An implementation of Or-Parallelism based on direct access to the MMU

Laurent Perron
Ecole Normale Supérieure
45 Rue d'Ulm,
75005 Paris, France
perron@aloes.ens.fr
http://www.ens.fr/users/perron/

1. Introduction

Non-determinism is the ability to travel freely through a tree of hypotheses. It allows the programmer to try different hypothesis independently. This capacity is found in Logic Programming (aka Prolog), Constraint Logic Programming and Concurrent Constraint Logic Programming. It is fundamental for instance in the goal resolution in Prolog. But its implementation is complex and costly. There are essentially two opposite implementation schemes.

Trailing is an efficient sequential implementation scheme on a uni-processor computer. Trailing means that when trying an hypothesis, every change which happens to the database is stored, which allows to undo the changes (backtrack) before trying another hypothesis. In practice, changes are pushed on a stack. Every time a hypothesis is tried, a marker is positioned on the stack, thus allowing nested hypotheses and multi-level undoing. This technique is used in the Warren Abstract Machine implementation of Prolog[26].

Encapsulated search is another implementation scheme. It is found for instance in the Oz system [20, 21]. Instead of considering a unique *store* where results of computations are written, encapsulated search develops the notion of local stores. This means that different hypotheses will be tried on independent local stores. Using this technique, backtracking is replaced by a dropping operation which consists in removing the local store and reclaiming the allocated memory. These two implementations offer drastically different performances. In a railing system, accessing the database or the value of a variable is very fast, but switching from one branch of the tree of hypothesis to another is very costly since it requires backtracking. In encapsulated systems, there is no backtracking but this improvement is balanced by the cost induced by the management of local stores.

On parallel architectures, there is no such optimal implementation as the trailing mechanism which is fundamentally sequential (two agent must not interfere on the trailing stack). There exist different solutions to help parallelize logic languages and their derivatives.

In *Prolog and in CLP languages*, various execution models have been proposed. The first one relies on **and-parallelism**[10, 16, 17]. It consists in exploring subgoals in parallel. This technique is interesting because it is compatible with the trailing mechanism since only one node of the tree of hypotheses is explored at a time. But it requires some control since two cooperative workers may lead to inconsistent computations. Two possible execution models have been proposed, *Independent and-parallelism*[9, 10, 17] which parallelize independent goals (goals which do not share variables) and *Stream and-parallelism*[23] which orders subgoals in a producer-consumer way.

The second execution model relies on **or-parallelism** [2, 22, 1, 8, 7, 4, 3, 28, 6]. It consists exploring different branches of a choice point in parallel. In this case, since multiple agents coexist, the same variable may be used in different places with different values. Thus the system has to cope with multiply bound variables and multiple environment[5].

Concurrent Constraint Programming[19, 18, 12, 24] is an elegant merging of CLP and of Concurrent programming. It allows a better control of the execution of processes and it incorporates parallel composition in its design. In this paradigm, the notion of independent and-parallelism and streamed and-parallelism is irrelevant as in CLP languages with a parallel $\wedge$ operator. And parallelism is explicit in CCLP languages. But the availability of both parallel composition and non-determinism forbids an implementation based on trailing since nothing can restrict an agent from backtracking while another is pushing some changes on the trailing stack. Thus the implementation of non-determinism has to use methods inherited from or-parallel languages. For instance, AKL[12] and Oz[13] use techniques inherited from or-parallelism languages.

The $\texttt{CC}(\mathcal{M})$ [14, 15] project is aimed at a generic implementation of the Concurrent Constraint class of languages[19, 18]. The $\texttt{CC}(\mathcal{M})$ language is a kernel language and constraint domains are implemented as libraries of this language. The $\texttt{CC}(\mathcal{M})$ language is designed as a concurrent object language. The contribution of this article is that the implementation of or-parallelism in the $\texttt{CC}(\mathcal{M})$ project relies on hardware functionalities (direct programmation of the Memory Management Unit of the CPU) for helping and speeding up the management of multiple environment. Since, with these functionalities, we are able to manage local stores for agents in an efficient way both for accessing variable values and for task switching.

The rest of the paper is sectioned if the following way. In section 2., we will define a constraint domain and the CC class of languages. Section 3. will present the state of the art in the implementation of or-parallelism. In section 4., we will give an overview of the $\texttt{CC}(\mathcal{M})$ project, its motivations and its multi-layers architecture. Section 5. will focus on the implementation of or-parallelism within the $\texttt{CC}(\mathcal{M})$ language and its use in the implementation of or-parallelism for any $\texttt{CC(X)}$ language. Section 6. will then justify the implementation choices by comparing the resource consumptions of the management of or-parallelism between the $\texttt{CC}(\mathcal{M})$ system and other existing systems. It will then present a small running application which will show the scalability of the $\texttt{CC}(\mathcal{M})$ language. We will then conclude and present future direction of research.

2. Preliminaries

In this article, $\mathcal{V}$ will denote a finite or countable set of variables. Elements of this set will be written using capital letters X or Y.

A *Constraint Domain*[19, 18] is a structure $\langle X, \vdash \rangle$ where $\vdash$ is a monotonic entailment relation. This structure X is closed under existential quantification and conjunction. It contains two distinguished elements **true** and **false**.

We give the constructors of deterministic CC.

Deterministic CC:

Program	::=	Decl . Agent	*Declaration and initial agent*
Decl	::=	Decl . Decl	*List of declarations*
	|	p :: Agent	*Declaration of a function*
	|	.	*Empty declaration*
Agent	::=	*tell* (c)	*Adding of a constraint*
	|	*ask* (c) $\rightarrow$ Agent	*Asking of a constraint*
	|	Agent $\|\|$ Agent	*Parallel composition*
	|	Agent $\oplus$ Agent	*Non-determinism*
	|	$\exists_X$ Agent	*Local variable*
	|	p(X)	*Function call*
	|	True	*Dummy agent*

We will simplify the notations by using $c\downarrow$ instead of *tell* (c) and by using $c \rightarrow$ A instead of *ask* $(c) \rightarrow$ A.

Let us briefly sketch the semantics of these operators. $c\downarrow$ first checks if the constraint c is compatible with the current store. In this case, the constraint c is added to the store. Otherwise, the *tell* fails and nothing is added to the store. This is done atomicaly. $c \rightarrow$ A will check if either c or $\neg$(c) is entailed by the store. If c is entailed, the *ask* is reduced to the agent A. If $\neg c$ is entailed by the store, the *ask* fails. If neither c not $\neg c$ is entailed by the store, the *ask* stops until either c or $\neg c$ is entailed. The semantic of *parallel composition* is clear. *Non-determinism* implies independent computations, and local stores. *Local variables* imply a notion of hiding as a local variable is independent from the rest of the store. When a *function call* is evaluated, it is reduced to the agent body in the declaration of the called function.

3. Related Works

3.1. Two sequential implementation schemes of non-determinism

The notion of non-determinism implies the ability to try a hypothesis and then to restore the values of the variables to their former state before trying another hypothesis. On uni-processor sequential computers, there exists two major implementation scheme for non-determinism: *trailing* and *encapsulated search.*

In a trailing implementation with cppying , there exists a unique store. When the resolution of a given goal implies a choice point, a marker is positioned on a trailing stack and every time a change happens in the database, a couple $\langle$variable, old value$\rangle$ is pushed on the stack. Thus, when the program needs to restore the old state, it just pops old values and variables from the trailing stack and overwrites the new values with the old ones.

In an encapsulated search implementation or backtracking without copying, the idea is different. Instead of storing negative information (from the current store to an old store), one stores positive information (from the old store to the the new one). As a consequence, each choice point contains elements like $\langle$variable, new value$\rangle$. In fact, the store is replaced by the original store (at the beginning of the computation) and by a list of choice points containing the new values for the local store.

The performances of these two implementation schemes are very different. In a trailing implementation, accessing the variables is very fast (in constant time in fact) but backtracking time is proportional to the number of changes which happened after the last choice point. On the contrary, in a encapsulated search implementation, there is no backtracking, just a dropping operation, but reading the value of a variable may require accessing a arbitrarily long ascending chain of list of differences.

3.2. Parallelization of trailing and encapsulated search

Many proposals have been made to implement or-parallelism for logic languages. These proposal were primarly motivated by hardware considerations. There are basically two kind of parallel computers which are *shared* memory and *non shared* memory computers. From these two categories of computer derives two categories of parallel implementation.

The first category relates to **non-shared** memory computers. It involves either *copying* of data when a new agent is launched or *recomputing* every changes which appeared before the fork. The advantage of such models is that the problem of multiply bound variables is completely solved by the memory *gap* between processors. Since each processors has its own memory completely separated from the rest of the computer memory, its own references to variables cannot interfere with references on other processors. But the cost of launching a new agent is very high because the whole data needed by the agent are to be copied from the parent agent to the child agent.

Conery's *closed environments*[8] reduce the copying cost by reorganizing the stack frame and by keeping every needed information on the same stack frame. This is achieved by a *closing* operation which implies a selective copying of the stack after each call. Ali's *BC-machine*[1] do not use this algorithm and incrementally copy parts of the WAM stacks when creating a new worker. Clocksin's *Delphi model*[7] is a model using recomputation.

In these models, the good aspect is that the execution of an agent is as efficient as in a sequential implementation but the cost of creating a new agent is very high as it requires copying or recomputing the whole database.

The second category of models relates to **shared** memory computers. In such models, since the difficulty of representing multiple bindings to the same variable cannot be solved by hardware considerations, each agent must be able to find the correct instance of any variable. The alternative here is whether of not each agent maintains a full set of references to the correct values of every variables.

If the answer is no then each agent stores the differences which happened during its own computation. In this case, the usual solution is that each agent has its own hash table which stores the results of local computation. This approach offers a fast task switching overhead since no extra computation is required during the context switch.Bbut it slows down writing to a variable since this is implemented by a writing to an hash table. It slows down even more the reading a variable value since this may require the exploration of any number of ascending hash-windows in order to find the exact location of the variable definition. We will call these models the *hashing* models. Here are some examples: works by Ciepielewski and Haridi[5, 6], the Argonne model [4], the PEPSys system[3], AKL[12] and Oz[24, 25].

If the answer is yes, then reading or writing the value of a variable requires only two more indirections (one to the current process and one to the variable array of this process) than direct access. This is a lot faster than accessing the hash table. But this set of references has to be maintained during the context switch. We will call these models the *vector* models. Various implementations have been proposed. In the SRI model[27], there is one array of references for the whole system. This model suffers from a very high task switching cost since it requires following the path from the old task to the new one and removing references as we go up in the tree[1] and adding references as

[1]the root is at the top of the tree

we go down. The Muse model[2] or the ACE model[16, 17] is very different. Each node contains the memory segment which differs from their parent node, thus task switching involves overwriting the stack. This could be worse than the SRI model but selective copying reduces this overhead. And since variables keep the same address, constraint handling is as fast as sequential prolog. Another model is the *Version Vector model* where each processor has one slot per variable. This model has worse memory performance and has never been implemented.

In these *shared memory* models, it is difficult to find the best implementation either accessing variables or context switching are very expensive. This is where the $\mathtt{CC}(\mathcal{M})$ implementation will be interesting as it will combine a low task switching overhead and a very fast access to variables.

4. The $\mathtt{CC}(\mathcal{M})$ project

4.1. Presentation of the $\mathtt{CC}(\mathcal{M})$ project

The notion of a generic implementation of constraint domain is not new and it can be found for instance in the AKL project[12]. In this project, a few overloaded functions implement the interface between the generic constraint solver and specific constraint domain. These functions deal with the installation, the removal, the entailment test, the garbaging ... of a constraint in the constraint domain. The approach in the $\mathtt{CC}(\mathcal{M})$ project is very similar.

The $\mathtt{CC}(\mathcal{M})$ project is a project aimed at the implementation of `CC(X)` languages where `X` is an admissible constraint domain as defined by Saraswat. In order to implement different constraint domains and possibly to mix different constraint solvers within the same application, the $\mathtt{CC}(\mathcal{M})$ project relies on a kernel language. This language should present two different aspects. The first one is that it must encode complex structures representing constraints. Our choice is that it will be an object-oriented language. As object languages offer a class hierarchy to encode complex objects and overloading to reuse generic parts of code, an object language is a good choice for the implementation of generic constraint solvers. Moreover, this language must offer the same primitives of the `CC` class of languages, i.e. parallel composition, non-determinism.

Therefore the kernel language should be a *concurrent object language*.

As shown in a previous paper[14], implementing a constraint domain over an object-like language is a multi-stage process. This is summarized in figure 2.

$$\begin{tikzcd} X \arrow[r, "(1)"'] & (X,\vdash) \arrow[r, "(2)"'] & \mathtt{CC}(X) \\ \mathcal{M} \arrow[u, "\text{representation}"'] \arrow[r, "\mathtt{CC}"'] & \mathtt{CC}(\mathcal{M}) \arrow[ur, dashed, "(3)"'] & \end{tikzcd}$$

Figure 1: Implementation scheme of $\mathtt{CC}(X)$

The first stage is the encoding of constraints in a class structure. This is the *representation* arrow. Then the implementor needs to program *entailment* over this new structure. This is the (1) arrow of the figure. The rest is automatic, adding the `CC` constructs to $(X, \vdash)$ is just using the same constructs in `CC`$(\mathcal{M})$. Thus, with minimal efforts, we are able to implement `CC` (X) over `CC`$(\mathcal{M})$. This is the (3) arrow.

The architecture of `CC`$(\mathcal{M})$ is based on three independent layers.

1. The first layer implements the interface to the system resources. It provides parallelism (real or simulated) and a database implementation (internal or external). This layer integrates general C libraries (QuickThread provides the thread library), the external database being any stand-alone database.
2. The second one implements the `CC`$(\mathcal{M})$ primitives (*ask*, *tell*, parallel composition, or-parallelism ...). It is programmed in C and interacts with the previous level in two ways: first it issues request to the C libraries and second, it shares with the parallelism libraries the internal description of an agent.
3. The third one provides `CC`$(\mathcal{M})$ with various extensions. These libraries implement constraint domains, an optimizing compiler, ...

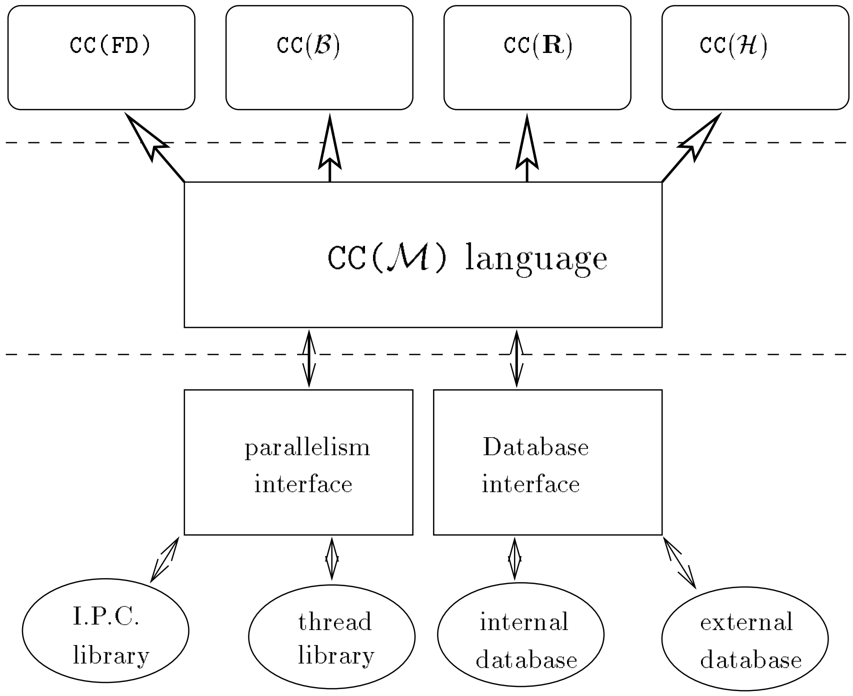

Figure 2: The `CC`$(\mathcal{M})$ architecture

4.2. Overview of the `CC`$(\mathcal{M})$ language

This section briefly presents the first two layers of the `CC`$(\mathcal{M})$ architecture.

4.2.1. Our memory model

Works on *non shared* memory models and *vectors* models have demonstrated many facts. First of all, it is inefficient to duplicate the whole memory during an agent creation. This was shown by the competitivity of models using recomputation on non shared memory computers. Moreover, the size of the area involved during a choice point is critical both from the point of view of memory consumption and from that of time consumption during an agent creation and during a task switching.

Conery's *closed environments*[8] have shown that it was worthy reorganizing the stack after each function call in order to keep the mutable data area as small and as compact as possible. In the ACE system[16], selective copying has shown that keeping variable access as fast as in sequential prolog is profitable even if it means overwriting the stack during each context switch.

The $\mathtt{CC}(\mathcal{M})$ memory model is inspired by all these experiences. But with a radical approach. Instead of examining the program code in order to find the mutable memory area, the $\mathtt{CC}(\mathcal{M})$ language implements a two level memory model (figure 3) consisting of

1. A common memory segment shared by all agents. It contains the system definition, the functions and every ground terms of the database.

2. A pool of local memory segment. When an agent becomes active, the local memory segment of the new agent is installed at the *same* address. This address is defined once for all in the system, thus every local memory segment appears at the same address.

Thus it is up to the programmer to define which variables are global and which are local. This approach has many benefits: It allows the mutable memory area to be as small as possible; it also allows some shared variables between agents[2].

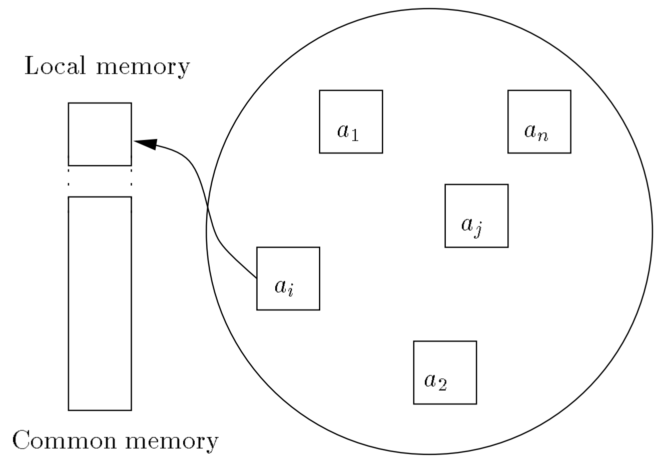

Figure 3: The $\mathtt{CC}(\mathcal{M})$ memory model

[2]This possibility is used when transforming the *branch and bound* algorithm into a *parallel cost* algorithm

Then the CC($\mathcal{M}$) system runs a preemptive scheduler which executes the following imperative code:

```
New <- NextThread()        (* Get next waiting thread  *)
if (New != Nil) then       (* New = Nil -> No waiting thread *)
   Old <- CurrentThread()  (* Old <- preempted thread *)
   EnqueueThread(Old)      (* Put back thread on pool *)
   InstallLocalMemory(New) (* arrow on figure *)
   Switch(Old, New)        (* switch from Old to New *)
   Continue(New)           (* Resume execution of New *)
endif                      (* Never reached *)
```

4.2.2. Classes and objects in the CC($\mathcal{M}$) *language*

Class and objects can be defined in the CC($\mathcal{M}$) language with the following syntax.

```
<new_class_name>  <: <superclass_name> ( feature : type , ...)
<new_object_name> :: [local] <class_name> ( feature <- value, ...)
```

The optional `local` modifier specifies that the next term is to be allocated on the local heap and not on the global one.

This `local` modifier is of course available each time some memory is allocated (variables, anonymous objects, arrays ...).

When comparing $\mathcal{M}$ with ψ-terms, it turns out that $\mathcal{M}$ implements a two level heirarchy (classes - objects) where ψ-terms implements a single level hierarchy (there is no distinction between classes and objects). Moreover, $\mathcal{M}$ is typed (each attribute of each class has a type decided at compile time) and ψ-terms are totally untyped. Apart from these differences, the two systems offer quite the same expressive power.

4.2.3. The parallelism primitives of CC($\mathcal{M}$)

These primitives relate to the calculus of processes and threads:

```
detach [local] <function call>
<variable> <- catch [local] <function call>
raise <exception>

CurrentThread ()          ThreadAbort ()
ThreadSleep ()            RestartThread ()
```

`detach` evaluates the function call in parallel. `catch` does the same, but it also puts a signal handler which catches exceptions raised by the `raise` instruction and returns the raised exception. The `local` modifier specifies if the local memory is to be duplicated during the fork.

`ThreadSleep` freezes the current thread which can be restarted by the `RestartThread` primitive.

5. Implementation of or-parallelism

5.1. The memory management of a workstation

In this section, we describe the functionment of the M.M.U. in a computer, how it can be accessed.

5.1.1. *A brief presentation of the M.M.U.*

On Unix workstations, one important aspect of the operating system is virtual memory. This means that each process lives in its own memory space and cannot see anything else. This behaviour is the key of general protection, identification and rights since a process is trapped in a virtual space without the possibility to escape[3].

This feature is essentially implemented by the hardware which provides the Memory Management Unit (MMU), a chipset located on the memory bus, which performs most of the job as described in figure 4.

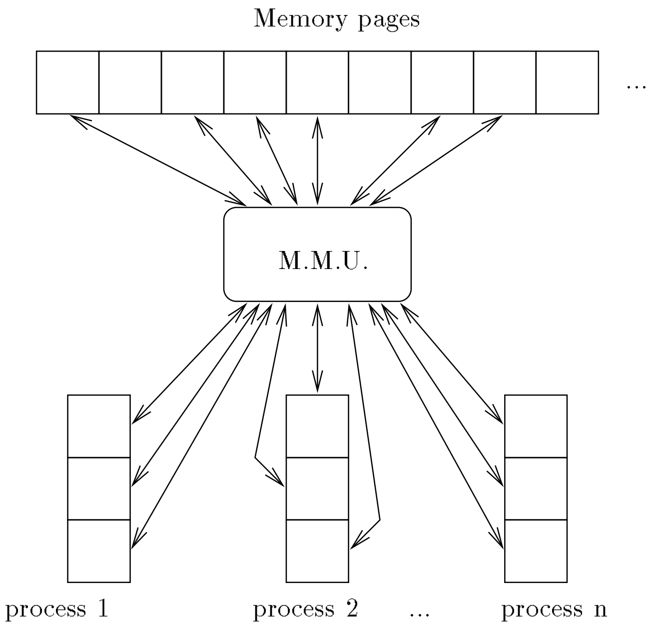

Figure 4: Description of the M.M.U.

Thus, the MMU uses translation tables and maps virtual memory pages to physical ones. This scheme allows the kernel to (1) map different pages at the same address (for different processes) and (2) to share a physical page among processes (as used in shared libraries).

5.1.2. *O.S. and I.P.C. libraries*

A common library in modern unices is the Inter Process Communication (IPC) library which offers various system calls to implement shared memory, semaphores and messages between processes. We will mainly use the first part of the library.

This library offers three primitives:

`shmget ( key, size, shmflag )`

[3]This is of course an utopia

```
shmat  ( shmid, shmaddr, shmflag )
shmdt  ( shmaddr )
```

`shmget()` creates a new memory segment of length `size` bytes and returns and identifier `shmid`. This identifier is then used by `shmat()` to map the designated memory segment at the virtual address `shmaddr`. The segment can be unmapped with the `shmdt()` system call.[4].

These system calls directly modify the translation pages of the MMU.

5.2. Implementation of the CC($\mathcal{M}$) memory model

5.2.1. Implementation using the I.P.C. library

Using the I.P.C. primitives, we are able to allocate a small memory segment for every agent. This memory segment is mapped at a predefinite address when a agent becomes active. Thus installing the local memory of an agent consists in removing an old local memory segment is done by a **shmdt** system call and then using the **shmdt** system call to map a previously allocated segment at the same address.

Therefore, we are able to implement the ACE model or the Muse model without actually copying the local data.

5.2.2. Distribution of the CC($\mathcal{M}$) *execution model*

As shown in figure 1 and figure 3, CC($\mathcal{M}$) relies on a multi-threading library. But using the I.P.C. facilities for shared memory between processors, the common memory segment can be shared between processes. Furthermore, since the local memory pool is visible from every process, threads (and local memory segments) can be executed by any process. Thus threads can migrate easily from process to process without any additionnal cost. And since the operating system and the hardware allow it, processes can be executed by different processors. The result is then true parallelism and the use of the whole computing power of the workstation.

Each process then runs a preemptive round-robin scheduler called by a timer whose code is the previous scheduler code modified to take into account the I.P.C. primitives and the distribution of threads.

```
New <- GetNewThread()     (* Get Waiting thread from pool *)
if (New != Nil) then      (* New = Nil -> No waiting thread *)
   Old <- CurrentThread() (* Old <- preempted thread *)
   UnmapLocalMemory(Old)  (* arrow (2)  on figure *)
   PutBackThread(Old)     (* Put back thread on pool *)
   MapLocalMemory(New)    (* arrow (1) on figure *)
   Switch(Old, New)       (* switch from Old to New *)
   Continue(New)          (* Resume execution of New *)
endif                     (* Never reached *)
```

This scheme which implements the migration of threads on different processes and processors is summarized in Figure 5.

[4]Further details are available by `man shmop` and `man shmget`

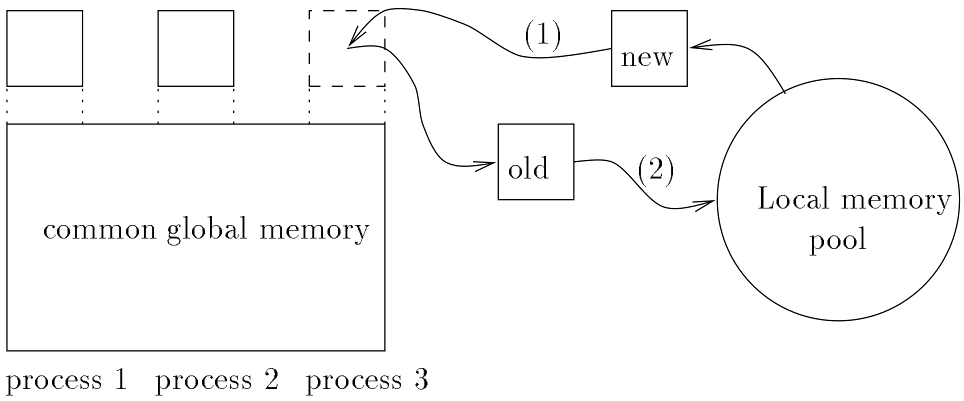

Figure 5: multi-processing environment

5.3. Implementing multiple environment

In order to implement multiple environments, we need to store logical and constraint variables in the local memory.

Internally, such a variable is a reference to a C structure. In order to keep the local memory as small as possible, only the last reference to the variable is to be local. This last reference is in fact the pointer to the C structure itself. The result is shown in figure 6.

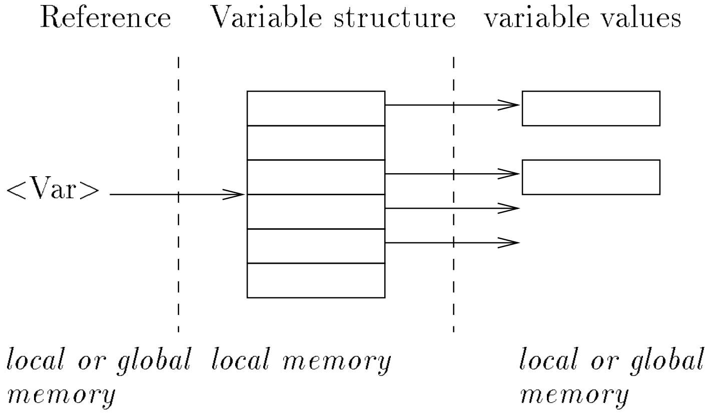

Figure 6: A variable implementation

5.4. Or workers and independent And workers

Using the `CC(`$\mathcal{M}$`)` primitives, launching a new cooperative agent is simply done by using the `detach` instruction.

Launching an Or-Parallel worker is slighty more complex. The implementation changes if the programmer wants every solutions or just the first one.

In case only the first solution is wanted, the following code is enough to implement a choice point:

```
Result <- catch ChoiceFunction ()
```

where the `ChoiceFunction` detaches every branches of the choice point.

```
foreach H in ChoicePoint
  detach local EvalBranch(H)
```

Each branch `EvalBranch` is then implemented as follows

```
(* CommitHypothesis *)
...
(* rest of the agent code *)
...
(* in case of success *)
raise ResultException ( result <- <found result>)

(* in case of failure *)
ThreadAbort ()
```

Thus `Result` is assigned the first found solution and `Nil` when every agent has failed, since a `catch` instruction returns a `Nil` value when all the sub-processes have aborted.

To keep all the solutions , the implementor needs a shared variable where every agent will enqueue its result before aborting.

6. Comparison with other systems

In this section, we analyze the resources consumption of our execution model and we compare it to other existing models.

6.1. Performance considerations

6.1.1. Typical benchmarks

We have run different benchmarks to evaluate the average cost of basic operations used in constraint programming languages. These costs will help us in the comparison of the `CC(`$\mathcal{M}$`)` project with other systems. These tests were made on a Sun SparcStation 5 with code compiled with `gcc` without any optimization.

The results are the followings:

Operation	costs
read/write from and to memory, no indirections	0.8 μs
read/write, with 1 indirection	2.4 μs
read/write, with 2 indirections	3.6 μs
unmap a memory segment, map a new one	178 μs
read a value from a partially filled hash table	20 μs
write a value to a partially filled hash table without test	8 μs
copy a memory block of 4096 bytes	204 μs

6.1.2. Comparison with vectors models

The implementation of or-parallelism requires three basic operations which are the agent creation, agent switching and accessing (reading or writing) a variable. For vector models[27, 2, 16, 17], an important notion is the distance between two tasks. Accessing variable is as fast in vectors models than in the CC($\mathcal{M}$) language (In fact, it is a bit faster in the CC($\mathcal{M}$) language because there is one less indirection) and the cost of copying a 4 Ko memory segment is comparable with the cost of unmapping-mapping a shared memory segment. Furthermore the overhead induced by the context switch is independent of the mutable area size in the CC($\mathcal{M}$) project and linear in vectors models like the ACE system[16, 17].

6.1.3. Comparison with hashing models

When comparing the CC($\mathcal{M}$) language to hashing models[5, 6, 4, 3, 12, 24, 25], the access time of a variable is so high that it justifies easily the overhead of the context switch. In fact, 9 (178 / 20) read operations (and read operations which succeed by using the first hash window) have the same cost than the unmap-map operation. And it is clear that between two context switches, the program will perform much more than 9 read operations. Therefore, the CC($\mathcal{M}$) implementation is much more efficient than implementations of hashing models.

6.2. A running CC(FD) example

We have tested our language on the bridge problem. Using a naive implementation, we have obtained the following results for the optimum (104 days) and its proof of optimality.

number of processors	1	2	3	4
elapsed time	1mn 32 s	48 s	33 s	26 s
speedup	1	1,91	2,79	3,53

This preliminary benchmarks show that the speed of the CC($\mathcal{M}$) language is correct (1mn for finding the solution of the bridge is a very good speed in a naive implementation which explores more than 45000 backtracks) and that the CC($\mathcal{M}$) language scales up well.

7. Conclusion

Using low-level techniques, we have implemented in the CC($\mathcal{M}$) project a hybrid system which inherits from many different implementation schemes of or-parallelism. The result is very promising, accessing variables is as fast in the CC($\mathcal{M}$) language as in fast prolog implementations (even the address of given local objects may be decided at compile time, to help the C compiler to optimize more the resulting code). Furthermore, the CC($\mathcal{M}$) memory model allows many interesting extensions of the standard prolog resolutions. The extensions come from the mixing of local variables and shared variables. In fact, this memory model is a good basis for experimenting cooperative resolution of problems, constructive negation, parallel pruning .... But this implementation does not solve the limited resources problem (how can we parallelize non determinism when the number of agents is bounded) and we know this is the biggest difficulties when implementing or-parallelism. For instance, every agent can be suspended waiting for another agent to release recources even if this is not possible. For the bridge problem, a booking algorithm was an easy solution but the general case is very challenging.

Acknowledgements

I would like to thank François Fages, François Laburthe and my brother Jean Philippe for their help and encouragements.

References

[1] K. Ali. Or-parallel execution of prolog on BC-Machine. In *Fifth International Conference of Logic Programming*, pages 1531–1545. MIT Press, 1988.

[2] K.A.M. Ali and R. Karlsson. Full prolog and scheduling or-parallelism in muse. In *International Journal of Parralel Programming*, number 19 in 6, page 445:475, 1991.

[3] U. baron, J.C. de Kergommeaux, H. Hailperin, M. Ratcliffe, P. Robert, and J.C. Syre nad H. Westphal. The paraller ECRC prolog system PEPSys: an overview and evaluation results. In *International Conference on Fith Generation Computer Systems*, pages 841–849, Tokyo, 1988. ICOT.

[4] R. Butler, E.L. Lusk, R. Olson, and R.A. Overbeek. ANLWAM: A parallel implementation of the warren abstract machine. Technical report, MCS division, Argonne National Laboratory, 1986.

[5] A. Ciepielewski and S. Haridi. Storage models for or-paralles execution of logic programs. Technical report, The Royal Institute of Technology, Stockholm, 1983.

[6] A. Ciepielewski, S. Haridi, and B. Hausman. Or-parallel prolog on shared memory multiprocessors. *Journal of Logic Programming*, 7:125:147, 1989.

[7] W. Clocksin. Principles of the DelPhi paralle inference machine. *COmputer Journal*, 30(5):386–392, 1987.

[8] J.S. Conery. Binding environments for parallel logic programs in nonshared memory multiprocessors. In *Proceedings of the 1987 Symposium on Logic Programming* [11], pages 457–467.

[9] D. DeGroot. Restricted AND-parallelism and side effects. In *International Conference on 5th Generation Computer Systems*, San Francisco, August 1987. IEEE Computer Society.

[10] M. Hermenegildo and K. Greene. &-prolog and its performance: Exploiting independant and-parallelism. In *1990 International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[11] IEEE. *Proceedings of the 1987 Symposium on Logic Programming*, San Francisco, August - September 1987. Computer Society Press.

[12] Sverker Janson. *AKL, A Multiparadigm Programming Language.* SICS, Uppsala University, 1994.

[13] Michael Mehl, Ralf Scheidhauer, and Christian Schulte. An Abstract Machine for Oz. Research Report RR-95-08, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, D66123 Saarbrücken, Germany, June 1995. Also in: *Proceedings of PLILP'95*, Springer-Verlag, LNCS, Utrecht, The Netherlands.

[14] Laurent Perron. cc($\mathcal{M}$), a kernel for implementing cc languages. In *Proc. of CCP'95*, 1995.

[15] Laurent Perron. A concurrent constraint kernel language based on messages. In *Proc. of ICLP'95*, 1995. Poster.

[16] E. Pontelli, G. Gupta, and M. Hermenegildo. &ACE:a high performance parallel prolog system. In *IIPS 95*, Santa Barbara, CA, 1995. IEEE Computer Society.

[17] Enrico Pontelli, Goral Gupta, and Manuel Hermenegildo. &ACE, the and-parallel component of ACE (a progess report on ACE). In Jonas Barlklund, Bharat Jayaraman, and Jiro Tanaka, editors, *Parallel and Datra Parallel Execution of Logic Programming*, volume 78, pages 65–78. UPMAIL, June 1994.

[18] Vijay Saraswat. Concurrent constraint programming. In *seventeenth ACM Symposium on Principles of Programming Languages*, 1990.

[19] Vijay Saraswat. *Concurrent Constraint Programming Languages.* MIT Press, 1990.

[20] Christian Schulte and Gert Smolka. Encapsulated search in higher-order concurrent constraint programming. In Maurice Bruynooghe, editor, *Logic Programming: Proceedings of the 1994 International Symposium*, pages 505–520, Ithaca, New York, USA, November 1994. MIT-Press.

[21] Christian Schulte, Gert Smolka, and Jörg Würtz. Encapsulated search and constraint programming in Oz. In A.H. Borning, editor, *Second Workshop on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, vol. 874, pages 134–150, Orcas Island, Washington, USA, May 1994. Springer-Verlag.

[22] E. Shapiro. An or-parallel execution algorithm for prolog and its FCP implementation. In *Fourth International Comference of Logic Programming*, pages 311–317. MIT Press, 1987.

[23] K. Shen. *Studies in And/Or Parallelism in Prolog.* PhD thesis, University of Cambridge, 1992.

[24] Gert Smolka. A calculus for higher-order concurrent constraint programming with deep guards. Research Report RR-94-03, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, February 1994.

[25] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[26] D.H.D. Warren. An abstract prolog instruction set. Technical Report 209, SRI International, 1983.

[27] D.H.D. Warren. Or-parralel execution models of prolog. In *TATSOFT'87*, pages 243,259. Springer Verlag, 1987.

[28] D.H.D. Warren. The SRI model for or-parallel execution of Prolog: Abstract design and implementation. In *Proceedings of the 1987 Symposium on Logic Programming* [11], pages 92–102.