INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center Street • Suite 600 • Berkeley, California 94704 • 1-510-642-4274 • FAX 1-510-643-7684

Near **or** *Far*

Hermann Härtig

TR-94-004

Abstract

To efficiently program massively parallel computers it is important to be aware of *nearness* or *farness* of references. It can be a severe *performance bug* if a reference that is meant to be near by a programmer turns out to be far. This paper presents a simple way to express nearness and farness in such a way that compile-time detection of such performance bugs becomes possible. It also allows for compile-time determination of nearness for many cases which can be used for compile time optimization techniques to overlap communication with processing. The method relies on the type system of a strongly typed object oriented language whose type rules are extended by three type coercion rules.

1 Introduction

In an abstract way, it has often been claimed that object oriented languages provide an optimal basis for programming massively parallel computers. This paper is different: it shows how, in a concrete and very simple way, an important problem in programming such computers is solved by directly making use of the type system of a strongly typed object oriented language.

After a long period of experimentation, massively parallel computers seem to be converging towards a *standard architecture*. That standard architecture makes use of today's off the shelf microprocessors and memories to build powerful computing nodes and employs sophisticated and very efficient networks to interconnect thousands of such nodes.

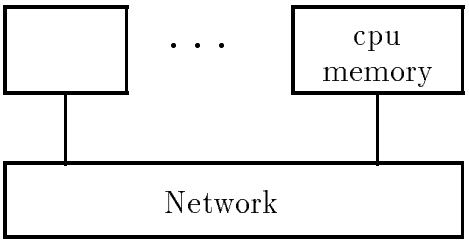

Figure 1: Standard Architecture

These architectures are characterized by a significant difference in memory access latency between *near accesses* (i.e., within a node) and *far accesses* (i.e., across the network). The minimum latency difference currently[1] seems to be a factor of around five, while a factor in the order of ten to one hundred is the normal case. To make efficient use of such architectures it is of utmost importance to always be aware of the nearness or farness of references and to overlap far accesses with processing within nodes and near accesses.

That convergence in hardware technology has lead to numerous efforts to develop programming models, languages, libraries, and environments. Two major directions can be identified:

- Compiler controlled:
  This direction is characterized by the employment of highly sophisticated compilers to organize and distribute parallel work over such architectures. Examples include the translation of functional programming languages[3] for those architectures, attempts to transfer well understood SIMD technology by translating sequential programs (e.g. FORTRAN[5]) as well as the extension of modern languages by "FORALL constructs". The common denominator of this direction is that compilers distribute the work and data structures. That approach, however, is usually limited to regular data structures(e.g., arrays) that can be distributed in a more or less obvious manner.

- Programmer controlled:
  Here the organization of work for a massively parallel computer is done by the "competent programmer". Examples for this approach include Split-C[4] and most designs published in a recent special issue of CACM[2]. These efforts are either based on a message passing(e.g. OCCAM[9]) or a (logical) shared address space programming paradigm. The common denominator here is that the distribution of work is organized by the programmer. That makes possible to use even highly irregular and complex data structures. Examples include distributed trees, event driven simulation, and bounded tree search algorithms. It seems that many of the *grand challenge* type of problems fit into this category.

Particular emphasis is placed on the shared address space paradigm of computation because a large number of sequential programs is implicitly based on that paradigm and it seems best understood. The problem solved in this paper is of particular interest in that approach.

A major obstacle to overcome is the *efficient* implementation of the logical address space on *standard* (i.e., non shared) memory architectures. To achieve that the employment of advanced compiler technology is needed e.g., to prefetch accesses to far memory(latency hiding) or to do alignment fitting to a concrete cache architecture. These optimizations require to determine nearness or farness of references at compile-time.

However, even in an efficiently implemented logical shared address space programming paradigm the programmer needs to be aware of the nearness or farness of references, e.g., to hide latency using software caches provided by libraries. It can be a severe performance bug if a reference that is meant to be near by a programmer turns out to be far.

Therefore, to support programmers and compilers in latency hiding, the "competent programmer" approach requires safe and efficient means to express *nearness or farness* of references. "Safe" means that — as in type safe languages — violations are detected at compile or run-time. "Efficient" means that the vast majority of violations must be detectable at compile time to avoid run time checks.

This paper describes a solution to this problem. It brings the nearness issue onto the language level and uses static type checking rules to determine, for a large number of references, whether they are near or far.

The remainder of the paper is organized as follows. First, the strongly typed object oriented language Sather is introduced as needed for this paper. pSather, a parallel extension of Sather, is introduced to demonstrate the nearness problem in more detail. Then the extensions required to express and to check nearness are introduced and implications and limitations are discussed. Finally, some conclusions are drawn.

2 Sather and pSather

The basic idea presented in this paper is to exploit properties of strongly typed object oriented languages. Since the programming language Sather[7, 8] has a clear, clean, and elegant type system, it is used as basis for the further explanation. This section essentially is a brief summary of [7, 8] and [6]. Section 2.1 is, for the most part, taken literally from the Sather report[8].

2.1 Sather

Data structures in Sather are constructed from *objects*. Each object has a unique *type* which defines the operations which may be performed on it. Each Sather variable has a declared type which determines the types of objects it may hold. Types are defined by textual units called *classes*. Sather programs consist of sets of classes. Classes define the following *features*: *object attributes* which make up the internal state of an object, *shared* ...attributes which are shared by all objects of a type, and *routines* which perform operations on objects ....

There are three kinds of objects in Sather: *value objects* which are passed by value (e.g., integers), *reference objects* which are referenced via pointers (e.g., strings) .... There are five kinds of types: *value types* describe value objects, *reference types* describe reference objects, ...

The *type graph* is a directed acyclic graph whose nodes are types and whose edges define *type conformance*. The type graph specifies the types of object that a variable may hold and imposes *conformance constraints* on type interfaces. If there is a path in the type graph from a type *t1* to a type *t2*, we say that *t1* is an *ancestor* of *t2* and that *t2* is a *descendant* of *t1*.

A type is said to *conform* to each of its ancestors in the type graph. The fundamental Sather typing rule is: *An object can only be held by a variable if the object's type conforms to the variable's declared type.* Sather is statically type-safe and it is not possible to write programs which compile and yet violate this rule. Variables declared by value or reference types can only hold objects of the same type. Variables declared by [a certain type] can typically hold more than one type of object [i.e., objects of any descendant type].

In the following we refer to attributes and local variables of reference type simply as a reference.

...classes may be *parameterized* by type parameters. Type parameters are local to a class and shadow any references to classes with the same name and no parameters. When the parameters of a parameterized class are specified, it behaves like a non-parameterized class whose body is a textual copy of the original class in which each occurance of a parameter is replaced by its specified value.

Implementation inheritance is defined by ...the "`include`" clause. [...It] includes an entire class and may undefine or rename features ....

The following code:

```
class A < $B is includes C ... end;
```

defines a type A as a subtype of B and includes C as the implementation of A.

Objects are constructed using *constructor expressions*. The following code:

```
a:=#A;
```

constructs a new object of type A and assigns it to a.

2.2 pSather

PSather is an extension of Sather for massively parallel computers. Two features of pSather are of interest to this paper, namely *threads* and the notion of object *location*. To avoid confusion, it must be pointed out that the Sather language extensions described in section 3 deviate from pSather, as defined by the current report[6]. (Serial) Sather extended with threads and the notion of location as found in pSather, and extended with few other features is used as the substrate to explain the contribution of this paper.

Threads

Each routine can be called either synchronously, i.e., the caller awaits its completion, or asynchronously, i.e., a new "thread" of control is started. The syntactic form of a synchronous call in Sather is:

```
a:=f( ... );
```

while an synchronous call has the form

```
a:-f( ... );
```

PSather contains language features to synchronize threads.

Location and Locus of Control

pSather uses the logical shared address space paradigm of computation. However, to reflect the distributed memory architecture of underlying machines, it contains the notion of a node, which corresponds to a hardware node as mentioned in the introduction. The notion of a node is used as follows:

- Each object is located as a whole on one node which is called the object's location.
- The node on which a thread is executed is called the thread's locus of control.

A thread's locus of control stays at one node unless it is redirected explicitly. New objects are created at the current thread's locus of control unless they are created on another node explicitly. It is, in general, not known at compile time, whether a thread's locus of control is at the location of the object whose routine it currently executes.

2.3 Near and Far discussion for pSather

With the exception of the location of local variables (including parameters of routines), no assumptions about the nearness of objects in relation to the current thread's locus of control can be made at compile time. A thread's locus of control s independent of the location of an object whose routine is currently executed. Furthermore, for attributes of classes and for local variables it cannot be determined at compile time, whether or not they refer to objects near to the object whose routine is currently executed. That implies that instructions have to be issued to check for nearness at runtime.

These checks can be reduced to one check per routine call if, in each routine, the nearness of the called object(self) is checked first and then either efficient code for the near or expensive code for the far case is executed. In practice, that causes the issuing of two versions of code for each class, the *near version* to be executed efficiently once the nearness has been established and the *far version*, that on most mpp architectures has to generate a message for each access.

An initial attempt has been made to overcome that problem by introducing an assertion statement:

```
withnear x,y,z do .. end
```

It establishes a hint that the variables x,y,z are near to the current locus of control. But there are no simple means to check that at compile time, i.e. instructions have to be issued to establish the correctness of the assertion at run time. Another drawback is that all objects referred to by attributes of x,y,z and their transitive closure are not covered by the assertion.

The importance of that problem becomes especially apparent when the reuse of sequential classes is considered. If nothing can be said at compile time about nearness, two versions of code must be issued for sequential classes. All checks for nearness have to be done. The withnear assertions cannot (or at least should not) be inserted into sequential classes afterwards.

It can be argued that, with the advent of hardware supporting large address spaces and far accesses, such as KSR1 and Cray T3D, the mentioned checks come for free. The hardware address space could be divided among the nodes and page fault mechanisms could be used to detect far references[1] or even handle far accesses. That still leaves the issue of code optimization to overlap far accesses with computation that is considered a key to the efficient usage of massively parallel computers. To enable a compiler to schedule far accesses ahead of near accesses, nearness needs to be known at compile time.

These problems are by no means restricted to pSather. Rather, they are general problems arising from the mapping of the shared address space programming paradigm to distributed memory hardware architectures. The following section elaborates on how these difficulties can be overcome using Sather's type system.

3 Far and Near Types

The basic idea is to explicitly state for each reference type variable (i.e., for each attribute of a class, for each local variable including parameters and the result of a routine holding a reference) whether it can hold a reference to objects on a different node. These are called variables of *far types*. All other variables are called variables of *near types*. Then, variables of near types may be assigned to variables of far types. Variables of far types, however, must not be assigned to variables of near types. The compiler's type system ensures that. Then, for the general case, if threads execute near the objects whose routines they execute, it

[1] That may not be advisable, however, since operating system page fault treatment adds significant overhead to far accesses and potentially disrupts cache flow.

is known at compile time which variable accesses can cause far memory accesses.

3.1 Example

Before a more precise description, a simple example (Figure 2) should provide an intuitive understanding of the notation and permissible assignments.

The assignments in line (2) and (6) are not allowed since `x` may refer to a far object. The assignment in (4) is not allowed since `x` and therefore `x.foo` may refer to an object of another node. The assignment in line (7) is not allowed, since the new object is potentially constructed on another node. The routine call in line (9) is not allowed since the actual parameter `a` is far from the point of view of `c`.

3.2 Model of execution

Objects as a whole reside on a node and cannot migrate; a predefined routine *location* delivers the id of that node.

If a thread calls a routine of another object the thread normally continues execution on the called object's location. This default can be ignored by the compiler (e.g., for purposes of inlining) and explicitly overiden by the programmer. Semantically, that does not make any difference. Different code, however, has to be issued in such cases and it will result in different efficiency.

Reference type attributes of an object normally refer to objects on the same node. Formal reference type parameters and results of routines normally refer to objects on the node, where the called object is located. All far references are marked using the built in FAR class. The formal description is given in subsection 3.3.

Value parameters (i.e., if the parameter's type is defined by a value class) in routine calls are sent to the called object's location when the routine is called. A value result is sent back to the caller's location when the routine call ends.

If a new object is constructed, its location is normally the location of the object whose routine is currently executing. This default can be overiden using the `@` operator.

3.3 Far types

The FAR{T} notation as used in the example is formalised here using terms of the Sather type system. For each type T, FAR{T} introduces a new type as direct ancester of T. FAR{T} has no ancestor. To

```
CLASS T is
 attr foo: FOO

 end;

CLASS C is
 attr x: FAR{T};-- can hold a reference to an
                -- object on a remote node
 attr y:  T;    -- cannot ....
 attr foo: FOO; -- cannot ....

 k(arg:T) is

   x:=y;              --  (1) allowed
   y:=x;              --  (2) not allowed

   foo:=y.foo;        --  (3) allowed
   foo:=x.foo;        --  (4) not allowed

   arg:=y;            --  (5) allowed
   arg:=x;            --  (6) not allowed

   end;  end; -- class C

  -- in some other routine of another class:

  a: T;
  b: C;
  c: FAR{C};

  b:=#C@SomeNode;     -- (7) not allowed
  c:=#C@SomeNode;     -- (8) allowed
  c.k(a);             -- (9) not allowed
```

Figure 2: Simple example

enable access to objects on a far cluster, a predefined parametrised class is defined for all types T in the following way:

```
class FAR{T} > T is include T end;
```

References to objects across node boundaries are only allowed to objects of FAR reference types.FAR{FAR{T}} is equivalent to FAR{T}.

Based on that definition, some of the restrictions, as intuitively introduced above, can be enforced at compile time. The assignments in line (2) and (6) of the example are prohibited due to the Sather type system rule: only conforming types may be used in assignments.

The assignments in line (4) and (7), however, and the routine call in line (9) would still be allowed. To enforce these restrictions at compile time near types T are coerced to far types FAR{T} whereever necessary. For that purpose the following type coercion rules must be added to the type system:

1. For an expression `a.b`:
   If `a` is of a far reference type and `b` is of any type T, then the type of `a.b` is coerced to FAR{T} for all uses other than as an assignment designator.

2. For a routine call, `a.f(arg)`:
   If a is of a far reference type and the actual parameter arg is of any type T, then the type of `arg` is coerced to FAR{T}

3. For an object construction #T@Node:
   The resulting type is coerced to FAR{T}.

Coercion rule (1) subsequently applies for any chain of a.b.c. Since in Sather all expressions are reduced to this form, the rule covers all expressions. E.g., a+b is just syntactic sugar for a.plus(b).

Some sample programs and one non-trivial application have shown that compile time analysis based on the type system and on the coercion rules are too pessimistic in some cases. These cases arise whenever variables or expressions of far reference types refer to near objects and this is known to the programmer. In such cases, runtime checked type casts, similar to run time checked type casts in strongly typed languages, are used. If `y` is a variable of near type T and `x` is of type FAR{T} then

```
a.y:=T(x);
```

causes a run time error if

```
(a.b.c).location \= x.location.
```

This again subsequently applies for a chain of a.b.c. I.e.,

```
a.b.c.y:=T(x);
```

causes a run time error if

```
(a.b.c).location \= x.location.
```

After such an assignment, all objects "hanging" at y are known at compile time to be near since y is a near type variable. A common case is the assignment of constructed objects:

```
a.b:=T(#T@a.location);
```

where the runtime check is unnecessary, if that situation can be discovered at compile time. To support the discovery at compile, a keyword `destination` is introduced. In an assignment or a copy opeartion

```
a.b:=#T@destination
a.b:=x.copy@destination
```

the new object is created at a.location. This changes the coercion rule (3) to:

- For an object construction or copy operation `#T@Node` or `x.copy@Node`:
  the resulting type is coerced to corresponding far type if `Node \= destination`.

The `destination` of a routine result is defined to be the calling object's location.

3.4 Shared variables

It can be observed that, for some classes, variables are needed that are shared between several objects. In general there are two appearances of this:

- Shared resources:
  the objects share a common pool of resources which are needed for the implementation of the class. An example is a string class that uses a pool of characters.

- Global variables:
  the objects use the shared variable as a global variable. An example is a variable that counts how many times a certain routine of a class is called.

In Sather, as in some other object oriented languages, there is no way to differentiate between these uses: *Shared variables are common to all objects of a class[8].* For parallel programs, two interpretations of shared variables come to mind:

- Variables are shared between all objects of a class that are located on the same node. This makes sense for the use of shared variables as shared resources, but makes the use as global variables impossible.

- Variables are shared for all objects on all nodes. This makes sense for the use as global variable,

but would make the more common case (i.e., the use as shared resources) prohibitively expensive

Far types deliver an elegant solution for this dilemma. Near shared attributes are common to all objects of a class on one node and far shared attributes are common to all objects of a class. Without far types, the matter of shared attributes leads to some ad hoc definitions and potentially hairy extensions of a language.

3.5 Limitations

The invention of far types does maintain the logical shared address space model of computation. All objects, whereever they may be located, can be accessed from anywhere. However, two limitations are introduced compared to the (logical) shared address space model without far types.

Copy

Suppose we have the class:

```
class A is
  attr foo: FOO -- FOO be a near ref. type
  end;
```

and an object O of that type, O.FOO holding a non void reference. Then

`P:=copy(O)@AnotherNode`

would cause `P.foo` to hold a reference to an object at another node.

Therefore objects of classes that contain near references must not be copied across node boundaries. This restriction can be enforced at compile time.

Sequential Classes

Sequential classes naturally do not contain far types. They can be reused in parallel programs and be compiled into efficient code since all their references are known at compile time to be near. However sequential classes cannot be used in an arbitrary manner. The following example should make the limitation clear:

```
class LIST{T} is
   -- Linked lists.

   attr next: SAME; --

   insert (SAME);

end; -- class LIST{T}

list: LIST{T}; -- be nonvoid

list.insert(#LIST{T}@AnotherNode)
```

Here the construction of an object at another node yields an object of type FAR{LIST{T}}, which cannot be used as an actual parameter, since the formal parameter is of type LIST{T}. This would violate the standard Sather conformance rule.

Discussion of limitations

These limitations are not arbitrarily imposed, but rather originate from and comply with the "competent programmer" principle mentioned in the introduction. A competent programmer has to be aware about the distribution of objects. The strong type system here is a reminder about that. See also section 5 for ideas to overcome both limitations.

4 Optimizations

This paper's main emphasis is on presenting a way to express, check, and determine nearness at compile time rather than on the use of that knowledge for optimization purposes. Some comments, however, are warranted.

Tests for nearness

If nearness is not known at compile it has to be established at runtime, requiring at least one test per reference attribute access for all accesses including accesses to objects defined in normal sequential classes. As pointed out in section 2.3, this causes the generation of a significant amount of extra code. If runtime determination can be left to the virtual address space hardware, additional overhead due to page fault handling is added to each far access. That additional cost can be avoided, however, if very sophisticated hardware is invented to use page faults to generate messages to far nodes. If the same effect can be achieved for simpler hardware using intelligent compiler optimizations, additional die space can be saved.

Instruction scheduling

Instruction scheduling is a well understood and important optimization technique for RISC machines. E.g., emphasis is placed on reordering instructions such that accesses to slow memory overlap with internal operations. Similar techniques can be employed to

overlap far with near accesses. If we assume that in well written programs the number of far accesses is low compared to near accesses, such reordering may lead to significant savings. Here is a rule of thumb calculation: given an architecture such as Cray T3D where near and far differences differ by a factor of around five, and given a program consisting of basic blocks with a five to one ratio of near and far accesses, far accesses come for free if optimal reordering is possible. Further experimental work is needed to establish how near real programs come close to such ideal assumptions.

Far Objects and Cache Consistency

Depending on the hardware architecture, writes to variables on another node may have to be implemented differently with regard to caching to maintain cache consistency. E.g., it may be necessary to invalidate caches on several nodes in such cases. The number of such invalidates might be signficantly reduced using compile time knowledge about farness. To enable such techniques, however, a notion of consistency of objects needs to be raised to the programming language level in an equally clearr way.

5 Future Work

Since the limitations of the scheme as mentioned in section 3.5 may eventually turn out to cause some unnecessary rewriting of sequential classes, program transformation schemes will have to be considered that replace near references by far references automatically. E.g., the sequential class LIST may be used to automatically produce a class D_LIST that can be used to build distributed lists as in the example above and whose objects can be copied across node boundaries. It is not clear yet if unnecessary far references can be avoided by intelligent transformation schemes. That would still comply with the "competent programmer" principle since it would still be left up to the programmer to decide whether the original or the derived class is used in a particular situation.

Another interesting question is raised when looking at the method presented in this paper. Even if nearness or farness is not raised to the language level, the method theoretically can be used to determine some near references at compile time. A simple algorithm to do that works as follows:
For all types T of a given program, the respective type FAR{T} is invented. Then for all references, all permutations of declarations as near or far reference are produced. The rules as presented in this paper can be used to determine at compile time, if a combination is valid or not. The resulting program which has no type errors and which has the largest number of far reference declarations is compiled into executable code.
Of course, the algorithmic complexity of this brute force algorithm is much to high for practical use. Future research, however, may emply type inference methods and may lead to heuristics to check a much smaller number of combinations to achieve some partial determination of nearness at compile time. The author, however, considers it of crucial importance that programmers declare which references they consider near or far. This redundancy can be used to check for performance bugs.

Far types as introduced in this paper are not yet part of an implemented parallel programming language. One reason for that is that an equally clear notion at the language level for consistency issues is not at hand yet. Further investigation of this consistency issue at a programming language level is seen as crucial to support the "competent programmer" approach to massively parallel programming.

Using the ideas of this paper, nearness of a large number of references can be determined at compile time. That opens the way for practical quantitative analysis of the gains that can be achieved from instruction scheduling and cache aligned heap allocation.

6 Related Work

Many papers have been written on object oriented languages for massively parallel programming. But no successful compile time determination scheme for nearness is known to the author.

As mentioned earlier, the `withnear` assertions may be used, but it's basic limitations are obvious:

- It is inefficient (one may argue that assertions needs not be checked, but that does not conform with the typesafety requirements of strongly type languages).

- It does not include the transitive closure of all near references of an object. For example, in

  ```
  withnear x,y,z do end
  ```

  just the nearness of x,y, and z is asserted, whereas in the scheme as described in this paper, once the nearness of an object has been established using a type cast or typeswitch, the transitive closure of all near references is safely known at compile time.

At the far end of related work, parallel system's programming languages need to be mentioned.

7 Conclusions

A very simple and safe way has been described to express nearness or farness of references. Violations can be checked at compile time using a strongly typed object oriented language, by:

- making nearness and farness visible at the language level and
- extending the type system by three type coercion rules.

The scheme can be used for optimizations based on compile time knowledge of nearness and farness. The resulting scheme is superior to inherently unsafe assertion based methods.

Furthermore, this complies with the "competent programmer" model of parallel computation even in the presence of highly complex data structures, i.e., not only for fairly regular data structure as in the "compiler controlled" model of parallel programming.

Unfortunately, some limitations are induced when reusing sequential classes. Future research might remove these limitations.

The experience using far types is limited to a on the paper implementation of a general load balancing class. It turned out that, for that example, the number of far types can be kept fairly low. As expected, in some cases simple assignments had to be replaced by explicit copy operations to bring objects *near* to the focus of activity. This complies with the intention to enforce programmer's awareness of nearness or farness of references.

8 Acknowledgements

Robert Griesemer has read several earlier versions of this paper. Jeff Bilmes carefully proofread the English translation. Both helped state some of the ideas in a clearer way. Jochen Liedtke contributed to the Future Work section. The paper is the result of lengthy discussions in the pSather group at ICSI which is headed by Jerry Feldmann.

References

[1] Cray t3d. personal communication.

[2] Concurrent object-oriented programming. Communications of the ACM, September 1993.

[3] D.C. Cann. The optimising sisal compiler: Version 123.0. Technical Report UCRL-MA-110080, Lawrence Livermoore National Laboratory Manaual, 1992.

[4] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurty, S. Lumetta, and K. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing'93*, 1993.

[5] High Performance Fortran Forum. *High Performance Fortran. Language Spec.*, December 1992.

[6] S. Murer, J. Feldman, Chu-Cheow Lim, and Martina-Maria Seidel. psather: Layered extensions to an object-oriented language for efficient parallel computation. Technical Report TR-93-028, ICSI, Dec 1993.

[7] Stephen M. Omohundro. The Sather Language. Technical report, International Computer Science Institute, Berkeley, Ca., 1991.

[8] Stephen M. Omohundro. The Sather 1.0 Specification. Technical report, International Computer Science Institute, Berkeley, Ca., 1994 (in preparation).

[9] Dick Pountain. A tutorial introduction to OCCAM programming. Inmos Occam Manual.