Introduction

Parallel programming is often viewed as much harder than serial programming. Unfortunately the general perception is correct, parallel programming is difficult. The parallel features of Sather, collectively known as pSather, have been under development and experimental use at ICSI for several years and have recently been integrated into the general distribution. Some of the simplest features are supported on all platforms, even those with no parallel capabilities. This is mainly to provide compatibility with parallel platforms. Unfortunately, to make any interesting use of pSather or of this tutorial you will need access to a platform that supports pSather threads. A current list of these can be found on the pSather home page http://www.icsi.berkeley.edu/~sather/psather.html. Because there is no generally accepted portable thread interface, it is a significant effort to port pSather to a new parallel system and there are separate compilation flags for each of these. The platforms that have the most extensive history are various Solaris implementations and these are used for the examples here.

There are many approaches to parallel programming languages and dozens of proposals for parallel (or concurrent) OO languages. The design goals of pSather are the same as for the serial portions of the language. The most important criteria are execution efficiency, safety and reusability. The project has, from the outset, been based on the belief that Sather's constructs and methodology will be even more valuable in the more challenging domain of parallel computing. Like Sather, pSather is an explicit imperative object-oriented language.

The underlying model remains one large shared address space, although there are also features for placement of objects and threads for greater efficiency. The current **cluster** model assumes that the underlying system has a specified number of clusters, each of which supports the concurrent execution of multiple threads. The presumption is that the communication cost across clusters is much greater than within a cluster.

The motivating example is a network of SMP workstations. At ICSI we have been using a low-latency Myrinet/active-message network of quad processor Sparc10s as a prototype. There is also an ethernet implementation, but the latencies make this impractical for all but the most loosely coupled parallel programs. An implementation on the Meiko CS-2 allows us to test scalability to larger systems. There also may be significant memory latency differences within a cluster. These are not accounted for in the cluster model, but will be in the **zone** model of David Stoutamire's UCB thesis.

We believe that pSather effectively supports all the standard styles of parallel programming such as data parallel, task parallel, actor, etc. and this tutorial will provide some indication of how we think this should go. The focus of the language design was to provide convenient constructs for writing libraries of parallel and distributed objects.The most fundamental additional extension to Sather is the notion of multiple **threads** of execution.Threads are not first-class objects in pSather. This and other design decisions are discussed in a variety of papers accessible through the web page. Although there are facilities for mutual exclusion (at most one thread active in an object), these are not mandatory and many pSather programs depend upon multiple active threads. In general, the language attempts to provide the library designer and application programmer with tools for creating systems that achieve maximum performance, but also supports parallel computing styles that concentrate on simplicity and safety.

The pSather language is relatively stable and we expect little change over the next few years. However, there will be a continuing growth of documentation, papers, and examples. The best way to get the latest story is to visit the web page http://www.icsi.berkeley.edu/~sather/psather.html. In particular, the theses of Claudio Fleiner and David Stoutamire are not yet available but we will cite them frequently here.

The parallel extensions to Sather that constitute pSather are currently divided into three conceptually distinct extensions. The Threaded Extension (Section 17) is the most basic and introduces the par, fork and parloop constructs. Programs that require no synchronization can be coded using only these mechanisms.The Synchronization Extension (Section 18)is the most complex and includes the abstract class $LOCK and its descendants and the various forms of the lock statement. It includes the powerful GATE construct (Section 18.4)that combines the semantics of futures, events, and locks found in other languages. The Distributed Extension (Section 19) adds the placement annotation, @, and the with...near statement. These do not affect the semantics of a correct program, but can support greatly improved performance on distributed platforms such as a network of workstations. Our current plan is that Sather 1.2 will extend these ideas with the Zone Extension, which supports a much richer model of data and thread locality. A final Section 20 covers a number of advanced topics including user defined locks.

The Threaded Extension

17.1 Introduction

The most basic parallel extension to serial Sather adds only threads and the ability to fork them and wait for a collection of threads to complete execution. Although this extension provides no capabilities for coordinating access to shared data or waiting for events, a wide range of problems can be coded in this extension. ICSI compilers starting with 1.1 will compile programs using the threaded extension on all platforms; on platforms without thread support, the various threads will each be serially run to completion. We will start with some simple "Hello World" variants and then discuss a range of medium sized tasks that use only the threaded extension capabilities.

17.1.1 Hello Worlds

We will give two example "Hello World" programs to illustrate some of the basic pSather constructs. The simplest one uses the two most basic pSather constructs, par and fork. Here there are two explicit threads forked within the par. end statement

```
class MAIN is
   main is
      par
            fork #OUT+"Hello World #1" +'\n' end;
            fork #OUT+"Hello World #2" +'\n' end;
      end;
   end;
end; -- class MAIN
```

Each fork statement establishes a separate thread of control executing the enclosed statement. Fork statements can only occur within par. end brackets. Control passes to the statement following the end after the termination of all of the statements forked within the brackets. There is no fixed order of execution of the forked statements; even this tiny program might print the greetings in either order. Runnable threads in pSather execute in undertemined order. A thread can relinquish control, either explicitly through a SYS::defer statement, or because it is blocked. As long as there is an upper bound to the running time of each thread, the implementation guarantees that every thread that remains ready to run will eventually do so - a weak form of fairness.

This program can be compiled and run on any Sather platform in the usual way, but of course can only execute in parallel on systems with thread support. The list of supported pSather platforms can be found in the web page. There is a whole range of command line options for pSather, the most important of which is the platform. It is easiest to develop code on a single processor, even if it is to eventually run on a large system. For Solaris, there are currently two platform designators:

solaris/smp -- this treats a uni- or multi-processor workstation as one shared memory

solaris/tcpip -- this is a slow but stable platform employing several workstations

The command line to compile the initial program, stored as hello.sa, could be:

```
cs -solaris/smp -o hello hello.sa
```

Here is another version of "Hello World"; this introduces the third and last construct in the Threaded Extension, parloop. Using fork statements within a par. end bracket, one can fork different threads of any variety; if all of the threads are essentially the same, it is often more convenient to use a parloop.

```
class MAIN is
   main is
      parloop i:=0.upto!(3) do
            #OUT + ("Hello World # " + i + '\n')
      end;
   end;
end; -- class MAIN
```

Here there will be separate threads forked for each value of i and this value will be passed as a parameter to the respective thread. When compiled appropriately, the second example will print its greetings, but probably not in order. By enclosing the entire greeting in parens, we assure that the entire string is constructed before any output is started; this normally assures that the greetings will not be garbled together even on an SMP with simultaneously active threads.

Suppose that we wanted to be sure that the Hellos were printed in order without serializing the computation. One way would be to use a global ARRAY{STR} indexed by the same loop variable i. Each thread could insert its greeting and the array printed after the parloop completes. You might want to try this. Notice that this depends on multiple threads writing to a single object; there is no danger here because each thread writes to a disjoint piece of the array. This fixes the output order, but says nothing about the order of execution of the threads. In general there is no guarantee on the execution order and interleaving of pSather threads and this can be a significant problem in constructing reliable code. The synchronization extensions discussed in the next section help significantly, but it we still usually rely upon known patterns.

17.2 Realistic Examples Using Threads

The ICSI pSather group has developed various small and medium sized examples for use in testing and benchmarking the system. Several of these use only the par, fork, and parloop constructs of the threads extension and these can serve as additional examples at this elementary level. One obvious benchmark is matrix multiplication. The following is the complete code for the parallel part of the

```
class MMULT is
   mult(a,b,c:MATRIX{FLT}) is
      parloop x::= b.rows.times!;
      do
             loop y::= b.rows.times!;
               a[y,x]:= 0.0;
               loop z::= c.cols.times!;
               a[y,x]:= a[y,x]+b[z,x]*c[y,z];
               end;
             end;
      end;
   end;
end;
```

example - just a parloop over the rows of the first multiplicand. This is, of course, very naive compared to the sophisticated loop parallelization of modern FORTRAN compilers. For a variety of reasons beyond the scope of this tutorial, pSather and other OO languages take an entirely different approach to the problems of parallelization. From the OO perspective, low-level optimizations should be encsapsulated in classes. For standard matrix/vector operations, Sather can do no better than the well developed FORTRAN packages such as BLAS and does not try. ICSI Sather compilers from 1.1 on provide a FORTRAN interface which is for just such purposes.

From the perspective of the tutorial, the interesting thing is how many standard benchmarks can be coded using only the threaded extension. Here is another typical example from the control loop of a program to compute heat flow expansion over time.<Fleiner thesis>

```
   heat_step is
      cl_size:=cluster_size;   -- just for formating
      t::=t1;
      t1:=t2;
      t2:=t;
      parloop p::=cl_size.times!;
      do
            loop x::=((cols*p)/cl_size).upto!(cols*(p+1)/cl_size-1);
               loop y::=rows!;
                  t1[x,y]:=heat_of(t2,x,y);
               end;
        end;
      end;
   end;
```

Here the parloop runs over the number of clusters in the current platform, breaking up the computation into that number of pieces by columns of the underlying heat_of array. Again, because there are no interactions and each thread treats a disjoint subarray, life is easy. This simplistic way of partitioning the problem is appropriate for a platform where clusters are separate processors sharing physical memory. We will later see some more sophisticated partitioning. The language allows parloops to be nested and this can be used to spread a problem over clusters and then further divide the work among threads on each cluster.

The Synchronization Extension

18.1 Barrier Synchronization and sync

The simplest synchronization primitive is the sync statement, which causes all the threads forked within a par or parloop block to suspend until every thread in the block has terminated or is also executing a sync command. This is called barrier synchronization in the computational science literature for obvious reasons. One important feature of the sync statement, in contrast with stopping and restarting the threads, is that the participating threads all retain their state. After all threads in the block meet the sync criterion, the unterminated ones are resumed. One basic use of the sync construct is to assure that all threads in a parloop are set up before any of them execute; the dining philosophers example in Section 18.3 does this. We will see another use of the sync command in the chunk maximum example of the next section. Realistic examples can be found in the benchmark programs <Fleiner thesis>. The next two sections introduce more sophisticated control mechanisms. The sync construct just has a collection of threads wait for one another. The lock statement of the next section provides various ways to control access to data among threads, but is still passive. Active signaling between threads requires the GATE construct, described in Section 18.4.

18.2 The lock Statement and the MUTEX Class

All of the other synchronization constructions in pSather use various forms of the lock statement. We will start with the simplest form of the lock construct and gradually increase the complexity of both the form of lock statement and the classes of objects to be locked. The basic form with a single unconditional mutual-exclusion lock suffices for many cases and should be considered first. In the following example, we suppose that a large vector has been stored as a number of separate "chunks" using a library class DVEC. We talk later about the design of such distributed classes, which uses the convention of an iterator, chunks!, for iterating the separate pieces of a distributed object, such as a DVEC. The code below is from an artificial example where the task is to print both the maximum value in the DVEC vec and the number of instances of it in each chunk. This makes prototypical uses of a MUTEX big_lk, the lock statement and the sync command.

```
class MAIN is
   main is
      vec:DVEC:=#DVEC(3,4); -- 3 chunks of size 4
      counts:ARRAY{INT}:=#(vec.num_chunks); -- max of chunk

      big::= -FLT::infinity;-- initialize
      big_lk::=#MUTEX;

      vec.init;-- kludge for example only
      parloop
          ch::=vec.chunks!;-- Iterate over each chunk
          idx::=0.up!;-- Places for result of each thread
          do;
             m::= -FLT::infinity;
             ct::=0;
             loop
               el::=ch.elt!;-- Iterate over elements,1
               if m=el then ct:=ct+1
               elsif m<el then m:=el; ct:=1
               end;
             end;
          if big<m then
             lock big_lk then big:=big.max(m) end
           end;
          sync;-- Wait for all threads
          if m=big then counts[idx]:=ct end;
    end; -- parloop

    #OUT + "The maximum value is: " + big + '\n';
    loop i::=0.for!(vec.num_chunks) ;
          #OUT + "Chunk "+i + " has "+counts[i]+" instances"+'\n';
    end;
  end; -- main
end; -- class MAIN
```

The parloop forks a thread for each chunk of data and also provides a consecutive index for each thread/chunk pair. The variables ch, idx, m, and ct are all instantiated separately for each thread. Each thread computes the maximum value in its chunk and its multiplicity. The way we have chosen to calculate the global maximum uses a global variable, big, and the accompanying mutex, big_lk. After each thread computes its local maximum, it compares it with the current global maximum, big. If the local one is bigger, it should become the global maximum.

But there is a synchronization problem. It could happen that several threads simultaneously have a local maximum larger than the current global winner. Using the lock statement ensures that only one thread at a time will try to update big. An additional subtlety is that, even after acquiring unique access to big, a thread can not assume that its local maximum still exceeds the global one. Between the time a thread checks the relative values and when it acquires the lock, another thread might have changed big. This is the kind of problem that makes parallel programming tricky and there is no good way around it while preserving performance. A helpful heuristic is to think about a thread being interrupted indefinitely between any two statements. Another is to use Murphy's law - consider the worst thing another thread could do to the current computation.

Now, after each thread has tried its luck at being the global maximum, they all wait at the sync statement barrier for the others. When all are finished, the true global maximum has been found and the various threads can output the number of occurrences of this in their chunk. Because we used a sync statement, all of the threads resume with context (values of local variables) retained, including the local count, ct, which is the desired result. Rather than print the results in execution order, they are all stored in a global array which is printed after the parloop. We will see many more examples of the MUTEX, sync and lock constructs. Notice that the pSather lock statement automatically releases locks upon completion, eliminating another common source of deadlock. Section 18.3.4 discusses when one would choose a read-write lock, RW_LOCK, instead of a MUTEX to control access to a global object.

18.2.1 Memory Consistency, Round One

The synchronization extension of pSather plays another important role in the language, but one that usually remains implicit. What we have discussed above is how a MUTEX can keep two threads from making conflicting updates to the same object. A more subtle problem can arise when one thread updates a shared variable and one or more other threads want to use the new value; how can they know when the write has completed? Essentially the same problem arises in modern high performance processors; writes to memory are not guaranteed to be observed in the same order in which they occur in the program. This improves cache performance, but is difficult to program and is the subject of considerable research<Stoutamire thesis>. For pSather, the critical requirement is to supply a clean assignment semantics that the user can rely upon and that all compilers must implement. Part of the semantics is that all assignments are atomic, it will never happen that only part of an write command is executed. This is obvious for single word assignments, but Sather immutable objects can be of any size and it is essential that one never has an assignment that is only partially carried out. Furthermore, Sather guarantees that any update executed by a thread will always be immediately observable by that thread. What is trickier is how to specify exactly when other threads will know about such an update.

To understand this, we need to define notions of import and export. Both of these are available as explicit commands in the SYS class, but using them directly is unusual. An example using explicit import and export statements is shown in Section 18.4.2. In essence, an export operation suspends the current thread until all of the updates that it has done are publicly known. An import operation suspends the executing thread until all publicly known updates are made in its context. It follows that an update is guaranteed to be seen by all threads that do an import after the updating thread has done an export. Since there is no global clock, "after" means that the code executes an explicit or implicit synchronization command between the export and the corresponding import. It turns out that this can be implemented efficiently on most platforms, <Fleiner thesis> tells all.This often doesn't help the programer that much because the he/she would still seem to need to know when to issue these import and export commands. However, pSather already has various synchronization constructs like the par statement of the previous section and the lock statement described in this section. The key idea is to associate implicit imports and exports with the appropriate pSather constructs. These are spelled out in a chart in Section 3.3.7 of the language description. The important point for now is that one can not assume anything about the relative timing of various threads that is not specified by some synchronization constructs. But, given the explicit synchronization, your intuition about memory consistency is preserved. We will see several examples of how this works.

18.3 Conjunctive Locking

One of the fundamental issues in synchronization is the treatment of multiple locks. A very common cause of deadlock is when two or more threads compete for multiple locks. It can be shown that no deadlock can arise from this situation if all the locks are linearly ordered and threads always acquire locks in this order. Sometimes the pSather programmer will need to carefully arrange the fixed order, but most cases (and a good bit more) are handled automatically by the conjunctive lock construct. This is illustrated with the classical dining philosophers problem. The setting is a round table with one chopstick between each two diners. Since two chopsticks are needed for eating, the diners need some way to manage the required resources.

```
class MAIN is
   attr chopsticks:ARRAY{MUTEX}; -- need two adjacent ones to eat

   main is
      chopsticks:=#(7);
      loop chopsticks.set!(#MUTEX) end;

      parloop i::= chopsticks.ind!;
      do  sync; -- wait for all to start
              philosopher(i)
      end; -- parloop;
   end; -- main

   philosopher(k:INT) is
     loop 3.times!;
            lock chopsticks[k], chopsticks[(k+1).mod(7)]
            then #OUT + ( "philosopher  " + k + "  is eating.\n" )
            end; -- lock
     end; -- loop
   end; -- philosopher
end; -- MAIN
```

This example is complete and can be run and modified. We use a separate MUTEX for each chopstick and it is natural to make these an array. The main routine initializes this array and starts the parloop which forks off a separate thread for each philosopher. The sync command ensures that all the threads are established before any start running; this is often needed for fairness. The conjunctive lock is in the code for each philosopher. Each one tries to conjunctively lock the mutex to its left and its right and, when it succeeds, prints its message and ends the lock statement, freeing the locks. The pSather lock implementation guarantees the absence of deadlock (or livelock) among the competing threads and also a weak form of fairness. No thread will compete indefinitely for an achievable set of locks without eventually winning and getting to execute. Although it is not illustrated here, there is also an explicit unlock statement that can be used to release one of the conjunctive locks before the entire body completes. More details on all this can be found in <Fleiner thesis>. Several of our later examples rely on conjunctive locks so we won't bother you with more at this point.

18.3.1 Read-Write Locks, three kinds

So far we have seen only mutual exclusion (MUTEX) locks. For many applications, it is very useful to have other forms of locking, such as the classical read-write locks. The idea is to allow many threads to lock as a reader, but to restrict modification (writer) to one thread at a time. This is another classic concept and is captured in pSather by the classes RW_LOCK, WR_LOCK, and FRW_LOCK. There are also a number of other kinds of lock objects in pSather and all of them are subtypes of the built-in abstract class $LOCK.

Consider again the example of Section 18.3.2 that computed the number of occurrences of the maximum value in each chunk of a distributed vector. The global maximum, big, was protected by a lock big_lk:MUTEX and accessed using the statement:

```
if big<m then  lock big_lk then  big:=big.max(m) end  end;
```

The first conditional did not need to be locked because the test is atomic. Now suppose instead that we needed to calculate the number of occurrences of the second largest value rather than the maximum. The obvious code for this again has each chunk thread compare its local values against both big and the second value, say next. The problem is that this multiple test is not atomic and can't be done unprotected by a lock. But we don't need exclusive access to big and next for checking purposes, just a guarantee that no changes will occur during our multiple tests. Enter the WR_LOCK construct. Suppose that we modify the example of 18.3.2 to have a second FLT, next, and a WR_LOCK, next_lk. Then the global update code fragment becomes:

```
lock next_lk.reader then
   if (m>big) or (m>next) then update:=true end
end;
if update then
   lock next_lk.writer  then  -- test again inside lock
      if m>big then  next:=big; big:=m
      elsif (m>next and m<big) then next:=m end
   end;
end; -- if update
```

Actually, testing just m>next would suffice in this case, but we will ignore this. Not only does the class WR_LOCK subtype from $LOCK but it defines two methods, reader and writer, that have return type $LOCK. There are several other methods with return type $LOCK and they play an important role in pSather. Here the reader lock protects the two tests from changes. Any attempt to write-lock the variable next_lk will wait until all reads have completed. The local BOOL update is set if updates are needed. We must exit the reader lock before attempting to get the writer lock to avoid deadlock.

There are two other variants of reader-writer locks defined in pSather; they differ in the relative priority given to readers and writers awaiting the same lock. For a RW_LOCK, readers are given priority. An WR_LOCK gives priority to writers and a (fair) FRW_LOCK treats readers and writers the same. Which of these is best for the example above? Writing, in this case, should have priority because this will sometimes eliminate extra lock-and-modify execution by other threads. Therefore a WR_LOCK is best. For our realistic example, we present the first of three stories involving tuple spaces.

18.3.2 Tuple Space, Round 1

One very general parallel construct is a tuple-space. A tuple space is a simple construct for coordinating parallel and often distributed computations. Processes can insert a tuple into the space and can request tuples from the space in one of four ways. The request can either leave the tuple or erase it and can also be blocking or non-blocking. The most famous language based on this idea is Linda <Carriero, N. and D. Gelerntner, How to Write Parallel Programs, MIT PRESS, 1990>, which includes full pattern matching on tuples as a fundamental construct. We will examine a simpler case where matching only happens on the first element of a tuple, which must be a STR. Our example follows that of Foster <Designing and Building Parallel Programs, Addison-Wesley, 1995>, p.152. Foster's terminology is a bit confusing. The final 'p' in readp and takep suggest that we have a predicate for the nonblocking cases. The two versions of 'take' commands can be thought of pulling tuples into worker objects and out of the tuple space. Foster's actual usage is even more confusing, rd for read and in for take. Our version also allows only one tuple type in each tuple-space; this is consistent with Sather's strong typing. Thus:

class TSPACE{TT<$TUP} is...

Foster includes five basic operations:

insert(tup:TT) -- put a new tuple into the space, duplicates Ok

read(s:STR):TT -- blocking read, wait for match to appear

readp(s:STR):TT -- return void if no match

take(s:STR):TT -- blocking move, wait then erase and read

takep(s:STR):TT -- non-blocking move, return void if no match

In round 1, we consider the non-blocking case for which we need only insert, takep and readp. For concreteness suppose that the tuples are to be stored in an A_LIST. In practice, one would use a more efficient container class. Then the code is very straightforward; the built-in WR_LOCK construct provides the basic functionality needed for writers (insert, takep) and readers (readp). Any number of readp operations can run in parallel, but insert and takep modify the tuple space and thus require a lock spacerw.writer. Using a WR_LOCK maximizes the chance that a tuple will be present when requested. Using a FRW_LOCK instead would not change the code, but would have the semantics of strictly obeying the arrival order of operations. However, this arrival order is usually not consistent in a multiple processor system; small variations in load or initial conditions can change the order. Both readp and takep search for a tuple with a matching key. The search in takep iterates over indices using ind! because A_LIST::remove_index(INT) requires the index.

```
abstract class $TUP is
   t1:STR;  -- all tuples have a STR key
end; -- class $TUP
-----------------------------------------------------------------
class TSPACE{ TT < $TUP } is
   attr b:A_LIST{TT};
   attr spacerw:WR_LOCK;  -- insert is mutator, read*, take* visitors

   create:SAME  is
      res::=new;
      return(res.init);
   end;

   init:SAME is
      b:=#;
      spacerw:=#;
      return self
   end;

   insert(tup:TT) is
      lock spacerw.writer then
              b.append(tup);
      end; -- spacerw lock;
   end;

   readp(s:STR):TT is
      el:TT;
      lock spacerw.reader then
             loop el:=b.elt!; if el.t1=s then break! end
             end;
      end; -- lock spacerw
      if el.t1=s then return el else return void end;
   end;

   takep(s:STR):TT is
      i:INT;
      el:TT;
      lock spacerw.writer then  -- mutator
            loop i:=b.ind!;
              if b.aget(i).t1=s then
                 el:=b.aget(i);
                 b.remove_index(i);
                 break!
               end
             end;
      end; -- lock spacerw.writer
      if el.t1=s then return el else return void end;
   end;
 end; -- class TSPACE{TT<$TUP}
```

18.3.3 Disjunctive Locking

There are two orthogonal dimensions of functionality in pSather locking, the various subtypes of $LOCK and the different forms of the lock statement. We have seen how conjunctive locking can be employed to solve hard problems in coordinating access to multiple resources. Disjunctive locking is our current solution to a wide range of thread termination and related problems. The motivation for the current design is discussed in <Fleiner thesis>. Briefly, there appears to be no good way to safely terminate a thread from the outside. Threads can cleanly self-destruct, but only if they are executing. Now it is frequently convenient to have threads suspended waiting for events that might or might not occur. This is standard well-known problem with a variety of proposed solutions. For pSather, disjunctive locking is by far the best. As always, the construct is finding a variety of other applications.

The protoypical use of disjunctive locking would be in a procedure that was waiting for some event or new data that might not materialize. At a higher level, a control program, probably the one that forked the waiting thread, knows when the waiter should terminate. We can't present a real example yet, because we first need to introduce gates, which are the basic pSather constructs for events (among other things). Schematically, the code would look like:

```
loop
   lock
   when terminate then return
   when event then action
   end;
end; -- loop
```

The general construct also allows one to prepend a boolean condition to any of the when branches. Each time through the loop these conditions are all evaluated and any branch whose prepended condition is false will be disabled. The idea of disjunctive guarded commands appears in several languages, most prominently Ada <Barnes, J.G.P., Programming in Ada, Addison -Wesley, 1994>. It is natural to incorporate these features into the pSather lock statement because, in a truly parallel environment, an event that is not locked might well change between the time it is triggered and when it gets handled. As we will see in the next section, pSather provides mechanisms that simultaneously resume a thread that is waiting for some event and grant it a lock on the corresponding $LOCK object.

Thus there are four increasingly flexible parallel coordination mechanisms in pSather. The simplest, barrier synchronization, was described in Section 18.1. Mutual exclusion mechanisms and conjunctive locking were discussed in Sections 18.2-3. Three variations on reader-writer locks were presented in Section 18.3. All of these and the event coordination constructions of the next chapter can be used disjunctively as described in this section.

18.4 GATE and GATE{T} classes

The Sather gate construct is the most complex and powerful feature of the language extensions and will be discussed in this section and the next one. Our hope is that most application programmers will be able to do fine using only the constructs described above, once the pSather libraries are in place. But for writing efficient and reusable libraries and for novel applications, the power tools can make all the difference. The Sather 1.1 specification discusses gates in conjunction with the classes FUTURE and ATTACH and the abstract class $ATTACH. These are minor variations on the basic concept and can be ignored for tutorial purposes. We will present the various aspects of gate functionality separately and then give examples of how they can be combined. The tables in section 3.3.5 of the 1.1 specification are complete and accurate, but are not very helpful in learning about gates. After you understand this tutorial, the specification is a much more compact reference.

18.4.1 Gates as Synchronizers and Queues

We will first describe the more general GATE{T} construct and then define how it specializes to the dataless GATE version. In early versions of pSather, gates were called monitors but the functionality has changed very little. Monitors were used, as gates can be, to collect results returned by functions forked as separate threads. Since several such threads could return values to the same monitor(gate), there needed to be some discipline for how the multiple values were stored. The FIFO queue was an obvious choice and, as often happens, the queue functionality of gates came to be used much more widely than anyone anticipated.

An object of type GATE{T} can be created and used rather like any other paramterized Sather container, but has a number of features built-in. One important feature is that the usual queue operations: set(T), get:T, enqueue(T), and dequeue:T are guaranteed to be atomic. In addition, any attempt to get or dequeue a value from an empty queue will suspend until a value is present, providing a simple "futures" capability. Objects of type GATE define the same operations, but with meanings appropriate for queues that have only counts, not a collection of values. This is all described adequately in the specification.

As a first example, we point out that a subset of the typed gate functionality is just what is needed for a simple message passing mechanism. Consider the following class.

```
class PORT{MSG} is
   attr channel:GATE{MSG}; -- GATE is a queue

   create:SAME is
      res::=new;
      res.channel:=#;
      return res   end;

   send(datum:MSG) is channel.enqueue(datum) end;

   receive:MSG is  return channel.dequeue  end; -- blocks when empty
end; -- class PORT
```

This implements a typed message port with the usual properties. The send operation is non-blocking and the receive blocks until there is a message and atomically pulls it off the queue. We will see some more elaborate message channels in later examples.

Since GATE subtypes from $LOCK, we could have used a gate instead of a mutex in the examples of sections 18.2 and 18.3. This is not normally useful in itself, but becomes quite powerful when combined with the other features of gates. Our current focus is on the queue functionality and this contains two functions that have return type $LOCK, empty and not_empty. Notice that GATE is a subtype of $LOCK and also has methods that whose return types are subtypes of $LOCK. This supports a variety of useful constructs, as subsequent examples will illustrate.

Lock statements can include conditions gate.empty or gate.non_empty. A GATE{T} satisfies the empty condition when there are zero elements in its queue. An untyped GATE satisfies the empty condition when its counter equals zero. There is also a non-locking function size:INT which returns the size of the queue.

The synchronization features of gates can be used to build other classes with similar capabilities. As a simple example, here is a class PSTACK{T}. The PSTACK class is a parallel computing interface to the standard Sather array-based stack. It guarantees atomicity of operations and also has a pop method that suspends when called on an empty stack. It is worth examining how the gate functionality supports this. It is often effective to replace a serial error condition (pop on an empty stack) with a wait in the executing thread, but this can make debuggin, caveat coder more difficult to detect errors. An error in serial code can turn into a deadlock or starvation situation in the multi-threaded case and this is a lot harder to detect, caveat coder.

.

```
class PSTACK{T} is                              -- pop waits if empty
   private attr s:A_STACK{T};
   private attr ct:GATE;

   create: SAME is
      res ::= new;
      res.s := #A_STACK{T};
      res.ct:=#GATE;
      return(res);
   end;

   is_empty: BOOL pre ~void(self) is return(ct.size = 0) end;

   push(e: T) pre ~void(self)  is
      lock ct then
        s.push(e);
        ct.enqueue;
      end;
   end;

   pop: T pre ~void(self) is
      lock ct.not_empty then
        ct.dequeue;
        return(s.pop);
      end;
   end;

   top: T pre ~void(self) is
     lock ct then
       assert ~ is_empty;
       return(s.top)
      end;
   end;

   size: INT  pre ~void(self) is return(ct.size) end;
end; -- class PSTACK{T}
```

All of the required functionality is conveniently packaged using one untyped gate, ct. The push and pop methods are both destructive and require mutual exclusion. The gate ct does this, but it is also used as a blocking counter of the size of the stack. Recall that enqueue increments the counter of an untyped gate and that dequeue decrements a non-zero counter and blocks on zero. The is_empty predicate can just use ct. The push method does the appropriate enqueue; notice that this is while ct is locked against other threads. The pop method starts with a lock on ct.not_empty; this blocks on an empty stack and atomically locks ct as soon as some other thread has pushed a value on to it. All of the coordination required for multiple users and for conjunctive and disjunctive locking is handled by the run-time lock manager. In general, the programmer just needs to understand that pSather has a very flexible event and lock mechanism, but only for a restricted set of event types under $LOCK. In our example, we were able to represent the events and locks required for the task by pSather primitives and the result was clean and efficient code. The top method is shown as non-blocking for contrast, a real library would probably have it block also.There will be additional examples below. For wizards, there is a way to define custom classes that subtype from $LOCK. This can be done in Sather, but requires some understanding of the lock manager details and is discussed in Section 20.4, Advanced Topics.

Our next example illustrates a pSather solution to another classic and important parallel computing problem, producers and consumers. We describe code for one producer and one consumer, the general case is essentially the same and would make a good exercise. In order to help visualize the whole mechanism, this example is present as a monolithic program rather than encapsulated classes. Another good exercise would be to use the PORT class described earlier in this section.

```
class MAIN is
   attr channel:GATE{INT};-- queue aspect of GATE exploited
   attr prod_cnt:GATE;-- used to count live producer

   main is
      channel:=#;
      prod_cnt:=#;
      par
             prod_cnt.enqueue; fork producer  end; -- one producer
             fork consumer end;
      end
   end; -- main

   producer is
      loop i::=3.upto!(8);
              channel.enqueue(i*i)
      end;
      prod_cnt.dequeue;   -- done, decrement producer count
   end;

   consumer is
     loop
            -- disjunctive lock, here mutually exclusive branches
            lock
              when channel.not_empty then #OUT+channel.dequeue+ '\n'
              when channel.empty,  prod_cnt.empty  then  return
            end; -- lock
     end -- loop
   end; -- consumer
 end; -- MAIN
```

Here there are two gates employed. A typed gate, channel, is the queue of data between producers and consumers. An untyped gate, prod_cnt, will count the number of active producers. As in the stack example we map a task condition - no active producers - onto an event handled by the locking mechanism - prod_cnt.empty. The main program just forks a producer and a consumer while adding 1 to the prod_cnt. Our producer sends its squares to the communication channel and signals that it is finished by decrementing prod_cnt. The consumer is a bit more complex and is our first real example of a disjunctive lock. The first disjunct is the normal case where data in the communication channel is printed. The classical problem is that the consumer function has no way to know when production has ceased. There are various unattractive solutions such as putting special sentinel values in the data stream. Here the second disjunct waits for both the absence of data on the channel and the signal that there are no active producers. Again, this only works out so nicely because we mapped problem conditions onto pSather primitives. For our last example of this section, we revisit the tuple space problem and add blocking reads and moves in an efficient way.

18.4.2 Tuple Space, Round Two

In section 18.3.2 we described a reduced version of the tuple space example from Foster in which only the non-blocking read and take operations were implemented. The more general case with blocking read and take is considerably more complex because we want to avoid any busy waiting or polling. Our solution follows a standard pSather pattern with each blocking read becoming a suspended thread waiting on some event. If we assume that blocked reads are relatively infrequent, a good solution is to treat these specially and leave the unblocked case efficient. The expanded TSPACE class has a wish list, wish, that holds elements of type WISH, each of which represents one or more blocked read/take requests. The class WISH is quite simple. A wish that comes from an take request should not be added to the tuple space when it appears and this is noted by the BOOL, claimed. The GATE que is where wishing threads wait for fulfillment.

```
class WISH{TT} is
   attr key:STR;
   attr claimed:BOOL;-- a "take" will snarf this wish
   attr que:GATE{TT};-- wisher waits with w.que.not_empty

   create(s:STR):SAME is
      res::=new;
      res.key:=s;
      res.claimed:=false;
      res.que:=#;
      return res
   end;
end; -- class WISH
-------------------------------------------------------------------
abstract class $TUP is
   t1:STR;   -- all tuples have a STR key
end; -- class $TUP
```

The full tuple space implementation is captured in the class TSPACE, which appears in the next three code blocks. Most of the code from the earlier version is preserved. In addition, the MUTEX wishlk controls mutual exclusion to wish. Since there is no guarantee that a blocked read/take command will ever be satisfied, there should be some way to clean up a tuple space and the GATE die is used to signal waiting threads that it is time to quit.

```
class TSPACE{ TT < $TUP } is
  private attr b:A_LIST{TT};
  private attr wish:A_LIST{WISH{TT}};
          -- WISH~key:STR,claimed:B,que:GATE{TT}
  private attr spacerw:RW_LOCK;--insert,take* mutate space;read* visits
  private attr wishlk:MUTEX;
  private attr die:GATE;

  create:SAME  is
      res::=new;
      return(res.init);
  end;

  init:SAME is
      b:=#;
      wish:=#;
      spacerw:=#;
      wishlk:=#;
      die:=#;
      return self
  end;

  insert(tup:TT) is
      i:INT:=0;
      w:WISH{TT}:=#("*");    -- initialize;

     lock wishlk, spacerw.writer then --can change both space and wish
        loop i:=wish.ind!;
           if wish.aget(i).key = tup.t1 then
              w:=wish.aget(i);
              break!
           end
        end;
        if w.key=tup.t1 then  -- if no matching wish, skip all this
           w.que.set(tup);  -- enables waiting read/take threads
           wish.remove_index(i); -- zap the wish list entry
           if w.claimed then return end -- don't insert tuple
        end; -- if w.key=tup.t1
        b.append(tup);          -- OK, tuple gets added to space
      end; -- lock wishlk, spacerw
  end; -- insert
```

The non-blocking readp and takep methods do not need to change at all from our previous solution. The blocking versions, read and take, each start with a call to the non-blocking counterpart. If the request is found, there is no loss of efficiency. Similarly for the insert method; if there are no unsatisfied wishes, the code is the same as the base case. The extra work is all in the wish list, and the synchronization problems can also be isolated there. Since any of insert, take, or read can modify the wish list, access to it is controlled by the MUTEX wishlk. Consider first the read method. If there is no match in the tuple space, a search of the wish list is done. If there is also no unclaimed match on the wish list then a new wish entry must be created and appended. A matching wish would be claimed if an unsatisfied take call will remove the tuple-to-come. All of this is expressed within the wishlk.

```
 read(s:STR):TT is
   v:TT:=void;
   el:TT:=readp(s);
   if ~(el=v) then return el end;

   w:WISH{TT}:=#("*");-- initialize to non-match;
   lock wishlk then
     el:= readp(s);        -- check again, might have been inserted
     if ~(el=v) then return el end;
     loop w:=wish.elt!;
        if w.key=s and ~w.claimed then break! end
     end;
     if ~(w.key =s) or w.claimed  then w:=#(s); wish.append(w)  end;
   end; -- lock wishlk

   lock   -- only if wishing
   when w.que.not_empty then return w.que.get
   when die.not_empty then return void
   end; -- lock
end;  -- read

readp(s:STR):TT is
   el:TT;
   lock spacerw.reader then
     loop el:=b.elt!;
        if el.t1=s then break! end
     end;
   end; -- lock spacerw
   if el.t1=s then return el else return void end;
end;  -- readp
```

The final code segment of read implements the wait for a matching tuple using a disjunctive lock statement. The second when branch waits for a global signal to die. The first branch is the normal case and relies upon the properties of the pSather GATE{T} construct. Notice that a wish list entry has three attributes: key:STR, claimed:BOOL, and que:GATE{TT} where TT is the tuple type. The GATE construct supports multiple threads waiting for a value and this is just what is needed here. The first when branch can be taken as soon as a value (here a tuple) is assigned to w.que and this value is returned as the result of the original blocked read.

The additional code required for the blocking take command is quite similar to this. The search for an unclaimed matching tuple is similar, except that the take command will, if it finds an unclaimed matching wish, mark the wish as claimed. If the take command does add a new wish, this must obviously be marked claimed. The disjunctive lock that implements waiting is also the same as in read; all wishes are removed by their fulfilling insert command so nothing extra needs to be done here.

```
    take(s:STR):TT is
      v:TT:=void;
      el:TT:=takep(s);
      if ~(el=v) then return el end;

      w:WISH{TT}:=#("*");-- initialize to non-match;
      lock wishlk then
        el:=takep(s);          -- check again, might have been inserted
        if ~(el=v) then return el end;
        loop w:=wish.elt!;
          if w.key=s and ~w.claimed then break! end
        end;
        if w.key =s and ~w.claimed then w.claimed:=true   -- claim it
        else w:=#(s); w.claimed:=true; wish.append(w) end; --add wish
      end; -- lock wishlk

      lock
      when w.que.not_empty then return w.que.get -- insert removes wish
      when die.not_empty  then return void
      end; -- lock
   end;

   takep(s:STR):TT is
      i:INT:=0;
      el:TT; -- non match;
      lock spacerw.writer then-- mutator
        loop i:=b.ind!;
           if b.aget(i).t1=s then
              el:=b.aget(i);
              b.remove_index(i);
              break!
           end
        end;
      end; -- lock spacerw.writer
      if el.t1=s then return el else return void end;
   end;

   done is die.set end;

end; -- class TSPACE{TT<$TUP}
```

It is the insert method that involves most of the extra complexity for dealing with the wish list, when present. The first loop searches for an existing wish index with the same key; if there is none, insertion reverts to our previous case. If there is a matching wish, this insertion is its answer and this is indicated by setting w.que to have a value of the tuple being inserted. Now, one or more threads are waiting for this GATE{TT} to be set and they will all be enabled. The current wish is removed and, if the tuple has been claimed, then the insert routine returns. If all of the waiting operations were read then the current tuple will not have been claimed and will be added to the main tuple store by the last line of code. Notice that insert conjunctively locks both the tuple space and the wish list to prevent either from being accessed during this operation. The only other method is done which sets the GATE:die as the signal for waiting threads to return. This completes the code for the tuple space example from the perspective of functionality. There will be a third round of consideration of this task when we discuss performance and the distributed extension in Section 19.2.

18.5 GATES and attached threads

Both the typed and untyped gate classes have another area of functionality that interacts with the queue and $LOCK properties described in the previous section. We consider first the typed example In certain programming styles, it is common to fork a (value returning) function linked to a variable that will eventually hold the result of the forked function, in the 'future'. The obvious semantics is that any access to such a future value will block until there is a value present. Historically, the original motivations for the pSather gate (originally monitor) construct was very much concerned with future values. In our terminology, a thread was 'attached' to the gate that would receive its value. We considered it important to allow multiple threads to be attached to the same gate and this led to the idea of a typed gate as a queue of values. It was also clear that, in a true parallel environment, retrieving a future must be atomic. The remaining three methods of the gate (and $ATTACH, etc.) classes: gate.has_thread:BOOL, gate.threads:$LOCK, and gate.no_threads:$LOCK deal with these issues.

The BOOL function, has_thread, is non-blocking and just gives a snapshot of whether there are any threads attached to this gate. The other two methods have return type $LOCK and participate in the full range of lock constructs. As expected, gate.threads will lock until gate has at least one thread attached and gate.no_threads will lock until there are no attached threads. Notice that in these cases, as well as empty and not_empty, the gate itself is also locked. The par ... end syntax was not included in early versions of pSather because this can be expressed in terms of the other primitives. You might want to try this; the answer will appear later in the text. There is a specific syntax for attaching a thread to a gate:

```
gate :- expression
```

The :- notation is intended to convey the notion of incomplete (future) assignment.

For untyped gates, everything is analogous. The procedure that is attached to an untyped gate must not return a value; on completion the counter of the untyped gate is incremented. In both cases, the calling code can either test if the method has returned (gate.size>0), block until this happens (gate.get or gate.enqueue) or lock on this condition (lock gate.not_empty ...). This provides a rich set of programming options for dealing with threads doing speculative computation., etc. For example, instead of enclosing a set of forked functions in a par ... end bracket, one could attach them all to some untyped gate, g, and then code: lock g.no_threads then end. The sync command discussed in Section 18.1 can also be used; executing a sync in a thread attached to a gate synchronizes with all other threads attached to the same gate.

There is currently much less use of the attach statement, futures, et. al. than we anticipated. We expect these features to be used heavily in parallel library classes, but these have just begun. Certainly the fork and parloop mechanisms are clearer when they apply. It is too early to know whether this trend will continue or whether we will start developing patterns that rely heavily on these more flexible mechanisms. We will present two artificial examples that give an idea of what can be easily done with these mechanisms. Suppose that we wanted to fork off a number of threads to try alternative ways of solving the same problem, this might be searching a data base or the internet or various approaches to an AI task. Our toy example just has different threads looking for a random number with a particular property; it is the control that is of interest.

```
class MAIN is
   attr num:GATE{INT};
   attr stop:BOOL;
   attr win:INT;

   main is
      i:INT;
      num:=#;
      stop:=false;
      loop i:=0.upto!(3);
              num :- worker(i);
      end;
      #OUT + num.dequeue + "   thread " + win  + '\n';
      stop:=true; SYS::export;  -- make this known
   end;  -- main

   worker(id:INT):INT  is
      RND::seed(31463*(id+43));
      loop  SYS::import;  -- stop is a global, import its value
              if stop then return(0) end;
              ans::=RND::int(0,10000);
              if ans.mod(71)=0 then win:=id; return ans end;
      end; --loop
   end; -- worker
  end; -- class MAIN
```

The GATE{INT}, num, will have the worker threads attached to it and their answers will be placed on its queue when available. The BOOL, stop, is a global signal for the other workers to stop after an answer has been found. The INT, win, is the thread number of the winning worker, this will usually vary even on a uniprocessor platform. The main program starts four workers and gives each its integer id. The forking thread is not blocked (as it would be with par ... end) and could do other computation. In this case it just waits (num.dequeue) for the first answer, sets the stop flag, exports it, and prints the first answer. The explicit export is needed to make the flag visible to all the workers.

The worker threads each initialize the random number generator differently and then loop until they find a number divisible by 71 or find out that another worker has done so. At the start of the loop, each thread does an explicit import to make sure that it has a current copy of stop. In this simple case the whole program terminates so stop isn't really needed. But termination is an important problem in general and we will return to this case in the next section.

The next example is a slight modification of this one that illustrates an important additional control option in pSather. Normally, any nested set of calls (whether forked or called in the same thread) must be completely unwound when the result is found and needs to be returned. The gate mechanisms make it easy to employ a kind of 'continuation passing' control technique that allows the direct return of a result to the top-level caller. It turns out that intermediate threads can terminate without causing any difficulty. The task is the same - several threads are given the task of finding a random number divisible by 71. The difference is that here we separate the attachment of threads from the return of answers We also fix a coordination bug in the previous example. Before we ret urned the id of the winning thread separately from the answer and could not be sure that the two values corresponded. Here we return a tuple (val, thread) and avoid that problem.

```
class MAIN is
   attr num:GATE{TUP{INT,INT}};  -- answer is (val,thread)
   attr dum:GATE;
   attr stop:BOOL;

   main is
      num:=#; dum:=#;
      stop:=false;
      i:INT;
      loop i:=0.upto!(3);
        dum :- worker(i,num);
      end;
      ans::=num.dequeue;
      #OUT + ans.t1 + "   thread " + ans.t2  + '\n';
      stop:=true; SYS::export;
   end;  -- main

   worker(id:INT, g:GATE{TUP{INT,INT}})  is
      RND::seed(31463*(id+43));
      ans:TUP{INT,INT}:=#(0,id);
      loop SYS::import;
        if stop then return end;
        try::=RND::int(0,10000);
        if try.mod(71)=0 then g.enqueue(ans.t1(try)); return end;
      end; --loop
   end; -- worker
end; -- class MAIN
```

There is one additional untyped gate, dum, which is a dummy used for attaching the worker threads. The worker code is changed so that it takes two parameters, its id and the gate, g, to which the answer should be returned and it now has no return value. When a worker finds an answer, it enqueues a tuple with the answer and its id on the gate given to it as a parameter and returns. The continuation idea isn't used here, but a worker could pass on the answer gate (and possibly task state) to another thread and terminate or do other work, for example return additional answers. Another possibility would be to have a worker that was attached to a typed gate enqueue results on the same gate before its final return.

A use of the attach construct in a real application can be found in Ben Gomes' forthcoming thesis. The task is to analyze a neural network graph and partition it segments that are placed on separate clusters.

18.5.1 Tasks, Actors, etc.

One popular style of parallel programing employs the metaphor of cooperating active agents. The pure form of this is given in the various Actor formulations <Agha,G., Actors, MIT Press,1988.>, but it occurs in many other forms. In pSather, it is fairly simple to create and manipulate active objects; the major issue is that the strong compile-time type system of Sather requires typed messages or run-time case statements. We will first present a general tasking package in pSather and then discuss how this could be modified and extended to support various paradigms of task programing. Since tasks should be free running once created, the central idea is to use the attach or :- statement to start a single thread within the object that is the actor or task. Tasks will communicate using objects of the simple PORT class, which was the first example in Section 18.4.1. The core tasking functionality will be encapsulated in a partial class TASK, listed in the table below. Various specific kinds of tasks will have their own class, each of which includes TASK; examples are given in the following code table. Since all different types of task must communicate, we define an abstract class $TASK, which expresses the common interface of all tasks. The abstract definition listed in the table is over-simplified in assuming that all messages are of type STR; we will discuss the general case shortly. Elements of the abstract signature include a routine, connect, executed in a receiving task, that connects the outport of some source to the inport of self and a reader of the outport that needs to be public for connect to work. Our design requires that each task type provide a subroutine, body, that is its main program. A discussion of the TASK class follows the table.

```
abstract class $TASK is
   connect(sender:$TASK);
   outport: PORT{STR};              -- real case is more general
   body;
end; -- class $TASK
-------------------------------------------------------------------
partial class TASK < $TASK is
   attr inport,outport: PORT{STR};
   private shared all:GATE:=#;          -- init should work soon.

   create:SAME is
      res::=new;
      res.inport:=#;
      res.outport:=#;
      return res
   end; -- create

   start:SAME is
      all:-body;
      return self
   end;

   stub body;
   send(datum:STR) is
      outport.send(datum)
   end; -- send

   receive:STR is                    -- blocks until data present
      return inport.receive
   end; -- receive

   connect(sender:$TASK) is          -- useage: receiver.connect(sender)
      sender.outport.channel:=inport.channel
   end;
end; -- class TASK
```

The partial class TASK will be included in the various specific task types. It has two public attributes, inport and outport, which are for now of type PORT{STR}. The private shared gate, all, is used to attach all tasks of a given type. The ability to initially assign to a shared is not yet realized. In pSather 1.1 there is no way to operate on these threads, but one can test or lock on whether there any threads attached to the gate, all. We have chosen to have a separate start routine so the create is straightforward. Recall that every task class must define a body, specified as a stub here. The start method attaches a thread executing the body to the shared gate, all,and returns self; this makes it convenient to create and start a task in one expression. The send and receive methods just wrap the same methods in the PORT class of Section 18.4.1. The connect method is used to set the output channel of a sender to be the input channel (a gate) of the receiver. A minimal example of task usage is provided in the following table.

```
class SINK  < $TASK is include TASK;
   body is
      s:STR;
      loop
        s:=receive;     -- waits for data
        #OUT + s;
        if s ="." then #OUT +'\n'; break! end
      end
   end;
end; -- class SINK
```

The SOURCE class is described below

```
class SOURCE < $TASK is include TASK create -> task_create;
   attr ok:GATE;

   create:SAME is
      res::=task_create;
      res.ok:=#;              -- the extra attr
      return res
   end;

   body is
      ok.get;                 -- wait for signal
      send("Hello ");
      send("World");
      send(".");
   end;end; -- class SOURCE
```

These are all tied together by the MAIN class

```
class MAIN is
   main is
      source:SOURCE:=#SOURCE.start;
      sink:SINK:=#SINK.start;
      sink.connect(source);
      source.ok.set;
   end;
end; -- class MAIN
```

There are two tiny task classes, SINK and SOURCE, each of which includes TASK and subtypes from $TASK, in the usual Sather fashion. The sink class consists only of the required body subroutine. It loops with a (blocking) receive command, included from TASK. If it sees a period in its input it prints a new -line and returns. The source task is a bit more complicated because it needs one extra attribute, ok:GATE, which will be its starting signal. The creation code is the standard Sather idiom, augmenting the create routine of the parent class. The body subroutine just waits for a starting signal and then sends three famous strings, again using a method included from TASK. Finally, there is a little main program that first creates and starts the two tasks, source and sink. We don't want the source to start generating before it is connected which is why we have it wait on ok. As an exercise, you might want to change SOURCE so its body waits for a starting message on its input channel instead. The main program just connects the sink to the source and signals ok. Because setting the gate ok is a synchronization operation, the implicit export/import (cf. Section 18.2.1) assures that the connect will be complete before source starts spewing text. Hopefully, people will rarely need to explicitly consider this kind of consistency issue.

18.5.2 Discussion and Extensions

The core task functionality just presented provides a framework for building up other task or actor based mechanisms. Even the existing code allows multiple sources to connect to a receiving task port. It is straightforward to allow multiple inport and outport channels, for example by using an ARRAY{PORT}. This is almost all that we need to capture the core programming model used by Taylor as the basis for his book<op.cit>, p.12-13. Taylor also allows references to ports to be passed in messages, supporting dynamic communication channels. Our task class already allows new assignments to the outport(s) of a task, but there is not yet a way to pass anything but strings in a message. The strong typing places constraints on how we can deal with various kinds of messages in a uniform way. We will outline one simple solution, retaining the simplification of a single inport and outport per task.

In this example, we will introduce a version of the task realization that can really be used to build significant systems. The main simplification in our first solution was requiring that each message be a string. We clearly want messages of many types and this must be reconciled with Sather's strong static typing. The saving grace is that all types of message should behave the same way at the level of task communication, differences only matter inside the body of tasks. This suggests that we define a general message type, MSG, with one attribute, datum, of type $OB; the tiny class for this is included in the next code table. The only change in the partial class TASK is that the three instances of STR are replaced by MSG and this modification is not shown. For our example, the class SOURCE requires no change at all and is also not repeated.

The most important change is in the abstract class, $TASK, which was just a hack in the previous example. We now can specify a compete and general interface for tasks. As we will see, all kinds of tasks can communicate using all kinds of messages with these general mechanisms. The abstract interface specifies the functionality needed by any task; any class that includes TASK and supplies a body will comply with the interface. It is easy to convert our previous example to use the more general MSG class. Consider the revised code for the SINK class, included as a code table. The new body code obviously needs a variable (m) of type MSG as well as the string from before. Recall that the data field of MSG is datum; the local variable md is needed because typecase takes an identifier. The code has two typecase branches, but ignore the second branch for now. The code in the first branch is unchanged, the new version of SINK just needs an extra level of indirection since the message isn't itself a string but has a string datum. If this were the only added functionality in the second task example, the main program would require no change.

```
class MSG is
   attr datum:$OB;

   create(dd:$OB):SAME is
      res::=new;
      res.datum:=dd;
      return res
   end;
end; -- class MSG
```

The revised $TASK abstraction is shown below

```
abstract class $TASK is
   connect(sender:$TASK);
   outport: PORT{MSG};             -- now more general
   create:SAME;
   start:SAME;
   send(datum:MSG);
   receive:MSG;
   body;
end; -- class $TASK
```

The SINK class is defined as

```
class SINK  < $TASK is include TASK;
   body is
      m:MSG; s:STR;
      loop
        m:=receive; md::=m.datum;
        typecase md
        when STR then s:=md;
           #OUT + s;
           if s ="." then #OUT +'\n'; break! end
        when RFCONN then
           connect(md.sender)
        end; --typecase
      end
   end;
end; -- class SINK
```

The RFCONN class is

```
class RFCONN is
   attr sender:$TASK;

   create(s:$TASK):SAME is
      res::=new;
      res.sender:=s;
      return res
   end;
end; -- class RFCONN
```

Our example could be carried out by the following main routine

```
class MAIN is
   main is
      source:SOURCE:=#SOURCE.start;
      sink:SINK:=#SINK.start;
      msg:MSG:=#(#RFCONN(source));
      sink.inport.send(msg);      -- unSathery, but simple
      source.ok.set;
   end;
end; -- class MAIN
```

However, now that we have messages of arbitrary type, we can fulfill our earlier promise to allow connections to be established by message. This is an important capability and can support systems with very dynamic connectivity. It is quite easy to do this with the mechanisms that we have established. Again we need a tiny class to define a message type, RFCONN (request-for-connection), shown in a code table above. It simply has one attribute of type $TASK, the sending task that needs to be connected to the inport of some receiver. We can now understand the second branch of the typecase statement in the SINK class. If a SINK object gets an RFCONN message, it executes its connect method which sets the outport of the task listed as the desired sender in the message. The revised main program illustrates how this might be used. Rather than directly link our one source to the one sink as before, the main program constructs and sends a message to do this. This is no improvement in the example, but should illustrate how connection requests can be passed and carried out within the design. This almost completes the requirements for Foster's tasks. He also requires that a task be placeable on different processors; the pSather mechanisms for this will be discussed in Section 19.2.

The tasking and rendezvous mechanisms of Ada are similar to the above with the major difference being guarded disjunctive method call. In fact, the disjunctive locking construct of Section 18.3.3 was partially based on the Ada guarded select statement. Rendezvous (both the caller and callee being blocking) does not seem like a good style for pSather's goals, but it is straightforward to achieve. One way to implement this in pSather would be to use explicit message passing, possibly with the task mechanisms of this section. An implementation closer to Ada syntax could be achieved by grouping the various accessible methods (corresponding to Ada entries) and associating each group with a MUTEX object. There would be one such MUTEX object for each select/accept block and a single MUTEX variable that corresponded to the current state (~ which select statement). A method would start by locking the controlling MUTEX. Recall that the natural state of a pSather object is passive; if no active threads are running the object is, in effect, waiting for a rendezvous. First consider the case where there is only one select statement in the Ada code, then a single MUTEX will ensure that exactly one of the entries executes at a time and callers of any others will wait for their rendezvous. A guarded select would be modeled by a guarded lock statement that exits if its guard fails. Multiple select statements would map to multiple MUTEXes, with the one corresponding to the current state being assigned to the controlling variable. There is a standard pSather coding style that is quite similar. A collection of methods is collected into a class without an active thread. Using classes can call these methods in either blocking or non-blocking (:-) mode. If mutual exclusion over some subset of methods is desired, these all start by locking on a shared MUTEX. In fact, using the advanced techniques of Section 20.4, one can define rendezvous locks directly in pSather.

Another widespread paradigm appears in the various forms of Actor systems, as exemplified by Agha's book <MIT Press, 1988>. Most of the required functionality for Actors can be built easily from the task and messaging facilities described in the this section. The interesting additional requirement is that an actor can change its behavior after processing a message. This is typically done with a `become` <behavior> statement. One can get some of this effect by simply having the `body` in your task class be a case statement that depends on some state variable. This is the standard compiled language approximation, but we can do something much more interesting using Sather's bound routines. The body of the task class can be written with a function closure which has one argument of the of the type `MSG`, say:

```
body_var: ROUT{MSG};
```

Then the actor "becoming" another behavior is modeled by assigning a function closure to this variable:

```
body_var:= bind( behavior6 )
```

and, obviously enough, the basic body code is:

```
body_var.call( receive );
```

With appropriate message types defined, we could easily have an actor receive a message naming a new behavior. It would not be obvious how to debug this kind of code, but it does illustrate the generality of the constructs. The 1993 UCB dissertation of Chu-Cheow Lim contains a nice discussion of pSather in relation to Actor systems and related issues including the inheritance anomaly

Performance and The Distributed Extension

19.1 Introduction

Performance is the raison d'etre of parallel processing, but has not yet been mentioned in this tutorial. This is consistent with the pSather design philosophy that attempts to allow users to develop and test their programs without concern for implementation details. The programmer is encouraged to code for the basic abstract machine consisting of a large shared address space and an unbounded number of threads of control. Sharing is determined by the rules of the language, not by where data happens to reside. We believe that this will make it relatively easy to develop complex codes and to port them between platforms. Of course there are performance penalties to pay for this oblivious view of the underlying platform. On some architectures, the penalty might be tolerable; a single SMP (symmetric multi-processor) or the Cray T3E provides an effective shared memory. There are major efforts <Stoutamire thesis> to achieve efficient emulation of shared memory through hardware and low-level software without help from the compiler or programmer, if these succeed it should be possible to develop efficient pSather code using only the mechanisms described above.

Hardware shared memory, or its equivalent, provides a best case on the kind of platform for which pSather is appropriate. There is also a maximum latency beyond which the pSather constructs offer no performance advantage over general message passing systems such as MPI<Foster, CH. 8>. If the ratio between local and remote operations exceeds 3 orders of magnitude (which it frequently does) then only the most loosely coupled computations can be parallelized efficiently. We believe that pSather can be effective with latency ratios up to several hundred and that systems within this range will continue to be important. Two paradigm examples are the Meiko CS-2, where local main memory operations are about 50 times faster than remote ones and our Myrinet network of quad-Sparc 10 workstations with a similar ratio. In contrast, our TCP/IP ethernet realization with the same workstations has a ratio of about 50, 000 and can not be used for most of the problems that interest us.

On some platforms there is a penalty for distributed code beyond the time needed to send and receive messages. Sometimes code must be generated to poll the network instead of using interrupts. This polling code can slow down otherwise local code although there is no messaging going on. Similarly, operating systems may impose overhead for managing virtual memory, threads, and other system resources.These software overheads depend, in sometimes nonintuitive ways, on the application, hardware, compiler, and libraries.

Achieving good performance is the central research goal of the pSather project. There are currently four doctoral projects focusing on different aspects of this. Claudio Fleiner is looking at how conventional and novel compiler optimizations can be employed in pSather. Ben Gomes is using pSather in a system for mapping neural network applications to parallel machines. Boris Vaysman is studying class library design, and is especially concerned with execution time adaptation and representation change. David Stoutamire's thesis focuses on locality and storage management and will introduce the'zone extension' that generalizes the current cluster based distributed extension. Both the current cluster version and the new zone system share the basic idea that a moderate amount of placement information supplied by the user can help a great deal in producing good code.

19.2 Placement and the @ operator.

The current pSather system has a built-in knowledge that the address space might be broken into several pieces, called clusters, and that it is expensive to reference data across cluster boundaries. For reasonable performance in a portable design, assume that it is hundreds of times more costly to reference a remote cluster. There are ways to incorporate detailed platform-dependent latency estimates, but that is beyond this tutorial. With penalties of this magnitude it is important to allocate objects and threads well. There is a large literature on programmer placement strategies and all of this appears to be usable in our context. From the pSather perspective, automatic and adaptive placement strategies should be encapsulated in classes and we have done some work on this.

The language primitives for placement are quite simple. Any expression of the language can be conjoined with the text '@exp' where exp must evaluate to an integer from 0 to the (number of clusters -1). The preceding expression is then evaluated on the corresponding cluster. In the usual case where no @ is specified, execution continues in the current cluster. Objects reside where they are created, so if the expression to the left of an @ is a create expression, the created object will reside on the cluster specified after the @. If the expression to the left of @ contains calls or other subexpressions, these will be evaluated before the @exp is calculated and thus will be on the current cluster. This is quite different than the 'owner computes' rule often built into parallel languages. One expression might involve objects resident on several different clusters and remote access is sometimes the best strategy. As described in the manual, the @ operator can also be used with the fork and parloop statements.

For repeated computations, it is always better to copy data so that the inner loop all happens on one cluster. To aid in this, pSather includes three location tests on objects. The method where(expression):INT returns the number of the cluster on which the value of the expression resides. The predicate near returns true if the value of its argument is on the executing cluster; far returns true if its argument is not on the executing cluster. The treatment of void and immutable arguments is described in the table in Section 3.4.2 of the specification. The two most basic patterns are moving the object to the operation and vice-versa. The schematic code for bringing the object to the code goes like this, given a variable v:T

local_v:T:=v;

if far(v) then local_v := v.copy end;

Of course, if we are modifying the variable v, just modifying the copy won't suffice. To execute an operation on the cluster where the object in variable v resides, one writes

operation@where(v);

For our first real examples, we return to two of our earlier sample programs. It turns out that some code that works fine with low-latency shared memory becomes awful on a platform with relative latencies in the hundreds. In Section 18.4.2, there were two variations on disjunctive search and one issue was to stop other threads once one had found an answer. Each worker thread was coded to check a global flag, stop, on each iteration. This seems harmless, but could totally dominate the execution time. The following revision illustrates some issues in coding for costly clusters.

```
class REFBOOL is
   attr val:BOOL;
   create:SAME  is return new end;
end; -- class REFBOOL
-------------------------------------------------------------------
class MAIN is
   attr num:GATE{INT};
   attr stops:ARRAY{REFBOOL};
   attr win:INT;

    main is
      i:INT;
      num:=#;
      stops:=#(clusters);
      loop i:=clusters!;
             num :- worker(i)@i;  -- workers to clusters
      end;
      ans::= num.dequeue;            -- wait for first response
      loop i:=clusters!;
            stops[i].val:=true;
      end;
      SYS::export;
      #OUT +ans  + " thread " + win  + '\n';
   end;  -- main

   worker(id:INT):INT  is
      stop:REFBOOL := #;     -- initially false
      stops[id]:=stop;
      sync;             -- everyone gets to start
      RND::seed(81463*(id+43));
      loop SYS::import;
             if stop.val then return(0) end;
             ans::=RND::int(0,10000);
             if ans.mod(71)=0 then win:=id; return ans end;
      end; --loop
   end; -- worker
end; -- class MAIN
```

The major difference in the main program is that workers are each forked to a different cluster, using the syntax worker(i)@i. To avoid costly checks of a shared signal, each worker should check a variable on its own cluster. Earlier versions of pSather had a primitive storage class, spread, that made it easy to do this case but was not general enough for all of our requirements. We will illustrate one standard pattern here and discuss others later. In this solution, each worker thread creates a local stop variable and registers it with the main program. When an answer is found, the main program sets all the flags. The only complication is that this requires a reference object, here realized by the class REFBOOL, suggesting a boxed boolean. Thus the test in each worker is on stop.val. The code as given might still be too slow if the import in each loop was costly, the obvious fix is to import and test only every Nth step.

As a second example, let's reconsider the program of Section 18.3.2 that computed the count in each chunk of some DVEC of the overall maximum value. We are now able to define the class DVEC, which was left implicit in the earlier example. It turns out that our earlier example was very unSathery code; one expects functionality to be encapsulated in object methods, not in the main program. The following example is over-simplified, but is characteristic of our approach to distributed objects in pSather. A major goal is to leverage the existing serial Sather classes, here VEC. A distributed vector, or DVEC, should have the same interface as the serial version allowing users to easily move code to a parallel platform. Distributed object classes always have a directory, dir, that points to the part of the d-object that is on each cluster. Here there is just one chunk per cluster and a total of num_chunks. The only other attribute is the fixed chunk size, ch_size.

```
class DVEC is
   private attr dir:ARRAY{VEC};-- directory is array of chunks
   private attr num_chunks:INT;
   private attr ch_size:INT;

   create(num,csize:INT):SAME is
      res::=new;
      res.num_chunks:=num;
      res.ch_size:=csize;
      res.dir:=#ARRAY{VEC}(num);
      loop j::=0.upto!(num-1); res.dir[j]:=#VEC(csize)@j  end;
      return res
   end;

   chunks!:VEC is-- Iterate over chunks
      loop j::=0.upto!(num_chunks-1); yield (dir[j]) end
   end;

   plus(v1:SAME):SAME is
      assert(aligned(v1));
      res::=#DVEC(num_chunks,ch_size);
      parloop j::=0.upto!(num_chunks-1) do@j
        res.dir[j]:= dir[j].plus(v1.dir[j])
      end;
      return res
   end; -- plus

   dot(v1:SAME):FLT is
      assert(aligned(v1));
      res:ARRAY{FLT}:=#(num_chunks);
      parloop j::=0.upto!(num_chunks-1) do@j
        res[j]:= dir[j].dot(v1.dir[j])
      end;
      r::=0.0;
      loop j::=0.upto!(num_chunks); r:=r+res[j] end;
      return r;
   end;

   aligned(v1:SAME):BOOL is
      if (num_chunks=v1.num_chunks and ch_size=v1.ch_size)
      then return true else return false end
   end;  --aligned
end; -- class DVEC
```

The code is all straightforward. Notice that in the create routine, the individual chunks of type VEC are created on separate clusters and thus live there. The chunks! iter used in our earlier example yields references to these distributed chunks; this isn't very efficient and the other methods don't use it. The example includes two of the many methods that are needed to duplicate the VEC interface. Both first require that the two vectors be aligned, i.e., have the same number and size of chunks. The predicate for testing this is also part of the public interface. The plus routine first creates a new DVEC, which itself is distributed over all the clusters. Then the parloop forks off threads to compute the separate chunks of the result on separate clusters. The code for the dot product is similar. Since the answer is the sum of the dot products of the chunks, some coordination is needed. Here each thread stores its local dot product in an entry of the shared array, res, and the total is computed after all subcomputations complete. Of course not all operations on distributed data structures partition so nicely, but it seems to be possible to provide functional interfaces to the user and bury the complexity in the library methods. It was this insight that led us to believe that OO methods will be even more important for parallel computing than they are for serial tasks.

19.2.1 Tuple Spaces, Round Three

In Section 18.4.2 we saw a fairly complex class TSPACE {TT <$TUP} that implemented the full non-blocking version of the tuple space example from Foster <op. cit.>. Here we show how this can be extended to the distributed case with no changes at all in the class TSPACE. As Foster points out, the key to a tuple (a STR) provides a natural way to distribute the tuple space over parallel machines. If we simply hash on the key then with high probability the tuple space will be split in an efficient way. The following block contains the complete code for the class DSPACE, which does this. The interface is identical to that of the underlying uni-processor class TSPACE.

```
class DSPACE{ TT < $TUP } is
   private attr tspace: ARRAY{TSPACE{ TT}};
   const n:INT:=4;-- number of subtables

   create:SAME  is
      res::=new;
      res.tspace:=#(n);
      return(res.init);
   end;
   init:SAME is
      loop i::=0.upto!(n-1); tspace[i]:=#TSPACE{ TT}@i  end;
      return self
   end;

   private hash(s:STR):INT is
      return s.hash.mod(n);-- number of subspaces
   end; -- hash

   insert(tup:TT) is k::=hash(tup.t1);
      tspace[k].insert(tup)@k
   end;

   readp(s:STR):TT is k::=hash(s);
      return tspace[k].readp(s)@k
   end;

   read(s:STR):TT is k::=hash(s);
      return tspace[k].read(s)@k
   end;

   takep(s:STR):TT is k::=hash(s);
      return tspace[k].takep(s)@k
   end;

   take(s:STR):TT is k::=hash(s);
      return tspace[k].take(s)@k
   end;

   done is
      loop k::=0.upto!(n-1); tspace[k].done@k end;
   end;
end; -- class DSPACE{TT<$TUP}
```

Everything is very simple, almost mechanical. A DSPACE has a private attr, tspace, which is an array of TSPACE, here 4 of them. The private hash function tells which of the tuple spaces to use and the various methods just call their uni-cluster counterparts. The done method must notify all the clusters when it is time to stop. Much of the Sather and pSather design has been driven by the requirement that extensions to functionality be as simple as this. In our current and future research, we plan to provide interfaces, like the tuple space one here, that shelter the user from knowing what kind of implementation is being employed and libraries that adaptively change representation as a function of load.

19.3 Addresses and the with ... near construct

We have come this far without saying anything about how pSather produces the illusion of one large shared address space on a platform, such as a network of workstations, where the reality is a number of distinct address spaces. There is still no need for implementation detail, but the fact this requires some kind of address translation must be taken into account for peak performance. The near and far predicates allow a programmer to test locality at execution time and act accordingly. But a little thought makes it clear that there must be some penalty paid because the compiler does not know if a reference type variable will refer to an object that is near or far. This could obviously involve extra storage, additional tests, etc. Moreover, the uncertainty about the locality of referenced objects can interfere with in-lining and other code optimizations. In short, it can be very useful for the compiler to know at compile time that certain variables hold object that will be on the same cluster as the executing thread. It is possible that flow analysis could determine enough of this to suffice, but we haven't convinced ourselves that this is the case.

What we have done is include into pSather a construct that allows the programmer to tell the compiler that the objects in certain reference variables will be near for a given block of computation. The syntax of this follows a common Sather pattern:

```
with <id list> near
  <statement list>
 else <statement list>
end;
```

The id list is a list of identifiers, and possibly self, that are guaranteed to be near for the block. At the start of the block, all of these are tested and, if any are not either near or void, the else clause is executed if present. It is a fatal error if the else clause is needed and is not present. The programmer has also promised the compiler that the contents of all these identifiers will remain near throughout the block. Obviously enough, checking this could be costly enough to wipe out the advantage of using the construct. This is handled in the standard Sather fashion, when the appropriate flags are set, the nearness of named identifiers is checked. As a simple first example, we can expand the schematic code of the previous section.

```
local_v:T:=v;
if far(v) then local_v := v.copy end;
with local_v  near  res:=local_v.ops end;
```

For a real example, consider the following code from a picture processing program. that applies a supplied filter to each pixel. The procedure apply is started on each cluster and gets a copy of the filter. It then makes a local copy of the filter and uses with near to inform the compiler about it.

```
class PHOTO is
   -- include S_PHOTO{SPREAD_AREF2{FLT}};
   -- include S_PHOTO{SPREAD_PEANO2{FLT}};
   -- include S_PHOTO{SPREAD_CHUNKS2{INT}};
   include S_PHOTO{BIN_CHUNKS2{INT}};

   apply(aa:FILTER) is
      t::=t1;
      t1:=t2;
      t2:=t;
      tmp::=t1;
      parloop
      do@clusters!;
         parloop p::=cl_size.times!;
         do
            i:INT;
            a:FILTER;
            if far(aa) then
               a:=aa.copy;
            else
               a:=aa;
            end;
            with a near
              loop
              x::=(t.ll2+(t.local.cs2*p/cl_size)).upto!(t.ll2+t.loc)
                  loop y::=t.ll1.upto!(t.ll1+t.local.cs1-1);
                     [x,y]:=pixel_for(t2,x,y,a);
                  end;
               end;
            end;
         end;
      end;
   end;
end;
```

We don't have enough context to understand all the details, but the structure of the code should be clear. Obviously the inner loop will run often and it is worth the set up costs to help the compiler generate the best possible code.

Advanced Topics

20.1 Thread-safe libraries

Much of the power of Sather lies in its class libraries. The pSather extension was designed to be compatible with the serial libraries and we plan to build pSather libraries as well. The current situation as of Fall 96 is suboptimal, one must pay explicit attention to the thread-safeness of serial libraries when writing pSather programs. We distinguish three successively stronger notions of thread safety.

- **Unsafe** Only usable by one thread at a time. Most Sather library classes do not use shared attributes, but any that do should be considered unsafe unless you know otherwise.

- **Class thread safe** Individual objects of the class should be considered unsafe for use by multiple threads, but there is no interference among separate objects of the same class. Most Sather library classes fall into this category and many common usage patterns will work naturally.

- **Object thread safe** Multiple threads can freely access the same object. Because of the nature of immutable classes, any immutable class that does not use shared attributes is already object thread safe. For reference classes, some form of coordination is required, but we expect to eventually provide this automatically. For the present only the class STACK{T} has an object thread safe version.

We have studied a number of ways to help automate the conversion of the serial Sather library to object thread safety and have not yet settled on the best strategy. Our current preference is to introduce explicit notations of **visitor** and **mutator** with compiler support. A discussion can be found in <Fleiner thesis>.

20.2 Yielding inside locks

The pSather primitives are powerful and as a consequence, there can be subtle interactions among them. We have tried to preserve orthogonality and have as few restrictions as possible and this has worked out fairly well. One of the more complex issues involved yield statements within lock constructs and this was prohibited in earlier releases. Recall that a yield statement in a Sather iter is a co-routine response and retains context for the return of control. An iterator could acquire a lock, and yield its value, but never be re-invoked because some other iterator finished and caused the enclosing loop to finish. This was the reason for originally banning yields within locks. But, when we require object thread safety, we can no longer afford this restriction and needed to develop a more complex implementation.

A very common operation on all sorts of container classes is to successively yield the elements of the container, for example using the elt! iter. If we want this operation to be object thread safe, there must be some kind of lock (usually an RW_LOCK.reader) to keep other threads from changing the structure of the container while the generation of its elements is in progress. This is the classic problem and is handled nicely by pSather now that we permit yields within lock statements. The semantics is that the locks within an iter are released when either the lock statement completes or when the loop containing the iter terminates. There are still some unresolved issues on how best to use these constructs and how they interact with safety features such as pre and post conditions, discussed in <Fleiner Thesis>.

20.3 Exceptions in pSather

Exception handling is complex under any conditions and parallelism only makes it worse. The basic design decision for serial Sather was to have simple terminating exceptions as described in Section 32.1.4 of the manual. For pSather, we considered a number of complex possibilities and then settled on the simplest possible solution. Exception handling only works on a per-thread basis. The code for a thread can include standard Sather protect statements. To the extent that these deal with any exceptions that are raised in that thread, computation can continue. It is a fatal error for an exception to be raised in a thread and not handled by that thread. Even so it turns out that a stack discipline is not enough to handle some of the cases that are discussed below.

Serial Sather implements exceptions in the usual way using a stack to record where an exception should jump back to at any point in execution. This gets considerably more complex for pSather and we don't yet have enough experience to know whether this will cause any difficulty for users. For one thing, pSather also needs to keep descriptors for lock statements on the exception stack, so the locks can be rolled back properly when an exception is raised. In addition, every thread needs its own exception stack at least in principle and these stacks also include entries for locks. Furthermore, threads can move their cluster of execution so the exception stack might span several clusters. Finally, loops having iterators that yield within locks also need a descriptor on the exception stack for the reasons described in Section 20.2 above. All of this means that the exception "stack" becomes a tree of stacks. Fleiner's thesis discusses this issue and its implications for performance.

20.4 User defined $LOCK classes

As we have discussed, the various forms of the lock statement work for any subtype of $LOCK. In Section 18.3 we discussed MUTEX and the various kinds of reader-writer locks. Section 18.4 was largely about the GATE and GATE{T} classes. As we mentioned briefly, there are two other pre-built classes that are restrictions of the full GATE functionality. The ATTACH class supports the attachment of multiple threads(cf. Section 18.4.2) but does not have return values. The FUTURE{T} classes do support return values, but only allow a single thread to be attached. The table in Section 3.3.5 of the language description describes exactly which methods are available in each class. This is not an advanced topic, but is only the proverbial tip of the iceberg. It turns out that the current pSather provides mechanism for a user to define his/her own $LOCK classes.

We now describe the interface that synchronization objects have to provide in order to subtype from the type `$LOCK` and be usable in lock statements. At the end of this description we will provide two examples, namely the MUTEX implementation and the skeleton of the READER/WRITER implementation. Other examples and descriptions of synchronization objects are available on-line in the pSather library code. This section is taken from Claudio Fleiner's dissertation, with minor modifications.

A synchronization object has an internal state that defines which threads may acquire[1] this object. The internal state may only change when the lock manager calls some of the functions defined below after a thread has acquired this object and before it is released again. Such a state change may be visible to other threads only after the object has been unlocked.

The class `$LOCK` is the superclass for all synchronization objects.

```
abstract class $LOCK is
    primary:$LOCK;
    reservable(tid:THREAD_ID):BOOL;
    reserve(tid:THREAD_ID);
    free(tid:THREAD_ID);
    request_reservation(tid:THREAD_ID);
    cancel_reservation(tid:THREAD_ID);
    combinations:ARRAY{ARRAY{$LOCK}};
    wait_for(tid:THREAD_ID):ARRAY{THREAD_ID};
end;
```

These functions must respect some special constraints for the lock manager to function:

- None of these functions may block, use the lock statement or raise an exception.

1. We use the term acquiring a synchronization object as a synonym to locking an object to avoid confusion, as locking an object has already a predefined meaning, which does not apply to all the synchronization objects defined in the pSather library, like RENDEZVOUS and BARRIER locks.

- the functions must be ‘class thread safe’, but not ‘object thread safe’, that is, the same function may be called in different objects at the same time, but only one function is called per object at any time.
- The state of a synchronization object may only be changed within a lock ... end block or within one of the functions defined in the $LOCK interface.
- The functions should not have side effects outside their lock objects.

The THREAD_ID class used in this example is a standard pSather class with the lock interface shown. A thread-id is, as far as the programmer is concerned, just an opaque value that cannot be used for anything else. The interface provided allows one to use thread-id’s in hash tables and to sort them, and, for debugging purposes, it is also possible to print an id. However, there is no guarantee about any special format of it, and the user should not depend on either the current size of the id or a particular format or order. It does not, for example, guarantee that threads created later get an id that is larger than threads created earlier. The only way for a thread to create thread-id’s is to ask for its own id, or the get a nil id which is guaranteed to be different from all other id’s.

```
immutable class THREAD_ID < $IS_LT{THREAD_ID}, $HASH, $NIL, $STR is
        nil:SAME;-- returns the nil id, which is different from all other thread id’s.
        me:SAME;-- returns the id of the calling thread.
        is_nil:BOOL;--returns true if self is the nil id.
        is_eq(e:THREAD_ID):BOOL;-- true if e and self are the same id.
        is_lt(e:THREAD_ID):BOOL;-- true if self is smaller than e.
        hash:INT; -- returns a hash value useful for hash tables
        str:STR;-- returns a string, useful for debugging
end;
```

20.4.1 Reservable, Reserve and Free

These three functions are the most important ones and must be defined by all synchronization objects. All three functions have one parameter, namely the ID of the thread that tries to acquire this lock. `THREAD_ID`'s can be compared, the class also defines a hash value and a str:STR function useful for debugging. See < Fleiner Thesis> for a more detailed description of this class.

Reservable returns true if the object can be acquired or locked by the thread passed as ID, while reserve actually acquires the object. free will release the lock again.

Those three functions are already enough to define the MUTEX class as shown below

```
class MUTEX < $LOCK is

        attr locked_by:THREAD_ID;-- ID of the thread that currently locks this MUTEX

        attr locked:INT;
        -- number of times that the thread stored in locked_by has locked this MUTEX

         reservable(tid:THREAD_ID):BOOL is
           -- returns true if either this MUTEX is not locked yet or already locked by the
           -- same thread that tries to lock it again
           return locked=0 or locked_by=tid;
        end;

        reserve(tid:THREAD_ID)
           -- locks this MUTEX for the thread tid
           pre locked=0 or locked_by=tid
        is
           locked:=locked+1;
           locked_by:=tid;
        end;

        free(tid:THREAD_ID)
           -- frees the lock, but only the thread that locked it can unlock it again.
           pre locked>0 and locked_by=tid
        is

          locked:=locked-1;
           -- not really necessary, but makes the code cleaner
           if locked=0 then
              locked_by:=THREAD_ID::nil;
           end;
        end;
end;
```

20.4.2 Primary

With the exception of the simple locks like MUTEX it is often necessary to have different lock objects that work on the same lock, like the reader and the writer of a reader/writer lock which form a lock family. The system has to know which lock objects work together in this way. primary is used by the system to get the “main” lock object of a family of lock objects. For all family members the method `primary` has to return the same object. Below we show the implementation of the reader/writer lock.

```
-- The fair reader/writer lock. The two attributes are just used to store the
-- reader and the writer part respectively.
class FRW_LOCK < $RW_LOCK is
        readonly attr reader:$READER_LOCK;
        readonly attr writer:$WRITER_LOCK;
        create:SAME is
           r::=new;
           r.writer:=#FRW_WRITER;
           r.reader:=#FRW_READER(r.writer);
           return r;
        end;
end;
```

The fair reader lock delegates its functionality to the writer for convenience, so that all the code is in one location.

```
class FRW_READER < $READER_LOCK is
        -- the reader delegates all calls to the writer. This way the code is concentrated
        -- in one class to make maintenance easier.
        private attr w:$RW_WRITER;
        primary:$RW_WRITER is return w; end;
        create(wr:$RW_WRITER):SAME is
           r::=new;
           r.w:=wr;
           return r;
        end;
        reservable(tid:THREAD_ID):BOOL is
           return w.r_reservable(tid); end;
        reserve(tid:THREAD_ID) is w.r_reserve(tid); end;
        free(tid:THREAD_ID) is w.r_free(tid); end;
end;
```

The meat of the lock definition is in the writer

```
class FRW_WRITER < $RW_WRITER is
        private attr writer_id:THREAD_ID;
        private attr write_locks,read_locks:INT;
        create:SAME is return new; end;
        primary:$LOCK is return self; end;

        -- the next three functions are used when working on the writer
        -- They work exactly the same way as in the MUTEX class  except that reservable has to
        -- make the additional check that there is no reader that has acquired this lock
        reservable(tid:THREAD_ID):BOOL is
           return (read_locks=0 and write_locks=0)
                  or writer_id=tid;
        end;
        reserve(tid:THREAD_ID) is
           write_locks:=write_locks+1;
           writer_id:=tid;
        end;
        free(tid:THREAD_ID) is
           write_locks:=write_locks-1;
           if write_locks=0 then
              writer_id:=THREAD_ID::nil;
           end;
        end;

        -- the next three functions do the work of the reader
        -- A reader cannot acquire the lock unless there is a writer
        -- (note that the same thread can acquire first the writer
        -- and then the reader, but not the other way around).
        r_reservable(tid:THREAD_ID):BOOL is
           return write_locks=0 or writer_id=tid;
        end;
        r_reserve(tid:THREAD_ID) is
           read_locks:=read_locks+1;
        end;
        r_free(tid:THREAD_ID) is
           read_locks:=read_locks-1;
        end;
```

20.4.3 Request_reservation, Cancel_reservation

Each time a thread waits for a lock, the system calls the function request_reservation, and, as soon as the thread continues, it will call cancel_reservation for all locks, regardless of whether the thread acquired some, all or none of the locks. Those functions are used as shown below to implement reader/writer locks with a priority for readers or writers, that is, as soon as a thread waits for the reader lock, no thread will be able to get the writer lock.

```
class WR_WRITER < $RW_WRITER is
        include FRW_WRITER r_reservable->,
                           request_reservation->;
        private attr writers_waiting:FSET{THREAD_ID};
        request_reservation(tid:THREAD_ID) is
           writers_waiting:=writers_waiting.insert(tid);
        end;
        cancel_reservation(tid:THREAD_ID) is
           writers_waiting:=writers_waiting.delete(tid);
        end;
        -- the reservable function does not change for writers, but readers can now only
        -- reserve the lock if no writer is waiting. There is also the special case where the
        -- same thread waits for a reader and a writer lock: in this case the reader
        -- can actually reserve the lock. This happens for code like
        --       lock rw.reader,rw.writer then ... end;
        -- and
        --       lock when rw.reader then ...
        --            when rw.writer then ...
        --       end;
        r_reservable(tid:THREAD_ID):BOOL is
           return
              (write_locks=0
               and (writers_waiting.size=0
                     or (writers_waiting.size=1
                         and writers_waiting.first_elt=tid)))
              or writer_id=tid;
        end;
end;
```

20.4.4 Combinations

This function defines which locks of a lock family have to be locked together, a feature used for rendezvous locks. Below, we show how the rendezvous class defined in the pSather library uses this function to define that the rendezvous main lock `self` can either be locked by itself, or the locks `r1` and `r2` have to be locked simultaneously by one or two threads.

```
        combinations:ARRAY{ARRAY{$LOCK}} is
           return ||self|,|r1,r2||;
        end;
```

20.4.5 Wait_for

This function is used for deadlock detection and should return the list of threads that have to release this lock before the thread passed as argument can eventually acquire it. The list of threads is returned as an array of `THREAD_ID`'s. This functions in a way that could be used in the `MUTEX` class.

```
wait_for(tid:THREAD_ID):ARRAY{THREAD_ID} is
   if locked>0 and tid/=locked_by then
      return |locked_by|;
   end;
   return void;
end;
```

20.4.6 Summary

This table lists all functions and shows how often they are called by the lock manager and whether they may change the state of the lock object or not.

Function	**Description**[a]	**May change internal state**[b]	**Call pattern**
`reservable`	returns true if the thread may acquire this lock		called whenever the object may have changed its state
`reserve`	acquires this lock for the given thread		once to acquire a lock
`free`	releases a lock	yes	once to release a lock
`request_reservation`	used to prioritize certain locks inside a lock family	yes	once for each lock object when a thread enters a lock statement
`cancel_reservation`	this is used to reverse the effect of request_reservation	yes	once for each lock object when a thread got the locks or executes the else part.
`combinations`	returns which locks of the lock family have to be locked together		once
`wait_for`	returns an array of threads that have to release the lock before the thread can get it		occasionally, but only if deadlock detection is enabled

a.All functions, with the exception of `primary` have a thread id as argument. This is the thread mentioned in the column "description", which is not necessarily the same as the thread that calls those functions.

b.The internal state of a lock object may also be changed by other functions, as long as this happens only inside a `lock ... end` block where this lock object has been locked.