A Simple Synchronization and Communication Multi-threaded Library for Automatic Distribution of Irregular Sequential Code [1]

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

We present a data-driven parallelization technique for sequential programs. The main originality of our approach is an absorption mechanism using specific wait/signal synchronizations based on special purpose counters called structural clocks. This mechanism allows to perform unmatching send/receive operations. Our parallelization technique can express unpredictable communications involved by irregular control structures and irregular data accesses. The granularity of the generated parallel code can be easily adapted tothe algorithm and to the architecture. Tests performed on Cray T3D provide encouraging results.

Keywords: *Automated Distribution of Sequential Program; Structural Clocks; Synchronization and Communication Model; Automatic Parallelization; Irregular Algorithm; Implementation Results on Cray T3D.*

1. Introduction

Automatic parallelization attempts to automate translation of sequential programs into efficient parallel ones. Classic parallelizations of static control loop nests use the polytope model [5] to compute the iteration domain, the set of operations and their data flow dependencies. The scheduling of parallel operations is based on a time function which takes into account mapping and dataflow dependencies. These methods fuse together expression of parallelism, data distribution and scheduling to generate a target code usually close to the data-parallel model, namely a sequence of parallel steps separated by global synchronizations. As the care to suppress useless synchronizations is then transferred to the parallel compiler, an important part of the parallelization work remains to be done.

An additional difficulty arises with irregular applications such as computations on sparse matrices. Automatic parallelizations have to face irregular remote data accesses due to irregular data distribution, dynamic control and load balancing. Collard [3] proposes a static analysis of programs involving dynamic control. He introduces *Fuzzy Array Dataflow Analysis* (FADA) yielding a conservative approximation of data dependencies. Such analysis only involves point to point synchronizations, but the main drawback with this approach is the mandatory introduction of a superset of the necessary synchronizations.

Inspector-executor approaches [10] implement a runtime way to perform parallel execution of loops involving irregular communication schemes. Before executing a loop, the inspector evaluates the dependencies between iterations. Then, iterations are distributed on the executors with the adapted synchronization scheme. As the dependence analysis is performed at runtime, this method is efficient mainly for loops involving repeatable communications schemes.

Efficient parallelization of both regular or irregular programs requires to manage complex point to point synchronizations schemes between processes. Classical message passing libraries [8, 11, 6] guarantee efficient executions. However, they impose coherent emissions and receptions, thereby needing great programming effort, especially when communications are unpredictable. To avoid these difficulties, the Bulk Synchronous (BSP) Programming model [9, 12] divides programs into a sequence of supersteps separated by global synchronizations. Each superstep contains arbitrary remote memory accesses which take effect at the next synchronization point. The model is attractive, but global synchronizations are an important limitation for performance. To reduce barrier overhead, it is possible to use an absorption mechanism, such as in DPML [7]. Barriers are replaced by pushing values of variables in stacks, as long as these stacks are not full.

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

In this paper, we propose a parallelization technique for sequential programs involving irregular data accesses. We show that it is possible to take advantage of the BSP programming model without synchronization overhead. We introduce an abstract machine yielding efficient execution of BSP-like programs. Communications take place through channels and rely on explicit *send/receive* instructions. Counters called *structural clocks* encode the position of each index with regards to the superstep structure. To handle irregular data accesses, we use a specific *wait/signal* method yielding an absorption mechanism for *send/receive* instructions . A *receive* instruction does not block on an empty channel, symmetrically, some *send* instructions may never be received. We show that it is then possible to perform a data-driven automatic translation of sequential arbitrary DOACROSS loops. The granularity of the generated parallel code can be easily adapted to the algorithm and to the architecture. A sparse Gauss-Seidel like algorithm is used to highlight our approach capabilities.

2. An asynchronous execution model

2.1. The asynchronous abstract machine

We consider a parallel and asynchronous abstract machine. Processors are called *indices* and denoted by u, v,... The constant $This$ provides the value u for each index u: $This|_u = u$. Each index u owns a *(private) memory* where it stores its private data. These data cannot be directly referenced by another index. Data exchanges take place through explicit *send/receive* communications. 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. 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. 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 program all channels are empty. Communications are asynchronous: only receive instructions are blocking.

Programs are divided in supersteps separated by logical validation of data exchanges. These supersteps allow to define a communication scheme using a temporarily masking of the values sent during a superstep. All indices carry out all supersteps. Note that there is no global or point to point synchronizations between two supersteps. The natural (sequential) logical ordering on supersteps is coded by local integer counters called *structural clocks*. When an index ends a superstep, it increments its local clock and then broadcasts it to all indices.

In each superstep, an index can send and receive messages. A send instruction is non blocking. The execution of a receive instruction relies on the superstep ordering. Let v be an index executing a receive from a channel (u, v, X) inside a superstep S. The index v owns the local clock t_v corresponding to S. It refers its local copy t_u of the clock of the sending index u. It waits for u if u has not completed all supersteps coming before S $(t_u < t_v)$. When the *waiting condition* $(t_u \geq t_v)$ is satisfied, the receive can be completed. To ensure communication determinism, a structural clock sent after a datum must also be received after. To avoid deadlock all indices must carry out exactly the same number of supersteps.

As send instructions are non-blocking, the channel (u, v, X) may contain data sent by u inside supersteps coming before or after S. Channels are managed as LIFO lists with respect to the ordering on superstep. An index that executes a send instruction stamps the sent datum by its current clock. So, structural clocks permit the index v to only consider messages in (u, v, X) sent when u was executing supersteps strictly before S. If no such message exists, v receives the `nil` value. Otherwise, it chooses the last data put in the channel during a previous superstep. Next, all data sent before the chosen value are removed from (u, v, X).

2.2. The structural clock library

Our model is implemented by a C library including five functions. Let us detail the actions performed by an index u executing each of these instructions.

scl_send (v, X) . The index u sends its (private) value of the variable X, stamped by its current clock, through the channel (u, v, X) to the index v.

scl_receive (v, X, Y) . The index u waits for the last value of X sent by the index v during previous supersteps (the current superstep is excluded). Then, it assigns this value to its variable Y and deletes this value and all the previous ones from the channel. If such value does not exists, it assigns `nil` to Y.

scl_c_send (v) . The index u sends its current clock to v.

scl_c_send_all . The index u broadcasts its current clock to all other indices.

scl_c_inc . The index u increases its clock by 1. This function call locally marks divisions between supersteps. Note that deadlock and non determinism can occur if all indices do not execute the same number of scl_c_inc instructions.

2.3. Example

The simple program displayed figure 1 encodes a pipeline using two indices, numbered 1 and 2. At the right hand side of each instruction, we display the structural clock an index owns when it has performed it (i corresponds to the loop counter). Thanks to the instruction scl_c_inc, each iteration is divided into two supersteps. Therefore, the receive instruction of iteration i refers the sending instruction of the same iteration.

Since the index 1 only executes send instructions, it is not involved in waiting steps. It repeatedly appends

for $i := 1$ to 100 do	0
if $This = 1$ and $L[i] \neq 0$ do	i
$X := i * L[i]$;	i
scl_send($This + 1$,X);	i
end;	i
scl_c_inc;	$i+1$
scl_c_send_all;	$i+1$
if $This = 2$ do	$i+1$
scl_receive($This - 1$,X,Y);	$i+1$
if $Y \neq nil$ do	$i+1$
$A[i] := A[i] + Y$;	$i+1$
end;	$i+1$
end;	$i+1$
end;	$i+1$

Figure 1. Program encoding a pipe-line between two indices.

its computation result into the channel $(1, 2, X)$. When it sends datum X at iteration i, it writes X stamped by its current clock i in the channel. Next, it increases its clock to $i + 1$ and broadcasts it. When the index 2 executes the receive of iteration i, it owns clock $i + 1$. At first, it waits for the sending index 1 as long as 1 has a structural clock strictly lower than its. Then, it assigns the last datum (with regard to its strucutral clock) stored in the channel $(1, 2, X)$ into Y and removes it. This datum corresponds to the one stamped by the structural clock i. If the channel is empty, the received datum is nil.

3. The translation function

We turn now to the distribution of sequential programs. The intuitive idea is to keep, as far as possible, the structure of the sequential program. In particular, the distributed code has the same loop structure than the sequential one. Loop counters are simply distributed on all indices. For sake of clarity, we focus on a subclass of sequential programs we call *simple* programs. A simple program S is a sequential program composed by assignments, conditionals, and for loops whose bounds are arbitrary functions of loop counters. In simple programs, as each index owns all counters, loop bounds can be evaluated locally at run time: only assignments and conditional statements have to be translated.

The translation function, denoted by $Trans$, is defined by induction on the program structure. As usual, we express the mapping of data onto indices by the way of a function $Owner$. We denote by $Owner(V)$ the index owning the variable V. The definition of $Owner$ can be easily extended to sets of variables: if Q is a set of variables, $Owner(Q)$ denotes the set of indices owning at least one of the variables of Q. Given a sequential program S, we denote by $Concern(S)$ a superset of the set of variables which appear at left hand side of assignments. The

```
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 2. Example.

set $Concern(S)$ holds *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 accessed for *reading* in the program S.

Example. Let us consider the program of figure 2. 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 $Ref(S) = \{A[i-1], A[i+1]\}$.

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

For sake of conciseness, we introduce some parallel SPMD instructions. We define the instruction msend Q to T where Q and T are sets of variables. An index executing a msend instruction, sends all values of variables belonging to it and appearing in Q to all indices owning at least a variable belonging to T. We also define the instruction mreceive Q where Q is a set of variables. An index that executes this instruction receives each variable $V \in Q$ from the index $owner(V)$ and stores them into local copies. If a `nil` value is received for $V \in Q$, then the local copy of V is not updated. The intuitive idea is the following: if an index u receives a `nil` value, it means that the value of V has not been updated by $Owner(V)$ since the last communication. In this case, the current local value of V corresponds to the remote value of V.

3.1. The translation of an assignment

We turn now to the translation of the assignment of an array component $X[Exp_1] := Exp_2$. The case of a simple variable can be easily deduced from. In accordance with the Owner Compute Rule the assignment is performed by the index $u = Owner(X[Exp_1])$ while the indices $Owner(Exp_2)$ have to send Exp_2 to u. We divide this translation in three supersteps (see figure 3) corresponding to three communication phases. First, a superset of needed values is sent. Next, the needed values are received to compute an intermediate result and this result is sent. Finally, the sent value is received. By ending a superstep, an index u makes available the last sent value for each chan-

$Trans($ $X[Exp_1] := Exp_2$ $)$	$Superstep\ 1$	msend $Ref(Exp_2)$ to $Concern(X[Exp_1] := Exp_2)$; msend $Ref(Exp_1)$ to $Resp(Concern(X[Exp_1] := Exp_2))$; scl_c_inc; scl_c_send_all;

	$Superstep\ 2$	if $This = Resp(Concern(X[Exp_1] := Exp_2))$ do
		  mreceive $Ref(Exp_1)$;
		  $Tmp := Exp_1$;
		  scl_send $(Owner(X[Tmp]), Tmp)$;
		end;
		scl_c_inc; scl_c_send_all;

	$Superstep\ 3$	if $This \in Owner(Concern(X[Exp_1] := Exp_2))$ do
		  scl_receive $(Resp(Concern(X[Exp_1] := Exp_2)), Tmp, Tmp)$;
		  if $Tmp \neq \texttt{nil}$ do
		    mreceive $Ref(Exp_2)$;
		    $X[Tmp] := Exp_2$;
		  end;
		end

Figure 3. The translation of an assignment.

nel (u, v, X). It also allows the completion of *all* receive (u, X, Y) instructions of the next superstep.

Superstep 1 Each index owning a *possibly* referred variable $V \in Ref(Exp_2)$ sends it to all indices owning *possibly* assigned variables $W \in Concern(X[Exp_1] := Exp_2)$. To choose the index v responsible for the evaluation of Exp_1 we define 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 $v = Owner(V)$. Therefore Each index owning a *possibly* referred variable $V \in Ref(Exp_1)$ sends it to $v = Resp(Owner(X[Exp_1] = Exp_2))$.

Superstep 2 This superstep concerns the evaluation by the index v of the address of the modified variable. It receives the data sent, evaluates the expression Exp_1 and sends the result to $u = Owner(X[Exp_1])$.

Superstep 3 In the third superstep, all indices owning a *possibly* assigned variable $W \in Concern(X[Exp_1] := Exp_2)$ perform a receive instruction. As only the index $u = Owner(X[Exp_1])$ will receive the message sent at the second step, this index performs the assignment. As there is no sending instruction in this step, no scl_c_inc instruction ends this translation. This last superstep will be ended by the end of the program or by the next scl_c_inc instruction in the translated program.

Remark: 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.

3.2. The translation of a conditional statement

Three supersteps are also needed to translate the conditional construct if Exp do S end.

Superstep 1 Each index owning a *possibly* referred variable $V \in Ref(S)$ sends it to all indices owning *possibly* assigned variables $W \in Concern(S)$. Then, each index owning a *possibly* referred variable $V \in Ref(Exp)$ sends it to $v = Resp(Owner(S))$.

Superstep 2 This superstep concerns the evaluation by the index v of the condition. It receives the data sent, next it evaluates the expression Exp and finally it sends the result to all indices owning a *possibly* assigned variable.

For sake of conciseness, we introduce the function scl_c_inc (N) that increases the clock of an index performing it by N. We also introduce the function COUNT_INC (P) where P is a parallel program. This function computes the number of instructions scl_c_inc appearing in P. Note that it does not take into account scl_c_inc (N) instructions and that it can be computed at compile-time.

Superstep 3 This step is conditioned by the result of the evaluation of the expression Exp. It consists in the execution of the program $Trans(S)$ by all concerned indices. Therefore, this step contains the supersteps of $Trans(S)$. Indices which are not concerned increase their structural clock from the number

$Trans($ if Exp do S end $)$	$Superstep\ 1$	msend $Ref(S)$ to $Concern(S)$; msend $Ref(Exp)$ to $Resp(Concern(S))$; scl_c_inc; scl_c_send_all;

	$Superstep\ 2$	if $This = Resp(Concern(S))$ do
		  mreceive $Ref(Exp)$;
		  $Tmp := Exp$;
		  msend $\{Tmp\}$ to $Concern(S)$;
		end;
		scl_c_inc; scl_c_send_all;

	$Superstep\ 3$	if $This \in Owner(Concern(S))$ do
		  scl_receive $(Resp(Concern(S), Tmp, Tmp)$;
		  if Tmp do;
		    $Trans(S)$;
		  else
		    scl_c_inc (COUNT_INC $(Trans(S))$);
		    scl_c_send_all;
		  end;
		end

Figure 4. The translation of a conditional statement

COUNT_INC $(Trans(S))$ of supersteps of the program $Trans(S)$. As the function COUNT_INC does not take into account instructions scl_c_inc (COUNT_INC $(Trans(S))$, this ensures the coherence between the number of supersteps performed by all indices.

Remarks :

- If $Concern(S)$ is reduced to a singleton, we have $Resp(Concern(S)) = Owner(Concern(S))$. So, the communications relating to the Tmp variable become useless.
- In contrast to classical data-driven parallelization techniques [1, 2], note that each index $u \notin concern(S)$ which does not have to update variables in the conditioned instructions, does not receive the values needed to evaluate the boolean condition.
- Our function can be easily extended to arbitrary non simple sequential programs. In fact, for loop bounds can be evaluated during the execution, using a method similar to the evaluation of the condition in the second step of the conditioning translation.
- The mapping takes place through the $Owner$ function which can be chosen after the code generation. This allows us to use ever method to choose a good mapping.

3.3. Example

To illustrate our translation scheme, we reuse the example of the figure 2. We have already computed supersets $Ref(S)$ and $Concern(S)$. The figure 5 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 execute the instruction scl_c_inc (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.

4. 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 value of the considered component only if the result is not null (not to propagate null values in the matrix). It is an interesting problem to parallelize since it includes numerous dependencies and anti-dependencies. The corresponding sequential program is displayed figure 6. 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

```
for i := 1 to 100 do
  msend {A[i]} to {A[i+1]};
  msend {} to {A[i+1]};
  scl_c_inc; scl_c_send_all;
  if This = Owner(A[i+1]) do
    mreceive {};
    Tmp := i+1;
    scl_send (Owner(A[i+1]), Tmp);
  end;
  scl_c_inc; scl_c_send_all;
  if This = Owner(A[i+1]) do
    scl_receive (Owner(A[i+1]), Tmp, Tmp);
    if 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]};
  scl_c_inc; scl_c_send_all;        1
  if This = Owner(A[i]) do
    mreceive {A[i+1]};              2
    Tmp := (A[i+1] ≠ 0);
    msend {Tmp} to {A[i]};
  end;
  scl_c_inc; scl_c_send_all;
  if This ∈ {A[i]} do
    scl_receive (Owner(A[i]), Tmp, Tmp);
    if Tmp do
      msend {A[i−1], A[i+1]} to {A[i]};
      msend {} to {A[i]};
      scl_c_inc; scl_c_send_all;
      if This = Owner(A[i]) do
        mreceive {};
        Tmp := i;
        scl_send (Owner(A[i]), Tmp);
      end;
      scl_c_inc; scl_c_send_all;
      if This = Owner(A[i]) do
        scl_receive (Owner(A[i]), Tmp, Tmp);
        if Tmp ≠nil do
          mreceive {A[i−1], A[i+1]};
          A[Tmp] := A[i−1] * A[i+1];
        end;
      end;
    else
      scl_c_inc (2); scl_c_send_all;
    end;
```

Figure 5. Translation of the program figure 2

in the translation of the conditional become useless (cf. remark 3.1 page 4 and remark 3.2 page 5). Note also that all underlined instructions can be deleted.

5. Optimizations and mapping

In this section, the distributed Gauss-Seidel like program is considered. First, we show that it is possible to reduce communication cost by introducing overlapping between communication and computation. Next, we take into account the mapping to perform additional optimizations. Finally, we reduce communications involved by structural clocks management.

5.1. Communication and computation overlap

A simple data dependence analysis allows to move some send instructions in order to favor computation and communication overlap. Each element of the matrix is modified exactly one time. Therefore, data can be sent as soon as they are evaluated. For a given iteration (I, J), the sending instructions of $A[I, J-1]$ and $A[I-1, J]$ is replaced by two send instructions respectively at the end of iterations $(I, J-1)$ and $(I-1, J)$. For the same reason, as any modification of the data $A[I, J+1]$ and $A[I+1, J]$ occur before they are sent, the sending instructions are replaced by two send instructions at the beginning of the program (page 9).

5.2. Optimizations linked to data mapping

To implement the distributed algorithm, we have to address the problem of mapping, i.e. we have to distribute data on the available number of processors. The mapping takes place through the $Owner$ function which can be chosen after the code generation.

Let us denote by Pe_Nb the number of available processors. We choose a fixed mapping, distributing cyclically 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: $Owner(A[I, J]) = (J\ div\ L)$.

Once the mapping function $Owner$ is chosen, it is possible to perform additional optimizations by taking this function semantics into account.

- **Communication optimizations.** Since all elements of a column are owned by the same index, all communications among the column become local, and then, can be removed (for example the instruction msend $(A[I, J])$ to $Owner(A[I+1, J])$).
- **Computation optimizations.** Since the translation function distributes each loop iteration on all the indices, an index u may executes some useless itera-

The sequential program

```
real:A(N:N)
do I=1,N-2
   do J=1,N-2
     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

```
for I := 1 to N − 2 do
  for J := 1 to N − 2 do
    msend(A[I, J − 1], A[I + 1, J], A[I, J + 1], A[I − 1, J]) to Owner(A[I, J]);
    scl_c_inc; scl_c_send_all;
    if This = Owner(A[I, J]) do
      mreceive(A[I, J − 1]);
      if 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]);
        scl_c_inc; scl_c_send_all;
        if This = Owner(A[I, J]) do
          mreceive(A[I + 1, J]);
          if 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]);
            scl_c_inc; scl_c_send_all;
            if This = Owner(A[I, J]) do
              mreceive((A[I, J + 1]);
              if 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]);
                scl_c_inc; scl_c_send_all;
                if This = Owner(A[I, J]) do
                  mreceive((A[I − 1, J]);
                  if 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]);
                    scl_c_inc; scl_c_send_all;
                    if 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
                  else scl_c_inc (1); scl_c_send_all; end
                else scl_c_inc (1);scl_c_send_all;end
              else scl_c_inc (2); scl_c_send_all; end
            else scl_c_inc (2);scl_c_send_all;end
          else scl_c_inc (3); scl_c_send_all; end
        else scl_c_inc (3);scl_c_send_all;end
      else  scl_c_inc (4); scl_c_send_all; end
    else  scl_c_inc (4); scl_c_send_all; end
  end
end
```

Figure 6. The Gauss-Seidel like algorithm and the corresponding translation.

```
#define scl_jump_block(jump_1,jump_2,first,last,clock_increment,
{first_iter},{intermediate_iter},{last_iter}){

int j;
scl_c_inc((jump_1)*clock_increment);
j=first;
{first_iter}
for(++j;j<last;j++){
   {intermediate_iter}}
j=last;
{last_iter}
scl_c_inc((jump_2)*clock_increment);}
```

Figure 7. The macro Scl_jump_block.

tions. Nevertheless it is often possible to detect at compile time such iterations thanks to a static analysis.

We say an iteration is *useless* for an index if this index has nothing else to do but performing tests and handling clocks (scl_c_inc, scl_c_send). We introduce a new loop structure called $jump$ loop. This loop retains the semantics of the for loop but each index u only performs iterations which might be *useful*. An index skipping an iteration does not perform scl_c_inc instructions. Therefore, in order to ensure structural clocks coherence, such index has to update its local clock from the number of scl_c_inc instructions performed in the skipped iteration. As we distribute the matrix by block of L columns, an index u only has to communicate with the index $u-1$ (respectively $u+1$) to compute the elements of the first (respectively last) column of its block. For the intermediate columns, all needed data are local. Thus, the mapping distinguish three different codes: first iteration, last iteration and the intermediate ones. Therefore, to implement a $jump$ loop, we introduce a macro called Scl_jump_block (see figure 7 where the parameters have the following rôle :

- `first` and `last` respectively represent the number of the first and last usefull iteration to perform.
- `jump_1` and `jump_2` respectively denote the number of useless iterations to skip before and after the `first` and `last` usefull iteration.
- The number of scl_c_inc instructions in one iteration corresponds to `clock_increment` (it does not take into account scl_c_inc (N)). The intuition is that an index skipping an iteration increments its clock from `clock_increment`.
- `{first_iter}`, `{intermediate_iter}` and `{last_iter}` denote codes corresponding respectively to the first, intermediate and last iteration codes. Note that clocks are incremented of the same value in all these codes and that this value can be computed at compile-time.

5.3. Optimizations linked to communication of structural clocks

Since structural clocks are incremented and broadcasted after each Scl_send instruction, we introduce some useless communications. We developed several optimizations to reduce this overhead. Since clocks are used only by receiving indices, it is sufficient to send a clock only one time, between a send and the corresponding receive, after the last clock update.

The final optimized program is displayed figure 8.

6. Implementation on Cray T3D

Performances results are obtained from an implementation of the structural clock library on a Cray T3D with the Cray library SHMEM [4]. Our main goal was to test the overhead due to the structural clock machinery and the practical interest of send/receive unmatching. We compare our results with the ones provided by the PAF parallelizer developed at the University of Versailles [5].

The PAF parallelizer translates sequential programs into equivalent data parallel programs. The obtained programs have been ported into CRAFT (Cray Research Adaptive Fortran Tools), a library allowing to program the T3D in the data parallel model.

To highlight the variations of the communication overhead according to the index number, we performed tests without null values and with various index numbers (Fig. 9 - top). Each index always owns the same number of data ($N = 1000$, $L = 2048$). For few processors, our parallel program reaches performances similar to the CRAFT program generated by PAF. However the CRAFT program appears to be less scalable.

We ran our program with different ratio of null values, with 32 indices, $N = 1000$ and $L = 2048$ (Fig. 9 - bottom).

```
for(i=1;i<N-1;i++){
  if( 0<This && This<Pe_nb){
    scl_send(This-1,a[i][This*L]);
  }
  scl_c_inc(1);

  scl_jump_block(This==0 ? 0 : This*L-1,
  This==Pe_nb-1 ? 0 : (Pe_nb-This-1)*L-1,
  This==0 ? 1 : 0,This==Pe_nb-1 ? L-2 : L-1,1,
    { /*code for the first iteration*/
      scl_c_send(this-1);
      if(a[i-1][j]!=0 && a[i+1][j] !=0
         && a[i][j+1]!=0 )
        {
          scl_receive(This-1,a[i][j-1],tmp1);
          if(tmp1!=0){
            a[i][j]=a[i-1][j]*a[i+1][j]*
              tmp1*a[i][j+1];
          }
         }
       scl_c_inc(1);

    },
   { /*code for the intermediate iterations*/
      if(a[i-1][j]!=0 && a[i+1][j] !=0
         && a[i][j-1]!=0  && a[i][j+1]!=0 ){
        a[i][j]=a[i-1][j]*a[i+1][j]
          *a[i][j-1]*a[i][j+1];
      }
      scl_c_inc(1);
    },
    { /*code for the last iteration*/
      if((a[i-1][j]!=0 && a[i+1][j]!=0)
         && a[i][j-1]!=0)
        {
        scl_receive(This+1, a[i][j+1],tmp2);
        if(tmp2!=0){
          a[i][j]=a[i-1][j]*a[i+1][j]*a[i][j-1]*tmp2;
        }
        scl_send(This+1,a[i][j]);
        scl_c_inc(1);
        }
    }
   )
}
```

Figure 8. The optimization of the Gauss-Seidel like program.

The performance improvement is relevant for each ratio of null values. Costly waiting steps are avoided since we do not carry out useless receive instructions. Obviously the CRAFT program yields a constant computation time, since computations does not take account of the sparsity.

7. Conclusion

We propose an asynchronous execution model for BSP like programs. Communications take place through channels and rely on explicit *send/receive* instructions. We introduced unbalanced and dynamic point to point synchronizations based on a specific *wait/signal* method. In irregular frameworks, non-matching *send* and *receive* improves performances.

We propose an automatic parallelization method mainly based on this capability. It allows to distribute a sequential program with irregular control structures, such as `if` statements or arrays with subscripts evaluable only at run-time. The generated code is independent of the mapping. Thus, mapping can be automatically evaluated at compile time (static load balancing), or at run-time and improved during execution (dynamic load balancing) when static analysis become too complex.

Experimental results on Cray T3D show that translated program performances overcome parallelized programs obtained with classic static approaches generating data parallel code. It also appears that sparsity can be efficiently exploited. These good performances are mainly due to structural clocks mechanism that yields loosely synchronized executions.

New developments in automatic parallelization, such as WHILE loops parallelization, are under progress. We also plan to introduce functions in our model.

References

[1] 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.

[2] C. Bareau and C. Jard. Dynamic Analysis of Parallelized Sequential Programs using Instrumented Semantics. In *European School of Computer Science Parallel Programing Environments for Hight Performance Computing*, Alpe d'Huez, April 1996.

[3] J.-F. Collard. *Parallélisation Automatique des Programmes à Contrôle Dynamique*. PhD thesis, École Normale Supérieure de Lyon, 1995.

[4] Cray Research. *Application Programmer Library Reference Manual*.

[5] 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.

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

[7] G. Hains and C. Foisy. The Data-Parallel Categorical Abstract Machine. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE '93: Parallel Architectures and Languages Europe*, pages 56–67. Springer-Verlag, Berlin, DE, 1993.

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

[9] W. F. McColl. General purpose parallel computing. In A. M. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation. Proc. 1991 ALCOM Spring School on Parallel Computation*, pages 337–391. Cambridge University Press, 93.

[10] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-Time Scheduling and Execution of Loops on Message Passing Machines. *Journal of Parallel ans Distributed Computing*, 8:303–312, 1990.

[11] 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.

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

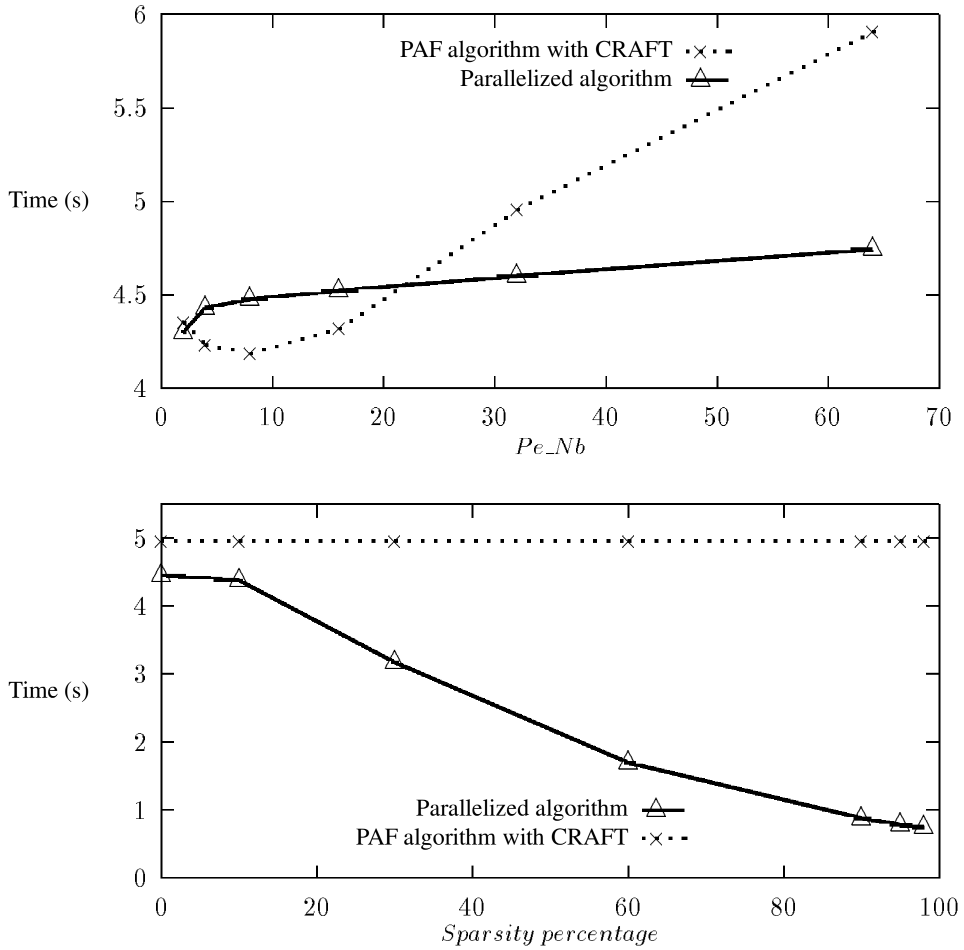

Figure 9. Gauss-Seidel like algorithm with a 1024*2048 matrix (Top: without null values; Bottom: with null values).