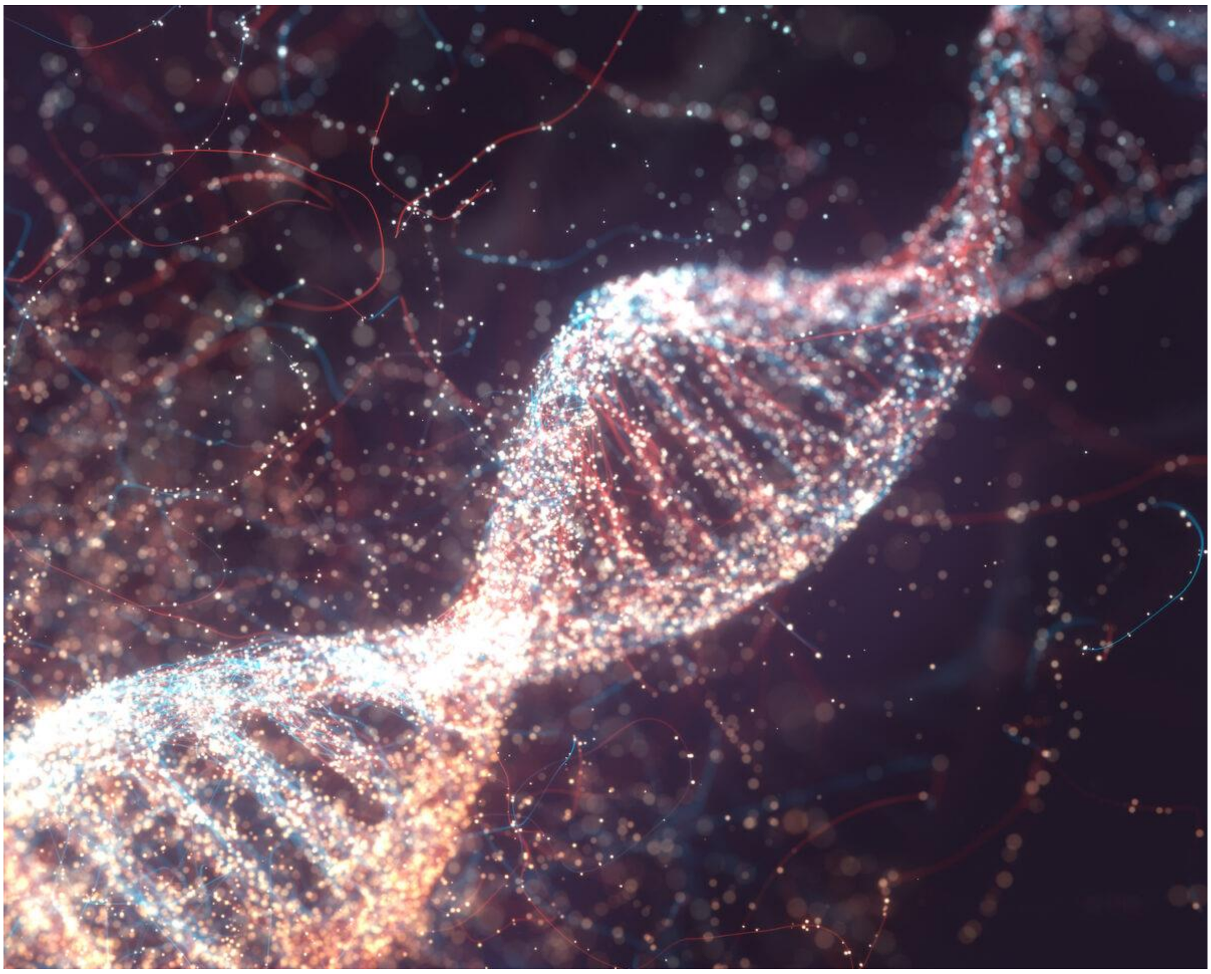
Parallelization Project



KMER SIGNATURES

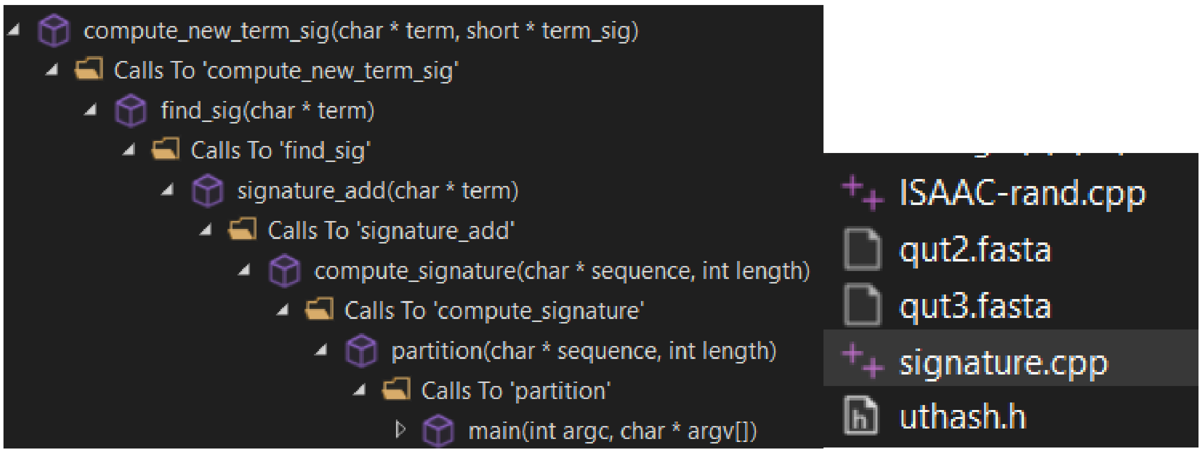
Jacob Fallows n10749527

CONTENTS

OVERVIEW

In the field of bioinformatics, Kmers are smaller subsets of a DNA or protein sequence. For example, the Kmer CTG is found in GTAGAGCTG. The appearance and frequency of a Kmer in a genome is used to compare and identify these biological sequences; referred to as a Kmer signature.

In the Kmer signatures application the list of genomes to be analysed is represented in a FASTA file. The program reads and interprets this file and generates Kmer signatures.



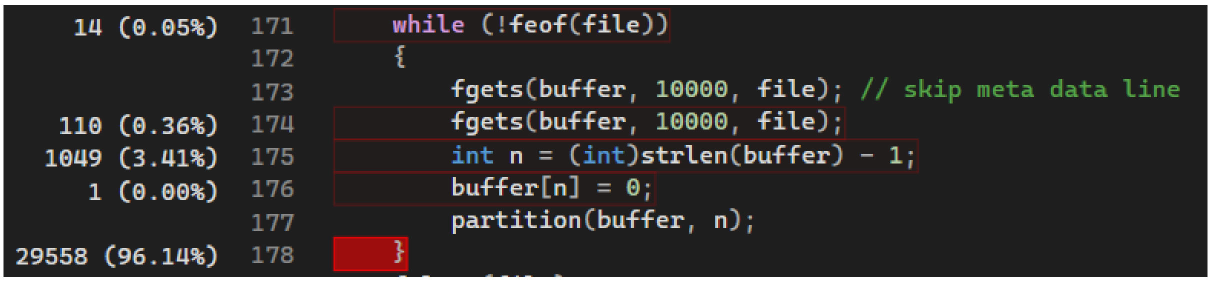
```
compute_new_term_sig(char * term, short * term_sig)
  Calls To 'compute_new_term_sig'
    find_sig(char * term)
      Calls To 'find_sig'
        signature_add(char * term)
          Calls To 'signature_add'
            compute_signature(char * sequence, int length)
              Calls To 'compute_signature'
                partition(char * sequence, int length)
                  Calls To 'partition'
                    main(int argc, char * argv[])
```

```
ISAAC-rand.cpp
qut2.fasta
qut3.fasta
signature.cpp
uthash.h
```

Above is the call hierarchy and the files used in the program. The main function gets each protein sequence from the FASTA file and parses it to the partition function. The partition function makes the sequence a more manageable size for our purpose and gives each partition to compute signature. The crucial function is compute_new_term_sig which uses the random number function from ISAAC-rand.cpp to generate a unique signature which is seeded with a term from the genome data and the length of each word. For this demonstration we will use a word length of 3, also known as a 3-mer.

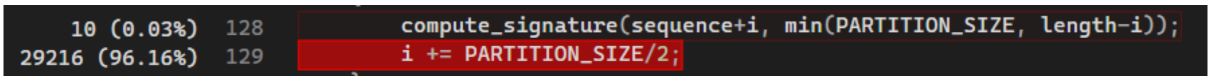
POTENTIAL PARALLELISM

Running the program using visual studio profiler will reveal existing loops and control flow constructs where parallelism of sufficient granularity may be found.



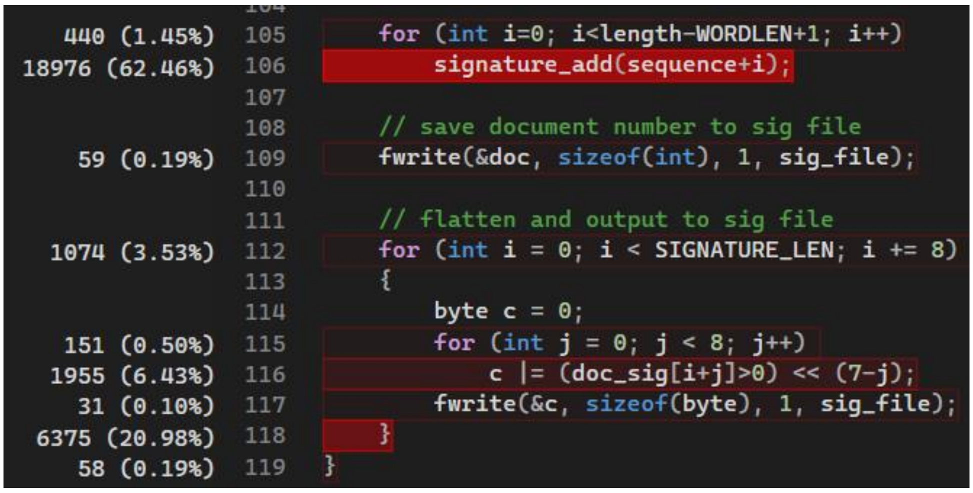
```
   14 (0.05%)      while (!feof(file))
                   {
                       fgets(buffer, 10000, file); // skip meta data line
  110 (0.36%)          fgets(buffer, 10000, file);
 1049 (3.41%)          int n = (int)strlen(buffer) - 1;
    1 (0.00%)          buffer[n] = 0;
                       partition(buffer, n);
29558 (96.14%)     }
```

Upon using the performance profiler in Visual Studio, we can see the major computation points in the software. Looking inside the main function, almost all the computation happens inside the while loop, which is reading in the data from the FASTA file and parsing it to the partition function. The program writes the signatures into the buffer which prevents it from being parallelized. Using a different method this section could be parallelized by removing the data dependency of the buffer variable.



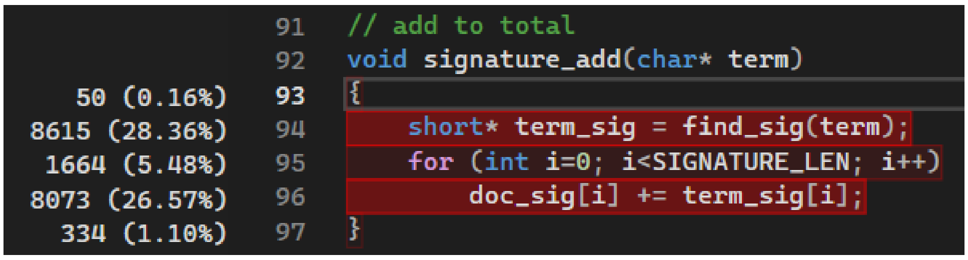
```
   10 (0.03%)          compute_signature(sequence+i, min(PARTITION_SIZE, length-i));
29216 (96.16%)         i += PARTITION_SIZE/2;
```

Looking in the next function, partition, most CPU usage is from the compute signature function. The visual studio profiler often shows the calculations on the line after hence the discrepancy.



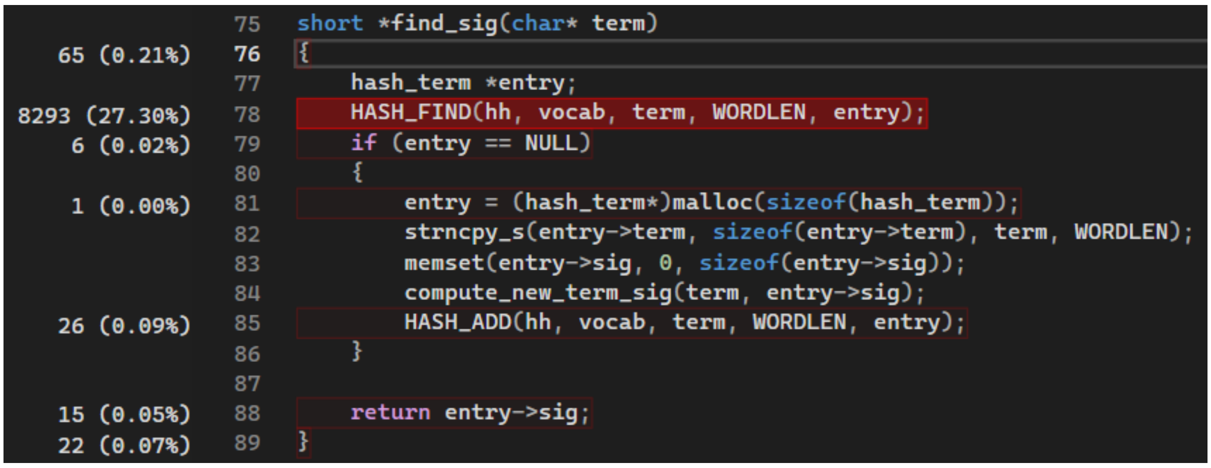
```
  440 (1.45%)      for (int i=0; i<length-WORDLEN+1; i++)
18976 (62.46%)         signature_add(sequence+i);

                   // save document number to sig file
   59 (0.19%)      fwrite(&doc, sizeof(int), 1, sig_file);

                   // flatten and output to sig file
 1074 (3.53%)      for (int i = 0; i < SIGNATURE_LEN; i += 8)
                   {
                       byte c = 0;
  151 (0.50%)          for (int j = 0; j < 8; j++)
 1955 (6.43%)              c |= (doc_sig[i+j]>0) << (7-j);
   31 (0.10%)          fwrite(&c, sizeof(byte), 1, sig_file);
 6375 (20.98%)     }
   58 (0.19%)  }
```

Looking inside compute signatures, most computation is from the signature add function. There is a significant bottleneck based around writing to the output file. Unfortunately, there is no potential parallelization here and is not of sufficient granularity, not to mention the Pareto principle also. The file writing section will need to be restructured to a parallel form.



```
                 91  // add to total
                 92  void signature_add(char* term)
   50 (0.16%)    93  {
 8615 (28.36%)   94      short* term_sig = find_sig(term);
 1664 (5.48%)    95      for (int i=0; i<SIGNATURE_LEN; i++)
 8073 (26.57%)   96          doc_sig[i] += term_sig[i];
  334 (1.10%)    97  }
```

The signature add function has two main parts. One is to get the signature whether it exists or not. The second is to compile the signatures for each word. Looking initially, there seem to be only input dependencies, however doc_sig is a global variable, so it won't hold up if this function is to be run in parallel.



```
                 75  short *find_sig(char* term)
   65 (0.21%)    76  {
                 77      hash_term *entry;
 8293 (27.30%)   78      HASH_FIND(hh, vocab, term, WORDLEN, entry);
    6 (0.02%)    79      if (entry == NULL)
                 80      {
    1 (0.00%)    81          entry = (hash_term*)malloc(sizeof(hash_term));
                 82          strncpy_s(entry->term, sizeof(entry->term), term, WORDLEN);
                 83          memset(entry->sig, 0, sizeof(entry->sig));
                 84          compute_new_term_sig(term, entry->sig);
   26 (0.09%)    85          HASH_ADD(hh, vocab, term, WORDLEN, entry);
                 86      }
                 87
   15 (0.05%)    88      return entry->sig;
   22 (0.07%)    89  }
```

The find sig function takes a significant portion of the computation. This whole function can be run in parallel, almost. Compute new term sig calls an external library so it is best practice to use a mutex around line 84. It is only called 26 times which tells us there were only 26 unique signatures in the FASTA file. Since it is a seeded random number generator and compute new term sig will always return the same signature given the same input it is not required to use the mutex.

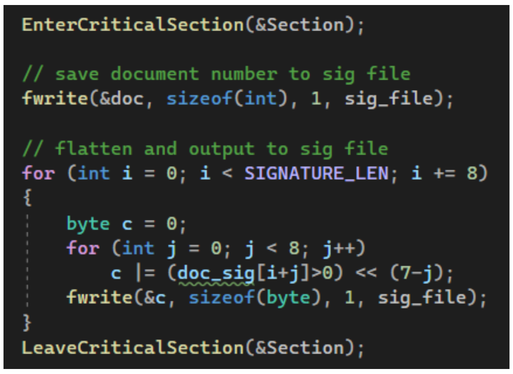
To transform the program into a parallel form, there are a couple of changes that need to happen. We will focus on the initial while loop in the main function while accommodating for the other functions. The main while loop will use an array to store each signature as a result, it will also have an id assigned to avoid race conditions. The other functions will write to the output file in a critical section. In this case the order of the results is non-critical but for testing purposes we need it to be. When successfully restructured, it will be scalable parallelism, scaling with the size of the FASTA file and the number of processors.

COMPUTATION

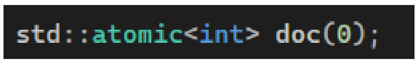
Each iteration of the while loop is divided up by the optimal number of processors. When running tests on my Core i-7 10750H the peak performance is at the number of physical cores, in this case, six cores. I used the Win32 library for parallel loops and synchronization.

Using a critical section to perform synchronization would work to eliminate the race conditions, however it would prevent the other threads from accessing it, nullifying the speedup. The race conditions will likely not appear since each protein sequence is not very long in this case.

I am using atomic integers to make sure the file writes are in order. I am also using win32 threads and critical sections.



```
EnterCriticalSection(&Section);

// save document number to sig file
fwrite(&doc, sizeof(int), 1, sig_file);

// flatten and output to sig file
for (int i = 0; i < SIGNATURE_LEN; i += 8)
{
    byte c = 0;
    for (int j = 0; j < 8; j++)
        c |= (doc_sig[i+j]>0) << (7-j);
    fwrite(&c, sizeof(byte), 1, sig_file);
}
LeaveCriticalSection(&Section);
```

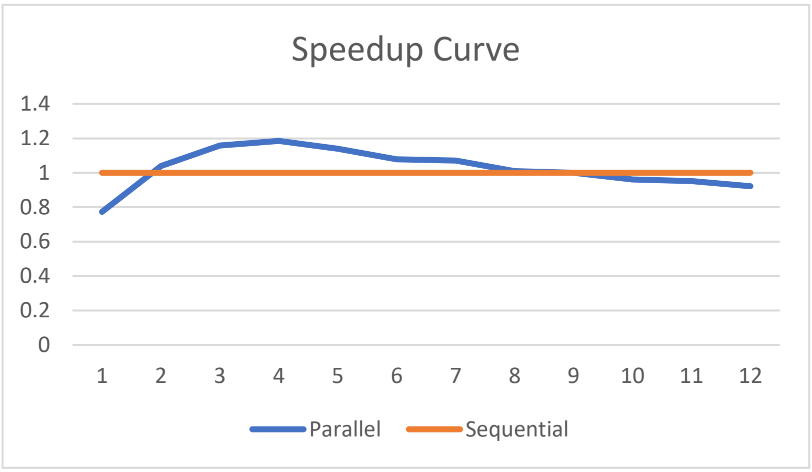


```
std::atomic<int> doc(0);
```

RESULTS

For all timing results I took the average of 10 runs in order to account for outliers and inconsistencies. All tests were run in the same computer environment using release mode x64.

Core #	Execution time (s)	Parallel	Sequential
1	6.47	0.772798	1
2	4.81	1.039501	1
3	4.32	1.157407	1
4	4.22	1.184834	1
5	4.39	1.138952	1
6	4.64	1.077586	1
7	4.67	1.070664	1
8	4.95	1.010101	1
9	5	1	1
10	5.2	0.961538	1
11	5.26	0.95057	1
12	5.43	0.92081	1

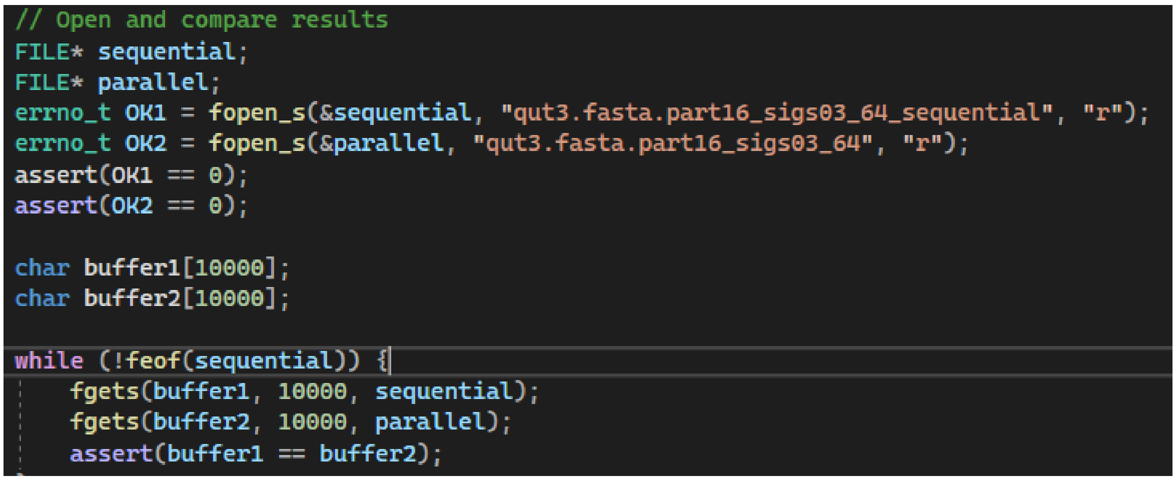


The sequential version takes 5 seconds so the speedup is 5 / sequential time.

Appendix A, Appendix B: Profiling Results before and after parallelization.

TESTING

To test the parallel version of the code produces the same results as the original we use an output file comparison. If the results are not identical an error will be raised in the program.



```
// Open and compare results
FILE* sequential;
FILE* parallel;
errno_t OK1 = fopen_s(&sequential, "qut3.fasta.part16_sigs03_64_sequential", "r");
errno_t OK2 = fopen_s(&parallel, "qut3.fasta.part16_sigs03_64", "r");
assert(OK1 == 0);
assert(OK2 == 0);

char buffer1[10000];
char buffer2[10000];

while (!feof(sequential)) {
    fgets(buffer1, 10000, sequential);
    fgets(buffer2, 10000, parallel);
    assert(buffer1 == buffer2);
```

SOFTWARE

Visual studio profiler

Win32, Critical Sections

Code restructuring

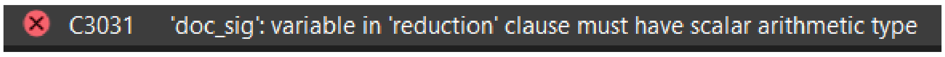
PARALLELIZATION PROBLEMS

The story of how you overcame performance problems/barriers (e.g. load imbalance, memory contention, granularity, data dependencies, etc) to improving parallel performance.

Data dependencies and code restructuring.

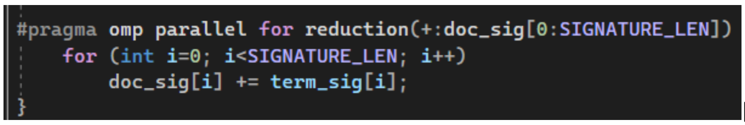
Ran into issues with open mp where it was only assigning 1 thread for the program. I forgot to enable it in the visual studio project settings.

The output file for the parallelized version was 5x smaller in bytes than the original sequential version. My original approach was flawed, my new approach is to parallelize the doc_sig for loop and that will avoid the need for changing the file writing system. See Appendix C



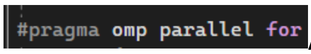
```
C3031    'doc_sig': variable in 'reduction' clause must have scalar arithmetic type
```

OpenMP has a reduction function designed for removing these race conditions.



```
#pragma omp parallel for reduction(+:doc_sig[0:SIGNATURE_LEN])
    for (int i=0; i<SIGNATURE_LEN; i++)
        doc_sig[i] += term_sig[i];
}
```

Before



```
#pragma omp parallel for
```

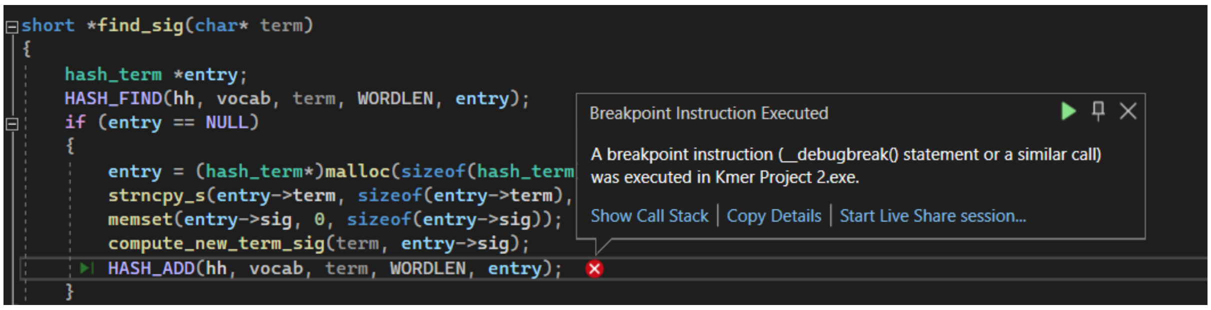
After, no reduction

Upon testing I have found this approach does not work. Since we are only adding two numbers together each time there will not be any race conditions and we don't need to use a reduction.

You cannot use a reduction on an array as it is designed for a sum variable. This would be useful in matrix multiply.

Parallelizing this loop is wrong because of the overhead. The program needs to create the overhead for each 2 lines in the FASTA file, resulting in a massive slowdown.

I avoided memory contention by using a critical section when writing to the hash map. Granularity and load imbalance where avoided by using a cyclic distribution among threads.



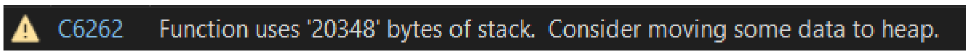
```
short *find_sig(char* term)
{
    hash_term *entry;
    HASH_FIND(hh, vocab, term, WORDLEN, entry);
    if (entry == NULL)
    {
        entry = (hash_term*)malloc(sizeof(hash_term
        strncpy_s(entry->term, sizeof(entry->term),
        memset(entry->sig, 0, sizeof(entry->sig));
        compute_new_term_sig(term, entry->sig);
        HASH_ADD(hh, vocab, term, WORDLEN, entry);
    }
```

Breakpoint Instruction Executed

A breakpoint instruction (__debugbreak() statement or a similar call) was executed in Kmer Project 2.exe.

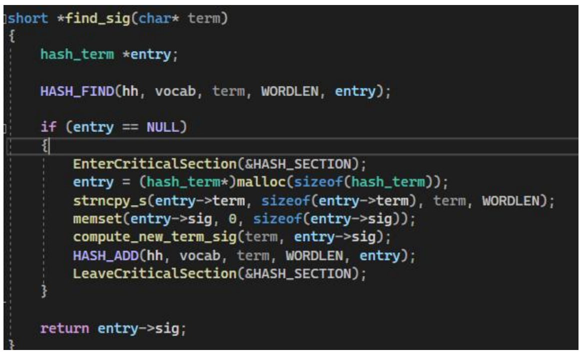
Show Call Stack | Copy Details | Start Live Share session...

Cannot do hash add in parallel, Add a critical section to add a new sig to the hash map

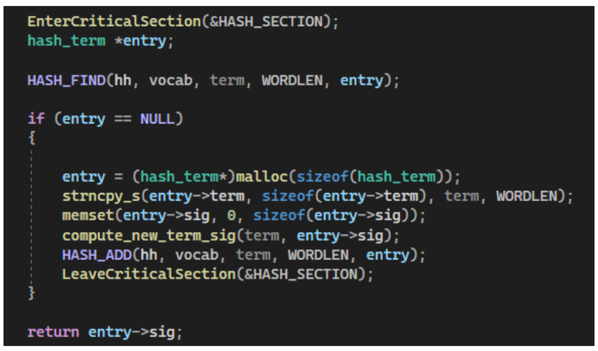


C6262   Function uses '20348' bytes of stack. Consider moving some data to heap.

Putting too much data into the vector, moving the file writing operation into the Foo function for win32 handle



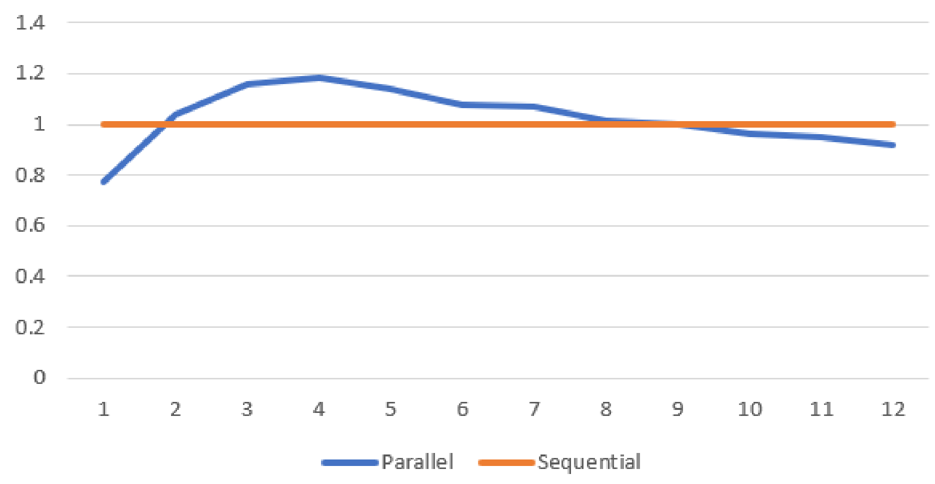
```
short *find_sig(char* term)
{
    hash_term *entry;

    HASH_FIND(hh, vocab, term, WORDLEN, entry);

    if (entry == NULL)
    {
        EnterCriticalSection(&HASH_SECTION);
        entry = (hash_term*)malloc(sizeof(hash_term));
        strncpy_s(entry->term, sizeof(entry->term), term, WORDLEN);
        memset(entry->sig, 0, sizeof(entry->sig));
        compute_new_term_sig(term, entry->sig);
        HASH_ADD(hh, vocab, term, WORDLEN, entry);
        LeaveCriticalSection(&HASH_SECTION);
    }

    return entry->sig;
}
```



```
    EnterCriticalSection(&HASH_SECTION);
    hash_term *entry;

    HASH_FIND(hh, vocab, term, WORDLEN, entry);

    if (entry == NULL)
    {

        entry = (hash_term*)malloc(sizeof(hash_term));
        strncpy_s(entry->term, sizeof(entry->term), term, WORDLEN);
        memset(entry->sig, 0, sizeof(entry->sig));
        compute_new_term_sig(term, entry->sig);
        HASH_ADD(hh, vocab, term, WORDLEN, entry);
        LeaveCriticalSection(&HASH_SECTION);
    }

    return entry->sig;
```

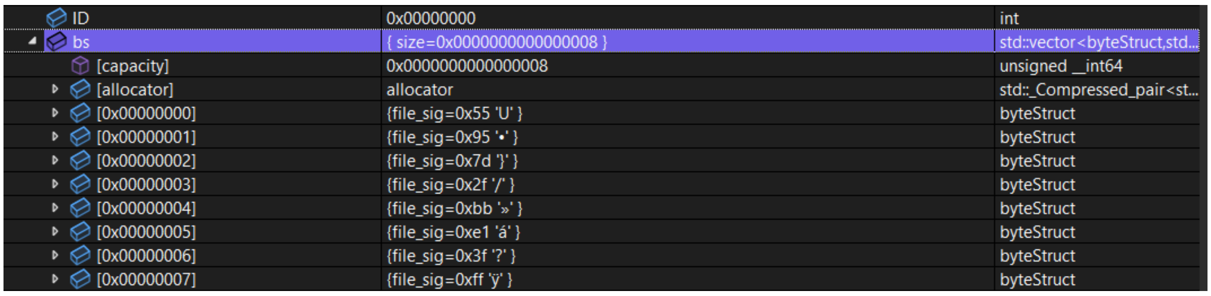
Before   After

This section of code: reading the hash map takes a lot of computation, so we cannot put the critical section before HASH_FIND.



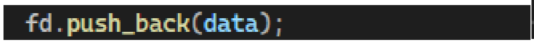
Looking at the speedup graph, the performance peaks at 4 cores and becomes a slowdown at 10 cores. This means the overhead is too large and is impacting the performance.

Upon analysing the 2 potential slowdowns where the critical sections are performed, we can find areas to improve the performance. Hash find only enters the critical section around 20 times when entering a new value. The other critical section is when writing to the file. This is also causing problems with race conditions as the file check is giving assertion errors that the results are different. In order to remove these 2 issues I changed back to a modified version of the first solution in appendix C where I write the results to another vector. Then we can write to the file sequentially and avoid this slowdown.

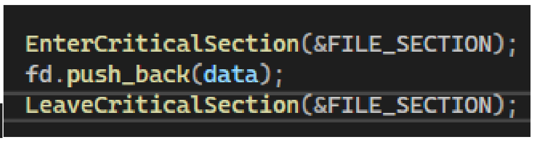


ID	0x00000000	int
bs	{ size=0x0000000000000008 }	std::vector<byteStruct,std...
[capacity]	0x0000000000000008	unsigned __int64
[allocator]	allocator	std::_Compressed_pair<st...
[0x00000000]	{file_sig=0x55 'U' }	byteStruct
[0x00000001]	{file_sig=0x95 '•' }	byteStruct
[0x00000002]	{file_sig=0x7d '}' }	byteStruct
[0x00000003]	{file_sig=0x2f '/' }	byteStruct
[0x00000004]	{file_sig=0xbb '»' }	byteStruct
[0x00000005]	{file_sig=0xe1 'á' }	byteStruct
[0x00000006]	{file_sig=0x3f '?' }	byteStruct
[0x00000007]	{file_sig=0xff 'ÿ' }	byteStruct

I was getting memory errors when writing to the file data struct which was fixed by placing a critical section on it.



```
fd.push_back(data);
```



```
EnterCriticalSection(&FILE_SECTION);
fd.push_back(data);
LeaveCriticalSection(&FILE_SECTION);
```

This causes a too big of a slowdown, the solution is to make a data structure for each thread that way two threads are not writing to the same variable and causing issues.

Currently the program runs a bit slower than the sequential due to excessive writing to vectors. This was done to avoid race conditions and memory contention issues.

CODE EXPLANATION

An explanation of the code that you added or modified to parallelize the application (including source code line count).

Line by line code explanation

```
#include <windows.h>
#include <atomic>
#include <vector>
#include <assert.h>
#include <map>
#include <array>
#include <algorithm>
```

Include statements

```
struct threadData {
    char buffer[10000];
    int n;
    int ID;
};
```

Thread data struct which holds 1 protein sequence from the fasta file. Buffer is the data, n in the length of the data and ID is the line number; equivalent to the doc variable.

```
struct fileData {
    int ID;
    int doc_sig[SIGNATURE_LEN];
};
```

File data contains the output data which is written to the output file at the end of the program.

```
const int numThreads = 6;
CRITICAL_SECTION HASH_SECTION;
CRITICAL_SECTION FILE_SECTION;

std::vector<threadData> td;
std::vector<fileData> fd[numThreads];


bool CompareByID(const fileData& a, const fileData& b) {
    return a.ID < b.ID;
}
```

Declare number of threads, declare critical section names, declare vector names and vector array, comparison function which sorts file data by ID

```
EnterCriticalSection(&HASH_SECTION); LeaveCriticalSection(&HASH_SECTION);
```

Critical section code for the hash map

```
std::atomic<int> doc = 0;
```

Change doc to use an atomic integer type

```
void compute_signature(char* sequence, int length, int ID, int threadId)
{
    int doc_sig [SIGNATURE_LEN];

    for (int i = 0; i < length - WORDLEN + 1; i++) {
            short* term_sig = find_sig(sequence+i);

            for (int i = 0; i < SIGNATURE_LEN; i++) {
                doc_sig[i] += term_sig[i];
            }
    }
    fileData data;
    data.ID = ID;
    memcpy(data.doc_sig, doc_sig, sizeof(doc_sig));
    fd[threadId].push_back(data);
}
```

Write the output data to the specific vector array. Use a memset to copy integer array type.

```
DWORD WINAPI Foo(LPVOID lpThreadParameter) {

    int threadId = (int)lpThreadParameter;

    // cyclic distribution
    for (int i = threadId; i < td.size(); i += numThreads) {

        char* buffer = td[i].buffer;
        int n = td[i].n;
        int ID = td[i].ID;

        partition(buffer, n, ID, threadId);
    }

    return 0;
}
```

Win32 Explicit thread code, send each line of thread data vector into partition and parse the threadId

```
InitializeCriticalSection(&HASH_SECTION);

int ID = 0; // initialize ID

char buffer[10000]; // initialize buffer
while (!feof(file))
{
    fgets(buffer, 10000, file); // skip meta data line
    fgets(buffer, 10000, file);
    int n = (int)strlen(buffer) - 1;
    buffer[n] = 0;
// put file data into vector and increment ID
    threadData data;
    strcpy_s(data.buffer, buffer);
    data.n = n;
    data.ID = ID;
    td.push_back(data);
    ID++;
}
// create handles
HANDLE* handle = new HANDLE[numThreads];

// create threads
for (int i = 0; i < numThreads; i++) {
    handle[i] = CreateThread(NULL, 0, Foo, (LPVOID)i, 0, NULL);
}

// wait for threads to finish
WaitForMultipleObjects(numThreads, handle, TRUE, INFINITE);

std::vector<fileData> res;
// concatenate results
for (int i = 0; i < numThreads; i++) {
    for (auto& j : fd[i]) {
        res.push_back(j);
    }
}

// Sort results to eliminate any race conditions
std::sort(res.begin(), res.end(), CompareByID);

// loop through sig_results and write to file
int count = 0;

for (auto& i : res) {

    fwrite(&i.ID, sizeof(int), 1, sig_file);
    auto doc_sig = i.doc_sig;

    for (int i = 0; i < SIGNATURE_LEN; i += 8)
    {
        byte c = 0;
        for (int j = 0; j < 8; j++)
            c |= (doc_sig[i + j] > 0) << (7 - j);
        fwrite(&c, sizeof(byte), 1, sig_file);
    }

    count++;
}
printf("Printed %d Lines", count);


fclose(file);

fclose(sig_file);



DeleteCriticalSection(&HASH_SECTION);

// Open and compare results
/*
FILE* sequential;
```

```
FILE* parallel;
errno_t OK1 = fopen_s(&sequential, "qut3.fasta.part16_sigs03_64_sequential",
"r");
errno_t OK2 = fopen_s(&parallel, "qut3.fasta.part16_sigs03_64", "r");
assert(OK1 == 0);
assert(OK2 == 0);

char buffer1[10000];
char buffer2[10000];

while (!feof(sequential)) {
    fgets(buffer1, 10000, sequential);
    fgets(buffer2, 10000, parallel);
    assert(buffer1 == buffer2);
}
printf("Files are the same!\n");
*/
```

REFLECTIONS

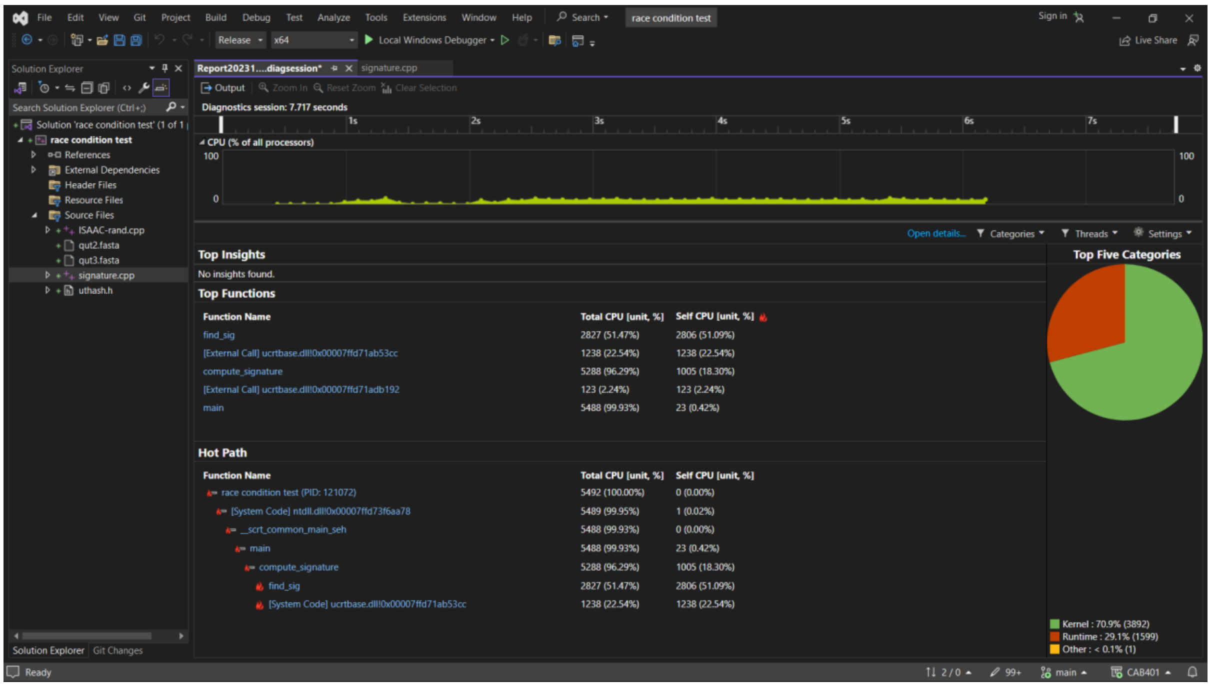
My attempt was unfortunately half successful as I ran into a bit of a roadblock / catch 22. To split the file among the threads I read it out into a vector and send the work to them. When concatenating the results however, I run into memory contention errors because multiple threads are trying to write to the output variable. You can’t use files for this reason also. So, I allocated 1 output variable per thread and then join the results vector at the end of the program. Unfortunately, this slows down the program significantly, and it won’t run faster with larger data sets, only slower. The only solution I can think of is to use a data type like atomic or like have that shared output that works in parallel, but I wasn’t able to find it.

Learnt- have to gain an in depth understanding of the original program, I ran into semantical issues where I assumed wrongly what the original code was doing. Also using the debugger to check variable values helped with this.

Your source code (both before and after versions) together with instructions for compiling, running, hardware requirements and realistic input data sets.

APPENDICES

Appendix A: Sequential Profiling results



Diagnostics session: 7.717 seconds

Top Insights

No insights found.

Top Functions

Function Name	Total CPU [unit, %]	Self CPU [unit, %]
find_sig	2827 (51.47%)	2806 (51.09%)
[External Call] ucrtbase.dll!0x00007ffd71ab53cc	1238 (22.54%)	1238 (22.54%)
compute_signature	5288 (96.29%)	1005 (18.30%)
[External Call] ucrtbase.dll!0x00007ffd71adb192	123 (2.24%)	123 (2.24%)
main	5488 (99.93%)	23 (0.42%)

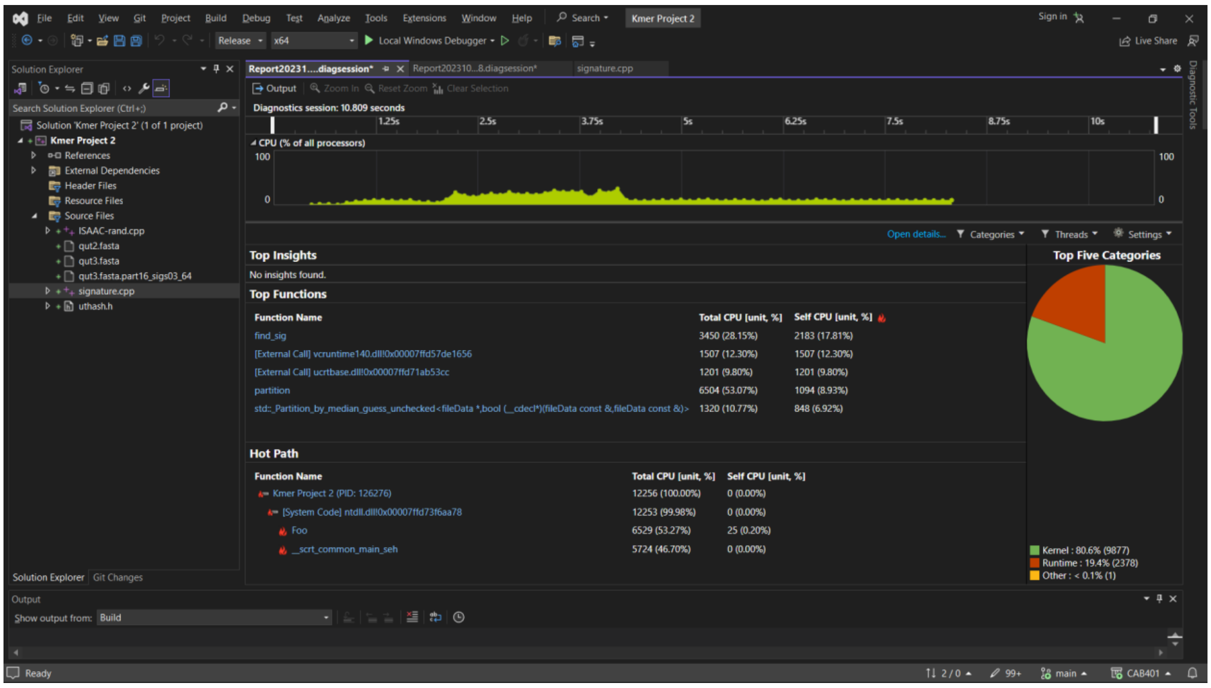
Hot Path

Function Name	Total CPU [unit, %]	Self CPU [unit, %]
race condition test (PID: 121072)	5492 (100.00%)	0 (0.00%)
[System Code] ntdll.dll!0x00007ffd73f6aa78	5489 (99.95%)	1 (0.02%)
__scrt_common_main_seh	5488 (99.93%)	0 (0.00%)
main	5488 (99.93%)	23 (0.42%)
compute_signature	5288 (96.29%)	1005 (18.30%)
find_sig	2827 (51.47%)	2806 (51.09%)
[System Code] ucrtbase.dll!0x00007ffd71ab53cc	1238 (22.54%)	1238 (22.54%)

Top Five Categories

🟩 Kernel : 70.9% (3892)
🟥 Runtime : 29.1% (1599)
🟨 Other : < 0.1% (1)

Appendix B: Parallel Version Profiling Results



Diagnostics session: 10.809 seconds

Top Insights

No insights found.

Top Functions

Function Name	Total CPU [unit, %]	Self CPU [unit, %]
find_sig	3450 (28.15%)	2183 (17.81%)
[External Call] vcruntime140.dll!0x00007ffd57de1656	1507 (12.30%)	1507 (12.30%)
[External Call] ucrtbase.dll!0x00007ffd71ab53cc	1201 (9.80%)	1201 (9.80%)
partition	6504 (53.07%)	1094 (8.93%)
std::_Partition_by_median_guess_unchecked<fileData *,bool (__cdecl*)(fileData const &,fileData const &)>	1320 (10.77%)	848 (6.92%)

Hot Path

Function Name	Total CPU [unit, %]	Self CPU [unit, %]
Kmer Project 2 (PID: 126276)	12256 (100.00%)	0 (0.00%)
[System Code] ntdll.dll!0x00007ffd73f6aa78	12253 (99.98%)	0 (0.00%)
Foo	6529 (53.27%)	25 (0.20%)
__scrt_common_main_seh	5724 (46.70%)	0 (0.00%)

🟩 Kernel : 80.6% (9877)
🟥 Runtime : 19.4% (2378)
🟨 Other : < 0.1% (1)

Appendix C: original attempt

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "uthash.h"
#include <chrono>
#include <vector>
#include <map>
#include <array>
#include <omp.h>
#include <assert.h>


typedef unsigned char byte;

#define SIGNATURE_LEN 64

int DENSITY  = 21;
int PARTITION_SIZE;

int inverse[256];
const char* alphabet = "CSTPAGNDEQHRKMILVFYW";


void seed_random(char* term, int length);
short random_num(short max);
void Init();


int WORDLEN;

FILE *sig_file; // output sig p file

std::map<int, std::array<byte, SIGNATURE_LEN>> sig_results; // map for sorted
results
```

```c
typedef struct {
    char* buffer;
    int n;
    int ID;
} file_data;


typedef struct
{
    char term[100];
    short sig[SIGNATURE_LEN];
    UT_hash_handle hh;

} hash_term;

hash_term *vocab = NULL;


short* compute_new_term_sig(char* term, short *term_sig)
{
    seed_random(term, WORDLEN);

    int non_zero = SIGNATURE_LEN * DENSITY/100;

    int positive = 0;
    while (positive < non_zero/2)
    {
        short pos = random_num(SIGNATURE_LEN);
        if (term_sig[pos] == 0)
       {
            term_sig[pos] = 1;
            positive++;
        }
    }

    int negative = 0;
    while (negative < non_zero/2)
    {
        short pos = random_num(SIGNATURE_LEN);
        if (term_sig[pos] == 0)
       {
            term_sig[pos] = -1;
            negative++;
        }
    }
    return term_sig;
}


short *find_sig(char* term)
{
    hash_term *entry;
    HASH_FIND(hh, vocab, term, WORDLEN, entry);
    if (entry == NULL)
    {
        entry = (hash_term*)malloc(sizeof(hash_term));
        strncpy_s(entry->term, sizeof(entry->term), term, WORDLEN);
        memset(entry->sig, 0, sizeof(entry->sig));

        compute_new_term_sig(term, entry->sig); // MUTEX
        HASH_ADD(hh, vocab, term, WORDLEN, entry);
    }

    return entry->sig;
}


void signature_add(char* term, int* doc_sig)
{
      short* term_sig = find_sig(term);

      for (int i=0; i<SIGNATURE_LEN; i++)
            doc_sig[i] += term_sig[i];
}

// int doc = 0; removed

void compute_signature(char* sequence, int length, int ID)
{
    int doc_sig[SIGNATURE_LEN];

    memset(doc_sig, 0, sizeof(doc_sig));

    for (int i=0; i<length-WORDLEN+1; i++)
        signature_add(sequence+i, doc_sig);


    std::array<byte, SIGNATURE_LEN> d;

    // flatten and output to sig_results
    for (int i = 0; i < SIGNATURE_LEN; i += 8)
    {
        byte c = 0;
        for (int j = 0; j < 8; j++)
            c |= (doc_sig[i + j] > 0) << (7 - j);
        d[i] = c;
    }
    sig_results[ID] = d;
}

#define min(a,b) ((a) < (b) ? (a) : (b))

void partition(char* sequence, int length, int ID)
{
    int i=0;
    do
    {
        compute_signature(sequence+i, min(PARTITION_SIZE, length-i), ID);
        i += PARTITION_SIZE/2;
    }
    while (i+PARTITION_SIZE/2 < length);
    // doc++; // global variable
}

int power(int n, int e)
{
    int p = 1;
    for (int j=0; j<e; j++)
        p *= n;
    return p;
}
```

```
int main(int argc, char* argv[])
{
    //const char* filename = "qut2.fasta";
    const char* filename = "qut3.fasta";

    WORDLEN = 3;
    PARTITION_SIZE = 16;
    int WORDS = power(20, WORDLEN);

    for (int i=0; i<strlen(alphabet); i++)
        inverse[alphabet[i]] = i;

    auto start = std::chrono::high_resolution_clock::now();

    FILE* file;
    errno_t OK = fopen_s(&file, filename, "r");

    if (OK != 0)
    {
        fprintf(stderr, "Error: failed to open file %s\n", filename);
        return 1;
    }

    char outfile[256];
    sprintf_s(outfile, 256, "%s.part%d_sigs%02d_%d_parallel", filename,
PARTITION_SIZE, WORDLEN, SIGNATURE_LEN);
    fopen_s(&sig_file, outfile, "w");

    char buffer[10000];

    int length = 200000; //400000 lines / 2
    int ID = 0;

    std::vector<file_data> fd; // vector for file data

    while (!feof(file))
    {
        // buffer = genome data from each line
        // n = data length
        fgets(buffer, 10000, file); // skip meta data line
        fgets(buffer, 10000, file);
        int n = (int)strlen(buffer) - 1;
        buffer[n] = 0;

        file_data line;
        line.buffer = buffer;
        line.n = n;
        line.ID = ID;

        fd.push_back(line);

        ID++; // increment ID
    }


#pragma omp parallel for num_threads(6)
    for (int i = 0; i < fd.size(); ++i) {
        partition(fd[i].buffer, fd[i].n, fd[i].ID);
    }


    // loop through sig_results and write to file
    for (const auto& entry : sig_results) {
        int ID = entry.first;
        auto sig = entry.second;
        fwrite(&ID, sizeof(int), 1, sig_file);
        fwrite(sig.data(), sizeof(byte), SIGNATURE_LEN, sig_file);
    }


    fclose(file);

    fclose(sig_file);

    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;

    printf("%s %f seconds\n", filename, duration.count());

    // Open and compare results
    /*FILE* sequential;
    FILE* parallel;
    errno_t OK = fopen_s(&sequential, "", "r");
    errno_t OKK = fopen_s(&parallel, "", "r");
    assert(OK == 0);
    assert(OKK == 0);

    char buffer1[10000];
    char buffer2[10000];

    while (!feof(sequential)) {
        fgets(buffer1, 10000, sequential);
        fgets(buffer2, 10000, parallel);
        assert(buffer1 == buffer2);
    }
    */



    return 0;
}
```