August 16, 1995

Sather Revisited: A High Performance Free Alternative to C++

David Stoutamire
International Computer Science Institute
1947 Center Street, Berkeley CA 94704
davids@icsi.berkeley.edu

Matthew Kennel
Engineering Technology Division
Bldg 9108, Mail Stop 8088
ORNL
Oak Ridge, TN 37831-8088
kennel@msr.epm.ornl.gov

Abstract

Sather offers safety, elegance and interoperability to computational scientists. High performance scalable language extensions were a design goal of the serial language, not an afterthought. This is a DRAFT of an article to appear in the September/October issue of *Computers in Physics.*

Introduction

Sather is an object-oriented language first introduced in 1991. Since then, considerable practical experience has been obtained with the language by the hundreds of users making up the Sather community. Sather offers many safety and convenience features to help programmers avoid common errors and reuse code. Some of these were discussed in a previous *Computers in Physics* article[10]; they include strong typing, garbage collection, object-oriented dispatch, multiple inheritance and parameterized classes. Here we mention other advantages that have come to maturity since then, including iteration abstraction and parallel and distributed language extensions.

This article skips most topics presented in the previous article. The compiler, libraries, documentation, examples, performance results and other Sather resources are available at the Sather WWW page:

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

Safety

Sather is designed to shield programmers from common sources of bugs. Two important language features are strong typing and garbage collection.

Sather programs are *strongly typed*, so variables can't point to memory of an incorrect type. For example, in C it is common practice to freely mix different data types, such as signed and unsigned integers, characters, and even pointers. This can lead to subtle bugs.

C, C++ and Fortran 90 require the programmer to explicitly manage dynamically allocated storage. Good programming practice suggests making procedures locally responsible for related data structures. Unfortunately, memory management issues often cut across natural abstraction boundaries, making interfaces unpleasant in having to include low level deallocation issues. Managing memory is taming an emergent complex, globally coupled system—interesting in real physics but quite undesirable in developing software!

Like many object-oriented languages, Sather is *garbage collected*, so programmers never have to free memory explicitly. The runtime system does so automatically when it can be proven to be safe. With explicit deallocation this work is done by the programmer, although such explicit storage management often has little effect on performance[12]; it is rarely worth the effort or complexity. Sather does allow the programmer to manually deallocate objects, letting the garbage collector handle the remainder. With checking compiled in, the system will catch dangling references from manual deallocation before any harm can be done.

When checking options have been turned on in a Sather program by compiler flags, the resulting program cannot crash disasterously or mysteriously. All sources of errors that cause crashes are either eliminated at compile-time or funneled into narrow circumstances (such as accessing beyond array bounds) that are found at run-time precisely at the source of the error.

Iteration Abstraction

Earlier versions of Sather used a conventional `until...loop...end` statement much like other languages. This made Sather susceptible to bugs that afflict looping constructs in most languages. Code which controls loop iteration is known for tricky "fencepost errors" (incorrect initialization or termination). Traditional iteration constructs also require the internal implementation details of data structures to be exposed when iterating over their elements.

An important language improvement in Sather 1.0 over earlier versions was the addition of *iterators* (or just *iters*); see the examples in Figure 1. Iterators encapulate user defined looping control structures just as routines do for algorithms. Code using iterators is more concise yet more readable than code using the cursor objects needed in C++. It is also safer, because the creation, increment, and termination check are bound together inviolably at one point. Each class may define many sorts of iters, whereas a traditional approach requires a different yet intimately coupled class for each kind of iteration over the major class. Cursor objects can also prevent optimizations on inner loops.[5]

Iterators are part of the class interface just like routines. Instead of a `return` statement, they use `yield` and `quit`, and may only be called in loops. When an iter yields, it returns control to the calling loop. When it is called in the next iteration of the loop, execution resumes in the iterator at the statement following the yield. When an iter quits, it terminates the loop in which it appears.

Iterator names must end with a '!', which textually points out all places where a loop may exit. The Sather loop construct is simply `loop...end`. Built-in iters `until!`, `while!`, and `break!` offer traditional control constructs. The standard libraries define many other useful iters in many classes such as `upto!` (generate successive numbers), `elt!` (yield the elements of a container) and `set!` (store elements into a container). Such iterators make it convenient and safe to traverse complicated data structures by isolating the details of the iteration from the client of the data structure abstraction.

Iterators act as a *lingua-franca* for operating on collections of items. Matrices define iters to yield rows and columns; tree classes have recursive iters to traverse the nodes in pre-order, in-order and post-order; graph classes have iters to traverse vertices or edges breadth-first and depth-first. Other container classes such as hash tables, queues, etc. all provide iters to yield and insert elements. Arbitrary iterators may be used together in loops with other code.

Performance

Sather does as little as possible behind the user's back. There are no *implicitly* constructed temporary objects that C++ encourages, and therefore no rules to learn or circumvent. This extends to class constructors: all calls that can construct an object are explicitly written by the programmer. In Sather, constructors are ordinary routines distinguished only by a convenient but optional calling syntax. There is no peculiar automatic chaining of constructors as in C++. With garbage collection there is no need for destructors; however, explicit finalization is available when desired.

Sather 1.1 implements matrix-vector operations using the same tuned BLAS libraries as Fortran; more generally, numerical codes from Fortran can be used by Sather code. The compiler translates through ANSI C code, so Sather performance is comparable to C for general numerical codes (without Numerical Recipes' array indirection). Sather has features not available in C++ or Fortran, such as iterators; with the present compiler these can be slower than the equivalent hand-written code implementing the same operations. Because aliasing possibilities are restricted in Sather, there is the potential for future compilers to offer performance equivalent to Fortran on such code. Detailed performance comparisons with other languages can be found at the Sather WWW page.

Interoperability

Because the compiler translates through ANSI C and uses exclusively IEEE arithmetic, it is extremely portable. Gdb can be used to debug Sather programs, and there is an emacs editing mode and library browsing tool. The compiler has been ported to a wide range of Unix and PC operating systems.

Practical languages must allow applications to cleanly call existing libraries in other languages. Sather 1.1 explicitly allows for this with "external" classes which tell the Sather compiler about subroutines and data structures in other languages. The compiler supports C (for calls to the native operating system) and Fortran (for calls to existing numerical software) by directly understanding their data types and providing built-in translations to Sather's types. For instance `F_REAL`, `F_LOGICAL`, and `F_ARR_COMPLEX` represent the obvious Fortran types. One can use these in external classes to declare signatures of external Fortran routines so that they may be called from inside Sather. The annoying underscore problem and diverse calling conventions are handled by configuration of the Sather compiler. Sather's native matrix libraries use Fortran's column-major layout so calling well-tuned Fortran matrix solvers is easy for the user, and is the default implementation in the standard library.

The external C interface supports both the calling of C routines as well as the direct inclusion of C types into Sather objects. For instance, a Sather object may have an attribute which is a native C `FILE*` or a handle to an X window system object and then make direct calls to C with these pointers, fully typesafe on both the Sather and C sides. It is intended that the scheme should support most calls to the UNIX API without needing glue routines. Syntax has been reserved for future support for interfaces to C++ and newly-emerging inter-language object standards such as CORBA, each of which define an object semantics of their own. These will be supportable as new kinds of external classes without altering core Sather semantics.

Parallelism and Languages

Reusable libraries will be much more important for parallel systems than serial systems because they are so much harder to write.

The comfortable illusion persists that compiler optimizations will allow conventional serial program to get high performance on parallel and distributed machines. Scientific parallel computing has mainly relied on automatically parallelizing compilers that can analyze fixed arrays. This work has improved our understanding of compiler optimizations that are not limited to Fortran. Nevertheless, no one knows how to extend this compiler-oriented approach to general data structures and algorithms.

Matrix/vector operations, and more generally data parallel programming, provide elegant and efficient solutions to some problems. Data parallel algorithms can execute efficiently on a wide spectrum of parallel hardware. However, languages which can only express data parallel operations are not suitable for the entire range of data structures and algorithms.

A successful approach to exploit parallelism has been to fall back on the low-level programming of C, C++ or Fortran with primitive library mechanisms for communication and synchronization, such as PVM or MPI. Any program can be written at this low level, but such programs are hard to write, test, maintain and verify.

Object orientation facilitates encapsulation of efficient parallel algorithms for machines with different characteristics. Compilers will not be able to perform the different algorithmic optimizations on the same source code necessary to achieve high performance on a single workstation, a T3D, and a network of workstations. The object-oriented approach allows the optimal version of an algorithm to be selected according to the machine it is actually running on. To do this, the language must be able to express the parallelism and synchronisation behavior of libraries.

There are many proposals to extend C++ with distributed features.[11] A limitation to these approaches is the low-level, weak typing. A new language which extends an old one inherits old difficulties while accreting new ones; this is particularly apparent in C++.[2] We now know more about language design and the properties that support efficient yet safe programming practice. pSather may be the first object-oriented language in which the serial and distributed versions co-evolved, although there will be many others.

pSather

pSather is the parallel extension to Sather. It has been under development since 1990,[8,9] but has only recently been released for public use. A major goal has been the easy reuse of serial code in parallel applications. pSather adds support for threads, synchronization, communication and placement of objects and threads.

In figure 1, iterators were used to traverse arbitrary data structures. Figure 2 shows the `parloop` construct combined with iterators to apply a routine over items in two containers in parallel. `parloop` automatically forks a thread for each pair of elements and waits for all threads to terminate.

pSather follows the Sather philosophy of shielding programmers from common sources of bugs. One of the great difficulties of parallel programming is avoiding bugs introduced by incorrect synchronization. Such bugs cause completely erroneous values to be silently propagated, threads starved out of computational time, or programs to deadlock. They can be especially troublesome because they may only manifest themselves under timing conditions that rarely occur (*race conditions*) and may be sensitive enough that they don't appear when a program is instrumented for debugging (*heisenbugs*).

pSather makes it easier to write deadlock and starvation free code by providing structured facilities for synchronization. The *lock statement* automatically performs unlocking when its body exits, even if this occurs under exceptional conditions. It automatically avoids deadlocks when multiple locks are used together. It also guarantees reasonable properties of fairness when several threads are contending for the same lock.

pSather differs from concurrent object-oriented languages that try to unify the notions of objects and processes by following the "actors" model.[1] There can be a grave performance impact for the implicit synchronization this model imposes on threads even when they do not conflict.[4] While allowing for actors, pSather treats object-orientation and parallelism as orthognal concepts, explicitly exposing the synchronization with new language constructs.

Figure 3 shows code from a program that forks a thread for each array element. Each thread then loops, asynchronously updating its element, taking care to lock the neighboring elements so that they will not change until its update is complete. In this simulation each element keeps an independent time, and each controlling thread stops when a maximum time is reached. Each cell's thread only synchronizes with its immediate neighbors, not the whole array. This would avoid idle time should some elements update much faster than others. Boundary elements are not updated.

In practice, synchronization between tasks must always coincide with communication. Many parallel languages and libraries distinguish primitives for communicating and synchronizing. Because these so frequently go hand-in-hand, pSather provides a powerful construct to do both at once. A pSather `GATE` is a queue with implicit synchronization. It generalizes many constructs found in other languages, such as *fork/join, barriers, semaphores, futures, condition variables,* and *mailboxes*. Figure 4 shows how to use a `GATE` as a placeholder for results not yet computed. It is also possible to control multiple threads with a single `GATE`.

High Performance Distributed Memory

Because of volume production, commercial workstations today offer better potential price/performance for general code than massively parallel processors. They also fit comfortably in the capital budget of most research grants. For these reasons networks of workstations considered as a single parallel computing facility will become an economically important platform.[13]

Networks of workstations have longer latencies than centralized machines. In order to achieve high performance, it is important to organize data so that they do not need to be moved between workstations when that will stall waiting computations. Some compiler optimizations can alleviate this, but generally the programmer must design a layout intimately integrated with the specific algorithm.[3]

Machines do not have to have large latencies to make data placement important. Because processor speeds are outpacing memory speeds, attention to locality can have a profound effect on the performance of even ordinary serial programs.[7] Existing serial languages can make life difficult for the performance-minded programmer because they do not allow much leeway in expressing placement. For example, extensions allowing the programmer to describe array layout as block-cyclic is helpful for matrix-oriented code but of no use for general data structures.

Some environments expose latencies to the programmer with a distributed memory model and explicit communication. It is much easier to program with a shared memory space which uses one name to refer to each datum no matter where the reference is. High performance still requires explicit human-directed placement, but we wish to avoid requiring heroic feats of virtuoso programing. pSather tries to provide the best of both worlds; the compiler implements the shared memory abstraction using the most efficient facilities of the target platform available, while allowing the programmer to provide placement directives for control and data (without requiring them). This decouples the performance-related placement from code correctness, making it easy to develop and maintain code enjoying the language benefits available to serial code. Parallel programs can be developed on simulators running on serial machines. A powerful object-oriented approach is to write both serial and parallel machine versions of the fundamental classes in such a way that a user's code remains unchanged when moving between them.

Languages with no shared memory abstraction are impractical for anything but small hand-tuned kernels. Languages which assume a shared memory implementation but expose none of the gritty performance consequences to the programmer can not deliver high performance. Languages with a compiler-implemented shared memory abstraction that exposes placement options will become the high-performance standard for general computing, because they have the potential to exploit the resources of a parallel computer more effectively[6] without abusing the programmer to do so.

Distributed pSather

The memory performance model of pSather has two levels. The basic unit of location in pSather is the *cluster*. It is assumed that reading or writing memory on the same cluster is significantly faster than on a remote cluster. A cluster corresponds to an efficient group in the memory hierarchy, and may have more than one processor. For example, on a network of workstations a cluster would correspond to one workstation, although that workstation may have multiple processors sharing a common bus. This model is appropriate for any machine for which local cached access is significantly faster than general access.

In pSather, each thread and object has a *location*, which may be *fixed* or *unfixed*; unfixed threads or objects may migrate without being directed to do so by the programmer. Locations are fixed by the '`@`' operator. There are also ways to find the location of objects to allow a program to make informed decisions about where to place new threads or objects.

In most languages there is no way to distinguish the locality of data that is referenced. This is convenient for the programmer but ignores the realities of modern machines, which introduce penalties for poorly placed data in the form of missing a cache line, TLB misses, or even paging to disk. This is especially important for distributed machines where data may reside on other nodes. pSather allows the programmer to help the compiler and runtime by providing explicit placement. If threads or objects are unfixed, the compiler and runtime can attempt to place the data somewhere suitable. For instance, the previous examples have had no explicit placement but would all run correctly no matter what the runtime data placement is, with the compiler and runtime helping if possible. Because pSather is a garbage collected language, there is even the potential for an aggressive implementation to relocate objects according to access patterns observed at runtime.

Normal pSather objects reside entirely on a single cluster. For some data structures such as distributed arrays, it is important that remote parts can be accessed without going through a directory to avoid indirection. pSather *spread* objects have a portion residing at the same address on every cluster. This is often the natural implementation of distributed matrices. Spread objects are managed with a special replicated heap.

Conclusion

Sather offers many advantages over C++ for the computational physicist, ranging from safety and garbage collection to high performance distributed extensions. One author (MK) has exploited Sather's object-orientation in nonlinear time series analysis. Sather's convenient interface to Fortran and column-major matrices encourage reuse of existing numerical codes; BLAS is the standard implementation of matrix-vector operations. The Sather libraries are the result of a cooperative effort by many people on the net and continue to grow; the compiler and all sources (the compiler is itself written in Sather) are free.

The Sather WWW site has much more information on Sather, including an expanded version of this article with specific motivations for physicists, discussion of C++, examples and performance data.

References

[1] G. Agha, "Actors: A Model of Concurrent Computation in Distributed Systems", The MIT Press, Cambridge, Massachusetts, 1986.

[2] S. Burson, "The Nightmare of C++", Advanced Systems November 1994, pp. 57-62. Excerpted from *The UNIX-Hater's handbook*, IDG Books, San Mateo, CA, 1994.

[3] S. Carr et al., "Compiler Optimizations for Improving Data Locality", *Proceedings of the Sixth Intnl. Conf. on Arch. Support and Prog. Lang*, ACM October 1994 pp. 252–262.

[4] A. Chien, "Concurrent Aggregates (CA): An Object Oriented Language for Fine-Grained Message-Passing Machines", PhD Thesis, MIT, July 1990. Also available as MIT Artifical Intelligence Laboratory Technical Report 1248.

[5] S. Murer, S. Omohundro and C. Szyperski, "Sather Iters: Object-Oriented Iteration Abstraction", International Computer Science Institute tech report TR-93-045 August 1993. Available at `http://www.icsi.berkeley.edu/Sather`.

[6] J. Larus, "Compiling for Shared-Memory and Message-Passing Computers", *ACM Letters on Prog. Lang. and Systems*, Vol. 2 No. 1-4 March-December 1993, pp. 165-180.

[7] A. Lebeck and D. Wood, "Cache Profiling and the SPEC Benchmarks: A Case Study", *IEEE Computer*, October 1994 pp. 15–26.

[8] C. Lim, "A Parallel Object-Oriented System for Realizing Reusable and Efficient Data Abstractions", International Computer Science Institute technical report TR-93-063, 1993. Available at `http://www.icsi.berkeley.edu/Sather`.

[9] S. Murer, J. Feldman, C. Lim, "pSather: Layered Extensions to an Object-Oriented Language for Efficient Parallel Computation", International Computer Science Institute technical report TR-93-028 June 1993. Available at `http://www.icsi.berkeley.edu/Sather`.

[10] S. Omohundro, "Sather Provides Nonproprietary Access to Object-Oriented Programming", Computers in Physics, Vol. 6, No. 5, Sep/Oct 1992 pp. 444–449.

[11] B. Wyatt, K. Krishna and S. Hufnagel, "Parallelism in object-oriented languages: a survey", *IEEE Computer*, Vol. 11, No. 6 November 1992 pp. 56-66.

[12] B. Zorn, "The Measured Cost of Conservative Garbage Collection", Univ. Colorado at Boulder tech report CU-CS-573-92, April 1992.

[13] D. Patterson, "The Case for a Network of Workstations", To appear in *IEEE Micro*; also at `http://now.cs.berkeley.edu/`.

Figure captions:

Figure 1: Using Sather iterators

Figure 2: The `parloop` construct

Figure 3: An asynchronous coupled map lattice simulation

Figure 4: Forking a parallel computation to be used later

Figure 1:

```
-- Compute sum = 1 + 2 + 3 + ... + 10;
loop sum := sum + 1.upto!(10) end;

-- Set all elements of array 'a' to 5.0
loop a.set!(5.0) end;

-- Fill 'a' with the square root of elements of 'b'
loop a.set!(b.elt!.sqrt) end;

-- Print 1 through 100 factorial
-- (INTI is the built-in infinite precision integer class)
prod:INTI:=1.inti;
loop
    i::=1.upto!(100);
    prod := prod * i.inti;
    #OUT + i + "! = " + prod + "\n"
end;

-- Walk two arbitrary ordered data structures 'x' and 'y'
-- to see if they have the same nodes in order; the actual
-- data structures may be different or even unknown
if x.size /= y.size then return false;
else
    loop
        if x.elt! /= y.elt! then return false end
    end
end;
return true
```

Figure 2:

```
-- perform some computation with e1 and e2, proceeding with all
-- such pairs in parallel.
parloop
    e1 := a.elt!;  -- e1 holds an element from a
    e2 := b.elt!   -- e2 holds corresponding element from b
do
    e1.frobnify_with(e2)  -- frobnify elementwise in parallel
end
```

Figure 3:

```
a:ARRAY{CELL};       -- an array of elements of type CELL
locks:ARRAY{MUTEX};  -- an array of locks, one for each element
...
parloop
    i ::= 1.upto!(a.size-2)     -- For every inner element
do
    loop
        -- continually update a[i] while neighbors are unlocked
        lock when locks[i-1], locks[i], locks[i+1] then
            a[i] := some_function(a[i-1], a[i], a[i+1]);
            until!(a[i].time > max_time);
        end
    end
end
```

Figure 4:

```
g := #GATE{FLT}   -- Create a GATE holding IEEE floats
g :- compute;     -- Start a parallel computation
...               -- Do something else while waiting
result := g.get;  -- Get result or wait if not yet available
```