Lecture 20: High Performance Computing Case Study

COMP26020 Part 1 (C) Lecture Notes

Pierre Olivier

These notes summarise the important points mentioned in the lectures. They are supposed to be a help for revising and not a way to avoid attending the live lectures and watching the videos. In other words, live lectures and videos may include examinable content that is not present in these notes.

The slides for this lecture are available here:
https://olivierpierre.github.io/comp26020-lectures/20-hpc-case-study.

Videos and recordings of live sessions can be found on the video portal: https://video.manchester.ac.uk/lectures.

In this lecture we discuss a first case study of the application of C: High Performance Computing (HPC).

C in HPC

C is extensively used in HPC because it is **fast**. This speed is due to many reasons. Contrary to more high-level programming languages (e.g. Python):

- C is *close to the hardware*, there are few layers to traverse when calling system functions. With Python your script need to run through the interpreter. With Java, things are similar with the JVM: these additional layers in high-level languages introduce many overheads compared to low-level ones such as C.
- Another source of overheads in high-level languages is automatic memory protection and management. For example languages such as Java or Python have a garbage collector, that automatically takes care of dynamic memory management tasks. To enforce memory safety these languages also perform bounds checking e.g. when indexing an array. In C memory management is manual: there is *no garbage collector*, and there is also *no memory protection*. Although this creates more burden for the programmer, C avoids the related overheads with the goal of execution speed.
- Another advantage of C is that it is very easy to *integrate with assembly*, which sometimes needs to be used to optimise specific operation for performance.
- And last but not least, C **gives the programmer control over the data memory layout**. We discuss this particular aspect in this lecture, showing how useful it can be when optimising a program for performance.

Memory Hierarchy 101

Here is a quick recap about a subset of the memory hierarchy present on all computers. The CPU has local memory in the form of registers, there are very few of these, between 10 and 30, and accessing them is very fast: data present in registers is instantaneously available for computations. Main memory stores most of the program data because it is large so bringing data from memory to registers for computations is slow: it can take hundreds of CPU clock cycles So we have this intermediary layer named the cache, it is made of a relatively small amount of fast memory (faster than the RAM, slower than registers) that holds a subset of the program's data. Data is loaded in the cache from memory on demand when requested by the CPU, and the unit of transfer between memory and the cache is a *cache line*, generally 64 contiguous bytes, to exploit the principle of locality. So, to get the best performance possible for a program – and this is the goal in HPC – it is important to fit as much as possible of the program's data in the cache.

Cache-unfriendly Program Example

Let's discuss an example. Consider the following program:

```c
typedef struct {
    char c[60];
    int i;
    double d;
} my_struct;

#define N 100000000
my_struct array[N];

int main(int argc, char **argv) {
  struct timeval start, stop, res;
  my_struct s;

  gettimeofday(&start, NULL);

  /* Randomly access N elements */
  for(int i=0; i<N; i++)
      memcpy(&s, &array[rand()%N], sizeof(my_struct));

  gettimeofday(&stop, NULL);
  timersub(&stop, &start, &res);
  printf("%ld.%06ld\n", res.tv_sec, res.tv_usec);

  return 0;
}
```

This program accesses a large array of data structures. The data structure contains a string member, an `int` member, and a `double` member. In a relatively long loop the program uses `memcpy` to read from the array a given member from a random index. We use `gettimeofday` to measure the execution time of the loop. This program is extremely memory intensive, there is not much computation: it is memory-bound by the memory copy operations.

It turns out that this program is not very cache friendly. If we look at the size of one member of the array, that is the sizeof `my_struct`, we can see that it's $60 + 4 + 8 = 72$ bytes. This is slightly larger than a cache line which is 64 bytes. So most calls to `memcpy` in the program's inner loop will require to fetch 2 cache lines from memory to be brought into the cache.

Fitting Data into the Cache

We can fix that issue by updating the structure and array definitions as follows:

```c
typedef struct {
    char c[52]; // down from 60, we have 52 + 4 + 8 == 64 bytes i.e. a cache line
    int i;
    double d;
} my_struct;

my_struct array[N] __attribute__ ((aligned(64)));  /* force alignment of the array itself */
```

We reduce the size of the structure to be equal to that of a cache line. More precisely we reduced slightly the size of the string so that the size of one instance of the data structure is 64 bytes. Then we use the special keyword `attribute` to make sure that the array is aligned to a cache line. In effect this will make that each member of the array is fully contained in a cache line.

On the computer this code was tested, this effectively speeds up the program by about 25%. Such optimisations are possible in C because the language gives the programmer a large area of control over the program's memory layout, in other word one can control the placement and sizes of data structures in memory.