Implementing Distributed Futures using MPI-2's one-sided commuinication

Dimitrios Chasapis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Knossou Av., P.O. Box 2208, Heraklion, GR-71409, Greece

Thesis Advisors : Prof. *Joel*, Dr. *Falcou*

Optionally: This work was partially supported by **fill in sponsor name**

University of Crete
Computer Science Department

Implementing Distributed Futures using MPI-2's one-sided commuinication

Thesis submitted by
Dimitrios Chasapis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author : _____
Dimitrios Chasapis

Committee approvals : _____
Joel Falcou, Assistant Professor, Thesis Supervisor
Informatique
Universite Paris-Sud XI

_____
Angelos Bilas, Professor
Computer Science Department
University of Crete

_____
Dimitrios S. Nikolopoulos, Professor
School of Electronics,
Electrical Engineering and Computer Science
Queen's University of Belfast

Departmental approval : _____
Yannis Manousakis
Professor, Director of Graduate Studies

Paris, February 2013

Abstract

In this work ...

Περίληψη

Στην εργασία αυτή ...

Acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We present an implementation of the future programming model for distributed memory, using MPI-2's one-sided communication.

1.1 Motivation

Parallel computing has been mpla mpla mpla. The two most dominant and widely used programming models are threads and message passing. Threads are used on shared memory machines and require locking (error prone) while message passing can be used on either shared or distributed memory. Describe message passing, say its tough, talk about one sided communication, refer to ARMCI, ARMI, Charm++, Global address space languages, MPI-2 etc. Say that mpi is widely used and is implemented on most machines, thus we want to check out its one sided interface, which has not received acceptance due to (?) (check all that stuff I've read about how tought it is, mpla mpla). Also need to mention the future interface, and why we use it. Ease o programming, exposing irregular patterns(?).

Note:global arrays guys have already made arguments about one sided comm of mpi-2

1.2 Background

1.2.1 Futures

Background on futures, mention languages that implement it as well as std and boost in C++. Example code and explanation.

1.2.2 MPI one-sided communication

Maybe mention again thins from intro. Explain the interface (windows, epochs, put, get). Maybe a small example.

1.3 Related Work

RMI, RMC, HPX, STAPL's comm library

1.4 Related Work

Chapter 2

Design and Implementation

We have implemented the distributed futures to be modeled, as closely as possible, to the shared memory futures from the C++ standard library. Parallelism in the future model is extracted by issuing functions asynchronously, using a special async funtion that takes the function to be executed in parallel as an argument (see section 2.1). In our library, functions called using the async function, refered as *jobs* from now on, are send to be executed by other avaible processes. The decision of who executes a *job* is made by a scheduler. The host process can retrieve the return value of a job using the future object associated with the corresponding async call. A process can only retrieve a future's value if the job has finished execution, else it will block. In our implementation, the blocked process will try to execute any pending *jobs* it might have while it is beeing blocked. In our design, different aspects of the runtime library, such as process communication, hide its underlying implementation (e.g. MPI), thus achieving a modular design. Our system consists out of three main modules: The communication module, which is the backbone of the system and used by all other components in order to exchange messages and create a shared address space. The Shared Memory Manager, which is an allocator for the shared address space between the processes. The Scheduler, which is responsible of handling how *jobs* are send/received between processes and also decides who will run which process.

Figure /ref shows the program flow of the simple hello world example in figure 2.1. Before any call to the library is made, the futures environment must be initialized, which in turn initializes all other library modules (e.g. communication, scheduler, memory manager). All processes execute the main function, but only the master process will return from it and continue with the user program execution. All other processes will run our runtime's scheduler code and wait to receive jobs, wich are issued by a call to the async function. The async function can be called from any process and within other async calls, thus allowing recursive algorithms to be expressed.

In the rest of this chapter, we will present the future interface and discuss our specific underlying implementations of the the three core modules.

```
1  class helloWorld {
2  public:
3    helloWorld() {};
4    ~helloWorld() {};
5    int operator()() {
6      int id = Futures_Id();
7      cout << "- Worker" << id << ":Hello Master" << endl;
8      return id;
9    };
10 };
11
12 FUTURES_SERIALIZE_CLASS(helloWorld);
13 FUTURES_EXPORT_FUNCTOR((async_function<helloWorld>));
14
15 int main(int argc, char* argv[]) {
16   Futures_Initialize(argc, argv);
17   helloWorld f;
18   future<int> message = async(f);
19
20   cout << "- Master :Hello " << message.get() << endl;
21
22   Futures_Finalize();
23 };
```

Figure 2.1: A simple hello world implementation using the distributed futures interface. The output of the program on process 0 would be "- Master :Hello 1".

2.1 Futures Interface

We replicate the futures interface from the C++ std::future library, with the only difference being that the the function being called must be a functor object. Figure 2.2 shows a recursive implementation of the fibonacci function using our future library. The user needs to create a functor, which must be serializable*(footnote here about boost::serialization), and use the macro FUTURES_EXPORT_FUNCTOR(async_fu int>) to expose the functor object to the serialization library. Moreover, the user can use the FUTURES_SERIALIZE(F) macro, where F is a functor object, which will create the necessary serialization routines automatically, but is not recommended if the functor object has members (see BOOST serialization for more details). Note that the argument to the macro command is always async_function<F, Args...>, where F is functor class and Args are the argument types, of any arbitary number, that are required by the overloaded call method of the functor F. A call to the async(F, Args...) function, where F is a functor object and Args is any number of arguments, will send the functor object to an available process or execute the

functor directly, if no such process is found (see SECTION for details). The async function returns a future object which can be used by the process that called the async function to retrieve the functor's return value. If the return value is an array, a pointer or any other form of container, the user should instead call a variation of the async function, async(N, F, Args...), where N is number of elements that will be returned. In order to retrieve the value, the owner of the future needs to call the get() method. This method is blocking, so calling it will cause the process to block until the value of the future becomes available. Alternatively, the future owner can call the is_ready() method, which is not blocking, to check if the value can be retrieved.

```
1  class fib {
2  public:
3    fib() {};
4    ~fib() {};
5    int operator()(int n) {
6      if(n == 0) return 0;
7      if(n == 1) return 1;
8      fib f;
9      future<int> fib1 = async(f, n-1);
10     future<int> fib2 = async(f, n-2);
11     return fib1.get() + fib2.get();;
12   };
13 };
14
15 FUTURES_SERIALIZE_CLASS(fib);
16 FUTURES_EXPORT_FUNCTOR((async_function<fib, int>));
```

Figure 2.2: A fibonacci implementation using the distributed futures interface

2.2 Communication

The communication module is responsible for message exchange between all the process in our runtime system, as well as providing a shared address space. In our implementation the communication module uses MPI-2'S one-sided communication libary and the boost.mpi's input and output archives, for object serialization.

The communication module acts as a layer of abstraction between our system and the MPI libary. It acts as a simple wrapper for initializing, finalizing MPI and simple send/receive operations. It is also capable of providing information of the MPI environment to the other components of our system (e.g. number of processs, rank e.t.c.). Moreover, it can be used to expose part of a process' address space to other processes.

2.2.1 Shared Address Space

In our implementation, the underlying message passing library used is MPI-2, thus we use MPI windows to expose such space. Exposing part of process' address space in the MPI-2 schema, requires that the needed space will be locally allocated to a pointer using the MPI_Alloc_mem, and then exposed to other processes through creating a window that is related to the pointer with MPI_Create_Win (See section 1.2.1). A drawback in MPI is that a window can be created only collectively over a group of processes, and in turn, a group can be created, again, only collectively over a paretnt group. This requires either all windows to be created a priori at initialization, since our system requires that when issuing a job, only the sender and receiver should take part in the communication. In order to overcome this limitation, we used the algorithm presented in **??**, which enables us to create a process group only between the processes that will be part of that group. This way we can dynamically allocate windows between any two processes, if needed. NOTE:say about epochs here?

The communication library also provides the routines needed to write and read data from a shared address space, using the special Shared_pointer construct (see section 2.2.2). This pointer keeps information of where the data is located within an MPI window in addition to the total size of the data associated with this pointer during its allocation. Figure /ref shows how shared address space is structured and accessed. Note that at the end of allocated space related with a pointer, there is a field that contains the size of the data stored in the particular space. This size, is the size of the archive class used to serialize a functor or any other C++ serializable object. Thus the starting location of the data is computed by ... (more details here, check implementation to refresh my memory). (Why two different sizes? explain)

2.2.2 Mutexes

In order to synchronize accesses to shared memory addresses and other critical sections in our system, designed a mutex library, whith the same interface as the standard C++ mutex library, which is implemented for shared memory. The only difference is that a call to lock, unlock or try_lock requires the user to specify the id of the tareget process. We have adopted MPICH's implementation of mutexes in our design. A mutex is a shared vector through an MPI window. Each vector element is a byte value corresponding to one process. When a process wants to hold the mutex lock, it sets its vector value at one and iterates through the rest of the vetor to check if another process wants or has acquired the lock. If the lock is acquired or another process waits for it, then the current process blocks until it receives a message. When unlocking, a process sets its vector value to zero and then iterates through the vector to find and send a message to next process that is waiting to acquire the lock.

2.3 Memory allocation

The Memory Manager module manages shared memory. It uses the communication module to create address spaces that are visible by all processes in our system and use the Shared_pointer construct to describe a location in such shared memory. This modules provides the functionallity of allocating and freeing space, from the shared address space among all processes. Our allocator is implemented using free lists in order to track free space as described in /refModern Operating Systems. However, we keep different free lists for different page sizes to deal with memory segmentation. Figure /ref shows how memory is organized. The shared address space is allocated a priori using the communication module, to create MPI windows in our current implementation. This is ofcourse transparent to the Memory Manager module, since it uses Shared_pointers to describe memory location, size etc. The Shared_pointer is a tuple ptr<ID, BA, SZ, PSZ, PN, ASZ>, where id is the id of the process whose address space we want to address, BA is the base address that the data is located in a shared address space, SZ is the size of the data we want to allocate, PSZ is the page size the allocator used to allocate for this data, PN is the number of pages used and ASZ is the the actual size, which is PN*PSZ. So, each freeList in figure /ref is actually a list of Shared_pointers and a corresponding MPI window. We choose to keep seperate windows for each free list because when locking an epoch access to an MPI window, the whole window is locked, since MPI cannot lock only part of it (see section 1.2.1).

When a process issues an async function, it needs to allocate space in its shared address space, for the worker process to store the future's value. To allocate such space, the host process uses the shared memory manager. The shared memory allocator tries to find the best page size fit for the data size, and searches the corresponding free list, using a first fit algorithm to find a large enough space for the new data. If no fitting page size is found then the allocator uses a special freeList, which does not use a predefined page size, but instead uses the data size to find free space. If not enough free space is found in the correct free list, then the allocator can try to find data in another free list, of different page size. Figure /ref shows a free list, before and after allocating an object of X size. The first fit algorithm will iterate the list from the start untill it finds a large enough space for the object. Each node in the free list, is a Shared_pointer, which describes how much continueous space there is avaible. When the allocator finds a large enough node, it removes from that node the size and number of pages it needs and sets it base address value accoringly. It then returns a new Shared_pointer, that describes the memory space that will be now occupied from the data object. In the example in figure /ref, the base address will be ...

As soon as a process retrieves a future value, it makes a local copy of it, and frees any shared address space that is associated with the future. In order to free shared space, a process needs to provide the Shared_pointer that was returned by the allocator routine. The Shared_pointer keeps information of the page size used to allocate space, thus finding the correct free list is trivial, we just need to use the

page size as an index. We then insert the Shared_pointer in the free list in a sorted fushion, using the base address for comparison. This way, all free lists are sorted lists of Shared_pointers by base address, so that if we find continueous space, we merge the list elements, resulting in larger block of free space. Figure /ref shows a free list, before and after freeing some shared memory.

(mention: Shared_pointer needs to be serializable)

2.4 Scheduler

In order to have a distributed memory interface similar to the shared memory one, we chose to implement a scheduler, which is responsible for deciding who will execute which *job*. If the user was responsible for distributing *jobs* among the processes, he would need to reason about dependencies between *jobs* and retrieving future values, else the program could easily end up in a deadlock. To make our case clear, consider the fibonacci example in figure 2.2. In our example, let's say we have 3 processes, one of them is the master process. We need to run fib(3), thus, process 0, the master process can issue async(f, 2) to process 1 and async(f, 1) to process 2. Process 1 can issue async(f, 1) to process 2 and run async(f, 0) on itself. In this scenario the program will execute correctly without any problems, since when any of the processes call a get() will either retrieve or wait for the value. But consider we want to compute fib(5). Process 1 may have to run async(f, 4) while process 2 will have to run async(f, 3). At some point, process 1 issues async(f, 3) to process 2, while process 2 issues an async(f, 2) to process 1. Both processes will return from the async calls and proceed calling get() to retrieve the value but will actually block forever, since neither process will run the pending fibonacci functions. This scenario is not a problem if processes are dynamically spawned, but if we have a static number of process, which is common for mpi programs, we need to address such issues.

Since it is not always trivial to reason about such dependencies, we have implemented our own *job* scheduler. We use MPI-2's one-sided communication library (via the communication module) to implement task stacks, similar to their shared memory counterparts. We choose to implement a stack because it suits better future logic, we need to execute the latest issued *job* in order for the get() not to block indefinitely in recursive algorithms. Using one-sided communication, only the issuer needs to copy the functor object to the workers stack, as in a shared memory environment. Figure /ref, show how a tasks stack is structured. Note that an entry is composed by the functor object and its arguments (they are considered on object) and the size of it. This is neccessary since different funtors and/or different arguments result in varying entry sizes. Thus, the exact location of a task is calculated using the stack head and functor object size values [1].

Figure /ref shows a control flow graph for the master and worker processes.

1. functors and arguments are send/received as outpur/input archives, using boost.serialize library.

The master simply initializes the futures environment and issues async functions while executing user code. At the end it finalizes the futures environment and calls the terminate routine from the scheduler. The workers initialize the futures environment, which must happen collectively among all workers and master and then enter a loop, looking for pending jobs in their queues until ther terminate routine returns true, in which case they exit the loop, finalize again collectively with all other processes and exit the program, without ever returning to the main function. The scheduler is responsible for providing the functionallity of the terminate routines. In our implementation the workers poll a local variable which they expose through the communication module as a shared variable. The master, when calling his terminate routine, will check the status of every worker. A process can be either idle, busy or terminated. Process status is again exposed by a shared variable on each process. The master will check the status of all the processes and if all of them are in idle status, he will set the terminated flag to true on all of them. If a process is still busy, meaning executes some *job* or has still pending *jobs* in its stack, the master must wait till all jobs are finished and then set the terminated flags.

When running user code or a *job* and an async call is made, the process will address the scheduler in order to get the id of the next avaible process and allocates enough space for the return value to be stored. Then it asks the scheduler to send the job to the worker process. In our implementation the scheduler pushes the job the processes stack. Our scheduler distributes *jobs* in a round robin fashion (excluding the master process, which should run user code).

Alternatives - Why we did this

Chapter 3

Conclusions and Future Work