An Introduction To Parallel Programming

Dimitri Stanojevic

July 2014

Contents

1 Assembly Programming

Before we dive into more high-level languages like C, Java, Python, etc. I want to first go over the root of computer programming, the programming language closest to machine instructions, Assembly. Nowadays Assembly is not really intended for actual programming of larger software because its very unintuitive and extremely hard to read. Also it is very hard to express even simple statements like printing some letters onto the screen. However Assembly is what's actually under the hood of any other programming language that is compiled to machine code. This is because Assembly in fact is a one to one translation from machine code to human readable text. This simply means that we can read and understand single assembly instructions easily. The problem with Assembly programs however is that even simple programs need lots of lines, where the instructions look all more or less the same. This chapter is therefore coupled with the chapter about computer hardware.
It's important to remark that the assembly language of every computer system has some differences. Therefore Assembly is not a programming to itself. It is rather a kind of programming languages. In this chapter we will look at Intel x86 assemlby as it is todays most popular computer architecture for desktop and laptop computers. Mobile devices often use AMD chips, which are more energy efficent and have a reduced instruction set. So if you write an x86 Assembly program, it will probably not run on other processors as they may not have the instructions you used in the program.
Also in x86 Assembly there is a difference between 32-bit and 64-bit architectures, because 64-bit architectures support instructions that 32-bit architectures don't. An obvious example would be summing up two 64-bit values.

1.1 A short introduction of what you should know about hardware

We start with the fact that the Assembly language has a very limited set of instructions. You can think of these as the most basic things your computer can do. Within an Assembly program you specify a sequence of such instructions and when your computer runs the proram, it will read and evaluate one instruction after the other.
To understand what these instructions do, we need to have an abstract view of where the computer stores its numerical values. First the processor itself has a few memory units and can hold about 20 values close to itself. We call these memory slots *registers* Then there is what we call *main memory* or *RAM (random access memory)*. The main memory can hold maybe 4GB of values that can be accessed by the processor at any

time. The main memory is further away from the processor and looking up values from the main memory takes much more time than looking them up in one of the processors registers.
Now a processors instructions are basically ways to manipulate values that are stored either in the registers or in main memory. The most basic instructions are moving a value from main memory into a register, moving a value from a register back to main memory, adding two values together, subtracting a value from another, etc.
When we want to express such an instruction concerning registers or values in main memory, we need a way to specify which register or which location in memory we mean.

As there are very few registers inside a processor, we have individual names for them: %eax, %ebx, %esp, etc. The '%' sign is just part of the syntax to specify a register. Some registers have a special use and we will cover that later.

To address the main memory, we use addresses which are simply numbers ranging from zero to the amount of memory we have. In 32-bit Assembly an address is a 32-bit value. Every address specifies a certain byte in memory. If you think about it, you will see why you cannot use more than 4GB of RAM in a 32-bit Computer. Therefore in 64-bit Systems, every address is a 64-bit value which leads to a much much bigger address space.

Just before we go on with the specific registers an instructions, I want to show you a short example assembly program so you get a feeling of what we are talking about. You don't have to be able to read the program yet, but you may come back and look at it from time to time as you read more about registers and instructions. Maybe you can already guess what it does. Tipp: The end result is stored in the register %eax.

```
            movl  $1, %ebx
            movl  $0, %eax

            movl  $0, %edi
            movl  $10, %esi

.Lcomp:
            cmp %esi, %edi
            jg  .Lend

            movl  %eax, %ecx
            addl  %ebx, %eax
            movl  %ecx, %ebx

            addl  $1, %edi

            jmp  .Lcomp
```

```
    .Lend:
```

1.2 An overview of the x86 registers

The x86 processor has eight general purpose registers, each being able to hold a 32-bit value. General purpose means, that we are able to address these registers in our assembly program. However only six of those registers are actually used for storing intermediate values. The other two have a special purpose and should not be overwritten with other values. The registers are called:
%eax, %ebx, %ecx, %edx, %esi, %edi, %esp, and %ebp
The last two registers %esp and %ebp have a special purpose.
%esp is called the *stack pointer*. It points to the lower end of the *stack*. The stack is the region in memory where a program stores temporary values in the case that all six registers are already occupied. %ebp is called the *base pointer* and also points to some location on the stack. We will cover this a bit later, when we look at the stack.
The other six registers can be used at will, but there are some conventions to be followed when we use them between different functions. %eax for example is usually used to store the return value of a function. We have not introduced functions yet, so when we do this will make more sense.

1.3 The x86 instruction set

In this chapter I want to list all important x86 Instructions.
All instructions start with the name of the instruction followed by one or two operands separated by a comma. These operands can be one of three things:

- *Register* A register is denoted with its name, for example %eax. When we use a register as an operand we actually mean the value that the register holds.
- *Memory location* A register can also hold memory addresses as memory addresses are 32-bit values and registers can hold 32-bit values.
  So assuming that our register %eax holds a memory address, then (%eax) denotes the value that is stored at that address.
- *Number value* If we want so simply secify a number, we can write $$n$ to specify the number n. For example we might want to move the number 0 to the register %eax, so we write movl $0, %eax. %eax then contains the value 0.

mov <op1>, <op2> Moves the value of the left operand to the value of the right operand. The instruction is used to either move a value from the main memory to a register or move the value stored in a register to a location in main memory. You cannot move a value between two memory locations with just one operation. To do that, you would first need to load the value into a register and then move it back to that other location in memory. The following combinations are possible:

```
mov %rax, %rbx            # moves from register to register
mov %rax, (%rbx)          # moves from register to memory
mov (%rax), %rbx          # moves from memory to register
mov $42, %rbx             # moves number to register
mov $42, (%rbx)           # moves number to memory
```

add <op1>, <op2> Adds the value of the left operand to the value of the right operand and stores it into the location provided by the right operand. The following combinations are possible:

```
mov %rax, %rbx            # adds register to register
mov %rax, (%rbx)          # adds register to memory
mov (%rax), %rbx          # adds memory to register
mov $42, %rbx             # adds number to register
mov $42, (%rbx)           # adds number to memory
```

sub <op1>, <op2> Subtracts the value of the left operand from the value of the right one and stores it into the location given by the right operand. This works analogously to to the add instruction.

lea (%rax), %rbx The *lea* instruction takes the address stored in %rax instead of its value. This may seem redundant at first, as we already have this address. However *lea* allows us to do certain address calculations on the fly like adding an offset to the address. For example
lea 4(%rax), %rbx
treats the value at %rax as an address, adds 4 to it and stores it back to %rbx.

push %rax This *push* instruction is an equivalent to
1) subtracting 4 from the stack pointer and
2) mov %rax to the memory location the stack pointer points to.
You can...
push %rax – ...push the value of a register
push (%rax) – ...push the value of a memory location
push $42 – ...push a constant value

pop %rax, %rbx *pop* behaves like the inverse to *push.* It
1) moves the value the stack pointer points to to %rax and then

2) increments the stack pointer by 4 You can... pop $\%rax$ – ...pop to a register or pop $(\%rax)$ – ...pop into memory

call %rax Call pushes the value of the instruction pointer onto the stack. Then it sets the instruction pointer to point to address given by the operand. This operand can also be a label.

1.4 Memory

1.5 Program Flow

1.6 Calling Conventions

2 Sequential Programming

This first chapter will give you an introduction to the most common style of programming that you will probably encounter towards the beginning of every computer science Bachelor. Languages that allow sequential programming are Assembly, C, C++, C#, Java, Python and many more. The concepts that are explained in this chapter form the basis of the most important things you need to know to start learning programming languages and other more interesting programming concepts. On the other hand if you already know a programming language like C or Java, you may carefully skip some of the sections in this chapter.
Sequential programming is so popular because it stays close to the way computers work. Computer generally have a sequence of instructions that they execute in a specified order and a sequential program defines that sequence. This is usually very intuitive as you can think of the first instruction always being finished before the second one starts executing. I want to give a simple example program to give you the intuition.

```
int x;              // declares a variable x of type int
x = 1;              // x gets the value 1
x = x + 2;          // x gets the value x+1 (=3)
x = x * 3;          // x gets the value x*3 (=9)
x = x / 3;          // x gets the value x/3 (=3)
x = x - 2;          // x gets the value x-2 (=1)
print(x);           // the method print is called with x
```

The '=' sign denotes an assignment, meaning that the value on the right side will get assigned to the variable on the left. The ';' (semicolon) denotes the end of an instruction. '//' marks the beginning of a comment. The comments in this example show which value the variables have after the corresponding instruction has been executed.

The important thing about this example is that the order of the operations is strict and cannot be changed by the computer. Eventually we will see that this is not entirely true when dealing with parallel programs (,which are programs that do several computations concurrently). But as long as were only talking sequentially, we can assume that the order of instructions always holds.

2.1 Important Programming concepts

2.2 Variables and Assignments

Variables are the most important concept in programming. A variable simply is a placeholder for a value. As we have seen in the chapter about Assembly programming, a modern processor can only store about six values at the same time. These six storage locations are called registers and each one has a specific name. If at some point we need more than six storage locations, we need to manually move values onto main memory. This is so inflexible and error prone that it makes programming a nightmare.
Variables are a very nice abstraction for storage locations. We can simply declare variables as we wish and store values into them. We don't have to care about how and where those values are actually stored cause the underlying system does that for us. We can also chose meaningful names for our variables like, "result", "index" or "size" to make the program more readable.

An assignment is a statement in which we store a value into a variable. Assignments in Java might look like this:

```
x = 2;            // x gets the value of two
x = x + 2;        // x gets the value of x + 2 (=4)
x = y + 42;       // x gets the value of y + 42
```

The "=" is used in many languages for example C, C++, C#, Java, Python. Some languages for example Pascal or Eiffel use the symbol ":=" instead. The left hand side of an assignment must be a variable. The right hand side is an expression that evaluates to a value. This might for example be a number, another variable or a calculation that contains those.

2.3 Data Types

Throughout our program we might have values of different *types*, for example numbers (called *integers*) or truth values (called *booleans*). When these values get stored in a variable, that variable gets associated with the type.
The type of a variable is the *kind of data* that the variable stores. So you can think of a type as a category or format of information. The most popular examples would be numbers, characters, arrays, etc.. Types are very important for program correctness because when we operate on variables we want to make sure that the kind of operation we apply actually conforms with the type of the variable. For example numbers can be added together, texts cannot. We can calculate the n'th fibonacci number assuming that n is natural number. If n is a rational number, we have a problem.
Therefore in many programming languages variables have a strictly defined type that cannot change during the rest of the program. So you cannot have a variable x that first

holds a number and then later holds a string. The type of the variable x needs to be known from the beginning and x is only allowed to hold values of that particular type. Languages that enforce this rule are called *statically typed.*

So whenever we declare a variable, we need to state its type. In C or Java for example this looks like this:

```
int x;            // x will hold integers
boolean y;        // y will hold truth values
float z;          // z will hold floating point numbers
```

Now in a statement like

```
x = y + z
```

the compiler will alert us that

1. it is not possible to sum together variables of type boolean and float and that
2. you cannot assign a value that is not an integer to a variable that holds integers

We will see a lot more interesting stuff about types in object oriented programming, where our data chunks will be called objects, and these objects will have types.
Also later in the chapter about formal methods we will construct proofs for showing that certain expressions conform to valid types.

2.4 Control Flow - IfElse, Switch statements

Often in our programs we will want to tell our programm what to do next, based on the state of some variables. For example if the user has pressed a button, we want it to do something and othwerise we want it to do nothig.

2.4.1 IfElse

If the variable a is bigger thant 5, then we want our program to do this, and otherwise we want it to do something else. Every programming language has constructs for modelling this kind of control flow. In sequential languages we often use the IfElse construct. It looks like this:

```
if (a > 5) {
        // code to be executed if a > 5
}
else {
        // code to be executed otherwise
}
```

The brackets following the **if** keyword contain the condition of the IfElse statement. The condition may be an arbitrary large boolean expression, so it is always evaluated to either true or false. The following code surrounded by curly brackets is called the **then** block. After the **else** keyword follows another block, called the **otherwise** block. Those blocks can contain an arbitrary number of statements and even other ifElse constructs. Depending on the language you use, variations of the ifElse statement are allowed such as leaving away the else part or the following consturuct:

```
if (a == 1) {
        // code to be executed if a == 1
}
else if (a == 2) {
        // code to be executed if a == 2
}
else if (a == 3) {
        // code to be executed if a == 3
}
else{
        // code to be executed otherwise
}
```

2.4.2 Loops

Often you will want to execute the same line of code multiple times or you will want to execute it as long as some condition is fulfilled. This is accomplished by the **while** statement. In the following example we want to sum up all number between 1 and 100:

```
int result = 0;
int i = 1;
int n = 100;
while (i <= n) {
        result = result + i;
        i = i + 1;
}
```

We call i the iterator. It gets initialized with the value 1 and the loop stops when i reaches a value that is greater than n. It is important that we increment the value of i at every loop iteration. If we do not, the loop will never terminate and our program will be stuck forever.
There is another variant of this construct found in many languages called the **for loop**. It is essentially just so called **syntactic sugar**. Syntactic sugar is a term that we use for syntax in a programming language that doesn't add anything new to the language but allows us to write something in a 'nicer' way. The previous example can be rewritten as:

```
int result = 0;
int n = 100;
for (int i = 1; i<= n; i = i + 1) {
        result = result + i;
}
```

You can see the same statements from the while construct appear simply at a different place. Despite the fact that this is semantically the same code as before, it has several advantages:

- The scope of the variable i now lies only within the loop body. This means that the variable is not visible from outside the loop and the programmer will not accidentally mix it up the later in the program with another variable called i.
- The statement which increments the loop variable has a specific place in our code and will therefore not be forgotten as easily. Seriously, forgetting to increment the iterator variable seems like a small mistake, but it happens quite often and can lead to a lot of frustration if the error is not detected.

Some older languages feature a **goto** keyword with which you could jump from one position of your code to any other. The use of this instruction is highly discouraged because it can lead to very complicated code.

2.4.3 Switch statement

The switch statement can be seen as a replacement for the nested ifElse statements we have seen earlier. Imagine you have just one variable a and depending on its value you want to do different things in your code. Before we had several ifElse statements covering the different values of a. This is a lot of unnecessary code to write for such a simple thing we want to achieve. The switch statement is more intuitive:

```
switch (a) {
case 1:
        // code to be executed if a == 1
        break;
case 2:
        // code to be executed if a == 2
        break;
case 3:
        // code to be executed if a == 3
        break;
default:
        // code to be executed otherwise
}
```

Here the variable in brackets after the **switch** keyword is the variable for which we want to do the case distinction. Depending on its value the program execution jumps to the corresponding line. The important thing to note is that the program will execute all the following cases too if it is not stopped by a **break** statement. The break statement simply tells the program to jump to the end of the switch statement. If none of the cases match the value of the variable the code behind the **default** keyword will be executed.

2.5 Methods

Methods are a way to modularize our program. We can write code that does a specific thing as a method and then call it from within another method! Here is an example:

```
void main(){
        int userInput = getUserInput();
        if(inputValid(userInput)){
                processInput(userInput);
        }
}

boolean inputValid(int input){
        if(input < 100){
                return false;
        }
        else if (input < 200){
                return true;
        }
        else{
                return false;
        }
}

void processInput(int input){
        // some code
}
```

This program just asks the user for some input, checks whether it is in the range between 100 and 200 and then processes it. ‘getUserInput’, ‘inputValid’ and ‘processInput’ are all methods that we use to better separate our code. Imagine if everything would be in a single block of code. Our program would get unreadable very fast.
As you see methods have **arguments** and **return types**. Arguments are values that we pass on when we call this method. A return value is what a methods returns. If a method has nothing to return its return type is marked as **void**.

2.6 Data structures

As far we have heard only of integer and boolean types. It would be cool if we could use these basic types to define other types to use for various applications. Also so far we have only dealt with a limited number of values at the same time so we had no problems storing them in local variables. What if we wanted to store several hundreds or thousands of values in our memory and still be able to access the elements that we need? We need to think about suitable data structures.

2.6.1 Arrays

Arrays are the most commonly used data structure to achieve this. There are other data structures more suitable for some tasks but from the way that our computer's memory works, arrays are the most intuitive way to store a larger amount of data. Imagine our computer's memory as number of continuous storage locations while each location is accessible through a corresponding address. An array is simply a reserved sequence in this memory where we can continuously store our values. When we declare an array, we have to tell the computer how large this sequence should be, eg how many values we will want to store. In programming terms we simply think of an array as a number of elements, each stored at a certain position called the **index**.
In Java an array might be used like this:

```
int[] sum = new int[101];
sum[0] = 0;
for (int i=1; i<=100 ; i=i+1) {
        sum[i] = sum[i-1] + i;
}
```

At the end of the loop the sum array should contain at every index i the sum of the numbers 1 to i. I initialized 101 values because the first index of an array in Java is 0. So here our array contains 101 elements, while sum[0] is our first value and sum[100] our last one.
It's important to note that arrays cannot dynamically change their size. This means that we have to specify the size of our array when we create it and cannot change it afterwards. In case we do not know, how many elements we want to store in our array, we should maybe use a different data structure.

2.7 Some important terminology

2.7.1 Compile Time vs Runtime

When we look at a program before it is being run, we call this the compile time. Even if some programs like for example Java programs are not really compiled, we still talk about compile time when we examine the program prior to running it. When we run a program we speak of runtime.
This distinction is important because we have different knowledge about a program when it is being run than before. For example we might only know the value of certain variables that are implicitly defined in our program. We do not know values that come from the outside like user inputs or data that the program will process.
The compiler of any language will try to detect errors in your code and report them to you before you run the program. The errors that it finds during compilation are called compile time errors. If your program crashes during execution the error will most probably be a runtime error, an error that could not have been detected before running your program.

An example would be the access of non-existing elements in an array, also called and **out-of-bounds** exceptions. Assuming we have an array of ten elements:

```
int[] A = new int[10];
```

The following out-of-bounds error will be detected by your compiler during compile time:

```
A[13] = 42;
```

The following code might lead to a runtime error:

```
int a = getUserInput();
A[a] = 42;
```

3 Object Oriented Programming

Object oriented programming a very popular programming concept that is being used a lot in practice, especially when designing larger software systems. The main idea behind object oriented programming is to define everything in terms of objects and how they interact with each other. You can think of an object as a well defined chunk of data that also has some functionality defined.
We've already seen that we can define our own data chunks in programming languages that are not object oriented. The difference here is that objects do have functionality attached to them, and structs don't.

3.1 Objects and Classes

I'd like to give a short example right here at the beginning. The following is a definition of an object in the programming language Java. These definitions are called classes. Once we have defined such a class we can build infinitely many objects out of it. So you can think of a class as the definition of a certain type of objects. All objects of the class Door *look* and *behave* the same way, namely exactly the way they have been defined in the class description.

```
class Door{
        // data that the object will contain
        private boolean is_open;
        private Color color;

        // functionality of that object
        public Door(){
                is_open = false;
                color = Color.WHITE;
        }

        public void open(){
                is_open = True;
        }

        public void close(){
                is_open = False;
        }
        public void set_color(Color new_color){
                color = new_color;
        }

        public void get_color(){
                return color;
        }
}
```

If at this point you have some trouble reading the above code, I would advise you to first read the chapter about sequential programming or even some parts of the Java introduction chapter. I will however shortly explain the pieces that are relevant for object oriented programming.
You notice that the function *Door* starts with a capital letter and has the same name as the class its defined in. In Java this means that *Door* is a constructor of this class. A constructor is a special function that defines how an object is initialized or in other words, what happens at the moment when an object of that class is initialized. It is called when an object is being created. In our case *is_open* is set to False at the object initialization process. Semantically this means that we can only create Doors that are initially closed.
We can now create objects of type Door in our program. I'll also call them *doors* from now as they are meant to be seen as abstractions of actual doors. When we have created a door, we know that:

- the door is either closed or open but initially it is closed
- the door has a certain color which initially is white.
- the door can be opened with the open() function.
- the door can be closed with the close() function.
- the door's color can be set with the set_color(Color c) function.
- and we can get the door's current color with get_color().

Just like *int* and *boolean*, Door is also a Type. Java makes a destinction between *primitive types* like boolean or int and *object types* like Door and Color that are defined in classes. However we will not make this distinction in this chapter as it is rather Java specific. In a pure object oriented language everything is an object, even so called primitive types. Java in that sense is not purely Object Oriented, but will ignore these details and address them in the chapter about Java programming.

3.2 Polymorphism and Inheritance

4 Parallel Programming

4.1 Basic Terminology and Concepts

4.1.1 Concurrency and Parallelism

There are two important terms in parallel programming that are often mixed up. The terms *parallel programming* and *concurrent programming* seem to mean the same but are used differently.
Concurrent programming is when a program is designed in a way that two or more computations can be run at the same time and thus their sequences of operations overlap each other. Overlapping means that if we have one computation with the instructions a_1, a_2, a_3 and the another one with the instructions b_1, b_2, b_3, in this order, the resulting sequence of instructions could be $a_1, b_1, a_2, b_2, a_3, b_3$ or $a_1, a_2, b_1, a_3, b_2, b_3$ or any other overlapping of this kind. In fact, the major problem with concurrent programs is that we can't know for sure how the sequences will overlap.
We call these independently running computations *threads.* If a concurrent program is run on a machine with multiple execution units - for example a processor with several cores or a graphics card - threads can be distributed among the cores and therefore be run at the same time, which we then call *parallel.* So a parallel program is a concurrent program which is run on multi-core machine. Or in other words, a concurrent program is designed with parallelism in mind, while a parallel program is a concurrent program that is actually being executed on a multi-core machine.
An important benefit of concurrent programs is obviously that it can be run much faster, because the work is distributed. But what is the advantage of a concurrent program when it is run on machine that has only a single core?
When we run a concurrent program on a single core then only one thread at the time can execute its code. The threads then change turns in when they are allowed to execute their code. This happens at the order of milliseconds such that it almost appears as if the threads were running in parallel. It is similar to the way an operating system allows you to run several programs at the same time. The mechanism that decides which thread when gets its turn is called *scheduler.*
This is were the advantage of concurrency lies: Several computations can be run concurrently without blocking each other. For example imagine a 3D animation software with the ability to edit and then render 3D animations to videos. Rendering is a very heavy computation which probably takes more than a few milliseconds. If the software was not designed concurrently then the whole program would freeze during the rendering period because the software would be busy rendering and could not process user input. A concurrent implementation would have two independently running threads, one for the

rendering process and one for checking user input. Thus the program could still accept user input and would not freeze.

4.2 Performance

When we want to make a computation run faster on a certain machine we can think of several ways to do so. One would be to design a better algorithm, one that has a better complexity. However with certain problems, we believe that it is not possible to find better algorithms. For example it is obvious that you cannot sort a list of n elements without at least looking at all n elements. A sorting algorithm needs at least n steps to sort the list of n elements.
From an engineering point of view we could try to optimize the hardware of that machine to make it do more instructions in less time. Nowadays processor manufacturers are getting to their limits when trying to build faster processors. There are several reasons for this, one of them being the overheating problem. So instead manufacturers decided to build more cores into a processor. It's what we call a *multi-core processor*, a processor that consists of several cores that can execute code independently of each other. From now on we will call these cores *execution units.*
Lets assume that an algorithm and a machine (with p execution units) are already given. To make the algorithm run as fast as possible we need to equally distribute the computational workload on as many execution units as possible. In the following we want to define a way to measure the performance of our program when it is run on a certain amount of cores:
Let T_1 be the time that the program needs to finish on a machine with only one processor and let T_p be the time that it takes the program to finish when it runs on a machine with p cores. What we would like to have is: $T_p = \frac{T_1}{p}$ which would mean that that the execution time was split up equally among all p cores. $T_p > \frac{T_1}{p}$ would mean that the program doesn't parallelize perfectly or that we have some *overhead* due to parallelization. Overhead is a term for all additional computation that is needed to parallelize the program, for example scheduling the threads. $Tp < \frac{T1}{p}$ would indicate that we gained more performance than theoretically possible (we will later see that this can actually happen due to caching effects).
We want to define the term *speedup* for saying "how much faster" our program is when run in parallel, that is on p processors instead of just one. Speedup is defined as $S_p = \frac{T_1}{T_p}$.
The speedup for a parallel program should desirably be linear to the number of cores ($S_p = p$), such that we would always be able to add more and more cores to our machine and see our program run faster and faster. Sadly in reality this difficult to obtain and we will see why.
Amdahl's Law is a way to theoretically calculate the speedup of a program given its percentages of sequential code (written as s) and parallelizable code ($1 - s$). To make this clear, we assume that a certain part of our code can be parallelized and the other part cannot. Only the percentage that can be parallelized can profit from the multi-core machine. The law says:

$T_p = T_1(s + \frac{1-s}{p})$ which leads to the theoretical speedup of $S_p = \frac{T_1}{T_p} = \frac{p}{1+s(p-1)}$
Amdahl's Law is purely theoretical and will only calculate the maximum possible speedup, ignoring implementation and hardware limitations.
A different approach to reason about speedup is *Gustafson's Law*. It takes the runtime as a fixed value and examines how the speedup behaves when the size of the problem rises. More processors are able to solve larger problems in the same time. It says:
$S_p = p - s(p - 1)$
To highlight the difference between Amdahl's and Gustafson's law we could say: Given more processors, with Amdahl's law we can solve the task faster while with Gustafson's law we can solve more of the task in the same time.
Here are two examples to practice this difference:
1) A computer scientist designed an algorithm to calculate prime numbers. 80% of his program runs in parallel. His goal is to make it calculate the one billionth prime number as fast as possible. How many threads should he create?
2) A company needs to sort very large amounts of data. The sorting algorithm they use can only be parallelized up to 20%. The company currently uses a machine with 4 cores and wants to reason about whether to buy a new machine with 16 equally fast cores.
Reason about the two problems. Would you use Amdahl's or Gustafson's law?

4.3 Parallel Programming Concepts

There are many different methods on how to construct parallel programs. Some of them are still rather theoretical while others are practical and well adapted in modern programming environments. Not every method is well suited for every problem. We will see various concepts and discuss their advantages and disadvantages. But before we do that we need to take a look at some terminology and difficulties that come with parallelism.

4.3.1 The state of a program

The state of a program is given by the objects and variables of the program. When we talk about an object oriented language, it makes sense to say that objects are components of a program and the state of every object is given by its fields (variables and references).
In parallel programming we reason about threads which execute their code independently of each other. A state is called *shared* if more than one thread has access to it.
This is a very important property because whenever a state is shared among threads, things can go wrong and we have to make sure they don't. The thing is: By default there is no coordination between threads. Let's say we have a variable that is shared among two threads: If one thread changes the value of that variable the other thread might not know immediately that the value has been changed.
Here is some Java code to show how such a scenario might look in practice:

```
public boolean free; // a shared variable
public m() // is called by multiple threads
        if(free){
                free = false;
                f(); // may only be called by one thread at a time
                free = true;
        }
}
```

Here $m()$ could be executed by several threads in parallel. The idea of the shared variable variable $free$ and the if-statement is to allow only one thread at the time to execute the method $f()$. If a thread reads the value *true*, it sets $free$ to $false$ such that the next thread knows not to enter the then block of the if-statement. This does not work as expected!!! It is very important to understand why:
Imagine an airport with just one landing stripe. Airplanes that land on this airport must make sure, that the landing stripe is free and no other airplane is using it. The landing stripe is our shared state. Now imagine that airplane A and airplane B both look at the landing stripe at the same time, see that it's empty and initialize their landing sequence. Even if plane A lands before B, and B now sees that the landing stripe is occupied, it will be too late to turn around. These timing problems also happen between threads. Let's say we model the landing stripe as our boolean variable $free$ and $f()$ represents landing. When two threads look up the value of $free$ at the same time, they will both read the *true* and therefore both execute the landing sequence. Setting $free$ to $false$ changes nothing because both threads are already in the then-block. This is what we in general call a *race condition* because the output of the program depends on which thread will "execute faster". This exact timing between threads is something we can impossibly reason about and we therefore have to find other ways to somehow coordinate threads.

If an object can change its state during runtime it is called *mutable* and otherwise *immutable*. Constants in Java for example are immutable while variables are not (hence their name). An immutable object is by itself safe to use with several threads because threads can only read from it but not modify it.

Let's have another example of a race condition problem: Consider a program for a device which detects and counts light rays. The device has several detectors which light up every time they are being hit. Every detector runs on an individual thread which increments a counter every time it detects a light ray. The counting of the light rays could be designed in two ways:

(1) The counter is a shared variable which is visible to all detector threads. It is being incremented every time a detector lights up. If two detectors light up at the same time, the variable is being accessed in parallel. The counter - a mutable object- can be accessed an modified by several threads at the same time.

(2) Every detector thread has its own counter variable. At some coordinated time

when the detectors are off-line one of the threads collects the counter variables of all detectors. This design leads to what is called *isolated mutability* because all mutable states (our counter variables) are only being modified by their corresponding *execution context* (thread).

What do you think are the advantages and disadvantages of the two designs? What's risky about design 1) and how could we remove this risk?

4.3.2 Data-, and Race conditions

Let's discuss the detector example from the previous section and assume the first design choice, namely that the counter is a global shared state where all threads can write to. In its simplest form the counter could be an integer field which contains the amount of times the device has been hit. The detector threads will increment the counter whenever they detect a light ray. Let's look at this operation in detail. To increment a value means to add 1 to its current value. On most machines the increment operation is done the following way:

1. Read the integer value from memory
2. calculate 1 plus this value
3. store the result back into memory

You see that this simple operation of incrementing a value, which in Java can be written as the one line

```
counter ++;
```

actually consists of three operations if we look at it at the machine level.
Say now, two detectors detect a light ray at exactly the same time. The following overlapping would be a possible scenario:

at the moment c contains the value 10

T1: reads the value 10

T2: reads the value 10

T1 and T2: both calculate the value 11

T1: writes back 11

T2: writes back 11

The correct counter value should be 12 but with the above overlapping of T1 and T2, the counter is set only to 11. Although the above scenario seems highly unlikely, its possibility is still quite high considering how many increment operations and how many concurrently operating threads we might have. Such a program must be considered unsafe!
Because the outcome of the program now depends on the relative timing of the threads (which can impossibly be predicted), the problem that occurs is called a *race condition* (the various threads are "racing" to execute their code first). A race condition which happens because of a shared mutable state is called a *data race.* Many (but not all!) cases of race conditions are data races.
There are several mechanisms that allow us to prevent data races by allowing only one thread at a time to have access to the state of a shared object. This is called *synchronization* or *"protecting the state".* However these synchronization mechanisms can make a program run slower. A badly written parallel program could perform even worse than its sequential version. This additional computation needed for synchronization is called *overhead.* Therefore one main concern in parallel programming is to find ways to prohibit race conditions while not decreasing the performance of the main program.