A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstations

Ajit Singh, Jonathan Schaeffer, *Member, IEEE,* and Mark Green, *Member, IEEE*

***Abstract*—Despite rapid growth in workstation and networking technologies, the workstation environment continues to pose challenging problems to shared processing. In this paper, we present a computational model and system for the generation of distributed applications in such an environment. The well-known RPC model is modified by a novel concept known as template attachment. A computation consists of a network of sequential procedures which have been encapsulated in templates. A small selection of templates is available from which a distributed application with the desired communication behavior can be rapidly built. The system generates all the required low-level code for correct synchronization, communication, and scheduling. This results in a system that is easy to use and flexible, and can provide a programmer with the desired amount of control in using idle processing power over a network of workstations. The practical feasibility of the model has been demonstrated by implementing it for Unix[1]-based workstation environments.**

***Index Terms*—Coarse grain concurrency, distributed computing, distributed software engineering, network systems, parallel programming, parallel programming models, workstation environment.**

I. Introduction

WORKSTATION environments have been in use for more than a decade now. In an ever increasing number of research, industrial, and academic environments, the bulk of the computing is now done using personal workstations. Large time-sharing systems, however, continue to be used for jobs that require large amounts of processing time and therefore cannot achieve acceptable performance on a single workstation. Although a network of workstations together represents a large amount of computing power ("the network is the supercomputer"[2]), a single user often cannot utilize this power for his applications.

Harnessing the computing power of a network of machines poses some interesting problems. First, the processors available to a distributed application and their capabilities may vary from one execution to another. Second, communication costs may be high in such an environment, restricting the types of parallelisms that can be effectively implemented. Third, users do not want to become experts in networking or low-level communication packages to utilize the potential parallelism. Unfortunately, there are few systems that are aimed at providing shared processing power in a workstation environment, while taking into account the constraints of the environment and the user [17], [26].

In this paper, we describe the *FrameWorks* system for writing applications to run in a distributed environment. This system has several important features.

1) Programs are written as sequential procedures enclosed in *templates.* The templates hide all the distributed computing implementation details, such as communication and synchronization. The code can be compiled to run in a sequential environment or in a distributed environment *with only minor changes (if any) on the user's part.*
2) The procedures themselves contain only a small amount of information as to how they interact with the rest of the system. Most of it is specified separately via *templates.* Such a decoupling of specifications results in an environment where applications can be readily adapted to the number of processors available.
3) A graphical interface facilitates the interactive construction of a distributed application and the specification of processor allocation constraints. The same interface is also used for querying or modifying these specifications.
4) *Contractor* templates provide a novel method for distributing work in environments with changing resources. Contractors dynamically contract out work to employee processes. They adapt to the changing constraints of the environment.
5) A user can exercise a wide range of control over the mapping of processes to processors. Using a high-level notation, the user can specify the processor assignments completely, partially, or leave it entirely up to the environment.
6) *FrameWorks* provides global system monitoring to achieve load balancing, detecting when workstations fall idle or become heavily loaded, and monitors the system performance for the user.

FrameWorks programs are written with the aid of a preprocessor for the C programming language. The user writes sequential C code and encapsulates it in templates, and the system translates this into an executable module that runs in the *FrameWorks* environment. Processes run in the background, taking advantage of idle machines when available and recognizing when machines become heavily loaded. In this way, we can keep the user community happy, while having applications profitably using machines that would otherwise be idle.

The rest of the paper is organized as follows. The concept of modules (Section II) and templates (Section III) describes what the user needs to know to write a *FrameWorks* application.

Manuscript received April 21, 1989; revised January 12, 1990 and May 4, 1990. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada. This work is a revised and extended version of the paper in the International Conference on Parallel Processing '89 [33].

The authors are with the Distributed Systems Research Laboratory, Department of Computing Science, University of Alberta, Edmonton, Alta., Canada T6G 2H1.

IEEE Log Number 9037581.

[1] Unix is a registered trademark of AT&T.

[2] A generalization by Carl Hamacher of Sun Microsystems's phrase "the network is the computer."

Section IV contains examples illustrating how to program using modules and templates. A specialized extension of the contractor template is described in Section V. The complete process of developing distributed applications as well as the software architecture of *FrameWorks* are discussed in Section VI. Some performance results are presented in Section VII. Section VIII compares our system to work done by others. Finally, Section IX presents some concluding remarks.

II. Modules

We assume an environment where workstations are connected over a local area network and run under a distributed or network operating system that provides a basic message passing mechanism [8], [21], [27], [37]. The overall organization of a distributed program in *FrameWorks* has striking similarities with the organization of a sequential program. Therefore, an application can be quickly altered to run in a sequential or distributed environment. At the macroscopic level, *FrameWorks* looks similar to several other systems that use system calls for process communication. However, the semantics, implementation, and user interface of *FrameWorks* differ significantly from other work (as described in Sections VI and VIII), yielding a system that is easy to use and requires little knowledge of distributed computing.

Briefly, an application consists of communicating processes. The programmer views each process as a sequential module or procedure. Parallelism is achieved using the simple *call* statement, which the programmer treats as a procedure call. All communication and synchronization is removed from the user's responsibility. *Templates* are used to encapsulate sequential code and indicate how the process is to interface to other parallel processes. From these templates, all commonly occurring forms of distributed process graphs, such as pipelines or master/slave relationships, can be constructed. In essence, the system provides a few simple building blocks, from which systems can be built quickly and reliably.

A complete application consists of modules that communicate with each other via remote procedure calls [5]. No common variables among modules are allowed. Each module contains a single *entry procedure* that can be called from other modules. In addition, a module can have one or more *local procedures* that can only be called from within the module. Each application has one *main* module that contains the *main procedure* in addition to its optional *entry procedure* and *local procedures*. The *main* module initiates the execution of the application. In many ways, this is analogous to programming with abstract data types, which provide well-defined means for manipulating data structures while hiding all the underlying implementation details from the user.

FrameWorks augments a programming language (C in our case) with the *call* statement, which has the same semantics as a procedure call, but is translated by the system into communication with another process. An arbitrary module X can communicate with a module Y by the simple

$$\text{call } Y(\text{input});$$

with input being the set of input parameters. X and Y are the names of modules specified by the programmer at compile-time. With no return value expected, the semantics of such a call are that Y is invoked asynchronously, without X waiting for its completion (*nonblocking* communication). Synchronous communication is possible using the blocking version of *call*

$$\text{output} = \text{call } Y(\text{input});$$

where output is the return value(s) from Y. In this case, the calling routine X is suspended until the reply (output) from Y is received. The called routine replies by executing an explicit

$$\text{reply(output);}$$

statement.

The input and output of modules are in the form of structured messages called *frames*. A frame is similar to a Pascal record or C structure except that pointer type variables are not allowed. For every module, the programmer declares an input frame (a structure containing all the input parameters needed for a *call*) and, if necessary, an output frame (containing all the *reply* or output values returned).

III. Templates

Most of the information regarding a module's interaction with other modules is added through a separate set of attribute bindings known as *template attachments*. A *template* represents a prepackaged set of characteristics which can be used for partially specifying the scheduling/synchronization structure for a sequential module. Depending upon his needs, a user selects a set of appropriate templates that completely describes the behavior of a module in the application. Every module needs up to three templates to fully describe its behavior in a concurrent environment (input, output, and body templates). As described below, an input template is responsible for correct scheduling and synchronization of incoming messages. Similarly, an output template deals with the scheduling and synchronization of calls originating from the given module to other modules. The use of a body template is optional. It is used to assign additional characteristics to a module that modify the module's execution behavior in the distributed environment.

Templates are used to describe the interaction of a module with other modules. Essentially, most distributed programs take advantage of several frequently occurring communication structures, such as pipelines, or master/slave relationships [7], [24]. After the user specifies which template(s) apply to their module, *FrameWorks* inserts the necessary code to set up the desired relationship. For example, if a user wants to set up a pipeline of processes, then they need only specify that each process gets its input from a pipeline, and that its output goes to a pipeline.

As shown in Fig. 1, the source code in a user's module is extended by several layers of code generated by the *FrameWorks* system to allow proper communication, synchronization, and scheduling. In addition to the code for the templates, there are two other layers of code that are required for the low-level communication package used (similar to [5]). The data layer is concerned with producing code necessary for sending and receiving frames in the form acceptable to low-level communication routines. The communication layer is concerned with the selection and routing of messages to the appropriate instance of a process when there are multiple executing copies of the called module. To distinguish the original code in a module from its augmented version, the latter is referred to as a process.

Currently three types of templates are supported. Fig. 2(a) gives the icons for these templates as used by the graphical interface. The templates are:

1) *initial:* Initial templates allow no input from other processes. Since such a process cannot be called by any other

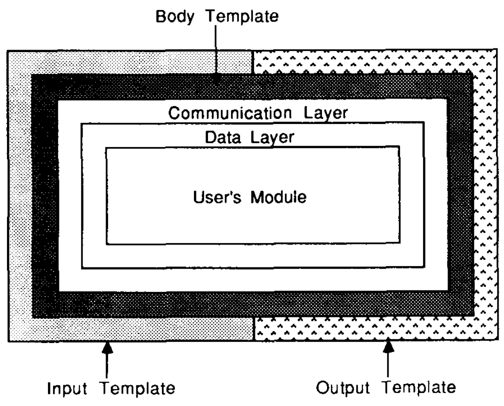


Fig. 1. Transformation of user module in process code.

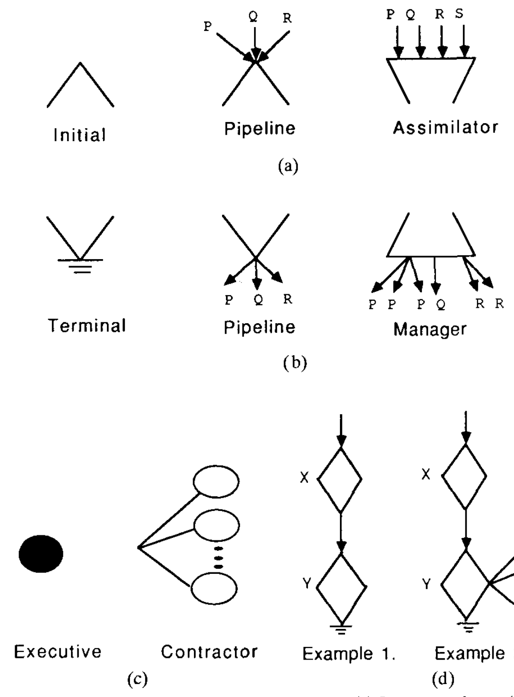


Fig. 2. Representation and use of templates. (a) Input templates. (b) Output templates. (c) Body templates. (d) Examples.

process, only the main module of the application may use this template. For example, an application where the *main* module does not have an *entry procedure* would require an initial template.

2) *in_pipeline:* A process with an in_pipeline input template can act as a server to any of its input processes, with calls from processes accepted in an FCFS manner. For example, this template would be used to specify the input interface for a process that is part of a pipeline.
3) *assimilator:* The assimilator template states that a process must have one input frame from each of its input processes before it starts processing any of its input frames. This template becomes handy if, for example, a process is to merge the outputs of several processes. To keep the semantics simple, an assimilator can only be called in a nonblocking mode.

There are three types of output templates [Fig. 2(b)]:

1) *out_pipeline:* This template allows for the output of a process to flow in a pipeline fashion to any process connected to its output.
2) *manager template:* Often some processes require more processing time than others and, to keep them in step with other parts of the application, a user may want multiple instances of the same process running. The manager template is concerned with the management and scheduling of multiple instances of the same module. The calling module is unaware of the existence of multiple instances of the called module. The manager will select the currently available instance for placing its call. For example, managers can be used to achieve master–slave relationships.
3) *terminal template:* This is simply a special case of the out_pipeline template. A process with terminal type output template does not call any other process.

Together, input and output templates provide several ways of structuring the interface of a module in a distributed environment. To illustrate the expansion of a user module, consider the module X in Example 1 of Fig. 2(d). Here, the input and output of module X have been structured as in_pipeline and out_pipeline, respectively. Fig. 3(a) shows a *FrameWorks* module that is part of a pipeline, and the pseudocode it is translated into is shown in Fig. 3(b). It should be noted how little programming was required in Fig. 3(a) to achieve Fig. 3(b). Of course, the savings are much more significant for more complicated structures such as managers or contractors (described below).

Body templates [Fig. 2(c)] provide further choice for specifying the behavior of a module. There are currently two choices for the body template. An *executive template* is meant to serve as a user interface to the application and hence only the main module of an application is allowed to use it. The executive template causes the process to have its input, output, and error streams directed to the user's terminal. Otherwise, these streams are directed to files whose names are based on the names of the module and the application [35]. If there is no executive type process in an application, then the complete application can be run in the background without any user intervention.

The default in *FrameWorks* is for a process to serve only one call at a time. A frequently occurring problem arises when some processes in an application require significantly more computation than others, perhaps creating a bottleneck. To solve this, one might try partitioning the work done by such a process by breaking it into smaller modules, or try using a manager template that distributes the work done by the process. However, each of these structures is defined at design time and uses a fixed number of processors. To increase flexibility at execution time in using idle processors and relieving the designer from specifying the exact number of processors required by such processes, the model provides a body template type called *contractor*. When a module's body is declared as a contractor, it means that the process gets its work done by *employee* processes. A module with a contractor body template is repeatedly and asynchronously called by its input nodes. Depending upon the number of available processors and the processor time requirements of the module, a contractor hires a dynamically varying number of employees to get the job done. The designer of the application does not take any part in the hiring and firing of employees; one simply specifies that the given module should function as a

```
---------------------------------------------------------------------------
/* The code in the user module does not have any information regarding  */
/* the type of templates that are used to encapsulate it.               */

interface_routine( input_frame )
{
     ...
     routine code
     ...
     output = call Y( input );
     ...
}

local_procedures ... /* local procedures may issue remote calls as well */
---------------------------------------------------------------------------
```

(a)

```
-----------------------------------------------------------------------
/* The expansion of user module as shown below is based on the        */
/* assumption that the module is assigned in_pipeline, out_pipeline,   */
/* input and output templates respectively. No body template is used.  */

in_pipeline()
{
     while( true ) {
          who = CheckMessage();
          if( who == A_VALID_CLIENT ) {
               ReceiveMessage( who, input );
               interface_routine( input );
          }
          else if( who == FW_MONITOR ) {
               ReceiveMessage( who, message );
               if( message == REQUEST_FOR_STATUS )
                    SendMessage( MONITOR, status );
               else if( message == TERMINATION_NOTICE ) {
                    CloseBooks();
                    Terminate();
               }
               else if {
                    /* some other actions (not included) */
               }
               else ErrorHandler();
          }
          else ErrorHandler();
     }
}

interface_routine(input)
{
     ...
     ...
     routine code
     ...
     /* expansion of "output=call Y( input )" for out_pipeline           */
     /* output template.                                                 */
     CheckValidServer( Y );
     status = WaitForStatus( Y );
     if (status != IDLE)
          ErrorHandler();
     SendMessage( Y, input, REPLY_MODE );
     ReceiveMessage( Y, output );
     /* end of expansion */
     ...
     ...
}

local_procedures...
-----------------------------------------------------------------------
```

(b)

Fig. 3. (a) User module. (b) Expansion of user module into process code.

contractor. The actual process of hiring employee processes is managed by the run-time environment and is transparent to the designer.

Both contractors and managers provide a means of running multiple instances of a process. In each case, replicated instances of a process compute independently without interacting with each other. Also, for the correctness of a computation, it is necessary that such processes do not contain any state information within themselves. Contractors hire and fire employees dynamically at execution time depending upon the workload. Managers provide a restricted form of the services provided by a contractor. However, since a manager executes only a fixed number of processes, its execution time overheads are lower than those of a contractor. Example 2 of Fig. 2(d) shows the revised representation of the process Y in Example 1, after it is declared to function as a contractor. *This is done without any change in the source code of the modules.*

The user is provided with a graphical user interface that is used to specify the template attachments to the modules, interconnection between modules, processor allocation preferences for the modules, etc. The interface between modules is checked for consistency. For example, a process cannot *call* another process whose input is an *initial* input template. The system also checks that frames match between communicating modules.

IV. An Example of a Distributed Animation System

To illustrate our model, this section describes an application for distributed animation that has been implemented using *FrameWorks* [16]. Computer animation involves creating a sequence of graphical images that, when shown in rapid succession, create the illusion of motion. The calculations required to produce these images are compute intensive and require a high volume of data. In the case of video, 30 images per second are required, each image having a resolution of at least 512×512 pixels and each pixel representing 8 to 24 bits of information. Determining the value of a pixel may require hundreds of floating point operations. Performance results related to the generation of a specific image sequence using several different configurations of this system are described in Section VII.

The complete system consists of about 2500 lines of code in our extended C language and is divided into three modules: *Model, PolyConv,* and *Split.* The overall structure of these modules, along with the frames used by them for communication, are shown in the Appendix. Fig. 4(a) shows one possible interconnection of these modules using templates, prepared using the *FrameWorks* system's graphic interface called *Module_Craft.* (More on *Module_Craft* will be presented in Section VI where we describe the implementation of the complete *FrameWorks* system.) The code in *Model* depends on the subject(s) of animation. It computes the location and motion of each object in the image and stores the results in a file on the distributed file system. Completion of this computation is communicated to *PolyConv* by issuing a nonblocking call. After this, *Model* continues with the computation of the next image in the animation sequence. At the same time, *PolyConv* reads the data from the disk and performs some data format transformations followed by viewing transformation, projection, sorting, and back-face removal. *PolyConv* passes the data in the image along with the image's sequence number to the *Split* module through a nonblocking call. *PolyConv* then waits to serve the next incoming call. The *Split* process performs hidden surface removal (using the *z*-buffer algorithm) and anti-aliasing. Finally, the rendered image is stored on a disk file.

Most of the computation done by the application comes from the *Split* module. In fact, it is the computational bottleneck for the complete system. The effect of this bottleneck can be reduced significantly by assigning a contractor body template to the *Split* module [Fig. 4(b)]. By doing so, several input images from *PolyConv* are processed concurrently by *Split.* To do this, *Split* dynamically hires a varying number of processors, each of which is assigned a single image for processing. Now, suppose we want to find out whether speeding up computations in the *PolyConv* module will speed up the application. Once again, this can be done by assigning a contractor body template to the *PolyConv* module [Fig. 4(c)]. With this configuration, *PolyConv* processes several input frames from *Model* concurrently, whereas *Split* processes several input frames from *PolyConv* concurrently. However, as shown by the performance results presented in Section VII, attaching a contractor template to *PolyConv* does not speed up the overall computation by a significant amount. Therefore, we revert back to the configuration in Fig. 4(b), by simply detaching the contractor template from the *PolyConv* module. The speed of the overall computation now depends on how fast *Model* and *PolyConv* can supply input data to *Split.* Finally, suppose we want to ascertain how the performance of the complete application scales as the number of processors employed by the *Split* module is varied. This can be done by modifying the configuration of Fig. 4(a) to the one shown in Fig. 4(d). Here, the output of *PolyConv* has a manager template and during each execution, a fixed number of *Split* processes are specified to be run under it. Sample performance results, for all the four configurations described here, are discussed in Section VII.

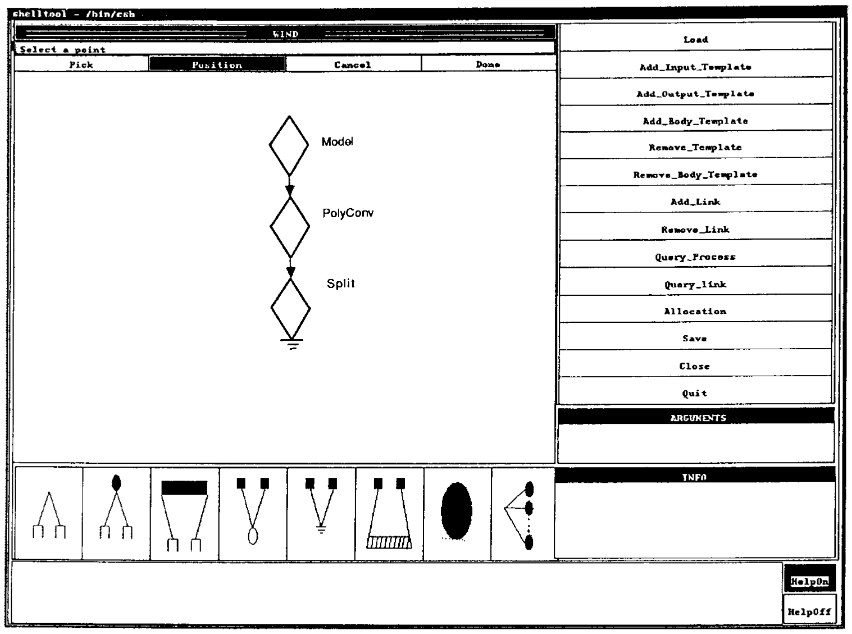
Our simple example here serves to illustrate several important features of *FrameWorks.* First, the complete application can be converted to run in a sequential environment by changing only two call statements in the whole source code.[3] This means that development and debugging can be done in the more familiar sequential environment and, once a correct solution is achieved, it can be quickly modified to run in a distributed environment. The initial version of the animation system was developed as a sequential application without any knowledge of the *FrameWorks* system. However, it was possible to convert it into a *FrameWorks'* form with a negligible amount of extra effort. Second, the performance of the time consuming *Split* module scales automatically. The user simply declares *Split* as a contractor and as many processors as are available are used to speed up the computation. Third, the load balancing in *Split* is done dynamically. If there are idle employee processes, they are released. If work is coming in at a faster rate, then more employees can be hired. Fourth, unnecessary context switching is absent. A new employee process is created only if a new (idle) processor is available. Finally, it should be noted that the configurations of the animation system in Fig. 4(a), (b), (c), and (d) have all been created by simply altering a graphic diagram, without making any modification to the code in the modules.

V. Dynamic Task Scheduling Within Contractor Templates

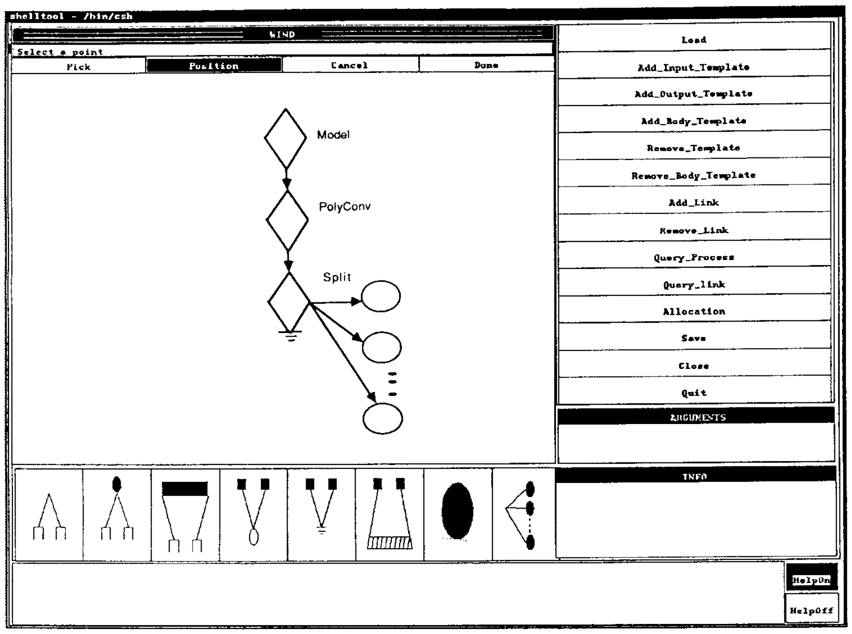
There is a large class of problems where the program structures are simple yet they require an enormous amount of computing. Often, in such programs, the strategy is to break the original problem down into several small independent subproblems. Each of these subproblems can be assigned to a different process and the results received from these processes can be combined to form a solution to the original problem. Sometimes, the number of subproblems and their sizes can be determined at program design time. The contractor or manager templates as described in Section III can be easily used for such situations. Often, however, the problem cannot be divided into independent subproblems of equal size at program design time [3], [14], [23], [29]. Rather, the creation and synchronization of subproblems proceeds recursively and dynamically, and these subproblems can be of widely uneven sizes. Although the contractor or manager templates can be used in such cases as well, inefficiencies result because there could be some processors that are idle, whereas the rest of the solution process cannot proceed until the results of some child processes become available (Fig. 5). Instances of such problems are quite commonly found in problems that use divide and conquer, depth first search, breadth first search, or branch and bound strategies.

To efficiently handle the scheduling of dynamically evolving medium-grained tasks, *FrameWorks* provides two more language

[3] Such a conversion process can be automated for a large class of applications where the network graph is acyclic, and each module of the application is history insensitive, i.e., the computation done by each module depends only upon the input to that module and not on the previous inputs processed by the module.



(a)



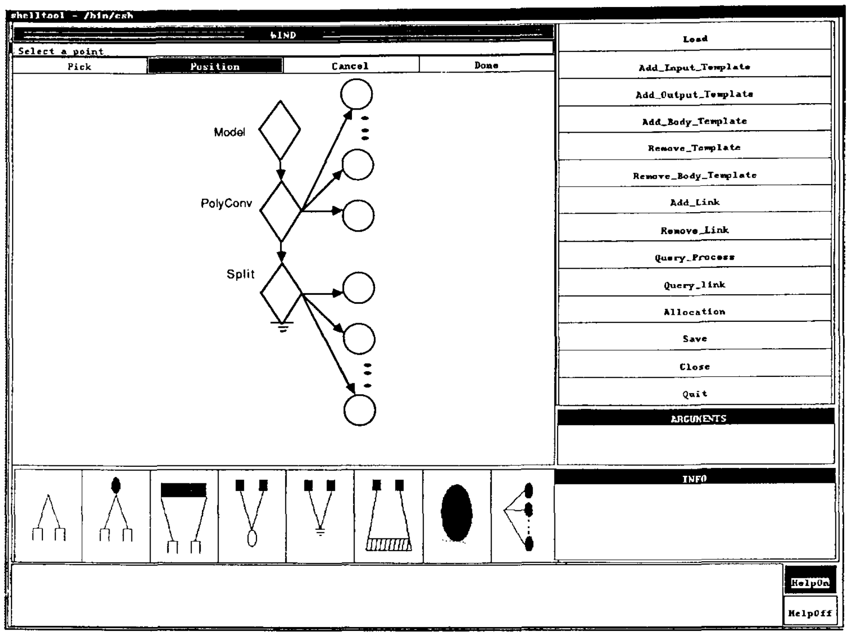
(b)

Fig. 4. (a) A pipeline structure of the animation system. (b) Split module assigned contractor template.

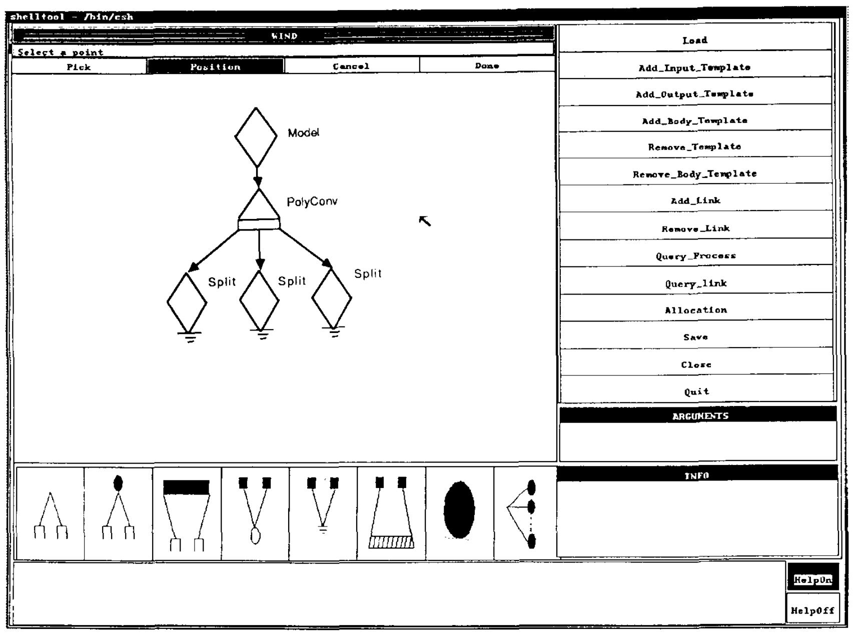
constructs that can be used within a module enclosed in a contractor body template. The call

$$\text{job\_id} = \text{split\_job}(\text{job\_frame}, \text{mode});$$

causes creation of a new employee process. The structure of the *job_frame* is the same as the structure of the module's input frame. The *mode* of call can be either blocking or nonblocking. In the case of a blocking *split_job* call, execution in the calling process is blocked until a new processor is available. The call returns a positive integer in *job_id* that serves as the identification number for the newly started subproblem. In the nonblocking mode, the call returns immediately. If an idle processor is found, the call returns a valid job identification number. Otherwise, a



(c)



(d)

Fig. 4. (*Continued*) (c) PolyConv and Split assigned contractor templates. (d) PolyConv used as a manager of Split processes.

negative number is returned and no new process is started. This allows an employee process to take advantage of any other currently idle employee in a dynamically expanding and shrinking pool of employees. To receive the results from processes started via *split_job* calls, another *FrameWorks* call *merge_job* is used. The call

result = merge_job(rep_frame, job_id, mode);

returns immediately if the nonblocking mode is used. In this case, the call returns 0 if the subproblem given by *job_id* has been completed, and -1 otherwise. If the subproblem has been completed, the reply frame is returned in *rep_frame*. In the case of blocking mode, the calling process is blocked until the specified subproblem is completed, after which the call returns with the reply frame being placed in *rep_frame*.

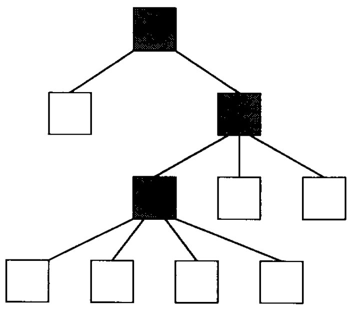


Fig. 5. Recursive splitting of a task.

The split_job and merge_job calls provide a powerful and flexible method for creation and control of hierarchical threads of execution and efficient dynamic scheduling of available processors. Using these calls, the employee processes of a contractor module can further subdivide their work without worrying about complicated processor management in the workstation environment. The *split_job* call provides a means by which a new process is not started until an idle processor is found. In such a situation, the calling process has the option of trying the *split_job* call later, or doing the job by itself. Although, the *split_job* and *merge_job* calls appear similar to other known methods of starting a new thread of execution, such as fork/join statements and methods based on spawning a new process [10], [11], there are some significant differences. In these other methods, a new process is started whether or not an idle processor is available. In our case, each processor runs a single process and a new process is added only when an idle processor is available. This has several positive implications. It eliminates logically pointless and expensive context switching, and avoids clogging of systems' job queues with jobs waiting for processor time. Also, these calls provide greater flexibility in using idle processors at execution time. The number of processes created can be adjusted according to the computational requirements of the application as well as the available processors. Allowing the idle processors to help out the more busy ones significantly reduces synchronization costs in the types of computations mentioned earlier. However, the most significant difference between these methods and the scheme based on *split_job* and *merge_job* is the ease of programming applications that require dynamic creation and control of hierarchical threads of execution. As an example, the parallel chess playing program ParaPhoenix [29] performs dynamic splitting of tasks for implementing distributed alpha–beta search. It conducts a variable depth distributed game tree search using workstations on a local area network via a carefully handcrafted implementation. The implementation took several man months of effort to obtain a relatively error-free program. The same functionality can be derived in a rather straightforward manner through the use of *split_job* and *merge_job* commands.

VI. FrameWorks System

The *FrameWorks* system has been implemented at the Department of Computing Science, University of Alberta. Our general computing environment consists of over 50 Sun workstations connected via Ethernet cables. In addition, there are 4 VAX 11/780's, two MIPS m/1000 machines, and several other kinds of special purpose workstations. Most of the workstations are situated in various laboratories and the rest are located in individual offices. Although one of the goals of *FrameWorks* is to allow the user to use processors with different architectures, in the current implementation only a homogeneous set of Unix machines can be used within a single application. *FrameWorks* has been implemented mostly using tools developed in our department. For the distributed computing aspects, the Network Multiprocessor Package (NMP) is used [22]. NMP is built using sockets [21] and it provides a library of high-level routines to support remote process initiation and message passing among processes. The visual interface is implemented for color as well as monochrome monitors using locally developed user interface management tools called, Diction, Chisel, and Vu [31], [32]. The following subsections briefly describe the implementation and functionality of *FrameWorks*, as well as the process of developing distributed applications using the system.

A. Developing Applications Using FrameWorks

The complete process of generating a distributed application is split over two phases as shown in Fig. 6. The first phase consists of input from the user. Two types of input are required. First, a user prepares source code modules as shown in the Appendix. These modules, called *FW* modules, are written in an extended version of the C language that allows *call, reply, split_job,* and *merge_job* statements. The second type of input consists of the assignment of templates to each module, specification of their interconnection, and the names of the input and reply frame(s) for each module. The interactive graphical editor *Module_Craft* is used for this purpose. Sample *Module_Craft* screen layouts are shown in Fig. 4. Using the command and item menus of *Module_Craft*, a user prepares and edits application specifications as shown in the previous sections. *Module_Craft* is also responsible for checking compatibility and other semantic constraints in the preparation of the communication diagram.

Allocation constraints of a module can be specified using *Module_Craft* via its *allocation* command. The *include* and *exclude* options allow the specification of processors that may be selected or avoided in allocating the given process. These commands may be repeated. The special name *free_pool* denotes the available list of processors other than those specifically specified. The list of preferences is scanned strictly sequentially from top to bottom and, once a selection is made, the rest of the list is left unscanned. For example, let us assume the following sequence of preferences for a process *X*:

```
include sun001 sun005;
exclude sun013;
include free_pool;
```

The allocator first attempts to allocate *sun001* or *sun005* if either is available. If not, then any processor except *sun013* is allocated. Processors are allocated in a depth-first manner starting from the *main* module. For a module that has not been assigned the contractor body template, the allocation of processors takes place just prior to initiating execution of the application. Therefore, depending upon availability of processors, during different executions, different processors may be selected. For a contractor module, the processors are selected for employee processes at execution time. These processors are again selected on the basis of allocation constraints. If the allocator cannot satisfy a user's allocation requirements with the current status of processors, the allocation procedure is aborted and the user is given an appropriate error message. The user then has two options. He can modify his allocation constraints so that they can be satisfied with the currently available processors, or he can

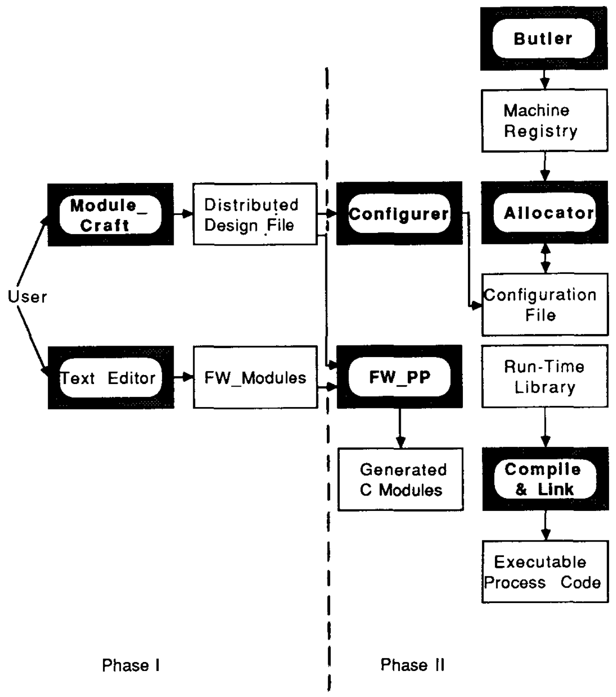


Fig. 6. Generation of distributed applications.

retry the allocation procedure later when the processors required by him are available. This processor allocation scheme is simple yet quite powerful. For example, it can be used to limit the maximum number of employee processes that can be created at execution by a certain contractor process.

The distributed design of the application, as specified through *Module_Craft*, is stored in a file called the *distributed_design_file (DDF)*. In the second phase, a system component called the *configurer* prepares a detailed distributed design, called the *configuration_file*, for the application in which each process is given a logical node identification number. The information in the *DDF*, along with the application's source modules, is processed by the *FrameWorks* preprocessor (*FW_PP*). *FW_PP* generates all the code for the layers discussed in Section III (Fig. 1) in order to transform a module into a process executable in the distributed environment. The output of *FW_PP* consists of C language modules. Executable modules are prepared by compiling these modules and linking them with the *FrameWorks* run-time library. Allocation of processors is performed by the system component called the *Allocator*. The system component *Butler* maintains a database (called the machine_registry) of the status of available processors and their load averages (similar to [26]). *Allocator* uses the information in the machine_registry to allocate processors to processes. Due to the lack of a process migration facility in the Unix operating system, processors are bound to processes for their entire lifetime. However, as described in the next section, *FrameWorks* does provide a limited facility for adding or removing processors at execution time in the case of a contractor.

An application runs in the presence of the *Butler* and an *Execution_Time_Monitor (ETM)* which provide execution time services required by an application (Fig. 7). The execution time environment of *FrameWorks* is described briefly in the next subsection.

B. Execution Time Environment of FrameWorks

The *Butler* and *ETM* together manage the dynamic allocation and release of processors used by contractor processes. The contractor template is implemented as a closely related group of processes consisting of a coordinator and its employees. The implementation is quite different from the implementation of threads in Mach [28] in the sense that coordinator or employees do not share address spaces. It is more similar to the "process-groups" of V [8] with the significant difference that the management of employee processes is transparent to the programmer, and is handled completely by the coordinator process. For every module which has been assigned a contractor body template, a separate coordinator process is started. This coordinator process is responsible for managing the employee processes. From the users' point of view, an employee process represents an instance of the user module functioning as a server. The actual number of employee processes at any time during execution depends upon the allocation constraints specified by the user, the number of processors available on the network, and processing requests that arrive from client nodes. In addition to managing the number of employee processes, the coordinator is responsible for receiving input from the input nodes, sending this input to a selected employee, receiving the input back from the employee, and sending the output to output nodes. An employee repeatedly serves calls assigned to it by its coordinator process until it is ordered to terminate by the coordinator. Employee processes also use the coordinator for providing services required by the *split_job* and *merge_job* commands for dynamic task scheduling among themselves. Thus, in addition to providing

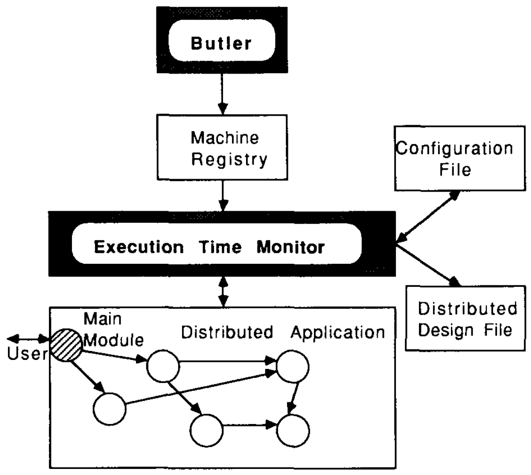


Fig. 7. Execution time environment.

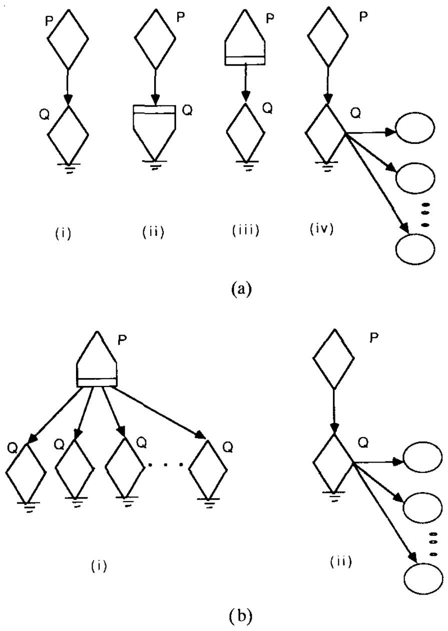


Fig. 8. (a) Experimental configurations for Fig. 9(a). (b) Experimental configurations for Fig. 9(b).

the complete functionality of a contractor template as described in Sections III and V, our implementation scheme avoids the inefficiency of starting a new process for each call and then terminating it after the call is served.

Distributed applications often need a mechanism for detecting the termination condition of their processes. For example, researchers have studied the problem of termination for the CSP model [9], [13]. Termination of a *FrameWorks'* application is the other important service provided by the *ETM*. The modules in a *FrameWorks* application act as servers to modules connected to their inputs. Therefore, each module (except the one with *initial* input template), after finishing the processing of the current input data, waits for more input to arrive. The *Execution_Time_Monitor* checks for situations where some or all of the processes fall idle, and also will not receive any further input. Such processes are marked as *ready for termination* and are terminated by the *ETM*. Several different termination strategies for *FrameWorks* applications have been investigated. These strategies and their relative merits are discussed in detail in [34].

In addition to the services mentioned above, *ETM* optionally can provide other services such as performance data collection for individual processes and deadlock detection at run time. The details of these services are given in [34] and are not described here.

VII. Experimental Results

The *FrameWorks* system is being used in our department for a variety of applications. Among them are an implementation of a parallel chess program [29], a distributed animation project [16], a distributed system for choosing subsets of pattern recognition properties from a given set [25], a distributed version of the Make program [12], and several smaller applications. In the following subsection, we present some application independent measurements taken with the *FrameWorks* system. These experiments were performed on diskless Sun 3/50 workstations with 8 megabytes of memory. These workstations share a Sun 3/180 file server and a Sun 3/60 page server. The workstations run under the Sun OS 3.5 and communicate over 10 Mbps Ethernet interfaces. Some performance results for the animation system described in Section IV are also presented. These results are not meant to be comprehensive. Rather, they are presented to give an insight into performance characteristics and overheads of *FrameWorks*, as well as some indication of speedups that may be attained by using the system.

A. Application Independent Measurements

Two different kinds of measurements based on configurations shown in Fig. 8(a) and (b) were taken. The results of these measurements are shown in Fig. 9(a) and (b), respectively. The first set of experiments were done to ascertain the cost of executing *FrameWorks'* blocking and nonblocking *call* statements with varying message sizes. These results are based on a variety of configurations shown in Fig. 8(a). Although, for the sake of simplicity and brevity, not all possible combination of templates are used, all different types of templates are represented in these configurations. The time taken for 1000 calls was measured for each configuration. The results are shown in Fig. 9(a). Since assimilator and contractor templates are called only in a nonblocking mode, there are no curves showing results for blocking calls for configurations (ii) and (iv). To provide a basis for comparison against some well-known standard, results for executing send and send–receive using bare sockets with the TCP protocols are also shown.

There are several observations that can be made regarding the results shown in Fig. 9(a). First, the overhead of executing a *call* statement in the case of templates that are not concerned with the management of replicated instances of a user's module is small (approximately 10–15%). However, templates which manage replications (contractors and managers) are about 4 to 10 times more expensive. In each case, the overhead is due to two factors: the execution of extra code associated with templates, and the exchange of control messages for management purposes. It should be noted that although templates add overhead in terms of extra code and messages, a similar overhead will be there even if the user himself writes code to achieve similar functionality as

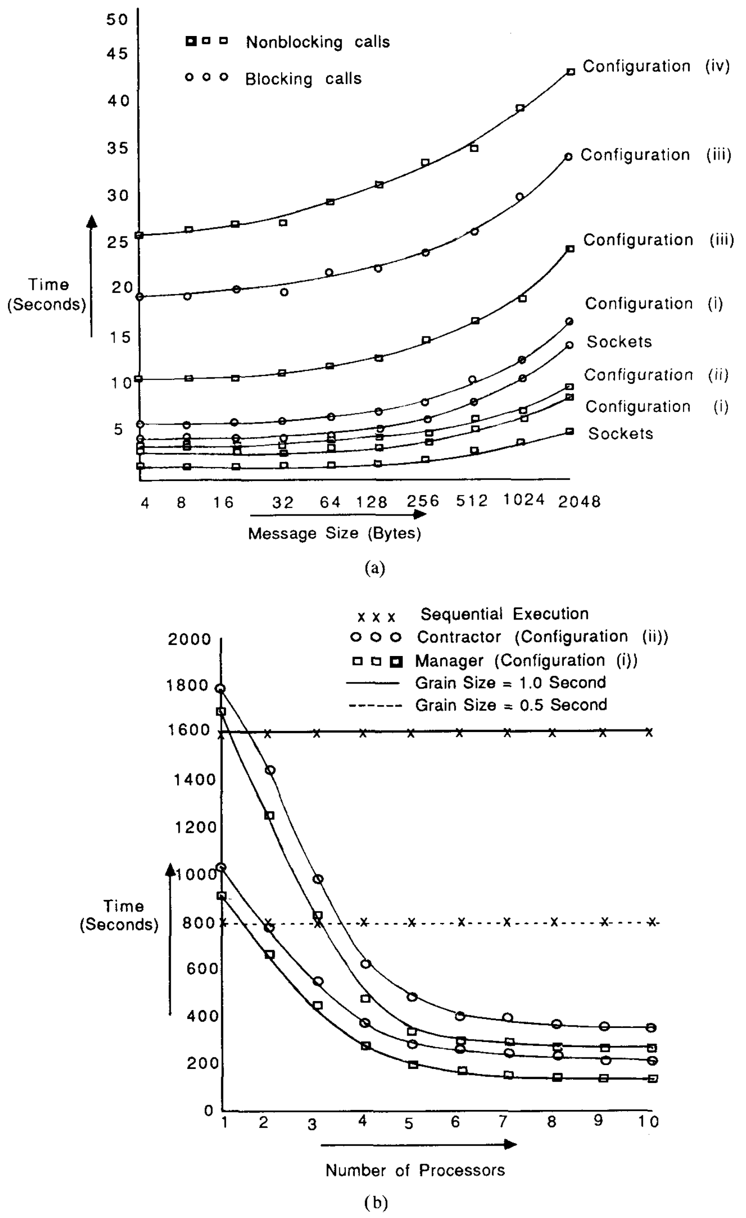


Fig. 9. (a) Call times for 1000 messages for configurations in Fig. 8(a). (b) Results for configurations in Fig. 8(b).

provided by templates. Except in the case of expert users, this overhead is not likely to be smaller than overheads associated with *FrameWorks'* templates. Therefore, although the overhead would exist whether the code is hand written or generated using templates, *FrameWorks* simply provides an easy way to get quick and correct code.

The second set of experiments was done to determine the behavior of templates that manage replication, namely the contractor and the manager. Configurations used for these experiments are shown in Fig. 8(b). In each case, the module P iteratively makes nonblocking calls to module Q. The module Q repeatedly computes floating point multiplications of two variables. The time taken in executing the module Q is controlled by the number of times this multiplication is done. This time represents the granularity of the work done by the replicated processes.

These results are shown in Fig. 9(b), where the number of workstations is varied from one to ten. Two different grain sizes were used, each having 1000 calls by module P to module Q. For the sake of comparison, the times taken when the same task is done sequentially are also shown. These results suggest that the overheads of a contractor are always higher than the overheads of a manager. Both contractor and manager perform well in cases where the amount of work done on each call is one second. When the work done on each call is reduced to 0.5 s, the contractor

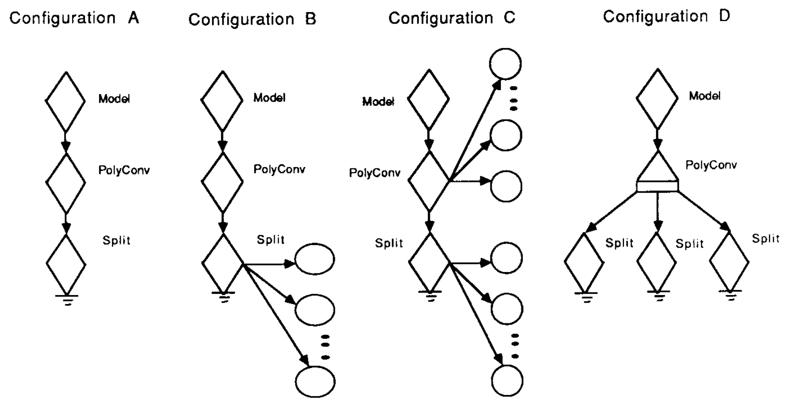


Fig. 10. Four configurations of the animation system.

does not do very well, but the manager still scales reasonably well. Although, the contractor's overheads are higher than those of manager's, in our opinion the advantages provided by the contractor template in an environment with changing processor resources justify its inclusion in our selection of templates.

The results in Fig. 9(a) and (b) show the overhead costs and range of acceptable grain sizes for good performance. Coarse grain applications, where computation times of modules are of the order of a few seconds and often even a few minutes (refer to the application in the following subsection), are not rare. In such applications, the grain sizes are well within the good performance range for the contractor as well as the manager. Even the worst case communication cost [0.04 s per call for configuration (iv) in Fig. 9(a)] would be quite acceptable in such cases.

B. The Animation System

In this subsection, some performance data taken using the animation system described in Section IV are presented. The *Model* module in this case deals with animating the simple schooling behavior of fish in a tank. The modeling and motion is computed through a model that takes into account behavioral characteristics of the fish and the laws of physics. In addition to the sequential version, all four distributed configurations discussed in Section IV were implemented. These configurations are shown together in Fig. 10. Fig. 11 shows the real time taken in computing 120 images. The variation of overall speedup with the total number of processors employed for configuration *D* is shown in Fig. 12, along with some of the data from Fig. 11. No additional speedup is gained when *Model* cannot supply input data faster than it can be processed by *PolyConv* or *Split*. The contractor processes in configurations *B* and *C* dynamically employ and release processors and therefore use an unspecified number of processors. In such cases, significantly better performance is observed during night times due to a larger number of lightly loaded or idle processors that are readily available for use.

Although this example is a simple one and is expected to get good performance, these experiments demonstrate that it is possible to achieve a fair amount of speedup using *FrameWorks*. The absolute value of the speedup is not important; given that our graphics group ran this program sequentially, *any* improvement in performance was welcome. That they achieved this with minimal programming effort was equally welcome. Distributed configurations, such as *B*, *C*, and *D*, have fairly complicated distributed structures operating behind them. Writing such an application using low-level primitives such as sockets or RPC's [5], [21] would require considerable effort on the programmer's part. Since such an effort would be replicated, perhaps unwittingly, by other programmers while writing different applications, *FrameWorks* provides an easy way of avoiding this wasted effort.

Configuration	No of Processors	Execution Time (In Minutes)	Speed-Up Ratio
Sequential Version	One	238	1
Configuration A	Three	138	1.7
Configuration B	Varying Dynamically	42	5.7
Configuration C	Varying Dynamically	40	6.0
Configuration D	Five	72	3.3

Fig. 11. Speedup achieved with various configurations.

VIII. Comparison to Related Works

A new approach to developing distributed applications in which parallel operations are encoded via templates has been presented. Each template can be looked upon as a collection of macros. Attachment of a template to a module causes these macros to be inserted and expanded at appropriate places in the user's code to handle low-level synchronization and communication. This approach is not entirely novel; a technique based on macro expansion is used in [20] to create parallel Fortran programs. Babb uses a technique based on macro expansion to restructure sequential Fortran programs to produce an acyclic network of processes that can be executed in parallel [2]. However, in both cases, the prescribed macros are at a much lower level and the programmer must insert them in the source code. In contrast, templates are attached externally to a module. Each template may represent encapsulation of several macros that are automatically inserted and expended at appropriate places in the source code. This makes the use of templates much simpler and allows significant flexibility in restructuring the application. Also, different templates can be used with the same procedure in different applications.

Comparison of *FrameWorks* to general purpose concurrent languages [4], [6], [18], or low-level distributed computing tools [5], [21], reveals some interesting benefits as well as some limitations of our template-based approach.

1) Unlike CSP or Ada, *FrameWorks* does not embody a

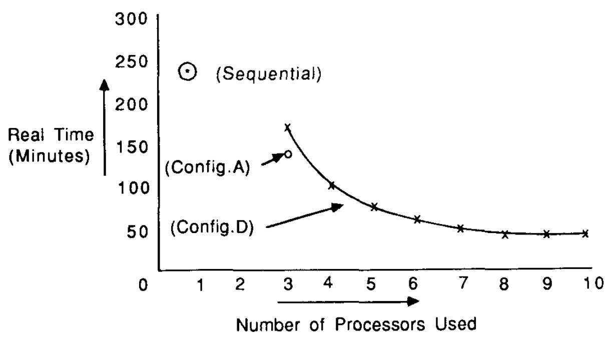


Fig. 12. Variation of real time with number of processors.

complete language for concurrent programming. Rather, it proposes a new parallel programming model which can be implemented on top of an existing sequential language like Pascal or C.

2) There is a large class of distributed applications where synchronization/communication/scheduling structures are much more restricted. It has been pointed out that such distributed applications can be generated more easily via specialized languages/systems than by general-purpose concurrent languages [7], [24]. *FrameWorks*, along with several other systems, can be put in the category of specialized systems/languages that aim at providing easy generation of commonly occurring forms of synchronization and concurrency [2], [7], [20], [24]. For large general-purpose programs, such as an operating system or a real-time transaction system where a group of processes interact or "converse" in an arbitrary manner [15], *FrameWorks* may not be a suitable tool. General purpose concurrent languages such as CSP or Ada, or low-level tools such as sockets, may be more appropriate for such situations.

3) Although *FrameWorks* provides a facile way of constructing distributed applications, it hides overhead and communication costs from the user. Consequently, the final design may not be as efficient as one that is custom built by an expert programmer. Customized generation of programs using low-level tools such as sockets is a time consuming and tedious process even for experienced programmers. However, it might be a suitable alternative for applications that critically depend on achieving maximum efficiency.

4) Most high-level concurrent languages do not allow a programmer to map his/her processes to processors. This is a handicap, especially in a workstation environment due to the nature of the division of ownership rights commonly found in such environments. *FrameWorks* provides a high-level notation in which a programmer can express his processor allocation constraints if desired.

5) Although the contractor/employee combination involves dynamic scheduling and termination of processes, for simplicity *FrameWorks* does not support dynamic initiation of processes by the user. Therefore, it is more suitable for environments where the computational structure of an application can be specified at compile time. For more complex situations, where the pattern of computation is unknown at compile time, languages such as Ada or other low-level tools [8], [21] might be more appropriate.

In Summary, *FrameWorks* is a special purpose tool for the generation of applications that can be structured in terms of commonly occurring communication/synchronization patterns. Such applications can be easily, quickly, and correctly generated using *FrameWorks* and they take advantage of available workstations in a flexible manner. However, the approach may not be suitable for programs with complicated communication/synchronization structures or programs that aim to attain very high levels of efficiency.

The idea of providing shared processing has been studied by several researchers [8], [17], [19], [26], [30]. Among the earlier efforts, the worm programs of Shoch and Hupp [30] developed some distributed applications in a workstation environment and demonstrated the feasibility of dynamically expanding and shrinking programs that utilized idle processors. However, their work did not concern itself with the development of any conceptual model or language for this purpose. Some other systems are based on managing a pool of available processors and supplying an application with an idle processor, or a processor with low load average, on demand [17], [19]. Yet another class of approaches is based on distributed operating systems [8], [27], [37]. Some systems allow the allocated processor to be reclaimed by a user who has a higher priority for using the processor [26]. Cosmic Environment and Reactive Kernel (CE/RK) systems [1] provide the usual low-level process spawning and message passing functions and also handle allocation of computers on a local area network to facilitate preparation of applications for running on large multicomputers. The *FrameWorks* system not only keeps track of the available processors but also offers a complete conceptual model for developing distributed applications, which is consistent with the constraints of the workstation environment. Using this model, an application can be easily modified to run with the available number of processors. Also, the contractor template allows processors to be gracefully added or removed from a running application. However, we have not been able to experiment with releasing an arbitrary processor from a running application for the lack of a process migration facility provided as part of the Unix operating system.

IX. Concluding Remarks and Future Plans

We have discussed the *FrameWorks* model and its implementation for a Unix-based workstation environment. The goals of the *FrameWorks* system are: 1) to relieve the programmer from writing low-level, error-free code for synchronization/communication with other modules, and 2) to develop a software tool that provides the programmer with different levels of control over the use of processors in a workstation environment. The underlying computational model makes it easy to develop, as well as to restructure, applications to match them to the available resources. The approach also, to a limited extent, facilitates the graceful addition and removal of processors from the application. However, the presence of a process migration facility is likely to make the use of our model much more pleasant [36].

So far our experience with *FrameWorks* indicates that in the case of preexisting sequential applications, a fair amount of performance can easily be obtained through the simple modifications needed to parallelize the applications. In several cases, partitioning of the complete application into modules was possible while keeping most of the code in the sequential version intact. However, in some cases more efficient partitioning of modules did require moderate amounts of work. In the case of applications that were designed with the *FrameWorks* system in mind, the amount of work required to switch between sequential and parallel versions was quite small. Also, restructuring the applications for experimenting with different templates often required either no modifications or only a small amount of modifications within the modules.

Our experience during the implementation of *FrameWorks* revealed an unexpected aspect of the template-based approach. We found that, unlike a compiler for a concurrent language, a template-based environment itself can be developed incrementally. New templates can be added and old templates can be refined incrementally as long as they remain compatible with the existing templates in the system. The initial versions of *FrameWorks* had fewer and less sophisticated templates. As the need for newer and better templates was felt, they were added incrementally.

At present, *FrameWorks* does not have features that provide fault tolerance in the case of node or communication failure. We feel that it is possible to integrate such features in the model and the system, and we plan to look at this aspect in the near future. Also, the present form of the *FrameWorks* model is especially suitable for the workstation environment. However, with minor modifications, variants of *FrameWorks* can be developed that would be suitable for implementation on tightly coupled MIMD architectures. One obvious advantage of implementing under such environments (or on faster networks) is that due to reduced communication costs, much finer grains of concurrency can be exploited as long as the application structures remain simple. Finally, the network diagrams produced by the graphic interface *Module_Craft* seem rather unconventional. However, they make the parallel structure of the application explicit. The assistance that such an interface can provide in visualizing, experimenting, documenting, and debugging the application makes it worth the effort. Currently, we are working on expanding the visual interface of *FrameWorks* to include a subsystem for providing visualization, debugging, and post-execution analysis. At this point, it is not entirely clear how our visual approach would be extended for more complex scenarios, e.g., applications with multiple entry points, or environments where both private as well as shared address spaces coexist. To explore the limits of our approach, we shall have to gradually bring these situations within our consideration.

Acknowledgment

We gratefully acknowledge the helpful comments of the referees in improving the earlier draft of this paper. Implementation and documentation support was provided by G. Klettke, G. Singh, W. Holst, and T. Breitkreutz.

References

[1] W. C. Athas and C. L. Seitz, "Multicomputers: Message-passing concurrent computers," *IEEE Comput. Mag.*, vol. 21, no. 8, pp. 9–24, Aug. 1988.
[2] R. G. Babb II, "Parallel processing with large grain data flow techniques," *IEEE Comput. Mag.*, vol. 17, no. 7, pp. 55–61, 1984.
[3] D. H. Ballard and C. M. Brown, "Scene labeling and constraint relaxation," in *Computer Vision.* Englewood Cliffs, NJ: Prentice-Hall, 1982, pp. 408–430, sect. 12.4.
[4] J. G. P. Barnes, "An overview of Ada," *Software—Practice and Experience,* vol. 10, pp. 851–887, 1980.
[5] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, pp. 38–59, Feb. 1984.
[6] P. Brinch Hansen, "Distributed processes: A concurrent programming concept," *Commun. ACM,* vol. 21, no. 11, pp. 934–941, Nov. 1978.
[7] G. Butler and M. J. Kendall, "The suitability of master/slave concurrency of Concurrent Euclid, Ada, Modula," *Software—Practice and Experience,* vol. 17, no. 2, pp. 117–134, Feb. 1987.
[8] D. R. Cheriton and W. Zwaenepoel, "The distributed V kernel and its performance for diskless workstations," in *Proc. Ninth ACM Symp. Oper. Syst. Principles,* Dec. 1983, pp. 129–140.
[9] S. Cohen and D. Lehmann, "Dynamic systems and their distributed termination," in *Proc. Symp. Principles Distributed Comput.*, 1982, pp. 29–33.
[10] M. E. Conway, "A multiprocessor system design," in *Proc. AFIPS Fall Joint Comput. Conf.*, Nov. 1963, pp. 139–146.
[11] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Commun. ACM,* vol. 9, no. 3, Mar. 1966.
[12] S. I. Feldman, "Make—A program for maintaining computer programs," *Software—Practice and Experience,* Apr. 1979.
[13] N. Francez, "Distributed termination," *ACM Trans. Prog. Lang. Syst.*, vol. 2, 1980.
[14] J. Gaschnig, "A constraint satisfaction method for inference making," in *Proc. 12th Allerton Conf. Circuit Syst. Theory,* Urbana, IL, Oct. 1974.
[15] E. F. Gehringer, D. P. Siewiorek, and Z. Segall, *Parallel Processing: The Cm* Experience.* Bedford, MA: Digital, 1987, pp. 225–270.
[16] M. Green and J. Schaeffer, "FrameWorks: A distributed computer animation system," in *Proc. Canadian Inform. Processing Soc. Edmonton '87,* 1987, pp. 305–310.
[17] R. Hagmann, "Process Server: Sharing processing power in a workstation environment," in *Proc. 6th Int. Conf. Distributed Comput. Syst.*, May 1986, pp. 260–267.
[18] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM,* vol. 21, no. 8, pp. 666–677, Aug. 1978.
[19] K. Hwang, W. J. Croft, and G. H. Goble *et al.*, "A UNIX-based local computer network with load balancing," *IEEE Comput. Mag.*, pp. 55–65, Apr. 1982.
[20] H. F. Jordan, "Structuring parallel algorithms in a MIMD, shared memory environment," *Parallel Computing,* vol. 3, pp. 93–110, May 1986.
[21] S. J. Leffler, W. N. Joy, and R. S. Fabry, "4.2BSD networking implementation notes," Univ. of California at Berkley, July 1983.
[22] T. A. Marsland, T. Breitkreutz, and S. Sutphen, "NMP—A network multi-processor," Tech. Rep. 88-12, Dep. Comput. Sci., Univ. of Alberta, Dec. 1988.
[23] J. T. McCall, J. G. Tront, F. G. Gray, R. M. Haralick, and W. M. McCormack, "Parallel computer architectures and problem solving strategies for consistent labeling problem," *IEEE Trans. Comput.*, vol. C-34, no. 11, pp. 973–980, Nov. 1985.
[24] P. Mehrotra and T. W. Pratt, "Language concepts for distributed processing of large arrays," in *Proc. Symp. Principles Distributed Comput.*, Aug. 1982, pp. 19–28.
[25] A. N. Mucciardi and E. E. Gose, "A comparison of seven techniques for choosing subsets of pattern recognition properties," *IEEE Trans. Comput.*, vol. C-20, pp. 1023–1031, Sept. 1971.
[26] D. A. Nichols, "Using idle workstations in a shared computing environment," in *Proc. Eleventh ACM Symp. Oper. Syst. Principles,* 1987, pp. 5–12.
[27] M. L. Powell and B. P. Miller, "Process Migration in DEMOS/MP," in *Proc. Ninth ACM Symp. Oper. Syst. Principles,* Dec. 1983, pp. 110–119.
[28] R. F. Rashid, "Threads of a new system," *Unix Rev.*, Aug. 1986, pp. 37–49.

Appendix
The Overall Structure of the Animation System

```
-----------------------------------------------------------------------------
/* File containing definition of frames used by modules */

#include "polygon.h" /* polygon.h contains the structure for poly_tbl */
#define MAXIMAGES 120

struct model_polyconv {
    int image_number;
};
struct polyconv_split {
    int image_number;
    int npoly;
    struct polygon poly_tbl[ MAXPOLY ];
};
-----------------------------------------------------------------------------
include "animation.h"                          /* Structure of Model module */

main()
{
    struct model_polyconv work;
    for( image = 0; image < MAXIMAGES; image++) { /* loop through images */
        compute_geometry();    /* do modeling and motion computation */
        write_data();                /* write the data on disk_file */
    }
                               /* Done!  Send work to polyconv process */
    work.image_number = frame;
    call PolyConv( work );
}
-----------------------------------------------------------------------------
#include "animation.h"                      /* Structure of PolyConv module */

PolyConv( job )
struct model_polyconv job;
{
    read_data();                                 /* read data from the disk */
    do_conversion();          /* view transformation etc.on the image */
    /* assign the polygon table to frame polycnv */
    polycnv.npoly = npoly;                  /* assign number of polygons */
    strncpy(polycnv.poly_tbl, polygon_table, npoly*sizeof(struct polygon) );
    call Split( polycnv );                             /* send data to Split */
}
-----------------------------------------------------------------------------
#include "animation.h"                         /* Structure of Split module */

Split( poly_table )
struct polyconv_split poly_table;
{
    do_hidden( poly_table ); /* Hidden surface removal and antialiasing */
    write_image();                                 /* store image on disk */
}
-----------------------------------------------------------------------------
```

[29] J. Schaeffer, "Distributed game-tree searching," *J. Parallel Distributed Comput.,* vol. 6, pp. 90–114, 1989.

[30] J. F. Shoch and J. A. Hupp, "The Worm programs—Early experience with a distributed computation," *Commun. ACM,* vol. 25, no. 3, pp. 172–180, Mar. 1982.

[31] G. Singh and M. Green, "Visual programming of graphical user interfaces," in *Proc. Workshop on Visual Languages,* Linkoping, Sweden, Aug. 1987, pp. 161–173.

[32] ——, "A high-level user interface management system," in *Proc. ACM SIGCHI'89,* New York, Apr. 1989.

[33] A. Singh, J. Schaeffer, and M. Green, "Structuring distributed algorithms in a workstation environment: The FrameWorks approach," in *Proc. Int. Conf., Parallel Processing,* vol. 2, Aug. 1989, pp. 89–97.

[34] A. Singh, "FrameWorks: A distributed computing environment," Ph.D. dissertation, Dep. Comput. Sci., Univ. of Alberta, 1990, in preparation.

[35] A. Singh, J. Schaeffer, and M. Green, "FrameWorks user manual," Dep. Comput. Science, Univ. of Alberta, 1990, in preparation.

[36] M. Theimer, K. Lantz, and D. Cheriton, "Preemptable remote execution facilities for the V system," in *Proc. Tenth ACM Symp. Oper. Syst. Principles,* Dec. 1985, pp. 2–12.

[37] B. Walker, G. Popek, and E. English *et al.,* "The LOCUS distributed operating system," in *Proc. Ninth ACM Symp. Oper. Syst. Principles,* Dec. 1983, pp. 49–70.

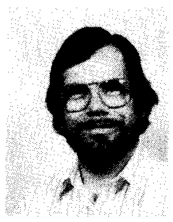
Ajit Singh finished his undergraduate study in Electronics and Communication Engineering in 1978 at Bihar Institute of Technology, India.

He worked for several years at the R & D department of Operations Research Group, the representative company for Sperry Univac Computers in India. During his M.Sc. work at University of Alberta, he worked on the subject of non-first-normal-form data models. Currently, for his Ph.D. degree work at University of Alberta, he is studying the problem of distributed computing using multicomputer networks.

Jonathan Schaeffer (S'82–M'84) received the B.Sc. degree in computer science from the University of Toronto in 1979, and the M.Math and Ph.D. degrees in computer science from the University of Waterloo in 1980 and 1986, respectively.

He joined the University of Alberta, Edmonton, Alta., Canada, in 1985 and now is an Associate Professor. His research interests include artificial intelligence (search algorithms, heuristics, and learning), and parallel and distributed computing. He is the author of the chess program *Phoenix,* which tied for first place in the 1986 World Computer Chess Championship, and co-author of the checkers program *Chinook,* 1989 World Computer Checkers Champion.

Dr. Schaeffer is a member of the Association for Computing Machinery, the American Association for Artificial Intelligence, and the Canadian Information Processing Society.



Mark Green (S'79–M'82) received the M.Sc. and Ph.D. degrees in computer science from the University of Toronto in 1979 and 1985.

He is an Associate Professor in the Computing Science Department at the University of Alberta. His principle research interests are user interfaces, computer animation, high performance graphics hardware, scientific computation, and distributed processing. He was an Assistant Professor at McMaster University from 1980 to 1983, and moved to the University of Albert in 1984.

Dr. Green founded the ACM UIST series of conferences on user interface software and was general chairman of the first of these conferences held in Banff Alberta on October 17–19, 1988. He is a member of the Association for Computing Machinery and SIAM.