# Using the Centaur system for data-parallel SIMD programming: a case study

Jean-Luc Levaire
LIP, ENS Lyon,
46 Allée d'Italie,
F-69364 Lyon Cedex 07, France
Email: jllevair@lip.ens-lyon.fr

**Abstract**

We discuss the application of Centaur to designing data-parallel SIMD languages. We first present the main functionalities of this system. We describe a simple language, called $\mathcal{L}$, which embodies the basic concepts of real SIMD languages like C*, MPL or POMPC. We give an operational semantics to this language and we discuss in detail its implementation under Centaur. Finally, we present a SIMD programming environment created with Centaur for the $\mathcal{L}$ language.

## Introduction

The Centaur system [17] is a software toolbox to design and develop programming languages. It provides a variety of tools to study their syntactic and semantic aspects, and enables to create specific editors, interpreters or debuggers. Centaur is based on the manipulation of abstract trees, which are the internal representation of programs in the specified language.

The syntactic aspect of a language is given by the correspondence between the representation of a program as a linear text and as an abstract tree. For the semantic aspect, Centaur uses the notion of natural semantics. Evaluating a program amounts to prove a particular proposition using a set of inference rules and axioms. This set constitutes in fact the semantic specification of the language. Finally, a windowing system is also provided to develop specialized multiwindow programming environments.

Centaur has already been used to study the semantics of sequential languages such as Pascal [6], functional languages such as ML [8] and parallel languages such as ADA. Until now, those parallel languages were *control-parallel* languages, aimed at MIMD (Multiple Instruction, Multiple Data) architectures [7]. Parallelism is there expressed with new parallel control structures. But *data-parallel* languages, such as those designed for SIMD (Single Instruction, Multiple Data) architectures are of interest too. In these languages, control is centralized and data are distributed: the processors execute all the same instruction, but on their own data.

Unlike MIMD languages, few theoretical studies of SIMD languages have been made, even though many such languages are commercially available. A number of SIMD languages are derived from the C language: C* [5] for the Connection Machine [9], MPL [11] for the Maspar MP-1 [1] and POMPC [14] for the data-parallel POMP machine [10]. Recently, Bougé and Garda have proposed in [2] and [3] a simple language called $\mathcal{L}$, which embodies the basic concepts of those languages. In particular, they show that $\mathcal{L}$ is minimal and expressive in a certain sense.

We discuss here the implementation of the $\mathcal{L}$ language using Centaur. We choose this language because of its simplicity and its expressiveness. This ensures that any SIMD algorithm can be written in the $\mathcal{L}$ language. We also describe the basic functionalities of a programming environment for SIMD languages. Centaur is well suited to design, study and improve such an environment.

First, we describe the Centaur system and its main components. Then, we present the $\mathcal{L}$ language and give its operational semantics. Based on this precise semantic definition, we describe its implementation under Centaur. Finally, we present the resulting programming environment through an example of program.

# 1 Centaur

Centaur is a software toolbox to assist users in designing programming languages and developing programming environments. It mainly consists in a kernel which manipulates internal objects, a number of specification languages to define the syntactic and semantic aspects of a language, and an interface which handles communication between Centaur and the user.

## 1.1 The kernel

The kernel is divided into two parts. The first part is concerned with the syntactic aspect of a language. The fundamental object manipulated by the kernel is the abstract syntax tree. It corresponds to the arborescent representation of program terms. The nodes of these trees are called *operators*. Two types of operators are distinguished.

- ▷ The atomic operators correspond to the leaves of the trees. One specifies the type of their values using the following classes: `INTEGER`, `IDENTIFIER`, `CHAR`, `STRING`, `SINGLETON` (operator without value). An additional class, called `TREE`, is used when the value of the operator is a tree from an another language. This mechanism allows to stratify the specification of languages, and to use common parts in different languages.
- ▷ The non-atomic operators correspond to the internal nodes. One specifies their arity (fixed arity, list, non-empty list) and the type of their descendents. Such a type consists for each possible son in a set of valid operators. This set is called a *phylum*.

The Virtual Tree Processor (VTP), a collection of Lisp [4] primitives, enables the user to create, modify or examine these trees.

The second part of the kernel is a semantic machine which evaluates programs written in the language under study. The semantics of a language is specified with a collection of rules concerning its terms. This machine is a logical engine, written in Mu-Prolog [13], which constructs proofs corresponding to a given term using this set of rules. Such a proof corresponds in fact to a computation of the program. Finally, communication primitives between the VTP and Mu-Prolog enable the conversion of abstract trees into Prolog terms and conversely. They are called before and after each evaluation.

## 1.2 Using Centaur

Implementing a language with Centaur consists in the following steps.

- ▷ The user specifies the concrete syntax (text) of his language, its abstract syntax (VTP tree) and the translation from the first syntax to the second one. He describes in fact a parser which is used when loading a program to detect syntax errors, and to construct the corresponding abstract syntax tree. The specification language used to define such a translation is called Metal (Meta-Language).
- ▷ Then, he has to describe the inverse process: translation from an abstract tree to text. This is done with PPML (Pretty Printer Meta Language). At this point, the user has generated a language-oriented editor for programs in his language.
- ▷ The semantic aspect is specified with the Typol language. This Typol specification is compiled into a set of Prolog clauses used by the logical engine to evaluate a program.
- ▷ Finally, the user builds a specific programming environment in the Lisp language, using window managing primitives. He can manipulate the abstract trees with the VTP, and call the semantic engine from his environment.

### 1.3 The Typol semantic specification language

A Typol program consists in a set of rules, which are either axioms, or inference rules. Inference rules look like

$$\frac{premises}{conclusion}$$

This means that the conclusion is proved as soon as premises are. A premise is either a condition, or a sequent. A sequent is a proposition (the consequent) associated with the hypotheses required to prove it. It is denoted

$$hypotheses \vdash consequent$$

A conclusion is always a sequent. Finally, different infix symbols (->, =>, :) can be used in the propositions, in order to separate various semantic aspects of terms in a language (instructions and expressions for instance). Intuitively, the expression at the left of the proposition symbol in a consequent corresponds to an abstract tree of the language. The right part represents its semantics. Typol uses pattern matching to find the inference rule or the axiom corresponding to the given program.

Let us consider the following rule

```
sigma |- EXP : v  & sigma |- X, v => sigma'
---------------
sigma |- assign(X, EXP) -> sigma';
```

It means

> Executing instruction `assign(X,EXP)` from environment `sigma` produces an environment `sigma'`, so that `sigma'` is deduced from `sigma` by substituing the value of `X` with `v`, where `v` is the evaluation of expression `EXP` in `sigma`.

The abstract tree of the consequent is the instruction `assign(X, EXP)`. Its evaluation (symbol `->`) yields an environment (a function which binds a variable to its value). In the same way, the evaluation (symbol `:`) of an expression yields a value, an integer for instance.

## 2 The $\mathcal{L}$ language and its semantics

The $\mathcal{L}$ language has been proposed by Bougé and Garda as a simple programming model. This model covers most recent SIMD machines, such as the Connection Machine CM-2 or the Maspar MP-1. We consider a set of Processing Elements (PEs), each managing a private memory. The PEs are controlled by a unique external sequencer, which broadcasts the common instruction to be executed. Thus, all PEs do the same instruction, but on their own data. Furthermore, an inhibition mechanism, called the *context*, is associated to each processor. A PE modifies its local memory only if its context is in an *active* state. A PE which is not active is said to be idle. This possibility is essential to implement parallel conditioning branch. Finally, a global bus links the set of PEs to the sequencer, and computes the global *or* of the elements of a boolean vector, of which each component is local to a PE. This feature enables the sequencer to detect the termination of loops.

The $\mathcal{L}$ language includes five constructions: assignation, communication, sequencing, iteration and conditioning. We give here a Structured Operational Semantics (SOS) [15] for $\mathcal{L}$. This kind of semantics defines a transition system, given by a set of rules and axioms. It works by induction on a program $P$. The states of the semantics consist in the program remaining to be executed, the environment $\sigma$, and a stack $ct$ of contexts, which are boolean vectors. A stack is necessary to implement nested conditional structures.

We adopt the following notations. Identifiers with an initial uppercase letter denote parallel variables, which we also called vectors (possibly multidimensionnal). Processor locations will be denoted $u$, $v$ etc. $X|_u$ denotes the element of $X$ located on the processor whose address is $u$. The states of the semantics will be denoted $\langle P, \sigma, ct\rangle$ and $\bullet$ denotes the empty program. $[\![E]\!](\sigma)$ is the evaluation of the expression $E$ in the environment $\sigma$. $\epsilon$ is the empty stack. By convention $Pop(\epsilon) = \epsilon$ and $Top(\epsilon) = Tt$, the boolean vector whose all elements have the *true* value. Finally, we define the *active* predicate as follows

$$active(u) \equiv (Top(ct)|_u = tt).$$

**Assignment** The instruction $X := E$ stores into variable $X$ the value of expression $E$. An idle processor remains idle, and leaves its local memory unchanged.

$$\langle X := E, \sigma, ct\rangle \longrightarrow \langle \bullet, \sigma', ct\rangle$$

with

- $\sigma'(X)|_u = [\![E]\!](\sigma)|_u$ if $active(u)$;
- $\sigma'(T)|_u = \sigma(T)|_u$ if $T \neq X$ or $\neg\, active(u)$.

In $\mathcal{L}$, all expressions are required to be elementwise: the value of an expression at a PE depends on the value of variables at this PE only.

**Communication** For simplicity, we consider only regular communication patterns here. The instruction `shift` $X$ `along` $d$ shifts the vector $X$ along the constant common direction $d$. Again, an idle processor leaves its local memory unchanged. For a PE located at $u$, $\bar{d}(u)$ is the address of the PE located in the direction opposite to $d$.

$$\langle \texttt{shift}\ X\ \texttt{along}\ d, \sigma, ct\rangle \longrightarrow \langle \bullet, \sigma', ct\rangle$$

with

- $\sigma'(X)|_u = \sigma(X)|_{\bar{d}(u)}$ if $active(u)$;
- $\sigma'(T)|_u = \sigma(T)|_u$ si $T \neq X$ or $\neg\, active(u)$.

**Sequencing** The construct $P;Q$ executes $P$ then $Q$.

$$\frac{\langle P,\sigma,ct\rangle \longrightarrow \langle P',\sigma',ct'\rangle \qquad P' \neq \bullet}{\langle P;Q,\sigma,ct\rangle \longrightarrow \langle P';Q,\sigma',ct'\rangle} \qquad \frac{\langle P,\sigma,ct\rangle \longrightarrow \langle \bullet,\sigma',ct'\rangle}{\langle P;Q,\sigma,ct\rangle \longrightarrow \langle Q,\sigma',ct'\rangle}$$

**Iteration** The construct `while` $B$ `do` $P$ `end` iterates program $P$ upto a point where the *elementwise* boolean expression $B$ evaluates to false at each *active* PE. Notice that this construct does not modify the context stack.

$$\frac{(\exists u\,([\![B]\!](\sigma)|_u \wedge active(u))) = tt}{\langle \texttt{while}\ B\ \texttt{do}\ P\ \texttt{end},\sigma,ct\rangle \longrightarrow \langle P;\texttt{while}\ B\ \texttt{do}\ P\ \texttt{end},\sigma,ct\rangle}$$

$$\frac{(\exists u\,([\![B]\!](\sigma)|_u \wedge active(u))) = ff}{\langle \texttt{while}\ B\ \texttt{do}\ P\ \texttt{end},\sigma,ct\rangle \longrightarrow \langle \bullet,\sigma,ct\rangle}$$

**Conditioning** The construct `where` $B$ `do` $P$ `end` inhibits, during the execution of program $P$, those PEs whose *elementwise* boolean expression $B$ evaluates locally to false. The new activity corresponds to the boolean vector $Top(ct) \wedge [\![B]\!](\sigma)$, and is pushed on the context stack. To keep track of conditional blocks, we introduce the new syntactic construction `begin` $P$ `end`.

$$\langle \texttt{where}\ B\ \texttt{do}\ P\ \texttt{end},\sigma,ct\rangle \longrightarrow \langle \texttt{begin}\ P\ \texttt{end},\sigma,ct'\rangle$$

with $ct' = Push(Top(ct) \wedge [\![B]\!](\sigma), ct)$. The following rules express that $P$ is executed up to its termination, and, at that time, the former activity is restored by popping the context stack.

$$\frac{\langle P,\sigma,ct\rangle \longrightarrow \langle P',\sigma',ct'\rangle}{\langle \texttt{begin}\ P\ \texttt{end},\sigma,ct\rangle \longrightarrow \langle \texttt{begin}\ P'\ \texttt{end},\sigma',ct'\rangle}$$

$$\langle \texttt{begin}\ \bullet\ \texttt{end},\sigma,ct\rangle \longrightarrow \langle \bullet,\sigma,Pop(ct)\rangle$$

## 3 Implementing $\mathcal{L}$ under Centaur

The first step of this implementation is to define a grammar for $\mathcal{L}$, particularly for the boolean and arithmetic expressions. We use the syntax of the C language, except incrementation and decrementation operators which have side effects. Thus, writing the Metal and PPML specifications presents no specific difficulties. The only technical point is to set correctly the priorities on the operators of the language. Vectors are implemented by list operators, so that we can choose the size and the geometry of them. Moreover, they have no concrete syntax, because $\mathcal{L}$ provides no way to define vectors extensively.

In the following Typol rules, `sigma` represents the environment. It is implemented as a list of assignment operators. These operators bind a variable name to its vectorial value. A variable named `POSITION` is defined in the initial environment. It associates to each processor a unique identifier, starting at 0. The context is denoted `ct` in the Typol rules. It is a list of vectors, and represents the stack *ct* previously defined. Two alternative approaches are possible to write the Typol specification of $\mathcal{L}$.

### 3.1 The direct method

This is the more natural and simple approach. It is used for the implementation of classical sequential languages. The method corresponds to a *Big-Step* operational semantics in that sense that a program evaluation consists in only one big derivation. The program is not modified by its evaluation, but only the environment. The definition of such a semantics amounts to describing the modifications of the environment made by the statements and the constructs of the language. This method could also be applied to SIMD languages, like $\mathcal{L}$. The only difference with the sequential case comes from the context, which is also modified during the evaluation. Thus, this semantics of $\mathcal{L}$ would be specified as a function with two variables. Let us consider the rules for iteration. The Prolog predicates `for_all_actif_not(ct,v)` and `exist_actif(ct,v)` respectively test whether the value of the vector v is false on all *active* PEs, and whether there exists at least one active PE such that this value is true.

```
sigma |- EXP : v & for_all_actif_not(ct,v)
---------------
sigma, ct |- while(EXP, P) -> sigma, ct ;

sigma |- EXP : v & exist_actif(ct,v) &
sigma, ct |- P -> sigma', ct' &
sigma', ct' |- while(EXP, P) -> sigma'', ct''
---------------
sigma, ct |- while(EXP, P) -> sigma'', ct'' ;
```

The first rule expresses the termination of the loop, in which case environment and context remain unchanged. The second one contains a recursive call to itself, after the derivation of one iteration. Notice here that the environment and context resulting from this iteration, `sigma'` and `ct'`, become the initial environment and context of the following iteration. This corresponds to function composition.

When using this type of semantics, the evaluation of a whole program consists in evaluating exactly one Prolog term. Indeed, this evaluation is recursively made on the structure of the program. The main advantage of this method is that Centaur provides directly a debugger during the evaluation of a term, the Typol debugger. It traces the Typol rules successively applied, and the user can set breakpoints in his program, or examine variables which appear in the rule. An interesting feature is that he can also make a rule to fail, which forces Prolog to backtrack. The program is then executed upside down. Figure 1 shows, in the left window, the environment with the rule to be applied, and, in the right window, the current value of `sigma`. It is assumed here that the architecture is a $4 \times 4$ square grid of PEs.

The main drawback of this method is that the whole evaluation of a program is made under Prolog: since it keeps track of its whole proof, environment and context are stored between each application of rules. The memory space is then rapidly exhausted especially if vectors of large size (say, more than 16 elements) are manipulated.

### 3.2 The transformational method

This approach is more complex than the previous one. It has already been used with Centaur for languages with explicit control parallelism, such as MIMD languages. It is based on a *Small-Step* operational semantics where a step modifies the program under evaluation. An entire program

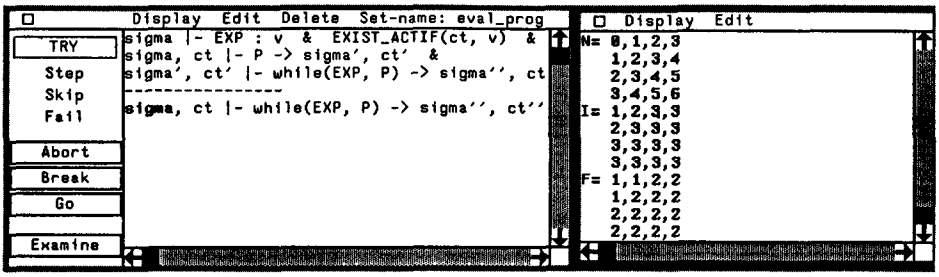


Figure 1: The Typol debugger.

evaluation consists then in many steps. This semantics transforms a program associated with an environment and a context into a new program, environment and context. It corresponds exactly to the operational semantics given in the previous section. The implementation is easier than for the MIMD case: the $\mathcal{L}$ language is deterministic, and communications are synchronous. We deduce Typol rules directly from the SOS rules. For instance, the rule for conditioning is

```
evaluate(EXP, sigma, V) &
push_and(ct, V, ct')
----------------
|- where(EXP, P), sigma, ct |-> begin(P), sigma, ct' ;
```

where `evaluate` and `push_and` are Prolog predicates which respectively evaluate the expression `EXP` and modify the context stack accordingly.

We can now see that the evaluation of a program consists in a sequence of term evaluation, each term being derived from the previous one by application of a semantic rule. Each call to Prolog corresponds to one transition in our semantics. Observe that it may involve many Typol inferences, as Typol rules are recursive in general. The result is a new state made of the new program, environment and context. Computing an entire program requires then as many calls to Prolog as transitions in the SOS semantics. The evaluation of a program stops when the returned program is $\bullet$ and no more rule applies.

The interesting point of this method is that the memory space required by Prolog for the evaluation of an entire program corresponds to that required for computing one transition. It amounts to the space used to represent the environment and the context, and to evaluate an expression. Hence, this method avoids the memory overflow if the complexity of expressions remains beyond reasonable limits.

On the other hand, we have to design a specific debugger. The Typol debugger is only available during the computation of a transition, that is during a call to Prolog. Between each Prolog computation, the control returns to the Lisp top-level. The debugger has to combine both Centaur and Lisp graphic primitives. It uses the communication primitives between the VTP and Prolog in order to exchange program, environment and context at each step. The skeleton of the debugger is a simple loop which iterates calls to Prolog and displays the new program at each iteration. In practice, this implementation works fine with bidimensionnal vectors of size $16 \times 16$. However, this size can be interactively modified by the user before the execution of a program.

### 3.3 The programming environment

Centaur and Lisp provide various graphic and interface functions to develop a specific programming environment. In our case, we have implemented the functionalities of the Maspar MP-1 programming environment MPPE (Maspar Programming Environment [12]) with several extensions.

Regarding the execution control, the user can choose between two modes: a step-by-step mode, with or without a call to the Typol debugger (`Step` and `Step(db)`), and a continuous mode (`Go`). In the second case, the evaluation can be stopped by clicking on a button (**Break**). The direct definition of breakpoints is difficult because the program is modified during its evaluation. It would require to add new structures to keep track of the localization of the breakpoints.

The more natural way to display vectors is to use grids. But rather than displaying the value of their elements, we show the value of a predicate concerning the selected variable, $X = 0$ for instance. To obtain the value of an element, the user simply clicks on it and a window will present the coordinates of the chosen element and its value. The context is displayed with a grid too, so the user can inspect at any time the current activity. Our implementation provides also the capability to trace predicates. These predicates can depend on both variable values and activity. For instance, the set of active processors so that the condition $A = (B + 1)$ holds is represented by the predicate `ACTIF && (A == (B + 1))`. This is an important improvement on MPPE, which resticts predicates to the comparison between one variable and a constant.

The following figures present these functionalities using a program which computes the connected component of a point in the image `IMAGE`. The point coordinates are $(10, 3)$, so it is located in the bottom left square of the image. The connected component `C` is build by dilating a wave from the initial point. The final component is found when the wave does not extend in the current iteration.

The left window of Figure 2 is the main window, containing the program and the execution control button. The window on the right displays the environment with a button (**Examine**) to display a selected variable, and a button (**Predicate**) to trace a predicate. The two left windows of Figure 3 show two variables, with the value of one element. The right window displays a predicate with a dialog box used to modify it.

## Conclusion

This work presents the first implementation of a SIMD data-parallel language under Centaur. The $\mathcal{L}$ language is fairly primitive. It can be extended to include the complex data-parallel flow control structures of MPL, such as the plural `while` with the associated plural **break** and `continue`. We actually implemented under Centaur an extension of $\mathcal{L}$ containing such control rupture statements.

We can observe a hierarchy of languages in terms of adequacy to Centaur. Classical sequential languages are the most easier to implement because of the simplicity of their control and data structures. Conversely, implementing MIMD languages is very difficult due to non-determinism, asynchronous communication and parallel execution. Finally, SIMD languages lie between these extreme cases. They are closer to sequential languages in terms of control. The main difficulty comes from the large size of data.

Centaur appears as an interesting semantic design tool. Implementing a language under Centaur helps the designer to study various alternatives for its semantics and to explore their

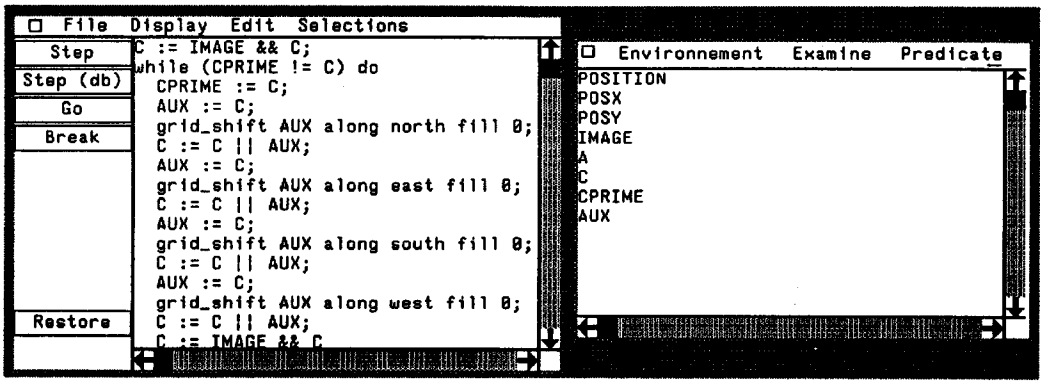


Figure 2: Connected component of a point in an image. Windows presenting the program and the environment.

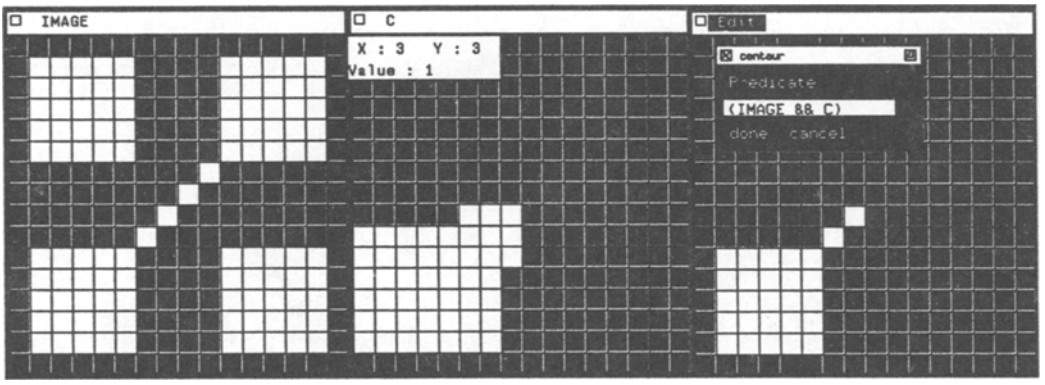


Figure 3: Windows presenting the variables IMAGE and C. The point is located in the bottom left square of the image IMAGE and variable C is the current component. The right window shows the predicate (IMAGE && C) with its dialog box.

impact. The existing SIMD languages present some differences as a consequence of the choices made during their design. For instance, in MPL the body of a conditional block is not executed if all the processors become inactive. This is not the case in C* and POMPC. Those differences may generate expressiveness problems between SIMD languages.

We can also use this implementation to refine SIMD programming environment and to develop software engineering tools. The existing software tools for sequential languages may be adapted to SIMD languages. These adaptations can be performed by the use of this application.

Finally, various extensions are possible. We can extend the language to include more generalized communication commands. We can also improve it by adding notion of shape as in C* or POMPC. Moreover the Centaur system can be used to provide intelligent syntax-directed editors to commercial languages. Our research group is currently developing a proof system based on Hoare's logic [16]. A future work will consist in integrating it in this environment.

## References


[1] Blank T. The MasPar MP-1 Architecture. Proc. of the 35th IEEE Computer Society Int. Conf. (Spring Compcon 90), San Fransisco, 1990, pp. 20–24.

[2] Bougé L. On the semantics of languages for massively parallel SIMD architectures. Proc. Parallel Arch. and Lang. Europe Conf. (PARLE), Eindhoven, Lect. Notes in Comp. Science 506, Springer Verlag, 1991, pp. 166–183.

[3] Bougé L., Garda P. Towards a Semantic Approach to SIMD Architectures and their Languages. Semantics of Systems of Concurrent Processes, Proc. LITP Spring School on Theor. Comp. Science, La Roche Posay, France, 1990, Lect. Notes in Comp. Science 469, Springer Verlag, pp. 142–175.

[4] Chailloux J., et al. LeLisp v15.2: Le manuel de référence. INRIA, 1986.

[5] C* Programming Guide. Thinking Machine Corporation, 1990.

[6] Van Deursen A. An algebraic specification for the static semantics of Pascal. Report CS-R91, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1991.

[7] Flynn M.J. Some Computer Organizations and Their Effectiveness. IEEE Trans. on Computers C-21, 9, 1972, pp 948–960.

[8] Hendriks P.R.H. Typechecking Mini-ML. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989, pp 299–337.

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

[10] Hoogvorst P., Keryell R., Matherat P., Paris N. POMP or how to design a massively parallel machine with small developments. Proc. Parallel Arch. and Arch. Europe Conf. (PARLE), Eindhoven, Lect. Notes in Comp. Science 505, Springer Verlag, 1991, pp. 83–100.

[11] MasPar Parallel Application Language Reference Manual. MasPar Computer Corporation, 1990.

[12] MasPar Programming Environment Reference Manual. MasPar Computer Corporation, 1991.

[13] Naish L. Mu-Prolog 3.2 Reference Manual. Tech. Rept. 85, University of Melbourne, 1987.

[14] Paris N. Définition de POMPC (version 1.9). Draft Version, LIENS, Paris, 1991.

[15] Plotkin G. An operational semantics for CSP. D. Bjorner (Ed.), Formal description of programming concepts, IFIP TC-2 Working Conference, Garmish-Partenkirchen, RFA, 1982.

[16] Utard G.E. Un système axiomatique pour les languages massivement parallèles SIMD. Master's Thesis, LIP, ENS Lyon, 1991.

[17] Centaur Reference Manual. Sema Group, 1990.