Specification and Control of Cooperative Work in a Heterogeneous Computing Environment

Guillermo J. Hoyos-Rivera[1], Esther Martínez-González[1], Homero V. Ríos-Figueroa[2]
Víctor G. Sánchez-Arias[2], Héctor G. Acosta-Mesa[3] and Noé López-Benítez[4]

[1]Maestría en Inteligencia Artificial
Universidad Veracruzana
Sebastián Camacho # 5
Xalapa, Veracruz 91000 MEXICO
Ph. (52 28) 17 29 57 Fax. (52 28) 17 28 55
{ghoyos, emartine}@mia.uv.mx

[2]Laboratorio Nacional de Informática Avanzada, LANIA, A.C.
Enrique C. Rébsamen # 80, colonia Isleta
Xalapa, Veracruz 91090 MEXICO
Ph. (52 28) 18 13 02 Fax. (52 28) 18 15 08
{hrios, vsanchez}@xalapa.lania.mx

[4]Texas Tech University
Computer Science Dept.
College of Engineering
Lubbock, Texas 79410
Ph. (806) 742 1194 Fax (806) 742 3519
nlb@ttu.edu

Abstract

The implementation of an interface to support cooperative work in a heterogeneous computing environment is based on previously proposed definitions referred to as Cooperative Work Model (CWM) and Cooperative Work Language (CWL). The Interface for Cooperative Work (ICW) and the Graphical Interface for Cooperative Work (GICW) are the main two components of a tool useful in the set up and control of a cooperative working environment in a general purpose heterogeneous computing platform. This tool is described in this paper as well as some desired characteristics to improve its effectiveness. The specification and control of a virtual parallel machine are illustrated with an algorithm for 3D-reconstruction from two stereoscopic images. Test results on this application are also reported.

1. Introduction

Cooperative work involves the coordination of several tasks during their execution. All tasks share a common goal and cooperation rules coordinate their actions that in turn use communication primitives to make their interaction possible. There are two important factors behind the motivation of this work. The first one is that many problems can be organized as a set of cooperative modules that could be executed in parallel. The second

[3] *Acosta Mesa is now with the Universidad Tecnológica de la Mixteca, Huajuapan de León, Oaxaca, Méx ico*

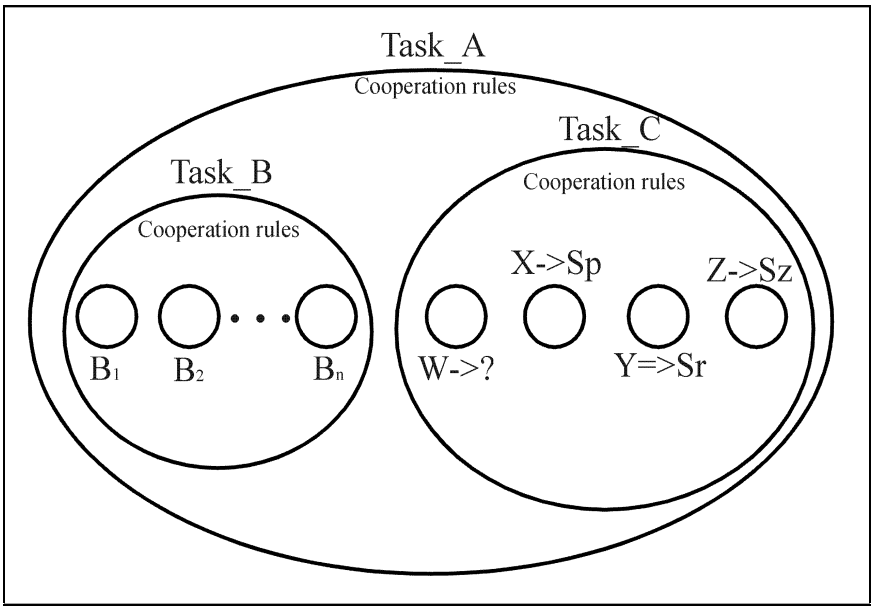

Figure 1. Hierarchical Layout of Task Processor Assignments

factor is the recognition that message passing computation is becoming more accessible today. It is no longer necessary to buy expensive devices to gain access to large computational power. Existing general purpose networks can now be used for cooperative work. These factors lead to the definition of a *Cooperative Work Model* (*CWM*) [18, 23], and a *Cooperative Work Language* (*CWL*) [19] with the purpose of making the specification of cooperative work, its parallelism and distribution easier. *CWM* and *CWL* are inspired upon the definition of *Communicating Sequential Processes* (*CSP)* [6], and the basic notions of *processes* and *pipes* used in *Unix* [17]. The *Interface for Cooperative Work* (*ICW*) and the *Graphical Interface for Cooperative Work (GICW)* are the main two components of the tool implemented based on *CWM* and *CWL*. *ICW* is useful in the set up and control of a cooperative working environment in a heterogeneous computing platform. A hierarchical specification of processes makes *ICW* different from other schemes such as *Cluster-M* [3] and *HeNCE* [2]. The main objective of *Cluster-M* is an efficient mapping of tasks into a set of processors. *HeNCE*, on the other hand, seeks the specification of tasks given in the form of a task graph such that parallelism is exploited. However, the nodes of the task graph refer to lower level specification such as procedures or routines. *ICW* allows a recursive refinement such that lower granularity is also possible when the application so requires it. Other tool that can also be compared with *ICW* is the *Wisconsin Wind Tunnel* (*WWT*) [16]. Unlike *ICW*, *WWT* is targeted to shared-memory oriented applications and can be used to simulate the behavior of hardware systems under design.

This paper describes the implementation of *ICW* and *GICW*. First, in section 2 we review the basic elements of the *CWM* model, i.e., tasks, cooperation rules, and intertask communication. In section 3 and 4, implementation issues are discussed. Section 5 deals briefly with the stereoscopic image reconstruction and section 6 reports some results obtained exploring task distribution schemes using the tool presented in this paper.

2. The Cooperative Work Model

The notion of cooperative work as a set of interrelated tasks arranged in a hierarchical structure was behind the proposed *CWM* model. In a distributed heterogeneous computing environment, some if not all tasks in the model can be executed in parallel and have the capability to communicate with each other by explicit message passing. The execution environment of tasks will be governed by predefined *cooperation rules*. Interaction among tasks will take place by using some established communication primitives.

Tasks may be assigned to any suitable host in the system. Figure 1 describes a possible hierarchical arrangement of sets of tasks. *Task_A* consists of its own execution environment and the execution environment of *Task_B* and *Task_C*. *Task_B* consists of its own execution environment and that of *n* replicated tasks denoted *as* B_i. Finally *Task_C* consists of its own execution environment and that of tasks *X* in host S_p, *Y* in any host of architecture

S_r, *Z* in host S_z and *W*. Tasks *X, Y, Z, W* and the *n* copies B_i do not contain other tasks inside them. Task *W* is a special case. It should be executed in a particular host that is not yet known. A search is required to determine the location of such a task. Note that this specification of tasks is different from that used for *Cluster-M* [3]. *CWL* specifies a hierarchical order governed by the cooperation rules between tasks and without regard at this point to any allocation scheme.

2.1. Tasks

The minimal work unit is the task. It is considered a completely executable program (a process) following the binary format of the operating system under which it was created.

By definition the tasks specified have the following characteristics:

- Each task starts and ends execution at some point in time,
- When a task starts execution, optionally receives some input parameters,
- No task shares memory with any other task, and
- The only way to share information with other tasks is by explicit message passing.

Any task may be classified according to the four different criteria described in the next paragraphs.

Types of tasks. 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, like /bin/ls, /bin/cp, $HOME/bin/print, etc. This kind of task does not have the need to communicate with other tasks. A CW task is a compiled executable program explicitly written for our interface.

Level in the hierarchy. Under these criteria, any task can be of any two types: atomic or composed. An atomic task will be any executable program. It can be either, a CW task or a generic task. Atomic tasks will consist only of its own execution environment. A composed task can only be a CW task. A composed task consists of its own execution environment and that 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 a composed task, and the composed task executing other tasks will be referred to as the caller task.

Place of execution. According to the possible places where tasks can be executed, they can be classified as explicitly located or not explicitly located tasks. A task not explicitly located is executed in any host of a virtual parallel computer. The place where these kinds 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 is due to any of three reasons: (1) the executable program exists only 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 because it may be the most suitable. A special case occurs when a task can be explicitly located but the host where the corresponding executable program resides is unknown. In this case a locator dynamically finds the host where the executable program resides.

Number of copies in a concurrent execution. Any replicated task can be explicitly located or not explicitly located. If they are explicitly located, then all copies of the task will be executed concurrently in the same host or in a subset of hosts of a specified architecture. Otherwise, each copy will be executed in any host of the system.

2.2. Cooperation Rules

Every member task will have associated with it a cooperation rule to control its execution. Three basic cooperation rules are defined: *SYNCP*, *ASYNCP* and *SEQ*.

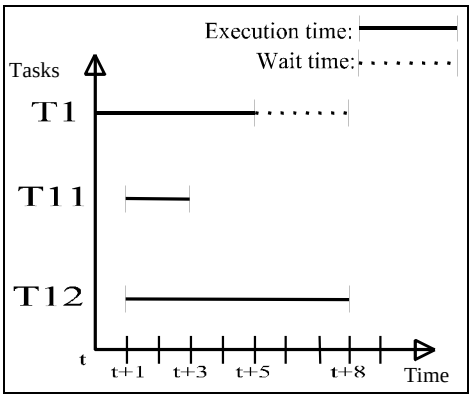

Figure 2. Behavior of SYNCP

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. As an example, consider the expression:

T1:SYNCP[T11, T12]

where *T1* is the caller task and the member tasks will be *T11* and *T12*. Task *T1* will not end until both *T11* and *T12* have terminated. Figure 2 describes *SYNCP*.

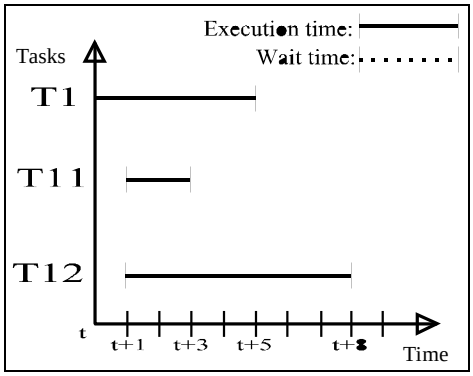

Figure 3. Behavior of *ASYNCP*

ASYNCP stands for *Asynchronous Parallelism*. This rule operates almost in the same way as *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 under this rule. The same example posed for the last rule is useful for this one, but with the difference that *T1* will be able to terminate independently of the termination time of tasks *T11* and *T12*. Figure 3 describes the behavior of this rule.

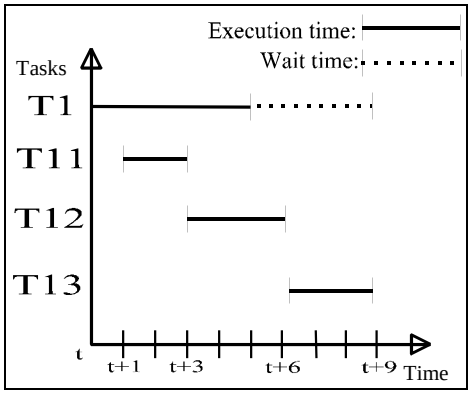

Figure 4. Behavior of *SEQ*

SEQ denotes a sequential execution. When using this rule on a series of member tasks, the next task to be executed will not be started until the previous one has finished. Figure 4 depicts its operation.

2.3. Communications

Each cooperating task in a *CWL* program has a *communication rank* that specifies those tasks that communicate with it. The communication rank of a task *T* running in *ICW* is formed by:

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

Messages sent are saved in a buffer that is accessed by the destination process when it is ready. If the message is not yet in the buffer, the receiving process must wait until it arrives.

3. Implementation

ICW maps the cooperative work defined by the model into a specific distributed heterogeneous environment. The interface provides all the necessary mechanisms to distribute, replicate and locate any task to be executed. The execution of the tasks will be monitored and controlled to guarantee a complete execution of all the programs or, in case of failure the interface will report it. The goal of this feature is to make the debugging process easier. To decide which tools to use in the *ICW* implementation, several alternatives were analyzed. Two of these alternatives include *LAM* (*Local Area Multicomputer*) [14] and *PVM* (*Parallel Virtual Machine*) [4]. *LAM* is a full extension of the *MPI* [13] (*Message Passing Interface*) standard. We initiated the implementation of *ICW* using *PVM* mainly because it offered several desirable features and it was available. Currently the interface is being updated, through *GICW*, to work with both, *PVM* and *LAM* at the user's choice.

3.1. Cooperative Work Language

A typical program written in *CWL* is composed of 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. These sections can be omitted if there are not explicitly located tasks.

The `TASKS` section contains the declaration of all the tasks that are involved in the execution. This section cannot be omitted.

The `GTASKS` section lists all the generic tasks. The interface will not accept any declared generic task to be used as a composed task. The existence verification of this type of tasks is carried out at runtime. This section is optional.

In both sections `TASKS` and `GTASKS` it is possible to indicate that a task is going to be executed in a particular

host or a particular architecture. The operator → indicates that a task is to be executed in a particular host and the operator ⇒ indicates a particular architecture.

The ROOT declaration indicates which task contains, directly or indirectly, other tasks. In other words, the task declared as ROOT is the superset in the 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 control the relation between tasks. Every task listed in this declaration will be validated against the tasks declared in the TASKS and GTASKS sections. A typical *CWL* program is shown in Figure 5.

3.2. Writing *CW* tasks

Programs for the interface are generic *C* programs with the only peculiarity that they must be compiled with the preprocessor directive `#include "ICW.h"`, which contains all the necessary definitions to control the execution and all the functions to communicate between tasks. This file includes two special functions: void `ICW_init(int argc, char *argv[])` and void `ICW_end(void)`.

```
ARCHS: SUN4,LINUX;

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

}
```

Figure 5. Typical CWL program

`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 hierarchy of tasks and then execute them, sending to every task the appropriate information to trigger execution. `ICW_init()` must be called with the standard arguments received by the *C* program because that information is used by the interface to determine the operating environment.

The last instruction of an *ICW* program must be `ICW_end()`. This function will test the correct termination of all the tasks spawned by the current task and will terminate with the appropriate exit code. To avoid conflicts and unpredictable behavior every internal function and variable of the interface start with the characters "`ICW_`". A minimal expression of an *ICW* program is shown in Figure 6.

Communication functions. All communication primitives to be used in CW programs will be limited by the communication rank of the tasks. To establish a communication with any task outside the rank it will suffice to either use some tasks as intermediaries to send messages or use directly the PVM identification and communication functions to determine the identity of the target task, and to send messages, respectively. For example, referring to Figure 1, task X may need to send a message to task B_1 but this task is outside the communication rank of X.

The *ICW* communication functions have been defined for each possible data type to be transmitted and all of them follow the same standard.

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

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

Figure 6. A minimal expression of a ICW task program

Typical functions to send and receive data have the following prototypes:

- *ICW_send_<datatype>(char *dest, int copy, int id, <datatype> *buffer, int len)* where <datatype> is a valid simple datatype in the programming language.

- *ICW_send_string(char *dest, int copy, int id, char *buffer).* This function is mainly used to send strings.

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

- *ICW_receive_string (char *orig, int copy, int id, <datatype> *buffer, long tout).*

Parameters. The parameters **char *dest** or **char *orig** indicates with an identifier of a program name, the task sending or receiving messages. The valid identifiers are `ICW_member`, `ICW_caller`, `ICW_next`, `ICW_previous` or a program name. If a program name is used, *ICW* will attempt to find a program within the communication rank of the task that calls the communication function with the specified name.

In the case of a replicated task with a copy parameter different from zero, the message will be sent to or received from the *n*th copy of the replicated task.

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

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

With `ICW_next` the message will be sent to or 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 sent to or accepted from the previous task in the same level in the hierarchy.

int copy indicates the number of copies of a replicated task, or the number of member tasks a message is sent to, or received from.

int id is an integer number that must match the sending and receiving processes. It is used as a validation of the message. A value of *-1* in the receiver tells the process to receive a message with any id number.

<type> *buffer is a pointer to the buffer that contains the data to be sent or where it will be received. Its type must match the type of the data in transit.

int len indicates the length of the data buffer.

Receive functions have an additional parameter:

long tout indicates how long a process should wait for a message to arrive. If it is zero the waiting time defaults to 300 seconds.

3.3. Execution of *CWL* programs

CW tasks must be executed through a *CWL* program. This program, although compiled, does not generate any executable code. If the execution of all tasks is successful the execution of the *CWL* program will be successful. If only one of the tasks fails the overall execution environment fails. The execution process is divided into two stages. One stage is the compilation of the *CWL* program, and the other stage is the execution of all the tasks involved in the cooperative work specified.

Compilation stage. The first step of the compilation stage attempts to contact the *PVM* daemon. If it is not possible to do it, the interface attempts to start it up. If this is not possible the program will not compile and the interface exits with an appropriate error code. *PVM* uses a hosts file to know which hosts will be included in the parallel virtual machine. The hosts file name is `.icwhosts` and resides in the user home directory. 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, otherwise, the interface will exit with an error. Next, the compiler will compile the `TASKS` and `GTASKS` declarations. The existence of executable programs in the hosts of the virtual computer is verified at runtime. However, the compiler will check the consistency of the declarations. The compiler will also check 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 check that the `ROOT` task has 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 followed during the execution of tasks. This hierarchy is used at runtime to determine the behavior of every part of the execution process.

Execution stage. This stage consists of the execution of the entire hierarchy of tasks involved in the cooperative problem. The first to be executed is the `ROOT` task, which will be forked and enrolled as a *PVM* process. The interface will wait for the end of the execution of the `ROOT` task. After three unsuccessful execution attempts the interface will exit with an error. As previously mentioned, the complete execution will be successful only if all its components are executed successfully. If at least one task fails, the overall execution process fails.

4. The Graphical Interface

An option to build and execute a *CWL* program is through the *Graphical Interface for Cooperative Work* (*GICW*). With the *GICW* is possible to create an efficient grouping for the objective Cooperative Work in an interactive and dynamic way. Figure 7 shows a view of *GICW*.

The *GICW* offers two operation modes. The first mode manages information elements for the Cooperative

Work of an application via a set of windows. It integrates the tasks (proper and generic), hosts and architectures.

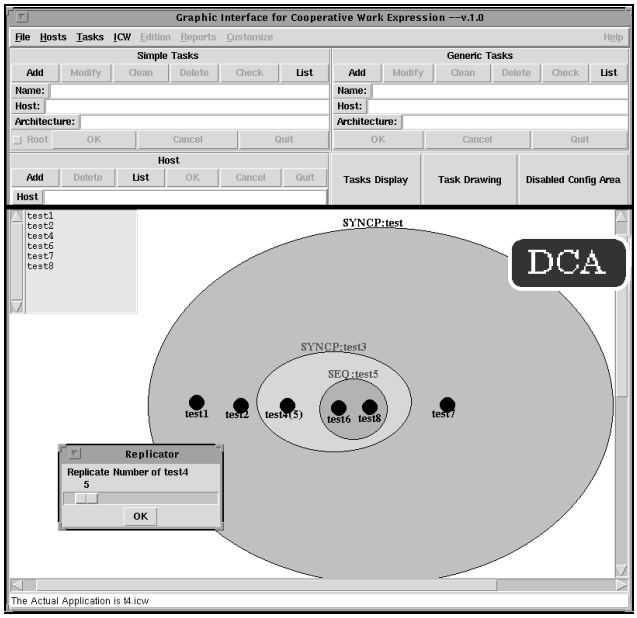

Figure 7. Graphic Interface for Cooperative Work

The second is the Graphic Configuration mode. It integrates all operations adding a *Dynamic Configuration Area* (*DCA*). The *DCA* shows the task representation looking like the graphs in the *CWM* shown in Figure 1. In the central part of this area appears the root task represented by an oval with its name and its cooperative rule. Within this oval is possible to integrate composed and atomic tasks required in the cooperative work of the current application.

A composed task is also represented with a circle with the name of the task and its cooperation rule. Small circles are used to represent atomic tasks. To add one task in the configuration it is selected from the box list located in the left side of the *DCA*. The selected task is then dragged into the root task area or into the area of a previously created composed task.

To create or modify tasks, hosts, or architectures, it is important to use the corresponding entries that appear in the upper side of the configuration area. These windows are useful to specify directly weather a task will execute in a particular host or architecture.

To integrate the number of copies of an atomic task is necessary to select it from de *DCA* and adjust the number of copies in the window that appears for this purpose. Code generation is performed according to the objects appearing in the *DCA* and their grouping. The output file generated is identified with the application name and the extension `.icw`. This file contains the *CWL* specification of the cooperative work.

The option *ICW Execution* is selected from the menu and a window appears to display execution results.

The implementation is based on the scripting language *Tcl* and its graphical toolkit *Tk* (version 8.0) which are widely portable and allow easy GUI programming [15].

The *GICW* has no validation mechanism for the existence of tasks. A parser is used when a configuration file is loaded. The *GICW* extensions are based on the *MPI* implementation of *ICW*. The implementation integrates 1) a state monitor mechanism of the tasks that compound the current cooperative work application, 2) the search in alternative paths that are not in the *PATH* environment variable, and 3) the dynamic configuration of the host in the virtual machine.

5. Application: 3D Reconstruction

One of the most important features of human vision is its capacity for perceiving a three dimensional world. This perceptual capacity is achieved through a highly evolved visual system composed of cooperative visual modules, which are able to recognize objects and describe the layout, and motion of our surroundings [21]. One visual module that is most relevant for the perception of depth is *stereopsis* [10]. This visual module takes as input two images of a scene taken from different locations (for example, one taken by the left eye and the other by the right eye) and computes the correspondence of features which most likely originate from the same 3D surface patch (Figure 8). From the features correspondence is possible to obtain the depth at these points [7].

We describe a distributed implementation of the Pollard, Mayhew & Frisby (PMF) algorithm for stereoscopic reconstruction. Our implementation has been coded in *ICW* and runs on a network of *SUN* and *Silicon Graphics* workstations [1]. The main stages of the PMF algorithm for stereoscopic reconstruction [10] are the following:

- Edge detection. The points with highest changes in intensity are detected in each digital image. This can be achieved with a variety of edge detectors, such as the Marr-Hildreth operator, Canny edge detector or Sobel's [5]. For simplicity, we have taken this last option.
- Stereoscopic correspondence. This is the core part of the whole algorithm that finds the most likely correspondences between edges in the left and right images.
- Reconstruction. Once the correspondences have been found it is possible to evaluate simple arithmetic expressions to find the depth at these locations.

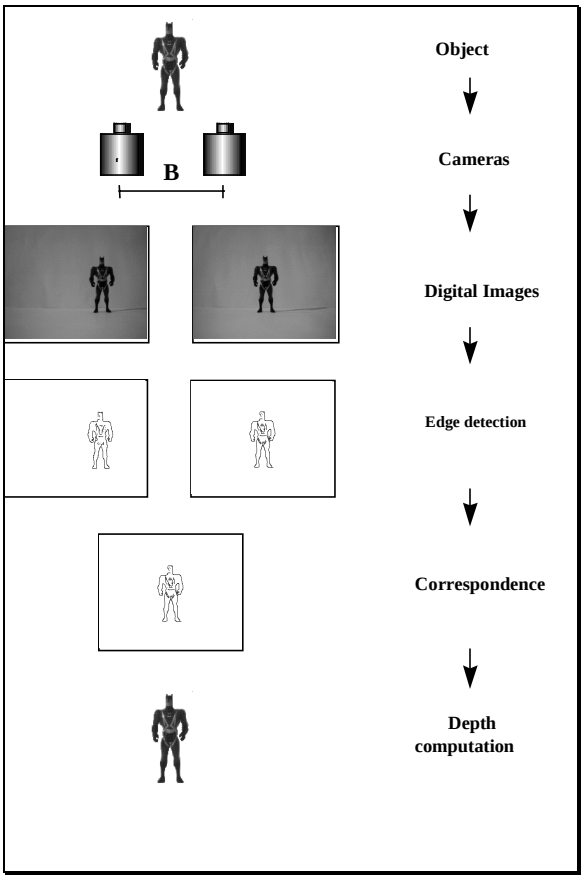

Figure 8. Stages in stereoscopic reconstruction

We take images from two cameras with parallel viewing directions to simplify the problem of stereo correspondence, because in this case, the corresponding features lie in epipolar lines. This means, that they are approximately in the same raster line in the two images.

Using the constraint that corresponding features usually have a *disparity gradient DG* less than one, we apply a search process to find the best matches. The disparity gradient is defined for a pair of matches, where each match associates one feature in the left image with one feature in the right image.

If feature *Pl = (plx,ply)* in the left image is matched with feature *Pr = (prx,pry)* in the right one, and feature *Ql = (qlx,qly*) in the left is matched with feature *Qr = (qrx,qry)* in the right as shown in Figure 9, then the *disparity D* for the match *(Pl, Pr)* is obtained as follows:

$$D(Pl,Pr) = plx - prx.$$

The *disparity difference DD* for the pair of matches *(Pl,Pr)* and *(Ql,Qr*) is just the disparity for the *P* match minus the disparity for the *Q* match. That is:

$$DD((Pl,Pr),(Ql,Qr)) = d(Pl,Pr) - d(Ql,Qr)$$

Now imagine the two images superimposed. The *cyclopean separation CS* is the distance from the mid-point of the line joining *Pl* and *Pr* to the mid-point of the line joining *Ql* and *Qr*. The gradient *DG* is the absolute value of the disparity difference divided by the cyclopean separation.

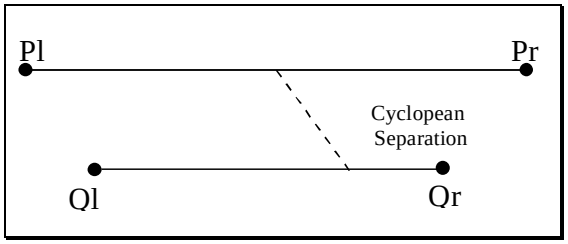

Figure 9. Geometry of the disparity gradient

In Figure 9, the disparity difference is the difference in length between the two horizontal lines. The cyclopean separation is the length of the slanting line. The *DG* can be expressed as follows:

$$DG = DD/SC \leq 1$$

If a point *(X,Y,Z)* projects at *(xl,y)* and *(xr,y)* in the left and right image respectively, we can find its position in space, in terms of the disparity xl – xr, using the formulas [7]:

$$X = \frac{B(xl + xr)}{2(xl - xr)}$$

$$Y = \frac{By}{xl - xr}$$

$$Z = \frac{Bf}{xl - xr}$$

where *B* is the separation between the camera's centers and *f* is the focal length.

6. Comparative results

The distributed implementation consists of dividing each image in bundles of lines and allocating a bundle to each workstation. Once a bundle is processed, the results are returned and concentrated by the host computer for display as described in Figure 10.

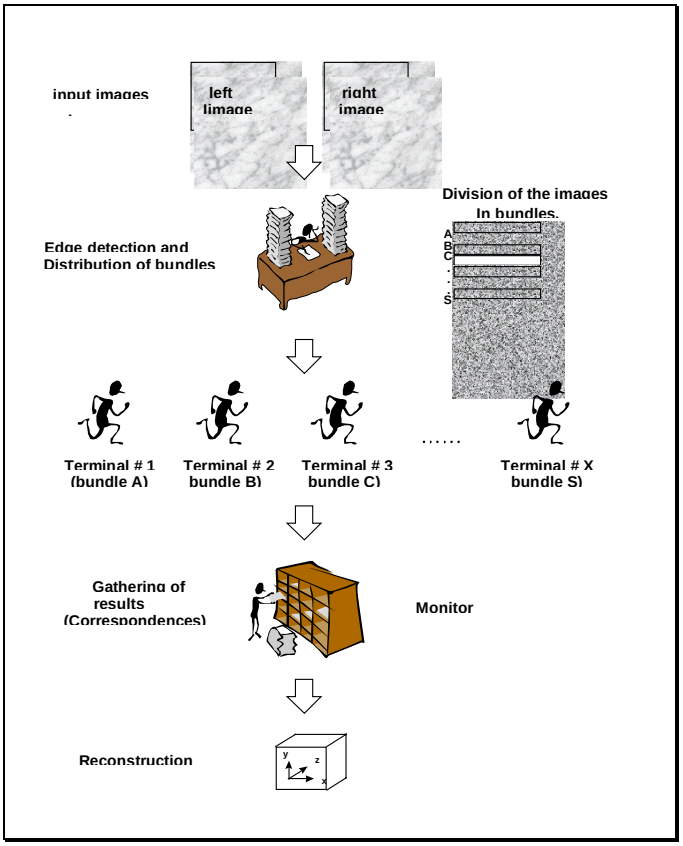

Figure 10. Distribution of tasks for stereoscopic correspondence

Each processed image is composed of 240 rows and 320 columns. The size of each bundle is obtained as the number of rows divided by the number of available workstations. In our implementation the size of the bundle can vary up to *240/n* lines, where *n = 1, 2, …, 12* is the number of workstations. For a uniform number of features in each bundle, the lines that compose each bundle are not taken consecutively but every *240/n* lines. Some results of the reconstruction are shown in Figure 11.

Two sets of tests were carried out. In the first set only 12 SUN units (models ELC, ILC and IPC with 24 Mbytes of RAM) were used. For the second set of tests Silicon Graphics (SGI) machines (Indy R4600 with 32 Mbytes of RAM) where introduced. Under the second scheme, a SUN unit distributes tasks to SGI units (labeled 1 to 5). The results obtained are shown in Figures 12 and 13. Both figures compare results obtained under no workload conditions and normal workload conditions.

Figure 12 shows a monotonic improvement in the execution time (this behavior is more consistent under a normal workload condition) up until the number of units reaches 10. An increase in the number of workstations does not show any improvement in the overall execution time. At this point, very likely the communication costs involved with further partitioning of the application upset any gain in execution times. In this regard similar behavior can be observed with the combination SUN and SGI workstations in Figure 13. Naturally, the introduction of SGI units renders a dramatic improvement in the execution time. However, particularly in the case of no workload conditions, performance remains constant indicating again the effect on communication costs. Under normal workload conditions, improvements are noticeable with additional units.

The objective of these experiments is to demonstrate the feasibility of using *ICW* to execute distributed applications. The results highlight the need to incorporate appropriate task allocation and scheduling heuristics [3, *11,12,20*] to map the set of tasks in the application to suitable units in the system and improve execution times. The integration of these schemes will make ICW a complete and useful tool in the analysis and implementation of large-scale parallel and distributed applications.

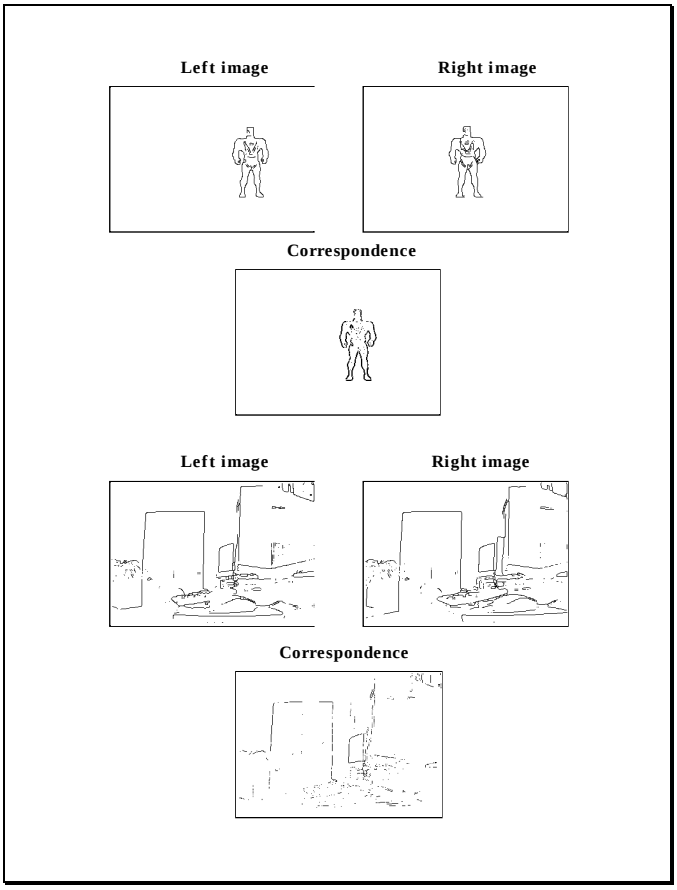

Figure 11. Results of the reconstruction

7. Conclusions and future work

At this time we have defined a model to specify cooperative work and completed the implementation of the first version of the interface (*ICW*). We are currently working on *LAM 6.1* and refining some management aspects of *CW* applications.

ICW is the first implementation of the *CWM* and the *CWL* models and although the project is at an early stage, trying to define the most desirable features has been the most time consuming endeavor. This work however, demonstrates the feasibility of the model, and will be the base for the analysis and implementation of a more complete *CWL*. The tool proposed facilitates the introduction of scientists into the world of parallel and distributed processing as it provides an easy interface to the specification of parallelism, writing, and debugging of communicating programs using installed general purpose networked resources.

However, to optimize performance the interface must be able to evaluate hosts configurations and detect the states of those processing units used in the distribution.

The tool has been written to deal with *C* programs. An upgraded version will incorporate transputers to facilitate the specification of lower level parallelism. Another expected development is to improve the mechanisms to detect and, if possible, recover from failures. Yet another important future development calls for the integration of task assignment heuristics and their evaluation to achieve a much improved task distribution in terms of execution times and resource utilization. In terms of future applications for which ICW will be used include cooperative virtual environments and gesture recognition [22] algorithms.

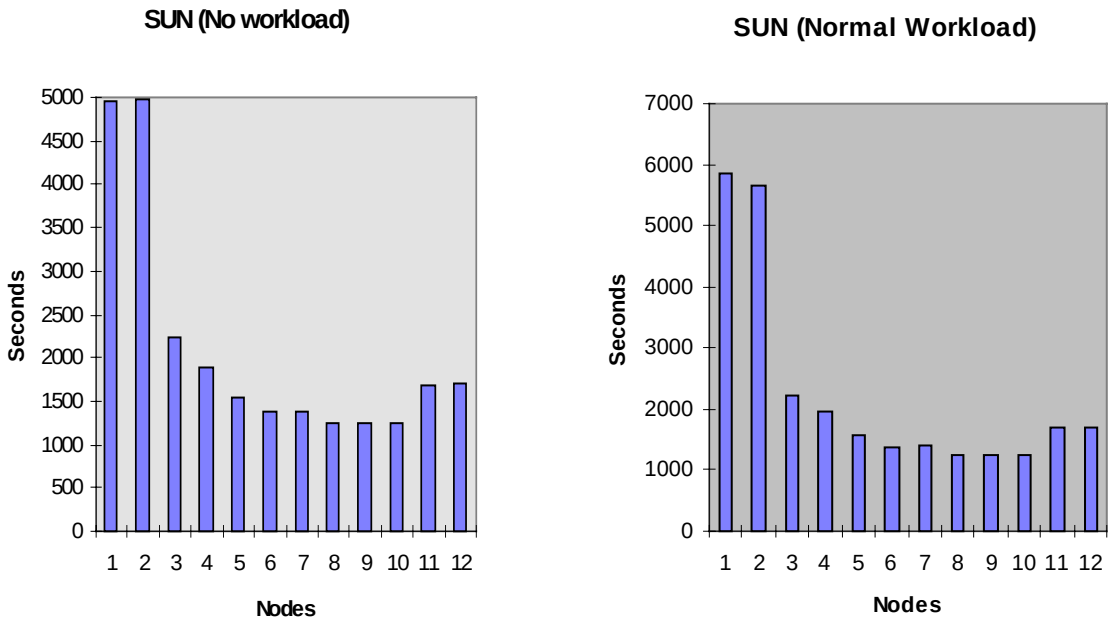

Figure 12. Execution times on a homogeneous system consisting of SUN workstations only.

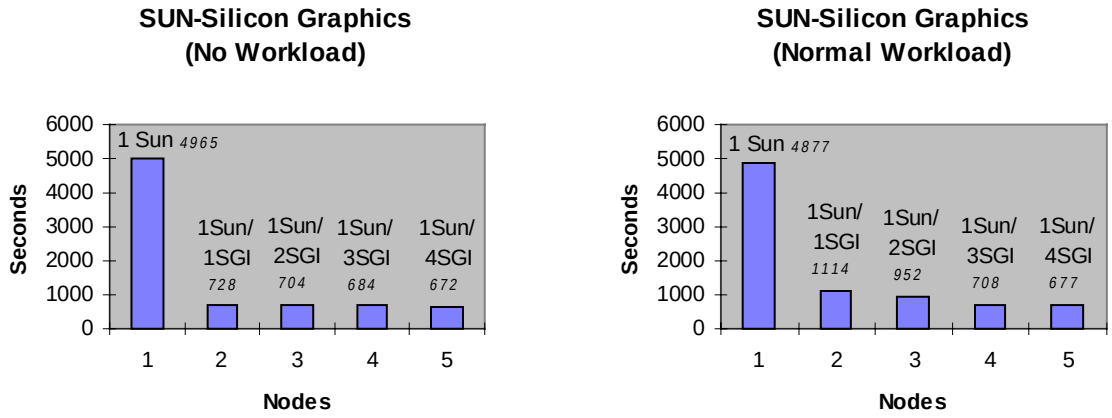

Figure 13. Execution times on a combined SUN-SGI configuration.

References

[1] H.G. Acosta-Mesa, H.V. Rios-Figueroa, *"Implementación distribuida de un módulo de reconstrucción estereoscópica"*, Maestría en Inteligencia Artificial, Universidad Veracruzana - LANIA, 1996

[2] A. L. Beguelin, J. J. Dongarra, G. A. Geist, R. Mancheck, and K. Moore "HeNCE: a Heterogeneous Network Computing Environment", University of Tennessee, Computer Science Dept. Technical Report No. 93-205, August 1993.

[3] M. M. Eshaghian and Y-C. Wu, "A Portable Programming Model for Network Heterogeneous Computing", in *Heterogeneous Computing*, Mary M. Eshagian, ed., Artech House, Inc., 1996.

[4] A. Geist, et al. "PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing", MIT Press. 1995

[5] R.C. González, R.E Woods, "Digital Image Processing", (3rd. edition) Addison-Wesley, 1992

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

[7] B.K.P, Horn, "Robot Vision", *The MIT Press*, 1987

[8] G.J, Hoyos-Rivera, V.G. Sánchez-Arias "Proposal of an Interface to Support Cooperative Work in a Distributed System Environment". I Encuentro de Computación. Taller de Sistemas Distribuidos y Paralelos. Memorias. Querétaro, Qro. México. September 1997. SMCC, SMIA, UNAM, Asoc. Filosófica de México, SMI, UAQ.

[9] G.J. Hoyos-Rivera, "Propuesta de una Interfaz para el Apoyo al Trabajo Cooperativo en un ambiente de Arquitectura Paralela y Sistemas Distribuidos", Maestría en Inteligencia Artificial, Universidad Veracruzana - LANIA, 1997

[10] J.E.W. Mayhew, J.P. Frisby, "3D Model Recognition from Stereoscopic Cues", *The MIT Press*, 1991

[11] A. R. McSpadden, N. Lopez-Benitez, "Stochastic Petri Nets Applied to the Performance Evaluation of Static Task Allocations in Heterogeneous Computing Environments", *IEEE Heterogeneous Computing Workshop, 1997*, Geneva, Switzerland.

[12] D. A. Menasce, D. Saha, S.C. Silva Porto, V.A.F. Almeida, S.K. Tripathi, "Static and Dynamic Processor Scheduling Disciplines in Heterogeneous Parallel Architectures", *Parallel and Distributed Computing*, Vol. 58, 1995, pp. 1-18.

[13] 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

[14] N. Nervin. "The Performance of LAM 6.0 and MPICH 1.0.12 on a Workstation Cluster. Ohio Supercomputer Center. Technical Report OSC-TR-1996-4 . Columbus Ohio.

[15] J. K. Ousterhout. Tcl and the Tk Toolkit. Addison-Wesley. Professional Computing Series. September, 1995.

[16] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J.C. Lewis, and D. A. Wood, "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers", ACM Sigmetrics Conference, May 1993.

[17] D.M. Ritchie, K. Thompson. "The Unix Time-Sharing System", The Bell System Technical Journal 57 No. 6 page 2. Jul-Aug, 1984

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

[19] V.G. Sánchez-Arias, G.J. Hoyos-Rivera. "Using PVM to Build an Interface to Support Cooperative Work in a Distributed Systems Environment". Lecture Notes in Computer Science. Proceedings of Recent Advances in Parallel Virtual Machine and Message Passing Interface 4th European PVM/MPI Users' Group Meeting. Cracow, Poland, November 1997. pp 127-134

[20] B. Shirazi, M. Wang, G. Pathak, "Analysis and Evaluation of Heuristic Methods for Static Task Scheduling", *J. of Parallel and Distributed Computing*, Vol. 10, 1990, pp. 222-223.

[21] S. Ullman, "Analysis of Visual Motion by Biological and Computer Systems" *Computer 14*, 57-69, 1981.

[22] K. Voss, H. V. Rios-Figueroa and J. Peña. "Head tracking by glasses detection". Proceedings of the Workshop on vision and robotics, National Computer Conference, Mexico, 1997.

[23] Sánchez-Arias, V. G. "Arquitectura para el apoyo al trabajo cooperativo basado en un ambiente de architectura paralela y systemas distribuidos", Proceedings 2nd. International Workshop on Parallel Processing, DEA-IIMAS-UNAM, Mexico, July 1996, pp 8-10.

Author Biographies

Guillermo J. Hoyos-Rivera received the BSc degree in Informatics in 1992, and the MSc degree in Artificial Intelligence in 1997 from the Universidad Veracruzana with honors. He has been a Lecturer in the Universidad Veracruzana as a Lecturer since 1991 and in several other schools and universities. Currently, he is an associate full-time Researcher in the Artificial Intelligence department in the Universidad Veracruzana and the Universidad Anahuac de Xalapa. His research interest include operating systems, computer networks, telecommunications, protocols and parallel and distributed processing. He is a member of the SMIA (Mexican Society of Artificial Intelligence).

Esther Martinez-Gonzalez received the BSc degree in Informatics from the Universidad Nacional Autonoma de Mexico (UNAM) in 1994. She is about to graduate with an MSc degree in Artificial Intelligence from the Universidad Veracruzana. Since 1994 she has been a Lecturer, first in the UNAM and currently at the Universidad Veracruzana. She is now engaged with the Research Coordination of the Universidad Veracruzana. Her research interest areas are in distributed systems and cooperative work.

Homero V. Rios-Figueroa received the BSc degree in Mathematics and the MSc degree in Computer Science from the Universidad Nacional Autonoma de Mexico (UNAM) in 1987 and 1989, respectively. The PhD degree in Computer Science and Artificial Intelligence from the University of Sussex, England, in 1994. From 1988 to 1990 he was a Lecturer with the UNAM. In 1994, he joined the Laboratorio Nacional de Informatica Avanzada (LANIA) in Xalapa, Mexico, where he is now a full time Researcher. His research interests include computer vision and virtual reality. He is a member of the SMIA (Mexican Society of Artificial Intelligence).

Victor G. Sanchez-Arias received the BSc degree in Control Comunications and Electronics and the MSc degree in Computer Science from the Universad Nacional Autonoma de Mexico (UNAM) in 1974 and 1976, respectively. He obtained the Diplomé D'Etudies Approfondies (DEA) in Informatics from the École Nationale Supérieure de l'Institut de Mathèmatiques Appliquées de Grenoble (IMAG), France. In 1982, he received the PhD degree in Informatics Engineering from the IMAG. He was with BULL, Paris, France from 1985 to 1988 where he was engaged in the research and development of networks, distributed systems and applications. He was a Researcher at the IMAG (1981-1984) and at the UNAM (1988-1991). Since 1992, he is titular Researcher and consultant at the Laboratorio Nacional de Informatica Avanzada (LANIA). He is a member of the SMIA (Mexican Society of Artificial Intelligence) and SMCC (Mexican Society of Computer Science). His research interest include distibuted and cooperative systems and parallelism.

Hector G. Acosta-Mesa received the BSc degree in Computer Systems from the Instituto Technologico de Veracruz. He also received his MSc degree in Artificial Intelligence from the Universidad Veracruzana. From 1991 to 1996 he was working for CFE (Comision Federal de Electriciad) in Xalapa, Veracruz. He is now a Researcher and Professor in the Universidad Technologica de la Mixteca in Oaxaca. Areas of research interest include robotics and computer vision. He is a member of the SMIA (Mexican Society of Artificial Intelligence).

Noe Lopez-Benitez received the BSc degree in Communications and Electronics from the University of Guadalajara, Guadalajara, Mexico. The MSc degree in Electrical Engineering from the University of Kentucky, and the PhD in Electrical Engineering from Purdue University in 1989. From 1980 to 1983, he was with the IIE (Electrical Research Institute) in Cuernavaca, Mexico. From 1989 to 1993, he served in the Dept. of Electrical Engineering at Louisiana Tech University. He is now a Faculty member in the Dept. of Computer Science at Texas Tech University. His research interests include fault-tolerant computing systems, reliability and performance modeling, and distributed processing. He is a member of the IEEE, the IEEE Computer Society, and the ACM.