University of Alberta

ENTERPRISE: AN INTERACTIVE GRAPHICAL PROGRAMMING ENVIRONMENT FOR DISTRIBUTED SOFTWARE DEVELOPMENT

by

Enoch Chan
Paul Lu
Jimmy Mohsin
Jonathan Schaeffer
Carol Smith
Duane Szafron
Pok Sze Wong

Technical Report TR 91-17
September 1991

DEPARTMENT OF COMPUTING SCIENCE
The University of Alberta
Edmonton, Alberta, Canada

Enterprise:

An Interactive Graphical Programming Environment

For Distributed Software Development

Enoch Chan
Paul Lu
Jimmy Mohsin
Jonathan Schaeffer
Carol Smith
Duane Szafron
Pok Sze Wong

Department of Computing Science,
University of Alberta,
Edmonton, Alberta,
CANADA T6G 2H1

{duane, jonathan}@cs.UAlberta.ca

ABSTRACT

Workstation environments have been in use for more than a decade now. Although a network of workstations together represents a large amount of aggregate computing power, single users often cannot utilize these resources for their applications. *Enterprise* is a programming environment for designing, coding, debugging, testing, monitoring, profiling and executing programs in a distributed hardware environment. *Enterprise* code looks like familiar sequential code; the parallelism is expressed graphically. The system automatically inserts the code necessary to handle communication, synchronization and fault tolerance, allowing the rapid construction of correct distributed programs. *Enterprise* programs run on a network of computers, absorbing the idle cycles on machines. The system supports load balancing, limited process migration, and dynamic distribution of work in environments with changing resource utilization. *Enterprise* offers a cost-effective method for increasing the productivity of programmers and the throughput of existing resources.

KEY WORDS: Distributed computing, parallel programming, programming environments, message passing, software engineering.

1. Introduction

Workstation environments have been in use for more than a decade now. Although a network of workstations together represents a large amount of computing power ("the network is the supercomputer"), single users often cannot utilize this power for their applications. In contrast to familiar sequential software that executes on a single machine, distributed software allows user applications to execute on many computers at once. Distributed software offers many advantages:

- programs potentially run faster, so the user spends less time waiting and more time working,
- combining the resources of several low-cost workstations can eliminate the need for more costly, high performance computers, and
- utilizing the processing time of idle workstations can reduce the total number of workstations needed by an organization.

However, writing distributed software is often perceived as a complicated endeavor. The design, implementation and testing of parallel software is considerably more difficult than comparable sequential software. Although for a large class of problems research has been done to develop efficient parallel algorithms, more often than not, these solutions do not find their way into practice. Finding a good parallel solution to a problem may only be a small fraction of the cost of implementing it. The majority of the costs usually results from taking into account concerns not found in the sequential environment, such as synchronization, deadlock, communication, heterogeneous computers and operating systems, and the complexity of debugging and testing programs that may be non-deterministic due to concurrent execution.

Further, harnessing the computing power of a network of machines poses some interesting problems. First, the processors available to an 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 parallelism that can be effectively implemented. Third, users do not want to become experts in networking or low-level communication protocols to utilize the potential parallelism. 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. There is a need for a mechanism that can be used to produce parallel and distributed software quickly, economically and reliably to take advantage of the untapped potential of idle workstations.

This mechanism must bridge the perceived complexity gap between distributed and sequential software, without forcing the user to undergo extensive re-training.

Enterprise is a programming environment for designing, coding, debugging, testing, monitoring, profiling and executing programs in a distributed hardware environment. *Enterprise* code looks like familiar sequential code since the parallelism is expressed graphically. The system automatically inserts the code necessary to handle communication, synchronization and fault tolerance, allowing the rapid construction of correct distributed programs. This bridges the complexity gap between distributed and sequential software. *Enterprise* programs run on a network of computers, absorbing the idle cycles on machines. The system supports load balancing, limited process migration, and dynamic distribution of work in environments with changing resource utilization. *Enterprise* offers a cost-effective method for increasing the productivity of programmers and the throughput of existing resources.

The *Enterprise* system is built with four objectives in mind:

1) to provide a simple high-level mechanism for specifying parallelism that is independent of low-level synchronization and communication protocols,

2) to provide transparent access to heterogeneous computers, compilers, languages, networks and operating systems,

3) to support the parallelization of existing programs to take advantage of the investment in the existing software legacy, and

4) to be a complete programming environment, to eliminate the overhead that arises from switching between it and other programming environments.

Enterprise has a number of features that distinguish it from other parallel and distributed program development tools (see Section 7):

1) Programs are written in a sequential programming language that is augmented by new semantics for procedure calls that allows them to be executed in parallel. Users do not deal with implementation details such as communication and synchronization. Instead, *Enterprise* inserts all of the necessary communication protocols automatically into the user's code.

2) *Enterprise* can generate these protocols automatically because most large-grained parallel programs make use of a small number of regular techniques, such as pipelines, master/slave processes, divide and conquer, etc. In *Enterprise*, the user

specifies the desired technique at a high level by manipulating icons using the graphical user interface. The user-written code that implements the parallel procedure is independent of the parallelization technique selected (although the code generated by *Enterprise* certainly is not). The de-coupling between the procedure that is to be parallelized and the parallelization technique allows applications to be easily adapted to a varying number and type of available processors without changing user-written code. It also provides a simple mechanism for experimentation and evaluation of how the various techniques fare on the user's particular application.

3) To simplify the way in which parallelism is expressed, *Enterprise* uses an analogy between the structure of a parallel program and the structure of an organization. The analogy eliminates inconsistent terminology (pipelines, master, slave, etc.) and replaces it with a uniform set of *assets* (contracts, departments, individuals, etc.). Organizations are inherently parallel and, often, have efficient parallelism. This analogy allows the programmer a different (but familiar) model for designing parallel programs. Although the use of analogies in computing science has both advocates (Booth, Schaeffer and Gentleman, 1982) and detractors (Dijkstra, 1982), we believe they will prove quite useful, as they have in object-oriented programming.

4) Several of the parallelization techniques supported by *Enterprise* can distribute work dynamically in environments with changing resource utilization. For example, a *contract* can be used to distribute work to a variable number of identical subordinates. A contract uses as many idle machines as possible to help complete the task. During peak hours, a program may only be able to use a handful of processors, while in the evening many more may be available to help fulfill the contract.

5) In most parallel/distributed computing tools, the user is required to draw communication graphs. The user usually draws a diagram connecting nodes (processes) by arcs (communication paths). In *Enterprise*, a similar diagram is created, but the user is spared the tedium of drawing the details. Instead, the user need only edit the diagram by coercing and expanding nodes. Coercion allows the user to express how a structure (asset) communicates with its neighbors; expanding a node allows the user to explore the hierarchical structuring of the application.

6) The user can exercise a desired amount of control over the mapping of processes to processors. Hiding hardware realities of the environment can result in major performance degradation of distributed systems (Jones and Schwartz, 1980).

However, *Enterprise* is quite flexible in this regard. Using a high-level notation, the user can specify the processor assignments completely, partially, or leave it entirely up to the environment.

7) *Enterprise* 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.

Using the graphical user interface, the user draws a diagram of the parallel computation and writes sequential code that is devoid of any parallel constructs. Based on the user's diagram, *Enterprise* automatically inserts all the necessary code for controlling the parallelism, communication, synchronization and fault tolerance. It then compiles the routines, dynamically assigns processes to processors and establishes the necessary connections. 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.

Enterprise is a complete redesign and rewrite of our successful *FrameWorks* system (Singh, Schaeffer and Green, 1989a, 1989b, 1991; Singh, 1991). *FrameWorks* was a prototype system used to demonstrate that some of the concepts worked. *Enterprise* contains many of the ideas in *FrameWorks*, but with major changes to the user's view of a parallel computation and major improvements in the tools available to the user. *Enterprise* represents several major advances over *FrameWorks*:

1) The notion of combining partial templates (input, output and body) in *FrameWorks* has been replaced by a small number of complete parallelization techniques called *assets*. Many of the combinations of templates in *FrameWorks* were illegal or impractical. *Enterprise* assets are equivalent to the useful combinations of *FrameWorks* templates.

2) *Enterprise* has removed the syntactic impositions of *FrameWorks*, including the use of keywords (*call* and *reply*) and the restriction on the number of parameters that can be passed to a process (from 1 to arbitrary). *Enterprise* automatically differentiates between procedure calls and process calls (*module calls*) based on the asset diagram.

3) The semantics of an *Enterprise* routine are the same as the semantics of sequential code. *FrameWorks* allowed calls to be made to processes either synchronously or asynchronously, but the user had to explicitly handle the case of an asynchronous call

returning a result. The *Enterprise* semantics hides this implicitly in the user's code, so the user need not differentiate between a procedure and module call.

4) The use of analogies was limited in *FrameWorks*. *Enterprise* uses analogies extensively as a means of creating a model of parallel computation that is easily explained, readily understood and consistent.

5) *Enterprise* has been implemented with portability in mind, unlike *FrameWorks*. By building our system using the X Window System (Scheifler and Gettys, 1986) and ISIS (Birman et al., 1991a, 1991b), the system is more powerful and will be usable on a wider variety of systems.

Section 2 contains a walk-through of a typical *Enterprise* session, illustrating the computational model and the graphical user interface. Section 3 describes the two key components of the *Enterprise* model: the semantics of the sequential code that the user writes, and the types of parallelism (assets) supported. Section 4 describes the user interface. In Section 5, the architecture of the system and its implementation can be found. Section 6 discusses the test suite we are using for benchmarking parallel and distributed programs. Section 7 discusses other parallel programming environments and contrasts them with *Enterprise*. *Enterprise* is currently being implemented and the project's status is discussed in Section 8.

2. Program Design in Enterprise

This section presents a simple example of how *Enterprise* can be used to construct a distributed program. Consider an animation program that displays a group of fish swimming across a display screen. There are three fundamental operations in the program: *Model*, *PolyConv* and *Split* with the following functionality:

- *Model*: Computes the location and motion of each object in a frame, stores the results in a file, calls *PolyConv* to process the frame and proceeds to the next frame.

- *PolyConv*: Reads a frame from the disk file, performs some data format transformations, viewing transformations, projections, sorts, back-face removal and calls *Split*, passing it a transformed frame and a sequence number.

- *Split*: Performs hidden surface removal, anti-aliasing and stores the rendered image in a file.

This problem was contributed by a research group in our Department and is obviously more complex than portrayed by our brief description. Examining the structure of the program shows that *Model* consists of a loop that, for each frame in the animation, performs some work on the frame and calls *PolyConv* with the results. *PolyConv* manipulates the image received from *Model* and calls *Split*. *Split* does the final polishing of the frame and writes the final image to disk.

An *Enterprise* user manipulates icons that represent high-level program components called *assets* (defined in the next section of this paper). For this example, assume that an asset represents a single C-language procedure/function, called an *entry procedure*, together with a collection of support procedures used by the entry procedure, all contained in a single file. A program will consist of several assets. In this example, there will be three assets: *Model*, *PolyConv* and *Split*.

After starting *Enterprise* and choosing *New Program* from the main menu, a dialog box appears asking for the name of the program. After entering the name of the program, *Animation* in this case, a single asset appears that represents the entire program as shown in Figure 1.

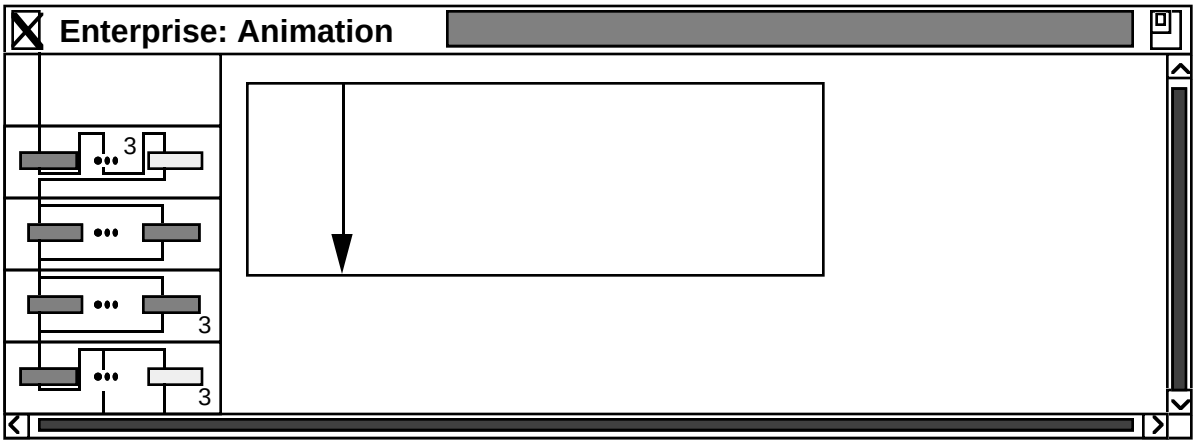

Figure 1: A new program consisting of a single individual asset.

The Enterprise window consists of an asset palette containing one icon for each asset kind and a canvas containing the program. A new program contains one *individual* asset that represents a sequential program component. The code for the procedures *Model*, *PolyConv* and *Split* could be associated with this single individual asset and run as a sequential program. However, there is no reason why *Model* should wait until *PolyConv* completes execution of the first animation frame to start processing the second frame. Similarly,*PolyConv* dows not need to wait for *Split*. Therefore, the individual asset can be *coerced* to (replaced by) a *line* asset by selecting the individual asset and then selecting the *line* icon from the asset palette. After entering three as the length of the line, the individual is coerced to a three component line as shown in Figure 2.

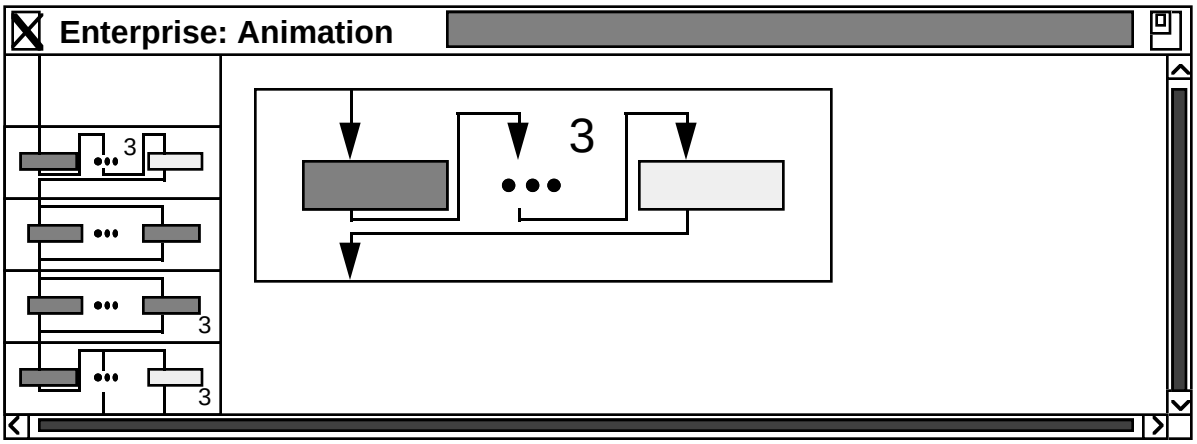

Figure 2: A collapsed line asset composed of three assets.

The line shown in the figure is *collapsed*; that is, its components cannot be seen. By selecting the line and choosing *Expand Asset* from the menu, the line is expanded so that the three individual assets that it contains are visible, as shown in Figure 3.

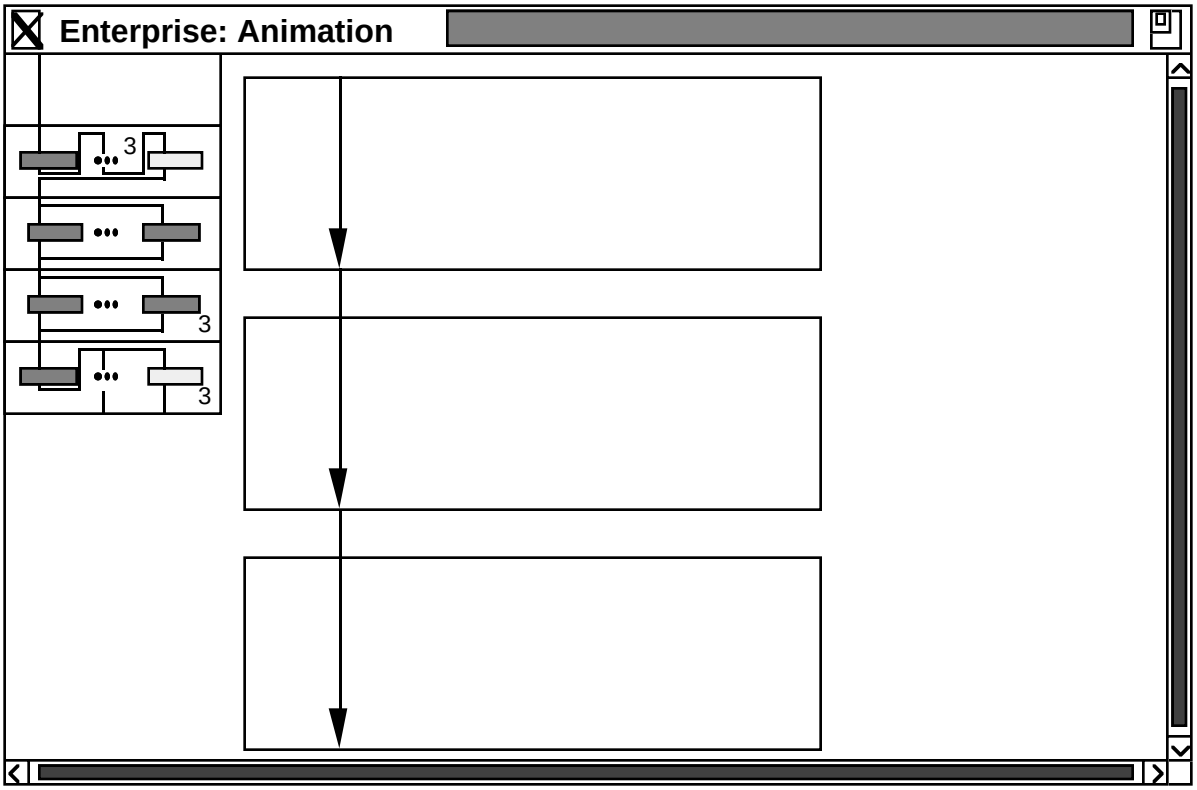

Figure 3: An expanded line asset composed of three individual assets.

To name an asset, the user selects it, chooses *Name Asset* from the menu and enters the name of the asset in the dialog box that appears. When the dialog box is closed, the name appears on the icon. To enter the C-language code for an asset, select it, choose *Edit Code* from the menu and enter the code using your favorite text editor, as shown in Figure 4. Note that *Enterprise* automatically names the output file for the code you enter as the name of the asset with a ".e" suffix. In the example, the *Model* code would be saved in the file Model.e.

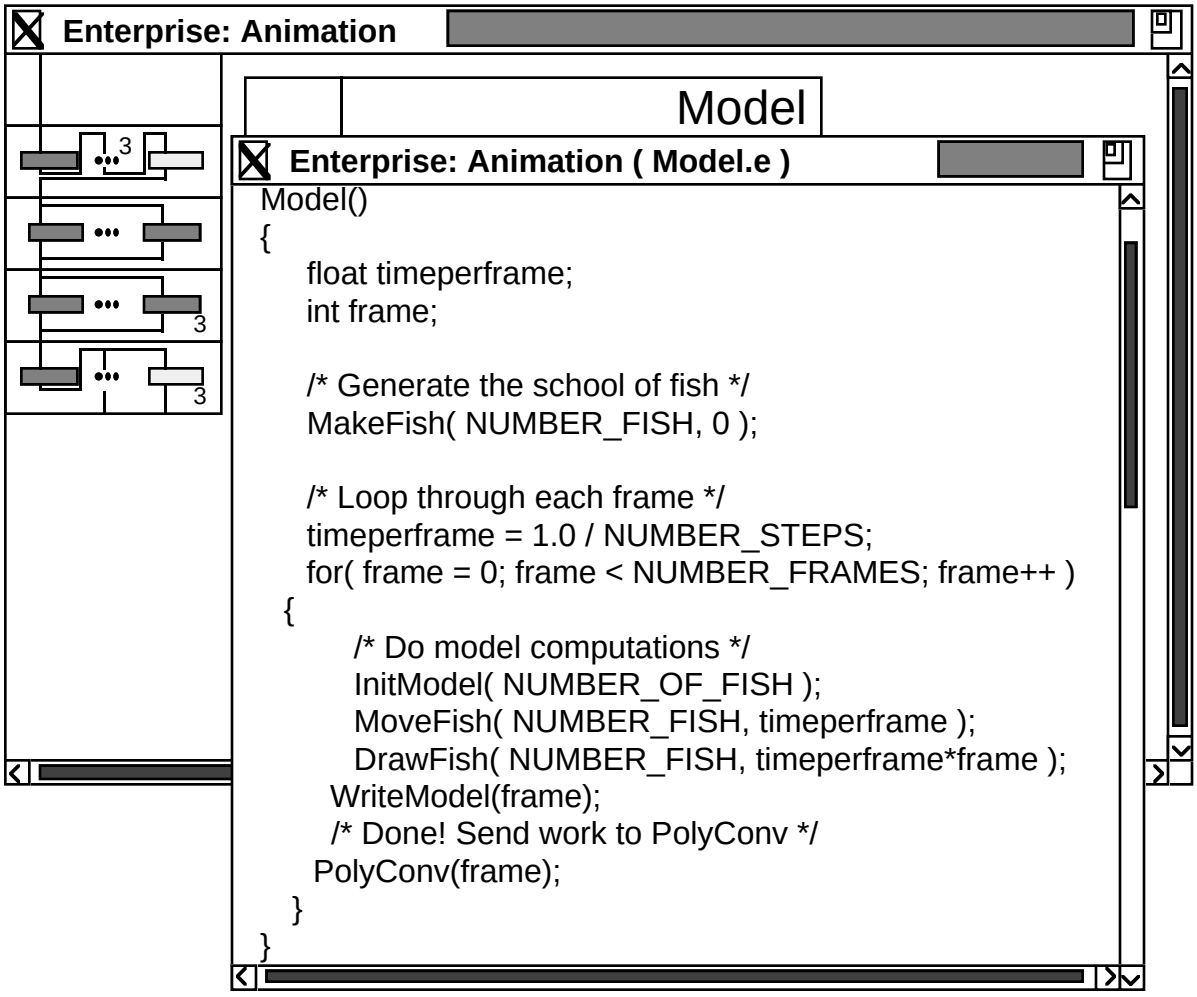

Figure 4: Entering the code for an asset.

The code for the entry procedures *Model*, *PolyConv* and *Split* are shown in Appendix A. To compile the application, the user selects *Compile* from a menu and the *Enterprise* system automatically inserts the code to handle the distributed computation, compiles the program and reports any errors back to the user. Appendix B shows the code that *Enterprise* inserts into the application to handle the communication, synchronization and fault tolerance (using the code in Appendix A, with the diagram in Figure 6). Once the program is compiled, the user selects the

Execute menu item and *Enterprise* finds as many processors as are necessary to start the program, initiates processes on the processors, monitors the load on the machines and (if a contract is used) dynamically adds additional processors to the application as they become available. For this animation example, a speed-up of 1.7 was obtained by using a line running on three processors instead of a single individual asset (a sequential program)†. Note that any timings are subject to large variations, depending on the number of available processors and the amount of traffic on the network. In this particular case, the timings were done late at night when sufficient processors were available and network traffic was light.

The strength of the *Enterprise* model can be seen by the ease with which it is possible to take a program and experiment with alternate parallelization techniques without changing the C-language source code. For example, by selecting the *Split* asset and selecting a contract asset from the asset palette, the *Split* asset is coerced from an individual to a contract as shown in Figure 5.

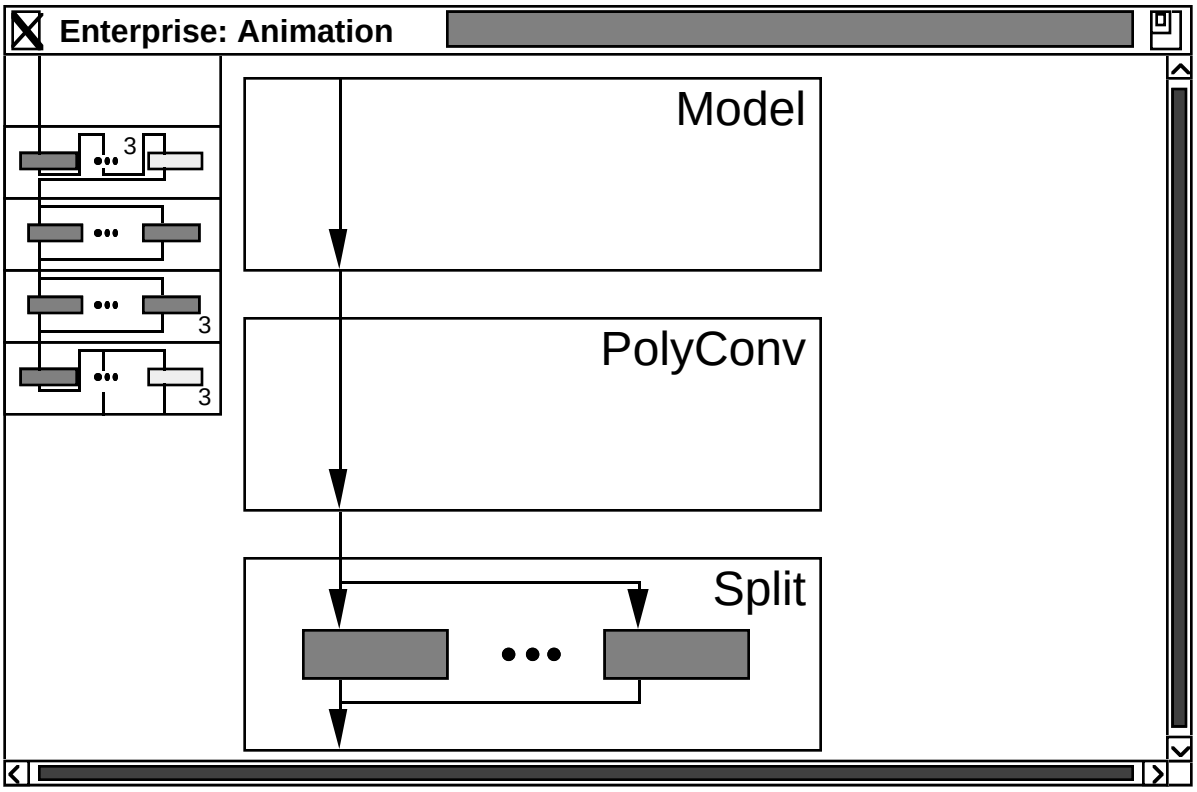

Figure 5: A collapsed contract asset in a line.

In *Enterprise*, each asset represents at least one process. If a call is made to the *individual Split*, it is executed as a process and if a subsequent call is made to *Split* before the first call is

† These timings were obtained with the *FrameWorks* system, but *Enterprise* should be subjectively similar.

complete, the second call must wait for the first call to finish. However, when a *contract* is executed, multiple processes can be used to execute multiple calls concurrently. That is, when *PolyConv* calls *Split*, a process is initiated and if a subsequent call is made to *Split* before the first call is done then a second process is initiated (if there is an available machine). *Enterprise* contracts are dynamic so that a contract may use as many processors as are available on the network.

Coercing the *Split* asset to a contract results in as much as a 5.7-fold speed-up (using a dynamically varying number of processors) compared to the sequential animation program, depending on when the program is run†. Of course, there is no reason why the user cannot coerce the *PolyConv* asset from an individual to a contract as well. However, this only resulted in a speedup of 6.0. This implies that the *Split* procedure is the real bottleneck in the animation program. That is, an individual *PolyConv* can almost keep up with its calls by *Model* but an individual *Split* cannot keep up with its calls by *PolyConv*.

Enterprise can be used to further experiment with this application. For example, the *Split* contract currently contains a single individual. By selecting *Split* and choosing *Expand Asset* from the menu, this individual can be viewed as shown in Figure 6.

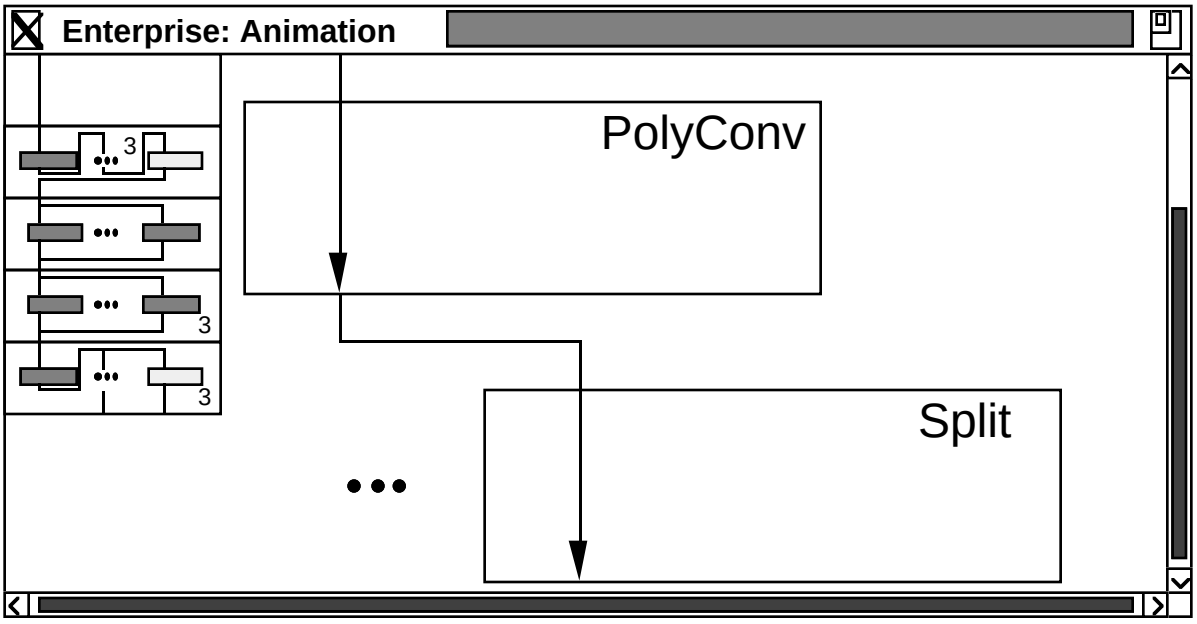

Figure 6: An expanded contract asset.

Note that the asset *Model* has been scrolled off the display and the components of the contract asset are indented to show the scope of the contract. The ... indicates that there are

† These timings were obtained with the *FrameWorks* system, but *Enterprise* should be subjectively similar.

multiple copies of the indented assets available since the contract contains a dynamic number of identical assets.

If the C code for the individual *Split* asset contained a sequence of procedure calls (or even a single procedure call at the end of it), the individual asset could be coerced to a line where each of the assets in the line represented one of the procedure calls, as shown in Figure 7. To proceed with this approach, the line would be expanded and the C-language procedure called by *Split* (that is currently inside the *Split* asset) would be moved to the second asset in the line and become an entry procedure of its own individual asset.

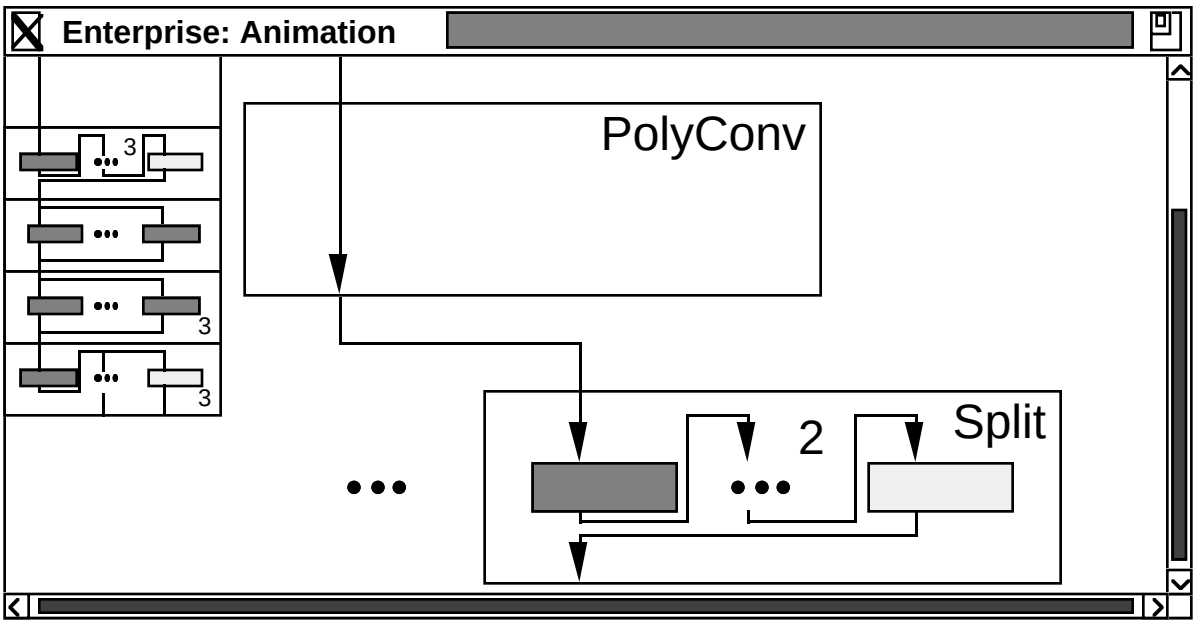

Figure 7: A hierarchy of assets where a line contains a contract that contains a line.

Several other asset kinds are supported by *Enterprise* and they can be combined in arbitrary hierarchies. The next section details the asset kinds that are available.

3. The Enterprise Model

The overall organization of a parallel or distributed program in *Enterprise* is similar to the organization of a sequential program. The structure of an application program is, in fact, unaffected whether it is intended for sequential or distributed execution. The user views an *Enterprise* program as a collection of modules. Each *module* consists of a single *entry procedure* that can be called by other modules and a (possibly empty) collection of *internal procedures* that are callable only by other procedures in that module. No common variables among modules are

allowed. 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.

Within any module, the code is executed sequentially. For example, a sequential program simply consists of a single module whose entry procedure is the main program. *Enterprise* introduces parallelism by allowing the user to specify the way in which the modules interact. Module interaction is specified by two factors: the *role* of a module and the *call* to a module. The role of a module defines which one of a fixed set of parallelization techniques (*asset* kinds) the module will use when it is invoked. The call to a module defines the identity of the called module, the information passed and the information returned. The role of a module is specified graphically while the call is specified in the code.

3.1 Module Calls

In a sequential program, procedures communicate using procedure calls. The calling procedure, say A, contains a procedure call to a procedure, say B, that includes a list of arguments. When the call is made, procedure A is suspended and procedure B is activated. Procedure B can make use of the information passed as arguments. When procedure B has finished execution, it can communicate results to procedure A via side-effects to the arguments and/or by returning a value if the procedure is in fact a function.

Enterprise module calls are similar to sequential procedure calls. As with procedure and function calls, it is useful to differentiate between module calls that return a result and those that do not. Module calls that return a result are called *f-calls* (function calls) and module calls that do not return a result are called *p-calls* (procedure calls).

Enterprise module calls differ from sequential calls in the following ways:

1) Arguments can not be pointers, nor can they contain any pointers. This implies that in the C-language version of *Enterprise*, module calls cannot return values by side-effects since the C-language uses call by value.

2) When a module, say A, calls another module, say B, module A is not suspended. Instead, module A continues to execute. However, if the call to module B was an f-call, then module A would suspend itself when it tried to use the function result, if module B had not yet finished execution.

There is no syntactic difference between procedure calls and module calls. This makes it easier to transform sequential programs to parallel ones and makes it trivial to change parallelization techniques using the graphical user interface without making changes to the code.

In *Enterprise*, an f-call is not necessarily blocking. Instead, the caller blocks only if the result is needed and the called module has not yet returned. Consider the following example:

```
result = B(data);
/* some other code */
value = result + 1;
```

When this code is executed, the calling module, say A, only blocks when the statement "*value = result + 1;*" is executed and only if module B has not completed execution. This concept is similar to the work on futures in object-oriented programming (Chatterjee, 1989). The p-call in the statement:

```
B(data);
/* some other code */
```

is non-blocking, so that A continues to execute concurrently with B. Of course in this case, B does not return a result to A.

3.2 Module Roles and Assets

The role of a module is based solely on a parallelization technique and is independent of its call. There are a fixed number of pre-defined roles corresponding to *asset* kinds. For example, in the previous section, the role of the *Split* module was changed from an individual to a contract without changing the call.

We have created an analogy between *Enterprise* programs and the structure of an organization to help describe module roles. In general, an organization has various assets available to perform its tasks. For example, a large task could be divided into sub-tasks where various sub-tasks are given to different parts of the organization (divisions, departments, pools, lines and/or individuals) to perform in parallel. Some tasks could even be completed by contract where the organization is not directly concerned about the nature or number of individuals that perform it. In addition, an organization usually provides many standard *services* (like time keeping, information storage and retrieval, etc.) that are available on demand to improve its functionality.

Currently, *Enterprise* supports the roles corresponding to seven different asset kinds: individual, line, pool, contract, department, division and service.

3.2.1 Individual

An *individual* contains no other assets. An individual is analogous to an individual person in an organization. For example, a clerk in a grocery store is an individual. When called, an individual executes its sequential code to completion. Therefore, any subsequent call to the same individual must wait until the previous call is finished. An individual may be called by any external asset using its name. Individuals can be viewed as a process executing a sequential program.

3.2.2 Line

A *line* contains a fixed number of heterogeneous assets in a fixed order. Each asset contains a call to the next asset in the line. A line is analogous to a construction, manufacturing or assembly line in an organization where at each point in the line, the work of the previous asset is refined. For example, a line might consist of an individual who takes an order, a department that fills it and an individual that addresses the package and mails it. A subsequent call to the line waits only until the first asset is finished its sub-task of the previous call, not until the entire line is finished. The first asset in a line serves as the *receptionist* for the line and is the only asset that is externally visible. That is, the first asset of a line is the only asset that may be called from an external asset and it shares its name with the line asset for this purpose. Lines are more often referred to as pipelines in the literature.

3.2.3 Pool

A *pool* contains a fixed number of identical assets. A pool is analogous to a pool in an organization where each pool member performs an identical task. For example, consider a pool of telephone operators. When a call is made to the pool, an idle asset executes the call. However, if all assets are busy, then the call waits for one of the assets to finish. Since pool members are externally indistinguishable, an external call cannot select a particular pool asset. Therefore, pool assets are called by external assets using a single name that is shared with the pool asset for this purpose. Since all assets in a pool are identical, they also share the same code. A pool is analogous to a master-slave construct with a fixed number of slaves.

3.2.4 Contract

A *contract* contains a collection of identical assets, so it is similar to a pool. However, the number of assets in a contract is dynamic and depends on the number of processors that are free at any time. A contract is analogous to a contract that an organization lets for the performance of a collection of identical tasks. For example, an organization might let a contract to a courier company for the delivery of its packages. When a package must be delivered, the courier company is informed. The organization doesn't care how many resources the courier company uses or the route it takes to deliver the packages. The delivery time can be affected by the number of resources used by the courier company and the amount of competing traffic. Similarly when an *Enterprise* call is made to a contract, an idle asset executes the call. However, if all assets are busy, then the call waits for one of the assets to finish. As is the case with a pool asset, a contract asset shares a common name with the identical assets it contains and these component assets also have common code. A contract is equivalent to a dynamic master-slave construct, where the number of slaves varies in response to program needs (demand) and resource utilization (environment).

3.2.5 Department

A *department* contains a fixed number of heterogeneous assets. Every department has a single receptionist asset that shares its name with the department so that the department can be called by external assets. However, unlike a line, the other assets in a department do not call each other in a fixed sequential order. Instead, all other assets in the department are called directly by the receptionist. A department is analogous to a department in an organization where a receptionist is responsible for directing all incoming communications to the appropriate place. Note that in our analogy, a department consists of a collection of assets of any kind: individuals, departments, lines, etc. The department has no analogous term in the literature.

3.2.6 Division

A *division* contains a hierarchical collection of identical assets with a fixed breadth and depth where work is divided and distributed at each level. Every division has a single receptionist asset that shares its name with the division so that the division can be called by external assets. Divisions can be used to parallelize divide and conquer computations.

3.2.7 Service

A *service* contains no other assets. However, unlike an individual that can only answer a single call at any one time, a service may be used by more than one asset at the same time. A service is analogous to any asset in an organization that is not consumed by use and whose order of use is not significant. For example, a clock on the wall and a counter that records the total number of vehicles that have passed through several service lanes can be considered services. A service may be called by any external asset using its name.

3.2.8 Other Asset Kinds

Assimilated versions of these assets are being considered at this time. An assimilated asset is one that has an assimilator that can be called from an external asset in addition to the receptionist. The assimilator is called to obtain the assimilated results of the computation performed by the asset.

3.3 Enterprise Diagrams

An *Enterprise* diagram can be built from any combination of assets. For example, one can construct a contract, where each asset is itself a line of individuals. The model allows the user to coerce an asset from one kind to another *without any changes to the user's source code*. The only change that might occur is the gathering or separation of functions from one file to another. For example, if a line is used for the animation example, there would be three individuals (*Model*, *PolyConv* and *Split*) each having their code in a separate file. If the line is coerced into an individual, the code needs to be gathered together (either in the same file or by using libraries).

There are ways that *Enterprise* could do this management automatically, but there are some issues we have yet to resolve.

4. The User Interface

Enterprise's user interface was designed to allow a user to express parallelism in a simple graphical manner. Although other parallel programming environments support graphical views, these views are either non-editable or are edited by drawing nodes and arcs that represent processes and communication paths (see Section 7). In *Enterprise*, the application graph is an asset graph and it is constructed in a novel way. The user starts with an individual asset and constructs the graph by coercing and expanding individual icons as illustrated in Section 2. This approach has several advantages over an arbitrary graph structure:

1) *Enterprise* assets represent high-level parallelization techniques, not individual processes. For example, contracts, pools, departments and divisions each represent multiple processes. This allows the user to design at a higher level of abstraction.

2) Assets themselves are not drawn and connected by the user in an arbitrary manner. Instead, assets are coerced and expanded to create a program. This reduces the drawing errors that result from indiscriminately connecting and disconnecting nodes using arcs.

3) The structure of an *Enterprise* program clearly indicates the type and degree of parallelism. The flow of information is from top to bottom while the degree of parallelism is from left to right. In other words, the length of the graph represents the critical path of an application, and the width reflects the degree of the parallelism.

4) *Enterprise* manages program complexity by allowing assets to be expanded and collapsed so that the program can be viewed at different levels of abstraction.

5) Experimentation is encouraged because the parallelization technique is specified graphically and is independent of the code.

The main *Enterprise* window contains a canvas and a palette. The canvas is used to display and coerce the graphical representation of the program. The palette contains one icon for each asset kind. Section 3 describes the asset kinds that are currently supported. When *Enterprise* is started a new program is displayed, consisting of a single individual asset.

A mouse is used to select an asset on the screen. If an asset is selected and the user clicks on an icon in the asset palette then the selected asset is coerced to an asset of the chosen kind. If the chosen kind is a line or a pool, then a dialog box asks the user for the number of components in the coerced asset before the coercion takes place.

A pop-up menu can be used to choose an operation. The following operations can be performed:

1) **Name** or re-name the selected asset.

2) **Expand** the selected asset so its component assets are displayed.

3) **Collapse** the selected asset so that it is no longer displayed, but its "parent" asset is.

4) Open an edit window on the **Code** of the selected asset.

5) **Save** the current program to disk.

6) Create a **New** program consisting of a single individual asset and display it in the main *Enterprise* window. If any changes have been made to the current program then use a dialog box to find out if the user wants to save the changes and if so, first save them to disk.

7) **Open** (load) an existing program by prompting the user for the program name and displaying it in the main *Enterprise* window. If any changes have been made to the current program then use a dialog box to find out if the user wants to save the changes and if so, first save them to disk.

8) **Compile** the current program.

9) **Execute** the current program.

10) Toggle the **Animation** flag. When this flag is set and the program is executed, the graph is animated.

11) Toggle the **Debug** flag. When the debug flag is set, the program executes in debug mode (allowing features such as tracing and break points).

12) Open a monitor window that displays **Statistics** about processor utilization.

13) Assign a **Version** number to the source code of the current program.

14) Retrieve a **Previous** version of the selected asset's source code.

15) Specify machine **Constraints** for execution of the selected asset.

5. The Architecture

5.1 System Design

As the name *Enterprise* implies, distributed application programs are modelled after an organization. However, the same analogy is used for the logical components of the *Enterprise* system itself. There are six logical components in the architecture of *Enterprise*: an interface manager, an application manager, a code librarian, an execution manager, a monitoring/debugging manager and a resource secretary as shown in Figure 8.

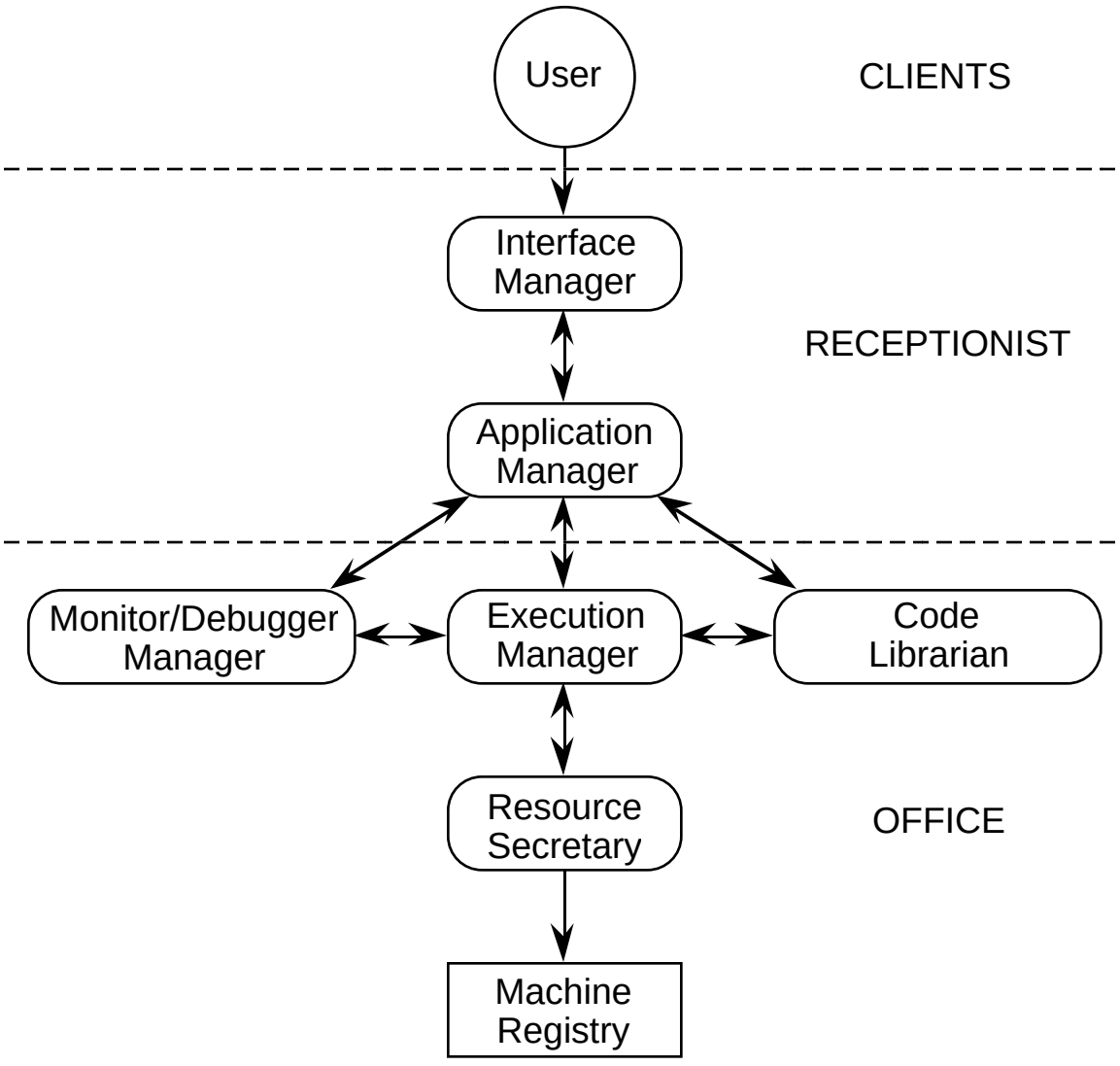

Figure 8. The architecture of *Enterprise*.

5.1.1 Interface Manager

The graphical interface used in *Enterprise* provides an environment for editing, debugging, compiling, configuring, executing and monitoring parallel programs. The user can develop parallel applications in a single unified programming environment. The tool includes an asset graph editor for users to design their applications using the organization analogy.

An object-oriented design was used for the user-interface. For example, each asset is an instance of an asset class and is responsible for knowing its name, attributes (like the length of a line asset or the size of a pool), components (like the components of a line), code, drawing itself, expanding itself, etc. Inheritance was used extensively since different asset classes share many responsibilities. Of course the other interface components (windows, menus and dialogs) are also objects.

The current implementation of the user interface was developed under the X Window System (Scheifler and Gettys 1986) on Sun workstations. This allows the interface to be easily ported to other workstations. The implementation was written in C++ using InterViews (Linton, Vlissides and Calder, 1989). This combination was chosen for easy integration with the rest of *Enterprise* (which is implemented in C), to remain faithful to the object-oriented design, to maintain portability and to reduce development time by utilizing the InterViews class library of interface objects.

Although a graphical user interface provides a simple and powerful programming environment, a textual interface is also needed since a user may want to re-configure a program when a graphics terminal is not available. Of course, the textual interface has limited capabilities compared to the graphical interface; for example the user will not be able to view the dynamic execution of a program.

5.1.2 Application Manager

The application manager is the control center for *Enterprise*. All of the permanent information about an application is maintained by the application manager. This includes the asset graph and the source code. The only way for the user to access application-specific information is through the application manager. This manager is also responsible for ensuring information provided by the user is correct and consistent with the current state of the program design.

5.1.3 Code Librarian

The role of the code librarian is to manage the source and object code of different modules in a parallel application. Since *Enterprise* is designed with a heterogeneous network of workstations in mind, the librarian may need to maintain multiple object codes for a variety of architectures. The

librarian builds and executes a *makefile* that is incrementally maintained as the user changes the specifications of an application, and is parameterized to support different target machines.

The librarian has to know where the source code and the corresponding object files for a particular asset are located. When an *Enterprise* application is about to be started, the librarian determines if all the required executables are available and, if not, performs the necessary compilations for the user. Compilation requires knowledge of the kinds of assets involved, which is available from the application manager. To compile an asset, the librarian inserts the appropriate *Enterprise* code into the module, depending on the asset kind. Re-compilation of an asset is only necessary when either the user changes the asset's code, the asset has been coerced to a different kind, or when the asset is to be run on a machine for which an executable is not available.

Requests for compilation come from two sources. First, the user can ask the interface manager to compile the application to reveal syntax and semantic errors. Second, the execution manager can request executables on demand at run-time. As machines become available, the code librarian informs the execution manager if an executable exists for that machine. If the executable is not available, the execution manager makes the decision as to whether to request the librarian to provide it or not.

5.1.4 Execution Manager

The execution of an *Enterprise* application is controlled by the execution manager. It is responsible for creating all the processes specified by the asset graph, and setting up the required communication channels. It uses the asset graph to discover the initial communication paths and the presence/absence of contracts. The decision on where a process is to be executed is made by the execution manager, implying that executables are general and do not have a specific machine name compiled into them.

One of the attributes associated with an asset is a *machine preferences list*. This list specifies any constraints on the machines to be used to run this asset. The default is that the program will run on any machine. The user, however, may choose to constrain the choice of machine in some way, say by execution speed or physical location. The method used to select machine preferences is similar to the technique employed in *FrameWorks*.

The resource secretary is responsible for providing information to the execution manager on which machines are available, and which machines are currently busy. Work on busy machines should be migrated to idle machines by the execution manager since assigning processes to machines is one of its responsibilities. In a homogeneous environment, this causes no difficulty. In a heterogeneous environment, problems occur. Which processes do you put on the fastest available processor? What if a very slow processor is available. Do you start using it, or wait for

a faster one to become available? These decisions become complicated in a dynamically changing environment and when contracts are present. We have not yet addressed these issues.

5.1.5 Resource Secretary

The list of available machines is maintained by the resource secretary in a *machine registry*. The machine registry is a list of specifications on each machine in the network, such as the machine name, its architecture and type, operating system version, compiler name and options, a speed rating, RAM size, type of monitor, etc. Periodically (a system adjustable parameter), the secretary receives an update on how busy each machine is. Changes in a machine's status (from idle to busy or visa-versa) are communicated to the execution manager.

Every machine in the network has a ".enterprise" file associated with it. In it, the machine's owner can specify when *Enterprise* programs are allowed to use the machine. The default is to allow programs to run on evenings and weekends only.

5.1.6 Monitor/Debugger Manager

Monitoring and debugging facilities are very important in the domain of parallel programming. These facilities should allow a user to identify the bottleneck in a program and to find potential synchronization problems by monitoring the execution of the program. The monitoring approach used in *Enterprise* is program animation. This is done by time stamping every call to an asset and either sending this information to the interface (for a dynamic, real-time portrayal of the program's execution), or saving the information in a file to allow the program to be replayed at the user's leisure. The debugger allows the user to step through an application at the asset call level and to set breakpoints any time an asset is called.

5.2 System Implementation

FrameWorks was built on NMP (Marsland, Breitkreutz and Sutphen, 1991), but its many limitations forced us to consider alternatives. *Enterprise* uses the ISIS package to do all the low-level communications. ISIS provides a high-level set of function calls to handle process creation and termination, communication, synchronization and fault tolerance for a heterogeneous collection of machines (Birman et al., 1991a, 1991b).

Enterprise is implemented as an ISIS program. Not only are the executables that *Enterprise* produces ISIS programs, but the architecture of *Enterprise* is designed as a multiple-process program that communicates using ISIS. All the components of the architecture described in Section 5.1 are communicating processes, except for the Application Manager which is conceptually a separate process, but for efficiency is implemented as part of the Interface Manager.

In ISIS, processes are grouped together and named as a unit. A process group may contain just a single member, but will often consist of a number of processes residing on machines anywhere in the system. Message passing in ISIS is performed by broadcasting to a process group as a whole, and collecting replies from the members in the same call. The broadcast can block, waiting for the reply, or it can be made to fork off as a task in which case the caller will rendezvous with it later to collect the results.

5.2.1 Fault Tolerance

In a distributed environment, it is desirable for a system to provide fault tolerance and process migration for distributed applications. Fault tolerance yields a more robust application. The ability to migrate processes from busy machines to less busy ones keeps the user community happy without significantly increasing the execution time of the application.

ISIS has extensive support for fault tolerance. Consider the case of implementing an individual. One approach for fault tolerance supported by ISIS is redundant computation. Several processes undertake the same task, with the caller waiting for the first response and continuing to execute without waiting for a response from the others. It achieves fault tolerance at the cost of wasted CPU cycles.

Another approach in ISIS is the coordinator-cohort computation (the standby approach). The method works by ranking the members of a process group and then labeling the lowest ranking member as the coordinator for a request. This process will execute the request and broadcast a reply to the caller. The other members are cohorts; they are passive unless a failure prevents the coordinator from terminating normally, in which case they take over one by one, in rank order. The coordinator is also able to send the cohorts a copy of the answer returned to the caller at the termination of the computation. ISIS does not add extra cohorts to a coordinator-cohort algorithm when it is already running, so fault tolerance is limited by the number of cohorts at the start of the computation. If several requests arrive concurrently, the job of being coordinator will be split over the members of the group in a uniform manner. Thus a single process may be coordinator for one or two requests while being cohorts for others. Moreover, there may be several coordinators at one time for different requests. The scheme thus exploits the distributed processing power of the group in a fault tolerant way.

If an asset is a contract or a department, when some of the processes fail to complete, the number of replies will be less than the number of broadcast messages sent. In this case, the program needs to either recompute the missing piece, or reissue the entire request. Using this approach, the application handles the recovery from fault.

Both replication and standby (coordinator-cohort) approaches achieve only partial failure resiliency. Computation results will be lost when the failure involves all the processors responsible for the computation. In ISIS, a tool is provided to log a process' events and, when failure occurs, to recover the state it was in from the log in the case where the process group responsible for the computation experienced a total failure. This is done by logging a copy of the checkpoint of the computation state onto stable storage.

5.2.2 Process Migration

Migrating a process from one machine to another involves (Eskicioglu, 1990):

1) suspending the process on the source machine,
2) transferring the process state to the destination machine, and
3) resuming its execution on the destination machine.

These operations require support from the operating system. The initial implementation of *Enterprise* has been done under the Sun Operating System, which does not provide any support for process migration. However, the development of *Enterprise* under an operating system which offers process migration, such as the V-System (Cheriton and Zwaenepoel, 1983), would allow for better load distributions. Under Sun OS, when it is necessary to relinquish a processor, it is sometimes possible to suspend the process in the middle of execution and restart it over again on another machine (although, obviously, some work will have been wasted). Also, in the case when the workstation is being used as part of a contract, it can be released when its unit of work is complete.

6. Towards A Test Suite of Parallel and Distributed Programs

It is common to demonstrate the flexibility and performance of a parallel programming environment by implementing selected programs in the system. Although a custom test suite is a practical way to highlight the strengths of an environment, the suite may be inappropriate for testing a different environment. Differences in performance criteria, design goals and the level of abstraction of alternate systems may make it pointless to simply adopt a previous program suite. Consequently, researchers tend to reinvent test suites in order to, understandably, focus on the advantages of their own system.

Performance evaluation, including the design of test suites, is an area of computing science that generates much debate and discussion. Benchmarks and test suites can be abused. There is the familiar adage that the best test is the problem that one personally wants to solve. Still, there are benchmarks and test suites that have proven to be particularly useful, despite their

acknowledged limitations. One well-known example is the LINPACK numerical libraries (Dongarra et al., 1976). This work has led to *de facto* methods of measuring the floating point performance of traditional supercomputers. Arguably, the success of LINPACK benchmarks is due to the importance of numerical algorithms in the "real world" applications being solved on supercomputers. In effect, the benchmarks are more meaningful because they are relevant to the user community. Therefore, we adopt that proven approach in the design of our own test suite.

An important aspect of our approach is that we want the test suite to guide the development of *Enterprise*, and not vice versa. Because we want a test suite that is not specific to any particular parallel architecture, including *Enterprise*'s, we select our programs from a wide range of scientific and engineering applications. We have not limited our selections to problems with large-grained parallelism and minimal communication needs. Surely, those applications would emphasize the strengths of *Enterprise*. But because we also wish to learn from the limitations of *Enterprise*, our test suite is designed to define the envelope of the system's capabilities. By constructing our test suite at a relatively early stage of *Enterprise*'s design, we hope that the lessons learned will benefit our design effort as much as it will benefit the potential users of the system.

Clearly, the test suite outlined below is preliminary and subject to change. We recognize that the suite will evolve over time, but we feel that it is important to address the issue at this time.

6.1 Related Work in Test Suites

As we have already pointed out, there are many test suites to choose from. But because these program suites tend to be closely tied to a particular system, they cannot easily be used as a general test suite. Some researchers have also recognized the need for a common test suite for evaluating different parallel systems.

The efforts of Singh, Weber and Gupta (1991) to develop the SPLASH (Stanford Parallel Applications for Shared-Memory) test suite are notable. SPLASH currently consists of six applications and related input files. The programs are all over 1000 lines of source code and go beyond the "toy" applications common in other work. From Singh, Weber and Gupta's (1991) Tables 1 and 2, we learn that the *Ocean* application simulates eddy currents in an ocean basin by a grid-based method which includes the solution of a set of partial differential equations. The application *Water* simulates the behavior of water molecules by an N-body method. *MP3D* is an application that simulates the rarefied hypersonic flow of, say, air around an airfoil in the upper atmosphere by particle-in-cell methods. *LocusRoute* uses an iterative refinement technique to route wires as part of the computer-aided design of VLSI circuits. *PTHOR* is a digital logic circuit simulation application based on distributed time discrete event simulation methods. Also, the application *Cholesky* factorizes a sparse matrix by the Cholesky method.

Although SPLASH is architecture dependent (i.e. shared-memory multiprocessing systems), it represents a significant effort in the area. Not only does it attempt to abstract the evaluation of parallel systems to a higher-level memory model, it also sets a standard of openness and thoroughness in its presentation. Equally important, the developers of SPLASH have drawn their applications from a wide scientific and engineering domain spectrum to increase the relevance of their test suite to the high performance computer user. Furthermore, the authors describe the behavior, implemented performance and simulated performance of each application in detail. The C or FORTRAN source code for each application is freely available. Although the *Enterprise* effort is currently not as progressed, we hope it will evolve to at least the level of SPLASH. Many of the SPLASH applications have been incorporated into our own test suite for the time being.

6.2 Road Map for the Enterprise Test Suite

The primary design goal of the test suite is to accurately represent the problems that are currently being solved on high performance computers. From experience, a clear pattern emerges regarding the demands being placed on supercomputers. Firstly, it is a well-known fact that numerical and floating-point intensive algorithms are among the most common applications on high performance computers. Secondly, computational simulation is rapidly becoming a popular third paradigm of science. Thirdly, there are problems in discrete mathematics and combinatorics with large solution spaces that can utilize as much computational power as is available. Lastly, there are computationally intensive problems in computing science. Each category represents a real world application of high performance computers, and each presents an unique set of challenges for a parallel programming environment.

Table 1 summarizes the algorithms and problems selected for the test suite. A more detailed description appears below.

Category	Application
numerically intensive applications	systems of equations (*Cholesky* from SPLASH) matrix manipulation (multiplication, eigenvalue, inversion) Fast Fourier Transform
simulation	animation of articulated bodies N-body simulation (*Water* from SPLASH) particle-in-a-cell (*MP3D* from SPLASH) circuit simulation (*PTHOR* from SPLASH)
discrete mathematics	convex hull travelling salesman alpha-beta search retrograde analysis
computing science	ray tracing sorting makefiles

Table 1. Test suite applications.

6.2.1 Numerically Intensive Applications

The fascination that supercomputer users and manufacturers have with floating point performance is a symptom of the numerically intensive problems that scientists have traditionally wanted to solve on computers. Solving a system of equations, various matrix manipulation operations and function transformations such as the Fast Fourier Transform (FFT) are typical of the types of algorithms used by scientists. Therefore, we include a representative selection of algorithms and problems from this area. Except for the *Cholesky* application from the SPLASH suite, all of the programs are relatively short.

6.2.2 Simulation

The versatility of computers has made the simulation of phenomena based on a mature body of theory practical for scientific research. Along with theory and physical experimentation, the simulation of complex systems and interactions by computers is becoming a valuable scientific tool. Although the types of simulations are as varied as the types of phenomena and the theories that describe them, there are some standard paradigms by which the simulations are designed. Aside from the numerical algorithms used to solve the mathematical models being simulated, the

techniques of animation, cellular automata and discrete state transformations are common in simulation. Therefore, the problems of animating articulated bodies, particle-in-the-cell and digital circuit simulation have been included in the test suite. Because computer simulations are complex applications, we have borrowed heavily from the SPLASH suite in this category

6.2.3 Discrete Mathematics

There are many problems in discrete mathematics that have large solution spaces. The problems may require a large amount of computation due to a large amount of data, a theoretical NP-complete or NP-hard complexity, or a large proof-tree requirement. Typically, the complexity of the problems grow exponentially and a large amount of additional computational power must be added to the solution in order to solve a marginally larger problem. Still, the solution may be of such fundamental importance that any decrease in solution time or increase in the solvable problem size is of great value. Therefore, the convex hull, travelling salesperson, alpha-beta search and retrograde analysis problems have been added to the test suite.

6.2.4 Computing Science

Finally, there are problems that are mainly of interest to computing scientists. Ray tracing and sorting are of fundamental interest in two areas of computing science. They are inherently highly parallel and are popular choices for parallel programming test suites. A parallel *makefile* is also a popular choice for test suites, so all three problems are included in our proposed test suite.

6.3 Future Work

Once again, it should be emphasized that the proposed test suite is preliminary. However, because we feel that a practical approach to evaluating the usefulness of *Enterprise* is important, we have laid out our plans at this relatively early stage of the system's development. Instead of using *Enterprise* to guide the development of the test suite, we are using the type of applications that we feel should be in the test suite to guide the development of *Enterprise*. We hope the end result will be a development environment that will be useful to the high performance computing community.

7. Related Work

In the world of sequential programming, we usually program on the conceptual machine (using a high-level language) and seldom concern ourselves with the physical architecture of the machine. However, in the world of parallel programming conceptual machines are not as common. The parallelism in a program is usually expressed in an architecturally specific way, forcing the programmer to consider issues such as communication and synchronization at a low

level. With the rapid advancement in programming environments, tools to: edit, test, debug, and visualize parallel programs on abstract computation models are now available. Programming at the conceptual level increases the portability and reusability of programs. It also allows the parallelism in a program to be expressed in a natural way, implying that the programmer need only be knowledgeable about the application, not about parallel programming. The formulation of parallel computation can involve the following steps (Carriero and Gelernter, 1989; Browne, 1985, 1986):

1) Choose the model of computation that is most natural for the problem.

2) Write a program using the method that is most natural for that abstract machine.

3) If the resulting program is not acceptably efficient, transform it methodically into a more efficient version by switching from a more-natural method to a more-efficient one.

It is widely believed that the abstract computation model provided is the heart of a programming environment. This section gives an overview of recent developments in parallel programming environments, classifying them by their conceptual computation model and comparing them on their strengths and weaknesses in expressing parallelism. Two related studies offer additional insight. Bal, Steiner and Tanenbaum (1989) focus on the language issue in distributed systems. Chang and Smith (1990) examined the features of parallel programming tools, but the notion of the conceptual model of computation is not addressed. The Linda language (Gelernter et al., 1985; Gelernter, 1989; Carriero and Gelernter, 1988a, 1988b) uses the concept of a *tuple space* for communication between concurrent processes. Processes use atomic operations to read, add or delete a tuple to/from the tuple space. This model is powerful but it does not provide a high-level parallel conceptual machine to describe a parallel program. This review therefore does not include Linda.

In Appendix C, a review of the 14 tools referenced in this section is presented. These tools are representative of the current spectrum of parallel programming environments.

7.1 Styles of Parallel Computation

There are many ways to classify parallel computation (Bal, Steiner and Tanenbaum, 1989; Carriero and Gelernter, 1989; Chang and Smith, 1990; King, Chou and Ni, 1990). Here we describe the ones that we believe best summarizes the different approaches to achieve parallelism. King, Chou and Ni (1990) characterize parallel computation from two perspectives: from the way the computation is partitioned and distributed, and from the way the computation is executed. From the first perspective, computations are characterized as *data-parallel* or *function-parallel:*

1) A function-parallel view has a program divided into subprograms of different functionality, which can be executed in parallel on different processors. Function-parallelism is suitable for applications that can be programmed using many independent subroutines (animation, for example).

2) A data-parallel view has the data partitioned among the processors. Processors may execute the same program but work on different data subsets. Data-parallelism is suitable for applications which perform the same set of operations repeatedly and independently on a large set of data. Programs with nested loops to handle static and regular data structures are suitable for data-parallel computation (image processing and ray tracing, for example).

From the second perspective, execution, a parallel computation can be characterized as either concurrent or pipelined (Hwang and Briggs, 1984):

1) Concurrency exploits spatial parallelism by utilizing different processors executing multiple independent tasks simultaneously. Tasks may be data-parallel or function-parallel.

2) Pipelining exploits temporal parallelism. Each processor behaves like a filter or transformer operating on its input data and passing output data to the succeeding processors. Data flows through the pipeline as it is processed stage by stage. A datum, once entering the system, will be used or modified repeatedly along the pipeline.

By combining these two perspectives together, four different styles of parallel computation can be identified as shown in Table 2.

Execution	Partition	
	Function	Data
Concurrent	Concurrent function-parallel	Concurrent data-parallel
Pipelined	Pipelined function-parallel	Pipelined data-parallel

Table 2: Different styles of parallel computation (King, Chou and Ni, 1990).

7.1.1 Concurrent Function-Parallel Computation

In this approach, different processors perform different functions simultaneously. The strategy is to break the original problem down into several small independent subproblems. Each of these subproblems can be assigned to a different processor and the results received from these processors can be combined to form a solution to the original problem. For example, in a particle-in-a-cell application, different boundary constraint tests can be applied in parallel to the same datum.

7.1.2 Concurrent Data-Parallel Computation

Processors perform the same operation simultaneously but on different data sets in the concurrent data-parallel approach. This is also known as domain decomposition and is similar to the agenda parallelism described by Carriero and Gelernter (1989). All the processors work in parallel on the same item on the agenda and then move on the next item on the agenda. Sometimes, the decomposition of the work can proceed dynamically and partitions can be of uneven sizes, as in problems that use divide and conquer and state space search strategies.

7.1.3 Pipelined Function-Parallel Computation

In the pipeline different stages perform different functions. When data flows through the stages, they are modified along the way. The fish animation application discussed in this report illustrates an application suitable for a pipeline.

7.1.4 Pipelined Data-Parallel Computation

The data-parallel approach has a number of processors performing identical operations on individual data elements. To carry out correct executions, different data streams have to flow along different directions to bring appropriate data to the required processors at the right moment. The operations are synchronized. Since the communication overhead associated with medium and large-grained multi-computers are considerable, and a global clock is required to synchronize the operation among the processors, this approach of parallelism is uncommon in MIMD and distributed parallel programming environments. It is most often seen in fine-grained granularity applications (such as systolic systems (Kung, 1979)). Carriero and Gelernter (1989) do not even consider this approach, however, research has been underway to explore its feasibility (King, Chou and Ni, 1990) and some programming environments support it (Segall and Rudolph, 1985).

7.2 Models of Computation

The models of computation can be described as a directed graph whose nodes represent units of computation, and whose arcs represent the data flow, control flow or messages between nodes (Browne, 1985). The models can be classified into three categories according to the structural

focus employed to incorporate the parallelism (Pancake and Utter, 1989): data oriented, task oriented and template attachment as shown in Figure 9.

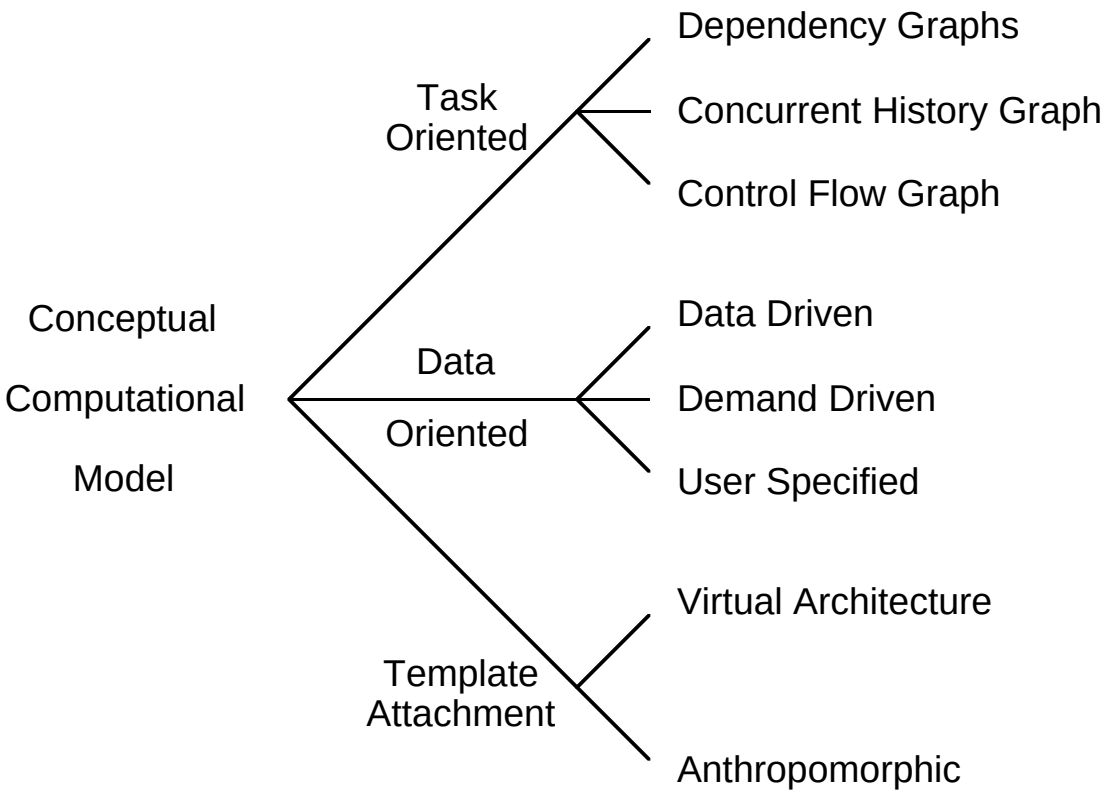

Figure 9: A classification scheme for parallel programming environment models.

7.2.1 Task Oriented Models

In these models, parallelization is achieved by partitioning the tasks to be performed into a collection of serial processes, each processing a unique thread of control. Models in this category can be represented by a directed graph whose nodes represent a procedure call or a synchronization point, and whose arcs represent operations that advance the program from one state to another. Models that fall in this category include those based on task dependencies graphs (PFG, E/SP and HeNCE) and control flow graphs (CAPER and PAT).

7.2.2 Data Oriented Models

For data oriented models, parallelism occurs as the simultaneous execution of an operation on multiple data elements. Whenever the size of the data warrants parallelization, activities are replicated for execution across data subsets. Program execution is viewed as a single thread of control which temporally diverges into parallel action sequences that later converge. Models in this category can be represented by a directed graph whose nodes represent units of computation and

whose arcs represent flow of data (Babb, 1984). Models can be further classified as being demand-driven, data-driven or user-specified.

In the demand-driven model of computation, the sink node requests data from the source node and the source node responds. The sink node is the terminating node producing the final output. The source node is the initial node in the program which is also responsible for getting the input data. From the sink node, each node first sends a *demand for data* request to its parent in the dependency graph. The demand is then propagated toward the input of the graph. From the source node, the result is then propagated back to the sink node. A data-driven methodology employs a communication protocol where source nodes send results to sink nodes and perhaps await acknowledgement of result arrival or use time-out. In both cases, the functions are embedded in the nodes and data objects are carried on the arcs. The number of messages in the demand-driven model is double that of the data-driven model and the size of the demand messages are small, resulting in a poor utilization of bandwidth. Some models (CODE, for example) integrate demand-driven and data-driven control by allowing some arcs to be data-driven and others to be demand-driven.

Environments that use the dataflow model or its variant includes CODE, DGL, LGDF, TDFL, PPSE, and Paralex.

7.2.3 Template Attachment Models

In these models, entities do not uniformly represent subsets of activity or data. The model is an orderly system of interaction among independent, self-contained entities. An entity may contain data, operations or both. The program is a system of black boxes emitting and receiving messages. Some environments model the computation entities after physical architecture characteristics and provide architectural templates like systolic arrays and master/slave relationships (PIE and PAL, for example). Some environments model the four styles of parallelism directly by providing templates for concurrency and pipeline (*FrameWorks* and *Enterprise*, for example).

7.2.4 Explicit / Implicit Parallelism

Explicit parallelism in an application is accomplished by using a procedural language with constructs for concurrency to specify the application. The programmer is responsible for the interaction between parallel tasks. Implicit parallelism is accomplished by a non-procedural language model. An application is described declaratively and viewed as a directed graph whose vertices denote functions and edges denote dependencies between functions with which they are incident. NMP (Marsland, Breitkreutz and Sutphen, 1991) and PVM (Sunderam) are examples of this type of parallelism.

7.3 Hierarchy

The data dependency graph contains redundant information for the parallelization process. Many dependencies are generated by scalar variable references, which offers little possibility of parallelism. Also, it is rare than every statement is involved in a compute-intensive segment. It is wasteful to store data dependence information for a region that is not compute intensive, since its parallelization achieves little. A model that supports hierarchies of abstraction permits the computation to be realized with increased resolution and limits the size of the graph to a manageable complexity. The highest level graph consists of a single node which contains the entire program. A node of the most resolved level contains single statements.

All three categories of computation models can support hierarchical abstractions on the computation graphs. In data-oriented and task-oriented models, the graphs are strictly hierarchical since expanding a node gives a sub-graph with similar properties as the original node. In template attachment models, graphs at different levels may exhibit different properties.

7.4 Mutability

Mutability refers to whether the size of a task can change dynamically at run time (Suhler et al., 1990). If mutability is supported, the program can allocate the computation and memory resources as required to process the current data. Otherwise, the programmer has to encapsulate all the data set into a single, static computation structure. In most parallel programming environments, mutability is explicit: the programmer uses commands embedded in the program to create and remove units of computation. It directly involves the programmer in resource management and can result in inefficiencies. On the other hand, implicit mutability makes it difficult for the programmer to use knowledge about what granularity might be appropriate for a given program.

Models that use *fork* and *join* or similar constructs are explicitly mutable, so task-oriented models are generally explicitly mutable. In the data-oriented environment, parallelism is implicit. With the availability of dependency information, a unit of computation can be partitioned into several units, or combined with other units of computation. Therefore, a data-oriented model can be mutable (TDFL, for example). Some task-oriented models, such as E/SP, support the automatic detection of parallelism, in which case implicit mutability is possible. In the template attachment model, the programmer decides the parallel template to use and the granularity of work, so mutability is explicit. However, to some extent, implicit mutability is supported in environments that allow dynamic allocation of worker processes to complete the work (contracts in *Enterprise* and contractors in *FrameWorks* are examples).

7.5 Expressive Power

Nodes in the template attachment models, by definition, may exhibit different properties. In general, the model can express any parallelism desired by designing new templates. The other two models, task and data oriented, are more limited because they usually support a single node type - the computation unit. Both these models work best for expressing concurrent function-parallel computation. It is possible to express concurrent data-parallel parallelism only when the data size and number of processors are static. Besides, to express a simple concurrent data-parallel computation, in which each processor takes an equal share of the tasks and works in parallel, the programmer has to partition the data and specify the routing of data for each node. Also, most models do not support recursive calls and dynamic concurrent data-parallel computation. Finally, it is impossible to express pipeline computations in these models. Therefore, some environments introduce special nodes to express the above parallelisms. In TDFL, the *DoAll* node is added to the dataflow model, which takes an incoming array data token and outputs an array data token. The size of the array determines the number of times the function is invoked. Each invocation writes to a different element of the array. Also, the *Self-Loop Arc* is added to support state retention.

7.6 Comments

In some cases, the dataflow diagram can be automatically translated from the dependency specification in the file (*makefiles* are a good example). Non-procedural languages, such as Lisp or Prolog, are better described by a data-oriented model. However, usually, dataflow diagrams are not readily available.

Because of the extensive research in the area of automatic parallel compilers, the program call structure and control flow information can be automatically generated in most cases. The programmer usually need only be concerned with the unresolved dependencies. However, the parallelism obtained by an automatic parallelizer may not be optimal since the analysis is based on static information only.

When the information is not available, it is time consuming and error prone for the programmer to draw a large number of nodes and edges when the data flow or call structure are complicated. Moreover, data dependence, program control flow and program states are all low-level information and can often be non-intuitive to the programmer. A tool that interacts with a programmer only in terms of the above information can easily overwhelm the programmer with a large amount of low-level information. Also, these graphs can be dense structures. For large programs, this may pose manageability problems to the programmer.

If the concern is to design a parallel program with the programmer in control, the best choice would be a parallel programming environment that used the template attachment model. If mutability and implicit parallelism are the concern, then a data-oriented environment is the choice. Task-oriented parallelism is best used as a tool to let the programmer view and modify the program after automatic parallel structuring has been performed.

8. Project Status

Enterprise is being implemented. The current status (August, 1991) is:

1) The Graphical Interface Manager is functional, but without all the user-friendly features fully working. A user can edit a diagram and its attributes, as well as save and load it. No effort has been devoted to developing the Text Interface Manager.

2) The Code Librarian manages the source and object correctly for a heterogeneous network of machines. Currently, the insertion of *Enterprise* code into a user's code is done manually; the program to do this automatically is under development.

3) The Execution Manager supports lines, pools, departments, contracts and individuals. Other assets are under construction.

4) No effort has been made to implement the Monitor/Debugger Manager.

5) Currently, the Resource Secretary assumes all machines are idle.

We expect to have a usable system by the end of 1991, with all the features implemented by the end of 1992.

In recent years, there has been an enormous increase in the number and quality of parallel programming tools described in the literature. The authors of these tools have diverse opinions as to where in the software development cycle and how these tools can increase a programmer's productivity. The *Enterprise* project aims for a complete, integrated programming environment that is suitable for the complete software development life cycle. By capturing an application's parallelism through the use of diagrams that are simple to edit, it is not difficult for the user to make the leap from sequential to parallel programming. Although the complexity of parallel systems, as portrayed in the literature, has been a powerful deterrent to growth in this area, we believe that with a simple model, all of the complexity of parallel programming can be hidden from the user. The analogical model used in *Enterprise* represents a different way of viewing an old problem.

Acknowledgements

This research was supported in part by research grants from the Central Research Fund, University of Alberta, and the Natural Sciences and Engineering Research Council of Canada, grants OGP-8173 and infrastructure grant 107880. Also, Rasit Eskicioglu provided us with a number of useful references.

References

B. Appelbe and K. Smith. PAT: Interactive Conversion of Sequential to Parallel Fortran, *IEEE COMPCON*, 1990, pp. 585-588.

O. Babaoğlu, L. Alvisi, A. Amoroso and R. Davoli. Paralex: An Environment for Parallel Programming in Distributed Systems, 1991, Technical Report UB-LCS-91-01, Department of Mathematics, University of Bologna, Bologna, Italy.

R.G. Babb. Parallel Processing with Large Grain Data Flow Techniques, *IEEE Computer*, 1984, vol. 17, no. 7, pp. 55-61.

H.E. Bal, J.G. Steiner and A.S. Tanenbaum. Programming Languages for Distributed Computing Systems, *ACM Computing Surveys*, 1989, vol. 2, no. 3, pp. 261-322.

V.A. Balasundaram. Mechanism for Keeping Useful Internal Information in Parallel Programming Tools: The Data Access Descriptor, *Journal of Parallel and Distributed Computing,* 1990, vol. 9, no. 2, pp. 154-170.

A. Beguelin, J.J. Dongarra, G.A. Geist, R. Manchek and V.S. Sunderam. The PVM and HeNCE Projects, electronic news group comp.parallel, 1990.

K. Birman, R.Cooper and B.Gleeson. Programming with Process Groups: Group and Multicast Semantics, 1991, Technical Report TR-91-1185, Computer Science Department, Cornell University.

K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck and M. Wood. The ISIS System Manual, Version 2.1, 1991, ISIS Project, Computer Science Department, Cornell University.

K.S. Booth, J. Schaeffer and W.M. Gentleman, Anthropomorphic Programming, 1984, Technical Report CS-82-47, Department. of Computer Science, University of Waterloo.

J.C. Browne. Formulation and Programming of Parallel Computations: A Unified Approach, in *Proceedings of the International Conference on Parallel Processing*, 1985, pp. 624-631.

J.C. Browne. Framework for Formulation and Analysis of Parallel Computation Structures, *Parallel Computing,* 1986, vol. 3, pp. 1-9.

J.C. Browne, T. Lee and J. Werth. Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment, *IEEE Transactions on Software Engineering*, 1990, vol. 16, no. 2, pp. 111-120.

J.C. Browne, K. Sridharan, J, Kiall, C. Denton and W. Eventoff. Parallel Structuring of Real Time Simulation Programs, *IEEE COMPCON*, 1990, pp. 580-584.

J.C. Browne, J. Werth and T. Lee. Intersection of Parallel Structuring and Reuse of Software Components: A Calculus of Composition of Components for Parallel Programs, in *Proceedings of the International Conference on Parallel Processing*, 1989, pp. 126-130.

N. Carriero and D. Gelernter. Applications Experience with Linda, in *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1988, ACM, New York.

N. Carriero and D. Gelernter. Linda in Context, *Communications of the ACM*, 1988, vol. 32, no. 4, pp. 444-458.

N. Carriero and D. Gelernter. How to Write Parallel Programs, *ACM Computing Surveys*, 1989, vol. 2, no. 3, pp. 323-357.

L. Chang and B.T. Smith. Classification and Evaluation of Parallel Programming Tools, 1990, Technical Report 1990-22, College of Engineering, University of New Mexico.

A. Chatterjee. Futures: A Mechanism for Concurrency Among Objects, in *Proceedings of Supercomputing '89*, 1989, pp. 562-567.

D.R. Cheriton and W. Zwaenepoel. The Distributed V Kernel and Its Performance for Diskless Workstations, in *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, 1983, pp. 129-140.

E.E. Dijkstra, *Selected Writings on Computing: A Personal Perspective*, 1982, Springer-Verlag, New York.

D.C. DiNucci and R.G. Babb II. Architecture of the Parallel Programming Support Environment, *IEEE COMPCON*, 1989, pp. 102-107.

J.J. Dongarra, J. Bunch, C. Moler and G. Stewart. LINPACK Users' Guide, 1976, SIAM Publishers, Philadelphia.

M.R. Eskicioglu. Design Issues of Process Migration Facilities in Distributed Systems, *IEEE TCOS Newsletter*, 1990, vol. 4, no. 2, pp. 3-13.

G.A. Geist and V.S. Sunderam. Network Based Concurrent Computing on the PVM System, 1991, Report Number TM-11760, Oak Ridge National Laboratory.

D. Gelernter, N. Carriero, S. Chandran and S. Chang. Parallel Programming in Linda, in *Proceedings of the International Conference on Parallel Processing*, 1985, pp. 255-263.

D. Gelernter. Information Management in Linda, in *Parallel Processing and Artificial Intelligence*, 1989, John Wiley, pp. 23-34.

W. Harrison. Tools for Multiple-CPU Environments, *IEEE Software*, 1990, vol. 7, May, pp. 45-51.

K. Hwang and F. Briggs. *Computer Architecture and Parallel Processing*, 1984, McGraw-Hill, New York.

R. Jagannathan, A.R. Downing, W.T. Zaumen and R.K.S. Lee. Dataflow-based Methodology for Coarse-Grain Multiprocessing on a Network of Workstations, in *Proceedings of the International Conference on Parallel Processing*, 1989, pp. 54-58.

A. Jones and A. Schwarz, Experience Using Multiprocessor Systems - A Status Report, *Computing Surveys*, 1980, vol. 12, no, 3, pp. 121-166.

C.T. King, W.H. Chou and L.M. Ni. Pipelined Data-Parallel Algorithms: Part I-Concept and Modeling, *IEEE Transactions on Parallel and Distributed Systems*, 1990, vol. 1, no. 4, pp. 470-485.

H.T. Kung. Lets Design Algorithms for VLSI Systems, in *Proceedings of the Caltech Conference on Very Large Scale Integration*, 1979, pp. 65-90.

T.G. Lewis and W.G. Rudd. Architecture of the Parallel Programming Support Environment, *IEEE COMPCON*, 1990, pp. 589-594.

M.A. Linton, J.M. Vlissides, and P.R. Calder. Composing User Interfaces with InterViews, *IEEE Computer*, 1989, vol. 22, no. 2, pp. 8-22.

T.A. Marsland, T. Breitkreutz and S. Sutphen. A Network Multi-processor for Experiments in Parallelism, *Concurrency: Practice and Experience*, 1991, vol. 3, in press.

C.M. Pancake and S. Utter. Models for Visualization in Parallel Debuggers, in *Supercomputing '89*, 1989, pp. 627-636.

R.W. Scheifler and J. Gettys. The X Window System, *ACM Transactions on Graphics*, 1986, vol. 5, pp. 79-109.

Z. Segall and L. Rudolph. Pie (A Programming and Instrumentation Environment for Parallel Processing), *IEEE Software*, 1985, vol. 2, no. 6, pp. 22-37.

A. Singh. A Template-Based Approach to Structuring Distributed Algorithms Using a Network of Workstations, 1991, Ph.D. Thesis, Department of Computing Science, University of Alberta.

A. Singh, J. Schaeffer and M. Green. Structuring Distributed Algorithms in a Workstation Environment: The FrameWorks Approach, in *Proceedings of the International Conference on Parallel Processing*, 1989, pp. 89-97.

A. Singh, J. Schaeffer and M. Green. A Template-Based Tool for Building Applications in a Multicomputer Network Environment, in *Parallel Computing*, 1989, D. Evans, G. Joubert and F. Peters (editors), North-Holland, pp. 461-466.

A. Singh, J. Schaeffer and M. Green. A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstations, *IEEE Transactions on Parallel and Distributed Systems,* 1991, vol. 2, no. 1, pp. 52-67.

J.P. Singh, W-D. Weber and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory, 1991, Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University.

K. Smith, B. Appelbe and K. Stirewalt. Incremental Dependence Analysis for Interactive Parallelization, in *Supercomputing '90* , 1990, pp. 330-341.

K. Smith and B. Appelbe. PAT-An Interactive Fortran Parallelizing Assistant Tool, in *Proceedings of the International Conference on Parallel Processing,* 1988, pp. 58-62.

S. Sobek, M. Azam and J.C. Browne. Architecture and Language Independent Parallel Programming: A Feasibility Demonstration, in *Proceedings of the International Conference on Parallel Processing,* 1988, pp. 80-83.

K. Sridharan, M. McShea, C. Denton, B. Eventoff, J. Browne, P. Newton, M. Ellis, D. Grossbard, T. Wise and D. Clemmer. An Environment for Parallel Structuring of Fortran Programs, in *Proceedings of the International Conference on Parallel Processing,* 1989, pp. 98-106.

K. Sridharan, R. Narayanaswamy, C. Denton and B. Eventoff. Parallel Structuring of Programs Containing I/O Statements, in *Proceedings of the International Conference on Parallel Processing,* 1990, pp. 224-228.

P.D. Stotts. The PFG Language: Visual Programming for Concurrent Computation, in *Proceedings of the International Conference on Parallel Processing,* 1988, vol. 2, pp. 72-79.

B. Sugla, J. Edmark and B. Robinson. An Introduction to the CAPER Application Programming Environment, in *Proceedings of the International Conference on Parallel Processing,* 1989, pp. 107-111.

P.A. Suhler. Heuristic Tuning of Parallel Loop Performance, in *Proceedings of the International Conference on Parallel Processing,* 1989, pp. 184-191.

P.A. Suhler, J. Biswas and K.M. Korner. TDFL - A Task-Level Data Flow Language, 1987, Technical Report TR-87-44, Department of Computer Science, University of Texas at Austin.

V. Sunderam. PVM: A Framework for Distributed Computing, *Concurrency: Practice & Experience,* 1990, vol. 2, no. 4.

D. Vrsalovic, Z. Segall, D. Siewlorek, F. Gregorettl, E. Caplan, C. Fineman, S. Kravltz, T. Lehr and M. Russinovich. Performance Efficient Parallel Programming in MPC, technical report CMU-CS-88-164, Department of Computer Science, Carnegie Mellon University, 1988.

Z. Xu and K. Hwang. Molecule: A Language Construct for Layered Development of Parallel Programs, *IEEE Transactions on Software Engineering,* 1989, vol. 15, no. 5, pp. 587-599.

Appendix A.

C Code for the Entry Procedures of the Animation Example

This appendix gives pseudo code for the Animation example. For brevity, only the main procedure calls of Model, PolyConv and Split are shown.

Asset Code: Model

```
/* Model asset */

#define  NUMBER_STEPS    4
#define  NUMBER_FISH    10
#define  NUMBER_FRAMES  20

Model()
{
   float timeperframe;
   int frame;

   /* Generate the school of fish */
   MakeFish( NUMBER_FISH, 0 );

   /* Loop through each frame */
   timeperframe = 1.0 / NUMBER_STEPS;
   for( frame = 0; frame < NUMBER_FRAMES; frame++ )
   {
      /* Do model computations */
      InitModel( NUMBER_FISH );
      MoveFish( NUMBER_FISH, timeperframe );
      DrawFish( NUMBER_FISH, timeperframe * frame );
      WriteModel( frame );

      /* Done! Send work to PolyConv process */
      PolyConv( frame );
   }
}
```

Asset Code: PolyConv

```
/* PolyConv asset */

#define MAX_POLYGONS 1000

PolyConv( frame )
int frame;
{
   polygon polygontable[ MAX_POLYGONS ];
   int npoly;

   /* Convert polygons and send to Split */
   DoConversion( frame );
   npoly = ComputePolygons( &polygontable );
   Split( frame, npoly, polygontable );
}
```

Asset Code: Split

```
/* Split asset */

#define  MAX_POLYGONS   1000

Split( frame, npoly, polygontable )
int frame, npoly;
polygon polygontable;
{
   HiddenSurface( frame, npoly, &work.polygontable );
   AntiAlias( frame, npoly, &work.polygontable );
}
```

Appendix B.

C Code for the Entry Procedures of the Animation Example

The example application in Appendix A, along with the *Enterprise* diagram in Figure 6, is translated automatically into an executable program. In this Appendix, the program that *Enterprise* produces is given. In the code, bold text refers to inserted code, while regular text identifies the user-written code. At the time of this writing, the ISIS implementation has not yet been finalized. This example should be used as an illustration of what *Enterprise* does to the user code, without the code necessarily being completely accurate.

Asset Code: Model

```
/* Model asset */

#define  NUMBER_STEPS    4
#define  NUMBER_FISH    10
#define  NUMBER_FRAMES  20

/* ISIS code for a module in a line.  Since this is the start */
/* of the computation, this routine is not called by another  */
/* module.   However it does call the next module in the line. */

#include  "isis.h"          /* Header file for ISIS libraries   */
#define  DONE_ENTRY  2      /* Entry id of routine handle exit. */

#define  port_nr  1612      /* ISIS port number */
address  *PolyConv_pg;      /* Addr of process serving the call */

/* Note, the main routine for a module without a caller (no   */
/* input parameter) is different from one with a caller (with */
/* a parameter(s)).                                           */
main()
{
   int  Model();

   isis_init(  port_nr  );
   isis_task(  Model,  "Model"  );
   isis_mainloop(  Model  );
}

Model()
{
   float timeperframe;
   int frame;

   /* Generate the school of fish */
   MakeFish( NUMBER_FISH, 0 );

   /* Loop through each frame */
   timeperframe = 1.0 / NUMBER_STEPS;
```

```
   for( frame = 0; frame < NUMBER_FRAMES; frame++ )
   {
      /* Do model computations */
      InitModel( NUMBER_FISH );
      MoveFish( NUMBER_FISH, timeperframe );
      DrawFish( NUMBER_FISH, timeperframe*frame );
      WriteModel( frame );

      /* Done! Send work to PolyConv process */

      /* ISIS message format is similar to that of the funct- */
      /* ion fprintf. Here, an integer argument is copied in- */
      /* to the message.  The number following the argument   */
      /* specifies the number of replies wanted. In this exam-
   */
      /* ple, all calls are p-calls and no reply is required. */
      bcast( PolyConv_pg, CALL_ENTRY, "%d",frame, 0 );
   }
   /* ISIS process will remain waiting to serve a message un- */
   /* less it is explicitly terminated by calling exit().     */
   /* A module needs to tell all called modules to exit too.  */
   bcast( PolyConv_pg, DONE, "", 0 );
   exit(0 );
}
```

Asset Code: PolyConv

```
/* PolyConv asset */

#define MAX_POLYGONS 1000

/* ISIS code for a module in a line. It is called by a module */
/* and also calls a module which is in a contract.            */
#include "isis.h"

#define port_nr 1612        /* ISIS port number               */
#define CALL_ENTRY 1
#define DONE_ENTRY 2

main()
{
   int PolyConv_main(), PolyConv(), term();

   isis_init( port_nr );

   /* Define frame format for user defined structure.  The    */
   /* Enterprise parser provides from its symbol table an     */
   /* unused format character for the new format.  Here it is */
   /* 'i', for example. It then records the assignment of     */
   /* the character 'i' to the type polygon. The last         */
   /* argument is a function to preserve data types across    */
   /* different architectures. On machines with same byte     */
   /* ordering, this can be just NULL or 0.                   */
   isis_define_type( 'i', sizeof( struct polygon ), 0);

   isis_task( PolyConv_main, "PolyConv_main" );

   /* An ISIS entry is a process which ISIS will start up in  */
```

```
   /* response to a message delivered to it. An entry is      */
   /* identified by the process address (PolyConv) and its    */
   /* entry number (CALL_ENTRY).   A process address can have  */
   /* more than one entry as shown below.                     */
   isis_entry( CALL_ENTRY, PolyConv, "PolyConv" );
   isis_entry( DONE_ENTRY, term, "term" );

   isis_mainloop( PolyConv_main );
}

address  *PolyConv_pg;
address  *Split_pg;
address  *Dispatcher_pg;
groupview  *Split_gv;          /* Calling module of a contract  */
                               /* or pool include these to talk */
                               /* to specific worker in pool.   */

PolyConv_main()
{
   PolyConv_pg = pg_join( "PolyConv_pg", 0 );
   Split_pg = pg_lookup( "Split" );
   Dispatcher_pg = pg_lookup( "Dispatcher_pg" );
   Split_gv = pg_getview(Split_pg);

   /* isis_start_done tells the ISIS system that the start-up */
   /* sequence is completed and it is ready to handle incoming*/
   /* message events. It is not really necessary here. ISIS   */
   /* automatically invokes isis_start_done when the main task*/
   /* terminates.                                             */
   isis_start_done();
}

/* This entry when called, call exit to terminate this module */
term( msg_p )
message  *msg_p;
{
   msg_get( msg_p, "");
   exit(0);
}

PolyConv( msg_p )
message  *msg_p;
{
   int npoly;
   polygon polygontable;

   /* The procedure header and the input parameters are con-  */
   /* verted to a call to get the parameters from an ISIS msg */
   int frame;
   msg_get( msg_p, "%d", &frame );

   /* Convert polygons and send to Split */
   DoConversion( frame );
   npoly = ComputePolygons( polygontable );

   /* Obtain an idle Split member from bulter service */
   fbcast( Bulter_pg, CHECK_OUT, "" ,1 ,"%d" ,&idle_member );
```

```
   /* The format items have an array version, obtained by re- */
   /* placing the lower case letter in the item by the corre- */
   /* sponding upper case letter.  Note that the array format */
   /* item %I corresponds to two arguments: the address of the*/
   /* first element to copy (polygontable) and the number of  */
   /* elements to copy (npoly).                               */
   fbcast( Split_gv->gv_members[idle_member], CALL_ENTRY, "%d%I",
                frame, polygontable, npoly, 0 );

}
```

Asset Code: Split

```
/* Split asset */

#define  MAX_POLYGONS   1000

/* ISIS code for a called module in a contract. It does not   */
/* call any module.                                           */

#include "isis.h"

#define port_nr 1612        /* ISIS port number               */
#define CALL_ENTRY 1
#define DONE_ENTRY 2

main()
{
   int Split_main(), Split();

   isis_init( port_nr );

   isis_define_type( 'i', sizeof( struct polygon ), 0 );
   isis_task( Split_main, "Split_main" );
   isis_entry( CALL_ENTRY, Split, "Split" );
   isis_entry( DONE_ENTRY, term, "term" );
   isis_mainloop( Split_main );
}

address *Split_pg;
address *Dispatcher_pg;

Split_main()
{
   Split_pg = pg_join( "Split_pg", 0 );
   Dispatcher_pg = pg_lookup( "Dispatcher_pg" );

   /* pg_rank returns the id of this process, which is an     */
   /* integer between 0 and the number of Split processes.    */
   /* This broadcast tells Dispatcher that it is free.        */
   fbcast( Dispatcher_pg, CHECK_IN, "%d",
                pg_rank( Split_pg,&my_address ), 0 );
   isis_start_done();
}

term( msg_p )
message *msg_p;
{
```

```
   msg_get(  msg_p,  "");
   exit(0);
}

Split(  msg_p  )
message  *msg_p;
{
   /*  After  this  call,  the  polygontable  in  the  message  will   */
   /*  be  stored  in  the  first  npoly  elements  of  the  array,  and */
   /*  npoly  will  have  the  number  of  elements  stored.           */
   int frame, npoly;
   polygon polygontable;
   msg_get(  msg_p,  "%d%I",  &frame,  polygontable,  &npoly  );

   HiddenSurface( frame, npoly, polygontable );
   AntiAlias( frame, npoly, polygontable );

   /*  This  processes  is  free  to  serve  next  call.    */
   fbcast(  Dispatcher_pg,  CHECK_IN,  "%d",
            pg_rank(  Split_pg,&my_address  ),  0  );
}
```

The Dispatcher service maintains a list of currently available contract processes that can serve a call. A CHECK_OUT request results in a process being removed from the list and its id returned to the caller. A CHECK_IN request arises when a contract process is finished and its id is added to the list. The CHECK_OUT and CHECK_IN entries are shown below:

```
condition  idle_wait  =  (condition)  0;      /*  Sleep,  waiting    */
                                              /*  for  an  event.    */

check_out(  msg_p  )
message  *msg_p;
{
   int  idle;

   if(  (  idle  =  pop(  idle_stack  )  ==  NONE  )
   {
      /*  if  none  available,  wait  */
      idle  =  t_wait(  &idle_wait  );
   }
   reply(  msg_p,  "%d",  &idle  );
}

check_in(  msg_p  )
message  *msg_p;
{
   int  idle;

   msg_get(  msg_p,  "%d",  &idle  );
   if(  t_waiting(  &idle_wait  )  )/*  if  someone  waiting,  wake    */
   {                               /*  it  up  and  send  it  the  num-  */
      t_sig(&idle_wait,idle);      /*  ber  of  the  newly  available  */
   }                               /*  available  process.            */
   else
      push(idle,  idle_stack);
}
```

Appendix C.

Survey of Other Parallel Programming Environments

In this appendix, a brief description of the environments mentioned in Section 7 is given. The focus is on the conceptual models they use to represent computation.

C.1 Data Oriented Models

CODE

Computation Oriented Display Environment (CODE) uses a large-grain dataflow model. Computations at each node can be specified as data- or demand-driven. The programmer describes the computation by a graph of nodes and dependencies. The graph can be hierarchically defined; a node contains a sub-graph of nodes, a filter, or a subprogram. Nodes that contain programs are called schedulable units of computation (SUC). SUCs communicate via dependencies. A filter node defines the transmission of some subset of data from its input dependencies to some subset of its output dependencies.

Comments: dataflow, user specify an arc as demand- or data-driven, implicit parallelism, no restructure support, hierarchical resolution of parallelism, software component reusable, cannot express pipeline parallelism.

References: Suhler, Biswas, Korner and Browne, 1990; Browne, Lee and Werth, 1990.

DGL

Direct Graph Language (DGL) is a parallel programming environment based on demand-driven, large-grain dataflow model. A vertex is associated with a function module. Each incoming edge is associated with a parameter taken by the function. Each outgoing edge is associated with a use of the result of the function. A function module is a function: it returns a single result and has no side effects.

Comments: dataflow, demand-driven, implicit parallelism, no restructure support, single level resolution of parallelism, explicitly mutable, cannot express data and pipeline parallelism.

References: Jagannathan et al., 1989.

LGDF

Large Grain Dataflow (LGDF) takes the form of a directed graph consisting of processes (represented as circles) and datapaths (represented as vertical lines or rectangles) selectively connected pair-wise by arcs. A process is a sequential program written in a high-level language such as C or Fortran. Each datapath contains a (possibly zero length) data structure specified by the programmer. For a process to read and/or write to this structure, the process and datapath must

be connected with an arc having read and/or write permission. These permissions are represented in the graph as arrowheads on the arc illustrating the allowed direction(s) of dataflow: toward the process (for read permission) and/or toward the datapath (for write permission). At any moment, a datapath can be in the state of being read from, written to or neither. A process may start execution (fire) when it can read from all of its connected arcs. Also, by applying the *reserve* and *grant* commands to a datapath, a process can exclude or permit other processes from accessing the datapath.

Comments: dataflow, data-driven, implicit parallelism, no restructure support, single level resolution of parallelism.

References: DiNucci and Babb, 1989.

Paralex

Paralex is a parallel programming environment based on a data-driven, large-grain dataflow model. There are 2 types of nodes. At a computation node, execution begins when all links incident at a node contain a value. The computation satisfies the functional paradigm. Data communications are specified by drawing links to connect the nodes. Filter nodes allow data values to be extracted on a per destination basis before they are transmitted to the next node, to avoid the transmission of unnecessary data.

Comments: dataflow, data-driven, implicit parallelism, no restructure support, single level resolution of parallelism, explicitly mutable, fault tolerant, cannot express data and pipeline parallelism.

References: Babaoglu, Alvisi, Amoroso and Davoli, 1991.

PPSE

Parallel Programming Support Environment (PPSE) is an integrated set of tools for the design and construction of parallel software. The PPSE Parallax graphical design editor supports a hierarchical graphical description language called ELGDF (Extended Large Grain Dataflow) to describe data and control dependencies between tasks. It offers graphical constructs such as pipes, loops, repeated nodes, and special dataflow arcs to indicate mutually exclusive access to data.

Comments: dataflow, implicit parallelism, no restructure support, hierarchical resolution of parallelism.

References: Lewis and Rudd, 1990.

TDFL

Task-Level Dataflow Language (TDFL) is evolved from CODE. It is the first corse-grain dataflow language that supports the dynamic modification of graphs (implicit mutability). As

described in Section 7, it is difficult to describe loop and do-all parallelism using the dataflow model. TDFL introduces new constructs to support parallelism not addressed in the formal dataflow models: *Loop*, *DoAll*, *Case* and *EndCase* nodes, and self-loop arcs.

Comments: dataflow, user specify an arc as demand- or data-driven, implicit parallelism, no restructure support, hierarchical resolution of parallelism, implicitly mutable, software component reusable, cannot express pipeline parallelism.

References: Sobek, Azam and Browne, 1988; Browne, Werth and Lee, 1989; Suhler, Biswas and Korner, 1987; Suhler, 1989.

C.2 Task Oriented Models

CAPER

CAPER is an application programming environment for message passing multi-processors. A program is specified by drawing a graph of nodes and lines. A node contains routines for algorithm or routines or transforming data between communicating algorithms (they take care of the distributing and restructuring of distributed data). A line represents the transfer of data between two routines. Each node is also assigned a number which specifies how many processors are used to run the encapsulated routine. The graph is hierarchical; nodes can be grouped together or expanded.

Comments: control flow graph, explicit parallelism, hierarchical resolution of parallelism, explicitly mutable, cannot express data and pipeline parallelism.

References: Sugla, Edmark and Robinson, 1989.

E/SP

E/SP is an environment for the parallel structuring of Fortran programs using a hierarchical dependency graph. The highest level graph consists of a single node plus an entry and exit node. A node of the most resolved level contains single statements. The next level of resolution is the basic block or the subprogram, and the highest level is typically a segment of a call tree. The Fortran language has been extended to include *fork* and *join* statements for expressing parallelism. The programmer collapses or expands a node to decide the granularity of a computation unit and the realization of the parallelism is done automatically by the tool. When dependences must be removed to achieve parallelism, the programmer is informed of the dependency type and prompted to change the portion of the program involved. The drawback is that only the divide and conquer paradigm of parallel processing can be realized.

Comments: hierarchical dependence graph, explicit parallelism, automatic structuring of parallelism, hierarchical resolution of parallelism, explicitly mutable, cannot express data and pipeline parallelism

References: Harrison, 1990; Sridharan, et al., 1989, 1990; Browne, Sridharan et. al., 1990.

HeNCE

The programming environment for a heterogeneous network of parallel machines, HeNCE, supports the creation, compilation, execution, debugging, and analysis of parallel programs for a heterogeneous group of computers. The programmer specifies the parallelism of a computation by drawing a graph describing the dependencies between user-defined procedures. HeNCE will then automatically execute these procedures over a user defined collection of machines on some network.

Comments: dependence graph, explicit parallelism, explicitly mutable.

References: Beguelin et. al., 1990; Geist and Sunderam 1991.

PAT

Parallelizing Assistant Tool (PAT) contains a parallelizer which examines a source Fortran program and suggest parallelization modifications, a static analyzer that simulates the execution of the source program and locates anomalies caused by the interaction of tasks and a debugger. The program analysis is built on using a control flow graph (CFG) of the program. Each node in the CFG represents either a basic block of the program, a branch or a merging of the program flows. Subroutine calls in the CFG are expanded in-line for simplicity (each call is expanded individually in context). The dependence information is extracted by tracing paths through the program, using reference lists to construct a global dependence graph. Browsing of dependences is provided through an interactive graphical interface at the statement or variable level.

Comments: control flow graph, explicit parallelism, static analysis to explore parallelism, hierarchical resolution of parallelism, explicitly mutable, cannot express data and pipeline parallelism.

References: Harrison, 1990; Smith and Appelbe, 1988; Smith, Appelbe and Stirewalt, 1990.

PFG

Program Flow Graph, PFG, has four nodes types: concurrency branch, non-deterministic branch, join and call. The nodes are connected with arrows. A call node contains a block of procedure call(s). A join node is the synchronization point of the incoming concurrent control paths, merging them into one. Call and join node may have no arcs or a single arc with no label leaving it. Each of the branch nodes contains a selector, and may have any number of arcs leaving

it. Execution proceeds from the initial node, and nodes are executed in the order they are encountered by following arcs. If a node contains a procedure call, the call is evaluated and the data state is altered. If a node contain a selector, then the selector is evaluated and a choice of the next node (or nodes) is made based on the result of evaluation. For a concurrency branch, two or more parallel threads are created. All selected arcs are concurrently followed. For a non-deterministic branch, one of the out-going arcs is chosen non-deterministically and followed.

Comments: explicit parallelism, no analysis to explore parallelism, single level resolution of parallelism, explicitly mutable, cannot express data and pipeline parallelism.

References: Stotts, 1988.

C.3 Template Attachment Models

FrameWorks

In this model, an application is viewed as a graph with nodes being communicating processes. Each node contains a sequential module or procedure. Communication and synchronization are specified by the properties of the node through up to 3 types of attribute bindings. The input template specifies the incoming message to a node, and can be one of the following: initial, which accepts no input from other nodes; in_pipeline, which specifies the node as a part of a pipeline; and assimilator, which states that the node merges the results of several nodes. The output template specifies the outgoing message from a node, and can be one of the following: out_pipeline, which specifies the output of the node to flow in a pipeline fashion to its connecting nodes; manager template, which specifies that a fixed number of multiple workers will be used to execute the called procedure; and terminal template, which specifies that the application terminates there. The body template specifies the execution mode of the node: an executive template causes the process to have its input, output and error streams directed to the terminal; and a contractor specifies that multiple workers are assigned dynamically at run-time to execute the specified node. The run-time dynamic task decomposition is supported by the language constructs *split* and *merge*, similar to *fork* and *join* in task-oriented models.

Comments: template attachment, 8 templates modelled after communication style, explicit parallelism, no automatic exploration of parallelism, single level resolution of parallelism, explicitly mutable, can express all style of parallelism except pipelined data-parallel computation.

References: Singh, Schaeffer and Green, 1989a, 1989b, 1991; Singh, 1991.

PAL

Parallel Language, PAL, is a procedural parallel language introducing the language construct, *molecule*. A *molecule* is a set of program objects that have some common properties. A *molecule* type characterizes a particular computation mode (SIMD, sequential, pipelining, array processing,

dataflow, multiprocessing, etc.). A user can characterize a particular mode by defining corresponding molecule types.

Comments: template attachment, system-supplied and user-defined templates modelled after a variety of architectures, explicit parallelism, no automatic exploration of parallelism, hierarchical resolution of parallelism, explicitly mutable, can express all styles of parallelism.

References: Xu and Hwang, 1989.

PIE

Parallel-programming and instrumentation environment, PIE, parallelism is realized by *join* and *detach* statements. Global data, and operations on that data, are encapsulated in *frames*. Frames are shared among specific tasks and/or C functions and shared abstracted data types can be constructed using frames. The most important idea in PIE is the implementation template construct, which defines and controls parallel computation activities and data. The pre-coded structure of an implementation (in terms of control, communication, synchronization, and data partition) is made available to the programmer in the form of a modifiable template. The template provides the send/receive operation and the programmer has to provide only the sequential code. Implementations can be nested with one implementation being a part of the other. The following implementations are available: master-slave, recursive master-slave (same as master-slave except that the slave could become recursively master for another set of activities), heap implementation (in which work is distributed through data structures like queues or heaps), pipeline and systolic multidimensional pipeline.

Comments: template attachment, 5 templates modelled after parallel and pipeline architectures, explicit parallelism, automatic exploration of parallelism, hierarchical resolution of parallelism, explicitly mutable, can express all style of parallelism.

References: Segall and Rudolph, 1985; Vrsalovic et al., 1988; Harrison, 1990.