KUNGL TEKNISKA HÖGSKOLAN
Royal Institute of Technology

Multi-EDA:
A Programming Environment for Parallel Computations

V.V. Vlasov, L-E Thorelli, H. Ahmed

Department of Teleinformatics

KUNGL TEKNISKA HÖGSKOLAN
Royal Institute of Technology

Multi-EDA:

A Programming Environment for Parallel Computations

V.V. Vlasov† , L-E Thorelli and H. Ahmed

This work was funded by NUTEK under contract number 93-3086.

† The author was supported by a scholarship from the Swedish Institute during his work on this project.

TRITA–IT–R 94:29
ISSN 1103–534X
ISRN KTH/IT/R—94/29—SE

Department of Teleinformatics

Contents

1 Introduction

This report presents the software implementation of the Extended Dataflow Actor model, EDA, using the Parallel Virtual Machine, PVM, system [2]. A formal description of the EDA model can be found in [1]. The goal of our research has been to develop Multi-EDA, which is a programming environment for testing and evaluating the different aspects of the EDA model using PVM. Several applications were tested in this environment using a cluster of workstations.

The remainder of this report is organized as follows. Section 2 briefly introduces the EDA model. In section 3 an overview of the most important features of the PVM system is given. Section 4 describes the Multi-EDA, mEDA, environment and the next section presents some applications which were developed using mEDA. Our conclusions are given in section 6.

2 The EDA Model

In EDA, a computation is presented by a dataflow graph whose nodes denote objects or actors, each containing data and a sequential thread of control. Data is classified into two basic categories: *local* and *shared*. Local data is private and can not be accessed by other objects. On the other hand, shared data is used by several objects. Shared data is distributed between objects according to the object oriented nature of EDA. The arcs connecting EDA nodes do not represent data dependency in the conventional sense, but rather they indicate the communication and synchronization requirements of its objects in terms of the shared variables.

A shared variable declared in an object is accessible by others using synchronization rules defined in terms of *fetch* and *store* operations. Each variable may be in one of two states: *full* (containing data) or *empty.* The state of each variable is indicated by a valid tag.

2.1 Shared Variables in EDA

A parallel computation model with a notion of *shared variables* must support at least four basic mechanisms for memory access synchronization:

- ❒ Accessing data in mutual exclusion.
- ❒ Single writer-multiple readers synchronization.
- ❒ Producer-consumer relationships.
- ❒ Barrier synchronization.

Since the synchronization requirements of these operations and their associated overheads vary considerably, an efficient computation model should treat each case on individual basis and support it by dedicated mechanisms. Furthermore, this distinction must be reflected at the programming language level to help both the user and the compiler to support the efficient execution of applications.

2.2 Shared Memory Operations

Based on this argument, EDA recognizes three kinds of shared variables: *x, i* and *s*, each with special synchronization requirements. In terms of the operations carried out on these variables, we can specify the following:

- ❒ *x*-operations, for accessing critical regions in mutual exclusion and supporting synchronous producer-consumer relationships.
- ❒ *i*-operations, for synchronizing single writer-multiple readers.
- ❒ *s*-operations, for supporting asynchronous producer-consumer relationships.

EDA defines four operations on shared variables: *fetch, store, clear* and *close*. Below, the semantics of these operations are described for each variable kind.

- ❒ Fetch and store operations on x-variables are synchronous and alternating, they also cause the executing thread to be suspended. A fetch operation destructively reads the value contained in the x-variable. On the other hand, a fetch from an empty variable enqueues the request on that variable. If the variable already has a queue of suspended fetch operations initiated by other objects, the latest request is appended to the existing queue[1]. After successful completion of a fetch operation from an x-variable, the first suspended store operation will be activated writing a new value to the variable and changing its state to full. A store operation to an empty x-variable writes a new value to it and changes its tag to valid. It will also cause the head of the queue of suspended fetch operations, if any, to resume. Otherwise, if the x-variable is full, the store operation is enqueued. In this case, a successful fetch operation will cause the first suspended store to resume.

1. This assumes a FIFO priority, however, requests may have other priorities, e.g. according to their deadlines.

- A store operation to an empty *i*-variable updates it and changes the tag to valid. It also causes all suspended fetch operations on that variable to resume. A store operation to a full *i*-variable is ignored leaving its contents unchanged. An *i*-store operation does not cause the executing object to be suspended. An *i*-fetch operation on an empty variable enqueues the request to that variable and the executing object is suspended. Fetching valid data from an *i*-variable leaves it (value and tag) intact.
- *S*-variables facilitate stream communication between objects. An *s*-store operation is supported by a buffering mechanism, therefore, it always[1] succeeds and the executing object continues without suspension. An *s*-store to an empty variable changes the tag to valid and causes the first suspended fetch operations on that variable to resume. If the variable is already full, the store operation is enqueued on that variable. An *s*-fetch operation on a full variable reads the data destructively and is asynchronous in nature, i.e. in principle it does not require the suspension of the executing object. A successful fetch operation allows the first waiting store to resume.

As mentioned earlier, two additional operations are defined on shared variables: *clear* and *close*. The first, resets the valid tag of the associated variable and initiates an exception routine to deal with any pending requests. The *close* operation, on the other hand, means that the shared variable is inaccessible, consequently, any future reference to it will also initiate an exception routine.

Barrier synchronization is not supported in EDA by a special dedicated mechanism, instead it can be shown that a barrier can be implemented efficiently using a combination of *x* and *i* operations.

3 PVM: The Parallel Virtual Machine

PVM is an integrated framework for heterogeneous computing [2]. It allows designers to exploit a set of networked computers as a virtual distributed-memory multiprocessor and supports parallel computation at the level of Unix processes, using message passing.

3.1 Overview of PVM

PVM contains two main components: (1) the PVM daemon, *pvmd*. (2) the PVM library, *libpvm*. An instance of the daemon runs on each host of the virtual machine and provides support for host-process interactions. The library contains routines which form the interface between PVM applications and the run-time system (PVM daemons).

PVM has the following features:

- The problem to be solved is divided according to its functional or data structure into tasks. A task is a complete program (C or FORTRAN code) which can be built independently.
- A PVM task can be spawned from another active task. Each obtains a unique identifier, *tid*, which is used to specify the destination address for message passing.
- PVM tasks communicate using tagged messages. A task can send messages asynchronously, and is responsible for receiving messages directed to it. It can use blocking or non-blocking receive procedures.

3.2 The PVM Library

The PVM library, *pvmlib*, includes the following groups of functions to facilitate process control, task-host information acquisition, virtual machine configuration and communication.

- Process control functions are used to start the PVM daemons (*pvm_start_pvmd*), to spawn new tasks (*pvm_spawn*), to enrol into PVM (*pvm_mytid*) and to exit (*pvm_exit*, *pvm_halt*).
- Informative functions are used to get information about virtual machine configuration (*pvm_config*, *pvm_mstat*, *pvm_notify*), about running tasks (*pvm_tasks*, *pvm_tidtohost*, *pvm_notify*) and about incoming messages (*pvm_bufinfo*).
- Configuration functions are used for adding hosts (*pvm_addhosts*) and deleting them (*pvm_delhosts*) from the virtual machine.
- Communication functions are used for packing data (e.g. *pvm_packf*), sending messages (e.g. *pvm_send*), receiving messages (e.g. *pvm_recv*, *pvm_nrecv*) and unpacking them (e.g. *pvm_unpackf*).

An example PVM program is provided in Appendix 1.

1. Of course, there is a physical limit on the size of this buffer. An exception routine may control flow of the stream if production of data is much faster than its consumption and there is a risk of buffer overflow.

4 Multi-EDA

Multi-EDA (mEDA) is a parallel distributed implementation of the EDA computational model in a PVM environment. It allows the user to develop applications using EDA concepts and to run them in a real parallel-distributed environment. mEDA consists of two parts: the EDA kernel and a library of functions and macros (see Appendix 2) which provides interface to applications.

4.1 mEDA Tasks

An mEDA program is structured as a number of tasks each containing several objects and shared variables. Tasks are spawned using the *eda_task* routine from the mEDA library, which facilitates creation of any number of instances of a task. During spawning, the parent task sends the task identifiers of its children to the kernel and also to each newly spawned child in order to notify them about their siblings. Each child is responsible to get this information using the *eda_inpvm* routine.

To exit mEDA, a task must contain the macro *eda_exit* which generates an exit request to the kernel. The mEDA kernel insures the synchronous exit of all tasks of the parallel program to avoid deadlocks while accessing shared variables which are distributed among the tasks.

4.2 Objects

An object is an activity, which is represented by a descriptor and a thread. The descriptor contains all necessary system data defining the context of an object during its life cycle. It is dynamically allocated when the object is created and is released when the object terminates. Objects are created using the macro *eda_obj* and inserted into a ready queue. A number of objects may share the same thread in which case each of them obtains a unique instance number. This number is used for accessing the object's local data and also for choosing a specific branch in the shared thread. An object can get its instance number using the macro *eda_mynum*.

The descriptor structure contains the following fields (see also Appendix 2):

- ❒ Object pointer: A pointer used to insert a given object into a queue (e.g. ready queue or waiting queues) when the object is not active.
- ❒ Object identifier: Is the instance number in a group of objects sharing the same thread.
- ❒ Arguments of a protected memory operation: These fields are used to store the following arguments when an object is suspended: destination address, size of data, addressing mode (direct or indirect) and operation code.
- ❒ Reactivation array: This array is used to save and restore the stack and CPU registers during object context switch.

A thread is a C function which never returns. It begins with the macro *eda_runthread* and ends with the macro *eda_exitthread*. Object context switch occurs when either the object exits or when it is suspended because of shared memory access failure. To support object context switch, mEDA uses *setjmp* and *longjmp* routines.

4.3 Shared Variables

Shared variables in mEDA are distributed among tasks and used to support interaction between objects. A shared variable is a static structure declared by the macro *eda_shared*. The structure of a shared variable contains the following fields (see also Appendix 2):

- ❒ Variable state: This field indicates the state of the shared variable which is either *full* or *empty.*
- ❒ Pointers to queue of waiting objects: These are two pointers, one to the head of the queue and the other to its tail. The waiting queue is a fifo formed by discriptors of objects which are suspended because their access to a given shared variable had failed.
- ❒ Variable value: It contains the actual value, or a pointer to it, if the state of the shared variable is *full*.

Shared variables allocated in the same task as the object which tries to access them are called *local shared variables*. Variables which belong to other tasks are called *remote*. Each object can access the variables allocated to any task. Local shared variables are accessed by means of *eda_fetch* and *eda_store* routines. Remote shared variables, on the other hand, are accessed using *eda_rfetch* and *eda_rstore*. In the latter case, a shared variable is addressed using two components: its task identifier and its address inside the task.

4.4 Run-Time Management

In mEDA, a task has two main common resources: the processing element and shared memory, which are used by the set of objects.

An object can be in one of three states: *ready, running* and *suspended*. The state of an object depends on the availability of both resources. An object is ready if its required shared data is available. All ready objects are appended to a ready queue. A ready object becomes running if it is at the head of the ready queue and at the same time can be assigned to the processing element. At any given time, only one object is running. This object is suspended whenever a shared memory access fails. If the object is suspended on a local shared variable it is inserted in a waiting queue for that variable. The object at the head of the queue is extracted and appended to the ready queue when the pending operation is resumed.

If the shared variable is allocated in another task, the object sends an access request to that task. In case of fetch or x-store operations the object is suspended until a reply is received. The remote task can receive and service a request only during context switching. If the request could not be satisfied, the remote task creates a "*hidden*" object which is inserted into the waiting queue of requested shared variable. This object is responsible for replying whenever the requested data becomes available. As soon as a reply arrives the suspended object becomes ready and is appended to the ready queue.

In the case of an s-store operation, if the accessed shared variable is full, a threadless object called a buffer object is created to save the value to be stored. This object is inserted into the appropriate waiting queue. When the buffered value if finally written to the shared variable, the buffer object is terminated.

An object initiates a context switch either when it exits or becomes suspended. In the first case, the descriptor of the object is released. In the second case, the context of the suspended object is saved in the object descriptor which is inserted in the waiting queue of a shared variable. In both cases a new object descriptor at the head of the ready queue is extracted and its object becomes running.

The termination problem is solved using a semaphore in the mEDA kernel. During spawning, the parent task notifies the kernel about its children and the semaphore is incremented by the number of spawned tasks. When a task exits, it notifies the kernel by sending an exit request and the semaphore is decremented. The task requesting to exit becomes suspended until an exit permission is received. At this stage the task does not contain any local active or waiting objects. However, it may still contain "*hidden*" and "*buffer*" objects which were created by remote requests from other tasks. When the semaphore becomes zero, the kernel broadcasts exit permissions to all tasks and halts PVM.

5 Applications

This section contains the description of three applications which were developed to verify the EDA model using the mEDA environment. The first application is a two-dimensional Laplace equation solver with Dirichlet boundary conditions, using the simultaneous displacement method of Jacobi [3]. This task represents a class of problems which can be parallelised using *data partitioning*. The second application is in the area of image processing and it involves zooming still images. This application represents a class of problems which can be parallelised using both *data* and *functional partitioning*. The third application is a pipelined video decoder-player based on the software MPEG decoder-player developed at the University of Berkeley. The mEDA version of the player was relalized as a two stage pipeline.

5.1 The Dirichlet Problem

This problem is formulated as:

$$\nabla^2\Phi = \frac{\partial^2\Phi}{\partial x} + \frac{\partial^2\Phi}{\partial y} = 0$$

Where x and y are coordinates of a surface and F is a function. To solve this equation using the Jacobi method, the surface is partitioned into an n x n grid whose elements represent the initial state of F. A new value of F can be calculated using:

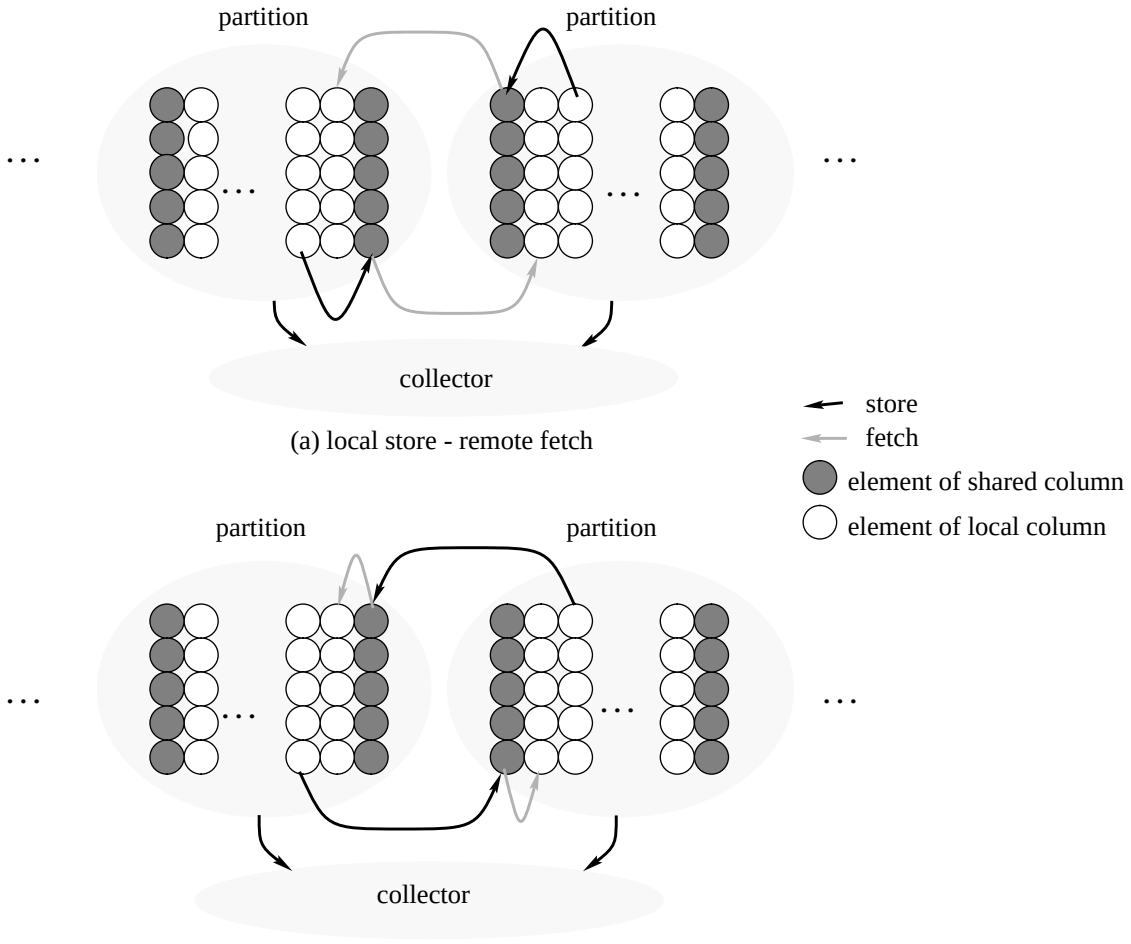
$$\Phi_{x,y} = (\Phi_{x-1,y} + \Phi_{x,y-1} + \Phi_{x+1,y} + \Phi_{x,y+1}) \times 0.25$$

The value of the function at the boundaries of the surface is known and constant. After $N = en^2/2$ iterations, a steady state value of F is reached with an error factor of 10^e.

In an mEDA program, the grid is represented by matrix F which contains n x n elements. The boundary conditions are represented by vectors of length n + 1. An mEDA program is constructed from two types of tasks: *partition* and *collector*. The collector task starts first, it creates m collector objects and spawns m partitions (m < n). Each partition task has

one object which calculates n/m columns of matrix F and sends a result to the collector using *s(x)*-store operations. On each iteration, the partition tasks exchange their boundary columns using *s(x)*-store and *x*-fetch operations. The collector objects fetch the final results from the partitions and the collector task sends them to the output.

Figure 1 illustrates two schemes for data communication between partitions. In the first scheme a partition uses local *s*-store operations for storing its boundary columns to its shared variables which are used to communicate with its neighbours. To get updated boundary columns from its neighbours, the partition uses remote *x*-fetch operations. In the second scheme a partition uses remote *s*-store and local *x*-fetch operations to communicate with its neighbours.



(a) local store - remote fetch

(b) remote store - local fetch

Figure 1: Data Communication Schemes of the Dirichlet Problem

The second scheme is faster than the first one since it requires less communication overhead. The reason is that a remote *x*-fetch requires two communication operations; One to send a fetch request and the other to receive a reply. In the case of a remote s-store however, only one communication is performed.

Figure 2 illustrates the run times of programs for both schemes as a function of the number of hosts, where each partition task is processed on a separate host. In these experiments, the size of matrix F is 512 x 512 floating point elements and the number of iterations N = 100.

The increase in execution time beginning from number of hosts equal to 16 is due to using additional hosts from another LAN. This causes messages to pass through a router which in turn increases the communication cost. To eliminate the influence of other network traffic, (e.g. mail, file servers access, etc.) the execution time for each point of the graph was measured at least 100 times at night, where traffic is minimal and the minimum values were selected.

The Dirichlet problem has a regular communication pattern and is a computaionally intensive task which involves more computation than communication. Because of this, it shows a speed up close to the ideal curve, as seen from the graph of Figure 2.

Another observation is that it is more efficient to allocate shared variables at the consumer object rather than at the object producing data. In the first case, the consumer uses remote fetch to access a shared variable allocated at the producer which requires two communication operations. In the second case, however, the producer stores the value to a shared variable allocated at the consumer which can use local fetch to access it.

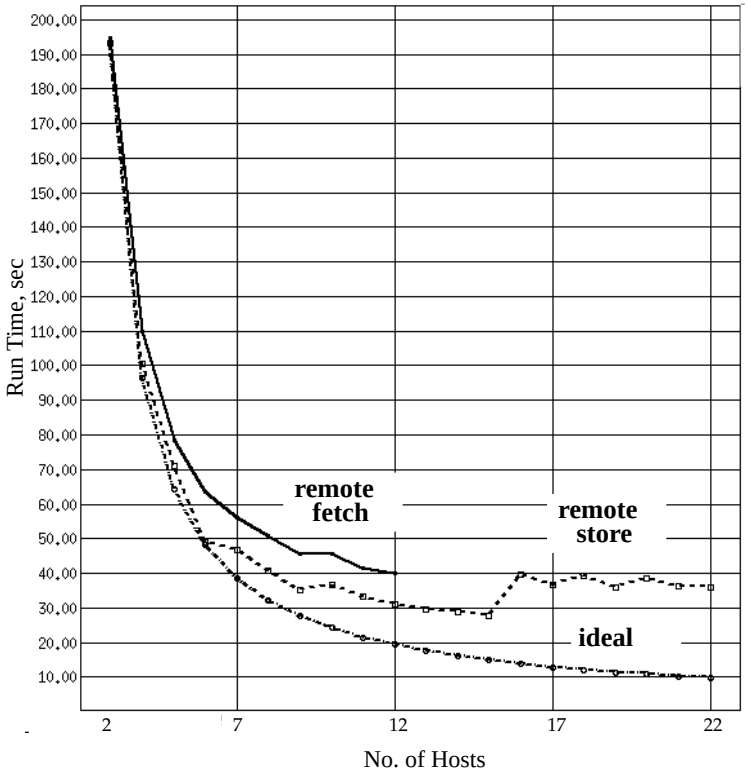


Figure 2: Run Times of the Dirichlet Problem as a Function of the Number of Hosts

5.2 Zooming of Still Images

This mEDA application is used for parallel zooming of RLE (Run Length Encoded) image files. The RLE format was developed at the University of Utah to compress and store multi-level raster images. An RLE image is stored in scanline form, with the data for each colour of the scanline grouped together. Runs of identical pixel values are compressed into a count and a value. An RLE file consists of two parts: a header with information about the image (e.g. size, number of colours, background colour value, etc.) and scanline data.

The zooming program is constructed from three types of tasks: *distributor, worker* and *collector*. The input file, number of workers and the zoom factors are defined in the command line of the distributor. The distributor task, which contains one object, starts first and spawns a number of zoom workers and one collector. The distributor reads the RLE file, divides it into scanlines then distributes them to the workers. The header and the skip line instructions of the RLE file are directly passed to the collector. Zoom workers multiply the scanlines by *x* factor to zoom the image on the *x*-axis and send the result to the collector. The collector receives the zoomed scanlines out of order, therefore each scanline has a unique number which is used to reconstruct the output image. The collector writes each scanline to an output file a number of times equal to the *y* factor, to zoom the image on the *y*-axis.

Figure 3 illustrates the data communication pattern for the zooming program. The run times of the program as a function of the number of hosts is depicted in Figure 4. In this experiment, the size of the input image file is 762 Kbytes and the x and y factors are 3.

The zooming task has an irregular communication pattern and the size of its data packets vary from tens of bytes up to several kilobytes. Furthermore, the ratio of communication to computation is high. The graph of Figure 4 illustrates that for this type of problems network latency becomes the dominant factor preventing any significant improvement in performance. The Figure shows that there is a slight speed up for up to 4 workstations, after that, the processing time is almost constant and increases from 13 hosts upwards.

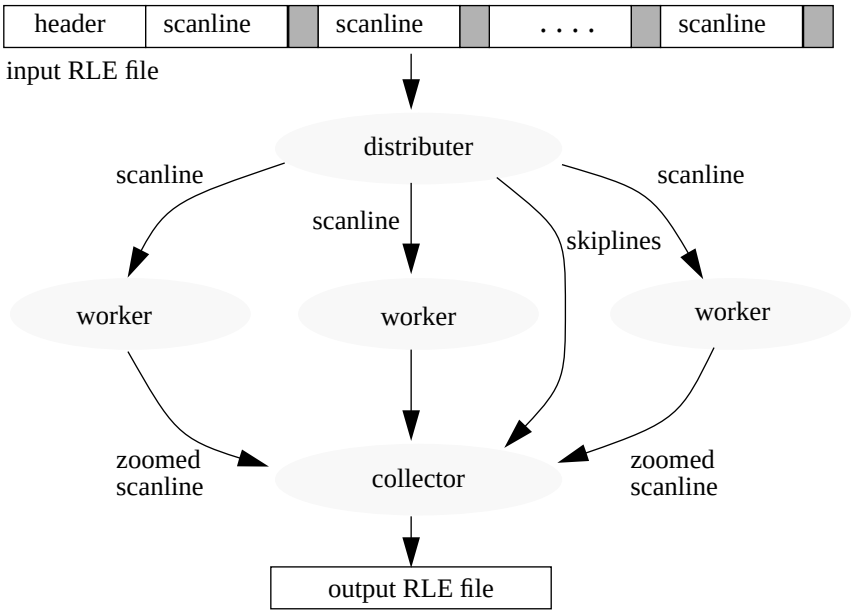


Figure 3: Communication Scheme of the Zooming Task

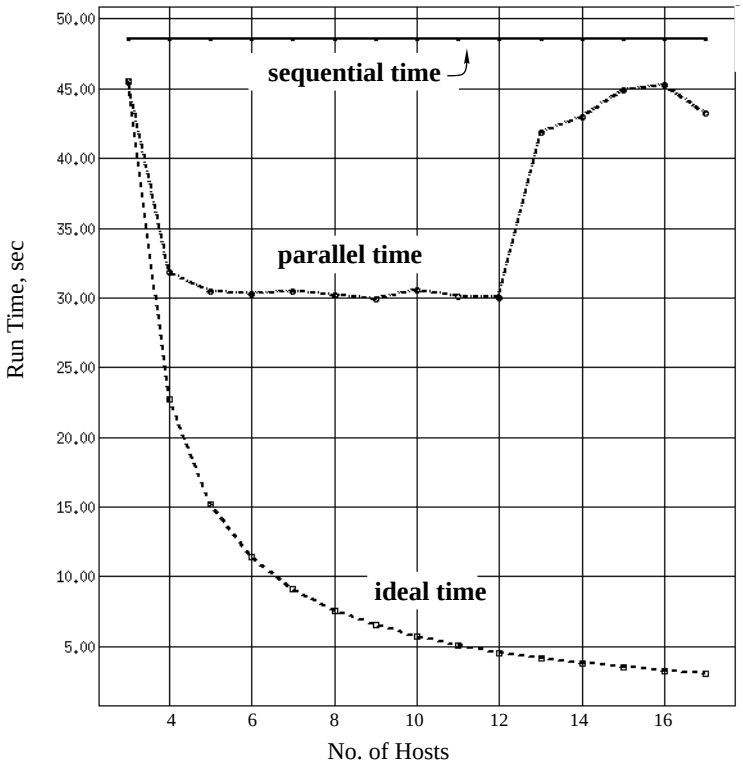


Figure 4: Run Times of the Zooming Task as a Function of the Number of Hosts

5. 3 A Pipelined MPEG Player

This section describes the implementation of a video decoder-player based on the MPEG-1 video compression technique [4, 5] described in Appendix 4. The software decoder used in our system was developed at the University of California-Berkeley [6]. It decodes MPEG-1 video streams and displays them in an X-window environment on an 8, 24 or 32 bit display. Several public domain programming extensions of this application were designed to provide a more flexible user interface. However, most of these attempts are focused on simple run-time commands like stop, rewind and step actions. For this reason, we developed a flexible graphical user interface which is described in Appendix 6.The first step in developing an MPEG decoder-player for the mEDA environment was to divide the application into a two stage pipe: the decoder stage and the player stage, as illustrated in Figure 5.

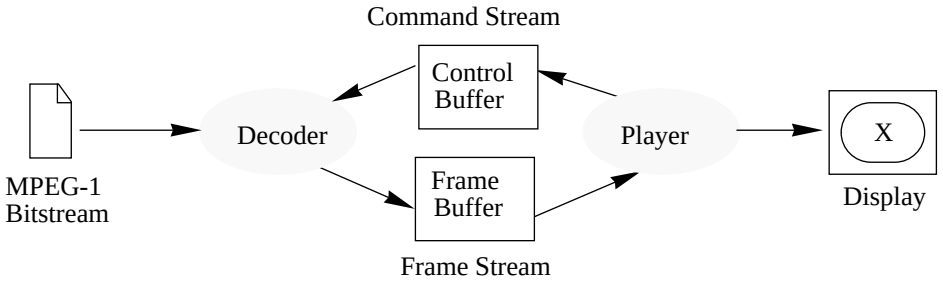


Figure 5: The Pipelined MPEG Video Player

Input to the decoder is MPEG-1 bitstream. The output is a frame stream in YC_rC_b space. This stream is directed to the player, which dithers and displays it in an X-window. To support the decoder-player interaction s-store and x-fetch operations are used. These operations create an s-data channel between the decoder and the player.

The flow of data from decoder to player consists of Y, C_r, C_b frames. The decoder parses the input MPEG bitstream, produces decoded frames and stores them into shared memory using *s-store* operations. The player reads these frames from shared memory using *x-fetch* operations and then dithers and displays them in an X-window. The user controls the player-decoder system through the following functions:

- *Start*, to start the decoder after loading a movie.
- *Pause*, to stop the player temporarily and resume later.
- *Rewind*, provides access to a random point in the video.
- *Search*, search the movie for one or more keywords.
- *Show*, display the frame associated with the given keywords

In order to synchronize interactions between the decoder and the player, the player must control the decoder through a feedback link. The player can issue control functions to the decoder in order to synchronize its operation. The following functions control the decoder: *Play, Position, Quit* and *Mode* (see Appendix 5).There are two ways to organize this control flow:

- After each frame, the decoder checks the control buffer to see if the player has changed the current command. To test the control buffer the decoder uses *x-fetch* operation. The player is responsible that after each frame either to repeat the current command without arguments or to store a new command in the command buffer.

- This is similar to the first approach, except that the decoder tests the command buffer after each *nth* frame is produced (*n* can be defined by the usr as a *mode* parameter).

This application was designed to run on mEDA using three hosts. However, it is possible to parallelize it even further by decomposing the decoding stage. The main purpose of this work was to test the EDA parallel computational model using an interactive image processing application.

6 Conclusions

In this report we presented the mEDA programming environment, which supports multithreaded parallel computations on the level of C-functions within a PVM task. mEDA can be regarded as an extension of the PVM system.

We demonstrated through three applications the correctness of the concepts of EDA as a parallel computation model with specific communication and synchronization technics.

We observed that computationally intensive applications, like partial differential equations, demonstrate a relatively linear speed up when the number of hosts is increased. On the other hand, communication intensive applications, like image processing, behave poorly when distributed on a large number of hosts.

The PVM platform for mEDA was configured on a number of workstations connected by the Ethernet, therefore the communication network constitutes a performance bottleneck which can undermine all gains from parallel execution. Furthermore, it becomes difficult to estimate actual execution and communication times because of other network traffic, consequently, it is difficult to support real-time applications.

Our work on the mEDA programming environment is continuing. Currently the PVM system can be ported to about 35 different platforms [2], including real multiprocessors, like Transputers nets. Our future plans include the implementation of mEDA on real multiprocessors and the development of real-time mechanisms for mEDA.

Appendix 1: A PVM Example

This Appendix contains an example of a PVM program, calculating of sum of n integer values. The parent task, called *sum_ar* starts first, It reads input *n* and generates an array of *n* integer values. The root task divides this array on two parts and sends them to two spawned child tasks for processing. Upon receiving the data, a child task decides whether to calculate a sum and send it back to its parent or to spawn two additional tasks and send them values for processing. Leaf tasks return their partial sums to their parent. An intermediate task waits for results from its children, sums the received values and sends a result to its parent. The root task prints the total sum.

```
#include <stdio.h>
#include <stdlib.h>
#include <pvm3.h>

#define ENCODE    PvmDataRaw
#define MAXSIZE   256
#define InData    98
#define ReturnSum 99

char    *me = “sum_ar”;
return_sum();
create_children();

main()
{
        int     ndata;
        register intptid;
        register inti = 0;
        register int*data;
        pvm_mytid();
        if ((ptid = pvm_parent()) < 0) { /* I am the root */
                do {
                        printf(“How many data (2-%3d)? “, MAXSIZE);
                        scanf(“%d”, &ndata);
                        if (ndata > MAXSIZE || ndata < 2)
                                puts(“Be attentive, please!\n”);
                } while (ndata > MAXSIZE || ndata < 2);
                data = (int * )malloc(ndata * sizeof(int));
                for ( ; i < ndata; i++)
                        data[i] = i + 1;
                create_children(data, ndata);
                printf(“I got %d\n”, return_sum());
        } else {        /* I am a child  */
                pvm_recv(ptid, InData);
                pvm_upkint(&ndata, 1, 1);
                data = (int * )malloc(ndata * sizeof(int));
                pvm_upkint(data, ndata, 1);
                switch (ndata) {
                case 2:
                        data[0] += data[1];
                        break;  /*  I am a leaf */
                case 1:
                        break;  /*  I am a leaf */
                default: /*  I am an intermediate node */
                        create_children(data, ndata);
                        data[0] = return_sum();
                }
                pvm_initsend(ENCODE);
                pvm_pkint(data, 1, 1);
                pvm_send(ptid, ReturnSum);
        }
        pvm_exit();
        exit;
} /* main */
create_children(data, ndata)
```

```
int     *data;
int     ndata;
{
        register inti = 0;
        int     tids[2];
        int     n;

        pvm_spawn(me, NULL, 4, NULL, 2, tids); /* spawn 2 children */
        for ( ; i < 2; i++) {
                pvm_initsend(ENCODE);
                n = ndata / 2 + ndata % 2 * i;
                pvm_pkint(&n, 1, 1);
                pvm_pkint(&data[ndata/2*i], n, 1);
                pvm_send(tids[i], InData);
        }
} /* create_children */


return_sum()
{
        register inti = 0;
        int     d[2];

        for ( ; i < 2; i++) {
                pvm_recv(-1, ReturnSum);
                pvm_upkint(&d[i], 1, 1);
        }
        return d[0] + d[1];
} /* return_sum */
```

Appendix 2: The mEDA Library[1]

The mEDA library *libeda.a* provides a C-interface between mEDA and applications. It contains the following functions and macros which are listed in alphabetical oder.

`eda_exit` - a macro to exit mEDA.

Synopsis

```
eda_exit(int flag)
```

Parameter

`flag` - integer specifying exit option. If flag is non-zero, then calling task requests to kill all other tasks running on PVM.

Description

The macro `eda_exit` generates an exit request to the kernel, which decrements the semaphore. The calling task becomes suspended until an exit permission is received. At this stage the task can service any remote requests to access its shared memory. If `flag` is non-zero, then `eda_exit` generates an exit-kill request to the kernel, which in turn kills all tasks running on PVM, excluding the calling task, and sends an exit permission to it.

`eda_exitthread` - a macro used to realease the discriptor of the calling object and to perform a context switch.

Synopsis

```
eda_exitthread
```

Description

Call to the `eda_exitthread` macro should be the last statement of the thread of an object. It realeses the discriptor of the calling object and performs a context switch, i.e. it extracts a ready object from the head of the ready queue. If there are no ready objects, `eda_exitthread` tests the PVM receive buffer for any remote requests that may have arrived. If a given task does not have any objects and no remote requests have arrived, `eda_exitthread` performs a non-local goto to the next statement after `eda_start` macro, which is usually: `eda_exit`.

`eda_*fetch` - local fetch macros. Each macro fetchs a value of a shared variable allocated in the calling task to a local variable.

Synopsis

```
eda_xfetch(int ac, char *svp, char *lvp, int size)
eda_ifetch(int ac, char *svp, char *lvp, int size)
```

Parameters

`ac`- integer specifying type of access: direct or inderect.
`svp` - pointer to the shared variable to be fetched.
`lvp` - pointer to the local varibale to be stored.
`size` - integer specifying the size of fetched data.

Description

These macros provide a general-purpose communication and synchronization package, based on the EDA parallel model. They can be used to fetch the value of a shared varible pointed by `svp` into a local variable pointed by `lvp`. The size of fetched value is defined by `size`. If the type of access `ac` is non-zero then the value field of shared varible contains a pointer, otherwise the size of the value field must be not less then `size`.

`eda_xfetch` is a blocking communication macro supporting mutually exclusive interprocess communication, as well as data streams between mEDA objects. It extracts a value from the shared variable specified by `svp` into the local variable specified by `lvp`. If the shared variable is empty the macro inserts the object discriptor into the waiting queue. The calling object becomes suspended until the shared variable becomes full by the matching `eda_xstore`, `eda_sstore`, `eda_istore` or `eda_ustore`. Once the value of shared variable has beed extracted, its state becomes empty.

`eda_ifetch` is blocking macro supporting multiple-read from shared variables. It copies a value from the shared variable specified by `svp` into the local variable specified by `lvp`. If the shared variable is empty the macro inserts the object discriptor into the waiting queue. The calling object becomes suspended until the shared variable becomes full by the matching `eda_xstore`, `eda_sstore`,

1. This Appendix contains the description of only the most important macros and functions from the mEDA library.

`eda_istore` or `eda_ustore`. Once a value of shared variable has beed copied, its state remains full.

`eda_inpvm` - function enrolls calling task in mEDA.

Synopsis

```
int *stids = eda_inpvm(int *mytid, int *ntasks, int *mynum, int
          *ptid, char *mem)
```

Parameters

`mytid` - integer returning the task identifier of the calling task.
`ntasks` - integer returning the number of siblings of the calling task (see `eda_task`).
`mynum` - integer returning the instance number of the calling task in the group of its siblings.
`ptid` - integer returning the task identifier of the parent of the calling tasks.
`mem` - pointer to a structure containing shared variables allocated in the calling task.

Return value

`stids` - integer array formed from the tids of siblings of the calling task.

Description

The routine `eda_inpvm` enrols the calling tasks into mEDA. It returns information about parent of the calling task and its siblings (see `eda_task`).

`eda_mynum` - this macro returns the instance number of the calling mEDA object in a group of objects sharing the same thread.

Synopsis

```
eda_mynum
```

Description

This macro returns a unique instance number of the calling mEDA object in the group of objects sharing the same thread. Instance numbers start at 0 and count up. This macro can be called multiple times.

`eda_obj` - a macro to create the discriptor of an object.

Synopsis

```
eda_obj(void(*thread)(), int nobj)
```

Parameters

`thread` - the name of a thread of the object to be created (which is a C function).
`nobj` - integer specifying the number of copies of the object to create.

Description

The macro `eda_obj` creates `nobj` object descriptors, which contain data defining contexts of objects during their life cycle. At the end `eda_obj` insertes created discriptors into the ready queue. All created objects share the same C-function called `thread`. Each object obtains a unique instance number which is stored in its descriptor. If required, an object can get its instance number using the `eda_mynum` macro.

An object descriptor has the following structure:

```
struct edat_minobj {
     struct edat_minobj *link;
     int oid;
     char *waddr;
     unsigned wsize;
     int widrc;
     int wcop;
     jmp_buf rea, ret;
}
```

Where:

`link` - is a pointer used to insert a given object into a queue (e.g. ready queue or waiting queues) when the object is not active.
`oid` - is the instance number in a group of objects sharing the same thread.
The next four fields `waddr`, `wsize`, `widrc` and `wcop` are used to store the following arguments of protected memory operation when an object is suspended:
`waddr` - address of a shared varible.
`wsize` - size of data

`widrc` - addressing mode (direct or indirect)
`wcop` - operation code
`rea`, `ret` - reactivation arrays: These arrays are used to save and restore the stack and CPU registers during object context switch

`eda_r*fetch` - remote fetch macros. Each macro fetchs a value of a shared variable allocated in another task to a local varible.

Synopsis

```
eda_rxfetch(int ac, int tid, int offset, char *lvp, int size)
eda_rifetch(int ac, int tid, int offset, char *lvp, int size)
```

Parameters

`ac`- integer specifying type of access: direct or inderect.
`tid` - integer specifying tid of the destination task.
`offset` - integer specifying an offset to a shared variable in the destination task.
`lvp` - pointer to the local varibale to be stored.
`size` - integer specifying size of fetched data.

Description

`eda_rxfetch` is a blocking communication macro supporting mutually exclusive interprocess communication, as well as data streams between mEDA objects. It sends a request to the remote task specified by `tid` to extract a value from the shared variable specified by `offset`. The macro blocks the calling object until a value has arrived from the remote task wich sends a reply as soon as the extract operation is performed, i.e. the shared variable is full. Otherwise the fetch request is queued until the shared variable becomes full by the matching `eda_xstore`, `eda_sstore`, `eda_istore` or `eda_ustore`. Once a value of the shared variable has beed extracted, the remote task resets its state to empty. Once the value has arrived and stored in the local variable, the calling object becomes ready.

`eda_rifetch` is blocking macro supporting multiple-read from shared variables. It sends a request to the remote task specified by `tid` to copy a value from the shared variable specified by `offset`. The macro blocks the calling object until a value has arrived from the remote task wich sends a reply as soon as the extract operation is performed, i.e. the shared variable is full. Otherwise the fetch request is queued until the shared variable becomes full by the matching `eda_xstore`, `eda_sstore`, `eda_istore` or `eda_ustore`. Once the value has arrived and stored in the local variable, tha calling object becomes ready.

`eda_r*store` - remote store macros. Each macro stores the value of a local variable to a shared varible, allocated in another task.

Synopsis

```
eda_rxstore(int ac, int tid, int offset, char *lvp, int size)
eda_rsstore(int ac, int tid, int offset, char *lvp, int size)
eda_ristore(int ac, int tid, int offset, char *lvp, int size)
eda_rustore(int ac, int tid, int offset, char *lvp, int size)
```

Parameters

`ac`- integer specifying type of access: direct or inderect.
`tid` - integer specifying tid of the destination task.
`offset` - integer specifying an offset to a shared variable in a structure of a shared memory at the destination task.
`lvp` - pointer to the local varibale to be stored.
`size` - integer specifying a size of stored data.

Description

These macros provide a general-purpose communication and synchronization package, based on the EDA parallel model. They can be used to store the value of the local varible pointed by `lvp` into a shared variable defined by its offset `offset` in the structure of the shared memory in the remote task specified by `tid`. The size of stored value is defined by `size`. If the type of access `ac` is non-zero then the value field of shared varible contains a pointer, otherwise the size of the value field must be not less then `size`.

`eda_rxstore` is a blocking communication macro supporting mutually exclusive interobject communication. It sends the value of the local variable `lvp` to the mEDA task specified by `tid` with a request to store it into shared variable specified by `offset`. The macro blocks the sending object until

acknowledgment has arrived from the remote task. The remote task receving the request sends an acknowledgment as soon as the store operation has been performed. If the shared variable is full, then the store request is queued as a “hidden” object until the shared variable is emptied by the matching `eda_xfetch`. Once acknowledgment has arrived, the calling object becomes ready.

`eda_ristore` is a non-blocking macro supporting write-once shared objects. It sends the value of a local variable `lvp` to remote mEDA task specified by `tid` with a request to store it into the shared variable specified by `offset`. Computation on the sending object continues. This EDA operation is performed as if the shared variable was empty. Otherwise the operation is ignored.

`eda_rsstore` is a non-blocking macro supporting data streams between mEDA objects. It sends the value of a local variable `lvp` to remote mEDA task specified by `tid` with a request to store it into the shared variable specified by `offset`. Computation on the sending object continues. This EDA operation is performed as if the shared variable was empty. Otherwise the remote task creates a buffer object to save received value. The buffer object is queued until the the shared variable is emptied by the matching `eda_xfetch`.

`eda_ustore` is a non-blocking macro supporting unconditional updating of shared variables. It sends the value of a local variable `lvp` to remote mEDA task specified by `tid` with a request to store it into the shared variable specified by `offset`. Computation on the sending object continues. The remote task always executes store requests independent of status of the shared variable.

`eda_runthread` - a macro used to save the initial state of the calling object before it is inserted into ready queue.

Synopsis

```
eda_runthread
```

Description

Call to the `eda_runthread` macro should be the first statement of a thread of an object. It saves the initial state of the stack and CPU registers of the calling object during creation of its discriptor. `eda_runthread` uses `setjump`(3V) and `longjmp`(3V) routines to save the state and to return. When the object becomes running, the next statement in its thread is performed.

`eda_shared` - this macro constructs a shared variable type.

Synopsis

```
eda_shared(type)
```

Parameters

`type` - type of the value of a shared varible.

Description

This macro constructs a type of a shared varible using the type of its value, e.g. the following variable definition:

```
eda_shared(float) x;
```

declares the variable `x` to be a shared variable with an integer value.

A shared varible has the following structure:

```
struct edat_minsv {
     int state;
     struct edat_minobj *head, *tail;
     double bound;
     type *value;
}
```

The structure of the shared variable contains the following fields:

`state` - integer field used to indicate state of the shared variavle: full or empty.

`head, tail` - object pointers used to form a fifo queue of suspended objects waiting to access the shared variable.

`bound` - an empty field used to align the boundary of the value field.

`value` - it contains the actual value or a pointer to the value of the shared variable.

`eda_start` - a macro to start mEDA context switch control.

Synopsis

```
eda_start
```

Description

The macro `eda_start` starts context switch control. It extracts the first ready object, a C function,

from the ready queue. This object becomes running.

eda_*store - local store macros. Each macro stores a value of a local variable to a shared varible, allocated in the calling task.

Synopsis

```
eda_xstore(int ac, char *svp, char *lvp, int size)
eda_sstore(int ac, char *svp, char *lvp, int size)
eda_istore(int ac, char *svp, char *lvp, int size)
eda_ustore(int ac, char *svp, char *lvp, int size)
```

Parameters

`ac`- integer specifying type of access: direct or inderect.
`svp` - pointer to a shared variable where value of local variable is stored.
`lvp` - pointer to the local varibale to be stored.
`size` - integer specifying size of stored data.

Description

These macros provide a general-purpose communication and synchronization package, based on the EDA parallel model. They can be used to store a value of the local varible pointed by `lvp` into a shared variable pointed by `svp`. The size of stored value is defined by `size`. If the type of access `ac` is non-zero then the value field of the shared varible contains a pointer, otherwise the size of the value field must be not less then `size`.

`eda_xstore` is a blocking communication macro supporting mutually exclusive interobject communication. It stores a value of the local variable `lvp` into the shared variable `svp` which is allocated in the calling task. If the state of the shared variable is full then `eda_xstore` inserts the object descriptor into the waiting queue of the shared variable and blocks the calling object causing a context switch. The object is suspended until the shared variable is emptied by the matching `eda_xfetch`. Once the calling object resumes it becomes ready and is inserted into the ready queue.

`eda_sstore` is a non-blocking macro supporting data streams between mEDA objects. It stores the value of a local variable `lvp` into the shared variable `svp` if it is empty. Otherwise `eda_sstore` creates a buffer object to save a copy of the value pointed by `lvp`. The buffer object is queued until the shared variable is emptied by the matching `eda_xfetch`.

`eda_istore` is a non-blocking macro supporting write-once shared variables. It stores the value of a local variable `lvp` into the shared variable `svp` which is allocated in the calling task. This EDA operation is performed if the shared variable is empty. Otherwise the operation is ignored.

`eda_ustore` is a non-blocking macro supporting unconditional updating of shared variables. This operation is performed independently of the state of the shared varible.

eda_task - function starts new mEDA processes on PVM.

Synopsis

```
int *tids = eda_task(char *task, char **argv, int flag, char *where,
          int ntasks)
```

Parameters

`task` - character string containing the executable file name of the mEDA process to be started.
`argv` - pointer to an array of arguments to the executable, excluding the executable name, with the end of the array specified by `(char*)0`.
`flag` - integer specifying spawn option. (see `pvm_spawn`(PVM3)).
`where` - character string containing a host name where to start new mEDA processes.
`ntasks` - integer specifying the number of copies of the task to start.

Return value

`tids` - integer array formed from the tids of the tasks started by the `eda_task` routine.

Description

The routine `eda_task` starts `ntasks` copies of `task`. The host on which the tasks are started is determined by the `where` argument. During spawning a parent task sends the tids of its children to the mEDA kernel and also to each newly spawned child in order to notify them about their siblings. Each child is responsible to get this information using the `eda_inpvm` routine.

Appendix 3: Source Code for the Dirichlet Problem

This Appendix contains the source files of the Dirichlet problem: `memstr.h`, `collector.c` and `partition.c`.

memstr.h:

```
typedef struct {
        float    x[SIZE+2];
} singlecol;

typedef struct {
        int      ni;      /* number of iterations */
        float    upval,   /* value of upper border*/
        loval,   /* value of lower border*/
        lval,    /* value of left border*/
        rval;    /* value of right border*/
} arglist;

/*
 * types (structures) of shared memory
 */

typedef struct {
        eda_shared(singlecol) left; /* s(x) variable */
        eda_shared(singlecol) right;/* s(x) variable */
} mem_partition;

typedef struct {
        eda_shared(singlecol*)  result[SIZE]; /* s(x) variable */
        eda_shared(arglist)     argp;   /* i variable    */
} mem_collector;
```

collector.c:

```
#include <meda2.h>
#include “memstr.h”

mem_collector mymemory;

singlecol *res[SIZE];
singlecol *r;
int      np;
int      width;
arglist     parg;    /* arguments for a partition */

void collector();

main(argc, argv)
int      argc;
char     **argv;
{
        int      i, j, k;

        eda_inpvm(NULL, NULL, NULL, NULL, &mymemory);
        eda_obj(collector(), 1);
        eda_start;
        eda_exit(EDASOS);
        eda_printmtime(NULL);
        printf(“\nnumber of hosts=%d”, eda_hosts(NULL));
        printf(“\nnumber of partitions=%d”, np);
        printf(“\nsize=%d”, SIZE);
        printf(“\nnumber of iterations=%d”, parg.ni);
        printf(“\n--------------------------------\n”);
        for (j = 1, width--; j <= SIZE; j++) {
                for (k = 0; k < np - 1; k++)
                        for (i = 0, r = res[k]; i <= width; i++)
                                printf(“%5.0f”, r[i].x[j]);
                for (i = 0, r = res[np-1]; i <= width + SIZE % np; i++)
                        printf(“%5.0f”, r[i].x[j]);
                printf(“\n”);
```

```
        }
        exit(0);
}


void collector()
{
        static int*ptids;
        static inti, j;

        eda_runthread;
        if (eda_onum == 0) {
                eda_time();
                do {
                        printf(" How many partitions (1-%1d)? ", SIZE);
                        scanf("%d", &np);
                        if (np > SIZE || np < 1)
                                puts("Be attentive, please!\n");
                        else
                                width = SIZE / np;
                } while (np > SIZE || np < 1);

                /*
                 * some more collectors
                 */
                eda_obj(collector(), np - 1);
                ptids = (int * )calloc(np, sizeof(int));
                eda_task("partition", NULL, 2, "SYMM", np, ptids);
                parg.upval = parg.loval = parg.lval = parg.rval = 100.0;
                do {
                        printf(" How many iterations (>0)? ");
                        scanf("%d", &parg.ni);
                        if (parg.ni < 0)
                                puts("Be attentive, please!\n");
                } while (parg.ni < 0);
                eda_istore(0, &mymemory.argp, &parg, sizeof(parg));
                eda_printmtime(NULL);
        }
        eda_xfetch(0, &mymemory.result[eda_onum], &res[eda_onum], sizeof(char * ));
        eda_exitthread;
} /* collector */
```

partition.c:

```
#include <meda2.h>
#include "memstr.h"

mem_partition mymemory;
singlecol  MyCols[SIZE+2];
singlecol NewCols[SIZE+2];
int      *tids;
int      ctid;
int      np;
int      mynum;

#define sz sizeof(singlecol)

void partition();

main()
{
        tids = eda_inpvm(&ctid, NULL, &np, &mynum, &mymemory);
        eda_obj(partition(), 1);
        eda_start;
        eda_exit(0);
        exit(0);
}


void partition()
{
```

```
        static arglist   arg;
        static intwidth;
        static inti, j;
        static singlecol* myCols = MyCols;
        static singlecol* newCols = NewCols;
        static singlecol* aux;
        eda_runthread;
        eda_rifetch(0, ctid, eda_offset(mem_collector, argp), &arg, sizeof(arg));
        if (mynum == np - 1)
                width = SIZE / np + SIZE % np;
        else
                width = SIZE / np;
        for (i = 1; i <= width; i++) {
                myCols[i].x[0]     = arg.upval;
                myCols[i].x[SIZE+1] = arg.loval;
                for (j = 1; j <= SIZE; j++)
                        myCols[i].x[j] = 0;
        }
        if (mynum == np - 1)
                for (j = 0; j < SIZE + 2; j++)
                        myCols[width+1].x[j] = arg.rval;
        if (mynum == 0)
                for (j = 0; j < SIZE + 2; j++)
                        myCols[0].x[j] = arg.lval;
        while (arg.ni--) {
                if (mynum == np - 1) {
                        eda_sstore(0, &mymemory.right, &myCols[width+1], sz);
                } else {
                        eda_rsstore(0, tids[mynum+1],
                            eda_offset(mem_partition, left),
                            &myCols[width], sz);
                }
                if (mynum == 0) {
                        eda_sstore(0, &mymemory.left, &myCols[0], sz);
                } else {
                        eda_rsstore(0, tids[mynum-1],
                            eda_offset(mem_partition, right),
                            &myCols[1], sz);
                }
                eda_xfetch(0, &mymemory.left, &myCols[0], sz);
                eda_xfetch(0, &mymemory.right, &myCols[width+1], sz);
                for (i = 1; i <= width; i++)
                        for (j = 1; j <= SIZE; j++)
                                newCols[i].x[j] = (myCols[i-1].x[j] + myCols[i+1].x[j] +
                                    myCols[i].x[j-1] + myCols[i].x[j+1]) / 4;
                aux = myCols;
                myCols = newCols;
                newCols = aux;
        }
        eda_rsstore(1, ctid, eda_offset(mem_collector, result[mynum]),
            &myCols[1], sizeof(singlecol) * width);
        eda_exitthread;
} /*partition*/
```

Appendix 4: MPEG Coding-Decoding Techniques

The Moving Picture Experts Group, MPEG, was established to produce an international standard for coding moving pictures and its associated audio for digital storage media, like CD-ROMs, DATs, and computer disks [4].

MPEG's activities started in 1988, as an ISO/IEC working group and is currently working jointly with the ITU-TS Study Group 15 "Experts Group for ATM Video Coding."

1. MPEG-1

The first result of these activities is the International Standard *"Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbits/s"* (ISO/IEC CD 11172) which consists of three components: system, video, and audio [5].

The standard ISO/IEC CD 11172 is called MPEG-1 and defines a bitstream for compressed video and audio, optimized to fit into a bandwidth (data rate) of 1.5 Mbits/s. Such a rate could permit numerous applications including video and associated audio on Compact Disc. MPEG is also developing MPEG-2 and MPEG-4 standards.

While the MPEG-1 compression algorithm is optimized for bit rates of about 1.5 Mbit/s, it can perform effectively over a wide range of bit rates and picture resolutions. The video standard does not recommend a particular way of encoding pictures and much flexibility is given to implementers of the standard to use the MPEG syntax to optimize the visual quality and access options. The colour resolution is given particularly high attention so as to support computer applications, games and animation.

Video is represented as a succession of individual pictures or *frames*. Each frame is treated as a two dimensional array of picture elements, or *pixels* [5]

The compression techniques developed in MPEG rely on the Discrete Cosine Transform (DCT) for spatial redundancy reduction and motion compensated inter-frame coding. It exploits the high temporal correlation of video signals and uses information from both the past and the future. The statistics of the resulting information can also be manipulated to further reduce the bit rate through the use of special codes known as Huffman codes.

MPEG-1 starts with a relatively low resolution video sequence of about 352 by 240 points and a frame rate of 30-25 pictures/s (for US and Europe, respectively).

The RGB-frames of a movie are digitally converted to the YUV space or YC_rC_b, where Y is the luminance plane indicating brightness and C_rC_b are the chrominance planes containing colour information. The chrominance is further compressed to 176 by 120 pixels , i.e. by halving the width and height of the C_r and C_b planes. This compression is possible because human eye is less sensitive to chrominance than luminance.

A frame is divided into *macroblocks*. A macroblock consists of 16x16 pixel values of Y and 8x8 pixels of C_r and C_b, respectively. Thus, a macroblock is composed of six 8x8 pixel *blocks*.

The basic scheme of the MPEG-1 encoding process is to predict motion from frame to frame in the temporal direction, and then to use DCT to organize the redundancy in the spatial directions. The DCT's are done on blocks and the motion prediction is done in the luminance (Y) plane on macroblocks. In other words, an encoder looks for a macroblock closely matched to the macroblock being coded in a previous or future frame.

The translated block from the known frame becomes a prediction for the block in the frame to be encoded. This technique relies on the fact that within a short sequence of frames of the same general scene, many objects remain in the same location while others move only a short distance [5].

There are three types of coded frames:

- Intra frames, or I-frames.
- Predicted frames, or P-frames.
- Bidirectional frames, or B-frames.

I-frames are coded as still images and do not contain any past history. Such frames are used as direct access points during the decoding process.

P-frames are predicted from the most recently constructed I or P frame. Each macroblock in a P-frame can either come with a difference vector of DCT coefficients, if a close match with a macroblock from the last I or P-frame was found, or it can just be "intra" coded like an I- frame if there was no good match.

B-frames are predicted from the closest two I- or P-frames, one in the past and one in the future. The encoder searchs for matching macroblocks in those frames, and tries to apply three different coding technics to choose which works best: using the forward vector, the backward vector, and averaging the two macroblocks from the future and past frames, and subtracting that from the macroblock being coded. If none of those works well, the encoder can intra-code the macroblock.

After motion compensation, the encoder transforms blocks using DCT, which converts a block of pixel values to an 8x8 block of horizontal and vertical spatial frequency coefficients. To reconstruct the block, the decoder uses inverse DCT.

The DCT coefficients are quantized and hopefully, many of them will end up being zero. The quantization can change for every macroblock.

The results of the encoding steps, which includes the DCT coefficients, motion vectors, and quantization parameters is Huffman coded using fixed tables.

A coded MPEG bitstream has a relatively regular hierarchical structure consisting of six layers (see Figure 6).

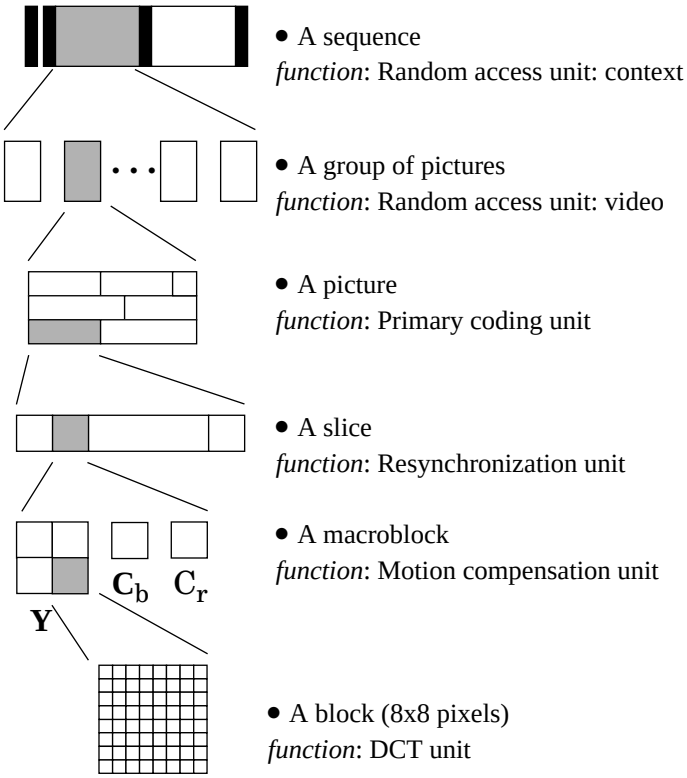


Figure 6: The MPEG-1 Bitstream

Decoding involves the following steps:

- ❑ Huffam decoding
- ❑ Inverse quantization
- ❑ Inverse DCT
- ❑ Motion compensation for P and B frames
- ❑ Displaying

Motion compensation requires storage of at least two frames; one from the future and another from the past, to reconstruct the current B or P frame.

The display step also requires buffering at least five frames because of the order of the input bitstream. The sequence of frames, in *display order*, is typically a mix of I, P, and B-frames:

I	B	B	**P**	B	B	**P**	B	B	**P**	B	B	**I**	B	B	**P** ...
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15 ...

whereas the sequence of decoded frames in *bitstream order* (processing order) would be:

I	**P**	B	B	**P**	B	B	**P**	B	B	**I**	B	B	**P**	B	B ...
0	3	1	2	6	4	5	9	7	8	12	10	11	15	13	14 ...

The bitstream order differs from the display order because of the synchronization requirements of the decoding and displaying processes.The decoder starts processing I-frame 0, then decoding P-frame 3 while displaying I-frame 0. Both frames are stored in memory, and B-frames 1 and 2 are decoded. Depending on the decoding algorithm, the decoder may display the processed blocks of B-frames 1 and 2 as it is decoding them, or it may display a B-frame only after the whole frame is decoded. Then P-frame 3 is displayed while P-frame 6 is decoded, and so on.

There are 12 frames from I to I. This is based on a random access requirement that an I-frame may occur at least once every 0.4 - 0.5 seconds. The ratio of P-frames to B-frames is based on experience. Typically, not more than two B-frames are inserted between each pair of I- or P-frames.

2. MPEG-2

MPEG is developing the MPEG-2 video standard, as well as MPEG-2 audio and system standards, which specifies the coded bitstream for high-quality digital video. As a compatible extension, MPEG-2 video builds on the completed MPEG-1 video standard, by supporting interlaced video formats and a number of other advanced features, including features to support HDTV.

As a generic international standard, MPEG-2 video is being defined in terms of extensible profiles, each of which will support the features needed by an important class of applications.

The MPEG-2 main profile was defined to support digital video transmission in the range of about 2 to 15 Mbits/s over cable, satellite, and other broadcast channels, as well as for Digital Storage Media (DSM) and other communications applications. This will be a compatible extension of MPEG-1, meaning that an MPEG-2 video decoder will decode MPEG-1 bitstreams (see Table 1).

Table 1: MPEG Parameters

Parameters	MPEG-1	MPEG-2
Compression	50-1 to 200-1	100 -1
Bit rate (Mbit/s)	1.2 to 1.5	2 to 15
Resolution[a]	640 x 480	704 x 480

a. Analog cable TV resolution is 500 x 480

Also, like MPEG-1, MPEG-2 will support interoperability with the CCITT H.261 video telephony standard. The development of further profiles is already well under way.

3. MPEG-4

Work on a new MPEG initiative for very low bit rate coding of audiovisual programs has been approved by unanimous ballot of all national bodies of ISO/IEC JTC1. This work is scheduled to result in a draft specification in 1997. When completed, the MPEG-4 standard will enable a whole spectrum of new applications, including interactive mobile multimedia communications.

Appendix 5: Specification of the Pipelined MPEG Player

1. Decoder

❑ Input-Output

Input data: file `filename.mpg` is an MPEG video bitstream.

Output data: arrays of pixel values representing frames of the movie in YUV space, with frame numbers and direct access points.

❑ Main Decoder Functions

The decoding algorithm performs the following main tasks:

Read input file.

Decode the bitstream until frame data is encountered.

Store the starting point of data in shared memory.

❑ Algorithm

```
while not quit {
    if mode
        {fetch mode information from protected memory, do settings}
    if play
        {read input file, decode bit-stream, produce array of pixel values of one frame, store it along with current
        frame number and current direct access point in shared memory}
    if position
        {fetch frame number and position pointer from protected memory, read input file and decode bitstream,
            until required frame is reached, produce array of pixel values (1 frame), store it in protected memory}
}
 quit.
```

2. Player

❑ Input-Output

Input data: file `filename.mpg`. These are arrays of pixel values in the YUV space received from decoder through protected memory, and possibly some index information (if the video is already indexed).

Output data: X images, depending on the dither mode and visual class of the display, and possibly some text.

❑ Main Player Functions

The player algorithm performs the following main tasks:

Fetch starting point of frame data from shared memory.

Read content information from content data structure.

❑ Algorithm

```
while not quit {
    if play /* play mode */
        {fetch array of pixel values (one frame) from shared memory, display the frame}
    if scan /* contents mode */
        {fetch array of pixel values (one frame) from protected memory, get next frame number and nearest direct
        access point from the contents data structure, store them in shared memory and display frame}
    if index /* index mode */
        { get the index text or keywords, store the index along with current frame number and current direct access
        point in contents structure}
    if contents /* contents mode */
        {display contents of movie}
    if show /* search mode */
        {from the contents structure, find pointer to the frame whose index was selected, display the frame}
    if search /* search mode */
        {find key words in contents structure, get position point}
}
 quit.
```

Appendix 6: Graphical User Interface

The graphical user interface, GUI, for the pipelined MPEG-1 decoder-player is designed with Open Windows Developer's Guide, *devguide* [7].

Start Up

The MPEG player is activated by typing the command:

`mpeg_pplay [`*file*`] [-h` *host*`]`

where *file* contains MPEG-1 compressed video data, and *host* is a string containing name of the host on which the decoder is run.

After starting, the player's base window appears, as shown in Figure 7.

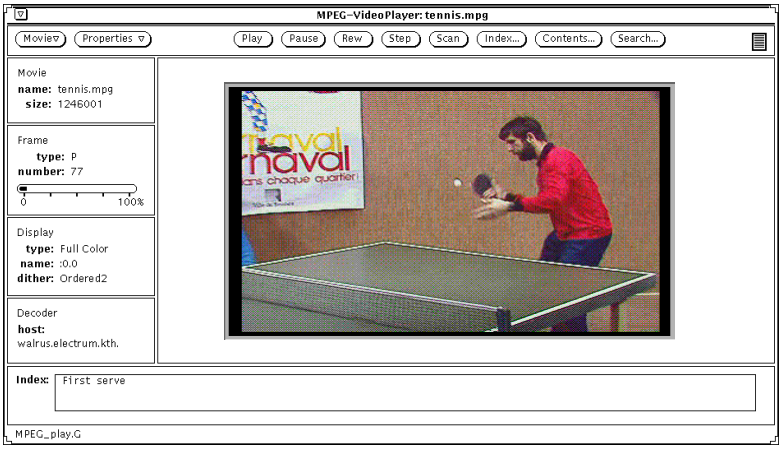


Figure 7: The MPEG Player GUI

This window contains the back screen, the play screen and the control and information fields required to operate it. The play screen is invisible until a movie is loaded. Controls are located at the top of the window and information fields at the left side and at the bottom of the base window. Here is a short description of the MPEG player's controls.

- ❑ The `Movie` menu button with menu items used to load, save, unload a movie and quit the player.
- ❑ The `Properties` menu button which enables the user to set up player properties, such as host names for the decoder, colour of the back screen, dither type and several other control options.
- ❑ The set of buttons controlling the player are: `Play, Pause, Rewind, Step and Scan`.
- ❑ The `Index` button enables the user to define an index for the current frame and add it to the contents structure of the movie.
- ❑ The `Contents` button enables the user to open the contents window for reading and editing the index contents of a movie.
- ❑ The `Search` button is used to find a frame in the movie whose index matches one or more keywords.
- ❑ The `Drag` and `Drop Target` provides a location for loading an MPEG file into the player.

All control buttons (`Play, Pause, Rewind, Step, Scan, Index, Contents and Search`) are inactive until a movie is loaded.

The player information fields appear along the right side and bottom of the base window (Figure 8). They give the user information on the current state of the player.

- ❑ The `Movie` field displays name and size of the movie.
- ❑ The `Frame` field displays the type (I, P or B) and number of the current frame.
- ❑ The `Display` field gives the visual class and name of display. It also indicates current type of dithering.
- ❑ The `Decoder` field displays name of the host, where the decoder is running.
- ❑ The `Index` field displays the current index of the loaded movie.

To exit the player, the `Quit` button is clicked (in the `Movie` menu, Figure 13). If the movie's index has been changes, the system gives the user two alternatives before quitting: `Cancel`, were the user can continue running the payer, or `Discard Changes`, were any changes to the index are discarded before quitting.

Properties Set Up

After starting the player, the user selects the host on which to run the decoder. This is done by choosing `Decoder...` from the `Properties` menu (Figure 9). The host window appears, and the user writes the host name (Figure 10). Cur-

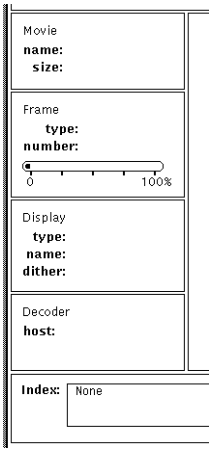


Figure 8: GUI Information Fields

Figure 9: Properties Menu

rent host is the default choice. The user must chose a host *before* loading a movie. In addition, the user *can no*t change the decoder's host until the current movie is unloaded.To set up other player properties the user chooses `Display...` from the `Properties` menu. The display window is shown in Figure 11.

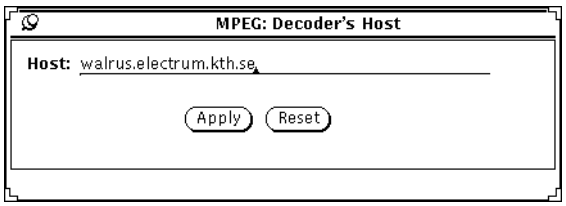


Figure 10: Decoder's Host Window

The properties include:

- Options (nonexclusive menu).
- Dither type (exclusive menu).
- Number of colours for YC_rC_b planes.

The `Options` enable the user to select the following properties:

- `No B Frames`: causes the player not to display any B frames.
- `No P Frames`: no display of P frames.
- `Loop Play`: the player loops back to the beginning of the movie after reaching the end.
- `Quiet`: suppresses printing of status information in `stderr`.
- `EachStat`: causes statistics to be displayed in `stderr` after each frame.
- `ShMem Off`: turns shared memory off.
- `No Index`: turns indexing off.
- `No Display`: dithers, but does not display.

The exclusive `Dither` setting is used to choose one of the dithering types [7]. Berkeley's player supports 13 types of dithering.

Three sliders for `Luminance, Chr.Red` and `Chr.Blue` enable the user to set the number of colours assigned to the Y, C_r and C_b planes, respectively (Figure 11). The product of these values should be less than the number of colours on the display. The player checks and corrects the user's values.

The `Back Screen` field from the `Properties` menu is used to choose a colour for the back screen. The back screen

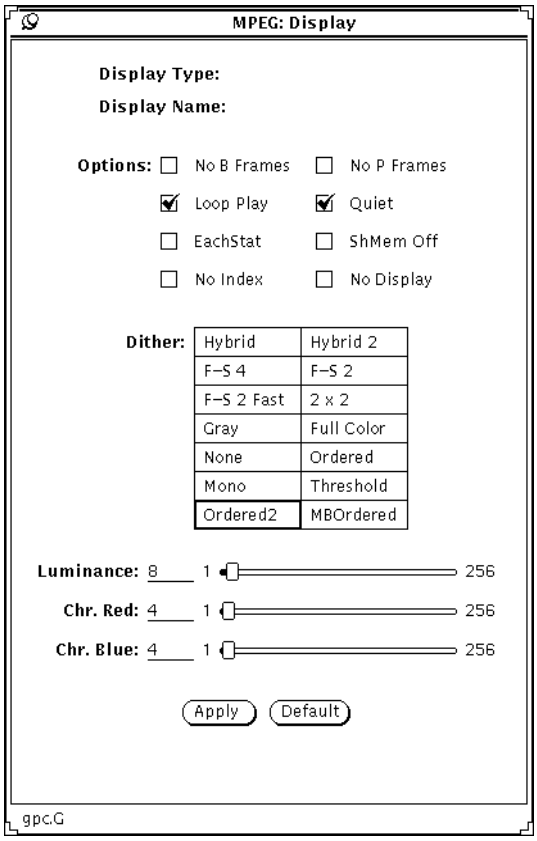


Figure 11: Display Properties Window

colour window is shown in Figure 12. The scrolling list of this window offers the user a choice of colours whose names and shades appear at the bottom of the window. The user can either click or type the desired colour (in the `colour` text field). Once a colour is chosen, the user can apply it by clicking the `Apply` button. To reset the chosen colour, the `Reset` button is used.

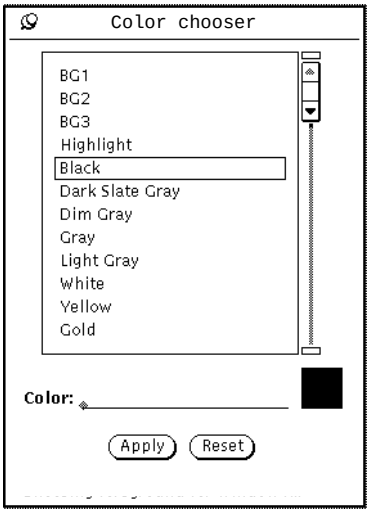


Figure 12: Colour Chooser

Loading and Saving Movies

There are two ways to load a movie:

- From the MPEG file chooser using `Load` (the `Movie` menu, Figure 13). Type or select a file name and press the `Load` button in the load window (Figure 14).
- From the Open Windows file manager. Drag the file to its target in the MPEG player window or drag the file to the play screen.

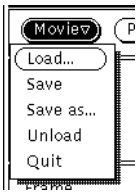


Figure 13: Movie Menu

There are also two ways to save a movie after modifying its index:

- ❑ Overwrite a movie with the same name choosing the `Save` command from `Movie` menu.
- ❑ Save a movie with a new name by choosing `Save As...` from the `Movie` menu.

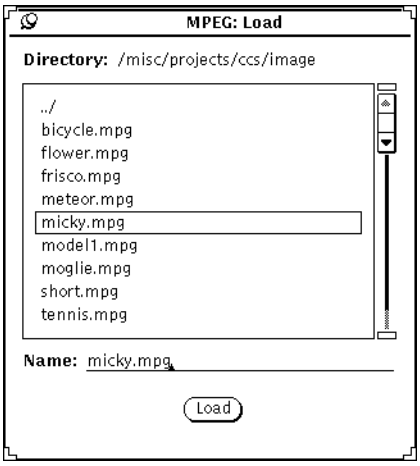


Figure 14: Load Window

Choosing `Unload` from the `Movie` menu the user can unload a movie, after which it is possible to change the decoder's host name, if needed.

Playing Movies

The player should work like a video player, with `Play, Pause` and `Rewind` buttons located at the top of the player's base window (Figure 15). To play a movie frame by frame, one can use the `Step` button. If the user wants to play only indexed frames, stopping after each such frame, the `Scan` command is used.

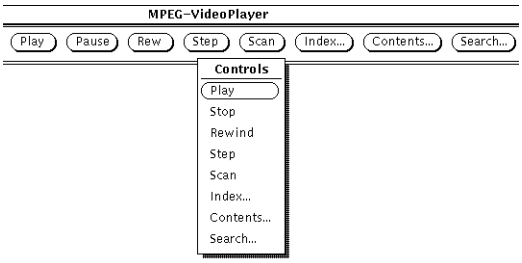


Figure 15: MPEG Player Controls

All control buttons are duplicated by the popup `Controls` menu, illustrated in Figure 15. The control buttons and the `Controls` menu are disabled until a movie is loaded.

If the `Loop Play` option is defined in player properties, the movie is played nonstop.

The fields `Movie, Frame` and `Index`, illustrated in Figure 8, display information on the name and size of the movie, type and number of current frame and the current index. To display the complete contents of the movie the `Contents...` button is used. This creates a window (Figure 16) in which the complete index list can be scrolled. The current index is marked in the scrolling list by a rectangular frame.

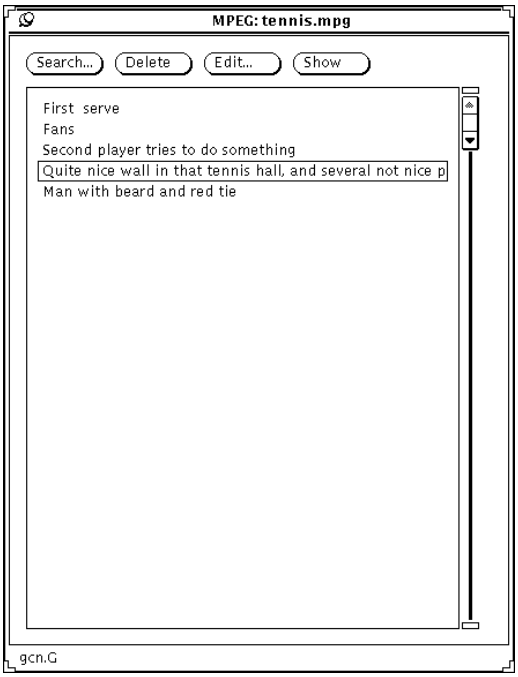


Figure 16: Contents Window

To play a movie from a specific index point, the user can scroll through the `Contents` window and click on the desired index, next the `Show` button is used for displaying the first frame with this index as a still image. To play from this point, the `Play` button on the base window or `Play` from the popup `Controls` menu is used (Figure 15).

Working with Movie Contents

The user can create, read or edit the textual contents of a movie. The contents are stored in the same file as the movie. The header of a movie, according to the MPEG-1 sequence layer, can contain user data labelled by a special start code (000001B5 in hexadecimal), called `user_data_start_code` [5]. According to the standard there is one restriction on the user data: it should not contain a string of 23 or more zero bits [5]. The standard also allows storage of user data on two other layers: the group of pictures layer and the picture layer, respectively.

An index has following structure:

- ❑ Index length (in bytes).
- ❑ Index text (string).
- ❑ Absolute address of the nearest access point in the past, from which the decoder can start parsing the file.
- ❑ Relative number of the frame marked with this index (relative access point).
- ❑ Absolute number of the frame from the top of the movie.

In the current version of the MPEG player all contents can be stored at the sequence layer. When the decoder encounters user data, while reading an input file, it sends the data to the player which in turn reads and stores the contents in a special list.

The player has following items in base window (see Figure 15) to work with contents:

- ❑ The `Index` test field, which displays the current index of a movie loaded
- ❑ The `Index...` button to assign an index to the current frame and add it to movie contents
- ❑ The `Contents...` button to open the `Contents` Window
- ❑ The `Search...` button to find a frame, which User is interested for, using key words.

All these controls buttons are duplicated as items in the pop-up `Controls` menu on Players screen (see Figure 15).

To mark interesting frame User should stop the Player clicking `Pause` (or `Step`) button and then `Index...` or just `Index`. The `Index` window appears (Figure 13). This window offers User to fill `Index` field with a new index.

To show contents the player create the `Contents` Window (Figure 16), which appears if User chooses `Contents...` button.

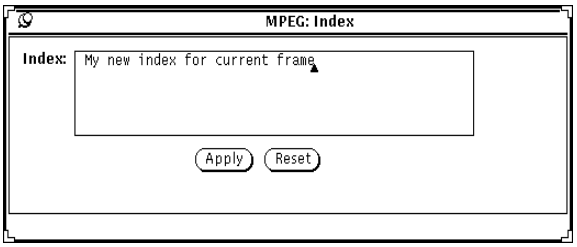


Figure 17: Index Window

The `Contents` window contains the control buttons to work with the contents as well as the Contents scrolling list. The controls of this window include:

❑ The `Search...` button, which has the same meaning that the `Search...` button at the top of the MPEG-Player base window (see Figure 15). To find interesting frame User should fill Index field in `Search` window (Figure 18), which appears when User clicks `Search...`

Character "|" is delimiter between key words being searched. A key word is substring, which need to find. The exclusive `Operation` Menu (AND/OR) enables to set search conditions.

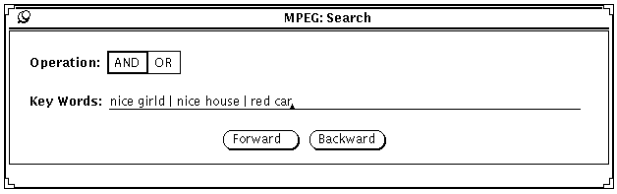


Figure 18: Search Window

❑ The `Delete` button. User chooses this button to delete an index chosen from the contents.

❑ The `Edit...` button, which enables User to edit the index chosen. If User clicks this button the `Index` window appears (Figure 17) with an old index. User edits it and clicks `Apply` to store a new index.

❑ The `Show` button can be used to look at the point, which User is interested for. This button supports "go to" action in the movie loaded.

References

1. Thorelli L-E, *"The EDA Multiprocessing Model"*. Technical Report TRITA-IT-R 94:28, CSLab, Dept. of Teleinformatic, KTH. 1994.

2. Geist A, *et al*, *"PVM3 User's Guide and Reference Manual"*. ORNL/TM-12187, Oak Ridge National Lab. May 1994.

3. Hockney R. W., Jesshope C. R., *"Parallel Computers"*. Adam Hilger Ltd. 1986.

4. LeGall, Didier, *"MPEG - A Video Compression Standard For Multimedia Application,"* Communication of the ACM, April 1991, Vol. 34, Num 4, pp 46-58.

5. ISO/IEC JTC 1/SC29, *"Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbits/s"*, Draft International Standard ISO/IEC DIS 11172, October, 30, 1992.

6. Patel, Ketan and Brian C. Smith, Lawrence A. Rowe, *"Performance of a Software MPEG Video Decoder,"* Proceedings ACM Multimedia'93, Anaheim CA, August, 1993.

7. *"Open Windows Developer's Guide 3.0.1. User's Guide"*, Sun Microsystems, Inc., 1993.