Experience with Parallel Programming Using Code Templates

Ajit Singh,[1] Jonathan Schaeffer,[2] Duane Szafron,[2]

asingh@etude.uwaterloo.ca, jonathan@cs.ualberta.ca, duane@cs.ualberta.ca

[1] University of Waterloo, Dept. of Electrical and Computer Eng., Waterloo, Ontario, Canada N2L 3G1

[2] University of Alberta, Dept. of Computing Science, Edmonton, Alberta, Canada T6G 2H1

Abstract

For almost a decade we have been working at developing and using template-based models for parallel computing. Template-based models separate the specification of the parallel structuring aspects from the application code that is to be parallelized. A user provides the application code and specifies the parallel structure of the application using high-level icons, called templates. The parallel programming system then generates the code necessary for parallelizing the application. The goal here is to provide a mechanism for quick and reliable development of coarse-grain parallel applications that employ frequently occurring parallel structures. Our initial template-based system, FrameWorks, was positively received but had a number of shortcomings. The Enterprise parallel programming environment evolved out of this work. Now, after several years of experience with the system, its shortcomings are becoming evident. Controlled experiments have been conducted to assess the usability of our system in comparison with other systems. This paper outlines our experiences in developing and using these systems. A list of desirable characteristics of template-based models is given. FrameWorks and Enterprise systems are discussed in the context of these characteristics and the results of our usability experiments. Many of our observations are relevant to other parallel programming systems, even though they may be based on different assumptions. Although template-base models have the potential for simplifying the complexities of parallel programming, they have yet to realize these expectations for high-performance applications.

1 Introduction

Along with the growing interest in parallel and distributed computing, there has been a corresponding increase in the development of models, tools and systems for parallel programming. Consequently, practitioners in the area are now faced with a somewhat difficult challenge: how to select parallel programming tools that will be appropriate for their applications. There is no easy answer. The decision is a function of many parameters including some that are specific to the user and their computing environment. These include the type of parallelism available in the application (for example: fine- or coarse-grained; data parallel or not; pipeline or master-slave), the target architecture(s) (for example: shared or distributed memory), language constraints and performance expectations. Other parameters are specific to the tool and its capabilities, including its feature set (portability, fault tolerance) and usability (ease of use, flexibility, expressive power).

As is evident from the formation of user groups such as the Parallel Tools Consortium, there is a concern in the community about the lack of post-development analysis and evaluation for various tools and technologies that are being proposed. Typically, researchers envision a new tool or technology, develop it and, depending on their initial experiences, report it in the literature. With few exceptions, long-term experiences with parallel programming systems and their relationships with similar systems are hardly ever reported.

Many different approaches have been taken towards the development of parallel programming models. A new parallel programming language is one approach (for example, a procedural language such as Orca [1], or a functional language like Sisal [2]). However, practical considerations, such as legacy code and the demand for Fortran/C-based languages, often make this an impractical choice. Alternatives that allow the programmer to take advantage of the existing code and expertise in common sequential languages have found much wider acceptance. One such approach includes providing libraries for parallelization (PVM [3], P4 [4] and MPI [5] are examples). Another approach is to extend an existing sequential language with compiler directives (High Performance Fortran [6]), or keywords (for example, Mentat [7] and PAMS [8]).

A relatively new alternative has begun to emerge that allows a programmer to benefit from the existing code and knowledge of a sequential program, while minimizing the modifications that are required for parallelization. The programmer provides a specification of the parallel structuring aspects of the application in the form of code annotations. One interesting approach to code annotation is to recognize that there are commonly occurring parallel techniques. A parallel programming tool can support these techniques by providing algorithmic skeletons [9], or templates, that capture the parallel behavior. The user provides the sequential application code and selects the templates required to parallelize the application (such as in PIE

[10] and HeNCE [11]). The system then generates the necessary parallel code. Template-based models separate the specification of the parallel structuring aspects – such as synchronization, communication and process-processor mapping – from the application code that is to be parallelized. A template implements commonly occurring parallel interactions in an application-independent manner. The goal here is to provide an easy approach for the initial development and restructuring of coarse-grain parallel applications that relies on commonly used parallelization techniques.

This paper discusses our long-term experiences with two template-based parallel programming systems for coarse-grained parallelism. Our research began in 1986 when we used templates to experiment with different parallel structures for a computer animation application [12]. We quickly realized that the approach was more general, and could be used to build a larger class of parallel applications. Building on this success, the FrameWorks parallel programming tool was developed [13, 14, 15]. Our initial experience with FrameWorks was encouraging. However, for a number of reasons described later in this paper, it was not possible to evolve the system beyond a certain point. Consequently, an entirely new project, called Enterprise, was initiated. Enterprise is a template-based parallel programming environment which offers a much wider range of related tools for parallel program design, coding, debugging and performance tuning [16, 17, 18, 19, 20, 21]. It has been publicly available since 1993 (http://web.cs.ualberta.ca/~enter).

Several other parallel programming systems have relied on techniques that are similar to the approach used by us (for example [22, 11, 23, 24, 25, 26, 10]). Many of our results and experiences with FrameWorks and Enterprise are applicable to such systems, as well as other high-level parallel programming systems.

Before we delve into details, it is useful to clarify a couple of points regarding our use of the term "template." In the past, techniques based on the use of application-independent common parallel structures have often been described under different names such as algorithmic skeletons [9], model programs based on parallel programming paradigms [27, 28], and parallel program archetypes [29]. In addition to us, some other researchers also have recognized or used the term "template" to refer to such techniques [28, 29, 30, 10]. Although the underlying details of these techniques vary significantly, they all have the common goal of specifying commonly occurring parallel structures in the form of application-independent and reusable code. For the last ten years, we have used the term "template-based" to refer to this technique. At this point, it should also be pointed out that our use of the term "template" here is quite distinct from C++ templates. Our usage is restricted to the context of parallel programming, where it is used to denote a prepackaged set of application-independent characteristics. This has no intended relationship with the C++ templates which are used to build generic classes in sequential programs.

In this paper we look at template-based parallel programming models from two viewpoints. First, as the designers, we can address the difficulties in the design and implementation of these tools. Second, we have had considerable interaction with users developing template-based parallel applications. Controlled experiments, which compared Enterprise with a number of tools including PVM, give insights about the strengths and weaknesses of the template-based approach. The result is that although template-based models have tremendous potential for bridging the gap between sequential and parallel code, there still remain a number of shortcomings that must be addressed before the technology will be widely used.

Section 2 describes the template-based approach. Section 3 explains the distinctions between this approach and other high-level techniques for parallel programming. Section 4 outlines the objectives for an ideal template-based parallel programming tool and discusses their significance. Section 5 briefly describes FrameWorks and its shortcomings. These problems with FrameWorks led to the design of Enterprise, as described in Section 6. Section 7 describes our experiences with template-based models and the lessons learned. Extending the template model to other aspects of parallel programming is discussed in Section 8. Section 9 describes the requirements for future template-based tools. Finally, Section 10 presents our conclusions.

This paper may seem to be overly critical of template-based approaches. Our intent is not to discourage research in this area. Rather, we believe that far too many papers in the parallel computing literature are long on praise and short on criticism. It is our hope that the issues discussed in this paper can be seriously tackled by the research community, so the full potential of template-based tools can be realized.

2 Template-based Programming

In the context of parallel programming, a template represents a prepackaged set of characteristics which can fully or partially specify the nature of scheduling, communication, synchronization and processor bindings of an entity. Templates implement various types of interactions found in parallel systems, but with the key components – the application-specific procedures – unspecified. A user provides the application-specific procedures and the tool provides the glue to bind it all together. The templates abstract commonly occurring structures and characteristics of parallel applications. The objective here is to allow users to develop parallel applications in a rapid and easy manner.

For example, consider a graphics animation program (*Animation*) consisting of three modules (*Generate()*, *Geometry()*, and *Display()*). It takes a sequence of graphical images, called frames, and animates them. *Generate()* computes the location and motion of each object for a frame. It then calls *Geometry()* to perform actions such as viewing transformations, projection and clipping. Finally, the frame is processed by *Display()* which performs hidden-surface removal and anti-aliasing. Then it stores the frame on the disk. After this, *Generate()* continues with the computation of the next frame and the whole process is repeated. Figure 1 shows the structure of a sequential version of the animation program.

(Figure 1 should be placed here.)

A simple way to parallelize this application would be to let the three modules work in a pipelined manner on different processors. After computing a frame, *Generate()* passes it to *Geometry()* for processing and starts working on the next frame. Similarly, *Geometry()* passes its output to *Display()* and then receive its next frame from *Generate()*. Therefore, all three modules work in parallel on different frames (see Figure 2a). Now, if *Display()* takes much longer to do its processing as compared to *Generate()* and *Geometry()* (which is generally the case in reality; hidden-surface removal and anti-aliasing require much more time than the other components of the program), more than one instance of *Display()* can be initiated. This is possible because the processing of each frame is independent. Similarly, if the performance of *Geometry()* is to be improved, several instances of it may be initiated as well. This situation is shown in Figure 2b where *Geometry()* and *Display()* have several active instances.

(Figure 2 should be placed here.)

This parallel version of *Animation* contains two of the commonly-used structures for parallel computing, namely the pipeline and the replication. Consider parallelizing this application on, for example, a network of workstations. Parallel program development would require a significant amount of time and effort if a low-level tool was used (for example, Unix sockets [31] or a message-passing library such as MPI). Further, the parallelism would be explicit in the code, increasing the complexity of the program. Each time the programmer wanted to experiment with a different parallel structure for the application, additional programming effort would be required to rewrite the code. Moreover, such an effort would be replicated, knowingly or unknowingly, by other programmers while writing other applications.

Template-based parallel programming systems provide skeletons (templates) of implementations of such parallel structures. A user simply provides sequential modules of code and selects the appropriate templates to structure the parallel application. As explained later, the templates of FrameWorks (or Enterprise) can be used to quickly generate the parallel structures shown in Figure 2. The procedural relationships in the diagram indicate that the three modules interact in a pipeline manner and that *Geometry()* and *Display()* can have multiple instances that execute independently from each other. The choice of template indicates which communication pattern the system automatically generates. The resulting parallel program automatically spawns the processes on available processors, establishes the communication links and ensures the proper communication and synchronization. From the user's point of view, all the coding is sequential; all the parallel aspects are generated by the system. By separating the application-specific code from the parallel implementation, template-based development tools aim to decrease program development time and reduce the number of program errors due to parallelization.

In addition to FrameWorks and Enterprise, there are several other template-based parallel systems in the literature (for example [22, 11, 23, 24, 25, 26, 10]). Typically these systems differ on several dimensions including the selection of templates available to the user, restrictions on the code associated with templates, restrictions on the data that can be passed between templates, and correctness properties (such as deadlock) of the generated program. It is the approach to these issues that distinguishes one system from another.

3 Templates Versus Other High-level Techniques

Several different high-level models have been used for the design of parallel programming tools. This section compares the important properties of template-based systems to those of other well-known, high-level techniques for building parallel applications.

A template encapsulates certain behavior in a parallel environment. A programmer using a template is concerned only with its specified behavior. The actual implementation may vary from environment to environment depending on, among other things, the architecture and the operating system. In some ways, this is analogous to programming with abstract data types, which provide well-defined means for manipulating data structures while hiding all the underlying implementation details from the user.

Macros and message-passing libraries are popular implementations of high-level parallel models. However, the separation of application code and parallelization code is a key difference between templates and these methods. For example, the programmer must explicitly insert macros or library functions in the application code. On the other hand, templates are non-intrusive: there need not be any reference in the user's sequential code to the templates. This has important implications both for new parallel program development and for the restructuring of existing parallel applications.

Application-specific parallel libraries provide a second form of high-level abstraction for parallel programming. For example, PBLAS implements library routines for parallel applications based on linear algebra. These routines hide the underlying details of the parallel solution from the user. The user only needs to supply the data for a particular instance of the problem. There are two fundamental differences between application-specific libraries and templates. First, libraries provide an application-specific parallel solution; templates are application-independent. The application-independent nature of templates has also been emphasized by other researchers [28, 29, 10]. Second, a programmer using templates has the freedom to choose between different parallel solutions to a problem; application-specific libraries usually provide a single solution.

New programming languages are a third technique for supporting high-level abstractions for parallel programming [32, 1]. Although the approach has some advantages, a serious disadvantage is that a programmer cannot make use of the existing code for the sequential version of an application. Some argue that parallel applications should be written from scratch. However, this argument is not consistent with the way complex tasks are usually solved. Initially, the emphasis is on finding a (sequential) solution to the task. It is only when the solution begins to take a significant amount of execution time that people start thinking about parallelizing the application. However, by this time a large investment has been made in the sequential solution. In a template-based system, the programmer can often reuse the existing sequential legacy code.

While developing parallel applications, programmers often think in terms of certain high-level abstractions, such as master-slave, pipeline, or divide-and-conquer. Refinement of these abstractions to low-level primitives is postponed until the implementation phase. Template-based systems attempt to directly support these abstractions. The user specifies the required abstractions, and the system generates the required code. To achieve the desired behavior, the system may have to insert code at many places in the user's sequential code. This is an important difference from techniques such as macro calls where the expanded code is localized at the point of the macro call.

The concept of templates is consistent with Simon's views on chunking of knowledge [33]. According to this view, people do not generally think in terms of individual low-level operations while solving complex tasks. Rather, they organize their thoughts in terms of strategies which consist of chunks of low-level operations structured in certain ways.

While templates encourage code reuse, they do not eliminate the need to rewrite sequential code to adapt it to a parallel environment. Like any other parallel tool, some code rewriting or restructuring may be necessary to expose the parallelism, satisfy the programming constraints of the tool, or achieve improved performance.

4 Desirable Characteristics of Template-Based Models

As we gain more insight into how programmers develop parallel applications, and how different template-based systems can be built, we get a better understanding of characteristics that should be (or could be) present in template-based systems. In this section, we outline what we feel are the important characteristics of the ideal template-based model. No tool presently exists that supports all of these features. The list is used in this paper to serve as a benchmark for analyzing FrameWorks, Enterprise and other systems. In the following discussion, each of the characteristic is given a short name which is shown inside parentheses. These names are used throughout the paper to refer to the corresponding characteristics.

4.1 Structuring the Parallelism

Template-based systems should allow the fewest possible restrictions on how the user can structure the parallelism in an application. The most important structural properties are:

1. Separation of Specification (*Separation*): This is the central feature of a template-based system. It means that it should be possible to specify the templates (i.e., the parallelization aspects of the application) separately from the application code. This characteristic is crucial for rapid prototyping and performance tuning of a parallel application. It also allows for the application code and its parallelization structures to be evolved in a semi-independent manner.

2. Hierarchical Resolution of Parallelism (*Hierarchy*): This allows the refinement of a component in a parallel application graph by expanding it using the same model. That is, templates can include other templates. Therefore, there is no need to have separate models for "programming-in-the-large" and "programming-in-the-small."

3. Mutually Independent Templates (*Independence*): It is not sufficient to define some templates that can be used with other templates. The meanings of all templates should be context insensitive so that they

can be used with other templates.

4. Extendible Repertoire of Templates (*Extendible*): It should be possible for a user to extend the set of templates available.

5. Large Collection of Useful Templates (*Utility*): The system should be useful over a wide range of applications.

6. Open Systems (*Open*): It should be possible for the programmer to include lower-level mechanisms, such as explicit message passing, in their application. The absence of such a feature results in a closed system where the only applications that can be developed are those whose required parallel structures match the templates. This is a very difficult requirement as it has significant implications for application development, debugging, and performance tuning.

4.2 Programming

Templates may impose constraints on how users write sequential code.

1. Program Correctness (*Correctness*): The system should offer some guaranteed properties of correctness. For example, absence of deadlocks, deterministic execution, and fault tolerance are some desirable correctness features.

2. Programming Language (*Language*): The system should build on an existing commonly-used language. Ideally, there should be no changes to the syntax or semantics of the language. This facilitates reuse of existing sequential code and makes it possible to take advantage of existing expertise in sequential programming.

3. Language Non-Intrusiveness (*Non-Intrusiveness*): A system may satisfy the *language* objective, but force the user to change sequential code to accommodate limitations in the parallel programming model. For example, to develop a parallel application using a message-passing library, the user may have to appropriately restructure the code and insert calls to the message-passing library in the code. The only way to properly eliminate this problem and also satisfy the *language* constraint is to have a compiler that automatically parallelizes the code. Unfortunately, for coarse-grained applications, the required compiler technology does not exist.

4.3 User Satisfaction

The system must satisfy a number of performance constraints, both at program development time and at run-time. These include:

1. Execution Performance (*Performance*): The maximum performance possible, subject to the combination of templates chosen by the user, should be achievable. There will always be limitations to the achievable performance. The complexity and interdependence of components external to the system (communication subsystem, operating system, network, etc.) make it very difficult to abstract and still attain the highest possible performance.

2. Support Tools (*Support*): The system should provide a complete set of design, coding, debugging and monitoring tools that support the template-based model. These tools must support the same level of abstraction as the programming model.

3. Tool Usability (*Usability*): The ideal tool should have a high degree of usability. It should be easy to learn and easy to use. Usability assessments have been neglected in the literature [20].

4. Application Portability (*Portability*): The tool should allow the user to port applications to a number of different architectures. Some performance losses may be expected for a poorly-chosen architecture, but the program should still run.

5 Outline of FrameWorks

This section provides a brief overview of the FrameWorks' model and system. A more complete description can be found in [13, 14, 15].

FrameWorks represents our initial attempt at developing a template-based system. In the FrameWorks model, an application consists of a fixed number of modules which are written using an extended version of a high-level language (*C*). A *module* consists of a set of procedures, exactly one of which is specified as the *entry* procedure. The entry procedure of a module can be called by other modules in the application in a manner similar to local *C* procedure calls. A module may also have *local* procedures which may be called only from within the module. There are no common variables among the modules. Each application has one *main* module, that contains the *main()* procedure. The main module may or may not have an entry procedure.

5.1 The Interconnection Structure

FrameWorks provides a set of templates for specifying the interconnection among communicating modules. A module's complete interconnection with other modules can be described by a 3-tuple:

(input-template, output-template, body-template).

For each type of template, the user must select one of the choices available and specify the input and output links for each module. This information is used by the system to generate an expanded version of a module containing the low-level code for parallel synchronization, scheduling and communication. To distinguish an original module from its expanded version, the latter is referred to as a process.

Input templates describe the interface through which a process receives its input. There are three options for input templates: *initial*, *in-pipeline*, and *assimilator* (Figure 3a). A process with an *initial* template does not receive any input from other modules. This template is used only by the main module of the application. A process using an *in-pipeline* receives its input from any of its input processes and serves them in a first-come-first-served manner. In the case of an *assimilator* template, the process takes exactly one input from each of its input processes before calling the entry procedure of the enclosed module.

(Figure 3 should be placed here.)

Similarly, there are three output templates: *out-pipeline*, *manager*, and *terminal* (Figure 3b). A process with an *out-pipeline* template can call any of its output processes. A *manager* template is used for executing a fixed number of copies of each of its output processes. A process whose output is marked as *terminal* does not call any other process.

A body template is used to assign additional characteristics to a module which modify the module's execution behavior in the distributed environment. The use of a body template is optional. There are two choices for the body template: *executive* and *contractor* (Figure 3c). The *executive* template causes the process to have its input, output and error streams directed to the user's terminal. The *contractor* template is useful for computationally intensive processes of an application by dynamically utilizing idle processors at run time. When a module's body is declared as *contractor*, the run-time environment executes a variable number of replicas of the given module. Each of these replicas is known as an *employee* of the *contractor*. A *contractor* process hires an unspecified number of employee processes to get the job done. The designer of the application does not take part in the hiring and firing of employee processes; the user simply specifies that the given process should function as a *contractor*. Process management is performed by the run-time environment and is transparent to the designer.

5.2 Communication Among Modules

Modules communicate with each other using programmer-specified structured messages called *frames*. A frame is similar to a C structure except that pointer type variables are not allowed. For each link between two modules, the programmer specifies two frames: an input frame (a structure containing all the input parameters needed for a call) and, if necessary, an output frame (containing all the reply or output values returned). Execution of an application is initiated by the *main()* procedure of the main module. Modules interact with each other using FrameWorks' call statements:

```
call name( inputframe );
       or
outputframe = name( inputframe );
```

where *name* is the name of the module called.

The two forms of FrameWorks calls shown above operate in the *non-blocking* and *blocking* modes, respectively. The non-blocking mode implies that the calling module will not wait for the completion of the call. Instead it will continue with its own execution as soon as the called module has received the data. In the blocking mode, the calling module waits until a reply frame is returned via another FrameWorks' construct, the *reply* statement. Within the called module, if the statement

```
reply( outputframe );
```

is encountered, the data in *outputframe* is returned to the calling module. After the called module returns, it starts waiting to serve another incoming call.

(Figure 4 should be placed here.)

As an example, to structure the graphics application discussed in Section 2, the user simply attaches input and output templates to the three modules, as shown in Figure 4a. The only required change to the source code is to extend the normal procedure call to *Geometry()* and *Display()* into FrameWorks' non-blocking call statements. To replicate the execution of *Geometry()* or *Display()* one simply attaches the *contractor* template to these modules (Figure 4b). No further modifications to the code are required. The system creates and manages execution of variable number of instances of *Geometry()* and *Display()* modules. The exact number of instances employed depends on the work load of these modules as well as availability of lightly loaded processors on the network.

5.3 Experience and Lessons of FrameWorks

FrameWorks was a prototype system used to demonstrate the feasibility of template-based concepts. Our experience with FrameWorks indicated that in the case of existing sequential applications, reasonable performance gains could be obtained using the simple modifications needed to create a coarse-grain parallel version of the applications. In several cases, partitioning of the complete application into modules was possible while keeping most of the code of the sequential version intact. In some cases, however, more efficient partitioning of modules required a significant amount of restructuring. In the case of applications that were designed with FrameWorks in mind, the amount of work to switch between the sequential and parallel version was quite small. In such cases, experimenting with different templates often required either no modifications or only a small number of modifications within the modules.

Although the initial experience with FrameWorks was encouraging, gradually several problems with the model and the system became apparent. The major limitations included:

1. The parallelism was expressed in the code (*call* and *reply*) *and* in the graphical user interface (violating the *separation* and *non-intrusiveness* objectives). The consequence was that changes in the template specification had to be mirrored in the code, increasing the chance of user error.

2. FrameWorks required the user to specify as many as three templates to fully describe the parallel structure of a process. There were some subtle constraints on how these templates could be combined, eliminating illegal and impractical combinations. Users often found these constraints confusing (poor *usability*).

3. The blocking version of the *call* primitive is a source of inefficiency. In this case, the calling module is blocked waiting for the reply frame even though it may not immediately need it to proceed with its computation (resulting in decreased *performance*).

4. The *call* and *reply* primitives can use only a single frame as a parameter for exchanging data. Frames

are limited to non-pointer data, restricting the parameter passing possibilities. Since sequential C programs often use pointers for passing data to functions, these restrictions often required significant modifications to the sequential code to support the FrameWorks method of parameter passing (failing the *non-intrusiveness* objective).

5. For its time, FrameWorks was quite novel in its approach toward structuring parallel applications. After its publication, we had several requests for the software from other researchers and practitioners. However, the FrameWorks system was not an easily portable system. The main reason for this was its dependence on a home-grown message-passing library and user-interface management tools. Although these tools helped us quickly develop the prototype system, porting FrameWorks to a new system meant installing all the tools and libraries it used. Some of the tools in turn depended on other locally developed research tools. These constraints made the job of porting FrameWorks to other sites very difficult (violating the *portability* objective).

6 Enterprise Parallel Programming System

Enterprise is not just a parallel programming tool; it is a parallel programming environment. It is a complete tool set for parallel program design, coding, compiling, executing, debugging and profiling. A detailed description can be found in [16, 17, 18, 19, 20, 21].

6.1 Improvements in Enterprise over FrameWorks

Enterprise represents an advancement over FrameWorks in several ways:

1. Enterprise combines the three-part templates of FrameWorks into single units, called *assets*, that represent all the useful cases. This eliminates the issue of illegal or impractical combination of partial templates (improving *usability*). Enterprise also introduces some new templates (improved *utility*).

2. In Enterprise, the use of FrameWorks' *call* and *reply* keywords was eliminated. By using a pre-compiler, Enterprise automatically differentiates between a procedure-call and a module-call based on the application graph (called the asset diagram). In effect, all the parallel specifications are in the asset diagram, not in the user code. This creates an orthogonal relationship between the application code (programming model) and the asset diagram (meta-programming model). Enterprise largely satisfies the *separation* objective.

3. Enterprise allows templates to be hierarchically combined to form a parallel program, almost without limitation (satisfying the *hierarchy* objective).

4. A useful debugging feature is that Enterprise programs can be run sequentially or in parallel often without changes to the code, asset diagram or recompiling. Also, the events in a parallel program execution can be logged, so that the program can be deterministically replayed.

5. An analysis of the operational model of FrameWorks' templates proved that a template would not cause a deadlock due to interactions within its components [34]. The analysis also showed, however, that deadlock is still possible in an application where modules make blocking calls to one another in a cyclic manner. Use of the assimilator template was also shown to cause a deadlock under some situations. Learning from this, Enterprise eliminated the assimilator template. It also restricted the application graph to be only tree-structured. This eliminated the possibility of an Enterprise application getting into a deadlock situation either due to its internal operation or due to cycles in the application's call graph. The user can, however, still write code to cause a deadlock. For example, an asset may be in an infinite loop due to some programming error thus resulting in an indefinite wait for the entire application. These deadlock properties contribute towards the *correctness* objective.

6. In FrameWorks, when a module call is made that returns a result, the caller is blocked until the callee replies. Enterprise uses *futures* [35] to let the caller proceed concurrently until it needs to access results. In effect, Enterprise uses compiler technology to postpone synchronization as long as possible. The result is improved *performance*. Enterprise has the synchronization implicit in the code; in Frameworks it is explicit.

7. Unlike FrameWorks, Enterprise module calls are not restricted to a single parameter. Moreover, Enterprise uses its pre-compiler to take care of marshaling and unmarshaling of parameters. This eliminates the need for frames and allows parallel procedure calls to look like sequential procedure calls. Further, Enterprise allows pointers to be passed as parameters, although the system does not support passing pointer data that itself contains pointer data. This considerably improves the *non-intrusiveness* of the system.

8. FrameWorks used analogies to illustrate the operations of its templates (for example, manager, master-slave and contractor). However, it was not quite consistent in its approach. Often it mixed these

with somewhat unclear terminologies, such as *in-pipeline* or *assimilator*. Enterprise relies on a single consistent analogy of a human organization to apply, document, and explain parallel structures. Human organizations are excellent examples of parallel systems. The analogies are intended to reduce the perceived difficulty of learning parallel programming, improving the *usability* of the system.

9. Enterprise has been implemented with the *portability* objective in mind. The system is implemented on top of existing, easily-accessible technology. Its user interface supports X-Windows and was written in Smalltalk. The pre-compiler was built using the Sage tools [36]. The run-time library can use any one of three message-passing kernels: PVM [3], ISIS [37] and NMP [38]. All these systems are available on a large number of systems.

6.2 Enterprise Programming Model

Consider a call from a module *A()* to a module *B()*:

```
Result = B( Param1, Param2, ..., ParamN );
/* some other code */
Value = Result + 1;
```

The sequential semantics of such a call is that *A()* calls *B()*, passing it N parameters, and then blocks waiting for the return value(s) from *B()* before resuming execution. Enterprise preserves the effects of the sequential semantics but allows *A()* and *B()* to execute concurrently. When *A()* calls *B()*, the parameters to *B()* are packaged into a message (marshaled) and sent to the process that executes *B()*. After calling *B()*, *A()* continues with its execution until it tries to access *Result* to calculate *Value*. If *B()* has yet not completed execution, then *A()* blocks until *B()* returns the *Result*. These so-called *futures* significantly increase the concurrency without requiring any additional specification from the user. In effect, a future is the synchronization primitive in Enterprise. For many applications, the sequential code looks identical to the parallel code and has equivalent semantics.

Enterprise allows pointer type parameters in module calls. The macros *IN()*, *OUT()* and *INOUT()* can be used to designate input, output and input-output type parameters. For example, consider the following program segment where *A()* calls *B()*:

```
int Data[100], Result;
...
Result = B( &Data[60], INOUT(10) );
/* some other code */
Value = Data[65] + 1;
```

The second parameter, *INOUT(10)*, indicates that 10 items of parameter *Data* are to be used for input as well as output. Here, the module call to *B()* sends elements 60..69 of *Data* to *B()*. When *B()* finishes executing, it copies 10 elements back to *A()*, overwriting data locations 60..69. *A()* will block when it accesses *Data[65]* if the call to *B()* has not yet returned. It should be noted that due to weak typing in C, it is not always possible to deduce the length of the pointer type argument. Therefore, an additional parameter indicating its length is necessary.

In fact, these macros are not necessary. If they were not included in Enterprise, then all pointer parameters would be treated as *INOUT*, preserving the sequential semantics. However, performance would be lower since all pointer data would be copied both on asset call and return. The macros have been included so that programmers can give the system important guidance to improve communication efficiency.

6.3 Enterprise Meta-Programming Model

The meta-programming model of Enterprise consists of templates, called assets, and a few basic operations that are used to combine different assets to achieve the desired parallel structures. As in FrameWorks, sequential code is attached to assets to get a complete parallel application. Enterprise currently supports assets whose icons are given in Figure 5.

(Figure 5 should be placed here.)

Enterprise: It represents a program and is analogous to an entire business organization. By default, every *enterprise* asset contains a single *individual*. A developer can transform this *individual* into a *line*, *department* or *division*, thus facilitating hierarchical structuring and refinement.

Individual: It represents a slave in traditional parallel programming 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: A *line* is analogous to an assembly or processing line (it is usually called a pipeline in literature). It contains a fixed number of heterogeneous assets in a specified order. The assets in a *line* need not necessarily be *individuals*; they can also be other *lines*, *departments* or *divisions*. 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 task for the previous call, not until the entire *line* is finished.

Department: A *department* represents a master/slave relationship in the traditional parallel computing 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.

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 assets 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 fanout at each level.

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 asset but other assets 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.

Enterprise provides a small set of building blocks from which users can construct complex programs using a simple mechanism. 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 assets can be expanded to reveal their default components. Component assets can be added to *lines* and *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 the 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.

(Figure 6 should be placed here.)

The graphics example of Section 2 can be parallelized in Enterprise with minimal changes to the original source code (mostly pointer parameters). Contrast the Enterprise asset diagram in Figure 6 to the Framework's diagram in Figure 4. Note the hierarchical composition: Figure 6a shows *Animation* as a single organization or *enterprise*. Expanding the icon reveals its inner structure, a *line* of assets (Figure 6b). Expanding that shows that the *line* consists of three *individuals* (*Generate()*, *Geometry()*, and *Display()*), one of which is replicated up to eight times (Figure 6c). The diagrams are easily modified. For example, to replicate *Geometry()*, the user need only select *replication* from *Geometry()*'s menu and specify the number of copies. This new parallel program will now run without any additional changes to the user's code.

Consider the parallel polynomial multiplication example of Figure 7a [21]. The program can be described as a *line* of two assets: a receptionist, *PolyMult()*, and a division, *Mult()*. *PolyMult()* reads in the coefficients of the polynomials to be multiplied. It then calls *Mult()* to recursively do the multiplication. The code for *Mult()* is shown in Figure 7a. To make this program run properly using Enterprise, the pointer parameters must be followed by an additional size parameter with an *INOUT* designation (not shown). These small changes violate the *non-intrusiveness* requirement.

Figure 7b shows the parallel structure of the program. Inside the double line rectangle is the expansion of the *enterprise* asset. Inside the dashed-line asset is the expansion of the *line* consisting of the two assets, *PolyMult()* and *Mult()*. The asset *Mult()* has been expanded as a *division*. Inside the *division* is another *division*. Here the *division* is replicated three times (because of three recursive calls to *Mult()*). The diagram shows the depth of recursion (two levels). Figure 7c shows the result of expanding the diagram into a standard call graph showing all of the processes (including the hidden Enterprise processes). The simple diagram of Figure 7c corresponds to a complex structure of 18 processes.

(Figure 7 should be placed here.)

An interesting property illustrated by the *PolyMult()* example is that Enterprise can execute an asset sequentially or in parallel at run-time. *Mult()* is recursive and if there are processes available to do the recursion in parallel, it is done in parallel. Once the recursion reaches the depth of the asset diagram tree, subsequent calls are processed sequentially.

Enterprise eliminates the possibility of certain types of common parallelization errors. For example, as mentioned earlier, it eliminates cyclic calls thus protecting users against a common cause of deadlocks. Errors such as waiting for a message (such as a reply) that will not be sent is checked by the system. Similarly, a missing connection error is prevented by checking asset calls to nonexistent assets. Also, the system handles packing and unpacking of parameters for message communication thus eliminating a common error in parallel programming.

The preceding has suggested that Enterprise comes close to satisfying the *separation of specification* objective. While largely true, there are two important places where this is violated. First, assets correspond to procedure/function calls in the user code. Changes in the asset diagram (for example, adding a new asset) must be reflected in the code, and visa-versa. Second, the asset diagram may force the user to restructure their code to achieve the desired parallelism. For example, replicating an asset isn't beneficial if there is only a single call to that asset. To maximize performance, the user might have to rewrite the code so that the asset gets called many times (by dividing the work of one call into multiple computationally-smaller calls).

7 Lessons and Experiences

There are several parallel programming systems that employ techniques similar to those found in FrameWorks/Enterprise (for example [22, 11, 23, 24, 25, 26, 10]). All these systems can be viewed as template-based. This section presents a critical evaluation of template-based parallel programming tools. It is intended to illustrate the large gap between current technology and what has to be improved before it can gain wide acceptance. Emphasis is placed on areas that require further research work.

7.1 Separation of Specification

The significance of separating (sequential) application program components from how these components interact has long been recognized. In early systems, component interaction was specified in separate text files [39]. The advent of workstation technology and their graphical user interfaces (GUI) greatly enhanced the ease, efficiency and effectiveness of specifying parallel structures [11, 24, 40].

Many of the systems that employ a separation of specifications between parallel structuring and application code are based on the data-flow model. Example systems are CODE [24], DGL [41], LGDF [42] and Paralex [43]. Typically, in these systems the programmer describes the data-flow using a graph, where nodes represent processes or programs and links represent the flow of data between nodes. A node can begin execution when all the links incident to that node have their inputs available. Some of these models also provide hierarchical resolution of parallelism [24, 40]; others don't [43, 42, 41]. In a pure data-flow model it is difficult to describe loops, self-loop arcs and static/dynamic node replication. For this reason, some systems modify the model to introduce these additional constructs. For example, CODE supports replicated nodes.

Several models based on control-flow that address the *separation* objective have emerged. Example systems include CAPER [44], PIE [10], and Parallel Utilities Library (PUL) [45]. The PIE system (Programming and Instrumentation Environment) supports implementation templates for master-slave, recursive master-slave, heap, pipeline and systolic multidimensional pipeline [10]. In PIE, a template can have another implementation template as part of it, thus facilitating the hierarchical resolution of parallelism. As another example, the parallel language PAL is a procedural language with a language construct called molecule [46]. A molecule can be used to define one of several types of parallel computation (SIMD, sequential, pipelined, data-flow, etc.). The PUL system [45] provides high-level templates for task as well as data parallelism. It also supports templates for parallel I/O.

Separate specification-based parallel computation models are also not limited to procedural programming languages. For example, Cole's algorithmic skeletons [9] and P3L [22] are designed using the functional programming model. Similarly, Strand uses logic programming to design its templates [47].

Some researchers advocate using application-independent parallel program skeletons not only for building parallel applications but also for educating programmers or documenting the solution strategies in parallel computing. A Programming Paradigm [27, 28] for parallel computing is defined as a class of algorithms that solve different problems but have the same control structure. A parallel program archetype [29] is a program design strategy for a class of parallel problems along with the associated program designs and example implementations. In both of these works, there is an added emphasis on enhancing developers' understanding of common classes of parallel problems.

The separation of specification between parallel structuring (meta-programming model) and the user code is important as it allows relatively independent evolution of the two. However, complete separation is (currently) not possible as the meta-program needs to connect to the user code at some point. This is true about Enterprise as well as all other similar systems that we are aware of.

Enterprise attempted to preserve the semantics of sequential C. Again, it was not possible to completely achieve this. There are semantic differences between the programming and meta-programming models. Also, the distributed memory model and futures force some subtle changes in semantics that can confuse some users. For example, in a MIMD model, each process has its own copy of the variables that may be declared as global variables in the original sequential program. This means that all information required by an asset, including access to the caller's global variables, must be added as extra parameters to the asset call. Depending on the application, this could require a major restructuring of the user's code. This is a source of many programming errors by first-time Enterprise users.

The above points illustrate that there are flaws in the Enterprise model. Similar weaknesses exist in other template-based models. All coarse-grained, distributed-memory systems that we are aware of require the user to make some changes to their sequential code for parallelization. The ideal orthogonal relationship between sequential code and parallel specifications is hard to achieve since the needs of the programming model and the meta-programming model are sometimes contradictory.

7.2 Trade-offs

An important weakness of any template-based model is that not all parallel algorithms can be readily expressed using the available repertoire of templates (affecting the *utility* of the system). For example, an algorithm that relies on a group of processes using peer-to-peer communication (as used, for example, in mesh problems) cannot be supported using current Enterprise assets. Although this problem can be alleviated somewhat as new assets are designed, it will never really disappear. There are two main reasons for this. First, it is probably impossible to predetermine a set of templates that can represent an arbitrary communication topology without reducing the level of the templates to a "connect-the-dots" approach. Second, there are certain trade-offs involved in designing a high-level parallel programming system. The difficulty in supporting peer-to-peer communication lies not with the implementation, but rather with the conflict that in such a system it may no longer be possible to offer any correctness guarantees such as an absence of deadlocks.

7.3 Performance

Often a solution generated by a high-level tool, such as Enterprise, may not achieve the same performance as a solution hand-crafted by an expert using a low-level communication library such as PVM. There are several reasons for this performance degradation:

1. An Enterprise template may include a hidden process. For example, the department asset is implemented via a representative process that manages various assets in the department. Although the additional process means that there is some performance overhead, its presence is desirable because it avoids splitting the code between several interacting processes and duplicating this code in all the assets. This would result in a poorly engineered system that would be difficult to understand and maintain. However, a particular instance of this type of application may be hand-crafted by a user without a representative process thus achieving better efficiency.

2. Asset calls sometimes transfer more data than necessary. The user knows exactly how much data to pass and can optimize a program to minimize it. Enterprise does not have the same intimate knowledge of the application as the user, and will always err on the side of transferring too much.

3. The Enterprise-generated code includes a lot of error checking. A hand-crafted application may want to eliminate most of it.

4. Enterprise provides facilities for collecting debugging and performance monitoring information. Even if these facilities are not used, they still create (small) run-time overhead.

5. Being a high-level system, Enterprise deals with general structures rather than specific instances. For example, the divide-and-conquer asset (*division*) uses generalized code that is valid for any specified values for depth and width. This results in some overhead in the form of extra code.

6. A user can use PVM to construct arbitrary communication graphs, exploiting communication shortcuts to improve performance. This is not possible in Enterprise or FrameWorks.

7. Enterprise is built on PVM. Therefore, even though it may be possible to apply certain performance enhancements to Enterprise, performance of an Enterprise application cannot exceed that of the best possible PVM implementation.

In summary, the trade-off is between better software engineering in exchange for possibly slower execution performance. The degree to which other similar systems [11, 24, 45] would suffer the performance loss would depend on whether these systems have similar reasons or not.

Although speed-up is only one factor in judging the utility of a parallel programming system, there is a segment of the parallel programming community that demands near peak performance from their applications. However, with the availability of relatively inexpensive multi-processor machines and the wide-spread use of networked single-processor workstations, more and more people are turning towards parallel computing. For such users, a shorter learning curve, ease of program design, development and debugging are just as important as speed-up. A tool that quickly achieves a performance improvement, even though it may provide less than peak performance, may be quite acceptable. The debate over peak performance is akin to a similar debate in the sequential programming world over the use of high-level compilers and fourth-generation tools instead of highly efficient assembly language programming. Hardly anyone now questions the utility of high-level language compilers.

7.4 Usability

A motivation for developing FrameWorks and Enterprise was to construct a parallel programming system with a high degree of *usability*. The system should be easy to learn, easy to use and, because of the high-level templates, capable of constructing correct parallel programs quickly. Our experience with Enterprise, as well as feedback from the user community, seemed to support these claims. Still, it was felt that some comparative assessment of Enterprise should be made to determine how well it fared against, for example, a low-level message-passing library.

In 1993, we conducted a controlled experiment in consultation with a cognitive psychologist [48, 20]. Half of the graduate students in a parallel/distributed computing class solved a problem using Enterprise while the rest used NMP [38], a PVM-like library of message-passing primitives. The student accounts were monitored to collect statistics on the number of compiles, program executions, editing sessions and login hours. When the students submitted their assignment for grading, the quality of their solution (speedup) was measured and the number of lines of code written was counted. Full details of the experiment can be found in [20].

The results of this experiment were a mixed bag of expected results as well as surprises. The statistics support our initial expectation that students would do less work with Enterprise, but get a more efficient solution with NMP. Enterprise students wrote 66% fewer lines of code than the NMP students, in addition to doing fewer edits, compiles and test runs. However, the NMP solutions ran 27% faster. One surprising result was that even though Enterprise users wrote less code, they had 26% more login hours than the NMP students. A detailed examination of the logged data revealed three main causes for this:

1. Better tool *support*: Enterprise users frequently used the animation feature of the system to replay a computation. This tool provided useful run-time information, but was quite slow to run.
2. *Usability*: The Enterprise compiler preprocessed the user's code several times before generating C code to be compiled. Consequently, compilations were at least 4-fold slower, something that all users found frustrating.
3. *Performance*: Since NMP performance was better, Enterprise users spent more time trying to improve the performance of their solution.

A second experiment was conducted in 1995 to assess three tools: Enterprise, PVM and PAMS (a commercially available tool that allows loop iterations to be done in parallel [8]). A graduate class of 20 students was divided into three groups, each group using a different parallel programming tool to do each assignment (graph theory related, sorting and tree searching). The students were asked to evaluate the tool used.

As expected, PVM solutions produced the best performance (on two of the assignments it was significantly better than Enterprise/PAMS), with Enterprise and PAMS producing slower, but comparable, results. The code inserted into the sequential program by PVM users averaged over 100 lines more than code inserted by Enterprise/PAMS users. Superficially, it seems like an obvious trade-off: better performance for more programming effort expended. However, things were not as they seemed: Enterprise/PAMS users spent more login time working on their assignments, typically from 33% to 100% additional hours. Again a seemingly paradoxical result appears: students using the high-level tools wrote less code but spent more time developing it. Why? Gradually three reasons emerged:

1. *Performance*: Graduate students are highly competitive. Before the start of the experiment, they were warned that some tools might perform significantly better than others on a particular assignment. They were encouraged to be competitive (get the best speedups) within the group that was using the same tool, and not to compete with students using the other tools. Despite this, many of the Enterprise/PAMS students tried very hard to get PVM-like speedups. They tried numerous clever

ways of circumventing the programming model, but were rarely rewarded with better performance. Enterprise and PAMS students said that for each assignment there was an "obvious" way to parallelize the program, and this they could do quickly and easily. However, after their initial success, they found it very hard to improve performance.

2. *Understanding*: Many students had difficulty grasping the notion that Enterprise would take care of "everything" for you. Templates hide a lot of detail from the user. If an asset made a call to another asset, even though the code and semantics of the call look sequential, the students knew it was being done in parallel. They felt they needed to understand how Enterprise worked which, of course, is defeating part of the purpose of a high-level tool.

3. *Language*: Both PAMS and Enterprise make subtle changes to the host programming language (C) semantics. Even though these semantic differences were properly documented in the manuals, this was still a source of confusion for some students. Programming in PVM, in contrast, was as easy as writing sequential code to many students. Even though they had to write more PVM code, the students found that they needed to know less than 10 PVM routines and, once these routines were learned, writing parallel code was easy.

The data suggests that the students most dissatisfied with Enterprise/PAMS were the ones who did their first assignment with PVM. In PVM, the user has complete control over the parallelism and can do whatever is desired. When these users tried Enterprise/PAMS, they quickly became frustrated at the lack of control they had. In the final class evaluation of the tools, lack of user control over the parallelism was cited as the biggest disadvantage of Enterprise/PAMS. We could summarize the implications of these experiments as follows:

1. These experiments and feedback from WWW users demonstrate that the Enterprise model and its support tools can be used to develop parallel programs. All too often, research tools are evaluated solely by the research group that developed the technology. There is a concern in the parallel programming community that the functionality and usability of parallel programming environments is often never validated [24].

2. If the goal of a user is to quickly generate an initial version of a parallel application, Enterprise (and PAMS) can be termed as "easy-to-use" systems as compared to PVM. However, if the goal is to get a parallel solution where performance is the overriding concern, communication libraries may provide a

better alternative. For reasons outlined earlier, it is very hard for Enterprise to generate code that is as efficient as the one generated by hand-crafted solutions.

3. Some users, particularly those who have worked with the low-level tools, do not like to lose the control and the flexibility that such tools provide. In systems like FrameWorks or Enterprise, where a user must develop a parallel program using only the high-level constructs provided by the system, the lack of *open*ness may be counterproductive. A possible solution might be to have more *open* and *extendible* systems where a user may use templates if desired, but can also access low-level primitives for efficiency and flexibility. We discuss the issue of *open* and *extendible* systems further in Section 9.

4. A high-level system should not introduce changes in the semantics of the underlying sequential language. In trying to preserve sequential compatibility, both Enterprise and PAMS introduce subtle changes to the semantics. These changes make the systems harder to learn and understand and, therefore, make it more difficult to develop and debug applications. Subtle changes were harder for the students to deal with; obvious changes (such as new keywords or library calls) were easier, since they would be more explicit in the code.

7.5 Template-Based Models and Low-Level Communication Libraries

Enterprise has a simple interface that allows it to use a variety of communication packages (PVM, ISIS and NMP). Enterprise can be viewed as a software layer on top of, for example, PVM. The question arises as to what the user gains and loses by moving to a higher level of abstraction in their code.

There are two main goals of the Enterprise system: to create a high-level programming environment that is easy to use, and to promote code reuse by encapsulating parallel programming code into templates. For example, Enterprise's model allows the user to achieve a near complete *separation* of specification. There is nothing in the user's code that indicates it is intended for parallel execution (other than optional parameter macros for performance). The use of a pre-compiler allows the Enterprise system to automatically insert communication, parameter packing and synchronization code into the user's application. In contrast, with PVM the user must explicitly address these issues by inserting PVM library calls into the code (violating the *non-intrusiveness* objective). It is the user's responsibility to structure the code so that a compiler flag can be used to selectively include/exclude the parallel code.

Enterprise offers the user additional benefits. For example, the model allows for the *hierarchical* use of the templates, thus ensuring deadlock-free structuring of applications. Also, the user has the assurance that the generated code for the specified structures is correct. Both points contribute to the *correctness* objective.

In moving to a higher-level model, such as Enterprise, the user has lost something. Most noticeable is the possible decreased *performance*. Message passing libraries allow for more flexibility; the user can easily tune a system to maximize performance. Further, these libraries have a large support infrastructure that has resulted in them being made available on most major platforms (excellent *portability*).

The choice between PVM and a higher-level tool is not easy. The decision can be simplified to a tradeoff between execution performance and software engineering. High-level parallel programming tools have the potential to enable users to build parallel applications more quickly and reliably. In return, they may have to accept (slightly) worse performance.

8 Expanding the Role of Templates

Most template-based parallel programming models use their templates to represent control flow. However, there are several more areas where the application of the template-based approach holds promise.

1. Templates for parallel I/O: There are a number of commonly occurring parallel I/O access patterns. These patterns can be abstracted into a set of useful templates. An Enterprise-based implementation involves the user annotating (through the asset diagram) each file with an appropriate template [49]. For example, the user can designate a file to be a *diary* or a *newspaper*, again using analogies to describe the data access patterns.

2. Templates for shared memory: Work is proceeding on enhancing Enterprise with distributed shared memory. Users specify the shared memory and its access templates via the user interface and the compiler analyses the user's code to insert locks in the appropriate places. Templates correspond to different access protocols, including facilities to preserve sequential semantics, guarantee deterministic execution or allow for chaotic results [50].

3. Templates for Data Parallelism: Templates can be used to describe alignment and distribution of data on processors (as in HPF [6]). The system can then generate SPMD code for the fine-grain data-parallel solution for the given function or segment of code. However, this approach is not suitable for applications that require redistribution or realignment of data during execution.

9 A Next Generation Tool

Templates represent a powerful abstraction mechanism. We believe templates have the potential to make as strong an impact on the art of parallel programming as macros and code libraries. However, from our experiences with FrameWorks and Enterprise, we have learned a number of lessons that must be remembered when developing new template-based tools:

1. *Open Systems*: Enterprise provides a high-level parallel programming model that the user *must* use. There are no facilities for the user to step back from the model to access lower-level primitives to achieve better performance, or to accommodate an application for which a suitable template is not available. For example, even though Enterprise generates PVM code, this code is hidden from the user. There is no easy way to use Enterprise to generate a correct PVM program, and then incrementally tune this program to achieve better performance. A high-level template-based tool must allow the user the possibility of accessing lower-level primitives. Also, it should be possible to develop an application partially with the use of templates and partially by using low-level communication primitives [51].

2. *Extendibility*: FrameWorks and Enterprise support a fixed number of templates. It is difficult add new templates to the system. An important step towards enhancing the *utility* of a template-based model would be to design a system that provides a standard interface for attaching templates to the user code. In such a system, it may be possible for the user to develop new templates. As long as the templates are mutually independent, it should be possible to integrate them into the rest of the system. This would result in a system that is *extendible* and can support a large number of templates [51].

3. *Portability*: It is imperative to continue building on top of existing, established technology. Some *de facto* standards seem to be emerging. For example, PVM (and possibly MPI soon) is currently adequate as the lowest-level building block. PICL seems to be a popular choice for parallel program instrumentation [52]. Given the significant effort required to build a parallel programming system, it seems foolhardy to continue to invent, when one can reuse. Also, the widespread availability of these tools on different platforms enhances the portability of the system.

4. *Language*: Many parallel programming tools make (subtle) changes to the semantics of an existing sequential language. We believe this is a mistake. Changing a programming language's semantics can increase the user's learning curve and result in difficulties in understanding and debugging parallel code.

5. Importance of Compiler Technology: Our research would greatly benefit from better compiler technology. For example: some of the semantic confusion in Enterprise could be eliminated; static analysis of the code can do a better automatic job of code reorganization to improve concurrency and delay synchronization, thereby improving performance; compilers can uncover data dependencies, possibly uncovering programming errors at compile-time rather than at run-time; flow control analysis can identify communication patterns that can assist in the initial process-processor mapping (Orca, for example, uses compile-time analysis to help distribute the data [53]).

6. Trade-Offs: Should we build a tool for the inexperienced user or the experienced user? For example, it is conceivable to build an *open* and *extendible* system such as outlined in item 1 and 2 above. However, in such a system, it may no longer be possible to give the correctness guarantees that Enterprise offers. The requirements of users vary with their skill and experience levels. For the former, simplicity of the model and ease of use are the most important considerations. For the latter, performance is often the only metric that matters.

10 Conclusions

Who are the potential users of parallel computing technology? There will always be a user community that uses parallel computing to squeeze even the last nano-second of performance out of a machine. We claim this group is a very small percentage of the potential user community. Local area networks of workstations are commonplace and the popularity of low-cost multi-processor shared-memory machines is rapidly growing. However, few people take advantage of the parallelism in these architectures. Many people want their programs to run faster but are unwilling to invest the time necessary to achieve this.

Consider the sequential world. With a simple optimization flag on a compile (-O), the user can get increased performance. To further improve performance, the user must read the compiler manual page to find any other optimization options that might be applicable (for example, in-lining functions). If better performance is still required, users will take the next step and use a tool to analyze their programs (such as an execution profiler). They will use this feedback to modify their code.

For most users, sequential program improvement stops at the compiler level. Ideally, the same should be true for coarse-grained parallel program development (such as is seen with vectorizing compilers). Given that compilation techniques are still in their infancy for coarse-grained applications, the next logical step is to provide a tool that allows users to parallelize their application with minimal effort. Template-based models offer real prospects of making this a reality.

Rather than putting forward yet another model for building parallel applications, this paper was meant to consolidate an existing approach to parallel programming. Usability experiments of Enterprise have added a new dimension to our understanding of how programmers with little or no experience in parallel computing build their parallel applications. There are a number of researchers working on similar high-level systems for parallel programming. A strong research interest in this area is evident from the fact that recently a mailing list was started on the Internet to discuss issues specific to skeleton or template-based approaches (the list has over 150 members, skeletons@dcs.ed.ac.uk). We hope that our experience in developing two such models into working systems, as well as the results of our experiments in estimating the usability of parallel programming systems, will be useful to researchers and practitioners in this area.

Perhaps the most damning comment on the state-of-the-art of parallel programming tools for coarse-grained parallelism is the continued wide-spread popularity of PVM/MPI. No matter how clever the implementation, template-based parallel programming tools cannot achieve the performance of hand-crafted solutions. However, there comes a point where the software engineering considerations gained by using a high-level tool out-weigh the incremental gain in performance. Unfortunately, we are not yet at that stage: the gap between the performance possible from a tool such as FrameWorks/Enterprise versus PVM is large enough that serious users will continue to program in PVM for the foreseeable future.

We have identified several areas (in Sections 8 and 9) where effort is necessary to enhance the usability of the template-based systems. Work on several of these issues is in progress [50, 49, 51]. Template-based techniques alone may not be enough to provide an easy to use, high-level parallel programming system that supports quick prototyping and restructuring of parallel applications, and supports code reuse. However, we believe that template-based techniques will play a significant role in building the ideal parallel programming systems of the future.

11 Acknowledgments

The constructive comments from Ian Parsons, Greg Wilson, and Stephen Siu are appreciated. This research was conducted using grants from the Natural Sciences and Engineering Research Council of Canada (OGP8173 and OGP0155467) and IBM Canada Ltd.

References

[1] H. Bal, M. Kaashoek, and A. Tannenbaum. "Orca: A Language for Parallel Programming of Distributed Systems". *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992.

[2] J. Feo, D. Cann, and R. Oldehoeft. "A Report on the Sisal Language Project". *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990.

[3] G. Geist and V. Sunderam. "Network-Based Concurrent Computing on the PVM System". *Concurrency: Practice and Experience*, 4(4):293–311, 1992.

[4] R. Butler and E. Lusk. "Monitors, Messages, and Clusters: The P4 Programming System". *Parallel Computing*, 20(4):547–564, 1994.

[5] D. Walker. "The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers". *Parallel Computing*, 20(4):657–673, 1994.

[6] D.B. Loveman. "High Performance Fortran". *IEEE Parallel and Distributed Technology*, 1(1):25–42, February 1993.

[7] A. Grimshaw, W.T. Strayer, and P. Narayan. "Dynamic Object-Oriented Parallel Processing". *IEEE Parallel and Distributed Technology*, 1(2):33–27, 1993.

[8] W. Karpoff and B. Lake. "PARDO - A deterministic, Scalable Programming Paradigm for Distributed Memory Parallel Computer Systems and Workstation Clusters". In *Supercomputing Symposium '93, Calgary*, pages 145–152, 1993.

[9] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Programming.* MIT Press, Cambridge, Mass., 1989.

[10] Z. Segall and L. Rudolph. "PIE: A Programming and Instrumentation Environment for Parallel Processing". *IEEE Software*, 2(6):22–37, 1985.

[11] A. Baguelin, J. Dongarra, G. Giest, R. Manchek, and V. Sunderam. "Graphical Development Tools for Network-Based Concurrent Computing". In *Supercomputing'91*, pages 435–444, 1991.

[12] M. Green and J. Schaeffer. "Frameworks: A Distributed Computer Animation System". In *Canadian Information Processing Society, Edmonton*, pages 305–310, 1987.

[13] A. Singh, J. Schaeffer, and M. Green. "Structuring Distributed Algorithms in a Workstation Environment: The FrameWorks Approach". In *International Conference on Parallel Processing*, volume II, pages 89–97, 1989.

[14] A. Singh, J. Schaeffer, and M. Green. "A Template-Based Tool for Building Applications in a Multicomputer Network Environment". In D. Evans, G. Joubert, and F. Peters, editors, *Parallel Computing 89*, pages 461–466. North-Holland, Amsterdam, 1989.

[15] A. Singh, J. Schaeffer, and M. Green. "A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstations". *IEEE Transactions of Parallel and Distributed Systems*, 2(1):52–67, January 1991.

[16] P. Iglinski, S. MacDonald, D. Novillo, I. Parsons, J. Schaeffer, D. Szafron, and D. Woloschuk. "Enterprise User Manual, Version 2.4". Technical Report No. 95-02, Department of Computing Science, University of Alberta, 1995.

[17] G. Lobe, D. Szafron, and J. Schaeffer. "The Enterprise User Interface". In *TOOLS 11 (Technology of Object-Oriented Languages and Systems)*, pages 215–229, 1994.

[18] S. MacDonald, D. Szafron, and J. Schaeffer. "An Object-Oriented Run-time System for Parallel Applications". In *TOOLS 14 (Technology of Object-Oriented Languages and Systems)*, to appear 1996.

[19] J. Schaeffer and D. Szafron. "Software Engineering Considerations in the Construction of Parallel Programs". In *High Performance Computing: Technology and Applications*, pages 271–288. Elsevier Science Publishers B.V., Netherlands, 1995.

[20] D. Szafron and J. Schaeffer. "An Experiment to Measure the Usability of Parallel Programming Systems". *Concurrency: Practice and Experience*, 8(2):146–166, 1996.

[21] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. "The Enterprise Model for Developing Distributed Applications". *IEEE Parallel and Distributed Technology*, 1(3):85–96, 1993.

[22] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. "P3L: A Structured Approach to High-Level Parallel Language, and its Structured Support". *Concurrency: Practice and Experience*, 7(3):225–255, 1995.

[23] A. Bartoli, P. Cosini, G. Dini, and C.A. Prete. "Graphical Design of Distributed Applications Through Reusable Components". *IEEE Parallel and Distributed Technology*, 3(1):37–51, 1995.

[24] J.C. Browne, M. Azam, and S. Sobek. "CODE: A Unified Approach to Parallel Programming". *IEEE Software*, pages 10–18, July 1989.

[25] J.C. Browne, S. Hyder, J. Dongarra, K. Moore, and P. Newton. "Visual Programming and Debugging for Parallel Computing". *IEEE Parallel and Distributed Technology*, 3(1):75–83, 1995.

[26] L. Schafers, C. Scheidler, and O. Kamer-Fuhrmann. "TRAPPER: A Graphical Programming Environment for Industrial High-Performance Applications". In *Parallel Architectures and Languages Europe*, pages 403–413, 1993.

[27] P. Brinch Hansen. "Studies in Computational Science: Parallel Programming Paradigms". *Prentice Hall, Inc*, 1995.

[28] P. Brinch Hansen. "Search for Simplicity: Essays in Parallel Programming". *IEEE Computer Society Press*, pages 422–446 (Chapter 22), 1996.

[29] K. M. Chandy. "Concurrent Program Archetypes". In *International Parallel Programming Symposium*, 1994. Keynote Address.

[30] M. Danelutto and S. Pelagatti. "Parallel Implementation of FP Using a Template-Based Approach". In *Proc. of the 5th Int, Workshop on Implementation of Functional Languages*, pages 7–21, Sept. 1993.

[31] S. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of 4.3 BSD Unix Operating System.* Addison-Wesley Publishing Company, Inc., 1990.

[32] G. Andrews, R.A. Olsson, M.A. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. "An Overview of the SR Language and Implementation". *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, 1988.

[33] H.A. Simon and W.G. Chase. "Skill in Chess". *American Scientist*, 61:394–403, August 1973.

[34] A. Singh. *A Template-Based Approach to Structuring Distributed Algorithms Using a Network of Workstations.* PhD thesis, Dept. of Computing Science, University of Alberta, 1992.

[35] A.R. Halstead. "MultiLisp: A Language for Concurrent Symbolic Computation". *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[36] D. Gannon, J. Lee, B. Shei, S. Sarukkai, S. Narayana, and N. Sundaresan. "SIGMA II: A Tool Kit for Building Parallelizing Compilers and Performance Analysis Systems". In *Programming Environments for Parallel Computing*, pages 17–36, North Holland, Netherlands, 1992.

[37] K. Birman, A. Schiper, and P. Stephenson. "Lightweight Causal and Atomic Group Multicast". *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

[38] T. Marsland, T. Breitkreutz, and S. Sutphen. "A Network Multiprocessor for Experiments in Parallelism". *Concurrency: Practice and Experience*, 3(1):203–219, 1991.

[39] J.C. Browne, A. Tripathi, S. Fedak, A. Adiga, and R. Kapur. "A Language for Specification and Programming of Reconfigurable Parallel Structures". In *International Conference on Parallel Processing*, pages 142–149, 1982.

[40] T.G. Lewis and Rudd W.G. "Architecture of the Parallel Programming Support Environment". In *IEEE COMPCON*, pages 589–594, 1990.

[41] R. Jagannathan, A.R. Downing, W.T. Zaumen, and R.K.S. Lee. "Dataflow Based Technology for Coarse-Grain Multiprocessing on a Network of Workstations". In *International Conference on Parallel Processing*, pages 209–216, August 1989.

[42] D.C. DiNucci and R.G. Babb II. "LGDF Parallel Programming Model". In *IEEE COMPCON*, pages 102–107, 1989.

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

[44] B. Sugla, J. Edmark, and B. Robinson. "An Introduction to the CAPER Application Programming Environment". In *International Conference on Parallel Processing*, pages 107–111, August 1989.

[45] L. Clarke, R. Fletcher, S. Trevin, R. Bruce, and S. Chapple. "Reuse, Portability and Parallel Libraries". In *Programming Environments for Massively Parallel Distributed Systems*, pages 171–182, Birkhauser Verlag, Basel, Switzerland, 1994.

[46] Z. Xu and K. Hwang. "Molecule: A Language Construct for Layered Development of Parallel Programs". *IEEE Transactions on Software Engineering*, 15(5):587–599, May 1989.

[47] I. Foster and S. Taylor. "Strand: A Practical Parallel Programming Tool". In *North American Conference on Logic Programming*, M.I.T. Press, 1989.

[48] D. Szafron and J. Schaeffer. "Experimentally Assessing the Usability of Parallel Programming Systems". In *Programming Environments for Massively Parallel Distributed Systems*, pages 195–201, Birkhauser Verlag, Basel, Switzerland, 1994.

[49] I. Parsons, R. Unrau, J. Schaeffer, and D. Szafron. "A Template Approach to Parallel I/O". *Parallel Computing*, to appear in 1996.

[50] D. Novillo. "High-Level Representations for Distributed Shared Memory". Technical report, Department of Computing Science, University of Alberta, 1995.

[51] S. Siu. "Openness and Extensibility in Design-Pattern-Based Parallel Programming Systems". Master's thesis, Electrical and Computer Engineering Dept., University of Waterloo, 1996.

[52] 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.

[53] H. Bal and M. Kaashoek. "Object Distribution in Orca using Compile-Time and Run-Time Techniques". In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 162–177, 1993.

```
/* Definition of structures used by functions */

/* geometry.h contains the structure for polytbl, objtbl, etc. */
#include "geometry.h"

#define MAXIMAGES 120

struct generategeometry {
        int imagenumber;
        struct obj objtbl[MAXOBJ];
};

struct geometrydisplay {
        int imagenumber, npoly;
        struct polygon polytbl[MAXPOLY];
};

main() /* Generate */
{
        struct generategeometry work;
        int image;

        for( image = 0; image < MAXIMAGES; image++ )
        {
                /* loop through images */
                ComputeObjects( work ); /* Modeling and motion computation */
                Geometry( work ); /* Send further processing to Geometry */
        }
}

Geometry( struct generategeometry work )
{
        struct geometrydisplay frame;

        DoConversion( work, frame ); /* View transformation on the image */
        Display( frame ); /* Send data to Display for further processing */
}

Display( struct geometrydisplay frame )
{
        DoHidden( frame ); /* Hidden surface removal and anti-aliasing */
        WriteImage( frame ); /* Store image on disk */
}
```

Figure 1: Structure of the *Animation* Application.

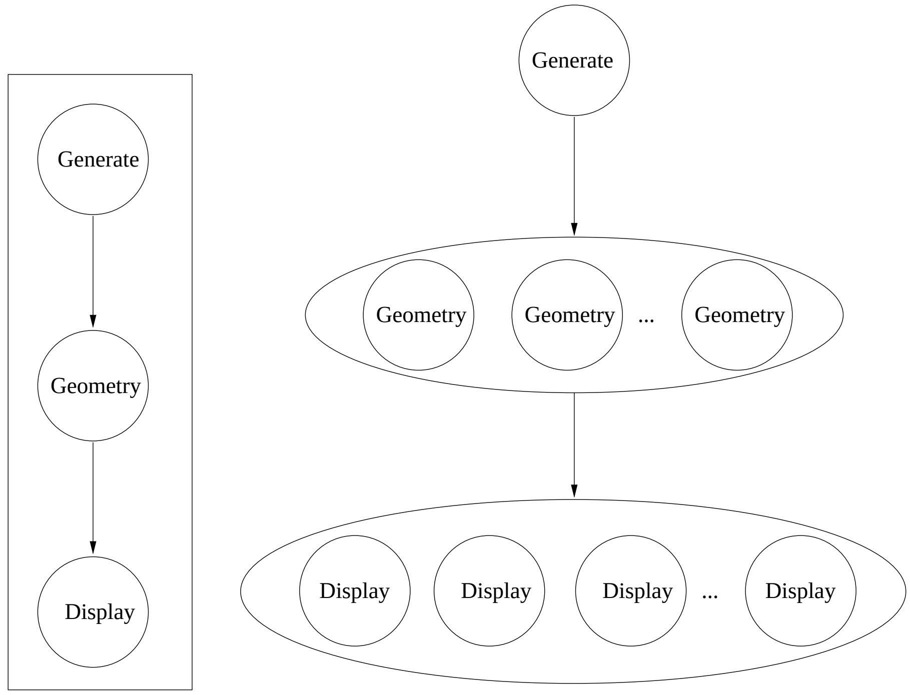


Figure 2a: A Parallel Version

Figure 2b: A Parallel Version with Replications

Figure 2: Potential Parallelizations of *Animation*.

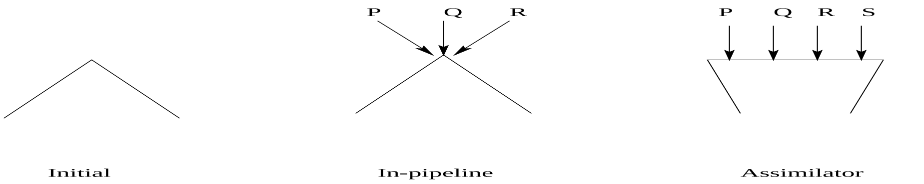


Figure 3a: Input Templates.

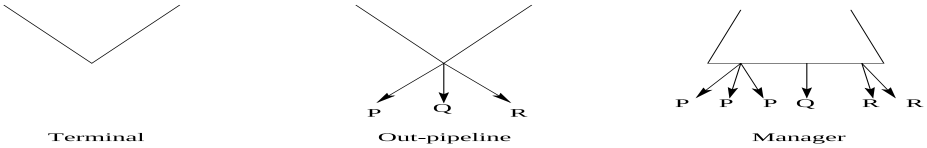


Figure 3b: Output Templates.

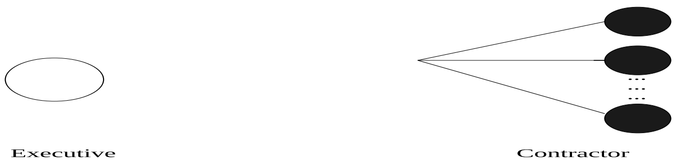


Figure 3c: Body Templates.

Figure 3: FrameWorks Templates.

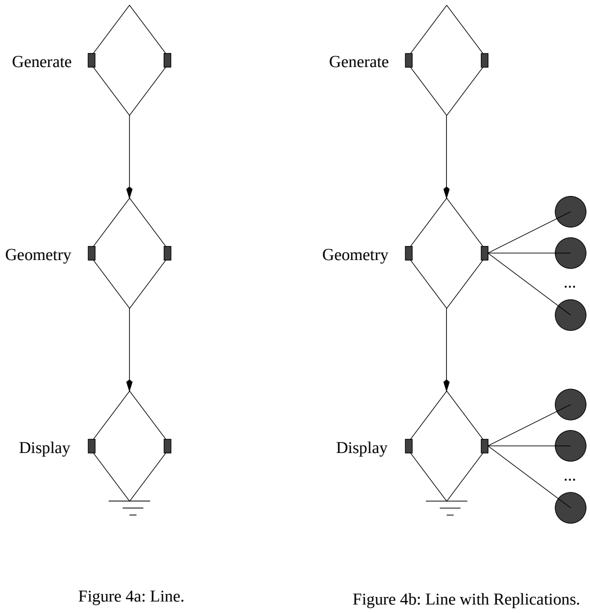


Figure 4: Parallel Versions of the *Animation* Program Using FrameWorks.

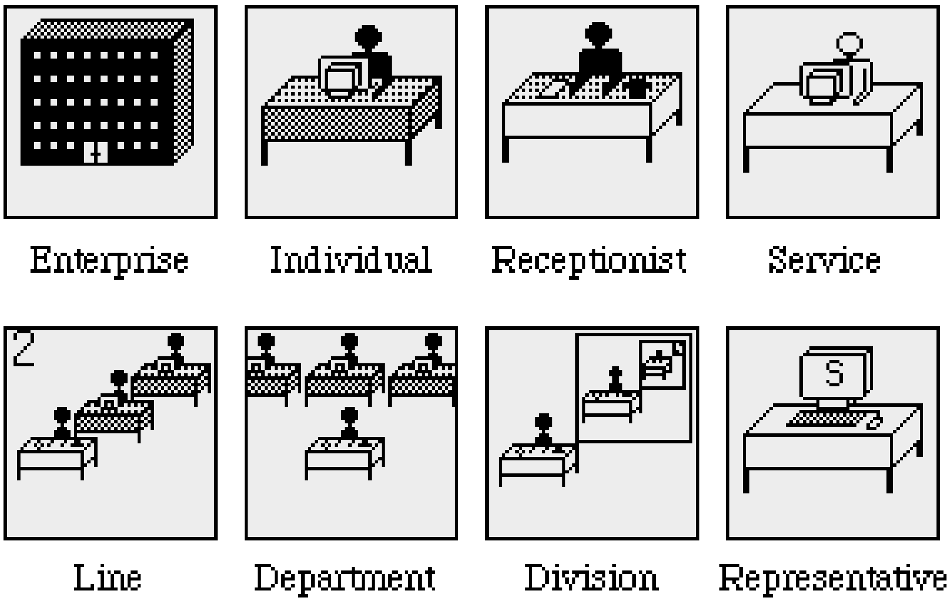
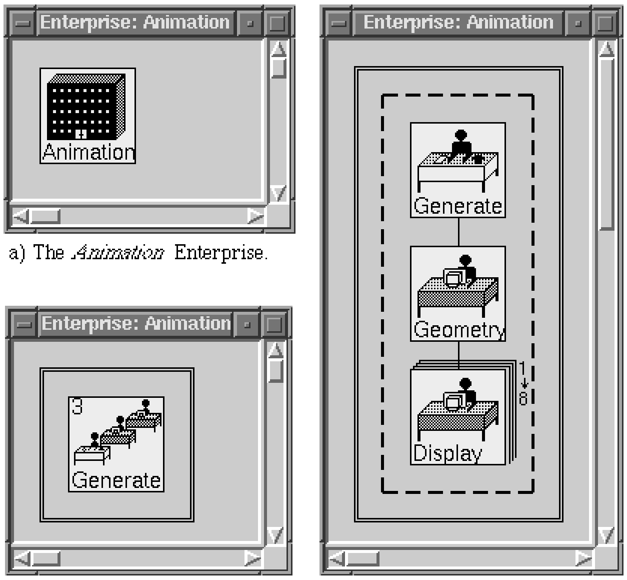


Figure 5: Enterprise Assets.



a) The *Animation* Enterprise.

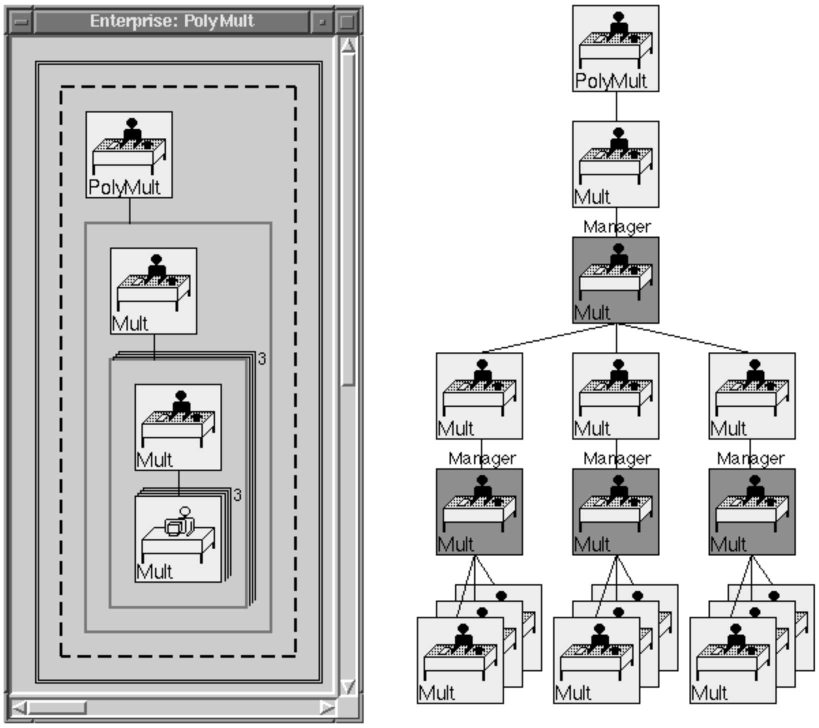
b) The *Animation* Line.

c) The Fully Expanded *Animation*.

Figure 6: The *Animation* Program in Enterprise.

```
/* 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;
}
```

(a) Polynomial Multiplication Pseudo-Code.



b) Fully Expanded *PolyMult* .   **c)** *PolyMult* Process Diagram.

Figure 7: Polynomial Multiplication in Enterprise.