A Unified Futures Interface for Shared and Distributed Memory

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

and

Université Paris Sud
Faculté des Sciences
Département d'Informatique
Campus d'Orsay, FR-91405, France

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

University of Paris-Sud XI
University of Crete
Computer Science Department

A Unified Futures Interface in C++ for Shared and Distributed Memory

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

THESIS APPROVAL

Author : [illegible]
Dimitrios Chasapis

Committee approvals : [illegible]
Joel Falcou, Assistant Professor, Thesis Supervisor
Computer Science Department
University of Paris-Sud XI

[illegible]
Lila Boukhatem, Associate Professor
Computer Science Department
University of Paris-Sud XI

[illegible]
Daniel Etiemble, Professor
Computer Science Department
University of Paris-Sud XI

Paris, June 2013

Abstract

In this work we present a C++11 library implementation of the futures programming model for distributed memory. Our implementation uses an interface similar to the C++11 standard library's one. The user can use the futures interface to express parallelism and synchronize his code, while the underlying runtime system schedules the functions the user issues to be run in parallel. Our runtime currently uses the MPI one-sided communication interface, to achieve asynchronous communication. We evaluate our runtime's performance and conclude that, in it's current state, it is only suitable for handling coarse grain tasks. We also share our experience using the MPI one-sided communication interface for implementing a high-performance runtime.

Περίληψη

Σε αυτή την εργασία παρουσιάζουμε την υλοποίηση μιας βιβλιοθήκης C++ του προγραμματιστικού μοντέλου των φυτυρες για Distributed Memory περιβάλλοντα. Η υλοποίησή μας χρησιμοποιεί ένα ιντερφαςε παρόμοιο με αυτό της C++ standard library. Ο χρήστης μπορεί να χρησιμοποιήσει το interface των futures για να εκφράσει παραλληλισμό και να συγχρονίσει την εφαρμογή του, ενώ το υποκείμενο σύστημα runtime μας είναι υπεύθυνο για τον καταμερισμό της εργασίας και συνχρονισμό των διαφορετικων διεργασιών. Το συστημά μας βασίζεται στην one-sided communication βιβλιοθήκη του MPI, για την επίτευξη ασύγχρονης επικοινωνίας. Αξιολογώντας τις επιδόσεις του runtime μας, καταλήγουμε στο συμπέρασμα ότι η τρέχουσα υλοποίηση είναι κατάλληλη μόνο για τον χειρισμό coarse-grain εργασιών. Επιπλέον, αξιολογούμε την χρηστικότητα του MPI one-sided communication interface χρησιμοποιώντας το για την υλοποίηση ενός runtime συστήματος υψηλών επιδόσεων.

Acknowledgements

I want to thank my supervisor of this Master Thesis, prof. Joel Falcou for his support and guidance and of-course for his great sense of humor. It has been a greate experience working with him. I also want to thank my other academic supervisors during this Master course, prof. Angelos Bilas and Dimitrios S. Nikolopoulos along with Polyvios Pratikakis. They introduced me to parallel computing and languages and their trust and guidance has inspired and enabled me to complete this work. I also want to thank Antoine Tran Tan for his help and cooperation during the time I have been working on this Thesis. Finally, I want to express my sincere thanks and love to my family, all my flatmates, in Greece and France alike, as well as my dearest friends for their support.

Contents

List of Figures

Introduction

We present an implementation of the future programming model for distributed memory, using MPI-2's one-sided communication. The interface is implemented as a runtime library that allows the user to expose parallelism, by issuing callable functor objects asynchronously. The future object is used like a simple communication channel, where the worker process will send through it the return value of the functor object. A future object is also used for synchronization. Such an object can be accessed at any time during execution. The process accessing it will block until the worker process finishes the execution of the functor object, and transmits the result to the future object.

Traditionally futures are implemented using threads in shared memory environments. In this work we show that the C++11 standard future interface [1] can be implemented meaningfully for distributed memory machines. We have chosen to build our system using the MPI-2 one-sided communication library, so that we can explore and evaluate it's potential to provide a completely asynchronous communication scheme. Another reason for using an MPI library is that it is the most commonly message passing library available on distributed and shared memory machines alike. The contributions of this work can sum up to:

- Implementation of a unified C++ futures interface for both shared and distributed memory machines, as a runtime system.
- Performance Evaluation of the our implementation.
- Evaluation of the MPI-2 one-sided communication interface, for implementing an advanced runtime system.
- Exploration of the potential of implementing a runtime on distributed memory using shared memory scheduling techniques.

Our evaluation shows that the interface implementation is possible, but, performance-wise, our implementation is only able to offer some speedup only when we use coarse grain tasks, due to the high cost of issuing functions asynchronously and/or inefficient synchronization schemes. Moreover, MPI-2 one-sided communication interface is not as versatile as we would like, especially regarding fine grain synchronization.

The rest of this report is organized as follows: In chapter 1, we present the current state and trends in parallel computing. We briefly introduce the concept of asynchronous execution models and present the futures programming model. At the end of the introduction, we describe MPI's one-sided communication interface. In chapter 2 we discuss other projects related with our work, and present their approach to asynchronous communication. In chapter 3 we present in details our system's design and implementation. In chapter 4 we assess our efforts in building the C++11 interface using MPI's one-sided communication and use some microbenchmarks and real applications to evaluate the performance of our runtime system. Finally, in chapter 5 we give our concluding remarks regarding our library along with our suggestions for its improvement.

Chapter 1

Background

1.1 Parallel Computing

High performance computing is today strongly related with parallel programming. On one end, computer architectures have been developing parallel machines or network configurations for clusters of machines in order to increase performance, and on the other, researchers have been trying to develop programming models that will allow programmers to develop or port efficiently their applications to these emerging technologies. When developing parallel applications the two most dominant and widely used programming models are threads and message passing.

The threads model is commonly used on shared memory machines, where the communication scheme would have one thread writing to a memory location and another thread reading the data from that location. This model does not require data to be transferred among threads but can lead to race conditions when two threads try to access the same data at the same time, if a thread does not respect RAW and WAR dependencies. In order to ensure correct program execution, the user must synchronize memory access by the threads using mutexes, semaphores, locks, barriers etc. Correct synchronization has proven to be a daunting and error-prone task for programmers, and often synchronization bugs in application can be the cause for erroneous results, or even worse, deadlocks. Pthreads and OpenMP [2] are two commonly used libraries that are used to program threads on shared memory machines. With Pthreads the user can create and launch threads, where each thread will have a specific work to do. The library also offers a variety of synchronization primitives such as locks, barriers, mutexes etc. OpenMP offers a higher abstraction level interface, where the programmer uses special #pragmas to annotate code sections that should be executed in parallel. These pragmas can denote loops that should be run in parallel or even organize parallel work into tasks [3], while the library takes care of creating and launching threads. However, the user is again responsible for synchronizing data accesses.

In contrast with the threads model, applications using the message passing model, use messages to share data between different processes and also for synchronization. The usual scheme requires a matching pair of send and receive operations where both application will have to eventually block at some point until the message has been received. Although the message passing model is considered more difficult to program than programming with threads on shared memory, it is easier to reason about data locality, thus message passing can potentially achieve very good performance. Moreover, message passing libraries are usually the only available option on large scale distributed machines, where different physical nodes do not share a global address space. A drawback of most message passing implementations is that two-sided communication is required when exchanging messages. This means that both sender and receiver must take active part in the communication, which usually means that they both need to block at some point, until the message is sent/received.

An alternative from the usual message passing two-sided communication model, is the one-sided communication model, where one process can remotely write or read from the address space of another process, while the latter is not required to take active part in the transaction. ARMCI [4] LAPI [5] and MPI-2 provide library implementations of such one-sided communication interface. OpenSHMEM [6] is an effort to standardize the SHMEM model. An attractive property of this model, is that communication can happen asynchronously, which also means however that the programmer needs to explicitly synchronize processes as in the shared memory model, using barriers and fences.

The emergence of the one-sided communication model has made it possible to develop libraries and languages that follow the PGAS (Partitioned Global Address Space) programming model. In this model, a virtual global address space is exposed to the programmer, when in fact, this address space is distributed among the different nodes or a logical partition dedicated to a single thread. This model tries again to exploit the benefits of the message passing's SIMD model while providing an easy way to address data as in the shared memory models. UPC, Chapel and Fortress are languages that use the PGAS model and are built on top of an one-sided communication library. Global Arrays [7] is also an API that follows the PGAS model and is built on top on ARMCI [4].

Because all of the previous models are either considered difficult to program or error prone, a lot of higher level programming models have been suggested in the literature, that are implemented on top of one of the previous, lower level, ones. The concept of organizing parallel work in functions that can be run concurrently has lead to the development of many task-based programming models [3, 8] in the shared memory environment and to similar models in distributed memory like Remote Procedure Calls (RPC) [9, 10, 11] or Remote Service Request (RSR)

[12]. Although this higher level abstraction makes it easier to organize parallel code in tasks, it is still up to the programmer to explicitly synchronize data accesses between tasks, using barriers, etc. To address and simplify synchronization problems, a lot of systems have been suggested in the literature, that provide implicit synchronization. In the scope of task based parallel models, these systems usually require some sort of task memory footprint description from the programmer [13, 14] and/or have the compiler statically infer dependencies among tasks [15, 16]. This scheme usually allows the programmer to describe the data-flow relations between different parallel tasks, while an underlying runtime systems will explicitly synchronize them. The drawback here is that there is usually additional overhead from the runtime system . Moreover, the automatic (dynamic or static) analysis used to automatically synchronize the code, is often conservative in order to maintain correctness, which harms performance.

1.2 Futures and Promises

Experience with parallel programming has shown that common synchronization techniques like barriers do not scale well on massively parallelization machines [17], with thousands of workers. One would like to use finer grain synchronization, but reasoning about the exact point an operation will complete is virtually impossible in a parallel environment. An alternative is to use asynchronous programming models, which allows the programmer to write programs where a thread or process can be oblivious to what actions the other threads/ processes are doing. However, he should still be able to retrieve the results of concurrent work and produce the correct result.

The futures (or promises) model is such an asynchronous programming model. A future is a special variable which may or may not have a value at the time that it is referenced in program. Usually a future is coupled with a promise. A promise is a special construct that is associated with a future and can be used by another thread or process to set the value of the future variable. Usually, the future is used only to read the variable value, while the promise is used to write to the same variable, thus defining a data-flow relation between different threads/processes. The promise construct is often hidden from the programmer. Instead he will have to declare a callable object (function, functor, etc). The library will offer a mechanism to use this callable object to set the future through the promise, after executing the user's callable object. Such is the use of the *async* function in the C++11 standard, where the user can issue a function or functor object and retrieve a future object using the *async* call. The *async* will be run by a thread, and the return value of the function or functor will be used to set the future object associated with that*async* call.

An important design decision for any futures implementation, is what happens when a future is referenced, while its value is not yet available. A common choice,

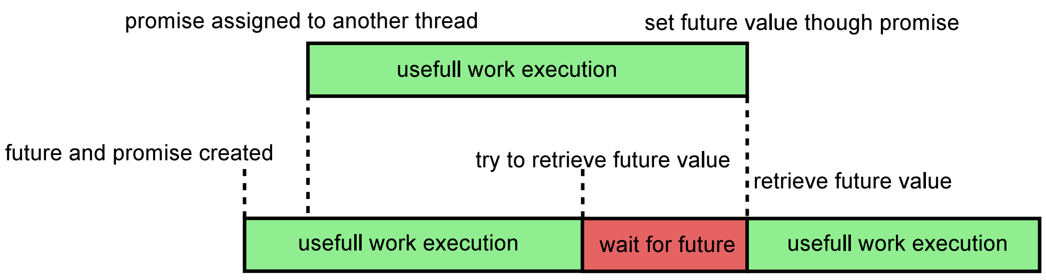
promise assigned to another thread
set future value though promise
usefull work execution
future and promise created
try to retrieve future value
retrieve future value
usefull work execution
wait for future
usefull work execution

Figure 1.1: The futures execution model of the blocking schematics.

is to have the caller block until the future value is resolved or implicitly try to resolve the future at the reference time (as with lazy evaluation schemes). Figure 1.1 shows the execution model of the blocking scheme. The green color is the time a thread spends doing useful computation, while the red color is the idle time a thread spends on waiting for the result of the future. This is the scheme used by C++11, an alternative is the Scala future implementation [18], where the user can set a callable object to be called when the future will be set, or if the future throws an exception (failure), using the callback mechanism. This scheme has the benefit that there will be no blocking at any point of the code, allowing true asynchronous execution. The C++11 standard, as most blocking future implementations, offer the option to ask whether a future is ready before referencing its value, in order to avoid any blocking if possible.

```
int fibonacci(int n) {
  if(n == 0) return 0;
  if(n == 1) return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

Figure 1.2: A sequential fibonacci implementation

```
int fibonacci(int n) {
  if(n == 0) return 0;
  if(n == 1) return 1;
  future<int> fib1 = async(fibonacci, n-1);
  future<int> fib2 = async(fibonacci, n-2);
  return fib1.get() + fib2.get();
}
```

Figure 1.3: A fibonacci implementation using the C++11 futures interface

Other than their asynchronous execution model, we believe that futures offer an easily programmable and expressive user interface. As a motivation to the reader, we present in figure 1.3 a Fibonacci function implementation, using the C++11 standard threads library [1] future interface. Figure 1.2 also shows the sequential equivalent. The parallel version simply requires the recursive calls to be issued using the *async* function, and the use of the get method on the future objects in order to retrieve the return values, of the recursive calls. Note, that the call to the get method here is blocking.

1.3 MPI one-sided communication

One of the most controversial features of MPI-2 is it's one-sided communication. Although PGAS programming models and languages have become widely accepted for developing code in large scale machines, programmers consider the MPI one-sided communication interface to be generally difficult to understand and use. In this section we try to familiarize the reader with the main concepts of the interface.

In order to perform remote access operations on some data, this data, residing on one process, needs to be exposed to the other processes, through an `MPI_Window` object. Thus, all processes need to create an `MPI_Window` that will expose part of their local address space to all other processes. `MPI_Windows` are created using the `MPI_Win_create` function. Figure 1.4 shows how a window can be created on line 13. This functions requires a pointer to a local address space, that was allocated with `MPI_Alloc_mem`. The rest of the arguments to `MPI_Win_create` are the number of elements and type size of the data to be shared, an MPI info flag, an MPI communicator and the window. This is a collective operation over a group of MPI processes. Each process can expose different size of data (or none) to the window. Note that only the processes in the group will be able to perform a remote operation on the created `MPI_Window`.

The two main operations that can be performed are `MPI_Put` and `MPI_Get`, which allow a process to remotely write and read some data respectively. In the example in figure 1.4 there are multiple calls to both `MPI_Put` and `MPI_Get`. A buffer that data will be written from or to has to be provided. The size of the local buffer must be defined as well as its MPI datatype. The same must be done for the remote buffer on the process on the other end of the communication. In order to target the correct remote buffer, the id of the target process and the window related to that buffer must also be provided. Another operation available is the `MPI_Accumulate`, that can be used to apply some action on the data that is remotely read and the local data on the process. An operation must also be supplied to this function.

In contrast to the two-sided communication interface, in the one-sided interface, get and put operation need not be paired and non-blocking. Synchronizing processes that perform these remote operations must be explicitly done by the programmer. Synchronization in the MPI one-sided communication interface is achieved using "epochs", that define the start and end of an operation. All one-sided operations must happen in one "epoch". MPI provides two different ways to define "epochs", called *active target* and *passive target*.

In the *active target* mode both processes are required to take part in the synchronization. The programmer need to declare the beginning and end of an "epoch" in the origin process, by explicitly calling `MPI_Win_start` and `MPI_Win_complete`, respectively. On the target process, `MPI_Win_post/wait` must be used to declare the beginning and end of the "epoch". `MPI_Win_start` needs to be paired with an `MPI_Win_post` and `MPI_Win_complete` must be paired with an `MPI_Win_wait`. Moreover, an "epoch" can be defined by using a pair of `MPI_Win_fence` calls to declare the start and end of the "epoch". This function is used for collectively synchronizing remote operations. All these functions require an MPI_Window to be provided as an argument.

The *passive target* mode requires only the origin process to define the start and end of an "epoch", by using `MPI_Win_lock/unlock` respectively. Again, the window on which the operation is performed is required to be passed as an argument along with the rank of the target process. An `MPI_Win_lock/unlock` can be either shared or exclusive. A shared lock allows concurrent operations to take place in the same "epoch", while the exclusive will force them to happen in different "epochs". However, note that concurrent conflicting accesses to the same MPI_Window can be erroneous, and MPI locks are not to be confused with mutual exclusion schemes.

In the example in figure 1.4 we use the *passive target* scheme to define the "epoch". Each remote operation is surrounded by a pair of `MPI_Win_lock`, `MPI_Win_unlock`. The master process first will send the ping message to the worker process. This is done in one matching "epoch" on both processes. At the first call of `MPI_Win_unlock` at line 21, the master process is required to have finished sending the data to the worker process. Respectively, at line 32, the worker process will need to receive the data before moving on. Note that this can happen asynchronously. In the second "epoch" the worker will send a pong message to the master, in the same fashion.

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char** argv) {
  int rank, procs, msg_size;
  char *shared_buff, *message;
  MPI_Win win;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  MPI_Alloc_mem(sizeof(char)*msg_size, MPI_INFO_NULL, &message);
  MPI_Win_create( message, msg_size, sizeof(int),
                  MPI_INFO_NULL, MPI_COMM_WORLD, &win);

  if(rank == 0) { //master code
    message = strdub("ping");
    //epoch 1
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 1, 0, win);
    MPI_Put(message, msg_size, MPI_CHAR, 0, 1, msg_size, MPI_CHAR, win);
    MPI_Win_unlock(target_rank, win);
    //epoch 2
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 1, 0, win);
    MPI_Get(shared_buff, msg_size, MPI_CHAR, 0, 1, msg_size, MPI_CHAR, win);
    MPI_Win_unlock(target_rank, win);
  }
  else { //worker code
    message = strdub("pong");
    //epoch 1
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 0, 0, win);
    MPI_Get(shared_buff, msg_size, MPI_CHAR, 0, 0, msg_size, MPI_CHAR, win);
    MPI_Win_unlock(target_rank, win);
    //epoch 2
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 0, 0, win);
    MPI_Put(message, msg_size, MPI_CHAR, 0, 0, msg_size, MPI_CHAR, win);
    MPI_Win_unlock(target_rank, win);
  }

  MPI_Win_free(&win);
  MPI_Free_mem(message);
  MPI_Finalize();
}
```

Figure 1.4: Simple "ping pong" example using MPI's one-sided communication interface

Chapter 2

Related Work

2.1 MPI one-sided communication Evaluation

Dinan et al [19] have implemented the Global Arrays (GA)[7], a PGAS model, over the one-sided communication interface of MPI. In their work they ported GA's low-level ARMCI [4] one-sided communication librari using the MPI API and compared it to ARMCI. Although, they succesfully delivered a high-performance runtime, they are critical on both interface usability and performance of MPI one-sided interface. Bonachea in his report [20] also supports that the MPI one-sided interface is not fit to be used for the implementation of PGAS languages. There are however examples [21, 22] where MPI's one-sided interface has been succesfully used to implement high-performance applications.

2.2 Distributed Futures Implementations

High Performance ParalleX (HPX) [23] is a parallel runtime system implementation of the ParalleX[24] execution model. One of ParalleX's many features is the futures synchronization model. The adopted futures interface is similar to the C++11 standard library one and is available for both shared and distributed memory. The model offers additional abstractions to the futures interface, that can be used to describe data-flow relations and asynchronous computations. In contrast with our work, it does not use an MPI library for communication, but a different batch system.

2.3 Other asynchronous distributed systems

Other high-performance systems that support Remote Method Invocation (RMI), RSR and RPC share similar specifications with our runtime system, regarding asynchronous execution.

ARMI[9] is a low-level hybrid (using both threads and message passing) communication library, which supports RMI. In ARMI, objects are shared between threads and processes, but requires manually setting the aggregation factor in order to have an object's data effectively distributed on processes that do not share a common address space. On such case, method calls of the object are done using the library's RMI primitives. These primitives are implemented on top of MPI and although they share similar asynchronous charasteristics with our system's implementation, they emulate asynchrony by polling at certain time intervals.

Tulip[10] is another object-parallel system. It provides implementations of remote access put/get and RPC primitives over different hardware setups and requires a compiler to create the handlers used in RPC and Active Messages. In some cases, these primitives are implemented using the MPI library and polling for messages, if DMA is not available for communication between processes.

Charm++ [25] is an parallel object-oriented extension to the C++ language. It is based on the *Actors* [26] but differentiates sequential and parallel objects. It uses a Message driven execution model, different from the traditional send/receive pairing, where computations begin when a message is received. The parallel, work unit in Charm++ is called a *chares* and different *chares* can communicate between themselves. Instead of RPC, it uses a futures implementation that provides the same interface for local and remote invocation, in order to have overlapping communication and computation. This implementation however is not based on MPI or another one-sided communication interface.

The RSR sheme from Nexus[12] is similar to the RPC. The user needs to define a handler for the RSR that is going to be run remotely, and the data the handler will operate on. The underlying system will decide on the mechanism used that the data will be communicated. Nexus offers a variaty of methods to achieve asynchronous commonucation in order to remotely execute RSR handlers, depending on available OS and/or hardware. We will discuss these different techniques shortly, at the end of this section.

Active-Messages is another communication model, where data that is transfered between processes is paired with a handler, which is an action that is performed upon the arrival of data on a process. This scheme shares some common asynchronous characteristics with the RPC model. AMMPI[27] is an Active-Messages implementation over MPI two-sided communication interface and LAPI [5] is a low-level communication library, that offers an interface similar to Active Messages.

Most of these systems require asynchronous communcation to be effective. There is a number of known solutions as to how to implement such systems in the literature. The two most commonly used methods are polling for work requests

[10, 9, 12, 28] and hardware interrupts. Polling would require a worker process to poll for incoming messages/work at certain time intervals. Extra care must be taken to define the polling period, since if polling happens too often, it can dominate computation, but infrequent polling could render the system unable handle requests in time. [9, 5]. Alternatively, a hardware interrupt could be sent to notify a process of an incoming message. This method however, is avoided because interrupts have to go through the OS, which has a significant cost. [9, 5, 12, 28]. The Nexus system [12], also suggests dedicating threads only for communication. These threads can either probe for pending messages or block (depending on the underlying communication library and OS capabilities). How responsive this implementation can be depends on the thread implementation and OS (for example if the OS supports priorities). A detailed discussion and comparison between using threads for communication versus probing or interrupts can be found in [12].

In our implementation, we use none of these methods, instead we use MPI's one-sided communication interface. The benefit of using a one-sided communication interface, lies in the fact that we can have real asynchronous execution. In contrast with polling, the the system can react without any delay (polling period), while it will not suffer from costly interrupts or the extra overhead and reponse delay of having a thread running, as we discussed above. However, synchronization in such a system can become a serious performance problem, as with shared memory models (fences, barries mutexes).

Chapter 3

Design and Implementation

In this chapter we describe the interface and the implementation details of our futures library. Our interface allows the programmer to issue callable objects, referred to as *jobs* from now on, to be executed asynchronously by other processes. In C++ a callable object is any language struct that can be treated as a function (function pointers, functors etc). An asynchronous call of such an object will return a future object instead of its normal return value. This future object can be used by any other process to retrieve the encapsulated value. If the functor has not been executed yet, the reference to the future's value will block[1] until it becomes available.

We designed our system to be modular, so that different aspects of the runtime library, such as process communication, hide its underlying implementation (e.g. MPI), thus different implementations of the same module should not interfere with other components of the library. Figure 3.1 shows the different component hierarchy.

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 which process will run a *job*.

All the above modules are initialized, finalized and managed by a system environment, an instance of which is present at every process. Note that it is not necessary for all processes to keep identical environments, which means that other

1. as we'll see in section 3.4, only the master process blocks while other processes will try to run any functor objects that are scheduled to be run.

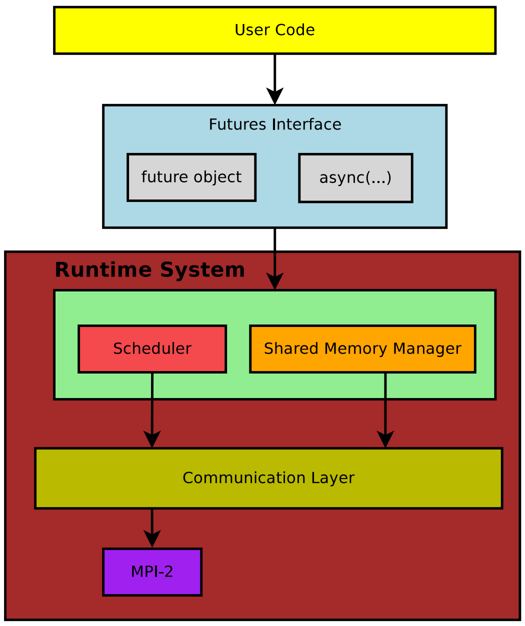

Figure 3.1: Overview of our futures library design.

than the initialization and finalization, processes are only responsible for their local environment, no updates are necessary.

Figure 3.3 shows the program flow for processes 0 and 1, of the simple hello world example in figure 3.2. 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*. The *async* function can be called from any process and within other async calls, thus allowing recursive algorithms to be expressed. In the example, process 0 issues a *job* by calling *async(f)*. It will then return from the call and continue until the message.get() call, at this point the process will either retrieve the message value or block until it's set.

The job is then scheduled to be executed by process 1. The worker process, here process 1, will wait until a *job* is send and then run it. When done, it will set the future's value and resumes waiting for other jobs or until it is terminated by the master process. When process 0 retrieves message's value, it prints it and continues until it reaches the Futures_Finalize() routine. At this point it will signal all other processes that the program has reached it's termination point and finalize the futures environment. All other processes will do the same after receiving the terminate signal.

```
class helloWorld {
public:
  helloWorld() {};
  ~helloWorld() {};
  int operator()() {
    int id = Futures_Id();
    cout << "- Worker" << id << ":Hello Master" << endl;
    return id;
  };
};

FUTURES_SERIALIZE_CLASS(helloWorld);
FUTURES_EXPORT_FUNCTOR((async_function<helloWorld>));

int main(int argc, char* argv[]) {
  Futures_Initialize(argc, argv);
  helloWorld f;
  future<int> message = async(f);

  cout << "- Master :Hello " << message.get() << endl;

  Futures_Finalize();
};
```

Figure 3.2: A simple hello world implementation using the distributed futures interface. The output of the program on process 0 would be "- Master :Hello 1".

In the rest of this chapter, we will present the future interface in section 3.1 and discuss our implementations of the the communication, shared memory manager and scheduler modules in sections 3.2, 3.3 and 3.4 respectively.

3.1 Futures Interface

An important goal of this work is to provide a unified interface for both distributed and shared memory machines. To meet this end we decided to replicate the C++11 futures interface from the standard threads library, with which the C++ community is already familiar and works well with generic programming. We had to make some additions to the interface and impose some restrictions, but they do not limit the capacity of the programmer to express parallelism, while still keeping the interface as simple as possible to use (we will discuss them shortly). Figure 3.5 shows a recursive implementation of the fibonacci function using our future implementation. The user can issue callable objects to be run asynchronously

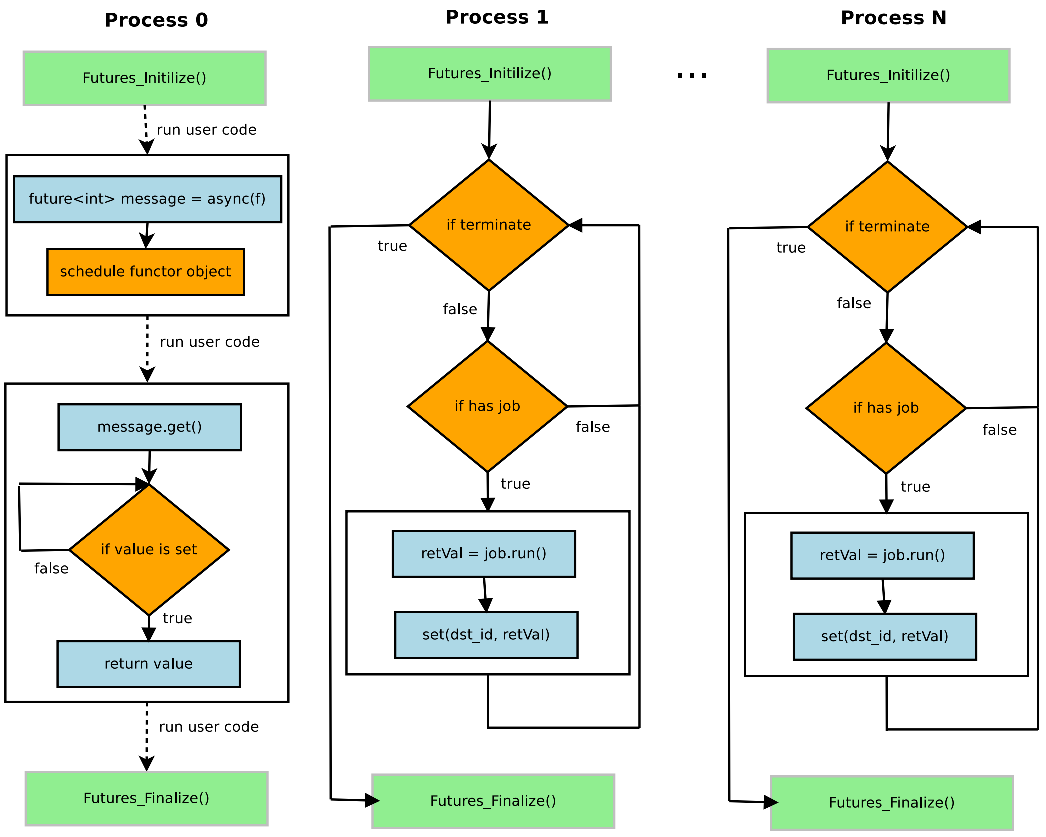

Figure 3.3: The control flow of the hello world program in figure 3.2.

by other processes, using the *async* function and passing the callable object, along with its arguments, as the *async*'s arguments. Note, that in our implementation, the callable object can only be a functor object. No normal functions, or function pointers, etc can be used here, which is the major restriction our implementation has, compared to the C++11 standard library. The reason for this is that we send the callable object through the communication module, using messages. We do not use a compiler, or require the programmer to identify the functions that are to be run asynchronously and provide a mapping of them to all processes. Instead, we serialize the functor object and send through the message passing library.

This restriction implies the one major limitation of our interface, which is that the functor object, as well as all of its arguments, must be serializable. Back to our example in figure 3.5, note that *fib1* and *fib2* are both functor objects. The user can either provide the serialization routines himself (see [29] for more details on how to serialize a C++ object with Boost serialization library[2].), or

2. We use the boost serialization library [29] and the input/output archives from the boost mpi library [30]

use the `FUTURES_SERIALIZE(F)`, here F is a functor object, which will create the necessary serialization routines automatically. The former is only recommended for very simple functors that have to state (members). Moreover, the user needs to expose the functor type to the underlying serialization library. This is done with the macro command `FUTURES_EXPORT_FUNCTOR(async_function<fib, int>)` in our example. Here the type declaration must be wrapped in the `async_function` type, which is the library internal template class for all *jobs*. The template type here is the functor type, fib, and the argument types that will be passed when calling the functor, here `int`. Instantiation of the `async_function` class, using C++ templates meta-programming capabilities, generates the appropriate routines, for setting the future's value, according to the functor's return type. It also facilitates all necessary information that are needed to be transferred to the worker process. Figure 3.4 shows the definition of the `async_function` class.

A call to the *async* function is non-blocking and returns a `future<T>` object immediately, where `T` is the return type of the encapsulated functor object, passed to *async*. 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 none blocking, to check if the value can be retrieved, and if not continue running user code until the future's value becomes available at a later point. Also, note that before using the futures library, the user has to explicitly call the `Futures_Initialize()` and `Futures_Finalize()`, which will initialize and finalize the futures environment, respectively.

3.2 Communication

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

The communication module acts as a layer of abstraction between the various system component and the MPI library. 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 process, rank e.t.c.). Moreover, it can be used to expose part of a process' address space to other processes in the same communication group.

```

template<typename F, typename... Args>
class async_function : public _job {
 ... //we have ommited here all the serialization routines
public:
    int src_id;
    int dst_id;
    Shared_pointer ptr;
    int data_size;
    int type_size;
    F f;
    std :: tuple<Args...> args;
    typename std::result_of<F(Args...)>::type retVal;
    async_function();
    async_function(int _src_id, int _dst_id,
                  Shared_pointer _ptr,
                  int _data_size, int _type_size,
                  F& _f, Args... _args);
    ~async_function();
    void run();
};
```

Figure 3.4: The `async_function` function class definition. All *jobs* in our system are instances of this class. The base class `_job` is used for serialization purposes as well.

3.2.1 Shared Address Space

In our implementation, the underlying message passing library used is MPI-2, thus we use MPI windows to expose such space among processes. Exposing part of process' address space in the MPI-2 schema, requires that the some space will be locally allocated to a pointer using the `MPI_Alloc_mem`, and then exposed to other processes through creating an MPI window that is correlated to the pointer with `MPI_Create_Win` (See section 1.3). A drawback in MPI is that a window can be created only collectively over an MPI communicator, and in turn, a communicator can be created, again, only collectively over an existing parent communicator. In our design, this requires that either all windows are created a priori at initialization, since when issuing a job, only the sender and receiver should take part in the communication. In order to overcome this limitation, we implemented the algorithm presented in [31], which requires only the processes that will join the communicator to take part in the communicator creation process. The algorithm needs an MPI group as input and progressively creates two adjacent groups of processes. If a process' id is even, then the process is added to the *right* group,

```
class fib {
public:
   fib () {};
   ~fib () {};
   int operator()(int n) {
      if(n == 0) return 0;
      if(n == 1) return 1;
      fib f;
      future<int> fib1 = async(f, n-1);
      future<int> fib2 = async(f, n-2);
      return fib1.get() + fib2.get ();;
   };
};

FUTURES_SERIALIZE_CLASS(fib);
FUTURES_EXPORT_FUNCTOR((async_function<fib, int>));
```

Figure 3.5: A fibonacci implementation using the distributed futures interface

if the process' id is odd it is added to the *left*. Every time a process is added to either group, an inteprocess communicator is created and then merged with the adjacent group's interprocess communicator. The algorithm's pseudocode can be found on [31, p.287]. Employing this algorithm we can dynamically allocate windows between any two processes that compose an MPI group.

The communication library also provides the routines needed to write and read data from an address space shared though an MPI window, using the special `Shared_pointer` construct (see section 3.3). 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 3.6 shows a simplified version for setting a future's value. The `ptr` variable has information on the location we need to write the data to on an MPI window. The `shared_space[ptr.page_size]` is a map that contains MPI windows. Section 3.3 explains how MPI windows are organized in this map, according to the page sized used during allocating space for a future. Note that the variable datatype, `MPI_Datatype` in this implementation, is inferred statically using template routines from the Boost MPI library, when instantiating the `async_function` class. The second overloaded `set_data` method is used for when the future's value is not a primitive data type and requires serialization. In the latter scenario, we need to store information on the archives size, thus the actual data is indexed at location `ptr.base_address+DATA_OFFSET`.

```
void set_data(void* val, int dst_id, Shared_pointer ptr,
              Datatype datatype) {

    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, dst_id, 0,
                shared_space[ptr.page_size]);

    MPI_Put(val, ptr.size, datatype, dst_id, ptr.base_address,
            ptr.size, datatype, shared_space[ptr.page_size]);

    MPI_Win_unlock(dst_id, shared_space[ptr.page_size]);
};

void set_data(boost::mpi::packed_oarchive& ar, int dst_id,
              Shared_pointer ptr) {

    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, dst_id, 0,
                shared_space[ptr.page_size]);

    MPI_Put(&ar.size(), 1, MPI_INT, dst_id, ptr.base_address,
            1, MPI_INT, shared_space[ptr.page_size]);

    MPI_Put(ar.address(), ar.size(), MPI_PACKED, dst_id,
            ptr.base_address+DATA_OFFSET,
            ar.size(), MPI_PACKED, shared_space[ptr.page_size]);

    MPI_Win_unlock(dst_id, shared_space[ptr.page_size]);
};
```

Figure 3.6: The function used to set a future's value. The first version is for primitive data types, where as the second is for serializable objects.

3.2.2 Mutexes

In order to synchronize accesses to shared memory addresses and other critical sections in our system, designed a mutex library, with 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 target 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 vector 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.

3.3 Shared Memory Management

The Memory Manager module is responsible for managing the systems shared address space. 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 functionality 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 [32, p. 185-187]. However, we keep different free lists for different page sizes to deal with memory segmentation. Figure 3.7 shows how the memory manager keeps a map of free lists indexed by a memory page size. The shared address space is allocated a priori using the communication module, to create MPI windows in our current implementation. This is of-course transparent to the Shared 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`. The information tracked by a `Shared_pointer` can be effectively used by the communication module to read/write data. The Shared Memory Manager modules simply holds a mapping of the shared address space, the actual local addresses are handled by the underlying communication library (MPI in our case). So, each freeList in figure 3.7 is actually a list of `Shared_pointers` to a corresponding MPI window. We choose to keep separate windows for each free list because when acquiring an epoch access to an MPI window, the whole window is locked, so even though we do not have overlapping accesses[3] (see section 1.3).

3.3.1 Memory Allocation/Deallocation

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 makes use of the shared memory manager module. The shared memory allocator tries to find the best page size fit for the data size (the one that is closest to the data's total size), and searches the corresponding free list, using a first fit algorithm [32, p. 185-187] 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

3. only one process needs to write to a future's shared address, since only one future is associated with one *job*.

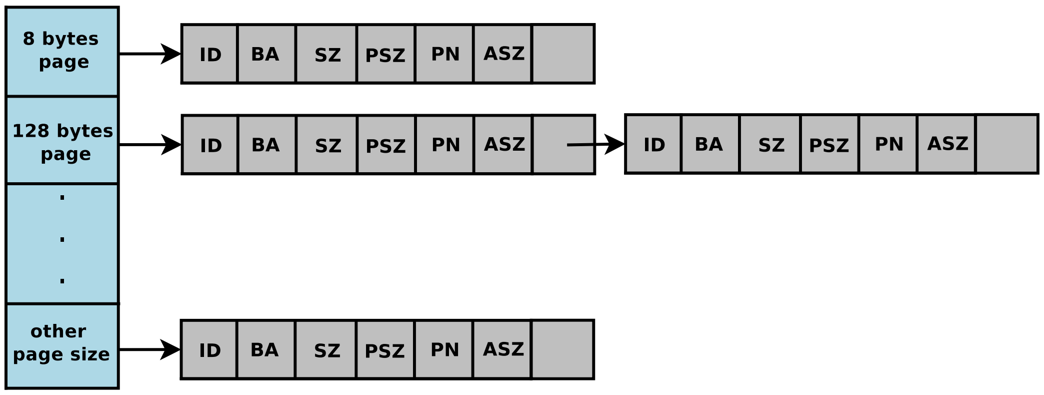

Figure 3.7: Shared Memory Manager keeps a map of free lists, indexed by the page size. For page size that do not match any predefined ones, we use the *other* page size free list.

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 3.8 shows a free list, before and after allocating a data object. The first fit algorithm will iterate the list from the start until it finds a large enough space for the object. Each node in the free list, is a `Shared_pointer`, which describes how much continuous space there is available. When the allocator finds a large enough node, it removes from that node the size and number of pages it needs and sets its base address value accordingly. 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 3.8, the first list node has enough space to fit an 128 size data object. Removing the reserved now space, from the beginning of the list, will leave us again with two free nodes, but the first one will now have 512 bytes left and the base address will be moved at the 128th byte.

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 fashion, 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 continuous space, we merge the list elements, resulting in larger block of free space. Figure 3.9 shows a free list, before and after freeing some shared memory. Because freeing 128 bytes at base address 512 creates a continuous space from byte 0 to byte 640, the two list nodes will be merged into one.

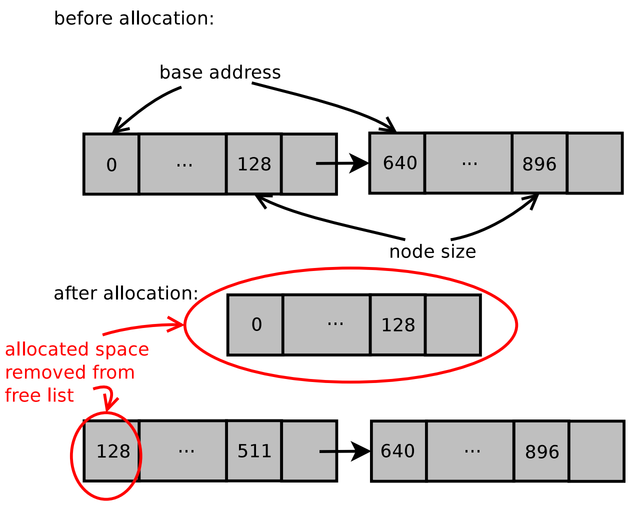

Figure 3.8: During allocation, when a large enough space is found, the allocated page is removed from the node.

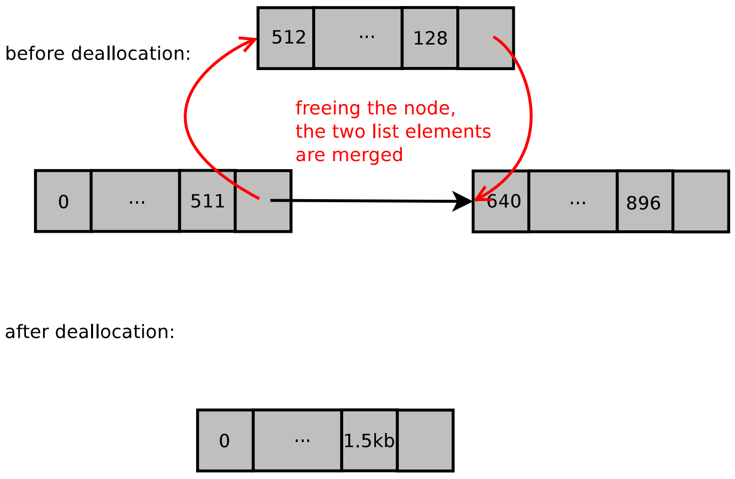

Figure 3.9: By freeing data at base pointer 512, creates a continuous space between base pointer 0 and base pointer 640, causing the list nodes to merge into one.

3.4 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.

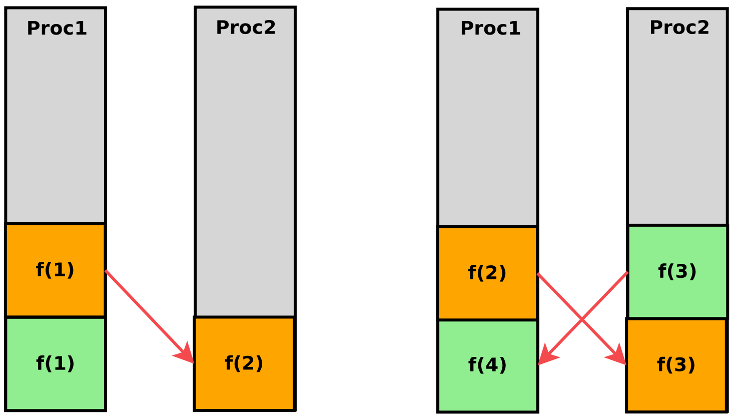

Figure 3.10: *Job* stacks for running fib(3) on the left and fib(5) on the right. Matching colors denote that *job*s are spawned from the same recursion path.

To make our case clear, consider the fibonacci example in figure 3.5. In our example, let's say we have 3 processes, one of them is the master process. Figure 3.10 shows how the *job* stacks would look like, for running fibonacci with argument 3 and 5 respectively. The arrows show how *job*s depend upon each other. A *job* blocks and waits until the *job* that the arrow points to finishes. Running fib(3), process 0, the master process issues `async(f, 1)` to process 1 and `async(f, 2)` to process 2. Process 2 issues `async(f, 1)` to process 1. In this scenario the program will execute correctly without any problems, since any of the processes' call to `get()` will eventually retrieve the future 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 be able to resolve the dependencies as shown in figure 3.10. This scenario is not a problem if processes are dynamically spawned, but if we have a static number of processes, which is common for MPI programs, we need to address this 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 *job* 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 3.11 shows how a *job* stack is structured. Note that an entry is composed by the functor object, its arguments (they are considered one object) and the size of the entry. This is necessary since different functors and/or

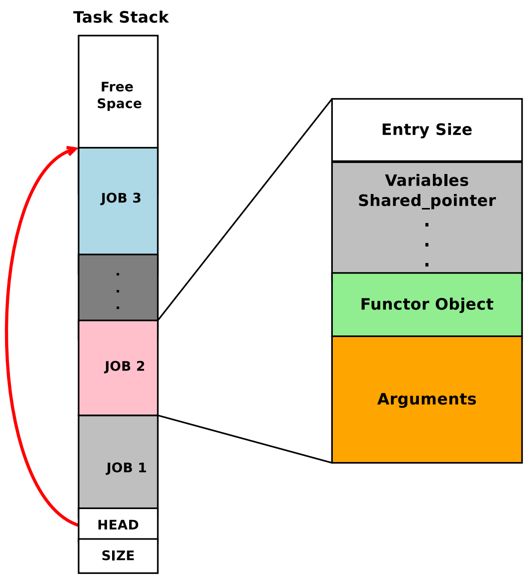

Figure 3.11: Shared stack where a worker process keeps its pending jobs. Entries can have varied sizes, this size is stored at the beginning of the entry and can be used to retrieve the corresponding job. Information for the specific stack,like size and head, are stored at the beginning of the shared space, so that other processes can access them.

different arguments result in varying entry sizes. Thus, the exact location of a *job* is calculated using the stack head and functor object size values[4]. Moreover, at the beginning of the shared space, the size and current head values are stored, so all processes can push *jobs*.

Figure 3.3 shows a control flow graph for the master and worker processes. 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 stacks until their 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 functionality 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

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

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 available 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 int the process' stack. Our scheduler distributes *jobs* in a round robin fashion (excluding the master process, which should run user code).

3.5 Extra Features

3.5.1 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 useful 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 at a later point of the code as an actual future.

In section 3.4 we described our scheduler module. This modules mainly facilitates the necessary infrastructure to send *job*s between processes asynchronously, but also it is responsible for making a decision on how *job*s will be distributed among these processes. Although a simple scheduler policy, like Round Robin, can be sufficient for many scenarios, it is possible that a more elaborate work or data distribution scheme is required for better performance. Especially in a distributed environment, we would like to be able to distribute data in fashion that takes advantage of data locality and/or avoid excessive data transmission through the network. For this reason, we have added a variation of the async function, `async_on`, which is identical to an *async* with the addition of the target process id. The `async_on` function still makes use of the scheduler infrastructure the same way the default *async*, it simply skips the step where the scheduler decides which process will receive the new *job*.

3.5.2 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,

```
template <class T>
class future {
private:
   ... // serialization   routine  omitted
      int ready_status;
      T data;
      int src_id, dst_id;
      Shared_pointer status_ptr;
      Shared_pointer data_ptr;
      int type_size;
      int data_size;
public:
      ... //constructors  and  destructors  omitted
      bool is_ready();
      T get();
};
```

Figure 3.12: The future object definition.

can be transferred to another one, thus synchronization can take place on the worker process, which allows finer granularity when synchronizing task. Consider the two examples in figures 3.13a and 3.13b, where we can observe a pipeline scheme implemented without future serialization and with future serialization. In figure 3.13a at pipeline loop stage2, the master process will have to wait on res1[0], even if re1[n], where n $>$ 0, is available before res1[0]. This forces sequential issuing of the stage2 *async*, which can limit performance and breaks the pipelining scheme. Now consider figure 3.13b, in this example each stage function will only need to wait on its corresponding future, even if previous futures in the arrays res1 and res2 are not available, a stage function can proceed normally its execution. This time we have an accurate pipeline implementation.

In our implementation, a future object is defined as shown in 3.12. The variable `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 3.3), 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` and `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

```
...
for(int i=0; i < N; i++) {
  res1[i] = async(stage1);
}

for(int i=0; i < N; i++) {
  res2[i] =
    async(stage2, res1[i].get());
}

for(int i=0; i < N; i++) {
  res3[i] =
    async(stage3, res2[i].get());
}
...
```

(a) Pipeline without serilizable futures

```
...
for(int i=0; i < N; i++) {
  res1[i] = async(stage1);
}

for(int i=0; i < N; i++) {
  res2[i] = async(stage2, res1[i]);
}

for(int i=0; i < N; i++) {
  res3[i] = async(stage3, res2[i]);
}
...
```

(b) Pipeline with serilizable futures

Figure 3.13: Different implementations of a pipeline scheme using our futures .

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 `status_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 changing the data's and status' location to another process, would require an update to all other processes that are associated with that future. This would require significant synchronization and communication. In practice, 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 still retrieve the future value from the original owner, or have it sent to multiple processes, since the actual data will always reside on the same place for everyone. This means that our future object behaves just like C+11's `shared_future` object.

Chapter 4

Evaluation

In this chapter, we asses our effort to implement the C++11 standard future library, as a unified interface for both shared distributed memory environments. We also evaluate the performance of our runtime system.

4.1 Interface Assessment

Figures 1.3 and 3.5 show an implementation of the fibonacci function using C++11 standard futures and our futures library, respectively. The interface used to express parallelism is identical. An *async* call is used to send asynchronous *jobs* for execution, while the return value is encapsulated in a future object. The return value can be retrieved in the same fashion in both libraries, using the get method. Our future object behaves also like a shared_future from the C++11 standard library, which means that it can be accessed by different processes concurrently.

One difference to the interfaces is that our implementation requires the data size of a *job*'s return value to be defined. This is limited only to the case where the return value is dynamically sized object or a pointer. However, this does not require significant effort, from the programmers part. Another difference between the two interfaces is that only functor objects can be issued by the *async* function and that the programmer has to explicitly identify the functors as *job*s, in order to expose their type and arguments to our runtime system. This has minimal impact on the user interface. It is done very easily just by adding an extra line of code, in the example in figure 3.5 this is done with the macro command `FUTURES_EXPORT_FUNCTOR(async_function<fib, int>)`. The implications of this are discussed further in 4.1.1.

At this moment, the interface lacks some secondary features, like defining a launch policy and timing out after a period of time upon waiting on a future value. Note however, these features are trivial to implement and of little interest to us at this point. Another feature that we have neglected to implement is the ability to return an exception instead of the functor's return value. On the other hand, we have added a couple of features (see 3.5). The `make_future` functions can be found both in the Boost futures library and in HPX. Moreover, the `async_on` function allows greater flexibility to the programmer. He can easily override the default scheduler and use a custom distribution scheme that matches his needs.

4.1.1 Limitations

The core limitation of our interface is that all *jobs* must be functor objects, whereas in the C++11 standard, it is possible to use any callable object (functions, function pointer, functors, lambda function etc.). This of-course, does not limit our library's expressiveness, when it comes to parallelization. A programmer can easily wrap any callable object in a functor object. The programmer's extra burden here is not note-worthy. On the other hand, the functor object, its return value and its arguments must be serializable, as well. This limitation manifests itself in our interface, both as an additional burden to the programmer and as a potential limitation, when porting codes with lots of pointers and maybe non-serializable objects to work with our library. To better understand the implications here, we share our experience when we tried to port Ferret, a benchmark from the PARSEC [33] suite, using our library.

Ferret is a content-based similarity search engine toolkit for feature rich data types (video, audio, images, 3D shapes, etc). Ferret uses a pipeline model for parallelization, with a thread pool at each stage. A queue exist for each stage and whenever there is an entry on that queue, a thread from the thread pool will pop it, and start execution. When it's done, the tread will push an entry on the next stage's queue and so no. This scheme can be expressed with futures extremely easily. Figure 3.13b shows how a pipeline scheme with futures would look like. We tried to implement this using both C++11 standard futures and our library. The algorithm was relatively easily modified to work with C++11 standard futures. However, in order to work with our version, we needed to modify the original queue entries, which our now the arguments to the *async* functions, in order for them to be serialiazable. This, proved to be quite a challenge. First, the original code was in C language and used void* extensively. All these pointers had to be encapsulated in C++ vector<unsigned char> objects. The challenge here is that we needed to modify most of the functions of the whole Ferret library, either to work with vectors instead of void pointers or simply to modify a number of functions to return the allocation size for each void pointer, in order to convert them to vector objects. In this example, this has proven to include a large number of functions (ferret counts more than 3000 LOC), as a result we have not yet completed its porting. We report this in order to show a case, where the requirement to serialize arguments, return values etc, can have a significant impact on ease of use. Of-course, as a counter argument here, it is still easier to use our interface instead of re-writing the whole ferret code to use a message passing library like MPI directly. It should also be noted, even with a message passing library, serialization or manually manipulating the data of the void pointers would still be required.

4.2 Performance Evaluation

We evaluated our runtime's implementation performance by running some microbenchmark applications and three small applications (fibonacci, quicksort, LU). We run all benchmarks on two Intel(R) Xeon(R) CPU E5645@2.40 GHz with 6 available cores on each machine, totaling to 12 cores connected through a network socket. We have compiled the runtime and application code using g++ version 4.6.3 with level 3 optimization enabled. For the MPI library we used OpenMPI version 1.4.3.

4.2.1 Microbenchmarks

Our first microbenchmark is a ping pong application, which is used to measure the time needed to send a message from the master to a worker node and the time needed for the worker node to respond back to the master. Using the future interface, the master simply calls async with a functor that takes a string argument ("ping") and returns only a string value ("pong"), without doing any other computations in the functor's body. We run the ping pong microbenchmark using the configuration described in 4.2 and the message was received by the master in 0.8ms.

The rest of our microbenchmarks, aim to help us understand better the time needed to issue a job from one node to another. To achieve this, we designed one microbenchmark application, where the master node issues a functor, with only a return statement in his body, which takes a variable number of arguments, each argument can be either a scalar value or a vector container. In figure 4.1 we report the time needed to issue a job that takes a variable number of scalar arguments comparing it with the time needed to issue a variable number of vector objects of one element. Although the vector object arguments are more complex to serialize, we see that the difference in execution time is marginal. Moreover, we see that the number of arguments has that need to be transferred, have minimal impact on execution time. Figure 4.2 shows the execution time of issuing functor objects with different vector argument sizes. In all cases the total number of elements totals to 1,200,000 (e.g. 1 vector of 1,200,000 elements or 4 vectors of 300,000 elements). Figure 4.3 shows the time for issuing 1 vector of different size. One can observe that there is a significant raise in cost as the number of elements reaches 1,000,000. The numbers reported are the median values of 20 runs. The cost differences here are again insignificant.

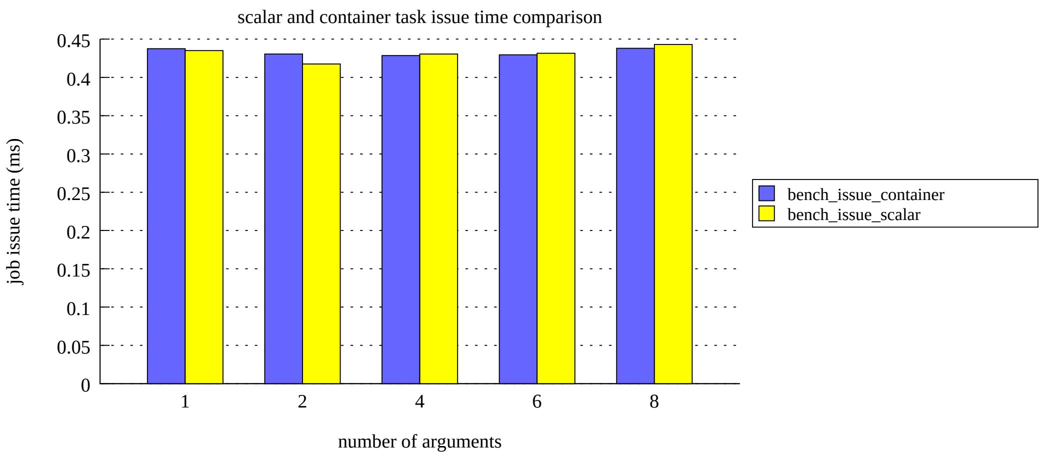

Figure 4.1: Comparison between issuing functors with scalar arguments versus vector objects of size 1

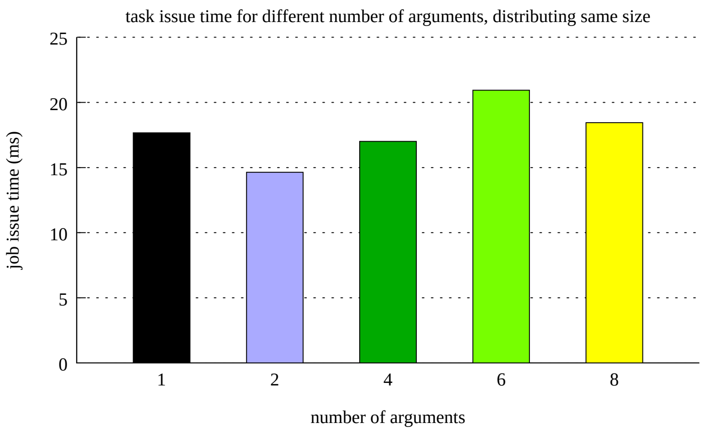

Figure 4.2: Comparison between issuing functors with different number of vector arguments, but total size of arguments is the same in all cases.

4.2.2 Real Application Benchmarking

In order to evaluate our runtime's performance we have implemented three algorithms using our future's interface.

Fibonacci: This is a simple implementation of the fibonacci function. Figure 3.5 shows our implementation. This recursive version is ideal to demonstrate the ease of use of the future's interface. We have modified the fibonacci code to run the sequential version of the code for values smaller than 30, so that each async function can have some amount of work. We run the fibonacci function with 45 as an argument.

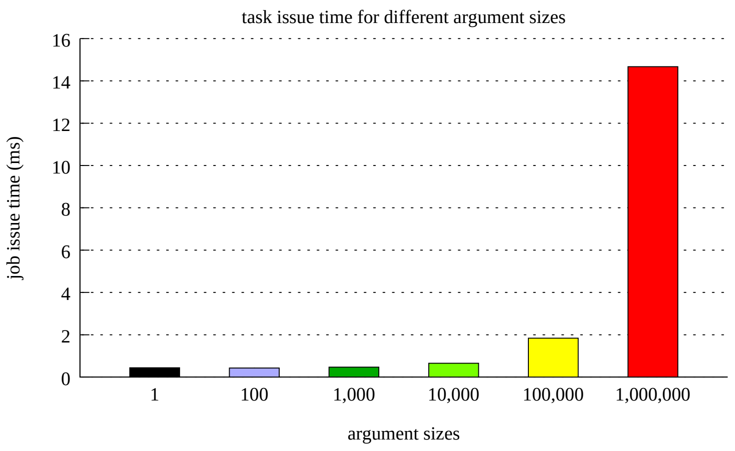

Figure 4.3: Comparison between issuing functors with 1 vector argument of different sizes

Quicksort: Figure 4.6 shows our implementation. The `QsSequantial` function itself is a pretty standard implementation of the common quicksort algorithm. Parallelization is extracted at the `quicksort` function, where the original array is partitioned and asynchronous quicksort functions are called until the `min_unit` value of elements is reached, where from that point on the sequential version of the quicksort algorithm is called on each partition. Notice that for the asynchronous branch of the code, we need to copy each partition in order to send it over the worker process and also merge the results of the async functions into the original array. This additional overhead along with the communication overhead makes it necessary to sort small sub arrays sequentially. For our experiments we sort an array of 100,000 doubles.

Tiled LU: We have implemented an LU factorization kernel using the Tiled LU algorithm as described in [34]. Figure 4.7 shows a simplified version of the tiled LU algorithm written in C++ style. All arrays are organized in tiles, each tile is a smaller sub array. Array A is the input array. In the first step an LU factorization is run on tile *A[k][k]* (*dgetrf* function). The resulting arrays are the lower triangular L, the upper triangular U, both of which are stored in *A[k][k]*, and the transmutation matrix *P[k][k]*. The *dgessm* function applies the L and U transformations on all tiles on row k, updating tiles *L[k][k...TOTAL_ TILES]*. *dtstrf* function performs a block LU factorization on the array formed by coupling the upper triangular part of *A[k][k]* with *A[k][k]*. This function returns an upper triangular array, stored in *A[k][k]*, a lower triangular array stored in *A[m][k]* and a permutation array *P[m][k]*. The *dssss* function updates the subarray formed by tiles *A[k+1...TOTAL_ TILES][k+1...TOTAL_ TILES]* by applying the transformation computed by *dtstrf* of the coupled array of the upper triangular part of *A[k][n]* and array *A[m][n]*. For the kernels *dgetrf*, *dgessm*, *dtstrf* and *dssssm*, we use the

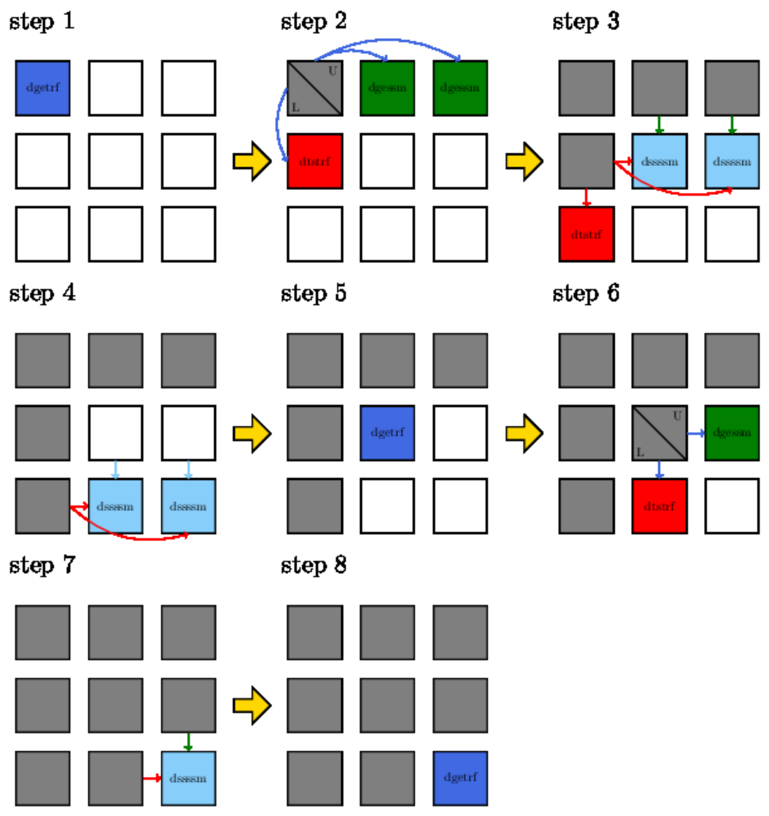

Figure 4.4: 3x3 example execution of our parallel tiled LU algorithm. The dark blue color denotes the dgetrf kernel. Red and green denote dtstrf and dgessssm kernels respectively. Light blue denotes dgssssm and gray denotes tiles that have completed.

implementation found in the Plasma project [35].

Figure 4.8 shows a simplified version of our parallel implementation. Figure 4.4 shows how the different kernels are applied at each tile for every step. It also shows how different tiles are depended upon each other. The example here is an array that contains 3x3 tiles. The master process starts by executing the *dgetrf* function. As soon as it completes, we can apply the *dgessm* function on the rest of the tiles on row k (k being the step index we are currently working on). Function *dgessm* only requires the L and U factors from *dgetrf* applied on A[k][k], thus we can issue them asynchronously, since all dependencies are met. We use here a special array of futures *fA* to hold the return value *dgessm*. Next, we apply the blocking LU transformation (*dtstrf*) on the rest of the tiles on column k. Here, because each *dtstrf* requires the updated A[k][k] tile from the previous application

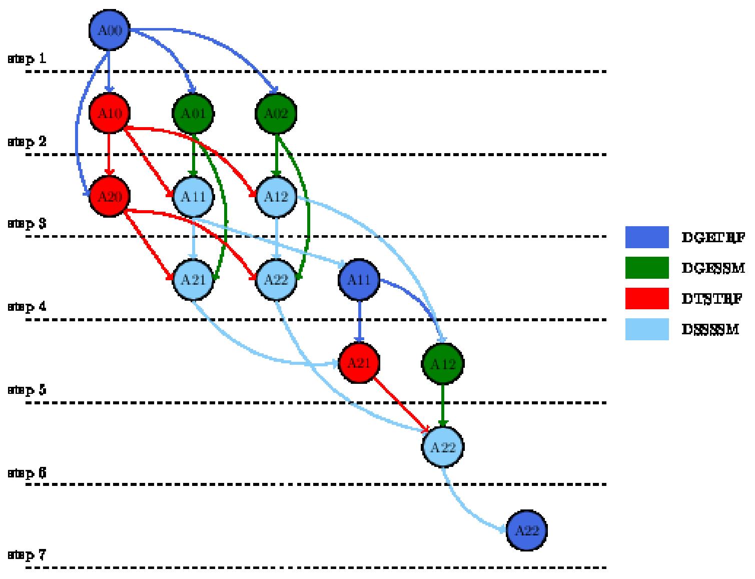

Figure 4.5: *Job* graph for tiled LU 3x3 example.

of *dtstrf*, we cannot issue them in asynchronously. Instead, after running a *dtstrf* we immediately issue asynchronous calls to *dssssm*. This function needs to wait from the *dtstrf* that is applied on the first tile on the row, for the *dgessm* function that will be applied on the first tile of the column, and from the previous, if any, application of *dgessssm* on the tile just above the current one, that *dgessssm* is applied. Because *dssssm* modifies two arrays, we use a struct to represent the coupling of tile A and the upper triangular array U. The variable cpldAU, is an array of futures of that struct type. The parallelization strategy described, allows us to work on each column asynchronously. Figure 4.5 shows the *job* graph for the Tiled LU 3x3 example. *Job*s on the same step can run in parallel. The arrows represent the data dependencies among the *jobs*. Note, that in figure 4.4 steps 4 and 5 can be merged, since all data dependencies on tile A11 are resolved. We run the tiled LU kernel for and array of 2000x2000 elements and block size of 200x200 elements.

In figure 4.9 we report the execution times for running the three applications on the machine setup we described in section 4.2. We measure only the algorithm and no initialization and finalization times of the runtime system, etc. We observe that we do not manage to get any speedup on quicksort and Tiled LU, on the contrary we get a slowdown (figure 4.10). We get a small speedup in Fibonacci when using more than 4 processes. In figure 4.11 we show the breakdowns for the master application and the slaves, for running the applications on 6 cores. *Job issue time* is the time needed to send a *job* from one process to another. This time includes time spend in the scheduler, to find the next worker. It also includes time spend on serializing and sending the *job* object and its arguments to a worker process. *Job execution time* is the time spent on running the actual code of the *job* that was issued via an *async* call. *User code time* is the time the master process spends running code that is not related with the runtime (only for master). *Idle time* is time spend waiting to retrieve a future value and for the workers, it's also time spent waiting for a *job* to become available in their stacks. *Rest of time* is the rest of the overhead that is imposed by the runtime. This fraction of time can be for example the time needed to send the return value of the asynchronous execution of a *job* and copies of objects done after deserialization on the workers. Figure 4.12 shows the same breakdowns with figure 4.11, with the addition of initialization and finalization times. The initialization and finalization include the creation and finalization of the communication module (in our experiments that's MPI), creation and destruction of the shared memory (MPI Windows) and scheduler. The initialization time is constant on all applications, since it mainly depends on the number of processes, while the finalization time is negligible. Compared to the useful *job execution* and *user code* times, the overhead is much greater. *Job issue time* or *rest of time* are the greatest sources of overhead, while a fair amount of time is also spent on *idle time*. The *idle time* is more a concern of the algorithm implementation of each benchmark, and explicitly related to our runtime's overhead. However, runtime overheads can implicitly be the cause for the *idle time*. Factors like how *job*s are distributed among processes and delay in issue time can play a significant role in this. In order to find the main source of overhead in our system, we used the callgrind tool to profile our code.

In the Fibonacci application the master spends most of his time waiting for the result on the get method. He does not make any useful work and only waits for the workers to finish. Callgrind reports that the workers spend 52% running useful code. They also spend a high amount (33.3%) of time on the scheduler trying to find the next available worker. However, out of the 33.3% only a very small fraction of time (around 3%) is spend on actual scheduler work. The rest 30% is spend waiting to acquire the lock of the distributed mutex implementation. One can see that the mutex implementation has a considerable cost.

The master process in quicksort spends 57% waiting for its workers to complete. It is again expected, since most of the work is supposed to be done by the workers, as in Fibonacci. The workers on the other hand, spend around 25% trying to acquire or release locks (distributed mutex) and 32% on vector allocations. Half of the 32% is called from within the serialization routine, which amounts to the total of 16% of total execution time. The rest of the vector allocation time is caused by copies of the `get` method's return value and local copies of the *async* function's argument, after deserialization.

For tiled LU, callgrind reports that the master spends 71% of its time trying to schedule/send *jobs* to the workers. It should be reminded, that all issuing happens by the master in this application, compared to the rest of our benchmarks. To quantify this statement, 330 *jobs* are issued by the master. Except a very small fraction (around 1% of the total execution time) is spend exclusively on the serialization routines. The master also spends 13% of its time on the `get` method, but 11% (of the total execution) is again deserialization of the return values (data send by the workers). The serialization routines here are indeed a vast amount of overhead. This implicitly causes the worker processes to be idle for a significant amount of time (46%). Around 11% is spend on serialization routines and 15.7% on useful work.

Boost serialization and the distributed mutex library can be identified as the two main causes of overhead in our system. Note that Fibonacci makes minimal use of the serialization routines, since it can directly send data (return values and arguments) as MPI datatypes. Serialization is required only for the *job* object. Thus fibonacci only suffers from the overhead caused by the distributed mutex library. The high cost of the distributed mutex library was expected, and we tried to use such locks only when completely necessary. It would be preferable to have a native MPI implementation of mutexes, but none of the MPI synchronization primitives can be used to define a critical region. Moreover, figure 4.11 shows that even if fibonacci, that there is some performance benefit, the overhead is considerable. One can also see that the fibonacci had a lot more useful work to complete compared to tiled LU and quicksort. We believe that the runtime can be only useful for coarser grain work, with the current implementation.

```
/* a sequential qs */
void QsSequential(vector<double>& array, const long left, const long right){
    if( left < right){
        const long part = QsPartition(array, left, right);
        QsSequential(array, part + 1, right);
        QsSequential(array, left, part - 1);
    }
}

/** A task dispatcher */
class quicksort {
public:
  quicksort() {};
  ~quicksort() {};
  vector<double> operator()(vector<double> array, const int deep) {
    const int left = 0;
    const int right = array.size()-1;
    if( left < right){
        if(array.size() > min_unit) {
            const long part = QsPartition(array, left, right);
            vector<double> subarrA((right)-(part+1)+1), subarrB(part-1-left+1);
            Copy(subarrA, array, part+1, right+1);
            Copy(subarrB, array, left, part);
            quicksort qsort;
            future<vector<double> > res1, res2;
            res1 = async2(subarrA.size(), qsort, subarrA, deep-1);
            res2 = async2(subarrB.size(), qsort, subarrB, deep-1);
            subarrA = res1.get();
            subarrB = res2.get();
            Merge(array, subarrB, subarrA);
        }
        else {
            const long part = QsPartition(array, left, right);
            QsSequential(array, part + 1, right);
            QsSequential(array, left, part - 1);
        }
    }
    return array;
  }
};
```

Figure 4.6: A quicksort implementation using the distributed futures interface

```
for(int k = 0; k < TOTAL_TILES; k++) {
  dgetrf(A[k][k], P[k][k]);
  for(int n = k+1; n < TOTAL_TILES; n++) {
    dgessm(A[k][n], A[k][k], P[k][k])
  }
  for(int m = k+1; m < TOTAL_TILES; m++) {
    dtstrf(A[k][k], A[m][k] , P[m][k]);
    for(int n=k+1; n < TOTAL_TILES; n++) {
      dssssm(U[k][n], A[m][n], L[m][k], A[m][k], P[m][k]);
    }
  }
}
```

Figure 4.7: The tiled LU kernel implementation

```
for(int k = 0; k < TOTAL_TILES; k++) {
  A[k][k] = cpldAU[k][k].get().A;
  dgetrf(A[k][k], P[k][k]);
  for(int n = k+1; n < TOTAL_TILES; n++) {
    A[k][n] = cpldAU[k][n].get().A;
    fA[k][n] = async(dgessm, A[k][n], A[k][k], P[k][k]);
  }
  for(int m = k+1; m < TOTAL_TILES; m++) {
    A[m][k] = cpldAU.get().A;
    dtstrf(A[k][k], A[m][k].get() , P[m][k]);
    for(int n=k+1; n < TOTAL_TILES; n++) {
      if(m == k+1)
        A[k][n] = fA[k][n].get();
      else
        A[k][n] = cpldAU.get().U;
      A[m][n] = cpldAU.get().A;
      cpldAU[m][n] = async( dssssm, A[k][n], A[m][n],
                            L[m][k], A[m][k], P[m][k]);
    }
  }
}
```

Figure 4.8: The tiled LU parallel kernel implementation

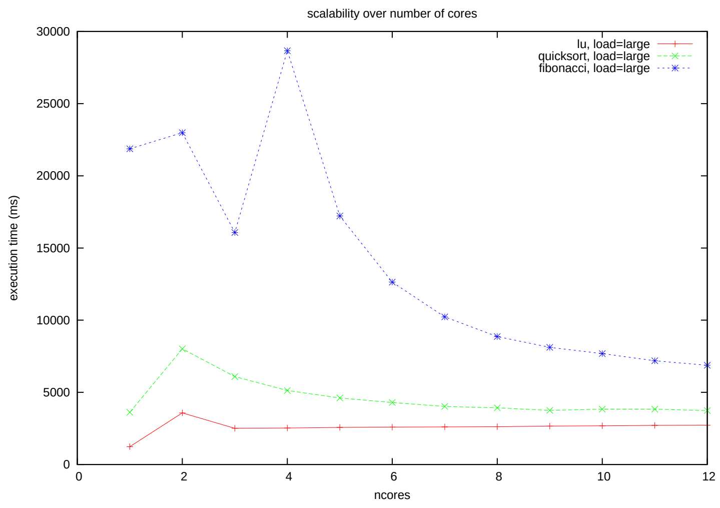

Figure 4.9: Scalability graph for fibonacci, quicksort and LU

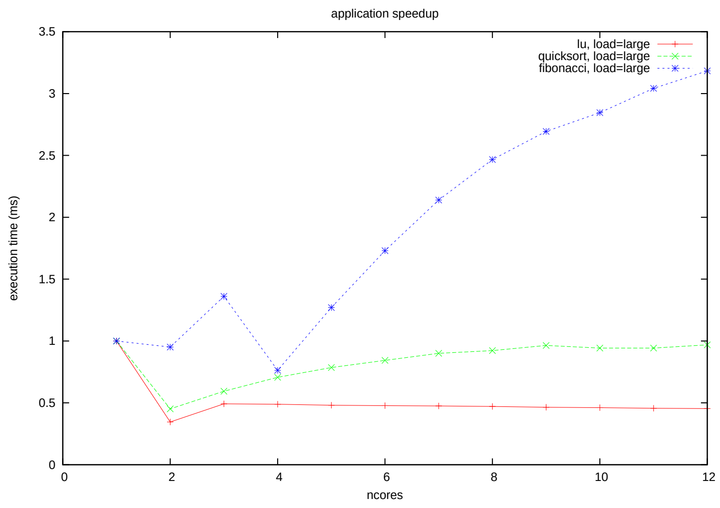

Figure 4.10: Speedup graph for fibonacci, quicksort and LU

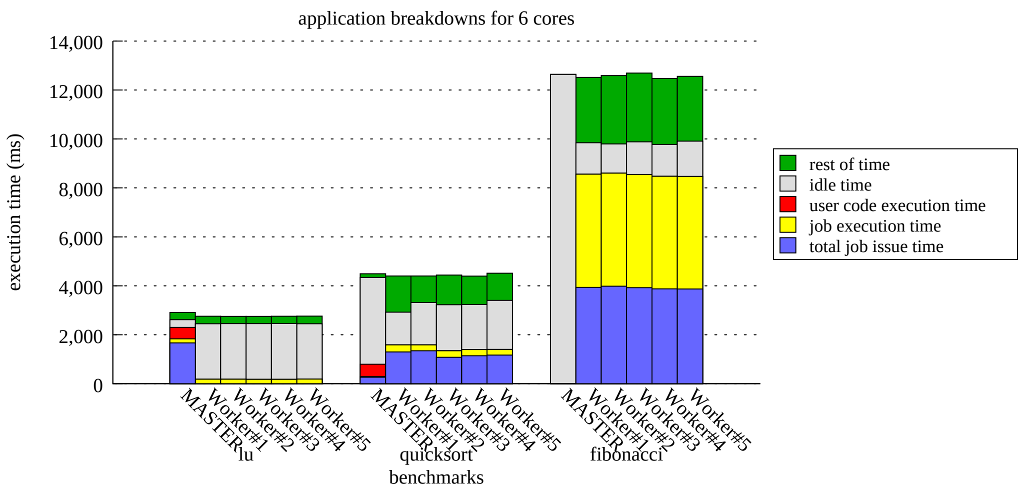

Figure 4.11: Breakdowns of master and worker execution time graph for fibonacci, quicksort and LU

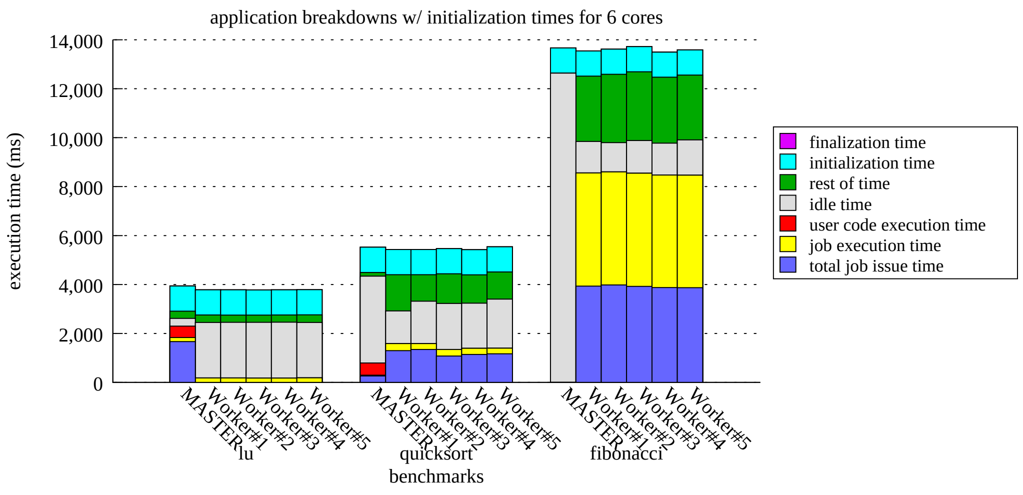

Figure 4.12: Breakdowns of master and worker execution time graph for fibonacci, quicksort and LU, with initialization and finalization times

Chapter 5

Conclusions and Future Work

In this work we presented an implementation of the futures programming model as a C++ library, for distributed and shared memory machines. We implemented our system using the MPI one-sided communication library for communication. We adopted shared memory scheduling techniques to implement our scheduler, making use again of the MPI one-sided interface. In terms of interface, we showed that it is possible to implement the shared memory C++11 standard interface. There are only minimal differences between the two interfaces, none of which limits the the expressiveness and usability of the futures model. Our performance evaluation of the system shows that the current implementation suffers from significant overheads, making it unsuitable for fine grain parallelization. For the most part, the high overhead can be mainly attributed to the serialization routines and the MPI based mutex library we use.

At this point, from our experience with the MPI one-sided communication interface, we believe that there exist some fundamental limitations in its design. These are:

1. MPI_Window creation is a collective operation over a group of MPI processes. In order to dynamically allocate data and share through a window, all processes must synchronize, calling the MPI_Window_create routine. For our asynchronous system this is a serious limitation, especially when we only want to create windows between only two processes at a time. The only solution to this problem would be to create a priori all possible groups for all pairs of processes, which can be costly. Instead, we were forced to preallocate a buffer for each process, that is shared through a window.
2. The *active mode* "epoch" definition scheme, requires both processes to take part in the communication, which we believe to be counter intuitive for an one-sided communication interface. What's more, we find that it is unusable in our asynchronous communication system.

3. The locking schematics of the *passive mode* "epoch" definition scheme, do not define well what happens when a window is concurrently accessed, which can cause erroneous results. This forced us to implement our own mutexes to synchronize data accesses on the same window. Moreover, acquiring an exclusive lock on a window will block other processes from accessing it, even if they access different, non-overlapping addresses in that window. The later constraint, limits fine grain locking. In our system, this is a very common scenario, where processes, different asynchronous *jobs*, need to write to different parts of the same window of the process owning the associated futures.
4. The lack of synchronization primitives, with the same schematics as their shared memory counterparts limits the usability of the model. Native implementations of such primitives, could offer much better performance than implementing them on top of the MPI library.

In the future we plan to address the performance issues of our system. Currently we use the Boost serialization library, which is not tuned for performance. We could try alternative serialization routine that could possibly match our needs. The overhead caused by the mutex library though, is tougher to address, while still using the one-sided communication Interface. Unless MPI will not provide a native mutex implementation, alternative one-sided communication libraries, like ARMCI, should be used. Less high profile goals include implementing all the secondary features of the C++11 futures library. We are also interested in exploring the potential of having a hybrid model with a unified interface. *Jobs* that run on the same machine will use a shared memory runtime, while *jobs* that run on different machines, will have to make use of the distributed memory runtime. Our main goal is to deliver a high performance runtime system.

Bibliography

[1] "C++ standard library:threads." [Online]. Available: http://en.cppreference.com/w/cpp/thread

[2] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998. [Online]. Available: http://dx.doi.org/10.1109/99.660313

[3] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of openmp tasks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 3, pp. 404–418, Mar. 2009. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2008.105

[4] J. Nieplocha and B. Carpenter, "Armci: A portable remote memory copy library for distributed array libraries and compiler run-time systems," in *Lecture Notes in Computer Science*. Springer-Verlag, 1999, pp. 533–546.

[5] G. Shah and C. Bender, "Performance and experience with lapi – a new high-performance communication library for the ibm rs/6000 sp," in *Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, ser. IPPS '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 260–. [Online]. Available: http://dl.acm.org/citation.cfm?id=876880.879642

[6] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: Shmem for the pgas community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ser. PGAS '10. New York, NY, USA: ACM, 2010, pp. 2:1–2:3. [Online]. Available: http://doi.acm.org/10.1145/2020373.2020375

[7] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà, "Advances, applications and performance of the global arrays shared memory programming toolkit," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 203–231, May 2006. [Online]. Available: http://dx.doi.org/10.1177/1094342006064503

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, “Cilk: An efficient multithreaded runtime system,” in *JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING*, 1995, pp. 207–216.

[9] S. Saunders and L. Rauchwerger, “Armi: an adaptive, platform independent communication library,” *SIGPLAN Not.*, vol. 38, no. 10, pp. 230–241, Jun. 2003. [Online]. Available: http://doi.acm.org/10.1145/966049.781534

[10] P. Beckman and D. Gannon, “Tulip: A portable run-time system for object-parallel systems,” in *Proceedings of the 10th International Parallel Processing Symposium*, 1996, pp. 532–536.

[11] S. S. Vadhiyar and J. J. Dongarra, “Gradsolve - a grid-based rpc system for remote invocation of parallel software,” *Journal of Parallel and Distributed Computing*, vol. 63, p. 1104, 2003.

[12] I. Foster, C. Kesselman, and S. Tuecke, “The nexus approach to integrating multithreading and communication,” *Journal of Parallel and Distributed Computing*, vol. 37, pp. 70–82, 1996.

[13] G. Tzenakis, A. Papatriantafyllou, J. Kesapides, P. Pratikakis, H. Vandierendonck, and D. S. Nikolopoulos, “Bddt:: block-level dynamic dependence analysisfor deterministic task-based parallelism,” *SIGPLAN Not.*, vol. 47, no. 8, pp. 301–302, Feb. 2012. [Online]. Available: http://doi.acm.org/10.1145/2370036.2145864

[14] J. M. Perez, R. M. Badia, and J. Labarta, “Handling task dependencies under strided and aliased references,” in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS ’10. New York, NY, USA: ACM, 2010, pp. 263–274. [Online]. Available: http://doi.acm.org/10.1145/1810085.1810122

[15] J. C. Jenista, Y. h. Eom, and B. C. Demsky, “Ooojava: software out-of-order execution,” in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP ’11. New York, NY, USA: ACM, 2011, pp. 57–68. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941563

[16] F. S. Zakkak, D. Chasapis, P. Pratikakis, A. Bilas, and D. S. Nikolopoulos, “Inference and declaration of independence: impact on deterministic task parallelism,” in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, ser. PACT ’12. New York, NY, USA: ACM, 2012, pp. 453–454. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370892

[17] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, “The influence of operating systems on the performance of collective operations at extreme scale,” in *Cluster Computing, 2006 IEEE International Conference on*, 2006, pp. 1–12.

[18] H. Philipp, P. Aleksandar, M. Heather, K. Viktor, K. Roland, and J. Vojin, "Scala documentation: Futures and promises."

[19] J. Dinan, P. Balaji, J. R. Hammond, S. Krishnamoorthy, and V. Tipparaju, "Supporting the global arrays pgas model using mpi one-sided communication," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, ser. IPDPS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 739–750. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2012.72

[20] D. Bonachea, "The inadequacy of the mpi 2.0 one-sided communication api for implementing parallel global address-space languages." [Online]. Available: http://www.cs.berkeley.edu/~bonachea/upc/mpi2.html

[21] R. T. A, G. P. B, H. M. P. C. A, and S. B. B, "Hydra-mpi: An adaptive particle-particle, particle-mesh code for conducting cosmological simulations on mpp architectures," 2003.

[22] Y. Cui, K. B. Olsen, T. H. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. K. Panda, A. Chourasia, J. Levesque, S. M. Day, and P. Maechling, "Scalable earthquake simulation on petascale supercomputers," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–20. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.45

[23] M. Adelstein-Lelbach, B. Anderson and H. Kaiser, "Hpx: A c++ standards compliant runtime system for asynchronous parallel and distributed computing," 2013. [Online]. Available: http://stellar.cct.lsu.edu/info/publications

[24] H. Kaiser, M. Brodowicz, and T. Sterling, "Parallex an advanced parallel execution model for scaling-impaired applications," in *Proceedings of the 2009 International Conference on Parallel Processing Workshops*, ser. ICPPW '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 394–401. [Online]. Available: http://dx.doi.org/10.1109/ICPPW.2009.14

[25] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," in *IN PROCEEDINGS OF THE CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS*, 1993, pp. 91–108.

[26] C. R. Houck and G. Agha, "Hal: A high-level actor language and its distributed implementation," in *ICPP (2)*, 1992, pp. 158–165.

[27] D. Bonachea, "Ammpi: Active messages over mpi." [Online]. Available: http://www.cs.berkeley.edu/~bonachea/ammpi/

[28] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," *SIGARCH Comput. Archit. News*, vol. 20, no. 2, pp. 256–266, Apr. 1992. [Online]. Available: http://doi.acm.org/10.1145/146628.140382

[29] R. Ramey and M. Troyer, "Boost seriliazation," 2002-2006. [Online]. Available: http://www.boost.org/doc/libs/1_52_0/libs/serialization/doc/index.html

[30] D. Gregor, "Boost mpi," 2005. [Online]. Available: http://www.boost.org/doc/libs/1_52_0/doc/html/mpi.html

[31] J. Dinan, S. Krishnamoorthy, P. Balaji, J. R. Hammond, M. Krishnan, V. Tipparaju, and A. Vishnu, "Noncollective communicator creation in mpi," in *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, ser. EuroMPI'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 282–291. [Online]. Available: http://dl.acm.org/citation.cfm?id=2042476.2042508

[32] A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.

[33] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[34] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Comput.*, vol. 35, no. 1, pp. 38–53, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2008.10.002

[35] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The plasma and magma projects," *Journal of Physics: Conference Series*, vol. 180, no. 1, p. 012037, 2009. [Online]. Available: http://stacks.iop.org/1742-6596/180/i=1/a=012037