MULTI- AND MIXED-PRECISION COMPUTATIONS IN R (MPCR)

Contents

1. **Introduction** 1
2. **Installation** 2
2.1. **Installation in R** 2
2.2. **Installation in C++** 2
2.3. **BLAS and LAPACK Libraries** 2
3. **Methodology** 3
3.1. **Data Type Promotion** 3
4. **Basic Usage** 3
5. **Maximum Likelihood Estimation of Matérn Covariance Function: A Real-Life Example** 5
6. **Benchmarks** 10
References 10

1. Introduction

Double-precision was once standard in scientific computations like weather forecasting [4], numerical linear algebra [1], and quantum models[5]. However, there is growing interest in using lower precision to reduce costs and improve performance in hardware acceleration, execution time, memory, and energy use. This approach has significantly impacted various fields, including numerical linear algebra, quantum chemistry, computational fluid dynamics, and deep learning.

MPCR is a package in R designed for multi- and mixed-precision computations, accommodating 64-bit and 32-bit data structures. This flexibility enables fast execution across various applications. The package enhances performance by optimizing operations in both precision levels, which is achieved by integrating with high-speed BLAS and LAPACK libraries like MKL and OpenBLAS. Including a 32-bit option caters to applications where high precision is unnecessary, accelerating computational processes whenever feasible. The package also provides support for tile-based algorithms in three linear algebra operations: CHOL(), TRSM(), and GEMM(). The tile-based algorithm splits the matrix into smaller tiles, facilitating parallelization through a predefined Directed Acyclic Graph (DAG) for each operation [3]. OpenMP enhances the efficiency of these operations, leveraging multi-core parallelism. Optional GPU acceleration via CUDA is also available for supported operations when CUDA Toolkit is detected. In this case, MPCR facilitates mixed-precision execution by permitting varying precision levels for different tiles [2]. This approach is advantageous in numerous applications, as it

maintains the accuracy of the application while accelerating execution in scenarios where single-precision alone does not significantly affect the accuracy of the application.

2. Installation

One of the notable additions in MPCR is the enhanced capability to execute and install the package seamlessly on both C++ and R environments. This extended functionality provides developers with increased flexibility, enabling them to run, test, and incorporate the package effortlessly into any existing C++ or R project.

2.1. **Installation in R.** You can install the stable version from CRAN using the usual install.packages():

```
install.packages("MPCR")
```

You can also install the latest release from GitHub using the following command:

```
remotes::install_github("stsds/MPCR")
```

For GPU acceleration, ensure CUDA Toolkit ($\geq$11.2) is installed. The package automatically detects CUDA during installation and enables GPU support for 32-bit and 64-bit operations. Note that 16-bit operations require GPU with half-precision support for Matrix-Matrix Multiplication only (`crossprod()`).

2.2. **Installation in C++.** To build the code as a C++ code with testing enabled, run the following:

```
cd MPCR/
./config.sh -t
./clean_build.sh
```

2.3. **BLAS and LAPACK Libraries.** A significant enhancement in the MPCR package involves its streamlined integration with an optimized BLAS/LAPACK library. Typically, most existing R packages link with RBLAS by default, and when an optimized library is necessary, users must make R environment modifications (e.g., building R from scratch with a different BLAS backend or altering RBLAS symbolic linking). This approach demands a comprehensive understanding of the modification process. Moreover, any library previously using RBLAS will be automatically directed to the newly linked library, introducing potential complexities.

In contrast, MPCR offers a distinct approach to choosing the BLAS/LAPACK library backend. The package first searches for any available BLAS backend on the

system (e.g., MKL, OpenBLAS, ATLAS, etc..). If found, MPCR seamlessly links with the located BLAS backend. Alternatively, if no existing backend is found, the package autonomously installs and links OpenBLAS in its dedicated space without any alterations to the system environment. Once the package installation is complete, there is no need for any further system modifications. Additionally, the behavior of other packages dependent on RBLAS remains unaffected by the changes introduced by MPCR. This method simplifies the utilization of optimized libraries and enhances the overall user experience. For GPU operations, MPCR utilizes cuBLAS and cuSOLVER libraries, which are included with the CUDA Toolkit and provide GPU-accelerated equivalents of BLAS and LAPACK routines.

---

3. **Methodology**

The package is entirely crafted in pure C++, seamlessly integrated with R through the Rcpp package. This approach yields numerous advantages, enriching the overall design of the package in multiple dimensions. This design facilitates the following:

(1) Enables fast and optimized development in C++ without the need for any in-depth Rcpp knowledge.
(2) Leverage the high-performance capabilities of C++ and write efficient code. This integration enhances the ability of R packages to handle large datasets more effectively, enabling improved efficiency in data processing.
(3) By leveraging C++ templates, the package provides seamless and rapid support for additional operations or precision adjustments, ensuring flexibility and ease of customization when required.

3.1. **Data Type Promotion.**

In MPCR, two primary promotion strategies are employed. For normal MPCR matrix/vector operations, promotion is executed to match the higher precision input. For instance, the operation involving single- and double-precision inputs results in a double-precision output. The BLAS/LAPACK operations in linear algebra operations necessitate the same precision input. Therefore, if an operation involves single and double-precision inputs, the single-precision input is promoted to double-precision before the operation is executed.

In the case of MPCR-Tile matrices, the promotion strategy differs. The precision of the output tile determines the promotion of input tiles. For example, if single- and double-precision tiles are used as input and the output tile is single-precision, the double-precision tile will be de-promoted to single-precision.

---

4. **Basic Usage**

This flow shows how the MPCR package can be used to create MPCR objects, i.e., vectors and matrices, and perform some basic operations on these data structures.

```
# Create new MPCR vector with 32-bit precision and size 50
# The vector will be allocated with all values equal to 0
```

```
MPCR_object <- new(MPCR,50,"single")

# To change the vector to matrix representation
# Represented in column major format, the following command can be
# used
MPCR_object$ToMatrix(5,10)

# And to print the MPCR object metadata
# Can be access using the properties of the S4 object.
MPCR_object$Row       #5
MPCR_object$Col       #10
MPCR_object$Size      #50
MPCR_object$IsMatrix  #TRUE

# To print the MPCR object values
MPCR_object$PrintValues()

# Changing the representation back to vector
MPCR_object$ToVector()

# Priting the object summary
MPCR_object

# GPU Usage Example
# Create MPCR object with GPU memory allocation
MPCR_gpu <- new(MPCR, 50, "single", "GPU")

# Set operations to execute on GPU (if available)
MPCR.SetOperationPlacement("GPU")

# Check GPU allocation status
MPCR_gpu$IsGPUAllocated() # Returns TRUE if allocated on GPU

# Free GPU memory when needed
MPCR_gpu$FreeGPU()

# To convert R matrix or vector to MPCR object.
x <- as.MPCR(c(1.21, 0.18, 0.13, 0.41, 0.06, 0.23,
                0.18, 0.64, 0.10, -0.16, 0.23, 0.07,
                0.13, 0.10, 0.36, -0.10, 0.03, 0.18,
                0.41, -0.16, -0.10, 1.05, -0.29, -0.08,
                0.06, 0.23, 0.03, -0.29, 1.71, -0.10,
                0.23, 0.07, 0.18, -0.08, -0.10, 0.36),
                nrow=6,ncol=6,precision="double")
# Perform cholesky decomposition on the MPCR matrix
chol_out <- chol(x)
```

5. Maximum Likelihood Estimation of Matérn Covariance Function: A Real-Life Example

An example demonstrating the advantages of the MPCR package is its application in evaluating high-dimensional likelihood functions, which are commonly needed in spatial statistics. Suppose we have a 2D Gaussian random field with 14,400 locations, each associated with a single measurement.

Our goal is to fit this Gaussian field using the Matérn spatial covariance function:

$$\operatorname{cov}\{Z(\mathbf{s}_i), Z(\mathbf{s}_j)\} = \frac{\sigma^2}{2^{\nu-1}\Gamma(\nu)} \mathcal{M}_\nu \left(\frac{\|\mathbf{s}_i - \mathbf{s}_j\|}{a} \right), \tag{5.1}$$

parameterized by $\boldsymbol{\theta} = (\nu, a, \sigma^2)^\top$, where $\nu > 0, a > 0, \sigma^2 > 0$ are the smoothness, spatial range, and variance parameters, respectively. Here, $\mathcal{M}_\nu(x) = x^\nu \mathcal{K}_\nu(x)$, $\mathcal{K}_\nu(\cdot)$ is the modified Bessel function of the second kind of order ν, and $\Gamma(\cdot)$ is the gamma function.

Algorithm 1 Adaptive Precision-Aware Runtime Decision.

1: Compute the Frobenius norm of $\boldsymbol{\Sigma}(\boldsymbol{\theta})$, i.e., $\|\boldsymbol{\Sigma}(\boldsymbol{\theta})\|_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} |\Sigma_{ij}(\boldsymbol{\theta})|^2}$.
2: Set the value of NT, the number of tiles in one dimension.
3: Divide $\boldsymbol{\Sigma}(\boldsymbol{\theta})$ into tiles, i.e., $\boldsymbol{\Sigma}_{ij}(\boldsymbol{\theta}), i = 1, \ldots, M_1$ and $j = 1, \ldots, M_2$, such that $NT = M_1 M_2$.
4: Choose two precisions for the mixed-precision arithmetic and set the values of the variables u_{low} and u_{high} to their equivalent machine epsilons, e.g., $u_{\text{low}} = 2^{-24}$ and $u_{\text{high}} = 1e^{-8}$.
5: **for** $i = 1 : M_1$ **do**
6: **for** $j = 1 : M_2$ **do**
7: Compute the Frobenius norm of tile $\boldsymbol{\Sigma}_{ij}(\boldsymbol{\theta})$.
8: **if** $\|\boldsymbol{\Sigma}_{ij}(\boldsymbol{\theta})\|_F \leq u_{\text{high}}\|\boldsymbol{\Sigma}(\boldsymbol{\theta})\|_F/(u_{\text{low}}NT)$ **then**
9: Store tile $\boldsymbol{\Sigma}_{ij}(\boldsymbol{\theta})$ in the lower (single) precision.
10: **else**
11: Store tile $\boldsymbol{\Sigma}_{ij}(\boldsymbol{\theta})$ in the higher (double) precision.
12: **end if**
13: **end for**
14: **end for**

(1) ***double-precision***

(a) Simulate a sample 2D spatial field from a Matérn spatial covariance function model.

```
cov.matern <- function(x, nu, a, sigma_sq){
if(nu == 0.5) return(sigma_sq*exp( -x / a))
ismatrix <- is.matrix(x)
if(ismatrix){nr=nrow(x); nc=ncol(x)}
x <- c(x / a)
output <- rep(1, length(x))
n <- sum(x > 0)
if(n > 0) {
  x1 <- x[x > 0]
  output[x > 0] <-
    (1/((2^(nu - 1)) * gamma(nu))) * (x1^nu) * besselK(x1, nu)
}
if(ismatrix){
  output <- matrix(output, nr, nc)
}
return(sigma_sq * output)
}

M <- 120
n.loc <- M*M
locs <- cbind(rep(0:(M-1), M)/(M-1), rep(0:(M-1), each=M)/(M-1))
x <- as.matrix(dist(locs)) # distance matrix
theta <- c(1, 0.05, 1) # true parameters
cov.R <- cov.matern(x, theta[1], theta[2], theta[3])

# Simulate the spatial field.
library(mgcv)
set.seed(4)
z.R <- rmvn(1, rep(0,M*M), cov.R)
```

(b) Create a function that computes the value of the negative Gaussian log-likelihood function. The function `nll` below takes in the argument `pars`, which is an `R` vector object that represents $\boldsymbol{\theta}$. Moreover, the parameter values contained in the vector `pars` are transformed to ensure that their values remain within their valid ranges.

```
nll <- function(pars, type, precision){
nu_param = 2 * 1 / (1 + exp(-pars[1]))
a_param = exp(pars[2])
sigma_param = exp(pars[3])

#Creating the covariance matrix as R-double-precision
V <- cov.matern(x, nu_param, a_param, sigma_param)
```

```
if(type == 'FULL'){
  if(precision == 'MPCR-Double'){
    #Casting the covariance matrix as MPCR double-precision
    cov_mpcr <- as.MPCR(V, nrow=M*M, ncol=M*M, precision='double')
    L <- chol(cov_mpcr)
    d <- log(diag(L))
    log.det.cov <- 2*d$Sum()
    z.mpcr <- as.MPCR(z.R, nrow=M*M, ncol=1, precision='single')
    inner_product <- MPCR.trsm(a=L, b=z.mpcr, side='L',
      upper_triangle=T, transpose=T, alpha=1)
    inner.prod <- inner_product$SquareSum()
  }else if(precision == 'R-Double'){
    L <- t(chol(V))
    log.det.cov <- 2*sum(log(diag(L)))
    z.new <- forwardsolve(L, z.R)
    inner.prod <- sum(z.new^2)
  }
}else if(type == 'BANDED'){
  cov.tile <- new(MPCRTile, nr, nc, tr, tc, V, prec.cov.banded)
  # NOTE: the result of chol is a lower triangular matrix
  cov.tile_chol <- chol(cov.tile, overwrite = F, num_threads = 4)
  d <- log(cov.tile_chol$Diag())
  log.det.cov <- 2*d$Sum()
  z.tile <- new(MPCRTile, nr, 1, tc, 1, z.R, prec.z)
  MPCRTile.trsm(a=cov.tile_chol, b=z.tile, side='L',
    upper_triangle=F, transpose=F, alpha=1)
  inner.prod <- z.tile$SquareSum()
}else if(type == 'ADAPTIVE'){
  cov.tile <- new(MPCRTile, nr, nc, tr, tc, V, prec.cov)
  V_norm <- sqrt(sum(V^2))
  upper_bound <- u_high * V_norm / (nt * u_low)
  for(ii in 1:tr_total){
    for(jj in 1:tc_total){
      test <- MPCRTile.GetTile(cov.tile, ii, jj)
      tile_norm <- sqrt(test$SquareSum())
      if(tile_norm < upper_bound){
        cov.tile$ChangeTilePrecision(ii, jj, "single")
      }
    }
  }
  # NOTE: the result of chol is a lower triangular matrix
  cov.tile_chol <- chol(cov.tile, overwrite = F, num_threads = 4)
  d <- log(cov.tile_chol$Diag())
  log.det.cov <- 2*d$Sum()
  z.tile <- new(MPCRTile, nr, 1, tc, 1, z.R, prec.z)
  MPCRTile.trsm(a=cov.tile_chol, b=z.tile, side='L',
```

```
    upper_triangle=F, transpose=F, alpha=1)
  inner.prod <- z.tile$SquareSum()
}
# Computing the negative Gaussian log-likelihood
nll <- 0.5*inner.prod + 0.5*log.det.cov + 0.5*n.loc*log(2*pi)
return(nll)
}
```

(c) Perform non-linear optimization. In this example, we use the `nloptr` function with the `BOBYQA` subroutine to minimize the log-likelihood function.

```
library(nloptr)
opts <- list("algorithm" = "NLOPT_LN_BOBYQA", "xtol_rel" = 1e-8,
"maxeval" = 1000)
init <- c(-0.3, -1.5, -0.3)
fit_R <- nloptr(x0 = init, eval_f = log.likel, type = 'FULL',
precision = 'R-Double', opts = opts)
fit_mpcr_double <- nloptr(x0 = init, eval_f = log.likel,
type = 'FULL', precision = 'MPCR-Double', opts = opts)
```

(2) ***Mixed-Precision***

(a) Define the tile sizes and the low and high precisions involved in the mixed-precision MLE.

```
u_low = 2^(-24)
u_high = 1e-8

nr <- nc <- n.loc
tr <- 2400
tc <- 2400
tr_total <- n.loc / tr
tc_total <- n.loc / tc
nt <- tr_total * tc_total

# Precisions for z
prec.z <- matrix(rep("single", tr_total), tr_total, 1)

# Precisions for cov
create_prec_banded_mat <- function (prec_matrix, bandwidth) {
d <- dim(prec_matrix)
outside_band <- .row(d) - n >= .col(d) | .col(d) - n >= .row(d)
prec_matrix[outside_band] <- 'single'
return(prec_matrix)
}
prec.cov <- matrix(rep("double", nt), n.loc/tr, n.loc/tc)
prec.cov.banded <- create_prec_banded_mat(prec.cov)
```

(b) Perform non-linear optimization.

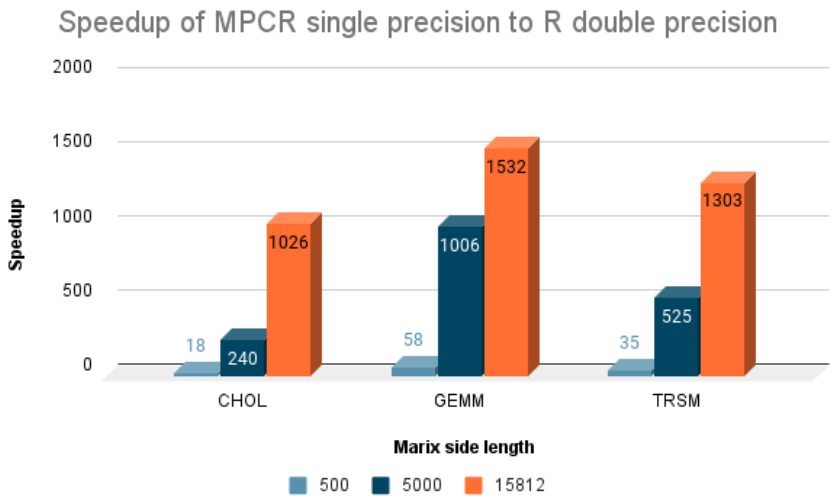
```
fit_tile_adaptive <- nloptr(x0 = init, eval_f = log.likel,
type = 'ADAPTIVE', precision = NULL, opts = opts)
fit_tile_banded <- nloptr(x0 = init, eval_f = log.likel,
type = 'BANDED', precision = NULL, opts = opts)
```

TABLE 1. Summary of MLE results under different types of precision for $n = 14{,}400$.

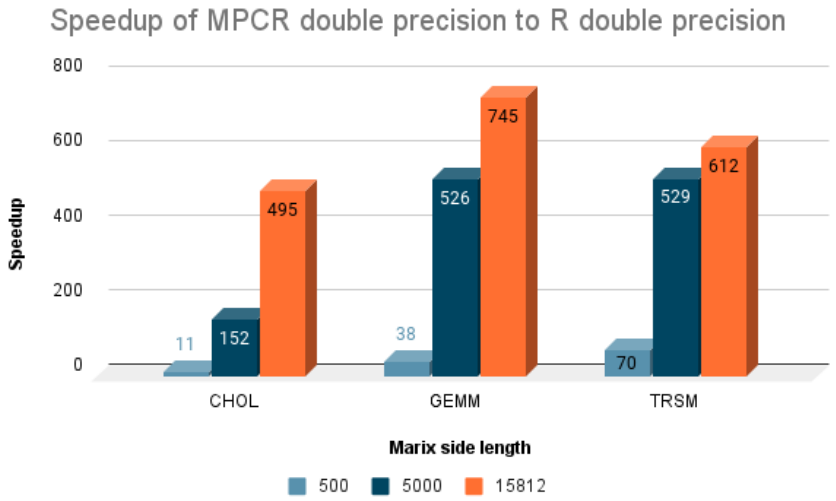
Precision	`nll`	Parameter Estimates	Execution Time
R-Double	$-7,077$	$\hat{\boldsymbol{\theta}} = (0.9862619, 0.0513225, 0.9893821)^\top$	48.96 hours
MPCR-Double	$-7,077$	$\hat{\boldsymbol{\theta}} = (0.9862613, 0.05132348, 0.9894147)^\top$	3.62 hours
MPCR-Mixed (adaptive)	$-7,077$	$\hat{\boldsymbol{\theta}} = (0.9862476, 0.05126803, 0.9873119)^\top$	3.44 hours
MPCR-Mixed (banded)	$-7,073$	$\hat{\boldsymbol{\theta}} = (1.054851, 0.04492239, 0.9298075)^\top$	1.97 hours

6. **Benchmarks**

This graph represents the speedup of MPCR single precision object to R double object in three major linear algebra functions.



This graph represents the speedup of MPCR double precision object to R double object in three major linear algebra functions.



References

1. Zvonimir Bujanović, Daniel Kressner, and Christian Schröder, *Iterative refinement of schur decompositions*, Numerical Algorithms **92** (2023), no. 1, 247–267.
2. Qinglei Cao, Sameh Abdulah, Rabab Alomairy, Yu Pei, Pratik Nag, George Bosilca, Jack Dongarra, Marc G Genton, David E Keyes, Hatem Ltaief, et al.,

Reshaping geostatistical modeling and prediction for extreme-scale environmental applications, SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2022, pp. 1–12.
3. Wolfgang Hackbusch et al., *Hierarchical matrices: algorithms and analysis*, vol. 49, Springer, 2015.
4. Simon TK Lang, Andrew Dawson, Michail Diamantakis, Peter Dueben, Samuel Hatfield, Martin Leutbecher, Tim Palmer, Fernando Prates, Christopher D Roberts, Irina Sandu, et al., *More accuracy with less precision*, Quarterly Journal of the Royal Meteorological Society **147** (2021), no. 741, 4358–4370.
5. Ryan Pederson, John Kozlowski, Ruyi Song, Jackson Beall, Martin Ganahl, Markus Hauru, Adam GM Lewis, Yi Yao, Shrestha Basu Mallick, Volker Blum, et al., *Large scale quantum chemistry with tensor processing units*, Journal of Chemical Theory and Computation **19** (2022), no. 1, 25–32.