Homework 1 - EECE 5640

Andrew Tu

1

System: COE Cluster (iota)

- model CPU: Intel(R) Xeon(R) CPU X5650 @ 2.67GHz
- cores: 2 sockets, 6 core/socket, 2 threads/core
- memory: 47GB
- operating system: CentOs 7

Benchmark 1: HPCG

1. 45.2548
2. 45.2565
3. 45.1685
4. 45.0477
5. 45.314
6. 64.7114 <- Significantly different
7. 45.212
8. 45.2815
9. 45.2651
10. 45.2562

Avg: 47.17677

Run number 6 took significantly longer than the rest of runs. This could be a result of someone else logging into the system and contending for compute resources by running another program.

Because HPCG's out of the tar-file make file came preloaded with a number of optimizations, I looked at how removing certain operations affected runtime performance. HPCG shipped with the O3 flag already set. By removing the flag, the run time of the program increased to 126.17 seconds! Another flag I test removing was the `-funroll-loops` flag. This flag attempts to optimize code by replacing loops with statements, allowing the processor to take advantage of pipelining instructions to achieve some level of speedup. This increases spatial locality and reduces instruction branches at the expense of temporal locality. The tradeoff here is binary size. The resulting binary size will be increased and more data will need to be pulled through the cache hierarchy, potentially hindering performance. In our benchmark, removing the flag caused a slight increase in run time to 47.5 seconds, a 2 second increase over the previous runs (ignoring the outlier case).

CPU Benchmark

https://github.com/arihant15/Performance-Evaluation-Benchmark (1,000,000 FLOP, 1 thread)

1. 0.210108
2. 0.209205
3. 0.209714
4. 0.208517
5. 0.209399
6. 0.209827
7. 0.208531
8. 0.209278
9. 0.209998
10. 0.209862

Avg: 0.2094439

Running with the -O3 flag enabled brought run time down to ~.0002 seconds. The O3 flag is short hand for a number of optimization flags which help reduce execution time at the expense of compile time and possibly binary size. The increased binary size and aggressive compiler time "optimizations" actually resulted in a decrease in performance over the O2 flag which resulted in a run time of .00006s. Seeing the difference in performance of `-funroll-loops` for HPCG, I tried applying the same flag to this benchmark but did not see a similar increase in performance.

For this assignment, I tried including every flag set by the `O1-O3` flags as described by the gcc documentation to achieve the same level of performance shown by the `-OX` flag. Unfortunately, I didn't see any additional speedups. One of the explanations I found for this problem was that the `-OX` flags also set a number of other optimization flags not available through the gcc configurations. These flags operate under the hood and outside direct manipulation by the user. Given that I set every flag mentioned by the documentation for `O1-O3`, I believe the bulk of the speedup given by the flags come from under the hood optimizations.

If I were to parallelize this benchmark using pthreads, I would split the operations functions out to individual threads. As a result, running the benchmark with 1000 operations and 10 threads would split the work to 100 operations per thread. This method would work well for scaling the total amount of work done in a fixed amount of time since each thread will be responsible for the same amount of work, but the total work done increases.

2

Sample Sort Using PThreads to sort 5 million numbers

Time to Sort

1 thread	2 thread	4 thread	8 thread
4.1s	3.77s	3.41s	3.39s

One of the challenges I faced when writing this sort was determining how to manage serial operations. In this implementation, we are performing system calls to read and write our data to a file serially. Further we are also performing the data sampling serially, leaving the only the actual sort of broken down arrays to be performed in parallel. Since the code running in serial still takes a significant amount of time, the amount of speedup we can gain from the system as a whole is greatly limited (as per Amdahl's law). In order to know our program finished, I also had to use a barrier to allow the threads to re-synchronize before writing the results out to a file. It was also important to think about data race conditions while writing with pthreads. As a result, threads were allowed to read out of shared memory into their own private memory buffers. They were then allowed to write back to specific areas of another shared memory buffer, taking care not to change values in something that was being read by multiple threads.

Scaling Efficiency

Scaling	1 thread	2 thread	4 thread	8 thread
Strong	100%	54.37%	30.06%	15.12%
Weak	100%	108.7%	120.2%	120.9%

Running on coe cluster (iota): 2 socket, 6 core/socket, 2 threads/core

Equations:

- Strong Scaling: fixed total problem size; Efficiency: $(t1/N*tN)*100%
- Weak Scaling: fixed problem size per processor; Efficiency (t1/tN)*100%

This implementation does not have strong scaling capabilities. Running the sort on a smaller sample size (e.g. 10,000), there is too little performance gain to be seen by the timer. This is because the amount of work that can be parallelized was too small compared to the total work done to sort the numbers.

The weak scaling capabilities of the system are much better in this case. We see an increase in efficiency when we increase the cores from 1 to 2 to 4. The transition between 4 and 8 cores is marked by a sharp drop in speedup difference. This drop is unsurprising because each CPU only has 6 cores; therefore, attempting to spawn 8 threads will force data to be sent between CPUs over the high speed interconnect between sockets (if using 1 thread per core) OR cause two cores to be oversubscribed, hindering performance of 4 threads (2 per core).

3

hostname: c2130

- model: Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz

```
caches: L1d cache:              32K     -> unique to each core
        L1i cache:              32K     -> unique to each core
        L2 cache:               256K    -> unique to each core
        L3 cache:               20480K  -> shared bewteen all
```

- 2 sockets, 8 core per socket, 2 thread per core
- OS: CentOS 7 (`cat /etc/os-release`)
- Network: 82599ES 10-Gigabit SFI/SFP+ Network Connection, Latency: ???

4

Trends:

- significant dropoff in peak performance from top 3 to rest of the top 10
  - 90 petaflops to ~20 petaflops (ignoring Tianhe2A @ 60 petaflops)
- the two top 10 Chinese based machines have significantly more cores and higher power consumption
- top 2 machines use IBM POWER9 CPUs with infinaband interconnect with fat tree topology
- heterogenous architectures (2 sockets + accelerators)/node
  - NVIDIA V100: Summit (6), Sierra (4), ABCI (4))
  - NVIDIA P100: Piaz Daint (1)
  - Intel Xeon CoProcessors:Tianhe (3)
- 2 types of nodes
  - login nodes (low compute. just an entry point)
  - compute nodes (high power, run computation)
- interconnects
  - Infinaband interconnect

  - Intel Omni Path
  - Cray Aries interconnect
- Topologies
  - Fat Tree Topologies
  - Dragonfly
- Utilize SSD's over hard disks

Designing a SC

Processing Power

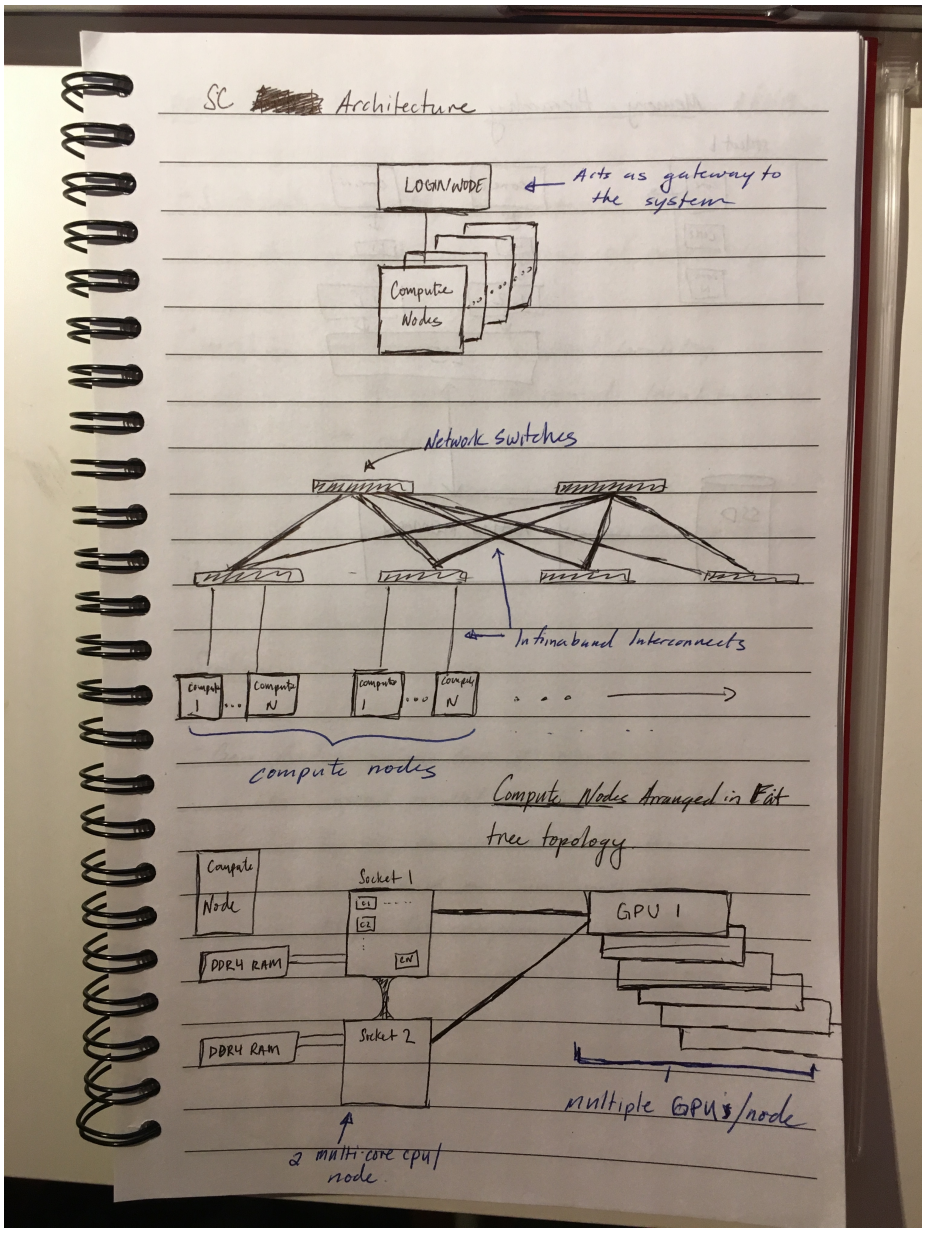
- 2 multicore CPUs **per node**: Having more powerful CPUs that can handle many threads is important for high performance.
- Multiple GPUs **per node** (or equivalent processors): takes significant advantage of parallelism within problems.
- NUMA memory access.

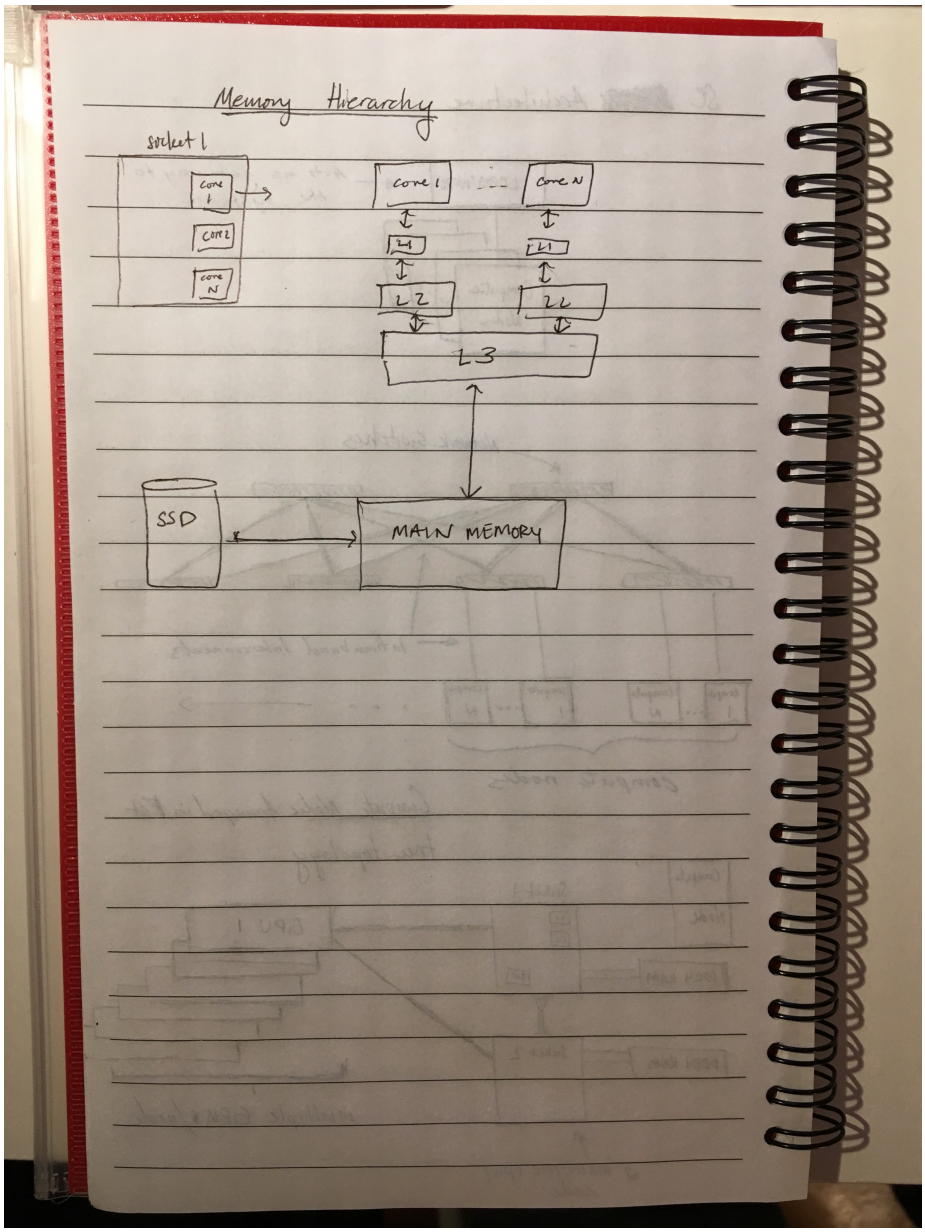
Interconnect

- Infinaband interconnect for HIGH speed data transfer.
- Nodes are arranged in a fat tree topology to minimize the likelihood of bottleneck.

Login Nodes & Compute nodes

- have a dedicated login node to prevent compute nodes from being bogged down by people signing in and launch jobs. Signing in and launching jobs from a dedicated node can streamline the process


SC Architecture
LOGIN/NODE
Acts as gateway to the system
Compute Nodes
Network switches
Infiniband Interconnects
Compute 1
Compute N
Compute 1
Compute N
compute nodes
Compute Nodes Arranged in Fat tree topology
Compute Node
Socket 1
C1
C2
CN
GPU 1
DDR4 RAM
DDR4 RAM
Socket 2
multiple GPUs/node
2 multi-core cpu/ node


Memory Hierarchy
socket 1
core 1
core 2
core N
core 1
core N
L1
L1
L2
L2
L3
SSD
MAIN MEMORY