ReLaCS for systolic programming

Frédéric Raimbault, Dominique Lavenier

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

ReLaCS for Systolic Programming

Frédéric Raimbault, Dominique Lavenier

N° 1981

Mai 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

INRIA
RENNES

ReLaCS for Systolic Programming

Frédéric Raimbault, Dominique Lavenier *

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet API

Rapport de recherche n° 1981 — Mai 1993 — 12 pages

Abstract: The ReLaCS language is a systolic programming language, which simplifies the programmer's task by making explicit the data-flow of systolic algorithms, and by exposing the data delivery mechanism. The underlying architecture model is different from other SIMD architectures in that it physically separates computation and data management. We introduce the ReLaCS language as a syntaxic and a semantic extension of the C language. We show in this article that the ReLaCS programming model provides a simple programming method for systolic algorithms, which is applicable to a variety of parallel machines.

Key-words: programming language, ReLaCS language, data-parallelism, SIMD architectures, MicMacs machine, systolic algorithms, Levenshtein algorithm.

(Résumé : tsvp)

Soumis à ASAP'93

*{raimbaul,lavenier}@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Programmation systolique avec ReLaCS

Résumé : ReLaCS est un langage de programmation systolique qui simplifie la tâche du programmeur en permettant l'expression des flôts de données caractéristiques des algorithmes systoliques. Le modèle d'architecture cible diffère d'autres modèle SIMD par la présence de deux controleurs; l'un gère la gestion des données, l'autre le calcul. Nous introduisons le langage ReLaCS comme une extension syntaxique et sémantique du langage C. Nous montrons dans cet article que le modèle de programmation ReLaCS proccure une méthode de programmation simplifiée des algorithmes systolique, qui est applicable sur un ensemble varié de machines parallèles.

Mots-clé : langage de programmation, langage ReLaCS, parallélisme de données, architectures SIMD, machine MicMacs, algorithmique systolique, distance de Levenshtein.

1 Introduction

Since the introduction of the systolic architecture concept by Kung and Leiserson in 1978 [21], numerous algorithms and designs have been proposed to solve compute-bound problems in signal and image processing, matrix arithmetic or dynamic programming applications. Their property of regularity and locality make them well suited for VLSI implementation.

To ensure of the validity of the algorithm prior to chip fabrication, a functional simulation is needed. Hardware description languages as VHDL [2] could be employed to simulate systolic architectures. But these languages depend on simulated execution with a speed inadequate for realistic tests. Only parallel execution can provide the computing power of thousand millions operations per second required for real-time simulation.

Programmable parallel machines are now available to develop, test and execute systolic algorithms. However, programming such machines correctly is a difficult task: both computation within the systolic cells and systolic data transfers between cells must be specified. The activity of the cells which provides the concurrency for computation is relatively easy to identify. But the input-output management which is responsible for communication and control raises subtle issues: synchronization protocols; dead-lock avoidance; I/O interfaces; and data partionning.

Three kinds of parallel programming languages are available. The first possibility is a dedicate language. For example, APPLY [11] works on low-level video processing, ASSIGN [26] on signal processing, AL [30] on matrix calculus. These languages have been proposed to map regular computations on parallel machines. But their limited used make them unsuitable for general systolic programming.

A second possibility is to describe the different tasks of cell computation and I/O management of a systolic network with a language based on communicating sequential processes such as Occam [25]. This kind of language introduces powerful concepts such as control parallelism and low-level communication primitives. On the other hand, it does not allow one to easily express regular structures. This drawback also applies to W2, the language developed for theWarp machine [4]. In addition W2 requires the programmer to regard low level architectural details such as communication link names.

The third possibility for systolic programming is based on the data parallelism [13]. The advantage of this approach lies in its synchronous programming model and its global view of data exchanges as operations acting on data collections. However, well-known data-parallel languages [7, 23, 29, 24] are highly dependent on the SIMD architectural model. They assume that a particular set of features is available, such as routing, virtual processor support, activity mask, reduction and scan operations [31]. On the other hand they do not resolve the problem of I/O, which is fundamental in systolic computations, as we will demonstrate further on this article. Recently, a langage based on C++, called *New Systolic Language*, has been proposed in[15]. But in NSL data flow directives are separated from computation statements and are expressed in their own language. In our opinion, this misleads the programmer and is not natural for programming.

To bridge the gap between systolic algorithms and parallel architectures, we propose a programming language, called ReLaCS. This language captures the essential features of systolic machines in a simple programming model and allows the programmer to get use of the capabilities of the underlying target architecture.

The paper is organized as follow: section 2 presents the programming model and the execution model of systolic machine we use. Section 3 introduces the ReLaCS programming language. Finally we deal with compilation aspects in section 4.

2 The systolic machine model

As characterized by H.T. Kung [20], a systolic network is a synchronous parallel architecture composed of identical simple cells that are locally and regularly connected; data move through the network at constant speed, and interact where they meet. To exploit this concept on programmable machines, cells are replaced by processors and an inter-processor communication mechanism which avoids the overflow of the local memory

is introduced [19]. Among others, such machines include the iWarp [6], MicMacs [22], SPLASH [10], B-SYS [14] and Blitzen [5].

We fist present in this section the ReLaCS programming model, the view of systolic machines held by a programmer writing in the ReLaCS language; second, we present the ReLaCS execution model used by the compiler to bridge the gap between the programming model and the target machine.

2.1 The ReLaCS programming model

The programming model we have retained is at first sight close to data-parallel [13]. The programmer sees his systolic machine as a programmable accelerator connected to a general purpose workstation. This accelerator appears as a SIMD network composed of a linear array of identical, conventional processors that communicate synchronously with their nearest neighbours. Only the two end-processors are linked to the host (see figure 1). The host broadcasts the same instruction to each processor, which executes the instruction on its own data.

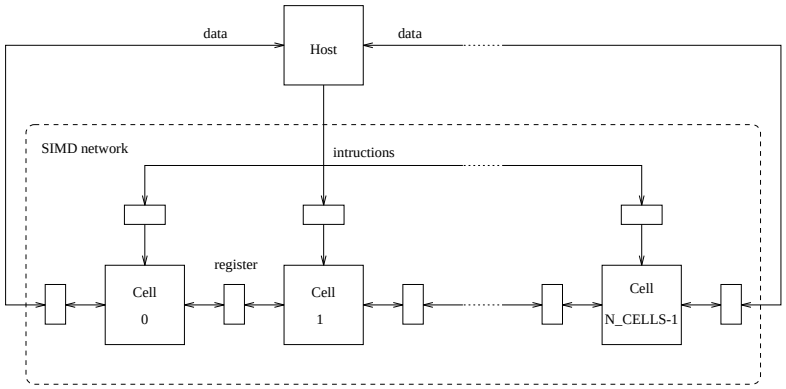


Figure 1: ReLaCS programming model

The user writes a single source program in ReLaCS, from which the compiler generates code for execution both on the host and on each cell of the network. The partitionning is explicitely done by data types and is described in section 3. The compiler handles the details of communication protocols between each component and code generation for the target machine; this is further explained in section 4.

2.2 The ReLaCS execution model

Our execution model aims at providing efficient use of a linear systolic array. The basic idea consists of exploiting the parallelism which exists between the computation performed on the array, and the computation performed outside the array for managing the input-output data.

The ReLaCS execution model makes the assumption of the concurrent execution of two processes on the target machine:

- The *computation process* which executes the calculation task in a systolic fashion with fine-grain access to the input/output ports. This takes place on a linear array of identical processors (the systolic array). Data transfers operate on word-size messages and are restricted to neighbouring processors. The execution mode of the processors depend on the target machine.

  Each elementary processor of the systolic array may contain a controller, in which case the execution mode is SPMD: The processors runs independently the same program and synchronize during communication phases.

  Alternatively, the elementary processor may not contain a controller but receive the instructions from a single controller for the whole array. In this case the execution mode is SIMD: The processors

run synchronously the same program. However it appears useful to provide some autonomy to the processor in SIMD execution mode. So we suppose that a conditional assignment operation is present in the instruction set of the processor. This kind of instruction exists, for instance, in the elementary processor of MicMacs, Blitzen and B-SYS machines.

- The *data management process* is responsible for correctly ordering data and for collecting results from the *computation process*. The *data management process* runs on a scalar processor with its own controller. It communicates directly with the extremities of the systolic array.

Synchronization between the processes occurs during communications and conditional instructions. These are generated by the compiler, which also takes care of the execution mode of the systolic array to produce optimized code for various machines (see section 4).

3 The ReLaCS language

The ReLaCS language is designed to take advantage of the architectural features of the programming model presented in the previous section. It is also intended to be independent of the cell processor and to provide a simple way of programming data movements across a systolic architecture. These goals led to the choice of an explicit form of parallel programming to reach maximal efficiency. On the other hand, as the systolic computer control is not radically different from that of a conventional computer, it would be wasted effort and a source of confusion to design a completely new language. So we chose to extend the C language [17], a well know and efficient compiled language. This approach has already been used to develop high-level programming languages for parallel architectures [18, 8, 23]. The extensions provide specific ways of expressing algorithms for systolic computers, without imposing a programming style different from that used for the host.

The following section presents these specific extensions and illustrates them on a typical systolic algorithm.

3.1 Data structures

The programmer of our machine model needs a way to differentiate between host variables (scalar variables) from those which are located on the systolic array (systolic variables). For this, we have extended the C language storage class set. We will just mention that a storage class specifier serves, in the C language to qualify a variable, in that it causes an appropriate amount of storage to be reserved. The ReLaCS language defines a new storage class specifier, the `systolic` class, that is used to reserved variables on each processor of the network. The default class, the `static` class, specifies a scalar variable residing on the host.

A program expression operating on systolic variables, produces a synchronous execution of the subsequent operations on each processor of the array. A program expression operating on scalar variables performs the subsequent operations only on the host. Variables implied in an expression must be of the same class. Exchanges between systolic and scalar variables occur during communication operations decribed in section 3.3.

The class storage just defined is independent of the usual type notion. Basic types of the ReLaCS language are the integers (`int`), the floating numbers (`float`) and the characters (`char`); the only type constructor allowed is the array (`[]`).

The predefined integer constant `N_CELLS` gives the number of available cells in the network at execution time; this is useful for dynamic loop control, especially in the case of propagation by systolic shifts.

The example given on figure 2 reflects the execution of the following assignment statement between two systolic variables on a network of 3 cells: `x= y`.

3.2 Control structures

The flow of control is sequential. The ReLaCS language offers the same iterative and conditional control structures `while()`, `for()` and `if()` as the C language. However their semantic in ReLaCS must take into

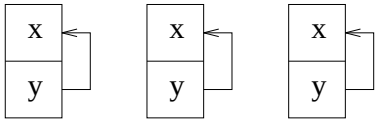


Figure 2: parallel data assignment

consideration the synchronous programming model. The SIMD execution model, which requires that all the processors receive the same instruction, complicates the treatement of the conditional jump.

Let us examine the case of a `if` clause from which depend instructions to be sent to the network of processors. Two cases can occured, depending on the storage class associated with the condition:

- The condition class is `static`. The condition is evaluated on the host and all the processors receive the instructions corresponding to the selected code branch.
- The condition class is `systolic`. In this case, all the processors may not have to execute the same code branch. Taking into account the programming model of ReLaCS (cf. section 2.1), the compiler points out a class error at the time of semantic checking of the program.

Classical SIMD languages which make the assumption of an activity mask on the target machine, use it to send successively the instructions of the two code branchs. Depending on their activity bit set by the condition evaluation, the processors execute the instructions of one code branch and remaind inactive during the other. Overlapped conditions are treated with context stack and restoration. We chose to ignore these compilation problems and to only support a much simpler SIMD model. Our experiences in the systolic programming field confirm the validity of this choice and the algorithms we studied did not require the activity mask concept. In general, the more local conditionals on SIMD machines lead to a sequentialization of the execution of the two code branch, and so to a reduction of array processor efficiency.

On the other hand, it appears to us useful to have a more limited local control; for example, to realize a maximum operation in each processor of the array. The conditional assignment instruction introduced in our programming model provides for this local test without requiring one sequencer per cell. And the conditional expression of ReLaCS supports such a local test. The expression has the same syntax but a different meaning than its C counterpart.

The basic form of a conditional expression is written `c ? a : b`, and returns `a` if `c` is true, otherwise `b`. That is the only case where a `systolic` class condition is allowed. For instance, the computation of the greater of two values in each processor is expressed by the following conditional expression `z= x>y ? x : y`. In C, this expression had the same meaning as the conditional instruction `if (x>y) z= x; else z= y` and the compiler translates it as such. In ReLaCS, the compiler generates a conditional assignment instruction from a basic conditional expression. However a much more elaborate conditional expression, where each member could be a complex expression, may be a function with hiden effect; it becomes quickly impossible to maintain the same semantic as the C language. The ReLaCS compiler generates successively the computation of each member, the condition evaluation, and finally the conditional assignment instruction. We note that two conditional members are always evaluated.

3.3 Communications

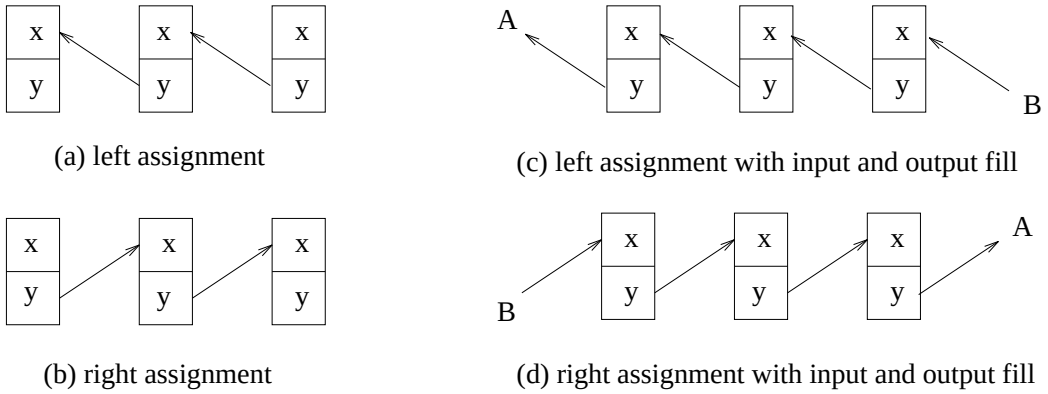
In systolic architectures, data transfers between processors are very important. Special care is devoted to this I/O mechanism in the ReLaCS language. New operators match the hardware architecture and express the tight coupling beetween neighbouring cells.

The programming model assumes a SIMD execution mode and synchronous communications. Correct use of this model implies that the emission of a value by a processor in one direction is followed by the reception of the data sent by the neighbouring processor located in the opposite direction. This sequence of operations is synthetized in assignment operators acting on `systolic` class variables:

- Left assignment operator, example: `x=< y`. Each processor sends a value to the left and receives one from the right (figure 3(a)). The extreme right side processor doesn't change the contens of its variable `x`.
- Right assignment operator, example: `x=> y`. Each processor sends a value on the right and receives one from the left (figure 3(b)). The extreme left side processor doesn't change the contens of its variable `x`.

The global effect of theses operators is a shift of the network variables, which reflects the data flows characteristic of systolic algorithms. From here it is quite natural to extend the range of these operators to bind the input-output of the network. Two optional parameters to systolic assignments handle the boundary conditions, i.e. assign the array input and output to `static` variables on the host:

- On the left assignment operator: `x : A=< y : B`. The extreme right side processor receives the value of `B` sended by the host. The processor located on the extreme right side sends the content of its `x` variable to the host which stores it in `A` (figure 3(c)).
- On the right assignment operator: `x : A=> y : B`. The extreme left side processor receives the value of `B` sended by the host. The processor located on the extreme right side sends the content of its `x` variable to the host which stores it in `A` (figure 3(d)).



(a) left assignment

(c) left assignment with input and output fill

(b) right assignment

(d) right assignment with input and output fill

Figure 3: systolic communications

The second way of expressing a communication between the host and the network consists of broadcasting the same value to all the processors. This communication is explicit, and we have also devoted one assignment operator to it:

- the global assignment operator: `x =| B`. The host emits the value of `B` which is stored by all the processors in the variable `x` (see figure 4).

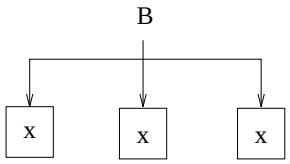


Figure 4: data broadcasting

The broadcast communication is not really a systolic operation because it doesn't preserve the locality concept. Despite this, we retain this operation because it had the following advantages:

- no hadware had to be added on a SIMD machine to support it: the value is transmitted by the instruction bus,
- it replaces with on instruction, a sequence of communication instructions to spread the value,
- it speeds execution of systolic algorithms on SIMD machines.

The introduction of these new operators free the programmer from the synchronization management during communication phases.

3.4 Example

We illustrate here the ReLaCS language features with a typical example of a systolic program. Moreover we will described in section 4 the compilation process of ReLaCS using this program. The example used is the Levenshtein distance calculation between two strings of symbols [32].

Let $R = r_1.....r_m$ and $T = t_1.....t_n$ be two strings to compare and $D(i,j)$ the distance between $r_1.....r_i$ et $t_1.....t_j$. The Levenshtein distance is given by the following recurrence relation:

$$D(i,j) = Min \begin{cases} D(i-1,j-1) + d(r_i,t_j) \\ D(i-1,j) + 1 \\ D(i,j-1) + 1 \ , \end{cases} \tag{1}$$

with the initial conditions:

$$\begin{array}{l} D(0,0) = 0 \\ D(i,0) = D(i-1,0) + 1 \quad 1 \leq i \leq m \\ D(0,j) = D(0,j-1) + 1 \quad 1 \leq j \leq n \ . \end{array} \tag{2}$$

where $d(r_i,t_j)$ represents the cost of replacing r_i by t_j, 1 the cost of adding t_j and the cost of omitting r_i.

The distance between R and T is the value $D(m,n)$ computed when solving this recurrence. In a typical application, such as spelling correction, this calculation must be repeated many times since the same test string must be compared to many reference strings (a full dictionary, for instance). This algorithm and its parallel implementation on systolic arrays has been carefully study in [9]. We will just outline the points useful for the explanation program.

The basic idea for the implementation on a linear array is to load one symbol of the string T per processor and to pass the R strings through the network from left to right, one symbol per computation cycle as shown on figure 5. When the last symbol of a string R is processed by the first processor, this one is available for a new computation, after re-initialization.

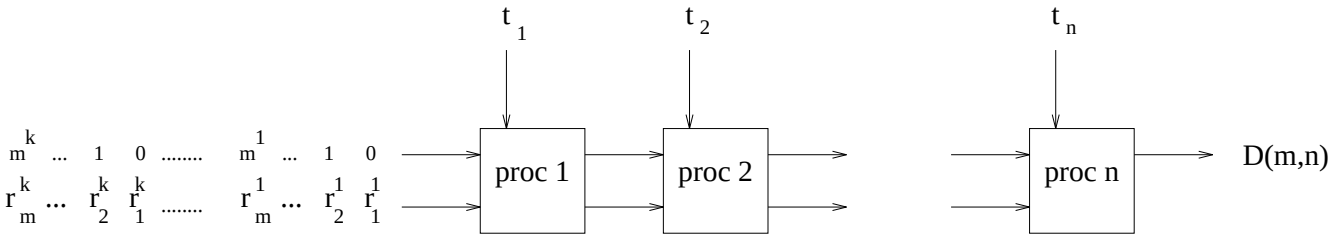


Figure 5: Levenshtein distance computation on a linear array

At each computation cycle the processor j receives from its left neighbour the partial distance $d(i,j-1)$ and $d(i-1,j-1)$ (the distance $d(i-1,j)$ is computed locally); then it computes the replacement cost, the addition cost and the omission cost; finally it retains the minimum according to the equation (1). So a

partial computation flows through the network from left to right too, and become a distance result when it reaches the right-end processor.

We give in figure 6 the interesting part of the basic computation loop (after network initializing and loading of test string). Comments appear between the delimitors `/*` and `*/`; parts of the program not detailed are summarized between angle brackets.

```
<... initializations ...>
  while ( J < B[M]+N_CELLS ){ /* reference symbols dictionary enumeration */
    d_s= d_a; /* d(i-1,j-1) -> d(i,j) */
    d_o= d; /* d(i,j-1) -> d(i,j) */
    <...initialization of D0_a by d(i,0)...>
    if ( J-N_CELLS+1 == B[K+1] ) /* test of complete distance computation */
       d_a : D[K++] => d : D0_a; /* shift the distance and store the result */
     else
       d_a => d : D0_a; /* shift the distance but do not store the result */
    <...reference symbol transfer...>
    <... editing cost computation...>
    d= MIN(c_s,c_a,c_o); /* equation computation */
   <...indicies incrementation...>
   }
```

Figure 6: Programmation du calcul de la distance de Levenshtein.

Data structures used on the host are: an array of characters containing the `M` words of the dictionnary; an array of integers (`B`) pointing to the beginning of each word of the dictionnary; an array of integers (`D`) storing the computed distances; two indecies (`J` and `K`) over the arrays `B` and `D`; and an integer (`D0_a`) seting the initializing value $d(0, j)$.

Data structures required on the systolic network are: integer variables `d_s`,`d_a`, `d_o`, `d` receiving respectively the partial distances $d(i-1, j-1)$, $d(i, j-1)$, $d(i, j-1)$, $d(i, j)$; and temporary integer variables `c_s`, `c_a`, `c_o` used for the cost calculation respectively of replacement, addition and omission.

The distance calculation is programmed as follows:

- a nested loop realizes the dictionnary enumeration. It requires as many iterations as the number of characters of the dictionary added to the number of processors (`N_CELLS`). This increase takes into account the network latency and allows us to treat the shut-down phase the same as the steady-state phase.
- The first assigments operate on systolic variables. They keep the distance $d(i-1, j-1)$ and $d(i, j-1)$.
- The conditional statement decides whether or not the distance computed by the right-most processor is complete. If yes, the shifting of the distances $d(i-1, j)$ is saved on the host. Otherwise, the distances are shifted but the last one is not stored on the host.
- The minimum function is computed locally according to the equation (1).

4 Compiling ReLaCS

Compiling source programs occurs in two phases: a analysis phase and a synthesis phase.

The analysis phase of a ReLaCS program uses well-known compiler techniques[1] to determine whether or not the program is syntaxically and semantically correct. Class memorisations notably are examined with like-type checking rules.

The synthesis phase of a ReLaCS program produces C source programs. We make the assumption of a C compiler and of communicating librairies on the target machine. This method simplifies the compiler development and permits a real portability to most existing machines. The compiler generates two concurrent C programs for a parallel target machine and one C program for a sequential target machine.

4.1 Compiling for parallel machines

Generating code from a ReLaCS program for parallel machines requires several steps. Among these steps:
- separating the parallel operations done in the network (systolic computation process) from the scalar ones (input-output data management process).
- Generating a C program for each process and adding communication and synchronization operations.
- Compiling separately each C program with the compiler and the communication librairies of the target machine.

For example, the compilation of the Levenshtein program for a SIMD target machine (see figure 6) gives the C programs of figure 7.

The program on the left holds the operations to be realized by the input-output process; that on the right holds the operations to be realized by the computation process. We find again the instructions of the source program, plus communication and control functions. The input-output management process tests loop terminations and conditionals, and passes on the results to the computation process using the functions `set_flag()` and `is-flag()`. The right assignment operators are translated into basic communication operations in the systolic computation program (`shift_right()` and (`shitf2_right()`) and in the input-output management program when it sends data to the network (`snd_right()`) or receives data (`rcv_left()`). The difference between `shift_right()` and `shitf2_right()` is that in the first case the right-most processor emits a value for the host and in the second case it does not.

As we have pointed it out in paragraph 2.2 it is possible to compile a ReLaCS program on a MIMD machine. In this case, the input-output data management code is loaded on a processor and the systolic computation code is loaded on the others. Finally, a virtual ring is etablished between all the processors. The equivalence between the asynchronous execution on a MIMD machine and the synchronous execution on a SIMD machine is guaranted by the synchronization points made up of the data exchanges and the condition broadcasts.

Often no hardware exists on MIMD machines to support efficiently the broadcast of the value of one processor to the others. On the other hand, each processor of a MIMD machine owns a sequencer which is able to independently compute the conditions, since they do not depend on communications. Using the technique of [12], the compilation of ReLaCS programs for MIMD machines is optimized by the duplication of control operations in the systolic computation program.

To illustate this optimizing technique, we give in figure 8 the programs generated for the Levenshtein algorithm on a MIMD machine. In comparaison with the programs of figure 7, the functions `set_flag()` and `is_flag()` have disappeared, replaced by loop and conditional tests and increment operations added to the systolic computation program.

4.2 Compiling for sequential machines

To be able to compile parallel programs for sequential machines is of most importance for algorithm developement and debugging. It allows deterministic execution, global state observation, and program profiling. The compilation of a ReLaCS program for a sequential machine produces a single C program.

This C program is obtained by indexing systolic variables and sequentializing parallel and communication operations. Indexing of systolic variables consists of translating variable into arrays of size `N_CELLS` (or multi-dimensional arrays if the variable is already an array). Sequentializing parallel operations is obtained by wrapping operations acting on systolic variables into a loop covering the processors' domain. Sequentializing communication operations consists of decomposing data transfers into `N_CELLS` shifts in the induced array.

```
<... initializations ...>
  while ( set_flag(J < B[M]+N_CELLS) ){


   <...initialization of D0_a by d(i,0)...>
   if ( set_flag(J-N_CELLS+1 == B[K+1]) ){
     snd_right(D0_a);
     rcv_left(&D[K++]);
   }
   else
     snd_right(D0_a);
   <...reference symbol transfer...>


   <...indecies incrementation...>
  }
```

```
while ( is_flag() ){
  d_s= d_a;
  d_o= d;

  if ( is_flag() )
    shift_right(d,&d_a);


  else
    shift_right2(d,&d_a);
 <...reference symbol transfer...>
 <... editing costs computation...>
 d= MIN(c_s,c_a,c_o);

}
```

Figure 7: Programs generated for the Levenshtein algorithm on SIMD machines: (a) input-output management program, (b) systolic computation program.

```
<... initializations ...>
  while ( J < B[M]+N_CELLS ){


   <...initialization of D0_a by d(i,0)...>
   if ( J-N_CELLS+1 == B[K+1] ){
     snd_right(D0_a);
     rcv_left(&D[K++]);
   }
   else
     snd_right(D0_a);
    <...reference symbol transfer...>


   <...indecies incrementation...>
  }
```

```
<... initializations ...>
  while ( J < B[M]+N_CELLS ){
    d_s= d_a;
    d_o= d;

    if ( J-N_CELLS+1 == B[K+1] )
      shift_right(d,&d_a);


    else
      shift_right2(d,&d_a);
    <...reference symbol transfer...>
    <... editing costs computation...>
    d= MIN(c_s,c_a,c_o);
   <...indecies incrementation...>
  }
```

Figure 8: Programs generated for the Levenshtein algorithm on MIMD machines: (a) input-output management program, (b) systolic computation program.

The example of the compiling of the Levenshtein program illustrates these ideas. Figure 9 represents the translation of the end-of-distance calculation test and the sequentialization of the minimum operation.

```
< ... >
    if ( J-N_CELLS+1 == B[K+1] ){
      D[K++]= d[N_CELLS];
      for (I_CELL= 1; I_CELL < N_CELLS; I_CELL++)
        d_a[(N_CELLS+1)-I_CELL]= d[(N_CELLS+1)-(I_CELL+1)];
      d_a[1]= D0_a;
    }
    else{
      for (I_CELL= 1; I_CELL < N_CELLS; I_CELL++)
        d_a[(N_CELLS+1)-I_CELL]= d[(N_CELLS+1)-(I_CELL+1)];
      d_a[1]= D0_a;
    }
    < ... >
    for (I_CELL= 1; I_CELL < N_CELLS+1; I_CELL++)
      d[I_CELL]= MIN(c_s[I_CELL],c_o[I_CELL],c_a[I_CELL]);
< ... >
```

Figure 9: Program generated for the Levenshtein algorithm on a sequential machine

5 Conclusion

We have presented the ReLaCS language for programming linear systolic arrays on parallel architectures. This language is based on the data-parallel concept to facilitate the programming of regular computations. Specific operators expressed as assignment statements match the data flow that occurs in a systolic architecture and the I/O of the network. The ReLaCS compiler translates the source program into C concurrrent programs corresponding to the computation task and the I/O management task of a systolic algorithm. The compiling method employed allows ReLaCS to be efficiently ported to a variety of machines, including SIMD and MIMD parallel machines. The programming environment also includes a compiling option for sequential execution and debugging. We have shown using a typical example, the Levensthein distance algorithm, the programming of a systolic computation and the C source files given by the compiler for, successively SIMD, MIMD, and sequential machines.

The ReLaCS environment includes an operational compiler for the following parallel machines : iPSC/2 [16], ArMen [27] and iWarp [6]. Porting ReLaCS to the iPSC/2 machine was of course not efficient, but it permits us to test our compiling optimization for MIMD machines. It has been followed by the study of a hardware communication improvement on message-passing-based machines. This work has been tested and validated on the programmable logic layer of the ArMen machine [28]. Recently, experiments conducted on the iWarp machine and linear speed-up obtained on a linear programming problem [3] confirm our approach. Among others applications developped with ReLaCS, matrix computations, video coding and string processing have been coded.

Current research involved the automatic partitionning of systolic algorithms on regular structures.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers : Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] R. Airiau, J.M. Berge, V. Olive, and J. Rouillard. *VHDL : du langage à la modélisation.* Presses Polytechniques et Universitaires Romandes, 1990.

[3] R. Andonov, F. Raimbault, and P. Quinton. *Dynamic Programming Parallel Implementation for the Knapsack Problem.* Internal Report 740, IRISA, July 1993. Submitted to JPDC. Available by anonymous ftp on irisa.irisa.fr.

[4] M. Annaratone, E. Arnould, T. Gross, H.T. Kung, M. Lam, O. Menzilcioglu, and J. Weeb. The Warp Computer : Architecture, Implementation, and Performance. *IEEE Transactions on Computers*, C-36(12):1523–1538, December 1987.

[5] D. Blewins, E. Davis, R. Heaton, and J. Reif. BLITZEN : A Highly Integrated Massively Parallel Machine. *Journal of Parallel and Distributed Computing*, 8(2):150–160, February 1990.

[6] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H.T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp :An integrated Solution to High-Speed Parallel Computing. In *International Conference on Supercomputing*, 1988.

[7] P. Christy. Software to Support Massively Parallel Computing on the Maspar MP-1. In IEEE, editor, *COMPCON SPRING*, February 1990.

[8] H. Dietz and D. Klapphholz. Refined C : A Sequential Language for Parallel Programming. In *International Conference on Parallel Processing*, pages 442–449, August 1985.

[9] P. Frison, E. Gautrin, D. Lavenier, and J.L. Scharbarg. Designing specific systolic array with the api15c chip. In *ASAP 90*, September 1990.

[10] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti. Building and Using a Highly Parallel Programmable Logic Array. *Computer*, January 1991.

[11] L. Hamey, J. Webb, and I-C. Wu. An Architecture Independent Programming Language for Low-Level Vision. *Computer Vision, Graphics, and Image Processing*, 2(48):246–264, November 1989.

[12] P. Hatcher and M. Quinn. *Data-Parallel Programming on MIMD Computers.* MIT Press, 1991.

[13] W. Hillis and G. Steele. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[14] R. Hughey. *Programmable Systolic Arrays.* PhD thesis, Brown University, 1991.

[15] R. Hughey. Programming Systolic Arrays. In *International Conference on Application Specific Array Processors*, 1992.

[16] *iPSC/2 and iPSC/860 Manual Set.* Intel Scientific Computers, 1990.

[17] Kernighan, W. Brian, and D. M. Ritchie. *The C programming language.* Prentice-Hall, 1978.

[18] J.T. Kuhen and H.J. Siegel. Extensions to the C programming language for SIMD/MIMD parallelism. In *International Conference on Parallel Processing*, 1985.

[19] H.T. Kung. Systolic Communication. In *International Symposium on Computer Architecture*, pages 695–703, May 1988.

[20] H.T. Kung. Why Systolic Architectures? *Computer*, 15(1):37–46, 1982.

[21] H.T. Kung and C.E. Leiserson. Algorithm for VLSI Processor Arrays. In *Introduction to VLSI systems*, chapter 8.3, Addison-Wesley, 1980.

[22] Dominique Lavenier. MicMacs : un réseau systolique linéaire programmable pour le traitement des chaines de caractères. Thèse de l'Université de Rennes 1, June 1989.

[23] K. Li and Herb Schwetman. Vector C : A Vector Processing Language. *Journal of Parallel and Distributed Computing*, (2):132–169, 1985.

[24] M. Metcalf and J. Reid. *FORTRAN 90 Explained.* Oxford Science Publications, 1990.

[25] *Occam 2, Reference Manual.* 1988.

[26] D. O'Hallaron. *The ASSIGN Parallel Program Generator.* Technical Report CMU-CS-91-141, Carnelie Mellon University, May 1991.

[27] B. Pottier. Armen, une machine parallèle intégrant un réseau de circuits reconfigurables. Thèse de l'université de Rennes I, June 1991.

[28] F. Raimbault, D. Lavenier, S. Rubini, and B. Pottier. Fine grain parallelism on a MIMD machine using FPGAs. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 2–8, Napa, California, April 1993. Available by anonymous ftp on irisa.irisa.fr under internal report number 728.

[29] J. Rose and G. Steele. *C*: an extended C language for data parallel programming.* Technical Report PL87-5, Thinking Machine Corporation, 1987.

[30] P. S. Tseng. A systolic Array Programming Language. In *International Conference on Application Specific Array Processors*, September 1990.

[31] R. Tuck. *Porta-SIMD : An Optimally Portable SIMD Programming Language.* PhD thesis, University of North Carolina, May 1990.

[32] R. A. Wagner and M. J. Fisher. The string to string correction problem. *Communications of the ACM*, 21:168–173, 1976.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399