pSather

David Stoutamire

(UCB, ICSI)

The pSather group at ICSI:

Jerome Feldman, Claudio Fleiner, Michael Philippsen, Boris Vaysman

Prior work by Chu-cheow Lim, Stephan Murer and others

Outline

- pSather goals
- What we want from this retreat
- Serial Sather
- Threads and synchronization
- Gates and locking
- Distributed constructs
- Consistency model
- Conclusion

Goals of pSather 0.2

- Retained safety and efficiency goals of serial Sather
- Extended serial Sather to multiple threads
- Language support for safe synchronization
- Shared memory model with language support for distributed nature of threads and objects
- Proof of concept on CM-5; applications

pSather 1.0

Extended Sather 1.0 using insights from pSather 0.2:

- Took advantage of features of Sather 1
- Refined syntax and semantics
- Relaxed memory consistency
- Allows (but does not require) dynamic migration
- Portable, including networks of workstations

pSather Libraries

Parallelism should be implicit in libraries with *apparently* serial interfaces, such as:

- Distributed matrix and vector classes
- Distributed priority queues, hash tables
- Parallel I/O

What we want from the retreat

pSather 1.0 implemented using threads on SMPs

pSather 1.1 aimed at performance on real applications and distributed systems

- What applications? (*Splash 2*)
- What platform? (*NOW*)

Sather 1.0

- Object-oriented language with long development
- Simple and safe - encourages reuse and maintainability of code
- Efficient - must be as efficient as C++
- Portable and non-proprietary
- Active user base, newsgroup

Sather 1.0 continued

- **Code reuse:** *parameterized classes, object-oriented dispatch, multiple inheritance,separate implementation and type inheritance, iterators*

- **Safety:** *statically checked (contravariant) typing, garbage collection, exception handling, preconditions, postconditions, and class invariants*

Sather 1.1

- Minor polishing of Sather 1.0
- Aggressive interfaces to C and Fortran (BLAS)
- pSather integrated into specification

Data parallel threads

Fork a thread for every element:

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

Or:

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

Gates

- Generalization of *fork/join*, *barriers*, *semaphores*, *futures*, *condition variables*, *mailboxes*
- Gates provide synchronization almost always required with data production-consumption

Gate Example

Use as a future:

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

Barrier Example

Wait for phase1 to complete before phase2:

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

This does the same thing:

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

Locking

It is hard to write deadlock and starvation free code.

Exceptions, iterator termination, and returning values make everything worse!

Locking contd.

Dining philosophers:

```
parloop i::=0.upto!(4) do -- A thread for each philosopher
      loop
            think;
            lock
                  when left(i), right(i) then eat
            end
      end -- loop
end -- parloop
```

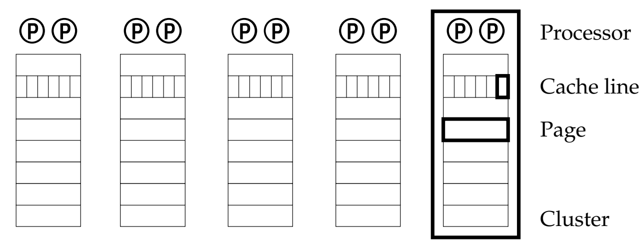
Locking can be complicated, but locks are always properly unlocked on exiting. Supports reader/writer locks and disjunctive locking.

The Cluster Model

pSather provides a shared memory abstraction, but...

...to be useful, language must reflect NUMA realities.

Many levels of locality:



Location

Every thread has a location, and this may be fixed:

```
compute @ cluster_id
```

Or unfixed:

```
compute @ any
```

Objects may be fixed or unfixed, and inherit location from the creating thread.

There is no way to migrate objects *on demand*; system may migrate unfixed objects.

Near and Far

Language dynamically distinguishes near vs. far:

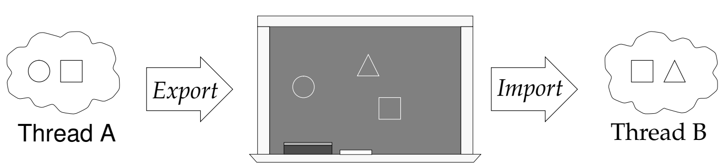
```
near_copy:NODE is
    if near(self) then
        return #(lchild.near_copy, rchild.near_copy)
    else
        return self
    end
end
```

Consistency Model

All assignments are atomic.

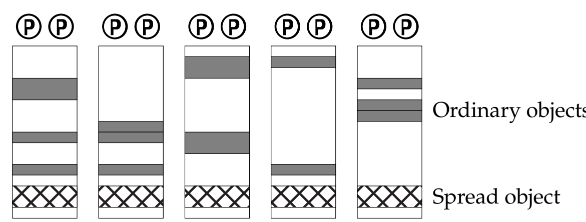
Relaxed memory consistency model, an intermediate position between

- Hardware-centric *release consistency* (Gharachorloo et al.)
- Software-centric *data-race-free-1* (Adve and Hill)



Spread objects

- Located at the same address on each cluster
- Avoids directory lookup



Conclusion

pSather 1 is simpler yet more powerful than 0.2.

- SMP implementation in 1.0 compiler
- Beginning distributed implementation (1.1)
- Looking to test expressiveness on real applications

Information on Sather and pSather available at

http://www.icsi.berkeley.edu/Sather

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

Signature	Description
create:SAME	Make a new unlocked GATE{T} with an empty queue and no attached threads.
size:INT	Returns number of elements in queue [GATE: returns counter].
has_thread:BOOL	Returns true if there exists a thread attached to the gate.
set(T) [GATE::set]	Replace head of queue with argument, or insert into queue if empty. [GATE: If counter is zero, set to one.]
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.]
enqueue(T) [GATE::enqueue]	Insert argument at tail of queue. [GATE: increment counter.]
dequeue:T [GATE::dequeue]	Block until queue is not empty, then remove and return head of queue. [GATE: Block until counter nonzero, then decrement.]
clear	Clears the gate, emptying the queue [GATE: clearing counter], detaching any threads, setting their SYS::cleared flag and possibly raising an exception.

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.

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
On exiting a par statement (children have terminated)	By a thread on termination
On entering the body of a lock statement	On entering an unlock, or exiting a lock
On exiting exclusive GATE operations.	On entering exclusive GATE operations
On completion of a sync statement	On initiation of a sync statement

SYS methods

defer	Tell scheduler that this is a good time to pre-empt this thread.
import	Execute an import operation
export	Execute an export operation

:

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. This may bring the compiler's attention to opportunities for optimization.

Spread Objects cont.

Distributed vector example:

```
spread class SPREAD_VEC is
    attr subrange:VEC;
    ...
    plus(b:SAME):SAME is
        res: := new;
        parloop do @ clusters!; -- This is a broadcast
            res.subrange := subrange + b.subrange
        end
        return res
    end
end
```

In general, construct efficient distributed classes by using existing, efficient serial classes

General Distributed Objects

```
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!;            -- A broadcast
            parloop i: := chunks.ind!; do -- Thread per chunk
                res.chunks[i] := chunks[i] + b.chunks[i]
            end
        end;
        return res
    end
    ...
end
```

Memory Consistency Example

This *may not* work:

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

This *will* work:

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