DataFlowTasks.jl

Julia `Tasks` which automatically handle data-dependencies

Luiz M. Faria[1] François Févotte[2]

[1]Research scientist
INRIA

[2]Chief Scientist
TriScale innov

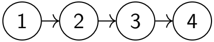
JuliaCon Local Eindhoven
December 1, 2023

Overview

- `DataFlowTasks.jl` is a Julia package dedicated to parallel programming on **multi-core shared memory CPUs**.
- Automatically infer `Task` interdependencies based on user annotations (`@R`, `@W`, `@RW`).
- Inspired by task programming libraries such as StarPU / PaRSEC
  - (in spirit, not in ambition)
- Simple API: `@dspawn` macro.

```julia
function foo!(A, b)
  fill!(b, 0)
  view(b, 1:2) .+= 2
  view(b, 3:4) .+= 3
  A \ b
end
```

```julia
function foo!(A, b)
  @dspawn fill!(@W(b), 0)             # task 1
  @dspawn @RW(view(b, 1:2)) .+= 2 # task 2
  @dspawn @RW(view(b, 3:4)) .+= 3 # task 3
  @dspawn @R(A) \ @R(b)             # task 4
end
```



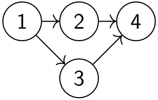


Table of Contents

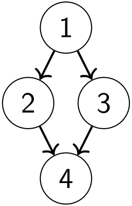
Table of Contents

Task based parallelism

- A task is a unit of execution or a unit of work
- `Task` objects can be created using `@task`
- Once created, `Task` objects must be scheduled for execution
- Usually, `Task`s are created + scheduled using `@spawn`
- Responsibility of synchronizing tasks is left to the programmer

Example (Sequential)

```
                        # Task
b = zeros(4)               # 1 - Initialization
b[1:2] .+= 2               # 2 - Work on first half
b[3:4] .+= 3               # 3 - Work on second half
res = A \ b                # 4 - Use entire vector
                           #
```

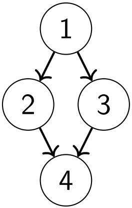


Task based parallelism

- A task is a unit of execution or a unit of work
- `Task` objects can be created using `@task`
- Once created, `Task` objects must be scheduled for execution
- Usually, `Task`s are created + scheduled using `@spawn`
- Responsibility of synchronizing tasks is left to the programmer

Example (Synchronizing tasks with return values)

```julia
                                            # Task
t1 = @spawn zeros(4)                        # 1
t2 = @spawn (b = fetch(t1); b[1:2] .+ 2)    # 2
t3 = @spawn (b = fetch(t1); b[3:4] .+ 3)    # 3
t4 = @spawn A \ vcat(fetch(t2), fetch(t3))  # 4
fetch(t4) # get result
```

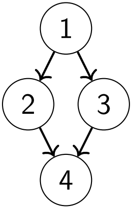


Task based parallelism

- A task is a unit of execution or a unit of work
- `Task` objects can be created using `@task`
- Once created, `Task` objects must be scheduled for execution
- Usually, `Task`s are created + scheduled using `@spawn`
- Responsibility of synchronizing tasks is left to the programmer

Example (Synchronizing tasks with explicit barriers)

```
b = Vector{Float64}(undef, 4)                            # Task
t1 = @spawn fill!(b, 0)                                  # 1
t2 = @spawn (wait(t1); view(b,1:2) += 2)                 # 2
t3 = @spawn (wait(t1); view(b,3:4) += 3)                 # 3
t4 = @spawn (wait(t2); wait(t3);  A \ b)                 # 4
fetch(t4) # get result
```



Motivation for DataFlowTasks

- Reasoning about Task interdependencies can be challenging
- Specially for algorithms making constant re-use of data
- Sometimes, it is simpler to reason about how Tasks depend on data than how Tasks depend on each other

Example (Synchronizing tasks)

```
b = rand(4)
t1 = @spawn fill!(b,0)                        # RW access to b
t2 = @spawn (wait(t1); view(b,1:2) += 2) # RW access to b[1:2]
t3 = @spawn (wait(t1); view(b,3:4) += 3) # RW access to b[3:4]
t4 = @spawn (wait(t2); wait(t3);  A \ b) # R  access to A and b
fetch(t4)
```

Motivation for DataFlowTasks

- Reasoning about `Task` interdependencies can be challenging
- Specially for algorithms making constant re-use of data
- Sometimes, it is simpler to reason about how `Task`s depend on data than how `Task`s depend on each other

Example (Synchronizing tasks)

```
b = rand(4)
t1 = @spawn fill!(b,0)                         # RW access to b
t2 = @spawn (wait(t1); view(b,1:2) += 2) # RW access to b[1:2]
t3 = @spawn (wait(t1); view(b,3:4) += 3) # RW access to b[3:4]
t4 = @spawn (wait(t2); wait(t3);  A \ b) # R  access to A and b
fetch(t4)
```

- Would like to declare only the task-to-data dependencies

Basic idea

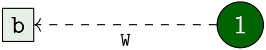
1. Extract data dependency from user annotations
2. Infer task dependency from data dependency
3. Schedule tasks with inferred dependencies

User-written code

```julia
b = rand(4)
```

Basic idea

1. Extract data dependency from user annotations
2. Infer task dependency from data dependency
3. Schedule tasks with inferred dependencies

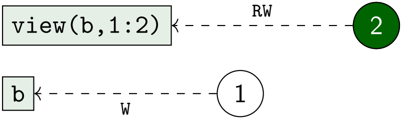


User-written code

```
b = rand(4)
@dspawn fill!(@W(b),0)
```

Basic idea

1. Extract data dependency from user annotations
2. Infer task dependency from data dependency
3. Schedule tasks with inferred dependencies

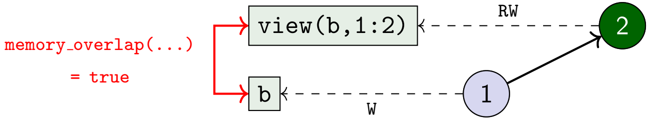


User-written code

```
b = rand(4)
@dspawn fill!(@W(b),0)
@dspawn @RW(view(b,1:2)) += 2
```

Basic idea

1. Extract data dependency from user annotations
2. Infer task dependency from data dependency
3. Schedule tasks with inferred dependencies



User-written code

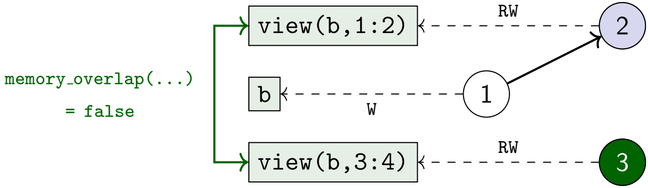
```
b = rand(4)
@dspawn fill!(@W(b),0)
@dspawn @RW(view(b,1:2)) += 2
```

Behind the scenes: DAG

```
# Note the quadratic complexity!
for i in 1:N, j in i:-1:1
  # Detect conflict between i and j
  for di in data(i), dj in data(j)
    if memory_overlap(di, dj)
      add_edge(j, i)
```

Basic idea

1. Extract data dependency from user annotations
2. Infer task dependency from data dependency
3. Schedule tasks with inferred dependencies



User-written code

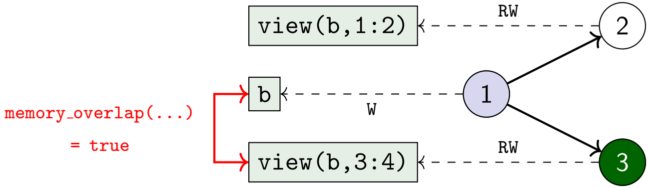
```
b = rand(4)
@dspawn fill!(@W(b),0)
@dspawn @RW(view(b,1:2)) .+= 2
@dspawn @RW(view(b,3:4)) .+= 3
```

Behind the scenes: DAG

```
# Note the quadratic complexity!
for i in 1:N, j in i:-1:1
  # Detect conflict between i and j
  for di in data(i), dj in data(j)
    if memory_overlap(di, dj)
      add_edge(j, i)
```

Basic idea

1. Extract data dependency from user annotations
2. Infer task dependency from data dependency
3. Schedule tasks with inferred dependencies



User-written code

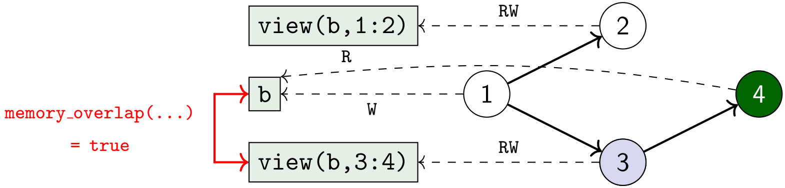
```
b = rand(4)
@dspawn fill!(@W(b),0)
@dspawn @RW(view(b,1:2)) += 2
@dspawn @RW(view(b,3:4)) += 3
```

Behind the scenes: DAG

```
# Note the quadratic complexity!
for i in 1:N, j in i:-1:1
  # Detect conflict between i and j
  for di in data(i), dj in data(j)
    if memory_overlap(di, dj)
      add_edge(j, i)
```

Basic idea

1. Extract data dependency from user annotations
2. Infer task dependency from data dependency
3. Schedule tasks with inferred dependencies



User-written code

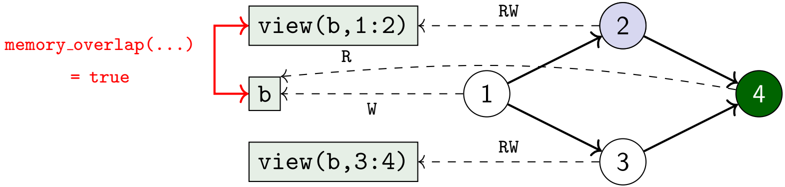
```
b = rand(4)
@dspawn fill!(@W(b),0)
@dspawn @RW(view(b,1:2)) += 2
@dspawn @RW(view(b,3:4)) += 3
t4 = @dspawn A \ @R(b)
fetch(t4)
```

Behind the scenes: DAG

```
# Note the quadratic complexity!
for i in 1:N, j in i:-1:1
  # Detect conflict between i and j
  for di in data(i), dj in data(j)
    if memory_overlap(di, dj)
      add_edge(j, i)
```

Basic idea

1. Extract data dependency from user annotations
2. Infer task dependency from data dependency
3. Schedule tasks with inferred dependencies



User-written code

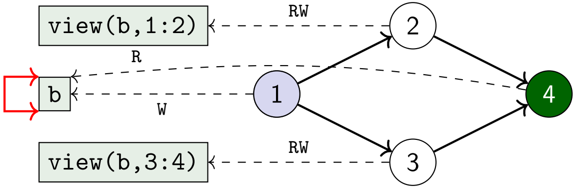
```
b = rand(4)
@dspawn fill!(@W(b),0)
@dspawn @RW(view(b,1:2)) .+= 2
@dspawn @RW(view(b,3:4)) .+= 3
t4 = @dspawn A \ @R(b)
fetch(t4)
```

Behind the scenes: DAG

```
# Note the quadratic complexity!
for i in 1:N, j in i:-1:1
  # Detect conflict between i and j
  for di in data(i), dj in data(j)
    if memory_overlap(di, dj)
      add_edge(j, i)
```

Basic idea

1. Extract data dependency from user annotations
2. Infer task dependency from data dependency
3. Schedule tasks with inferred dependencies



User-written code

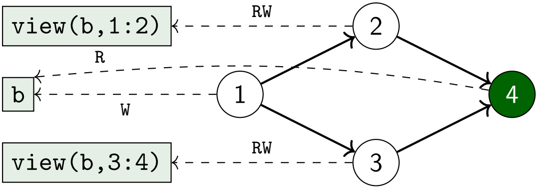
```
b = rand(4)
@dspawn fill!(@W(b),0)
@dspawn @RW(view(b,1:2)) += 2
@dspawn @RW(view(b,3:4)) += 3
t4 = @dspawn A \ @R(b)
fetch(t4)
```

Behind the scenes: DAG

```
# Note the quadratic complexity!
for i in 1:N, j in i:-1:1
  # Detect conflict between i and j
  for di in data(i), dj in data(j)
    if memory_overlap(di, dj)
      add_edge(j, i)
```

Basic idea

1. Extract data dependency from user annotations
2. Infer task dependency from data dependency
3. Schedule tasks with inferred dependencies



User-written code

```
b = rand(4)
@dspawn fill!(@W(b),0)
@dspawn @RW(view(b,1:2)) += 2
@dspawn @RW(view(b,3:4)) += 3
t4 = @dspawn A \ @R(b)
fetch(t4)
```

Behind the scenes: task scheduling

```
t4 = Threads.@spawn begin
  wait(t2); wait(t3)
  A \ b
end
```

Table of Contents

Tiled Cholesky factorization: idea

- Objective: take a SPD matrix A, find lower triangular L such that

$$A = LL^\top$$

- Idea for a tiled algorithm:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{12}^\top & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^\top & L_{21}^\top \\ & L_{22}^\top \end{bmatrix}$$

Tiled Cholesky factorization: idea

- Objective: take a SPD matrix A, find lower triangular L such that

$$A = LL^\top$$

- Idea for a tiled algorithm:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{12}^\top & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^\top & L_{21}^\top \\ & L_{22}^\top \end{bmatrix}$$

1. $A_{11} = L_{11} L_{11}^\top$
   ▷ Cholesky factorization to get L_{11} `cholesky!`

Tiled Cholesky factorization: idea

- Objective: take a SPD matrix A, find lower triangular L such that

$$A = LL^\top$$

- Idea for a tiled algorithm:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{12}^\top & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^\top & L_{21}^\top \\ & L_{22}^\top \end{bmatrix}$$

1. $A_{11} = L_{11} L_{11}^\top$
   $\triangleright$ Cholesky factorization to get L_{11} `cholesky!`
2. $A_{12} = L_{11} L_{21}^\top$
   $\triangleright$ Triangular solve to get L_{21} `ldiv!`

Tiled Cholesky factorization: idea

- Objective: take a SPD matrix A, find lower triangular L such that

$$A = LL^\top$$

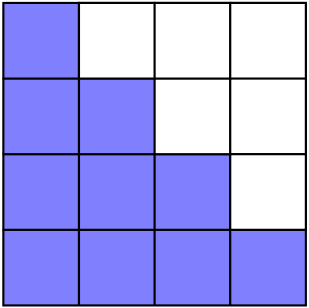
- Idea for a tiled algorithm:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{12}^\top & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^\top & L_{21}^\top \\ & L_{22}^\top \end{bmatrix}$$

1. $A_{11} = L_{11} L_{11}^\top$
   ▷ Cholesky factorization to get L_{11} `cholesky!`
2. $A_{12} = L_{11} L_{21}^\top$
   ▷ Triangular solve to get L_{21} `ldiv!`
3. $A_{22} - L_{21} L_{21}^\top = L_{22} L_{22}^\top$
   ▷ Matmul to get $A_{22} - L_{21} L_{21}^\top$ `mul!`
   ▷ Cholesky factorization to get L_{22} `cholesky!`

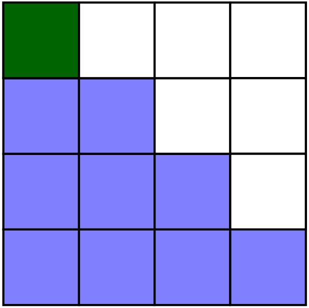
Tiled Cholesky factorization: algorithm

- Input: symmetric positive-definite matrix A with $N \times N$ blocks
- Goal: compute lower-triangular L such that $A = LL^\top$



Tiled Cholesky factorization: algorithm

- Input: symmetric positive-definite matrix A with $N \times N$ blocks
- Goal: compute lower-triangular L such that $A = LL^\top$



i=1

```
for i in 1:N
```

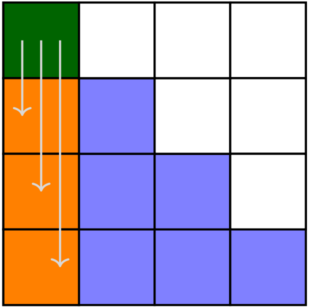
1. cholesky!

   $A_{i,i} \leftarrow \texttt{chol!}(A_{i,i})$

   $L_{i,i} = \texttt{LowerTriangular}(A_{i,i})$

Tiled Cholesky factorization: algorithm

- Input: symmetric positive-definite matrix A with $N \times N$ blocks
- Goal: compute lower-triangular L such that $A = LL^\top$



i=1

for i in 1:N

1. cholesky!

   $A_{i,i} \leftarrow \texttt{chol!}(A_{i,i})$

   $L_{i,i} = \texttt{LowerTriangular}(A_{i,i})$

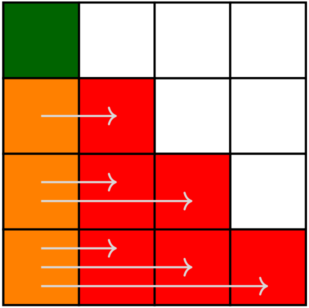
2. ldiv!

   for k in 2:N

   $A_{k,i} \leftarrow \texttt{ldiv!}(L_{i,i}, A_{k,i}),$

Tiled Cholesky factorization: algorithm

- Input: symmetric positive-definite matrix A with $N \times N$ blocks
- Goal: compute lower-triangular L such that $A = LL^\top$



i=1

for i in 1:N

1. cholesky!

   $A_{i,i} \leftarrow \texttt{chol!}(A_{i,i})$

   $L_{i,i} = \texttt{LowerTriangular}(A_{i,i})$

2. ldiv!

   for k in 2:N

   $A_{k,i} \leftarrow \texttt{ldiv!}(L_{i,i}, A_{k,i})$,

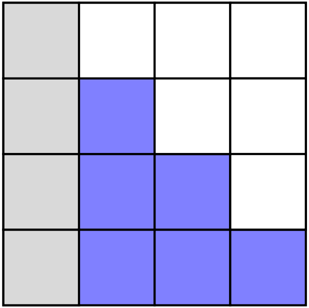
3. mul!

   for k in i+1:N, j in i+1:k

   $A_{k,j} \leftarrow A_{k,j} - A_{k,i}A_{k,i}^\top$

Tiled Cholesky factorization: algorithm

- Input: symmetric positive-definite matrix A with $N \times N$ blocks
- Goal: compute lower-triangular L such that $A = LL^\top$



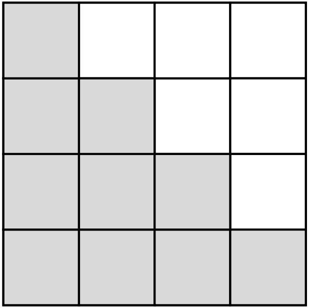
i=2, repeat

for i in 1:N

1. cholesky!
   $A_{i,i} \leftarrow \texttt{chol!}(A_{i,i})$
   $L_{i,i} = \texttt{LowerTriangular}(A_{i,i})$
2. ldiv!
   for k in 2:N
   $A_{k,i} \leftarrow \texttt{ldiv!}(L_{i,i}, A_{k,i})$,
3. mul!
   for k in i+1:N, j in i+1:k
   $A_{k,j} \leftarrow A_{k,j} - A_{k,i} A_{k,i}^\top$

Tiled Cholesky factorization: algorithm

- Input: symmetric positive-definite matrix A with $N \times N$ blocks
- Goal: compute lower-triangular L such that $A = LL^\top$



for i in 1:N

1. cholesky!

   $A_{i,i} \leftarrow \texttt{chol!}(A_{i,i})$

   $L_{i,i} = \texttt{LowerTriangular}(A_{i,i})$

2. ldiv!

   for k in 2:N

   $A_{k,i} \leftarrow \texttt{ldiv!}(L_{i,i}, A_{k,i}),$

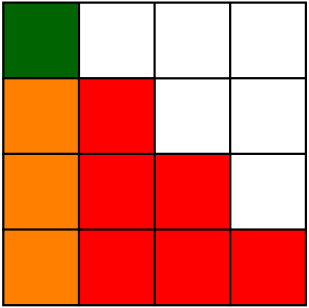
3. mul!

   for k in i+1:N, j in i+1:k

   $A_{k,j} \leftarrow A_{k,j} - A_{k,i}A_{k,i}^\top$

- Output: L in the lower-triangular part of A

Tiled Cholesky factorization: complexity



For an $N \times N$ block matrix:

- $\mathcal{O}(N)$ cholesky factorizations
- $\mathcal{O}(N^2)$ triangular solves
- $\mathcal{O}(N^3)$ matrix multiplications

Globally compute-bound:

- $\mathcal{O}(N^3)$ flops for
- $\mathcal{O}(N^2)$ bytes

$\Rightarrow$ potential for efficient parallelism

Tiled Cholesky factorization: sequential implementation

```julia
function cholesky_tiled!(A, ts)
  m = size(A, 1); n = m ÷ ts      # Matrix size & number of tiles
  T = [view(A, tilerange(i, ts), tilerange(j, ts)) for i in 1:n, j in 1:n]

  for i in 1:n
    cholesky!(T[i,i])             # Diagonal cholesky serial factorization

    U = UpperTriangular(T[i,i])   # Left tiles update
    for j in i+1:n
      ldiv!(U', T[i,j])
    end

    for j in i+1:n, k in j:n      # Submatrix update
      mul!(T[j,k], T[i,j]', T[i,k], -1, 1)
    end
  end

  # Construct the factorized object
  return Cholesky(A, 'U', zero(LinearAlgebra.BlasInt))
end
```

Tiled Cholesky factorization: parallel implementation

```
function cholesky_dft!(A, ts)
  m = size(A, 1); n = m ÷ ts       # Matrix size & number of tiles
  T = [view(A, tilerange(i, ts), tilerange(j, ts)) for i in 1:n, j in 1:n]

  for i in 1:n
    @dspawn cholesky!(@RW(T[i,i]))    label="chol ($i,$i)"

    U = UpperTriangular(T[i,i])
    for j in i+1:n
      @dspawn ldiv!(@R(U)', @RW(T[i,j]))    label="ldiv ($i,$j)"
    end

    for j in i+1:n, k in j:n
      @dspawn mul!(@RW(T[j,k]),@R(T[i,j])',@R(T[i,k]),-1, 1) label="schur($j,$k)"
    end
  end

  # Construct the factorized object
  r = @dspawn Cholesky(@R(A), 'U', zero(LinearAlgebra.BlasInt)) label="result"
  return fetch(r)
end
```

Tiled Cholesky factorization: performance

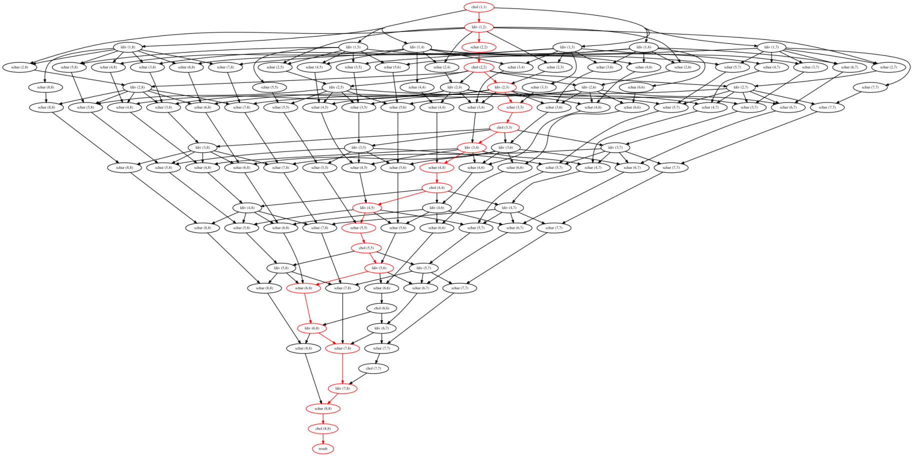
```
julia> using BenchmarkTools

julia> n = 4096; ts = 512;

julia> t_seq = @belapsed(cholesky_tiled!(A, ts),
                         setup=(A=spd_matrix(n)), evals=1)
0.559141485

julia> t_par = @belapsed(cholesky_dft!(A, ts),
                         setup=(A=spd_matrix(n)), evals=1)
0.108479304

julia> (; threads = Threads.nthreads(),
          speedup = t_seq / t_par)
(threads = 8, speedup = 5.15436089063056)
```

Tiled Cholesky factorization: profiling & debugging

```
julia> n = 4096; ts = 512;

julia> A = spd_matrix(n);

julia> log_info = DataFlowTasks.@log cholesky_dft!(Ac, ts)
LogInfo with 121 logged tasks

julia> DataFlowTasks.describe(log_info;categories=["chol","ldiv","schur"])
• Elapsed time           : 0.297
  +  Critical Path       : 0.253
  +  No-Wait             : 0.188

• Run time               : 2.379
  +  Computing           :   1.507
     o  chol             :     0.061
     o  ldiv             :     0.414
     o  schur            :     1.032
     o  unlabeled        :     0.000
  +  Task Insertion      :   0.003
  +  Other (idle)        :   0.870
```

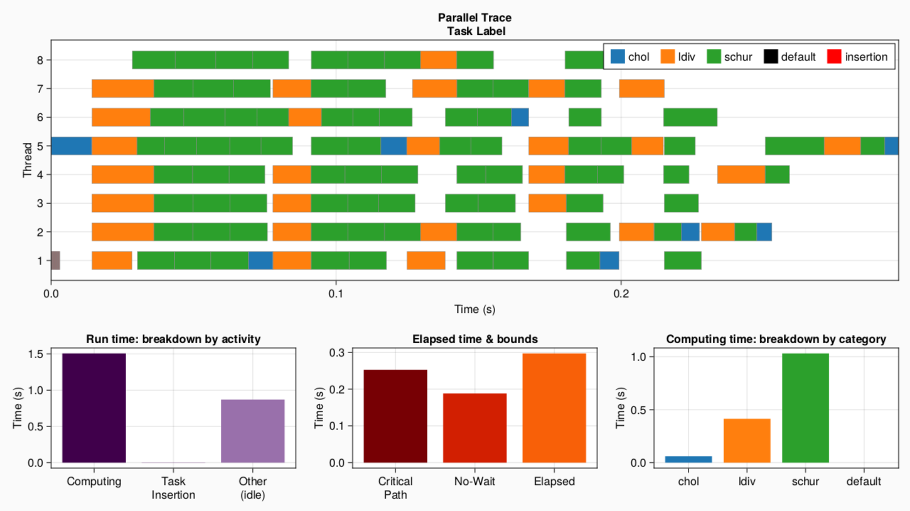
Tiled Cholesky factorization: profiling & debugging

```
julia> DataFlowTasks.stack_weakdeps_env!()
julia> using GraphViz
[ Info: Loading DataFlowTasks dag plot utilities
julia> GraphViz.Graph(log_info)
```



Tiled Cholesky factorization: profiling & debugging

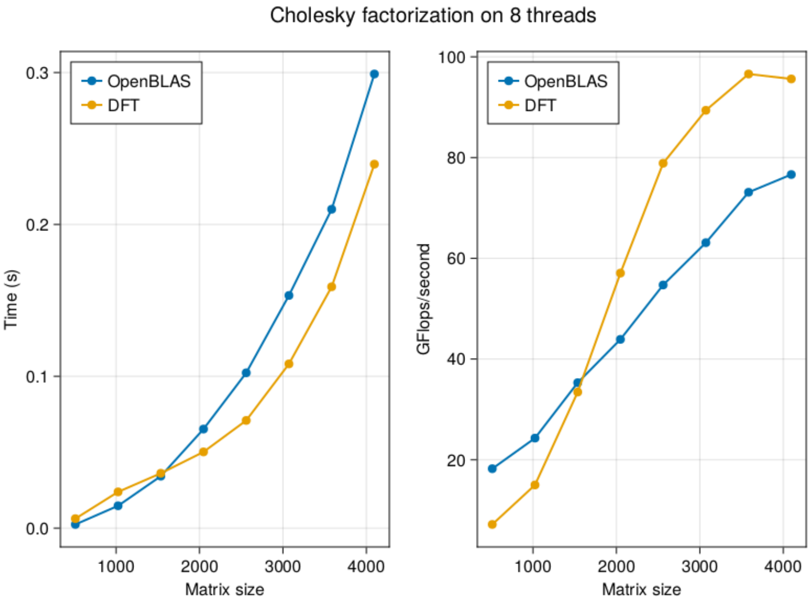
```
julia> using CairoMakie # or GLMakie for more interactivity
[ Info: Loading DataFlowTasks profile plot utilities
julia> plot(log_info; categories=["chol", "ldiv", "schur"])
```



Comparison to openBLAS

How does it compare to multi-threaded OpenBLAS?

Tiled Cholesky factorization: performance



Tiled Cholesky factorization: performance

Table of Contents

Concluding remarks

- `DataFlowTasks.jl` tries to make parallel programming easier for *certain types of algorithms*
- Simple API: `@dspawn` macro
- Several supplementary examples in the online documentation

`https://github.com/maltezfaria/DataFlowTasks.jl`

If you think `DataFlowTasks.jl` may help parallelize your algorithm, please give it a try or reach out to us!

Appendix

Table of Contents

Main types

The package revolves around three main types:

- `DataFlowTask`: wrapper around a `Task` with user-declared data dependencies

Main types

The package revolves around three main types:

- `DataFlowTask`: wrapper around a `Task` with user-declared data dependencies
- `DAG`: graph data structure to represent dependencies between `DataFlowTasks`

Main types

The package revolves around three main types:

- `DataFlowTask`: wrapper around a `Task` with user-declared data dependencies
- `DAG`: graph data structure to represent dependencies between `DataFlowTasks`
- `TaskGraph`: a `DAG` and some helper functions/workers to manage it

Main types

The package revolves around three main types:

- `DataFlowTask`: wrapper around a `Task` with user-declared data dependencies
- `DAG`: graph data structure to represent dependencies between `DataFlowTasks`
- `TaskGraph`: a `DAG` and some helper functions/workers to manage it

Next: some technical details about their implementation.

DataFlowTask: @dspawn macro

DataFlowTasks are created using @dspawn. The macro does the following:

- Scan the Expr for @R, @W, @RW annotations
- Create a data and mode tuples
- Remove annotations from the Expr
- Parse keyword arguments
- Create an anonymous function wrapping the new Expr
- Insert a call to DataFlowTask constructor

User-written code

```
@dspawn(foo!(@W(B), @R(A)),
        label="foo!")
```

Macro expansion (approx.)

```
_t = DataFlowTask(
        () -> foo!(B, A),
        (B, A),
        (WRITE, READ);
        label="foo!")
spawn(_t)
```

DataFlowTask structure

Inner-constructor of DataFlowTask handles much of the insertion/removal logic:

```julia
mutable struct DataFlowTask
  data::Tuple
  access_mode::NTuple{<:Any,AccessMode}
  task::Task
  ...
  function DataFlowTask(f,data,mode,taskgraph)
    tj = new(data, mode) # incomplete initialization
    addnode!(taskgraph, tj, true)
    deps = inneighbors(taskgraph, tj) |> copy
    tj.task = @task do
      foreach(wait,deps)
      res = f() # run the underlying function
      put!(taskgraph.finished, tj)
      return res
    end
  end
end
```

DAG

Graph structure used to represent dependencies between `DataFlowTasks`:

- Dynamic: nodes are added and removed on the fly
- Buffered: limit the number of active nodes
- Thread-safe: multiple threads can add/remove nodes
- Efficient: easy access to both in- and out-neighbors

```julia
struct DAG{T}
  inoutlist::OrderedDict{...}
  cond_push::Condition
  lock::ReentrantLock
  sz_max::Base.RefValue{Int}
  ...
end
```

DAG

Graph structure used to represent dependencies between `DataFlowTasks`:

- Dynamic: nodes are added and removed on the fly
- **Buffered: limit the number of active nodes**
- Thread-safe: multiple threads can add/remove nodes
- Efficient: easy access to both in- and out-neighbors

```julia
struct DAG{T}
  inoutlist::OrderedDict{...}
  cond_push::Condition
  lock::ReentrantLock
  sz_max::Base.RefValue{Int}
  ...
end
```

- ▷ `addnode!(dag,node)` calls `wait(cond_push)` if full
- ▷ `removenode!(dag,node)` calls `notify(cond_push)`

DAG

Graph structure used to represent dependencies between `DataFlowTasks`:

- Dynamic: nodes are added and removed on the fly
- Buffered: limit the number of active nodes
- **Thread-safe: multiple threads can add/remove nodes**
- Efficient: easy access to both in- and out-neighbors

```julia
struct DAG{T}
  inoutlist::OrderedDict{...}
  cond_push::Condition
  lock::ReentrantLock
  sz_max::Base.RefValue{Int}
  ...
end
```

- ▷ mutating the `DAG` requires acquiring/releasing the `lock`
- ▷ pattern: `@lock dag.lock code`

Taskgraph

Essentially a `DAG{DataFlowTask}` with

- A channel to store finished tasks
- A dedicated `Task` to remove nodes from the graph

```julia
mutable struct TaskGraph
  dag::DAG{DataFlowTask}
  finished::FinishedChannel
  dag_cleaner::Task
  function TaskGraph(sz)
    dag = DAG{DataFlowTask}(sz)
    finished = FinishedChannel()
    tg = new(dag, finished)
    start_dag_cleaner(tg)
    return tg
  end
end
```

- ▷ Insertion done by `DataFlowTask` constructor
- ▷ Removal done in two steps:
  1. The `node.task` moves the node into the finished channel
  2. Dedicated task handles finished channel

Logging

Some logging capabilities available:

- `@log` macro logs the execution of block
- `describe(loginfo)` shows a summary
- `Graph(loginfo)` displays the `DAG`
- `plot(loginfo)` plots the execution trace

Logging

Basic idea:

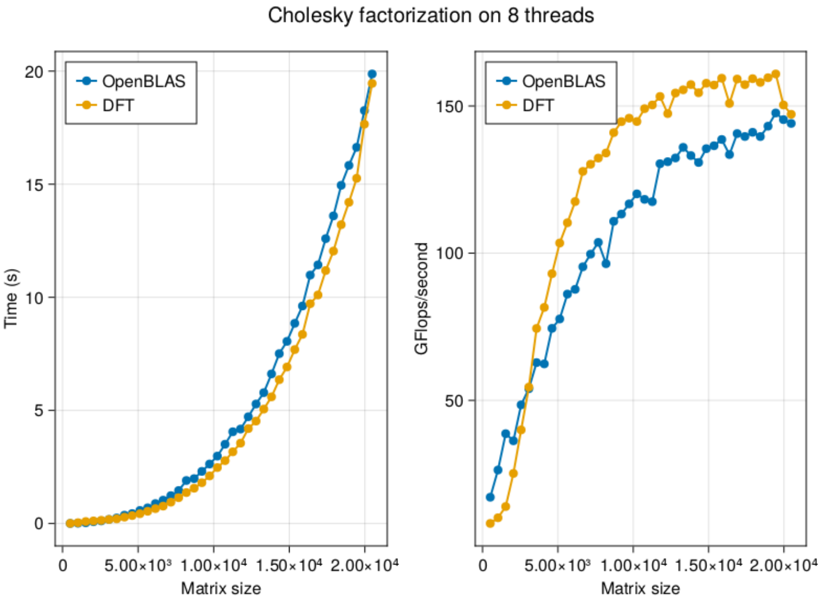
- Redefine the function `should_log()` to control logging
- Tasks conditionally create a `TaskLog` object
- Information dumped into `LogInfo` object
- Logging should have zero overhead when disabled

```julia
struct TaskLog
  tag::Int
  time_start::UInt64
  time_finish::UInt64
  tid::Int
  inneighbors::Vector{Int64}
  label::String
end
```

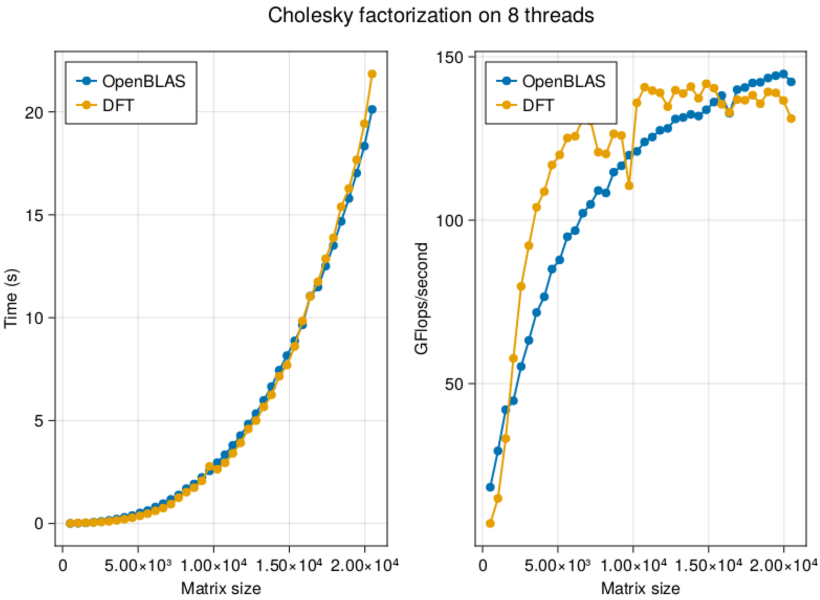
Known limitations:

- If the function `yields`, logged task time is not representative of execution time

Tiled Cholesky factorization: performance



Tiled Cholesky factorization: performance



Tiled Cholesky factorization: performance