Computational Science In High Schools: Defining Curricula And Environments

Paolo Mori and Laura Ricci

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

Abstract

The paper presents a new approach for the introduction of computational science into high level school curricula. It also discusses a set of real life problems that are appropriate for these curricula because they can be described through simple models. The computer based simulation of these systems require an ad hoc environment, including a programming language, suitable for this target of age. The paper proposes a new environment, the *ORESPICS* environment, including a new programming language. The sequential part of the language integrates the classical imperative constructs with a simple set of graphical primitives, mostly taken from the *Logo* language. The concurrent part of the language is based on the message passing paradigm. The solutions of some classical problems through *ORESPICS* are shown.

Key words: Computational Science; Parallel Programming; Didactic Language; Message Passing;

1 Introduction

Computational science is a new interdisciplinary research area which applies concepts and techniques from mathematics and computer science to real life problems. Computational science has introduced a new methodology to investigate real life problems because, 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 and its evolution can be monitored by visualizing the program output. 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 high school curricula have also been presented [5,9]. The goal of these proposals is to increase

reprinted from Future Generation Computer Systems 19(8) (2003) 1349–1361

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 adoption of the computers makes the learning even more appealing. Yet, the introduction of computational science into high school curricula and/or undergraduate courses requires to settle some issues. First of all, the basic mathematical skills to support a first training in computational science have to be defined. These skills should support the development of models for a minimal, yet significative, kernel of applications. The applications in this kernel should be characterized either by a simple mathematical model 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. 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* [12,15], or *OpenMp* are not suitable, because of their complexity. As a matter of fact, these libraries include several semantic equivalent primitives differing only because of their implementation. A didactic environment should introduce a single construct for each different concept of the language. Furthermore, the didactic language should support a simple *graphical interface* as well, so that the students can monitor the behavior of the concurrent activities directly through an user friendly interface. 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 have been introduced in [6,7]. Sect. 2 introduces the main constructs of the language and briefly describes the *ORESPICS* environment. To describe how *ORESPICS* can be exploited in an introductory course, Sect. 3 introduces some simple, yet significative problems and it shows their *ORESPICS* solutions. The environment is fully described through the development of one problem. Sect. 4 presents some related works. Sect. 5 draws some conclusions.

2 The ORESPICS Environment

The *ORESPICS* environment has been defined starting from a programming language, *ORESPICS-PL*, to support the development of simple models of real systems. *ORESPICS-PL* is an imperative language and its concurrent part is based upon the *message passing* paradigm [4].
An *ORESPICS-PL* program includes a set of *agents*, interacting through messages exchanged within a *microworld*. Each agent may be paired with a distinct code, while the same code may be associated with a *breed of agents* characterized by the same behavior. In this case, the agents are programmed according to a *SPMD* programming style. The development of an *ORESPICS* application requires an initialization phase, where the user defines the kind of the agent, i.e. single, belonging to a breed,...,etc, pairs each agent with a set of predefined images and defines the main features of the microworld where the agents interact.
The sequential part of *ORESPICS* integrates the traditional imperative constructs (repeat, while, if,...), with the turtle primitives of the Logo language [2]. These primitives are exploited by the agents to move inside the screen. Both turtle-relative commands, i.e. command to move forward and backward and to turn left and right, and cartesian-style graphics commands are defined. The sequential part of the language includes also a set of commands to choose, during the execution, the aspect of an agent from a set of predefined images and the set of sounds emitted by the agent. All elementary data types (integer, boolean,..) and the list data structure are supported by the language.
Agents 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 send commands are:

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

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

Buffer management for buffered communications is delegated to the run time sup-

port. The semantics of the asynchronous send S guarantees that if the corresponding receive R is executed after the completion of S, i.e. after S has stored its message in the system buffer, R gets the message. *ORESPICS-PL* does not support any other communication modes. This is consistent with the choice to include only constructs corresponding 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 but are not required to introduce computational science. 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

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

Inmessage(*agent*)

supports the polling of incoming messages, without actually receiving them. *Inmessage(A)* returns true if there is at least an incoming message from agent *A*. Then, one 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 two versions of the broadcast communication, respectively, synchronous and asynchronous. The corresponding sends are, respectively

SendAllAndWait()

SendAllAndnoWait()

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

Example 2.1 *Let us consider a system including two kind of particles, respectively, the α and the β ones . The behavior of the system is defined by a single rule. If at least 2/3 of the particles belong to the same kind, these particles "dominate" the system and "expel" the other ones. The resulting system is* stable.
A microworld including two breeds of agents, the $alpha$ and the $beta$ ones, may be exploited to describe the system. Each agent notifies its existence to the other ones through a message. This is implemented by a broadcast send. Then, each agent determines the number of particles of each breed by counting the number of messages received from agents of that breed. At first, each agent tests the presence of messages incoming from a breed, for instance from the alpha *breed*

Inmessage fromBreed *(alpha)*

and, if some message is present, it gets the message from the agent of the corresponding breed:

WaitAndReceiveAny *()* **fromBreed***(alpha)*

this is iterated until no more messages from the considered breed exist. This procedure is iterated for each breed. Then, each expelled agent leaves the system by moving outside the screen. ♢

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*. Afterwords the value is dynamically updated.

Example 2.2 *Let us consider the example 2.1 again, and suppose that the particles of the system are randomly partitioned into a sequence of groups. Let us suppose that initially all the groups but the first are stable. When a particle is expelled by a group, it moves to the next one. In this way, a group that is initially stable may later become an unstable one.*
To distinguish among agents belonging to distinct groups, agents belonging to the same breed are further identified by a property defining the identifier of the group the agent belongs to. The property is defined in the declarative part of the agent

Property *Group*

where Group is the name of the property. The value of the property is initialized with the identifier of the group, that is chosen randomly by the agent when execution starts. This value is dynamically updated*, when an agent moves from one group to the next one. The wait command is modified as follows:*

WaitAndReceive *()* **fromBreed***(alpha)* **withProp** *(Group=Mygroup)* ♢

The set of collective communications includes the synchronization barrier

Waitagents()

The following example shows how the synchronization barrier can be exploited to guarantee the correctness of previous examples.

Example 2.3 *Consider the systems described in the previous examples again. In both cases we have to guarantee that any message has been inserted into the system buffer, before any receive is executed. This can be guaranteed if the send phase and the receive one are separated by a global* synchronization barrier. ♢

3 Didactic Strategies

This section shows how *ORESPICS* should be exploited in introductory computational science curricula. These curricula should propose the modeling of a set of phenomena characterized by a simple mathematical model or by a complex model that can be simplified without loosing the connections with the real phenomenon. This section discusses a set of systems characterized by these models and shows how *ORESPICS* supports the modeling of these systems.

A classical area of computational science is related to the development of models for fluid or gas dynamics. Computational fluid dynamics describes physical phenomena through partial differential equations, 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. However, the resulting models are still close 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 computed as the average of those of the four neighboring points. In *ORESPICS*, it is rather simple to define a data parallel concurrent program, where the agents correspond to the points of the sheets. The program could display the temperature of the sheet by pairing distinct temperature values with distinct colors. Section 3.1 shows a cellular automata simulating the dynamic behavior of a gas.

Another interesting class of computational science problems consists in 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. Section 3.2 discusses the solution of a searching problem.

A field that has recently received considerable attention is that of *artificial life*. This research area attempts to recreate biological and social phenomena within the

computer. In this case, several life-like behaviors such as growth, adaptation, reproduction, socialization, and death are simulated through the computer. In this case, it is interesting to study how social structures and behaviors arise from the interactions of a large set of individuals. [10] presents a set of classical examples, like the behavior of termites gathering wood chips into piles, the movement of cars on a highway and forming traffic jam, the bird flocks and so on. All these systems are characterized by simple models and can be easily modeled in *ORESPICS*. Section 3.3 shows how *ORESPICS* can be exploited to model the behavior of a simple social system.

3.1 A Cellular Automata

An important class of computational models for molecular dynamics is that of *lattice gas automata* [14,8] 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 $HPPmodel$ [8], exploits a square lattice and it considers only a simple collision rule: when exactly two particle 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, the particles do not change their direction.
Fig. 1 shows an *ORESPICS* agent implementing a single particle: all the particles share the same behavior. 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 guarantees 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 particle 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 before analysing collisions.
ORESPICS supports a straightforward implementation of both the concurrent behavior of the automata and the graphical interface. As far as concerns concurrency,

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

Fig. 1. The cellular automata

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 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 heading values is $180°$. Furthermore, each particle involved in a collision, simply turns toward its own left through $90°$.

Fig. 2 shows how the evolution of the system in the various phases can be mon-

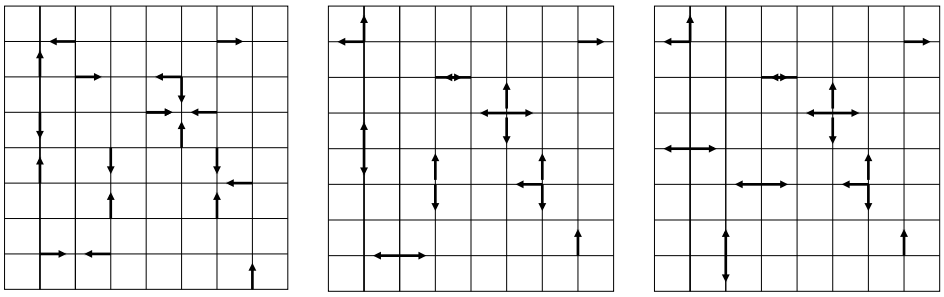

Fig. 2. Evolution of the cellular automata

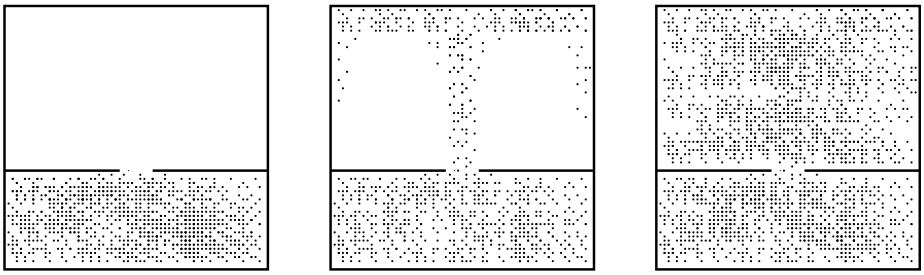

Fig. 3. Time evolution of a HPP gas

itored 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 behavior of the particles bumping against the walls of the container is modeled by adding a new rule to the automata: a particle 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* [14] 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.

3.2 Searching and Optimization Problems

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 $h(\theta, \rho)$*:*

Agent Hiker$_i$

WaitAndReceive (x$_{min}$,x$_{max}$) **from** Master

x $\leftarrow$ random(x$_{min}$,x$_{max}$)

goto $(x, h(\overline{\theta}, x))$

setimage Astronaut $\Delta \leftarrow \epsilon$ stop $\leftarrow$ 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 $\leftarrow$ true

endif

endif

until stop

Fig. 4. Hill climbing: the code

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

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*

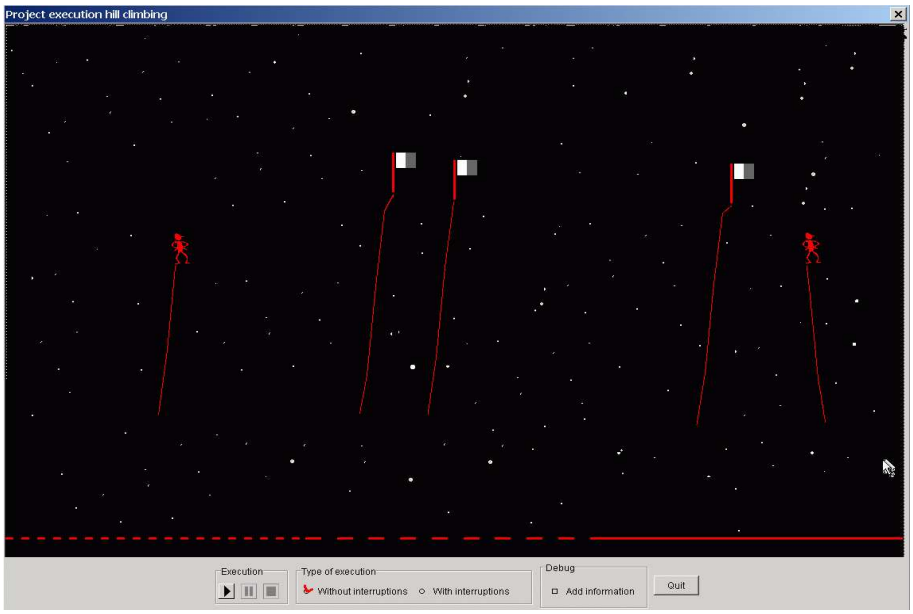

Fig. 5. Hill Climbing

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 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 the 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 these techniques exploit a large set of randomly generated values to define alternative, 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 master simply partitions the area to be explored 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 at random. 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 by choosing a new starting point. The evolution of the microworld can be monitored in the execution window, as shown in Fig. 5. In this figure, each astronaut performs two explorations before stopping. 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. In Fig. 5, the hiker assigned to the central area has com-

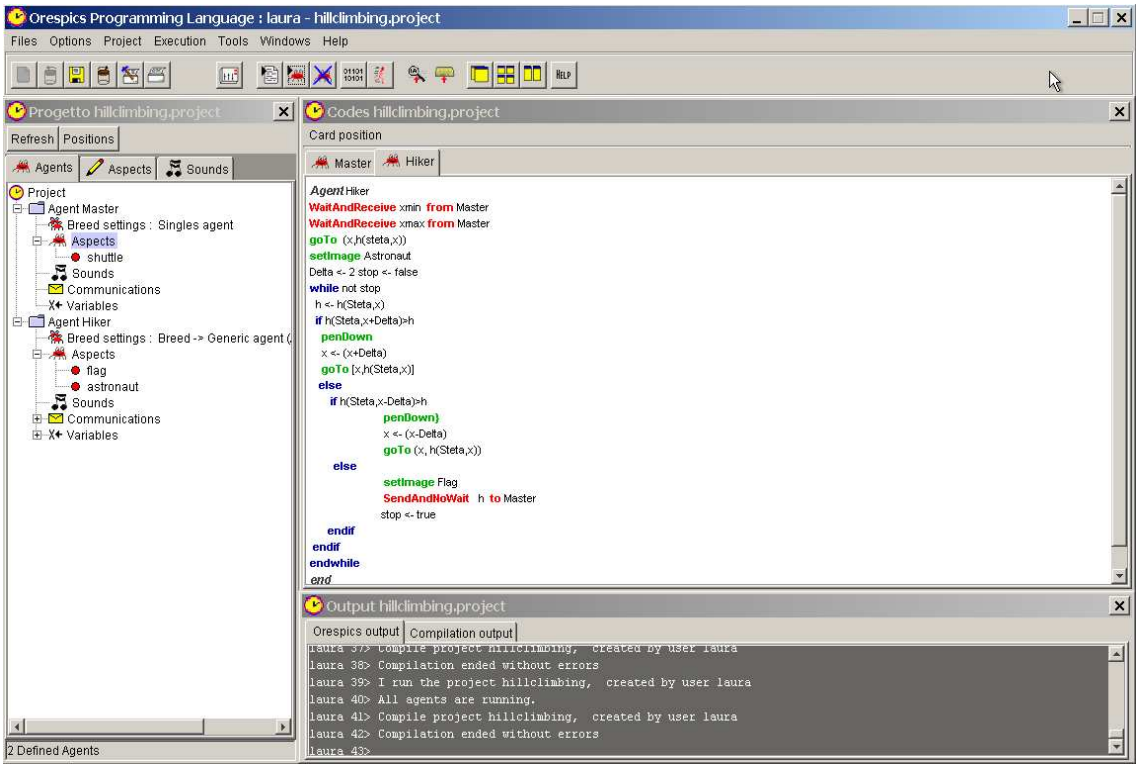

Fig. 6. The Orespics Environment

pleted its exploration, while the others are still climbing. The resulting load unbalance can be solved through a dynamic assignment. The master partitions the area into smaller segments and initially assigns a segment to each astronaut. When an astronaut reaches a local peak, it asks for a new area to be explored. When no more areas are available, the master sends a termination message to each astronaut. Note that the window includes buttons to start the execution, to stop it and a further button displays debug information.

Fig. 6 shows the main features of the *ORESPICS* development environment. The toolbar shown on the top of the figure includes buttons to create a new project, to define the agents, to compile and execute a project and other utilities. The characteristics of each agent are defined in the *AgentBrowser* window, the leftmost window in Fig. 6. The agent browser includes a folder for each agent of the microworld, i.e. the master and the hiker agent. Initially, each folder is empty and is gradually updated as soon as a new feature of the agent is defined. For instance, the agent browser shows that two agents have been created, the Master Agent and the Hiker one, and summarizes the main characteristics of each agent. The master is a single agent, while the hiker agent belongs to a breed, i.e. it is a generic agent. Even if *ORESPICS* associates a default icon with any agent upon its creation, the user can modify the icon or choose a new one. Furthermore, several icons can be defined for the same agent. For instance, a single icon, named *shuttle* is associated with the Master agent. Two different icons, instead, are associated with the Hiker Agent. The *astronaut* icon describes the hiker while it is climbing. The *flag* icon, instead, is displayed when a local maximum is found. Each icon is uniquely identified by a

name. This name may be exploited by the agent to select the proper image during the simulation, through the *setimage* command. *ORESPICS* supports animation as well. For instance, an animation can be defined to show the hiker jumping when it reaches the top of an hill. Each agent may be also paired with a set of sounds to make the microworld more realistic. For instance, the effort made by the hiker in climbing the hill can be modeled by a sound which mimics a labored breath.
The microworld definition window can be displayed by selecting the Position button in the agent Browser. It is possible to choose the size of the microworld, a background image and music. Furthermore, 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 defined by the system.
The rightmost windows in Fig. 6 are the Code definition window (the top one) and the Project window (the bottom one). The Code definition window includes a folder for each agent, identified by the standard icon of the agent. The figure shows the code of the Hiker agent. Different primitives of *ORESPICS-PL* are displayed by exploiting different colors and these colors can be defined in the *ORESPICS* preferences.
Finally, the Project window displays the message produced by the system. For instance, the window is exploited to display the output of the compiler. If the compilation is not successful, the errors are displayed in a distinct folder for each agent.

3.3 Social Systems Modeling

Let us consider again the Example 2.2. The example is exploited in [13] to describe a social system. Let us suppose that the agents are no longer particles in a physics system, but people at a cocktail party. The $alpha$ particles represent men, whilst the $beta$ particles represents women. The behavior of each individual is the same of the particles in 2.2. The resulting system models a situation where if one group has a majority of one gender, people of the other gender are not at ease and move to a neighboring group. Obviously people are not expelled, but leave the group voluntarily. The final effect is that most groups in the cocktail party, at the end, include only men or only women.

4 Related Work

[16] observes that concurrency can be introduced in introductory courses on programming, since concurrency is not harder than sequential computing, if it is introduced by a proper environment. The proposed environment integrates the CSP programming model with the JAVA language.
The introduction of computational science in high level schools has previously been proposed. In [9] 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 [5].
Like *ORESPICS*, Starlogo [10] 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 and it consists of hundreds of patches that may be seen 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 a centralized point of control.
Recently, several visual environments [1,11] 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. To this purpose, the user has to link some classical graphical library to the C program.

5 Conclusions

This paper has 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.A.R.Hoare. Communicating Sequential Processes. In *ACM Communications*, volume 21, pages 666–677, 1978.

[5] C.Swanson. Computational science education. In *www.sgi.com/education/whitepaper.dir/*.

[6] G.Capretti, M.R.Lagana', and L.Ricci. Learning concurrent programming: a constructionist approach. *Parallel Computing Tecnologies, PaCT*, 662:200–206, September 1999.

[7] 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.

[8] J.Hardy, Y.Pomeau, and O.de Pazzis. Time Evolution of Two-Dimensional Model System. Invariant States and Time Correlation Functions. *Journal Mathematics Physics*, 14:1746–1759, 1973.

[9] 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. Symposium on Computer Science Education*, pages 240–245, 1992.

[10] M.Resnick. *Turtles, termites and traffi c jam: exploration in massively paralle micro-world*. MIT Press, 1990.

[11] P.Kacsuk and al. A Graphical Development And Debugging Environment For Parallel Programming. *Parallel Computing Journal*, 22(13):747–770, February 1997.

[12] P.Pacheco. *Parallel Programming with MPI*. Morgan Kauffmann, 1997.

[13] Mitchel Resnick and U.Wilensky. Diving into Complexity: Developing Probabilistic Decentralized Thinking through Role-Playing Activities. *Journal of Learning Science*, 7(2).

[14] U.Frish, B.Harlacher, and Y.Pomeau. Lattice-Gas Automata for the Navier-Stokes Equation. *Physical Review Letters*, 56(14):1505–1508, 1986.

[15] V.Sunderam. PVM: a Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.

[16] P. H. Welch. Process Oriented Design For JAVA: Concurrency for all. In *International Conference on Computational Science*, volume 2330, pages 687–687. Keynote Tutorial.