4-execution

June 2, 2024

1 New execution strategies in C++17/20

C++17 takes a further step in the direction of automatic code parallelization by adding a new argument, the **execution policy**, to most of the algorithms in the standard library.

The available policies are `std::execution::seq`, `std::execution::par` and `std::execution::par_unseq`. C++20 is adding `std::execution::unseq`.

WARNING: the exceptions are not compatible with the algorithms which takes an execution policy as first argument.

WARNING: the actual application of parallelization is highly dependent on the hardware and the implementation of the compiler and its standard library. This requires most of the time (if not always) a **backend technology**. For example, GCC relies on Intel TBB.

Let's browse a toy code which takes a collection of floating point numbers, raises them to a given power, then prints their mean value.

[1]:

```
%%file tmp.execution-common.h

#include <vector>
#include <cmath>
#include <algorithm>
#include <numeric>
#include <iostream>
#include <cassert>

using Real = double ;
using Reals = std::vector<Real> ;
```

```
Overwriting tmp.execution-common.h
```

[2]:

```
%%file tmp.execution-prepare-input.h

// random numbers in [1.-1./scal,1.]
void generate( Reals & rs, Real scale )
 {
  srand(1) ;
  for ( auto & r : rs )
   { r = 1.-rand()/scale/RAND_MAX ; }
 }
```

```
Overwriting tmp.execution-prepare-input.h
```

[3]:

```
%%file tmp.execution-process.h

// compute xs^degree and store it into ys
void pow
 ( Reals const & xs, int degree, Reals & ys )
 {
  std::transform(xs.begin(),xs.end(),ys.begin(),
   [degree]( Real x )
   {
    Real y {1.0} ;
    for ( int d=0 ; d<degree ; ++d )
     { y *= x ; }
    return y ;
   }) ;
 }
```

```
Overwriting tmp.execution-process.h
```

[4]:

```
%%file tmp.execution-post-process-output.h

// compute the mean
void postprocess( Reals const & rs )
 { std::cout<<"mean: "<<(std::accumulate(rs.begin(),rs.end(),Real{0.})/rs.
 ↪size())<<"\n" ; }
```

```
Overwriting tmp.execution-post-process-output.h
```

[5]:

```
%%file tmp.execution.cpp

#include "tmp.execution-common.h"
#include "tmp.execution-prepare-input.h"
#include "tmp.execution-process.h"
#include "tmp.execution-post-process-output.h"

// main program
int main ( int argc, char * argv[] )
 {
  assert(argc==3) ;
  std::size_t dim {std::stoul(argv[1])} ;
  int degree = atoi(argv[2]) ;

  // prepare input
  Reals input(dim) ;
  generate(input,degree) ;
  // compute ouput
  Reals output(dim) ;
  pow(input,degree,output) ;

  // post-process
  postprocess(output) ;
 }
```

```
Overwriting tmp.execution.cpp
```

[6]:

```
%%file tmp.execution.sh
echo

rm -f tmp.execution.exe \
&& g++ -std=c++17 -O2 tmp.execution.cpp -o tmp.execution.exe\
&& time ./tmp.execution.exe $*

echo
```

```
Overwriting tmp.execution.sh
```

[9]:

```
!bash -l tmp.execution.sh 1024 100000
```

```

mean: 0.62705

real    0m0.099s
user    0m0.099s
sys     0m0.000s
```

1.1 Parallel execution policy

When using an algorithm from the standard library, we can **suggest** a multi-threaded execution using a simple additional argument:

[10]:

```
%%file tmp.execution-process.h

#include <execution>

// compute xs^degree and store it into ys
void pow
 ( Reals const & xs, int degree, Reals & ys )
 {
  std::transform(std::execution::par,xs.begin(),xs.end(),ys.begin(),[degree](␣
 ↪Real x )
   {
    Real y {1.0} ;
    for ( int d=0 ; d<degree ; ++d )
     { y *= x ; }
    return y ;
   }) ;
 }
```

```
Overwriting tmp.execution-process.h
```

Yet, for the time being, with GCC, do not forget to add the library `-ltbb`, because Intel TBB is the backend implementation which enables GCC to apply `std::execution::par`.

The simplicity of the written code as a drawback : additional installations and compilation options, at least today.

[13]:

```
%%file tmp.execution.sh
echo

rm -f tmp.execution.exe \
&& g++ -std=c++17 -O2 -ltbb tmp.execution.cpp -o tmp.execution.exe\
&& time ./tmp.execution.exe $*

echo
```

```
Overwriting tmp.execution.sh
```

[15]:

```
!bash -l tmp.execution.sh 1024 100000
```

```

mean: 0.62705

real	0m0.008s
user	0m0.105s
sys	0m0.000s
```

1.2 Unsequenced execution policy

It seems just as easy to suggest the use of vectorized SIMD instructions, with the `std::execution::unseq` execution policy (C++20 only)…

[16]:

```
%%file tmp.execution-process.h

#include <execution>

// compute xs^degree and store it into ys
void pow
 ( Reals const & xs, int degree, Reals & ys )
 {
  std::transform(std::execution::unseq,xs.begin(),xs.end(),ys.begin(),[degree](␣
 ↪Real x )
   {
```

```
   Real y = 1.0 ;
   for ( int d=0 ; d<degree ; ++d )
    { y *= x ; }
   return y ;
  }) ;
}
```

```
Overwriting tmp.execution-process.h
```

… but the speedup is not there. An important point is that the use of an execution policy argument **allows some parallelism, but does not make it mandatory**. It is an invitation given to the compiler, that it will accept or not, depending on the code context and the backends available.

[17]:
```
!bash -l tmp.execution.sh 1024 100000
```

```
mean: 0.62705

real	0m0.100s
user	0m0.096s
sys	0m0.004s
```

1.3 Availability of this features (for what concerns g++)

The effective implementation of the new parallel execution strategies requires a background implementation.

The last version of g++ (11.2), together with Intel TBB as backend, automatically apply multithreading when one ask a `std::execution::par` policy.

On the contrary, currently, g++ does not care about `std::execution::unseq`, and rather relies on its auto-vectorization feature.

2 Questions ?

2.1 Resources

- Rainer Grimm’s blog
- Rainer Grimm’s blog
- Bartek’s coding blog
- Compiler support (see “Standardization of Parallelism TS”)
- Intel parallel STL
- Intel TBB