Course notes for High Performance Programming and Systems

Kenneth Skovhede, Troels Henriksen

December 9, 2020

0.1 Introduction

These notes are written to supplement the textbooks in the course *High Performance Programming and Systems*. Consider them terminally in-progress. These notes are not a textbook, do not cover the entire curriculum, and might not be comprehensible if isolated from the course and its other teaching activities.

Contents

Chapter 1

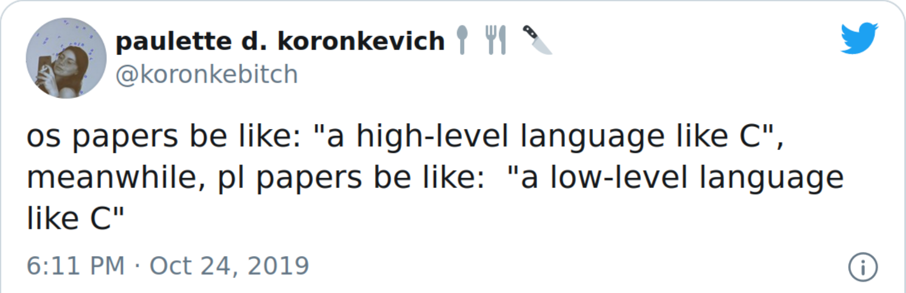
Compiled and Interpreted Languages

A computer can directly execute only machine code, consisting of raw numeric data. Machine code can be written by humans, but we usually use symbolic *assembly languages* to make it more approachable. However, even when using an assembly language, this form of programming is very tedious. This is because assembly languages are (almost) a transparent layer of syntax on top of the raw machine code, and the machine code has been designed to be efficient to *execute*, not to be a pleasant programming experience. Specifically, we are programming at a very low level of abstraction when we use assembly languages, and with no good ability to build new abstractions. In practice, almost all progamming is conducted in *high-level languages*.

1.1 Low-level and High-Level Languages

For the purpose of this chapter, a high-level programming language is a language that is designed not to directly represent the capabilities and details of some machine, but rather to *abstract* the mechanical details, in order to make programming simpler. However, we should note that "high-level" is a spectrum. In general, the meaning of the term "high-level programming language" depends on the speaker and the context (fig. 1.1). The pioneering computer scientist Alan Perlis said: *"A programming language is low-level when its programs require attention to the irrelevant"*. During the course you will gain familiarity with the programming language C, which *definitely* requires you to pay attention to things that are often considered irrelevant, which makes it low-level in Perlis's eyes. However, we will see that the control offered by C provides some capabilities, mostly the ability to *tune* our code for high performance, that are for some problems *not* irrelevant. The term *mid-level programming language* might be a good description of C, as it fills a niche between low-level assembly languages, and high-level languages such as Python and F#.

Generally speaking, low-level languages tend to be more *difficult* to program



paulette d. koronkevich 🥄🍴🔪
@koronkebitch

os papers be like: "a high-level language like C", meanwhile, pl papers be like: "a low-level language like C"

6:11 PM · Oct 24, 2019

Figure 1.1: A remark on the clarity of terms in computer science.

in, while offering greater potential *performance* (i.e. they are faster). Higher-level languages are much easier to program in, but run slower and require more machine resources (e.g. memory). Given the speed of modern computers, this is a price we are often willing to pay—especially in the common case where the slowest part of our program is waiting for information from disk or network. Do not make the mistake of assuming that a program written in a low-level language is *always* faster than one written in a high-level language. Choice of algorithm is often more important than choice of language. Further, some high-level languages are designed specifically to execute very quickly. But there is no free lunch: these languages tend to make tradeoffs in other areas. There is no objective measure of where a language lies on the scale of "level-ness", so while a statement such as *"Python is more high-level than C"* is unlikely to raise any objections, it is usually pointless to try to rank very similar languages on this spectrum.

1.2 Compilers and Interpreters

As the computer natively understands only its machine code, other languages must be *translated* to machine code in order to run. For historical reasons, this is called *compilation*. We say that a compiler takes as input a file with a *source program*, and produces a file containing an executable *machine program* that can be run directly. This is a very simplified model, for the following reasons:

1. Strictly speaking, a compiler does not have to produce machine code. A compiler can also produce code in a different high level languag. For example, with the rise of browsers, it has become common to write compilers that produce Javascript code.

2. The machine program normally cannot be *directly* executed, as modern systems have many layers of abstraction on top of the processor. While the compiler does produce machine code, it is usually stored in a special file format that is understood by the *operating system*, which is responsible for making the machine code available to the processor.

3. The actual compiler contains many internal steps. Further, large programs are typically not compiled all at once, but rather in chunks. Typically, each *source file* is compiled to one *object file*, which are finally *linked* to form an executable program.

While compilers are a fascinating subject in their own right, we will discuss them only at a practical level. For a more in-depth treatment, you are encouraged to read a book such as Torben Mogensen's *Basics of Compiler Design*[1] for more information.

In contrast, an *interpreter* is a program that executes code directly, without first translating it. The interpreter can be a compiled program, or itself be interpreted. At the bottom level, we always have a CPU executing machine code, but there is no fundamental limit to how many layers of interpreters we can build on top. However, the most common case is that the interpreter is a machine code program, typically produced by a compiler. For example, Python is an interpreted language, and the `python` interpreter program used by most people, is written in C, and compiled to machine code.

Interpreters are generally easier to construct than compilers, especially for very dynamic languages, such as Python. The downside is that code that is interpreted generally runs much slower than machine code. This is called the *interpretive overhead*. When a C compiler encounters an integer expression `x + y`, then this can likely be translated to a single machine code instruction (possibly preceded by instructions to read `x` and `y` from memory. In contrast, whenever an interpreter encounters this expression, it has to analyse it and figure out what is supposed to happen (integer addition), and then dispatch to an implementation of that operation. This is usually at least an order of magnitude slower than actually doing the work. This means that interpreted languages are usually slower than compiled languages. Many programs spend most of their time waiting for user input, for a network request, or for data from the file system. Such programs are not greatly affected by interpretive overhead.

As an example of interpretive overhead, let us try writing programs for investigating the Collatz conjecture. The Collatz conjecture states that if we repeatedly apply the function

$$f(n) = \left\{ \begin{array}{ll} \frac{n}{2} & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{array} \right\}$$

to some initial number greater than 1, then we will eventually reach 1. To investigate this function, the Python program `collatz.py` in listing 1.1 takes an initial k from the user, then for every $1 \leq n < k$ prints out n followed by the number of iterations of the function it takes to reach 1.

[1] `http://hjemmesider.diku.dk/~torbenm/Basics/`

Listing 1.1: A Python program for investigating the Collatz conjecture.

```
import sys

def collatz(n):
    i = 0
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        i = i + 1
    return i

k = int(sys.argv[1])
for n in range(1, k):
    print(n, collatz(n))
```

In a Unix shell we can time the program for $k = 100000$ as follows, where we explicitly ignore the output[2]:

```
$ time python3 ./collatz.py 100000 >/dev/null

real    0m1.368s
user    0m1.361s
sys     0m0.007s
```

The `real` measurement tells us that the program took a little more than $1.3s$ to run in the real world (we'll talk about the meaning of `user` and `sys` later in the course).

Now let us consider the same program, but written in C, which we call `collatz.c`, and is shown in listing 1.2.

C is a compiled language, so we have to compile `collatz.c`:

```
$ gcc collatz.c -o collatz
```

And then we can run it:

```
$ time ./collatz 100000 >/dev/null

real    0m0.032s
user    0m0.030s
sys     0m0.002s
```

[2] This is a very naive way of timing program—it's adequate for programs that run for a relatively long time, but later we will have to discuss better ways to measure performance. In particular, it ignores the overhead of starting up the Python interpreter, and it is sensitive to noise, because we only to a single run.

Listing 1.2: A C program for investigating the Collatz conjecture.

```
#include <stdio.h>
#include <stdlib.h>

int collatz(int n) {
  int i = 0;
  while (n != 1) {
    if (n % 2 == 0) {
      n = n / 2;
    } else {
      n = 3 * n + 1;
    }
    i++;
  }
  return i;
}

int main(int argc, char** argv) {
  int k = atoi(argv[1]);
  for (int n = 1; n < k; n++) {
    printf("%d␣%d\n", n, collatz(n));
  }
}
```

Only $0.032s$! This means that our C program is

$$\frac{1.368}{0.032} = 42.75$$

times faster than the Python program. This is not unexpected. The ease of use of interpreted languages comes at a significant overhead.

1.2.1 Advantages of interpreters

People implement interpreters because they are easy to construct, especially for advanced or dynamic languages, and because they are easier to work with. For example, when we are compiling a program to machine code, then the compiler throws away a lot of information, which makes it difficult to relate the generated machine code with the code originally written by the compiler. This makes debugging harder, because the connection between what the machine physically *does*, and what the programmer *wrote*, is more complicated. In contrast, an interpreter more or less executes the program as written by the user, so when things go wrong, it is easier to explain where in the source code the problem occurs.

In practice, to help with debugging, good compilers can generate significant amounts of extra information in order to let special *debugger* programs connect the generated machine code with the original source code. However, this does tend to affect the quality of the generated code.

Another typical advantage of interpreters is that they are straightforwardly *portable*. When writing a compiler that generates machine code, we must explicitly write a code generator every CPU architecture we wish to target. An interpreter can be written once in a portable programming language (say, C), and then compiled to any architecture for which we have a C compiler (which is essentially all of them).

As a rule of thumb, very high-level languages tend to be interpreted, and low-level languages are almost always interpreted. In practice, things are not always so clear cut, and *any* language can in principle be compiled—it may just be very difficult for some languages.

1.2.2 Blurring the lines

Very few production languages are *pure* interpreters, in the sense that they do no processing of the source program before executing it. Even Python, which is our main example of an interpreted language, does in fact compile Python source code to Python *bytecode*, which is a kind of invented machine code that is then interpreted by the Python *virtual machine*, which is an interpreter written in C. We can in fact ask Python to show us the bytecode corresponding to a function:

```
>>> import dis
>>> def add(a,b,c):
...     return a + b + c
...
>>> dis.dis(add)
  2           0 LOAD_FAST                0 (a)
              2 LOAD_FAST                1 (b)
              4 BINARY_ADD
              6 LOAD_FAST                2 (c)
              8 BINARY_ADD
             10 RETURN_VALUE
```

This is not machine code for any processor that has ever been physically constructed, but rather an invented machine code that is interpreted by Python's bytecode interpreter. This is a common design because it is faster than interpreting raw Python source code, but it is still much slower than true machine code.

JIT Compilation

An even more advanced implementation technique is *just-in-time* (JIT) compilation, which is notably used for languages such as C#, F# and JavaScript.

Here, the source program is first compiled to some kind of intermediary bytecode, but this bytecode is then further compiled *at run-time* to actual machine code. The technique is called just-in-time compilation because the final compilation typically occurs on the user's own machine, immediately prior to the program running.

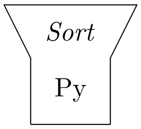
The main advantage of JIT compilation is that programs run much faster than when interpreting bytecode, because we ultimately do end up executing a machine code version of the program. Because JIT compilation takes place while the program is running, it is also able to measure the actual run-time behaviour of the program and tailor the code generation to the actual data encountered in use. This is useful for hightly dynamic languages, where traditional *ahead-of-time* (AOT) compilers have difficulty getting good results. In theory, a JIT compiler can always be *at least as good* as an AOT compiler, but in practice, AOT compilers tend to generate better code, as they can afford to spend more time on compilation. In practice, JIT compilers are only used to compute those parts of the program that are "hot" (where a lot of time is spent), and an interpreter is used for the rest. This tends to works well in practice, due to the maxim that 80% of the run-time is spent in 20% of the code. An AOT compiler will not know which 20% of the code is actually hot, and so much dedicate equal effort to every part, while a JIT compiler can measure the run-time behaviour of the program, and see where it is worth putting in extra effort.

The main downside of JIT compilation is that it is difficult to implement. It has been claimed that AOT compilers are 10× as difficult to write as interpreters, and JIT compilers are 10× as difficult to write as AOT compilers.

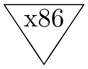
1.3 Tombstone Diagrams

Interpreters and compilers allow us to consider programs as input and output of other programs. That is, they are *data*. *Tombstone diagrams* (sometimes called *T-diagrams*) are a visual notation that lets us describe how a program is translated between different languages (*compiled*), and when execution takes place (either through a software interpreter or a hardware processor). They are not a completely formal notation, nor can they express every kind of program transformation, but they are useful for gaining an appreciation of the big picture.

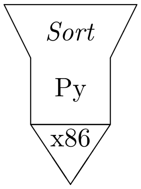
As the most fundamental concepts, we have programs, which are written in some language. Suppose we have a sorting program written in Python, which we draw as follows:



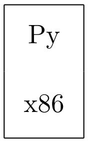
This is an incomplete diagram, since it contains programs we have not described how to execute. A machine that executes some language, say x86 machine code is illustrated as a downward-pointing triangle:



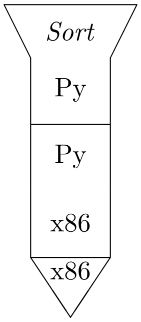
We can say that the Python program is executed on this machine, by stacking them:



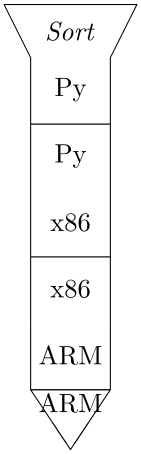
But this diagram is *wrong* — we are saying that a program written in Python is running on a machine that executes only x86. When putting together a tombstone diagram, we must ensure that the languages of the components match. While on paper, we can just assume a Python machine, this is not very realistic. Instead, we use an interpreter for Python, written in x86, written like this:



We can then stack the Python program on top of the interpreter, which we then stack on top of the machine:

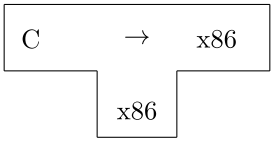


But maybe we are actually running on an ARM machine (as can be found in most phones), but still only have a Python interpreter in x86. As long as we have an x86 interpreter written in ARM machine code, this is no problem:

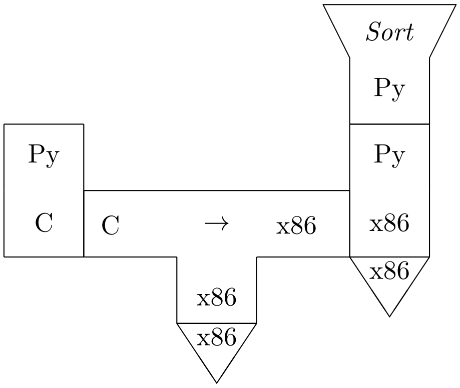


There is no limit to how high we can stack interpreters. All that matters is that at the end, we have either a machine that can run the implementation language of the bottommost interpreter. Of course, in practice, each level of interpretation adds overhead, so while tombstone diagrams show what is *possible*, they do not necessarily show what is a good idea. Tall interpreter stacks mostly occur in retrocomputing or data archaeology, where we are simulating otherwise dead hardware.

The diagrams above are a bit misleading, because the Python interpreter is not actually written in machine code—it is written in C, which is then translated by a compiler. With a tombstone diagram, a compiler from C to x86, where the compiler is itself also written in x86, is illustrated as follows:

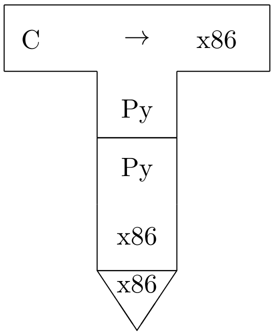


We can now put together a full diagram showing how the Python interpreter is translated from C to x86, and then used to run a Python program:



For a diagram to be valid, every program, interpreter, or compiler, must either be stacked on top of an interpreter or machine, or must be to the left of a compiler, as with the Python interpreter above.

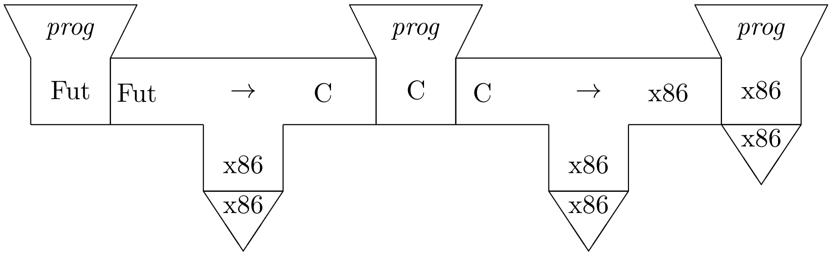
Compilers are also just programs, and must either be executed directly by an appropriate machine, or interpreted. For example, the following diagram shows how to run a C compiler in Python, on top of a Python interpreter in x86 machine code:



How the Python interpreter has been obtained, whether written by hand or compiled from another language, is not visible in the diagram.

We can also use diagrams to show compilation pipelines that chain multiple compilers. For example, programs written in the Futhark[3] programming language are typically compiled first to C, and then uses a C compiler to generate machine code, which we can then finally run:

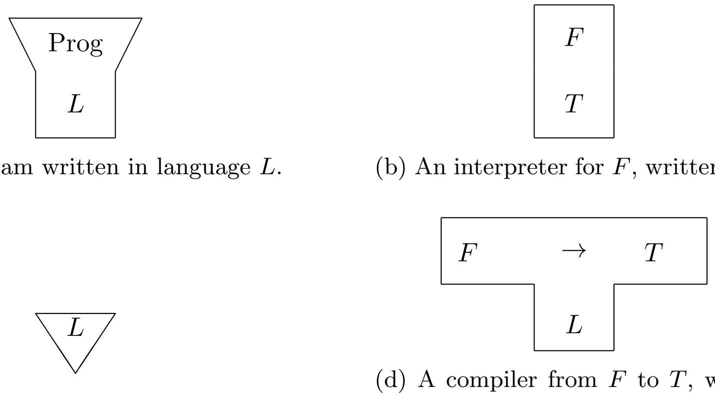
[3] `https://futhark-lang.org`



Many compilers have multiple internal steps—for example, a C compiler does not usually generate machine code directly, but rather generates symbolic assembly code, which an *assembler* then translates to binary machine code. Typically tombstone diagrams do not include such details, but we can include them if we wish, such as with the Futhark compiler above.

Tombstone diagrams can get awkward in complex cases (sometimes there will be no room!), but they can be a useful illustration of complex setups of compilers and interpreters. Also, if we loosen the definition of "machine" to include "operating systems", then we can use these diagrams to show how we can emulate Windows or DOS programs on a GNU/Linux system.

Tombstone diagrams hide many details that we normally consider important. For example, a JIT compiler is simply considered an interpreter in a tombstone diagram, since that is how it appears to the outside. Also, tombstone diagrams cannot easily express programs written in multiple languages, like the example shown in section 1.4. Always be aware that tombstone diagrams are a very high-level visualisation. In practice, such diagrams are mostly used for describing *bootstrapping* processes, by which we make compilers available on new machines. The tombstone diagram components are summarised in fig. 1.2.



(a) A program written in language L.

(b) An interpreter for F, written in T.

(c) A machine that runs L programs.

(d) A compiler from F to T, written in L.

Figure 1.2: A summary of tombstone diagram building blocks.

1.4 Combining Python and C

As discussed above, interpreted languages are typically substantially slower than compiled languages, especially for languages with high *computational intensity*. By this term, we mean how much of the execution time is spent directly executing program code, and how much is spent waiting for data (e.g. user input or network data). For programs with low computational intensity, an interpreted language like Python is an excellent choice, as the interpretive overhead has little impact. However, Python is also very widely used for computationally heavy programs, such as data analysis. Do we just accept that these programs are much slower than a corresponding program written in C? Not exactly. Instead, we use high-performance languages, such as C, to write *computational kernels* in the form of C functions. These C functions can then be called from Python using a so-called *foreign function interface* (FFI).

As a simple example, let us consider the `collatz.c` program from listing 1.2. Instead of compiling the C program to an executable, we compile it to a so-called *shared library*, which allows it to be loaded by Python[4]:

```
$ gcc collatz.c -fPIC -shared -o libcollatz.so
```

We can now write a Python program that uses the `ctypes` library to access the compiled in the `libcollatz.so` library, and call the `collatz` function we wrote in C:

Listing 1.3: A Python program that uses a C implementation of `collatz`.

```
import ctypes
import sys

c_lib = ctypes.CDLL('./libcollatz.so')

k = int(sys.argv[1])
for n in range(1, k):
    print(n, c_lib.collatz(n))
```

Let's time it as before:

```
$ time python3 ./collatz-ffi.py 100000 >/dev/null

real    0m0.165s
user    0m0.163s
sys     0m0.003s
```

The pure Python program ran in $1.3s$, the pure C in $0.032s$, and this mixture in $0.165s$ - significantly faster than Python, but slower than C by itself. The difference is mostly down to the implicit work required to converte Python

[4] Don't worry about the details of the command line options here—the technical details are less important than the overall concept.

values to C values when calling `c_lib.collatz`. The overhead is particularly acute for this program, because each call to `collatz` does relatively little work.

While this example is very simple, the basic idea is *fundamental* to Python's current status as perhaps the most popular language for working data scientists and students. Ubiquitous libraries such as NumPy and SciPy have their computational core written in high-performance C or Fortran, which is then exposed in a user-friendly way through Python functions and objects. While a program that uses NumPy is certainly much slower than a tightly optimised C program, it is *much* faster than a pure Python program would be, and *far* easier to write than a corresponding C program.

Chapter 2

Data Layout

One of the things that makes C a difficult programming language is that it does not provide many built-in data types, and provides poor support for making it convenient to work with new data types. In particular, C has notoriously poor support for multi-dimensional arrays. Given that multi-dimensional arrays are perhaps the single most important data structure for scientific computing, this is not good. In this chapter we will look at how we *encode* mathematical objects such as matrices (two-dimensional arrays) with the tools that C makes available to us. One key point is that there are often multiple ways to represent the same object, with different pros and cons, depending on the situation.

2.1 Arrays in C

At the surface level, C does support arrays. We can declare a $n \times m$ array as

```
double A[n][m];
```

and then use the fairly straightforward `A[i][j]` syntax to read a given element. However, C's arrays are a second-class language construct in many ways:

- They decay to pointers in many situations.
- They cannot be passed to a function without "losing" their size.
- They cannot be returned from a function at all.

In practice, we tend to only use language-level arrays in very simple cases, where the sizes are statically known, and they are not passed to or from functions. For general-purpose usage, we instead build our own representation of multi-dimensional arrays, using C's support for *pointers* and *dynamic allocation*. Since actual machine memory is essentially a single-dimensional array, working with multi-dimensional arrays in C really just requires us to answer one central question:

How do we map a multi-dimensional index to a single-dimensional index?

Or to put it another way, representing a d-dimensional array in C requires us to define a bijective[1] *index function*

$$I : \mathbb{N}^d \to \mathbb{N} \tag{2.1}$$

The index function maps from our (mathematical, conceptual) multi-dimensional space to the one-dimensional memory space offered by an actual computer. This is sometimes also called *unranking*, although this is strictly speaking a more general term from combinatorics.

As an example, suppose we wish to represent the following 3×4 matrix in memory:

$$\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{pmatrix} \tag{2.2}$$

We can do this in any baroque way we wish, but the two most common representations are:

Row-major order, where elements of each *row* are contiguous in memory:

11	12	13	14	21	22	23	24	31	32	33	34

with index function

$$(i, j) \mapsto i \times 4 + j$$

Column-major order, where elements of each *column* are contiguous in memory:

11	21	31	12	22	32	13	23	33	14	24	34

with index function

$$(i, j) \mapsto j \times 3 + i$$

The index functions are generalised on fig. 2.1. Note that the two representations contain the exact same values, so they encode the same mathematical object, but in different ways. The intuition for the row-major index function is that we first skip i rows ahead to get to the row of interest, then move j columns into the row.

Row-major order is used by default in most programming languages and libraries, but not universally so—the scientific language Fortran is famously column-major. The NumPy library for Python uses row-major by default (called `C` in Numpy), but one can explicitly ask for arrays in column-major order (called `F`), which is sometimes needed when exchanging data with systems that expect a different representation.

[1] A bijective function is a function between two sets that maps each element of each set to a distinct element of the other set.

$$(i, j) \mapsto i \times m + j \tag{2.3}$$

(a) Row-major indexing.

$$(i, j) \mapsto j \times n + i \tag{2.4}$$

(b) Column-major indexing.

Figure 2.1: Index functions for $n \times m$ arrays represented in row-major and column-major order. For an example of why computer scientists tend to prefer 0-indexing, try rewriting the above to work with 1-index arrays instead.

2.1.1 Implementation in C

Let's look at how to implement this in C. Let's say we wish to represent the matrix from eq. (2.2) in row-major order. Then we would write the following (assuming `n=3`, `m=4`):

```
int *A = malloc(n*m*sizeof(int));
A[0] = 11;
A[1] = 12;
...
A[11] = 34;
```

Note that even though we *conceptually* wish to represent a two-dimensional array, the actual C type is technically a single-dimensional array with 12 elements. If we when wish to index at position (i, j) we then use the expression `A[i*4+j]`.

Similarly, if we wished to use column-major order, we would program as follows:

```
int *A = malloc(n*m*sizeof(int));
A[0] = 11;
A[1] = 21;
...
A[11] = 34;
```

To C there is no difference—and there is no indication in the types what we intended. This makes it very easy to make mistakes.

Note also how it is on *us* to keep track of the sizes of the array—C is no help. Don't make the mistake of thinking that `sizeof(A)` will tell you how big this array is—while C will produce a number for you, it willindicate the size of a *pointer* (probably 8 on your machine).

Some C programmers like defining functions to help them generate the flat indexes when indexing arrays:

```
int idx2_rowmajor(int n, int m, int i, int j) {
  return i * m + j;
}

int idx2_colmajor(int n, int m, int i, int j) {
  return j * n + i;
}
```

Note how row-major indexing does not use the `n` parameter, and column-major indexing does not use `m`.

However, these functions do not on their own fully prevent us from making mistakes. Consider indexing the `A` array from before with the expression

```
A[idx2_rowmajor(n, m, 2, 5)].
```

Here we are trying to access index $(2, 5)$ in a 3×4 array—which is conceptually an out-of-bounds access. However, by the index function, this translates to the flat index $2 \times 3 + 5 = 11$, which is in-bounds for the 12-element array we use for our representation in C. This means that handy tools like `valgrind` will not even be able to detect our mistake—from C's point of view, we're doing nothing wrong! This like this make scientific computing in C a risky endeavour. We can protect ourselves by using helper functions like those above, and augment them with `assert` statements that check for problems:

```
int idx2_rowmajor(int n, int m, int i, int j) {
  assert(i >= 0 && i < n);
  assert(j >= 0 && j < m);
  return i * m + j;
}
```

We can still make mistakes, but at least now they will be noisy, rather than silently reading (or corrupting!) unintended data.

2.1.2 Size passing

With the previously discussed representation, a multidimensional array (e.g. a matrix) is just a pointer to the data, along with metadata about its size. The C language does not help us keep this metadata in sync with reality. When passing one of these arrays to a function, we must manually pass along the sizes, and we must get them right without much help from the compiler. For example, consider a function that sums each row of a (row-major) $n \times m$ array, saving the results to an n-element output array:

Listing 2.1: Summing the rows of a matrix.

```
void sumrows(int n, int m,
             const double *matrix, double *vector) {
  for (int i = 0; i < n; i++) {
    double sum = 0;
    for (int j = 0; j < m; j++) {
      sum += matrix[i*m+j];
    }
    vector[i] = sum;
  }
}
```

C gives us the raw building blocks of efficient computation, but we must put together the pieces ourselves. We protect ourselves by carefully documenting the data layout expected of the various functions. For the `sumrows` function above, we would document that `matrix` is expected to be a row-major array of size $n \times m$.

2.1.3 Slicing

In high-level languages like Python, we can use notation such as `A[i:j]` to extract a *slice* (a contiguous subsequence) of an array. No such syntactical niceties are available in C, but by using our knowledge of how arrays are physically laid out in memory, we can obtain similar effect in many cases.

Suppose `V` is a vector of `n` elements, and we wish to obtain a slice of the elements from index `i` to `j` (the latter exclusive). In Python, we would merely write `V[i:j]`. In C, we compute the size of the slice as

```
int m = j - i;
```

and then compute a pointer to the start of the slice:

```
double *slice = &V[i];
```

Now we can simply treat `slice` as an `m`-element array, which uses the same underlying storage as `V`—just as in Python.

Similarly, if `A` represents a matrix of size `n` by `m` in row-major order, then we can produce a vector representing the `i`th row as follows:

```
double *row = &A[i*m];
```

The restriction is that such slicing can only produce elements that are *contiguous* in memory. For example, we cannot easily extract a column of a row-major array, because the elements of a column are not contiguous in memory. If we wish to extract a column, then we have to allocate space and copy element-by-element, in a loop[2].

2.1.4 Even higher dimensions

The examples so far have focused on the two-dimensional case. However, the notion of row-major and column-major order generalises fine to higher dimensions. The key distinction is that in a row-major array, the *last* dimension is contiguous, while for a column-major array, the *first* dimension is contiguous. For a row-major array of shape $n_0 \times \cdots \times n_d$, the index function where p is a d-dimensional index point is

$$p \mapsto \sum_{0 \leq i < d} p_i \times \prod_{i<j<d} n_j \tag{2.5}$$

[2] There are more sophisticated array representations that use *strides* to allow array elements that are not contiguous in memory—NumPy uses these, but their representation are outside the scope of our course.

where p_i gets the ith coordinate of p, and the product of an empty series is 1.

Similarly, for a column-major array, the index function is

$$p \mapsto \sum_{0 \leq i < d} p_i \prod_{0 < j < i} n_j \tag{2.6}$$

We can also have more complex cases, such as a three-dimensional array where the two-dimensional “rows” are stored consecutively, but are individually column-major. Such constructions can be useful, but are beyond the scope of this course (and are a nightmare to implement).

Chapter 3

Networks

Although there are many kinds of computer networks created, the vast majority of network equipment is based on the TCP/IP stack, which we will cover in these notes. The course book covers networks from a programmer's perspective, inspecting details on how to use the C programming language to implement network applications.

In most (scientific) applications this is far too low-level. The course notes focus instead on explaining how the network is implemented and the properties that arise from the implementation. The focus is on explaining the properties that are most likely to impact scientific applications.

3.1 OSI layers

The OSI layers are a model that explains the implementation of the internet. Each of the layers in the OSI model relies on the previous layer and makes certain assumptions about it. Using the services provided by the underlying layer, each layer can add new services.

The OSI model is intended to describe many kinds of networks, but in the text here we map the concepts to the implementation on global internet and consider this the only implementation.

3.2 Physical layer

The physical layer is the lowest layer in the OSI model and is concerned with transmission of bits via various propagation media. The first networks were implemented as wired networks, propagating electrical signals through a shared copper wire. Modern implementations use radio signals, to implement services such as Wi-Fi and 3/4/5G.

The physical layer provides the service of sending and receiving bits. It is important to note that due to properties of the propagation medium, the timings, bandwidth, and error correction properties are different. Each of the layers above needs to accept this, such that the layers work similarly with different physical layers.

3.2.1 Sharing a medium

The most significant work done by the physical layer is the ability to transmit messages on a shared medium, without any out-of-band control mechanism. The primary method for solving this is the use of a multiple-access protocol. For both (wired) Ethernet and Wi-Fi, the protocol is the carrier-sense multiple-access collision, CSMA, protocol. The protocol requires that each host can "sense" if data is being sent, and refrains from sending data which would cause a collision, making the data impossible to read. Since there is no out-of-band communication, collisions will eventually happen when two hosts communicate. The CSMA protocol uses a concept of exponential back-off with random starting values to ensure that collisions are eventually resolved.

3.2.2 Sharing with radio signals

For radio-based networks, it is possible to have a situation where the base station (the Wi-Fi access point) is placed between two hosts, such that neither can sense the other host's signals. In this case the CSMA protocol is extended to use collision avoidance, as the detection is not always possible. The concept is explored in this animated video: `https://www.youtube.com/watch?v=iKn0GzF5-IU`.

3.2.3 Connecting: Hub

Since the physical layer is about transmitting signals, and a shared media is supported, the physical layer allows multiple machines to be joined. For Ethernet (wired) networks, a hub can be used to physically connect multiple devices. With a network connected by a hub, the entire bandwidth for the network is shared with all hosts, resulting in poor scalability. If a network comprises 100 hosts and is using 100 Mbps Ethernet, each host will reach less than 1 Mbps if all hosts attempt to communicate at full speed.

3.3 Link layer

With a physical layer that is capable of transmitting bits, the link layer provides the option for addressing a single host. The bits are *framed* to allow sending a number of bits. Since the physical layer is expected to be using a shared medium, the link layer assumes that all frames are broadcast to every host.

3.3.1 Addressing network cards: MAC

To solve the problem of sending a frame to a particular host, the link layer adds *media access control*, MAC, addresses. Every network device has a globally unique address that is typically burned into the device but can be changed in some cases. When a host wishes to communicate with another host, a frame is transmitted on the physical medium, containing the MAC address of the recipient and sender.

The format of the MAC address is using six dual-digit hexadecimal numbers, for instance: `01:23:45:67:89:ab`. The special address `ff:ff:ff:ff:ff:ff` is used for broadcasting frames to all recipients.

3.3.2 Connecting: Switch

To make larger networks more efficient, a switch can be used in place of a hub. Where the hub operates on the physical layer, a switch operates on the link layer and forwards frames. When a switch is powered on, it has no knowledge of the network. Each frame it receives, it will broadcast to all other ports with a cable plugged in. For each package it receives, it will record the sender MAC address as well as the originating port. Since each MAC address is globally unique, the switch can use this simple scheme to learn where to send a frame and can avoid broadcasting to the entire network. Note that this feature works without any configuration, and even supports networks of networks, as the switch will allow mapping multiple MAC addresses to a single port. If a host is moved to another port, there will be a short period where the switch will forward frames incorrectly, until it sees a package from the new port.

3.3.3 Switches can increase available bandwidth

Once the network has levels, or just a large number of hosts, switches become a crucial component in ensuring performance of the network. More expensive switches can allow multiple parallel full-duplex data paths, such that multiple pairs of ports may use the full bandwidth. This allows the network to scale better with the addition of multiple hosts but may still have bottlenecks if there is cross-switch communication.

3.3.4 Transmission errors

Since the physical layer *may* have transmission errors, the link frames typically include a simple checksum that the receiver verifies. If the checksum is incorrect, or any parts of the frame are invalid, the frame is *dropped* with no notification to the sender.

3.4 Network layer

With a layer that supports sending frames, the network layer adds routing and global addressing. While we *can* build larger networks with a switch, the idea of broadcasting becomes problematic for a global network.

3.4.1 Addressing hosts: IP address

Because the MAC address is fixed, it is not usable for global routing. Imagine that a core router on the internet would need to keep a list of all MAC addresses for all machines in Europe. The size and maintenance of such a list would be an almost impossible task.

To simplify routing, each host on the network layer has at least one IP address, used to transmit *packages*. The IP addresses are assigned in a hierarchical manner, such that a geographic region has a large range, which is then divided into smaller and smaller ranges. At the bottom is the internet service provider, ISP, who owns one or more ranges. From these ranges they assign one or more IP addresses to their customers.

The hierarchy is important when considering global-scale routing. Instead of having a core router that keeps track of all IPs for Europe, it can just know that one of the ports "eventually leads to Europe" and then assign a few IP ranges to that port.

3.4.2 IP, CIDR, networks and masks

The IP addresses are 32 bit numbers, and usually presented as four 8-bit decimal numbers, called the dotted-decimal notation: `123.211.8.111`. The routing works locally by comparing one IP address with another, using bitwise `XOR`. By `XOR`'ing two IP addresses, any bits that are the same will be zero. The subnet mask then defines what bits are ignored, so we can use bitwise `AND` on the result. After these two operations, the result will only be all zero bits if two IP addresses are on the same subnet.

As an example, consider the two IP addresses `192.168.0.8` and `192.168.0.37`. Given a subnet mask of `255.255.255.192`, we can apply the `XOR` operation to the two numbers and get `0.0.0.45`. When we apply the `AND` operation to the result we get `0.0.0.0`, meaning that the two hosts are on the same subnet.

As the subnet masks are always constructed with leading zeroes (i.e. it is not valid to use `255.0.255.0`), we can simply count the number of leading bits in the mask. This gives the classless interdomain routing notation, where we supply the IP address and number of bits in a compact notation: `192.168.0.0/26`. This notation is equivalent to supplying an IP address and a subnet mask, as we can convert trivially between the two.

3.4.3 Connecting networks: Router

If we consider a package sent from a home-user in Europe to a server in the USA, the package will initially be sent to the *uplink* port of the routers, until it reaches a router that can cross the Atlantic ocean. After it has crossed the Atlantic ocean, it will reach more and more specific IP ranges until it arrives at the destination. This system allows each router to know only general directions, simplified as "one port for away, multiple closer". As mentioned in the video, routing is often done with "longest prefix matching", where multiple rules are stored as a tuple of: `(destination port, IP pattern in CIDR)`. The router can then sort the rules by the number of fixed bits (i.e. the `/x` part of the CIDR address), and use the first forwarding rule where the bits match. This approach allows the router to store broad "general rules" and then selectively change smaller ranges. As the only operations required are `XOR` + `AND`, it can be performed efficiently in hardware.

3.4.4 A router is a host

The router device itself works on the network layer, such that it can read the IP address. It can be considered a specialized computer, because the first routers were simply ordinary computers with multiple network cards. Unlike the switch and hub, a router is visible on the network and is addressable with its own IP addresses.

To function correctly, a router needs to know what ranges its ports are connected to, which requires manual configuration. As the network can also change, due to links being created and removed, as well as traffic changes and fluctuating transfer costs, the routers need to be dynamically updated.

3.4.5 Updating routing tables

The dynamic updates are handled inside each owners' network, and across the networks using various protocols, constantly measuring capabilities, traffic load, and costs. One of the protocols for communicating routes between network owners is called the Border Gateway Protocol and is unfortunately not secure yet. Bad actors can incorrectly advertise short and routes, which causes the networks to start sending all traffic over a particular link, either for disruption or eavesdropping purposes. Occasionally this also happens due to human errors, leaving parts of the internet unreachable for periods of time.

For an overview of the layers until now, and the different components that connect them, there is an animated video here: `https://www.youtube.com/watch?v=1z0ULvg_pW8`.

3.5 Transport layer

With a network layer that is capable of transmitting a package from one host to another, the transport layer provides one protocol for doing just that: User Datagram Protocol, UDP.

3.5.1 Adding ports

A datagram in UDP adds only a single feature to the service provided by the network layer: ports. To allow multiple processes on a given host to use the network, UDP adds a 16-bit port number. When describing and address on the transport layer, the port number is often added to the IP address or hostname after a colon: `192.168.0.1:456`. The operating system kernel will use the IP address and port number to deliver packages to a particular process. However, since UDP uses the network layer, there is no acknowledgement of receipt and no signals if a package is lost. Because the routers update dynamically, it is also possible for packages to reach the destination via different routes, causing packages to arrive in a different order than they were sent.

UDP can be acceptable for some cases, for instance game updates or video streams, where we would rather loose a frame than have a stuttering video. But for many other cases, such as transferring a file or dataset, the UDP service is not useful.

3.5.2 Transmission control protocol: TCP

The Transmission Control Protocol, TCP, is the most widely used protocol and most often used with IP to form TCP/IP. The TCP protocol builds a reliable transfer stream on top of a the unreliable delivery provided by the network layer. The protocol itself is very robust, with understandable mechanics, but can require some trials to accept that it works in all cases.

3.5.3 Establishing a connection

Since the network layer does not tell us if a package has been delivered, the TCP protocol uses *acknowledgements*, ACKs, to report receipt of a package. Before a connection is establish, the client sends a special SYN message, and awaits an ACK, and sends an ACK. This exchange happens before any actual data is transmitted but allows for ”piggy backing” data on the last ACK message. When designing an application, it is important to know that this adds an overhead for each established connection, and thus connections should preferably be reused.

3.5.4 A simple stop-and-go TCP-like protocol

Once the connection is established, data can flow, but due to the network layers unreliability, we can receive packages out-of-order or lose them. For a simple protocol, we can just drop out-of-order packages, treating them as lost. The remaining problem has two cases: loss of package and loss of ACK. Since the sender cannot know which of these has occurred, it assumes the data is lost, and re-transmits it after a timeout has occurred while waiting for an ACK. If we prematurely hit a timeout, we will re-send a received package, but this is the same case as a lost ACK will produce.

The recipient can simply discard a package it has already received, so if we add a package number, called a sequence number[1], to the package, it is trivial for the recipient to know which packages are new and which are retransmits. It should be fairly simple to convince oneself that this works in all cases, in

[1] In the text and the video, we use a number pr. package. In the real TCP protocol, the sequence number counts bytes to support split packages.

the sense that the recipient will eventually get the package, and the sender will eventually get an ACK.

3.5.5 Improving bandwidth with latency hiding

However, due to the communication delay, we are waiting some of the time, instead of communicating. For even moderate communication delays, this results in poor utilization. To work around this, the TCP protocol allows multiple packages to be ”in-flight”, so the communication can occur with full bandwidth. This requires that the sender needs to keep track of which packages have gotten an ACK and which are still pending. We also need to keep copies of multiple packages, such that we can retransmit them if required. This does not change the way the simple stop-n-go protocol works, it simply adds a counter on the sender side. The choice of ”how many” is done with a ramp-up process, increasing until no improvements are seen, which further adds to the delay of new connections.

3.5.6 Minimizing retransmission

We could stop there, and be happy that it works, but if we get packages out-of-order it means not being able to send the ACK, and many in-flight packages needs to be retransmitted. This is solved in TCP by keeping a receive buffer, allowing packages to arrive in out-of-order. This does not change the protocol, except that the recipient needs to send an ACK, only when there are no ”holes” in the sequence of received packages. This is simplified a bit in TCP, where an ACK is interpreted as vouching for all data up until the sequence number. This means that lost ACK messages are usually not causing disturbance, as a new one arrives shortly afterwards. But it also makes it easier for the recipient to just send an ACK for the full no-holes sequence.

If we lose a package, the recipient will notice that it keeps getting packages with higher sequence numbers. Since it can only ACK the last one, it will keep doing so. The sender can then notice getting multiple ACKs (specifically: 3) for the same package, and guess that it needs to re-transmit the next package. This improvement makes it possible to transmit at full speed even with some package loss.

Hopefully you can convince yourself that the protocol works correctly in all cases, despite the performance enhancements.

3.5.7 Flow control

When designing and evaluating network performance, it is important to know the two complementary mechanisms that both end up throttling the sending of packages. The original throttling mechanism is called flow-control and works by having the recipient include the size of the receive buffer with each ACK message. The sender can monitor this value and reduce the sending rate if it notices that the remaining space is decreasing. Once the space is zero, the sender will wait for a timeout or an ACK with a non-zero receive buffer size. The timeout is a protection against the case where the ACK is lost.

3.5.8 Congestion control

The other mechanism is congestion control, which is a built-in protection against overflowing the network itself. While any one machine cannot hope to overflow the core routers in the network, many hosts working together can. What happened in the early days of the internet was that some routers were overwhelmed and started dropping packages. The hosts were using TCP and responded by retransmitting the lost packages, causing a build-up of lost packages, to the point where no connection was working. The TCP protocols are implemented in software, so the TCP implementations were gradually updated with congestion control additions. Unlike flow-control, there is no simple way of reporting the current load of all routers in the path. Instead, congestion control monitors the responses from the client, and makes a guess of the network state. Various implementations use different metrics, where the loss of packages was once seen as the right indicator. However, since package loss occurs *after* the congestion issues have started, this was later changed to measure the time between ACK messages. Once the routers start to get overloaded, the response time increases, and modern TCP implementations use this to throttle the sending speed.

3.5.9 Closing a connection

The layers on top of TCP can assume that packages have arrived and are delivered in-order. But how do we know if we have received all packages, and not lost the last one? The TCP implementation handles this by sending a package with the FIN bit set. Like other packages, the FIN package can be lost, so we need to get an ACK as well. Again, the ACK package may be lost, so we need another package, and so on. TCP has a pragmatic solution, where the side that wishes to close the connection will send FIN, wait for an ACK and then send a final ACK. The final ACK *may* be lost, in which case the recipient does not know if the sender has received the ACK. If this happens, TCP will keep the connection in a ”linger” state for a period before using a timeout and closing the connection. Even if this happens, both sides can be certain that all messages have been echanged.

3.5.10 Network Address Translation

In the original vision of the internet, all hosts were publicly addressable with their own IP address. Later that turned out to be a bad idea for security reasons, and for economic reasons. Each ISP has to purchase ranges of IP addresses and assign these to their customers. But some customers may have several devices, increasing the cost. Likewise, some customers and companies may like to have an internal network with printers and servers which is not exposed to the internet, but at the same time be able to access the internet.

The technique that was employed rely on the router being a computer itself, with an internal and external IP address. When a host sends a package destined for the external network, the router will pick a random unused port number and forward the package, using the routers external IP address and the randomly chosen port. This information is stored in a table inside the router, such that when a response package arrives that is destined for the particular port and external IP, the router will forward it to the internal network, using the original IP and port.

This operation is transparent to the hosts inside and outside the network, allowing ISP customers to have multiple internal hosts, sharing a single external IP address.

When compared to the original vision for the internet, the NAT approach breaks with the assumption that each host needs a unique IP. In practice this means that a NAT'ed host can only initiate connections, it cannot receive new connections (i.e., it cannot be the server, only the client).

In some cases, it might be desirable to use NAT, but still have a host act as a server. For this situation, most NAT capable routers allow pre-loading of static rules, for instance "external IP, port 80" should go to "10.0.0.4:1234". This is the same operation that happens automatically for outbound connections, but just always active.

Although NAT is not strictly a security feature, it does provide some protection against insecurely configured machines being directly accessible from the internet. That is unless your NAT capable router has Universal Plug-n-Play, UPNP, which allows any program on your machine to request insert a preloaded NAT rule, exposing everything from printers to webcams without the owners knowledge.

3.5.11 IPv6

In the above we have only considered IPv4, which is where the addresses are 32-bit. Despite the visions of giving every host their own IP address, the number of hosts in the world has long exceeded the available IPv4 numbers. Thanks mostly to NAT techniques, and a similar concept for ISPs called carrier-grade-NAT, this has not yet stopped the growth of the internet.

Before it became apparent that NAT would ease some of the growing pains, the IPv6 standard was accepted and ratified. With IPv6 there are now 128 bits for an address, essentially allowing so many hosts, that it is unnecessary to have ports or NAT anywhere.

Where the transport layer is implemented in software and easily updated, the network layer is embedded in devices with special-purpose chips. Changing these is costly and has so far dragged out for more than a decade.

The number of IPv6 enabled hosts and routers continue to increase, but there are still many devices with a physical chip that cannot upgrade to IPv4. Many of these are IoT devices, attached to an expensive TV, surveillance cam or refrigerator. As the devices work fine for the owners, there is virtually no incentives for replacing it, leaving us with a hybrid IPv4 and IPv6 network for some time to come.

New internet services would likely strive to use IPv4 addresses as there are plenty of ISP that only offer IPv4. If a company only has IPv6 servers, they would be inaccessible to a number of potential customers.

Currently the most promising upgrade seems to be, once again, relying on NAT to perform transparent IPv4 to IPv6 translations. This could allow the internet as a whole to switch to IPv6 while also being accessible to IPv4 users.

3.6 Application layers

With the transport layer providing in-order delivery guarantees for communicating between to processes, it opens up to a multitude of applications. The most popular one being the use of HTTP for serving web pages, but also many other services, including the network time protocol, which keeps your computer clock running accurately even though it is has low precision.

3.6.1 Domain Name System

One application that runs on top of the transport layer is the domain name system, DNS. It is essentially a global key-value store for organizing values belonging to a given domain name. In the simplest case it is responsible for mapping a name, such as `google.com` to an IP address. This makes it easier for humans to remember, but also allows the owner of the domain name to change which host IPs are returned, without needing to contact the users.

The design of the system is based on a hierarchical structure, where 13 logical servers replicate the root information. In practice these 13 servers are implemented on over 700 machines, geographically distributed over the entire globe.

The root nodes store the IP addresses of the top-level domain servers, where a top-level domain could be `.com`. A top-level domain server, naturally replicated on multiple hosts, keeps a list of the name servers for each domain ending with the top-level domain (i.e. the top-level server for `.com` keeps track of `twitter.com`, `google.com`, etc).

This means that each domain must run their own nameserver, but in practice, most nameservers are run by a set of registrars, where a small number of machines are responsible for thousands of domains.

The domain name server is the final step, containing all information related to a given domain name. This server can be queried to obtain IP addresses for all subdomains (i.e., `www.google.com`, `docs.google.com`) as well as the domain itself (i.e., `google.com`).

Apart from the IP addresses for hosts, the DNS records contain email servers, known as `MX` records, free text, called `TXT` records, and other domain related information.

To keep the load on DNS servers down, each host in the DNS system will cache the values it receives for a period. The exact period is also stored in the DNS records with a time-to-live, `TTL`, value expressed in seconds.

Chapter 4

OpenMP

Writing multi-threaded programs using the raw POSIX threads API is tedious and error-prone. It is also not portable, as POSIX threads is specific to Unix systems. Also, writing efficient multi-threaded code is difficult, as thread creation is relatively expensive, so we should ideally write our programs to have a fixed number of *worker threads* that are kept running in the background, and periodically assigned work by a scheduler. In many cases, particularly within scientific computing, we do not need the flexibility and low-level control of POSIX threads. We mostly wish to parallelise loops with iterations that are *independent*, meaning that they can be executed in any order without changing the result. For such programs we can use *OpenMP*. We will use only a small subset of OpenMP in this course.

4.1 Basic use of OpenMP

OpenMP is an extension to the C programming language[1] that allows the programmer to insert high-level *directives* that indicate when and how loops should be executed in parallel. The compiler and runtime system then takes care of low-level thread management. OpenMP uses the the *fork-join* model of parallel execution: a program starts with a single thread, the *master thread*, which runs sequentially until the first *parallel region* (such as a parallel loop) is encountered. At this point, the master thread creates a group of threads ("fork"[2]), which execute the loop. The master thread waits for all of them to finish ("join"), and then continues on sequentially. For example:

```
#pragma omp parallel for
for (int i=0; i<n; i++) {
  A[i] = A[i]*2;
}
```

[1] OpenMP is not C specific, and is also in wide use for Fortran.

[2] Note an unfortunate mix-up of nomenclature: this has nothing to do with the Unix notion of `fork()`, which creates *processes*, not *threads*

The **#pragma** omp parallel **for** line is an OpenMP directive that indicates that the iterations of the following for loop can be executed in parallel. If this loop is compiled with a compiler that supports OpenMP, the n partitions of the loop will be divided among some worker threads, which will then execute them in parallel.

The number of threads used can be controlled at run-time, and is usually not equal to the number of iterations in the parallel loop. This is because when the amount of work per iteration is small (as above), it would not be efficient to have one thread per iteration.

One important idea behind OpenMP is that to understand the semantics of a program, we can always remove the directives, and consider what the remaining sequential C program would compute. This is called the *sequential elision*. This is a great advantage over low-level multi-threaded programming.

When we ask OpenMP to parallelise a loop, we solemnly swear that the following **for** loop is actually parallel. If we break this vow then the parallel and sequential execution of the code will give different results; the API does not provide any guarantees about the absence of race-conditions. For example, the iterations of the following loop are not independent, yet OpenMP will not stop us from asking it to be executed in parallel:

```
sum = 0;
#pragma omp parallel for
for (int i=0; i<N; i++) {
  sum += A[i];
}
```

Since all threads executing the parallel region have access to the same data, it is easy to have accidental race conditions in OpenMP programs. In **??** we will look in detail at determining when it is safe to execute a loop in parallel, and how to transform loops so they become safe to execute in parallel.

4.1.1 Compiling and running OpenMP programs

To compile with support for OpenMP directives in gcc, pass the -fopenmp option to the compiler. The number of threads that are going to be used for parallel execution can be set by environment variable OMP_NUM_THREADS. For example,

```
$ export OMP_NUM_THREADS=8
```

sets the environment variable for the current shell session, such that any OpenMP we run will use eight threads. Determining the optimal number of threads to use for a given program on a particular machine is something of a black art. In practice, we just try a few different numbers and see what runs fastest.

For example, suppose the contrived program in listing 4.1 is stored in the file openmp-example.c. We can then compile as follows:

Listing 4.1: A very simple example of using OpenMP.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int n = 100000000;

  int *arr = malloc(n*sizeof(int));

#pragma omp parallel for
  for (int i = 0; i < n; i++) {
    arr[i] = 0;
  }

  free(arr);
}
```

```
$ gcc -o openmp-example openmp-example.c -fopenmp
```

And then run with various values of `OMP_NUM_THREADS` to investigate the impact of parallelisation:

```
$ time OMP_NUM_THREADS=1  ./openmp-example
real    0m0.124s
user    0m0.034s
sys     0m0.090s
$ time OMP_NUM_THREADS=2  ./openmp-example
real    0m0.076s
user    0m0.033s
sys     0m0.104s
$ time OMP_NUM_THREADS=4  ./openmp-example
real    0m0.054s
user    0m0.039s
sys     0m0.133s
$ time OMP_NUM_THREADS=8  ./openmp-example
real    0m0.046s
user    0m0.054s
sys     0m0.184s
```

Note how the *real* time drops as we use more threads—although it's not quite eight times as fast with eight threads as with one. This is likely because this contrived program does so little work compared to the amount of memory we are accessing.

Listing 4.2: OpenMP dot product with reduction clause.

```
double dotprod(int n, double *x, double *y) {
  double sum = 0;
#pragma omp parallel for reduction(+:sum)
  for (int i = 0; i < n; i++) {
    sum += x[i] * y[i];
  }
  return sum;
}
```

4.2 Reductions

A loop whose iterations are completely independent can be parallelised with the **#pragma** omp parallel **for** directive, as shown before. Another common case is when the iterations are *almost* independent, but they all update a single accumulator - for example, when summing the elements of an array. For such loops, OpenMP provides *reduction clauses*, as used to compute a dot product on listing 4.2.

Note that all iterations of the loop update the same `sum` variable. The reduction clause `reduction(+:sum)` that we added to the OpenMP directive indicates that this update is done with the + operator. The compiler will transform this loop such that each thread gets its own *private* copy of `sum`, which they then update independently. At the end, these per-thread results are then combined to obtain the final result.

Reduction clauses only immediately work with a small set of built-in binary operators: `+`, `*`, `-`, `&&`, `||`, `^`, `|`, `^`, `max`, and `min`. It is also possible to use user-defined functions, but this is beyond the scope of this text. The common property shared by these operators is that they are *associative*, and have a *neutral element*. By associativity, we mean that for some operator $\oplus$, we have

$$(x \oplus y) \oplus z = x \oplus (y \oplus z).$$

That is, the "order of evaluation" does not matter. This is what allows us to partition the iterations of a reduction loop between multiple threads, without changing the result. Note that subtraction and division is *not* associative—this is why OpenMP does not allow them in a reduction clause.

A neutral element $0_\oplus$ for some operator $\oplus$ is a "natural zero" that does not change the result of evaluation:

$$x \oplus 0_\oplus = 0_\oplus \oplus x = x$$

For example, if the operator is addition, then the neutral element is 0. If the operator is multiplication, then the neutral element is 1. OpenMP requires that the initial value of each reduction variable (`sum` for listing 4.2) is the neutral element of the operator.

4.2.1 Associativity of Floating Point Operations

Some of you might recall that addition and multiplication of floating-point numbers is *not* associative, due to roundoff errors. Yet we perform a reduction on **`double`** values in listing 4.2! How can that be valid? The short answer is that OpenMP allows us to shoot ourselves in the foot if we wish. In practice, floating-point operations are often *almost* associative, and we can get useful results by treating them as if they were. In particular, there is no reason to believe that a sequential left-to-right summation of floating-point numbers is going to be more numerically accurate than a parallelisation of that loop. This does mean that an OpenMP program that uses reductions on floating-point values might compute a different result than the original sequential program.

Chapter 5

Parallel Speedup and Scaling

While *concurrent programming* is often done to model the problem domain nicely (e.g. have a thread per connection to a web server), *parallel programming* is primarily concerned with speeding up our programs. This chapter will introduce nomenclature for talking about and comparing the performance of programs, and also discuss ways in which we can predict the potential performance advantage from parallelising a program.

5.1 Speedup

Suppose we are given some program and asked to speed it up. We then hack on it for a bit based on our knowledge of low-level programming. But how do we quantify our improvements? The standard approach to comparing the performance of two programs is by computing the *speedup* of one over the other.

5.1.1 Speedup in latency

The easiest way to quantify the performance of a program by itself is to run it and measure how long it takes. This is called program *latency* (often called *runtime*): how long from it starts until the result is ready? This is usually measured in *wall time*, because it corresponds to the real-world time we can measure with a clock on our wall. In contrast to this is *CPU time*, which is the total amount of time spent executing code on the CPUs we have available. When we parallelise a program, we decrease the wall time, but typically not the CPU time—16 CPUs that simultaneously run for 60 seconds equates 960 seconds of total CPU time, but will only have taken 60 seconds of wall time.

We usually have to put in effort to make sure that our time measurement is reliable. For example, we must make sure that we are measuring what we intend to measure—sometimes we do not wish to measure e.g. startup overhead, or loading data from files. It's also an easy mistake to make to measure CPU time rather than wall time, which will hide the advantage of parallelisation. Also, particularly with short-running programs, we must perform multiple measurements to average out random timing effects caused by random scheduling decisions taken by the operating system, or background tasks waking up and causing cache evictions.

Once we have reliable runtime measurements for both the original program and our modified program, we compare them by computing the *speedup*:

Definition 1 (Speedup in latency) *If T_1, T_2 are the runtimes of two programs P_1, P_2, then the* speedup in latency *of P_2 over P_1 is*

$$\frac{T_1}{T_2}$$

For example, if we have a sequential program that runs in $25s$ and we manage to write a parallel program that runs in $10s$ on our machine, then we compute the speedup of the parallel program as

$$\frac{25}{10} = 2.5$$

We would then say that the speedup we obtain is 2.5. Speedup is a dimensionless quantity, but it's common to write it with a tailing ×, as in 2.5×. The speedup formula can explain why programmers sometimes say "program A is twice as fast as B", when they really mean "program A runs in half the time as B"—they are talking about the speedup being 2.

5.1.2 Speedup in throughput

Latency speedup is useful for programs where the workload is *fixed*. But often we are in a situation where the workload is infinite, for example in a long-running server that constantly processes new requests. Here latency is only meaningful within a single request, and to quantify the performance of the entire system, it is more interesting to look at the *throughput* of how many requests per time unit can be processed. Measuring throughput also allows us to compare the performance of programs that operate on different data sets.

The throughput Q is computed simply as the *workload* W processed in some time-span T:

$$Q = \frac{W}{T}$$

How we measure the workload depends on the concrete program. For a web server, we would measure requests. For matrix multiplication, we might measure total number of input elements accessed. Once we have computed throughput, we can then compute the speedup.

Definition 2 (Speedup in throughput) *If Q_1, Q_2 are the throughputs of two programs P_1, P_2, then the* speedup in throughput *of P_2 over P_1 is*

$$\frac{T_2}{T_1}$$

For example, suppose we have a program P_1 that can sum a megabyte in $69\mu s$, and a program P_2 that can sum a gigabyte in $28,589\mu s$. Since the workloads are different, we cannot directly compare their latency, but we can compute the throughputs as follows:

$$\begin{aligned} Q_1 =& \frac{2^{10}B}{69\mu s} = 15196B/\mu s = 14.2GiB/s \\ Q_2 =& \frac{2^{30}}{28589} = 37558B/\mu s = 35.0GiB/s \end{aligned}$$

The speedup in throughput of P_2 over P_1 is

$$\frac{35.0GiB/s}{14.2GiB/s} = 2.46$$

Note that while lower numbers are better for latency, higher numbers are better for throughput. In both cases, a higher speedup is better.

5.2 Amdahl's Law

Before we start on the often significant task of parallelising a program, or using a larger and more parallel computer to run it, it is worthwhile to estimate the potential performance gain. Unfortunately, it is not all parts of a program that benefit from increased parallelisation. For example, suppose a program needs 20 hours to run, but a 1-hour part of the program cannot possibly be parallelised. This is not unlikely: perhaps that hour is spent reading configuration data, loading code, formatting human-readable reports, or waiting for the human operator to interact with the system somehow. Even if we optimise the program such that the optimisable 95% of the program runs in *zero time*, we have only achieved a speedup of 20.

Gene Amdahl phrased the now-famous *Amdahl's Law* [**10.1145/1465482.1465560**] to describe the theoretical speedup from parallelisation:

Definition 3 (Amdahl's Law) *If p is the proportion of execution time that benefits from parallelisation, then $S(N)$ is maximum theoretical speedup achievable by execution on N threads, and is given by*

$$S(N) = \frac{1}{(1-p) + \frac{p}{N}}$$

We can see that

$$S(N) \leq \frac{1}{1-p}$$

This means that the potential speedup by optimising part of a system is bounded by how dominant this part is in the overall runtime. It tells us that we should spend our time optimising the parts that take the most time to run.

While Amdahl's Law is usually applied to parallelisation, it can be used to characterise *any* situation where we are optimising a part of some system.

5.3 Gustafson's Law

As parallel supercomputers became more common in the 80s, researchers found that they routinely achieved speedup far in excess of what Amdahl's Law would predict. This is because Amdahl's Law is quite pessimistic, as it assumes that the workload stays *fixed* as we gain access to more computational resources. In practice, as workloads increase in size, the parallelisable fraction tends to *increase* in its share of the overall runtime. Also, when we get access to a larger machine, we tend not to be interested in solving our old problems faster, but in solving bigger problems in the same time as it took to solve our old problems. *Time* is the constant, not the workload.

Suppose we scale the runtime to be 1 and use s, p to indicate the fraction of this unit runtime spent in sequential and parallel code respectively on a parallel system with N threads. Then a sequential processor would require $s + N \times p$ time to execute the program. The scaled speedup of parallel execution is then

$$\frac{s + p \times N}{s + p} = s + p \times N = N + (1 - N) \times s$$

This observation was first published by John L. Gustafson [**10.1145/42411.42415**] and is therefore called Gustafson's Law:

Definition 4 (Gustafson's Law) *If s is the proportion of execution time that must be sequential, then $S(N)$ is maximum theoretical speedup achievable by execution on N threads, and is given by*

$$S(N) = N + (1 - N) \times s$$

Neither Amdahl's nor Gustafson's Laws are *laws* in the common sense of the word. Despite providing conflicting predictions, they can both be true under different circumstances. Amdahl's Law tells us about the limitations of parallelism under a fixed workload, while Gustafson's Law tells us about the limitations of parallelism where we assume we the workload grows proportionally with the amount of parallelism.

Chapter 6

Bibliography