Project 4: Introduction to Parallel Programming

Technical Computation in the Earth Sciences

19th October, 2020

Introduction.1: The need for parallel programming.

It has been observed that the number of transistors in a new CPU generally doubles about every two years, an observation termed “Moore’s Law.” For many years, from ~1970 until ~2004, the clock speed (in Hertz) of new CPUs also reliably doubled about every two years. Since 2004, however, there has been virtually no improvement in CPU clock speed.

This is sometimes attributed to “breakdown of Moore’s law”, but that isn’t technically correct – number of transistors on a CPU is still happily doubling away. Instead, what broke is something called *Dennard Scaling*, the relationship between transistor size and power consumption. Clock speed is limited, more or less, by how fast we can make a transistor switch on and off without it melting. Since making transistors smaller no longer makes them more power efficient, we can’t keep making them faster any more.

So instead, the extra transistors from Moore’s Law now go into making *more cores*. To take advantage of these new cores though, we have to write code that can run on multiple cores at the same time – AKA *in parallel*.

Introduction.2: Types of parallel programming.

- SIMD

Apply a single instruction to multiple data in the same CPU core

SIMD stands for “Single Instruction, Multiple Data”, which is a pretty good description of what it means: applying the same operations to multiple numbers at the same time. GPUs do this, but your CPU actually can too (albeit at a smaller scale).

```julia
using LoopVectorization
@avx for i = 1:N
    # do something
end
```

The only thing you have to be careful of here is that the individual iterations of the loop have to be *entirely independent of each other*. In other words, the `i=2` iteration can’t depend on the result of the `i=1` or any other previous iteration, and so on.

See also: Advanced Vector Extensions / Vector Processor

- Shared-memory parallel programming

Multiple tasks, on different CPU cores, but still all with access to the same memory

This is also fairly simple. In Julia, you can do this with `Base.Threads.@threads`. You can set the number of threads you want (up to the number of cores in your CPU) in your Juno ("Julia-client" package) or VSCode ("language-julia" extension) preferences, or from the command line by running `julia --threads 4` (or some other number). If you're wondering how many threads your Julia repl currently has access to, you can check with `Base.Threads.nthreads()`

```julia
Base.Threads.@threads for i = 1:N
    # do something
end
```

This will effectively partition the `for` loop into `Base.Threads.nthreads()` different smaller loops, each of which then runs on a different core of your CPU. Since you still have access to the same memory, your different threads can operate on different elements of the same array. The restrictions from above about the iterations of the loop being independent still apply. There is also some new overhead from scheduling and coordinating these threads.

In a compiled language like C or Fortran, you might do the same thing with OpenMP. Keep in mind though that this only works for as many cores as you have on a *single* CPU – if you want to coordinate between different CPUs, you'll need something like MPI instead.

- Distributed-memory parallel programming

Multiple tasks, possibly on multiple CPUs, each with their own separate memory

You don't have to use MPI for this problem set, but here's an example of the parallel version of "Hello World" in case you want to try it out.

```julia
using MPI

# Initialize
MPI.Init()

# Find out who and where we are
size = MPI.Comm_size(MPI.COMM_WORLD)
rank = MPI.Comm_rank(MPI.COMM_WORLD)
print("Hello from $rank of $size processors!\n")

# Finalize
MPI.Finalize()
```

To run an MPI program, you first have to install an MPI runtime (either OpenMPI or MPICH) and then run your program from the command line with (for Julia programs) something along the lines of `mpiexec -np N julia ./hello.jl` (where `N` is the number of MPI tasks you want). If your different MPI tasks have to talk to each other, then you have to explicitly have them talk to each other with functions like `MPI.send` and `MPI.recv`.

Pretty much all modern supercomputers, like those on the Top500, run programs that coordinate communication between nodes with MPI, though they may use other things within the node – which brings us to:

- "Hierarchical parallelism"

Sounds fancy, but just means combining multiple of the above.

1

Part 1

Consider two simple for loops operating on a large array A

```
for i = 1:length(A)
    A[i] = foo(A[i])
end
```

versus

```
A[1] = foo(A[1])
for i = 2:length(A)
    A[i] = foo(A[i]) / foo(A[i-1])
end
```

Which of these can be readily parallelized? Why or why not?

2

Part 2

Consider a simple function

```
function foo(x; N=10)
    for _ = 1:N # The _ is because this is a dummy variable
        x = sqrt(x)
    end
    return x
end
```

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

Generate a random 10000 by 10000 matrix

```
A = rand(10000,10000)
```

and write a normal function `apply foo` that loops through each dimension of A with plain `for` loops, applying `foo` to each element of A, and time the performance of your function with something along the lines of

```
using BenchmarkTools
@benchmark apply_foo_serial($A)
```

Should it matter in what order you index `A`? If it does, pick the fastest one.

Then make a new function with an `@avx` added in front of your `for` loop. Does this change the benchmark performance?

Finally, write a version of `apply_foo` that uses `Base.Threads.@threads`. Benchmark the run times again, adjusting the number of threads available from 1 to 8. How does the performance change as a function of the number of threads? Can you tell how many cores your CPU has from looking at a plot of run times versus number of threads?