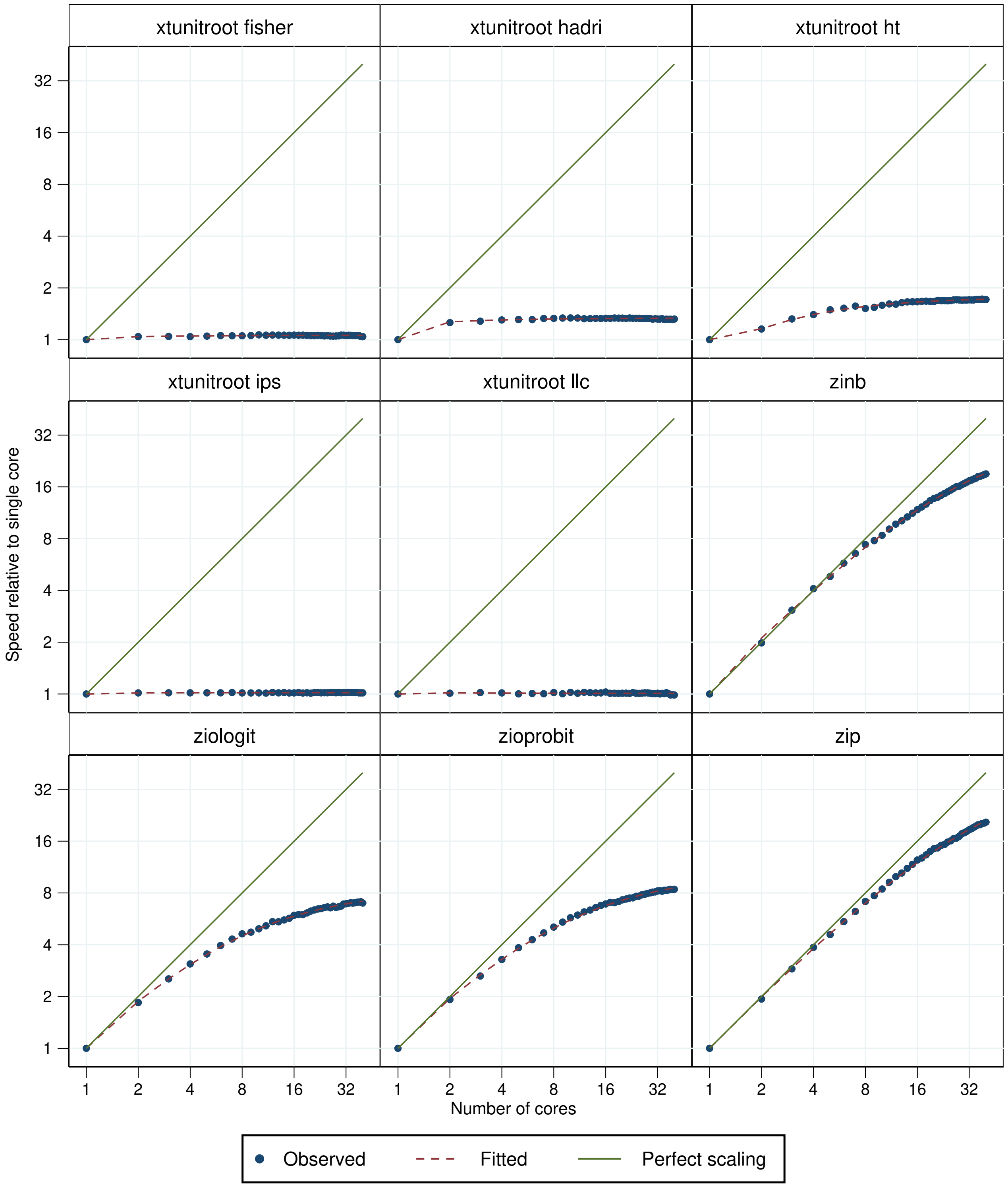


Figure 702. **Parallelization performance plots.**

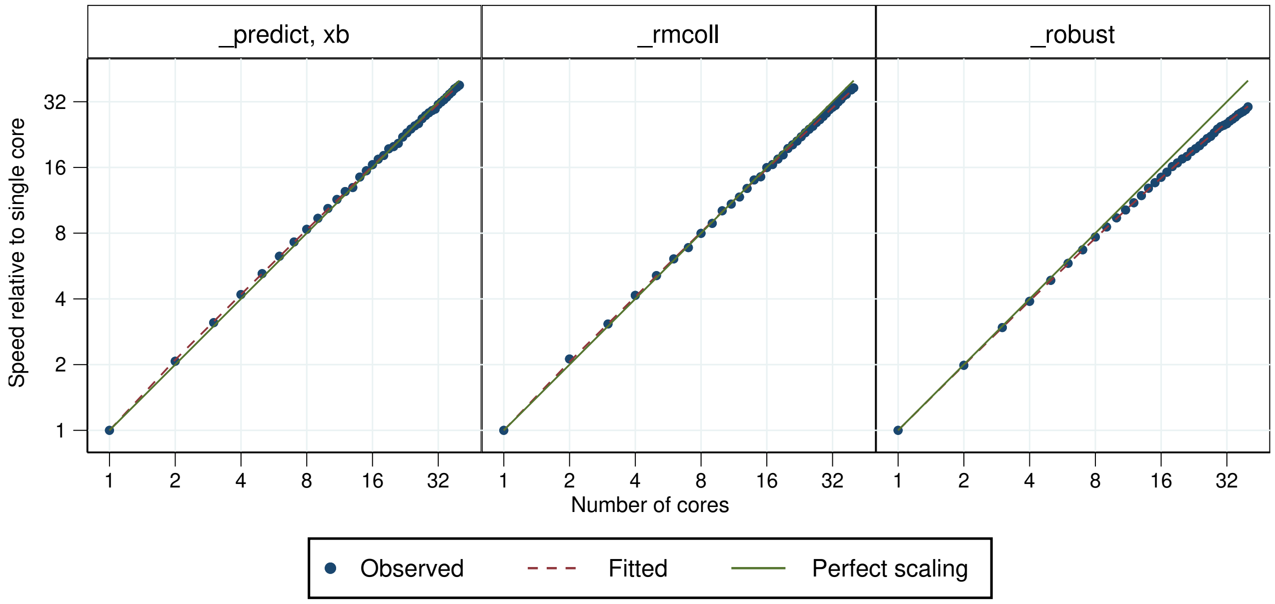


Figure 703. **Parallelization performance plots.**

# C Command names and descriptions

Table 2. Command descriptions

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

Table 2. Command descriptions

| Command | Description |
|---|---|
| `binreg` | Generalized linear models: extensions to the binomial family |
| `biplot` | Biplots |
| `biprobit` | Bivariate probit regression |
| `biprobit` (seemingly unrelated) | Seemingly unrelated probit regression |
| `bitest` | Binomial probability test |
| `blogit` | Logistic regression for grouped data |
| `boxcox` | Box–Cox regression models |
| `bprobit` | Probit regression for grouped data |
| `brier` | Brier score decomposition |
| `bsample` | Sampling with replacement |
| `bstat` | Compute and report bootstrap statistics |
| `by: generate` | Create new variables over longitudinal/panel data |
| `by: generate` (small groups) | Create new variables over longitudinal/panel data, small panels |
| `by: replace` | Replace variable values over longitudinal/panel data |
| `by: replace` (small groups) | Replace variable values over longitudinal/panel data, small panels |
| `ca` | Simple correspondence analysis |
| `candisc` | Canonical linear discriminant analysis |
| `canon` | Canonical correlations |
| `cc` | Case–control odds ratio |
| `by: cc` | Case–control odds ratio over groups |
| `centile` | Report centile and confidence interval |
| `churdle linear` | Cragg hurdle regression |
| `ci means` | Confidence intervals for means, normal distribution |
| `ci means, poisson` | Confidence intervals for means, Poisson distribution |
| `ci proportions` | Confidence intervals for proportions |

Table 2. Command descriptions

| Command | Description |
|---|---|
| `clogit` (k1 to k2 matching) | Conditional (fixed-effects) logistic regression, k1 to k2 matching |
| `clogit` (1 to k matching) | Conditional (fixed-effects) logistic regression, 1 to k matching |
| `cloglog` | Complementary log-log regression |
| `cluster averagelinkage` | Hierarchical cluster analysis—average linkage |
| `cluster centroidlinkage` | Hierarchical cluster analysis—centroid linkage |
| `cluster completelinkage` | Hierarchical cluster analysis—complete linkage |
| `cluster generate` | Generate summary and grouping variables from a cluster analysis |
| `cluster kmeans` | Kmeans cluster analysis |
| `cluster kmedians` | Kmedians cluster analysis |
| `cluster medianlinkage` | Hierarchical cluster analysis—median linkage |
| `cluster singlelinkage` | Hierarchical cluster analysis—single linkage |
| `cluster wardslinkage` | Hierarchical cluster analysis—Ward's linkage |
| `cluster waveragelinkage` | Hierarchical cluster analysis—Ward's average linkage |
| `cmclogit` | Conditional logit (McFadden's) choice model |
| `cmmprobit` | Multinomial probit choice model |
| `cmroprobit` | Rank-ordered probit choice model |
| `cnsreg` | Constrained linear regression |
| `codebook` | Describe data contents |
| `collapse` | Make dataset of summary datasets |
| `compare` | Compare two variables |
| `compress` | Compress data in memory |
| `contract` | Make dataset of frequencies and percentages |
| `corr2data` | Create dataset with specified correlation structure |
| `correlate` | Correlations (covariances) of variables or estimators |
| `corrgram` | Tabulate and graph autocorrelations |

Table 2. Command descriptions

| Command | Description |
|---|---|
| `count` | Count observations satisfying specified condition |
| `cpoisson` | Censored Poisson regression |
| `cs` | Cohort study risk-ratio |
| `by: cs` | Cohort study risk-ratio over groups |
| `ctset` | Declare data to be count-time data |
| `cttost` | Convert count-time data to survival-time data |
| `cumul` | Cumulative distribution |
| `cusum` | Cusum plots and tests for binary variables |
| `datasignature` | Determine whether data have changed |
| `decode` | Decode labeled numeric into string |
| `destring` | Convert string variables to numeric variables |
| `dfactor` | Dynamic-factor models |
| `dfgls` | DF-GLS unit-root test |
| `dfuller` | Augmented Dickey–Fuller unit-root test |
| `didregress` | Difference-in-differences estimation |
| `discrim knn` | Discriminant analysis—*k*th-nearest-neighbor |
| `discrim lda` | Discriminant analysis—linear |
| `discrim logistic` | Discriminant analysis—logistic |
| `discrim qda` | Discriminant analysis—quadratic |
| `dotplot` | Comparative scatterplots |
| `drawnorm` | Draw sample from multivariate normal distribution |
| `drop if` *exp* | Eliminate observations using `if` expression |
| `drop in` *range* | Eliminate observations using `in` range |
| `dsge` | Linearized dynamic stochastic general equilibrium model |
| `dsgenl` | Nonlinear dynamic stochastic general equilibrium model |

Table 2. Command descriptions

| Command | Description |
|---|---|
| `dslogit` | Double-selection lasso logistic regression |
| `dspoisson` | Double-selection lasso Poisson regression |
| `dsregress` | Double-selection lasso linear regression |
| `dstdize` | Direct and indirect standardization |
| `dvech` | Diagonal vech multivariate GARCH models |
| `egen group()` | Extensions to generate—create grouping variable |
| `by: egen mean` | Extensions to generate—create means over groups |
| `eivreg` | Errors-in-variables regression |
| `encode` | Encode string into numeric |
| `eregress` | Extended linear regression with endogenous covariates, treatement assignment, and sample selection |
| `esize twosample` | Effect size for two independent samples using groups |
| `esize unpaired` | Effect size for two independent samples using variables |
| `eteffects (exponential), ate` | Endogenous treatment-effects estimation, exponential-mean model, average treatment effect in population |
| `eteffects (linear), ate` | Endogenous treatment-effects estimation, linear model, average treatment effect in population |
| `eteffects (linear), pomeans` | Endogenous treatment-effects estimation, linear model, potential-outcome means |
| `eteffects (probit), ate` | Endogenous treatment-effects estimation, probit model, average treatment effect in population |
| `etpoisson` | Poisson regression with endogenous treatment effects |
| `etregress, poutcomes` | Linear regression with endogenous treatment effects, ML estimation with potential outcomes |
| `etregress, twostep` | Linear regression with endogenous treatment effects, two-step estimation |
| `exlogistic` | Exact logistic regression |
| `expand` # | Duplicate observations |
| `expand` *varname* | Duplicate observations using a variable |
| `expandcl` # | Duplicate clustered observations |
| `expandcl` *varname* | Duplicate clustered observations using a variable |
| `expoisson` | Exact Poisson regression |

Table 2. Command descriptions

| Command | Description |
|---|---|
| `factor` | Factor analysis |
| `fcast compute` | Dynamic forecasts after VAR or VEC estimation |
| `fillin` | Rectangularize dataset |
| `fmm 2: poisson` | Finite mixture model with two Poisson outcomes |
| `fmm 2: regress` | Finite mixture model with two linear outcomes |
| `fmm 3: poisson` | Finite mixture model with three Poisson outcomes |
| `fmm 3: regress` | Finite mixture model with three linear outcomes |
| `fracreg probit` | Fractional probit regression |
| `frontier` | Stochastic frontier models |
| `fvrevar` (factors) | Create indicators for factor variables |
| `fvrevar` (interaction) | Create indicators for factor variables—interactions |
| `generate` (small expressions) | Create or change contents of variable—small expressions |
| `generate` | Create or change contents of variable |
| `glm, family(gamma)` | Generalized linear models—gamma distribution |
| `glm, family(gaussian)` | Generalized linear models—Gaussian distribution |
| `glm, family(igaussian)` | Generalized linear models—inverse Gaussian distribution |
| `glm, family(nbinomial)` | Generalized linear models—negative binomial distribution |
| `glm, family(poisson)` | Generalized linear models—Poisson distribution |
| `glogit` | Weighted least-squares logistic regression for grouped data |
| `gmm` | Generalized method of moments estimation |
| `gmm` (with derivatives) | Generalized method of moments estimation with derivatives |
| `gprobit` | Weighted least-squares probit regression for grouped data |
| `graph bar` | Bar charts |
| `graph box` | Box plots |
| `graph pie` | Pie charts |

Table 2. Command descriptions

| Command | Description |
|---|---|
| `grmeanby` | Graph means and medians by categorical variables |
| `gsem, oprobit (CFA, 2-level)` | Ordered probit multilevel confirmatory factor analysis |
| `gsem, oprobit (CFA)` | Ordered probit confirmatory factor analysis |
| `gsem, logit group()` | GSEM: Logistic regression on 5 groups |
| `gsem, group()` | GSEM: Linear regression on 5 groups |
| `gsem, ologit group()` | GSEM: Ordinal logistic regression on 5 groups |
| `gsem, poisson group()` | GSEM: Poisson regression on 5 groups |
| `gsort` | Ascending and descending sort |
| `hausman` | Hausman specification test |
| `heckman` | Heckman selection model—maximum likelihood estimator |
| `heckman, twostep` | Heckman selection model—two-step estimator |
| `heckoprobit` | Ordered probit model with sample selection |
| `heckpoisson` | Poisson regression with sample selection |
| `heckprob` | Probit model with selection |
| `hetoprobit` | Heteroskedastic ordered probit regression |
| `hetprob` | Heteroskedastic probit model |
| `hetregress` | Heteroskedastic linear regression, ML estimation |
| `hetregress, twostep` | Heteroskedastic linear regression, two-step estimation |
| `histogram` | Histograms for continuous and categorical variables |
| `hotelling` | Hotelling's $T$-squared generalized means test |
| `icc, mixed` | Intraclass correlations for two-way mixed-effects model |
| `icc` (one-way) | Intraclass correlations for one-way random-effects model |
| `icc` (two-way) | Intraclass correlations for two-way random-effects model |
| `import delimited` | Import delimited text data |
| `intreg` | Interval regression |

Table 2. Command descriptions

| Command | Description |
|---|---|
| `ir` | Incidence-rate ratio |
| `by: ir` | Incidence-rate ratio over groups |
| `irf create` | Create IRFs and FEVDs after VAR and VEC estimation |
| `irt 1pl` | Item response theory one-parameter logistic model |
| `irt 2pl` | Item response theory two-parameter logistic model |
| `irt 3pl` | Item response theory three-parameter logistic model |
| `irt grm` | Item response theory graded response model |
| `irt nrm` | Item response theory nominal response model |
| `irt pcm` | Item response theory partial credit model |
| `irt rsm` | Item response theory rating scale model |
| `istdize` | Indirect standardization |
| `ivpoisson cfunction` | Poisson regression with endogenous regressors, control-function estimator |
| `ivpoisson gmm, additive` | Poisson regression with endogenous regressors, GMM with additive regression errors |
| `ivpoisson gmm, multiplicative` | Poisson regression with endogenous regressors, GMM multiplicative regression errors |
| `ivprobit` | Probit model with endogenous regressors |
| `ivregress 2sls` | Instrumental-variables regression—two-stage least squares |
| `ivregress gmm` | Instrumental-variables regression—GMM |
| `ivregress liml` | Instrumental-variables regression—LIML |
| `ivtobit` | Tobit model with endogenous regressors |
| `kap` | Interrater agreement |
| `kappa` | Interrater agreement |
| `kdensity` | Univariate kernel density estimation |
| `keep if` *exp* | Retain observations using `if` expression |
| `keep in` *range* | Retain observations using `in` range |
| `keep` *varlist* | Retain variables |

Table 2. Command descriptions

| Command | Description |
|---|---|
| `ksmirnov` | Kolmogorov–Smirnov equality-of-distributions test |
| `ksmirnov, by()` | Kolmogorov–Smirnov equality-of-distributions test over groups |
| `ktau` | Kendall's rank correlation coefficients |
| `kwallis` | Kruskal–Wallis equality-of-populations rank test |
| `ladder` | Ladder of powers |
| `lasso linear` | Linear lasso for prediction and model selection |
| `lasso logit` | Logistic lasso for prediction and model selection |
| `lasso poisson` | Poisson lasso for prediction and model selection |
| `gsem, lclass(C 2)` | Latent Class Analysis, logit outcomes, 2 classes |
| `gsem, lclass(C 3)` | Latent Class Analysis, logit outcomes, 3 classes |
| `levelsof` | Levels of variable |
| `loadingplot` | Score and loading plots after `factor` and `pca` |
| `logistic` | Logistic regression, reporting odds ratios |
| `logit` | Logistic regression, reporting coefficients |
| `loneway` | Large one-way ANOVA, random effects, and reliability |
| `lowess` | Lowess smoothing |
| `lpoly` | Kernel-weighted local polynomial smoothing |
| `ltable` | Life tables for survival data |
| `manova` (one-way) | Multivariate analysis of variance and covariance, one-way |
| `manova` (two-way) | Multivariate analysis of variance and covariance, two-way |
| `margins` | Marginal means and predictive margins |
| `margins, dydx() exp()` | Marginal effects of an expression |
| `margins, dydx()` | Marginal effects |
| `margins, exp()` | Predictive margins of an expression |
| `markout` | Mark observations for exclusion |

Table 2. Command descriptions

| Command | Description |
|---|---|
| `marksample` | Mark observations for inclusion |
| `marksample if` *exp* | Mark observations for inclusion, with `if` expression |
| `matrix accum` | Form cross-product matrices of variables over observations |
| `matrix eigenvalues` | Eigenvalues of a matrix |
| `matrix score` | Inner product of matrix with variables over observations |
| `matrix svd` | Singular value decomposition |
| `matrix symeigen` | Eigenvalues of a symmetric matrix |
| `matrix syminv` | Inversion of a symmetric matrix |
| `mca` | Multiple and joint correspondence analysis |
| `mcc` | Matched case–control studies |
| `mds` | Multidimensional scaling for two-way data |
| `mdslong` | Multidimensional scaling of proximity data in long format |
| `mean` | Estimate means |
| `mecloglog` | Multilevel mixed-effects complimentary log-log regression |
| `median` | Equality tests on unmatched data |
| `meintreg` | Multilevel mixed-effects interval regression |
| `melogit` | Multilevel mixed-effects logistic regression |
| `menbreg, dispersion(constant)` | Multilevel mixed-effects negative binomial regression, constant dispersion |
| `menbreg, dispersion(mean)` | Multilevel mixed-effects negative binomial regression, mean dispersion |
| `menl` | Nonlinear mixed-effects regression for a linear outcome |
| `meologit` | Multilevel mixed-effects ordered logistic regression |
| `meoprobit` | Multilevel mixed-effects ordered probit regression |
| `mepoisson` | Multilevel mixed-effects Poisson regression |
| `meprobit` | Multilevel mixed-effects probit regression |
| `mestreg, distribution(exp)` | Multilevel mixed-effects survival models, exponential distribution |

Table 2. Command descriptions

| Command | Description |
|---|---|
| `mestreg, distribution(weibull)` | Multilevel mixed-effects survival models, Weibull distribution |
| `metobit` | Multilevel mixed-effects Tobit regression |
| `mgarch` | Multivariate generalized autoregressive conditional-heteroskedasticity (MGARCH) models |
| `mhodds` | Ratio of odds of failure for two categories |
| `mhodds` (adjusted) | Ratio of odds of failure for two categories adjusting for levels |
| `by: mhodds` | Ratio of odds of failure for two categories over groups |
| `mhodds` (trend) | Ratio of odds of failure testing for trend |
| `mi estimate: logit (flong)` | Logistic regression with multiply imputed data—`flong` style data |
| `mi estimate: logit (flongsep)` | Logistic regression with multiply imputed data—`flongsep` style data |
| `mi estimate: logit (mlong)` | Logistic regression with multiply imputed data—`mlong` style data |
| `mi estimate: logit (wide)` | Logistic regression with multiply imputed data—`wide` style data |
| `mi estimate: mlogit` | Multinomial logistic regression with multiply imputed data |
| `mi estimate: ologit` | Ordered logistic regression with multiply imputed data |
| `mi estimate: regress (flong)` | Linear regression with multiply imputed data—`flong` style data |
| `mi estimate: regress (flongsep)` | Linear regression with multiply imputed data—`flongsep` style data |
| `mi estimate: regress (mlong)` | Linear regression with multiply imputed data—`mlong` style data |
| `mi estimate: regress (wide)` | Linear regression with multiply imputed data—`wide` style data |
| `mi impute chained (flong)` | Impute missing values using chained equations–`flong` style data |
| `mi impute chained (flongsep)` | Impute missing values using chained equations–`flongsep` style data |
| `mi impute chained (mlong)` | Impute missing values using chained equations–`mlong` style data |
| `mi impute chained (wide)` | Impute missing values using chained equations–`wide` style data |
| `mi impute logit (flong)` | Impute missing values using logistic regression—`flong` style data |
| `mi impute logit (flongsep)` | Impute missing values using logistic regression—`flongsep` style data |
| `mi impute logit (mlong)` | Impute missing values using logistic regression—`mlong` style data |
| `mi impute logit (wide)` | Impute missing values using logistic regression—`wide` style data |

Table 2. Command descriptions

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

Table 2. Command descriptions

| Command | Description |
|---|---|
| `mvencode` | Recode missing values to numeric |
| `mvreg` | Multivariate regression |
| `mvtest correlations` | Multivariate test—correlations |
| `mvtest covariances` | Multivariate test—covariances |
| `mvtest means, heterogeneous` | Multivariate test—means, heterogenous covariances |
| `mvtest means, homogeneous` | Multivariate test—means, homogeneous covariances |
| `mvtest means, lr` | Multivariate test—means, likelihood-ratio test |
| `mvtest normality` | Multivariate test—normality |
| `nbreg` | Negative binomial regression |
| `newey` | Regression with Newey–West standard errors |
| `nl` | Nonlinear least-squares estimation |
| `nlogit` | Nested logit regression |
| `nlsur` | Estimation of nonlinear systems of equations |
| `npregress kernel` | Nonparametric kernel regression |
| `nptrend` | Test for trend across ordered groups, Cusick |
| `nptrend_carmitage` | Test for trend across ordered groups, Cochran-Armitage |
| `nptrend_jterpstra` | Test for trend across ordered groups, Jonckheere-Terpstra |
| `nptrend_linear` | Test for trend across ordered groups, linear-by-linear |
| `ologit` | Ordered logistic regression |
| `ologit, vce(cluster)` | Ordered logistic regression, cluster–robust standard errors |
| `ologit, vce(robust)` | Ordered logistic regression, robust (Huber/White) standard errors |
| `oneway` | One-way analysis of variance |
| `oprobit` | Ordered probit regression |
| `oprobit, vce(cluster)` | Ordered probit regression, cluster–robust standard errors |
| `oprobit, vce(robust)` | Ordered probit regression, robust (Huber/White) standard errors |

Table 2. Command descriptions

| Command | Description |
|---|---|
| `orthog` | Orthogonalize variables and compute orthogonal polynomials |
| `pca` | Principal component analysis |
| `pcorr` | Partial correlation coefficients |
| `pctile` | Create variable containing percentiles |
| `pergram` | Periodogram |
| `pkcollapse` | Generate pharmacokinetic measurement dataset |
| `pkexamine` | Calculate pharmacokinetic measures |
| `pksumm` | Summarize pharmacokinetic data |
| `poisson` | Poisson regression |
| `poisson, vce(cluster)` | Poisson regression, cluster–robust standard errors |
| `poisson, exposure()` | Poisson regression, with exposure |
| `poisson, vce(robust)` | Poisson regression, robust (Huber/White) standard errors |
| `pologit` | Partialling-out lasso logistic regression |
| `popoisson` | Partialling-out lasso Poisson regression |
| `poregress` | Partialling-out lasso linear regression |
| `pperron` | Phillips–Perron unit-root test |
| `prais` | Prais–Winsten and Cochrane–Orcutt regression |
| `predict, cooksd` | Obtain Cook's distance predictions after estimation |
| `predict, covratio` | Obtain COVRATIO predictions after estimation |
| `predict, dfbeta` | Obtain DFBETAs for a variable after estimation |
| `predict, dfits` | Obtain DFITS predictions after estimation |
| `predict, e` | Obtain predictions given upper and lower truncation after estimation |
| `predict, leverage` | Obtain leverage of observations after estimation |
| `predict, pr` | Obtain probability-in-range predictions after estimation |
| `predict, residuals` | Obtain residuals after estimation |

Table 2. Command descriptions

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

Table 2. Command descriptions

| Command | Description |
|---|---|
| `replace` (small expressions) | Create or change contents of variable, simple expression |
| `reshape long` | Convert data from wide to long |
| `reshape wide` | Convert data from long to wide |
| `robvar` | Robust tests for equality of variance |
| `rocfit` | Fit ROC models |
| `roctab` | Receiver operating characteristic (ROC) analysis |
| `rotate` | Orthogonal and oblique rotations after `factor` and `pca` |
| `rotatemat` | Orthogonal and oblique rotations of a Stata matrix |
| `rreg` | Robust regression |
| `runtest` | Test for random order |
| `scobit` | Skewed logistic regression |
| `scoreplot` | Score and loading plots after `factor` and `pca` |
| `screeplot` | Scree plot of eigenvalues |
| `sdtest1` | Variance-comparison test against constant |
| `sdtest2` | Variance-comparison test between variables |
| `sdtest, by()` | Variance-comparison test over groups |
| `sem, method(adf) (CFA)` | Confirmatory factor analysis, ADF estimation |
| `sem, method(ml) (CFA)` | Confirmatory factor analysis, ML estimation |
| `sem, method(mlmv) (CFA)` | Confirmatory factor analysis, ML estimation with missing values |
| `sem (SEM latent)` | Structural equations model with latent variables, ML estimation |
| `sem (SEM observed)` | Structural equations model on observed variables, ML estimation |
| `separate` | Create separate variables |
| `sfrancia` | Shapiro–Francia test for normality |
| `signrank` | Equality tests on matched data |
| `signtest` | Equality tests on matched data |

Table 2. Command descriptions

| Command | Description |
|---|---|
| `sktest` | Skewness and kurtosis test for normality |
| `slogit` | Stereotype logistic regression |
| `sort` | Sort data |
| `spearman` | Spearman's rank correlation coefficients |
| `sspace` | State-space models |
| `stack` | Stack data |
| `stci` | Confidence intervals for means and percentiles of survival time |
| `stcox` | Fit Cox proportional hazards model |
| `stcrreg` | Competing-risks regression |
| `stgen` | Generate variables reflecting entire histories |
| `stintcox` | Cox proportional hazards model for interval-censored survival-time data |
| `stintreg, d(exponential)` | Fit parametric models for interval-censored survival-time data, exponential distribution |
| `stintreg, d(weibull)` | Fit parametric models for interval-censored survival-time data, Weibull distribution |
| `stir` | Report incidence-rate comparison |
| `stmc` | Calculate rate ratios with the Mantel–Cox method |
| `by: stmc` | Calculate rate ratios with the Mantel–Cox method over groups |
| `stmh` | Calculate rate ratios with the Mantel–Haenszel method |
| `by: stmh` | Calculate rate ratios with the Mantel–Haenszel method over groups |
| `stptime` | Calculate person-time, incidence rates, and SMR |
| `strate` | Tabulate failure rates and rate ratios |
| `streg, distribution(exponential)` | Fit parametric survival models, exponential distribution |
| `streg, dist(exp) vce(cluster)` | Fit parametric survival models, exponential distribution with cluster–robust standard errors |
| `streg, dist(exp) frailty()` | Fit parametric survival models, exponential distribution with individual frailty |
| `streg, dist(exp) frailty() shared()` | Fit parametric survival models, exponential distribution with shared frailty |
| `streg, dist(exp) vce(robust)` | Fit parametric survival models, exponential distribution with robust standard errors |

Table 2. Command descriptions

| Command | Description |
|---|---|
| `streg, distribution(ggamma)` | Fit parametric survival models, generalized-gamma distribution |
| `streg, dist(ggamma) vce(cluster)` | Fit parametric survival models, generalized-gamma distribution with cluster–robust standard errors |
| `streg, dist(ggamma) vce(robust)` | Fit parametric survival models, generalized-gamma distribution with robust standard errors |
| `streg, distribution(gompertz)` | Fit parametric survival models, Gompertz distribution |
| `streg, dist(gompertz) vce(cluster)` | Fit parametric survival models, Gompertz distribution with cluster–robust standard errors |
| `streg, dist(gompertz) frailty()` | Fit parametric survival models, Gompertz distribution with individual frailty |
| `streg, dist(gomp) frailty() shared()` | Fit parametric survival models, Gompertz distribution with shared frailty |
| `streg, dist(gompertz) vce(robust)` | Fit parametric survival models, Gompertz distribution with robust standard errors |
| `streg, distribution(llogistic)` | Fit parametric survival models, log-logistic distribution |
| `streg, dist(llogistic) vce(cluster)` | Fit parametric survival models, log-logistic distribution with cluster–robust standard errors |
| `streg, dist(llogistic) frailty()` | Fit parametric survival models, log-logistic distribution with individual frailty |
| `streg, dist(llog) frailty() shared()` | Fit parametric survival models, log-logistic distribution with shared frailty |
| `streg, dist(llogistic) vce(robust)` | Fit parametric survival models, log-logistic distribution with robust standard errors |
| `streg, distribution(lnormal)` | Fit parametric survival models, log-normal distribution |
| `streg, dist(lnormal) vce(cluster)` | Fit parametric survival models, log-normal distribution with cluster–robust standard errors |
| `streg, dist(lnormal) frailty()` | Fit parametric survival models, log-normal distribution with individual frailty |
| `streg, dist(lnorm) frailty() shared()` | Fit parametric survival models, log-normal distribution with shared frailty |
| `streg, dist(lnormal) vce(robust)` | Fit parametric survival models, log-normal distribution with robust standard errors |
| `streg, distribution(weibull)` | Fit parametric survival models, Weibull distribution |
| `streg, dist(weibull) vce(cluster)` | Fit parametric survival models, Weibull distribution with cluster–robust standard errors |
| `streg, dist(weibull) frailty()` | Fit parametric survival models, Weibull distribution with individual frailty |
| `streg, dist(weib) frailty() shared()` | Fit parametric survival models, Weibull distribution with shared frailty |
| `streg, dist(weibull) vce(robust)` | Fit parametric survival models, Weibull distribution with robust standard errors |
| `sts generate` | Create new variables containing survival, hazard, and related functions |

Table 2. Command descriptions

| Command | Description |
|---|---|
| `sts list` | Compute and list survival and related functions |
| `sts test` | Test the equality of the survival function across groups |
| `stset` | Declare data to be survival-time data |
| `stsplit` | Split time-span records |
| `stsum` | Summarize survival-time data |
| `stteffects ipw (weibull)` | Treatment-effects estimation for survival data, inverse-probability weighting, Weibull distribution |
| `stteffects ipwra (weibull)` | Treatment-effects estimation for survival data, inverse-probability weighted regression adjustment, Weibull distribution |
| `stteffects ra (weibull)` | Treatment-effects estimation for survival data, regression adjustment, Weibull distribution |
| `stteffects wra (weibull)` | Treatment-effects estimation for survival data, weighted regression adjustment, Weibull distribution |
| `stvary` | Report variables that vary over time |
| `suest` | Seemingly unrelated estimation |
| `summarize` | Summary statistics |
| `sunflower` | Density-distribution sunflower plots |
| `sureg` | Zellner's seemingly unrelated regression |
| `svar` | Structural vector autoregression models |
| `svmat` | Convert variables to matrix and vice versa |
| `svy brr: logit` | Logistic regression using survey data—balanced repeated replications |
| `svy brr: poisson` | Poisson regression using survey data—balanced repeated replications |
| `svy brr: regress` | Linear regression using survey data—balanced repeated replications |
| `svy jackknife: logit` | Logistic regression using survey data—jackknife |
| `svy jackknife: poisson` | Poisson regression using survey data—jackknife |
| `svy jackknife: regress` | Linear regression using survey data—jackknife |
| `svy linearized: logit` | Logistic/logit regression using survey data—linearization |
| `svy linearized: poisson` | Poisson regression using count survey data—linearization |
| `svy linearized: regress` | Linear regression using survey data—linearization |

Table 2. Command descriptions

| Command | Description |
|---|---|
| `swilk` | Shapiro–Wilk test for normality |
| `symmetry` | Symmetry and marginal homogeneity tests |
| `table` (one-way) | Table of frequencies, summaries, and command results, one-way |
| `table` (two-way) | Table of frequencies, summaries, and command result, stwo-way |
| `tabodds` | Tabulate odds of failure by category |
| `tabodds` (adjusted) | Tabulate odds of failure by category adjusting for levels |
| `tabstat` | Display table of summary statistics |
| `tabstat, by()` | Display table of summary statistics over groups |
| `tabulate` (one-way) | Tables of frequencies, one-way |
| `tabulate` (two-way) | Tables of frequencies, two-way |
| `teffects aipw (linear)` | Treatment-effects estimation for linear regression, augmented inverse-probability weighting |
| `teffects aipw (probit)` | Treatment-effects estimation for probit regression, augmented inverse-probability weighting |
| `teffects ipw (logit)` | Treatment-effects estimation for linear regression, inverse-probability weighting |
| `teffects ipwra (linear)` | Treatment-effects estimation for linear regression, inverse-probability weight regression adjustment |
| `teffects ipwra (probit)` | Treatment-effects estimation for probit regression, augmented inverse-probability weighted regression adjustment |
| `teffects nnmatch` | Treatment-effects estimation, nearest-neighbor matching |
| `teffects psmatch, logit` | Treatment-effects estimation, propensity-score matching |
| `teffects ra (linear)` | Treatment-effects estimation for linear regression, regression adjustment |
| `teffects ra (probit)` | Treatment-effects estimation for probit regression, regression adjustment |
| `telasso (, linear) (, probit), ate` | Treatment-effects estimation using lasso, linear model, average treatment effect |
| `telasso (, linear) (, probit), atet` | Treatment-effects estimation using lasso, linear model, average treatment effect on the treated |
| `telasso (, linear) (, probit), pomeans` | Treatment-effects estimation using lasso, linear model, potential-outcome means |
| `telasso (, logit) (, probit), ate` | Treatment-effects estimation using lasso, logistic model, average treatment effect |
| `telasso (, logit) (, probit), atet` | Treatment-effects estimation using lasso, logistic model, average treatment effect on the treated |
| `telasso (, logit) (, probit), pomeans` | Treatment-effects estimation using lasso, logistic model, potential-outcome means |

Table 2. Command descriptions

| Command | Description |
|---|---|
| `telasso (, poisson) (, probit), ate` | Treatment-effects estimation using lasso, Poisson model, average treatment effect |
| `telasso (, poisson) (, probit), atet` | Treatment-effects estimation using lasso, Poisson model, average treatment effect on the treated |
| `telasso (, poisson) (, probit), pomeans` | Treatment-effects estimation using lasso, Poisson model, potential-outcome means |
| `telasso (, probit) (, probit), ate` | Treatment-effects estimation using lasso, probit model, average treatment effect |
| `telasso (, probit) (, probit), atet` | Treatment-effects estimation using lasso, probit model, average treatment effect on the treated |
| `telasso (, probit) (, probit), pomeans` | Treatment-effects estimation using lasso, probit model, potential-outcome means |
| `tetrachoric` | Tetrachoric correlations for binary variables |
| `threshold, threshvar()` | Threshold regression, single threshold for the intercept |
| `threshold, threshvar() regionvars()` | Threshold regression, single threshold for the intercept and some coefficients |
| `tnbreg` | Truncated negative binomial regression |
| `tobit` | Tobit regression |
| `tostring` | Convert numeric variables to string variables |
| `total` | Estimate totals |
| `tpoisson` | Truncated Poisson regression |
| `truncreg` | Truncated regression |
| `tsfilter bk` | Time-series filter, Baxter-King |
| `tsfilter bw` | Time-series filter, Butterworth |
| `tsfilter cf` | Time-series filter, Christiano-Fitzgerald |
| `tsfilter hp` | Time-series filter, Hodrick-Prescott |
| `tsrevar` | Create time-series operated temporary variables |
| `tsset` | Declare a dataset to be time-series data |
| `tssmooth exp` | Exponential smoothing of univariate time-series data |
| `tssmooth ma` | Moving average smoothing of univariate time-series data |
| `ttest1` | Mean comparison test against constant null hypothesis |
| `ttest2` | Mean comparison test against between variables |

Table 2. Command descriptions

| Command | Description |
|---|---|
| `ttest, by()` | Mean comparison test against over groups |
| `twoway fpfit` | Compute and graph fractional-polynomial fit |
| `twoway lfitci` | Compute and graph linear fit with confidence intervals |
| `twoway mband` | Compute and graph median bands |
| `twoway mspline` | Compute and graph spline smooth |
| `ucm, model(rwdrift)` | Unobserved-components model, random walk with drift |
| `var` | Vector autoregression models |
| `vargranger` | Perform pairwise Granger causality tests after `var` or `svar` |
| `varlmar` | Obtain LM statistics for residual autocorrelation after `var` or `svar` |
| `varnorm` | Test for normally distributed disturbances after `var` or `svar` |
| `varsoc` | Obtain lag-order selection statistics for VARs and VECMs |
| `varstable` | Check the stability condition of VAR or SVAR estimates |
| `vec` | Vector error-correction models |
| `veclmar` | Obtain LM statistics for residual autocorrelation after `vec` |
| `vecnorm` | Test for normally distributed disturbances after `vec` |
| `vecrank` | Estimate the cointegrating rank using Johansen's framework |
| `vecstable` | Check the stability condition of VECM estimates |
| `vwls` | Variance-weighted least squares |
| `wntestb` | Bartlett's periodogram-based test for white noise |
| `wntestq` | Portmanteau ($Q$) test for white noise |
| `xcorr` | Cross-correlogram for bivariate time series |
| `xpologit` | Cross-fit partialling-out lasso logistic regression |
| `xpopoisson` | Cross-fit partialling-out lasso Poisson regression |
| `xporegress` | Cross-fit partialling-out lasso linear regression |
| `xtabond` | Arellano–Bond linear, dynamic panel-data estimation |

Table 2. Command descriptions

| Command | Description |
|---|---|
| `xtabond, twostep` | Arellano–Bond linear, dynamic panel-data estimation, two-step estimation |
| `xtcloglog, re` | Random-effects cloglog models |
| `xtdata, be` | Compute between transform of panel data |
| `xtdata, fe` | Compute within (fixed-effects) transform of panel data |
| `xtdata, re` | Compute random-effects transform of panel data |
| `xtdidregress` | Difference-in-differences estimation |
| `xtdpd` | Linear dynamic panel-data estimation |
| `xtdpdsys` | Arellano–Bover/Blundell–Bond linear dynamic panel-data estimation |
| `xteregress` | Extended linear regression with random effects |
| `xtfrontier` | Stochastic frontier models for panel data |
| `xtgee, family(gaussian) corr(ar2)` | GEE estimation of Gaussian panel-data model with 2-period autocorrelation |
| `xtgee, fam(gauss) corr(unstruct)` | GEE estimation of Gaussian panel-data model with unstructured correlation |
| `xtcloglog, pa` | Population-averaged cloglog models |
| `xtlogit, pa` | Population-averaged logit models |
| `xtnbreg, pa` | Population-averaged negative binomial models |
| `xtpoisson, pa` | Population-averaged Poisson models |
| `xtprobit, pa` | Population-averaged probit models |
| `xtreg, pa` | Population-averaged linear models |
| `xtgls` | Fit panel-data models using GLS |
| `xthtaylor` | Hausman–Taylor estimator for error-components models |
| `xtile` | Panel-data line plots |
| `xtintreg` | Random-effects interval data regression models |
| `xtivreg, be` | Instrumental variables and two-stage least squares for panel-data models—between effects |
| `xtivreg, fd` | Instrumental variables and two-stage least squares for panel-data models—first differences |
| `xtivreg, fe` | Instrumental variables and two-stage least squares for panel-data models—fixed effects |

Table 2. Command descriptions

| Command | Description |
|---|---|
| `xtivreg, re` | Instrumental variables and two-stage least squares for panel-data models—random effects |
| `xtlogit, fe` | Fixed-effects logit models |
| `xtlogit, re` | Random-effects logit models |
| `xtmlogit, fe` | Fixed-effects multinomial logit models |
| `xtmlogit, re` | Random-effects multinomial logit models |
| `xtnbreg, fe` | Fixed-effects negative binomial models |
| `xtnbreg, re` | Random-effects negative binomial models |
| `xtologit` | Random-effects ordered logistic models |
| `xtoprobit` | Random-effects ordered probit models |
| `xtpcse` | OLS or Prais–Winsten models with panel-corrected standard errors |
| `xtpoisson, fe` | Fixed-effects Poisson models |
| `xtpoisson, re` | Random-effects Poisson models |
| `xtprobit, re` | Random-effects probit models |
| `xtrc` | Random-coefficients regression |
| `xtreg, be` | Between-effects linear models |
| `xtreg, fe` | Fixed-effects linear models |
| `xtreg, fe vce(robust)` | Fixed-effects linear models, cluster–robust standard errors |
| `xtreg, mle` | Random-effects linear models, ML estimation |
| `xtreg, re` | Random-effects linear models |
| `xtregar, fe` | Fixed-effects linear models with an AR(1) disturbance |
| `xtregar, re` | Random-effects linear models with an AR(1) disturbance |
| `xtset` | Declare data to be panel data |
| `xtstreg, distribution(exponential)` | Random-effects survival models, exponential distribution |
| `xtstreg, distribution(weibull)` | Random-effects survival models, Weibull distribution |
| `xtsum` | Summarize panel data |

Table 2. Command descriptions

| Command | Description |
|---|---|
| `xttab` | Tabulate panel data |
| `xttobit` | Random-effects tobit models |
| `xtunitroot breitung` | Panel-data unit-root test—Breitung |
| `xtunitroot fisher` | Panel-data unit-root test—Fisher |
| `xtunitroot hadri` | Panel-data unit-root test—Hadri Lagrange multiplier |
| `xtunitroot ht` | Panel-data unit-root test—Harris–Tzavalis |
| `xtunitroot ips` | Panel-data unit-root test—Im–Pesaran–Shin |
| `xtunitroot llc` | Panel-data unit-root test—Levin–Lin–Chu |
| `zinb` | Zero-inflated negative binomial regression |
| `ziologit` | Zero-inflated ordered logit regression |
| `zioprobit` | Zero-inflated ordered probit regression |
| `zip` | Zero-inflated Poisson regression |
| `_predict, xb` | Obtain predictions, residuals, etc., after estimation programming command—option `xb` |
| `_rmcoll` | Remove collinear variables |
| `_robust` | Robust variance estimates |

## D Problem sizes

The following table (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 $n_{\text{iter}}$ records the number of times the command was run on the problem to generate a single timing.

Table 3. Problem sizes

| Command | Observations $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
|---|---|---|---|---|---|---|
| `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 |
| `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 |

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

Table 3. Problem sizes

| Command | Observations | | | | | |
|---|---|---|---|---|---|---|
| | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `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 |
| `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 |
| `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 |

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

Table 3. Problem sizes

| | Observations | | | | | |
|---|---|---|---|---|---|---|
| Command | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `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 |
| `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 $n_{\text{iter}}$, number of iterations.

Table 3. Problem sizes

| Command | Observations $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
|---|---|---|---|---|---|---|
| `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 |
| `destring` | | 4000 | 2000 | | | 1 |
| `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` *exp* | 10000000 | | | 4 | | 1 |
| `drop in` *range* | 10000000 | | | 4 | | 1 |
| `dsge` | 10000 | | | 4 | | 1 |
| `dsgenl` | 10000 | | | 4 | | 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 $n_{\text{iter}}$, number of iterations.

Table 3. Problem sizes

| | Observations | | | | | |
|---|---|---|---|---|---|---|
| Command | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `dslogit` | 100000 | | | 40 | | 1 |
| `dspoisson` | 100000 | | | 40 | | 1 |
| `dsregress` | 100000 | | | 40 | | 1 |
| `dstdize` | | 10 | 150 | 200 | | 1 |
| `dvech` | 500 | | | 2 | | 1 |
| `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` *varname* | 100000 | | | 100 | 5 | 1 |
| `expandcl` # | | 12000 | 10 | 100 | | 1 |
| `expandcl` *varname* | | 30000 | 10 | 80 | 5 | 1 |
| `expoisson` | 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 $n_{\text{iter}}$, number of iterations.

Table 3. Problem sizes

| | Observations | | | | | |
|---|---|---|---|---|---|---|
| Command | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `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 |
| `glogit` | 2000000 | | | 100 | 50 | 1 |
| `gmm` | 1000 | | | 10 | | 1 |
| `gmm` (with derivatives) | 100000 | | | 10 | | 1 |
| `gprobit` | 3000000 | | | 100 | 50 | 1 |
| `graph bar` | 500000 | | | 10 | 3 | 1 |
| `graph box` | 200000 | | | 2 | 10 | 1 |
| `graph pie` | 2500000 | | | 10 | 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 $n_{\text{iter}}$, number of iterations.

Table 3. Problem sizes

| | Observations | | | | | |
|---|---|---|---|---|---|---|
| Command | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `grmeanby` | 300000 | | | 4 | 10 | 1 |
| `gsem, oprobit (CFA, 2-level)` | | 1000 | 10 | 4 | 1 | 1 |
| `gsem, oprobit (CFA)` | 5000 | | | 4 | 1 | 1 |
| `gsem, logit group()` | | 5 | 50000 | 40 | | 1 |
| `gsem, group()` | | 5 | 50000 | 40 | | 1 |
| `gsem, ologit group()` | | 5 | 50000 | 40 | | 1 |
| `gsem, poisson group()` | | 5 | 50000 | 40 | | 1 |
| `gsort` | 1000000 | | | 5 | | 1 |
| `hausman` | 200 | | | | | 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 |

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

Table 3. Problem sizes

| | Observations | | | | | |
|---|---|---|---|---|---|---|
| Command | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `ir` | 10000000 | | | | | 1 |
| `by: ir` | 10000 | | | | 200 | 1 |
| `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 |
| `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 |
| `ivregress 2sls` | 800000 | | | 50 | 20 | 1 |
| `ivregress gmm` | 1500000 | | | 20 | 20 | 1 |
| `ivregress liml` | 2000000 | | | 20 | 20 | 1 |
| `ivtobit` | 150000 | | | 50 | 20 | 1 |
| `kap` | 500000 | | | 2 | 10 | 4 |
| `kappa` | 2000000 | | | 10 | 20 | 1 |
| `kdensity` | 10000000 | | | | | 1 |
| `keep if` *exp* | 10000 | | | 4000 | | 1 |
| `keep in` *range* | 20000 | | | 4000 | | 1 |
| `keep` *varlist* | 50000 | | | 4000 | | 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 $n_{\text{iter}}$, number of iterations.

Table 3. Problem sizes

| | Observations | | | | | |
|---|---|---|---|---|---|---|
| Command | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `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` | 20000000 | | | | 20 | 1 |
| `loadingplot` | 2000000 | | | 60 | | 1 |
| `logistic` | 300000 | | | 200 | | 1 |
| `logit` | 300000 | | | 200 | | 1 |
| `loneway` | 2000000 | | | 500 | | 1 |
| `lowess` | 90000 | | | 1 | | 1 |
| `lpoly` | 1000000 | | | | | 1 |
| `ltable` | 50000 | | | 1 | | 40 |
| `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 |

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

Table 3. Problem sizes

| | Observations | | | | | |
|---|---|---|---|---|---|---|
| Command | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `marksample` | 1200000 | | | 200 | | 1 |
| `marksample if` *exp* | 2300000 | | | 100 | | 1 |
| `matrix accum` | 3000000 | | | 200 | | 1 |
| `matrix eigenvalues` | 500 | | | 500 | | 1 |
| `matrix score` | 6000000 | | | 1000 | | 1 |
| `matrix svd` | 300 | | | 300 | | 1 |
| `matrix symeigen` | 600 | | | 600 | | 1 |
| `matrix syminv` | 2000 | | | 2000 | | 1 |
| `mca` | 1000000 | | | 3 | 5 | 1 |
| `mcc` | 10000000 | | | | | 1 |
| `mds` | 800 | | | 400 | | 1 |
| `mdslong` | | 600 | 1 | | | 1 |
| `mean` | 1000000 | | | 200 | | 1 |
| `mecloglog` | | 2000 | 10 | 2 | 1 | 1 |
| `median` | 8000000 | | | 5 | | 1 |
| `meintreg` | | 1000 | 50 | 5 | 1 | 1 |
| `melogit` | | 4000 | 10 | 10 | 1 | 1 |
| `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 |

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

Table 3. Problem sizes

| Command | Observations $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
|---|---|---|---|---|---|---|
| `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 |
| `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 |

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

Table 3. Problem sizes

| Command | Observations | | | | | |
|---|---|---|---|---|---|---|
| | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `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 |
| `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 |
| `mvdecode` | 500000 | | | 20 | 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 $n_{\text{iter}}$, number of iterations.

Table 3. Problem sizes

| | Observations | | | | | |
|---|---|---|---|---|---|---|
| Command | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `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` | 1000000 | | | 10 | 1000 | 1 |
| `nptrend_carmitage` | 1000000 | | | 10 | | 1 |
| `nptrend_jterpstra` | 1000000 | | | 10 | 1000 | 1 |
| `nptrend_linear` | 1000000 | | | 10 | 1000 | 1 |
| `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 |

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

Table 3. Problem sizes

| | Observations | | | | | |
|---|---|---|---|---|---|---|
| Command | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `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 |
| `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 |

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

Table 3. Problem sizes

| Command | Observations $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
|---|---|---|---|---|---|---|
| `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 |
| `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 $n_{\text{iter}}$, number of iterations.

Table 3. Problem sizes

| | Observations | | | | | |
|---|---|---|---|---|---|---|
| Command | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `replace` (small expressions) | 150000 | | | 4000 | | 1 |
| `reshape long` | | 50000 | 20 | | | 1 |
| `reshape wide` | | 50000 | 15 | 5 | | 1 |
| `robvar` | 200000 | | | 2 | | 1 |
| `rocfit` | 100000 | | | 1 | 5 | 1 |
| `roctab` | 600000 | | | 1 | 20 | 1 |
| `rotate` | 10000 | | | 80 | | 1 |
| `rotatemat` | 80 | | | 80 | | 1 |
| `rreg` | 100000 | | | 200 | | 1 |
| `runtest` | 6000000 | | | 1 | | 1 |
| `scobit` | 120000 | | | 200 | | 1 |
| `scoreplot` | 400000 | | | 20 | | 1 |
| `screeplot` | 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 |

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

Table 3. Problem sizes

| | Observations | | | | | |
|---|---|---|---|---|---|---|
| Command | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `sktest` | 6000000 | | | 2 | | 1 |
| `slogit` | 20000 | | | 10 | 5 | 1 |
| `sort` | 9000000 | | | 10 | | 1 |
| `spearman` | 400000 | | | 3 | | 1 |
| `sspace` | 5000 | | | 20 | | 1 |
| `stack` | 500000 | | | 100 | | 1 |
| `stci` | 200000 | | | 1 | | 1 |
| `stcox` | 250000 | | | 10 | | 1 |
| `stcrreg` | 2000 | | | 5 | | 1 |
| `stgen` | 30000000 | | | 2 | | 1 |
| `stintcox` | 1500 | | | 2 | | 1 |
| `stintreg, d(exponential)` | 500000 | | | 30 | | 1 |
| `stintreg, d(weibull)` | 200000 | | | 20 | | 1 |
| `stir` | 4500000 | | | 1 | 2 | 1 |
| `stmc` | 900000 | | | | | 1 |
| `by: stmc` | 600000 | | | | 50 | 1 |
| `stmh` | 1500000 | | | | | 1 |
| `by: stmh` | 1500000 | | | | 10 | 1 |
| `stptime` | 9000000 | | | 1 | 60000 | 1 |
| `strate` | 1000000 | | | 1 | 5 | 1 |
| `streg, distribution(exponential)` | 600000 | | | 100 | | 1 |
| `streg, dist(exp) vce(cluster)` | 200000 | | | 200 | 1000 | 1 |
| `streg, dist(exp) frailty()` | 60000 | | | 200 | | 1 |
| `streg, dist(exp) frailty() shared()` | 200000 | | | 100 | 1000 | 1 |
| `streg, dist(exp) 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 $n_{\text{iter}}$, number of iterations.

Table 3. Problem sizes

| Command | Observations |  |  |  |  |  |
|---|---|---|---|---|---|---|
|  | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `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 |
| `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 |

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

Table 3. Problem sizes

| | Observations | | | | | |
|---|---|---|---|---|---|---|
| Command | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `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 |
| `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 |

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

Table 3. Problem sizes

| | Observations | | | | | |
|---|---|---|---|---|---|---|
| Command | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `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 |
| `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 |
| `telasso (, logit) (, probit), pomeans` | 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 $n_{\text{iter}}$, number of iterations.

Table 3. Problem sizes

| | Observations | | | | | |
|---|---|---|---|---|---|---|
| Command | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `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 |
| `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 |

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

Table 3. Problem sizes

| | Observations | | | | | |
|---|---|---|---|---|---|---|
| Command | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `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 |
| `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 |
| `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 |

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

Table 3. Problem sizes

| | Observations | | | | | |
|---|---|---|---|---|---|---|
| Command | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `xtabond, twostep` | | 100000 | 10 | 2 | | 1 |
| `xtcloglog, re` | | 20000 | 5 | 5 | | 1 |
| `xtdata, be` | | 15000 | 5 | 200 | | 1 |
| `xtdata, fe` | | 500000 | 5 | 5 | | 1 |
| `xtdata, re` | | 300000 | 5 | 5 | | 1 |
| `xtdidregress` | | 3000 | 50 | 25 | | 1 |
| `xtdpd` | | 40000 | 5 | 5 | | 1 |
| `xtdpdsys` | | 60000 | 5 | 5 | | 1 |
| `xteregress` | | 1000 | 5 | 1 | 1 | 1 |
| `xtfrontier` | | 4000 | 10 | 50 | | 1 |
| `xtgee, family(gaussian) corr(ar2)` | | 50000 | 5 | 10 | | 1 |
| `xtgee, fam(gauss) corr(unstruct)` | | 60000 | 5 | 10 | | 1 |
| `xtcloglog, pa` | | 100000 | 5 | 5 | | 1 |
| `xtlogit, pa` | | 100000 | 5 | 5 | | 1 |
| `xtnbreg, pa` | | 80000 | 5 | 5 | | 1 |
| `xtpoisson, pa` | | 30000 | 10 | 5 | | 1 |
| `xtprobit, pa` | | 60000 | 10 | 5 | | 1 |
| `xtreg, pa` | | 100000 | 5 | 10 | | 1 |
| `xtgls` | | 5 | 200000 | 5 | | 1 |
| `xthtaylor` | | 100000 | 10 | 4 | 4 | 1 |
| `xtile` | 100000 | | | | | 1 |
| `xtintreg` | | 15000 | 5 | 5 | | 1 |
| `xtivreg, be` | | 120000 | 5 | 5 | 5 | 1 |
| `xtivreg, fd` | | 80000 | 5 | 5 | 5 | 1 |
| `xtivreg, fe` | | 80000 | 5 | 5 | 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 $n_{\text{iter}}$, number of iterations.

Table 3. Problem sizes

| Command | Observations |  |  |  |  |  |
|---|---|---|---|---|---|---|
|  | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `xtivreg, re` |  | 150000 | 5 | 5 | 5 | 1 |
| `xtlogit, fe` |  | 20000 | 10 | 50 |  | 1 |
| `xtlogit, re` |  | 40000 | 5 | 5 |  | 1 |
| `xtmlogit, fe` |  | 10000 | 5 | 10 | 3 | 1 |
| `xtmlogit, re` |  | 5000 | 10 | 10 | 3 | 1 |
| `xtnbreg, fe` |  | 70000 | 5 | 10 |  | 1 |
| `xtnbreg, re` |  | 40000 | 5 | 10 |  | 1 |
| `xtologit` |  | 8000 | 10 | 10 | 0 | 1 |
| `xtoprobit` |  | 8000 | 10 | 10 | 0 | 1 |
| `xtpcse` |  | 3 | 80000 | 50 |  | 1 |
| `xtpoisson, fe` |  | 20000 | 5 | 50 |  | 1 |
| `xtpoisson, re` |  | 30000 | 5 | 50 |  | 1 |
| `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 |

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

Table 3. Problem sizes

| Command | Observations | | | | | |
|---|---|---|---|---|---|---|
| | $N$ | $m$ | $t$ | $k$ | $d_1$ | $n_{\text{iter}}$ |
| `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 |
| `xtunitroot ips` | | 1000 | 20 | | | 1 |
| `xtunitroot llc` | | 100 | 500 | | | 1 |
| `zinb` | 150000 | | | 50 | 50 | 1 |
| `ziologit` | 200000 | | | 30 | 30 | 1 |
| `zioprobit` | 200000 | | | 30 | 30 | 1 |
| `zip` | 250000 | | | 50 | 50 | 1 |
| `_predict, xb` | 5000000 | | | 1000 | | 1 |
| `_rmcoll` | 6000000 | | | 100 | | 1 |
| `_robust` | 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 $n_{\text{iter}}$, 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 parallization 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`, `fdadescribe`, `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 `symplot`.

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()`, `invweibulltail()`, `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()`, `weibulltail()`, `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

| | Speed relative to a single core[a] | | | | |
|---|---|---|---|---|---|
| | Number of cores | | | | Percentage |
| Command | 2 | 4 | 8 | 16 | parallelized[b] |
| 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 regresion—same as `xtlogit, re` |
| RE regression | Continuous (Gaussian distribution) model with random intercepts—same as `xtreg, re` |
| 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 regresion, 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.

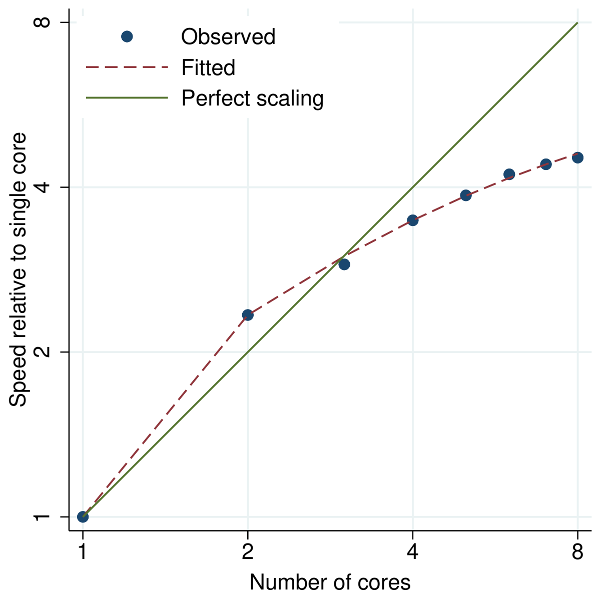


Figure 704. Finite mixture model performance plot.

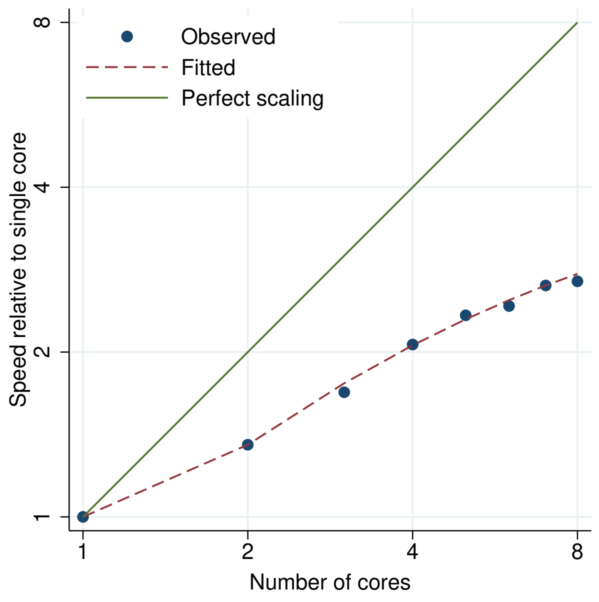


Figure 705. Item response model performance plot.

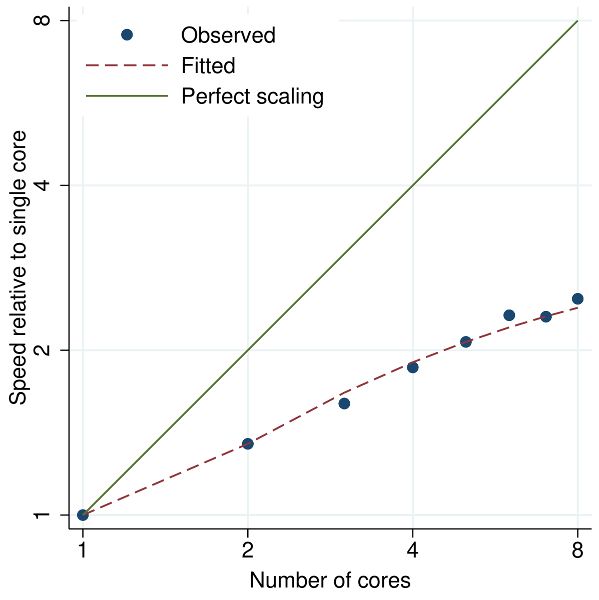


Figure 706. Latent class model performance plot.

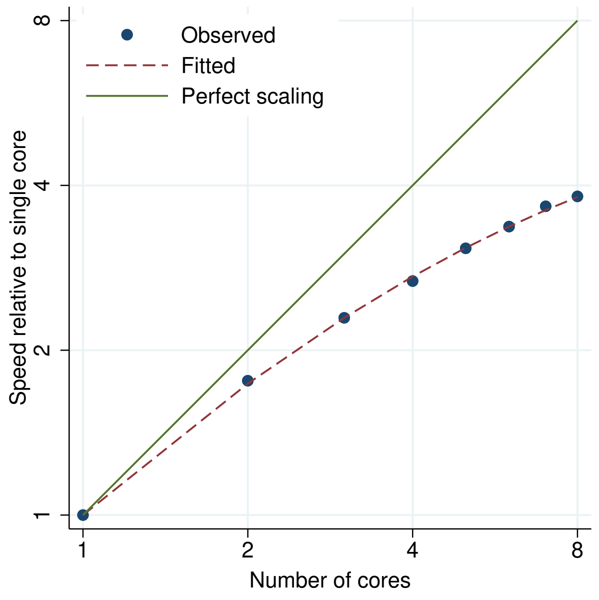


Figure 707. Measurement error model performance plot.

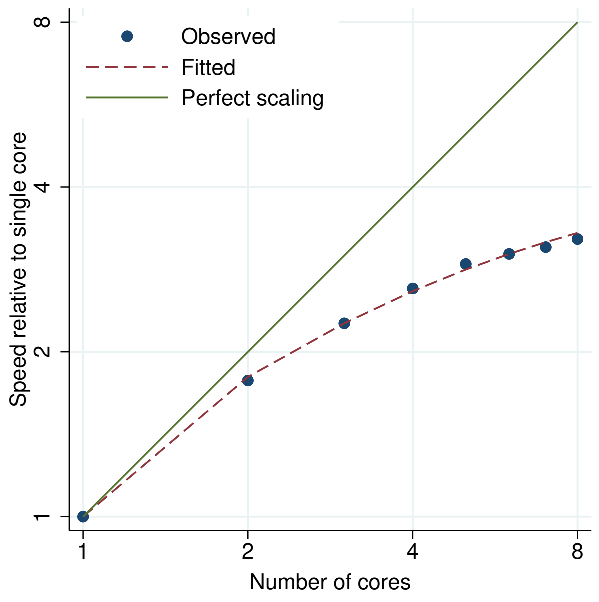


Figure 708. Rank-outcome latent class performance plot.

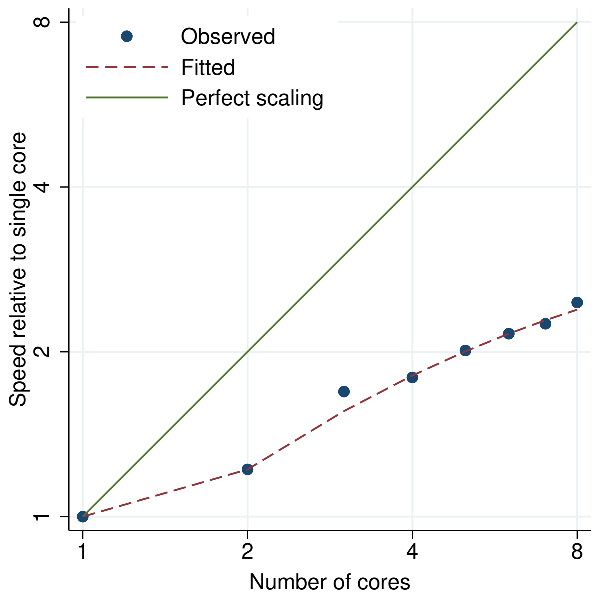


Figure 709. MIMIC model performance plot.

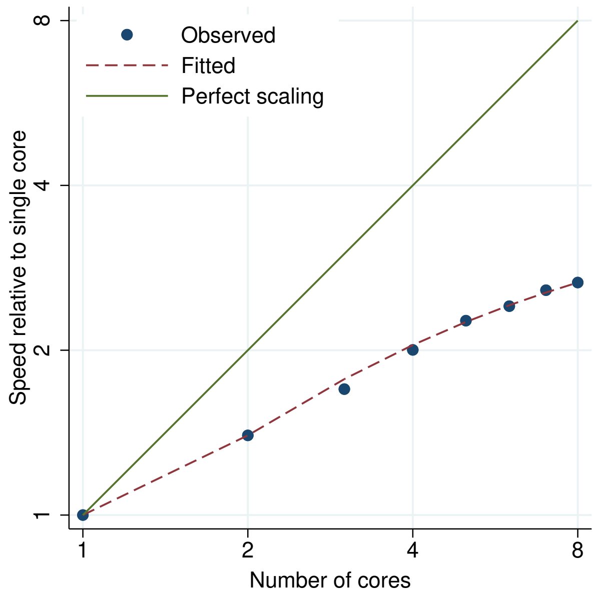


Figure 710. Random-effects logistic performance plot.

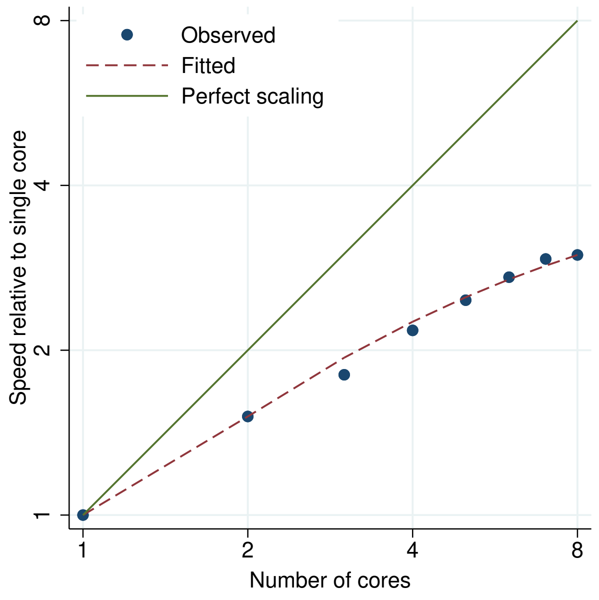


Figure 711. RE regression performance plot.

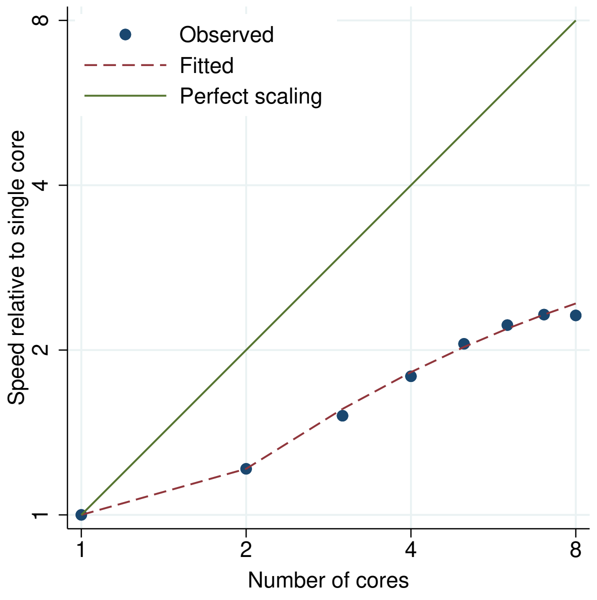


Figure 712. Two-level RE logistic performance plot.

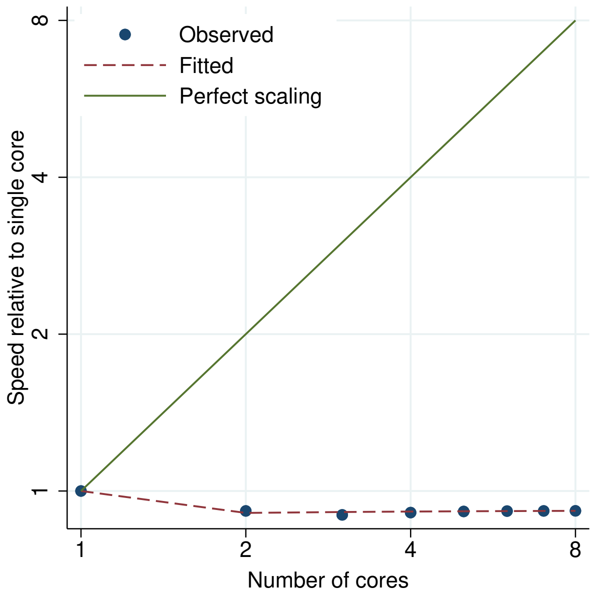


Figure 713. Random-coefficients Poisson performance plot.

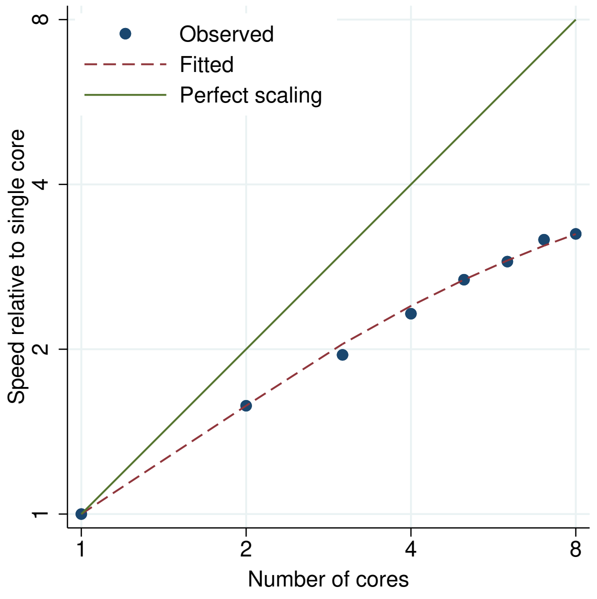


Figure 714. RE logistic with constant performance plot.

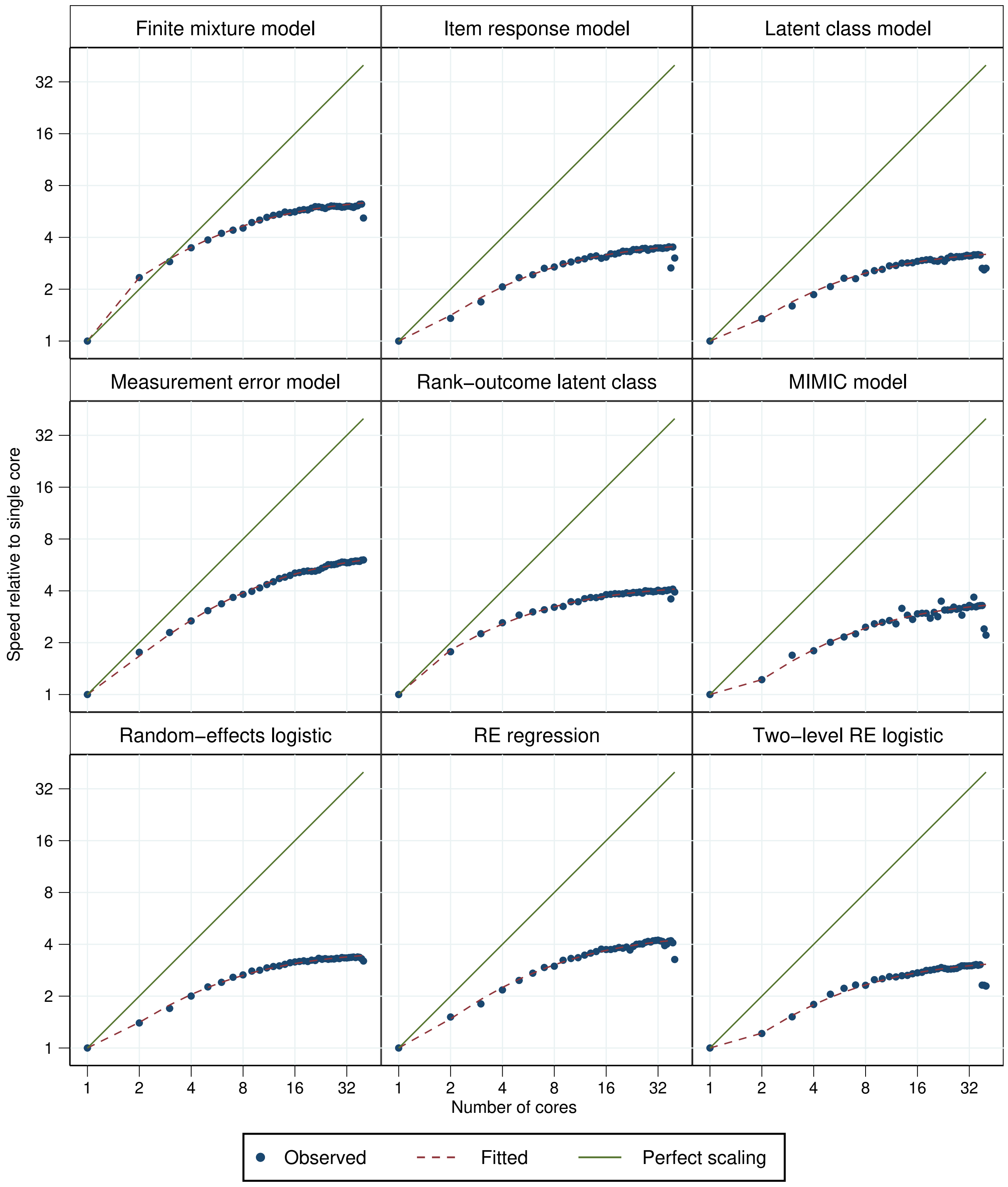


Figure 715. **Parallelization performance plots.**

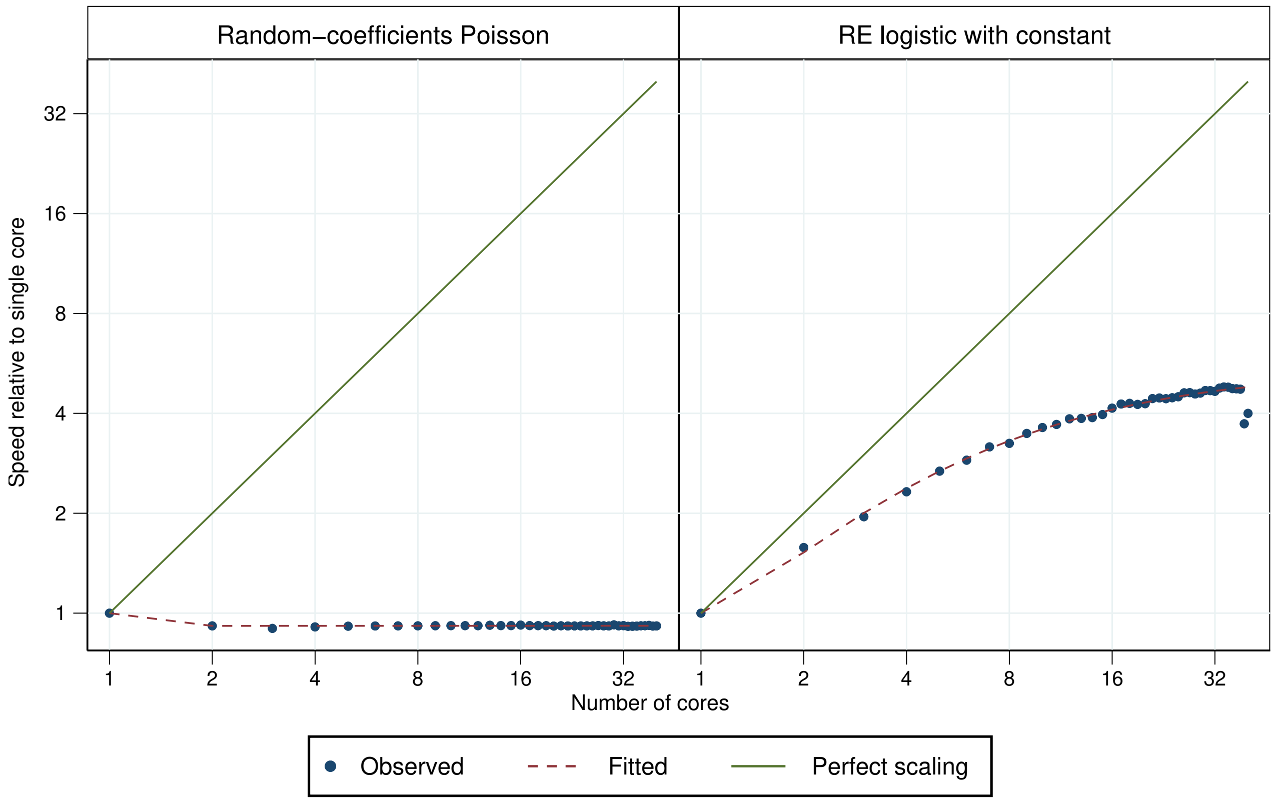


Figure 716. **Parallelization performance plots.**

## 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.