# PageRank Algorithm

## **Installation**

```
1 1. Clone this repository
2 2. make pagerank
3 3. ./pagerank
```

## **Running Perf, Benchmark & Validity**

In order to run perf tests (outputted to out ), timing and validity tests type:

```
1 | 1. ./test.sh
```

# **Description**

NOTE: All versions of the pagerank methods can be found in pagerank.c just modify their names or the method being called in main.

The data structure used for the implementation of the PageRank algorithm was a simple struct:

```
1  /**
2  * The struct to store the page and its score
3  */
4  struct page_score {
5    double score[2];
6    page* page;
7  };
```

Each would store an array of the scores rather than having a separate previous and new\_score value for each struct. Alternatively (as discussed in <a href="Score Array and Unnecessary Copying">Score Array and Unnecessary Copying</a>), a score array was not the first approach but rather using two variables to store the old and new value with a copy after every iteration.

Previous Struct

```
struct page_score_2D {
   double new_score;
   double old_score;
   page* page;
};
```

Furthermore each page\_score struct would be initialised to hold the page of list in sequential order such that their indices correspond to the index in the struct page.

```
1  /**
2  * Initialise the values of the struct array of page scores
```

```
3
       * @param plist, the list of pages
 4
       * @param npages, the number of pages
 5
       * @return the array of page score structs
       */
 6
 7
    struct page_score* init_pageranks(list* plist, int npages) {
        struct page_score* page_scores = malloc(sizeof(struct page_score) * np
 8
     ages);
 9
        register double initial_value = 1/(double)(npages);
        node* current = plist->head;
10
11
        for (size_t i = 0; i < npages; i++) {
            page_scores[i].page = current->page;
12
            page_scores[i].score[0] = initial_value;
13
            page_scores[i].score[1] = initial_value;
14
15
            current = current->next;
17
        return page_scores;
    }
18
19
```

Hence, through this there was a way to hold the score at each iteration for each page without having to modify the provided node or page structs and without having to do an unnecessary search every time.

Afterwards the sequential implementation just required to loop until convergence. Within the loop every page\_score is changed givens its page inlinks and outlinks.

```
// Loop through until the convergence threshold is reached
 2
    while (diff > EPSILON) {
        diff = 0.0;
        for (size_t i = 0; i < npages; i++) {
 4
 5
            page_scores[i].score[x] = dampening_value;
 6
            register double total = 0.0;
 8
            // Get the list of pages that inlink to this page
 9
            list* inlist = page_scores[i].page->inlinks;
10
            // If null then add diffrerence and continue looping
11
12
            if (inlist == NULL || current->page->noutlinks == 0) {
                diff += pow(page_scores[i].score[x] - page_scores[i].score[!x],
13
    2);;
14
                continue;
15
            // Get the node to loop
16
            node* current = inlist->head;
17
18
            // Loop through the list
19
20
            while (current != NULL) {
                 total += (page_scores[current->page->index].score[!x]) /
21
    ((double)current->page->noutlinks);
22
                current = current->next;
23
            }
24
            page_scores[i].score[x] += total * dampener; // Update the new score
25
            diff += pow(page_scores[i].score[x] - page_scores[i].score[!x], 2);
26
        }
        x = (x + 1) \% 2;
                                // Update the value so we do not have to copy
27
        diff = sqrt(diff);
                                // Get the total difference
28
29
    }
```

# **Implementation and Benchmarks**

## **Score Array and Unnecessary Copying**

Based on the previous part of the assignment it was decided that using some knowledge gained off a 3D array was useful in this scenario. Rather than having a copy loop which would copy all previous values of the iteration to the next.

```
1  for (size_t i = 0; i < npages; i++) {
2     page_scores[i].prev = page_scores[i].current;
3  }</pre>
```

Instead the struct page\_score would contain a double[2] array and would alternate between 1 and 0 after each iteration.

```
for (size_t i = 0; i < npages; i++) {
 2
        page_scores[i].score[x] = dampening_value;
       // CODE BELOW - removed for this example
        while (current != NULL) {
            total += (page_scores[current->page->index].score[!x]) /
    ((double)current->page->noutlinks );
           current = current->next;
9
        }
10
        page_scores[i].score[x] += total * dampener; // Update the new score
11
        diff += (page_scores[i].score[x] - page_scores[i].score[!x]) *
    (page_scores[i].score[x] - page_sco res[i].score[!x]);
13
    }
14
   x = (x + 1) \% 2; // Update the value so we do not have to copy
15
16
```

Specifically there would have had to have been something like this:

```
1  for (size_t i = 0; i < npages; i++) {
2    page_scores[i].old_score = page_scores[i].new_score;
3  }</pre>
```

These resulted in significantly longer runtimes:

```
1 ------ TIME TESTS ------
2 Test: sample.in
3 
4 real 0m0.001s
5 user 0m0.001s
6 sys 0m0.000s
7 Test: test01.in
```

```
9
    real
          0m0.001s
10
    user
             0m0.000s
             0m0.001s
11
    sys
12
    Test: test02.in
13
14
    real
             0m0.001s
15
             0m0.001s
    user
             0m0.000s
16
    sys
17
    Test: test03.in
18
19
             0m0.001s
    real
20
             0m0.000s
    user
21
             0m0.000s
    sys
22
    Test: test04.in
23
24
             0m0.001s
    real
25
             0m0.001s
    user
26
             0m0.000s
    sys
27
    Test: test05.in
28
29
             0m0.001s
    real
             0m0.000s
30
    user
31
    sys
             0m0.001s
32
    Test: test06.in
33
34
    real
             0m0.001s
35
    user
             0m0.001s
             0m0.000s
36
    sys
37
    Test: test07.in
38
39
    real
             0m0.001s
40
             0m0.001s
    user
             0m0.000s
41
    sys
42
    Test: test08.in
43
44
    real
             0m0.001s
             0m0.000s
45
    user
46
    sys
             0m0.001s
    Test: test09.in
47
48
49
    real
             0m0.001s
50
    user
             0m0.000s
51
    sys
             0m0.001s
52
    Test: test10.in
53
54
             0m0.001s
    real
55
    user
             0m0.001s
56
    sys
             0m0.000s
57
    Test: test11.in
58
59
    real
             0m0.001s
60
    user
             0m0.000s
61
    sys
             0m0.001s
62
    Test: test12.in
63
    real
64
             0m8.163s
    user
65
             0m8.152s
66
    sys
             0m0.002s
```

Specifically for test12.in it resulted in a run-time greater than 0.4 seconds when compared to the exact same method but using a score array as described above.

```
Test
          Time
 1
 2
   sample 0.000010
   test1 0.000008
   test2 0.000003
 5 test3 0.000004
 6 test4 0.000003
 7
   test5 0.000004
   test6 0.000004
   test7 0.000003
9
10
   test8 0.000008
11 test9 0.000010
12
   test10 0.000042
13 test11 0.000042
14 test12 7.895575
```

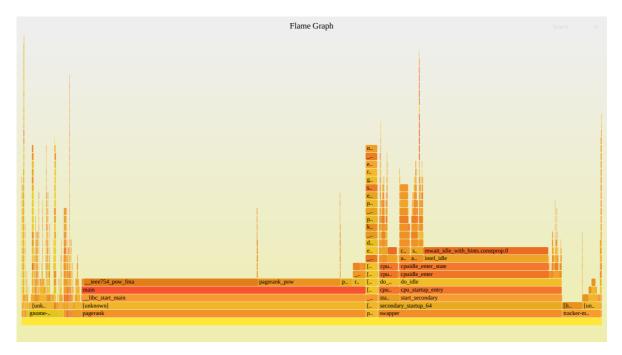
And using <code>omp\_get\_wtime()</code> the total difference is significant between the two implementations (see below).

## Score Array and pow Runtime

```
----- TIME TESTS ------
2
   Test: sample.in
   real 0m0.001s
   user 0m0.000s
4
   sys 0m0.001s
7
   Test: test01.in
   real 0m0.001s
   user 0m0.000s
9
10
   sys 0m0.001s
11
   Test: test02.in
12
13 real 0m0.001s
   user 0m0.001s
14
15
   sys 0m0.000s
16
17
   Test: test03.in
18
   real 0m0.001s
   user 0m0.000s
19
20
   sys 0m0.001s
21
22
   Test: test04.in
23
   real 0m0.001s
24
   user 0m0.000s
25
        0m0.001s
   sys
26
27
   Test: test05.in
28
   real 0m0.001s
29
   user 0m0.001s
        0m0.000s
   sys
```

```
31
32
   Test: test06.in
33
   real 0m0.001s
34
   user 0m0.000s
35
   sys 0m0.001s
36
37
   Test: test07.in
38
   real 0m0.001s
39
   user 0m0.000s
   sys 0m0.001s
40
41
42
   Test: test08.in
43
   real 0m0.001s
   user 0m0.001s
44
   sys 0m0.000s
45
46
   Test: test09.in
47
   real 0m0.001s
48
49
   user 0m0.001s
50
   sys 0m0.000s
51
52
   Test: test10.in
   real 0m0.001s
53
54 user 0m0.001s
   sys 0m0.000s
55
56
57
   Test: test11.in
   real 0m0.001s
58
59 user 0m0.000s
   sys 0m0.001s
60
61
62
   Test: test12.in
63 real 0m7.766s
64 user 0m7.739s
         0m0.001s
65 sys
```

When performing a previous project, it came across that a considerable amount of time was being spent in certain library calls, one of them being <code>pow</code>. In this case I attempted to discover if there was any impact here. As we can see from the Flame Graph there is a considerable amount of time spent calling the <code>pow</code> function and hence it was much more efficient to manually power by multiplying the two values together.



The tests were run by modifying main to get a more accurate result:

```
/* run pagerank and output the results */
double start = omp_get_wtime();
pagerank(plist, ncores, npages, nedges, dampener);
double end = omp_get_wtime();
printf("%lf\n", (end - start));
```

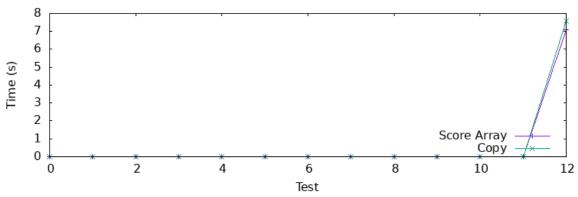
Using the omp\_get\_wtime() results:

```
Test
                Time
 2
    sample
                0.000008
 3
    test1
                0.000008
 4
    test2
                0.000005
    test3
                0.000006
 6
                0.000004
    test4
 7
    test5
                0.000003
8
    test6
                0.000007
9
    test7
                0.000006
                0.000008
10
    test8
11
    test9
                0.000011
                0.000058
12
    test10
13
    test11
                0.000057
    test12
                6.924148
14
```

## **Comparison With Unnecessary Copying**

Below the graph details the differences in run-time between an additional copy and the score array optimisation:





# Without pow Runtime

Due to the issues of pow above it was decided to eliminate this and rather multiply as such:

```
if (inlist == NULL) {
    diff += (page_scores[i].score[x] - page_scores[i].score[!x]) *
    (page_scores[i].score[x] - page_scores[i].score[!x]);
    continue;
}

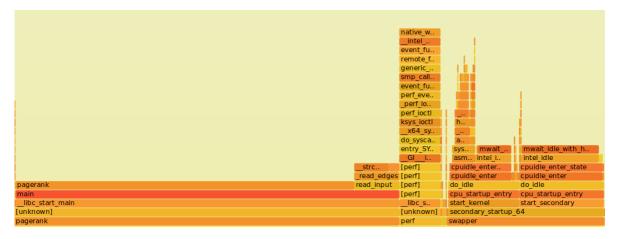
// Rather than
diff += pow(page_scores[i].score[x] - page_scores[i].score[!x], 2);
```

The following times are listed below:

```
1
    ----- TIME TESTS -----
 2
   Test: sample.in
 3
   real
           0m0.001s
 4
   user
           0m0.001s
           0m0.000s
 5
   sys
 6
 7
   Test: test01.in
8
           0m0.001s
    real
9
   user
           0m0.001s
10
           0m0.000s
   sys
11
12
   Test: test02.in
           0m0.001s
13
    real
   user
           0m0.000s
14
           0m0.000s
15
   sys
16
   Test: test03.in
17
18
    real
           0m0.001s
19
           0m0.001s
   user
           0m0.000s
20
   sys
21
   Test: test04.in
22
23
    real
           0m0.001s
24
    user
           0m0.000s
25
    sys
           0m0.000s
26
```

```
Test: test05.in
27
28
    real
           0m0.001s
29
   user
           0m0.000s
30
   sys
           0m0.000s
31
32
   Test: test06.in
33
    real 0m0.001s
34
   user 0m0.000s
35
         0m0.001s
   sys
36
37
   Test: test07.in
38
    real 0m0.001s
39
   user 0m0.000s
40
         0m0.000s
   sys
41
   Test: test08.in
42
    real 0m0.001s
43
   user 0m0.000s
44
         0m0.001s
45
   sys
46
47
   Test: test09.in
48
         0m0.001s
    real
49
   user 0m0.001s
         0m0.000s
50
   SYS
51
52
   Test: test10.in
         0m0.001s
53
    real
   user 0m0.001s
54
         0m0.000s
55
   sys
56
57
   Test: test11.in
58
         0m0.001s
    real
59
   user 0m0.001s
60
         0m0.000s
   sys
61
62
   Test: test12.in
63
    real
           0m3.069s
64
   user
           0m3.064s
65
           0m0.001s
    sys
```

After modifying the pow and using a simple multiplication we see a huge reduction in the time.



The times using omp\_get\_wtime() gave:

```
0.000001
   sample
 3
    test1
                   0.000001
   test2
                   0.000001
   test3
                   0.000001
   test4
                   0.000001
 7
   test5
                  0.000001
8
                  0.000001
   test6
9
   test7
                  0.000001
                 0.000002
10
   test8
11
   test9
                   0.000002
                  0.000014
12
   test10
                   0.000014
13
   test11
                   2,685944
14 test12
```

## **Loop Unrolling**

As an extension in an attempt to optimise the loop overhead, loop unrolling was used on the inner loop:

```
for (; i \le npages - 4; i += 4) {
 2
        page_scores[i].score[x] = dampening_value;
 3
        diff += update_score(page_scores, &page_scores[i], dampener, x);
 5
        page_scores[i + 1].score[x] = dampening_value;
 6
        diff += update_score(page_scores, &page_scores[i + 1], dampener, x);
 7
 8
        page_scores[i + 2].score[x] = dampening_value;
 9
        diff += update_score(page_scores, &page_scores[i + 2], dampener, x);
10
        page_scores[i + 3].score[x] = dampening_value;
11
12
        diff += update_score(page_scores, &page_scores[i + 3], dampener, x);
    }
13
14
15
    for (; i < npages; i++) {
16
        page_scores[i].score[x] = dampening_value;
17
        diff += update_score(page_scores, &page_scores[i], dampener, x);
    }
18
```

Although this seemed as if it would provide major performance improvements it only seemed to decrease performance by approximately 0.6 seconds.

```
Test
 1
                Time
 2
    sample
                0.000001
 3
    1
                0.000001
 4
    2
                0.000000
 5
    3
                0.000000
 6
                0.000000
    4
 7
    5
                0.000000
 8
                0.000000
    6
    7
                0.000000
 9
10
                0.000001
    9
                0.000001
11
    10
                0.000021
12
                0.000015
13
    11
```

This increase in time could possibly be due to the fact that we were making multiple function calls compared to the previous implementation which just performed the whole operation inside the pagerank function. Moreover since we are technically increasing the amount of code there is a significant amount of increased usage of the register which contributes to the reduction in performance. Just unrolling the print statement after provides no huge benefit as well achieving a time of only 2.727 seconds.

## **OpenMP Thread Runtime**

```
----- TIME TESTS -----
 2
    Test: sample.in
 3
           0m0.001s
 4
   real
   user
 5
           0m0.000s
 6
   sys
          0m0.001s
   Test: test01.in
 7
           0m0.001s
9
    real
           0m0.001s
10
   user
11
   sys
          0m0.000s
12
   Test: test02.in
13
           0m0.001s
14
    real
           0m0.000s
15
   user
          0m0.001s
16
   sys
17
   Test: test03.in
18
19
    real
           0m0.001s
20
   user
           0m0.001s
          0m0.000s
21
    sys
22
   Test: test04.in
23
24
    real
           0m0.001s
25
   user
           0m0.000s
26
           0m0.001s
    sys
   Test: test05.in
27
28
    real
           0m0.001s
29
30
   user
           0m0.001s
           0m0.000s
31
    sys
   Test: test06.in
32
33
    real
           0m0.001s
34
35
   user
           0m0.000s
36
    sys
           0m0.001s
37
   Test: test07.in
38
39
    real
           0m0.001s
40
    user
           0m0.000s
41
    sys
           0m0.001s
   Test: test08.in
42
43
```

```
44
   real 0m0.001s
45
   user 0m0.000s
        0m0.003s
46
   sys
47
   Test: test09.in
48
49
   real 0m0.001s
50
   user 0m0.000s
   sys
51
         0m0.001s
   Test: test10.in
52
53
   real 0m0.001s
54
   user 0m0.001s
55
   sys
         0m0.000s
56
   Test: test11.in
57
58
   real 0m0.002s
59
   user 0m0.003s
60
   sys
         0m0.003s
61
   Test: test12-1.in
62
63
64
   real 0m19.969s
   user 0m19.910s
65
   sys
66
         0m0.014s
67
   Test: test12-2.in
68
69
   real 0m16.194s
   user 0m31.585s
70
   sys
71
         0m0.051s
72
   Test: test12-8.in
73
74
   real 0m20.139s
75
   user 2m30.563s
   sys
         0m0.111s
76
77
   Test: test12.in
78
79
   real 0m14.317s
   user 0m55.286s
80
81
   sys
         0m0.084s
```

The parallel implementation made use of threading the main for loop. What is interesting to note is that a static scheduling seemed like the most efficient approach as the work would easily be divided amongst the threads:

```
#pragma omp parallel for schedule(static)
```

But the idea here is a bit tricky, since the **difference** value is being changed hence the update to difference needs to be extracted.

```
#pragma omp parallel for private(i)
for (i = 0; i < npages; i++) {
    page_scores[i].score[x] = dampening_value;
    register double total = 0.0;

// Get the list of pages that inlink to this page
list* inlist = page_scores[i].page->inlinks;

// If null then add diffrerence and continue looping
```

```
if (inlist == NULL) {
10
11
            continue;
12
        }
13
      // Get the node to loop
14
       node* current = inlist->head;
       // Loop through the list
15
        while (current != NULL) {
16
            total += (page_scores[current->page->index].score[!x]) /
17
    ((double)current->page-> noutlinks);
18
            current = current->next;
        }
19
        page_scores[i].score[x] += total * dampener; // Update the new score
21
22
    }
```

As shown above the difference update is removed into a separate for loop such that there isn't a race condition and eliminates the need for an atomic or critical section of the for loop as such:

```
1  #pragma omp atomic update
2  diff += ...
```

The following times were achieved for a default scheduling:

```
Test
          Time
   sample 0.000013
2
3
         0.000013
   1
   2
        0.000001
4
5 3
        0.000001
6
   4
        0.000001
7
   5
        0.000018
        0.000016
   6
8
9 7
        0.000016
10 8
        0.068941
11
   9
        0.000003
12 10
        0.000026
   11
       0.230591
13
14 | 12-1 3.385419
15 12-2 2.316247
16
   12-4 2.099469
17
   12-8
         1.864362
```

But still these times are nowhere near expected for a parallel optimisation of the pagerank algorithm.

### **Pragma Critical Around Difference Sum**

Attempting a pragma omp critical in a single for loop gives excessively higher runtimes as we can expect multiple threads to be conflicting in contention for this line:

```
#pragma omp critical
diff = diff + page_scores[i].difference;
```

The times below show the inefficiency of a pragma omp critical directive in this scenario:

```
Time
   Test
 2
   sample 0.000015
        0.000016
 3
   1
        0.000001
 4 2
 5 3
        0.000001
6 4
        0.000001
 7
  5
        0.000017
   6
       0.000015
8
9 7
        0.000018
        0.000032
10 8
11 9
        0.000002
12 10
        0.000024
13 11 0.000153
14 12-1 5.359157
15 12-2 13.226314
16 | 12-4 23.349664
17 12-8 41.443465
```

### **Pragma Atomic**

Since the pragma omp critical directive was so time consuming then a more efficient atomic operation was decided upon since the time of contention would be quite small. This means that a **busy-waiting** approach rather than suspending the thread would be much more useful in this scenario:

```
#pragma omp atomic write
diff = diff + page_scores[i].difference;
```

The results were better than with <code>critical</code> yet still not sufficient enough for a parallel solution. But as the number of threads increase to 8 we see an insane projection:

```
Test
              Time
 2
   sample.in 0.000014
 3 test01.in 0.000014
 4 test02.in 0.000001
 5 test03.in 0.000001
 6 test04.in 0.000001
 7
   test05.in 0.000019
8 test06.in 0.000012
   test07.in 0.000013
9
10 test08.in 0.066317
11 test09.in 0.000003
12 test10.in 0.000024
13 test11.in 0.461696
14 test12-1.in 4.138385
15 test12-2.in 8.020848
16 test12-4.in 19.682065
17 test12-8.in 76.166786
```

### **Padding Struct**

Since some of the issues with time would have been to do with false sharing, the struct was padded to be a total of 64 bytes such that it fits into a cache line directly.

```
----- TIME TESTS -----
2
   sample.in 0.000015
3 test01.in 0.000020
4 test02.in 0.000001
5 test03.in 0.000001
6 test04.in 0.000001
7 test05.in 0.000017
8 test06.in 0.000018
9 test07.in 0.000018
10 test08.in 0.000030
11 test09.in 0.000002
12 test10.in 0.000028
13 test11.in 0.000067
14 test12-1.in 3.062184
15 test12-2.in 1.774298
   test12-8.in 1.606186
17 test12.in 1.704996
18
```

### **Reduction on Difference Sum**

Instead a reduction on the sum of differences was chosen to help optimise the second for loop:

```
#pragma omp parallel for reduction (+:diff)
for (i = 0; i < npages; i++) {
    diff = diff + page_scores[i].difference;
}</pre>
```

The results with this reduction are very surprising with a consistent decrease every time.

```
1 0.000015
 2
   0.000015
 3 0.000001
4 0.000001
5 0.000001
6 0.000017
   0.000020
 7
   0.000015
9
   0.000027
10 0.000003
11 0.000025
12
   0.000065
13 3.374477
14 2.002390
15
   1.850901
```

# Static (Reduction)

With chunk sizes of 2, 4 and 8 we have:

#### Chunk Size = 2

```
----- TIME TESTS ------
   sample.in 0.000004
2
3 test01.in 0.000004
   test02.in 0.000001
5 test03.in 0.000001
6 test04.in 0.000000
7
   test05.in 0.000001
8 test06.in 0.000004
   test07.in 0.000004
10 test08.in 0.000007
11 test09.in 0.000002
12 test10.in 0.000021
13 test11.in 0.000048
   test12-1.in 3.178239
15
   test12-2.in 3.051578
16
   test12-8.in 2.711370
   test12.in 2.639630
17
18
```

### Chunk Size = 4

```
----- TIME TESTS -----
   sample.in 0.000003
3 test01.in 0.000004
4 test02.in 0.000001
5
   test03.in 0.000001
   test04.in 0.000000
7
   test05.in 0.000001
   test06.in 0.000004
9
   test07.in 0.000005
   test08.in 0.000007
10
11
   test09.in 0.000002
12 test10.in 0.000019
13
   test11.in
             0.000046
   test12-1.in 3.309710
14
15
   test12-2.in 3.019476
16
   test12-8.in 2.212091
17 test12.in 2.483555
```

#### Chunk Size = 8

```
1 0.000033
   0.000034
   0.000029
4 0.000027
5 0.000028
6 0.000037
7 0.000026
   0.000024
9 0.000027
10 0.000023
11 0.000076
12 0.000951
13 0.147611
14 0.000360
15 0.000318
16 0.000345
```

### **Dynamic (Reduction)**

With the dynamic tests we get a very interesting predicament.

From the data below it seemed as if false sharing was occurring in the loop as we have a sufficiently increased time when using dynamic with a chunk size for 2, 4 and 8 threads. Modifying the chunk sizes to 2, 4, and 8 are shown below:

#### Chunk Size = 2

```
----- TIME TESTS -----
2 sample.in 0.000007
3 test01.in 0.000006
4 test02.in 0.000001
5 test03.in 0.000001
6 test04.in 0.000001
7 test05.in 0.000002
   test06.in 0.000003
9 test07.in 0.000003
10 test08.in 0.157384
11 test09.in 0.000003
12 test10.in 0.000049
   test11.in 0.000090
13
14 | test12-1.in 6.832125
15 test12-2.in 10.849397
16 test12-8.in 8.730513
17 test12.in 12.916811
```

#### Chunk Size = 4

```
5  test03.in  0.000001
6  test04.in  0.000001
7  test05.in  0.000002
8  test06.in  0.000003
9  test07.in  0.000003
10  test08.in  0.146991
11  test09.in  0.000003
12  test10.in  0.000048
13  test11.in  0.000085
14  test12-1.in  6.833124
15  test12-2.in  11.300713
16  test12-8.in  8.779134
17  test12.in  12.723585
```

#### Chunk Size = 8

However for 8 chunks the results show that there are fewer instances of false sharing:

```
1 ------ TIME TESTS -----
2 sample.in 0.000006
3 test01.in 0.000006
4 test02.in 0.000001
5 test03.in 0.000001
6 test04.in 0.000001
7 test05.in 0.000002
8 test06.in 0.000003
9 test07.in 0.000003
10 test08.in 0.131622
11 test09.in 0.000003
12 test10.in 0.000025
13 test11.in 0.000061
14 test12-1.in 3.773181
15 test12-2.in 3.216287
16 test12-8.in 2.219597
17 test12.in 3.090391
```

# **Comparison On Number of Pages**

As an extension, the methods were run against inputs wherein the number of pages would vary from 100 - 50000000.

