CPEG 455 Lab 1

Shane Cincotta

9/11/19

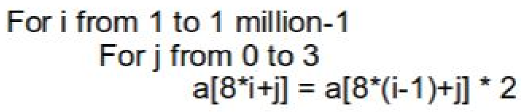
Abstract:

The goal of this lab is to implement C code in 3 variations which implement the same workload while having non-trivial speed differences. The speed difference should not be the result of artificially introduced overhead or underload. By implementing 3 versions of C code (while including the same workload), I was able to develop functionally identical code, while resulting in different run times.

Detailed Strategy:

To begin this lab, I first had to find a way to measure time that would be highly accurate. To achieve this, I used the *time* library and called the *time* method with a seed, this allows me access to another method, *gettimeofday()*. Before each of my implementations ran, I called the *gettimeofday()* method at the start and end of my code. I then computed the difference between the two times, the resulting time differential is the total time it took my implementation to complete.

After I had a way to keep track of time, I had to implement the control case for my C code. The control code is as follows-



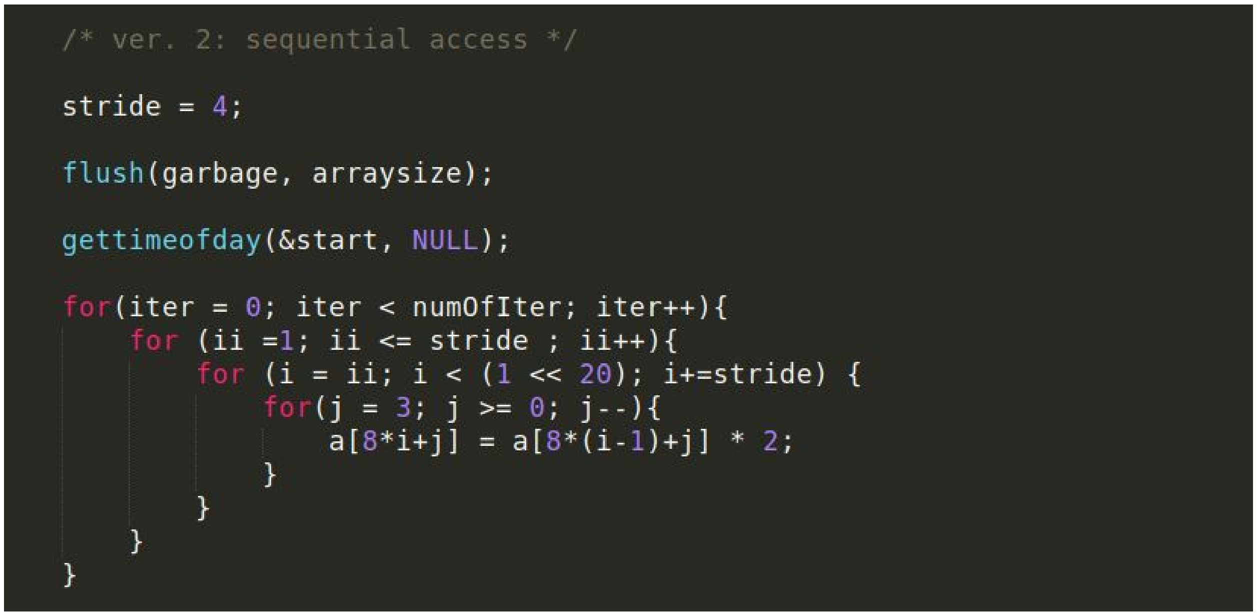
```
For i from 1 to 1 million-1
        For j from 0 to 3
                a[8*i+j] = a[8*(i-1)+j] * 2
```

Now I had to create different implementations of the preceding code, while maintaining the same workload. The first idea I hypothesised was to use a stride.

Hypothesis 1

A stride controls the increment of the variable you're iterating over in a *for loop*. By default, the stride is one, however changing the stride results in a difference in execution time. This occurs because of spatial locality. Spatial locality refers to the access of data elements which are stored close together in memory. In computing, if one memory address is accessed, there is a high probability that a nearby memory address will be accessed as well (such as a *for loop* iterating over an array). This principle of spatial locality can be utilized by a cache. If a data element is being loaded from memory and loaded into cache, the processor can load succeeding elements into the cache as well, anticipating that they will be accessed in the near future.

If stride is 1, the processor will add the first data element to the cache, as well as for the next 15 data elements. Thus in the *for loop*, many of the data elements that are being accessed will already be loaded into the cache (due to spatial locality). This concept can be used to slow a program down as well. In my code, I set the stride to 4. The start of the process is similar, the first data element is accessed, and the next 15 data elements are loaded into the cache. I hypothesize that, since my stride is 4, there will be many cache misses because the *for loop* skips over data elements that were already loaded into the cache.

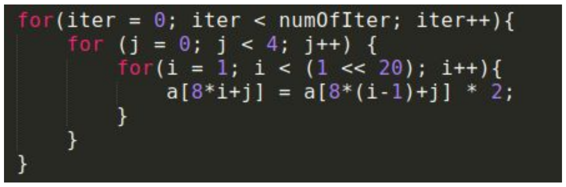


```
/* ver. 2: sequential access */

stride = 4;

flush(garbage, arraysize);

gettimeofday(&start, NULL);

for(iter = 0; iter < numOfIter; iter++){
    for (ii =1; ii <= stride ; ii++){
        for (i = ii; i < (1 << 20); i+=stride) {
            for(j = 3; j >= 0; j--){
                a[8*i+j] = a[8*(i-1)+j] * 2;
            }
        }
    }
}
```

Hypothesis 2

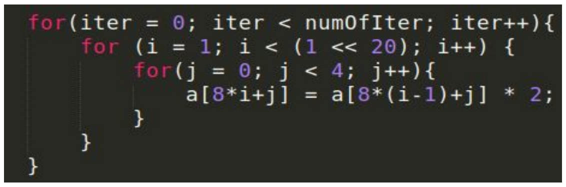
My second hypothesis was that changing the order of the 2 nested *for loops* would have an impact on execution time, most likely making the program take longer to run. For this method, I swapped the 2 innermost *for loops* from the preceding example, and removed the stride.

a.



```
for(iter = 0; iter < numOfIter; iter++){
    for (j = 0; j < 4; j++) {
        for(i = 1; i < (1 << 20); i++){
            a[8*i+j] = a[8*(i-1)+j] * 2;
        }
    }
}
```

b.

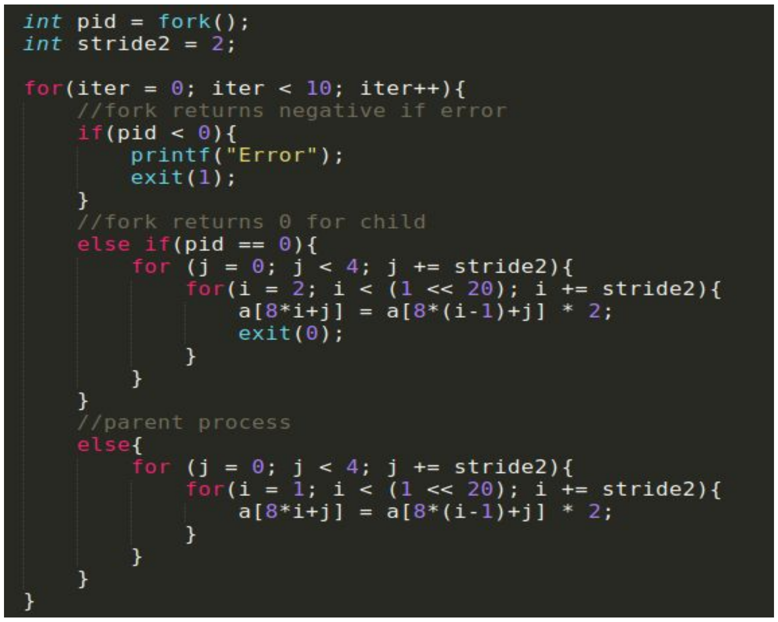


```
for(iter = 0; iter < numOfIter; iter++){
    for (i = 1; i < (1 << 20); i++) {
        for(j = 0; j < 4; j++){
            a[8*i+j] = a[8*(i-1)+j] * 2;
        }
    }
}
```

In example *a,* the innermost *for loop* iterates over the variable *i* 1million times, which then gets repeated 3 more times by the *j for loop*. In example *b*, the *j for loop* iterates 4 times, which then gets repeated almost a million times. I hypothesize example *b* is more efficient because it accesses 4 data elements many times, each time the data elements will be present in the cache.

Hypothesis 3

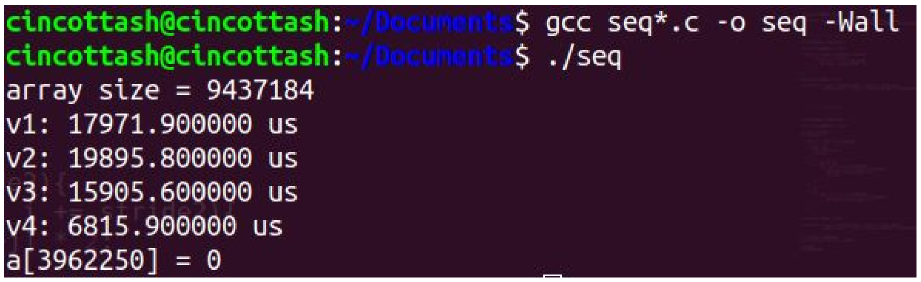
My last hypothesis was if I included parallelization in my program, I could reduce the execution time. To achieve parallelization, I forked my program to create a child and parent process. I set my child process to loop over the even indices, and the parent process to loop over the odd indices.



```
int pid = fork();
int stride2 = 2;

for(iter = 0; iter < 10; iter++){
    //fork returns negative if error
    if(pid < 0){
        printf("Error");
        exit(1);
    }
    //fork returns 0 for child
    else if(pid == 0){
        for (j = 0; j < 4; j += stride2){
            for(i = 2; i < (1 << 20); i += stride2){
                a[8*i+j] = a[8*(i-1)+j] * 2;
                exit(0);
            }
        }
    }
    //parent process
    else{
        for (j = 0; j < 4; j += stride2){
            for(i = 1; i < (1 << 20); i += stride2){
                a[8*i+j] = a[8*(i-1)+j] * 2;
            }
        }
    }
}
```

Results:

At the end of each implementation, I kept track of the execution time using the method described in the beginning. I printed this time out to the console in microseconds.



```
cincottash@cincottash:~/Documents$ gcc seq*.c -o seq -Wall
cincottash@cincottash:~/Documents$ ./seq
array size = 9437184
v1: 17971.900000 us
v2: 19895.800000 us
v3: 15905.600000 us
v4: 6815.900000 us
a[3962250] = 0
```

The results show that version 4 (parallelization) was the fastest in terms of execution speed. This confirms my original hypothesis that parallelization would reduce execution time, but also showed that it was the most adept method at doing so. The rest of the versions all had similar execution speeds, but it is worth noting that version 3 (switching *for loops*) did not show an increase in execution time. This might have resulted from not using enough iterations of the *for loops*. Both the remaining 2 versions had identical execution speeds.

Conclusion:

After implementing 3 identical programs which implement the same workload, I was successfully able to find non-trivial speed differences. The speed difference with the largest magnitude resulted from my fourth implementation (the parallelization). The other 2 implementations had about the same speed.

If I was to do this lab again, I would try creating this program with even more threads to make it run even faster as in version 4. I would also try experimenting with longer strides as my stride implementation did not change the execution time as drastically as I expected.