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

- description of the library given from the professor
- parsing tree build
- description of the problem
- Tools used, testing library, thread library

This report describes the solution we ha have found of the thrid 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 trasform the current version of the library into a multithreaded one and avoiding to modify the multiplication operation. After that we were asked to parallelise only on non-repeated indexes of the tensors. We solved the problem by the mean of standard threads provided by the c++ standard library. Plus we also used a testing library developed by some students during this course available on github (see the references).

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 funtion by increasing the values of the current pointer and the index of the proxy tensor (with the respect of the bounds). Theese 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.

- explain better why threading is here

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 informations to the fields of the tensor to perform the parallelisation 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 funtion 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 behaive to parallelize the operations.

For fist we caluculate 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 wich 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 Other solutions

Since we really wanted to improve the performances we approached the problem in a different way. The main problem up to us was the fact that every thread was forced to run an enormous number of nexts to arrive at the right point to modify the cell that was assigned to him. This was surely a bottle-neck. We thought that passing a more complex object to the **eval** function that embedded more informations could have lead to a lower number of nexts. this at the end should have lead to better performances regardless to a slightly more overhead due to the more complex object passed.

the solution is called PtrWrapper and the code is here below:

```
//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;
};
```

4 Performances

We have measured all the performances at every test and we have noticed that no improvement was introduced by the mean of the multithreading. So we tried to silence(ie comment) our entire code searching a lower bound for the improvement and we discovered that the initial code is really efficient and that the code we used to parallelise introduced a little overhead but since the code is already efficient our code was equally performing or worst performing. After this simple

experiment we ended up at the idea that the greatest cost of all was the building of the parsing tree.

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, no contractions an no multiplications but only summations. We ended up discovering that with a rank 3 tensor with dimensions <400,400,400> summed for itself our multi-threaded version can gain up to 13 seconds with respect to the sequential version.

```
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;

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

Sequential	Sequential	Concurrent	Concurrent	Improved?	PtrWrapper?
Test 34, 1000x1000x1000	246sec / 246327362 µs	Test 35	240 sec : 240171331 µs	Yes	No
Test 36, 4000x4000	4 sec : 4521288 µs	Test 37	7 sec : 7599215 µs	No	No
Test 24, hundred sums	500ms	Test 25	475ms	Yes	Yes

Many other tests were perfromed but only on few cases there were a perceptible improvement.