# Distributed Map / Parallel For Pattern

Parallel and Distributed Systems: Paradigms and Models

Tonci Nicolò - Matricola: 503919
Master Degree in Computer Science & Networking

January 2021

**Abstract**

Map is a structured data parallelism pattern in which a functional operation is applied to all elements of a sequence. In this document we present an implementation of a distributed map micro framework leveraging raw sockets, FastFlow[1] and Cereal[2]. We also provide some performance measurements on a cluster of 16 servers interconnected with a Gigabit switched Ethernet.

## 1 Introduction

The Map operation function is applied to each element of a homogeneous collection of data. It is usually represented by an high-order function that takes in input a collection and a closure and returns a new collection as result. If the closure is stateless (or even has a read-only state), the execution of the Map pattern can be parallelized by simply splitting the collection in partitions that may be computed in a completely independent way. The result is produced from the trivial merging process of sub-results.
In practice, the collection is split in partitions and scattered to the workers which compute the function on all items of the partition and give back results to a gather entity which implements a barrier and is in charge of merging all sub-results.
More formally, the Map skeleton can be expressed using RISC-Parallel Builing Block Library[3] (RISC-pb$^2$l) as follow:

$$Map(f, nw) = \triangleleft_{scatter} \bullet [|((f))|]_{nw} \bullet \triangleright_{gatherall}$$

where $f$ represents the stateless function to be applied and $nw$ the parallel degree.

This parallel pattern has been implemented for distributed memory environments (e.g., clusters) composed by shared memory multi core machines. In particular, a cluster of 16, Gigabit interconnected, NUMA multi-core machines have been used to evaluate and test performances of the proposed solution.

## 2 Design

To illustrate the design of our micro framework we will proceed in a top-down description. From a general view point, the implementation skeleton is a master-worker template of processes, where communications are in the distributed memory domain, as shown in Figure 1.

The master in our design is in charge of scheduling sub-tasks to a set of identical workers and collecting from the workers the computed results. The master node performs both *scatter* and *gatherall* operations. In addition, since the framework allows the map function

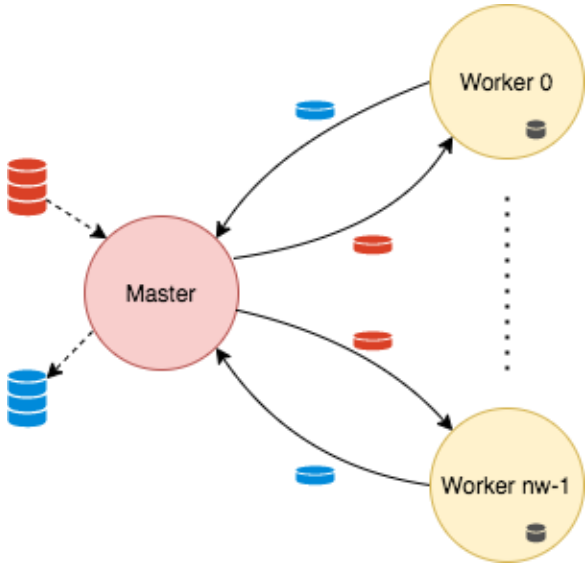


Figure 1: Master-Worker schema (with *nw* workers) implementing the distributed Map. Dotted arrows represent shared memory communications, whereas solid arrows represent distributed memory communications.

to access shared read-only resources, the master is responsible, during the startup phase, to send those resources to all worker processes.

In the following paragraphs and sections we will provide more details both to the design of the master as well as of the worker processes.

Each process is made by a *FastFlow*'s pipeline of three stages: the first is a thread receiver, the second is a thread that executes either the scheduling code (in the case of the master process) or the business code (in the case of the worker process), and the third thread is the sender. In Figures 2 and 3 are sketched the internals of master process and worker process respectively.

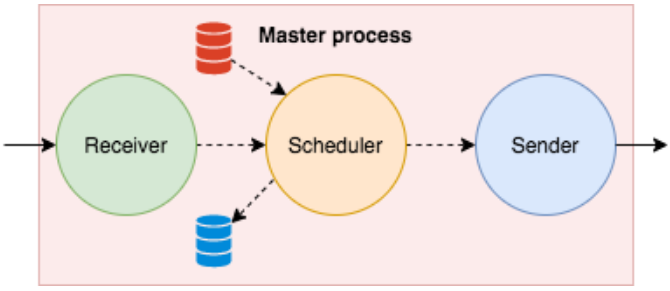


Figure 2: Internal pipeline of master process. Dotted arrows are shared memory communications, included the scan of the input collection and the write of the results collection. Solid arrows are distributed memory channels.

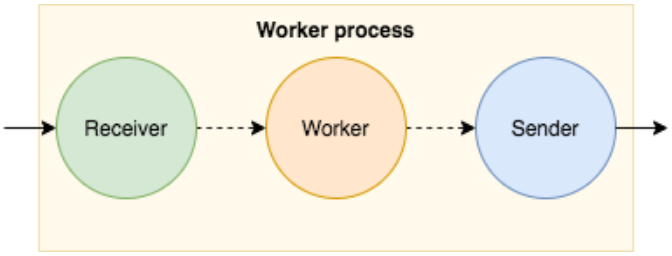


Figure 3: Internal pipeline of worker process. Dotted arrows are shared memory communications. Solid arrows are distributed memory channels.

Each pipeline is a *FastFlow*'s[1] `ff_pipeline` building block and each stage is a single-input single-output `ff_node_t` node. In the following sections we will describe with more details each stage.

## 2.1 Scheduler

As previously and briefly described, the scheduler is in charge of scatter tasks to workers and collect back results. It supports static scheduling and dynamic scheduling.
Static scheduling rely only on the quantity of workers processes we have instantiated. For this case is performed a block distribution, i.e. having nw workers and a collection of size m, we split it in chunks of size $\Delta = \frac{m}{nw}$. For instance the i-*th* chunk includes $[i \times \Delta, (i+1) \times \Delta[$ tasks. Note that, $\Delta$ must be an integer number, so, in cases where nw is not a divider of m, an operation of ceiling is required.
On the contrary, dynamic scheduling rely only on the specific chunk size ($\Delta$). The chunk size must be defined explicitly by the user to enable and exploit the dynamic policy. At the beginning the scheduler scatter one (or more) task of size $\Delta$ per worker, then, as soon as a worker finished its task, the scheduler dispatch a new task (always of size $\Delta$ or less), if available.
Without doubt, dynamic scheduling has more overhead that the static one, since its an on going process which requires also an higher number of messages exchange between master and worker processes. But it is successful, if used properly, when task are not balanced in terms of execution time or when workers are heterogeneous in computing capabilities (e.g. one worker is running on an high loaded machine or the latter has a less performing hardware respect to the others).
Dynamic scheduling policy requires some idle time for the worker, which is the time needed to ask for a new concurrent activity to be computed (see Figure 4). To solve this problem in our design is sufficient to send at the beginning more than one task to each worker, so that as soon as a task is finished the worker can pop a new task from the local queue[1]. Please note that this method only works when requesting a new task takes less time than executing it, at steady state an idle time can not be avoided. Another consideration to take into account is that this method could damage the load balancing achieved with the dynamic scheduling. A possible fix to this issue is to randomly chose the first sub-task to minimize the probability of send multiple heavy sub-task to the same worker.

[1]The number of task to be sent per worker at the beginning is another parameter that need to be tuned properly depending on chunk size and type of computation.

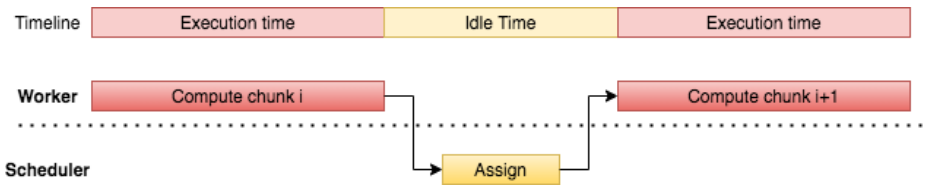


Figure 4: Time-lined representation of idle time of a worker between two consecutive subtask execution, in case of dynamic scheduling.

## 2.2 Worker

The worker stage is very straightforward: it gets from the receiver an homogeneous collection $x$ and apply on it the specified function $f$, at the end send the vector of results $y$ to the sender. Apart from the communications, everything can be summarized as follow:

$$\forall i. y_i = f(x_i) \tag{1}$$

Which can be easily translated into code as :

```
for(i = 0; i < x.size(); i++)
    y[i] = f(x[i]);
```

Since there are no dependencies between every pair of elements of the input collection, the above function can be computed in parallel. So the (1) can be wrapped in a **parallel-for**. Please note, we are introducing here another level of parallelism, we can call it *intra-process parallelism*. Internally the *FastFlow*'s parallel-for is used, configured to exploits the higher parallelism degree available in the execution environment.
As state before, the function to be applied takes an element and, optionally, another read-only data structure which is synchronized between all worker processes. This data structure was presented and called Environment. It useful when the function has some parameters which are not known at compile time.

## 2.3 Sender and Receiver stages

The first stage of the pipeline of each process is a receiver node. It implements a socket[2] based server which supports multiple input connections using a single threaded model exploiting the Unix's `select` System Call. The receiver receives from the network, deserializes the input message and send to the next stage of the pipeline what it has just received.
Instead, the last stage of the pipeline of each process is a sender. It serializes the input data and sends the corresponding message to the destination process. The sender of a process is connected to the receiver of another process through a persistent TCP/IP connection. The sender, also, implements a sleep-retry-loop policy for the connection setup in the start-up phase, which does not force to launch the receiver nodes always before the sender nodes.
The protocol used to communicate over the network is quite simple and use just one type of low-level message which is depicted in the following table:
Both nodes are able to handle and propagate *FastFlow*'s EOS flags for a graceful termination. EOS flags are represented with a massage where size is set to 0.

[2]Sender and receiver support both `AF_INET` and `AF_LOCAL` socket types. Type of socket must be defined at compile time using `REMOTE` or `LOCAL` macros, respectively.

| bool | long | |
|---|---|---|
| isEnv | Size | serialized DATA |

Table 1: The first field is a boolean indicating if the serialized data represents an Environment, then the second field is a long integer representing the size of the serialized data, which is transmitted/received immediately after. All the fields are sent in Network Byte Order.

Serialization and deserialization were entrusted to *Cereal*, which is a header-only C++11 serialization library. It enables to serialize all fundamental types out of the box. For user-defined types (e.g., struct or classes) it is needed to implement a template method called `serialize`[3].An example is listed below:

```
struct ExampleStruct{
    int intParam; float floatParam; vector<string> strVector;

    template <class Archive>
    void serialize( Archive & ar ){
        ar( intParam, floatParam, strVector );
    }
}
```

# 3 Performance model

In this section we outline the performance model of our implementation skeleton distinguishing the case of static scheduling and dynamic scheduling. Since we are evaluating a data parallel pattern, latency (or completion time) is the only metric taken into account. To proceed in this direction, we first introduce the parallel Map cost model that is given by the following formula for the latency[4] ($L$):

$$L = T_{split} + T_{worker} + T_{merge} \tag{2}$$

in which the $T_{worker}$ execution time is usually represented by:

$$T_{worker} = \frac{m \cdot T_s}{nw} \tag{3}$$

where: $m$ is the size of the collection, $T_s$ is the execution time of a single item of the collection, $nw$ is the parallelism degree, $T_{split}$ is the time needed to scatter sub-tasks to workers and $T_{merge}$ is the time needed to collect back results from workers.
We now briefly expand the Equation 2 in the case of static scheduling of our distributed implementation. We have:

$$T_{split} = nw \cdot T_{sendSetup} + \frac{m \cdot sizeof(inputType)}{B} \tag{4}$$

$$T_{worker} = \frac{\frac{m}{nw} \cdot sizeof(inputType)}{B} + \frac{m \cdot T_s}{nw \cdot nt} + \frac{\frac{m}{nw} \cdot sizeof(outputType)}{B} \tag{5}$$

$$T_{merge} = \frac{m \cdot sizeof(outputType)}{B} + nw \cdot T_{receiveSetup} \tag{6}$$

[3]Refer to libray's GitHub page[2] for references.
[4]Expressed in seconds.

where: $T_{scatter}$ is the time to generate sub-tasks, $nw$ is the number of worker processes, $m$ is the size of the collection, $B$ is the maximum available unidirectional bandwidth between any pair of processes (since the interconnection taken into account has flat topology) expressed in bytes/s, $nt$ is the number of thread used within each worker process, $T_s$ is the execution time of a single item, $T_{gatherall}$ is the time needed to merge and return the final result to the user, $T_{sendSetup}$ and $T_{receiveSetup}$ are the times needed to setup a single send and receive operation, respectively.

Now let's see what is the cost model in case of dynamic scheduling policy in the master process. Assuming that we are sending multiple sub-tasks to each worker at the beginning (as described in Section 2.1 while presenting dynamic scheduling) to nullify the idle time and enable communication and sub-task computation overlap for intermediary sub-tasks. The resulting cost model is as follows:

$$T_{split} = nw \cdot T_{sendSetup} + \frac{nw \cdot chunk \cdot sizeof(inputType)}{B} \tag{7}$$

$$T_{work} = \frac{chunk \cdot sizeof(inputType)}{B} + \frac{m \cdot T_s}{nw \cdot nt} + \frac{chunk \cdot sizeof(outputType)}{B} \tag{8}$$

$$T_{merge} = \frac{nw \cdot chunk \cdot sizeof(outputType)}{B} + nw \cdot T_{receiveSetup} \tag{9}$$

where: already defined parameters maintain their definition and *chunk* represents the size of each chunk (also depicted as $\Delta$ during the presentation of the scheduler on Section 2.1). It is worth noting that: for $\Delta = \frac{m}{nw}$ the dynamic scheduling match the static scheduling and for $\Delta > \frac{m}{nw}$ could lead to a static scheduling which do not exploits all the available workers and inevitably results in weaker performances.

# 4 Benchmarks

Tests were carried out in a flat gigabit interconnected cluster of 16 Intel Xeon machines. Each machine has 2 CPU sockets, each CPU has 10 cores 2-way hyper-threading, for a total of 40 processing units. Computing loads are always mimicked through the usage of an active wait function.

## 4.1 Benefits over the shared memory approach

When starting to deal with distributed systems, the first question is: "When a distributed system must be taken into consideration with respect to a multi-core / shared memory system?". It is hard to give a complete answer to this question since more and more powerful multi-core machines are available nowadays. However, from a general viewpoint (leaving aside fault-tolerant requirements), a trivial answer could be: "when all resources of a multi-core machine are saturated". For instance, the optimal parallelism degree of a program is 8 but the architecture has only 4 core contexts, or the program works on a data structure of 16Gb but the available memory is just 8Gb, and a continuous memory swap might degrade the overall performances. Those are cases when a distributed system should be preferred. However, since a distributed system potentially may introduce, as shown in our performance model, high communications overheads, it can be considered when the computation time is (much) larger than the communication time for distributing the data. In our benchmark

test, we are going to give a practical perception of the metrics a distributed system produces and an idea of their orders of magnitude.

In our first test, we assume to have a bunch of machines, all having 10 core contexts. Our benchmark consists of a function that has to be applied to all elements of a collection. We assume that the execution time of the function on a single collection element is 10ms. The number of elements in the collections is 100k. Each worker process exploits all available cores using an internal parallelism degree of 10. The baseline is the shared memory version employing a plain *FastFlow* parallel-for using 10 threads.

We consider two possible scenarios for the distributed memory implementation: one where the size of the collection is small 800kb, i.e., each data element is 8 bytes (e.g. an integer number); the second case where each data element has a size of 10kb (e.g. a complex struct) for a total size of 1Gbyte[5]. Those two cases represent two opposite scenarios. In the first one, the ratio between the execution time and the data size moved is high, representing a case in which using a distributed system should be advantageous. In contrast, in the second case, the ratio is much smaller, representing a case in which the communication overhead may severely impact the benefit of using a distributed system. The completion times are plotted in Figure 5.

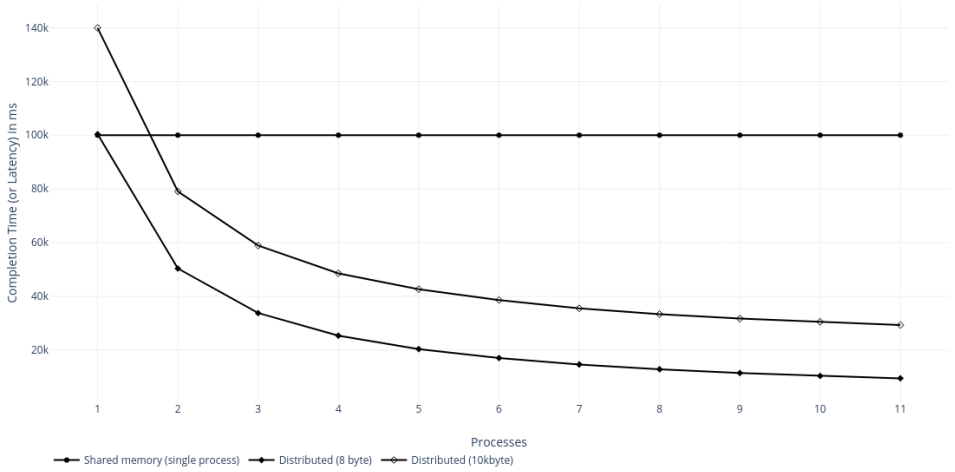


Figure 5: Completion Time vs number of Worker processes. 100k tasks, each one taking 10ms, 10 threads per worker (the same for the baseline shared memory solution)

From the plot can be clearly seen how the communications impact the performance of the execution. The communication overhead is constant with respect to the number of processes and, as shown in Section 3, is directly proportional to the size of the data to be transferred.
A first impression of speedup can be obtained from the results of the previous benchmark. Speedup gives a measure of how good is our parallelization with respect to the "best" sequential computation, which, in our case, is the execution of a plain *FastFlow* parallel-for

[5]In those examples the input size is equal to the output size, i.e., $sizeof(InputType) = sizeof(OutputType)$

in the shared memory domain.
The speedup plot distinctly shows that fine-grained[6] parallelism (i.e., the Distributed 10kbyte solution) should be more efficiently executed in a multi-core environment, which aids in exploiting data locality. From those values of speedup we expect also that the efficiency, which measures the ability of the parallel application in making good usage of the available resources, is very high for the Distributed 8byte version and low for the 10kbyte one. The efficiencies are plotted in Figure 7.

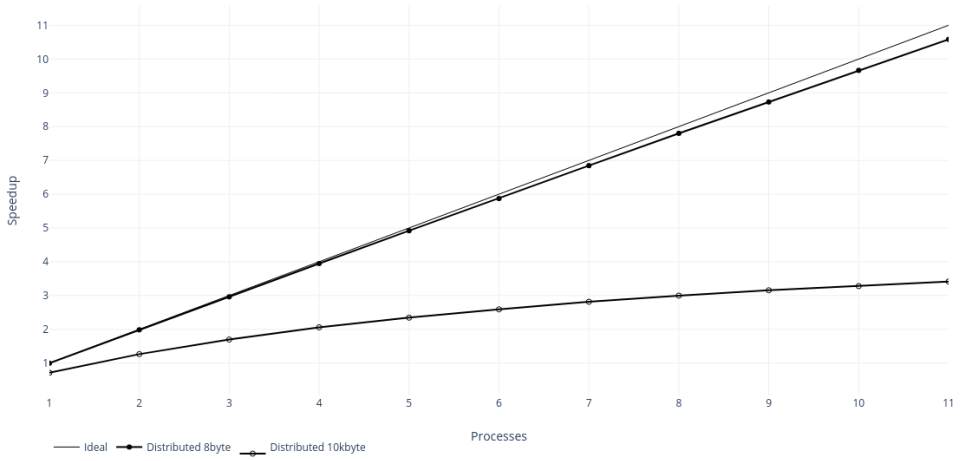


Figure 6: Speedup vs number of worker processes. The ideal speedup is represented by the bisector $speedup(n) = n$

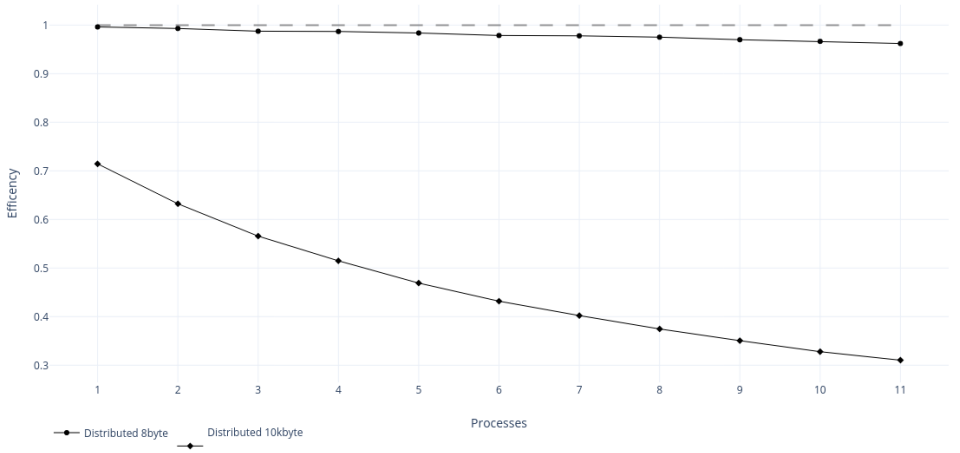


Figure 7: Efficiency vs number of worker processes. The maximal efficiency is represented by the dashed horizontal line $Efficiency(n) = 1$

[6]The granularity of a parallel computation can be express as the ratio between computation time and communication time. $G = \frac{T_{compute}}{T_{communication}}$

### 4.2 Static vs. Dynamic scheduling

This benchmark aims to show what is the difference in terms of performances of the two different scheduling policy implemented.
The first case study is when the whole task in unbalanced, i.e some items are processed very fast, some others need a longer processing time and those types are not uniformly distributed. In this benchmark, it has been generated a collection of 100k elements where the execution time decreases until the half of the collection and then re-increase until the end, and the intra-process parallelism degree is set to 40. The Figure 8 shows the completion time of the static policy and dynamic policy with different configuration of the chunk size $\Delta$ used for the scheduling of sub-tasks.

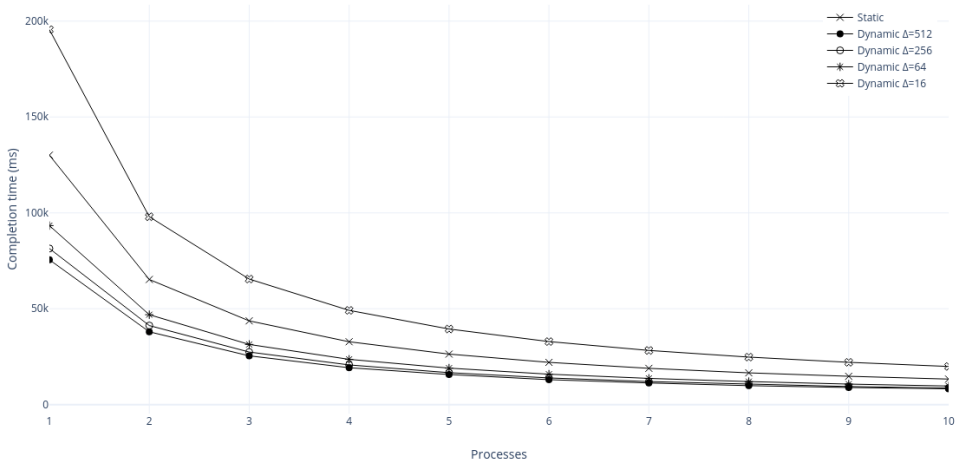


Figure 8: Static policy vs Dynamic policy on an unbalanced collection.

From the benchmark can be seen: (1) the dynamic scheduling is a successful candidate when dealing with unbalanced computations, (2) if the grain of the chunk size is reduced too much the overhead increases and it's not anymore convenient. The chunk size is a critical parameter that need to be tuned to achieve an optimal trade-off between load-balancing and overhead.

The second case study involves situations when all the workers deployed in the cluster do not perform in the same way: because of hardware discrepancies or because there are other processes running in the same machine. To recreate this scenario assume we want to run a balanced task of 100k items each taking 100ms of execution time on a cluster of 8 machines: 4 has support for 4 concurrent thread and 4 has support for 40 concurrent thread. In Figure are shown the completion times of the static policy and dynamic policies with different parameter $\Delta$. This test clearly demonstrate that also in this case the dynamic scheduling has to be preferred. As well, it shows that the best (i.e. minimize the completion time) chunk size $\Delta$ for this kind of computation is around 512 and 256. The validation of the dynamic scheduling is also confirmed by the distribution of sub-tasks: the first 4 workers (less performing) receive only 5 sub-tasks while the other (more performing) receive 55 sub-tasks.

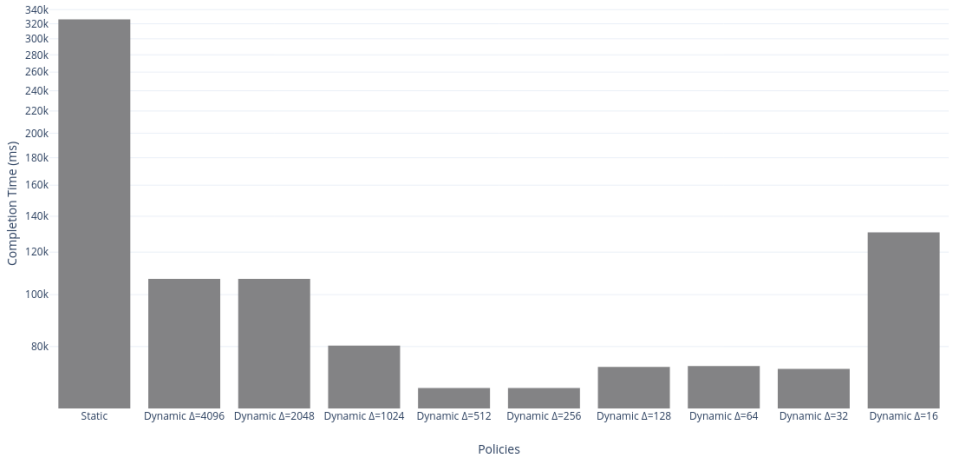


Figure 9: Static policy vs Dynamic policy on unbalanced worker's performances.

# 5 Usage

The skeleton is provided to the user as a C++17 header only library. The dependencies are: *FastFlow*[1] and *Cereal*[2]. A high-level interface to the skeleton is provided by the function DMap::map by simply including the `DMap.hpp` header file. The function takes an `Exec` struct to determine the execution behaviour, a transform function which takes a reference of an item and returns the transformed one, a pair of iterators for the input (start, end), an iterator for the output, a chunk size (0 for static scheduling, dynamic scheduling otherwise), a pointer to a read-only environment, the number of the internal parallelism degree of each worker node. The signature of the function is the following:

```
template<typename Function, typename InputIterator, typename OutputIterator, Env = void>
DMap::map(Exec e,
          TransformFunction f,
          InputIterator start,
          InputIterator end,
          OutputIterator start,
          size_t chunk_size = 0,
          Env* env = nullptr,
          int nt = FF_AUTO
         )
```

where Exec has the following structure:

```
struct Exec {
    bool isMaster;
    std::string masterAddr;
    std::vector<std::string> workers_addrs;
}
```

It is already builtin a argument parser to populate the Exec struct as a constructor of Exec itself. The guideline for the positional argument to be given to processes are:

- Master Process: `exec true <listen_addr> <worker1_addr> ... <workerN_addr>`

- Worker Process: `exec false <listen_addr> <master_addr>`

In the examples folder there are two naive "algorithms" implemented using this micro framework. A translator (`translator.cpp`) along essentially like the one seen in class during the course: which read from disk a text and transform it all to uppercase. And a sample of matrix per vector multiplication (`mvMultiplication.cpp`).

In addition, some scripts are available to run the solutions on the cluster[7], for example `runExec.sh` which automatically start remote processes with the right parameters (e.g. addresses and ports), `runExec_local.sh` which execute the solution locally using `AF_LOCAL` sockets, `killProcessByName.sh` which kill a process by name in all the machine of the cluster. By default the makefile compile targets to support `AF_INET` sockets. To compile for `AF_LOCAL` socket you may just define the `LOCAL` macro in the makefile or compile using: `make LOCAL=1 <target>`. Please have a look to the `README.md` file to see an example of compilation.

# 6 Conclusion & Possible improvements

We have proposed a micro framework implementing the Map pattern suitable for targeting clusters of multi-core workstations. Using a small set of benchmarks, we have demonstrated that the solution proposed succeeds in exploiting distributed cluster resources providing good efficiency for coarse-grained computations. Moreover, we described the benefits obtained by the dynamic scheduling policy in different scenarios.
Some optimizations can be introduced to our implementation, including but not limited to:

- introducing support for multiple Maps enabling the possibility to express iterative data-parallel algorithms (e.g., a Map computation within a loop);
- introducing facilities to support Map taking multiple input sequences (n-ary Map);
- transforming the worker pipeline in a sequential combination (using `ff_comb`) in cases of static scheduling, since just one item will flow in that pipe.
- enabling the users to implements their serialization and de-serialization policies to avoid using third party libraries and achieve better performance for data structures which are contiguous in memory. In this way some copies during communications can be avoided.

# References


[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: high-level and efficient streaming on multi-core. *Programming multi-core and many-core computing systems, parallel and distributed computing*, 2017.

[2] W Shane Grant and Randolph Voorhies. cereal–a c++ 11 library for serialization. *URL https://github. com/USCiLab/cereal*, 2013.

[3] Marco Danelutto and Massimo Torquati. A risc building block set for structured parallel programming. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 46–50. IEEE, 2013.


[7] Actually the script is written for running on *openhpc2* cluster hosted by University of Pisa.