CAB401 Project

Oliver Strong n11037580

October 25, 2024

Contents

1 Introduction

This report will detail the process of parallelising a serial application, and the results of this process. The original application is a C program that reads in any number of files and performs a byte pair encoding process.

2 Original Application

The application is a procedural program with a simple command line interface to control it. It has two modes, both train and status. The train mode takes a project name, target vocabulary size, output filepath, and a n-long list of filepaths to train on. The status mode can take a filepath to the output file and will print the vocabulary to stdout as JSON from which the status of the training can be determined.

The call graph of the application was generated by Doxygen and dot and is shown below.

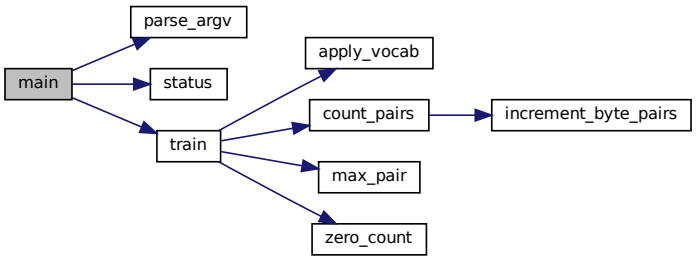

Figure 1: Call graph of the original application

The general algorithm of the original application is as follows

Algorithm 1 Original Application(mode, projectName, vocabSize, outputFilepath, inputFilepaths[..N])

\\Input: String *mode* which is either train or vocab, String *projectName*, Natural number *vocabSize*
\\ String outputFilepath, many strings inputFilepaths that is N long
\\Output: File *outputFilepath* with the vocabulary

$vocabulary \leftarrow \{\}$
if mode == train **then**
  for $i \leftarrow 0$ **to** *vocabLength* **do**
    $pairCounts \leftarrow \{\}$
    for $j \leftarrow 0$ **to** N **do**
      $file \leftarrow inputFilepaths[j]$
      $buff \leftarrow Read(file)$
      $buff \leftarrow transform(vocabulary, buff)$
      $pairCounts \leftarrow pairCounts \cup \text{countPairs}(buff)$
    end for
    $vocabulary \leftarrow vocabulary \cup \text{mostCommon}(pairCounts)$
    $outputFilepath \leftarrow vocabulary$
  end for
else if mode == status **then**
  $stdout \leftarrow JSON(vocabulary)$
end if

If called with the command `./train_vocab train project 260 output.vocab main.c` then the program will create a file called *output.vocab* that contains the vocabulary of the files *main.c*. There will be a list of four replacements which are the most common byte pairs where each is recursively replaced from the start of the list. As main.c is a C file it will contain common pairs found in C source code. You can expect that the first replacement made will be $(32, 32)$ to 256 as 32 is the ASCII code for a space character. If the author of the code used four spaces for the indentation then the next most common pair will be $(256, 256)$ which will be replaced with 257. as the four 32's will have been replaced with two 256's and then that will be replaced with a 257.

This vocabulary output can then be loaded at a later stage after being trained and used to encode some file you wish to feed into a language model.

A typical use case for this type of application is to take a large dataset, potentially multiple terrabytes of data and use it to train a vocabulary so some statistical model can be fitted on the encoded data. This technique allows for the model to see a more uniform distribution of token values as performing token level predictions on unencoded data would lead to it overfitting on the most common tokens.

3 Potential Parallelism

From the algorithm 1 it can be seen that how the vocabulary is used is a flow, output, and anti dependence with the loop from i to *vocabLength*. This is because the vocabulary starts empty and is built up on each iteration, this requires the previous iteration's value to be available so the files can be correctly transformed, This is a flow dependence. Then when the vocabulary is updated in each iteration it is an output dependence, and then in subsequent iterations the vocabulary is used to transform the files which is later overwritten when it is updated, this is an anti dependence.

As the nature of the algorithm is to build up the vocabulary in a loop, it is not possible to parallelise the outer loop. This makes the inner loop the first candidate for parallelisation. The inner loop is a loop over the files that are to be trained on. The original application is designed to preserve memory so on every iteration of the inner loop

in algorithm 1 the file is read into memory, and then transformed. This makes sure it is in a state that matches the current point of training. These files could be processed in parallel, this would give too rough of a granularity which is not desirable. If parallelising at the file level the program could have to wait on very large files to be processed in a serial manner. This would lead to poor utilisation of the available compute if the files are inbalanced in size regardless of the data to compute map.

Given the predicted poor load balance of the file level parallelism the operations the loop performs were investigated.

3.1 Byte pair tokeniser transform

To perform the tranformation there are two nested loops as per the following algorithm

Algorithm 2 Transform(vocabulary, buff)

\\Input: Array of vocab *vocabulary*, Array of symbols *buff*
\\Output: Array of symbols *buff* with byte pairs replaced

```
1: for i ← 0 to len(vocabulary) do
2:     for j ← 0 to len(buff) do
3:         if buff[i] == SKIP_TOKEN then
4:             continue
5:         end if
6:         if buff[i] == vocabulary[j].b1 then
7:             p ← 1
8:             while j + p < len(buff) do
9:                 if buff[j + p] ≠ SKIP_TOKEN then
10:                    break
11:                end if
12:                p ← p + 1
13:            end while
14:            if buff[i + 1] == vocabulary[j].b2 then
15:                buff[i] ← vocabulary[j].rep
16:                buff[i + 1] ← SKIP_TOKEN
17:            end if
18:            j ← j + p
19:        end if
20:    end for
21: end for
```

From the algorithm 2 it can be seen that the inner loop and outer loops both have flow and anti dependences. This makes the algorithm unsuitable for parallelisation. This means for this algorithm applying it to many files in parallel is the only suitable option.

3.2 Counting byte pairs

The algorithm for counting byte pairs is as follows

Algorithm 3 CountPairs(buff)

\\Input: Array of symbols *buff*
\\Output: Array of array of numbers *pairCounts*

```
1: pairCounts ← {}
2: for i ← 0 to len(buff) do
3:     if buff[i] == SKIP_TOKEN then
4:         continue
5:     end if
6:     p ← 1
7:     while i + p < len(buff) do
8:         if buff[i + p] ≠ SKIP_TOKEN then
9:             break
10:        end if
11:        p ← p + 1
12:    end while
13:    if i + p == len(buff) then
14:        return
15:    end if
16:    b1 ← buff[i]
17:    b2 ← buff[i + p]
18:    pair[b1][b2] ← pair[b1][b2] + 1
19: end for
```

In the algorithm it can be seen that there is a flow and anti dependence in the loop. As the only dependence is the pair array though it is classed as a reduction operation. The reduction operation in this algorithm is non trivial as when indexing the pair there are possible branches and memory allocations that could be performed. This makes OpenMP's reduction clause unsuitable for this operation. This would require a custom reduction operation to be implemented.

3.3 The Clean Slate

If the application is written from scratch and the parallelism available in the existing application is not binding then through significant refactoring the application could be made to be more parallelisable.

3.3.1 Dealing with data in chunks

It was discussed earlier that the file level parallelism would be too rough of a granularity to be useful. However if "files" were to be replaced with "chunks" then the granularity would be more fine grained. The chunk can have a size that is controlled by the programmer and can be adjusted to suit the data being processed.

The tokenisation step has to be performed on a file in its entirety as the byte pairs are not guaranteed to not span the chunk boundary. This means the tokenisation step has to be performed on each iteration, before creating the chunked views of the data that can have the byte pair counting performed on them.

This improves load balance as the chunks can be distributed to the threads in a more balanced manner, however the tokenisation step could still potentially be poorly balanced. This lead to the choice behind the methods of mapping the computation to the processors.

4 Mapping computation to processors

A number of tools were considered for the parallelisation of the application. The first tool considered was OpenMP. OpenMP could be used to parallelise the loops using the `#pragma omp parallel for` directive. This would greatly simplify the implementation process, however it would also leave performance on the table.

As found in the GCC implementation of OpenMP the work is divided to threads in an unequal manner**??**. As 1/n of the iterations are provided to the next free thread, the first thread ends up with most of the work. This is not ideal, particularly if the file paths are delivered to the program in a descending order of file size. This would lead to many threads stalling while the program waits on all threads to finish.

The next tool considered was POSIX pthreads. This would allow for a more fine grained control over the work division. A thread pool construct was built from scratch to manage the threads. The thread pool took a function pointer and a void pointer to the function argument from a hand made thread safe FIFO queue implementation. When a thread was free it pulled the next task from queue. This ensured that if a thread was free it would always be able to request more work.

This meant that if there was no requirement for synchronisation between different tasks then the next task could be added to the queue so the pool could initiate work on it without delaying for synchronisation.

4.1 Queue class implementation

The queue was implemented as a circular buffer, which was an array of pointers with a mutex and two semaphores. One semaphore signaled when the queue had space to add a new element, and the other signaled when the queue was non empty. The mutex was used to protect the indicies of the circular buffer from being modified by multiple threads at once.

The implementation of the queue resulted in a very small memory footprint. This meant that the queue was fast to access and modify.

4.2 ThreadPool class implementation

The threadpool was implemented as a class that took a number of desired threads in the constructor and launched each thread to a function in the class. This `ThreadPool::start(void *arg)` function allowed the thread to pull tasks from the threadsafe queue and execute them.

The class also implemented a destructor to join all the threads and free the memory used by the threads, abiding by the RAII principle.

5 Timing and profiling

6 Testing of logical equivalence

7 Tools used

To compile the parallel application GCC-12 was used with the -O3, -fopenmp, and -lpthread flags. When debugging the application the *strace* tool was particularly helpful. This allowed insight into the system calls that the application was making. This was useful to determine why particular threads were going to sleep or not grabbing work from the queue. *-fsanatize=address* was also invaluable in finding memory related bugs in the program. There were instances where race conditions occured, particularly use after free bugs due to destructors being called on data before threads had finished using it.

Intel Advisor was then used to profile the application. The roofline analysis was particularly useful to determine which loops were worth optimising further and which were not.

8 Process and Problems encountered

9 Explaination of the code

10 Learnings

11 References

https://github.com/gcc-mirror/gcc/blob/1cb6c2eb3b8361d850be8e8270c597270a1a7967/libgomp/iter.c