Advanced algorithms and programming methods

Year: 2017/2018

Assignment 3

Matrix Library

6th May 2018

Antonio Emanuele Cinà 854866
Federico Marcuzzi 853770
Gianluca Bigaglia 854406

Index

1 Introduction

The goal of this document is to describe strategies adopted to solve this third assignment. In particular, the request of this one is to extend the matrix library, developed until now, so that the operations are performed in parallel when it is possible. The execution of multi operations in parallel has the advantage of improving the performance, but there also some particular issues that will be discussed during the following document.

2 Problem Analysis

In some cases matrix operations can be performed parallelly. For this reason providing an architecture that takes advantage from this could bring to optimal results in terms of performance. Here we present some cases in which operations can be performed in parallel:

- $(A + B) + (C + D)$
- $(A + B) * (C + D)$
- $A * B * C * D$

On the first case we consider the sum between two matrices produced by other sum expressions, the idea of parallelism, in this case, consists in performing the sum $(A + B)$ and $(C + D)$ at the same time since they are independent and it does not change the final result. When the two parallel sums are finished, the main thread has to combine the two results and execute the final sum. The second case is very similar to the first one, but with the difference that at the end the main thread has to perform the matrix product of the two results instead of a sum. The third case is a little bit difficult, since now there is not special order given by the usage of parentheses so we rely on the product optimization realized on the previous assignment. The idea is that $A * B * C * D$ can be optimized by the product optimizer such that the final expression could be for instance $(A * B) * (C * D)$, and in this case we reduce the problem to the second case.

The second requirement for this assignment is to apply another improvement to the matrix product, such that each matrix A is composed of several submatrices A_{ij} and the product $C = A * B$ can be expressed like:

$$C_{ij} = \sum_k A_{ik} * B_{kj}$$

Where $*$ represents the product operator between submatrices. The advantage of this technique is that each component C_{ij} can be computed independently, taking advantage of possible parallelism algorithms. Note that k is an arbitrary constant that define the size of each block of the matrix. So at the end of this assignment matrix product operator should be able to apply multiple product in parallel and each pair of expression can be performed in parallel using the previous formula.

3 Implementation

In this section we are going to discuss about code implementation, class responsibilities, motivations and limits behind different design choices.

3.1 Design

During the design phase of the library we thought about several solutions, but some of them did not satisfy all the requirements of the project or they had strong drawbacks.
For instance our first idea was to encapsulate a future object inside the `matrix` class. In this way the internal structure of a matrix is substituted with the resulting matrix only during the evaluation. The problem of this solution was to define correctly the evaluation time. We decided to make transparent to the user as much as possible technical knowledge of the library, in order to provide a readable and clear syntax and semantics.

```
matrix<int> out = A + B
matrix<int> ev = (A+B).eval()
```

On the first case the expression is by evaluated hiding technical aspects of library implementation. Instead the second one requires the knowledge of certain behaviors inside the library and it is more verbose. But this choice brought to some problems for the previous solution, since it was necessary to define a strategy that converts a `matrix` object to another `matrix` of the same type; but if the original matrix has a future it must be evaluated first. Our idea was to implement this strategy inside the move constructor but doing that we encountered the problem of **copy/move elision**. From *C++11* it was introduced an optimization to C++ compiler that omits copy and move constructors when they are unecessary. For instance, when a temporary object with the same type should be created, it is constructed directly in the location where it would otherwise be copied or moved. In this case if the result of $A+B$ is a matrix object it will be created directly inside the final object, without calling move or copy constructor. At the end it was necessary to apply eval operation to the final matrix. There is the possibility to remove this optimization by setting a specific flag, but the resulting performance was not so good. For this reason we decided to adopt another strategy, based on the separation of complete matrix and future one, discussed on the following subsections.

3.1.1 Parallel sum

In order to improve the performance of sum operator, we decided to adopt a strategy that can take advantage of parallel expressions. Considering for instance the following expression:

$$(A + B) + (C + D)$$

We can see that $A + B$ and $C + D$ can be performed parallelly, and when results are ready, the sum of the two resulting matrices is computed. By doing this, a problem arose: how can we distinguish the previous expression from this one: $A + B$? At compile time we have no idea of the total structure of the expression, since they are evaluated considering possible parentheses and operators from left to right. In this case the first term that the compiler can see is $(A + B)$ which have no difference with respect to $A + B$, so it will execute the classic sum operation without performing any parallel operations. So it is necessary to introduce a temporary object that accumulates or keeps track of the expression seen before and perform operation when it is necessary. For this reason we have introduced the concept of **future_matrix**.

```
template<typename T>
class future_matrix<T, Plain> : public matrix_ref<T, Plain> {
  ...
  matrix_ref<T, Plain> eval() {
      return future.get();
    };

  std::shared_future<matrix<T>> future;
};
```

A future_matrix is like a "promise", it will return the result of an expression when the `eval` method is called. Its role is to encapsulate a future operation and when a caller asks for the evaluation the future matrix waits until the operation is not completely performed and later on returns the final resulting matrix. It is used by operator function that, instead of returning the final result will return a "matrix that will give the final result". Just to provide an example of usage we want to introduce the sum operator between dynamic matrices:

```
operator + (const matrix_ref<T,LType>& left , const matrix_ref<U,RType>& right) {
  ...
  /*Create future */
  typedef matrix<typename op_traits<T, U>::sum_type> matrix_type;
  std::shared_future<matrix_type> ft =
      std::async(std::launch::async,
      sum<T, LType, U, RType, matrix_type>,
      left,right)
  .share();
  /* return future dynamic matrix */
  return future_matrix<typename op_traits<T, U>::sum_type, Plain>(std::move(ft),
    left.get_height(), left.get_width());
}
```

As we can see the sum operator creates the future operation `ft` that will be used by the future matrix in order to synchronize eval operation with function `sum`. The function `sum` performs the sum between two matrices but it is assigned to an asynchronous thread that will deal the operation. When an entity asks for the evaluation of the operation to the future matrix, it can happen that for some reasons (like scheduling or complexity of the operation) the asynchronous thread has not completed the operation. In this case is essential for the future matrix to synchronize its operations with the future object returned by the asynchronous thread and wait until the operation is not completed.
Very similar is the behavior of this sum operator, but now it is applied to a standard dynamic matrix and a future one:

```
operator+(const matrix_ref<T, LType> &left, future_matrix<U, Plain> &&right) {
  auto sum=[](const matrix_ref<T,LType>&left, future_matrix<U,Plain> &&right){
    matrix_ref<T,Plain> right_mat = right.eval();
    return sum<T,LType,U,Plain,matrix_type>(left, right_mat);
  };
  std::shared_future<matrix_type> ft =
  std::async(std::launch::async, sum, left, std::move(right)).share();

  return future_matrix<typename op_traits<T, U>::sum_type, Plain>(std::move(ft),
    left.get_height(), left.get_width());
}
```

We can see that more or less the two sum operators have the same structure, but in this case, since the right matrix is a `future_matrix`, it is necessary to request its evaluation for performing the operation. The following example can explain better what is happening on this function:

$$A + B + C$$

Considering the above expression, in a first moment the evaluation of the sum between A and B is performed. During the evaluation, a brand new `future_matrix` which returns the result of $A+B$ is created. We define as t_1 the thread responsible for the sum between A and B. The expression $future\_matrix(A, B) + C$ is immediately evaluated, now the new thread t_2 must synchronize its operation with the thread t_1, since before applying the final sum, t_2 requires the result of the previous sum. This synchronization is performed using the eval of the `future_matrix`, that will block the execution of t_2 until t_1 has completed the operation.

Methods discussed until now are defined and used for dynamic matrices but we know that the library allows the usage of static matrices, which provide information about their dimension also at compile time. For this reason there are two version of `future_matrix`:

```
class future_matrix<T, Plain>: public matrix_ref<T, Plain>{ ... }

class future_matrix<T, Sized<h, w>>: public matrix_ref<T, Sized<h, w>>{ ... }
```

The first one is used for dynamic operations and the second one for static operations. This allows also to maintain, when it is possible, static information inside the `future_matrix`. In any case the two types of `future_matrix` are converted to dynamic or static complete matrices when they are assigned to an object of `matrix` type. When the assignment is performed, the `future_matrix` is evaluated and after that, data is moved from the result matrix to the destination object.

```
matrix(future_matrix<T, Plain> &&X) {
  matrix<T> result = X.eval();
  height = result.get_height();
  width = result.get_width();
  data = std::move(result.data);
}
```

A good question could be: Why don't use a `matrix_wrap`? It can be reasonable to use a `matrix_wrap` and defining a unique `future_matrix` that uses it. But our choice was to avoid the usage of `matrix_wrap` since we know that it introduces some overhead that can affect drastically the performance of sum operator.

We decided to define `future_matrix` as a subclass of `matrix_ref`. The reason is strictly related to the possibility of the library to be extended to include other more complex operations like:

$$(A + B)^T$$

For the moment the result of $A + B$ must be stored or evaluated explicitly and only after this, the transpose operator can be used. So, this expression can be computed with this code: $(A+B).eval().transpose()$, but it is necessary to give knowledge about the library to the user. It can be interesting to extend primitive operations/transformations of matrices hiding this technical knowledge to users. For instance, the `transpose` method of a `future_matrix` has to evaluate first the future operation, after that, the transpose operation can be applied to the result.

The general behavior can be summarized with the following schema:

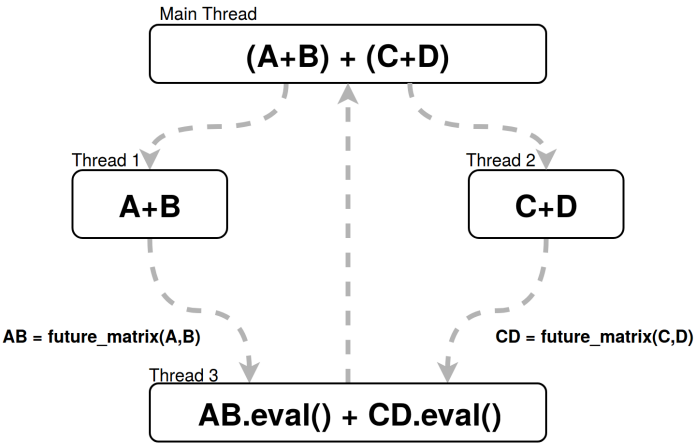

Figure 1: Parallel sum schema

3.1.2 Parallel product

The second operation that can be performed using parallel execution is the multiplication between matrices. As we anticipated before there are essentially three cases that can be summarized with this two examples:

- $(A + B) * (C + D)$
- $A * B * C * D$

The first case is an extension of the parallel sum and it is solved simply defining this function:

```
operator*(future_matrix<T, LType> &&left , future_matrix<T, RType> &&right) {
  ...
  return left.eval() * right.eval();
}
```

So it is necessary to apply the standard product to the two evaluated matrices. The product is not performed until the expression on the left and on the right are completed.
The general behavior of parallel sum is so summarized in the following image:

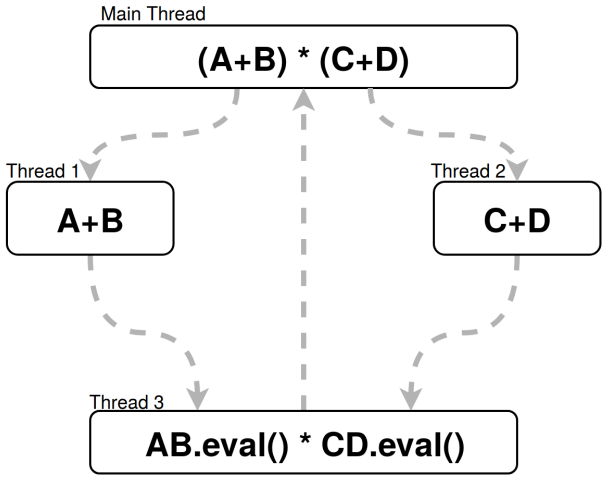

Figure 2: Parallel sum

For the second case we know from the previous assignment that this sequence of products can be optimized considering the size of the matrices. Optimization reduces the number of iteration necessary to compute the result. Another characteristic of product operation is that it is performed using `matrix_wrap` and for this reason it is not possible to use the structures defined before with `future_matrix`, since it is not possible to get directly the right type of matrix wrapped in a `matrix_wrap`. For this reason we decided to take advantage of the structure defined from the `matrix_wrap` and define a new implementation `pimpl` that follows the same idea of a `future_matrix`. In order to do that we have defined a new virtual method *eval* inside the `matrix_wrap_impl` interface. The role of *eval* method is to return an implementation to a physical and evaluated matrix. All the standard implementations return the current implementation, instead the new future implementation uses the evaluation of the future operation and returns a pointer to the new implementation.

```
class concrete_matrix_wrap_impl<T, Future> : public matrix_wrap_impl<T> {
  ...
  std::unique_ptr<matrix_wrap_impl<T>> eval() const override {
    matrix<T> mat = future.get();
    return std::make_unique<concrete_matrix_wrap_impl<T, Plain>>(future.get());
  }
private:
  std::shared_future<matrix<T>> future;
};
```

The new *pimpl* allows now the `matrix_wrap` to wrap not physical/complete matrices but also future matrices, produced by product operation.

```
matrix<T> do_multiply(matrix_wrap<T> lhs, matrix_wrap<T> rhs) {
  assert(lhs.get_width() == rhs.get_height());

  lhs.eval();
  rhs.eval();
  ...
}
```

The function `do_multiply` perform first the evaluation of the two matrices given in input. This allows to decide correctly the right implementation. If at least one of the two wraps a future, its pimpl will be substituted with the result, meaning that `do_multiply` is synchronized and has to wait for the evaluation. For example, imagine that the *lhs* matrix wraps a future and the *rhs* matrix is complete. Assume also that the thread responsible for the operation of *lhs* is identified by t_1. During the evaluation of *lhs*, the current thread has to wait that t_1 completes the operation. When the result is ready the new *pimpl* is created and now *lhs* is a complete matrix. Evaluation of *rhs* returns the *pimpl* to itself. This structure allows also to maintain the same code developed until now and include only the parallelism with the usage of threads. The thread development is added to the method `resolve_one`.

```
void resolve_one() {
  typename std::list<matrix_wrap<T>>::iterator lhs = find_max();
  typename std::list<matrix_wrap<T>>::iterator rhs = lhs;
  ++rhs;

  auto run_multiply = [this](matrix_wrap<T> lhs, matrix_wrap<T> rhs) {
    return do_multiply(lhs, rhs);
  };

```

```
    std::shared_future<matrix<T>> ft =
11    async(std::launch::async,run_multiply, std::move(*lhs),std::move(*rhs));

13  matrix_wrap<T> mat(std::move(ft),lhs->get_height(),rhs->get_width());
    typename std::list<matrix_wrap<T>>::iterator result=matrices.emplace(lhs,mat);
15
    matrices.erase(lhs);
17  matrices.erase(rhs);
}
```

We can see that the future is wrapped in a `matrix_wrap` that at the end is pushed inside the expression list. Later on, when it will be picked-up in order to perform a new operation it is first evaluated inside the `do_multiply` method and the result is provided and repeat the procedure. Only the last step does not push inside the optimization list and returns directly the final result.

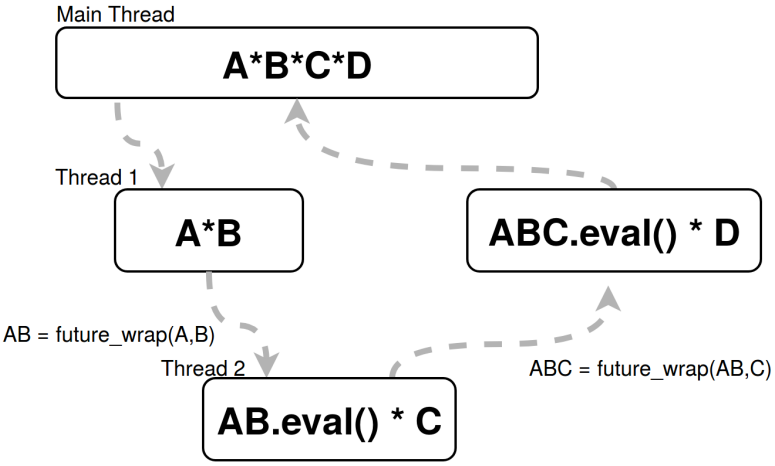

Figure 3: Parallel product

3.1.3 Block multiplication

The second task of the assignment requires the implementation of a parallel matrix multiplication. This can be obtained by dividing the matrix in blocks of size K (where K is an arbitrary constant); every pair of block-matrices is multiplied in a dedicated thread. The provided implementation extends the one described in Section 3.1.2.
Once the block size K has been chosen, the number of row-blocks and column-blocks that compose the resulting matrix can be calculated by taking the maximum size resulting from the division by K:

$$\#rows = \max\left(\left\lceil\frac{H_r}{K}\right\rceil, \left\lceil\frac{H_l}{K}\right\rceil\right), \quad \#cols = \max\left(\left\lceil\frac{W_r}{K}\right\rceil, \left\lceil\frac{W_l}{K}\right\rceil\right)$$

where:

- H_l is the left hand matrix height.
- W_l is the left hand matrix width.
- H_r is the right hand matrix height.
- H_r is the right hand matrix width.

This strategy is valid only if matrices can be divided in equally sized partitions, this is not always possible; for this reason they are padded with zeros. In fact every cell of a block matrix is copied in a brand new $K \times K$ matrix that is filled with zeros.
Every block of the resulting matrix $C = A \times B$ can be computed with the following formula:

$$C_{ij} = \sum_k A_{ik} * B_{kj}$$

for instance, given $K = 2$:

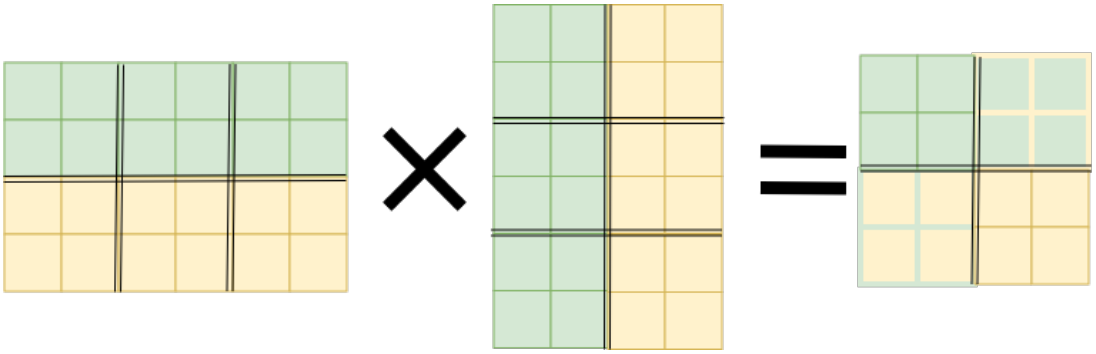

Figure 4: Multithread product

In order to perform a parallel block-multiplication, threads are required. There are several approaches that can be used:

- ◇ Simple threads
- ◇ Promise/Future semantics
- ◇ Async function
- ◇ Packaged task

The full usage of threads is not required, so the *Promise/Future* semantics can be used in order to add an abstraction layer to thread management. An improvement to this approach is the `std::async` function. It reduces further more the capabilities of threads, but simplifies the *Promise/Future* semantics, since the creation of a *Promise* is not required. In this case the function to run is passed as functor or as a lambda-abstraction to `std::async` this will result in a more clear code.

```
std::async(std::launch::async, foo, x, y)
```

Given an arbitrary size K, when a multiplication between two matrices is performed, every block A_{ik} and B_{kj} is copied to a plain `matrix<T>` object. The standard matrix product is performed between `matrix_wrap<T>` objects (plain matrices are wrapped in a `matrix_wrap<T>` list) that have an high polymorphic overhead because they need to pass through the *pimpl*.
A repeated and alternated access to two possibly huge matrices leads to a continuous access to the memory; the performance is very low if the access need to pass through a stack of classes. The process can be sped up by copying the matrices to two plain `matrix<T>` objects:

```
void multiply(const matrix_wrap<T> &lhs, const matrix_wrap<T> &rhs, matrix<T> &
    result, unsigned start_row,
            unsigned start_column) {

    ...

    matrix<T> block(K, K);
    matrix<T> Aij(K, K);
    matrix<T> Bji(K, K);
    for (unsigned j = 0; j < col_blocks; ++j) {
        matrix<T> Aij =
            lhs.get_block({start_row,start_row + K,curr_col,curr_col + K});
        matrix<T> Bji =
            rhs.get_block({curr_col,curr_col + K,start_column,start_column + K});
        curr_col = curr_col + K;
        matrix<T> prod = simple_prod(Aij , Bji);
        block = block + prod;
    }

    unsigned max_height = std::min(start_row + K, result.get_height());
    unsigned max_width = std::min(start_column + K, result.get_width());
    for (unsigned i = start_row; i < max_height; ++i) {
        for (unsigned j = start_column; j < max_width; ++j) {
            result(i, j) = block(i - start_row, j - start_column);
        }
    }
}
```

By doing this, the overhead of the access operation due to polymorphism is reduced, in fact the two matrices are retrieved only once and later on, only small portions of them are accessed. A possible strategy is to use the `window` function, but it cannot be used on `matrix_wrap<T>` objects because of cyclic type expansion that leads to an infinite loop. A simpler solution is to a perform a deep copy of the matrix portion that is required. This approach is more inefficient than an access by reference, but is better than an access by reference in a `matrix_ref<T>` object.

```
matrix<T> get_block(window_spec spec) const override {
    matrix<T> block(K, K);
    for (unsigned i = spec.row_start; i < spec.row_end; ++i) {
            for (unsigned j = spec.col_start; j < spec.col_end; ++j)
                    block(i - spec.row_start, j - spec.col_start) = mat(i, j);
    }
    return block;
}
```

When only the *pimpl* is visible, the `get_block` operation is delegated to it in order to avoid an access through `matrix_wrap<T>`:

```
matrix<T> get_block(window_spec spec) const {
    unsigned max_height = std::min(spec.row_start + K, height);
    unsigned max_width = std::min(spec.col_start + K, width);

    return pimpl->get_block({spec.row_start,max_height,spec.col_start,max_width});
}
```

In order to reduce the overhead more and more, the product between two blocks does not rely on the multiplication infrastructure built for the library usage (the lazy-like one). In that implementation, matrices are wrapped in `matrix_wrap<T>` objects and are inserted in a list, to avoid this, a simple matrix product is used. After this computation, the block is copied to the output matrix. Every product between blocks is performed in a dedicated thread.

```
matrix<T> do_multiply(matrix_wrap<T> lhs, matrix_wrap<T> rhs) {
    ...
    matrix<T> result(lhs.get_height(), rhs.get_width());

    std::vector<std::future<bool>> workers;
    ...

    for (unsigned i = 0; i < row_blocks; ++i) {
        for (unsigned j = 0; j < col_blocks; ++j) {
            workers.emplace_back(std::async(std::launch::async, product, lhs, rhs,
     std::ref(result), i * K, j * K));
        }
    }

    for (auto &worker: workers)
        worker.get();

    return result;
};
```

Generally, when more threads access concurrently to the same object, it is necessary to introduce some synchronisation mechanisms like *mutexes* and *monitors*. In this case, every thread accesses to the original matrices only for reading, whereas even if the output matrix is accessed for writing by several threads at the same time, every thread write to a portion of the matrix that is needed only by itself, so the behaviour is closer to parallelism than to concurrency. We can consider the access to the matrix as non-atomic.

Even if K is arbitrary, its value can affect the overall performance of the program. For instance, let $K = 5$, it can be considered as a small block size if $n, m \approx 30$, in this case matrices are

split in a lot of partitions, several threads run in parallel; if the matrix is smaller than 5 then only a small set of cells is wasted by padded zeros. In the case of matrices that are greater than a multiple of five, another block has to be added, but the number of padded cells is at most $5 \times \max(n, m)$.
On the other hand, choosing an high K implies a lower number threads involved in the computation, but the load of every thread will be heavier. If the matrix is small, several cells are wasted by padded zeros. In the case of matrices that are greater than a multiple of K, a great amount of memory will likely be wasted by padding.
In order to get the best performance from the library, a profiling should be performed trying to tune the block size K. The worst cases is so described:

$$n, m \ll K$$

$$n, m \in I^+(K, \varepsilon), \qquad K, \varepsilon \in \mathbb{N}$$

where $I^+(K, \varepsilon)$ is defined as a neighborhood of K:

$$I^+(K, \varepsilon) = \{k \in \mathbb{N} \quad s.t. \quad K < x < K + \varepsilon\}$$

The worst case is given when $n = m = K + 1$, the reason is that we create 4 matrices in which only one is dense, and the other 3 have only one column or 1 row with non-zero values. The following picture can explain better this case:

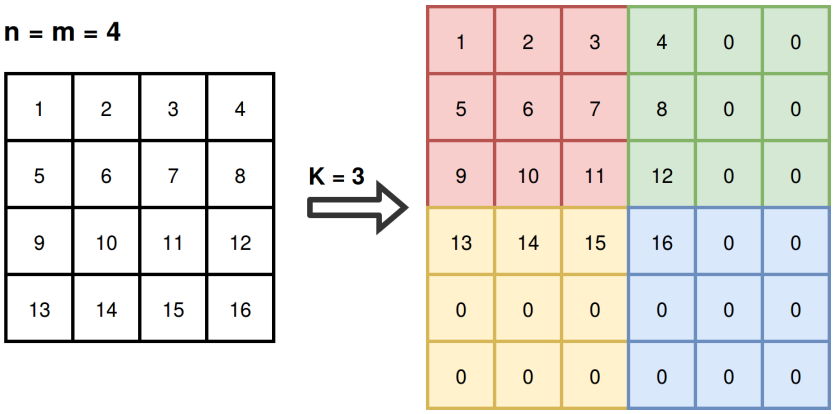

Figure 5: Worst case for K

So K should be tuned in order to use those matrices as little as possible. A possible improvement of the library in terms of performance could consist on an algorithm that every time that a product is performed, chooses the block size depending on the size of the two matrices, selecting the value that allow to reduce as much as possible the number of padded zeros.

4 Assumptions and Conclusion

Our extension essentially satisfies the requests for this assignment and introduce a structure that can be easily extended introducing new possible combinations. In fact parallel sum was not necessary for this assumption but the idea used for product was essentially the same so we decide to take advantage of this similarity including also parallelism for sum operator. More complex expression can be executed and performed as much as possible with multiple thread, for we don't provide all the expression, here we show few examples:

$$E * (F * E) \qquad (A + B)^T$$

On the first case, on the left, is is necessary to define a new operator that get in input a matrix on the left and an optimizer on the right. This is no provided from the previous version because this case is given by the usage of parenthesis inside expressions that introduce different priorities. The second case, on the right, as we have seen is a simple transformation a `future_matrix`, for the moment is not available but it is necessary only to override the original methods performing first the `eval` method and later return the matrix transformed. These operations are not required for this assignment but the entire structure allows to do that assuming that the matrix will be completed in the future. The constraints of this structure are: re-definition of the same structure for the `future_matrix`, to distinguish the static or dynamic matrices, given by decision of avoiding possible wraps; large set of possible combinations for operations, each case has be implemented separatly also if they have more or less the same code and behavior.
In order to give a complete library could be interesting to introduce also this operations and possibly other ones. And as we have seen another interesting improvement could be to develop an dynamic algorithm that adjust the value of K for the block parallel product between matrices. We have seen that the values of K have an important role in term of performance, for that reason a possible heuristic or strategy could be reasonable to adapt.