# Package 'makeParallel'

July 31, 2018

**Version** 0.1.1

**Date** 2018-07-18

**Title** Transform Serial R Code into Parallel R Code

**Maintainer** Clark Fitzgerald <clarkfitzg@gmail.com>

**Depends** R (>= 3.1.0)

**Imports** methods, utils, graphics, parallel, codetools, CodeDepends, whisker

**Suggests** igraph, roxygen2, knitr, rmarkdown, testthat

**Description** Writing parallel R code can be difficult, particularly for code that is not ``embarrassingly parallel''.
This experimental package automates the transformation of serial R code into more efficient parallel versions. It identifies task parallelism by statically analyzing entire scripts to detect dependencies between statements. It implements an extensible system for scheduling and generating new code. It includes a reference implementation of the 'List Scheduling' approach to the general task scheduling problem of scheduling statements on multiple processors.

**License** MIT + file LICENSE

**URL** https://github.com/clarkfitzg/makeParallel

**BugReports** https://github.com/clarkfitzg/makeParallel

**RoxygenNote** 6.0.1

**VignetteBuilder** knitr

**NeedsCompilation** no

**Author** Clark Fitzgerald [aut, cre] (<https://orcid.org/0000-0003-3446-6389>)

**Repository** CRAN

**Date/Publication** 2018-07-31 10:30:03 UTC

# R topics documented:

DependGraph-class . . . 2
file,DependGraph-method . . . 3
file<- . . . 3
forLoopToLapply . . . 4
generate . . . 4
GeneratedCode-class . . . 5
inferGraph . . . 5
makeParallel . . . 6
mapSchedule . . . 8
MapSchedule-class . . . 9
MeasuredDependGraph-class . . . 9
plot,TaskSchedule,missing-method . . . 10
runMeasure . . . 11
schedule . . . 11
Schedule-class . . . 12
scheduleTaskList . . . 12
SerialSchedule-class . . . 13
TaskSchedule-class . . . 14
use_def . . . 14
writeCode . . . 15

**Index** **16**

---

`DependGraph-class` *Dependency graph between expressions*

---

## Description

Dependency graph between expressions

## Slots

`code` input code

`graph` data frame representing the graph with indices corresponding to code

## file,DependGraph-method *Get File containing code*

### Description

Get File containing code

### Usage

```
## S4 method for signature 'DependGraph'
file(description)

## S4 method for signature 'Schedule'
file(description)

## S4 method for signature 'GeneratedCode'
file(description)
```

### Arguments

| | |
|---|---|
| description | object that may have a file associated with it |

## file<- *Set File for generated code object*

### Description

Set File for generated code object

### Usage

```
file(description) <- value

## S4 replacement method for signature 'GeneratedCode,character'
file(description) <- value
```

### Arguments

| | |
|---|---|
| description | GeneratedCode |
| value | file name to associate with object |

## forLoopToLapply *Transfrom For Loop To Lapply*

### Description

Determine if a for loop can be parallelized, and if so transform it into a call to `lapply`. This first version will modify loops if and only if the body of the loop does not do any assignments at all.

### Usage

```
forLoopToLapply(forloop)
```

### Arguments

| | |
|---|---|
| `forloop` | R language object with class `for`. |

### Details

Recommended use case:

The functions in the body of the loop write to different files on each loop iteration.

The generated code WILL FAIL if:

Code in the body of the loop is truly iterative. Functions update global state in any way other than direct assignment.

### Value

call R call to `parallel::mclapply` if successful, otherwise the original forloop.

## generate *Generate Code From A Schedule*

### Description

Generate Code From A Schedule

Produces executable code that relies on a SNOW cluster on a single machine and sockets.

### Usage

```
generate(schedule, ...)

## S4 method for signature 'MapSchedule'
generate(schedule, ...)

## S4 method for signature 'TaskSchedule'
generate(schedule, portStart = 33000L,
  minTimeout = 600)
```

### Arguments

| | |
|---|---|
| schedule | object inheriting from class Schedule |
| ... | additional arguments to methods |
| portStart | first local port to use, can possibly use up to n * (n - 1) / 2 subsequent ports if every pair of n workers must communicate. |
| minTimeout | timeout for socket connection will be at least this many seconds. |

### Value

x object of class GeneratedCode

### See Also

schedule generic function to create Schedule, writeCode to write and extract the actual code, and makeParallel to do everything all at once.

---

## GeneratedCode-class *Generated code ready to write*

---

### Description

Generated code ready to write

### Slots

schedule contains all information to generate code

code executable R code

file name of a file where code will be written

---

## inferGraph *Infer Task Dependency Graph*

---

### Description

Statically analyze code to determine implicit dependencies

### Usage

```
inferGraph(code, ...)

## S4 method for signature 'character'
inferGraph(code, ...)

## S4 method for signature 'language'
inferGraph(code, ...)

## S4 method for signature 'expression'
inferGraph(code, ...)
```

### Arguments

| | |
|---|---|
| code | the file path to a script or an object that can be coerced to an expression. |
| ... | additional arguments to methods |

### Value

object of class DependGraph

### Examples

```
g <- inferGraph(parse(text = "
  a <- 1
  b <- 2
  c <- a + b
  d <- b * c
"))

ig <- as(g, "igraph")
plot(ig)
```

---

## makeParallel *Make Parallel Code From Serial*

---

### Description

makeParallel is a high level function that performs all the steps to generate parallel code, namely:

### Usage

```
makeParallel(code, graph = inferGraph(code), run = FALSE,
  scheduler = schedule, ..., generator = generate, generatorArgs = list(),
  file = FALSE, prefix = "gen_", overWrite = FALSE)
```

## Arguments

| | |
|---|---|
| code | file name or expression from parse |
| graph | object of class DependGraph |
| run | logical, evaluate the code once to gather timings? |
| scheduler, | function to produce a Schedule from a DependGraph. |
| ..., | additional arguments to scheduler |
| generator | function to produce GeneratedCode from a Schedule |
| generatorArgs | list of named arguments to use with generator |
| file | character name of the file to write the generated script. If FALSE then don't write anything to disk. If TRUE and code comes from a file then use prefix to make a new name and write a script. |
| prefix | character added to front of file name |
| overWrite | logical write over existing generated file |

## Details

1. Infer the task graph
2. Schedule the statements
3. Generate parallel code

The arguments allow the user to control every aspect of this process. For more details see vignette("makeParallel-concep

## Value

code object of class GeneratedCode

## Examples

```
# Make an existing R script parallel
script <- system.file("examples/mp_example.R", package = "makeParallel")
makeParallel(script)

# Write generated code to a new file
newfile <- tempfile()
makeParallel(script, file = newfile)

# Clean up
unlink(newfile)

# Pass in code directly
d <- makeParallel(parse(text = "lapply(mtcars, mean)"))

# Now we can examine generated code
writeCode(d)

# Specify a different scheduler
pcode <- makeParallel(parse(text = "x <- 1:100
```

```
y <- rep(1, 100)
z <- x + y"), scheduler = scheduleTaskList)

# Some schedules have plotting methods
plot(schedule(pcode))
```

---

`mapSchedule` *Data Parallel Scheduler*

---

## Description

This function detects parallelism through the use of top level calls to R's apply family of functions and through analysis of `for` loops. Currently supported apply style functions include `lapply` and `mapply`. It doesn't parallelize all for loops that can be parallelized, but it does do the common ones listed in the example.

## Usage

```
mapSchedule(graph)
```

## Arguments

| | |
|---|---|
| `graph` | DependGraph |

## Details

Consider using this if:

- `code` is slow
- `code` uses for loops or one of the apply functions mentioned above
- You have access to machine with multiple cores that supports `makeForkCluster` (Any UNIX variant should work, ie. Mac)
- You're unfamiliar with parallel programming in R

Don't use this if:

- `code` is fast enough for your application
- `code` is already parallel, either explicitly with a package such as parallel, or implicitly, say through a multi threaded BLAS
- You need maximum performance at all costs. In this case you need to carefully profile and interface appropriately with a high performance library.

Currently this function support `for` loops that update 0 or 1 global variables. For those that update a single variable the update must be on the last line of the loop body, so the for loop should have the following form:

```
for(i in ...){   ...   x[i] <- ... }
```

If the last line doesn't update the variable then it's not clear that the loop can be parallelized.

Road map of features to implement:

- Prevent from parallelizing calls that are themselves in the body of a loop.

## Examples

```
# Each iteration of the for loop writes to a different file- good!
# If they write to the same file this will break.
pfile <- makeParallel(parse(text = "
     fnames <- paste0(1:10, '.txt')
     for(f in fname){
         writeLines('testing...', f)
     }"))

# A couple examples in one script
serial_code <- parse(text = "
     x1 <- lapply(1:10, exp)
     n <- 10
     x2 <- rep(NA, n)
     for(i in seq(n)) x2[[i]] <- exp(i + 1)
")

p <- makeParallel(serial_code)

eval(serial_code)
x1
x2
rm(x1, x2)

# x1 and x2 should now be back and the same as they were for serial
eval(writeCode(p))
x1
x2
```

---

`MapSchedule-class` *Data parallel schedule*

---

## Description

Class for schedules that should be parallelized with apply style parallelism

---

`MeasuredDependGraph-class`
*Graph where each expression has been executed, timed, and the size of the variables have been measured.*

---

## Description

Will export once full pipeline works.

## Slots

`time` time in seconds to run each expression

---

`plot,TaskSchedule,missing-method`
*Gantt chart of a schedule*

---

## Description

Gantt chart of a schedule

## Usage

```
## S4 method for signature 'TaskSchedule,missing'
plot(x, blockHeight = 0.25,
  main = "schedule plot", xlab = "Time (seconds)", ylab = "Processor",
  evalColor = "gray", sendColor = "orchid", receiveColor = "slateblue",
  labelTransfer = TRUE, labelExpr = NULL, rectAes = list(density = NA,
  border = "black", lwd = 2), ...)
```

## Arguments

| | |
|---|---|
| `x` | TaskSchedule |
| `blockHeight` | height of rectangle, between 0 and 0.5 |
| `main` | title |
| `xlab` | x axis label |
| `ylab` | y ayis label |
| `evalColor` | color for evaluation blocks |
| `sendColor` | color for send blocks |
| `receiveColor` | color for receive blocks |
| `labelTransfer` | add labels for transfer arrows |
| `labelExpr` | NULL to use default numbering labels, FALSE to suppress labels, or a character vector of custom labels. |
| `rectAes` | list of additional arguments for `rect` |
| `...` | additional arguments to `plot` |

## runMeasure *Run and Measure Code*

### Description

Will export this once I the full pipeline works.

### Usage

```
runMeasure(code, graph = inferGraph(code), envir = globalenv(),
  timer = Sys.time)
```

### Arguments

| | |
|---|---|
| code | to be passed into inferGraph |
| graph | object of class DependGraph |
| envir | environment to evaluate the code in |
| timer | function that returns a timestamp. |

### Details

Run the serial code in the task graph and measure how long each expression takes to run as well as the object sizes of each variable that can possibly be transferred.

This does naive and biased timing since it doesn't account for the overhead in evaluating a single expression. However, this is fine for this application since the focus is on measuring statements that take at least on the order of 1 second to run.

### Value

graph object of class MeasuredDependGraph

## schedule *Schedule Dependency Graph*

### Description

Creates the schedule for a dependency graph. The schedule is the assignment of the expressions to different processors at different times. There are many possible scheduling algorithms. The default is mapSchedule, which does simple map parallelism using R's apply family of functions.

### Usage

```
schedule(graph, maxWorker = 2L, ...)

## S4 method for signature 'GeneratedCode'
schedule(graph, maxWorker = 2L, ...)

## S4 method for signature 'DependGraph'
schedule(graph)
```

### Arguments

| | |
|---|---|
| graph | object of class DependGraph |
| maxWorker | integer maximum number of parallel workers |
| ... | additional arguments to methods |

### References

See *Task Scheduling for Parallel Systems*, Sinnen, O. for a thorough treatment of what it means to have a valid schedule.

---

## Schedule-class *Schedule base class*

---

### Description

Schedule base class

### Slots

graph DependGraph used to create the schedule

evaluation data.frame assigning expressions to processors

---

## scheduleTaskList *Minimize Expression Start Time*

---

### Description

Implementation of "list scheduling". This is a greedy algorithm that assigns each expression to the earliest possible processor.

### Usage

```
scheduleTaskList(graph, maxWorker = 2L, exprTime = NULL,
  exprTimeDefault = 1e-05, sizeDefault = as.numeric(utils::object.size(1L)),
  overhead = 8e-06, bandwidth = 1.5e+09)
```

### Arguments

| | |
|---|---|
| graph | object of class DependGraph as returned from inferGraph |
| maxWorker | integer maximum number of processors |
| exprTime | time in seconds to execute each expression |
| exprTimeDefault | numeric time in seconds to execute a single expression. This will only be used if exprTime is NULL. |
| sizeDefault | numeric default size of objects to transfer in bytes |
| overhead | numeric seconds to send any object |
| bandwidth | numeric speed that the network can transfer an object between processors in bytes per second. We don't take network contention into account. This will have to be extended to account for multiple machines. |

### Details

This function is experimental and unstable. If you're trying to actually speed up your code through parallelism then consider using the default method in schedule for data parallelism. This function rewrites code to use task parallelism. Task parallelism means two or more processors run different R expressions simultaneously.

### Value

schedule object of class TaskSchedule

### References

Algorithm 10 in *Task Scheduling for Parallel Systems*, Sinnen (2007)

### Examples

```
code <- parse(text = "a <- 100
     b <- 200
     c <- a + b")

g <- inferGraph(code)
s <- scheduleTaskList(g)
plot(s)
```

---

## SerialSchedule-class *Schedule that contains no parallelism at all*

---

### Description

Schedule that contains no parallelism at all

## TaskSchedule-class *Task Parallel Schedule*

### Description

Task Parallel Schedule

### Slots

`transfer` transfer variables between processes

`maxWorker` maximum number of processors, similar to `mc.cores` in the parallel package

`exprTime` time in seconds to evaluate each expression

`overhead` minimum time in seconds to evaluate a single expression

`bandwidth` network bandwidth in bytes per second

## use_def *Use Definition Chain*

### Description

Compute a data frame of edges with one edge connecting each use of the variable x to the most recent definition or update of x.

### Usage

```
use_def(x, all_uses, all_definitions)
```

### Arguments

| | |
|---|---|
| `x` | variable name |
| `all_uses` | list containing variable names uses in each expression |
| `all_definitions` | list containing variable names defined in each expression |

### Value

data frame of edges suitable for use with `graph_from_data_frame`.

writeCode *Write Generated Code*

## Description

Write the generated code to a file and return the code.

## Usage

```
writeCode(code, file, ...)

## S4 method for signature 'GeneratedCode,logical'
writeCode(code, file, overWrite = FALSE,
  prefix = "gen_")

## S4 method for signature 'GeneratedCode,missing'
writeCode(code, file, ...)

## S4 method for signature 'GeneratedCode,character'
writeCode(code, file, overWrite = FALSE,
  ...)
```

## Arguments

| | |
|---|---|
| code | object of class GeneratedCode |
| file | character name of the file to write the generated script. If FALSE then don't write anything to disk. If TRUE and code comes from a file then use `prefix` to make a new name and write a script. |
| ... | additional arguments to methods |
| overWrite | logical write over existing file |
| prefix | character prefix for generating file names |

## Value

expression R language object, suitable for further manipulation

## See Also

`generate` to generate the code from a schedule, `makeParallel` to do everything all at once.

# Index


DependGraph, *6–8*, *12*
DependGraph *(*DependGraph-class*)*, 2
DependGraph-class, 2

file,DependGraph-method, 3
file,GeneratedCode-method
*(*file,DependGraph-method*)*, 3
file,Schedule-method
*(*file,DependGraph-method*)*, 3
file<-, 3
file<-,GeneratedCode,character-method
*(*file<-*)*, 3
forLoopToLapply, 4

generate, 4, *15*
generate,MapSchedule-method *(*generate*)*, 4
generate,TaskSchedule-method
*(*generate*)*, 4
GeneratedCode, *3*, *5*, *7*, *15*
GeneratedCode-class, 5
graph_from_data_frame, *14*

inferGraph, 5, *11*, *13*
inferGraph,character-method
*(*inferGraph*)*, 5
inferGraph,expression-method
*(*inferGraph*)*, 5
inferGraph,language-method
*(*inferGraph*)*, 5

lapply, *8*

makeForkCluster, *8*
makeParallel, *5*, 6, *15*
mapply, *8*
MapSchedule *(*MapSchedule-class*)*, 9
mapSchedule, 8, *11*
MapSchedule-class, 9
MeasuredDependGraph
*(*MeasuredDependGraph-class*)*, 9
MeasuredDependGraph-class, 9

parse, *7*
plot,TaskSchedule,missing-method, 10

rect, *10*
runMeasure, 11

Schedule, *5*, *7*
Schedule *(*Schedule-class*)*, 12
schedule, *5*, 11, *13*
schedule,DependGraph-method *(*schedule*)*, 11
schedule,GeneratedCode-method
*(*schedule*)*, 11
Schedule-class, 12
scheduleTaskList, 12
SerialSchedule *(*SerialSchedule-class*)*, 13
SerialSchedule-class, 13

TaskSchedule, *10*
TaskSchedule *(*TaskSchedule-class*)*, 14
TaskSchedule-class, 14

use_def, 14

writeCode, *5*, 15
writeCode,GeneratedCode,character-method
*(*writeCode*)*, 15
writeCode,GeneratedCode,logical-method
*(*writeCode*)*, 15
writeCode,GeneratedCode,missing-method
*(*writeCode*)*, 15