The pSather 1.0 Manual

December 2, 1994

David Stoutamire[1]

International Computer Science Institute
1947 Center Street, Suite 600
Berkeley, California 94704

1. Direct email correspondence to `davids@icsi.berkeley.edu`

pSather 1.0 - Tutorial

The pSather 1.0 Specification

pSather 1.0 - Tutorial

EXAMPLES

Gate examples

Gates can be used as *futures*:

```
g: := #GATE{FLT}; -- Create a gate with queue of FLTs
g :- compute;
...
result := g.get;
```

The statement "`g :- compute;`" creates a new thread to do some computation; the current thread can continue to execute. It is suspended only if the result is needed and not available.

Obtaining the first result from several competing searches:

```
g :- search(strategy1);
g :- search(strategy2);
g :- search(strategy3);
result := g.get;
g.clear;
```

When one of the threads succeeds, its result is enqueued in `g`. After retrieving this result the other threads are terminated by "`g.clear`".

Fork, par, and parloop examples

In the following code A and B can execute concurrently. After both A and B complete, C and D can execute concurrently. E must wait for A, B, C and D to terminate before executing.

```
par
    par
        fork A end;
        B
    end;
    fork C end;
    D
end;
E
```

This code applies frobnify using a separate thread for each element of an array.

```
par
    loop   e: := a.elt!;
        fork e.frobnify end
    end
end
```

The same code can be written:

```
parloop e: := a.elt! do
    e.frobnify
end
```

This code applies phase1 and phase2 to each element of an array, waiting for all phase1 to complete before beginning phase2 (barrier sync):

```
parloop e: := a.elt! do e.phase1 end;
parloop e: := a.elt! do e.phase2 end
```

This code does the same thing without iterating over the elements for each phase:

```
parloop e: := a.elt! do
    e.phase1;
    cohort.sync;
    e.phase2;
end
```

Because local variables declared in the parloop become unique to each thread, the explicit barrier sync in the last example is useful to allow convenient passing of state from one phase to another through the thread's local variables, instead of using an intermediate array with one element for each thread.

It is possible to clear all threads at once and resume execution with the thread outside the par statement in this way:

```
answer:FOO; -- This is outside the par so it is shared
parloop e: := a.elt! do
    result: := e.search;
    if ~void(result) then
        answer := result;
        cohort.clear -- This clears all threads in the par, even the thread executing the 'elt!'
    end
end
-- Now answer can be used
```

Locking examples

The following code computes the maximum value in an array by using a thread to compute the max of each subrange:

```
global_max:FLT:=a[0]; -- Outside the par body so this is shared
parloop
    i:=0.upto!(a.size-1, 1024);-- Step by 1024. Each thread works on 1024 elements
do
    m:FLT:=a[i]; -- This is local to each body thread
    loop
        m:=m.max(a.elt!(i,1024)) -- Yield 1024 elements starting at index 'i'
    end
    lock cohort then -- Obtain mutual exclusion
        global_max:=global_max.max(m)
    end
end
```

This code implements five dining philosophers:

```
chopsticks : := #ARRAY{MUTEX}(5);
loop chopsticks.set!(#MUTEX) end;
parloop
    i: := 0.upto!(4);
do
    loop
        think;
        lock chopsticks[i], chopsticks[(i+1).mod(5)] then
            eat
        end
    end
end
```

Memory consistency examples

The following incorrect code may loop forever waiting for `flag`, print “`i is 1`”, or print “`i is 0`”. The code fails because it is trying to use `flag` to signal completion of “`i:=1`”, but there is no appropriate synchronization occuring between the forked thread and the body thread. Even though the thread terminates, the modification of `flag` may not be observed because there is no import in the body thread. Even if the modification to flag is observed, there is no guarantee that a modification to i will be observed before this, if at all.

```
i:INT; -- These variables are shared
flag:BOOL;
par
    fork
        i := 1;
        flag := true;
    end;
    loop until!(flag) end-- Attempt to loop until change observed
    #OUT + “i is” + i + ’\n’
end
```

The code below will always print “i is 1” because there is no race condition. An export occurs when the forked thread terminates, and an import occurs when par completes. Therefore the change to ‘i’ must be observed.

```
i:INT; -- This is a shared variable
par
    fork i:=1 end;
end
#OUT + “i is” + i + ’\n’
```

Locality examples

This code creates an unfixed object and then inserts it into a table, taking care that the insertion code runs at the same cluster as the table:

```
table.insert(#FOO @ any) @ where(table);
```

To make sure the object is fixed at the same cluster as the table, one could write

```
loc: := where(table);
table.insert(#FOO @ loc) @ loc;
```

or

```
fork @ table(where);
    table.insert(#FOO)
end
```

To recursively copy only that portion of a binary tree which is near,

```
near_copy:NODE is
    if near(self) then return #NODE(lchild.near_copy, rchild.near_copy)
    else return self
    end
end
```

Spread example

On a machine with one processor per cluster, spread might be used to implement a spread vector with subranges distributed across the clusters:

```
spread class SPREAD_VEC is
    attr subrange:VEC; -- There is one vector on each cluster; this is a pointer to it.
    ...
    plus(b:SAME):SAME is
        res: := new;
        parloop do @ clusters!; -- Idiom recognized by compiler; implemented as a broadcast
            res.subrange := subrange + b.subrange
        end;
        return res
    end
    ...
end
```

This implementation will not perform well on architectures with more than one processor per cluster; a more portable class would be written

```
spread class DIST_VEC is
    attr chunks:ARRAY{VEC}; -- There are one or more chunks per processor
    ...
    plus(b:SAME):SAME is
        res: := new;
        res.chunks := #(size);
        parloop do @ clusters!; -- Execute on each cluster
            parloop i: := chunks.ind!; do -- Fork for each chunk
                res.chunks[i] := chunks[i] + b.chunks[i]
            end
        end;
        return res
    end
    ...
end
```

A Code Example

This program takes a distributed vector of FLTs, computes the maximum value, and prints how many times this value appears in each of the chunks constituting the distributed vector.

```
class MAIN is
    main is

        -- 'vec' is a Distributed VECtor, composed of many VECs, each
        -- of which is an array of FLT (IEEE single precision).
        vec:DVEC:= ... -- Read vec in from a file

        -- 'big' is a local FLT variable which will be shared by all
        -- threads in this routine.  'big_lk' will be used to guarantee
        -- atomicity during the max computation.
        big::= - FLT::infinity;
        big_lk::=#MUTEX;

        -- 'counts' is an array of the results, one element per chunk
        counts::=#ARRAY{INT}(vec.num_chunks);

        parloop
            -- The code between 'parloop' and 'do' is executed serially
            -- as in an ordinary loop.

            ch::=vec.chunks!;       -- Iterate over each chunk
            idx::=0.up!;            -- Find a place to put the result

        do @ where(ch);
            -- A thread is forked for the code following the 'do'.
            -- We want it to execute at the location of the chunk.

            -- 'm' is the maximum value seen by this thread.  Because it
            -- is declared inside the parloop, it is private to this thread.
            -- Similarly, 'ct' is private to this thread.
            m::= - FLT::infinity;
            ct::=0;

            -- Scan the local chunk for the maximum and update the count
            loop
                el::=ch.elt!;          -- Iterate over all elements
                if m=el then ct:=ct+1;
                elsif m<el then m:=el; ct:=1;
                end;
            end;

            -- Now update the global maximum.  The 'if' isn't strictly
            -- necessary, but avoids synchronization overhead for the
            -- common case.  The lock guarantees atomicity in case another
            -- thread has the same idea.
            if big<m then
                lock big_lk then big:=big.max(m); end;
            end;
```

```
            cohort.sync;     -- Wait for all threads (barrier sync)

            -- If the local max is the same as the global max, then our
            -- count is legitimate, so store it.  Otherwise, the count
            -- should stay zero as the array was initialized.
            if m=big then
                counts[idx]:=ct;
            end;

        end; -- parloop

        -- Print out what has been discovered
        #OUT + “The maximum value is: “ + big + ‘\n’;
        loop i::=0.for!(vec.num_chunks);
            #OUT + “Chunk “ + i + “ has “ + counts[i] + “ instances” + ‘\n’;
        end;

    end; -- routine main
end; -- class MAIN

class DVEC < $DIST{VEC} is
    dir:DIRECTORY{VEC}; --DIRECTORY is a spread class, lists chunks
    num_chunks:INT;
    create(num_chunks:INT):SAME is ... end;
    chunks!:VEC is ... end; -- Iterate over chunks
    ...
end; -- class DVEC
```

The pSather 1.0 Specification

INTRODUCTION

Sather is an object oriented language that supports highly efficient computation and powerful abstractions for encapsulation and code reuse. pSather is a set of extensions to (serial) Sather to allow scalable reusable software units on shared or distributed-memory architectures. This document builds upon and assumes knowledge of the Sather 1.0 specification, available at http://http.icsi.berkeley.edu/Sather.

The pSather syntax is specified by grammar rules expressed in a variant of Backus-Naur form, following the Sather 1.0 specification. The full pSather 1.0 grammar is formed by the union of the grammar rules from the Sather 1.0 specification and those in this document.

This specification differs in many important ways from the previous “pSather 1.0” as described in ICSI TR-93-028. Many of the changes were prompted by conflicts with changes in serial Sather, while others are generalizations or streamlining of the previous pSather constructs.

PARALLEL EXTENSIONS TO SATHER

Threads

In serial Sather there is only one thread of execution; in pSather there may be many. Multiple threads are similar to multiple serial Sather programs executing concurrently, but threads share variables of a single namespace. A new thread is created by executing a *fork*, which may be either a fork statement (page 16) or an attach (page 14). The new thread is a *child* of the forking thread, which is the child's *parent*. pSather provides operations that can *block* a thread, making it unable to execute statements until some condition occurs. pSather threads that are not blocked will eventually run, but there is no other constraint on the order of execution of statements between threads that are not blocked. Threads no longer exist once they *terminate*. When a pSather program begins execution it has a single thread corresponding to the main routine.

Serial Sather defines a total order of execution of the program's statements; in contrast, pSather defines only a partial order. This partial order is defined by the union of the constraints implied by the consecutive execution order of statements within single threads and pSather synchronization operations between statements in different threads. As long as this partial order appears to be observed it is possible for an implementation to overlap multiple operations in time, so a child thread may run concurrently with its parent and with other children.

Gates

Gates are powerful synchronization primitives which generalize fork/join, mailboxes, semaphores and barrier synchronization. Gates have the following unnamed attributes:

- A locked status (*unlocked*, or *locked* by a particular thread);
- In the case of GATE{T}, a queue of values which must conform to T, or
- In the case of the unparameterized class GATE, an integer counter;
- A set of *attached* threads. Every pSather thread is attached to exactly one gate[1]. Attached threads may be thought of as producers that enqueue their return value (or increment the counter) when they terminate.

One way that threads can be created is by executing an *attach*:

attach ⇒ *expression* :- *expression*

1. Even the main routine, which may be considered dynamically inside an implicit par (page 16), is attached to a gate; the program terminates when all threads have terminated.

The left side must be of type GATE or GATE{T}. If the left side is of type GATE{T}, the return type of the right side must conform to T. If the left side is of type GATE, the right side must not return a value. The right side must be a routine call or a call on a bound routine object.

The left side is evaluated; then any subexpressions of the right side are evaluated left to right. If the gate is locked by another thread, the executing thread is suspended until the gate becomes unlocked; then a new thread is created to execute the specified routine. The new thread is attached to the gate. When the forked routine returns, the thread terminates, detaches itself from the gate, and enqueues the return value or increments the counter.

Gates can be used to signal threads about changes in their queue or attached threads. For example, a thread that wants to continue when a gate's queue has a value can use get to wait without looping.

In addition to having threads attached, gates support the following operations:

Table 1: Basic operations on GATE{T} [and GATE]

Signature	Description	Exclusive?
create:SAME	Make a new unlocked GATE{T} with an empty queue and no attached threads.	N/A
size:INT	Returns number of elements in queue [GATE: returns counter].	No
has_thread:BOOL	Returns true if there exists a thread attached to the gate.	No
set(T) [GATE::set]	Replace head of queue with argument, or insert into queue if empty. [GATE: If counter is zero, set to one.]	Yes
get:T [GATE::get]	Return head of queue; do not remove from queue. Blocks until queue is not empty. [GATE: Blocks until counter is nonzero.]	Yes
enqueue(T) [GATE::enqueue]	Insert argument at tail of queue. [GATE: increment counter.]	Yes
dequeue:T [GATE::dequeue]	Block until queue is not empty, then remove and return head of queue. [GATE: Block until counter nonzero, then decrement.]	Yes
clear	Clears the gate, emptying the queue [GATE: clearing counter], detaching any threads, setting their SYS::cleared flag and possibly raising an exception; see page 19.	Yes
sync	Blocks until all attached threads are also blocking on a call to sync or have terminated. It is an error for sync to be called by a thread not attached to this gate.	Yes

Some gate operations are *exclusive*: these lock the gate before proceeding and unlock it when the operation is complete. Only one thread may lock a gate at a time. The exclusive operations also perform imports and exports significant to memory consistency (page 19). Gates also support the operations listed on page 18.

par *and* fork

A threads may be created by an attach, but may also be created with the *fork statement*, which must be syntactically enclosed in a *par statement*:

fork_statement ⇒ fork *statement_list* end

par_statement ⇒ par *statement_list* end

When a fork statement is executed, it forks a *body thread* to execute the statements in its body. Local variables which are declared outside the body of the innermost enclosing par statement are shared among all threads in the par body. The rules for memory consistency apply to body threads, so they may not see a consistent picture of the shared variables unless they employ explicit synchronization (page 19).

Each body thread receives a unique copy of every local declared in the innermost enclosing par body. When body threads begin, these copies have the value that the locals did at the time the fork statement was executed. Changes to a thread's copy of these variables are not observed by other threads. Iterators may not occur in a fork statement unless they are within a loop which is also enclosed by the fork.

The thread executing a par statement creates a GATE object and forks a thread to execute the body. This newly created thread as well as all threads created by fork statements syntactically in the par body are attached to this same gate. The gate may be accessed by the special expression cohort, which must be syntactically enclosed in a par statement:

expression ⇒ cohort

The thread executing a par statement blocks until there are no threads attached to the cohort gate. This will occur even if threads exit the par body due to an enclosing loop terminating because of an iterator, a return statement, or an exception. yield is not permitted within a par body.

The *parloop statement* is syntactic sugar to make convenient a common parallel programming idiom:

parloop_statement ⇒ parloop *statement_list* do *statement_list* end

This is syntactic sugar for:

```
par
    loop
        statement_list
        fork
            statement_list
        end
    end
end
```

Locks

Synchronization objects control the blocking and unblocking of threads. GATE, GATE{T} and MUTEX are special synchronization objects which provide a mutual exclusion lock. Only one thread may *acquire* the lock at a time. The thread then *holds* the lock until it *releases* it. A single thread may acquire a lock multiple times concurrently; it will be held until a corresponding number of releases occur.

Locks may be safely acquired with the *lock and try statements*:

lock_statement ⇒ lock *expression { , expression }* then *statement_list* end

try_statement ⇒ try *expression { , expression }* then *statement_list [* else *statement_list]* end

The type of all expressions must be subtypes of $LOCK (page 18). The thread executing the lock statement is said to be *locking on* the listed locks. The statement list following the then is called the *lock body*. A lock statement guarantees that all listed locks are atomically acquired before the body executes. The try statement does not block, and if it fails to acquire all the locks, it will instead execute the statements following the else, if present.

Because all listed locks are acquired atomically, deadlock can never occur due to concurrent execution of two or more lock statements with multiple locks, although it is possible for deadlock to occur by dynamic nesting of lock statements or in other ways.

The implementation of lock statements also ensures that threads that can run will eventually do so; no thread will face starvation because of the operation of the locking and scheduling implementation. However, it is frequently good practice to have threads whose programmer supplied enabling conditions are never met in a given run (exceptional cases) or are not met after some time (alternative methods). One thread in an infinite loop can prevent other threads from executing for an arbitrary time.

All locks acquired by the lock statement are released when the lock body stops executing; this may occur due to executing the last statement of the body, termination of a loop by an iterator, a return, or an exception. Objects may also be unlocked before exiting a lock body by an unlock statement:

unlock_statement ⇒ unlock *expression*

An unlock statement must be syntactically within a lock body, and must not occur within a loop statement unless it is also within the body of a lock statement also within the loop. It is a fatal error if the expression does not evaluate to a $LOCK object which is locked by the enclosing lock statement.

$LOCK *Classes*

All synchronization objects subtype from $LOCK.The following primitive $LOCK classes are built-in:

- GATE{T} and GATE (page 14)
- MUTEX: a simple mutual exclusion lock. Two threads may not simultaneously lock a MUTEX. MUTEX is a subset of the functionality of GATE, and may require less overhead.

In addition to these primitive $LOCK classes, some synchronization classes return $LOCK objects to allow different kinds of locking. The concrete type of the returned object is implementation dependent:

- GATE and GATE{T} define empty, not_empty, threads and no_threads. These operations are not exclusive. Other gate operations are listed on page 14.

Table 2: Operations on GATE{T} [and GATE] that return $LOCK objects

Signature	Description
empty:$LOCK	Returns a lock which blocks until the gate is lockable and the gate's queue is empty [GATE: counter zero]; then the gate is locked. Holding this lock does not prevent the holder from making the queue become not empty [counter become nonzero].
not_empty:$LOCK	Returns a lock which blocks until the gate is lockable and the gate's queue is not empty [GATE: counter nonzero]; then the gate is locked. Holding this lock does not prevent the holder from making the queue become empty [counter zero].
threads:$LOCK	Returns a lock which blocks until the gate is lockable and there is some thread attached to the gate; then the gate is locked. Holding this lock does not prevent the completion of attached threads.
no_threads:$LOCK	Returns a lock which blocks until the gate is lockable and there are no threads attached to the gate; then the gate is locked. Holding this lock does not prevent the attachment of threads by the holder.

- RW_LOCK is used to manage reader-writer synchronization. If rw is an object of type RW_LOCK, then a lock on rw.reader or rw.writer blocks until no thread is locking on rw.writer. Threads may upgrade from a reader to a writer without deadlock.

Memory consistency model

Threads may communicate by writing and then reading variables or attributes of objects. All assignments are atomic (the result of a read is guaranteed to be the value of some previous write); assignments to value objects atomically modify all attributes. Writes are always observed by the thread itself. Writes are not guaranteed to be observed by other threads until an *export* is executed by the writer and a subsequent *import* is executed by the reader. Exports and imports are implicitly associated with certain operations:.

Table 3: Operations with implicit imports and exports

An import occurs:	An export occurs:
In a newly created thread	In parent thread when a child thread is forked
When exiting a par statement (children have terminated)	In the child on thread termination
When entering the body of a lock or try statement	When exiting the body of a lock or try
On exclusive GATE and GATE{T} operations (page 14)	On exclusive GATE and GATE{T} operations

This model has the property that it guarantees sequential consistency to programs without data races.

The SYS *Class*

pSather extends the SYS class with these functions:

cleared:BOOL	False when a thread begins; this becomes true if clear is called on the gate the thread is attached to. This value is unique to each thread.
trap_clear:BOOL	This variable is unique to each thread. It is true when a thread begins; if true at the time the gate is cleared, the thread encounters a CLEARED_EX exception. Otherwise the thread continues but will observe cleared to be true.
defer	Hint to scheduler that this is a good time to preempt this thread.
import	Execute an import operation (page 19).
export	Execute an export operation (page 19).

Exceptions

Exceptions in a par body will not be raised outside the body until all threads created in the body have terminated; exceptions in a lock body will not be raised outside the body until all associated locks have been released.

As in serial Sather, it is a fatal error if an exception occurs in a thread which is not handled within that thread by some protect statement. There is an implicit handler for CLEARED_EX on all threads which executes nothing; clearing a gate will by default (see page 19) kill any attached threads by raising a CLEARED_EX which they do not catch.

DISTRIBUTED EXTENSIONS

Locality and the Cluster Model

The pSather executing environment defines a number of *clusters*. At any time a thread has an associated *cluster id* (an INT), its *locus of control*. Every thread also has a *fixed* or *unfixed* status. Unless modified explicitly, the locus of a fixed thread remains the same throughout the thread's execution. The locus of control of an unfixed thread may change at any time. When execution begins, the main routine is unfixed[2]. The unfixed status or fixed locus of control of a child thread is the same as the status or locus of its parent at the time of the fork.

The locus of a thread may be explicitly fixed or unfixed for the duration of the evaluation of an expression:

expression ⇒ *expression* @ (*expression* | any)

An expression following the '@' must evaluate to an INT, which specifies the cluster id of the locus of control the thread will be fixed at while it evaluates the preceeding expression. It is a fatal error for a cluster id to be less than zero or greater than or equal to clusters (page 19). If any is given instead of a cluster id, the thread will be unfixed while it evaluates the expression. The '@' operator has lower precedence than any other operator.

2. The fixed or unfixed status of the main routine and other characteristics of unfixed threads and objects may be changed by compiler options. It is a legitimate implementation to have "unfixed" threads and objects behave as though they were fixed at the point of creation.

The '@' notation may also be used to explicitly fix or unfix body threads of fork and parloop statements.

fork_statement ⇒ fork @ (*expression* | any) ; *statement_list* end

parloop_statement ⇒ parloop *statement_list* do @ (*expression* | any) ; *statement_list* end

Although for these constructs the location expression may appear to be within the body, it is really part of the header. The location expression is executed before the threads are forked and is *not* part of the body threads.

All reference objects have a unique associated cluster id, the object's *location*, as well as a fixed or unfixed status. When a reference object is created by a fixed thread, its location will be the same as the locus of control when the new expression was executed. If the creating thread was unfixed, the object will be unfixed and its location may change at any time. A reference object is *near* to a thread if its current location is the same as the thread's locus of control, otherwise it is *far*.

There are several built-in expressions for location:

Expression	Type	Description
here	INT	The cluster ID of the locus of control of the thread.
where(*expression*)	INT	The location of the argument. It is a fatal error for the argument to evaluate to void. If the argument is a value or spread type it returns here.
near(*expression*)	BOOL	True if the argument is not void and near. If the argument is a value or spread type it returns ~void(x).
far(*expression*)	BOOL	True if the argument is not void and far. If the argument is a value or spread type it returns false.
clusters	INT	Number of clusters. Although a constant, may not be available at compile time.
clusters!	INT	Iterator which returns all cluster ids in some order. This may bring the compiler's attention to opportunities for optimization.

It is also possible to assert that particular reference objects remain near at run-time:

with_near_statement ⇒ with *identifier_list* near *statement_list* [else *statement_list*] end

The *identifier_list* may contain local variables, arguments, and self; these are called *near variables*. When the with statement begins execution, the identifiers are checked to ensure that all of them hold either near objects or are void. If this is true then the statements following near are executed, and it is a fatal error if the identifiers stop holding either near objects or void at any time. Unfixed objects will not change location while they are held by near variables. It is a fatal error if some identifiers hold neither near nor void and there is no else. Otherwise, the statements following the else are executed.

Spread Objects

A *spread class* replicates object attributes and array elements across all clusters.

class ⇒
  [spread | value | external] class *uppercase_identifier*
  [{ *parameter_declaration* {, *parameter_declaration* } }] [*subtyping_clause*]
  is [*class_element*] { ; [*class_element*] } end

An object of a spread class has a distinct instance of each attribute and array element on each cluster. Attribute and array accesses read or write only the instance on the cluster of the locus of execution; therefore the instance on a particular cluster can be accessed with the idiom "*spread_var*@*location*". The new expression in a spread class is used just as in a reference class; there is a single integer argument if the class has an array portion.