A Loosely Synchronized Execution Model for a Simple Data-Parallel Language (Extended Abstract)

Yann Le Guyadec[2], Emmanuel Melin[1], Bruno Raffin[1]
Xavier Rebeuf[1] and Bernard Virot[1]*

[1] LIFO - IIIA Université d'Orléans
4, rue Léonard De Vinci - BP 6759 F-45067 Orléans Cedex 02 - FRANCE
[2] Currently affiliated to VALORIA 8 rue Montaigne - BP 1104
F-56014 Vannes - FRANCE

Abstract. Data-parallel languages offer a programming model structured and easy to understand. The challenge consists in taking advantage of the power of present parallel architectures by a compilation process allowing to reduce the number and the complexity of synchronizations. In this paper, we clearly separate the synchronous programming model from the asynchronous execution model by the way of a translation from a synchronous data-parallel programming language into an asynchronous target language. The synchronous data-parallel programming language allows to temporarily mask local computations. The asynchronous target language handles explicit and partial synchronizations through the use of structural clocks.

In the area of parallel programming, a crucial step has been performed with the emergence of the data-parallel programming model. Parallelism is expressed by means of data-types which are promoted from scalars to vectors. Its leads to a distribution mechanism which maps components of data-types over a network of *virtual processors*. From the programmer's point of view, a program is a sequential composition of operations (local computations or global rearrangements) which are applied to restricted parts of large data-structures. The distribution mechanism is easily scalable and the model is adapted to many scientific applications. The challenge consists in taking advantage of the power of present parallel architectures by a compilation process allowing to reduce the number and the complexity of synchronizations. The effort is then transferred to the compiler which has to fill the gap between the abstract *synchronous* and *centralized* programming model and an *asynchronous* and *distributed* execution model that depends on the target architecture. Data-parallel compilers have to perform complex data-flow analysis to manage data distribution and desynchronizations between processors, especially in loop structures. The efficiency of the produced

* Authors contact: Bernard Virot (`Bernard.Virot@lifo.univ-orleans.fr`). This work has been partly supported by the French CNRS Coordinated Research Program on Parallelism PRS.

code heavily depends on static informations relating to data dependences which are available at compile time. Automatic parallelizers can detect regular dependences especially in nested loops. When complex and irregular data-structures must be handled, explicit expression of dependences is mandatory in order to obtain automatic and efficient desynchronizations.

In existing data-parallel languages data-dependence is the default and only independence is made explicit (cf. the FORALL and INDEPENDENT constructs in HPF [2]). We propose a kernel data-parallel language called $\mathcal{TML}$ (which stands for Twin Memory Language) which purpose is to offer both a synchronous programming model and an asynchronous execution model. In this language independence is the default and data dependence is made explicit in the syntax. To achieve this aim, we reuse the twin memory model, previously introduced for the language $\mathcal{D}$ [10], to temporarily mask local computations. The $\mathcal{TML}$ language yields a synchronous programming model, similar to classical data-parallel languages such as C^* [12], HyperC [9], Data–Parallel C [8] or $\mathcal{L}$ [1]. Taking advantage of the absence of implicit dependence in the language, our main contribution consists in showing that it is possible to perform an automatic desynchronization of $\mathcal{TML}$ programs. We propose an execution model relying on the translation of $\mathcal{TML}$ programs into the asynchronous and non-deterministic data-parallel language $\mathcal{SCL}$, previously introduced in [5, 7]. The desynchronization is based on a partial synchronization algorithm relying on structural clocks. We extend the $\mathcal{SCL}$ language with a wait instruction performing a point to point synchronization. Determinism of translated $\mathcal{SCL}$ programs yields the correctness of the translation function.

In this paper, we first introduce the data-parallel model. We give an informal description of the $\mathcal{TML}$ language. Next, we briefly recall the semantics of the $\mathcal{SCL}$ language. The translation of $\mathcal{TML}$ towards $\mathcal{SCL}$ is presented in the third section. Finally we illustrate the benefit of our approach with an example.

1 The $\mathcal{TML}$ language

We turn now to the description of the $\mathcal{TML}$ language (which stands for Twin Memory Language) which aim is to offer both a synchronous programming model and an asynchronous execution model.

1.1 The data-parallel programming model

In the data-parallel programming model, basic objects are arrays with parallel access, called vectors. They are denoted with uppercase initial letters. The component of the parallel variable X located at index u is denoted by $X|_u$. Expressions are evaluated by applying operators componentwise to parallel values. Each action is associated with the set of array indices to which it is applied. This set is called the *activity context* or *extent of parallelism*. Indices to which an operation is applied are called *active*, whereas others are *idle*. Legal expressions are usual *pure* expressions, i.e. without side effect, like the definition of *pure* functions in HPF [2]. The value of a pure expression at each index u only depends on the values of variable components at index u. We make use of a special vector constant called $This$. The value of its component at each index u is the value u itself: $This|_u = u$.

1.2 The twin memory model

$\mathcal{TML}$ reuses the twin memory management previously introduced for the language $\mathcal{D}$ [10]. The model relies on an abstract machine where each index u owns a *private memory* containing the components of all vector variables at index u, and a *public memory* containing a copy of the private memory of the same index. Private memories are dedicated to local computations (evaluation of *pure* expressions or assignments) whereas public memories are referred by communications. Public memory updates are explicitly executed through a specific instruction (dump). As only the dump instruction makes a local transition visible for other indices, this mechanism suppresses all implicit dependences.

1.3 Informal description of the $\mathcal{TML}$ language

We turn now to an informal description of the $\mathcal{TML}$ language.

No action: skip. This instruction does nothing and just terminates.

Memory copy: dump. All *active* indices copy their private memory into their public one.

Assignment: $X := E$. For each active index u, the component $X|_u$ of the *private* memory is updated with the local value $E|_u$. As E is a *pure* expression, $E|_u$ is evaluated in the *private* memory of the index u.

Communication: get X from A into Y. The address A is a *pure* expression. Each active index u evaluates $A|_u$ in its private memory. Then, u fetches the remote value of X in the *public* memory of the index $A|_u$. The result is assigned to the component $Y|_u$, in the *private* memory of the index u.

Sequence: $S; T$. At the end of the execution of the last instruction of S, the execution of the instructions of T starts.

Conditioning: where B do S elsewhere T end. This construct splits the set of currently active indices into two disjoint subsets depending on the value of the *pure* expression B. The active indices evaluating B to $true$ execute S. *Next*, the other active indices execute T. This is the instruction if/else of MPL [3] or where/elsewhere of HyperC [9].

Iteration: loopwhere B do S end. Iteration is expressed by classical loop unfolding. Only the active indices evaluating the *pure* expression B to $true$ execute S with the new extent of parallelism. The other ones do not execute the loop. Note that the set of active indices may decrease at each loop iteration. An execution of a loop terminates if all active indices evaluate B to $false$. This is the instruction while of MPL [3] or the whilesomewhere of HyperC [9].

Example. In this $\mathcal{TML}$ program (Fig. 1), after the first instruction, all the indices copy their private memory in their public one. Then, the indices executing the first branch of the where/elsewhere construct execute a dump again. Hence, at the end of the execution, the indices for which $This > 3$ store the value 0 in their component of Y and the index 3 stores 1. The data dependences between indices occur only through the dump and the get instructions.

```
X := 0;
dump;
where (This < 3) do
  X := 1;
  dump
elsewhere
  get X from This - 1 into Y
end
```

Fig. 1. A $\mathcal{TML}$ program.

2 The target language $\mathcal{SCL}$

For the sake of completeness, in this section we briefly present the target language $\mathcal{SCL}$ previously introduced in [6, 7]. It yields an asynchronous and nondeterministic execution model.

2.1 Informal description

Let us turn now to an informal description of the $\mathcal{SCL}$ language. Like $\mathcal{TML}$, this language uses a twin memory management. From $\mathcal{TML}$, it reuses some instructions with the same semantics: assignment, skip, get, dump. We only present the other $\mathcal{SCL}$ instructions concerning sequence, loop, conditioning and synchronization.

Asynchronous sequence: $S; T$. Each index independently executes S and then T. Hence, the sequence yields only local synchronizations: an index may execute T before the other indices terminate the execution of S.

Concurrent conditioning: where B do S elsewhere T end. This statement is an asynchronous double-branched conditioning construct as found in some optimizations of HyperC [11]. It was also introduced in the $\mathcal{D}$ language [10] to express parallel execution of disjoint programs. The set of active indices is divided into two sets depending on the value of the pure expression B, which is evaluated in the private memories. An active index u which evaluates the expression $B|_u$ to $true$ (respectively $false$) executes the program S (respectively T). In contrast to the where/elsewhere statement of $\mathcal{TML}$, the two branches are executed concurrently.

Asynchronous loop: loopwhere B do S end. This is an asynchronous version of the $\mathcal{TML}$ instruction loopwhere. Each currently active index u repeats the computation of the block S while the pure expression $B|_u$ evaluates to $true$ in its private memory. Note that the set of active indices may decrease after each loop iteration. When an index terminates the loop, it goes on executing instructions that directly follow the loop without any implicit synchronization.

Waiting: wait A. The wait A statement is used to manage the consistency of public memory references by serializing instructions which interfere with this memory. The expression A is pure. The instruction wait A delays the index u which is executing it, until the target index $A|_u$ has stepped over all the instructions referencing public memory and preceding the wait instruction. Since where and loopwhere govern the activity, if the wait point is nested in such control structures, it becomes possible that v could never reach some preceding instruction referencing the public memory. In this case u can step over the wait point as soon as v is engaged in a different branch from u in a where/elsewhere structure, or as soon as v has terminated the current loopwhere executed by u. We call *meaningful* the instructions that play a part in the wait management, namely dump, get, where and loopwhere. To sum up, the instruction wait A can terminate if and only if the two following conditions **C1** and **C2** are satisfied.

- **C1 :** The index $A|_u$ has evaluated the boolean conditions of all enclosing where and loopwhere constructs.
- **C2 :** If the index $A|_u$ enters the innermost instruction block inside which u is waiting, it must have passed all *meaningful* statements coming before the wait instruction.

Remark. Note that an index may step over a waiting point even if the waited index needs to execute non-meaningful statements before reaching it.

Synchronization: wait_all. This instruction is a generalization of the wait instruction. An index terminates this instruction when it has performed a wait towards all other indices. Note that this instruction is a partial synchronization since some indices may not execute it.

2.2 Structural Clocks presentation

In order to formalize index positions we introduce *Structural Clocks* [7]. Each index owns a clock that encodes its current position, during program execution, with regard to the meaningful statements. An index position is defined by the meaningful control structures it is nested in, and by the last meaningful instruction executed in the innermost one. The *Structural Clock* t_u of an index u is expressed by a list of pairs. Each term of the list corresponds to a nesting level. Each pair (l, c) is composed of a label l and an instruction counter c. The counter c represents the number of meaningful instructions already executed in the corresponding instruction block. The label l is used to distinguish which branch of a where/elsewhere statement an index is inside. Lists are built up by popping or pushing pairs on their right hand side.

- When the index u executes a get or a dump instruction its clock $t_u = t(l, c)$ becomes $t_u = t(l, c+1)$.
- After evaluating the condition B of a where/elsewhere statement, if $B|_u$ is $true$ (resp. $false$) its clock $t_u = t$ becomes $t_u = t(1, 0)$ (resp. $t_u = t(2, 0)$). When the index u exits a where or a elsewhere branch, its clock $t_u = t(l, c)(m, d)$ becomes $t_u = t(l, c+1)$.

In a loopwhere, an index which does not compute an iteration directly exits the loop. Therefore, it turns out that we do not have to consider each iteration as a new nesting level for the structural clock list. We only push a new term on the list at the first loop iteration and next, we count instructions already executed in the loopwhere body.

- When the index u enters for the first time a loopwhere instruction, if the condition $B|_u$ evaluates to $true$, then its clock $t_u = t$ becomes $t_u = t(1, 0)$. Otherwise, its clock $t_u = t(l, c)$ becomes $t_u = t(l, c+1)$.
- If it has already executed at least one loop iteration, then if $B|_u = true$ its clock $t_u = t$ remains unchanged, otherwise $t_u = t(l, c)(1, d)$ becomes $t_u = t(l, c+1)$.

Remark. At the beginning of the execution, each index initializes its structural clock to $(0, 0)$.

Structural clock ordering. To order two structural clocks, we compare the local counters corresponding to the common instruction block of the innermost nested level. We so define this partial ordering as a lexicographical order based on partially ordered pairs.

Definition. An index u is said to be *later* than an other index v (denoted by $t_u \prec t_v$) if one of the following conditions holds :

- there exists t not empty such that $t_v = t_u\ t$;
- there exists $t^1,\ t^2,\ t^3$ (possibly empty) and $c_u,\ c_v,\ l$ such that $t_u = t^1(l, c_u)t^2,\ t_v = t^1(l, c_v)t^3$ and $c_u < c_v$.

Handling wait statements using structural clocks. Structural clocks yield a general mechanism to handle wait statements. The conditions **C1** and **C2** above are formalized by the following theorem.

Theorem 1. *Consider an index u with structural clock t_u that executes an instruction* wait A*. This index can terminate the wait statement if and only if the index $v = A|_u$ holds a structural clock t_v which satisfies $\neg(t_v \prec t_u)$.*

3 Translation from $\mathcal{TML}$ into $\mathcal{SCL}$

In this section, we show that it is possible to perform an automatic desynchronization of $\mathcal{TML}$ programs by the way of a translation function $\mathcal{F}$ from $\mathcal{TML}$ programs to $\mathcal{SCL}$ ones.

We introduce some necessary notations to explain the translation function. The predicate $Hasdump$ allows to know if a program S contains an instruction dump: $Hasdump(S) = true$ if S contains at least one dump, $Hasdump(S) = false$ otherwise. In the same way, the predicate $Hasget(S)$ denotes the presence of get instructions in the program S.

The function $\mathcal{F}$ (Fig. 2) must translate a program while preserving its semantics. We guarantee determinism using several mechanisms:

- To prevent wrong references, an index can perform a dump instruction only if there is no index later than it. Since all the indices can refer a dump, the function $\mathcal{F}$ introduces a partial synchronization wait_all before each instruction dump (cf. 1, Fig. 2).
- An index can perform a communication only if the referred index has updated its public memory. Moreover, the instruction get points out accurate index dependences. Therefore the function $\mathcal{F}$ introduces an instruction wait before each instruction get (cf. 2, Fig. 2).
- Since the two branches of a $\mathcal{SCL}$ conditioning structure are concurrent, a $\mathcal{TML}$ where/elsewhere can be directly replaced by a $\mathcal{SCL}$ where/elsewhere only if the two branches are independent. The translation is based on a simple syntactic analysis. Independence between the two branches is ensured if a get instruction does not occur in a branch whereas a dump instruction occurs in the other one (cf. 4, Fig. 2). If the analysis does not prove the independence, the $\mathcal{TML}$ where/elsewhere is replaced by a sequence of two $\mathcal{SCL}$ where/elsewhere. The former triggers the execution of the first branch of the $\mathcal{TML}$ where/elsewhere, whereas the latter is concerned by the second branch (cf. 3, Fig. 2).

We turn now to the correctness proof of the translation function. The intuitive idea is the following. The unique (synchronous) computation of a $\mathcal{TML}$ program S can be seen as a particular computation of the translated program $\mathcal{F}(S)$. Therefore, to ensure the correctness of the translation function, it is sufficient to prove that all computations of the translated program are equivalent, i.e. translated programs are deterministic.

Definition. A synchronous execution of a $\mathcal{SCL}$ program is a computation performing a global synchronization after the execution of each instruction.

The next theorem states that for the execution of a $\mathcal{TML}$ program S and the synchronous execution of $\mathcal{F}(S)$ are equivalent.

Theorem 2. *The execution of a $\mathcal{TML}$ program terminates if and only if the synchronous execution of $\mathcal{F}(S)$ terminates. If they both terminate, they yield the same final state for the twin memories.*

$\mathcal{TML}$ $\longrightarrow$	$\mathcal{SCL}$	
$\mathcal{F}(\mathsf{skip})$	skip	
$\mathcal{F}(X := E)$	$X := E$	
$\mathcal{F}(\mathsf{dump})$	wait_all; dump	(1)
$\mathcal{F}(\mathsf{get}\ X\ \mathsf{from}\ A\ \mathsf{into}\ Y)$	wait A; get X from A into Y	(2)
$\mathcal{F}(S;T)$	$\mathcal{F}(S);\mathcal{F}(T)$	
If $(Hasdump(S) \wedge Hasget(T)) \vee$ $(Hasget(S) \wedge Hasdump(T))$ *then* $\mathcal{F}\begin{pmatrix}\mathsf{where}\ B\ \mathsf{do}\ S\\ \mathsf{elsewhere}\ T\\ \mathsf{end}\end{pmatrix}$	$\begin{cases} Tmp := B;\\ \mathsf{where}\ Tmp\ \mathsf{do}\ \mathcal{F}(S)\\ \mathsf{elsewhere}\ \mathsf{skip}\\ \mathsf{end};\\ \mathsf{where}\ Tmp\ \mathsf{do\ skip}\\ \mathsf{elsewhere}\ \mathcal{F}(T)\\ \mathsf{end}\end{cases}$	(3)
otherwise $\mathcal{F}\begin{pmatrix}\mathsf{where}\ B\ \mathsf{do}\ S\\ \mathsf{elsewhere}\ T\\ \mathsf{end}\end{pmatrix}$	$\begin{cases}\mathsf{where}\ B\ \mathsf{do}\ \mathcal{F}(S)\\ \mathsf{elsewhere}\ \mathcal{F}(T)\\ \mathsf{end}\end{cases}$	(4)
$\mathcal{F}(\mathsf{loopwhere}\ B\ \mathsf{do}\ S\ \mathsf{end})$	loopwhere B do $\mathcal{F}(S)$ end	

Fig. 2. The translation function $\mathcal{F}$.

Theorem 3. *All $\mathcal{SCL}$ programs produced by the translation function $\mathcal{F}$ are deterministic.*

From the two previous theorems, we deduce that all the executions of a translated $\mathcal{TML}$ program always compute the expected result. The correctness of the translation function $\mathcal{F}$ is thus proven.

3.1 Example

The example (Fig. 3) illustrates the $\mathcal{TML}$ and $\mathcal{SCL}$ potentialities. Thanks to the synchronous programming model, the behavior of the $\mathcal{TML}$ program is simple. Initially each index u holds a value $X|_u$. First, it computes the maximum of the values belonging to the indices on its left side, by using a scan. Then, it computes an average between its result and its neighbor ones.

The translated $\mathcal{SCL}$ program is loosely synchronized. Overlapping of communications by computations can occur. For instance, the index 1 can execute the instruction of the line 15 (Fig. 3) while the index 2 refers its public memory through the instruction **get** of the line 7. But the index 1 must wait the index 2

line	*instruction*	*clock*
1	$Val := X;$	$(0,0)$
2	wait_all;	$(0,0)$
3	dump;	$(0,1)$
4	$I := 1;$	$(0,1)$
5	loopwhere $This - I > 0$ do	$(0,1)(1,3j)$
6	wait $This - I$;	$(0,1)(1,3j)$
7	get Val from $This - I$ into Y;	$(0,1)(1,3j+1)$
8	$Val := max(Val, Y);$	$(0,1)(1,3j+1)$
9	wait_all;	$(0,1)(1,3j+1)$
10	dump;	$(0,1)(1,3j+2)$
11	$I := I * 2$	$(0,1)(1,3j+2)$
12	end;	$(0,2)$
13	wait $This - 1$;	$(0,2)$
15	get Val from $This - 1$ into Aux_1;	$(0,3)$
16	wait $This + 1$;	$(0,3)$
17	get Val from $This + 1$ into Aux_2;	$(0,4)$
18	$Val := (Val + Aux_1 + Aux_2)/3$	$(0,4)$

Fig. 3. The second column displays a $\mathcal{TML}$ program and its $\mathcal{SCL}$ translation. For the sake of consistness, the new $\mathcal{SCL}$ instructions added by the translation are displayed on the right hand side. For each instruction, the structural clock an index owns, *when it has finished to execute it*, appears in the third column. The variable j denotes the number of loop iterations already executed. We consider an array of 2^n indices numbered from 1 to 2^n. The get instructions addressing a non existing index return the default value 0.

before it executes the instruction of the line 17. The structural clock mechanism forbids the index 1 to step over the wait of the line 16 since its structural clock $(0,3)$ is greater than the structural clock $(0,1)(1,0)$ of the index 2.

Remark. The partial synchronization wait_all (line 2, Fig. 3) introduced by the translation function is useless since in the program there is no communication before. A more precise syntactic analysis can permit to detect such configurations.

4 Conclusion

In this paper, we have presented a kernel data-parallel language $\mathcal{TML}$ relying on a twin memory management. It offers a synchronous data-parallel programming model where dependences between indices are explicit in the syntax and simple to express. The execution model based on a translation into $\mathcal{SCL}$ is asynchronous, and thereby adapted to present MIMD architectures.

The translation suppresses some useless synchronizations and preserves the initial semantics of $\mathcal{TML}$ programs by only imposing waiting points. It is possible to take advantage of more syntactic information to reduce synchronization requirements [5]. We also provide formal semantics of $\mathcal{TML}$ and $\mathcal{SCL}$ languages in [5]. This theoretical framework allows us to validate the correctness of the translation function. The extension of the $\mathcal{SCL}$ language to other structures, especially functions and escape operators, is a current research direction.

An implementation of the $\mathcal{TML}$ language is under progress. This is a meaning step to tackle the problem of comparison with other approaches of compilation and optimization of data-parallel programs [4, 8].

References

1. L. Bougé and J.-L. Levaire. Control Structures for Data–Parallel SIMD Languages: Semantics and Implementation. In *Future Generation Computer Systems*, pages 363–378. North Holland, 1992.
2. C.H.Koelbel, D.B.Loveman, R.S.Schreiber, G. Jr., and M.E.Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.
3. Digital Equipment Corporation. *DECmpp Programming Langage, Reference Manual*, 1992.
4. F.Coelho, C.Germain, and J. Pazat. State of the Art in Compiling HPF. In *Spring School on Data Parallelism*. Springer-Verlag, 1996. *(To appear)*.
5. Y. L. Guyadec, E. Melin, B. Raffin, X. Rebeuf, and B. Virot. A Loosely Synchronized Execution Model For a Simple Data-Parallel Language. Technical Report RR96-5, LIFO, Orléans, France, February 1996.
6. Y. L. Guyadec, E. Melin, B. Raffin, X. Rebeuf, and B. Virot. Horloges structurelles pour la désynchronisation de programmes data-parallèles. In *Actes de RenPar'8*, pages 77–80. Université de Bordeaux, France, may 1996.
7. Y. L. Guyadec, E. Melin, B. Raffin, X. Rebeuf, and B. Virot. Structural Clocks for a Loosely Synchronized Data-Parallel Language. In *MPCS'96* . IEEE, may 1996. *(To appear)*.
8. P. J. Hatcher and M. J. Quinn. *Data–Parallel Programming on MIMD Computers*. The MIT Press, 1991.
9. Hyperparallel Technologies. *HyperC Documentation*, 1993.
10. Y. Le Guyadec. Désynchronisation des programmes data-parallèles: une approche sémantique. *TSI*, 14(5):619–638, 1995.
11. N. Paris. Compilation du flot de contrôle pour le parallélisme de données. *TSI*, 12(6):745–773, 1993.
12. Thinking Machines Corporation, Cambridge MA. *C* Programming Guide*, 1990.

This article was processed using the LaTeX macro package with LLNCS style