Virtual Topologies: A New Concurrency Abstraction for High-Level Parallel Languages

(Preliminary Report)

James Philbin[1], Suresh Jagannathan[1], Rajiv Mirani[2]

[1] Computer Science Division, NEC Research Institute, 4 Independence Way, {philbin|suresh}@research.nj.nec.com
[2] Department of Computer Science, Yale University, New Haven, CT mirani@cs.yale.edu

Abstract. We present a new concurrency abstraction and implementation technique for high-level (symbolic) parallel languages that allows significant programmer control over load-balancing and mapping of fine-grained lightweight threads. Central to our proposal is the notion of a *virtual topology*. A virtual topology defines a relation over a collection of *virtual processors*, and a mapping of those processors to a set of physical processors; processor topologies configured as trees, graphs, butterflies, and meshes are some well-known examples. 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, expressive and efficient high-level framework for defining complex thread/processor mappings that abstracts low-level details of a physical interconnection.

1 Introduction

Abstractions used by designers of parallel algorithms [12] are very different from those available to implementors of these algorithms. High-level parallel languages [11, 19, 15] often provide no mechanism for programmers to specify the mapping, scheduling and locality properties of a computation; policy decisions of this kind are usually hard-wired as part of the language implementation. The efficiency of a parallel algorithm, however, is typically dictated precisely by how well it optimizes these concerns. Moreover, the abstract process graph manipulated by an algorithm is often dissimilar to the physical topology of the machine on which implementations of the algorithm execute; in general, there is no well-defined, machine independent mechanism to reconcile these differences.

In this paper, we present a new concurrency abstraction and implementation technique for high-level (symbolic) parallel languages that address these issues. Central to our proposal is the notion of a *virtual topology*. A virtual topology defines a relation over a collection of *virtual processors*; processor topologies configured as trees, graphs, hypercubes, and meshes are some well-known examples. 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 over a physical processor 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.

There need be no *a priori* correlation between the number of virtual processors in a topology, and the number of physical processors in the machine on which the computation will run. Because virtual processors are first-class, virtual topologies can be specified on a per-application basis. Threads created by a computation are mapped to processors in the virtual topology via mapping functions associated with the topology. These mapping functions are user definable.

Virtual topologies also permit significant performance gains for many high-level parallel programs. Our benchmark results indicate that using virtual topologies *even on physically shared-memory machines* can yield performance benefits ranging from 1.5 to over a factor of 6 relative to an identical system that uses a simple round-robin or random thread scheduling policy. The performance gains are due primarily to improved locality and load-balancing. Thus, virtual topologies are not only an expressive language abstraction, but also an important efficiency tool. To our knowledge, this is the first implementation of virtual topologies that seriously addresses efficiency and programmability on real multiprocessor platforms.

A high-level parallel program is thus parameterized over a virtual topology. Because programs no longer need to rely on details of physical processors and their interconnection, code becomes machine independent and highly portable. All concerns related to thread mapping and locality are abstracted in the specification of the virtual topology used by the program, and the manner in which nodes in the topology are traversed during program execution.

2 Implications

Using virtual topologies leads to several important benefits; we enumerate them here, and elaborate in the following sections:

1. *Portability:* Implementations of parallel algorithms need not be specialized for different *physical* topologies. An implementation is defined only with respect to a given *virtual* topology.
2. *Control over Thread Placement:* Since the mapping algorithm used to associate threads with processors is specified as part of a virtual topology, programmers have fine control over how threads should be mapped to virtual processors.
3. *Load Balancing:* If the computation requirements of the threads generated by a program are known in advance, the ability to explicitly allocate these threads onto specific virtual processors can lead to better load balancing than an implicit mapping strategy.

4. *Exploiting Algorithmic Structure:* The structure of a control and dataflow graph defined by a parallel algorithm can be exploited in a number of ways:
   (a) *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.
   (b) *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 near each other in the virtual topology.
   (c) *Improved Thread Granularity:* If thread T_1 has a data dependency with the value yielded by another thread T_2, we can map T_1 and T_2 on the same virtual processor. In fine-grained programs where processors are busy most of the time, the ability to schedule data-dependent threads on the same processor leads to opportunities for improved thread granularity [10, 14, 23].
5. *Dynamic Topologies:* Certain algorithms have a process structure that unfolds as the computation progresses; adaptive tree algorithms [12] are a good example. These algorithms are best executed on topologies that permit dynamic creation and destruction of virtual processors.

The remainder of the paper is structured as follows. Section 3 describes a simple parallel program used to motivate the use of virtual topologies. Section 4 introduces virtual topologies. Section 5 describes virtual and physical processors. Section 6 reformulates the example given in Section 3 to use virtual topologies. Section 7 gives a concrete specification of a virtual tree topology. Benchmark results are given in Section 8. Conclusions and comparison to related work are presented in Section 9.

Our discussion and examples use Scheme [4] as the base language, but virtual topologies and first-class virtual processors are language independent abstractions.

3 An Example

To help motivate the utility of virtual topologies, consider the program fragment shown in Figure 1. This program is written in a parallel dialect of Scheme that uses *futures* [6] to create parallel threads of control. The expression, `(future` E`)`, returns a future object. Associated with this future is a thread responsible for computing E. When the thread finishes, it stores the result of E in the future. The future is then said to be *determined*. An expression that *touches* a future either blocks if E is still being computed or yields v if the future is determined.

`D&C` defines a parallel divide-and-conquer procedure; when given a merge procedure M, and a list of data streams as its arguments, it constructs a binary tree that uses M to

```
(define (D&C merge streams)
  (let loop ((streams streams))
    (future
      (if (null? (cdr s))
          streams
          (let ((output-stream (make-stream)))
            (merge (loop (left-half streams))
                   (loop (right-half streams))
                   output-stream)))))))
```

Fig. 1. A divide-and-conquer parallel program parameterized over a merge procedure.

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 data and communication 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.

4 Virtual Topologies

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. Figure 2 enumerates some of the topologies we have implemented, along with their interface procedures. 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 1. 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

Topology	*Interface Procedures*
Array	(make-array-topology *dimensions*) (get-node dim_1 dim_2 $\ldots dim_n$) (move-up-in-nth-dimension n) (move-down-in-nth-dimension n)
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. 2. Interfaces for built-in topologies.

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.

The implementation of a virtual tree topology on a physical ring as implemented in our system is given in Section 7. Figure 3 illustrates such a virtual topology.

5 The Abstractions

5.1 Virtual Processors

In instances where a desired virtual topology is not available, programmers can build their own using two basic abstractions. The first is a *virtual processor*. A virtual processor is a first-class object that abstracts policy decisions related to thread scheduling, mapping, and migration. Threads execute on a virtual processor. There may be many threads mapped to the same processor, and no constraints are imposed by the abstraction on how these threads are scheduled. There are five simple operations on virtual 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.

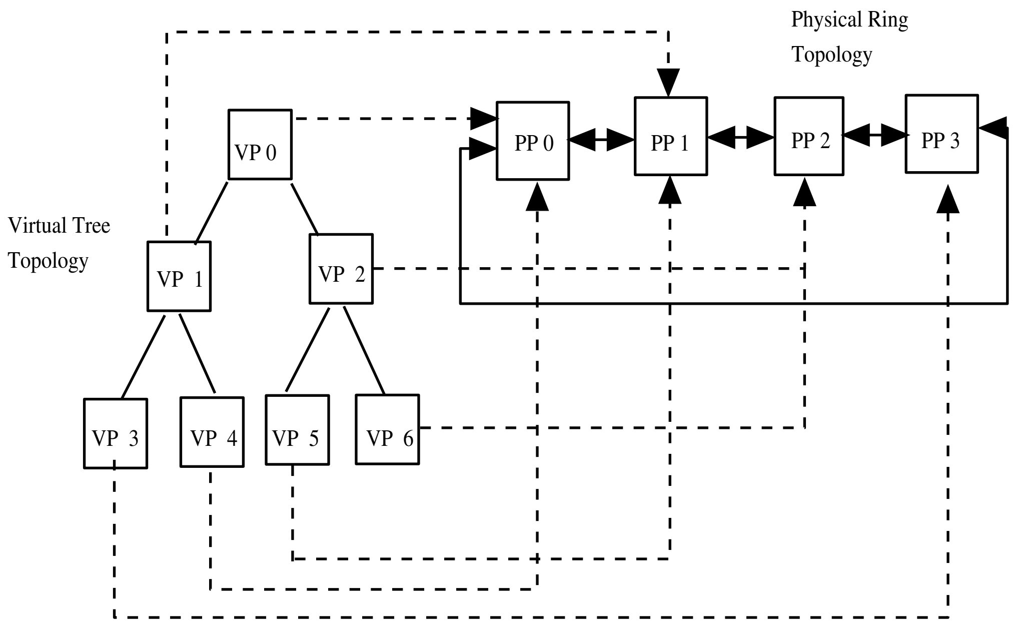

Fig. 3. A virtual topology that embeds a tree onto a ring.

`(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*.

Each virtual processor implements a very simple control logic: it repeatedly removes a thread from its thread ready queue for evaluation or, if the queue is empty, calls a policy procedure. This procedure may choose to migrate a new thread from some other virtual processor, or initiate a context-switch to the underlying physical processor (PP). The PP may at this point choose to run any other VP in its VP ready queue, migrate VPs from other processors, or perform bookkeeping operations. Running a new VP is tantamount to shifting the locus of control to a new node in a virtual topology.

Notice that two levels of context-switching takes place in this design. Threads context-switch among VPs either because of preemption or because of synchronization constraints imposed by the application. VPs context switch among PPs to improve load-balancing and overall efficency; for example, a VP will choose to initiate a context-switch if its thread queue is empty, thereby allowing threads resident on another (currently idle) VP to run.

5.2 Physical Processors

The second abstraction necessary to construct topologies are *physical processors*. A physical processor is a software abstraction of a physical computing device. Each of these physical processors exist in a physical topology that reflects an underlying hardware interconnection structure. 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. Given procedures for accessing virtual and physical processors, topology mappings can be expressed using well-known techniques [5, 12].

6 Example Revisited

```
(define (D&C merge streams)
  (let* ((depth (log (length streams)))
         (root  (make-static-tree-topology depth)))
    (let loop  ((streams streams)
                (node (current-node)))
      (future
        (if (null? (cdr s))
            streams
            (let ((output-stream (make-stream)))
              (merge (loop (left-half streams) (left-child))
                     (loop (right-half streams) (right-child))
                     output-stream)))
        node))))
```

Fig. 4. A divide-and-conquer procedure using virtual topologies.

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 1. Using topologies, this program fragment could be rewritten as shown in Figure 4.

The changes to the code from the original are shown in italics. Although the changes all small, they lead to potentially significant gains in efficiency and expressivity. Each application of a `future` now takes as its argument a virtual processor that specifies where the thread created by the future should evaluate on the virtual topology. `Left-child` and `right-child` return the left and right child of the current VP in the virtual topology created by `tree`.

7 A Virtual Tree Topology

The code shown in Figure 5 defines procedures for constructing a static tree topology of some depth. A physical topology accessor procedure is passed as an argument to `child` in `make-tree`; the definition of these accessors is given in Figure 6. The call to `make-tree-node` returns a tree node structure that contains fields for traversing trees; the `define-structure` special form provides accessor and setter procedures on structure fields.

The `left-child` accessor procedure returns the left child of the virtual processor on which this operation is executed. A call to a tree accessor procedure performed by some thread whose virtual processor V is mapped to a leaf node in this topology simply returns V.

7.1 Mapping Trees onto Rings

The code shown in Figure 6 demonstrates how a tree virtual topology may be embedded in a ring physical topology. Given a virtual processor `vp`, `pp-map-left-child` returns the physical processor on which `vp`'s left child should execute.

8 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 MB of main memory and a one MB unified secondary cache; each processor has a 64KB data and a 64KB instruction primary cache.

Offhand, it might appear that the benefits of virtual topologies would be nominal on a 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. Communication 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 cache behavior 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.

8.1 Baseline Costs

Since there may be many virtual processors executing on a single physical one, implementations of virtual topologies must ensure that the cost of creating and managing

```
(define-structure node
  vp
  left-child
  right-child)

(define (make&init-node vp)
  (let ((node (make-node)))
    (set-node-vp node vp)
    (set-node-left-child! node nil)
    (set-node-right-child! node nil)
    (set-vp->address vp node)
    node))

(define (make-tree depth)
  (let ((root (make&init-node (make-vp (current-pp)) 0)))
    (let loop ((node root)     ;; local recursion
               (cur-depth 0))
      (cond ((= cur-depth depth) node)
            (else
             (let* ((child
                      (lambda (pp-mapper)
                        (make&init-node
                          (make-vp (pp-mapper (node-vp node))))))
                    (left  (loop (child pp-map-left-child)
                                 (+ cur-depth 1)))
                    (right (loop (child pp-map-right-child)
                                 (+ cur-depth 1))))
               (set-node-left-child!  node left)
               (set-node-right-child! node right)))))))

(define (left-child)
  (node-left-child (vp->address (current-vp))))
(define (right-child)
  (node-right-child (vp->address (current-vp))))
```

Fig. 5. The specification of a tree topology.

virtual processors on a physical one is small. In the implementation described here, the cost of creating a virtual processor is roughly comparable to the cost of creating a thread; a VP is represented by a data structure whose size is approximately 20 long words. Encapsulated in this data structure are fields containing the physical processor to which this VP is mapped, and references to exception handlers, scheduler procedures, and various thread queues.

Figure 7 gives some baseline costs for creating and managing virtual processors. Our implementation of virtual topologies was built as part of Sting [10, 16] intended to serve as a high-level operating system for high-level programming languages such as

```
(define (pp-map-left-child vp)
  (let* ((index (vp->address vp))
         (pos (+ (* 2 index) 1))
         (shift (mod pos num-pps)))
    (if (> pos num-pps)
        (move-left shift)
        (move-right shift))))

(define (pp-map-right-child vp)
  (let* ((index (vp->address vp))
         (pos (+ (* 2 index) 2))
         (shift (mod pos num-pps)))
    (if (> pos num-pps)
        (move-right shift)
        (move-left shift))))
```

Fig. 6. Embedding a virtual tree onto a physical ring.

Scheme [4], ML [13], or Haskell [8].

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. 7. Baseline times for managing virtual processors and threads.

"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. "Adding a VP to a VM" is the cost of updating a virtual machine's state with a new virtual processor. "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 static vector topology. "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 dynamic tree topology.

8.2 Applications

We consider three benchmarks to highlight various aspects of our design. All times derived using a particular topology mapping are compared relative to a round-robin global FIFO scheduler on a simple processor configuration in which there is a one-to-one map between virtual and physical processors. All benchmarks were structured so as not to trigger garbage collection. All times are given in seconds.

Details of the benchmarks are given below. Since all these benchmarks were executed on a physically shared-memory platform, we expect the performance benefits to increase significantly on disjoint memory architectures.

1. **MinMax.** The topology of a program cannot always be determined statically. For many algorithms, the process structure is dependent on the computation itself. Virtual topologies are well suited to these problems since virtual processors are dynamic objects: they may be created at run time, assigned to a physical processor and garbage collected when no longer needed. In addition, virtual topologies can be reconfigured dynamically by moving virtual processors from one physical processor to another.

   As an example, consider the minmax algorithm for game trees. Typically, the depth to which a branch of the game tree is explored depends on the perceived potential of the game positions along that branch. Thus, the process tree is skewed and its structure cannot be determined statically.

   A naive implementation of this algorithm on a complete tree topology may result in poor performance. For example, one may choose to map all virtual processors in a subtree below a certain depth to the same physical processor (arguing that this gives the best locality) and then find that most of the computation takes place along one long branch in that subtree. An alternate method would delay the creation of the topology until the computation requires allocating a thread on some portion of the topology that does not yet exist. At this point, the topology can be extended with the addition of a new virtual processor. Since more information is available when this happens, better load balancing may result. It is worth emphasizing that, since virtual processors are very light weight, the cost of dynamic manipulation is small. To test this functionality, we measured the performance of a static topology mapping versus a dynamic one using a tree of depth 150 with varying cutoff depths on 8 processors; these cutoff depths indicate the point at which no new virtual processors are added:

Cutoff	2	3	4	5	6	7	8
Static	5.88	5.87	5.76	5.71	5.67	5.71	5.78
Dynamic	5.92	6.05	5.69	5.40	5.26	4.96	5.15

   At a depth cutoff of 7, for example, minimax performed approximately 14% faster using a dynamic topology than a static one. Using a simple round-robin scheduler, this program took 10.23 seconds to execute on 8 processors, roughly 50% slower than the dynamic tree topology.

2. **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 [10], or task stealing [14] 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 thread absorption. The impact of improved cache locality and greater absorption is evident in the wallclock times shown below – 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).

	Processors			
	1	2	4	8
Dynamic Tree (depth 6)	8.57	4.51	2.45	1.7
Round-Robin	8.49	4.95	3.16	2.51

3. **Hamming** computes the extended hamming numbers up to 10000 for the first 16 primes. The extended hamming problem is defined thus: given a finite sequence of primes *A, B, C, D,* ... 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 1 captures the basic structure of this problem. There is a significant amount of communication 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:

	Processors			
	1	2	4	8
Static Tree (Depth 5)	143.5	108.3	46	15.3
Round-Robin	140.4	118.24	100	96.5

9 Conclusions and Comparison to Related Work

Concurrent dialects of high-level languages such as Scheme [9, 11], ML [15, 17], or Concurrent Prolog [3, 21] typically encapsulate all scheduling, load-balancing and thread management decisions as part of a runtime kernel implementation. Consequently, locality and communication properties of an algorithm must often be inferred by a runtime system, since they cannot be specified by the programmer even when readily apparent.

Concurrent computing environments such as PVM [22] 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. Systems such as Linda [1] that provide operations for dynamically creating and synchronizing threads provide no high-level mechanism to specify how threads should be load-balanced or mapped onto processors since virtual topologies are not part of their abstract machine model.

Our work is closely related to *para-functional* programming [7]. Para-functional languages extend implicitly parallel, lazy languages such as Haskell [8] 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.

Schwan and Bo[20] describe a topology abstraction for programming distributed memory machines. A topology in their system "represents a distributed object as an arbitrary communication graph with nodes that contain the object's distributed representation and execute the object's operations." While the motivation for topologies is similar to ours, the designs of the two systems are very different. Our topologies are first-class objects built using virtual processors. As a result, virtual topologies can be dynamically reconfigured or extended. Furthermore, the implementation of topologies is given via first-class procedures, rather than in terms of a separate abstract data type representation. Finally, our primary goal in defining virtual topologies was to exploit communciation and data locality, and to provide mechanism for customization of diverse scheduling policies. In contrast, the system described in [20] focuses on synchronization, message-passing, and global communication abstractions.

Finally, there has been much work in realizing improved data locality by sophisticated compile-time analysis [18, 24] or by explicit data distribution annotations [2]; 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. Deriving efficient data distributions is straightforward given a topology that provides a relation over these threads.

While our results are still preliminary, they support our hypothesis that topology mapping and first-class processors will have significant benefit in optimizing communication and data locality for many high-level parallel applications.

References

1. Nick Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444 – 458, April 1989.
2. Marina Chen, Young-il Choo, and Jingke Li. Compiling Parallel Programs by Optimizing Performance. *Journal of Supercomputing*, 1(2):171–207, 1988.
3. K.L Clark and S. Gregory. PARLOG: Parallel Programming in Logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, 1986.
4. William Clinger and Jonathan Rees, editors. Revised^4 Report on the Algorithmic Language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.
5. David Saks Greenberg. *Full Utilization of Communication Resources*. PhD thesis, Yale University, June 1991.
6. Robert Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.
7. Paul Hudak. Para-functional Programming in Haskell. In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*. ACM Press, 1991.
8. Paul Hudak *et.al.* Report on the Functional Programming Language Haskell, Version 1.2. *ACM SIGPLAN Notices*, May 1992.
9. Takayasu Ito and Robert Halstead, Jr., editors. *Parallel Lisp: Languages and Systems*. Springer-Verlag, 1989. LNCS number 41.
10. Suresh Jagannathan and James Philbin. A Customizable Substrate for Concurrent Languages. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1992.
11. 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.
12. F. Thomas Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan-Kaufmann, 1992.
13. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

14. 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.

15. 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.

16. James Philbin. *An Operating System for Modern Languages*. PhD thesis, Dept. of Computer Science, Yale University, 1993.

17. 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.

18. 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.

19. Vijay Saraswat and Martin Rinard. Concurrent Constraint Programming. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 232–246, 1990.

20. Karsten Schwan and Win Bo. Topologies – Distributed Objects on Multicomputers. *ACM Transactions on Computer Systems*, 8(2):111–157, May 1990.

21. Ehud Shapiro, editor. *Concurrent Prolog Collected Papers*. MIT Press, Cambridge, Mass., 1987.

22. V.S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice & Experience*, 2(4), 1990.

23. M. Vandevoorde and E. Roberts. WorkCrews: An Abstraction for Controlling Parallelism. *International Journal of Parallel Programming*, 17(4):347–366, August 1988.

24. Michael Wolfe. *Optimizing Supercompilers for SuperComputers*. MIT Press, 1989.

This article was processed using the LaTeX macro package with LLNCS style