Parallelization in Multiple Imputation

Sven Nekula, Joshua Simon and Eva Wolf
Otto Friedrich University, Bamberg

What is Parallelization?

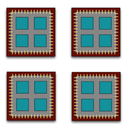

Figure 1: 4 processors with 4 cores each [2]

Parallelization is a technique to fasten time-consuming computations. It uses several cores on a **CPU** parallely and splits up the computational work between the cores. Afterwards, the results are merged. This can reduce the runtime significantly. The use of parallelization is however limited to certain tasks: The computations to be parallelized must be independent from each other, as information cannot be cross-accessed over the cores while the process is running.

Theory

In **parallel programming**, the multiple cores of a computational system want to be used best to decrease computation time. Gene **Amdahl** was the first one to desribe the boundaries of that project: Every parallel process also requires additional workload, so called "data management housekeeping". The speed up through parallel processing will tend to 0 at a certain amount of processing units involved, as this **overhead** workload exceeds the capacity of the computational unit which it is assigned to [2].

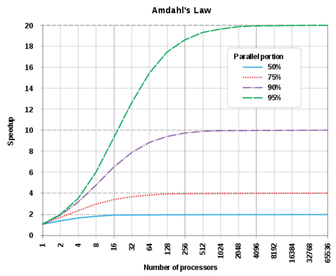

Figure 2: Amdahl's Law [2]

Multiple imputation MI is a method to complete a dataset with missing data. It relies on the estimation of the missing values through different methods. What is common to them is that not a single imputation run is used, but several. The results of all imputation runs are then merged and lead to realistic uncertainty of the estimators given the missing data. It is prone to be parallelized, as the several imputation runs can be processed independend of each other on different cores and easily merged afterwards [4].

Methodology

As the **dataset** we used a simple data generator of normally distributed random variables. The data set created contains 10 variables, some of which depend on each other. The sample size is $n = 10000$. Parallelization favors complex data sets, so the amount of variables and their interdependency should not be to small.

Time measurement was done with the *system.time* function, which returns three values: User CPU-, User System- and Elapsed time. User CPU is the time needed for the current task such as an execution in R. System CPU describes the time needed by the operating system to organize that task such as opening folders or asking for the System time. Elapsed time is the wall clock time that passed while the function and background processes were running.

The **method** of the **mice** algorithm was set on defaut for numeric data, pmm [4].

The **speed up** value is calculated by the serial time (runtime without parallelization) divided by the runtime of the current parallelization implementation [1]. Values below one show that the parallelized run took longer than the serial run.

Results

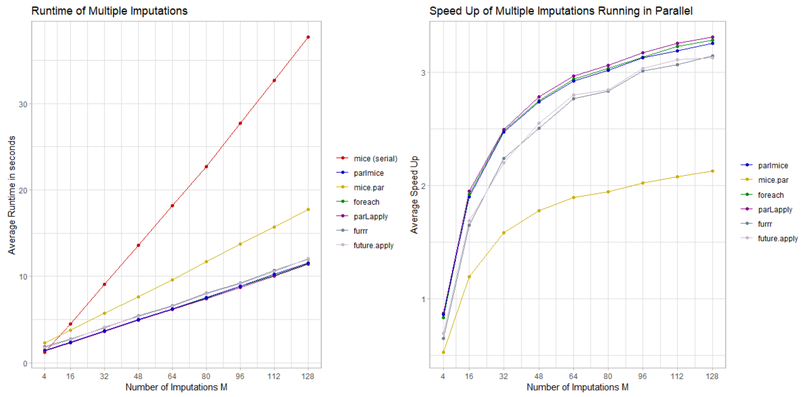

Figure 3: Runtime and Speedup from 4 up to 128 Imputation runs on a quad-core CPU

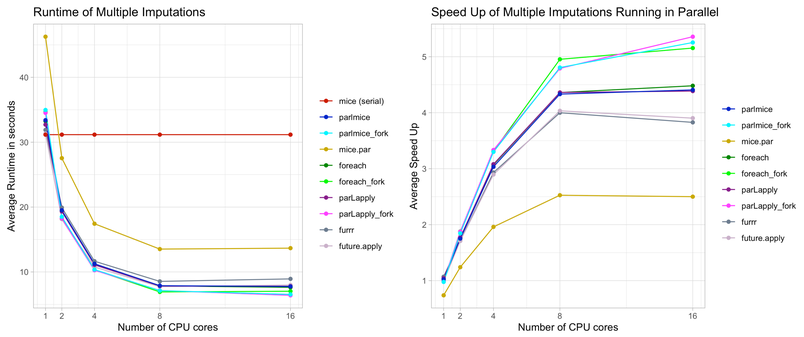

Figure 4: Runtime and Speedup from 1 up to 16 cores for $M = 128$ imputations

	Function	Cores	Average				
			User Time	System Time	Elapsed Time	Speed Up	Parallelism
1	foreach	16	0.25	0.03	7.63	4.48	0.83
2	foreach_fork	16	0.21	0.17	7.03	5.15	0.85
3	furrr	16	1.07	0.09	8.93	3.83	0.79
4	future.apply	16	0.89	0.07	7.99	3.90	0.79
5	mice (serial)	16	31.04	0.06	31.11	1.00	-
6	mice.par	16	1.15	0.10	13.67	2.50	0.64
7	parLapply	16	0.33	0.04	7.80	4.39	0.82
8	parLapply_fork	16	0.23	0.17	6.40	5.36	0.87
9	parlmice	16	0.30	0.02	7.76	4.41	0.82
10	parlmice_fork	16	0.21	0.18	6.54	5.25	0.86

Table 1: Time components for the different implementations for $M = 128$ imputations

Further Applications

The opposite case to paralellized processes is the intended time-intensive, serial processing of Big Data with **disk framing**. In that setting we have data that exceeds the **RAM capacity** of the system, so that the data needs to a split up into several smaller chunks. These are then processed in portions that fit the available memory. Nevertheless, also for **disk framing**, parallelization is a useful tool to decrease computation time. The system chooses as many chunks of data as they fit into the RAM and then processes them in parallel, so that use of the computational power of the system can be made, while the memory capacity limits the speed by the amount of data which is processed at once. [5]

Considering today's trends in parallel and high performance computing, **scalability** plays a big role. So-called distributed systems are used here. These can be seen as the fusion of several computers into one system, often referred to as a **computing cluster**. In order to run a computer program on such an architecture, a **message passing interface (MPI)** is used, which now handles the communication between multiple CPUs and their cores. Popular applications are cloud solutions like Amazon's AWS (Amazon Web Services). The R language can make use of MPI with packages like **Rmpi** or **pbdMPI** [1] and together with the **OpenMPI** implementation utilize the lower level of parallelism. [3]

Conclusion

- Parallelization is a powerful technique to reduce the runtime of complex workloads.
- **parLapply** generally has the best performance, because it not only utilizes parallelization, but also the vectorisation of the **apply** functions in R.
- When parallelizing MI, mice's built in **parlmice** function is recommended. It utilizes parLapply under the hood, which results in good performance, but it's easier to use. **mice.par** should be avoided due to poor performance.
- **parLapply** can be tricky to use inside of functions. **foreach, furr** and **future.apply** are good alternatives that only perform marginally worse.
- Parallel backends determine the operating system level utilization of parallel processes. Here, **FORK** yields better performance on UNIX systems. **PSOCK** is the default environment and works cross-platform.
- If RAM is an issue, **disk framing** in conjunction with parallelization is an alternative.
- Running complex workloads on clusters or multi-CPU systems requires a **MPI** and is a lot more difficult to implement.

References

[1] Simon R Chapple et al. *Mastering Parallel Programming with R*. Packt Publishing Ltd, 2016.

[2] Matt Jones. *Parallel Computing in R*. `https://nceas.github.io/oss-lessons/parallel-computing-in-r/parallel-computing-in-r.html`. Accessed: 2022-13-01. 2017.

[3] Robert Robey and Yuliana Zamora. *Parallel and High Performance Computing*. Simon and Schuster, 2021.

[4] Stef Van Buuren. *Flexible imputation of missing data*. CRC press, 2018.

[5] Dai Zhuo Jia and Jacky Poon. *Disk Framing*. `https://CRAN.R-project.org/package=disk.frame`. Accessed: 2022-14-01.