Advanced algorithms and programming methods - 2 [CM0470]

Third assignment's report: Einstein's Concurrency

Group members:

- Bernardi Riccardo 864018
- Buoso Tommaso 864055
- Benetollo Lorenzo 864882

Table of contents

1 Introduction and general structure

This report describes the solution we ha have found of the third assignment about the parallelisation of addition, subtraction and contraction on tensors. We started from the solution provided by the professor and we noticed that the library was implemented to build a parsing tree of the Einstein expression given as input. The base class is

```
template<typename T> class einstein_expression<T,dynamic,einstein_proxy>
```

and it takes as a template parameter an expression proxy, that can be another einstein expression in order to perform the basic operations described above, which are the following :

```
template<typename T, class E1, class E2> class
einstein_expression<T,dynamic,einstein_multiplication<einstein_expression<T,dy
namic,E1>,einstein_expression<T,dynamic,E2>>>
template<typename T, class E1, class E2> class
einstein_expression<T,dynamic,einstein_binary<einstein_expression<T,dynamic,E1
>,einstein_expression<T,dynamic,E2>>>
template<typename T, class E1, class E2> class
einstein_expression<T,dynamic,einstein_addition<einstein_expression<T,dynamic,
E1>,einstein_expression<T,dynamic,E2>>>
template<typename T, class E1, class E2> class
einstein_expression<T,dynamic,einstein_subtraction<einstein_expression<T,dynam
ic,E1>,einstein_expression<T,dynamic,E2>>>
```

We were asked to transform the current version of the library into a multi-threaded one and avoiding to modify the multiplication operation. After that we were asked to parallelize only on non-repeated indexes of the tensors. We solved the problem by the mean of standard threads provided by the c++ standard library.

2 The move assignment operator

2.1 Before

Before our modification the move operator performed sequential operations as we can see below:

```
while(!end()) {
    eval() += x.eval();
    next();
    x.next();
}
```

As we can see in the code above, operations are performed iteratively calling the functions eval and next, which respectively write in the new expression and update the current pointer to data. The **eval** function actually access data and is used in all of the classes involved in the operations, but slightly differs between them, depending one the mathematical operation to be performed. The **next** function updates the pointer to data that will be accessed by eval function by increasing the values of the current pointer and the index of the proxy tensor (with the respect of the bounds). These are the core operations that we want to perform in parallel, so we need to coordinate multiple access to the pointer in order to resolve race condition and avoid access to the same location.

2.2 After

Before start executing the expressions, we permit to the user to choose the number of thread that he want to use for parallelize the different operations. The function check if the number of thread given in input is at least equal to 1, otherwise set it to 1, and less then the maximum number of threads that the machine can support, otherwise set it to the max number of threads.

```
void set_thread(size_t n_threads = 1){
    unsigned int max_n = std::thread::hardware_concurrency();

    if(max_n == 0 or n_threads == 0){
        std::cout << "Number of threads set to 1: " << max_n << std::endl;
        N = 1;
    }
    else if(n_threads <= max_n){
        N = n_threads;
    }
    else{
        std::cout << "Number of threads reduced to the maximum supported by
the machine: " << max_n << std::endl;
        N = max_n;
    }
}
```

We added information to the fields of the tensor to perform the parallelization such as a matrix that contains N_threads rows and a column for every index that is present in the output tensor.

Our idea is to create multiple threads, each of them performing the eval function over a specific location. For this purpose we overload the eval function, specifically designed for thread evaluation, which accepts an index as a parameter and calculates the correct location at which the thread needs to access.

```
T& eval(std::vector<size_t> indxs) const {
    auto ptr = start_ptr;

    for(int i = indxs.size() - 1; i >= 0; --i){
        ptr += indxs[i] * strides[i];
    }

    return *ptr;
}
```

After calling this new eval, index is incremented by one with the respect of the bounds.

From our previous experience we know that to parallelise matrix operations you have to split the matrix in N_THREADS parts that are independent and only then launching threads on them. This part of the code in the move was the part of the code that seems to necessitate this kind of improvement since inside it is performed a sum between the resulting tensor and the right-expression of the move operator.

The move assignment operator was also modified but we are going only to describe how it works from an abstract point of view, to understand more about it we advise to check the code attached.

The move checks if the N_THREADS is set to 1 and in this case the code that is run is the one provided by the initial library, instead if the number is larger then 1, the code behave to parallelize the operations.

At first we calculate the vector span, that contains at position i the number of cell that the i-th thread should elaborate.

```
size_t counter = 1;
for(auto w = widths.begin(); w != widths.end(); ++w){
    counter *= (*w);
}
std::vector<int> span =  std::vector<int>(N, counter / N); //number of jobs
for each thread

size_t tmp = counter % N;

while(tmp > 0){          //if counter % N > 0 some threads will has more jobs
then others
    ++span[tmp - 1];
    --tmp;
}
```

After that it is computed the index that should be accessed by the tensors(both this and x). The vector thread_indxs contains at position i the index at which the i-th thread start to compute.

```
int tpos = 0, old = 0; //using old position to maintain tpos value, since tpos
is modified in next loop
for(int i = 0; i < N; ++i){
    thread_indxs.at(i) = std::vector<size_t>(widths.size());
    old = tpos;


    for(int j = widths.size()-1; j >= 0; --j){
        thread_indxs.at(i).at(j) = tpos % widths.at(j);
        tpos = tpos / widths.at(j);
    }
    tpos = old + span[i];
}
```

This last fact is peculiar and it is because we cannot store and pass a vector of pointer, because x could be composed by several subexpression with several pointers, for this reason we have to pass the tensor index, then the new **eval** of every einstein expression will compute the right position using the own's strides, it means that doing this way we are guaranteed that every tensor computes the right position without generating a race condition.

After this preamble comes the core of our implementation that launches a lambda that as parameters take the span(number of cells to be computed) and the starting index assigned to that precise thread, furthermore we pass in the closure also the **this** and the **x**. Every threads cycles **span** times to compute all assigned cells calling the **eval** on the **this** and on the **x** and moving the current index of one position forward. After incrementing the current index it is checked that it respects the bounds imposed by the **widths** of the tensor.

```
threads.emplace_back(([this, &x](int span, std::vector<size_t> indxs){
    for(int k = 0; k< span; ++k) {
        eval(indxs) += x.eval(indxs);

        unsigned index = indxs.size()-1;
        ++indxs[index];

        while(indxs[index] == widths[index] && index>0) {
            indxs[index] = 0;
            --index;
            ++indxs[index];
        }
    }
}), span[i], thread_indxs[i]);
```

At the end all the threads are caught by a **join**.

2.2.1 Eval in different expressions

We could have different cases of expression on which the eval can be computed, in every case, like the basic case, the function take as input the an index vector and return a T object.

In the case of multiplication we execute the eval on each factor and then return their product.

```
T eval(std::vector<size_t> indxs) {
    return exp1.eval(indxs) * exp2.eval(indxs);
}
```

In the case of addition or subtraction we execute the eval on each element and then return their sum or difference.

```
T eval(std::vector<size_t> indxs) {
    return exp1.eval(indxs) + exp2.eval(indxs);
}
```

```
T eval(std::vector<size_t> indxs) {
    return exp1.eval(indxs) - exp2.eval(indxs);
}
```

3 Better solutions - Pointer Wrapper

Since we really wanted to improve the performances we approached the problem in a different way. The main problem up to us was that every time the new eval was called it has to start from the start_ptr, to avoid race conditions, of the expression and iterate to the pointer that correspond to the index given as input. We thought that passing a more complex object to the **eval** function that embedded the pointer and increments it without generating race conditions.

For this purpose we create an object called **PtrWrapper** (friend class of einstein_expression) that maintain the index of the cell to point, the pointer in case the expression is contain only by one proxy tensors (in our case the right element of the move operator), and two shared pointer child1 and child2 that point to two other PtrWrapper in case the expression is composed by two sub-expressions (for example if we have a multiplication).

```
//wrapper class that contain the current index and current value of an
expression
template<typename T> class PtrWrapper{
  public:

    PtrWrapper(const PtrWrapper& p) = default;

    PtrWrapper(T* ptr, std::vector<size_t>& idx) : ptr(ptr), idx(idx),
child1(nullptr), child2(nullptr){}

    PtrWrapper(std::vector<size_t>& idx) : ptr(nullptr), idx(idx),
child1(nullptr), child2(nullptr){}

    PtrWrapper(PtrWrapper* p1, PtrWrapper* p2) : child1(p1), child2(p2),
ptr(nullptr){}

    template<typename T2, class IDX2, class type2> friend class
einstein_expression;

  private:
    std::vector<size_t> idx;

    T* ptr;
    std::shared_ptr<PtrWrapper> child1;
    std::shared_ptr<PtrWrapper> child2;
};
```

Thanks to this new pointer wrapper we can simplify the overloaded eval function that now takes as input a PtrWrapper and return the value pointed by its ptr, this remove the overhead introduced by the iterations of the previous version, and we also implement an overloaded **next()** function that takes as input a PtrWrapper and increment its ptr and index fields, with the respect of the bounds; so we build a new lambda expression with this new functions.

```
threads.emplace_back(([this, &x](int span, std::vector<size_t> indxs){
```

```
    auto p1 = PtrWrapper<T>(indxs);
    auto p2 = PtrWrapper<T>(indxs);

    setPtr(p1);
    x.setPtr(p2);

    for(int k = 0; k< span; k++) {
        eval(p1) += x.eval(p2);

        next(p1);
        x.next(p2);
    }
}), span[i], thread_indxs[i]);
```

The function **setPtr()** intialize the pointer of PtrWrapper to the position given by the index takes as input by the constructor. This function, as in the previous implementation, prevents race conditions because starts from a copy of start_ptr and increment it since obtaining the correct value.

```
void setPtr(PtrWrapper<T>& p) const{
    p.ptr = start_ptr;

    for(int i = strides.size() - 1; i >= 0; --i){
        p.ptr += (p.idx)[i] * strides[i];
    }
}
```

The solution obtained by using PtrWrapper has better performances then the previous since the eval function directly return the data pointed by the pointer contained in the PtrWrapper object without calculating it each time the function is called, and the increment of this pointer doesn't introduce overhead because it's done in parallel with the increment of the index.

3.1 PtrWrapper in different expressions

When the expression is composed by two sub-expressions (multiplication, addition, subtraction), the new eval takes as input a PtrWrapper and compute the result of the operation between the eval() of the two child of the PtrWrapper given as input. In the same way, the new next() function compute two next() that take as input the two child of the PtrWrapper given as input.

The main difference shows up in the setPtr() function in which we assign to child1 and child2 two new shared pointer to new PtrWrapper object, and then we call the setPtr() on the two child for the respective subexpression.

```
void setPtr(PtrWrapper<T>& p) const{
    p.child1 = std::make_shared<PtrWrapper<T>>( PtrWrapper<T>( p.idx ) );
    p.child2 = std::make_shared<PtrWrapper<T>>( PtrWrapper<T>( p.idx ) );

    exp1.setPtr( *(p.child1) );
    exp2.setPtr( *(p.child2) );
}
```

With this implementation we build a tree of PtrWrappers with their children that has at its leafs the objects that contain the pointer to the data of the tensor. So each thread builds its own tree of PtrWrapper with the pointers to the data matching the index on which the thread is working.

4 Performances

We have measured all the performances at every test and we have noticed that only a small improvement was introduced by the mean of the multi-threading. So we tried to add some tests with a lower number of rank but many thousands of cells for every index. At this point the improvement becomes visible and very effective.

We also decided to go deeper into the possible operations that could perform better exploiting the threads regardless of the multi-threading overhead. We tried so to produce big tensors of low rank. We ended up discovering that with a rank 3 tensor with dimensions <1000,1000,1000> summed for itself our multi-threaded version can perform more then twice better with respect to the sequential version.

An example of the tests we tried is here below:

```
void test_great_matrix_low_rank_n_thread(){
    set_thread(threads);
    tensor<int,rank<3>> t1(1000,1000,1000);

    for(auto iter=t1.begin(); iter!=t1.end(); ++iter)
        *iter = 1;

    auto i=new_index;
    auto j=new_index;

    a.tic();
    tensor<int> t4 = t1(i,i,j) + t1(i,i,j);
    a.toc();
}
```

Some remarkable measure that we gained are summarized here below:

Test	Operation	Sequential	Concurrent
test_great_matrix_low_rank, 1000x1000x1000	addition	52 ms : 52353 µs	19 ms : 19420 µs
test_big_contaction, 1000x1000x1000	contaction	139 ms : 139763 µs	15 ms : 15260 µs
test_big_multiplication, 4000x4000	mutiplication	52 sec : 52813747 µs	25 sec : 25164497 µs

Many other tests were performed but the most significant were presented.

5 Conclusion and further development

In conclusions we can say that the initial sequential version of the library is already pretty optimized in terms of performance, so that for operations between small tensors we cannot see so much difference, we can get a considerable improvement only if we perform operations between tensors that has great dimensions.

One further development could be using better performing library for multi-threading instead the standard thread (i.e. boost).