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

e-mail: {jonathan, duane}@cs.ualberta.ca

Summary

Enterprise lets developers write sequential code and then graphically express the parallelism in their applications using an analogy to a business organization. The system automatically converts the application to run on a network of workstations. Because the (sequential) code that calls the parallel procedures is independent of these procedures, programmers can adapt applications to varying numbers and types of processors without rewriting their code. Enterprise's analogy between program structure and organizational structure eliminates inconsistent terminology (pipelines, masters, slaves, and so on) and introduces a consistent terminology based on assets.

Jonathan Schaeffer is an associate professor of computer science at the University of Alberta. His research interests include parallel programming environments and algorithms, and heuristic search. He received a PhD and MMath from the University of Waterloo, and a BS from the University of Toronto. He is a member of the IEEE, ACM, and AAAI.

Duane Szafron is an assistant professor of computer science at the University of Alberta. His research interests include object-oriented computing, programming environments, and user interfaces. He received a PhD from the University of Waterloo, and an MS and BS from the University of Regina.

Ian Parsons is a doctoral candidate at the University of Alberta. His research interests include parallel programming environments for distributed memory applications. He received an MS (in 1992) and a BS (in 1991) in computing science from the University of Alberta, and a BS in chemistry from the University of Western Ontario in 1978.

Greg Lobe is works for Bell Northern Research in Ottawa. His research interests include object-oriented and parallel programming. He received his MS and BS in computing science from the University of Alberta in 1993 and 1989 respectively.

Introduction

Designing, implementing, and testing distributed software is considerably more difficult than for comparable sequential software. There are efficient parallel algorithms for a large class of problems, but finding the right one can be only a small part of the implementation cost. Writing distributed software is also complicated by problems not found in the sequential environment, including synchronization, deadlock, communication, fault tolerance, heterogeneous computers and operating systems, and the complexity of debugging and testing programs that may be nondeterministic due to concurrent execution.

Harnessing the power of a network of machines poses additional problems: The available processors available and their capabilities can vary from one execution to another; high communication costs can restrict the types of parallelism that can be implemented efficiently, and many developers do not want to become experts in networking or low-level communication protocols to gain the advantages of an application's potential parallelism. A few systems account for these problems, but programmers also need a system that can produce distributed software quickly, economically and reliably. The system must help developers bridge the complexity gap between distributed and sequential software without extensive training.

Our system — Enterprise — is a programming environment for designing, coding, debugging, testing, monitoring, profiling, and executing programs for distributed hardware. Developers using Enterprise do not deal with low-level programming details such as marshalling data, sending/receiving messages, and synchronization. Instead, they write their programs in C, augmented by new semantics that allow procedure calls to be executed in parallel. Enterprise automatically inserts the necessary code for communication and synchronization.

However, Enterprise does not choose the type of parallelism to apply. The developer is often the best judge of how parallelism can be exploited in a particular application, so Enterprise lets the programmer draw a diagram of the parallelism using a familiar analogy that is inherently parallel: a business organization, or enterprise, which divides large tasks into smaller tasks and allocates *assets* to perform those tasks. These assets correspond to techniques used in most large-grained parallel programs — pipelines, master/slave processes, divide-and-conquer, and so on — and the number and kinds of assets used determine the amount of parallelism. For example, an individual performs a task sequentially, but four individuals can perform four similar tasks concurrently. A department can subdivide a task among components that can then perform the subtasks concurrently. An assembly or processing line can start a second task as soon as the first component of that line has passed the first task to the second component.

Most parallel/distributed computing tools require the developer to draw communication graphs in which nodes (processes) are connected by arcs (communications paths). In Enterprise, a programmer constructs an organization chart from the top down, expanding assets to explore the application's hierarchical structure, and transforming assets from one type to another to specify how they communicate with their neighbors.

Based on the organization chart, Enterprise inserts parallel constructs and communications protocols into the code, compiles the routines, assigns processes to processors, establishes the necessary connections, launches the processes, executes the program, and (when desired) logs events for performance monitoring and debugging. Because the (sequential) code that calls the parallel procedures is independent of those procedures (although the code generated by Enterprise certainly is not), programmers can adapt applications to varying numbers and types of processors without rewriting their code. Thus, they can experiment with different kinds of parallelism, construct recurring parallel structures more easily, generate code more quickly, and reduce errors.

Enterprise supports load balancing and the dynamic distribution of work in environments with changing resources. Although it does not support arbitrarily structured parallel programs, for many applications it relieves users from the tedious details of distributed communication, letting them concentrate on algorithm development. Enterprise currently runs on a network of workstations (because those are the resources available to us), but there is nothing in the programming model that precludes its application on parallel multiprocessor architectures.

The programming model

Enterprise's model consists of two components: the programming component defines the semantics of the source code, while the metaprogramming component defines the parallelism of the program using an asset graph. Using the business organization analogy, the programming component defines the role of each individual in the organization, and the metaprogramming component defines how the individuals interact.

The programming component defines a program as a collection of interacting modules. Each module contains an *entry procedure* that other modules can call, and *internal procedures* that can be called only by other procedures in that module. This is similar to programming with encapsulation, which provides well-defined ways to manipulate data structures while hiding the implementation details from the programmer. Code executes sequentially within a module, so a sequential program consists of a single module whose entry procedure is the main program. Since modules may execute on different processors with no shared memory, no common variables are allowed.

The way in which modules interact determines a program's parallelism. A developer specifies that interaction in two ways:

- Module *calls* define a module's identity, the information passed to it, and the information it returns. These are defined in the source code (in the programming component).
- A module's *role* defines the kind of parallelism it uses when invoked. These are defined in the asset graph (in the metaprogramming component).

Although the two components are related, they can often be edited independently. A programmer can modify the metaprogram (asset graph) to specify different parallelism for the same program (source code), or modify the program to some extent without changing the metaprogram's structure.

Programming semantics

In a sequential program, procedures communicate using procedure calls: Procedure A contains a call to procedure B that includes a list of arguments. When it makes the call, A blocks (suspends execution) and B activates. When B has finished, it communicates its results to A via side effects to the arguments, or by returning a value if the procedure is a function.

Conceptually, the only difference between an Enterprise module and a sequential function or procedure is the parallelism. Module calls have the same syntax as procedure calls, but we differentiate between module calls that return a result *(f-calls)* and those that do not return a result either explicitly or by side effect *(p-calls)*. Module A does not block when it makes a p-call to module B. However, if A makes an f-call to B and tries to use the result before B has returned it, then A does block. For example, when the code

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

executes on A, the module packages the argument of B, Data, into a p-call, sends it to module B (somewhere on the network), and continues executing. A needs no result from B. This is *purely asynchronous* parallelism.

Now consider code that requires an f-call:

```
Result = B(Data);
/*some other code*/
Value = Result + 1;
```

The module packages the argument as before and sends a message on the network to B, but if A tries to access Result to calculate Value, and B has not yet completed execution, then A blocks until

B returns the Result. This concept is similar to *futures* in object-oriented programming[1]. We call this deferred synchronization a *lazy synchronous* call.

Enterprise modules accept any scalar parameters such as integers and characters, as well as structures. Arrays and pointers are also 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). We included this restriction because we cannot always statically determine the size of an array parameter. This feature lets modules pass dynamic arrays and subarrays.

Enterprise uses three macros for parameter passing, similar to those in Concert/C[2].

- In () specifies that a parameter should be sent from A to B, but not returned.
- Out () specifies a parameter, with no initial value, that B sets and returns to A.
- InOut () copies the parameter from A to B on the call, and copies its value back from B to A on the return.

A module blocks 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;
```

sends elements 60..69 of Data to B. When module B finishes executing, it copies 10 elements back to module A, overwriting Data locations 60..69. Module A will block when it accesses Data[65] if the call to module B has not yet returned.

If no parameter specification is included, Enterprise uses In as the default, making the programmer more aware of the overhead of returning data through side effects. (Sequential C programs pass arrays by reference, so InOut is their default). Since the Out and InOut parameters cause side effects, they can be considered part of a function's return value. In the above example, if module A accessed Data[65], it must block until module B returns, just as for the result of the call. Procedure calls with Out or InOut parameters actually return results, albeit by side effect, and are therefore really function 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, we can often gain more concurrency by further relaxing the synchronization of function calls. This is possible when the order in which a module returns is irrelevant. In 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];
```

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, we could obtain more concurrency by using Result[1] or Result[4] in place of Result[0], and Result[0] later in place of another result.

To increase concurrency, Enterprise lets the programmer specify whether a module is *ordered* or *unordered*. If it is unordered, its return values are consumed in the order they return, not the order in which they are referenced. For example, if the B module were unordered, then the reference to Result[0] in the second loop would refer to the first value returned by module B, even though it might not be Result[0]. Eventually all results are added and the value of Sum is the same as if the second loop had waited for the results in order. A module's order attribute (ordered or unordered) is part of its role; the user specifies it using the graphical user interface, independent of the module code.

Two other noteworthy subtleties in the programming semantics are aliasing and I/O. Currently, Enterprise does not support aliasing of returned values (the cost of detecting them may be prohibitive). That is, if there is an aliased reference to a variable that has not yet been returned, Enterprise will not block when a module refers to that alias. Enterprise also assumes the existence of a global file system (such as Network File System) and provides no special concurrency control mechanisms for file I/O.

Metaprogramming semantics

Enterprise's analogy between program structure and organizational structure eliminates inconsistent terminology (pipelines, masters, slaves, and so on) and introduces a consistent terminology based on assets. Figure 1 shows the assets Enterprise supports and their icons.

An *enterprise* represents a program and is analogous to an organization. Every enterprise asset contains a single component — an *individual* asset, by default — but a developer can transform it into a line, department, or division. When Enterprise starts, the interface shows a single enterprise asset, which the developer can name and expand to reveal the individual asset inside.

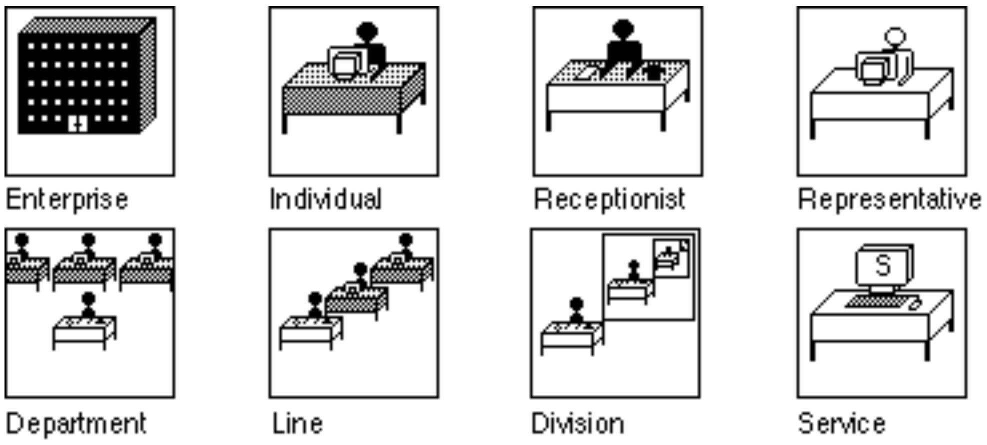


Figure 1. Enterprise asset icons.

An *individual* is analogous to a person in an organization. It does not contain any other assets; in terms of Enterprise's programming component, it represents a module that contains no other modules. An individual has source code and a unique name, which it shares with the module it represents. When an individual's module is called, it executes its sequential code to completion. Any subsequent call to that individual must wait until the previous call is finished. If a developer entered all the code for a program into a single individual, the program would execute sequentially.

A *line* is analogous to an assembly or processing line (it is usually called a *pipeline* in the literature). It contains a fixed number of heterogeneous assets in a specified order. The assets in a line are not necessarily individuals. A line could contain individuals, departments, divisions, and other lines.

Each asset in the line refines the work of the previous one and contains a call to the next. For example, a line might consist of an individual that takes an order, a department that fills it, and an individual that addresses the package and mails it. The first asset in a line is a *receptionist*. A subsequent call to the line waits only until the receptionist has finished its subtask for the previous call, not until the entire line is finished.

A *department* is analogous to a department in an organization: It contains a fixed number of heterogeneous assets and a receptionist that directs all incoming communications to the appropriate assets.

A *division* contains a hierarchical collection of identical assets among which work is distributed. Developers can use divisions to parallelize divide-and-conquer computations. When created, a division contains a receptionist and a *representative*, which represents the recursive call that the receptionist makes to the division itself. Divisions are the only recursive assets in Enterprise. Programmers can increase a division's breadth by *replicating* the representative (discussed below), or add a level to the recursion's depth by replacing the representative with a division. The new division will also contain a receptionist and a representative. The tree's breadth at any level is determined by the replication factor (breadth) of the representative or division it contains. This approach lets developers specify arbitrary fan-out at each division level.

A *service* asset is analogous to any asset in an organization that is not consumed by use and whose order of use is not significant. It contains no other assets. For example, a clock on the wall and a counter that records the total number of vehicles that have passed through several service lanes are services. Any other asset can call a service.

REPLICATING ASSETS

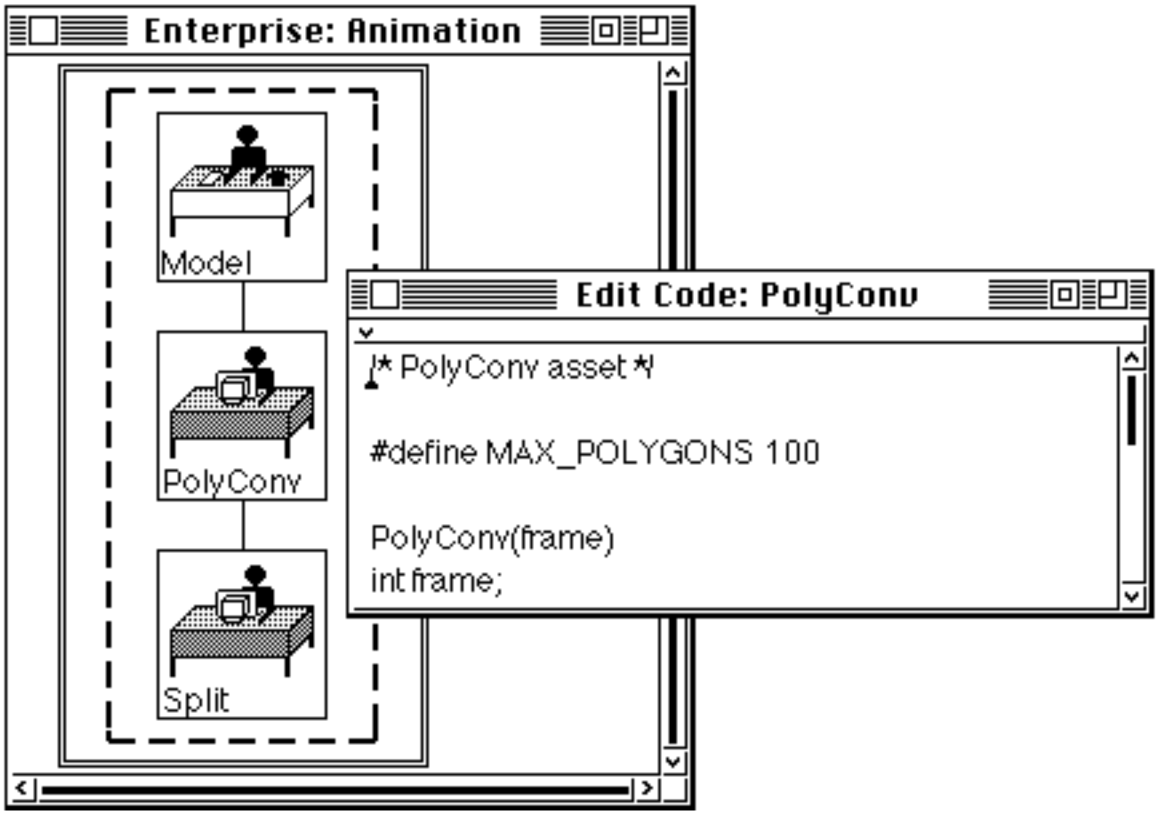
Developers can replicate assets so that more than one process can simultaneously execute the code for the same asset. The programmer specifies a minimum and maximum (possibly unbounded) replication factor, and Enterprise dynamically creates as many processes as possible, one asset per processor, up to the maximum. Users can explicitly replicate all assets except receptionists and enterprises. A receptionist cannot be explicitly replicated because it is the entry point for a composite asset (a 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, a reference to its return value will receive the first value returned from a copy of that asset.

AN EXAMPLE LINE

Let's consider an animation program that executes a loop for each frame. The procedure Model constructs a model for the frame and calls the PolyConv procedure to convert the model to polygons and render it. The last task in PolyConv is a call to Split, which does the final rendering. The program does not need to wait until PolyConv completes the first frame to start processing 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.

We could model this by transforming the default individual asset into a line of three individuals (Model, PolyConv, and Split) and entering the appropriate code for each. Since Model is the first individual in the line, it is actually a receptionist. Figure 2 shows the program design: An expanded enterprise asset (with a double-line border) contains an expanded line asset (with a dashed-line border), which in turn contains the three individuals. Figure 2 also shows a code editor for the PolyConv asset. (Enterprise uses the host windowing system. The figures here show the Macintosh implementation. Displays look slightly different in X Windows).

If we selected the Compile option in the user interface (not shown), Enterprise would insert code to handle the distributed computation, compile the program, and report any compilation errors in a window. If we then selected Run, Enterprise would allocate three processors, one for each individual in the line. In this example, we obtained a 1.7-fold speedup by using a line running on three processors instead of a single individual asset (a sequential program). All timings are subject to large variations, depending on the number of available processors and the network traffic; we did the timings on a quiet network.



```
/* PolyConv asset */

#define MAX_POLYGONS 100

PolyConv(frame)
int frame;
```

Figure 2. A parallel design for the Animation program.

Now let's see how metaprogramming lets us try alternate forms of parallelism without changing the C source code. Let's replicate the Split asset so that multiple processes can execute multiple calls concurrently. PolyConv's call to Split initiates a process; if PolyConv makes a subsequent call to Split before the first call is done, a second process is initiated (if there is an available machine). Replication is dynamic, using as many processors as are available on the network, subject to the user specified lower and upper bounds.

Replicating Split using eight processors results in as much as a 5.7-fold speed-up compared with the sequential program, depending on network conditions when the program is run. Replicating PolyConv resulted in a 6.0-fold speedup (using additional processors), which implies that Split is the real bottleneck in the program. That is, an individual PolyConv can almost keep up with its calls from Model, but an individual Split cannot keep up with its calls from PolyConv.

Finally, the C code for the Split asset contains two local procedure calls, HiddenSurface and AntiAlias. We could transform the Split asset into a line that contains one local procedure call to HiddenSurface, which in turn contains an asset call to AntiAlias. We would have to move the code for AntiAlias from the Split asset to the AntiAlias asset, but we would not need to edit it. Figure 3 shows this modified design, with the Split asset replicated. In fact, this modification did not improve performance because of the small granularity of the operations. However, the fact that such modifications can be made quickly in Enterprise encourages users to tune for performance by experimenting with many types of parallelism.

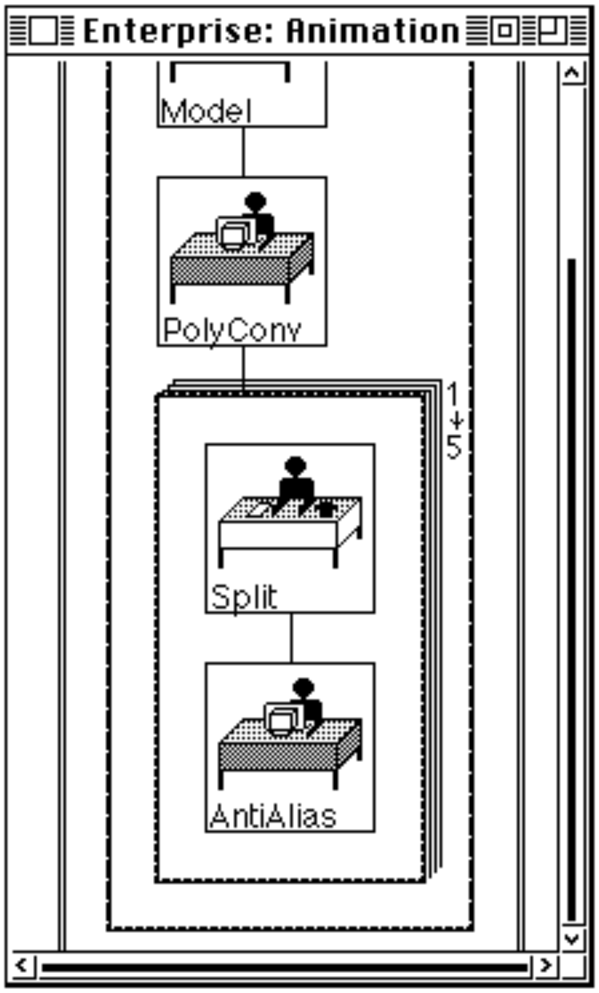


Figure 3. Animation program.

AN EXAMPLE DEPARTMENT

Now consider code that solves a series of linear equations represented by matrices and writes the solution to disk. Depending on the property of the matrices, one of three procedures is selected:

```
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 );
   }
}
```

Here are three Enterprise solutions to this problem:

(1) We can enter the code as an individual, SolveLinear, with local procedure calls. Enterprise compiles this and runs it sequentially.

(2) We can transform SolveLinear into a department asset containing the individuals SolveSparse, SolveTri, and SolveGauss. When a call is made to one of these procedures, say SolveSparse, execution continues. Since the Solve routines do not have return values or Out/InOut parameters, they are p-calls and do not block when there is a subsequent reference to any of their parameters. In the next loop iteration, if SolveTri is called, it executes concurrently with SolveSparse. However, if a subsequent iteration calls SolveSparse again, the second message waits until the outstanding call has completed.

(3) To reduce the time spent waiting, we can replicate SolveSparse, SolveTri, and SolveGauss with no maximum replication factor, then Enterprise assigns all the available processors dynamically to maximize the concurrency.

Whether this code is executed as an individual, a department of individuals, or a department of replicated individuals, the developer changes only the diagram, not the source code.

The current implementation

Enterprise's architecture consists of a graphical user interface, a precompiler, and an execution manager. The interface lets the programmer create asset graphs and enter source code. We implemented it in Smalltalk-80 to run as a Unix process on workstations[3]. The interface produces a text-file representation of the asset graph that is then used by the precompiler and the execution manager. When the user selects the Compile and Run operations, the precompiler and execution manager start as Unix processes from within the user interface.

The precompiler is based on the Gnu C compiler gcc. It first reads the graph file and then makes two passes through the source code to find the asset calls and the futures, thereby determining where to insert communication and synchronization system calls. The precompiler checks for consistency between the aseset calls in the user code and the graph file, and it reports any inconsistencies to the user. A conventional C compiler then does the final compilation.

When Enterprise executes an application, the execution manager creates a process for each asset, establishes the communication links between them, and then monitors the system, detecting idle and busy machines. For replicated assets, it allocates one process, called an *asset manager*, to manage all calls to that asset, and allocates one process for each replica. The asset manager routes calls to idle replicas and queues calls when all replicas are busy. It also maintains contact with the execution manager to take advantage of new processors as they become available.

The Enterprise runtime library can use any one of three message-passing kernels for its low-level communications: Isis,[4] the Parallel Virtual Machine (PVM),[5] and the Network Multiprocessor (NMP).[6] Each provides a high-level set of function calls to handle process creation and termination, communication, and synchronization. In addition, Isis provides a limited degree of fault tolerance and PVM provides support for a network of heterogeneous computers. NMP is a "friendly" interface to sockets that provides high performance. However, none of these kernels provides *all* the features we would like for Enterprise.[7].

A smaller user community is testing an alpha version of Enterprise on a homogeneous network of Sun workstations. Enterprise currently supports hardware heterogeneity only to the extent provided by the underlying message-passing kernel, such as PVM. The alpha version lets users animate computations (in replay mode) and inspect message contents for performance monitoring and debugging.[3]

AN EXAMPLE DIVISION

Let's see what this implementation does with the sequential code for a Mergesort routine:

```
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 );
      }
}
```

The algorithm's divide-and-conquer nature makes a division the obvious asset choice. 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, we must alter the sequential code to make this InOut relationship explicit:

```
Mergesort( &Data[0]  , InOut(Mid) );
Mergesort( &Data[Mid], InOut(Size - Mid) );
```

We can take the individual asset Mergesort and convert it into a division. For this example, the obvious choice for the breadth is 2 (but we can choose any size). This choice would result in three Mergesort processes — a parent and two children. If the sort is large, we might transform each child from an individual to a division as well. Figure 4 shows such a division of divisions, each of breadth 2. (When the minimum and maximum replication factors are the same, Enterprise displays the common value).

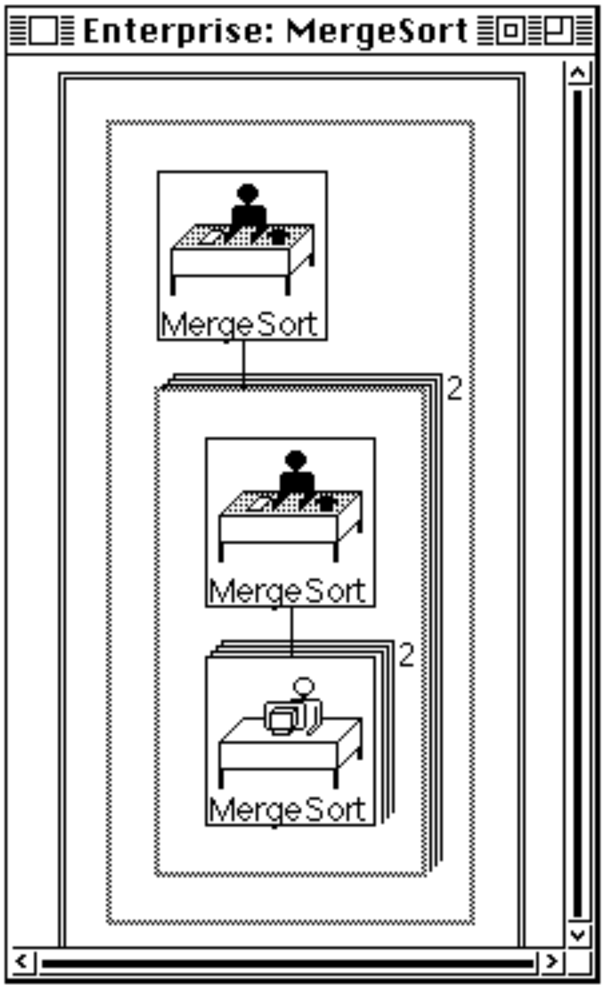


Figure 4. Mergesort program.

The first call to Mergesort divides the list in half and sends each part to a different process. The processes in turn divide the list and pass the parts to their children. These processes have no children, so the subsequent recursive calls are done sequentially.

In a division, Enterprise inserts code that permits both parallel and sequential recursive calls. Thus, we can change the division's breadth and depth without changing the asset code. Figure 5 shows what Enterprise does to the sequential Mergesort code to make it run in parallel. The code inserted by Enterprise is bold; we have edited it for brevity.

```
char * _ENT_Mergesort( msg )
   char * msg;
   {
         int * data, size;

         /* Get the parameters to the function from a message */
         _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( sSize > 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_RETURNVALUE );
               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_RETURNVALUE );
               else Mergesort( &Data[Mid], InOut(Size - Mid) );

               /* Future: wait for InOut parameter to return */
               if( _ENT_Child != NULL ) {
                     _ENT_Wait( &Data[0], Mid );
                     _ENT_Wait( &Data[Mid], Size - Mid );
               }
               Merge( Data, Size );
         }
   }
```

Figure 5. *Enterprise* code for Mergesort.

Figure 6 shows the processes that Enterprise creates for this example and their interconnections. All processes are connected to the execution manager (which coordinates the activities on the machines) and the monitor manager (which reports to a graphical display so the user can view the computation's progress). In addition, each division's asset manager keeps track of which processes are busy or idle, and forwards work to the first available asset. For this example, a more efficient implementation would bypass the managers and hard code the connections, but this would reduce the system's flexibility. The managers support an arbitrary number of assets, so a developer can change the replication factor or specify unlimited breadth.

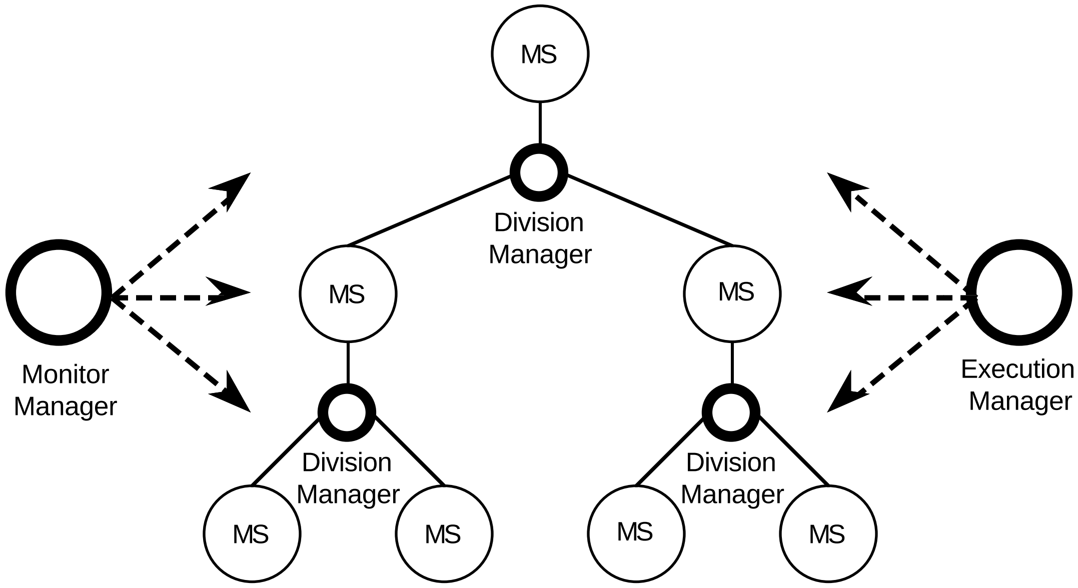


Figure 6. Process diagram for Mergesort (MS) example.

Programming in Enterprise

We have implemented several nontrivial applications using Enterprise.[7] Let's compare Enterprise results with programs hand coded in PVM and Concert/C. Since Enterprise can be used with several communication packages, we'll note which one we used to obtain the results — for example, Enterprise (PVM) uses the PVM kernel. Although we concentrate here on performance, Enterprise compares favorably to other parallel programming systems based on other measures as well.[8]

GAUSS-SEIDEL

Gauss-Seidel is an iterative algorithm for solving families of linear equations. Given the equation $Ax = b$, where A is an $N \times N$ matrix and x and b are $N \times 1$ vectors, this algorithm solves for x. It starts with an initial value for x and then iterates, refining the solution vector until it converges. The parallel algorithm for finding x can operate asynchronously, with each processor responsible for a portion of the *x* vector, assuming shared memory to simplify result sharing. The algorithm benefits from the lack of processor synchronization, but yields a nondeterministic solution.

We implemented the parallel Gauss-Seidel algorithm as a line of two — a receptionist and an individual — and we replicated the individual. A service asset simulated shared memory. Messages sent between Enterprise processes ranged from 2 to 16 Kbytes, depending on the problem size. For a 2,000 x 2,000 matrix on 10 processors, Enterprise (NMP or PVM) gave us a 7.6-fold speedup, while Enterprise (Isis) achieved a 4.8-fold speedup. A hand-coded PVM version achieved a 7.7-fold speedup, while a hand-coded Concert/C version achieved a 4.4-fold speedup. This application is completely asynchronous, so we expected linear speedups. Implementing and debugging the hand-coded PVM and Concert/C versions took several days. Creating the Enterprise program from the sequential code took 20 minutes.

ALPHA-BETA TREE SEARCH

We also used Enterprise to implement an algorithm that builds a search tree in a recursive, depth-first manner to a prescribed depth. A user-defined distribution assigned random values to leaf nodes. The parallel version uses the *principal variation splitting algorithm*, which recursively descends the leftmost branch of the tree, searching the siblings in parallel while backing up the result. As with Gauss-Seidel, Enterprise implements this algorithm as a line of two assets, with the second asset replicated.

This algorithm has synchronization points that limit the potential speedup. It is computationally intensive and generates few messages in relation to the CPU costs. Enterprise (NMP or PVM) achieved better speedup than Enterprise (Isis) or hand-coded PVM and Concert/C: 4.4-fold (matching the expected values). The hand-coded PVM speedup was only two-thirds that of Enterprise (NMP or PVM), due to Enterprise's superior work-allocation scheme. Changing the default work distribution scheme would improve the performance of the hand-coded-coded PVM version, but at the expense of several hours of programming.

ADJUSTMENTS TO THE SEQUENTIAL CODE

Some Enterprise applications required a few extra lines of sequential code in three places (the hand-coded solutions required significantly more). However, some programs may require more extensive modifications, depending on their data structures and control-flow models.

Additional asset calls

Since concurrency cannot occur unless there is more than one call to an asset, we sometimes must 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 an application that implements transitive closure, one routine accounts for 99 percent of the execution time. There is only one call to this routine:

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

TC takes two index parameters and iterates from the start to the end index. We transformed this call into a loop containing multiple calls:

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

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

Global variables

Since Enterprise does not support shared variables (except through services), it must pass some values as parameters in asset calls that may be declared as global variables in sequential programs. For example, the sequential version of the alpha-beta search program uses five global variables that contain the state of the search tree. To adapt this program to Enterprise, we placed these variables in a structure that is passed as a parameter to each asset call. The structure size was about 2 Kbytes. Because of the high computation-to-communication ratio, this did not hurt parallel performance.

Repeated asset calls

Repeated calls to an asset must return their values to different variables to achieve maximum parallelism. In the following code, only one copy of asset A executes at a time because of the dependency on the future Sum:

```
Sum = 0;
for( i = 0; i < Size; i++ )
   Sum += A( i );
```

Changing the code so that each call to *A* returns its result to a different future gives us the maximum parallelism:

```
for( i = 0; i < Size; i++ )
   s[i] = A( i );
Sum = 0;
for( i = 0; i < size; i++ )
   Sum += s[i];
```

Limitations

Enterprise does not limit the number of processors that an application can use, but our tests have not involved more than 40. An application's scalability depends on the number and granularity of the assets and any limitations imposed by the communications package. It is not clear yet whether Enterprise is suitable for massively parallel applications.

The Enterprise model also has several limitations at both the programming and the metaprogramming level.

PROGRAMMING LIMITATIONS

Enterprise does not support shared variables. This may be a major problem for existing sequential programs, since global variables are common in legacy code. Enterprise's lack of support for shared variables is historical:[9] It was originally designed for distributed computing applications with no shared memory. We could simulate shared memory by using service assets, but the cost may be prohibitive due to communication overhead. In the Gauss-Seidel implementation, the access is infrequent enough that communication overhead is not significant.

Also, Enterprise supports the use of pointers as parameters, but it does not support the passing of pointers to data that contain additional pointers (double indirection). Again, the reason for this restriction is the lack of shared memory. If Enterprise is to be effective on a shared-memory architecture, we must create a good asset-level mechanism to remove these two coding restrictions.

METAPROGRAMMING LIMITATIONS

Enterprise does not support arbitrary graphs. For example, there can be no cycles of asset calls except in the controlled case of a division asset. However, this restriction eliminates the possibility that an application will deadlock in a parallel-recursive situation. Enterprise's division assets provide a controlled mechanism for parallel-recursive calls. If the division asset proves not general enough to solve most problems that require parallel recursion, we will develop new assets to solve the new problems

Enterprise cannot directly support applications based on data-parallel algorithms. Enterprise assets let users design the parallel control flow of an application, but implementing data-parallel algorithms may require extensive code modifications. We do not know if we can generalize existing assets as a simpler way to express data parallelism or whether we must add some new concept to the model. Our preliminary work with ordered and unordered assets suggests that adding asset attributes may be sufficient.

The use of an asset manager for replicated assets creates significant communication overhead. If a replicated asset is busy when it is called, the manager finds an idle replica and forwards the message to it. When the replica is finished, it sends a reply to the manager, which then forwards it to the original caller. For large messages, the overhead from this process is significant. We can reduce it by having the replica reply to the original caller directly. Alternately, the asset manager could avoid forwarding the message to the replica by replying to the original caller with a reference to an idle replica. The original caller would then communicate with the replica directly. The relative costs of these approaches depend on communication costs and message sizes. Regardless of which we choose for Enterprise, a programmer who knows the application requirements and parameters — and who uses a low-level message-passing system like PVM or NMP — will almost always achieve better runtime performance with a hand-coded solution. Developers will always have to ask, "Is the time saved by using Enterprise worth the performance degradation?". However, sequential developers already face this same question when they decide whether to use assembly language or a higher level language.

Finally, Enterprise does not support division assets with dynamic depths. We could resolve this problem by changing the semantics of divisions, but we need to study the issue more to determine whether this is useful in practice.

Conclusions

Many parallel-programming tools are intended for specific phases of the software life cycle. However, our goal is to build an integrated programming environment suitable for the entire life cycle. We have not yet adequately addressed execution monitoring, statement-level debugging, integration with other programming languages, or performance optimization. Nevertheless, we have taken the first steps in making users more at ease and more productive with distributed systems.

High-level programming tools such as Enterprise have been criticized for hiding many hardware details, making it easy to create inefficient programs.[10] In fact, programmers do need a good understanding of the parallelism in their applications if they want high performance from their Enterprise applications. For example, many procedures and functions that could be done in parallel should not be, because the benefits of the parallelism do not outweigh the cost of communication in a distributed domain. However, Enterprise's assets are application independent and perform well on a large cross section of coarse-grained applications. And after using Enterprise to express coarse-grained parallelism, a developer can compile individual assets using other parallel systems to take advantage of fine- or medium-grained parallelism.

ACKNOWLEDGMENTS

The Enterprise project has benefited from the efforts of many people, including Enoch Chan, Randal Kornelsen, Paul Lu, Stan Melax, Ron Meleshko, Jimmy Mohsin, Carol Smith, Pok Sze Wong and Zhonghua Yang. This research was supported in part by grants from the Central Research Fund of the University of Alberta, the Natural Sciences and Engineering Research Council of Canada, grants OGP-8173, infrastructure grant 107880, and a research grant from IBM Canada Ltd.

REFERENCES

1. R. Halstead, "MultiLisp: A Language for Concurrent Symbolic Computation", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 4, October 1985, pp. 501-538.
2. J. Auerbach et al., "Interprocess Communication in Concert/C", IBM Technical Report RC17341, IBM T.J. Watson Research Centre, Yorktown Heights, 1992.
3. G. Lobe, D. Szafron and J. Schaeffer, "The Object-Oriented Components of the Enterprise Parallel Programming Environment", *Proc. Tools USA 93*, Prentice Hall, New York, 1993, pp. 215-229.
4. K. Birman et al., "The ISIS System Manual, Version 2.1", ISIS Project, Computer Science Dept., Cornell University, 1991.
5. G.A. Geist and V.S. Sunderam, "Network-Based Concurrent Computing on the PVM System", *Concurrency Practice and Experience*, Vol. 4, No. 4, 1992, pp. 293-311.
6. T.A. Marsland, T. Breitkreutz and S. Sutphen, "A Network Multi-processor for Experiments in Parallelism", *Concurrency: Practice and Experience*, Vol. 3, No. 1, 1991, pp. 203-219.
7. I. Parsons, "Evaluation of Distributed Communication Systems", *CASCON '93*, IBM Toronto, 1993 (to appear). Available via anonymous ftp from menaik.cs.ualberta.ca.
8. G.V. Wilson, J. Schaeffer, and D. Szafron, "Enterprise in Context: Assessing the Usability of Parallel Programming Environments," Proc. Cascon '93, IBM Canada, Toronto, 1993 (to appear). Available via anonymous FTP from menaik.cs.ualberta.ca.
9. D. Szafron et. al., "The Enterprise Distributed Programming Model", *Programming Environments for Parallel Computing*, N. Topham, R. Ibbett and T. Bemmerl, editors, Elsevier Science Publishers, 1992, pp. 67-76.
10. A. Jones and A. Schwarz, "Experience Using Multiprocessor Systems - A Status Report", *Computing Surveys*, Vol. 12, No. 3, 1980, pp. 121-166.