# Expressing Fine-Grained Parallelism Using Distributed Data Structures

Suresh Jagannathan
NEC Research Institute
Princeton, NJ 08540
suresh@research.nec.com

**Abstract**

Despite the fact that distributed data structures have long been advocated as a powerful tool for specifying communication and synchronization constraints in explicitly parallel languages, their utility in expressing fine-grained parallelism has thus far been limited. In this paper, we consider semantic analysis techniques and runtime support mechanisms that can be used to make distributed data structures an efficient device in this regard.

We consider a compiler/runtime system for a higher-order expression-based parallel language supporting distributed data structures that relies on (1) a type inference system to answer questions regarding how shared distributed objects should be represented and partitioned, and (2) a high-level runtime kernel to effectively manage and schedule dynamically instantiated lightweight threads of control. We argue that a tightly integrated system of this kind can be used to efficiently implement a variety of fine-grained parallel applications.

## 1 Introduction

Efficient support for fine-grained parallelism on stock hardware has been the focus of much recent research interest[1, 18]. The realization of this goal involves the design of linguistic devices for expressing concurrency, compile-time analysis techniques for constructing efficient implementations of shared data objects, and a runtime support system for managing lightweight threads. In this paper, we examine one specific approach to realizing fine-grained parallelism based on this general strategy. Our starting point is an explicitly parallel language that relies on *distributed data structures* for expressing synchronization and communication constraints. We argue that given a toolbox of compile-time analyses for optimizing the representation of these structures, and a robust runtime system tailored for managing dynamically instantiated lightweight processes, fine-grained parallel applications can be made to run efficiently on stock multiprocessor hardware.

A distributed data structure (also referred to as a *distributed object*) is a device that serves as a communication and synchronization repository for a collection of asynchronously executing processes. Explicitly parallel languages such as FCP[16], Concurrent Smalltalk[10, 12], C-Linda[6], MultiLisp[11] etc. permit many producers and consumers simultaneously to modify and read the contents of shared distributed objects. Consumers that access a component of such an object block until a producer provides a value. Advocates argue that programming with distributed data structures leads naturally to a methodology in

*Appeared in the Proceedings of the the 1991 Workshop on Research Directions in High-Level Parallel Programming Languages. Published as Springer-Verlag LNCS 574.

which low-level process synchronization concerns are abstracted to high-level algorithmic design issues involving data structure access and generation.

On the surface, one would expect that the flexibility provided by distributed data structure-based systems would make them an ideal device for expressing fine-grained parallelism. This has not been the case, however, for two reasons. First, the generality afforded by the semantics of most distributed object proposals makes generating efficient low-level representations difficult in the absence of advanced compile-time analysis or specialized hardware[2, 10]. No system to our knowledge has seriously pursued the former approach; the latter alternative suffers from lack of portability and high cost. Consequently, many languages that implement some form of distributed data structure do so by requiring user annotations to aid the compiler in generating a more optimal representation (*e.g.,* [3, 8, 14]). Requiring user annotations compromises abstraction, however; to take advantage of any annotation-based optimizations, users must be aware of the underlying machine topology and often require knowledge of the compiler implementation. Second, explicitly parallel languages such as C-Linda or FCP typically rely on low-level OS kernel support for process management and scheduling. The fact that synchronization and context switching is managed by the kernel means that fine-grained parallel programs written in these languages incur significant performance penalties; all scheduling decisions and thread management schemes must be implemented outside the purview of dedicated runtime system.

In this paper, we consider compile-time analysis techniques and runtime support mechanisms that offer the possibility of making distributed data structures a viable tool for expressing fine-grained parallelism. To answer questions regarding how shared distributed objects should be represented and partitioned, we define a suitably structured type inference system; the "type" of a distributed data structure contains information regarding its patterns of usage. The runtime kernel effectively manages and schedules dynamically instantiated lightweight threads of control at user level; the design of the runtime system makes it easy to control the amount of concurrency generated, and to dynamically load-balance available work among a processor ensemble.

The type system permits the construction of customized implementations of first-class distributed data structures even in the presence of higher-order functions. The manner in which structures are shared, the kinds of operations performed on a distributed object, the input-output behavior of these objects, and the communication structure of processes are all attributes that can be derived by analysis of a distributed data structure's type. The representation chosen for such structures reflects its use – thus, different instantiations of the same (high-level) structure may entail different (low-level) implementations.

The scheduling policies of the runtime system permit users to define a "lazy" process evaluation strategy. In the context of fine-grained parallelism, this permits users to create many more processes than exist physical processors *without* inducing any performance degradation. Processes are evaluated only if resources are available, or if their results are required. In the latter case, the process may be executed within the same context as the accessor; no separate thread of control need be created here. In addition to lazy scheduling, users can specify various priority schemes for evaluating threads; the priority scheme chosen is influenced by information gleaned from the type system.

## 2 The Context

Our discussion is based on a semantics for distributed data structures that is a significant generalization of Linda's tuple-space abstraction[6]. The sequential core of the kernel language (called $\mathcal{TS}^k$) used in this paper forms a higher-order strongly-typed strict functional language.

Tuple-spaces in $\mathcal{TS}^k$ are first-class objects. By way of introduction, a first-class tuple-space is a synchronization repository that may be bound to variables, passed as an argument to (or returned as a result from) functions, or embedded within data structures or tuples. In the abstract, a tuple-space defines

$$
\begin{array}{lll}
\mathrm{E} & ::= & \texttt{true} \mid \texttt{false} \mid c \mid x \mid s \mid \\
 & & (\lambda\ (x_1\ x_2 \ldots x_n)\ \mathrm{E}) \mid (\mathrm{E}_1\ \mathrm{E}_2 \ldots \mathrm{E}_n) \mid \\
 & & \texttt{rec}\ x_1 = E_1,\ x_2 = E_2\ \ldots\ x_n = E_n\ \texttt{in}\ E \mid \\
 & & (\texttt{begin}\ E_1;\ E_2;\ \ldots;E_n) \mid \\
 & & [\ \ ] \mid [\mathrm{E}:\mathrm{E}] \mid (\texttt{hd}\ \mathrm{E}) \mid (\texttt{tl}\ \mathrm{E}) \mid \\
 & & \mathrm{E}_b \rightarrow \mathrm{E}_t\ ;\ \mathrm{E}_f \mid \\
 & & (\texttt{make-ts}) \mid \\
 & & (\texttt{put}\ \mathrm{E}\ (\mathrm{E}_1, \mathrm{E}_2, \ldots, \mathrm{E}_n)) \mid \\
 & & (\texttt{spawn}\ \mathrm{E}\ (\mathrm{E}_1, \mathrm{E}_2, \ldots, \mathrm{E}_n)) \mid \\
 & & (\texttt{rd}\ \mathrm{E}\ (\mathrm{T}_1, \mathrm{T}_2, \ldots, \mathrm{T}_n)\ \mathrm{E}_b) \mid \\
 & & (\texttt{get}\ \mathrm{E}\ (\mathrm{T}_1, \mathrm{T}_2, \ldots, \mathrm{T}_n)\ \mathrm{E}_b) \\
 & & \\
\mathrm{T} & ::= & \mathrm{E} \mid ?x
\end{array}
$$

Figure 1: Grammar for $\mathcal{TS}^k$.

a content-addressable shared associative memory. Its elements are ordered set of heterogeneous typed values or processes known as *tuples*. Fields in tuples can be retrieved using a simple pattern-matching procedure; failure to procedure a satisfactory substitution causes the initiating process to block. Such a process may resume only after a matching tuple is deposited into the specified tuple-space. The $\mathcal{TS}^k$ type system constrains all elements found in a given tuple-space to have the same length and type.

The paper is organized as follows. The next section describes the kernel language and provides further motivation for the problems addressed. Section 5 gives an overview of the type system and details how it can be used to optimize the representation of tuple-spaces. Section 7 provides a description of the runtime kernel on which $\mathcal{TS}^k$ runs; we develop some scheduling policies for $\mathcal{TS}^k$ processes that can be used to effectively realize fine-grained parallelism.

# 3 The Language

$\mathcal{TS}^k$ is a higher-order parallel programming language whose grammar is shown in figure 1. Its sequential core forms a simple higher-order lexically-scoped functional language with abstraction ($\lambda$), recursion (`letrec`), conditionals ($\rightarrow$), sequencing (`begin`) and lists (:). $\mathcal{TS}^k$ also supports five operations for creating and manipulating tuple-spaces.

## Creating a Tuple-Space

The expression, `(make-ts)`, returns a reference to a new tuple-space object[2]. Since `(make-ts)` yields a reference to a tuple-space (and not the tuple-space itself), it follows that a tuple-space can never directly contain another tuple-space object. The semantics of tuple-spaces in this regard resembles the treatment of complex data structures in Lisp and other heap-based languages.

[2]Throughout this paper, we will blur the distinction between a tuple-space object and its reference when clear from context.

## Generating New Tuples

There are two operators that are used to deposit tuples. Given expressions $e_1, \ldots, e_n$ and an expression $E_t$ that yields a tuple-space object $T$, the operation

`(put` $E_t$ `(`$e_1$`,`$e_2$`,`$\ldots$`,`$e_n$`))`

evaluates (in the current evaluation environment) $E_t$ to get $T$ and evaluates $e_1$, $e_2$,...,$e_n$ to get values $v_1$, $v_2$,..., $v_n$. A value is either a constant of base type (*e.g.,* an integer, Boolean or string), a list whose elements are all values, a function closure, or a tuple-space object. The tuple yielded by replacing each $e_i$ with $v_i$ is deposited into $T$ The value of `put` is unspecified; it is executed for effect, not value.

`Spawn` is the *non-strict* counterpart of `put`. Evaluating the expression

`(spawn` $E_t$ `(`$e_1$ $e_2$ $\ldots$ $e_n$`))`

deposits a tuple containing a process for each of the $e_i$; the processes found in this "active" tuple are evaluated in arbitrary order. The issue of whether processes are evaluated lazily or eagerly is left unspecified.

Processes in a `spawn`-generated tuple can communicate only via the tuple-spaces accessible to them. This is the sole form of inter-process communication available in the language – processes do not transmit information by mutating non-tuple-space shared variables.

## Retrieving Tuples

`Get` and `rd` are used to remove and read elements from a tuple-space. Consider the expression:

`(rd` $E_t$ `(`$e_1$`,` $e_2$`,` $\ldots$`,` $e_n$`)` *body*`)`

Unlike tuple-generator expressions, each of the $e_i$ may be either an ordinary $\mathcal{TS}^k$ expression *or* an identifier prefixed by a "?"; such an identifier is referred to as a *formal.* The evaluation of the above expression takes place in three steps. First, $E_t$ is evaluated to yield a tuple-space object $T$. Second, every $\mathcal{TS}^k$ expression is evaluated to a value. The resulting object is known as a *tuple-template*: it consists of values and formals.

This template is pattern-matched against the passive tuples found in $T$. A successful pattern-match results in each of the "?"-prefixed identifiers in the template obtaining as their binding-value the value of the corresponding field in the tuple to which this template was matched. We define the notion of a *match* in figure 2.

The bindings associated with these formals are used in the evaluation of *body.* The value of the expression is the value yielded by *body.* Free references in *body* not defined by any of the formals are resolved relative to the `get` or `rd`'s lexical environment.

If the pattern-matching operation fails, the executing `rd` or `get` blocks. The evaluation of the expression resumes when a matching passive tuple is deposited into the requisite tuple-space. If there are several tuples that match a template, only one is chosen (non-deterministically) to establish the binding-values of the template's formals.

The `get` operation is semantically identical to `rd` except that the tuple chosen for the match is removed from the given tuple-space.

**Definition 1** Let $t$ be a tuple-template with non-formals at fields $i_1, i_2, \ldots, i_k$ and formals at fields $j_1, j_2, j_l$ and let $t'$ be a tuple.

Let $v(i, tuple)$ denote the value of the $i^{th}$ field in $tuple$, and let $\tau(j, tuple)$ denote the type of the $i^{th}$ field in $tuple$.

Then, $t$ *matches* with $t'$ if

1. if $t$ and $t'$ are of the same length.
2. $v(i_m, t) = v(i_m, t')$, $1 \leq\ m \leq\ k$.
3. $\tau(j_n, t) = \tau(j_n, t')$, $1 \leq\ n \leq\ l$.

Figure 2: Definition of a tuple match in $\mathcal{TS}^k$. Value equality for lists and functions are presumed to be "eq"-based in the Lisp sense: list and abstraction expressions evaluate to references to the corresponding list or closure. Testing equality of these objects is via simple pointer comparison[15].

# 4 Motivation

First-class tuple-spaces are a flexible tool for specifying parallelism, but this flexibility comes at a price. In a naive implementation, matching a tuple template against the tuples found within a tuple-space $T$ would require an exhaustive search of $T$'s contents. The cost of such an operation obviously limits the utility of tuple-spaces for expressing fine-grained concurrency. Tuple-spaces containing a large number of elements that are accessed frequently by many processes would pose a significant bottleneck since each tuple-space access would require searching a large database; the latencies induced by such an operation would be intolerable.

Carriero[7] describes an approach to limit the overhead of tuple-space access in C-Linda that involves constructing partitions of the global tuple-space object; this construction is based on an examination of the structure and type of tuples found within a C-Linda program. Tuple structure is derived by syntactic analysis, and type information is gleaned from program declarations. Tuple operations whose tuple arguments are type and length consonant get placed within the same partition; the representation of individual partitions is therefore customized to the structure of the tuples that occupy it.

Partitioning of this kind is unnecessary in $\mathcal{TS}^k$ since all components of a given tuple-space must have the same type signature; the length of the tuples occupying a tuple-space $T$ contribute to $T$'s type. On the other hand, the fact that tuple-spaces are now denotable (first-class) objects embedded within a higher-order language means that sharing and aliasing analysis is more complicated; simple textual examination of a program will not reveal all operations performed on a given tuple-space object. The inference system given below provides a basis for implementing these analyses in purely type-theoretic terms.

The type system gives information on the structural attributes of tuples and tuple-spaces, but it does not address important runtime issues such as dynamic process scheduling or protocols for fast context switching. We take the position that these issues are best tackled at runtime by a robust kernel that permits fast creation of lightweight processes, and whose data structures are chosen to make context switching cheap. One way that process creation costs are minimized is by deferring the evaluation of lightweight threads of control. Allocating a thread[3] does not involve a system trap, or initial allocation of storage. Deferred or "lazy" threads can execute within the context of other processes. Furthermore,

[3] A thread is the system representation of a process.

```
make-cell =
  (λ (initial-value)
       let cell = (make-ts)
       in
        (begin
           (put cell (initial-value));
           cell))

 read-cell =
  (λ (cell)
       (rd cell (?v) v))

 write-cell =
  (λ (cell v)
      (get cell (?old-v)
           (put cell (v))))
```

Figure 3: A cell abstraction using first-class tuple-spaces. A cell contains exactly one element – before a new element is written, the old element is removed. The `let` form used in `make-cell` is syntactic sugar for application.

since storage for these threads are acquired only when they are run, cache and page locality is improved. Optimizations of this sort on runtime data structures are designed to realize efficient fine-grained parallelism using conventional hardware. The compiler and runtime system are closely integrated; the compiler suggests priority schemes for managing threads within a given $TS$ program based on process communication patterns it detects via type inference.

To motivate these techniques, we consider two simple examples. The first is a $\mathcal{TS}^k$ implementation of a memory cell (shown in figure 3); the second is a fine-grained parallel matrix multiplication procedure (shown in figure 4).

In the case of the memory cell abstraction, the following information can be deduced by the type system described in Section 5:

1. A cell contains only tuples of length one.
2. Cells that are read but never written do have not have any `get` operations performed upon them, and thus do not need read locks.
3. There are no constant-valued fields in a cell.
4. There are no processes that execute within the tuple-representation of a cell.

Flow analysis on this program would further reveal that every `put` operation either occurs after a `get` or after a `(make-ts)`; furthermore, a `rd` never precedes a `put`. Thus, a cell can never contain more than one element and can never be read while empty.

Using this information, we can construct a suitable representation for the cell abstraction. Cell instances can be implemented as a single word object; write operations require the cell to be locked, whereas no locks are necessary on reads. In the absence of any structural or flow information on the tuples inhabiting a tuple-space, the representation must be chosen with no assumptions on the kind of operations performed, or the structure of the tuples read and deposited.

The matrix multiplication code provides a simple and elegant solution to the problem at hand; unfortunately, conventional hardware and compilation techniques would not do a very good job in running applications of this procedure fast. Most (efficient) parallel implementations of this problem often use a more complex program structure (*e.g.,* master-slave[5]) in which the number of processes generated is

```
matrix_multiply =
  (λ (A B m n k)
      let C = (make-ts)
          build-vectors =
            (λ (A-row B-col)
                  rec loop = (λ (i dot)
                                  i > n →
                                    dot;
                                    (loop (1+ i)
                                          (+ dot (* (rd A (A-row i ?a) a)
                                                    (rd B (i B-col ?b) b)))))
                  in (loop 1 0))

       in rec loop = (λ (i)
                          i > m → C;
                          (begin
                             rec innerloop = (λ (j)
                                                 j <= k →
                                                 (begin
                                                    (spawn C (i j (build-vectors i j)))
                                                    (innerloop (+ j 1))))
                              in (innerloop 1);
                              (loop (+ 1 i))))
           in (loop 1))
```

Figure 4: A fine-grained parallel matrix multiplication routine. The function takes two matrices with dimension `m x n` and `n x k` resp. and computes their product. Matrices are represented as tuple-spaces whose elements are triples, `i,j,v`, where `(i,j)` denotes a matrix index, and `v` is the value found at that index.

manually determined by the programmer, rather than automatically inferred by the compiler/runtime system. Elegance and simplicity are compromised for the sake of efficiency.

The type inference system described below will deduce that $A$, $B$, and $C$ should be implemented in terms of a matrix object with suitable locks associated with its elements (*e.g.,* if the result matrix is never mutated by `put` or `get` operations, it need not be locked before read). However, the inference system can make no judgments regarding optimal scheduling policies for the processes generated within `innerloop`. In a naive system where each `spawn` generated process creates an active thread of control, performance may be significantly compromised – many processes may compete for the same limited resources, reducing throughput and increasing contention. It should be clear that clever runtime support of the kind described earlier is necessary to implement this procedure efficiently.

Consider, therefore, the following implementation scheme for `spawn`. Each tuple field $t$ found in a `spawn` operation evaluates to a *lazy thread.* Such a thread contains a thunk that when applied, evaluates $t$; in addition, the thread also contains information about $t$'s dynamic and exception environment. No thread storage is allocated, but the resulting tuple (containing process-value fields) is deposited within the tuple-space.

A tuple operation that reads a tuple $t$ containing lazy threads initiates the evaluation of all fields in the tuple necessary to determine if a successful pattern-match exists *provided* that the thread has not already been chosen for execution by another process or by a VP. Having one process evaluate the code associated with another is known as *stealing.* Stealing a thread does not require the allocation of a new process context; new processes can run within the same context as the processes that demand their value.

Idle processors can execute any runnable thread (lazy or eager). The number of executing processes created is bounded by the number of processors; thread creation is throttled because threads are executable (*i.e.,* storage is allocated for them) only when an idle processor exists or when another thread requires its value.

Other implementations are conceivable that evaluate processes in a more eager fashion. We discuss some of these alternative strategies in Section 7. The choice of scheduling policy depends on the application; in the runtime system envisioned, programmers have flexibility in specifying the policy implementation of their choice.

# 5 The Type System

In order to statically determine the structure of a tuple-space, we define a type system that associates every $\mathcal{TS}^k$ expression with a type; notably, the type of a tuple-space will reflect the structural characteristics of the tuples which inhabit it. Thus, to determine properties of tuple-spaces useful in building a customized representation, we examine their types.

Our type language is given by the grammar shown in figure 5. (To simplify the presentation, we assume that $\mathcal{TS}^k$ functions are written in a curried style.)

We let $\sigma$ range over type variables. It is useful to distinguish between type variables that get mapped to tuple-space types and those that get mapped to other types; we let $\alpha$ range over tuple-space type variables (TSvars), and we use $\beta$ to denote type variables for other types. $\phi$ denotes the "null" type used to type `put` and `spawn` expressions, and $\iota$ ranges over the set of primitive types. Besides tuple-space and tuple types, there are two other type constructors in our type language: $\rightarrow$ builds function types, and **List** constructs list types.

For now, we consider all types to be monomorphic; we briefly discuss the interaction of polymorphic types and tuple-spaces at the end of this section.

$$
\begin{array}{lll}
\tau & ::= & \sigma \mid \iota \mid \phi \mid \tau \rightarrow \tau \mid \mathbf{List}[\tau] \mid \gamma \\
\sigma & ::= & \alpha \mid \beta \\
\gamma & ::= & (\delta, \mathcal{K}, \mathcal{C}, \mathcal{F}, \mathcal{A}) \\
\delta & ::= & \tau \mid \delta \times \delta \\
\mathcal{K} & ::= & \mathcal{P}(\text{READ, WRITE, REMOVE, EVAL}) \\
\mathcal{C} & ::= & \mathcal{C}_\perp \mid \mathcal{C}[(\mathbf{N}, \mathrm{E})] \\
\mathcal{F} & ::= & \mathcal{F}_\perp \mid \mathcal{F}[\mathbf{N}] \\
\mathcal{A} & ::= & \mathcal{A}_\perp \mid \mathcal{A}[\mathbf{N}]
\end{array}
$$

Figure 5: The Type Language

## 5.1 Tuple-Space and Tuple Types

Let $t$ be a tuple-space and let $\gamma$ be its type. Then, $t$'s type structure captures the following attributes:

1. $\delta$: the type of the tuples which inhabit $t$. A tuple-type is a cross product over types such that the $i^{th}$ component in the cross product corresponds to the type of the $i^{th}$ field in the tuples found within $t$.
2. $\mathcal{K}$ : the powerset over the *kind* of tuple operations performed over $t$.
3. $\mathcal{C}$ : the tuple-space attribute that captures the indices and values of manifest constants found in tuple operations. $\mathcal{C}_\perp$ signifies a tuple-space which has no tuple operation performed on it containing manifest constants; a tuple-space type with $\mathcal{C}$ component:

$$\mathcal{C}_\perp[(i_1, E_{i_1})][(i_2, E_{i_2})] \ldots [(i_k, E_{i_k})]$$

   denotes a tuple-space that has tuple operations performed upon it such that these operations read or construct tuples with manifest constants $E_j$ in field indices $j$, $1 \leq j \leq k$.

4. $\mathcal{F}_\perp$: the tuple-space attribute that defines the indices of formals occurring in `rd` and `get` operations performed on $t$; a tuple-space type with $\mathcal{F}$ component:

$$\mathcal{F}_\perp[i_1][i_2] \ldots [i_k]$$

   denotes a tuple-space that has `rd` and `get` operations performed upon it whose tuple-template arguments have formals occurring in positions $i_1, i_2, \ldots, i_k$.

5. $\mathcal{A}$ defines the indices of actuals (as opposed to constants) found in tuple arguments of tuple operations performed on $t$.

If $\gamma$ is a tuple-space type of the form:

$$(\tau, K, C, F, A)$$

then $\gamma \downarrow \delta$ yields $\tau$, $\gamma \downarrow \mathcal{K}$ yields $K$ and so forth.

The structure of a tuple-space is determined collectively by the set of tuple expressions that operate on it. To express this fact within the type system, we introduce an ordering relation ($\leq$) on tuple-space types:

**Definition 2** *Let $\gamma_1$ and $\gamma_2$ be tuple-space types. Then, $\gamma_1 \leq \gamma_2$ iff:*

1. $\gamma_1 \downarrow \delta = \gamma_2 \downarrow \delta$.
2. $\gamma_2 \downarrow \mathcal{K} \subseteq \gamma_2 \downarrow \mathcal{K}$.
3. $\gamma_2 \downarrow \mathcal{F} = \mathcal{F}_\perp[i_1][i_2]\ldots[i_m]$, $\gamma_1 \downarrow \mathcal{F} = \mathcal{F}_\perp[j_1][j_2]\ldots[j_n]$ *and for every* $i_k$, $1 \leq k \leq m$ *there exists a* $j_l$, $1 \leq l \leq n$.
4. $\gamma_2 \downarrow \mathcal{A} = \mathcal{A}_\perp[i_1][i_2]\ldots[i_m]$, $\gamma_1 \downarrow \mathcal{A} = \mathcal{A}_\perp[j_1][j_2]\ldots[j_n]$ *and for every* $i_k$, $1 \leq k \leq m$ *there exists a* $j_l$, $1 \leq l \leq n$.

## 5.2 An Inference System

We define an inference system that associates a type expression with every $\mathcal{TS}^k$ syntactic construct. In presenting the rules, we use $A$ to represent the current type environment; a type environment is a map from variables to types. We write:

$$A \vdash e : \tau$$

to indicate that $\mathcal{TS}^k$ expression $e$ has type $\tau$ given the type bindings defined by $A$.

The *extension* of $A$ by $x$ with type $\tau$ (written $A[x \mapsto \tau]$) defines a new type environment as follows:

$$A[x \mapsto \tau](y) = \begin{cases} A(y) & \text{if } y \neq x \\ \tau & \text{otherwise} \end{cases}$$

(Type Variable Instantiation)

$$\frac{A \vdash e : \tau}{A \vdash e : \tau[\sigma/\tau_1]}$$

(Subtype)

$$\frac{A \vdash e : \gamma}{A \vdash e : \gamma'} \quad \gamma \leq \gamma'$$

(Identifiers)

$$A \vdash x : A(x)$$

(Abstraction)

$$\frac{A[x \mapsto \tau] \vdash e_b : \tau_b}{A \vdash (\lambda\ (x)\ e_b) : \tau \to \tau_b}$$

(Application)

$$\frac{\begin{array}{c} A \vdash e_1 : \tau_1 \to \tau_f \\ A \vdash e_2 : \tau_1 \end{array}}{A \vdash (e_1\ e_2) : \tau_f}$$

(Recursion)

$$\frac{\begin{array}{c} A[x \mapsto \tau] \vdash e_1 : \tau \\ A[x \mapsto \tau] \vdash e_b : \tau_b \end{array}}{A \vdash \texttt{rec}\ x = e_1\ \texttt{in}\ e_b : \tau_b}$$

(Conditional)

$$\frac{\begin{array}{c} A \vdash e_b : \textbf{Boolean} \\ A \vdash e_t : \tau \\ A \vdash e_f : \tau \end{array}}{A \vdash e_b \to e_t\ ;\ e_f : \tau}$$

(List)

$$\frac{\begin{array}{c} A \vdash e_1 : \tau \\ A \vdash e_2 : \textbf{List}[\tau] \end{array}}{A \vdash [e_1 : e_2] : \textbf{List}[\tau]}$$

(Tuple-Space)

$$A \vdash (\texttt{make} - \texttt{ts}) : \gamma$$

(Generate)

To type tuple generator expressions, we assume a tuple-space type $\gamma$ whose tuple-type component reflects the attributes of the tuple being deposited. Note that the subtype rule effectively allows any tuple-space expression with type $\gamma'$ to be well-typed in this context provided that $\gamma' \leq \gamma$. The auxiliary functions, *Constants* and *Actuals* return type attributes for the constants and actuals appearing in `put`'s tuple argument.

$$\frac{\begin{array}{c} E = (e_1, e_2, \ldots, e_n) \\ A \vdash e_i : \tau_i, 1 \leq i \leq n \\ t = (\tau_1, \tau_2, \ldots, \tau_n) \\ \delta = \tau_1 \times \tau_2 \times \ldots \times \tau_n \\ \gamma = (\delta, \textsc{write}, Constants(E, t), \mathcal{F}_\perp, Actuals(E)) \\ A \vdash e_T : \gamma \end{array}}{A \vdash (\texttt{put}\ e_T\ (e_1, e_2, \ldots, e_n)) : \phi}$$

(Read)

The type rule for `rd` and `in` expressions is similar to those for `put` and `spawn` except that the body of the expression is evaluated relative to a binding environment that maps each formal in the tuple-template to its type.

$$\frac{\begin{array}{c} E = (e_1, e_2, \ldots, e_n) \\ A \vdash\ e_i : \tau_i, 1 \leq\ i \leq\ n \\ t = (\tau_1, \tau_2, \ldots, \tau_n) \\ \delta = \tau_1 \times \tau_2 \times \ldots \times \tau_n \\ x_1, x_2, \ldots, x_j \text{ be the formals in E} \\ \gamma = (\delta, \text{READ}, Constants(E,t), Formals(E,t), Actuals(E,t)) \\ A \vdash\ e_T : \gamma \\ A[x_k \mapsto \tau_k] \vdash\ e_b : \tau \qquad\qquad j = 1 \leq\ k \leq\ j \end{array}}{A \vdash\ (\texttt{rd}\ e_T\ (e_1, e_2, \ldots, e_n) e_b) : \tau}$$

### 5.3 Polymorphism

Generalizing the type system to support polymorphic types is straightforward *provided* that TSVars are not quantifiable. The primary goal of the inference system is to collect information on patterns of tuple usage; however, in the presence of a polymorphic operator (such as an ML-style `let` expression), this ability would be compromised. To see why, consider the fragment:

```
let x = (make-ts)
in  (begin
       (f x);
       (g x))
```

where `let`-bound variables are assumed generic. Given an ML type semantics in which `(make-ts)` yields a polymorphic type variable, the two instances of `x` that occur within the `let` body get associated with different monomorphic types derived from the type signature of `f` and `g` resp. Thus, tuple operations performed on this object by `f` and `g` will be reflected in these corresponding type instances, but will not be combined. The type ultimately assigned to `x` will simply be a universally quantified type variable that contains no concrete structural information on the operations performed upon it by `f` and `g`. Restricting tuple-space variables to be monomorphic eliminates this problem.

## 6 From Types to Implementation

We can use type information to generate efficient compiled code. The representation chosen for a tuple-space is a reflection of the structure of the operations performed; the evaluation strategy used to schedule $\mathcal{TS}^k$ processes is predicated (partially) on the kinds of tuple-space operations executed by these processes.

For example, a tuple-space that has elements added to it, but never read or removed need not be shared among processors; a `spawn` operation executed for effect, not value is a case in point.

`Spawn` operations performed on a tuple-space $T$ can be compiled to take advantage of the structure of operations on $T$ that read and remove tuples. Thus, if all `rd` expressions on $T$ are of the form:

```
(rd T (x,?y) y)
```

it is reasonable to defer the evaluation of $E_2$, and to proceed with the evaluation of $E_1$ in the expression:

(spawn $T$ ($E_1$, $E_2$))

This is because the value of the formal in the `rd` operation is required only if the value of the first field (`x`) matches the value yielded by $E_1$. Insofar as the type system gives information about which fields serve directly in the matching process, and which fields do not, it can be used as a tool that provides a first approximation for determining process priorities.

Consider another simple priority scheme derived from type analysis. In this scheme, the order in which processes are evaluated depends on the kind of tuple operations they execute. Processes that do not manipulate or access tuple-space have higher priority than those that do. A process that makes no use of tuple-spaces is guaranteed not to block. Similarly, processes that manipulate tuple-spaces only via `put` and `spawn` never block while executing these operations; these processes should have higher priority than those that read or remove tuple-space elements. The type system associates each tuple space with a type; associated with each potential process is a union of the types of all tuple-spaces it accesses. This union is used to associate a priority number with the process.

We outline below some other optimizations that can be derived from information gleaned through type analysis.

*Input*: A tuple-space $T$ with type $(\delta, K, C, F, A)$.

Constant Elimination:

If $F = \mathcal{F}_\perp$ and $A = \mathcal{A}_\perp$, and $C$ contains a set of constants, we associate each constant with a general semaphore. `Put` operations perform a $V$ operation on the corresponding semaphore; `get` or `rd` expressions perform a $P$ operation.

Search Key Reduction:

Tuple-space that have non-empty $\mathcal{F}$ components are associated with a set of *search keys.* A search key is any index $i$ such that $\mathcal{F}$ is not of the form:

$$\mathcal{F}_\perp \ldots [i] \ldots$$

The set of search keys for a given tuple-space define exactly those fields that are involved in a match.

If the search keys for a given tuple-space denote tuple indices whose contents are all constants (with each search key associated with a homogeneous type), the tuple-space can be implemented as an $n$ dimensional matrix where $n$ denotes the number of search keys for the tuple-space. The structure of this matrix is given by allocating a dimension for each tuple index corresponding to a search key; the contents of the $n^{th}$ dimension contains a parallel queue to hold the binding-values for the formals found in `rd` or `get` expressions that access this tuple-space. A distributed array (*e.g.,* the matrices defined in the matrix multiplication example) or a homogeneous stream would satisfy these constraints.

Lock Elimination

Tuple-spaces that are read, but not removed need not have read locks. Such tuple-spaces also can be replicated; `put` operations entail a broadcast, but `rd` operations can use the local image of the tuple-space. This is a valid optimization since there are no global synchronization constraints imposed within this programming model. In applications that are predominately read-only, the ability to replicate tuple-spaces may yield significant performance benefits.

Hash Key Construction:

In the general case, a tuple-space contains a set of search key whose corresponding tuple-field contents are not known statically (*i.e.,* the search keys are not constants). The contents of the tuple-fields represented by the search keys are used here as keys in a generalized hashing procedure. A `put` expression is implemented by hashing the contents of each search key in its tuple argument to a pair containing the tuple and the search key. A `get` or `rd` expression computes the intersection of the set of tuples returned by hashing on each of the search key arguments. If this intersection is empty, it means that no matching tuple has yet been generated; if the intersection set contains more than one element, it means that several matching tuples have been deposited. Any tuple can be chosen from this set non-deterministically.

# 7 Runtime Support

The inference system described in the previous section has its limitations – while it can be used to transform amorphous tuple-spaces into well-structured distributed data objects, it does not implement or manage lightweight processes. To address this concern, we have developed a runtime kernel built on top of Scheme[15] called SOS that we intend to use as a high-level operating system kernel for implementing fine-grained parallel programs written in $\mathcal{TS}^k$ and related dialects. Processes are created in $\mathcal{TS}^k$ via the `spawn` operation; in the abstract, this operator creates a separate thread of control for each tuple element in its tuple argument. The runtime kernel is responsible for managing the creation and scheduling of these threads. $\mathcal{TS}^k$ operations on tuple-spaces are implemented in terms of SOS operations on threads. The detailed design of SOS is described elsewhere[13]; we list the salient properties of the system relevant to our discussion below:

1. Threads are treated as a Scheme abstraction with a well-defined representation. As a result, most thread management routines are expressed in terms of ordinary Scheme procedures. The implementation relies heavily on Scheme's efficient support for higher-order lexically-scoped procedures, and makes few demands on the underlying OS kernel. Because threads are integrated into the base language, it is straightforward to customize or extend their implementation; unlike thread packages built on top of particular operating systems (such as CThreads/Mach[9, 17] or FastThreads/Topaz[1, 4, 18]) our implementation makes *no* assumptions on the availability or efficiency of particular OS services. Thus, there is no interface specification between our augmented Scheme system and the underlying operating system that must be upheld in order to effect any modifications to the thread abstraction.

2. The design provides flexibility to define new scheduling policies (*e.g.,* FIFO, LIFO, round-robin, priority, realtime, etc.). The SOS scheduler is implemented by a virtual processor (VP) that defines the high-level kernel for the thread system. A VP runs on top of a lightweight OS-managed (low-level kernel) thread. Each VP runs a dedicated scheduler responsible for providing thread-related services. All thread management structures used by the scheduler are represented in terms of Scheme objects. The heart of the scheduler is defined by a straight-line code procedure that allocates no storage.

3. Thread creation is cheap since SOS permits deferred or "lazy" evaluation of thread objects. Allocating a thread does not involve trapping into the system kernel, nor does it require initial allocation of storage. It is straightforward to implement storage allocation policies for threads in which stack and heap allocation is delayed until the thread is actually scheduled to run; the ability to express such a policy permits possible reuse of storage created by previously executed threads. This strategy lead to better cache and page utilization. It also allows users to experiment with a variety of different execution regimes – from *fully delayed* to *completely eager* evaluation.

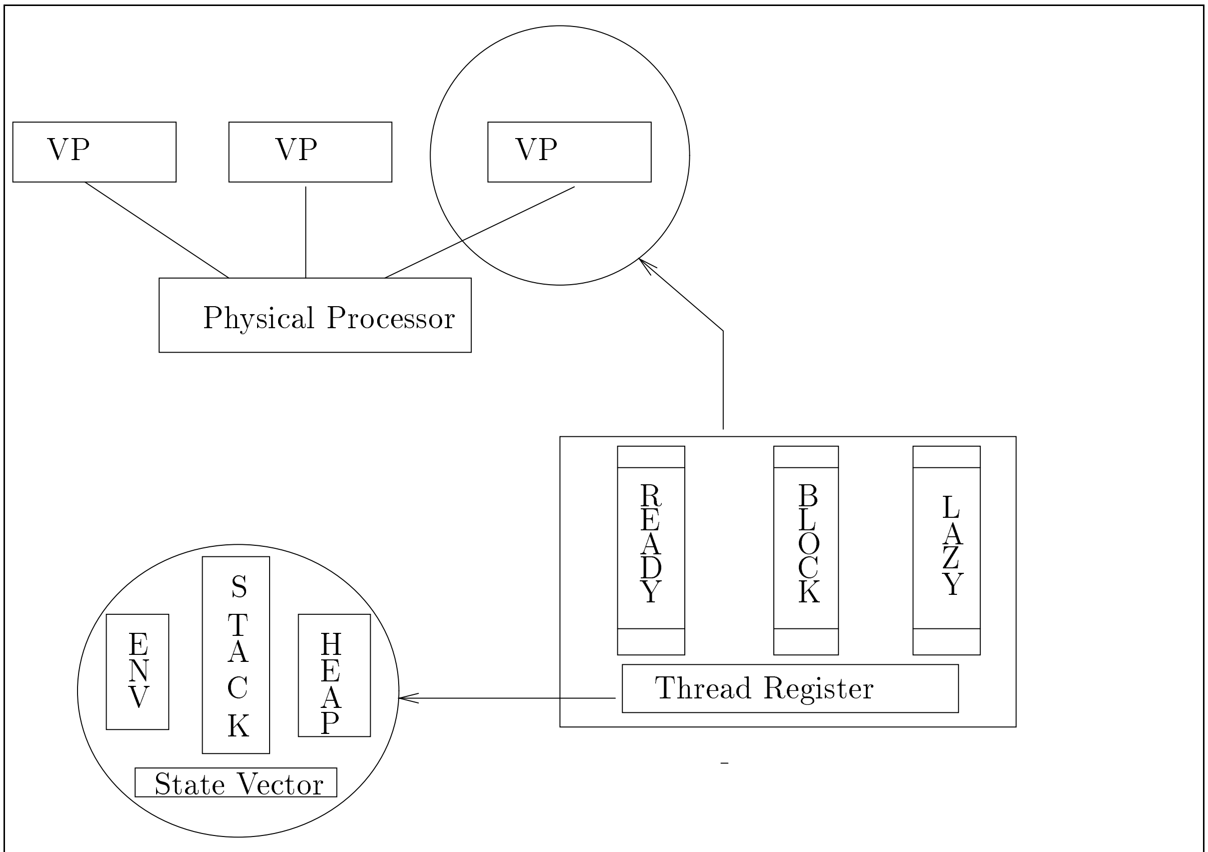


Figure 6: Abstract Architecture of the SOS runtime kernel.

The virtual machine on which $\mathcal{TS}^k$ processes execute consists of a number of queues responsible for holding threads in various states (see figure 6). The ready queue hold threads that are runnable; the block queue holds threads that are waiting for some programmer-specified condition to be satisfied; the lazy queue holds lazy threads. A lazy thread is a thread template – it holds information about the environment within which the process is to be run, but contains no associated storage (*i.e.,* stack or heap). Lazy threads are an important innovation in SOS because they contribute directly to the efficient implementation of lightweight $\mathcal{TS}^k$ processes. In most thread-based implementations of parallelism, fine-grained parallel programs can cause the scheduler to spend more time creating and initializing threads than in actually running them. Moreover, since each thread has separately allocated stack and heap storage, context switches among threads often cannot take any advantage of cache or page locality.

A lazy thread is executed only when one of the following conditions are satisfied:

1. When the virtual processor has no runnable threads on its ready queue, it may choose to evaluate a thread object found on its lazy queue.

2. When a virtual processor has no local runnable or lazy threads, it may choose to migrate a lazy thread from another VP. Note that because lazy threads have no associated storage, this migration operation does not entail significant data movement of stack or heap.

3. When a process $P$ requires the value yielded by a lazy thread $T$, it can execute $T$'s code within its own context. To do so, we need only swap $P$'s dynamic environment information (*e.g.,* exception

handler codes etc.) with $T$'s. $T$ can now run using $P$'s stack and heap; no new process state is created. When $T$ completes, $P$'s dynamic context is restored and execution proceeds using $T$'s result.

These implementation options can be used to implement lazy process evaluation in $\mathcal{TS}^k$. To support fine-grained parallelism, we can specify that the expression

(spawn $T$ ($E_1$, $E_2$, ..., $E_n$))

creates $n$ lazy threads. Each thread is a closure over the $E_i$. The effect of executing `spawn` is to deposit a tuple containing these threads into $T$. A `rd` or `get` operation that is subsequently performed on $T$ which extracts this tuple can be implemented by:

1. transforming each of these lazy threads to "eager" ones, and then proceeding to examine other tuple elements, or
2. evaluating only those lazy threads corresponding to the $E_i$ whose value is needed to determine if a match exists. These threads are evaluated in some sequential order within the execution environment of the process in which the `rd` or `get` operation is found. The lazy threads matched to formals are evaluated within the body of the tuple only when needed. Or,
3. evaluating one thread in the execution environment of the current process, and transforming the others to eager threads. The current process blocks until the value of these threads is determined.

All these implementations help throttle the creation of new threads; delaying thread creation minimizes the time used by the runtime system to context switch and manage storage. SOS permits programmers to choose their own scheduling policy; thus, different applications can be executed using scheduling disciplines tailored to their particular runtime requirements.

Consider the matrix multiplication example given earlier. The inner loop of the procedure generates $3*(m*k)$ processes; there are $m*k$ processes responsible for computing the value of a field in the result matrix. Using a fully lazy evaluation strategy coupled with thread stealing, the result matrix would be defined as a tuple-space containing $m*k$ tuples – these tuples contain lazy threads that are initiated either when their values are demanded or when free processors remove them from the lazy queue and evaluate them. Thus, evaluating the `rd` expression in the following program fragment:

```
result = (matrix_multiply A B m n k);
(rd result (1,1,?v) v)
```

causes a tuple to be read from `result`; this tuple consists of three lazy threads. The first two processes compute indices of the result entry; the last process computes the value of this entry. At the time the above `rd` expression accesses such a tuple, its component fields are either:

1. Lazy threads that have been transferred to the ready queue of some virtual processor and are currently under evaluation.
2. Lazy threads that have finished evaluating.
3. Lazy threads that have not been allocated to a virtual processor and have not been stolen.

In the first case, the executing `rd` blocks on the appropriate threads and waits for a result. In the second case, the values generated by evaluating the lazy thread are retrieved from the thread state. In the last case, the accessed thread is stolen and evaluated within the execution context of `rd`'s process. Note that if all three lazy threads in the accessed tuple have not yet evaluated, the stealing protocol dictates that `rd` only evaluate the first two fields since the last field is not used to determine a match.

Since the evaluation of `rd`'s body depends on the value of this formal, a forcing operation is performed only when `v` is accessed. This operation transforms the lazy thread into an active process; the value yielded by the process is the value yielded by `rd`. Since the result computed by instantiating a lazy thread is cached within the thread's descriptor, subsequent `rd`'s on this tuple do not involve establishing new thread contexts. Note that if the value of a formal is never demanded within the body of the `rd` operation, *e.g.,* as in

```
result = (matrix_multiply A B m n k);
(rd result (1,1,?v) true)
```

the process corresponding to the formal is not stolen.

If the tuple retrieved by `rd` does *not* match with its template argument, the operator must search for another potential matching tuple.

The type system can provide information useful in building a priority queue for lazy threads. By analyzing the type of `result`, it can be determined that the first two fields of `result`'s tuples are always used in matching operations whereas the third field is not. By building a priority queue structure for lazy threads in which these fields have higher priority than the third, we can increase the likelihood that any given `rd` will encounter fully evaluated matching fields when a matrix element is accessed.

# 8 Conclusions

A major criticism of distributed data structures has been that efficient implementations are difficult to construct. To some extent, these criticisms have been valid particularly when considered in the context of fine-grained concurrency. Previous implementations of tuple-space languages, for example, have by and large ignored issues of runtime scheduling and storage management, and have not fully addressed the implications of using semantic-based compile-time analysis for building optimal representations of tuple-space structures.

This paper has focused on both concerns. Type analysis is used to infer structural properties of distributed data structures. Generating efficient representations for tuple-spaces becomes tractable once their type structure is derived. A runtime kernel that permits deferred evaluation of thread objects makes it possible for programs to generate many fine-grained processes; resources for these processes are allocated only when the values they produce are required. Thus, the creation of active threads is dictated wholly by a program's runtime dynamics.

We expect these techniques will make distributed data structures a natural linguistic device within which to efficiently exploit fine-grained parallelism in a variety of applications. Implementation of these ideas is currently underway.

# References

[1] Thomas Anderson, Edward Lazowska, and Henry Levy. The Performance Implications of Thread Management Alternatives for Shared Memory MultiProcessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[2] Arvind and Rishiyur Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. In *Proceedings of the PARLE Conference*, 1987. Published as Springer-Verlag LNCS Volume 259.

[3] Rajive Bagrodia and Sharad Mathur. Efficient Implementation of High-Level Parallel Programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 142–151, 1991.

[4] A.D. Birrell, J.V. Guttag, J.J. Horning, and R. Levi. Synchronization Primitives for a Multiprocessor: A Formal Specification. In *Proceedings of the 11$^{th}$ Symposium on Operating Systems Principles*, pages 94–102, November 1987.

[5] Nick Carriero and David Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3), September 1989.

[6] Nick Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444 – 458, April 1989.

[7] Nick Carriero and David Gelernter. Tuple Analysis and Partial Evaluation Strategies in the Linda Precompiler. In *Second Workshop on Languages and Compilers for Parallelism*. MIT Press, August 1989.

[8] Marina Chen, Young-Il Choo, and Jingke Li. Compiling Parallel Programs by Optimizing Performance. *Journal of Supercomputing*, 2:171–207, 198.

[9] Eric Cooper and Richard Draves. C Threads. Technical Report CMU-CS-88-154, Carnegie-Mellon University, June.

[10] William Dally *et. al.* The J-Machine: A Fine-Grain Concurrent Computer. In *Proceedings of the 1989 IFIPS Conference*, 1989.

[11] Robert Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[12] Waldemar Horwat, Andrew Chien, and William Dally. Experience with CST: Programming and Implementation. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 101–109, June 1989.

[13] Suresh Jagannathan and Jim Philbin. A foundation for an efficient multi-threaded scheme system. Technical Report 91-009-3-0050-2, NEC Research Institute, 1991.

[14] Charles Koelbel and Piyush Mehrotra. Supporting Shared Data Structures on Distributed Memory Machines. In *Second ACM Symposium on Principles and Practice of Parallel Programming*, pages 177–187, March 1990.

[15] Jonathan Rees and William Clinger, editors. The Revised$^3$ Report on the Algorithmic Language Scheme. *ACM Sigplan Notices*, 21(12), 1986.

[16] Ehud Shapiro, editor. *Concurrent Prolog : Collected Papers.* MIT Press, 1987. Volumes 1 and 2.

[17] 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.

[18] Charles Thacker, Lawerence Stewart, and Edward Satterthwaite, Jr. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.