CO524 Parallel Computing
Performance of Parallel Algorithms

In the raw code provided, the two variables feed two two threads and the average execution times of the threads are calculated. Variables defined as below.

```
int u[2] __attribute__ ((aligned (64)));
int at1 __attribute__ ((aligned (64)));
int at2 __attribute__ ((aligned (64)));
```

`at1` and `at2` variables' memory addresses are aligned to multiplications of 64. `u[0]` and `u[1]` are two integers which their addresses adjacent to each other by 4 bytes. This adjacency nature of `u[0]` and `u[1]` make them to fall into a same block of memory when cache line is fetched from the memory. When considering a processor with two cores and dedicated L1 data caches to each of them. The two variables `u[0]` and `u[1]` both appear in same line the both caches. When one core is accessing a one variable and do a write to that the cache coherence protocol will invalidate the whole line in all the caches. The L1 cache is write-through which is immediately writing to L2 shared cache for all cores. This results the L1 cache to read L2 when every time a core reads from it because of the copy in the L1 invalidation due to write from the other thread. The writes from two threads will invalidate the lines of both cache repetitively. This will cause a extra memory access time to the cache making its performance low.

Using aligned attribute for single variable does not guarantee the aligned memory addresses of variables are adjacent. Addresses can be scatter in the memory. To overcome this we can use an array with an aligned memory address. Accessing different elements with predefined distance we can guarantee that the addresses are separated with that distance.

```
int var_array1024[SIZE][1024] __attribute__ ((aligned(1024)));
```

The use of 2D array is to make sure the elements does not appear in the cache from the previous fetching. A large alignment is used because it will be align to all small factors (16, 32, 64, 128, 256) of it. The column size use as a multiple of the alignment to align all the rows to same alignment.

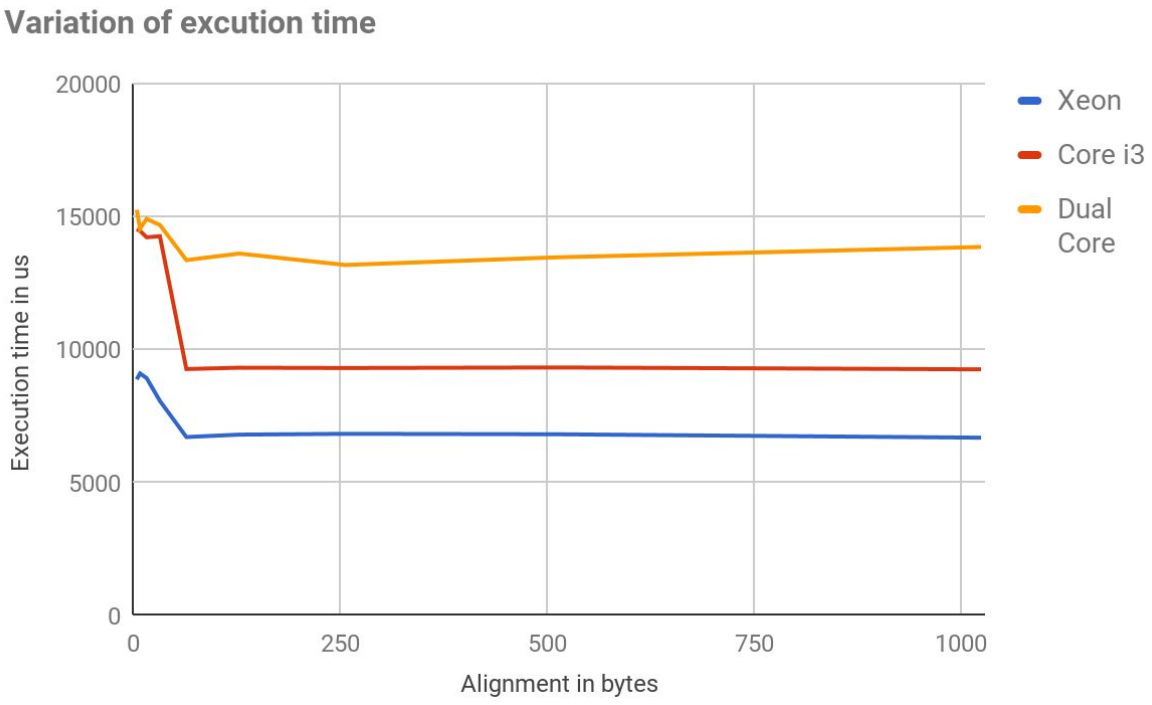
The result shown below is the program running on two core four core i3 machine. The addresses are shown in decimal and the difference of the addresses of the two element are also showed. Finally the average execution time shown in microseconds.

```
at4     (4299648 and 4299652)   diff = 4        Avg=14532
at8     (4303744 and 4303752)   diff = 8        Avg=14461
at16    (4307840 and 4307856)   diff = 16       Avg=14222
at32    (4311936 and 4311968)   diff = 32       Avg=14262
at64    (4316032 and 4316096)   diff = 64       Avg=9266
at128   (4320128 and 4320256)   diff = 128      Avg=9316
at256   (4324224 and 4324480)   diff = 256      Avg=9304
at512   (4328320 and 4328832)   diff = 512      Avg=9325
at1024  (4332416 and 4333440)   diff = 1024     Avg=9255
NULL    (0 and 0)               diff = 0        Avg=9795
```

We can see a dramatic reduction of execution time of two threads in 64 bytes alignment.

Below shows a graph of execution times with alignment of variables in different machines.

- Xeon - 32 core, 64 thread, L1d 32KB, L2 256KB and L3 20MB
- Core i3 - 2 core, 4 thread, L1d 32KB, L2 256KB and L3 3MB
- Dual core - 2 core, 2 thread, L1d 32KB, L2 256KB and L3 2MB



All three machines shows large increase of performance when switching to the 64 bytes alignment between variables and continue that performance to larger alignments also. This shows that the invalidation not happen in this scenario. That means the two variables are in two separate cache lines. This makes writing to one does not invalidate the other variable. So we can see that the cache line of L1 cache is 64 bytes long.