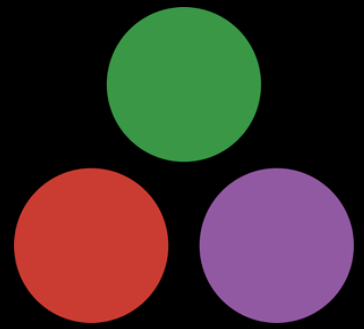


Distributed Computing

庄竞泽

Tsinghua, 2022.12

https://github.com/jzzhuang/Distributed

Embarrassingly parallel

Definition: *little or no effort is needed to separate the problem*

- Monte Carlo

```
function func()
    rand()
    ...
    return result
end

[func() for i in 1:1000]
```

- Multiple models

```
function func(model)
    ...
    return result
end

models = [...]
[func(models[i]) for i in 1:1000]
```

Contents

- Introduction: Distributed.jl
  - Minimal Example
  - Variable Scope
- Parallel Module in Our Project
  - A Glance at Our Project
  - How Our Files Organized
  - Code Review

Our Aim

- Create workers
- Load all needed functions
- Calculate
- Save data by scenario

For time longer than 1 hour

- Check progress
- Stop & Recover

Minimal example: Distributed.jl

- `addprocs(2)`

  Create n workers

- `@distributed for`

  distribute N tasks “equally”

  1st worker: first N/n

  2nd worker: next N/n

  …

```
using Distributed

addprocs(2)
@distributed for delay_time in [1, 1, 10, 10]
    println(delay_time);
    sleep(delay_time);
end
```

```
Task (runnable)
@0x00000000d229ae90
      From worker 2:    1
      From worker 3:    10
      From worker 2:    1     ← Worker 2 DONE!
      From worker 3:    10

Total time: 20s >> Expected 11s
```

Minimal example: Distributed.jl

- `pmap`

  distribute N tasks dynamically

- Large N & Little tasks:

  extra effort to distribute

  → `@distributed for`

```
using Distributed

addprocs(2)
pmap(delay_time ->(
        println(delay_time);
        sleep(delay_time);
    ),
    [1, 1, 10, 10])
```

From worker 3: 1
From worker 2: 1
From worker 3: 10
From worker 2: 10

Total time: 11s = Expected 11s

Variable Scope

```
function func()
    x="Julia is the best language"
    println(x)
end
```

`x` only in local scope

```
julia> func()
Julia is the best language

julia> println(x)
ERROR: UndefVarError: x not defined
```

Variable Scope

```
function func()
    x="Julia is the best language"
    println(x)
end
```

`x` only in local scope

```
julia> func()
Julia is the best language

julia> println(x)
ERROR: UndefVarError: x not defined
```

Scope Among Workers

```
addprocs(3)
function func()
    x="Julia is the best language"
    println(x)
end
```

Wrong: `func()` only in worker 1

```
julia> @everywhere func()
ERROR: On worker 2: UndefVarError: func not defined
```

Define `func()` @ everywhere

```
@everywhere function func()
@everywhere include("Func.jl")
```

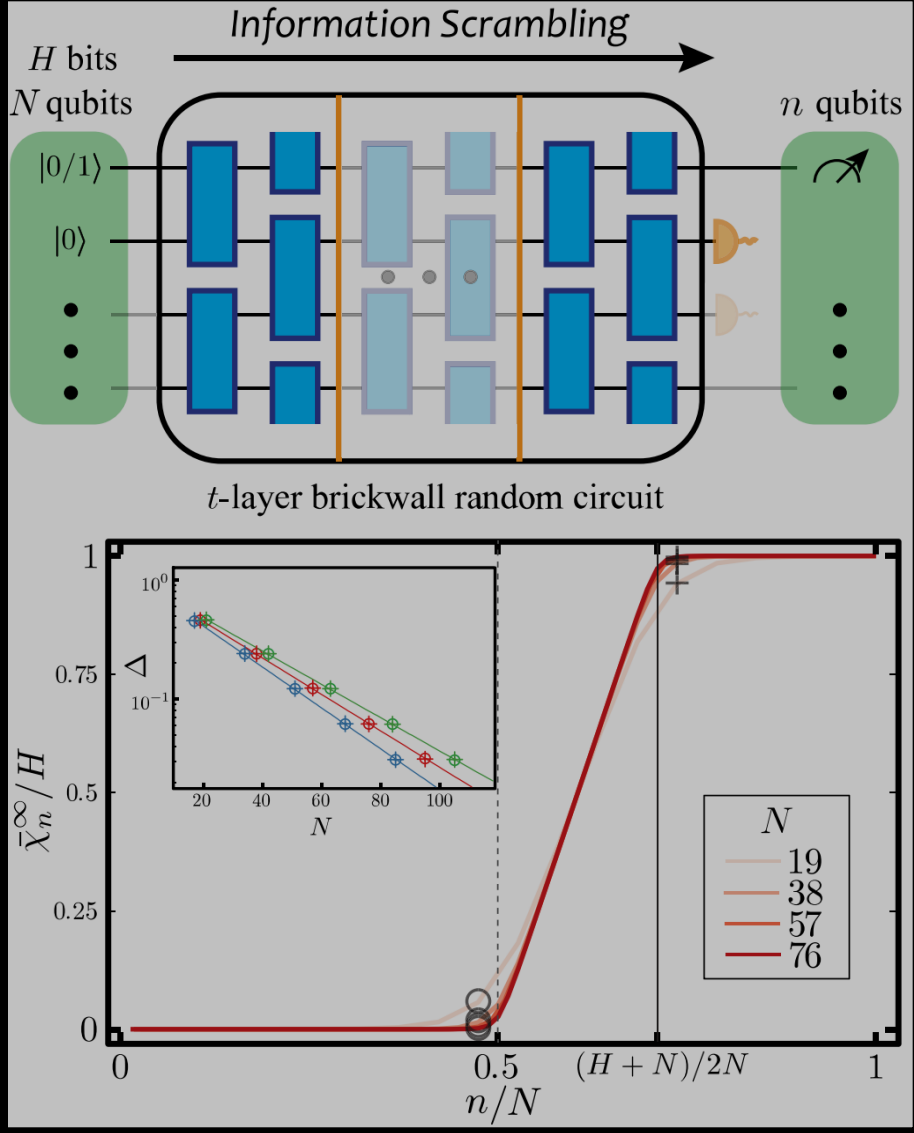
From worker 2: “Julia the best”
From worker 3: “Julia the best”

A Glance at Our Project

We focus on the scrambling of information χ

- Input: χ on subsystem S_{input}
- Process: random Clifford unitary U
- Output: amount of χ on S_{output}

➢ For various settings S_{input}, S_{output}, we need to sample random U many times.



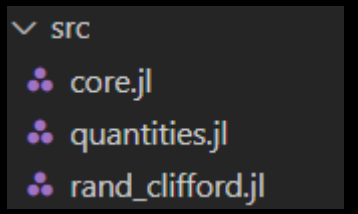
Phase-transition-like behavior in information retrieval of a quantum scrambled random circuit system
Zhuang, J.-Z., Y.-K. Wu, and L.-M. Duan. *Physical Review B* 106, no. 14: 144308

How Our Files Organized

Core

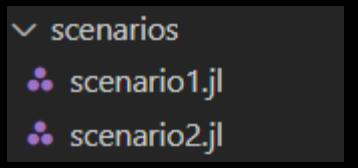
- Common calculation method
- Model description
- Random model generation
- Commonly used tools

Clifford state
Brick-wall random circuit
Random Clifford



Different scenarios

- Physical quantities
- Initial condition



Inside a scenario: $\{S_{input},\ S_{output}\}_m$, each sample U N times

What We Need

- Create workers
- Load all needed functions
- Calculate
- Save data by scenario

For time longer than 1 hour

- Check progress
- Stop & Recover

scenario1.jl

- filename and include
- scenario-specific functions
- store models in `iters`
- calculate `main(iters[i]...)` `main(iters[i])` `num_repeat` times

```julia
#scenario1.jl
filename = "scenario1"
include("../src/core.jl")

function func1()
end
function func2()
end

function main(a, b, c)
    ...
end

iters = [(a, b, c) for a in ..., b in ..., c in ...][:]

num_repeat = 500
num_workers = 180
parallel_param = ParallelParam(Tout = Matrix{Int16},
    num_repeat = num_repeat, num_workers = num_workers)


parallel(filename, main, iters, parallel_param)
```

core.jl

- `using` and `include`
- data and plot directory
- create dir if not exists

```
1   if !(@isdefined myid)
2       using Distributed
3       using Plots, LaTeXStrings, MathLink
4   end
5   using Serialization, Random
6   using Parameters
7
8   include("core1.jl")
9   include("core2.jl")
10  include("parallel.jl")
11
12  if !(@isdefined filename)
13      filename = "temp"
14      if myid()==1
15          println("filename not defined")
16      end
17  end
18
19  save_dir = "/data/$filename"
20  plot_dir = "/$filename"
21
22  if !isdir(save_dir)
23      create_directory(save_dir)
24  end
```

parallel.jl

```
1   function parallel(filename::String, func::Function, iters, parallel_param)
2       if (myid()!=1)  exit() end
3
4       spawn(filename, procs_needed, parallel_param)
5
6       if dynam_sched
7           pmap(i->parallel_main(i), i_iters)
8       else
9           @sync @distributed for i in i_iters
10              parallel_main(i)
11          end
12      end
13
14      arr = [fetch(@spawnat i getfield(Main, :sub_arr)) for i in 2:num_workers+1]
15      count = [fetch(@spawnat i getfield(Main, :sub_count)) for i in 2:num_workers+1]
16      rmprocs(2:num_workers+1)
17
18      println("Done.")
19
20      serialize("$save_dir/$(timename)$(is_running ? "" : "_temp").dat", (arr, count))
21  end
```

- spawn workers
- calculation
- collect all the data
- save

parallel.jl – spawn workers

```
1   procs_needed = n_iters * num_repeat
2
3   function spawn(filename, procs_needed, parallel_param)
4       num_workers = min(procs_needed, num_workers)
5
6       if length(workers()) == 1
7           add_workers = num_workers
8       elseif length(workers()) < num_workers
9           add_workers = num_workers - length(workers())
10      else
11          add_workers = 0
12      end
13      addprocs(add_workers)
14
15      @everywhere collect(2:num_workers+1) include("src/InfoClifford.jl")
16      if filename != "temp"
17          @everywhere collect(2:num_workers+1) include($filename*".jl")
18      end
19
20      return length(workers())
21  end
```

parallel.jl – before calculation

```
1   # unpack is necessary because func(large_array...) has non-allocating args
2   if unpack
3       @everywhere subprocs_func = x -> $func(x...)
4   else
5       @everywhere subprocs_func = $func
6   end
7
8   @everywhere sub_count, sub_arr = Int64[], $Tout[]
9   @everywhere sub_filename = "$save_dir/data/$(timename)_procs$(myid())_temp.dat"
```

- Pass variables to worker: add prefix $
- Worker save file to `sub_filename` where `timename` is the launching time

parallel.jl – calculation

```
11  function parallel_main(i)
12      global iters, sub_count, sub_arr
13      global is_running
14
15      if !is_running return end
16
17      if !isfile("$save_dir/data/is_running_$timename.mark")
18          serialize(sub_filename, (sub_arr, sub_count))
19          is_running = false
20          return
21      end
22
23      result = subprocs_func(iters[i])
24
25      push!(sub_count, i); push!(sub_arr, result)
26
27      now_hour = Dates.hour(now())
28
29      if (0 <= (now_hour - subprocs_utils.last_save_hour + 24) % 24 <= 20)
30          serialize(sub_filename, (sub_arr, sub_count))
31          subprocs_utils.last_save_hour = (now_hour + 1) % 24
32      end
33  end
```

- File “is_running” marks whether we want to stop
- When deleted, worker abandon their work
- Calculate
- Save

  if last save is > 1 hour ago

Thank You!

https://github.com/jzzhuang/Distributed