Implementation on Cray T3D/T3E of $\mathcal{SCL} - \mathcal{C}han$, a Programming Language Unifying Data and Task Parallelism *

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
Tel: (+33) 02 38 41 73 18 - Email:Bernard.Virot@lifo.univ-orleans.fr

September 23, 1997

Abstract

Among parallel programming tools, task parallelism and data parallelism are the most common programming models. Task parallelism enables to reach high performances but requires an important programming effort. In opposite, data parallelism provides programming facilities close to the sequential model at the expense of performances.

In this paper, we show that it is possible to propose a language that integrates a task parallel execution model and a data parallel programming model in a unique and unifying framework. This ability depends on a specific synchronization/communication model relying on structural clock mechanism. We describe an implementation of our model on Cray T3D and Cray T3E. It highlights experimental performances that outperform classical MPI programming.

Keywords: Design of Parallel Programming Languages; Task Parallel Language; Data Parallel Programming Model; Structural Clocks; Implementation Results on Cray T3D/T3E.

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 capabilities of parallel computers except at the expense of an important programming effort.

Among parallel programming tools, two main programming models rise up. The former, called data parallelism, relies on parallelism expressed through a sequential composition of actions on parallel objects. Action execution is synchronous. Therefore, programs are deterministic (all possible executions of a program yield the same final result) and deadlock free (any program execution either terminates or loops endless). These properties are essential to control program correction. Furthermore, synchronism makes easier program design by enabling a sequential program reading. The difficulty is then transferred to the compiler. It has to efficiently run data parallel programs on present loosely synchronized MIMD architectures, while preserving the semantics of the synchronous and centralized programming model. So, data parallel compilers have to realize complex optimizations up to manage data distribution and time desynchronizations between processors [9, 13].

The latter, called task parallelism, relies on an explicit distribution of data and computation tasks on asynchronous processors with physically distributed memories. Classically, it uses a message passing communication model. Data communications between processors take place through explicit send/receive instructions [14, 5]. This communication model mixes up communications and synchronizations since a receiving processor is blocked as long as no data is received. So, send/receive instructions must match to

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

avoid deadlocks. Instruction executions are ordered by communications which generally forbid a sequential program reading. Task parallelism is so difficult to handle. The programmer is moreover responsible for handling non-determinism and for guaranteeing the absence of deadlock. Nevertheless, task parallelism is expressive and close to the execution model provided by most of present parallel computers thereby yielding efficient executions.

Other approaches propose bridging models. Fortran M [5] relies on a task parallel model, restricted to essential communication mechanisms, like single reader/single writer channels or single-assignment variables, which enforce program determinism. It also allows to integrate calls to data parallel programs such as HPF ones. Therefore the programmer can express task parallelism to help its data parallel compiler to reduce synchronizations without endanger program determinism. Nevertheless, the programmer is still confronted with deadlock problems and the programmer still have to manage multiple control flows ordered only by data communications. Sequential program reading is generally forbidden thereby making programming harder.

The BSP model [15, 10] proposes a coarse grain sequential ordering of control flows. A program is divided in a sequence of supersteps, each followed by a global synchronization. A superstep consists in parallel computation with remote memory accesses which take effect at the global synchronization point. 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. Moreover BSP offers a simple model to analyze performances of parallel applications. Nevertheless, expressiveness is limited since the model provides a unique type of instruction ordering due to the mandatory use of global synchronizations. This type of control flow ordering has also been studied in an irregular framework by Diderich and Gengler [4].

In this paper, we show that it is possible to provide a unifying framework consisting of a synchronous programming model, thereby enabling sequential program reading, and an asynchronous execution model yielding efficient executions. The central idea is a logical partial ordering of instances of program instructions. Our approach is based on a parallel programming language, called $\mathcal{SCL}-\mathcal{C}han$, introduced in [12]. It is a SPMD and task parallel language. Communications take place through channels and rely on explicit send/receive instructions. In opposite to classical message passing, synchronizations and communications are dissociated. The receive instruction does not block on empty channels, but it waits that the sending processor has performed all the instructions logically preceding the receive statement. So, communications do not influence the ordering of instruction execution, but depend on the ordering imposed by program structure. This logical ordering is implemented by using structural clocks [8, 7, 11]. Each processor owns a structural clock that encodes its logical position with regard to send instructions. Such model authorizes send/receive unmatching. Thereby, communications can be reduced when they are not foreseeable as in some sparse computations.

We briefly recall 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 introduce a synchronous programming model. We give the idea of the proof showing that the execution corresponding to this synchronous model is equivalent to any other possible asynchronous execution with respect to the returned result.Last, we present performance results. We have implemented the $\mathcal{SCL}-\mathcal{C}han$ language on a Cray T3D and T3E. Algorithms with various data dependences and different communication requirements, such as Gauss-Seidel relaxation and Cholesky factorization, were tested. It appears that $\mathcal{SCL}-\mathcal{C}han$ generally overcomes classical MPI programming. We also show that send/receive unmatching can significantly improve performances for sparse computations.

2 The language

2.1 An asynchronous execution model

2.1.1 $\mathcal{SCL}-\mathcal{C}han$ abstract machine

The abstract machine is an asynchronous parallel one with Pe_Nb processors. Processors are also 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 $X|_u$. These data cannot be directly referenced by another index. We also define the special constant $This$ that provides the index identifiers: $This|_u = u$.

Data exchanges between indices take place through explicit send/receive communications. Therefore, a data of the private memory of an index u becomes available for a distant index v only if this data 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.

2.1.2 Language instructions

The asynchronous execution model of $\mathcal{SCL} - \mathcal{C}han$ relies on 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 the receive instruction 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.

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 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 $true$.

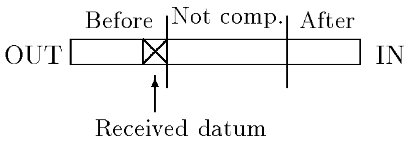
Send: send X to A. This instruction is non blocking. The expression A must be pure. An active index u sends the value $X|_u$ to the index $v = A|_u$, i.e. it adds this value to the channel (u, v, X).

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

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 active index 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***.*

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 in 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 is a blocking instruction. Its execution depends on a *waiting condition*: an active index u waits that the sending index $v = A|_u$ has executed all the send instructions coming before this receive (in the sense of the ordering previously defined). When this condition is satisfied, the channel (v, u, X) is considered. The data of the channel can be divided into three sets BE, NC and AF. 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 the data sent by send coming after (see the above figure). If the set BE is empty, the index u assigns `nil` to $Y|_u$. Otherwise, it first selects the most recent datum of BE, next it assigns it to $Y|_u$ and last it removes all the data of BE. Therefore, channels are managed as LIFO lists with respect to the instruction 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. A datum sent from one branch to the other can only be received by a receive executed after exiting the where/elsewhere. It can be seen as if these send have been performed just at the end of the where/elsewhere construct.
- Exact matching between send and receive instructions is not mandatory. More send than receive can be executed on a same channel or vice versa. This feature is important as it allows the programmer to limit communications when they are not foreseeable [12]. Classical message passing languages and libraries [5, 6] do not offer such a possibility.

2.2 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 instruction currently executed. 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 the stamps of data in channels to complete the receive. In order to encode index positions, we reuse the *Structural Clocks* introduced in previous papers [8, 7, 11, 12]. 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. 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 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 consider only meaningful instructions, this ordering formalizes the one defined in section 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.

2.3 Example

forwhere $i := 0$ to $i < 100$ 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;	$(0,1)(1,2i+2)$
end;	$(0,2)$

y This simple 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 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.

2.4 Fundamental properties of the asynchronous model

In this section we sketch the proofs of two fundamental properties of the asynchronous execution model of $\mathcal{SCL} - \mathcal{C}han$, i.e. deadlock freedom and computation determinism. These properties ensure that either all possible asynchronous computations finish returning the same result, or they all loop endless. They are fundamental on two points. The detection of deadlock and non determinism, generally complex and time consuming tasks, are avoided. Moreover, they are necessary conditions to ensure the correctness of the synchronous programming model we shall see in the next section.

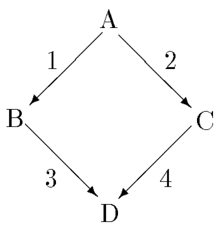
2.4.1 Deadlock freedom

At first, we proof that the execution model is deadlock free, i.e. that any program execution either terminates or loops endless.

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

The intuitive idea is that a deadlock configuration is impossible since indices with minimal structural clocks cannot be blocked. If such an index is blocked, then the waiting condition is not satisfied, i.e. the waited index owns a lower structural clock than its. Note that there always exists minimal structural clocks since a finite number of indices is considered.

2.4.2 Determinism of computations



$\mathcal{SCL}-\mathcal{C}han$ is deterministic, i.e. all possible executions of a program always yield to 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 the adjoining figure) can be executed concurrently from a state A and they yield different states B and C, then their exists two instructions 3 et 4 that can be executed from B et C and that lead to the same state D.

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

Let us give the main ideas of the proof. Instructions can be executed concurrently only if they are executed 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 concurrently executed 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 the written structural clock by a greater one. The receiving index reads it to check if it is not lower than its local structural clock. Since we assume that the receive can be executed, the wait condition is already satisfied from the initial state. Therefore, the execution of the meaningful instruction, before or after the receive execution, does not affect the receive execution.

- Conflicts could appear when send/receive instructions access 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 in section 2.1). Hence, since the receiving index only reads and erases data of BE no conflict occurs.

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 relies 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, i.e. sequential 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 the 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 `nil` value.

Let us 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 sub-sets of indices that execute sequentially the instruction blocks S and T. The active indices that evaluate the pure expression B to $true$ execute S while the other indices are idle. Next, T is executed by the active indices that evaluate the pure expression B to $false$ while the other indices are idle. Channel management must lead to independent branches as in the asynchronous framework. Therefore, we temporarily hide the send instructions that could be executed from one branch of the where/elsewhere instruction to the other. When two indices u and v execute a different branch of the same where/elsewhere construct, they no more communicate through the common channels (u, v, X), but each index uses its own channels denoted by $(u, v, X)_u$ and $(u, v, X)_v$. These channels are intialized with the value `nil` and with the data stored in (u, v, X) respectively. When the conditioning ends, if a channel $(u, v, X)_u$ is different of `nil` it is copied in the channel (u, v, X), otherwise $(u, v, X)_v$ is copied in (u, v, X). Next, the channels $(u, v, X)_u$ and $(u, v, X)_v$ are destroyed.

3.2 Fundamental equivalence

We show that the synchronous execution of a $\mathcal{SCL}-\mathcal{C}han$ program provides the same result as any possible asynchronous execution of the execution model. The proof relies on an equivalence on the computations of the two models. It ensures the coherency of the programming model with regard to the execution model.

The following theorem states that a synchronous computation is no more than a particular computation of the asynchronous execution model. It requires to define an equivalence between computations. Two computations are equivalent if starting with identical private memories, they both end with the same ones or they both loop endless.

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

The proof relies on an induction on the program structure. The intuitive idea is the following. At first, ones shows that for each instruction execution in the synchronous model one can build a corresponding asynchronous computation. This computation consists in the execution of the considered instruction by the active processors and no action for the others. Conversely, one shows that if for an asynchronous computation each index initially owns the same program, the same structural clock, then, if the computation ends, we can exhibit an asynchronous computation (starting from the same state and so ending with the same state since $\mathcal{SCL}-\mathcal{C}han$ is deadlock free and determinsitic - see theorems 2,1) that can be decomposed in steps corresponding to the synchronous computation.

4 Implementation on Cray T3D

Let us now complement our theoretical results by performances results. They are obtained from a implementation of the $\mathcal{SCL}-\mathcal{C}han$ asynchronous abstract machine on Cray T3D.

Our main goal was to test the way the overhead associated to structural clock machinery penalizes performances. 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 [2].

4.1 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. $\mathcal{SCL}-\mathcal{C}han$ library allows to write C programs equipped with functions to increment structural clocks and to send them to other processors. Send and receive functions manage

```
real ::  A(0:99,0:99)
do i = 1,98
  do j=1,98
    A(i,j)=(A(i,j-1)+A(i+1,j)
      +A(i,j+1)+A(i-1,j))/4
  enddo
enddo
```

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

```
real ::  A(0:99,0:99)
do j = 0,99
  do k=0,j-1
    do i=j,99
      A(i,j)=A(i,j)-A(i,k)*A(j,k)
    enddo
  enddo
  A(i,i)=sqrt(A(i,i))
  do k=j+1,99
    A(k,j)=A(k,j)/A(j,j)
  enddo
enddo
```

```
forwhere J := 0 to J < N do
 J_p := J div Pe_Nb;
 where This < J AND This <> J mod Pe_Nb do
  forwhere D := 0 to D < N - j do S[D] := 0 end;
  forwhere K := 0 to This + Pe_Nb * K < J do
   forwhere D = J to D < N do
    S[D - J] := S[D - J] + A[D,K] * A[J,K];
   end;
  end;
  send S[0 : N - J - 1] to J mod Pe_Nb;
 end;
 where This = J mod Pe_Nb do
  forwhere K := 0 to This + Pe_Nb * K < J do
   forwhere D := J to D < N do
    A[D,J_p] = A[D,J_p] - A[D,K] * A[J,K];
   end;
  end;
  forwhere K := 0 to K < Pe_Nb AND K < J do
   where This <> K do
    receive S[0 : N - J - 1] from K into R;
    forwhere D := J to D < N do
     A[D,J_p] := A[D,J_p] - R[D - J];
    end;
   end;
  end;
 end;
 where This = J mod Pe_Nb do
  A[J,J_p] := sqrt(A[J,J_p]);
  forwhere D := J + 1 to D < N do
   A[D,J_p] := A[D,J_p]/A[J,J_p];
  end;
 end;
end;
```

Figure 1: On the left hand side are displayed programs that compute the Gauss-Seidel relaxation. The right hand side is concerned with the Cholesky factorization. Top programs are written in Fortran. Bottom ones are $\mathcal{SCL} - \mathcal{C}han$ programs. Note that the underlined parts of the bottom left program correspond to additional code introduced to avoid useless communications of null data.

data communications according to $\mathcal{SCL} - \mathcal{C}han$ paradigms. This library mainly uses 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. 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 be received after it also. All the data sent before the clock which has validated the waiting condition must be stored in the channel. Otherwise, it can lead to an error since data can lack in the BE set. Therefore, communication ordering between data and structural clocks must be respected. It is ensured by Cray T3D routing protocol.

In section 2.1, 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 fit our communication protocol to manage such channels. The send becomes a potentially blocking instruction. Each index counts the number of send instructions performed on a channel. A send is blocked as long as the target channel is full, i.e. as long as the sender has not freed a position. During receptions the receiver sends acknowledgments to the sender according to the freed positions. Moreover, as long as it is blocked (because of a send or a receive), each index flushes its channels of useless data. An index that has finished program execution sends a special acknowledgment not to block the indices that could send it a datum. This protocol is deadlock free if channel sizes are strictly greater than two.

4.2 Experimental results

In the following, a vector of Pe_Nb indices numbered from 0 to $Pe\_Nb - 1$ is considered. For sake of convenience, we use a forwhere $I := L$ to B do S end instruction defined as $I := L$; loopwhere B do S; $L := L + 1$; end.

4.2.1 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. Our channel structure needs a simple communication protocol, lighter than MPI one.

4.2.2 Gauss-Seidel relaxation

Let us now consider more complex programs where structural clocks have a more important impact. The Gauss-Seidel relaxation algorithm consists in going through a two dimensional array from right to left and top to bottom, while computing the average of the values of the four neighbors (Fig. 1, the top left FORTRAN program). It is an interesting problem to parallelize since we have numerous dependences and anti-dependences.

We programmed this algorithm using $\mathcal{SCL} - \mathcal{C}han$ (Fig. 1, bottom left program without the underlined parts) by distributing the array A by blocks of columns among the Pe_Nb indices. Each index owns an array A of $N \times L$ components. Computations are performed line by line. For a line I, an index u begins by sending $A[I, 0]$ to its left neighbor $u - 1$. Next, it receives $A[I, N - 1]$ from $u - 1$, it computes the average for all the line elements but the last one. It receives $A[I, 0]$ from $u + 1$. So, it can compute the new value of $A[I, N - 1]$ and send it to $u + 1$. Such a communication pattern requires only one channel between neighbor indices. Note that the forwhere $J := 1$...do loop is required to match send/receive instructions in the expected ordering. Each index is active only during one iteration. The others are simply necessary to properly increment structural clocks. Note that the synchronous programming model provides a clear ordering of computation tasks, enabling to easily control the program correction.

Structural clock management: Let us now detail structural clock management. We defined the control structures as meaningful instructions since the execution of a send can depend on the enclosing control structures (see section 2.2). Therefore, in our program, the control structures that do not enclose a send are not taken into account for structural clock update. We developed several optimization levels for structural clock communication. The idea is that structural clocks are useful only for receiving indices. First of all, we do not optimized, i.e. a structural clock is sent to all the other indices each time it is modified but after a send. When a send is performed, the sole index that is interesting in the structural clock of the sender

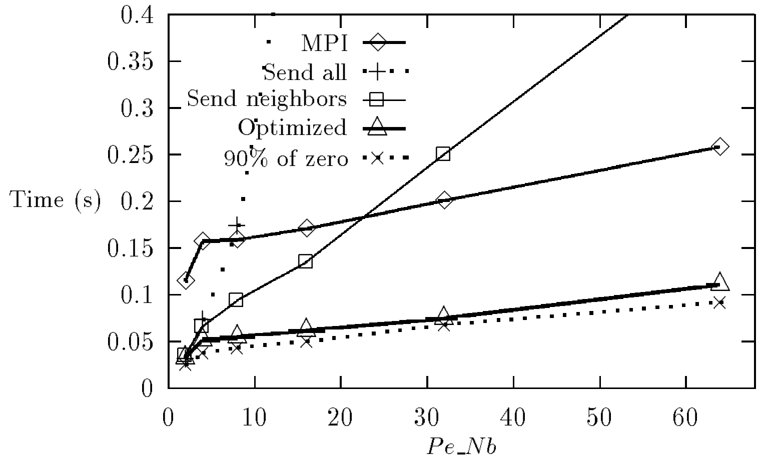


Figure 2: Performance results for Gauss-Seidel algorithm. For all tests, each index owns a matrix of size 1000×5.

is the receiver (for the wait condition). Therefore, the new structural clock computed after the execution of a send is sent only to the receiver. This version is called "send all". Since data communications take place only between neighbor indices, a simple optimization consists in sending structural clocks to neighbors only (the "send neighbors" version). The most optimized version sends structural clocks only one time between a send and a receive, after the last structural clock update (the "optimized" version).

Sparse version: $\mathcal{SCL}-\mathcal{C}han$ authorizes send/receive unmatching. Therefore, if our array contains numerous zero, it is possible only to send the non null values. At reception, if there is no matching send, it ensures that the data that would have been sent is null. The receiver easily detects it since `nil` is received. The bottom left program with the underlined parts manages such a situation (Fig. 1). Structural clock management is identical to the non sparse version. Note that even when a send is not executed, the sending index must count it and send its new structural clock to the receiver not to block it.

MPI program: For the equivalent MPI program, we reused the communication pattern of the non sparse version. It was just necessary to replace send/receive instructions by `MPI_Isend/MPI_Recv` ones and to remove all function calls that manage structural clocks. Moreover, since MPI communications rely on FIFO channels, the program has been organized differently such that less computations are performed by each index. The forwhere $J := 1...$do loop was suppressed and the forwhere $I := 1...$do loop was distributed among the index tasks (delimited by `if ...This...` conditionals). Note that since send/receive matching is mandatory, it is difficult to take advantage of null values to reduce communications like with $\mathcal{SCL}-\mathcal{C}han$.

Results: We performed tests with various index numbers. Each index always owns the same data number ($N = 1000$, $L = 5$). It enables us to see the variations of the communication overhead according to index number only. The value L is low not to mask communication cost behind computations. It appears that with the two first optimization levels, "send all" and "send neighbors", structural clock overhead rapidly becomes crippling (Fig. 2). However, the "optimized" version yields computations three times as speed as MPI ones. We also tested an "optimized" version with 90% of null values. The gap performances is not really relevant. The algorithm enables to save send of short data (integers) only. Not to execute a send enables to receive the structural clock sent afterwards sooner. Nevetheless saved messages are too short to improve significantly performances.

Gauss-Seidel like algorithm : To show the interest of unmatching communications, we slightly modified the Gauss-Seidel algorithm. Instead of computing the average of the four neighbor states, we compute the product of the four neighbor states and we modify the state of the considered component only if this result is not null. when an index computes a new state for $A[I, 0]$ or $A[I, N-1]$, it always stores locally three of the neighbor states. Therefore, as soon as one of this data is null, it knows that it is useless to receive the distant one. Hence, an index executes a send only if the data to send is not null and it executes a

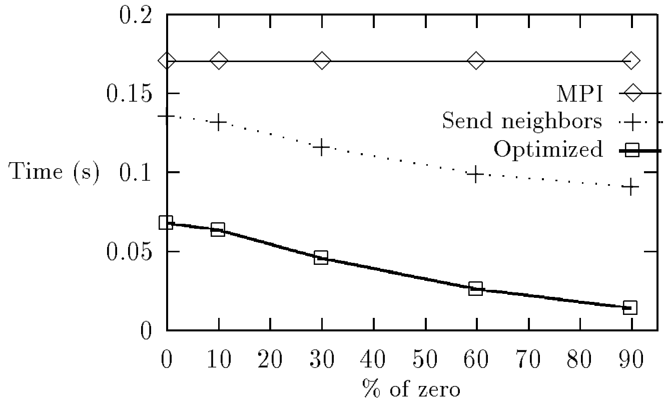


Figure 3: Performance results for Gauss-Seidel like algorithm. For all tests, each index owns a matrix of size 1000×5. 16 indices are engaged.

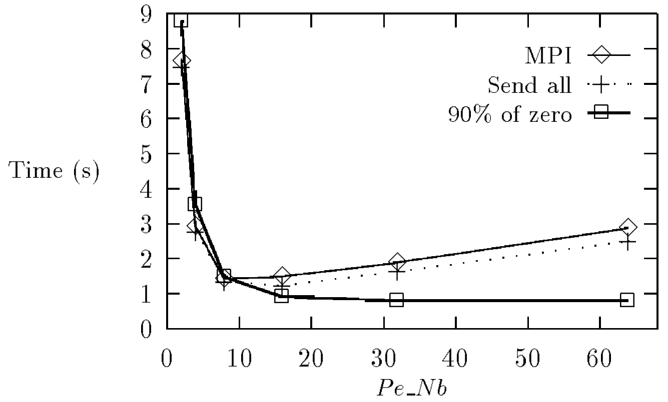


Figure 4: Performance results for the Cholesky factorization algorithm. A matrix of size 640×640 is cyclically distributed by columns among the indices.

receive only if the three neighbor states are not null. For sake of conciseness, the corresponding $\mathcal{SCL} - \mathcal{C}han$ program is not inserted in the paper. We present tests performed with 16 indices, $N = 1000$ and $L = 5$ (Fig. 3). We ran a "send neighbor" version and an "optimized" with different ratio of zero values (between 0% and 90%). The performance improvement is now relevant since we avoid useless receive instructions, i.e. costly waiting steps. Obviously MPI yields a constant computation due to send receive matching.

4.3 Cholesky factorization

In this section, the Cholesky matrix factorization is considered. This algorithm allows to test $\mathcal{SCL} - \mathcal{C}han$ with sending of data that can reach a full column of matrix. It works on a lower triangular matrix (Fig. 1, the top right FORTRAN program). Columns are computed one after the other from the left to the right. Each element of a column j is computed from its current value and from products of elements taken in the left hand columns. Next, the column j is normalized.

This algorithm is parallelized with $\mathcal{SCL} - \mathcal{C}han$ (Fig. 1, bottom right program) by distributing cyclically the columns of the matrix A among the Pe_Nb indices. Therefore, each index owns an array A of $N \times (N/Pe\_Nb)$ components. Computations are performed column by column. Let us consider the computation of a column J. The index $u = J \ mod \ Pe\_Nb$ owns this column. All the other indices v that own columns required for the computation ($u < J$ and $u \neq J \ mod \ Pe\_Nb$) perform a partial computation. They computes a vector S of $N-J$ elements from all the columns they store and that are involved in the computation of J. When the computation of S is completed, they send it to u. Before to receive these data the index u begins the computation with its local data. Next, it receives S for each index v and subtract it from the column J.

In this program, only two meaningful instructions are necessary to count: the send instruction and the enclosing where structure. Since the where contains exactly one send, it is useless to hide the number of meaningful instruction executed inside the where. Therefore, the indices that enter the where increase their structural clocks after the send execution, whereas the others increase their structural clocks when they evaluate the condition to $false$. The "send all" version corresponds to the send of the structural clock by the sender to the receiver after the send, and to a send to all the other indices after the other structural clock increase. The "optimized" version simply consists in removing the second structural clock send. It is useless since this structural clock update does not take place between a send and a receive.

We also fitted our program for sparse computation. When an index computes the vector S, it multiplies each element of the treated columns by the data $A[J,K]$. If this value is null for each column involved in the computation of S, the vector S is null. Therefore, the sparse version simply consists in sending the vector S if at least one data $A[I,K]$ is not null. The communication pattern of the non sparse version is kept for structural clocks. For sake of conciseness, this program is not displayed.

For the equivalent MPI program, it was just necessary to replace $\mathcal{SCL}-\mathcal{C}han$ communication instructions by MPI ones, and to remove all function calls that manage structural clocks. A simple code reorganization, as for the Gauss-Seidel algorithm, is impossible due to the data distribution.

We performed tests with various index numbers. We took a matrix of size 640×640. So, data number own by each index decreases while the number of indices increases. It enables us to see the parallelization efficiency with regard to the index number (Fig. 4). Performances for "send all" and "optimized" versions are very closed. We so displayed results of one version only . $\mathcal{SCL}-\mathcal{C}han$ performances are slightly better than MPI ones. The difference is not as significant as for Gauss-Seidel algorithm. First at all, communication cost is less important than computation one (for $Pe\_Nb \leq 16$). For more indices the communication cost increases but it is mainly due to transfer time (a machine dependent parameter), and not to communication protocol.

We tested the sparse version with 90% of zero. Because the number of columns decreases while Pe_Nb increases, the probability to perform a send increases with Pe_Nb. So, when Pe_Nb is low, additional code to control sends penalizes performances. Next, we save more and more send instructions which improves results. The saved communications allow to reach a speed up of 3.5 with regard to MPI. Note that for this program, communications correspond to long messages. Therefore, to avoid useless send instructions improves significantly performances.

5 Implantation on Cray T3E

We ported the $\mathcal{SCL}-\mathcal{C}han$ library on Cray T3E. It leads us to test two different versions.

5.1 Computer and Shmem library features

Cray T3E is a computer very close to T3D one. It is approximatively three time as speed as T3D. An important difference is that routing protocol no more guarantees a reception order identical to the sending one. This distinction appears through the Shmem semantics. Two instructions **shmem_put** executed by an index u toward an index v can write in the memory of v in the reversed ordering of their execution. To avoid the non determinism that can lead such an inversion, Shmem has a instruction **shmem_fence** that comes in between **shmem_put** instructions. It ensures that **shmem_put** instructions executed before **shmem_fence** write into the memory of the target index before the one located after.

5.2 Porting the library using shmem_fence

Porting the library using **shmem_fence** was trivial. We obtain an average time of $12\mu s$ for the ping-pong test. Communication is therefore twice as speed as on T3D. For comparison, MPI takes less advantage of the T3E power since it needs $27.5\mu s$ for the same computation. Its performances are only multiplied by 1.18 with regard to the ones on T3D.

5.3 Porting the library without `shmem_fence`

We developed an other version of the library using its own mechanism to control message ordering without `shmem_fence` instruction. With regard to the initial protocol of the library built for the T3D, not to keep message ordering leads to three problems:

- **Data send with regard to clock send.** When an index u validate a waiting condition with a clock t_v of an index v, then all the data sent before t_v send must be stored in the memory of u to have a complete set BE.

- **Acknowledgment send.** Assume that the index v sends an acknowledgment pointing out that the channel (u, v, X) is freed until 5^{th} position, and that next it sends a second acknowledgment telling u that all the positions are freed until the 10^{th}. If message ordering is reversed, the value 5 can erase the data 10 before the index u can read it. Therefore the index u can be blocked when trying to send a data because it does not know that the positions between 6 and and 10 are available.

- **Structural clock send.** To reverse the ordering between two distant writes of clocks erases a clock by a lower one. This situation can lead to a deadlock if the erased clock was necessary to validate a waiting condition.

To solve these problems, we use the instruction *shmem_get* and message numbering. The instruction *shmem_get* allows one processor to read directly into another processor's memory without that processor being notified.

- **Data send with regard to clock send.** The index u numbers each data sent in the channel (u, v, X). Each time, it send a data or a clock to v, it also send him the number associated to the last data sent in channels (u, v, X).When an index v executes a `receive` on the channel (u, v, X), after the waiting condition being validated by a clock t_u, it runs through the channel from the last inspected position. During the course, the index checks that messages have consecutive numbers. A break has two possibles causes:

  - ▷ The index v has identified the set BE. No other data was sent or they are not yet received. We check that BE is complete by comparing the last message number that must be equal to the number associated to t_u send.
  - ▷ The set BE is not yet identified. The lacking message(s), which were nevertheless sent before t_u, have not yet reach their destination. The index v wait them they arrive before to continue.

  When the BE set is identified, the index u selects the datum to receive and to deletes the data of the BE set.

- **Acknowledgment send.** If an index u is blocked during a `send` beyond a time limit t, then We consider that the acknowledgment necessary to authorize the send was deleted by a message inversion. The index u directly read it by using an instruction *shmem_get* on the memory of the receiving index. The operation is repeated while the index u is blocked.

- **Structural clock send.** As in the previous case, after a given time limit, an index that has not yet validated a waiting condition read the clock of the sending index by a *shmem_get*.

5.3.1 Performances

With this library version it takes $6.4\mu s$ for the ping-pong test. The gap with the previous library version is not really significant. The main interest is to show that it is possible to develop a efficient $\mathcal{SCL} - \mathcal{C}han$ library without hardware or software keep message ordering. Such a protocol could fit a computer owning two communication network, one dedicated to clock broadcast and the other to data communication. x

6 Conclusion

$\mathcal{SCL} - \mathcal{C}han$ is a task parallel and SPMD language relying on a specific message passing communication model. It only uses one-sided and point to point synchronizations through receive executions. These synchronizations are dissociated from data communications. The termination of a synchronization depends on the waited index position in the program structure and not on a data reception. The structural clocks, specific data that encodes index positions, are used to manage synchronizations and channels.

In this paper we show that the language is deterministic and deadlock free. Note that send/receive communications through channels allow to simply guaranty determinism. This is not the case with previous approaches using the dump/put paradigm [8, 7, 11]. Moreover, since communications respect program structure, we show that is is possible to propose a synchronous programming model. Therefore, the programmer only needs to handle a single control flow as for sequential programming, while the underlying execution model is asynchronous.

Performance results confirm $\mathcal{SCL} - \mathcal{C}han$ language relevancy. It generally outperforms MPI. Nevertheless, it underlines the necessity to optimize structural clock communication pattern. It also appears that some algorithms can take advantage of send/receive unmatching.

Eventually, $\mathcal{SCL} - \mathcal{C}han$ provides both expressiveness and efficiency of task parallelism, and programming facility of synchronous data parallelism. In this way, it can be seen as a unifying approach of task and data parallelism.

We are currently studying the way asynchronous function calls could be introduced in $\mathcal{SCL} - \mathcal{C}han$. We are also developing automatic parallelization techniques using $\mathcal{SCL} - \mathcal{C}han$ as a target language.

References

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

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

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

[4] 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, August 1996.

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

[6] 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.

[7] 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.

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

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

[10] 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.

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

[12] E. Melin, B. Raffin, X. Rebeuf, and B. Virot. $\mathcal{SCL} - \mathcal{C}han$: An Asynchronous Data-Parallel Language for Irregular Algorithms. In IEEE, editor, *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.

[13] 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 27–44. Springer, 1996.

[14] 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.

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