Course notes for High Performance Programming and Systems

Kenneth Skovhede, Troels Henriksen

December 13, 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

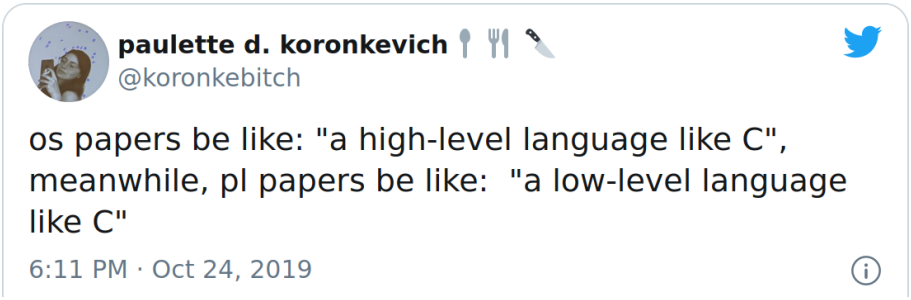


Figure 1.1: A remark on the clarity of terms in computer science.

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.

```c
#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.

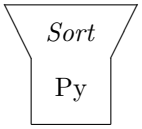
The main advantage of JIT compilation is that programs run much faster than when interpreting bytecode, because we ultimately do end up executing a machine code version of the program. Because JIT compilation takes place while the program is running, it is also able to measure the actual run-time behaviour 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

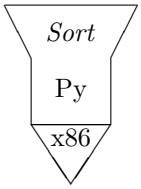
Interpreters and compilers allow us to consider programs as input and output of other programs. That is, they are *data*. *Tombstone diagrams* (sometimes called *T-diagrams*) are a visual notation that lets us describe how a program is translated between different languages (*compiled*), and when execution 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:

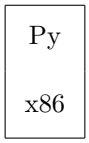


This is an incomplete diagram, since it contains programs we have not described how to execute. A machine that executes some language, say x86 machine code is illustrated as a downward-pointing triangle:

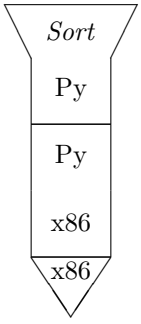
We can say that the Python program is executed on this machine, by stacking them:



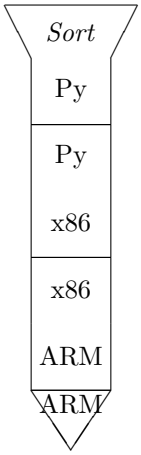
But this diagram is *wrong* — we are saying that a program written in Python is running on a machine that executes only x86. When putting together a tombstone diagram, we must ensure that the languages of the components match. While on paper, we can just assume a Python machine, this is not very realistic.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

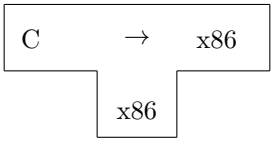


But maybe we are actually running on an ARM machine (as can be found in most phones), but still only have a Python interpreter in x86. As long as we have an x86 interpreter written in ARM machine code, this is no problem:

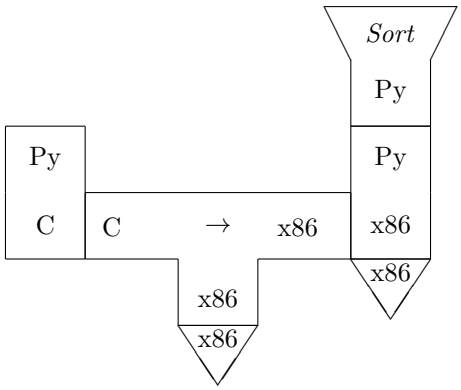


There is no limit to how high we can stack interpreters. All that matters is that at the end, we have either a machine that can run the implementation language of the bottommost interpreter. Of course, in practice, each level of interpretation adds overhead, so while tombstone diagrams show what is *possible*, 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:

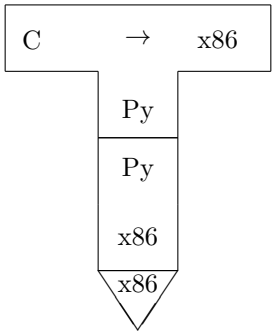


We can now put together a full diagram showing how the Python interpreter is translated from C to x86, and then used to run a Python program:



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.

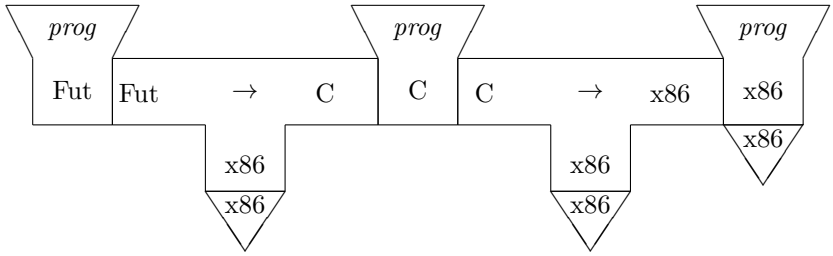
Compilers are also just programs, and must either be executed directly by an appropriate machine, or interpreted. For example, the following diagram shows how to run a C compiler in Python, on top of a Python interpreter in x86 machine code:



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:

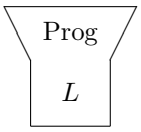
[3] `https://futhark-lang.org`



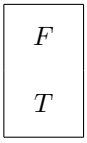
Many compilers have multiple internal steps—for example, a C compiler does not usually generate machine code directly, but rather generates symbolic assembly code, which an *assembler* then translates to binary machine code. Typically tombstone diagrams do not include such details, but we can include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

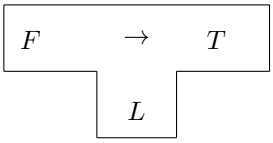


(a) A program written in language L.



(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.

$$(i, j) \mapsto i \times m + j \qquad (2.3)$$

(a) Row-major indexing.

$$(i, j) \mapsto j \times n + i \qquad (2.4)$$

(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

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

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. This is called an *implicit barrier*: a point where execution pauses until all threads reach it. 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` iterations 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 chapter 6 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] = i;
  }

  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.

4.1.2 Parallelism versus Concurrency

The terms *parallelism* and *concurrency* are frequently and historically used interchangeably. If you look then up in a dictionary, you will find them to have almost the same definitions. In computer science, they are terms of art with distinct (although related) meanings.

To illustrate concurrency, consider an FPS video game, which from a programming perspective is basically a real-time interactive simulation. Many things need to happen concurrently:

- We need to figure out what sounds and music to play and send it to the IO device connected to the speakers.
- Many times per second, we need to draw to the screen a rendering of the world as observed by the player.
- Perhaps we wish to guess at where the player is headed next, and preload those parts of the game world.
- We need to run artificial intelligence for computer-controlled enemies.
- We need to compute physics interactions.
- Probably we also wish to perform cleanup tasks, such as removing objects from the game world that after a while are no longer necessary (e.g. the remains of deceased enemies), to clear up system resources.

In terms of programming, it is nicer if we can write each of these parts as separate flows of control. The artificial intelligence code should not worry about constantly checking whether it's time to draw a new screen frame, or whether the player hit some key. We might implement each of these parts as a distinct thread, and depend on the operating system to *context-switch* between them as needed, and maybe use some form of *scheduler policy* that understands that it's more important for the threads responsible for music and graphics to run when they need to, than the cleanup thread. If we have only a single processor, then all these different threads run *concurrently*, in that they overlap in time, but only one will physically be executing instructions at any given point in time. This means that concurrency can be a useful programming model even when the goal is not to make the program faster. For that matter, threads are not the only way to implement concurrency—asynchronous event loops are a popular technique in highly scalable web servers, but outside the scope of this course.

Definition 1 (Concurrency) *Concurrency is the use of multiple, possibly communicating, logical flows of control.*

Parallelism is about making programs faster by performing several computations at the same time. When we have a program with multiple threads, such as the video game example above, then if we have a machine with more than

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;
}
```

one processing core (which is essentially every machine these days), we can run several of those threads in parallel. While in Unix, threads are the fundamental *implementation mechanism* for obtaining parallelism, we often program in languages or frameworks that do not directly expose threads, because they can be difficult to work with. For example, OpenMP is a parallel programming model, but for simple parallel loops, we do not concern ourselves with actual threads, and we only have a single logical control flow.

Definition 2 (Parallelism) *Parallelism is the simultaneous use of multiple processing units, with the goal of speeding up a computation.*

In scientific computing we are mostly concerned with parallelising loops with independent iterations, and not with concurrent flows of control. While OpenMP allows us to peek beneath the covers and interact with the threads that it uses to *implement* the parallel loop abstraction, there does exist forms of parallelism, and parallel programming languages, that do not expose this abstraction, and are truly parallel without exposing any concurrency.

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.

4.3 Nested loops

We can prefix every parallelisable **`for`**-loop with an OpenMP parallelisation directive. But what happens if we nest multiple parallel loops such as the following?

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
#pragma omp parallel for
  for (int j = 0; j < m; j++) {
    ...
  }
}
```

In principle, the answer is easy: the original master thread launches worker threads to handle each of the `n` outer iterations, and these individual worker threads may then launch more worker threads to handle the inner `m` iterations. This is called *nested parallelism*: iterations of a parallel loop may itself contain more parallel loops. The overhead of nested parallelism quickly becomes significant, in particular if we have recursive functions so that the nesting is *dynamic*, so in practice it is not widely used in OpenMP. In fact, OpenMP implementations are likely to ignore nested parallelisation directives, unless the `OMP_NESTED` environment variable is set to `True` when running the program.

However, a common case of nested parallelism is iterating across all elements of a multidimensional array, such as above, where we are conceptually covering all indexes of an `n` by `m` array. This is called *regular nested parallelism*, because the iteration count of the inner loop (`m`) is *invariant* (the same) for all iterations of the outer loop. Such a nested loop can be collapsed to a single loop that performs `n*m` iterations:

```
#pragma omp parallel for
for (int ij = 0; ij < n*m; ij++) {
  int i = ij / m;
  int j = ij % m;
  ...
}
```

We use division and modulo operations to extract the intended indexes from the "combined" index `ij`—this is actually the inverse of the row-major two-dimensional index function.

However, writing code like this is not very nice, as it obscures our intent. Fortunately, OpenMP provides the `collapse` clause that we can use to tell OpenMP to parallelise multiple *perfectly nested* loops. By perfectly nested, we mean that the outermost loops contain only a loop, and no other statements. An example is shown in listing 4.3, where we tell OpenMP to treat the two-deep *loop nest* as a single parallel loop.

Listing 4.3: Matrix addition with OpenMP.

```
void matadd(int n, int m,
            const double *x, const double *y,
            double *out) {
#pragma omp parallel for collapse(2)
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      out[i*n+j] = x[i*n+j] + y[i*n+j];
    }
  }
}
```

You should generally use the `collapse` clause when writing such perfectly nested loops, which occurs frequently when implementing matrix operations. But more generally, it is usually not worth worrying too much about parallelising all inner loops of an OpenMP program. All we have to do is provide enough parallel work such that all processors on the system have work to do, and as of this writing, even a very large computer is unlikely to have more than 256 CPU cores—and on a personal computer, 16 is more likely.

4.4 Scheduling

When we use a directive to ask OpenMP to execute a loop in parallel, the compiler and runtime system will decide how the iterations should be distributed among the threads. By default, OpenMP uses *static scheduling*.

Definition 3 (Static scheduling) *When entering a parallel loop, we assign each thread exactly the number of iterations to execute.*

For example, when executing a loop with n iterations on m threads, we might assign $\frac{n}{m}$ iterations to each thread.

Static scheduling can be non-optimal when a loop is not *load-balanced*, meaning that not all loop iterations take the same time. For such an *imbalanced* loop, some threads may finish their iterations quickly and then sit idle while the other threads finish theirs. In such cases, we can ask OpenMP to use *dynamic scheduling*.

Definition 4 (Dynamic scheduling) *When entering a parallel loop, we assign each thread an iteration. When a thread finishes an iteration, it receives a new one to execute.*

The advantage of dynamic scheduling is that an idle thread will receive more work (if any is available). The disadvantage is that dynamic scheduling requires additional communication and synchronisation—while this is done for us by the runtime system, it carries a performance overhead.

As an example of the benefit of dynamic scheduling, consider parallelising loops where each iteration computes Fibonacci numbers using the recursive function defined in listing 4.4[3].

Since computation of $fib(i+1)$ takes over twice the time of $fib(i)$, we can produce a very imbalanced loop by letting iteration i compute $fib(i)$. Listing 4.5 shows how to parallelise this with a static schedule in OpenMP. On my machine, for `n=45`, this program runs in $5.2s$. We can ask for dynamic scheduling by using the the `schedule(dynamic)` clause, as shown on listing 4.6. On my machine, this version runs in $2.27s$ - a speedup (section 5.1) of $2.29\times$.

[3]This is an inefficient way to compute Fibonacci numbers—we only use as an expensive computation that will take some time.

Listing 4.4: Recursive Fibonacci function.

```
int fib(int n) {
  if (n <= 1) {
    return 1;
  } else {
    return fib(n-1) + fib(n-2);
  }
}
```

Listing 4.5: Fibonacci loop with static scheduling.

```
#pragma omp parallel for schedule(static)
  for (int i = 0; i < n; i++) {
    fibs[i] = fib(i);
  }
```

Listing 4.6: Fibonacci loop with dynamic scheduling.

```
#pragma omp parallel for schedule(dynamic)
  for (int i = 0; i < n; i++) {
    fibs[i] = fib(i);
  }
```

By default, dynamic scheduling assigns single loop iterations to threads. This is not a problem for the Fibonacci example, because there are few loop iterations, and they take a fairly long time to run. For loops with many iterations, where dynamically scheduling single iterations at a time would involve too much overhead, we can add a *chunk size* to the scheduling clause. For example,

```
#pragma omp parallel for schedule(dynamic, 100)
```

will schedule in chunks of 100 iterations at a time.

The `guided` schedule is similar to `dynamic`, but starts out with big chunks and may then decrease the chunk size during program run-time if the work is imbalanced.

OpenMP's default behaviour will usually do a good job scheduling most loops. But if we do not see the performance gains we would expect, and we see in a process monitor that some of our processors are idle while our program is running, it is worth considering whether our loops could benefit from a scheduling clause.

Chapter 5

Parallel Speedup and Scalability

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 5 (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 $\times$, as in $2.5\times$. 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 sometimes 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 6 (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*

$$\frac{Q_2}{Q_1}$$

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:

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

The speedup in throughput of P_2 over P_1 is

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

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 Scalability

By *scalability* we mean how the system improves in its capacity (runtime or throughput) as we add more resources, such as more processors. It can also be used to describe how the performance changes as the problem size increases—this is essentially what big-O notation is for. With respect to parallelisation, we are interested in how the performance of a system changes as we add or exploit more processors. We distinguish two forms of scalability.

Definition 7 (Strong scaling) *How the runtime varies with the number of processors for a fixed problem size.*

Definition 8 (Weak scaling) *How the runtime varies with the number of processors for a fixed problem size relative to the number of processors.*

5.2.1 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* [1] to describe the theoretical speedup from parallelisation:

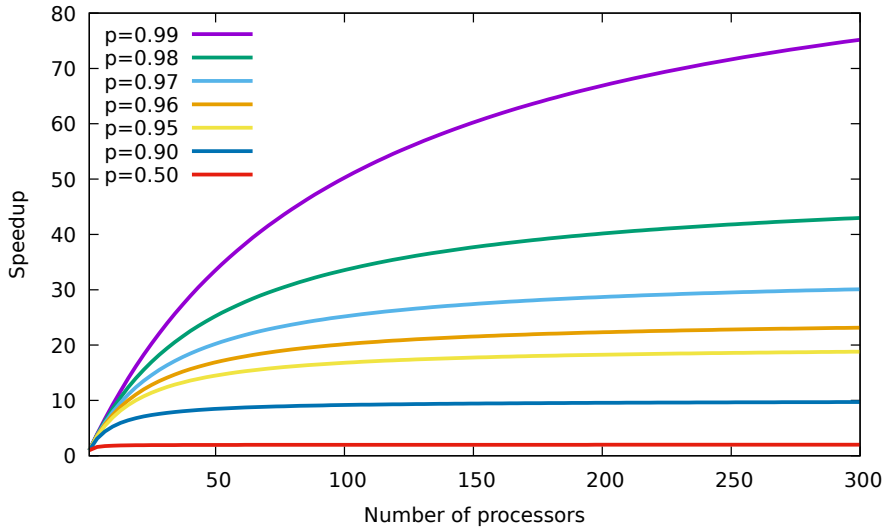


Figure 5.1: A graph of Amdahl's Law, plotted for various values of p.

Definition 9 (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. As fig. 5.1 shows, it is a rather pessimistic law—even in the case where 99% of the program can be parallelised, execution on 300 processors will give us a speedup of about 75 over a single processor.

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.

5.2.2 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.

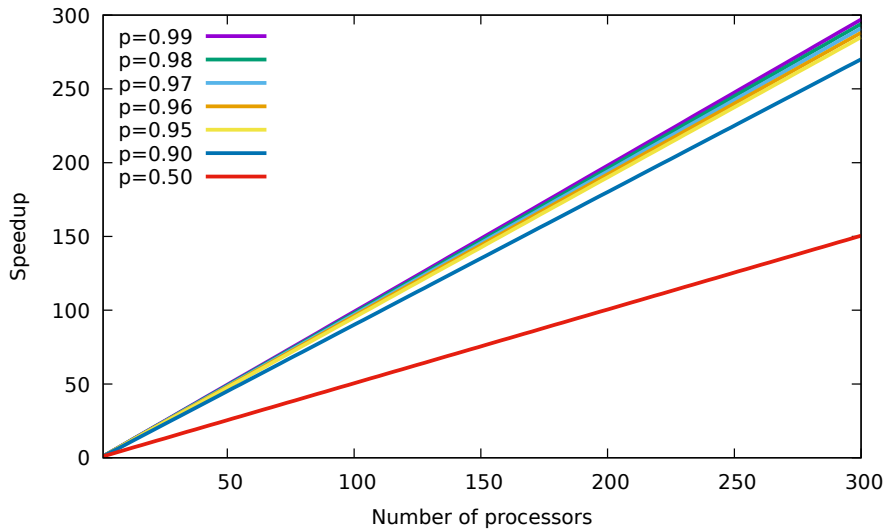


Figure 5.2: A graph of Gustafson's Law, plotted for various values of p (note that $p = 1 - s$).

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 [2] and is therefore called Gustafson's Law:

Definition 10 (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*

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

Compared to Amdahl, Gustafson is much more of an optimist—as shown on fig. 5.2, Gustafson's Law plots as a *line*, meaning that the speedup as we add more processors is *linear*.

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. Broadly, Amdahl's Law predicts strong scalability, and Gustafson's law predicts weak scalability.

Both laws make significant simplifying assumptions—in practice, little scientific code consists of enormous fully parallel loops with completely independent iterations, but will tend to require some form of routine communication, proportional to the number of processors involved. Specifically, these laws tend to discount the nonlinear scaling of accessing large amounts data due to locality effects.

Chapter 6

Loop dependence analysis and loop transformations

This chapter is adapted with permission from the PMPH Lecture Notes written by Cosmin Oancea.

So far, we have assumed that the user writes a fresh implementation of a known algorithm, and that it can be straightforwardly expressed as fully parallel loops, or as a simple reduction. However, there is a lot of legacy sequential scientific code written in imperative languages such as C++, Java, Fortran, and either the precise algorithm to which they correspond to (i) may have been forgotten (not documented), or (ii) a fresh implementation is infeasible (e.g., because it costs too much). At some point you may wish (or be asked) to parallelise such code to run efficiently on a certain hardware.

This will require you to:

1. Identify the loop nests[1] where most of the runtime is spent.
2. Parallelise these loops by reasoning at a low level of abstraction about which loops in the nest are parallel.
3. Decide on the manner in which loop nests can be re-written in order to optimise locality of reference, load balancing, thread divergence, etc.

The main source of inspiration for the material presented in this and chapter 7 has been the book "Optimizing compilers for modern architectures: a dependence-based approach" [3].

This chapter is organised as follows:

- Section 6.1 introduces various nomenclature, in particular the notion of cross-iteration dependency, and it shows how to summarise dependencies across the iteration space into a succinct representation, named direction vectors, that promotes reasoning about various code transformations.

[1] A *loop nest* is a collection of multiple loops nested within each other.

```
S1:   X  = ..      S1:   .. = X       S1:   X = ...
S2:   .. = X       S2:   X  = ..      S2:   X = ...
```

(a) RAW: Read after write. (b) WAR: Write after read. (c) WAW: Write after write.

Figure 6.1: Three different kinds of dependencies.

- Section 6.2 presents a simple theorem that allows easy identification of loop-level parallelism.

6.1 Direction vectors

We start by defining the various reasons why a program statement must be executed after some previous. C executes statements in *program order*—the order they occur in the source code. Dependence analysis is about distinguishing when the program order is crucial for correct execution, and in which cases it is arbitrary because C requires us to write statements in *some* order. If a statement S_2 *depends* on S_1, then S_2 must always be executed after S_1. Dependencies typically arise because the two statements interact with the same values, with at least one of the statements changing the value. When two statements are not dependent on each other, they can in principle be executed in parallel. Our goal is to use dependence analysis to identify when entire *loop iterations* are independent, which means that they can be executed in parallel.

The possible kinds of dependencies are depicted in fig. 6.1 between two statements S_1 and S_2, which reside for simplicity in the same *basic block*—a straight-line of code which is always entered by the first statement and is exited after the execution of the last statement. For example, the body of a loop can be considered a basic block if it contains no control flow and no `continue` textttbreak statements. The possible kinds of dependencies are as follows:

RAW (fig. 6.1a): refers to the case when a write to a register or memory location is followed, in program order, by a read from the same register or memory location; this is typically referred to as a read-after-write hazard in hardware-architecture nomenclature, and as a *true* dependency in loop-based analysis nomenclature. The word *true* refers to the fact that such a dependency denotes a producer-consumer relation, in which the value produced by S_1 is used in S_2. The producer-consumer relation is an algorithmic property of the program; such a dependency cannot be eliminated other than by changing the underlying algorithm.

WAR (fig. 6.1b): refers to the case when a read from a register or memory location is followed, in program order, by a write to the same register or memory location; this is referred to as a write-after-read hazard, and equivalently as an *anti* dependency. The problem here is that, if the two statements are reordered—meaning S_2 executes before S_1, then the

value needed by S_1 is no longer available because it has been already overwritten by S_2.

WAW (fig. 6.1c): refers to the case when a write from a register or memory location is followed, in program order, by another write to the same register or memory location; this is referred to as a write-after-write hazard, and equivalently as an *output* dependency. The problem here is that, if the two statements are reordered—meaning S_2 executes before S_1, then the final value stored in register or memory location is that of S_1 rather than that of S_2.

In what parallelism or loop analysis is concerned, we are primarily interested in analysing the (true, anti and output) *dependencies that occur across different iterations of the loop*. For example such a true dependency would correspond to the case in which an early iterations `i` writes/produces an array element that is subsequently read/consumed in a later iteration `j > i`.

In what parallelisation is concerned, the main limiting factor are the true dependencies—which correspond to an algorithmic property—because the anti and output dependencies can be typically eliminated by various techniques, as we shall see.

6.1.1 Loop notation and lexicographic ordering of iterations in a loop nest

In the following we will write loop nests using the **do**-loop notation, inspired by Fortran:

```
do i = low_bound, high_bound, stride:
    ... loop body ...
```

These **do** loops have the property that the loop index `i` cannot be modified inside the loop. As such the first iteration uses `i = low_bound`, the second iteration uses `i = low_bound + stride`, the third uses `i = low_bound + 2*stride`, and so on, for as long as `i` $\leq$ `high_bound`. When the stride is one, one may use the abbreviation **do** `i = low_bound, high_bound`.

In the following we will also assume that iterations in a loop nest are represented by a vector, in which iterations numbers are written down from the corresponding outermost to the innermost loop in the nest, and are ordered *lexicographically*—i.e., are ordered consistently with the order in which they are executed in the (sequential) program. This means that in the loop nest below:

```
do i = 0, N-1:
  do j = 0, M-1:
    ... loop-nest body ...
```

iteration $\vec{k}$=`(i=2,j=4)` is smaller than iteration $\vec{l}$=`(i=3,j=3)` (i.e., $\vec{k} < \vec{l}$), because the second iteration of the outer loop is executed before the third

iteration of the outer loop, no matter what the iteration numbers are executed for the inner loop (of index `j`). In essence the iteration numbers of inner loops are only used to discriminate the order in the cases in which all the outer-loop iterations are equal, for example $\vec{k}$=`(i=3, j=3)` $<$ $\vec{l}$=`(i=3, j=4)`

6.1.2 Dependency definition

The precise definition of a dependency between two statements located inside a loop nest is given below.

Definition 11 (Loop Dependency) *There is a dependency from statement S_1 to statement S_2 in a loop nest* if and only if *there exists loop-nest iterations $\vec{k}$, $\vec{l}$ such that $\vec{k} \leq \vec{l}$ and there exists an execution path from statement S_1 to statement S_2* such that:

1. *S_1 accesses some memory location M in iteration $\vec{k}$, and*

2. *S_2 accesses the same memory location M in iteration $\vec{l}$, and*

3. *one of these accesses is a write.*

In such a case, we say that S_1 is the source *of the dependence, and that S_2 is the* sink *of the dependence, because S_1 is supposed to execute before S_2 in the sequential program execution.*
Dependencies can be visually depicted by arrows pointing from the source to the sink of the dependence.

The definition basically says that in order for a dependency to exist, there must be two statements that access *the same memory location* and one of the accesses must be a write—two read instructions to the same memory location do not cause a dependency. The nomenclature denotes the statement that executes first in the program order as *the source* and the other as *the sink* of the dependency. We represent a dependency graphically with an arrow pointing from the source to the sink.

Optimisations for instruction-level parallelism (ILP)—meaning eliminating as much as possible the stalls from processor's pipeline execution[2]—typically rely on intra-iteration analyses (i.e., $\vec{k} = \vec{l}$). Higher-level optimisations, such as detection of loop parallelism, are mostly concerned with analysing inter-iteration dependencies (i.e., $\vec{k} \neq \vec{l}$). For example the main aim could be to disprove the existence of inter-iteration dependencies, such that different iterations may be scheduled out of order (in parallel) on different cores, while the body of an iteration is executed sequentially on the same core. In such a context, intra-iteration dependencies are trivially satisfied, and so are not very interesting.

[2]Outside the scope of HPPS.

```
do i = 0, N-1:
  do j = 0, N-1:
    S1 : A[j,i] = A[j,i]...
```

(a)

```
do i = 1, N-1:
  do j = 1, N-1:
    S1 : A[j,i] = A[j-1,i-1]...
    S2 : B[j,i] = B[j-1,i]...
```

(b)

```
do i = 1, N-1
  do j = 0, N-1
    S1 : A[i,j] = A[i-1,j+1]...
```

(c)

Figure 6.2: Three simple running code examples that will be used to demonstrate data-dependency analysis and related transformation.

6.1.3 Aggregating dependencies with direction vectors

Assume the three loops presented in fig. 6.2, which will be used as running example to demonstrate data dependence analysis and related transformations. We make the important observation that the code is not in three-address code (TAC) form: a statement such as `A[j,i] = A[j,i] + 3` would correspond to three TAC or hardware instructions: one that loads from memory `tmp1 = A[j,i]`, followed by one that performs the arithmetic operation `tmp2 = tmp1 + 3`, followed by one that writes to memory `A[j,i] = tmp2`. Automated analysis is for simplicity usually carried out on programs in TAC form but, for brevity, our analysis will be carried out at the statement level. A human may start analysing dependencies:

- by depicting the iteration space in a rectangle in which the x axis and y axis correspond to iteration numbers of the inner loop j and outer loop i, respectively, and
- then by reasoning point-wise about what dependencies may happen between two iterations.

A graphical representation of the dependencies of the three running code examples is shown in fig. 6.3. They can be intuitively inferred as follows:

- For the loop in fig. 6.2(a), different loop-nest iterations (i_1, j_1) and (i_2, j_2) necessarily read and write different array elements `A[`j_1, i_1`]` and `A[`j_2, i_2`]`.

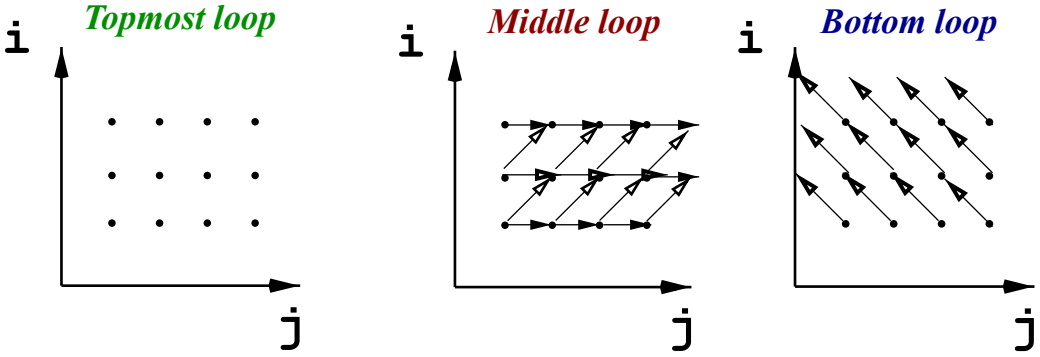


Figure 6.3: Graphical representation of the dependencies for the three running examples shown in fig. 6.2; the x and y axis correspond to the index of the inner and outer **do** loop, respectively.

This is because our assumption is that $(i_1, j_1) \neq (i_2, j_2)$, hence it cannot be that both $i_1 = i_2$ and $j_1 = j_2$. As such, the representation of dependencies should be a set of points (no arrows), meaning that all dependencies actually occur inside the same iteration—in fact they are anti intra-iteration dependencies (WAR) because `A[j,i]` is read first and then `A[j,i]` is written inside the same iteration.

- For the loop in fig. 6.2(b) we reason individually for statements S_1 and S_2 because each statement accesses (one) array `A` and `B`, respectively:

  S_1 : Let's take an iteration, say $(i_1 = 2, j_1 = 3)$, which *reads* element `A[`j_1`-1,`i_1`-1] = A[2,1]`. Since an iteration (i, j) always writes the element `A[i,j]`, we can reason that iteration $(i_2 = 1, j_2 = 2)$ will *write* the same element `A[2,1]`. It follows that we have discovered a true (RAW) dependency, depicted in the figure with and arrow, from the source iteration $(i_2 = 1, j_2 = 2)$—which writes `A[2,1]`—to the sink iteration $(i_1 = 2, j_1 = 3)$—which reads `A[2,1]`. This is because iteration $(1, 2) < (2, 3)$ according to the lexicographical ordering, and as such, the read happens after the write (RAW) in program order. One can individually reason for each point of the iteration space and fill it with oblique, forward-pointing arrows denoting true dependencies between different instances of statement S_1 (executing in different iterations).

  S_2 : Following a similar rationale, iteration $(i_1 = 2, j_1 = 3)$ *reads* element `B[`j_1`-1,`i_1`] = B[2,2]`, and iteration $(i_2 = 2, j_2 = 2)$ *writes* element `B[2,2]`. It follows that we have discovered a true (RAW) dependency with source $(i_2 = 2, j_2 = 2)$ and sink $(i_1 = 2, j_1 = 3)$, because $(2, 2) < (2, 3)$ in lexicographic ordering. Since $i_1 = i_2$ we depict the arrow parallel with the horizontal axis (that depicts values

of j). One can fill in the rest of the iteration space with horizontal arrows.

- For the loop in fig. 6.2(c) we reason in a similar way: take iteration $(i_1 = 2, j_1 = 3)$ that *reads* element `A[2-1,3+1] = A[1,4]`. This element is *written* by iteration $(i_2 = 1, j_2 = 4)$. It follows that we have discovered a true (RAW) from source $(i_2 = 1, j_2 = 4)$ to sink $(i_1 = 2, j_1 = 3)$—because the read happens in iteration $(2, 3)$ which comes after the write in iteration $(1, 4)$, i.e., $(1, 4) < (2, 3)$. Thus, one can fill in the iteration space with oblique, backward-pointing arrows, denoting true dependencies between instances of S_1 executing in different iterations.

We have applied above a human type of reasoning and, as a result, we have a graphical representation of all dependencies. However, such a reasoning is not suitable for automation because (i) the loop counts are statically unknown—they depend on the dataset—hence one cannot possibly represent an arbitrary large iteration space, and, more importantly, (ii) even if the loop counts would be statically known it is still inefficient to maintain and work with all this pointwise information.

A representation that promotes compiler reasoning should succinctly capture the repeated pattern in the figure. Intuitively and imprecisely, for fig. 6.2(a) the pattern would correspond to a point, for fig. 6.2(b) it would correspond to two arrows—one oblique and one horizontal forward pointing arrows—and for fig. 6.2(c) it would correspond to an oblique, backward-pointing arrow. These patterns are formalized by introducing the notion of *direction vectors*.

Definition 12 (Dependency direction vector) *Assume there exists a dependency with source S_1 in iteration $\vec{k}$ to sink S_2 in iteration $\vec{l}$ ($\vec{k} \leq \vec{l}$). We denote by m the depth of the loop nest, we use i to range from $0, \ldots, m-1$, and we denote by x_i the i^{th} element of some vector $\vec{x}$ of length m.*

The direction vector *between the instance of statement S_1 executed in some source iteration $\vec{k}$ and statement S_2 executed in sink iteration $\vec{l}$ is denoted by $\vec{D}(S_1 \in \vec{k}, S_2 \in \vec{l})$, and corresponds to a vector of length m, whose elements are defined as:*

$$D_i(S_1 \in \vec{k}, S_2 \in \vec{l}) = \begin{cases} < & \text{if it is provably that } k_i < l_i, \\ = & \text{if it is provably that } k_i = l_i, \\ > & \text{if it is provably that } k_i > l_i, \\ \star & \text{if } k_i \text{ and } l_i \text{ are statically uncomparable.} \end{cases}$$

The first three cases of the definition above assume that the ordering relation between k_i and l_i can be statically derived in a generic fashion (for any source k_i and l_i); if this is not possible than we use the notation $\star$ which *conservatively* assumes that any directions may be possible—i.e., star should be understood as simultaneous existence of all `<, =, >` directions. For example, the loop

```
do i = 0, N-1:
  S1 : A[ X[i] ] = ...
```

would result in direction vector `[*]` corresponding to a potential output dependency (WAW), because the write access to `A[ X[i] ]` is statically unanalysable—for example under the assumption that the index array `X` is part of the dataset—and, as such, all direction vectors may possibly hold between various pairs of instances of statement S_1 executed in different iterations.

Note that the symbols `<, =, >` are *not* connected at all to the type of the dependency, e.g., true (RAW) or anti (WAR) dependency. The type of the dependency is solely determined by the operation of the source and that of the sink: If the source is a write statement and the sink is a read then we have a true (RAW) dependency; if the source is a read and the sink is a write then we have an anti (WAR) dependency; if both source and sink are writes then we have an output (WAW) dependency.

The meaning of the symbol > at some position i is that the source iteration at loop-level i is greater than the sink iteration at loop-level i. This case is possible, for example the code in fig. 6.2(c) shows a dependency with source iteration $(1, 4)$ and sink iteration $(2, 3)$. At the level of the second loop, we have $4 > 3$ hence the direction is > but still the source iteration is less than the sink iteration $(1, 4) < (2, 3)$ because of the first loop level. This observation leads to the following corollary:

Corollary 1 (Direction vector legality) *A direction vector is legal (well formed), if removing the = entries does* not *result in a leading > symbol, as this would mean that an iteration depends on a future iteration, and depending on a future event is considered impossible, and as such illegal.*

It remains to determine the sort of reasoning that can be applied to compute the direction vectors for the code examples in fig. 6.2:

The loop in fig. 6.2a: dependencies can occur only between instances of statement S_1, executed in different (or the same) iterations. We recall that, by the definition of dependency, the two (dependent) iterations must access the same element of `A` and at least one iteration should perform a write. Since statement S_1 performs a read and a write to elements of array `A`, two kinds of dependencies may occur:

WAW: an output dependency may be caused by two write accesses in two different iterations, denoted (i_1, j_1) and (i_2, j_2). The written element is thus `A[`j_1`,`i_1`]`, which must be the same as `A[`j_2`,`i_2`]` for a dependency to exist. This results in the system of equations

$$\begin{cases} i_1 = i_2 \\ j_1 = j_2 \end{cases}$$

which leads to direction vector `[=,=]`. Hence, an output dependency from S_1 to S_1 happens in the same iteration, but statement

S_1 executes only one write access in the same iteration. The conclusion is that no output dependency can occur, hence the direction vector is discarded.

RAW: a true or anti dependency—we do not know yet which—will be caused by the read access from `A` and the write access to `A` in different (or same) iterations. Remember that a statement such as `A[j,i] = A[j,i] + 3` actually corresponds to three hardware instructions, hence either a cross- or an intra-iteration dependency will necessarily occur. Assume some iteration (i_1, j_1) reads from `A[`j_1`,`i_1`]` and iteration (i_2, j_2) writes to `A[`j_2`,`i_2`]`. In order for a dependency to exist, the memory location of the read and write must coincide; this results in the system of equations

$$\begin{cases} i_1 = i_2 \\ j_1 = j_2 \end{cases}$$

from which we can derive the direction vector: `[=,=]`. This implies that the dependency happens in the same iteration, hence it is an intra-iteration dependency. Furthermore, since the write follows the read in the instruction order of an iteration, this is an anti dependency (WAR).

For the loop in fig. 6.2b: dependencies may possibly occur between instances of statement S_1 and between instances of statement S_2. The case of output dependencies is disproved by a treatment similar to the bullet above. It remains to examine the dependency caused by a read and a write in different instances of S_1 and S_2, respectively:

S_1: assume iteration (i_1, j_1) and iteration (i_2, j_2) reads from and writes to the same element of `A`, respectively. Putting this in equation results in the system

$$\begin{cases} i_1 - 1 = i_2 \\ j_1 - 1 = j_2 \end{cases}$$

which necessarily means that $i_1 > i_2$ and $j_1 > j_2$. However, we do not know yet which iteration is the source and which is the sink. Assuming that (i_1, j_1) is the source results in the direction vector `[>,>]`, which is illegal by corollary 1, because a direction vector cannot start with the > symbol. It follows that our assumption was wrong: (i_2, j_2) is the source and (i_1, j_1) is the sink, which means that this is a cross-iteration true dependency (RAW)—because the sink iteration reads the element that was previously written by the source iteration—and its direction vector is `[<,<]`.

S_2: a similar rationale can be applied to determine that two instances of S_2 generate a true cross-iteration dependency (RAW), whose direction vector is `[=,<]`. In short, using the same notation results in

the system of equations

$$\begin{cases} i_1 = i_2 \\ j_1 - 1 = j_2 \end{cases}$$

hence the source must be (i_2, j_2) and the sink must be (i_1, j_1) and the direction vector is `[=,<]`.

For the loop in fig. 6.2c: dependencies may possibly occur between instances of statement S_1. Assume iteration (i_1, j_1) and (i_2, j_2) reads from and writes to the same element of `A`, respectively. Putting this into equation from definition 12 results in the system

$$\begin{cases} i_1 - 1 = i_2 \\ j_1 + 1 = j_2 \end{cases}$$

which necessarily implies that $i_1 > i_2$ and $j_1 < j_2$. Choosing (i_1, j_1) as the source of the dependency results in direction vector `[>,<]`, which is illegal because it has > as the first non-= outermost symbol, as stated by corollary 1. It follows that (i_1, j_1) must be the sink and (i_2, j_2) must be the source, which results in the direction vector `[<,>]`, which is legal. Since the source writes and the sink reads, we have a true dependency (RAW). Moreover since the direction vector indicates that the source iteration is strictly less than the sink iteration, this is also a cross-iteration dependency.

Definition 13 (Dependency direction matrix) *A direction matrix is obtained by stacking together the direction vectors of all the intra- and cross-iteration dependencies of a loop nest (i.e., between any possible pair of write-write or read-read instruction instances).*

In conclusion, the direction matrices for the three running code examples:

Figure 6.2a: $\begin{cases} \texttt{[=,=]} \end{cases}$

Figure 6.2b: $\begin{cases} \texttt{[<,<]} \\ \texttt{[=,<]} \end{cases}$

Figure 6.2c: $\begin{cases} \texttt{[<,>]} \end{cases}$

The following sections will show how the legality of powerful code transformations can be reasoned in a simple way in terms of direction vectors/matrices.

6.2 Determining loop parallelism

A loop is said to be parallel if its execution does not cause any (true, anti, or output) dependencies between iterations—the loop execution is assumed to be fixed in a specific iteration of an (potentially empty) enclosing loop context.

The following theorem states that a sufficient condition for a loop to be parallel is that for all the elements in the loop's corresponding direction-matrix column, it holds that the element is either = or there exists an outer loop whose corresponding direction is < (on that row). In the latter case we say that the outer loop *carries* all the dependencies of the inner loop, i.e., fixing an iteration of the outer loop (think executing the outer loop sequentially) would guarantee the absence of cross-iteration dependencies in the inner loop.

Theorem 1 (Parallel loop) *We assume a loop nest denoted by $\vec{L}$, whose direction matrix is denoted by M and consists of m rows. A sufficient condition for a loop at depth k in $\vec{L}$, denoted L_k, to be parallel is that $\forall i \in \{0, \ldots m-1\}$ either $M[i,k]$ is = or there exists an outer loop at depth $q < k$ such that $M[i,q]$ is <. Proof left as an exercise.*

Theorem 1 claims to give only a sufficient condition for loop parallelism because it assumes that symbols such as $\star$ may be part of the direction vector elements—we recall that $\star$ conservatively assumes that all directions <,=,> may be possible. If $\star$ does not appear in the direction matrix, then the condition becomes *necessary* as well as sufficient. Let us analyse the parallelism of each loop in our running examples:

Figure 6.2a: The direction matrix is `[=,=]`, hence by theorem 1, both loops in the nest are parallel because all the directions are =.

Figure 6.2b: The direction matrix is

$$M = \begin{cases} \texttt{[<,<]} \\ \texttt{[=,<]} \end{cases}$$

hence neither the outer nor the inner loop can be proven parallel by theorem 1. In the former case this is because $M[0,0]$ is < and there is no other outer loop to carry dependencies. In the latter case this is because $M[1,1]$ is < and the outer loop for that row has direction = (instead of <, which would have been necessary to carry the dependencies of the inner loop).

Figure 6.2c: The direction matrix is `[<,>]`, which means that the outer loop is not parallel—because it has a leading < direction—*but the inner loop is parallel* because the outer loop starts with < on the only row of the direction matrix, and, as such, it carries all the dependencies of the inner loop. To understand what this means, take a look again at the actual code in fig. 6.2c. Suppose we fix the outer iteration number to some

value i. Then the read accesses always refer to row $i-1$ of matrix `A` and the write accesses always refer to row i of `A`; hence a cross-iteration dependency cannot happen in the inner loop because no matter the value of j, the read and write statement instances cannot possibly refer to the same location of `A`.

Chapter 7

Loop transformations

o

This chapter is adapted with permission from the PMPH Lecture Notes written by Cosmin Oancea.

This chapter is organised as follows:

- Section 7.1 presents a simple theorem that gives necessary conditions for the safety of the transformation that interchanges two perfectly nested loops.
- Section 7.2 discusses the legality and the manner in which a loop can be distributed across the statements in its body.
- Section 7.3 discusses techniques for eliminating cross-iteration write-after-read and write-after-write dependencies.
- Section 7.4 introducing a simple transformation, named stripmining, which is always valid, and shows how block and register tiling can be derived as a combination of stripmining, loop interchange and loop distribution.

7.1 Loop interchange: legality and applications

Direction vectors are not used only for proving the parallel nature of loops, but can also enable powerful code restructuring techniques. For example they can be straightforwardly applied to determine whether it is safe to interchange two loops in a perfect loop nest[1]—which may result in better locality and even in changing an inner loop nature from dependent (sequential) to independent (parallel).

The following theorem gives a sufficient condition for the legality of loop interchange—i.e., for the transformation to result in code that is semantically equivalent to the original one.

[1]A perfect loop nest is a nest in which any two loops at consecutive depth levels are not separated by any other statements; for example all loop nests in fig. 6.2 are perfectly nested.

Theorem 2 (Legality of Loop Interchange) *A sufficient condition for the legality of interchanging two loops at depth levels k and l in a perfect nest is that interchanging columns k and l in the direction matrix of the loop nest* does not result *in a (leading) > direction as the leftmost non-= direction of any row.*

The theorem above shows that the legality of loop interchange can be determined solely by inspecting the result of permuting the direction matrix in the same way as the one desired for loops. For the rationale related to why a row-leading > direction is illegal, we refer the reader to corollary 1: a non-= leading > direction would correspond to depending on something that happens in the future: this currently seems impossible in our universe, and as such it signals an illegal transformation. The following corollary can be easily derived from theorem 2:

Corollary 2 (Interchanging a parallel loop inwards) *In a perfect loop nest, it is always safe to interchange a* parallel loop *inwards one step at a time (i.e., if the parallel loop is the k^{th} loop in the nest then one can always interchange it with loop $k+1$, then with loop $k+2$, etc.).*

The corollary says that if we somehow know the parallel nature of a loop, then we can safely interchange it in the immediate inward position, without even having to build the dependence-direction matrix.

Let us analyse the legality of loop interchange for the three loop nests of our running example:

Figure 6.2a: The direction matrix is `[=,=]` and, as such, it is legal to interchange the two loops, because it would result in direction matrix `[=,=]`. Moreover applying loop interchange in this case is highly beneficial because it *optimises locality of reference*: the loop of index `i` appears in the innermost position after the interchange, which optimally exploits spatial locality for the write and read accesses to `A[j,i]`.

Figure 6.2b: The direction matrices are

$$M = \begin{cases} [<,<] \\ [=,<] \end{cases}$$

and

$$M^{intchg} = \begin{cases} [<,<] \\ [<,=] \end{cases}$$

before and after interchange, respectively. It follows that the loop interchange is legal—because M^{intchg} satisfies theorem 2—and it also optimises spatial locality (as before). What is interesting about this example is that after the interchange, *the innermost loop has become parallel*, by theorem 1, because the outer loop caries all dependencies—the direction column corresponding to the outer loop consists only of < directions.

Figure 6.2c: The direction matrix is `[<,>]` and *interchanging the two loops is illegal* because the direction matrix obtained after the interchange `[>,<]` starts with a > direction; this would mean that the current iteration depends on a future iteration, which is impossible, hence the interchange is illegal.

7.2 Loop distribution: legality and applications

This section introduces a transformation, named loop distribution, where a loop is distributed across its statements. Potential benefits are:

- Loop distribution provides the bases for performing vectorisation: the innermost loop is distributed across its statements, and then the distributed loops are chunked (stripmined, section 7.4) by a factor that permits utilisation of processor's vector instructions.
- Loop distribution may enhance the degree of parallelism that can be statically mapped to the hardware. As discussed in section 4.3, OpenMP `collapse` clauses only apply to perfect loop nests. Distribution lets us split apart complex loop nests to create perfect nests of parallel constructs, which can then be parallelised efficiently with OpenMP.

Loop distribution requires the construction of a dependency graph, which is defined below.

Definition 14 (Dependency graph) *A dependency graph of a loop is a directed graph in which the nodes correspond to the statements of the loop nest and the edges correspond to dependencies. An edge is directed (points) from the source to the sink of the dependency, and is annotated with the direction corresponding to that dependence.*

In the case when the loop contains another inner loop, then the inner loop is represented as a single statement that conservatively summarises the behavior of all the statements of the inner loop.

The dependency graph of a loop can be used to characterise its parallel behavior:

Theorem 3 (Dependency cycle) *A loop is parallel* if and only if *its dependency graph does not have cycles.*

If the loop contains a cycle of dependencies, then it necessarily exhibits at least a cross iteration dependency (needed to form the cycle), and thus the loop is not parallel. The following theorem specifies how the transformation can be implemented:

Theorem 4 (Loop distribution) *Distributing a loop across its statements can be performed in the following way:*

1. *The dependency graph corresponding to the target loop is constructed.*

2. *The graph is decomposed into strongly-connected components (SCCs)*[2]*, and a new graph G' is formed in which the SCCs are nodes.*

3. *The loop can be safely distributed across its strongly-connected components, in the graph order of G'. Assuming a number k of SCCs, this means that the result of the transformation will be k loops, each containing the statements of the corresponding SCC. Inside an SCC, the statements remain in program order, but the distributed loops are ordered according to G'.*

4. *Array expansion (section 7.2.1) must be performed for the variables that*

   - *are either declared inside the loop or overwritten in each iteration (output dependencies),* **and**
   - *are used in at least two strongly-connected components.*

The theorem above says that the statements that are in a dependency cycle must remain in (form) one loop (which is sequential by theorem 3). As such, the loop can be distributed across groups of statements corresponding to the strongly connected components (SCC) of the dependency graph. If the graph has only one SCC then it cannot be distributed. The resulting distributed loops are written in the order dictated by the graph of SCCs. We demonstrate theorem 4 on the simple code example presented below:

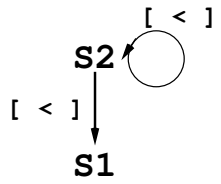
```
do i = 2, N:
  S1 : A[i] = B[i-2] ...
  S2 : B[i] = B[i-1] ...
```

The code has two dependencies:

$S_2 \rightarrow S_1$**:** In order for a dependency on `B` to exist the read from `B` in iteration i_1 of S_1 and the write to `B` in iteration i_2 of S_2 must refer to the same location. Hence $\texttt{i}_1\texttt{-2} = i_2$, which means $\texttt{i}_1 > \texttt{i}_2$, hence S_2 is the source, S_1 is the sink and the direction vector is `[<]`;

$S2 \rightarrow S2$**:** similarly, there is a dependency between the read from `B` in S_2 and the write to `B` in S_2 of direction vector `[<]`.

The dependency graph is thus:



[2] A graph is said to be strongly connected if every vertex is reachable from every other vertex, i.e., a cycle. It is possible to find the strongly-connected components of an arbitrary directed graph in linear time $\Theta(V + E)$, where V is the number of vertices and E is the number of edges.

and it exhibits two strongly-connected components: one formed by statement S_2 and one formed by statement S_1. Loop distribution results in the following restructured code:

```
do i = 2, N:
  S2 : B[i] = B[i-1] ...
do i = 2, N:
  S1 : A[i] = B[i-2] ...
```

in which, according to the graph order, the loop corresponding to statement S_2 appears before the one corresponding to statement S_1. Note that this does not match the program order of statements S_1 and S_2 in the original program. Also note that the first loop is *not* parallel because the SCC consisting of S_2 has a (dependency) cycle, but the second loop is parallel because the SCC corresponding to S_1 does not have cycles.

If a loop is parallel then it can be straightforwardly distributed across its statements in program order because:

- by theorem 3, the loop dependency graph has no cycles and thereby each statement is a strongly connected component;
- the program order naturally respects all dependencies.

Corollary 3 (Parallel loop distribution) *A parallel loop can be directly distributed across each one of its statements. The resulting loops appear in the same order in which their corresponding statements appear in the original loop.*

7.2.1 Array expansion

Finally, it remains to demonstrate array expansion, mentioned in the fourth bullet of theorem 4. Assume the slightly modified code:

```
float tmp;
do j = 0, N-1:
      i = j + 2
  S1 : tmp = 2 * B[i-2]
  S2 : A[i] = tmp
  S3 : B[i] = tmp + B[i-1]
```

Statements S_1 and S_3 are in a dependency cycle, because there is a dependency $S_3 \rightarrow S_1$ with direction $<$ caused by the write to and the read from array `B`, and a dependency $S_1 \rightarrow S_3$ with direction $=$ caused by `tmp`. Statement S_2 is not in a dependency cycle, but there is a dependency $S_1 \rightarrow S_2$, and hence its distributed loop should follow the distributed loop containing S_1 and S_3. If we do not perform array expansion, the distributed code:

```
float tmp;
do j = 0, N-1:
      i = j + 2
  S1 : tmp = 2 * B[i-2]
```

```
  S3 : B[i] = tmp + B[i-1]
do j = 0, N-1:
      i = j + 2
  S2 : A[i] = tmp
```

does not respect the semantics of the original program because the second loop uses the same value of `tmp`—the one set by the last iteration of the first loop—while the original loop writes and then reads a different value of `tmp` for each iteration. We fix this by performing array expansion for `tmp`, which means that we must expand it with an array dimension equal to the loop count and replace its uses with corresponding indexing expressions of the expanded array. This results in the following *correct* code:

```
float tmp[N];
do j = 0, N-1:
      i = j + 2
  S1 : tmp[j] = 2 * B[i-2]
  S3 : B[i] = tmp + B[i-1]
do j = 0, N-1:
      i = j + 2
  S2 : A[i] = tmp[j]
```

Array expansion requires us to normalise the loop first—this means rewriting the loop such as its index starts from 0 and increases by 1 each iteration. This is why we have not written our example as **do** `i = 2, N+1`.

7.3 Eliminating false dependencies (WAR and WAW)

Anti and output dependencies are often referred to as *false* dependencies because they can be eliminated in most cases by copying or privatisation operations:

- Cross-iteration anti dependencies (WAR) typically correspond to a read from some original element of the array—whose value was set before the start of the loop execution—followed by an update to that element in a later iteration. As such, this dependency can be eliminated by copying (in parallel) the target array before the loop and rewriting the offending read access inside the loop such that it refers to the copy of the array.

- Cross-iteration output dependencies (WAW) can be eliminated by a technique named privatisation (or renaming), whenever it can be determined that *every read access* from a scalar or array location *is covered by an update* to that scalar or memory location that was previously performed *in the same iteration.* Semantically, privatisation moves the declaration of the offending variable inside the loop, because it has been already determined that the read/used value was produced earlier in the same iteration.

- Reasoning based on direction vectors is limited to relatively simple loop nests; for example it is difficult to reason about privatisation by means of direction vectors.

7.3.1 Eliminating WAR dependencies by copying

Consider the simple C code below which rotates an array in the right dimension by one:

```
float tmp = A[1];
for (int i=0; i<N-1; i++) {
    A[i] = A[i+1]; // S1
}
A[N-1] = tmp;
```

The loop exhibits a cross-iteration anti dependency (WAR) $S_1 \rightarrow S_1$ (with direction vector `[<]`), and, as such, it is not safe to execute it in parallel. However, one can observe that the reads from `A` inside the loop correspond to the original elements of `A` before the loop, because they are rewritten in a later iteration. As such one can perform a copy of `A` before the loop, and replace the read access inside the loop to operate on the copy of array `A`. This preserves the original loop semantics and results in a parallel loop because the read and write accesses operate on different arrays, hence a dependency cannot occur. For example, using OpenMP:

```
float Acopy[N];
#pragma omp parallel for
for(int i=0; i<N; i++)} {
    Acopy[i] = A[i];
}
tmp = A[1];
#pragma omp parallel for
for (int i=0; i<N-1; i++) {
    A[i] = Acopy[i+1];
}
A[N-1] = tmp;
```

Note that in a real program, we would allocate the `Acopy` array with `malloc()` rather than creating a potentially very large stack allocation.

7.3.2 Eliminating WAW dependencies by privatisation

Consider the contrived and ugly looking C code below:

```
int A[M];
for (int i=0; i<N; i++){
  for (int j=0, j<M; j++) { // writes slice A[0:M-1]
    A[j] = (4*i+4*j) % M;   // S1
  }
```

```
  for (int k=0; k<N; k++) { // reads A[j] where j∈{0,...M-1}
                            // because % denotes modulus op
    X[i,k] = X[i,k-1] * A[ A[(2*i+k)%M] % M];      // S2
  }
}
```

Analysing the cross-iteration dependencies of the outer loop, one can observe that there are frequent output dependencies $S_1 \rightarrow S_1$ of all directions ($*$), because, in essence, all elements of `A` at indices $0 \ldots M-1$ are (over)written in each iteration of the outer loop. This also causes frequent cross-iteration WAR and RAW dependencies between S_1 and S_2 of all directions $*$ because S_2 reads some of the values of `A` which are written in S_1. The read access is also statically unanalysable because the index into `A` depends on a value of `A` (i.e., it is an indirect-array access `A[ A[...] ]`).

It would thus seem that this is a hopeless case and parallel execution is a pipe dream. Not so! Actually the rationale of how to transform the outer loop into a parallel one is quite simple. One may observe that each iteration of the outer loop writes the same indices of `A`, namely the ones belonging to the closed integral interval `[0,M-1]`. One may also observe that S_2 reads from `A` elements whose indices necessarily belong to `[0,M-1]`—due to the two modulus-`M` operations. As such, one may conclude that any value read in S_2 must have been produced in the same iteration of the outer loop (in the inner loop enclosing S_1).

It follows that it is safe to rewrite the loop in the following way:

1. Declare a new variable `A'` of the same dimensions as `A` just inside the outer loop (or equivalently perform array expansion of array `A'` with a new outer dimension of size `N`).

2. Replace all the uses of `A` in the outer loop by uses of `A'`. The resulting loop is safe to execute in parallel because there can be no dependencies on `A'` since each iteration uses a different array `A'`.

3. As a last step, after the parallel execution of the loop terminates, one must copy (in parallel) the elements produced by the last iteration of the outer loop (i.e., `A'[0,...,M-1]` back to `A`.

The parallel OpenMP code that implements these steps is presented below:

```
int A[M];
#pragma omp for lastprivate(A)
for (int i=0; i<N; i++) {
    for(int j=0, j<M; j++) {
        A[j] = (4*i+4*j) % M;
    }
    for(int k=0; k<N; k++) {
        X[i,k]=X[i,k-1] * A[ A[(2*i+k) % M] % M];
    }
}
```

Declaring array `A` as private (by using the clause `private(A)`) would result in semantically performing steps (1) and (2) above. Using the `lastprivate(A)` clause instructs the OpenMP compiler to also perform step (3)—to copy back the privately-maintained result of `A` of the last executing iteration into the globally-declared array `A`.

Please also note that the OpenMP execution will not allocate a new `A'` for each iteration of the outer loop—this is actually equivalent to performing array expansion which is also applicable here—but instead it will *allocate a copy of* `A` *for each active thread*, thus significantly reducing the memory footprint and/or the number of (de)allocations.

Privatisation can be applied whenever one can prove that every read access in an iteration is covered by a previously-performed write access in the same iteration. Privatisation can be implemented by performing either array expansion or moving the declaration of the target variable from outside to inside the loop. However, it saves memory to allocate the private copy per active thread rather than per iteration, which is what OpenMP is doing.

7.4 Loop stripmining, block and register tiling

This section discusses several simple transformations that are going to be combined in various ways to optimise locality of reference (both temporal and spatial locality).

Stripmining refers to the following transformation, which is always safe to apply:

```
for(int i = 0; i<N; i++) {          for(int ii = 0; ii<N; ii+=T){
  iteration body              ⇒     for(int i=ii, i<min(ii+T,N); i++)
}                                         iteration body
                                    } }
```

In essence, a normalized loop is split into a perfect nest of two loops, in which the first loop goes with stride `T`, and the second one goes with stride `1`. Please notice that the resulting loop nest executes the same number of statements and in the same order as the original loop.

Block tiling refers to the transformation that stripmines several consecutive innermost loops in a perfect loop nest—named $l_{k+1} \dots l_{k+n}$—and then interchanges inwards the resulting loops of stride 1. The transformation is valid/safe if in the original program it is safe to interchange any of the loops $l_{k+i},\ i \in \{1, \dots, n-1\}$ in the innermost position. For example, the code below demonstrates block tiling a perfect loop nest of depth two:

```
for(i = 0; i<N; i++) {          for(ii=0; ii<N; ii+=T1) {
  for(j = 0; j<M; j++) {          for(jj=0; jj<M; jj+=T2) {
    iteration body          ⇒     for(i=ii; i<min(ii+T1,N); i++) {
  }                                   for(j=jj; j<min(jj+T2,M); j++) {
}                                       iteration body
                                  } } } }
```

Unroll and jam refers to the transformation that partially unrolls one or more of the outer loops in a perfect nest and then fuses ("jams") the resulting loops. Equivalently, one can stripmine an outer loop, then interchange (distribute) it in the innermost position, then completely unroll it. The transformation is aimed at decreasing the number of memory loads and stores by storing to and reusing values from registers, and thus it is applied when the original loop nest contains data references that allow for temporal reuse—e.g., their indexes are invariant to some of the loops in the nest. Due to this, it is also known as "*register tiling*". We demonstrate the transformation on the matrix-matrix multiplication code below:

```
for(i=0; i<N; i++) {
  for(j=0; j<M; j++) {
    float c;
    c = 0.0;
    for(k=0; k<N; k++) {
      c += A[i,k] * B[k,j];
    }
    C[i,j] = c;
  }
}
```

The plan is to stripmine the loop of index `j` by a tile of size 2, and to interchange it to the innermost position, while performing the necessary loop distribution and array expansion:

```
for(i=0; i<N; i++) {
  for(jj=0; jj<M; jj+=2) {
    float cs[2];
    for(j=jj; j<min(jj+2,M); j++) {
        cs[j-jj] = 0.0;
    }
    for(k=0; k<N; k++) {
      for(j=jj; j<min(jj+2,M); j++) {
        cs[j-jj] += A[i,k] * B[k,j];
    } }
    for(j=jj; j<min(jj+2,M); j++) {
        C[i,j] = cs[j-jj];
    }
} }
```

One can observe that the access `A[i,k]` is invariant to its immediately contained loop of index `j` and thus it can be hoisted outside it and saved into a register. Then the loops of index `j` can be unrolled, and array `cs` can be scalarized as well:

```
for(i=0; i<N; i++) {
  for(jj=0; jj<M; jj+=2) {
```

```
    float c1, c2;
    if (jj   < M) c1 = 0.0;
    if (jj+1 < M) c2 = 0.0;
    for(k=0; k<N; k++) {
      float a;
      a = A[i,k];
      if (jj   < M) c1 += a * B[k,jj  ];
      if (jj+1 < M) c2 += a * B[k,jj+1];
    }
    if (jj   < M) C[i,jj  ] = c1;
    if (jj+1 < M) C[i,jj+1] = c2;
} }
```

In the resulted code, the accesses to the elements of `A` have been halved. We can similarly apply unroll and jam for the loop of index `i` with a tile size equal to `3`. This will cut down the accesses to `B` by a factor of 3. The resulted code is shown in fig. 7.1.

```
for(ii=0; ii<N; ii+=3) {
  for(jj=0; jj<M; jj+=2) {

    float c11, c12, c21, c22, c31, c32;

    if (ii   < N && jj   < M) c11 = 0.0;
    if (ii+1 < N && jj   < M) c21 = 0.0;
    if (ii+2 < N && jj   < M) c31 = 0.0;
    if (ii   < N && jj+1 < M) c12 = 0.0;
    if (ii+1 < N && jj+1 < M) c22 = 0.0;
    if (ii+2 < N && jj+1 < M) c32 = 0.0;

    for(k=0; k<N; k++) {

      float a1, a2, a3, b1, b2;

      if (ii   < N) a1 = A[ii  ,k];
      if (ii+1 < N) a2 = A[ii+1,k];
      if (ii+2 < N) a3 = A[ii+2,k];
      if (jj   < M) b1 = B[k,jj  ];
      if (jj+1 < M) b2 = B[k,jj+1];

      if (ii   < N && jj   < M) c11 += a1 * b1;
      if (ii+1 < N && jj   < M) c21 += a2 * b1;
      if (ii+2 < N && jj   < M) c31 += a3 * b1;
      if (ii   < N && jj+1 < M) c12 += a1 * b2;
      if (ii+1 < N && jj+1 < M) c22 += a2 * b2;
      if (ii+2 < N && jj+1 < M) c32 += a3 * b2;
    }

    if (ii   < N && jj   < M) C[ii,  jj  ] = c11;
    if (ii+1 < N && jj   < M) C[ii+1,jj  ] = c21;
    if (ii+2 < N && jj   < M) C[ii+2,jj  ] = c31;
    if (ii   < N && jj+1 < M) C[ii,  jj+1] = c11;
    if (ii+1 < N && jj+1 < M) C[ii+1,jj+1] = c21;
    if (ii+2 < N && jj+1 < M) C[ii+2,jj+1] = c31;
  }
}
```

Figure 7.1: Result of unroll-and-jam applied to matrix-matrix multiplication, where the first and second outer loops were tiled with sizes 3 and 2, respectively. The number of accesses to `A` and `B` has been reduced by a factor of $2\times$ and $3\times$, respectively, at the expense of introducing some conditional statements.

Bibliography

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery.

[2] John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[3] Ken Kennedy and John R Allen. Optimizing compilers for modern architectures: a dependence-based approach. 2001.

[4] B. Lu and J. Mellor-Crummey. Compiler Optimization of Implicit Reductions for Distributed Memory Multiprocessors. In *Int. Par. Proc. Symp. (IPPS)*, 1998.

[5] Cosmin E. Oancea and Lawrence Rauchwerger. Scalable conditional induction variables (civ) analysis. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 213–224, Washington, DC, USA, 2015. IEEE Computer Society.