3-async

June 2, 2024

1 Retrieve the results from a thread: `future` & `promise`

C ++ 11 offers a few ways to retrieve the output of a thread, without having to go through shared data and locks (apparently). This simpler style of programming is referred to as "asynchronous programming".

1.1 Using the function `async`

If you do not want to interfere in some thread during its progress, but simply launch its execution, do something else in parallel and then wait and use thread's result: use a call to `std::async` instead.

The call to **`std::async`** is **non-blocking**, and immediatly returns a variable of type **`std::future <T>`**. Only later on, when one consult the value of this variable thanks to a call to **`get()`**, then the program will **block** until the result is available.

[1]:
```
%%file tmp.async.h

#include <cstdio>
#include <chrono>
#include <thread>
#include <future>
#include <cassert>

using namespace std::chrono_literals ;
```

```
Writing tmp.async.h
```

[2]:
```
%%file tmp.async-add.h

int addition( int nb )
 {
  int res = 0 ;
  for ( int i=1 ; i<=nb ; ++i )
   {
    std::this_thread::sleep_for(10us) ;
    int sum = i+i ;
    printf("...addition : %d => %d\n",i,sum) ;
    res += i ;
```

```
   }
  return res ;
 }
```

```
Writing tmp.async-add.h
```

[3]:
```
%%file tmp.async-mul.h

int multiplication( int nb )
 {
  int res = 1 ;
  for ( int i=1 ; i<=nb ; ++i )
   {
    std::this_thread::sleep_for(50us) ;
    int square = i*i ;
    printf("...multiplication : %d => %d\n",i,square) ;
    res *= i ;
   }
  return res ;
 }
```

```
Writing tmp.async-mul.h
```

[4]:
```
%%file tmp.async.cpp

#include "tmp.async.h"
#include "tmp.async-add.h"
#include "tmp.async-mul.h"

int main( int argc, char * argv[] )
 {
  assert(argc==2) ;
  int nb = atoi(argv[1]) ;
  std::future<int> res1 = std::async(addition,nb) ;
  std::future<int> res2 = std::async(multiplication,nb) ;
  //...
  printf("=> final addition: %d\n",res1.get()) ;
  printf("=> final multiplication: %d\n",res2.get()) ;
  return 0 ;
 }
```

```
Writing tmp.async.cpp
```

[5]:
```
%%file tmp.async.sh
echo

rm -f tmp.async.exe \
&& g++ -std=c++17 -pthread tmp.async.cpp -o tmp.async.exe\
&& ./tmp.async.exe $*
```

```
echo
```

```
Writing tmp.async.sh
```

[6]:
```
! bash -l tmp.async.sh 5
```

```
…addition : 1 => 2
…multiplication : 1 => 1
…addition : 2 => 4
…addition : 3 => 6
…multiplication : 2 => 4
…addition : 4 => 8
…multiplication : 3 => 9
…addition : 5 => 10
…multiplication : 4 => 16
=> final addition: 15
…multiplication : 5 => 25
=> final multiplication: 120
```

1.2 Making promises

More generally, if we want to have a little more control over the course of the underlying thread, we can organize the recovery of a result by connecting one or more objects of type `future` in the main client code, together with one or several objects of type `promise` in to the supplier thread.

The same function can make several promises. In addition, it may still have things to do after the results are made available: the blocking call to `get()` on all expected values does not mean that the threads have finished what they have to do. Unlike using `std::async`, you have to add explicit calls to `join()` on all threads again, or they will be finished cleanly.

[7]:
```
%%file tmp.async-add.h

void addition( int nb, std::promise<int> prom )
 {
  int res = 0 ;
  for ( int i=1 ; i<=nb ; ++i )
   {
    res += i ;
    std::this_thread::sleep_for(10us) ;
    printf("...addition : %d => %d\n",i,res) ;
   }
  prom.set_value(res) ;
  std::this_thread::sleep_for(100us) ;
  printf("...addition cleaning\n") ;
 }
```

```
Overwriting tmp.async-add.h
```

[8]:
```
%%file tmp.async-mul.h

void multiplication( int nb, std::promise<int> prom )
 {
  int res = 1 ;
  for ( int i=1 ; i<=nb ; ++i )
   {
    res *= i ;
    std::this_thread::sleep_for(50us) ;
    printf("...multiplication : %d => %d\n",i,res) ;
   }
  prom.set_value(res) ;
  std::this_thread::sleep_for(100us) ;
  printf("...multiplication cleaning\n") ;
 }
```

```
Overwriting tmp.async-mul.h
```

[13]:
```
%%file tmp.async.cpp

#include "tmp.async.h"
#include "tmp.async-add.h"
#include "tmp.async-mul.h"

int main( int argc, char * argv[] )
 {
  assert(argc==2) ;
  int nb = atoi(argv[1]) ;

  std::promise<int> res1_promise ;
  std::promise<int> res2_promise ;

  std::future<int> res1_future = res1_promise.get_future() ;
  std::future<int> res2_future = res2_promise.get_future() ;

  std::thread t1(addition,nb,std::move(res1_promise)) ;
  std::thread t2(multiplication,nb,std::move(res2_promise)) ;

  //...

  printf("=> global addition: %d\n",res1_future.get()) ;
  printf("=> global multiplication: %d\n",res2_future.get()) ;

  t1.join() ;
  t2.join() ;
 }
```

```
Overwriting tmp.async.cpp
```

[14]:
```
! bash -l tmp.async.sh 5
```

```
…addition : 1 => 1
…multiplication : 1 => 1
…addition : 2 => 3
…multiplication : 2 => 2
…addition : 3 => 6
…addition : 4 => 10
…multiplication : 3 => 6
…addition : 5 => 15
=> global addition: 15
…multiplication : 4 => 24
…addition cleaning
…multiplication : 5 => 120
=> global multiplication: 120
…multiplication cleaning
```

1.3 Our shared future

Like an instance of `std::thread`, an instance of `std::future` is non-copiable, only movable. If we want to have several threads waiting for the same input asynchronous computation, we will instead use an instance of `std::shared_future`, which is copiable. Each client thread will then have its own copy of the result.

Below, we make the two threads wait together for the availability of the value of `nb` before actually starting their calculations. No longer taking advantage of the `multiplication` launch delay, `addition` takes a longer time to start.

[15]:
```
%%file tmp.async-add.h

int addition( std::shared_future<int> nb_future )
 {
  int res = 0 ;
  int nb = nb_future.get() ;
  for ( int i=1 ; i<=nb ; ++i )
   {
    std::this_thread::sleep_for(10us) ;
    int sum = i+i ;
    printf("...addition : %d => %d\n",i,sum) ;
    res += i ;
   }
  return res ;
 }
```

```
Overwriting tmp.async-add.h
```

[16]:
```
%%file tmp.async-mul.h

int multiplication( std::shared_future<int> nb_future )
 {
  int res = 1 ;
  int nb = nb_future.get() ;
  for ( int i=1 ; i<=nb ; ++i )
   {
    std::this_thread::sleep_for(50us) ;
    int square = i*i ;
    printf("...multiplication : %d => %d\n",i,square) ;
    res *= i ;
   }
  return res ;
 }
```

```
Overwriting tmp.async-mul.h
```

[17]:
```
%%file tmp.async.cpp

#include "tmp.async.h"
#include "tmp.async-add.h"
#include "tmp.async-mul.h"

int main( int argc, char * argv[] )
 {
  assert(argc==2) ;
  int nb = atoi(argv[1]) ;

  std::promise<int> nb_promise ;
  std::shared_future<int> nb_future(nb_promise.get_future()) ;

  std::future<int> res1 = std::async(addition,nb_future) ;
  std::future<int> res2 = std::async(multiplication,nb_future) ;

  nb_promise.set_value(nb) ;

  //...

  printf("=> global addition: %d\n",res1.get()) ;
  printf("=> global multiplication: %d\n",res2.get()) ;

  return 0 ;
 }
```

```
Overwriting tmp.async.cpp
```

[18]:
```
! bash -l tmp.async.sh 5
```

```
…addition : 1 => 2
…multiplication : 1 => 1
…addition : 2 => 4
…addition : 3 => 6
…multiplication : 2 => 4
…addition : 4 => 8
…multiplication : 3 => 9
…addition : 5 => 10
=> global addition: 15
…multiplication : 4 => 16
…multiplication : 5 => 25
=> global multiplication: 120
```

2 Questions ?

3 Exercise

Modify the program below to use the `std::async()` function instead of the explicit threads.

We have already modified the function `complexes_pow`, so that it returns its slice of results, instead of storing it into an arry received by reference. But now **it does not compile** (on purpose) : you still have to upgrade the section *compute* in the main program.

Check your results with the usual commands and parameters.

[19]:
```
%%file tmp.async.cpp

#include <complex>
#include <vector>
#include <iostream>
#include <cassert>
#include <cmath>
#include <thread>

using Real = double ;
using Complex = std::complex<Real> ;
using Complexes = std::vector<Complex> ;

// random unitary complexes
void generate( Complexes & cs )
 {
  srand(1) ;
  for ( auto & c : cs )
   {
    Real angle {rand()/(Real(RAND_MAX)+1)*2.0*M_PI} ;
    c = Complex{std::cos(angle),std::sin(angle)} ;
```

```
   }
 }

// compute a slice of xs^degree and return it
Complexes complexes_pow
 ( std::size_t num_slice, std::size_t nb_slices,
   Complexes const & xs, int degree )
 {
  assert((xs.size()%nb_slices)==0) ;
  auto slice_size {xs.size()/nb_slices} ;
  auto min {num_slice*slice_size} ;
  Complexes ys(slice_size) ;
  for ( decltype(slice_size) i {0} ; i<slice_size ; ++i )
   {
    ys[i] = Complex{1.,0.} ;
    for ( int d=0 ; d<degree ; ++d )
     { ys[i] *= xs[i+min] ; }
   }
  return ys ;
 }

// display the angle of the global product
void postprocess( Complexes const & cs )
 {
  Complex prod {1.,0.} ;
  for( auto c : cs ) { prod *= c ; }
  double angle {atan2(prod.imag(),prod.real())} ;
  std::cout<<"result = "<<static_cast<int>(angle/2./M_PI*360.)<<"\n" ;
 }

// main program
int main ( int argc, char * argv[] )
 {
  assert(argc==4) ;
  std::size_t nbtasks {std::stoul(argv[1])} ;
  std::size_t dim {std::stoul(argv[2])} ;
  int degree {std::stoi(argv[3])} ;

  // prepare input
  Complexes input(dim) ;
  generate(input) ;

  // compute
  Complexes output(dim) ;
  std::size_t numtask ;
  std::vector<std::thread> workers ;
  for ( numtask = 0 ; numtask<nbtasks ; ++numtask )
```

```
  { workers.emplace_back(complexes_pow,numtask,nbtasks,std::
↪ref(input),degree,std::ref(output)) ; }
 for ( auto & worker : workers )
  { worker.join() ; }

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

Overwriting tmp.async.cpp

[20]:
```
%%file tmp.async.sh
echo

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

echo
```

Overwriting tmp.async.sh

[21]:
```
! bash -l tmp.async.sh 2 2 3
```

```
In file included from /usr/local/include/c++/13.2.0/thread:45,
                 from tmp.async.cpp:7:
/usr/local/include/c++/13.2.0/bits/std_thread.h: In instantiation of
‘std::thread::thread(_Callable&&, _Args&& …)
[with _Callable = std::vector<std::complex<double> > (&)(long unsigned
int, long unsigned int, const std::vector<std::complex<double> >&, int); _Args =
{long unsigned int&, long unsigned int&,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > >, int&,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > >}; <template-parameter-1-3> =
void]’:
/usr/local/include/c++/13.2.0/bits/new_allocator.h:187:4:
required from ‘void
std::__new_allocator<_Tp>::construct(_Up*, _Args&& …)
```

```
[with _Up = std::thread; _Args = {std::vector<std::complex<double>,
std::allocator<std::complex<double> > > (&)(long unsigned int, long unsigned
int, const std::vector<std::complex<double>, std::allocator<std::complex<double>
> >&, int), long unsigned int&, long unsigned int&,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > >, int&,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > >}; _Tp = std::thread]'
/usr/local/include/c++/13.2.0/bits/alloc_traits.h:537:17:
required from 'static void std::allocator_traits<std::allocator<_CharT>
>::construct(allocator_type&, _Up*, _Args&& ...) [with
_Up = std::thread; _Args = {std::vector<std::complex<double>,
std::allocator<std::complex<double> > > (&)(long unsigned int, long unsigned
int, const std::vector<std::complex<double>, std::allocator<std::complex<double>
> >&, int), long unsigned int&, long unsigned int&,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > >, int&,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > >}; _Tp = std::thread; allocator_type =
std::allocator<std::thread>]'
/usr/local/include/c++/13.2.0/bits/vector.tcc:117:30:   required
from 'std::vector<_Tp, _Alloc>::reference std::vector<_Tp,
_Alloc>::emplace_back(_Args&& ...) [with _Args =
{std::vector<std::complex<double>, std::allocator<std::complex<double> > >
(&)(long unsigned int, long unsigned int, const
std::vector<std::complex<double>, std::allocator<std::complex<double> > >&,
int), long unsigned int&, long unsigned int&,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > >, int&,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > >}; _Tp = std::thread; _Alloc =
std::allocator<std::thread>; reference = std::thread&]'
tmp.async.cpp:68:26:   required from here
/usr/local/include/c++/13.2.0/bits/std_thread.h:157:72:
```

```
error: static assertion failed: std::thread arguments must be
invocable after conversion to rvalues
  157 |                                     typename
decay<_Args>::type…>::value,
      |
^~~~~

/usr/local/include/c++/13.2.0/bits/std_thread.h:157:72:
note: ‘std::integral_constant<bool,

false>::value’ evaluates to false
/usr/local/include/c++/13.2.0/bits/std_thread.h: In instantiation of
‘struct

std::thread::_Invoker<std::tuple<std::vector<std::complex<double>,

std::allocator<std::complex<double> > > (*)(long unsigned int, long unsigned

int, const std::vector<std::complex<double>, std::allocator<std::complex<double>

> >&, int), long unsigned int, long unsigned int,

std::reference_wrapper<std::vector<std::complex<double>,

std::allocator<std::complex<double> > > >, int,

std::reference_wrapper<std::vector<std::complex<double>,

std::allocator<std::complex<double> > > > > >’:
/usr/local/include/c++/13.2.0/bits/std_thread.h:236:13:   required
from ‘struct std::thread::_State_impl<std::thread::_Invoker<std::tuple<s

td::vector<std::complex<double>, std::allocator<std::complex<double> > >

(*)(long unsigned int, long unsigned int, const

std::vector<std::complex<double>, std::allocator<std::complex<double> > >&,

int), long unsigned int, long unsigned int,

std::reference_wrapper<std::vector<std::complex<double>,

std::allocator<std::complex<double> > > >, int,

std::reference_wrapper<std::vector<std::complex<double>,

std::allocator<std::complex<double> > > > > >’
/usr/local/include/c++/13.2.0/bits/std_thread.h:164:29:   required
from ‘std::thread::thread(_Callable&&, _Args&& …)
```

```
[with _Callable = std::vector<std::complex<double> > (&)(long unsigned
int, long unsigned int, const std::vector<std::complex<double> >&, int); _Args =
{long unsigned int&, long unsigned int&,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > >, int&,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > >}; <template-parameter-1-3> =
void]'
/usr/local/include/c++/13.2.0/bits/new_allocator.h:187:4:
required from 'void
std::__new_allocator<_Tp>::construct(_Up*, _Args&& ...)
[with _Up = std::thread; _Args = {std::vector<std::complex<double>,
std::allocator<std::complex<double> > > (&)(long unsigned int, long unsigned
int, const std::vector<std::complex<double>, std::allocator<std::complex<double>
> >&, int), long unsigned int&, long unsigned int&,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > >, int&,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > >}; _Tp = std::thread]'
/usr/local/include/c++/13.2.0/bits/alloc_traits.h:537:17:
required from 'static void std::allocator_traits<std::allocator<_CharT>
>::construct(allocator_type&, _Up*, _Args&& ...) [with
_Up = std::thread; _Args = {std::vector<std::complex<double>,
std::allocator<std::complex<double> > > (&)(long unsigned int, long unsigned
int, const std::vector<std::complex<double>, std::allocator<std::complex<double>
> >&, int), long unsigned int&, long unsigned int&,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > >, int&,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > >}; _Tp = std::thread; allocator_type =
std::allocator<std::thread>]'
/usr/local/include/c++/13.2.0/bits/vector.tcc:117:30:   required
```

```
from ‘std::vector<_Tp, _Alloc>::reference std::vector<_Tp,
_Alloc>::emplace_back(_Args&& …) [with _Args =
{std::vector<std::complex<double>, std::allocator<std::complex<double> > >
(&)(long unsigned int, long unsigned int, const
std::vector<std::complex<double>, std::allocator<std::complex<double> > >&,
int), long unsigned int&, long unsigned int&,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > >, int&,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > >}; _Tp = std::thread; _Alloc =
std::allocator<std::thread>; reference = std::thread&]’
tmp.async.cpp:68:26:   required from here
/usr/local/include/c++/13.2.0/bits/std_thread.h:291:11:
error: no type named ‘type’ in ‘struct
std::thread::_Invoker<std::tuple<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > (*)(long unsigned int, long unsigned
int, const std::vector<std::complex<double>, std::allocator<std::complex<double>
> >&, int), long unsigned int, long unsigned int,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > >, int,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > > >
>::__result<std::tuple<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > (*)(long unsigned int, long unsigned
int, const std::vector<std::complex<double>, std::allocator<std::complex<double>
> >&, int), long unsigned int, long unsigned int,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > >, int,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > > >’
  291 |           _M_invoke(_Index_tuple<_Ind…>)
      |           ^~~~~~~~~
/usr/local/include/c++/13.2.0/bits/std_thread.h:295:9:
```

```
error: no type named ‘type’ in ‘struct
std::thread::_Invoker<std::tuple<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > (*)(long unsigned int, long unsigned
int, const std::vector<std::complex<double>, std::allocator<std::complex<double>
> >&, int), long unsigned int, long unsigned int,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > >, int,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > > >
>::__result<std::tuple<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > (*)(long unsigned int, long unsigned
int, const std::vector<std::complex<double>, std::allocator<std::complex<double>
> >&, int), long unsigned int, long unsigned int,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > >, int,
std::reference_wrapper<std::vector<std::complex<double>,
std::allocator<std::complex<double> > > > >’
  295 |         operator()()
      |         ^~~~~~~~
```

[11]:
```
! bash -l tmp.async.sh 4 1024 100000
```

```
result = -77

real    0m0.543s
user    0m2.110s
sys     0m0.008s
```