Proposal of an Interface to support Cooperative Work in a Distributed Systems Environment *

Guillermo de Jesús Hoyos-Rivera
Maestría en Inteligencia Artificial
Universidad Veracruzana-LANIA, A.C.
Sebastián Camacho # 5
Xalapa, Veracruz
91000 MEXICO
e-mail: ghoyos@mia.uv.mx

Víctor Germán Sánchez-Arias
LANIA, A.C.
Enrique C. Rébsamen # 80
colonia Isleta
Xalapa, Veracruz
91090 MEXICO
e-mail: vsanchez@xalapa.lania.mx

Abstract

In the work presented we propose the definition, design of a *Cooperative Work Model* (*CWM*) and a *Cooperative Work Language* (*CWL*), and the implementation of an interface to support *Cooperative Work* in a *Distributed Systems Environment*, called *ICW*[1]. Up to now this interface is already working, and we have already defined several characteristics it should have in order to enhance it and make it more effective at work.

Keywords: Cooperation, Parallel, Distributed, Model, Language, Communication.

1 Introduction

Cooperative work implies the coordination of several tasks during their execution, all of them sharing a common goal, cooperation rules that coordinate the actions, and communication primitives that make interaction possible. We take this idea as the basis to pose the desirable characteristics an interface of this kind should have.

1.1 Inspiration for this work

There are two facts that we have taken as the main inspiration for this work.

The first one is that many problems can be expressed as cooperative modules. Taking this assumption into account, we can make all the modules of the solution to run in parallel, doing execution faster.

The second one is that *Message Passing Computation* (*MPC*) is becoming more accessible everyday. It is not necessary anymore to buy a mainframe (like a *Cray*), or any expensive special hardware (like the *Transputers*) to get large computational power. At present time many universities and enterprises have computer networks installed in their buildings. A computer network can be viewed as a set of loosely coupled processors[2], so almost any computer network can be viewed and used as a large single computational resource, provided that the appropriate software is installed. So every university or enterprise that has a computer network has, potentially, a *virtual parallel mainframe*.

According to these ideas we have defined a *Cooperative Work Model* (*CWM*) **[Sánchez-Arias, 1996]**, and a *Cooperative Work Language* (*CWL*) **[Sánchez-Arias-2, 1996]**. Their main goal is to make the expression of *Cooperative Work* (*CW*), its parallelism and distribution easier.

CWM and *CWL* are inspired upon the definition of *CSP* **[Hoare, 1978]**, and the basic notions of *processes* and *pipes* used in *Unix* **[Ritchie and Thompson, 1984]**.

2 Cooperative Work Model

Our *CWM* defines cooperative work as a set of interrelated tasks, arranged in a hierarchical structure. These tasks will be run in parallel in a distributed environment, and will have

[1] *ICW* stands for *Interface for Cooperative Work*.

[2] Loosely coupled processors do not share memory. Information is shared by explicit message passing.

* This work is being developed as a joint work by Maestría en Inteligencia Artificial of the Universidad Veracruzana, and Laboratorio Nacional de Informática Avanzada, LANIA, A.C.
This project is part of a major project called "*Modelo y Arquitectura de Cómputo Paralelo, Distribuido y Cooperativo*" which is supported by CONACyT under contract No. 1124P-A

the capability to communicate among them by explicit message passing. The execution of the tasks will be ruled by predefined *cooperation rules*, and communication will be taken into effect by using some predefined communication primitives.

Tasks are to be executed in any of the hosts of the whole system. At any moment of the execution time of a *CW* program, one particular host should be capable to execute one or more tasks.

As a general overview, we can see the tasks arranged as sets. In *Figure # 1* we can appreciate a typical hierarchy represented in this way. Task *TC* consists of its own execution and the execution of tasks *TR* and *TP*. Task *TR* consists of its own execution and the execution of *n* copies of the task *A*. Finally *TP* consists of its own execution and the execution of tasks *X* in host *Sp*, *Y* in host *Sr*, *Z* in host *Sz* and *W* in a particular unknown host expressed by the question mark (?). Tasks *X*, *Y*, *Z*, *W* and the *n* copies of task *A* depend upon their own execution only, that is, they do not contain other tasks into them.

Let us analyze the special case. *W* should be executed in a particular host, however it is unknown which one it is, so this case will make necessary to find the location where the executable program resides in order to execute it.

As an example let us imagine we have the set of hosts *Sp*, *Sr*, *Sz*, *Sw* and *Ss*, and the *CW* expressed in *Figure # 1*. As *TC*, *TR*, *TP* and all the *n* copies of *A* (let us imagine *n=6*) do no have an explicit place where they should be executed, they can be executed in any host, meanwhile *X*, *Y*, *Z* and *W* should be executed in a particular host each one. In the special case of task *W*, we will assume that after calling the locator process, it was determined that the corresponding executable program resides in *Sw*.

Given the previous assumptions, we get the execution as illustrated in *Figure # 2*.

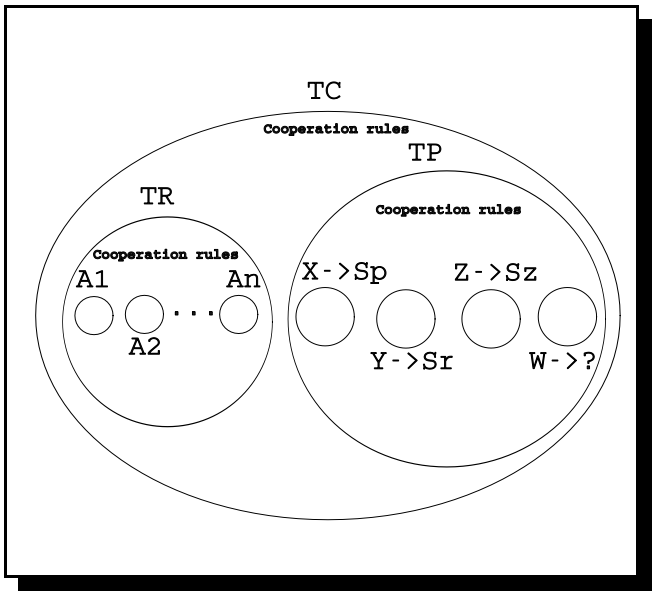

Figure # 1

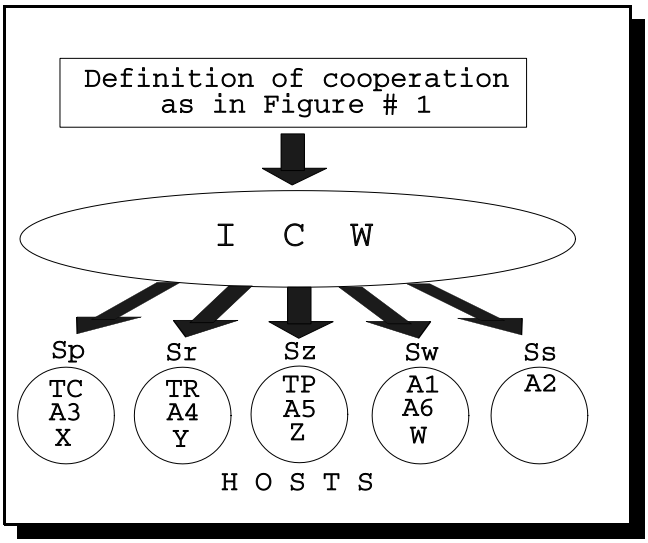

Figure # 2

2.1 Tasks

The minimal work unit of the cooperative work will be the task. In our approach we consider a task as a completely executable program following the standards and binary format of the particular operating system it was created for.

By definition, any task should have the following characteristics:

a). any task executed in the interface starts and ends its execution at some moment in time.

b). when execution is started, the task should optionally receive some input parameters,

c). any particular task does not share memory with any other task,

d). the only way to share information with other tasks is by explicit message passing.

Any task may be classified according to four different criteria: **(1)** the type of the task, **(2)** the level of the task into the hierarchy, **(3)** the possible place where it is to be executed, and **(4)** the number of copies of the program to be executed concurrently.

Types of tasks

According to this criteria, a task can be a *generic task*, or a *CW task*. A generic task is any general-purpose program that can be executed by writing the command name in the operating system prompt. Examples of this kind of tasks are `/bin/ls`, `/bin/cp`, `$HOME/bin/print`, etc. A *CW task* is a compiled executable program explicitly written for our interface. The standards to write this last kind of tasks will be explained later in this document.

Level in the hierarchy

According to this criteria, any task can be of any of two types: *atomic* or *composed*.

An atomic task will be any executable program. It can be

either, a *CW task*, or a generic task. This kind of task will consist only of its own execution. In *Figure # 1*, all the copies of task *A*, and tasks *X*, *Y*, *Z* and *W*, are atomic tasks.

A composed task can only be a *CW task*. This kind of task will be the one that consists of its own execution, and the execution of one or more atomic or composed tasks spawned by it. A composed task has its own executable code.

The tasks executed by a composed task will be called *members* of that task, and the composed task executing other tasks will be called the *caller*.

In *Figure # 1*, tasks *TC*, *TR* and *TP* are composed tasks.

Any task, atomic or composed, will be capable to communicate with other tasks according to the *Communication Rank*. This issue will be explained later in this document.

Place of execution

According to the possible place where the tasks can be executed, they can be classified as *explicitly located* or *not explicitly located* tasks.

A *not explicitly located* task is the one that can be executed in any host of the *virtual parallel computer*. It can be possible by two means: **(1)** there is a copy of the executable program in every host of the system or **(2)** the executable program is accessible through a *NFS* system.

The place where this kind of tasks are to be executed will be determined dynamically at runtime.

An *explicitly located task* will be executed always in the same host, or a set of hosts of the same architecture. This can be due to any of three reasons: **(1)** the executable program does only exist in one host of the system, **(2)** the executable program was compiled for a particular architecture or **(3)** it is desirable to execute the program in some particular host due to some particular reason, as could be the speed of execution.

When using *explicitly located tasks* it will be necessary to specify the host or the architecture where it should be executed.

There is a special case: a task can be *explicitly located*, but the host where it resides is unknown. Then it will use a *locator*, in order to find, dynamically at runtime, the host where the executable program resides, and then execute it.

Number of copies in concurrent execution

The interface should have the capability to execute several copies of any task. Any *replicated task*, as they will be called in future references, can be *explicitly located* or *not explicitly located*. If they are *explicitly located*, then all the copies of the task will be executed concurrently in the same host or, if it was specified an architecture, the copies will be executed in the subset of hosts of the specified architecture. Otherwise, each copy will be executed in any host of the system.

2.2 Cooperation rules

Every task which is member of a composed task will have associated a cooperation rule to control its execution. We have already defined two basic rules: `SYNCP` and `ASYNCP`

`SYNCP`

`SYNCP` stands for *Synchronous Parallelism*. When some tasks are ruled by `SYNCP` all of them are started at the same time and keep working concurrently. The caller task will not end until all the member tasks have terminated, however, any member task does not depend on its caller task, nor on the tasks that were spawned at the same time than it.
As an example, if we have the next expression:

```
t:SYNCP[t1, t2];
```

the caller task will be `t`, and the member tasks will be `t1` and `t2`. The caller task will not end until all the member tasks have terminated, in this particular case, `t1` and `t2`.
These ideas are illustrated in *Figure # 3*.

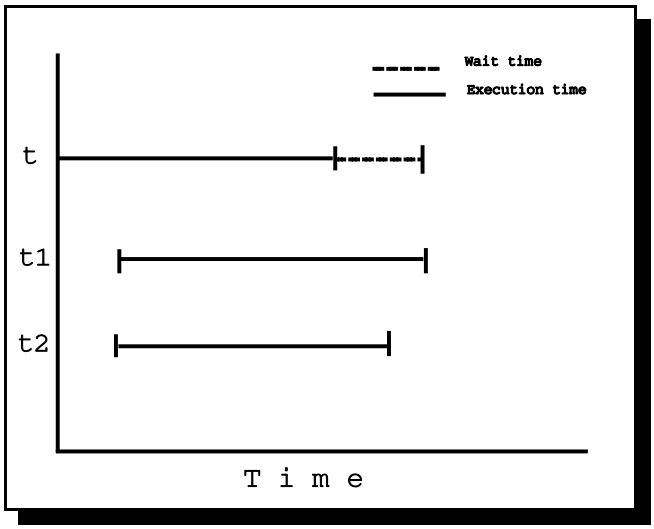

Figure # 3

`ASYNCP`

`ASYNCP` stands for *Asynchronous Parallelism*. This rule operates almost in the same way than `SYNCP`, but in this case the caller task does not have to wait for all the member tasks to terminate. Any task, including the caller task can terminate without having to wait for the termination of any other task working with this rule.

The same example posed for the last rule is useful for this one, but with the difference that `t` will be able to terminate independently of the termination time of tasks `t1` and `t2`.

As can be seen in *Figure # 4*, `t` does not have to wait for `t1` and `t2` to be finished.

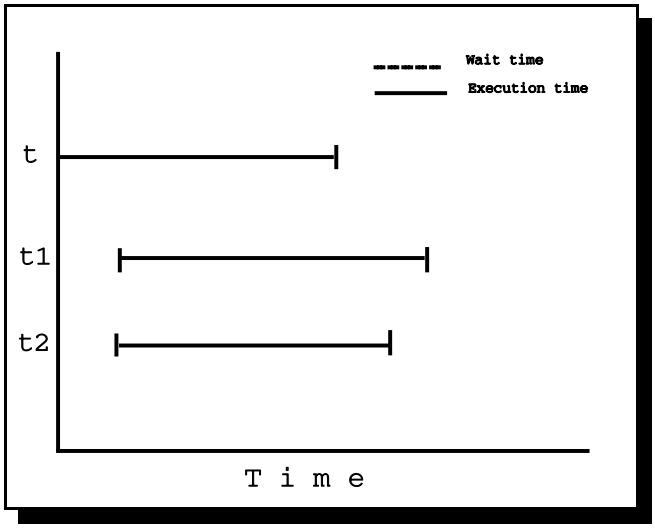

Figure # 4

2.3 Communications

All *CW tasks* in a *CWL* program have a *Communication Rank*. The Communication Rank specifies which tasks a determined task will be able to communicate with. The *Communication Rank* of a task *T* running into *ICW* is formed by:

* The composed task that spawned *T*
* All the tasks spawned at the same time than *T*
* All the tasks spawned by *T*

Taking again the model in *Figure # 1*. Tasks *X*, *Y*, *Z* and *W* will only be able to communicate with task *TP* and with each other, but not with tasks *TR* nor *TC*, which are outside the *Communication Rank*. The only tasks that can communicate with task *TC* are *TR* and *TP*.

When sending a message, the task involved in this operation does not have to wait for it to be received. The message is allocated in a buffer, and when the target process is ready to receive, the message is taken from that buffer.

If when receiving a message there is no appropriate buffer containing information addressed to the task involved in this operation, it must wait until a message arrives.

3 The implementation

The developed interface is the medium by which it will be able to map the model posed to an operating distributed environment. Any task is an executable program residing in one or several hosts of the system, or accessible through a *NFS* system.

The interface will provide all the necessary mechanisms to distribute, replicate and locate any task to be executed. The execution of the tasks will be monitored and controlled trying to guarantee a complete execution of all the programs or, if a failure is found, the interface will report it. The goal of this last characteristic is to make the debugging process easier.

According to the previously explained definitions, we try now to define the operating instrument to take cooperative work into effect. The whole development is be called *Interface for Cooperative Work* (*ICW*), and will be constructed out of three components: **(1)** a base software on which it will be constructed our interface, **(2)** a proprietary language called *Cooperative Work Language* (*CWL*) to specify the cooperation relation between tasks, **(3)** a set of standards for writing *CW tasks*. The interface will have a control strategy defined into itself.

3.1 The base software

In order to decide the tools we should use to implement our interface we analyzed several alternatives. The main two were *LAM*[3] (*Local Area Multicomputer*) and *PVM* **[Geist et al., 1995]** (*Parallel Virtual Machine*). After studying both alternatives, we decided to use *PVM*.

PVM is a portable message-passing programming system, designed to link separate hosts to form a *virtual machine*, which is a single, manageable computing resource.

The main two reasons to select *PVM* over *LAM* are: **(1)** its native capability to spawn tasks and **(2)** its ability to be installed and run on different architectures.

As it may be supposed, it is strictly necessary to have installed *PVM* in order to be able to use this interface.

3.2 Cooperative Work Language

A typical program written in *CWL* has six sections:

* Architectures declaration (`ARCHS`)
* Hosts declaration (`HOSTS`)
* Tasks declaration (`TASKS`)
* Generic tasks declaration (`GTASKS`)
* Root task declaration (`ROOT`)
* Cooperative Work declaration (`CW`)

The first two sections (`ARCHS` and `HOSTS`) contain the list of names of architectures and hosts respectively. Both, architectures and hosts, declared in these sections must be configured in the virtual computer in order to be accepted as valid, although it is not necessary to have listed all the hosts and architectures configured in the *virtual computer* in these sections. It is not mandatory have these two sections. Then can be omitted if there will not be *explicitly located tasks*.

The `TASKS` section contains the declaration of all the tasks that are to be involved in the execution.

The existence verification of this kind of tasks will be done at runtime.

In the `GTASKS` section will be listed all the generic tasks (see previous definition). The interface will not accept any generic task to be declared and used as a composed task. The existence verification of this kind of tasks will be done

[3] *LAM* is an implementation of the *MPI* **[MPIF, 1994]** (*Message Passing Interface*) standard.

at runtime. It is not mandatory to have a GTASKS declaration.

In both sections TASKS and GTASKS it will be possible to indicate that a task is going to be executed in a particular host or a particular architecture. To indicate that a task is to be executed in a particular host it should be used the operator ->, and to indicate that it is a particular architecture the operator =>.

If there is a declaration of a task in a particular host or architecture, the name of the host or the architecture will be validated with the hosts or architecture declaration respectively.

The ROOT declaration indicates which task will be the one that will contain, directly or indirectly, the other tasks. In other words, the task declared as ROOT will be the superset of the whole hierarchy. The ROOT task must be declared previously in the TASKS declaration. If not, or if it is declared in the GTASKS declaration, it will not be accepted.

Finally, the CW declaration indicates the structure of the execution, and the rules that will control the relation between tasks. Every task listed in the declaration of the cooperative work will be validated against the tasks declared in the TASKS and GTASKS sections.

Any task declared in the GTASKS section will not be accepted to be a composed task.

A typical *CWL* program looks like the next example:

```
ARCHS: SUN4,LINUX;
HOSTS: afrodita,hefestos,cronos;
TASKS:
test,test1->cronos,test2,test3,test4,tes
t5,test6=>LINUX;
GTASKS: test7;
ROOT: test;
CW {
test: SYNCP[test1,test2];
test2: PIPE[test3,test4];
test3: ASYNCP[test5(3)];
test4: SYNCP[test6(10),test7];
}
```

3.3 Writing *CW* tasks

As previously mentioned, *PVM* applications can be programmed in either *C*, *C++* or *Fortran*.

By now we have developed the *C* version only, however it is open to a future implementor the implementation of the corresponding interfaces for *C++*, *Fortran*, or any other language.

Programs for the interface are generic *C* programs with the only peculiarity that they must be compiled with the preprocessor directive

```
#include "ICW.h"
```

This header file contains all the necessary definitions to start working into the interface, to control the execution and all the functions to communicate between tasks. Into this file there are two main special functions:

```
void ICW_init(int argc, char *argv[])
```

and

```
void ICW_end(void).
```

ICW_init() must be called as the first executable instruction of the function main() in every *CW task*. This function will receive from the caller the set of tasks it should execute, and then execute them, sending to every one the appropriate information for them to be able to start working. It must be called with the standard arguments received by the C program since that information is used by the interface to determine the operating environment.

The last instruction of a *ICW*'s proper task program must be ICW_end(). This function will test the correct termination of all the tasks spawned by the present task and will terminate either, successfully or with a failure. It is not necessary to call the standard *C* function exit(). ICW_end() will do it with the appropriate exit code.

Every internal function and variable of the interface start with the characters "ICW_", so it is not recommendable to use variable or function names following this convention in order to avoid conflicts and unpredictable behaviors.

The minimal expression of a *proper ICW task program* looks like the next example:

```
#include <stdio.h>
#include "ICW.h"

void main(int argc, char *argv[]) {
  ICW_init(argc, argv);
  ICW_end();
}
```

Of course these programs should be compiled with an *ANSI C* compiler to get the executable file. Every *ICW* task should be compiled with the next flags:

```
-I$PVM_ROOT/include
-L$PVM_ROOT/lib/SUN4
```

and

```
-lpvm3
```

Every *CW task* should be placed in the directory $HOME/pvm3/bin/$PVM_ARCH. It is possible to override this standard by specifying the absolute path to the file in the *CWL* program, or by changing this standard in the *PVM hosts file* (see *PVM* documentation).

Communication functions

We have defined a set of communication functions to communicate the *CW tasks* running in the *ICW* environment. As we have defined previously, every task has a communication rank, which defines the natural reach for the communication between tasks.

All the communication primitives to be used into the *CW tasks*, will be delimited by this rank. If it were necessary to establish communication with any task outside the rank, it would be necessary to do one of two things: **(1)** use some tasks as intermediaries to send the message or **(2)** use directly the *PVM* identification functions to determine the identity of the task which we want to communicate with, and the use the *PVM* communication functions to send the message. For an example, let us go back to *Figure # 1*. Imagine that the task *X* needs to send a message to the task A_1. As it is clear, A_1 is outside the communication rank of *X*.

One way to solve the problem is to use tasks *TP* and *TR* as intermediaries. Doing so, *X* would send the message to task *TP*, then *TP* would send it to task *TR*, which would finally send it to task A_1.

The other way is to use functions like *pvm_tasks()* to know the *PVM tid* of the task the message should be sent to, then using the standard *PVM* functions to send the message.

ICW communication functions have been defined for each possible datatype to be transmited, and all of them follow the same standard.

In general there are functions for sending and receiving data. The typical function to send data has the next prototype:

```
ICW_send_<datatype>(char *dest, int copy, int id, <datatype> *buffer, int len)
```

where <datatype> is a valid simple datatype in the programming languaje.

The only send function that is a bit different is that for sending strings, and its prototype is as shown next:

```
ICW_send_string(char *dest, int copy, int id, char *buffer)
```

On the other hand we have the receive functions. The prototype of the typical receive functions is as shown next:

```
ICW_receive_<datatype>(char *orig, int copy, int id, <datatype> *buffer, int len, long tout)
```

Here againt <datatype> is a valid datatype of the programming languaje. The function for receiving strings has the next prototype:

```
ICW_receive_string(char *orig, int copy, int id, char *buffer, long tout)
```

The meaning of the parameters is shown next:

Parameters meaning

`char *nombre`: Indicates with an identifier of a program name, the task a message is expected from, or the task a message is going to be sent to. The valid identifiers are `ICW_member`, `ICW_caller`, `ICW_next`, `ICW_previous` or a program name.

If a program name is used in this parameter, *ICW* will try to find a program within the communication rank of the task that calls the communication function with the specified name.

If it is the case of a replicated task, and the parameter copy is different from zero, the message will be sent to, or received from the *n*th copy of the replicated task.

If it is used `ICW_member` there are three possible results. If it is used to send a message, and the `copy` parameter is zero, the message will be sent to all the member tasks. If it is used to send or receive a message, and is called with the `copy` parameter different from zero, say *n*, the message will be sent to, or received from, the *n*th task of the member tasks. If it is used to receive a meesage and the `copy` parameter is zero, a message from any of the member tasks will be accepted.

If it is used the `ICW_caller` identifier, the message will be sent to, or will be accepted from the caller task.

With `ICW_next` the message will be sent to, or will be accepted from the next task in the same level in the hierarchy. The same happens with `ICW_previous`, but in this case it will be the previous task in the same level in the hierarchy.

`int copy`: This parameter indicates the number of copy of a replicated task, or the number of member task a message is goint to be sent to, or is going to be received from.

`int id`: This is an integer number that must match between the sending and the receiving processes, and is used as a validation of the messsage. A -1 in the receiver tells the process to receive a message with any id number.

`<type> *buffer`: Is a pointer to the buffer of data to be sent, or the storage area where data should be received. Its type must match the type of the data being transmited.

`int len`: Indicates the length of the data buffer.

`long tout`: Indicates how long a process should wait for a message to arrive. If it is zero, the wait time will be 300 seconds.

3.4 Execution of *CWL* programs

CW tasks must be executed through the *CWL* program. This program, although compiled, does not generate any executable code. If compilation is successful, all the tasks defined in the *CWL* program will be tried out. If the execution of all of them is successful, then the execution of the *CWL* program will be successful. If only one of the tasks fails, then the complete execution fails.

The compiler is called `icw`. The syntax to use it is as shown next:

```
icw <progname>
```

where `progname` is the name of the *CWL* program. The `progname` must end with an `.icw` extension, otherwise it will not be recognized.

Stages of execution

Execution is divided in two stages: the compilation of the

CWL program, and the execution of all the tasks involved in the cooperative work definition.

Compilation of the *CWL* program

In the first stage the *CWL* program is compiled.

The first step of the compilation consists in trying to contact the *PVM daemon* (called *pvmd3*). It is absolutely necessary that the interface be enrolled as a *PVM* task in order to make possible the communication between it and the `ROOT` task of the application.

If it is not possible to contact the *PVM daemon*, the interface will try to start it. If it is not possible, it will be impossible to compile the program, and the interface exits with an error exit code.

PVM uses a *hosts file* to know which hosts will be included in the *parallel virtual machine*[4]. We have defined the name for the hosts file as `.pvmhosts`, and it must reside in the home directory of the user who is using the interface.

Once it is verified the *PVM daemon* is running, the *CWL* program will be compiled.

Once verified the syntax is right, the compiler will check that all the declared architectures in the `ARCHS` declaration, and hosts declared in the `HOSTS` declaration, really exist in the *PVM* environment. If this is not the case, the interface will exit with an error.

Next the compiler will compile the `TASKS` and `GTASKS` declarations. The existence of executable programs into the hosts of the virtual computer is not verified at this moment, but at runtime, however the compiler will check the consistency of the declarations. The compiler will check too that all the architectures and hosts used in the declaration of the *explicitly located tasks* had been previously declared in the corresponding sections.

Finally it will be tested that the task declared as the `ROOT` task had been declared in the `TASKS` declaration, as well as all the tasks referenced in the `CW` declaration.

The result of compiling the `CW` section will be an internal representation of the hierarchy that will follow the execution of the tasks. This hierarchy will be used at runtime to determine the behavior of every part of the complete execution.

Once compilation has been determined as successful and the hierarchy has been obtained, the second stage is started.

Execution of the tasks

This stage consists of the execution of all the tasks involved in the cooperative problem. The first one to be executed is the `ROOT` task, which will be `fork`ed and enrolled as a *PVM* task.

The reason to use `fork` instead of the natural spawning process of *PVM* is that any task spawned by *PVM* does not have access to the standard input (*stdin*) nor the standard output (*stdout*). Due to this restriction it is impossible the use of these means to establish any communication with the user. Since the use of the standard arguments `argc` and `argv` is restrictied due to the specifications of the interface, not having `stdin` and `stdout` would isolate the program completely.

The interface will `wait` for the end of the execution of the `ROOT` task. If it was successful, the execution terminates successfully, otherwise the interface will retry up to three times the execution of the `ROOT` task. If after the three times the execution continues terminating with failure, then the interface will exit with an error.

Execution control

This kind of programs need a good structure for the control of the execution. It is necessary to know whether a program finished successfully or with an error.

As previously mentioned, the complete execution will be successful only if all its components were executed successfully, and will be unsuccessful if only one of the tasks fails.

There is one special condition on execution control. If a task is not a explicitly located task, or if it is explicitly located in an architecture, and it has not been possible to execute it in the first attempt, the interface will assume that it may be possible to execute it on another machine, so it will be called the *locator* in order to try to find the corresponding executable program in some other machine. If the program is found, it will be executed, otherwise, the complete execution fails.

The reason because the failure of one of the tasks produces the failure of the complete execution is that at any moment, any process can communicate with any other process. In other words, communication between processes is arbitrary, anyway, directly or indirectly. So if the failed process were restarted after the failure, it would not be possible to guarantee the consistency of the messages sent and the messages expected to be received.

4 Current state of the work

At the present time we have defined the model to express *Cooperative Work*, and we have finished implementing the first version of the interface.

At the moment we have tested it with some small programs, and it has demonstrated to work well.

We have tested it with a program that requires intensive computation to construct a *3d* image from two stereo-optical images, and the interface has demostrated to be useful in both ways: expression of cooperative work, and speed of execution.

5 Conclusions

Although this is a young project, we have dedicated much time trying to define the most desirable characteristics an interface of this kind should have.

[4] See the *PVM* documentation to learn more about the options of this file.

ICW is the first implementation of the *CWM* and the *CWL*. This work demonstrated the feasibility of the model, and will be the base for the analysis and implementation of a more complete *CWL*.

Finally we must mention that this work is one of the first approaches to solve problems through the cooperative paradigm in distributed processing environments.

We think new paradigm can facilitate the introduction of scientists into the world of parallel and distributed processing since it provides an easy interface to both, the expression of parallelism, and the writing and debugging of communicating programs.

We consider this investigation and development area as very promising for the near future.

6 Future work

There is a lot of work to do in the future. We have written the interface to be used with *C* programs, however nowadays *PVM* can work with *C++* and *Fortran*, so the corresponding interfaces could be developed. Even, in the future, *PVM* could be implemented for other programming languages, so the interface could be developed to work with those programming languages when it happens.

It could be a good idea to write the interface to work on Transputers, so it would be easy to profit the benefits offered by this hardware.

Another development could be done in the future is to improve the mechanisms to detect and, if possible, recover failures. It would be desirable to have a more sophisticated error recovering system.

Another important future development is the construction of a more complex communication system. *PVM* has a very versatile communication system, and all its virtues could be employed profitably.

Finally it would be good to develop a *GUI* for this interface, and a load balance system.

References

[Sánchez-Arias, 1996]
Sánchez-Arias, Víctor Germán
Arquitectura para el apoyo al trabajo
cooperativo basado en una red de sistemas
paralelos y distribuidos
Reporte interno LANIA
R1-1124P-A, marzo 1996

[Sánchez-Arias-2, 1996]
Sánchez-Arias, Víctor Germán
Lenguaje de trabajo cooperativo
Reporte interno LANIA
R2-1124A-P, marzo 1996

[Ritchie and Thompson, 1984]
Rtichie, Dennis M. & Thompson, K
The Unix Time-Sharing System
The Bell System Technical Journal 57
No. 6 pag 2.
Jul-aug, 1984

[Hoare, 1978]
Hoare, C. A. R.
Communicating Sequential Processes
Communications of the ACM
21(8): 666-667
1978

[Geist et al., 1995]
Geist, Al et al.
PVM: Parallel Virtual Machine
A User's Guide and Tutorial for Networked
Parallel Computing
The MIT Press
Cambridge, Massachusetts
London, England
http://netlib2.cs.utk.edu/pvm3/book/pvmbook.ps

[MPIF, 1994]
Message Passing Interface Forum
MPI: A Message-Passing Interface Standard
CRPC-TR94439
Center for Research on Parallel
Computation
Rice University
April, 1994
http://netlib2.cs.utk.edu/papers/mpibook/mpibook.ps

[Hoyos-Rivera, 1996]
Hoyos-Rivera, Guillermo de Jesús
Descripción de ICW
(Interface for Cooperative Work)
Reporte técnico interno
Maestría en Inteligencia Artificial
Universidad Veracruzana - LANIA, 1996