The Enterprise Model for Developing Distributed Applications

Jonathan Schaeffer, Duane Szafron, Greg Lobe, Ian Parsons

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

{duane, jonathan}@cs.ualberta.ca

ABSTRACT

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 correctly handle communication and synchronization, allowing the rapid construction of distributed programs. The system supports load balancing and dynamic distribution of work in environments with changing resource utilization. This paper focuses on the Enterprise programming model and some experiences using it.

Keywords: Distributed computing, parallel programming, programming environments, message passing, programming model

1. Introduction

Developing distributed software is often perceived as a complicated endeavor. Its design, implementation and testing is considerably more difficult than comparable sequential software. Although efficient parallel algorithms exist for a large class of problems, finding the right algorithm may only be a small fraction of the cost of implementing it. Writing distributed software is complicated by additional problems not found in the sequential environment. These problems include synchronization, deadlock, communication, fault tolerance, 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 additional 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 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 correctly handle communication and synchronization, allowing the rapid construction of distributed programs. This bridges the complexity gap between distributed and sequential software. The system supports load balancing 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. Although *Enterprise* currently runs on a network of workstations (because those are the resources that are available to us), there is nothing in the programming model that precludes its application on parallel multiprocessor architectures. Therefore in this paper, we often use the word parallel instead of distributed.

Enterprise has a number of features that distinguish it from other parallel and distributed program development tools (more details can be found in Section 7):

1) Programs are written in a sequential programming language (C) that is augmented by new semantics for procedure calls that allows them to be executed in parallel. Users do not deal with low-level programming details such as marshalling data, sending/receiving messages and synchronization. Instead, *Enterprise* inserts all of the necessary communication protocols automatically into the user's code. Since the user is often the best judge of how parallelism can be exploited in a particular application, *Enterprise* does not automatically parallize a program. It provides the user with a high-level mechanism for specifying the parallelism.

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 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 introduces a consistent terminology based on *assets* (individuals, departments, receptionists, etc.).

4) *Enterprise* supports 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 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 legal diagrams in a top-down manner by transforming and expanding nodes. Transformation 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. In a "connect-the-nodes" system, the diagram may be illegal until the user is finshed (e.g. an isolated node with no connections) whereas in *Enterprise* the diagram is always legal.

6) The *Enterprise* run-time libraries have a simple generic interface to a message-passing kernal. Currently, two different comminications kernals, ISIS (Birman et. al., 1991) and NMP (Marsland et. al., 1991), have been connected to Enterprise, and two more are being implemented, PVM (Geist and Sunderam, 1992) and Concert/C (Auerbach et. al., 1992). The facilities provided by ISIS allow *Enterprise* to support a limited degree of fault tolerance with little implementation effort. On the other hand, PVM and Concert/C provide support for a network of heterogeneous computers. The portable interface has allowed us to contrast the features of these kernals and to benchmark their performance (Parsons, 1993). None of the kernals tested provides all of the features we would like in *Enterprise*.

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 and

synchronization. It then compiles the routines, assigns processes to processors and establishes the necessary connections.

A frequently cited criticism of high-level programming tools such as *Enterprise* is that since they hide many of the hardware details the resulting programs are often inefficient. In fact, hiding hardware specifics in a distributed environment can result in major performance degradation (Jones and Schwartz, 1980). The assets supported by *Enterprise* are application independent and produce good performance results on a large cross-section of coarse-grained applications. However, the user is not restricted only to the parallelism provided by *Enterprise*. After using *Enterprise* to express the coarse-grained parallelism in an application, the user can compile individual assets using another parallel system to take advantage of fine or medium-grained parallelism.

The *Enterprise* model described in this document is an evolution of an earlier design (Chan et al., 1991; Szafron et al., 1992) and our *Frameworks* system (Singh, Schaeffer and Green, 1991). This paper discusses the *Enterprise* programming model. Additional implementation details can be found elsewhere (Lobe et al., 1992). A description of the animation and debugging facilities are beyond the scope of this paper and are described in (Lobe, Szafron and Schaeffer, 1993).

Section 2 describes the key components of the *Enterprise* model: the programming semantics (source code) and the meta-programming semantics (diagrams). Section 3 describes an example application. Section 4 gives a brief description of the implementation and section 5 discusses some *Enterprise* programming experiences. Section 6 describes some of the limitations of the system. Section 7 discusses the relationship between *Enterprise* and the work of others. Section 8 puts the *Enterprise* project in perspective.

2. The Enterprise Parallel Programming Model

The user views an *Enterprise* program as a collection of modules. Each *module* contains 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. Within a module, the code is executed sequentially, but different modules may be executed concurrently. Since *Enterprise* is a distributed computing model, no common variables can exist between modules (no shared memory). Since a module may only call the entry procedure of other modules, this abstraction is similar in many ways to programming with encapsulation, which provide well-defined means for manipulating data structures while hiding all the underlying implementation details from the user. Since a module can contain an unlimited number of internal procedures, a sequential program 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 *call* to a module and the *role* of a module. The call to a module defines the identity of the called module, the information passed and the information returned. The role of a module defines which one of a fixed set of parallelization techniques the module will use when it is invoked. The call of a module is specified in the source code and the role of a module is specified graphically.

The *Enterprise* Parallel Programming Model has two components: a programming semantics and a meta-programming semantics. The first component defines the semantics of module calls. The second component defines the semantics of module roles. It is called the meta-programming component because *Enterprise* reads the source code and the graph and then generates its parallel code. The two components are somewhat orthogonal. That is, the meta-program (graph) can be modified to specify different parallelization techniques for the same program (source code) or the program can be modified to some extent without changing the structure of the meta-program.

2.1 Enterprise Program Semantics

In a sequential program, procedures communicate using procedure calls. The calling procedure, say *A*, contains a call to a procedure, say *B*, that includes a list of arguments. When the call is made, *A* is suspended and *B* is activated. *B* can make use of the information passed as arguments. When *B* has finished execution, it can communicate results to *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 function and procedure calls and 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 and an *Enterprise* module 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 *A* would suspend itself when it tried to use the function result, if *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* (somewhere on the network). *A* would continue executing concurrently with *B*. However, when *A* tried to access the result of the call (`value = result + 1`), it would block until *B* had returned a message containing its `result`. This concept is similar to 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 accept a variable number of parameters of varying types. Arrays and pointers are valid parameters but they must be immediately followed by a 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 an array parameter. This feature allows users to pass dynamic arrays and sub-arrays. The data being passed cannot contain pointers (which would be meaningless because of the distributed memory).

Enterprise-defines three macros for parameter passing, similar to those used in Concert/C (Auerbach, 1992). 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. The caller will block if it accesses the value of an OUT or INOUT parameter that has not yet been returned. For example, the code

```
int data[100], result;
. . .
result = B( &data[60], INOUT(10) );
/* some other code */
value = data[65] + 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`. *A* will block when it accesses `data[65]` if the call to *B* has not yet returned. If no parameter specification is included then IN is assumed. Note that in sequential C programs, since arrays are passed by reference, INOUT is the default. In *Enterprise*, choosing IN as the default will help make the programmer more aware of the overhead of passing data.

The use of OUT or INOUT parameters cause side-effects. Thus, in some sense, they can also be considered part of the return value of a 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. Procedure calls containing OUT or INOUT parameters actually return results, albeit by side effect, and are therefore really f-calls. Purely asynchronous calls cannot return a result and must have only IN parameters.

Although lazy synchronous calls provide greater opportunities for concurrency than synchronous calls, it is often 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 situations where the order in which results are returned by a module 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 module may have to block and wait for `result[0]`. However, other results may have already been returned, say `result[1]` and `result[4]`. Since the value of sum is independent of the order of summation of the results, 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 a module is *ordered* or *unordered*. If unordered, then the return values of the module are consumed in the order that the module values are returned, not the order in which they are referenced. For example, if the *B* module 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. The order attribute (ordered or unordered) of a module is part of the role of the module and is specified when meta-programming using the graphical user interface. The order attribute is independent of the module code.

There are a few additional subtleties in the programming semantics that are not discussed here (see Lobe at al., 1992, for details). Two worth mentioning are aliasing and I/O. Currently, *Enterprise* does not support aliasing of returned values (the cost may be prohibitive). That is, if an aliased reference exists to a variable that has not yet been returned, *Enterprise* will not block when

a reference to the alias is made. *Enterprise* also assumes the existence of a global file system (e.g. NFS) and no special concurrency control mechanisms for file I/O are provided.

2.2. Enterprise Meta-programming Semantics

This section defines the meta-programming semantics of the *Enterprise* Model. The purpose of this component of the model is to specify the role of modules; which of the modules in a program will be executed in parallel and what form that parallelism will take.

Enterprise uses an analogy between the structure of programs and the structure of organizations to describe a program's parallelism. The analogy is successful because organizations are both familiar and inherently parallel. An organization divides large tasks into smaller tasks and allocates its assets to perform these tasks. The number and kind of asset used to perform a task determine the amount of parallelism that results. For example, an *individual* performs a task sequentially, but four individuals can perform four similar tasks concurrently. A *department* can sub-divide a task and its components can perform the sub-tasks concurrently. An assembly or processing *line* can start a second task as soon as the first component of the line has passed the first task to the second component of the line.

Enterprise modules correspond to organizational assets since they are resources that can be executed in parallel. Therefore, in the meta-programming model, we will refer to *Enterprise* modules as *assets*. There are two attributes that determine the parallelism associated with an asset: its kind and its replication factor. *Enterprise* currently supports the following asset kinds: *enterprise*, *individual, receptionist, representative, department, line, division* and *service*. Figure 1 shows the icons that represent each of the asset kinds in *Enterprise*.

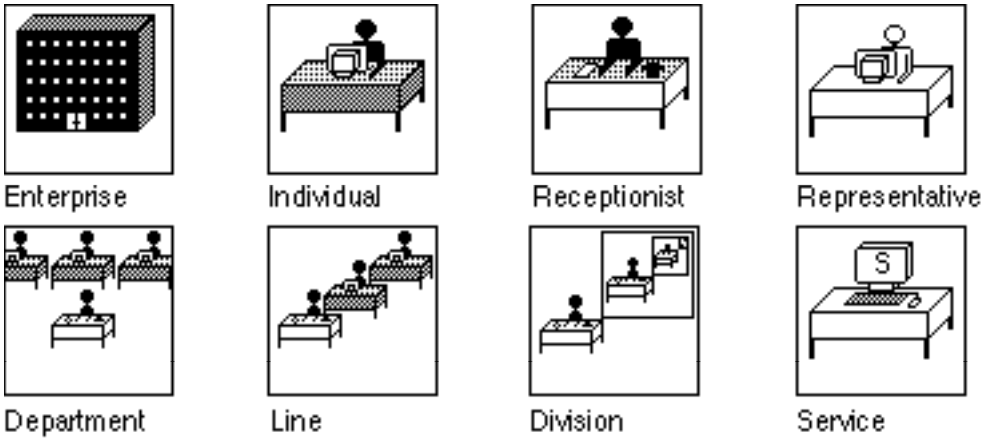

Figure 1: Enterprise asset icons.

Two examples, a graphics Animation program and a Mergesort program will be used to illustrate the parallel semantics of each of these asset kinds and the meaning of replication. The

Animation program was first introduced in *Frameworks* and its evolution is a good illustration of the superiority of the *Enterprise* model over our earlier *Frameworks* approach.

2.2.1 Enterprise Assets

An *enterprise* asset represents a program and is analogous to a single organization (or enterprise). An enterprise asset shares its name with the program. Every enterprise asset contains a single component asset. This component is an individual asset by default, but it is possible to *transform* the component to a different kind of asset.

For example, when *Enterprise* is started, the *Enterprise* window contains a single pane, called the *Program Pane*, that contains a single enterprise asset. The enterprise asset can be named *Animation* and can be expanded to reveal its single component asset that is an individual.

2.2.2 Individual Assets

An *individual* asset is an asset that does not contain any other assets and is analogous to a person in an organization. An individual asset represents a module that does not contain any other modules. It has source code and a unique name that it shares with the module that it represents. When the module of an individual asset is called, it executes its sequential code to completion. Therefore, any subsequent call to the same individual must wait until the previous call is finished. An individual can be transformed to a line, department or division.

For example, the user could enter all of the code for the Animation program into a single individual asset that is contained in the Animation program. If the program was then compiled and run, the program would execute sequentially.

2.2.3 Line Assets

A *line* is an asset that 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. Line assets are usually called pipelines in the literature.

For example, the Animation program contains a loop that is executed once for each frame in the animation. Inside the loop, a model is constructed for the frame and a procedure called *PolyConv* is called to convert the model to polygons and render it. Inside procedure *PolyConv*, the last task is a call to procedure *Split* that does the final rendering operations. However, there is

no reason for the program to wait until *PolyConv* completes the first animation frame to start processing the model for the second frame. Similarly, *PolyConv* does not need to wait for *Split*. In other words, the program could be regarded as three modules in a line. Therefore, an *Enterprise* user could transform the single individual component of the Animation program into a line of three individuals (*Model*, *PolyConv* and *Split*) and enter the appropriate code for each. Figure 2 shows the Animation program design consisting of an expanded enterprise asset (with a double line border) that contains an expanded line asset (with a heavy solid and dashed line border) that contains the three individuals. Figure 2 also shows a code editor for the PolyConv asset.

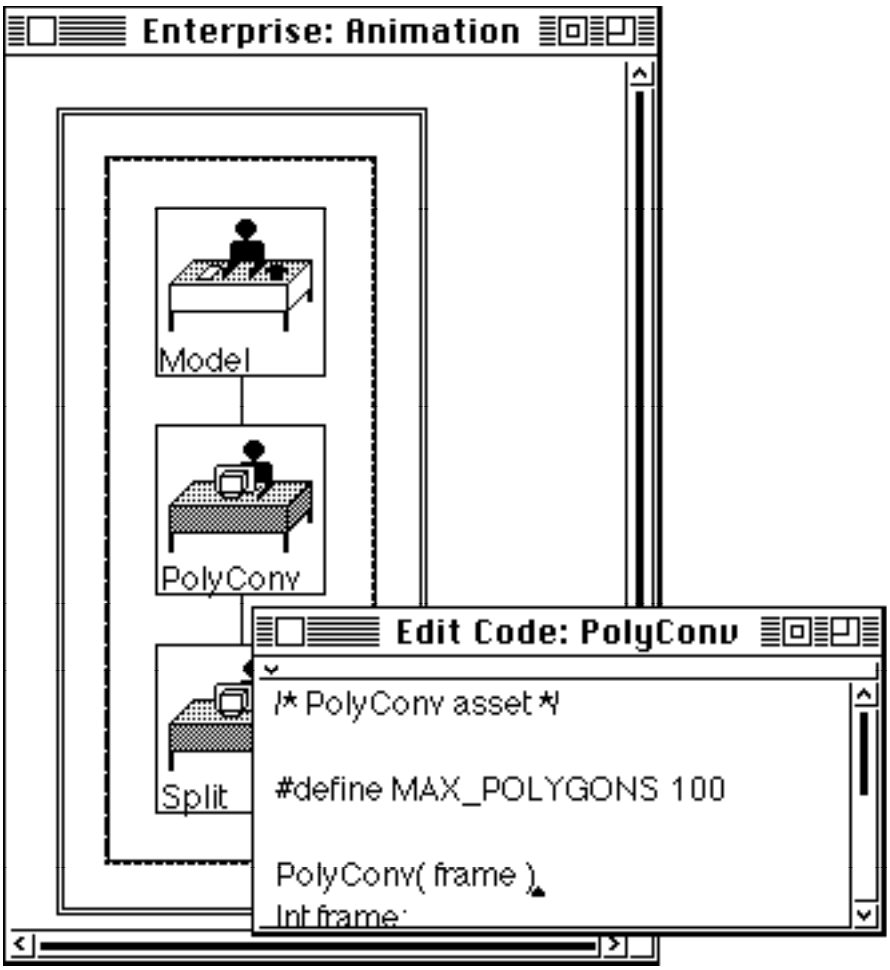

Figure 2: A parallel design for the Animation program.

Enterprise uses the host windowing system. The figures in this paper were generated on the Macintosh implementation of the *Enterprise* user interface and look slightly different in X-windows or Sun Open Windows. If the user selects the *compile* option in the user interface, *Enterprise* would insert the code necessary to handle the distributed computation, compile the program and report any compilation errors in a window. If the *run* option is then selected, *Enterprise* would

allocate three processors for the program, one for each individual in the line. In this 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.

2.2.4 Replicating Assets, Receptionist Assets and Ordering

Assets may be replicated so that when an *Enterprise* program is run, more than one process can simultaneously execute the code for the same asset. The user specifies a minimum and maximum (possibly unbounded) replication factor and *Enterprise* dynamically creates as many processes as possible (one per processor) up to the maximum. All assets except receptionists and enterprises (programs) can be explicitly replicated. A *receptionist* cannot be explicitly replicated since it serves as the entry point for a composite asset (line, department or division). 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. Section 2.1 contains an example that illustrates the utility of unordered assets.

One of the strengths of the *Enterprise* model is that meta-programming allows the user to experiment with alternate parallelization techniques without changing the C source code. In the Animation example, the *Split* asset can be replicated so that multiple processes can be used to execute multiple calls concurrently. When *PolyConv* calls *Split*, a process will be initiated and if a subsequent call is made to *Split* before the first call is done then a second process will be initiated (if there is an available machine). Since replication is dynamic, 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* (using 8 processors) 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*.

2.2.5 Department Assets

A *department* asset contains a fixed number of heterogeneous assets. A single receptionist asset shares its name with the department. 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. For example, the code:

```
SolveLinear()
{
   matrix * m;
   int property, numElements;
   . . .
   for( ; ; )
   {
         m = GetMatrix( &numElements );
         if( m == NULL )
               break;
         property = MatrixProperty( m, numElements );
         if( property == SPARSE )
               SolveSparse( m, numElements );
         else if( property == TRI_DIAGONAL )
               SolveTri( m, numElements );
         else  SolveGauss( m, numElements );
   }
}
```

solves a series of linear equations that are represented by matrices and writes the solution to disk. There are three different ways of solving the systems, depending on the property of the matrix that represents it. This code could be entered as an individual, *SolveLinear*, with local procedure calls, compiled and run sequentially. Alternately, the individual, *SolveLinear* could be transformed into a department asset that contains the individuals, *SolveSparse*, *SolveTri* and *SolveGauss*. In this case, when a p-call is made to one of these procedures, say *SolveSparse*, execution continues. In the next loop iteration, if *SolveTri* is called then it is executed concurrently with *SolveSparse*. However, if a subsequent iteration calls *SolveSparse* again, the caller will block until the outstanding call has completed. To prevent this block, *SolveSparse* can be replicated to increase the concurrency. If each of *SolveSparse*, *SolveTri* and *SolveGauss* are replicated with no maximum replication factor, then all the available processors will by dynamically assigned to maximize the concurrency. Note that whether this code is executed as an individual or as a department of indiduals or a department of replicated individuals, no changes are made to the users source code. A simple change to the diagram is all that is required.

2.2.6 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. Similarly, line assets could contain other line assets.

For example, the C code for the *Split* asset contains two local procedure calls, HiddenSurface and AntiAlias. Instead, the *Split* asset could be transformed to a line that contains one local

procedure call to HiddenSurface that in turn contains an asset call to *AntiAlias*. The code for *AntiAlias* would need to be moved from the *Split* asset to the *AntiAlias* asset but would not need to be edited. Figure 3a shows this modified design where the *Split* asset has been replicated.

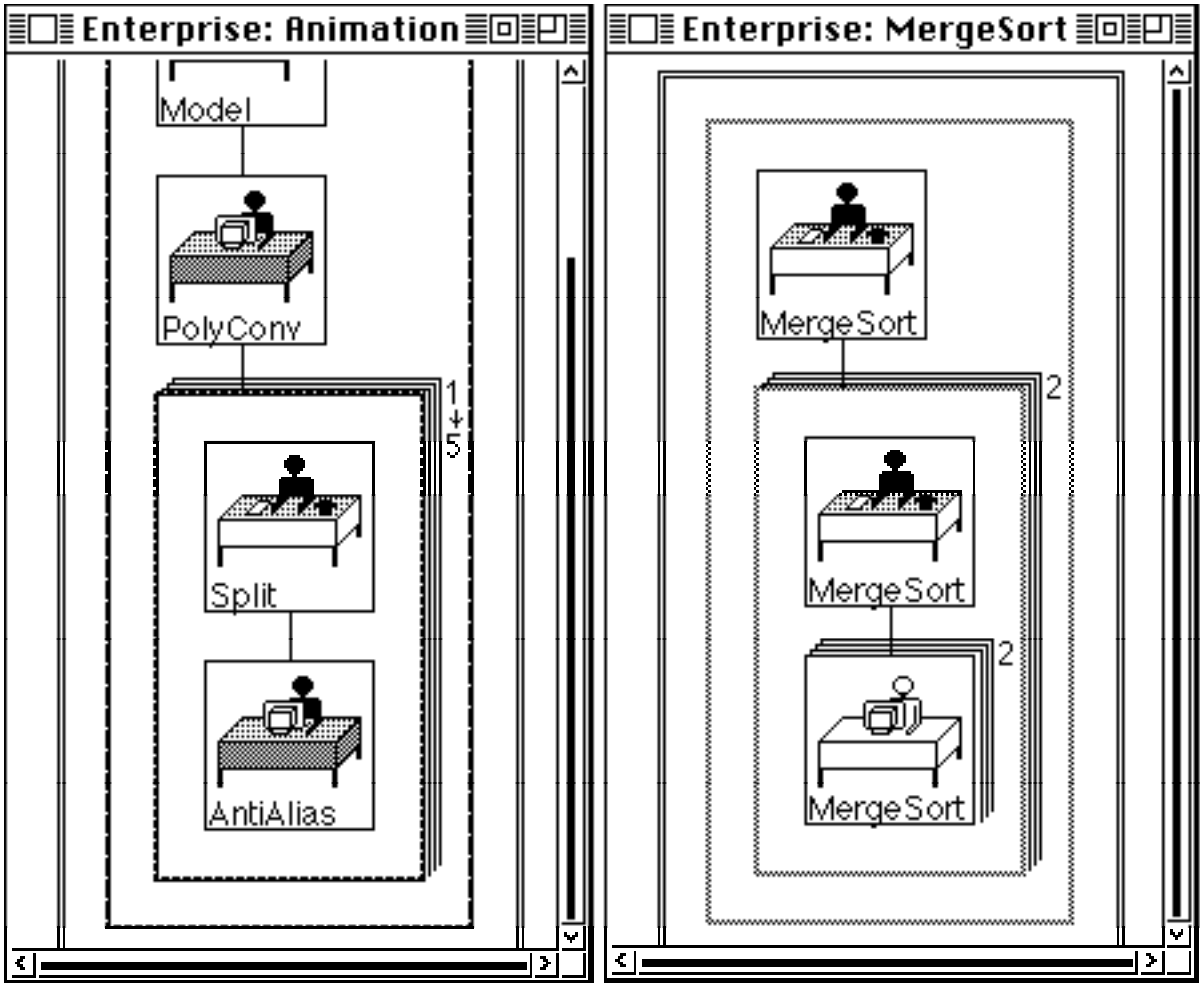

Figure 3: *Enterprise* designs for the Animation and Mergesort programs.

2.2.7 Division and Representative Assets

A *division* asset contains a hierarchical collection of identical assets where work is divided and distributed at each level. It can be used to parallelize divide-and-conquer computations. When a division is created, it contains a single receptionist asset that shares its name with the division and a single *representative* asset that represents the recursive call made by the receptionist to the division itself. Divisions are the only recursive assets in *Enterprise*. The user may change the breadth of the division 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 will contain 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 3b shows a Mergesort program with a breadth of 2 at each level and a depth of 2. Note that when the minimum and maximum replication factors are the same, the common value is displayed.

2.2.8 Service Assets

A *service* asset 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. Example

Consider a parallel Mergesort routine. The sequential code might look something like this:

```
Mergesort( data, size )
int * data, size;
{
      int mid = size / 2;
      if( size > 1 ) {
            Mergesort( &data[0]  , mid );
            Mergesort( &data[mid], size - mid );
            Merge( data, size );
      }
}
```

Because of the divide-and-conquer nature of the algorithm, the obvious asset choice for Mergesort is a division. The calls to Mergesort modify the `data` array by side-effect. Since the values of data must be passed to Mergesort and the modified values returned, the programmer must alter the sequential code to make this INOUT relationship explicit:

```
            Mergesort( &data[0]  , INOUT( mid ) );
            Mergesort( &data[mid], INOUT( size - mid ) );
```

The user can take the individual asset Mergesort and convert it to a division. For this example, the obvious choice for the breadth is 2 although, of course, any size could be chosen. This would result in 3 Mergesort processes, a parent and two children. If the sorts are large, the user might want to transform each of the children from individuals to divisions as well. Figure 3b shows a division of divisions, each of breadth 2.

```
char  *  _ENT_Mergesort(  msg  )
char  *  msg;
```

```
{
      int  *  data,  size;

      /*  Get  the  parameters  to  the  function  from  a  message  */
      /*  %A  is  a  special  format  used  for  arrays.  */
      _ENT_UnpackMessage_MergeSort(  msg,  &data,  &size  );'
      Mergesort( data, size );
      _ENT_PackMessage_MergeSort(  msg,  data,  size  );
      return(  msg  );
}

Mergesort( data, size )
int * data, size;
{
      int mid = size / 2;

      if( size > 1 ) {
            /*  Asset  call  if  process  connected  else  sequential  */
            if(  _ENT_Child  !=  NULL  )
                  /*  Pack  parameters  into  message  and  send  */
                  _ENT_SendCall_MergeSort(  &data[0],  INOUT(mid),
                                                _ENT_NO_REPLY  );
            else Mergesort( &data[0], INOUT(mid) );

            /*  Asset  call  if  process  connected  else  sequential  */
            if(  _ENT_Child  !=  NULL  )
                  /*  Pack  parameters  into  message  and  send  */
                  _ENT_SendCall_MergeSort(  &data[mid],  INOUT(size-mid),
                                                _ENT_NO_REPLY  );
            else Mergesort( &data[mid], INOUT(size-mid) );

            /*  Future:  wait  for  INOUT  paramter  to  return  */
            if(  _ENT_Child  !=  NULL  )  {
                  _ENT_Wait(  &data[0],  mid  );
                  _ENT_Wait(  &data[mid],  size-mid  );
            }
            Merge(data, size);
      }
}
```

Figure 4. *Enterprise* code for Mergsort.

The first call to Mergesort will divide the list in half and send each part independently to two other processes. They, in turn, will divide the list and pass the parts on to their children. These processes have no 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. Figure 4 illustrates what *Enterprise* does to the sequential Mergesort code to make it run in parallel (code inserted by *Enteprise* is in bold, and has been edited for brevity). Figure 5 illustrates the processes created by *Enterprise* and their interconnections. All processes are connected to the Execution Manager (who monitors the activity on each machine and decides when process migration should occur) and the Monitor Manager (who interfaces to a graphical display so

the user can view the progress of the computation). In addition, each division has its own manager process. These managers keep track of which processes are busy or idle and forwards work to the first available asset. Although for this particular example, a more efficient implementation could be had by bypassing the managers and hard coding the connections, this would reduce the flexibility of the system. The managers support an arbitrary number of assets (for example, if the user changes the replication factor, or specifies unlimited breadth).

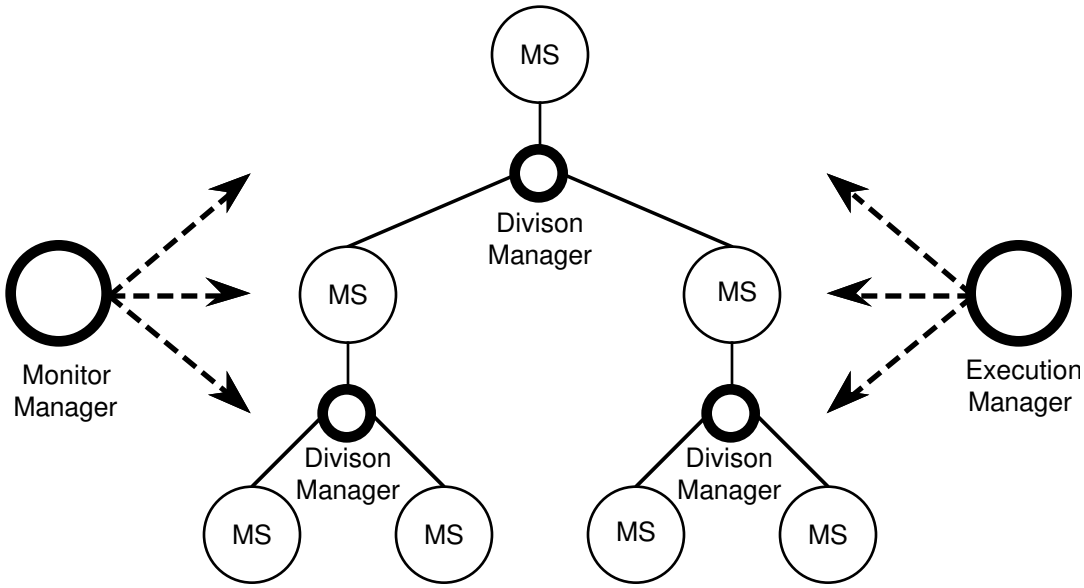

Figure 5. Process Diagram for Mergesort (MS) example.

Once the user has specified the *Enterprise* diagram and provided the code, the user can select the *compile* button on the interface menu. *Enterprise* incrementally maintains a makefile and re-compiles the changed portions of the program. Selecting the *run* button causes *Enterprise* to go out, find as many available machines as it can (either user specified or automatically allocated), spawn the processes, establish the interconnections and start the user's program running.

4. Implementation

The architecture of *Enterprise* is described elsewhere (Chan et al., 1991). Briefly, it consists of a graphical user interface, a code librarian, a pre-compiler 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 pre-compiler 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. Currently, hardware heterogeneity is only supported to the extend provided by the underlying message-passing kernal (e.g. PVM or Concert/C).

Graphical User Interface: The *Enterprise* user interface is implemented in Smalltalk-80 and runs as a Unix process on Sun workstations (Lobe et. al., 1993). 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.

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).

Pre-compiler: The 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. The pre-compiler checks for consistency between the assets calls in the user code and the graph file and reports any inconsistencies to the user.

Execution Manager: 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.

The*Enterprise* run-time library has been generalized to use one of several message-passing kernals (ISIS, NMP, PVM or Concert/C) to do all the low-level communications. Each provides a high-level set of function calls to handle process creation and termination, communication and synchronization. In addition, ISIS supports fault tolerance while Concert/C and PVM support a heterogeneous collection of machines. Locally, we use the NMP implementation because of its simplicity and high performance.

Enterprise is a functional system and an alpha version is currently being tested by a small user community. The system allows the user to animate computations (in real-time or replay mode) and inspect message contents for performance monitoring and debugging purposes (Lobe et al., 1993).

5. Programming in Enterprise

Several non-trivial applications have been implemented using *Enterprise* (Parsons, 1992). This section highlights two 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. 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). This algorithm benefits from the lack of processor synchronization, but results in a non-deterministic solution.

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.

Another *Enterprise* application is an Alpha-Beta tree-search algorithm that builds a search tree in a recursive, depth-first manner to a prescribed depth. Leaf nodes were assigned random values according to a user-defined distribution. 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.

Table 1 contains the number of lines of source code for several applications. Each application was implemented in both NMP and *Enterprise* (neither the Animation nor Alpha-Beta applications appear in the table since there were no NMP implementations available). The common libraries of sequential code were extracted before doing the comparison. These line counts are inflated since they include blank lines, comments, debugging code and statistics gathering routines. In fact, about half of the lines are blank or comments. Note that in the first two applications, no extra code is inserted to parallelize the program using *Enterprise* since existing procedure calls were just converted to *Enterprise* assets. Block Matrix Multiply is discussed in Parsons (1992). Transitive Closure was implemented as an NMP assignment in a graduate course. This table shows that for these applications, very little code needs to be added to parallelize the program using *Enterprise*. This economy of expression also translates to reduced development time.

Name	Sequential Source Code	NMP Source Code	Enterprise Source Code	Sequential Libraries
Gauss-Seidel	647	1357	647	22
Block Matrix Multiply	336	765	336	467
Transitive Closure	126	537	132	4177

Table 1. Comparison of source code size for *Enterprise* and NMP

In most applications no changes are made to the sequential code. In our experience, there are two places where changes should be made:

1) Since concurrency cannot occur unless there are more than one call to an asset, sometimes it is necessary to modify the sequential program to introduce additional asset calls, each with a smaller granularity but with the same cumulative effect. For example, in the sequential version of the Transitive Closure application, there is one routine that accounts for 99% of the execution time and there is only one call to this routine:

```
TC (0, ProblemSize );
```

This call was transformed into a loop containing multiple calls:

```
size = ProblemSize / Pieces;
for ( i = 0; i < Pieces; i++ )
{
   TC (i * size, (i + 1) * size );
}
```

Obviously, the value of `Pieces` is implementation dependent. Setting it too large creates many small tasks which may not be cost-effective. Setting it too low limits the concurrency.

2) Since there is no support for shared variables in the Enterprise model (except through services), some global variables in sequential programs must be passed as parameters in asset calls. For example, the sequential version of the Alpha-Beta search program used five global variables that contained the state of the search tree. To adapt this program to the *Enterprise* model, these variables were placed in a structure that was passed as a parameter to each asset call. The structure size was approximately two thousand bytes which, because of the coarse granularity of the assets' computations, did not adversely affect its parallel performance.

Some programs may require more extensive modifications, depending on their data structures and flow of control model.

These experiences support the assertion that the *Enterprise* model reduces the time and effort required to develop an application. 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.

#3) Although there are no limitations in Enterprise to the number of processors that can be used by an application, we have not used more than 20 processors in our tests. The scalability of an application depends on the number and granularity of the assets.

6. Limitations of Enterprise

- programming: no shared variables, no embedded pointers

- model: more assets may be required, no dynamic depths, a module name can appear only once in a graph.

- performance: extra processes and communication overhead

- no low level access e.g. testing if a result has returned without blocking

- control flow but no assets to support data parallel

- changes to the graph may involve code changes

7. Related Work

In recent years, graphical parallel programming environments have been an area of active research. In these environment, a parallel program is usually specified as a graph with nodes containing a textual description of a sequential program. Most of the environments specify parallelism based on the large grain dataflow model developed by Babb (1984) (for example, CODE: Browne, Azam and Sobek, 1990; DGL: Jagannathan, et al., 1989; LGDF: DiNucci and Babb, 1989; Paralex: Babaoglu, et al., 1991; PPSE: Lewis and Rudd, 1990; TDFL: Suhler, et al., 1990). In these models, an application is usually defined as a dataflow graph whose nodes contain sequential modules and the edges represents data dependencies between the modules. Typically, the granularity of work supported by these applications is finer than in *Enterprise*.

The dataflow approach has a number of important drawbacks. Usually it requires the programmer to supply information which is often not intuitive to the programmer. Parallelism by replication of processes is hard to specify. In many of the systems, the programmer has to draw the data dependencies connecting every node. The graph can become a dense structure, even with a small number of modules, and it is often time consuming and error prone to draw. In contrast, since design in *Enterprise* is at a higher-level, the graphs are smaller.

Another research area is in graphical environments to support dependency analysis of programs which are usually written in Fortran and run on machines with a small granularity (for example, PAT: Smith, Appelbe and Stirewalt, 1990; E/SP: Sridharan, et al., 1989; CAPER: Sugla, Edmark and Robinson, 1989). Yet another class of approaches provide a set of predefined parallel objects which define the communication and execution behavior of the enclosed module. Among the earlier efforts, the Parallel-programming and Instrumentation Environment (PIE) made available to the programmer pre-coded templates of parallel structures such as master-slave, pipeline and systolic multidimensional pipeline (Segall and Rudolph, 1985). A template provided an implementation of the communication and synchronization structures and the programmer needed only to provide the sequential code. However, the parallelism supported is at a much smaller grain size than that offered by *Enterprise* (such as the provisions for systolic structures). In contrast, *Enterprise* focuses on problem solving at a higher level, using the analogy to the structure of organizations. This allows new parallel templates to be defined whenever necessary. In our earlier system, *FrameWorks*, dynamic initiation of replicated process according to available idle processors was introduced. The programmer also needed to specify parallelism, in a less

intuitive way, through templates defining the input and output communication pattern of the process.

There are a variety of other programming models such as Linda (Carriero and Gelernter, 1988), the Chare-Kernel (Fenton, et al., 1991), PVM and Hence (Beguelin, et al., 1991). However in these systems, the onus is still on the programmer to use the supplied programming language enhancements and explicitly code all the parallelism.

8. Conclusions

Enterprise is a tool for simplifying the process of designing and implementing distributed applications. By keeping the programming model as close to the familiar sequential model as possible, many complexities of distributed programming become hidden from the user. The graphical interface allows the user to easily construct frequently occurring parallel structures, without undergoing the time-consuming, error prone process of hand-coding it. The user must still have a good understand of the parallelism in their application if they want good performance. *Enterprise* does not support arbitrarily structured parallel programs. However, for a useful class of applications, *Enterprise* relieves the user from the tedious details of distributed communication, allowing the user to concentrate on the development of the algorithm.

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 goal of the *Enterprise* project is to build an integrated programming environment that is suitable for the complete parallel software development life cycle. The user interface provides a good mechanism for design. Automatic generation of parallel code provides a means of generating code quickly and reduces the probablility of programmer error. The orthogonality of asset diagrams and asset code simplifies the process of experimenting with different parallel configurations. The animation facilities provide support for both performance tuning and high-level debugging. Currently, we have not adequately addressed issues such as execution monitoring, statement-level debugging, integeration with other programming languages and performance optimization.

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, familiar programming model, most of the complexity of parallel computing can be hidden from the user.

Acknowledgements

The *Enterprise* project has benefitted from the efforts of many people, including Paul Lu, Stan Melax, Ron Meleshko, Jimmy Mohsin, Carol Smith, Pok Sze Wong and Zhonghua Yang. 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

J. Auerbach, M. Kennedy, J. Russell and S. Yemini. Interprocess Communication in Concert/C, IBM Technical Report RC#17341, IBM T.J. Watson Research Centre, Yorktown Heights, 1992.

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

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

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

A. Beguelin, J.J. Dongarra, G.A. Geist, R. Manchek and V.S. Sunderam. Graphical Development Tools for Network-Based Concurrent Supercomputing, Supercomputing 91, pp. 435-444, 1991.

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

J.C. Browne, M. Azam and S. Sobek. CODE: A Unified Approach to Parallel Programming, *IEEE Software*, vol. 6, no. 6, pp. 10-18, 1989.

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

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

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, Technical report TR 91-17, Dept. of Computing Science, University of Alberta, 1991.

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

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

W. Fenton, B. Ramkumar, V.A. Saletore, A.B. Sinha and L.V. Kale. Supporting Machine Independent Programming on Diverse Parallel Architectures, *International Conference on Parallel Processing*, pp. 193-201, 1991.

G.A. Geist and V.S. Sunderam. Network-Based Concurrent Computing on the PVM System, *Concurrency Practice and Experience*, vol. 4, no. 4, pp. 293-311, 1992.

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, *International Conference on Parallel Processing*, pp. 54-58, 1989.

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

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

G. Lobe, P. Lu, S. Melax, I. Parsons, J. Schaeffer, C. Smith and D. Szafron. The *Enterprise* Model for Developing Distributed Applications, Technical report TR 92-20, Dept. of Computing Science, University of Alberta, 1992, available by ftp from menaik.cs.ualberta.ca.

G. Lobe, D. Szafron and J. Schaeffer. The Object-Oriented Components of the Enterprise Parallel Programming Environment, TOOLS USA 93, August, 1993. A longer version appears as: Program Design and Animation in the *Enterprise* Parallel Programming Environment, Technical report TR 93-04, Dept. of Computing Science, University of Alberta, 1993, available by ftp from menaik.cs.ualberta.ca.

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

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

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

I. Parsons. Evaluation of Distributed Communication Systems, CASCON '93, IBM Toronto, 1993.

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

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,* vol. 2, no. 1, pp. 52-67, 1991.

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

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, *International Conference on Parallel Processing,* pp. 98-106, 1989.

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

P.A. Suhler, J. Biswas, K.M. Korner and J. C. Browne. TDFL: A Task-Level Dataflow Language, *Journal of Parallel and Distributed Computing*, vol. 9, no. 2, pp. 103-115, 1990.

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

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