4-17-2007

MPSCM: A Distributed Extension to MzScheme

Alexander Starche
Trinity University

MPSCM: A Distributed Extension to MzScheme

Alexander Starche

Abstract

MPSCM is an extension to the MzScheme dialect of Scheme that provides facilities for distributed programming with a message passing base and higher-level distributing constructs designed in a more functional style. This paper provides a description of the MPSCM environment and an analysis of the results in terms of performance, expressivity, and usability.

MPSCM: A Distributed Extension to MzScheme

Alexander Starche

Acknowledgments

Thanks to Dr. Eggen for the advice and direction, Glenn Kavanagh for the scripts, Dr. Lewis and Dr. Howland for reading, everyone who has taught me, and everyone who has listened to me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

MPSCM is posed as an experiment with MzScheme, as a means to assess the feasibility of building a functional-style distributed environment on top of an existing Scheme implementation. This paper investigates the issues of syntax and performance that arise from this approach. Another important issue is how well the MPSCM environment maintains the character and advantages of the MzScheme language and exploits the natural decomposability of Scheme programs. This analysis will focus on ease of use and functional programming style. Finally, this paper will examine the overall practical viability of an approach like MPSCM as a distributed programming environment.

1.1 Background

1.1.1 About MzScheme

MPSCM is a library of parallelizing functions for MzScheme, an implementation of the Scheme programming language developed by PLT-Scheme. MzScheme is R^5RS-compliant but provides a great deal of additional functionality that is helpful and, in some cases, necessary for creating a distributed programming environment. Like most Scheme dialects, MzScheme provides a read-eval-

print loop (REPL), interface. In a REPL a user enters a functional expression into a prompt and, in this case, MzScheme evaluates it and prints the result. MzScheme's non-standard functions include constructs for TCP/IP, threading, and synchronization, [6] all of which are important in MPSCM. It is the goal of this project to thoroughly explore the possibilities of MzScheme but to not unnecessarily restrict the scope of interest of this research. Non-standard functions that are only available in MzScheme should be used in isolated, replaceable, and necessary situations.

The abilities of MzScheme library functions very much define the implementation of MPSCM. It is meant to be an extension to this environment, not an offshoot. All library functions, with the exception of the boot functions, which use some BASH scripts with SSH to initialize the environment, are coded directly in MzScheme. The syntax and operation of the parallelizing functions follow the rules of MzScheme: MPSCM programs are MzScheme programs first. In some cases this presents some difficulties, the assessment and handling of which are essential to this project. All that said, the author of this paper makes no claim that MPSCM takes full advantage of MzScheme's capabilities in these areas and further investigation of all problems addressed would be welcomed.

1.1.2 MPI

The Message Passing Interface (MPI), is a standard for distributed libraries that is extensively used in distributed programming, particularly with C and FORTRAN. It is a standardization of the message passing paradigm, which is, itself, the standard approach to distributed computing. [16] Message passing is, at its core, a very simple approach to parallel computing. A group of networked computers, called processes or nodes, is initialized, each with a path of communication to each other node and a unique integer ID, or rank. In MPI programs, generally each node runs an identical program but has its own dataspace, in an approach called Single Program Multiple Data (SPMD). [18] Such a program generally contains conditional logic based on node rank with `0` being the rank

of the master, or user-occupied, node. The nodes communicate data with each other using send and receive functions that transmit a value along with header information: a tag, the rank of the sending process, information about the value being sent. Generally speaking, each node computes a subproblem of the computation and sends the value to another node to continue with the computation or to compile the final result.

The basic parallelization constructs implemented in MPSCM are derived from a subset of MPI but it does not adhere to the MPI standard. MPSCM uses message passing in an extremely simplified form. This message passing core forms the lowest levels of MPSCM. The ultimate goal is to provide as much of the capability of an MPI-style approach as possible while still preserving the functional programming experience, both in the strict sense of proper treatment of state and function calls, and in giving users, students, and educators a comfortable and expressive functional style environment in which to work. To balance this desire with an MPI-based framework, MPSCM uses additional layers of functionality to create typical functional parallel constructs that use message passing without explicit direction from the programmer to do so. However, it is possible to use MPSCM to write MPI-style programs, albeit with a simplified library.

By modeling the base environment after MPI, MPSCM fits into the basic expectations of a distributed programming environment and also our understanding of the physical structure of such an environment. That is, each node in the network has its own data-space and a set method for communicating data to other machines. This abstraction is exactly our understanding of the hardware involved: each physical machine has a processor and its own local memory and the nodes are connected in a network via TCP. This is attractive in certain ways for any distributed programming platform because it is simple and expressive and can be modeled intuitively. However, because this approach is so tied to the distributed environment, and to state and sequence across the nodes, it limits the aspects of Scheme, particularly the ability to program in a functional style, that allow for simple problem decomposition. This is one of

the central problems of this paper and of MPSCM itself. How can an MPI-like foundation provide the basis for a powerful, elegant, and functional distributed programming experience without losing the advantages of both Scheme and MPI?

1.1.3 Functional Programming

In evaluating the success of the MPSCM implementation, this paper will focus on the preservation of the functional programming experience. The essentials of functional programming are different to different people, and some are easier to pin down than others. A few characteristics that are generally associated with functional programming, and are especially of interest to distributed programming and the MPSCM project, will serve as a point of reference for this discussion.

Declarativeness

Declarativeness, as a characteristic of a programming language or a program, does not seem to have an agreed upon definition, but this thesis will use that of Barth, Nikhil and Arvind. That is, a declarative program describes *what* to do, not *how* to do it [1]. A declarative program describes a function, not how to compute it. A programming language is declarative inasmuch as it allows the programmer to program in a declarative manner. Strictly speaking, programming languages can not usually be classified as either declarative or undeclarative but, rather, more or less declarative. However, there is a clear divide with functional languages, like Scheme, on one side and procedural languages, like C, on the other.

The desire to maintain declarativeness produces an immediate problem for extending a functional programming language to allow for distributed programming. Distributed programming is essentially about "*how*" a programmer wants a program done. That is, by decomposing the problem into subproblems and distributing the work among a network of nodes. Beyond this, the problems are

a little more subtle but ultimately more meaningful. MPSCM must balance the desire for overall declarativeness with a useful interaction between the programmer and the distributed environment, as well as account for practical concerns of building this type of environment entirely on top of an existing one.

Referential Transparency

Referential transparency, like declarativeness, is a property of a programming language or program that is generally associated with functional programming. This property can be stated in several ways: either that "equals can be replaced with equals" [11] or that a function call can be replaced with its return value without changing the result of the containing function or program. Generally speaking, referential transparency involves the elimination of side effects [11]. For example, if a function prints something to standard output or sets the value of a variable that is used outside of the function, it is not referentially transparent. It is the second situation that is significant in MPSCM, as it is generally more important and more to the point with respect to distributed programming. Scheme is a programming language that allows for and encourages a high measure of referential transparency in programs, but also contains facilities for creating side-effects such as setting state variables.

In MPSCM, referential transparency is depended on as much as it is preserved, though both come into play. Referentially transparent programs are easy to decompose for concurrent evaluation because it is guaranteed that all dependencies are explicit in the definition of a function. Two function calls to referentially transparent functions can be evaluated concurrently if neither function call requires the result of the other in order to evaluate. This is not generally true of functions that are not referentially transparent. Under the stronger condition that a function not only does not mutate state variables but does not reference them either, said function can be evaluated on a remote machine without making any extra provisions, such as copying over variables.

Also, standard distributed computing constructs generally lack referential

transparency. In a typical distributed programming environment, such as MPI, each node has its own state variables. The approach taken in MPI is, generally, that nodes communicate the values of these variables between the nodes as appropriate, and use them to compute functions and eventually compile an answer. With simple message passing functions added to Scheme, this approach can be taken but is not a good fit. Also, this approach generally lacks referential transparency because the sends and receives are experienced as side-effects and the variable states must be tracked in the interim. MPSCM seeks an approach that avoids this type of state-tracking as much as possible, and to make local state variables on the individual nodes arbitrary to distributed computations. This is a key goal of the higher-level functions in MPSCM.

1.1.4 Parallel Functional Computing

Parallel programming with functional languages in general, and Scheme and its parent language LISP in particular, has a rich history including a number of languages, academic papers, and even specialized machines [9]. There have been many different approaches to this problem and several of them have been useful in the design of MPSCM. While MPSCM uses MPI as the point of reference for the basics of parallelization, earlier implementations of functional parallel programming environments provide the point of reference and the inspiration for the ultimate design and interface of MPSCM.

The parallelizing constructs discussed in M-Structures [1] and Mutilisp [15], both written in the 1980s, provide an essential base for the style of parallel programming that is drawn on in MPSCM. There are also several recent projects that provide very sophisticated environments for distributed programming in Scheme such as Termite Scheme [8] and Kali Scheme [5], which also provide a point of reference for this project.

Lazy Evaluation

At the center of many functional-style approaches to parallel functional computing is lazy evaluation. Lazy evaluation is not necessarily a parallelizing construct but the concepts are very much tied to each other. In lazy evaluation an expression is not forced to evaluate until the return value is needed, or asked for explicitly. When expressions are evaluated lazily more than one expression can be computed at once and other operations can take place while waiting for the evaluation of an expression. This is essential to parallel computing. Lazy evaluation is the core concept of Multilisp's `future` function [15] and a slight variation, what is called non-strict evaluation, is used in M-Structures [1]. Some modern functional languages, such as Haskell [12], use lazy evaluation implicitly, requiring no special instructions from the programmer or extra constructs to do so. These languages are especially suited for concurrent programming. MzScheme, like other Scheme implementations, is not such a language, and its evaluation is strict by default. It does, however, contain some constructs that are designed for lazy evaluation and others that can be used to design other lazy evaluation constructs. The MzScheme `delay` function even shares some of the terminology from Multilisp in its `promise` return value [6].

It is important to note that lazy evaluation is linked to the concept of threaded computation. Lazy evaluation is basically equivalent to evaluating a function in a new thread of control, synchronizing, and collecting the value in an imperative program. This is at the root of lazy evaluation's usefulness for concurrent programming, but also makes it more suited to parallel processing that occurs in a single dataspace, or in threads, rather than distributed processing. To implement lazy evaluation remotely, a distributed environment such as MPSCM must not only be able to evaluate expressions lazily, but also to evaluate remotely and retrieve the value.

Concurrent ML

Concurrent ML is an extension to the functional language Standard ML that allows for concurrent programming. It is an interesting case study in functional parallel and distributed computing because it uses very explicit parallelizing constructs while maintaining key aspects of the functional approach as well as the underlying, largely functional, base language. Concurrent ML uses threads and synchronization constructs to create concurrent programs [14]. Instead of designing the constructs around a concept of function evaluation, as in lazy evaluation, standard process control and synchronization methods are used. Concurrent ML fits this into the functionally-based framework of Standard ML in a fashion that is powerful and usable. Such an approach, if used well by the programmer, maintains the advantages of natural decomposability of functional programs, while clearly displaying the parallelizing mechanisms. This type of extension is also a reasonable choice for ML because, like Scheme and MzScheme even more so, ML contains imperative and state changing functions in its core language [6] [14]. This approach would be considered less declarative, in the terminology of this paper, than the lazy evaluation approach and the parallelizing constructs are in what could be called an imperative style [1], though well integrated into the functional framework of the base language.

Generally speaking, MPSCM is designed for a lazy evaluation approach to concurrent programming. However, the underlying functionality of MPSCM has much in common with Concurrent ML, in a simplified form. MzScheme is generally more suited to the Concurrent ML approach to concurrent programming because it has significant synchronization and threading constructs built in and evaluates strictly. Because MPSCM is so tied to the underlying language, as a set of library functions in the language, these characteristics will tend to show through. Also, the basis in message passing, a very explicit form of parallelization, further ties MPSCM to an explicit or imperative style. This will not necessarily produce a lot of direct similarities to Concurrent ML, which was not a major source in the development of this project, but it serves as a mainstream

example of an alternative approach to parallel programming in functional languages and a point of reference for more or less imperative style parallelization in a functional programming language.

1.1.5 Scheme and Education

The Scheme programming language is commonly used in education, especially as an introduction to functional programming languages and even to computer programming in general, famously as the language used in the introductory programming course in MIT's computer science curriculum, using *Structure and Interpretation of Computer Programs* as a text [10]. Scheme is typically chosen for this purpose because it does not bog down the the programmer with low-level details [3] and for its clear syntax and semantics [17], which allow it to be easily learned and simply expressive. While Scheme is suited for complex applications as well, MPSCM focuses particularly on creating an environment that preserves the ease of use of Scheme.

In the transition to distributed execution, MPSCM should provide an approach that does not mire the issue in details about the distributed environment. At the same time, MPSCM should represent problem decomposition as clearly as Scheme represents the problem itself. This approach is a departure from declarativeness, at least in the sense that it is usually understood. By requiring the programmer to focus on problem decomposition, MPSCM requires a framework for how the problem must be solved or at least a knowledge of how it can be broken up. The ideal distributed environment for Scheme, with respect to maintaining declarativeness, would look exactly like standard Scheme, but would automatically distribute work to remote nodes. This is impractical, first of all, from a hardware perspective. Network speeds, particularly in networks that are not at the absolute cutting edge of high performance computing make it difficult to efficiently distribute work in the general case. This is because the communication overhead that is associated with sending and receiving data over the network is enough to significantly slow down the computation if

distributing decisions are not made wisely. Since MPSCM is not designed for high-performance computing, where extremely fast networks would be available, communication overhead is of particular concern. Algorithmically decomposing an arbitrary Scheme function may be feasible for a threaded multi-processor environment, but in a distributed one it is only appropriate for a small subset of functions or for a very specially designed base language. This is especially critical in an environment like Scheme where many computations are quite small and should not be distributed. Also, such an environment is outside the scope of MPSCM, which is built on top of an existing Scheme implementation. Implicit parallelization is most practical at the lowest levels of a programming language and cannot easily be added to the top without modifying syntax. Still, this ideal is kept in mind, and addressed in the design of MPSCM as possible and appropriate. It may not be possible to implement an effective distributed Scheme that looks exactly like Scheme, but much can be done to make it look more like Scheme than MPI.

As opposed to creating an environment where problem distribution is hidden, MPSCM works toward an approach which is declarative with respect to problem decomposition. That is, that the program states how the problem is to be decomposed, not how to decompose it. This serves as an effective introduction to distributed programming and that a Scheme-based environment, particularly one implemented in the fashion of MPSCM, provides a basis for this that is meaningfully rooted in functional programming but generally applicable to other approaches to distributed programming. Because MPSCM has its base in message passing many of the lessons to be learned from it are applicable to the MPI environment, even as they are very different in program structure. The same issues of problem decomposition and distributed state apply, but MPSCM takes an angle that emphasizes the advantages of functional programming in those very areas.

1.2 Motivation

MPSCM is based on ideas very similar to the essential constructs that appear in some of previous implementations of distributed functional programming environments, but is not meant to be an equivalent environment. It is not nearly as complete or robust as, for example, Termite Scheme. The first and original goal of MPSCM is to use MzScheme build an MPI-like set of library functions for building distributed programs and then analyzing the performance and expressivity of such an environment, as well as analyzing to what extent it is still a meaningfully Scheme-like or MPI-like environment. The implementation of typical functional style parallelization is essential to this investigation in order to demonstrate how well such functions can be implemented. However, many of the functions used in MPSCM are substantially different from those presented in other implementations for several reasons.

First, the requirements of syntax and function definition in MzScheme and the underlying message-passing framework, largely dictate the inner workings and the interface of MPSCM. The second motivation behind MPSCM's constructs comes from a distinct line of inquiry. The environment for MPSCM looks both to previous academic research and to concepts in education for functional programming languages and parallel programming. MPSCM's constructs should provide a meaningful base for an introduction to parallel computing in functional programming languages and in general. This is not a pure distinction from previous implementations, but rather a difference in emphasis.

MPSCM is also intended to add to the overall discussion of distributed programming in Scheme, though not in the form of radical or profound change. This area of discussion, while it is rich with existing research, is in many ways not yet fully realized. Both in terms of an agreed upon standard, like MPI for message passing, or for taking advantage of certain applications, education in particular. MPSCM's very specific approach to distributed computing allows a unique perspective on some of the problems and solutions of distributed computing in Scheme.

Chapter 2

Implementation

MPSCM is built in three layers, each built directly on the one below it. The first layer is the communication layer, which is modeled after MPI sends and receives. This layer provides the interface for connections between the nodes. The first layer contains the functions `MPSCM-send` and `MPSCM-recv` as well as the boot function, `MPSCM-boot`. The second layer of MPSCM consists of a single function called `MPSCM-eval`. This function is a lazy evaluation construct that serves as the basis for the style of distributed programming advocated in MPSCM. The third layer is the top-level functions: `MPSCM-let`, `MPSCM-map`, and `MPSCM-branch`. These functions are designed around traditional Scheme constructs and programming styles, and are meant to provide a structured approach to distributed programming. The following sections provide an in-depth description of the MPSCM implementation.

2.1 Initialization

MPSCM is initialized by the programmer from the MzScheme console using the function call, `MPSCM-boot`. This function uses a combination of BASH scripts and MzScheme library calls and can be thought of as an equivalent to LAM/MPI's `lamboot` [4], except originating from MzScheme's REPL command

environment instead of a standard UNIX terminal as the point of execution. The computer attended by the programmer can be thought of as the master, although the MPSCM protocol allows styles of distributed programming besides master/worker. `MPSCM-boot` takes the file-name of a file containing a list of node addresses as an argument. This list must start with the originating node, as in LAM/MPI. Also, it takes a TCP port number, which will be used for all communications, and the number of nodes to be used. If the number is less than the number of nodes given by the list file, it will take that number from the beginning of the list, if it is more, an error will be reported.

The originating node uses SSH via a BASH script to start MzScheme on all nodes, and each node, including the master, executes a function called `MPSCM-init`. Every node opens a `tcp-listener` and then connects to each node with lower rank and accepts from each node with higher rank. This creates a network of nodes where every node has a TCP input and output port to every other node. This is accomplished with MzScheme's `tcp-connect` and `tcp-accept` functions [6]. Also, each node has a rank, given locally as `MPSCM-myrank`, and that rank is associated with every port to and from that node. Ranks are assigned, similarly to MPI, as integer values starting with 0 for the originating node.

Although MPSCM is designed for computers on a shared Linux domain able to execute programs remotely using SSH, much of the infrastructure could be adapted to a heterogeneous environment thanks to the portability of MzScheme. The only aspect that relies on homogeneity is the boot script initiated by the `MPSCM-boot` function.

2.2 Message Passing functions

In MPSCM, communication occurs via message-passing using `MPSCM-send` and `MPSCM-recv` functions, which are substantially similar in conception to MPI's send and receive functions. MPI is, itself, a very lengthy and complicated standard. This is a result of its need to provide a robust and general standard for

distributed computing [4]. MPSCM uses only two functions to approximate all of MPI's functionality. Send and Receive functions provide the basic functionality of message passing and are sufficient, not only for programming in an MPI style, but also for implementing higher-level functions using message passing as a base, the ultimate goal of MPSCM.

2.2.1 MPSCM-send

The `MPSCM-send` function is extremely simple. `MPSCM-send` takes, as arguments, the rank of the addressed node, the data object to be sent, and an identifying tag. It then packages the data object with the tag in a list and writes the list to the TCP output port associated with the addressed node. `MPSCM-send` is a non-blocking send, and all handling of the message on the sending end is performed by the MzScheme library functions and the TCP implementation. There are no guarantees or provisions made for sending of messages other than those provided by TCP [2].

2.2.2 MPSCM-recv

All the complexity of the message-passing level occurs on the receiving end. In early versions of MPSCM, the `MPSCM-recv` function consisted only of reading the input port associated with the process from which data was expected. MzScheme's implementation of `read` makes this a blocking operation that waits until data is written and then returns. This implementation is actually acceptable for many simple or specialized distributed programs and allows for a lightweight backbone for message passing-based environments.

However, this approach is too simple for a distributed environment that aspires to general usability or comparability to MPI. It does not scale well to complicated programs and does not provide enough flexibility for different patterns of communication and execution. Such a simple approach to message passing also produces problems in constructing higher-level abstractions for the MPSCM library. Because of this, MPSCM uses a more intricate system built

around receive buffers that store all incoming messages. `MPSCM-recv` takes a rank and an identifying tag as arguments. It first checks the buffer for a message with the specified tag and rank. If the message is not available, it waits until it is received and then returns the data from the message. MzScheme provides library functions for hash-tables and synchronization that make these buffers quite practical to implement [6]. Background threads are started on initialization of the MPSCM environment to handle the receive buffers. These threads wait, using event-based synchronization, on each of the input ports for incoming messages. When messages are received, they are then hashed in a shared map that uses semaphores for mutual exclusion.

This is one area where the programming of the MPSCM library itself strays particularly far from ideal functional programming. The use of buffers and multiple background threads is state-dependent, lacking in referential transparency. This is a necessary practical consideration that is justified by the requirements of the problem of creating a message passing interface and by the flexibility that it allows the programmer using higher-level functions. At the lowest levels, MPSCM is not significantly different from MPI, albeit much simpler, in terms of structure, and the approach is, in some respects, C-like. Early attempts to avoid this type of programming did not produce good results in either function or style. The ideal for the design of MPSCM is to provide a foundation, much like the code behind a typical Scheme implementation, for a functional programming environment. The MzScheme code in MPSCM plays the role of a lower-level language providing the base for a higher-level one. MzScheme, as with all Scheme implementations, has the flexibility to work in this style, and ultimately allows for a good mix of functional and imperative-style programming in the MPSCM libraries.

2.2.3 Technical Limitations

`MPSCM-send` and `MPSCM-recv` can only send and receive objects that can be written to MzScheme's `ports`. These are the serializable objects, or those that

can be represented as data in some static form. In MzScheme, this is a separate group from first-class objects, which include several object types, such as functions, threads, continuations, that can not be written to ports. Serializable objects in MzScheme are similar to those in languages like C: numbers, characters, strings, etc. Lists and vectors are also included in the list of serializable objects, as are quoted expressions.

2.2.4 Message Passing Examples

It is possible to write perfectly good programs in MzScheme using only the `MPSCM-send` and `MPSCM-recv` functions. Scheme in general is flexible enough to allow very C-like programming, and these functions allow for MPI/C-style programs looking something like this:

```
(begin
   (MPSCM-send proc-id 'Message tag)
   <perform some computation>
   (set! var (MPSCM-recv proc-id tag))
   <perform final computation>))
```

Also, one can imagine an MPI-style parallelization in a somewhat more functional style. Here's a trivial program of only two nodes where node 0 sends `value` to node 1, who sends back `f` of the received value:

```
;;worker snippet
(let ((received-val (MPSCM-recv 0 0)))
   (MPSCM-send 0 (f received-val) 1))

;;master snippet
(begin
   (MPSCM-send 1 value 0)
   (MPSCM-recv 1 1))

;;combination, SPMD function
(define mpi-fun
   (lambda (value)
      (if (= MPSCM-myrank 0)
```

```
(begin
   (MPSCM-send 1 value 0)
   (MPSCM-recv 1 1))
(let ((received-val (MPSCM-recv 0 0)))
   (MPSCM-send 0 (f received-val) 1)))))
```

This type of programming has some appeal, particularly for a programmer with some familiarity with both Scheme and MPI. However, it is generally difficult to deal with these sends and receives over an arbitrarily large network of nodes. Also, the complication of pairing sends and receives involves more "plumbing"[1] and concerns with sequence than what is desired in declarative Scheme programs.

2.3 Higher-level functions

While MPI-style programming is not only possible but reasonably user-friendly and expressive in MPSCM, it is ultimately not the desired application for this project. MPSCM is intended to preserve the functional, or functional-like, experience provided by MzScheme. Because of this, MPSCM also provides higher-level functions that are intended to help bridge this gap. These functions are based on past approaches to parallelization in functional programming and on commonly used standard Scheme functions. The goal of the higher-level functions of MPSCM is, first, to create a method for remotely evaluating function calls so that distribution can occur on the function-level instead of on the program and data level, as in MPI. The second goal is to identify naturally decomposable Scheme programming constructs and design distributed versions of them that exploit this decomposability.

2.3.1 MPSCM-eval

The building-block of the higher-level functions is `MPSCM-eval`. This function is meant to provide a way to evaluate a function on a remote node and then receive the result. This is essentially a paired send and receive, but in a structured

framework that is more fitting to the Scheme style. `MPSCM-eval` is a construct for enabling distributed computing via lazy evaluation [1]. Multilisp's `future` type is a good point of reference [15]. Because functions are not serializable in MzScheme, `MPSCM-eval` uses quoted expressions, which can be serialized, and the Standard Scheme function `eval` as a workaround. This also gets around the problem of MzScheme's strict evaluation because the quoted expressions are not evaluated until `eval` is called. `MPSCM-eval` is called with a quoted function call as an argument, which is packaged into an extended function call. This extended function call is an instruction to evaluate the original expression and send the result back to the requesting node using `MPSCM-send`. The extended quoted expression is then sent to the remote process with a special tag identifying it as a command to be evaluated. Any time such a message is received the quoted function is evaluated using the standard Scheme `eval` function in a thread and nothing is stored in a buffer. The return value of the `MPSCM-eval` function is a MzScheme construct called a `channel`. This construct has two pertinent associated functions: `channel-put` and `channel-get`. `channel-get` blocks the current thread of control until `channel-put` is called with some value in another thread and then returns the value that is put into the `channel` [6]. The process that calls `MPSCM-eval` will have a thread waiting, with `MPSCM-recv`, on the return value. When the value is received, it is put into the `channel`. when this value is needed it can be obtained by calling `channel-get` on the `channel` returned by the `MPSCM-eval` function. The value will only be available once because the `channel` is cleared when `channel-get` is called. `MPSCM-eval` can also be called by a process using itself as the remote process, this will merely evaluate the quoted expression in a thread and return a `channel` waiting on the result.

`MPSCM-eval` is intentionally more like a thread-based parallelizing construct than a distributed construct. A thread-like approach is inherently more friendly to a functional programming style than a traditional message passing approach because message passing is naturally imperative and state dependent. From the programmer's point of view, `MPSCM-eval` is very much like a thread with

a structured method of obtaining a return value, which is patterned on the MzScheme `channel` construct. This is driven home by the fact that when a process calls `MPSCM-eval` with itself as the remote node, this is literally what happens. `MPSCM-eval` provides a lazy evaluation construct for distributed computing that retains the advantages of typical thread-based lazy evaluation.

The only element of programming with `MPSCM-eval` that directly references the distributed environment is the assignment of a remote node. Since, for referentially transparent functions, the processor on which the computation is performed is arbitrary to the evaluation of the expression, `MPSCM-eval` can also be called without a node assignment argument. When `MPSCM-eval` is called in this fashion, it chooses a processor using the MPSCM library function `MPSCM-get-next-proc!`. The method of node selection currently used in `MPSCM-get-next-proc!` is to iterate over processes with each successive call to the function. This is done using the local state variable `MPSCM-next-proc`.

Distribution algorithms could be created and easily added to attempt to provide optimal load balancing in general or for specific applications. Such solutions have not been investigated in MPSCM as a means of providing optimal efficiency. However, the feasibility of this approach is investigated in the method `MPSCM-get-load`. This method obtains the current load of whichever node it is called on using standard UNIX `aux` and `ps` functions called from MzScheme. By calling this method with `MPSCM-eval`, one node can figure out the current load of any or all other nodes and make distributing decisions appropriately. This type of function could be extended to also check variables in a heterogeneous network, such as processor power and number of processors on a given node. This type of flexibility is ideal for MPSCM programming because it maximizes the effectiveness of MPSCM parallelizations and because it can do so behind the scenes, allowing the programmer to focus only on writing properly decomposed functions. However, because MzScheme should be effective in the most simple and general cases, high overhead load-distributing mechanisms are overkill. For this reason, MPSCM uses the simple rotating approach to load distribution.

The primary rationale for the implementation of the receive buffers and

threaded listeners used as the back end for `MPSCM-recv` is to allow for `MPSCM-eval` messages to be evaluated in threads independently of each other and of the data coming in through traditional message passing. This allows for an arbitrary number of threaded `MPSCM-eval` calls to be evaluated throughout the network and, as long as they are without side-effects, experienced much like threaded function calls, or lazy evaluation, by the user. For this reason, MPSCM automatically takes advantage of nodes with multiple processors without requiring additional commands. A node with two processors can simply be sent two subproblems and they will automatically be evaluated concurrently.

It is ideal for commands called with `MPSCM-eval` to not contain references to local state variables so that the node on which it is evaluated is, indeed, arbitrary. Programming in MPSCM is much friendlier when it is used in this fashion, and this emphasizes the natural parallelizability of proper functional programs. When this is done, an `MPSCM-eval` function can be called *transparent*. That is, it is equivalent to the *sequentialised* version [13]. If the `MPSCM-eval` call is replaced with `eval`, and the `channel-get` is removed, the result of the computation is the same from the user's perspective. However, as with Scheme in general, MPSCM offers some flexibility in this respect. It has not been a point of significant study, but `MPSCM-eval` could be used along with localized state values and specific node assignments to write effective distributed programs. Such programs would be much like MPI programs, but with a particular structure to communication that is more in the functional style.

A typical `MPSCM-eval` program:

```
(define MPSCM-eval-fun
   (lambda (args)
      ;; bind a helper function for building a list of
      ;; channels
      (letrec ((helper
         (lambda (hargs)
            (if (cond)
               '()
               ;; call MPSCM-eval on some sub-function
               ;; and cons it to the recursive call.
```

```
            (cons (MPSCM-eval '(f arg))
                  (helper
                     (iterating-function args)))))))
      ;; generate the solution by performing some
      ;; finalizing function on the list of channels.
      (finalizing-function
         (helper (setup-function args))))))
```

2.3.2 MPSCM-ceval

In addition to `MPSCM-eval`, there is the almost equivalent function provided in MPSCM called `MPSCM-ceval`, or compose and eval. Instead of an entire quoted function call, `MPSCM-ceval` takes a quoted function name and a list of arguments. `MPSCM-ceval` then composes the complete quoted expression for `MPSCM-eval` from the quoted function name and the list of arguments by `cons`ing them together. This function is, rather obviously, not significantly distinct from `MPSCM-eval`. It is intended as a convenience function to avoid some of the list manipulation headaches caused by `MPSCM-eval`. It should be noted that any functions in the list of arguments must be quoted and any function calls will be evaluated prior to distribution, per Scheme's rules for evaluation order.

A typical `MPSCM-ceval` program:

```
(define MPSCM-ceval-fun
   (lambda (args)
      ;; bind a helper function for building a list of
      ;; channels
      (letrec ((helper
         (lambda (hargs)
            (if (cond)
               '()
               ;; call MPSCM-ceval on some sub-function
               ;; and cons it to the recursive call.
               (cons
                  (MPSCM-ceval
                     'f (distributing-function  hargs))
                   (helper (iterating-function args)))))))
        ;; generate the solution by performing some
        ;; finalizing function on the list of channels.
```

```
(finalizing-function
   (helper (setup-function args))))))
```

2.3.3 Top Level Functions

Using `MPSCM-eval` as a building-block, MPSCM includes additional parallelizing mechanisms based on Scheme constructs and commonly used functions that can be naturally parallelized.

MPSCM-map

`MPSCM-map` is a parallel version of the `map` function, commonly used in Scheme, that maps a function on to each element of a list. `MPSCM-map` splits up the list into a specified number of sub-lists, uses `MPSCM-eval` to map a quoted function onto each of the sub-lists, and then recompiles the full mapped list. `MPSCM-map` is available in either lazy or strict form. The default `MPSCM-map` is strict, as is `MPSCM-strict-map`. The strict distributed maps return the mapped lists directly. They are called strict because they are experienced by user programs as a strictly evaluating function and because they do not require a function to finalize the return value. However, each uses lazy evaluation, via `MPSCM-eval`, to perform the mapping itself.

The alternate form, `MPSCM-lazy-map`, returns a list of channels representing the `MPSCM-eval map` calls on each sub-list. `MPSCM-lazy-map` is used much like `MPSCM-eval`, but, instead of finalizing with `channel-get`, a specialized function called `MPSCM-map-finalize` is used to finalize all of the `channels` and create the full mapped list. `MPSCM-lazy-map` has the same advantages and disadvantages as `MPSCM-eval` in that it returns control to the user immediately but requires the extra step of finalizing the value.

Each of these can be replaced by an equivalent call to a standard `map` function with the same result, although the `MPSCM-lazy-map` requires a call to `MPSCM-map-finalize`. Because of this, `MPSCM-map` is an ideal MPSCM function, as it can be applied by the programmer with only parallelizing hints, with

the choice to use MPSCM being the major decision, and without changing program's structure.

`map` is a naturally parallelizable function because each of the elements of the list has the mapped function applied to it independent of the others. It is also a commonly used function in functional programming and a parallel version can be used interestingly and expressively.

`MPSCM-map`, especially the lazy form, could be handled in a more complex manner to allow for pipelining of the original list's data to the remote nodes and for partial access to incomplete mappings, with elements becoming available as they are needed. In general it is difficult to design an ideal solution for pipelining, so I have chosen the most basic decomposition of the list: splitting it into equal parts. Access to incomplete mappings is a fairly complex problem that is outside of the scope of this paper. There is some room for flexibility and experimentation in implementation of other versions of `MPSCM-map` and it may be used as a starting point for more specialized solutions to similar problems.

A typical use of `MPSCM-map`:

```
(MPSCM-strict-map 'f lst)

;; or with MPSCM-lazy-map

(let ((a (MPSCM-lazy-map 'f lst)))
   (<do whatever you want>)
     (some-function (MPSCM-map-finalize a) args))
```

MPSCM-let

`MPSCM-let` is a parallel version of the Standard Scheme `let` construct where the argument of each binding must be a quoted expression, which is evaluated remotely using `MPSCM-eval`. As with a standard `let`, each of the expressions must be independent of each other. There are two versions of `MPSCM-let`: `MPSCM-strict-let` and `MPSCM-lazy-let`. The strict let binds the return values of the evaluated expressions to each variable where the lazy let binds `channel`s

in the fashion of `MPSCM-eval`. As with `MPSCM-map`, the unqualified `MPSCM-let` is the same as `MPSCM-strict-let`.

Typical functions using `MPSCM-let` and `MPSCM-lazy-let`:

```
(define MPSCM-let-fun
   (lambda (args)
      (MPSCM-let ((a '(f1 a-args)) (b '(f2 b-args)))
         (f3 a b other-args))))

(define MPSCM-lazy-let-fun
   (lambda (args)
      (MPSCM-lazy-let ((a '(f1 a-args)) (b '(f2 b-args)))
         (f3 (channel-get a) (channel-get b) other-args))))
```

`MPSCM-let` is based on the idea of parallel blocks from M-Structures [1] and `pblock`s from Multilisp [15], where each expression in a block of expressions is evaluated concurrently. However, `MPSCM-let` uses the existing `let` construct as a basis rather than building an entirely new one. The `let` construct is, itself, essentially parallelizable because each binding is independent of all other bindings. This is underscored by the existence of `let*`, where this is not the case. The definition of the `let` construct guarantees *"exploitable concurrency"* [18]. By using a well-known Standard Scheme construct, `MPSCM-let` allows for parallelization that makes sense to Scheme and to Schemers. This is also especially useful as an instructive example of problem subdivision for distributed programming because the decomposition is built into the construct itself.

Also, `MPSCM-let` is an excellent example of how MPSCM programs should ideally relate to standard Scheme programs. First, an `MPSCM-let` program is identical in structure to an equivalent Scheme program. With a strict `MPSCM-let`, this can be demonstrated by replacing it with a standard `let` and removing the quotes before the expressions. `MPSCM-lazy-let` requires additional calls to `channel-get` for each of the bindings. Looking at the extremely general example functions, we can imagine exactly what a Standard Scheme equivalent would look like. However, this is not necessarily the way a programmer would choose to write such a program in Standard Scheme. For example, in that context, it

might be more reasonable to forgo the `let` altogether and evaluate `f1` and `f2` as arguments of `f3` or even to write a separate function that is the equivalent to the composition of the three.

`MPSCM-let` provides a construct that may occasionally be easily and effectively applicable to existing Scheme programs, but generally will require the programmer to make parallelizing decisions in the design process using an environment that is expressive and meaningfully Scheme-like. Also, the `let` construct encourages a structured approach to parallelization in a way that parallel blocks do not. The programmer must identify independently evaluatable subexpressions of appropriate computational complexity and use the `MPSCM-let` construct to decompose and distribute them. The `let` construct itself provides the decomposition's structure and distribution is handled implicitly by `MPSCM-eval`. This again underscores the academic applications of this construct and the MPSCM environment that are not present in an MPI approach.

MPSCM-branch

`MPSCM-branch` is a specialized construct designed for branching recursion that distributes the branches, using `MPSCM-eval`, and combines the results with a given function. In branching computations it is inefficient to distribute the sub-problems at every branch unless each branch involves a great deal of computation, which is often not the case. A solution-space search provides a good example of a typical case where each branch often contains only trivial computation. Because of this, there should be some rule for when to distribute the sub-problems and when to go ahead with computing them locally. A call to `MPSCM-branch`, therefore, looks like this:

```
(MPSCM-branch conditional
              combining-operator
              'exp1 ... 'expn)
```

Where `exp1...expn` are functional expressions that are evaluated remotely if the `conditional` evaluates to true and locally if it evaluates to false. After

this, `channel-get` is called if necessary and the results are combined using the `combining-operator`, which must be a function that can be applied to the return values of the quoted expressions. `sum` and `max` are examples of typical combining operators.

The `MPSCM-branch` function can be fairly easily implemented using `if` statements and `MPSCM-eval`, but the creation of a construct specifically for this situation is important for several reasons. First, by providing upper-level constructs that represent ideas that are essential to both Scheme programming and to parallel programming, MPSCM encourages a structured approach to these problems that takes advantage of the natural decomposability of typical Scheme functions. Distribution of branching recursion is an essentially functional method of problem decomposition that is useful for a wide variety of problems and can be adapted to different programming environments, though generally not so readily as with `MPSCM-branch`. Also, like the strict versions of `MPSCM-map` and `MPSCM-let`, `MPSCM-branch` can be seen as equivalent to the sequential version. This is a little more obscure in the case of `MPSCM-branch` than with the other two functions because `MPSCM-branch` is contrived to fit the `MPSCM-eval` framework rather than adapted from an existing function. However, it is basically a wrapper for the `combining-operator` and evaluates equivalently to:

```
(combining-operator exp1 ... expn)
```

In fact, while `MPSCM-branch` is named and designed for branching recursion, it can be used to apply the `combining-operator` to an arbitrary set of quoted expressions, somewhat like the `MPI_Reduce` function in MPI [16], which combines the value of every local instance of a given variable using one of a defined set of combining operators. However, `MPSCM-branch` is used in a very different, much more functional, context.

2.4 The Distributed Environment

MPSCM should, as much as can be reasonably done, provide an environment that is as close to the Standard Scheme experience as possible. While completely hiding the fact that these computations are going on on multiple computers, with their own dataspace, is not practical and ultimately not desirable given our capabilities and framework, MPSCM must still provide some basic functions that allow programmers to go about there state-based business in a comfortable and useful way. `MPSCM-eval-all` and `MPSCM-gload` are MPSCM's functions for dealing with state across all nodes or "global" state. Also, the function `MPSCM-halt` is used to stop MPSCM altogether.

2.4.1 MPSCM-eval-all

`MPSCM-eval-all` is the more general method for dealing with the distributed environment. It is used to evaluate a quoted expression, using `MPSCM-eval`, on every node. With `MPSCM-eval-all` the channels returned by `MPSCM-eval` are simply lost, it is meant only as a maintenance function. It can be used as an entry-point for an MPI-style SPMD program where each node is given a function with conditional commands based on rank, perhaps, that produce a meaningful final result. MPSCM, particularly for evaluating user functions, is built around more of a Master/Worker model of designing programs, so `MPSCM-eval-all` is deemphasized for SPMD programming. However, this could potentially be developed into a useful approach.

2.4.2 MPSCM-gload

`MPSCM-gload` is the global load method. This is a convenience method that is built directly from `MPSCM-eval-all` and merely evaluates a `load` call on all nodes. Paths given to `MPSCM-gload` must be relative to the `MPSCM-home-dir`, which is kept as a state variable on each node once the MPSCM libraries are loaded. This function is essential so that `MPSCM-eval` can be called with user-

defined functions.

2.4.3 Notes on Global State

Beyond the use of `MPSCM-gload` for creating a common library between the nodes, MPSCM is not designed to handle global state. In fact, it is not designed to deal with state at all, with the exception of the use of simple variables on the user-occupied node to conveniently keep track of values and hold onto `channel`s within the REPL environment. The higher-level functions are designed specifically so that state complications that naturally arise from a distributed environment are discouraged and can be avoided.

It could potentially be interesting to experiment with MPSCM using global `define`s and `set!`s to create a more developed and meaningful idea of global state but that is not within the scope of this project. MPSCM adopts an idea of distributed functional programming that shuns such notions and emphasizes the expressivity and inherent parallelizability of a stateless approach to parallel programming.

2.4.4 MPSCM-halt

The `MPSCM-halt` uses `MPSCM-eval-all` to call a function named `MPSCM-fin` on all remote nodes and on the master. `MPSCM-fin` simply closes all of MPSCM's TCP ports, stops the receive buffer threads and, for the remote nodes, exits MzScheme. All of the output generated during the remote MzScheme sessions is received, and printed to the REPL console of the master, when `MPSCM-halt` is called because that is when the boot scripts actually finish executing and their output is available.

Chapter 3

Example Programs

3.1 Numerical Integration

Numerical integration, particularly using the trapezoidal rule, is often used as an introduction to parallel programming because the decomposition is easily derivable from the description of the algorithm. Two different examples of numerical integration using MPSCM are included in Figures 3.1 and 3.2, one distributed and one using only threads, along with a C/MPI example in Figure 3.3 to serve as a point of comparison between the two environments.

The function `par-trap` in Figure 3.1 is a simple distributed implementation of numerical integration with the trapezoidal rule using MPSCM. The listed function uses the MPSCM libraries and a standard Scheme function `trap` (not listed):

```
(trap left-bound right-bound number-of-subdivisions f)
```

which approximates the integral of `f` from `left-bound` to `right-bound` using the given `number-of-subdivisions`. The listed function, `par-trap`, takes an additional parameter, `nparts`, which gives the number of sub-problems to break the function into. Each sub-problem will be composed into a call to `MPSCM-eval`.

```
(define par-trap
   (lambda (l r n f nparts)
      ;; compute parameters for the subproblems
      (let* (  (steps-per-part (/ n nparts))
               (step-size (/ (- r l) n))
               (part-size (* steps-per-part step-size)))
         ;; helper function generates one subproblem call
         ;; and evaluates it with MPSCM-eval
         (letrec ((helper
            (lambda (p)
               ;; compute specific subproblem parameters
               ;; and build the quoted expression
               (let* (  (start (* part-size p))
                        (finish (* part-size (+ p 1)))
                        (eval-list (list
                                    'trap
                                    start
                                    finish
                                    steps-per-part
                                    'f))
                        ;; call MPSCM-eval and hold the
                        ;; channel
                        (chan (MPSCM-eval eval-list)))
                  ;; recursive calls must be done such that
                  ;; all MPSCM-eval calls are made before
                  ;; waiting for any to evaluate
                  (if (= p 0)
                     (channel-get chan)
                     (let ((next-val (helper (- p 1))))
                        (+ next-val
                           (channel-get chan))))))))
         (helper (- nparts 1))))))
```

Figure 3.1: A distributed solution for numerical integration in MPSCM using the trapezoidal rule. Uses `MPSCM-eval` and a recursive helper function to create and evaluate subproblems.

```
(define thread-trap
   (lambda (l r n f nparts)
      (let* (  (steps-per-part (/ n nparts))
               (step-size (/ (- r l) n))
               (part-size (* steps-per-part step-size)))
         (letrec ((helper
            (lambda (p)
               (let* (  (start (* part-size p))
                        (finish (* part-size (+ p 1)))
                        (eval-list (list
                                    'trap
                                    start
                                    finish
                                    steps-per-part
                                    'f))
                        (chan (MPSCM-eval eval-list 0)))
                  (if (= p 0)
                     (channel-get chan)
                     (let ((next-val (helper (- p 1))))
                        (+ next-val (channel-get chan))))))))
         (helper (- nparts 1))))))
```

Figure 3.2: A threaded solution for numerical integration in MPSCM using the trapezoidal rule. Uses `MPSCM-eval` with node `0` as the assigned node to evaluate locally.

```
int main(int argc, char** argv)
{
   //Declare all necessary variables
   int myrank, nprocs, n, localn, i;
   double l, r, value, h, locall, localr, totalvalue;

   MPI_Status status;

   //Initialize the MPI environment
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

   //Get parameters from command line arguments
   l=atof(argv[1]);
   r=atof(argv[2]);
   n=atoi(argv[3]);
   h=(b-a)/n;

   //Compute the parameters for this process's
   //suproblem
   localn=n/nprocs;
   locall=l+myrank*localn*h;
   localr=locall+localn*h;

   //Compute the subproblem using unlisted Trap
   value=Trap(locall, localr, localn, h);

   //"Master" waits for responses from "Workers"
   //and sums the results
   if (myrank==0)
   {
      totalvalue=value;
      for(i=1; i<nprocs; i++){
         MPI_Recv(&value,
                  1,
                  MPI_DOUBLE,
                  MPI_ANY_SOURCE,
                  0,
                  MPI_COMM_WORLD,
                  &status);
         totalvalue+=value;
      }
   }
```

Figure 3.3: A distributed solution for numerical integration in C/MPI using the trapezoidal rule. Subproblems are computed at each node, using `MPI_Comm_rank` to differentiate, and sent to the master with `MPI_Send` where the answer is compiled and printed.

```
    //"Worker" processes send result to "Master"
    else
    {
       MPI_Send(&value, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }

    //The result is printed to standard out of the "Master"
    if (myrank==0)
    {
       printf("The value of the integral is %.16f\n",
                n, totalvalue);
    }

    //Finalize MPI
    MPI_Finalize();
}
```

This example is a typical and straightforward use of the MPSCM libraries using `MPSCM-eval` to build a list of `channel`s waiting on the completion of the sub-problem. When the list is built, each `channel` is finalized with `channel-get` and the sum of the results is returned as the approximation. The call to `MPSCM-eval` is done without a specified node ID so the next one is selected using `MPSCM-get-next-proc!`. Also notice that this function can be changed to a purely threaded function by using `0` as the second argument to `MPSCM-eval` as in `thread-trap`, shown in Figure 3.2. In the `thread-trap` example, each sub-problem is executed locally in a thread. The same approach could be used to perform the computation in threads on a remote machine, using the rank of another node instead of `0`.

A distributed solution for numerical integration programmed in C/MPI is available in Figure 3.3. The C program depends on an unlisted sequential function for computing the trapezoidal rule, called `Trap`, that is essentially the same as the unlisted Scheme function. Once past the syntax and program structure of the two examples, it is worth noting that these programs are quite similar in there approach to parallelization. They both decompose the problem in the same manner prior to execution of sub-problems, and then sum the results. Also, both problems are comparable in complexity: given a similar familiarity with C/MPI and MzScheme with MPSCM, neither is particularly more or less complicated than the other.

However, this example reveals many essential dissimilarities between the C/MPI approach and the `MPSCM-eval` approach to parallelization. First, the MPSCM version is explicitly decomposing and distributing the problem, whereas in the C/MPI version distribution is, especially in this example, primarily a product of the SPMD execution environment. This distinction is related to the essential difference between the two environments: MPSCM distribution is based on function evaluation, while MPI is based on data communication. In MPI, the distributed evaluation happens at the program level by running the parallel program on all nodes, which conceptually separates distribution from the description of the program. Because of this, MPSCM more clearly repre-

sents problem decomposition in terms of the problem, rather than in terms of the distributed environment.

3.2 Map and Sort

```
(define map-and-sort
   (lambda (ls)
      (let* (  (gran (/ (length ls) 10))
               (split-ls (split-list ls gran)))
         (merge-all (MPSCM-map 'sort split-ls gran)))))
```

Figure 3.4: A novel approach to distributed sorting in MPSCM that uses `MPSCM-map` to map a sort function to sublists.

The function `map-and-sort` in Figure 3.4 shows a novel approach to sorting using `MPSCM-map`. It uses three standard Scheme functions that are not listed: `split-list` which splits a list into a list of the specified number of sub-lists, `merge-all` which merges a list of sorted lists into one sorted list, and `sort` which sorts a list. Using these functions, `MPSCM-map-and-sort` splits up the list passed as an argument and maps the `sort` function to each element of the list of lists returned by `split-list`, using `MPSCM-map`. Finally, it merges the sorted lists returned directly by the strict version of `MPSCM-map`.

This function is interesting for several reasons. First, once implemented, it can be used in place of `sort` in any Scheme function, as long as the MPSCM environment is initialized. This portability is a nice side-effect that is allowed by the Scheme environment. The modularity of MPSCM is not totally unavailable in an environment like MPI, but the ability to implement parallelization at the function level is a gain in terms of usability. Also, the `map-and-sort` function does not take any arguments other than the list itself. All of the parallelization decision-making occurs automatically through the `MPSCM-map` function and in the `map-and-sort` function definition, which decides on a granularity, given by `gran`, the value passed to `MPSCM-map`.

This function is also interesting in that it is, even for a Scheme programmer, a decidedly off-the-wall approach to sorting. At the same time, it is essentially appropriate to the MPSCM approach. The programmability of a function such as `map-and-sort` is not a great accomplishment in itself, but it shows how MPSCM encourages novel ways of thinking about parallelization within the context of simple Scheme constructs. This function may itself be something of a joke, but it is a compelling joke that emphasizes the connection between a Scheme approach to programming and problem decomposition for distributed computing.

3.3 Fibonacci

```
(define par-fib
   (lambda (n)
      (if (<= n 1)
         1
         (MPSCM-branch (> n 15)
                       +
                       (list 'par-fib (- n 1))
                       (list 'par-fib (- n 2))))))
```

Figure 3.5: An MPSCM function to find the n^{th} Fibonacci number that uses `MPSCM-branch` for branching recursion.

The `MPSCM-branch` function can be difficult to grasp because it is not directly connected to a Standard Scheme construct or commonly used function. Because of this, it is appropriate to investigate it with a simple example that is often used as an introduction to recursion: the Fibonacci sequence. An MPSCM function to find the n^{th} Fibonacci number is given in Figure 3.5. This example is quite straight-forward: the base case is coded as in a standard Fibonacci definition and instead of recursive calls there is a call to `MPSCM-branch`. The condition argument is `(> n 15)`, so if `par-fib` is called to compute any Fibonacci number greater than fifteen the recursion branches using `MPSCM-eval` to evaluate the

recursive expressions. For n smaller than fifteen, the recursive calls to `par-fib` are evaluated locally using `eval`. This number can be adjusted based on the problem and the computing resources available to try and reach an optimal balance between distributing work and communication and thread overhead.

The two expressions are quoted recursive calls for the $n - 1^{th}$ and $n - 2^{th}$ Fibonacci numbers, which will be combined with the + operator. The use of the `list` method to form the quoted expressions is necessary because `(- n 1)` and `(- n 2)` must be evaluated prior to the calls to `MPSCM-eval` while the recursive calls to `par-fib` must not. This complication is the same as the one addressed by `MPSCM-ceval`.

This function is appealing because it is very similar to the typical standard Scheme description of the Fibonacci sequence, where the `MPSCM-branch` call would simply be replaced by:

```
(+ (par-fib (- n 1)) (par-fib (- n 2)))
```

Although, granted, the standard Scheme function would probably be given a different name. Also, the distributed function clearly describes how the decomposition takes place and is natural to the problem itself. The presence of the conditional operator isolates problem decomposition and distribution decisions to the call for parallelization and gives a simple method for adjusting granularity in terms of the structure of the recursive solution.

Chapter 4

Performance

The primary goal of MPSCM is not to provide a competitive alternative to quicker approaches to parallel programming, such as C with MPI, for high-performance computing. Existing Scheme implementations and MzScheme in particular are not currently competitive with C in this respect, particularly in areas, such as pure number-crunching, where parallel programming is often used. However, parallel programming is always about speed and MPSCM is no exception. MPSCM may not be able to compete with C for performance, but it does provide good speed up over sequential MzScheme under normal circumstances. MPSCM allows us to assess the performance capacities of MzScheme's TCP/IP functions, in particular, as well as some its the other advanced constructs. This chapter will also include an evaluation of the performance of MPSCM's higher-level functions.

4.1 Comparison to MPI

Initial performance data was collected with a bare-bones early version of MPSCM that provided Send and Receive commands, a lá MPI, using only `read`s and `write`s to TCP ports for communication and without buffers or advanced synchronization constructs. Data from these comparative tests, using equiva-

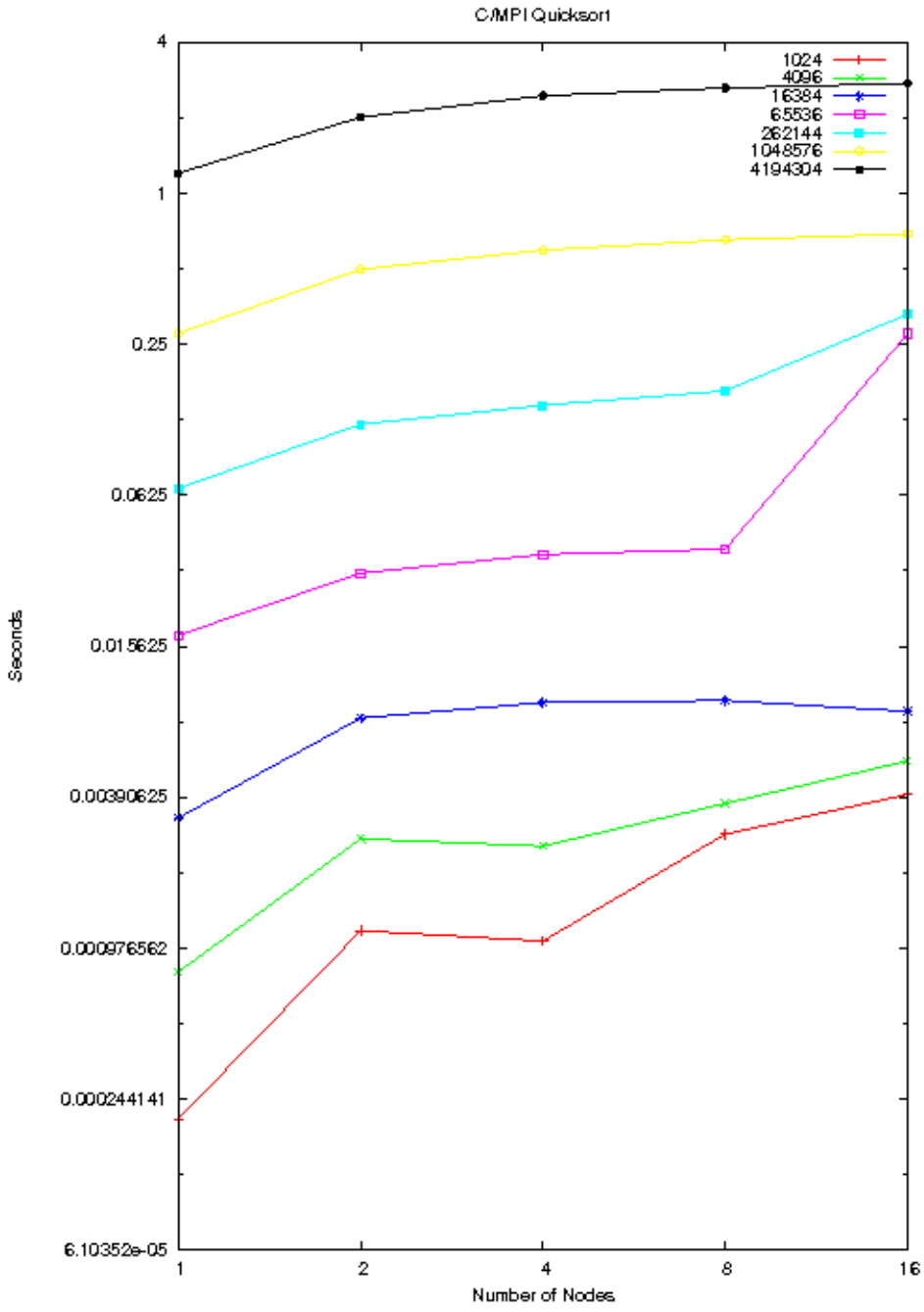

Figure 4.1: Plot of MPI quicksort timing data using variably sized integer arrays and networks of 1-16 nodes. Sequential C is used for the single processor data. Tabular data is available in Table A.1.

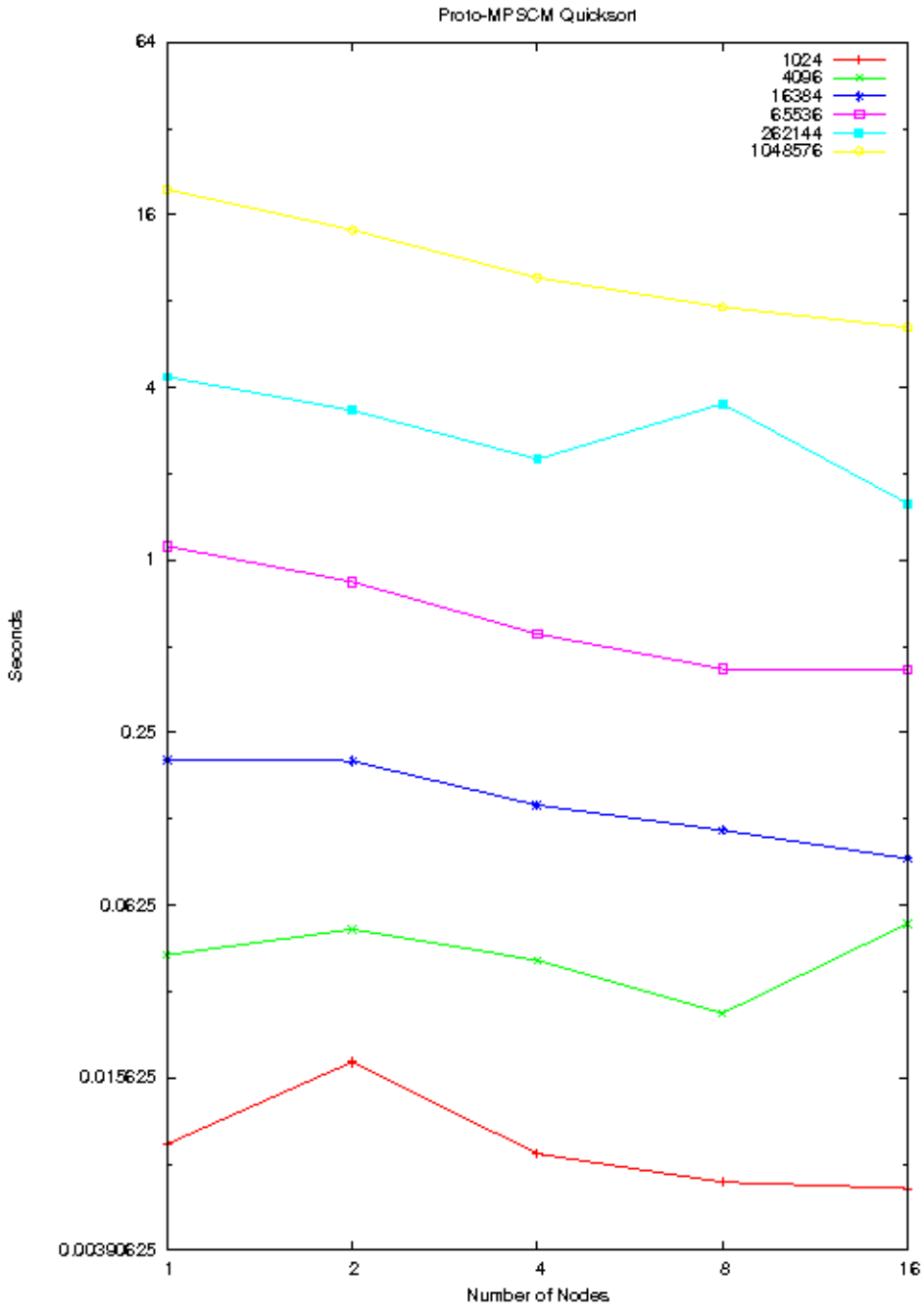

Figure 4.2: Plot of Proto-MPSCM quicksort timing data using variably sized number vectors with networks of 1-16 nodes. Standard MzScheme is used for the single processor example. Tabular data is available in Table A.2.

Elements	Nodes			
	2	4	8	16
1024	0.178166	0.195590	0.073329	0.050911
4096	0.294670	0.314755	0.212968	0.144195
16384	0.399727	0.346393	0.339684	0.375757
65536	0.565072	0.475743	0.453099	0.062765
262144	0.555342	0.466856	0.408467	0.202023
1048576	0.555374	0.465875	0.424097	0.402039
4194304	0.597763	0.490818	0.457380	0.438062

Table 4.1: Speedup for MPI distributed quicksort relative to Sequential C quicksort. Speedup is given as Sequential Execution Time / Execution Time.

Elements	Nodes			
	2	4	8	16
1024	0.517768	1.083323	1.361667	1.431535
4096	0.813840	1.044780	1.596558	0.776728
16384	1.008935	1.438315	1.758998	2.207607
65536	1.330426	2.022788	2.677692	2.685742
262144	1.309208	1.937149	1.240066	2.762406
1048576	1.382668	2.031565	2.571014	3.023777
4194304	1.375793	2.085849	2.583813	3.030176

Table 4.2: Speedup for Proto-MPSCM distributed quicksort, given relative to Standard MzScheme quicksort. Speedup is given as Sequential Execution Time / Execution Time.

lent distributed quicksort algorithms in C/MPI and Proto-MPSCM, is available graphically in Figures 4.1 and 4.2 or in tabular form in Appendix A, Tables A.1 and A.2. This data provides a baseline for assessing processing speed in C versus MzScheme as well as an idea of communication costs for MzScheme's TCP/IP functions. The overall performance in MPSCM is orders of magnitude slower than C/MPI, which is expected, but speedup, relative to single-processor Scheme and with increasing numbers of computing nodes, is encouraging. Tables 4.1 and 4.2 show Speedup, given as Sequential Execution Time / Execution Time, relative to single processor performance for all distributed data from Tables A.1 and A.2. Speedup for the MPSCM example is not processor proportional, it is never much better than half the number of processors. However, it is important to note that distributed sorting is an inherently high overhead affair, because it requires very large transmissions of data across the network, and that quicksort is a very efficient function relative to the size of the data to which it is applied. Also, the speed-up acquired by adding nodes in MPSCM compares favorably to the C/MPI data. In fact, subproblem distribution in C/MPI actually increases execution time in all examples shown. That MPSCM provides significant speedup for a problem that is communication intensive and for which it is typically quite difficult to improve execution time via subproblem distribution, can be taken as an indication that MzScheme's TCP functions can handle the data distribution aspect of MPSCM.

Table 4.3 provides an alternate presentation of the MPSCM quicksort data from Table A.2 that gives an interesting perspective on MPSCM's performance relative to standard MzScheme. This table gives the elements per second performance for the Standard MzScheme example and the proto-MPSCM distributed examples. This data shows a clear degradation of elements per second performance for Standard MzScheme quicksort on vectors of increasing size. This is a problem with many typical MzScheme solutions because of the use of deep recursion and specialized data types. Distributed solutions limit this degradation by decreasing the maximum depth of recursion, and the data indicates that the proto-MPSCM examples provide much more constant elements per second

performance.

Elements	Nodes				
	1	2	4	8	16
1024	111558.99	57761.67	120854.48	151906.25	159700.56
4096	97913.13	79685.62	102297.70	156323.94	76051.84
16384	81747.51	82477.96	117578.67	143793.72	180466.37
65536	58781.95	78205.06	118903.42	157399.98	157873.18
262144	60301.42	78947.08	116812.83	74777.74	166576.96
1048576	53574.46	74075.68	108839.98	137740.70	161997.23
4194304	49661.36	68323.77	103586.09	128315.66	150482.65

Table 4.3: Elements per second performance for Proto-MPSCM quicksort. Data is taken from timing results in Table A.2.

4.2 Numerical Integration

Performance testing for the later versions of MzScheme focuses on speed-up relative to equivalent Standard MzScheme functions rather than comparisons to C/MPI. Data collected from execution of the numerical integration examples discussed earlier, applied to the function $f(x) = \frac{4}{1+x^2}$, will serve as a case-study for this analysis. This data is presented graphically in Figure 4.3 and in tabular form in Table A.3.

It became clear very early in the development and research of MPSCM that MzScheme provides some idiosyncrasies in performance that make analysis more complicated. For example, using the same *Standard* Scheme `trap` function used as a helper function in the parallel examples as a benchmark, the timing results in Table 4.3 do not, as expected, scale linearly with the number of subintervals used in the approximation. In fact, execution time increases much more quickly. This is, essentially, the nature of a declarative language, such as Scheme, where a function describes what is to be done, but not the way it will be done [1].

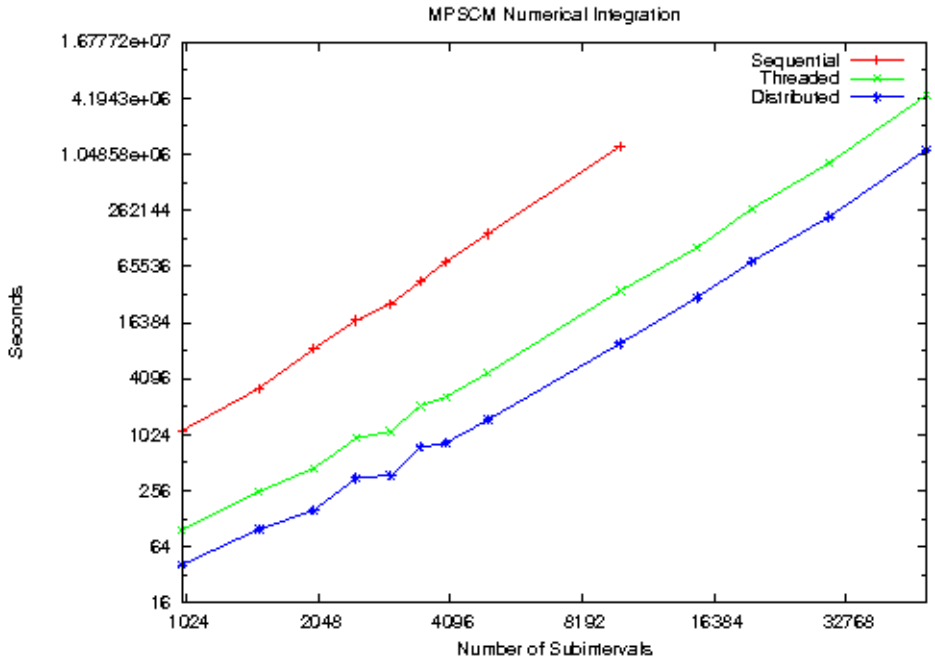

Figure 4.3: Plot of MPSCM numerical integration timing data for the function $f(x) = \frac{4}{1+x^2}$. Threaded and Distributed data is from the examples in Figures 3.2 and 3.1, and the Sequential data uses Standard MzScheme. The Distributed data is collected using four dual-core machines that are each identical to the ones used for the Threaded and Sequential data. Both parallel versions use eight threads. Tabular data is available in Table A.3.

Subintervals	Speedup	
	Threaded	Distributed
1000	11.7500	27.5122
1500	12.8560	32.4646
1000	19.1076	54.2803
2500	18.4979	49.2926
3000	23.4018	69.4545
3500	21.5244	59.6061
4000	28.0917	87.4566
4500	15.4270	65.9267
5000	30.9144	98.8301
6000	31.8666	104.3957

Table 4.4: Speedup relative to the Standard Scheme trapezoidal rule function for the Threaded and Distributed MPSCM examples (Figures 3.2 and 3.1). Speedup is given as Sequential Execution Time / Execution Time. Data is taken from the timing results in Table A.3.

Subintervals	Speedup
1000	2.341463
1500	2.525253
1000	2.840764
2500	2.664773
3000	2.967914
3500	2.769231
4000	3.113253
4500	4.273460
5000	3.196898
6000	3.276024
7000	3.361885
8000	2.734606
9000	3.748856
10000	3.689307
11000	3.659921
15000	3.419386
20000	3.683941
30000	3.780513

Table 4.5: Speedup relative to the MPSCM Threaded trapezoidal rule function (Figure 3.2) for the Distributed MPSCM example (Figure 3.1). Speedup is given as Threaded Execution Time / Execution Time. Data is taken from the timing results in Table A.3.

The increase in execution time comes from MzScheme's use of exact numbers throughout the calculation, rather than inexact floating points. In any language using floats, the computation would lose accuracy, rather than performance, as the number of subintervals increased. Clearly, some changes could be made to the description of the Standard Scheme numerical integration function that would improve performance, including telling MzScheme to use inexact numbers, but the essential problem remains. In MPSCM, decomposition can change the computation being performed in ways that are not apparent in the function description. This point is driven home by the timings for the *Threaded* version of the numerical integration function, which were taken using eight threads on a dual-core machine. Table 4.4 shows speedup for the *Threaded* and *Distributed* functions relative to the *Sequential* using the data from Table A.3. Even for the smallest example, the threaded function displays, not only better than processor proportional speedup, but better than thread proportional speedup!

This property of MzScheme generally serves to make MPSCM parallelization *more* effective at improving performance, because it limits the size of the data used in intermediate function evaluations (by using smaller numbers, for example). However, as much as these quirks may help MPSCM, they should be kept out of performance measures as much as possible. MPSCM is not intended to take advantage of quirks in the sequential execution of MzScheme functions that are hidden deep in the background, but to provide a direct, functional-style approach to exploiting the natural decomposability of Scheme programs for distributed computation. MzScheme sequential arithmetical performance will not be addressed in any more depth and there may be all manner of similar and related issues that make this problem, and other questions of MzScheme performance, even more complicated.

While there are some complications in evaluating the performance of MPSCM, the overall goal is clear. The performance analysis of MPSCM should show that it is distributing work and aggregating data, through the use of the `MPSCM-eval` function, without producing unreasonable overhead. Also, we need to make sure that MPSCM provides the tools, in the form of the higher-level

parallelizing constructs, to easily make programs that provide a worthwhile performance boost over there non-distributed counterparts. These issues can be easily addressed, even if it is difficult to completely explain execution time relative to single-processor equivalents.

The question of the performance of the distributing mechanisms can be partially answered by comparing the *Threaded* timing data to the *Distributed* data. This comparison is given in Table 4.5. The *Distributed* data is from evaluations of the `par-trap` function (Figure 3.1) using eight threads on four dual-core machines, each equivalent to the machines used in the *Threaded* and *Sequential* examples. The *Threaded* data is taken from evaluations of the `thread-trap` function (Figure 3.2), also with eight threads. Because the number of subproblems is the same in the *Threaded* and *Distributed* examples, and because the problem decomposition is handled in the same manner, equivalent problems are being evaluated in both instances. So, comparing performance for these two examples should give us a good idea of how well the distribution scheme is working. Comparison of the timing data for the *Distributed* and *Threaded* functions suggests that the distributed approach provides nearly processor proportional speedup, in this case that would be $\times 4$, but does introduce non-negligible overhead. Also, there appear to be fluctuations in the speedup with the number of steps, without converging to a particular value. This can be expected, to some degree, on a non-dedicated local network where there may be some fluctuations in traffic, but repeated executions have produced similar results. Analyzing the distributed performance involves some of the same types of problems as analyzing Standard Scheme performance. Just as arithmetical functions in Scheme are not always performing the same computation in the same number of steps as the values change, transmission of MzScheme number data-types is not as straightforward as C doubles, for example. It is expected that the communication overhead from the TCP reads and writes is not constant in the MPSCM distributed implementation of numerical integration to the same degree that it is in MPI.

The timing results for numerical integration with distributed MPSCM leave

some concerns, as far as overhead, but are ultimately encouraging. Speed-up, especially over the sequential version, is enough to suggest that MPSCM can provide significant practical use in similar applications.

4.3 Map and Sort

Sadly, while there are a lot of good things to say about the `map-and-sort` function, it is not at all effective for improving performance. This approach to distributed sorting involves a lot of overhead, both with the list manipulation to set up the problem decomposition and the very large communications required to send the sublists over the network. `MPSCM-map` is most effectively used when the mapped function is complicated and the size of the list elements is small. Testing for `map-and-sort` showed it to be almost exactly equivalent in performance to the `sort` function it is built on for all reasonable choices of granularity.

4.4 Fibonacci

When compared to the standard Scheme function described in the *Examples* section, the MPSCM approach to parallelizing the Fibonacci function (Figure 3.5) not only does not give near processor proportional speed-up, but is drastically slower! The reason for this is perhaps more essential to the performance problems with MPSCM than communication and arithmetic concerns: evaluating Scheme expressions with `eval` is much slower than applying the functions directly. Using `eval` prevents Scheme's interpreter from optimizing the code and it must, instead, be evaluated as it is come upon. Because the `par-fib` function has very little actual computation other than the recursive calls, this overhead overwhelms the speed-up provided by distributing the work.

Because of this, the `MPSCM-branch` construct is basically useless, from a practical perspective, as a method of increasing performance for branching recursive algorithms unless they contain a great deal of other computation and do not go into deep recursion. It is possible to write specialized functions that distribute

work in the same way, but don't use `eval` when computing the recursive calls locally, once the distributing condition is no longer true. However, this cannot be readily adapted into a usable construct in the `MPSCM-eval` framework. This is a fundamental problem in the MPSCM approach, and while it certainly does not render it useless for improving performance, it adds significant overhead to a wide range of problems.

4.5 Response to Performance

While MPSCM does not compare favorably to high-performance computing environments and its higher-level functions introduce significant performance concerns, it still has practical use. MPSCM is designed for a set of applications and a class of programmers that is completely separate from typical applications of MPI. Because of this, the performance of MPSCM must be addressed on different terms.

Where MPSCM might pick up some performance advantage is in ease of programming. While this could be argued about, there are certainly those who find Scheme an easier programming environment than C, or anything else for that matter. Also, MPSCM lends itself to certain types of problems in a way that MPI does not. An MPSCM-like approach is particularly useful for small, moderately computation intensive problems in the context of a large set of computations. The Scheme REPL environment can be thought of as a sketch-pad or blackboard in a way that a standard C-style programming environment cannot. For example, it is quite practical to use a Scheme console in place of a calculator even for extremely simple arithmetic. This is less true of the typical C environment. MPSCM maintains much of this advantage, especially for problems that fit its constructs. Also, using `MPSCM-eval` in the REPL can be useful as a stand-alone construct, rather than a building-block for concurrent programming. Using the `MPSCM-eval` function, computation intensive calculations can be computed as asides while the programmer deals with other computations. This ability to call on distributed resources with trivially simple function calls

is absent in MPI. While it is certainly lacking in overall performance, in certain situations MPSCM, or a similar environment, could be a very useful tool for reducing execution time and overall time spent by the programmer.

Chapter 5

Further Discussion and Conclusions

5.1 Parallel O/S

MzScheme is interesting in contrast to imperative languages like C not only in the way commands are structured and expressed, but also in the actual environment in which the programmer is working. The REPL interpreter, which is the standard interface for Scheme and many other functional programming languages, is essentially different from the compiler/executable approach used in C/MPI. In MPSCM, the MzScheme environment takes the place of the BASH shell and process management and program execution are both handled by MzScheme within its scope. From this, along with the use of function-level problem distribution, a sort of parallel operating system naturally emerges, where the MPSCM infrastructure acts as an intermediary for distributed computing resources.

This concept very much resonates with the ideas of some of the creators of PLT Scheme as expressed in "Programming Languages as Operating Systems" [7], which describes the role of MrEd in managing the DrScheme GUI

environment. In the designers' description, the high-level Scheme language acts as the platform for the GUI environment, what they describe as a "meta-circular implementation that is virtually synonymous with LISP." An ideal approach to distributed programming in Scheme parallels, in many ways, DrScheme's approach to GUIs. In the distributed environment, using an `MPSCM-eval` base for parallelization, the local process interacts with the remote processes much like a surrogate user interacting with another read-eval-print loop. In MPSCM, the remote processes act very much this way, where "read" occurs over a TCP connection, "eval" is `eval` and "print" is redirected as a return value. However, there is a great deal of independence between the processes, which is undesirable for capturing this effect entirely. The remote nodes are tied together by the pseudo-REPL but are not literally tied together in the same way that MrEd ties together nested windows, for example, such that real control and interdependence is exerted. This is, in a sense, because it's not possible. It is unreasonable to expect that they can be tied together in the "same way." However, to hide the distributed aspects as much as possible, and provide a comfortable and useful environment, these ties should be as strong as possible. This is achievable, though perhaps difficult, in an MPSCM-like environment that is built on MzScheme.

5.2 Evaluation of MzScheme

This project was intended as an experiment with MzScheme as much as a practical software creation. The original question was not "how can we make the best functional or Scheme programming environment?" but "how can we use these tools in MzScheme to do so?" Overall this was a success, MzScheme provides the communication and synchronization functions necessary to easily and efficiently design necessary base functions for distributed computing. It also provides syntax definition facilities, which were used in creating the `MPSCM-let` construct. Above all, MzScheme simply has a lot of functionality: there may yet be some MzScheme stones unturned that could lead to the development

of even better distributed Scheme environments. Clearly, MzScheme is not a high-performance computing environment, and MPSCM does not change that. However, creating an environment that would compete with MPI in that respect was never a realistic notion from a Scheme base. From a performance perspective, MzScheme reached the goal of enabling near proportional speedup, relative to single processor execution, for functions with a sufficient level of parallelism.

Some improvements to MzScheme would have been of help to MPSCM. The ability to serialize higher order objects to communicate over TCP would allow more flexibility for distributing functional expressions and allow for more natural syntax. Also, better performance of the `eval` function would make branching recursion with `MPSCM-eval` and `MPSCM-branch` a viable option for improving execution time.

5.3 Evaluation of Design

5.3.1 Bad News First

The difficulties with MzScheme were largely a result of the more general difficulty of building a distributed environment of MPSCM's type on top of an environment that was not designed to support it. It is important to note that MzScheme easily and efficiently implements standard message passing and threaded execution functions. An environment more similar to Concurrent ML would provide less trouble than an environment like MPSCM. Since ML and Scheme are similar languages, Concurrent ML is probably designed the way it is for this reason. Both languages are well suited to an imperative style of parallelization. Trying to implement a lazy evaluation approach to distributed programming in MzScheme is essentially problematic because it is a strict evaluating language. Also, the problem of implementing implicit distributing functions in a higher-level library poses some difficulties of syntax, particularly when compounded with the lazy evaluation problem.

The result of these problems can be seen in the implementation of `MPSCM-eval`.

This function has a lot to recommend it as a fundamental parallelizing function both in its application and meaningful representation of functional distributed programming. However, this approach also has some problems, in addition to the performance issues outlined earlier. The chief issue is that it is a pain to work with. Syntax is an essential problem in creating parallelizing and, especially, distributing functions for an environment not specifically designed for it. This is true, in particular, for creating higher-level functions. All widely-used high-level programming languages provide facilities for the transfer and communication of data in some fashion or another, making message-passing a simple and natural extension. Syntax for this type of operation can be easily designed to be what the programmer naturally expects because it is, after all, explicitly the transmission of data. Where MPSCM has problems is that the expectations for a function like `MPSCM-eval` are somewhere in between the expectations for a message passing function and for a Standard Scheme function. The quoted expression passed to `MPSCM-eval` is meant to be thought of like a function call, but it is also a serializable data object and is supposed to be lazily evaluated in a strict evaluation language. Both of these characteristics require it to be conceptually separated from a standard function call. Using `eval` as the basis for the building-block parallelizing function is handy in that it provides a familiar point of reference and mirrors what is actually going on: a command is being sent to a remote REPL, much like the user typing a command. Also, `eval` with quoted expressions allows for serialization and lazy evaluation. However, this approach is clunky and inconvenient, especially when trying to build and use higher-level functions. The `eval` function is a useful point of reference for distributed programming, but building programs entirely out of `eval`s is inefficient from an execution and from a programming standpoint. Requiring every function call to be quoted gives MPSCM more of a learning curve than it should really have.

Additionally, the lazy evaluation implementation is overly distracting in its requirement to keep track of the returned `channel` and finalize it with `channel-get`. For a construct like `MPSCM-eval` to work, particularly in the

simple, educational, applications that it is generally best suited for, a system of lazy evaluation more similar to the one in Haskell [12], where lazy evaluation occurs implicitly, would be preferable. A compromise approach could also be taken, where lazy evaluation must be explicitly asked for but finalization happens implicitly, as required. This is not a disaster, because it fits into and respects the MzScheme environment and provides the framework of a very useful model for distributed computing, but the obstacles introduced here significantly reduce the declarativeness and practical usability of MPSCM.

The strict evaluation functions: `MPSCM-strict-map`, `MPSCM-strict-let`, and `MPSCM-branch`, are more attractive with respect to usability because they can replace their Standard Scheme equivalents with no change in the outcome of the function and do not require finalization. However, these functions often require a change in program structure to be really effective, and they apply directly to a relatively small portion of Scheme problems. The `map` and `let` functions are not general enough to provide a complete approach to distributed programming, at least not one that is substantially similar to Scheme in approach to problem solving. However, building strict-evaluation functions out of `MPSCM-eval` functions finalized in an aggregating manner provides a promising approach to programming in the MPSCM environment. Such functions would not necessarily be as general as `MPSCM-map` and `MPSCM-let`, but could provide a basis for increasing performance in a large number of specialized applications where user functions could be programmed in a declarative style, without need for finalization. In fact, some of the sample programs in this paper, numerical integration and sorting, are good examples of this.

5.3.2 The Upside

As this paper has advocated the MPSCM approach to distributed programming throughout, this section will be kept brief. While MPSCM has some problems with both syntax and performance, overall it provides a nice environment for distributed programming with many interesting and useful characteristics. The

focus on simple and usable programming constructs has produced good results that give a good starting point for further investigation. Though the complications of syntax hinder the educational usefulness of MPSCM somewhat, the focus on easily understood and well-structured parallelizing functions based on common Scheme programming constructs provides a good framework for the use of a Scheme-based environment as a simple and instructive tool for distributed programming. MPSCM demonstrates problem decomposition in a structured and useful manner for a meaningfully general set of problems. Also, it balances an understanding of transmission of data in a distributed environment and lazy evaluation as a deliberate decision in parallel applications, with the encouragement of good functional programming practice. This makes MPSCM useful as a conceptual point of reference for all types of distributed programming environments, including imperative approaches like MPI/C, because the decomposition process is well presented in the approach.

5.4 Suggestions for Further Research

5.4.1 Error Reporting

Error reporting on remote processes is a key shortcoming of the current MPSCM implementation and simple additions in this area would make MPSCM much more viable as a practical environment. This includes error checking in calls to parallel functions, including checks that messages are serializable and checks to `MPSCM-let` to verify that it is not being treated as a `let*`.

5.4.2 MPSCM for Heterogeneous Networks

MPSCM could be fairly easily modified to be useful in a network of nodes with heterogeneous Operating Systems. The MPI-style approach of remotely starting MzScheme on remote nodes could probably not be replicated for Windows, but initialization could be handled with a separate application. Also, this could include the ability to dynamically add nodes to the network. MzScheme and

the majority of the MPSCM implementation allows for portability and it would be useful to take advantage of this. A system that allows for flexibility in the make-up and initialization of the distributed environment is appropriate for MPSCM, which is meant to take advantage of available computing power in an informal manner and not necessarily to operate on a specialized network.

5.4.3 Distributed Computing with Compiled Scheme Code

All performance testing for MPSCM was done using the MzScheme REPL interpreter as the point of execution. It would be interesting to conduct similar experiments with compiled MzScheme code, which can be created using `mzc` [6]. The use of compiled code, either in the MPSCM libraries or in user functions, could provide an increase in performance, perhaps even without significant cost in programmer time.

5.4.4 Syntax

Investigation of the syntax of the parallelizing functions in MPSCM would be helpful to make it (or a related environment) more usable. This could be worked on either from further investigation of MzScheme syntax definitions or more work at lower levels to make a more specialized parallel environment based on MzScheme or another MzScheme dialect.

5.4.5 MzLib

Making MPSCM into a proper MzLib library would make it fit better in MzScheme's system of additions [6].

5.4.6 Imperative Style Parallelizing Constructs

It would be interesting to investigate distributed programming in MzScheme with an approach that is more similar to Concurrent ML [14], perhaps with a base in MPSCM. MzScheme lends itself to an approach that uses more impera-

tive style parallelization constructs, such as threads and messages, and has the functionality to easily implement this type of construct.

5.4.7 Distributed Scheme and Education

Further investigation into the use of Scheme dialects with libraries for distributed programming as educational tools for distributed programming could provide very useful results.

5.4.8 Performance Comparisons

It would be helpful to provide a more extensive set of performance comparisons to other distributed programming environments, particularly other functional-based programming environments.

5.5 Conclusion

MPSCM provides a rough implementation of distributed programming libraries for Scheme. It demonstrates the feasibility of developing this type of environment as a library extension to MzScheme and presents an approach that, while not without problems, provides significant promise as a distributed programming tool, especially for an educational environment. MPSCM also takes advantage of the characteristics of Scheme that make it appropriate for concurrent programming and useful as a language in general: its natural decomposability and its ease of use. The distributed computing constructs in MPSCM provide the basis for an interesting and useful approach to problem decomposition and concurrent evaluation for a general set of problems. MPSCM should help to encourage further investigation of the use of Scheme in this context.

Bibliography

[1] P. S. Barth, R. S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-Strict, Functional Language with State. In *FPCA'91 — Conference on Functional Programming Languages and Computer Architectures*, volume 523, pages 538–568, Harvard, MA, 1991. Springer-Verlag.

[2] Christian Benvenuti. *Understanding Linux Network Internals*. O'Reilly Media, Inc., 2005.

[3] Harvey Brian. Symbolic programming vs. the a.p. curriculum. *The Computing Teacher*, 18, February 1991.

[4] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[5] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems*, 17(5):704–739, September 1995.

[6] Matthew Flatt. PLT MzScheme: Language manual. Technical Report PLT-TR2006-1-v360, PLT Scheme Inc., 2006.

[7] Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming languages as operating systems (or revenge of the son of the lisp machine). In *International Conference on Functional Programming*, pages 138–147, 1999.

[8] Guillaume Germain. Concurrency oriented programming in termite scheme. In *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 20–20, New York, NY, USA, 2006. ACM Press.

[9] Jr. Guy L. Steele and W. Daniel Hillis. Connection machine lisp: fine-grained parallel symbolic processing. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 279–297, New York, NY, USA, 1986. ACM Press.

[10] Julie Sussman Harold Abelson, Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1984.

[11] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989.

[12] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to Haskell 98, 1999.

[13] Luc Moreau. A parallel functional language with first-class continuations (programming style and semantics). *Computers and Artificial Intelligence*, (14(2)):173–205, 1995.

[14] John H. Reppy. Concurrent ML: Design, application and semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 165–198, 1993.

[15] Jr. Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[16] Marc Snir and Steve Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[17] George Springer and Daniel P. Friedman. *Scheme and the Art of Programming*. McGraw-Hill, Inc., New York, NY, USA, 1990.

[18] Berna L. Massingill Timothy G. Mattson, Beverly A. Sanders. *Patterns for Parallel Programming*. Addison-Wesley, 2005.

Appendix A

Additional Data Tables

Elements	Time (s)				
	1	2	4	8	16
1024	0.000204	0.001145	0.001043	0.002782	0.004007
4096	0.000785	0.002664	0.002494	0.003686	0.005444
16384	0.003227	0.008073	0.009316	0.009500	0.008588
65536	0.017259	0.030543	0.036278	0.038091	0.274978
262144	0.066520	0.119782	0.142485	0.162853	0.329270
1048576	0.276221	0.497360	0.592908	0.651315	0.687051
4194304	1.199954	2.007406	2.444802	2.623541	2.739232

Table A.1: Quicksort timing data for C and C/MPI with variously sized arrays and networks.

Elements	Time (s)				
	1	2	4	8	16
1024	0.009179	0.01772802	0.008473	0.006741	0.006412
4096	0.041833	0.051402	0.040040	0.026202	0.053858
16384	0.200422	0.198647	0.139345	0.113941	0.090787
65536	1.114900	0.838002	0.551170	0.416366	0.415118
262144	4.347228	3.320503	2.244137	3.5056420	1.573711
1048576	19.572310	14.155469	9.634107	7.612681	6.472802
4194304	84.4580910	61.388649	40.490998	32.687391	27.872342

Table A.2: Quicksort timing data for MzScheme and Proto-MPSCM with variously sized vectors and networks.

Subintervals	Time (ms)		
	Standard	Threaded	Distributed
1000	1128	96	41
1500	3214	250	99
2000	8522	446	157
2500	17351	938	352
3000	25976	1110	374
3500	44943	2088	754
4000	72589	2584	830
5000	146565	4741	1483
10000	1277777	35849	9717
15000		103645	30311
20000		273691	74293
30000		840030	222200
50000		4491266	1175034

Table A.3: Trapezoidal rule timing data for MzScheme and the MPSCM Threaded and Distributed examples (Figures 3.2 and 3.1) applied to the function $f(x) = \frac{4}{1+x^2}$, with various numbers of subintervals. The Distributed data is collected using four dual-core machines that are each identical to the ones used for the Threaded and Sequential data. Both parallel versions use eight threads.

Appendix B

Quick Start Guide

This guide assumes that you are on a network that could run MPI (SSH permission, common directory structure, etc.).

1. Obtain the following files:
   - mp4.scm
   - runscmmp
   - runhelper
2. Move all files to the same directory. Open "mp4.scm" for editing and change the `define` statement for `MPSCM-home-dir` to the fully qualified path of the directory you put the files in.
3. In that directory, create a text file containing a list of available nodes. Make sure the computer you're currently at (or will be programming from) is listed first.
4. Start MzScheme with the command `mzscheme`.
5. Load the libraries:
   ```
   (load "mp4.scm")
   ```
6. Boot the MPSCM environment:

```
(MPSCM-boot "<name of your list file>"
            <available port #>
            <# of nodes>)
```

7. Available commands:

   - `MPSCM-eval`, returns a MzScheme `channel` datatype, which can be finalized with `channel-get`:

     ```
     (define chan (MPSCM-eval 'expression
                              [optional node argument]))
     (channel-get chan)
     ```

   - `MPSCM-ceval`:

     ```
     (MPSCM-ceval 'f args [optional node argument])
     ```

   - `MPSCM-eval-all` evaluates an expression on all nodes, return values are lost:

     ```
     (MPSCM-eval-all 'expression)
     ```

   - `MPSCM-gload` loads a function definition file on all nodes. Path must be given relative to `MPSCM-home-dir`:

     ```
     (MPSCM-gload "file.scm")
     ```

   - `MPSCM-map`, returns the mapped list directly.

     ```
     (MPSCM-map 'function list nparts)
     ```

   - `MPSCM-lazy-map`, returns a list of channels that can be finalized with `MPSCM-map-finalize`:

     ```
     (define val (MPSCM-lazy-map 'function list nparts))

     (MPSCM-lazy-map-finalize val)
     ```

- `MPSCM-let` binds the return values of the quoted expressions:

```
(MPSCM-let ((x1 '(f1 args1)) ... (xn '(fn argsn)))
   (some-function x1 ... xn))
```

- `MPSCM-lazy-let` binds channels to the returned values:

```
(MPSCM-lazy-let ((x1 '(f1 args1))
                 ...
                (xn '(fn argsn)))
   (some-function (channel-get x1)
                  ...
                  (channel-get xn)))
```

- `MPSCM-branch` returns combining-operator applied to the evaluated expressions. Evaluates remotely if condition is true:

```
(MPSCM-branch condition
              combining-operator
              'exp1
              ...
              'expn)
```

8. The distributed environment can be halted with:

```
(MPSCM-halt)
```

Appendix C

Library Code: mp4.scm

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                  MPSCM State Variables
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Configuration
(define MPSCM-home-dir "")
(define MPSCM-lib-file "mp4.scm")

;; Variables initialized by MPSCM-init and boot scripts
(define MPSCM-listener '())
(define MPSCM-io-list '())
(define MPSCM-procls '())
(define MPSCM-myname "")
(define MPSCM-myrank -1)
(define MPSCM-nprocs 0)

;; Automatic tag and next proc variables for MPSCM-eval
(define MPSCM-next-proc 1)
(define MPSCM-curr-eval-tag -1)

;; Semaphores and Maps for the Receive buffers
(define MPSCM-curr-eval-tag-semaphore (make-semaphore 1))
(define MPSCM-recv-map (make-hash-table 'equal))
(define MPSCM-recv-map-semaphore (make-semaphore 1))
(define MPSCM-end-semaphore (make-semaphore))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;                     MPSCM-eval functions
```

```
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          MPSCM-eval
;;
;; A method of lazy-evaluation using 'eval', MzScheme's 'channel'
;; construct, and paired 'MPSCM-send' and 'MPSCM-recv.'
;;
;; parameters:    cmd - a quoted command, as would be passed to
;;                           eval.  All necessary functions must
;;                           be loaded on remote node (see
;;                           MPSCM-gload)
;;                proc (optional) - the node to evaluate the
;;                           cmd at.  If none is provided, one
;;                           is chosen using MPSCM-get-next-proc!
;; return:        a channel that will eventually have the result
;;                     of the evaled cmd
;; side-effects:  Maintenance to tag sequence, and next-proc
;;                     rotation.  A thread waiting for the return
;;                     value.  The idea is to have the illusion of
;;                     no side effects, for the user, but many are
;;                     involved
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define MPSCM-eval
   (lambda (cmd . proc)
      (let* (
         (tag (MPSCM-get-next-eval-tag!))
         ;; create the channel that will be used
         (ch (make-channel))
         (eval-proc (if (null? proc) (MPSCM-get-next-proc!) (car proc)))
         ;; if the command is a procedure, quote it
         (qcmd
            (if (procedure? (car cmd))
               (begin
                  (cons (quote (car cmd)) (cdr cmd)))
               (begin
                  cmd)))
         ;; append "SEND BACK TO ME" to the command string
         (ncmd (append (list 'MPSCM-send MPSCM-myrank)
                       (list qcmd)
                       (list tag))))
         ;; function creates a thread that sends out the command
         ;; and the waits to receive the response and put it in the
         ;; channel.
         (letrec ((makethread (lambda ()
            (if (= eval-proc 0)
               ;; If the proc is 0, evaluate here
               (thread (lambda()
                  ;;channel-put will cause a pending or future call to
                  ;;channel-get to return the value that is put there.
                  ;;see MzScheme channel documentation
                  (channel-put ch (eval cmd))))
               (thread (lambda ()
```

```
                (begin
                   (MPSCM-send eval-proc ncmd "MPSCM-CMD")
                   (channel-put ch (MPSCM-recv eval-proc tag)))))))))
            (begin
               ;; make the thread
               (makethread)
               ;; return the channel
               ch)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                         MPSCM-ceval
;;
;; Composes a quoted function with a list of arguments (cons) and
;; sends it to MPSCM-eval.
;;
;; parameters:    cmd - a quoted function-name
;;                lst-args - a list of arguments to be use in the
;;                              function.  All elements must be
;;                              synchronizable (functions must be
;;                              quoted)
;;                proc (optional) - as in MPSCM-eval
;; return:        A channel, as in MPSCM-eval
;; side-effects:  See MPSCM-eval
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-ceval
   (lambda (cmd lst-args . proc)
      (if (null? proc)
         (MPSCM-eval (cons cmd lst-args) (MPSCM-get-next-proc!))
         (MPSCM-eval (cons cmd lst-args) (car proc)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;                      MPSCM-Map functions
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                         MPSCM-map
;;
;; Maps the quoted function f onto the list ls, but does so
;; by spliting the list into MPSCM-nprocs parts, which are evaluated
;; with MPSCM-eval. MPSCM-map is a strict map, so after this
;; occurs, it waits on all the channels from the MPSCM-evals and
;; compiles the mapped list.
;;
;; parameters:    f - a quoted function name
;;                ls - the list to map onto
;;                MPSCM-nprocs - the number of parts to split the map
;;                              into
;; return:        the mapped list
```

```
;; side-effects:  standard side-effects associated with MPSCM-eval,
;;                  should be experienced by the user the same as
;;                  a standard map, aside from performance.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-map
   (lambda (f ls nparts)
      ;;split up the list
      (let ((splitlist (MPSCM-split-list ls nparts)))
         ;;call-remote makes all the MPSCM-eval calls by constructing
         ;;a map statement on each sublist
         (letrec ((call-remotes
            (lambda (sls n)
               (if (< n nparts)
                  (let ((csls (car sls)))
                  (cons
                     (MPSCM-eval
                        (append '(map)
                           (cons f '())
                           (cons (cons 'quote
                              (cons csls '())) '()))
                        (MPSCM-get-next-proc!))
                     (call-remotes (cdr sls) (+ n 1))))
                  '()))))
            ;; calls channel-get on all items is a list
            (channel-get-all
               (lambda (ls)
                  (if (null? ls)
                     '()
                     (cons (channel-get (car ls))
                           (channel-get-all (cdr ls)))))))
            ;; call the remotes, get the channels and reconstitute the list
            (MPSCM-unsplit-list
               (channel-get-all (call-remotes splitlist 0)))))))


;; MPSCM-strict-map is the same as MPSCM-map
(define MPSCM-strict-map MPSCM-map)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                        MPSCM-lazy-map
;;
;; Maps the quoted function f onto the list ls, but does so
;; by spliting the list into MPSCM-nprocs parts, which are evaluated
;; with MPSCM-eval.  Returns a list of channels, rather than the
;; final list, that can be finalized with MPSCM-map-finalize.
;;
;; parameters:    f - a quoted function name
;;                ls - the list to map onto
;;                MPSCM-nprocs - the number of parts to split the map
;;                               into
;; return:        A list of channels waiting that can be finalized
;;                     into the mapped list with MPSCM-map-finalize
;; side-effects:  standard side-effects associated with MPSCM-eval
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define MPSCM-lazy-map
   (lambda (f ls MPSCM-nprocs)
      (let (
         ;; split the list into parts
         (splitlist (MPSCM-split-list ls MPSCM-nprocs)))
         ;; a helper function that calls MPSCM-eval of map
         ;; onto the sublists and builds the list of channels
         (letrec ((call-remotes
            (lambda (sls n)
               (if (< n MPSCM-nprocs)
                  (let ((csls (car sls)))
                  (cons
                     (MPSCM-eval
                        (append '(map)
                           (cons f '())
                           (cons (cons 'quote
                              (cons csls '())) '()))
                        (MPSCM-get-next-proc!))
                     (call-remotes (cdr sls) (+ n 1))))
                  '()))))
            ;;call the helper and return its value
            (call-remotes splitlist 0)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          MPSCM-map-finalize
;;
;; Finalizes the return of lazy map
;;
;; parameters:    ls - list of channels, or the return value of
;;                          MPSCM-lazy-map
;; return:        A list representing the final value from
;;                     MPSCM-lazy-map, as though map were called
;; side-effects:  channel-get on all channels from the lazy-map
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-map-finalize
   (lambda (ls)
      (letrec (
          (channel-get-all
               (lambda (ls)
                  (if (null? ls)
                     '()
                     (cons (channel-get (car ls))
                           (channel-get-all (cdr ls)))))))
         (MPSCM-unsplit-list (channel-get-all ls)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;                          MPSCM-let
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                        MPSCM-lazy-let
;;
;; Adapted from JRM's 'Syntax-rules Primer for the Merely Eccentric'
;; bind-variables function.  Like a standard let, but instead of
;; binding a variable to a value, binds it to the channel returned
;; by MPSCM-eval called on the command given it (lazy-evaluation).
;; The processor to be used by MPSCM-eval is determined by
;; MPSCM-get-next-proc! on a rotating basis.  This is meant to be
;; like a let, not a let*, but will not properly report errors if
;; used improperly.
;;
;; This is a syntax object, not a function.
;;
;; Form:
;;               (MPSCM-lazy-let ((x1 '(f1 args1))
;;                                ...
;;                                (xn '(fn argsn)))
;;                     (some-function (channel-get x1)
;;                                ...
;;                                (channel-get xn)))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-syntax MPSCM-lazy-let
   (syntax-rules ()
     ((MPSCM-lazy-let () form . forms)
      (begin form . forms))

     ((MPSCM-lazy-let
         ((variable value0 value1 . more) . more-bindings) form . forms)
      (syntax-error "MPSCM-let illegal binding"
            (variable value0 value1 . more)))

     ((MPSCM-lazy-let ((variable value) . more-bindings) form . forms)
      (let ((variable (MPSCM-eval value (MPSCM-get-next-proc!))))
            (MPSCM-lazy-let more-bindings form . forms)))

     ((MPSCM-lazy-let bindings form . forms)
      (syntax-error "Bindings must be a list." bindings))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                        MPSCM-strict-let
;;
;; Adapted from MPSCM-lazy-let.  uses lazy-let to bind the channels
;; and then MPSCM-strict-let-helper to channel-get the channels
;; and bind the returns.  As with lazy-let, this is meant to be like
;; a let, not a let*, but will not properly report errors if used
;; improperly.
;;
;; This is a syntax object, not a function.
;;
;; Form:
;;             (MPSCM-strict-let ((x1 '(f1 args1))
```

```
;;                                    ...
;;                                    (xn '(fn argsn)))
;;                    (some-function x1
;;                                   ...
;;                                   xn))
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-syntax MPSCM-strict-let
   (syntax-rules()
      ((MPSCM-strict-let bindings . forms)
         (MPSCM-lazy-let bindings
            (MPSCM-strict-let-helper bindings . forms)))))


(define-syntax MPSCM-let
   (syntax-rules()
      ((MPSCM-let bindings . forms)
         (MPSCM-lazy-let bindings
            (MPSCM-strict-let-helper bindings . forms)))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                      MPSCM-strict-let-helper
;;
;; This is a helper syntax for strict-let, not of any use on its
;; own.  Assumes that MPSCM-lazy-let has already bound the values
;; channels, and rebinds them to the channel-get og those channels.
;; This is admittedly odd, which is a product of the odd-ness of
;; the construct (from Scheme's perspective) and my own unfamiliarity
;; with syntax objects.
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-syntax MPSCM-strict-let-helper
      (syntax-rules ()
     ((MPSCM-strict-let-helper () form . forms)
      (begin form . forms))

     ((MPSCM-strict-let-helper
         ((variable value0 value1 . more) . more-bindings) form . forms)
            (syntax-error "MPSCM-let illegal binding"
            (variable value0 value1 . more)))

     ((MPSCM-strict-let-helper ((variable value) . more-bindings) form . forms)
      (let ((variable (channel-get variable)))
            (MPSCM-strict-let-helper more-bindings form . forms)))

     ((MPSCM-strict-let-helper bindings form . forms)
      (syntax-error "Bindings must be a list." bindings))))




;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
```

```
;;                        MPSCM-branch
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                         MPSCM-branch
;;
;; Takes a condition (\#t or \#f) a combining operator (function
;; that can be applied to a list) and some number of quoted
;; functional expressions (must be callable with MPSCM-eval).
;; The expressions are evaluated locally if the conditional
;; is false and with MPSCM-eval if it is true.  Then the
;; results are combined with the combining operator.
;;
;; parameters:      condition - whether to distribute
;;                  comb - combining operator
;;                  exps - quoted expressions
;; return:         The result of applying comb to the evaled
;;                     expressions.
;; side-effects:   Standard side-effects associated with MPSCM-eval,
;;                     none from the user's perspective.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(define MPSCM-branch
   (lambda (condition comb . exps)
      (letrec (
         ;; calls MPSCM-eval on a list of expressions
         ;; returns a list of channels
         (MPSCM-eval-list
            (lambda (expressions)
               (if (null? expressions)
                  '()
                  (cons (MPSCM-eval (car expressions))
                        (MPSCM-eval-list (cdr expressions))))))
         ;; evals a list of expressions
         ;; returns a list of return values
         (eval-list
             (lambda (expressions)
              (if (null? expressions)
                  '()
                  (cons (eval (car expressions))
                        (eval-list (cdr expressions))))))
         ;; channel-get every item in a list and return a
         ;; list of the return values
         (channel-get-list
            (lambda (list-channels)
               (if (null? list-channels)
                  '()
                  (cons (channel-get (car list-channels))
                        (channel-get-list (cdr list-channels)))))))
      ;; If the condtion is true, evaluate with MPSCM-eval, else
      ;; use regular eval
      (if condition
```

```
         (apply comb (channel-get-list (MPSCM-eval-list exps)))
         (apply comb (eval-list exps))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;                    MPI-style functions
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                         MPSCM-send
;;
;; basic send function, sends a message composed of a tag and the
;; message itself.  out and tag can be anny serializable values.
;;
;; parameters:     procno - id of the process addressed
;;                 out    - the message to be sent
;;                 tag    - an identifying tag for the message
;; return:         '()
;; side-effects:   writes composed message to the output buffer
;;                   associated with the identified process
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-send
   (lambda (procno out tag)
      (let (
            ;;prepare the message
            (outbuff (cons out (cons tag '())))
            ;;get the output port from the io list
            (o (caddr (MPSCM-queryio procno MPSCM-io-list))))
         (begin
            ;;write the message to the output port and flush the port
            (write outbuff o)
            (flush-output o)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                         MPSCM-recv
;;
;; Receieve a message from a given proc number with a given tag.
;; Synchronizes with the recv-loop threads to make sure to get the
;; data if it's there and when it comes, using semaphores and the
;; MPSCM-recv-map.  Format of messages matches MPSCM-send.  Blocks
;; until a message is received.
;;
;; parameters:    procno - id of node to recv from
;;                tag - tag to receive
;; return:        The value from the message received
;; side-effects:  Synchronization actions, removes received item
;;                     from map
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(define MPSCM-recv
   (lambda (procno tag)
      ;; get whatever's in the map
      (begin (semaphore-wait MPSCM-recv-map-semaphore)
      (let ((obj (hash-table-get MPSCM-recv-map
                                 (cons procno tag)
                                 "MPSCM-NOTHING")))
         ;; if it's nothing, hash a semaphor and wait on it
         (if (equal? obj "MPSCM-NOTHING")
            (let ((sema (make-semaphore)))
               (begin
                  (hash-table-put! MPSCM-recv-map  (cons procno tag) sema)
                  (semaphore-post MPSCM-recv-map-semaphore)
                  ;; sync to the semaphore, decrementing when you get it
                  ;; messages can only be received once
                  (sync sema)
                  ;; return whatever's in the map after the sync
                  (semaphore-wait MPSCM-recv-map-semaphore)
                  (let ((val (hash-table-get MPSCM-recv-map
                                             (cons procno tag)
                                             "MPSCM-NOTHING")))
                     (begin
                        ;; put the semaphore back
                        (hash-table-put! MPSCM-recv-map (cons procno tag) sema)
                        (semaphore-post MPSCM-recv-map-semaphore)
                        val))))
            ;; if the object is a semaphor, wait on it, using sync
            (if (semaphore? obj)
               (begin
                  (semaphore-post MPSCM-recv-map-semaphore)
                  ;; sync to the semaphore
                  (sync obj)
                  ;; return whatever's in the map after the sync
                  (semaphore-wait MPSCM-recv-map-semaphore)
                  (let ((val (hash-table-get MPSCM-recv-map
                                             (cons procno tag)
                                             "MPSCM-NOTHING")))
                     (begin
                        ;; put the semaphore back, it will be reused if
                        ;; the same tag is requested again
                        (hash-table-put! MPSCM-recv-map (cons procno tag) sema)
                        (semaphore-post MPSCM-recv-map-semaphore)
                        val
                  )))
               ;; if the message is already there, return it
               (begin
                  (semaphore-post MPSCM-recv-map-semaphore)
                  obj)))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;                       Boot and Halt
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                        MPSCM-boot
;;
;; Boot mzscheme on the first MPSCM-nprocs processors in the specified hostfile.
;; Must be called from the first computer on the list (this will be the
;; master).  Uses shell-ex to run the boot scripts which will set up
;; the proc lists and run MPSCM-init on each remote node.  Waits on
;; this to complete and returns control.
;;
;; parameters:    hostfile - list of computer identifiers (standard
;;                           SSH rules apply), first must be
;;                           the calling node
;;                port - the port number to be used for MPSCM
;;                           communication
;;                MPSCM-nprocs - the number of procs to use from the list
;;                           of hosts, must be <= the length of
;;                           said list
;; return:        none
;; side-effects:  Lots!  Initializes ports and listeners on all nodes
;;                  Sets the value of MPSCM's host list.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-boot
   (lambda (hostfile port MPSCM-nprocs)
      (begin
         (set! MPSCM-procls (MPSCM-create-procls hostfile MPSCM-nprocs))
         ;;spawn one thread to call init on this process
         (let ((a (thread
               (lambda () (MPSCM-init (car MPSCM-procls) port))))
            ;; and another to run the script
            (b (thread
               (lambda ()
                  ;; Use shell-ex
                  (MPSCM-shell-ex
                     (string-append MPSCM-home-dir "runscmmp" )
                     (string-append MPSCM-home-dir hostfile )
                     (number->string port)
                     (number->string MPSCM-nprocs)
                     MPSCM-home-dir MPSCM-lib-file)))))
            (thread-wait a)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                        MPSCM-halt
;;
;; evaluate MPSCM-fin at every process.  Ends the MPSCM session.
;;
;; parameters:    none
;; return:        none
;; side-effects:  Closes all the ports, etc.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-halt
   (lambda ()
```

```
      (MPSCM-eval-all '(MPSCM-fin))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;                  The Distributed Environment
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          MPSCM-gload
;;
;; The global load function.  Loads a file (standard Scheme load
;; method) on all nodes.  The file path should be given relative
;; to the MPSCM-home-dir.  Uses MPSCM-eval-all
;;
;; parameters:    filename
;; return:        none
;; side-effects:  Loads the file on all nodes
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-gload
   (lambda (filename)
      (MPSCM-eval-all (list 'load (format "~a~a" MPSCM-home-dir filename)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          MPSCM-eval-all
;;
;; parameters:    a quoted command to be evaluated
;; return:        '()
;; side-effects:  Uses MPSCM-eval to eval cmd at every node.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-eval-all
   (lambda (cmd)
      (letrec ((helper
         (lambda (n)
            (if (<= n 0)
               (eval cmd)
               (begin
                  (MPSCM-send n cmd "MPSCM-CMD")
                  (helper (- n 1)))))))
         (helper (- MPSCM-nprocs 1)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          MPSCM-shell-ex
;;
;; Uses MzScheme subprocess function to execute an external program
;; as from a shell.  Waits for the external program to complete.
;; This is used by MPSCM-boot to run the boot scripts.
;;
;; parameters:    args - the arguments for the external program
;; return:        the output of the program
;; side-effects:  Sets the MPSCM-next-proc value for the next time
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-shell-ex
   (lambda args
      ;; build the subprocess call using append, then eval it
      (let-values (((a b c d) (eval (append '(subprocess \#f \#f \#f) args))))
         (begin
            ;; wait for subprocess to complete
            (subprocess-wait a)
            ;; display stdout (max 100000 chars)
            (let ((ret (read-string 100000 b)) (err (read-string 100000 d)))
               (begin
                  ;; display the output and the error from the subprocess
                  (display ret)
                  (display "\\n")
                  (display err)
                  (display "\\n")
                  ;; close ports
                  (close-output-port c)
                  (close-input-port b)
                  (close-input-port d)
                  ret))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                         MPSCM-get-next-proc!
;;
;; Determines the next proc for MPSCM-eval on a rotating basis and
;; sets up for the next call, using MPSCM-next-proc.  This
;; could be replaced with another function to use more advanced load
;; balancing (see MPSCM-get-load for the skeleton of such an approach)
;;
;; parameters:      none
;; return:          The ID of the next proc to be used
;; side-effects:    Sets the MPSCM-next-proc value for the next time
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-get-next-proc!
   (lambda ()
      (let ((ret MPSCM-next-proc))
         (begin
            (set! MPSCM-next-proc
                  (remainder (+ MPSCM-next-proc 1) MPSCM-nprocs))
            ret))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          MPSCM-get-load
;;
;; This is an example of a function that could be used for load
;; balancing.  MPSCM-eval allows the calling node to check the
;; current load on all other nodes as well as, perhaps, other
;; information (number of processors, etc) and use this in the
;; MPSCM-get-next-proc! logic.  The MPSCM project has not involved
;; any investigation of these possibilities with the exception of
;; this function as a demonstration of feasibility.
```

```
;;
;; parameters:      none
;; return:          A number auxed out of ps giving the load for
;;                  whatever machine this function is called on
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; string.ss needed for regex
(require (lib "string.ss"))
;; sums all the percentages for all processes running on the machine, uses ps
(define MPSCM-get-load
  (lambda ()
    (letrec
      ;; function to sum the nth item in a space separated string for
      ;; each string in a list
      ((sum-nth-of-each (lambda (ls n)
         ;; drop the last one, it's " "
         (if (null? (cdr ls))
            0
             ;; split on spaces
            (+ (string->number (list-ref (regexp-split "[ ]+" (car ls)) n))
              (sum-nth-of-each (cdr ls) n))))))
      ;; get the output from ps
      (let* ((ps-output (MPSCM-shell-ex "/bin/ps" "aux"))
             ;; parse on \\n
             (ps-parsed (regexp-split "\\n" ps-output)))
            ;; return the sum
            (sum-nth-of-each (cdr ps-parsed) 2)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                        MPSCM-create-procls
;;
;; Create a list of nProcs computer names from a given file, path
;; should be fully qualified.  Called by MPSCM-boot
;;
;; parameters:      filename - path of the file to be used
;;                  nProcs - the number of procs to use
;; return:          a list of computer names from the file
;; side-effects:    none
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-create-procls
   (lambda (filename nProcs)
      (let ((fis (open-input-file filename)))
         (letrec ((helper (lambda (is n)
            (let ((toke (read is)))
               (if (= n nProcs)
                  (begin
                     (close-input-port is)
                     '())
                  (if (eof-object? toke)
                     (begin
                        (display (string-append
                           "The provided file does not contain "
                           (number->string nProcs)
                           " processes.  Please update the file or "
```

```
                        "use a different number of processors.\\n\\n"))
                     (close-input-port is)
                     '())
                  (begin
                     (let ((str (symbol->string toke)))
                        (cons str (helper is (+ n 1))))))))))
         (helper fis 0)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                      MPSCM-get-next-eval-tag!
;;
;; Get the next available tag for MPSCM-eval pairings.  Set
;; MPSCM-curr-eval-tag to a new value.  All MPSCM-eval tags are
;; negative numbers, which should not be used in MPSCM-send calls.
;;
;; parameters:      none
;; return:          the next eval tag (one less than the previous)
;; side-effects:    Sets the MPSCM-curr-eval-tag value as one less
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-get-next-eval-tag!
   (lambda ()
      (begin
         (semaphore-wait MPSCM-curr-eval-tag-semaphore)
         (let ((ret MPSCM-curr-eval-tag))
            (begin
               (set! MPSCM-curr-eval-tag (- MPSCM-curr-eval-tag 1))
               (semaphore-post MPSCM-curr-eval-tag-semaphore)
               ret)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                        MPSCM-init
;;
;; Initialize the MPSCM communication ports for a given node and
;; starts the recv-loop threads (buffer management), returning
;; control to master node (originating) and leaving control in
;; recv buffer threads for workers
;;
;; parameters:      procname - the name of this node (as it appears
;;                             in the proc list)
;;                  port - the port that MPSCM is using
;; return:          '()
;; side-effects:    initializes the MPSCM-io-list and starts the recv
;;                     buffer threads.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-init
   (lambda (procname port)
      (begin
         ;;set the identifying info for this node
         (set! MPSCM-myname procname)
         (set! MPSCM-myrank (MPSCM-reverse-index procname MPSCM-procls))
         (set! MPSCM-nprocs (length MPSCM-procls))
         ;;set the listener
```

```
         (MPSCM-set-port! port)
         (letrec
            ;;helper functions to build the list of paired input ports
            ;;each node connects to all nodes lower than it and waits
            ;; on all higher nodes (determined by id, or place in list)
            ((connect (lambda (n pls)
               (if (< n MPSCM-myrank)
                  (let ((cls (connect (+ n 1) (cdr pls))))
                     (cons (connect-one (car pls) port) cls))
                  '())))
            (connect-one (lambda (proc port)
               (let-values (((i o) (tcp-connect proc port)))
                  (cons (MPSCM-reverse-index proc MPSCM-procls)
                        (cons i (cons o '()))))))
            (accept (lambda (n pls)
               (if (null? pls)
                  '()
                  (if (<= n MPSCM-myrank)
                     (accept (+ n 1) (cdr pls))
                     (begin
                        (let-values (((i o) (tcp-accept MPSCM-listener)))
                           (cons (cons n (cons i (cons o '())))
                                 (accept (+ n 1) (cdr pls))))))))))
            ;;set the MPSCM-io-list using helper functions
            (set! MPSCM-io-list
               (append
                  (connect 0 MPSCM-procls)
                  (accept 0 MPSCM-procls))))
            ;;check if this is the master/originating node
            (if (not (= MPSCM-myrank 0))
               ;; workers give worker loop the thread of control
               (MPSCM-worker-loop)
               ;;master runs it in the background
               (thread (lambda () (MPSCM-worker-loop))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                        MPSCM-start-recv-buffer
;;
;; Start a thread running MPSCM-recv-loop for each remote node.
;;
;; parameters:      none
;; return:          A list of threads running MPSCM-recv-loop
;; side-effects:    Those of MPSCM-recv-loop for each one
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-start-recv-buffer
   (lambda ()
      (letrec ((start-for-processes (lambda (n)
         (if (= n MPSCM-nprocs)
            '()
            ;; skip if it's you
            (if (= n MPSCM-myrank)
               (start-for-processes (+ n 1))
               ;; call MPSCM-recv-loop in a thread
               (cons (thread (lambda () (MPSCM-recv-loop n)))
```

```
                    (start-for-processes (+ n 1))))))))
         (start-for-processes 0))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                        MPSCM-recv-loop
;; To be run in a thread.  Loops on the input port of one node
;; and adds anything received to the buffer to be accessed by
;; MPSCM-recv.  Runs until killed.
;;
;; parameters:      procno - id of the proc to recv from
;; return:          does not return
;; side-effects:    Reads input ports and puts data in the
;;                     MPSCM-recv-map
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-recv-loop
   (lambda (procno)
      ;; read the in buffer
      (let ((msg (MPSCM-query-in procno)))
            (begin
               ;; cadr off the message tag
               (let ((tag (cadr msg)))
                  ;; check to see if the message is a command
                  (if (equal? tag "MPSCM-CMD")
                     ;; if it is a command, just eval it in a thread
                     (thread (lambda () (eval (car msg))))
                     ;; hash the message on (procno.tag)
                     (begin
                        ;; get the semaphore
                        (semaphore-wait MPSCM-recv-map-semaphore)
                        (let ((current-contents
                                 (hash-table-get MPSCM-recv-map
                                                 (cons procno tag)
                                                 "MPSCM-NOTHING")))
                           (begin
                              ;; put the message in the hash table
                              (hash-table-put! MPSCM-recv-map
                                               (cons procno tag) (car msg))
                              ;; give it back
                              (if (semaphore? current-contents)
                                 (semaphore-post current-contents))
                              ;; semaphore-post
                              (semaphore-post MPSCM-recv-map-semaphore))))))
               ;; loop, until killed
               (MPSCM-recv-loop procno)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                        MPSCM-worker-loop
;;
;; This is the loop to be run by all processes to handle io for
;; MPSCM.  With the master process, this method is started in
;; a background thread, but in the workers, it's the main thread
;; of control.
```

```
;;
;; parameters:      none
;; return:          '()
;; side-effects:    Starts recv buffers an then waits until the
;;                     MPSCM-end-semaphore is posted, at which point
;;                     it kills all the recv-buffer threads and
;;                     returns
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-worker-loop
   (lambda ()
      (letrec ((kill-all-threads
         (lambda (lst-threads)
            (if (null? lst-threads)
               '()
               (begin
                  (kill-thread (car lst-threads))
                  (kill-all-threads (cdr lst-threads)))))))
         (let ((lst-proc-recv-threads (MPSCM-start-recv-buffer)))
            (begin
               (sync (semaphore-peek-evt MPSCM-end-semaphore))
               (kill-all-threads lst-proc-recv-threads))))))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                        MPSCM-query-in
;;
;; Method used in receive threads to check incpming messages
;;
;; parameters:      procno - id of process to check input from
;; return:          value read from the input port associated with
;;                     the given node (from the MPSCM io list)
;; side-effects:    same side-effects as 'read', clears the port
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-query-in
   (lambda (procno)
      (read (cadr (MPSCM-queryio procno MPSCM-io-list)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                        MPSCM-fin
;;
;; Finalizes MPSCM on the current node
;;
;; parameters:      none
;; return:          '()
;; side-effects:    abandons ports, closes listeners, and sets the
;;                     end semaphore to stop the buffer threads
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-fin
   (lambda ()
      (begin
         (semaphore-post MPSCM-end-semaphore)
```

```
         (MPSCM-abandon-all)
         (tcp-close MPSCM-listener))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                      MPSCM-queryio
;;
;; Gets the nth item in an MPSCM-io-list of port pairs
;;
;; parameters:      n - index of port pair to be retrieved
;;                  ls - list of ports
;; return:          port pair (i o)
;; side-effects:    none
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-queryio
   (lambda (n ls)
      (if (null? ls)
         ls
         (if (= (caar ls) n)
            (car ls)
            (MPSCM-queryio n (cdr ls))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                         MPSCM-abandon-all
;;
;; abandon all tcp ports in use by MPSCM using MPSCM-abandon-io
;;
;; parameters:      none
;; return:          '()
;; side-effects:    abandons the io port list for MPSCM
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-abandon-all
   (lambda ()
      (begin
         (MPSCM-abandon-io MPSCM-io-list))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          MPSCM-abandon-io
;;
;; parameters:      iols - list of paired ports ((i o) (i o) ... )
;; return:          '()
;; side-effects:    Abandons all ports in the list.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-abandon-io
   (lambda (iols)
      (if (null? iols)
         '()
         (begin
            (tcp-abandon-port (cadar iols))
            (tcp-abandon-port (caddar iols))
            (MPSCM-abandon-io (cdr iols))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;                          MPSCM-set-port!
;;
;; Sets the listener for MPSCM on the given port, used in MPSCM-init.
;; Uses the MzScheme tcp-listen function.
;;
;; parameters:      port number
;; return:          '()
;; side-effects:    sets the value of MPSCM-listener
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-set-port!
   (lambda (port-number)
      (if (null? MPSCM-listener)
         (set! MPSCM-listener (tcp-listen port-number 4 \#t \#f))
         (begin
            (tcp-close MPSCM-listener)
            (set! MPSCM-listener (tcp-listen port-number))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                              MPSCM-wait
;;
;; Calls thread-wait on every member of a list of threads, returns null
;; when all threads are finished.
;;
;; parameters:      threadls - list of threads
;; return:          '()
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define MPSCM-wait
   (lambda (threadls)
      (if (null? threadls)
         '()
         (begin
            (thread-wait (car threadls))
            (MPSCM-wait (cdr threadls))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          MPSCM-split-list
;;
;; basic split list using take and drop
;; the last sub-list will be the smallest instead of the largest
;; if mod <> 0 because that process is the last to get it's sublist
;; (slower anyway).  used in the MPSCM-map functions
;;
;; parameters:      ls - a list
;;                  n - the number of parts to split it into
;; return:          ls as a list of n lists, containing the elements
;;                     in order
;; side-effects:    none
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define MPSCM-split-list
  (lambda (ls n)
    (let ((real-n (min n (length ls))) (size (floor (/ (length ls) n))))
      (letrec (
          (take
```

```
           (lambda (lst num)
             (if (or (null? lst) (= num 0))
                 '()
                 (cons (car lst) (take (cdr lst) (- num 1))))))
          (drop
           (lambda (lst num)
             (if (or (null? lst) (= num 0))
                 lst
                 (drop (cdr lst) (- num 1)))))
          (split
           (lambda (lst count)
             (if (= count 1)
                 (list lst)
                 (let ((tooken-list (take lst size))
                       (dropped-list (drop lst size)))
                   (append (list tooken-list)
                           (split dropped-list (- count 1))))))))
       (split ls real-n))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                        Utility Functions
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;basic reverse index method
(define MPSCM-reverse-index
   (lambda (val ls)
      (letrec ((helper (lambda (val ls n)
         (if (null? ls)
            -1
            (if (equal? val (car ls))
               n
               (helper val (cdr ls) (+ n 1)))))))
         (helper val ls 0))))

;; rebuild a list that has been split up
(define MPSCM-unsplit-list
   (lambda (ls)
      (if (null? ls)
         '()
         (append (car ls) (MPSCM-unsplit-list (cdr ls))))))

;; basic list flatten to correct after split
(define MPSCM-flatten-list
  (lambda (ls)
    (if (null? ls)
        '()
        (if (pair? (car ls))
            (append (flatten-list (car ls)) (flatten-list (cdr ls)))
            (cons (car ls) (flatten-list (cdr ls)))))))

;; apppends the take to drop
;; created to allow split list in O(n) car/cdr/cons
```

```
(define MPSCM-take/drop
  (lambda (ls n)
    ;;use a for the drop portion
    (let ((a '()))
      (letrec
          ((helper
            (lambda (ls n)
              ;; if everything's tallied off
              (if (or (= n 0) (null? ls))
                  (begin
                    ;; set the drop
                    (set! a ls)
                    ;; return null
                    '())
                  ;; recursive call
                  (cons (car ls) (helper (cdr ls) (- n 1)))))))
        ;; append the return of the helper to its side effect set of a
        (append (list (helper ls n)) (list a))))))
```

Appendix D

Boot Script: runscmmp

```
#!/bin/bash

# Requires that the first entry in the processes file be
# the master process.

# checks if correct number of args passed in

if [ $# -ne 5 ]  #  if [ num_args <is not equal> 4 ]
then
   echo "usage: $0 <file of host machines>"
   echo "<port no> <nprocs> <home directory> <library file name>"
   exit 1
fi

# sets hosts to be the contents of the file that was passed in
hosts=‘cat "$1"‘

# loop iterator
count=0




for h in $hosts
do
   if [ $count -eq $3 ]  # basecase; reached the total number of
   then                  #   processes given
      exit 0
   else
      if [ $count -ne 0 ]  # if not the first entry, send command
      then

         /usr/bin/ssh -n $h $4/runhelper $h $2 $1 $3 $4$5 &
         sleep 1
```

```
          echo "Process initialized on $h"
       fi
    fi
    count=‘expr $count + 1‘   # increment counter
done

exit 0
```

Appendix E

Boot Script: runhelper

```
#! /bin/sh

#|
exec /usr/bin/mzscheme -qr "$0" ${1+"$@"}
|#
(load (vector-ref (current-command-line-arguments) 4))
(define MPSCM-procls
        (MPSCM-create-procls
                (vector-ref (current-command-line-arguments) 2)
                (string->number
                        (vector-ref (current-command-line-arguments) 3))))
(MPSCM-init
        (vector-ref (current-command-line-arguments) 0)
        (string->number (vector-ref (current-command-line-arguments)
        1)))
(exit)
exit 0
```