A Programming Tool for the Development of Parallel Computer Vision and Image Processing Algorithms and Applications

Odemir Martinez Bruno¹, Luciano da Fontoura Costa²

¹ Department of Computer Science and Statistics - ICMC, University of São Paulo
Caixa postal 668, CEP 13560-970, São Carlos, SP, Brazil
{obruno@icmc.sc.usp.br}

² Cybernetic Vision Research Group – IFSC, University of São Paulo
Caixa Postal 369, CEP 13560-970, São Carlos, SP, Brazil.
{luciano@if.sc.usp.br}

Abstract—
This article presents the development, implementation, and applications of the CVMP parallel programming tool for image processing and computer vision. Running on Borland Delphi and C++ Builder, this simple and easy-to-use tool incorporates visual programming and object-oriented capabilities, having already allowed a series of applications and results, which are also outlined.

Keywords—Image Processing, Parallel Computing

I. INTRODUCTION

Although parallel computing is posed to enhance image processing, computer vision and related areas, some obstacles have constrained its effective and broad application. One of the principal problems is the difficulty to implement concurrent programs, a consequence of the fact that most development tools for parallel programming are destined to experts in this area [BRU 00a]. This difficulty is aggravated by the existing variety of available programming environments and languages, each with its specific tools and parallel structures. As an alternative to these problems, a new methodology has been conceptualized in order to provide simple and effective access to parallel programming to those computer vision researchers and practitioners who are not experts in concurrent computation. The initial results of this enterprise, which are reported in the current paper, are based on the message-passing tool called CVMP (for Cybernetic Vision Message Passing). The project principal characteristics are:

- **Popular Development Platform:** The approach is based on a conventional and popular development platform, namely Borland Delphi and C++ Builder.
- **Visual Programming:** In order to speed up the development, visual programming concepts have been adopted.
- **Object Oriented Programming:** OOP represents one of the key points in CVMP, allowing simple and secure development, code reuse, and effective modeling of complex systems.

The current paper presents the CVMP approach, covering its basic elements, some of the already obtained implementations and results, and the conclusions and perspectives for future works.

II. THE CVMP TOOL

The CVMP approach includes a set of tools for the development of parallel programs in the Delphi/C++ Builder platforms, using concepts of visual programming and OOP. It is composed of a set of components (VCL - Visual Component Library), native in Delphi, and additional applicatives organized under the following five groups: CVMP basic, CVMP Extended, CVMP processor farm, CVMP image processing, Statistics and Launcher, which are discussed below. Figure 1 shows part of the CVMP components palette in the Borland Delphi environment.

Fig.1 CVMP components palette in Delphi

A. CVMP Basic

The applicatives in this group include two components supplying the message passing primitives: a component for distributed memory MIMD systems (network connected PCs) and another for shared memory MIMD (multiple processors PCs). These two components provide the
ing processes. This feature, part of HPD standard, proved to be useful to debug parallel programs. It allows debugging groups of processes with similar characteristics as well as selecting processes suspected to be erroneous by the programmer. This feature was also implemented successfully in the p2d2 and Node Prism parallel debuggers. The main difference here is the simplicity of PADI, since selection can be done directly from buttons in the Main View.

In conjunction with the selection mechanism is the visualization of processes. Visualization has been applied in different areas of computer science, mainly to make interfaces more user-friendly. p2d2 created a very smart visualization grid, but it does not seem to scale well when lots of processes are being debugged (the identification of processes becomes difficult when the number of processes grows). The PADI approach uses the selection mechanism as a filter to processes visualization, clearing the Processes Area and showing just processes being inspected. Besides, the PADI visualization area includes information like process states (by color), process names or pids and tids (internal ids).

Besides improving the prototype in order to prepare a version for initial distribution, future work includes providing different types of visualizations (currently there exists only one) and to develop runtime system interaction, so that PADI can receive specific information from a specific system and show this information in the interface (e.g., process creation, send and receive events, etc.).

REFERENCES


view are the interface intuitiveness, the programming models accepted and the platform. The goal of PADI is to be an easy to use tool available to popular parallel programming libraries (like PVM and MPI) and platforms (Linux clusters). Also, it will be available without costs and will be used for further research in on-line debugging and monitoring areas.

Some considerations about other tools were already done at the "Related work" section and are retaken in the "Conclusions" section. This one is more concerned in making some kind of practical comparisons with knowed tools like p2d2, Prism and TotalView. Unfortunately, a practical comparison with p2d2 is not possible, since it is not available for download and its use is restrict to NASA (see the p2d2 site in [HO099]). Prism was formerly conceived at Thinking Machine Corporation and now is part of Sun HPC environment [SUN01], hence it is not available for Linux clusters. Both accept PVM and MPI programming models.

Totalview is a commercial tool, but it has an evaluation distribution for Linux clusters and others (available in [ETN01]). As a wide used commercial tool, TotalView is full implemented and has a lot of useful features. A comparison, at least for while, can be made only at interface level, that is the primary goal of PADI.

Let's consider the same PVM farmer/workers example presented at section "User interaction with PADI". When debugging that example with PADI it is easy to separate the farmer (master in figure 1) and the workers (slaves) into two groups. This can be done by selecting the farmer as user (just selecting its ellipse with the mouse) and choosing the User group to select it as the current group or choosing the Not user to select the workers.

The notion of a group in Totalview is less flexible. There are only two types of groups: the control group and the share group. The control group includes the parent process and all related processes. The share group is the set of processes within a control group that share the same source code. When debugging the same example with the TotalView defaults it is possible to experiment the effect of a control group by stepping the program. If the Step group option from the pop-up menu is choosed, all processes will do the step. Thus, the step can only be done individually or by all processes in the control group (that are all processes - farmer and workers - in this case).

The default breakpoint semantics is different in TotalView. When setting a breakpoint in a process it will be shared among processes with the same source code. By default, breakpoints follow the shared group semantics. The manual explains how to change the properties of an action point, but (at least apparently) this feature was not available in the tested evaluation version of TotalView (Linux x86 TotalView 4.1.0-1).

One of the main design choices of PADI is to separate the interface into two different kinds of views: one for parallel commands and other for individual commands. Such a structure clarifies the semantics of the debugger. In PADI it is possible to visualize all processes that will be affected by a debugging command before it is actually sent to execution. The PADI's Main View is the responsible for all distributed commands. The counterpart of this view in TotalView is the Root window, that is a textual list of the debugging processes and its states. However, all debugging commands are available by the pop-up menu from the processes views.

VII. CONCLUSION

PADI is a parallel on-line debugger interface whose main goal is to provide intuitive parallel debugging facilities. In order to achieve this goal, the two levels of a parallel debugger (coordination and process levels) were defined and implemented separately. Thus, the process level was made completely familiar to programmers, since it is very similar to traditional sequential debuggers. In addition, the coordination level of PADI was developed to embody parallel features, like distributed commands, selection mechanism and parallel visualization.

The coordination level of PADI is represented by its Main View. Several design choices contribute to making PADI intuitive and easy-to-use. The most important is the fact that the Main View embodies all the parallel commands. Other choices contributing to the goal of intuitiveness and simplicity include the fact that parallel commands, similar to traditional, sequential ones, were made directly available as buttons. Another interesting feature is the definition of groups of processes, that allow programmers to select processes to receive parallel commands. This is very useful since applications can have many processes performing different tasks during execution. Finally, visualization of parallel processes makes the tool very easy-to-use, since users can easily identify processes and access their process level view (the Process View) through their icons in the visualization area (Processes Area of the Main View).

Comparing PADI with tools already mentioned in this paper, shows that existing tools inspired some important design decisions (even to adapt some good ideas or to choose a different design).

The first one is the decision of separating, at user level (interface), the distributed commands from the serial ones. This approach is different from TotalView's approach since it is not possible to send any distributed debugging command from TotalView's Main Window, but just to choose the processes to be debugged. PADI's approach proved to be clearer and more practical, since one window (the Main Window) concentrates all the distributed actions, including the distributed debugging commands.

One of these distributed actions is the possibility of select-
MainView() / if action = run new RunAction() 
RunAction() for all_procs do_verify() if OK new Send(run) 
Send() in = mount_msg() do _verify() new RunAction() ifOK good <small><i>cond</i></small> 1

Proc() doBreak() { status = stopped update_interface() }

Break() Proc p; p = get_in_server(tid) p.doBreak()

Thread_recv() m = recv_from_fiddle() unpack_msg(m) if type = breakpoint new Break() 

Fig. 4: Example of the processing flow in PADI: the boxes are the objects and the arrows are the relationship between them (creation or method invocation). The superior row of objects represents the processing flow to send a command (run in this case) to Fiddle. The inferior row represents the processing flow that occurs when an event from Fiddle (a breakpoint hitting in this case) is received until it is showed in PADI.

vocated by the interface and its processing flow until it is sent to Fiddle. The objects responsible for the interface control (MainView() or ProcView()) receive the command and create the corresponding action object (the RunAction(), in this case) to perform the verifications (status and selection). For each process that passes the verification, a Send() object will be created that mounts a message with required information about the command (the tid of the process and the run arguments in this case) and sends it to Fiddle.

After the sending of the run command an asynchronous event can occur, like hitting a breakpoint. This situation and the event back flow is also depicted in figure 4. There is a thread (named in figure 4 as the Thread_recv() object) that receives events like the arrival of a process at a breakpoint and creates the appropriate object to treat the event (the Break() object in the example). Then, this class gets the object reference to the process that caused the event and invokes the corresponding method in the Proc() class. This method (doBreak() in the example) updates the status of the process (that is stopped when arriving in a breakpoint) and also updates the interface: change the color of the process in the Processes Area and mark the corresponding line at source code (if the Process View is opened and showing the source code).

At this point it is important to note that commands like run can be distributed, what will create a number of Send() objects, one for each process. On the other hand, the responses are centralized in one thread that will receive all the asynchronous events from Fiddle. This centralized thread is used in order to establish a communication with Fiddle via a pipe mechanism. Meanwhile, Fiddle is being modified to accept Java clients, so the pipe is a temporary solution.

The PADI processes’ objects (instances of Proc()) are created by load or attach commands. Then, when a debugging command, except of these two, is started in the interface, processes’ objects states are verified. If the command was originated in a process view, that saves the tid of the owner, only the process’ object holding this tid is verified and the command is sent only to it. If the command was originated from the main view, all processes’ objects, whose references are maintained by the server, are verified and commands will be sent only to the tids that satisfy the conditions described earlier. This way of proceeding may seem heavy, but the advantage is that it is made locally. In other words, it avoids that a command that cannot be executed by a process in any way be sent through the network and be renegotiated by the target process. For example, a process whose state is terminated cannot be stepped. A step command will be intercepted by PADI locally, avoiding unnecessary network traffic.

VI. PRELIMINARLY RESULTS

The PADI prototype can already be used such that it is possible to formulate some practical comparisons with some existing tools. In the present phase of the work, the most important characteristics to be analyzed from a practical point of
backbone onto which all the other CVMP tools are implemented.

The communication between CVMP objects adopts the master-slave scheme. The nature of the CVMP Basic object, which can be either master or slave, is determined through one of its properties. Virtual channels between master/slave objects are established [BRU 97] [BRU 00b] [BRU 00a] in order to implement communication between objects, a concept inspired in the Transputer initiative [INM 88]. Figure 2 presents some of the possible configurations allowed by this flexible strategy.

Fig.2 Two examples of the CVMP configurations

The set of the CVMP primitives are shown and described in Table 1. These primitives consist of the properties and methods of the CVMP Basic object, that allow the communications between the processes (message exchange). An example of the CVMP Basic programming is shown in Figure 3, presenting the Delphi code for the parallel computation of four processes using two machines (master and slave).

Master process algorithm

cmp1.mp_init_file('init.ini');
cmp1.mp_master;
cmp1.mp_send('start');
  Calculate B
  cmp1.mp_send(b);
  Calculate C
  (A in B)
while f<>end do begin
  application.processmessages;
  f:=cmp1.mp_receive;
  end;
result2:=cmp1.mp_receive;
show (result1, result2);

Slave process algorithm

cmp1.mp_init_file('slave.ini');
cmp1.mp_slave;
while f<>start do begin
  application.processmessages;
  f:=cmp1.mp_receive;
  end;
Calculate A

b:=cmp1.mp_receive;
cmp1.mp_send(a);
Calculate D
(B in A)

Fig.3 Delphi code example of how to use the CVMP Basic primitives.

<table>
<thead>
<tr>
<th>TABLE I</th>
<th>CVMP Main Basic primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>mp_init_file</td>
<td>Property – Load the configuration file.</td>
</tr>
<tr>
<td>mp_slave</td>
<td>Method – Set the object as slave.</td>
</tr>
<tr>
<td>mp_master</td>
<td>Method – Set the object as master.</td>
</tr>
<tr>
<td>mp_send(mes:str)</td>
<td>Method – Sends a message through the Virtual Channel.</td>
</tr>
<tr>
<td>str:=mp_receive</td>
<td>Method – Return the first message of messages FIFO.</td>
</tr>
<tr>
<td>Setblocking</td>
<td>Property – Set the messages as blocking.</td>
</tr>
<tr>
<td>sendfile(str)</td>
<td>Method – Send a high granularity message (file) through the Virtual Channel.</td>
</tr>
<tr>
<td>receivefile(str)</td>
<td>Method – Receive a high granularity message (file).</td>
</tr>
<tr>
<td>mp_close</td>
<td>Close a Virtual Channel connection.</td>
</tr>
</tbody>
</table>

B. CVMP Extended

This includes components similar to the CVMP Basic, but with additional properties and methods (encapsulated in the CVMP Basic), addressing the handling of message packets, message-passing synchronization, semaphores, as well as partition and distribution of images [ALM 94] [COD 94]. Often, parallel image processing algorithms require the partition of the image in order that each portion is simultaneously processed in several distinct processors. The CVMP Extended is capable of partitioning the images in several ways. The images can be divided into portions of several sizes, in order to help the load balance in heterogeneous systems. In many situations, the simple division of the image implies data dependence (see Figure 4). A typical example is the convolution of templates in the space domain, a technique underlying several image processing algorithms [GON 93]. In order to cope with this requirement, the pixels adjacent to each image portion are also enclosed. Different neighborhood sizes can be considered depending on the type of data dependence.

Fig.4 Examples of image partitioning with incorporation of neighboring pixels.

223
C. CVMP Processor Farm

The processor farm paradigm [ALM 94], one of the most frequently adopted strategies in parallel image processing [BRU 00a], is characterized by a master which distributes the task to a number of supervised slaves (see Figure 5). The CVMP incorporates components specifically designed to implement this parallelization strategy, supporting configurations up to 16 slave machines or processes.

![Fig.5 The Processors Farm Paradigm.](image)

Through the CVMP processor farm, the user easily implements a distributed application by using visual means, without the need of a single code line. Indeed, it is enough to configure some properties of the master and slaves (e.g. number of tasks, network configuration, etc.) and to add the calls to the parallel algorithms into the slaves code.

D. CVMP Image Processing

This set incorporates components containing ready-to-use image processing parallel algorithms, which can be included just by dragging them into the forms, make a few configurations, and add a few lines of code. Each technique is composed by combinations of master/slave components, following the CVMP. Currently, the following algorithms are available:

- Local operators – convolution in space domain [GON 93].
- Chromatic channels – operations involving chromatic images, with parallel execution between the chromatic planes.
- Fourier transform.
- Hough transform [SCH 89].
- Hough transform with backmapping, a technique proposed by Gerig & Klein [GER 86] in order to enhance the peaks produced by the Hough transform.
- Fractal dimension: includes algorithms based on Minkowski's sausages and box-counting [TRI 95][KAY 94].

E. CVMP Statistic

Includes two components, one for statistical analysis and another for dynamic execution analysis, allowing proper statistical analysis of the behavior of the execution and message exchanges in concurrent programs developed in CVMP. The statistics component is capable of measuring the performance directly through the program code, which has to include calls to the statistic procedures at critical points of the code, defined by the programmer.

Figure 6 presents two windows of the statistics application. It is used to visualize the statistical data obtained through the CVMP statistics components. The first window considers four machines running concurrently, the x-axis indicates the time in milliseconds, and through the boxes it is possible observing the duration, start and finish time of each process. The second window shows the respective numerical information.

![Fig.6 Windows of the Statistics application, used to visualize the CVMP statistics data.](image)

F. CVMP Launcher

Includes a component and a demon executing in background in the network machines, in order to enable the proper triggering of the processes in the network machines. Through calls to the object methods, the demon can trigger the execution of a specific program in any specific machine in the network.
III. PARALLEL IMAGE PROCESSING ALGORITHMS

In this section there are presented four examples of parallel image processing algorithms (local operators, Hough transform, Hough transform backmapping and fractal dimension) implemented by using the CVMP. Each one is discussed in terms of implementation and performance. Because of its strong OOP base, CVMP allows any implemented algorithm (implemented by using CVMP) to be encapsulated and integrated into the CVMP image processing (see II.D).

A. Local Operators

This technique is broadly used in image processing and computer vision and provides a good example of local operator [GON 93]. Basically, it consists of the convolution of a template with the image, obtaining a processed image. The kind of processing (which can include low and high pass filtering, template matching, etc.) will be determined by the template characteristics.

The adopted parallel strategy consists in partitioning the image, distributing the slices to the machines, that is processed concurrently, and finally join the results of this processing, obtaining the processed image.

Figure 7 presents the parallel strategy using four machines. The involved stages are: the original image (I), image division and its distribution (II), the processing of each image part in a different processor unit (III), results reconstruction (IV) and the combination of the results (V). The load balance can be accomplished by controlling the size of the slices in such a way that the most powerful processors receive the larger slices.

![Fig.7 Parallel strategy for local operator algorithm using 4 processors units.](image)

The reported experiment was performed on a computer network composed of 4 similar computers - AMD K6 II 375 MHz, connected by ethernet NE2000 of 10 Mb/s and Fastethernet of 100 Mb/s. The considered operation was a low pass filtering [GON 93] involving the sum of the elements of the mask for the number of components of the filter. The parallel and sequential algorithm execution times were measured while processing a image with 500x500 pixels by using operators with templates of different sizes (3x3, 5x5, 9x9, 15x15, 31x31, 51x51, 71x71, 85x85 and 101x101 elements).

Figure 8 exhibits the comparison of the speed-ups obtained for execution in a ethernet based computer network (10 Mb/s) and fast-ethernet based (100 Mb/s) of a 500x500 pixels image. The faster communication allowed a substantial increase of the performance for the smaller filters (3x3, 5x5, 9x9 and 15x15). For larger filters (31x31, 51x51, 71x71, 85x85 and 101x101), the performances of both networks (10 Mb/s and 100 Mb/s) get closer because the processing becomes more highly compute bound, while the communication time becomes relatively smaller.

![Fig.8 Comparison between ethernet 10 Mb/s network and fast-ethernet 100 Mb/s computer network.](image)

B. Hough Transform

Introduced by Hough in 1959 to calculate particle trajectories [HOU 59], the Hough Transform is largely used on image analyses as a global pattern recognized method. The basic idea of the method consists to find curves on image that can be parameterized, such as line segments, polynomials, circles, ellipses, etc. The parallel Hough transform implemented in this paper was used to detect line segments on the image, which constitutes it most frequent use [SCH 89].

![Fig.9 Parallel Hough transform architecture using four processing units.](image)
Figure 9 shows the parallel strategy used to implement the algorithm using four machines. The diagram shows six stages that correspond to: original image (I), image division (II), distribution of the image slices (III), parallel computation of the Hough arrange (p,θ) for each image slice, sending the arrange to the master machine and joining them (V), obtaining the final result of the processing (VI).

The Parallel Hough transform algorithm was implemented using CVMP and executed on a computer network (10 Mb/s) with four similar machines (AMD K6 II - 375 MHz). Figure 10 presents a diagram comparing the execution times of the sequential algorithm (1 processor) with the parallel approach (2, 3 and 4 processors). A 500x500 pixels image was used in the experiment.

C. Hough Transform Backmapping

The Hough transform backmapping technique [GER 86] consists of the creation of a new accumulator array. For this new space, however, only the cells corresponding to the maximum values along each sinusoidal produces by the Hough transform are increased. In this way, a reinforcement of the Hough transform is obtained in an attempt to better locate the local peaks and to reduce the background noise caused by occasional alignments of points produced by interference between objects and segments with few points.

Due to the new transform computation and the continuous search for maximum points along the Hough space, corresponding to each point of the image, that technique involves considerable overhead, consuming substantial computer power and motivating parallel implementations.

Figure 11 exhibits a diagram illustrating the parallel approach to backmapping considering four processing elements, which can be easily modified. The stages in the diagram mean: image split and fragments distribution (I), distribution of the Hough space (II), backmapping processing from the data of the image fragments (III), transmission of the new Hough space generated by the backmapping processing to the master process, where the elements of each Hough space will be added in order to obtain the sought result (IV).

We implemented the parallel version of the technique in a distributed system in a computer network based on ethernet (10 Mb/s), with four similar machines (AMD K6 II - 375 MHz). Figure 12 presents the processing time of the sequential and parallel implementations for two different image sizes (250x250 and 500x500 pixels).

D. Fractal Dimension

Fractal dimensions can be used as a means to determine shape and image complexity. Although initially related to fractal geometry research, the concept of fractal dimension popularized quickly, allowing applications in several areas such as: material sciences, geology, computer vision [BIS 98], neuromorphology [COS 99], etc.

Several techniques for fractal dimension estimation have been described in the literature [KAY 94], including the particularly simple Minkowski sausage method [TRI 95] considered in this paper.

The parallelization of this technique is particularly simple and can be implemented directly with the CVMP Processor Farm component. Firstly, a Processor Farm architecture is defined. The master process distributes a copy of the image to all the slaves and a radius parameter (r) specific for each slave. The basic task consists in the convoluting a circular region with radius r and the
The respective area determination. The results (areas) are sent to the master as soon as the calculations are made.

The experiment was performed for a binary image with 512x512 pixels, considering 32 disks with radii varying by 2. Figure 12 presents the execution times, obtained through the CVMP statistics component, each subsequent box (represented in light and dark gray) indicates the execution of one process. Due to the small data flow implied by the message passes in the net, the system allowed particularly good performance and a small number of execution gaps. The excellent performance is corroborated by the fact that the parallel system with four heterogeneous machines (200 MHz, 250 MHz, 300 MHz and 375 MHz) is approximately 2.7 times faster than the sequential version executed in the fastest machine (375Mhz).

![Figure 12 Parallel fractal dimension algorithm executing time for four machines](image)

**IV. IMPLEMENTATIONS USING CVMP**

CVMP is the current platform supporting the development of high performance applications in the Cybernetic Vision Research Group. Through its usage, computer vision and image processing researchers have benefited with parallelism in the development of several projects, some of which are outlined in the following.

**A. Cyvis-1**

The Cyvis-1 project [COS 94][BRU 97] (standing for Cybernetic Vision, version 1) represents an attempt at obtaining versatile computer vision systems through the progressive incorporation of biological principles such as selective attention, Zeki and Shipp's multi-stage integration framework [ZEK 88][ZEK 93], and effective integration of top-down and bottom-up processing (see Figure 13). As such approaches inherently demand the use of parallelism in their respective implementation, the research on Cyvis-1 has strongly depended on the CVMP. The already obtained results include the integration of visual attributes, the distributed environment, and the incorporation of parallelism in several algorithms developed as part of the Cyvis-1 project.

**B. TreeVis**

TreeVis [BRU 00a], an abbreviation for Tree Vision, is a system for the automated recognition of arboreal plants through the comprehensive extraction of features from images of leaves and the respective statistical classification. The bases of the system consists of the systematic exploration of the leaves geometrical properties through a large number of visual features, that implicates on a large time of processing (about five minutes for each sample), justifying a parallel solution. The parallelism in the TreeVis project has been implemented by using the CVMP Processor Farm components (see Figure 14), through the replication of the feature extraction module.

![Figure 14 Overall organization of TreeVis](image)

On the experiment reported here, the system included six identical machines (Pentium II – 300 MHz) connected through a network (10 Mb/s). Figure 15 shows an execution time diagram for 12 samples, where each sample is represented as a block. Observe that each system machine executes a sample concurrently. A speed-up near the expected maximum (i.e. 6) was obtained, as the sample number is a multiple of the number of system processors.
The TreeVis approach is based on statistical analyses, needing several samples of each species for optimal training and classification. The performance of the processor farm approach reaches the optimal when the number of samples is a multiple of the number of processors or is close to it. However, the performance decreases in two situations: (i) when the number of samples is less than the number of processors and (ii) when the number of samples is higher than the number of processors, but it is not close to the multiple.

The Parallel implementation of Treevis has an automatic configuration architecture, capable of changing the parallel approach in execution time. The system compares the number of samples and the number of processors, and if one of the two situations (i) or (ii) are detected, the system change its parallel approach, in order to minimizes the execution time and improve its performance.

Figure 16 exhibits the concurrence between the feature extraction submodules approach (using three machines) and Figure 17 shows its execution time diagram for 3 samples running on 6 machines. Although, it performance is worse than the processor farm approach (when the number of samples is a multiple of the processors number), its utilization guarantees the system speed-up for the two situations (i) and (ii) which the processor farms approach performance decreases.

C. Ζynergos

The CVMP has also provided the basic support for parallelization in a project oriented to the integration of several important approaches in computer vision and pattern recognition, such as algorithm validation, performance comparison, datamining, psychophysical experiments, artificial intelligence, etc [BRU98] [BRU01]. Basically, it is expected that the several advantages and disadvantages of these approaches complement each other in order to catalyze the development and application of computer vision concepts and tools. More specifically, the CVMP has been used for the parallel implementation and execution of the genetic algorithm, required for datamining tasks.
Figure 18 presents the time diagram for the parallel execution of the genetic algorithm in four different machines under CVMP, connected through 10Mbit/s ethernet connection. Each subsequent box (represented in light and dark gray) indicates the execution of one of the basic algorithm steps.

V. CONCLUSION

This article has reported the development, implementation, and application of a tool designed to support parallelization in image analysis and computer vision. Its basic elements and some of its already successful applications have been described. The benefits of the CVMP have been substantiated in practice, helping several computer vision researchers to solve several distinct problems.

In addition to the continuation of the outlined projects (e.g. Cyvis-1 and TreeVis), future developments should include the implementation of other image analysis techniques and the extension of CVMP in such a way that it interacts with MPI [PAC 97] or PVM [GEI 96].

REFERENCES


[BRU 00a] BRUNO, O. M. Parallelism in Natural and Artificial Vision, University of S. Paulo, Brazil, 2000 (Ph.D. Thesis in Portuguese).

