# A Futures Programming Model Runtime System for Distributed Memory

Dimitrios Chasapis
Laboratoire de Recherche en Informatique
Parall Group

June 3, 2013

# Contents

# Chapter 4

# Extra Features

## 4.1 Additions to the User Interface

Up to this point, the interface we have described includes only the very basic routines of the std futures interface. Other futures interfaces, such as HPX and boost have enriched their interfaces with additional routines. One routine we found to be useful, is the *make_future* routine. In our library, the *make_future* routine is used to create a future variable and initialize it with a value. This is usefull in cases we would like to have a future value but we already know the value it should hold, while we would like to use this variable a later point of the code as an actual future.

Another addition is the *async_on* function, which can be used if the user wishes to schedule a job on a process of preferance instead of using the librarys scheduler. This function is useful when the user needs to schedule his jobs on specific nodes, so that he can take advantage of data locality and/or avoid transfering data through the network.

## 4.2 Future Serialization

An important addition to our futures library, is the serialization of a futures object. By serializing a future object, we can practically pass it as an argument the async function. This is important, because this way a future created on one process, can be transfered to another one, thus synchronization can take place on the worker process, which allows finer granularity when synchronizing taks. Consider the tiled LU in **??**. In this example, all the future objects are created on the master and all synchronization takes also place on the master. It would be better if the process that will run the

functor object issued with an async, for instance dgessm, will wait for its tiles instead of the master. This can be achieved by serializing the future objects related with the tiles that are needed as input to dgessm.

In our implementation, a future object is defined as shown in **??**. *ready_status* is the current status of the future which is true if the value has been set or false otherwise. *Data*, is a local storage for the the future's value, once the future has been set by the remote process. The variables *src_id* and *dst_id* hold the id value of the owner of the future and the process that will set the future value respectively. The *data_ptr* and *status_ptr* variables are *Shared_pointers* (see section **??**), which hold all the information needed for the owner of the future to retrieve the data and the future status from his shared address space. Finally, *data_size* amd *type_size* are the number of elements and the type size of the data the future wraps around. A future object can be trivially serialized by serializing each of its member using the boost serialization library. One however must be aware that when a new future is created the Shared Memory Allocator module will first allocate the memory needed for the future's data and status in the shared memory segment of the original future owner. This is done for performance reasons, since accessing local variables costs considerably less than accessing remote ones, and the way the get method is implemented requires regular polling on the emphstatus_ptr variable. However, when we serialize a *Shared_pointer* variable, it will still point on the same memory, on the original owner. This implies, that the new owner will have to access the *data_ptr* and *status_ptr* using the underlying communication library of our implementation. The reason for this limitation is that once the future is created there is no way to alert the process that will run the asynchronous job and that will set the future's value, that that it should update its own copy *Shared_pointer* variable using the current asynchronous communication scheme our library uses. In practive, the original owner of the future, will act as a proxy between the new future owner and the process that will set its value. Also note that it is possible for the user to retrieve still retrieve the future value from the original owner, or have it sent to multiple processes, since the actual data everyone will access will always reside on the same place for everyone.

# Bibliography