Linda-Smalltalk: Parallel Implementation of Little Smalltalk Using Tuple Space Communication of Linda

Jean H. Chung
Timothy A. Budd
Department of ComputerScience
Oregon State University
Corvallis, Oregon 97331-3902

90-60-12

Linda-Smalltalk: Parallel Implementation of Little Smalltalk Using Tuple Space Communication of Linda

Jean H. Chung
Timothy A. Budd
Department of ComputerScience
Oregon State University
Corvallis, Oregon 97331-3902

90-60-12

LINDA-SMALLTALK : Parallel Implementation of Little Smalltalk Using Tuple Space Communication of Linda

Jean H. Chung and Timothy A. Budd
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331
chungj@cs.orst.edu, budd@cs.orst.edu

May 7, 1990

Abstract

A new technique of implementing the object-oriented language Little Smalltalk is developed using tuple space communication of Linda. In this implementation, the language's abstract constructs such as object, class, and method are naturally mapped into tuple space. Since Linda's tuple space is designed to facilitate parallel processing, this implementation technique makes it particularly easy to introduce parallelism into the language. In other words, parallelism is achieved as a natural outgrowth of the implementation technique, without bringing excessive complexity to the system. Although it may be natural in abstract level, mapping the entire objects into the tuple space turned out to be unrealistic in implementation level owing the the inherent characteristics of the tuple space.

A simple case of implicit parallelism can be detected by syntactic context of the language. General issues on object-oriented concurrent programming are also discussed.

1 Introduction

Parallel hardwares exist in a variety of forms nowadays, but thinking in simultanieties in order to compose a parallel program is not an easy task, and is usually considered a step up in software complexity. Many kinds of approaches to parallel programming have been proposed, mostly during the past decade or so, and no single approach seems to have been widely accepted among the parallel processing community.

Models of parallel computing in general can be categorized by a few orthogonal factors, for example, implicit versus explicit parallelism, and data-level versus control-level parallelism.

1.1 Explicit vs. Implicit Parallelism

Explicit parallelism involves either developing new languages or extending existing languages for handling parallel programming. Examples of language features that may be provided include constructs for creating processes, message passing as process communications, and shared variables. There are several systems built or proposed for various existing languages such as Fortran [CaK88], C [LiS85, RoS87], Pascal [Ree84, PCMM85], and Smalltalk [YoT86]. A difficulty with this approach is that new parallel languages tend to be highly dependent on the underlying machine because different parallel hardwares are usually so radically different in their architectures that they affect the ways of programming. For example, programming shared memory machine and programming hyper-cube machine usually seem to bear little resemblance.

The implicit approach, on the other hand, tries to develop *smart* translators for existing languages, without modifying the language, to detect parts of program that can be executed in parallel from a sequential program. Although this may sound fancy, its power and efficiency are practically very limited because most programs have parts that are inherently sequential. It seems we can hardly expect to come up with a *general* parallelizing translator if its *base* language does not direct the programmer towards a way of thinking in which sequential solutions are so laid out that programs can be efficiently parallelized. There is a growing agreement that, for general purpose programming, programs must be designed to be parallel, or at least, be written with parallelization in mind to make effective use of parallel hardware. In fact, research in this direction has focused mostly on restricted domains of problems, most notably, vector and matrix manipulations [Wol88, Lam74, ACK86]. Fortran has been one of the major target languges for vectorizing compiler, and a few other languages have been considered as good candidates, for example, APL [Bud84]. APL is particularly interesting in this sense since it has a large variety of built-in vector and matrix operations.

1.2 Data-Level vs. Control-Level Parallelism

On the other axis of parallel computing models, data-level and control-level parallelism contrast with each other. *Data parallel programming* [HiS86], an evolving term from late 80's, basically aims at performing a parallel *synchronous* operation on massive homogeneous *parallel data* objects, where each object can be a heterogeneous structure. In other words, the basic idea is to distribute multiple data over many processing elements and perform the same operation on all the data objects at once. This approach inherently assumes a SIMD programming model and the ability to make the number of processing elements a function of the problem size, rather than a funtion of the target machine. C* [RoS87], originally developed for Connection Machine^{TM} [Hil85], is a language based on this model. Although C* is desiged for a special synchronous hardware, it is shown that the language can be implemented on various architectures such as shared memory machine [QuH89] and hypercube [QuH88]. The systems considered in [CaK88, LiS85, Ree84, PCMM85] can also be viewed as data parallel systems in that they are interested in data partitioning. The extent of application area for data parallel model also tends to be limited to array and matrix operations.

At the other end of this axis is *control parallelism*, where we are more concerned with a functional decomposition of the problem solution, rather than data decomposition of the problem domain. Thus, problems that are inherently parallel, such as dining philosopher [Dij71, Hoa78] and producer/consumer, or that have several more or less independent processes communicating with each other from time to time, seem to fit better in this model. Most object-oriented concurrent programming (OOCP) systems [Agh85, Hew77, YBS86, YoT86] fall into this category, as well as data flow model based on functional programming [Vee86, Den80] and Hoare's CSP [Hoa78]. There has been much research done in the direction of object-oriented programming (OOP) with concurrency. Some of this will be discussed in the next section.

Our research emphasizes the control level parallelism with both implicit and explicit parallelism. The system can detect implicit parallel computation of a simple case, while allowing the user to explicitly specify parts of program to run in parallel.

2 Concurrency in Object-Oriented Programming

2.1 System model for OOCP

Object-oriented programming(OOP), in general, tends to motivate programmers to structure a problem and solution in terms of behavior of various components of the system. Typically, a big problem is decomposed into various components and each component is assigned certain responsibilities (or functionalities). Then composing a program becomes, in a sense, coordinating activities among objects by sending messages, which is sometimes called *responsibility driven design* [WiW89]. Responsibility driven design is inspired by *client-server* model. Objects send messages to other objects to accomplish something. Then the receiver object of the message might compute the information, or it might delegate the request for information to another object. This can be illustrated as in Figure 1, which we will call an *execution graph*. The figure shows a typical pattern of message passing in OOP.

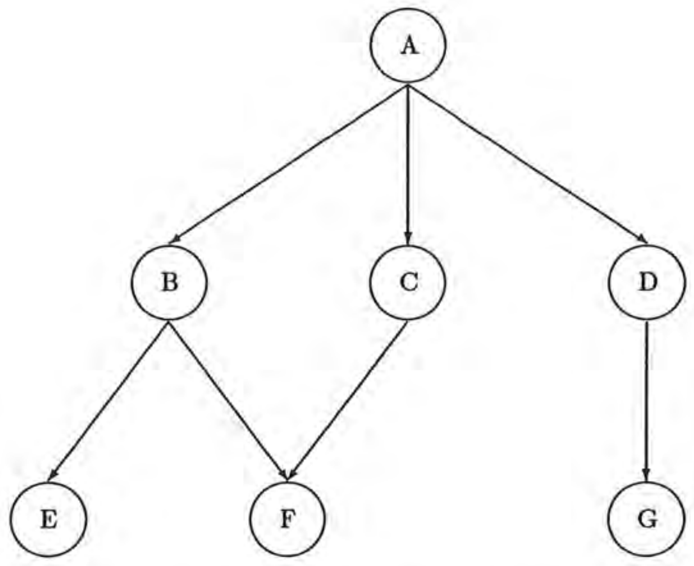
As indicated in Figure 1, the order of execution for sequential computation is `E F B F C G D` and `A`, which is like a post-order tree traversal, although the graph is not a tree. Object `A` sends messages to three different objects, `B`, `C`, and `D`, to accomplich its computation. But after `A` sending a message to `B` it waits until `B` finishes its execution before it sends another to `C`. Object `B` in turn sends messages to other objects, and so on. Thus, `A` finally terminates only after all the objects in its subgraph finish. Note, however, we have a chance of concurrency that `D`, `E` and `G` can run in parallel with `B`, `C`, and `F` since there is no interaction between the two groups of objects.

From the concurrency point of view, we can think of an object as a self-contained computational agent with a state (instance variables) and functionalities (methods or procedures). This view gives a good flavor to OOCP in that objects and processes are unified into the single self-contained notion of concurrent objects. Then a software system can be modeled as a collection of concurrently executable objects that interact with one another by sending messages. The central idea in OOCP is then to decide what and whose activities should be carried out in parallel, and how such concurrent activities should interact with another.

2.2 Difficulties of Parallelism in OOP

Although it may sound very natural and easy to introduce concurrency into OOP, in practice we are faced with at least one difficult problem, that is, serialization of interacting messages. In Figure 1, for example, there must be a way to seralize messages to F so that the message from `C` can not even get started before the message from `B` finishes, for parallel execution should yield the same result as the sequential execution.

Unfortunately, however, the execution order is *dynamically* determined in most existing OOP languages. In other words, objects and methods (or procedures) get bound dynamically so that the structure of execution graph is determined only at run time. This feature makes it extremely difficult or even impossible to do such things as implicit parallelism detection or compile time concurrency analysis. There have been several approaches to get around this problem.



The order of execution is E F B F C G D A.

Figure 1: An execution graph for a typical pattern of message passing in OOP

2.3 Solutions Proposed - Related Researches

2.3.1 Roll-back mechanism

This solution is proposed by [BBP88] for implicit parallelism. The idea is to send all messages for execution in parallel, and resolve message conflicts *at run-time.* The system checks the run-time interactions using *time stamps* on messages. When the system detects a message that is processed in wrong order in a program, it forces the program to roll back to a previous state using what they call input queue, output queue, and state stack.

Although we may be able to build such a system that *works* correctly, the system itself should involve a fair amount of overhead and inefficiency because it must remember all the states of execution history until it becomes convinced that there is no conflict. In addition, when it rolls back, it simply has to abandon a certain amount of computation and re-compute that much.

2.3.2 Message queue

This approach is adopted by Actors [Hew77, Agh85], ABCL/1 [YBS86], and Concurrent Smalltalk [YoT86]. An object is capable of processing one message at a time, but it can be sent multiple messages. Each object maintains a message queue, where messages are kept in the order that they arrived, and thus messages sent to an object is *serialized* by the queue. An object takes a message from the queue, processes the message, and when done, takes the next one, and so on. An object remains *passive* when there is no message in the queue.

2.3.3 Future object

Actors, ABCL/1, and Concurrent Smalltalk all rely on the notion of *future object* (or *future message*), in one way or another. A future object acts as a place holder for a result of a message. After sending a message, a program can continue its execution without waiting for a result to return if it is not needed immediately, and may use the result later.

Consider, for example, the segment of code[1] in Figure 2. We use an "&" character at the end of a message to express that the message is to be sent in parallel (future message). When `msg1` is sent, a special object `F`, a *future object*, is created and assigned to `a` by the system, and a pointer to `x`, the receiver object, is given to `F` so that `x` can return its result to `F` later. Then the program continues without waiting for `x` to return. When the actual value of `a` is needed, i.e., when we send `msg2:` in the example, it is taken from the future object, `F`, if available, or the program blocks until `x` finishes its computation and returns the result to `F`.

Future objects are used as a mean to achieve *synchronization* among asynchronous messages. ABCL/1 provides three types of message passing: past type, now type, and future type. *Past* type message simply creates a new asynchronous process: it sends and goes on. A *now* type message is a synchronous message which waits until the message is finished. A *future* type message is an asynchronous one that can be synchronized by its future object.

[1] The syntax we use in Figure 2 does not assume any particular system

```
a <- x msg1 &
...
some computation in between
...
b <- y msg2:  a
...
rest of computation
...
```

Figure 2: Behaviour of Future Object

2.4 Our Approach

Our approach is closely related to the concept of future object and future message. We use Little Smalltalk[2] [Bud87] as the base language. This base language is augmented with parallel processing capabilities based on the tuple space communication of Linda[3] [Gel85, CaG88]. We first map the entire Little Smalltalk objects into the tuple space, and then structure the interpreter of the language as processes around the tuple space so that processes can communicate and interact with each other through the tuple space. Future objects can be easily implemented using this technique as will be explained later.

Similar works have been done by Matsuoka and Kawai [MaK88]. They also adopted the notion of *tuple space communication* to implement an OOCP language, *Tuple Space Smalltalk*, whose base language is Smalltalk-80. They proposed some useful transformations on the semantics of communication model originally provided by Linda to make it more efficient and suitable for their language.

3 Linda – Tuple Space and Its Operations

The system Linda [Gel85, CaG88] is designed to facilitate parallel programming by means of tuple and tuple space. Linda consists of a small number of primitive operations which can be built into a *host* language, such as C, C++, Fortan, and Smalltalk, to build a parallel dialect[4] of the host language.

The basic idea is to let processes be able to communicate with each other by placing and taking tuples into and out of a special kind of abstract shared memory[5], called *tuple space* (see Figure 3). Tuples play the

[2] *Little Smalltalk* is a dialect of Smalltalk-80 [GoR84]. It is a textual language, that is, without the graphic user interface of Smalltalk-80.

[3] *Linda* is a kind of parallel programming environment that is designed to make communication and interaction among processes easy. See Section 3.

[4] The resulting languages are called, by convention, C-Linda, Fortran-Linda, and so on.

[5] This does not mean that Linda can be built only on shared memory machine, although it is easier and efficient. There are versions of Linda on message passing machines.

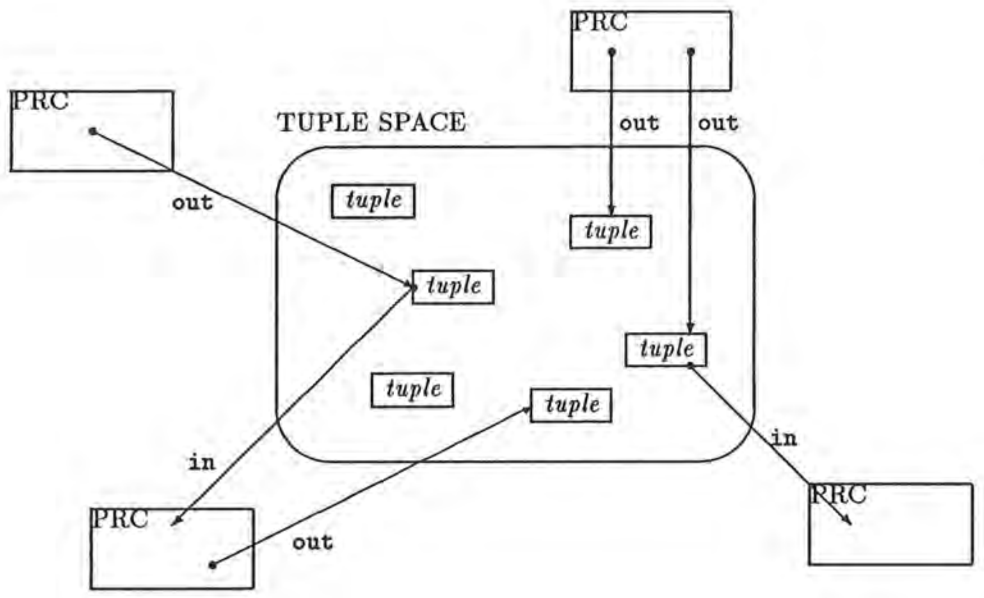


Figure 3: Processes, Tuples, and Tuple Space in Linda

role of *messages*[6] passed among processes. They may contain information of arbitrary type that the host language supports. In order for a process to send or receive a message, it needs not specify the destination or source of the message, respectively. Instead, a process can place any tuple into tuple space, and can take any tuple out of tuple space. One nice thing about Linda is that the notion of tuple and tuple space is such a high-level concept that Linda can be built on several different parallel architectures without loosing portability.

3.1 The Linda Model

The Linda model is a memory model. Linda's abstract memory, called *tuple space*, is the basis for process creation, communication, and synchronization. The tuple space serves as a storage for tuples, and can be thought of as a high-level associative memory that is addressed by pattern matching.

[6]Note the term *message* in this context has little to do with the term used in OOP.

A *tuple* is an ordered collection of data of any type, and of any number. Conceptually, tuples are very much like records in that they may consists of heterogeneous data elements of any number. Examples of tuples are:

```
[ "any string", 3.14, x, -65536, f(y) ]
[ "RS", reply_id, ?result ]
```

A tuple element prefixed by a question mark, e.g. `result`, is called a *formal* element, and one without a question mark an *actual* element. A formal element is like a place holder to which a value can be assigned. Any two tuples are said to *match* each other, if:

1. they have the same number of elements,
2. elements on the corresponding positions in the two tuples have the same value, in case both are actual elements,
3. elements on the corresponding positions in the two tuples have the same type, in case one is an actual element and the other formal.

For example, the following two tuples match,

```
[ "RS", 3.14, ?i ]
[ "RS", 3.14, 89 ]
```

if `i` is of type `integer`. The following two tuples may or may not match depending on the value of `x`, which is of type integer.

```
[ "MG", 1, x ]
[ "MG", 1, 9 ]
```

Tuples are manipulated by a few basic operations of Linda, as described in the next section.

3.2 Tuple Space Operations

Linda has been implemented on various systems[7], and some are somewhat different from others in syntax and in the way tuples are formed, although the basic ideas are more or less the same. In this section, we will try to show the general idea without refering to any particular Linda implementation in our notations.

3.2.1 Writing operations - `Out`

A tuple can be placed into the tuple space by `out` primitive.

`out(` *<a tuple>* `)`

[7] The version of Linda we use is SCA^{TM} v2.0 on the parallel computer Sequent/Balance^{TM}.

All the elements of the tuple get evaluated *before* it is stored in the tuple space. For example, if an `out` is done as follows:

```
out(x, foo(x, y), 3.14)
```

first, `x` and `foo(x, y)` will be evaluated, say to `3` and to `0.123`, respectively, and then a tuple of the form

```
[ 3, 0.123, 3.14 ]
```

will be placed in the tuple space. If a formal element is used in an `out`, it just gives type information.

3.2.2 Reading operations - `In` and `Rd`

There are two kinds of operations for reading a tuple from the tuple space: destructive read (`in`), and non-destructive read (`rd`).

```
in( <a tuple> )
rd( <a tuple> )
```

Both operations search the tuple space to find a tuple that matches their operand tuple. If one is found, `in` takes it out of the tuple space, whereas `rd` simply *reads* the tuple leaving it there in the tuple space. If there are more that one tuple that matches the operand tuple, any one is selected *non-deterministically*. If a matching tuple is *not* found, then the process that used one of these operations suspends until there appears one. This is the way Linda enforces *synchronization*.

If a formal elements is used in `in` or `rd` operation, it gets bound to the value of the corresponding element of the matching tuple. For example, suppose we have three tuples in the tuple space.

```
[ "MESSAGE", 1, 7, 3.14 ]
[ "MESSAGE", 1, 8, 3.14 ]
[ "MESSAGE", 1, 9, 3.14 ]
```

Then, a reading operation

```
in("MESSAGE", 1, ?i, ?r)
```

will take out any one of the three tuples non-deterministically, and `i` will get bound to either `7`, `8` or `9`, and `r` to `3.14`. If we used `rd`, the tuple that matched would still be in the tuple space while `i` and `r` get bound in the same way.

3.2.3 Predicate functions - `Inp` and `Rdp`

Linda provides two predicate functions, `inp` and `rdp`, to be able to test to see if there is a matching tuple in the tuple space.

```
inp( <a tuple> )
rdp( <a tuple> )
```

`Inp` and `rdp` work exactly the same as `in` and `rd`, respectively, except that they return either `true` or `false` depending on whether there is a matching tuple or not, without suspending the process that used them.

3.2.4 Process creation - `Eval`

Conceptually, Linda treats a process as a special tuple, called an *active* tuple. So process creation is simply done by placing an active tuple into the tuple space. An active tuple is created by **`eval`** operation.

```
eval( <a tuple> )
```

Unlike `out` which evaluates all its tuple elements before placing the tuple, **`eval`** blindly put its argument tuple in the tuple space, and the process that issued **`eval`** continues its execution. Then the tuple is said to be evaluated *in* the tuple space. When all the elements are evaluated, an active tuple becomes *passive*. Tuples placed in the tuple space by `out` are all passive.

Consider the following example.

```
eval(x, foo(x, y), 3.14)
```

In this case, the process that executed the **`eval`** operation does not take time to wait for **`foo`** to return a value. Instead, it continues its execution while the **`eval`**'s argument tuple is being evaluated in the tuple space. In effect, a new process is created to evaluate the function **`foo`**.

3.3 Linda Examples

The parallel programming environment provided by Linda's tuple space and its operations is very powerful and flexible so that wide variety of computational abstractions can be expressed elegantly within its paradigm. Two simple examples related to the later discussions are given in this section: one for message passing mechanism[8] in the form of remote procedure call, and the other for data structure implementation.

3.3.1 Remote procedure call

Remote procedure call is a kind of higher level message-passing constructs [AnS83, LiS82] which is designed to make it easy to program client/server interactions. They are used in the following general form.

```
call service(value_args; result_args);

remote procedure service (in value_parameters; out result_parameters)
begin
    procedure body
end remote procedure
```

This can be easily expressed in Linda as shown in Figure 4. A client can ask for a service by dropping a tuple that represents a service request. The remote procedure is in an infinite loop waiting for a service request by `in` at the beginning of the loop. When there appears a request tuple in the tuple space, `in` takes the tuple, and the the loop body goes on to compute its service, and `out` gives back its answer through the tuple space so that the client can take it and resume its work. Note that a single server can serve various clients by using `reply_tag`, and that a client can do a certain amout of work while the service is being done.

[8] Again, the term *message passing* in this context has little to do with OOP.

```
remote procedure call :
    ...
    out("request", reply_tag, arg1, arg2, arg3);
    ... maybe, some work in between ...;
    in("reply", reply_tag, ?answer);
    ...


procedure service()
begin
    while (TRUE) begin
        in("request", ?client, ?param1, ?param2, ?param3);
        ...
        compute answer;
        ...
        out("reply", client, answer);
    end;
end-procedure;
```

Figure 4: Remote Procedure Call Implemented in Linda

```
[ "A", 1, 1, val11 ]    [ "A", 1, 2, val12 ]    [ "A", 1, 3, val13 ]
[ "A", 2, 1, val21 ]    [ "A", 2, 2, val22 ]    [ "A", 2, 3, val23 ]
[ "A", 3, 1, val31 ]    [ "A", 3, 2, val32 ]    [ "A", 3, 3, val33 ]
```

Figure 5: A 2-dimensional Array Represented as Tuples

3.3.2 Data structure representation - An array

Tuples can be used as data representation scheme, as well as message passing among processes. This is particularly useful when we need structured data items *shared* by multiple processes. For example, a two dimensional array can be repersented as shown in Figure 5. With this representation, we can update the value of `A[i,j]` by

```
in("A", i, j, ?val);
val :=  a-new-value;
out("A", i, j, val);
```

Although this looks a little ugly, it is convenient since both `in()` and `out()` operations are made to be *atomic*, a user need not concerned about race condition.

4 Little Smalltalk - Implementaion Issues

Little Smalltalk is a subset of Smalltalk-80 [GoR84], and has most basic features of Smalltalk-80 except the graphical user interface. This section discusses the implementation issues, as well as some of the fundamental aspects, of Little Smalltalk[9] which will be contrasted against those of Linda-Smalltalk described in the next section.

4.1 Distinguishing Characteristics of OOP Systems

There are a few distiguishing characteristics that make OOP systems different from other kinds of programming enviroments. Among these are *message passing* for requesting action (computation) and composition of abstract data types through *classification*. A *class* in OOP represents an abstract data type that has both state (*instance variables* in Smalltalk terms), and functionalities (*methods*). Classes are generally structured as superclass-subclass hierarchy to take advantages of the mechanism called *inheritance*.

Message passing is like a procedure call, except that the actual code for a message, a method, gets bound dynamically at run time depending on the *receiver*, the first argument of the message. Consider, for example,

[9]Little Smalltalk has evolved since it first came out. The one we use as the base language in this research is version 3.05. The book *A Little Smalltalk* [Bud87] is written for version 1.0.

the following message.

```
a foo: x bar: y
```

`a`, being the first argument, is called the receiver of the message `foo:bar:`. `x` and `y` are the second and third arugments of the message, respectively. There may be more than one method having the same name in the system, and which one to execute for a message is determined at run time depending on the class of the receiver. First a search is made, in the above example, for a method named `foo:bar:` in the class of the receiver, `a`. If such a method is found there, execute it. Otherwise, another search is made in the super class of the class of `a`, and so forth.

Note that this is how we achieve inheritance through class hierarchy. The class of `a` may not need to define its own method for the message `foo:bar:` to have the functionality. It can simply *inherit* from one of its super classes. Inheritance is a powerful mechanism by which we can enhance code reusability, code sharing, and polymorphism.

4.2 Representation of Objects

As discussed in the previous section, message passing is like a procedure invokation with an overloaded procedure name which gets resolved at run time based on the receiver's class and class hierarchy. Thus, besides what is needed for procedure call, e.g., activation record and argument evaluation, we also need the information on the class of the receiver and the class hierarchy. This means objects must carry their class information, and classes must know what methods they have and what their super classes are.

In Little Smalltalk, an object is represented as a C-structure as shown in Figure 6. All the objects in the system are stored in a big array of `objectStruct`, called `objectTable`[10]. The data type `object` is an integer type so that reference to an object is just an index to the object table. A class is simply an object of class `Class` which has instance variables for, among others, methods and super class.

4.3 The System Structure

Figure 7 shows the structure of the system: interpreter, scheduler, process, and object table. Just about everything in Little Smalltalk is object, including class, method, process, and scheduler. The figure, in fact, is not accurate in that processes and process stacks are really part of the object table.

Interpreter is a software module that acts like a CPU for a virtual message passing machine[11], whose machine language is called *bytecode* (see Section 4.4). So, we can view the system as a virtual machine with the interpreter as the CPU and the object table as main memory. Then there is a scheduler that allocates processes to the single CPU (interpreter) in round-robin fashion. A process consists of a process stack (or an execution stack) and a stack pointer. A process stack is initialized to contain, among other things, a method to execute and arguments. The interpreter is given a process to execute.

[10] Actually, the object table does not store instance variables. Instead, it keeps a pointers to heap areas where instance variables are stored.

[11] *Message passing* in OOP's sense.

```
struct objectStruct {
        object class;           /* class of object */
        short referenceCount;   /* for garbage collection */
        short size;             /* number of instance variables */
        object *memory;         /* pointer to heap where inst vars are stored */
        }

objectStruct objectTable[MAX_OBJECT];
```

Figure 6: Structure of an Object and Object Table in Little Smalltalk

4.4 The Representation of Methods - Bytecodes

Method is just another name for procedure in conventional programming languages. In Little Smalltalk, a method is represented by an internal form called *bytecodes* together with *literals* array. We can think of bytecodes as assembly language instructions for a virtual stack-based message passing machine.

Bytecodes are best understood by examples. Consider a method for a message named **`isEmpty`**:

```
isEmpty
    ^ (self size) = 0
```

This method returns either **`true`** or **`false`** depending on whether the receiver (**`self`**) of the message is empty or not, that is, whether the size of the object is zero or not. The body of the method consists of two messages: **`size`** tells the size of an object, and = tests object equality. This method would be represented in the bytecode format as shown in Figure 8. Each line of bytecodes is encoded as a byte: the upper nibble[12] for the opcode, and the lower nibble for the argument of the opcode.

There are 14 opcode instructions and 10 special instructions used in the system. Conceptually, they can be classified into four groups: *stack push*, *stack pop*, *message send*, and *return*. Since the only control mechanism in Smalltalk is message passing, these are used mainly to support message passing: *push* arguments onto stack, *send* message with all the arguments, and then *return* the result, assigning objects (*pop*ping the top of the stack into an object) occasionally . The opcode **`markArguments`** tells how many arguments there are for the message to be sent next.

5 Implementation Using Linda

As once briefly mentioned earlier, Linda's tuple space is used basically for two purposes: shared data representation and message passing mechanism.

[12] A *nibble* is a half of a byte, 4 bits.

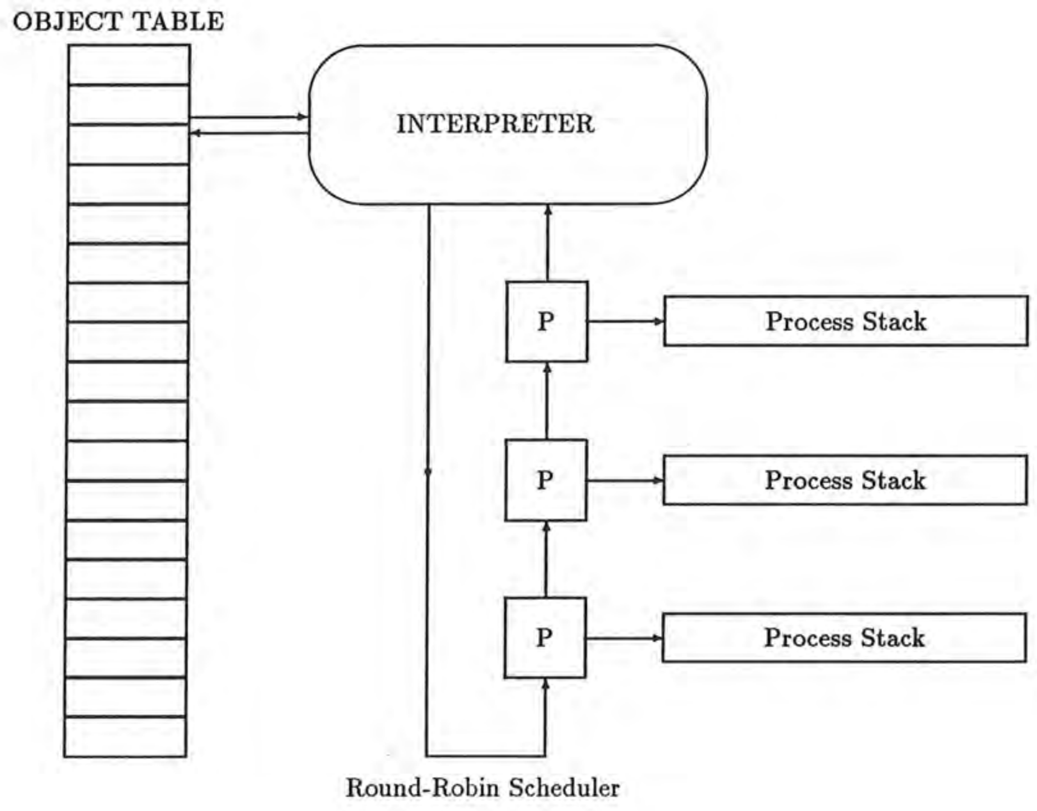


Figure 7: Process Execution Model in Little Smalltalk

opcode	argument	comment
`PushArgument`	`1`	; push 1^{st} arg onto stack
`MarkArguments`	`1`	; there are 1 arg for next message
`SendMessage`	`1`	; msg name is 1^{st} elmt of `literals`
`PushLiteral`	`2`	; literal is 2^{nd} elmt of `literals`
`MarkArguments`	`2`	; there are 2 args for next message
`SendMessage`	`3`	; msg name is 3^{rd} elmt of `literals`
`ReturnStack`	`noarg`	; return top of the stack

`literals : #("size", 0, "=")` ; an array of objects

Figure 8: Bytecode Representation of Method `isEmpty`.

5.1 Object Representation

Since we will have multiple processes running concurrently (see Section 5.2 below), and all the processes need to have the ability to access any object in the system, the *object table* need to be allocated in some sort of *shared memory*, a kind of memory that all the processes can access. Linda memory, the tuple space, serves well for this purpose. It gives a clean interface for manipulating tuples, and its operations are *atomic* so that we are relieved from dealing with race conditions.

All the objects, including classes and methods, are mapped into tuple sapce. An object is represented by two tuples repesenting class and size, and zero or more tuples for instance variables depending on size. For example, an object with three instance variables is mapped onto the tuple space as shown in Figure 9.

Note that we can use the same representation scheme consistently for objects of class `Array` by keeping as many tuples as the number of elements of an array, although an `Array` object conceptually has only one instance variable, an array. For `ByteArray`[13] objects, however, we keep one tuple for the whole array for efficiency because `ByteArray` objects are typically used as a whole entity, not element by element.

5.2 Overall System Structure

Figure 10 shows the overall structure of the system: interpreter, tuple spaces[14], listener, and process. Note that each process has its own interpreter, where as in Little Smalltalk (Figure 7) there is only one interpreter that executes each process in turn for a time-slice in round-robin fashion. This is a good contrast, with the analogy of single CPU virtual machine view of Little Smalltalk, in that now we can think of Linda-Smalltalk

[13]`ByteArray` is an array whose elements are of byte size.

[14]The version of Linda we use has only one global tuple space, while some others can have multiple tuple spaces. A tuple space can always be logically divided into any number of sub-spaces by adding in every tuple a special tuple element for designating the spaces.

```
[id, "CL", class-object]      ; class
[id, "SZ", 3]                 ; size
[id, "IV", 1, object-x]       ; first instance variable
[id, "IV", 2, object-y]       ; second instance variable
[id, "IV", 3, object-z]       ; third instance variable
```

† `Id` is an id number (an integer) of the object.
‡ 1, 2, and 3 in `"IV"` tuples serve as object indices.

Figure 9: Tuples Representing an Object with Three Instance Variables

as multiple-CPU virtual machine. Also note that process stack is not a part of a process any more, but is a part of the interpreter.

In fact, the notion of process is one of the major features that make Linda-Smalltalk different from Little Smalltalk. In Little Smalltalk, a process is just an object, and it needs to be allocated the interpreter by the scheduler. In Linda-Smalltalk, on the other hand, a process is a sort of an *autonomous* computational agent that is capable of executing itself, and therefore, need not be scheduled. We actually dispense with the Little Smalltalk class `Scheduler` and *even with* the class `Process` !. Processes get created on the fly either by `fork` or by implicit parallel messages, execute themselves, and terminate when done.

Listener is an independent system process that keeps *listening* to the message space to see if there is any message to execute in parallel. The code for listener is shown in Figure 11. If there is a tuple that represents a asynchronous message, then listener takes the tuple and forks off yet another interpreter process. So a user process can request for a new process by simply placing a tuple that represents an asynchronous message in the message space. From users' point of view, placing a tuple in message space is like a remote procedure call for a process creation service. Note the similarity between the listener and the remote procedure (Figure 4). The mechanism for asynchronous message passing is explained in detail in Section 5.4.

5.3 Implicit Parallelism Detection

The parser can detect implicit parallelism by syntactic context. We deal with only a simple case of implicit parallelism - concurrent execution of arguments of a message. Consider, for example, the following message:

```
(a foo) aMessage: (b bar) withArg: c
```

Here the message `aMessage:withArg:` has three arguments, `(a foo)`, `(b bar)`, and `c`, and two of these arguments are results from another messages, `foo` and `bar`. Note that since the order of argument evaluations is of no significance, we can evaluate them in parallel without problems. If a message of such a pattern is found the parser generates code that causes the interpreter to create new interpreter processes for evaluating the argument messages.

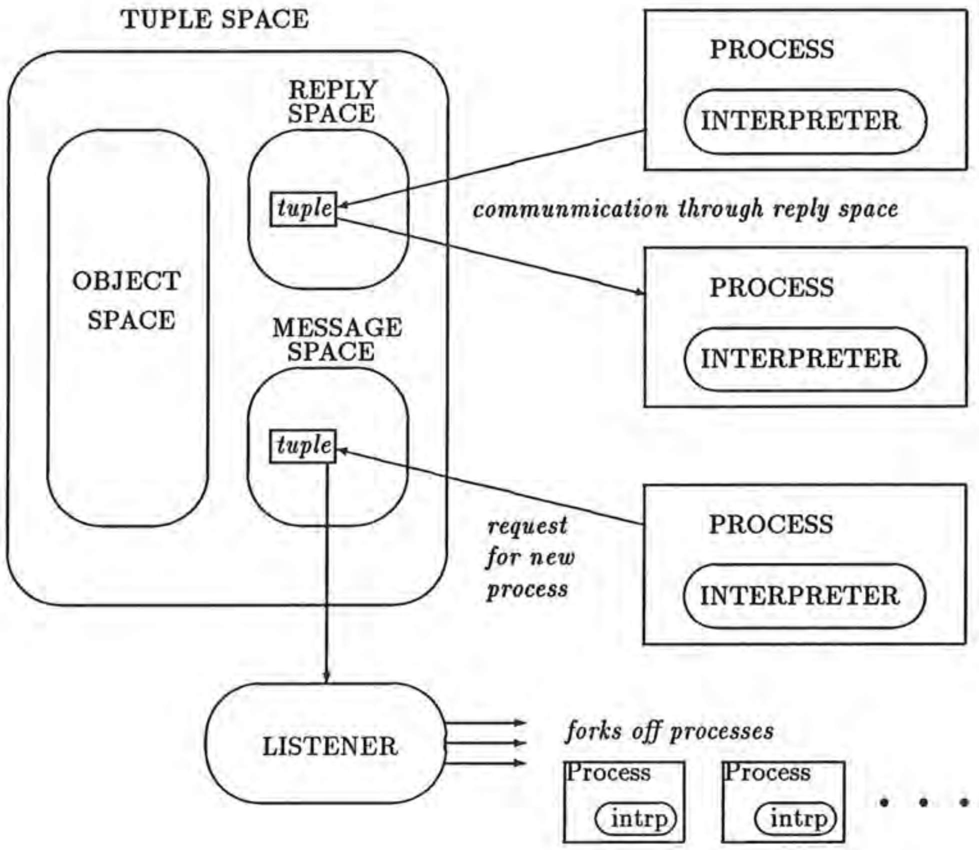


Figure 10: Process Execution Model in Linda-Smalltalk

```
void listener ()
{   int replyTag;
    object message, method, arguments;

    while (TRUE) {
        in("MESSAGE SPACE", ?message, ?arguments, ?replyTag);
        ...
        find method for message;
        ...
        eval("REPLY SPACE", replyTag, interpreter(method, arguments));
        }
}
```

Figure 11: Listener Process in Linda-Smalltalk

5.4 Parallel Message Passing Mechanism

In Little Smalltalk, when a message is to be sent with more than one arguments, the interpreter evaluates each argument in turn and pushes the value onto a process stack, and then sends the message with the argument values. However, in Linda-Smalltalk, in order to evaluate all the arguments concurrently, an interpreter forks off processes, one for each argument that is to be evaluated in parallel, giving each process a unique id number, called a *reply tag*. The interpreter uses the reply tags later to collect the results through reply space. So, instead of pushing the values of arguments, the interpreter pushes the reply tags, and when it is time to pop, it pops the reply tag and uses the tag to get the result from the reply space. But, we need to have a certain way to distinguish whether the item popped off from the stack is a tag or an normal object. A special object called tagMarker, which is not visible to users, serves this purpose. Figure 12 shows the C macros used for stack manipulation.

Note that we use `in()` to get the actual value for a reply tag. This is how processes usually get synchronized in Linda program. Here, when popping the stack, the interpreter waits by blocking itself with `in()` until the child processes finish their argument evaluation.

All that the parser should do for a parallel message detected is to produce code for calling a special primitive. This primitive causes an interpreter to simply put out in the message space a tuple containing the message and a unique *reply tag*, instead of the interpreter executing the message. Then, the listener takes this tuple from the reply message space, and forks off an interpreter process to evaluate the message, giving the child interpreter a reply tag, the method to execute, and arguments[15]. This is to be contrasted with Little Smalltalk where the interpreter is given a process that has a process stack in it. (See Section 4.3.)

[15] Actually, an interpreter is given a reply tag and a special object, called `Context` object, which contains method and arguments. We avoid refering to context objects here for ease of discussion.

```
push(x) *++sp = x

pushTag(x) *++sp = replyTag; *++sp = tagMarker

pop(x) x = *sp; *sp-- = 0;
       if (x == tagMarker)
           { replyTag = *sp; *sp-- = 0; in("RS", replyTag, ?x) }
```

† `sp` is a stack pointer.

Figure 12: Stack Manipulation for Reply Tags

The new process returns the result to its parent process through the reply space using the reply tag.

5.5 The Interpreter

The interpreter is the heart of system. It can be thought of as a software central processing unit (CPU) for a hypothetical stack machine. Given a method and arguments, it processes the *bytecodes* associated with the method. The interpreter basically loops over each bytecode until the method it is given returns. The skeletal C code for the interpreter is shown in Figure 13.

6 Problems Encountered and Possible Extensions

Although the current approach is conceptually simple and natural in mapping objects into the tuple space and in structuring the interpreter around the tuple space, it has revealed some critical problems at the implementation level. The system is unacceptably slow, mainly due to the way the tuple space is used. There is, however, a possible remedy to this problem. In addition, the system can be easily extended to be augmented with more flexible inter-process communications without introducing too much complexity.

6.1 Problems with the Current Approach

6.1.1 Use of tuple space

Although Linda itself is already slow in its tuple space searching, it should be more of the way the tuple space is used that makes Linda-Smalltalk really slow. Mapping the entire objects into the tuple space makes the tuple space far larger than can be searched in a reasonable amount of time. Incidentally, this argument seems

```
object interpreter ( method, arguments )
   object method, arguments

{  variable declarations including processStack;

   ... some initialization ...
   while (not bottom of processStack) {
      get the next bytecode
      switch (bytecode) {
         case PushInstance:  ...;
         case PushArgument:  ...;
         ...
         case PopInstance:  ...;
         case PopArgument:  ...;
         ...
         case MarkArguments:  ...;
         case SendMessage:  ...;
         ...
         case DoPrimitive:  ...;
         ...
         case ReturnStack:  ...; /* a special opcode */
         } /* switch */
      } /* while */
} /* interpreter */
```

Figure 13: Code for the Interpreter in Linda-Smalltalk

to be true of any Linda application software in general: in order to have a reasonable speed performance, tuple space should not be loaded with too many tuples.

As mentioned in Section 3.1, we can think of the tuple space a high-level associative memory which is addressed by pattern matching. Therefore, a tuple access in the tuple space is inherently time-consuming and can not be done on less than linear time-order. We may expect the searching time would be of $\theta(n)$, of $\theta(n^2)$, or even worse than that, depending upon the way that Linda is implemented, where n is the number of tuples residing in the tuple space.

In the mean while, we learned that it is important for an interpreter-based system like Smalltalk that the system be able to ***randomly access*** any object in a constant time regardless of the number of objects in the system. In Smalltalk, data and program are uniformly represented using ***objects***. Therefore, the number of objects in the system is almost linearly proportional to the size of programs This means that, if we use a kind of object memory that can ***not*** be accessed in a constant time, the execution time depends on the size of the program as well as on that of data. The situation is even worse particularly for Smalltalk if we consider the fact that Smalltalk already has a considerable amount of ***basic system codes*** without which the system can not work. We have to pay for the system codes as well as the application codes. In short, we have to use a kind of randomly accessable memory, like an array, for storing objects to avoid this problem.

The only reason that we mapped the entire object table onto the tuple space is to make it possible for *all* processes to access any object. If, however, Linda-Smalltalk is to be implemented on a shared memory machine, we may speed up the system vastly by allocating the object table as an array in the shared memory, instead of mapping the object table onto the tuple space. In this case, the tuple space would be used only for asynchronous message passing mechanism, and the system is expected to be much faster that it currently is. But the version of Linda we used does not allow a user program to access the shared memory directly. An earlier project [RaB89] implemented Little Smalltalk on Sequent/Balance^{TM} using the shared memory to have a *real* muti-processing Little Smalltalk without changing the semantics of the language except that of the message `fork`.

6.1.2 Grain size

Linda seems to suit better for larger grain size. Since most methods in Smalltalk is small in its code size, and since the unit of parallelism is a message in our approach, the overhead of process creation and interprocess communication can overwhelm the possible speedup we gain from paralellism.

6.2 Possible Extensions to the Current Approach

6.2.1 Smalltalk-Linda – not Linda-Smalltalk

Since we have built Little Smalltalk on top of Linda, it should be fairly staightforward to introduce the tuple space operations, `in`, `out`, etc., to the language itself. This will make interprocess communication in Linda-Smalltalk considerably easier. (The resulting system should, however, be called Smalltalk-Linda, not Linda-Smalltalk.)

6.2.2 Explicit parallel construct – Future object

It should also be straightforward to adopt the notion of *future object* into Linda-Smalltalk. (See Section 2.3.3 for future object.) In our implementation scheme, a future object would have one instance variable:

replyTag. Then an access to every objects must be preceded by a test to see if it is a future object or not. If an object is a future object, **in** operation must be done with the **replyTag** of the future object to get the real value of the object. At this point, **in** may suspend if the future object is not yet assigned a value. This technique is very similar to the one in which we used **markerObject** to handle the process stack in Section 5.4. Future objects, like **markerObject**, is for the system's use only: a user need not deal with it directly.

7 Concluding Remarks

A new way of structuring an OOP language interpreter around multiple processors is explored using tuple space communication provided by Linda. In this implementation scheme, Linda's tuple space is used for two different purposes: object representation and parallel message passing mechanism. Conceptually, both of these purposes are well fullfilled in that the object representation is simple and natural and that the parallel message passing mechanism through inter-process communication is elegantly implemented by using `in()` and `out()` operations of Linda. Particularly, the multiple processor analogy of the system viewed as a virtual machine (Figure 10, Section 5.2) is so clear, easy to understand, and easy to implement that it would serve as a good model for structuring a multi-processing interpreter for object-based languages.

Linda-Smalltalk, however, turned out to be far too slower than expected. As mentioned in Section 6.1.1, this is mainly because of the way the tuple space is used for object mapping. Another source of difficulty is the grain size of parallelism. The unit of parallelism in our approach is a Smalltalk message, which is rather small in its amount of computations. The process creation operation in Linda, **eval()**, however, seems fit better for larger grain size. As a result, the overhead of process creation overwhelmed the potential performance gain we might have by sending parallel messages.

We started out this research with a slightly more ambitious goal, that is, exploring implicit parallelism in object-oriented programming possibly with some amount of performance gain. The semantics of Smalltalk, after all, does not seem to provide good chances for implicit parallelism. As explained in Section 2.2, there is at least one restricting factor in OOP in exploring implicit parallelism: the code to execute for a message passed gets bound dynamically at run time. This fact makes it extremely difficult or evern impossible to do things like static concurrency analysis or implicit parallelism detection. It is our belief that in order for a sequential program to be efficiently converted for parallel execution, the program written in the *base* language must somehow reflect the parallel structure of the problem. In other words, the base language must be designed to direct programmers towards a way of thinking in which solutions are envisioned with parallelization in mind, although not explicit.

References

[ACK86] Allen, J.R., Callahan, D., and Kennedy, K., "Automatic Decomposition of Scientific Programs for Parallel Execution", Dept. of Computer Science, Rice University, Houston, TX, Nov. 1986.

[Agh85] Agha, G., "A Model of Concurrent Computation in Distributed Systems", AI Technical Report 844, MIT, 1985.

[AnS83] Andrews, Gregory R. and Schneider, Fred B., "Concepts and Notations for Concurrent Programming", *Computing Surveys*, Vol. 15, No. 1, March 1983, pp. 3 – 43.

[BBP88] Bensley, E.H., Brando, T.J., and Prelle, M.J., "An Execution Model for Distributed Object-Oriented Computation", *OOPSLA*† *88 Conference Proceedings*, 1988, pp. 316 – 322.

[Bud84] Budd, Timothy A., "An APL Compiler for a Vector Processor", *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 3, July 1984, pp. 297 – 313

[Bud87] Budd, Timothy A., *A Little Smalltalk*, Addison-Wesley, 1987.

[Bud89] Budd, Timothy A., "Talking to Linda", Technical Report, Dept. of Computer Science, Oregon State University, 1989.

[CaG88] Carriero, Nicholas and Gelernter, David, "Linda in Context", Research Report, Yale University, Department of Computer Science, 1988.

[CaK88] Callahan, David and Kennedy, Ken, "Compiling Programs for Distributed-Memory Multiprocessors", *The Joural of Supercomputing 2*, 1988, pp. 151 – 169.

[Den80] Dennis, J.B.,"Dataflow Supercomputers", *Computer*, Vol 13, No. 4, November 1980, pp. 48 – 56.

[Dij71] Dijkstra, E.W., "Hierachical Ordering of Sequential Processes", *Acta Informatica*, Vol. 1, 1971, pp. 115 – 138.

[Gel85] Gelernter, David, "Generative Communication in Linda", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, January 1985, pp. 80 – 112.

[GoR84] Goldberg, Adele and Robson, David, *Smalltalk-80: The Language and Its Implementaition*, Adison-Wesley, 1984.

[Hil85] Hillis, W. Daniel, *The Connection Machine*, MIT Press, Cambridge, Massachusetts, 1985.

[HiS86] Hillis, W. Daniel and Steele, Guy L., Jr., "Data Parallel Algorithms", *Communications of the ACM*, Vol. 29, No. 12, December 1986, pp. 1170 – 1183.

[Hew77] Hewitt, C., "Viewing Control Structures as Patterns of Passing Messages", *Journal of Artificial Intelligence*, Vol. 8, No. 3, 1977, pp. 323 - 364.

[Hoa78] Hoare, C.A.R., "Communicating Sequential Process", *Communications of the ACM*, Vol. 21, No. 8, August 1978, pp. 666 – 677.

[Lam74] Lamport, L., "The Parallel Execution of DO Loops", *Communications of the ACM*, Vol. 17, No. 2 (Feb 1974), pp. 365 – 382.

[LiS82] Liskov, B.L. and Scheifler, R., "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", *Proceedings of 9th ACM Symposium: Principles of Programming Languages*", Jan 1982, pp. 7 – 19.

[LiS85] Li, Kuo-Cheng and Schwetman, Herb, "Vector C: A Vector Processing Language", *Journal of Parallel and Distributed Computing 2*, 1985, pp. 132 – 169.

[MaK88] Matsuoka, Satoshi, and Kawai, Satoru, "Using Tuple Space Communication in Distributed Object-Oriented Languages", *OOPSLA*† *88 Conference Proceedings*, September, 1988, pp. 276 – 284.

[Str86] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1986.

[QuH88] Quinn, Michael J. and Hatcher, Philip J., "Compiling C* Programs for a Hypercube Multicomputer", *SIGPLAN Notices*, Vol. 23, No. 9, September, 1988, or *Proceedings of the ACM/SIGPLAN PPEAL*, 1988.

[QuH89] Quinn, Michael J. and Hatcher, Philip J., "Implementing a Data Parallel Language on a Tightly Coupled Multiprocessor", Technical Report CSTR-89-60-20, Dept. of Cpmputer Science, Oregon State Univesity, 1989.

[RoS87] Rose, John R. and Steele, Guy L, Jr., "C*: An Extended C Language for Data Parallel Programming"', Thinking Machines Technical Report PL87-5, April 1987.

[YBS86] Yonezawa, A., Briot, J-P., and Shibayama, E., "Object-Oriented Concurrent Programming in ABCL/1", *OOPSLA*† *86 Conference Proceedings*, 1986, pp. 258 – 268.

[YoT86] Yokote, Y. and Tokoro, M., "The Design and Implementation of Concurrent Smalltalk", *OOPSLA*† *86 Conference Proceedings*, 1986, pp. 331 – 340.

[PCMM85] Perrott, R.H., Crookes, D., Milligan, P., and Purdy, W.R.M, "A Compiler for an Array and Vector Processing Language", *IEEE Transactions on Software engineering*, Vol. SE-11, No. 5, May 1985, pp. 471 – 478.

[RaB89] Raghavendra, Srinivas and Budd, Timothy A., "A Multiprocessor Implementation of Little Smalltalk", Technical Report CSTR-89-60-15, Dept. of Computer Science, Oregon State University, 1989

[Ree84] Reeves, Anthony P., "Parallel Pascal: An Extended Pascal for Parallel Computers", *Journal of Parallel and Distributed Computing 1*, 1984, pp. 64 – 80.

[Vee86] Veen, Arthur H., "Dataflow Machine Architecure", *ACM Computing Surveys*, Vol. 18, No. 4, December 1986, pp. 365 – 395.

[WiW89] Wirfs-Brock, Rebecca and Wilkerson, Brian, "Object-Oriented Design: A Responsibility-Driven Approach", *OOPSLA*† *89 Conference Proceedings*, October 1 – 6, New Orleans, Louisiana, pp. 71 – 75.

[Wol88] Wolfe, Micheal, "Vector Optimization vs. Vectorization", *Journal of Parallel and Distrubuted Computing 5*, Oct 1988, pp. 551 – 567.

† OOPSLA : Object-Oriented Programming: Systems, Languages and Applications