

# A Set-based Language for Prototyping Parallel Algorithms

Robert Hummel, New York University and INRIA-Rocquencourt\*

Rob Kelly, New York University and Grumman Data Systems†

Susan Flynn Hummel, Ecoles des Mines de Paris and Polytechnic University of NY‡

## Abstract

Using the set-based language SETL, we describe a number of enhancements that permit its use for programming parallel algorithms. The resulting language is versatile, allowing for specification of SIMD and MIMD algorithms, executable on a uniprocessor with keyword-substitution preprocessing, and comprehensible, in that the syntax of SETL is based on usual mathematical set theory notation. The language is imperative and incorporates associative accessing, shared variables, both static and dynamic concurrency, pipeline parallelism, parallel prefix and reduction operators, and Fetch& $\Phi$  operators. Algorithms may be specified at a high level, and may be made gradually more specific to particular implementation strategies and particular architectures and topologies. We illustrate the language with a number of example algorithms.

## 1. Introduction

SETL is a very high-level prototyping language based on set theory. The language supports an unusual data structure: the *set*, which is an unordered collection of heterogeneous elements. The iterates of a loop over a set can be executed in any sequential order. Traditional programming languages (the so-called von Neumann languages) force programs to over-specify the flow of control: loop iterations are specified as serial even though their serial execution may not be required. It is therefore unnatural to express parallel algorithms with such languages, since concurrent processing requires that the set of operations be unordered. By using sets, SETL allows programs to specify that they "don't care" about the order of operations. Sets thus provide a convenient mechanism for expressing algorithms: the programmer can concentrate on *what* rather than *where*, *when* and *how* [1].

In this paper, we describe a number of enhancements that permit the use of SETL in prototyping parallel algorithms. Using a language that has sets and maps (sets of tuples) as primitive data structures for a parallel language is logical, because there are multiple sets and maps involved in a primitive way in a parallel specification: namely, there are a set of processors, sets of tasks to be performed, maps between processors that represent communication channels, maps to invoke random access reads and writes, and maps from the processors to the sets of tasks.

The resulting language, Parallel SETL, is extremely versatile. By varying the amount of synchronization in the specified algorithms, Parallel SETL can be used to code SIMD algorithms or MIMD algorithms. By fixing a graph structure in advance, the language can be used to specify algorithms for particular topologies, as well as for parallel random access models. The language supports three major categories of parallel algorithms (see [2, 3]): data parallelism (both SIMD and MIMD), work queues, and pipelines. Finally, the language uses the shared variable model, but allows certain variables to be stored locally on processors, while avoiding special declarations and maintaining clarity of the identity of objects.

Our goal was to design a parallel language that we believe is suitable for the specification of parallel algorithms, and will serve for prototyping algorithms from high-level specifications to architecture-specific implementations. In particular, the language design should: (1) seamlessly integrate parallelism into SETL, (2) provide support for the three main classes of parallel algorithms, and (3) only add features that can be implemented efficiently on a wide range of machines. Below, we discuss the first two desiderata in relation to other parallel languages.

Parallel SETL has features in common with other parallel languages, including Unity, \*Lisp, C\*, Linda, Occam and Ada. The goal of Parallel SETL is similar to that of the language Unity [1]. Both languages use sets to avoid forcing programmers to over-specify control flow, and maps in order to specialize an algorithm to a particular topology. However, a Unity program has no control flow, consisting of a collection of guarded instructions that are

\*Courant Institute, New York University and NYU Center for Neural Science, currently on sabbatical visiting Project Epidaure, INRIA-Rocquencourt, BP 105, 78153 Le Chesnay cedex, France. Email: hummel@cs.nyu.edu

†Grumman Data Systems, 1000 Woodbury Road, Woodbury NY 11718. Email: robkelly@gdstech.grumman.com

‡Polytechnic University of New York, currently on leave at Ecoles des Mines de Paris, Centre de Recherche en Informatique, 35 Rue St-Honoré, 77305 Fontainebleau, France. Email: hummel@mono.poly.edu

executed infinitely often. We believe that abandoning control flow altogether is too drastic: programmers are accustomed to imperative languages, and moreover, programs generally consist of both sequential and parallel sections. Parallel SETL allows for a natural transition from serial to parallel algorithms.

\*Lisp and C\* are SIMD data-parallel languages targeted to the Connection Machine™. Like Parallel SETL, they are extensions to existing sequential programming languages. However, because the base languages are not very high-level, the addition of data-parallel constructs is somewhat awkward. Moreover, both languages only support ordered collections, whereas Parallel SETL supports unordered and ordered data parallelism. Because it is set-based, SETL has regularly been a candidate language for the Connection Machine™[4,5]. Reduction and scan operations on collections, which are the cornerstones of many parallel algorithms, are supported by \*Lisp and C\*. SETL includes reduction operations, and scan operations have been added to Parallel SETL. In fact, for a range of operators  $\Phi$ , Parallel SETL supports Fetch& $\Phi$ , which applies  $\Phi$  to a single element,  $\Phi$ -reduction, which applies  $\Phi$  to reduce the elements of a collection into a single element, and  $\Phi$ -scan, which applies  $\Phi$  to reduce subsequences of the elements of a collection.

Linda [6] is not really a language, but a number of parallel programming primitives that can be added to "host" languages. These primitives are built around associatively-accessed shared collections of tuples (records), called *tuple spaces*. All accesses to shared data are synchronous. In Parallel SETL, we allow both synchronous and asynchronous operations on shared data. Because SETL incorporates sets of tuples that can be associatively accessed, it has been proposed to add the Linda primitives to SETL [7]; however, we feel that the primitives are too low-level to be used as specification tools, and are subsumed by the facilities of Parallel SETL. Linda also does not allow related collections of tasks to be created together. Tasks are created sequentially (using *evals*, which are similar to the Unix *fork*). Thus Linda is not well-suited to programming highly-parallel machines. In Parallel SETL, the parallel creation of collections of tasks is specified using loops.

The parallel constructs of Occam and Ada are primarily designed to support system programming, e.g., to write operating systems, rather than applications such as image and graph algorithms. Hence these languages support explicit tasking (i.e., threads of control with individual program counters) and mechanisms for inter-task communication. Since the goal of Parallel SETL is to specify parallel algorithms, we chose to not include explicit tasking, as this would unnecessarily complicate the language. Further, the pairwise communication mechanisms of Occam and Ada are inappropriate for coordinating large numbers of tasks. We therefore chose instead to support shared variables and the hierarchy of synchronization mechanisms mentioned above. Our experience has been that these mechanisms and parallel loops provide all the required functionality. Although collections of related tasks can be created together in Ada, individual tasks do not know their identities. It is useful for tasks to have unique identities, so that they can "help themselves" to work, for instance, to determine which elements of a matrix to calculate. Because the parallel construct of Parallel SETL is a loop, iterations automatically have unique identities.

## 2. The SETL Language

SETL was initially developed in the mid-1970s by Professor Jack Schwartz and researchers at New York University [8]. The aim of the language was to permit the high-level executable description of algorithms and systems, in comprehensible self-documenting code. SETL facilitates prototyping by allowing a program to be refined into successively finer detail while staying within the language. A book describing the SETL system was published in 1986 [9]. A milestone of the SETL research program was achieved with the world's first validated Ada compiler, written in SETL by Professors Dewar and Schonberg and the NYU Ada/Ed group at NYU [10]. More recent work has resulted in a new interpreter, SETL2, [11], which simplifies some of the syntax, adds object-oriented features, and provides much improved performance over previous versions of SETL. SETL2 is currently available publicly for many different platforms, including Sun3, Sun4, PC, and Macintosh machines. A complete description of the language is available along with documentation in the distribution [12]. A simplified description can be found in [13]. Parallel SETL as presented here is derived from an earlier version, described in [14]; many examples of parallel algorithms, especially image algorithms, coded in Parallel SETL, may be found there.

For the most part, SETL (and Parallel SETL) can be understood by examples. We provide here only a few comments on some of its more salient features. The main datatypes in SETL that are not present in other standard declarative languages are the *set* and *tuple* types. Sets are always finite, and are denoted with curly brackets '{' and '}'. Tuples are finite ordered collections, and unlike sets, permit repetition of elements. They are denoted with brackets '[' and ']'. The elements in sets and tuples are not required to be of a uniform datatype. Thus a set can contain integers, strings, tuples, and other sets as elements. A map is a set of tuples, where each tuple has two elements. Elements of a map are accessed through a function-like notation  $m(x)$ , which associatively accesses the tuple in the set whose first element is designated as the domain element  $x$ . Parallelism is introduced into SETL by explicit and implicit loops over sets and tuples. Sets and tuples are treated as uniformly as possible, leading to orthogonal parallel constructs and a clean design.

### 3. Example Parallel SETL Code

At this point, we present several algorithms coded in Parallel SETL, to give the reader the flavor of the language. Our first example is the classic two-dimensional mesh algorithm for matrix multiplication. We assume that the  $a(i, j)$  and  $b(i, j)$  matrices are stored in the local memories of processors, with the processor indexed by coordinates  $(i, j)$  containing the data  $a(i, j)$  and  $b(i, j)$ . When completed, the  $O(n)$  time SIMD algorithm results in the product matrix,  $c(i, j)$ , similarly stored in local processor memory.

```
procedure matrix_multiply(a,b);
-- The standard MC^2 (2-D mesh) parallel algorithm for matrix multiply

-- Set up topology maps.
-- These lines are not necessarily executed on the parallel machine.

indices := { [i,j] : i in {1..n}, j in {1..n} };
north := { [i,j, [ ( (i-1)-1 mod n) + 1 , j ] ] : [i,j] in indices };
south := { [i,j, [ ( (i-1)+1 mod n) + 1 , j ] ] : [i,j] in indices };
east := { [i,j, [ i , ( (j-1)+1 mod n) + 1 ] ] : [i,j] in indices };
west := { [i,j, [ i , ( (j-1)-1 mod n) + 1 ] ] : [i,j] in indices };

-- Input is a(i,j) and b(i,j) for [i,j] in indices. We consider these
-- already present

for k in [1..n-1] seq_loop -- Matrix staggering
  s := {}; t := {};
  for [i,j] in indices simd_par_loop
    if k < i then
      s(i,j) := a(east(i,j));
    else
      s(i,j) := a(i,j);
    end if;
    a(i,j) := s(i,j);
    if k < j then
      t(i,j) := b(south(i,j));
    else
      t(i,j) := b(i,j);
    end if;
    b(i,j) := t(i,j);
  end simd_par_loop;
end seq_loop;

c := { [i,j,0] : [i,j] in indices }; -- Initial parallel assignment

for k in [1..n] seq_loop -- Matrix summing
  for [i,j] in indices simd_par_loop
    c(i,j) += a(i,j)*b(i,j);
    a := { [i,j,a(east(i,j))] : [i,j] in indices }; -- Parallel read
-- from east
    b := { [i,j,b(south(i,j))] : [i,j] in indices }; -- Parallel read
-- from south
  end simd_par_loop;
end seq_loop;
end matrix_multiply;
```

The above code is already at a fairly low architecture-specific level. A high-level specification of matrix multiplication is:

```
c := {[i,j, +/(a[i,k]*b[k,j] : k in {1..n})] : i in {1..n}, j in {1..n}}
```

We note that certain standard image processing algorithms also admit high-level parallel specifications. For example, histogramming is:

```
hist := { [v,0] : v in Values };
for [i,j] in Pixels par_loop
  hist(f(i,j)) += 1;
end par_loop;
```

Similarly, edge extractions, Hough transforms, and other simple image processing functions are easily coded.

As a nontrivial example, we give a MIMD version of the Shiloach/Vishkin connected components algorithm [15]. This was originally defined as a SIMD algorithm for finding the connected components of a general graph, but may be applied as a MIMD algorithm, and may also be applied to the binary image labeling problem. The algorithm consists of a sequential loop that iterates  $O(\log n)$  times. Within each iteration, there are four steps: a shortcutting step, an ordered hooking step, a stagnant node hooking step, and another shortcutting step. The algorithm operates by making changes to the "pointer graph," which is a collection of pointers, one for each node, where pointers always point to other nodes, and the entire collection is always organized as a forest. Initially, every node points to itself. Upon termination, the pointer graph has the property that every node in any given connected component points to the same node in that component.

It has been previously shown that each substep within each iteration of the original Shiloach/Vishkin connected components algorithm may be processed asynchronously [16], providing there is synchronization between the steps. Further, it has been shown how lists of nodes and edges can be maintained so that all possible productive shortcutting and hooking is guaranteed to happen by actions within the subset of nodes and subset of edges. These lists can be made progressively smaller from iteration to iteration. In this way, the original Shiloach/Vishkin SIMD connected components algorithm is made suitable for MIMD task queue operation. A C-code version of the same algorithm may be found in [17]. The Parallel SETL version is shorter, and also considerably more comprehensible.

```

procedure mimd_shiloach_vishkin(Graph);
  [Vertices,Edges] := Graph

  [ Ptr, Shortcutlist1, Shortcutlist4, Edgelist2, hooked] := -- parallel setups
  [ { [x,x] : x in Vertices }, -- Ptr
    { }, -- Shortcutlist1
    Vertices, -- Shortcutlist4
    Edges, -- Edgelist2
    { [x,0] : x in Vertices } ]; -- hooked

  while Shortcutlist4 /= { } seq_loop

    nonstagnant := { }; --Shortcut 1
    for x in Shortcutlist1 par_loop
      old(x) := Ptr(x);
      Ptr(x) := Ptr(Ptr(x));
      if old(x) /= Ptr(x) then
        nonstagnant with:= Ptr(x);
      end if;
    end par_loop;

    Edgelist3 := { }; --Ordered hooking
    for [x,y] in Edgelist2 par_loop
      u(x,y) := Ptr(x); v(x,y) := Ptr(y);
      dohook(x,y) := is_root(u(x,y)) and
        v(x,y) < u(x,y) and
        (Fetchor(hooked(u(x,y),1) = 0) ;
      if dohook(x,y) then
        Ptr(u(x,y)) := v(x,y);
        nonstagnant with:= v(x,y);
      end if;
      if u(x,y) /= v(x,y) and not dohook(x,y) then
        Edgelist3 with:= [x,y];
      end if;
    end par_loop;

    Edgelist2 := { }; --Stagnant hooking
    for [x,y] in Edgelist3 par_loop
      u(x,y) := Ptr(x); v(x,y) := Ptr(y);
      dohook(x,y) := is_root(u(x,y)) and
        v(x,y) /= u(x,y) and
        u(x,y) notin nonstagnant and
        (Fetchor(hooked(u(x,y),1) = 0) ;
      if dohook(x,y) then
        Ptr(u(x,y)) := v(x,y);
        nonstagnant with:= v(x,y);
  
```

```

        end if;
        if u(x,y) /= v(x,y) and not dohook(x,y) then
            Edgelist2 with:= [x,y];
        end if;
    end par_loop;

    Newlist := { }; Shortcutlist1 := { };
    for x in Shortcutlist4 par_loop --Shortcut 2
        old(x) := Ptr(x);
        Ptr(x) := Ptr(Ptr(x));
        if old(x) /= Ptr(x) then
            Shortcutlist1 with:= Ptr(x);
        end if;
        if old(x) /= Ptr(x) or Ptr(x) in nonstagnant then
            Newlist with:= x;
        end if;
    end par_loop;
    Shortcutlist4 := Newlist;

end seq_loop;

return { { x in Vertices | Ptr(x) = y } : y in range Ptr };
end mind_shiloach_vishkin;

```

We assume that the procedure `is_root(x)` is defined by:

```

procedure is_root(x);
return Ptr(x) = x ;
end is_root;

```

#### 4. Goals for Parallel SETL

We have three design principles for Parallel SETL:

- (1) It should produce readable code that specifies the algorithm and the parallelism within the algorithm as clearly as possible;
- (2) It should add as little as possible to the SETL language syntax while adding considerable power for expressing parallelism; and
- (3) It should be executable on a uniprocessor using a standard SETL interpreter with very few, easily automated, changes to the parallel code, in order to verify the correctness of the algorithm and the specification for the case of a single processor.

We are not proposing to produce a Parallel SETL compiler or translator for a multiprocessor. For our purposes, the existence of such a compiler is unimportant. For execution on any machine beyond a uniprocessor, it is intended that the Parallel SETL should be refined to lower-level parallel code, in order to give more accurate indications of algorithm efficiencies. This route has proven successful for serial SETL programs: the high-level Ada/Ed compiler was translated into low-level SETL constructs, then into C, and finally retargeted to shared-memory multiprocessors [18]. Progress is being made in automating the translation of high-level serial constructs into low-level ones, and similar techniques may be applicable to parallel constructs.

Further, we do not intend that a correctly-specified serial SETL program will automatically translate to valid Parallel SETL code. There are too many convenient features of SETL that we wish to imply concurrent operation (such as implied loops). Although Parallel SETL makes parallelism obvious to a compiler, its primary use is as a vehicle for conveniently and clearly specifying parallel algorithms.

After code written in Parallel SETL has been verified on a uniprocessor, there is no guarantee that the algorithm and code will work on a multiprocessor. The ability to execute the code on a uniprocessor provides one level of testing and analysis, but will not exercise the synchronization facilities, nor permit the discovery of race conditions or other potential problems with the parallel code. However, since Parallel SETL is so high-level, its expressive constructs can be exploited by a program verifier (either automatic and manual) to prove correctness in a parallel environment.

Using the object-oriented extensions in SETL2, it is easy to define and make use of the datatype of *bags*. A bag is a set which may contain duplicate elements. Equivalently, we can think of a bag as a tuple with no specific ordering. Parallel SETL is easily extended to make use of bags; they are especially useful for parallel-prefix operations, and parallel loops. Likewise, parallelism over strings is easily defined. However, in the description below, we

generally restrict the discussion to the use of sets and tuples.

## 5. Parallel SETL

We now present extensions to the SETL language that facilitate its use as a language for specifying parallel algorithms. We begin by defining the constructs that will be used to specify concurrent execution, and then discuss the data management and operations that are permitted during concurrent processing. Finally, we discuss how Parallel SETL is converted into SETL.

### 5.1. Expressing Concurrency

This section lists all of the methods used in the Parallel SETL language to express concurrent execution. Generally, loops using the `par_loop`, `simd_par_loop`, or `pipe_loop` keyword denote parallel loops, while `seq_loop` is used to denote sequential loops. Additionally, implicit loops over sets and tuples denote parallelism, providing the operations are legal for concurrency. Parallel loops within loops are generally intended to imply additional parallelism, under the assumption that there is a sufficient supply of processors. Parallel loops have a block structure, in the same way that SETL2 is block-structured language. In practice, the degree of parallelism will be limited by the supply of processors and the ability of the coder or compiler to detect all levels of intended parallelism.

Most importantly, the algorithms should be designed so that correctness does not depend upon concurrent execution whenever concurrency is expressed — the algorithm should be correct given a limited pool of processors that are able to cooperate on the concurrent tasks that are assigned. In particular, the algorithm should be correct when executed using a single processor. Finally, the specification is legal only if the operations with side-effects within concurrent execution are restricted to a set of specified functions. We list the allowable operations in the 5.2.

There are three kinds of parallelism permitted in Parallel SETL: static concurrency, dynamic concurrency, and pipeline concurrency. These correspond to the three kinds of parallelism described in [2], *geometric* (data parallelism), *farm* (task pool), and *algorithmic* (pipeline) parallelism. (The Linda community calls these terms: result, activity and structure parallelism [6].)

In static concurrency, the object indexing the set of processes that may be executed concurrently is fixed, whereas dynamic concurrency refers to a "pool of tasks" that may change dynamically while members of the pool are executing. Pipeline concurrency implies a sequence of steps which operate sequentially on each element from a list of data, but may operate in parallel on different elements from the list.

**5.1.1. Static Concurrency.** There are three kinds of static concurrency: `par_loop`'s, `simd_par_loop`'s, and implicit concurrent loops.

`Par_loop`'s. These are used to specify MIMD parallel loops over a fixed collection of data elements. Suppose `nodes` is a set or tuple. Then

```
for x in nodes par_loop
```

introduces a parallel loop over the elements. The key-word `seq_loop` may be used to emphasize that the processing is serial:

```
for x in nodes seq_loop
```

and it is appropriate style to specify all loops as either `par_loops` or `seq_loops`. Throughout, `x` stands for any valid left-hand side of an assignment, and is assigned to individual set elements. The semantics of the static parallel loop is that the set of `nodes` is evaluated, and then as many processors as are available or necessary are assigned independently to iterates. The iterate variable is replicated, so that each iterate has a separate value of the iterate variable `x`. When a processor completes the processing of a loop iterate, and if there are elements of `nodes` that have not yet been processed, then the processor is assigned to another element, and continues processing.

The loop constructs implies synchronization at the end of the loop whenever there is a matching `end par_loop` statement. If the matching statement is an

```
unsynchronized_and par_loop;
```

statement, then synchronization is not assured at the end of the loop, and processors may proceed to look for succeeding parallel work.

`Simd_par_loop`'s. These are used to specify SIMD concurrent loops over a fixed collection of data. The construct:

```
for x in nodes simd_par_loop
```

is a special form of a `par_loop`, where extra synchronization is implied within the body of the loop. We choose to assume that synchronization occurs after every statement, so that

```

for x in nodes simd_par_loop
  statement1;
  statement2; ...
end simd_par_loop;

```

is equivalent to:

```

for x in nodes par_loop
  statement1;
end par_loop;
for x in nodes par_loop
  statement2;
end par_loop; ...

```

That is, each statement is implicitly contained in a single loop, so that all iterates of each statement is finished before any iterate of the next statement is performed. If the statement is a conditional one containing multiple blocks of statements, such as in a case statement, then we make a slight departure from SIMD processing. We recursively allow the subblocks to execute concurrently, with processors executing each subblock synchronizing between statements, and a barrier synchronization at the end of the conditional statement.

Implicit static concurrency. Iterators over sets and tuples denote concurrency providing the operations applied to the iterates are permissible for concurrent operation. We specify four kinds of implicit loops: (1) Set and tuple constructors, (2) Quantified tests, (3) Compound operators and parallel prefix operators, and (4) Tuple element assignment. In all forms, we will always assume synchronization at the end of the loop. As usual, the set indexing the parallel tasks must be static during the parallel execution.

(1) Implicit iterators can be used in sets and tuple constructors. For example,

```

new_graph := {[x,min/graph(x)] : x in nodes};

```

is equivalent to

```

new_graph := { };
for x in nodes par_loop
  new_graph with:= [x,min/graph(x)];
end loop;

```

which is permissible for concurrent execution, since insertion into a set is allowed within concurrent code (as discussed below). In fact, the same code may be written as:

```

new_graph := { };
for x in nodes par_loop
  new_graph(x) := min/graph(x) ;
end par_loop;

```

Note, however, that

```

graph := {[x,min/graph(x)] : x in nodes};

```

is not the same as

```

for x in nodes par_loop
  graph(x) := min/graph(x) ;
end par_loop;

```

since we cannot assume that there will be complete parallelism in the `par_loop` form, so that `graph` may be intermediately updated before all iterates have been performed. In the implicit loop form, the entire set on the right hand side will be created using parallel processing, and processors will synchronize before `graph` is reassigned.

(2) Compound operators applied to sets or tuples also express concurrency, providing the operation is a binary associative operator among the set of allowable operators for concurrent execution (a  $\Phi$  operator, see 5.2 (4)). Such constructs can be viewed as *reduction* operations. For example, summing up the elements in a set:

```

+ / set

```

can be accomplished by a number of processors performing additive-writes on a shared variable. Alternatively, if the set (or tuple) has  $n$  elements, then  $n$  processors could cooperate in  $\log n$  steps to perform the sum. The exact method used to implement the reduction will depend on the architecture, but the construct implies the possibility of parallelism. Note, for example, that in the example assigning `graphs(x)` in the example above, the computation of the minimum value over `graph{x}` in the can be regarded as a parallel loop within a parallel loop, and expresses concurrent execution.

Similarly, *parallel prefix* operations [19] (also called *scan* operations in the Connection Machine™ parlance) are built into Parallel SETL, using the added syntax:

$\Phi$ // nodes

where  $\Phi$  is some binary operation and nodes is either a set or tuple (or bag). The result is an object of the same type containing the results of a parallel prefix operation. Thus, for example,

+// tuple

applied to a tuple results in a tuple of all partial sums, equivalent to

```
[ +/ tuple(1..i) : i in [1..#tuple] ]
```

Likewise, the summing operation can be replaced with any legal binary associative operator, such as max. When applied to a set, the parallel prefix operation arbitrarily orders the elements, finds the partial sums (or partial results), and forms a new set (eliminating duplicates). More usefully, the parallel prefix operation applied to a bag produces a bag of partial results. These operations are not a part of regular SETL, but are so important to parallel algorithms that we have added the syntax in order to indicate explicitly that efficient parallel operations can be used.

(3) The existential and the universal quantified tests, exists and forall, denote concurrent execution. Thus

```
exists x in nodes | Condition(x)
```

and

```
forall x in S | Condition(x)
```

will be performed in parallel. Evaluation of the condition in both cases is restricted to allowable operations, and side-effects of the evaluation are generally not permitted. Further, as soon as one process finds an iterate that decides the test (i.e., a value of  $x$  that satisfies the exists test or a value of  $x$  that invalidates the forall test), then all outstanding iterates are canceled. Although these abort semantics are somewhat complex, they are less so than Ada abort semantics, which admits an efficient multiprocessor implementation [18]. We assume that all such details are handled in the refinement and implementation of the tests.

As with serial SETL, whenever an exists test is found to be true, or a forall test is found to be false, the value of the variable is bound to an arbitrary witness at the end of the construct. If there are multiple witnesses discovered in concurrent execution, then an arbitrary witness will be bound to the variable. There is no guarantee with tests over a tuple that the first such witness in the tuple is the one that will be bound to the variable.

(4) A tuple of variables is a valid left-hand-side in SETL:

```
[v1,v2, ... , vn] := tuple;
```

In Parallel SETL, whenever a tuple of variables appears as a valid left hand side, an implicit parallel loop is invoked, and concurrent execution is implied for the evaluation and assignment of values. Thus, one way to imply concurrent execution of two procedures is to assign their values to tuple components:

```
[a,b] := [proc1(),proc2()];
```

**5.1.2. Dynamic Concurrency.** Concurrent execution of tasks extracted from queues are expressed using the syntax:

```
while queue /= [ ] par_loop
  x fromb queue;
  ...
end par_loop;
```

This notation is to be read as a single construct, since it implies fairly complex queue management. Within the body of the loop, and before entrance to the loop, it is permissible to append elements to queue, which is either a set, tuple, or bag, by using insertion, concatenation, or the with operator or variants. (Appropriate syntax will depend on the type of queue.) In this way, the "pool of tasks" that are maintained for concurrent execution can vary dynamically. Items may also be removed from the queue by active processors, by using the operators from, frome or fromb. Highly-parallel algorithms, involving less than ten instructions per operation, are available [18,20,21] and are envisioned for the refinement of all queue operations. Note that  $x$  is a variable or any valid left-hand side. When using a tuple for the queue, the serial semantics imply a LIFO or FIFO queue, but there is no guarantee of ordering for concurrent accesses. We also permit concurrent frome and fromb operators on the same queue at the same time.

The semantics of this special syntax include task allocation and task termination conventions. Specifically, it is assumed that processors extract tasks from the queue, removing the queue elements indivisibly with the allocation. Each processor thus extracts a unique element from the queue, and receives a binding of a variable to a queue element. Concurrent inserts and deletes may occur during the loop, and if the loop is embedded within other concurrent processing, or if there is an unsynchronized\_end preceding the loop, then concurrent inserts and deletes may take place before the loop. In all cases, the queue must be a local variable relative to any enclosing parallel loops, and



must be initialized and nonempty before the loop begins.

The loop terminates when all processors have completed loop iterates, are waiting for new queue elements to extract, and the queue is empty. However, if the queue is empty and if some processors are still executing loop iterations, then idle processors must wait for the other processors to terminate before proceeding outside of the `while`-loop, since items may be added to the queue. There is thus always a synchronization at the end of the loop.

A special additional declaration called `withident` may be added to the `while` construct. If it appears, it has the form

```
withident variable;
```

and occurs after the loop statement but before the iterate extraction:

```
while queue /= [ ] par_loop
  withident i;
  x fromb queue;
  ...
end par_loop;
```

The meaning of the `withident` declaration is that the variable `i` will be assigned a unique identifier (of type `atom`) with each iterate extracted from the queue. The effect is as though the assignment `i := newat()` occurs indivisibly with the extraction of the bound variable `x` from the queue. The identifier `i` is always a unique value, different for each element extracted from the queue, even if the element values `x` are the same due to repetitions in the queue, and even if the elements have been extracted at different times but from the same position in the queue.

**5.1.3. Pipeline Concurrency.** Finally, we define the construct

```
for x in nodes pipe_loop
  statement1; ... statementn;
end pipe_loop;
```

to define pipeline processing of the elements in the set or tuple of `nodes`. Each statement is viewed as a `stage` in the pipeline. Thus, the `n` statements are allowed to operate concurrently, and each can be assigned to a processor. The iterates are piped through the statements in order, with each statement being executed once per iterate. The semantics guarantee that statements are executed sequentially for each iterate. That is, the functionality is the same if the loop is executed serially. Communication between instances of statements (pipe stages) can be accomplished via the iterate variable, maps on the iterate variable, and variables that are declared in loop-localize concurrent, as defined in the next subsection.

## 5.2. Parallel Data Management

In keeping with the data-parallel paradigm, it is assumed that certain variables may be stored locally in the memory of a processor that executes a given iterate of a parallel loop. Parallel SETL allows one to specify the variables that will be "localized" in this fashion. In particular, we use the convention that values that are obtained from maps defined on a set of unique identifiers will be stored locally. For static loops over sets, such a *localized* variable is recognized as an evaluate of a map defined on the set element of the current iterate, as `f(x)` in

```
for x in S par_loop
  f(x) := PtI(x);
```

For loops that are nested, the map value is localized only if it is defined and evaluated on all enclosing iterate variables, as `C(x,y)` in

```
for x in S1 par_loop
  for y in S2 par_loop
    C(x,y) := PtI(x);
  ...
```

Map evaluates defined on all iterate variables up to a given level of nesting are localized at that level, but are shared among all iterates at nested levels. Note however that map variables conditioned on iterate identifiers are nonetheless global variables — they can be accessed by other iterates (through random access reads and writes), and can even be accessed after the loop terminates.

We also define the notion of declared localized variables. In Parallel SETL, each parallel loop creates a nested scope. Since SETL2 is a block-structured language, we permit the declaration of parallel localized loop variables, whose scope is limited to the current and embedded loops. At the start of any `par_loop` (static or dynamic), `simd_par_loop`, or `pipe_loop`, an optional `localize` declaration is allowed that declares variables to be replicated for each iterate at that nesting level. Such replicated variables are local to, and hence stored locally by, its iterates. Accordingly, a localized variable behaves like a map conditioned on the iterate variable (and iterate variables to that level), but has a scope limited to the current and embedded loops. The form looks like:

```

for x in nodes par_loop
  withident y;
  localize z1, z2, t;
  ...
end par_loop;

```

The localize statement is simply a convenience to avoid excessive use of maps on iterate variables.

The iterator variable or set of variables (the latter when  $x$  is a nonsimple valid left-hand side of an assignment) are replicated implicitly for each iterate. An iterator when extracted from a set provides the iteration with a unique identifier. The identifier variable declared by a withident statement is replicated and is always a unique identifier.

For static loops over a tuple and all dynamic loops, a localized variable is recognized as a map evaluated on the withident variable, as  $f(i)$  in:

```

f := ();
while queue /= [ ] par_loop
  withident i;
  x fromb queue;
  f(i) := Ptr(x);
  ...
end par_loop;

```

If the loop is contained inside another parallel loop, then the localized variables must be constrained by all iterates or withident variables at higher levels.

Although they may be stored locally, data-parallel constructs defined by maps on unique identifiers will exist after the termination of the loop over which they are formed. However, in order to be accessed, the elements of the set of unique identifiers must be known. In the case of the withident variables, these may either be inserted into a shared set in order to recall the identifiers at a later time, or obtained from the domain operator on such a map.

Within concurrent processing, certain operations, such as modifying a complex data structure, require new semantics. For example, if we say

```
S := S less x;
```

in a concurrent loop, then the set  $S$  will be updated, but we do not guarantee that the update is atomic (so  $S$  may be changed between the time it is read on the right hand side and the time it is written on the left hand side). In order to specify atomic operations, certain conventions must be followed. For example, in SETL, the statement

```
S less:= x;
```

is functionally equivalent to the preceding statement; in Parallel SETL, it will mean that the element deletion from  $S$  is atomic. The following form the legal atomic operations and rules for concurrent processing:

- (1) Access and modifications to localized variables by the corresponding iterate are always atomic, except that iterate variables may not be changed.
- (2) A functional style of programming, without side-effects, on non-localized objects is always permitted. In particular, concurrent readers are atomic.
- (3) Concurrent writes to a variable are allowed, and an assignment is considered atomic. (An implementation on a multiprocessor may need to employ critical sections when accessing complex data structures to ensure their integrity.) For example, concurrent writes of sets to a particular object is guaranteed to result in one of the assignments being successful, rather than resulting in a melange of values from the sets. However, in static-loops over sets, the set may not be modified.
- (4) Fetch& $\Phi$  operations on simple types are added to the language. A Fetch& $\Phi$  operation is a function which returns a value of a shared variable, and indivisibly modifies the value of that memory location according to the associative binary operator  $\Phi$ , such as addition ('+') or boolean ('or'), using some value that is provided. Specifically, Fetch& $\Phi$  is logically equivalent to the following function:

```

procedure Fetch& $\Phi$ ( variable, increment ) ;
  var x;
  x := variable
  variable  $\Phi$ := increment
  return x;
end Fetch& $\Phi$ ;

```

where the entire function executes as a critical section. For example, Fetch&+(t,3) returns the value of the integer  $t$  and at the same time increments it by 3. Many proposed parallel architectures provide hardware support for concurrent Fetch& $\Phi$  operations, and thus do not result in an explicit serialization. We thus assume that Fetch& $\Phi$  is provided as a class of built-in functions in our language, for some reasonable range of  $\Phi$  operators. Critical sections for

other operations may then be invoked, by making use of (for example) Fetch&+ [20].

(5) Additive-writes, maximum-writes, and other  $\Phi$ -writes are allowed: in essence, Fetch& $\Phi$ 's where the fetch-part is discarded. These must be denoted using the assigning operator syntax in SETL, such as in:

```
sum += f(x) ;
```

which adds the amount  $f(x)$  to the shared variable `sum` by means of an additive write.

(6) Concurrent inserts, deletes, unions, and intersections on shared sets are allowed. The operation is atomic providing an assignment operator (i.e.,  $\Phi :=$  or `from`) is used. The efficiency of these operations will depend upon the representation of the set, and the algorithms used to perform the concurrent modifications. In particular, set-formers can be parallel, through the use of concurrent inserts into the set. Thus

```
s := { x : x in graph(y) | C(x) } ;
```

specifies concurrent operation, with the number of processors up to the number of elements in the set `graph(y)`.

(7) Concurrent operations on shared tuples using the `with`, `fromb`, and `frome` operators are permitted. Concurrent concatenation using the assignment operator `+=` is also allowed, but the order of the concatenations are unpredictable. It is permissible to access the tuple of tasks in the

```
par_loop while queue /= [ ] do
```

construct, as long as the access is restricted to these allowable operations. Access to internal tuple elements by slice operations is not permitted.

An important point when writing concurrent code using Parallel SETL is that one cannot assume that all iterates in a `par_loop` are operating at the same time, and in particular one cannot assume that expressions and statements within the loop are evaluated in lockstep. To the contrary, Parallel SETL is written under the understanding that the processing of the iterates over a set that defines concurrent execution actually takes place asynchronously, and the code should be correct even if the execution is serialized. Thus any modification to a shared variable or a variable localized by another iterate (a random-access write) may influence the values read by other iterates, even in statements that come before the write within the block of concurrent code. The same is true of `simd_par_loops`, except that synchronization is assumed to occur at the end of every statement (denoted by a ';'), but not within any given statement.

### 5.3. Preprocessing Parallel SETL into SETL

In order to convert code that is written in the above Parallel SETL language into code that can be interpreted by the normal serial SETL interpreter, only a very few changes are needed. They are the following.

- (1) All `par_loop`, `simd_par_loop`, `pipe_loop`, and `seq_loop` keywords must be replaced with the `loop` keyword.
- (2) All `unsynchronized_end` keywords must be replaced with the `end` keyword.
- (3) All `withident` declarations must be changed to assign a new atom assignment, so that `withident k;` becomes `k := newat();`
- (4) The declarations of `localize` variables should be removed.
- (5) Procedures for all `Fetch& $\Phi$`  operators must be provided.
- (6) Within the scope of a `simd_par_loop`, statements with a block structure, such as `if-then-else` statements, should be changed into a sequence of simple statements of the form `if-then` with a single simple statement. This applies recursively to sub-blocks. However, embedded parallel loops should be left unchanged.
- (7) All resulting semicolons within the scope of a `simd_par_loop` (except for the final one) must be replaced by the loop termination `end loop`; followed by a repetition of the statement defining the loop.
- (8) All parallel prefix operations must be elaborated. A parallel prefix operator is recognized by a pair of slash tokens ('//'). The following object is either a simple variable object, or a set formed from '{..}' symbols or '[..]' symbols. The construct can be replaced with a procedure call of the form `parallel_prefix("op_string", object)`, which can then be coded to return the proper object.

Once these changes are made, the resulting SETL code may be executed, and should give correct results if the Parallel SETL code is correct.

## 6. Conclusions

There are many possible ways of incorporating parallelism and parallel features into SETL. The Parallel SETL language that we have described is a reasonably minimal extension to SETL, and yet its parallel expressive power is rich. We have presented a few examples of some code to demonstrate the versatility of Parallel SETL, but our design choices are based on numerous other examples. Care was also taken to select parallel features that can be

implemented on machines ranging from uniprocessors to highly parallel SIMD and MIMD machines.

We believe that Parallel SETL offers a natural platform for the specification of highly-parallel algorithms, such as image algorithms and graph algorithms. As with serial SETL programs, these specifications can be made successively more detailed and mapped onto specific topologies. With the proper tools, Parallel SETL should be an excellent vehicle for prototyping and developing efficient parallel code.

### Acknowledgements

Portions of this work form part of the Ph.D. thesis of R. Kelly. We thank Vrije Universiteit, Amsterdam, where some of the research and ideas for this report were developed in 1989 during visits by R. Hummel and S. Flynn Hummel. We thank F. Henglein for comments and suggestions. Rob Kelly acknowledges support from Grumman Data Systems.

### References

- [1] Hirschberg, D., A. Chandra, and D. Sarwate, "Computing connected components on parallel computers," *Communications of the ACM* 22, pp. 461-464 (August, 1979).
- [2] Hey, A. J. G., "Reconfigurable transputer networks: practical concurrent computation," in *Scientific Applications of Multiprocessors*, R. J. Elliott and C. A. R. Hoare, (Eds.). Prentice Hall (1989).
- [3] Danelutto, M., R. Di Meglio, S. Pelagatti, and M. Vanneschi, "A methodology for the development and the support of massively parallel programs," *Proceedings of the Parallel and Distributed Workstation Systems Workshop*, (Sep. 1991).
- [4] Hillis, D., *The Connection Machine*, MIT Press, Cambridge, MA (1985).
- [5] Belloch, G. E., "Scan primitives and parallel vector models," Ph.D. Thesis MIT, Tech Report #MIT/LCS/TR-463, (Oct. 1989).
- [6] Carriero, N. and D. Gelernter, "Linda in context," *Communications of the ACM* 32, pp. 444-458 (April, 1989).
- [7] Hasselbring, W., "Combining SETL/E with Linda," *Proceedings of the Workshop on Linda-like Systems and Their Implementation*, pp. 76-91 (June 1991).
- [8] Schwartz, J. T., "On programming, An interim report on the SETL project," NYU Technical Report, (1973).
- [9] Dewar, R. B. K., E. Dubinsky, E. Schonberg, and J. T. Schwartz, *Programming with Sets: An Introduction to the SETL Programming Language*, Springer-Verlag (1986).
- [10] Dewar, R. B. K., G. A. Fisher Jr., E. Schonberg, R. Froehlich, S. Bryant, C. F. Goss, and M. G. Burke, "The NYU Ada translator and interpreter," *The Proceedings of the IEEE CompSoc '80 Conference*, (October, 1980).
- [11] Snyder, W. K., "The SETL2 Programming Language," Technical Report No. 490, Courant Institute, NYU, (January 1990).
- [12] Snyder, K., "The SETL2 programming language," Available by anonymous ftp from cs.nyu.edu under pub/setl2, (1990).
- [13] Dewar, R.B.K., "The SETL2 programming language," Available by anonymous ftp from cs.nyu.edu under pub/local/hummel/setl2/intro.ps.Z, (December, 1990). Modified by R. Hummel.
- [14] Kelly, R., "The development of parallel image algorithms by prototyping," Ph.D. Thesis, New York University, (June, 1991).
- [15] Shiloach, Y. and U. Vishkin, "An  $O(\log n)$  parallel connectivity algorithm," *Journal of Algorithms* 3, pp. 57-67 (1982).
- [16] Hummel, Robert A., "Connected component labeling in image processing with MIMD architectures," in *Intermediate-level Image Processing*, M. J. B. Duff, (Eds.). Bonas, France: Academic Press (1986).
- [17] Hummel, Robert and Kaizhong Zhang, "Dynamic processor allocation for parallel algorithms in image processing," *Proceedings of the Optical and Digital Pattern Recognition session of the SPIE Conference on EO-Imaging, SPIE Vol. 754*, pp. 268-275 (January, 1987).
- [18] Flynn Hummel, S., "SMARTS — Shared-memory Multiprocessor Ada Run Time Supervisor," Ph.D. Thesis, New York University, NYU Tech Report #495, (December, 1988).
- [19] Kruskal, C., L. Rudolph, and M. Snir, "The power of parallel prefix," *Proceedings of the International Conference on Parallel Processing*, pp. 180-185 (Aug. 1985).
- [20] Gottlieb, A., B. Lubachevsky, and L. Rudolph, "Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors," *ACM Transactions on Programming Languages and Systems* 5, pp. 164-189 (April, 1983).
- [21] Flynn Hummel, S. and E. Schonberg, "Low-Overhead Scheduling of Nested Parallelism," *To appear in IBM Journal of Research and Development*, (1991).