nb-l26-multithread

February 4, 2020

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 task and retrieve result

1.1 Packages, setup

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

    injupyter() = isdefined(Main, :IJulia) && Main.IJulia.init

[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

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

[3]: mytask (generic function with 1 method)
```python
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
def mytask()
    # Task body
end
```

```
threads_hello (generic function with 1 method)
```

```
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

```python
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
```

```
partition (generic function with 1 method)
```

Check this
Calculate part of scalar product from n0 to n1

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

Calculate scalar product in parallel

```plaintext
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
```

Compare times, check accuracy

```plaintext
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
```

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”
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

forkjoin_mydot_primitive (generic function with 1 method)

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

test_forkjoin_mydot_primitive (generic function with 1 method)

test_forkjoin_mydot_primitive(N = 400000)

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

function forkjoin_mydot_atomic(a,b)
    result = Threads.Atomic{Float64}(0.0)
    N = length(a)
```julia
Threads.@threads for i=1:N
    Threads.atomic_add!(result,a[i]*b[i])
end
return result[]
end
```

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

```julia
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
```

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

... at least it is correct
1.1.6 Schönauer vector triad:

- \( d[i] = a[i] + b[i] \times c[i] \)
- Vary length of the array
- \( \rightarrow \) Memory performance issues

```plaintext
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
```

Results hint on the experimental character of the implementation
Handling of the implicit barriers can be a problem
\begin{verbatim}
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_parallel)
return [N, GFlops/t_scalar,GFlops/t_parallel,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)
\end{verbatim}
run_triad()

# nthreads=4

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