DataFlowTasks.jl

Julia `Tasks` which automatically handle data-dependencies

Luiz M. Faria[1] François Févotte[2]

[1]Chargé de recherche INRIA
POEMS Laboratory

[2]Chief Scientist
TriScale innov

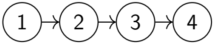
November 28, 2023

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`).
- Heavily inspired by task programming libraries such as StarPU.
- Simple API: `@dspawn` macro.

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

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



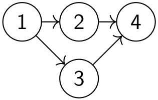


Table of Contents

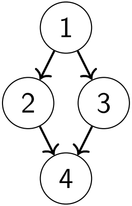
Table of Contents

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
A = zeros(4)       # 1 - Initialization
A[1:2] .+= 2       # 2 - Work on first half
A[3:4] .+= 3       # 3 - Work on second half
res = sum(A)       # 4 - Reduction
                   #
```

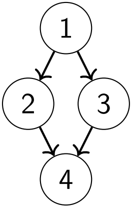


Task based parallelism

- 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)

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

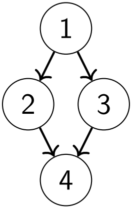


Task based parallelism

- 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)

```
A = rand(4)                                          # Task
t1 = @spawn fill!(A,0)                               # 1
t2 = @spawn (wait(t1); view(A,1:2) += 2)             # 2
t3 = @spawn (wait(t1); view(A,3:4) += 3)             # 3
t4 = @spawn (wait(t2); wait(t3); sum(A))             # 4
fetch(t4) # get result
```



Motivation

- 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)

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

Motivation

- 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)

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

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

Basic idea

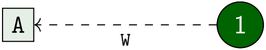
1. Extract data dependency from user annotations
2. Infer task dependency from data dependency
3. Schedule tasks with inferred dependencies

User-written code

```
A = rand(4)
```

Basic idea

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

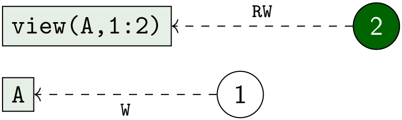


User-written code

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

Basic idea

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

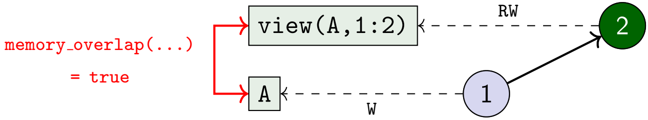


User-written code

```
A = rand(4)
@dspawn fill!(@W(A),0)
@dspawn @RW(view(A,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

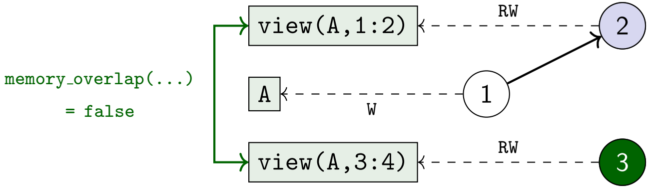
```
A = rand(4)
@dspawn fill!(@W(A),0)
@dspawn @RW(view(A,1:2)) += 2
```

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

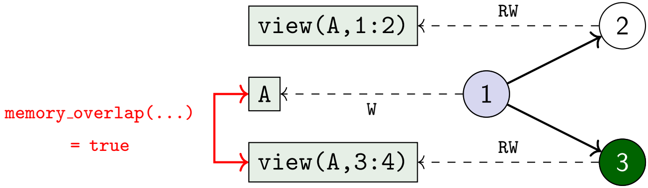
```
A = rand(4)
@dspawn fill!(@W(A),0)
@dspawn @RW(view(A,1:2)) += 2
@dspawn @RW(view(A,3:4)) += 3
```

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

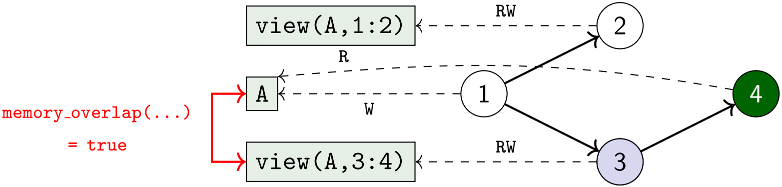
```
A = rand(4)
@dspawn fill!(@W(A),0)
@dspawn @RW(view(A,1:2)) += 2
@dspawn @RW(view(A,3:4)) += 3
```

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

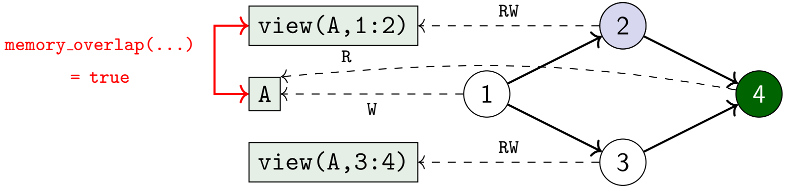
```
A = rand(4)
@dspawn fill!(@W(A),0)
@dspawn @RW(view(A,1:2)) += 2
@dspawn @RW(view(A,3:4)) += 3
t4 = @dspawn sum(@R(A))
fetch(t4)
```

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

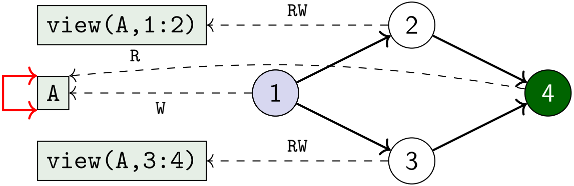
```
A = rand(4)
@dspawn fill!(@W(A),0)
@dspawn @RW(view(A,1:2)) += 2
@dspawn @RW(view(A,3:4)) += 3
t4 = @dspawn sum(@R(A))
fetch(t4)
```

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

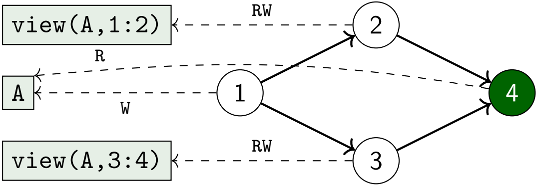
```
A = rand(4)
@dspawn fill!(@W(A),0)
@dspawn @RW(view(A,1:2)) += 2
@dspawn @RW(view(A,3:4)) += 3
t4 = @dspawn sum(@R(A))
fetch(t4)
```

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

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

Behind the scenes: task scheduling

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

Demo: parallel merge sort

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:

```
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

```
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

```
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

```
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

```
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

Table of Contents

Live examples

Some example use cases of `DataFlowTasks.jl`:

- Tiled cholesky factorization
- Blur-Roberts filter
- Longest common subsequence
- Merge sort

Live examples

Some example use cases of `DataFlowTasks.jl`:

- **Tiled cholesky factorization**
- Blur-Roberts filter
- Longest common subsequence
- Merge sort

Tiled Cholesky decomposition

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{12}^{T} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^{\top} & L_{21}^{\top} \\ & L\top_{22} \end{bmatrix}$$

1. $A_{11} = L_{11} L_{11}^{\top}$ (Cholesky factorization)
2. $A_{12} = L_{11} L_{21}^{\top}$ (Triangular solve)
3. $A_{22} - L_{21} L_{21}^{\top} = L_{22} L_{22}^{\top}$ (multiplication and Cholesky factorization)

Table of Contents

Roadmap

- Priority scheduling
- Nesting `DataFlowTasks`
- ...