Derek Hernandez (djh119)

1)

A)

Number of cores: 2
Threads per core: 2 (total 4)
L1 instruction cache size: 32768
L1 data cache size: 32768
L2 cache size: 262144
CPU frequency(e.g. base, turbo, max): base: 2.3 Ghz ,turbo: 3.6 Ghz

Command Line: sysctl hw

```
hw.ncpu: 4
hw.byteorder: 1234
hw.memsize: 8589934592
hw.activecpu: 4
hw.physicalcpu: 2
hw.physicalcpu_max: 2
hw.logicalcpu: 4
hw.logicalcpu_max: 4
hw.cputype: 7
hw.cpusubtype: 8
hw.cpu64bit_capable: 1
hw.cpufamily: 260141638
hw.cacheconfig: 4 2 2 4 0 0 0 0 0 0
hw.cachesize: 8589934592 32768 262144 4194304 0 0 0 0 0 0
hw.pagesize: 4096
hw.pagesize32: 4096
hw.busfrequency: 100000000
hw.busfrequency_min: 100000000
hw.busfrequency_max: 100000000
hw.cpufrequency: 2300000000
hw.cpufrequency_min: 2300000000
hw.cpufrequency_max: 2300000000
hw.cachelinesize: 64
hw.l1icachesize: 32768
hw.l1dcachesize: 32768
hw.l2cachesize: 262144
hw.l3cachesize: 4194304
hw.tbfrequency: 1000000000
hw.packages: 1
hw.optional.floatingpoint: 1
hw.optional.mmx: 1
hw.optional.sse: 1
hw.optional.sse2: 1
hw.optional.sse3: 1
hw.optional.supplementalsse3: 1
hw.optional.sse4_1: 1
hw.optional.sse4_2: 1
hw.optional.x86_64: 1
hw.optional.aes: 1
hw.optional.avx1_0: 1
hw.optional.rdrand: 1
```

Derek Hernandez (djh119)

```
hw.optional.f16c: 1
hw.optional.enfstrg: 1
hw.optional.fma: 1
hw.optional.avx2_0: 1
hw.optional.bmi1: 1
hw.optional.bmi2: 1
hw.optional.rtm: 1
hw.optional.hle: 1
hw.optional.adx: 1
hw.optional.mpx: 0
hw.optional.sgx: 0
hw.optional.avx512f: 0
hw.optional.avx512cd: 0
hw.optional.avx512dq: 0
hw.optional.avx512bw: 0
hw.optional.avx512vl: 0
hw.optional.avx512ifma: 0
hw.optional.avx512vbmi: 0
hw.targettype: Mac
hw.cputhreadtype: 1
```

B) (attached)
C)

N	Row Compute time:	Mega_elements/sec:	Column Compute time:	Mega_elements/sec:
500	0.0007	368.155	0.0007	367.154
10000	0.2120	471.714	0.2107	474.584
15000	0.4829	465.896	0.4516	498.280
20001	0.9410	425.142	0.8065	496.043
50000	65.0239	-27.605	73.0901	-24.558

D) To simulate warm vs cold cache, I decided to loop through MatricOps with preset sizes and loop. It looped exactly 3 times through the code, and I averaged each respective size from the loops. What I can conclude is that it does make a difference in time towards the end of the loop, the time starts to increase quickly under the Row Compute time.

Test:
Project 6 Array Traversal: Array size = 500 Bytes
by row compute time: 0.0005 seconds mega_elements/sec: 486.400
by column compute time: 0.0007 seconds mega_elements/sec: 364.491

Project 6 Array Traversal: Array size = 10000 Bytes
by row compute time: 0.2083 seconds mega_elements/sec: 480.097
by column compute time: 0.2168 seconds mega_elements/sec: 461.231

Project 6 Array Traversal: Array size = 15000 Bytes
by row compute time: 0.4695 seconds mega_elements/sec: 479.196
by column compute time: 0.4564 seconds mega_elements/sec: 493.028

Derek Hernandez (djh119)

Project 6 Array Traversal: Array size = 18000 Bytes
by row compute time: 0.7604 seconds mega_elements/sec: 426.114
by column compute time: 0.6591 seconds mega_elements/sec: 491.573

Project 6 Array Traversal: Array size = 20001 Bytes
by row compute time: 1.8625 seconds mega_elements/sec: 214.783
by column compute time: 0.8217 seconds mega_elements/sec: 486.843
Test 1 Done:
Test:
Project 6 Array Traversal: Array size = 500 Bytes
by row compute time: 0.0005 seconds mega_elements/sec: 486.333
by column compute time: 0.0005 seconds mega_elements/sec: 497.969

Project 6 Array Traversal: Array size = 10000 Bytes
by row compute time: 0.4661 seconds mega_elements/sec: 214.557
by column compute time: 0.2038 seconds mega_elements/sec: 490.706

Project 6 Array Traversal: Array size = 15000 Bytes
by row compute time: 1.2807 seconds mega_elements/sec: 175.690
by column compute time: 0.4572 seconds mega_elements/sec: 492.139

Project 6 Array Traversal: Array size = 18000 Bytes
by row compute time: 1.8283 seconds mega_elements/sec: 177.213
by column compute time: 0.6873 seconds mega_elements/sec: 471.405

Project 6 Array Traversal: Array size = 20001 Bytes
by row compute time: 2.2676 seconds mega_elements/sec: 176.418
by column compute time: 0.8235 seconds mega_elements/sec: 485.783
Test 2 Done:
Test:
Project 6 Array Traversal: Array size = 500 Bytes
by row compute time: 0.0010 seconds mega_elements/sec: 262.576
by column compute time: 0.0007 seconds mega_elements/sec: 344.393

Project 6 Array Traversal: Array size = 10000 Bytes
by row compute time: 0.2104 seconds mega_elements/sec: 475.181
by column compute time: 0.2150 seconds mega_elements/sec: 465.192

Project 6 Array Traversal: Array size = 15000 Bytes
by row compute time: 1.2914 seconds mega_elements/sec: 174.234
by column compute time: 0.5002 seconds mega_elements/sec: 449.805

Project 6 Array Traversal: Array size = 18000 Bytes
by row compute time: 2.1508 seconds mega_elements/sec: 150.644
by column compute time: 0.7808 seconds mega_elements/sec: 414.962

Project 6 Array Traversal: Array size = 20001 Bytes
by row compute time: 2.3582 seconds mega_elements/sec: 169.636
by column compute time: 0.8586 seconds mega_elements/sec: 465.933
Test 3 Done:

2)

A) I found that there are multiple occurrences of a number other than 40,000. The number is 30,000 (erroneous value), and it occurs 10 times out of the 10,000 totals. Frequency: 10/10,000 = .1%. The reason for these erroneous values could be due to each thread working on its own value, and the returning the value in a random order independently.

Derek Hernandez (djh119)

Yes, the lock works. There are no erroneous values in the .out file. There are only 40,000 values throughout .out file

B)

* dotprod_mutex.c is with comments enabled, preventing the lock. If the lock was enabled, then it would average an additional .5 Milliseconds on top of the current time.

.C file	Time MIliSeconds
dotprod_serial.c	.87000
dotprod_mutex.c	2.0078

3)

A)

* Running on eros, I had errors with -fopenmp on my mac and couldn't compile. When trying to solve the issues it was most likely due to the version of XCode's -clang. Which I was unable to update.

```
DOT_PRODUCT
  C/OpenMP version

  A program which computes a vector dot product.

  Number of processors available = 4
  Number of threads =              4

  Sequential      1000    1.000000e+03     0.0000114073
  Parallel        1000    1.000000e+03     0.0041714441

  Sequential     10000    1.000000e+04     0.0000578160
  Parallel       10000    1.000000e+04     0.0038754619

  Sequential    100000    1.000000e+05     0.0005396809
  Parallel      100000    1.000000e+05     0.0040634647

  Sequential   1000000    1.000000e+06     0.0060330280
  Parallel     1000000    1.000000e+06     0.0120190568

DOT_PRODUCT
  Normal end of execution.
```

Derek Hernandez (djh119)

	T1 / P1	T1/(4*P1)
Vectors	Speed Up	Efficiency
1000	.00273	.00068
10000	.01492	.00373
100000	.13281	.03320
1000000	.50196	.12549

B)

pragma omp parallel shared (n, x, y) private (i)

- "i" is private because its private to each thread and is assigned to each thread.

pragma omp for reduction (+ : xdoty)

- This allows for multiple threads to split work of the variable, where each will contribute towards the final value. It's a way to split work, and similar to a checksum.