Queensland University of Technology

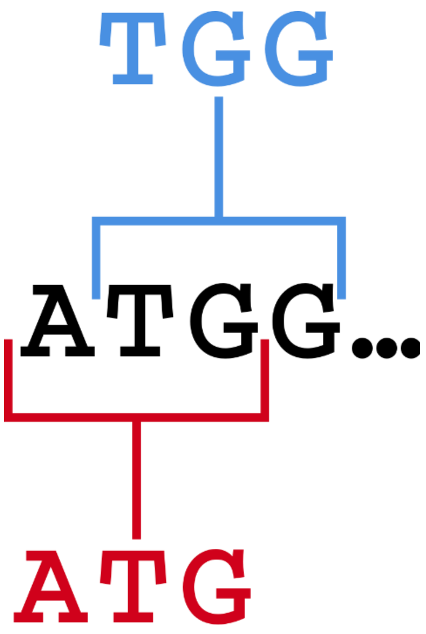
CAB401: High Performance and Parallel Computing

Parallelization Project Submission

Due: 11:59 pm Monday 1st November (Week 14)

Name: Jol Singh

ID Number: n10755756



1 Introduction

The software application I've chosen to parallelise is bioinformatics, k-mer signatures, associated with computational genomics and sequence analysis, this application is written in C++, and operates by reading a **.fasta** file (binary format) containing sequences, then computing a k-mer signature for each sequence, and saving these signatures into a binary file format. In bioinformatics, k-mers are substrings of length ***k*** contained within a genomic sequence, which represents certain signatures to describe different genomes or regions in a sequence. Moreover, the frequency set of k-mers in a class of genomic sequences can be classified as a "signature" of the underlying sequence, and when comparing frequencies, it's computationally easier than sequence alignment, which's defined as a way of arranging genomic sequences to identify similarities between sequences. Furthermore, it's also an important method in alignment-free sequence analysis which approaches molecular sequence, and structure data providing an alternative over alignment-based approaches.

Moving on, I've constructed a class diagram shown below in figure 1.1 to provide a visual representation of the original sequential applications' structure, and the functions shown in the diagram can be found within the signature.cpp file as it's the main program. The application begins at the main function where it loads a **.fasta** file, then writes signatures into an output file created by the function, and from figure 1.2, we can see that the while loop reads lines from the **.fasta** file to compute and write signatures for each sequence. Moreover, the partition function (fig 2.1) utilises **compute_signature** in computing signatures for each partition of a sequence, and **compute_signature** (fig 2.3) uses an array called **doc_sig** to store signatures for a specific partition. Additionally, signatures are stored within the **doc_sig** array by using **signature_add**, where **signature_add** uses a function called, **find_sig** to retrieve signatures, and iterates over each signature in order to be added to **doc_sig**. Furthermore, **find_sig** utilises another function called **compute_new_term_sig** to create new signatures, and the computed signatures from this function are then added to a **hashmap**, this **hashmap** is a critical component for the program as it ensures that if similar names or values were to appear, signatures wouldn't need to be recomputed, but can instead be retrieved from this **hashmap**.

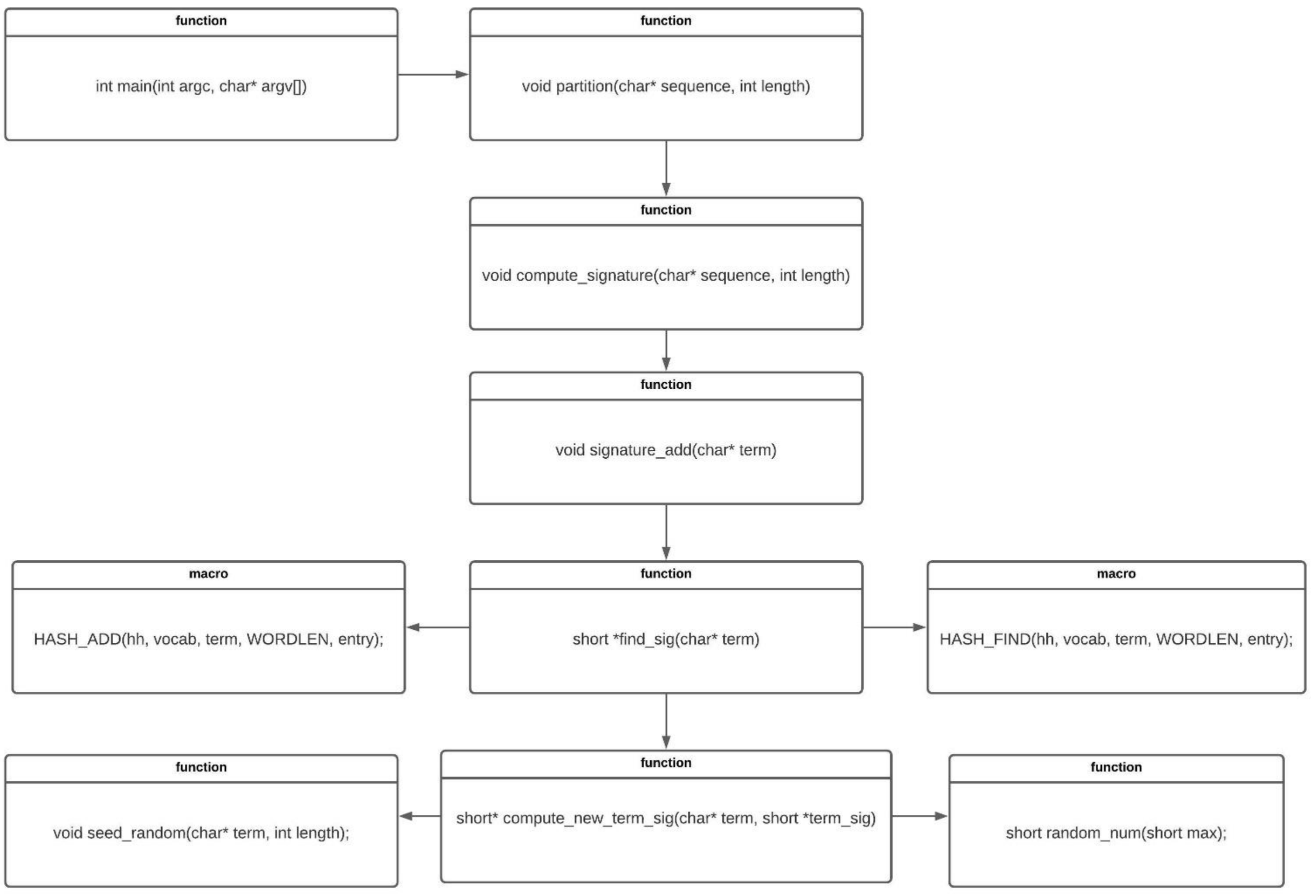
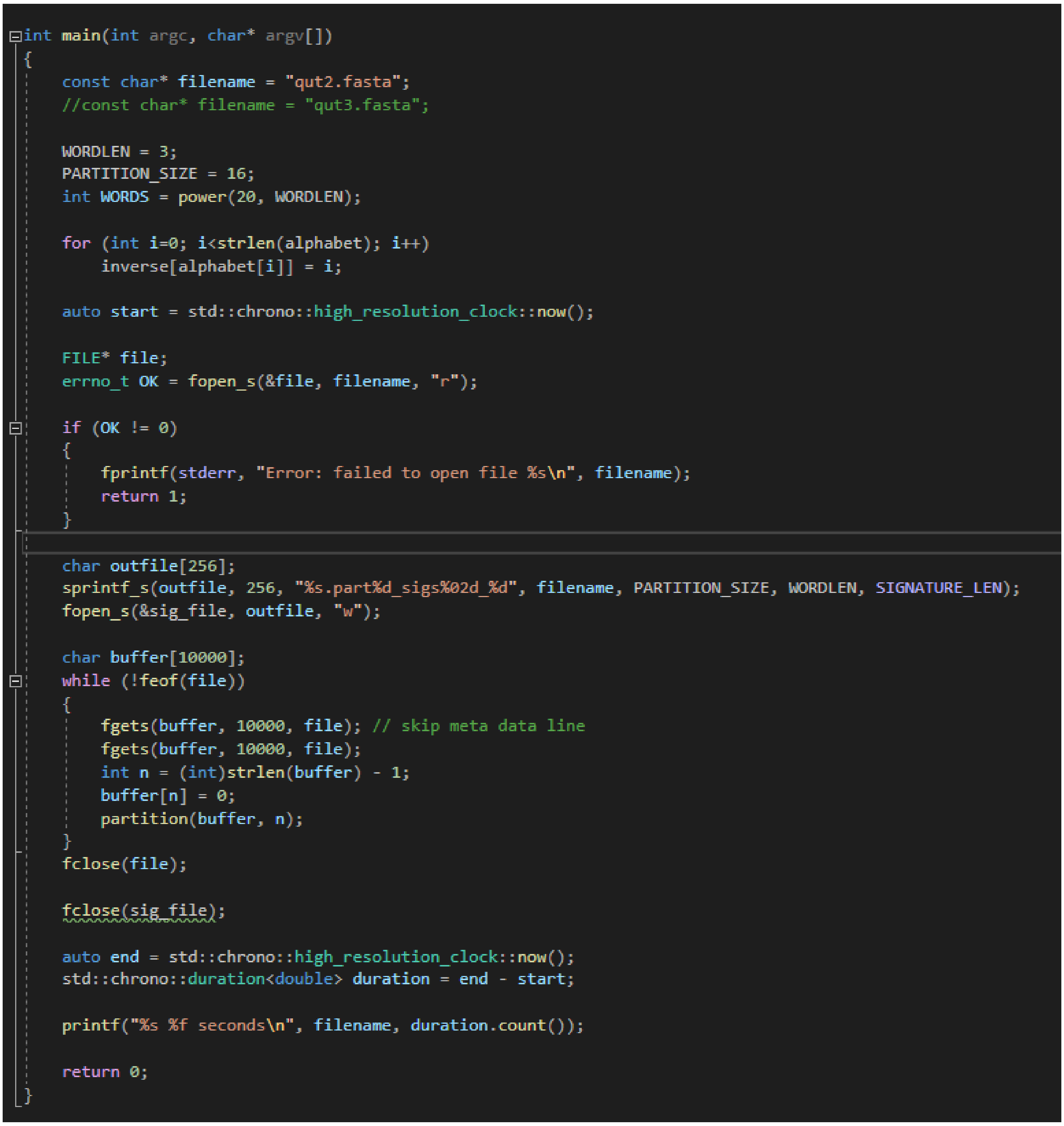

function
int main(int argc, char* argv[])
function
void partition(char* sequence, int length)
function
void compute_signature(char* sequence, int length)
function
void signature_add(char* term)
macro
HASH_ADD(hh, vocab, term, WORDLEN, entry);
function
short *find_sig(char* term)
macro
HASH_FIND(hh, vocab, term, WORDLEN, entry);
function
void seed_random(char* term, int length);
function
short* compute_new_term_sig(char* term, short *term_sig)
function
short random_num(short max);

Figure 1.1 Class Diagram of Original Sequential Application Structure

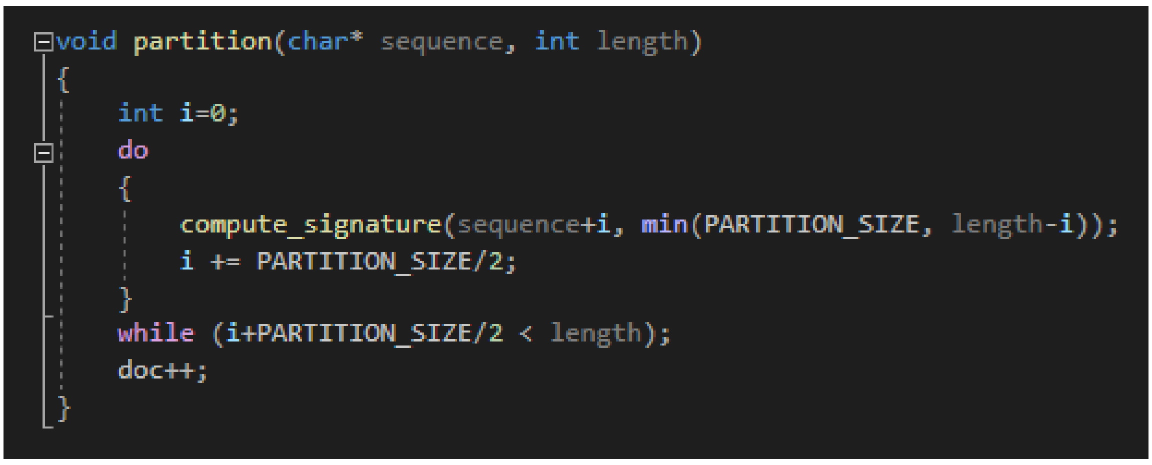


```
int main(int argc, char* argv[])
{
    const char* filename = "qut2.fasta";
    //const char* filename = "qut3.fasta";

    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", filename, PARTITION_SIZE, WORDLEN, SIGNATURE_LEN);
    fopen_s(&sig_file, outfile, "w");

    char buffer[10000];
    while (!feof(file))
    {
        fgets(buffer, 10000, file); // skip meta data line
        fgets(buffer, 10000, file);
        int n = (int)strlen(buffer) - 1;
        buffer[n] = 0;
        partition(buffer, n);
    }
    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());

    return 0;
}
```

Figure 1.2 Main Function in Signature.cpp

2 Analysis

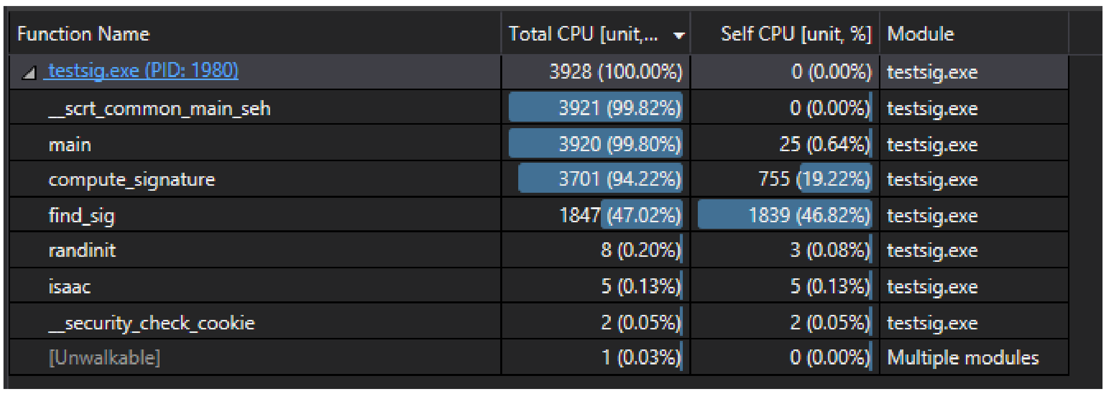
Bioinformatics, k-mer signatures, offers many opportunities for exploitations of parallelism. Signatures for each partition of a sequence can be computed independently, and every sequence located within the **.fasta** files has the ability to function independently as there are no data dependencies between them. Furthermore, this means the viability for only a true dependence is that signatures and document numbers are required to be orderly written into a **.fasta** file identical to that of the original sequential application, but this now means that scalable parallelism is needed as writing signatures to **.fasta** files sequentially is practically unavoidable. Moreover, because of the evident obstacles, as well as, CPU usage results obtained from running the profiling tool provided by Visual Studio 2019 (fig 2.2) allowed me to identify a couple of safe areas for potentially an exploitable parallelism, which is the main loop shown in figure 1.2, **compute_signature** loop shown in figure 2.3, and **partition** loop in figure 2.1.



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

Figure 2.1: Partition Function in Signature.cpp

"Takes extracted sequences from main loop and passes partitions to compute_signature"

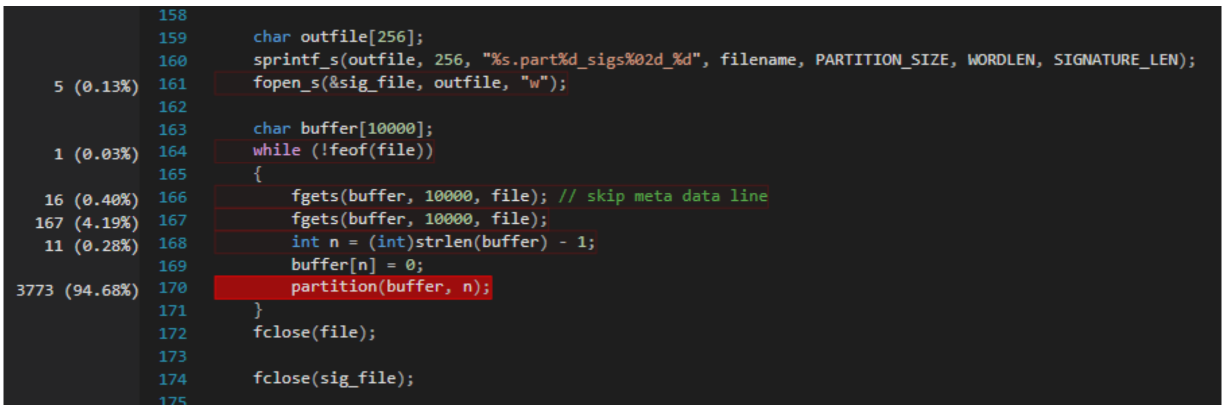


Function Name	Total CPU [unit,...	Self CPU [unit, %]	Module
testsig.exe (PID: 1980)	3928 (100.00%)	0 (0.00%)	testsig.exe
__scrt_common_main_seh	3921 (99.82%)	0 (0.00%)	testsig.exe
main	3920 (99.80%)	25 (0.64%)	testsig.exe
compute_signature	3701 (94.22%)	755 (19.22%)	testsig.exe
find_sig	1847 (47.02%)	1839 (46.82%)	testsig.exe
randinit	8 (0.20%)	3 (0.08%)	testsig.exe
isaac	5 (0.13%)	5 (0.13%)	testsig.exe
__security_check_cookie	2 (0.05%)	2 (0.05%)	testsig.exe
[Unwalkable]	1 (0.03%)	0 (0.00%)	Multiple modules

Figure 2.2: CPU Usage of main, compute_signature, and find_sig in Sequential

Moving on, the profiler has highlighted resource heavy functions in certain loops such as the partition function within the main loop, **partition(buffer,n)** (fig 2.3), which's causing the primary bottleneck. This function references the partition loop in figure 2.1 where it computes the signatures for each partition of a sequence from a loaded **.fasta** file, and then uses **buffer** as a temporary storage for the data at a length of **n.** I believe pursuing parallelism for this loop would be beneficial in boosting the applications performance as we can use methods that allow signatures for each partition of a sequence can be computed much faster and efficiently.

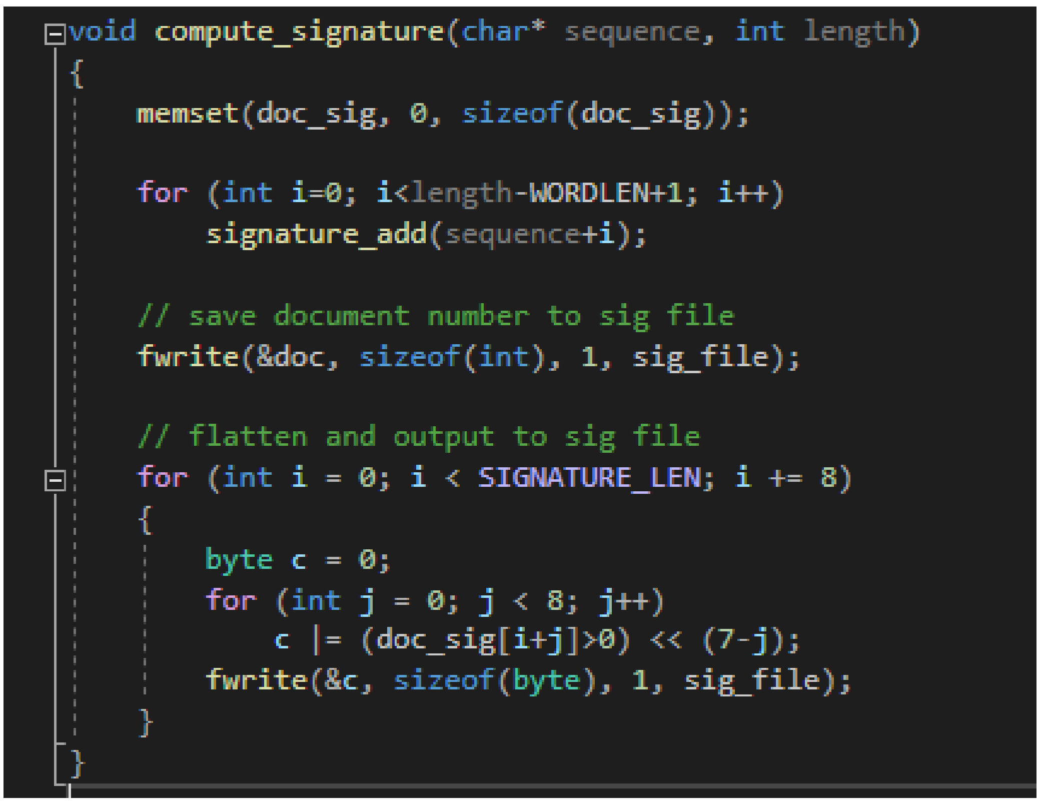
Moreover, if we were to parallelise the **compute_signature** loop shown in figure 2.4, it'll expose an area of parallelism, allowing us to avoid trivial matters in terms of when the loop writes to **.fasta** files, and as it's also one of the simplest areas that requires' the least amount of code restructuring transformation, and would possibly increase our timing as it's the second largest consumer of CPU during execution. In addition, we'll have to also modify and reconstruct the **doc_sig** array, as well as, parallelising the for-loop. Furthermore, the **find_sig** loop is another performance bottleneck, and parallelism of this loop would mean signatures can be retrieved much more efficiently. Meanwhile, the identification of these possible exploitable parallelism areas are sufficient granularity, as we look further down the call-tree it becomes too fine-grained, and not worth the time or effort to parallelise.



```
                    char outfile[256];
                    sprintf_s(outfile, 256, "%s.part%d_sigs%02d_%d", filename, PARTITION_SIZE, WORDLEN, SIGNATURE_LEN);
    5 (0.13%)       fopen_s(&sig_file, outfile, "w");

                    char buffer[10000];
    1 (0.03%)       while (!feof(file))
                    {
   16 (0.40%)           fgets(buffer, 10000, file); // skip meta data line
  167 (4.19%)           fgets(buffer, 10000, file);
   11 (0.28%)           int n = (int)strlen(buffer) - 1;
                        buffer[n] = 0;
 3773 (94.68%)          partition(buffer, n);
                    }
                    fclose(file);

                    fclose(sig_file);
```

Figure 2.3: Partition function resource heavy within while loop of main method.

"Iterates through each line within the .fasta file and passes extracted sequences to partition"



```
void compute_signature(char* sequence, int length)
{
    memset(doc_sig, 0, sizeof(doc_sig));

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

    // 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);
    }
}
```

Figure 2.4: compute_signature function in Signature.cpp.

3 Speed-up Method

Before attempting parallelisation, I conducted extensive research looking into external software's, tools and technologies that could assist me in performing parallelisation. I ended up finding Parallel Patterns Library (PPL) created by Microsoft for Visual Studio. The library provides programming models that resemble the C++ Standard Library, it enables imperative data parallelism by providing parallel algorithms such as **parallel_for** that can distribute computations across collections or a set of data, as well as, enabling task parallelism that allows task objects to be distributed amongst multiple independent operations of computing resources. Moreover, PPL builds on scheduling and resource management of Concurrency Runtime (ConCRT), raising the level of abstraction between an applications code and underlying threading mechanism by providing collectively secure algorithms and containers that act on data in parallel, as well as, providing alternatives to shared state.

Moreover, PPL contains a number of abstractions to perform multi-core programming, as mentioned above it includes features such as Task Parallelism, Parallel Algorithms, Parallel Containers and Objects. Task Parallelism is where a process or task is able to operate on top of Windows ThreadPool while executing several tasks in parallel by using the **concurrency::task** class, and Parallel Algorithms are generic algorithms such as **parallel_for** where operation can be carried out on top of Concurrency Runtime while acting on collections of data in parallel. Whereas, Parallel Containers and Objects are generic container types such as **concurrent_vector** provide safe concurrent access to their elements.

Furthermore, as quoted earlier, "*PPL builds on scheduling and resource management of Concurrency Runtime (ConCRT)...*", PPL is an extension of Concurrency Runtime (ConCRT), where Concurrency Runtime is by default a feature of Visual Studio, it's a feature that helps in writing parallel applications by raising the level of abstraction in a way that certain details related to concurrency don't need to be managed, and can also be utilised in specifying scheduling policies of an application. The Concurrency Runtime architecture consists of four components (fig 3.1): PPL, Asynchronous Agents Library, Synchronisation Data Structures, and Task Schedular; I utilised some of the functions provided by these components to achieve the mapping of data to processors, and performing synchronisation to achieve a noticeable speed-up.

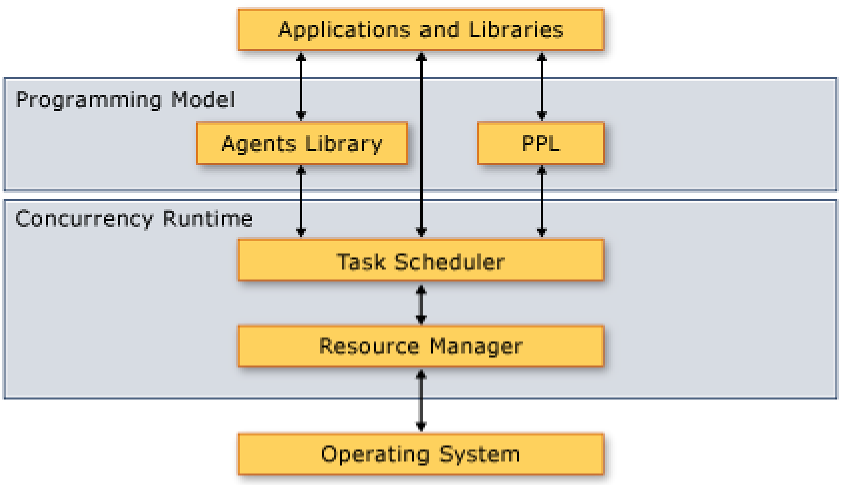
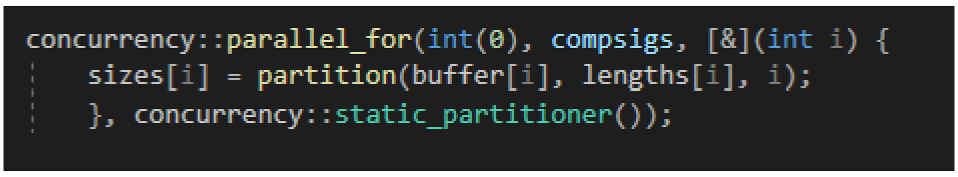


Figure 3.1: Concurrency Runtime Architecture

3 Continued

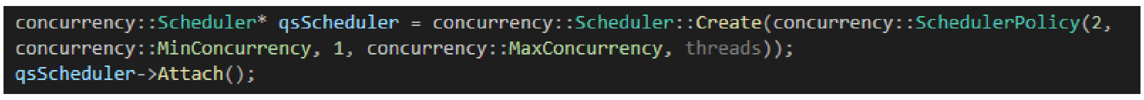
I approached parallelism by utilising the **concurrency::parallel_for** function (fig 3.2) that uses the **concurrency::structured_task_group** function, operating within the **parallel_for** function without being called, and allowing the parallel loop iterations to be performed. In addition, the **concurrency::parallel_for** function repeatedly performs the same task in parallel, and each tasks are parameterised by an iteration value. Furthermore, the **parallel_for** algorithm partitions tasks in the most efficient and optimal way for parallel execution, this function uses various techniques to balance partitions when workloads become unbalanced, and if one loop iteration were to be blocked cooperatively, then the runtime, redistributes tasks from one thread to another; additionally, this function also supports nested parallelism where one parallel loop can contain another parallel loop, and the runtime can efficiently coordinate data processing between the two loops.



```
concurrency::parallel_for(int(0), compsigs, [&](int i) {
    sizes[i] = partition(buffer[i], lengths[i], i);
    }, concurrency::static_partitioner());
```

Figure 3.2: concurrency::parallel_for

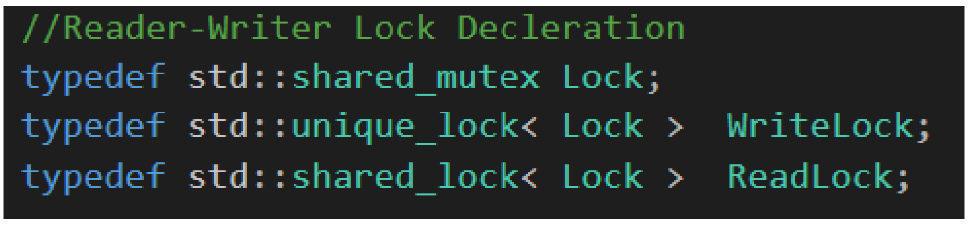
Moreover, in order to produce a noticeable speed-up, I was able to map a select amount of processors by using the **concurrency::scheduler** function from scheduler instances of Task Schedular. The runtime creates a default schedular once I've used a runtime functionality such as **parallel_for**, and this default scheduler that's created becomes the current schedular where it's used in initiating a given parallel task such as the one shown in figure 3.4. Furthermore, these schedulers are extremely resourceful in allowing explicit scheduling polices to be assigned for specific tasks, a scheduler instance can be created to run some tasks at an elevated thread priority, while the default scheduler can be used to run the other set of tasks at normal thread priority. In addition, schedular instances don't require specific configurations and with the create method **concurrency::Scheduler::Create**, it can be set to a maximum amount of concurrency (fig 3.4).



```
concurrency::Scheduler* qsScheduler = concurrency::Scheduler::Create(concurrency::SchedulerPolicy(2,
concurrency::MinConcurrency, 1, concurrency::MaxConcurrency, threads));
qsScheduler->Attach();
```

Figure 3.3 concurrency::Scheduler & concurrency::Scheduler::Create

Moving on, I was able to perform synchronisation with the **reader_writer_lock** class (fig 3.4) belonging to Synchronisation Data Structures. This class follows quiet similarly to mutex locks, in a way that it provides thread-safe read/write operations where reader/writer locks can be used when multiple threads require concurrent read access to shared data. Furthermore, the **reader_writer_lock** operates in way that if a thread wants to write to shared data, it requires this class, but if other threads want access they'll be blocked until the writer releases the lock; basically, a writer queue-based operation that grants first in – first out (FIFO) access to writers while readers are blocked under a continuous load of writers.



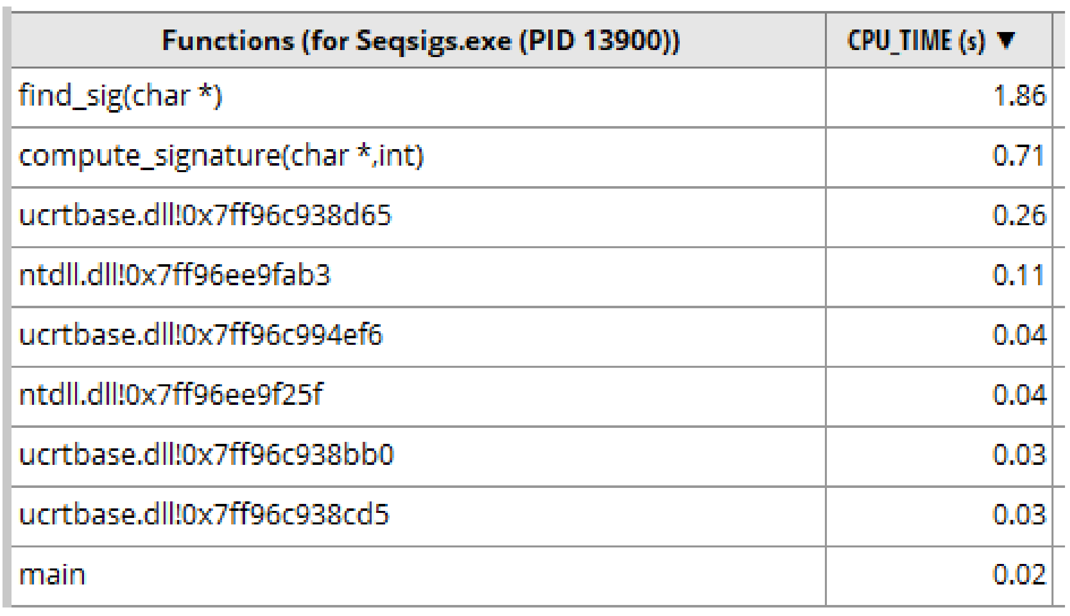
```
//Reader-Writer Lock Decleration
typedef std::shared_mutex Lock;
typedef std::unique_lock< Lock >  WriteLock;
typedef std::shared_lock< Lock >  ReadLock;
```

Figure 3.4: Using reader_writer_lock function to perform synchronisation
"It's quite similar to mutex locks showed in week 5"

4 Timing & Profiling Results

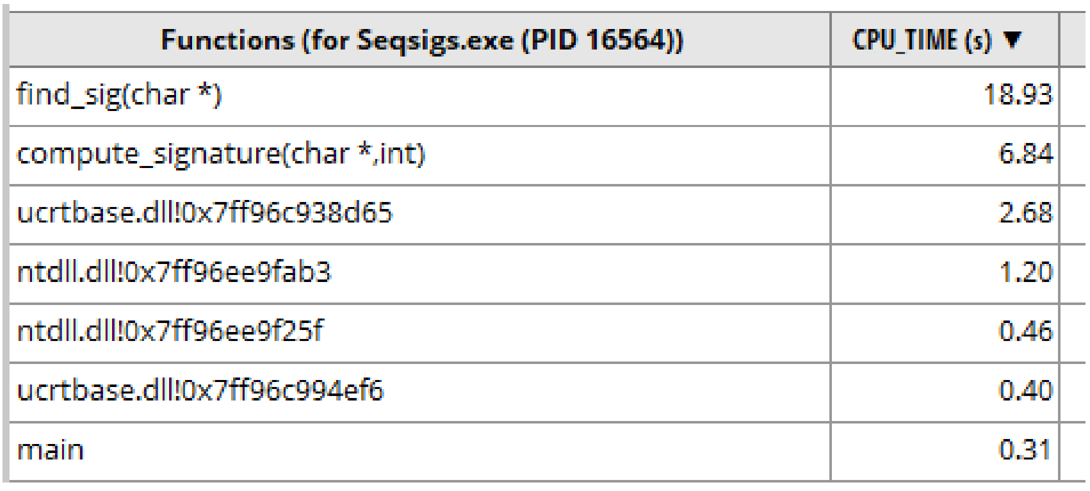
Sequential

qut3.fasta



Functions (for Seqsigs.exe (PID 13900))	CPU_TIME (s) ▼
find_sig(char *)	1.86
compute_signature(char *,int)	0.71
ucrtbase.dll!0x7ff96c938d65	0.26
ntdll.dll!0x7ff96ee9fab3	0.11
ucrtbase.dll!0x7ff96c994ef6	0.04
ntdll.dll!0x7ff96ee9f25f	0.04
ucrtbase.dll!0x7ff96c938bb0	0.03
ucrtbase.dll!0x7ff96c938cd5	0.03
main	0.02

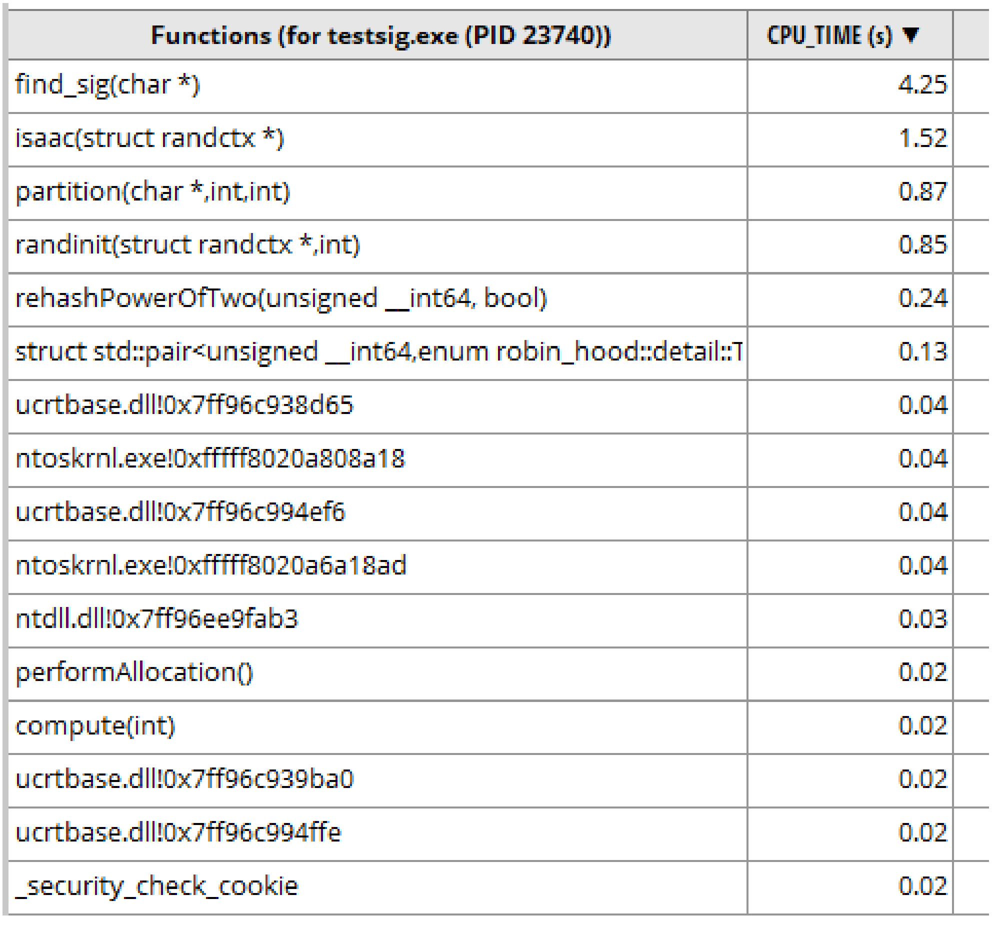
qut2.fasta



Functions (for Seqsigs.exe (PID 16564))	CPU_TIME (s) ▼
find_sig(char *)	18.93
compute_signature(char *,int)	6.84
ucrtbase.dll!0x7ff96c938d65	2.68
ntdll.dll!0x7ff96ee9fab3	1.20
ntdll.dll!0x7ff96ee9f25f	0.46
ucrtbase.dll!0x7ff96c994ef6	0.40
main	0.31

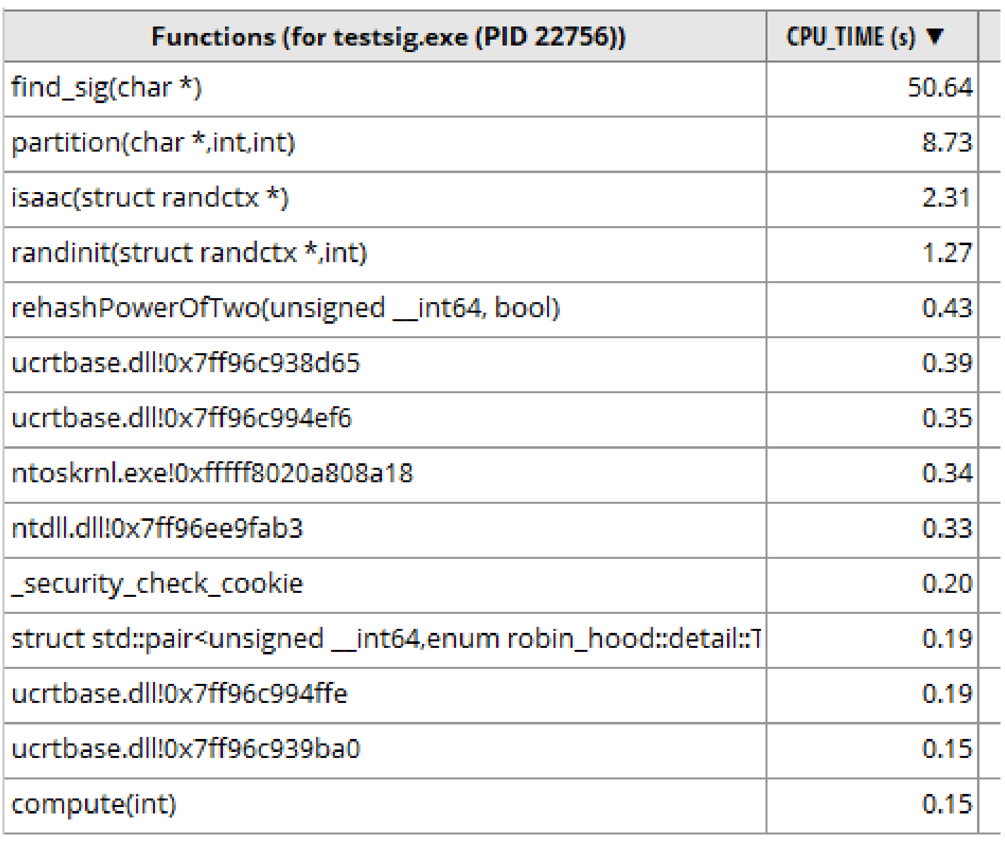
Parallelised

qut3.fasta



Functions (for testsig.exe (PID 23740))	CPU_TIME (s) ▼
find_sig(char *)	4.25
isaac(struct randctx *)	1.52
partition(char *,int,int)	0.87
randinit(struct randctx *,int)	0.85
rehashPowerOfTwo(unsigned __int64, bool)	0.24
struct std::pair<unsigned __int64,enum robin_hood::detail::T	0.13
ucrtbase.dll!0x7ff96c938d65	0.04
ntoskrnl.exe!0xfffff8020a808a18	0.04
ucrtbase.dll!0x7ff96c994ef6	0.04
ntoskrnl.exe!0xfffff8020a6a18ad	0.04
ntdll.dll!0x7ff96ee9fab3	0.03
performAllocation()	0.02
compute(int)	0.02
ucrtbase.dll!0x7ff96c939ba0	0.02
ucrtbase.dll!0x7ff96c994ffe	0.02
_security_check_cookie	0.02

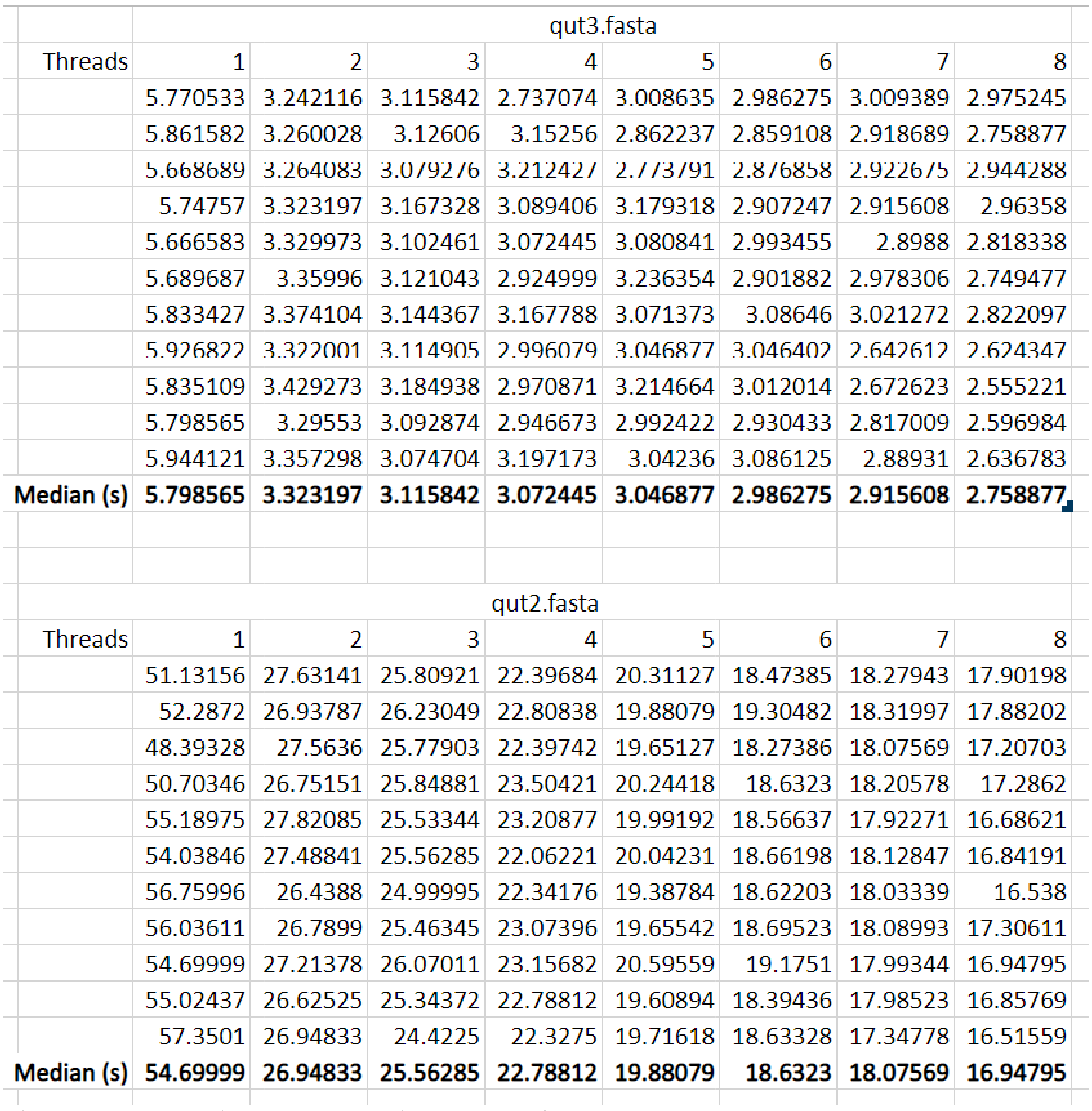
qut2.fasta



Functions (for testsig.exe (PID 22756))	CPU_TIME (s) ▼
find_sig(char *)	50.64
partition(char *,int,int)	8.73
isaac(struct randctx *)	2.31
randinit(struct randctx *,int)	1.27
rehashPowerOfTwo(unsigned __int64, bool)	0.43
ucrtbase.dll!0x7ff96c938d65	0.39
ucrtbase.dll!0x7ff96c994ef6	0.35
ntoskrnl.exe!0xfffff8020a808a18	0.34
ntdll.dll!0x7ff96ee9fab3	0.33
_security_check_cookie	0.20
struct std::pair<unsigned __int64,enum robin_hood::detail::T	0.19
ucrtbase.dll!0x7ff96c994ffe	0.19
ucrtbase.dll!0x7ff96c939ba0	0.15
compute(int)	0.15

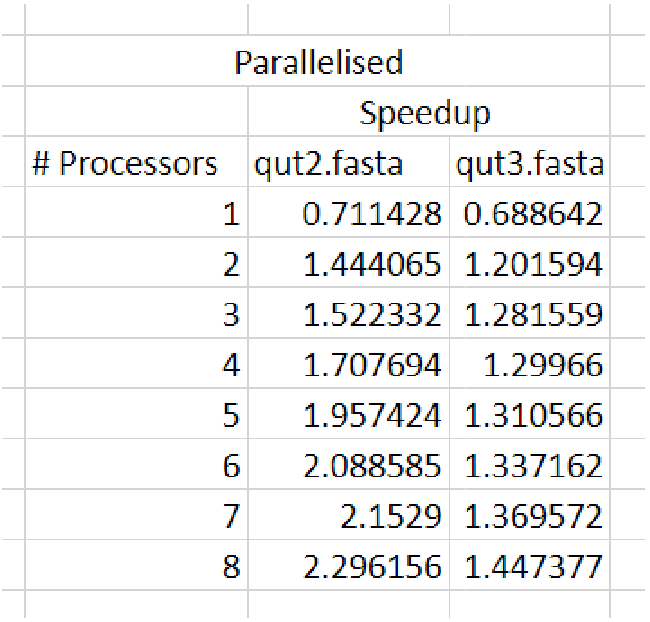
Note: The results of the parallelised version may seem to be slower than the sequential since CPU spin time is consumed by the **find_sig** function, but in the parallelised version, CPU spin time is spent less on **fwrite**, and more evenly distributed amongst other functions.

Timing Results – Parallelised



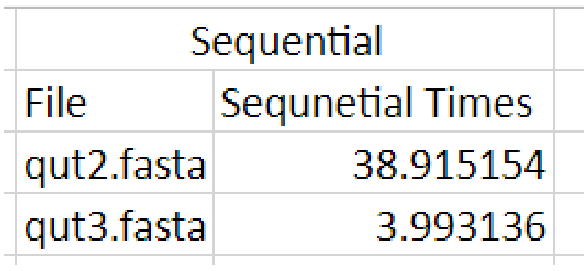
	qut3.fasta							
Threads	1	2	3	4	5	6	7	8
	5.770533	3.242116	3.115842	2.737074	3.008635	2.986275	3.009389	2.975245
	5.861582	3.260028	3.12606	3.15256	2.862237	2.859108	2.918689	2.758877
	5.668689	3.264083	3.079276	3.212427	2.773791	2.876858	2.922675	2.944288
	5.74757	3.323197	3.167328	3.089406	3.179318	2.907247	2.915608	2.96358
	5.666583	3.329973	3.102461	3.072445	3.080841	2.993455	2.8988	2.818338
	5.689687	3.35996	3.121043	2.924999	3.236354	2.901882	2.978306	2.749477
	5.833427	3.374104	3.144367	3.167788	3.071373	3.08646	3.021272	2.822097
	5.926822	3.322001	3.114905	2.996079	3.046877	3.046402	2.642612	2.624347
	5.835109	3.429273	3.184938	2.970871	3.214664	3.012014	2.672623	2.555221
	5.798565	3.29553	3.092874	2.946673	2.992422	2.930433	2.817009	2.596984
	5.944121	3.357298	3.074704	3.197173	3.04236	3.086125	2.88931	2.636783
Median (s)	**5.798565**	**3.323197**	**3.115842**	**3.072445**	**3.046877**	**2.986275**	**2.915608**	**2.758877**

	qut2.fasta							
Threads	1	2	3	4	5	6	7	8
	51.13156	27.63141	25.80921	22.39684	20.31127	18.47385	18.27943	17.90198
	52.2872	26.93787	26.23049	22.80838	19.88079	19.30482	18.31997	17.88202
	48.39328	27.5636	25.77903	22.39742	19.65127	18.27386	18.07569	17.20703
	50.70346	26.75151	25.84881	23.50421	20.24418	18.6323	18.20578	17.2862
	55.18975	27.82085	25.53344	23.20877	19.99192	18.56637	17.92271	16.68621
	54.03846	27.48841	25.56285	22.06221	20.04231	18.66198	18.12847	16.84191
	56.75996	26.4388	24.99995	22.34176	19.38784	18.62203	18.03339	16.538
	56.03611	26.7899	25.46345	23.07396	19.65542	18.69523	18.08993	17.30611
	54.69999	27.21378	26.07011	23.15682	20.59559	19.1751	17.99344	16.94795
	55.02437	26.62525	25.34372	22.78812	19.60894	18.39436	17.98523	16.85769
	57.3501	26.94833	24.4225	22.3275	19.71618	18.63328	17.34778	16.51559
Median (s)	**54.69999**	**26.94833**	**25.56285**	**22.78812**	**19.88079**	**18.6323**	**18.07569**	**16.94795**



Parallelised		
	Speedup	
# Processors	qut2.fasta	qut3.fasta
1	0.711428	0.688642
2	1.444065	1.201594
3	1.522332	1.281559
4	1.707694	1.29966
5	1.957424	1.310566
6	2.088585	1.337162
7	2.1529	1.369572
8	2.296156	1.447377

Timing Results – Sequential

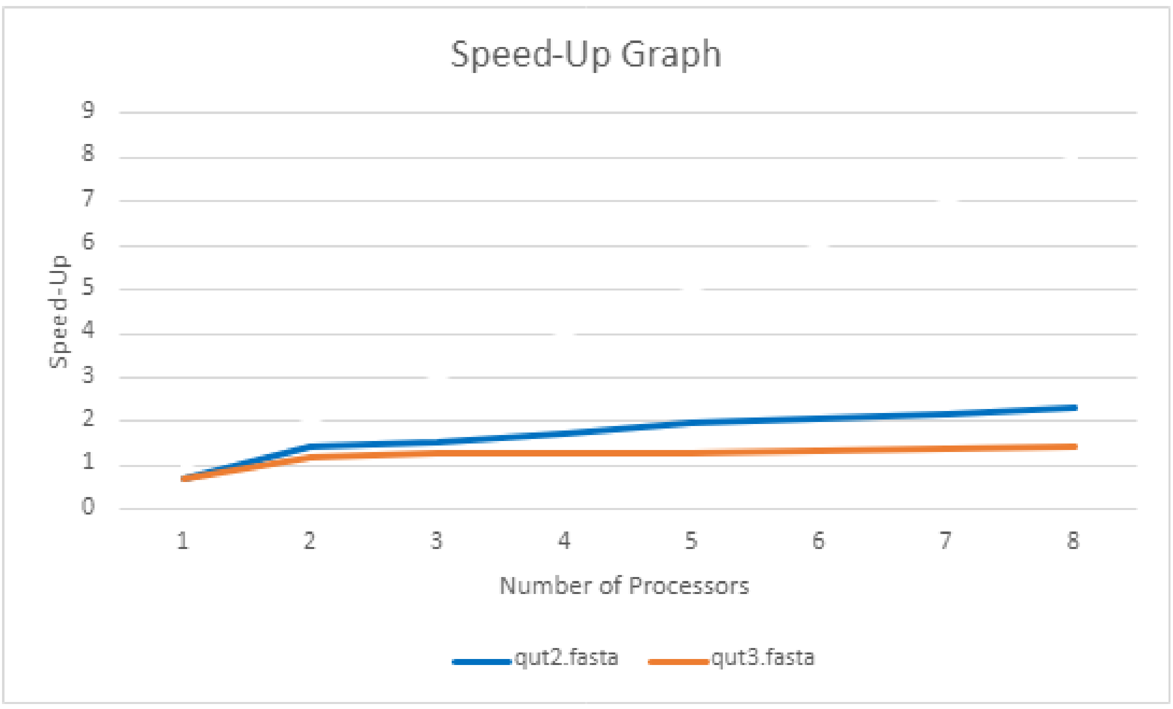


Sequential	
File	Sequnetial Times
qut2.fasta	38.915154
qut3.fasta	3.993136

Note: To construct my speedup graph, I used the following speed up equation given in week 3 lecture:

Speedup is defined as:

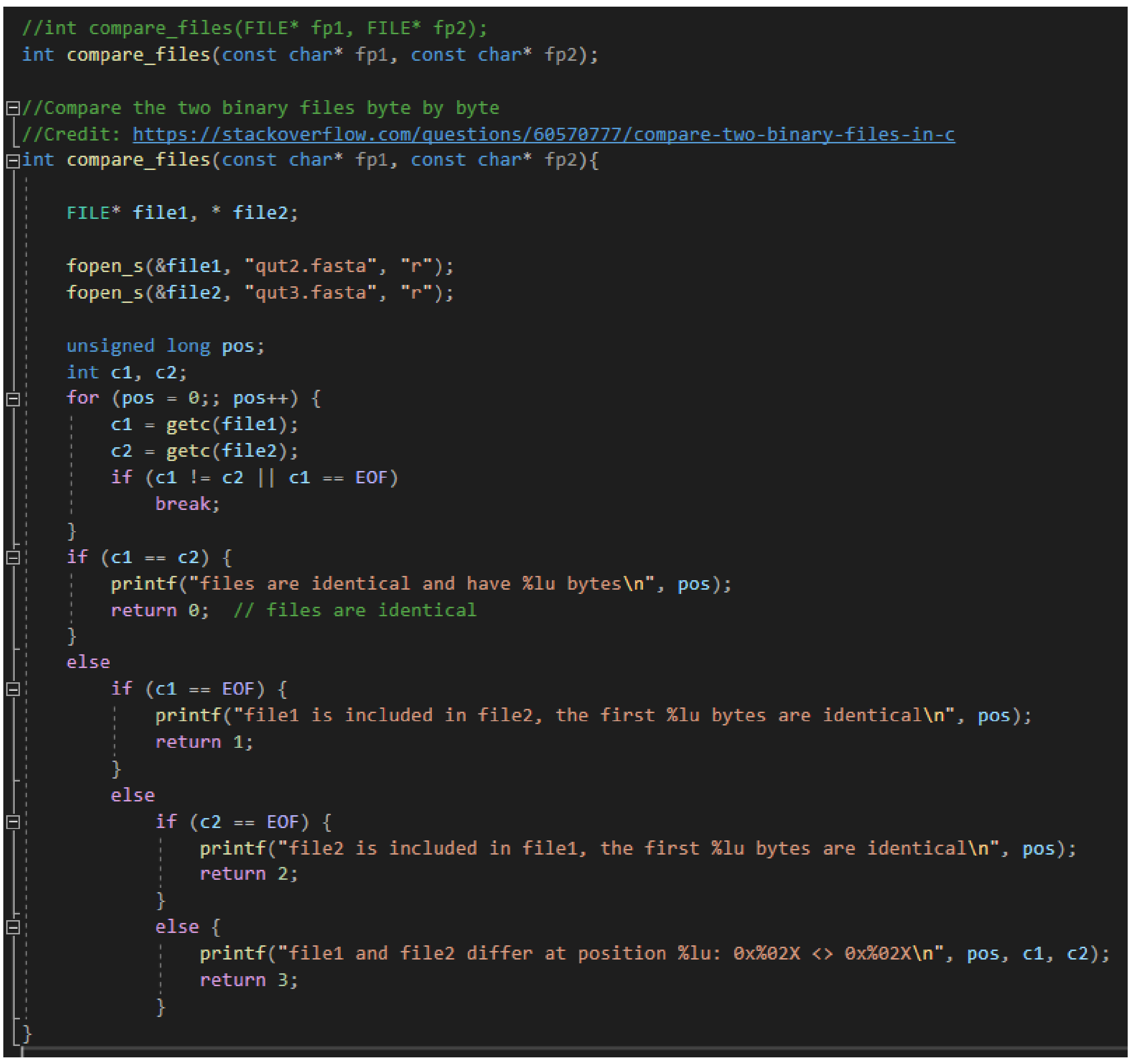
$$\frac{\text{execution time of best sequential program}}{\text{execution time of parallel program}}$$



Scalable Parallelism.

5 Testing

Since the application supplies two **.fasta** binary files, I initially decided to merge both files with the intention that testing would be fair, but when I applied the merged file to both versions of the application it outputted incorrect values, therefore scraping this idea. So, I ended up running each file individually in release mode at x64 in both the sequential and parallelised applications. In addition, I also conducted a bit more research to see if I could do anymore in ensuring the parallelised application produced the exact same results as the original sequential version. I wound up finding the following function on stack overflow that compares both files byte by byte which returns a message if byte position for either file differ or are identical (fig 5.1).



```
//int compare_files(FILE* fp1, FILE* fp2);
int compare_files(const char* fp1, const char* fp2);

//Compare the two binary files byte by byte
//Credit: https://stackoverflow.com/questions/60570777/compare-two-binary-files-in-c
int compare_files(const char* fp1, const char* fp2){

    FILE* file1, * file2;

    fopen_s(&file1, "qut2.fasta", "r");
    fopen_s(&file2, "qut3.fasta", "r");

    unsigned long pos;
    int c1, c2;
    for (pos = 0;; pos++) {
        c1 = getc(file1);
        c2 = getc(file2);
        if (c1 != c2 || c1 == EOF)
            break;
    }
    if (c1 == c2) {
        printf("files are identical and have %lu bytes\n", pos);
        return 0;  // files are identical
    }
    else
        if (c1 == EOF) {
            printf("file1 is included in file2, the first %lu bytes are identical\n", pos);
            return 1;
        }
        else
            if (c2 == EOF) {
                printf("file2 is included in file1, the first %lu bytes are identical\n", pos);
                return 2;
            }
            else {
                printf("file1 and file2 differ at position %lu: 0x%02X <> 0x%02X\n", pos, c1, c2);
                return 3;
            }
}
```

Figure 5.1: Function to compare two binary files.

6 Tools

Software, Tools, and Technologies:

- **Visual Studio 2019 IDE**
    - Since, the kmer-signatures application is written in c++, I chose to use Visual Studio 2019 IDE developed by Microsoft as my primary parallelisation tool. The IDE is essentially a complier, debugger, and performance profiler all in one that supports C#, C++, Python, .NET and other programming languages. Furthermore, Visual Studio allows the integration of third-party tools and libraries, and it also can be used on the Windows and Linux operating systems.

- **AMD µProf**
    - I briefly utilised AMD µProf profiler as I have an AMD processor, it offered a bit more information where Visual Studio Profiler could not such as CPU spin time. The profiler is a performance analysis tool that's used to effectively identify bottlenecks, identify ways to optimise code, analyse thread concurrency, and many more applications. In addition, AMD µProf supports both Windows and Linux operating systems as well as supporting C, C++, Fortran Assembly, Java, and .NET programming languages.

- **PPL (Parallel Patterns Library)**
    - PPL isn't a third-party library, it's a default library already included in Visual Studio, but as stated earlier it provides programming models similar to that of the C++ standard library for scalability and ease-of-use in development of concurrent applications.

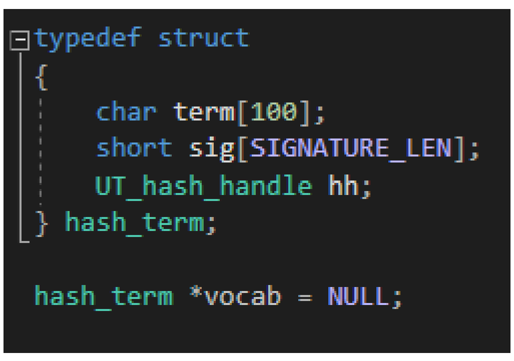
- **PC**
    - I obviously had to use my pc to parallelise the k-mer signatures application, it runs windows 10 and has an 8-core AMD processor.

7 Obstacles

I initially experimented with parallelisation at certain functions within the call tree such as the **compute_signature** loop, which ended up being a bust as after running the profiler tool, it showed that the program had become slower than the sequential version due to the parallel overheads; so, I decided to scrap this idea, and focus my attention on parallelising further up the call tree which was the partition loop. I knew that this loop computed signatures in groups that then sequentially writes to the **.fasta** files, and which made parallelism much more difficult, but I was confident that a decent speed-up could be achieved. As I mentioned previously, I used a **reader-writer** lock mechanism in controlling thread-safe access to the hashmap, however, this method of synchronisation at first seemed promising as I assumed threads reading the hashmap were more likely to occur than writing to the hashmap, but it became apparent through the use of both Visual Studio and AMD µProf profiling tools, the application still lacked in performance as these profiling tools highlighted that the threads were spending time waiting for the locks to be released which instantly stopped any chance of achieving a decent speed-up; ultimately leading me to ditching the reader-writer function.

Meanwhile, I began thinking if the hashmap was killing my performance so I tested this by removing the entire hashmap, and computing each signature individually, although being a painful process it allowed me to understand that the hashmap was indeed needed in my speed-up process, but now the problem was to avoid computing each signature individually multiple times, and the solution I discovered was to allocate a locally threaded hashmap, where each thread is assigned a hashmap, thus eliminating the factor of having a thread to compute a signature for a sequence already computed. Moreover, after applying this approach, it produced a decent speed-up I was looking for, and so, I decided to rerun the profiler tools to discover if any other components could be optimised further. I ended up finding that a significant amount of CPU time was being consumed on the **fwrite** function located within the partition loop, and so, I optimised this loop to write computed signatures in chunks of 8 bytes which helped in improving the applications performance.

Furthermore, the information provided by the profiling tools had highlighted that an awful amount of time was also being spent on finding signatures within the hashmap, this was the typedef loop shown below in figure 7.1 that provides access to the **uthash** hashmap (fig 7.1), and I had assumed this hashmap was reasonably quick, but looking into other hashmap implementations proved to be useful as I stumbled across the **robin_hood** hashmap, and so, I utilised this hashmap which yielded far greater results than the **uthash** hashmap. Additionally, whilst I was achieving scalable parallelism thanks to the new code, I still believed more could be accomplished in achieving a far greater speed-up; So, I decided to run the profiling tools a couple more times which showed the program still wasn't increasing in speed when more resources were being allocated, and thanks to these profiling results I was also able to pinpoint the source of this bottleneck which was the computation of signatures, and sequences being read in sperate chunks of 8, this meant that some threads had to wait on other threads to finish computing before the loop could exit. After discovering this, I experimented a bit with the chunk sizes, and found that a chunk size of 2000 worked concurrently with more resources than the previous chunk size.

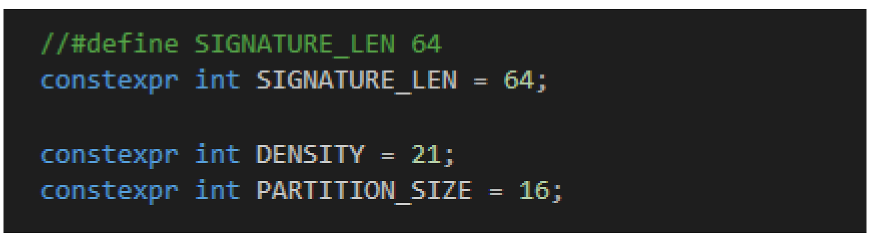


```
typedef struct
{
    char term[100];
    short sig[SIGNATURE_LEN];
    UT_hash_handle hh;
} hash_term;

hash_term *vocab = NULL;
```

Figure 7.1: calling uthash hashmap for threads

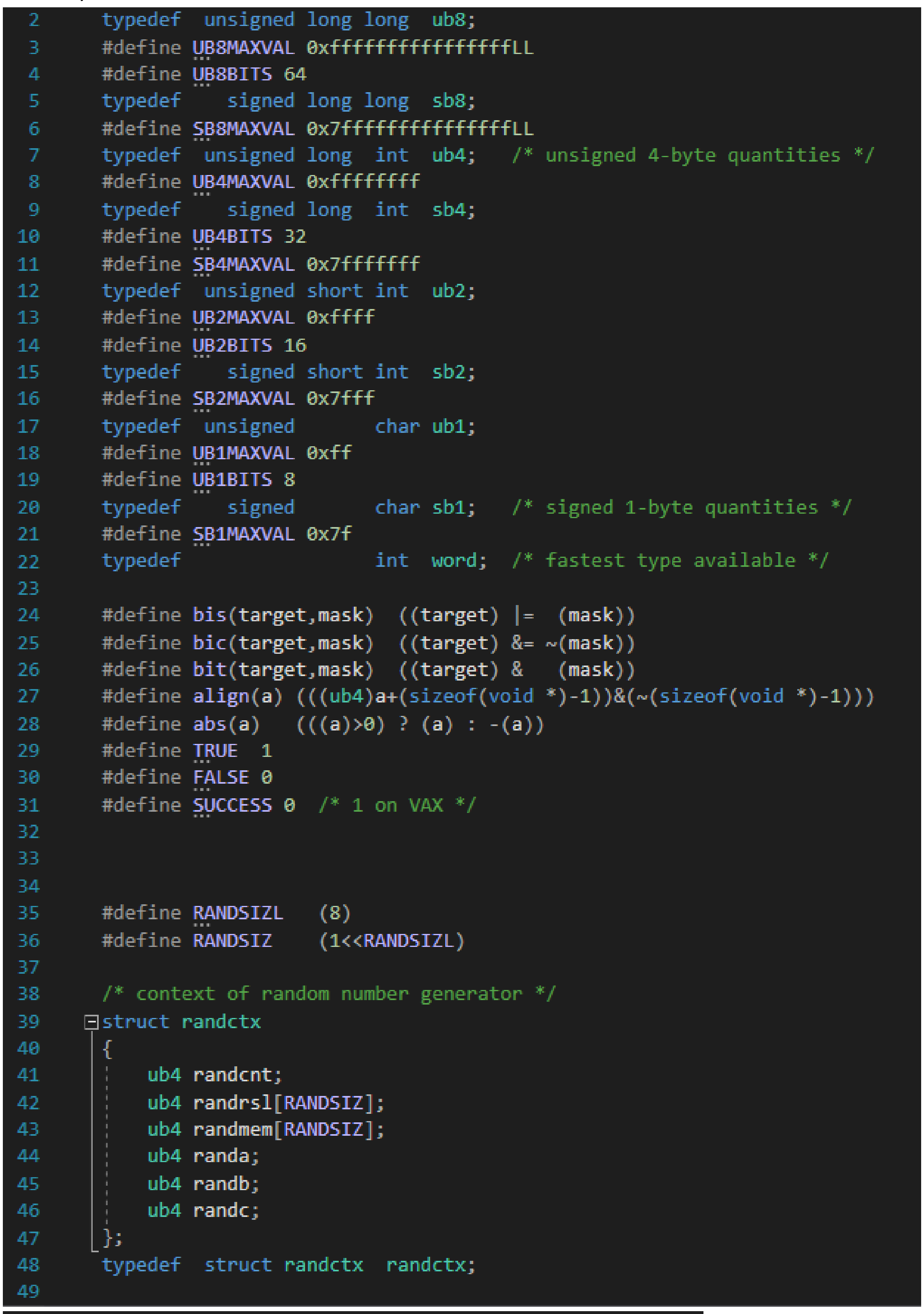
8 Code Explanation

I decided to add the **constexpr** variable to int declarations instead of a const as it allows values assigned to the variable to be initialised at compilation time rather than at run time which in turn allowed the program to gain a slight speed-up.

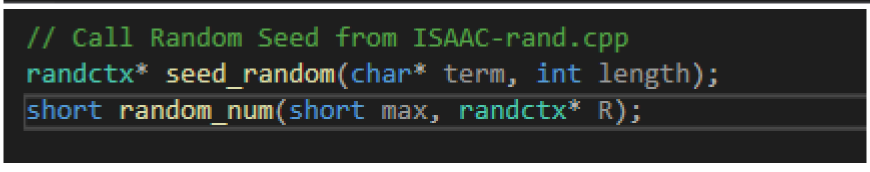


```
//#define SIGNATURE_LEN 64
constexpr int SIGNATURE_LEN = 64;

constexpr int DENSITY = 21;
constexpr int PARTITION_SIZE = 16;
```

I had to shift the entire first half of code from ISAAC-rand.cpp file into a header file called rand.hpp as the PPL library was throwing errors in unrecognising the unsigned byte quantities, as well as, calling the seed random functions from the main ISAAC-rand.cpp file so the program worked correctly as shown below.

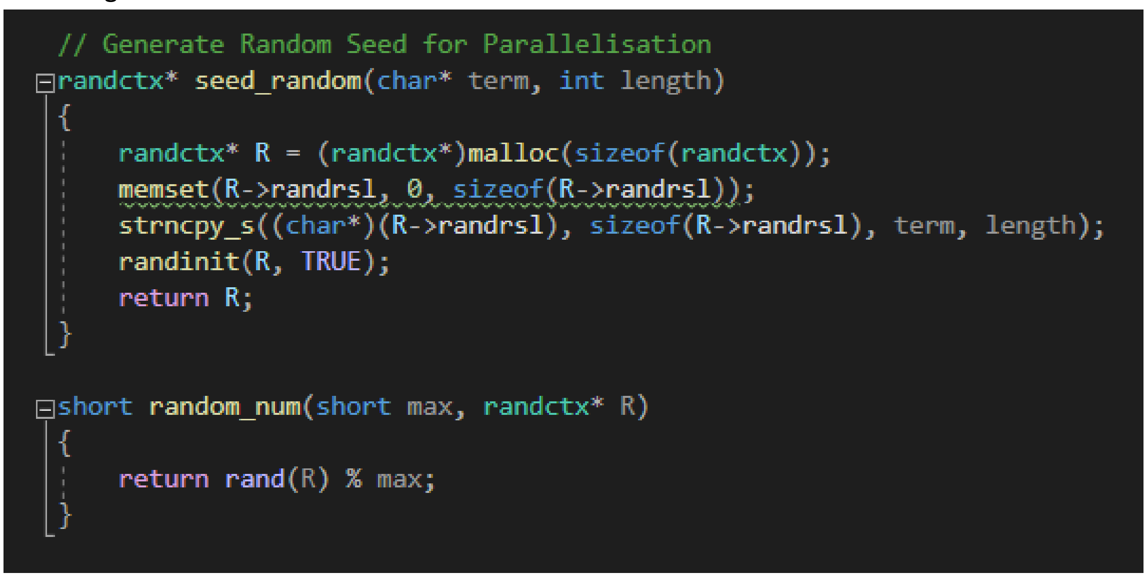


```
typedef  unsigned long long  ub8;
#define UB8MAXVAL 0xffffffffffffffffLL
#define UB8BITS 64
typedef    signed long long  sb8;
#define SB8MAXVAL 0x7fffffffffffffffLL
typedef  unsigned long  int  ub4;   /* unsigned 4-byte quantities */
#define UB4MAXVAL 0xffffffff
typedef    signed long  int  sb4;
#define UB4BITS 32
#define SB4MAXVAL 0x7fffffff
typedef  unsigned short int  ub2;
#define UB2MAXVAL 0xffff
#define UB2BITS 16
typedef    signed short int  sb2;
#define SB2MAXVAL 0x7fff
typedef  unsigned       char ub1;
#define UB1MAXVAL 0xff
#define UB1BITS 8
typedef    signed       char sb1;   /* signed 1-byte quantities */
#define SB1MAXVAL 0x7f
typedef                 int  word;  /* fastest type available */

#define bis(target,mask)  ((target) |=  (mask))
#define bic(target,mask)  ((target) &= ~(mask))
#define bit(target,mask)  ((target) &   (mask))
#define align(a) (((ub4)a+(sizeof(void *)-1))&(~(sizeof(void *)-1)))
#define abs(a)   (((a)>0) ? (a) : -(a))
#define TRUE  1
#define FALSE 0
#define SUCCESS 0  /* 1 on VAX */



#define RANDSIZL   (8)
#define RANDSIZ    (1<<RANDSIZL)

/* context of random number generator */
struct randctx
{
    ub4 randcnt;
    ub4 randrsl[RANDSIZ];
    ub4 randmem[RANDSIZ];
    ub4 randa;
    ub4 randb;
    ub4 randc;
};
typedef  struct randctx  randctx;
```



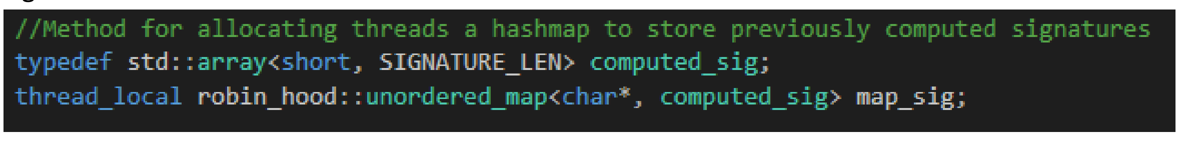
```
// Call Random Seed from ISAAC-rand.cpp
randctx* seed_random(char* term, int length);
short random_num(short max, randctx* R);
```

Within the ISAAC-rand.cpp file, I modified **seed_random** loop to allow for multi-threading, so now this function returns random seeds, and the function underneath called **random_num** was also slightly modified to accept this random seed as a parameter instead of it being stored as a global which was the variable **randctx R**. The **randctx** variable is a context constructer for the random number generator as shown above.



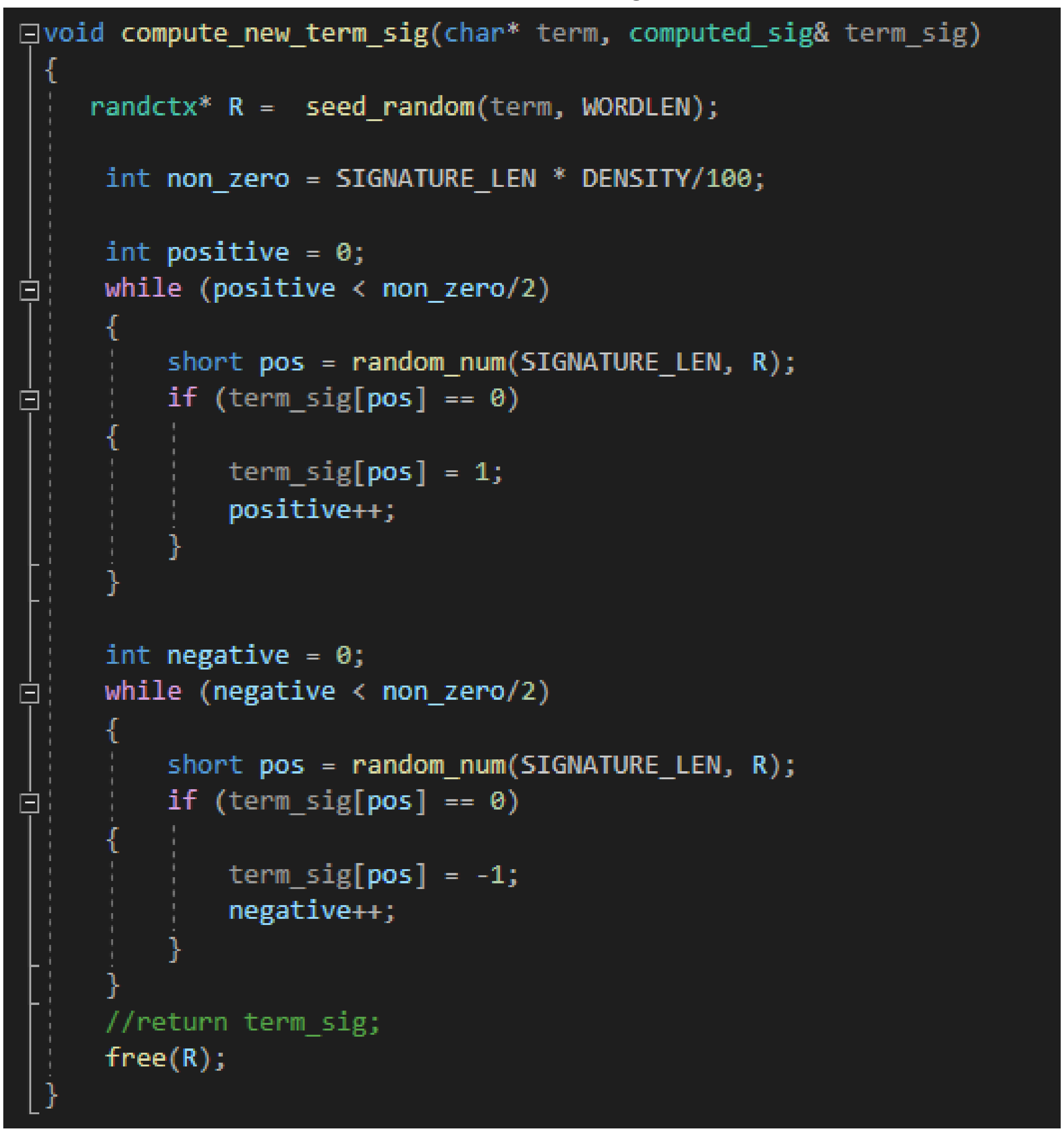
```
// Generate Random Seed for Parallelisation
randctx* seed_random(char* term, int length)
{
    randctx* R = (randctx*)malloc(sizeof(randctx));
    memset(R->randrsl, 0, sizeof(R->randrsl));
    strncpy_s((char*)(R->randrsl), sizeof(R->randrsl), term, length);
    randinit(R, TRUE);
    return R;
}

short random_num(short max, randctx* R)
{
    return rand(R) % max;
}
```

I removed the **utash hashmap** method in the main signature file, and replaced it with the following, this new method uses the **robin_hood hashmap** that allows threads to be allocated their own hashmap creating a much more efficient method in storing and retrieving previously computed signatures.



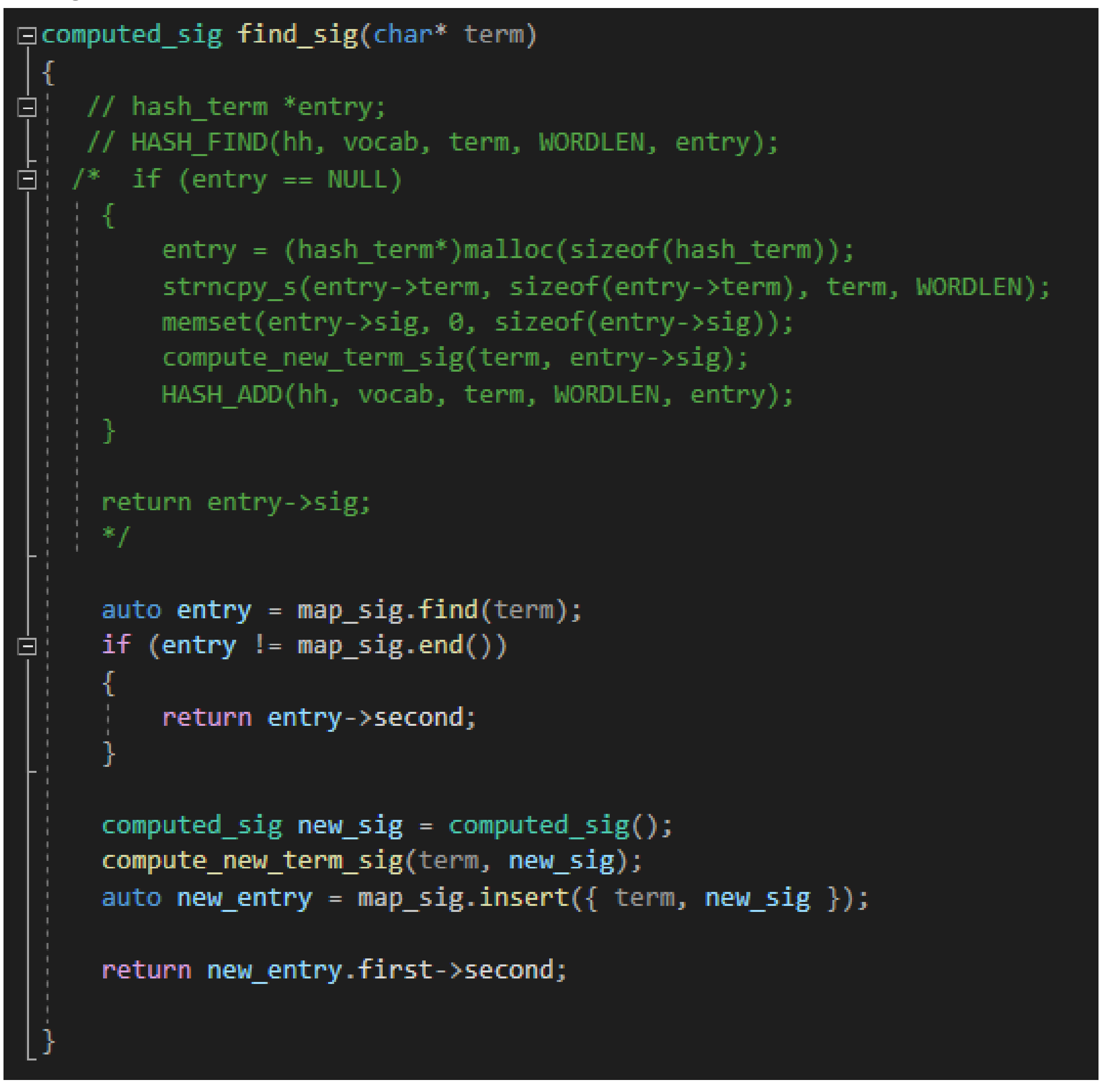
```
//Method for allocating threads a hashmap to store previously computed signatures
typedef std::array<short, SIGNATURE_LEN> computed_sig;
thread_local robin_hood::unordered_map<char*, computed_sig> map_sig;
```

I called the function **randctx** into the **compute_new_term_sig** loop so that it has direct access to the random seed/number generator from the ISAAC-rand.cpp file, and I also changed the result type from a short to void so that it doesn't overload and has global access.



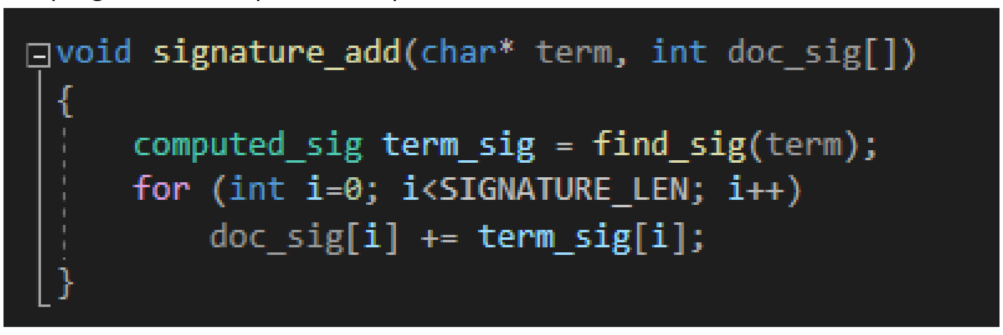
```
void compute_new_term_sig(char* term, computed_sig& term_sig)
{
   randctx* R =  seed_random(term, WORDLEN);

    int non_zero = SIGNATURE_LEN * DENSITY/100;

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

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

I had to perform a massive overhaul of the find_sig loop as I found a different hashmap as discussed previously, and changed the short result type to the typedef **computed_sig** name to allow easier coding.



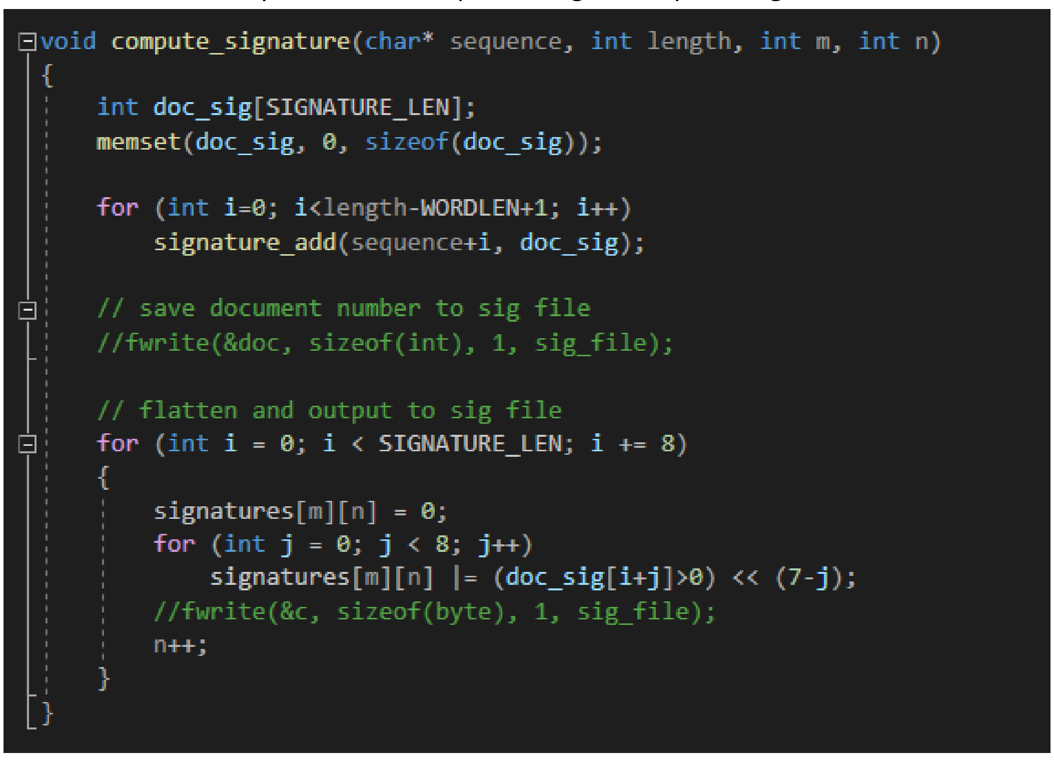
```
computed_sig find_sig(char* term)
{
   // hash_term *entry;
   // HASH_FIND(hh, vocab, term, WORDLEN, entry);
  /*  if (entry == NULL)
    {
        entry = (hash_term*)malloc(sizeof(hash_term));
        strncpy_s(entry->term, sizeof(entry->term), term, WORDLEN);
        memset(entry->sig, 0, sizeof(entry->sig));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

    return entry->sig;
    */

    auto entry = map_sig.find(term);
    if (entry != map_sig.end())
    {
        return entry->second;
    }

    computed_sig new_sig = computed_sig();
    compute_new_term_sig(term, new_sig);
    auto new_entry = map_sig.insert({ term, new_sig });

    return new_entry.first->second;

}
```

I added the typedef declaration, computed_sig, and provided access to the doc_sig array, so that the signature_add loop can efficiently access the hashmap to add new signatures that don't exist, and so the program doesn't produce duplicates.



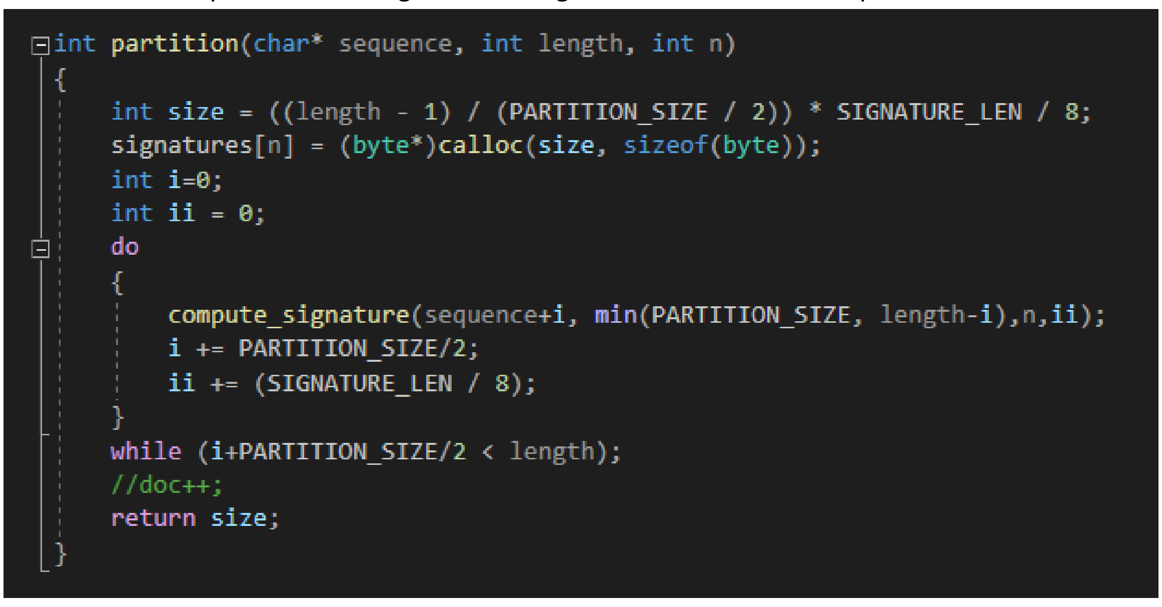
```
void signature_add(char* term, int doc_sig[])
{
    computed_sig term_sig = find_sig(term);
    for (int i=0; i<SIGNATURE_LEN; i++)
        doc_sig[i] += term_sig[i];
}
```

I performed minor changes to the **compute_signature** by localising **doc_sig** instead of it being a global variable, as well as, removing the **fwrite** method since the double loop called compute performs a similar method to this, and I've also added two int variables, m and n, to place within a two-dimensional array to allow faster processing of computed signatures.



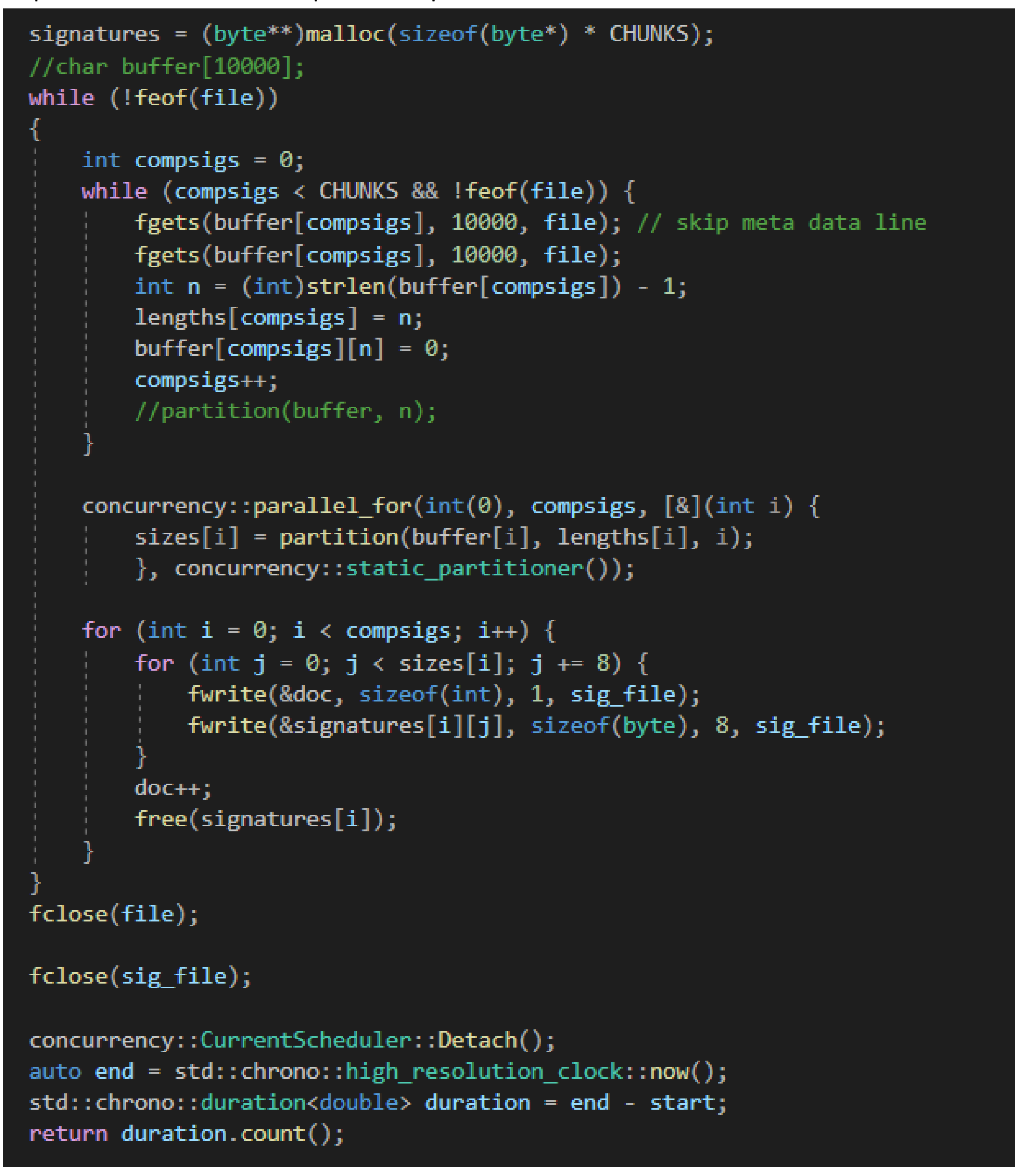
```
void compute_signature(char* sequence, int length, int m, int n)
{
    int doc_sig[SIGNATURE_LEN];
    memset(doc_sig, 0, sizeof(doc_sig));

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

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

The partition loop was slightly modified to allocate space for memory to the signatures sub-array, as well as, incrementing the array index of signatures, and the loop returns the size of signatures so that the main loop knows the length of each signature that needs to be processed.



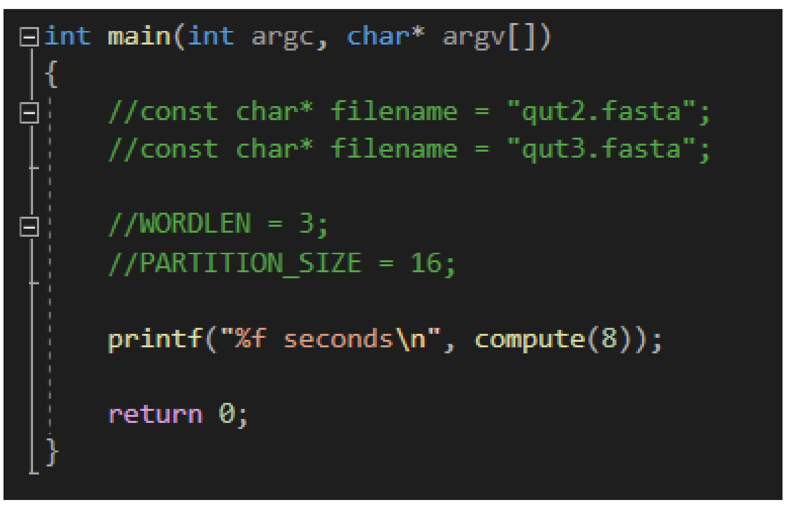
```
int partition(char* sequence, int length, int n)
{
    int size = ((length - 1) / (PARTITION_SIZE / 2)) * SIGNATURE_LEN / 8;
    signatures[n] = (byte*)calloc(size, sizeof(byte));
    int i=0;
    int ii = 0;
    do
    {
        compute_signature(sequence+i, min(PARTITION_SIZE, length-i),n,ii);
        i += PARTITION_SIZE/2;
        ii += (SIGNATURE_LEN / 8);
    }
    while (i+PARTITION_SIZE/2 < length);
    //doc++;
    return size;
}
```

A chunk of the following code was taken from the main loop (fig 1.2) and embedded into a double loop called compute where the while loop was modified to read sequences from an input file in chunks of 2000 as I explained previously, and that was then followed up by the parallel_for function to process the extracted sequences in parallel.



```
signatures = (byte**)malloc(sizeof(byte*) * CHUNKS);
//char buffer[10000];
while (!feof(file))
{
    int compsigs = 0;
    while (compsigs < CHUNKS && !feof(file)) {
        fgets(buffer[compsigs], 10000, file); // skip meta data line
        fgets(buffer[compsigs], 10000, file);
        int n = (int)strlen(buffer[compsigs]) - 1;
        lengths[compsigs] = n;
        buffer[compsigs][n] = 0;
        compsigs++;
        //partition(buffer, n);
    }

    concurrency::parallel_for(int(0), compsigs, [&](int i) {
        sizes[i] = partition(buffer[i], lengths[i], i);
        }, concurrency::static_partitioner());

    for (int i = 0; i < compsigs; i++) {
        for (int j = 0; j < sizes[i]; j += 8) {
            fwrite(&doc, sizeof(int), 1, sig_file);
            fwrite(&signatures[i][j], sizeof(byte), 8, sig_file);
        }
        doc++;
        free(signatures[i]);
    }
}
fclose(file);

fclose(sig_file);

concurrency::CurrentScheduler::Detach();
auto end = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> duration = end - start;
return duration.count();
```

The main loop was reduced from the figure 1.2 shown above to only printing out the time taken to process either file from the double loop called compute, and the value within the brackets represents the number of threads used to process the data. The number within the brackets can be adjusted to any value.



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

    //WORDLEN = 3;
    //PARTITION_SIZE = 16;

    printf("%f seconds\n", compute(8));

    return 0;
}
```

9 Reflection

Throughout the parallelisation process of k-mer signatures, I've learnt quite a bit about parallel computing and programming in C++, and being quite versed in using Visual Studio I knew that the performance profiler tool would immediately be helpful in identifying bottlenecks, as well as, determining how effective my attempts at parallelisation of the application are. Moreover, the information I've learnt from the lecture material and self-paced practicals, I was able to translate this understanding into my own parallelisation project by deciding to locally thread a hashmap over a shared hashmap as I realised the benefits of not including the mutex lock were far greater than potentially computing a sequence multiple times. Additionally, I've also learnt that synchronisation methods such as mutex locks mentioned above, have a large parallel overhead that need careful consideration when required, and it's paramount to restructure an application in a way that's best suited for parallelisation rather than an application written sequentially.

Meanwhile, I've also learnt that it's worthwhile to conduct a bit of research into other well-known libraries that can be beneficial towards the parallelisation process. I took this exact approach before I began parallelising k-mer signatures, and ended up finding the Parallel Patterns Library developed by Microsoft. Furthermore, I also discovered through some research and trials that hashmap implementations like **std::unordered_map** may produce a decent gain, but believed a greater speed-up could be achieved which led to me discovering a hashmap called **robin_hood.** The benchmarks provided by martinus, the **robin_hood** hashmap author, gave me the confidence that using **robin_hood::unordered_map** would definitely be faster. Overall, I would say my attempts at parallelisation were adequate enough to produce a reasonable speed-up, but I still firmly believe that an even faster speed-up could've been achieved, and if I were given the opportunity again to parallelise this application, I would like to focus more on parallelising coarser grained parts than the fine grained, as well as, exploring a solution that involved the producer-consumer problem, where one thread reads and writes while the other threads are processing.

10 Instructions

I've included both the sequential and parallelised versions of the k-mer signatures application where the sequential file is named Seqsigs, and the parallelised file named testsigs. I recommend that you should have Visual Studio 2019 installed on your PC before attempting to open either solution. I've also included the two .fasta files, qut2 and qut3; Once you enter either solution make sure you uncomment or comment out one of the files in **only** the double loop and not the main loop. The program requires one file to be processed at a time, and ensure that your debugger in Visual Studio is set to release mode at x64 to see the best results. Furthermore, in the main function (fig 8.3) where the number is enclosed within the brackets, you are able to set the number of threads to your own preference.

References

Docs.microsoft.com. 2019. *constexpr (C++)*. [online] Available at: https://docs.microsoft.com/en-us/cpp/cpp/constexpr-cpp?view=msvc-160 [Accessed 31 October 2021].

Docs.microsoft.com. 2019. *Parallel Patterns Library (PPL)*. [online] Available at: https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl?view=msvc-160 [Accessed 31 October 2021].

Docs.microsoft.com. 2019. *Synchronization Data Structures*. [online] Available at: https://docs.microsoft.com/en-us/cpp/parallel/concrt/synchronization-data-structures?view=msvc-160#reader_writer_lock [Accessed 31 October 2021].

Docs.microsoft.com. 2019. *Task Scheduler (Concurrency Runtime)*. [online] Available at: https://docs.microsoft.com/en-us/cpp/parallel/concrt/task-scheduler-concurrency-runtime?view=msvc-160 [Accessed 31 October 2021].

Docs.microsoft.com. 2019. *Parallel Algorithms*. [online] Available at: https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-algorithms?view=msvc-160 [Accessed 31 October 2021].

Docs.microsoft.com. 2021. *Concurrency Runtime*. [online] Available at: https://docs.microsoft.com/en-us/cpp/parallel/concrt/concurrency-runtime?view=msvc-160 [Accessed 31 October 2021].

En.wikipedia.org. 2021. *k-mer - Wikipedia*. [online] Available at: https://en.wikipedia.org/wiki/K-mer [Accessed 11 September 2021].

AMD. 2021. *AMD μProf - AMD*. [online] Available at: https://developer.amd.com/amd-uprof/ [Accessed 29 October 2021].

Sanfoundry. n.d. *C Program to Compare two Binary Files, Printing the First Byte Position where they Differ - Sanfoundry*. [online] Available at: https://www.sanfoundry.com/c-program-compare-two-binary-files/ [Accessed 28 October 2021].

C, C. and Thakar, B., 2020. *Compare two binary files in C*. [online] Stack Overflow. Available at: https://stackoverflow.com/questions/60570777/compare-two-binary-files-in-c [Accessed 28 October 2021].