Distributed Functional Programming in Scheme

by

Alex Schwendner

S.B., Massachusetts Institute of Technology (2009)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2010

Author .....
Department of Electrical Engineering and Computer Science
May 21, 2010

Certified by .....
Saman Amarasinghe
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

Distributed Functional Programming in Scheme

by

Alex Schwendner

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2010, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, we present a framework for writing distributed computer programs in Scheme using simple "future" semantics. This allows a Scheme program originally written for execution on a single computer to be run in a distributed manner on a cluster of computers with very little modification to the program. The structure of the computation can be extremely general and need not be specified in advance. In order to provide this programming environment, we implemented a system of job execution servers which transparently execute arbitrary Scheme code written in a functional or mostly functional style. The use of a functional, side effect free style of programming simplifies the challenges of data consistency. A number of demonstrations of the system are presented, including a distributed SAT solver. The performance of the system is shown to increase roughly linearly with the number of servers employed.

Thesis Supervisor: Saman Amarasinghe
Title: Professor of Computer Science and Engineering

Acknowledgments

First of all, I would like to thank my thesis supervisor, Saman Amarasinghe, for his patience and guidence. I would also like to thank my academic advisor, Professor Peter Szolovits; Anne Hunter; Vera Sayzew; and Professors Rob Miller and Srini Devadas for their help in Course IV. A number of my friends were kind enough to talk about my thesis with me, including David Benjamin, Anders Kaseorg, and many others, and for that I am immensely grateful. I would like to thank all of my friends who have made my time at MIT so special. Above all, I thank my parents, for everything.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Getting better performance for computationally challenging tasks is getting progressively more difficult. Microprocessors have stopped getting faster at executing single-threaded code on a single computer. While computational power still increases, it does so with more cores per computer chip and with cheaper hardware. Thus, to benefit from advances in computing, we must use approaches which allow us to use multiple cores and multiple computers in a distributed manner.

Of particular importance is developing applications that leverage commodity hardware to solve problems too large for a single computer. Progressively cheaper hardware has made this a very attractive approach. Moreover, with more computation moving to cloud computing environments, it is becoming ever easier (and more necessary) to take advantage of large amounts of commodity hardware.

Yet, developing distributed programs is difficult. Better tools and frameworks are needed to enable the development of distributed software applications. While paradigms are emerging for parallel, shared memory architectures [BJK+95], programming for distributed, private memory environments remains challenging. Especially challenging are the heterogeneous environments presented by cloud computing. Computer and communication parameters vary between cloud computing environments. Thus, what is needed are better ways to abstract the complexities of heterogeneous cloud computing environments and present powerful abstractions for distributed computation.

In this work, we present a solution based on functional programming. We show how functional programming can be used to easily write distributed programs using clean, natural semantics. We demonstrate an instantiation of this idea in the functional programming language Scheme by providing a framework of distributed Scheme computation servers which can run very general Scheme code with minimal modification. We provide examples of our system, showing how natural it is for writing distributed programs.

Before describing our work in more detail, we first further motivate our project by characterizing existing approaches and considering how functional programming is uniquely suited to writing distributed systems.

1.1 Existing Approaches

Two general approaches have generally been used to write distributed programs. The first is to analyze the specific problem to be solved and to write a bespoke system to break up the problem into subtasks and to distribute, solve, and combine the subtask results. This can yield good results for a specific problem, but requires a large amount of code to be rewritten for each distributed program. This makes the creation of such a distributed program a time consuming and difficult process which is only attempted by experienced programmers. Examples of this approach include [CW03] and [FMM91]. While tools — such as the Message Passing Interface — exist which aid the development of such applications, they still view a distributed application as a collection of individual programs rather than as a unified whole. For specific problems of high importance, this approach may be acceptable, but it is not a practical way for most programmers to efficiently write distributed applications.

The second approach is to restrict the structure of programs which can be written in such a way as to pigeonhole distributed algorithms into a small class of allowed solution techniques. Perhaps the best known example of such an approach is MapReduce [DG04]. Another example is [IBY+07]. While such approaches have been successful at streamlining the programming of a large class of distributed computations, their restrictiveness and semantics leave something to be desired. On the one hand, such approaches generally force programmers to rigidly define the structure of their computation in advance, something which may be possible for some types of problems but not for others. On the other hand, it forces programmers to write code in a very different style than they are used to. These systems are often awkwardly embedded in a host programming language. Idioms which are common in the host certain programming language lead to syntactically valid but semantically broken programs when executed in the distribution framework.

1.2 The Promise of Functional Programming

Our approach to distributed computation also restricts the style of allowed programs, but it does so in a natural way which is already familiar to many programmers, namely functional programming. Programming in a functional style without mutation or side effects simplifies distribution semantics. No longer is data consistency a problem: if there is no mutation, then data values cannot change and consistency is automatic. Data may be freely copied between computers and computations may be performed anywhere.

Additionally, programs written in a functional programming language, such as Scheme, can be statically checked for safety by simply removing mutation primitives from the programming environment. Programs which compile in such a setting are thus purely functional and guaranteed to behave similarly in a distributed setting as on a single computer. Alternatively, programs can be written using limited use of mutation and side effects so long as it is understood that such side effects only apply to local variables, not to global variables shared between subproblems.

Prior examples of using functional programming to simplify distributed computation

include Glasgow Parallel Haskell [AZTML08] and Kali Scheme [CJK95].

1.3 Future Semantics for Distributed Computation

We wish to use functional programming to provide as natural a set of semantics as possible. The semantics which we have chosen for are work are those of *futures*. A future represents a value which will be needed at some time in the future. The future may potentially be evaluated before it is needed. The creation of a future spawns a task or a job for the evaluation of the future's value. In a single-threaded context, this job will be lazily executed when its value is needed. In a parallel or distributed context, the job may be started immediately, or at least before its value is required, if sufficient computing resources are available.

Future semantics fit naturally with functional programming. A number of functional programming languages have implementations of futures for parallel, shared memory computation, including PLT Scheme [Mor96] [Swa09] and Haskell [HMJ05]. As an example of the semantics of futures, consider the example of computing the Fibonacci numbers shown in Figures 5-1 and 5-2. Figure 5-1 shows a trivial Scheme function to compute the Fibonacci numbers in a single-threaded manner. Figure 5-2 shows the same program rewritten to use futures.

1.4 Our Work

In the subsequent chapters of this document, we present our work. First, in Chapter 2, we build a distributed variable abstraction suited to functional programming which we will find useful in building our system. This distributed variable abstraction, which we call a `dbox`, represents a single value which may be read from any computer and written to from any computer, but which may only be written to once. (In Section 2.4, we relax this restriction somewhat.) Representations of this variable are serializable and may be transfered from computer to computer just like ordinary data.

Then, in Chapter 3, we show how to build a framework of Scheme computation servers implementing our intended semantics. We explain how to serialize functions and computational tasks and transfer them transparently between computers. We then build a task management framework for executing jobs, performing job context switches, and transferring jobs to different computers. We also explain how we use work stealing to implement a simple protocol for balancing load among multiple computers.

Lastly, we review applications of our system and provide some data on its performance.

1.5 Contribution

Our contribution, in this thesis, is that it is, in fact, possible to define simple yet powerful semantics for distributed computation. That, more than anything else, is our message.

Whether or not future systems resemble ours, we hope that paradigms are developed which make programming distributed applications elegant.

Our other main message is that functional programming is a useful way of thinking about computation, and distributed computation in particular. Functional programming vastly simplifies many of the complexities associated with writing distributed applications. While requiring functions to be side-effect free and variables to be immutable may seem restrictive to some, functional programming has already proven to be an extremely successful paradigm for writing applications on a single computer. It is even more powerful for writing distributed applications.

More specifically, we provide a particular model for distributed functional programming centered on the notion of functional *futures*. This model is simple and easy, yet powerful enough to describe the parallelism demanded. We instantiate this model as a framework of Scheme computation servers evaluating dependent tasks. We shall see how easy it is to write an effective distributed program using such a system.

Chapter 2

Distributed Inter-process Communication

When a job is created, a `future` object is returned. Since this job might be evaluated on a different computer, we needed a way to return the result of the job. Moreover, the `future` might be passed inside an argument to the creation of a new job. This means that we needed to be able to serialize the `future` and copy it to another computer, and that any computer with the future should be able to get the result of the job once it has been evaluated.

This section describes how we supported these requirements with our `dbox` abstraction. The requirements themselves are detailed in Section 2.1. In Section 2.2, we describe `mvars`, semaphore-protected variables for safe concurrent access on a single computer. Lastly, we present the implementation of our `dbox` distributed variable abstraction in Section 2.3.

2.1 Requirements

We specifically needed a kind of distributed variable with the following semantics:

- The variable descriptor can be serialized on one computer, transfered to another computer, and deserialized, yielding an equivalent descriptor.
- The variable can be read from any computer which has a descriptor for the variable. If no value has yet been assigned to the variable, the read can be performed with either blocking or polling semantics.
- The variable may be used as an event, notifying user code when the variable has a value. This allows programmers to implement event-driven rather than polling semantics.
- The variable can be written to from any computer which has a descriptor for the variable. Only one distinct value may ever be written to the variable. If multiple different values are written, the variable may become inconsistent across different computers.

2.2 Local Synchronized Variables

We implemented an `mvar` abstraction in Scheme, based on Haskell's `MVar` [PGF96]. An `mvar` may be thought of as a box, which may be either empty or full. An empty box may be filled with a value, or a full box may be emptied, returning a value. The `mvar` is protected by two semaphores, one signalling the box as full and the other signalling the box as empty. See Figure 2-1. An `mvar` may be read from or written to multiple times in multiple threads, but only on the same machine. An `mvar` is not serializable.

```
(define-struct mvar
  ([value #:mutable] ; :: any
   ; The value of the mvar, if full, or some arbitrary value, if
   ; empty.
   full  ; :: semaphore?
   ; A semaphore which is ready when the mvar is full.
   empty ; :: semaphore?
   ; A semaphore which is ready when the mvar is empty.
   ))
```

Figure 2-1: The Scheme definition of an `mvar`. An `mvar` stores a value and two semaphores. When the `mvar` is full, the `full` semaphore has value 1 and the `empty` semaphore has value 0. When the `mvar` is empty, the `full` semaphore has value 0 and the `empty` semaphore has value 1. Both semaphores are required in order to support operations which require the `mvar` to be full (e.g. `take-mvar`) and operations which require it to be empty (e.g. `put-mvar`).

2.2.1 `mvar` operations

An `mvar` supports the following fundamental operations:

`new-mvar` Constructs a new `mvar` which begins full with a supplied value.

`new-empty-mvar` Constructs a new empty `mvar`.

`take-mvar!` Blocks until the `mvar` is full and then empties it and returns its value.

`try-take-mvar!` If the `mvar` is full, empties it and returns its value. If the `mvar` is empty, immediately returns some specified failure result. The failure result is `#f` by default.

`put-mvar!` Blocks until the `mvar` is empty and then fills it with a given value.

`try-put-mvar!` If the `mvar` is currently empty, fills it with a given value and returns `#t`. Otherwise, immediately returns `#f`.

put-mvar-evt! Creates a new synchronizable event which becomes ready when the mvar is empty. Synchronizing the event fills the mvar with a given value.

prop:evt A structure property specifying how the mvar can be used as a *synchronizable event*, which becomes ready when the mvar is full. The PLT Scheme sync procedure can be used to choose between multiple synchronizable events in a similar manner to that in which the select() system call may be used to choose between multiple file descriptors. [PLT] Synchronizing the mvar empties the mvar and returns its value.

Additionally, some select combinations of the above operations are provided as atomic operations. For instance, a read-mvar operation is provided which combines take-mvar with put-mvar but which maintains the lock across both operations and prevents another thread from writing a new value to the mvar in between. The compound actions provided as atomic operations include:

read-mvar Blocks until the mvar is full and then returns its value without modifying the mvar.

try-read-mvar If the mvar is currently full, returns its value without changing the mvar. Otherwise, immediately returns some specified failure result. The failure result is #f by default.

read-mvar-evt Returns a fresh synchronizable event which becomes ready when the mvar has a value. Synchronizing the event returns the value without modifying the mvar.

swap-mvar! Atomically swaps the value in the mvar for another value and returns the previous value. If the mvar is empty, blocks until the mvar is full.

with-mvar Atomically gets a value from the mvar, applies a function to it, and returns the result of the function. This operation does not itself change the value stored in the mvar, although the applied function might mutate it. If the evaluation of the function fails, the mvar is left in an unusable state.

In general, it is not always possible to construct correct concurrent procedures using only an mvar. This is a consequence of the fact that, as noted by [HMPJH05], locks do not compose. However, the most common use case for an mvar is to protect access to a single object, such as a hash table. A procedure needing access to the object takes the value from the mvar, performs any number of operations on the object, and then writes the value back into the mvar. As long as no procedure maintains a reference to the object after writing it back into the mvar and as long as no distinct objects are written to the mvar, each procedure using the mvar will have exclusive access to the object and mvar during its execution. This corresponds most directly to the map-mvar and with-mvar functions listed above.

2.3 Consistent Distributed Variables

The `mvar` abstraction is useful for a single computer, but its failing is that it is not serializable and cannot be used to synchronize data across multiple computers. Therefore, we built an abstraction on top of `mvars` which was serializable. We now present our `dbox` — short for *distributed box* — abstraction for distributed variables with write once, read anywhere semantics.

A `dbox` is a variable which may be created on any computer in the network and is initially empty. A `dbox` descriptor may be serialized, transfered to another computer, and deserialized to a corresponding `dbox` descriptor. All such `dbox` descriptors across multiple computers represent the same variable and share consistent semantics. Any process on any computer with a `dbox` descriptor may write a value to the `dbox` and all other computers with the `dbox` will receive the value. In order to manage this, all computers in the network run `dbox` server processes which communicate with other computers to propagate values. Only one distinct value should ever be written to the dbox, among all computers in the network.

2.3.1 Local **`dbox`** representation

A `dbox` descriptor on a single computer contains four fields:

- A randomly assigned 128-bit **unique identifier** (a version 4 UUID). Using a 128-bit identifier space ensures that the system is extremely unlikely to have a collision between two different `dbox` identifiers as long as the number of `dbox` variables does not approach 2^{64}.

- An **mvar** holding the local version of the variable. The `mvar` synchronizes the variable for concurrent access on a single computer. This ensures that only one value can be written to the `dbox` from a single computer. It also provides blocking and event semantics which can be used to notify threads when the a value arrives in the `dbox`.

- The **hostname** of the computer which originally created the `dbox`. If the `dbox` was created locally, this contains an external hostname which may be used to access the `dbox` server.

- The **port number** of the `dbox` server socket on the originating host.

These fields are represented in Scheme code as shown in Figure 2-2.

Additionally, the `dbox` server maintains some local state for all `dbox` variables which have descriptors present on the computer. This state includes:

`dbox-table` A hash table mapping unique identifiers to `dbox` descriptors. This hash table is used to intern `dbox` descriptors so that at most one descriptor exists on any one computer for a given `dbox`.

```
(define-struct dbox
  (id ; :: uuid?
   ; The unique identifier for this dbox.
   mvar ; :: mvar?
   ; An mvar which contains the value of the dbox or is empty.
   hostname ; :: string?
   ; The hostname for reporting/getting the value.
   port-no ; :: exact-nonnegative-integer?
   ; The port number for reporting/getting the value.
   ))
```

Figure 2-2: The Scheme definition of a `dbox` descriptor. Each `dbox` descriptor has a unique identifier which is a randomly-generated (version 4) 128-bit UUID, an `mvar` to hold the local version of the variable and synchronize access, and the hostname and port number of the originating `dbox` server.

`listener-table` A hash table mapping unique identifiers to lists of listeners. Each listener is a thread who cares about the value of the `dbox`. When the `dbox` is filled with a value, the `dbox` descriptor is sent as a message to each listener thread.

2.3.2 Network communication

The `dbox` servers on the computers in the network communicate over TCP using an S-expression-based text protocol with two basic types of messages:

`(listen ID)` A message installing a listener on the receiving computer for the `dbox` with unique identifier equal to `ID`, where `ID` is written in a standard hexadecimal format. If the `dbox` has a known value on the receiving computer, then that value is immediately returned with a `post` message. Otherwise, the value will be returned in a `post` message as soon as the value is known on the receiving computer. Note that the receiving computer is not required to already have a descriptor for the `dbox`: if a `listen` message is received before any descriptors for the corresponding `dbox` are present, the receiving computer will maintain the listener until the `dbox` descriptor has been received and has a value.

`(post ID VALUE)` A message informing the receiving computer of the value of the `dbox` with unique identifier equal to `ID`, where `ID` is written in a standard hexadecimal format. The value `VALUE` is a structured S-expression representation of the `dbox`'s value as serialized using the PLT Scheme serialization framework. If there is no descriptor for the `dbox` on the receiving computer then the `post` message is ignored. If the received value of the `dbox` is already known by the the receiving computer, then the message is ignored. If a value for the `dbox` was already known and a different value was received, then a warning is logged.

The use of these messages in order to implement serialization and deserialization is described in Section 2.3.4. An example interaction between `dbox` servers is shown in Figure 2-3.

2.3.3 `dbox` operations

The `dbox` module provides the following operations:

`new-dbox` Creates a new (empty) `dbox` based on the local machine and returns its descriptor.

`dbox-get` Blocks until the `dbox` has a value and then returns it.

`dbox-try-get` If the given `dbox` has a known value on the local machine, returns that value. Otherwise, immediately returns some specified failure result. The failure result is `#f` by default.

`dbox-post!` Puts a value into the `dbox`, which should be empty on all computers in the network. If the `dbox` has a known value on the local machine, this function throws an error. It is possible for the `dbox` to be globally full but locally empty, in which case there will be a consistency error if a different value is posted to the `dbox`. Only one distinct value should ever be posted to this `dbox` among all computers in the network.

`dbox-install-listener` Installs a listener for the `dbox` with a given unique identifier. When the `dbox` has a known value, the `dbox` descriptor will be sent as a message to the listener thread. If the `dbox` already has a known value on the local machine, the `dbox` descriptor is immediately sent to the listener thread.

prop:evt A structure property specifying how the `dbox` can be used as a synchronizable event, which becomes ready when the `dbox` has a known value on the local machine. Synchronizing the `dbox` returns its value. Synchronizing the `dbox` is equivalent to synchronizing the underlying `mvar`.

We note that these supported operations closely match the requirements specified in Section 2.1.

2.3.4 `dbox` serialization

We designed the `dbox` abstraction specifically so as to allow `dbox` descriptors to be serialized and transferred between computers. Moreover, we integrated our `dbox` serialization procedures with PLT Scheme's serialization framework so as to allow the serialization of complex data structures which include one or more `dbox` descriptors. PLT Scheme's serialization framework is capable of serializing heterogeneous, cyclic data structures into structured S-expressions which can be deserialized to produce equivalent data structures. [PLT] Making a structure serializable within this framework generally requires implementing three procedures:

- A **serialization** procedure which takes an instance of the structure and returns a vector, each of whose elements is itself serializable, representing the instance's contents.

- A **deserialization** procedure which takes the elements of such a vector and produces a equivalent instance of the structure.

- A **cyclic deserialization** procedure which updates an empty instance of the structure with given data. This is used to break cycles in serialized data structures.

Complicating the implementation of these procedures is the additional state maintained by the local `dbox` server. Serialization is simple: the `dbox` descriptor's unique identifier, hostname, and port number are serialized, along with the currently known value of the `dbox`, if any. To deserialize a `dbox` descriptor, we first lookup its unique identifier in `dbox-table`. If the `dbox` is already known locally, then we return the existing `dbox` descriptor. Otherwise, the `dbox` descriptor is deserialized and added to `dbox-table`. A connection is made to the home host and port of the `dbox` and a `listen` message is sent, so that the local computer will receive any update to the value of the `dbox` which is posted from another computer. Lastly, a listener is installed in `listener-table` which will report any changes in the status of the `dbox` to the `dbox`'s home host and port. Thus, any local updates will be propagated to the rest of the network and any remote updates will be reported to the deserializing computer.

For an example of how `dbox` value are propagated, see Figure 2-3.

2.3.5 **dbox** server architecture

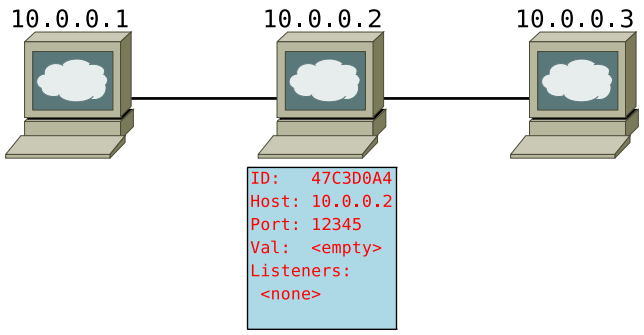
The `dbox` module is designed to support multiple concurrent accesses to `dbox` data by client threads. Thus, the `dbox` module is written with a small number of `mvar`-protected global variables which can be accessed by exported `dbox` functions called from client code. These global variable are `dbox-table` and `listener-table`, described in Section 2.3.1. The exported `dbox` operations (described in Section 2.3.3) do not directly make use of the network but do send messages to server threads which do communicate over the network.

The server threads maintained by the `dbox` server are:

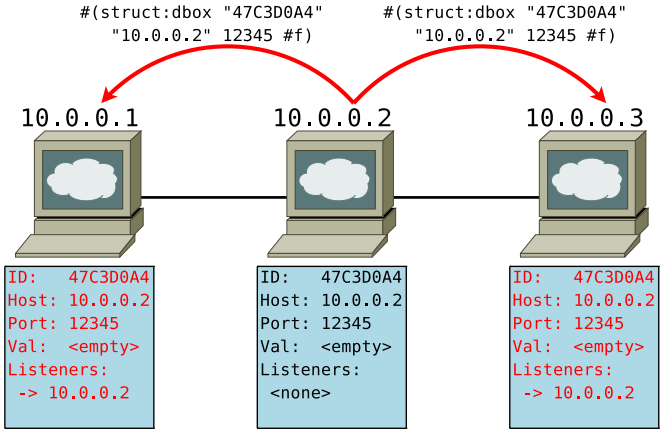
- An **accepting connections** thread which listens on the server's TCP port. When a remote host connects to the server, this thread creates socket reader and writer threads for the connection.

- **Socket reader** threads which read messages from a particular TCP connection and process them. The types of messages are described in Section 2.3.2.

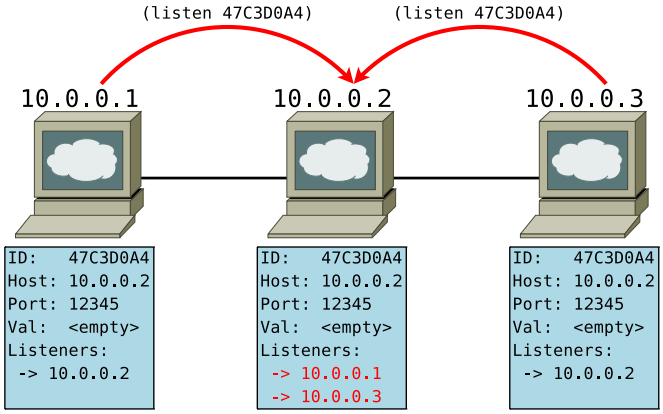
- **Socket writer** threads which accept messages from their thread mailboxes and forward them to the remote hosts. A `dbox` received in the thread mailbox is automatically translated into a `post` message sent to the remote host.



(a) A new dbox is created on host 10.0.0.2. The dbox hostname and and port number are those of the local dbox server. The dbox is initially empty and has no listeners.

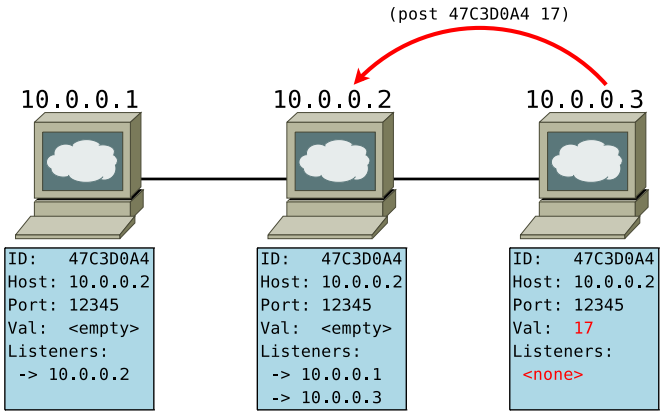


(b) The dbox descriptor is serialized and sent to hosts 10.0.0.1 and 10.0.0.3, where it is deserialized by the dbox module. The deserializer on each computer installs the socket handler for the connection to 10.0.0.2 as a listener for the dbox in listener-table.

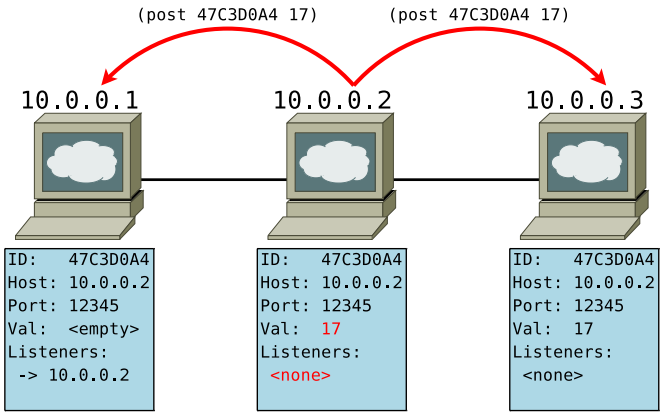


(c) Both 10.0.0.1 and 10.0.0.3 send listen messages to the dbox creator asking to be notified of the eventual value of the dbox.

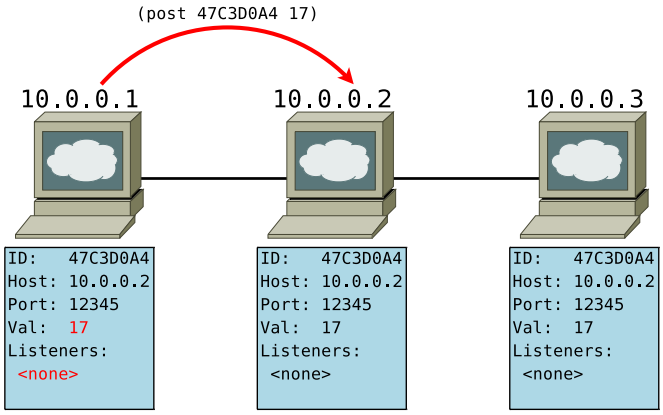
Figure 2-3: An example of dbox value propagation. In this example, the network contains three hosts. A dbox is created on one of the hosts, serialized, and sent to the other two hosts. When a value is put in the dbox by one host, all receive the value.



(d) Host `10.0.0.3` posts a value to the `dbox`. The local value on `10.0.0.3` is changed to 17 and a `post` message is sent to each listener. The only listener is the socket handler for the connection to 10.0.0.2. After sending the `post` messages, the list of listeners is emptied.



(e) Host `10.0.0.2` receives the `post` message. Since it has no local value for the `dbox`, it sets the value to 17 and sends `post` messages to all of the listeners installed.



(f) Hosts `10.0.0.1` and `10.0.0.3` receive the `post` messages from `10.0.0.1`. Since `10.0.0.3` already has the value of the `dbox`, it ignores the message. Host `10.0.0.1` receives the message and sets the value to 17. Since `10.0.0.1` has `10.0.0.2` registered as a listener, it sends a `post` message to `10.0.0.2` to inform it of the value. (Host `10.0.0.1` does not remember that `10.0.0.2` already has the value 17.) Host `10.0.0.2` receives this `post` message and ignores it.

In addition to the `dbox-table` and `listener-table` hash tables, the `dbox` server maintains the `writer-table` hash table which maps hostname/port pairs to socket writer threads. This hash table is used to reduce the number of extraneous connections which must be made by storing all current connections.

2.4 Extensions to Variable Semantics

Our `dbox` abstraction nicely represents distributed variables without mutation. However, there are some cases in which it is difficult to implement an efficient parallel algorithm without mutation. This section considers how to support a limited, safe form of mutation with the `dbox` abstraction.

2.4.1 Motivation

Suppose that we wish to write a game playing program using alpha-beta pruning. With alpha-beta pruning, alpha-beta values discovered in one branch of the game tree may be used to prune a different branch of the game tree. How do we represent this functionally?

The straightforward solution would be to introduce a `dbox` variable for each alpha-beta value. When considering two different branches of the game tree, the first branch may set one or more alpha-beta values by posting them to the corresponding `dbox` variables. The second branch may then read the alpha-beta values from the `dbox` variables and prune accordingly.

However, this destroys the parallelism we are trying to achieve. By making the second subproblem dependant on the completion of the first subproblem, we force the subproblems to be evaluated in serial rather than in parallel.

As a second example of this problem, consider a DPLL SAT-solver. In such a solver, rules may be learned when exploring one branch of the search tree. These rules will constrain the remainder of the search, speeding it up. How do we represent this? As with the alpha-beta pruning example, we could introduce one or more `dbox` variables for the rules learned. But, as before, if one subproblem writes to a `dbox` and another subproblem reads from it, this forces the subproblems to be evaluated in serial rather than in parallel.

2.4.2 Admissible values

To fix this, note that these applications do not require a specific correct value: they can each produce the correct answer given a range of admissible values. A game playing program will return the correct value for a given game state so long as each alpha value is less than or equal to the true value and each beta value is greater than or equal to the correct value. (In particular, $\alpha = -\infty$ and $\beta = \infty$ are always admissible assignments.) Likewise, a DPLL SAT-solver will correctly determine whether a boolean formula is satisfiable so long as each rule in its inferred rule set is a valid rule. (In particular, the empty set is always a valid set of inferred rules.)

Thus, we introduce the notion of a `dbox` with multiple possible *admissible* values. Any computer may post to this `dbox` at any time, potentially multiple times. The values posted to the `dbox` by different computers or at different times need not be the same; they need only be *admissible*, for some definition of admissibility which will vary from problem to problem.

Each such `dbox` can be equipped with a *law of combination* (an update function) which describes how to merge different updates to the `dbox`. In the case of alpha-beta pruning, the correct way to merge two alpha values is to take their maximum; the correct way to merge two beta values is to take their minimum. In the case of a DPLL SAT-solver, the correct way to merge two sets of learned rules is to take the union of the two sets. Thus, values posted on one computer will eventually be propagated to other computers. During this process, different computers may have different values for the `dbox`, but all values will be admissible.

2.4.3 Idempotent updates

In order to implement distributed admissible variables, we need the law of combination — the update rule — to have two key properties at a low level:

- Updates should be **commutative**. That is, it should not matter in what order the updates are performed. Different computers will receive the updates at different times, and so the order of updates must not change the final result.

- Updates should be **idempotent**. That is, after an update has been applied, repeated applications of that update should have no effect.

Examples of update functions (i.e. laws of combination) with these properties include:

- Minimum and Maximum

- Set union and set intersection

- Maintaining the k largest/smallest distinct elements

These update functions can support, among other applications, alpha-beta pruning (with min/max) and DPLL learning (by taking the union of learned rules, or alternatively just the k best rules).

An noteworthy update function which does *not* have these properties is addition. Although addition is commutative, addition is not idempotent. If a computer receives multiple extraneous messages for the same update, it will think that they represent multiple updates and perform multiple additions, yielding a value larger than it should be. As shown in Figure 2-3, our system does indeed generate redundant messages for the same update. Although our system could potentially be redesigned, it will still be the case that we may need to send the same message multiple times in the event of network errors. Thus, extraneous update messages should not corrupt the variable. The idempotence property guarantees this.

2.4.4 Non-idempotent updates

It is also possible to support non-idempotent updates by adding a deduplication layer to variable updates. Thus, if there are multiple messages for the same update, the receiving computer will know to only perform the update once.

Each unique update is assigned a random update identifier. On each computer, the variable's update function maintains the set of updates performed so far. To combine two variables, we take the union of the two sets of updates. As before, updates should be commutative.

Examples of non-idempotent updates which we can support in this fashion include:

- Addition
- Logging

Chapter 3

The Scheme Computation Server

In this chapter, we discuss how our system manages jobs, performs job context switches, and transfers jobs to different computers. Overall, this chapter describes the workings of a Scheme computation server. The distributed variables described in Chapter 2 are used as building blocks for communication both on a single computer and across computers.

We first describe the PLT Scheme serialization framework in further detail. This framework is used to serialize jobs and their attached code and is central to the workings of the Scheme server. We then show how to define a job in such a way that we will able to serialize it using this framework.

Next, we describe how the Scheme server manages jobs. The Scheme server has an associated *manager* which maintains the collection of all jobs to be processed. Multiple jobs may be in progress at any one time, but only one job is running at a time. The manager tracks the dependencies of jobs on other jobs and restarts blocked jobs when their dependencies are ready. Lastly, the manager packages and provides useful semantics to the running job for creating other jobs and for getting their values. Overall, the manager represents a managed code framework for jobs running in the system.

Lastly, we describe the networking activities of the server and how it communicates with other servers and shares jobs.

3.1 Job Serialization

In order to transfer jobs from one computer to another, we must have some way to serialize and deserialize jobs and a way to return the result of the job to the originating computer. This section describes how this is done.

3.1.1 PLT Scheme serialization framework

PLT Scheme includes a powerful serialization framework capable of serializing heterogeneous, cyclic data structures into structured S-expressions [PLT]. These S-expressions can later be

deserialized to produce equivalent data structures. The serialization framework can serialize many types of data, including:

- Booleans, numbers, characters, symbols, character strings, byte strings, file paths, and the empty list.
- Pairs and lists.
- Vectors (i.e. arrays).
- Hash tables.
- Any structure with the `prop:serializable` structure property (e.g. `dbox`, as explained in Section 2.3.4).

Primitive data types are self-contained in their serialization: they need no external resources present in order to be deserialized. However, serialized structures contain references to the deserialization information for the corresponding structure type and to the module in which this information is defined. For the `dbox` descriptor structure, this information is the pair `(lib "alex/thesis/dbox.ss") . deserialize-info:dbox-v0)`. The first element of the pair names the module in which the deserialization information can be found and the second element is the identifier for the deserialization information within that module. As an example, see Figure 3-1.

3.1.2 Serializable closures

The PLT Scheme serialization framework, as described in Section 3.1.1, already supports most of the basic data types needed for distributing purely functional programs. However, there is one key omission: closures (functions) are not typically serializable. Thankfully, this can be fixed. Instead of a primitive Scheme function, it is possible to define a PLT Scheme structure type with the `prop:serializable` structure property (hence serializable) which also has the `prop:procedure` structure property. Such a structure can be serialized but can also be used as a procedure.

A sophisticated implementation of this idea can be found in the PLT Web Server collection [McC], which defines a `serial-lambda` macro which acts like a `lambda` but which produces a special serializable procedure instead of a regular procedure. The macro takes the formal function arguments and body, analyzes the body to identify the free variables, and captures and explicitly stores the free variables in the procedure structure. Serializing this structure produces the captured free variables and a reference to the binding of the function specification in the defining module. For an example, see Figure 3-2.

Helpful though serializable closures are, they do not, by themselves, allow closures to be serialized on one computer and deserialized on a different computer, as the source file paths for the closures will be invalid on the receiving computer. In order to account for this, we need to keep track of the source code and code path of a job. This is addressed in Section 3.1.3.

```
((2)
 2
 (((lib "alex/thesis/dbox.ss")
   .
   deserialize-info:dbox-v0)
  ((lib "alex/thesis/uuid.ss")
   .
   deserialize-info:uuid-v0))
 0
 ()
 ()
 (0
  (1
   (u
    .
    #"\304\242\263\247B\362F\267\241\236\220\374\37*\232\372"))
  #f
  #f
  (u . "128.30.60.231")
  39994))
```

Figure 3-1: The serialized representation of a `dbox` descriptor. The serialization is a structured list. The first element of the list, `(2)`, is a version number. The second element of the list is the number of distinct structure types serialized. The third element of the list is a list of pairs, each pair describing a structure type. Typically, the first element is the module in which the deserialization information for that structure type is defined and the second element of the pair is the name of the binding for the deserialization information within that module. The following elements of the list represent the (possibly cyclic) data graph and the type of the output. In the last element, the numbers `0` and `1` are indices into the list of structure types. Thus, `0` represents a `dbox` structure while `1` represents a UUID structure.

```
(define (foo x)
  (serial-lambda
   (y)
   (+ x y)))

(serialize (foo 5))  ⟹

((2)
 1
 ((#"/home/alex/temp/serial-lambda-example.ss"
   .
   "lifted.1"))
 0
 ()
 ()
 (0 5))
```

Figure 3-2: An example of a serializable closure. The `foo` function creates and returns a new serializable closure which captures the value of the argument `x`. A serializable closure is created for $\mathtt{x} = 5$ and the closure is serialized.

The serialization includes a single structure type, namely the serializable structure type representing the procedure `(λ (y) (+ x y))`. This structure type is represented in the serialization as a pair. The first element of this pair is a byte string representation of the file path in the local file system of the source file in which the procedure is defined. The second element is a generated name identifying which serializable closure structure type in the source file is intended.

Lastly, the value of the serialization is `(0 5)`, where this represents an instance of the structure type defined at index 0 instantiated with field `5`. The `5` is the value captured by the closure, which is explicitly stored by the serializable closure.

```
(define-struct job
  (f ; :: serializable?
  ; The function to compute, which must be serializable
  ; and must return a serializable result.
  args ; :: (listof serializable?)
  ; The function arguments, which must be serializable.
  code ; :: bytes?
  ; A bytestring representing a tar.gz archive containing
  ; the code for the program corresponding to the job.
  codepath ; :: string?
  ; A string representing the path in the local filesystem of the
  ; root of the job's code. This should correspond with the root
  ; of the code archive.
  dbox ; :: dbox?
  ; A dbox accepting the result of the computation when
  ; finished. The result should be one of:
  ;   (value . [VALUE])
  ;   (exn . [STRING])
  ;   (raise . [VALUE])
  ))
```

Figure 3-3: The Scheme definition of a job object. Each job object contains the function to apply, the function arguments, the source code, the code path, and a descriptor for the `dbox` receiving the value of the job.

3.1.3 Job representation

A self-contained representation of a job comprises several fields:

- The **function** to apply, which must be serializable as explained in Section 3.1.2.
- The **arguments** to which to apply the function, which must all be serializable.
- The **source code** tar.gz archive containing the code for the job.
- The **code path** of the local copy of the source code. This code path should be consistent with the other fields. In particular, the function (and any other closures passes as arguments to the function) should be serializable closures referring to source files in the code path. Also, the contents of the code path should be the same as the contents of the source code tar.gz archive.
- A descriptor for the **dbox** receiving the value of the job.

These fields are represented in the Scheme definition of a job object as shown in Figure 3-3.

3.1.4 Remote serialization & deserialization

With our understanding for how the serialization of closures can be done, we can describe how to serialize and deserialize job objects. We found that it was most convenient to define the serialization and deserialization of job objects outside of the Scheme serialization framework.

Looking at the fields in Figure 3-3, we see that all of them are serializable within the Scheme serialization framework. (Serialization of `dbox` variables is explained in Section 2.3.4.) However, the serialized form of the function refers to one or more paths in the local file system. Additionally, if any closures are included in or among the job arguments, then the serialized form of the closure will similarly refer to one or more paths in the local file system. These will not trivially deserialize on a different computer.

Thus, we establish a custom serialization procedure:

1. Gather together the job's `dbox` descriptor, the function, and the function arguments as a list `(list* dbox f args)`.

2. Serialize that list.

3. Examine all structure deserialization information contained in the serialization. Translate any paths in the local file system by removing the code path from the beginning of the path. Leave system library paths unchanged.

4. Return a list of the code archive — encoded as a byte string using the Base64 encoding scheme — and the translated serialization of `(list* dbox f args)`. This description of the job is now self-contained and independent of the local file system.

To deserialize such a representation of a job:

1. Hash the code archive and look it up in a hash table. If that code archive is already known on the receiving computer, set the code path of the deserialized job to be equal to the root of the existing local copy of the archive's contents. If the code archive is not known locally, create a temporary directory and extract the code archive into it. In this case, the code path is the newly created directory.

2. Translate the serialization of `(list* dbox f args)` by prepending any file system paths with current code path.

3. Deserialize `(list* dbox f args)`.

4. Return a job with the given code path and other fields.

By caching the code archives and by only extracting a given archive once, we reduce the file system usage and prevent module conflicts. Thus, if we have a single master job with many subjobs, none of the deserializations of subjobs after the first deserialization will require the use of the file system.

3.2 Job Management State

Now that we have a representation of a job and can serialize and deserialize it, we must describe how jobs are evaluated. Jobs are queued and evaluated by the server's *manager*. The manager maintains enough state to decide which job should be run next and what should happen with the result of the job. If a job should become blocked, waiting for the value of a future which is the output of another job, the manager will suspend the currently running job and run other jobs, returning to the previous job when it becomes ready.

In this section, we describe the state maintained by the manager.

3.2.1 Thread-local state

Some of the state required for job management is specific to the individual task threads managed by the manager and is not directly used by the manager. The state components with these properties are:

- **The identity of the manager.** Because the system supports multiple distinct managers running simultaneously in the same Scheme instance, each managing some subset of the instance's jobs, each task thread must have a reference to the manager which is managing it.
- **The job code.** When a job spawns subjobs, these subjobs must be associated with the job's source code. That way, if the job is serialized and transfered to another computer, the function to apply can be deserialized using that source code.
- **The local code path.** The job code is stored in an archive with paths relative to some code root (in our system, the root directory of the original computer running the application). However, this archive will be extracted in some different directory of the local file system of a remote computer. This means that the serialization paths discussed in Section 3.1.2 must be updated when a job is transferred from one computer to another. This requires that we maintain the local root directory for the code.

This state is constant for a given job and for a given task thread. Thus, a convenient way to store each component is in a *thread cell*. Thread cells are a feature of PLT Scheme which provide per-thread storage. A thread cell may be thought of as a box which may refer to a different value for each thread.

3.2.2 Manager state

The manager maintains three principle state components:

- A **job queue** of jobs waiting to be started.
- A **job stack** of currently active task threads which are ready to be resumed.

- A **blocked jobs table**, which is a hash table mapping job identifiers to other jobs which are blocking on them. This hash table stores all currently active jobs which are blocking on the result of another job.

The distinction between the job queue and the job stack is twofold. First, the job queue contains job objects for jobs which have not been started, while the job stack contains the *task threads* of jobs which have already been started. Second, the two have different control flow requirements. For fairness, all top-level jobs are processed in the order in which they are received. Thus, we maintain a *queue* of unstarted jobs. However, once we have started processing a job, we prefer to finish that job, and all of its subjobs, before moving on to other jobs. Thus, we choose a *stack* to minimize the number of active jobs in progress at any one time.

3.3 Task Threads

Each job is executed in its own *task thread*. A job's task thread is responsible for initializing its thread state, safely executing the job, and then reporting the result to the system and updating the manager's state.

The task thread operates as follows:

1. Initialize the thread-local state, as described in Section 3.2.1. Specifically, set the values of the thread cells corresponding to the manager, to the code archive, and the code path.

2. Install exception handlers catching all serializable values thrown and all exceptions thrown.

3. Run the job.

   - If the job returned one or more values `vs`, then the result of the job is `(cons 'values vs)`. (In the usual case of a function which returns a single value, `vs` is a list of a single element.)
   - If the job threw a serializable value `v`, the result of the job is `(cons 'raise v)`.
   - If the job threw an exception `e`, the result of the job is `(cons 'exn (exn-message e))`, where `(exn-message e)` is the string error message of the exception. (Note that exceptions are not serializable.)

4. Write the result of the job to the job's `dbox`.

The Scheme code for starting a job and its corresponding task thread is given in Figure 3-4.

```
; Starts a job in a task thread and waits until it finishes or
; is suspended.
; run-job :: (-> job? void?)
(define (run-job job)
  (let* ([f    (job-f job)]
         [args (job-args job)]
         [code (job-code job)]
         [path (job-codepath job)]
         [td
          (thread
           (lambda ()
             ; Initialize the thread-local state
             (thread-cell-set! manager-cell self)
             (thread-cell-set! code-cell code)
             (thread-cell-set! codepath-cell path)
             (finish-job
              job
              (with-handlers*
               ([serializable?
                 (lambda (v)
                   (cons 'raise v))]
                [exn?
                 (lambda (e)
                   (cons 'exn (exn-message e)))])
               (cons 'value (apply f args))))))])
    ; Wait until the thread is finished or suspended.
    (sync (thread-dead-evt td)
          (thread-suspend-evt td))))
```

Figure 3-4: The Scheme code for starting a new job. The function extracts the job information from the job object, starts a new task thread, and waits for the task thread to finish or be suspended. The task thread initializes the thread-local state, installs exception handlers, and then safely runs the job. Upon completion, the result of the job is written to the job's `dbox`.

3.4 Control Flow

During the execution of a job, control will pass from the manager to the task thread, back to the manager, and potentially back and forth several more times. The manager and its task threads thus implement *cooperative multithreading*. At most one task thread is running at any one time, and if a task thread is running then the manager is not running. Task threads explicitly yield execution to the manager. The manager explicitly waits on the current task thread.

The manager operates in a continual loop:

1. While there are active jobs on the job stack:
   (a) Pop the top task thread off of the job stack.
   (b) Resume the task thread.
   (c) Wait until the task thread is either finished or suspended.
2. Wait until we either have a new job in the job queue, or we receive a signal. The signal represents that some `dbox` has received a value which may unblock a suspended job.
   - If we received a new job, start a new task thread, as described in Section 3.3. Wait until the task thread is either finished or suspended.
   - If we received a signal, this signifies that an active job may have been unblocked. We check the blocked jobs table and push all jobs which were blocking on the finished job back onto the job stack.
3. Return to step 1.

The interesting question is: when running a job, how is control returned to the manager? If the job does not spawn any subjobs and does not touch any futures (i.e. does not require the results of any existing subjobs), then the job runs until it produces a result, at which point the task thread posts the result to the job's `dbox` and finishes. However, what if the job requires the value of another job? We must arrange for the task thread to *touch* the future corresponding to the value of the other job. Thus must be done such that the current task is suspended, the prerequisite task is run, and the current task is subsequently continued and finished.

The *touch* operation, in which a task thread waits on the result of a job, works as follows:

- Access the table of blocked jobs. Add the current task thread to the list of jobs which are blocking on another job. If the current job is the first job blocking on that job, then install a listener with the `dbox` server for that job, so that the manager will be notified if the other job is finished on a different computer.
- Suspend the current thread, returning execution to the manager.
- When control returns to the task thread, it should be because the prerequisite job has been finished. Return its value.

The Scheme code for this can be seen in Figure 3-5. A simple example of control flow can be seen in Figure 3-6.

3.5 Scheme Server Communication

The Scheme server implements a very simple protocol for communicating with other Scheme servers. This protocol allows jobs to be transfered between servers using *work stealing* [BL99]. In work stealing, each server performs its own work. The client application is itself a server, and the application's tasks are enqueued in the local server's manager's job queue. Idle servers in the network randomly query other servers asking for jobs. If an idle server queries a busy server, the busy server will dequeue one of its unstarted jobs and give it to the idle server.

Thus, our very simple protocol comprises just two types of messages:

`(steal)` This message is sent by an idle server to another server. The message asks for an unstarted job to be given to the requesting server, if available.

`(job JOB)` This message is sent in reply to a `steal` message. It contains the serialization of a job. The job is enqueued at the receiving server. If the receiving server is still idle, the job is started immediately. If the server is now busy — perhaps with a job from a different server — then the job in enqueued.

In order to maximize efficiency, we want all servers to have a roughly equal amount of work, on average. We would never want job to be transfered among a small fraction of the servers while other servers remain idle. In order to minimize the chance of this, the job stealing process is randomized. When a server is first idle, it sends out a random number of `steal` messages to random servers. In our implementation, the average number of `steal` message sent at once was arbitrarily chosen to be 5. If no jobs are received in response, the server then waits a random amount of time before sending additional `steal` messages. In our implementation, the "boredom timeout" was randomly chosen between 1 and 2 seconds. This randomization is important, as without it we observed certain cases of extremely poor load balancing.

```
; Implements the special blocking required in case we're in the
; manager right now. If we're not in the manager, just blocks on
; the future. If we are in the manager, adds us to the list of
; jobs to be continued upon evaluation of the future. If the
; future is ready, then just return its value.
(provide/contract
 [touch (-> future? any/c)])
(define (touch future)
  (define manager (current-manager))
  (cond
    [(future-ready? future)
     (future-get-now future)]
    [manager
     (let* ([this-thread (current-thread)]
            [blocked-jobs-mvar
             (manager-blocked-jobs-mvar manager)]
            [code (current-code)]
            [blocked-jobs (take-mvar blocked-jobs-mvar)])
       ; Add ourselves to the blocked jobs list...
       (hash-update!
        blocked-jobs (future-id future)
        (lambda (rest) (cons this-thread rest))
        (lambda ()
          ; (We're the first job who wants this value, so add a
          ; listener for this dbox.)
          (dbox-install-listener (future-id future)
                                 (manager-thread manager))
          '()))
       (put-mvar blocked-jobs-mvar blocked-jobs)
       ; ... suspend ourselves and wait to be woken up...
       (unless (future-ready? future)
         (thread-suspend this-thread))
       ; ...and we're back!
       (future-get-now future))]
    [else (future-wait future)]))
```

Figure 3-5: The Scheme code for the *touch* function, which blocks on the value of a future. If the value of the future is known, just return its value. If the current thread is not managed, simply wait for the future to have a value. If the current thread is managed, we must yield to the manager so that the other task can be run. In this case, we add the current task thread to the list of tasks to be continued when the future is ready. We then suspend the current thread, allowing the manager to continue execution.

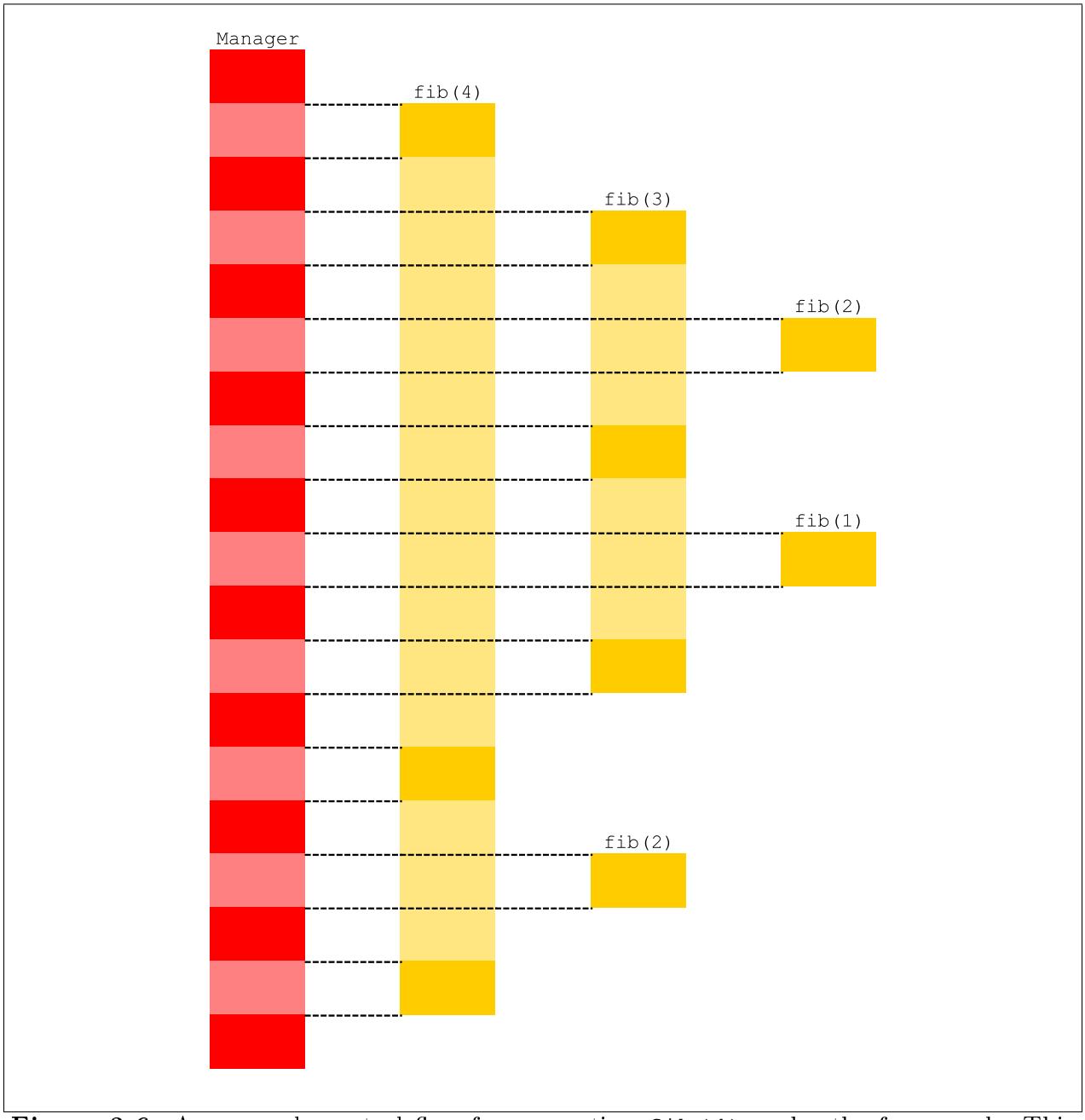


Figure 3-6: An example control flow for computing `fib(4)` under the framework. This diagram assumes that the base cases for `fib(n)` are $n \leq 2$. First, the manager starts. The manager receives the job `fib(4)` and starts it. The job `fib(4)` spawns off two subjobs which are placed on the manager's job stack. The job `fib(4)` then touches the future for `fib(3)` which causes it to be suspended. Job `fib(3)` starts and creates subjobs `fib(2)` and `fib(1)`. It blocks on `fib(2)` and is suspended. Job `fib(2)` is evaluated and `fib(3)` is continued. It blocks on `fib(1)` and is suspended. Job `fib(1)` is evaluated and `fib(3)` is continued and finishes. Job `fib(4)` is continued, blocks on `fib(2)`, and is suspended. Job `fib(2)` is evaluated and `fib(4)` is continued and finished.

Chapter 4

Applications

Ideal applications of our system share certain common features.

- The problem should be **computationally challenging** and cannot be solved quickly enough on a single computer.
- The problem should involve **modest amounts of data** so as to make the communications cost feasible.
- The problem should be solvable with a **divide and conquer** algorithm, with each problem divisible into independent subproblems.
- The problem may potentially have an **ad hoc** structure in which the specific subproblems are not known at the start of the computation.

Many problems have these features. Potential problems can be found among polynomial-time problems (P), nondeterministic polynomial-time (NP) problems, and polynomial-space problems (PSPACE), and other exponential-time problems (EXPTIME). We could not possibly list all such applications. However, we can give some key exemplars. In particular, we shall briefly discuss how our system can be used to approach SAT, as an NP-complete problem, and various game playing problems, which are often PSPACE- or EXPTIME-complete.

4.1 SAT solver

The *boolean satisfiability problem* (SAT) is an excellent application of our system. It is a real-world problem with many practical applications, especially in electronic design automation. Examples include circuit equivalence checking, test-pattern generation, and bounded model checking [MSS00].

In SAT, we are given a boolean formula φ in some n variables $x_1 \ldots x_n$. We wish to know whether there exists an assignment of $x_1 \ldots x_n$ to True and False such that the formula φ evaluates to True. Usually, instances of SAT are formulated in conjunctive normal form

(CNF), which is to say that the formula is a conjunction ("AND") of clauses, where each clause is a disjunction ("OR") of literals. Each literal is either a variable x_i for some i or the negation of such. For example,

$$\varphi = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$$

is a simple instance of CNF-SAT.

Both SAT and SAT restricted to CNF formulae are known to be NP-complete [Coo71]. Thus, it is unlikely that a polynomial-time algorithm exists. Nonetheless, it is both desirable and practical to solve specific instances of SAT. Thus, considerable work has been done on designing efficient SAT solvers.

Most SAT solvers employ variants of the DPLL algorithm [DLL62], using a recursive search with backtracking and learning, guided by various heuristics. In the DPLL algorithm, possible variable assignments are tried until a solution is found or a contradiction is reached. If a contradiction is reached, the algorithm tries to use the variables most directly involved in the contradiction to infer a new clause, constraining the SAT problem. The exact details of this learning process differ between implementations.

Because SAT is a computationally challenging problem, we would like to use clusters of computers to solve SAT more effectively. Indeed, distributed SAT solvers do exist; one such is GridSAT [CW03]. Our contribution, as with all applications, is the ease with which a distributed version of the application is built.

We implemented a SAT solver in PLT Scheme partly based on the open source DPLL-based SAT solver *MiniSAT*, written in C++ [ES03]. Our solver uses many of the same tricks for fast searching and backtracking, rewritten in a function style. Although our solver does not currently implement learning, learning could potentially be implemented in a future version using the extensions to our `dbox` semantics described in Section 2.4.

Our SAT solver is somewhat long — about 300 lines of Scheme code. However, the modifications required to enable distribution in our framework are extremely minor. The core functions of the SAT solver can be seen in Figure 4-1. The versions of those functions modified to use our distribution framework can be seen in Figure 4-2. No other changes are required in our to use the framework.

4.2 Game playing programs

A broad class of applications of our system is in writing distributed game playing programs. Searching a game state tree, typically using minimax, naturally decomposes into subproblems, where each subproblem is a subtree of the game tree. Thus, game playing programs are very amenable to parallel or distributed approaches. One distributed game playing program, for chess, can be found in [FMM91].

Games generally have all of the properties required to make use of our system. Game trees are generally exponential and exploring them is challenging. Moreover, the game state can usually be simply described. Lastly, subtrees of the game tree can be evaluated

```
(define (search trail)
  (let loop ([i 0])
    (if (< i n)
        (if (null? (vector-ref assigns (make-literal i #t)))
            (or (assign search trail (make-literal i #t) #f)
                (assign search trail (make-literal i #f) #f))
            (loop (+ i 1)))
        (for/list ([var (in-range n)])
                  (cons (literal->symbol (make-literal var #t))
                        (vector-ref assigns
                                    (make-literal var #t)))))))

(define (assign cont trail lit from)
  ...
  )
```

Figure 4-1: SAT solver Scheme code for a single computer

```
(define search
  (serial-lambda
   (trail)
   (let loop ([i 0])
     (if (< i n)
         (if (null? (vector-ref assigns (make-literal i #t)))
             (let ([fT (spawn assign search trail
                              (make-literal i #t) #f)]
                   [fF (spawn assign search trail
                              (make-literal i #f) #f)])
               (or (touch fT) (touch fF)))
             (loop (+ i 1)))
         (for/list ([var (in-range n)])
                   (cons (literal->symbol (make-literal var #t))
                         (vector-ref assigns
                                     (make-literal var #t))))))))

(define assign
  (serial-lambda
   (cont trail lit from)
   ...
   ))
```

Figure 4-2: SAT solver Scheme code modified to use the distribution framework

independently. Generally, game playing programs using alpha-beta pruning, transposition tables, and other techniques to speed the evaluation of later branches of the game tree. However, since the evaluation of different branches of the tree does not depend on other branches for correctness, only for tree pruning, we can use our `dbox` extensions (see Section 2.4) to support alpha-beta pruning and transposition tables.

Chapter 5

Metrics

Here, we present our data on the performance of the system.

5.1 Benchmark

As a simple benchmark, we used a simple, exponential-time algorithm for computing the Fibonacci numbers. Although this is a very simple benchmark, it captures the essence of the paradigm well. The subproblems for computing the Fibonacci numbers are of different sizes and require different computation depths. This makes the subproblems somewhat heterogeneous, unlike in many other frameworks for distributed computation.

Our template code for computing the Fibonacci numbers is shown in Figure 5-1. We modified this code to use our distributed computation framework. The modified code is shown in Figure 5-2.

Our system ran this code with some modifications for profiling, reporting the total CPU time used across all processes. The system was configured with a "star" pattern of network connections, in which each Scheme computation server was connected directly to the original client's Scheme server, but not to the other servers.

We ran the code with varying numbers of Scheme computation servers on two different platforms. The first platform was an 8-core computer comprising two quad-core Intel Xeon E5345 processors running at 2.33GHz. The second platform was the Amazon Elastic Compute Cloud (EC2), part of Amazon.com's cloud computing services. The results are described in Sections 5.2 and 5.3, respectively.

5.2 Multi-core Platform

We ran the benchmark on an 8-core computer comprising two quad-core Intel Xeon E5345 processors running at 2.33GHz. Although all network traffic was local, the full distribution framework was used. Parallelism was provided by running multiple Scheme computation servers — up to one server per core. The servers communicated with each other using

```
(define (fib n)
  (if (< n 3)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

Figure 5-1: The Scheme code for computing the Fibonacci numbers on a single computer.

```
(define fib
  (lambda
   (n)
   (cond
    [(<= n 2) n]
    [(<= n 41)
     (+ (fib (- n 1))
        (fib (- n 2)))]
    [else
     (let*
         ([f1 (spawn fib (- n 1))]
          [f2 (spawn fib (- n 2))]
          [v1 (touch f1)]
          [v2 (touch f2)])
       (+ v1 v2))])))
```

Figure 5-2: The Scheme code for computing the Fibonacci numbers using the distributed computation framework. Here, the Fibonacci code has been modified to use our framework. Small problem instances are directly evaluated, without the creation on new jobs. Larger instances are evaluated by spawning two subjobs, which may be evaluated in parallel. The cutoff of $n = 41$ for switching to direct evaluation was chosen such that each leaf job required on the order of 1 or 2 seconds of CPU time. Overhead is low and the granularity could easily be made more fine.

sockets. The results can be seen in Figure 5-3 and Table 5.1. Note how the speed increases linearly with the number of servers running.

Generally, as the number of servers used increases, the number of jobs which are stolen from one server by another server increases, as can be seen in Table 5.1. This is a randomized process, so the growth is not uniform. The total number of jobs was 5167.

5.3 Cloud Computing Platform

We configured our system to run on Amazon.com's Elastic Compute Cloud (EC2). This service hosts Xen virtual machines in an Amazon cluster. These virtual machines — called "instances" — are available in various sizes. We used the "Small Instance" (m1.small), which represents 1 EC2 compute unit, roughly the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. The host machine architecture is a 2.66GHz Intel Xeon E5430. Since the virtual machine is sharing the host machine with other virtual machines, the CPU time available to the virtual machine is substantially lower than the real time elapsed. The cost for a Small "On-Demand" instance is currently $0.085 per hour. However, "spot instances", with slightly lower reliably guarantees, are typically available from Amazon for only $0.030–$0.035 per hour.

The results can be seen in Figure 5-4 and Table 5.2. Note how the speed increases approximately linearly with the number of servers running, but with some substantial variation.

The variation in performance is due to differences in load balancing effectiveness. Figure 5-5 shows the CPU utilization in two different runs, each with 15 servers. In the run shown in Figure 5-5(a), utilization is uniformly high. In the run shown in Figure 5-5(b), utilization is approximately 70% for part of the run. This reflects different success at load balancing between the servers. The load balancing algorithm is randomized and different runs with the same configuration see different results. Nonetheless, in all cases yet observed, total average utilization has been in the range 75%–100%.

The load balancing issue is due to the network configuration. Since all servers are connected to the original client's server but not to each other, jobs need to be transferred through the central server. In the "bad" run, in Figure 5-5(b), 14 of the 15 servers finish their jobs efficiently. The last server has jobs remaining. The central server steals jobs off of this server. Thus, the utilization on the loaded server and the central server is high. Utilization on the other servers is below 100% as the central server does not offload all of the jobs quickly enough.

This dynamic can also be seen in Figures 5-6 and 5-7. Figure 5-6 shows the incoming and outgoing network traffic for each server in a typical "good" run. (The run shown used 20 servers.) Since all servers are connected to the initial client's server, that server has the highest network traffic. The initial server offloads tasks to the other servers, with few tasks going in the other direction. Thus, the other servers receive substantial incoming traffic from the initial server. The outgoing task objects include the source code archive and are relatively large — approximately 1.5kb. The task results returned to the initial server are quite small (less than 100 bytes), so network traffic in that direction is lower.

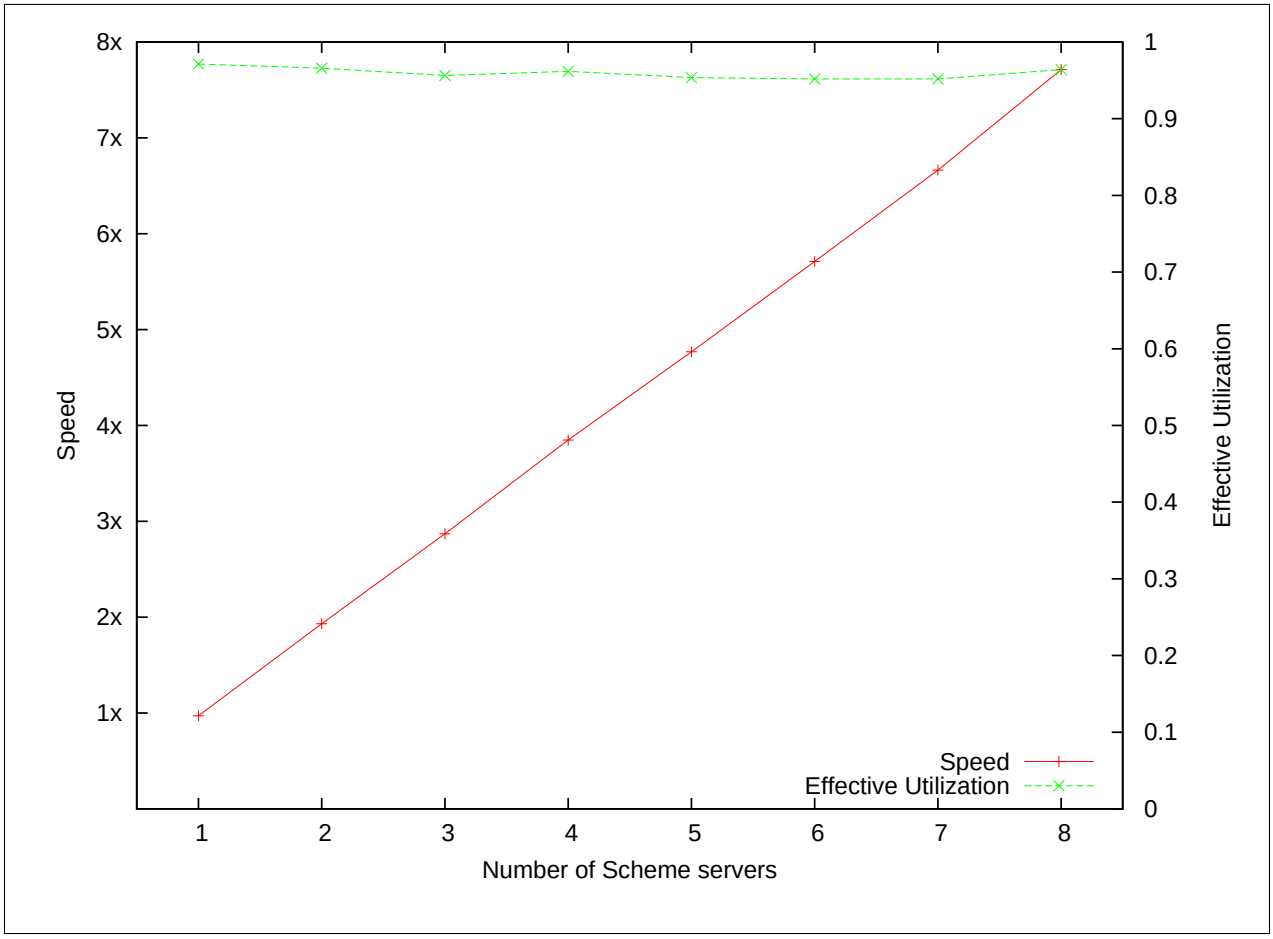


Figure 5-3: This graph shows the results of computing the 57^{th} Fibonacci number on an 8-core computer. The computation was run using between 1 and 8 Scheme computation servers. The left axis shows the speed of the computation, less overhead associated with the system, versus a single-threaded version. The right axis shows the effective utilization (i.e. speed divided by number of servers). Note that the effective utilization stays very close to 1. The raw data for this plot is given in Table 5.1.

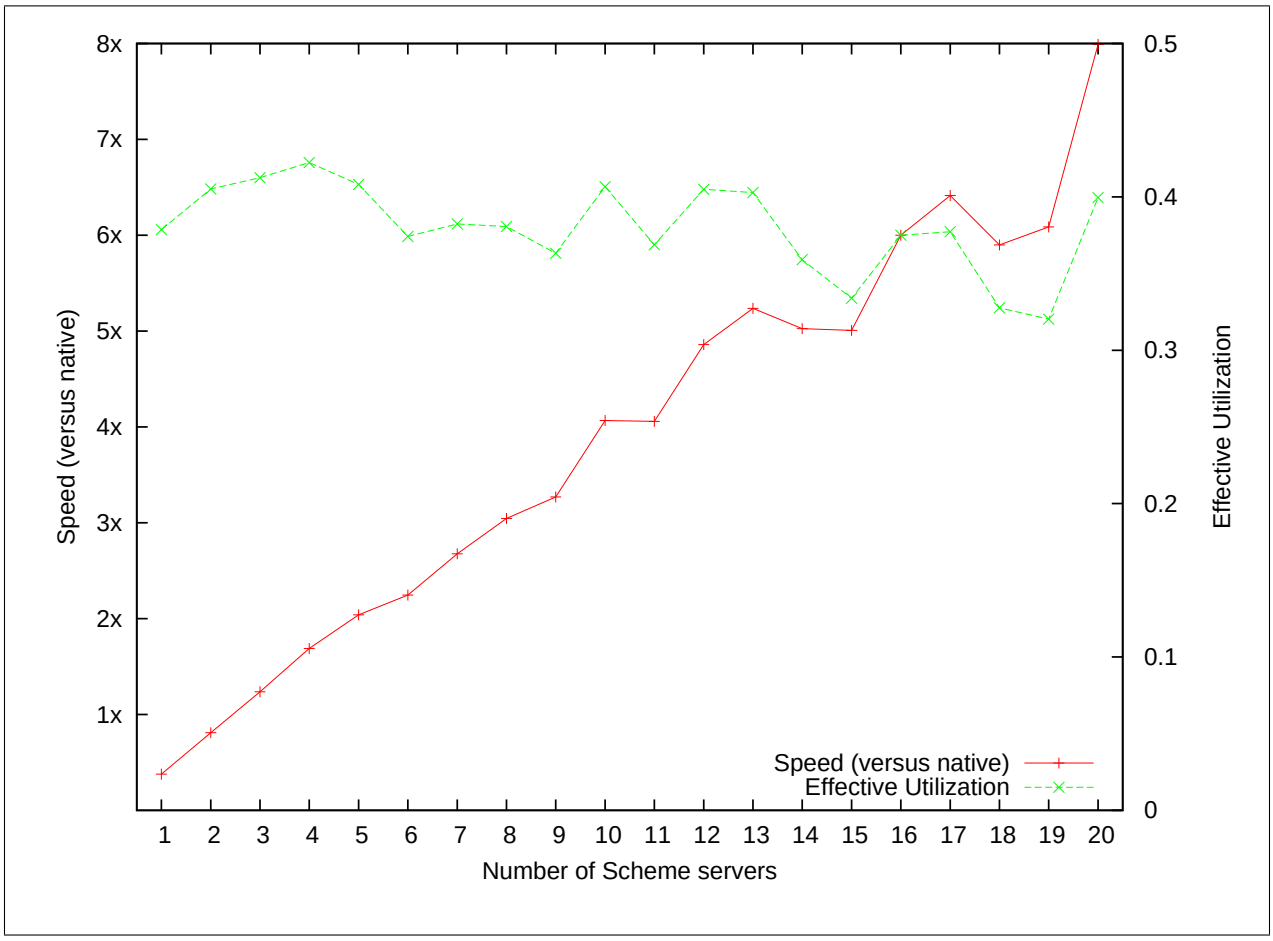
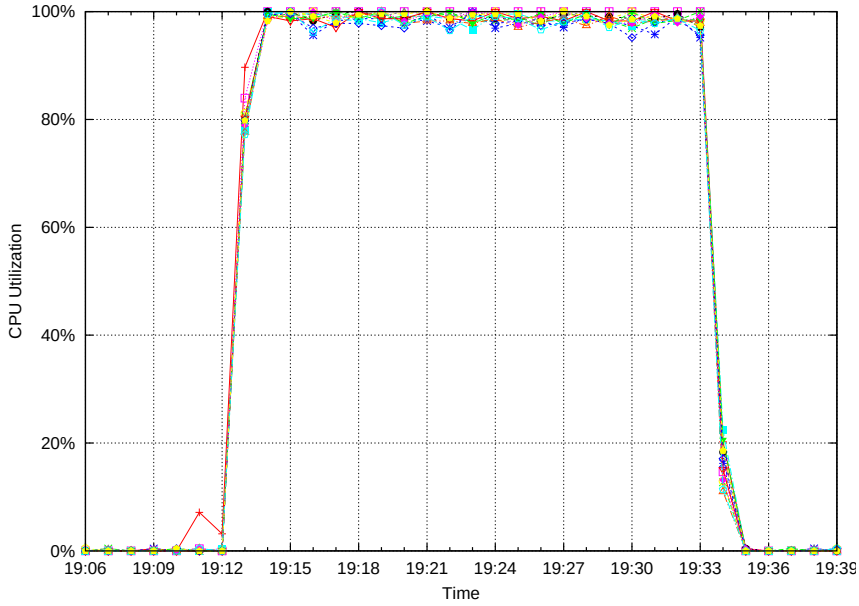


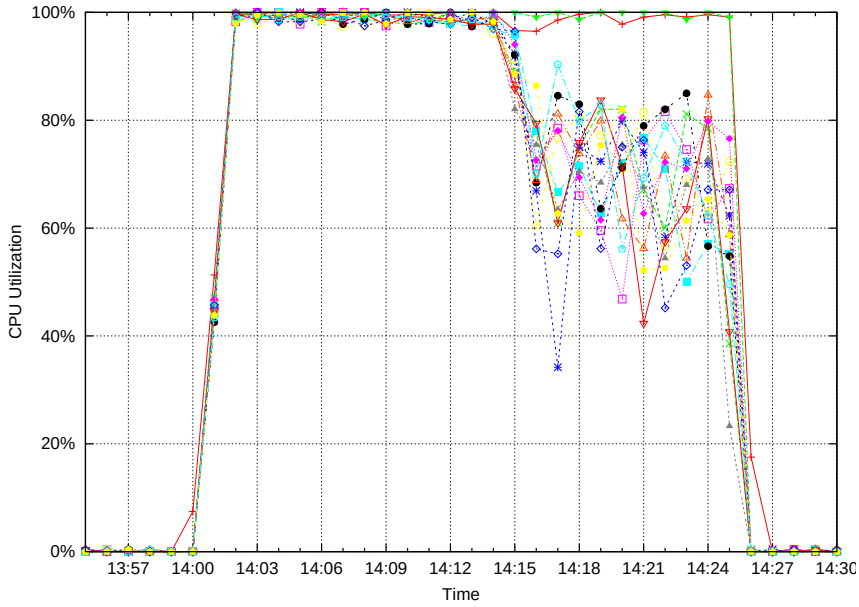
Figure 5-4: This graph shows the results of computing the 57^{th} Fibonacci number on Amazon EC2. The computation was run using between 1 and 20 instances (Xen virtual machines) each running one Scheme computation server. Each instance is a "Small Instance" with 1 virtual core. The left axis shows the speed of the computation as measured in CPU time (less overhead) divided by real time. Because the virtual machine has only part of the resources of the host machine, the effective speed of a single instance is less than 1. The right axis shows the effective utilization (i.e. speed divided by number of servers). The raw data for this plot is given in Table 5.2.

Figure 5-7 shows the network traffic in a typical "bad" run. (The run shown used 15 servers.) Part way through, most of the servers have finished their assigned tasks. One server has not. The central server then begins stealing tasks from the loaded server. The transfer of job objects out of the loaded server and back to the central server is the cause of the high outgoing traffic from the loaded server starting at around 14:15.

Future versions of the benchmark may connect all servers pairwise, to ensure that CPU utilization is more consistently uniform. Alternatively, the load balancing algorithm could be modified to steal tasks more aggressively.

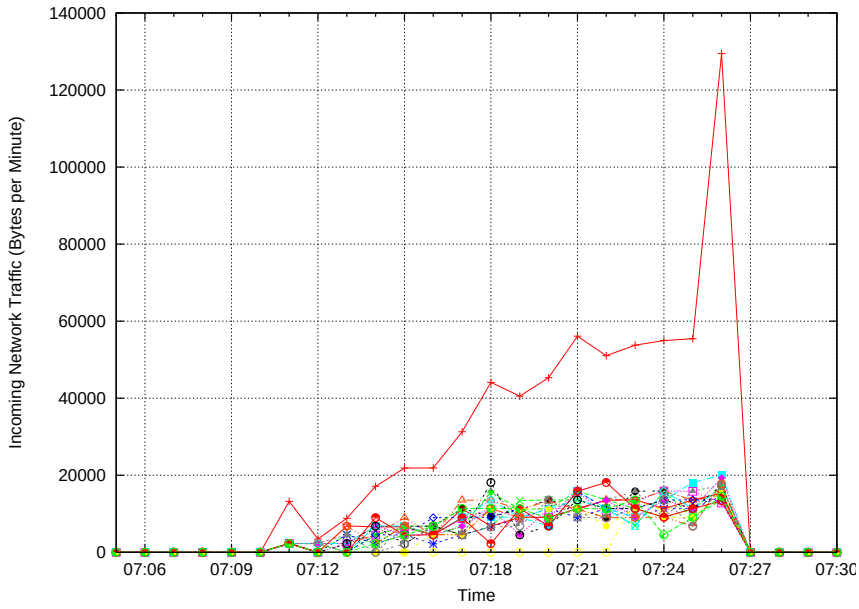


(a) CPU utilization for each server in a "good" run with 15 servers.

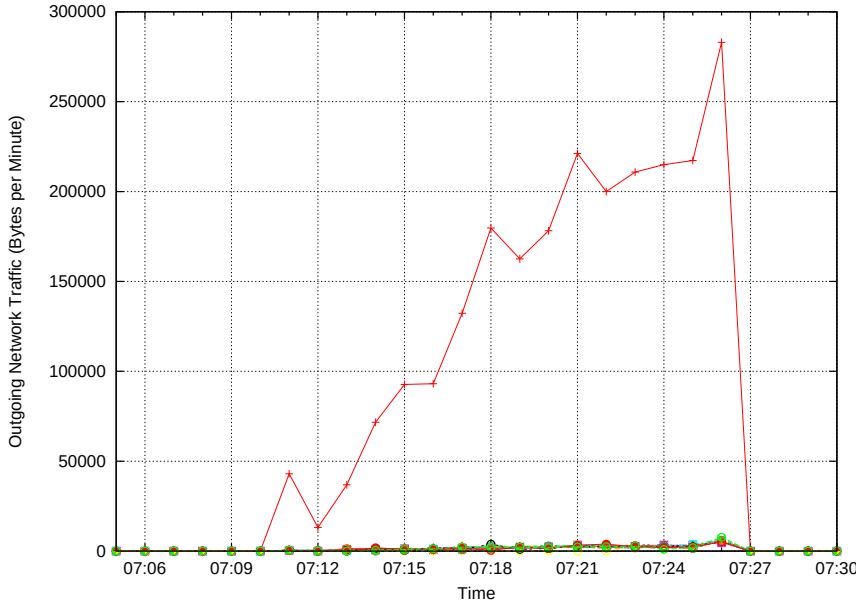


(b) CPU utilization for each server in a "bad" run with 15 servers.

Figure 5-5: The CPU utilization for each instance in two different EC2 runs. Both runs used 15 Scheme computation servers. In 5-5(a), the utilization is high and uniform for all servers throughout the computation. In 5-5(b), the utilization is initially high but falls half way through on most of the servers as they finish their assigned jobs. Jobs are transfered from the one busy server through the client's server to the other servers, keeping their utilization at around 70%. The run shown in 5-5(b) is the run used in Figure 5-4.

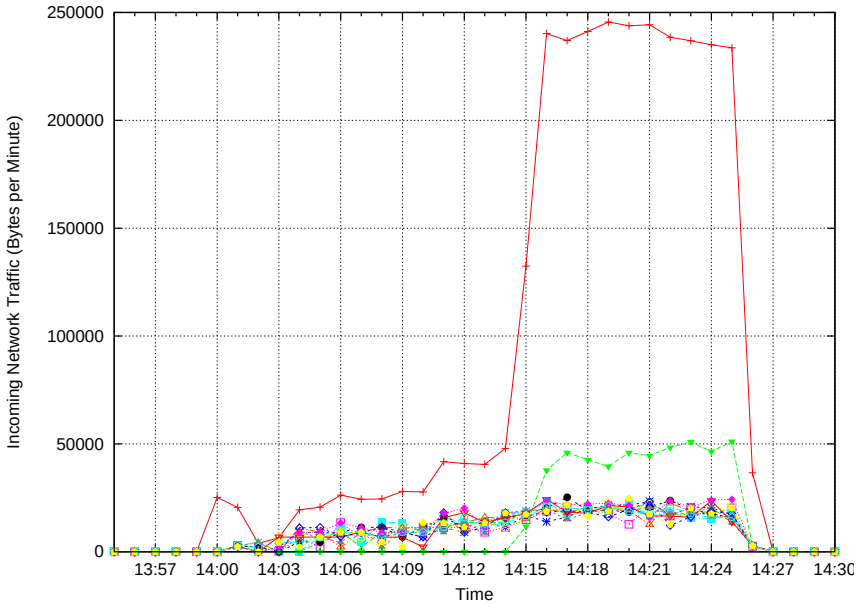


(a) Incoming network traffic.

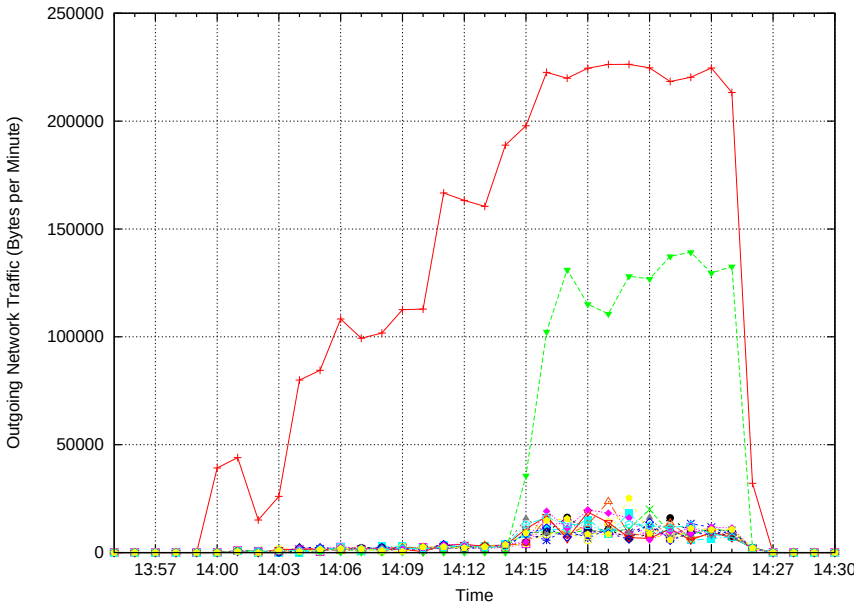


(b) Outgoing network traffic.

Figure 5-6: The network traffic between servers on EC2 in a "good" run. The run shown used 20 Scheme computation servers. Figure 5-6(a) shows incoming network traffic for each server while Figure 5-6(b) shows outgoing network traffic. Since all servers are connected to the initial client's server, that server has the highest network traffic.



(a) Incoming network traffic.



(b) Outgoing network traffic.

Figure 5-7: The network traffic between servers on EC2 in a "bad" run. The run shown used 15 Scheme computation servers. Figure 5-7(a) shows incoming network traffic for each server while Figure 5-7(b) shows outgoing network traffic. Since all servers are connected to the initial client's server, that server has the highest network traffic. Additionally, another server also has high traffic as it has high load and spends the end of the computation trying to give its tasks back to the central server.

Servers	1	2	3	4	5	6	7	8
Jobs	5167	5167	5167	5167	5167	5167	5167	5167
Job transfers over a socket	0	416	628	1136	1119	1320	1107	2154
Total CPU time (seconds)	7969	7945	7936	8480	7957	7927	7955	8110
Real time (seconds)	8204	4113	2766	2204	1669	1388	1194	1051
Effective CPUs	0.97	1.93	2.87	3.85	4.77	5.71	6.67	7.71
Effective utilization (%)	97.1	96.6	95.6	96.2	95.4	95.2	95.2	96.4

Table 5.1: This table shows the results of computing the 57^{th} Fibonacci number on an 8-core computer. The computation was run using between 1 and 8 Scheme computation servers. The number of subjobs is the same for all runs. The number of jobs which were serialized and transfered between computers is given in the "job transfers" line. The effective number of CPUs is the total CPU time (across all jobs) divided by the real time. The effective utilization is the effective number of CPUs divided by the number of cores. A plot of these data can be seen in Figure 5-3. For comparison, a C++ program for computing the 57^{th} Fibonacci number (also using the exponential algorithm) takes 1574 seconds on the same computer.

Servers	CPU time	Real time	Speed	Utilization
1	7434.73s	19639.20s	0.379x	37.9%
2	7151.28s	8824.60s	0.810x	40.5%
3	7208.14s	5824.74s	1.238x	41.3%
4	7444.81s	4406.99s	1.689x	42.2%
5	7174.01s	3515.75s	2.041x	40.8%
6	7199.08s	3206.13s	2.245x	37.4%
7	7189.30s	2686.28s	2.676x	38.2%
8	7200.47s	2364.20s	3.046x	38.1%
9	7425.61s	2271.59s	3.269x	36.3%
10	7188.80s	1767.60s	4.067x	40.7%
11	7347.79s	1811.08s	4.057x	36.9%
12	7171.81s	1476.00s	4.859x	40.5%
13	7217.64s	1378.30s	5.237x	40.3%
14	7330.73s	1458.53s	5.026x	35.9%
15	7403.51s	1478.41s	5.008x	33.4%
16	7258.04s	1209.85s	5.999x	37.5%
17	7248.08s	1129.99s	6.414x	37.7%
18	7392.60s	1253.36s	5.898x	32.8%
19	7402.66s	1216.32s	6.086x	32.0%
20	7197.00s	900.71s	7.990x	40.0%

Table 5.2: This table shows the results of computing the 57^{th} Fibonacci number on Amazon EC2. The computation was run using between 1 and 20 instances (Xen virtual machines) each running one Scheme computation server. Each instance is a "Small Instance" with 1 virtual core. The number of subjobs is the same for all runs. The speed is the total CPU time (across all jobs) divided by the real time. Because the virtual machine has only part of the resources of the host machine, the effective speed of a single instance is less than 1. The utilization is the speed divided by the number of servers. A plot of these data can be seen in Figure 5-4. For comparison, a C++ program for computing the 57^{th} Fibonacci number (also using the exponential algorithm) takes 4647 seconds of real time and 1753 seconds of CPU time.

Chapter 6

Conclusion

In this work, we first presented a distributed variable abstraction suited to functional programming. Then, we showed how to use this abstraction to build a sophisticated system for running distributed Scheme programs. This system used a network of Scheme computation servers which ran individual tasks as part of a larger program. Our framework implemented clean parallel "future" semantics for distributed computation on top of this network of servers.

6.1 Future Work

The two key next steps for this system are implementing better load balancing algorithms and better reliability. Figure 5-5 shows the importance of load balancing. Better configurations of network connections and better load balancing algorithms could improve performance by approximately 25% for the "bad" cases.

Additionally, we would like to improve the reliability of the system. Both computers and computer programs fail unpredictably. As the number of computers involved increases, so does the chance of a failure somewhere. However, functional programming is well suited to building a fault-tolerance distributed system. If jobs have no side effects, then a failed job can be restarted freely if needed. Some existing distributed systems, such as MapReduce [DG04], are similarly able to obtain fault-tolerance by eliminating side effects, and this is arguably a key component of their success.

Further ahead, we would like to be able to support more data intensive applications. Our system, as it is currently, is most effective on problems of high computational complexity, with small data sets and large amounts of CPU time. This lends our system to exponential search problems. However, we would also like to support problems which may not be as CPU intensive but which may have vast amounts of data. In our system, a computation starts on a single computer and move outwards. This would seem to require that the entire problem description fit on a single computer. For many applications with large amounts of data, such as web indexing, this may not be the case. Thus, we would like to augment our system with distributed data storage and data structures such that a computation may be performed on a data set which does not fit on a single computer.

Bibliography

[AZTML08] Abdallah D. Al Zain, Phil W. Trinder, Greg J. Michaelson, and Hans-Wolfgang Loidl. Evaluating a high-level parallel language (gph) for computational grids. *IEEE Trans. Parallel Distrib. Syst.*, 19(2):219–233, 2008.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995.

[BL99] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[CJK95] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-order distributed objects. *ACM Trans. Program. Lang. Syst.*, 17(5):704–739, 1995.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[CW03] Wahid Chrabakh and Rich Wolski. Gridsat: A chaff-based distributed sat solver for the grid. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 37, Washington, DC, USA, 2003. IEEE Computer Society.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. December 2004.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver, 2003.

[FMM91] R. Feldmann, P. Mysliwietz, and B. Monien. A fully distributed chess program, 1991.

[HMJ05] Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a shared-memory multiprocessor. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 49–61, New York, NY, USA, 2005. ACM.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. March 2007. European Conference on Computer Systems (EuroSys).

[McC] Jay McCarthy. Web server: Plt http server. Version 4.2.5, PLT website. `http://docs.plt-scheme.org/web-server-internal/`.

[Mor96] Luc Moreau. The semantics of scheme with future. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 146–156, New York, NY, USA, 1996. ACM.

[MSS00] ao P. Marques-Silva, Jo and Karem A. Sakallah. Boolean satisfiability in electronic design automation. In *DAC '00: Proceedings of the 37th Annual Design Automation Conference*, pages 675–680, New York, NY, USA, 2000. ACM.

[PGF96] Simon Peyton, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. pages 295–308. ACM Press, 1996.

[PLT] Reference: Plt scheme. Version 4.2.5, PLT website. `http://docs.plt-scheme.org/reference/`.

[Swa09] James Swaine. Scheme with futures: Incremental parallelization in a language virtual machine. Master's thesis, Northwestern University, December 2009.