A COMPARATIVE STUDY OF FIVE PARALLEL PROGRAMMING LANGUAGES

Henri E. Bal

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam
bal@cs.vu.nl

ABSTRACT

Many different paradigms for parallel programming exist, nearly each of which is employed in dozens of languages. Several researchers have tried to compare these languages and paradigms by examining the expressivity and flexibility of their constructs. Few attempts have been made, however, at *practical* studies based on actual programming experience with multiple languages. Such a study is the topic of this paper.

We will look at five parallel languages, all based on different paradigms. The languages are: SR (based on message passing), Emerald (concurrent objects), Parlog (parallel Horn clause logic), Linda (Tuple Space), and Orca (logically shared data). We have implemented the same parallel programs in each language, using real parallel machines. The paper reports on our experiences in implementing three frequently occurring communication patterns: message passing through a mailbox, one-to-many communication, and access to replicated shared data.

1. INTRODUCTION

During the previous decade, a staggering number of languages for programming parallel and distributed systems has emerged [And83, Bal89a]. These languages are based on widely different programming paradigms, such as message passing, concurrent objects, logic, and functional programming. Both within each paradigm and between paradigms, heated discussions are held about which approach is best [Car89b, Kah89, Sha89].

The intent of this paper is to cast new light on these discussions, using a practical approach. We have implemented a number of parallel applications in each of several parallel languages. Based on this experience, we will draw some conclusions about the relative advantages and disadvantages of each language. So, unlike most of the discussions in the literature, this paper is based on actual programming experience in several parallel languages on real parallel systems.

The languages studied in this paper obviously do not cover the whole spectrum of design choices. Still, they represent a significant subset of what we feel are the most important paradigms for parallel programming. We discuss only a single language for each paradigm, although other languages may exist within each paradigm that are significantly different.

The languages that have been selected for this study are: SR, Emerald, Parlog, Linda, and Orca (see Table 1). SR represents message passing languages. It provides a range of message sending and receiving constructs, rather than a single model. Emerald is an object-based language. Parlog is a concurrent logic language. Linda is a set of language primitives based on the Tuple Space model. Orca is representative of the Distributed Shared Memory model.

We focus on languages for parallel applications, where the aim is to achieve a speedup on a single application. These applications can be run on either *multiprocessors* with shared memory or *distributed* systems without

This research was supported in part by the Netherlands Organization for Scientific Research (N.W.O.).
This paper was first published in the Proceedings of the EurOpen Spring 1991 Conference on Open Distributed Systems in Perspective, Tromso, 20-24 May 1991.
A preliminary version of the paper appeared in the Proceedings of the PRISMA Workshop on Parallel Database Systems, Noordwijk, The Netherlands, September 1990.

Language	Paradigm	Origin
SR	Message passing	University of Arizona
Emerald	Concurrent object-based language	University of Washington
Parlog	Concurrent logic language	Imperial College
Linda	Tuple space	Yale University
Orca	Distributed shared memory	Vrije Universiteit

Table 1: *Overview of the languages discussed in the paper*

shared memory. We have selected only languages that are suitable for *both* architectures. So, we do not discuss shared-variable or monitor-based languages, since their usage is restricted to shared-memory multiprocessors. Functional languages are not discussed either. Most functional languages are intended for different parallel architectures, (e.g., dataflow or graph reduction machines) and often try to hide parallelism from the programmer. This makes an objective comparison with the other languages hard. We also do not deal with distributed languages based on atomic transactions (e.g., Argus [Lis88]), since these are primarily intended for fault-tolerant applications. The issue of fault-tolerant parallel programming is discussed in a separate paper [Bal90d].

The outline of the rest of this paper is as follows. In Section 2, we will briefly describe the applications we have used, focusing on their communication patterns. Next, in Sections 3 to 7, we will discuss each of the five languages, one language per section. Each section has the following structure:

- Background information on the language. (All languages have been described in a recent survey paper [Bal89a], so we will be very brief here.)
- A description of our programming experience. We will comment on the ease of learning the language and on the effort needed to implement the communication patterns discussed in Section 2.
- Comments on the language implementation and its performance. Unfortunately, there is no single platform on which all the languages run, so we had to use many different platforms. The systems we used differ in the number of processors, processor type and speed, as well as in the way processors are interconnected. A fair comparison between the languages is therefore not possible, but the measurements do give some rough indication of the relative speedups that can be obtained.
- Conclusions on the language.

Finally, in Section 8, we will compare the approaches used for the different languages.

2. THE APPLICATIONS AND THEIR COMMUNICATION PATTERNS

There are many ways to compare parallel languages. One way is a theoretical study of the expressiveness of their primitives. This works well for languages using the same paradigm (e.g., message passing), but is more problematic for comparison between different paradigms. Comparing, say, remote procedure calls and shared logical variables is not a trivial task.

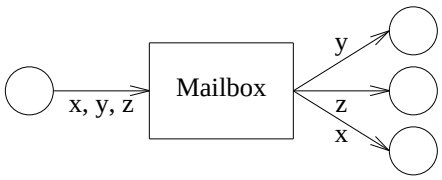
The approach taken in this paper is to implement a set of small, yet realistic, problems in each language, and compare the resulting programs. The example problems we have used include matrix multiplication, the All Pairs Shortest Paths problem, the Traveling Salesman Problem, alpha-beta search, and successive overrelaxation.

The applications and the algorithms used for them are described in detail in [Bal90e]. For this paper, we will restrict ourselves to only two applications: the All Pairs Shortest Paths problem and the Traveling Salesman Problem. These applications will be described below. We will focus on the communication aspects of the applications, since, from a parallel programming point of view, these are most interesting.

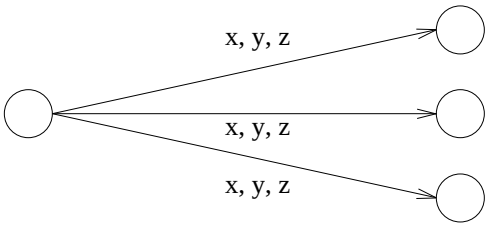
The Traveling Salesman Problem (TSP)

The Traveling Salesman Problem computes the shortest route for a salesman among a given set of cities. The program uses a simple branch-and-bound algorithm and is based on replicated workers style parallelism [And89, Car86a]. The TSP program uses two interesting communication patterns: mailboxes and replicated shared data.

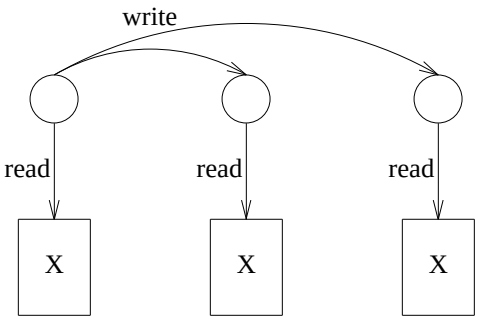
A *mailbox* (see Figure 1a) is a communication port with **send** (nonblocking) and **receive** operations [Bal89a]. Mailboxes can be contrasted with direct message passing, in which the sender always specifies the destination process (receiver) of the message. With mailboxes, any process that can access the mailbox can receive a message sent to it. So, each message sent to a mailbox is handled by one process, but it is not determined in advance which process will accept the message.



(a) *Message passing through a mailbox (used by TSP)*



(b) *One-to-many communication (used by ASP)*



(c) *Communication through replicated shared data (used by TSP)*

Figure 1: *Communication patterns used by the two applications discussed in the paper*

The TSP program uses a mailbox for distributing work. A process that has computed a new job (to be executed in parallel) sends it to a mailbox, where it will eventually be picked up by an idle worker process. Since it is not known in advance which worker process will accept the job, mailbox communication is required here, rather than direct message passing.

The second communication pattern used in the TSP program is *replicated shared data* (see Figure 1c). The branch-and-bound algorithm requires a global variable containing the length of the current best solution. This variable is used for pruning partial solutions whose initial paths are already longer than the current best full route.

In a distributed system, this global variable cannot be put in shared memory, since such systems lack shared memory. One solution is to store the variable on one processor and let other processors access it through remote operations. For TSP (and many other applications), however, a much more efficient solution is possible. The bound is usually changed (improved) only a few times, but may be used millions of times by each processor, so its read/write ratio is very high. Therefore, the variable can be implemented efficiently by *replicating* it in the local memories of the processors. Each processor can directly read the variable. Physical communication only occurs when the variable is written, which happens infrequently.

The All Pairs Shortest Paths Problem (ASP)

The second application is the All Pairs Shortest Paths problem, which computes the lengths of the shortest paths between each pair of nodes in a given graph. ASP uses a parallel iterative algorithm. Each processor is assigned a fixed portion of the rows of the distances matrix. At the beginning of each iteration, one process sends a *pivot row* of the matrix to all the other processes. Each process then uses this pivot row to update its portion of the matrix.

The most important communication pattern of the ASP program thus is *one-to-many* communication (see Figure 1b). This pattern transmits data from one process to many others, *all* of which use these data. (In contrast, a

message sent to a mailbox is used by only *one* process.)

Of course, this pattern can be simulated through multiple point-to-point messages, but frequently much better solutions are possible. Many networks have a *multicast* or *broadcast* capability, which can be used to speed up one-to-many communication significantly. So, there are two issues involved here: how one-to-many communication is *expressed* in a given language and how it is actually *implemented*. For ASP, it is very important that the implementation uses a real multicast. Otherwise, the communication costs may easily become a dominating factor.

3. SYNCHRONIZING RESOURCES (SR)

SR [And86, And88] is a language for writing distributed programs, developed by Greg Andrews, Ron Olsson, and their colleagues at the University of Arizona and the University of California at Davis. The language supports a wide variety of (reliable) message passing constructs, including shared variables (for processes on the same node), asynchronous message passing, rendezvous, remote procedure call, and multicast.

Programming experience

Given its ambitious goal of supporting many communication models, it is not surprising that SR is a fairly large language. Yet, we found it reasonably easy to learn. With regard to the sequential parts, the syntax, type system, and module constructs are different from most other languages. Nevertheless, these were fairly easy to learn, although the type system is far from perfect [Bal90a].

SR tries to reduce the number of concepts for distributed and parallel programming by using an *orthogonal* design. There are two ways for sending messages (blocking and nonblocking) and two ways for accepting messages (explicit and implicit). These can be combined in all four ways, yielding four different communication mechanisms. We agree with the designers that this orthogonality principle simplifies SR's design. Unfortunately, there also are some less elegant design features. The concurrent-send (**co**) command, for example, is a rather ad-hoc extension of the basic model, with specialized syntax rules.

Our programming experience indicates that, even within the restricted domain of parallel programming, nearly all facilities provided by SR are useful. We found uses for synchronous and asynchronous message invocation, explicit, implicit, conditional, and ordered message receipt, and multicast [Bal90a]. Below we will report on our experiences in implementing the three communication patterns of Figure 1 in SR.*

Mailbox communication. Despite its large number of features, SR does not directly support message passing through a mailbox. The receiver of a message is fully determined when the message is *sent*. With a mailbox, the destination process is not determined until the message is *accepted* (serviced).

In contrast with the sender of a message, the receiver need *not* specify the other party, so in this sense message passing in SR is asymmetric. This observation also implies a solution to the mailbox problem. We can simply add an intermediate *buffer* process between the sender and receivers, as shown in Figure 2. The sender sends its message to this buffer process, so the (initial) destination is fixed. The receivers ask the buffer process for a message, whenever they need one.

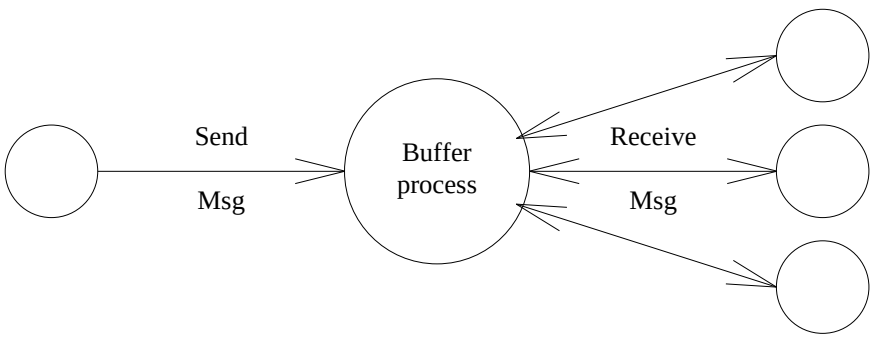


Figure 2: *Simulating message passing through a mailbox in SR*

The buffer process accepts *SendMsg* and *ReceiveMsg* requests one at a time; if the buffer is empty, only *SendMsg* will be accepted. With this scheme, the destination of each message is fixed: it is sent to the buffer

* The language used for this paper is referred to as SR Version 1.1. The SR designers are currently working on Version 2, in which many of the problems described here will be solved.

process. In this way, the asymmetry of message passing is worked around, at the overhead of implementing an extra process.

In applications where only one process is sending messages, a simpler solution can be used. When the sender wants to send a message, it blocks until a receiver asks for a message. In this case, the receiver can directly fetch a message from the sender, thus eliminating the need for a buffer process.

One-to-many communication. The second communication pattern, one-to-many communication, is supported in SR through a special language construct:

```
co (i := 1 to P)
    send receiver[i].SendMsg(msg)
oc
```

The **co** statement sends a message concurrently to several processes, as specified in the array *receiver*. This approach to multicasting has an important disadvantage, however. If two SR processes concurrently multicast two messages, these messages need not arrive in the same order everywhere. In other words, multicast in SR is not *indivisible*. Applications for which the ordering matters must do it themselves.

Shared data. The third communication pattern, replicated shared data, is not supported by SR. Only processes running on the same node can share variables, other processes must communicate through message passing.

Of course, shared data can be simulated by storing them on one server process and having other processes send messages to this server. As stated before, however, we assume that the read/write ratio of the shared data is very high, in which case it is far more efficient to *replicate* the shared data in the local memories. Each processor keeps its own local copy, which is used for *reading*. Whenever the variable is *written*, all these copies are updated.

Concurrent updates of the shared data will have to be synchronized. For example, if two processes *P* and *Q* simultaneously write the shared data, all copies should be updated in a consistent way. It should never be the case that part of the copies are set to *P*'s value while another part is set to *Q*'s value. With message passing this requirement is difficult to realize, because messages are not globally ordered. In other words, the update messages sent by *P* and *Q* may arrive in different orders at different receivers.

The solution we have taken is to send update messages through a central manager process. This process orders the update messages and forwards them in a consistent order to all other processes. These update messages are accepted implicitly by each receiver, which means that the run time system will automatically create a new process for servicing such a message. This is important, since it is not known in advance when the update messages may arrive.

Implementation and performance

SR has been implemented on a range of multiprocessors (Encore, Sequent Balance, Sequent Symmetry) and distributed systems (homogeneous networks of VAXes, Sun-3s, Sun-4s, and others). The compiler and run time system are available from the University of Arizona.

We have done some initial performance measurements on a Sequent Symmetry with 6 CPUs. Although this machine has a shared memory, SR uses it only for implementing message passing, so the machine is not really used as a multiprocessor.

For the All Pairs Shortest Paths problem, we have measured a speedup of 4.08 (on 6 CPUs). The reason why this speedup is less than linear is the fact that the **co** statement currently is not implemented as a true (physical) multicast. The message is copied once for every receiver. The communication overhead of the ASP program therefore is high, which prevents a linear speedup.

For the Traveling Salesman Problem, we measured a maximum speedup of 5.87. The latter program uses the simplified solution for mailbox communication (i.e., the manager generates only one job at a time and blocks until this job has been accepted).

Conclusions on SR

Since SR provides so many communication primitives, it is a flexible language. SR is also more expressive than most other message passing languages. It can be argued, however, that message passing is a low level of abstraction. As we will see, for several applications other mechanisms than message passing are simpler to use. These higher-level mechanisms are frequently more expressive yet less flexible. In conclusion, SR is reasonably suited for virtually all applications. It is seldom spectacularly good or bad for any application.

4. EMERALD

Emerald [Bla87, Jul88] is an object-based language, designed at the University of Washington by Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. An object in Emerald encapsulates both static data and an active process. Objects communicate by invoking each other's operations. There can be multiple active invocations within one object, which synchronize through a *monitor*. The remote invocation mechanism is *location transparent*.

Central to Emerald's design is the concept of *object mobility*. An object may migrate from one processor to another, as initiated either by the programmer or the system. Emerald uses a novel parameter mode, *call-by-move*. This mode has similar semantics as call-by-reference, but additionally moves the object parameter to the node of the invoked object.

Programming experience

Emerald is reasonably easy to learn. Since it is an object-based language, it treats all entities as objects. Unlike object-*oriented* languages, it does not support inheritance. Nonwithstanding its object-based nature, Emerald contains many constructs also found in procedural languages (e.g., nested scopes, functions, expressions, assignment and control statements). The type system is one of the more important contributions of the language. Although it is not easy to get used to, it is flexible and features static type checking and polymorphism.

Below, we will discuss how mailboxes, one-to-many communication, and shared data can be implemented in Emerald.

Mailbox communication. A message (or operation) in Emerald is always sent to a specific object, so mailbox-style communication is not provided. It is possible, however, to construct a mailbox *object*, which can be accessed by the senders and receivers. Such an object has the following user-defined polymorphic type:

```
type Mailbox
    operation AddMsg[eType]             % Add a message to the mailbox
    operation GetMsg -> [job: eType]    % Fetch a message from the mailbox
end Mailbox
```

The object type is implemented using a queue of messages. To synchronize access to the queue, it is encapsulated in a monitor. The *AddMsg* and *GetMsg* operations are thus executed in a mutually exclusive way. Also, the *GetMsg* operation will block on a condition variable if the queue is empty; this condition variable will be signalled by an invocation of *AddMsg*.

This implementation of mailboxes is roughly similar to the SR version, except that a passive object rather than an active process is used for storing the message queue. Also, the synchronization of the queue operations is entirely different. In the SR version, the buffer process synchronizes the operations by accepting them one at a time and by delaying requests for messages when the buffer is empty. The Emerald version uses a monitor and a condition variable for synchronizing the operations.

One-to-many communication. Emerald does not support any form of one-to-many communication. To send data to multiple objects, a sequential **for** loop has to be used. A subtle problem arises here that does not occur in the other languages. Emerald provides a uniform parameter mechanism: all objects are passed by reference, no matter where the sender and receiver are located. With multicasting, however, each receiver should be given a *copy* of the data, not a remote reference to it. What is needed here is call-by-value semantics, which is not supported in Emerald.

Thus, the sender must copy the data explicitly and pass this copy as call-by-move parameter. A distinct copy must be made for every receiver. So a multicast is simulated as follows in Emerald:

for *all receivers r* **do**
  *r.send[***move** *copy[msg]]*

Here, *copy* is a user-defined procedure that copies a message.

The Emerald implementation of one-to-many communication is fairly complex. In addition, the solution is far from efficient. Not only does it refrain from using physical multicast, but it also forces the sender to copy the message once for every receiver, which may become a sequential bottleneck.

Shared data. Although Emerald supports a shared namespace for objects, this is not sufficient for implementing replicated shared data. If a shared variable were stored in a single object, nearly all accesses to the variable would require physical communication, including read-only operations. What is needed is a replicated object, which is not provided in Emerald.

The programmer therefore has to replicate data explicitly. A copy of the shared data is kept by each process

needing the data. To update these copies, a similar scheme as for SR is used, based on implicitly received messages. The main difference with the SR solution is the usage of a monitor for synchronizing access to the local copy of the shared variable.

Implementation and performance

A prototype implementation of Emerald exists on networks of VAXes or Sun-3 workstations, connected by an Ethernet. The Emerald system is not yet available to other users. We have not been able to do any meaningful performance measurements on the prototype system.

Conclusions on Emerald

Support for parallel and distributed programming in Emerald is best understood using two levels of abstraction. At the highest level, we have concurrent objects that invoke each other's operations in a synchronous (blocking) way, certainly a nice and simple abstraction. To see what is really going on, we need to look at how invocations are implemented and synchronized. Here, we are at the level of *monitors*. Monitors are well understood, but are harder to program than most other mechanisms discussed in this paper. This clearly shows of in the implementation code: most of our Emerald programs are significantly longer than their counterparts in the other languages.

For parallel programming, Emerald is less flexible than SR. It provides only one form of interprocess communication: synchronous remote procedure calls that are accepted implicitly. The parameter mechanism is consistent (call-by-reference is used throughout), but copying parameters is a problem. In principle, call-by-value parameters could have been allowed for passive objects (not containing a process). This extension would have made the parameter mechanism less uniform, however, and would have created a distinction between active and passive objects.

Emerald probably is more suitable for distributed applications (e.g., electronic mail, name servers) than for parallel applications. For such distributed applications, features like object migration and location independent invocations are more beneficial and the need for copying objects (e.g., electronic mailboxes) will be less.

5. PARLOG

We have chosen Parlog [Cla88, Cla86, Con89, Gre87] as representative for the large class of concurrent logic languages. Parlog has been developed at Imperial College, London, by Keith Clark, Steve Gregory, and their colleagues.

The language is based on AND/OR parallelism and committed-choice nondeterminism. The user can specify the order (parallel or sequential) in which clauses are to be evaluated. For this purpose, sequential and parallel conjunction and disjunction operators can be used.

Programming experience

The time needed for learning Parlog depends on ones background education in concurrent logic programming. The language itself is quite simple. In addition, there are certain programming idioms one should master, such as streams and objects built with shared logical variables.

Mailbox communication. As in most concurrent logic languages, processes in Parlog can communicate through message streams. Such streams can easily be built out of shared logical variables. Streams, however, have one disadvantage: the receiving end can *scan* over the stream, but it cannot *remove* items from it [Car89b]. Thus, mailbox-type communication cannot be expressed easily with streams.

Instead, we can use similar solutions as for SR, which means either adding a buffer process between the sender and receivers (see Figure 2), or blocking the sender of the message. For our TSP program [Bal90b], we have chosen the latter option. There is only a single sender, which blocks when it wants to send a message. The sender takes a stream of incomplete messages of the form *getmsg(Msg)* as input. These messages are generated by the receivers. After receiving such a message, the sender instantiates the logical variable *Msg* to the next message it wants to send.

One-to-many communication. One-to-many communication is easy to express using shared logical variables. All that is needed is a stream of messages shared among the sender and the receivers. All receivers can scan this stream, thus receiving all the messages.

It depends on the language implementation whether physical multicast is used for this type of one-to-many communication. For example, multicast is used to some extent in the hypercube implementation of Flat Concurrent Prolog [Tay87]. The Parlog system we have used uses shared memory, which takes away the need for physical

multicast.

Our Parlog ASP program uses an even simpler approach to one-to-many communication. Rather than creating a fixed number of long-living processes, it creates a new set of parallel processes for each iteration of the algorithm. The pivot row for the next iteration is passed as parameter to each of these processes. In other words, the program does not send a message to existing processes, but it creates new processes and passes the message as a parameter. This approach only works well because the Parlog system efficiently supports fine-grained parallelism. With the other languages discussed in this paper, the overhead of creating new processes for each iteration would be far too high.

Shared data. Parlog supports shared logical variables, but these variables can be assigned only once. Implementing mutable shared variables in Parlog is much more complicated. We represent such a variable as a stream of values, the last one of which is the current value of the variable. The predicate *current_value* scans the stream until the tail is an unbound variable, and returns the current last element of the stream as output value:

```
mode current_value(Stream?, Value^).  % Stream is input, Value is output

current_value([V|Vs], Value) <- var(Vs): Value = V;        % tail is unbound
current_value([_|Vs], Value) <- current_value(Vs, Value).  % try next element
```

To update the variable, a new value is appended to the end of the stream. A process using the variable must periodically check for new values, by scanning the stream until the end. (This technique is also used by Huntbach [Hun89]).

An important issue is how often to check the stream. Since scanning streams is expensive, it cannot be done too often. On the other hand, if it is done infrequently, the process will usually have an old value of the shared variable. For branch-and-bound applications like TSP, this means pruning will become less efficient, so more nodes will be searched (the so-called *search overhead*).

This solution is somewhat similar to the SR and Emerald implementations described above. The stream representing the shared variable can be regarded as a stream of *update* messages. An important difference is the way these messages are accepted. In SR and Emerald, a new process is created when a message arrives, which will service the message immediately (i.e., the message is received implicitly). Parlog does not have implicit message receipt, so the receiver must explicitly look for new messages. Since it is not know in advance when update messages may arrive, there is a problem in deciding *when* to look for them.

Implementation and performance

An interpreter for Parlog has been implemented on several shared-memory multiprocessors (Sequent Balance and Symmetry, Butterfly). A commercially available subset of Parlog, called Strand, has also been implemented on distributed systems (hypercubes, networks). The Parlog system is available from Imperial College.

We have used a 6-CPU Sequent Balance for running some initial performance measurements. This implementation of Parlog relies on the presence of shared memory. Also, the implementation is based on an interpreter and runs on slow processors, so its absolute performance currently is one to two orders of magnitude less than that of the other languages described in this paper. These two issues taken together result in a relative communication overhead that is far less than what would be expected in a production-quality, distributed implementation.

We have measured a speedup of 5.33 for ASP and 4.98 for TSP, using 6 CPUs. The speedup for ASP is fairly high, due to the low communication overhead. For TSP, the speedup is not optimal, because the global bound is not kept up-to-date everywhere. The TSP program therefore suffers from a search overhead.

Conclusions on Parlog

The shared logical variable is at a higher level of abstraction than message passing. For some applications, it is spectacularly expressive. Our Parlog program for ASP, for example, is just as simple as the original *sequential* algorithm. The synchronization of the parallel tasks is done implicitly, using suspension on unbound logical variables. On the negative side, it is not clear whether the program will run efficiently on a realistic large-scale parallel system.

For other applications, shared logical variables are less suitable, but one can then fall back on message passing through streams. This form of message passing has some drawbacks, however, as discussed in [Car89b].

6. LINDA

Linda is a set of language primitives developed by David Gelernter and colleagues at Yale University [Ahu86, Car89a, Car89b]. Linda is based on the Tuple Space model of communication. The Tuple Space is a global memory consisting of tuples (records) that are addressed associatively. Three atomic operations are defined on Tuple Space: **out** adds a tuple to TS; **read** reads a tuple contained in TS; **in** reads a tuple and also deletes it from TS, in one atomic action.

Programming experience

Of all five languages discussed in this paper, Linda undoubtedly is the simplest one to learn. It adds only a few primitives to an existing base language. Below, we will discuss how these primitives can be used to implement the three communication patterns.

Mailbox communication. The simulation of a mailbox in Linda is simple. A mailbox is represented as a distributed data structure [Car86a] in Tuple Space. To send a message to the mailbox, a new tuple containing the message is added to this data structure. To receive a message, a tuple is retrieved from Tuple Space and its contents are read.

To preserve the ordering of the messages, a sequence number field is added to each message tuple. The tuples are generated and retrieved in the same order. The next sequence number to generate and the sequence number of the next message to accept are also stored in tuples. They are initialized to zero, by the statements:

```
out("head", 0);   # initialize tuple containing index of head of queue
out("tail", 0);   # initialize tuple containing index of tail of queue
```

To send a message *msg* to a mailbox, the following code is executed:

```
in("tail", ? &tail);    # obtain next sequence number
out("tail", tail + 1);  # put back next sequence number
out("MB", msg, tail);   # put message with sequence number in TS
```

The **in** operation blocks until a matching tuple is found. Next, it assigns the formal parameters of the **in** (denoted by a "?") the corresponding values of the tuple. Finally, it deletes the tuple from Tuple Space. All of this is done atomically.

Receiving a message from a mailbox is implemented through the following code:

```
in("head", ? &head);    # first obtain sequence number
out("head", head+1);    # put sequence-number tuple back in TS
in("MB", ? &msg, head); # now fetch message with right sequence number
```

The tuples can be thought of as forming a distributed *queue* data structure, with pointers (indices) to the head and tail of the queue.

This example clearly illustrates the advantages and disadvantages of Linda. The mailbox implementation is very simple: it requires only a few lines of code. On the other hand, the operations used for accessing the mailbox are fairly low-level. For example, three Tuple Space operations are needed for sending or receiving a single message. It is far from trivial that this code is correct. Also, the implementation must do extensive optimization to make the send/receive operations efficient.

One-to-many communication. In Linda, data can be transferred from one process to all the others by putting the data in Tuple Space, where it can be read by everyone. So, expressing one-to-many communication in Linda is trivial; it just requires a single **out** statement:

```
out(msg);
```

A key question that remains, however, is what *really* happens. For efficiency, it makes considerable difference whether the data are transferred through a real multicast protocol or not.

There are many different implementations of Tuple Space to consider. The S/Net system replicates all tuples everywhere, using the S/Net broadcast capability [Car86b]. The hypercube and Transputer implementations of Linda, on the other hand, hash each tuple onto one specific processor and do not replicate tuples [Bjo89, Zen90]. In this case, the data in the message will not be multicast. Each receiver will have to fetch the data itself, using a **read** statement. The communication overhead will thus be linear to the number of receivers. In conclusion, expressing one-to-many communication is Linda is trivial, but the performance will be hard to predict.

Shared data. In theory, a shared variable can be simulated in Linda by storing it in Tuple Space. This solution makes heavy demands on the implementation of Tuple Space, however. If the variable is read very frequently (as is true in TSP), the overhead of reading it must be very low. So, for efficiency each processor should have a

local copy of the tuple. Not all Tuple Space implementations have this property. The hypercube and Transputer implementations mentioned above, for example, store each tuple on only a single processor. An additional performance problem is the associative addressing of Tuple Space. Part of this overhead can be optimized away [Car87], but it is not clear whether it can be eliminated entirely. So, whether or not the above solution is practical, depends on the implementation.

Implementation and performance

Linda has been implemented on many parallel machines, both with and without shared memory, and has been used for numerous applications [Car89b]. The system is distributed as a commercial product. (The Linda system we have used for our performance measurements is not the most recent one; newer versions of the Linda software may obtain better performance.)

We have used a VME-bus based multiprocessor for some initial performance measurements. For the All Pairs Shortest Paths problem, we have measured a speedup of 7.4 on 8 CPUs. Since the implementation uses shared memory, the distribution of the pivot rows is efficient. Each new pivot row is put in a tuple in shared memory, where it can be read by all processors.

The Traveling Salesman Problem program obtains a speedup of 7.06 on 8 CPUs. The program stores the global bound in a tuple. In our Linda system, using this tuple for every read access is too expensive. Therefore, each processor also keeps a local copy of the variable. These copies are updated occasionally. So, this implementation is similar to the Parlog implementation, except that the bound is stored in a tuple rather than in a stream. Updating the local copies is relatively cheaper in the Linda version, so it can be done more frequently. As a result, the relative search overhead in the Linda program is less than that of the Parlog version.

Conclusions on Linda

Most of the criticism on Linda in the literature is related to efficiency. The associative addressing and global visibility of the Tuple Space have led many people to believe that Linda cannot be implemented efficiently. However, its implementors have made considerable progress during the past few years in optimizing the performance on several machines. The **in** operation, for example, hardly ever scans the entire Tuple Space, but typically uses hashing or something even more efficient. Just as with virtual memory, however, there will probably always remain cases where the easy-to-program approach will not be optimal. So, the performance of Linda programs may sometimes be hard to predict.

An important decision in Linda is to hide the physical distribution of data from the user. In contrast, Emerald gives the programmer control over the placement of data, by supporting user-initiated object migration. The Linda approach is simpler, but it makes heavier demands on the implementation. Again, the transparent approach will sometimes be less efficient, but it remains to be seen how big the differences in performance are for actual programs.

The concept of distributed data structures is probably one of the most important contributions of Linda. However, the way Linda implements distributed data structures—through a fixed number of operations on Tuple Space—is rather low-level, in our view [Kaa89].

7. ORCA

Orca is a language for implementing parallel applications on distributed systems. Orca was designed at the Vrije Universiteit in Amsterdam [Bal90e, Bal89b, Bal90c, Bal88].

The programming model of Orca is based on logically shared data. The language hides the physical distribution of the memory and allows processes to share data even if they run on different nodes. In this way, Orca combines the advantages of distributed systems (good price/performance ratio and scalability) and shared-memory multiprocessors (ease of programming).

The entities shared among processes are data objects, which are variables of user-defined abstract data types. These data objects are replicated in the local memories, so each process can directly read its own copy, without doing any communication. The language run time system atomically updates all copies when an object is modified.

This model is similar to that of Distributed Shared Memory (DSM) systems [Li89]. In Orca, however, the unit of sharing is a logical (user-defined) object rather than a physical (system-defined) page, which has many advantages [Bal90e].

Programming experience

Orca is a new language rather than an extension to an existing sequential language. An important disadvantage of extending a base language is the difficulty of implementing pointers and global variables on systems lacking shared memory. These problems can more easily be avoided if the language is designed from scratch. Orca, for example, supports first-class *graph* variables rather than pointers. Unlike pointers, graphs can freely be moved or copied from one machine to another. Of course, this approach also implies that programmers have to learn a new language. The design of Orca has been kept as simple as possible, so this disadvantage should not be overestimated.

Mailbox communication. A mailbox can be implemented in Orca in a similar way as in Emerald, by using a shared mailbox object. The specification of a generic abstract data type *Mailbox* in Orca is shown below:

```
generic (type T)
object specification GenericMailbox;
    operation AddMsg(Msg: T);
    operation GetMsg(): T;
end generic;
```

The implementation of the mailbox is simpler than the one in Emerald, because operations in Orca are indivisible. In other words, mutual exclusion synchronization is done automatically in Orca, whereas Emerald requires the usage of a monitor construct for this purpose. Also, Orca provides a powerful mechanism for condition synchronization (based on guarded commands), so blocking the receivers when the mailbox is empty is easy to express.

One-to-many communication. Orca's shared data-objects can be used for expressing one-to-many communication. If one process applies a write-operation to an object, all other processes sharing the object can observe the effects. Our ASP program in Orca, for example, uses an object-type *RowCollection*, with the following operations:

```
object specification RowCollection;
    type RowType = array[integer] of integer;

    operation AddRow(iter: integer; R: RowType);
        # Add the row for the given iteration number
    operation AwaitRow(iter: integer): RowType;
        # Wait until the row for the given iteration is available,
        # then return it.
end;
```

The process that wants to send the pivot row applies the operation *AddRow* to the object. The run time system will then update all copies of this object by multicasting the operation [Bal89b]. A process requiring the pivot row invokes the operation *AwaitRow*, which blocks until the requested row has been added to the object and then returns this row. The latter operation is done locally, without needing any communication. So, the Orca solution is efficient, since it uses physical multicasting, if available.

Shared data. Orca has the support for logically shared data as a design goal, so it is no surprise that communication through shared data is easy to express in this language. The shared variable is put in a data object shared among all processes. The run time system automatically replicates the object in the local memories, so processes can directly read the value. Whenever the object is changed, all copies are updated immediately, by broadcasting the new value. Moreover, atomicity of the operations is already guaranteed by the language. This solution is both simple and efficient. The only overhead in reading the value is that of a local operation invocation. When the variable is changed, its new value is broadcast to all processors containing a copy.

Implementation and performance

Orca has been implemented on top of Amoeba [Tan90] as well as on a collection of MC68030s connected through an Ethernet. The latter implementation uses the physical multicast capability of the Ethernet. The Orca implementation is being distributed as part of the Amoeba system.

We have done many performance measurements on these systems, as described in detail elsewhere [Bal90e]. Here, we will present some recent results for the multicast system, using 16 CPUs.

The measured speedup for the All Pairs Shortest Paths problem on 16 CPUs is 15.9. This high speedup is mainly due to the efficient broadcast protocol, which is used for transmitting the pivot rows. For the Traveling Salesman Problem, the speedup on 16 CPUs is 14.44. Since all copies of the global bound are updated immediately, the search overhead is low.

Conclusions on Orca

Orca is *not* an object-based language; it merely provides abstract data types. It supports both active processes and passive data-objects. Since objects in Orca are purely passive, they can be replicated, which is a very important goal in the implementation.

An important difference with Linda is the support for user-defined, high-level operations on shared data [Kaa89]. Linda only provides a fixed number of built-in operations on tuples, but Orca allows programmers to construct their own atomic operations. Unlike Linda, Orca uses direct rather than associative addressing of shared data, and thus avoids any problems with associative addressing.

For some applications, Orca has important advantages over other languages. Programs that need logically shared data are easy to implement in Orca and are efficient. Orca also is one of the few languages that uses physical broadcasting in its implementation. As we have seen, for ASP this is of critical importance. On the other hand, there also are cases where the model is less efficient, for example when plain point-to-point message passing is required.

8. DISCUSSION

In the previous three sections we have looked at how the five languages deal with three example communication patterns. The results of this study are summarized in Table 2. Below, we will compare the approaches taken for the different languages.

	Mailboxes	One-to-many communication	Replicated shared data
SR	Buffer process	Concurrent send	Messages with implicit receive
Emerald	Shared-object message queue	Point-to-point messages	Messages with implicit receive
Parlog	Buffer process	Shared stream (or solution with fine-grained parallelism)	Messages with explicit receive
Linda	Distr. data structure message queue	Shared data	Shared tuple (or m.p. with explicit receive)
Orca	Shared-object message queue	Shared data	Distributed shared memory

Table 2: *Summary of the solutions taken for all five languages to the three communication patterns*

For communication through mailboxes, there are three different solutions. For Linda, we store a mailbox as a distributed data structure in Tuple Space. This solution requires only a few lines of code. For Emerald and Orca, a mailbox is represented as an abstract object, with operations to send and receive messages. This approach requires more code, especially for synchronizing access to the mailbox. On the other hand, the abstract operations on a mailbox object are higher level than the Linda operations on tuples. The third solution, used for SR and Parlog, is to add an extra buffer process between the sender and receivers.

For one-to-many communication, Parlog, Linda, and Orca provide the simplest solutions, all based on shared data. SR has a concurrent-send primitive built in, but it does not make any guarantees about the order in which messages are delivered. Emerald has no provision for one-to-many communication, so it must be simulated with multiple point-to-point messages, which are sent sequentially. An important issue is how one-to-many communication is implemented: as a physical multicast or not. Most language implementations do not use multicast, Orca and Linda being two notable exceptions.

The third communication pattern, replicated shared data, is simple to express in Orca and Linda, since these languages provide logically shared data. For Linda, the performance of the resulting programs is hard to predict, because many different strategies are used for distributing tuples. Orca, on the other hand, always tries to replicate shared objects wherever they are needed. For the other languages, we simulate shared data through message passing. Here, the ability to accept messages implicitly (i.e., by a newly created process) is very important. SR and Emerald both provide this facility. Parlog uses only explicit message receipt, which makes efficient updating of the copies of shared data harder.

ACKNOWLEDGEMENTS

The work on SR and Emerald was done while the author was visiting the University of Arizona, Department of Computer Science, Tucson, AZ. The work on Parlog was done while he was at Imperial College, Department of Computing, London. The author is grateful to both departments for receiving him as an academic visitor. Also, he would like to thank Nick Carriero, Greg Andrews, Dave Bakken, Gregg Townsend, Mike Coffin, Norman Hutchinson, Keith Clark, Jim Crammond, and Andrew Davison for the discussions on their languages. The work on Linda has been done in cooperation with Frans Kaashoek. Erik Baalbergen, Arnold Geels, Frans Kaashoek, and Andy Tanenbaum provided useful comments on an earlier version of the paper.

REFERENCES

Ahu86. S. Ahuja, N. Carriero, and D. Gelernter, ''Linda and Friends,'' *IEEE Computer* **19**(8), pp. 26-34 (Aug. 1986).

And83. G.R. Andrews and F.B. Schneider, ''Concepts and Notations for Concurrent Programming,'' *ACM Computing Surveys* **15**(1), pp. 3-43 (March 1983).

And86. G.R. Andrews and R.A. Olsson, ''The Evolution of the SR Programming Language,'' *Distributed Computing* **1**, pp. 133-149 (July 1986).

And88. G.R. Andrews, R.A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend, ''An Overview of the SR Language and Implementation,'' *ACM Trans. Program. Lang. Syst.* **10**(1), pp. 51-86 (Jan. 1988).

And89. G.R. Andrews, ''Paradigms for Process Interaction in Distributed Programs,'' TR 89-24, University of Arizona, Tucson, AZ (Oct. 1989). (accepted for publication in ACM Computing Surveys)

Bal88. H.E. Bal and A.S. Tanenbaum, ''Distributed Programming with Shared Data,'' *Proc. IEEE CS 1988 Int. Conf. on Computer Languages*, Miami, FL, pp. 82-91 (Oct. 1988).

Bal89a. H.E. Bal, J.G. Steiner, and A.S. Tanenbaum, ''Programming Languages for Distributed Computing Systems,'' *ACM Computing Surveys* **21**(3), pp. 261-322 (Sept. 1989).

Bal89b. H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum, ''A Distributed Implementation of the Shared Data-object Model,'' *USENIX Workshop on Experiences with Building Distributed and Multiprocessor Systems*, Ft. Lauderdale, FL., pp. 1-19 (Oct. 1989).

Bal90a. H.E. Bal, ''An Evaluation of the SR Language Design,'' IR-219, Vrije Universiteit, Amsterdam, The Netherlands (August 1990).

Bal90b. H.E. Bal, ''Heuristic Search in PARLOG using Replicated Worker Style Parallelism,'' IR-229, Vrije Universiteit, Amsterdam, The Netherlands (November 1990).

Bal90c. H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum, ''Experience with Distributed Programming in Orca,'' *Proc. IEEE CS 1990 Int. Conf. on Computer Languages*, New Orleans, LA, pp. 79-89 (March 1990).

Bal90d. H.E. Bal, ''Fault-tolerant Parallel Programming in Argus,'' IR-214, Vrije Universiteit, Amsterdam, The Netherlands (May 1990).

Bal90e. H.E. Bal, *Programming Distributed Systems*, Silicon Press, Summit, NJ (1990).

Bjo89. R. Bjornson, N. Carriero, and D. Gelernter, ''The Implementation and Performance of Hypercube Linda,'' Report RR-690, Yale University, New Haven, CT (March 1989).

Bla87. A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter, ''Distribution and Abstract Types in Emerald,'' *IEEE Trans. Softw. Eng.* **SE-13**(1), pp. 65-76 (Jan. 1987).

Car86a. N. Carriero, D. Gelernter, and J. Leichter, ''Distributed Data Structures in Linda,'' *Proc. 13th ACM Symp. Princ. Progr. Lang.*, St. Petersburg, FL, pp. 236-242 (Jan. 1986).

Car86b. N. Carriero and D. Gelernter, ''The S/Net's Linda Kernel,'' *ACM Trans. Comp. Syst.* **4**(2), pp. 110-129 (May 1986).

Car87. N. Carriero, ''The Implementation of Tuple Space Machines,'' Research Report 567 (Ph.D. dissertation), Yale University, New Haven, CT (Dec. 1987).

Car89a. N. Carriero and D. Gelernter, ''How to Write Parallel Programs: A Guide to the Perplexed,'' *ACM Comp. Surveys* **21**(3), pp. 323-357 (Sept. 1989).

Car89b. N. Carriero and D. Gelernter, ''Linda in Context,'' *Commun. ACM* **32**(4), pp. 444-458 (April 1989).

Cla86. K.L. Clark and S. Gregory, ''PARLOG: Parallel Programming in Logic,'' *ACM Trans. Program. Lang. Syst.* **8**(1), pp. 1-49 (Jan. 1986).

Cla88. K.L. Clark, ‘‘PARLOG and Its Applications,’’ *IEEE Trans. Softw. Eng.* **SE-14**(12), pp. 1792-1804 (Dec. 1988).

Con89. T. Conlon, *Programming in PARLOG*, Addison-Wesley, Wokingham, England (1989).

Gre87. S. Gregory, *Parallel Logic Programming in PARLOG*, Addison-Wesley, Wokingham, England (1987).

Hun89. M. Huntbach, *Combinatorial Search in PARLOG Using Speculative Computation*, Imperial College, London (May 1989).

Jul88. E. Jul, H. Levy, N. Hutchinson, and A. Black, ‘‘Fine-Grained Mobility in the Emerald System,’’ *ACM Trans. Comp. Syst.* **6**(1), pp. 109-133 (Feb. 1988).

Kaa89. M.F. Kaashoek, H.E. Bal, and A.S. Tanenbaum, ‘‘Experience with the Distributed Data Structure Paradigm in Linda,’’ *USENIX Workshop on Experiences with Building Distributed and Multiprocessor Systems*, Ft. Lauderdale, FL., pp. 175-191 (Oct. 1989).

Kah89. K. M. Kahn and M.S. Miller, ‘‘Technical Correspondence on ‘‘Linda in Context’’,’’ *Comm. ACM* **32**(10), pp. 1253-1255 (Oct. 1989).

Li89. K. Li and P. Hudak, ‘‘Memory Coherence in Shared Virtual Memory Systems,’’ *ACM Trans. Computer Systems* **7**(4), pp. 321-359 (Nov. 1989).

Lis88. B. Liskov, ‘‘Distributed Programming in Argus,’’ *Commun. ACM* **31**(3), pp. 300-312 (March 1988).

Sha89. E. Shapiro, ‘‘Technical Correspondence on ‘‘Linda in Context’’,’’ *Comm. ACM* **32**(10), pp. 1244-1249 (Oct. 1989).

Tan90. A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, A.J. Jansen, and G. van Rossum, ‘‘Experiences with the Amoeba Distributed Operating System,’’ *Comm. ACM* **33**(12), pp. 46-63 (Dec. 1990).

Tay87. S. Taylor, S. Safra, and E. Shapiro, ‘‘A Parallel Implementation of Flat Concurrent Prolog,’’ *Int. J. of Parallel Programming* **15**(3), pp. 245-275 (1987).

Zen90. S.E. Zenith, ‘‘Linda Coordination Language; Subsystem Kernel Architecture (on Transputers),’’ RR-794, Yale University, New Haven, CT (May 1990).