C-STOLIC User Guide

D. Lavenier[1] - G. Le Fol[1] - F. Raimbault[2]

(1) IRISA – Campus de Beaulieu – 35042 Rennes cedex – France
(2) Université de Bretagne Sud – 1 rue de la Loi – 56000 Vannes – France

Abstract

This document is a quick reference to the C-stolic language and to the C-stolic compiler freeware provided for UNIX workstations.[1]

1 Introduction

C-stolic was first designed for programming a linear systolic array: the MicMacs machine [3]. Then the C-stolic compiler evolved to support simulation of systolic algorithms on parallel machines [6], such as the iPSC/2 [2], the iWarp [9], the ArMen machine [4] or the MasPar [1]. In addition, a simulator running on UNIX workstations has been developed for testing and debugging purposes.

Only the simulator is provided within the freeware. It allows programmers to easily design, experiment, and validate systolic algorithms. Currently, C-stolic supports only the linear topology architecture.

The use of the C-stolic simulator is straightforward. The programmer writes one or several C-stolic procedures and calls them from a main C program. The compilation process is reduced to a single command, as compared to the normal use of the C-stolic compiler which requires a *makefile* command file when compilation for parallel machines is targeted. But the same C-stolic program working on a workstation should also run on a parallel machine and provide identical results.

The reader interested in further information on the C-stolic parallel implementation may refer to [6] [8] [7]. The complete source code can be given on simple request, but no support will be provided.

The C-stolic language is based on a virtual architecture from which a programming model is derived (cf section 3). It provides a simple way of expressing data movements across a systolic architecture. To reach maximal efficiency, an explicit form of parallel programming

[1]http://www.irisa.fr/CSTOLIC

is proposed in addition to parallel objects exactly matching the size of the array. But as the systolic control is not radically different of a conventional computer control, C-stolic is based on a simple extension of the C language.

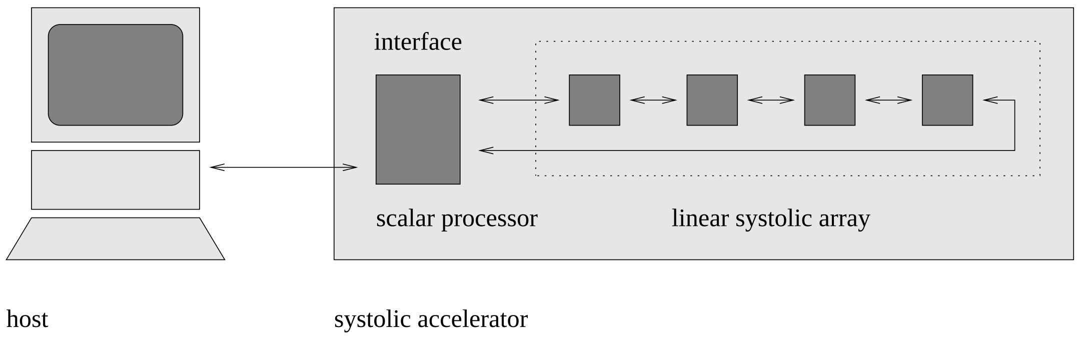
The extensions provide specific ways of expressing algorithms for systolic architecture without imposing a programming style different from that used for writing C programs. Actually, C-stolic introduces a minimum of change compared to C: a new class storage – the `systolic` class – to make the distinction between parallel and scalar objects, and three *systolic* assignment operators allowing the objects to communicate.

However, the C-stolic compiler takes only a restricted syntax of C. For example, complex structures are not allowed. These decisions were taken for two reasons: (1) to simplify the design of the C-stolic compiler; (2) because the parallel computations on systolic architecture most often use only simple data structures.

The rest of the document is organized as follows: section 2 indicates how to insert C-stolic procedure into a main C program. Section 3 presents the C-stolic programming model. Section 4 is devoted to the extensions introduced into the C-stolic language. Section 5 presents the C-stolic compiler for UNIX workstations.

2 Programming environment

The C-stolic language has been designed for programming systolic accelerators. It adopts the architecture target scheme as depicted on the figure below: a workstation (the host) connected to a linear systolic array through an interface (a scalar processor) which manages I/O communication.



The main application is written in C and calls procedures written in C-stolic. These procedures are supposed to be executed on the systolic accelerator.

The data exchanges between the host and the accelerator are expressed using the procedure arguments. Let us consider the following C and C-stolic codes:

```
C code

main()
{
  int i;
  int A[100];
  int B[100];

  for (i=0;i<100;i++)
    scanf("%d",&A[i]);
  CsOn();
  Reverse(A,B);
  CsOff();
  for (i=0;i<100;i++)
    printf("%d ",B[i]);
}
```

```
C-stolic code

Reverse(X,Y)
int X[100], Y[100];
{
  systolic int k;
  int i;

  for (i=0;i<100;i++)
    k => k : X[i];

  for (i=0;i<100;i++)
    k : Y[i] =< k;
}
```

The `Reverse` procedure body is not detailed here. We focuss on the call procedure mechanism. In the main C program, the `Reverse` procedure takes two arguments, array `A` as input and array `B` as output. There is no way, as in C, to differentiate input and output arguments. However, in order to handle properly the exchanges, the C-stolic compiler determines automatically the transfer direction. From the C-stolic sight point of view, the two parameters `X` and `Y` are objects located on the scalar processor.

The programmer has to write two source files: the computation done on the host (C file) and the computation done on the accelerator (CS file). Hence, in the above example, the main procedure will be written in the `example1.c` file, while the `Reverse` procedure will be written in the `example1.cs` file.

The `CsOn()` function initializes the connection with the accelerator. It verifies, for example, that the hardware is running and loads the C-stolic code before execution. When running on a sequential machine this function forks a UNIX process which communicates through pipes with the main program. The `CsOff()` function terminates the connection with the accelerator.

3 The C-stolic programming model

This section focuses on the systolic accelerator. It gives the programming model which must be taken into consideration by the programmer to write correct C-stolic programs. The programming model is illustrated by figure 1.

The programmer sees the systolic accelerator as a programmable linear array connected to a scalar processor. The array supports the parallel implementation of the algorithms, while the scalar processor is reponsible for supplying the array with data. Both are driven by a unique sequencer in a SIMD fashion: the sequencer broadcasts the same instruction to each

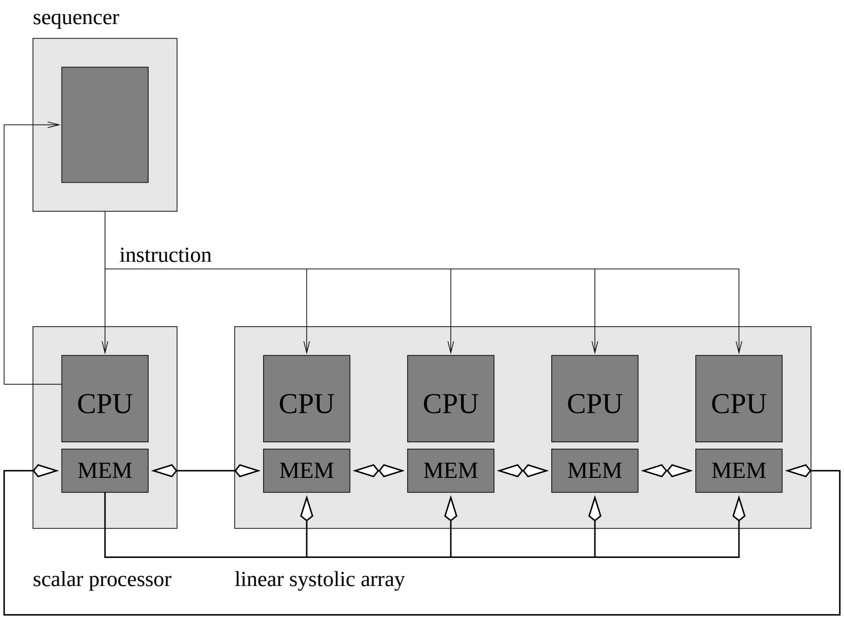


Figure 1: *The C-stolic programming model: the programmer consider a single sequencer which broadcast instructions both to the array and to the scalar processor. The instruction sequencing control can only be affected by the scalar processor.*

processor which executes the instruction with its own data. Depending on the nature of the instruction, it is executed either by the array or either by the scalar processor. Section 4.1 explains how this distinction is made.

Only the scalar processor can do instruction sequencing. Conditions on loop controls or branch selections must depend on scalar variables. There is no way to detect directly a global state on the array and to make a decision accordingly (eg, tst of convergence). This choice will be explained in section 4.3.

The array is composed of identical processors that communicate synchronously with their nearest neighbours. Only the two end-processors can exchange data with the scalar processor. An extra link has been added for broadcasting a data from the scalar processor to all the processor of the array. This doesn't follow the systolic concept since the locality is not preserved, but in many cases, this possibility simplify greatly the programming initialization steps.

The C-stolic programming model doesn't include the host communication channels. Actually, when programming the systolic accelerator, the programmer makes the assumption that data are present in the scalar processor memory. As mentioned in the previous section, the compiler determines automatically which arguments are input or output data. Hence, it is supposed that the execution of a C-stolic procedure is always preceded by an input acquisition prologue, and always followed by an output delivery epilogue. These activities are hidden from the programmer and depend on the implementation.

This programming model has been developed for simplifying the programming task. But recall that this is only the programmer's view! Execution schemes other than SIMD execution may be implemented, depending on the hardware structure [5]. The C-stolic compiler has been designed with this idea in mind. The complete version of the C-stolic compiler is able to generate code for different target architectures, including MIMD parallel machines.

4 The C-stolic language

This section presents the C-stolic extension of the C language. Basic knowledge of C is assumed.

4.1 Data structures

The programmer needs a way to differentiate the scalar variables from those which are located on the systolic array. The C language storage class has been extended accordingly: C-stolic defines a new storage class specifier, the `systolic` class, that is used to reserved variables on each processor of the systolic array. Other classes specifiy scalar variables, residing on the scalar processor.

A program expression operating on systolic variables produces a synchronous execution of the subsequent operation on each processor of the array. A program expression operating on scalar variables performs the subsequent operations only on the scalar processor. Exchanges between systolic and scalar variables occur during systolic assignment described in section 4.2. Note that arithmetic expressions cannot mix systolic and scalar data!

The class storage just defined is independent of the usual notation. Basic types of the C-stolics 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 `NBCELL` gives the number of available cells in the systolic array at execution time. This is useful for dynamic loop control, especially in the case of propagation by systolic shift.

4.2 Systolic Assignments

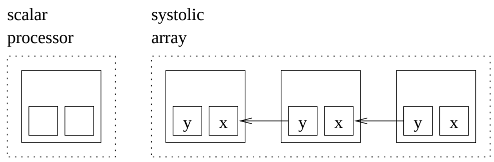
In systolic architectures, data transfers between processors are important. Special care has been devoted, when defining the C-stolic language, to handle properly the I/O mechanism. New operators match the hardware architecture and express the tight coupling between neighboring cells.

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

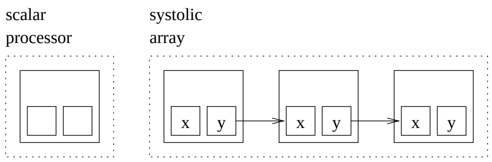
- **left assignment operator**: `x =< y`. Each processor sends a value (a systolic variable or a systolic expression) to the left and receives one from the right. The extreme right side processor doesn't change the contents of its variable `x`.

- **right assignment operator**: `x => y`. Each processor sends a value (a systolic variable or a systolic expression) to the right and receives one from the left. The extreme left side processor doesn't change thec contenst of its variable `x`.

The global effect of these operators is a shift of the array variables, which reflects the data-flow characteristics of systolic algorithms.



left assignment x =< y

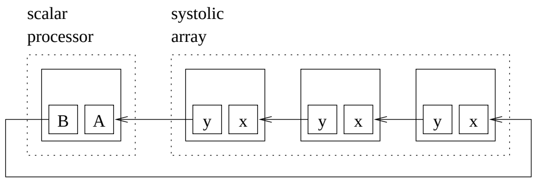


right assignment x => y

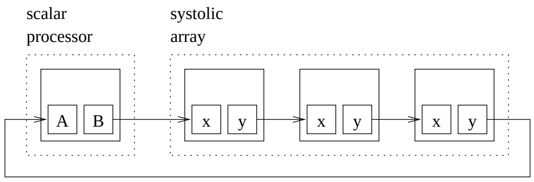
From here, it is quite natural to extend the range of these operators to bind the input-output of the array. Two optional parameters to systolic assignments handle the boundary conditions, i.e. assign the array input and output to scalar variables:

- for the left assignment operator: `x : A =< y : B`. The rightmost processor receives the value of the variable `B` (or any scalar expression) and stores it in `x`. The leftmost processor sends `y` which is stored in the scalar variable `A`

- for the right assignment operator: `x : A => y : B`. The leftmost processor receives the value of the variable `B` (or any scalar expression) and stores it in `x`. The rightmost processor sends `y` which is stored in the `static` variable `A`

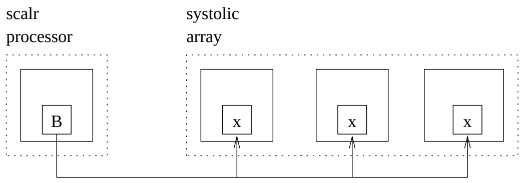


left assignment x : A =< y : B



right assignment x : A => y : B

The second way of expressing a communication between the scalar processor and the array consists of broadcasting the same value to all the processors. This communication is explicit and one operator is devoted to this task: the global assignment operator: `x =| B`. The scalar processor emits the value of `B` (or any scalar expression) which is stored by all the processors in the variable `x`.



data broadcasting x =| B

4.3 Control structures

The flow of control is sequential. The C-stolic language offers the same iterative and conditional control structure `while()`, `for()` and `if()` as the C language. However, in C-stolic, their semantic must take into consideration the synchronous model. The SIMD execution mode, which requires that all the processors receive the same instruction, complicates the treatment of the conditional jump.

Let us examine the case of a `if` clause which may cause different instructions to be sent to different processors of the array. Two cases can occur, a correct case and a wrong case, depending on the storage class associated with the condition:

- **correct case:** the condition class is not `systolic`. The condition is evaluated on the scalar processor and all the processors of the array receive the instruction corresponding to the selected code branch;
- **wrong case:** the condition class is `systolic`. In that case, all the processors may not have to execute the same code branch. Taking into account the programming model of C-stolic, *a class error is detected.*

Thus, the condition of an *if* statement cannot contain systolic class varaibles.

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 conditional evaluation, the processors execute the instructions of one branch and remain inactive during the other. C-stolic does not support this execution mode. Our experiences in the systolic programming field confirm the validity of this choice:

local conditions on SIMD machines lead to a sequentialization of the execution of the two code branchs, and consequently to a reduction of the efficiency.

On the other hand, it appears useful to have a more limited local control; for example, to compute a maximum operation in each processor. The conditional assignment instruction provides a local test without requiring one sequencer per cell.

The basic form of a conditional expression is written `c ? a : b`, and returns `a` if `c` is true and `b` otherwise. This is the only case where a `systolic` class condition is permitted inside a condition. For instance, the computation of the greater of two values is expressed by the following conditional expression: `z = x>y ? x : y;`

Notice that the expression have the same syntax as in C but have a different meaning: the local condition expression `z = (cond) ? (exp_1) : (exp_2)` evaluates both the expressions `exp_1` and `exp_2` before assigning `z`.

5 The C-stolic compiler

The C-stolic compiler permits the user to simulate systolic algorithms on different parallel programmable machines. The compilation process is divided in two main steps: the generation of a C program from C-stolic source code, and the compilation of this C program with the native compiler of the parallel machines. Actually, this is a little bit more complicated: the C-stolic compiler generates several C files which are linked with libraries depending of the target parallel machine.

For UNIX platforms, the C-stolic compiler generates two C files which correspond respectively to a host-accelerator interface procedure and a program executed by the systolic accelerator. These files – and appropriate libraries – must be linked with the main C program to obtain the executable code.

The C-stolic compiler is invoked with the `cs` command. For instance, the file `csprog.cs` which contains C-stolic procedures must be compiled as follows:

```
cs csprog.cs
```

The result of the compilation is two C files named `csprog.host.c` and `csprog.simd.c`. These files must then be compiled with the native C compiler (or another C compiler, such as gcc for example) of the workstation and linked with some libraries.

In order to simplify the compiling process, a higher-level command is available for the default case. It constrains the programmer to give the same file name for the main C program (suffixed by `.c`) and the C-stolic procedure (suffixed by `.cs`). This command is invoked as follows:

```
ccs filename NBCELL
```

As mentioned above, `filename.c` contains the main C program and `filename.cs` contains the C-stolic procedures. The second argument, `NBCELL`, indicates the size of the linear array.

The `ccs` command generates two executable codes named `filename` and `filename.simd`. To start the simulation one has to invoke `filename` which runs the main application. When the `CsOn()` function is meet, the program starts a new process (`filename.simd`) which emulates the systolic accelerator.

References

[1] T. Blank. The Maspar MP1 Architecture. In *COMPCON*, feb 1990.

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

[3] D. Lavenier. *MicMacs : un réseau systolique linéaire programmable pour le traitement des chaines de caractères.* Thèse, Université de Rennes 1, Jun 1989.

[4] D. Lavenier, B. Pottier, F. Raimbault, and S. Rubini. Fine grain parallelism on a MIMD machine using FPGAs. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 2–8, Napa, CA, USA, apr 1993. IEEE Computer Society Press.

[5] D. Lavenier, F. Raimbault, and P. Frison. I/O and Computation Overlap on SIMD Systolic Arrays. *Journal of VLSI Signal Processing*, 9(3), April 1995.

[6] F. Raimbault. *Etude et réalisation d'un environnement de simulation parallèle pour les algorithmes systoliques.* Thèse, Université de Rennes 1, jan 1994.

[7] F. Raimbault and D. Lavenier. Relacs for Systolic Programming. In *ASAP 93*, pages 132–135, Venice, Italy, Oct 1993. IEEE Computer Society Press.

[8] F. Raimbault, P. Quinton, and D. Lavenier. Architectures Systoliques et Parallélisme de données; l'environnement de programmation ReLaCS. *TSI*, 12(5):597–620, 1993.

[9] Borkar S., Cohn R., Cox G., Gleason S., Gross T., Kung H.T., Lam M., Moore B., Peterson C., Pieper J., Rankind L., Tseng P. S., Sutton J., Urbanski J., and Webb J. iwarp: An integrated solution to high-speed parallel computing. In *Proceedings of Supercomputing '88*, pages 330–339. IEEE Computer Society and ACM SIGARCH, nov 1988.