A Futures Programming Model Runtime System for Distributed Memory

Dimitrios Chasapis
Laboratoire de Recherche en Informatique
Parall Group

June 5, 2013

Abstract

In this work we present a C++ library implementation of the futures programming model for distributed memory. Our implementation uses an interface similar to the the C++ 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 is currently implemented on top of 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.

Contents

1 Introduction **2**
1.1 Futures and Promises 4
1.2 Background 6
1.2.1 MPI one-sided communication 6
1.3 Related Work 7

2 Design and Implementation **9**
2.1 Futures Interface 12
2.2 Communication 13
2.2.1 Shared Address Space 14
2.2.2 Mutexes 17
2.3 Shared Memory Management 17
2.3.1 Memory Allocation/Deallocation 18
2.4 Scheduler 20
2.4.1 Alternative Implementations/Design Versatility 22

3 Evaluation **24**
3.1 Microbenchmarks 24
3.2 Real Application Benchmarking 25

4 Conclusions and Future Work **33**

Chapter 1

Introduction

High performance computing is today stronlgy related with parallel programming. On one end, computer architectures have been developing parallel machines or netword 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.

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 transfered among threads but can lead to race conditions when two threads try to access the same data at the same time or, 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 [1] are two commonly used libraries that are used to program threads on shared memory machines. With Pthreads the user can create and launch and threads, where each thread will have a specific work to do. The library also offers a variaty 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 [2], 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 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 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 [3] LAPI [4] and MPI-2 provide library implementations of such one-sided communication interface. OpenSHMEM [5] is an effort to standardize the SHMEM. 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 emerge 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 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 ontop of a one-sided communication library. Global Arrays [6] is also an API that follows the PGAS model and is built ontop on ARMCI [3].

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 [2, 7] in the shared memory environment and to similar models in distributed memory like Remote Procedure Calls (RPC) [8, 9, 10] or Remote Service Request (RSR) [11]. 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 the synchronization problems, a lot of systems have been suggested in the literatture, 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 [12, 13] and/or have the compiler statically infer dependendencies among tasks [14, 15]. This scheme usually allows the programmer to describe the data-flow relations between different parallel tasks, and an underlying runtime systems will explicitly synchronise them. The drawback here is that there is usually an additional overhead from the runtime system and/or the automatic (dynamic or static) analysis used to automatically synchronise the code, is often conservative in order to maintain correctness, which harms performance.

1.1 Futures and Promises

The futures (or promises) programming model, tries to simplify the daunting task of synchronizing accesses to shared data. In this model, all data shared between threads should be encapsulated in a special variable called a future. A future should be paired with another special variable called a promise. The future variable can be used to access the shared data while the promise variable to write to the data. When a thread accesses the future variable, it will either retrieve the encapsulated data or block and wait for it to become available, depending whether the data has been set using the associated promise variable. The promise variable can be used by another thread or the same thread that will access the future value. Other future interfaces, following the task programming model paradigm, allow the issuing

of functions asynchronously, so that they are run concurrently with the calling thread. In these model, the return value of the function is in fact a future and a promise pair. The thread that executes the issued function, will set the promise and the thread that issued the function will retrieve the return value using the future variable. We believe the the futures programming model is a reasonable compromise between having an easily programmable environment and the ability to efficiently express parallel algorithms.

The futures model is traditionally implemented using threads over shared memory environments. HPX [16] is a runtime system that offers an implementation of the futures model for both shared and distributed memory environments. In this work we aim to provide an implementation of the futures interface as it is defined in the standard C++ library [17], but for distributed memory environments. We have chosen to build our system using the MPI-2 one-sided communicaton 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 available library on distributed and shared memory machines alike. The contributions of this work can sum up to:

- Implementation and evaluation of a runtime library of the futures programming interface for distributed memory.
- 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 runtime is only able to offer some speedup only when we use coarse grain tasks, due to the high cost of issuing functions asynchronously, communication and using locks implemented over an one-sided communication interface. Moreover, MPI-2 one-sided communication interface is not as versatile as we would like, especially due to the fact that it can only expose data from one process to another through collective operations.

1.2 Background

1.2.1 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 some of theirs local address space to all other processes. MPI_Windows are created using the MPI_Win_create function. This functions requires a pointer to a local address space and the size of the data to be shared, through 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. Both operations are applied on an MPI_Window and the rank of the process, whose address space we want to remotely write to or read from. In addition, a buffer has to be supplied to each function, that either points to the data that needs to be written to the remote address or to the local memory that the remote data will be stored to. Along with the buffers, an MPI_Datatype and buffer size must be supplied. 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, thus 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/complete. 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 or concurrent operations to take place in the same "epoch", while the exclusive will force them to happen in different "epochs".

1.3 Related Work

Dinan et al [18] have implemented the Global Arrays (GA)[6], a PGAS model, over the one-sided communication interface of MPI. In their work they ported GA's low-level ARMCI [3] one-sided communication librari using the MPI API. 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 [19] also supports that the MPI one-sided interface is not fit to be used for the implementation of PGAS languages. There are however examples [20, 21] where MPI's one-sided interface has been succesfully used to implement high-performance applications.

High Performance ParalleX (HPX) [16] is a parallel runtime system implementation of the ParalleX[22] execution model. One of ParalleX's many features is the futures synchronization model. The adopted futures interface is similar to the C++ standard library one and is available for both shared and distributed memory. In contrast with our work, it does not use an MPI library for communication, and is instead an alternative to the MPI model.

Other high-performance systems that support Remote Method Invocation (RMI), RSR and RPC share similar specifications with our runtime system. LAPI[4], Charm++[23], ARMI[8], which support RMI, are required to perform remote operations asynchronously. They however use two-sided communication message passing libraries, thus employing different techniques than we do, in order to achieve asynchronous communcation. The RSR sheme from Nexus[11] also operates over a two-sided communication interface and employes the same techniques with the aforementioned runtimes. 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 is also similar to the RPC model. AMMPI[24] is an Active-Messages implementation over MPI two-sided communication interface. The techniques used by these systems to provide asynchronous communication are briefly described in section 2.4.1

Chapter 2

Design and Implementation

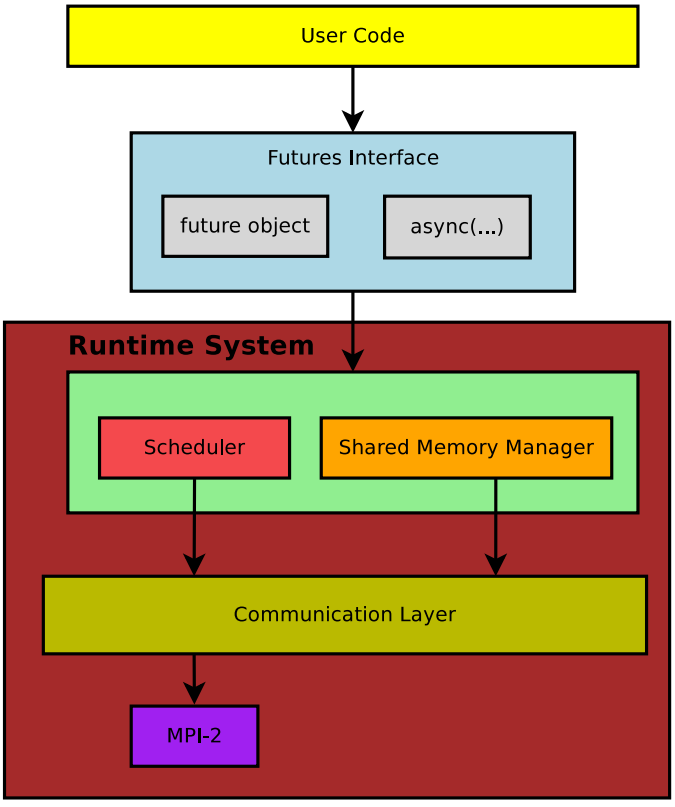

Figure 2.1: Overview of our futures library design.

We have modeled the distributed futures, as closely as possible, to the

shared memory futures from the C++ standard library. Parallelism in the future model is extracted by issuing functions asynchronously, using a *special* async function that takes the function to be executed in parallel as an argument (see section 2.1). In our library, functions called using the async function, referred to as *jobs* from now on, are send through a communication library to be executed by other available processes. The decision of who executes a *job* is made by a scheduler. The host process can retrieve the return value of a *job* using the future object associated with the corresponding *async* call. A process can only retrieve a future's value if the job has finished execution, in which case the remote process will set the futures value on the host process, else the host process will will block. In our implementation, a blocked process will try to execute any pending *jobs* it might have while it is beeing blocked. We designed our system 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 2.1 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 an system environment, an instance of which is present at every process. Note that it is not necessary however for every environment instance to be the same, for example process 1 can be only responsible for it's own shared address space and be aware only of the shared addresses of other processes, which hold futures associated with *jobs* that are supposed to run by process 0.

Figure 2.3 shows the program flow for processes 0 and 1, of the simple hello world example in figure 2.2. 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

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

Figure 2.2: A simple hello world implementation using the distributed futures interface. The output of the program on process 0 would be "- Master :Hello 1".

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 retrive the message value or block untill 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 return to 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.

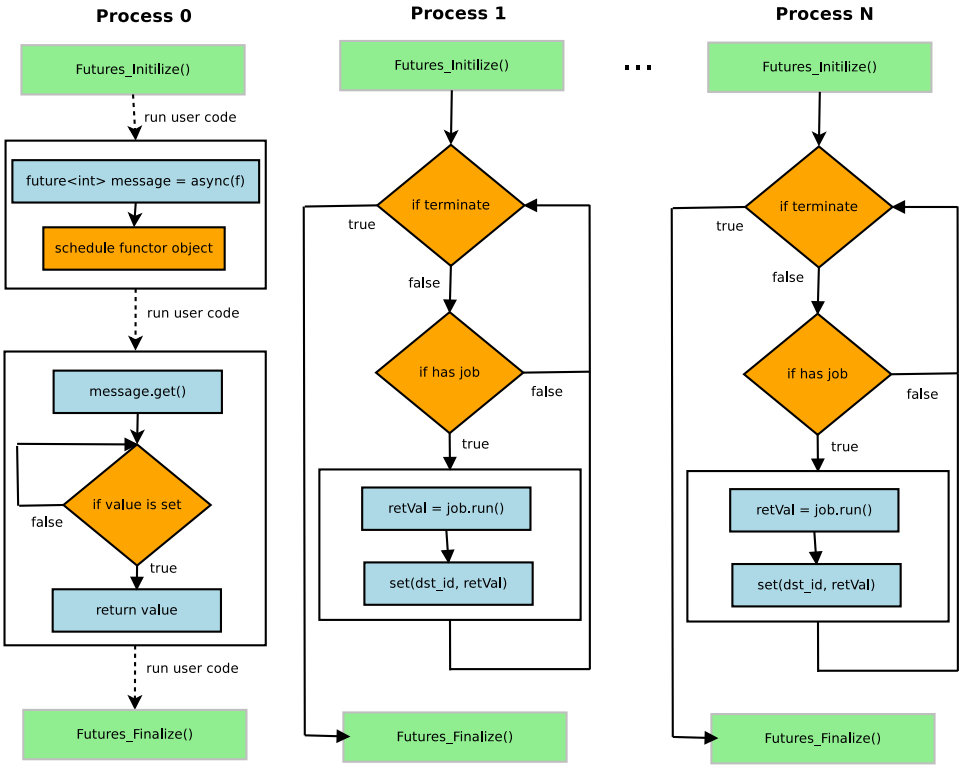

Figure 2.3: The control flow of the hello world program in figture 2.2.

In the rest of this chapter, we will present the future interface in section 2.1 and discuss our implementations of the the communication, shared memory manager and scheduler modules in sections 2.2, 2.3 and 2.4 respectively.

2.1 Futures Interface

We replicate the futures interface from the C++ standard threads library, with one major difference being that the the function being called must be a functor object. Figure 2.5 shows a recursive implementation of the fibonacci function using our future implementation. The user needs to create a functor, which must be serializable, and use the macro FUTURES_EXPORT_FUNCTOR(async_function<fib,

int>) to expose the functor object to the underlying serialization library [1]. Moreover, the user can use the FUTURES_SERIALIZE(F) macro, where F is a functor object, which will create the necessary serialization routines automatically, but it is not recommended if the functor object has members (see [25] for more details on how to serialize a C++ object with Boost serialization library). Note that the argument to the first macro command is always *async_function<F, Args...>*, where F is a functor class and Args are an aribary number argument types, that are to be passed to the overloaded call method of the functor F. *Async_function* is a special class that wraps around the functor objects and its arguments. Instantiation of this class, using C++ templates meta-programming capabilities, generates the appropriate routines, for setting the future's value, according to the value's type. It also facilitates all necessary information that are needed to be transfered to the worker process. Figure 2.4 shows the definition of the *async_function* class.

A call to the *async(F, Args...)* function, where F is a functor object and Args is any number of arguments, will send the functor object to an available process or execute the functor directly, if no such process is found (see 2.4 for details). The async function returns a future object which can be used by the process that called the *async* function to retrieve the functor's return value. If the return value is an array, a pointer or any other form of container, the user should instead call a variation of the *async* function, *async(N, F, Args...)*, where N is number of elements that will be returned. In order to retrieve the value, the owner of the future needs to call the get() method. This method is blocking, so calling it will cause the process to block until the value of the future becomes available. Alternatively, the future owner can call the is_ready() method, which is not blocking, to check if the value can be retrieved, 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.

2.2 Communication

The communication module is responsible for message exchange between all of the processes in our system, as well as providing the infrastructure for

[1] We use the boost serialization library [25] and the input/output archives from the boost mpi library [26]

```
1
2  template<typename F, typename... Args>
3  class async_function : public _job {
4  ... //we have ommited here all the serialization routines
5  public:
6      int src_id;
7      int dst_id;
8      Shared_pointer ptr;
9      int data_size;
10     int type_size;
11     F f;
12     std::tuple<Args...> args;
13     typename std::result_of<F(Args...)>::type retVal;
14     async_function();
15     async_function(int _src_id, int _dst_id,
16                    Shared_pointer _ptr,
17                    int _data_size, int _type_size,
18                    F& _f, Args... _args);
19     ~async_function();
20     void run();
21 };
```

Figure 2.4: 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.

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.

2.2.1 Shared Address Space

In our implementation, the underlying message passing library used is MPI-2, thus we use MPI windows to expose such space 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,

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

Figure 2.5: A fibonacci implementation using the distributed futures interface

and then exposed to other processes through creating an MPI window that is correlated to the pointer with MPI_Create_Win (See section 1.2). 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 [27], 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, 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 [27, 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 a address space shared though an MPI window, using the special Shared_pointer construct (see section 2.3). This pointer keeps

```
1  void set_data(void* val, int dst_id, Shared_pointer ptr,
2              Datatype datatype) {
3
4      MPI_Win_lock(MPI_LOCK_EXCLUSIVE, dst_id, 0,
5                shared_space[ptr.page_size]);
6
7      MPI_Put(val, ptr.size, datatype, dst_id, ptr.base_address,
8             ptr.size, datatype, shared_space[ptr.page_size]);
9
10     MPI_Win_unlock(dst_id, shared_space[ptr.page_size]);
11 };
12
13 void set_data(boost::mpi::packed_oarchive& ar, int dst_id,
14              Shared_pointer ptr) {
15
16     MPI_Win_lock(MPI_LOCK_EXCLUSIVE, dst_id, 0,
17                shared_space[ptr.page_size]);
18
19     MPI_Put(&ar.size(), 1, MPI_INT, dst_id, ptr.base_address,
20            1, MPI_INT, shared_space[ptr.page_size]);
21
22     MPI_Put(ar.address(), ar.size(), MPI_PACKED, dst_id,
23            ptr.base_address+DATA_OFFSET,
24            ar.size(), MPI_PACKED, shared_space[ptr.page_size]);
25
26     MPI_Win_unlock(dst_id, shared_space[ptr.page_size]);
27 };
```

Figure 2.6: 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.

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 2.6 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 **??** 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.

2.2.2 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 tareget process. We have adopted MPICH's implementation of mutexes in our design. A mutex is a shared vector through an MPI window. Each vector element is a byte value corresponding to one process. When a process wants to hold the mutex lock, it sets its vector value at one and iterates through the rest of the vetor to check if another process wants or has acquired the lock. If the lock is acquired or another process waits for it, then the current process blocks until it receives a message. When unlocking, a process sets its vector value to zero and then iterates through the vector to find and send a message to next process that is waiting to acquire the lock.

2.3 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 [28, p. 185-187]. However, we keep different free lists for different page sizes to deal with memory segmentation. Figure 2.7 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 2.7 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 [2] (see section 1.2).

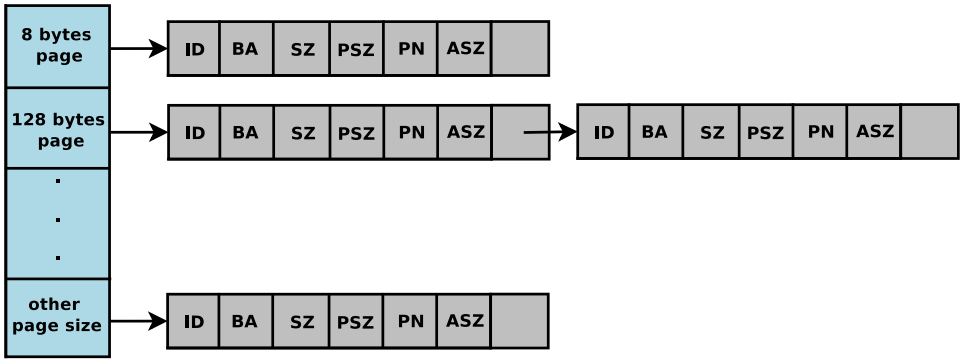

Figure 2.7: Shared Memory Manager keeps a map of free lists, indexed by the page size. For page size that do not match any predefined ones, we use the *other* page size free list.

2.3.1 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 uses the shared memory manager. The shared memory allocator tries to find the best page size fit for the data size, and searches the corresponding free list, using a first fit algorithm to find a large enough space for the new data. If no fitting page size is found then the allocator uses a special freeList, which does not use a predefined page size, but instead uses the data size to find free space. If not enough free space is found in the correct free list, then the allocator can try to find data in another free list, of different page size. Figure 2.8 shows a free list, before and after allocating a data object. The first fit algorithm will iterate the list

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

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 2.8, 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.

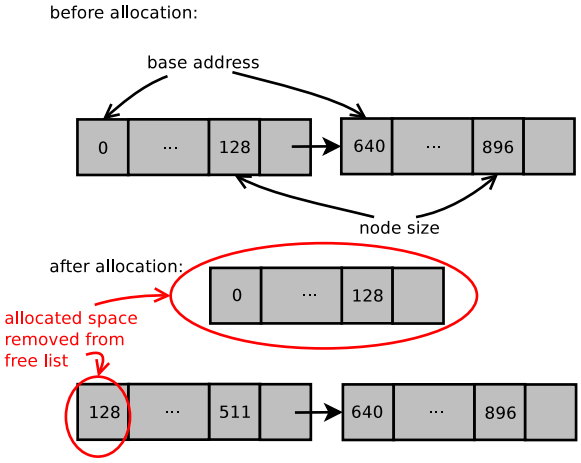

Figure 2.8: During allocation, when a large enough space is found, the allocated page is removed from the node.

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 2.9 shows a free list, before and after freeing some shared memory. Because freeing 128 bytes at base address 512 creates a continueous space from byte 0 to byte 640, the two list nodes will be merged into one.

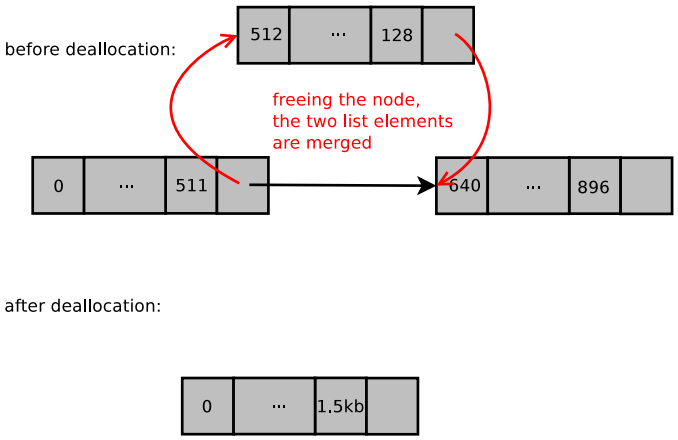

Figure 2.9: By freeing data at base pointer 512, creates a continuous space between base pointer 0 and base pointer 640, causing the list nodes to merge into one.

2.4 Scheduler

In order to have a distributed memory interface similar to the shared memory one, we chose to implement a scheduler, which is responsible for deciding who will execute which *job*. If the user was responsible for distributing *jobs* among the processes, he would need to reason about dependencies between *jobs* and retrieving future values, else the program could easily end up in a deadlock. To make our case clear, consider the fibonacci example in figure 2.5. In our example, let's say we have 3 processes, one of them is the master process. We need to run fib(3), thus, process 0, the master process can issue async(f, 2) to process 1 and async(f, 1) to process 2. Process 1 can issue async(f, 1) to process 2 and run async(f, 0) on itself. In this scenario the program will execute correctly without any problems, since when any of the processes call a get() will either retrieve or wait for the value. But consider we want to compute fib(5). Process 1 may have to run async(f, 4) while process 2 will have to run async(f, 3). At some point, process 1 issues async(f, 3) to process 2, while process 2 issues an async(f, 2) to process 1. Both processes will return from the async calls and proceed calling get() to retrieve the value but will actually block forever, since neither process will run the pending fibonacci functions. This scenario is not a problem if processes are dynamically spawned, but if we have a static number of process, which is common for mpi programs, we need to address such issues.

Since it is not always trivial to reason about such dependencies, we have

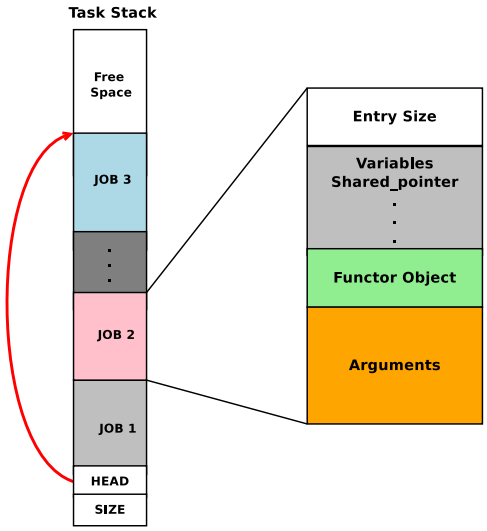

Figure 2.10: Shared stack where a worker process keeps its pending jobs. Entries can have varied sizes, this size is stored at the beginning of the entry and can be used to retrieve the corresponding job. Information for the specific stack,like size and head, are stored at the beginning of the shared space, so that other processes can access them.

implemented our own *job* scheduler. We use MPI-2's one-sided communication library (via the communication module) to implement task stacks, similar to their shared memory counterparts. We choose to implement a stack because it suits better future logic, we need to execute the latest issued *job* in order for the get() not to block indefinitely in recursive algorithms. Using one-sided communication, only the issuer needs to copy the functor object to the workers stack, as in a shared memory environment. Figure 2.10 shows how a task 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 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 [3]. Moreover, at the beginning of the shared space, the size and current head values are stored, so all processes can push *jobs*.

Figure 2.3 shows a control flow graph for the master and worker processes. The master simply initializes the futures environment and issues

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

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 process. The master will check the status of all the processes and if all of them are in idle status, he will set the terminated flag to true on all of them. If a process is still busy, meaning executes some *job* or has still pending *jobs* in its stack, the master must wait till all jobs are finished and then set the terminated flags.

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

2.4.1 Alternative Implementations/Design Versatility

We have designed the system to be able to use alternative scheduler implementations. However, all must meet the following interface requirements.

- Implement routines to send and receive jobs. The call to the *async* function should be able to send a *job* to the worker process without blocking. The worker should be able to handle multiple *jobs*, without breaking the asynchronous model. Moreover, *communication* should only require the process that issued the *job* and the one that receives it to participate.
- Implement a routine which returns a valid process id that is available to run a *job* issued by call to *async*.

- Implement the terminate routine, which should return true for a process to exit and should also not block execution if it does not return true.

Remote Procedure Call (RMC) and Remote Method Invocation (RMI) libraries often have the above requirements as well. 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 [9, 8, 11, 29] and hardware interrupts. In our system, polling would require a worker process to poll for incoming *jobs* 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 we also need a process to poll frequent enough to ensure timely responses [8, 4]. 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. [8, 4, 11, 29]. The Nexus system [11], 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 [11].

Chapter 3

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@2.40GHz with 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 optimizations enabled. For the MPI library we used OpenMPI version 1.4.3.

3.1 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 3 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 3.1 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 transfered, have minimal implact on execution time. Figure 3.2 shows the execution time of issuing functor objects with different vector argument sizes. In all cases the total number of elements totals to 1200000 (e.g. 1 vector of 1200000 elements or 4 vectors of 300000 elements). The numbers reported are the median values of 20 runs, and by manual observation we can verify that the numbers have been consistent. As of now, we have no insight as to why having 2 vector objects is more cost efficient than having only one and the differences cannot be considered "noise". In figure 3.3 we can observe how the size of the arguments can affect execution time. We see that the size of the arguments (here the size of 1 vector object) can exponentially increase execution time. As expected, the size of the arguments is the factor that can influence performance the most.

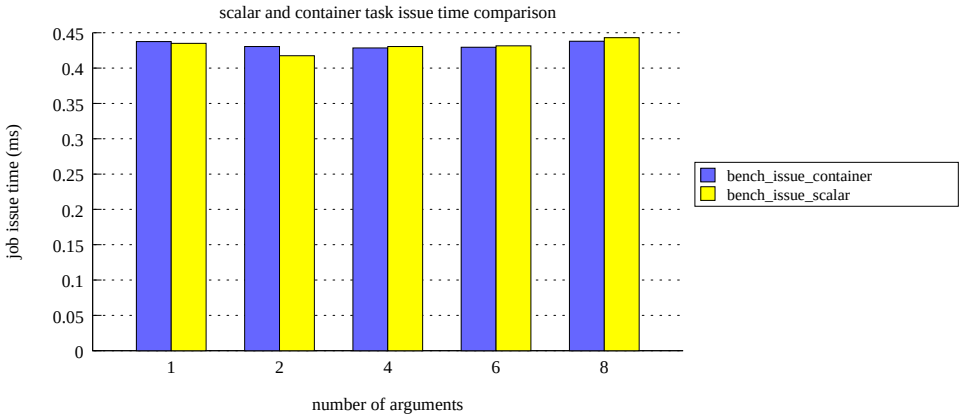

Figure 3.1: Comparison between issuing functors with scalar arguments versus vector objects of size 1

3.2 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 2.5 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 verson of the code for values smaller than 30, so

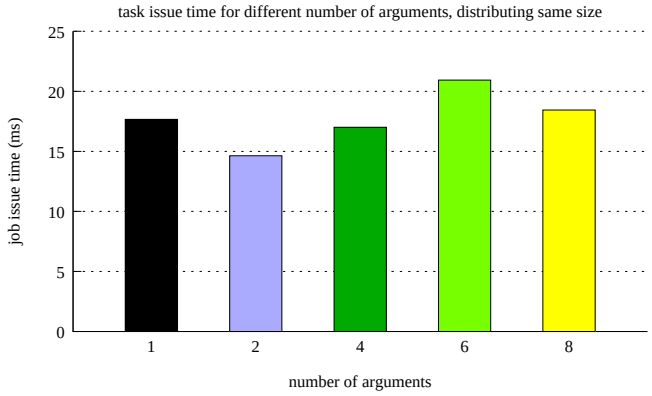

Figure 3.2: Comparison between issuing functors with different number of vector arguments, but total size of arguments is the same in all cases.

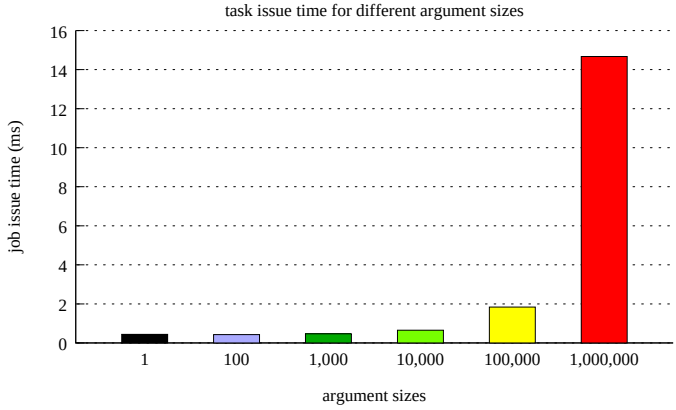

Figure 3.3: Comparison between issuing functors with 1 vector argument of different sizes

that each async function can have some amount of work. We run the fibonacci function with 45 as an argument.

Quicksort: Figure 3.4 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 asynchrous 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 [30]. Figure **??**lst:tiledLUseq) 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 emphA[k][n] and array emphA[m][n]. For the kernels *dgetrf*, *dgessm*, *dtstrf* and *dssssm*, we use the implementation found in the Plasma project [31]

Figure 3.6 shows a simplified version of our parallel implementation. 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 currenlty 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 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.

```
1  /* a sequential qs */
2  void QsSequential(vector<double>& array, const long left, const long right){
3      if(left < right){
4          const long part = QsPartition(array, left, right);
5          QsSequential(array,part + 1,right);
6          QsSequential(array,left,part -1);
7      }
8  }
9
10 /** A task dispatcher */
11 class quicksort {
12 public:
13   quicksort() {};
14   ~quicksort() {};
15   vector<double> operator()(vector<double> array, const int deep) {
16     const int left = 0;
17     const int right = array.size()-1;
18     if(left < right){
19         if(array.size() > min_unit) {
20             const long part = QsPartition(array, left, right);
21             vector<double> subarrA((right)-(part+1)+1), subarrB(part-1-left+1);
22             Copy(subarrA, array, part+1, right+1);
23             Copy(subarrB, array, left, part);
24             quicksort qsort;
25             future<vector<double> > res1, res2;
26             res1 = async2(subarrA.size(), qsort, subarrA, deep-1);
27             res2 = async2(subarrB.size(), qsort, subarrB, deep-1);
28             subarrA = res1.get();
29             subarrB = res2.get();
30             Merge(array, subarrB, subarrA);
31         }
32         else {
33             const long part = QsPartition(array, left, right);
34             QsSequential(array,part + 1,right);
35             QsSequential(array,left,part -1);
36         }
37     }
38     return array;
39   }
40 };
```

Figure 3.4: A quicksort implementation using the distributed futures interface

In figure 3.7 we report the execution times for running the three applica-

```
1  for(int k = 0; k < TOTAL_TILES; k++) {
2    dgetrf(A[k][k], P[k][k]);
3    for(int n = k+1; n < TOTAL_TILES; n++) {
4      dgessm(A[k][n], A[k][k], P[k][k])
5    }
6    for(int m = k+1; m < TOTAL_TILES; m++) {
7      dtstrf(A[k][k], A[m][k] , P[m][k]);
8      for(int n=k+1; n < TOTAL_TILES; n++) {
9        dssssm(U[k][n], A[m][n], L[m][k], A[m][k], P[m][k]);
10     }
11   }
12 }
```

Figure 3.5: The tiled LU kernel implementation

```
1  for(int k = 0; k < TOTAL_TILES; k++) {
2    A[k][k] = cpldAU[k][k].get().A;
3    dgetrf(A[k][k], P[k][k]);
4    for(int n = k+1; n < TOTAL_TILES; n++) {
5      A[k][n] = cpldAU[k][n].get().A;
6      fA[k][n] = async(dgessm, A[k][n], A[k][k], P[k][k]);
7    }
8    for(int m = k+1; m < TOTAL_TILES; m++) {
9      A[m][k] = cpldAU.get().A;
10     dtstrf(A[k][k], A[m][k].get() , P[m][k]);
11     for(int n=k+1; n < TOTAL_TILES; n++) {
12       if(m == k+1)
13         A[k][n] = fA[k][n].get();
14       else
15         A[k][n] = cpldAU.get().U;
16       A[m][n] = cpldAU.get().A;
17       cpldAU[m][n] = async(dssssm, A[k][n], A[m][n], L[m][k], A[m][k], P[m][k]);
18     }
19   }
20 }
```

Figure 3.6: The tiled LU parallel kernel implementation

tions on the machine setup we described in section 3. 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 on quicksort and Tiled LU, on the contrary we get a slowdown (figure 3.8). We get a small speedup in Fibonacci when using more than 4 processes. We also notice a strange behaviour of the runtime when running our applications on

very few cores. In these cases we have noticed an increase in communication when the worker try to return the future value, but we have yet to identify the cause of that issue. In figure 3.9 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 and time spend on serialization of the *job* object and sending it to the worker. *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. *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 time can be for example the time needed to send the return value of the asynchronous execution of a *job* by a worker. In all applications we see that the either *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*. These overhead times are too great compared to the actual work done by any of the processes, as a result our runtime is not fit to run problems with fine grain granurality. At this stage it is unclear whether the issue is the MPI library or the implementation. We will further investigate the current issues. Moreover, figure 3.10 shows the same breakdowns with figure 3.9, 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.

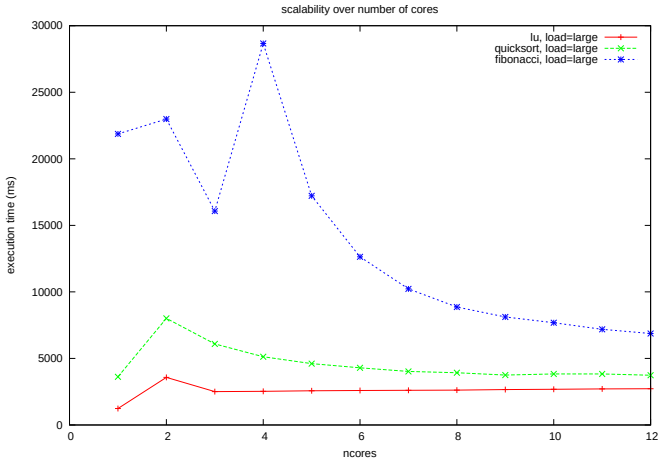

Figure 3.7: Scalability graph for fibonacci, quicksort and LU

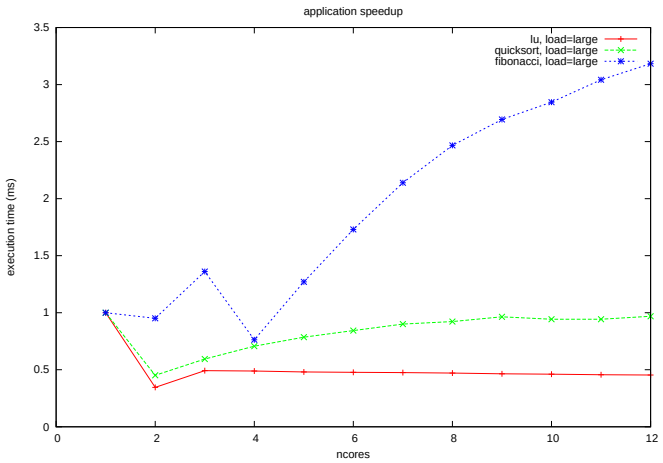

Figure 3.8: Speedup graph for fibonacci, quicksort and LU

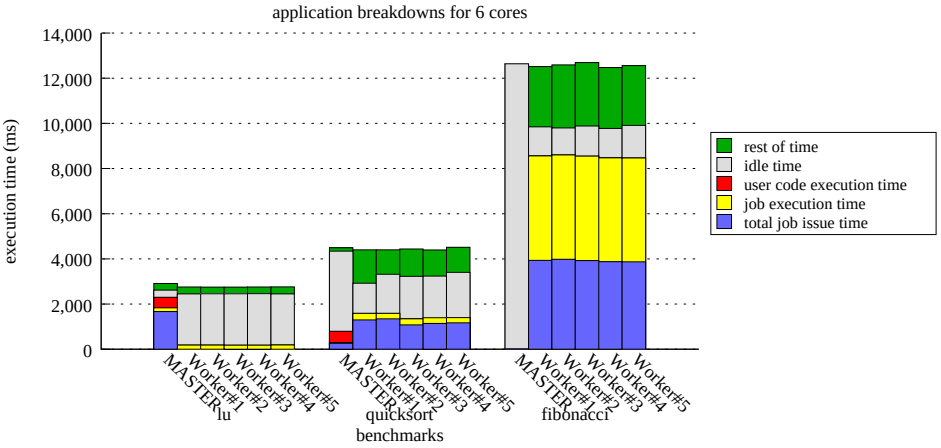

Figure 3.9: Breakdowns of master and worker execution time graph for fibonacci, quicksort and LU

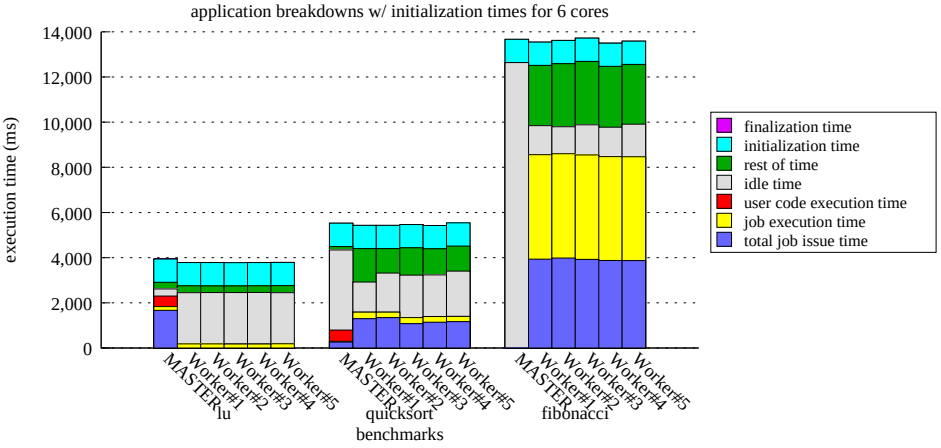

Figure 3.10: Breakdowns of master and worker execution time graph for fibonacci, quicksort and LU, with initialization and finalization times

Chapter 4

Conclusions and Future Work

In this work we presented an implementation of the futures programming model as a C++ library, for distributed memory machines. We implemented our system using the MPI one-sided communication library for communication and we adopted shared memory scheduling techniques to implement our scheduler, making use again of the MPI one-sided interface. Our evaluation of the system shows that the current implementation suffers from significant overheads, especially when issuing asynchronous *jobs* on processes. In order to be usable, the user must issue coarse grain *jobs*. At this point, we are inconclusive whether the large overhead can be attributed to the MPI library or to other implementation issues in our runtime system (e.g. scheduling policies, mutex implementation). We suspect however, that busy waiting, the technique we use to check for future availability, even on local variable that are shared through MPI windows can be costly (acquiring locks on an "epoch" for example).

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.

In the future we plan to further investigate the cause of our systems high overhead. If our suspicions are correct and the cause of the overhead is our scheduling scheme, we will try different implementations for the scheduling and communication layers. We would also like to explore the potential of having a hybrid model, where *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 focus will be to deliver a high performance runtime system. We also plan to add important features to the runtime, such as the ability to serialize and send futures to other processes. This will be very useful in applications like our Tiled LU, where the master issues all the async *jobs*, but instead each process will wait for it's futures to be available. This will allow finer grain synchronization and the master will not become a bottleneck.

Bibliography

[1] 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

[2] 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

[3] 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.

[4] 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

[5] 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

[6] 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

[7] 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.

[8] 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

[9] 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.

[10] 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.

[11] 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.

[12] 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

[13] 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

[14] 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

[15] 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

[16] 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

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

[18] 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

[19] 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

[20] 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.

[21] 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

[22] 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

[23] 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.

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

[25] 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

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

[27] 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

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

[29] 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

[30] 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

[31] 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