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.

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.)
2. Of the functions `omp_countTokensElementsFirst` and `omp_countTokensTokensFirst`
   A. Which function performs (unsafe) sharing in the tokens array?
   B. Which function assigns different elements of the token array to different threads?
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.
4. For the unrolled loop, why is it more efficient? What computations are avoided?

Answers

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.)

The amount of `tokens` are less than `elements`. It is more efficient to iterate over the `tokens` in the inner loop because there are less chances of cache misses. The overhead would then be reduced when iterating throughout the inner loop.

1. Of the functions omp_countTokensElementsFirst and omp_countTokensTokensFirst A. Which function performs (unsafe) sharing in the tokens array?

`omp_countTokensElementsFirst` would perform (unsafe) sharing in the tokens array since if we distribute the elements into different threads, a data race could occur when

there is a scenario where two threads are updating the same element of `tokens_counts`. The result would either be much slower to process the updated cache or there would be an error in the token_counts with this function.

B. Which function assigns different elements of the token array to different threads?

`omp_countTokensTokensFirst` - This is due to the fact that we have parallelized the outerloop and in doing so, this function will be sending different elements of token to different threads.

1. 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.

Say that there were cores A and B, where they had shared memory (using the same cache line). This means that both cores can load from the same cache line, no problem. BUT False sharing arises when core A updates (in our case, increments an element of `token_counts`), Core B doesn't know what to do, or rather cannot do its computation(s) since the cache line is marked dirty (caused by core A's instructions) which invalidates it. Core A has to send out a an updated cache line to main memory and then core B has to update its cache line from the modified cache.

This memory access pattern causes a lot of flushes and loadings, which leads to the decrease effect of parallelism.

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

Unrolling the loop means the loop more efficient since there are less comparisons involved for the if statements to check if equal. Performing less computations would mean save you a lot more time.

THIS IS MY WORK FOLLOWING THE TODO INSTRUCTIONS IN ACTIVITY2_TOKENS.CPP

In [6]:
```
#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* eler
    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 p
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_tokens,
                  unsigned int* elements, unsigned int* tokens, unsigned int* t

    /* 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_tokens,
                             unsigned int* elements, unsigned int* tokens, unsi

    /* 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_token
                  unsigned int* elements, unsigned int* tokens, unsigned int* t

    //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_tokens,
```

```
                            unsigned int* elements, unsigned int* tokens, unsi

    //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 int nu
                            unsigned int* elements, unsigned int* tokens, unsi

    //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_
                            unsigned int* elements, unsigned int* tokens, unsi

    //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, unsigned
                            unsigned int* elements, unsigned int* tokens, unsi

    //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]++;
            # }

            // LOOP UNROLLING 8 TIMES HERE
            if (elements[el] == tokens[tok + 0]) {token_counts[tok + 0]++;}
            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, token_c

    // 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, toke
    }
    // 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" << st

    // 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, token_


// countTokensElementsFirst
start = std::chrono::high_resolution_clock::now();
for(int j=0; j<loop_iterations; j++) {
    countTokensElementsFirst(num_elements, num_tokens, elements, tokens, to
}
end = std::chrono::high_resolution_clock::now();
duration = end - start;
std::cout << "Elements First time: " << duration.count() << " seconds" << s

// 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, tokens,
}
end = std::chrono::high_resolution_clock::now();
duration = end - start;
std::cout << "OMP Tokens First time: " << duration.count() << " seconds" <<

// 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, tokens
}
end = std::chrono::high_resolution_clock::now();
duration = end - start;
std::cout << "OMP Elements First time: " << duration.count() << " seconds"

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

// omp_countTokensTokensFirst_reduce
start = std::chrono::high_resolution_clock::now();
for(int j=0; j<loop_iterations; j++) {
    omp_countTokensTokensFirst_reduce(num_elements, num_tokens, elements, t
}
end = std::chrono::high_resolution_clock::now();
duration = end - start;
std::cout << "OMP Tokens First Reduce time: " << duration.count() << " seco

// omp_countTokensElementsFirst_reduce
start = std::chrono::high_resolution_clock::now();
for(int j=0; j<loop_iterations; j++) {
    omp_countTokensElementsFirst_reduce(num_elements, num_tokens, elements,
}
end = std::chrono::high_resolution_clock::now();
duration = end - start;
std::cout << "OMP Elements First Reduce time.: " << duration.count() << " s
```

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

    // unroll_omp_countTokensElementsFirst_reduce
    start = std::chrono::high_resolution_clock::now();
    for(int j=0; j<loop_iterations; j++) {
        unroll_omp_countTokensElementsFirst_reduce(num_elements, num_tokens, el
    }
    end = std::chrono::high_resolution_clock::now();
    duration = end - start;
    std::cout << "Unroll OMP Elements First Reduce time.: " << duration.count()
}
```

```
  Cell In[6], line 5
    // initialize elements to random integer values 0 to range-1
    ^
SyntaxError: invalid syntax
```

In [7]:
```
My time results:

Tokens First time: 3.88932 seconds
Elements First time:  3.48921 seconds
OMP Tokens First time:  1.19023 seconds
OMP Elements First time:  1.00565 seconds
OMP Tokens First Reduce time:  1.13888 seconds
OMP Elements First Reduce time.: 1.00023 seconds
Unroll OMP Elements First Reduce time.: 0.55529 seconds
```

```
  Cell In[7], line 1
    My time results:
       ^
SyntaxError: invalid syntax
```

In [ ]: