THE FRAMEWORK PROGRAMME FOR RESEARCH AND INNOVATION

HORIZON 2020

PRQA
Programming Research

How to avoid bottlenecks when converting serial code to multithreaded

Wojciech Basalaj
Richard Corden

ACCU 2018

REPHRASE

www.programmingresearch.com

Background

- The RePhrase Project is an EU Horizon 2020 research project
- New software engineering tools that help tackle multi-core computing
- PRQA's contributions
  - Rule set facilitating parallel programming
  - Qualitative analysis of sources before and after transformations

Outline

- Review a code sample
- Identify code constructs that interfere with parallelism
- Check relevant constraints in the C++ standard
- Formulate coding rule(s)
- Rinse and repeat
- Come away with coding guidelines for parallel ready code

Simple Accumulator

```
#include <vector>

int reentrantFunc (int);

int simpleAcc (std::vector <int> const & data)
{
  int result = 0;

  for (int i = 0; i < data.size (); ++i)
  {
    result += reentrantFunc (data [i]);
  }

  return result;
}
```

Simple Accumulator – range for

```
#include <vector>

int reentrantFunc (int);

int simpleAcc2 (std::vector <int> const & data)
{
  int result = 0;

  for (auto d : data)
  {
    result += reentrantFunc (d);
  }

  return result;
}
```

Simple Accumulator – STL algorithm

```
#include <vector>
#include <numeric>

int reentrantFunc (int);

int simpleAcc3 (std::vector <int> const & data)
{
  auto ftor = [] (int sum, int d) {
    return sum + reentrantFunc (d);
  };

  return std::accumulate (std::execution::par
    , data.cbegin (), data.cend (), 0, ftor);
}
```

Simple Accumulator – C++17

```
#include <vector>
#include <numeric>
#include <execution>

int reentrantFunc (int);

int simpleAcc4 (std::vector <int> const & data)
{
  return std::transform_reduce (std::execution::par
                // or switch to std::execution::seq
    , data.cbegin ()
    , data.cend ()
    , 0               // initialisation of sum
    , std::plus <> () // reduce operation
    , reentrantFunc); // transform operation
}
```

Simple Accumulator – C++ constraints

- Data races [intro.races]

  "(2) Two expression evaluations conflict if one of them modifies a memory location and the other one reads or modifies the same memory location."

  "(3) The library defines a number of atomic operations and operations on mutexes that are specially identified as synchronization operations..."

Simple Accumulator – rules

- no access to objects visible outside of the loop/functor
  - data races when parallelised
    - need synchronisation
- use range for or an STL algorithm where possible
  - easier conversion between serial and parallel execution with C++17 parallel algorithms
    - limited compiler support
      - Intel System Studio 2018
      - Visual C++ 2017

Simple Accumulator – VC++ 2017

```
#include <vector>
#include <numeric>
#include <algorithm>
#include <execution>

int reentrantFunc (int);

int simpleAcc4 (std::vector <int> const & data)
{
  std::vector <int> temp (data.size ());
  std::transform (std::execution::par
    , data.cbegin ()
    , data.cend ()
    , temp.begin ()
    , reentrantFunc);

  return std::reduce (std::execution::par
    , temp.cbegin ()
    , temp.cend ());
}
```

Has Negative

```
#include <vector>

int reentrantFunc (int);

bool hasNegative (std::vector <int> const & data)
{
  for (auto d : data)
  {
    auto v = reentrantFunc (d);
    if (v < 0)
    {
      return true;
    }
  }
  return false;
}
```

Has Negative – fixed number of iterations

```
#include <vector>

int reentrantFunc (int);

bool hasNegative2 (std::vector <int> const & data)
{
  bool result = false;

  for (auto d : data)
  {
// is short-circut really wanted: variable number of calls
    result = result || (reentrantFunc (d) < 0);
//  result = (reentrantFunc (d) < 0) || result;
  }
  return result;
}
```

Has Negative – C++17 – everything OK?

```
#include <vector>
#include <algorithm>
#include <execution>

int reentrantFunc (int);

bool hasNegative4 (std::vector <int> const & data)
{
  auto ftor = [] (int d) { return reentrantFunc (d) < 0; };

  return std::any_of (std::execution::par
    , data.cbegin ()
    , data.cend ()
    , ftor);
}
```

Has Negative – all singing and dancing

```
bool hasNegative5 (std::vector <int> const & data)
{
  auto ftor = [] (int d) noexcept {
    bool result = true; // or perhaps false?
    try
    {
      result = reentrantFunc (d) < 0;
    }
    catch (...) {}

    return result;
  };

  return std::any_of (std::execution::par
    , data.cbegin ()
    , data.cend ()
    , ftor);
}
```

Has Negative – C++ constraints

- Uncaught exceptions in functors
  - The std::terminate() function [except.terminate]

“(1.11) — for a parallel algorithm whose ExecutionPolicy specifies such behavior, when execution of an element access function of the parallel algorithm exits via an exception”

Has Negative – rules

- no break, goto, return, throw, or other non-returning statement/function call inside a loop
  - hard/impossible to parallelise
- no uncaught exception or a non-returning statement/function call inside a functor

Nested algorithms – what have we missed?

```
// #includes
int reentrantFunc (int) noexcept;

void nested (std::vector <std::vector <int> > & data)
{
  std::for_each (std::execution::par
    , data.begin ()
    , data.end ()
    , [] (auto & inner) noexcept {
    std::transform (std::execution::par
      , inner.cbegin ()
      , inner.cend ()
      , inner.begin ()
      , reentrantFunc); // transform operation
  });
}
```

Nested algorithms – fixed

```
void nested2 (std::vector <std::vector <int> > & data)
{
  bool hadException = std::find_if (std::execution::par
    , data.begin ()
    , data.end ()
    , [] (auto & inner) noexcept {
    bool result = false;
    try
    {
      std::transform (std::execution::par
        , inner.cbegin ()
        , inner.cend ()
        , inner.begin ()
        , reentrantFunc);
    }
    catch (std::bad_alloc const & e)
    {
      result = true;
    }
    return result;
  }) != data.end ();
}
```

Nested algorithms – C++ constraints

- Parallel algorithm exceptions [algorithms.parallel.exceptions]
  - “(1) During the execution of a parallel algorithm, if temporary memory resources are required for parallelization and none are available, the algorithm throws a bad_alloc exception.”

Nested algorithms – rules

- Handle a std::bad_alloc exception that can be thrown by a parallel algorithm

Random Accumulator

```
#include <vector>
#include <cstdlib>

int reentrantFunc (int) noexcept;

int randomAccumulator (std::vector <int> const & data)
{
  int result = 0;

  for (auto d : data)
  {
    auto random = std::rand ();
    auto v = reentrantFunc (d);
    if (v > random)
    {
      result += v;
    }
  }

  return result;
}
```

Random Accumulator – fixed

```
int randomAccumulator2 (std::vector <int> const & data)
{
  std::vector <int> test (data.size ());

  // The rand function is not required to avoid data races with other calls to
  // pseudo-random sequence generation functions.
  // serial execution only
  std::generate (test.begin (), test.end (), std::rand);

  int result = 0;
  auto it = test.cbegin ();
  for (auto d : data) // potential parallel execution
  {
    auto v = reentrantFunc (d);
    if (v > *it)
    {
      result += v;
    }
    ++it;
  }

  return result;
}
```

Random Accumulator – C++17

```cpp
int randomAccumulator3 (std::vector <int> const & data)
{
  std::vector <int> test (data.size ());

  // serial execution only
  std::generate (test.begin (), test.end (), std::rand);

  auto ftor = [] (int d, int randNum) {
    auto v = reentrantFunc (d);
    return (v > randNum) ? v : 0;
  };

  return std::transform_reduce (std::execution::par // or switch to
   std::execution::seq
    , data.cbegin ()
    , data.cend ()
    , test.cbegin ()
    , 0
    , std::plus <> () // reduce op
    , ftor);          // transform op
}
```

Random Accumulator – C++ constraints

- The C++ standard library has data race guarantees [res.on.data.races]
  - the implementation will not implicitly introduce data races
  - arguments passed may cause data races
- non-reentrant functions in the C standard library
  - asctime, ctime, gmtime, and localtime [ctime.syn]
  - strerror and strtok [cstring.syn]
  - multibyte / wide string and character conversion functions [c.mb.wcs]
  - <locale> and <clocale> functions [locales] & [c.locales]
  - rand [c.math.rand]
  - tmpnam [cstdio.syn]

Random Accumulator – rules

- no call to a non-reentrant function inside a loop or functor
  - data races when parallelised

Recursive

```
int recursiveHelper (int i, std::vector <int> & data)
{
  data [i] = reentrantFunc (data [i]);
  if (i > 0)
  {
    data [i] += recursiveHelper (i - 1, data);
  }
  return data [i];
}

void recursive (std::vector <int> & data)
{
  if (! data.empty ())
  {
    recursiveHelper (data.size () - 1, data);
  }
}
```

Recursive – rules

- no direct or indirect recursion
  - needs to be converted to loop(s) before parallelisation

Iterative – better?

```
void iterative (std::vector <int> & data)
{
  int num = data.size ();

  data [0] = reentrantFunc (data [0]);

  for (int i = 1; i != num; ++i)
  {
    data [i] = reentrantFunc (data [i]);
    data [i] += data [i - 1];
  }
}
```

Iterative – C++17 – everything OK?

```
void iterative3 (std::vector <int> & data)
{
  std::transform (std::execution::par
    , data.cbegin ()
    , data.cend ()
    , data.begin ()
    , reentrantFunc);

  // serial execution only
  int num = data.size ();
  for (int i = 1; i != num; ++i)
  {
    data [i] += data [i - 1]; // loop carried dependence
  }
}
```

Iterative – fixed

```
void iterative4 (std::vector <int> & data)
{
  std::transform_inclusive_scan (
    std::execution::par
    , data.cbegin ()
    , data.cend ()
    , data.begin ()
    , std::plus <> () // prefix sum
    , reentrantFunc); // transform op
}
```

Iterative – C++ constraints

- Container data races [container.requirements.dataraces]

  “2 ... implementations are required to avoid data races when the contents of the contained object in different elements in the same container, excepting vector<bool>, are modified concurrently.”

Iterative – rules

- avoid/isolate loop carried dependences
  - When a statement in one iteration of a loop depends in some way on a statement in a different iteration of the same loop

Mean – is it correct?

```
#include <numeric>
#include <execution>

int mean (std::vector <int> const & data)
{
  auto ftor = [] (int i, int j) {
    return (i + j) / 2;
  };

  return std::reduce (std::execution::par
    , data.cbegin ()
    , data.cend ()
    , 0
    , ftor);
}
```

Mean – fixed

```
int mean2 (std::vector <int> const & data)
{
  return std::reduce (std::execution::par
    , data.cbegin ()
    , data.cend ()
    , 0
    , std::plus <> ()) / data.size ();
}
```

Mean – C++ constraints

- Non-determinism
  - Reduce [reduce]
  - Transform reduce [transform.reduce]

"(8) [Note: The difference between reduce and accumulate is that reduce applies binary_op in an unspecified order, which yields a nondeterministic result for non-associative or non-commutative binary_op such as floating-point addition. —end note]"

Mean – rules

- Only use an associative and commutative binary op with std::reduce or std::transform reduce

Parallelisation Blockers: control flow

- Complex loops
  - Cannot be easily represented as STL algorithms or range-for
  - break/goto/return/throw/other non-returning statement
- Complex functors
  - Uncaught exceptions or a non-returning statement/function call
    - functor should always return
- Recursion
  - Convert into a loop first

Parallelisation Blockers: control flow - continued

- Prefer an STL algorithm with a functor/lambda that always returns
  - RePhrase rules 2.7, 4.1, 4.5, and 4.6
  - Easy conversion to a parallel algorithm
- std::bad_alloc exception can be thrown by a parallel algorithm
  - RePhrase rule 4.7

Parallelisation Blockers: data races

- Access to objects visible outside of the loop/functor
- Calling non-reentrant library functions
  - strtok/rand etc.
- Loop carried dependence
  - Evaluation of an iteration depends on the result of another
- Require synchronisation to prevent races
  - reduce performance gains from parallelisation
- RePhrase rules 4.2 and 4.9

Parallelisation Blockers: non-determinism

- Non-associative or non-commutative binary op used with std::reduce or std::transform_reduce
- RePhrase rule 4.8

Useful concurrency sources

- C++17 Draft
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf
- Anthony Williams
  - C++ Concurrency in Action 2nd Ed
- Scott Meyers
  - Effective Modern C++
- Herb Sutter
  - Effective Concurrency
  - www.herbsutter.com/2010/09/24/effective-concurrency-know-when-to-use-an-active-object-instead-of-a-mutex/

Multi-threading misuse rules

- HIC++ Coding Standard
  - www.codingstandard.com/section/18-concurrency/
  - C++11 Thread support library issues
- CWE and CERT C++ concurrency rules
  - wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046460
  - data race
  - deadlock
  - use of non-reentrant functions
- RePhrase Coding Standard
  - rephrase-eu.weebly.com/uploads/3/1/0/9/31098995/hicppmp.pdf
  - Completely focused on concurrency
  - More rules than presented here

Concurrent – what’s wrong

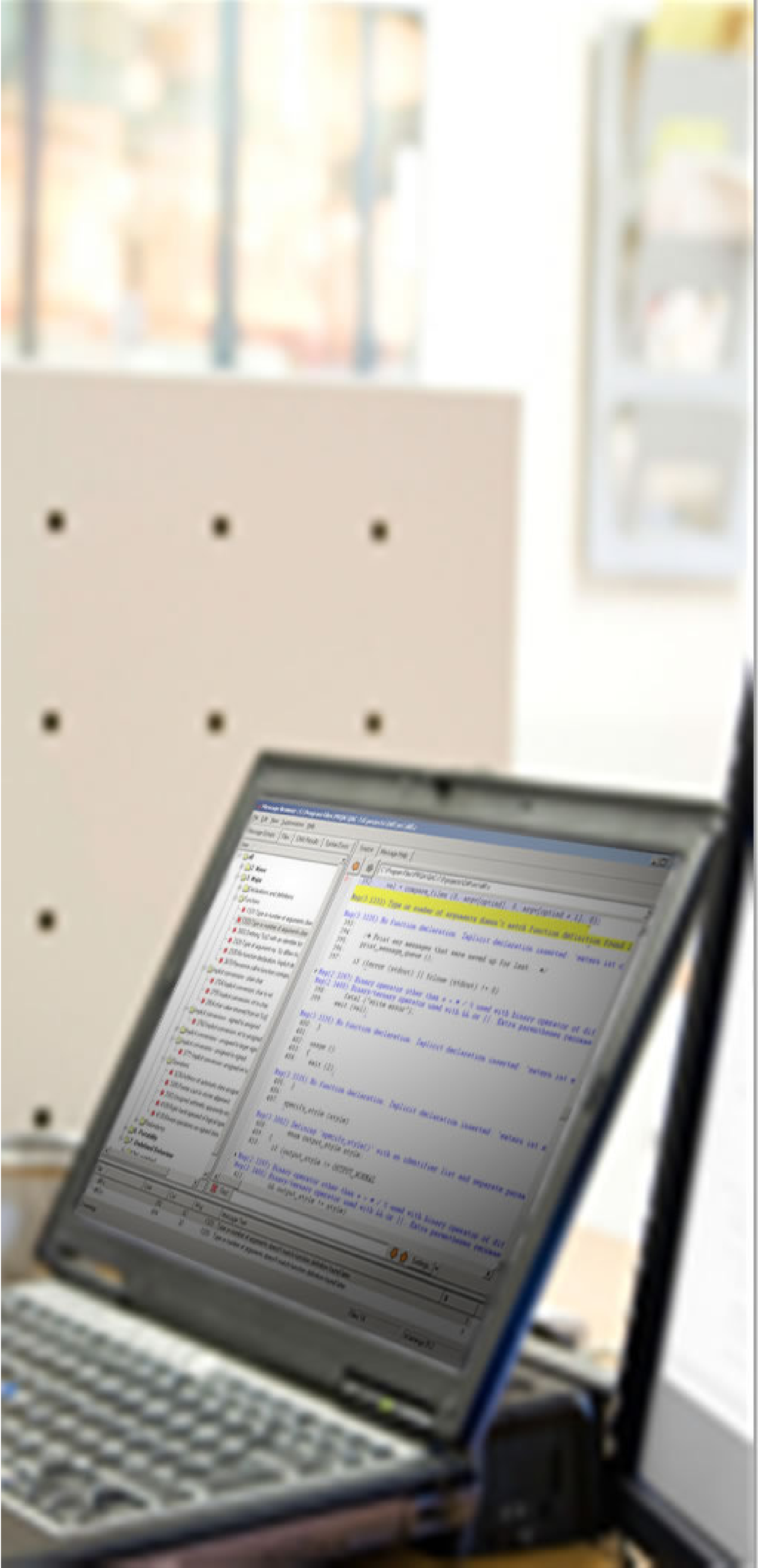
```
void concurrent () {
  std::mutex m; std::condition_variable cond; std::queue<int> data;

  auto f = std::async ([&m, &cond, &data] () {
    for (unsigned i = 0; i != 10; ++i) {
      const int value = reentrantFunc (i);
      std::lock_guard<std::mutex> guard (m);
      data.push (value);
      cond.notify_one ();
    }
  });

  for (unsigned i = 0; i != 10; ++i) {
    std::unique_lock<std::mutex> lock (m);
    cond.wait (lock, [&data] () { return ! data.empty(); });
    int result = data.front ();
    data.pop ();
    lock.unlock ();
    // do something with ‘result’
  }
  f.wait();
}
```

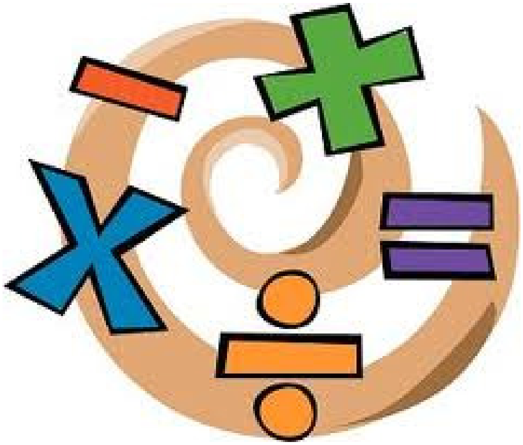
Concurrent – fixed

```
void concurrent () {
  std::mutex m; std::condition_variable cond; std::queue<int> data;

  auto f = std::async (std::launch::async, [&m, &cond, &data] () {
    for (unsigned i = 0; i != 10; ++i) {
      const int value = reentrantFunc (i);
      std::lock_guard<std::mutex> guard (m);
      data.push (value);
      cond.notify_one ();
    }
  });

  for (unsigned i = 0; i != 10; ++i) {
    std::unique_lock<std::mutex> lock (m);
    cond.wait (lock, [&data] () { return ! data.empty(); });
    int result = data.front ();
    data.pop ();
    lock.unlock ();
    // do something with ‘result’
  }
  f.wait();
}
```

Concurrent – rules

- Explicitly specify a launch policy for std::async
  - RePhrase rule 3.3
  - Default is std::launch::async | std::launch::deferred
- …





Questions?

wojciech_basalaj@prqa.com
richard_corden@prqa.com