Computational Science In High School Curricula: The ORESPICS Approach

P. Mori and L. Ricci

Dipartimento di Informatica,Università di Pisa
Corso Italia 40, 56125-Pisa (Italy)
{mori,ricci}@di.unipi.it

Abstract. This paper presents a new approach for the introduction of computational science into high level school curricula. The approach is based on the definition of an ad hoc environment, including a programming language, suitable for this target of age. The language includes a set of simple constructs supporting both concurrency and the management of a graphical interface. The solutions of some classical problems are shown.

1 Introduction

Computational science is a new interdisciplinary research area which applies concepts and techniques from mathematics and computer science to solve real life problems. Computational science has introduced a new methodology to investigate real life problems. Instead of defining a theory of a physical phenomenon and verify it through a set of experiments, a computational model of the phenomenon that can be simulated through the computer is defined. In this way, the phenomenon can be monitored directly through the computer. This methodology is currently supported by sophisticated environments resulting from the recent advances in computer technology.

Several university curricula include computational science courses. Furthermore, a set of proposals for the introduction of computational science into the high school curricula have also been presented [4, 8]. The goal of these proposals is to increase the interest of a larger number of students in scientific disciplines: the rationale is that students are more interested in learning mathematical concepts if these can be applied to real life problems. Furthermore, the use of the computer makes the learning even more appealing. Yet, the introduction of computational science into the high school curricula and/or in the undergraduate courses requires to settle some issues. First of all, the basic mathematical skills to support a first training in computational science are to be defined. These skills should support the development of models for a minimal, yet significative, kernel of applications. These applications should be characterized either by simple mathematical models or by a complex one that may be simplified without loosing its connection with real life. A further critical issue is a software environment suitable for young students. This issue is one of the most challenging because most current tools to develop computational software have been defined for expert users only. As a matter of fact, most applications are developed through *FORTRAN* or *C* extended with a set of libraries supporting concurrency and visualization of scientific data. These libraries are often tied to a specific operating system and the user needs some knowledge of this system as well.

We believe that a more friendly environment, including a programming language, should be developed specifically for these introductory courses. In this way, all the constructs are integrated in the same language, rather than spread across several libraries. The language should preserve the main features of existing ones, like concurrency and graphical interface support. On the other hand, the set of constructs should be reduced to a minimal kernel.

The didactic language should be *concurrent* because concurrency is a powerful tool to simplify the description of the applications. Several phenomena can be modeled as a set of concurrent, interacting entities. Consider, for instance, the simulation of the dynamics of a fluid or of a gas which can be modeled as a set of interacting molecules. An example in shown in sect. 3. Furthermore, most scientific applications are developed on highly parallel systems because of their high computational needs and the software development for these systems usually requires the knowledge of a concurrent language. Hence, the basic concepts of concurrency should be acquired as soon as possible. However, libraries such as *MPI*, *PVM* [11, 13], or *OpenMp* are not suitable, because of their complexity. As a matter of fact these libraries include several semantic equivalent primitives differing, for instance, only because of their implementation. A didactic environment should introduce a single construct for each different concept of the language. The didactic language should support a simple *graphical interface* as well, so that the student can monitor the behaviour of the concurrent activities directly on the screen. Complex visualization techniques based on sophisticated mathematical techniques, like rendering or textures, are not required in an introductory didactic environment. However, the teacher can exploit the basic mechanisms of the language to implement more sophisticated visualization techniques.

The remainder of this paper presents the *ORESPICS* environment, the new didactic environment we propose to support the teaching of computational science in high schools. The environment includes a new language, *ORESPICS-PL* that integrates a minimal set of graphical primitives with a minimal set of concurrent constructs. The graphical primitives are mostly taken from the Logo language [2], while the concurrent part of the language is based on the *message passing paradigm*. The environment and the language are fully described in [5, 6]. Sect. 2 briefly recalls the main constructs of the language and describes the ORESPICS environment. To describe how *ORESPICS* can be exploited in an introductory course, sect. 3 and 4 show some simple, yet significative problems and their *ORESPICS* solutions. Sect. 5 presents some related work. Sect. 6 draws some preliminary conclusions.

2 The ORESPICS Language and Development Environment

The sequential part of Orespics-PL includes traditional imperative constructs (repeat, while, if,...), the turtle primitives of the Logo language [2] to control the agents' movements and a set of primitives to modify the external aspect of an agent and to create sounds. The language supports all the elementary data types (integer, boolean,..) and the only data structure is the list.

An Orespics-PL program includes a set of *agents*, interacting through messages exchange. It is possible both to pair each agent with a different code and to define a *breed* of agents characterized by the same behaviour. Each agent belonging to a breed may be identified by a set of indexes. In this way, a *SPMD* programming style can be exploited.

Agent interact through a minimal, but complete set of communication primitives. Two basic kinds of communication modes, corresponding respectively to *synchronous* and *buffered* communication modes of MPI, are available. The corresponding sends are:

SendAndWait *msg* **to** *agent*
SendAndnoWait *msg* **to** *agent*

Buffer management for buffered communications is delegated to the run time support.

Orespics-PL does not define other communication modes. This is consistent with the choice to include constructs corresponding only to the *basic mechanisms* of the message passing paradigm. Other communication modes in fact, like blocking, ready or persistent modes of MPI, can be considered *optimized versions* of the previous ones, to enhance the performance of parallel programs.
The receive construct:

WaitAndReceive msg **from** agent

waits until a message is received from the selected agent. Orespics-PL also defines an *asymmetric version* of the receive:

WaitAndReceiveAny msg **from** agent

In this case, the receiver selects, according to a *non deterministic strategy*, one of the messages sent by any active agent of the microworld. Furthermore, the function

inmessage(*agent*)

allows polling of incoming messages, without actually receiving them. *Inmessage(A)* returns true if there is at least an incoming message from agent *A*. The message can be received through a **WaitAndReceive** command. The function *Inmessage(Any)* tests the presence of messages incoming from any agent. The set of collective communications includes the synchronization barrier:

Waitagents()

and two versions of the broadcast send, respectively synchronous and asynchronous:

SendAllAndWait()
SendAllAndnoWait()

Each agent involved in a broadcast communication executes a different primitive, a broadcast send, or a receive. A single primitive with distinct semantics relating to the agent executing it, could be confusing. Other collective communication, like MPI scatter, gather or reduce, can be emulated through point to point or broadcast communications. Since Orespics-PL provides a mechanism to define macros, the students can develop their own implementation of these primitives.

Collective communication involving *subsets* of agents, can be defined in Orespics-PL by associating a set of properties with each agent. The simplest kind of property is its *breed*. The breed of an agent is defined during the initialization phase and can be exploited in the communication commands to restrict the set of senders/receivers in a communication. Each agent belonging to a breed can be further identified by a set of properties. The value of these properties may be statically initialized in the declarative part of the agent's code, through the *property construct*. Afterwards it can be dynamically updated. The property mechanism is fully described in [6] and it will be exploited in the applications of section 3.

The Orespics environment supports the development of Orespics-PL programs. The user may define the appearance and the kind of each agent and pair a set of animations and/or sound with it. Furthermore, the user defines the execution environment, i.e. a microworld where the agents moves and interact. The kind of an agent *A* defines if there is a single instance of *A* or if it belongs to a breed and, in this case, how many instances of *A* should be created. Even if Orespics associates a default icon with each agent, the user can change this icon or choose a new one from a predefined set of images. Furthermore, several images can be defined for each agent. We will see an example of this in sect. 4. Each image is uniquely identified by its name, and each agent, at run time, can select its image through the *setimage* command. An agent may be paired with an animation as well. This is realized by selecting a set of frames which can be displayed cyclically or from the first to the last one and on the other way round, continuously. It is also possible to associate a sound with any agent. Each sound is uniquely identified and can be selected by an agent through the *setsound* command.

The execution environment is initialized by choosing the background image and music. At the microworld initialization, the standard icons of all the agents are displayed. The user defines the initial position of each agent by simply dragging and dropping its icon, whilst the initial position of the agents belonging to a breed is automatically decided by the system, but it can be updated through the *ORESPICS-PL* positioning commands. The whole environment is based on a friendly interface, based on a set of windows. A detailed description of the environment is presented in [6].

3 A Cellular Automata

This section and the following one show how $ORESPICS$ should be exploited in introductory computational science courses. The first example, presented in this section, shows a cellular automata simulating the dynamic behaviour of a gas. Sect. 4 discusses the solution of searching and optimization problems.

The definition of models for fluid or gas dynamics is an active area in computational science. Computational fluid dynamics describes physical phenomena through partial differential equation, like the *Navier-Stokes* ones, whose solution requires non trivial mathematical techniques which are generally acquired in advanced mathematical course. Nevertheless, simpler models result from solving these equations through *finite differencing methods* that introduce a set of discrete approximations and these models are closer to the real phenomena. These models can be exploited to present basic concepts of computational fluid dynamics in introductory courses. As an example of a simple, yet realistic, problem consider the diffusion of heat on a square metal sheet. The temperature at an inner point can be computes as the average of the temperatures of the four neighboring points. In $ORESPICS$, it is rather simple to define a data parallel concurrent program, where the agents corresponds to the points of the sheets: the program could display the temperature of the sheet by pairing distinct temperature values with distinct colors.

```
*Agent Particle*$_{i,j}$
    **property** mypos
    **setimage** Particle
    initial-positioning()
    **repeat**
        **Waitagents()**
        $\backslash *$ *Movement*
        **forward** 1
        mypos $\leftarrow$ **pos()**
        mydir $\leftarrow$ **heading()**
        **Waitagents()**
        $\backslash *$ *Directions Echange*
        **SendAllAndNoWait** mydir **toBreed** (Particles) **withProp** (position=mypos)
        **Waitagents()**
        $\backslash *$ *Conflict Resolution*
        count $\leftarrow$ 0
        turn $\leftarrow$ **false**
        **while inmessage() fromBreed**(Particles)
            **WaitAndReceive** dir **fromBreed**(Particles)**withProp** (position=mypos)
            count $\leftarrow$ count +1
            **if** abs(mydir-dir)= $180^\circ$ **then** turn $\leftarrow$ true
        **endwhile**
        **if** (count=1 **and** turn) **then left** $90^\circ$
    **forever**
```

Fig. 1. The cellular automata

Another class of computational models for molecular dynamics is that of *lattice gas automata* [12, 7] that model a fluid as a system of particles moving on the edges of a lattice, according to a set of rules. In general, these models assume that at most one particle enters a given node of the lattice, from a given direction. The particles move at discrete time steps, at a constant speed. Particles entering the same node at the same time step may collide: the rules to solve collisions guarantee the conservation of the total number of particles and of the angular moment. Several lattice gas automata have been proposed. The simplest one, based on the $HPPmodel$ [7], exploits a square lattice and it considers only a simple collision rule: when exactly two particles enters the same lattice vertex from opposite directions, a collision is detected and the particles change their direction by turning left 90° . In all other case, for instances when the directions of two particles are orthogonal or when more that two particles find themselves at the same vertex, the particles do not change their direction.

In Fig. 1, we show an *ORESPICS* agent implementing a single particle: all the particles are characterized by the same behaviour.

Each agent is characterized by a property, defining its coordinates on the screen: the value of the property is dynamically updated whenever the agents moves. Initially, each agent places itself at a lattice node and chooses a direction. The initial positioning must

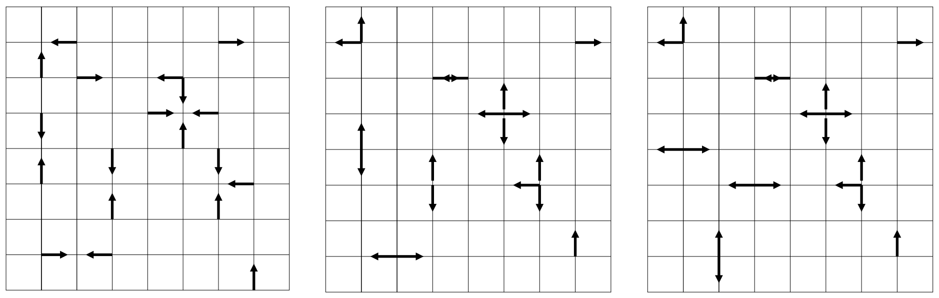

Fig. 2. Evolution of the cellular automata

guarantee that at most a particle is positioned at a vertex with a given direction. After the initialization phase, each agent iteratively executes three distinct phases. In the movement phase, it moves one step along its direction. In the second phase, all particles lying in the same node exchange their directions. The last step implements the interactions among particles: each particles collects all the incoming messages and detects any possible collision. If a collision is detected, each particle involved in the collision changes its direction, by a $90°$ left turn. These phases are separated by *synchronization barriers*, implemented through the *Waitagents()* primitive, to guarantee that each phase is initiated by any agent only when all others agents have completed the previous phase. For instance, the second barrier guarantees that each particle at node *N* starts collecting the messages only after any particle at *N* has sent its direction. In this way, all messages will be received.

ORESPICS supports a straightforward implementation of both the concurrent behaviour of the automata and the graphical interface. As far as concerns concurrency, the property mechanism is exploited in the second phase, to select the proper set of receivers, i.e. the set of particles lying at the same vertex. The graphical interface exploits the *LOGO* turtle graphics to show the evolution of the automata. We recall that, in *LOGO*, each turtle is characterized by its position, i.e. its coordinate on the screen and by its heading which is measured in degrees clockwise from North. Since two particles may collide if and only if their headings are directed against each other, the collision may be detected by checking if the absolute value of the difference of the headings' values is $180°$. Furthermore, each particle involved in a collision, simply turns towards its own left through $90°$.

Fig. 3 shows how the evolution of the system in the various phases can be monitored in *ORESPICS*. The left snapshot shows the initial situation, the central one the situation after the movement of the particles, the left one the situation after conflict resolution.

To model a realistic situation, some constraints have to be added to this simple model. For instance, consider a gas constrained in a container. The container is divided into two parts, separated by a wall with a hole. Initially, the gas particles are confined in the bottom part of the container, then they start flowing through the hole to the upper part till an equilibrium is reached. The behaviour of the particles bumping against the walls of the container is modeled by adding a new rule to the automata: a particle

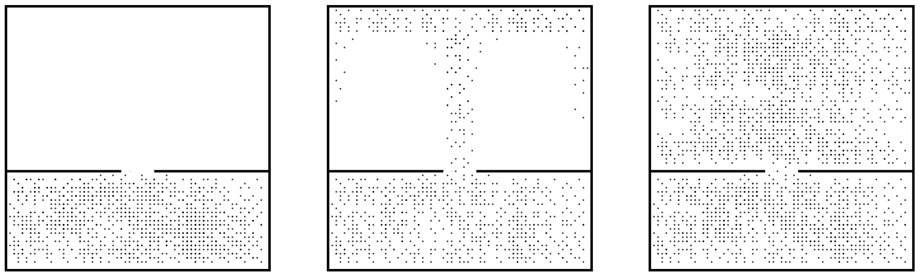

Fig. 3. Time evolution of a HPP gas

bumping against a wall, bounces back from where it came. The student can monitor the evolution of the system as shown in Fig. 3.

The code shown in Fig. 1 can be easily modified to implement more complex lattice models. For instance, in the *FHH model* [12] the particles moves along the edges of an hexagonal lattice. Even if a larger number of conflict situations have to be considered, each conflict can be simply implemented by changing the direction of a particle through $ORESPICS$ graphical commands.

4 Searching and Optimization Problems

Another class of computational science problems are search and optimization ones. The heuristic techniques usually exploited in this class, e.g. branch and bound, hill climbing, simulated annealing, genetic algorithm, often present a simple mathematical formulation and can be applied to real life problems. Hence, these problems are suitable for our target.

This section shows how the hill climbing search technique can be introduced through a real life problem. The problem is proposed in [3] as follows:

The recently discovered planetoid, Geometrica, has a most unusual surface. By all available observation, the surface can be modeled by the function $h(\theta, \rho)$:

$$h = 35000 sin(3\theta) sin(2\rho) + 9700 cos(10\theta) cos(20\rho) - 800 sin(25\theta + 0.03\pi) + 550 cos(\rho + 0.2\pi)$$

where h is the height above or below sea level, θ is the angle in the equatorial plane (defines longitude on earth), and ρ is the angle in the polar plane (defines latitude on Earth). A space-ship has landed on Geometrica. The main goal of the astronauts is to find the (θ, ρ) position of the highest point above the sea level on Geometrica surface.

To reach the topmost point of Geometrica, an astronaut may adopt an *hill climbing* strategy and move always uphill. Obviously, this does not guarantee that the highest point will be reached, because the astronaut can be stucked at the top of a low hill. To increase the probability of reaching the top of Geometrica, the national minister for space

```
Agent Hiker_i
    WaitAndReceive ($x_{min}$,$x_{max}$) from Master
    x ← random($x_{min}$,$x_{max}$)
    goto $(x, h(\overline{\theta}, x))$
    setimage Astronaut
    $\Delta \leftarrow \epsilon$
    stop ← false
    repeat
        $h \leftarrow h(\overline{\theta}, x)$
        if $h(\overline{\theta}, x + \Delta) > h$ then
            pendown
            $x \leftarrow (x + \Delta)$
            goto $(x, h(\overline{\theta}, x))$
        else
            if $h(\overline{\theta}, x - \Delta) > h$ then
                pendown
                $x \leftarrow (x - \Delta)$
                goto $(x, h(\overline{\theta}, x))$
            else
                setimage Flag
                SendAndnoWait h to Master
                stop ← true
            endif
        endif
    until stop
```

Fig. 4. Hill climbing

missions engages a large number of astronauts: each astronaut should start climbing at a different position, chosen randomly. The chief of the mission remains on space-ship and collects the results from the hikers, thus determining the global maximum.

It is worth noticing that this example could be exploited also to introduce *Monte Carlo numerical techniques* because the basis of these techniques is the use of a large set of randomly generated values used to define different, independent computations.

To simplify the $ORESPICS$'s implementation of the previous problem, we consider an equivalent $2D$ problem, by fixing the parameter $\theta = \overline{\theta}$ in the h function. Furthermore, each hiker performs a single exploration in its area. The resulting implementation is shown in Fig. 4. The code of the master is not shown because it is very simple. It partitions the area to be searched among the different astronauts, collects the results, and computes the maximum height. This corresponds to a static assignment of the tasks to the astronauts. Each hiker receives the coordinates of its area and puts itself in a position of the area chosen randomly. Then, it tries to move uphill: if this is not possible, it puts a flag on the top of the hill, to show it has been visited. This is implemented through the *setimage* command which changes the aspect of the hiker. At this point, the hiker can stop (as in Fig. 4) or continue the exploration choosing a new starting point.

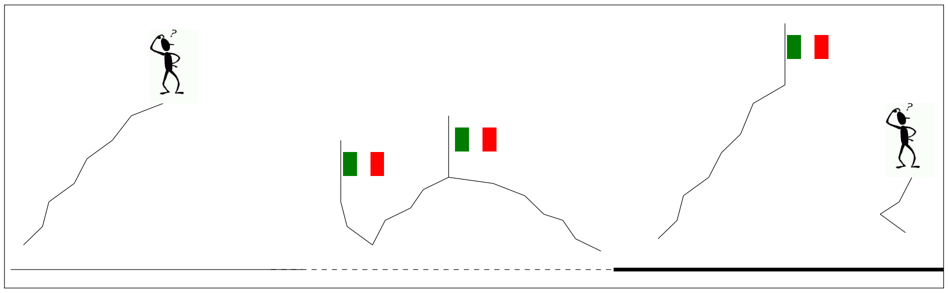

Fig. 5. Hill Climbing

The evolution of the search is shown in Fig. 5. Segments representing areas assigned to distinct hikers are represented through different line styles. We can note that some astronauts may have a longer way than others to reach their local peak, or some astronauts may climb faster because they are younger. For instance, in Fig. 5 the hiker assigned to the central area has completed its exploration, while the others are still climbing. This can be solved through a dynamic assignment. The master partitions the area into smaller segments and initially assign a segment to each astronaut. When an astronaut reaches a local peak, it asks for a new area. When no more areas to search are left, the master sends a termination message to each astronaut.

5 Related Work

Several proposals to introduce computational science in high level schools have been proposed. In [8] a set of proposals for the introduction of computational science education in high school curricula is presented. This paper discusses also how the introduction of supercomputers and high-performance computing methodology can be instrumental in getting the attention of the teenagers and attracting them to science. A presentation of more recent proposals can be found in [4].

Like *ORESPICS*, Starlogo [9] is a programming environment which is based on an extension of *LOGO*. This language has been proposed to program the behavior of decentralized systems. A student may program and control the behavior of hundreds of turtles. The world of the turtles is alive: it is composed of hundreds of patches that may be thought of as programmable entities but without movement. Turtles move parallel to one another and use the patches to exchange messages. Since the underlying concurrency paradigm is the shared memory one, this completely differentiates Starlogo from Orespics. The main goal of the Starlogo is the analysis and the simulation of the decentralized systems of the world, in contrast with more traditional models based on centralized ones. It helps users to realize that the individuals of a population may organize themselves without the presence of a centralized point of control.

Recently, several visual environments [1, 10] have been defined to support the development of parallel programs. These proposals do not define a language designed for didactic purposes, but provide support for editing and monitoring the execution of parallel programs written in C with calls to the PVM or MPI library. No particular support is provided to program real life situations: the user has to link some classical graphical library to the C program.

6 Conclusions

In this paper, we have presented *ORESPICS*, a programming environment supporting the learning of computational science in high school curricula. We are currently experimenting the system with a group of students and the first results are satisfactory. Problems from different areas, i.e. cellular automata programming, genetic programming, simulated annealing, have been programmed through *ORESPICS*. The system has also been adopted to introduce some classical computational science algorithms, like algorithms from matrix algebra, or graph algorithms. As an example, we have defined a set of animations to introduce systolic algorithms for matrix manipulation, like matrix multiplication, transposition and transitive closure computation. Currently, we are improving the system in several directions. A richer set of functionalities to monitor the execution of the programs will be defined. Furthermore, we are defining a library, including a set of complex visualization techniques through *ORESPICS* basic constructs. Finally, we plan to extend the language with constructs to support the shared memory paradigm as well.

References

1. A.Beguelin, J.Dongarra, A.Geist, and V.Sunderam. Visualization and debugging in a heterogeneous environment. *IEEE Computer*, 26(6), June 1993.
2. B.Harvey. *Computer Science Logo style*. MIT press, 1997.
3. B.Wilkinson and M.Allen. *Parallel Programming techniques and applications using networked workstations and parallel computers*. Prentice Hall, 1999.
4. C.Swanson. Computational science education. In *www.sgi.com/education/whitepaper.dir/*.
5. G.Capretti, M.R.Lagana', and L.Ricci. Learning concurrent programming: a constructionist approach. *Parallel Computing Tecnologies, PaCT*, 662:200–206, September 1999.
6. G.Capretti, M.R.Lagana', L.Ricci, P.Castellucci, and S.Puri. Orespics: a friendly environment to learn cluster programming. *IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID 2001*, pages 498–505, May 2001.
7. J.Hardy, Y.Pomeau, and O.de Pazzis. Time evolution of two-dimensional model system. Invariant states and time correlation functions. *Jour.Math.Phys.*, 14:1746–1759, 1973.
8. M.Cohen, M.Foster, D.Kratzer, P.Malone, and A.Solem. Get high school students hooked on science with a challange. In *ACM 23 Tech. Symp. on Computer Science Education*, pages 240–245, 1992.
9. M.Resnick. *Turtles, termites and traffi c jam: exploration in massively paralle micro-world*. MIT Press, 1990.
10. P.Kacsuk and al. A graphical development and debugging environment for parallel programming. *Parallel Computing Journal*, 22(13):747–770, February 1997.
11. P.Pacheco. *Parallel programming with MPI*. Morgan Kauffmann, 1997.
12. U.Frish, B.Harlacher, and Y.Pomeau. Lattice-gas automata for the navier-stokes equation. *Physical Review Letters*, 56(14):1505–1508, 1986.
13. V.Sunderam. PVM: a framework for parallel distributed computing. *Concurrency;Practice and experience*, 2(4):315–339, 1990.