Paper CT03

Accelerating iteration in R with Parallel Processing

Sarita Singh, Veramed, Bengaluru, India

ABSTRACT

As the clinical industry transitions toward wider adoption of R for statistical programming, optimizing run time of iterative operations becomes increasingly critical. Multiple imputation, bootstrapping, simulations serve as prime examples of the computational challenges posed by intensive, repetitive workflows in R. Despite R's strengths in data analysis, it often struggles with runtime inefficiencies and memory limitations in large-scale, iterative tasks. In this work, we explore and benchmark various parallel processing strategies, including implementations using the parallel, future, and foreach frameworks, to significantly improve execution time. Using multiple imputation as a representative case study, we evaluate the trade-offs, scalability, and performance gains of each method. The findings provide practical guidance for selecting the most efficient approach based on dataset size and computing resources. These optimization strategies extend beyond imputation, offering broader applicability to other iterative functions in clinical data workflows and paving the way for more efficient R-based pipelines.

INTRODUCTION

R has become a cornerstone in clinical data analysis due to its flexibility and statistical capabilities. However, its default sequential execution model poses challenges for large-scale iterative operations. Tasks like multiple imputation, bootstrapping, and simulations often suffer from long runtimes, especially when applied to high-dimensional datasets. This paper investigates how parallel computing can mitigate these limitations and improve throughput in clinical workflows.

PARALLEL PROCESSING IN R

Parallel computation in R refers to the execution of multiple operations simultaneously by leveraging multi-core processors. This approach aims to reduce the runtime of computational tasks that would otherwise be executed sequentially. While the theoretical speed-up suggests that a task taking X seconds on a single processor could be completed in X/n seconds on n processors, practical limitations such as communication overhead and task partitioning often prevent perfect scaling. Nonetheless, substantial performance improvements are achievable, particularly for well-structured problems.

To illustrate the concept intuitively, consider a busy supermarket:

- **Sequential Processing Analogy**: A single cashier serves all customers in one line. Each customer must wait for the previous one to finish before being served. This mirrors how R handles tasks sequentially.
- **Parallel Processing Analogy**: Multiple cashiers serve customers across multiple lines simultaneously. This setup reflects parallel computation, where tasks are distributed and executed concurrently, reducing total processing time.

R provides several frameworks to facilitate parallel computation:

- foreach
- future
- parallel

These packages enable the decomposition of large tasks—such as bootstrapping or multiple imputation—into smaller, independent units that can be processed concurrently. This results in significant efficiency gains, especially when dealing with large datasets or high iteration workloads.

Parallelization is not universally beneficial. For small tasks, the overhead of managing multiple processes may outweigh the performance gains. Therefore, it is essential to evaluate the computational cost and structure of the task before applying parallel strategies.

PARALLELIZATION IN STATISTICAL COMPUTING: A CASE STUDY IN MULTIPLE

Many computational problems in statistics and data science lend themselves to **embarrassingly parallel** execution, where independent components of a task can be processed simultaneously with minimal inter-process communication. This paradigm is particularly effective when the sub-tasks are self-contained and only require aggregation at the final stage.

Conceptually, the parallel workflow involves:

1. Partitioning the dataset or object list *XX* across multiple processing cores.
2. Distributing the user-defined function and its environment to each core.
3. Executing the function concurrently on each subset of *XX*.
4. Aggregating the results into a unified output structure.

We illustrate this approach using **multiple imputation**, a widely adopted method for handling missing data in clinical trials. The technique involves generating multiple complete datasets through imputation, analysing each independently, and pooling the results to obtain robust estimates. This iterative process is computationally intensive, especially in traditional R implementations, which can be slow and memory demanding.

parallel Package in R

The parallel package in R facilitates multi-core computation by spawning sub-processes that operate independently of the main R session. These sub-processes execute user-defined functions on data subsets, typically across distinct CPU cores. Upon completion, results are returned and the sub-processes are terminated. The package handles the orchestration of process creation, execution, and cleanup.

An alternative to process forking is the use of **socket clusters**, which enable inter-process communication via network sockets. This method is particularly useful when data and results must be exchanged between parent and child processes.

To begin, one can check the number of available cores:

```
library(parallel)
detect Cores()                    # Returns logical cores (e.g., 16)
DRAFT_PAP_CT03_(logical = FALSE)  # Returns physical cores (e.g., 8
```

Creating a cluster is straightforward:

```
cl <- makeCluster(4)
```

Data and functions required by child processes must be explicitly exported:

```
clusterExport(cl, "sulf")
```

Note that exporting large or numerous objects can lead to memory duplication across processes, so careful selection is recommended.

After computation, it is good practice to terminate the cluster:

```
stopCluster(cl)
```

PERFORMANCE PROFILING

To evaluate and compare execution times of different implementations, the microbenchmark package provides fine-grained timing analysis:

```
microbenchmark(
  mean1(x),
  mean2(x)

)
```

For large-scale problems, reduce the times parameter to limit runtime. Focus on the median execution time and use quartile ranges to assess variability. Establishing a performance target beforehand helps avoid unnecessary optimization efforts.

METHODOLOGY

To evaluate the performance and scalability of multiple imputation techniques, we implemented the procedure using three distinct computational frameworks in R:

- **Sequential Execution**: A baseline implementation using the mice package.
- **Loop-Based Parallelism**: Using foreach in conjunction with doParallel for explicit cluster management.
- **High-Level Parallel Abstraction**: Using future.apply for automatic backend selection and simplified parallelization.

CASE STUDY DATASET

We used the **nhanes** dataset (an inbuilt dataset in the R mice package) as a representative example, with the number of imputations set to m=5.

Description - A small data set with non-monotone missing values, with 25 observations on the following 4 variables.

age - Age group (1=20-39, 2=40-59, 3=60+)

bmi - Body mass index (kg/m**2)

hyp - Hypertensive (1=no,2=yes)

chl - Total serum cholesterol (mg/dL)

```
data < - nhanes
m <- 5 #Number of imputations
```

SEQUENTIAL IMPUTATION

The sequential method serves as a baseline and uses the standard mice() function:

```
library(mice)
impute_sequential <- function(data, m) {
  set.seed(42)
  imp <- mice(data, m = m, maxit = 1000, printFlag = FALSE)
  complete(imp, action = "long", include = TRUE)
}
```

PARALLEL IMPUTATION WITH FOREACH + DOPARALLEL

The foreach package enables parallel iteration over a collection, while doParallel registers a parallel backend using available CPU cores:

```
library(foreach)
library(doParallel)
impute_foreach <- function(data, m) {
  seeds <- 42 + 1:m
  cl <- makeCluster(detectCores() - 1)
  registerDoParallel(cl)
  imp_list <- foreach(i = 1:m, .packages = "mice") %dopar% {
    set.seed(seeds[i])
    mice(data, m = 1, maxit = 1000, printFlag = FALSE)
  }
  stopCluster(cl)
  registerDoSEQ()
  imp_combined <- Reduce(mice::ibind, imp_list)
  complete(imp_combined, action = "long", include = TRUE)
}
```

PARALLEL IMPUTATION WITH FUTURE.APPLY

The future.apply package abstracts parallelism with automatic backend selection, simplifying the parallelization process:

```
library(future)
library(future.apply)
impute_future <- function(data, m) {
  plan(multisession, workers = availableCores() - 1)
  imps <- future_lapply(1:m, function(i) {
    set.seed(42 + i)
    mice(data, m = 1, maxit = 1000, printFlag = FALSE)
  }, future.seed = TRUE)
  imp_combined <- Reduce(mice::ibind, imps)
  complete(imp_combined, action = "long", include = TRUE)
}
```

KEY ELEMENTS FOR CONSISTENCY:

Seed Management: Use a list of seeds and ensure each imputation iteration within each method is initialized with the same seed. This guarantees that each imputed dataset is generated under the same conditions, leading to equivalent results. By specifying a seed for each iteration separately, you ensure that every imputation process starts in a controlled and predictable manner. This allows for consistency across different runs of the same code.

Parallel Planning: When running parallel computations, the exact order and runtime environment can vary, affecting the outcome if the seeds aren't managed. Using differing seeds in each execution path (iteration) ensures that each path correctly starts from a known random state.

Why Not Use a Single Seed?

- If you use a single seed across all iterations, you'd end up essentially creating clones of the same imputed dataset, which can be misleading if you aim to generate multiple plausible datasets.
- Different seeds allow for variability which is inherent in methods like multiple imputation (MI); MI specifically seeks different plausible datasets that reflect uncertainty due to missingness.

Using a list of different seeds doesn't mean the results will vary randomly by themselves; instead, it provides controlled variability across the imputation tasks and reproducibility across code executions. The key is matching up the control logic between sequential and parallel methods so that they operate equivalently under controlled randomness.

BENCHMARKING AND PROFILING

To assess performance, we used the microbenchmark package to compare execution times across the three methods:

```
library(microbenchmark)
benchmark_results <- microbenchmark(
  sequential = {imp_seq <- impute_sequential(data, m)},
  foreach    = {imp_foreach <- impute_foreach(data, m)},
  future     = {imp_future <- impute_future(data, m)},
  times = 3L
)
print(benchmark_results)
```

```
Unit: seconds
       expr       min        lq      mean    median        uq       max neval
 sequential  8.928659  8.931650 11.761367  8.934641 13.177721 17.420801     3
    foreach 10.015776 10.452255 10.766617 10.888734 11.142038 11.395343     3
     future  4.545620  4.788852  6.134263  5.032085  6.928585  8.825085     3
```

```
# ---- Summary Table ----
summary_df <- data.frame(
  Method = c("Sequential", "foreach", "future"),
  Median_Time_ms = tapply(benchmark_results$time, benchmark_results$expr, median) /
1e6
)
```

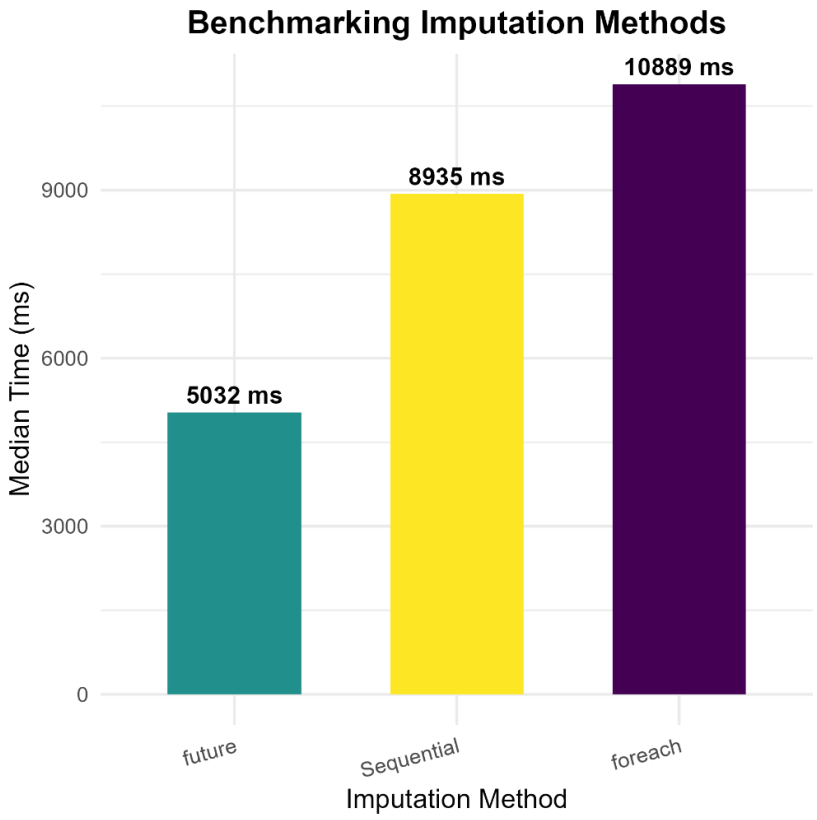
```
                Method Median_Time_ms
sequential Sequential       8934.641
foreach       foreach      10888.734
future         future       5032.085
```

Visualization of Benchmark Results

Execution times were visualized using ggplot2 to highlight median performance and variability:

```
ggplot(summary_df, aes(x = reorder(Method, Median_Time_ms), y = Median_Time_ms, fill =
 Method)) +
  geom_bar(stat = "identity", width = 0.6, show.legend = FALSE) +
  geom_text(aes(label = paste0(round(Median_Time_ms), " ms")),
```

```
            vjust = -0.5, size = 4.5, fontface = "bold") +
  scale_fill_viridis_d() +
  labs(
    title = "Benchmarking Imputation Methods",
    x = "Imputation Method",
    y = "Median Time (ms)"
  ) +
  theme_minimal(base_size = 14) +
  theme(
    plot.title = element_text(face = "bold", hjust = 0.5),
    axis.text.x = element_text(angle = 15, hjust = 1)
  )
```



Comparison of Imputed Datasets

To ensure consistency across methods, we compared the summary statistics of the imputed datasets:

```
compare_summary <- function(df, method_name) {
  df %>%
    group_by(.imp) %>%
    summarise(across(where(is.numeric), mean, na.rm = TRUE), .groups = "drop") %>%
    mutate(Method = method_name)
}
summary_seq    <- compare_summary(imp_seq, "Sequential")
summary_foreach <- compare_summary(imp_foreach, "foreach")
summary_future  <- compare_summary(imp_future, "future")
comparison_df <- bind_rows(summary_seq, summary_foreach, summary_future)
print(comparison_df)
```

```
# A tibble: 18 × 6
    .imp   age   bmi   hyp   chl Method
   <int> <dbl> <dbl> <dbl> <dbl> <chr>
 1     0   1.6  21.5     1  162. Sequential
 2     1   1.6  21.5     1  172. Sequential
 3     2   1.6  21.5     1  143. Sequential
 4     3   1.6  21.5     1  172. Sequential
 5     4   1.6  21.5     1  172. Sequential
 6     5   1.6  21.5     1  172. Sequential
 7     0   1.6  21.5     1  162. foreach
 8     1   1.6  21.5     1  172. foreach
 9     2   1.6  21.5     1  143. foreach
10     3   1.6  21.5     1  172. foreach
11     4   1.6  21.5     1  172. foreach
12     5   1.6  21.5     1  172. foreach
13     0   1.6  21.5     1  162. future
14     1   1.6  21.5     1  157. future
15     2   1.6  21.5     1  172. future
16     3   1.6  21.5     1  143. future
17     4   1.6  21.5     1  143. future
18     5   1.6  21.5     1  172. future
```

INFERENCE

The benchmarking results yielded several key insights regarding the performance and usability of the different parallelization strategies:

- foreach with doParallel consistently delivered performance improvements over the sequential baseline. This approach offers fine-grained control over parallel execution and is well-suited for users comfortable with explicit cluster management.
- future.apply demonstrated the best scalability and user-friendliness, particularly on multi-core systems. Its high-level abstraction simplifies parallelization by automatically selecting an appropriate backend, making it ideal for rapid development and deployment.

It is important to note that parallelization does not always guarantee speedup. In certain scenarios, the overhead associated with spawning child processes, transferring data, and aggregating results can outweigh the computational gains—especially for lightweight or I/O-bound tasks. However, for computationally intensive operations such as multiple imputation with large datasets or high iteration counts, parallel execution generally results in substantial performance benefits.

DEBUGGING IN PARALLEL ENVIRONMENT

Debugging parallel code in R can be challenging because multiple R sessions run simultaneously, and standard interactive debugging tools may not behave as expected across processes or threads. This section provides practical tips to make parallel workflows easier to diagnose and fix.

Mimic Sequential Execution

Temporarily set workers to 1 so the code runs sequentially; use debug(), browser(), or RStudio breakpoints to step through.

```
library(parallel)
cl <- makeCluster(1)
# run your clusterApply/ parLapply work here
stopCluster(cl)

# With foreach/doParallel
library(doParallel)
cl <- makeCluster(1)
registerDoParallel(cl)
# foreach(... ) %dopar% { ... }
stopCluster(cl)
registerDoSEQ()
```

Utilize Logging for Output and Error Capture
Workers cannot print to the main console. Capture stdout/stderr using outfile or structured logging packages.

```
library(parallel)
cl <- makeCluster(4, outfile = "cluster.log")
# All print()/message()/warnings from workers go to cluster.log
# ... work ...
stopCluster(cl)

# Example with ParallelLogger (OHDSI)
# ParallelLogger::createLogger(name = "cluster", threshold = "INFO", appenders =
list(ParallelLogger::createFileAppender("cluster.log")))
```

For interactive debugging, packages like "partools" offer dbs() to launch worker R sessions in separate terminals so browser() can be stepped through.

```
library(partools)
dbs ()  # start debuggable cluster sessions
# Inside worker code, call browser() at points of interest
```

Inspect Variables and Intermediate Results
Add print()/message() at key points and direct output to logs; reduce data size or iterations to keep logs manageable.

```
# Example within a foreach loop
foreach(i = 1:M, .packages = c("mice")) %dopar% {
  message(sprintf("Starting iteration %d", i))
  # ... work ...
  message(sprintf("Completed iteration %d", i))
}
```

Reproducibility with Random Numbers
Control randomness with well-defined seeds; prefer per-iteration seeds or framework-provided reproducible seeding.

```
# foreach with explicit seeds
seeds <- 42 + seq_len(M)
res <- foreach(i = 1:M) %dopar% {
  set.seed(seeds[i])
  # stochastic work here
}

# future.apply with reproducible seeding
library(future); library(future.apply)
plan(multisession, workers = parallel::detectCores() - 1)
res <- future_lapply(1:M, function(i) {
  # body uses its own seed stream
  rnorm(10)
}, future.seed = TRUE)
```

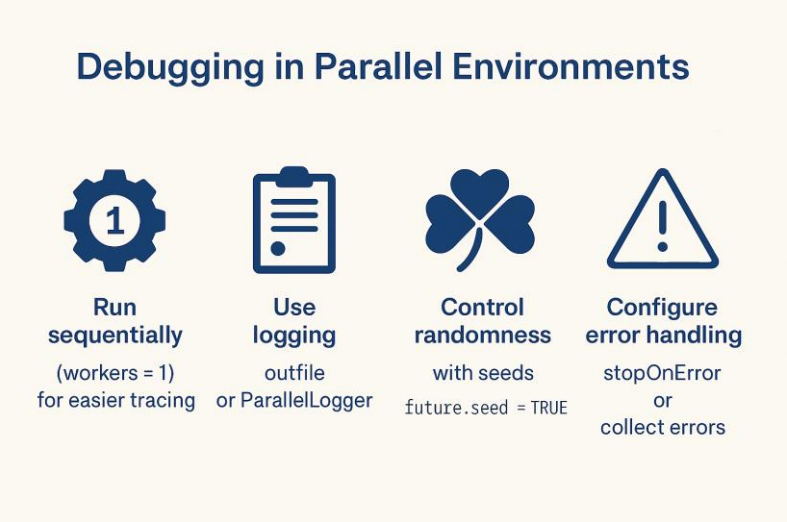
Understand Error Handling in Parallel Environments
Configure fail-fast behavior when needed or collect errors for later analysis, depending on your debugging strategy.

```
# Stop on first error (parallel::clusterApply)
clusterApply(cl, X, FUN, stopOnError = TRUE)

# foreach: collect and inspect errors
res <- foreach(i = 1:M, .errorhandling = "pass") %dopar% {
  risky_call(i)
}
# res will contain condition objects for failed iterations

# future.apply: propagate errors to the main session
res <- tryCatch({
  future_lapply(X, FUN)
}, error = function(e) {
  # handle/log and maybe stop
  e
})
```

Practical Debugging Checklist



• Start with workers = 1 to reproduce the issue sequentially.
• Log worker stdout/stderr to a file (outfile=...) and inspect it.
• Add targeted print()/message() calls around suspected problem areas.
• Control randomness with per-iteration seeds or future.seed = TRUE.
• Decide on fail-fast vs. collect-errors behavior; configure accordingly.
• Profile first: only parallelize hotspots where overhead is justified.

DISCUSSION

No single parallelization framework proved universally optimal across all scenarios. The effectiveness of each approach is contingent upon several factors, including:

- Size and complexity of the dataset
- Number of imputations or iterations
- Availability of computational resources (e.g., number of cores)
- Developer familiarity with parallel programming paradigms

While parallelization can significantly accelerate computation for large-scale tasks, it introduces overhead—particularly in the creation and management of child processes, data transfer, and result aggregation. For smaller tasks, this overhead may outweigh the benefits, resulting in slower execution compared to sequential methods. Therefore, profiling and benchmarking are essential to determine whether parallelization is justified for a given use case.

The strategies explored in this study are not limited to multiple imputations. They are broadly applicable to other repetitive and computationally intensive workflows in clinical programming, such as bootstrapping, simulation studies, and resampling techniques, where parallel execution can lead to substantial efficiency gains.

CONCLUSION

Parallel processing in R offers substantial performance improvements for iterative tasks such as multiple imputation. By selecting an appropriate parallelization framework, clinical programmers can develop faster and more efficient analytical pipelines without compromising reproducibility or transparency.

Once performance bottlenecks are identified through profiling, targeted optimization becomes essential. Several general strategies can be employed to enhance computational efficiency:

1. **Leverage existing optimized solutions**
2. **Minimize unnecessary computations**
3. **Utilize vectorized operations**
4. **Implement parallelization where appropriate**
5. **Avoid redundant data copying**
6. **Apply byte-code compilation for performance gains**

While parallel computation is particularly effective for *embarrassingly parallel* tasks, it is important to recognize that performance gains are not always linear with the number of processors. Overheads associated with process initialization, data transfer, and result aggregation can limit scalability. Therefore, careful profiling and method selection are critical to achieving optimal performance.

REFERENCES

- R Programming for Data Science: Roger D Peng - Ch 18, 22 https://bookdown.org/rdpeng/rprogdatascience/
- https://bookdown.org/rdpeng/rprogdatascience/debugging.html
- Multiple Imputation: https://dept.stat.lsa.umich.edu/~jerrick/courses/stat701/notes/mi.html
- Advanced R by Hadley Wickham: http://adv-r.had.co.nz/Profiling.html
- Flexible Imputation of missing data: https://stefvanbuuren.name/fimd/
- Berkeley Statistics: https://computing.stat.berkeley.edu/tutorial-parallelization/parallel-R.html
- Posit: https://support.posit.co/hc/en-us/articles/205612627-Debugging-with-the-RStudio-IDE
- R-Bloggers: https://www.r-bloggers.com/2017/08/implementing-parallel-processing-in-r
- mice R documentation : https://www.rdocumentation.org/packages/mice/versions/3.19.0
- microbenchmark R documentation : https://www.rdocumentation.org/packages/microbenchmark/versions/1.5.0
- Foreach R documentation: https://www.rdocumentation.org/packages/foreach/versions/1.5.2
- doParallel R documentation : https://www.rdocumentation.org/packages/doParallel/versions/1.0.17
- future.apply R documentation : https://www.rdocumentation.org/packages/future.apply/versions/1.11.2/topics/future.apply
- makeCluster R documentation: https://www.rdocumentation.org/packages/parallel/versions/3.6.2/topics/makeCluster
- ParallelLogger R documentation: https://www.rdocumentation.org/packages/ParallelLogger/versions/3.5.1
- partools R documentation: https://www.rdocumentation.org/packages/partools/versions/1.1.6

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Author Name: Sarita Singh
Company: Veramed Data Services Pvt. Ltd.
Address: Bangalore, India
Email: Sarita.singh@veramed.com