The Enterprise Distributed Programming Model

Duane Szafron, Jonathan Schaeffer, Pok Sze Wong, Enoch Chan, Paul Lu and Carol Smith

Dept. of Computing Science, University of Alberta, Edmonton, Alberta, CANADA T6G 2H1

Abstract

Workstation environments have been in use for more than a decade. Although a network of workstations represents a large amount of aggregate computing power, single users often cannot utilize these resources for their applications. *Enterprise* is a programming environment for designing, coding, debugging, testing, monitoring, profiling and executing programs in a distributed hardware environment. Programs written using *Enterprise* look like familiar sequential C code; the parallelism is expressed graphically. The system automatically inserts the code necessary to handle communication, synchronization and fault tolerance, allowing the rapid construction of correct distributed programs. *Enterprise* programs run on a network of computers, absorbing the idle cycles on machines. The system supports load balancing, limited process migration, and dynamic distribution of work in environments with changing resource utilization. This paper concentrates on the user's view of programming in *Enterprise*.

Keyword Codes: D.2.6; D.1.3; C.2.4
Keywords: Programming Environments; Concurrent Programming; Distributed Systems

1. Introduction

Enterprise is a programming environment for designing, coding, debugging, testing, monitoring, profiling and executing programs in a distributed hardware environment. It represents the evolution of our successful *Frameworks* system [14][15]. Programs written in the *Enterprise* environment look like familiar sequential C code since the parallelism is expressed graphically. The system automatically inserts all code necessary to handle communication, synchronization and fault tolerance, allowing the rapid construction of correct distributed programs. This bridges the complexity gap between distributed and sequential software. *Enterprise* programs run on networks, absorbing the idle machine cycles. The system supports load balancing, limited process migration, and dynamic distribution of work in environments with changing resource utilization. *Enterprise* offers a cost-effective method for increasing the productivity of programmers and the throughput of existing resources.

This paper concentrates on the model that the programmer uses for writing distributed applications. A discussion of the implementation and run-time issues (such as process-processor assignment, process migration, load balancing, debugging, etc.) can be found in [7]. From the user's point of view, *Enterprise* has a number of features that distinguish it from other parallel and distributed program development tools (see Section 4):

- Programs are written in a sequential programming language (C) that is augmented by new semantics for procedure/function calls that allows them to be executed in parallel. The semantic changes are simple and have the important property that parallel code is indistinguishable from sequential code. Users do not deal with implementation details such as communication and synchronization. Instead, *Enterprise* automatically inserts all of the necessary communication protocols into the user's code.
- *Enterprise* can automatically generate these protocols because most large-grained parallel programs make use of a small number of regular techniques, such as pipelines,

This is a pre-print of a copyrighted article in IFIP Transactions A11: Progamming Environments for Parallel Computing, 1992, pp. 67-76.

master/slave processes, divide-and-conquer, etc. In *Enterprise*, the user specifies the desired technique at a high level by manipulating icons using the graphical user interface. The user-written code that implements the parallel procedure is independent of the parallelization technique selected (although the code generated by *Enterprise* certainly is not). The decoupling of the procedure that is to be parallelized and the parallelization technique allows applications to be easily adapted to a varying number and type of processors, usually without changing user-written code. It also provides a simple mechanism for experimentation and evaluation of how the various parallelization techniques fare on the user's particular application.

- For simplicity in expressing parallelism, *Enterprise* uses an analogy between the structure of a parallel program and the structure of an organization. The analogy eliminates inconsistent analogies used in the past (pipelines, masters, slaves, etc.) and replaces it with a uniform set of *assets* (contracts, departments, individuals, etc.). An organizational analogy was chosen because organizations are common and inherently parallel. This allows the programmer a different (but familiar) model for designing parallel programs.
- Several of the parallelization techniques supported by *Enterprise* can distribute work dynamically in environments with changing resource utilization. For example, a *contract* can be used to distribute work to a variable number of identical subordinates. A contract uses as many idle machines as possible to help complete the task. During peak hours, a program may only be able to use a handful of processors, while in the evening many more may be available to help fulfill the contract.
- In most parallel/distributed computing tools, the user is required to draw communication graphs. The user usually draws a diagram connecting nodes (processes) by arcs (communication paths). In *Enterprise*, a similar diagram is created, but the user is spared the tedium of drawing the details. Instead, the user need only edit the diagram by coercing and expanding nodes that represent assets. Coercion provides the user with a high-level technique to alter the method that a process (asset) uses to communicate with its neighbors and corresponds to the choice of a common parallelization technique. Expanding a node allows the user to explore the hierarchical structuring of the application.

Using the graphical user interface, the user manipulates a diagram of the parallel computation and writes sequential code that is devoid of any parallel constructs. Based on the user's diagram, *Enterprise* determines where to insert the parallel code. It then compiles the routines, dynamically assigns processes to processors and establishes the necessary connections. Processes run in the background, taking advantage of idle machines and recognizing when machines become heavily loaded. In this way we can keep the user community happy, while having applications profitably using machines that would otherwise be idle.

Section 2 contains a typical *Enterprise* session, illustrating the model and the graphical user interface. Section 3 describes the two key components of the *Enterprise* model: the semantics of the sequential code that the user writes, and the kinds of parallelism (assets) supported. Section 4 discusses other parallel programming environments and contrasts them with *Enterprise*. The current status of *Enterprise* is discussed in Section 5.

2. Program Design in Enterprise

This section presents a simple example of how *Enterprise* can be used to construct a distributed program. Consider an animation program that displays a group of fish swimming across a display screen. There are three fundamental operations in the program: *Model*, *PolyConv* and *Split* with the following functionality:

- *Model*: Computes the location and motion of each object in a frame, stores the results in a file, calls *PolyConv* to process the frame and proceeds to the next frame.
- *PolyConv*: Reads a frame from the disk file, performs some data format transformations, viewing transformations, projections, sorts and calls *Split*, passing it a transformed frame and a sequence number.
- *Split*: Performs hidden surface removal, anti-aliasing and stores the image in a file.

This problem comes from a Departmental research group and is more complex than portrayed by our brief description. However, examining the structure of the program shows that *Model* consists of a loop that, for each frame in the animation, performs some work on the frame and calls *PolyConv* with the results. *PolyConv* manipulates the image received from *Model* and calls *Split*. *Split* does the final polishing of the frame and writes the final image to disk.

An *Enterprise* user manipulates icons that represent high-level program components called *assets* (defined in the next section of this paper). For this example, assume that an asset represents a single C procedure/function, called an *entry procedure*, together with a collection of support procedures used by the entry procedure, all contained in a single file. A program consists of several assets. In this example, there are three assets: *Model*, *PolyConv* and *Split*.

After starting *Enterprise* and choosing *New Program* from the main menu, a dialog box appears asking for the name of the program. After entering the name of the program, *Animation* in this case, a single asset appears that represents the entire program. Each asset is represented by a box, with the arrow indicating the flow of data. In *Enterprise* diagrams, the length of the critical data path is represented by the height of the diagram, while the degree of replicated (non-line) parallelism is represented by the width of the diagram.

The *Enterprise* window consists of an asset palette containing one icon for each asset kind and a canvas containing the program. A new program contains one *individual* asset that represents a sequential program component. The code for the procedures *Model*, *PolyConv* and *Split* could be associated with this single individual asset and run as a sequential program. However, there is no reason why *Model* should wait until *PolyConv* completes execution of the first animation frame to start processing the second frame. Similarly, *PolyConv* does not need to wait for *Split*. Therefore, the *individual* asset can be *coerced* to (replaced by) a *line* asset (pipeline in more traditional terminology) by selecting the individual asset and then selecting the *line* icon from the asset palette. After entering three as the length of the line, the *individual* is coerced to a three component line as shown in Figure 1.

The line shown in the figure is *collapsed*; that is, its components cannot be seen. By selecting the line and choosing *Expand Asset* from a pop-up menu, the *line* is expanded so that the three *individual* assets that it contains are visible.

To name an asset, the user selects it, chooses *Name Asset* from a pop-up menu and enters the name of the asset in the dialog box that appears. When the dialog box is closed, the name appears on the icon. To associate code with an asset, select it, choose *Edit Code* from a pop-up menu and enter the C code using your favorite text editor. Note that *Enterprise* automatically names the output file for the code you enter as the name of the asset with a ".c" suffix. In the example, the *Model* code would be saved in the file Model.c, illustrating the close relationship between *Enterprise* code and sequential C code.

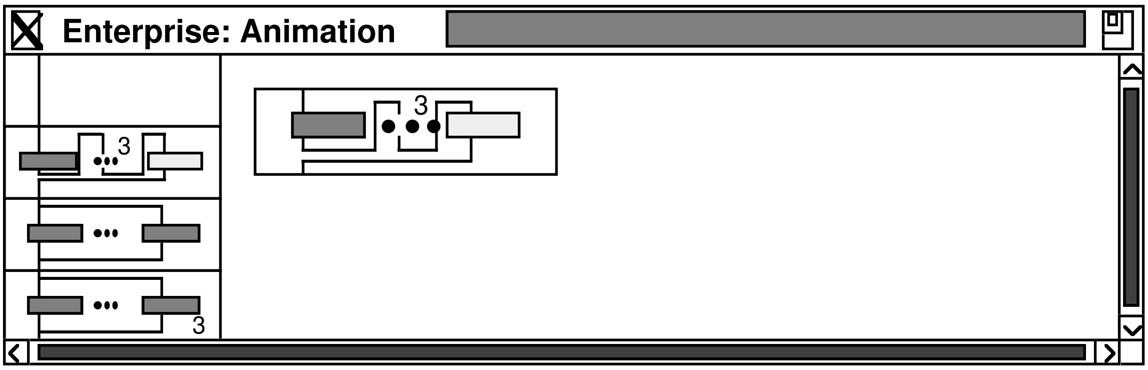

Figure 1. A Collapsed Line Asset Composed of Three Assets

To compile the application, the user selects *Compile* from a menu and the *Enterprise* system automatically inserts the code to handle the distributed computation, compiles the program and reports any errors back to the user. Once the program is compiled, the user selects the *Execute* menu item and *Enterprise* finds as many processors as are necessary to start the program, initiates processes on the processors, monitors the load on the machines and (if a contract is used) dynamically adds additional processors to the application as they become needed. For this animation example, a speed-up of 1.7 was obtained by using a line running on three processors instead of a single individual asset (a sequential program). Note that any timings are subject to large variations, depending on the number of available processors and the amount of traffic on the network.

The strength of the *Enterprise* model can be seen by the ease with which it is possible to take a program and experiment with alternate parallelization techniques *usually without changing the C source code*. For example, by selecting the *Split* asset and selecting a *contract* asset from the asset palette, the *Split* asset is coerced from an *individual* to a *contract* as shown in Figure 2.

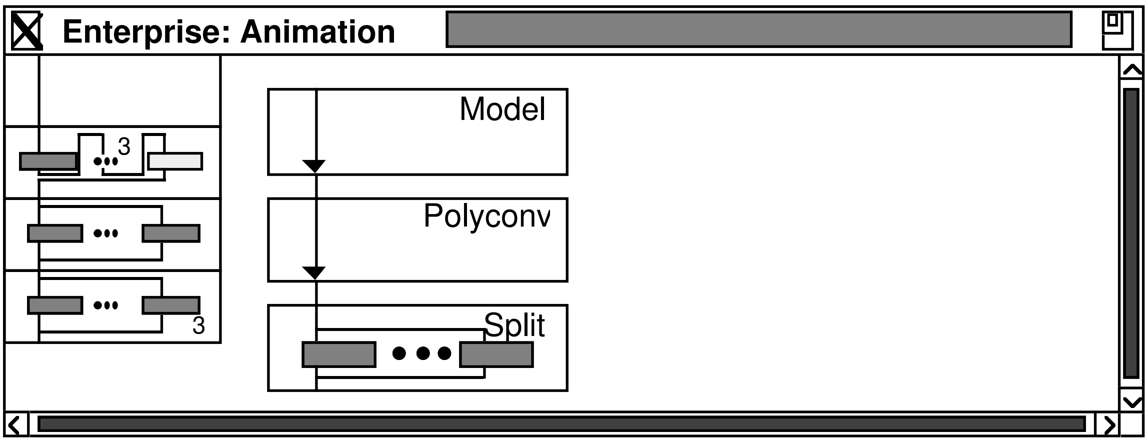

Figure 2. A Collapsed Contract Asset in a Line

In *Enterprise*, each asset represents at least one process. If a call is made to the *individual Split*, it is executed as a process and if a subsequent call is made to *Split* before the first call is complete, the second call must wait for the first call to finish. However, when a *contract* is executed, multiple processes can be used to execute multiple calls concurrently. That is, when *PolyConv* calls *Split*, a process is initiated and if a subsequent call is made to *Split* before the first call is done then a second process is initiated (if there is an available machine). *Enterprise* contracts are dynamic so as many processors as are available are used.

Coercing the *Split* asset to a contract results in as much as a 5.7-fold speed-up (using a dynamically varying number of processors) compared to the sequential animation program, depending on when the program is run. Of course, there is no reason why the user cannot coerce the *PolyConv* asset from an individual to a contract as well. However, in this case, a speedup of only 6.0 was obtained. This implies that the *Split* procedure is the real bottleneck in the animation program. That is, an individual *PolyConv* can almost keep up with its calls by *Model* but an individual *Split* cannot keep up with its calls by *PolyConv*.

If the C code for the individual *Split* contained a sequence of procedure calls (or even a single procedure call at the end of it), then *Split* could be expanded to reveal its individual asset components that could then be coerced to a line. Each of the assets in the line would represent one of the procedure calls. Several other asset kinds are supported by *Enterprise* and they can be combined in arbitrary hierarchies. The next section details the asset kinds that are available.

3. The Enterprise Model

The organization of a parallel or distributed program in *Enterprise* is similar to the organization of a sequential program. The structure of a program is independent of whether it is intended for sequential or distributed execution. The user views an *Enterprise* program as a collection of modules. Each *module* consists of a single *entry procedure* that can be called by other modules and a (possibly empty) collection of *internal procedures* that are known only within that module. No common variables among modules are allowed. In many ways, this is analogous to programming with abstract data types, which provide well-defined means for manipulating data structures while hiding all of the implementation details from the user.

Within any module, the code is executed sequentially. For example, a sequential program simply consists of a single module whose entry procedure is the main program. *Enterprise* introduces parallelism by allowing the user to specify the way in which the modules interact. Module interaction is specified by two factors: the *role* of a module and the *call* to a module. The role of a module defines which one of a fixed set of parallelization techniques (*asset* kinds) the module will use when it is invoked. The call to a module defines the identity of the called module, the information passed and the information returned. The role of a module is specified graphically while the call is specified in the code.

3.1 Module Calls

In a sequential program, procedures communicate using procedure calls. Procedure A contains a call to procedure B, that includes a list of arguments. When the call is made, procedure A is suspended and procedure B is activated. Procedure B can make use of the information passed as arguments. When procedure B has finished execution, it can communicate results to procedure A via side-effects to the arguments and/or by returning a value if the procedure is in fact a function.

Enterprise module calls are similar to sequential procedure calls. As with procedure and function calls, it is useful to differentiate between module calls that return a result and those that do not. Module calls that return a result are called *f-calls* (function calls) and module calls that do not return a result are called *p-calls* (procedure calls).

Enterprise module calls differ from sequential calls in the following ways:

- Arguments cannot be pointers, nor can they contain any pointers.

- When module A, calls another module, B, module A continues to execute. However, if the call to module B was an f-call, then module A would suspend itself when it tried to use the function result, if module B had not yet finished execution.

There is no syntactic difference between procedure calls and module calls. This makes it easier to transform sequential programs to parallel ones and makes it trivial to change parallelization techniques using the graphical user interface, usually without making changes to the user's code.

In *Enterprise*, an f-call is not necessarily blocking. Instead, the caller blocks only if the result is needed and the called module has not yet returned. Consider the following example:

```
result = B(data);
/* some other code */
value = result + 1;
```

When this code is executed, the calling module, say A, only blocks when the statement "*value = result + 1;*" is executed and only if module B has not yet returned the value of *result.* This concept is similar to the work on futures in object-oriented programming [6]. The p-call in the statement:

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

is non-blocking, so that A continues to execute concurrently with B. Of course in this case, B does not return a result to A.

3.2 Module Roles and Assets

The role of a module is based solely on a parallelization technique and is independent of its call. There are a fixed number of pre-defined roles corresponding to *asset* kinds. For example, in the previous section, the role of the *Split* module was changed from an *individual* to a *contract* without changing the call.

We have created an analogy between *Enterprise* programs and the structure of an organization to help describe module roles. In general, an organization has various assets available to perform its tasks. For example, a large task could be divided into sub-tasks where various sub-tasks are given to different parts of the organization (divisions, departments, pools, lines and/or individuals) to perform in parallel. Some tasks could even be completed by contract where the organization is not directly concerned about the nature or number of individuals that perform it. In addition, an organization usually provides many standard *services* (like time keeping, information storage and retrieval, etc.) that are available on demand to improve its functionality.

Currently, *Enterprise* supports the roles corresponding to seven different asset kinds: individual, line, pool, contract, department, division and service, with more being developed.

3.2.1 Individual

An *individual* contains no other assets. It is analogous to an individual person in an organization. When called, an individual executes its sequential code to completion. Therefore, any subsequent call to the same individual must wait until the previous call is finished. An individual may be called by any external asset using its name.

3.2.2 Line

A *line* contains a fixed number of heterogeneous assets in a fixed order. Each asset contains a call to the next asset in the line. A line is analogous to a construction, manufacturing or assembly line in an organization where at each point in the line, the work of the previous asset is refined. For example, a line might consist of an individual who takes an order, someone else to fill it and a third person to address the package and mail it. A subsequent call to the line waits only until the first asset is finished its sub-task of the previous call, not until the entire line is finished. The first asset in a line serves as the *receptionist* for the line and is the only asset that is externally visible. That is, the first asset of a line is the only asset that may be called from an external asset and it shares its name with the line asset for this purpose. Lines are often referred to as pipelines in the literature.

3.2.3 Pool

A *pool* contains a fixed number of identical assets. It is analogous to a pool in an organization where each member performs an identical task. For example, consider a pool of telephone operators. When a call is received, an idle operator services the call. However, if all the operators are busy, then the call waits for one of them to finish. Since pool members are externally indistinguishable, an external call cannot select a particular pool asset. Therefore, they are called by external assets using a single name that is shared with the pool asset for this purpose. Since all assets in a pool are identical, they also share the same code. A pool is analogous to a master-slave construct with a fixed number of slaves.

3.2.4 Contract

A *contract* contains a collection of identical assets, so it is similar to a pool. However, the number of assets in a contract is dynamic and depends on the number of processors that are free at any time. A contract is analogous to a contract that an organization lets for the performance of a collection of identical tasks. For example, an organization might sign a contract to a courier company for the delivery of its packages. When a package must be delivered, the courier company is informed. The organization doesn't care how many resources the courier company uses or the route it takes to deliver the packages. The delivery time can be affected by the number of resources used by the courier company and the amount of competing traffic. Similarly when an *Enterprise* call is made to a contract, an idle asset executes the call. However, if all assets are busy and no more are available for hire, then the call waits for an asset to become available. As is the case with a pool asset, a contract asset shares a common name with the identical assets it contains and these component assets also have common code. A contract is equivalent to a dynamic master-slave construct, where the number of slaves varies in response to program needs (demand) and resource utilization (environment).

3.2.5 Department

A *department* contains a fixed number of heterogeneous assets. Every department has a single receptionist asset that shares its name with the department so that it can be called by external assets. However, unlike a line, the other assets in a department do not call each other in a fixed sequential order. Instead, all other assets in the department are called directly by the receptionist. A department is analogous to a department in an organization where a receptionist is responsible for directing all incoming communications to the appropriate place. Note that in our analogy, a department consists of a collection of assets of any kind: individuals, departments, lines, etc. The department has no analogous term in the literature.

3.2.6 Division

A *division* contains a hierarchical collection of identical assets with a fixed breadth and depth where work is divided and distributed at each level. Every division has a single receptionist asset that shares its name with the division so that the division can be called by external assets. Divisions can be used to parallelize divide-and-conquer computations.

3.2.7 Service

A *service* contains no other assets. However, unlike an individual that can only answer a single call at any one time, a service may be used by more than one asset at the same time. A service is analogous to any asset in an organization that is not consumed by use and whose order of use is not significant. A service may be called by any asset using its name.

3.3 Enterprise Diagrams

An *Enterprise* diagram can be built from any combination of assets. For example, one can construct a contract, where each asset is itself a line of individuals. The model allows the user to coerce an asset from one kind to another *usually without any changes to the user's source code* (there are some problems users can encounter, particularly if pointers are used to alias data). However, some gathering or separation of functions from one file to another may be needed. For example, if a line is used for the animation example, there would be three individuals (*Model*, *PolyConv* and *Split*) each having their code in a separate file. If the line is coerced into an individual, the code needs to be gathered together (either in the same file or by using libraries). There are ways that *Enterprise* could do this management automatically, but there are some issues we have yet to resolve.

4. Related Work

In recent years, graphical parallel programming environments have been an area of active research. In these environment, a parallel program is specified as a graph with nodes containing a textual description of a sequential program. Most of the environments specify parallelism based on the large grain dataflow model developed by Babb [2] (for example: CODE [4], DGL [11], LGDF [8], Paralex [1], PPSE [12], TDFL [19]). In these models, an application is usually defined as a dataflow graph whose nodes contain sequential modules and the edges represents data dependencies between the modules.

TDFL introduces the idea of mutable computation graphs to support the mutable execution of recursive functions. If the data input to a function is above a threshold size, then additional processes are created to help process the data. The creation of nodes continues until the data size is within the threshold. However, data size may not be the best criteria to determine the degree of mutability, since processor usage is usually not necessarily proportional to data size. *Enterprise* also supports the mutable execution of recursive functions, but the criterion for creating additional processes is the depth of recursion instead of data size.

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

Another research area is in graphical environments to support dependency analysis of programs which are usually written in Fortran and run on machines with a small granularity (for example: PAT [16], E/SP [17], CAPER [18]). Yet another class of approaches provide a set of predefined parallel objects which define the communication and execution behavior of the enclosed module. Among the earlier efforts, PIE [13] made available to the programmer pre-coded templates of parallel structures such as master-slave, pipeline and systolic multidimensional pipeline. A template provided an implementation of the communication and synchronization structures and the programmer needed only to provide the sequential code. However, the parallelism supported is at a much smaller grain size than that offered by *Enterprise*. In contrast, *Enterprise* focuses on problem solving at a higher level, using the analogy to the structure of organizations. This allows new parallel templates to be defined whenever necessary. In our earlier system, *FrameWorks*, dynamic initiation of replicated process according to available idle processors was introduced [14][15]. The programmer also needed to specify parallelism, in a less intuitive way, through templates defining the input and output communication pattern of the process.

Linda, although not a graphical environment, is an important parallel programming model [5]. Linda uses the concept of a virtual shared memory, called the *tuple space*, for communication between concurrent processes. Processes use atomic operations to read, add, or delete a tuple to/from the tuple space. In contrast, *Enterprise* (and Paralex) is implemented using the object-based, message passing ISIS system [3]. Processes communicate by broadcasting messages to a message handler of a specified process.

Another relevant parallel programming model is the Chare-Kernel [9]. A *chare* is a module to be executed in parallel. It can have multiple entries in which enclosed program segments and new chares can be created (even recursively) to execute the entries. The portability of applications is a tremendous asset, however the onus is still on the programmer to use the supplied programming language enhancements and explicitly code all the parallelism.

5. Project Status

Enterprise is built on a number of available tools. The user interface was implemented using X Windows [10] (Motif), and is written in C++. It is currently 50% completed, but a textual interface is available now. Enterprise programs are executed using the communications and fault tolerance facilities of ISIS [3] and lines, pools, services, contracts and individuals have been implemented. A code librarian uses the UNIX *make* facility to automatically recompile source code when needed. The GNU C compiler has been modified to insert *Enterprise* code into the user's source program. The Monitor and debugging tools are in the design phase.

In recent years, there has been an enormous increase in the number and quality of parallel programming tools described in the literature. The authors of these tools have diverse opinions as to where and how in the software development cycle they can be used to increase a programmer's productivity. The *Enterprise* project aims for a complete, integrated programming environment that is suitable for the complete software development life cycle. By capturing an application's parallelism through the use of diagrams that are simple to edit, it is not difficult for the user to make the leap from sequential to parallel programming. Although the complexity of parallel systems, as portrayed in the literature, has been a powerful deterrent to growth in this area, we believe that with a simple model, all of the complexity of parallel programming can be hidden from the user. The analogical model used in *Enterprise* represents a different way of viewing an old problem.

Acknowledgement

This research was supported in part by research grants from the University of Alberta's Central Research Fund and the Natural Sciences and Engineering Research Council of Canada, grants OGP-8173 and infrastructure grant 107880. Also, Rasit Eskicioglu provided us with a number of useful references.

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

2 R. Babb. *Parallel Processing with Large Grain Data Flow Techniques*, IEEE Computer, 1984, Vol 17, No 7, pp 55-61.

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

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

5 N. Carriero and D. Gelernter. *Linda in Context*, CACM, 1988, Vol 32, No 4, pp 444-458.

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

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

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

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

10 J. Gettys, P. Karlton and S. McGregor. *The X Window System, Version II*, Software - Practice and Experience, 1990, Vol 20, No s2, pp s35-s67.

11 R. Jagannathan, A. Downing, W.T. Zaumen and R.K.S. Lee. *Dataflow-based Methodology for Coarse-Grain Multiprocessing on a Network of Workstations*, ICPP, 1989, pp 54-58.

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

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

14 A. Singh. *A Template-Based Approach to Structuring Distributed Algorithms Using a Network of Workstations*, 1991, Ph.D. Thesis, Dept. of Computing Science, University of Alberta.

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 on Parallel and Distributed Systems, 1991, Vol 2, No 1, pp 52-67.

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

17 K. Sridharan, M. McShea, C. Denton, B. Eventoff, J. Browne, P. Newton, M. Ellis, D. Grossbard, T. Wise and D. Clemmer. *An Environment for Parallel Structuring of Fortran Programs*, ICPP, 1989, pp 98-106.

18 B. Sugla, J. Edmark and B. Robinson. *An Introduction to the CAPER Application Programming Environment*, ICPP, 1989, pp 107-111.

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