A General but Simple Technique to Handle Asynchronous Data-Parallel Control Structures

Emmanuel Melin, Bruno Raffin, Xavier Rebeuf and Bernard Virot
LIFO - IIIA Université d'Orléans
4, rue Léonard De Vinci - BP 6759 F-45067 Orléans Cedex 02 - FRANCE
Bernard.Virot@lifo.univ-orleans.fr

Abstract

Nowadays, most of distributed architectures are easily scalable MIMD (Multiple Instruction streams, Multiple Data streams) parallel computers or networks of workstations. The challenge consists in taking advantage of the power of these architectures. It has been shown that data-parallel languages offer both a programming model easy to understand and several execution models which are able to exploit these distributed architectures. The compilation process has nevertheless to fill the gap between the synchronous and centralized programmer's point of view and an asynchronous and distributed execution model. An important problem is to reduce the number and the complexity of synchronizations between processors. This leads to complex program transformations, such as absorption techniques, in conditioning control structures, loops or routines. We propose a simple general technique based on structural clocks that permits the use of one-sided wait instructions instead of global synchronizations. Hence, no program transformation is needed and synchronization requirements are reduced. In particular, management of recursive routines is greatly facilitated. To illustrate these techniques, we propose an application to Strassen's algorithm for matrix product.

Keywords: *Design of Parallel Programming Languages; Data-Parallel Programming Model; Desynchronization; Asynchronous Execution Model; Partial Synchronization; Structural Clock; Data-Parallel Routine; Strassen's Algorithm.*

1. Introduction

The data-parallel programming model allows to take advantage of the scalability of recent distributed architectures, such as MIMD computers or networks of workstations, by expressing the parallelism through a sequential composition of actions applied to parallel data. Indices (virtual processors) to which actions are applied are active while the others are idle. From the programmer's point of view data-parallelism involves a single program, the control is centralized and very close to a sequential one. This model is structured and scalable. It inherits its qualities from classical sequential programming. The difficulty is then transferred to the compiler which has to efficiently run a data-parallel program on loosely synchronized architectures, while preserving the initial semantics of the synchronous and centralized programming model. Therefore data-parallel compilers have to realize complex optimizations up to manage data distribution and time desynchronizations between indices, especially regarding loops or routines.

In [6], Hatcher and Quinn propose a systematic approach to desynchronize data-parallel programs. Starting from the initial synchronous interpretation, considering every sequential composition as a global synchronization, the problem consists in removing unnecessary synchronizations. They keep synchronizations surrounding every communication phase to ensure that *potentially* exchanged data are the same as data that would be exchanged in the synchronous interpretation. With this approach, synchronizations are global to avoid handling dynamical sets of indices. Global synchronizations nested within control structures require a program transformation involving a mechanism of synchronization absorption [3, 6]. Nested synchronizations, which could only be accessed by active indices, are extracted from control structures in such a way that inactive indices are also engaged. The management of loops needs auxiliary variables to force indices which have terminated a loop to be engaged into the synchronizations for all the iterations. A similar program transformation is applied to routine calls to ensure that all indices will perform it.

In previous papers [4, 5], we have introduced a small asynchronous data-parallel language called $\mathcal{SCL}$ (which stands for Structural Clock Language). This language is designed to manage asynchronous data-parallel control structures (conditioning and loop). We use the program structure, a shared static information, to generate *structural clocks*. Each index owns a structural clock encoding its position in the program structure. Therefore, by the means of a partial order on structural clocks, each index knows its relative position towards the positions of other indices. This knowledge allows to locally decide if an index has to wait for another one. In this way we reduce the coupling between indices. This framework allows us to introduce partial synchronizations, thereby avoiding absorption machineries for conditioning control structures or loops.

In this paper we show that it is possible to reuse our approach to handle asynchronous, and possibly recursive, data-parallel routines. We propose a semantics of parameter-passing. We use structural clocks to obtain a simple management of recursion. In contrast with the approach of Hatcher and Quinn [6], there is no need for program transformation linked to routines.

We first provide an informal description of the $\mathcal{SCL}$ language and we present the structural clock construction. Next, we introduce asynchronous and recursive routine calls. Finally, we present an application of asynchronous routines through a matrix product algorithm using the Strassen's method inspired by McColl in the BSP framework [9].

2. The $\mathcal{SCL}$ language: syntax and informal description

In this section, we first briefly present the data-parallel programming model. Next, we define a suitable execution model and we give an informal description of the $\mathcal{SCL}$ language.

2.1. 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. Two kinds of actions can be applied to these objects: *componentwise* operations or *communications*. A program is a sequential composition of such actions. Each action is associated with the set of array indices at which it is applied. An index at which an action is applied is said to be *active* for this action. Legal expressions are usual *pure* expressions, i.e. without side effects, like the definition of *pure* functions in HPF [1]. The value of a pure expression at each index u only depends of 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: $This|_u = u$.

2.2. An execution model based on a twin memory management

We briefly recall the execution model previously defined for the $\mathcal{D}$ language [8]. It allows to reduce data-dependencies and is fully reusable in our framework. It is based on a simple array of asynchronous virtual processors. Each index u owns a *private memory* containing the components of all vector variables at u, and a *public memory* containing a copy of the private memory of the same index. Updating of public memories are explicitly executed through a specific instruction (dump). Private memories are dedicated to local computations. Public memories are reserved for communications. This mechanism allows to express more asynchronism, as local transitions between two dump instructions are temporarily masked for other indices.

2.3. Basic statements

We give an informal description of the $\mathcal{SCL}$ language.

- skip **statement.** This instruction does nothing and just terminates.
- dump. Each index executing this instruction locally copies its private memory in its public one.
- **Assignment:** $X := E$**.** Each index u executing this instruction updates its component $X|_u$ of its *private* memory 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**.** Each index u executing this instruction, evaluates the address $A|_u$ using its own private memory. Next, the component $Y|_u$ is updated, in the private memory of u, with the remote value $X|_{A|_u}$. The expression $X|_{A|_u}$ is evaluated using the public memory of the index $A|_u$.

2.4. Control structures

We turn now to the control structures available in $\mathcal{SCL}$.

- **Sequencing:** $S; T$**.** As soon as an index u ends the execution of S, it starts the execution of T. This statement does not involve any implicit synchronization.
- **Concurrent conditioning:** where B do S elsewhere T end**.** This statement is an asynchronous double-branched conditioning construct. All active indices are divided into two disjoint sets

depending on the current value of the pure expression B. An active index u which evaluates the expression $B|_u$ to true (respectively false) executes the program S (respectively T). This statement does not involve any implicit synchronization.

Remarks:

- The where/elsewhere construct is a concurrent version of the if/else of MPL [2] or where/elsewhere of HYPERC [7].
- It makes explicit in the syntax the absence of dependence between two sets of indices. The programmer has to guarantee this independence. In turn, a synchronization mechanism may take advantage of this information.

- **Iteration:** loopwhere B do S end**.** Each currently active index u repeats the computation of the block S while the pure expression $B|_u$ evaluates to $true$. When an index terminates the loop, it goes on executing the instructions that directly follow the loop without any implicit synchronization.

  Remark: This is an asynchronous version of the while of MPL [2] or whilesomewhere of HYPERC [7].

2.5. Routine calls

Now, we extend our language with routine calls. A routine call has the following syntax: call $Routine\_name(variable\_list)$. An index active for this instruction executes the program associated with the routine $Routine\_name$, with its arguments instantiated by the variable list $variable\_list$. A routine declaration consists in a routine name $Routine\_name$, several arguments typed by attributes, a list of local variables $variable\_list$, a $\mathcal{SCL}$ program S and a routine end:

$Routine\_name($ $attribute : argument\_list;$
$\qquad\qquad attribute : argument\_list)$
$local\_variable\_list;$
$S;$
end$\{Routine_name\}$;

Each argument is preceded by an attribute that specifies the way data exchanges are authorized. The attribute *in* corresponds to a call by value. If a parameter holds this attribute, then its final values (in public and private memories), at the end of the execution of the routine call, are **not** propagated outside this routine. The attribute *in-out* corresponds to a call by address. If a parameter holds this attribute, then its final values (in public and private memories), at the end of the execution of the routine call, are propagated to the calling program.

Such a specification of routines involves, for each index, a stack of public memories and a stack of private memories. An index beginning a routine call pushes a new instance of its private memory on the stack. It pops this instance at the end of the routine.

In a similar way, an index pushes a corresponding instance of its public memory on the stack at the beginning of the execution of a routine call. However, before popping such a stack level, an index must ensures that no other index may later reference data stored there. Hence, an index u pops the public memory when all indices have step over the routine call. For sake of simplicity, in the following, we only update the public memory in the main program. This restriction avoids to handle stacks of public memories.

Example: The $\mathcal{SCL}$ language allows public memory clashes between communications and updatings, thereby yielding non-determinism. Let us give a simple example. In the left hand side program, the first branch of the where/elsewhere construct only modifies A in private memory and the call to $Assign$ in the second branch references the value of A in public memory. Therefore, the two branches are *independent*. In the right hand side program, the first branch updates the public copy of A while the second branch references this value. Hence, the two branches are not *independent*.

$Assign($in X; in-out: $Y)$ get X from 1 into Y; end; $\{Assign\}$ $\{Mainprogram\}$ where $(This < 2)$ do $A := 1;$ elsewhere $Assign(A, B);$ end;	$Assign($in X; in-out: $Y)$ get X from 1 into Y; end; $\{Assign\}$ $\{Mainprogram\}$ where $(This < 2)$ do $A := 1;$ dump; elsewhere $Assign(A, B);$ end;

2.6. Waiting mechanism

In order to ensure determinism, the programmer has to manage the consistency of public memories references by serializing instructions which interfere with these memories. With this aim in view, classical approaches [6] use a global synchronization mechanism. We suggest to substitute global synchronizations for a wait notion. We introduce the waiting statement wait A, where A is a pure expression. The intuitive idea is the following. This instruction delays the index u which is executing it, until the target index $A|_u$ has stepped over all the instructions referencing public memories (dump,get) and preceding the wait instruction. Note that $A|_u$ might never reach some preceding instruction referencing the public memory due to loops and conditioning structures. So, an index u can step over the wait point as soon as $A|_u$ executes a different branch from u in a where/elsewhere structure, or as soon as $A|_u$ has terminated the current loopwhere executed by u. Therefore, we see that only dump, get, where/elsewhere and loopwhere instructions play a rôle in the wait management. We qualify them as *meaningful* instructions. Note that routine calls do not interfere with the public memories and do not modify index activity. Therefore, they are not meaningful statements. 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 stepped over all *meaningful* statements coming before the wait instruction.

We can now formalize the semantics of the $\mathcal{SCL}$ waiting instruction:

- **Waiting point:** wait A**.** An index terminates this instruction if the conditions **C1** and **C2** are satisfied.

Partial synchronization. The wait_all instruction is a generalization of the wait statement. An index terminates this instruction when it has performed a wait for all other indices.

Remarks:

- An index u may step over a wait A instruction even if the index $A|_u$ needs to execute non-meaningful statements before to reach the waiting point.
- A wait instruction can be seen as a one-sided point to point synchronization.
- A wait_all can be viewed as a partial synchronization as some indices may not execute it.
- Note that with our approach, routine calls are non-meaningful statements. Therefore they do not bound asynchronism.

Determinism: Two simple programming rules are sufficient to avoid non determinism.

- It is impossible to impose an ordering on the execution of the instructions of the two branches of a where/elsewhere construct. To avoid this source of non-determinism, it is sufficient to forbid the presence of get X from A into Y instructions in one branch, if the variable X is modified and copied (dump) in public memory in the other branch.
- Moreover, to avoid clashes between dump and get instructions, it is sufficient to introduce a wait_all before each dump and a wait A before each get X from A into Y. The wait_all instruction ensures that other indices have passed all previous get instructions before to execute the dump. In the same way, the wait A instruction ensures that the target index at the address $A|_u$ has executed all previous dump instructions.

Remark: Although the preceding rules are sufficient to ensure determinism, it is often possible to reduce synchronization requirements by a more careful analysis of data dependences (Cf. 3)

2.7. Structural clock presentation

For the sake of completeness, we recall now the construction of structural clocks . Each index owns a structural 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. To record an index position we just need to know the number of meaningful statements already executed in each instruction block of each nesting level. Therefore, a structural clock encodes index positions by a list. Each term of the list corresponds to a nesting level. The *Structural Clock* t_u of an index u is so expressed by a list of pairs. 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 level. The label l is used to distinguish which branch of a where/elsewhere statement an index is inside. Lists are built by popping or pushing pairs on their right hand side.

- **The** get **and** dump **instructions**

  When an index u executes a get or a dump instruction its clock $t_u = t(l, c)$ becomes $t_u = t(l, c + 1)$.

- **The** where **structure**

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

- **The** loopwhere **structure**

  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 in 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. We define a partial order on structural clocks. 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:

A 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* $t^1,\ t^2,\ t^3,\ c_u,\ c_v$ *and l such that* $t_u = t^1(l,c_u)t^2,\ t_v = t^1(l,c_v)t^3$ *and* $c_u < c_v$.

The former condition above formalizes the intuitive idea that the index v is inside the first meaningful control structure that follows the instruction currently executed by u. Note that there is no meaningful instruction between this instruction and the control structure. The latter one expresses that, with regard to the block encoded by $t^1(l,...)$, the index u is later than v.

2.8. Handling wait statements using structural clocks

Structural clocks yield a general mechanism to handle wait statements. The conditions **C1** and **C2** above are captured by the following rule:

Consider an index u with structural clock t_u that asks for a wait A*. This index can terminate the wait instruction if the index $v = A|_u$ with a structural clock t_v satisfies $t_v \prec t_u$.*

Remark: We can easily prove that any $\mathcal{SCL}$ program is deadlock free using the following property: an index owning a minimal clock is never blocked (if it has not finished the program execution, otherwise all indices have finished).

3. Example of recursive program: the Strassen's algorithm

We turn now to an example (Fig. 1). This $\mathcal{SCL}$ program illustrates the use of recursion in our asynchronous model. It can be viewed as an improvement of a BSP program initially proposed by Mc Coll for Strassen's Algorithm [9].

Let A and B be two $n \times n$ matrices and consider the problem of computing the product $C = A.B$. A classical recursive algorithm consists in regarding each matrix as four $n/2 \times n/2$ sub-matrices. The computation of C is then given by the computation of the four sub-matrices C_{ij} $(i,j \in \{1,2\})$ where $Cij = A_{i1}.B_{1j} + A_{i2}.B_{2j}$. This yields eight sub-matrix products that are treated recursively by the same method. The Strassen's algorithm consists in reducing these eight sub-matrix products to seven, by using shrewd matrix additions and subtractions.

We consider matrices A, B and C whose size are $n = 4^k.7^k$. These matrices can be seen as 4^{2k} sub-matrices of 7^{2k} elements. The underlying geometry of indices corresponds to a two dimensional array $[1..7^k, 1..7^k]$. Each index is identified with one of the 7^{2k} element position within each sub-matrix. An index initially holds all the 4^{2k} elements which are in that position within their sub-matrix.

We now describe the algorithm starting from this configuration (Fig. 1). The Strassen's algorithm is programmed in $\mathcal{SCL}$ by using four routines $Stras\_par1$, $Stras\_par2$, $Communication$ and $Strassen\_local$. The parallel routine $Stras\_par1$ initiates the algorithm, halting each chain of recursive calls when the level of recursion reaches $2k$. Each step of the recursion consists in computing seven matrices $L[K]$ and $R[K]$ $(1 \leq K \leq 7)$ by additions and subtractions of the four sub-matrices of A and B. Then, we proceed to the next recursion level by computing the seven matrix products $L[K].R[K]$. All the indices are engaged during these steps. We obtain 7^{2k} recursion leaves that are numbered from 1 to 7^{2k}. When a leaf number Q is reached, the two matrices to be multiplied are of size 7^k. Each index owns exactly one element of each matrix. To record locally these values we use two arrays $Tmp1[Q]$ and $Tmp2[Q]$.

When the execution of $Stras\_par1$ terminates, the parallel routine $Communication$ starts. Each index gathers the data stored in the memories of other indices, in the components of $Tmp1$ and $Tmp2$ that corresponds to its index number. These data are transferred in the matrices $A1$ and $B1$. So, the matrix product linked to each leaf can be computed locally, in parallel on all indices, using $Strassen\_local$, a sequential version of the Strassen's algorithm. This yields a matrix $C1$ for each index.

```
Stras_par1( in-out:Tmp1, Tmp2;in:A,B;in-out:Q; in:N)
L,R:array[1..7] of array[1..4^k, 1..4^k] of real;
I,J,K:integer;
where N ≥ 2 do
  forwhere I := 1 to N/2 do
    forwhere J := 1 to N/2 do
      L[1][I, J] := A[I, J];
      L[2][I, J] := A[I, J + N/2];
      L[3][I, J] := A[I + N/2, J]
                   +A[I + N/2, J + N/2];
      L[4][I, J] := A[I, J] − A[I + N/2, J];
      L[5][I, J] := L[3][I, J] − A[I, J];
      L[6][I, J] := A[I, J + N/2] − L[5][I, J];
      L[7][I, J] := A[I + N/2; J + N/2];
      R[1][I, J] := B[I, J];
      R[2][I, J] := B[I + N/2, J];
      R[3][I, J] := B[I, J + N/2] − B[I, J];
      R[4][I, J] := B[I + N/2, J + N/2]
                   −B[I, J + N/2];
      R[5][I, J] := R[4][I, J] + B[I, J];
      R[6][I, J] := B[I + N/2, J + N/2];
      R[7][I, J] := B[I + N/2, J] − R[5][I, J];
    end;
  end;
  forwhere K := 1 to 7 do
    call Stras_par1(Tmp1, Tmp2, L[K], R[K],
                    Q, N/2);
  end;
elsewhere (N=1)
  Tmp1[Q] := A[1, 1];
  Tmp2[Q] := B[1, 1];
  Q := Q + 1;
end;
end{Stras_par1};
```

```
Communication( in: Tmp1,Tmp2;
               in-out: A1, B1 )
I,J:integer;
forwhere I := 1 to 7^k do
  forwhere J := 1 to 7^k do
    wait (I,J);
    get Tmp1[This] from (I, J) into A1[I, J];
    get Tmp2[This] from (I, J) into B1[I, J];
  end;
end;
end{Communication};
```

```
Stras_par2( in-out:C; in: C1;in-out:Q;in:N)
M:array[1..7] of array[1..4^k, 1..4^k] of real;
T:array[1..2] of array[1..4^k, 1..4^k] of real;
I,J,K:integer;
where N ≥ 2 do
  forwhere K := 1 to 7 do
    call Stras_par2(M[K], C1, Q, N/2);
  end;
  forwhere I := 1 to N/2 do
    forwhere J := 1 to N/2 do
      T[1][I, J] := M[1][I, J] + M[5][I, J];
      T[2][I, J] := T[1][I, J] + M[4][I, J];
      C[I, J] := M[1][I, J] + M[2][I, J];
      C[I, J + N/2] := T[1][I, J]
                      +M[3][I, J] + M[6][I, J];
      C[I + N/2, J] := T[2][I, J] + M[7][I, J];
      C[I + N/2, J + N/2] := T[2][I, J]
                            +M[3][I, J];
    end;
  end;
elsewhere (N=1)
  wait Q;
  get C1[This_x, This_y] from Q into C[1, 1];
  Q := Q + 1;
end;
end{Stras_par2};
```

```
{Main program}
A, B, C:array[1..4^k, 1..4^k] of real;
A1,B1,C1:array[1..7^k, 1..7^k] of real;
Tmp1,Tmp2:array[1..7^k] of real;
Q:real;

Q := 1;
call Stras_par1(Tmp1, Tmp2, A, B, Q, 4^k);
dump;
call Communication(Tmp1, Tmp2, A1, B1);
call Strassen_local(A1, B1, C1);
dump;
Q := 1;
call Stras_par2(C, C1, Q, 4^k);
```

Figure 1. This $\mathcal{SCL}$ program illustrates the use of recursion in our asynchronous model.

In a last part a new recursion is initiated with $Stras\_par2$. Computations are performed during the backtrack on the recursion tree. First of all, the results of the local matrix products are distributed among the indices. For a leaf number Q, each index (i, j) gets $C1[i, j]$ from the index Q. Each time a recursion level is finished, we obtain a matrix product $M[K] = L[K].R[K]$. From the seven matrices $M[K]$, the matrices $T[1], T[2]$ and then the matrix C are locally computed by each index.

Two addressing systems are used. The parallel constants $1 \leq This\_x \leq 7^k$ and $1 \leq This\_y \leq 7^k$ appoint respectively the line number and the column number of each index. The parallel constant $1 \leq This \leq 7^{2k}$ provides a linear coordinate. We also use an instruction forwhere that merely corresponds to the loopwhere instruction with an iteration counter.

```
forwhere J := 1 to N do              J := 1;
  S                          ≡       loopwhere (J <= N) do
end                                    S;
                                       J := J + 1
                                     end
```

Communications are performed through global variables by using public memories. The program uses two dump instructions executed by all indices. They are necessary to copy the variables $Tmp1$, $Tmp2$, and then $C1$ in public memories.

Remarks:

- In section 2.6 we proposed an approach to systematic $\mathcal{SCL}$ programming, by putting a wait_all before each dump and a wait A before each get X from A into Y. Nevertheless, careful analysis of data dependences can yield interesting optimizations. We use this technique for our program.
  - A wait_all instruction is useless before the first dump since no get precedes it.
  - Only one wait is required before the get instructions of the routine $Communication$ because the two get instructions are contiguous and access the same target index.
  - The get instructions of the routine $Communication$ and the second dump are not separated by a wait_all. Therefore, such a get can be executed after the dump. The acceded data in $Tmp1$ and $Tmp2$ are nevertheless not erroneous since no modification of these data occurs after the first dump.
- This program may be written differently removing all wait instructions and putting a wait_all after each dump instruction. Such a program would have been close to the original BSP one [9]. The interest of our approach is to allow more asynchronism by interleaving point to point synchronizations and computations.

4. Conclusion

In this paper, we have proposed a structured approach mixing data-parallelism and task parallelism. In contrast to classical synchronous data parallelism, asynchronism allows us to introduce task parallelism through the where/elsewhere construct. Our approach relies on the structural clock mechanism to manage synchronizations. The structural clocks record index positions with regard to the program structure. The waiting algorithm then uses an ordering on structural clocks to compare index positions. Structural clocks yield safe synchronizations as they manage waiting instructions without deadlock.

The structural clock mechanism requires additional computations and data exchanges. The size of structural clocks, and so the complexity of the comparison algorithm, depends on the program. It is equal to the deepest nesting of control structures and routine calls. The amount of additional exchanged data evolves at most linearly with the number of indices, as each index owns exactly one structural clock.

Loosely synchronized programs are obtained without the machinery needed by other desynchronization methods [6, 3]. Moreover, these methods generally use global synchronization barriers whereas the wait instruction authorizes one-sided and partial synchronizations.

We have shown that it is possible to extend the $\mathcal{SCL}$ language with recursive routines. In a restricted framework where communications are authorized only on global variables, we had shown that neither routines nor recursion reduce asynchronism.

The potentialities of our language and of structural clocks are exemplified through Strassen's Algorithm. Using $\mathcal{SCL}$ programming, the involved harness of synchronizations and communications is expressed in a very simple and intuitive way.

Note that communications on local variables and *in* arguments would require stacks of public memories. If routine calls are not surrounded by wait instructions, informations on index positions are then necessary to ensure consistency of public memories. Structural clocks can be used to provide these informations and so to handle such routines without additional synchronizations.

Our approach is general enough to be extended easily to more complex languages. It relies on a clear theoretical model which provides well-founded program transformations [10].

References

[1] C.H.Koelbel, D.B.Loveman, R.S.Schreiber, G. Jr., and M.E.Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[2] Digital Equipment Corporation. *DECmpp Programming Langage, Reference Manual*, 1992.

[3] G.Utard and G.Hains. Deadlock-free absorption of barrier synchronisations. *Information Processing Letters*, 56:221–227, 1995.

[4] Y. L. Guyadec, E. Melin, B. Raffin, X. Rebeuf, and B. Virot. A Loosely Synchronized Execution Model for a Simple Data-Parallel Language. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *EuroPar'96 Parallel Processing*, number 1123 in LNCS. Springer-Verlag, 1996.

[5] Y. L. Guyadec, E. Melin, B. Raffin, X. Rebeuf, and B. Virot. Structural Clocks for a Loosely Synchronized Data-Parallel Language. In *MPCS'96* . EUROMICRO and Istituto di Ricerca su Sistemi Informatici Parallele, may 1996. *(Preliminary Proceedings)*.

[6] P. J. Hatcher and M. J. Quinn. *Data–Parallel Programming on MIMD Computers*. The MIT Press, 1991.

[7] Hyperparallel Technologies. *HyperC Documentation*, 1993.

[8] Y. Le Guyadec. Désynchronisation des programmes data-parallèles: une approche sémantique. *TSI*, 14(5):619–638, 1995.

[9] W. F. McColl. A BSP Realisation of Strassen's Algorithm. Oxford University Computing Laboratory, May 1995.

[10] E. Melin, B. Raffin, X. Rebeuf, and B. Virot. A Loosely Synchronized Execution Model for a Simple Data-Parallel Language.
Technical Report RR96-11, LIFO, Orléans, France, October 1996. `http://web.univ-orleans.fr/~virot`.