1-thread

June 2, 2024

1 Threads

The C ++ language itself offers no support for parallelism, but since C ++ 11, the standard library offers support for multi-threaded programming. It is essentially a common interface to long-available implementations on Linux (pthread) and Windows.

BEWARE: under Linux, despite the common interface offered by C ++ 11, it is still essential to put the specific option `-lpthread` on the link command, because the binary will need the **pthread backend** at runtime.

1.1 Launching and waiting for a thread

We can outsource the execution of a function to another thread simply by creating an instance of **`std::thread`**, to which we give as argument the name of the function to execute. The creation of this instance is **non-blocking**: while the thread takes over the execution of the outsourced function, the main program continues its execution without waiting.

When we need to make sure that a thread has finished its execution, we use the **`join()`** member function, which will be **blocking** as long as the process is not finished.

BEWARE: for each thread launched, you will need at least one `join()` call in your main program. Without this, if your main program terminates while there are still running threads, the operating system will consider your program to be globally terminated and it will kill any running threads.

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

#include <cstdio>
#include <chrono>
#include <thread>

using namespace std::chrono_literals ;
```

```
Writing tmp.thread.h
```

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

void addition()
 {
  int res = 0 ;
  for ( int i=1 ; i<=5 ; ++i )
```

```
   {
    res += i ;
    std::this_thread::sleep_for(10us) ;
    printf("...addition : %d => %d\n",i,res) ;
   }
  printf("=> final addition: %d\n",res) ;
 }
```

```
Writing tmp.thread-add.h
```

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

void multiplication()
 {
  int res = 1 ;
  for ( int i=1 ; i<=5 ; ++i )
   {
    res *= i ;
    std::this_thread::sleep_for(50us) ;
    printf("...multiplication : %d => %d\n",i,res) ;
   }
  printf("=> final multiplication: %d\n",res) ;
 }
```

```
Writing tmp.thread-mul.h
```

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

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

int main()
 {
  std::thread t1(addition) ;
  std::thread t2(multiplication) ;
  //...
  t1.join() ;
  t2.join() ;
  return 0 ;
 }
```

```
Writing tmp.thread.cpp
```

[5]:
```
!rm -f tmp.thread.exe && g++ -std=c++17 -lpthread tmp.thread.cpp -o tmp.thread.
 ↪exe
```

[6]:
```
!./tmp.thread.exe
```

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

1.2 Passing arguments

When building the `std::thread` instance, we can add additional arguments that will be passed to the called function:

[7]:
```
%%file tmp.thread.h

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

using namespace std::chrono_literals ;
```

```
Overwriting tmp.thread.h
```

[8]:
```
%%file tmp.thread-add.h

void addition( int nb )
 {
  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) ;
   }
  printf("=> final addition: %d\n",res) ;
 }
```

```
Overwriting tmp.thread-add.h
```

[9]:
```
%%file tmp.thread-mul.h

void multiplication( int nb )
```

```
{
 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) ;
  }
 printf("=> final multiplication: %d\n",res) ;
}
```

```
Overwriting tmp.thread-mul.h
```

[10]:
```
%%file tmp.thread.cpp

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

int main( int argc, char * * argv )
 {
  assert(argc==2) ;
  int nb = atoi(argv[1]) ;
  std::thread t1(addition,nb) ;
  std::thread t2(multiplication,nb) ;
  //...
  t1.join() ;
  t2.join() ;
 }
```

```
Overwriting tmp.thread.cpp
```

[11]:
```
!rm -f tmp.thread.exe && g++ -std=c++17 -lpthread tmp.thread.cpp -o tmp.thread.
 ↪exe
```

[12]:
```
!./tmp.thread.exe 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
=> final addition: 15
…multiplication : 4 => 24
…multiplication : 5 => 120
=> final multiplication: 120
```

As you may notice, all the functions proposed above do not return anything. Since their execution is delegated to another thread, there is **no immediately available result that can be stored in a variable**.

An alternative is to pass the function argument **by reference**, in which the function can store its result, but…

BEWARE: when creating an instance of `std::thread`, because of the time it takes to prepare the process, the function arguments are **duplicated by value**, before being subsequently passed to the function. If you want to pass arguments by reference to the function in order to preserve their reference qualification, you must wrap the value within `std::ref()`. In addition, you cannot be sure the variable has been filled, until you make a call to `join()`.

[13]:
```
%%file tmp.thread-add.h

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

```
Overwriting tmp.thread-add.h
```

[37]:
```
%%file tmp.thread-mul.h

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

```
Overwriting tmp.thread-mul.h
```

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

#include "tmp.thread.h"
#include "tmp.thread-add.h"
#include "tmp.thread-mul.h"
```

```
int main( int argc, char * argv[] )
 {
  assert(argc==2) ;
  int nb = atoi(argv[1]) ;
  int res1, res2 ;
  std::thread t1(addition,nb,std::ref(res1)) ;
  std::thread t2(multiplication,nb,std::ref(res2)) ;
  //...
  t1.join() ;
  printf("=> final addition: %d\n",res1) ;
  t2.join() ;
  printf("=> final multiplication: %d\n",res2) ;
 }
```

```
Overwriting tmp.thread.cpp
```

[20]:
```
!rm -f tmp.thread.exe && g++ -std=c++17 -lpthread tmp.thread.cpp -o tmp.thread.
 ↪exe
```

[22]:
```
!./tmp.thread.exe 5
```

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

1.3 Use of other “callable” entities

An instance of `std::thread` can handle not only functions, but any kind of callable object, such as function objects:

[42]:
```
%%file tmp.thread-add.h

class Addition
 {
  public :
    void operator()( int nb, int & res )
     {
      res = 0 ;
      for ( int i=1 ; i<=nb ; ++i )
```

```
      {
       res += i ;
       std::this_thread::sleep_for(10us) ;
       printf("...addition : %d => %d\n",i,res) ;
      }
    }
} ;
```

```
Overwriting tmp.thread-add.h
```

[43]:
```
%%file tmp.thread-mul.h

class Multiplication
 {
  public :
    Multiplication( int nb ) : m_nb(nb) {}
    void operator()( int & res)
     {
      res = 1 ;
      for ( int i=1 ; i<m_nb ; ++i )
       {
        res *= i ;
        std::this_thread::sleep_for(50us) ;
        printf("...multiplication : %d => %d\n",i,res) ;
       }
     }
  private :
    int const m_nb ;
 } ;
```

```
Overwriting tmp.thread-mul.h
```

[44]:
```
%%file tmp.thread.cpp

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

int main( int argc, char * argv[] )
 {
  assert(argc==2) ;
  int nb = atoi(argv[1]) ;
  int res1, res2 ;
  std::thread t1(Addition(),nb,std::ref(res1)) ;
  std::thread t2(Multiplication(nb),std::ref(res2)) ;
  //...
  t1.join() ;
  printf("=> final addition: %d\n",res1) ;
```

```
  t2.join() ;
  printf("=> final multiplication: %d\n",res2) ;
  return 0 ;
 }
```

```
Overwriting tmp.thread.cpp
```

[45]:
```
!rm -f tmp.thread.exe && g++ -std=c++17 -lpthread tmp.thread.cpp -o tmp.thread.
↪exe
```

[46]:
```
!./tmp.thread.exe 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
=> final addition: 15
…multiplication : 4 => 24
=> final multiplication: 24
```

Or even lambdas:

[55]:
```
%%file tmp.thread-add.h

auto make_add()
 {
  return []( int nb, int & res )
   {
    res = 0 ;
    for ( int i=1 ; i<=nb ; ++i )
     {
      res += i ;
      std::this_thread::sleep_for(10us) ;
      printf("...addition : %d => %d\n",i,res) ;
     }
   } ;
 }
```

```
Overwriting tmp.thread-add.h
```

[56]:
```
%%file tmp.thread-mul.h

auto make_mul( int nb )
 {
  return [nb]( int & res )
```

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

Overwriting tmp.thread-mul.h

[60]:
```
%%file tmp.thread.cpp

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

int main( int argc, char * argv[] )
 {
  assert(argc==2) ;
  int nb = atoi(argv[1]) ;
  int res1, res2 ;
  std::thread t1(make_add(),nb,std::ref(res1)) ;
  std::thread t2(make_mul(nb),std::ref(res2)) ;
  //...
  t1.join() ;
  printf("=> final addition: %d\n",res1) ;
  t2.join() ;
  printf("=> final multiplication: %d\n",res2) ;
  return 0 ;
 }
```

Overwriting tmp.thread.cpp

[61]:
```
!rm -f tmp.thread.exe && g++ -std=c++17 -lpthread tmp.thread.cpp -o tmp.thread.
 ↪exe
```

[62]:
```
!./tmp.thread.exe 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
```

```
=> final addition: 15
…multiplication : 4 => 24
…multiplication : 5 => 120
=> final multiplication: 120
```

2 Questions ?

3 Exercise

Step 1, get familiar with the code Test the cells below.

The program takes as first argument the number of tasks to run in parallel. The function `complexes_pow` is designed to work on a single slice within a global array, but for now the main program executes these slices one after the other.

The first execution of the script `tmp.thread.sh 2 2 3` manipulates arrays of 2 complexes and raises them to the power of 3. It mainly allows you to check that your calculations are correct and always return the same final result.

The second execution `tmp.thread.sh 4 1024 100000` manipulates arrays of 1024 complex numbers and raises them to the power of 100000. It is on this heavier calculation that you should observe an acceleration once your parallelization is successful.

Step 2, make it faster with threads Modify the program using instances of `std::thread`, to result in a faster parallel execution (this happens mostly on the side of the `// compute` section of the `main ()` function).

After each incremental result, check that you did not break the code with `tmp.thread.sh 2 2 3`.

When you think you have finished, check that `tmp.thread.sh 4 1024 100000` runs significantly faster. What happens when you push upper the number of threads ? What's the optimal value ?

[63]:
```
%%file tmp.thread.cpp

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

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 store it into ys
// xs.size() must be a multiple of nb_slices
void complexes_pow
 ( std::size_t num_slice, std::size_t nb_slices,
   Complexes const & xs, int degree, Complexes & ys )
 {
  assert((xs.size()%nb_slices)==0) ;
  auto slice_size {xs.size()/nb_slices} ;
  auto min {num_slice*slice_size} ;
  auto max {(num_slice+1)*slice_size} ;

  for ( auto i {min} ; i<max ; ++i )
   {
    ys[i] = Complex{1.,0.} ;
    for ( int d=0 ; d<degree ; ++d )
     { ys[i] *= xs[i] ; }
   }
 }

// 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" ;
 }

// programme principal
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 ouput
```

```
  Complexes output(dim) ;
  for ( std::size_t numtask {0} ; numtask<nbtasks ; ++numtask )
   { complexes_pow(numtask,nbtasks,input,degree,output) ; }

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

Overwriting tmp.thread.cpp

[70]:
```
%%file tmp.thread.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 store it into ys
// xs.size() must be a multiple of nb_slices
void complexes_pow
 ( std::size_t num_slice, std::size_t nb_slices,
   Complexes const & xs, int degree, Complexes & ys )
 {
  assert((xs.size()%nb_slices)==0) ;
  auto slice_size {xs.size()/nb_slices} ;
  auto min {num_slice*slice_size} ;
  auto max {(num_slice+1)*slice_size} ;

  for ( auto i {min} ; i<max ; ++i )
   {
```

```
    ys[i] = Complex{1.,0.} ;
    for ( int d=0 ; d<degree ; ++d )
     { ys[i] *= xs[i] ; }
   }
 }

// 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.thread.cpp
```

[71]:
```
%%file tmp.thread.sh
echo

rm -f tmp.thread.exe \
```

```
&& g++ -std=c++17 tmp.thread.cpp -o tmp.thread.exe\
&& time ./tmp.thread.exe $*

echo
```

```
Overwriting tmp.thread.sh
```

[72]:
```
! bash -l tmp.thread.sh 2 2 3
```

```
result = -106

real    0m0.001s
user    0m0.001s
sys     0m0.000s
```

[84]:
```
! bash -l tmp.thread.sh 4 1024 100000
```

```
result = -77

real    0m0.314s
user    0m1.235s
sys     0m0.000s
```