Jesse Batt (15004481)

Parallel Computing - Logbook

Contents

Introduction to Parallel Algorithms

When considering programs and algorithms, the question of parallel versus sequential will often be raised. Sequential algorithms will work in a fashion where each process is run after the other, so the next process in the sequence can not start until the previous one has completed. Processes in sequential algorithms execute in a consecutive and ordered way, with the design of said algorithms often based around a chronological series of events.

Parallel algorithms work very differently, processes in these algorithms execute concurrently (at the same time - in parallel to each other). The programming of parallel algorithms will usually focus on shared memory programming, or distributed memory programming. As a whole, Parallel Computing will utilise multiple resources such as cores in a processor in multi/many core systems, or a number of machines in multi computer systems such as Clusters or Grids.

Parallel algorithms are suited well for the computation of complex problems, that involve large, or a large number of, calculations. When it comes to these inevitably more complex problems, using a parallel approach to solving them is a more efficient and effective utilisation of the resources available in a system.

Some limitations however include portability. With APIs such as OpenMP and MPI, portability is less of an issue compared to in the past, but changes in Operating Systems and hardware architectures can have a detrimental effect on portability.

OpenMP and How It Is Used

OpenMP (Open Multi-Processing) is an API for C/C++ that supports multi platform shared memory programming. It contains a set of library routines, environment variables and

compiler directives that can all be used to manipulate runtime behaviour, and is one of the key means of parallelisation presented in this report.

In short, OpenMP is a means of multithreading. This is the idea that a master thread, which contains a set of consecutive instructions, and can fork a number of slave threads to divide tasks and workload accordingly. The number of threads can be specified, but will depend on the environment being used. These slaves thread then execute simultaneously (in parallel). The following illustration (Figure 1) gives a clear representation of the principle and execution of multithreading and how it is used in the execution of parallel tasks in a program.

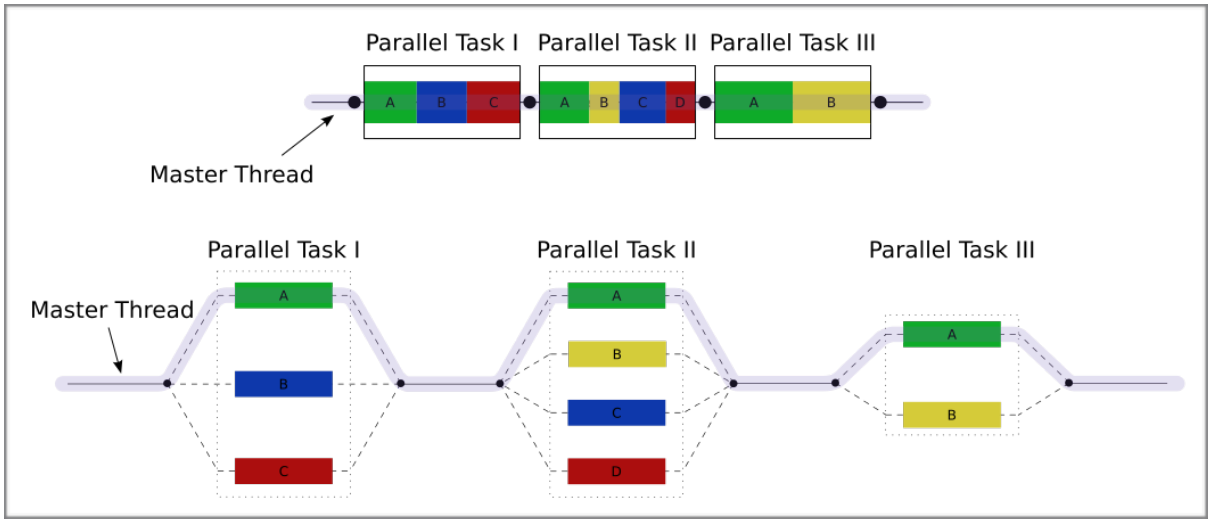

Figure 1 - An illustration of multithreading demonstrating how a master thread forks a number of slave threads that will execute blocks of code in parallel. (Source - Wikipedia)

To implement OpenMP within programs, you may define a section of code as a Parallel Region, code within this region may be optimised using OpenMP parallelisation and code outside of the region will execute sequentially. In a program using OpenMP, you can specify the number of parallel threads you wish to use (typically, this is set to 4 threads by default, providing that there are enough processing cores available on the system).

OpenMP also allows for use of Parallel For loops, which operate as one would expect from a for loop - but in this cause will be executing in parallel. For example, a user could assign one thread to a given for loop, and another thread to a different for loop, thus executing both in parallel.

Critical sections are also really useful in OpenMP programs, a typical use for Critical sections is for updating a variable that might be affected by the actions of various threads. A Critical section will enforce a lock, which will result in significant overhead, but at this point the user can then safely update a variable without the operation resulting in any false sharing of variables.

MPI and How It Is Used

MPI (Message Passing Interface) is a message passing API that is used for communication between systems and can be used for distributed memory programming. When considering a parallel computing context, one might encounter a Cluster made of multiple systems known as Nodes. Typically, each Node in a Cluster will work on a specific section of a program/problem. In doing so you can assume that there will be the need for

synchronisation between Nodes, information exchange will need to take place, and some form of command and control will also be a necessity.

MPI provides a variety of functions to facilitate this synchronisation, communication and control between Nodes. In a similar way to how threads operate, each Node will be assigned work via a message, and will then return the result of said work. Similarities between threads and MPI is that they both incorporate the distribution of tasks and separate processing in each, and that the overhead will increase as more Nodes become involved in the system, so this requires delicate distribution of the work load by responsibility of the user.

The key difference between threads in OpenMP and Nodes in MPI is that MPI Nodes have their own memory (distributed memory), whereas OpenMP threads actually have shared memory. Hence the need for message passing between Nodes for information exchange, synchronisation and so on.

An Overview of the Assignment Problem

The assignment task is to implement a Brute Force Search, otherwise known as an exhaustive search, in order to find the solution for a 16 bit key that will then be used to decrypt an encrypted message. The task implementation will be completed using AES (Advanced Encryption Standard) and a library called OpenSSL. OpenSSL is an open source library for C/C++ which is created for use within security related contexts and contains very useful cryptographic functions and makes use of SSL and TSL protocols.

The Brute Force Search itself works like in a “trial and error” approach, so will test all combinations of the input, even if they don't produce the expected output.

Implementation Log and Observations

- **15/11/18 Researched Assignment Problem (Brute Force)**

I began the assignment by researching the assignment problem, Brute Force Search. In doing so I gained a clearer understanding of what the assignment program would need to implement in terms of its functionality. I also briefly researched the Advanced Encryption Standard (AES) for some background knowledge.

- **17/11/18 Analysed References From Specification**

Next, I looked into the reference links provided by the Assignment Specification. Information provided regarding EVP Encryption was useful as it explains how the example code provided operated in more detail, and can be used in our own Brute Force programs for encryption and decryption.

- **21/11/18 Researched OpenSSL Documentation**

The online resources for OpenSSL has pages dedicated to Symmetric Encryption using EVP functions as part of OpenSSL [1]. I researched these functions for use within my program. These functions for encryption and decryption will be invaluable for ensuring the successful implementation of EVP encryption and decryption within the Brute Force Search program. An error handler is also provided, which will also be useful.

Jesse Batt (15004481)

Serial Brute Force Search Implementation

Brute Force Search Approach and Initialisation

The way I decided to implement the Brute Force Search was a simple, albeit somewhat ugly, approach of using nested for loops in the program to implement the Brute Force Search in a “Generate and Test” fashion.

In my program, I have set up a True Key of 16 bits (“######1234######” for example), using hashes as padding to limit the search parameters for the sake of the program running within a reasonable length of time, yet still demonstrating the functionality and principles of the Brute Force Search. I also made a blank key variable to serve as the starting point for the Brute Force Search, which is formatted as “######xxxx######”. Aforementioned blank key variable will be used to store the potential solutions throughout the search.

The program starts by encrypting a plain text message using the provided encrypt function from the OpenSSL resources, and will encrypt the message using the hard coded True Key. It is important to note here that in a realistic context, the key would not be hard coded into the program but again, this is only so that we can observe the brute force search principles and functionality without overcomplicating the program.

To begin the Brute Force Search for the key solution, I have set up nested for loops, working to find the correct character for each location in the key. These loops search by using ASCII values for characters, and pushing each into the Test Key during the search.

Verification of Solution

In order to verify the solutions for the key, I am using the strcmp function as is part of the string library in C. This function is used for string comparison and will take in two string arguments, returning a value of 0 if the strings are identical, and a value of 1 if the strings do not match.

In this case, I am using strcmp to continually check throughout the search whether the current combination of the Test Key matches the actual True Key, the correct result. Once the Test Key matches the True Key, the program displays that the key has been found, and will then decrypt the encrypted message from the start of the program, now instead using the Key found as a result of the Brute Force Search. If the search and string comparison are both correct, then we can expect the decryption result to be identical to the initial plain text, and if so, then we can assume that the search performs successfully.

Observations / Issues Faced

An issue I encountered was that with certain keys, the program would not actually detect the solution, and would be stuck endlessly reiterating over the search and after a certain point in time, the program would encounter a segmentation fault - after some analysis into the code, it became obvious that this was a sign that the

conditions of the for loops were faulty. After some experimentation this turned out to be a case of changing

```
for (a = ASCIISTART; a < ASCIIEND; a++)
{

}
```

Into

```
for (a = ASCIISTART; a <= ASCIIEND; a++)
{

}
```

Which then ensured that no characters where unintentionally skipped over when searching for the key solution.

OpenMP Brute Force Search Implementation

Defining the Parallel Region and Setting Threads
The first step to take in implementing OpenMP into my program is to set the amount of threads I wish to use, which in this case is 4. I then assessed the serial/sequential version of my program and decided where to best initialise a Parallel Region. In this case, I felt it was best to do so immediately before the start of the nested for loops used, as including variable initialisation within the parallel region is unnecessary.

Research into Nested Parallelism and Work Sharing Loops
As much as possible, I wanted to ensure that the core program used the serial implementation of the Brute Force Search is not heavily altered for use within the OpenMP implementation, as to keep the programs as similar as possible for a more consistent platform for comparison during the benchmarking and performance analysis stage.

In order to achieve this, I had to research into efficient ways to implement nested parallelism. From OpenMP documentation and other resources [2], I was introduced to the Collapse clause. The Collapse clause can be used to “collapse” perfectly nested loops into one loop, the work load of which is then divided into “chunks”, and these chunks are then assigned to threads. By default, threads will not usually have more than one chunk assigned to them, but this will also depend on the implementation. The Collapse clause takes in a positive numerical argument, used to indicate the amount of iterations/loops required to collapse in the program.

This appeared to be a very reasonable and efficient solution to parallelising my serial algorithm for the Brute Force Search without compromising the functionality of the program or heavily editing the algorithm for the search itself. So I used the Collapse clause in my program, by outlining the section with “#pragma omp parallel for (4)” to specify that there are 4 iterations needed here to collapse. Beyond this, the functionality of the main search algorithm remains intact here.

Synchronisation at Solution Stage

Following this, I decided that it would be a useful step to include a Critical section, in order to allow for variables to be shared in order to generate the solution. Without the critical section it is possible that the program will keep searching despite a solution being found due to all 4 threads being active. Using a Critical Section here should ensure that only one thread completes the solution/decryption stage.

I decided to implement a Critical section around the section of the program which handles the solution stage (when the solution is found and decryption takes place). When running the program, the solution is still found correctly, but the runtime of the program increases astronomically. I was able to negate this by moving the critical section to start after the solution is found, as opposed to before. This resulted in a huge decrease in runtime, of approximately 9 seconds.

Now that this issue was resolved, I could now observe that the program operates successfully, still finding the correct key solution, exhibiting the correct decryption result and all while utilising tools from OpenMP.

Benchmarking and Performance Analysis

In order to compare and test the results of each program, I ran each program with different key solutions to be found, 5 times each then calculated the average runtime for all keys. As shown in the table below. The OpenMP implementation is running with 4 threads active in this case. I am running the programs on an Ubuntu Virtual Machine (using Parallels Desktop), on a MacBook Pro (Late 2017) with a 2.3 GHz Intel Core i5 (7th Gen) CPU. The Ubuntu Virtual Machine is configured to have access to all 4 cores of the CPU on the machine. Timings here are all measured in Wall Clock Time using the “omp_get_wtime()” function in OpenMP.

KEY	SERIAL BEST RUNTIME (s)	SERIAL AVG RUNTIME (s)	OPENMP BEST RUNTIME (s)	OPENMP AVG RUNTIME (S)	SPEEDUP	COST	TOTAL OVERHEAD	EFFICIENCY
ab57	0.4046	0.4748	0.5340	0.5533	0.8581	2.132	1.738	0.2145e
f78G	0.4337	0.5065	0.5771	0.5859	0.8644	2.344	1.837	0.2161e
9hb3	0.1605	0.1723	0.2190	0.2345	0.7348	0.938	0.766	0.1837e

From the results displayed in the table above, it is clear that while the OpenMP solution is still functional, it is in fact noticeably slower across all the key solution tests shown here and with sub optimal Speedup, Efficiency and Overhead. There are a number of overhead factors that could all influence the outcome of this data, including communication, idling, wasted computation and so on.

Here we are mainly concerned with Communication Overhead. Allowing threads to communicate and synchronise during program execution will actually introduce a considerable amount of runtime overhead, resulting in a program that takes longer to complete. In terms of my implementation of the OpenMP Brute Force Search, you can see that in making use of a Critical Section, I am therefore introducing significant communication overhead.

This overhead will also grow as more threads are introduced, one would assume that a greater overhead would be apparent if the implementation was running on a system using 8 threads. From this it is clear that the OpenMP implementation used here is actually suffering from pitfalls in its parallelisation, although it is unclear what aspects are causing it, having known the limitations of a Critical Section, you would be inclined to believe that is the direct cause.

There are alternatives to the Critical Section such as Locks, and Atomic sections. While Atomic sections will incur much less overhead, they are in fact very limited in terms of the operations that can be performed within them. A viable alternative in this scenario would be to use Locks on the Solution stage of the program to achieve the same desired outcome.

Conclusion

Overall it is clear that parallelisation methods are well suited for problems such as the Brute Force Search as outlines in the assignment task here, but when implementing this parallelisation into programs, the user must be careful when using certain tools to ensure that the program's functionality does not come at the cost of significant performance issues. In my program, I found this to be the case.

References

[1] OpenSSL (2017) *EVP Symmetric Encryption and Decryption.* Available from: https://wiki.openssl.org/index.php/EVP_Symmetric_Encryption_and_Decryption [Accessed 21 November 2018]

[2] OpenMP Architecture Review Board (2018) *OpenMP Application Programming Interface* [online] OpenMP. Available from: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf [Accessed 29 November 2018]

GitHub Link

https://github.com/JL-Batt/PC-Assignment