Performance Evaluation of Parallel Algorithms in R Statistical Package on Multicore Parallel Architectures

A. F. Kummer Neto\(^1\) and A. Charão\(^2\) and P. P. Barcelos\(^2\) and B. O. Stein\(^2\)

\(^1\)Post-Graduate Program in Computer Science
\(^2\)Department of Languages and Computing Systems
Federal University of Santa Maria, Brazil

Abstract—This paper presents an empirical performance evaluation of some parallel algorithms implemented in R statistical software. We focus on multicore parallel architectures, as other related works are more concerned with parallel distributed architectures. Such an evaluation is important, given that we live in an era in which the amount of data streams is very large, thus requiring high performance techniques and tools. The results show that the parallel algorithms can be effective but not so efficient.

Keywords: r statistical software; parallel algorithms; performance evaluation; multicore architecture

1. Introduction

R is a prominent free development environment used for statistical data analysis [1]. It comprises a domain specific programming language and a number of extensions and function libraries that implement algorithms for data mining such as sorting, clustering, association, among many others.

In statistical data analysis, there are many algorithms that can be significantly time-consuming, depending on the size of the input dataset. This becomes more of a concern in the current scenario of big, voluminous amount of data that is generated at speeding rates and has the potential to be analyzed for extracting useful information.

Given R’s popularity in this domain, and also given the advances on parallel computer architectures, some authors and developers proposed solutions for parallel data analysis using R [2], [3], [4]. This gave rise to packages such as multicore [5], snow [6] and, finally, parallel [7], which is included in the core distribution of R. The functions provided by these libraries can also be combined with other packages to speedup time-consuming data analysis, as with, for example, Caret (short for Classification and Regression Training) [8] and Boruta [9] packages. The first is a compilation of functions that facilitates the creation of data models. The second is a framework that implements algorithms for feature selection and machine learning tasks.

Back to Caret package, it is used to simplify complex regression and classification problems. One of its features is the capability of concurrent execution of R code snippets via multicore package. As a performance concern, the Caret package does not load its entire function collection at once. These are loaded as needed to prevent memory waste [8].

Concerning to parallel processing, we can explore parallelism in distributed and shared memory systems through snow [6] and multicore [5] R extensions, respectively. These packages manages parallel jobs, dividing the computation between the available cores/hosts available. Note that these extensions are available for in Unix-based systems only.

Since R 2.14.0, parallel package [12] offers a drop-in replacement for snow and multicore functionalities—some of them depicted in Table 1—and brings support for dynamic work scheduling between workers.
All functions listed in Table 1 have an interface similar to native \texttt{lapply} R function. \texttt{clusterApplyLB} and \texttt{parLapply} were designed to reduce communication between workers due to high overhead inherent to network communication. Communication is made via via sockets.

To exploit parallel processing in \textit{shared memory} architectures, we can use \texttt{mclapply} function to spawn workers on a multicore system. In current version of R, the workers are implemented via processes and piped communication since R interpreter is not reentrant.

### 3. Benchmarks and Performance Metrics

To evaluate performance gains achieved with parallelism facilities available in R, we build a set of four benchmark applications. The first is application aim to take a user perspective of parallel computing. The remaining benchmark applications are elaborated to stress specific parts of interpreter as an attempt to reach a more robust analysis of software infrastructure. A complete list of benchmark applications is available in Table 2.

<table>
<thead>
<tr>
<th>Benchmark application</th>
<th>Test case</th>
</tr>
</thead>
<tbody>
<tr>
<td>bench-caret</td>
<td>User point of view; automatic parallelism over Caret native support for multicore architectures</td>
</tr>
<tr>
<td>primes</td>
<td>Unbalanced, independent tasks with simple reduction procedure; low memory and communication requirements</td>
</tr>
<tr>
<td>primes-repeat</td>
<td>Repetition of homogeneous tasks; CPU-bound application</td>
</tr>
<tr>
<td>firesim</td>
<td>Monte Carlo simulation for fire spreading on forest</td>
</tr>
</tbody>
</table>

### 4. Case study: \texttt{bench-caret.R}

Elimination of meaningless information of a dataset is a common steps in data mining technique.

Elimination of meaningless information of a dataset is a common steps in data mining scenarios. An analysis with all the variables of a particular object of study is unviable, since the vast amount of data degrades the performance of algorithms and overload computational resources—sometimes worsening end result of analysis. To overcome this situations is usual to employ a characteristics selection filtering to eliminates any variable that is irrelevant for mining.

To elucidate the practice of data mining, we used a benchmark written in R called \texttt{bench-caret.R} [15]. It is structured as follows: initially, the benchmark loads the Caret and RandomForest packages (set of algorithms used for sorting); random samples of data with increasing size are generated; the generated data are analyzed by the code snippet responsible for processing. Lastly, the result is printed, stating the size of the sample tested and the time spent on execution.

To evaluate the performance of \texttt{bench-caret.R} benchmark, a computer with two Intel Xeon 2.00 GHz 4-core processors (8 cores the total) was used, 64-bit operating system Linux (Kernel 2.6.20), the statistical environment R with Multicore and Caret packages.

The tests were performed with 1 to 8 cores, with in each case three rounds were carried out the same test. This allowed a check fluctuations in time for the same number of cores. After collecting performance data, one can calculate the mean, standard deviation and the coefficient of variation for each core, encompassing three rounds each. There were more executions performed because the coefficient of variation was low (from 0 to 0.03) for all cases. Furthermore, a round was performed without multicore package to ascertain the effect on function of time.

As shown in Figure 1, the test took longer was the use of only one core. There was a big difference in time using...
1, 2, 3, 4 and 5 cores. From there the differences became smaller. It should be noted that with 5 and 6 cores, times were nearly identical. Intuitively, one might think that with 8 test centers would have been done faster. However, it was not what happened. The test performed faster with 7 cores. For the sample sizes of 4, 5, 6, 7, 8, 9 and 10, the 8 cores were slower than 7. By rotating the test without Multicore, there was a decrease in execution time regarding the test 1 with core and with the function. This is because this type of function generates overhead.

Figure 2 brings the result of acceleration (speedup). It has a representation of what would be the ideal performance, but as you can see, is not what happens. Note, however, that the higher the data size, the greater the acceleration.

5. Case study: primes application

The primes application is used to calculate how many primes exists from 1 to a upper limit value \(ub\), as illustrated in Routine (1). On the parallel version, we perform computations of lines (4 – 8) concurrently through functions of Table (1). As output, the application count how many primes exists in range \((1, ub]\).

For the following results, we conducted our computational experiments on a Dell PowerEdge R720 machine equipped with two Intel Xeon E5-2697 processors (each one working at 2.7 GHz) and 64 GiB of main memory (DDR3 1600 MHz SDRAM). As the version of R available on repositories of distribution used by this machine is old (R 3.1.1 in Debian 8 official repositories), we opted to a manual install of official R distribution 3.2.4 with standard library bundle. The machine uses a 64-bit Linux operating system 3.16.0.

The first Figure (3) show speedup reached up to 24 workers. The results show that primes application does not scales well on any of test cases, mainly due to unbalanced nature of tasks. Contrary to what we expect, dynamic assignment of task does not help to improve speedup of any test cases.

On package documentation, the authors pointed communication overhead could make dynamic load balancing prohibitive. That does not justificates poor performance of primes application since each task accomplished need to communicate only one byte to job scheduler process.

Low speedup rates result in poor parallelism efficiency. In
the best case (speedup of $\approx 6$ for $n = 12, 24$), parallelism efficiency is always below 60%--value which decrease as number of workers grows up.

6. Case study: primes-repeat application

Similar to previous benchmark, primes-repeat application test for primes repeatedly over a small range of values. The choice of elements is such that each number takes almost the same time to be tested. For this, we substitute the range presented at line (2) of Routine (1) by a list of numbers to test.

We use R bundle function `rep.int(x, times)` to build a list with `times` repetitions of `x` value and use as input of benchmark. With a list build by `rep.int(5 \times 10^5, 5 \times 10^3)` we verified almost linear speedups as shown in Figure (5).
As expected, we observed great efficiency rates for all test cases with static task scheduling, as show in Figure (6). The best results are obtained with mclapply with an average 89% of efficiency. Besides, worst results are found with mclapply too, but with dynamic task schedule, due to repetitive creation and destruction of worker processes [12]. The remaining results pointed that clusterApplyLB and parLapply obtained 87% of average efficiency for both static and dynamic task scheduling (2% worse than best results) and up to 16 cores.

Increase from 16 to 24 cores do not promote any improvement in computation times. Besides, we verified a efficiency drop of 29% of efficiency for mclapply and static work schedule, the test case of best results for primes-repeat application.

7. Case study: firesim application

firesim is a benchmark application based on a Monte Carlo method to simulate fire spreading on a forest. The application perform several trials to estimate the percentage of forest burnt with distinct fire spread probabilities.

The application uses a matrical representation of forest, in which each position of matrix is a tree. Each tree can be in one of the following states: unburnt, smoldering, burning and burnt. A tree in burning state can spread fire to its neighbors following a 4-connected model.

Each trial starts with one smoldering tree. The trial ends after several iterations of 4-connected fire spreading simulation, when there is no trees in smoldering and burning state.

In our computational experiments we used a 30 × 30 forest (900 trees), spreading probabilities from 0% to 100%–a total of 10 spreading scenarios, each one repeated 5000 times.

Similar to previous results, we observed low speedup rates–up to 14 times faster than serial implementation–as pointed in Figure 7. The result is shown in Figure 8 indicates less than 60% of parallelism efficiency when all cores of machine be used.

8. Conclusions

The combination of Caret and multicore packages was effective and achieved satisfactory results with relatively large data samples. The [15] benchmark revealed that the parallel processing feature improves the performance of the tests in most cases. The acceleration, however, does not hold up to eight available cores.
The other tests shown that great speedup and parallelism efficiency are achieved with coarse-grained tasks, as well as dynamic job scheduling proved to be ineffective to surpass unbalanced tasks.

It is intended to continue this research, primarily to ascertain the cause of the poor performance with 8 cores. This will only be possible with a deeper analysis of the Caret packages, Multicore and own mining algorithm.

References


