Advanced Programming

Mini-Introduction to Parallel Programming in C++

Parallel programming allows a program to execute multiple tasks simultaneously, improving performance, especially on multi-core processors. C++ provides several tools and libraries for parallel programming, enabling developers to write efficient, concurrent code.

This document serves as supplementary material to introduce basic parallel programming concepts in C++, focusing on threads (`std::thread` class), and OpenMP

1 Threads and the `std::thread` Class

A thread is a sequence of instructions that can be executed independently of other code.

1.1 Creating and Managing Threads

Threads in C++ are managed using the 'std::thread' class. Let's see an example:

```cpp
#include <thread>
#include <iostream>

// define a function to be executed by a thread
void complete_msg() {
            std::cout << "task" << std::endl;
}

int main()
{
    std::thread t(complete_msg); // Create a new thread that runs our function
    std::cout << "Hello from: ";
    t.join(); // Wait for the thread to finish
    return 0;
}
```

The output of this code is `Hello from: task`. In this case a thread `t` is created and runs the function `complete_msg`, however, to get the output from the thread we need to use the function `join()`. This function waits for the thread to finish before continuing with the execution of the rest of the code. This is important when the output of functions running on a thread are required further in the code. Alternatively, we can let the thread run independently of the rest of the code by detaching threads. This can be useful for executing some tasks (e.g writing files) that do not affect execution.

```cpp
#include <iostream>
#include <fstream>
#include <thread>

void write_hello() {
    // Create and open a text file
    std::ofstream msg_file("msg.txt");

    // Write to the file
    msg_file << "Hello from file";

```

```
    // Close the file
    msg_file.close();
}

int main() {
    std::thread t(write_hello);
    t.detach(); // Detach the thread
    // Main thread continues without waiting for t
    std::cout << "Hello from terminal ";
    return 0;
}
```

Important: Detaching a thread means the main program will not wait for the thread to complete, which can lead to undefined behaviour if the thread accesses resources that are destroyed before it finishes.

1.2 Multiple threads

So far we have seen a example using threads for a single task. Let's see some examples creating more than one threads.

```
#include <thread>
#include <iostream>

// define functions to be executed by threads
void complete_msg1() {
           std::cout << "task 1" << std::endl;
}

void complete_msg2() {
           std::cout << "task 2" << std::endl;
}

int main()
{
        // Create new threads that run our functions
    std::thread t1(complete_msg1);
    std::thread t2(complete_msg2);
    std::cout << "Hello from: ";
    t1.join(); // Wait for the thread to finish
    std::cout << "Hello from: ";
    t2.join(); // Wait for the thread to finish

    return 0;
}
```

This approach is simple, but it's a little unpractical for largest number of tasks. Additionally, the output can be different as expected by just reading the code (Try yourself).

Alternatively, we can create a container of threads:

```
#include <thread>
#include <iostream>
#include <vector>

// define a function to be executed by threads
// include an argument to identify the task in the
// output
void complete_msg(int i) {
          std::cout << "Hello from task: " << i << std::endl;
}
```

```cpp

// define a function to create n threads
// this function returns an std:: vector
// with each element of type std::thread
std::vector<std::thread> create_threads(int n)
{
    // declare a vector of threads
    std::vector<std::thread> t_vec(n);
    for (int i = 0; i < n; i++) {
        // define each element of the vector as a thread that runs our function
        // note that in the secon parameter we are passing the argument
        // required by the function complete_msg
        t_vec[i] = std::thread(complete_msg, i);
    }
        return t_vec ;
}


int main()
{
    // define the number of threads
    int num_threads = 12;
    // create and run the threads
    std::vector<std::thread>t_vec = create_threads(num_threads);
    // remember to use join to see the output
    // we need to use join on each thread
    for (auto& t_i : t_vec) {
        t_i.join();
    }
    return 0;
}
```

Try yourself and check the output, particularly the order of the output in your screen.

Using threads through the `std::threads` class provides an option for executing tasks concurrently. This approach requires appropriate memory management, control flow and exception handling, making actual implementations complex. Additionally, creation of threads makes use of some resources, hence, it's not advisable to use threads for simple tasks.

2 OpenMP

OpenMP allows for multi-threaded shared memory parallelisation through compiler's directives.

Important: Include the `-fopenmp` flag for compilation, i.e `g++ -std=c++17 -fopenmp`

Let's see an example:

```cpp
#include <iostream>
#include <omp.h> // include the required library

// define a function to be executed by threads
// include an argument to identify the task in the
// output
void complete_msg(int i)
{
    std::cout << "Hello from task: " << i << std::endl;
}

int main()
{
    // define a parallel region
```

```
    #pragma omp parallel
    {
        // get thread id
        int thread_id = omp_get_thread_num();
        // print message on screen
        complete_msg(thread_id);
    }
    return 0;
}
```

The output of this code should be equivalent to the output of the previous example using the `std::threads` class. However, this option is shorter and simpler. The key is the use of the directive *`#pragma omp parallel`*. This directive creates a parallel section of code executed by all threads. The general structure for defining a a parallel region is:

```
// serial code

// definition of parallel section
#pragma omp parallel // some clauses can be added here
{
    // section of code to execute by threads
}

// more serial code
```

Contrary to `std::threads`, OpenMP automates threads creation and tasks parallelisation. By default, OpenMP creates as many threads as cores available in your computer. However, you can set the number of threads inside the code using `omp_set_num_threads(int)` or using the environment variable `OMP_NUM_THREADS`.

```
#include <iostream>
#include <omp.h>
#include <unistd.h>

void complete_msg(int i)
{
    std::cout << "Hello from task: " << i << std::endl;
}

int main()
{
    // define a parallel region with 5 threads
    #pragma omp parallel num_threads(5)
    {
        // get thread id
        int thread_id = omp_get_thread_num();
        // print message on screen
        complete_msg(thread_id);
    }
    return 0;
}
```

We have added the clause `num_threads(5)` to the *`#pragma omp parallel`* directive. This can be useful for some sections of code that should run on a specific number of threads, however, it can be restrictive. Alternatively, we can use the environment variable `OMP_NUM_THREADS`, e.g for a single run of the code:

`OMP_NUM_THREADS=4 ./main`

Since `OMP_NUM_THREADS` is an environment variable, you can set it for a terminal session (e.g using VSCode terminal) or permanently for a computer by adding it in the `.bashrc` file.

Additionally, OpenMP offers a set of directives suitable for particular applications, e.g `for` loops. For this example we will create a header file (`parallel_test.h`):

```cpp
#include <vector>
#include <cmath>
#include <chrono>
#include <ctime>
#include <omp.h>

// define a class for measuring execution time of serial and parallel calculations
class Timer
{
public:
    void start()
    {
        startTime = std::chrono::system_clock::now();
    }

    void stop()
    {
        endTime = std::chrono::system_clock::now();
    }

    double elapsedMilliseconds()
    {

        return std::chrono::duration_cast<std::chrono::milliseconds>(endTime -
        startTime).count();
    }

    double elapsedSeconds()
    {
        return elapsedMilliseconds() / 1000.0;
    }

private:
    std::chrono::time_point<std::chrono::system_clock> startTime;
    std::chrono::time_point<std::chrono::system_clock> endTime;
};

// define a class for calculating the average of values in a vector
class SumVector
{
public:
    std::vector<double> v;
    SumVector(std::vector<double> input_vector)
    {
        v = input_vector;
    }
    // serial
    double serial()
    {
        double ave = 0.;
        for (int i = 0; i < v.size(); ++i)
        {
            ave += v[i] / v.size();
        }
        return ave;
    }
    // parallel
    double parallel()
    {
```

```
        double ave = 0.;
        #pragma omp parallel for num_threads(4)
        for (int i = 0; i < v.size(); ++i)
        {
            ave += v[i] / v.size();
        }
        return ave;
    }
};
```

Note the directive is now `#pragma omp parallel for`. Additionally we have set the loop to run on 4 threads. Now, let's move to our cpp file:

```
#include <iostream>
#include <vector>
#include <numeric>
#include <algorithm>
#include "parallel_test.h"

int main()
{
    const int n = 1000000;
    std::vector<double> v(n);
    int i = 0;
    std::generate(v.begin(), v.end(), [&i]
                  { return ++i; });

    // create objects
    SumVector sum_serial(v);
    SumVector sum_parallel(v);
    Timer timer_serial;
    Timer timer_parallel;

    // Calculations
    // serial
    timer_serial.start();
    double result_serial = sum_serial.serial();
    timer_serial.stop();
    // parallel
    timer_parallel.start();
    double result_parallel = sum_parallel.parallel();
    timer_parallel.stop();

    // Outputs
    // serial
    std::cout << "Result serial: " << result_serial << std::endl;
    std::cout << "Time: " << timer_serial.elapsedMilliseconds() << "ms" << std::endl;
    std::cout << "Time: " << timer_serial.elapsedSeconds() << "s" << std::endl;
    // parallel
    std::cout << "Result parallel: " << result_parallel << std::endl;
    std::cout << "Time: " << timer_parallel.elapsedMilliseconds() << "ms" << std::endl;
    std::cout << "Time: " << timer_parallel.elapsedSeconds() << "s" << std::endl;

    return 0;
}
```

If you compile and run this code you will notice a particular output. The result from the parallel calculation is wrong (smaller that the expected value), and there's no significant reduction in time (it is probably slower than serial). This is because each of the 4 threads defined are carrying out the task independently, without considering

the updates to `ave` from other threads and finishing at different times, leading to overwriting the contribution of another thread to the final value. Additionally, since threads are not allowed to write a value to a variable at the same time, each thread has to until another thread finishes before writing the value to a variable, resulting in an increased time. This behaviour is known as *data race*. There are different ways for handling a data race, depending on the type of operations involved in the task. In this particular case we can use the `reduction` clause for the *`#pragma omp parallel for`* directive, hence, we just need to change a line in our header file:

```
#pragma omp parallel for num_threads(4) reduction(+ : ave)
```

After this, the parallel calculation should output the correct result with a significant speed–up.
You can find more information about OpenMP directives and clauses in the OpenMP cheat sheet.

3 Conclusion and Further Reading

Parallel programming in C++ can significantly enhance the performance of your applications, particularly on multi-core processors. However, it requires careful management of resources to avoid common pitfalls such as race conditions and deadlocks.

For further reading, consider the following resources:

- Anthony Williams, *C++ Concurrency in Action.*
- Herb Sutter, *Effective Concurrency.*
- Bjarne Stroustrup, *The C++ Programming Language* (C++11 and beyond).