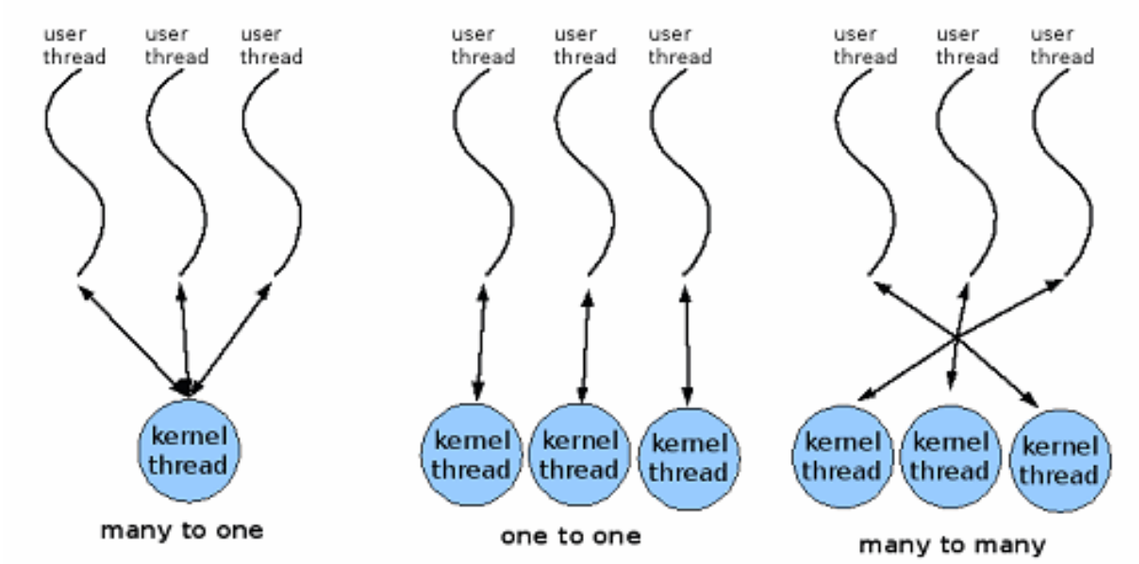

user thread
user thread
user thread
kernel thread
many to one
user thread
user thread
user thread
kernel thread
kernel thread
kernel thread
one to one
user thread
user thread
user thread
kernel thread
kernel thread
kernel thread
many to many

```cpp
void hello()
{
   std::cout << "Hello World! thread id = "
     << std::this_thread::get_id() << std::endl;
}

int main(int argc, char** argv)
{
  std::thread t(hello);
  return 0;
}
```

```
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_spawn$
clang++ -std=c++14 thread1.cpp
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_spawn$ ./
a.out
libc++abi.dylib: terminating
Abort trap: 6
```

std::thread

- We ran ~thread(); when we exited from the scope of main and it destroyed the thread object.

- **If *this has an associated thread std::terminate() is called**

- We hold a thread as long as:
  (joinable() == true)

- We have to wait that the thread terminates its execution either calling join or detach

Fix

```
void hello()
{
   std::cout << "Hello World! thread id = " << std::this_thread::get_id() << std::endl;
}

int main(int argc, char** argv)
{
  std::thread t(hello);
  t.join();
  return 0;
}
```

```
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_spawn$
clang++ -std=c++14 thread1.cpp
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_spawn$
./a.out
Hello World! thread id = 0x102909000
```

will this code work?

```
int main(int argc, char** argv)
{
   // lambda function
   auto f = []()
   {
    std::cout << "ID of this thread = " <<std::this_thread::get_id() << std::endl;
    };

    scoped_thread th(std::thread{f});

   return 0;
}
```

scoped thread

```
class scoped_thread
{
    std::thread t_;
public:
    explicit scoped_thread(std::thread t ) : t_(std::move(t))
    {
        if(t_.joinable() == false )
            std::logic_error("This is not a thread!!");

    }
    ~scoped_thread()
    {
       if(t_.joinable())
         t_.join();
    }

    scoped_thread(scoped_thread&& x) : t_(std::move(x.t_))
    {}

    scoped_thread(scoped_thread&) = delete;
    scoped_thread& operator=(const scoped_thread&) = delete;
};
```

join vs detach

```
void complex_fnt( std::string& s)
{
   std::stringstream buff;
   std::ofstream f("./test.txt");
   //write to file string passed + thread id
   if(f.is_open())
   {
      buff << s << std::this_thread::get_id() << std::endl;
      f << buff.str();
      f.close();
   }
}


   std::string s = "Hello world ";

   //complex object passed by reference
   std::thread t(complex_fnt, std::ref(s));

   //assure job is done and return
   t.join();
```

join vs detach

```
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_spawn$ ./a.out  1
-- Join a complex task --
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_spawn$ ls
a.out*           fancy_object.h   test.txt         thread2.cpp      thread4.cpp
async_check.cpp  scoped_thread.h  thread1.cpp      thread3.cpp
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_spawn$ cat test.txt
Hello world 0x103c27000
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_spawn$
```

join vs detach

```
void complex_fnt( std::string& s)
{
   std::stringstream buff;
   std::ofstream f("./test.txt");
   //write to file string passed + thread id
   if(f.is_open())
   {
      buff << s << std::this_thread::get_id() << std::endl;
      f << buff.str();
      f.close();
    }
}


  std::string s = "Hello world ";
  //complex object passed by reference
  std::thread t(complex_fnt, std::ref(s));

  //detach the thread... if the program exits before the threads
  completes its job, no job is done
  t.detach();
```

join vs detach

```
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_spawn$ ./a.out  2
-- Detach a complex task --
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_spawn$ ls
a.out*          async_check.cpp  fancy_object.h   scoped_thread.h  thread1.cpp
thread2.cpp     thread3.cpp      thread4.cpp
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_spawn$
```

join vs detach

```
std::string s = "Hello world ";
//complex object passed by reference
std::thread t(complex_fnt, std::ref(s));
//detach the thread... if the program exits before the threads
completes its job, no job is done
t.detach();

//wait 5ms in order to assure that file is written on disk
std::this_thread::sleep_for(std::chrono::nanoseconds(x));
```

join vs detach

```
./a.out 2 5000
-- Detach a complex task --
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_spawn$ ls
a.out*           fancy_object.h   test.txt         thread2.cpp      thread4.cpp
async_check.cpp  scoped_thread.h  thread1.cpp      thread3.cpp
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_spawn$ cat test.txt
Hello world 0x10f069000
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_spawn$
```

Be sure all resources that your detached thread uses are still alive even when your program terminates

std::mutex and std::lock_guard

```
void fnt(int id, std::string s)
{
   std::cout << "id # = " << id
         << " - Functional object - I am a thread with ID = "
         << std::this_thread::get_id()
         << " custom msg = "<< s
         <<std::endl;
}


  std::vector<std::thread> threads;
  threads.emplace_back(fnt,0,"Hi");
  threads.emplace_back(fnt,1,"Salut");
  threads.emplace_back(fnt,2,"Ciao");
  threads.emplace_back(fnt,3,"Hola");

  for(auto& t: threads )
   t.join();
```

Output ... ???

```
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_spawn$
clang++ -std=c++11 thread4.cpp
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_spawn$
./a.out  0
iiiidddd    ####    ====    0123    ----
FFFFuuuunnnncccctttttiiiioooonnnnaaaallll    ooooobbbbjjjjeeeeccccttttt    ----
IIII    aaaammmm    aaaa    tttthhhhrrrreeeeaaaadddd    wwwwiiiitttthhhh
IIIIDDDD    ====    0000xxxx1111000077773344080814 7a000000000000
ccccuuuussssttttoooommmm    mmmmssssgggg    ====    HSCHiaio
laluoat


nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_spawn$
```

std::mutex and std::lock_guard

```
std::mutex _m;

void fnt(int id, std::string s)
{
    std::lock_guard<std::mutex> lk{_m};

    std::cout << "id # = " << id
     << " - Functional object - I am a thread with ID = “
     << std::this_thread::get_id()
     << " custom msg = "<< s
     <<std::endl;
}
```

Output

```
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_spawn$ ./a.out  0
id # = 0 - Functional object - I am a thread with ID = 0x10051d000 custom msg = Hi
id # = 1 - Functional object - I am a thread with ID = 0x1005a0000 custom msg =
Salut
id # = 3 - Functional object - I am a thread with ID = 0x1006a6000 custom msg = Hola
id # = 2 - Functional object - I am a thread with ID = 0x100623000 custom msg = Ciao
```

for more details about mutexs
see: http://en.cppreference.com/w/cpp/thread

High level Interface

(not all APIs will be discussed)

Hardware Threads vs Software Threads

- 1 Core today == 2 hardware threads (due to hyper-threading)

- How can we understand how many software threads I can really run in parallel ???

- Usually more threads my application spawns (exceeding the number of hardware threads) and less job I get done…. let’s see in few slides

Get physical threads

```
const unsigned physical_thread_number = std::thread::hardware_concurrency();

 std::cout << "Available physical thread = "
 << available_threads << std::endl;
```

```
~/GitHub/cpp_sandbox/multithreading/thread_sample$ ./a.out  100 10
 …
Available physical thread = 4
 …
```

std::async + std::future

```
std::mutex m;
using lock = std::lock_guard<std::mutex>;
std::map<std::thread::id,bool> ids;

void f(unsigned i)
{
    lock lk{m};
    auto id = std::this_thread::get_id();
    std::cout << "thread #"<<i<< " id = " << id << std::endl;
    ids.insert(std::make_pair(id, false));
}


std::vector<std::future<void>> futures(10);

for(unsigned i = 0; i<10; ++i)
  futures[i] = std::async(std::launch::any, f, i);

for(auto&f: futures )
  f.wait();
```

Possible output

```
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_spawn$ ./a.out
thread #0 id = 0x1041f0000
thread #1 id = 0x104273000
thread #2 id = 0x1042f6000
thread #3 id = 0x1041f0000
thread #4 id = 0x104273000
thread #5 id = 0x1042f6000
thread #6 id = 0x1041f0000
thread #7 id = 0x104273000
thread #8 id = 0x1042f6000
thread #9 id = 0x1041f0000
Actual thread spawned = 3
0x1041f0000
0x104273000
0x1042f6000
```

std::async

```
template< class Function, class... Args >
std::future<typename std::result_of<Function(Args...)>::type>
async( std::launch policy, Function&& f, Args&&... args );
```

- The template function async runs the function f asynchronously (potentially in a separate thread which may be part of a thread pool) and returns a std::future that will eventually hold the result of that function call.
- Policies to spawn computation are:
  - std::launch::async
  - std::launch::deferred
  - launch::any (bitwise or async | deferred)

from http://en.cppreference.com/w/cpp

std::future

```
template< class T > class future;
template< class T > class future<T&>;
template<>          class future<void>;
```

- The class template std::future provides a mechanism to access the result of asynchronous operations.
- You can use one of these objects to get a std::future back
  - std::promise
  - std::package_task
  - std::async

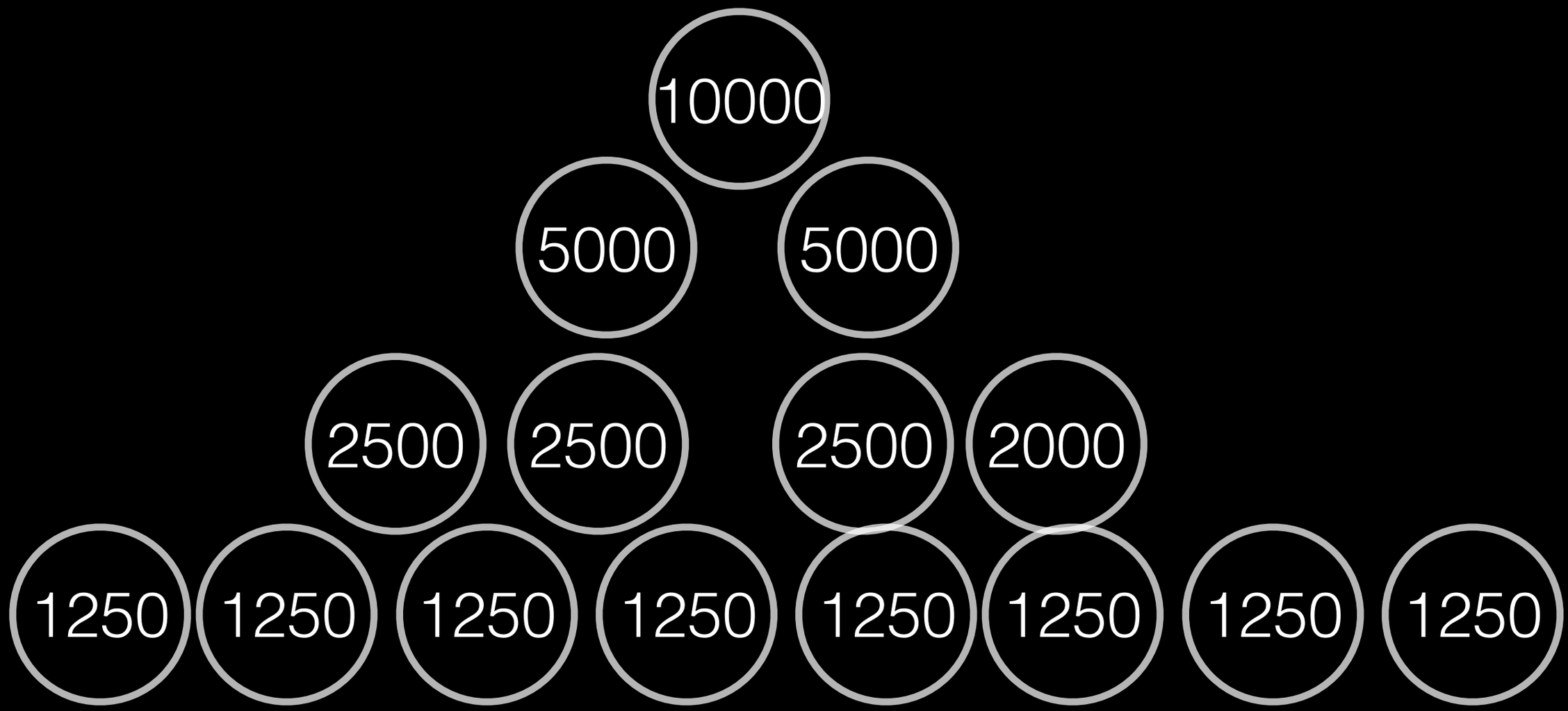
from http://en.cppreference.com/w/cpp

Write parallel code

Problem:

Compute parallel sum of an array of integers

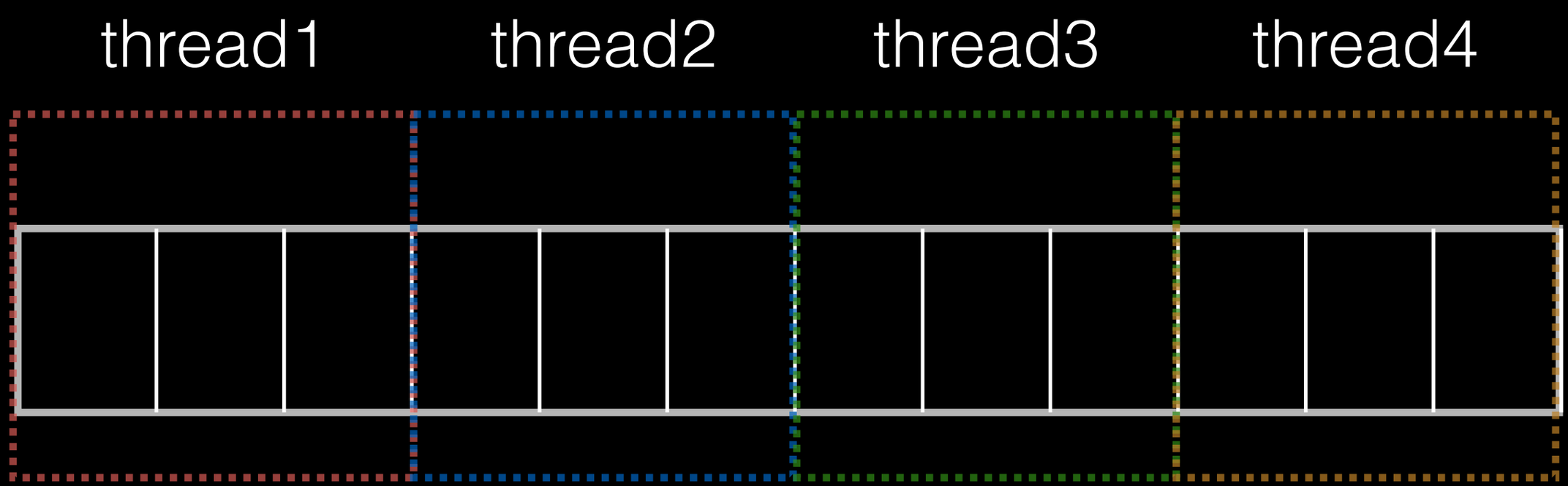
N 10000
block size = 1250



```
template<typename I>
long async_accumulate(I begin, I end)
{
    const auto len = end - begin;
    if (len <= block)
        return std::accumulate(begin, end, 0);

    I mid = begin + len / 2;
    auto handle = std::async(async_accumulate<I>, mid, end);
    int sum = async_accumulate<I>(begin, mid);
    return sum + handle.get();

}
```


thread1
thread2
thread3
thread4

```cpp
template<typename I>
long async_accumulate(I begin, I end)
{
   auto len = std::distance(begin, end);

   if (len == 0)
      return 0;

   const unsigned num_threads = compute_number_of_threads(len);
   std::vector<std::future<long>> res(num_threads-1);
   const unsigned block_size = len / num_threads;

   auto f =[](I b, I e, long init) { return std::accumulate(b,e,init); };

   auto start_block = begin;
   for(unsigned i =0; i < num_threads-1; ++i)
   {
      auto end_block  = start_block;
      std::advance(end_block, block_size);
      res[i] = std::async(f, start_block, end_block, 0);
      start_block = end_block;
   }

   long sum = f(start_block, end, 0);
   for(auto& f : res )
      sum += f.get();

   return sum;
}
```

Results

N = 10000 - Block = 1000

	Single thread	Divide/ Conquer	Array Split
	86us	614us	389us

N = 100000 - Block = 10000

	Single thread	Divide/ Conquer	Array Split
	510us	593us	458us

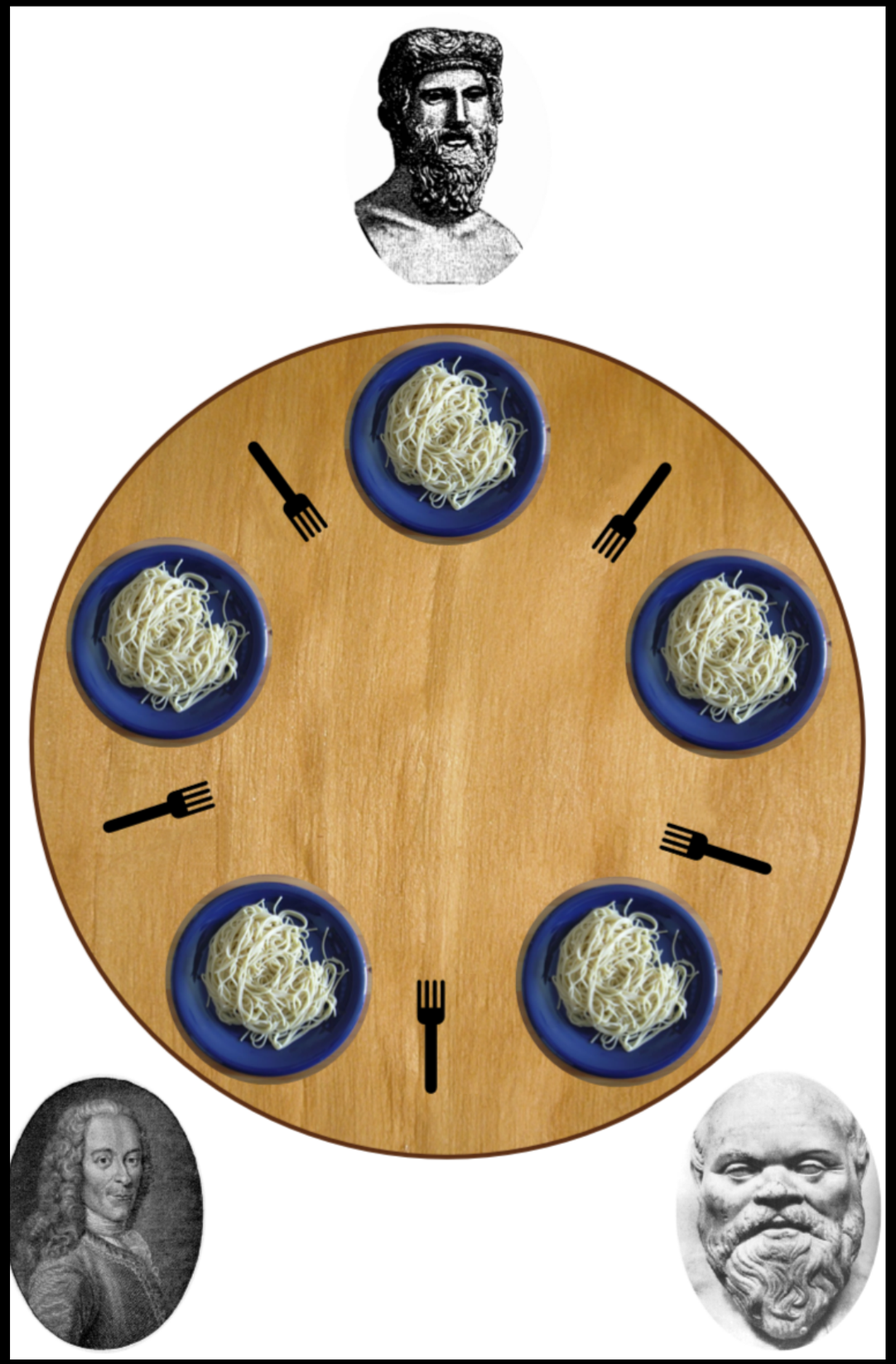
N = 1000000 - Block = 100000

	Single thread	Divide/ Conquer	Array Split
	~5.2 ms	~3.1 ms	~3.1 ms

Message Queue

Dining philosophers

Many producers
Many consumers



```cpp
template<typename T>
class message_queue
{
   std::mutex _m;
   std::queue<T> _q;

    using lock_guard = std::lock_guard<std::mutex>;

public:
    using value_type = T;

    message_queue() = default;
    message_queue(const message_queue& ) = delete;
    message_queue& operator=(const message_queue&) = delete;

  void push(T item)
  {
      lock_guard lk{_m};
      _q.push(std::move(item));
  }

bool pop(T& t)
  {
      lock_guard lk{_m};
      if(_q.empty())
          return false;

      t = std::move(_q.front());
      _q.pop();
      return true;
  }
```

Producer

```
void push(Q& q, unsigned id)
{
    for(unsigned i=id*LOOP_SIZE; i<(id+1)*LOOP_SIZE; ++i)
        q.push(i);
}
```

Consumer

```
void pop(Q& q)
{
    unsigned i;
    while(true)
    {
        q.pop(i);
    }
}
```

N Producers/ M Consumers

```
std::vector<std::future<void>> consumers(M);
std::vector<std::future<void>> producers(N);

int i = 0;
for (auto& p : producers)
  p = std::async(std::launch::async, push, std::ref(q), i++);

for (auto& c : consumers)
  c = std::async(std::launch::async, pop, std::ref(q));

for (auto& c : consumers)
  c.wait();

for (auto& p : producers)
  p.wait();
```

```
Processes: 284 total, 3 running, 6 stuck, 275 sleeping, 1329 threads
15:37:55
Load Avg: 1.52, 1.58, 1.49  CPU usage: 7.0% user, 31.40% sys, 61.59% idle   SharedLibs: 1196K resident,
0B data, 0B linkedit.
MemRegions: 90828 total, 2462M resident, 65M private, 620M shared. PhysMem: 8116M used (1303M wired), 73M
unused.
VM: 724G vsize, 1026M framework vsize, 8785612(0) swapins, 9281977(0) swapouts. Networks: packets:
26748134/33G in, 15538508/3923M out.
Disks: 2499813/95G read, 3051851/126G written.

PID    COMMAND      %CPU  TIME     #TH   #WQ  #PORT MEM    PURG   CMPRS  PGRP  PPID  STATE    BOOSTS
%CPU_ME %CPU_OTHRS UID  FAULTS
26591  top          3.4   00:00.89 1/1   0    19    3084K  0B     0B     26591 25987 running  *0[1]
0.00000 0.00000    0    39637+
26590  a.out        99.8  00:22.29 5/1   0    13    368K   0B     0B     26590 25842 running  *0[1]
0.00000 0.00000    501  342
26585  Python       0.0   00:01.75 32    0    53    37M    0B     0B     26585 26575 sleeping *0[1]
0.00000 0.00000    501  31977
26575  Vim          0.0   00:01.60 31    0    71    22M    0B     0B     26575 5829  sleeping *0[4]
0.00000 0.00000    501  13019
```

Signal events

Signal events

```
std::condition_variable _cv;
```

Push and notify

```
void push_and_notify(T item)
{
  {
    lock_guard lk{ _m };
     _q.push(std::move(item));
  }
   _cv.notify_all();
}
```

sleep until there are new items to pop

```
void wait_and_pop(T& item)
{
  std::unique_lock<std::mutex> lk{ _m };
  _cv.wait(lk, [this]{ return !_q.empty(); });
   item = std::move(_q.front());
  _q.pop();
}
```

```
Processes: 284 total, 3 running, 5 stuck, 276 sleeping, 1340 threads
15:50:13
Load Avg: 1.52, 1.47, 1.48  CPU usage: 2.62% user, 3.10% sys, 94.27% idle   SharedLibs: 1196K resident, 0B data,
0B linkedit.
MemRegions: 90613 total, 2434M resident, 65M private, 640M shared. PhysMem: 8119M used (1295M wired), 72M
unused.
VM: 725G vsize, 1026M framework vsize, 8785612(0) swapins, 9281977(0) swapouts. Networks: packets: 26752134/33G
in, 15542577/3924M out.
Disks: 2508267/95G read, 3057143/126G written.

PID    COMMAND      %CPU  TIME     #TH   #WQ   #PORT MEM    PURG   CMPRS  PGRP  PPID  STATE    BOOSTS
%CPU_ME %CPU_OTHRS UID  FAULTS
26631  a.out        0.0   00:00.00 5     0     14    368K   0B     0B     26631 25842 sleeping *0[1]
0.00000 0.00000    501  344
26627  Python       0.0   00:01.64 32    0     53    35M    0B     0B     26627 26617 sleeping *0[1]
0.00000 0.00000    501  33469
26617  Vim          0.0   00:01.12 31    0     71    22M    0B     0B     26617 5829  sleeping *0[5]
0.00000 0.00000    501  13083
26595  QuickLookSat 0.0   00:00.36 2     0     41    11M    0B     0B     26595 1     sleeping  0[0]
0.00000 0.00000    501
```

- if you need more performances you can
    - decrease granularity of locks (example implementing a forward linked list with double pointers to head and tail)
    - lock free algorithms (more error prone though)

DeadLocks

```
std::vector<std::future<void>> tasks;

for(unsigned i = 0; i < 1; ++i)
{
  tasks.push_back(std::async(transfer,0,1,10));
  tasks.push_back(std::async(transfer,1,0,10));
}
```

```
void transfer(int from, int to, int sum)
{
    // deadlock
    auto& acc1 = _accounts[from];
    auto& acc2 = _accounts[to];

    lock_guard lk1(acc1.get_mutex());
    lock_guard lk2(acc2.get_mutex());

    std::cout << "Moving money from = " << from
      << " to = " << to << " sum = " << sum << "\n";

      if (acc1.balance() >= sum)
      {
          acc1.deposit(-sum);
          acc2.deposit(sum);
      }
}
```

```
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_sync$
clang++ -std=c++14 deadlock.cpp
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_sync$
./a.out
Moving money from = 0 to = 1 sum = 10
```

```
void transfer(int from, int to, int sum)
{
        // no deadlock
        auto& acc1 = _accounts[from];
        auto& acc2 = _accounts[to];

        std::lock(acc1.get_mutex(), acc2.get_mutex());
        lock_guard lk1(acc1.get_mutex(), std::adopt_lock);
        lock_guard lk2(acc2.get_mutex(), std::adopt_lock);
        std::cout << "Moving money from = " << from
        << " to = " << to << " sum = " << sum << "\n";

        if (acc1.balance() >= sum)
        {
            acc1.deposit(-sum);
            acc2.deposit(sum);
        }
}
```

```
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_sync$
clang++ -std=c++14 deadlock.cpp
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_sync$
./a.out
Moving money from = 0 to = 1 sum = 10
Moving money from = 1 to = 0 sum = 10
Program end ... Balance 0 = 100 - Balance 1 = 0
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_sync$
```

Memory model

- The biggest introduction made in C++11
- Memory memory deals with:
    - atomic operations
    - visible effects of these operations
    - atomic access to object members given memory layout

is this code thread safe??

```
struct S
{
  long a;
  char b;
 };
```

only C++11

```
//update a
auto f = [&s]()
{
   s.a = 10;
};

//upddate b
auto g = [&s]()
{
   s.b = 20;
};
```

```
//sample how to exploit memory model objects layout
auto fut1 = std::async(std::launch::async,f);
auto fut2 = std::async(std::launch::async,g);

fut1.wait();
fut2.wait();
```

std::atomic

```
template< class T > struct atomic
template<> struct atomic<Integral>;
template< class T > struct atomic<T*>;
```

- Each instantiation and full specialization of the std::atomic<> template defines an atomic type.

- Objects of atomic types are the only C++ objects that are free from data races; that is

- if one thread writes to an atomic object while another thread reads from it, the behaviour is well-defined.

from http://en.cppreference.com/w/cpp

```cpp
std::atomic<double> b;
std::atomic<long> c;
std::atomic<int> d;
std::atomic<short> e;
std::atomic<char> f;
std::atomic<long long> g;

std::cout << "double is atomic = "<< std::boolalpha <<  b.is_lock_free() << std::endl;
std::cout << "long is atomic = "<< std::boolalpha <<  c.is_lock_free() << std::endl;
std::cout << "int is atomic = "<< std::boolalpha <<  d.is_lock_free() << std::endl;
std::cout << "short is atomic = "<< std::boolalpha <<  e.is_lock_free() << std::endl;
std::cout << "char is atomic = "<< std::boolalpha <<  f.is_lock_free() << std::endl;
std::cout << "long long is atomic = "<< std::boolalpha <<  g.is_lock_free() <<
std::endl;
```

```
clang++ -std=c++14 atomic.cpp
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_sync$ ./a.out
double is atomic = true
long is atomic = true
int is atomic = true
short is atomic = true
char is atomic = true
long long is atomic = true
```

```cpp
std::atomic<int> ay{0};
int x = 0;

void atomic_read_barrier()
{
    std::cout << "y = " << ay.load() << std::endl;
    std::cout << "x = " << x << std::endl;
    std::cout << std::endl;
}

void atomic_write_barrier()
{
    x = 42;
    ay.store(20);
}

std::thread t2(atomic_read_barrier);
std::thread t1(atomic_write_barrier);
t1.join();
t2.join();
```

x	y
0	0
42	0
42	20

```
void th_read_lock()
{
    lock lk{m};

    std::cout << "x = " << x << std::endl;
    std::cout << "y = " << y << std::endl;
    std::cout << std::endl;
}
void th_write_lock()
{
    lock lk{m};
    x = 42;
    y = 20;
}
```

```
std::thread t2(th_read_lock);
std::thread t1(th_write_lock);
t1.join();
t2.join();
```

x	y
0	0
42	20

Cool.. but can I craft something useful with all this stuff???

```
class spinlock
{
    std::atomic_flag _flag;
public:
    spinlock() : _flag(ATOMIC_FLAG_INIT)
    {}

    spinlock(const spinlock&) = delete;
    spinlock& operator=(const spinlock&) = delete;
    spinlock(spinlock&&) = default;

    void lock()
    {
        while(_flag.test_and_set(std::memory_order_acquire));
    }

    void unlock()
    {
        _flag.clear(std::memory_order_release);
    }
};
```

```cpp
void th_write(spinlock& spin, int& a )
{
    spinlock_guard lk{spin};
    a = 42;
}

void th_read(spinlock& spin, int& a)
{
    spinlock_guard lk{spin};
    std::cout << " value = " << a << "\n";
}
```

```
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_sync$
clang++ -std=c++14 spinlock.cpp
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_sync$ ./a.out
 value = 42
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_sync$ ./a.out
 value = 42
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_sync$ ./a.out
 value = 0
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_sync$ ./a.out
 value = 0
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_sync$ ./a.out
 value = 42
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_sync$ ./a.out
 value = 42
nik@Nicolas-MacBook-Air:~/GitHub/cpp_sandbox/multithreading/thread_sync$ ./a.out
 value = 42
```

Point to…

GitHub where I put all the code (there is also stuff I haven't shown)

https://github.com/nicola-cab/cpp_sandbox/tree/master/multithreading

Resources used

- http://www.amazon.com/C-Concurrency-Action-Practical-Multithreading/dp/1933988770
- https://www.justsoftwaresolutions.co.uk/blog/
- http://herbsutter.com/category/effective-concurrency/

Thanks…

Questions?