Cache Awareness

Course Level:
CS1/CS2

PDC Concepts Covered:

PDC Concept	Bloom Level
Locality	C
False Sharing	C

Programming Knowledge Prerequisites:
- Know how to compile Java
- Be able to understand loops and functions

Tools Required:
For Java: Java Development Kit (JDK) 8

Activity:

Consider the code below. It creates an array of size `SIZE*SIZE` and then loops over the array setting each element to zero. The time it takes to loop over the array is measured and printed out. The loop is implemented with two for loops, each iterating from `0` to `SIZE`. This lets us think of the array as a matrix with `SIZE` rows and `SIZE` columns. Then `i` represents the row index and `j` represents the column index. The expression `i*SIZE+j` calculates the actual index in the array. As we can see from the loops, the matrix is zeroed row by row.

Compile and run the following code

```java
public class CacheAwareness
{
        public static void main(String[] args)
        {
                final int SIZE = 20000;
                int[] array = new int[SIZE*SIZE];

                long start = System.nanoTime(); //get the current time

                for(int i = 0; i < SIZE; i++)
                        for(int j = 0; j < SIZE; j++)
                                array[i*SIZE+j]=0;

                long finish = System.nanoTime();

```

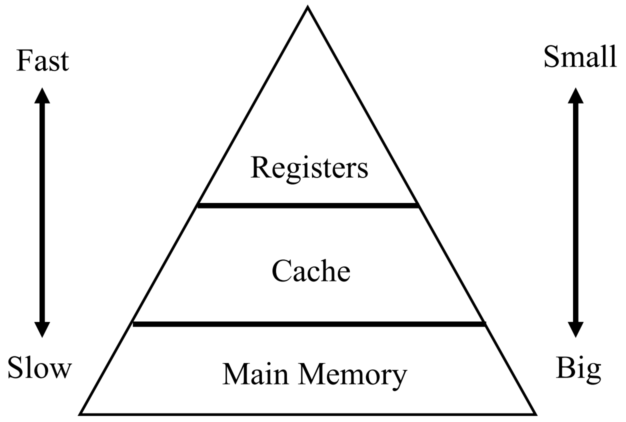
```
                  //this prints out the time in seconds between
                  //the two calls to nanoTime
                  System.out.println("Time: "+(finish-start)/1e9);
        }
}
```

Note: You must run using `java -Xmx3G CacheAwareness`

What if we want to zero the matrix column by column? This is as simple as swapping the two for loops. Do this, recompile, and run it. What is going on? Why would simply changing how we loop over the matrix make it take so much longer?

The cause of this requires us to look at the memory hierarchy. There are three main levels of memory:

- Registers: This is where additions and multiplications take place, fast and small
- Cache: Between Registers and Main Memory, intermediate speed and size
- Main Memory: RAM, large and slow



When the processor requires data it first checks to see if it is in the cache. If it finds the data in the cache then this is known as a cache hit. If it doesn't find the data then this is a cache miss and the processor must go out to main memory to get the data. It can take ten times longer to get the data on a cache miss than a cache hit.

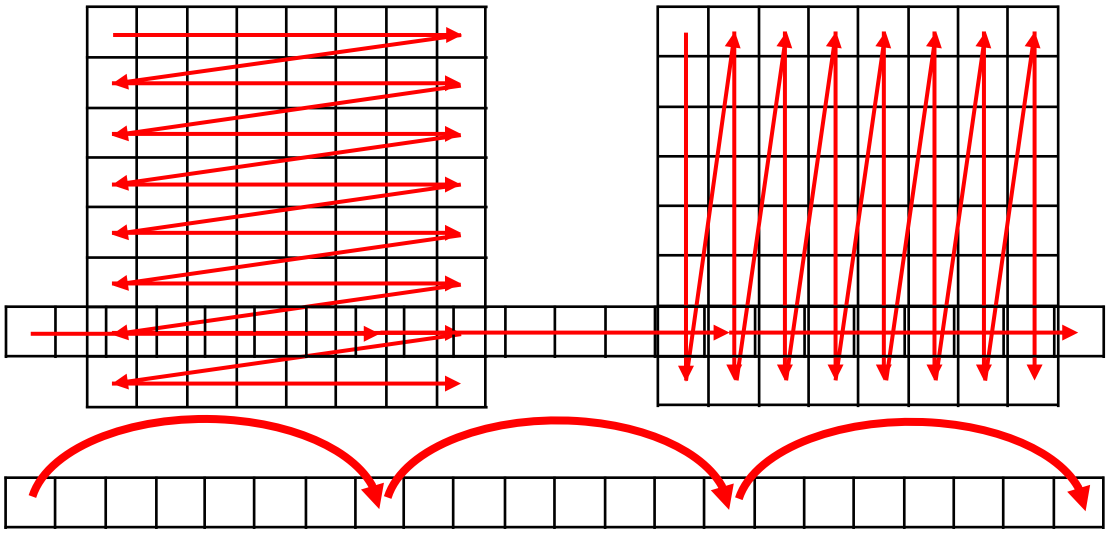
When the processor gets memory it does not get a single byte. It gets a larger block of memory called a cache line. This is generally 64 bytes. This is the reason for the large difference in speed between the two programs.

First consider the row by row program. It loads a cache line of the array from main memory, a cache miss, and sets the first element to zero. When it goes to set the second element to zero the cache line is already in the cache, so we get a cache hit. The next few elements will also be cache hits until we iterate far enough that the elements are in a different cache line.

Now consider the column by column program. The first element results in a cache miss like the row by row program. However, the next element will also be a cache miss as the elements are not next to each other in memory. Every single element results in a cache miss when we loop over the columns.

This is the cause of the performance difference. The row by row program only has to wait on a slow cache miss every eight or so elements while the column by column program cache misses on every element.

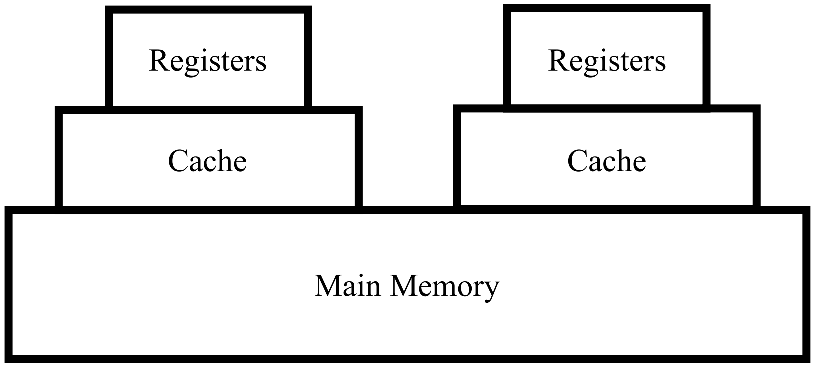
As programmers, we cannot change how the processor fetches memory, but by changing how we access our data we can make sure the cache and cache lines help us instead of hurting us.



Now let us see how cache lines might affect parallel programming. Consider the following program:

```
public class CacheAwareness
{
        public static void main(String[] args)
        {
                final int NUM_THREADS = 1;
                int[] array = new int[NUM_THREADS];
                for(int i = 0; i < NUM_THREADS; i++)
                        array[i]=0;

                long start = System.nanoTime(); //get the current time
                //#omp parallel num_threads(NUM_THREADS)
shared(NUM_THREADS,array)
                {
                    int index = Pyjama.omp_get_thread_num();
                    for(int i = 0; i < 1e9; i++)
                        array[index]++;

                }


                long finish = System.nanoTime();

                //this prints out the time in seconds between
                //the two calls to nanoTime
                System.out.println("Time: "+(finish-start)/1e9);
        }
}
```

There are `NUM_THREADS` threads that each increments their own int in a loop. Try running the program with 1, 2, and more threads. The reason for the decrease in performance is once again due to cache lines. Now this should sound wrong. After all this time `array` fits on a single cache line. Shouldn't this be very good for performance? The problem is that now we are dealing with multiple threads. The threads will be running on their own cores and each core has its own cache. When the first thread reads `array[0]` it loads the cache line into the first processor's cache. At the same time the second thread reads `array[1]` and this loads the same cache line into the second processor's cache. At this point there is not a problem. The same cache line can be in two different caches at the same time. The issue occurs when either thread writes the incremented value back to `array`. Now the cache line in the other processor's cache is wrong. The other processor must now get the updated cache line from main memory, a cache miss. Now you might think that this is unnecessary in this instance as the first thread does not care about the second thread's `array` value and vise versa. However, remember that memory transfers work at the cache line level. If thread one kept its cache line despite thread two updating its value, then when thread one writes the cache line back to main memory it would overwrite thread two's value with the old value.



This phenomenon is known as false sharing. The way to prevent it is to make sure each thread is working on different cache lines. To this end consider a modification to our previous code:

```
public class CacheAwareness
{
        public static void main(String[] args)
        {
                final int NUM_THREADS = 2;
                final int SPACING = 1;
                int[] array = new int[NUM_THREADS*SPACING];
                for(int i = 0; i < NUM_THREADS; i++)
```

```
                       array[i*SPACING]=0;

               long start = System.nanoTime(); //get the current time
               //#omp parallel num_threads(NUM_THREADS) shared(NUM_THREADS,
SPACING, array)
               {
                   int index = Pyjama.omp_get_thread_num()* SPACING ;
                   for(int i = 0; i < 1e9; i++)
                       array[index]++;

               }

               long finish = System.nanoTime();

               //this prints out the time in seconds between
               //the two calls to nanoTime
               System.out.println("Time: "+(finish-start)/1e9);
        }
}
```

Now the integers in `array` do not have to be next to each other. Instead they have indexes that are a multiple of `SPACING`. Play around with different `SPACING` sizes; once it gets big enough the slow performance should go away as each value is on its own cache line.