Parallel processing

Agent-based modelling, Konstanz, 2024

Henri Kauhanen

21 May 2024

Parallel processing

- One way of making your code faster is to make use of parallel processing
- Here, **parallel processing** = when several simulation runs are computed simultaneously
- This is not always easy, and may not be worth doing unless you're running into serious efficiency issues...

A bit of history

- My first computer (mid-90s) had a 66 MHz processor
- My current office computer runs at 4.5 GHz – almost 70 times the frequency[1]
- Engineering problem/reality: the faster you make a processor run, the hotter it gets
- Engineering advance: smaller and smaller "lithographies" for processor manufacturing, meaning you can fit more transistors in a given surface area (Moore's Law)
- Put these two together and you have the idea of a **multi-core processor**

Parallel processing

- But now you have a **software challenge**: how do I make all processor cores work in parallel, if my code is a linear sequence of operations?
- In ABM, you often need to repeat a simulation many times.
  - Excellent use case for multi-core processors: each core can run an independent simulation in parallel
  - After every core has finished, we just collect the results

[1] It should be noted that processor frequency is *not* the only thing affecting the overall speed of a computer. There have also been improvements in memory speed, memory bus speed, the speed of storage media, and so on. These all play a role.

Distributed.jl

- In Julia, the *Distributed.jl* package makes this easy
- If your processor has, say, 4 cores and you want to engage them all, write:

```
using BenchmarkTools
using Distributed

addprocs(4)
```

- This makes Julia run in 4 parallel processes, each process being sent to an individual processor core

Example

- Let's simulate a population of 1000 `SimpleVL`s over 100,000 time steps:

```
@everywhere include("../jl/VL.jl")
@everywhere using .VL

@everywhere function simulation()
  pop = [SimpleVL(0.1, 0.01, 0.4, 0.1) for i in 1:1000]
  [interact!(rand(pop), rand(pop)) for t in 1:1_000_000]
end

simulation()
```

```
1000000-element Vector{Float64}:
 0.099
 0.099
 0.099
 0.099
 0.099
 0.099
 0.099
 0.099
 0.099
 0.099
 0.10900000000000001
 0.099
 0.099
 0.7756139769533772
 0.7165702959436901
 0.7283948757303422
 0.6614666780354128
 0.7214256766831326
 0.6796228748435917
 0.7444645825482015
 0.6440435681838119
 0.7210202889379991
 0.7387954982496935
 0.6210025622875553
 0.5896568573415677
```

- What if we want to repeat this 10 times?
- We can do it in series:

```julia
for rep in 1:10
  simulation()
end
```

- Or we can parallelize:

```julia
@distributed for rep in 1:10
  simulation()
end
```

- Here, each repetition gets assigned to a process/core. Once the process finishes, it becomes available for another repetition.
- The function definition additionally has to be prepended by the `@everywhere` macro, which makes the function available to

How much speed-up do we obtain?

```julia
@benchmark for rep in 1:10
  simulation()
end
```

```
BenchmarkTools.Trial: 2 samples with 1 evaluation.
 Range (min … max):  3.517 s …   3.913 s   GC (min … max): 11.54% … 20.68%
 Time  (median):     3.715 s               GC (median):    16.35%
 Time  (mean ± ):   3.715 s ± 279.864 ms   GC (mean ± ):  16.35% ±  6.46%



  3.52 s         Histogram: frequency by time         3.91 s <

 Memory estimate: 4.99 GiB, allocs estimate: 90010030.
```

```
#@benchmark @distributed for rep in 1:10
#  simulation()
#end
```

Thinking about the speed gain

- We've called upon 4 cores
- Why is the code not 4 times faster?
- Complex reasons, but, for example: the cores still share the same bus to memory, so there is an inevitable bottleneck

Collecting the results

- The output of every simulation is a vector of numbers, namely, the `p` field of a randomly chosen speaker
- We can collect these into a matrix, so that each vector becomes one of the columns of the matrix. This is accomplished with `hcat` ("horizontal catenate"):

```
#result = @distributed (hcat) for rep in 1:10
#  simulation()
#end
```

Amdahl's Law