NODAL.jl Documentation

Release v0.3.0

Pedro Bruel

Dec 19, 2017

Contents

NODAL provides tools for implementing parallel and distributed program autotuners. This Julia package provides tools and optimization algorithms for implementing different Stochastic Local Search methods, such as Simulated Annealing and Tabu Search. NODAL is an ongoing project, and will implement more optimization and local search algorithms.

You can use NODAL to optimize user-defined functions with a few Stochastic Local Search basic methods, that are composed by building blocks also provided in the package. The package distributes evaluations of functions and technique executions between Julia workers. It is possible to have multiple instances of search techniques running on the same problem.

CHAPTER 1

Manual

1.1 Introduction

This page will give an overview of the functionalities in the package, how they work, and describe the manual.

1.2 Getting Started

This page will tell you how to install, test and contribute to the package.

1.2.1 Installing

NODAL.jl runs on Julia nightly. To get the latest version run:

```
julia> Pkg.clone("NODAL")
```

1.2.2 Running Tests

Run all tests with:

```
$ julia --color=yes test/runtests.jl
```

1.2.3 Contributing

You will need Julia nightly. Check the project's `REQUIRE` file for an up-to-date dependency list. Please, feel free to fork the repository and submit a pull request. You can also check the **GitHub** issues page for things that need to be done.

1.3 Examples

This page provides examples that will help you learn the package's API.

Note: The package is in heavy development. If something does not work as is show here, it is likely the API changed but the docs didn't. Submit an issue at the GitHub repository and the docs will be updated.

1.3.1 The Rosenbrock Function

The following is a very simple example, and you can find the source code for its latest version in the GitHub repository.

We will optimize the Rosenbrock cost function. For this we must define a `Configuration` that represents the arguments to be tuned. We also have to create and configure a tuning run. First, let's import NODAL.jl and define the cost function:

```
addprocs()

import NODAL

@everywhere begin
    using NODAL
    function rosenbrock(x::Configuration, parameters::Dict{Symbol, Any})
        return (1.0 - x["i0"].value)^2 + 100.0 * (x["i1"].value - x["i0"].value^2)^2
    end
end
```

We use the `addprocs()` function to add the default number of Julia workers, one per processing core, to our application. The `import` statement loads NODAL.jl in the current Julia worker, and the `@everywhere` macro defines the `rosenbrock` function and the module in all Julia workers available.

Cost functions must accept a `Configuration` and a `Dict{Symbol, Any}` as input. The `Configuration` is used to define the autotuner's search space, and the parameter dictionary can store data or function configurations.

Our cost function simply ignores the parameter dictionary, and uses the `"i0"` and `"i1"` parameters of the received configuration to calculate a value. There is no restriction on the names of `Configuration` parameter.

Our configuration will have two `FloatParameters`, which will be `Float64` values constrained to an interval. The intervals are `[-2.0, 2.0]` for both parameters, and their values start at `0.0`. Since we already used the names `"i0"` and `"i1"`, we name the parameters the same way:

```
configuration = Configuration([FloatParameter(-2.0, 2.0, 0.0, "i0"),
                               FloatParameter(-2.0, 2.0, 0.0, "i1")],
                               "rosenbrock_config")
```

Now we must configure a new tuning run using the `Run` type. There are many parameters to configure, but they all have default values. Since we won't be using them all, please see `Run`'s source for further details:

```
tuning_run = Run(cost                = rosenbrock,
                 starting_point      = configuration,
                 stopping_criterion  = elapsed_time_criterion,
                 report_after        = 10,
                 reporting_criterion = elapsed_time_reporting_criterion,
                 duration            = 60,
                 methods             = [[:simulated_annealing 1];
                                        [:iterative_first_improvement 1];
```

```
                                        [:iterated_local_search 1];
                                        [:randomized_first_improvement 1];
                                        [:iterative_greedy_construction 1];])
```

The `methods` array defines the search methods, and their respective number of instances, that will be used in this tuning run. This example uses one instance of every implemented search technique. The search will start at the point defined by `starting_point`.

The `stopping_criterion` parameter is a function. It tells your autotuner when to stop iterating. The two default criteria implemented are `elapsed_time_criterion` and `iterations_criterion`. The `reporting_criterion` parameter is also function, but it tells your autotuner when to report the current results. The two default implementations are `elapsed_time_reporting_criterion` and `iterations_reporting_criterion`. Take a look at the code if you want to dive deeper.

We are ready to start autotuning, using the `@spawn` macro. For more information on how parallel and distributed computing works in Julia, please check the Julia Docs. This macro call will run the `optimize` method, which receives a tuning run configuration and runs the search techniques in the background. The autotuner will write its results to a `RemoteChannel` stored in the tuning run configuration:

```
@spawn optimize(tuning_run)
result = take!(tuning_run.channel)
```

The tuning run will use the default neighboring and perturbation methods implemented by NODAL.jl to find new results. Now we can process the current result. In this example we just `print` it and loop until `optimize` is done:

```
print(result)
while !result.is_final
    result = take!(tuning_run.channel)
    print(result)
end
```

Running the complete example, we get:

```
$ julia --color=yes rosenbrock.jl
[Result]
Cost               : 1.0
Found in Iteration: 1
Current Iteration : 1
Technique          : Initialize
Function Calls     : 1
  ***
[Result]
Cost               : 1.0
Found in Iteration: 1
Current Iteration : 3973
Technique          : Initialize
Function Calls     : 1
  ***
[Result]
Current Iteration : 52289
Technique          : Iterative First Improvement
Function Calls     : 455
  ***
[Result]
Cost               : 0.01301071782455056
Found in Iteration: 10
Current Iteration : 70282
Technique          : Randomized First Improvement
```

```
Function Calls    : 3940
  ***
[Result]
Cost              : 0.009463518035824526
Found in Iteration: 11
Current Iteration : 87723
Technique         : Randomized First Improvement
Function Calls    : 4594
  ***
[Final Result]
Cost                  : 0.009463518035824526
Found in Iteration    : 11
Current Iteration     : 104261
Technique             : Randomized First Improvement
Function Calls        : 4594
Starting Configuration:
  [Configuration]
  name      : rosenbrock_config
  parameters:
    [NumberParameter]
    name : i0
    min  : -2.000000
    max  : 2.000000
    value: 1.100740
    ***
    [NumberParameter]
    name : i1
    min  : -2.000000
    max  : 2.000000
    value: 1.216979
Minimum Configuration :
  [Configuration]
  name      : rosenbrock_config
  parameters:
    [NumberParameter]
    name : i0
    min  : -2.000000
    max  : 2.000000
    value: 0.954995
    ***
    [NumberParameter]
    name : i1
    min  : -2.000000
    max  : 2.000000
    value: 0.920639
```

Note: The Rosenbrock function is by no means a good autotuning objetive, although it is a good tool to help you get familiar with the API. NODAL.jl certainly performs worse than most tools for this kind of function. Look at further examples is this page for more fitting applications.

1.3.2 Autotuning Genetic Algorithms

1.3.3 Autotuning LLVM Pass Ordering and Parameters

CHAPTER 2

Library

2.1 Core

This page will describes core and utility functions.

2.1.1 `Run` - `mutable struct`

The `Run` type encapsulates the configuration parameters of a tuning run. All parameters of `Run` are named and have default values, so it is possible to call its constructor with no parameters.

`cost::Function`

The `cost` parameter is the function that computes your program's fitness value, or cost, for a given `Configuration`. This function must receive a `Configuration`. Optionally, it can also receive a `Dict` with extra invariant parameters. The `cost` default value is a constant function:

```
cost::Function = (c(x) = 0)
```

`cost_arguments::Dict{Symbol, Any}`

```
cost_arguments::Dict{Symbol, Any} = Dict{Symbol, Any}()
```

`cost_evaluations::Integer`

```
cost_evaluations::Integer = 1
```

cost_values::Array{AbstractFloat, 1}

```
cost_values::Array{AbstractFloat, 1} = [0.0]
```

starting_point::Configuration

```
starting_point::Configuration = Configuration("empty")
```

starting_cost::AbstractFloat = 0.0

```
starting_cost::AbstractFloat = 0.0
```

report_after::Integer

```
report_after::Integer = 100
```

reporting_criterion::Function

```
reporting_criterion::Function = iterations_reporting_criterion
```

measurement_method::Function

```
measurement_method::Function = measure_mean!
```

stopping_criterion::Function

```
stopping_criterion::Function = iterations_criterion
```

duration::Integer

```
duration::Integer = 1_000
```

methods::Array{Any, 2}

```
methods::Array{Any, 2} = [[:simulated_annealing 2];]
```

channel::RemoteChannel

```
channel::RemoteChannel = RemoteChannel(()->Channel{Any}(128))
```

2.2 Types

This page will describe all Julia types defined by the package.

2.2.1 Abstract Types

2.2.2 Concrete Types

2.3 Search

This page will describe all search techniques, building blocks, and the search module execution flow.

2.3.1 Building Blocks

2.3.2 Techniques

2.4 Measurent

This page will describe the measurement module.

2.4.1 Parallel and Distributed Execution