TransPive: A Distributed Lisp System

José M. Piquer*
Christian Queinnec†

November 23, 1992

Abstract

This paper exposes an overview of a distributed Lisp system, called TransPive, designed to run on a loosely-coupled multi-processor system.

The main goal of the system is to provide a transparent distributed programming environment, sharing data structures using remote pointers, as a platform to prototype distributed algorithms and applications.

The TransPive message passing primitives create the remote pointers, with local caches for the object's value (to improve performance of read accesses). A protocol to invalidate the caches is invoked when a modifier is applied. A Distributed Garbage Collector is included on TransPive to reclaim distant pointed objects. A simple mechanism to distribute the computation over the system is also provided.

All the mechanisms have been implemented without any operating system support (as virtual memory), and so they are portable to any reliable message-passing distributed environment. In particular, the first version has been implemented on a Transputer network.

The paper is an overview of the remote pointer mechanism, the memory structures and protocols used to share data, the garbage collector and the remote process creation primitives.

1 Introduction

Many parallel Lisp systems have been designed and implemented in the last years, notably MultiLisp [Hals 85] and Mul-T [Kran 89], MultiScheme [Mill 87] and Qlisp [Gabr 84]. However they are designed for shared memory multi-processors.

Some other languages provide support for distributed programming sharing data, as Orca [Bal 88] and some object-oriented systems as Emerald [Blac 87].

There are also many distributed operating systems which provide a simulated shared memory on distributed architectures based on virtual memory paging systems[Li 86, Fitz 86, Rozi 88] permitting the coupling of virtual memory spaces, sharing pages although sometimes restricted to be *read only* or *copy on write*.

TransPive is a distributed Lisp and, in the Lisp tradition, all the distributed memory management system is built in the language itself. It includes mechanisms to share read/write objects, to reclaim unused objects and to distribute processes across the different processors. All the mechanisms are written in Lisp, and can be replaced with new implementations (providing the same semantics) so TransPive itself is very useful to test new distributed memory management algorithms.

TransPive was designed as a model of a distributed Lisp system, general enough to run on any loosely-coupled multiprocessor system with independent memories and without any virtual memory support (our

*Universidad de Chile, Casilla 2777, Santiago, Chile — jpiquer@dcc.uchile.cl.
†LIX (URA 1439), École Polytechnique, 91128 Palaiseau Cedex, France — queinnec@polytechnique.fr.

current implementation runs on a set of Transputer[1] processors), making page-oriented algorithms unusable. The only required operating system support is reliable message-passing and unique identifiers for every site.

TransPive is not a simple Lisp extension with some extra primitives to send messages, it is a complete distributed memory management system, portable to different distributed systems and even to different languages.

Our current implementation of TransPive was added to a Lisp dialect called Le-Lisp [Chai 84] and is running on a Transputer network. However, the TransPive model is general enough to be implemented on other Lisp dialects and computer architectures. We are currently working on a version running on a TCP/IP heterogeneous network.

TransPive is composed by two independent systems: the distributed memory management system (composed of remote pointers and a distributed garbage collector) and the remote process creation system.

The overall system is composed by a group of Lisp systems, running on different processors. Each Lisp runs a small local multi-tasking system called Pive[Serp 84], permitting task creation, communication and synchronization with semaphores, messages, pipes and signals, within the same processor.

TransPive adds some primitives to Pive, permitting two Lisp systems running on different processors to communicate with each other (creating remote pointers), and to create remote processes. One Lisp system acts as the master Lisp, including a toplevel. This master creates the other Lisp processes, and it distributes the first processes to the slave systems.

In this paper we will describe the basic remote pointer implementation, an overview of the read and write protocols to share data, the garbage collector and the process management primitives to permit remote evaluation and parallelism. Finally, some measures for simple parallel examples are presented.

2 Remote Pointers

The TransPive's external message passing primitives are able to send and receive any valid Lisp object or data structure. The semantics of the sending operation is to send the object *by reference*. So, the sender and the receiver physically share the object, the receiver holding a remote pointer to it.

The external message sending primitives receive a type as argument. This type is used to find how to transform the Lisp object into a stream of bytes, and at the receiving site it is used to find the inverse function to transform the stream into a Lisp object. The semantics of object sending are always the same, but the type is used to choose a sending mechanism, for example *lazy copying*, *eager copying*, etc. There is also a support to send user-typed objects, specifying particular methods to apply to given types instead of using the default generic functions.

To implement the sending *by reference*, the object is received at the other end as a 'remote pointer', which means that the receiver is not the *owner* of the object. The receiver can keep a read-only copy of the value but it cannot update it nor distribute it. We will always preserve that a given object has one and only one owner at a given time.

Remote pointers behave exactly as ordinary Lisp pointers, and they can be manipulated transparently by any application program. The different actions are performed upon read accesses, modification and garbage collection by TransPive. A remote pointer is not a new Lisp data type, it is only another kind of pointer. In fact, the type and the size of the remote-pointed object are always kept locally to test invalid accesses.

The external message passing primitives are the only existing mechanism to create remote pointers.

3 Sharing Data Structures

We try to preserve Lisp semantics, so any function receiving a pointer as an argument can receive a remote pointer, and must perform the same actions on the local or remote object.

[1] Transputer is a trademark of INMOS Ltd.

The semantics of the system is to consider that objects are sent *by reference*, without duplication, and so they are shared, designed by local and remote pointers. An operation on a remote object is translated to a remote operation, much like a remote procedure call[Birr 84] or a remote application. This implies an RPC semantics of accesses, meaning:

- Serialization

  Concurrent update operations on the same object are serialized, but the order is random.

- Global Order

  Every site sees the update operations executed on the same order. (The only way to see an update, is to read the object's value.)

This provides the semantics a programmer is used to when objects only have one physical copy (as in a shared memory model). We outline here the mechanisms implemented to permit this, and they are studied in more detail in [Piqu 90] and in [Piqu 91c].

3.1 RPC semantics and asynchronism

The main point here is that we can keep the RPC semantics, even using replication to improve the availability of data. Read accesses use the local replica (that represents the object, but is not the object itself), and modifiers must use a coherency protocol to preserve the RPC semantics. The direct solution is to block the modifying process until every replica has been updated or invalidated. However, this is very costly, and we will prefer an asynchronous solution.

As is noted by Lamport [Lamp 78], in a distributed system there is no *total ordering* of events, but a *partial ordering* induced by message passing between sites. This implies that, without a common clock, two independent events on different sites are not ordered, and nobody can tell which one happened before the other. Independent events are also called concurrent events, in the sense that they could have happened at the same time. An ideal external observer (with a clock) could tell which event happened before (in one specific execution), however, for the system itself this order is not observable, and if we could change the order of both events, the results of the execution would be exactly the same.

Based on this definition, to keep the RPC semantics means that the observable order of the events must be the same (Global Order) and that updates on the same object must be serialized (Serialization). This is important for modifications: if a process is modifying an object and in parallel (and independently) another process is reading it, the exact result of the read operation is not defined.

A modifier applied on a remote object can return before every copy is updated, provided that any read access on the remote object *explicitly dependent* of that modification, returns the new value (or a newer one). This can be implemented using causally ordered messages[Rayn 89, Schi 89].

3.2 Remote Objects

To implement remote-pointed objects, every object created on any Lisp in TransPive has a unique global identifier. When the object is sent to a distant processor, the message contains the necessary information to create remote pointers at the other side (the object identifier and the Lisp identifier where to ask for it). Each time an object is received, its object identifier is looked upon a local hash table which contains every known external object, permitting the physical sharing of objects.

A remote pointer is in fact a normal pointer to a local *cache* for the value of the distant object. An extra field in every object contains the descriptor needed to keep the remote information. The general structure can be seen in Figure 1.

If a Lisp operator is invoked with a remote pointer as argument, and the local cache is empty, it generates an *object fault* which is similar to a page fault in a demand paging system. TransPive calls a different exception

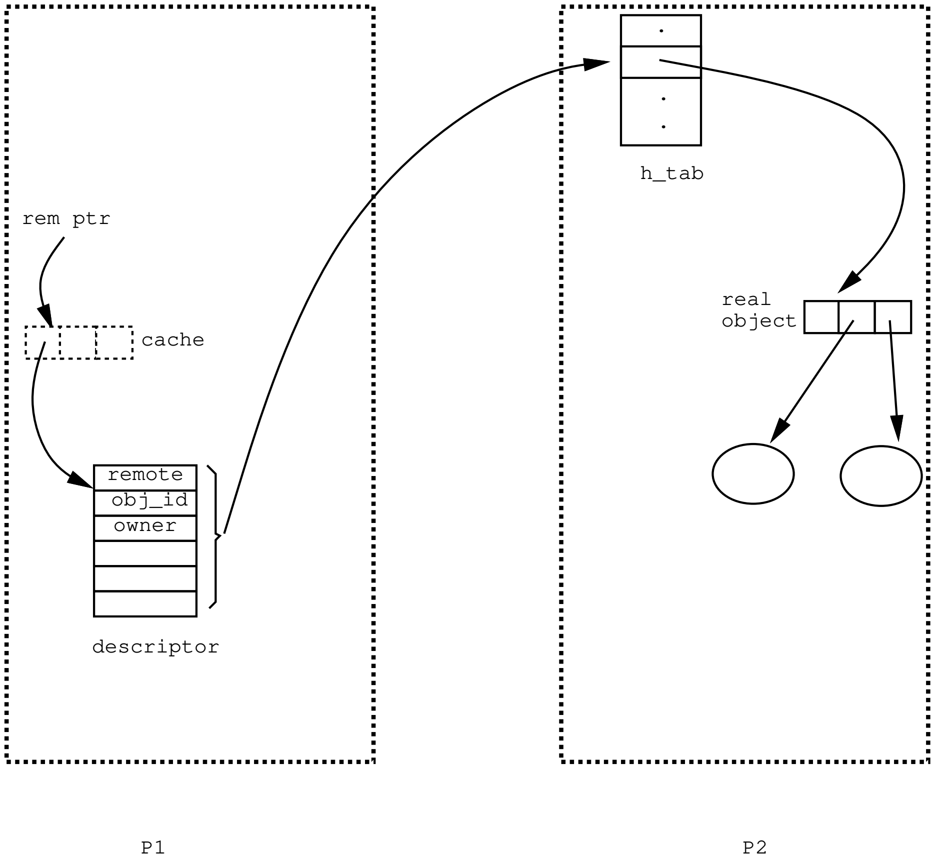

Figure 1: Remote Pointers

routine as the object is accessed for reading or writing. The exception routine for reading should be able to get the current value of the remote-pointed object and to put it in the cache. This is done by changing the descriptor, which keeps a state telling if the cache value is valid or not.[2] The exception routine for writing is more complicated as it should guarantee that the modification is performed on every copy of the value, preserving the imposed order. The exception routines can be implemented in many ways, providing the same semantics. We describe here the simplest implementation.

3.3 The read protocol

Each time that a remote pointer is accessed for reading, the state is tested to see if the cache has a valid copy or not. If there is a valid copy, the access continue as usual. If not, a special read handler is called. The read routine uses the information stored in the remote pointer to contact the memory server of the distant Lisp (the object's *owner*) asking it to send back the value of the object. This value is copied locally into the cache, and the state of the remote pointer is changed to mark the copy as valid. To avoid an excess of traffic and exceptions, the memory server uses a lazy send, which follows the pointers sending remote pointers and values until a certain level. Then it sends only the remote pointers to stop the copying. Once this is done, the routine returns, and the access to the object can be resumed.

[2]In fact, the remote/local information is kept on a bit of the pointer to the descriptor to permit faster testing.

3.4 The write protocol

In a Lisp system, we expect the programming model of a mostly functional language, so physical modification of data structures should be rare. However, we do not want to limit access to distributed structures on a read-only basis. As many sites can hold copies of the values, an object's update must be propagated to all sites holding a replica. This is known as the memory coherency problem.

There are many protocols described in the literature to solve the memory coherence problem, but they are usually very complex [Li 86, Katz 85] or based on system provided *multicast* primitives [Bal 88], with strong order properties. The synchronous coherency algorithms block the write operator until every copy of the value has been updated. The asynchronous coherency algorithms resume the write operator without waiting, but a read operation, *following* (in the Lamport sense) the write operation, must return, at least, the new value.

For the memory coherence problem we chose to implement an invalidation protocol, transforming the state of every remote copy of the value of a modified object into invalid. We use an invalidation rather than an update protocol (writing the new value into every distant copy) based on its simplicity, but also on a locality effect. We already know that modifying shared data structures will be slow, and so we will try to avoid it. However, if necessary, one site may want to administrate an object. To do so, it synchronizes with the rest, performs some modifications and then resumes the others. In the update protocol every modification must be done everywhere. With the invalidation protocol only the first modification produces communication, and after this, the manager can continue to do modifications locally without any overhead (until some remote site accesses the value).

We have implemented an asynchronous coherency protocol using a *global broadcast* primitive[Piqu 91c] and imposing a delivery order between messages and broadcasts. The broadcast is asynchronous (it does not wait for acknowledgements of reception) but it guarantees that:

1. A broadcast causally preceding a message is always received before the message.
2. A message sent before a broadcast (at the same originator site) is always received before the broadcast.

The invalidation protocol uses the broadcast to invalidate every copy of the object, without having to keep track of every site having copies. This system is more expensive than using a multicast only to the sites concerned by a given object, but permits a simpler solution to the coherence problem.

The outline of the write protocol is simple. Every object has an *owner*. The *owner* is the only site with permission to send copies of the value of this object to the outside. When the owner wants to modify the object, it broadcasts an *invalidation* message and then resumes the modification.

When another site wants to modify the object, it must first become its owner (asking the permission to the current owner) and then it can resume the modification (without broadcasting an *invalidation* message).

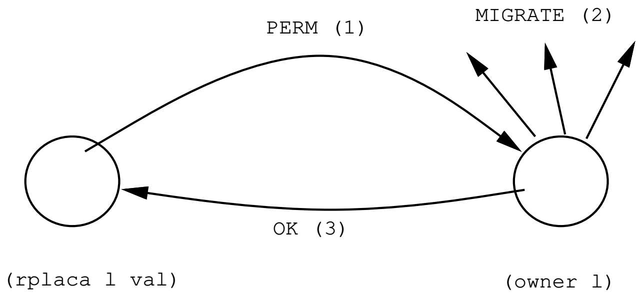

Figure 2: The Migration Protocol

When the owner receives a message requesting the ownership, it performs a broadcast with a *migration* message, including the object and its new owner (the requesting site), and then replies the original message.

Upon reception of a *migration* message, every site invalidates its local copy (if any) and changes its owner to the new site. An example of the migration protocol can be seen in Figure 2, the numbers in parenthesis showing the order of the events. The idea is that the owner can continue to modify the same object without having to broadcast again.

This system serializes the write operations (shown as *owner* changes), and every site sees the modifications in the same order (due to the causality of broadcasts).

4 The Garbage Collector

We have implemented a simple garbage collector (called Indirect Reference Counting[Piqu 91b]) to manage remote pointers only, keeping the local garbage collectors (almost) unmodified. To avoid the problem of the *increment* messages on reference count algorithms [Lerm 86] instead of using a Weighted Reference Count [Wats 87, Beva 87] algorithm, we simply construct a parallel structure, using two fields per remote pointer. The structure is the diffusion tree of the pointer, and the root is always the current owner of the object. When a site sends a remote pointer to another, the descriptor at the receiving site becomes a son of the descriptor at the sender. When a pointer is deleted at a site, the corresponding node is deleted from the tree only if it was a leaf, if not, the descriptor is kept in the structure until it becomes a leaf (to avoid doing a distributed deletion, the leaf case being trivial).

The tree is pointed from the children to the parent, and each node keeps a counter with the number of children. This counter is in fact the indirect reference count, because when a counter reaches zero, the descriptor and the remote pointer can be deleted locally without any problem. When the counter at the root node (the owner) is zero, the complete object can be deleted. The tree structure can be seen at Figure 3.

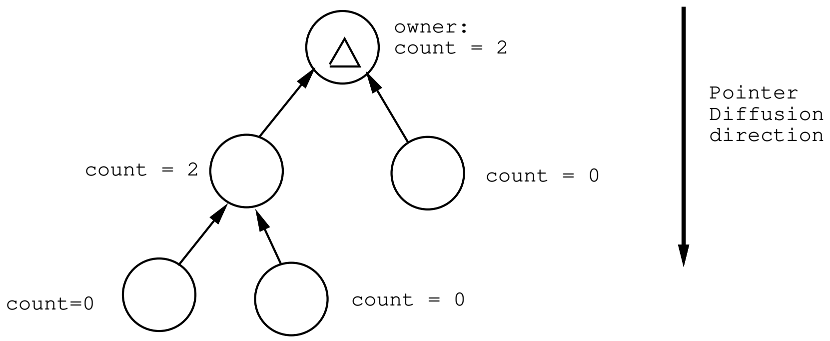

Figure 3: The GC structure

The advantage is that this structure can be updated with only a few messages (to decrement the counters), and even migrations are handled neatly, because in such a tree it is trivial to take any node and make it the new root of the tree.

We must note that every node also keeps a pointer to the root: the *owner* field of the remote pointer. Accesses to values are done directly using this pointer, the indirect pointers to the parent are only used for garbage collection purposes.

Comparing it with Weighted Reference Count, this algorithm consumes more space per pointer, but never creates *indirection cells* and permits an easier migration protocol.

5 The Remote Process Creation Primitives

5.1 Task Creation

Probably the most known Lisp primitive to create tasks is the *future* construct[Hewi 77, Hals 85], which simply creates a placeholder for a value which is computed in parallel. There are many implementations with subtly different semantics[Hals 85, Kran 89, Mohr 90] but the idea is always the same.

5.2 The Shared Memory Model

In the shared memory model, the form `(future` X`)` immediately returns a *future* for the value of X and creates a task to concurrently evaluate X. When the evaluation of X yields a value, the value *replaces* the *future* (the future *resolves* to a value), and any task that tries to use the value before being computed is blocked waiting for it. This means that every access must be tested to see if the value is a *future* or not. On the other hand, this permits (in principle) to add the *future* construct on any sequential program, obtaining the same results with an eventual speedup. However, this is not completely true, as side-effects (including I/O) are not necessarily executed in the same order. Using side-effects and *futures* can result in deadlocks, with a task waiting for the result of another task which is also waiting for the first. The *future* is a synchronization object, permitting the parallel execution of the continuation (the calling process) and the evaluation of the argument expression.

5.3 The Distributed Memory Model

In a distributed system, the creation of tasks is a very delicate point. The overhead associated with creating a task on another processor is very high, and so the possible granularity is only medium to coarse. It is not our intention to take sequential programs, just annotate them and then run them on our system. So, we chose a different *future* semantics, and `(future` X`)` returns always a *future* (the placeholder), which is a new Lisp data type with only one operator to get its value, called *touch*. In fact, this *future* can be seen as a communication channel of size one, with a *write-once* and *read-by-many* behavior. Using it this way contribute to clarify the semantics of the operations. We can even say that the evaluation of the expression can be done in parallel or not (when the load is high), without changing the definition.

We believe that the explicit *touch* permits better control of the task creation, and has no running time overhead. Side effects can be used on the system, but the order is not guaranteed. Also deadlocks on the *touch* relation are less probable as they become more visible. On the other hand, we force an indirection even if the task is finally not created, and the programmer has more work to do, because the addition of a *future* is not transparent.

5.4 Non-determinism

In a distributed system, non-determinism can be very useful, permitting *speculative* computation[Osbo 90], to create parallel computations whose results are eventually not used (as a parallel or). We have added a primitive called `select-future`, which receives a list of *futures* as argument and return a resolved future from the list. If there is no resolved future on the list, `select-future` blocks waiting for one. A similar primitive, called `test-future` avoids the blocking phase and just returns a resolved *future* or *NIL*. This permits to create many tasks and just wait for the first result.

5.5 Implementation Issues

We have build a prototype version of TransPive, on an interpreted system running on a Transputer network. The *future* primitive just looks for a neighbor processor to execute the task. If every processor is heavyly loaded, then it evaluates its argument locally, without creating a new task (this has been called *inlining* in [Hals 85]). This can add deadlocks in unexpected places (as is noted in [Mohr 90]), and the rule to respect is that a *future* can only be used safely if the code works independently of being run sequentially or in parallel.

The *touch* primitive is coupled with the remote pointer mechanism, and we guarantee that an unresolved future is never sent outside the owner site. A *touch* simply sends a message to a server on the owner site, which adds the requester to a waiting queue (if the *future* is not resolved) or replies with the value.

The **select-future** primitive is more complicated, as we send messages to every site holding a *future* but we wait for only one answer. This is done going through a local server. The remaining requests are not aborted, the local server just accepts the extra answers (with no one waiting for them), keeping a table with every request already sent out, and still not replied. This is based on the assumption that the process will probably wait on the same unresolved futures again. If a *wait-future* is issued a second time on the same *future*, there is a request already sent for it, so a second message is not needed.

The *future* primitive always creates a placeholder, and returns a pointer to it. If there is an available processor, the placeholder is created there and the pointer is just a remote pointer to it. While the *future* is unresolved there is a waiting queue for the pending touches, and a flag indicating that the task is still running. The future structure with the queue is never sent outside the processor where it is being computed, until it is resolved. Upon completion, only the value is sent to every process of the waiting queue.

For the **wait-future** and **test-future** primitives, there is a *future server* which distributes the requests and waits for the answers (see Figure 4). If the *touch* is applied to a local *future* the actions are the same (just sending messages to local servers).

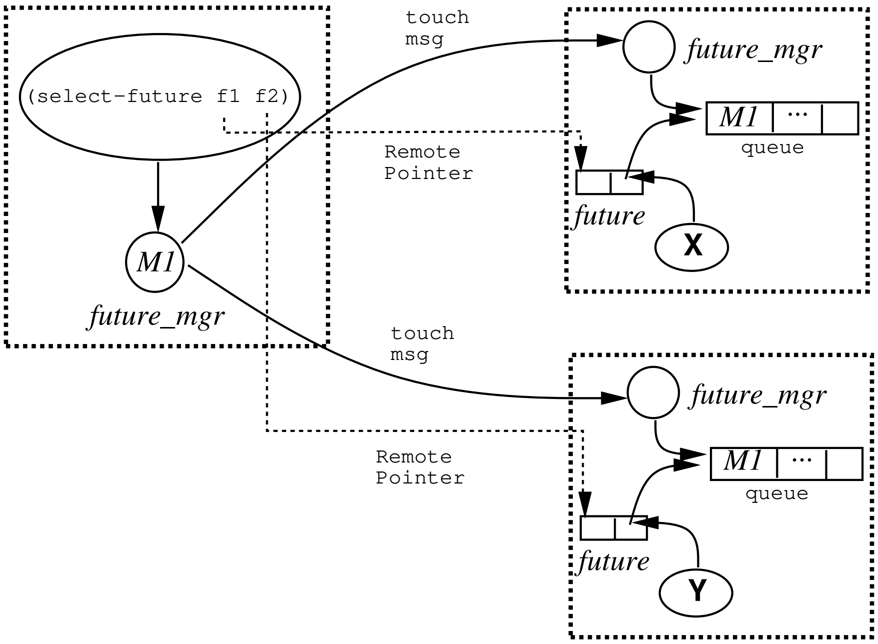

Figure 4: The future implementation

6 Distributed Algorithm Prototyping

TransPive provides a nice testbed for distributed algorithm prototyping. We have build some examples using *futures*, and some important ideas have been found for fast prototyping.

It is usually a good idea to begin with a sequential algorithm and try to find the places where to put some *futures*. The general idea is to put a *future* where the expression to be evaluated as a *future* has a high expected execution time (compared to the remaining computation). However, the experience shows that other considerations are very important:

- Computation versus Space

  If the *future* expression requires an important amount of data from the original space, it will be copied on demand to the evaluating processor. If the order of magnitude of computation and data are comparable, the parallel version will be probably slower than the sequential version. The same is true if the resulting data used by the original processor is big.

- Shared Data

  If there is shared data, modified concurrently by both computations, a *future* should be avoided. Not only the execution time is bad, also the code will probably be broken, as we are introducing parallelism where the program was expecting sequential behavior.

- Recursive Calls

  A nice place where to put *future*'s is on non-terminal recursive calls. However, caution should be used in order to control the amount of *future* calls at execution time. The idea is to use a *future* only when a recursive call will take an important computation time. Sometimes this is not obvious, because even at execution time some programs can not know how much work will correspond to a given call (a tree traversal for example).

- Non-Determinism

  Many recursive algorithms use a loop of recursive calls, looking for a valid result (a graph search for example). The loop can be used to create a *future* for each call, and then wait for any result using `select-future`. This code is very efficient, but care should be taken with the evaluations that are never waited for. Currently, these computations continue using processing power until finished.

7 Measures and Conclusions

Our first prototype runs on an interpreted Lisp (the compiler exists but must be ported). However it is very useful to test the gains we can obtain parallelizing the applications. The grain is very coarse, the call `(touch (future 0))` can take near 0.4 seconds, and so a *future* must appear only before an expression taking more than that to evaluate. For example, on a tic-tac-toe playing program, we don't use more *future*'s when there are less than five empty cases to analyze, which takes about 0.5 seconds sequentially. For the playing program, we use the non-deterministic `select-future` primitive, to get the first result already calculated. If the program find a move that permits to win the game, it returns immediately, without waiting for the other results.

We present the results of a tic-tac-toe playing program (with an empty board to evaluate), using *futures* (limited to a certain level) and non-determinism. A Fibonacci function, using low-level task creation is also measured, and an extensive search on a graph, looking for the shortest path between two nodes. The sequential versions are given without any *future* in the code, to show the real speedup. The parallel version for the graph search has two values, for the first and second execution with the same data. The second time it goes faster because the data is already distributed on the local memories and is kept on the local caches.

	tic tac toe	fib 24	search
sequential	663	26	76
5 processors	257	6	52/37
11 processors	136	–	—

Total Running Time (s)

We can see that, with careful programming, we can get speedups even with only five processors. Considering that all the global address space and the message passing system is included in the language (and interpreted in Lisp), the performances are interesting.

The design of TransPive makes it portable to any distributed machine of the MIMD type, as we are not considering any hardware support (excepting some message-passing mechanism).

Many future work is planned, new garbage collectors, new update propagation and load balancing. The data structure sharing mechanism should be tuned to get better performances along with the low-level message passing primitives.

8 Acknowledgements

This work has been partially financed by INRIA-Rocquencourt project ICSLA, the French-Chilean cooperation program, the University of Chile and the FONDECYT project 92-1163.

References

[Bal 88] H. E. Bal, A. S. Tanenbaum, *Distributed Programming with Shared Data*, Proc. of the IEEE CS 1988 International Conference on Computer Languages, Miami, Fla. October 9-13, 1988.

[Beva 87] D. I. Bevan, *Distributed Garbage Collection Using Reference Counting*, LNCS 259, PARLE Proceedings Vol II, Eindoven, Springer Verlag, June 1987.

[Birr 84] A. D. Birrel, B. J. Nelson, *Implementing Remote Procedure Calls*, ACM TOCS, V. 2, N. 1, pp. 39-59, February 1984.

[Blac 87] A. Black, N. Hutchinson, E. Jul, H. Levy and L. Carter, *Distribution and Abstract Types in Emerald*, IEEE Trans. Software Eng., Vol. SE-13, N. 1, pp. 65-76, January 1987.

[Chai 84] J. Chailloux, M. Devin, J. M. Hullot, *Le-Lisp: A Portable and Efficient Lisp System* Proc. 1984 ACM Symposium on Lisp and Functional Programming, Août 1984.

[Chai 90] D. Chaiken, C. Fields, K. Kurihara and A. Agarwal, *Directory-Based Cache Coherence in Large-Scale Multiprocessors*, IEEE Computer, June 1990.

[Fitz 86] R. Fitzgerald and R. F. Rashid, *The Integration of Virtual Memory Management and Interprocess Communication in Accent*, ACM Trans. on Computer Systems, Vol. 4, N. 2, pp 147-177, May 1986.

[Gabr 84] R. P. Gabriel and J. McCarthy, *Queue-based Multi-processing Lisp*, ACM Symp. on Lisp and Functional Programming, Austin, Texas, August 1984.

[Hals 85] R. Halstead, *Multilisp: A Language for Concurrent Symbolic Computation*, ACM Trans. on Prog. Languages and Systems, Vol. 7, N. 4, October 1985, pp. 501-538.

[Hewi 77] C. Hewitt, H. Baker, *The Incremental Garbage Collection of Processes*, ACM Conference on AI and Programming Languages, 1977, pp. 55-59.

[Katz 85] R.H. Katz, S.J. Eggers, D.A. Wood, C.L. Perkins, R.G. Sheldon, *Implementing a Cache Consistency Protocol*, Proc. of the 12th annual Symp. on Computer Architecture, pp. 276-283, June 1985.

[Kran 89] D. A. Kranz, R.H. Halstead, E. Mohr, *Mul-T: A High-Performance Parallel Lisp*, SIGPLAN '89 Conference on Prog. Languages Design and Implementation, Portland, Oregon, June 21-23, 1989.

[Lamp 78] L. Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*, Comm. ACM, Vol. 21, N. 7, pp. 558-565, July 1978.

[Lerm 86] C. W. Lermen, D. Maurer, *A Protocol for Distributed Reference Counting*, Proc. of the ACM Conference on LISP and Functional Prog., Cambridge, Massachussets, August 1986.

[Li 86] K. Li, P. Hudak, *Memory Coherence in Shared Virtual Memory Systems*, Proc. of the 5th annual ACM Symp. on Principles of Distributed Computing, Calgary, Canada, August 13-18, 1986.

[Mill 87] J. Miller, *MultiScheme: A Parallel Processing System Based on MIT Scheme*, Proc. SIGPLAN '86 Symp. on Compiler Construction, June 1986, pp. 219-233.

[Mohr 90] E. Mohr, D. A. Kranz and R. H. Halstead, Jr., *Lazy Task Creation: A technique for Increasing the Granularity of Parallel Programs*, Proc. 1990 ACM Conference on Lisp and Functional Prog., Nice, France, June 1990, pp. 185-197.

[Osbo 90] R. B. Osborne, *Speculative Computation in Multilisp, An Overview*, Proc. 1990 ACM Conference on Lisp and Functional Prog., Nice, France, June 1990, pp. 198-208.

[Piqu 90] J. M. Piquer, *Sharing Data Structures in a Distributed Lisp*, Proc. High Performance and parallel Computing in Lisp Workshop, Twickenham, London, UK, November 1990.

[Piqu 91a] J. M. Piquer, *Parallélisme et Distribution en Lisp*, Thèse de Doctorat d'Informatique, École Polytechnique, Paris, France, 1991.

[Piqu 91b] J. M. Piquer, *Indirect Reference Counting: A Distributed GC*, LNCS 505, PARLE '91 Proceedings Vol I, pp. 150-165, Springer Verlag, Eindhoven, The Netherlands, Jun. 1991.

[Piqu 91c] J. M. Piquer, *Preserving Distributed Data Coherence Using Asynchronous Broadcasts*, Proc. of the XI International Conference of the Chilean Computer Science Society, Plenum Press, Santiago, Chile, Oct. 1991.

[Rayn 89] M. Raynal, A. Schiper, S. Toueg, *The Causal Ordering Abstraction and a Simple Way to Implement It*, INRIA Research Report 1132, Dec. 1989.

[Rozi 88] M. Rozier, et al., *Chorus Distributed Operating Systems*, Computing Systems, Vol. 1, N. 4, 1988, pp. 305-367.

[Schi 89] A. Schiper, A. Sandoz, J. Eggli, *A New Algorithm to Implement Causal Ordering*, 3rd. Workshop on Distributed Algorithms, LNCS 392, pp. 219-232, 1989.

[Serp 84] B. P. Serpette, *Contextes, Processus, Objets, Séquenceurs: FORMES*, Thèse de 3ème cycle, LITP, Paris VI, October 1984.

[Wats 87] P. Watson, I. Watson, *An Efficient Garbage Collection Scheme for Parallel Computer Architectures*, LNCS 259, PARLE Proceedings Vol II, Eindoven, Springer Verlag, June 1987.