Parallel & Distibuted Computing: Lecture 9

Alberto Paoluzzi

April 3, 2017

Make parallel: domain integration

1. Tools preparation and testing
2. Parallel prototype development
3. Timing and speed-up computing
4. Documentation

Tools preparation and testing

Integration codes

Look at the integration method

Finite integration formulas over polyhedral domains

Julia implementation

Basic integration functions
vertices and cells by columns !!

Test data preparation: 1/n

minimal surface: standard unit triangle in $z = 0$

```
using PyCall

V = [
    0.0     1.0     0.0;
    0.0     0.0     1.0
    0.0     0.0     0.0
]

FV = [ 1; 3; 2; ]''

P = V,FV
II(P, 0,0,0)
```

Test data preparation: 2/n

unit square: standard unit square in $z = 0$

```
V = [
    10.0    11.0    10.0    11.0;
    0.0     0.0     1.0     1.0;
    0.0     0.0     0.0     0.0;
]

FV = [ 1    2;
       2    4;
       3    3; ]

P = V,FV
II(P, 0,0,0)
```

Test data preparation: 2/n

Rotated unit square in $z = 0$

```
R1 = [cos(pi/6) -sin(pi/6) 0; sin(pi/6) cos(pi/6) 0; 0 0 1 ]

FV = [ 1    2;
       2    4;
       3    3; ]

P = R1*V,FV
II(P, 0,0,0)
```

Test data preparation: 2/n

Rotated unit square in $z = 0$

```
R2 = [cos(pi/6) 0 -sin(pi/6); 0 1 0;  sin(pi/6) 0 cos(pi/6)]

FV = [ 1    2;
       2    4;
       3    3; ]

P = R2*V,FV
II(P, 0,0,0)
II( (R2*R1*V, FV), 0,0,0)
```

Test data preparation: 2/n

Unit Tetrahedron in $\mathbb{E}^3$

```
V = [
    0.0     1.0     0.0     0.0;
    0.0     0.0     1.0     0.0;
    0.0     0.0     0.0     1.0;
    ]
FV = [ 1    2    4; 1   3   2;  4   3   1;  2   3   4]'

III( (V,FV), 0,0,0)
```

Test data preparation: 2/n

Build a tetrahedralization of the standard unit square

```
V = [
    0.0     1.0     0.0     1.0     0.0     1.0     0.0     1.
    0.0     0.0     1.0     1.0     0.0     0.0     1.0     1.
    0.0     0.0     0.0     0.0     1.0     1.0     1.0     1.
    ]
FV = ??

III( (V,FV), 0,0,0)
```

Test data preparation: 2/n

1. Translate and rotate the tetrahedra in $\mathbb{E}^3$
2. test the volume values

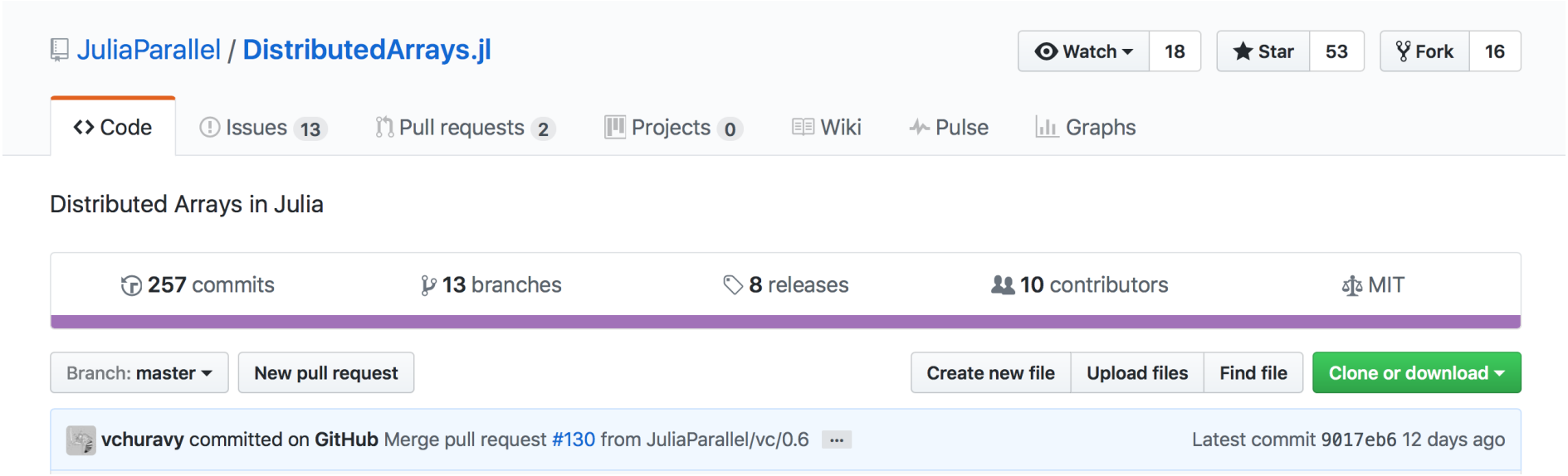
Parallel prototype development

Choose the Julia's parallel programming primitive

Distributed vs Shared Arrays

Shared Arrays use system shared memory to map the same array across many processes.
While there are some similarities to a DArray, the behavior of a SharedArray is quite different.
In a DArray, each process has local access to just a chunk of the data, and no two processes share the same chunk;
in contrast, in a SharedArray each "participating" process has access to the entire array.
A SharedArray is a good choice when you want to have a large amount of data jointly accessible to two or more processes on the same machine.

Distributed Arrays



Distributed Arrays

Common kinds of arrays can be constructed with functions beginning with d:

```
dzeros(100,100,10)
dones(100,100,10)
drand(100,100,10)
drandn(100,100,10)
dfill(x,100,100,10)
```

In the last case, each element will be initialized to the specified value x.

These functions automatically pick a distribution for you. For more control, you can specify which processes to use, and how the data should be distributed:

```
dzeros((100,100), workers()[1:4], [1,4])
```

Distributed Arrays

```
distribute(a::Array)
```

converts a local array to a distributed array

```
Pkg.add("DistributedArrays")
```

```
@everywhere using DistributedArrays
```

Assignment

Write the code

Timing and speed-up computing

Generate a bigger instance of data

Translate the bunny.obj to be read by julia . . .

Some alternatives:

- Write a file from python
- write a file from `.obj source` using regular expressions

read files from Julia

Introducing Julia/Working with text files

Let's timing the computation . . .

Check the speed-up . . .

Documentation

Use Nubew ...