Sequential, parallel, or parallel execution with vectorisation

We will now learn how to execute our algorithm according to a certain execution policy.

WE'LL COVER THE FOLLOWING

By using an execution policy in C++17, you can specify whether the algorithm should run sequentially, in parallel, or in parallel with vectorization.

Execution Policies

The policy tag specifies whether an algorithm should run sequentially, in parallel, or in parallel with vectorization.

- `std::execution::seq`: runs the algorithm sequentially
- `std::execution::par`: runs the algorithm in parallel on multiple threads
- `std::execution::par_unseq`: runs the algorithm in parallel on multiple threads and allows the interleaving of individual loops; permits a vectorised version with SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata) extensions.

The following code snippet shows all execution policies.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main(){
  std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

```
  // standard sequential sort
  std::sort(v.begin(), v.end());

  // sequential execution
  std::sort(std::execution::seq, v.begin(), v.end());

  // permitting parallel execution
  std::sort(std::execution::par, v.begin(), v.end());

  // permitting parallel and vectorised execution
  std::sort(std::execution::par_unseq, v.begin(), v.end());
  return 0;
}
```

The execution policy

The example shows that you can still use the classic variant of `std::sort` without execution policy. In addition, in C++17 you can specify explicitly whether the sequential, parallel, or the parallel and vectorised version should be used.

> **ℹ Parallel and Vectorised Execution**
> Whether an algorithm runs in a parallel and vectorized way depends on many factors. For example, it depends on whether the CPU and the operating system support SIMD instructions. Additionally, it depends on the compiler and the optimization level that you use to translate your code.
>
> The following example shows a simple loop for creating a new vector.
>
> ```
> const int SIZE= 8;
>
> int vec[] = {1, 2, 3, 4, 5, 6, 7, 8};
> int res[] = {0, 0, 0, 0, 0, 0, 0, 0};
>
> int main(){
>   for (int i = 0; i < SIZE; ++i) {
>     res[i] = vec[i] + 5;
>   }
> }
> ```
>
> The expression `res[i] = vec[i] + 5` is the key line in this small example. Thanks to the compiler Explorer we can have a closer look at the assembler instructions generated by x86-64 clang 3.6.
>
> Without Optimisation

Here are the assembler instructions. Each addition is done sequentially.

```
movslq -8(%rbp), %rax
movl vec(,%rax,4), %ecx
addl $5, %ecx
movslq -8(%rbp), %rax
movl %ecx, res (,%rax,4)
```

With maximum Optimisation

By using the highest optimisation level, `-O3`, special registers such as `xmm0` that can hold 128 bits or 4 `int`s are used. This means that the addition takes place in parallel on four elements of the vector.

```
movdqa .LCPI0_0(%rip), %xmm0   # xmm0 = [5,5,5,5]
movdqa vec(%rip), %xmm1
paddd %xmm0, %xmm1
movdqa %xmm1, res(%rip)
padd vec+16(%rip), %xmm0
movdqa %xmm0, res+16(%rip)
xorl %eax, %eax
```

77 of the STL algorithms can be parametrised by an execution policy.