- Solution

Let's look at the solution to the exercise from the previous lesson.

WE'LL COVER THE FOLLOWING

Solution

```
#include <algorithm>
#include <future>
#include <iostream>
#include <thread>
#include <deque>
#include <vector>

class SumUp{
public:
  SumUp(int b, int e): beg(b), end(e){}
  int operator()(){
    long long int sum{0};
    for (int i= beg; i < end; ++i ) sum += i;
    return sum;
  }
private:
    int beg;
    int end;
};

static const unsigned int hwGuess = 4;
static const unsigned int numbers = 10001;

int main(){

  std::cout << std::endl;

  unsigned int hw = std::thread::hardware_concurrency();
  unsigned int hwConcurr = (hw != 0)? hw : hwGuess;

  // define the functors
  std::vector<SumUp> sumUp;
  for (unsigned int i = 0; i < hwConcurr; ++i){
    int begin = (i*numbers)/hwConcurr;
    int end = (i+1)*numbers/hwConcurr;
```

```
    sumUp.push_back(SumUp(begin , end));
  }

  // define the tasks
  std::deque<std::packaged_task<int()>> sumTask;
  for (unsigned int i = 0; i < hwConcurr; ++i){
    std::packaged_task<int()> SumTask(sumUp[i]);
    sumTask.push_back(std::move(SumTask));
  }

  // get the futures
  std::vector< std::future<int>> sumResult;
  for (unsigned int i = 0; i < hwConcurr; ++i){
    sumResult.push_back(sumTask[i].get_future());
  }

  // execute each task in a separate thread
  while (! sumTask.empty()){
    std::packaged_task<int()> myTask = std::move(sumTask.front());
    sumTask.pop_front();
    std::thread sumThread(std::move(myTask));
    sumThread.detach();
  }

  // get the results
  int sum = 0;
  for (unsigned int i = 0; i < hwConcurr; ++i){
    sum += sumResult[i].get();
  }

  std::cout << "sum of 0 .. 100000 = " << sum << std::endl;

  std::cout << std::endl;

}
```

Explanation

C++11 has a function `std::thread_hardware_concurrency`. It provides a hint as to the number of cores on our system. In this case, the C++ runtime has no clue.It's conforming to the standard to return 0. So we should verify that value in our program (lines 28 and 29). With the current GCC, clang, or Microsoft compiler, we get the right answer: 4. We used the number of cores to alter the program from the last example we saw, in order to adjust the software according to our hardware.

In the next lesson, we'll discuss the class templates `std::promise` and `std::future` which provide us full control over tasks.