HPCJulia

October 24, 2016

1 Parallelism and HPC Julia

Julia's documentation: http://docs.julialang.org/en/release-0.5/manual/parallel-computing/

In this notebook we will go over the different aspects of parallelism present in Julia.

1.1 SIMD

SIMD, Single Instruction Multiple Data, is a form of parallelism from executing multiple similar commands at once using specialized instructions in the processor. Julia applies SIMD automatically to loops and some functions due to the `-O3` optimization in its JIT compilation. However, SIMD can be explicitly added to a loop by using the `@simd` macro (note, this may slow down the calcuation. It is wise to let the auto-optimizer apply SIMD, and finer control of SIMD can be achieved using the SIMD.jl library.

Another form of multiple instruction is fused multiply add for calculations of type `a*b+c`. There are two forms present in Julia. The first is `muladd`. This is the recommended form for performance. `muladd(a,b,c)` will only apply a fused multiplication/addition if it will help with performance. On the otherhand, `fma(a,b,c)` will always apply fused multiplication/addition.

1.2 Multithreading

As of Julia v0.5, experimental multithreading is native to Julia via the `Threads.@threads` macro.

1.3 Distributed Parallelism

Julia's native parallelism is a distributed form of parallelism through TCPIP/ssh.

1.3.1 Libraries

The following libraries are helpful for solving parallel problems:

- DistributedArrays.jl
- ParallelDataTransfer.jl

2 Projects

2.1 Project 1: Getting Started with Distributed Parallelism

Use the following tutorial to test Julia's distributed parallelism:

http://www.stochasticlifestyle.com/multi-node-parallelism-in-julia-on-an-hpc/

2.2 Project 2: Coding a Distributed Algorithm

Extend your `least_squares` implementation to a distributed algorithm.

- Now generate a much larger `X` and `y`
- Use DistributedArrays.jl or ParallelDataTransfer.jl to evenly split the data amongst worker processes
- Apply the `@spawnat` macro to use the `least_squares` function on the remote processes
- Retrive the results of the `least_squares` algorithm, and average them together
- Now try a different approach using `pmap`

2.2.1 Benchmarks

- Benchmark the two codes on your computer (or cluster!). How does the performance scale with the number of processes? Look at http://www.stochasticlifestyle.com/236-2/
- Try to make a multithreaded version of the algorithm. How well does it benchmark? Check for type instabilities!

---

3 Extras

3.1 Job Scripts

3.1.1 UC Irvine Cluster (SGE)

In [ ]:
```
#!/bin/bash

#$ -N jbtest
#$ -q <Queue>
#$ -pe mpich 128
#$ -cwd                                # run the job out of the current directo
#$ -m beas
#$ -ckpt blcr
#$ -o output/
#$ -e output/
module load julia/0.4.3
julia --machinefile jbtest-pe_hostfile_mpich.$JOB_ID test.jl
```

3.1.2 XSEDE Comet (Slurm) Job Script

In [ ]:
```
#!/bin/bash
#SBATCH -A <account>
#SBATCH --job-name="juliaTest"
#SBATCH --output="juliaTest.%j.%N.out"
#SBATCH --partition=compute
#SBATCH --nodes=8
#SBATCH --export=ALL
#SBATCH --ntasks-per-node=24
```

```
#SBATCH -t 01:00:00
export SLURM_NODEFILE=`generate_pbs_nodefile`
./julia --machinefile $SLURM_NODEFILE /home/crackauc/test.jl
```

3.2 Test function

In [ ]:
```
## Script which prints out the hostnames of the worker processes
# Used in conjunction with a job script to ensure multi-node parallelism

hosts = @parallel for i=1:120
       run(`hostname`) end
println(hosts)
```