Programming the Connection Machine by using the Metaobject Protocol

Lothar Hotz, Gerd Kamp

Universität Hamburg, Fachbereich Informatik,
Bodenstedtstrasse 16, 22765 Hamburg, Germany,
e-mail: hotz@informatik.uni-hamburg.de

In this paper, we demonstrate how the Metaobject Protocol of the Common Lisp Object System can be used to program the Connection Machine. In this approach, metaobject classes, which are responsible for internal representations of instances, map instances to processors in various ways. With these classes parallel structures are implemented to form a language which allows an easy usage of a parallel machine.

1. INTRODUCTION

A widely used approach to join object-oriented (O-O) programming and parallel computers consists in identifying an "object" as a main source of parallelism or concurrency. Objects (or actors, guardians) communicate by messaging, i.e. sending messages to each other. Such active objects may run concurrently if they reside on different processors. Examples for systems following this approach are described in [1–5]. Besides objects and messages O-O programming supplies additional features that may be used for programming parallel computers. These features (e.g. found in the Common Lisp Object System (CLOS) [6, 7]) are generic functions, multiple inheritance, method combination, and the metaobject protocol.

In this paper, we show how CLOS can be used to program massively parallel computers (MPCs) especially the Connection Machine (CM-2) [8]. Our goal is to hide any kind of explicit parallel syntax to the user by extending an already well-known language with less change of its syntax. If we can define a set of appropriate data structures, whose operations are internally parallelized, and whose user-interface is within the scope of the language, the user is not puzzled about explicitly thinking or programming in parallel terms. These data structures are manifested in a set of large (i.e. numerous elements) complex structures such as matrices, lists, sets, relations, and graphs with appropriate operations on them (e.g. multiplication, graph traversion, transitive hull, union). Thus, our goal is to design a language of complex data structures (called *DisGraL*, distributed graph language) and join these structures that are problem-oriented, machine-independent, commonly usable, and parallelizable with appropriate operations.

In a first approach, we concentrate on data parallel machines like the CM-2, because they have a nature different from control-oriented languages; detailed knowledge of the machine's architecture are necessary to program them [9]. Thus, complex structures and their operations are realized by data parallel algorithms. The interface between such abstract data structures and their data parallel realization is implemented via the metaobject protocol. This is a way to "open languages up", allowing users to adjust the

design and implementation of a programming language to suit their particular needs. In particular, in [7] a subset of CLOS is defined and extended with facets, dynamic slots etc. by using the metaobject protocol of CLOS (MOP). In a similar way, we extend PCL (Portable CommonLoops, a portable version of CLOS) to supply data mappings to the CM-2, i.e. known distributions of objects (e.g. [19, 20]) are supported by our extensions. Thus, we show how the MOP can be used to implement an interface, between CLOS and the CM-2. This interface is given by language features, which smoothly fit in the syntax of CLOS. Thus, no additional explizit parallel constructs must be known to a user of these extensions. In this sense, the paper describes how to bridge the gap between low level, machine-dependent, more or less efficient parallel algorithms and high level, abstract applications. We are not concerned with improving specific algorithms or solving a specific problem on a parallel machine in an efficient way. Thus, neither efficiency results are presented nor parallel algorithms are described. However, we assume that there exist efficient parallel algorithms and that there are applications that should be solved with parallel machines.

We first give a short overview of terms used (Section 2). After a short description of levels of abstraction (3) (for a more detailed view of them we refer to [10]), we present the developed metaobject classes for handling massively parallel constructs (4). These classes are used to implement complex structures as described and illustrated with a short example in Section 5. Finally, in Section 6 we discuss some other approaches.

2. THE CM-2, O-O PROGRAMMING, AND THE MOP

The connection machine CM-2 is a massively parallel single instruction multiple data (SIMD) computer consisting of up to 64K processors. To the user the parallel processing unit appears as an extension of the normal environment of a standard sequential computer, such as a Sun-4. *Lisp [11] is a data parallel extension of Common Lisp for the CM-2. The data parallel extensions of *Lisp include a large number of functions and macros, and one important abstraction, the "parallel variable", or "pvar". A pvar is a variable that has a separate value for each processor in the CM-2.

Besides widely used O-O features like classes, slots (data fields of classes), inheritance, and instances, the data stores of slot values of a specific class, we focus on metaobject classes, and protocols. Classes, which control representation of instances of classes are called "metaobject classes". With them a programmer gets access to the internal representation of language constructs. The specification of interacting objects is manifested in a collection of generic functions called "protocol". Protocols are specializable, modifiable interfaces between objects. CLOS is implemented in CLOS, i.e. all features of CLOS (e.g. classes, generic functions) are represented as instances of predefined classes named "standard metaobject classes". These instances are called "metaobjects". The behavior of these metaobjects (and thus of CLOS) is described with protocols. These protocols describe specific points of the CLOS implementation to be changed by a programmer. For example, the instance allocation protocol specifies that *allocate-instance* is called when an instance is allocated. Because these functions are generic functions they can be specialized as in an ordinary O-O program. The language implementation itself is structured as an O-O program. Thus, extensions can be made by subclassing standard metaobject

classes to "specialized metaobject classes". These classes extend or override predefined language behavior, e.g. to access a processor element and not an element of an array when a slot value is refered to. In Section 4, we describe the modification of the instance allocation protocol. Changing the implementation of a programming language might look ugly and suspect, but it is not when doing it in a portable way. Because the protocols are summarized in the quasi standard of the MOP (see [7, 12]), all implementations of CLOS using it will understand changes of CLOS. To the user (programmer) of a such modified CLOS the extensions are part of the language and clearly integrated in the syntax and semantics of the language.

3. LEVELS OF ABSTRACTION

To clarify our view of parallel languages, we describe levels of abstraction and languages (see also [10]). The usefulness of separating low-level parallel constructs from high level problem oriented constructs is also discussed in other papers like [13–16]. We extend *Lisp with PCL to get an O-O interface to the data parallel CM-2. Furthermore, we define classes and metaobject classes to map data to parallel machines. This level is called **CLOS*. With this extension, classes and operations for regular structures like *vector*, *matrix*, *set* and irregular structures are defined. Irregular structures such as *relation*, *graph*, *tree*, and *list* change dynamically and are not constant in size or are arbitrarily distributed. Thus, it is not possible to map these structures one-to-one to regular internal constructs. However, it is possible to implement such structures on the CM-2 [17, 18]. Operations on irregular structures are defined on the entire structure, not on single objects. Our approach integrates these structures and operations in a set of classes and methods that is called *DisGraL*. The main issue with *DisGraL* is the need to define problems in terms of complex structures and operations, i.e. the programmer does not think in parallel or sequential structures but in complex ones. *DisGraL* is the basis for a tool for implementing various Artificial Intelligence (AI) applications such as vision, neural nets, and the selection problem on parallel computers. In Section 5, we give one example, which gives a short insight of the tool.

4. CLASSES AND METAOBJECT CLASSES FOR THE CM-2

In this section, we present one class and three specialized metaobject classes, one of them is used to implement the complex structure *relation-class* (see Section 5). These classes describe distinct mappings of instances (i.e. data holding structures of O-O languages) to processors, reflecting our data parallel approach. They are part of **CLOS*. As mentioned above, the instance allocation protocol is specialized by these classes. The standard protocol specifies that *allocate-instance* results in an array for storing slot values. By indicating slots to be not CLOS slots but parallel ones, the standard protocol does no longer allocate storage on the sequential machine. Instead storage is allocated on the CM-2 by a specialized method for *allocate-instance*, which uses *LISP functions. The distribution can be done in various ways, depending on the desired data distribution.

pvar-per-instance-class This class is an ordinary CLOS class. Each instance of this class represents one pvar. The pvar is allocated during initialization of a new in-

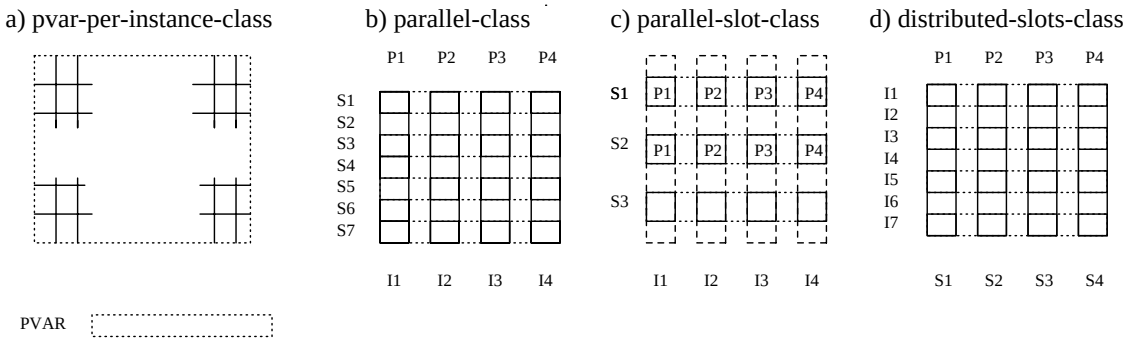

Figure 1. a) - d). Specialized parallel metaobject classes. Pi stands for Processors, Si for Slots, and Ii for Instances.

stance and initialized with a given content, e.g. elements of a sequential array or a particular number. Thus, the initialization protocol is extended to allocate the appropriate pvar depending on the given content. In Figure 1 a), a two dimensional pvar is shown which corresponds to one instance representing a matrix.

parallel-class This metaobject class maps each instance of a class to a processor, i.e. *all* slot values of an instance are stored on the *same* processor. In Figure 1 b) a column illustrates one instance by its slot values Si. Each instance Ii corresponds to one processor Pi. This data mapping is similar to that used in PARKA [19]. The difference of course is that PARKA is a direct implementation in *Lisp and uses no O-O programming.

parallel-slot-class The metaobject class *parallel-class* maps an instance as a whole to a processor. With *parallel-slot-class* particular slots (called *parallel slots*) of an instance are mapped to processors, while other slots are normal CLOS slots. Thus, one or more slots of an instance may belong to a pvar. This is indicated in Figure 1 c), slots $S1$, $S2$, and $S3$ are parallel slots. Pvars for parallel slots are allocated when a class is defined. By extending the instance structure protocol by the function *get-processor* that computes a processor for a given slot value, the distribution of data can be controlled. The default behavior maps each slot value to a distinct processor. Other kinds of data mappings can be implemented by specializing this function. Thus, *get-processor* is a parallel extension to the inherited, sequential protocol. With it, particular load balancing methods may be implemented.

distributed-slots-class Besides the vertical storing of instances done by *parallel-class*, they may be stored horizontally, i.e. each slot value of an instance is located in a *distinct* processor. This is useful for large instances with many slots (compare the implementation in [20]). Again as in *pvar-per-instance* one pvar is allocated for each instance and the desired slot values are stored in distinct processors. Accesses to slot values are implemented by specializing the instance structure protocol. Figure 1 d) illustrates this metaobject class.

Summarizing these proposed class definitions, parallel constructs are allocated during class definition and specific processors are referenced during instantiation of such a class. For this, the initilization protocol and the instance allocation protocol are specialized. To refer to the right slot values on the CM-2 additionally the instance structure protocol is modified. Thus, this modified method establishes a part of the interfaces between CLOS and the CM-2. All parallel constructs such as declaration and allocation of processors and pvars or usage of machine-dependent operations are hidden by metaobject classes and their specializations of MOP functions. Thus, these metaobject classes modify the internal representation of CLOS structures to connect CLOS with *Lisp's parallel functions.

After this mapping of data (instances) to processors the next step is to define operations on parallel structures. Besides extending functions of the MOP additional protocols are attached to complex structures. In Section 5, we describe a small part of them.

5. COMPLEX STRUCTURES

We see a complex structure as a kind of object that represents a large data structure whose elements are related in a specific kind (e.g. matrices, graphs, relations, lists). In this sense, complex structures are abstract data types given by our language and not defined by a user (i.e. an application programmer using a parallel language). The notion is, that specific (large) data structures are necessary to program a parallel machine, and should be part of a parallel language, because of difficult, possibly machine-dependent implementations of such structures. The selection of necessary complex structures is guided by AI applications. In this section, we describe one example of a complex structure: a class for handling relations (*relation-class*). Relations are used in knowledge representation for describing concepts and their connections.

The class *relation-class* is a direct subclass of *parallel-slot-class*, i.e. *relation-class* is an other specialized metaobject class (because *parallel-slot-class* is one). Thus, each parallel slot is associated with a parallel structure but additionally it is interpreted as a relation between classes as domain and range of the relation. Consider the following shortened definition of the class (or concept) *PC*:

```
(defclass PC ()
   ((has-parts :relation-values (CPU harddisk display)
               :allocation parallel))
   (:metaclass relation-class))
```

The parallel allocation specification *:allocation parallel* is inherited from *parallel-slot-class*. Instances of *PC* are organized by *relation-class*, which is indicated through *(:metaclass relation-class)*. *:relation-values* describes three new relation elements for the relation *has-parts*, i.e. between *PC* and *CPU*, *harddisk*, and *display* respectively. Defining a further class, say *comfortable-PC* with an additional part (e.g. *:relation-values (keyboard)*) extends the relation *has-parts* by four new relation elements: three inherited from *PC* and a new one. Thus, the parallel structure representing the parallel slot is extended, and no additional parallel structure is created. All other slots of *comfortable-PC* are inherited by CLOS inheritance and their relations are similarly extended, i.e. corresponding parallel structures are shared by diverse classes. After the classes are defined a parallel structure

contains all created relation elements of one relation.[1] Instances of a class are represented by markings of the class' relation elements, i.e. they describe subsets of the relations specified by the class. Markings are parallel structures, too, e.g. boolean-pvars.

Parallel operations on complex structures are defined by generic functions. We distinguish between simple generic functions on single elements of complex structures and complex generic functions on a whole complex structure. Examples of simple generic functions are comparing functions (like *max* and *min*) and element functions (like *position-of-element* and *apply-to-elements*, which applies a function to each element of a structure)[2]. Besides arithmetic functions (like *mult*, *add*, and *sub*) and creating functions (like *make-instance*, *shift-to-parallel-structure*) complex functions are e.g. focussing functions, generating functions, reduce functions, or structure specific functions as transitive hull and subsumption for relations, or clique graph computation for graphs. For a short description of some of these function classes we refer to [10]. These operations are defined by generic functions, i.e. they build a protocol for complex structures. For example, the subsumption is defined as simple intersection of instances, and implemented by comparing relation elements in parallel. This predefined behavior might be specialized for specific applications.

To demonstrate the usage of the described classes and functions we present a possible implementation for solving the selection problem. (We can only emphasize main issues, details are avoided.) Within this problem, a domain is represented by concepts and relations, which describe a large set of possible real world objects (here called "configurations"). We have already seen the concept *PC* with its relation *has-parts*. Imagining a number of such concepts representing a greater number of real PC's, we want to select a suitable, complete PC when a demand of a user is present. We assume, the demand is in the scope of the domain, i.e. it uses the same relations as our domain describes.

Our approach is to define relations and concepts by *relation-class* as described above. Thus, parallel structures for relations are allocated and the protocol for *relation-class* can be used. The demand is represented by an instance having all desired relations, which are marked on the desired relation elements. This instance is simply compared with the defined relations by using the subsumption function, which tests intersections, as described above. Thus, parallelism is achieved for relation elements described by one or (in the case of sharing parallel structures) of some relations. When all relations are checked, the computed markings determine the chosen concepts. They describe the configurations that subsume the demand. The presented representation is one of a multitude of alternatives.

6. RELATED WORK

Our approach combines diverse methods and constructs, i.e. O-O programming, high-level structures guided by AI applications, massively parallel machines, and an implementation technique made possible by a MOP. Thus, it is related to many other approaches, which usually concentrate on one or two of these terms.

As mentioned above O-O languages proposed for parallel machines are mostly related

[1] With this implementation one gets parallelism on one relation, to improve that, one can hold several relations in one parallel structure which is not discussed here.

[2] In [13] and [21] similar functions are proposed for applying to *parallel-sets* and *paralations*.

to distributed machines, where message-passing is mapped to communicating processors. Because the MOP specifies all kinds of O-O features (i.e. not only generic function invocation, which corresponds roughly to message-passing, but also instance storage allocation, inheritance and so on) it is possible to modify CLOS in specific places, and only there. Thus, it is not necessary to define a whole new language, but pointed modifications are possible. However, other O-O languages defined for massively parallel machines are introduced in [13] and [14]. But in [13] only elementwise functions are proposed and in [14] a sequential model lies on top, i.e. no new programming styles are supported, as with our complex structures. In [22] the message-passing language pC++ is presented, which introduces structures named "collections" that are similar to our complex structures. The collections selected there are mostly related to linear algebra. But, the approach of hiding low level implementations in class hierarchies[3] is the same as in *DisGraL*. The main differences of course are distinct implementation techniques. Because the MOP is part of CLOS, no additional programming environment (as described in [22]) is used. Instead of compiling *DisGraL* we use the MOP as an interface between O-O programming and the machine-dependent language *Lisp. Thus, we do not give a compiling scheme for, say, complex structures (as it is done for pC++) but take possibly high machine-dependent programs of a parallel language and use them to implement complex structures. By this means, existing parallel programs and specific implementation techniques are integrated.

Summarizing this, implementing the sequential language CLOS (as it is done by *Lisp (which is ordinary Common Lisp plus some parallel constructs) and our adding of PCL) and getting access to parallel machine internals (as it is done by *Lisp's parallel constructs) enables the implementation of high level abstractions in a known syntactic and semantic environment. The reflecting properties of CLOS (i.e. the possibility to manipulate language features within the language itself) enables incorporating parallel constructs by using a clean interface. This is also emphasized in [16].

7. CONCLUSION

We presented an O-O language for programming massively parallel machines. This language is implemented by using the Metaobject Protocol of CLOS. Instead of concentrating on message-passing, we use multiple features of CLOS, such as generic functions, multiple inheritance, and its special implementation with the MOP. We show that it is easy to implement an interface between CLOS and the CM-2 by using this open language implementation. Because we focus on this interface, traditional problems of parallel computing (e.g. load balancing, synchronization) are not discussed. Instead, we described some metaobject classes that produce various kinds of object (or instance) distribution. With them, we implement diverse abstract data types, which are part of our proposed language. With an example taken from AI we demonstrate the usage of the language. The extensions made, lead to a new easy understandable language, because new features are integrated precisely in the syntax and semantics of CLOS. Because the MOP is concerned with sequential operations, yet small additional protocols are fit in to support parallel features like load balancing. The integration of further parallel protocols (e.g. for handling sparse matrices) will be done in the future. To develop an appropriate tool, we

[3] In our case multiple inheritance leads to heterarchies.

will examine diverse areas of AI to find commonly usable structures and operations for programming parallel machines. If implemented appropriatly, these abstractions might be useful to program other machines like multiple instruction multiple data or distributed machines.

REFERENCES

1. P. C. Treleaven, editor. *Parallel Computers, Object Oriented, Functional, Logic.* Wiley & Sons, 1988.
2. A. Yonezawa and M. Tokoro, editors. *Object-Oriented Concurrent Programming.* MIT Press, Cambridge, MA, 1987.
3. C. Houck and G. Algha. HAL: A High-level Actor Language and Its Distributed Implementation. In *Proc. Int. Conf. on Parallel Processing '92*, pages II–158 – II–165, 1992.
4. E. Moss. Panel Discussion: Object-Oriented Concurrency. In *OOPSLA Addendum to the Proceedings*, volume 23 of *ACM SIGPLAN Notices*, pages 119 – 127, 1987.
5. L. V. Kale and S. Krishnan. Charm++: Portable Concurrent Object Oriented System Based On C++. Technical report, University of Illinois, 1991.
6. S. E. Keene. *Object-Oriented Programming in Common Lisp.* Addison-Wesley Publishing Company, 1989.
7. G. Kiczales, D. G. Bobrow, and J. des Rivières. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, 1991.
8. W. D. Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.
9. C. K. Yuen. *Parallel Lisp Systems.* Chapman & Hall, 1993.
10. L. Hotz. An Object-Oriented Approach for Programming the Connection Machine. In H. Kitano, editor, *Second International Workshop on Parallel Processing for Artificial Intelligence, PPAI'93.* Elsevier Science Publishers, 1993. To appear.
11. *Lisp. *Getting Started in *Lisp, Version 6.1.* Thinking Machines Corporation, Cambridge, MA, 1991.
12. R. P. Gabriel. Book Review: G. Kiczales, J. des Rivières, and D. G. Bobrow, The Art of the Metaobject Protocol. *Artificial Intelligence*, 61:331–342, 1993.
13. M. F. Kilian. Object-Oriented Programming for Massively Parallel Machines. In *Proc. Int. Conf. on Parallel Processing '91*, pages II–227 – II–230, 1991.
14. J.-M. Jézéquel. EPEE: an Eiffel Environment to Program Distributed Memory Parallel Computers. In *Proc. ECOOP '92*, pages 197–212, 1992.
15. G. C. Fox. Hardware and Software Architectures for Irregular Problem Architectures. In R. Voigt, P. Mehrotra, and J. Saltz, editors, *Unstructured Scientific Computation on Scalable Multiprocessors*, pages 125–160. The MIT Press, 1992.
16. H. Masuhara, S. Matsuoka T Watanabe, and A. Yoneyawa. Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently. In *Proc. OOPSLA '92*, ACM SIGPLAN, pages 127 – 144, 1992.
17. D. Dahl. Mapping and Compiled Communication on the Connection Machine System. In *Proc. of the 5th Distributed Memory Computing Conference IEEE Computer Society*, pages 756–766, Charleston, South Carolina, April 1990.
18. S. S. Nielsen and S. A. Zenios. Data Structures for Network Algorithms on Massivley Parallel Architectures. *Parallel Computing*, 18:1033–1052, 1992.
19. M. Evett, J. Hendler, and L. Spector. Parka: Parallel Knowledge Representation on the Connection Machine. Technical Report UMIACS-TR-90-22, University of Maryland, 1990.
20. J. L. Kolodner and R. Thau. Design and Implementation of a Case Memory. Technical Report RL88-1, Georgia Institute of Technology, 1988.
21. G. Sabot. *The Paralation Model.* MIT Press, Cambridge, MA, 1988.
22. J. K. Lee, S. Yang, S. Narayana, and D. Gannon. Compiling an Object-Oriented Parallel Language for Parallel Machines. In *Proc. of 1992 International Conference on Parallel and Distributed Systems*, pages 314 – 321, Hsinchu, Taiwan, December 1992.