Report

December 17, 2019

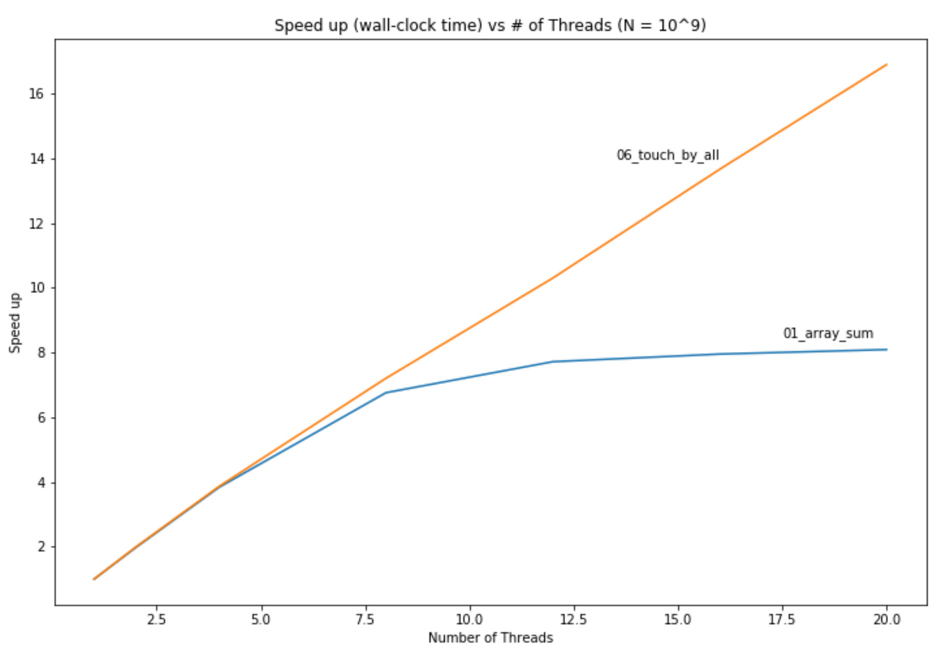
1 Exercise 0

1.1 Strong Scaling Test

1-measure the time-to-solution of the two codes in a strong-scaling test (use some meaningful value for N, like 10^9), using from 1 (using the serial version) to N_c cores on a node;

In order to test the strong scaling of codes, $N = 10^9$ chosen and each program was run using from 1 (serial version) to 20 threads. Both wall-clock time and elapsed time considered and tested for each step.

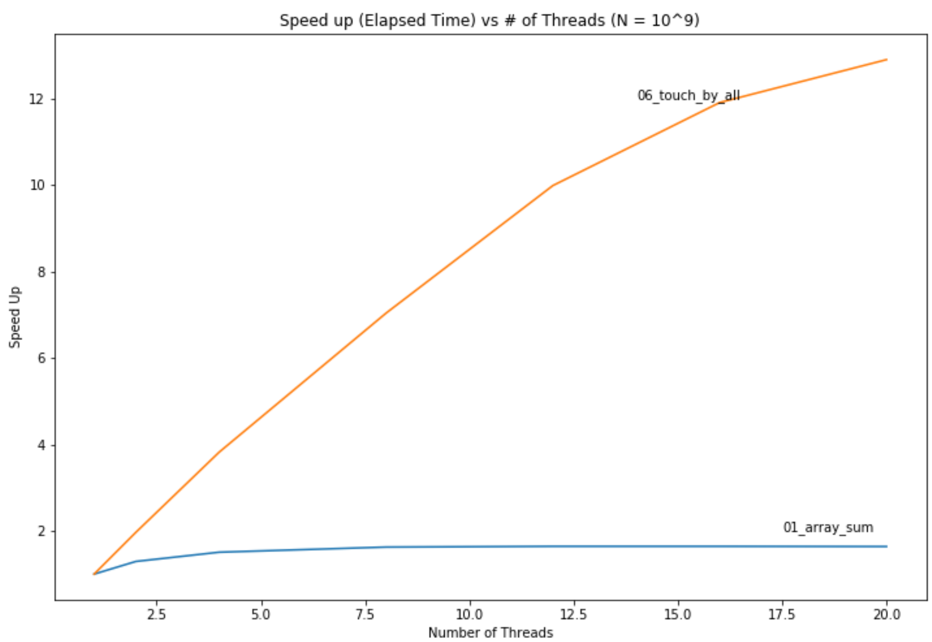
Speed Up (Wall-Clock) / # Threads	1	2	4	8	12	16	20
Touch By First	1	1.976	3.839	6.76	7.717	7.953	8.091
Touch By All	1	1.993	3.868	7.207	10.302	13.662	16.896



Above plot shows us, touch by all and touch by first scale similar up to 4 threads, after this point touch by first stopped scaling and touch by all continued to scale.

Elapsed Time / # Threads	1	2	4	8	12	16	20
Touch By First	6.19	4.79	4.11	3.81	3.77	3.77	3.78
Touch By All	6.19	3.15	1.62	0.88	0.62	0.52	0.48

Speed Up (Elapsed Time) / # Threads	1	2	4	8	12	16	20
Touch By First	1	1.292	1.506	1.625	1.642	1.642	1.638
Touch By All	1	1.965	3.821	7.034	9.984	11.904	12.896



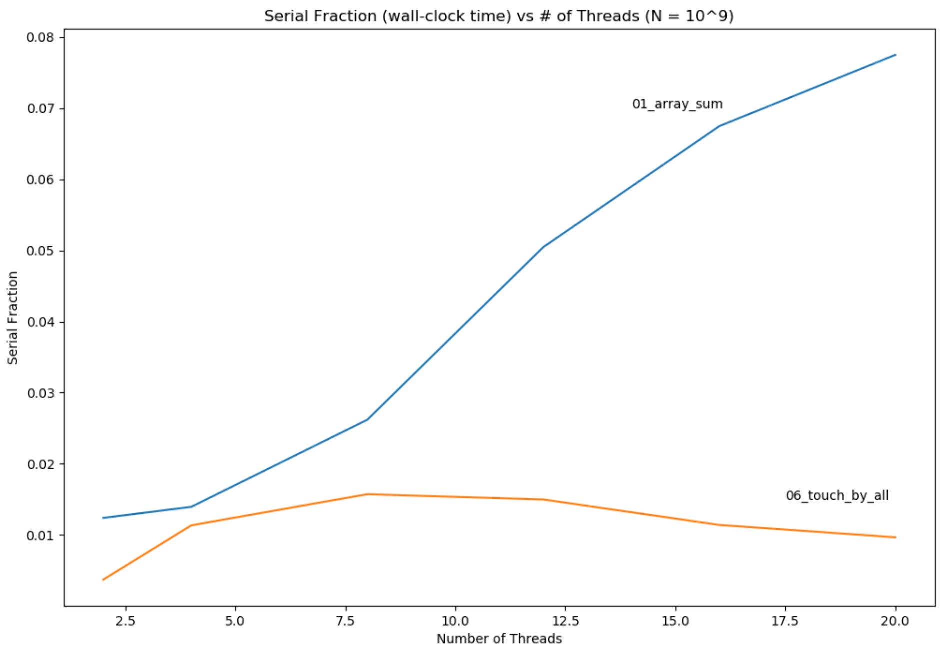
Above plot shows us touch by all scales better than touch by first in terms of elapsed times. Actually touch by first doesn't scale at all. On the other hand touch by all scales very good up to 16 threads but after this point, appropriately to the Amdahl's Law it stoped scaling.

1.2 Parallel Overhead

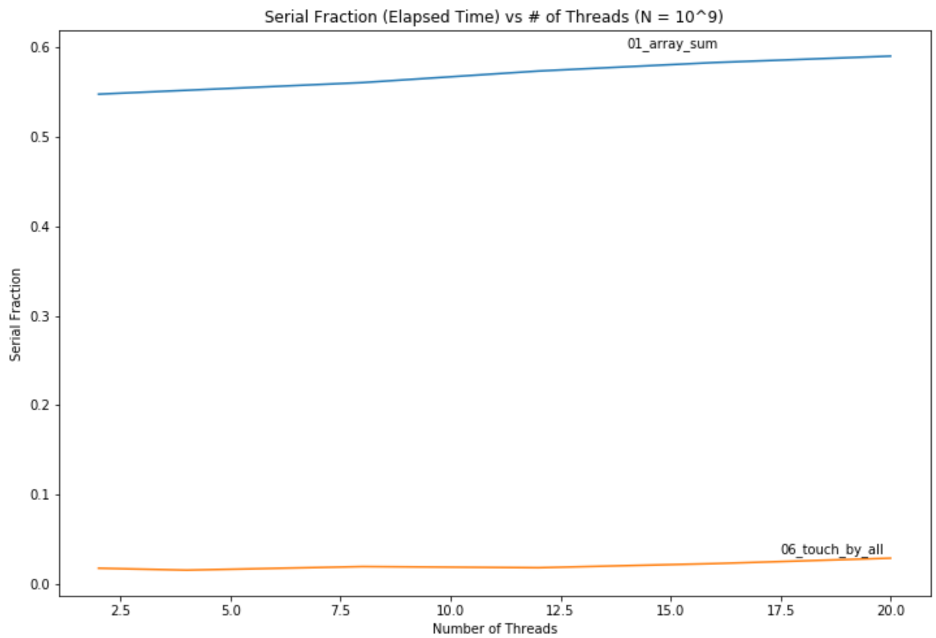
2-measure the parallel overhead of both codes, from 2 to N_c cores on a node;

For measuring overhead two methods applied. First for both code serial fractions are calculated to understand if there is overhead or not. After that to estimate it one of the optimistic formula is used.

Serial Fraction (wct) / # Threads	2	4	8	12	16	20
Touch By First	0.012	0.014	0.026	0.05	0.067	0.077
Touch By All	0.004	0.011	0.016	0.015	0.011	0.01



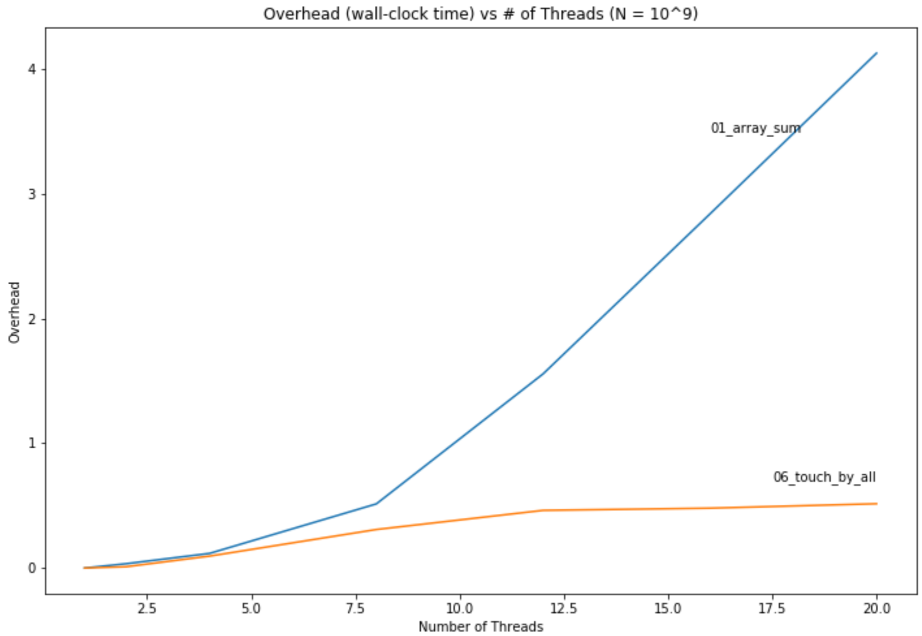
As it can be undestood from the plot touch by all has lower serial fraction than touch by first and serial fraction of touch by first increase. However touch by all is more stable.



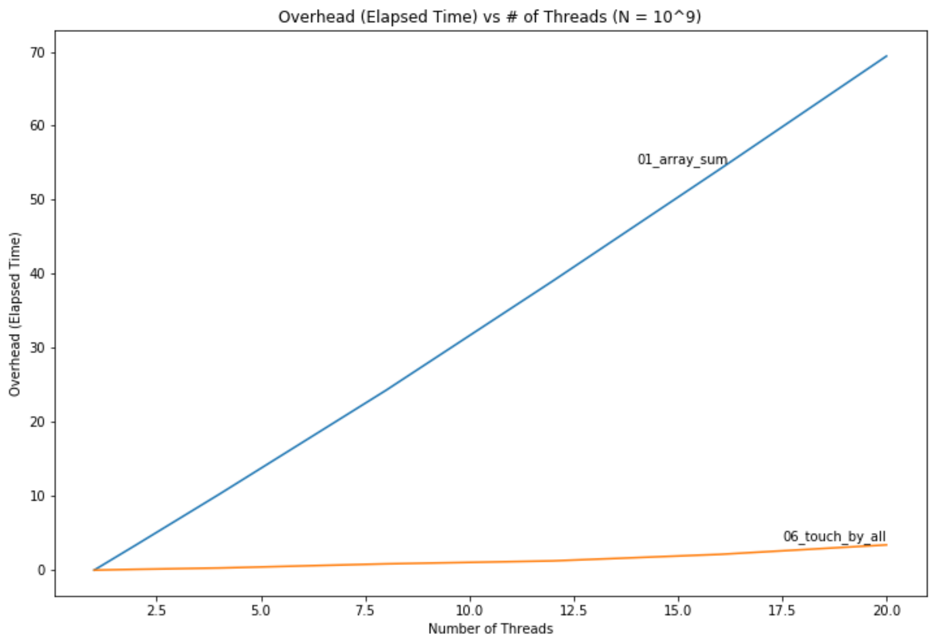
Serial Fraction (elapsed) / # Threads	2	4	8	12	16	20
Touch By First	0.548	0.552	0.561	0.574	0.583	0.59
Touch By All	0.018	0.016	0.02	0.018	0.023	0.029

According to serial fraction calculations change (increasing serial fraction means lack of scaling is also due to the parallelization overhead) on the other hand if it is stable lack of scaling is due to the serial workload

In order to measure estimated overhead for the codes, I have used general formula, overhead function $T_o = p \times T_p - T_S$ reference (page 2)



Overhead (wct) / # Threads	1	2	4	8	12	16	20
Touch By First	0	0.035	0.117	0.514	1.556	2.837	4.127
Touch By All	0	0.01	0.096	0.309	0.462	0.48	0.515



Overhead (Elapsed Time) / # Threads	1	2	4	8	12	16	20
Touch By First	0	3.39	10.25	24.29	39.05	54.13	69.41
Touch By All	0	0.11	0.29	0.85	1.25	2.13	3.41

In terms of overhead with the increasing number of computation units touch by first method's overhead increase too much so this situation shows us touch by all method is more efficient than touch by first method. In order to understand this difference, deeper analyze must be performed. According to this context these codes will be profilied by using perf.

1.3 Profiling Codes

3-provide any relevant metrics that explain any observed difference;

In order to specify difference between two codes, I have used perf to profile codes (collecting hardware, software events). There is no significant difference between two codes. In order to get statistically significant results data collection repeated 10 times. Results are from Ulysses (20 threads)

```
**Ulysses Computational Node** (with 20 threads)

 Performance counter stats for './01_array_sum_parallel.x 1000000000' (10 runs):

      33845418144 cpu-cycles                #    3.226 GHz                     ( +-  0.12% ) [80.02%]
      27367678869 instructions              #    0.81  insns per cycle
                                            #    0.93  stalled cycles per insn  ( +-  0.05% ) [89.99%]
        180765348 cache-references          #   17.229 M/sec                   ( +-  0.06% ) [89.93%]
        150130073 cache-misses              #   83.052 % of all cache refs     ( +-  0.02% ) [89.94%]
       2086147013 branch-instructions       #  198.836 M/sec                   ( +-  0.10% ) [89.96%]
           119766 branch-misses             #    0.01% of all branches         ( +-  0.83% ) [90.00%]
      25395178982 stalled-cycles-frontend   #   75.03% frontend cycles idle    ( +-  0.16% ) [90.02%]
   <not supported> stalled-cycles-backend
     10491.858119 cpu-clock                                                    ( +-  0.12% )
     10491.814773 task-clock                #    2.762 CPUs utilized           ( +-  0.12% )
   <not supported> L1-dcache-loads
        380863046 L1-dcache-load-misses     #    0.00% of all L1-dcache hits   ( +-  0.03% ) [90.04%]
         84430448 LLC-loads                 #    8.047 M/sec                   ( +-  0.12% ) [90.05%]
         58937920 LLC-load-misses           #   69.81% of all LL-cache hits    ( +-  0.11% ) [90.05%]

      3.798835260 seconds time elapsed                                         ( +-  0.12% )
```

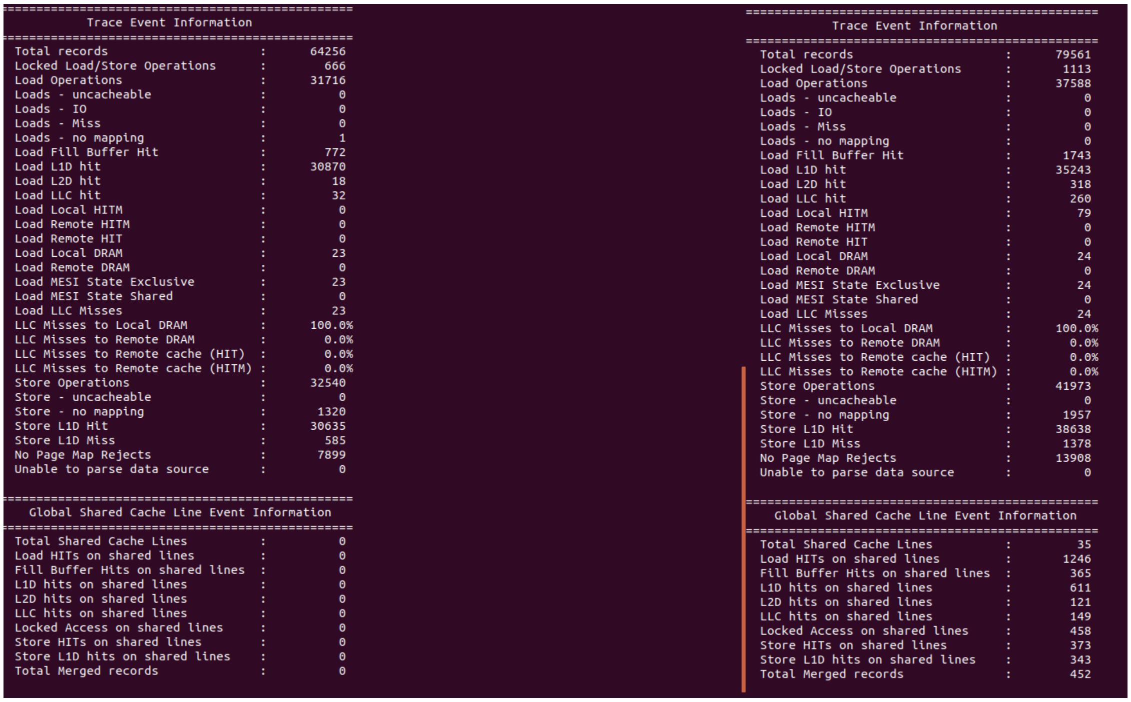
```
 Performance counter stats for './06_touch_by_all_parallel.x 1000000000' (10 runs):

      28103325429 cpu-cycles                #    3.066 GHz                     ( +-  0.67% ) [80.00%]
      27395801411 instructions              #    0.97  insns per cycle
                                            #    0.72  stalled cycles per insn  ( +-  0.10% ) [89.98%]
        142379842 cache-references          #   15.533 M/sec                   ( +-  0.17% ) [89.94%]
        132912631 cache-misses              #   93.351 % of all cache refs     ( +-  0.12% ) [89.95%]
       2099494448 branch-instructions       #  229.042 M/sec                   ( +-  0.23% ) [89.97%]
           136217 branch-misses             #    0.01% of all branches         ( +-  0.84% ) [89.99%]
      19606804162 stalled-cycles-frontend   #   69.77% frontend cycles idle    ( +-  1.01% ) [90.01%]
   <not supported> stalled-cycles-backend
      9166.426081 cpu-clock                                                    ( +-  0.72% )
      9166.408297 task-clock                #   17.454 CPUs utilized           ( +-  0.72% )
   <not supported> L1-dcache-loads
        380786971 L1-dcache-load-misses     #    0.00% of all L1-dcache hits   ( +-  0.04% ) [90.03%]
         13799532 LLC-loads                 #    1.505 M/sec                   ( +-  2.06% ) [90.05%]
          9623979 LLC-load-misses           #   69.74% of all LL-cache hits    ( +-  2.03% ) [90.07%]

      0.525183551 seconds time elapsed                                         ( +-  2.25% )
```

Since there is no significant difference between two codes. Chosen events and small differences will be explained.

- Best case for cpu-cycle and instructions must be multiple instructions are executed in a single cycle so in terms of instruction per cycle touch by all policy is better than touch by first but there is no big difference.
- In terms of cache misses results for both code is similar even touch by first is bit better than touch by all. It seems that both codes don't perform well about cache so it cause execution delays by requiring the program to fetch the data from other cache level. (perf c2c will be used to analyze this event better)
- Branch instructions and misses are more or less same for each codes and it looks efficient.
- The cycles stalled in the front-end are waste because front-end doesn't feed the back-end but for these two code percentage is more or less same
- Task clock shows time spent on the profiled task. So in this context we can say that touch by all policy is way better than touch by first since utilizaiton of CPU (with 20 threads) in other words parallelization of touch by all (usage of threads) are more efficient than touch by first.
- To understand better, difference between two method perf c2c command used. In Ulyses perf c2c command can not be used so I used it in my local computer (with 8 threads). You can find my computers architecture and compiler info in readme file.



```
=================================================
            Trace Event Information
=================================================
  Total records                     :      64256
  Locked Load/Store Operations      :        666
  Load Operations                   :      31716
  Loads - uncacheable               :          0
  Loads - IO                        :          0
  Loads - Miss                      :          0
  Loads - no mapping                :          1
  Load Fill Buffer Hit              :        772
  Load L1D hit                      :      30870
  Load L2D hit                      :         18
  Load LLC hit                      :         32
  Load Local HITM                   :          0
  Load Remote HITM                  :          0
  Load Remote HIT                   :          0
  Load Local DRAM                   :         23
  Load Remote DRAM                  :          0
  Load MESI State Exclusive         :         23
  Load MESI State Shared            :          0
  Load LLC Misses                   :         23
  LLC Misses to Local DRAM          :     100.0%
  LLC Misses to Remote DRAM         :       0.0%
  LLC Misses to Remote cache (HIT)  :       0.0%
  LLC Misses to Remote cache (HITM) :       0.0%
  Store Operations                  :      32540
  Store - uncacheable               :          0
  Store - no mapping                :       1320
  Store L1D Hit                     :      30635
  Store L1D Miss                    :        585
  No Page Map Rejects               :       7899
  Unable to parse data source       :          0

=================================================
    Global Shared Cache Line Event Information
=================================================
  Total Shared Cache Lines          :          0
  Load HITs on shared lines         :          0
  Fill Buffer Hits on shared lines  :          0
  L1D hits on shared lines          :          0
  L2D hits on shared lines          :          0
  LLC hits on shared lines          :          0
  Locked Access on shared lines     :          0
  Store HITs on shared lines        :          0
  Store L1D hits on shared lines    :          0
  Total Merged records              :          0
```

```
=================================================
            Trace Event Information
=================================================
  Total records                     :      79561
  Locked Load/Store Operations      :       1113
  Load Operations                   :      37588
  Loads - uncacheable               :          0
  Loads - IO                        :          0
  Loads - Miss                      :          0
  Loads - no mapping                :          0
  Load Fill Buffer Hit              :       1743
  Load L1D hit                      :      35243
  Load L2D hit                      :        318
  Load LLC hit                      :        260
  Load Local HITM                   :         79
  Load Remote HITM                  :          0
  Load Remote HIT                   :          0
  Load Local DRAM                   :         24
  Load Remote DRAM                  :          0
  Load MESI State Exclusive         :         24
  Load MESI State Shared            :          0
  Load LLC Misses                   :         24
  LLC Misses to Local DRAM          :     100.0%
  LLC Misses to Remote DRAM         :       0.0%
  LLC Misses to Remote cache (HIT)  :       0.0%
  LLC Misses to Remote cache (HITM) :       0.0%
  Store Operations                  :      41973
  Store - uncacheable               :          0
  Store - no mapping                :       1957
  Store L1D Hit                     :      38638
  Store L1D Miss                    :       1378
  No Page Map Rejects               :      13908
  Unable to parse data source       :          0

=================================================
    Global Shared Cache Line Event Information
=================================================
  Total Shared Cache Lines          :         35
  Load HITs on shared lines         :       1246
  Fill Buffer Hits on shared lines  :        365
  L1D hits on shared lines          :        611
  L2D hits on shared lines          :        121
  LLC hits on shared lines          :        149
  Locked Access on shared lines     :        458
  Store HITs on shared lines        :        373
  Store L1D hits on shared lines    :        343
  Total Merged records              :        452
```

According to above image (perf c2c results), touch by all (right) seems more efficient than touch by first (left) because the cache of each thread is warmed-up with the data. Cache hits are better for touch by all. There is no global shared cache line event for touch by first but touch by all has this event and naturally performs better.

1.4 Optional Part

figure out how you could allocate and correctly initialise the right amount of memory separately on each thread

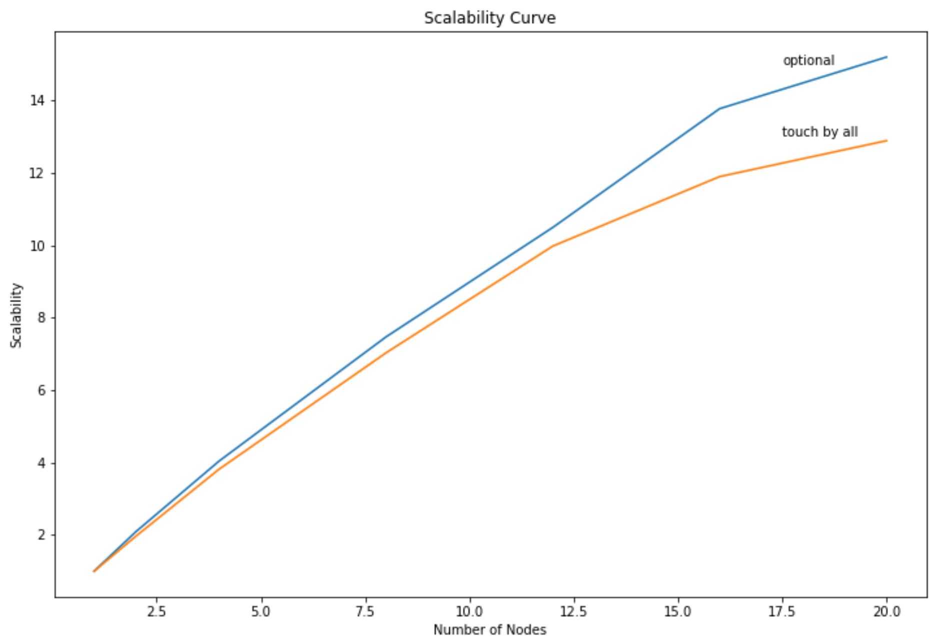
For the optional part, I realized that opening two parallel region is inefficient and increase overhead so I combined them in one parallel region. Also I have used schedule directives and most suitable one is guided with chunk size 50. This change gave me smaller elapsed times but since I am initializing and summing the array in the same for loop wall-clock time increased. Also question is specifically about initializing array so I ignored the summing up without initializing it.

1.4.1 Compare Strong Scalability with Touch By All

Elapsed Time / # Threads	1	2	4	8	12	16	20
Optional	4.41	2.11	1.09	0.59	0.42	0.32	0.29
Touch By All	6.19	3.15	1.62	0.88	0.62	0.52	0.48

Speed Up / # Threads	1	2	4	8	12	16	20
OpenMP	1	2.09	4.046	7.475	10.5	13.781	15.207

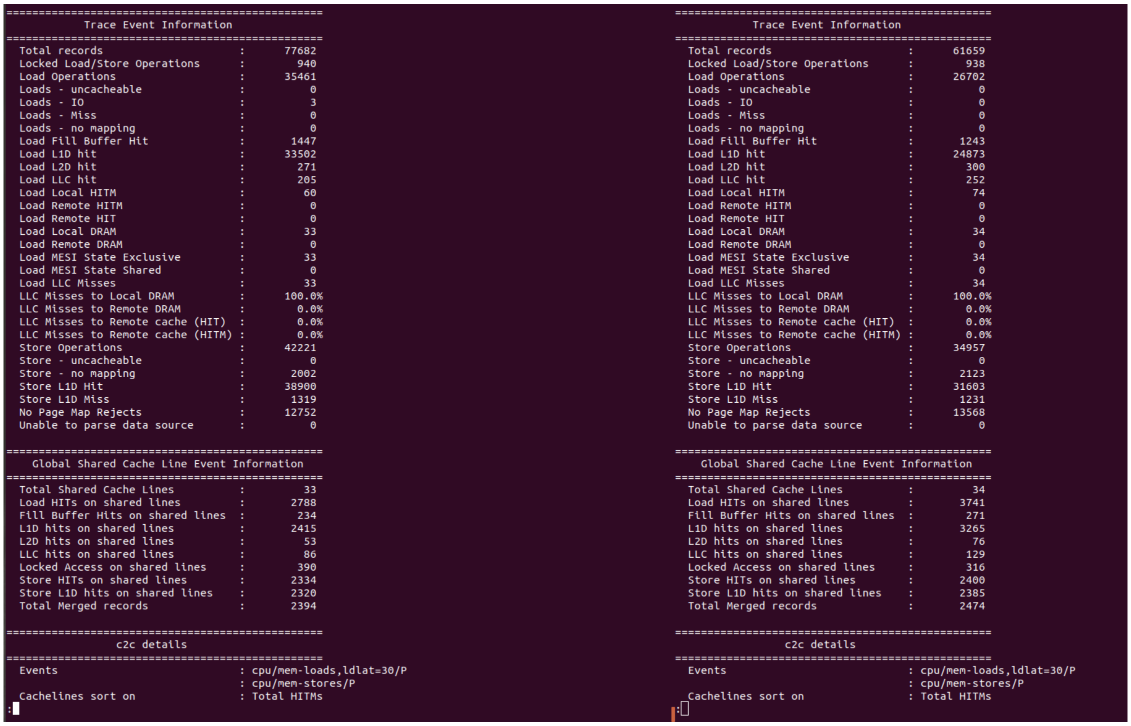
Speed Up / # Threads	1	2	4	8	12	16	20
Touch By All	1	1.965	3.821	7.034	9.984	11.904	12.896



So it can be seen that, by using schedule (guided) and combining parallel loops made same program more efficient. It scaled super linearly up to 8 threads even after that it is more efficient than touch by all.

1.4.2 Perf Cache Events

To see the difference between codes, I have used perf c2c record and report command in my computer.



As it can be seen, optional (right side) is more efficient global shared cache hits (except L2 cache hits)

2 Exercise 1

I tried to openmp-ize serial application of monte carlo pi. After few trials and modifications I got better run times than serial one. During development process generating random numbers were bit hard, first I applied standard routines but estimations of pi was terrible. That is why I changed the way and with the help of Appendix 1 and some google search (drand48_r function requires structure) I could obtain better code. Also opening regular parallel regions (without specifying private, shared) doesn't give better results than the serial one.

Refer the code openmp_pi.c

2.1 Weak and Strong Scalability

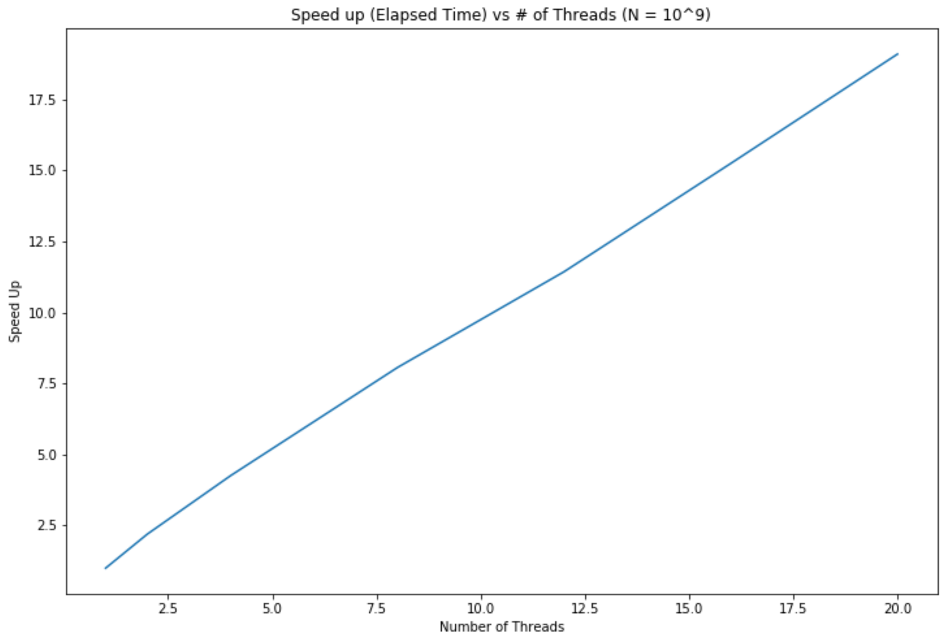
1-establish its weak and strong scalability;

2.1.1 Strong Scalability

Since elapsed time and walltime is more or less same, for this tests I'll only use elapsed time.

Elapsed Time / # Threads	1	2	4	8	12	16	20
OpenMP	19.66	8.95	4.62	2.44	1.72	1.29	1.03

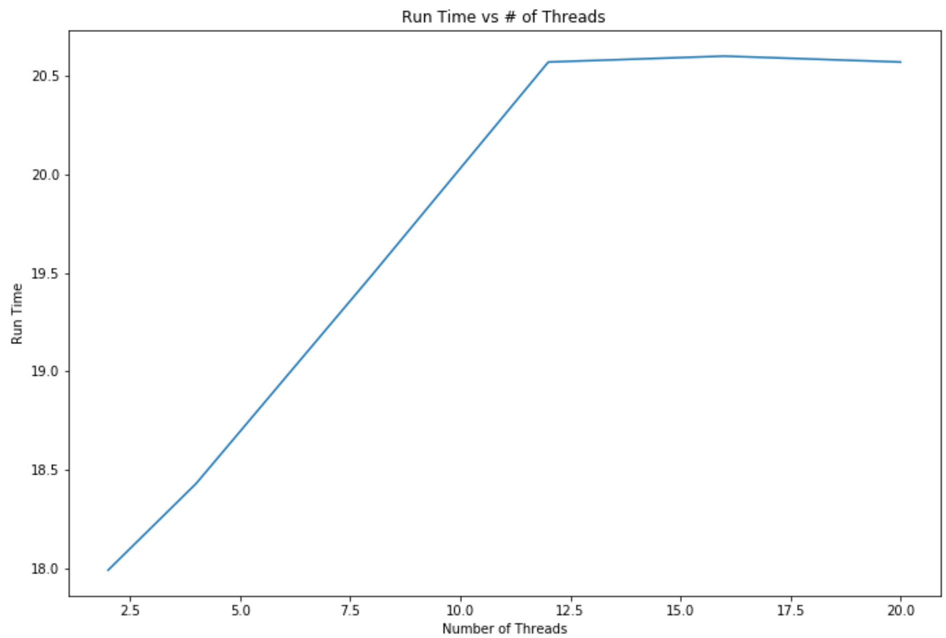
Speed Up (Elapsed Time) / # Threads	1	2	4	8	12	16	20
OpenMP	1	2.197	4.255	8.057	11.43	15.24	19.087



According to result of program there is super linear scaling up to 12 threads for $N = 10^9$.

2.1.2 Weak Scaling

Elapsed Time (Weak Scaling) / # Threads	2	4	8	12	16	20
OpenMP	17.99	18.43	19.49	20.57	20.6	20.57

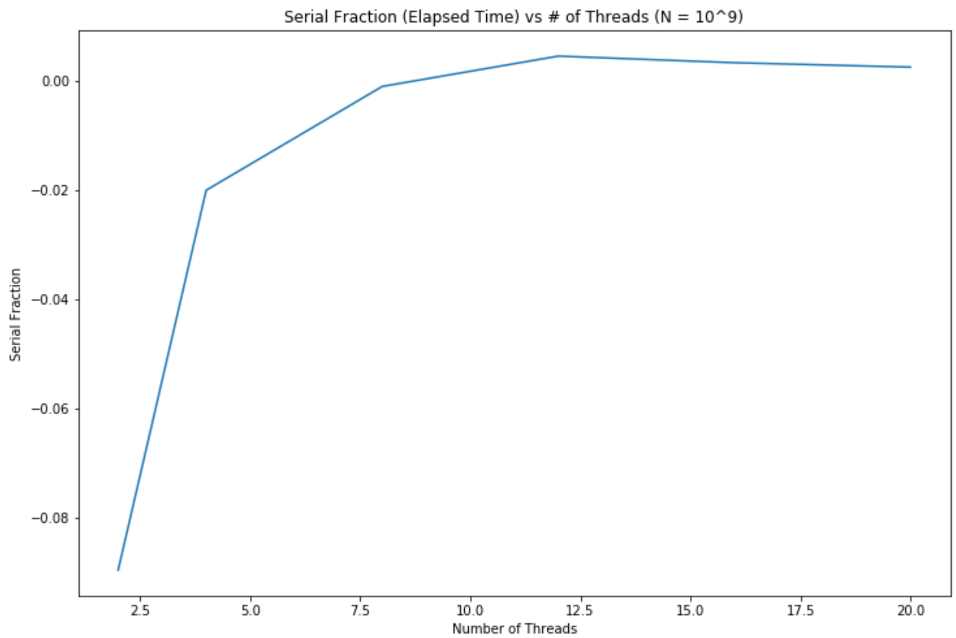


According to logic of weak scalability when we increase the number of cores with the N run time supposed to be same. However for this example there are small differences. There might be some room for optimization of parallelization part. But after 12 threads run time became more or less constant which is good.

2.2 Parallel Overhead

2-estimate the parallel overhead;

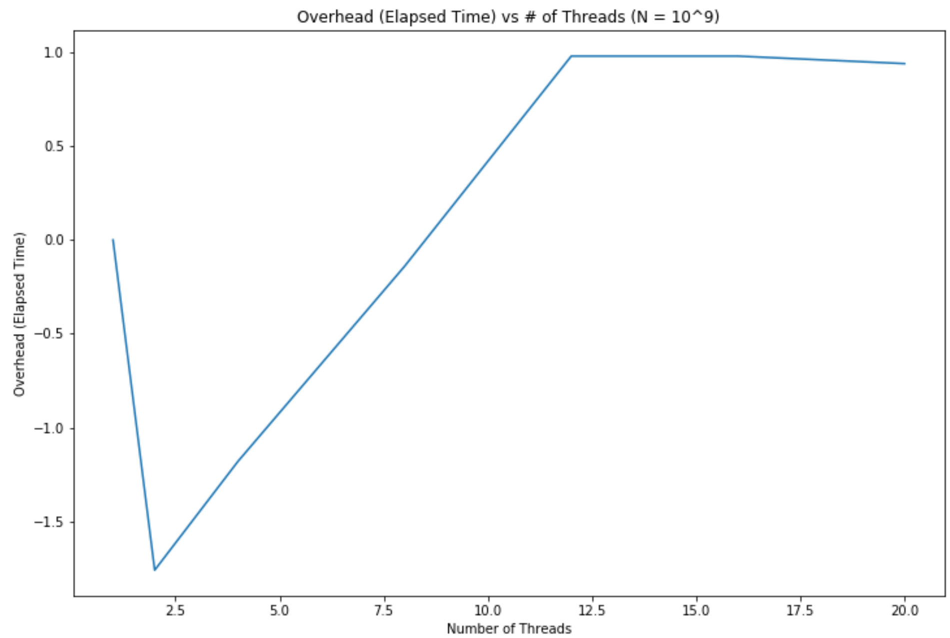
Serial Fraction (elapsed) / # Threads	2	4	8	12	16	20
OpenMP	-0.09	-0.02	-0.001	0.005	0.003	0.003



For serial fraction, above plot shows us first there is increase (which my be due to the overhead) after that it became more or less constant so in order to estimate overhead $p \times Tp - Ts$ will be used.

There is super linear scalability up to 8 threads, that is why there are negative overhead. After 8 threads changing the node spawn overhead and it increase.

Overhead Elapsed Time) / # Threads	1	2	4	8	12	16	20
OpenMP	0	-1.76	-1.18	-0.14	0.98	0.98	0.94



Similar to strong scaling results, program scales perfectly up to 16 threads but after because of overhead it slowed down. It means that there might be still room for parallel optimizaiton especially for higher number of threads (16 and 20)

2.3 Comparing with OpenMPI

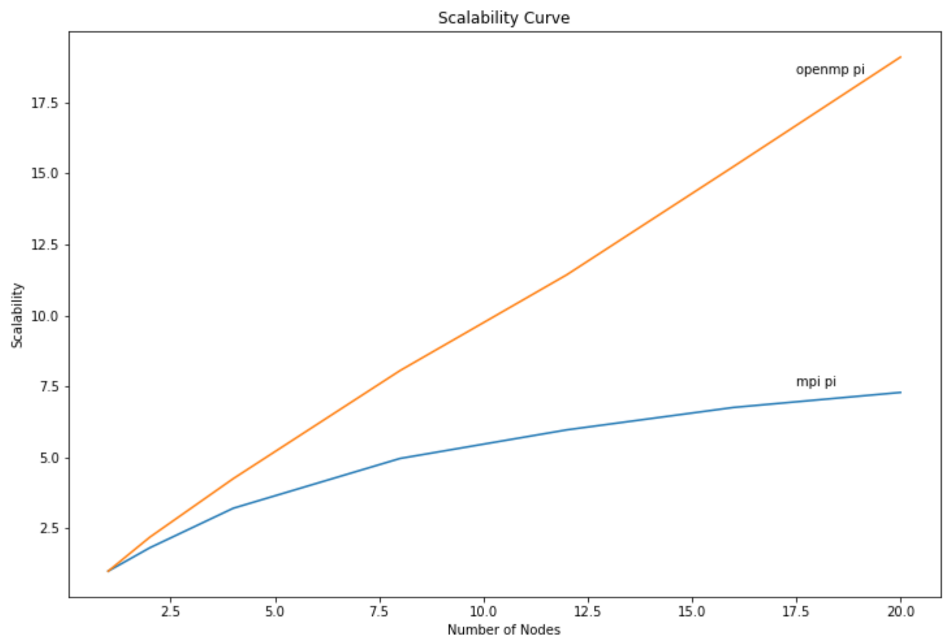
3-compare the performance of your OpenMP version and of the MPI version, in terms of time-to-solution and of parallel efficiency. Run the MPI version with N_c processes (i.e. N_c = the largest number of physical threads that you have on the node) both on the single node that you use for the OpenMP version and on multiple nodes (keeping constant the number of processes). That should allow you to understand the impact of the network and how good is the shared-memory implementation of the MPI library.

2.3.1 Single Node Comparision

Elapsed Time / # Threads	1	2	4	8	12	16	20
MPI	21.5	11.79	6.69	4.33	3.6	3.18	2.95
OpenMP	19.66	8.95	4.62	2.44	1.72	1.29	1.03

Speed Up (Elapsed) / # Threads	1	2	4	8	12	16	20
MPI	1	1.824	3.214	4.965	5.972	6.761	7.288

Speed Up (Elapsed) / # Threads	1	2	4	8	12	16	20
OpenMP	1	2.197	4.255	8.057	11.43	15.24	19.087



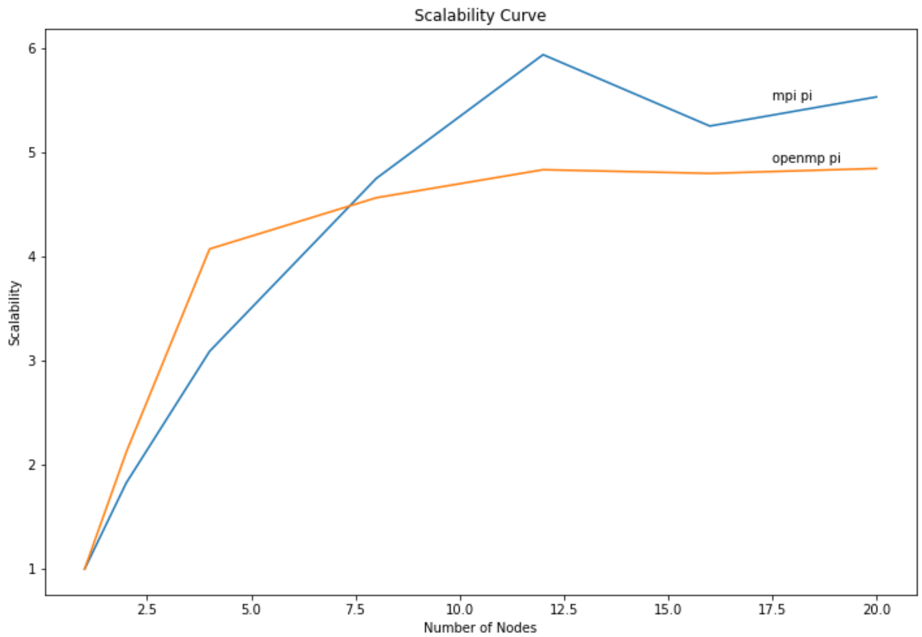
According to this comparasion, in single node openmp performs better than mpi paradigm. (scalability and run time is better than mpi paradigm.) MPI program stopped scaling after 8 cores but openmp scaled linearly up to 20 threads.

2.3.2 Multiple Node Comparision

In order to understand difference between openmp and mpi paradigm same test will be done for multiple nodes with the same number of cores (I got 4 nodes and 5 cores for each node). In this way impact of network and shared memory implementation of mpi will be better understood. It doesn't make sense to run openmp program for more than one node but I run the openmp program for the same conditions to see what happens for openmp after that I will compare mpi program with single mpi performance, single node openmp performance verbally.

Multiple Node Elapsed Time / # Threads	1	2	4	8	12	16	20
MPI	21.86	11.96	7.07	4.6	3.68	4.16	3.95
OpenMP	19.68	9.27	4.83	4.31	4.07	4.1	4.06

Multiple Node Speed Up/ # Threads	1	2	4	8	12	16	20
MPI	1	1.828	3.092	4.752	5.94	5.255	5.534
OpenMP	1	2.123	4.075	4.566	4.835	4.8	4.847



This plot shows us, in multiple nodes mpi paradigm works better. Openmp program didn't scale after 8 threads which is bigger than 5 because openmp is for shared memory. However in comparasion to mpi with single node and openmp single node, it doesn't perform better than them which shows communication between nodes makes program slower.