Parallel Abstractions - Structure-Oriented Programming

Lothar Hotz
Labor für Künstliche Intelligenz
Fachbereich Informatik
Universität Hamburg

Abstract *In this paper, we describe parallel abstractions for implementing parallel AI methods at a highly abstract level. A programming model is introduced which replaces explicit parallel programming with a structure-oriented programming approach. In this approach control abstractions are introduced which intern are used to implement applications. For demonstrating the applicability of the programming model an example from the field of constraint propagation is presented.*

Keywords: Parallel Programming Models, Parallel Data Types, Object-Oriented Technology, Artificial Intelligence

1 Parallel programming using parallel abstractions — the programming model

Parallel programming is a difficult task, because of the possibly big number of parallel flows of control. In low-level parallel languages the handling of these flows (i.e. distribution, synchronization etc.) is left to the programmer. This makes parallel programming difficult and complex. To make parallel programming easier, we introduce a layer between low-level constructs and the application programming level - the structure-oriented programming level (see Figure 1). The main task of this level is the abstraction of parallel flows of control.

The possibilities for parallelizing multiple flows of control are directly related to the structure of the data to be processed. If flows operate on same data, they have to be synchronized. These flows depend on each other. If the data being processed is not stored in a common data structure, i.e. if the data is unstructured, the dependency structure is difficult to handle. Manual parallelizing or more or less automatic code analysis have to be applied. If the data are structured, the flows can be bundled into operations. If those operations are defined on data as a whole, synchronization can be done within these operations. Thus, operations on structures do have structure dependent parallelization facilities. That means, if the data has a certain structure certain flows of control can be used to operate on that structure. Thus, structures and operations on them abstract parallel flows, i.e. are parallel abstractions. Parallel abstractions representing complex structures and operations on them are introduced on the structure-oriented level, examples are an arbitrary net, a tree or graph structure.

With this approach parallel data types of various kind are introduced for implementing applications. Parallel data types used in languages like in Fortran90, NESL [2], or Paralation Lisp [16] are mainly based on parallel arrays. In those approaches other data types like trees and graphs have to be mapped to parallel arrays in a more or less awkward way. Thus, introducing further parallel data types besides parallel arrays is obvious.

For pragmatically determining what parallel abstractions are useful for implementing ap-

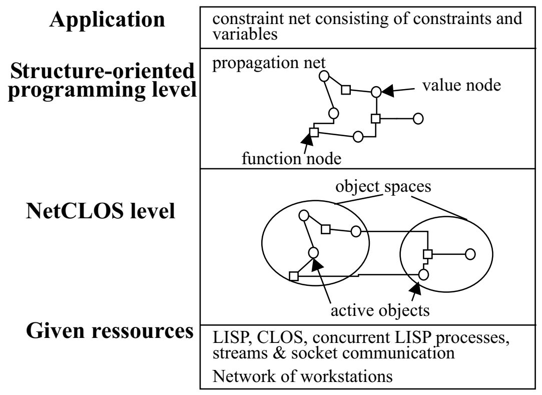
Application
constraint net consisting of constraints and variables
Structure-oriented programming level
propagation net
value node
function node
NetCLOS level
object spaces
active objects
Given ressources
LISP, CLOS, concurrent LISP processes, streams & socket communication
Network of workstations

Figure 1: Levels of used abstractions.

plications, we consider a discipline where parallel processing is not common but can have substantial contribution for improving performance: Artificial Intelligence (AI). Thus, the structure-oriented level (upper level) is intended for easy use by AI programmers inexperienced with parallel programming, while the lower level is intended for implementing the upper level. This implementation can be done by parallel or *structure programmers* experienced with parallel programming.

The lower level, an integration of an actor-like language [5] in Common Lisp and its object-oriented part CLOS (Common Lisp Object System) [13, 19] is shortly described in Section 2, for a more detailed view see [11]. In Section 3, a schema of parallel abstractions for programming AI applications is introduced. In two examples, structures are used to implement constraint nets and qualitative simulations (Section 4). In Section 5, related work is discussed.

2 Short description of the lower level — NetCLOS

The lower level (see [11]), called NetCLOS[1], is used to implement the structures of the upper level on a workstation cluster. It extends Common Lisp with features for parallel and distributed object-oriented programming. The parallel and the distribution aspect of the implementation of a structure can be described independently. The parallel programming is done with active objects. These have their own processes and communicate via synchronous and asynchronous message passing (for a similar model see [1, 22]).

The distribution model of NetCLOS is based on the notion of one *object space* (implemented with a lisp image) residing on each workstation of a cluster. An object space contains all information to handle active objects, e.g., all necessary classes and functions are known to each object space. Every active object resides on exactly one object space. But an active object can be referenced from each object space, not only the local one. Thus, the identity of an active object is guaranteed over all object spaces, i.e. each active object is unique and can be referenced from diverse object spaces. When messages are sent or an active object is passed as argument of a message, it makes no difference whether the object is locally or remotely referenced. To distribute these active objects over a workstation cluster, active objects can be created on every workstation and moved from one workstation to the other. A runtime environment enables distributed garbage collection and transparent remote message passing.

This way, we can divide the implementation of the structure types into two steps: A machine-independent description of the potential parallelism using active objects; and a description of the mapping of these active objects to the workstation cluster.

In the current implementation of NetCLOS, we focus on a workstation cluster. We think for other parallel architectures other parallel extensions may be useful. One example is *Lisp [14] which introduces parallel mechanisms for array processors. Its integration with CLOS (named *CLOS) is described in [8]. However, the structure-oriented programming level would be implemented with *CLOS or NetCLOS depending on the underlying architecture, but the upper level does not change for application programmers. Thus, a porta-

[1]NetCLOS is implemented as an extension to Allegro Common Lisp by using the metaobject protocol of CLOS (see [9, 13]).

bility of application programms written with structures is ensured.

With NetCLOS, basic concurrent and distributed abstractions are introduced in CLOS. The structure-oriented level is oriented towards application programmers, and aims at defining a high level parallel programming language. Furthermore, both levels are implemented as open extendable protocols by using diverse objects for handling the concurrency mechanisms (like message-handler, object spaces etc.). Thus, we follow a reflective approach for object-based parallel programming (see [3] for further classifications).

3 Introducing parallel abstractions for programming AI applications

Consider the control abstractions described in Figure 2. A program with a single flow is a sequential program. Programs with multiple flows can be of different types - independent or dependent flows. The flows are independent of each other if there are no common used data. Furthermore, the relations between data are expressed in structures, i.e. are explicitly given by operations, e.g. *parent* functions in trees. An arbitrary access of elements, e.g. array-elements via indices, is not seen as an explicit relation because it is not manifested in a structure. Thus, self-destructive operations on e.g. arrays are only possible in operations defined for the control abstraction *array*, but cannot be done by application programmers.

We distinguish three types of dependency structures. A *fixed* dependency structure is known at developers time[2] and thus, can be expressed by static data structures. A *derived* dependency structure is not known at developers time but functions can be given which are used to compute the dependency structure. This computation is done before the parallel processing takes place. If the dependency structure is computed during the parallel processing we speak of a *dynamic* dependency structure.

[2]i.e. the time when the application program is written down.

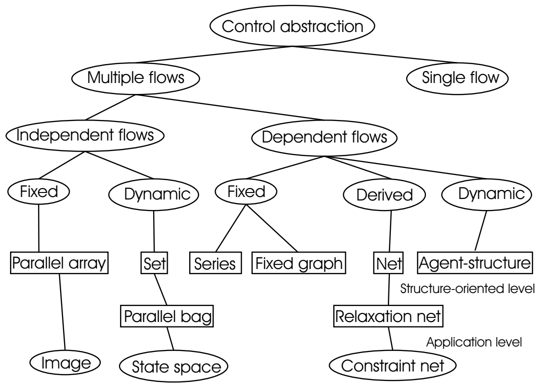

Figure 2: Control abstractions and their integration in application classes.

Each dependency structure demands a distinct load balancing and synchronization strategy. With this scheme not concurrency and distribution mechanisms are described with classes but structure of data. Concurrency mechanisms are introduced for classes of the structure-oriented level and may be totally different for distinct classes. This is no drawback, because experiences show that a structure of synchronization classes is difficult to specify and moreover to reuse, because of the high interdependency between the synchronization conditions for different methods [3].

For programming, complex structure classes are introduced, which belong to specific dependency structures. For image processing e.g., each pixel can be computed independently and the size and type of the structure is known in advance. Therefore, a *parallel array* is used to implement an image. For constraint filtering e.g., the structure of a constraint net is not known at developers time but can be computed before parallel processing. Thus, the parallel abstraction used for a constraint net, i.e. a *relaxation net*, has a derived dependency structure.

A further example is given by the abstraction *parallel bag*. A typical communication scheme for computing a best search is a server-client algorithm. A server distributes the work to be done to several clients at hand. A direct implementation with parallel constructs would mix parallel constructs with application specific functions. Thus, it would only be useful for the specific problem of computing best search. An abstraction of the server-client communication scheme would enable multiple applications to make use of it. The abstraction *parallel bag* hides the necessary communication scheme (a NetCLOS program) to an application programmer. The view at a parallel bag is as follows: A bag is present, where balls can be put into. A ball contains an object, some arguments and a function-name. When a ball is put into the bag, the ball is evaluated by the bag. The bag will call the function with the object and the arguments and put the result in the ball. If the evaluation of the ball is completed, the ball jumps out of the bag and the programmer can catch it. If multiple balls are put into the bag, the parallel bag probably will evaluate them in parallel. Thus, for an application programmer following functions are given: `create-bag`, `create-ball`, `put-ball`, `get-next-result`. The selection of the best ball have to be done in this scheme by the application programmer. The abstraction handles the distribution of balls on the parallel machine and calls the function of each ball for computing the result. Furthermore, when the structure programmer implements the abstraction, he/she can be sure, that the objects to be parallelized (the balls) are independent of each other.

4 Using parallel abstractions for implementing a constraint net and qualitative simulations

We tested our programming model by implementing a parallel abstraction named *relaxation-net* with NetCLOS. *Relaxation net* is an abstraction implementing parallel discrete relaxation (see [7] for a similar approach). It consists mainly of

- a class of active objects named *value nodes* acting as shared stores. Accesses to these stores are automatically synchronized, i.e. this is done by the NetCLOS level (implemented by the structure programmer). These active objects can be used to implement the variables of a constraint net, which is realized by the application programmer.
- a class of active objects named *function nodes* which, when activated, compute a function of the content of a set of stores. These active objects can be used to implement constraints.
- a structure class which organizes stores and functional objects into a network and provides for iterated activation and parallel execution of the functional objects (i.e. a relaxation operation). This *relaxation net* can be used to implement a constraint net.

To distribute the *relaxation net*, function and value nodes are modeled as tasks of a Task Interaction Graph. To distribute this graph on a workstation cluster we use a combination of bisection [17] and the Kernighan-Lin algorithm [12]. The distribution is realized by the library CHACO [4]. The use of such a library reflects the fact, that results coming from distributed systems are not reimplemented for our needs, but are used and connected to our environment when necessary.

The main operation on relaxation nets is the function *relax*, which computes a fix point. This function can be processed in parallel if the domain of the function can be partitioned in parts and the function itself can be partitioned in independent component functions (see [7, 21] for details). To use the parallel abstraction *relaxation net*, the application programmer implements subclasses of the value and function node classes and the structure class *relaxation net* and redefines some methods, i.e. implements a normal object-oriented sequential interface. There is no need for any explicit parallel programming, only the sequential interface of the parallel abstraction must be known to an application programmer.

Thus, the relaxation-net contains a net-like reference structure of active objects and a fix point operation on that structure. The net is distributed on a workstation cluster by the abstraction and the operation is executed in parallel on diverse parts of the net, i.e., distribution and parallel processing is done by the abstraction. This abstraction is used for implementing a local propagation algorithm for constraint nets by using *relax* [20].

To get a gain through parallel processing, one has to take high communication costs into account which are related to the infrastructure of a workstation net, e.g., an ethernet or TCP/IP. Thus, as usual in such a case, the computation time on one host should be high enough to compensate the communication costs. This is also the result of experiments we made. When solving a line-diagram labeling problem [15], we only got a speed-up for constraint propagation when raising the number of constraints. For n=1000 stairs we got a speed-up of 3.6 for constraint propagation on 7 machines. The number of constraints is $6n + 7$ and of variables is $4n + 7$. The speed-up strongly depends on the communication traffic on the ethernet and on the kind of workstations used, which are typically heterogeneous (e.g. from Sparc Classics to Ultra Sparcs). The distribution strategy does not yet consider such kind of information. Because of using a derived not a dynamic structure the constraint net is first created on one object space and than distributed to the other. This still yields to high distribution costs. However, the experiments show that one can get a speed-up for constraint propagation, when using parallel abstractions and NetCLOS.

In another example, we examined qualitative simulations of electric circuits of fork-lift trucks. For a diagnosing task of such a technical system our conception requires the simulation of all single and some selected multiple faults of technical systems. The simulation is done qualitatively because faults like "lossy wire" cannot expressed quantitatively [10]. However, the necessary simulations can be considered independently and therefore are easy to parallelize. But alternative distributions of the task are possible. For example, each simulation of a fault can be seen as one task, or all simulations of one component (like a resistor or a wire) can be seen as one task. With an parallel abstraction describing such alternatives we could easily evaluate them and got a quite useful speed-up of 3.4 when using 4 workstations and a component wise distribution.

5 Related work

Another approach which introduces data parallel abstractions is CMLisp [6]. This showed that programming with abstractions can simplify parallel programming, but CMLisp is restricted to run on single instruction multiple data machines (i.e. the Connection Machine 2) and thus, is hardly usable for workstation clusters. This is similar to [7], where a relaxation operation is introduced to solve constraint problems, but the implementation is done on a Sequent Symmetrie, not on a more common workstation cluster.

Formal approachs on the topic of parallel data types are based on Bird-Meertens Formalism [18] or skeleton languages [23]. In [18] for so called *categorical data types* a basic set of higher order functions (such as *map*, *filter*, *reduct*) is defined, which can be mapped to distinct parallel architectures. However, our approach gives a pragmatic way to implement parallel data types in an object-oriented environment and show their usage. How those parallel data types are related to categorical data types have to be examined in further research.

6 Conclusion

A high level programming model based on abstractions for parallelizing is described. The main point of this structure-oriented model is to introduce control abstractions, because multiple flows of control make parallel programming difficult. These control or parallel abstractions are realized by giving diverse predefined classes (like parallel-array, net, series) to the application programmer. These classes hide specific synchronization and load balancing schemes. With these parallel abstractions, not only common used parallel data types like parallel arrays but further parallel data types like lists, trees, and graphs are introduced. Beside other applications, a constraint system is implemented with this model where constraints and variables are distributed over a workstation net and proceed in parallel. For distribution a Task Interaction Graph model in coordination with bisection and the Kernighan-Lin algorithm is used. For an appropriate problem size a speed-up for constraint propagation could be achieved.

References

[1] G. A. Agha. Concurrent object-oriented programming. *Communications of the ACM, 33(9)*, 125–141, September 1990.

[2] Guy Blelloch. NESL: A Nested Data-Parallel Language. CMU-CS-93-129, April 1993.

[3] J. P. Briot, and R. Guerraoui. A Classification of Various Approaches for Object-Based Parallel and Distributed Programming. To appear in LNAI 1624, 1999.

[4] Chaco. *The Chaco user's guide: Version 2.0.* Tech. Rep. SAND94-2692, Sandia National Laboratories, Albuquerque, NM, July 1995.

[5] C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Artificial Intelligence 8*, 323–364, 1977.

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

[7] K. Ho. *High-Level Abstractions for Symbolic Parallel Programming.* PhD thesis, University of California at Berkeley, 1994.

[8] 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.

[9] L. Hotz and G. Kamp. Programming the Connection Machine by using the Metaobject Protocol. *Parallel Computing, Trends and Applications*, North Holland, 1994. Elsevier Science Publishers.

[10] L. Hotz, P. Struss and T. Guckenbiehl (edt.). *Intelligent Diagnosis in Industrial Applications.* Shaker Verlag, 2000.

[11] L. Hotz and M. Trowe. NetCLOS - Parallel Programming in Common Lisp. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'99*, H. R. Arabnia (edt.), Volume IV, 2034 – 2040, CSREA Press, 1999.

[12] B. Kernighan and S. Lin. *An Efficient Heuristic Procedure for Partitioning Graphs.* Bell System Technical Journal 29, 121–133, 1970.

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

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

[15] P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufman, San Mateo, California, 1995.

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

[17] H. Simon. Partitioning of Unstructured Problems for Parallel Processing. *Proc. Conference on Parallel Methods on Large Scale Structured Analysis and Physics Applications.* Pergammon Press, 1991.

[18] D. B. Skillicorn. *Foundations of Parallel Programming.* Cambridge University Press, 1994.

[19] G. L. Steele. *Common Lisp The Language Second Edition.* Digital Press, 1990.

[20] E. Tsang Foundations of Constraint Satisfaction. Academic Press, 1993.

[21] M. Trowe. *An Abstraction for Parallel Programming in Lisp on a Workstation Cluster.* Diplomarbeit in German. Universität Hamburg, 1998.

[22] A. Yonezawa and J.-P. Briot and E. Shibayama. *Object-Oriented Concurrent Programming in ABCL/1.* ACM SIGPLAN Notices, 21(11), 259–268, 1986.

[23] A. Zavanella. *Skeletons and BSP: Performance Portability for Parallel Programming.* PH.D. Thesis, Pisa, 1999.