Locality Abstractions for Parallel and Distributed Computing

Suresh Jagannathan

Computer Science Research, NEC Research Institute, 4 Independence Way, Princeton, NJ 08540, *suresh@research.nj.nec.com*

Abstract. Temporal and spatial locality are significant concerns in the design and implementation of any realistic parallel or distributed computing system. Temporal locality is concerned with relations among objects that share similar lifetimes and birth dates; spatial locality is concerned with relations among objects that share information. Exploiting temporal locality can lead to improved memory behavior; exploiting spatial locality can lead to improved communication behavior. Linguistic, compiler, and runtime support for locality issues is especially important for unstructured symbolic computations in which lifetimes and sharing properties of objects are not readily apparent.

Language abstractions for spatial and temporal locality include mechanisms for grouping related threads of control, allowing programs flexibility to map computations onto *virtual processors*, reusing dynamic contexts efficiently, and permitting asynchronous garbage collection across multiple processors. These abstractions give users and implementations a large degree of mobility to exploit inherent locality properties found within many dynamic parallel applications.

We have investigated a number of these abstractions within a high-level language framework and within compilers targeted for such a framework. In this paper, we discuss several of these abstractions and justify their importance.

1 Introduction

Despite significant advances in hardware construction and parallel algorithm design, there remain a large class of algorithms which typically perform poorly on conventional stock multiprocessors. These problems (broadly classified under the name *symbolic programming*) include combinatorial optimization, simulation, symbolic algebra, database searches, graph algorithms, expert systems, etc.. Symbolic programming applications exhibit characteristics not found in more well-structured numerical algorithms: data is generated dynamically and often have irregular shape and density, data sets typically consist of objects of many different types and structure, the communication requirements of generated tasks is a rarely apparent from simple examination of the source program, and the natural unit of concurrency often is difficult to determine statically.

High-level symbolic programming languages such as Scheme[3] or ML[25] nonetheless offer the promise of being ideal vehicles within which to express a variety of issues related to concurrency, distribution and communication. Because these languages support the liberal use of abstract data types and structures, first-class procedures with well-defined encapsulation rules, and first-class continuations[13], many concerns in the design of parallel or distributed system can be elegantly defined. However, there has been little effort to investigate the role of these languages and their implementations within such contexts.

One obvious way of integrating concurrency into a high-level symbolic language is to use generic thread packages[4, 34] or operating system-defined library routines. At first glance, such an approach appears to offer greater extensibility and generality than available within a high-level language optimized for a particular paradigm. Efficiency and expressivity are compromised in several significant respects, however: (1) because there is no compile-time or runtime system sensitive to the semantics of the concurrency abstractions being implemented, there is little or no optimization performed on these abstractions; (2) relying on external libraries and operating system services for concurrency management and communication effectively prohibits utilizing features of the sequential language to implement concurrent and distributed extensions, and (3) because scheduling and migration policies, storage management decisions, etc., are managed exclusively by generic thread routines, building customized implementations of different paradigms is problematic. The latter shortcoming makes this approach especially ill-suited for implementing diverse parallel and distributed symbolic algorithms.

In this paper, we present design, compilation and implementation strategies that address the issue of implementing high-level parallel languages for symbolic computing. We focus predominantly on locality issues since most symbolic applications exercise communication seriously and in ways that are often difficult to analyze statically. We first describe a system geared towards high-performance tightly-coupled parallel applications. We then present a compile-time analysis framework for a higher-order language core capable of deriving information that can be used by optimizers sensitive to locality considerations.

We classify locality considerations into two categories: *temporal locality* is concerned with relations among objects that share similar lifetimes and birth dates; *spatial locality* is concerned with relations among objects that share information. Exploiting temporal locality requires mechanisms which allow objects that are created close to another in time to live close to another in space. Exploiting spatial locality requires mechanisms which allow objects that live close to another in space to be accessed close to another in time. Temporal locality is most directly influenced by memory organization – caches, for example, improve temporal locality since caches are organized to allow short access times for newly allocated objects. Spatial locality is most directly influenced by communication structure – a shared immutable data structure that is replicated, for example, does not require inter-processor communication in order to be referenced.

In the next section, we present an overview of Sting, a high-level operating system for parallel computing implemented in Scheme. In Section 3, we define Sting's storage organization model tuned for optimizing locality inherent in lightweight parallel computations. Section 4 describes *virtual topologies*, an abstraction that provides a mapping between a logical task structure and a physical interconnection. Section 5 presents some benchmark results, and Section 6 presents work related work to the Sting design. Section 7 discusses compile-time optimizations for shared locations in the context of a system such as Sting's.

2 Sting Overview

Sting is a parallel dialect of Scheme[3] designed to serve as a high-level operating system for modern symbolic parallel programming languages. A detailed description of its implementation is given in [19, 20, 28]; we concentrate here on features explicitly designed to exploit locality.

The abstract architecture of Sting (see Figure 1) is organized hierarchically as a layer of abstractions. The lowest level abstraction is a *physical machine*. A physical machine consists of a collection of *physical processors*. A physical processor is a faithful abstraction of an actual computing engine. A physical machine contains as many physical processors as there are nodes in a multiprocessor environment.

A *virtual machine* is an abstraction that is mapped onto a physical machine or a portion thereof. Virtual machines manage a single address space. They are also responsible for mapping global objects into their local address space. In addition, virtual machines contain the root of a live object graph (*i.e.,* root environment) that is used to trace the set of live objects in this address space. A virtual machine is closed over a set of *virtual processors* in the system. Virtual processors execute on physical processors.

Physical machines are responsible for defining new address spaces, managing global shared objects, handling hardware device interrupts and coordinating physical processors. Associated with each physical processor P is a virtual processor policy manager that implements policy decisions for the virtual processors that execute on P.

Virtual processors (VPs) are responsible for managing user-created *threads*, an abstraction of a separate locus of control; threads are represented as a generalized form of one-shot continuations[13]. In addition, virtual processors also handle non-blocking I/O and software interrupts (page faults, thread quantum expiration, etc.). Each virtual processor V is closed over a policy manager (TPM) that (a) schedules threads executing within V, (b) migrates threads to/from other

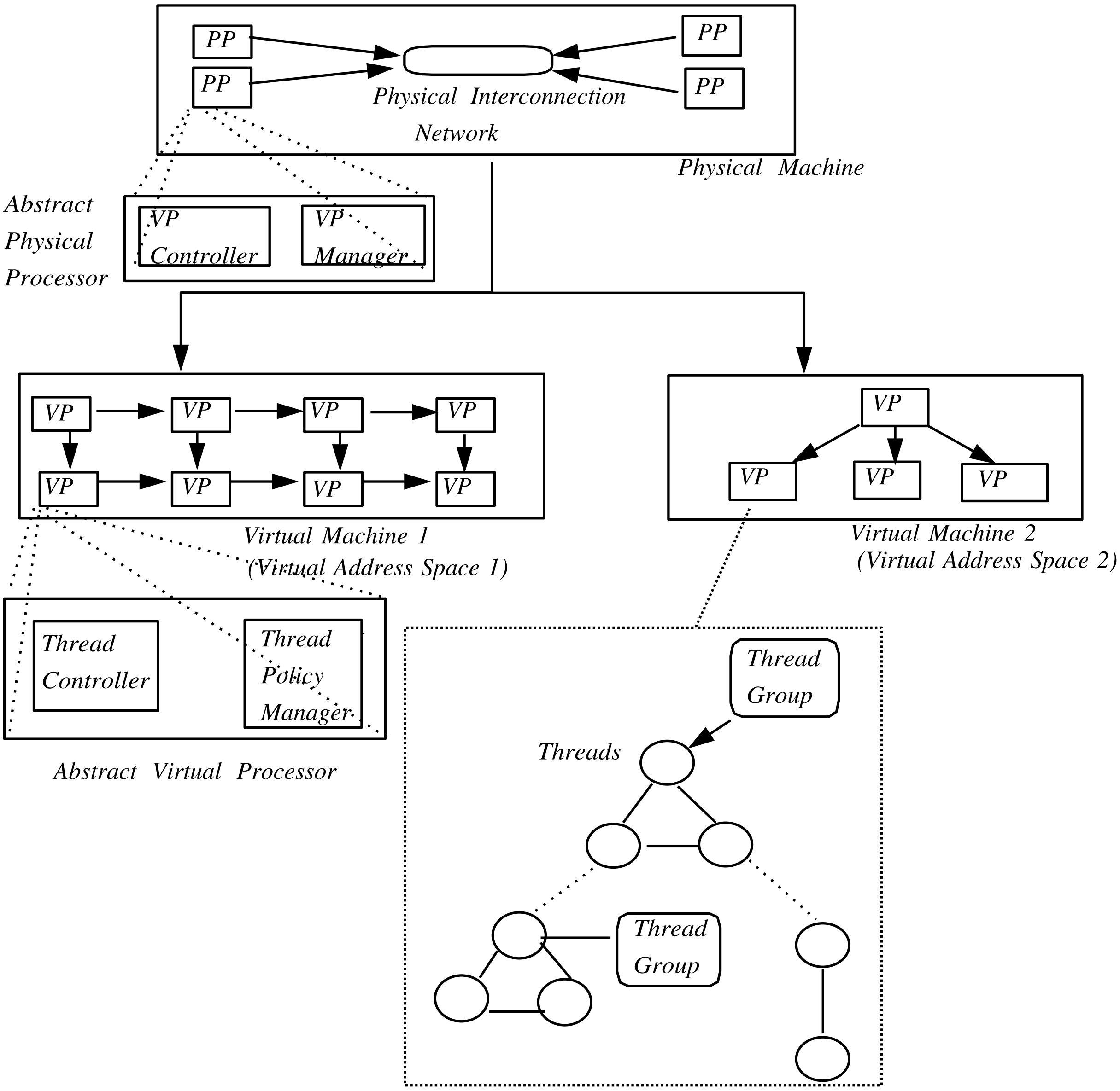

Fig. 1. The Sting software architecture.

VPs, and (c) performs initial thread placement on the VPs defined within a given virtual machine .

Just as VPs context switch threads, a physical processor will context switch virtual processors because of preemption, or because a VP specifically requests a context switch (*e.g.,* because of an I/O call initiated by its current thread). We discuss virtual processors in greater detail in the following sections. Thread functionality is implemented by a thread controller that implements the thread interface and thread state transitions.

Physical machines, physical processors, virtual machines, virtual processors and threads are all first-class objects in Sting. This implies that the policy decisions of a processor (both physical and virtual) can be customized; for example, different virtual processors (even in the same virtual machine) may be closed over different policy managers. No performance penalty is incurred as a result of this flexibility.

2.1 Locality

A major goal of the Sting design was to exploit inherent locality in parallel Scheme programs. Consequently, there are a number of optimizations and design features in the system that are sensitive to temporal locality:

1. Threads allocate private data on local heaps; thus, data with close birth dates are likely to be allocated close to one another within a thread.
2. Thread control blocks (TCBs) are recycled on a per processor basis. Thus, a thread that terminates on processor P will have its TCB assigned to a new thread subsequently instantiated on P, thus improving cache and page locality.
3. TCBs can be shared by many threads simultaneously if data dependencies warrant. Thus, data allocated by a thread T can use the same memory resource as data allocated by T' if T has a data dependency with T'.

Sting also provides mechanisms for exploiting spatial locality:

1. *Thread groups* are first-class objects that encapsulate a shared heap. Data shared among threads within a given group is allocated on the shared heap; in general, inter-thread communication is mediated via access and mutation of the shared heap.
2. *Virtual topologies* provide a means for a program's logical task structure to be mapped onto a physical topology. A virtual topology can be thought of as a scheduler and thread placement manager parameterized over logical and concrete algebraic structures whose elements are processors. For example, one can construct a virtual topology that maps logical trees to concrete meshes; the nodes in the tree and mesh correspond to virtual and physical processors, respectively. The role of a topology is to provide a mapping that optimizes communication among related threads in a program's logical task structure with respect to an underlying physical interconnect.

3 Sting Storage Management

Storage management decisions are undertaken at various points during the evaluation of a thread; these decisions relate directly to temporal locality:

Local Storage Management

Threads are closed over their own local stack and heap. Thus, they are free to garbage collect inaccessible data without having to synchronize with other evaluating threads. Since a thread-local object O may contain references to other objects on shared heaps, garbage collecting O requires notifying the shared heaps into which O has references that O is no longer a live object. When these shared heaps garbage collect, they use such information to determine the root set used to trace all their live objects.

3.1 Thread Control Blocks

The Sting implementation defers the allocation of storage for a thread until necessary. In many thread packages, the act of creating a thread involves not merely setting up the environment for the process to be forked, but also allocating and initializing storage. This approach lowers efficiency in two important respects: first, in the presence of fine-grained parallelism, the thread controller may spend more time creating and initializing threads than actually running them. Second, since stacks and process control blocks are immediately allocated upon thread creation, context switches among threads often cannot take advantage of cache and page locality.

Page locality in thread create operations is obviously important. Consider a virtual processor P executing thread T_1. Suppose T_1 spawns a new thread T_2. Under a fully eager thread creation strategy, storage for T_2 will be allocated from a new TCB. T_2's heap and stack will be mapped to a new set of physical pages in P's virtual machine. Thus, assume T_2 runs on P only after T_1 completes (*e.g.,* because there is no other free processor available or because T_1 is strict in the value yielded by applying T_2's thunk). Setting up T_2 now involves a fairly costly context switch. The working set valid when T_1 was executing is now possibly invalid; in particular, the contents of the data and instruction cache containing T_1's physical pages will probably need to be flushed when T_2 starts executing.

Thread control blocks are recyclable resources that are managed by virtual processors. A dynamic context (*i.e.,* thread control block) is allocated for a thread *only* when the thread begins evaluation. A thread control block consists of stacks and heaps plus a small amount of bookkeeping information used to record the status of the evaluating thread.

Thread control blocks are allocated from a pool local to each VP. This pool is organized as a LIFO queue. When a thread terminates, its TCB is recycled on the TCB pool of the VP on which it was executing; the LIFO organization of this pool guarantees that this TCB will be allocated to the next thread chosen for execution on this VP. Since it is likely that the most recently used pieces of the TCB will be available in the physical processor's working set, temporal locality between newly terminated threads and newly created ones can be exploited. Sting incorporates one further optimization on this basic theme: if a thread T terminates on VP V, and V is next scheduled to begin evaluation of a new thread (*i.e.,* a thread that is not yet associated with a dynamic context), T's TCB is immediately allocated to the new thread; no pool management costs are incurred in this case.

Besides local VP pools, VPs share access to a global TCB pool. Every local VP pool maintains an overflow and underflow threshold. When a pool overflows, its VP moves half the TCBs in the pool to the global pool; when the pool underflows, a certain number of TCBs are moved from the global pool to the VP-local one. Global pools serve two purpose: (1) they minimize the impact of program behavior on TCB allocation and reuse, and (2) they ensure a fair distribution of TCBs to all virtual processors. Since new new TCBs are created only if both the global and the VP local pool are empty, the number of TCBs actually created during the evaluation of a Sting program is determined collectively by all VPs.

3.2 Dynamic Storage Management

The distinction between threads that are scheduled and those that are evaluating is used in Sting to optimize the implementation of non-strict structures such as *futures*[12]; it is also used to throttle the unfolding of the process call tree in fine-grained parallel programs. In a naive implementation, a process that accesses a future (or which initiates a new process whose value it requires) blocks until the future becomes determined (or the newly instantiated process yields a value). This behavior is sub-optimal for the reasons described above – temporal locality is compromised, bookkeeping information for context switching increases, and processor utilization is not increased since the original process must block until the new process completes.

Sting implements the following optimization: a thunk t associated with a thread T that is in a *scheduled* or *delayed* state is applied using the dynamic context of a thread S if S demands T's value[1]. The semantics of future/touch (or any similar producer/consumer protocol) is not violated. The rationale is straightforward: since T has not been allocated storage and has no dynamic state information associated with it, t can be treated as an ordinary procedure and evaluated using the TCB already allocated for S. In effect, S and T share the same dynamic storage. T's state is set to *absorbed* as a consequence. A thread is absorbed if its thunk executes using the TCB associated with another running thread.

[1] Scheduled and delayed threads do not have an associated TCB.

Thus, a thread can run the code associated with another unevaluated thread whose value it requires using its own dynamic context – no TCB is allocated in this case and the accessing thread does not block. A consumer blocks only when the producer has already started evaluating. The producer is already associated with a TCB in this case, and no opportunity for absorption presents itself.

3.3 Thread Groups

Sting allows threads to be organized in ways that improve spatial locality. In particular, threads that share information among one another can be organized logically into groups. There are two kinds of groups supported in the system. Thread groups define a locus that programmers can use to clump related threads; such groups permit efficient sharing of data, but have no implications for thread placement or migration. A more elaborate group structure can be constructed using virtual topologies. Topologies are implicitly associated with a thread placement and scheduling discipline.

Sting provides *thread groups* as a means of gaining control over a related collection of threads[23]. A thread group is created by a call to `fork-thread-group`; this operation creates a new group and a new thread that becomes the *root thread* of that group. A child thread shares the same group as its parent unless it explicitly creates a new group. A thread group is includes a *shared heap* accessible to all its members. When a thread group terminates via the call,

```
(thread-group-terminate group),
```

all live threads in the group are terminated and its shared heap is garbage collected.

A thread group also contains debugging and thread operations that may be applied *en masse* to all of its members. Thread groups provide operations analogous to ordinary thread operations (*e.g.,* termination, suspension, etc.) as well as operations for debugging and monitoring (*e.g.,* listing all threads in a given group, listing all groups, profiling, genealogy information, etc.) Thus, when thread T is terminated, users can request all of T's children (which are defined to be part of T's group to be terminated) thus:

```
(thread-group-terminate (thread.group T))
```

Thread groups are an important tool for controlling sharing in a hierarchical memory architecture. Since objects shared by members in a group are contained in the group's shared heap, they are physically close to one another in virtual memory, and thus better spatial locality can be realized. Threads groups can also be used as a locus for scheduling. For example, a thread policy manager might implement a scheduling policy in which no thread in a group is allowed to run unless all threads in the group are allowed to run; this scheduling regime is similar to a "gang scheduling"[7] protocol.

4 Virtual Topologies

Sting programs can be parameterized over a virtual topology. A virtual topology defines a relation over a collection of Sting virtual processors; processor topologies configured as trees, graphs, hypercubes, and meshes are some well-known examples[31]. A virtual topology need not have any correlation with a physical one; it is intended to capture the interconnection structure best suited for a given algorithm. We consider a virtual processor to be an abstraction that defines scheduling, migration and load-balancing policies for the threads it executes. Thus, virtual topologies are intended to provide a simple and expressive high-level framework for defining complex thread/processor mappings that abstracts low-level details of a physical interconnection.

Efficient construction of virtual topologies and processor mappings lead to several important benefits; among the most important is improved communication and data locality:

1. *Improved Data Locality:* If a collection of threads share common data, we can construct a topology that maps the virtual processors on which these threads execute to the same physical processor. Virtual processors are multiplexed on physical processors in the same way threads are multiplexed on virtual processors.
2. *Improved Communication Locality:* If a collection of threads have significant communication requirements, we can construct a topology that maps threads which communicate with one another onto virtual processors close together in the virtual topology.

4.1 An Example

To help motivate the utility of virtual topologies, consider the program fragment shown in Figure 2.

`D&C` defines a parallel divide-and-conquer abstraction; when given a merge procedure M, and a list of data streams as its arguments, it constructs a binary tree that uses M to merge pairs of streams. Each pair of leaves in the tree are streams connected to a merge node that combines their contents and outputs the results onto a separate output stream. The root of the tree and all internal nodes are implemented as separate threads.

There is significant spatial locality in this program since data generated by child streams are accessed exclusively by their parents. Load-balancing is also an important consideration. We would like to avoid mapping internal nodes at the same depth to the same processor whenever possible since these nodes operate over distinct portions of the tree and have no data dependencies with one another. For example, a merge node N with children C_1 and C_2 ideally should be mapped onto a processor close to both C_1 and C_2, and should be mapped onto a processor distinct from any of its siblings.

```
(define (D&C merge streams)
  (let loop ((streams streams))        ;; iteration
    (future
      (let ((output-stream (make-stream)))
        (cond ((null? (cddr streams)) ;; even number of streams
               (merge (car streams) (cadr streams) output-stream))
              (else (let ((left  (left-half  streams))
                          (right (right-half streams))
                      merge results onto output stream
                      (merge (loop left)
                             (loop right)
                             output-stream))))))))
```

Fig. 2. A divide-and-conquer parallel program parameterized over a merge procedure.

4.2 A Topology Abstraction

To address the issues highlighted in the previous section, we define a high-level abstraction that allows programmers to specify how threads should be mapped onto a virtual topology corresponding to a logical process graph. In general, we expect programmers to use a library of topologies that are provided as part of a thread system; each topology in this library defines procedures for constructing and accessing its elements. The cognitive cost of using topologies is thus minimal for those topologies present in the topology library.

However, because virtual processors are first-class objects, building new topologies is not cumbersome. Programmers requiring topologies not available in the library can construct their own using the abstractions described in the next section. For example, a tree is an obvious candidate for the desired thread/processor map in the example shown in Figure 2. A tree, however, may not correspond to the physical topology of the system; virtual topologies provide a mechanism for bridging the gap between the algorithmic structure of an application and the physical structure of the machine on which it is to run. Mapping a tree onto a physical topology such as a bus or hypercube is defined by a topology that relates virtual processors to physical ones.

4.3 Virtual Processors

In instances where a desired virtual topology is not available, programmers can build their own using operations over virtual and physical processors:

– `(make-vp` *pp*`)` returns a new virtual processor mapped onto physical processor *pp*.

– `(current-vp)` returns the virtual processor on which this operation is evaluated.
– `(vp->address` *vp*`)` returns the address of *vp* in the topology of which it is a part.
– `(set-vp->address` *vp addr*`)` sets *vp*'s address in a virtual topology to be *addr*.
– `(vp->pp` *vp*`)` returns the physical processor on which *vp* is currently executing.

Note that running a new VP is tantamount to shifting the locus of control to a new node in a virtual topology.

4.4 Physical Processors

The second abstraction necessary to construct topologies are *physical processors*. Operations analogous to those available on virtual processors exist for physical ones.

A set of physical processors along with a specific physical topology form a *physical machine*. Each physical machine provides a set of topology dependent primitive functions to access physical processors. For example, a machine with a ring topology may provide: (a) `(current-pp)` which returns the physical physical processor on which this operation executes; (b) `(pp->address` *pp*`)` that returns the address in the physical topology to which `pp` is associated; (c) `(move-left` *i*`)` which returns the `pp` *i* steps to the left of `(current-pp)`; and, (d) `(move-right` *i*`)` which behaves analogously.

A topology object defines procedures for navigating on a particular topology, but we also require a mapping that translates addresses of virtual processors in a given virtual topology to addresses in the current physical topology. Given procedures for accessing virtual and physical processors, such mappings can be expressed using well-known techniques [11].

4.5 Example Revisited

We use topologies in parallel programs by allowing virtual processors to be provided as explicit arguments to thread creation operators. Consider the merge stream example shown in Figure 2. Using topologies, this program fragment could be rewritten as shown in Figure new-merge-streams.

The changes to the code from the original are slight, but lead to potentially significant gains in efficiency and expressivity. In fact, the only modification (outside of the structure of the outer loop which now binds a virtual processor to the root of a topology tree) is in the interface to `future`. Each application of `future` now takes a virtual processor as its argument that specifies where the thread

```
(define (D&C merge streams)
  (let loop  ((streams streams) ;; letrec
              (root (make-static-tree-topology (log (length streams)))))
    (future
       (let ((output-stream (make-stream)))
         (cond ((null? (cddr streams))
                (merge (car streams) (cadr streams) output-stream))
               (else (let ((left  (left-half streams))
                           (right (right-half streams)))
                        merge results onto output stream
                        (merge (loop left  (left-child))
                               (loop right (right-child))
                                output-stream)))))
        (tree-node-vp root)))))
```

Fig. 3. A divide-and-conquer procedure using virtual topologies.

created by the future should evaluate on the virtual topology; `tree-node-vp` returns the VP associated with the tree node passed as its argument. `Left-child` and `right-child` return the left and right child in the virtual topology created by `tree` .

4.6 Built-in Topologies And Their Interface Functions

We have constructed a number of topologies that reside as part of the Sting environment; some of these topologies along with their interfaces are enumerated in Fig. refbuilt-in.

5 Sting Benchmarks

The benchmarks shown in this section were implemented on an eight processor Silicon Graphics 75MHz MIPS-R3000 shared memory machine. The machine contains 256 MBytes of main memory and a one Mbyte unified secondary cache; each processor has a 64Kbyte data and a 64Kbyte instruction primary cache.

Fig. 5 gives baseline figures for various thread operations; these timings were derived using a single global FIFO queue.

Figure 6 gives some baseline costs for creating and managing virtual processors:

"Creating a VP" is the cost of creating, initializing, and adding a new virtual to a virtual machine; note that it is roughly the cost of creating a thread on a tree or vector processor topology. "Fork/Tree" is the cost of creating a thread on a static tree topology; "Fork/Vector" is the cost of creating a thread on a vector

Topology	*Interface Procedures*

Array	`(make-array-topology` *dimensions*`)`
	`(get-node` dim_1 dim_2 $\ldots dim_n$`)`
	`(move-up-in-nth-dimension)`
	`(move-down-in-nth-dimension)`

Ring	`(make-ring-topology` *size*`)`
	`(get-vp` *n*`)`
	`(move-right` *n*`)`
	`(move-left` *n*`)`

Static Tree
Dynamic Tree
	`(make-static-tree-topology` *depth*`)`
	`(make-dynamic-tree-topology` *depth*`)`
	`(get-root)`
	`(left-child)`
	`(right-child)`

Butterfly	`(make-butterfly-topology` *levels*`)`
	`(move-straight-left)`
	`(move-straight-right)`
	`(move-cross-left)`
	`(move-cross-right)`

Fig. 4. Some built-in topologies.

Case	*Timings(in μseconds)*
Thread Creation	40
Thread Fork and Value	86.9
Thread Enqueue/Dequeue	14.5
Synchronous Context Switch	12.8
Thread Block and Resume	27.9

Fig. 5. Baseline timings.

Operation	*Times* (μ-seconds)
Creating a VP	31
Adding a VP to a VM	5.4
Fork/Tree	25
Fork/Vector	20
VP Context Switch	68
Dynamic Tree	207

Fig. 6. Baseline times for managing virtual processors and threads.

topology. "Adding a VP to a VM" is the cost of updating a virtual machine's state with a new virtual processor. "VP Context Switch" is the cost of context-switching two virtual processors on the same physical processor; it also includes the cost of starting up a thread on the processor. "Dynamic Tree" is the cost of adding a new VP to a tree topology.

Fig. 7 shows benchmark times for four applications (solid lines indicate actual wallclock times; dashed lines indicate ideal performance relative to single processor times):

Benchmark	Processors			
	1	2	4	8
Alpha-Beta	108	57.6	31.9	16.7
N-Body	751	363	251	137
MST	81.8	43.5	24.3	13.7
Primes	219.5	112.6	58.1	34.1

Fig. 7. Wallclock times on benchmark suite.

1. **Primes:** This program computes the first million primes. It uses a master-slave algorithm; each slave computes all primes within a given range.
2. **Minimum Spanning Tree:** This program implements a geometric minimum spanning tree algorithm using a version of Prim's algorithm. The input data is a fully connected graph of 5000 points with edge lengths determined by Euclidean distance. The input space is divided among a fixed number of threads. To achieve better load balance, each node not in the tree does a parallel sort to find its minimum distance among all nodes in the tree. Although 46,766 threads are created with this input, only 16 TCBs are actually generated. This is again due to Sting's aggressive treatment of storage locality, reuse and context sharing.
3. **Alpha-Beta:** This program defines a parallel implementation of a game tree traversal algorithm. The program does α/β pruning[8, 14] on this tree to minimize the amount of search performed. The program creates a fixed number of threads; each of these threads communicate their α/β values via distributed data structures. The input used consisted of a tree of depth 10 with fanout 8; the depth cutoff for communicating α and β values was 3. To make the program realistic, we introduced a 90% skew in the input that favored finding the best node along the leftmost branch in the tree.
4. **N-body:** This program simulates the evolution of a system of bodies under the influence of gravitational forces. Each body is modeled as a point mass and exerts forces on all other bodies in the system. The simulation proceeds over time-steps, each step computing the net force on every body and thereby updating that body's position and other attributes. This benchmark ran the simulation on 5000 bodies over six time-steps.

Speedup and efficiency ratios for these problems are shown in Fig. 8.

Benchmark	Speedup	Efficiency
Primes	6.4	81
MST	5.9	75
N-Body	5.5	69
Alpha-Beta	6.5	81

Fig. 8. Efficiency and speedup rations on 8 processors.

We consider two application points to highlight the potential impact of virtual topologies in exploiting spatial locality (see Figure 9); in one of the applications (quicksort), topology mapping also had a beneficial impact on temporal locality, illustrating that spatial and temporal locality are *not* mutually distinct attributes of an application.

1. **Quicksort** sorts a vector of 2^{18} integers using a parallel divide-and-conquer strategy. We use a tree topology that dynamically allocates virtual processors; the maximum depth of the topology tree in this problem was 6; calls to `left-child` and `right-child` result in the creation of new virtual processors if required. The virtual to physical topology mapping chooses one child at random to be mapped onto the same physical processor as its parent in the virtual topology. Internal threads in the process tree generated by this program yield as their value a sorted sub-list; because of the topology mapping used, more opportunities for thread absorption [19], or task stealing [26] present themselves relative to an implementation that performs round-robin or random allocation of threads to processors. This is because at least one child thread is allocated on the same virtual processor as its parent; since the child will not execute before the parent unless migrated to another VP, it will be a strong target for dynamic inlining or thread absorption. The impact of improved temporal locality is evident in the times – on eight processors, the mapped version was roughly 1.4 times faster than the unmapped one; roughly 14% more threads were absorbed in the topology mapped version than in the unmapped case (201 threads absorbed vs. 171 absorbed with a total number of 239 threads created).
2. **Hamming** computes the extended hamming numbers up to 100000 for the first 16 primes. The extended hamming problem is defined thus: given a finite sequence of primes $A, B, C, D, \ldots$ and an integer n as input, output in increasing magnitude without duplication all integers less than or equal to n of the form

$$(A^i) \times (B^j) \times (C^k) \times (D^l) \times \ldots$$

This problem is structured in terms of a tree of mutually recursive threads that communicate via streams. The program example shown in Figure 2 captures the basic structure of this problem. There is a significant amount of communication or spatial locality exhibited by this program. A useful virtual topology for this example would be a static tree of depth equivalent to the depth of the process tree. By mapping siblings in the tree to different virtual processors, and selected children to the same processor as their parent, we can effectively exploit communication patterns evident in the algorithm. On eight processors, the implementation using topologies outperformed an unmapped round-robin scheduling policy by over a factor of six.

Offhand, it might appear that the benefits of virtual topologies would be nominal on a physical shared-memory machine since the cost of inter-processor communication is effectively the cost of accessing shared memory. However, even in this environment, virtual topologies can be used profitably. Spatial and temporal locality effects easily observable on a distributed memory system are modeled on a shared-memory machine in terms of memory locality. For example, programs can exhibit better spatial locality if communicating threads (or threads with manifest data dependencies) are mapped onto the same processor when possible, provided that such mappings will not lead to poor processor utilization. A system that supports fast creation and efficient management of threads and virtual processors would thus benefit from using virtual topologies regardless of the underlying physical topology.

Benchmark	Processors			
	1	2	4	8
Hamming	143.5	108.3	46	15.3
	140.4	118.24	100	96.5
Quicksort	8.57	4.51	2.45	1.7
	8.49	4.95	3.16	2.51

Fig. 9. Wallclock times for two benchmarks. A static tree topology of depth 5 was used for Hamming; a dynamic tree of depth 6 was used for Quicksort. The first row for each benchmark indicates processor times when virtual topologies were used; the second row indicates times using a default round-robin scheduler with no topology information.

6 Related Work

Besides providing an efficient multi-threaded programming system similar to other lightweight thread systems[4, 5, 24], Sting's liberal use of procedural and control abstractions provide added flexibility and important efficiency gains through improved temporal locality. Advanced memory management techniques are facilitated as a consequence of this design; TCB reuse, thread absorption, per-thread garabage collection all contributed to increased locality benefits. Sting is built on an abstract machine intended to support long-lived applications, persistent objects, and multiple address spaces. Thread packages provide none of this functionality since (by definition) they do not define a complete program environment.

Concurrent dialects of high-level languages such as Scheme [17, 23], ML [27, 29], or Concurrent Prolog [2, 32] typically encapsulate all scheduling, load-balancing and thread management decisions as part of a runtime kernel implementation; programmers have little control in exploiting locality and communication properties manifest in their applications. Consequently, such properties must often be inferred by a runtime system, since they cannot be specified by the programmer even when readily apparent. In contrast, Sting's virtual topology abstraction give programmers greater flexibilty to optimize programs based on the application's logical task structure.

Concurrent computing environments such as PVM [33] permit the construction of a type of virtual machine; such machines, however, consist of heavyweight Unix tasks that communicate exclusively via message-passing. In general, it is expected that there will be as many tasks in a PVM virtual machine as available processors in the ensemble.

Our work on virtual topologies is closely related to *para-functional* programming [15]. Para-functional languages extend implicitly parallel, lazy languages such as Haskell [16] with two annotations: *scheduling expressions* that provide user-control on evaluation order of expressions that would otherwise be evaluated lazily, and *mapping expressions* that permit programmers to map expressions onto distinct virtual processors.

While the goals of para-functional programming share much in common with ours, there are numerous differences in the technical development. First, we subsume the need for *scheduling expressions* by allowing programmers to create lightweight threads wherever concurrency is desired; thus, our programming model assumes explicit parallelism. Second, because VPs are closed over their own scheduling policy manager and thread queues, threads with different scheduling requirements can be mapped onto VPs that satisfy these requirements; in contrast, virtual processors in the para-functional model are simply integers. Third, there is no distinction between virtual and physical processors in a para-functional language; thus virtual to physical processor mappings cannot be expressed. Finally, para-functional languages, to our knowledge, have not been the focus of any implementation effort; by itself, the abstract model provides no insight into the efficiency impact of a given virtual topology in terms of increased data and communication locality, etc..

Finally, there has been much work in realizing improved temporal and spatial locality by sophisticated compile-time analysis [30, 36] or by explicit data distribution annotations [1]; the focus of these efforts has been on developing compiler optimizations for implicitly parallel languages. Outside of the fact that we assume explicit parallelism, our work is distinguished from these efforts insofar as virtual topologies require annotations on threads, not on the data used by them.

7 Static Analysis of Locality Properties

Despite various runtime optimizations to support locality, it is clear that without sophisticated compile-time support, parallel symbolic computing systems will be too inefficient to work well on large, realistic problems. To address this issue, we have investigated compile-time analysis techniques for parallel languages well-suited to run in an environment of the kind provided by Sting.

In this section, we informally introduce an analysis of a parallel language in which tasks communicate via first-class mutable shared locations. Details on the implementation and underlying theory of the analysis can be found in [21, 35].

7.1 The Language

Consider a simple language that could form the basis of a Scheme or SML[25] core. The language has constants, variables, functions, primitive applications, call-by-value function applications, conditionals, recursive function definitions, and a process creation operation. The primitives include operations for creating and accessing pairs and *shared locations*. Constants include integers and Booleans. (See Fig. 10.)

Communication in this language takes place through shared locations. We think of shared locations as an implementation substrate on top of which higher-level abstractions (*e.g.,* futures[12], tuple-spaces[18], etc.) may be constructed. Shared locations are created using the **mk-loc**operator, and are initially unbound. If x is a location, both **read**(x) and **remove**(x) return the value of x, blocking if x is unbound. In addition, if x is bound, **remove**(x) marks x as unbound. Blocked **remove** and **read** expresions may unblock when a **write** subsequently occurs on the location of interest. The value of a **write** expression is the location written. The act of writing or reading a location, or removing a location's contents is atomic.

To create a lightweight thread of control to evaluate e, we evaluate **spawn** e. **Spawn** returns immediately after creating a new thread; its value is TRUE. The environment in which a newly spawned thread evaluates is the same as its parent thread. Threads are completely asynchronous, and communicate exclusively via shared locations.

For example, the MultiLisp [12] expression, (**future** e) , is equivalent to,

$$
\begin{array}{ll}
e & \in Exp \\
c & \in Const = Int + Bool \\
x & \in Var \\
f & \in Func \\
p & \in Prim = \{\mathbf{cons}, \mathbf{car}, \mathbf{cdr}, \mathbf{mk\text{-}loc}, \mathbf{write}, \mathbf{read}, \mathbf{remove}, \ldots\}
\end{array}
$$

$$
\begin{array}{ll}
e ::= & c \\
& \mid x \\
& \mid f \\
& \mid p(e_1 \ldots\ e_n) \ \mid\ (p) \\
& \mid e_1\,(e_2\ e_3\ \ \ldots\ e_n) \ \mid\ (e) \\
& \mid \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \\
& \mid \mathbf{letrec}\ x_1 = f_1\ \ \ldots\ x_n = f_n\ \mathbf{in}\ e \\
& \mid \mathbf{spawn}\ e
\end{array}
$$

$$
f ::= \lambda x_1\ x_2\ \ \ldots\ x_n .\, e
$$

Fig. 10. The kernel language.

let *loc* = (**mk-loc**)
in begin
    spawn write(*loc*, *e*)
    loc
  end

and (**touch** v) is equivalent to,

if location? (v)
  then read(v)
  else v

(Note that "**let**" and "**begin**" are syntactic sugar for simple and nested application, respectively.)

As another example, the following expression implements a simple *fetch-and-op* abstraction [10] using locations. The procedure representing this abstraction atomically returns the current contents of the cell, and stores a new value using the procedure argument provided.

$$
\begin{array}{l}
\lambda\, init.\, \mathbf{let}\ \ cell\ =\ (\mathbf{mk\text{-}loc})\\
\qquad \mathbf{in\ begin}\\
\qquad\qquad \mathbf{write}(cell, init)\\
\qquad\qquad \lambda\, op.\, \mathbf{let}\ \ val\ =\ \mathbf{remove}(cell)\\
\qquad\qquad\qquad \mathbf{in\ begin}\\
\qquad\qquad\qquad\qquad \mathbf{write}(cell, op\,(val))\\
\qquad\qquad\qquad\qquad val\\
\qquad\qquad\qquad\quad \mathbf{end}\\
\qquad\quad \mathbf{end}
\end{array}
$$

7.2 The Analysis

With shared locations as the main synchronization vehicle, it is natural to focus on compile-time analyses targeted at tracking the movement of these locations through procedures, and across independently executing threads of control. Effective analysis of how and where shared locations are manipulated may lead to a number of interesting and important optimizations. We consider several in the next section.

Our approach to analyzing the behavior of shared locations is based on an abstract interpretation framework[6]. Operationally, abstract interpretation is concerned with the construction of an interpreter for a language that generates *abstract* or approximate values; this is in contrast to the behavior of an exact interpreter that generates exact values. We consider an interpretation framework that is based on a *labelled* transition system for an operational semantics. In this context, every expression is uniquely labelled.

Typically, an approximate value is a finite description of the set of (exact) values its generating expression may yield. Thus, one approximate value of the variable x in the following expression would be the set of (syntactic) **mk-loc** program points to whose dynamic instances x could be bound; the elements in this set correspond to the abstract values of the procedure's arguments:

$$\lambda\, x.\mathbf{read}(x)$$

Broadly speaking, we can regard an abstract interpreter as a constraint based evaluator. The abstract state is represented as a directed graph in which nodes correspond to expressions and edges represent the flow of data. Each node stores an abstract value which corresponds to the value of the expression at that node. Each edge can be viewed as a (subset) constraint on the abstract values stored at the nodes it connects. The evaluation of an expression can change the abstract value stored at a node, which in turn may violate a number of constraints (i.e., edges) emanating from that node. This may cause the evaluation of further expressions, the violation of further constraints, etc. as a further complication, because the input language contains data structures and higher-order procedures, edges may be added dynamically. The interpreter terminates when all constraints are satisfied.

In the presence of asynchronously executing threads, we have a variety of choices in choosing our approximations. For example, we might consider ignoring threads altogether in the abstract interpretation. In such a formulation, the value of an abstract location might be the label (*i.e.,* syntactic program point) from which its dynamic instances are created. Such an approximation chooses not to disambiguate different instances of a value based on the abstract threads which create them. Alternatively, we might choose to label an abstract location with both its syntactic program point, as well as the syntactic label of the **spawn** expression from which it was created. The latter analysis is clearly more accurate, but also incurs greater cost.

We have developed a framework[22] in which the accuracy and precision of such an analysis is parameterized along several dimensions:

1. *Contours:* A contour represents an abstraction of an activation frame or call string. The abstract value of an expression is determined in part by the contour within which it is evaluated. Thus, an interpretation that has only one contour effectively collapses all instances of a given (syntactic) expression. Alternatively, an interpretation that treats a contour as a call-string of length one effectively uses the current (syntactically apparent) application point to disambiguate the abstract values of expressions found in the procedure being applied from other calls made to the same procedure from different (syntactially apparent) call sites.
2. *Abstract binding environments:* The abstract values of free variables found in a procedure are determined via an abstract binding environment associated with the procedure. Abstract binding environments map variables to contours; given a contour, the abstract value of a variable is determined via an abstract store. As with contours, the choice of how binding environments are constructed influences the accuracy and cost of the analysis.
   Consider the expression,
   $$\lambda\ x.\lambda\ y.x$$
   The outer lambda may be instantiated at a number of different points in a program. However, the abstract value of x in the inner procedure depends on the definition of environment extension and creation. A refined environment extension function will map x in each instance of the inner procedure to its abstract value defined by the appropriate application of the outer procedure. A coarse environment extension function will join all abstract values of x in all instances of the outer procedure.
3. *Spawn Contours:* Contours help to abstract intra-thread control-flow; spawn contours help abstract inter-thread control-flow. In the presence of asynchronously executing threads, an abstract value is associated with a <*contour, spawn contour*> pair. A spawn contour, like a regular contour,

is represented as a sequence of labels, each element in the sequence corresponding to a spawn point.
For example, an interpretation that disregards spawn contours completely is not concerned with inter-thread control-flow. Values generated by a thread during its evaluation will be visible to other threads even if these values are used locally and are not accessible via shared data structures. A spawn contour of length one will disambiguate control-flow emanating from syntactially different occurrences of spawn expressions in a program, but will not disambiguate different instances generated from the same spawn point.
4. *Spawn Binding Environments:* The binding environment within which a thread begins execution is also a target for parameterization. One obvious choice for the environment in which a spawned thread evaluates is the environment of its parent. However, other possibilities also exist. For example, evaluating threads in an empty environment causes control-flow information to not cross thread boundaries; in this formulation, each thread effectively builds and maintains its own "copy" of relevant bindings.

7.3 An Example

To illustrate the utility of this analysis, consider the following program written in our kernel language. Both procedures and locations are used in higher-order contexts, thus making it non-trivial to understand the control-flow behavior of the program by mere textual examination of the source.

```
let f = λ ⟨q, r⟩.
          if null?( q)
          then λ ().r
          else spawn let  z = (read(q))
                      in ... z ...
in spawn let x = (mk-loc)
             g = write(x, (cons ( let v = (mk-loc)
                                      w = (mk-loc)
                                  in write(v, λ ().f( nil write(w, 0) )
                                 nil))
         in f( car(read(g)) (mk-loc) )
```

A simple (polynomial-time) instantiation of our interpretation framework can deduce several interesting properties of this program:

1. The argument q in f is always bound to either the constant **nil** or a location created by the **mk-loc** operation in the outer **spawn**.
2. The argument r in f is always bound to a location.
3. Location x is written and read only by threads created by the outer **spawn**.
4. The contents of x is a cons cell created by the **cons** operation in the outer **spawn**.

5. The contents of location v is the procedure,

   $$\lambda\,().f(\,\mathbf{nil}\;\mathbf{write}(y,0)\,)$$

6. The **car** of g is always bound to an instance of a location created by the inner **mk-loc** in the outer spawn, *i.e.,* the **mk-loc** operation bound to v.
7. Location v is written in the outer **spawn**, but only read in the inner one, *i.e.,* the **spawn** found in f.
8. The value of the **mk-loc** argument to the outer application of f is unbound.
9. The result of f is a procedure precisely when x is **nil**.

Based on such information, we can apply a number of important optimizations and heuristics related to locking, synchronization, type checks, and locality management:

1. No locks need to to be allocated for the location bound to f's second argument in f's outermost call.
2. Since all instances of the location bound to x are written and read by threads created by the outer **spawn**, a separate heap for x accessed exclusively by these threads can be created.
3. Since locations are never removed once written, no locks need to be acquired by readers when accessing a non-empty location.
4. The types of the contents of all locations are known, thus obviating the need to introduce runtime type checks when applying a strict operation (such as **car**) on a location's contents.
5. Since threads created by the inner **spawn** (defined in f) read locations created by the outer **spawn**, instances of these threads can be allocated close to another in a topology since share an explicit data dependency via location v.

Applying such an analysis may lead to significantly improved performance especially for programs that use higher-order procedures, or in which inter-thread control flow is non-trivial. Many parallel symbolic algorithms would likely be significiantly more efficient if they were the subject of such analyses.

Effective analysis of shared locations directly leads to improved locality. Both temporal and spatial locality are improved because tasks that never share locations need not be placed close to one another, or share common storage. Tasks that do are likely to exhibit improved performance if they reside close to another in a virtual topology, or if locations they share reside in a separate heap accessible only to these tasks.

Besides the optimizations listed above, one can imagine other optimizations more closely allied with a particular program methodology or abstraction. For example, touch elimination[9] is a potentially important optimization in programs that use *futures* exclusively for synchronization and task creation. In the absence of this optimization, strict operations must explicitly check their arguments to see if they are bound to a placeholder object of the kind yielded by *future* evaluation. Touch elimination, much like runtime type elimination, eliminates such checks when it can be determined that the arguments are not placeholders. It is easy to show that an abstract interpretation technique of the kind described here can be used to implement touch elimination operations.

Acknowledgments

Sting was designed jointly with James Philbin, who is responsible for its shared-memory implementation. Work on compile-time analysis techniques for higher-order parallel languages is joint with Stephen Weeks.

References

1. Marina Chen, Young-il Choo, and Jingke Li. Compiling Parallel Programs by Optimizing Performance. *Journal of Supercomputing*, 1(2):171–207, 1988.
2. K.L Clark and S. Gregory. PARLOG: Parallel Programming in Logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, 1986.
3. William Clinger and Jonathan Rees, editors. Revised^4 Report on the Algorithmic Language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.
4. Eric Cooper and Richard Draves. C Threads. Technical Report CMU-CS-88-154, Carnegie-Mellon University, June.
5. Eric Cooper and J.Gregory Morrisett. Adding Threads to Standard ML. Technical Report CMU-CS-90-186, Carnegie-Mellon University, 1990.
6. Patrick Cousot. Semantic Foundations of Program Analysis. In *Program Flow Analysis: Theory and Foundation*, pages 303–342. Prentice-Hall, 1981.
7. J. Dongarra, D. Sorenson, and P. Brewer. Tools and Methodology for Programming Parallel Processors. In *Aspects of Computation on Asynchronous Processors*, pages 125–138. North-Holland, 1988.
8. Raphael Finkel and John Fishburn. Parallelism in Alpha-Beta Search. *Artificial Intelligence*, 19(1):89–106, 1982.
9. Cormac Flanagan and Matthias Felleisen. The Semantics of Future and Its Use in Program Optimization. In *ACM 22nd Annual Symposium on Principles of Programming Languages*, January 1995.
10. Allan Gottlieb, B. Lubachevsky, and Larry Rudolph. Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.
11. David Saks Greenberg. *Full Utilization of Communication Resources*. PhD thesis, Yale University, June 1991.
12. Robert Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

13. Christopher Haynes and Daniel Friedman. Embedding Continuations in Procedural Objects. *ACM Transactions on Programming Languages and Systems*, 9(4):582–598, 1987.
14. Feng hsiung Hsu. *Large Scale Parallelization of Alpha-Beta Search: An Algorithmic and Architectural Study with Computer Chess.* PhD thesis, Carnegie-Mellon University, 1990. Published as Technical Report CMU-CS-90-108.
15. Paul Hudak. Para-functional Programming in Haskell. In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers.* ACM Press, 1991.
16. Paul Hudak *et.al.* Report on the Functional Programming Language Haskell, Version 1.2. *ACM SIGPLAN Notices*, May 1992.
17. Takayasu Ito and Robert Halstead, Jr., editors. *Parallel Lisp: Languages and Systems.* Springer-Verlag, 1989. LNCS number 41.
18. Suresh Jagannathan. TS/Scheme: Distributed Data Structures in Lisp. *Lisp and Symbolic Computation*, 7(2):283–305, 1994.
19. Suresh Jagannathan and James Philbin. A Customizable Substrate for Concurrent Languages. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1992.
20. Suresh Jagannathan and James Philbin. A Foundation for an Efficient Multi-Threaded Scheme System. In *Proceedings of the 1992 Conf. on Lisp and Functional Programming*, June 1992.
21. Suresh Jagannathan and Stephen Weeks. Analyzing Stores and References in a Parallel Symbolic Language. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 294–306, 1994.
22. Suresh Jagannathan and Stephen Weeks. A Unified Treatment of Flow Analysis in Higher-Order Languages. In *ACM 22nd Annual Symposium on Principles of Programming Languages*, January 1995.
23. David Kranz, Robert Halstead, and Eric Mohr. Mul-T: A High Performance Parallel Lisp. In *Proceedings of the ACM Symposium on Programming Language Design and Implementation*, pages 81–91, June 1989.
24. Sun Microsystems. *Lightweight Processes*, 1990. In SunOS Programming Utilities and Libraries.
25. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.
26. Rick Mohr, David Kranz, and Robert Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, June 1990.
27. J. Gregory Morrisett and Andrew Tolmach. Procs and Locks: A Portable Multiprocessing Platform for Standard ML of New Jersey. In *Fourth ACM Symposium on Principles and Practice of Parallel Programming*, pages 198–207, 1993.
28. James Philbin. *An Operating System for Modern Languages.* PhD thesis, Dept. of Computer Science, Yale University, 1993.
29. John Reppy. CML: A Higher-Order Concurrent Language. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 293–306, June 1991.
30. Anne Rogers and Keshave Pingali. Process Decomposition Through Locality of Reference. In *SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 69–80, 1989.
31. Karsten Schwan and Win Bo. "Topologies" – Distributed Objects on Multicomputers. *ACM Transactions on Computer Systems*, 8(2):111–157, 1990.

32. Ehud Shapiro, editor. *Concurrent Prolog Collected Papers.* MIT Press, Cambridge, Mass., 1987.
33. V.S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice & Experience*, 2(4), 1990.
34. A. Tevanian, R. Rashid, D. Golub, D. Black, E. Cooper, and M. Young. Mach Treads and the UNIX Kernel: The Battle for Control. In *1987 USENIX Summer Conference*, pages 185–197, 1987.
35. Stephen Weeks, Suresh Jagannathan, and James Philbin. A Concurrent Abstract Interpreter. *Lisp and Symbolic Computation*, 7(2):171–191, 1994.
36. Michael Wolfe. *Optimizing Supercompilers for SuperComputers.* MIT Press, 1989.