# MapReduce Skeleton

Giuseppe Miraglia
University of Pisa

Mar 7, 2016

## Contents

## 1 Introduction and Specifications

MapReduce is a generic programming model which, in his "Google version", allows users to specify a *map* function that processes a key-value pair to generate a set of intermediate key-value pairs, and a commutative and associative *reduce* function that merges all intermediate values associated with the same intermediate key [1].
More formally, given an input collection of initial key-value pairs $P_{init} = \langle\, p_1, p_2, \dots p_n \,\rangle$, a map function $f$ and a reduce function $\oplus$, if $P_{interm} = \{\, p \in P_{interm_i} \,|\, P_{interm_i} = f(p_i), \quad i = 1, \dots, n \,\}$ is the set of all the intermediate key-value pairs and $K_{interm} = \{k_1, \dots k_s\}$ is the set of all different intermediate keys of that set, then the final result will be:

$$\begin{aligned} mapreduce(f, \oplus)(X) = \langle & \bigoplus_{\substack{(k_1, v_j) \in P_{interm} \\ j=1,\dots,n_1}} (k_1, \langle\, v_{1_1}, \dots v_{1_{n_1}} \rangle\,), \dots \\ & \dots, \bigoplus_{\substack{(k_s, v_j) \in P_{interm} \\ j=1,\dots,n_s}} (k_s, \langle\, v_{s_1}, \dots v_{s_{n_s}} \rangle\,) \,\rangle \end{aligned}$$

Nowadays many real world tasks are expressible using this model, both in distributed and parallel systems and architectures.

In the following sections, this document wants to show the different phases used to structure, model, develop and implement a MapReduce C++11 library, based on *pthread*, suitable for POSIX shared memory multicore architectures.
From a more abstract way, the MapReduce library has been structured as an algorithmic skeleton modelling the composition of the two well known data parallel exploitation patterns, *Map* and *Reduce*. The skeleton has been implemented using a template-based approach.

In particular, two NUMA shared memory multicore machines have been used in order to evaluate and test the performances of the skeleton: the Intel® Xeon, which have 8 double hyper-threaded cores, and the Intel® Xeon Phi, having 60 four hardware-threaded cores.

## 2 Skeleton design and semantics

In order to model, illustrate the design and capture the semantic of the MapReduce skeleton, we can exploit the *building blocks* and the *component expressions* of the **RISC-pb$^2$l** library. Among all the possible alternatives that could be used to model the skeleton, it has been considered also the possibility to give its functional semantic through the definition of *high order functions*, together with a short statement representing the parallel semantic. At the end, the first option has been preferred because of the expressive power of RISC-pb$^2$l building blocks and of the easy and clean syntax of its component expressions.

### 2.1 Principles and features

Following the newest reviewed Cole's algorithmic skeleton definition, the MapReduce skeleton (must) own all the following fundamental features:

- it models a *known*, *common* and *efficient* data parallel exploitation patterns (i.e. the composition of *Map* and *Reduce* patterns);
- it is *portable*, provided recompilation, over all the shared memory environments compatible with POSIX and C++11, since it is provided as library written using the programming language itself;
- it is *parametric* over the following set of functional and non-functional parameters:

  1. the user-defined *map* function, indicated by $f$ and defined as follow:

     $$f : (K_1, V_1) \rightarrow [\,(K_2, V_2)\,] \tag{1}$$

     where:

     – $K_1$ and $K_2$ are the types of the initial and intermediate keys, respectively;
     – $V_1$ and $V_2$ are the types of the initial and intermediate values, respectively;
     – $[\,(K_2, V_2)\,]$ represents the list of intermediate key-value pairs obtained as result of the application of the function.

2. the user-defined *reduce* function, commutative and associative, indicated by $\oplus$ and defined as follow:

$$\oplus : (K_2, [\,V_2\,]) \;\rightarrow\; [\,V_2\,] \tag{2}$$

where:

   – $K_2$ and $V_2$ are the type of the intermediate keys and values, respectively;
   – $[\,V_2\,]$ represents the list final values obtained as result of the application of the function.

3. the user-defined *comparison* function, used to compare intermediate keys, indicated by $\doteq$ and defined as follow:

$$\doteq : (K_2,\, K_2) \;\rightarrow\; int \tag{3}$$

where:

   – $K_2$ is the type of the intermediate keys;
   – the result is -1, 0 or 1 depending if the first key is less, equal or greater than the second, respectively.

4. the *number of processing elements* to use during the computation (i.e. the parallelism degree used by the skeleton);

- it is *reusable* over an huge set of applications, without requiring modifications, thanks to its parametricity and to the nature of the modeled parallel exploitation pattern itself;

## 2.2 Design and formal semantic

The design chosen for the MapReduce skeleton can be explained through the following component expression:

$$MapReduce(f, \oplus, \doteq, n) = Map(f, n) \;\bullet\; Aggr(\doteq) \;\bullet\; Reduce(\oplus, n) \tag{4}$$

where, in turn:

$$\begin{aligned} Map(f, n) &= \lhd_{scatter} \;\bullet\; [|\, ((\, f \,))\, |]_n \;\bullet\; \rhd_{gatherall} \\ Reduce(\oplus, n) &= \lhd_{scatter} \;\bullet\; [|\, ((\, \oplus \,))\, |]_n \;\bullet\; {}_{\oplus}\rhd \end{aligned} \tag{5}$$

Let's now analyze individually each component of the design layout:

1. the *Map* component uses the *scatter* building block to split the single input collection into $n$ blocks and to direct each of them toward a concurrent activity among the $n$ available in the parallel string, which in turn are able to compute $f$. Then, once all the intermediate key-value pairs has been computed, the component uses the *gatherall* building block to merge the $n$ blocks into a single block. Finally, it moves the single block to the next stage;
2. the $Aggr(\doteq)$ component receives in input the merged block of intermediate key-value pairs and exploits the comparison function $\doteq$ to "aggregate" similar intermediate keys close one each other. Finally, it moves the aggregated block of intermediate key-value pairs to the next stage;
3. the *Reduce* component, like the Map component, uses the *scatter* to split the (aggregated) input into $n$ output blocks and to direct each of them toward a concurrent activity among the $n$ available in the parallel string, which in turn are able to compute $\oplus$. Then, the procedure is repeated recursively moving the reduced data toward the next level of the tree, until more than a single result is obtained. Finally, a single block of data representing the MapReduce result will be moved on the output.

It is important to remark that the previous described model works only under the assumption of commutativity and associativity of $\oplus$. Without commutativity, the order among the reduces would be relevant, preventing aggregation operations. In order to show the relevance of the associativity, instead, a very simple example can be built using $\oplus = mean$ on the input $[\,(k,3),\ (k,5),\ (k,7)\,]$. Obviously, the result of $\oplus(k,[\,3,\ 5,\ 7\,])$ should be 5, but if two concurrent activities process in parallel $\oplus(k,[\,3,\ 5\,])$ and $\oplus(k,[\,7\,])$, respectively, we would first obtain $\oplus(k,[\,4,\ 7\,])$, then the result 5.5, that is clearly incorrect, since *mean* is not an associative function.

## 3 Skeleton implementation

During the development of the MapReduce skeleton, various *implementation templates* have been considered with the aim of both obtain the best performances and a modular implementation design. Pros and cons have been evaluated for each of them, and in all cases the selection process aims to choose an implementation template individually for each component of the (4).

Furthermore, since all the templates has been developed for shared memory multicore architectures under POSIX/C++11 environments, during the description we will always consider this assumption (except for where different specified) and we will restrict the possible heterogeneity of processing elements considering only **threads** running on the same machine.

## 3.1 Input file(s) reading and Records extraction

In order to make able the skeleton to obtain read data, the class *FileRecordReader* has been implemented. The class is able to read file(s) located at the specified File System's path and, using an user defined *RecordFormat*, to extracts input *Record*s from the files. At the end of the process, a vector of Record will be loaded in main memory. The file reading process is performed sequentially on a single file, but it has been optimized in order to read multiple file in parallel.

## 3.2 Map templates

In the following sections we will describe implementation templates for all the components of the *Map* part of the MapReduce skeleton (see (5)).

### 3.2.1 Input splitting phase

This phase has been implemented by the *TaskGenerator* class, which offers a simple and intuitive behaviour: given an input vector of Records and a number of blocks to produce, it *logically* splits the vector in the requested number of blocks. The class *Task* represents a block: it simply stores two indexes of the input vector, representing the logical delimiters for a portion of data which will be processed by a thread during the Map or Reduce phase. The component first fills each Task in such a way they will contain the same (maximum) number of Records, then it spreads the remaining Records among all the Tasks extending the contiguous range of them. This method prevents both race conditions and false sharing.

### 3.2.2 Splits assignment phase

This phase has been implemented by (part of) the *Master* class. Given a vector of Tasks, the Master assigns a task to a **Mapper** (implemented by the class *Mapper*), which represents a thread able to extract the initial pairs from the Records and to apply the map function on them, of course in parallel w.r.t. the other threads. After that all the Tasks have been assigned to the Mappers, the Master needs to "synchronize" with them in order to gather the results of the computation.

### 3.2.3 Map function computation

During this phase, defined in the *Mapper* class, each Mapper scans iteratively the portion of Records vector represented by the assigned Task, performing the following actions:

1. it extracts the initial key-value pair from the $i$-th input Record, using an user-defined *KeyValueFormat*;
2. it applies the map function on the above extracted initial pair and stores the results into the dedicated result vector.

After the Task has been processed, the Mapper synchronizes with the Master, as described in the next section.

### 3.2.4 Master-Mappers synchronization

About this purpose, two alternatives mechanisms have been considered.

#### Active wait Master

A first alternative consists in performing **active wait** on the termination of the Mappers. Executing in this way, the Master is able to aggregate the data through a *gather* policy, instead of *gatherall*. As it is simple to imagine, this behaviour has pros and cons:

- in positive, if the Tasks are internally unbalanced, the aggregation of already available results is overlapped w.r.t. the excess time spent by the other Mappers;
- in negative, the skeleton implicitly decreases its *parallelism degree* by 1, since a thread is monopolized by the Mapper.

### “Sleeping” Master and dedicated Mapper-results

A second approach (the one used in the final version of the skeleton) provides a good alternative to the previous solution. First of all, in order to recover the maximum parallelism degree, a **mutex** and **condition variable** have been shared among the Master and the Mappers. After the Tasks assignment phase, the Master waits on the condition variable until all the Mappers have not finished their work. Each Mapper, instead, at the end of the work checks if it is the last active: if so, it notifies the Master on the condition variable, as implemented in the *Synchronizable* class. Second, a *std::vector<std::vector>* of size $n$ has been used: the idea is to assign the **$i$-th sub-vector to the $i$-th Mapper** in such a way each Mapper is able to store the results without using any kind of synchronization w.r.t. the other Mappers.

## 3.3 Aggregator templates

In the following sections we will describe implementation templates for the *Aggregate* part of the MapReduce skeleton (see (4)). The goal of this phase is to move all the intermediate pairs having the same key close one from each other, in order to simplify the future work of the Reducers.

### 3.3.1 Hash-based aggregation

A strategy that can be used is to compute an hash function on the keys, mapping all the same keys on a particular Reducer. For this purpose, an *std::unordered_map* [2] can be used like an hash table in which insert all the intermediate pairs. The data structure maps each key in a vector of values, and .
This approach works well in a distributed context, but in our case it has been discarded for the following reasons:

- the worst computational time in inserting a pair into the hash table is $O(m)$, where $m$ is the size of the data structure itself; this cost must be added to the time spent to compute the hash function on the key;
- in order to access the vector of values associated to a key, we need first to pay the memory accesses for the key itself: it is more convenient to use directly a vector;

- because of the structure and of the allocation of the hash table, a Reducer cannot exploit data locality and prefetching among different groups of keys;
- after the mapping of the keys, a local sort must be performed by each Reduce in order to improve the application of the reduce function.

### 3.3.2 Sort-based aggregation

Another, more direct, strategy that can be used to aggregate intermediate pairs is to sort them by key using the comparison function. In this way, the results of the sorting algorithm will grant a direct access to sorted data and can be efficiently exploited by the next phases in terms of data locality and prefetching.
The naïve approach uses the *std::sort* algorithm, but as we will see in the section 4.3, some optimizations can be performed in order to made the sorting algorithm more efficient.

## 3.4 Reduce templates

In the following sections we will describe implementation templates for all the components of the *Reduce* part of the MapReduce skeleton (see (5)).

### 3.4.1 Input splitting phase

Two different templates have been studied for the input splitting phase. Both the templates aims to organize in a tree the computations to apply on the splitted data, but they differs in the way with which the tree is structured.

#### Data-based tree

A tree having as leaves all the intermediate key-value pairs of the aggregated vector is "built". If there were $m$ pairs, then the tree would have $\log_2(m)$ levels. In this case, even if the number of applications of the reduce function would be only $log_2(m)$, in order to be computed in an efficient way they need $m$ threads running in parallel. For this reason, the template described in the following section has been preferred in the final implementation.

#### Thread-based tree

Instead of organize the tree based on input data, this template exploits the variable arity of the reduce function building a logical **two-levels** tree having as leaves the $n$ available threads, and as root a single thread performing a final reduce on the $n$ results obtained from the leaves. In this way, the $m$ aggregated intermediate pairs will be distributed among the different threads (each thread will process $m/n$ pairs). So, as already described for the *Map*, this phase has been implemented by the *TaskGenerator* class, performing nothing different w.r.t. as described in 3.2.1. The only difference is on the input which, this time, is a vector of **aggregated** intermediate key-value pairs.

### 3.4.2 Splits assignment phase

As already described for the *Map*, this phase has been implemented by (part of) the *Master* class. Given a vector of Tasks, this time the Master assigns each task to a **Reducer** (implemented by the class *Reducer*), which represents a thread able to apply the reduce function on the intermediate pairs of the assigned Task, of course in parallel w.r.t. the other threads. After all the Tasks have been assigned to the Reducers, the Master needs to "synchronize" with them in order to gather the results and apply the final reduce.

### 3.4.3 Reduce function computation

This phase (defined in the *Reducer* class) is executed scanning iteratively the portion of aggregated intermediate pairs represented by the assigned Task. During the scanning, each Reducer performs the following actions:

1. it tries to identify an entire group of values having the same key, using the comparison function;
2. once an entire group has been identified, it applies the reduce function on that group and stores the results into the dedicated result vector;
3. finally, if no other groups exists, the procedure ends; otherwise, the process continue from the step 1) starting from the first intermediate pair of the new group.

After that the whole task has been processed, the Reducer synchronizes with the Master, as described in the next section.

### 3.4.4 Master-Reducers synchronization

Also in this case, the "sleeping" Master alternative has been used in order to synchronize Master and Reducers (see 3.2.4).

## 4 Template optimizations

This section aims to tell some optimizations performed on the templates chosen for the components used to implement the final version of MapReduce skeleton.

### 4.1 Threads creation

As told in the above sections, before assign a Task to a Mapper or a Reducer, the Master needs to start a thread for it, introducing a non-negligible and non-avoidable overhead which impacts the skeleton performances, especially when the chosen number of threads to use is big. In order to reduce this overhead, a **pool of "ready" threads** has been implemented through the *ThreadManager* class. Immediately before the Map phase, the Master initializes the ThreadManager with $n$ threads which will be used both as Mappers and Reducers. Each thread of the ThreadManager has been implemented through the class *ActiveWaitThread*, which is a convenience wrapper containing a C++11 thread performing active wait together with some synchronization field. Each thread looks for the availability of an *Executable* job, that can be assigned only by the Master. The active wait approach has been preferred because basically:

- there are no constrains on the usage of the computational resources;
- it provides better performances w.r.t. an approach that uses synchronization.

In case the resources usage becomes a fundamental requirement, the ThreadManager can be easily modified in such a way the threads are synchronized using condition variables and/or low level signals.

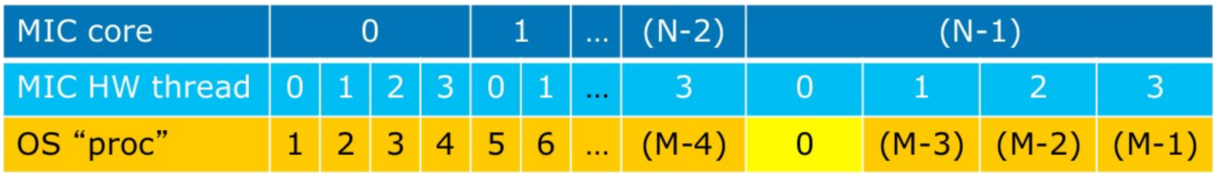
Finally, in this phase has been individuated a compatibility problem between the *icc compiler* and the C++11 *atomic struct* [3]: even if the compiler produces no errors/warnings, the program crashes (SIGSEGV) at runtime on the Xeon Phi. Even if the compiler installed on the machine is up-to-date, the problem is probably caused by missing the required *gcc 4.8* environment, so a full C++11 support, as described at [4].

## 4.2 Threads affinity

Much attention has been spent to set the **affinity** of each thread: each thread is pinned on a different processor for the whole computation. Furthermore, depending on the architecture on which the code is compiled, different **affinity assignment policy** have been implemented. The class *Scheduler* changes the affinity assignment policy based on the definition of some macro:

- if the macro XEON is defined, the affinity of the threads will be set on the cores 0, 2, . . . , 13, 15, in the order;
- if the macro PHI is defined, the affinity of the threads will be set on the cores 1, 5, 9, . . . , 237, 238, 239, 0, in the order;
- if no macro are defined, the affinity of the threads will be set following the natural order of the cores.

An architecture-personalized affinity assignment policy has been designed in order to improve performances, but of course needs knowledge about the architecture itself. For example, as shown in the following figure, the cores of the Intel Xeon Phi are numbered from 0 to 59, and the OS is placed on the processor 0, which lives on the last core, then it is convenient to pin the threads in order to avoid that core until it is not necessary.



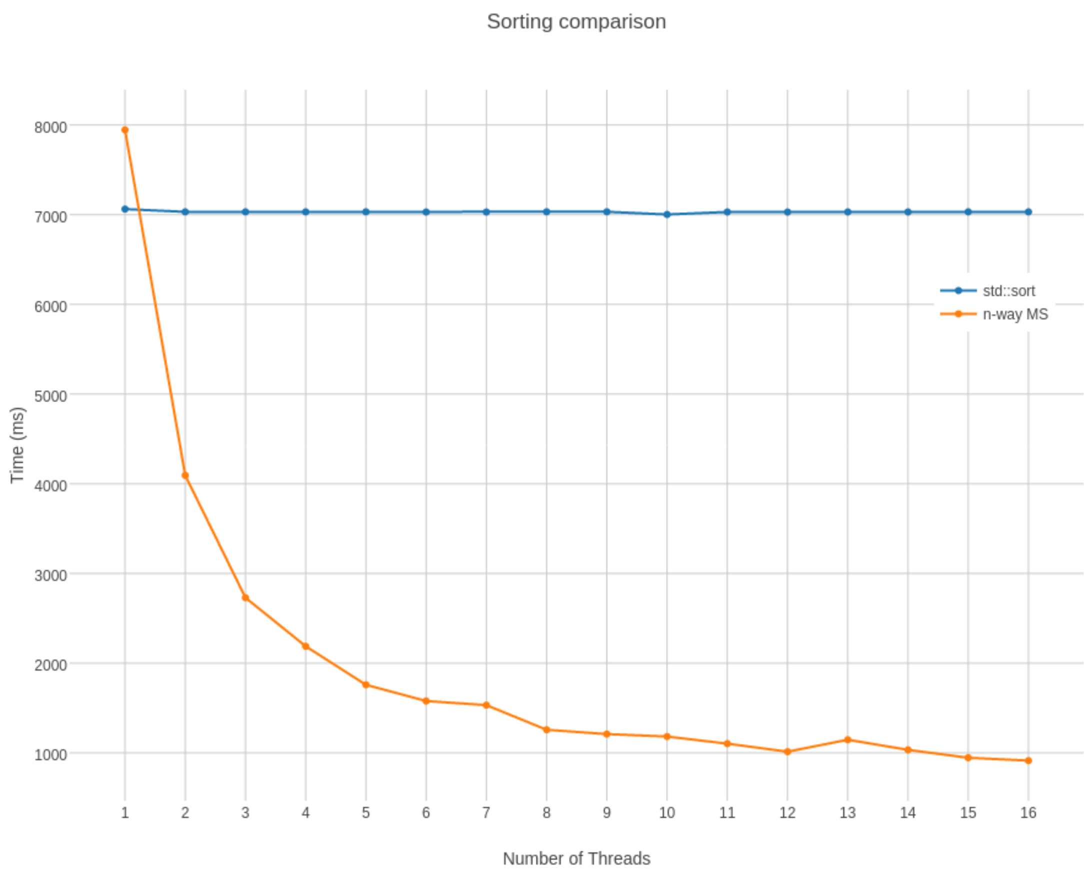
| MIC core | 0 | | | | 1 | | ... | (N-2) | (N-1) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MIC HW thread | 0 | 1 | 2 | 3 | 0 | 1 | ... | 3 | 0 | 1 | 2 | 3 |
| OS “proc” | 1 | 2 | 3 | 4 | 5 | 6 | ... | (M-4) | 0 | (M-3) | (M-2) | (M-1) |

### 4.3 n-way Merge-Sort

In order to improve the performance of the sorting algorithm, the **n-way Merge Sort** has been implemented through the class *Sorter*. The algorithm runs as follow:

1. first, each Mapper locally sorts the own portion of intermediate pairs;
2. then, the Master initializes a **min-Heap** (implemented by the class *ImplicitHeap*) with the first intermediate pair of the local result vector of each Mapper (max $n$ pairs inserted);
3. iteratively, the Master extracts the smallest intermediate pair from the Heap ($O(1)$ time) and moves it into the sorted vector. Then, another intermediate pair (belonging to the same vector result of last extracted pair) is inserted into the Heap ($O(log_2(n)$ time).

At the end of the process, all the local results vector of the Mappers will be empty and the intermediate keys will be in a sorted vector. A comparison among the *std::sort* and the $n$-way merge sort on a vector of 100000000 items is shown in the following plot.



## 5 Performance model

Based on the chosen component templates, the performance model of the template used to implement the effective MapReduce skeleton is:

$$\begin{aligned} T_{C_{seq}}(N_{input}, f, \oplus, \doteq) = T_{readfile}(1) + N_{input}\left(T_{initpairsextract} + T_f\right) + \\ + T_{sort}(N_{intermpairs}, \doteq) + N_{intermpairs}\, T_{\oplus} \end{aligned} \tag{6}$$

$$\begin{aligned} & T^{templ}_{C_{MapReduce}}(N_{input}, n, f, \oplus, \doteq) \\ & \quad = T_{readfile}(n) + T_{taskgen}(N_{input}, n) + T_{map}(N_{input}, n) + \\ & \quad + T_{sort}(N_{avgintermpairs}, \doteq) + T_{nwayms}(N_{intermpairs}, \doteq, n) + \\ & \quad + T_{taskgen}(N_{intermpairs}, n) + T_{reduce}(N_{intermpairs}, n) \end{aligned} \tag{7}$$

where

$$\begin{aligned} T_{map}(N_{input}, n) &= \frac{N_{input}}{n} \left( T_{initpairsextract} + T_f \right) \\ T_{reduce}(N_{intermpairs}, n) &= \left( \frac{N_{intermpairs}}{n} + n \right) T_{\oplus} \end{aligned} \tag{8}$$

and

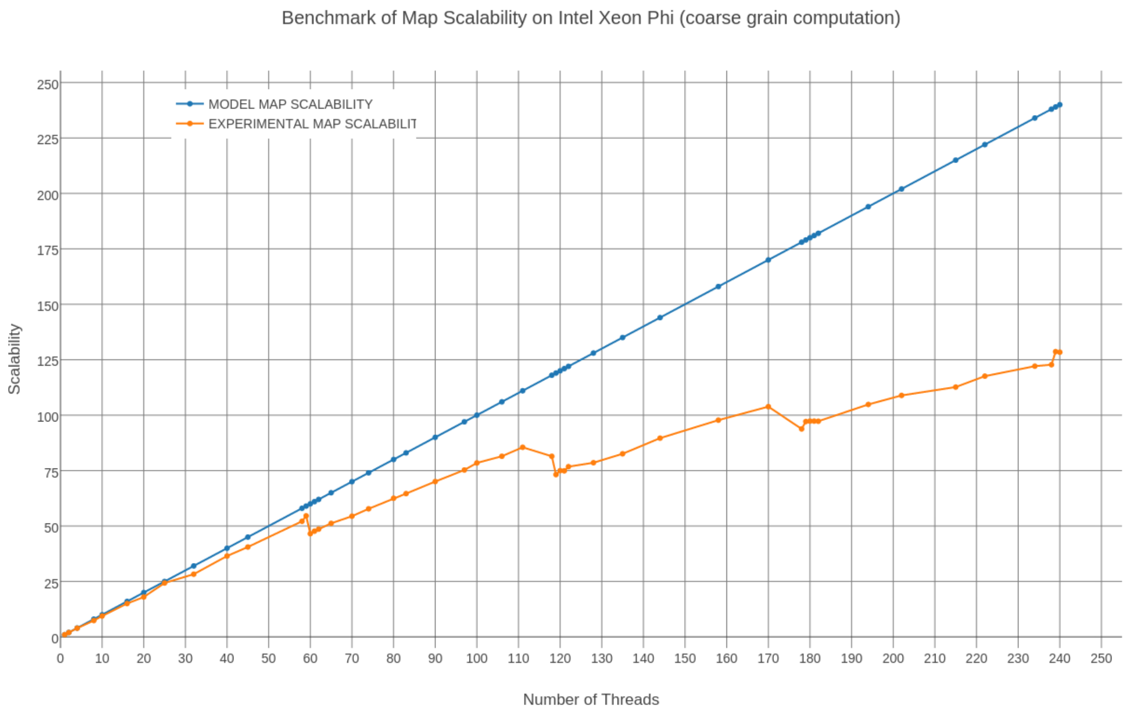
- $N_{input}$ is the size of the input vector of Record;
- $N_{intermpairs}$ is the total number of intermediate pairs obtained from the Map phase;
- $N_{avgintermpairs}$ is the average number of intermediate pairs obtained by a Mapper;
- $n$ is the parallel degree, i.e. the number of threads to use during the computation;
- $f$ is the map function and $T_f$ the time needed for an its application;
- $\oplus$ is the reduce function and $T_{\oplus}$ the time needed for an its application;
- $T_{readfile}(k)$ is the time needed to read the input file(s) using at most $k$ threads;
- $T_{taskgen}(k, n)$ is the time needed to generate $n$ Tasks from an input of $k$ items;
- $T_{initpairsextract}$ is the time needed to extract an initial pair from a Record;
- $T_{sort}(k, \doteq)$ is the time needed to sort a vector of $k$ items through the C++11 $std :: sort$ algorithm and using $\doteq$ as comparison function;
- $T_{nwayms}(k, \doteq, n)$ is the time needed to apply the $n$-way merge-sort to sort an input of $k$ intermediate pairs using the comparison function $\doteq$.

## 6 Experimental results

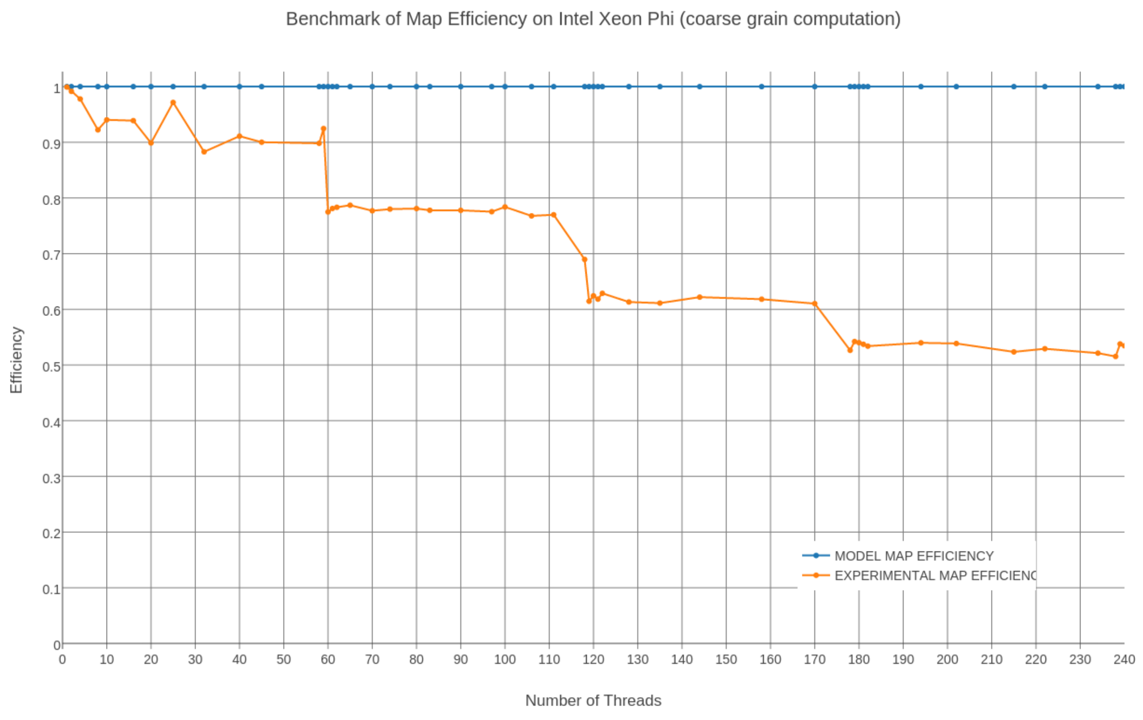
The performances of the skeleton have been tested on both the Intel Xeon and Intel Xeon Phi, but the first has been used only as comparison metric w.r.t. the second, so its result have not been report in this section. Mainly, the performances has been tested through two kinds of applications: *CosCos* and *UserID Count*.

## 6.1 CosCos

This kind of application, implemented into the file *CosCos.hpp*, has been studied in order to show the behaviour of the skeleton on **coarse grain** computations: both the map and reduce functions repeatedly apply the mathematics *cosine* function. In particular, the reduce function applies the cosine a number of times that is proportional to the number of values for the processed key: in this way, the weight of the reduce is effectively "reduced" on the second level of the tree. In this section we will show plots about the performance of the Map and Reduce phases and about the completion time of the application. First of all we show some plots about the Map phase derived from experimental results made on Intel Xeon Phi.

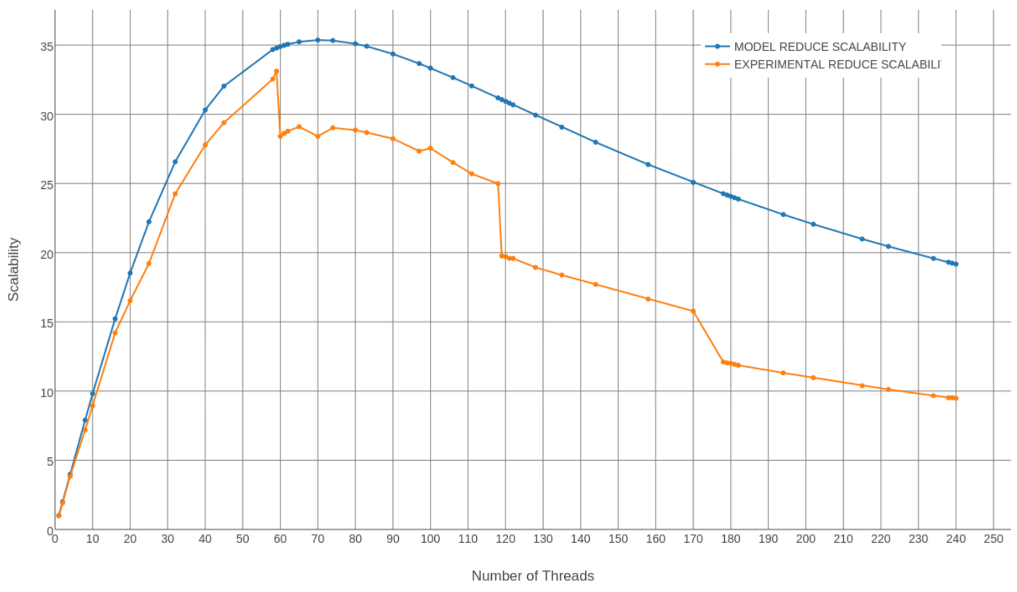


We can see from the scalability plot that, as expected, the behaviour on the first 60 processors is approximately the expected one, while on the second, third and fourth 60 processors the scalability increases more and more slowly: this is caused by the thread pinning policy applied for the Phi and by the fact that the L2 cache is shared among the processors of the same core (note the losses when we use 60, 119 and 179 threads). The efficient resources usage can be better shown on the efficiency plot.
The plot clearly shows a loss of efficiency when we start to use the second,

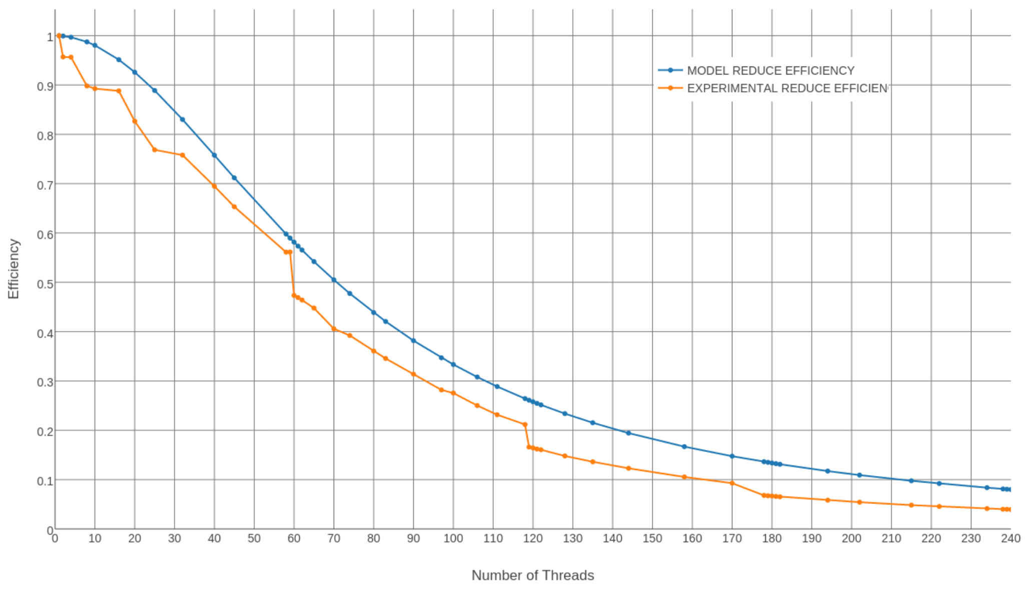


third and fourth processor of each core, respectively, and it shows also a stable “pattern” during filling the same “level” of processors. The same kind of description can be made in order to the describe the Reduce phase. This time, the shape of the scalability is not the classic $y = x$ function, instead it is a curve which, in this case, starts decreasing when we use about 70 threads. The reason why this happens is obviously related to the layout of the Reduce implementation and so on its performance model (see (8)): increasing the number of threads, the first term will decrease, but the second will increase linearly. Again, through the plot of the efficiency, we can better see the exponential decreasing of the efficient resource usage.

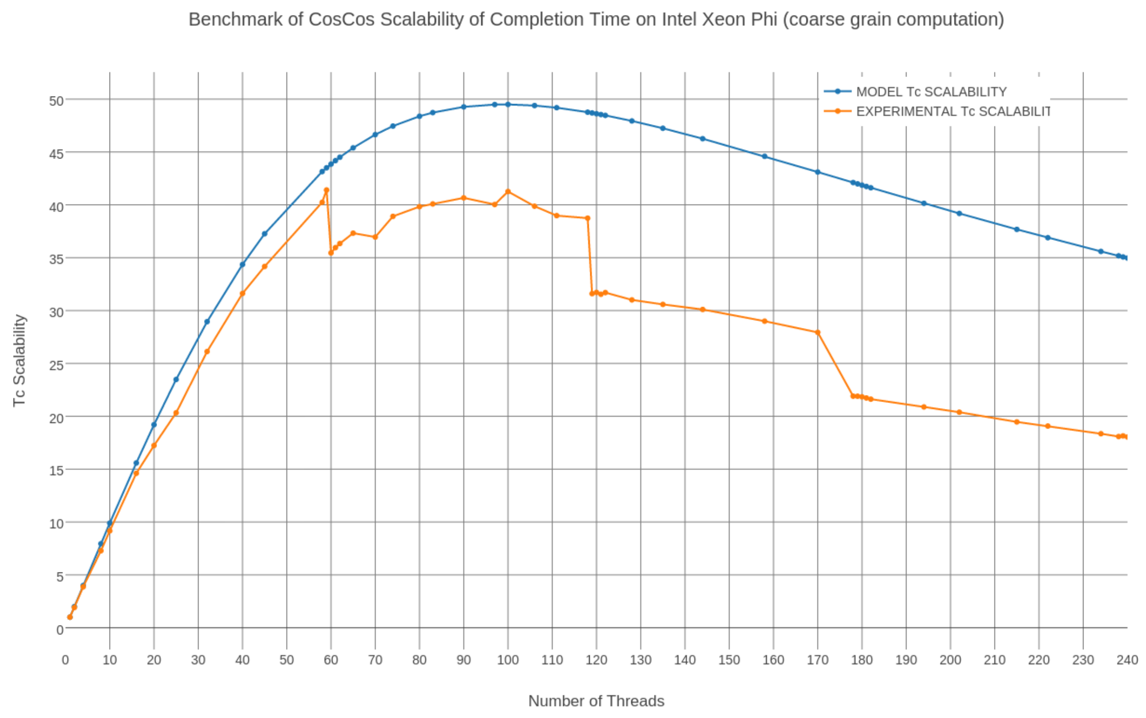
Benchmark of Reduce Scalability on Intel Xeon Phi (coarse grain computation)



Benchmark of Reduce Efficiency on Intel Xeon Phi (coarse grain computation)



Finally, we show a plot of the scalability of the whole application w.r.t. the completion time: even if the shape is similar to the one shown for the scalability of the Reduce, since both the Map and Reduce phases dominate the computation, the curve reaches an highest level w.r.t. the Reduce itself.

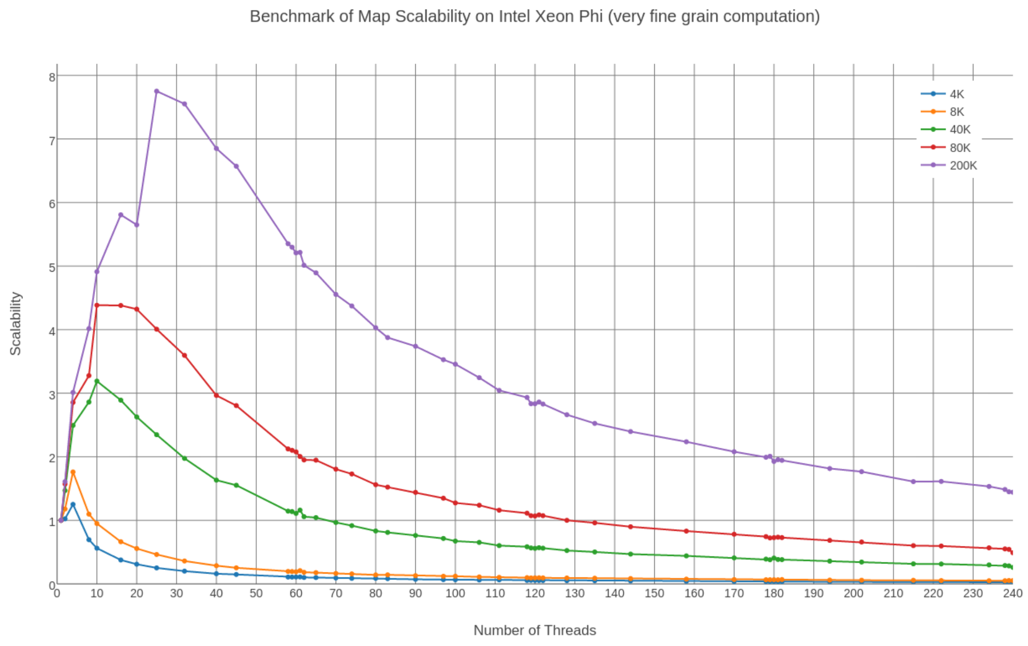


The plots representing the speedup have not been reported because of the quite similarity w.r.t. the plots of the scalability.

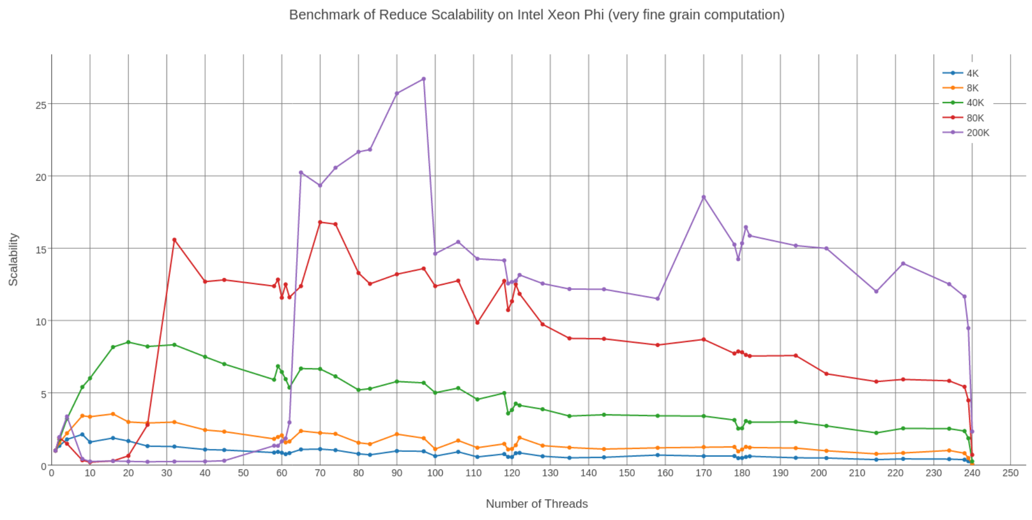
## 6.2 UserID Count

Another application, implemented into the file *UserIDCount.hpp*, has been studied in order to show the behaviour of the skeleton on **very fine grain** computations. The application is a simple integer-variation of the well known *word count*. As already did for the previous example, let's consider some experimental results for increasing number of initial pairs. With respect to the CosCos application, from the plots on next page we can see this time we achieved very poor performances.

As expected, the very fine grain of the computation is dominated by memory and skeleton management times. In particular, further investigating about this performance degradation, we analysed that the time needed to apply the map function on a single item of a Task is very low w.r.t. the time

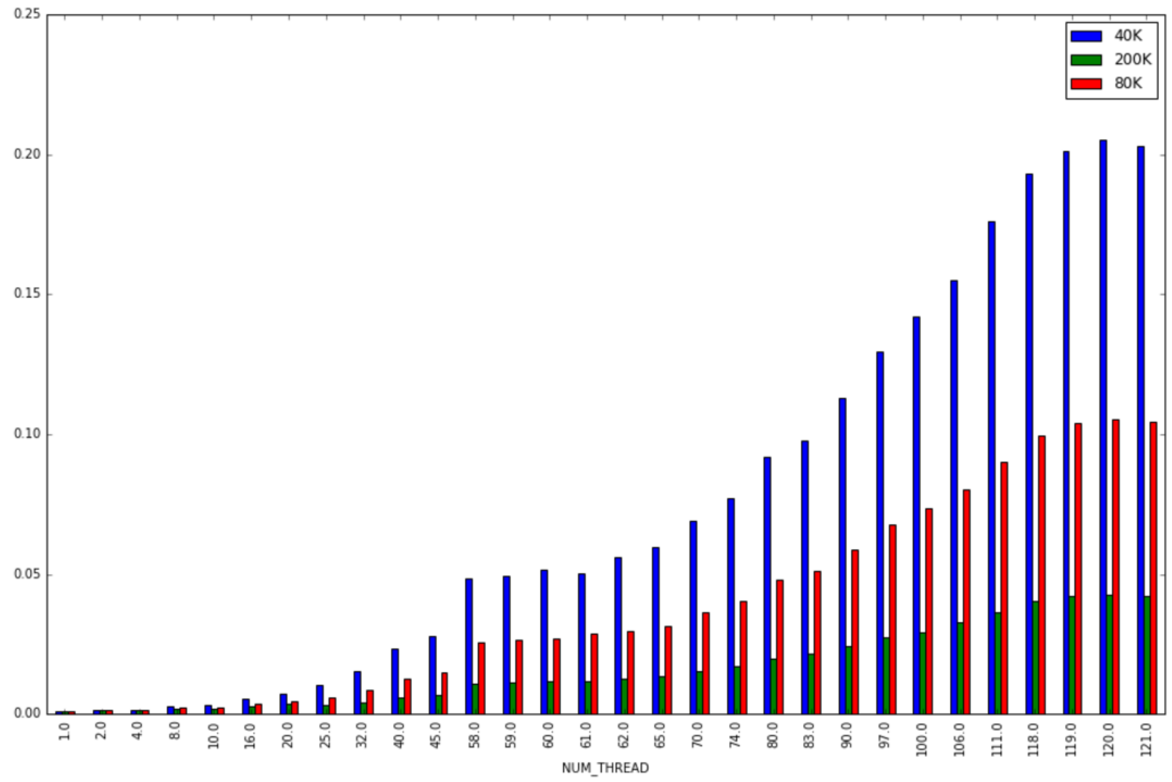


Benchmark of Map Scalability on Intel Xeon Phi (very fine grain computation)
Scalability
Number of Threads
4K
8K
40K
80K
200K



Benchmark of Reduce Scalability on Intel Xeon Phi (very fine grain computation)
Scalability
Number of Threads
4K
8K
40K
80K
200K

spent to load from the cache the Task itself, since the first one consists in simply allocating and emitting a new intermediate pair. In this way, while the computation of the map function on all the items of an huge Task is overlapped to the time spent to load the items themselves from the cache (only the first block, of course, because of the prefetching), the time will be not covered if we consider a block which is too small. Furthermore, because of the scatter policy, increasing the number of thread lead us to have small blocks. This means that, soon or later, depending on the size of the input data, the memory will become the bottleneck of the application, completely dominating all the other times.



From the plot shown above, we can observe that the average time needed to compute the Map phase increase significantly together with the number of threads, but it is low when we consider bigger amounts of data. We can so conclude that the chosen skeleton implementation is not suitable for such kinds of fine grain computations. As well known, already existing MapReduce implementations like those provided by *Hadoop* [5] and *Spark* [6] use ad-hoc memory management, distributed File Systems and very sophisticated optimizations in order to achieve very high performances also on fine grain computations.

## 7 User manual

The skeleton is provided to the final user as a C++11 library. To use the library, the user must includes the *MapReduce.hpp* header into its code. The class constructor has different parameters which have to be specified in order to use the skeleton. The meaning and the usage of each parameter is well described in the header file itself.
For example, the file *MR.cpp* represents a main program containing different applications with which the tests have been performed. In order to compile the program on the target architecture, the *Makefile* contained in the same directory can be used. Each compile rule of the Makefile has a name which respects the following structure:

$$compileARCH\,[ACTION]$$

where:

- $ARCH$ could be a value in $\{standard, xeon, mic\}$ and it is used to specify the kind of architecture for which the program must be compiled;
- $ACTION$ could be an optional value in $\{verbose, check, report\}$ and it could be used to insert/delete pieces of code which make the program either interactive or able to generate a report of the performances.

After the compilation, in the case of *MR.cpp*, the program can be executed specifying the number of thread to pass to the skeleton, simply invoking the command:

$$./MR\ \ num\_thread$$

## References

[1] *Google MapReduce*
http://static.googleusercontent.com/media/research.google.com/it//archive/mapreduce-osdi04.pdf

[2] *C++ std::unordered_map*
http://www.cplusplus.com/reference/unordered_map/unordered_map/insert

[3] *C++ std::atomic*
http://en.cppreference.com/w/cpp/atomic/atomic

[4] *icc: supported features*
https://software.intel.com/en-us/articles/c0x-features-supported-by-intel-c-compiler

[5] *Apache Hadoop*
http://hadoop.apache.org/

[6] *Spark: Lightning-fast cluster computing*
http://spark.apache.org/