

Contents

1	Introduction	2
2	Aim of this assignment	2
3	Tests to be performed	2
4	Starting program	3
5	Vtune Example results	4
6	Testing with a thread size of 2	6
6.1	Loop (1)	6
6.2	Loop (2)	8
6.3	Loop (3)	9
6.4	Conclusion from results	9
7	Using 8 threads	10
7.1	Loop (1)	10
7.2	Loop (2)	11
7.3	Loop (3)	11
7.4	Conclusion from results	12
8	Conclusion from all results	12
9	Recommendations	12
10	Appendix Code Before	13
11	Appendix Code After	14

1 Introduction

Vtune is a program used to analyse applications, specifically aimed at finding hotspots as well as regions which can be parallelized. This makes it incredibly useful when looking at multi core programming.

2 Aim of this assignment

The following were the assumptions of the aims of the assignment:

- Use Intel Vtune amplifier to optimize a program specifically looking to optimize scheduling as well as chunk sizes.
- Display understanding of outputs generated by Vtune amplifier

To assure that these aims are completed I will test the results of each parallel region using different chunk sizes as well as different scheduling techniques, using Vtune amplifier to reduce the scheduling time needed as well as time wasted because of imbalances. Because these two parameters being directly linked to concurrency I mainly used vtuner's concurrency tests.

3 Tests to be performed

I will be testing how:

- Cores used
- Scheduling type used
- Parallel section vs Parallel for
- Chunks used by each scheduling type

Effects the time spent scheduling threads, also looking at imbalances.

4 Starting program

A starting point for the program is displayed below, we can see that we have 3 for loops which omp can use for parallel computing, these are labelled and throughout this report I will be using the numbering provided below.

```
int main(int argc, char *argv[])
{
    int thread_count = 2;
    // set array size
    int size = 5000000;
    // allocate space for our array
    double *arrayA = (double*)malloc(size * sizeof(double));
    double *arrayB = (double*)malloc(size * sizeof(double));
    // fill our array in parallel using guided schedule
#pragma omp parallel for schedule(guided) num_threads(thread_count)
    for (int i = 0; i < size; i++)
    {
        arrayA[i] = sin(i);
        arrayB[i] = sin(i + 5);
    }
    // calculate the sum of each array
    double sumOfA = 0;
    double sumOfB = 0;
    // work out the sum in parallel using guided scheduling
#pragma omp parallel for schedule(guided) reduction(+:sumOfB) reduction(+:sumOfA) num_threads(thread_count)
    for (int i = 0; i < size; i++)
    {
        sumOfB = sumOfB + arrayB[i];
        sumOfA = sumOfA + arrayA[i];
    }
    // work out the means of both arrays
    double meanOfA = sumOfA / size;
    double meanOfB = sumOfB / size;
    // initialize SD and pearsons to 0.
    double standardDevA = 0;
    double standardDevB = 0;
    double pearsons = 0;
    // calculate part of SD and Pearsons in parallel
#pragma omp parallel for schedule(static) reduction(+:standardDevA) reduction(+:standardDevB) reduction(+:pearsons) num_threads(thread_count)
    for (int i = 0; i < size; i++)
    {
        standardDevA += (arrayA[i] - meanOfA)*(arrayA[i] - meanOfA);
        standardDevB += (arrayB[i] - meanOfB)*(arrayA[i] - meanOfA);
        pearsons += ((arrayA[i] - meanOfA)*(arrayB[i] - meanOfB));
    }
    // finish calculating SD and pearsons.
    standardDevA = sqrt(standardDevA / size);
    standardDevB = sqrt(standardDevB / size);
    pearsons = ((pearsons / size) / (standardDevA*standardDevB));
    // free it
    free(arrayA);
    free(arrayB);
    // print it out
    printf("sumA = %lf, standDevA = %lf, averageA = %lf\nsumB = %lf, standDevB = %lf, averageB = "
           "%lf\nPearsons = %lf\n", sumOfA, standardDevA, meanOfA, sumOfB, standardDevB, meanOfB, pearsons);
}
```

1

2

3

5 Vtune Example results

After we have run a test we are greeted by the following page, this shows us the results from our program including useful information such as elapsed time, time lost due to imbalances or serial spinning across all omp sections, and also time spent scheduling threads. We also notice serial methods which have used a long period of time to execute and time spent in each region.

⌵

Elapsed Time[?]: 0.804s

⌵

CPU Time[?]:

0.386s

⌵

Effective Time[?]:

0.371s

⌵

Spin Time[?]:

0.016s

⌵

Imbalance or Serial Spinning[?]:

0.016s

⌵

Lock Contention[?]:

0s

⌵

Other[?]:

0s

⌵

Overhead Time[?]:

0s

⌵

Creation[?]:

0s

⌵

Scheduling[?]:

0s

⌵

Reduction[?]:

0s

⌵

Atomics[?]:

0s

⌵

Other[?]:

0s

⌵

Wait Time[?]:

0.027s

⌵

Total Thread Count:

2

⌵

Paused Time[?]:

0s

⌵

OpenMP Analysis. Collection Time[?]: 0.804

⌵

Serial Time (outside parallel regions)[?]: 0.605s (75.3%)

⌵

Top Serial Hotspots (outside parallel regions)

This section lists the loops and functions executed serially in the master thread outside of any OpenMP region and consuming the most CPU time. Improve overall application performance by optimizing or parallelizing these hotspot functions. Since the Serial Time metric includes the Wait time of the master thread, it may significantly exceed the aggregated CPU time in the table.

Function	Module	Serial CPU Time [?]
free_base	ucrtbase.dll	0.012s

⌵

Parallel Region Time[?]: 0.199s (24.7%)

⌵

Estimated Ideal Time[?]:

0.199s (24.7%)

⌵

OpenMP Potential Gain[?]:

0.000s (0.0%)

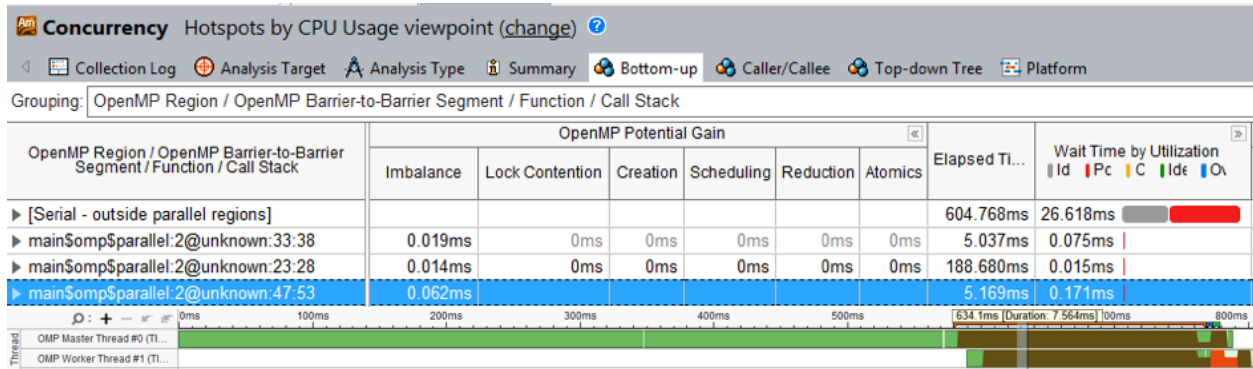
⌵

Top OpenMP Regions by Potential Gain

This section lists OpenMP regions with the highest potential for performance improvement. The Potential Gain metric shows the elapsed time that could be saved if the region was optimized to have no load imbalance assuming no runtime overhead.

OpenMP Region	OpenMP Potential Gain [?]	(%) [?]	OpenMP Region Time [?]
main\$omp\$parallel:2@unknown:33:38	0.000s	0.0%	0.005s
main\$omp\$parallel:2@unknown:23:28	0.000s	0.0%	0.189s
main\$omp\$parallel:2@unknown:47:53			0.005s

Because this report mainly focuses on reducing imbalances within our omp sections we can now click "imbalances or serial spinning" which will give a more detailed view of our results this is shown below, we can see that the our program only starts doing meaningful work from the graph at the bottom of the following image, denoted by the brown block after 0.634ms I believe this overhead is caused by Vtune starting its timing from launching the executable however because I am only using one machine this should be similar in run.



We can also see from the graph at the bottom of the image that there is almost no time used either scheduling threads or lost due to imbalances within our omp region, this would be denoted by red chunks rather than brown ones. We can also see that the time spent in omp region (1) denoted by the orange slider above the graph is substantially higher than the following omp regions denoted in blue and green.

For this Assignment, I am specifically interested in the time taken to schedule threads as well as how much time was wasted in a region due to imbalances, this information is clearly available in the previous image above the graph. In the above image, we see three omp parallel regions at lines 23:28, our first parallel region (1), another which is from lines 33:38, our second parallel region (2) and a third parallel region from lines 47:53 (3), each lost differing amounts of time due to imbalances for example (1) took 0.014ms, (2) took 0.019ms and (3) took 0.062ms. For our analysis, I will not be showing the table displayed result and instead retrieve from the table the results that I am interested in. I will be using graphs in the case of needing to show why scheduling is high or imbalances are high however has said previously this is un-needed due to only looking at scheduling and imbalance times.

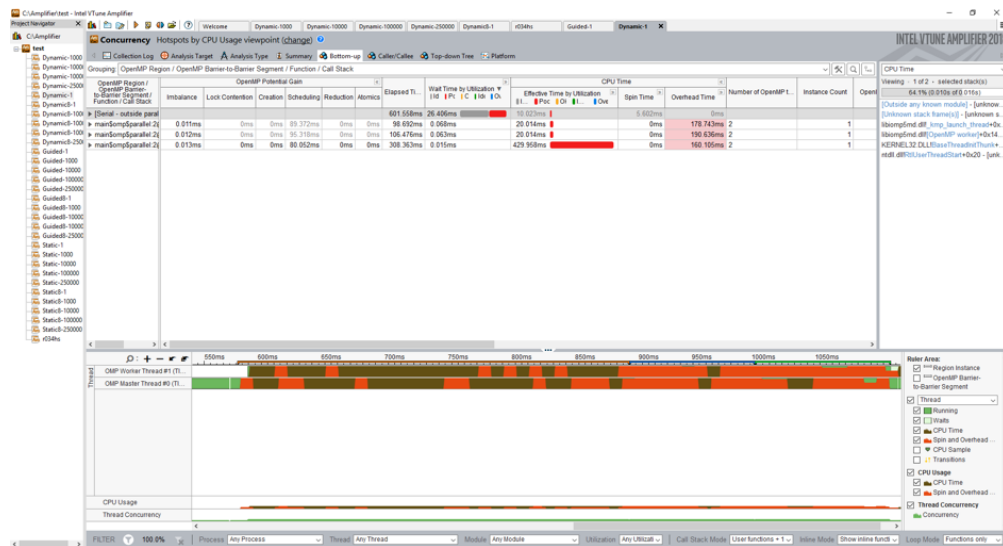
6 Testing with a thread size of 2

I will now be looking at how changing chunk size as well as scheduling type effects the imbalance time as well as scheduling time, this will be done for each schedule as well as chunk sizes 1,1000,10000,100000 and 250000. This will be listed for each loop separately to allow for easy lookup of results.

6.1 Loop (1)

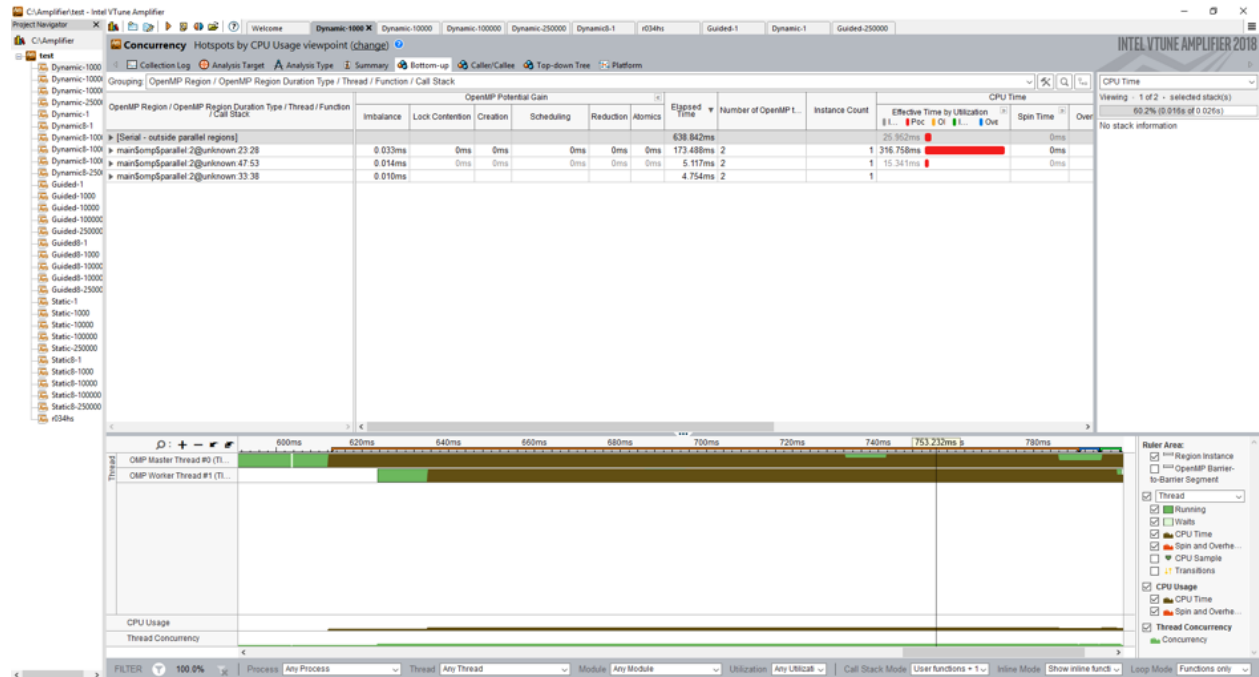
Scheduling type	Chunk size	Imbalance time(ms)	Scheduling time(ms)	Sum(ms)
Guided	1	0.013	0	0.013
Guided	1000	0.035	0	0.035
Guided	10,000	0.051	0	0.051
Guided	100,000	0.728	0	0.728
Guided	250,000	4.811	0	4.811
Static	1	0.367	0	0.367
Static	1000	1.322	0	1.322
Static	10,000	5.989	0	5.989
Static	100,000	6.439	0	6.439
Static	250,000	7.060	0	7.060
Dynamic	1	0.014	130	130.014
Dynamic	1000	0.033	0	0.033
Dynamic	10,000	0.103	0	0.103
Dynamic	100,000	3.018	0	3.018
Dynamic	250,000	0.900	0	0.900

From the table above we can see certain characteristics, even if our scheduling time taken by dynamic scheduling was 0 i.e. all scheduling was instant our imbalance time would be greater than that of guided scheduling, this may be due to how few threads exist as changes made to the scheduler will have less of an impact on the time saved / wasted. However, we can see that using dynamic scheduling our time taken to schedule was not 0 it was considerably higher than this being 130, If I would like to analyse why this was the case within Vtune analysis I could use the following page with the associated graph.



From this graph we can see that our time spent scheduling is split up across both threads, this is represented by non-continuous brown blocks, we notice that the time in red blocks (time spent scheduling) is massive compared to our previous example.

This is even more obvious in the following figure where we have no scheduling issues or imbalances.



Using the table for our results we can see some trends however we also see some outliers, this is could be caused by differing loads on the computer at runtime since the program is so small even the smallest difference on the load on the computer could have a major effect.

The most obvious trend is that of time wasted due to imbalances as well as scheduling time, we can see that all scheduling types had this issue, however some scaled worse than others, for example guided scheduling scaled slower than static scheduling, and dynamic scheduling when removing outliers performed well when increasing the chunk size from 1 to 1000, this is due to how chunks are distributed in using dynamic scheduling. however, after this point we see that it actually scales better than either static or guided scheduling.

Due to the time lost because of scheduling being so low it is worth mentioning that results that have a low difference are potentially switched therefore this however isn't the case for the first loop where Guided scheduling comes out as the best candidate by 0.02ms. Therefore, we can summarize that guided scheduling with a chunk size of 1 using only 2 threads is the best scheduling type.

6.2 Loop (2)

Scheduling type	Chunk size	Imbalance time	Scheduling time	Sum
Guided	1	0.011	0	0.011
Guided	1000	0.009	0	0.009
Guided	10,000	0.012	0	0.012
Guided	100,000	0.017	0	0.017
Guided	250,000	0.049	0	0.049
Static	1	0.101	0	0.101
Static	1000	0.236	0	0.236
Static	10,000	0.027	0	0.027
Static	100,000	0.017	0	0.017
Static	250,000	0.132	0	0.132
Dynamic	1	0.013	85.564	85.577
Dynamic	1000	0.010	0	0.010
Dynamic	10,000	0.010	0	0.010
Dynamic	100,000	0.065	0	0.065
Dynamic	250,000	0.010	0	0.010

From the above table we can see a similar trend to that mentioned previously however we see that dynamic scheduling did not actively increase or decrease when increasing the chunk size, however static and guided scheduling followed the pattern mentioned previously of increasing slowly. We can notice in this graph that dynamic scheduling with a chunk size of 1000 is the best scheduling type however due to guided's results being within 0.001 it is hard to state with certainty that Dynamic with a chunk size of 1000 is better than that of guided scheduling with only a chunk size of 1 therefore the optimal choice is either guided scheduling with chunk size of 1 or dynamic scheduling with a chunk size of 1000.

6.3 Loop (3)

Scheduling type	Chunk size	Imbalance time	Scheduling time	Sum
Guided	1	0.011	0	0.011
Guided	1000	0.014	0	0.014
Guided	10,000	0.011	0	0.011
Guided	100,000	0.013	0	0.013
Guided	250,000	0.030	0	0.030
Static	1	0.013	0	0.013
Static	1000	0.025	0	0.025
Static	10,000	0.027	0	0.027
Static	100,000	0.025	0	0.025
Static	250,000	0.023	0	0.023
Dynamic	1	0.011	85.564	85.575
Dynamic	1000	0.014	0	0.014
Dynamic	10,000	0.012	0	0.012
Dynamic	100,000	0.073	0	0.073
Dynamic	250,000	0.198	0	0.198

Looking at the previous table denoting loop 3 we notice trends very similar to our first loop but to a lesser degree and as such note that our optimal schedule for loop 3 is using guided scheduling with a chunk size of 1.

6.4 Conclusion from results

We can conclude from our results that the optimal solution is one that has loop 1 being scheduled using guided with chunk size of 1, loop 2 being scheduled using Dynamic with a chunk size of 1000 or Guided using a chunk size of 1 and chunk 3 being scheduled using Guided with chunk size of 1.

I will now look for our optimal solution whether there are any increases to be made by switching from dedicated parallel for loops to a parallel section with multiple omp for loops. For this we will be looking at the elapsed time for the program. As we are only changing one parameter in our program.

	Elapsed Time
Parallel section	0.717s
Multiple Parallel for	0.723s

We can see from our results that there is only a difference between changing from a parallel section to multiple dedicated parallel for sections this difference is so small that it is within the margin of error and therefore we cannot make any solid conclusions about which is better.

7 Using 8 threads

7.1 Loop (1)

Scheduling type	Chunk size	Imbalance time	Scheduling time	Sum
Guided	1	6.936	0	6.936
Guided	1000	0.107	0	0.107
Guided	10,000	0.801	0	0.801
Guided	100,000	4.067	0	4.067
Guided	250,000	2.541	0	2.541
Static	1	49.046	0	49.046
Static	1000	16.448	0	16.448
Static	10,000	12.222	0	12.222
Static	100,000	12.273	0	12.273
Static	250,000	9.691	0	9.691
Dynamic	1	0.013	80.052	85.065
Dynamic	1000	0.058	0	0.058
Dynamic	10,000	0.643	0	0.643
Dynamic	100,000	8.374	0	8.374
Dynamic	250,000	0.198	0	0.198

From the table above we can see similar things to our solution using only 2 threads for example using dynamic scheduling with only a chunk size of 1 still creates a high scheduling time.

Using 8 threads our trends are much more distinct, we can see that increasing the chunk size from 1 to 1000 decreases our scheduling time substantially for dynamic scheduling, we can also see that compared to other scheduling techniques dynamic drastically outperformed all of them when using a chunk size of 1000. we actually see a different trend when only looking at guided and static scheduling noticing that we actually gain performance when increasing our chunk size ranging from using 49ms using a chunk size of one for static scheduling to using only 9ms when using a chunk size of 250,000. From this table, we can state with high certainty that Dynamic scheduling using a chunk size of 1000 is our optimal value from the chunk sizes we have decided upon however we can note that further performance increases may be possible if we were to further explore using chunk sizes between 1 and 1000.

7.2 Loop (2)

Scheduling type	Chunk size	Imbalance time	Scheduling time	Sum
Guided	1	0.248	0	0.248
Guided	1000	0.120	0	0.120
Guided	10,000	0.054	0	0.054
Guided	100,000	0.126	0	0.126
Guided	250,000	0.197	0	0.197
Static	1	9.905	0	9.905
Static	1000	1.307	0	1.307
Static	10,000	0.049	0	0.049
Static	100,000	0.916	0	0.916
Static	250,000	0.724	0	0.724
Dynamic	1	0.011	89.372	89.383
Dynamic	1000	0.058	0	0.058
Dynamic	10,000	0.049	0	0.049
Dynamic	100,000	0.156	0	0.156
Dynamic	250,000	0.198	0	0.198

The table above is much less clear as to the trend of each scheduler however we can see a clear outlier when looking for the optimal scheduler, Dynamic with a chunk size of 1000, as stated previously a more optimal solution may be possible and could be found by testing more chunk sizes within the range of 1:1000.

7.3 Loop (3)

Table 1: My caption

Scheduling type	Chunk size	Imbalance time	Scheduling time	Sum
Guided	1	0.547	0	0.547
Guided	1000	0.033	0	0.033
Guided	10,000	0.052	0	0.052
Guided	100,000	0.013	0	0.013
Guided	250,000	0.155	0	0.155
Static	1	7.484	0	7.484
Static	1000	1.037	0	1.037
Static	10,000	0.052	0	0.052
Static	100,000	1.533	0	1.533
Static	250,000	1.251	0	1.251
Dynamic	1	0.012	95.318	95.33
Dynamic	1000	0.014	0	0.014
Dynamic	10,000	0.052	0	0.052
Dynamic	100,000	0.223	0	0.223
Dynamic	250,000	0.198	0	0.198

The final table showing values related to the third loop shows slight trends for guided scheduling, which seems to increase in efficiency as its chunk size increases. However, the best performing scheduling method is that of Dynamic scheduling with a chunk size of 1000.

7.4 Conclusion from results

We can conclude that the best scheduling techniques for running the program when using 8 threads are to use Dynamic scheduling for all three loops with a chunk size ranging from 1 to 1000.

Now we have found the optimal scheduling methods for all three omp loops we need to test whether any performance can be gained from combining our dedicated parallel for sections into one parallel section and 3 omp for loops.

Table 2: My caption

	Elapsed Time
Parallel section	0.73s
Multiple Parallel for	0.715s

We can see from the results that we have a clearer separation in our elapsed time and can therefore conclude that when we are using 8 threads we can gain added efficiency by only using one parallel region with 3 omp for loops contained within it.

8 Conclusion from all results

We have looked into how to use Vtune amplifier to optimize omp sections, helping transform our original piece of code which took 0.742s to run into our new version which only took 0.715 saving us about 0.027s. The original code as well as the changed code is located in the appendix.

We have also concluded that the amount of threads used dramatically changes the optimal scheduling technique, as only using 2 threads yielded a completely different optimal solution than using 8.



9 Recommendations

Due to using university machines to test our code on which may have varying load on them at any time it would be better to use a system which only has the minimal applications to run our analysis's as even running the code multiple times can still cause erroneous data, having ran each test twice to gather these values.

Additionally when looking at the output of Vtune amplifier we can see that there are major contributors to our elapsed time, this being the implementation of sin because of this it would be highly advised to look into more efficient ways to calculate these sin values.

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time 
_libm_sin_e7	libmmd.dll	0.338s
__kmp_fork_barrier	libiomp5md.dll	0.158s 
main\$omp\$parallel_for@37	ConsoleApplication1.exe	0.039s
[Outside any known module]		0.026s
_libm_sin_l9	libmmd.dll	0.020s
[Others]		0.033s

10 Appendix Code Before

```
int main(int argc, char *argv[])
{
    int thread_count = 2;
    // set array size
    int size = 5000000;
    // allocate space for our array
    double *arrayA = (double*)malloc(size * sizeof(double));
    double *arrayB = (double*)malloc(size * sizeof(double));
    // fill our array in parallel using guided schedule
#pragma omp parallel for schedule(guided) num_threads(thread_count)
    for (int i = 0; i < size; i++)
    {
        arrayA[i] = sin(i);
        arrayB[i] = sin(i + 5);
    }
    // calculate the sum of each array
    double sumOfA = 0;
    double sumOfB = 0;
    // work out the sum in parallel using guided scheduling
#pragma omp parallel for schedule(guided) reduction(+:sumOfB) reduction(+:sumOfA) num_threads(thread_count)
    for (int i = 0; i < size; i++)
    {
        sumOfB = sumOfB + arrayB[i];
        sumOfA = sumOfA + arrayA[i];
    }
    // work out the means of both arrays
    double meanOfA = sumOfA / size;
    double meanOfB = sumOfB / size;
    // initialize SD and pearsons to 0.
    double standardDevA = 0;
    double standardDevB = 0;
    double pearsons = 0;
    // calculate part of SD and Pearsons in parallel
#pragma omp parallel for schedule(static) reduction(+:standardDevA) reduction(+:standardDevB) reduction(+:pearsons) num_threads(thread_count)
    for (int i = 0; i < size; i++)
    {
        standardDevA += (arrayA[i] - meanOfA)*(arrayA[i] - meanOfA);
        standardDevB += (arrayB[i] - meanOfB)*(arrayA[i] - meanOfA);
        pearsons += ((arrayA[i] - meanOfA)*(arrayB[i] - meanOfB));
    }
    // finish calculating SD and pearsons.
    standardDevA = sqrt(standardDevA / size);
    standardDevB = sqrt(standardDevB / size);
    pearsons = ((pearsons / size) / (standardDevA*standardDevB));
    // free it
    free(arrayA);
    free(arrayB);
    // print it out
    printf("sumA = %lf, standDevA = %lf, averageA = %lf\nsumB = %lf, standDevB = %lf, averageB = "
           "%lf\nPearsons = %lf\n", sumOfA, standardDevA, meanOfA, sumOfB, standardDevB, meanOfB, pearsons);
}
```

11 Appendix Code After

```
int main(int argc, char *argv[])
{
    int thread_count = 8;
    // set array size
    int size = 5000000;
    // allocate space for our array
    double *arrayA = (double*)malloc(size * sizeof(double));
    double *arrayB = (double*)malloc(size * sizeof(double));
    double sumOfA = 0;
    double sumOfB = 0;
    double standardDevA = 0;
    double standardDevB = 0;
    double pearsons = 0;
    double meanOfA = 0;
    double meanOfB = 0;
    // fill our array in parallel using guided schedule
    #pragma omp parallel num_threads(thread_count)
    {
        #pragma omp for schedule(guided,1)
        for (int i = 0; i < size; i++)
        {
            arrayA[i] = sin(i);
            arrayB[i] = sin(i + 5);
        }
        // work out the sum in parallel using guided scheduling
        #pragma omp for schedule(guided,1000) reduction(+:sumOfB) reduction(+:sumOfA)
        for (int i = 0; i < size; i++)
        {
            sumOfB = sumOfB + arrayB[i];
            sumOfA = sumOfA + arrayA[i];
        }
        #pragma omp single
        {
            // work out the means of both arrays
            meanOfA = sumOfA / size;
            meanOfB = sumOfB / size;
        }
        // calculate part of SD and Pearsons in parallel
        #pragma omp for schedule(guided,1) reduction(+:standardDevA) reduction(+:standardDevB) reduction(+:pearsons)
        for (int i = 0; i < size; i++)
        {
            standardDevA += (arrayA[i] - meanOfA)*(arrayA[i] - meanOfA);
            standardDevB += (arrayB[i] - meanOfB)*(arrayB[i] - meanOfB);
            pearsons += ((arrayA[i] - meanOfA)*(arrayB[i] - meanOfB));
        }
    }
    // finish calculating SD and pearsons.
    standardDevA = sqrt(standardDevA / size);
    standardDevB = sqrt(standardDevB / size);
    pearsons = ((pearsons / size) / (standardDevA*standardDevB));
    // free it
    free(arrayA);
    free(arrayB);
    // print it out
    printf("sumA = %lf, standDevA = %lf, averageA = %lf\nsumB = %lf, standDevB = %lf, averageB = "
           "%lf\nPearsons = %lf\n", sumOfA, standardDevA, meanOfA, sumOfB, standardDevB, meanOfB, pearsons);
}
```