A Structured Synchronization and Communication Model Fitting Irregular Data Accesses *

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

January 17, 1998

*This work has been partly supported by the "Institut du Développement et des Ressources en Informatique Scientifique" (IDRIS), Orsay, France. Project number 970829.

Abstract

In this paper, we present a parallel programming and execution model based on a *logical* ordering of control flows. We show that it is possible to provide a unifying framework consisting of a synchronous programming model, thereby facilitating the mastery of programs, and an asynchronous execution model yielding efficient executions. Our approach is based on a SPMD and task parallel programming language, called $\mathcal{SCL}-\mathcal{C}han$. Communications take place through channels and rely on explicit send/receive instructions. In opposite to classical message passing models, synchronizations and communications are dissociated. We show that it is possible to perform a data-driven automatic translation of sequential and arbitrary DOACROSS loops into $\mathcal{SCL}-\mathcal{C}han$, by using non-matching send/receive instructions. Our parallelization technique allows to handle irregular control, and leads to optimizations of communications in irregular computations.

Keywords: Design of Parallel Programming Languages; Task Parallel Language; Data Parallel Programming Model; Structural Clock; Automatic Parallelization; Irregular Algorithm; Implementation Results on Cray T3D.

- **Running title** : A Structured Synchronization and Communication Model

- **Contact:** B. Virot

  LIFO - IIIA Université d'Orléans

  4, rue Léonard De Vinci - BP 6759 F-45067 Orléans Cedex 02 - FRANCE

  Tel: (+33) (0) 2 38 41 73 18 - Email: Bernard.Virot@lifo.univ-orleans.fr

1 Introduction

Nowadays, parallelism appears as an inescapable technique to reach large computation abilities. Nevertheless, in spite of recent efforts of parallel language designers, only a reduced community of specialists is able to program efficiently in a straight parallel model. Automatic parallelization attempts to automate translation of sequential programs into efficient parallel ones. The aim is to offer the programmer the possibility to work in a classical sequential framework and not to pay attention to parallelism. But at present time, only simple regular programs succeed to be efficiently parallelized. Moreover, an additional difficulty level appears with irregular applications such as computations on sparse matrices. Both manual and automatic parallelization have to face irregular data distribution, dynamical load balancing and irregular control, leading to irregular remote data accesses. Then, an efficient program execution needs to manage complex point to point synchronizations between processes. In this framework, a crucial problem is to have a clear programming model enabling to express such fine synchronization schemes, while ensuring program correctness and execution efficiency.

Synchronizations define an ordering on instances of program instructions of which depends the available parallelism. The most expressive programming model corresponds to a causality ordering [17]. It is called task parallelism and relies on an explicit distribution of data and computation tasks on asynchronous processors with physically distributed memories. Data communications take place through explicit send/receive instructions [10, 16, 23]. This programming model fits the execution model provided by present parallel computers, thereby enabling efficient executions. As a receiving processor is blocked as long as no data is received, communications and synchronizations are mixed. Send/receive instructions have to match to avoid deadlocks. The multiplicity of control flows makes programs mastery a difficult task. Executions do not follow the syntactic structure of the code. As a consequence, sequential and syntax-driven reading of programs becomes impracticable for the programmer.

On the opposite side from task parallelism, the data parallel approach restricts the expression of parallelism to provide a structured programming model. Parallelism is expressed through a sequential composition of synchronous actions on parallel objects. Control flow ordering is sequential. Programs are deterministic and deadlock free. Synchronism makes easier program design and verification [3, 4, 14] by enabling syntax-driven program reading. Nevertheless, the model is too restrictive to handle irregular data accesses. The difficulty is then transferred to the compiler. It has to efficiently translate data parallel programs into task parallel ones, in order to yield efficient executions, while preserving the semantics of the synchronous and centralized programming model. So, data parallel compilers have to realize complex optimizations up to manage time desynchronizations between processors [15, 22].

Other approaches propose bridging models. Fortran M [10] relies on a task parallel model. But its essential communication mechanisms, like single reader/single writer channels or single-assignment variables enforce program determinism. Moreover, a parallel task can call a data parallel program, such as an HPF one. The programmer can therefore express task parallelism that the data parallel compiler does not need to extract, thereby helping it to reduce synchronizations without endanger program correctness. Nevertheless, the programmer still has to face deadlock problems and complex management of multiple control flows, ordered only by data communications. Syntax-driven program reading is generally forbidden, thereby making programming harder.

The BSP model [18, 24] proposes a coarse grain sequential ordering of control flows. A program is divided in a sequence of supersteps, where each superstep is followed by a global synchronization. A superstep consists in parallel computations with remote memory accesses taking effect at global synchronization points. BSP is deadlock free and determinism is easily guaranteed by enforcing simple rules at the superstep level, such as to prevent processors from writing in the same distant variable. Note that this type of control flow ordering has also been considered in an irregular framework by Diderich and Gengler [7]. Nevertheless, expressiveness is limited, since the model provides a unique instruction ordering, due to the mandatory use of global synchronizations.

In this paper, we present a different approach based on a *logical* ordering of control flows. We show that it is possible to provide a unifying framework consisting of a synchronous programming model, thereby enabling syntax-driven program reading, and an asynchronous execution model yielding efficient executions. Our approach is based on a SPMD and task parallel programming language, called $\mathcal{SCL}-\mathcal{C}han$. Communications take place through channels and rely on explicit send/receive instructions. In opposite to classical message passing models, synchronizations and communications are dissociated. A receive instruction does not block on an empty channel, but it waits for the sending processor to perform all instructions *logically* preceding the receive statement. As send/receive matching is not mandatory, it becomes possible to easily express irregular communication schemes.

To highlight $\mathcal{SCL}-\mathcal{C}han$ capabilities to manage irregular remote data accesses, we show that it is possible to use $\mathcal{SCL}-\mathcal{C}han$ as a target language for an efficient data-drivens automatic translation of sequential arbitrary DOACROSS loops. Our parallelization technique can handle irregular control and irregular data accesses, thereby enabling to optimize communications in sparse computations for example. The approach proposed in [2] also relies on a SPMD data-driven parallelization. However, mandatory matching of send/receive instructions makes difficult to avoid an exhaustive evaluation of numerous guards. Other classical techniques, such as the polytope model [9], are not well-adapted to manage the irregular control required by sparse computations. A static approach [8] consists in integrating a data analysis that takes null values into account during the compilation process. Although this approach is interesting, it requires a new compilation each time data are modified. $\mathcal{SCL}-\mathcal{C}han$ allows to manage such data analysis at run-time by introducing irregular control in the code.

We first present the syntax and informal asynchronous semantics of $\mathcal{SCL}-\mathcal{C}han$. Next, we show that $\mathcal{SCL}-\mathcal{C}han$ yields deterministic and deadlock free programs. We show that it is possible to define a synchronous programming model. We prove that synchronous executions are equivalent to any other possible asynchronous execution with respect to the returned result. Next, we present a translation function from a sequential code to a loosely synchronized $\mathcal{SCL}-\mathcal{C}han$ program. The parallel code is independent from mapping, thereby facilitating dynamic load balancing. Moreover, we show that by choosing "good" data distributions, it becomes possible to consider additional optimizations. We give two examples, with various data dependences and different communication requirements, which highlight the capabilities of our approach: a sparse Gauss-Seidel like algorithm and a sparse Cholesky factorization. We propose optimizations taking into account fine semantic properties of the algorithm in order to reduce communications and synchronizations. Last, we present performance results on a Cray T3D. It appears that $\mathcal{SCL} - \mathcal{C}han$ performances are comparable to classical MPI programming and that exploiting sparsity by using send/receive unmatching significantly improves performances.

2 The language: an asynchronous execution model

2.1 The $\mathcal{SCL} - \mathcal{C}han$ abstract machine

The abstract machine is an asynchronous parallel one with Pe_nb processors. Processors are called *indices* and denoted by u, v,... Each index u owns a *(private) memory* where it stores its own data. Memory allocation is uniform, i.e. each variable X is systematically allocated in each private memory. The value of a variable X stored in the private memory of an index u is denoted by $X|_u$. It cannot be directly referenced by another index.

Data exchanges between indices take place through explicit send/receive communications. Therefore, a datum of the private memory of an index u becomes available for a distant index v only if it is explicitly *sent* to v. The index v can record such a value in its private memory only if it explicitly *receives* it. The data in transit, i.e. the data sent and not yet received are stored in *channels*. There exists exactly one channel by variable and by pair of indices. Each channel corresponds to a LIFO (Last-In First-Out) list managed with respect to a specific ordering we shall present in the next section. We denote by (u, v, X) the channel associated to the sender u, the receiver v and the variable X. At the beginning of a $\mathcal{SCL} - \mathcal{C}han$ program all channels are empty.

2.2 Language instructions

The asynchronous execution model of $\mathcal{SCL} - \mathcal{C}han$ relies on the SPMD paradigm. All the indices execute the same program. An index that executes an instance of an instruction is called *active* for this instance. Indices execute instructions asynchronously. Only receive instructions can block.

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

Assignment: $X := E$. An active index u updates its value $X|_u$ of X in its private memory with the value $E|_u$. The expression E must be *pure*, i.e. evaluable in the private memory of the index u.

Synchronous sequence: $S; T$. When an active index finishes the execution of S, it starts the execution of T.

Concurrent conditioning: where B do S elsewhere T end. If an active index u evaluates the pure condition $B|_u$ to $true$, then it executes the instruction block S, otherwise it executes T.

Remark that a single branch conditioning construct where B do S end may simply be defined as where B do S elsewhere skip end.

Loop: loopwhere B do S end. Iteration is expressed by classical loop unfolding. An active index repeatedly executes S while it evaluates the pure expression $B|_u$ to $true$.

Send: send X to A. This instruction is non blocking. The expression A must be pure. An active index u sends the value $X|_u$ to the index $v = A|_u$, i.e. it adds this value to the channel (u, v, X).

Receive: receive X from A into Y. The expression A must be pure. The semantics of this instruction relies on a logical partial ordering on the instances of program instructions. Hence, we turn now to the description of this ordering. The aim is to capture the intuitive

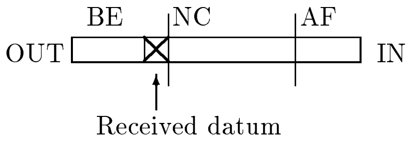

Figure 1: Representation of a channel.

notion of *possible* dependences between instructions, according to their relative positions in the syntactic structure of the program.

Definition 1 (Instruction logical ordering) *An instance of an instruction i_1 is* **before** *an instance of an instruction i_2 if there exists at least one active index for i_1 that may become later active for i_2. If there exists no index which can execute both i_1 and i_2, then these instruction instances are* **not comparable**.

Examples.

- Consider the program $X := 1$; loopwhere $X < 10$ do $X := X + 1$; end. The instruction $X := 1$ comes before the first instance of the instruction $X := X + 1$ (the instruction executed at the first loop iteration) which, in turn, comes before the second instance of $X := X + 1$.
- Two instructions, each one belonging to a different branch of the same where/elsewhere construct, are not comparable.

We turn now to the informal semantics of the receive X from A into Y instruction. The receive instruction is blocking. Its execution depends on a *waiting condition*: a receiving index u waits for the sending index $v = A|_u$ to execute all send instructions coming before the receive (according to the logical ordering previously defined). When this condition is satisfied, the channel (v, u, X) is considered. Data in the channel can be divided into three sets BE, NC and AF (see figure 1). The set BE is composed of the data sent by instances of send instructions coming before the receive. The set NC corresponds to the data sent by not comparable send instructions, and the set AF contains data sent by send instructions coming after the receive. If the set BE is empty, then index u assigns a special value denoted by `nil` to $Y|_u$. Otherwise, it first selects the most recent datum of BE, next it assigns it to $Y|_u$ and last it deletes all data of BE. To sum up, channels are managed as LIFO lists with respect to the instruction logical ordering.

Remarks:

- As two instructions, each one belonging to a different branch of the same where/elsewhere construct are not comparable, no communication can take place between these two branches: the where/elsewhere expresses disjoin parallelism [1]. A datum sent from one branch to the other can only be received by a receive instruction executed after exiting the where/elsewhere. It can be seen as if the send has been performed just at the end of the where/elsewhere construct.

- Communication semantics ensures that a receive instruction cannot refer to a send not comparable or coming after it with regard to the logical ordering on instructions.

- Exact matching between send and receive instructions is not mandatory. More receive than send instructions can be executed on a channel without risk of deadlock. This feature is important as it allows the programmer to limit communications when they are unpredictable [20]. Classical message passing languages and libraries [10, 11] do not offer such a possibility.

2.3 Structural Clocks

To be feasible, the execution model requires that each index should know its position in the program structure according to the instance of the instruction currently executed. An index that executes a send instruction can stamp the sent data by its current position. An index executing a receive instruction knows if it needs to wait for the sending index. Next, it can use the stamps of data in channels to complete the receive. In order to encode index positions, we introduce *Structural Clocks* [13, 12, 19, 20]. Structural clocks have to record index positions with regard to the instances of instructions that store data in channels, i.e. send instructions. Since the execution of these instructions can depend on enclosing control structures, the instructions taken into account, called the *meaningful* ones, are send, loopwhere and where/elsewhere. 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. Note that we do not have to consider each iteration of a loopwhere instruction as a new nesting level. We only push a new term at the first loop iteration and next, we count instructions already executed in the loopwhere body.

With the aim of ordering two structural clocks, we have to compare the counters corresponding to the common instruction block of the innermost nesting level. We so define this partial ordering as a lexicographical ordering based on partially ordered pairs.

Definition 2 *Let u and v be two indices with their structural clocks t_u and t_v. An index u is said to be* **before** *an other index v (denoted by $t_u \prec t_v$) if one of the following conditions holds:*

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

If we only consider meaningful instructions with regard to structural clock updates, this ordering is isomorphic to the logical ordering on instructions. An index u is *before* v ($t_u \prec t_v$) if and only if the instruction instance currently executed by u comes *before* the one executed by v.

2.4 Example

For sake of clarity, we introduce a forwhere $I := L$ to B do S end instruction, defined as $I := L$; loopwhere $I \leq B$ do S; $I := I + 1$; end.

The simple program displayed figure 2 encodes a pipe-line between two indices, numbered 1 and 2. It exemplifies the way structural clocks are handled.

The array Z contains the initial values. At the right hand side of each instruction, we display the structural clock an index owns when it has finished to execute it. The value i corresponds to the loop iteration number. By using where instructions, a specific task is assigned to each index. The index 1 computes S from local data and sends it to index 2. Index 2 receives it and adds it to $A[i]$. Each task is repeated 100 times.

Since index 1 executes only send instructions, it is not involved in waiting steps. It repeatedly adds its computation result into the channel $(1, 2, S)$. When it sends the datum $S|_1$ at iteration i, it writes $S|_1$ in the channel, stamped with its current structural clock $(0, 0)(0, 2i)(0, 0)$. Next, it increases its clock to $(0, 0)(0, 2i)(0, 1)$. When index 2 executes the receive instruction of iteration i, it owns the clock $(0, 0)(0, 2i + 1)(0, 0)$. At first, it waits for the sending index 1 as long as 1 has a clock strictly lower than its own, i.e. as long as 1 has not executed the second where of iteration i. Then, it assigns the last datum (with regard to the instruction ordering) stored in the channel $(1, 2, S)$ into R and removes it. This datum corresponds to the one stamped by the structural clock $(0, 0)(0, 2i)(0, 0)$. Note that there exists no older datum to remove from the channel since each sent datum is received.

Remark. Note that, as we only use single branch conditioning constructs, the first component of each pair of a clock is always 0.

2.5 Fundamental properties of the asynchronous model

In this section we sketch the proofs of two fundamental properties of the execution model of $\mathcal{SCL} - \mathcal{C}han$, i.e. deadlock freedom and computation determinism [21]. They ensure that either

forwhere $i := 0$ to 99 do	$(0,0)(0,2i)$
where $This = 1$ do	$(0,0)(0,2i)(0,0)$
$S := i * Z[i]$;	$(0,0)(0,2i)(0,0)$
send S to $This + 1$;	$(0,0)(0,2i)(0,1)$
end;	$(0,0)(0,2i+1)$
where $This = 2$ do	$(0,0)(0,2i+1)(0,0)$
receive S from $This - 1$ into R;	$(0,0)(0,2i+1)(0,0)$
$A[i] := A[i] + R$;	$(0,0)(0,2i+1)(0,0)$
end;	$(0,0)(0,2i+2)$
end;	$(0,1)$

Figure 2: Program encoding a pipe-line between two indices.

all possible asynchronous computations finish returning the same result, or they all loop endless. They are fundamental on two points. At first, detections of deadlock and non determinism, generally complex and time consuming tasks, are avoided. Moreover, these properties are necessary conditions to ensure the correctness of the synchronous programming model we shall see in the next section.

2.5.1 Deadlock freedom

At first, we prove that the execution model is deadlock free, i.e. that all possible executions of a program either all terminate or all loop endless.

Theorem 1 *The $\mathcal{SCL} - \mathcal{C}han$ language is deadlock free.*

The intuitive idea of the proof is that a deadlock configuration is impossible since indices with minimal clocks cannot be blocked: if such an index was blocked, then the waited index would own a strictly lower clock. Note that there always exists minimal structural clocks since only a finite number of indices is considered.

2.5.2 Determinism of computations

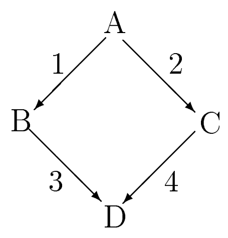

Figure 3: Illustration of the diamond property.

$\mathcal{SCL}-\mathcal{C}han$ is deterministic, i.e. all possible executions of a program always yield the same final result (they all terminate or all loop endless). This property is obtained from the *Diamond Property* [1]. The intuitive idea of the diamond property is that if two instructions 1 and 2 (see figure 3) can be executed concurrently from a state A, and if they lead to different states B and C, then their exists two instructions 3 et 4 that can be executed respectively from B et C, leading to the same final state D.

Theorem 2 *The $\mathcal{SCL}-\mathcal{C}han$ language satisfies the diamond property.*

Let us give the main ideas of the proof. Concurrent instructions can be executed only by different indices. Conflicts could appear only if the same data can be written concurrently by two indices, or written by one and read by an other one. The sole possible cases occur when a send instruction is executed concurrently with a meaningful one (send,where/elsewhere, loopwhere) or with a receive :

- A structural clock can be read by a distant index that executes a receive, while the corresponding sending index writes it after the execution of a meaningful instruction. A meaningful instruction always replaces a clock by a greater one. The receiving index reads it to check if it is not lower than its local clock. Since we assume that the receive can be executed, the waiting condition is already satisfied from the initial state. Therefore, the execution of the meaningful instruction, before or after the receive, does not affect the receive execution.

- Conflicts could appear when send/receive instructions use the same channel. Since the waiting condition is satisfied for the receive, the sending index either owns a structural clock greater than the one of the receiving index, or one not comparable with it. So, the send adds a data in the channel that does not belong to the set BE (see the figure 1). Since the receiving index only reads and erases data of BE no conflict can occur.

3 A synchronous (data parallel) programming model

In the previous section, we presented a task parallel language based on a specific message passing model. Communications are dissociated from synchronizations. Synchronizations rely on index positions in the program structure and not on data receptions. In this section, we show that the asynchronous $\mathcal{SCL} - \mathcal{C}han$ execution model enables us to define a synchronous (data parallel) programming model. This model ensures that sequential and syntax-driven program reading is always possible. The central idea is that a synchronous computation is merely a particular asynchronous one and that all asynchronous computations are equivalent with respect to the returned result. Therefore, the programmer can design its programs in a synchronous framework, whereas the underlying execution model is loosely synchronized.

3.1 The model

Our synchronous model is in fact a classical data-parallel one. Each instruction is executed synchronously by all *active* indices while the other ones stay *idle*. Since all indices always have the same position, channel structure can be simplified. We still have one channel by index pair and by variable, but instead of LIFO lists, each channel is a *one position channel*. At the beginning of a $\mathcal{SCL} - \mathcal{C}han$ program all channels are initialized with the special value `nil`.

We now define a new instruction semantics for the synchronous programming model. We detail only instructions that take part in channel management. The others merely become synchronous.

Send: send X to A. It is a non blocking send. Each active index u evaluates the address $A|_u$ and the value $X|_u$ in its private memory. Then, u sends the value $X|_u$ to the index $A|_u$, i.e. it writes $X|_u$ in the one position channel $(u, A|_u, X)$. Thereby, the previous value stored in the channel is erased.

Receive: receive X from A into Y. Each active index u evaluates the address of the sending index $v = A|_u$ in its private memory. It assigns the value read in the (one position) channel $(A|_u, u, X)$ to $Y|_u$ and writes `nil` in the channel. So, a data can be received only once.

Concurrent conditioning: where B do S elsewhere T end. This instruction splits the set of active indices into two subsets of indices that execute respectively the instruction blocks S and T. The active indices that evaluate the pure expression B to $true$ execute S while other indices are idle. Next, T is executed by the indices that have evaluated B to $false$, while other indices are idle. Channel management must lead to independent branches as in the asynchronous framework. Therefore, we must temporarily hide send instructions that could be executed from one branch of the where/elsewhere to the other one. When two indices u and v execute different branches of the same where/elsewhere construct, the send instructions that may execute the index u are stored in a temporary channel, denoted by $(u, v, X)_{Tmp}$, whereas the receiver v still uses (u, v, X). The channel $(u, v, X)_{Tmp}$ is initialized with `nil`. When the conditioning ends, if the channel $(u, v, X)_{Tmp}$ is different from `nil` it is copied into the (u, v, X).

Remark. As a BSP program can simply be seen as a sequence of nested where/elsewhere instructions, $\mathcal{SCL} - \mathcal{C}han$ encompasses the BSP model [24]. Each outermost where/elsewhere represents a superstep and allows to hide communications until the superstep ends. So, $\mathcal{SCL} - \mathcal{C}han$ can provide a firm semantic framework for BSP, and it also gives an efficient execution model limiting synchronizations to effective communications.

3.2 Fundamental equivalence

We show that the synchronous execution of a $\mathcal{SCL} - \mathcal{C}han$ program yields the same result as any possible asynchronous execution. The proof relies on an equivalence result on the computations of the two models. Two computations are equivalent if starting with identical private memories, either they both end with the same ones or they both loop endless.

The following theorem states that a synchronous computation is no more than a particular computation of the asynchronous execution model. It ensures the coherency of the programming model with regard to the execution model.

Theorem 3 *Let us consider a synchronous computation C_s starting from a program S, with all the indices being active and all channels initialized with* `nil`*. Let C_a be an asynchronous computation such as all indices begin with the same program S, the same structural clock, empty LIFO channels and the same private memories as in the synchronous case. Then, the synchronous computation C_s and the asynchronous one C_a are equivalent.*

The proof relies on an induction on the program structure [21]. The intuitive idea is the following. At first, one shows that for each instruction execution in the synchronous model, one can build an equivalent asynchronous computation. It consists in the execution of the considered instruction by the active processors and no action for the others. Conversely, one considers an asynchronous computation where each index initially owns the same program and the same structural clock. Next, one shows that if the computation ends, one can build an other asynchronous computation starting from the same initial state that can be decomposed in steps corresponding to the transitions of the synchronous computation. This asynchronous computation ends with the same state since $\mathcal{SCL} - \mathcal{C}han$ is deadlock free and deterministic.

4 The translation function

We turn now to the translation of sequential programs into $\mathcal{SCL} - \mathcal{C}han$ ones. For sake of clarity, we focus on a subclass of sequential programs called simple programs.

```
for i = 1 to N do
  A[i+1] = A[i-1];                  }
  if A[i+1] ≠ 0 do                  }
     A[i] = A[i-1] * A[i+1];        }  = S
  end                               }
end
```

Figure 4: Example.

Definition 3 *A simple program S is a sequential program composed by assignments, conditionals, and* do *loops whose bounds are arbitrary functions of loop counters.*

The translation function, denoted by $Trans$, is defined by induction on the program structure. Loop counters are expanded in such a way that each index owns its counters. To express the mapping of data onto indices, we introduce a function $Owner$: the expression $Owner(V)$ denotes the index owning the variable V. The definition of $Owner$ can be easily extended to sets of variables: if Q is set of variables, $Owner(Q)$ denotes the set of indices owning at least one variable of Q. We impose the Owner Compute Rule : the sequential assignment $A := B$ is performed by the index $Owner(A)$, while the index $Owner(B)$ has to send B to $Owner(A)$.

Given a simple program S, we denote by $Concern(S)$ a superset of the set of variables which appear at left hand side of assignments. The set $Concern(S)$ owns *possibly* assigned variables of S. We denote by $Ref(Exp)$ a superset of the set of variables which are referred in the expression Exp. Variables belonging to $Ref(Exp)$ are *possibly* referred by the index evaluating Exp. The definition of $Ref(Exp)$ can be easily extended from expressions to programs: $Ref(S)$ denotes a superset of the set of variables which are acceded *for reading* in the program S.

Example. Let us consider the program of figure 4. A is a one dimensional array. This program iters N times a block S. For each iteration i, we may choose: $Concern(S) = \{A[i], A[i+1]\}$ and $Ref(S) = \{A[i-1], A[i+1]\}$.

msend Q to T end	mreceive Q

foreach $V \in Q$ do
  foreach $W \in T$ do
    where $Owner(V) = This\ and\ Owner(W) \neq This$ do
      send V to $Owner(W)$;
    end;
  end;
end;
	foreach $V \in Q$ do
	  where $Owner(V) \neq This$ do
	    receive V from $Owner(V)\ into\ Tmp$;
	    where $Tmp \neq$ `nil` *do*
	      $V := Tmp$;
	    end;
	  end;
	end;

Figure 5: Definition of the msend and mreceive instructions.

Remark. Note that since loop counters are distributed on each index, they do not have to be communicated. Therefore, they never appear in the sets Ref and $Concern$.

For sake of conciseness, we now introduce some new instructions.

We use a foreach $E \in Q$ do S end instruction such as each active index executes S for all element $E \in Q$ (Q is local to each index). Note the execution ordering is not specified.

We introduce the instruction msend Q to T end where Q and T are sets of variables. An index executing a msend instruction, sends all variables belonging to Q to all indices owning at least one variable belonging to T (see figure 5). We also define the instruction mreceive Q where Q is a set of variables (see figure 5). An index executing this instruction receives all the variables of Q from the indices which own them. When the value of a non-local variable is received for the first time, a local copy of the variable is created to store it. Values received afterwards will be stored in the same local variable. Note that for arrays, local copies of elements keep the same subscripts. If a `nil` value is received for $V \in Q$, then the value of the local copy of V is not updated. The intuitive idea is the following: if a given index u receives a `nil` value, it means that the value of V has not been updated by the index $Owner(V)$ since the last communication. In this case the current local value of V is correct and does not need to be updated.

We now describe the $Trans$ function. It respects the program structure. Loop nests are distributed on each index via a translation from DO loops to forwhere loops. Let us now detail the translation of assignments and conditional statements.

4.1 The translation of an assignment

The translation of the assignment $X[Exp_1] := Exp_2$ is divided this translation in three steps (see figure 6). First, each index owning a *possibly* referred variable $V \in Ref[Exp_2]$ sends it to all indices owning *possibly* assigned variables $W \in Concern(X[Exp_1] := Exp_2)$.

The second step concerns the evaluation of the address of the modified variable. We choose an index u responsible for the evaluation of the subscript Exp_1. Each index owning a *possibly* referred variable $V \in Ref(Exp_1)$ sends it to u. Then, the index u sends the value of Exp_1 to the index $Owner(X[Exp_1])$.

To formalize the choice of the index u responsible for the evaluation of Exp_1, we introduce a function *Resp* from sets of variables to indices. For any set Q of variables, $Resp(Q)$ chooses a variable $V \in Q$ and returns $u = Owner(V)$.

In the third step, all indices owning a *possibly* assigned variable (with regard to the set $Concern(X[Exp_1] := Exp_2)$) perform a receive instruction. Only one of them will receive the message sent at the second step. This index performs the assignment.

Remarks.

1. The index $u = Resp(Concern(X[Exp_1] := Exp_2))$ belongs to the set of indices owning possibly assigned variables.

2. If Exp_1 is a function of loop counters, it is possible to locally determine which element of the array has to be assigned. In this case, $Owner(Exp_1)$ is a function of loop counters too and it is judicious to choose *Resp* such as $Resp(Concern(X[Exp_1] := Exp_2)) = Owner(X[Exp_1])$. The third step can then be modified such as indices do not receive Tmp, but locally evaluate Exp_1 to determine if they have to perform the assignment.

3. Note that some sent values are never received. The semantics of $\mathcal{SCL} - \mathcal{C}han$, and especially the LIFO management of channels, ensures that useless messages (involved by Ref and $Concern$) do not disrupt the needed communications.

$Trans($ $X[Exp_1] := Exp_2$ $)$	*First step*	msend $Ref(Exp_2)$ to $Concern(X[Exp_1] := Exp_2)$;

	Second step	msend $Ref(Exp_1)$ to $Resp(Concern(X[Exp_1] := Exp_2))$;
		where $This = Resp(Concern(X[Exp_1] := Exp_2))$ do
		  mreceive $Ref(Exp_1)$;
		  $Tmp := Exp_1$;
		  send Tmp to $Owner(X[Tmp])$;
		end;

	Third step	where $This \in Owner(Concern(X[Exp_1] := Exp_2))$ do
		  receive Tmp from $Resp(Concern(X[Exp_1] := Exp_2))$
		      into Tmp;
		  where $Tmp \neq \texttt{nil}$ do
		    mreceive $Ref(Exp_2)$;
		    $X[Tmp] := Exp_2$;
		  end;
		end

Figure 6: The translation of an assignment.

4.2 The translation of a conditional statement

The translation of the conditional construct if Exp then S end is made of three similar steps (see figure 7). First, among indices owning *possibly* referred variables in S, some do not have *possibly* modified variables in S. In this case, they can send the values of the referred variables without entering the conditional construct. The second step concerns the evaluation of the expression Exp. All indices owning *possibly* referred variables in Exp send values of referred variables to the index $Resp(Concern(S))$. Then, the index $Resp(Concern(S))$ evaluates Exp and sends the result to all the indices owning a *possibly* assigned variable in S. The third step is conditioned by the result of the evaluation of the expression. It consists in the execution of the program S by all concerned indices.

Remarks.

1. If only one index has to execute S, the where construct of the second step becomes useless. Then, the evaluation of Exp can be performed in the third step. Moreover, the msend instructions of the first and the second step can be gathered into one single msend.

$Trans($ if Exp then S endif $)$	*First step*	msend $Ref(S)$ to $Concern(S)$;

	Second step	msend $Ref(Exp)$ to $Resp(Concern(S))$;
		where $This = Resp(Concern(S))$ do
		  mreceive $Ref(Exp)$;
		  $Tmp := Exp$;
		  msend Tmp to $Concern(S)$;
		end;

	Third step	where $This \in Owner(Concern(S))$ do
		  receive Tmp from $Resp(Concern(S))$ *into* Tmp;
		  where Tmp do
		    $Trans(S)$;
		  end;
		end

Figure 7: The translation of a conditional statement

2. Our translation function can be easily extended to arbitrary DO loops. In fact, loop bounds can be evaluated during the execution, using a method similar to the evaluation of the condition in the second step of the translation of a conditional.

3. The code generation does not take account of the semantics of the *Owner* function, therefore the produced program is totally independent of the mapping. It is possible to specify the function *Owner* before or after the parallelization.

4.3 Example

To illustrate our translation scheme, we reuse the example figure 4. We have already computed supersets $Ref(S)$ and $Concern(S)$. The figure 8 displays the translation of the sequential program. The index $Owner(A[i])$ can execute the instruction mreceive $\{A[i+1]\}$ (marked **2**) only if the index $Owner(A[i+1])$ have evaluated the conditional (marked **1**). It ensures that the index $Owner(A[i])$ receives the updated value of $A[i+1]$. Underlined instructions are useless (for example, communication instructions from an index to itself). They can be easily suppressed thanks to a static analysis performed at compile time.

```
for i := 1  to  100  do
  msend  {A[i]} to {A[i + 1]};
  msend  {} to {A[i + 1]};
  where  This = Owner(A[i + 1]) do
    mreceive  {};
    Tmp := i + 1;
    send  Tmp  to  Owner(A[i + 1]);
  end;
  where  This = Owner(A[i + 1]) do
    receive  Tmp  from  Owner(A[i + 1]) into  Tmp;
    where  Tmp ≠nil do
      mreceive  {A[i]};
      A[Tmp] := A[i];
    end;
  end;
  msend  {A[i + 1], A[i − 1]} to {A[i]};
  msend  {A[i + 1]} to {A[i]};
  where  This = Owner(A[i]) do            1
    mreceive  {A[i + 1]};            2
    Tmp := (A[i + 1] ≠ 0);
    msend  {Tmp} to {A[i]};
  end;
  where  This ∈ {A[i]}  do
    receive  Tmp  from  Owner(A[i])  into  Tmp;
    where  Tmp do
      msend  {A[i − 1], A[i + 1]} to {A[i]};
      msend  {} to {A[i]};
      where  This = Owner(A[i]) do
        mreceive  {};
        Tmp := i;
        send  Tmp  to  Owner(A[i]);
      end;
      where  This = Owner(A[i]) do
        receive  Tmp  from  Owner(A[i])
          into  Tmp;
        where  Tmp ≠nil do
          mreceive  {A[i − 1], A[i + 1]};
          A[Tmp] := A[i − 1] ∗ A[i + 1];
        end;
      end;
    end;
```

Figure 8: Translation of the program figure 4

4.4 Correctness of the translation

We turn now to the problem of correctness of the translation function $Trans$.

Definition 4 (Correctness Property) *A sequential program S satisfies the Correctness Property if S and $Trans(S)$, starting from the same memory state, compute the same final values.*

The next theorem can be proved by induction on the program structure.

Theorem 4 *All simple programs satisfy the Correctness Property.*

The intuitive idea of the induction is the following. As the $\mathcal{SCL} - \mathcal{C}han$ language ensures that all programs are deterministic (theorem 3), we can focus on synchronous computations of the translated program. Each induction step relies on the following observation: as for any program S, the set $Ref(S)$ (resp. $Concern(S)$) includes all referred variables in S (resp. all assigned variables in S) owned by index u , all expected values are sent and received.

5 Example : A Gauss-Seidel like algorithm

Our next example is inspired by the Gauss-Seidel relaxation algorithm. It consists in going through a two dimensional array from right to left and top to bottom, while computing the product of the values of the four neighbors. To show the interest of unmatching communications, we modify the state of the considered component only if the result is not null (not to propagate null values). It is an interesting problem to parallelize since it includes numerous dependences and anti-dependences.

We apply the translation function to the sequential program S displayed figure 9. All variables are arrays with loop counters as subscripts. Therefore, we can compute supersets Ref (respectively $Concern$) which exactly matches the sets of variables referred (respectively assigned) during the execution. Let us denote by S the assignment $A[I,J] := A[I,J-1] * A[I+1,J] * A[I,J+1] * A[I-1,J]$. As we have $Concern(S) = \{A[I,J]\}$, it leads to $Resp(Concern(S)) = Owner(A[I,J])$. Therefore, the second step in the translation of the assignment and the communications relating to the broadcast of the condition in the translation of the conditional become useless (cf. remark 2 section 4.1 and remark 1 section 4.2).

Some simple optimizations can be performed on the translated program. In the nested conditional statement, the condition $This = Owner(A[I,J])$ is redundant. Therefore, the underlined conditional statements are useless. The first instruction msend communicates all needed variables to the index $Owner(A[I,J])$. Since these variables are not modified afterwards, other sending instructions (underlined in the program) become useless. Moreover, since all channels are empty, the underlined instruction mreceive does nothing and just terminates: the index $Owner(A[I,J])$ uses its local copies of distant variables. Therefore, the (underlined) mreceive can be deleted too.

6 Example: Cholesky factorization

In this section we parallelize the classical Cholesky factorization algorithm (see the program figure 10). In contrast to the previous example, the program does not reduce to a perfect loop nest. With the aim to exploit sparsity, we introduce a conditional control construct: the value

The sequential program

```
real:A[0:N,0:N]
do I=1,N-1
   do J=1,N-1
     if A[I,J-1]≠ 0 then
       if A[I+1,J]≠ 0 then
         if A[I,J+1]≠ 0 then
           if A[I-1,J] ≠ 0 then
             A[I,J]=A[I,J-1]*A[I+1,J]*A[I,J+1]*A[I-1,J]
           endif
         endif
       endif
     endif
   enddo
enddo
```

The translation

```
forwhere I := 1 to N − 1 do
  forwhere J := 1 to N − 1 do
    msend {A[I, J − 1], A[I + 1, J], A[I, J + 1], A[I − 1, J]} to Owner(A[I, J]);
    where This = Owner(A[I, J]) do
      mreceive {A[I, J − 1]};
      where A[I, J − 1] ≠ 0 do
        msend {A[I, J − 1], A[I + 1, J], A[I, J + 1], A[I − 1, J]} to Owner(A[I, J]);
        where This = Owner(A[I, J]) do
          mreceive {A[I + 1, J]};
          where A[I + 1, J] ≠ 0 do
            msend {A[I, J − 1], A[I + 1, J], A[I, J + 1], A[I − 1, J]} to Owner(A[I, J]);
            where This = Owner(A[I, J]) do
              receive {A[I, J + 1]};
              where A[I, J + 1] ≠ 0 do
                msend {A[I, J − 1], A[I + 1, J], A[I, J + 1], A[I − 1, J]} to Owner(A[I, J]);
                where This = Owner(A[I, J]) do
                  mreceive {A[I − 1, J]};
                  where A[I − 1, J] ≠ 0 do
                    msend {A[I, J − 1], A[I + 1, J], A[I, J + 1], A[I − 1, J]} to Owner(A[I, J]);
                    where This = Owner(A[I, J]) do
                      mreceive {A[I, J − 1], A[I + 1, J], (A[I, J + 1], (A[I − 1, J]};
                      A[I, J] := A[I, J − 1] ∗ A[I + 1, J] ∗ A[I, J + 1] ∗ A[I − 1, J]
                    end
                  end
                end
              end
            end
          end
        end
      end
    end
  end
end
```

Figure 9: The Gauss-Seidel like algorithm and the corresponding translation.

$A[j,k]$ is read only if $A[i,k]$ is not null (see instruction marked in (1)). Since the value $A[i,k]$ is not modified in the innermost loop, the test can be performed outside of this loop.

As in the previous example, all variables are arrays with loop counters as subscripts. Therefore, we can choose supersets Ref and $Concern$ which respectively match the sets of variables read and assigned during the execution (for example $Ref(A[I,J] := A[I,J] - A[J,K] * A[I,K]) = \{A[I,J], A[J,K], A[I,K]\}$). We also choose $Resp(S) = Owner(A[I,I])$ (see figure 10 for the instructions corresponding to S).

Remarks.

1. Some communications and computations are saved if the communicated value $A[i,k]$ is null. This capability is directly linked to the $\mathcal{SCL} - \mathcal{C}han$ semantics of communications. Since send instructions can be performed without matching receive instructions, it is possible to manage conditionals with receive instructions nested inside.

2. As in the previous example, straightforward syntactic optimizations allow to suppress all underlined instructions.

7 Optimizations and mapping

In this section, the parallelized programs of the Gauss-Seidel like algorithm and of the Cholesky factorization are considered. The parallelized code does not depend on the choice of the mapping function $Owner$. However, it is possible to perform additional optimizations by taking this function semantics into account. The aim is to reduce communication and computation overhead. On the one hand, a refined data dependence analysis allows to move some send instructions and to partially reorganize some control structures in order to favor computation and communication overlap. It also leads to gather communications. On the other hand, since the translation function distributes each loop iteration on all indices, an index u may execute some useless iterations. It appears that it is often possible to detect at compile time such iterations thanks to a static analysis, and thereby to avoid their execution.

Sequential program

```
REAL,SPARSE ::  A[N,N]
do i = 1,N
   do k = 1,i-1
     if A[i,k] ≠ 0 then          (1)
        do j = i,N             }
           A[j,i]=A[j,i]       }
             -A[j,k]*A[i,k]    } S
        enddo                  }
     endif
   enddo
   A[i,i]=sqrt(A[i,i])
   do d = i+1,N
     A[d,i]=A[d,i]/A[i,i]
   enddo
enddo
```

Translated program

```
forwhere I := 1 to N + 1 do
   forwhere K := 1 to I − 1 do
      msend (A[I,I],..,A[N + 1,I])
        to (A[I,I],..,A[N + 1,I]);
      msend (A[I,K],..,A[N + 1,K])
        to (A[I,I],..,A[N + 1,I]);
      msend (A[I,K]) to (A[I,I]);
      where This = Owner(A[I,I]) do
        mreceive (A[I,K]);
        Tmp := (A[I,K] ≠ 0);
        msend Tmp to (A[I,I],..,A[N + 1,I]);
      end;
      where This ∈
        {Owner(A[I,I]),..,Owner(A[N + 1,I])} do
        receive Tmp from Owner(A[I,I]);
        where Tmp do
          forwhere J := I to N do
             msend (A[J,I],A[I,K],A[J,K]) to A[J,I];
             where This = Owner(A[J,I]) do
               mreceive (A[J,I],A[J,K],A[I,K]);
               A[J,I] := A[J,I] − A[J,K] ∗ A[I,K]
             end
          end
        end
      end
   end ;
   msend (A[I,I]) to Owner(A[I,I]);
   where This = Owner(A[I,I] do
      mreceive (A[I,I]);
      A[I,I] := √A[I,I];
   end ;
   forwhere D := I + 1 to N do
      msend (A[D,I],A[I,I]) to Owner(A[D,I]);
      where This = Owner(A[D,I]) do
        mreceive (A[D,I],A[I,I]);
        A[D,I] := A[D,I]/A[I,I]
      end
   end
end
```

Figure 10: The Cholesky factorization algorithm and the corresponding translation.

7.1 Gauss-Seidel like algorithm

Let us begin with the Gauss-Seidel like algorithm (see section 5). Let us denote by Pe_Nb the number of available processors. We choose a fixed mapping, distributing the matrix A by blocks of columns among a vector of Pe_Nb indices numbered from 0 to $Pe\_Nb - 1$. For sake of convenience, we assume that N is a multiple of Pe_Nb and we denote by L the number $L = N/Pe\_Nb$. Each index owns $N \times L$ components. This mapping corresponds to a simple $Owner$ function: $Owner(A[I, J]) = (J\ div\ L)$. Matrix components are renamed so that each index owns an array A of $N \times L$ components. The innermost loop is split into two loops, a local one (loop index K) for the computation by each index of the piece of line stored in its private memory and a global one (loop index J) to coordinate computations between indices. Since the indices 0 and Pe_Nb-1 do not have to perform computations on one of the columns they own, we also unfold the first and last iterations of the outermost loop. Note that this code reorganization is performed without changing the logical ordering between the potentially matching send/receive instructions. So, the program semantics is not modified. Moreover, send instructions sending data to a left hand side index are linked to anti-dependences. Therefore, they can be moved at the beginning of loop I without changing program semantics too. This last modification favors computation and communication overlap. Then, we obtain the $\mathcal{SCL} - \mathcal{C}han$ program displayed figure 11.

Remark.

In the underlined loop structure, as the underlined conditional structure is a simple equality test between $This$ and the loop counter J. Each index is active only during one iteration. The other iterations are useless but to properly increment structural clocks. In such a configuration, an idle index may skip these useless iterations, while updating its clock.

```
forwhere I := 1 to N − 2 do
 where 0 < This AND This < Pe_Nb do
  send A[I,0] to This − 1;
 end;
 where This = 0 do
  forwhere K := 1 to L − 2 do
   Compute A[I,K];
  end;
  where local neighbors of A[I,L − 1] ≠ 0
   receive A[I,0] from 1 into R;
   Compute A[I,L − 1];
  end;
  send A[I,L − 1] to 1;
 end;
 where 0 < This AND This < Pe_Nb − 1 do
  forwhere J := 1 to Pe_Nb − 2 do
   where This = J do
    where local neighbors of A[I,0] ≠ 0 do
     receive A[I,L − 1] from J − 1 into R;
     Compute A[I,0];
    end;
    forwhere K := 1 to L − 2 do
     Compute A[I,K];
    end;
    where local neighbors of A[I,L − 1] ≠ 0 do
     receive A[I,0] from J + 1 into R;
     Compute A[I,L − 1];
    end;
    send A[I,L − 1] to J + 1;
   end;
  end;
 end;
 where This = Pe_Nb − 1 do
  where local neighbors of A[I,0] ≠ 0 do
   receive A[I,L − 1] from This − 1 into R;
   Compute A[I,0];
  end;
  forwhere K := 1 to L − 2 do
   Compute A[I,K];
  end;
 end;
end;
```

Figure 11: The optimized algorithm corresponding to the translation of the Gauss-Seidel like algorithm.

7.2 Cholesky

In the Cholesky factorization algorithm (see section 6), computations take place on the lower triangular part of the matrix and data dependences involve computations from the first column towards the last one. Therefore, since a cyclic column distribution balances work load, the Cholesky factorization algorithm is mapped on Pe_Nb indices by distributing cyclically the columns of the matrix A. Each index owns an array A of $N \times (N/Pe\_Nb)$ components. We define the function $Owner$ as follow: $Owner(A[I,J]) \;=\; J \; mod \; Pe\_Nb$. As for the Gauss-Seidel like algorithm, matrix components are renamed so that each index owns an array A of $N \times (N/Pe\_Nb)$ entries. Optimizations are performed in two steps.

The first optimization, corresponding to the program displayed figure 12, is based on the following observation. The computation performed to obtain a new value of a column I (before the normalization) relies on a subtraction of two entries of right hand side columns. Therefore, it is possible to do a partial computation on each index, using only local data involved in the computation of the column I. As products are no more executed by the receiver, but by the sender, the sender has to test the condition $A[I,K] \neq 0$ before performing the corresponding computation and sending. Thus, the program performs more receive instructions than send ones: to receive `nil` is interpreted as a reception of a null result.

The second optimization corresponds to the program figure 13. Since there is no data dependence between the elements of a sent column, we may gather all these elements in a single message. In this way, an index owning a column I can execute only one send instruction after all the partial computation corresponding to this column. In order to save a communication, a $ToSend$ variable is set to $true$ if one pivot is not null and to $false$ otherwise. Then, the considered column is sent only if $ToSend$ is $true$. Note that we denote by $S[0 : N-I-1]$ a message made up of the components from 0 to $N-I-1$ of the array S.

```
forwhere I := 0 to N − 1 do
 I_p := I div Pe_Nb;
 where This < I AND This ≠ I mod Pe_Nb do
  forwhere D := 0 to N − I − 1 do S[D] := 0 end;
  forwhere K := 0 to (I − This) div Pe_Nb − 1 do
   where A[I,K] ≠ 0 do
    forwhere D = I to D < N do
     S[D − I] := S[D − I] + A[D,K] * A[I,K];
     send S[D − I] to I mod Pe_Nb;
    end;
   end;
  end;
 end
 where This = I mod Pe_Nb do
  forwhere K := 0 to (I − This) div Pe_Nb − 1 do
   forwhere D := I to N − 1 do
    A[D,I_p] = A[D,I_p] − A[D,K] * A[I,K];
   end;
  end;
```

```
  forwhere K := 0 to min{Pe_Nb; I} − 1 do
   where This ≠ K do
    forwhere D := I to N − 1 do
     receive S[D − I] from K into R[D − I];
     where R ≠nil do
      A[D,I_p] := A[D,I_p] − R[D − I];
     end;
    end;
   end;
  end;
 end;
 where This = I mod Pe_Nb do
  A[I,I_p] := √A[I,I_p];
  forwhere D := I + 1 to N − 1 do
   A[D,I_p] := A[D,I_p]/A[I,I_p];
  end;
 end;
end;
```

Figure 12: The first optimization corresponding to the translation of the Cholesky factorization.

```
forwhere I := 0 to N − 1 do
 I_p := I div Pe_Nb;
 where This < I AND This ≠ I mod Pe_Nb do
  forwhere D := 0 to N − I − 1 do S[D] := 0 end;
  ToSend := false;
  forwhere K := 0 to (I − This) div Pe_Nb − 1 do
   where A[I,K] ≠ 0 do
    ToSend := true;
    forwhere D = I to D < N do
     S[D − I] := S[D − I] + A[D,K] * A[I,K];
    end;
   end;
  end;
  where ToSend ≠ 0 do
   send S[0 : N − I − 1] to I mod Pe_Nb;
  end;
 end;
 where This = I mod Pe_Nb do
  forwhere K := 0 to (I − This) div Pe_Nb − 1 do
   forwhere D := I to N − 1 do
    A[D,I_p] = A[D,I_p] − A[D,K] * A[I,K];
   end;
  end;
```

```
  forwhere K := 0 to min{Pe_Nb; I} − 1 do
   where This ≠ K do
    receive S[0 : N − I − 1] from K into R;
    where R ≠nil do
     forwhere D := I to N − 1 do
      A[D,I_p] := A[D,I_p] − R[D − I];
     end;
    end;
   end;
  end;
 end;
 where This = I mod Pe_Nb do
  A[I,I_p] := √A[I,I_p];
  forwhere D := I + 1 to N − 1 do
   A[D,I_p] := A[D,I_p]/A[I,I_p];
  end;
 end;
end;
```

Figure 13: The second optimization corresponding to the translation of the Cholesky factorization.

8 Implementation on Cray T3D

Let us now complement our theoretical results by performance results. They are obtained from an implementation of the $\mathcal{SCL} - \mathcal{C}han$ asynchronous abstract machine on a Cray T3D. Our main goal is to test the overhead associated to structural clock machinery and the practical interest of send/receive unmatching. We compare our results with the ones provided by equivalent MPI programs. We use an optimized version of the MPI library, developed at the University of Edinburgh for T3D computers [5].

8.1 The implementation

We developed a simple library on Cray T3D. It provides all the functions necessary to handle structural clocks and $\mathcal{SCL} - \mathcal{C}han$ channels. Send and receive functions manage data communications according to the $\mathcal{SCL} - \mathcal{C}han$ paradigms. This library mainly uses the *put* communication instruction of Cray SHMEM library [6]. The *put* allows one processor to write directly into another processor's memory without that processor be notified. It is a very efficient instruction with a low latency, close to $1\mu s$. Note that in the $\mathcal{SCL} - \mathcal{C}han$ model, a structural clock sent after a data must also be received after it: when a receiving index is notified that the waiting condition is satisfied, all data sent before must be already stored in channels. Therefore, communication ordering between data and structural clocks must be respected. This is ensured by the Cray T3D routing protocol.

In section 2, we presented a $\mathcal{SCL} - \mathcal{C}han$ asynchronous semantics based on possibly infinite channel sizes. Nevertheless, a realistic implementation requires limited channel sizes. Therefore, we had to adapt the communication protocol to manage such channels. The send becomes a potentially blocking instruction. Indices count the number of send instructions performed on each channel. A send is blocked as long as the target channel is full. As long as it is blocked (because of a send or a receive), an index v flushes its channels (u, v, X) of useless data. Next, for each channel (u, v, X), the index v sends acknowledgments to the index u according to the freed positions. An index that has finished program execution broadcasts a special acknowledgment not to block indices that could send a datum to it. This protocol is deadlock free if channel sizes are strictly greater than two.

8.2 Experimental results

8.2.1 Ping pong test

As a first test, we programmed a ping pong between two indices. An integer is exchanged numerous times between indices. $\mathcal{SCL}-\mathcal{C}han$ requires an average time of $25.5\mu s$ for sending an integer and receiving it back, whereas MPI needs more than $65\mu s$. The gap comes from the communication protocol. Since only a single writer and a single reader are associated to each channel, the $\mathcal{SCL}-\mathcal{C}han$ communication paradigm needs a simple communication protocol, lighter than the MPI one.

8.2.2 Gauss-Seidel like algorithm

In this section the optimized Gauss-Seidel program presented in 7.1 (see figure 11) is considered.

Structural clock management: Let us detail structural clock management. Since the execution of a send can depend on the enclosing control structures (see section 2.3), we have defined all control structures as meaningful instructions. However, structural clock updates are optimized, since only control structures that enclose a send may be taken into account for structural clock updates.

We developed several optimization levels for structural clock communication. The idea is that structural clocks are useful only for indices executing receive instructions. First of all, we did not optimize, i.e. a clock is broadcasted each time it is modified but after a send. When a send is performed, the sole index that is interested in knowing the clock of the sender is the receiver. Therefore, the new clock computed after the execution of a send instruction is sent only to the receiver. This non optimized version is called "send all". Since data communications take place only between neighbor indices, a simple optimization consists in sending clocks to neighbors only ("send neighbors" version). The most optimized version sends clocks only one

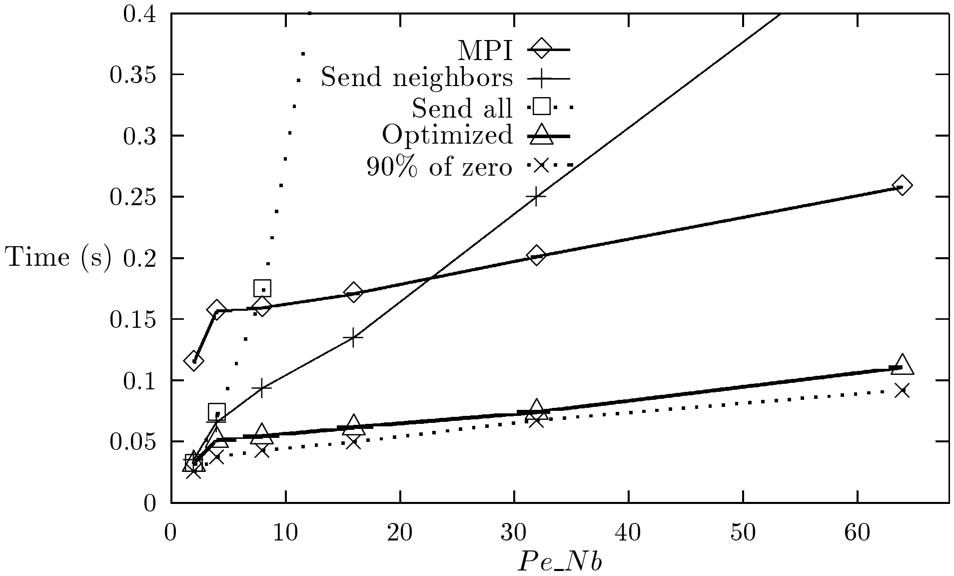

Figure 14: Performance results for the Gauss-Seidel like algorithm. Each index owns a matrix of size 1000×5.

time between a send and a receive, as soon as the clock is great enough to validate the waiting condition ("optimized" version).

MPI program: For the equivalent MPI program, we reused the same communication pattern without the conditional structures that control send and receive instructions to avoid send/receive unmatching. We replaced send/receive instructions by `MPI_Isend/MPI_Recv` ones and we removed all instructions of structural clock management.

Results without null values: At first, we performed tests without null values and with various values of Pe_Nb (Fig. 14). Each index always owns the same volume of data ($N = 1000$, $L = 5$). It enables us to see the variations of the communication overhead, according to the indices only. The value of L is chosen sufficiently small, so as not to mask the communication cost behind computations. It appears that, with the two first optimization levels, "send all" and "send neighbors", structural clock overhead rapidly increases. However, the "optimized" version yields computations three times as speed as MPI ones, in keeping with the results of the ping pong test.

Results with null values: We present tests performed with $Pe\_Nb = 16$, $N = 1000$ and

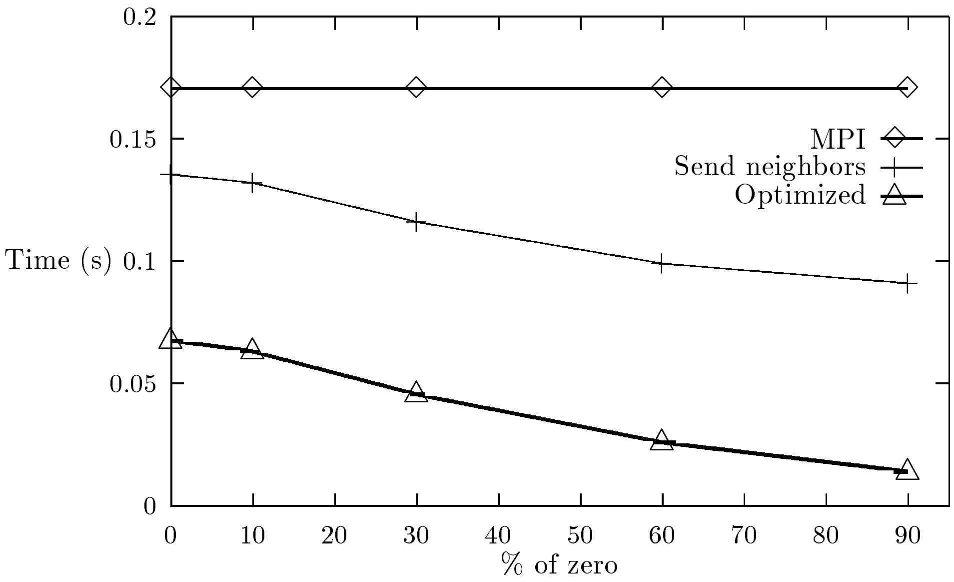

Figure 15: Performance results for Gauss-Seidel like algorithm, using various ratio of null values. Each index owns a matrix of size 1000×5 and $Pe\_Nb = 16$.

$L = 5$ (Fig. 15). We ran a "send neighbor" version and an "optimized" version with different ratio of zero values. Obviously since send receive matching is mandatory, MPI yields a constant computation time. We extended the test with 90% of zero to various number of indices (Fig. 14). As expected, since the number of communications executed by each index is independent of Pe_Nb, performance improvement is constant.

8.2.3 Cholesky factorization

In this section, the optimized Cholesky program presented in section 7.2 is considered (see figure 13). The main interest is to highlight our approach capabilities to take advantage of message gathering.

As for the Gauss-Seidel algorithm, only two meaningful instructions are relevant: `send` instructions and enclosing `where` constructs. Since each where construct contains exactly one `send`, it is useless to hide the number of meaningful instruction executed. Therefore, indices that enter the outermost `where` construct (underlined two times - see figure 13) increase their clocks after the execution of the innermost `where` (underlined one time). Other indices increase their clocks

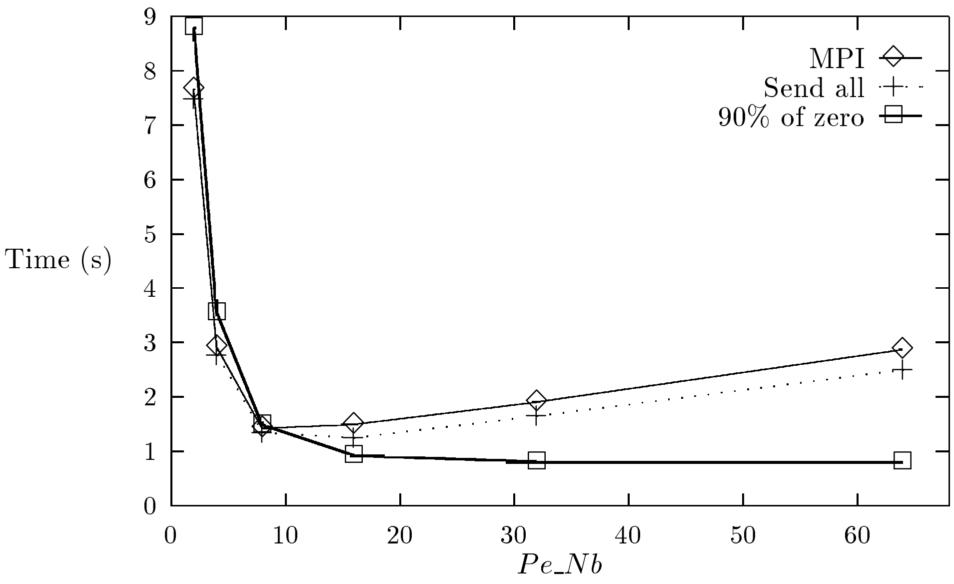

Figure 16: Performance results for the Cholesky algorithm. A matrix of size 640×640 is cyclically distributed by columns.

after the condition evaluation. The "send all" version corresponds to the sending of a clock by the sender to the receiver after the innermost where, and to a sending to all other indices after the condition evaluation. The "optimized" version simply consists in removing the second clock sending: as it does not take place between a send and a receive, it is useless. Note that even when a send is not executed because the values to send are null, the sending index must increase and send its clock to the receiver so as not to delay it and to avoid deadlocks.

We performed tests for various values of Pe_Nb. To highlight the efficiency of the parallelization with regards to the number of indices, we used a matrix of fixed size 640x640: the volume of data owned by each index decreases while Pe_Nb increases. Performances for "send all" and "optimized" versions are very closed. We so only display the results of the first version (Fig. 16).

We tested the sparse version with 90%, 94% and 98% of zero. As communications correspond to long messages, to avoid useless send instructions improves significantly the performances.

Remark. It turn out that an SHMEM put instruction is very well suited to implementing the $\mathcal{SCL} - \mathcal{C}han$ abstract machine. An implementation on the top of MPI is straightforward, but would certainly be less efficient. However, the overhead would certainly be comparable to the one due to the use of the MPI library, instead of machine primitives, in classical message passing programming.

9 Conclusion

In this paper, we presented a task parallel and SPMD language, relying on a new message passing communication paradigm. The main originality of our model is based on a *logical* ordering of control flows. It allows to unify in a single framework a synchronous programming model, thereby facilitating the mastery of programs, and an asynchronous execution model yielding efficient executions. Since send/receive matching is not mandatory $\mathcal{SCL}-\mathcal{C}han$ can easily handle irregular remote data accesses.

We proposed an automatic parallelization method mainly based on this capability. It allows to translate a sequential program with irregular control structures, such as `if` statements or arrays with subscripts evaluable only at run-time, into a $\mathcal{SCL}-\mathcal{C}han$ one. The generated code is independent of the mapping, thereby facilitating dynamic load balancing. Moreover, to take a "good" data distribution into account can lead to additional optimizations.

Experimental results on Cray T3D show that $\mathcal{SCL}-\mathcal{C}han$ performances are comparable to MPI ones. Thus, the structural clock machinery does not impose a significant overhead. It also appears that send/receive unmatching can be easily and efficiently exploited to reduce communications. Structural clocks appears to be an interesting tool for the design of libraries and execution supports for irregular applications.

Since a BSP program can be seen as a sequence of where/elsewhere constructs, the theoretical framework designed for $\mathcal{SCL}-\mathcal{C}han$ can be reused for BSP as well. The corresponding execution model does not involve mandatory global synchronizations.

We are currently working on a model of performance analysis able to take into account $\mathcal{SCL}-\mathcal{C}han$ expressiveness. New developments in automatic parallelization, such as WHILE loops parallelization, are under progress. We plan to extend $\mathcal{SCL}-\mathcal{C}han$ by dynamic constructs. So, since our automatic parallelization can yield fine grain parallelism, it will be possible to perform dynamic load balancing by thread migration.

Acknowledgement. We thank the anonymous referee for useful remarks.

References

[1] B. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, 1991.

[2] C. Bareau, B. Caillaud, C. Jard, and R. Thoraval. Correctness of Automated Distribution of Sequential Programs. Technical Report 665, Laboratoire de l'Informatique et du Parallélisme, Rennes, France, 1992.

[3] L. Bougé, D. Cachera, Y. L. Guyadec, G. Utard, and B. Virot. Formal Validation of Data-Parallel Programs: Introducing the Assertional Approach. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *LNCS*, pages 197–219. Springer-Verlag, 1996.

[4] L. Bougé, D. Cachera, Y. L. Guyadec, G. Utard, and B. Virot. Formal Validation of Data-Parallel Programs: a Two-Component Assertional Proof System for a Simple Language. *Theoretical Computer Science B*, 189(1-2):71–107, 1997.

[5] K. Cameron, L. J. Clarke, and A. G. Smith. CRI/EPCC MPI for CRAY T3D. In *First European Cray T3D Workshop.* EPFL, September 1995.

[6] Cray Research. *Application Programmer Library Reference MANUAL.*

[7] C. G. Diderich and M. Gengler. Synchronization as a Strategy for Designing Efficient Parallel Algorithms. In *Parallel Algorithms for Irregularily Structured Problems (IRREGULAR'96)*, pages 229–236. LNCS, August 1996.

[8] F.Delaplace and R.Adle. Extension of the Dependence Analysis for the Sparse Computation. Technical Report 14-95, LAMI, Université d'Evry, France, 1995.

[9] P. Feautrier. Automatic Parallelization in the Polytope Model. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *LNCS*, pages 79–100. Springer-Verlag, 1996.

[10] I. Foster. *Designing and Building Parallel Programs.* Addison–Wesley, 1994.

[11] W. Gropp, E. Lusk, and A. Skjellum. *USING MPI Portable Parallel Programming with the Message-Passing Interface.* Scientific and Engineering Computation Series. The MIT Press, 1994.

[12] 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, pages 732–741. Springer-Verlag, 1996.

[13] Y. L. Guyadec, E. Melin, B. Raffin, X. Rebeuf, and B. Virot. Structural Clocks for a Loosely Synchronized Data-Parallel Language. In *Second International Conference on Massively Parallel Computing Systems - MPCS'96*, pages 482–489, Ischia, Italy, May 1996. IEEE Computer Society Press.

[14] Y. L. Guyadec and B. Virot. Sequential-like Proofs of Data-Parallel Programs. *Parallel Processing Letters*, 6(3):415–426, 1996.

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

[16] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[17] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–564, 1978.

[18] W. F. McColl. BSP Programming. In G. E. Blelloch, K. M. Chandy, and S. Jagannathan, editors, *Proceedings of DIMACS Workshop*, volume 18 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 21–35, Princeton, May 1994. American Mathematical Society.

[19] E. Melin, B. Raffin, X. Rebeuf, and B. Virot. A General but Simple Technique to Handle Asynchronous Data-Parallel Control Structures. In *Fifth Euromicro Workshop on Parallel and Distributed Processing - PDP'97*, pages 189–196, London, United Kingdom, January 1997. IEEE Computer Society Press.

[20] E. Melin, B. Raffin, X. Rebeuf, and B. Virot. $\mathcal{SCL} - \mathcal{C}han$: An Asynchronous Data-Parallel Language for Irregular Algorithms. In *Second International Workshop on High-Level Parallel Programming Models and Supportive Environments - HIPS'97 (in conjonction with 11th International Parallel Processing Symposium - IPPS'97)*, pages 64–71, Geneva, Switzerland, April 1997. IEEE Computer Society Press.

[21] B. Raffin. *Un Modèle Structuré de Communication et de Synchronisation pour le Parallélisme de Tâches.* PhD thesis, LIFO, Université d'Orléans, France, 1997.

[22] R. S. Schreiber. An Introduction to HPF. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *LNCS*, pages 27–41. Springer-Verlag, 1996.

[23] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI, The Complete Reference.* Scientific and Engineering Computation. The MIT Press, 1996.

[24] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

List of Figures

Emmanuel Melin is currently working towards the Ph.D. degree in computer science at the University of Orléans. His research interests include the study of synchronization models and the design of translation functions of sequential program into parallel one using such models.

Bruno Raffin has obtained in 1997 a PhD from the University of Orléans after a DEA at the Ecole Normale Supérieure de Lyon. Its dissertation is related to the design of structured communication and synchronization models for task parallelism. At the moment, he performs researches at the Iowa State University, USA. He works on the performance evaluation of parallel computers, with a particular emphasis on scalability problems. Neural networks and real-time systems are his other main centers of interests.

Xavier Rebeuf is currently working towards the Ph.D. degree in computer sciences at the University of Orléans. His research interests include the study of synchronization models and the design of complexity function for task parallelism.

Bernard Virot joined the LIFO laboratory in 1988. He is currently co–responsible for the Parallel Programming Research Group. His main research interests focus on program semantics and formal validation.