parallelDist

Alexander Eckert

parallelDist version 0.2.6 as of February 3, 2022

Abstract
This document highlights the performance gains for calculating distance matrices with the parallelDist package and provides basic usage examples.

Contents

1 Introduction 1
2 Performance 1
3 Quick start 3
   3.1 Using matrices as input parameter 3
   3.2 Using a list of matrices as input parameter 4
   3.3 Using user-defined distance functions 4
   3.4 Using objects of other R packages 5

1 Introduction
The parallelDist package provides a fast parallelized alternative to R’s native dist function to calculate distance matrices for continuous, binary, and multi-dimensional input matrices and offers a broad variety of predefined distance functions from the stats, proxy and dtw R packages, as well as support for user-defined distance functions written in C++. For ease of use, the parDist function extends the signature of the dist function and uses the same parameter naming conventions as distance methods of existing R packages.

The package is mainly implemented in C++ and leverages the Repp [EF11] and RcppParallel [AFU+16] package to parallelize the distance computations with the help of the TinyThread library. Furthermore, the Armadillo linear algebra library [San10] is used via RcppArmadillo [ES14] for optimized matrix operations for distance calculations. The curiously recurring template pattern (CRTP) technique is applied to avoid virtual functions, which improves the Dynamic Time Warping calculations while keeping the implementation flexible enough to support different step patterns and normalization methods.

2 Performance
The initial motivation for building this package was the need for a fast Dynamic Time Warping implementation which uses multiple cores and supports multi-dimensional (time) series. DTW
Figure 1: Distance matrix computation time for Dynamic Time Warping

is an expensive distance measure, where the computation of the DTW distance between two
series of length $N$ has a complexity of $O(N^2)$. This motivates an efficient and parallelized
implementation in C++.

Figure 1 shows a performance comparison between the \texttt{parDist} function of \texttt{parallelDist} and
the \texttt{dist} function in conjunction with the \texttt{dtw} package.

The benchmark has been performed on a system with the following specifications:
\begin{itemize}
  \item Intel(R) Xeon(R) E3-1230 v3 @ 3.30 GHz, 4 cores with hyper-threading
  \item 32 Gb RAM
\end{itemize}

As depicted in figure 1, \texttt{parDist} makes the calculation of large distance matrices with DTW
up to 3 orders of magnitudes faster.

The \texttt{parDist} function can be used as a replacement for the \texttt{dist} function of the \texttt{stats} package,
since it supports all other distance methods of the \texttt{stats} package and most of the distances of
the \texttt{proxy} package. Figure 2 shows the performance comparison of the \texttt{parDist} function with
Distance matrix computation time (5000 series of length 10)

Figure 2: Distance matrix computation times

the distance methods of \texttt{stats} and the \texttt{proxy} package when calculating distance matrices with 5000 series of length 10.

3 Quick start

3.1 Using matrices as input parameter

The function signature of \texttt{parDist} is based on dist. To calculate a distance matrix for 10 series of length 10, a matrix is passed to the \texttt{parDist} function where each row corresponds to one series.

> # matrix where each row corresponds to one series
> sample.matrix <- matrix(c(1:100), ncol = 10)

Here the \texttt{parDist} function calculates the distance matrix using the euclidean distance and
returns a dist object, like the dist function.

> # euclidean distance
> dist.euclidean <- parDist(sample.matrix, method = "euclidean")

The dist object can easily converted into a matrix, or can be used as an input for R’s clustering algorithms.

> # convert to matrix
> as.matrix(dist.euclidean)
> # create hierarchical agglomerative clustering model
> hclust.model <- hclust(dist.euclidean, method="ward")

Some distance methods require additional arguments (see ?parDist). These additional arguments can be passed directly to the `parDist` function.

> # minkowski distance with parameter p=2
> parDist(x = sample.matrix, method = "minkowski", p=2)
> # dynamic time warping distance normalized with warping path length
> parDist(x = sample.matrix, method = "dtw", norm.method="path.length")

A list of all available distance methods can be found in the `parDist` documentation.

> ?parDist

The number of threads to use can be set via the `threads` parameter.

> # use 2 threads
> dist.euclidean <- parDist(sample.matrix, method = "euclidean", threads = 2)

### 3.2 Using a list of matrices as input parameter

`parDist` also supports the calculation of distances between multi-dimensional series. Instead of one single matrix a list of matrices is used as input parameter. One matrix with M rows and N columns corresponds to a series with M dimensions and length N.

In the example below, a list with 2 matrices is defined where each matrix corresponds to a series with 2 dimensions of length 10.

> # defining a list of matrices, where each
> # list entry row corresponds to a two dimensional series
> tmp.mat <- matrix(c(1:40), ncol = 10)
> sample.matrix.list <- list(tmp.mat[1:2,], tmp.mat[3:4,])

The sample matrix now can be used to calculate a distance matrix for the multi-dimensional DTW distance.

> # multi-dimensional dynamic time warping
> parDist(x = sample.matrix.list, method = "dtw")

### 3.3 Using user-defined distance functions

Since version 0.2.0 of `parallelDist` custom user-defined distance measures can be defined to calculate distances matrices in parallel. To ensure a performant execution, the user-defined function needs to be defined and compiled in C++ and an external pointer to the compiled C++ function needs to be passed to `parDist` with the `func` argument.

The user-defined function needs to have the following signature:
double customDist(const arma::mat &A, const arma::mat &B)

Note that the return value must be a double and the two parameters must be of type const arma::mat &param. More information about the Armadillo library can be found at [Arm] or as part of the documentation of the RcppArmadillo [ES14] package.

Defining and compiling the function, as well as creating an external pointer to the user-defined function can easily be achieved with the cppXPtr function of the RcppXPtrUtils package. The following code shows a full example of defining and using a user-defined euclidean distance function:

```r
> # RcppArmadillo is used as dependency
> library(RcppArmadillo)
> # Use RcppXPtrUtils for simple usage of C++ external pointers
> library(RcppXPtrUtils)
> # compile user-defined function and return pointer (RcppArmadillo is used as dependency)
> euclideanFuncPtr <- cppXPtr("double customDist(const arma::mat &A, const arma::mat &B) {
+ return sqrt(arma::accu(arma::square(A - B))); }",
+ depends = c("RcppArmadillo"))
> # distance matrix for user-defined euclidean distance function
> # (note that method is set to "custom")
> parDist(matrix(1:16, ncol=2), method="custom", func = euclideanFuncPtr)
```

As displayed in table 1, the performance between a user-defined and a predefined distance function is close to equal for large matrices.

<table>
<thead>
<tr>
<th>matrix</th>
<th>method</th>
<th>min</th>
<th>lq</th>
<th>mean</th>
<th>median</th>
<th>uq</th>
<th>max</th>
<th>neval</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 10x10</td>
<td>euclidean</td>
<td>0.04</td>
<td>0.05</td>
<td>0.08</td>
<td>0.06</td>
<td>0.13</td>
<td>0.17</td>
<td>100.00</td>
</tr>
<tr>
<td>2 10x10</td>
<td>custom</td>
<td>0.16</td>
<td>0.18</td>
<td>0.24</td>
<td>0.22</td>
<td>0.30</td>
<td>0.40</td>
<td>100.00</td>
</tr>
<tr>
<td>3 100x10</td>
<td>euclidean</td>
<td>0.09</td>
<td>0.11</td>
<td>0.14</td>
<td>0.13</td>
<td>0.18</td>
<td>0.24</td>
<td>100.00</td>
</tr>
<tr>
<td>4 100x10</td>
<td>custom</td>
<td>0.22</td>
<td>0.24</td>
<td>0.31</td>
<td>0.29</td>
<td>0.37</td>
<td>0.51</td>
<td>100.00</td>
</tr>
<tr>
<td>5 1000x10</td>
<td>euclidean</td>
<td>4.19</td>
<td>4.29</td>
<td>4.44</td>
<td>4.35</td>
<td>4.50</td>
<td>5.54</td>
<td>100.00</td>
</tr>
<tr>
<td>6 1000x10</td>
<td>custom</td>
<td>4.34</td>
<td>4.48</td>
<td>4.66</td>
<td>4.60</td>
<td>4.80</td>
<td>5.14</td>
<td>100.00</td>
</tr>
<tr>
<td>7 10000x10</td>
<td>euclidean</td>
<td>448.87</td>
<td>451.33</td>
<td>490.99</td>
<td>453.26</td>
<td>465.36</td>
<td>644.65</td>
<td>100.00</td>
</tr>
<tr>
<td>8 10000x10</td>
<td>custom</td>
<td>452.83</td>
<td>454.99</td>
<td>492.68</td>
<td>456.36</td>
<td>464.66</td>
<td>678.49</td>
<td>100.00</td>
</tr>
</tbody>
</table>

Table 1: Performance comparison between user-defined and predefined euclidean distance function (in ms)

### 3.4 Using objects of other R packages

The parDist supports different kinds of step patterns for calculating DTW distance matrices (see ?parDist). For ease of use, it is also possible to use the StepPattern objects of the dtw package as input parameters for parDist.

```r
> # load dtw package
> library(dtw)
> # print the step pattern
> print(symmetric2)
> # use the symmetric2 object as input parameter for the parDist function
> parDist(x = sample.matrix, method = "dtw", step.pattern = symmetric2)
```
References


