Using Mobility and Blackboards to Support a Multiparadigm Model Oriented to Distributed Processing

Jorge Luis Victória Barbosa [1,3], Adenauer Corrêa Yamin[1,3], Patrícia Kayser Vargas[2], Débora Nice Ferrari[2], Alberto Egon Schaeffer[3], Cláudio Fernando Resin Geyer[3]

[1] Informatics Department, Catholic University of Pelotas (UCPel)
Pelotas, RS, Brazil
{barbosa, adenauer}@atlas.ucpel.tche.br

[2] Informatics Department, LaSalle University Center (UNILASALLE)
Canoas, RS, Brazil
{kayser, nice}@lasalle.tche.br

[3] Informatics Institute, Federal University of Rio Grande do Sul (UFRGS)
Porto Alegre, RS, Brazil
{barbosa, adenauer, egon, geyer}@inf.ufrgs.br

Abstract—
***Holoparadigm* (*Holo*) is a multiparadigm model oriented to development of parallel and distributed programs. In this paper we propose the *Distributed Holo* (*DHolo*), a model to support the distributed execution of programs developed in Holo. DHolo is based on object mobility and blackboards. This distributed model can be fully implemented on Java platform. Specifically, mobility is implemented using Voyager and blackboard using Jada tuple space.**

Keywords— **Multiparadigm, Mobility, Blackboard and Distributed Processing.**

I. Introduction

In the last years the multiparadigm theme has been continually researched [HAN 94, MUL 95, AMA 96, LEE 97, APT 98, PIN 99]. Researchers have proposed models of software development through the integration of basic paradigms (mainly: imperative, logic, functional, and object-oriented paradigms). Using this approach, they have been looking for two goals: (a) to overcome the specific limitations of each paradigm and (b) to take advantage of the most useful characteristics of each one through their combination.

Each paradigm has sources of implicit parallelism. For example, the exploitation of AND parallelism and OR parallelism in logic programming [BAR 00, VAR 00]. Another example is object-oriented paradigm that allows the exploitation of inter-object parallelism and intra-object parallelism [NGK 95, CIA 96]. The multiparadigm approach integrates paradigms. So, it also integrates their parallelism sources. In this context, interest in automatic exploitation of parallelism in multiparadigm software has emerged. Enlargement of this approach guides the studies to distributed systems where the mobility, the heterogeneous hardware and the use of networks as parallel architectures are considered. The development of distributed software using multiparadigm models has received attention of the scientific community [NGK 95, CIA 96, ROY 97, HAR 98, HAR 99].

In this context, we are creating the *Holoparadigm* (*Holo*). Holo is a multiparadigm model oriented to automatic exploitation of parallelism and distribution. Holo has a coordination model based on levels of blackboards (called *histories*) encapsulated in new programming entities called *beings*. A new language (*Hololanguage* [BAR 01]) implements the concepts proposed by the Holoparadigm.

In this paper, we propose a model that supports the distributed execution of programs developed in Holo. This model is called *Distributed Holo* (*DHolo*). DHolo has a network as physical execution environment and is based on object mobility and blackboard. It was implemented using Java [JAV 01] and two special libraries to support mobility (Voyager [VOY 01]) and blackboards (Jada [CIP 01]).

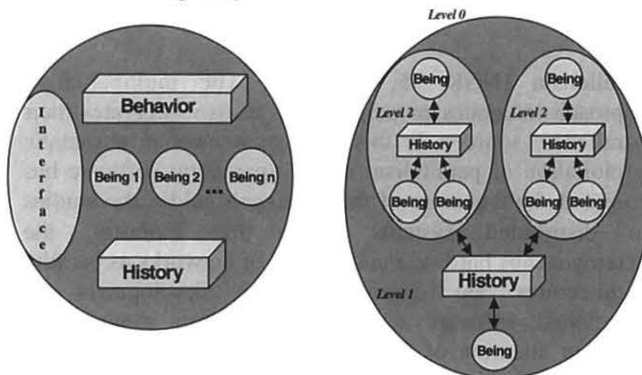
The paper is organized as follow. The section two presents the main concepts of Holoparadigm and describes the principles of the Hololanguage. In the section three the

Distributed Holo is proposed. Experimental results are presented and discussed in the section four. The section five describes related works. Finally, the section six draws some conclusions and presents directions for future works.

II. HOLOPARADIGM AND HOLOLANGUAGE

Being is the main Holoparadigm abstraction. There are two kinds of beings: *elementary being* (atomic being without composition levels) and *composed being* (being composed by other beings). An elementary being is organized in three parts: *interface*, *behavior* and *history*. The interface describes the relations between beings. The behavior contains actions, which implement functionalities. The history is a shared storage space in a being. A composed being (figure 1a) has the same organization, but may be composed by others beings (*component beings*).

Each being has its history. The history is encapsulated in the being. In composed being, the history is shared by component beings. Therefore, it is possible to exist several levels of encapsulated history. A being uses the history in a specific composition level. For example, figure 1b shows two levels of encapsulated history in a being with three composition levels. Behavior and interface parts are omitted for simplicity.



(a) Composed being structure (b) Example of composition (3 levels)

Fig. 1 Being organization

Automatic distribution is one of the main Holoparadigm goals. Figure 2 exemplifies a possible distribution to the being presented in the figure 1b. Besides that, the figure presents the mobility in Holo. The being is distributed in two nodes of the distributed architecture. The history of a distributed being is called *distributed history*. This kind of history can be implemented using DSM techniques [PRO 99] or distributed shared spaces [CIP 94, AMB 96].

Mobility [IEE 98] is the dislocation capacity of a being. In Holo, there are two kinds of mobility: *logical mobility* (being is moved when crosses one or more borders of beings) and *physical mobility* (dislocation between nodes of distributed architectures). Figure 2 exemplifies two possible mobilities in the being initially presented in the figure 1b. After the dislocation, the moveable being is unable to contact the history of the source being (figure 2, mobility A). However, now the being is able to use the history of the destiny being. Here, physical mobility only occurs if the source and destiny beings are in different nodes of the distributed architecture (it is the case in our example). Logical and physical mobilities are independent. Occurrence of one does not imply in the occurrence of the other. For example, the mobility B in the figure 2 is a physical mobility without logical mobility. In this example, the moved being does not change its history view (supported by the blackboard). This kind of situation could happen if the execution environment aims to speedup execution through locality exploitation.

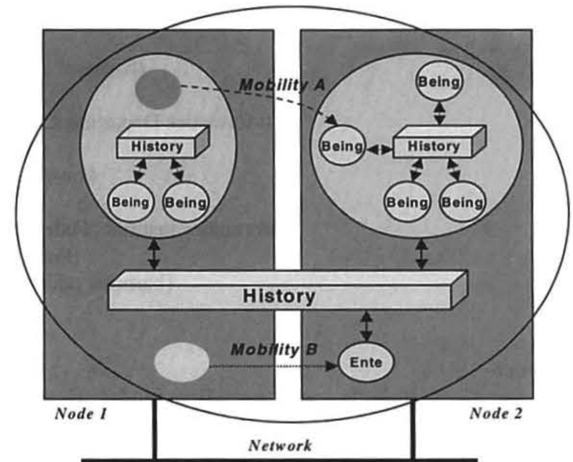
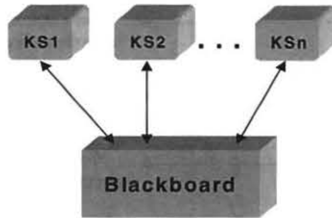


Fig. 2 Distributed being and mobility

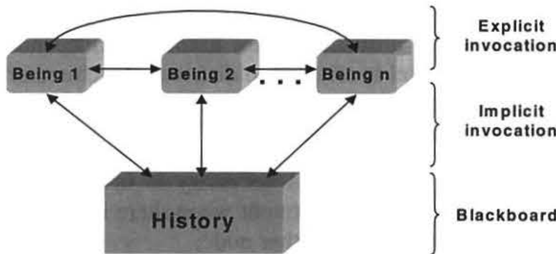
The coordination model used in Holo is based on the *blackboard architecture* [NII 86] (figure 3a). This architecture is composed by a common data area (blackboard) shared by a collection of programming entities called *knowledge sources* (KSs). Control is implicit in the blackboard access operations. The read and write operations in the blackboard are used to communication and synchronization between KSs. This kind of control is called *implicit invocation*. A composed being architecture is similar to the blackboard architecture, since several components sharing a common data area. In Holo, KSs are beings and the blackboard is the history. By the way, there are several limitations in blackboard implicit invocation. Introduction of *explicit invocation* in the coordination model eliminates these limitations. Any direct call between entities is called explicit invocation. In Holo, the beings influence the others using the history, but can change information directly too.

Figure 3b shows the Holo coordination model. History is a logic blackboard, i.e., the information stored is a group of logic terms. Interaction with the history uses two kinds of Linda-like [CAR 89] operations: *affirmation* and

question. An affirmation puts terms in the history, like asserts in Prolog databases. Moreover, a question permits to consult terms from the history. A consult does a search in the database using unification of terms. A question is *blocking* or *non-blocking*. A blocking question only returns when a unifying term is found. Therefore, blocking questions synchronize beings using the implicit invocation. In a non-blocking question, if a unifying term is not found, the question immediately fails. Besides that, a question is *destructive* or *non-destructive*. A destructive question retracts the unifying term. The non-destructive one does not remove it.



(a) Blackboard architecture



(b) Holo coordination model

Fig. 3 Holo coordination model

Hololanguage (so-called Holo) [BAR 01] is a programming language that implements the concepts of Holoparadigm. A program is composed by descriptions of beings. Figure 4 shows a description of a being using the structure presented in the figure 1a. Interface shows the actions accessible to other beings. Behavior contains the actions, which implement the being functionality. History is a logic blackboard used by actions and component beings. Initial state of history is a set of logic terms introduced by the programmer.

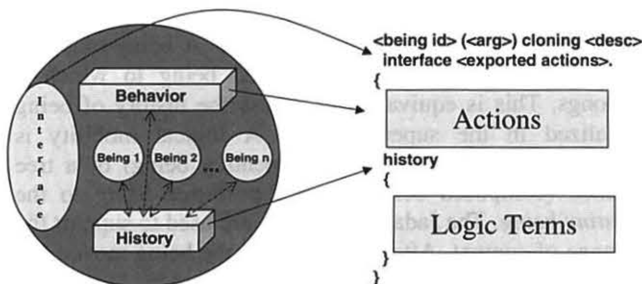


Fig. 4 Being description

The behavior supports five kinds of actions: logic actions (LA), imperative actions (IA), modular logic actions (MLA), modular imperative actions (MIA) and multiparadigm actions (MA). LA is a logic predicate. IA is a group of imperative commands. MLA contains several logic actions encapsulated in a module. MIA encapsulates several imperative actions. MA integrates logic and imperative actions.

Actions are composed using an *Action Composition Graph* (ACG, figure 5a). Following the Peter Wegner's opinion [WEG 93] about the impossibility of mixing logic and imperative behaviors, we have created the *Action Invocation Graph* (AIG) presented in the figure 5b. This graph determines the possible order of action calls during a program execution. MAs, IAs and MIAs call any action. LAs and MLAs only call LAs and MLAs. Therefore, there are two regions of actions during an execution, namely, imperative and logic regions (figure 5b).

If an execution flow goes in the logic region, the only way to return to the imperative region is finishing the flow (returning the results asked from the imperative region). This methodology eliminates many problems, which emerge when logic and imperative commands are mixed (for example, distributed backtracking [BOS 96]). We believe AIG is an important contribution to the discussion presented by Wegner [WEG 93, WEG 97].

The language supports logical mobility (command *move*), concurrency between actions of a being (command *spawn*) and several kinds of cloning (command *clone*). Besides that, the Hololanguage permits both kinds of blackboard interaction proposed by the Holoparadigm. Affirmation uses the symbol "!". Moreover, there are several question types. The symbol "." indicates a blocking question and the symbol "?" is used for non-blocking. Besides that, there are non-destructive (default) or destructive (symbol "#") questions. Finally, a question can return multiple answers. The symbols ".", "?", "#", and "*" (all answers) configure the questions.

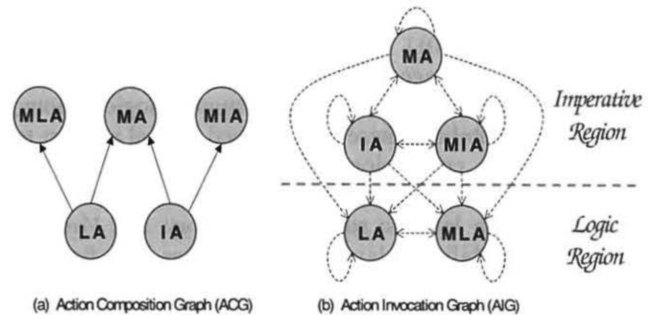


Fig. 5 Composition and invocation graphs

III. DISTRIBUTED HOLO

Holo is oriented to development of distributed systems. It was created to support the implicit distribution, i.e., automatic exploitation of distribution using mechanisms provided by basic software (compiler and execution environment). Looking for this, we propose a platform to the Holoparadigm (called Holoplatform, see figure 6). Two parts compose the Holoplatform:

- *development platform*: tools used for software development (HoloCase, HoloJava and Java compiler);
- *execution platform*: hardware and software used to support the distributed execution of programs (DHolo and distributed architecture).

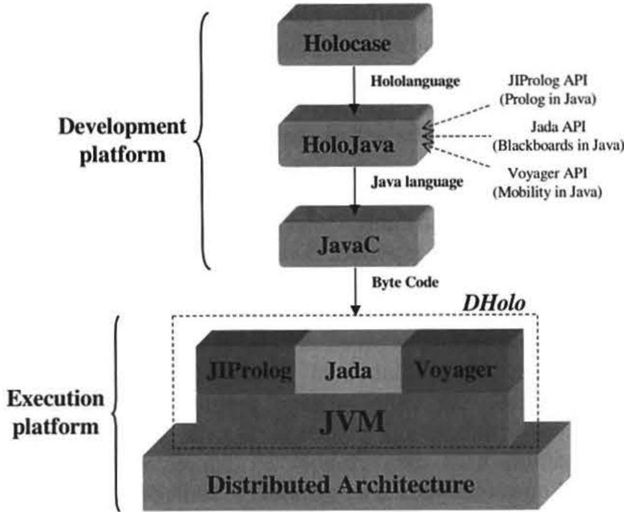


Fig. 6 Holoplatform

HoloCase supports visual programming based on the abstractions proposed by Holo and generates programs in the Hololanguage. HoloJava converts programs in Holo to Java using a transformation policy. Several works indicated that Java [JAV 01] is adequate to be used like an intermediate language [HAJ 96, PRO 01]. On the other hand, standard Java cannot directly support logic actions, history and mobility. These aspects need special support to implement the conversion. We use *JIProlog* [CHI 01] (Prolog in Java) to implement logic actions, *Jada* [CIP 01] (blackboards in Java) to support the history, and *Voyager* [VOY 01] (mobility in Java) to move beings. The figure 7 shows the HoloJava transformation.

Concurrent actions (*spawn* command) generate threads in Java. Logic actions are converted in JIProlog methods. Besides that, each *move* command can generate a *moveTo* method of Voyager. Each history is equivalent to a Jada tuple space and all kinds of history invocations can be directly converted in operations to spaces in Jada.

Fig. 7 HoloJava transformation policy

Holoparadigm abstractions are hardware independent. However, the model is oriented to distributed architectures. When the hardware is distributed, there are two main characteristics to be considered:

- *mobility support*: it is necessary to implement the physical mobility treatment when there is a move to a being located in another node;
- *dynamic and hierarchical history support*: distribution involves data sharing between beings in different nodes (distributed history). There are several levels of history (hierarchical history). Besides that, access history is adapted during the execution to support the mobility (dynamic history).

DHolo is the software layer that supports the distributed execution of programs in Holo. It creates support to physical mobility and dynamic/hierarchical history in a cluster of workstations. DHolo project is based on a structure called *Tree of Beings*. This structure is used to organize a being during its execution. For example, the being in the figure 1b has the tree shown in the figure 8a. The tree organizes the beings in levels. A being only can access the history of the composed being to which it belongs. This is equivalent to access the history of being localized in the superior level. A logical mobility is implemented moving a leaf (elementary being) or a tree branch (composed being) from the *source being* to the *destiny being*. The Jada tuple spaces are used to support the change of context. After the mobility, the being moved has direct access to the destiny being's space (composed being's history).

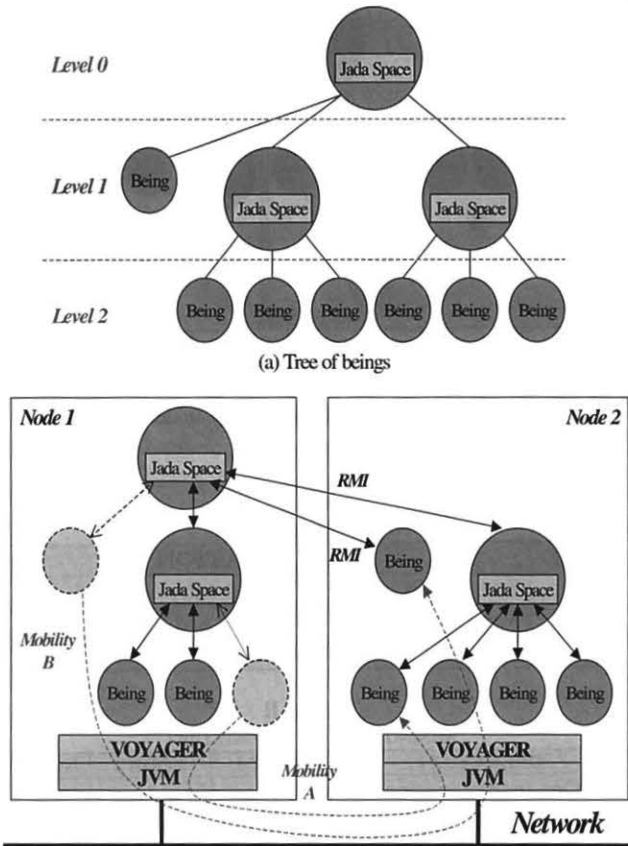


Fig. 8 Distributed Holo

The figure 8b presents the DHolo architecture to the being initially shown in the figure 2. The figure shows the tree of beings distributed in two nodes. The changes to both mobilities of figure 2 are demonstrated. Each being is implemented using an object (*interface* and *behavior*) and a Jada space (*history*). DHolo installation involves the creation of a *Voyager-enabled program* (Voyager environment) in each node that will be used. Since Voyager executes on the Java Virtual Machine each node will also have a running JVM.

During the installation, a *Table Environment (TE)* indicates the nodes that will be involved by DHolo. During a program execution, if a logical mobility results in a physical mobility, it is realized a *moveTo* operation in Voyager (mobility A, figures 2 and 8b). When a physical mobility is realized without logical mobility (mobility B, figures 2 and 8a), the tree of beings does not change, but a *moveTo* operation in Voyager is realized. This kind of mobility does not have any kind of relation with the program. It is a decision of the environment to support a specific functionality (load balancing, tolerance fault, etc). DHolo does not support this kind of decision yet.

IV. Experimental Results

We have done a group of experiments using a distributed architecture with the following goals:

- evaluate the DHolo model. This goal involves the evaluation of logical and physical mobility control using the distributed tree. Besides that, it is important to evaluate the context change control using tuple spaces;
- evaluate the technologies used to implement the DHolo prototype, more specifically, Voyager and Jada.

Our experiments are based on a specific application: datamining. We have simulated three standard cases of datamining in a network environment:

A) *Sequential datamining in one node* (figure 9): one miner mines three databases (mines) in the same node. The miner changes its level using mobility and goes in the first mine (mark 1). It mines the history using a specific number of read operations to Jada space (mark 2). After that, the miner goes out of the mine (mark 3) and writes the data mining result (mark 4) in the history of the most general being (called Holo). This behavior is repeated to the others two mines;

B) *Sequential datamining in three nodes* (figure 10): one miner mines three mines localized in different nodes. The behavior is the same of the first situation. However, the two last mines are located in different nodes. This implies in two important points. First, when the miner goes in the second and third mines (marks 1), it is necessary to use physical mobility (*moveTo* command of Voyager). Second, when the miner writes the results of the second and third mining (marks 2), it is necessary to use remote method invocation (RMI). Both points result in costs that should be measured;

C) *Parallel datamining in three nodes* (figure 11): three miners mining in parallel three mines localized in different nodes. The situation is similar to the case B. The unique difference is that there are three miners that work in parallel over mines. This solution involves the realization of two physical mobilities (marks 1) and two RMIs (marks 2).

Holoparadigm is a generic model created to be used in any kind of distributed architecture. Following this idea, our experiments have used heterogeneous nodes, a common situation in a typical computers network. Table I contains the nodes specification. Table II shows the cost of one basic datamining operation at each node. Our basic operation is one tuple read (two fields, a string and an integer) from the mine tuple space (for example, mark 2 in figure 9). Each node has a different cost due to heterogeneity. Table III

presents the network costs involved in our experiment. Table IV shows the versions of software utilized.

Each case was executed using five grains (number of mining operations). The table V contains the average of several executions and the standard deviation. Besides, figure 12 presents a graphic representing the results listed in the table V. All times are shown in milliseconds and the network used in the experiment has a bandwidth of 10 Mbps.

Considering the results, some interesting points can be underlined. Thinking in performance, case C is the best solution after around 2500 operations. This fact stimulates the parallelism exploitation. Besides, case B overcomes the case A after around 3750 operations. This results from the fact that the case A was executed in the node of smaller computational power (node 1, see tables I and II). The network costs (see table III) were overcome by the use of more powerful nodes to work in the mines 2 and 3. Thinking in functionality, it is also important to say that almost all the times the mines distribution is related to data locality. Getting speedup is desired but not required.

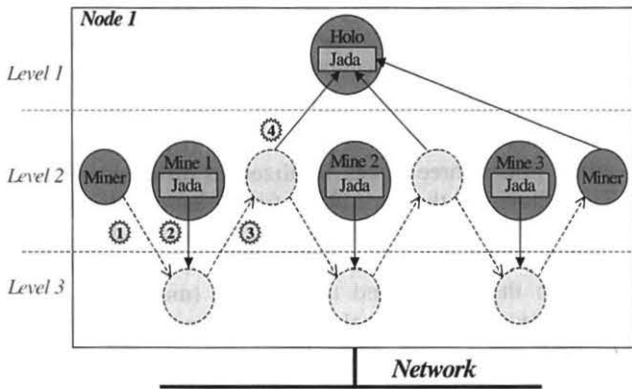


Fig. 9 Sequential datamining in one node (case A)

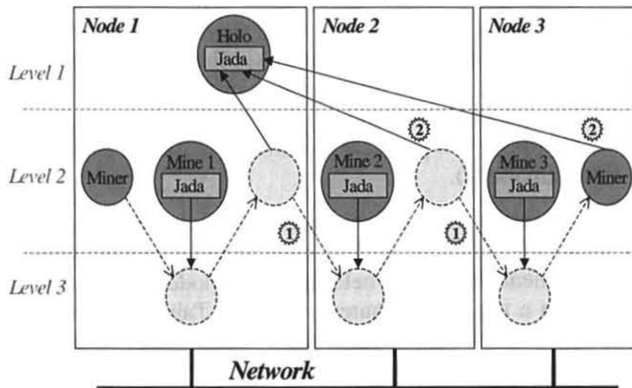


Fig. 10 Sequential datamining in three nodes (case B)

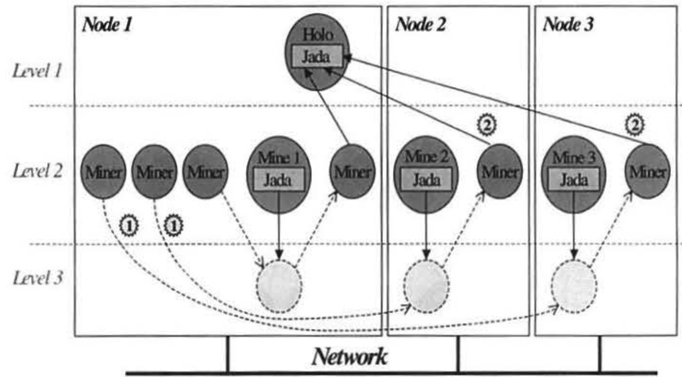


Fig. 11 Parallel datamining in three nodes (case C)

TABLE I
NODES SPECIFICATION

NODE	SPECIFICATION
1	Sun SPARCstation 20 – 128 Mbytes RAM
2	Sun Ultra 10 – 128 Mbytes RAM
3	Sun Ultra 5 – 192 Mbytes RAM

TABLE II
MINING OPERATION COSTS (ms)

OPERATION	NODE 1	NODE 2	NODE 3
One read operation (like mark 2, figure 9)	0.129	0.032	0.040

TABLE III
NETWORK COSTS (ms)

OPERATION	COST
One remote write (like marks 2, figure 10)	193
One physical mobility (like marks 1, figure 10)	594

TABLE IV
SOFTWARE VERSIONS

SOFTWARE	VERSION
Operating system	SunOS Release 5.7
Voyager	Version 3.3
Java	Version 1.2
Jada	Version 3.0 beta 7

TABLE V
DATAMINING BENCHMARKS (ms)

Mining operations	CASE A		CASE B		CASE C	
	Aver.	St dv	Aver.	St dv	Aver.	St dv
1000	746.4	12.4	1635.3	75.1	1298.3	66.2
2000	1273.6	8.0	1841.1	36.2	1413.6	15.4
3000	1818.5	17.2	2079.1	43.6	1605.0	85.0
4000	2409.5	16.8	2310.7	23.2	1714.8	56.6
5000	2903.9	85.8	2674.2	142.4	1874.6	88.4

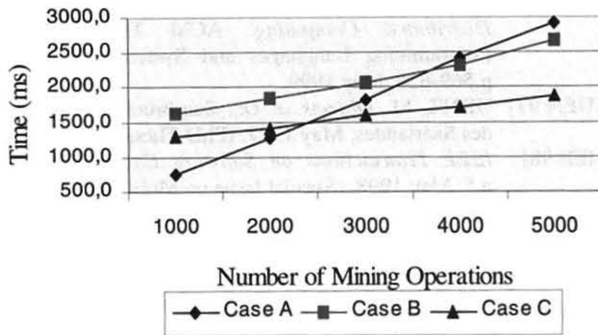


Fig. 12 Datamining benchmarks

V. RELATED WORKS

There are other multiparadigm implementations over distributed environment. I^{+} model [NGK 95] supports the distribution of objects, which implement methods using functions (functional classes) and logic predicates (logic predicates). The implementation is based on the translation of functional classes into LML modules and translation of logic predicates into Prolog modules. The distributed architecture is a network of Unix workstations using 4.3 BSD sockets to implemented message passing. The run-time environment was initially implemented using C language, Quintus Prolog and Lazy ML (LML). In a second phase, programs were only translated into C language. This proposal does not focus mobility. In addition, none kind of shared space is supported between objects.

Ciampolini et al [CIA 96] have proposed DLO, a system to create distributed logic objects. This proposal is based on previous works (Shared Prolog [BRO 91], ESP [CIP 94] and ETA [AMB 96]). The implementation is based on the translation of DLO programs into clauses of a concurrent logic language called Rose [BRO 90]. The support to Rose execution is implemented on a MIMD distributed memory parallel architecture (transputer-based Meiko Computing Surface). The run-time environment consists of a parallel abstract machine which is an extension of the WAM [AIT 91]. This proposal does not support mobility and it is not applied in a network of workstations. DLO does not support levels of spaces.

Oz multiparadigm language [MUL 95] is used to create a distributed platform called Mozart [ROY 97]. Oz uses a constraint store similar to a blackboard and supports the use of several paradigm styles [MUL 95, HEN 97]. Besides, Mozart has special support to mobility of objects [HAR 98] and distributed treatment of logic variables [HAR 99]. Mozart distributed architecture is a network of workstations providing standard protocols such as TCP/IP. The run-time environment is composed by four software layers [ROY 97]: Oz centralized engine (Oz virtual machine [MEH 95]), language graph layer (distributed algorithms to decide when to do a local operation or a network communication), memory management layer (shared communication space and distributed garbage collection) and reliable message layer (transfer of byte sequences between nodes). The physical mobility supported by Mozart is completely transparent, i. e., the system decides when to move an object. None kind of logical mobility is used. The shared spaced supported by Mozart is monotonic and stores constraints. Being's history is a non-monotonic logic blackboard that stores logic tuples (terms). In addition, Mozart does not provide levels of encapsulated contexts composed by objects accessing a shared space.

Tarau has proposed Jinni [TAR 99], a logic programming language that supports concurrency, mobility and distributed logic blackboards. Jinni is implemented using BinProlog [BOS 96] a multi-threaded Prolog system with ability to generate C/C++ code. Besides that, it has special support to Java, such as a translator allowing packaging of Jinni programs as Java classes. Jinni is not a multiparadigm platform. In addition, Jinni does not work with logical mobility and with levels of encapsulated blackboards.

So, Oz and Jinni have a kind of mobility. In addition, they can be executed over network of workstations. However, we believe that the support to hierarchy of spaces as proposed by Holo is an innovation.

VI. CONCLUSION

We have proposed an environment to distributed execution of a new multiparadigm language. Our main contribution is the transparent distributed support to the Holo programming model. DHolo automatically manages the tree of beings distribution.

One important aspect of Holo is the coordination model, which simplifies the management of mobility. For this coordination, the Holo model uses a logic blackboard while DHolo proposes the use of a tuple space to implement it. Another important concept is the autonomous management of mobility. Holo model does not deal with physical distribution so mobility is always at logic level, i.e, between beings. DHolo execution environment can define what kind of mobility is necessary: a logical or a physical one. A logical requires changes in history sharing while physical also involves Java objects mobility.

Our experiments have shown interesting results. Voyager and Jada can work cooperatively. Besides, parallel datamining has got good performance. In sequential datamining, a surprising result was that the heterogeneous characteristics of nodes have overcome the network costs.

Future works will improve our proposal. One ongoing work [YAM 99] aims to propose a dynamic scheduling of distributed objects, which can be directly used in DHolo. Besides, optimizations over initial execution kernel are under development.

REFERENCES

[AIT 91] AIT-KACI, H.; *Warren's Abstract Machine – A Tutorial Reconstruction*. MIT Press, 1991.

[AMA 96] AMANDI, A.; PRICE, A. *A Linguagem OWB: Combinando Objetos e Lógica*. I Simpósio Brasileiro de Linguagens de Programação, p.305-318, 1996.

[AMB 96] AMBRIOLA, V.; CIGNONI, G. A.; SEMINI; L. *A Proposal to Merge Multiple Tuple Spaces, Object Orientation and Logic Programming*. Computer Languages, Elmsford, v.22, n.2/3, p.79-93, July/October 1996.

[APT 98] APT, R. et al. *Alma-0: An Imperative Language that Supports Declarative Programming*. ACM Transactions on Programming Languages and Systems, New York, v.20, September 1998.

[BAR 00] BARBOSA, J. L. V.; VARGAS, P. K.; GEYER, C.. *GRANLOG: An Integrated Granularity Analysis Model for Parallel Logic Programming*. Workshop on Parallelism and Implementation Technology (constraint) Logic Programming, London, 2000.

[BAR 01] BARBOSA, J. L. V.; GEYER, C. F. R. *Uma Linguagem Multiparadigma Orientada do Desenvolvimento de Software Distribuído*. V Simpósio Brasileiro de Linguagens de Programação (SBLP), maio 2001.

[BOS 96] BOSSCHERE, K.; TARAU, P. *Blackboard-based Extensions in Prolog*. Software – Practice and Experience, v.26, n.1, p.49-69, January 1996.

[BRO 90] BROGI, A. *AND-parallelism without shared variables*. Seventh International Conference on Logic Programming. MIT Press, p.306-324, 1990.

[BRO 91] BROGI, A.; CIANCARINI, P. *The Concurrent Language, Shared Prolog*. ACM Transaction on Programming Languages and Systems. New York, v.13, n.1, p.99-123, January 1991.

[CAR 89] CARRIERO, N.; GELERNTER, D. *Linda in context*. Communications of the ACM, v.32, n.4, p.444-458, 1989.

[CHI 01] CHIRICO, U. *JIProlog – Java Internet Prolog*. www.geocities.com/jiprolog, April 2001.

[CIA 96] CIAMPOLINI, A.; LAMMA, E.; STEFANELLI, C; MELLO, P. *Distributed Logic Objects*. Computer Languages, v.22, n.4, p.237-258, December 1996.

[CIP 01] CIANCARINI, P.; ROSSI, D. *JADA: A Coordination Toolkit for Java*. www.cs.unibo.it/~rossi/jada, April 2001.

[CIP 94] CIANCARINI, P. *Distributed Programming with Logic Tuple Spaces*. New Generating Computing, Berlin, v.12, n.3, p.251-283, 1994.

[HAJ 96] HARDWICK, J. *Java as an Intermediate Language*. School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-96-161, August 1996.

[HAN 94] HANUS, M. *The Integration of Functions into Logic Programming from Theory to Practice*. Journal of Logic Programming, New York, v.19/20, p.583-628, May/July 1994.

[HAR 98] HARIDI, S. et al. *Programming Languages for Distributed Applications*. New Generating Computing, v.16, n.3, p.223-261, 1998.

[HAR 99] HARIDI, S. et al. *Efficient Logic Variables for Distributed Computing*. ACM Transactions on Programming Languages and Systems, v. 21, n.3, p.569-626, May 1999.

[HEN 97] HENZ, M. *Objects in Oz*. Saarbrüchen: Universität des Saarlandes, May 1997. (PhD Thesis)

[IEE 98] *IEEE Transactions on Software Engineering*, v.24, n.5, May 1998. (Special Issue on Mobility)

[JAV 01] *Java*. http://www.sun.com/java, April 2001.

[LEE 97] LEE, J. H. M.; PUN, P. K. C. *Object Logic Integration: A Multiparadigm Design Methodology and a Programming Language*. Computer Languages, v.23, n.1, p.25-42, April 1997.

[MEH 95] MEH, M.; SCHEIDHAUER, R.; SCHULTE, C. *An Abstract Machine for OZ*. Seventh International Symposium on Programming Languages, Implementations, Logics and Programs (PLIP'95), Springer-Verlag, LNCS, September 1995.

[MUL 95] MULLER, M.; MULLER, T.; ROY, P. V. *Multiparadigm Programming in Oz*. Visions for the Future of Logic Programming: Laying the Foundations for a Modern Successor of Prolog, 1995.

[NGK 95] NG, K. W.; LUK, C. K. *I+: A Multiparadigm Language for Object-Oriented Declarative Programming*. Computer Languages, v.21, n.2, p. 81-100, July 1995.

[NII 86] NII, H. P. *Blackboard systems: the blackboard model of problem solving and the evolution of blackboard architectures*. AI Magazine, v.7, n.2, p.38-53, 1986.

[PIN 99] PINEDA, A.; HERMENEGILDO, M. *O'CIAO: An Object Oriented Programming Model Using CIAO Prolog*. Technical report CLIP 5/99.0 , Facultad de Informática, UMP, July 1999.

[PRO 99] *Proceedings of the IEEE*, v.87, n.3, march 1999. (Special Issue on Distributed DSM)

[PRO 01] *Programming Languages for the Java Virtual Machine*.http://grunge.cs.tu-berlin.de/~tolk/vmlangua ges .html (March 2001)

[ROY 97] ROY, P. V. et al. *Mobile Objects in Distributed Oz*. ACM Transactions on Programming Languages and Systems, v.19, n.5, p.804-851, September 1997.

[TAR 99] TARAU, P. *Jinni: Intelligent Mobile Agent Programming at the Intersection of Java and Prolog*. PAAM'9, The Practical Applications Company, 1999.

[VAR 00] VARGAS, P. K.; BARBOSA, J. L. V.; FERRARI, D.; GEYER, C. F. R.; CHASSIN, J. *Distributed OR Scheduling with Granularity Information*. XII Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho, SBC, 2000.

[VOY 01] *Voyager*. http://www.objectspace.com, April 2001.

[WEG 93] WEGNER, P. *Tradeoffs between Reasoning and Modeling*. In: Agha, G.; Wegner, P.; Yonezawa, A. (eds.). Research Direction in Concurrent Object-Oriented Programming. Mit Press, p.22-41, 1993.

[WEG 97] WEGNER, P. *Why interaction is more powerful than algorithms*. Communications of the ACM, v. 40, n. 5, p.80-91, May 1997.

[YAM 99] YAMIN, A. C. *An Execution Environment for Multiparadigm Models*. PPGC/UFRGS, 1999. (PHD Proposal)