Activity 2.0: Loop Ordering and False Sharing

This activity reinforces the concept of reduction and the caching principles taught in the lecture on Cilk on Sep. 18. It is recommended that you run this on the CS machines `gradx.cs.jhu.edu` or `ugradx.cs.jhu.edu` . The results make sense here. It is OK to run this on any machine that has at least 4 cores. If you run on different machine, you may end up with slightly different results. It is OK if your results don't track exactly with the expected findings. On my M1 laptop the results get confusing.

Due date: Thursday September 28, 2023, 9:00 pm EDT.

Instructions for Submission: Submit via Gradescope.

The Program

The is a nested loop program that counts the number of occurences of a list of tokens in an array of elements. This is a common computing pattern in data analytics. This could be used to count the number of messages sent in a network from a set of sources.

There are two serial versions of the program. These are:

- `countTokensElementsFirst` : loop over the larger elements array in the outer loop and the smaller tokens array in the inner loop.
- `countTokensTokensFirst` : loop over the larger elements array in the outer loop and the smaller tokens array in the inner loop.

This is not a 2-d dimensional data structure like our previous examples. It is 2 separate arrays.

Programming

Complete the *TODO* instructions in [activities/tokens_omp.cpp].

1. Add `parallel for` directives to functions:
    - `omp_countTokensElementsFirst`
    - `omp_countTokensTokensFirst`
2. Add `parallel for` and `reduction` directive for the array `token_counts` for:
    - `omp_countTokensElementsFirst_reduce`
    - `omp_countTokensTokensFirst_reduce`

The array reduction clause was added to OpenMP and requires one to specify the length of the array. A simple example is provided in https://dvalters.github.io/optimisation/code/2016/11/06/OpenMP-array_reduction.html (https://dvalters.github.io/optimisation/code/2016/11/06/OpenMP-array_reduction.html).

1. Unroll the loop 8 times in `unroll_omp_countTokensElementsFirst_reduce` . You may assume the the tokens array is evenly divisible by 8.

On the `gradx.cs.jhu.edu` machine after I added this code, I got the timing results

```
Tokens First time: 8.07097 seconds
Elements First time: 6.93468 seconds
OMP Tokens First time: 2.10465 seconds
OMP Elements First time: 1.78919 seconds
OMP Tokens First Reduce time: 1.99353 seconds
OMP Elements First Reduce time.: 1.78073 seconds
Unroll OMP Elements First Reduce time.: 0.926184 seconds
```

building with the command line

```
g++ -O0 -fopenmp tokens_omp.cpp
```

Compiling with `-O0` turns off all compiler optimizations to prevent the compiler from making unknown optimizations that would confound our results.

Questions

Provide brief but complete answers to the following questions in the following cell.

1. Why is it more efficient to iterate over the `tokens` in the inner loop?

(*Note: Access to both arrays is sequential. This is a question of cache capacity and cache misses.*)

In our program, `num_els = 4096` and `num_tokens = 128`. The number of elements is 32 times greater than the number of tokens.

In `countTokensElementsFirst`, for each element in the elements array, the function goes through the tokens array. Since `num_tokens` is relatively small, there's a good chance that most of tokens array fits into the CPU cache. This means that, after the first few iterations for the first element, subsequent iterations for the next elements will likely benefit from cache hits when accessing the tokens array, making the memory accesses faster.

Conversely, in `countTokensTokensFirst`, for each token, the function iterates over the entire elements array. Given the size of the elements array, it's less likely that the entire array will fit into the CPU cache, leading to more cache misses and slower memory accesses.

2. Of the functions `omp_countTokensElementsFirst` and `omp_countTokensTokensFirst`

1. Which function performs (unsafe) sharing in the tokens array?
2. Which function assigns different elements of the token array to different threads?

1. The function `omp_countTokensElementsFirst` could engage in unsafe sharing of the tokens array. The outer loop here compares one element with each token in the tokens array. Compartively, `omp_countTokensElementsFirst` assigns one token to memory and then compares it with each element in the elements array. Since the inner loop has access to all the tokens in the `omp_countTokensElementsFirst` function, it has more potential for unsafe sharing of the tokens array.

2. The function `omp_countTokensTokensFirst` assigns different elements of the tokens array to different threads. This is because we are parallelizing the outer loop and the `parallel for` directive is being applied to this outer loop. Each thread is then responsible for a range of `tok` values and will compare them against all the elements.

3. For the function that assigns different tokens to different threads, how does false sharing arise? Be specific about the memory access pattern or include a drawing.

In the function `omp_countTokensElementsFirst`, the problem arises when two or more threads try to update counts for tokens that reside on the same cache line in the token_counts array.

Even if they're updating different tokens, if those tokens' counts are close enough in memory to be on the same cache line, the hardware may end up constantly invalidating and refreshing that cache line because of the simultaneous updates. This is false sharing.

Imagine a scenario where you have an array called token_counts with a size of 8. This means you have 8 different tokens you're counting. For simplicity, let's assume each unsigned int takes 4 bytes and a cache line in the system is 32 bytes.

Now, let's say that two threads, Thread A and Thread B, are running concurrently due to the OpenMP parallelization. Thread A is currently processing some elements that correspond to Token_1, while Thread B is processing elements that correspond to Token_3.

Given our assumption of cache line size and the size of an unsigned int, Token_0 through Token_7 all fit within a single cache line:
Alt text

When Thread A increments the count for Token_1, it brings that cache line (with all tokens from Token_0 to Token_7) into its local cache and updates the value. Almost simultaneously, Thread B does the same for Token_3.

Now, from the perspective of the cache coherence mechanism in the CPU:

```
Thread A's cache sees an update to the cache line for Token_1.
Thread B's cache sees an update to the same cache line for Token_3.
```

Even though Thread A and Thread B are updating different tokens, they are both modifying the same cache line, which results in cache invalidations. The cache line will be marked as "dirty" in one cache and will be invalidated in other caches. This means that the next time another thread (or even the same thread) tries to access any token from that cache line, it may have to fetch the cache line from a higher-level cache or even from main memory, which is much slower than accessing it from the local cache.

This constant invalidation and refreshing of the cache line due to multiple threads writing to it is what constitutes false sharing. The threads are not technically interfering with each other's data, but they are causing performance degradation due to cache line contention.

4. For the unrolled loop, why is it more efficient? What computations are avoided?

By unrolling the loop, we're able to perform more computations in each iteration.

At the end of each iteration, the looping variable's index needs to be incremented, tested against the loop condition and then branched back to the top of the looping code. With loop unrolling, we're able to reduce the number of times these operations are performed. In this activity's case, these "loop end" operations are reduced by a factor of 8.

The number of branches created are also less (because the number of iterations are less) and which results in the processor having more free space to process instructions.

Hence, unrolled loops are more efficient.

Appendix: Modified `activity2_tokens.cpp`

```
#include <iostream>
#include <chrono>
#include <omp.h>

// initialize elements to random integer values 0 to range-1
void initElements (unsigned int range, unsigned int num_els, unsigned int*
elements) {
    for (int i=0; i<num_els; i++)  {
        elements[i] = rand() % range;
    }
}

// initialize tokens to search.  again 0 to range-1
// note, we should probably enforce that tokens are unique. not important
for performance.
void initTokens (int range, int num_toks, unsigned int* tokens) {
    for (int i=0; i<num_toks; i++)  {
        tokens[i] = rand() % range;
    }
}

// initialize all token counts to zero
void initCounts (int num_toks, unsigned int* token_counts) {
    for (int i=0; i<num_toks; i++)  {
        token_counts[i] = 0;
    }
}

// count the number of appearances of each token in the data
void countTokensElementsFirst (unsigned int num_els, unsigned int num_toke
ns,
                  unsigned int* elements, unsigned int* tokens, unsigned i
nt* token_counts) {

    /* for all elements in the array */
    for (int el=0; el<num_els; el++) {
        /* for all tokens in the list */
        for (int tok=0; tok<num_tokens; tok++) {
            /* update the count for the token */
            if (elements[el] == tokens[tok]) {
                token_counts[tok]++;
            }
        }
    }
}

// count the number of appearances of each token in the data
```

```
void countTokensTokensFirst (unsigned int num_els, unsigned int num_token
s,
                             unsigned int* elements, unsigned int* tokens,
unsigned int* token_counts) {

    /* for all tokens in the list */
    for (int tok=0; tok<num_tokens; tok++) {
        /* for all elements in the array */
        for (int el=0; el<num_els; el++) {
            /* update the count for the token */
            if (elements[el] == tokens[tok]) {
                token_counts[tok]++;
            }
        }
    }
}

// count the number of appearances of each token in the data
void omp_countTokensElementsFirst (unsigned int num_els, unsigned int num_
tokens,
                  unsigned int* elements, unsigned int* tokens, unsigned i
nt* token_counts) {

    //TODO parallel for
    /* for all elements in the array */
    #pragma omp parallel for
    for (int el=0; el<num_els; el++) {
        /* for all tokens in the list */
        for (int tok=0; tok<num_tokens; tok++) {
            /* update the count for the token */
            if (elements[el] == tokens[tok]) {
                token_counts[tok]++;
            }
        }
    }
}

// count the number of appearances of each token in the data
void omp_countTokensTokensFirst (unsigned int num_els, unsigned int num_to
kens,
                             unsigned int* elements, unsigned int* tokens,
unsigned int* token_counts) {

    //TODO parallel for
    /* for all tokens in the list */
    #pragma omp parallel for
    for (int tok=0; tok<num_tokens; tok++) {
        /* for all elements in the array */
```

```
        for (int el=0; el<num_els; el++) {
            /* update the count for the token */
            if (elements[el] == tokens[tok]) {
                token_counts[tok]++;
            }
        }
    }
}

// elements first with reduction
void omp_countTokensElementsFirst_reduce (unsigned int num_els, unsigned i
nt num_tokens,
                            unsigned int* elements, unsigned int* tokens,
unsigned int* token_counts) {

    //TODO parallel for reduction
    /* for all elements in the array */
    #pragma omp parallel for reduction(+:token_counts[:num_tokens])
    for (int el=0; el<num_els; el++) {
        /* for all tokens in the list */
        for (int tok=0; tok<num_tokens; tok++) {
            /* update the count for the token */
            if (elements[el] == tokens[tok]) {
                token_counts[tok]++;
            }
        }
    }
}

// tokens first with reduction
void omp_countTokensTokensFirst_reduce (unsigned int num_els, unsigned int
num_tokens,
                            unsigned int* elements, unsigned int* tokens,
unsigned int* token_counts) {

    //TODO parallel for reduction
    /* for all tokens in the list */
    #pragma omp parallel for reduction(+:token_counts[:num_tokens])
    for (int tok=0; tok<num_tokens; tok++) {
        /* for all elements in the array */
        for (int el=0; el<num_els; el++) {
            /* update the count for the token */
            if (elements[el] == tokens[tok]) {
                token_counts[tok]++;
            }
        }
    }
}
```

```
// unroll tokens elements first with reduction
void unroll_omp_countTokensElementsFirst_reduce (unsigned int num_els, uns
igned int num_tokens,
                              unsigned int* elements, unsigned int* tokens,
unsigned int* token_counts) {

    //TODO parallel for reduction
    /* for all elements in the array */
    #pragma omp parallel for reduction(+:token_counts[:num_tokens])
    for (int el=0; el<num_els; el++) {
        /* for all tokens in the list */
        for (int tok=0; tok<num_tokens; tok+=8) {
            //TODO unroll loop 8 times
            /* update the count for the token */
            if (elements[el] == tokens[tok]) {
                token_counts[tok]++;
            }
            if (elements[el] == tokens[tok + 1]) {
                token_counts[tok + 1]++;
            }
            if (elements[el] == tokens[tok + 2]) {
                token_counts[tok + 2]++;
            }
            if (elements[el] == tokens[tok + 3]) {
                token_counts[tok + 3]++;
            }
            if (elements[el] == tokens[tok + 4]) {
                token_counts[tok + 4]++;
            }
            if (elements[el] == tokens[tok + 5]) {
                token_counts[tok + 5]++;
            }
            if (elements[el] == tokens[tok + 6]) {
                token_counts[tok + 6]++;
            }
            if (elements[el] == tokens[tok + 7]) {
                token_counts[tok + 7]++;
            }
        }
    }
}


int main() {

    const unsigned int range = 4096;
```

```
    const unsigned int num_tokens = 128;
    const unsigned int num_elements = 4096*256;
    const unsigned int loop_iterations = 16;

    unsigned int tokens[num_tokens];
    unsigned int elements[num_elements];
    unsigned int token_counts[num_tokens];

    initElements(range, num_elements, elements);
    initTokens(range, num_tokens, tokens);
    initCounts(num_tokens, token_counts);

    omp_set_num_threads(4);

    // run once to warm the cache
    countTokensTokensFirst(num_elements, num_tokens, elements, tokens, tok
en_counts);

    // countTokensTokensFirst
    // Start the timer
    auto start = std::chrono::high_resolution_clock::now();
    for(int j=0; j<loop_iterations; j++) {
        countTokensTokensFirst(num_elements, num_tokens, elements, tokens,
token_counts);
    }
    // Stop the timer
    auto end = std::chrono::high_resolution_clock::now();
    // Calculate the duration
    std::chrono::duration<double> duration = end - start;
    // Print the duration in seconds
    std::cout << "Tokens First time: " << duration.count() << " seconds"
<< std::endl;

    // reset counts only works right if running one loop_iteration
    initCounts(num_tokens, token_counts);

    // run once to warm the cache
    countTokensElementsFirst(num_elements, num_tokens, elements, tokens, t
oken_counts);


    // countTokensElementsFirst
    start = std::chrono::high_resolution_clock::now();
    for(int j=0; j<loop_iterations; j++) {
        countTokensElementsFirst(num_elements, num_tokens, elements, token
s, token_counts);
    }
    end = std::chrono::high_resolution_clock::now();
```

```
    duration = end - start;
    std::cout << "Elements First time: " << duration.count() << " seconds"
<< std::endl;

    // reset counts only works right if running one loop_iteration
    initCounts(num_tokens, token_counts);


    // omp_countTokensTokensFirst
    start = std::chrono::high_resolution_clock::now();
    for(int j=0; j<loop_iterations; j++) {
        omp_countTokensTokensFirst(num_elements, num_tokens, elements, tok
ens, token_counts);
    }
    end = std::chrono::high_resolution_clock::now();
    duration = end - start;
    std::cout << "OMP Tokens First time: " << duration.count() << " second
s" << std::endl;

    // reset counts only works right if running one loop_iteration
    initCounts(num_tokens, token_counts);


    // omp_countTokensElementsFirst
    start = std::chrono::high_resolution_clock::now();
    for(int j=0; j<loop_iterations; j++) {
        omp_countTokensElementsFirst(num_elements, num_tokens, elements, t
okens, token_counts);
    }
```

Appendix: Output of modified `activity2_tokens.cpp`

```
Tokens First time: 4.91057 seconds
Elements First time: 4.76158 seconds
OMP Tokens First time: 0.604227 seconds
OMP Elements First time: 0.558478 seconds
OMP Tokens First Reduce time: 0.552167 seconds
OMP Elements First Reduce time.: 0.572344 seconds
Unroll OMP Elements First Reduce time.: 0.389798 seconds
```