- Solution

In the lesson, we will discuss the solution to the task of the previous exercise.

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 the function `std::thread_hardware_concurrency`. It provides a hint for the numbers of cores in your system.

- In case the C++ runtime has no clue, it is conforming to the standard to return 0. You should verify that value in your program (line 29).

- With the current compiler, we get the right answer, 4. We use this number of cores in the variation of the above program to adjust the software to our hardware, making it fully utilized.

For further information, read std::packaged_task

In the next lesson, we will see how class templates `std::promise` and `std::future` provide you full control over tasks.