Math 4997-3

Lecture 7: Asynchronous programming

https://www.cct.lsu.edu/~pdiehl/teaching/2019/4977/

Reminder

Lecture 6

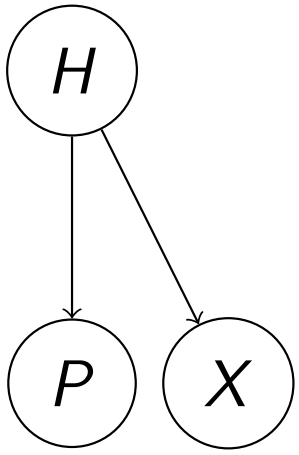
What you should know from last lecture

- Shared memory parallelism
- Parallel algorithms and execution policies
- Data races and dead locks

Asynchronous programming

Synchronous programming

Dependency graph



Code

```
auto P = compute();
auto X = compute();
auto H = compute(P,X);
```

- The program is executed line by line
- Each time a function is called the code waits until the functions finishes
- We can not compute P and X at the same time, since the data is independent

Asynchronous programming [3]

Code

```
int P,X = 1;

std::future<int> f1 = std::async(compute,P);
auto f2 = std::async(compute,X);

std::cout << compute(f1.get() + f2.get()) << std::endl;
```

- The program is some times executed line by line
- Calling `std::async` the next line is executed, even if the function has not finished yet
- We have to use the `std::future` to synchronize the asynchronous function calls

More details: CppCon 2017: H. Kaiser "The Asynchronous C++ Parallel Programming Model"[1]

[1] `https://www.youtube.com/watch?v=js-e8xAMd1s`

Asynchronous execution of functions[2]

```
bool is_prime (int x) {
  std::cout << "Calculating. Please, wait...\n";
  for (int i=2; i<x; ++i) if (x%i==0) return false;
  return true;
}

std::future<bool> f = std::async (is_prime,313222313);
```

- The first argument `fn` is a function pointer
- The second argument is the first argument of the function, and so on
- The return value is a `std::future<T>` where `T` is the return type of the function

For each call of `std::async` launches a new thread to execute the function the function pointer `fn` points to.

---

[2] `http://www.cplusplus.com/reference/future/async/`

Futurization[3]

A `std::future` provides a mechanism to access the result of asynchronous operations, like `std::async` and provides methods for synchronization.

Synchronization

- `.get()` returns the result of the functions and wait until the computation finished
- `.wait()` waits until the computation finished
- `.wait_for(std::chrono::seconds(1))` returns if it is not available for the specified timeout duration
- `.wait_until(std::chrono::seconds(1))` waits for a result to become available. It blocks until specified timeout time has been reached or the result becomes available, whichever comes first.

[3] `https://en.cppreference.com/w/cpp/thread/future`

Parallelism using asynchronous programming

Example: Taylor series

$$sin(x) = \sum_{n=0}^{n} (-1)^{n-1} \frac{x^{2n}}{(2n)!}$$

Approach

1. Split n into slices, e.g. 2 times $n/2$ for two threads
2. Start two times `std::async` where each thread computes $n/2$
3. Use the two futures to synchronize the results
4. Combine the two futures to obtain the result

Implementation I

Function

```
double taylor(size_t begin, size_t end,
double x,size_t n){
double res = 0;

for( size_t i = begin ; i < end ; i++)
{
res += pow(-1,i-1) * pow(x,2*n) / factorial(2*n);
}
return res;
}
```

- With `begin` and `end`, the range is defined
- The range needs to be adapted to the amount of threads you want to launch

Implementation II

Launching

```
auto f1 = std::async(taylor,0,49,2,100);
auto f2 = std::async(taylor,50,99,2,100);
```

Gathering the results

```
double result = f1.get() + f2.get();
```

Compilation

```
g++ main.cpp -o futures -phtread
```

We need to add `-pthread` to our compiler to use the POSIX threads to launch the functions asynchronous (`std::async`)

More details about POSIX threads [1, 2].

Lambda functions

Lambda expression[4]

Structure

```
[ capture clause ] (parameters) -> return-type
{
   definition of method
}
```

Notes

- Generally return-type in lambda expression are evaluated by compiler
- Capture clause:
  - `[&]` : capture all external variable by reference
  - `[=]` : capture all external variable by value
  - `[a, &b]` : capture a by value and b by reference

  More about the capture clauses in lecture 11/12.

[4] https://en.cppreference.com/w/cpp/language/lambda

Practical example

```
std::vector<int> v {4, 1, 3, 5, 2, 3, 1, 7};
```

Classical function

```
void print(int i){
std::cout << i << std::endl;
}
std::for_each(v.begin(), v.end(), print);
```

Lambda expression

```
std::for_each(v.begin(),v.end(),
        [](int i){std::cout<< i << std::endl;})
```

More examples

`find_if`[5]

```
std::vector<int>:: iterator p = find_if(
v.begin(),
    v.end(),
    [](int i)
{
    return i > 4;
});
std::cout << "First number greater than 4 is :
" << *p
<< endl;
```

Many more algorithms are available in the
`#include <algorithm>`[6]

[5] https://en.cppreference.com/w/cpp/algorithm/find
[6] https://en.cppreference.com/w/cpp/algorithm

Summary

Summary

After this lecture, you should know

- Asynchronous programming `std::async` and `std::future`
- Lambda functions

References

References I


David R Butenhof.
Programming with POSIX threads.
Addison-Wesley Professional, 1997.


Steve Kleiman, Devang Shah, and Bart Smaalders.
Programming with threads.
Sun Soft Press Mountain View, 1996.


Anthony Williams.
C++ concurrency in action : practical multithreading.
Manning, Shelter Island, NY, 2012.