Documentation Assignment 3

Marco Busato, 852074
Martina Donadi, 865737
Fabio Dainese, 857661

July 04, 2020

Contents

1 Introduction

1.1 Assignment Description

In this third part of the project we have to parallelize operations on tensors (additions/subtractions and contractions) with threads.
In particular we have to make sure that any deferred expression launches a given number of worker threads and distribute the operations over those threads. Each thread computes the operations for a given range of non-repeated indices, writing the result on the target tensor.
In other words, we have been asked to implement the current version of the library into a multi-thread one using the standard thread class provided by C++ library.
For this part of the project, we decided to start from the professor's implementation of the second part of the project.

1.2 Concurrency in C++

The C++ Standard Library support for concurrency includes:

- Memory Model: a set of guarantees for concurrent access to memory
- Programming without locks: low-level mechanisms to avoid data race conditions
- Thread Library: support for traditional threading-and-locks-style concurrent programming at system level
- Task Library: facilities to support concurrent programming at task level

1.2.1 Memory Model

The C++ memory model assure that two threads of execution can update and access separate memory locations without interfering with each other. A **memory location** can be

- An object of scalar type (arithmetic type, pointer type, enumeration type, null pointer)
- The largest contiguous sequence of bit fields of non-zero length

Cache memory can be shared among several threads on the same or different processing units. This area is evolving rapidly. Besides it may takes long to get the value of a word from memory into a cache and then into a register. **Memory Ordering** is the term used to describe what a programmer can assume about what a thread sees looking up a value from memory. When a thread reads a value from a memory location, it may see the initial value, the value written in the same thread, or the value written in another thread. The simplest memory order is the **sequentially consistent** memory order in which every threads sees the effects of every operation done in the same order. In this sense operations are **atomic**: they force a consistent view of the memory. There are many possible sequentially consistent orders for a given set of threads and they can impose significant

synchronization costs on some machines. These costs can be eliminated relaxing the rules while avoiding data races.

1.2.2 Programming without Locks

Lock-free programming techniques allow for writing concurrent programs without using explicit locks. This is guaranteed by means of primitive atomic operations used in the higher-level implementation of concurrency mechanisms. For each atomic operation a thread will eventually make progress, even if other threads compete for access to an atomic object. Atomic types and operations usually rely on assembly code or system-specific primitives. Synchronization operations determine when a thread sees the effect of another thread, i.e. the memory order. By default the memory order is set to memory_order_seq_cst (sequentially consistent memory order). The standard memory orders are:

- memory_order_relaxed: no operation orders memory
- memory_order_consume: a load operation performs a consume operation on the affected memory location
- memory_order_acquire: a load operation performs an acquire operation on the affected memory location
- memory_order_release: a store operation performs a release operation on the affected memory location
- memory_order_acq_rel: a load operation performs an acquire operation on the affected memory location and a store operation performs a release operation on the affected memory location
- memory_order_seq_cst: a load operation performs an acquire operation on the affected memory location and a store operation performs a release operation on the affected memory location

It is architecture-specific whether a memory order make sense.

1.2.3 Threads and Tasks Libraries in C++

We call a **task** an activity potentially executed concurrently by a thread. Threads may share an address space with other threads. In The C++ Standard library threads are intended to map one-to-one with the operating system's threads. To avoid data races a few concepts need to be considered:

- stacks are not shared between threads, unless a pointer to a local variable is passed to another thread
- in lambdas by-reference context is binding

In particular stack memory sharing need to be done very carefully.
Thus in C++ Standard Library a thread represent a system resource: a system thread, possibly with dedicated hardware. A thread can be moved but not copied. After it is moved it can not be joined

anymore.

The std::thread class (see [4]) is introduced in version 11th of C++ and offers the following facilities:

Member types

Member type	Definition
`native_handle_type` (optional)	*implementation-defined*

Member classes

`id`	represents the *id* of a thread (public member class)

Member functions

(constructor)	constructs new thread object (public member function)
(destructor)	destructs the thread object, underlying thread must be joined or detached (public member function)
`operator=`	moves the thread object (public member function)
Observers	
`joinable`	checks whether the thread is joinable, i.e. potentially running in parallel context (public member function)
`get_id`	returns the *id* of the thread (public member function)
`native_handle`	returns the underlying implementation-defined thread handle (public member function)
`hardware_concurrency` [static]	returns the number of concurrent threads supported by the implementation (public static member function)
Operations	
`join`	waits for a thread to finish its execution (public member function)
`detach`	permits the thread to execute independently from the thread handle (public member function)
`swap`	swaps two thread objects (public member function)

Non-member functions

`std::swap``(std::thread)` (C++11)	specializes the `std::swap` algorithm (function)

Figure 1: std::thread

2 Implementation

2.1 Jobs allocation to workers

In order to make concurrent operations we first define the function *set_workers_number* to allow users for threads initialization.
The number of workers is set by default equal to 1, if the user does not explicitly set it when the function is called. On the other hand the user can allocate threads as much as he wants, so the function *hardware_concurrency* tell us how many workers the system can reserve for the user and if the number of workers specified by the user exceed the number given by the system, the latter is set.

```
int NW = 1;

void set_workers_number(size_t num_workers = 1){
    unsigned int w_pool_size = std::thread::hardware_concurrency();
    if(w_pool_size <= 0 or num_workers <= 0){
        // std::thread::hardware_concurrency() may return 0 if the number of current
        //threads supported is not well defined or not computable
        NW = 1;
        std::cout << "Number of threads currently supported not computable: reduced to
            1" << std::endl;
    }else{
        NW = num_workers <= w_pool_size ? num_workers : w_pool_size;
        std::cout << "Supported threads set to: " << NW << std::endl;
    }
}
```

The other significant modification of the code concerns how the workers previously declared share the works to be done.
If the number of workers is set to 1, i.e. no concurrency operations, the code is not modified, and proceed as usual.
Otherwise, the system equally split the jobs among the threads, if there are some extra jobs, they will be distributed among the workers.

```
std::vector<int> jobs = std::vector<int>(NW, counter / NW); //jobs vector
//holds the queue of jobs for each worker thread
size_t extrajob = counter // NW; if the modulo division is not 0 it means that some
//workers will have more jobs in their queue
while(extrajob > 0){
// distribute the remaining jobs to workers: for example if 5 jobs are left
//add them to the queue of workers 4, 3, 2, 1, 0 respectively
    ++jobs[extrajob - 1];
    --extrajob;
```

```
}
```

Once the workers are set up, they are ready to perform the assigned jobs.
Notice that we only pass the tensor index in order to avoid threads' race condition. The new *eval* functions will compute the right position.

```
threads.at(i) = std::thread([this, &x](int nj, std::vector<size_t> idxs) {
for (int k = 0; k < nj; ++k) {
    eval(idxs) += x.eval(idxs);
    unsigned index = idxs.size() - 1;
    ++idxs[index];
    // iterate through tensor positions to accomplish current job
    while (idxs[index] == widths[index] && index > 0) {
        idxs[index] = 0;
        --index;
        ++idxs[index];
        }
    }
}, jobs[i], job_idxs[i]);
```

Finally the threads are catched up by the *join* function.

For a more accurate explanation we highly suggest to have a look at the comments present in the submitted code.

In the next section we run some test in order to evaluate the performance of parallelization on tensors operations.

3 Performance Analysis

3.1 Hardware Specification

The tests were performed on iMac computer with the following specifications:

```
Model Name: iMac
Model Identifier: iMac17,1
Processor Name: Quad-Core Intel Core i7
Processor Speed: 4 GHz
Number of Processors: 1
Total Number of Cores: 4
L2 Cache (per Core): 256 KB
L3 Cache: 8 MB
Hyper-Threading Technology: Enabled
Memory:  32 GB
```

An interesting future experiment is to run these same tests in a more performing machine, i.e. full-feature server, and compare the results. For more information about further improvements, we suggest to have a look at the 'Section 4'.

3.2 Tensor Sum/Difference Test

Performing many different sum operations on tensors we have noticed that:

- The sum operation in tensors of small size has better performance when executed sequentially
- The sum operation in tensors of quite big size has slightly better performance when executed in parallel

Function name	Number of workers							
	1	2	3	4	5	6	7	8
test_sum	0.057357 ms	0.162773 ms	0.143418 ms.	0.20506 ms	0.148702 ms	0.154027 ms	0.167154 ms	0.207192 ms
test_long_sum	243.516 ms	243.047 ms	243.828 ms	244.078 ms	245.7 ms	243.282 ms	244.701 ms	242.749 ms
test_long_sum (big size tensors)	295.324 ms	277.598 ms	248.79 ms	248.435 ms	247.968 ms	249.379 ms	250.221 ms	247.05 ms

Table 1: Test cases on sum/difference of tensors

3.3 Tensor Multiplication Test

Performing many different multiplication operations on tensors we have noticed that:

- The multiplication operation in tensors of small size has better performance when executed sequentially

- The multiplication operation in tensors of quite big size has also better performance when executed sequentially

Function name	Number of workers							
	1	2	3	4	5	6	7	8
test_multiplication	0.050183 ms	0.139427 ms	0.193015 ms	0.165647 ms	0.171893 ms	0.178827 ms	0.179004 ms	0.194316 ms
test_long_multiplication	0.344243 ms	0.480836 ms	0.473737 ms	0.519016 ms	0.472298 ms	0.478069 ms	0.520823 ms	0.489963 ms
test_long_data_multiplication	21597.9 ms	206289 ms	145154 ms	122411 ms	114324 ms	111278 ms	104238 ms	103298 ms

Table 2: Test cases on multiplication of tensors

4 Final considerations

As we can see from the tests results, multi-threading may worsen the performance. There can be various factors that can make multi-threading slowdown our application, for example:

- A computer with N cores can do computations at most N times as fast as a computer with 1 core. This means that when we have T threads where T > N, the computational performance will be capped at N.
- A computer can only perform a certain number of read/write operations per second on main memory. If we have an application where the demand exceeds what the memory subsystem can achieve, it will stall (for a few nanoseconds). If there are many cores executing many threads at the same time, then it is the aggregate demand that matters.
- A typical multi-threaded application working on shared variables or data structures will either use volatile or explicit synchronization to do this. Both of these increase the demand on the memory system.
- **Thread creation is expensive**. If the task that you perform with the thread is too small, the setup costs can outweigh the possible speedup.

The last point well summarize our case, there can be an high waste of time when a worker is created compared to the actual time to perform the operations.
Like described in the 'Table 1' and 'Table 2' any time that an operation was performed on a small tensor there was not any significant improvement (sometimes also worsening), meaning that it was better the serialization version rather than the parallel one.
On the other side, performing the same operation on long tensors it lead to a slightly improvement overall, e.g. on the sum/difference operation (see 'Table 1').

A possible solution, leave it to the user as a future improvement, is to consider the implementation of the above task using some more sophisticated library (e.g. Boost) or at least creating a more thread-friendly code environment (e.g. pull-thread creation).
Another further approach is to switch to a *GPU* based environment in order to gain even more computational power. In this regard it will be highly recommended an implementation using *CUDA* [5], *OpenCL* [6] or Metal [2] library.

As a final suggestion we want to propose the reading of the *"Comparison of threading programming models [7]"* paper in which is well analyzed the comparison of language features and runtime systems of commonly used threading parallel programming models.

References

[1] Torsello Andrea. Slide of the course (advanced algorithms and programming methods).

[2] Apple. Metal shading language specification (v2.3). https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf.

[3] C++ Community. Class thread c++. http://www.cplusplus.com/reference/thread/thread/.

[4] Cpp Ref Community. Thread in c++. https://en.cppreference.com/w/cpp/thread/thread.

[5] NVIDIA Corporation Cyril Zeller. Cuda c/c++ basics. https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf.

[6] OpenCL Community (Khronos Group). The opencl specification. https://www.khronos.org/registry/OpenCL/specs/2.2/html/OpenCL_API.html.

[7] S. Salehian, Jiawen Liu, and Yonghong Yan. Comparison of threading programming models. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 766–774, 2017.