nb-l26-multithread

February 4, 2020

Scientific Computing, TU Berlin, WS 2019/2020, Lecture 26

Jürgen Fuhrmann, WIAS Berlin

1 Multithreading in Julia

- Start a function as a Task on a available thread `Threads.@spawn`.
- `wait(task)`: wait for the completion of the task
- `fetch(task)` wait for the completion of the taks and retrieve result

1.1 Packages, setup

[1]:
```
using BenchmarkTools
using LinearAlgebra
using Printf
using PyPlot

injupyter()=isdefined(Main, :IJulia) && Main.IJulia.inited
```

[1]: `injupyter (generic function with 1 method)`

- In order to allow multithreading, one has to start Julia with the environment variable `JULIA_NUM_THREADS` set to the desired number
- Figure out the number of threads

[2]:
```
Threads.nthreads()
```

[2]: 4

1.1.1 Starting threads using spawn()

A task which just returns the number of the thread executing it

[3]:
```
function mytask()
    return Threads.threadid()
end
```

[3]: `mytask (generic function with 1 method)`

[4]:
```
function threads_hello(;ntasks=10)
    println("number of tasks: $(ntasks)")
    tasks=[Threads.@spawn mytask() for i=1:ntasks]
    for i=1:length(tasks)
        ithd=fetch(tasks[i])
        println("task #$(i) was executed thread #$(ithd)")
    end
end
```

[4]:
```
threads_hello (generic function with 1 method)
```

[5]:
```
threads_hello(ntasks=10)
```

```
number of tasks: 10
task #1 was executed thread #3
task #2 was executed thread #4
task #3 was executed thread #1
task #4 was executed thread #2
task #5 was executed thread #4
task #6 was executed thread #2
task #7 was executed thread #3
task #8 was executed thread #4
task #9 was executed thread #1
task #10 was executed thread #2
```

1.1.2 Multithreaded dot product calculation

- Interesting example as one has to collect the result into one variable
- Start with creating a subdivision of the loop length into equal parts

[6]:
```
function partition(N,ntasks)
    loop_begin=zeros(Int64,ntasks)
    loop_end=zeros(Int64,ntasks)
    for itask=1:ntasks
        ltask=Int(floor(N/ntasks))
        loop_begin[itask]=(itask-1)*ltask+1
        if itask==ntasks # adjust last task length
            ltask=N-(ltask*(ntasks-1))
        end
        loop_end[itask]=loop_begin[itask]+ltask-1
    end
    return (loop_begin,loop_end)
end
```

[6]:
```
partition (generic function with 1 method)
```

Check this

[7]:
```
partition(100000,3)
```

[7]:
```
([1, 33334, 66667], [33333, 66666, 100000])
```

Calculate part of scalar product from n0 to n1

[8]:
```
function mydot(a,b,n0,n1)
    result=0.0
    for i=n0:n1
        result+=a[i]*b[i]
    end
    result
end
```

[8]:
```
mydot (generic function with 1 method)
```

Calculate scalar product in parallell

[9]:
```
function threaded_mydot(a,b,N,ntasks)
    loop_begin,loop_end=partition(N,ntasks)
    tasks=[Threads.@spawn mydot(a,b,loop_begin[i],loop_end[i]) for i=1:ntasks]
    return    mapreduce(task->fetch(task),+,tasks)
end
```

[9]:
```
threaded_mydot (generic function with 1 method)
```

Compare times, check accuracy

[10]:
```
function test_threaded_mydot(;N=400000, ntasks=4)
    a=rand(N)
    b=rand(N)
    res_s=@btime mydot($a,$b,1,$N)
    res_p=@btime threaded_mydot($a,$b,$N,$ntasks)
    res_s   res_p
end
```

[10]:
```
test_threaded_mydot (generic function with 1 method)
```

[11]:
```
test_threaded_mydot(N=400000, ntasks=4)
```

```
  363.683  s (0 allocations: 0 bytes)
  98.672  s (44 allocations: 3.64 KiB)
```

[11]:
```
true
```

1.1.3 Multithreaded dot product calculation based on fork-join model

The fork-join model promises that we can skip the scheduling "by hand"

[12]:
```julia
function forkjoin_mydot_primitive(a,b)
    result=0.0
    N=length(a)
    Threads.@threads for i=1:N
        result+=a[i]*b[i]
    end
    result
end
```

[12]:
```
forkjoin_mydot_primitive (generic function with 1 method)
```

[13]:
```julia
function test_forkjoin_mydot_primitive(;N=400000)
    a=rand(N)
    b=rand(N)
    res_s=@btime mydot($a,$b,1,$N)
    res_p=@btime forkjoin_mydot_primitive($a,$b)
    res_s  res_p
end
```

[13]:
```
test_forkjoin_mydot_primitive (generic function with 1 method)
```

[14]:
```julia
test_forkjoin_mydot_primitive(N=400000)
```

```
  358.839  s (0 allocations: 0 bytes)
  7.154 ms (800031 allocations: 12.21 MiB)
```

[14]:
```
false
```

What was wrong ?

- In the parallel loop, there is no control over the access to `result`:
- A possible scenario:
  - Thread 1 wants to perform `result+=x`
  - Thread 2 wants to perform `result+=y`
  1. Thread 1 reads `result` and gets its actual value `r0`
  2. Thread 1 reads `result` and gets its actual value `r0`
  3. Thread 1 performs its operation and writes back `r0+x`
  4. Thread 2 performs its operation and writes back `r0+y`
  - In the result, we have `result=r0+y` instead of the correct value `r0+x+y`

1.1.4 Atomic variables

- Atomic variables are protected from unscheduled update:
- Thread 2 would have to wait until Thread 1 is done with its operation
- This is expensive due to the necessary communication infrastructure

[15]:
```julia
function forkjoin_mydot_atomic(a,b)
    result = Threads.Atomic{Float64}(0.0)
    N=length(a)
```

```
    Threads.@threads for i=1:N
        Threads.atomic_add!(result,a[i]*b[i])
    end
    return result[]
end
```

[15]:
```
forkjoin_mydot_atomic (generic function with 1 method)
```

[16]:
```
function test_forkjoin_mydot_atomic(;N=400000)
    a=rand(N)
    b=rand(N)
    res_s=@btime mydot($a,$b,1,$N)
    res_p=@btime forkjoin_mydot_atomic($a,$b)
    res_s   res_p
end
```

[16]:
```
test_forkjoin_mydot_atomic (generic function with 1 method)
```

[17]:
```
test_forkjoin_mydot_atomic(N=400000)
```

```
  351.310  s (0 allocations: 0 bytes)
  14.337 ms (33 allocations: 3.08 KiB)
```

[17]:
```
true
```

… at least it is correct

1.1.5 Reduction variables

- Our threaded result was correct, because each thread had its own temporary variable
- Transfer this concept to the fork-join model
- Introduce a reduction variable

[18]:
```
function forkjoin_mydot_reduction(a,b)
    N=length(a)
    result=zeros(Threads.nthreads())
    Threads.@threads for i=1:N
        ithd=Threads.threadid()
        result[ithd]+=a[i]*b[i]
    end
    return sum(result)
end

function test_forkjoin_mydot_reduction(;N=400000)
    a=rand(N)
    b=rand(N)
    res_s=@btime mydot($a,$b,1,$N)
    res_p=@btime forkjoin_mydot_reduction($a,$b)
```

```
    res_s   res_p
end
```

[18]: test_forkjoin_mydot_reduction (generic function with 1 method)

[19]:
```
test_forkjoin_mydot_reduction(N=400000)
```

```
  350.891  s (0 allocations: 0 bytes)
  269.738  s (31 allocations: 3.14 KiB)
```

[19]: true

- Results hint on the experimental character of the implementation
- Handling of the implicit barriers can be a problem

1.1.6 Schönauer vector triad:

- d[i]=a[i]+b[i]*c[i]
- Vary length of the array
- -> Memory performance issues

[20]:
```
function vtriad(N,nrepeat)

    a = Array{Float64,1}(undef,N)
    b = Array{Float64,1}(undef,N)
    c = Array{Float64,1}(undef,N)
    d = Array{Float64,1}(undef,N)

    Threads.@threads for i=1:N
        a[i]=i
        b[i]=N-i
        c[i]=i
        d[i]=-i
    end

    t_scalar=@elapsed begin
        @inbounds @fastmath for j=1:nrepeat
            for i=1:N
                d[i]=a[i]+b[i]*c[i]
            end
        end
    end

    t_parallel=@elapsed begin
        for j=1:nrepeat
            Threads.@threads for i=1:N
                @inbounds @fastmath  d[i]=a[i]+b[i]*c[i]
            end
```

```
        end
    end
    GFlops=N*nrepeat*2.0/1.0e9
    @printf("% 10d % 10.3f % 10.3f % 10.3f\n", N, GFlops/t_scalar,GFlops/
 ↪t_parallel,t_scalar/t_parallel)
    return [N, GFlops/t_scalar,GFlops/t_parallel,t_scalar/t_parallel]
    GC.gc()
end
```

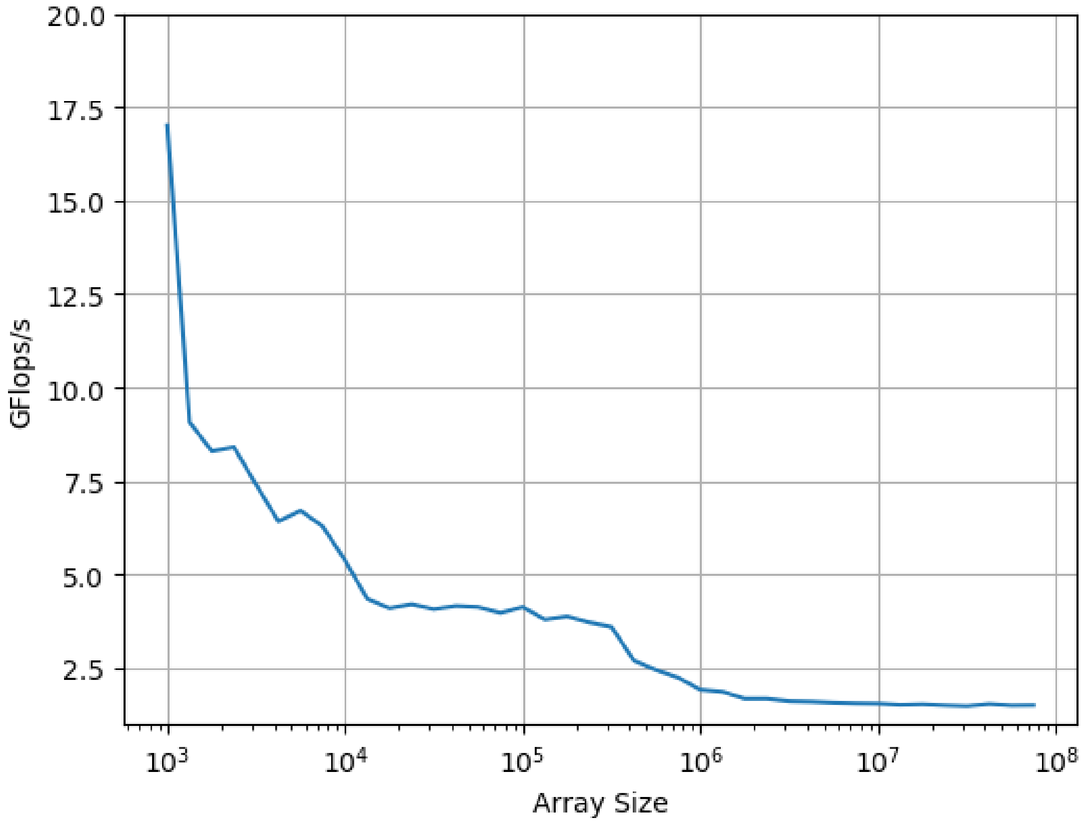
[20]: vtriad (generic function with 1 method)

[21]:
```
function run_triad()
    # Approximate number of FLOPs per measurement
    flopcount=5.0e8
    # Smallest array size
    N0=1000
    # Data points per decade (of array size)
    ppdec=8
    # Number of array size increases
    nrun=40
    # File header
    @printf("# nthreads=%d\n",Threads.nthreads())
    @printf("#        N    S_GFlops/s  P_GFlops/s  speedup\n")
    # Loop over measurements
    N=N0
    result=zeros(4,nrun)
    for irun=0:nrun-1
        # Have exact powers of 10 in the measurement
        if (irun%ppdec==0)
            N=N0
            N0*=10
        end
        # Calculate number of repeats so that overall effort stays    constant
        nrepeat=flopcount/N
        result[:,irun+1].=vtriad(Int(ceil(N)),Int(ceil(nrepeat)))
        N=N*10^(1.0/ppdec)
    end
    PyPlot.figure(1)
    PyPlot.semilogx(result[1,:], result[2,:])
    PyPlot.grid()
    PyPlot.ylim(1,20)
    PyPlot.xlabel("Array Size")
    PyPlot.ylabel("GFlops/s")
end
```

[21]: run_triad (generic function with 1 method)

[22]:
```
run_triad()
```

```
# nthreads=4
#         N   S_GFlops/s  P_GFlops/s  speedup
      1000     17.006       0.234       0.014
      1334      9.075       0.313       0.035
      1779      8.300       0.408       0.049
      2372      8.407       0.534       0.064
      3163      7.399       0.715       0.097
      4217      6.421       0.921       0.143
      5624      6.706       1.214       0.181
      7499      6.286       1.600       0.255
     10000      5.378       2.034       0.378
     13336      4.352       2.567       0.590
     17783      4.101       3.311       0.807
     23714      4.207       3.994       0.949
     31623      4.076       4.997       1.226
     42170      4.160       5.178       1.245
     56235      4.131       6.570       1.591
     74990      3.974       7.512       1.890
    100000      4.132       8.701       2.106
    133353      3.796       9.274       2.443
    177828      3.879       9.811       2.529
    237138      3.720      10.608       2.852
    316228      3.601      10.617       2.948
    421697      2.704       4.601       1.702
    562342      2.449       3.219       1.314
    749895      2.243       2.701       1.204
   1000000      1.923       2.293       1.193
   1333522      1.867       2.052       1.099
   1778280      1.683       1.867       1.109
   2371374      1.684       1.752       1.040
   3162278      1.617       1.764       1.091
   4216966      1.603       1.746       1.089
   5623414      1.575       1.702       1.080
   7498943      1.558       1.676       1.075
  10000000      1.551       1.648       1.062
  13335215      1.516       1.654       1.091
  17782795      1.532       1.651       1.078
  23713738      1.504       1.643       1.092
  31622777      1.485       1.629       1.097
  42169651      1.539       1.612       1.047
  56234133      1.502       1.643       1.094
```




```
74989421      1.507      1.597      1.059
```

[22]: `PyObject Text(24.000000000000007, 0.5, 'GFlops/s')`

This notebook was generated using Literate.jl.