Tasking with Julia

1. The NCSA Supercomputer Delta
   - using a real supercomputer
2. Programming Parallel Shared Memory Computers
   - tasking with Julia
3. Parallel Recursive Functions
   - the Fibonacci numbers
   - parallel recursive quadrature
   - parallel merge sort
4. Basic Linear Algebra Subprograms
   - multithreaded matrix multiplication

MCS 572 Lecture 13
Introduction to Supercomputing
Jan Verschelde, 25 September 2024

Tasking with Julia

1. The NCSA Supercomputer Delta
   - using a real supercomputer
2. Programming Parallel Shared Memory Computers
   - tasking with Julia
3. Parallel Recursive Functions
   - the Fibonacci numbers
   - parallel recursive quadrature
   - parallel merge sort
4. Basic Linear Algebra Subprograms
   - multithreaded matrix multiplication

The NCSA Supercomputer Delta

From the top 500 of June 2024:

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)
227	**Delta** - Apollo 6500, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100, Slingshot-10, HPE NCSA United States	49,600	3.81	8.05

Our course has access to the CPU nodes of Delta.

Delta offers 124 CPU nodes consisting of:

- Dual AMD 64-core 2.45 GHz Milan processors
- 256 GB DDR4-3200 RAM
- 800 GB NVMe solid-state disk

getting started

Consider `mpi_hello_world.c` from Lecture 4.

Use `scp` to get the program on your NCSA account.

Then at the terminal when logged in at an interactive node, type

```
mpicc -o hello mpi_hello_world.c
```

to compile the program. The output is in the file `hello`.

Type `accounts` to see the balance on our project.

Our account name should be used in the `slurm` script.

Look at the NCSA System Documentation Hub, on Delta.
The sample scripts of the Quick Start Guide are great.

SLURM = Simple Linux Utility for Resource Management.

the script to run 8 mpi jobs on delta

```
#!/bin/bash
#SBATCH --mem=4g
#SBATCH --nodes=8
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --partition=cpu
#SBATCH --account=bdje-delta-cpu # returned by "accounts"
#SBATCH --job-name=mpi_hello_world
#SBATCH --time=00:01:00         # hh:mm:ss for the job
#SBATCH --constraint="scratch"
#SBATCH -e slurm-%j.err
#SBATCH -o slurm-%j.out

module reset          # drop modules
module load openmpi   # load modules needed
module list           # job documentation and metadata
echo "job is starting on `hostname`"
srun hello
```

submitting a job

If the script is saved as `mpi_hello_world.slurm`,
submit the job with `sbatch`:

```
sbatch mpi_hello_world.slurm
```

Type `squeue | more` to see your job.

The output will be in a file with extension `out`
and error messages in the file with extension `err`.

Tasking with Julia

1. The NCSA Supercomputer Delta
   - using a real supercomputer

2. Programming Parallel Shared Memory Computers
   - tasking with Julia

3. Parallel Recursive Functions
   - the Fibonacci numbers
   - parallel recursive quadrature
   - parallel merge sort

4. Basic Linear Algebra Subprograms
   - multithreaded matrix multiplication

tasking with Julia

Julia is a new programming language for scientific computing designed for performance.

The tasking in Julia is inspired by parallel programming systems like Cilk, Intel Threading Building Blocks, and Go.

This lecture is based on a blogpost, of 23 July 2019,
`https://julialang.org/blog/2019/07/multithreading`
by Jeff Bezanson, Jameson Nash, and Kiran Pamnany,
as an early preview of Julia version 1.3.0.

Tasks are units of work, mapped to threads.

Tasking with Julia

the Fibonacci numbers

The sequence of Fibonacci numbers F_n are defined as

$$F_0 = 0, \quad F_1 = 1, \quad \text{and for } n > 1 : F_n = F_{n-1} + F_{n-2}.$$

This leads to a natural recursive function.

1. The recursion generates many function calls.
2. While inefficient to compute F_n,
   this recursion serves as a parallel pattern.

The parallel version is the opener of the blogpost.

a parallel recursive Fibonacci function

The Fibonacci function with tasking

- demonstrates the generation of a large number of tasks with one thread.
- No parallelism will result from this example.

But it is instructive to introduce basic task constructs.

- With `t = @spawn F()`
  we start a task `t` to compute `F()`, for some function `F`.
- The `fetch(t)` waits for `t` to complete
  and gets its return value.

command line arguments

```
# Shows the name of the Julia program
# and the command line arguments.

print(PROGRAM_FILE, " has ", length(ARGS))
println(" arguments.")
println("The command line arguments :")
for x in ARGS
    println(x)
end
```

the number of threads

If the file `showthreads.jl` contains

```
using Base.Threads

nbt = nthreads()
println("The number of threads : ", nbt)
```

then run via typing

```
JULIA_NUM_THREADS=8 julia showthreads.jl
```

at the command prompt. Alternatively, type

```
julia -t 8 showthreads.jl
```

a parallel recursive Fibonacci function

```
import Base.Threads.@spawn

function fib(n::Int)
    if n < 2
        return n
    end
    t = @spawn fib(n-2)
    return fib(n-1) + fetch(t)
end

if length(ARGS) > 0
    nbr = parse(Int64, ARGS[1])
    println(fib(nbr))
else
    println(fib(10))
end
```

about `fibmt.jl`

Run typing

```
time JULIA_NUM_THREADS=8 julia fibmt.jl 10
```

at the command prompt to compute the 10-th Fibonacci number with tasks mapped to 8 threads.

The recursive function `fib` illustrates the starting of a task and the synchronization of the sibling task.

- `t = @spawn fib(n-2)` starts a task to compute `fib(n-2)`
- `fetch(t)` waits for `t` to complete and gets its return value

There can not be any speedup because of the only computation, the `'+'` happens after the synchronization.

Tasking with Julia

parallel recursive quadrature

Apply a numerical integration rule $R(f, a, b, n)$ to $\int_a^b f(x)dx$.

The rule $R(f, a, b, n)$ takes on input

- the function f, bounds a, b of $[a, b]$, and
- the number n of function evaluations.

The rule returns and approximation A and an error estimate e.

If e is larger than some tolerance, then

1. $c = (b - a)/2$ is the middle of $[a, b]$,
2. compute $A_1, e_1 = R(f, a, c, n)$,
3. compute $A_2, e_2 = R(f, c, a, n)$,
4. return $A_1 + A_2, e_1 + e_2$.

This is the same pattern as Fibonacci.

the composite Trapezoidal rule applied recursively

Using n subintervals of $[a, b]$, the rule is

$$R(f, a, b, n) = \frac{h}{2}(f(a) + f(b)) + h \sum_{i=1}^{n-1} f(a + ih), \quad h = \frac{b - a}{n}.$$

Our setup: $f(x) = e^x$, $[a, b] = [0, 1]$, $\displaystyle\int_0^1 e^x dx = e - 1$.

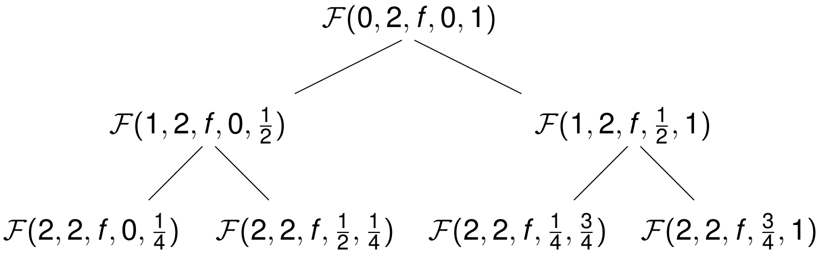
Keep n fixed. Let d be the depth of the recursion. The level is ℓ.

$\mathcal{F}(\ell, d, f, a, b, n)$:

If $\ell = d$ then
  return $R(f, a, b, n)$
else
  $c = (b - a)/2$
  return $\mathcal{F}(\ell+1, d, f, a, c, n) + \mathcal{F}(\ell+1, d, f, c, b, n)$.

the tree of function calls

The root of the tree is the first call, omitting the value for *n*.



At the leaves, the rule is applied.

As all computations are concentrated at the leaves,
we expect speedups from a parallel execution.

a recursive parallel integration function

```
function rectraprule(level::Int64,depth::Int64,
                     f::Function,a::Float64,
                     b::Float64,n::Int64)
    if level == depth
        return traprule(f,a,b,n)
    else
        middle = (b-a)/2

        t = @spawn rectraprule(level+1,depth, \
                               f,a,middle,n)
        return rectraprule(level+1,depth, \
                           f,middle,b,n) + fetch(t)
    end
end
```

runs with Julia 1.5.3 on pascal, depth = 4

```
$ time JULIA_NUM_THREADS=2 julia traprulerecmt.jl 4
1.718281828459045e+00
1.718281829227196e+00   error : 7.68e-10

real    0m5.207s
user    0m9.543s
sys     0m0.734s
$ time JULIA_NUM_THREADS=4 julia traprulerecmt.jl 4
1.718281828459045e+00
1.718281829227196e+00   error : 7.68e-10

real    0m3.120s
user    0m9.872s
sys     0m0.727s
$ time JULIA_NUM_THREADS=8 julia traprulerecmt.jl 4
1.718281828459045e+00
1.718281829227196e+00   error : 7.68e-10

real    0m1.985s
user    0m10.617s
sys     0m0.735s
$
```

Tasking with Julia

parallel merge sort

Merge sort works by divide and conquer, recursively as:

1. If no or one element, then return.
2. Split in two equal halves.
3. Sort the first half.
4. Sort the second half.
5. Merge the sorted halves.

The two above sort statements are recursive.

The sort algorithm will work in place, modifying the input, without returning. Instead of `fetch`, we use `wait`.

The `wait(t)` waits on task `t` to finish.

the function psort!

```
"""
Sorts the elements of v in place, from hi to lo.
"""
function psort!(v, lo::Int=1, hi::Int=length(v))
    if lo >= hi
        return v
    end
    if hi - lo < 100000          # no multithreading
        sort!(view(v, lo:hi), alg = MergeSort)
        return v
    end
```

The above code handles the base cases.

split and sort

The function continues:

```
mid = (lo+hi)>>>1              # find the midpoint

# task to sort the first half starts
half = @spawn psort!(v, lo, mid)

# runs with the current call below
psort!(v, mid+1, hi)

# wait for the lower half to finish
wait(half)
```

then next comes the merge ...

merging the sorted halves

```
    temp = v[lo:mid]       # workspace for merging
    i, k, j = 1, lo, mid+1 # merge the two sorted sub-arrays

    @inbounds while k < j <= hi
        if v[j] < temp[i]
            v[k] = v[j]
            j += 1
        else
            v[k] = temp[i]
            i += 1
        end
        k += 1
    end
    @inbounds while k < j
        v[i] = temp[i]
        k += 1
        i += 1
    end
    return v
end
```

the main function, with `@time`

```
"""
Calls the psort! once
to avoid compilation overhead.
"""
function main()
    a = rand(100)
    b = copy(a)
    psort!(b)
    a = rand(20000000)
    b = copy(a)
    @time psort!(b)
end

main()
```

runs with Julia 1.5.3 on pascal

```
$ for n in 1 2 4 8; do JULIA_NUM_THREADS=$n julia mergesortmt.jl; done
  2.219275 seconds (3.31 k allocations: 686.950 MiB, 3.34% gc time)
  1.439491 seconds (3.59 k allocations: 686.959 MiB, 6.41% gc time)
  0.920875 seconds (3.63 k allocations: 686.963 MiB, 3.90% gc time)
  0.625733 seconds (3.73 k allocations: 686.969 MiB, 4.45% gc time)
$
```

Compare to the wall clock time:

```
$ time JULIA_NUM_THREADS=8 julia mergesortmt.jl
  0.618549 seconds (3.72 k allocations: 686.969 MiB, 4.78% gc time)

real    0m1.220s
user    0m3.579s
sys     0m1.015s
$
```

Tasking with Julia

1. The NCSA Supercomputer Delta
   - using a real supercomputer

2. Programming Parallel Shared Memory Computers
   - tasking with Julia

3. Parallel Recursive Functions
   - the Fibonacci numbers
   - parallel recursive quadrature
   - parallel merge sort

4. Basic Linear Algebra Subprograms
   - multithreaded matrix multiplication

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 with Julia 1.5.3 on pascal

```
$ 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)
3.3312896110138868e11

julia> peakflops(16000)
3.4752698471120818e11

julia> peakflops(4000)
3.1302047295730540e11
```

Exercises

0. Use Delta to solve exercise 2 of Lecture 9.
1. Execute the recursive trapezoidal rule for different number of evaluations and increasing depths of recursion.
   For which values do you observe the best speedups?
2. Run the `peakflops` on your computer.
   For which dimension do you see the highest value?
   Compute the number of flops and relate this to the specifications of your computer.