Software Engineering Considerations in the Construction of Parallel Programs*

Jonathan Schaeffer and Duane Szafron

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

In many papers describing parallel programming tools, the authors illustrate the strengths of their approach by presenting some impressive speedup results. However, is this the only metric by which we should judge the quality of their tool? Many of these tools offer significant software engineering advantages that reduce program development time and increase code reliability. This paper uses the Enterprise programming environment for coarse-grained parallel applications to illustrate the advantages of these tools. For most users, high performance is not an important evaluation criteria; other criteria, such as tool usability and program development savings, are often far more important.

1. INTRODUCTION

All too often in the literature, the only evaluation metric given for the implementation of a parallel algorithm is the ubiquitous speedup. Speedup only measures the benefits of parallelization at execution time, and completely ignores the software development investment required to achieve this result. Parallel programming adds an extra dimension of complexity to software design, development, testing and debugging. This is a real cost that must be factored into any decision to allocate programming resources to building parallel solutions. A better measure of the effectiveness of a parallel implementation might be the *Utility*† :

Utility = F(S, SD, PD, I)

where

S:	execution speedup achieved,
SD:	total sequential program development time of the program,
PD:	total parallel program development time,
I:	importance of achieving the speedup, and
F	a function that maps its arguments into a cost measure.

Utility augments the speedup metric by considering the importance of achieving a good speedup and the development resources required to achieve this. For example, code which is run frequently or for long periods of time, has a higher *I* value than programs that are run once.

---

* This research has been funded in part by NSERC grants OGP-8173 and OGP-8191, a grant from IBM Canada Limited and the Netherlands Organization for Scientific Research (NWO).

† A modified definition from a personal communication with Greg Wilson.

This is a preprint of a copyrighted article that will appear in a forthcoming book.

Essentially, *Utility* is a cost-benefit measure, reflecting the cost of human and computer resources, versus the benefits of a faster program.

In the frequently occurring case of legacy code, where the sequential code already exists, SD is effectively 0, and PD is the additional effort required to convert the program to run in parallel. Consider a sequential legacy program that executes for a month when it is run. A programmer may convert this code to run on a network of workstations using a low-level tool (such as PVM [11]). It may take one month of programmer time to create a parallel version of the program that now takes only a day to execute. Another programmer may tackle the same problem using a high-level tool (such as Enterprise [19]). What if his solution takes only a week to develop, but the program takes 1.5 days to execute? Which is the better solution? Programmer time costs real dollars; compute cycles are often free or inexpensive. We argue that a high-level parallel programming tool allows the user to develop solutions quickly, freeing up programmer resources for other tasks. In the case of Enterprise, *if* the execution time is not acceptable, you can still use the tool to generate a correct parallel program, and then edit the generated code to provide your own application-specific enhancements.

This paper discusses the software engineering advantages of using the high-level parallel programming tool Enterprise for developing applications to run on a network of workstations. Enterprise inserts code into a program to handle all the parallel considerations, reducing the programming time required to turn a sequential application into a parallel one. This allows the user to concentrate on the domain-dependent code which the user understands best. Enterprise produces a correct parallel program structure that will often achieve performance close to that of a hand-crafted solution. In other words, there is a trade-off: better software engineering and shorter development time for (possibly) slower execution performance.

There is a community that demands high performance from their applications, but this community represent a small percentage of the potential user community for parallel computing technology†. With the increased availability of relatively low-cost multi-processor machines and the proliferation of networked single-processor machines, more people will be tempted to test-drive parallel software technology. However, these people will usually not demand high performance. A parallel tool that allowed them to quickly achieve improved throughput would be welcome, even if the performance was not as high as might be achieved through significantly increased programming effort.

In the past decade, there has been significant progress in developing tools to ease the pain of programming coarse-grained parallel applications. These tools adopt a variety of approaches to add parallelism to existing languages (primarily C and Fortran) such as providing library calls (e.g. PVM [11] and Linda [6]), loop parallelism (e.g. Myrias PAMS [14]), extending a language (e.g. HPF [16]) and expressing the parallelism graphically (e.g. Hence [4] and Enterprise [19]). More radical approaches include defining a new procedural language that supports parallelism (e.g. Orca [3]) or alternative programming paradigms (such as a functional language like Sisal [9]). However, given the large investment in C and Fortran software, we see these latter approaches as not being practical in the short-term. All of these approaches, referred to as parallel programming systems (PPS), offer software engineering advantages to the software developer. This paper discusses Enterprise because it offers us greater scope for illustrating many of the potential advantages of using higher-level tools. Enterprise is more than just a parallel programming model, it is also a software development environment.

This paper is not an introduction to Enterprise (see [19]), nor is it intended to be a sales presentation. Instead we concentrate on some of the major advantages of using a high-level parallel programming model and a complete integrated parallel programming environment. These advantages are available in a variety of tools and, rather than turn this paper into a literature survey, we concentrate on one representative system, Enterprise, and give a few sample references to other systems. We argue that from the software engineering point of view, high-level parallel programming tools can considerably reduce program development

---

† However, they do represent a large percentage of the parallel cycles used in the world.

time and increase the reliability of the resulting code. In many cases, these advantages far outweigh the benefits obtained from a faster hand-coded solution.

2. PROGRAMMING: COMMUNICATION

Multiple agents, all working together towards a common goal, must have a means of communication. In distributed systems, the method of communication is message passing. The traditional programmer's view of message passing involves four steps:

1. pack the data,
2. send the message,
3. receive the message and
4. unpack the data.

In some systems, such as PVM, these steps must be explicitly programmed in the user's code. Alternative systems, such as Sun RPC [22], only require the user to provide packing and unpacking routines. Both of these low-level approaches require the user to write additional lines of code that only increase the probability of introducing an error. In contrast, Linda handles the data transparently, but expects the programmer to explicitly call routines to do the communication [6]. Hence [4] allows the user to express this information graphically, but parameter information must be specified both in the interface and the code (causing a redundancy problem, see Section 6). Concert/C eliminates the packing/unpacking routines by having the user provide additional type information describing the data [12]. In all these cases, the higher-level tools can significantly reduce the programming effort.

One approach to message passing is to make this form of communication look like procedure calls that happen to be executed remotely (but without the programming effort and synchronous semantics of RPC). Ideally, the user would write the code without any knowledge of whether the procedure was to be executed in parallel or sequentially. In practice, this is not possible with a distributed memory; there are no globally shared variables unless additional facilities are added to the language (Orca is an example of a language that supports shared objects in a distributed environment [3]). However, modern software engineering practices limit the use of global variables anyway and procedures that use them can be rewritten using extra parameters.

For example, let's assume the procedure `Callee()` is to be executed in parallel with `Caller()` (a technique for specifying this is discussed in Section 4):

```
Caller()
{
   char a;
   USER_TYPE b;
   double d[100];
   int dnumb;
   int result[100];
   ...
   result[5] = Callee( a, b, &d[10], dnumb );
   ...
}

int Callee( char a, USER_TYPE b, double * d, int dnumb )
{
   ...
}
```

Since Enterprise includes a pre-compiler, we have access to symbol table information and can determine the size of data. Hence, Enterprise can automatically generate the code to marshal the parameters of `Callee()` into a message, rather than forcing the user to do this explicitly. In effect, the call to `Callee()` looks like sequential code, but may in fact be executed remotely. The consequence of this approach is that parallel code looks like ordinary sequential code.

Note that because of weak typing in C, it is not possible for the compiler to figure out all data typing information at compile-time. An example is the `d` pointer above. Although in this example, the compiler can compute the number of elements pointed to by `&d[10]` (100 - 10 = 90 elements), in general it may not be able to compute the number of elements since a pointer can be coerced to point to anything. Enterprise solves this problem by requiring each pointer parameter to be followed by an additional parameter that specifies the number of elements to pass (`dnumb` in the example). This approach is analogous to Fortran code, where the user must pass an array and the size of the array as parameters. Concert/C overcomes many of the C type ambiguity problems by having the user provide additional type information describing the data [12]. If a PPS wants to handle passing an arbitrary C data structure between processes, there appears to be no alternative than to follow the Concert/C approach.

By letting the compiler generate the code to do data packing and unpacking, the probability of programming errors is greatly reduced.

3. PROGRAMMING: SYNCHRONIZATION

There are a variety of ways that PPSs can introduce parallelism. A common method is to use compiler technology to automatically detect loops that can be executed in parallel. While this has been successful for detecting fine-grained parallelism, it seems to have limited utility for coarse-grained applications. Alternatively, some systems use a keyword, such as *pardo* [14], to explicitly tell the system which loop iterations (or which sections of independent code) should be executed in parallel. Both methods require users to reorganize code to make loop iterations independent of each other. However, these approaches have the advantage that the PPS does most of the work. In contrast, using library calls to introduce parallelism puts the onus solely on the shoulders of the programmer (PVM and a variety of *fork/join* models, such as [18]).

In Enterprise, designated functions can be executed in parallel. This means that users express parallelism by the way they write their functions, rather than the way they reorganize their loops. However, if `Callee()` is designated as a parallel call in the `Caller()` statement

```
result[5] = Callee( a, b, &d[10], dnumb );
```

it appears to have no concurrency. Even though `Callee()` may be invoked in parallel, it appears that `Caller()` must wait for the return value to set `result[5]` correctly. In Enterprise, *futures* are used to increase concurrency during parallel calls [13]. Enterprise relaxes the semantics of a parallel call to allow the calling routine to continue executing concurrently with the call, until it needs the result. For example, in this code fragment:

```
sum = 0;                                                (1)
for( i = 0; i < 100; i++ )                              (2)
   result[i] = Callee( a, b, &d[i], 1 );                (3)
for( i = 0; i < 100; i++ )                              (4)
   sum = sum + result[i];                               (5)
```

`Caller()` is allowed to continue executing after each call to `Callee()` on line (3) and does not block until at least line (5). On line (5), when `result[i]` is accessed, `Caller()` blocks only if `Callee()` has not yet returned with the new value of `result[i]`. Thus, the semantics of sequential C are preserved. When pointers are used as arguments to parallel calls for the purpose of side effects, Enterprise treats them as return values. Enterprise provides optional macros for improving execution efficiency to specify that the data passed to a routine need not actually be copied back.

Futures are the synchronization tool of Enterprise. `Caller()` continues to execute and only blocks when it accesses a result that has not yet returned. This allows the user to increase concurrency by moving code around to keep `Caller()` busy doing other things until `Callee()` returns. With this model, the programmer does not need to specify any explicit synchronization point in their code. However, users have the responsibility of organizing their code into parallel functions, and generating enough futures to keep the computational resources busy.

A second technique is used to increase concurrency. In a situation where a series of calls are made to the same parallel function and the order of the results is not critical, the function may be declared to be *unordered*. This attribute is set in the user interface and is independent of the user's code. Unordered semantics means that the first reference to a particular return value for an asset will receive the first corresponding parameter value returned, regardless of whether the variable name matches or not. The second value returned will be placed in the second reference, etc. For example, in the previous code fragment, when statement (5) is executed, the program 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 can be obtained by using `result[1]` or `result[4]` in place of `result[0]` and using `result[0]` later in place of another result. It is important to realize that the unordered option violates the semantics of sequential C, but does allow additional concurrency in some computations.

In Enterprise, great care has been taken to avoid explicit references to parallelism in the code. The goal is that the user should write code that will execute sequentially or in parallel, with no changes. For example, consider the parallel divide-and-conquer code of Figure 1. This code multiplies the coefficients of two polynomials together to produce the coefficients of the product. If `Mult()` is to be executed recursively in parallel, then in many systems the user must specify some condition for switching between parallel and sequential recursion, such as:

```
if( N >= threshold )
       Parallel_Mult( ... );
else Sequential_Mult( ... );
```

Defining `threshold` can be tricky since it is a function of the hardware used and must be changed when the execution environment changes. In Enterprise, each executable binary is capable of executing any parallel routine sequentially. Currently, the depth of the parallel recursion (the point where the code switches between parallel and sequential execution) is expressed graphically and is independent of the code generated (see next section). However, there is on-going research on having the system dynamically decide when to switch between parallel and sequential execution, based on run-time statistics. Again, the goal is to reduce the onus on the programmer.

```
/* Multiply two polynomials together, with coefficients in Pointer1 and */
/* Pointer2 arrays, and put the product coefficients in the Answer-     */
/* Pointer array.                                                       */

Mult( Pointer1, Pointer2, N, AnswerPointer )
{
      localvars Result1, Result2, Result3, Cross1, Cross2;

      if( N == 1 ) {
            AnswerPointer[0] = Pointer1[0] * Pointer2[0];
      } else {
            /* Multiply the low and high order terms */
            Mult(  Pointer1,       Pointer2,      N/2, Result1 );
            Mult( &Pointer1[N/2], &Pointer2[N/2], N/2, Result2 );

            /* Low and high crossover terms */
            Cross1 = CrossOverTerms( Pointer1, Pointer2, N/2 );
            Cross2 = CrossOverTerms( Pointer2, Pointer1, N/2 );
            Mult( Cross1, Cross2, N/2, Result3 );

            /* Sequentially combine results to give the answer */
            Combine( Result1, Result2, Result3, N, AnswerPointer );
      }
      return;
}
```

Figure 1. Pseudo-code for polynomial multiplication [20].

4. META-PROGRAMMING: ASSETS

There are several parallelization structures that are used frequently in writing code for coarse-grained parallel applications. Examples include pipelines, master-slave structures and parallel divide-and-conquer. There are a number of tools that provide easy generation of these structures (for example, PIE [21] and P3L [2]) that can significantly reduce the program development time for applications well-suited to these forms of parallelism.

Although these structures have been developed and named in an *ad-hoc* fashion, each can be considered a high-level specification for a parallelization mechanism. However, two fundamental concepts are required to translate these unrelated parallelization mechanisms into a suite of powerful software engineering components. The first requirement is a single unifying vehicle that will allow each of these disparate techniques to be documented, explained and applied in a consistent manner. This would allow novices to learn a second or third technique more quickly after the first technique is mastered. It would also allow different experts to communicate results more easily with fewer misunderstandings. Using a single vehicle to describe entities that appear different is not a new idea, just a powerful one. For example, BNF is used as a single mechanism to describe the syntax of a variety of programming languages.

It is not enough to develop a notation that regularly describes these parallelization techniques, but only allows them to be used in isolation from each other. The second requirement is some high-level mechanism that can be used to combine these techniques so they can be used as building blocks in the same application. For example, a flowchart not only provides a vehicle for representing different control structures, it also allows a variety of control structures to be combined in the same flow control diagram. Ideally however, this combining mechanism should allow the parallelization techniques to be combined in a hierarchical fashion, not just a linear one. Perhaps our ideal vehicle is closer to the data-flow diagrams that are used in software engineering to describe a multitude of different software structures that can be combined in a hierarchical fashion.

Whereas other parallel tools stick with the traditional names for parallelization techniques, Enterprise uses an analogical metaphor to serve as our vehicle for documenting, explaining, applying and combining these different parallelization techniques at a high level. In fact, a major goal of the Enterprise PPS is to validate our metaphor. Business organizations are working systems that exploit parallelism. Every day, businesses receive orders and distribute tasks to different divisions, departments, assembly lines and individuals who work concurrently to fill these orders. We have chosen the structure of a business organization as our metaphor for parallel computation. We use the generic term *asset* to represent a resource that can be used to complete a task, and define a suite of assets that represent our high-level parallelization techniques by business units. Enterprise uses this metaphor both to represent parallelization techniques and to combine them in programs. Although experienced parallel programmers might scoff at the analogy, our experience shows that it is easy to learn, and is of significant benefit for explaining parallelism to sequential programmers [23]. Parallel programming still has a steep learning curve and anything that can reduce programmer startup time is beneficial.

4.1. Parallelization Techniques

Enterprise currently supports the assets whose icons are given in Figure 2. Each asset has a name and associated code, consisting of a function with the same name as that of the asset. Using the business analogy, each asset represents a named person and a definition of its role in the organization (or enterprise). The user can expand composite assets to see their underlying structure and use them to build hierarchical structures. It is also possible for the user to replicate an asset so that multiple calls to that asset do not have to wait until the first replica has finished. The mechanics of transforming, expanding and replicating assets are discussed in Section 4.2. All assets have unique names that can appear only once in the diagram, meaning that cyclic calls (possibly introducing deadlock) are not possible (such as in Hence [4]).

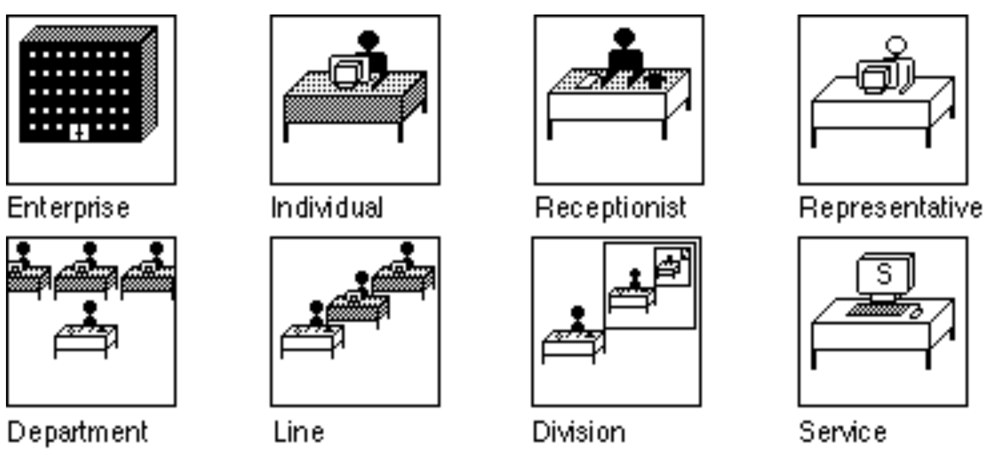

Figure 2. The icons for the Enterprise assets.

Enterprise: it represents a program and is analogous to an entire business organization. Every enterprise asset contains a single component.

Individual: it represents a slave in traditional parallel terminology and 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 procedure that executes sequentially. An individual has source code and a unique name. When an individual 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.

Line: it represents a pipeline in traditional parallel terminology and is analogous to an assembly or processing line. It contains a fixed number of heterogeneous assets in a specified order. The assets in a line need not be individuals; they can be any legal combination of Enterprise assets. Each asset in the line refines the work of the previous one and contains a call to the next. The first asset in a line is the *receptionist.* A subsequent call to the line waits only until the receptionist has finished its task for the previous call, not until the entire line is finished.

For example, consider a graphical animation program. For each frame it creates objects, converts the objects to polygons and renders the frame. Enterprise could represent this program using a line asset that contains three individual assets (*CreateObjects*, *Polygon* and *Render*) that get called for each frame (i.e. three functions, with *CreateObjects* containing a call to *PolyGon*, and *PolyGon* containing a call to *Render*).

Department: it represents a master/slave relationship in traditional parallel terminology and is analogous to a department in an organization. It contains a fixed number of heterogeneous assets and a receptionist that directs each incoming communication to the appropriate asset. All assets execute in parallel.

For example, consider a program that solves a series of sets of linear equations. There are many different techniques for solving sets of equations, depending on the properties of the coefficient matrix. The program repeatedly reads a matrix, selects a technique and calls a solver that implements that technique. In Enterprise, this program would appear as a department whose receptionist reads each matrix and calls the appropriate individual asset to solve the equations.

Division: it represents a divide-and-conquer computation and contains a hierarchical collection of individual assets among which the work is distributed. When created, a division contains a receptionist and a *representative* that represents a leaf node. Divisions are the only recursive asset in Enterprise. Programmers can increase a division's breadth by replicating the representative. The depth of recursion can be increased one level at a time by transforming the representative (leaf node) into a division. This approach lets developers specify arbitrary fan-out at each level.

For example, consider the polynomial multiply program shown in Figure 1. A call to `Mult()` divides each polynomial into three pieces (lower, upper and cross-over) and makes recursive calls on each. By making `Mult()` a division, each recursive call can be done in parallel. This parallel recursion continues until the lowermost division asset is reached. Further recursive calls are executed sequentially so a division is a combination of parallel and sequential recursive calls.

Service: it represents a monitor and is analogous to any asset in an organization that is not consumed by use and whose order of use is not important. It cannot contain or call any other assets, but any asset can call it. A wall clock is an example of a service. Anyone can query it to find the time, and the order of access is not important.

For example, consider a program that uses shared memory. This can be simulated using a service asset that accepts two types of calls: one to set the value in shared memory and the other to retrieve the value. Any other asset can call this service to set/get the value. Alternately, a service could be used to perform a more complicated function. For example, a service could be used to queue requests to a shared resource, such as a printer, guaranteeing mutual exclusion for the requests.

4.2. Meta-Programming: Combining Parallelization Techniques

Each of the Enterprise assets can be regarded as a template, or algorithmic skeleton [8], for parallelizing a section of sequential source code. However, a mechanism must be provided to combine these assets into a working program. Some PPS's represent each process with a node and require the user to "connect-the-dots" to define communication paths ([1] and [4], for example). Although this approach produces a (possibly complex) diagram of the inter-process communication structure of a program, it does little to reveal the high-level parallelization techniques and algorithms that are being used. At the other extreme, some systems have templates defined for a class of problems, hiding most of the parallel implementation details. This requires little interaction on the user's part since everything is pre-packaged. These tools, such as PUL [7], can support powerful high-level structures, but with restricted domain applicability.

Enterprise provides a small set of building blocks from which the user can construct arbitrarily complex programs using a simple mechanism. In Enterprise, the user begins by representing a program as a single enterprise asset containing a single individual. This "one person business" represents a sequential program. Four basic operations are used to transform this sequential program into a parallel one: asset expansion, asset transformation, asset addition and asset replication. Using the analogy, the simple business grows into a (possibly complex) organization.

The initial enterprise asset can be expanded to reveal its internal structure; a single individual. The individual asset can then be transformed into a composite asset like a department, line or division and the composite asset can be expanded to reveal its default components. Component assets can be added to lines or departments. If there are more calls to an asset than it can handle in a reasonable time, the asset can be replicated to produce multiple identical copies. If a call to a replicated asset has not returned by the time a subsequent call is made to that asset, one of the replicas transparently handles the call. Finally, component assets at any level can be replicated and expanded so a program can consist of a hierarchy of assets to an arbitrary level.

For example, consider a program that consists of a department asset that contains a single receptionist, `DEPT()`, and calls to three individual component assets, say `A()`, `B()` and `C()`, in a loop. Assume that `A()` is not compute bound, but that assets `B()` and `C()` are. If `B()` has an internal structure that does not lend itself to further parallelization, it can still be replicated so that multiple calls to it can execute concurrently. Assume that `C()` has an internal structure that could be represented by a line. `C()` can then be transformed into a line asset with receptionist (also called `C()`) and additional components `D()` and `E()`. All three assets (`C()`, `D()` and `E()`) can then execute concurrently. If the line `C()` is still compute bound, it can also be replicated. Multiple copies of it (and all of its components) can then execute concurrently. Figure 3 illustrates the structure of this program as it would appear in Enterprise. Inside the double-line rectangle is the expansion of the enterprise asset. Inside the solid-line rectangle is the expansion of the department `DEPT()`. Asset `B()` has been replicated three times and asset `C()` two times. Inside the dashed-line rectangle is the expansion of the line `C()`, containing the receptionist `C()` and individuals `D()` and `E()`. In theory, there is no limit to the complexity of a parallel program. In practice, we have never had to build a diagram as complex as this one for any application.

An important point illustrated with this diagram is the separation of sequential code from parallel specification (Section 5). The user can change this diagram, for example by modifying a replication factor or making `C()` an individual instead of a line, *without having to change the code*.

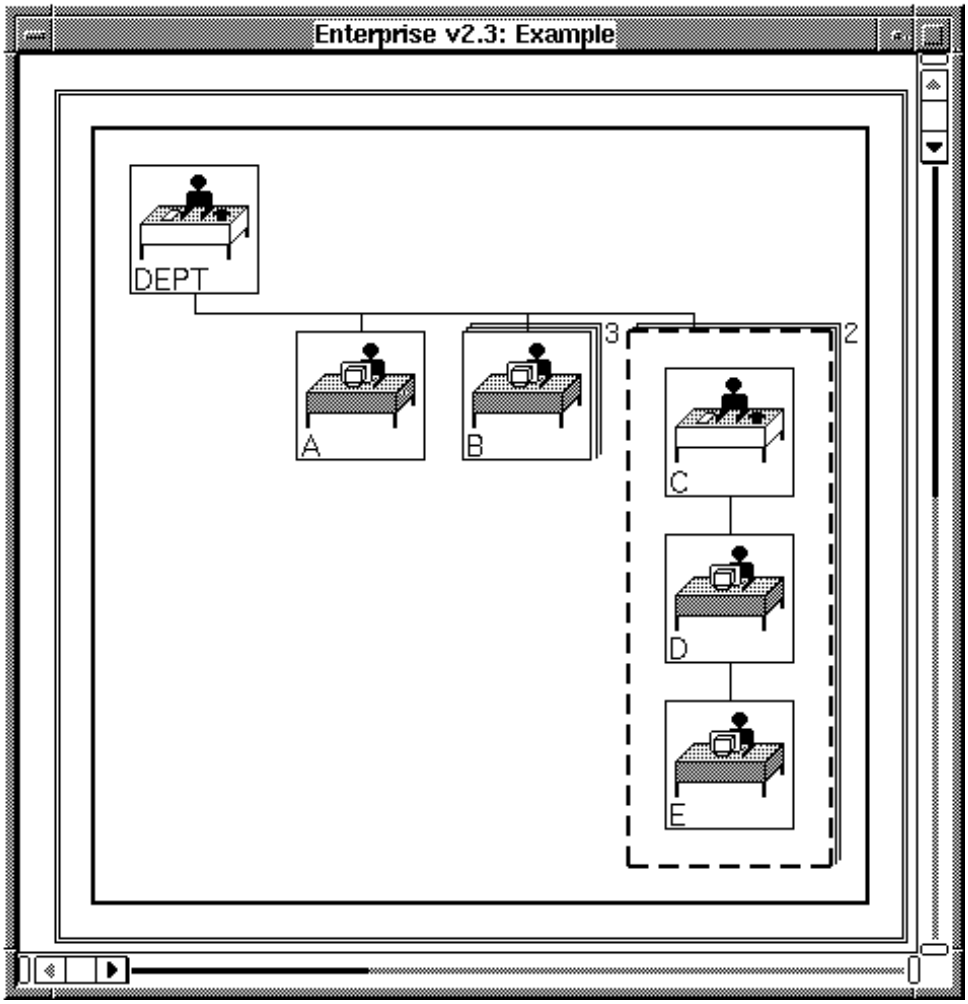

Figure 3. The hierarchical representation of a complex program.

Consider the parallel polynomial multiplication program of Figure 1. The program can be described as a line of two assets. `PolyMult()` reads in the coefficients of the polynomials to be multiplied. It then calls `Mult()` to recursively do the multiplication. Figure 4 shows the parallel structure of the program. Inside the double-line rectangle is the expansion of the enterprise asset. Inside the dashed-line rectangle is the expansion of the line of two. Inside the lightly-shaded rectangle is the expanded division for the second asset in the line. Inside the division is another division. Here the division is replicated three times (because of the three recursive calls to `Mult()`). The diagram shows the depth of the recursion (two levels). Figure 5 shows the results of expanding the diagram into a standard call graph showing all of the processes (except for some hidden Enterprise processes). The simple diagram of Figure 4 corresponds to a complex structure of 13 processes.

There are two important consequences to the way that legal Enterprise diagrams are constructed. The first is that not all parallel algorithms can be expressed using the Enterprise meta-programming model. For example, an algorithm that relies on a group of processes using peer-to-peer communication cannot be supported using current Enterprise assets. Although this problem can be alleviated somewhat as new Enterprise assets are designed, it will never really disappear. It is probably impossible to pre-determine a set of templates that can represent an arbitrary communications topology without reducing the level of the templates to a "connect-the-dots" approach.

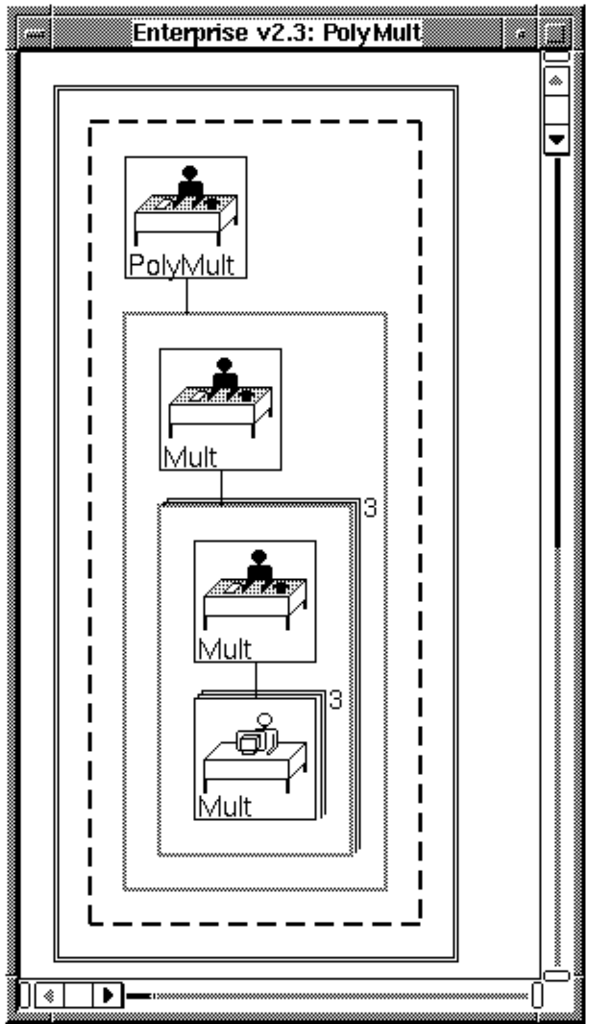

Figure 4. `PolyMult` asset diagram.

The second consequence is that it is impossible to draw a diagram that contains a logical parallelization error. For example, deadlock cannot occur in an Enterprise program because the only process graph cycles that can occur are strictly managed in the division asset†. Similarly, connection errors cannot occur due to missing communication channels, since these channels are established by Enterprise generated code, not user code.

Enterprise provides a consistent metaphor that allows parallelization techniques to be documented, explained, applied and combined.

† The user can create deadlock in their code by, for example, having one process go into an infinite loop or by having one process wait for an event (such as a file i/o) that never occurs. Enterprise guarantees that the communication structure cannot deadlock.

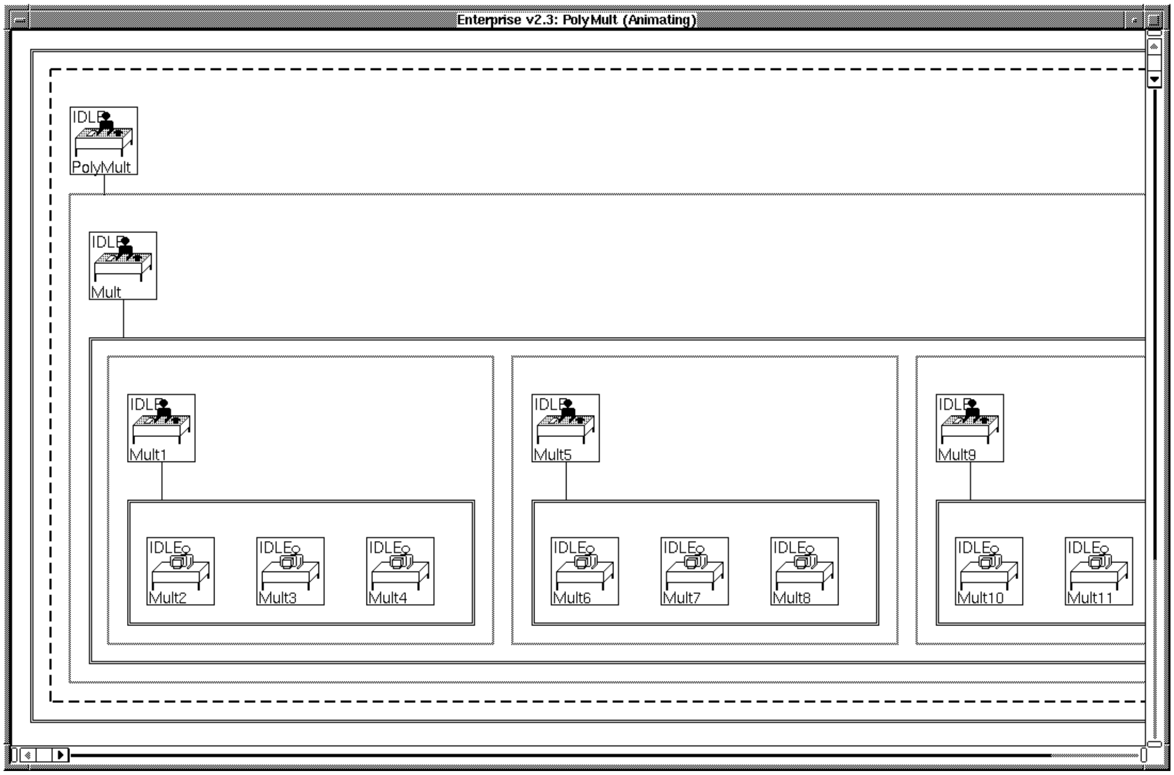

Figure 5. Expanded `PolyMult` asset diagram.

5. PROGRAMMING AND META-PROGRAMMING: ORTHOGONALITY

There are two aspects to executing a parallel program: the code used at compile time and the parameters needed to execute it at run-time. In older systems, these two properties were intertwined; changing the process/processor mapping meant editing the code and re-compiling. Newer systems abstract this information into a configuration file which specifies the process-processor mappings (for example, NMP [17] and P4 [5]). This file must be edited every time the execution environment of the program changes (such as machine availability) or run-time parameters are altered (such as a replication factor). This extra editing effort can be error-prone.

An Enterprise program consists of sequential C code and a diagram (meta-program) that represents the desired parallelization at a high level. What are the relationships between the program and its meta-program? First, every asset in the diagram must appear as a procedure call in the program. Second, the call structure of the diagram must match the call structure of the program. Third, every asset must be in its own disk file. All three of these requirements are checked by the Enterprise compiler and if there is a violation, the user is informed. Except for these three constraints, a program and its meta-program are completely orthogonal. That is, many parts of a meta-program can be changed without affecting the program code and visa-versa. For example, program and asset features (such as the replication factor, ordered or unordered attribute, which of its parameters are recorded during event logging, etc.) can be changed without changing the program source code. Alternatively, the internal code of any asset can be changed without affecting the asset diagram. Figure 6 illustrates this orthogonality.

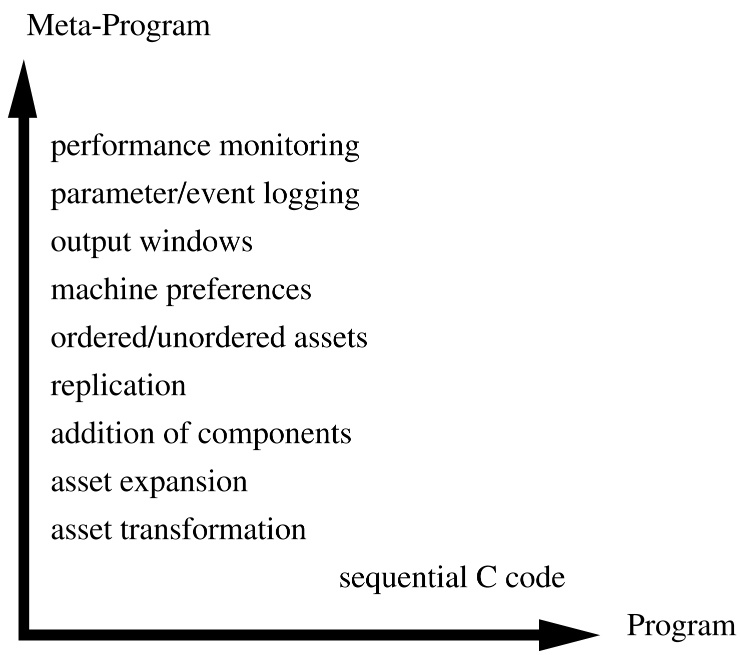

Figure 6. The orthogonality of a program and its meta-program.

This separation of a program's code and its parallel structure supports experimentation with different parallelization techniques and rapid performance tuning. The same program code can be mapped to a meta-program that uses a department or a line simply be re-drawing the diagram. Replication factors can be increased in locations where all replicas of an asset are busy and decreased where some replicas are idle. This can be done either manually by the user or automatically by the system.

As an example, consider the program of Figure 3 again. This program can be executed with the diagram as shown and performance information obtained (either by timing or by using some of the performance monitoring tools described in the next section). The meta-program (diagram) can then be edited to collapse the assets `C()`, `D()` and `E()` into a single individual asset. The program can then be recompiled without changing the source code (the asset calls to `D()` and `E()` are interpreted as local procedure calls now) and the program can then be re-executed. The performance of the modified meta-program can then be compared with the previous one and the meta-program with the better performance can be selected. Alternately, the replication factors of the replicated individual `B()` and the replicated line `C()` can be traded off, shifting available processors between `B()` and `C()` to obtain maximum performance. This modification to the meta-program can be made rapidly without changing the source code (or even recompiling in this case). The change to the meta-program that causes the biggest change to a user's program occurs when an individual is transformed into a composite asset like a line or a department. However, even in this case, the only change to the program is to shift the asset code for the component assets (some procedures) to separate files. The actual code itself may not have to change[†]!

The virtual orthogonality of sequential source code (a program) and its parallelization technique (a meta-program) allows users to focus on each of the two components independently. It is analogous in some respects to the art of programming with abstract data

[†] A change may be necessary if the parameters do not meet Enterprise requirements. For example, when executed sequentially the function may have relied on accessing values through global variables; they now have to be passed as parameters.

types where the use and implementation are independent. This approach significantly reduces the complexity of parallel programming.

6. ENVIRONMENT: INTEGRATED INTERFACE

Although a text editor and a compiler are the only tools necessary to write a computer program, the complexity of modern applications has led to the development of CASE tools to support the complete software development process. Unfortunately, in practice there are few CASE tools used to support the development of parallel applications. Those parallel tools that are used tend to concentrate on a single phase of the development process: coding, performance monitoring or debugging.

The two main goals for using tools are an increase in programmer productivity and a decrease in design and programming errors. Unfortunately, using an assortment of unrelated tools can have a negative effect on both of these goals.

Three problems are created by differences in tools. The first is a steep learning curve as programmers learn each new tool. This may exacerbate an already difficult situation if a programmer is simultaneously struggling with a switch from the sequential to parallel domain. The second problem is the number of context switches necessary when switching from tool to tool. Each tool reflects an underlying model or development philosophy, and switching models in mid-development not only requires additional time, it can also lead to errors that must be found and fixed. The third problem is that these tools often use different representations, and format translations can be time consuming and error-prone. For example, a post-mortem debugger and a performance monitor may both require an event log, but if they are in different formats then a third tool is required to translate the events. Fortunately, the third problem is being addressed by the parallel community, with the introduction of community standards (such as MPI [25]) and popular standards (such as PICL [10]).

The user interface of the Macintosh computer is a good example of the potential benefits of a common user interface. After a user learns one Macintosh application, it is easy to learn a second and a third because of the similarities in the user interface. It is also easy to switch from one application to another, and the clipboard provides a simple mechanism for moving information between them. Significant benefits can accrue from having a comprehensive programming environment that supports the entire software life-cycle of parallel applications using a common user interface.

Enterprise has a uniform interactive graphical user interface that supports a common programming metaphor across all of its components [15]. Enterprise provides three main views of a program: a design view, an animation view and a replay view. In all of the views, the program is represented as an asset diagram. The "look and feel" of the interface is identical across all three views.

6.1 Design View

The design view is used to edit the asset diagram (meta-program) of a program as shown in Figure 3. This view supports edit, dialog, compile and run windows that the user can use to edit source code, set asset attributes (like ordered/unordered), set compile/run options and to compile/run a program. Enterprise has an automatic *makefile* facility so that only the necessary files will be recompiled when a compilation is done (again, relieving programmers from the additional effort of constructing their own *makefiles*).

6.2 Animation View

The animation view is used to display an animation of program execution as recorded in an event log. The animation view is similar to the design view except that replicated assets are flattened so that each replica can be seen. Figure 7 shows the animation view of a program. In addition to the assets, messages and message queues are shown. This view actually animates messages and shows them moving between assets. The state of each asset is shown (IDLE, BUSY, BLOCKED, DEAD) using a label and a color code. The user can start, stop, resume and single step the animation. At any time, a message queue can be opened to display a list of messages and the logged parameters of a message can be viewed as well.

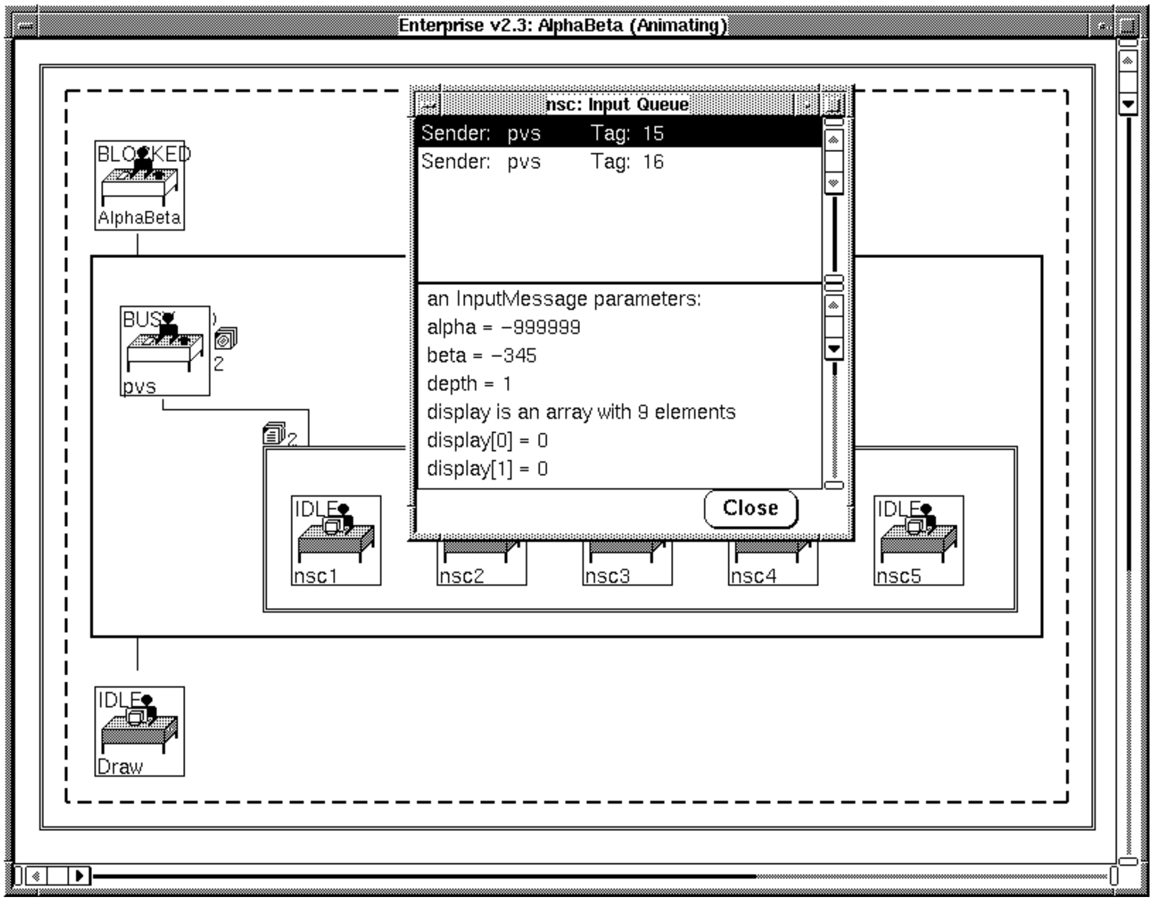

Figure 7. The animation view of Enterprise.

A variety of performance windows can be displayed in the animation view including graphs that show program speed-up as a function of execution time during a run (in analogy to a profits and loss chart for a business organization) and pie charts that show the relative idle, busy and blocked times for each asset. This information can be used to tune performance by quickly returning to the design view, changing the meta-program, re-running the program and viewing another animation. The common model and uniform user-interface make this a simple process.

6.3 Execution Replay View

Debugging distributed programs is a difficult endeavor. In most situations, sequential code is deterministic so that repeated execution with the same input data follows the same execution path. This makes it relatively easy to isolate and correct programming errors by executing the program repeatedly. Unfortunately, a parallel program may be non-deterministic. That is, it may follow different execution paths for the same input data due to race conditions between processors with different loads. A particular logic error may only occur on one run out of ten. To isolate and correct such an error, it may be necessary to reproduce the error run many times. To help reproduce such non-deterministic errors, a parallel debugger can include an execution replay mechanism (such as [24]). Execution replay works as follows. The program is executed with event-logging until the error occurs. The debugger then forces the program to re-execute, following the same event order as the event-log for the error run. The program can be re-executed in this order as many times as are necessary to isolate the error. Of course, the debugger should also provide a breakpointing mechanism to isolate the problem during these re-executions.

Enterprise contains a debugger capable of forcing re-execution of a program based on an event-log. The event-log is in the same format as is required by the animation view. In addition, the user interface for the replay view is a direct generalization of the user interface for the animation view so that once users have learned one, the other is easy. The Enterprise debugger allows the user to set breakpoints based on a wide variety of conditions like message type (send message, send reply, receive message, receive reply), message collaborators (sender asset and receiver asset) parameter values (like `a[i] = 3`) and event counts (like the third message sent). Figure 8 shows a replay view and a breakpoint browser.

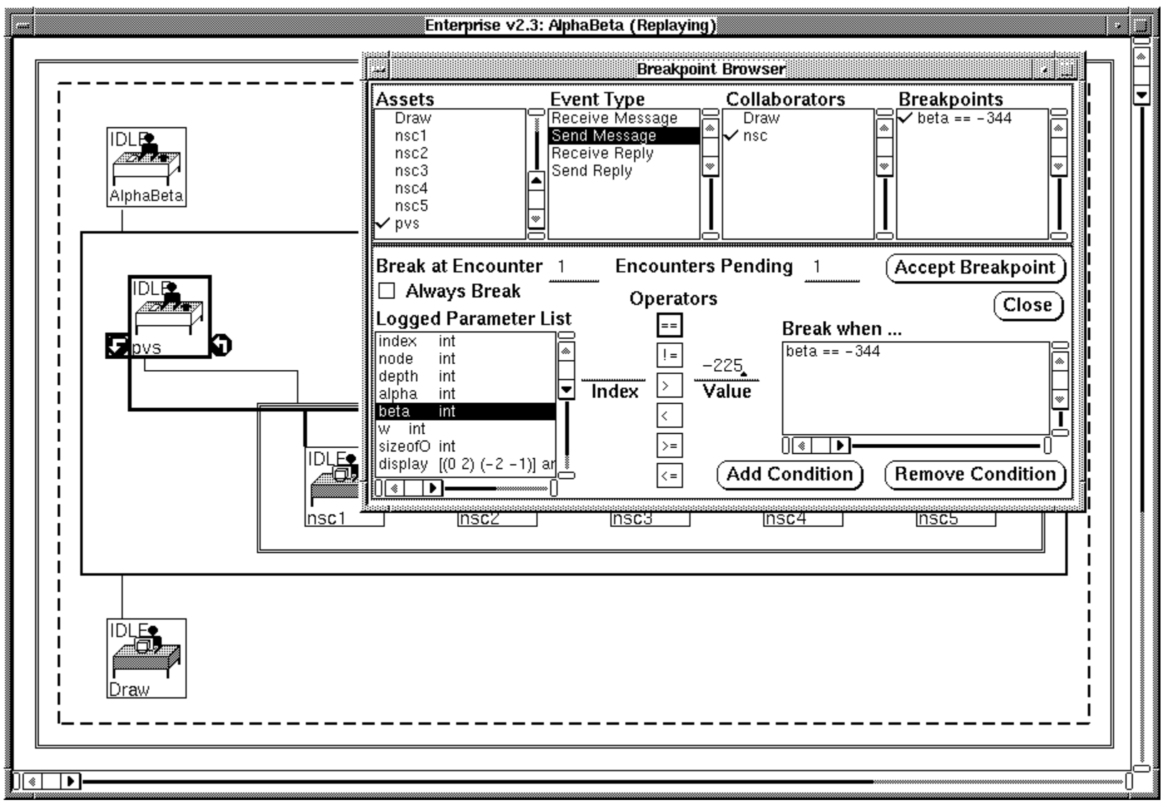

Figure 8. The replay view of Enterprise and a breakpoint browser.

The common user interface and uniform model make context switches between the design, animation and replay views completely seamless and increases the productivity of parallel programmers.

7. EXECUTION: RESOURCES

Most coarse-grained distributed applications are run in environments with changing conditions. The identities, types, configuration and availability of machines on the network can change frequently, even while a computation is running. The user should *not* be responsible for knowing the up-to-date status of the network resources available. However, many systems make it the user's responsibility to know the names of all machines used in the computation beforehand (such as NMP [17]). In others, dynamic reconfiguration is possible, but it is the user's responsibility to identify when it should be done. For example, user code may check to see if the processors are busy and take corrective actions. Again, this does not seem to be a good way to spend programming resources.

When a program is started, Enterprise decides which machines to use (although there are options for the user to make the selection if desired) based on availability and load average. The user does not need to know the identities of the machines used in a computation beforehand. Current research has Enterprise monitoring the status of machines on the network and dynamically changing the resources used by the program. This moves the network management overhead from the user to the system where it belongs.

Finally, a PPS should show interoperability between different systems. For example, network computing must face issues of different machine architectures, operating systems and file systems. PVM has made great progress in hiding many of these details from the user. Systems like Enterprise only exploit what PVM provides.

8. CONCLUSIONS

In many parallel programming tool papers, the authors illustrate the strengths of their approach by presenting some impressive speedup results. Is this the only metric that we should use in judging the quality of the tool?

We have conducted a controlled experiment that compared programming in Enterprise to programming with a PVM subset (NMP) [23]. Half of a graduate student class did an assignment using Enterprise; the other half used NMP [17]. Enterprise users ended up writing 66% less code than did NMP users. This translated into fewer compiles, editing sessions and test runs. It was interesting to compare the errors contained in the programs submitted by the students. Some NMP students had problems with program correctness; some Enterprise students had problems with performance. We argue that correctness is far more important than performance. Performance issues should only be addressed once correctness has been established.

Enterprise allows the user to quickly develop a correct parallel structure so that potential errors may only lie in the application-specific sequential code. Only when the program is running properly should performance be a consideration, and only if the performance does not meet expectations. In our experience there are many applications which can benefit from tools like Enterprise, but there is a mental block in the computing community that equates parallel programming with being a difficult task. Users are delighted when they can use a tool like Enterprise and quickly obtain speedups for their application. That the amount of the speedup might not be as high as is achievable using some low-level tool is irrelevant; any performance gain is welcome.

Although this paper used Enterprise as an example, most of the features described can be found in other tools. The state of the art in parallel programming systems is improving rapidly and the user community is more receptive to these new products. Over the last few years, PVM has emerged as a *de facto* standard for writing distributed applications (although MPI may change this [25]). We are still waiting for a higher-level tool, one that builds on top of PVM/MPI, to emerge and gain wide-spread acceptance.

ACKNOWLEDGMENTS

Our thanks to the current Enterprise team members: Paul Iglinski, Steve MacDonald, Chris Morrow, Diego Novillo, Ian Parsons and David Woloschuk.

REFERENCES

1. O. Babaöglu, L. Alvisi, A. Amoroso, R. Davoli and L. Giachini. Paralex: An Environment for Parallel Programming Systems. Technical report UBLCS-92-4, Laboratory for Computer Science, University of Bologna, 1992.
2. B. Bacci, M. Danelutto and S. Pelagatti. Resource Optimization via Structured Parallel Programming. *Programming Environments for Massively Parallel Distributed Systems*, Birkhauser Verlag, Basel, Switzerland, pp. 13-25, 1994.
3. H. Bal, M. Kaashoek and A. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, vol. 18, no. 3, pp. 190-205, 1992.
4. A. Beguelin, J. Dongarra, G. Geist, R. Manchek and V. Sunderam. Graphical Development Tools for Network-Based Concurrent Supercomputing. *Supercomputing '91*, pp. 435-444, 1991.
5. R. Butler and E. Lusk. Monitors, Messages, and Clusters: The p4 Parallel Programming System. *Parallel Computing*, vol. 20, no. 4, pp. 547-564.
6. N. Carriero, D. Gelernter, T. Mattson and A. Sherman. The Linda Alternative to Message-passing Systems. *Parallel Computing*, vol. 20, no. 4, pp. 633-655, 1994.
7. L. Clarke, R. Fletcher, S. Trewin, R. Bruce, A. Smith and S. Chapple. Reuse, Portability and Parallel Libraries. *Programming Environments for Massively Parallel Distributed Systems*, Birkhauser Verlag, Basel, Switzerland, pp. 171-182, 1994.
8. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Programming.* MIT Press, Cambridge, Mass., 1989.
9. J. Feo, D. Cann and R. Oldehoeft. A Report on the Sisal Language Project. *Journal of Parallel and Distributed Computing*, vol. 10, 1990.
10. G. Geist, M. Heath, B. Peyton and P. Worley. PICL: A Portable Instrumented Communication Library, Technical report ORNL/TM-11130, Mathematical Sciences Section, Oak Ridge National Laboratory, 1990.
11. G. Geist and V. Sunderam. Network-Based Concurrent Computing on the PVM System. *Concurrency: Practice and Experience*, vol. 4, no. 4, pp. 293-311, 1992.
12. A. Goldberg. Concert/C: A Language for Distributed C Programming. IBM T.J. Watson Research Center, Yorktown Heights, New York, 1993.
13. A.R. Halstead. MultiLisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 4, pp. 501-538, 1985.
14. W. Karpoff and B. Lake. PARDO - A Deterministic, Scalable Programming Paradigm for Distributed Memory Parallel Computer Systems and Workstation Clusters. *Supercomputing Symposium '93*, Calgary, pp. 145-152, 1993.
15. G. Lobe, D. Szafron and J. Schaeffer. The Enterprise User Interface. *TOOLS (Technology of Object-Oriented Languages and Systems) 11*, R. Ege, M. Singh and B. Mayer (editors), pp. 215-229, 1993.
16. D. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, vol. 1, no. 1, pp. 25-42, 1993.
17. T. Marsland, T. Breitkreutz and S. Sutphen. A Network Multiprocessor for Experiments in Parallelism. *Concurrency: Practice and Experience*, vol. 3, no. 1, pp. 203-219, 1991.

18. D. Ritchie and K. Thompson. The UNIX Timesharing System. *Communications of the ACM*, vol. 17, no. 7, pp. 365-375, 1974.

19. J. Schaeffer, D. Szafron, G. Lobe and I. Parsons. The Enterprise Model for Developing Distributed Applications. *IEEE Parallel and Distributed Technology*, vol. 1, no. 3, pp. 85-96, 1993.

20. R. Sedgewick. *Algorithms.* Addison-Wesley Publishing Company, 1983.

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

22. Sun Microsystems. Remote Procedure Call Programming Guide. Sun Microsystems, 1986.

23. D. Szafron and J. Schaeffer. Experimentally Assessing the Usability of Parallel Programming Systems. *Programming Environments for Massively Parallel Distributed Systems*, Birkhauser Verlag, Basel, Switzerland, pp. 195-201, 1994.

24. D. Taylor. The Use of Process Clustering in Distributed-System Event Displays. *CASCON '92*, vol. 1, IBM Toronto, pp. 29-42, 1992.

25. D. Walker. The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers. *Parallel Computing*, vol. 20, no. 4, pp. 657-673, 1994.