$\mathcal{SCL} - \mathcal{C}han$: an Asynchronous Data-Parallel Language for Irregular Algorithms

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

Parallelism suffers from a lack of programming languages both simple to handle and able to take advantage of the power of present parallel computers. If parallelism expression is too high level, compilers have to perform complex optimizations leading often to poor performances. One the other hand, too low level parallelism transfers difficulties toward the programmer.

In this paper, we propose a new programming language that integrates both a synchronous data-parallel programming model and an asynchronous execution model. The synchronous data-parallel programming model allows a safe program designing. The asynchronous execution model yields an efficient execution on present MIMD architectures without any program transformation. Our language relies on a logical instruction ordering exploited by specific send/receive communications. It allows to express only the effective data dependences between processors. This ability is enforced by a possible send/receive unmatching useful for irregular algorithms. A sparse vector computation exemplifies our language potentialities.

Keywords: *Design of Parallel Programming Languages; Synchronous and Data-Parallel Programming Model; Asynchronous Execution Model; Structural Clock; Irregular Algorithm.*

1. Introduction

Nowadays, parallelism appears as an inescapable technique to reach real large computation abilities. Nevertheless, its use is still restricted to a community of specialists. One of the main barriers for a larger diffusion is the lack of softwares up to efficiently exploit the capabilities of parallel computers except at the expense of an important programming effort.

Numerous researchers have tackled the design of parallel programming languages. Two main programming models rise up. The former, data-parallelism, relies on parallelism expressed through a sequential composition of actions on parallel data. Action execution is synchronous, which greatly makes easier the control of program semantics. The difficulty is then transferred to the compiler. It has to efficiently run data-parallel programs on loosely synchronized MIMD architectures, while preserving the 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 processors [10, 16].

The latter programming model, task-parallelism, requires the programmer to explicitly control each processor in an asynchronous framework. He must handle synchronizations to ensure determinism while avoiding deadlocks. The most famous programming tools, PVM [6] or MPI [7], merely uses function libraries. Such a programming model is not really structured and so difficult to handle. Nevertheless, it is close to the execution model provided by most of present parallel computers thereby yielding efficient programs.

Other approaches propose bridging models. The Fortran M language [5] adds structure to task-parallelism by handling communication channels as variables. Program determinism becomes easier to control, but the programmer still have to manage multiple control flows ordered only by communication events. The BSP model [17, 12] eases programming by dividing programs in communication or computation stages ended by global synchronizations. Theses stages are called supersteps. Communications are remote memory operations completed when a superstep ends. Although this model is simple to program and avoid complex compiler optimizations, efficiency is limited since only global synchronizations are allowed.

In this paper, we propose a new parallel language, called $\mathcal{SCL}-\mathcal{C}han$. It relies on explicit parallelism and yields deterministic and deadlock free programs. It offers both a synchronous and data-parallel programming model and an asynchronous execution model. The synchronous programming model allows the programmer to design programs by handling a single control flow. Therefore, programming effort is reduced. The asynchronous execution model leads to loosely synchronized programs, thereby enabling efficient executions on present MIMD computers. Compiler task is then reduced since there is no need for extracting parallelism and asynchronism through complex data dependence analysis. $\mathcal{SCL}-\mathcal{C}han$ widely inherits its semantics from the $\mathcal{TML}$ and $\mathcal{SCL}$ languages introduced in previous papers [9, 8, 13, 14], but it deeply differs in the communication paradigm. Communications rely on send/receive operations using point to point channels. The central idea is a logical partial ordering of instances of program instructions. It allows us to keep the synchronous semantics of the programming model, while authorizing an asynchronous execution through point to point one-sided synchronizations. Synchronizations are handled by a blocking receive instruction. In contrast to classical approaches [7, 5], it does not block on empty channels, but it waits for the sending process to have performed all the instructions logically preceding the receive statement. To implement the logical ordering of program instructions, we reuse the structural clock mechanism previously introduced for the $\mathcal{SCL}$ language [8]. As it is not mandatory to match send/receive statements, the execution of an $\mathcal{SCL}-\mathcal{C}han$ program can be driven by effective data dependences. This feature is especially interesting for irregular problems such as sparse computation. The potentialities of $\mathcal{SCL}-\mathcal{C}han$ are exemplified through a simple example of sparse vector computation.

2. The synchronous programming model of $\mathcal{SCL}-\mathcal{C}han$

In this section we present the syntax and the informal semantics of the $\mathcal{SCL}-\mathcal{C}han$ synchronous programming model. For the sake of conciseness, we do not detail memory management.

2.1. The data-parallel programming model

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

2.2. The $\mathcal{SCL}-\mathcal{C}han$ programming model

The programming model is data-parallel and synchronous. We call *index* a virtual processor in reference to the data-parallel model. Each index u owns a *(private)* memory where it stores its own data (the local component of each parallel variable). Such a memory is used only for evaluation of *pure* expressions and for assignments. The data it stores cannot be directly referenced by another index.

Data exchange between indices takes 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 this datum is explicitly *sent* to v. The index v can record such a value in its private memory only if it explicitly *receives* it.

2.2.1 The instructions

Let us describe the $\mathcal{SCL}-\mathcal{C}han$ instructions.

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

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

Synchronous sequence : $S; T$. When all the active indices finish the execution of S, they start the execution of T.

Concurrent conditioning: where B do S elsewhere T end. This instruction splits the set of the active indices into two independent sub-sets of *active* indices that executes concurrently the instruction blocks S and T. Each active index u which evaluates the pure expression $B|_u$ to $true$ executes S, otherwise it executes T.

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

Loop: loopwhere B do S end. Iteration is expressed by classical loop unfolding. Only the active indices evaluating the *pure* expression B to $true$ execute S with the new extent of parallelism. The others are idle. Note that the set of active indices may decrease at each loop iteration. An execution of a loop terminates when all active indices evaluate B to $false$. This is the while instruction of MPL [4] or the whilesomewhere of HYPERC [11].

Send: send X to A. It is a non blocking send. The expression A must be *pure*. 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 $A|_u$.

Receive: receive X from A into Y. The semantics of this instruction relies on a logical partial ordering on the instances of program instructions. Hence, we first define this ordering.

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

Example.

- In 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 of 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 instruction, are not comparable.

Now, we can present the informal semantics of the receive X from A into Y instruction. The expression A must be *pure*. Each active index u evaluates the address of the sending index $v = A|_u$ in its private memory. Then, u considers all the instructions executed by v, of type send X to B, where $B|_v = u$ and coming before the instruction receive. Different cases have to be considered:

- if such instructions do not exist, the index u assigns `nil` to $Y|_u$;
- otherwise, it selects the *more recent* one (the greater with respect to the instruction ordering):
  - ▷ if the datum sent by this instruction has been already received by a previous receive instruction, the index u assigns `nil` to $Y|_u$;
  - ▷ otherwise, the datum sent by this instruction is received and assigned to $Y|_u$.

Remark. As two instructions, each one belonging to a different branch of the same where/elsewhere instruction, are not comparable, no communication can take place between these two branches. They can be executed concurrently without problem of data coherency.

2.3. Example

```
Y := 0;
X := 1;
send X to This + 1;
where This = 1 do
    X := 2;
    send X to This + 1;
elsewhere
    receive X from This + 1 into Y;
end;
```

```
Y := 0;
X := 1;
send X to This + 1;
where This = 1 do
    X := 2;
    send X to This + 1;
end;
where This = 2 do
    receive X from This + 1 into Y;
end;
receive X from This + 1 into Y;
```

The two above examples of $\mathcal{SCL} - \mathcal{C}han$ programs illustrate its semantics. A torus of indices numbered from 1 to 2 is considered. When the top program is executed, the index 2 receives the value 1 from the index 1 into its component of the variable Y. The value 2 sent through the second send instruction cannot be received since the receive and send instructions belong to different branches of the same where/elsewhere structure. If one wants to put these instructions in correspondence the where/elsewhere must be rewritten as shown in the bottom program. So, the second send instruction precedes the first receive. The index 2 receives the value 2 from the index 1. The second receive instruction assigns the value 1 to $Y|_1$ (the first send precedes it) and `nil` to $Y|_2$ (the last sent value has already been received).

3. An asynchronous execution model for $\mathcal{SCL} - \mathcal{C}han$

In the previous section we have presented a synchronous programming model for $\mathcal{SCL} - \mathcal{C}han$. In this section, we show that it is possible to propose an asynchronous execution model that yields loosely synchronized programs without changing the program semantics.

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

The underlying abstract machine is an asynchronous parallel one. Data in transit, i.e. data sent by an index u to an index v and not yet received, are stored in a specific memory called a *channel*. There exists one channel by index pair and by variable. We denote by (u, v, X) the channel storing the values $X|_u$ of the variable X sent by u to v. Each channel corresponds to a LIFO (last-in first-out) list with respect to the instruction ordering defined in section 2.2.1. At the beginning of a $\mathcal{SCL} - \mathcal{C}han$ program all channels are empty.

3.2. The instruction semantics in the asynchronous frame

The sole possible interaction between indices occurs through communications. The initial program semantics can be lost, through an asynchronous execution, if we are not able to ensure that the received values are identical to those received in the synchronous frame. The correctness of our asynchronous execution model is so linked to channel management. Therefore, we detail all the instructions interfering with channels. For the other ones, they merely become asynchronous, i.e. they are executed independently by each active index.

Send: send X to A. This instruction is asynchronous and non blocking. Each active index u sends the value $X|_u$ to the index $v = A|_u$, i.e. it adds this value in the channel (u, v, X).

Receive: receive X from A into Y. This instruction blocks. An index u that executes this instruction waits that the sending index $v = A|_u$ has executed all the send instructions coming before (in the sense of the ordering defined in section 2.2.1) this receive. This waiting ensures that the value which would have been received in the synchronous case is then stored in the channel. Next, the index u considers all the data stored in the channel (v, u, X) and sent by instances of send instructions coming before the receive. If such data do not exist, the index u assigns `nil` to $Y|_u$. Otherwise, it selects the *more recent* sent datum, assigned it to $Y|_u$ and removes the received datum and all the preceding ones from the channel (v, u, X).

Remark. $\mathcal{SCL} - \mathcal{C}han$ is deadlock free. The receive, the sole blocking instruction, relies on an instruction ordering. Therefore, among all the executed instruction instances, a minimal one cannot block an index executing it.

3.3. Structural Clocks

To be feasible, the asynchronous execution model requires that each index should know its position in the program structure according to the instance of the currently executed instruction. Therefore, an index that executes a send instruction can stamp the send 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 channel data stamps to complete the receive. In order to formalize index positions we reuse the *Structural Clocks* introduced in previous papers [9, 13, 8, 14]. Structural clocks record index positions with regard to the instances of instructions that store data in the 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. 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. 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 on the list 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 compare the local counters corresponding to the common instruction block of the innermost nested level. We so define this partial ordering as a lexicographical order based on partially ordered pairs.

Definition 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 consider only the meaningful instructions, this ordering formalizes the one defined in section 2.2.1. The index u is *before* v ($t_u \prec t_v$) if and only if the instruction instance executed by u is *before* the one executed by v.

3.4 Example of Structural Clock use

The simple following program encodes a pipe-line between two indices numbered 1 and 2. It exemplifies the way structural clocks are handled. 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 the where instruction a specific task is assigned to each index. The first index computes S from local data and sends it to the index 2. The index 2 receives it and adds it to $A[i]$. Each task is repeated 100 times.

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

Remarks:

- No read/write conflict is possible on structural clocks since they always increase. Their communication do not involve synchronizations.
- Depending on the considered program, it is often possible to reduce structural clock updates without endanger the semantics. Structural clocks are used to know the index position with regard to send instructions. Therefore, since the below program uses only one send, it is sufficient for the two indices to count each instance of send. The index 1 can only increase its structural clock after the send and the index 2 after the evaluation of the first where.
- $\mathcal{SCL} - \mathcal{C}han$ needs structural clocks to be communicated between indices. Nevertheless, depending on the program, it is not necessary to broadcasts all structural clocks to all indices each time they are modified. Below, each waiting step ends as soon as the index 2 receives the structural clock updated after the the second where. So, it is sufficient that the index 1 sends its structural clock to 2 after the evaluation of the second where of each iteration.

$i := 0$	$(0,0)$
loopwhere $i < 100$ do	$(0,0)(1,0)$
where $This = 1$ do	$(0,1)(1,2i)(1,0)$
$S := i * Z[i]$;	$(0,1)(1,2i)(1,0)$
send S to $This + 1$;	$(0,1)(1,2i)(1,1)$
end;	$(0,1)(1,2i+1)$
where $This = 2$ do	$(0,1)(1,2i+1)(1,0)$
receive S from $This - 1$ into R;	$(0,1)(1,2i+1)(1,0)$
$A[i] := A[i] + R$;	$(0,1)(1,2i+1)(1,0)$
end;	$(0,1)(1,2i+2)$
$i := i + 1$	$(0,1)(1,2i+2)$
end;	$(0,2)$

4. Example of send/receive unmatching

```
integer ::  A(0:12)
do j = 3,12
   A(j)=A(j)+A(j-2)*A(j-3)
enddo
```

```
J := 3;
   loopwhere J ≤ 12 do
      where This = J − 3 do
         send A to J;
      end;
      where This = J − 2 do
         send A to J;
      end;
      where This = J do
         receive A from This − 2 into Tmp2;
         receive A from This − 3 into Tmp3;
         A := A + Tmp2 ∗ Tmp3;
      end;
      J := J + 1;
   end;
```

One of the features of the $\mathcal{SCL}-\mathcal{C}han$ language is that exact send/receive matching is not mandatory. If there is no receive for a send, the sent data is destroyed. If a send lacks for a receive, a `nil` value is received. We will see that we can take advantage of such a semantics to reduce communications in irregular applications, where necessary communications are not foreseeable. We illustrate it through a simple example of a sequential loop parallelization handling a sparse vector.

Let us consider the above Fortran loop on the array A. Loop parallelization involves communications since we have data dependences between iterations. One way to perform it by using $\mathcal{SCL}-\mathcal{C}han$, leads to the $\mathcal{SCL}-\mathcal{C}han$ program that appears below the Fortran one. A loopwhere instruction is necessary to express the expected ordering between the communications and the computations on A. The synchronous programming model easily provides the communication ordering. For example, it is obvious that an instance of the instruction receive A from $This - 2$ into $Tmp2$ receives the datum sent by the preceding send of the same iteration. From the point of view of the asynchronous execution model, each index executes independently the program as soon as it does not encounter a receive instruction. If so, it must wait for the sending index until the structural clock of the sending index is no more later than its own, i.e. it waits as long as the sender executes a meaningful instruction coming before the considered receive. Then, it can continue its computation.

```
J := 3;
   loopwhere J ≤ 12 do
      where This = J − 3 AND A ≠ 0 do
         send A to J;
      end;
      where This = J − 2 AND A ≠ 0 do
         send A to J;
      end;
      where This = J do
         receive A from This − 2 into Tmp2;
         where Tmp2 ≠nil do
            receive A from This − 3 into Tmp3;
            where Tmp3 ≠nil do
               A := A + Tmp2 ∗ Tmp3;
            end;
         end;
      end;
      J := J + 1;
   end;
```

If the array A is sparse, i.e. if numerous values are null, we would not like to send useless data. Since send/receive matching is not necessary with $\mathcal{SCL}-\mathcal{C}han$, we can modify our program such as a datum is sent only if it is not null. Moreover, since the computation of A relies on the product of two distant data, it is possible to decide not to receive an operand of the mulptiplication (for example $Tmp3$), if the first received operand ($Tmp2$) is null. Therefore, the execution of a useless receive instruction can be avoided. To integrate these optimizations, we just add the underlined parts in our initial program (see the above program). At reception, if no data was sent, the concerned channel contains no datum stamped by a structural clock later than the one of the receiving index. The `nil` value is then returned. These optimizations allow to improve performances by reducing network load and by suppressing the synchronization time linked to the receive instructions not performed.

If one wants to implement such an algorithm with a classical message passing library like MPI [7] or with Fortran M [5], one must ensure exact send/receive matching to avoid deadlocks. Therefore, it is a difficult task to handle array sparsity in this framework.

5. Problem of Performances

We have implemented a simple library on Cray T3D to test the way the overhead assoiciated to structural clock machinery penalizes performances [15]. These first experimentations also highlight the performance improvement provided by unmatching send/receive instructions.

Our library provides all the functions necessary to handle structural clocks and channels. It mainly relies on the `put` communication instruction of Cray SHMEM library [3]. The `put` allows one processor to write directly into another processor's memory without that processor being notified. This instruction leads to a low latency time close to $1\mu s$. In order to further computation/communication overlap, we chose to locate the channels in the memory of the receving index. So, when an index executes a send, it executes a `put` on the memory of the receiving one. An index also sends it structural clock to other ones by `put` instructions. Note that the waiting executed during a receive is used to ensure that the data that were sent before the receive are stored in the channel. Therefore, when an index receives a structural clock of a sending index, all the data sent before must already be received. This ordering between data and structural clock communications is ensured by the T3D communication protocol.

Tests were performed with a maximum of 64 processors. We compared our performance results with one a provided by equivalent MPI programs. We first programmed a simple ping-pong test. Two indices exchange an integer numerous times. It appears that $\mathcal{SCL} - \mathcal{C}han$ is 2.5 times as speed as MPI. This gap is due to the lighter communication required by $\mathcal{SCL} - \mathcal{C}han$.

More complex programs were also tested. They show that $\mathcal{SCL} - \mathcal{C}han$ generally overcomes MPI. Nevertheless, these good results may depend on an optimization of the communication pattern of structural clocks. The idea of such otpimizations is that an index needs the structural clock of an other index, only if the former can receive data from the latter. Moreover, an index does not need to send a structural clock after each update if there is no receive instruction between the positions of these different updates. With these optimizations, $\mathcal{SCL} - \mathcal{C}han$ always yields better performances than MPI.

Tests with sparse computations showed that to save send or receive instructions, by using the possibility of $\mathcal{SCL} - \mathcal{C}han$ to handle not foreseeable communications, can significantly improve performances. The main source of gains comes from waits during the receive execution. When receive instructions are saved, the wait is suppressed. If the send instructions correspond to long messages, then the wait is shorten in an important way. Since the structural clock send is executed after the send, not to perform the send allows to receive the structural clock sooner. For example, we considered a computation with 90% of null values, that enables to save send and receive instructions. Run with 16 processors, $\mathcal{SCL} - \mathcal{C}han$ improves MPI time by a ratio of 13.

6. Why LIFO lists and not FIFO ones?

Classically, the send/receive paradigm used in control flow languages relies on FIFO (first-in first-out) list [5, 7], i.e. the first sent data is received. The receive instruction is blocking until a data is available. $\mathcal{SCL} - \mathcal{C}han$ semantics is deeply different since the waiting step of the receive instruction depends on the sender position and not on a reception event. In this framework, communications can be based either on LIFO channels or on FIFO ones with the same expressiveness. We chose LIFO lists for three main reasons.

- **Structured communications.** Communications through LIFO lists are more structured since a receive always refers to the last preceding send or receive that uses the same channel. It is simple to recognize which send has provided a received data or if `nil` is returned. In opposite, FIFO based communications require the programmer to remember an important part of the sending index past, in order to be able to identify the received data. The introduction of routines that hide communications even makes more complicated this task.
- **A deadlock free send.** The presented communication model relies on potentially infinite lists, but a realistic implementation requires limited list sizes. In this case, when a list is full, the sending index must wait for a data to be removed from the channel. With LIFO lists, if the sender u is before the receiver v, there is at most two valid data in the channel (u, v, X):
  - ▷ the last data sent by a send coming before the receive;
  - ▷ the last data sent by a send not comparable with the receive.

  Therefore, it is possible to built a deadlock free protocol as soon as channel sizes are strictly greater than 2. If FIFO lists are considered, a deadlock is always possible since a position in a channel can be freed only if a data is received.
- **Not foreseeable communications.** To receive the last sent value allows to forget how many send were performed before. For some applications where exact matching of send and receive is costly, like synchronous Boltzmann Machines [1], this fact eases communication management and might lead to optimizations. At each step, a random set of indices (neurons) update their states from neighbor states. Hence, a neuron that updates its state only needs the last state of its neighbors. The preceding sent states are useless. With FIFO channels, the sole way to suppress a data from a channel is to receive it. Therefore, if the same

channel is used more than once, it is mandatory to receive all the sent data not to introduce confusion in data coherency.

7. Conclusion

In this paper, we have presented the $\mathcal{SCL} - \mathcal{C}han$ language. It provides, in a structured framework, both a synchronous data-parallel programming model and an asynchronous execution one. $\mathcal{SCL} - \mathcal{C}han$ owns the expected characteristics for a parallel programming language. It represents a tradeoff between programming facility and execution efficiency. It is portable, deterministic and deadlock free. From a synchronous data parallel program, $\mathcal{SCL} - \mathcal{C}han$ yields a loosely synchronized one without any data dependence analysis and program transformation. In opposite, classical data-parallel languages need such interventions that often lead to over-synchronized programs [10, 16]. Even the bridging model BSP [12] or the languages $\mathcal{TML}$ and $\mathcal{SCL}$ [8] leads to useless synchronizations.

$\mathcal{SCL} - \mathcal{C}han$ main originality relies on a specific communication/synchronization model that depends on a logical instruction ordering formalized through Structural clocks. The instruction ordering leads to an asynchronous execution model. It uses only one-sided and point to point synchronizations. The termination of a synchronization depends on the position of the waited index in the program structure and not on a reception event, like for the classical receive instructions encountered in message passing libraries [7] or in Fortran M [5]. Therefore, the language authorizes send/receive unmatching. It allows to express only effective data dependences between indices. The presented sparse computation shows how to express not foreseeable communications. $\mathcal{SCL} - \mathcal{C}han$ sematics allows to manage them at run time wihtout any problem of data coherency.

We are currently working on a formal semantics for $\mathcal{SCL} - \mathcal{C}han$.

References

[1] R. Azencott. Boltzmann machines: high-order interactions and synchronous learning. In A. F. P. Barone, editor, *Stochastic Models in Image Analysis*, Lecture Notes in Statistics. Springer-Verlag, 1992.

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

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

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

[5] I. Foster. *Designing and Buiding Parallel Programs*. Addison–Whesley, 1994.

[6] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 User's Guide and Reference Manual.* Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1994.

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

[8] 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.

[9] 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).*

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

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

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

[13] 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`. This Research Report is a full version of [8].

[14] E. Melin, B. Raffin, X. Rebeuf, and B. Virot. A General but Simple Technique to Handle Asynchronous Data-Parallel Control Structures. In IEEE, editor, *5th EUROMICRO Workshop on Parallel and Distributed Processing*, London, United Kingdom, January 1997. To appear.

[15] E. Melin, B. Raffin, X. Rebeuf, and B. Virot. A Unifying Approach of Data and Task Parallelism Based on Structural Clock Communication Paradigm. Technical Report RR97-01, LIFO, Orléans, France, February 1997. `http://web.univ-orleans.fr/~virot`.

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

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