Parallel Processing

Multicore processors are becoming commonplace
They typically consist of several processors, and a shared memory:

The different cores execute different pieces of code in parallel
If several cores do useful work at the same time, the processing becomes faster

Imperative Programming and Parallel Processing

Imperative programming is inherently sequential
Statements are executed one after another
However, imperative programs can be made parallel through parallel processes, or threads:

Side Effects and Parallel Processing

Imperative programs have side effects
Variables are assigned values
These variables may be accessed by several processes
This introduces the risk of race conditions
Since processes run asynchronously on different processors, we cannot make any assumptions about their relative speed
In one situation one process may run faster, in another situation another one runs faster
A Race Condition Example

Two processes \( P_1 \) and \( P_2 \), which both can write a shared variable \( \text{tmp} \):

\[
\begin{align*}
\text{P1:} & \quad \text{P2:} \\
& \quad \text{tmp} = 4711 \\
& \quad \text{z} = \text{tmp} + 3 \\
& \quad \text{tmp} = 17. \\
& \quad \text{y} = 2 \times \text{tmp}. \\
\text{tmp} = 17 & \\
y = 2 \times \text{tmp} &
\end{align*}
\]

The intention of the programmer was that both \( P_1 \) and \( P_2 \) should set \( \text{tmp} \) and then use its newly set value right away. This would yield the following, intended, final state:

\[
\text{tmp} = 17 \text{ or } 4711, \quad \text{y} = 34, \quad \text{z} = 4714
\]

But the programmer missed that the processes have a race condition for \( \text{tmp} \), since they may run at different speed. Here is one possible situation:

\[
\begin{align*}
\text{P1:} & \quad \text{P2:} \\
& \quad \text{tmp} = 17 \\
& \quad \text{tmp} = 4711 \\
& \quad \text{y} = 2 \times \text{tmp}. \\
& \quad \text{z} = \text{tmp} + 3
\end{align*}
\]

Final state: \( \text{tmp} = 4711, \text{y} = 9422, \text{z} = 4714 \)

Wrong value for \( \text{y} \)!

Here is another possible situation:

\[
\begin{align*}
\text{P1:} & \quad \text{P2:} \\
& \quad \text{tmp} = 4711 \\
& \quad \text{tmp} = 17. \\
& \quad \text{z} = \text{tmp} + 3 \\
& \quad \text{y} = 2 \times \text{tmp}.
\end{align*}
\]

Final state: \( \text{tmp} = 17, \text{y} = 34, \text{z} = 20 \)

Wrong value for \( \text{z} \)!

Parallel Programming is Difficult

To avoid these race conditions, the programmer must use different synchronization mechanisms to make sure processes do not interfere with each other.

But this is difficult! It is very easy to miss race conditions.

Debugging of parallel programs is also difficult! Race conditions may occur only under certain conditions, that appear very seldom. It can be very hard to reproduce bugs.

This is a very bad situation, since multicore processors are becoming commonplace. We are heading for a software crisis!

The heart of the problem is the side effects – they allow different processes to trash each others’ data.
Pure functional programs have no side-effects.
The evaluation order does not matter.
Thus, different parts of the same expression can always be evaluated in parallel:

\[ f(x) + g(x) \]

Sometimes called expression parallelism.

Data types like lists, arrays, sequences, are sometimes called *collections*.
Functions like map and fold are called *collection-oriented*.
Collection-oriented functions often have a lot of inherent parallelism.
If one can express computations with these primitives, then parallelization often becomes easy.
In imperative programs the same computations are often implemented by loops. Loops are inherently sequential. A good parallelizing compiler might retrieve some of it, but there is a risk that parallelism is lost.
If \( \text{op} \) is associative, then the expression tree can be balanced.

With sufficiently many processors, parallel \texttt{fold} can be done in \( O(\log n) \) time \((n = \text{no. of elements}) \).

**Fault Tolerance Through Replicated Evaluation**

If an expression has no side effects, then it can be evaluated several times without changing the result of the program.

This can be used to increase the *fault tolerance*: if one processor fails, we can use the result from another one computing the same expression.

### Summing Up

Freedom from side effects simplifies parallel processing a lot.

Collection-oriented operations are also very helpful for this.

It’s the design and thinking that is important – not necessarily that a functional language is used.

The same principles can be applied also when using conventional programming languages.