Map/Reduce operations for scientific computing in Julia

Journée autour du langage Julia (Groupe Calcul CNRS)
Lyon

Xavier VASSEUR (xavier.vasseur@isae-supaero.fr)

January 31 2019, Journée Julia

Institut Supérieur de l'Aéronautique et de l'Espace

Why considering Julia for scientific computing ?

Why considering Julia for scientific computing ?

- Julia's **mathematical syntax** makes it an ideal way to express numerical algorithms.
- **Rich ecosystem** for scientific computing with state-of-the-art packages.
- Julia is able to handle **large volume** of data efficiently.
- Julia is designed for **parallelism**, and provides built-in primitives for parallel computing at every level: **instruction level parallelism**, **multi-threading** and **distributed computing**.
- The **Celeste.jl** project [Regier et al, 2018] achieved **1.5 PetaFLOP/s** on the Cori supercomputer at NERSC using **620,000** cores.
- Julia won the **2019 James H. Wilkinson Prize for Numerical Software**.

Current challenges in high performance data analytics

Current challenges in high performance data analytics

- **Increasingly large amount of data** in current applications (web search, machine learning, social networks, genomics/proteomics data, $\cdots$).
- **State-of-the-art** deterministic methods of numerical linear algebra were designed for an environment where the matrix **fits into memory** (RAM) and the key to performance was to minimize the number of **floating point operations** (FLOP) required.
- Currently, **communication** is the real bottleneck
  - Moving data from a hard drive
  - Moving data between nodes of a parallel machine
  - Moving data between nodes of a cloud computer.
- Ideally we should target for efficient algorithms scaling **linearly** with the problem size and with **minimal data movement**.

Current challenges in high performance data analytics

Main features to analyze

- **Data distribution**.
- **Load balancing** property of the algorithm.
- **Weak and strong** scalability properties of the algorithm.
- **Resiliency** and **fault-tolerant** properties of the algorithm.

Distributed data analysis and scientific computing [Gittens et al, 2016]

- Apache **Hadoop Map/Reduce** (**RDD**: Resilient Data Distribution).
- Spark Apache **MLlib**.
- **Message Passing Interface** (MPI).
- **R and Distributed R** (Rmpi, RHadoop).

Map/Reduce algorithms

Map/Reduce algorithms

- **Framework** for processing parallelizable problems across large datasets using a large number of nodes on a cluster.
- **Methodology**:
    - **Map**: Each **worker node** applies the "**map()**" function to the local data, and writes the output to a temporary storage. A master node ensures that only one copy of redundant input data is processed.
    - **Shuffle**: **Worker nodes** redistribute data based on the output keys (produced by the "**map()**" function), such that all data belonging to one key is located on the same worker node.
    - **Reduce**: **Worker nodes** now process each group of output data, per key, in parallel.
- An efficient **distributed file system** is usely required.

Goals of the talk

Goals of the talk

- Introduce **map/reduce strategies** available in Julia at an introductory level.
- Provide two **illustrations** with a main focus on the basic concepts.
- Discuss an **application** using map/reduce strategies in a HPC context.

My own experience in Julia

- Used for teaching in numerical analysis since 2016.
- Used for research in scientific computing (for its fast prototyping and efficiency).
- At a beginner level nevertheless !
- Julia (v 1.0.3) has been used in this talk.

Outline

1 Context

2 Map/Reduce operations in Julia
- Map, reduce and mapreduce operations
- Two simple examples
- Distributed map operation

3 A first illustration: mean computation

4 A second illustration: communication avoiding Cholesky-QR2 factorization

5 Conclusions

Map operation

```
help?> map
  search: map map! mapfoldr mapfoldl mapslices mapreduce asyncmap asyncmap!

  map(f, c...) -> collection

  Transform collection c by applying f to each element. For multiple
  collection arguments, apply f elementwise.

  See also: mapslices

  Examples
  ==========

  julia> map(x -> x * 2, [1, 2, 3])
  3-element Array{Int64,1}:
  2
  4
  6
```

Reduce operation

```
help?> reduce
  search: reduce mapreduce

  reduce(op, itr; [init])

  Reduce the given collection itr with the given binary operator op. If
  provided, the initial value init must be a neutral element for op that will
  be returned for empty collections. It is unspecified whether init is used
  for non-empty collections.

  ...

  Examples
  ==========

  julia> reduce(*, [2; 3; 4])
  24

  julia> reduce(*, [2; 3; 4]; init=-1)
  -24
```

Reduce operation

- The **reduce** operation operates on a collection (or iterable) (usually the result of map) and reduces it to a **single** object.
- **Example**: if a collection c has 4 elements, $reduce(op, c)$ successively calculates

$$op(c_1,\ c_2),$$

Reduce operation

- The **reduce** operation operates on a collection (or iterable) (usually the result of map) and reduces it to a **single** object.
- **Example**: if a collection c has 4 elements, $reduce(op, c)$ successively calculates

$$op(c_1,\ c_2),$$

$$op(\ op(c_1,\ c_2),\ c_3),$$

Reduce operation

- The **reduce** operation operates on a collection (or iterable) (usually the result of map) and reduces it to a **single** object.
- **Example**: if a collection c has 4 elements, $reduce(op, c)$ successively calculates

$$op(c_1,\ c_2),$$

$$op(\ op(c_1,\ c_2),\ c_3),$$

$$op(\ op(\ op(c_1,\ c_2),\ c_3),\ c_4).$$

- **reduce** computes the combination of the reduce result so far and a new element in the collection returned from a map operation.
- The associativity of the reduction operation is implementation dependent.

Mapreduce operation

```
help?> mapreduce
  search: mapreduce

  mapreduce(f, op, itr; [init])

  Apply function f to each element in itr, and then reduce the result using
  the binary function op. If provided, init must be a neutral element for op
  that will be returned for empty collections. It is unspecified whether init
  is used for non-empty collections. In general, it will be necessary to
  provide init to work with empty collections.

  mapreduce is functionally equivalent to calling reduce(op, map(f, itr);
  init=init), but will in general execute faster since no intermediate
  collection needs to be created. See documentation for reduce and map.

  Examples
  ==========

  julia> mapreduce(x->x^2, +, [1:3;]) # == 1 + 4 + 9
  14
```

Outline

1 Context

2 Map/Reduce operations in Julia
- Map, reduce and mapreduce operations
- Two simple examples
- Distributed map operation

3 A first illustration: mean computation

4 A second illustration: communication avoiding Cholesky-QR2 factorization

5 Conclusions

Midpoint rule integration (naive implementation)

```
function my_mapper_Integration_Midpoint_Rule(x)
    return 4./(1 + x^2)
end

function my_reducer_Integration_Midpoint_Rule(s_x,s_y)
    return s_x+s_y
end

function Integration_Midpoint_Rule(a, b, n)
     h  = (b-a)/float(n)
     s  = mapreduce((x)->my_mapper_Integration_Midpoint_Rule(x),
                    (x,y)->my_reducer_Integration_Midpoint_Rule(x,y),
                    [a + (i-0.5)*h for i=1:n])
     return h*s
end
```

$\int_a^b f(x)dx \approx \frac{(b-a)}{n}\sum_{i=1}^{n} f(x_i)$ with $x_i = a + (i - \frac{1}{2})\frac{(b-a)}{n}, \quad 1 \leq i \leq n.$

Monte Carlo integration

```julia
function my_array_function(x)
    return 4 ./ (1 .+ x.^2)     # To favor array operations
end

function my_mapper_Monte_Carlo(array_sample)
    return sum(my_array_function(array_sample))
end

function my_reducer_Monte_Carlo(s_x,s_y)
    return s_x+s_y
end

function MonteCarlo(a, b, n, nsample)
    s   = mapreduce((x)->my_mapper_Monte_Carlo(x),
                    (x,y)->my_reducer_Monte_Carlo(x,y),
                    [rand(Uniform(a,b),1,n) for i=1:nsample])
    return (b-a)/float(n*nsample)*s
end
```

Outline

1 Context

2 Map/Reduce operations in Julia
- Map, reduce and mapreduce operations
- Two simple examples
- Distributed map operation

3 A first illustration: mean computation

4 A second illustration: communication avoiding Cholesky-QR2 factorization

5 Conclusions

Distributed map operation

- **Distributed** computing is available in Julia through the Distributed package.
- This allows to perform **map** operations in a **distributed** setting using the notion of workers and tasks available in Julia.

```
help?> pmap
  search: pmap promote_shape typemax PermutedDimsArray process_messages

  pmap(f, [::AbstractWorkerPool], c...; distributed=true, batch_size=1,
  on_error=nothing, retry_delays=[], retry_check=nothing) -> collection

  Transform collection c by applying f to each element using available
  workers and tasks.

  For multiple collection arguments, apply f elementwise.

  ...
```

Distributed map operation: example

- Performing independent linear algebra operations is then made easy and efficient.
-

```julia
using LinearAlgebra
m, n = 1000, 500
M    = [rand(m,n) for i=1:10]
pmap(svd,M)
```

Outline

1 Context

2 Map/Reduce operations in Julia

3 A first illustration: mean computation
- Problem statement
- Implementation with Map/Reduce: analysis and results

4 A second illustration: communication avoiding Cholesky-QR2 factorization

5 Conclusions

Problem statement

- **Goal**: computation of the mean of a possibly large array $v \in \mathbb{R}^m$ with the map/reduce concept.
- **Data decomposition**: partition the array v in ℓ contiguous chunks of data $v_i \in \mathbb{R}^{m_i}$ $(1 \leq i \leq \ell)$ to be able to perform **independent** computations.
- **Map**: perform local mean computation related to a local array $w \in \mathbb{R}^{m_w}$:

$$\bar{w} \quad = \quad \frac{\sum_{j=1}^{m_w} w_j}{m_w},$$

- **Reduce**: Update the mean information by combining the current result of the reducer with a result of a mapper function.

Outline

1. Context
2. Map/Reduce operations in Julia
3. A first illustration: mean computation
   - Problem statement
   - Implementation with Map/Reduce: analysis and results
4. A second illustration: communication avoiding Cholesky-QR2 factorization
5. Conclusions

Analysis

- **Data decomposition**: partition the array v in ℓ contiguous chunks of data $v_i \in \mathbb{R}^{m_i} (1 \leq i \leq \ell)$ to be able to perform **independent** computations.
- **Map**: the mapper function performs local mean computation and should return as output the (m_i, μ_i) information (number of elements in v_i, local mean, respectively.)
- **Reduce**: the reducer function should combine two results (one related to the reducer, the other to a result of a map) to yield an update of the mean $((m_u, \mu_u))$ as follows:

$$
\begin{aligned}
m_u &= m_r + m_j, \\
\mu_u &= \frac{m_j \mu_j + m_r \mu_r}{m_u}.
\end{aligned}
$$

Implementation of the mapper function in Julia

```julia
function my_mapper(x)
    n        = length(x)
    average  = sum(x)/float(n)
    return (n,average)
end
```

Implementation of the reducer function in Julia

```
function my_reducer(element_x,element_y)
    n_x, average_x  = element_x
    n_y, average_y  = element_y
    n               = n_x + n_y
    average         = (n_x * average_x + n_y * average_y )/float(n)
    return (n,average)
end
```

```
function compute_average(x, nsub)
    nchunks = Int64(ceil(length(x))/nsub)
    s       = mapreduce((x)->my_mapper(x), (e_x,e_y)->my_reducer(e_x,e_y),
                  [x[(i-1)*nchunks+1:min(i*nchunks, length(x))] for i=1:nsub])
    return s
end
```

Outline

1. Context
2. Map/Reduce operations in Julia
3. A first illustration: mean computation
4. A second illustration: communication avoiding Cholesky-QR2 factorization
   - Context and goals
   - Problem statement
   - Implementation with Map/Reduce: analysis and results
   - Towards large-scale simulations with Cholesky-QR2
5. Conclusions

Context: dimensionality reduction

Context

- Given $A \in \mathbb{C}^{m \times n}$ with $p = (m, n)$ we seek to compute a rank-k approximation, typically with $k \ll p$ (say $m, n \sim 10^4, 10^6, 10^8, \cdots$ and $k \approx 10$ or 10^2) as

$$A \approx E\ F^H, \quad E \in \mathbb{C}^{m \times k}, \quad F \in \mathbb{C}^{n \times k}.$$

- Solving this problem usually requires algorithms for computing the **Singular Value Decomposition** (SVD), which is marginally parallel [Dongarra et al, 2018].
- **Goal**: implement in Julia a simple communication-minimizing factorization for tall and skinny matrices based on **map/reduce** strategies.

Singular Value Decomposition

SVD [Beltrami, 1873], [Jordan, 1874], [Sylvester, 1889], [Picard, 1910]

- Given $A \in \mathbb{C}^{m\times n}$ with $p = (m, n)$, the **full** singular value decomposition of A reads:

$$A = U\ \Sigma\ V^H,$$

with $U \in \mathbb{C}^{m\times m}$, $V \in \mathbb{C}^{n\times n}$ unitary ($U^H U = I_m$, $V^H V = I_n$) and $\Sigma \in \mathbb{R}^{n\times m}$.

- $\Sigma = diag(\sigma_1, \cdots, \sigma_p)$ with $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_p \geq 0$.
- $\sigma_i, (i = 1, p)$ are called **singular values** of A.

R-SVD

R-SVD [Chan, 1982]

- **Idea**: Perform an initial QR decomposition if the matrix is sufficiently tall relative to its width (i.e. $m \geq n$ with at least by a factor of 1.5): **tall and skinny** matrix.
- **First step**: QR factorization of $A \in \mathbb{C}^{m \times n}$ as $A = QR$ where $Q \in \mathbb{C}^{m \times n}$ has orthonormal columns ($Q^H Q = I_n$ and $R \in \mathbb{C}^{n \times n}$ is a triangular matrix).
- **Second step**: SVD decomposition of R as $R = U_R \Sigma_R V_R^H$.
- **Final step**: $A = U\ \Sigma_R\ V^H$ with $U = QU_R$ and $V = V_R$.
- **Complexity**: $4mn^2 + 22n^3$.
- **Parallel performance**: Tall and Skinny QR ($TSQR$) algorithm [Demmel et al, 2012] to be favored for the first step to obtain parallel performance [Benson et al, 2013]

Outline

1. Context
2. Map/Reduce operations in Julia
3. A first illustration: mean computation
4. A second illustration: communication avoiding Cholesky-QR2 factorization
   - Context and goals
   - Problem statement
   - Implementation with Map/Reduce: analysis and results
   - Towards large-scale simulations with Cholesky-QR2
5. Conclusions

Problem statement: Cholesky-QR factorization

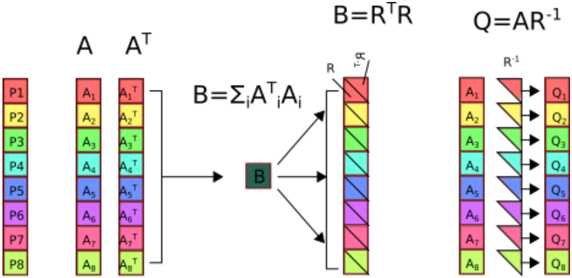
Cholesky-QR [Golub and Van Loan, 2012]

- $A \in \mathbb{R}^{m \times n}$ with $m >> n$ of full column rank.
- QR factorization of $A \in \mathbb{R}^{m \times n}$ as $A = QR$ where $Q \in \mathbb{R}^{m \times n}$ has orthonormal columns ($Q^T Q = I_n$ and $R \in \mathbb{R}^{n \times n}$ is a triangular matrix).
- **First step**: Compute the symmetric positive definite matrix $B = A^T A$.
- **Second step**: Perform the Cholesky factorization of the $n \times n$ matrix B as $B = R^T R$, where $R \in \mathbb{R}^{n \times n}$ is upper triangular. This step provides the R factor of the QR factorization.
- **Third step**: To deduce the Q factor, we simply have to solve:

$$R^T Q^T = A^T.$$

- Cholesky-QR is **not numerically stable**: deviation from orthogonality $\|Q^T\, Q - I_n\|_F$ is proportional to $\kappa_2(A)^2$.

Cholesky-QR2 factorization [Fukaya et al, 2014]



$Q, R \leftarrow$ **CholeskyQR**(A), figure from [Huetter et al, 2019]

- $\widetilde{Q}, R_1 \leftarrow$ **CholeskyQR**(A),
- $Q, R_2 \leftarrow$ **CholeskyQR**($\widetilde{Q}$),
- $R \leftarrow R_2 R_1$.
- Deviation from orthogonality $\|Q^T\, Q - I_n\|_F$ is $O(\varepsilon)$ if $\kappa_2(A) = O(\frac{1}{\sqrt{\varepsilon}})$. [Yamamoto et al, 2015]

Outline

1. Context

2. Map/Reduce operations in Julia

3. A first illustration: mean computation

4. A second illustration: communication avoiding Cholesky-QR2 factorization
   - Context and goals
   - Problem statement
   - Implementation with Map/Reduce: analysis and results
   - Towards large-scale simulations with Cholesky-QR2

5. Conclusions

Analysis of Cholesky-QR

- **Data decomposition**: partition the tall and skinny matrix $A \in \mathbb{R}^{m \times n}$ into ℓ panels $A_i \in \mathbb{R}^{m_i \times n} (1 \leq i \leq \ell)$ with $\sum_{i=1}^{\ell} m_i = m$.
- **Map**: the mapper function should perform the local matrix-matrix product $A_i^T A_i$.
- **Reduce**: the reducer function should combine the current result of the reducer (B) with a result of a mapper function related to panel j to update the contribution block B i.e. $B \leftarrow B + A_j^T A_j$.
- **Cholesky factorization of** $B \in \mathbb{R}^{n \times n}$ as $B = R^T R$: this can be performed straightforwardly without any map/reduce strategy.

Implementation of the mapper/reducer function (computation of the triangular factor)

```julia
using LinearAlgebra

function my_mapper_R(M)
     return (M'*M)
end

function my_reducer_R(P,Q)
     return (P+Q)
end
```

Analysis of Cholesky-QR: computation of the orthogonal factor

- **Data decomposition**: partition the tall and skinny matrix $A \in \mathbb{R}^{m \times n}$ into ℓ panels $A_i \in \mathbb{R}^{m_i \times n} (1 \leq i \leq \ell)$ with $\sum_{i=1}^{\ell} m_i = m$.
- **Map**: the mapper function performs the solution of the triangular system of equations $R^T Q_i^T = A_i^T$ and returns $Q_i \in \mathbb{R}^{m_i \times n}$.
- **Reduce**: the reducer function should combine the current result of the reducer with a result of a mapper function (say Q_r and Q_j) i.e. concatenate vertically $[Q_r; Q_j] \in \mathbb{R}^{(m_r + m_j) \times n}$.

Implementation of the mapper/reducer function (computation of the orthogonal factor)

```
using LinearAlgebra

function my_mapper_R(M)
     return (M'*M)
end

function my_mapper_Q(M)
     return ((Tfactor')\(M'))'
end

function my_reducer_R(P,Q)
     return (P+Q)
end

function my_reducer_Q(P,Q)
     return [P;Q]
end
```

Outline

1. Context
2. Map/Reduce operations in Julia
3. A first illustration: mean computation
4. A second illustration: communication avoiding Cholesky-QR2 factorization
    - Context and goals
    - Problem statement
    - Implementation with Map/Reduce: analysis and results
    - Towards large-scale simulations with Cholesky-QR2
5. Conclusions

Towards large-scale simulations with Cholesky-QR2

- Favor **BLAS** or **LAPACK** kernels for linear algebra operations: this can be performed with the BLAS package of Julia.
- Perform **inplace** operations to control allocations and memory management.
- Depending on your platform and installation of Julia, use **multithreading** in the numerical linear algebra libraries.
- Problems with $m \approx 10^6$ and $n \approx 64$ can be performed quite efficiently on a laptop [Notebook].
- Numerical results on **Olympe** (@CALMIP) to be discussed next.

Numerical experiments on Olympe

Olympe	Cholesky-QR2		
	$n = 64$		
m	$\|A - QR\|_F / \|A\|_F$	$\|Q^T Q - I_n\|_F / \sqrt{n}$	τ (seconds)
1024	$3.30\ 10^{-16}$	$1.45\ 10^{-15}$	0.002
16384	$3.11\ 10^{-16}$	$3.25\ 10^{-16}$	0.16
262144	$3.22\ 10^{-16}$	$3.21\ 10^{-16}$	2.6
1048576	$3.24\ 10^{-16}$	$8.61\ 10^{-16}$	11.6

- Experiments on **dense** rectangular random matrices performed on a single node of Olympe.
- This uses Julia 1.0.2 with **multithreaded OpenBLAS**.
- Use **distributed computing** through pmap for larger problem sizes !

Summary

Summary

- We have first discussed why **map and reduce** strategies are increasingly popular in high performance data analytics.
- We have then briefly reviewed **map and reduce operations** in Julia.
- We have provided two instructional **illustrations** in Julia.
- We have discussed numerical results related to a possible implementation of a **communication-minimizing** factorization method for dimensionality reduction.

What we have learnt about Julia

What we have learnt about Julia

- **Clear mathematical high-level syntax**: it is easy to express numerical algorithms leading to short codes.

What we have learnt about Julia

What we have learnt about Julia

- **Clear mathematical high-level syntax**: it is easy to express numerical algorithms leading to short codes.
- **Map/Reduce**: functional programming is great ! Other patterns (collect, filter) are available in Julia.

What we have learnt about Julia

What we have learnt about Julia

- **Clear mathematical high-level syntax**: it is easy to express numerical algorithms leading to short codes.
- **Map/Reduce**: functional programming is great ! Other patterns (collect, filter) are available in Julia.
- **Parallelism**: built-in primitives for parallel computing at multiple levels are available in Julia.

What we have learnt about Julia

What we have learnt about Julia

- **Clear mathematical high-level syntax**: it is easy to express numerical algorithms leading to short codes.
- **Map/Reduce**: functional programming is great ! Other patterns (collect, filter) are available in Julia.
- **Parallelism**: built-in primitives for parallel computing at multiple levels are available in Julia.

Thanks to CALMIP and Groupe Calcul @ CNRS.

Thank you for your attention !

References I

A. Benson, D. Gleich and J. Demmel.
Direct QR factorization for tall-and-skinny matrices in MapReduce architectures.
IEEE International Conference on Big Data, 2013.

T. Chan.
An improved algorithm for computing the Singular Value Decomposition.
ACM Trans. Math. Softw., 8:1, 84-88, 1982.

J. Demmel, L. Grigori, M. F. Hoemmen, and J. Langou.
Communication-optimal parallel and sequential QR and LU factorizations.
SIAM J. Sci. Comput., 34-1:206–A239, 2012.

References II

J. Dongarra, M. Gates, A. Haidar, J. Kurzack, P. Luszczeck, S. Tomov and I. Yamazaki.
The Singular Value Decomposition: anatomy of optimizing an algorithm for extreme scale.
SIAM Review, 60-4:808–865, 2018.

T. Fukaya, Y. Nakatsukasa, Y. Yanagisawa and Y. Yamamoto.
CholeskyQR2: A Simple and Communication-Avoiding Algorithm for Computing a Tall-Skinny QR Factorization on a Large-Scale Parallel System.
5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, New Orleans, LA, 2014.

A. Gittens, A. Devarakonda, E. Racah, M. Ringenburg, L. Gerhardt, J. Kottalam, J. Liu, K. Maschhoff, S. Canon, J. Chhugani, P. Sharma, J. Yang, J. Demmel, J. Harrell, V. Krishnamurthy and M. Mahoney.
Matrix factorizations at scale: A comparison of scientific data analytics in Spark and C+ MPI using three case studies.
IEEE International Conference on Big Data, pp. 204-213, 2016.

Y. Yamamoto, Y. Nakatsukasa, Y. Yanagisawa and T. Fukaya.
Roundoff error analysis of the CholeskyQR2 algorithm.
Electronic Transactions on Numerical Analysis, 44:306-326, 2015.

References IV

J. Regier, K. Pamnany, K. Fischer, A. Noack, M. Lam, J. Revels, S. Howard, R. Giordano, D. Schlegel, J. McAuliffe, R. Thomas, and Prabhat.
Cataloging the visible universe through Bayesian inference at petascale.
International Parallel and Distributed Processing Symposium (IPDPS), 2018.
https://github.com/jeff-regier/Celeste.jl