Multi-author books prepared in LaTeX

Parallel Processing for Artificial Intelligence
L. Kanal, V. Kumar, H. Kitano, and C. Suttner, editors

An Object-Oriented Approach for Programming the Connection Machine

Lothar Hotz

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

Abstract

In this paper, we describe an approach using object-oriented programming for massively parallel computers. We show how to integrate high level structures and operations to support various applications in the area of Artificial Intelligence.

1. Programming Model

At present, programming massively parallel computers (MPC) demands much experience and a detailed knowledge of the machine's architecture. Especially data parallel machines like the Connection Machine (CM-2) [9, 35] and their programming languages have a nature different from control-oriented languages [37]. 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 operations. *DisGraL* establishes an abstraction of parallel constructs, as they are introduced in parallel languages like POOL [2], *LISP [23], etc. This is done by focussing on large data structures (e.g. matrices, sets and pointer structures, such as lists, trees, and graphs) and operations on them (e.g. multiplication, graph traversion, spanning trees, union). With these constructs the user (programmer) of *DisGraL* need not think or program in parallel terms.

For the design of *DisGraL*, we use object-oriented (O-O) programming, with its proved properties of reusability, extensibility, robustness, and abstraction [25, 17]. To integrate the parallel structures mentioned above, we extend the notion of an object by the notion of a complex structure. An

object might have such a predefined complex structure like a matrix or a graph. The main point is that the structure that is formed by a multitude of objects is a supported feature of *DisGraL* as well as objects are. Complex structures are first-class objects of the language and are processed with parallel algorithms. Thus, data parallelism as described in [18, 14], instead of control-oriented parallelism as described in [34, 36, 2, 33, 12, 26, 15] is supported. We do see distribution of instances of a class, which manifest data stores of an O-O language, as the main source of parallelism, but not message-passing, which would lead to actor languages like HAL [12]. The global view on objects (as manifested in complex structures) makes it possible to hide low-level, parallel constructs in complex operations.

The main point of our approach is to bring together basic operations which are used in implementations of Artificial Intelligence methods or more general, in data parallel algorithms as they are described e.g. in [32, 5, 19]. A good example is the function *reduce* (described below). Shortly speaking, this function multiplies a matrix with itself, by using problem-dependent combining and joining functions instead of multiplication and addition. Thus, it can be used e.g. to compute path consistency of a set of time intervals (as described in [1]) or to compute all shortest paths of a graph given by an adjacency matrix (as described in [29]). Another example is the data structure *graph* used by the function *generate*. Both are used to explore a domain-dependent graph (generated by scalable generation functions), e.g. for various search problems.

Due to the underlying implementation of *DisGraL*, these data structures and operations are processed in parallel. An important point of our implementation is the use of the Metaobject Protocol of the Common Lisp Object System (CLOS) [16, 17] combined with parallel algorithms for implementing operations (e.g. [6, 10, 3]). [17] describes a way to "open languages up", allowing users to adjust the design and implementation of a programming language to suit their particular needs. This is achieved by structuring the language implementation itself as an O-O program via *metaobject protocols*. The adjustment of classes and generic functions is done by specializing given classes (named *metaobject classes*). In our approach, class metaobject classes map instances of complex structures in a suitable way to the processors, e.g. by using special clustering methods (e.g.[4]). Operations, such as diverse search methods for implementing *generate*, are implemented by generic function metaobject classes, which use given parallel algorithms (e.g. [21, 30, 22]).

Other approaches (e.g. [18, 14]) also introduce O-O languages for programming MPC, but do not support complex data structures and operations for them. In [18] only elementwise functions (i.e. constructs that apply functions to each element of a complex structure) no functions on

complex structures as a whole are given. In [14] a sequential model lies on top, i.e. no new programming styles are supported, as in our case complex structures are. However, the usefulness of separating low parallel constructs from high level problem-oriented constructs is also mentioned in [18, 14, 6, 24].

2. A more detailed view

To clarify our approach, we describe levels of different languages. We illustrate these levels with a piece of code taken from a computer vision program, which filters a picture (a float matrix) by using the iterative Jacobi method (see [7] for a detailed description of implementing such algorithms). We describe several levels of abstraction: Machine-dependent languages, O-O extensions, classes and methods for complex structures, and the application level.

The lowest level is built by languages like *LISP, C*, CRAY C, which mainly support regular structures like vectors, and matrices, i.e. the elements of a complex structure are regularly distributed over the processors. In *LISP a parallel variable (pvar) roughly represents these structures. Some languages supply operations like *multiplication* (FORTRAN 90) or *scan* (*LISP) to process these structures. Consider following *LISP Code:

```
(*defun compute-right-side (picture-pvar)
  (declare (type single-float-pvar picture-pvar))
  (*set picture-pvar
        (/!!
          (+!! (news!! picture-pvar 1 0)
               (news!! picture-pvar 0 1))
          (!! 6.0))))
```

picture-pvar is declared as a single float pvar. Functions postfixed with "!!" or prefixed with "*" are *LISP functions. *news!!* shifts a matrix according to a given distance, e.g. the result of the first call to *news!!* is a pvar with elements shifted one point in the first dimension (this is similar to *cshift* in FORTRAN 90). The *LISP functions are specified for typed pvars, which are handled as a datatype like integer or float. Thus, integers (like 6.0) have to be converted to pvars using function *!!*. **set* is used to change the values in the processors. The resulting code is CM-2 specific.

By defining operations on matrices, languages like FORTRAN 90 already introduce an abstract data type[1], but can't go further because no language mechanism is given to integrate this feature in the language. With an O-O language one gets a base for a unique view to different data types.

[1] In [6] a code reduction between two and three is mentioned when FORTRAN 90 instead of FORTRAN 77 is used.

Thus, the next level is the incorporation of O-O features such as classes, inheritance, instances, generic functions, methods, method combination, and metaobjects. We do this by extending *LISP with PCL to get an O-O interface to the data parallel CM-2. With this extension one can define classes that map data structures to the parallel machine and specify methods that implement parallel algorithms for operations on these data structures. Especially metaobject classes, which may be defined in PCL, organize the internal representation of instances of a class and are used to manifest this level. To implement the complex structures proposed in the next level we use specialized metaobject classes that supply different kinds of data mapping e.g. storing instances in different processors or distributing slots of one instance to different processors (see [11]).

The first easy classes and methods are those defined on regular data structures like vectors and matrices to integrate parallel languages in the O-O approach. Besides arithmetic operations, we introduce structure manipulating operations like translation or bisection of a matrix. In our example, the above mentioned function is now defined as a method for the application class *picture-class*, which is a subclass of the predefined class *matrix*:

```
(defmethod compute-right-side ((the-picture picture-class))
   (div
     (add (shift the-picture '(1 0))
          (shift the-picture '(0 1)))
     6.0))
```

This code is similar to the first piece of code except for the following distinctions:
a) *the-picture* is no more a pvar (i.e. a *LISP dependent structure) but an instance of a class specified by subclassing a predefined class (here *matrix* as mentioned above). Thus, the application programmer might define its own methods and subclasses for it. But more important, she can use methods defined for the class *matrix* (e.g. *div*, *add*, and *shift*). This is achieved by using the O-O feature of inheritance.
b) The *LISP specific "!!"-functions are replaced by generic functions (here by *add*, *div*, and *shift*), which are part of the class *matrix*, and can handle any combination of numbers and matrices as arguments. The main point is, that the functions are not only replaced, but useful operations on matrices are implemented by *LISP functions by using the O-O feature of generic functions. Especially the possibility of defining generic functions on multiple arguments, not only on one class as in other O-O languages, makes it easy to implement different methods for different kinds of argument combinations.
c) All parallel managment functions, such as declarations, allocation of

processors, and setting of processors (in the previous example done with **set*) are done by the underlying implementation of predefined methods.

In this easy case, one-to-one mappings from operations to *LISP functions are used. The next step is to define useful operations on matrices and other complex structures e.g.:

```
(div (join-elements picture
                    :pattern  '((1 0) (0 1))
                    :with     #'add)
     6.0)
```

The user (application programmer) of operations like *join-elements* does no longer think in "low-level" constructs like *news!!* or *shift* but can think in application terms like "At each point of my picture I want to join specific neighbours by using the function *add*". Thus, she takes *join-elements*, selects a pattern, and gets a joined result.

A next sort of classes describes irregular data structures such as *relation*, *graph*, *tree*, or *list*. They are not constant in size and may be arbitrarily distributed. Thus, it is not possible to map theses structures one-to-one to regular internal constructs. However, it is easy to implement such structures on the CM-2 [31, 4, 27, 20, 8]. 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 distributed graph language (*DisGraL*).

The last level is the application level, which is implemented by using complex structures, not parallel or sequential operations. Thus, the previous level must supply suitable complex structures to make an easy implementation of applications possible. Typical applications are those already realized for parallel computers e.g. implementation of libraries for linear algebra operations [7], knowledge representation [5], case-based reasoning [32], search algorithms [22, 13, 30, 28].

The main issue with the proposed abstractions 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. In other languages these constructs are supported by libraries as in [25], but are not part of the language itself if supported at all. Thus, porting of *DisGraL* to another machine (like MIMD, distributed machines) would involve the porting of complex structures.

Next, we describe how instances, the datastores of an O-O language, are mapped to processors of a data parallel machine [11]. Instances of regular classes correspond to data structures of the low level implementation language *LISP, i.e. parallel variables (pvars). An instance of an irregular class (i.e. a class describing a complex structure) consists of instances

which are part of this complex structure. Such an instance might correspond to one processor, i.e. all slot values of one instance are stored in the same processor.[2] Thus, these instances can be processed in parallel - a vertical distribution of instances. A third alternative to parallelize instances is given by *parallel-slots*, i.e. one or more slots of a class belong to a complex structure. In this case, not the whole instance, but only slot values of *parallel-slots* are stored in parallel processors. Thus, these parts of instances are processed in parallel. A further alternative is the distribution of each slot value of one instance to another processor, i.e. a horizontal distribution of instances (see [19]). For all kinds of such alternatives, classes or metaobject classes are defined in *DisGraL*, which distribute instances and slot values in the suitable way.

For these classes, operations are defined by generic functions, i.e. a collection of distinct methods which implement parallel algorithms with respect to the types (classes) of their arguments. 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)[3]. 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: *apply-shadowed-function* hides elements of a complex structure by a domain-dependent predicate *:hide-predicate*. Thus, only active elements (i.e. elements for which the predicate is *true*) are applied to a function in parallel.

Generating functions: *generate* creates a dynamic structure (e.g. a graph). The graph is defined by a root, a successor function, and a comparison predicate. *generate* uses the successor function to expand the graph until a given goal node is reached.

Reduce function: *reduce* combines fixed elements of a complex structure and changes them. It computes new values for each element by combining specific elements. Thus, it takes two domain-dependent functions as arguments: a *:combine-function* for combining elements and a *:join-function* for joining the new value with the old value of the element. For vectors *reduce* corresponds to the *LISP function

[2] The mentioned mapping to processors supports the programmers model of instances but may be changed by the implementation e.g. by clustering graph nodes when necessary.

[3] In [18] and [30] similar functions are proposed for applying to *parallel-sets* and *paralations*.

scan. In the two-dimensional case *reduce* corresponds to matrix multiplication with given functions, i.e. each column is combined with each row and the cross point of both is joined. A graph combines each neighbour of a node and joins the result with the old value of a node.

Functions that are given as arguments to the mentioned complex functions are sequential functions, i.e. ordinary functions implemented in LISP. But, they are applied to all active elements of a complex structure in parallel. Thus, our implementation compiles them into parallel versions. This is done by a generic function metaobject class that uses knowledge concerning the specific use of the function, e.g. a generating function for graphs.

3. Conclusion

We described a new combination of the parallel language *LISP with the object-oriented language CLOS for the Connection Machine.[4] Furthermore, the developed system showed that the Metaobject Protocol is a powerful method to change the predefined behavior of an O-O language for parallel implementations. We propose to define a language for large data structures instead of programming with explizit parallel language constructs. Classes and methods were implemented to support the representation of some small examples. To develop an appropriate tool for implementing various AI applications such as case-based reasoning, low-level vision, neural nets, or for the selection problem, we will examine these areas to find common usable structures and operations.

References

[1] J. F. Allen. Temporal reasoning and planning. In J. F. Allen, H. A. Kautz, R.N. Pelavin, and J.D. Tenenberg, editors, *Reasoning about Plans*, chapter 1, pages 1–67. Morgan Kaufmann, San Jose, CA, July 1991.

[2] J. K. Annot and P. A. M. den Haan. POOL and DOOM: The object oriented Approach. In P. C. Treleaven, editor, *Parallel Computers, Object Oriented, Functional, Logic*, pages 47–79. Wiley & Sons, 1988.

[3] L. E. Cannon. *A cellular computer to implement the Kalman filter algorithm.* PhD thesis, Montana State Univ., 1969.

[4] 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.

[4] At least to our knowledge.

[5] M. Evett and J. Hendler. Achieving Computationally Effective Knowledge Representation via Massively Parallel Lisp Implementation. In *Proc. Europal 90*, pages 1–13, 1990.

[6] 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.

[7] T. L. Freeman and C. Phillips. *Parallel Numerical Algorithms*. Prentice Hall, 1992.

[8] S. W. Hammond and R. Schreiber. Mapping Unstructured Grid Problems to the Connection Machine. In R. Voigt, P. Mehrotra, and J. Saltz, editors, *Unstructured Scientific Computation on Scalable Multiprocessors*, pages 11–29. The MIT Press, 1992.

[9] W. D. Hillis. *The Connection Machine*. MIT Press, Cambridge, MA, 1985.

[10] W. D. Hillis and JR. G. L. Steele. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[11] L. Hotz. Programming the Connection Machine by using the Metaobject Protocol. In G. R. Joubert, D. Tystram, and F. J. Peters, editors, *ParCo'93: Conference on Parallel Computing, Proc. of the International Conference, Grenoble, France*. Elsevier Science Publishers, 1993. To appear.

[12] 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.

[13] S. Huang and L. S. Davis. Parallel Iterative A* Search: An Admissible Distributed Heuristic Search Algorithm. In *Proc. of the Int. Joint Conf. on Artificial Intelligence '89*, pages 23–29, 1989.

[14] J.-M. Jézéquel. EPEE: an Eiffel Environment to Program Distributed Memory Parallel Computers. In *Proc. ECOOP '92*, pages 197–212, 1992.

[15] L. V. Kale and S. Krishnan. Charm++: Portable Concurrent Object Oriented System Based On C++. Technical report, University of Illinois, 1991.

[16] S. E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley Publishing Company, 1989.

[17] G. Kiczales, D. G. Bobrow, and J. des Rivières. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[18] M. F. Kilian. Object-Oriented Programming for Massively Parallel Machines. In *Proc. Int. Conf. on Parallel Processing '91*, pages II–227 – II–230, 1991.

[19] J. L. Kolodner and R. Thau. Design and Implementation of a Case Memory. Technical Report RL88-1, Georgia Institute of Technology, 1988.

[20] S. G. Kratzer. Massively Parallel Sparse-Matrix Computations. In R. Voigt, P. Mehrotra, and J. Saltz, editors, *Unstructured Scientific Computation on Scalable Multiprocessors*, pages 179–186. The MIT Press, 1992.

[21] W. Lau and V. Singh. An Object-Oriented Class Library for Scalable Parallel Heuristic Search. In *Proc. ECOOP '92*, pages 252–267, 1992.

[22] G. Li and B. W. Wah. Parallel Iterative Refining A* Search. In *Proc. Int.*

Conf. on Parallel Processing '91, pages II–608 – II–615, 1991.

[23] *Lisp. *Getting Started in *Lisp, Version 6.1.* Thinking Machines Corporation, Cambridge, MA, 1991.

[24] 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.

[25] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[26] E. Moss. Panel Discussion: Object-Oriented Concurrency. In *OOPSLA Addendum to the Proceedings*, volume 23 of *ACM SIGPLAN Notices*, pages 119 – 127, 1987.

[27] S. S. Nielsen and S. A. Zenios. Data Structures for Network Algorithms on Massivley Parallel Architectures. *Parallel Computing*, 18:1033–1052, 1992.

[28] C. Powley, C. Ferguson, and R. E. Korf. Depth-first heuristic search on a Simd machine. *Artificial Intelligence*, 60:199–242, 1993.

[29] G. Rote. Path Problems in Graphs. *Computing*, 7:159–189, 1990.

[30] G. Sabot. *The Paralation Model.* MIT Press, Cambridge, MA, 1988.

[31] J. A. Solworth. Programming Language Constructs for Highly Parallel Operations on Lists. *The Journal of Supercomputing*, 2:331–347, 1988.

[32] C. Stanfill and David Waltz. Toward Memory-based Reasoning. *Communications of the ACM*, 29(12):1213–1227, December 1986.

[33] K. Takashio and M. Tokoro. Drol: An Object-Oriented Programming Language for Distributed Real-Time Systems. In *Proc. OOPSLA '92*, ACM SIGPLAN, pages 276 – 294, 1992.

[34] P. C. Treleaven, editor. *Parallel Computers, Object Oriented, Functional, Logic.* Wiley & Sons, 1988.

[35] L. W. Tucker and G. G. Robertson. Architecture and Applications of the connection machine. *Computer*, pages 26–38, August 1988.

[36] A. Yonezawa and M. Tokoro, editors. *Object-Oriented Concurrent Programming.* MIT Press, Cambridge, MA, 1987.

[37] C. K. Yuen. *Parallel Lisp Systems.* Chapman & Hall, 1993.