Parallel Computing with Julia

1. Introduction
   - an overview
   - message passing
   - GPU acceleration

2. Multiprocessing and Multithreading
   - using `Distributed`
   - using `Base.Threads`
   - multithreaded matrix multiplication with BLAS

MCS 507 Lecture 11
Mathematical, Statistical and Scientific Software
Jan Verschelde, 15 September 2023

Parallel Computing with Julia

how Julia works

Julia works with an LLVM JIT compiler framework,

- LLVM = Low Level Virtual Machine,
- JIT = Just In Time,

used for just-in-time generation of machine code.

The two stages in running a Julia function:

1. The function is parsed and types are inferred.
2. LLVM code is generated by the JIT compiler, which is then optimized and compiled to native code.

The second time the function is called, the native code is executed.

functions are generic and dynamic

```
julia> f(x) = 4x - 3
f (generic function with 1 method)
```

The code is dynamic because we did not specify the type of `f` or `x`.

Functions are by default thus generic, ready to work with different data types for the variables.

the JIT bytecode

```
julia> code_llvm(f, (Float64,))

;  @ REPL[1]:1 within `f'
define double @julia_f_15970(double) {
top:
;  @ promotion.jl:314 within `*' @ float.jl:399
   %1 = fmul double %0, 4.000000e+00
;
;  @ promotion.jl:315 within `-' @ float.jl:397
   %2 = fadd double %1, -3.000000e+00
;
  ret double %2
}
```

generated assembly code

```
julia> code_native(f, (Float64,))
.text
;  @ REPL[1]:1 within ‘f’
movabsq $140620982887696, %rax  # imm = 0x7FE4DFBBA510
;  @ promotion.jl:314 within ‘*’ @ float.jl:399
mulsd (%rax), %xmm0
movabsq $140620982887704, %rax  # imm = 0x7FE4DFBBA518
;
;  @ promotion.jl:315 within ‘-’ @ float.jl:397
addsd (%rax), %xmm0
;
retq
nopl (%rax)
;
```

parallel computations

We distinguish between three types of parallel computations.

1. Distributed Memory Parallel Computers

   Message passing is a common method of development.

2. Shared Memory Parallel Computers

   Parallel programs apply multithreading on multicore processors.

3. Acceleration with Graphics Processing Units

Parallel Computing with Julia

message passing – MPI wrappers for Julia

Message passing is a common manner to develop programs on parallel distributed memory computers.

The Message Passing Interface (MPI) provides a standard definition.
`MPI.jl` is a Julia interface to MPI, inspired by mpi4py.

Available at `https://github.com/JuliaParallel`.

Its installation requires a shared binary installation of a C MPI library, supporting the MPI 3.0 standard or later.

The MPI.jl is a Julia package, install as `using MPI`.

Simon Byrne, Lucas C. Wilcox, and Valentin Churavy:
MPI.jl: Julia bindings for the Message Passing Interface.
In *JuliaCon Proceedings*, 1(1), 68, 2021.

the Julia program `mpi_hello_world.jl`

Adapted from `JuliaParallel/MPI.jl`, from the docs/examples:

```
using MPI
MPI.Init()

comm = MPI.COMM_WORLD
myid = MPI.Comm_rank(comm)
size = MPI.Comm_size(comm)

print("Hello from $myid of $size.\n")

MPI.Barrier(comm)
```

Run with `mpiexecjl`, locate and adjust path.

point-to-point communication with Julia

```
using MPI
MPI.Init()

comm = MPI.COMM_WORLD
myid = MPI.Comm_rank(comm)

if myid == 0
    data = Dict('a' => 7, 'b' => 3.14)
    println("$myid sends $data to 1")
    MPI.send(data, comm; dest=1, tag=11)
elseif myid == 1
    data = MPI.recv(comm; source=0, tag=11)
    println("$myid received $data from 0")
end
```

Running in a Terminal Windows, at the command prompt:

```
$ mpiexecjl -n 2 julia mpi_point2point.jl
0 sends Dict{Char, Real}('a' => 7, 'b' => 3.14) to 1
1 received Dict{Char, Real}('a' => 7, 'b' => 3.14) from 0
```

Parallel Computing with Julia

1. Introduction
   - an overview
   - message passing
   - GPU acceleration

2. Multiprocessing and Multithreading
   - using `Distributed`
   - using `Base.Threads`
   - multithreaded matrix multiplication with BLAS

acceleration on Graphics Processing Units

Tim Besard, Christophe Foket, and Bjorn de Sutter.
Effective Extensible Programming: Unleashing Julia on GPUs.
IEEE Transactions on Parallel and Distributed Systems
30(4):827–841, 2019.

The LLVM (Low Level Virtual Machine) compiler is capable to target both CPUs and CUDA GPUs.
`https://github.com/JuliaGPU/CUDAnative.jl`

Parallel Computing with Julia

running multiple processes

We can start the Julia interpreter with multiple worker processes.

```
$ julia -p 4

julia> using Distributed

julia> nprocs()
5

julia> nworkers()
4
```

estimating π

The script `estimatepi1.jl` contains

```
"""
    function estimatepi(n)

Runs a simple Monte Carlo method
to estimate pi with n samples.
"""
function estimatepi(n)
   count = 0
   for i=1:n
       x = rand()
       y = rand()
       count += (x^2 + y^2) <= 1
   end
   return 4*count/n
end
```

timing the code

The script continues below.

```
timestart = time()
estpi = estimatepi(10_000_000_000)
elapsed = time() - timestart
println("The estimate for Pi : $estpi")
println("The elapsed time : $elapsed seconds")
```

a distributed memory version

```
using Distributed, Statistics

# @everywhere function estimatepi(n)
#
# Runs a simple Monte Carlo method
# to estimate pi with n samples.

@everywhere function estimatepi(n)
   count = 0
   for i=1:n
       x = rand()
       y = rand()
       count += (x^2 + y^2) <= 1
   end
   return 4*count/n
end
```

mapping and timing the code

The script continues.

```
parallelpi(N) = mean(pmap(n->estimatepi(n),
          [N/nworkers() for i=1:nworkers()]));

np = nprocs()
nw = nworkers()
println("number of processes : $np")
println("number of workers : $nw")

timestart = time()
estpi = parallelpi(10_000_000_000)
elapsed = time() - timestart
println("The estimate for Pi : $estpi")
println("The elapsed time : $elapsed seconds")
```

getting the wall clock time

```
$ time julia estimatepi1.jl
The estimate for Pi : 3.1415660636
The elapsed time : 122.15529894828796 seconds

real    2m2.422s
user    2m2.511s
sys     0m0.766s

$ time julia -p 2 estimatepidp.jl
number of processes : 3
number of workers : 2
The estimate for Pi : 3.1415942171999998
The elapsed time : 77.822107076649 seconds

real    1m20.103s
user    2m37.395s
sys     0m2.400s
```

running many processes

Running version on two 22-core 2.2 GHz Intel Xeon E5-2699 processors in a CentOS Linux workstation with 256 GB RAM.

p	wall clock time	elapsed time
1	2m 2.422s	122.155s
2	1m 20.103s	77.822s
4	42.557s	39.992s
8	28.448s	25.221s
16	17.459s	12.729s
32	15.024s	7.595s
64	19.606s	6.821s
41	15.062s	6.268s
42	15.394s	6.226s
43	17.898s	8.498s

distributed memory parallelism

Distribubed memory parallel programming is built on two primitives:

1. *remote calls* execute a function on a remote processor, and
2. *remote references* that are returned by the remote processor to the caller.

In addition, Julia provides

- a distributed array data structure,
- a `pmap` implementation, and
- the `@parallel` macro for parallel loops.

Parallel Computing with Julia

launching Julia with many threads

At the command prompt we can define how many threads the Julia interpreter can use.

```
$ JULIA_NUM_THREADS=8 julia

julia> using Base.Threads

julia> nthreads()
8
```

a parallel for loop

```
julia> threadid()
1

julia> @threads for i=1:8
           print(" $(threadid())")
       end
 2 4 7 5 3 6 8 1
julia>
```

threads write into a shared array

```
julia> A = zeros(nthreads());

julia> @threads for i=1:nthreads()
           A[i] = threadid()
       end

julia> A
8-element Array{Float64,1}:
 1.0
 2.0
 3.0
 4.0
 5.0
 6.0
 7.0
 8.0
```

a multithreaded Monte Carlo

```
using Base.Threads
import Statistics

myrand(x::Int64) = (1103515245x + 12345) % 2^31

"""
    function estimatepi(n)

Runs a simple Monte Carlo method
to estimate pi with n samples.
"""
function estimatepi(n)
   r = threadid()
   count = 0
   for i=1:n
       r = myrand(r)
       x = r/2^31
       r = myrand(r)
       y = r/2^31
       count += (x^2 + y^2) <= 1
   end
   return 4*count/n
end
```

the script continues

```
nt = nthreads()
println("The number of threads : $nt")
estimates = zeros(nt)

timestart = time()
@threads for i=1:nt
    estimates[i] = estimatepi(10_000_000_000/nt)
end
estpi = Statistics.mean(estimates)
elapsed = time() - timestart

println("The estimate for Pi : $estpi")
println("The elapsed time : $elapsed seconds")
```

getting the wall clock time

```
$ time JULIA_NUM_THREADS=1 julia estimatepimt.jl
The number of threads : 1
The estimate for Pi : 3.1415934104
The elapsed time : 62.05984592437744 seconds

real    1m2.313s
user    1m2.438s
sys     0m0.730s

$ time JULIA_NUM_THREADS=2 julia estimatepimt.jl
The number of threads : 2
The estimate for Pi : 3.1415889355999997
The elapsed time : 32.41803598403931 seconds

real    0m32.722s
user    1m3.881s
sys     0m0.765s
```

running on many threads

Running on two 22-core 2.2 GHz Intel Xeon E5-2699 processors in a CentOS Linux workstation with 256 GB RAM.

p	wall clock time	elapsed time
1	1m 2.313s	62.060s
2	32.722s	32.418s
3	22.471s	22.190s
4	17.343s	17.042s
5	14.170s	13.896s
6	12.300s	11.997s
7	10.702s	10.442s

Parallel Computing with Julia

inplace matrix matrix multiplication

```
julia> using LinearAlgebra

julia> A=[1.0 2.0; 3.0 4.0]; B=[1.0 1.0; 1.0 1.0];

julia> C = similar(B); mul!(C, A, B)
2×2 Array{Float64,2}:
 3.0  3.0
 7.0  7.0
```

multithreaded matrix multiplication

Basic Linear Algebra Subprograms (BLAS) specifies common elementary linear algebra operations.

```
help?> BLAS.set_num_threads
  set_num_threads(n)

  Set the number of threads the BLAS library should use.
```

Setting the number of threads provides a parallel matrix multiplication.

a Julia program matmatmulmt.jl

```
using LinearAlgebra

if length(ARGS) < 2
    println("use as")
    print("        julia ", PROGRAM_FILE)
    println(" dimension nthreads")
else
    n = parse(Int, ARGS[1])
    p = parse(Int, ARGS[2])

    BLAS.set_num_threads(p)
    A = rand(n, n)
    B = rand(n, n)
    C = similar(B)
    @time mul!(C, A, B)
end
```

runs on two 22-core processors

```
$ julia matmatmulmt.jl 8000 1
 20.823673 seconds (2.70 M allocations: 130.252 MiB)

$ julia matmatmulmt.jl 8000 2
 11.338446 seconds (2.70 M allocations: 130.252 MiB)

$ julia matmatmulmt.jl 8000 4
  6.242092 seconds (2.70 M allocations: 130.252 MiB)

$ julia matmatmulmt.jl 8000 8
  3.853406 seconds (2.70 M allocations: 130.252 MiB)

$ julia matmatmulmt.jl 8000 16
  2.487637 seconds (2.70 M allocations: 130.252 MiB)

$ julia matmatmulmt.jl 8000 32
  1.864454 seconds (2.70 M allocations: 130.252 MiB)
$
```

the peak flops performance

`peakflops` computes the peak flop rate of the computer by using double precision `gemm!`.

```
julia> using LinearAlgebra

julia> peakflops(8000)
3.3312896110138 68e11

julia> peakflops(16000)
3.475269847112081e11

julia> peakflops(4000)
3.1302047295730 54e11
```

Summary and Exercises

Ivo Balbaert, Avik Sengupta, Malcom Sherrington:
Julia: High Performance Programming. Leverage the power of Julia to design and develop high performing programs.
Packt Publishing, 2016.

Exercises:

1. In the distributed Monte Carlo simulation, verify that the seed for the random number generator are different for each process.
2. Examine the quality up for the distributed Monte Carlo simulation. If we can afford to wait the same amount of time, how many more samples can be take?
3. Answer the same question as in the previous exercise, but now for the multithreaded Monte Carlo simulation.