Tensor Library - Concurrency and Parallelism

The goal of the assignment is to perform **parallelize operations** on tensors (additions/subtractions and contractions).
So, expressions launch a given number of worker threads and operations are distributed over those threads. Each thread computes the operations for a given range of non-repeated indices, writing the result on the destination tensor.

To the previous Einstein tensor library, classes `job`, `worker_thread` and `scheduler` were added.

Out of the above classes, there are two constant values:
- `const unsigned int NUM_THREAD = 5;` which indicates the maximum number of thread that the program creates and runs
- `const unsigned int NUM_IDX_PER_JOB = 5;` which indicates the maximum number of cells (of a same index) that a single job can manage (and compute)

JOB

A job could works over one or more indexes in such a way to avoid lack of memory and computing resources.
Job class is parameterized on `template<typename T>` and it has a default constructor and a constructor which takes in input a `std::vector<size_t>& indexes` and an `unsigned int max_pos` which are respectively the vector containing the initial position of the indexes that a job can manage (and compute) and the maximum number of cells that a job can manage (and compute).
The computation starts from the initial position of an index and it will continue to the next position of the same index until `NUM_IDX_PER_JOB` is reached.

The job list is created when `operator =` in the `einstein_expression` class is invoked, and indexes of this class are divided among the jobs.
Each job is executed by a worker, defined inside the `worker_thread` class.

WORKER THREAD

The class `worker_thread` is parameterized on `template <typename T1, class C1, typename T2, class C2>` and it contains the definition of the thread work.
Inside this class, since each thread perform the same task on different indexes, there is a static function, `static void my_thread()`, which takes several inputs:

- `std::vector<job<T1>>& job_list` : a list containing all the jobs
    - `einstein_expression<T1,dynamic>& dest` : the destination tensor expression
    - `einstein_expression<T2,dynamic,C2> source` : the source tensor expression
    - `std::mutex& jobs_mutex` : a mutex to access to the job list
    - `std::map<unsigned int,std::unique_ptr<std::mutex>>& dest_mutex` : a mutex map to access to a single cell of the target tensor

SInce there are some variables that will be accessed and modified by different thread, we need to prevent race

condition by using a locking mechanism over them: the two mutex written before.
One mutex is used to access to the job list and to get a job at time. The other one is used to write the result of einstein expression in a cell of the target tensor (it was done by maintaining a map that associates a cell to the respective mutex).

JOB SCHEDULER

The `scheduler` class is parameterized on `template<typename T1, class C1, typename T2, class C2>` and has a constructor used to intialize its internal members:

- `std::vector<job<T1>>& job_list;`
- `std::array<std::thread,NUM_THREAD> thread_list;`
- `einstein_expression<T1,dynamic>& dest;`
- `einstein_expression<T2,dynamic,C2> source;`
- `std::mutex jobs_mutex;`
- `std::map<unsigned int, std::unique_ptr<std::mutex>>& dest_mutex;`

that are the same explained in the Worker Thread section except the `thread_list` which is a list of thread in execution (its maximum size is specified by the constant `NUM_THREADS`).
Inside this class, there a method `void run()` used to create and run the threads (`std::thread`) and wait for their termination (`join()`).
Each thread is an execution of the static method contained in the `worker_thread` class.

Moreover, was modified the `eval()` method in each `einstein_expression` class in such a way that an autonomous thread could directly access to the tensor by providing it an index.
So, after indexes setup, the job list is created and it is passed to the `scheduler` class with the required parameters.
Then, `scheduler.run()` is performed.

RESULTS

I tried to execute different times this program (on the same expression) by changing the two constant values (`NUM_THREAD` and `NUM_IDX_PER_JOB`) and I computed the elapsed time for each computation.
Here the results:

NUM_THREAD	NUM_IDX_PER_JOB	Time
1	1	0,0028 sec
1	5	0,0017 sec
3	1	0,0030 sec
3	5	0,0012 sec
5	1	0,0032 sec

NUM_THREAD	NUM_IDX_PER_JOB	Time
5	5	0,0011 sec
10	1	0,0034 sec
10	5	0,0014 sec

We can think that increasing the number of threads we could get an important improvement of the performance.
But not always there is a very good improvement. In fact, also the number of cells per job is an important factor so we need to balance both the values in order to get the best performance on computing a tensor einstein expression, always keep in mind the dimensions of the source tensors that compose the expression.

COMPILING

Compile with `g++ einstein.cc -o einstein -std=c++17` .
Then execute `./einstein`