HOPSA — a High-level Programming Language for Parallel Computations

Manfred Broy, Claus Dendorfer, Ketil Stølen
SFB342, Technische Universität München
Arcisstraße 21, 80290 München
E-mail: broy,dendorfe,stoelen@informatik.tu-muenchen.de

Abstract

The use of massive parallel computer architectures for the solution of computation intensive tasks requires specific programming concepts and thus makes programming more difficult. This is because the parallel execution and the particular properties of the chosen machine architecture must be taken into consideration. An abstract programming language more closely reflecting the specification notation is therefore desirable. Programs written in this language should allow a translation into efficient code for massive parallel computers. In that connection, one may ask: which aspects of parallel programming should be treated explicitly in the source code, and which aspects (like load balancing, parallelization and process administration) should be generated by a translator with certain analyzing capabilities. Our long term goal is the implementation of such a language based, for example, on the operating system MMK, *Multitasking Multiprocessing Kernel,* which has been developed at the Technische Universität München.

1 Introduction

Because of the very complicated process interactions, parallel programming is much more difficult than programming in traditional sequential languages. Since parallel programs often are expected to work in environments where high reliability is essential (telecommunication, industrial process controlling etc.), some means are needed to develop such programs in a systematic fashion, and to formally verify their critical parts with respect to specifications. As an additional difficulty, in many cases, because of high performance requirements, massive parallel machines have to be used. Therefore the programs must be written in languages which permit translation into efficient code for such architectures. Such languages should offer a suitable model for parallel processing that does not depend upon a particular computer architecture and therefore can be adapted to a variety of machines. Hence there are (at least) four major issues to consider in the development of realistic parallel programs:

- ease of programming and appropriate programming concepts,
- reasonably efficient implementation,

- embedding into a development method which allows for formal verification,
- suitable abstraction from concrete computer architectures.

In this introduction, we will discuss some approaches to parallel programming with respect to these criteria.

Conventional shared-state languages with parallel constructs like some dialects of FORTRAN usually provide very efficient implementations. However, in such programming languages even the behavior of programs consisting of only a few lines of code can be hard to figure out. Moreover, large and complicated proofs are often required to formally verify that such programs satisfy certain properties. For many of these languages there is not even a proper semantics.

An interesting new development are "parallel" languages that have no explicit parallel constructs at all. This is for example the case in Jade [RSL92], where a "parallel" program is basically a sequential program augmented with pragmatic declarations intended to help the compiler find a sensible parallelization. All parallel executions of a Jade program deterministically generate the same result as a sequential execution. Thus nondeterminism and time-dependencies, which normally make parallel programs hard to debug, cannot occur. Standard refinement calculi for sequential programs like [Jon90], [Mor90] can easily be extended to allow for the development of programs in a language like Jade (see for example [Len82]). However, the fact that nondeterminism and time-dependency cannot occur also indicates the weaknesses of this approach: only a restricted amount of parallelization can be gained, and moreover, the language is unsuitable for modeling inherently concurrent systems like communication systems.

Sequential programming languages are often "parallelized" by adding some additional programming constructs for parallel executions. Linda [CG89] is a formalism for conducting such extensions. Linda only deals with process creation and communication. The actual computing must be coded in the sequential language in which Linda is embedded. Linda can be used to model coarse, medium and fine-grained approaches to parallelism. The communication primitives of Linda are quite low-level, which means that the use of formal methods can be difficult.

PCN [FT91] is a procedural language with explicit programming constructs for parallel execution. The individual processes are coded in C, and processes can communicate only over definitional variables. A definitional variable is initially undefined, and what is thereafter assigned to it can never be overwritten. This ensures that there is no nondeterminism due to different interleavings of atomic statements. Since definitional variables for example can be of type stream, stepwise read/write communication is possible. The creators of PCN estimate that only 3% of the total execution time is spent by the parts written in PCN code. Thus, the overhead is low. Moreover, the exclusive use of definitional variables simplifies formal reasoning.

Another well-known concept for procedural parallel languages appears in formalisms like CSP [Hoa86] and languages like Occam [Ltd88]. A CSP process can only send and receive messages by synchronous communication. This means that the processes interact in a much more controlled way than in a shared-state language. A problem is that Occam is not really a high-level language; it is rather designed for low-level system programming.

There are also many approaches to parallel programming in the object-oriented tradition. However, so far there are no fully compositional development methods for this type

of languages (some interesting preliminary attempts are described in [Ame89], [Jon92], [Mey93]).

Often it is claimed that declarative languages are easier to use. For example, pure functional languages like Haskell [HW90] seem to be very convenient for programming. Also the efficiency of sequential implementations of these languages has been improved to such a degree that functional and C [KR88] programs reach the same order of magnitude in execution times [AJ89]. Because functional programming is very close to writing equations, it is quite easy to design such programs and reason about them. Theoretically, functional programs can be executed both by sequential and parallel machines without modification [PJ89], [Rep91]. There is, however, currently no efficient distributed implementation of such a programming language. An additional problem is that functional programming languages offer too many constructs that do not fit well into a development method. We think that by suitably restricting the programming languages program engineering becomes easier. Note that there are also approaches which enrich functional programming languages with explicit parallel constructs and communication primitives (see for example [vENPS90]).

There are a number of other languages which claim to be both easy to use and specification close. In some application areas, the use of executable specification languages has become standard, for example in the field of communication systems [Hog89]. For less specialized applications, it can be argued that some sort of logic programming language would be very suitable. However, we do not think that programs written in such a language are more understandable than functional programs, and the implementations of logic programming languages are much less efficient. Constraint programming may be an alternative, but currently it is difficult to use (see [SHW93]).

In this paper we will present a simple, data-flow like, functional language, called HOPSA, currently being designed at the Technische Universität München. In this language, the sequential processes are coded in functional programming notation, while "standard" communication tools are employed to connect the processes in a systematic way. The communication structure is basically characterized by a set of equations.

Programs written in this language are easy to write. A development method for HOPSA programs, called FOCUS, is also available. HOPSA programs are specification close and therefore well-suited for formal program development.

All the mentioned programming languages have their respective merits. However, they differ from HOPSA in that the latter has been specially designed to meet the four requirements stated above. In this paper, we will devote one section to each of these requirements. In particular, we will comment on

- the programming language HOPSA,
- a possible implementation of HOPSA,
- the development method FOCUS aimed at writing correct HOPSA programs,
- methods to map HOPSA programs onto different computer architectures using dynamic load balancing techniques.

2 The Programming Language HOPSA

A HOPSA program can be modeled in terms of networks of *agents*. Each agent has a fixed number of input/output ports, which are connected to other agents by directed, asynchronous communication channels. Agents communicate only via these channels. This model is compositional in the sense that whole networks of agents can be viewed as agents again. Hence there are two types of agents:

- *basic* agents, which can be thought of as the sequential processes of our language or the atomic building blocks performing the computations,
- *composite* agents, i.e. agents representing whole networks of agents.

A HOPSA program consists of three disjoint sets of declarations:

- declarations of basic agents,
- declarations of composite agents,
- declarations of channels.

The declaration of a basic agent is nothing else than a functional program defining a continuous function which, given a tuple of complete input histories represented by the streams of messages received on the input channels, yields a tuple of complete output histories represented by the streams of messages sent along the output channels. The syntax for the declaration of basic agents does not differ significantly from the syntax of well-known functional languages like Haskell or ML [HMM86]. An example is given in Figure 2.

The declaration of a composite agent is basically a let construct with a set of channel declarations, characterizing a network of agents, in its body.

The declaration of a channel is an equation with free channel variables. It characterizes the way messages are assigned to channels.

The declarations of channels together with the declarations of composite agents constitute what we will refer to as the *network definition*.

The network definition of a simple sorting network is given in Figure 1 and represented graphically in Figure 3. It consists of two channel declarations defining the channels i and o, respectively, and a declaration of the composite agent *sort*, which has two additional channel declarations in its right-hand side. $frontend$ and *cell* are basic agents. $frontend$ basically sends the sequence of messages to be sorted along i and receives the sorted sequence on o.

The functional definition of *cell* is given in Figure 2. The agent *cell* can store one message and has two input channels and two output channels. If the next message received along its first input channel is less than the stored message, the new message is stored, and the old message is sent along its second output channel, otherwise the stored message remains in its store, and the new message is forwarded. When a special end of sequence message eof is received, the agent *cell* passes this message on, sends the stored message

$$
\begin{aligned}
& i = frontend(o) \\
& \\
& o = \mathsf{await}\ i\ \mathsf{then}\ sort(i) \\
& \\
& sort(c) = \mathsf{let} \\
& \qquad\qquad (d, b) = cell(c, a) \\
& \qquad\qquad a = \mathsf{await}\ b\ \mathsf{then}\ sort(b) \\
& \qquad\quad \mathsf{in}\ d
\end{aligned}
$$

Figure 1: Network Definition of the Sorting Network.

$$
\begin{aligned}
cell &: StreamData \times StreamData \rightarrow StreamData \times StreamData \\
store &: Data \times StreamData \times StreamData \rightarrow StreamData \times StreamData \\
copy &: StreamData \times StreamData \rightarrow StreamData \times StreamData \\
& \\
cell(\mathsf{eof}\ \&\ i, a) &= [\mathsf{eof}, \langle\rangle]\ \&\ cell(i, a) \\
cell(d\ \&\ i, a) &= store(d, i, a) \\
& \\
store(s, \mathsf{eof}\ \&\ i, a) &= [s, \mathsf{eof}]\ \&\ copy(i, a) \\
store(s, d\ \&\ i, a) &= [\langle\rangle, max(s, d)]\ \&\ store(min(s, d), i, a) \\
& \\
copy(i, \mathsf{eof}\ \&\ a) &= [\mathsf{eof}, \langle\rangle]\ \&\ cell(i, a) \\
copy(i, d\ \&\ a) &= [d, \langle\rangle]\ \&\ copy(i, a)
\end{aligned}
$$

Figure 2: Functional Definition of a Single Sort Cell.

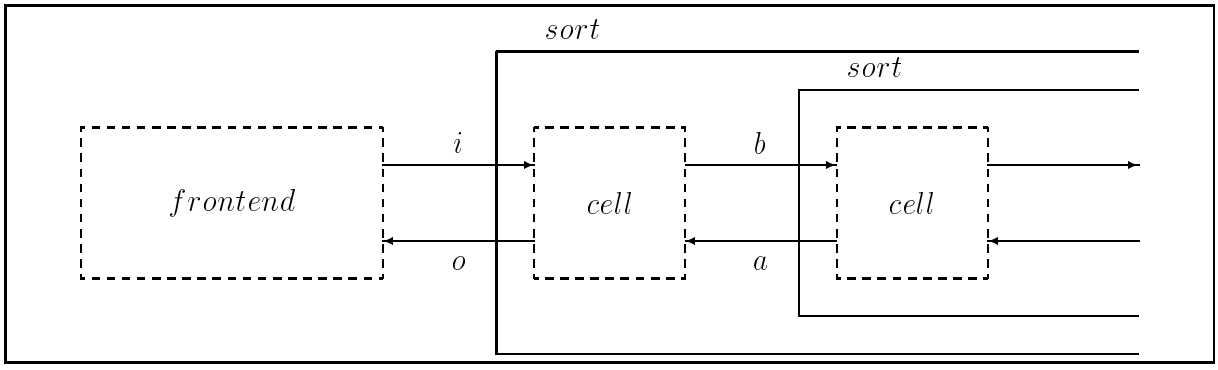

Figure 3: Graphical Representation of the Sorting Network.

along the first output channel, and thereafter forwards the messages it receives on its second input channel along its first output channel until it once more receives the end of sequence message, which has been "reflected" by the first *cell* not storing a message.

Clearly, the unfolding of the potentially infinite network of sort cells must be conducted in line with the progress of the computation, so that there are always "enough" sort cells available, and, at the same time, only the required number of sort cells is created. This is the task of the composite agent *sort*, i.e. it controls the unfolding of the sorting algorithm. Each *sort* call creates a new *cell* agent, and when (if at all) the first message is received on the local channel b, a new *sort* agent is also created. Thus, await b then $sort(b)$ waits until a message is received on b and then creates a new *sort* agent. As a result we get an unbounded (potentially infinite) list of sorting cells which forward new messages until they "fall" in place and then send them back in sorted order. The algorithm is linear with respect to both time and space.

Observe that in channel declarations global names for channels are introduced, while the channel variables occurring in, say, *sort*, are just formal parameters. If we substitute i and o for respectively c and d in *sort* we get exactly the same network. However, if we substitute c and d for respectively i and o in the second channel declaration only, we get a network with four unconnected channels.

The await construct may seem unnecessary at a first glance. An obvious alternative would be "lazy creation", where each process is created as soon as the first input message arrives. However, this would lead to unexpected results for agents declared, for example, by $f(x) = \langle 1, 2, 3 \rangle$. Such an agent would only produce output if at least one input message is received on its input channel, which does not agree with its (seemingly) obvious equational definition. Therefore, we decided to control process creation explicitly in terms of the await construct.

A second alternative would be to implement a demand driven process creation mechanism, where new processes are only created if their output is actually needed. However, this would require the use of a request-supply protocol for each of the asynchronous channels, which does not fit very well into the simple framework of asynchronous communication.

3 The Implementation of HOPSA

Some implementation experiments have been carried out based on the core language as defined above. In particular, a compiler has been written that translates network definitions into system calls of MMK [BKMR91]. MMK provides basic process creation mechanisms and supports process communication in terms of so-called mailboxes. These mailboxes can be configured in a number of ways, in particular they can be used to approximate asynchronous, buffered channels (in fact there is a maximum buffer length, which we disregarded in our experiments).

The compiler is a rather small program written in Standard ML, which takes HOPSA programs and produces a variety of files that are further processed by an MMK preprocessor and the standard C compiler.

Up to now, the sequential agents themselves have to be written in pure C extended with some of MMK's communication instructions. A compiler for the translation of sequential agents written in the functional programming notation is of course also planned. However,

since the sequential implementation of functional languages is quite well understood, we have so far concentrated on the implementation of the network definitions.

4 The Development Method FOCUS

FOCUS is a general framework, in the tradition of [Kah74], [Kel78], for the formal specification and development of distributed systems. A system is modeled by a network of agents communicating asynchronously via unbounded FIFO channels. A large number of different reasoning styles and techniques is supported (see for example [BDD+92], [Bro92c], [Bro92a], [SDW93]).

FOCUS provides mathematical formalisms which support the formulation of highly abstract, not necessarily executable specifications with a clear semantics. Moreover, FOCUS offers powerful refinement calculi, which allow distributed systems to be developed in the same style as sequential programs can be developed in VDM [Jon90] and the refinement calculi of [Bac88], [Mor90]. Finally, FOCUS is modular in the sense that design decisions may be checked at the point where they are taken, that component specifications can be developed in isolation, and that already completed developments can be reused.

A development of a program in FOCUS can be split into three main phases:

- a requirement phase where the requirement specification is formulated,
- a design phase where the architecture independent system development is carried out,
- an implementation phase where the system specification is mapped onto a particular architecture.

Any FOCUS specification can be modeled by a set of timed stream processing functions as defined in [Bro92b]. The same type of semantics can be assigned to HOPSA. Because of the close relationship between stream processing functions and the programming notation in HOPSA, FOCUS is particularly suited for the development of HOPSA programs. HOPSA may be used either as the final implementation language or as an executable prototype language.

5 The Mapping of Agents onto Processors

A network of agents characterized by a HOPSA program must be mapped onto a real processor architecture in some suitable way. As for example in UNITY [CM88], a HOPSA program as defined above only specifies the *maximum* possible parallelism. Considering that communication usually is rather expensive, it is not sensible to map every single agent to a new processor. This is especially obvious in the sorting example above, where a sort cell only stores one data element, and the number of sort cells required equals the length of the input.

Such a mapping from agents (and channels) to processors is a very basic theoretical concept. Nevertheless, it allows us to model and analyze all difficult pragmatic questions

of process scheduling and load balancing. The mapping may change dynamically, i.e. during the execution of the program, which mirrors dynamic load balancing and process migration. An interesting question at this point is of course to what degree this mapping should (and can) be expressed explicitly in the HOPSA language, and how efficient heuristic mapping strategies are.

6 Extensions

As already pointed out, the HOPSA language is still at an experimental stage. In fact so far only a core language has been fixed, and a number of extensions are currently being considered.

In particular additional constructs for the manipulation of dynamic networks are needed. Currently, a network can be *expanded* via the await construct. However, there is no possibility to shrink a network by deleting agents and channels. Thus, pulsating networks cannot be expressed in the current version of HOPSA. One relatively straightforward extension to handle this problem is to have a construct which kills some part of the network structure when a certain end of computation message is received. For example with respect to the sorting algorithm above, we could employ this construct in the declaration of *sort* to kill the next *cell* and *sort* agents together with the local channels b and a when a certain message is received on the channel a. Here the question arises whether agents should be "killed" or "sent asleep", i.e., whether agents that are repeatedly deleted and created in a pulsating network should each time start from the same initial state, or from the last state of the previous incarnation. This leads also to questions regarding the garbage collection of processes.

We are also considering more general types of agent communications such as broadcasting. The communication of higher-order messages can also be used to express dynamic networks. Higher-order messages are nothing else than agents communicated via channels. Some interesting implementation issues arise when higher-order messages are allowed, for example whether the whole program should be transferred or just a pointer to the (shared) code and some status information.

It is well-known that certain weakly time-dependent agents like fair merge are hard to express in a functional setting [Kel78]. One straightforward way to handle this problem in HOPSA is to incorporate a specific fair merge construct [Bro88]. This construct characterizes an agent performing a fair merge of the messages received on its input channels. This construct can be modeled in the same way as any other HOPSA agent — by a set of timed stream processing functions.

The issue of dynamic network reconfiguration deserves further attention. The FOCUS framework seems to be well-suited for the:

- description of dynamic networks,
- reconfiguration of networks during execution,
- mapping of dynamic process networks onto static networks of processors,
- reconfiguration of processor mappings.

These issues are of high importance for the modeling of distributed systems. We plan to study them in more detail in the future.

7 Conclusions

In this paper we have advocated the use of a high-level, data-flow like, functional programming language called HOPSA. We have outlined a possible implementation, a design methodology, and some thoughts about efficient execution on parallel hardware.

Because

- the individual, sequential agents are conveniently coded in a functional programming notation,
- an efficient implementation is possible since the communication structure is given explicitly,
- the programming language is embedded into a formal development method,
- the concept of agents and channels, which are dynamically mapped onto processors, provides a suitable abstraction level for the writing of parallel programs,

we claim that the concept on which HOPSA is based meets the basic requirements stated in the introduction.

References

[AJ89] L. Augustsson and T. Johnsson. The Chalmers lazy-ML compiler. *The Computer Journal*, 32:127–141, 1989.

[Ame89] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1:366–411, 1989.

[Bac88] R. J. R. Back. A calculus of refinments for program derivations. *Acta Informatica*, 25:593–624, 1988.

[BDD+92] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The design of distributed systems — an introduction to Focus. Technical Report SFB 342/2/92 A, Technische Universität München, 1992.

[BKMR91] T. Bemmerl, C. Kasperbauer, M. Mairandres, and B. Ries. Programming tools for distributed multiprocessor computing environments. Technical Report SFB 342/31/91 A, Technische Universität München, 1991.

[Bro88] M. Broy. Nondeterministic data flow programs: How to avoid the merge anomaly. *Science of Computer Programming*, 10:65–85, 1988.

[Bro92a] M. Broy. Compositional refinement of interactive systems. Working Material, International Summer School on Program Design Calculi, August 1992.

[Bro92b] M. Broy. Functional specification of time sensitive communicating systems. In M. Broy, editor, *Proc. Programming and Mathematical Method*, pages 325–367. Springer, 1992.

[Bro92c] M. Broy. (Inter-) action refinement: the easy way. Working Material, International Summer School on Program Design Calculi, August 1992.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32:444–458, 1989.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design, A Foundation*. Addison-Wesley, 1988.

[FT91] I. Foster and S. Tuecke. Parallel programming with PCN. Technical Report ANL-91/32, Version 1.2, Argonne National Laboratory, 1991.

[HMM86] R. W. Harper, D. B. MacQueen, and R. G. Milner. Standard ML. Technical Report ECS-LFCS-86-2, University of Edinburgh, 1986.

[Hoa86] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1986.

[Hog89] D. Hogrefe. *Estelle, LOTOS and SDL, Standard-Spezifikationssprachen für verteilte Systeme*. Springer, 1989.

[HW90] P. Hudak and P. Wadler. Report on the programming language Haskell. Technical Report YALEU/DCS/RR-777, Yale University, 1990.

[Jon90] C. B. Jones. *Systematic Software Development Using VDM, Second Edition*. Prentice-Hall, 1990.

[Jon92] C. B. Jones. An object-based design method for concurrent programs. Technical Report UMCS-92-12-1, University of Manchester, 1992.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J.L. Rosenfeld, editor, *Proc. Information Processing 74*, pages 471–475. North-Holland, 1974.

[Kel78] R. M. Keller. Denotational models for parallel programs with indeterminate operators. In E. J. Neuhold, editor, *Proc. Formal Description of Programming Concepts*, pages 337–366. North-Holland, 1978.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[Len82] C. Lengauer. *A Methodology for Programming with Concurrency*. PhD thesis, University of Toronto, 1982.

[Ltd88] INMOS Ltd., editor. *Occam 2 Reference Manual*. Prentice Hall, 1988.

[Mey93] B. Meyer. Systematic concurrent object-oriented programming. Technical Report TR-EI-37/SC, ISE, Santa Barbara, 1993.

[Mor90] C. Morgan. *Programming from Specification*. Prentice-Hall, 1990.

[PJ89] S. L. Peyton Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32:175–186, 1989.

[Rep91] J. H. Reppy. *Concurrent Programming with Events — The Concurrent ML Manual*. Cornell University, 1991.

[RSL92] M. C. Rinard, D. J. Scales, and M. S. Lam. Heterogeneous parallel programming in Jade. In *Proceedings of Supercomputing'92*, pages 245–256, 1992.

[SDW93] K. Stølen, F. Dederichs, and R. Weber. Assumption/commitment rules for networks of asynchronously communicating agents. Technical Report SFB 342/2/93 A, Technische Universität München, 1993.

[SHW93] G. Smolka, M. Henz, and J. Würtz. Object oriented concurrent constraint programming in OZ. Technical Report RR-93-16, DFKI, 1993.

[vENPS90] M. van Eekelen, E. Nöcker, R. Plasmeijer, and S. Smetsers. *Concurrent Clean (Manual)*. University of Nijmegen, 1990.