LISP AND SYMBOLIC COMPUTATION: An International Journal ?: ??–??, 1992

Implementing Concurrent Scheme for the Mayfly Distributed Parallel Processing System*

R. KESSLER (*kessler@cs.utah.edu*)
H. CARR (*carr@cs.utah.edu*)
L. STOLLER (*stoller@cs.utah.edu*)
M. SWANSON (*swanson@cs.utah.edu*)

Center for Software Science
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112 U.S.A

(Received: August, 1991)

Abstract. This paper discusses a parallel Lisp system developed for a distributed memory, parallel processor, the Mayfly. The language has been adapted to the problems of distributed data by providing a tight coupling of control and data, including mechanisms for mutual exclusion and data sharing. The language was primarily designed to execute on the Mayfly, but also runs on networked workstations. Initially, we show the relevant parts of the language as seen by the user. Then we concentrate on the system Lisp level implementation of these constructs with particular attention to *agents*, a mechanism for limiting the cost of remote operations. Briefly mentioned are the low-level kernel hardware and software support of the system Lisp primitives.

1 Introduction

The quest to harness more computing power for the execution of symbolic computations continues. At the same time, advances in circuit design, fabrication, and basic technology appear to be approaching fundamental physical limits. Concurrency continues to *promise* much sought-after increases in computing power. Harnessing concurrency will require advances in both architecture and software. The Mayfly parallel processor [**?**, **?**] reflects the necessary architectural advances. Our research addresses a portion of the "software problem," by proposing language constructs and an efficient implementation to extend the Lisp class of computer languages used for symbolic computing and artificial intelligence applications.

As in much current concurrent/parallel language research, an existing language has been chosen as the starting point for this work. Scheme (as

*Work Supported in part by the Hewlett-Packard Corporation.

described in R^3RS) [?] was selected as the base language for a number of reasons:

- it is a compact language; its relatively small runtime support package and small number of primitives make it amenable to producing an efficient implementation in a short time;
- it is widely known and well understood;
- it does not require implementation of fluid (or dynamic) variables, a feature that can add significantly to the complexity and cost of multi-threading;
- its utility for symbolic and AI applications has already been demonstrated.

Concurrent Scheme (CS) is standard Scheme extended to provide explicit control parallelism. Thus, the intent of CS is to provide a language in which the programmer can implement parallel algorithms; it is not the intent of CS to use program analysis (including compilation techniques) to "discover" the parallelism in existing programs. Also, CS is oriented towards emphasizing control parallelism rather than data parallelism. Work in data parallelism has achieved a good deal of success in exploiting the concurrency possible with large, regular data sets. Control parallelism, and CS in particular, seeks to address the large body of applications that are not dominated by iterative or recursive application of the same procedure, or composition of procedures, to components of large aggregate data structures.

Concurrent Scheme has been designed explicitly for the Mayfly architecture. Computation in CS involves multiple, fully encapsulated objects, which interact via method invocation. Method invocation is implemented via a remote procedure call mechanism that uses the concept of agents to deliver high performance. A representative computation involves multiple simultaneous threads of control acting upon many objects.

1.1 A Model for Distributed Memory

The physical separation of memories in a distributed multicomputer presents several choices in the model of memory presented to the programmer. An obvious choice is that of one unified virtual space versus a collection of disjoint spaces. Lisp (including most parallel implementations to date) implicitly assumes a single address space with unrestrained traffic in pointers. In a uni-processor implementation (or in a shared-memory multi-processor) this model is inexpensive to provide. With physically disjoint memories, however, the execution costs of providing a unified memory space model increase significantly. The costs must include some non-empty subset of the following:

- wasted processor time while relatively high-latency remote memory accesses are satisfied;
- wasted processor time resulting from context switches to suspend and resume threads making high-latency remote memory accesses;
- global garbage collection with its attendant global synchronization;
- bookkeeping overhead to keep track of exported and imported objects;
- overhead resulting from low-level address checking or trap handling on pointer dereferencing to detect remote accesses.

In light of these costs, a model was chosen for Concurrent Scheme that presents the user with multiple, disjoint spaces. Copying of structured data between these spaces occurs as call-by-value arguments for certain classes of procedures and return-by-value results for the same procedures. This choice does eliminate some of the costs listed above and serves to limit the impact of others. On the other hand, we will see that copying has far-reaching and fundamental effects on the behavior of the values of global variables and of procedure call arguments and values.

Finally, Concurrent Scheme is intended to operate in a distributed memory multi-computer system. Provisions must be made to allow placement of computation across the available processing elements comprising the system. Each of these areas will be addressed in a separate section below.

2 Parallel Language Constructs

The bulk of the research to date has concentrated on explicit parallelism on shared-memory machines. Notable among these was MultiLisp [?], the first implementation of the **future** construct [?]. Other systems include MultiScheme [?], Butterfly PSL [?], Qlisp [?], and Spur Lisp [?]. A few groups have explored parallelism on distributed-memory platforms; these will be described in more detail.

CCLisp[?] is an attempt to realize increased Lisp performance on a distributed-memory, hypercube-connected multicomputer. The model used is of loosely-coupled processes communicating via streams. Facilities for synchronous and asynchronous *remote evaluation* are provided, but general multithreading of each processor is not.

Avalon Common Lisp[?] is an example of a system pursuing traditional distributed computing goals (i.e., not the exploitation of parallelism for performance). It does provide remote evaluation of expressions; consequently, it must address many of the same problems of transmission of structured data and identity of values that any distributed Lisp faces.

Remote Closures[?] describes a system that utilizes distributed machines to support explicit concurrency in Lisp. The machines are loosely coupled, communicating over a local area network. The basic mechanism relies on the *location* of closures to specify the *locus* of evaluation, a concept similar to the *gateway* mechanism in Concurrent Scheme.

Marti[?] reports on the use of distributed single-threaded PSL-based servers in a local-area network environment. The topology used is a star (one client, several servers). An interesting feature is the systems' integrated support for Time-Warp[?] type simulations.

Concurrent Scheme explores an area that has received scant attention in the existing body of research: explicit parallelism for increased performance on a base of tightly-coupled distributed-memory machines with multi-threading at each processing element. It has been necessary to draw on the results of the shared-memory research as well as the distributed-memory work, and to go outside the Lisp literature to work in other imperative languages.

CS is an attempt to provide a language that will allow a programmer to utilize effectively the computational power offered by a distributed memory multicomputer. Several factors contribute to the level of success in achieving this goal. Among these are:

- the choice of language constructs and their appropriateness for the class of architectures hosting the language;
- the quality and organization of the design of the language;
- the efficiency of the concrete implementation.

This first section will briefly discuss the language constructs of CS and address their appropriateness for the target architecture class. The main focus of this paper, however, is to examine the design of the implementation of those constructs. More complete expositions of CS can be found in[?] and in [?]; the latter describes an earlier version of CS. The use of CS for applications programming is discussed in[?].

2.1 Closures and Domains

The foundation of object-based computation within Concurrent Scheme is the closure[?]. Closures are similar to objects in the object-based programming sense in that they associate computational methods and local state. They can be used explicitly as objects if they are defined to include both local procedures and data, and a *dispatch* procedure[**?**] used to invoke the appropriate local "method." Closures can also be used as the basis for a more advanced object-oriented programming system such as DPOS[**?**].

One of the challenges of designing a parallel programming system is to develop a mechanism that solves the problem of mutual access to shared data. In Scheme, closures can be used to contain data, or state. In CS, they (closures) are the primary site for a program's persistent mutable state; i.e., the use of global variables for this purpose is discouraged. The access control mechanism must work cohesively with closures. To address this question of mutual exclusion, we looked outside the Lisp family of languages. In the language Hybrid[**?**], Nierstrasz presents a model for active concurrent objects. The model introduces *domains*, a mechanism for ensuring mutual exclusion. A *domain* defines a basic indivisible unit of concurrency. In his model, concurrency can exist between domains, but not within them. Nierstrasz' model also provides the *delegate* mechanism, to perform remote (i.e., inter-domain) procedure calls that do not block the calling domain. Our version of domains is similar. At most one thread of control may execute within a domain at any time. Other threads needing to execute within an *occupied* domain are queued outside the domain by the runtime system. A domain operationally encapsulates data and is a dynamic version of Hoare's monitor concept [**?**]. A domain differs from a monitor in that *all* computation occurs within these monitor-like objects (i.e., mutual exclusion is the norm) and data that enters or leaves a domain is *copied* rather than passed by reference. A relationship exists between closures and domains: any particular closure is properly contained within some domain so that the mutable state of the closure is protected from conflicting updates by the mutual exclusivity of the domain.

The potential for concurrency exists when multiple threads of control exist and those threads execute in different domains. When a thread executing in one domain invokes a closure residing in another domain the thread will execute in (and *occupy*) that other domain if it is not currently occupied. Otherwise it will *wait* to enter that domain. The invoking closure's domain will remain occupied while the thread is executing in the current domain, executing in a called domain, or waiting to execute in a called domain. A thread's *footprint* covers all the domains that it simultaneously occupies.

The significant operations are:

- `(make-domain :node` <*node-num*> `:size` <*size*> `:name` <*name*>`)` creates a new domain on the given node with a heap of the given size. The domain name provides the programmer with a way to attach a symbolic identity to a domain for debugging and performance evaluation purposes. All of the arguments are optional. If the node is not specified then the domain is created on the current node. Size defaults to a user-settable value. The name defaults to `'UNNAMED`. `Make-domain` returns a global (system-wide unique) reference to the domain. The new domain is not contained within the domain that caused its creation.[1]

- `(apply-within-domain` *<proc>* *<arglist>* `.` *<domain>*`)` first *enters* the specified domain and then applies the procedure to the given arguments. The domain argument is optional in the case that the `proc` argument is a closure. The return value of the applied procedure is the value of `apply-within-domain`.
- `(delegate` *<proc>* *<arglist>* `.` *<domain>*`)` provides a way for a thread to leave the current domain without following the normal procedure exit protocol. Before the given procedure is applied to its arguments, the thread leaves the current domain, causing it to become unoccupied. It then enters a different domain for execution. When the procedure returns, the thread is placed on the queue to reenter the original domain when it again becomes available. The value of `delegate` is the value returned by the procedure application.

A side effect of the encapsulation of mutable data is that storage allocation and garbage collection (GC) can be limited to the scope of individual domains. Therefore GCs are strictly local to each domain obviating the need for global synchronization for allocation and GC (Section **??**).

2.2 Threads

Concurrency is explicitly specified by the programmer through the action of creating a new thread of control to evaluate a procedure.

- `(make-thread` *<proc>* *<arglist>* `:domain` *<domain>* `:for-value` *<flag>* `:name` *<name>* `)`, given a procedure object, argument list and keyword arguments, creates a new thread of control. The new thread is not necessarily run immediately, so the thread object returned by a `make-thread` can contain a *placeholder* [**?**] and the call returns as soon as the thread is created. The creating process can then continue, using the placeholder in place of the value that `proc`

[1] Note that a syntactic feature of Common Lisp, keyword arguments, has been introduced in the implementation of Concurrent Scheme primitives. It has not, however, been added to CS as a general feature.

will eventually return. `make-thread` is similar to the `future` construct [?], except it cannot be wrapped around arbitrary forms, but is limited to performing procedure application.

`Make-thread` must: determine from the given procedure the initial domain of execution for the new thread; copy the arguments; initiate creation and scheduling of the new thread; create a placeholder to receive the value of the thread's evaluation; and finally return a thread object.

If the procedure given to `make-thread` is not a closure then the domain argument must be specified; the thread will start execution in that domain. If the procedure is a closure then the new thread will start in the domain that contains the closure.

`Make-thread` always does a structure-preserving copy of the procedure's arguments. If the actions of the procedure are intended to produce side-effects such data must be accessible in the procedure's environment rather than being passed as arguments.

When `make-thread` returns, it is not guaranteed that the new thread has been scheduled and/or started. Threads created by sequential calls on `make-thread` may be scheduled or started in a different sequence.

Providing for the return value of a thread incurs costs. When such a value is neither needed nor wanted (i.e. when a thread is executed for its side-effects only), supplying `#f` for the `:for-value` flag suppresses the generation of a placeholder for the thread.

2.3 Placeholders

In addition to the implicit creation of a placeholder by `make-thread`, placeholders may be created explicitly for use as synchronization and communication objects. `(Make-placeholder)` creates a new, unresolved placeholder structure:

```
(defstruct placeholder
  determined? ; #t->placeholder has been resolved
  value       ; the value, whenever it gets one
  waiting     ; ptr to list of threads waiting for this value
  external    ; export table reference for this placeholder
  thread)     ; thread that will determine placeholder or '()
```

An entry for the new placeholder is placed in the export table and the global reference to the placeholder is returned. When `Make-thread` invokes `make-placeholder`, it saves the global reference to the placeholder in the thread's `target` slot, and puts a reference to the thread in the `placeholder`'s `thread` slot (the latter for debugging).

Placeholders are global and can be used to provide synchronization. Placeholders can also be used to "broadcast" values to threads, as many threads may hold references to a placeholder.

An unresolved placeholder is one that has not yet been given a value. Attempting to use the value of an unresolved placeholder in a strict operation causes the current thread to block. The blocked thread will wait until the placeholder receives a value.

Placeholders can receive values in two ways: a placeholder created by `make-thread` will receive the value returned by that thread upon completion; a placeholder may be explicitly given a value by `determine`.

- `(determine` <*placeholder*> <*value*>`)`, given an unresolved placeholder and a value, `determine` makes that value the value of the placeholder. If the placeholder already has a value, `determine` signals an error. The value given to the placeholder is actually an immutable copy of the value and does not exist within any (user-accessible) domain.
- `(determined?` <*placeholder*>`)` is used to see if a placeholder has been resolved.
- `(touch` <*any*> `.` <*flag*> `)`. If the argument is an unresolved placeholder `touch` causes the current thread to block until the placeholder receives a value. Otherwise it returns the argument (or the value of a placeholder). If a second non `#f` argument is given to `touch`, then `touch` will leave the current domain unoccupied before attempting to `touch` the placeholder. The thread will re-enter the domain when the `touch` operation completes. Use of this option allows the domain to be entered by other threads while the `touch` operation completes and gives the programmer flexibility in controlling access to the domain. The value of `touch` is a copy of the placeholder's value.

CS also provides other synchronization mechanisms (such as *delay-queues* [?]) which we do not discuss in this paper.

2.4 Mapping Domains to Processors

The specification of CS does not directly address the issue of automatic resource allocation. Resource allocation is left to the programmer in the form of explicit specification of the location (node) of each newly created domain. Programmer specified resource allocation is supported by the provision of current system configuration information:

- `(this-node)` returns the zero-relative logical node number. It differs depending upon which node a thread is running.
- `(node-count)` returns the number of nodes in the system.

Using these values, a program can compute a node number to use as an argument to `make-domain` in order to distribute domains (and their potential computations) among the available nodes. Resource allocation in CS is entirely dynamic, in that domain creation always occurs at run-time.

2.5 An Example CS Program

This example solves the N-queens problem. The program creates a separate thread to search each tree corresponding to initial placement of a queen into a particular column of the first row of the board. `Conflict` determines if a queen can be a placed on in a given position of an existing a board and `store`, a closure, stores generated solutions. `Make-thread` is used to create the threads for each search subtree. The threads are distributed across the system in a round robin fashion using `make-domain`, which returns a domain which can potentially exist on another node. For example, `(queens 4)` would create 4 new threads, each running the `queens1` procedure. If there were 4 nodes in the system, each would run in parallel with the others.

```
(define (queens size)
  (set! store
        (apply-within-domain make-store '()
                             (make-domain :name 'store)))
  (do ((col 1 (+ 1 col)))
      ((> col size))
      (make-thread queens1 (list size nil 1 col)
                           :domain (make-domain :node col))))

(define (queens1 size board row col)
  (if (null? (conflict board row col))
    (begin (set! board (cons (cons row col) board))
           (if (eq? row size)
             (make-thread store (list board))
             (let ((next-row (+ 1 row)))
               (do ((j 1 (+ 1 j)))
                   ((> j size))
                   (queens1 size board next-row j)))))))

(define (conflict ....)
   ...)
```

```
(define (make-store)
   ...)
```

3 Implementation of Parallel Language Constructs

CS is implemented in three layers. The first layer, the microkernel, is written in C. It defines a set of low-level primitives for doing message packetization, packet transmittal and reception, primitive scheduling, etc. The abstractions it provides fall generally into the two categories of ports and coroutines. A port can be viewed as a queue of messages. In particular, the microkernel provides a "Lisp port" for the transmission and receipt of messages by the next higher layer. The microkernel is language-neutral and is also used to support a distributed C++ implementation.

The middle layer (Lisp kernel) utilizes those abstractions to implement scheduling policy, storage management, thread creation, and other operating system-like services. Examples of calls at this level are `sys-touch`, `sys-determine`, `sys-determined?`, and `schedule-thread`. Though implemented in Lisp, these system calls differ from user level Lisp in that no preemption occurs at this level, providing de-facto atomicity for all operations. It is implemented using microkernel system calls and provides a set of remote operations, including procedure call, all built upon a message-passing framework.

The third layer is user-level library code which provides the interface to the Lisp kernel. It performs correctness checking for arguments and implements those CS functions that do not require guaranteed atomicity.

3.1 Threads

A thread represents an independent computational activity; it is the concurrency abstraction presented to the user by CS. A coroutine performs a thread's activity within a given domain. A coroutine is made up of a stack (initially small and expandable), a program counter, and the identity of the thread it executes on behalf of. Threads may have zero or more coroutines associated with them. The zero case is for a thread which is either terminated but still "live" (referenced) or created but not yet started.

```
(defstruct thread
  coroutine    ; the coroutine currently executing the thread
  identifier   ; system-wide unique id
  state        ; one of {running, blocked, runnable, etc.}
  target       ; ref to a placeholder or '()
  return-value); used by agent to pass value
```

Coroutine creation is an expensive activity, relative to the small grain of other thread activities, such as function call. CS addresses this with a pool of daemon coroutines and agents. A daemon coroutine loops forever, reading a message from the Lisp port and performing the activity specified in the message. Normally, several daemons will be blocked waiting to read from the Lisp port. When a message arrives, one daemon is awakened and given that message. If the daemon pool is empty when a daemon is awakened, it creates a new daemon to wait for the next message.

3.2 Agents

In a CS program, the computational weight, or grain size, of threads can vary dramatically. Further, since the evaluation of a thread in CS may span several domains, a thread's activity actually is comprised of a series of subactivities. This subdivision is necessitated by the distribution of domains across separate nodes. The grain size of these subactivities is necessarily smaller than the thread's total activity, and in many cases is known to be extremely small. Clearly, any subactivity must be performed by some coroutine. Some subactivities are extremely simple and demand very little context (e.g., a subactivity may have no possibility of blocking, or it may never need to perform storage allocations). At the other extreme is the subactivity of creating a new thread; this involves the full context and unpredictable grain size associated with threads. To avoid paying the relatively high cost of coroutine creation and establishment of context when these are unnecessary, while maintaining the flexibility needed to create new threads, CS uses *agents*.

Agents are the anonymous daemon coroutines introduced in Section **??**. They possess minimal context of their own. When an existing thread needs to execute in another domain (e.g., remote procedure application), or needs to manipulate global objects such as placeholders (which entails working "outside of" any user domains), an agent is used. The agent establishes only the required amount of context to perform the subactivity and, when it is done, it de-establishes that context in preparation for performing the next available subactivity. They are called agents because they work on behalf of a thread and, when necessary, assume that thread's identity.

Messages specify remote agents to perform a specific task on remote nodes. The activities these agents perform include `touch`, `determine`, remote procedure application, remote procedure return, etc. For example, a remote procedure call (RPC) consists of a pair of messages. We discuss the case of RPC in this context.

3.2.1 *Remote Procedure Calls as Message Passing*

Remote procedure call is a central mechanism in CS; inter-domain application of closures, `apply-within-domain` and `delegate` are implemented as RPC. An RPC is like a normal procedure call; the difference is that the application of the procedure will occur in a different domain. The caller will be a coroutine operating within some domain. The destination node is determined based on the domain the called procedure resides in, or is explicitly specified by `apply-within-domain`. Once system Lisp detects a remote procedure call or an `apply-within-domain` application it invokes the message passing software to perform the RPC transaction.

To initiate an RPC, a message containing the procedure, the copied arguments (Section **??**), the unique identifier of the calling thread, a reference to the calling coroutine/agent, and the domain reference is constructed. The message and the destination node number are given to the `send-message` system call (Section **??**). The calling thread identifier is included so that the agent executing on the remote node can assume the thread identity of the calling coroutine (and thread). The agent receiving the message assumes that thread identity and applies the procedure to the given arguments within the specified domain. It must be capable of executing arbitrary Lisp code and hence needs to establish a full context. In particular, it enters some specified domain, an operation that requires the thread identity and which may require the coroutine to block. When the procedure returns, the agent constructs a message containing a reference to the calling agent and a copy of the result value. This message is sent back to the originating node. The agent de-establishes its context (which might include saving heap allocation pointers) and returns to reading from the Lisp port.

When the return message arrives at the originating node it is queued on the Lisp port. A daemon reads the message and, using the agent reference contained therein, schedules the (original) calling thread for resumption. This is a case of an agent that requires minimal context to perform its work, as it need not assume the thread identity, nor need it be capable of being blocked.

3.3 Other Remote Operations

Besides RPC, remote implementations of certain *operations* must be provided. These operations include `touch`, `determine`, and `determined?`, when they are applied to remote placeholders. The same mechanisms of message passing and agents are used for these.

For example, when a thread touches a placeholder that resides on a remote node, the use of an agent is required. In terms of weight or grain, this agent lies in between the RPC application activity and RPC thread resumption activity. It can potentially block if the placeholder is not yet determined; it must also copy an arbitrary value into a return message. It does not need to assume the identity of the invoking thread (except perhaps for informational/debugging purposes). A simple optimization can dramatically decrease this agent's weight: rather than blocking on an undetermined placeholder, it can queue the return message it would have sent on the placeholder and let the `determining` thread or agent send that message when the placeholder is `determined`. Thus, this agent need not be capable of blocking.

3.4 Agent Management

When the Concurrent Scheme system starts up, a number of agents are created (currently on the order of eight). If this number is insufficient to handle the concurrently active threads and remote operations, additional agents are created as needed, up to a limit imposed by the available memory for coroutine control blocks and stack allocation. Creating an agent is relatively expensive (see Section **??**), thus their reuse for many operations is necessary to amortize initial creation costs.

3.5 Intensional Copying

Any data that passes a domain boundary is copied. CS does intensional copying to preserve structure sharing so as not to deviate from standard Scheme semantics (other than the copying of arguments). The complexity of the copying algorithm is $O(n)$, where n is the size of the item being copied. Copying itself, whether structure-preserving or not, is performed to provide the encapsulation semantics specified by Concurrent Scheme.

All data types are contained within a domain except domains themselves, threads, and placeholders. These are de facto global items and are the only data passed by reference (besides closures). These can be viewed as "metatypes" which implement the parallelism in the program. Since they exist outside the address space of any user computation we use *export* tables to associate their global references to their actual locations.

A global reference consists of a tag (to determine its metatype) a node number, and an index into the export table of the node. Whenever one of the above metatypes is created, an entry for it is made in the export table of the node upon which the object is created. These global references are what are returned to the users by the constructor procedures.

When a closure is exported out of a domain, it is *not* copied. Rather, an entry is made for it in the export table and a global reference is created. The global reference is passed in the message. On the receiving end, the global reference is wrapped up in a closure (*gateway*) so that it is an applicable object.

3.6 Storage Management

There are two views of Lisp storage management in CS. Within each view there are two levels of management. The first view occurs when we restrict ourselves to a particular node. There we have a level of domain-local allocation/collection. We also have a level of managing multiple disjoint domain heaps within the heap address space of the particular node. The second view is system-wide. Here we are interested in reclaiming global items that have become inaccessible (i.e., domains, threads, placeholders, exported closures). This is accomplished by a reference-counting mechanism; domain-local garbage collections cooperate in maintaining these counts.

The domain-local algorithm is a standard stop-and-copy collector [**?**, **?**]. The management of the node virtual space is done with explicit deallocation and via the buddy system [**?**]. The global reclamation is done with a distributed reference counting algorithm using asynchronous messages.

4 Microkernel Implementation

The microkernel can be viewed as two components. One is a microkernel thread performing message passing and coroutine switching. The other part comprises a set of system calls executed by Lisp threads in the microkernel context.

4.1 Kernel Thread Tasks

The lowest level thread is the microkernel thread. It handles primitive scheduling, message packetization, and packet transmittal and reception.

Briefly, low-level message reception operates as follows. The microkernel contains a queue of packets that have arrived from the Post Office [**?**, **?**], the HPIB, or network (depending on the host architecture). Packets are taken from that queue and placed in the proper position (with potential out-of-order packet arrival) in the packet's associated message buffer; a buffer is created if there is none. When all message packets have arrived, the message buffer is placed on the Lisp port to be picked up by a daemon coroutine. Low-level message transmission simply picks up message buffers from an outgoing message queue and sends the data as individual packets.

There are several low-level scheduling operations. For instance, a user level coroutine will need to switch to Lisp kernel mode to grow its stack, to be preempted, or to execute a parallel primitive. Such coroutines, as well as newly forked coroutines and coroutines that blocked in Lisp kernel mode, are scheduled on a "system Lisp request queue" to be executed in system Lisp mode. The microkernel schedules daemon coroutines to handle incoming messages on the Lisp port, and it schedules coroutines that are ready to be run in user mode on a "user level run queue."

4.2 Microkernel System Calls

The microkernel system calls fall into two groups: message passing and coroutine management. When a coroutine needs to send a message it gives the `send-message` system call a destination and a message buffer. It simply queues it for later transmission by the microkernel thread. Received messages are handled by daemon coroutines, as mentioned above. Message buffer allocation and deallocation are also handled by system calls at this level.

New coroutines are created by a `lisp-fork` system call. Coroutine execution is managed by `block-coroutine`, `resume-coroutine`, `enter-user-mode`, and the `enter-lisp-kernel-mode` system calls.

5 Realizations

The first version of CS was running in January 1989. Since then it has undergone four major revisions and refinements. We currently have realizations on a number of systems.

5.1 CS on the Mayfly

Each Mayfly node is an asymmetric shared-memory dual processor. One processor (the evaluation processor — EP) runs user code. The other processor (the "message processor" — MP) runs Lisp kernel code and the microkernel code. The EP only communicates with the node's memory system and the floating point processor, making it appropriate for user applications. The MP communicates with the EP, memory system, and message passing hardware.

A unique feature of the memory system is the *context cache*. The context cache provides fast context switching. Each coroutine has a context (analogous to a process control block in an operating system) contained in the context cache.

5.2 CS on Networked Workstations and the BBN Butterfly

The network version provides message passing services via BSD sockets [?, ?] to communicate between nodes, which in this case are workstations on a local area network. It currently runs on homogeneous collections of either HP Series 9000/300 and 400 or HP Series 9000/800.

The Butterfly [?] version was used to emulate Mayfly nodes before the hardware became available. It used shared-memory for message passing. It is no longer supported.

5.3 CS on Multicomputers under Mach

A version of CS that relies on Mach ports for communications and on Mach[?, ?] kernel threads for its coroutines is being developed. It will provide a CS that is portable (in terms of low-level support) to any multicomputer running a Mach microkernel. Other issues impact portability, of course, such as availability of a Scheme compiler for the processor architecture.

6 Preliminary Timing Results

The following timing measurements were taken on the network version of CS running on HP Series 9000/370s running BSD 4.3. Timing measurements for some of the typical system operations are as follows.

operation	microsecond
simple procedure call	12
allocate message	500
create coroutine	1750
empty Lisp system call	140
make placeholder	170
make domain	240
determine	265
apply-within-domain	1530

Times to note in particular are those for âpply-within-domain and coroutine creation. In the absence of agents, the time to perform an RPC (`apply-within-domain`) would also include a coroutine creation, for a total time of 3250 microseconds.

The following timing results were obtained by running the example program of Section ??. Other results can be found in [?].

The N-queens example program, when given a size argument of 8, uses 362 threads and makes 9 domains.

Number of nodes	time in seconds
1	6.4
2	3.9
3	3.2
4	2.5
5	2.4
6	2.3

The identical program with `make-thread` and `make-domain` removed (i.e., no parallelism) ran in 5.3 seconds.

7 Conclusion

Concurrent Scheme is intended to be a language that is easy to use and understand and that fits well on a distributed memory architecture such as the Mayfly. The programming model envisioned is of large scale systems running programs consisting of even larger numbers of heterogeneous objects.

Ease of use and understanding results from the small number of well-defined and mutually consistent mechanisms that CS provides. In particular, in the domain mechanism it has merged the concepts of mutual exclusion, a necessity in concurrent imperative systems, with data encapsulation, a feature of object-based programming. This marriage reduces the danger of unexpected asynchronous modifications to shared state that can arise in other concurrent Lisps, while allowing familiar imperative programming techniques to be used within the boundaries of domains. Because mutual exclusion is automatic, accidental decoupling of mutual exclusion from the resource it protects is impossible. On the other hand, facilities are provided that allow the programmer to *explicitly* relax the mutual exclusion properties. The task of reasoning about a CS program should be eased by these guarantees and by the explicitness of any violation of them.

In managing to avoid overspecifying those mechanisms and in discouraging programmer dependence on certain others (e.g., mutable top-level variables), it leaves room for an implementor to choose strategies that deliver the most from his target architecture. Tests of the existing local area network implementation, an "architecture" that is deficient in many important areas such as communication latency, bandwidth, and contention and context switch overhead, still yield moderate speedup and scalability. Implementations on more suitable architectures such as the Mayfly, that do not suffer these deficiencies, should deliver considerably better speedup and scalability.

The current implementation of Concurrent Scheme affords ease of portability by its layered nature and by the limited demands it makes of the host operating system. It offers reasonable performance in part due to its use of persistent coroutines (agents) as the concrete realization of abstract threads and remote operations, thus avoiding costly frequent coroutine creation and deletion activities.

8 Future Work

A class on parallel programming at the University of Utah was taught during winter of 1991. Based on that experience and our own ongoing experience in writing example applications, we will be tuning the system and modifying the parallel constructs provided to the user.

Under the present CS programming model, it is up to the applications programmer to map a computation to individual nodes. Automated resource allocation and domain migration is being investigated.

CS software is downloaded to Mayfly nodes by a small "loader" kernel which also provides access to the HPIB (primitive input/output), and assembly language level debugging. This bare-bones "operating system" is too fragile for general use. We are in the process of replacing it with a multiprocessing system kernel — MACH.

We currently program directly in CS. An effort is underway to provide a more sophisticated object-oriented parallel programming language on top of the CS primitives.

We are also beginning to explore parallel extensions to other programming languages such as C++. CS domains work in conjunction with lexical closures with indefinite extent. Since this type of closure is a feature of Scheme, domains and Lisp work well together. Mapping the domain idea (or coming up with a new model) to C++ is an active area of research.

The network version of CS is primarily used to develop and debug applications programs to be run on the Mayfly. However, the network version is useful in its own right, showing speedup on certain programs, and giving the ability to partition large data sets across many machines. Our current network version depends on the text space (code) being identical on each node. To be able to utilize available machines we plan to design an alternate representation of remote procedure pointers such that machines of different types (such as m680x0 and HP-PA RISC) can run a CS application between them.

References

[AR88] Norman Adams and Johnathan Rees. Object-oriented programming in scheme. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 277–288, July 1988.

[AS85] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[Bac88] B.. Bacarisse. Remote closures. Technical Report HPL-ACS-88-41, Hewlett-Packard Research Laboratory, August 1988.

[Bak78] H.G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[BBT87] David Billstrom, Joseph Brandenburg, and John Teeter. Cclisp on the ipsc concurrent computer. In *Proc. 6th National Conference on Artificial Intelligence*, 1987.

[BH77] H. Baker and C. Hewitt. The incremental garbage collection of processes. AI Memo AIM-454, MIT AI Laboratory, Cambridge MA, December 1977.

[But87] BBN Advanced Computers, Inc., Cambridge MA. *Butterfly Parallel Processor Overview*, 1987.

[CN90] L. D. Clamen, S. M.and Leibengood and J. M. Nettles, S. M.and Wing. Reliable distributed computing with avalon/common lisp. In *1990 International Conference on Computer Languages*, 1990.

[Dav91] A. L. Davis. Mayfly: A general-purpose, scalable, parallel processing architecture. In *(Submitted) International Joint Conference on Artificial Intelligence*, 1991.

[DR85] A. L. Davis and S. V Robison. The architecture of the faim-1 symbolic multiprocessing system. In *Proc. IJCAI-85*, pages 32–38, 1985.

[EK91] John D. Evans and Robert R. Kessler. Dpos: A metalanguage and programming environment for parallel processing. In *(Submitted) International Joint Conference on Artificial Intelligence 1991*, 1991.

[FY69] R. R. Fenichel and J. C. Yochelson. A lisp garbage-collector for virtual memory computer systems. *Communications of the ACM*, 12(11), November 1969.

[GG88] R. Goldman and R.P. Gabriel. Preliminary Results with the Initial Implementation of Qlisp. In *Proc. 1988 ACM Conference on Lisp and Functional Programming*, 1988.

[HJ85] R.H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[HJ86] R.H. Halstead Jr. An Assessment of Multilisp: Lessons from Experience. *International Journal of Parallel Programming*, 15(6):459–501, December 1986.

[Hoa74] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Jef85] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[Knu68] D. E. Knuth. *Fundamental Algorithms, The Art of Computer Programming, Volume 1.* Addison Wesley, 1968.

[KS90] R. R. Kessler and M. R. Swanson. Concurrent scheme. In T. Ito and R. H. Halstead, editors, *Lecture Notes in Computer Science, Parallel Lisp: Languages and Systems*, pages 200–234. Springer-Verlag, 1990.

[Lea86] S.J. Leffler et. al. An advanced 4.3bsd interprocess communication tutorial. Computer science research group, University of California, Berkeley, Department of Electrical Engineering and Computer Science, 1986.

[Mac90] M. Mackey. An evaluation of concurrent scheme. Hewlett-Packard Pisa Science Center Technical Report HPL-PSC-90-9, Hewlett-Packard, Pisa, Italy, December 1990.

[Mar88] Jed Marti. RISE: The RAND integrated simulation environment. In David Jefferson Brian Unger, editor, *Distributed Simulation*, volume 19, San Diego, California, 1988. Simulation Councils, Inc.

[Mil87] J. S. Miller. *MultiScheme, A Parallel Processing System Based on MIT Scheme.* PhD thesis, Department of Electrical Engineering and Computer Science, MIT, August 1987.

[Nie87] O. M. Nierstrasz. Active Objects in Hybrid. In *Object-Oriented Programming Systems, Languages, and Applications 1987 Conference Proceedings*, pages 243–253, 1987.

[Ras86] R. F. Rashid. From RIG to Accent to Mach: The Evolution of a Network Operating System. In *Proceedings of the ACM/IEEE Computer Society, Fall Joint Computer Conference, ACM*, November 1986.

[Ras87] R. F. et. al. Rashid. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the 2nd Symposium on Architectural Support for Programming Languages and Operating Systems, ACM*, October 1987.

[RC86] J. Rees and W. Clinger. Revised^3 Report on the Algorithmic Language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[Sec86] S. Sechrest. An introductory 4.3bsd interprocess communication tutorial. Computer science research group, University of California, Berkeley, Department of Electrical Engineering and Computer Science, 1986.

[SKG88] M. Swanson, R. Kessler, and Lindstrom G. An Implementation of Portable Standard LISP on the BBN Butterfly. In *Proc. 1988 ACM Conference on Lisp and Functional Programming*, pages 132–142, 1988.

[SRD86] Kenneth S. Stevens, Shane V Robison, and A.L. Davis. "The Post Office – Communication Support for Distributed Ensemble Architectures". In *Proceedings of 6th International Conference on Distributed Computing Systems*, pages 160 – 166, May 1986.

[Ste86] Kenneth S. Stevens. The Communications Framework for a Distributed Ensemble Architecture. AI 47, Schlumberger Palo Alto Research, February 1986.

[Swa91] M. R. Swanson. *DOMAINS–A Mechanism for Specifying Mutual Exclusion and Disciplined Data Sharing in Concurrent Symbolic Programs.* PhD thesis, Department of Computer Science, University of Utah, June 1991.

[ZHL+89] B. Zorn, K. Ho, J. Larus, L. Semenzato, and P. Hilfinger. Multiprocessing extensions in spur lisp. *IEEE Software*, 6(4):41–49, July 1989.