University of Alberta

The Enterprise Model for Developing Distributed Applications

by

Greg Lobe
Paul Lu
Stan Melax
Ian Parsons
Jonathan Schaeffer
Carol Smith
Duane Szafron

Technical Report TR 92-20
November 1992

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

The Enterprise Model for Developing Distributed Applications

Greg Lobe
Paul Lu
Stan Melax
Ian Parsons
Jonathan Schaeffer
Carol Smith
Duane Szafron

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

{duane, jonathan}@cs.ualberta.ca

ABSTRACT

Workstations have been in use for more than a decade now. Although a network of workstations represents a large amount of aggregate computing power, there is a need for software that can harness this power for single, distributed applications. *Enterprise* is an interactive graphical programming environment for designing, coding, debugging, testing, monitoring, profiling and executing programs in a distributed hardware environment. *Enterprise* code looks like familiar sequential code because the parallelism is expressed graphically and independently of the code. The system automatically inserts the code necessary to handle communication and synchronization, allowing the rapid construction of correct distributed programs. The system supports load balancing, limited process migration, and dynamic distribution of work in environments with changing resource utilization. *Enterprise* utilizes the combined power of a cluster of workstations by providing a high-level programming model and environment that eliminates the perceived complexity in writing parallel software.

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

1. Introduction

Workstations 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"), the software advances necessary to exploit this power has lagged behind the hardware advances. In contrast to familiar sequential software, distributed software allows user applications to execute on many computers at once. Distributed software offers many advantages. Not only do programs potentially run faster, but a cluster of low-cost workstations can be a cost-effective approach to solving compute-intensive applications.

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 research has been done to develop efficient parallel algorithms for a large class of problems, finding a good parallel solution to a problem may only be a small fraction of the cost of implementing it. Writing distributed software is expensive because of problems 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; the execution environment is dynamic. Second, communication costs may be high in such an environment, restricting the types of parallelism that can be efficiently 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 system that is capable of producing parallel and distributed software quickly, economically and reliably. This system 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 and is independent of the code. The system automatically inserts the code necessary to handle communication and synchronization, allowing the rapid construction of correct distributed programs. This bridges the complexity gap between distributed and sequential software. 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:

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 decoupling of 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

consistent terminology based on *assets* (individuals, departments, receptionists, etc.). Organizations are inherently parallel and, often, have efficient parallelism. This analogy provides the programmer with 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) *Enterprise* supports the dynamic distribution of work in environments with changing resources. For example, assets can be replicated a variable number of times. During the life of a program, the amount of resources committed to it can vary depending on the resources available.

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 constructs correct diagrams in a top-down manner by re-classifying and expanding nodes. Re-classification 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.

A more detailed discussion of other parallel programming tools and environments and how they compare to *Enterprise* can be found in (Chan et al., 1991).

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, assigns processes to processors and establishes the necessary connections.

The *Enterprise* model described in this document is an evolution of an earlier design (Chan et al., 1991; Szafron et al., 1992) and our successful *Frameworks* system (Singh, Schaeffer and Green, 1989a, 1989b, 1991; Singh, 1991). Compared to the earlier draft of the model, this version represents several important advances:

1) The asset collection has been simplified.

2) The semantics of divisions have been defined.

3) Contracts have been replaced by user-defined minimum and maximum limits on the number of replicated assets.

4) The graphical user interface has been fully defined, simplified and is functional.

5) The programming model has been expanded to include the passing of arrays and dynamic pointers, and the returning of values using side-effects.

Some of these changes were motivated by feedback generated by our small user community.

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. A brief description of the implementation is given in Section 5. Section 6 discusses some experience programming in *Enterprise*. Section 6 describes the current status of the project.

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 and pseudo-code. A more detailed description of the code can be found in Appendix A.

- The main procedure, *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 goes to the next frame.

```
Model()
{
   for each frame
   {
```

```
      /* compute location and motion of objects */
      PolyConv( 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.

```
PolyConv( frame )
{
   /* perform transformations and projections */
   Split( frame, polygons );
}
```

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

```
Split( frame, polygons )
{
   /* hidden surface removal and anti-aliasing */
}
```

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). 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*.

When *Enterprise* is started, the *Enterprise* window contains a single pane called the *Program Pane*. It contains the icon for a single *Program* asset that represents the new program. Associated with each asset is a context sensitive pop-up menu. For example, if the user selects *Name* from the asset menu of the *Program* asset and types the word *Animation* into the dialog box that appears, the program would be named *Animation* and appear as in Figure 1. Note that *Enterprise* uses the host windowing system. The figures in this report were generated on the Macintosh implementation of the *Enterprise* user interface and look slightly different in X-windows or Sun Open Windows.

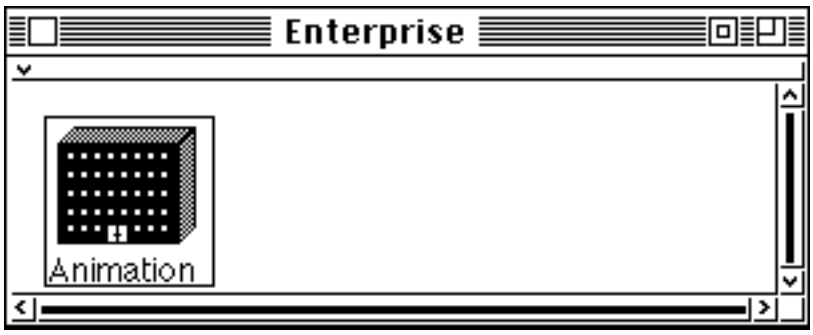


Figure 1: A new program called *Animation.*

If the user then selects *Expand* from the asset menu, the *Program* asset icon will expand to reveal the single *Individual* asset that it contains. If the user selects *Name* from the asset menu of the *Individual* asset and types the word *Model* into the dialog box that appears, then the program would appear as shown in Figure 2.

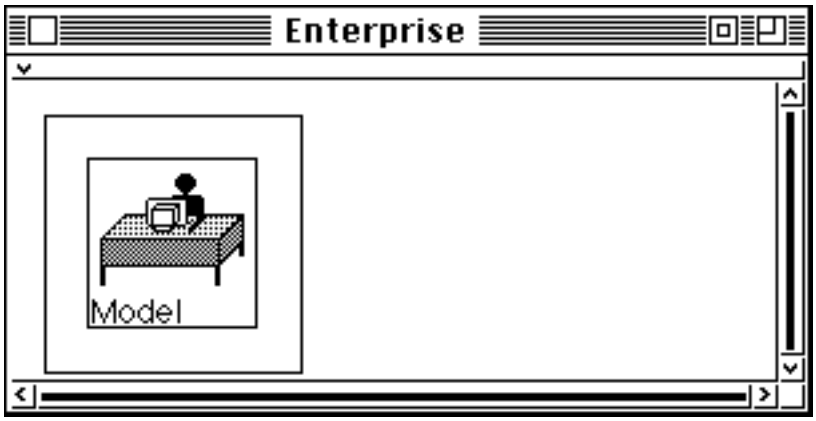


Figure 2: A sequential program that contains a single *Individual* asset called *Model.*

The user could enter all of the code for *Model*, *PolyConv* and *Split* into this single *Individual* asset and run the program sequentially. However, there is no reason why *Model* should wait until *PolyConv* completes the first animation frame to start processing the second frame. Similarly, *PolyConv* does not need to wait for *Split*. In other words, these three routines could act as a pipeline. Therefore, if the user selects *Line* from the asset menu of *Model*, it is re-classified as a *Line* asset. Before the re-classification, a dialog box asks the user for the number of subordinate assets that the *Line* should contain. In this case, the user should answer with 2, one each for *PolyConv* and *Split*. After the re-classification, the *Enterprise* window looks like Figure 3.

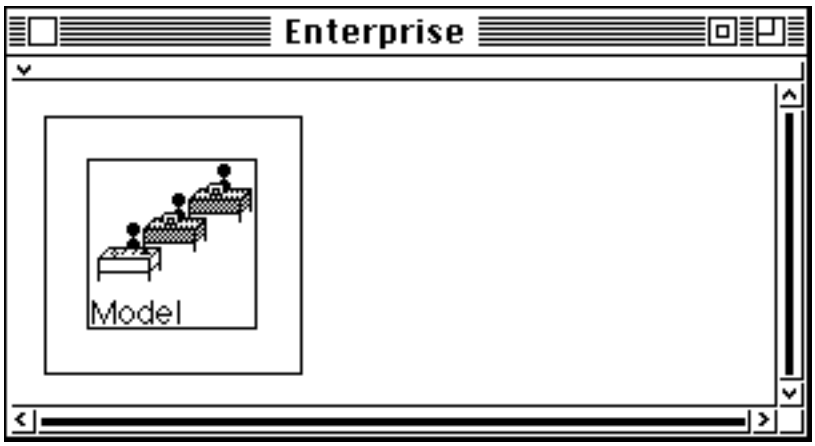


Figure 3: A program consisting of a *Line* asset called *Model.*

If the user selects *Expand* from the asset menu of *Model*, it is expanded so that the *Enterprise* window appears as shown in Figure 4.

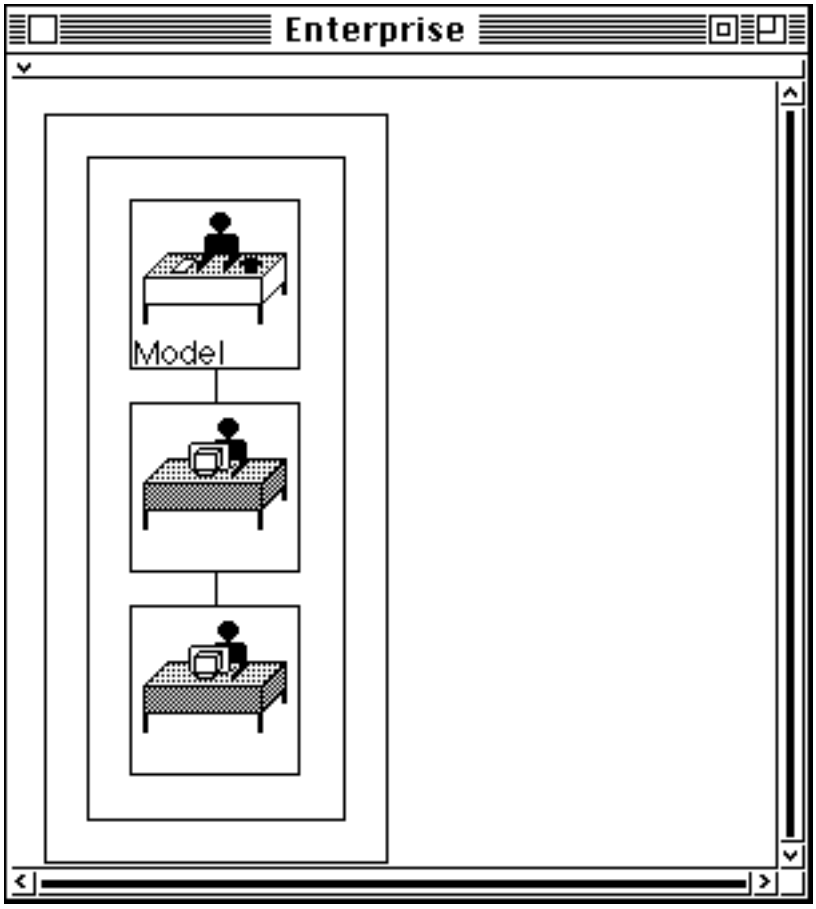


Figure 4: A program containing an expanded *Line* with two subordinates.

The outer rectangle represents the *Program*. The inner rectangle represents the *Line* and the three icons inside of the *Line* represent its components. The first component is a *Receptionist* asset that shares the name, *Model*, with the *Line* asset that contains it. The other two assets are subordinate *Individuals*.

The user can select *Name* from the menu of each *Individual* asset in turn and name them *PolyConv* and *Split*. The user can then select *Code* from the menu of each asset in turn and enter the their C-language source code into the text editor window that appears as shown in Figure 5.

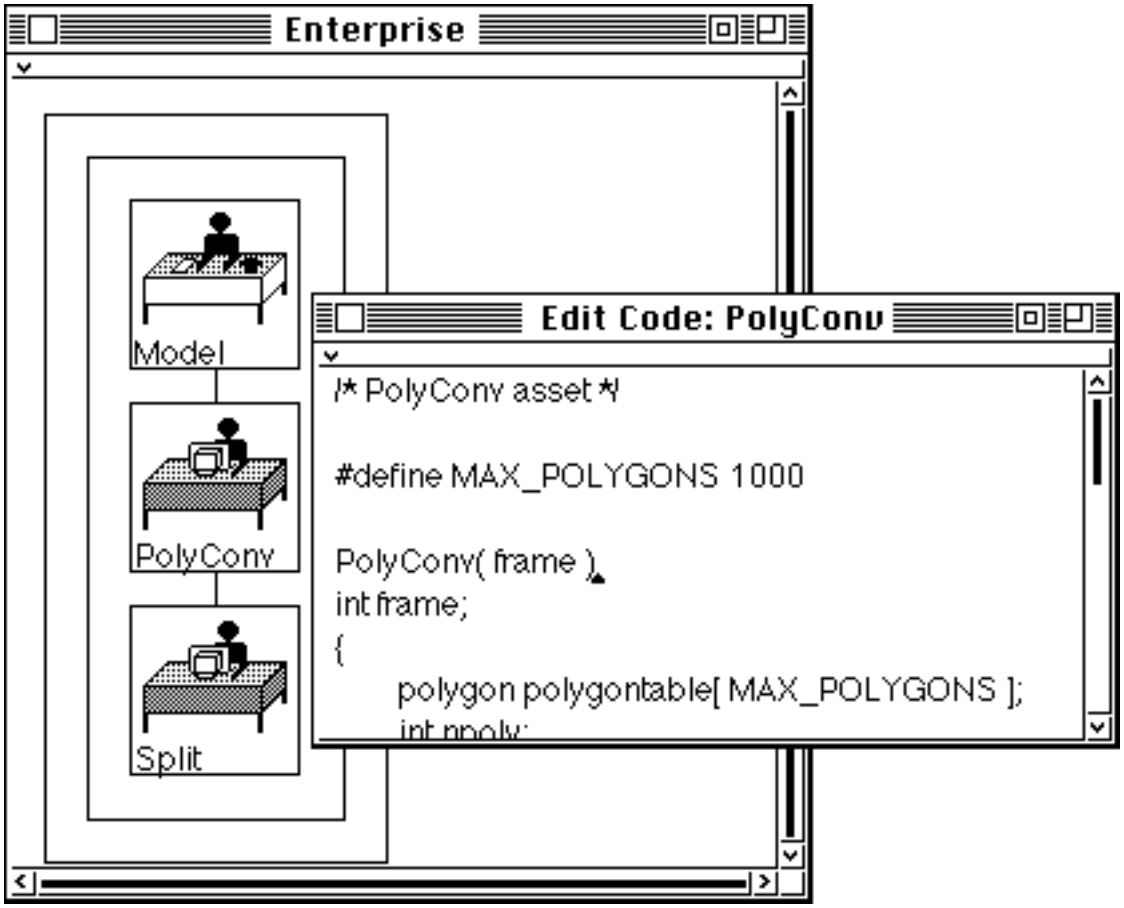


Figure 5: Editing the C-language code for *PolyConv*.

The code for the entry procedures *Model*, *PolyConv* and *Split* are shown in Appendix A. If the user selects *Compile* from the *Program* pane menu then *Enterprise* automatically inserts the code to handle the distributed computation, compiles the program and reports any errors in a window. Appendix B shows the code that *Enterprise* inserts into the application to handle the communication and synchronization (using the code in Appendix A with the diagram in Figure 5). Once the program is compiled, the user selects *Execute* from the *Program* pane menu and *Enterprise* finds as many processors as are necessary to start the program, initiates processes on

the processors and monitors the load on the machines. 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 all timings are subject to large variations, depending on the number of available processors and the amount of traffic on the network. The timings were done on a quiet network without other compute-bound jobs competing for resources.

One of the strengths of the *Enterprise* model is that it is easy to take a program and experiment with alternate parallelization techniques without changing the C-language source code. Each asset represents at least one process. If a call is made to the *individual Split*, it is executed by a process and if a subsequent call is made to *Split* before the first call is complete, the second call must wait for the first call to finish. However, if the *Split* asset is *replicated* then multiple processes can be used to execute multiple calls concurrently. For example, if the user selects *Replicate* from the asset menu of *Split*, and when a dialog box appears, enters 1 and 5 as minimum and maximum replication factors, then the Animation program appears as in Figure 6.

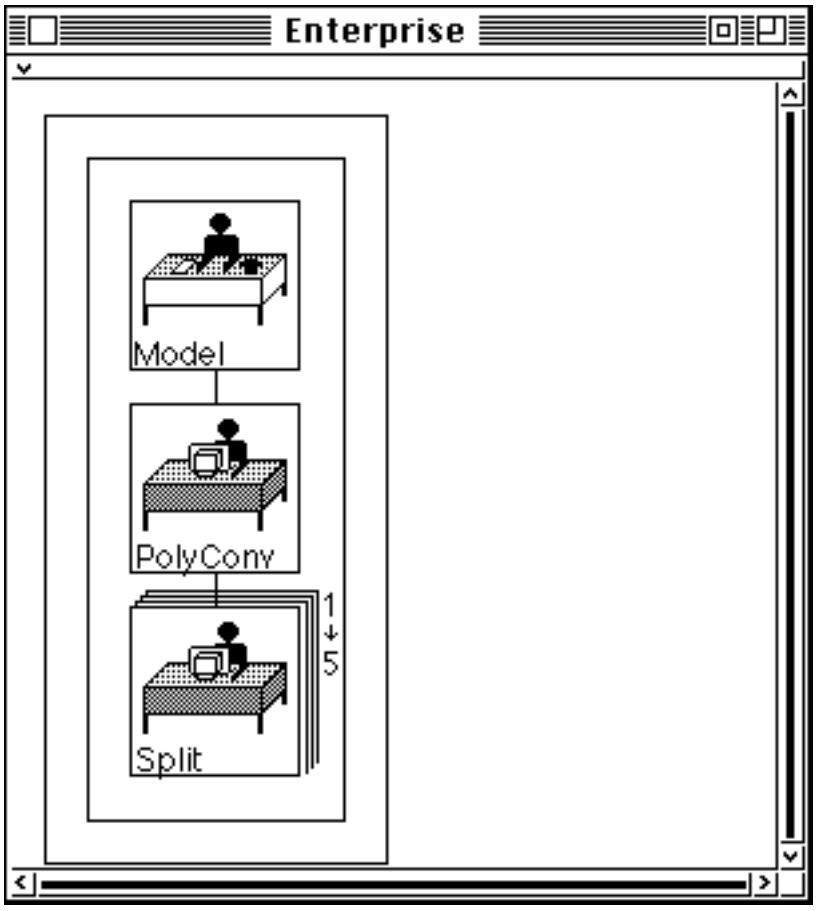


Figure 6: A replicated asset.

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). Replication can be dynamic in *Enterprise* so that as many processors as are available on the network may be used, subject to a lower and upper bound supplied by the user.

Replicating *Split* results in as much as a 5.7-fold speed-up compared to the sequential animation program, depending on when the program is run. Of course, there is no reason why the user cannot replicate *PolyConv* 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, if the C code for the *Split* contained a sequence of procedure calls, it could be re-classified as a *Line* where each of the assets in the *Line* represented one of the procedure calls. Figure 7 shows the result of this operation.

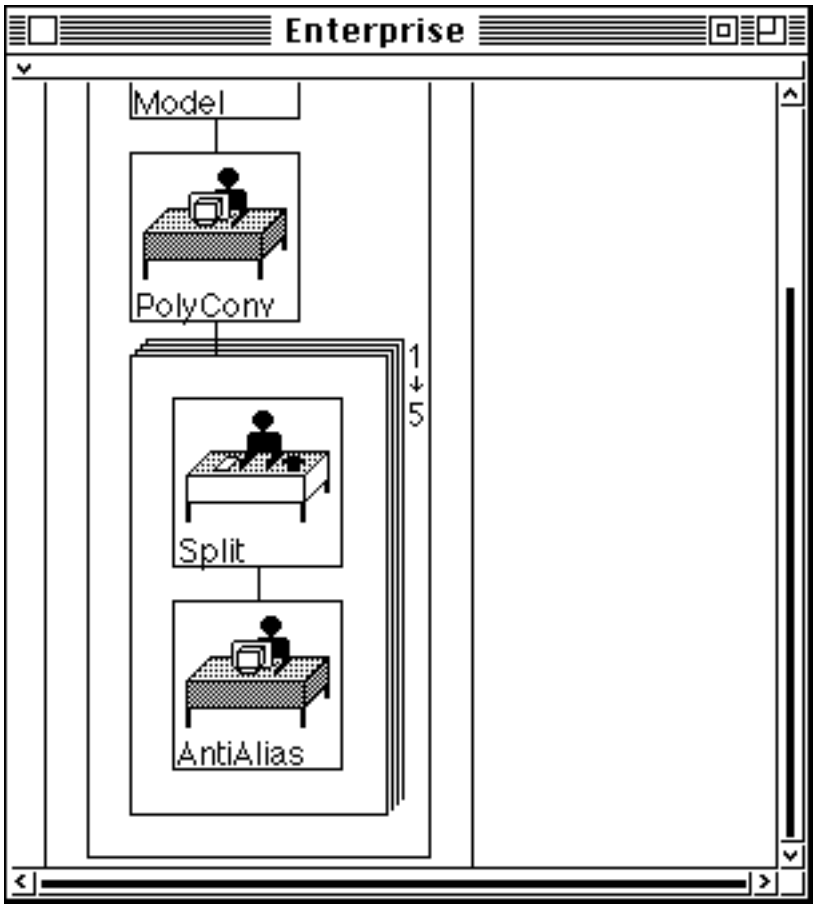


Figure 7: A line asset that contains a replicated line asset.

Note that the asset named *Model* has partially scrolled off the display. *Split* contained two procedure calls, *HiddenSurface* and *AntiAlias*. The former remains a local procedure call in *Split*, but the latter has been converted to an asset. Its code must be copied from the *Split* asset and pasted into the *AntiAlias* asset but need not be edited.

Several other asset kinds are supported by *Enterprise* and they can be combined in arbitrary hierarchies. The next section contains a description of all of 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 invoking *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). Conceptually, there is no difference between a sequential function call and an *Enterprise* module call except for the parallelism. When module *A* calls module *B*, *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.

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 arguments of *B* (`data`) would be packaged into a message and sent to *B*. *A* would continue executing in parallel with *B*. However, when the calling module, *A*, tries to access the result of the call to *B* (`value = result + 1`), it blocks until *B* has returned a message containing its `result`. This concept is similar to the work on futures in object-oriented programming (Chatterjee, 1989). If *B* is defined as a function, but used as a procedure (i.e. the return value is not used), the result is thrown away. We call this deferred synchronization a *lazy synchronous* call.

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*. We call this form of parallelism *purely asynchronous*.

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.

Enterprise modules can accept a variable number of parameters of varying types. Arrays and pointers are valid as parameters but they must be immediately followed by an additional size parameter that specifies the number of elements to be passed (unnecessary in sequential C). Unfortunately, this restriction is needed because it is not always possible to statically determine the size of the array to be passed. This feature allows users to pass dynamic storage as well as parts of arrays. The data being passed cannot itself contain pointers (which would be meaningless because of the distributed memory).

Enterprise-defines three macros for parameter passing. The IN() macro specifies that a pointer should have its values sent from the caller *A* to the callee *B*, but not returned. The macro OUT() specifies a parameter with no initial value, but one that gets set by *B* and returned to *A*. The macro INOUT() copies the parameter from *A* to *B* on the call, and copies its value back from *B* to *A* on the return. Note that it is the caller's responsibility to allocate storage for all returned results. For example, the code

```
int data[100], result;
. . .
result = B( &data[60], INOUT(10) );
/* some other code */
value = result + 1;
```

would send the elements 60..69 of `data` to *B*. When *B* is finished execution, it will copy back to *A* 10 elements, over-writing locations 60..69 of `data.` If IN, INOUT, or OUT is not specified, IN is assumed.

OUT and INOUT data implement parameter side-effects. Thus, in some sense, they can also be considered part of the return value of the function. In the above example, if *A* accessed `data[65]` before it accessed `result`, it would have to block until *B* returned, just as it would for the `result` of the call. Consequently, only p-calls without OUT and INOUT parameters are purely asynchronous; all other p-calls and all f-calls are lazy synchronous.

Although lazy synchronous calls provide greater opportunities for concurrency than synchronous calls, it is often useful possible to gain more concurrency by further relaxing the synchronization of f-calls (and p-calls that have side-effects). This can be done in those situations where the order in which results are returned by an asset is irrelevant. For example, consider the code:

```
int data[100], result[100], sum;
. . .
for( i = 0; i < 100; i++ )
{
   result[i] = B( data[i] );
}
sum = 0;
for( i = 0; i < 100; i++ )
{
   sum = sum + result[i];
}
```

Assume that ordinary lazy synchronous calls are used. When the statement in the second loop is executed, the asset may have to block and wait for result[0]. However, it may be the case that other results have been returned, say result[1] and result[4]. Since the value of sum is independent of the order of summation of the results in the second loop, more concurrency could be obtained

by using result[1] or result[4] in place of result[0] and using result[0] later in place of another result.

To increase concurrency, *Enterprise* allows the programmer to specify whether an asset is *ordered* or *unordered*. If an asset is unordered, then the return values of the asset are consumed in the order that the asset values are returned, independently of the order in which they are referenced. For example, if the *B* asset was unordered then the *reference* to result[0] in the second loop would refer to the *first* value returned by *B*, even though it may not be result[0]. Eventually all results would be added and the value of sum would be the same as if the second loop had blocked, waiting for the results in order. A user can specify the order attribute (ordered or unordered) of an asset using the graphical user interface so it is independent of the asset code.

There are a few implementation points that may not be obvious from the above discussion:

1) When an *Enterprise* function returns, the results of all outstanding *Enterprise* calls will be ignored. For example, consider asset *A*() making several calls to asset *B*(). Perhaps one of the results returned by *B* means that *A* has now completed its task and it returns. If there are any calls that have been made to *B* that have not yet returned, they are flagged so that they are ignored when they return. Note that in the current *Enterprise* implementation, outstanding calls are not cancelled - they are allowed to run to completion.

2) Two lazy synchronous calls that modify the same memory locations are not allowed to be active at the same time. For example, the following code would be executed sequentially:

```
int data[100], result[10];
. . .
for( i = 0; i < 5; i++ )
{
   result[i] = B( &data[i*10], INOUT(20) );
}
```

Each iteration through the loop has the side-effect of modifying overlapping regions of data. Since the sequential semantics of this loop impose the ordering constraints that the second call to *B* would use the copy of data returned from the first call, *Enterprise* cannot execute the calls in parallel.

3) Note that the previous point implies that if a member of an array is returned by a function, or is an OUT/INOUT parameter, *Enterprise* must generate code for all subsequent references to that array to see if they are accessing a returned result. Consider the following example:

```
int data[100], result[10];
. . .
for( i = 0; i < 10; i += 2 )
```

```
      result[i] = B( &data[i*10], IN(10) );
. . .
a = result[j];
```

The reference to `result[j]` might refer to the result of any one of the calls to *B*, depending on the run-time value of `j`. *Enterprise* must keep track of all addresses that a call to an asset can modify.

4) *Enterprise* does not recognize aliasing. Consider the following code fragment:

```
int data[100], result, *r;
. . .
r = &result;
. . .
result = B( &data[50], IN(10) );
. . .
b = *r;
```

In this example, `*r` is an alias for `result`. When `*r` is referenced, the program should stop and wait for *B* to return. To do this properly would require checking all pointer references to see if they are the object of an *Enterprise* call (or an OUT/INOUT parameter). The cost of correctly handling this is too high, both at the implementation level and, more importantly, at the performance level.

5) Divisions are a combination of sequential and parallel recursive calls. In the following example, assume QuickSort is defined as a division asset (divisions are discussed in the next section). Divisions allow assets to call themselves recursively and are ideal for divide-and-conquer algorithms.

```
QuickSort( a, size )
int * a, size;
{
   int pivot;
   if( size >= threshold )
   {
      pivot = Partition( a, size );
      if( pivot > 0 )
      {
         QuickSort( &a[0], INOUT(pivot - 1) );
      }
      if( pivot < size )
      {
         QuickSort( &a[pivot+1], INOUT(size - pivot) );
      }
   }
   else  InsertionSort( a, size );
}
```

For this program, the division should be defined with a breadth of two and we will assume a user-defined depth of two. The first call to QuickSort will divide the list and send each part independently to two other processes. These processes have no division children, which means the recursive calls will be done sequentially. In a division, *Enterprise* inserts code that allows both the parallel and sequential recursive calls to be made. Thus the user can change the breadth and depth of the division and without changing the asset code.

Appendix B gives an example of the modifications that *Enterprise* makes to user code.

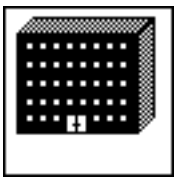
3.2 Module Roles and Assets

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 line 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 (individuals, departments, lines or divisions) to perform in parallel. 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 six different asset kinds: *enterprise, individual, department, line, division* and *service*. In addition, two other specialized individual assets are defined: *receptionist* and *representative.*

3.2.1 Enterprise



An *enterprise* is a single program. It is analogous to one organization or enterprise.

3.2.2 Individual

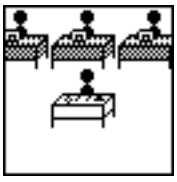


An *individual* contains no other assets. An individual is analogous to an individual person in an organization. For example, a clerk in a grocery store is an individual. When called, an individual executes its sequential code to completion. Therefore, any subsequent call to the same individual must 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. In general, an individual can be replaced by a line, department or division at any time. However, there are two special kinds of individuals. One is called a *receptionist* and the other is called a *representative*. Receptionists serve as the first element of any composite asset. They cannot be replaced by any other asset nor can they be replicated. Representatives can be replicated but they can only be replaced by divisions. Receptionists provide the name by which an asset can be called. The reasons for these restrictions will become apparent as the other asset kinds are described. Each individual (including receptionists and representatives) has code associated with it.

3.2.3 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 has finished its sub-task for the previous call, not until the entire line is finished. The first asset in a line is a *receptionist* that shares its name with the line. It 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. Lines are more often referred to as pipelines in the literature.

3.2.4 Department



A *department* contains a fixed number of heterogeneous assets. A single receptionist asset shares its name with the department so that it 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 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 and divisions. A department has no analogous term in the literature.

3.2.5 Division

A *division* contains a hierarchical collection of identical assets where work is divided and distributed at each level. They can be used to parallelize divide-and-conquer computations. When a division is created, it has a single receptionist asset that shares its name with the division so that it can be called by external assets. In addition it has a single representative asset that represents the recursive call made by the receptionist to the division itself. The user may change the breadth of the division's first level by replicating the representative. The user may add a level to the depth of the recursion by replacing the representative by a division. The new division contains a receptionist and a representative. The breadth of the tree at any level is determined by the replication factor of the representative or division it contains. This approach is capable of specifying arbitrary fan-out at each level of the division. For example, Figure 8 shows a typical divide-and-conquer process communication graph and Figure 9 shows the *Enterprise* division asset that achieves that effect. Divisions are the only recursive assets in *Enterprise*.

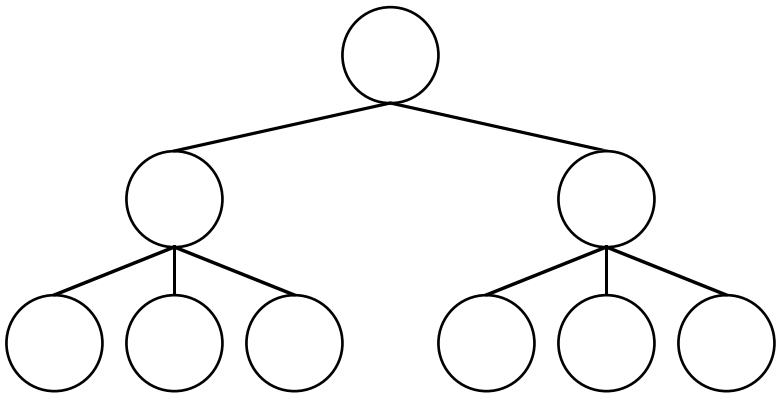


Figure 8: The run-time structure of a simple division asset.

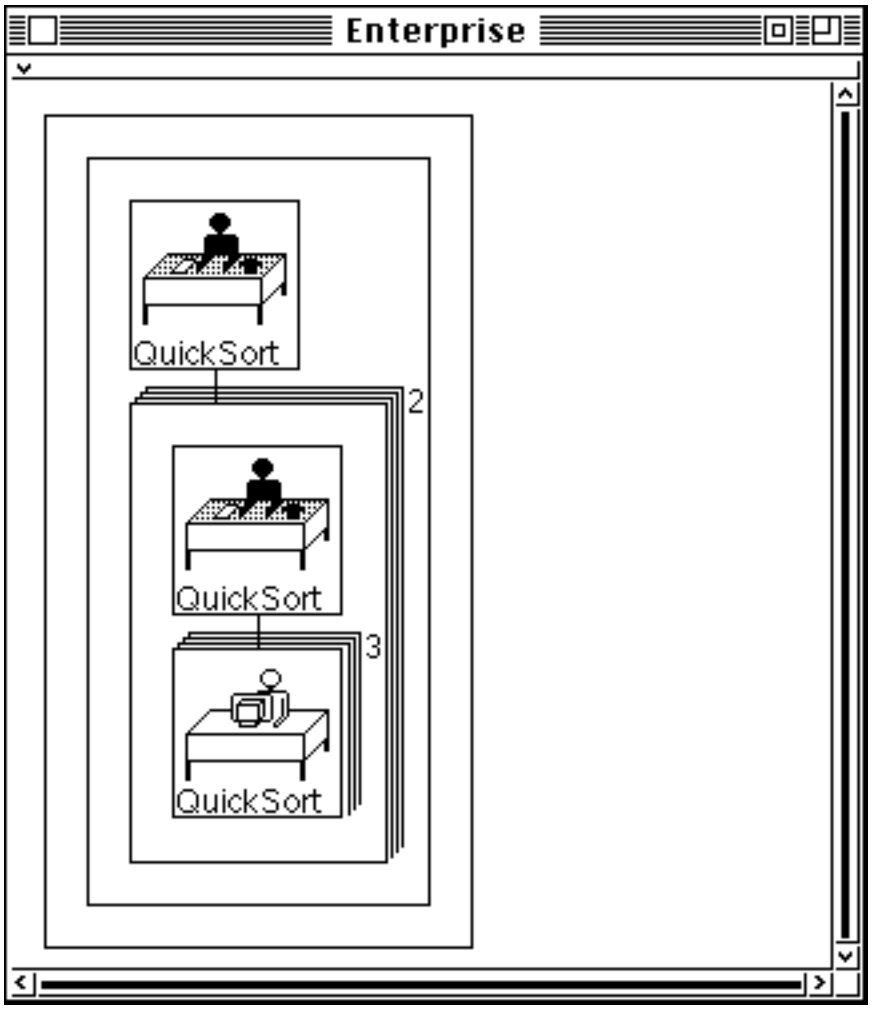


Figure 9: The representation of a simple division asset.

The QuickSort program is as described in the previous section. Note that although the user has the freedom to chose the fan-out at each level of the tree, there are cases where some fan-out may be wasted. For example, in the QuickSort program, only two recursive calls are made to QuickSort at each level, so the fan-out of 3 at the second level of Figure 8 is legal but not useful since the third processor will remain idle.

3.2.6 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 other asset using its name.

3.2.7 Replication and Order

Assets may be replicated so that when an *Enterprise* program is run, more than one process can simultaneously execute the code for the asset. The user specifies a minimum and maximum replication factor and *Enterprise* dynamically allocates as many processors as are available (one per process) up to the maximum. All assets except receptionists can be explicitly replicated. A receptionist cannot be explicitly replicated since it serves as the entry point for the asset. However, since an asset that contains a receptionist can be replicated, receptionists can be implicitly replicated.

Replicated assets may be ordered (default) or unordered. If an asset is unordered any reference to the return value of the asset will receive the first value returned from a copy of that asset. See section 3.1 for an example of the utility of unordered assets.

3.2.8 Hierarchical Assets

Enterprise assets can be built from any combination of assets. For example, one can construct a *department*, where one subordinate asset is itself a *line* of individuals and another is a *division*. The model allows the user to replace an asset by 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, in the animation program, the *line* of three individuals (*Model, PolyConv* and *Split*) each has its code in a separate file. If the *line* is re-classified as an *individual*, the code would have to be gathered together into the single *individual* asset's file. 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. In *Enterprise*, the application graph is an asset graph and it is constructed in a novel way. The user starts with an *enterprise* asset and constructs the graph by

replacing 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, lines, 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 replaced 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 (departments) and front to back (replication). In other words, the length of the graph represents the critical path of an application, and the width and depth reflect 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 one or two canvases. The *program canvas* is used to display and edit the graphical representation of the program. The *service canvas* (which can be displayed or hidden) contains the service assets that the program uses. Section 3 describes the asset kinds that are currently supported. When *Enterprise* is started a single *enterprise* asset is displayed on the program canvas.

Each asset has a context-sensitive menu. If the user presses the middle mouse button when the cursor is over an asset, then a pop-up menu appears containing all of the operations that are valid for that asset. If the button is pressed when the cursor is over the canvas itself then a pop-up menu appears containing all of the operations that can be performed on the program as a whole. The user does no drawing or moving of assets. Only the operations in the menus are required to modify the program graph.

The following operations can be performed on assets, although not all are valid for all assets. For example, only individual assets have code, receptionist assets may not be replicated and line assets may not be replaced by line assets.

1) **Name** or re-name an asset.

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

3) **Replicate** an asset by providing minimum and maximum replication factors and set the order attribute.

4) Replace an asset by a **Department** asset.

5) Replace an asset by a **Division** asset.

6) Replace an asset by an **Individual** asset.

7) Replace an asset by a **Line** asset.

8) Set the compilation and execution **Options** for an asset.

9) **Compile** an asset.

10) **Expand** an asset so its component assets are displayed.

11) **Collapse** an asset so that its component assets are hidden.

12) **Delete** a subordinate asset in a line or department.

13) **Add** a subordinate asset to a line or department.

The following operations can be performed on the program canvas.

1) Create a **New Program** consisting of a single *enterprise* asset and display it on the canvas.

2) **Save** the current **Program** to disk.

3) **Edit** an existing **Program** by prompting the user for the program name and displaying its graph on the canvas.

4) **Compile** the current **Program**.

5) **Run** the current **Program**.

6) Set **Program Options**.

The following operation can be performed on service assets, as well as Name, Code, Options and Compile that are described above.

1) **Delete** a **Service** from the program.

The following operation can be performed on the service canvas:

1) **Add** a new **Service** that the program can use.

5. Implementation

The architecture of *Enterprise* is described elsewhere (Chan et al., 1991). Briefly, it consists of a graphical user interface, a code librarian and an execution manager. The interface allows the program to create *Enterprise* graphs and enter source code. The code librarian manages all the source and object code for an application. From the application's asset graph, the code librarian determines where to insert the communication and synchronization system calls into the user's code and compiles the asset(s). The execution manager spawns the processes at run-time and monitors the system detecting idle and busy machines. This section provides a brief description of the implementation of these components of *Enterprise*. Implicit in the discussion is a computing environment of homogeneous Unix workstations. Heterogeneity is supported but is not yet complete.

5.1 Graphical User Interface

The *Enterprise* user interface is implemented in Smalltalk-80 and runs as a Unix process on Sun workstations. The interface produces a text file representation of the *Enterprise* graph that is used by the code librarian and execution manager. The code librarian and execution manager are started as Unix processes from within the user interface when the *Compile* and *Run* operations are selected.

5.2 Code Librarian

The code librarian manages the user's source and object code (Chan, 1992). Several directories are created and maintained including Src (source code), Obj (object code, organized by target architecture), Inc (include files), Lib (user-defined libraries), and Ent (containing the *makefiles*, graph files and temporary files).

The *Enterprise* pre-compiler is based on the Gnu C compiler *gcc*. Compilation consists of three passes: two by the pre-compiler (one to mark where to insert code and the other to do the insertion) and then one by any conventional C compiler to do the compilation.

5.3 Execution

When an *Enterprise* application is executed, a process is created for each asset and the communication links between processes are established. Replicated assets are allocated one process for each replica as well as one additional process, called an asset manager, to manage all calls to that asset. The asset manager routes calls to replicas that are idle and queues calls when all replicas are busy. The asset manager maintains contact with the execution manager to take advantage of new processors as they become available.

Enterprise uses the ISIS package to do all the low-level communications (Wong, 1992). 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).

Unfortunately, although ISIS appears on paper to provide all the facilities needed in an execution manager, the implementation has serious problems. Our group has uncovered a series of major errors in ISIS which dramatically affect the performance of our programs. We have been working with ISIS for two years now and although most of our bug reports have been addressed by the ISIS developers, problems continue to crop up on a regular basis. Considerable resources have been devoted to tracking down ISIS-related problems. In the next section some of the ISIS performance issues are addressed.

6. Programming in Enterprise

Several applications have been implemented using *Enterprise* (Parsons, 1992). This section highlights a few of the applications to give some insight into the model, its current state of implementation, and system performance.

Gauss-Seidel is an iterative algorithm for solving families of linear equations (Golub and Van Loan, 1989). Given the equation $Ax = x$, with A an N X N matrix and x a N X 1 vector, this algorithm solves for x. The algorithm starts with an initial value for x and then iterates, continually refining it until it converges. A parallel algorithm for finding x can operate asynchronously, with each processor being responsible for a portion of the x vector, assuming shared memory to simplify the sharing of results (Baudet, 1978). Unfortunately, while this algorithm benefits from the lack of processor synchronization, it results in a non-deterministic solution, with respect to the machine precision.

The parallel Gauss-Seidel algorithm was implemented as a line of two with the last member replicated. The shared memory needed for Baudet's algorithm was simulated using a service. Messages sent between *Enterprise* processes ranged from 2k to 16k bytes depending on the problem size. Speedups of 5.4 on 10 processors for a 1500 X 1500 matrix were observed. This particular application is completely asynchronous, so linear speedups are to be expected.

To better understand the reasons for the poor performance, the *Enterprise*-generated ISIS communication code was manually replaced with calls to the NMP communications package, a "friendly" interface to UNIX sockets (Marsland, Breitkreutz and Sutphen, 1991). The NMP version of Gauss-Seidel achieved speedups that were almost twice that achieved with ISIS. This experiment clearly showed that the poor *Enterprise* results were due to bottlenecks in ISIS communication and that *Enterprise* applications suffer a large performance penalty from using ISIS. Note also that it took several days of programming time to implement and debug the NMP version. Creating the *Enterprise* program from the sequential code took only 20 minutes.

Several experiments were done with the ISIS code generated by *Enterprise* in an attempt to isolate the performance problem(s). They revealed that there is an increasing delay in message delivery due to the failure of ISIS to de-allocate forwarded messages. This delay amounts to more that forty times the cost of messages that did not need to be forwarded. The failure to de-allocate messages created the additional problem of the ISIS system processes accumulating these messages in the processor's swap space, sharply reducing the potential problem size solvable before paging costs swamped the execution time. This problem was exacerbated when data-intensive algorithms were used.

Another algorithm implemented was block matrix multiplication. This data-intensive algorithm contrasted two different parallel constructs. The first construct was a line of three assets: create-work, a replicated multiplier, and an assimilator. The second construct was a department with one subordinate. The receptionist of the department generated work for a replicated multiplier asset, and assimilated the results. Both implementations generated frequent and large messages (160k bytes). The best results produced a 2-fold speedup for 2 through 8 processors. Again, because of the problems with ISIS, performance was limited.

An alpha-beta tree-search algorithm has also been implemented in *Enterprise*. The algorithm builds the search tree in a recursive, depth-first manner to a prescribed depth. Leaf nodes were assigned random values according to a user-defined distribution (Marsland, Reinefeld and Schaeffer, 1987). The parallel version used the Principal Variation Splitting Algorithm, which recursively descends the left-most branch of the tree, searching the siblings in parallel during the backing up of the result (Marsland and Campbell, 1982). The implementation had a line of two assets, with the last asset replicated. This algorithm has synchronization points which limits the potential speedup. It has been theoretically shown that with this algorithm, one can expect a speedup that is proportional to the square root of the number of processors (Fishburn, 1981). This computationally-intensive algorithm generates few messages in relation to the CPU costs and is not adversely affected by ISIS. The theoretical speedups were achieved.

Although the number of *Enterprise* applications implemented is small, several points are worth noting:

1) The effort to convert a sequential program to an *Enterprise* program has, in some cases, been almost trivial. The user concentrates on the parallelism in the application, and not the means of implementing the parallelism.

2) Understanding what *Enterprise* does to a program is essential to achieving good performance. Many procedures/functions that could be done in parallel should not be, because the benefits of the parallelism do not out-weigh the cost of communication (ISIS problems aside).

3) ISIS is a serious performance liability. Unless this problem is resolved soon, we may have to consider designing our own communications package.

7. Project Status

Enterprise is a functional system. The current status (November, 1992) is:

1) The graphical user interface is complete.

2) The code librarian manages the source and object correctly for a heterogeneous network of machines. The insertion of *Enterprise* code into a user's code is correctly working for a subset of the *Enterprise* features. Currently, the passing and return of arrays/pointers is not supported.

3) The execution manager is complete and implemented in ISIS. For reasons explained in the previous section, this component may be rewritten in the near future.

4) No effort has been made to implement any performance monitoring and debugging features.

The system is currently used by a small user community. We expect to have the missing features working within the next 6 months.

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. Especially since the diagrams are constructed top-down and are constrained to be semantically correct so that the user can not make a drawing error! 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.

References

G.M. Baudet. The Design and Analysis of Algorithms for Asynchronous Multiprocessors, Ph.D. thesis, Carnegie-Mellon University, 1978.

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.

E. Chan. The *Enterprise* Code Librarian, 1992, M.Sc. thesis, Department of Computing Science, University of Alberta.

E. Chan, P. Lu, J. Mohsin, J. Schaeffer, C. Smith, D. Szafron and P.S. Wong. *Enterprise*: An Interactive Graphical Programming Environment for Distributed Software Development, 1991, Technical report TR 91-17, Department of Computing Science, University of Alberta.

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

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

G.H. Golub and C.F. Van Loan. Matrix Computations, 2nd edition, John Hopkins University Press, Baltimore, Maryland, 1989.

J. Fishburn. Analysis of Speedup in Distributed Algorithms, Ph.D. thesis, Computer Sciences Department, University of Wisconsin-Madison, 1981. University of Wisconsin-Madison, 1981.

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

T.A. Marsland, T. Breitkreutz and S. Sutphen. A Network Multi-processor for Experiments in Parallelism, *Concurrency: Practice and Experience*, 1991, vol. 3, no. 1, pp. 203-219.

T.A. Marsland, A. Reinefeld and J. Schaeffer. Low Overhead Alternatives to SSS*, *Artificial Intelligence*, 1987, vol. 31, no. 1, pp. 185-199.

T.A. Marsland and M.S. Campbell. Parallel Search of Strongly Ordered Game Trees, *Computing Surveys*, 1982, vol. 14, no. 4, pp. 533-551.

I. Parsons. An Appraisal of the *Enterprise* Model, 1992, M.Sc. thesis, Department of Computing Science, University of Alberta, Edmonton.

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.

D. Szafron, J. Schaeffer, P.S. Wong, E. Chan, P. Lu, C. Smith. The *Enterprise* Distributed Programming Model, *Programming Environments for Parallel Computing*, 1992, N. Topham, R. Ibbett and T. Bemmerl, editors, Elsevier Science Publishers, pp. 67-76.

P.S. Wong. The *Enterprise* Executive, 1992, M.Sc. thesis, Department of Computing Science, University of Alberta.

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( argc, argv )
int argc;
char ** argv;
{
   float timeperframe;
   int NumberFish, NumberSteps, NumberFrames, frame;

   /* Extract from program parameters (argc, argv) values for NumberFish,  */
   /* NumberSteps and NumberFrames.                                        */

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

   /* Loop through each frame */
   timeperframe = 1.0 / NumberSteps;
   for( frame = 0; frame < NumberFrames; frame++ )
   {
      /* Do model computations */
      InitModel( NumberFish );
      MoveFish( NumberFish, timeperframe );
      DrawFish( NumberFish, 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 */
   DoConversion( frame );
   npoly = ComputePolygons( &polygontable );

   /* Done!  Send the work to the Split process.  This code appeared in    */
   /* the sequential program:                                              */
   /* Split( frame, npoly, polygontable );                                 */
   /* To pass arrays in Enterprise, we need the pointer and the number of  */
   /* elements.  Hence this is the correct Enterprise code:                */
   Split( frame, npoly, polygontable, npoly );
}
```

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.

Enterprise Code for the Entry Procedures of the Animation Example

enterprise.h

```
#define  ENTERPRISE        0
#define  ENT_MODEL         1
#define  ENT_POLYCONV      2
#define  ENT_SPLIT         3
#define  ENT_MAX_FUNCTIONS 4

#define  ENT_MAINLINE      1
```

enterprise.c

```
void _Ent_Functions[ ENT_MAX_FUNCTIONS ] = {
   _Enterprise(), _Ent_Model(), _Ent_PolyConv(), _Ent_Split()
};

main( argc, argv )
int argc;
char ** argv;
{
   /* Strip off Enterprise parameters (profiling, debugging, monitoring)   */
   /* and a flag which indicates whether this is the mainline of the pro-  */
   /* gram. Adjust arc and argv to be passed on to the programs mainline.  */
   if( mainline )
         _Ent_Functions[ ENT_MAINLINE ]( argc, argv );
   else  _Ent_Main();
}

_Ent_Main()
{
   int function;

   /* Loop receiving messages.  Read the first 4 bytes, telling us which   */
   /* function is being called.                                            */
   for( ; ; )
   {
      function = _Ent_Receive();
      _Ent_Functions[ function ]();
   }
}
```

Model.c

```
/* Model asset */

#define  NUMBER_STEPS    4
#define  NUMBER_FISH    10
```

```
#define  NUMBER_FRAMES  20

Model( argc, argv )
int argc;
char ** argv;
{
   float timeperframe;
   int NumberFish, NumberSteps, NumberFrames, frame;
   char * msg;

   /* Extract from program parameters (argc, argv) values for NumberFish,  */
   /* NumberSteps and NumberFrames.                                        */

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

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

      /* Done! Send work to PolyConv process */
      msg = _Ent_PackMessage( "%d", frame );
      _Ent_SendMessage( ENT_POLYCONV, msg );
   }
}
```

PolyConv.c

```
/* PolyConv asset */

#define MAX_POLYGONS 1000

_Ent_PolyConv()
{
   polygon polygontable[ MAX_POLYGONS ];
   int npoly;
   char * msg;

   /* Get the parameters to the function from a message */
   int frame;
   _Ent_UnPackMessage( "%d", &frame );

   /* Convert polygons. */
   DoConversion( frame );
   npoly = ComputePolygons( &polygontable );

   /* Done! Send the work to the Split process.  Note that %A is a special */
   /* Enterprise format character (array).  It takes 3 parameters: a       */
   /* pointer to the array, the size of each element in the array and      */
   /* the number of elements of the array to pass.  Note here that we do   */
   /* pass the entire array; only the portion that the user wants.         */
   msg = _Ent_PackMessage( "%d%d%A", frame, npoly,
```

```
                &polygontable, sizeof( polygon), npoly );
   _Ent_SendMessage( ENT_SPLIT, msg );
}
```

Asset Code: Split

```
/* Split asset */

#define  MAX_POLYGONS   1000

_Ent_Split()
{
   /* Get the parameters to the function from a message */
   int frame, npoly, numb;
   polygon polygontable;
   _Ent_UnPackMessage( "%d%d%A", &frame, &npoly, &polygontable, &numb );

   HiddenSurface( frame, npoly, &polygontable );
   AntiAlias( frame, npoly, &polygontable );
}
```