Parallel Computing in Networks of Workstations with Paralex

Renzo Davoli **Luigi Alberto Giachini** **Özalp Babaoğlu**
Alessandro Amoroso **Lorenzo Alvisi**

Technical Report UBLCS-92-4

October 1992
Revised December 1994

Laboratory for Computer Science
University of Bologna
Piazza di Porta S. Donato, 5
40127 Bologna (Italy)

The University of Bologna Laboratory for Computer Science Research Technical Reports are available in gzipped PostScript format via anonymous FTP from the area `ftp.cs.unibo.it:/pub/TR/UBLCS` or via WWW with the URL `http://www.cs.unibo.it/`. Plain-text abstracts organized by year are available in the directory `ABSTRACTS`. All local authors can be reached via e-mail at the address *last-name*`@cs.unibo.it`. Written requests and comments should be addressed to `tr-admin@cs.unibo.it`.

UBLCS Technical Report Series

93-23 *A Scalable Architecture for Reliable Distributed Multimedia Applications*, F. Panzieri, M. Roccetti, October 1993.

93-24 *Wide-Area Distribution Issues in Hypertext Systems*, C. Maioli, S. Sola, F. Vitali, October 1993.

93-25 *On Relating Some Models for Concurrency*, P. Degano, R. Gorrieri, S. Vigna, October 1993.

93-26 *Axiomatising ST Bisimulation Equivalence*, N. Busi, R. van Glabbeek, R. Gorrieri, December 1993.

93-27 *A Theory of Processes with Durational Actions*, R. Gorrieri, M. Roccetti, E. Stancampiano, December 1993.

94-1 *Further Modifications to the Dexter Hypertext Reference Model: a Proposal*, W. Penzo, S. Sola, F. Vitali, January 1994.

94-2 *Symbol-Level Requirements for Agent-Level Programming*, M. Gaspari, E. Motta, February 1994.

94-3 *Extending Prolog with Data Driven Rules*, M. Gaspari, February 1994.

94-4 *Schedulability Checking of Data Flow Tasks in Hard-Real-Time Distributed Systems*, R. Davoli, L. A. Giachini, March 1994.

94-5 *A Shared Dataspace Language and its Compilation*, M. Gaspari, March 1994.

94-6 *Exploring the Coordination Space with LO*, S. Castellani, P. Ciancarini, April 1994.

94-7 *Comparative Semantics of LO*, S. Castellani, P. Ciancarini, April 1994.

94-8 *Distributed Conflicts in Communicating Systems*, N. Busi, R. Gorrieri, G. Siliprandi, April 1994.

94-9 *Pseudopolar Array Mask Algorithm for Spherical Coordinate Grid Surfaces*, A. Amoroso, G. Casciola, May 1994.

94-10 *MPA: a Stochastic Process Algebra*, M. Bernardo, L. Donatiello, R. Gorrieri, May 1994.

94-11 *Describing Queueing Systems with MPA*, M. Bernardo, L. Donatiello, R. Gorrieri, May 1994.

94-12 *Operational GSPN Semantics of MPA*, M. Bernardo, L. Donatiello, R. Gorrieri, May 1994.

94-13 *Experiments in Distributing and Coordinating Knowledge*, P. Ciancarini, May 1994.

94-14 *A Comparison of Parallel Search Algorithms Based on Tree Splitting*, P. Ciancarini, May 1994.

94-15 RELACS: *A Communications Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems*, Ö. Babaoğlu, M.G. Baker, R. Davoli, L.A. Giachini, June 1994.

94-16 *Replicated File Management in Large-Scale Distributed Systems*, Ö. Babaoğlu, A. Bartoli, G. Dini, June 1994.

94-17 *Parallel Symbolic Computing with the Shared Dataspace Coordination Model*, P. Ciancarini, M. Gaspari, July 1994.

94-18 *An Algorithmic Method to Build Good Training Sets for Neural-Network Classifiers*, F. Tamburini, R. Davoli, July 1994.

94-19 *On Group Communication in Large-Scale Distributed Systems*, Ö. Babaoğlu, A. Schiper, July 1994.

94-20 *Dynamic Allocation of Signature Files in Multiple-Disk Systems*, P. Ciaccia, August 1994.

94-21 *Parallel Independent Grid Files Based on a Dynamic Declustering Method Using Multiple Error Correcting Codes*, P. Ciaccia, November 1994.

95-1 *Performance Preorder and Competitive Equivalence*, F. Corradini, R. Gorrieri, M. Roccetti, January 1995.

95-2 *Clepsydra Methodology*, P. Ciaccia, O. Ciancarini, W. Penzo, January 1995.

95-3 *Debugging Distributed Executions Using Language Recognition*, Ö. Babaoğlu, E. Fromentin, M. Raynal, January 1995.

95-4 *Effective Applicative Structures*, A. Asperti, A. Ciabattoni, January 1995.

95-5 *An Open Framework for Cooperative Problem Solving*, M. Gaspari, E. Motta, A. Stutt, January 1995.

95-6 *Considering New Guidelines in Group Interface Design: a Group-Friendly Interface for the CHAOS System*, D. Bottura, C. Maioli, S. Mangiaracina, January 1995.

Parallel Computing in Networks of Workstations with Paralex[1]

Renzo Davoli[2] **Luigi Alberto Giachini**[2] **Özalp Babaoğlu**[2]
Alessandro Amoroso[2] **Lorenzo Alvisi**[3]

Technical Report UBLCS-92-4

October 1992
Revised December 1994

Abstract

Modern distributed systems consisting of powerful workstations and high-speed interconnection networks are an economical alternative to special-purpose supercomputers. The technical issues that need to be addressed in exploiting the parallelism inherent in a distributed system include heterogeneity, high-latency communication, fault tolerance and dynamic load balancing. Current software systems for parallel programming provide little or no automatic support towards these issues and require users to be experts in fault-tolerant distributed computing. The Paralex system is aimed at exploring the extent to which the parallel application programmer can be liberated from the complexities of distributed systems. Paralex is a complete programming environment and makes extensive use of graphics to define, edit, execute and debug parallel scientific applications. All of the necessary code for distributing the computation across a network and replicating it to achieve fault tolerance and dynamic load balancing is automatically generated by the system. In this paper we give an overview of Paralex and present our experiences with a prototype implementation.

Index Terms: *Distributed computing, parallelism, speed-up, supercomputing, fault tolerance, load balancing, graphical programming, heterogeneity, large-grain data flow.*

---

1. An earlier version of this work appeared under the title "Paralex: An Environment for Parallel Programming in Distributed Systems" in *Proceedings of the 6th ACM International Conference on Supercomputing*, Washington, D.C., July 1992, pp. 178–187. Partial support for this work was provided by the Commission of European Communities under ESPRIT Programme Basic Research Project 6360 (BROADCAST), the United States Office of Naval Research under contract N00014-91-J-1219, IBM Corporation, the Italian National Research Council (CNR–Contributo 91.02368.12) and the Italian Ministry of University, Research and Technology.
2. Department of Mathematics, University of Bologna, Piazza Porta S. Donato 5, 40127 Bologna, Italy.
3. Department of Computer Science, 4130 Upson Hall, Cornell University, Ithaca, New York 14853 USA.

1 Introduction

There is general agreement that significant future increases in computing power will be possible only through exploiting parallelism. Distributed systems comprising powerful workstations and high-speed communication networks represent valuable parallel computational resources. In fact, the amount of raw computing power that is present in a typical modern distributed system with dozens, if not hundreds, of general-purpose workstations may be comparable to an expensive, special-purpose supercomputer. Thus, it is tempting to try to harness the massive parallelism available in these systems for single, compute-intensive applications. There are, however, several obstacles that remain before networks of workstations can become a poor man's supercomputer.

Distributed systems differ from special-purpose parallel computers in that they (i) exhibit asynchrony with respect to computation and communication, (ii) communicate over relatively low-bandwidth, high-latency networks, (iv) lack architectural and linguistic homogeneity, (v) exhibit increased probability of communication and processor failures, and (vi) fall under multiple administrative domains.

As a consequence, developing parallel programs in such systems requires expertise not only in distributed computing, but also in fault tolerance. A large number of important applications (e.g., genome analysis) require days or weeks of computations on a network with dozens of workstations [27]. In these applications, many hours of computation can be wasted not only if there are genuine hardware failures, but also if one of the processors is turned off, rebooted or disconnected from the network. Given that the most common components of a distributed system are workstations and that they are typically under the control of multiple administrative domains (typically individuals who "own" them), these events are much more plausible and frequent than real hardware failures.

We claim that current software technology for parallel programming in distributed systems is comparable to assembly language programming for traditional sequential systems — the user must resort to low-level primitives to accomplish data encoding/decoding, communication, remote execution, synchronization, failure detection and recovery. It is our belief that reasonable technologies already exist to address each of these problems individually. What remains a challenge is the task of integrating these technologies in software support environments that permit easy development of reliable applications to exploit the parallelism and fault tolerance offered by distributed systems.

The Paralex project has been undertaken to explore the extent to which the parallel application programmer can be isolated from the complexities of distributed systems. Our goal is to realize an environment that will encompass all phases of the programming activity and provide automatic support for distribution, fault tolerance and heterogeneity in distributed and parallel applications. Paralex makes extensive use of graphics for expressing computations, controlling execution and debugging. The programming paradigm supported by Paralex promotes the view of parallel computations as collages of ordinary sequential programs. The "glue" necessary for combining computations consists of interdependencies and data flow relations that are expressed using a graphical notation. In the limit, interesting new parallel programs can be constructed by reusing existing sequential software and without having to rewrite a single line of traditional code. As such, Paralex also addresses the issue of software reusability [18].

The rest of the paper is organized as follows. The next section defines the programming model supported by Paralex. In Section 3 we give an overview of the principal components of Paralex and illustrate the user interface through examples. Section 4 is a description of how Paralex uses passive replication to automatically render programs fault tolerant. A novel use of replication to also implement dynamic load balancing is discussed in Section 5. Section 6 compares debugging Paralex programs to distributed debugging in general and describes the mechanisms to support it. Performance results for two realistic applications obtained with the current prototype are reported in Section 7. Paralex is put in perspective with respect to related work in Section 8. Section 9 discusses some of our design decisions and directions for future work. Section 10 concludes the paper.

2 The Paralex Programming Paradigm

The choices made for programming paradigm and notation are fundamental in harnessing parallelism in a particular application domain [19]. The programming paradigm supported by Paralex can be classified as static data flow [1]. A Paralex program is composed of *nodes* and *links*. Nodes correspond to computations (functions, procedures, programs) and the links indicate flow of (typed) data. Thus, Paralex programs can be thought of as directed graphs (and indeed are visualized as such on the screen) representing the data flow relations plus a collection of ordinary sequential code fragments to indicate the computations.

Unlike classical data flow, nodes of a Paralex program carry out significant computations. This so-called *large-grain* data flow model [6] is motivated by the high-latency, low-bandwidth network that is typically available for communication in distributed systems. Only by limiting the communication-to-computation ratio to reasonable levels can we expect acceptable performance from parallel applications in such systems. It is up to the user to adopt an appropriate definition for "large grain" in decomposing an application to its parallel components for a particular system.

While extremely simple, the above programming paradigm has several desirable properties. First, application parallelism is explicit in its notation — all nodes that have no data interdependencies can execute in parallel. Second, the small number of abstractions that the programmer has to deal with are familiar from sequential programming. In particular, there are no new linguistic constructs for communication or synchronization. Finally, composing parallel programs by interconnecting sequential computations allows automatic support for heterogeneity and fault tolerance and facilitates software reuse as discussed in subsequent sections.

2.1 Computation Nodes

The basic computational unit of a Paralex program is a *multi-function* mapping some number of inputs to outputs. The graphical representation for the multi-function itself is a *node* and that of the inputs and outputs are incoming and outgoing *links*, respectively. The semantics associated with this graphical syntax obeys the so-called *strict enabling rule* of data-driven computations in the sense that when all of the incoming links contain values, the computation associated with the node starts execution and transforms the input data to the output data. Paralex functions must be pure in that they cannot have side effects. In particular, persistent internal state or interactions with external components such as files, devices and other computations are not permitted. Motivations and implications of this restriction are discussed in Section 9.

The actual specification of the computation may be done using whatever appropriate notation is available, including sequential programming languages such as C, C++, Fortran, Pascal, Modula or Lisp[4]. It is also possible for computations to be carried out through compiled binaries or library functions subject to architectural compatibility as discussed in Section 3.3.

How multiple outputs are specified for multi-functions depends on the language being used to program the nodes. One possibility is to implement the functions as procedures and return results through call-by-reference parameters. Another possibility is to use simple functions and pack multiple results into composite data types such as structures, records or arrays. We pursue this option in the next section.

2.2 Filter Nodes

Filters permit multi-functions to be implemented using simple functions. They allow the single (structured) result to be picked apart to produce multiple outputs. In this manner, subsets of the data produced by the multi-function may be sent to different destinations in the computation graph. This is a principal paradigm for data-parallel computing. For example, a single large matrix produced by some node in the computation may be "filtered" by extracting each of the quadrants to produce four sub-matrices to be processed in parallel at the next level.

Conceptually, filters are defined and manipulated just as regular nodes and their "computations" are specified through sequential programs. In practice, however, all of the data filtering

4. The exact list of languages permissible for node computations is determined by type and calling convention compatibility with C

computations are executed in the context of the single process that produced the data rather than as separate processes. Associating filters with the producer of the data not only saves network bandwidth, it also economizes on data buffers necessary at the consumers.

Note that if node computations included explicit constructs for communication, outputs could be sent directly to their destinations as they were being produced and filters would be unnecessary. This, however, would be contrary to our goal of "plugging in" existing sequential code into the nodes without having to extend them with new language constructs. In the case of compiled binaries or library functions where source code is not available, such extensions may not even be physically possible.

2.3 Subgraph Nodes

Paralex computation graphs may be structured hierarchically. Any node may contain a graph structure rather than sequential code to carry out its computation. These *subgraph* nodes obey the same semantics as primitive multi-function nodes and may be further decomposed themselves. Subgraphs are to Paralex what procedures are to sequential programs — a structuring abstraction that renders programs not only easier to understand, but also easier to construct using pre-programmed components.

2.4 Cycle Nodes

At the inter-node level, Paralex computation graphs are acyclic. Any single node of the graph, however, can be invoked repetitively during execution as long as its outputs match (in number and type) its inputs. The termination condition for the cycle can be dynamic as it is defined explicitly by the user as a function of the node inputs.

The Paralex cycle construct has a "while-loop" semantics and operates as follows. If the termination function evaluates to false, the node computation is skipped and the input values appear as output. Otherwise, the outputs produced by the node computation are "wrapped around" and become inputs for the next iteration. While the cycle is active, external input to the node is blocked.

3 Overview of Paralex

Paralex consists of four logical components: a graphics editor for program development, a compiler, an executor and a runtime support environment. The first three components are integrated within a single graphical programming environment. It is, however, possible to edit, compile or execute Paralex programs also from machines with no graphics support. In this section we illustrate some of the principal characteristics of the user interface through examples.

3.1 Site Definition

The distributed system on which Paralex is to run is defined through a file called `paralex.site`. This file can be viewed as an "architecture description database." Each host of the system that is available for Paralex has a single-line descriptor. The first field of the line is the name of the host. Following the name is a comma-separated list of attributes. An example of a site file is shown in Figure 1.

Attributes may be binary (e.g., `sparc, fpu`) or numeric (e.g., `gfx, spec`). A binary attribute is associated with a host simply by naming it. Numeric attributes are given integer values through assignment and are set to one by default. Naming another host as an attribute is a shorthand for defining similar machines (e.g., `jago` is exactly the same as `fyodor`). Paralex neither defines nor interprets keywords that define the set of attributes. They are used by the loader in selecting sets of hosts suitable for executing nodes through a pattern matching scheme. This mechanism allows new attributes to be introduced and old ones to be modified at will by the user.

A minimum description of a host must contain its name and the processor architecture family (e.g., `sparc, mips, vax`). In case two hosts share the same processor architecture family but

```
elettra   sparc, SunOS, rel=4, rev=1, fpu, gfx=2, spec=21
xenia     sparc, SunOS, rel=4, rev=1, fpu, spec=13
fyodor    sparc, SunOS, rel=4, rev=1, fpu, gfx, spec=10
nabucco   mips, fpu, gfx, color, spec=18
tosca     m68020-SunOS, SunOS, rel=4, rev=0, fpu, spec=9
violetta  m68020-A/UX, spec=5
turandot  RS6000, AIX, fpu, spec=12
carmen    vax, gfx, memory=16, spec=6
jago      fyodor
```

Figure 1. Paralex Site Definition File.

are not binary compatible, additional information such as the operating system type must be included in the characterization to distinguish them. In the example of Figure 1, `tosca` and `violetta` are both based on `m68020` but are not binary compatible. If present, the attribute `spec` is an indication of the raw processor power measured in SPECmarks [25]. The SPECmark value of a host is used by the mapping and dynamic load balancing algorithms to associate computations with hosts. If the SPECmark rating of a particular host is not available, Paralex assumes that it is lower than all the other hosts.

Note that the site definition contains no explicit information about the communication characteristics of the distributed system. The current version of Paralex assumes that each pair-wise communication between hosts is possible and uniform. This assumption is supported by broadcast LAN-based systems that are of immediate interest to us. With the advent of high-speed wide-area networking technologies, the site definition file could easily be extended to include explicit communication topology information and permit parallel computing over non-uniform and long-haul communication networks.

3.2 The Graphics Editor

The Paralex graphics editor allows computation graphs to be constructed using a pointing device such as a mouse and pull-down menus. An actual screen image of a Paralex session is displayed in Figure 2. The application being programmed is a parallel solution to the Synthetic Aperture Radar (SAR) problem where radar echo data collected from an aircraft or spacecraft flying at a constant altitude are used to reconstruct contours of the terrain below despite cloud cover, speed fluctuations, etc. The steps necessary for producing high-quality images from SAR data consist of the following sequence of computations: two-dimensional discrete Fourier transform, binary convolution, two-dimensional inverse discrete Fourier transform and intensity level normalization for visualization. Each of these steps can be parallelized by initially partitioning the input data and then reconstructing it at intermediate stages through appropriate shuffling exchanges. Figure 2 illustrates the data flow patterns that result from a 4-way parallelization of the problem.

As can be seen from the Figure, the application consists of a top-level graph called `SAR` and a number of subgraphs corresponding to the two-dimensional discrete Fourier transform, convolution, two-dimensional inverse discrete Fourier transform and intensity level normalization computations. Nodes that are subgraphs or cycles can be *opened* to reveal their structure in separate windows. Primitive nodes, subgraphs and cycles are distinguished by their graphical representation. Function outputs are designated as small bubbles attached to a node. They act as place holders for naming the output and associating a filter with it. Links originating from the same bubble carry the same data. In their expanded form, subgraphs and cycles have their inputs and outputs named through *anchors* drawn as gray (virtual) nodes. Finally, multiple links between a pair of nodes can be *bundled* together to eliminate visual clutter. This is the case for all of the links in the top-level graph `SAR`.

The graphics editor implements a small number of primitives for creating, deleting and repositioning of nodes and links. The graphics editor also permits cut-and-paste operations and

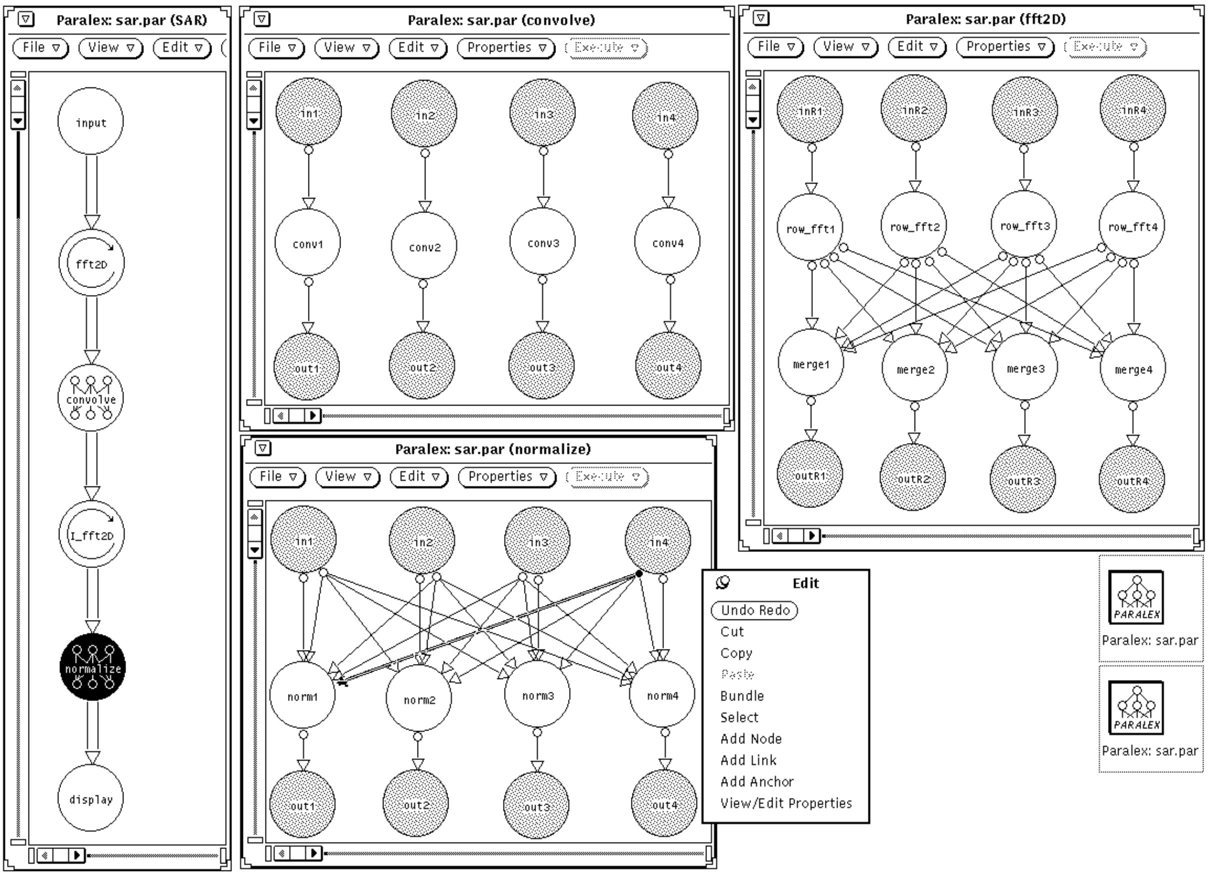

Figure 2. Paralex Graphics Editor.

undoing the last edit function performed. The current computation can be saved for future use or an existing graph can be loaded for inclusion in the program. Paralex saves program graphs as plain text files with a C-like syntax for describing the attributes and structure. This permits Paralex programs to be manipulated, compiled and executed from machines that do not have graphics support.

3.3 Specifying Program Properties

The next stage in program development is to specify the properties of the various program components. This is accomplished by filling in property charts called *panels*. At the application level, the only property to specify is the fault tolerance degree. As described in Section 4, Paralex will use this information to automatically replicate the application. At the node level, the properties to be specified are more numerous.

Figure 3 displays the node panel associated with the highlighted node of subgraph `convolve` where the following fields can be specified:

Name Used to identify the node on the screen.

File Name of the file containing the computation associated with the node. It may be a source file, compiled binaries or a library.

Include Name of a file to be included before compilation. Used to include user-defined constants

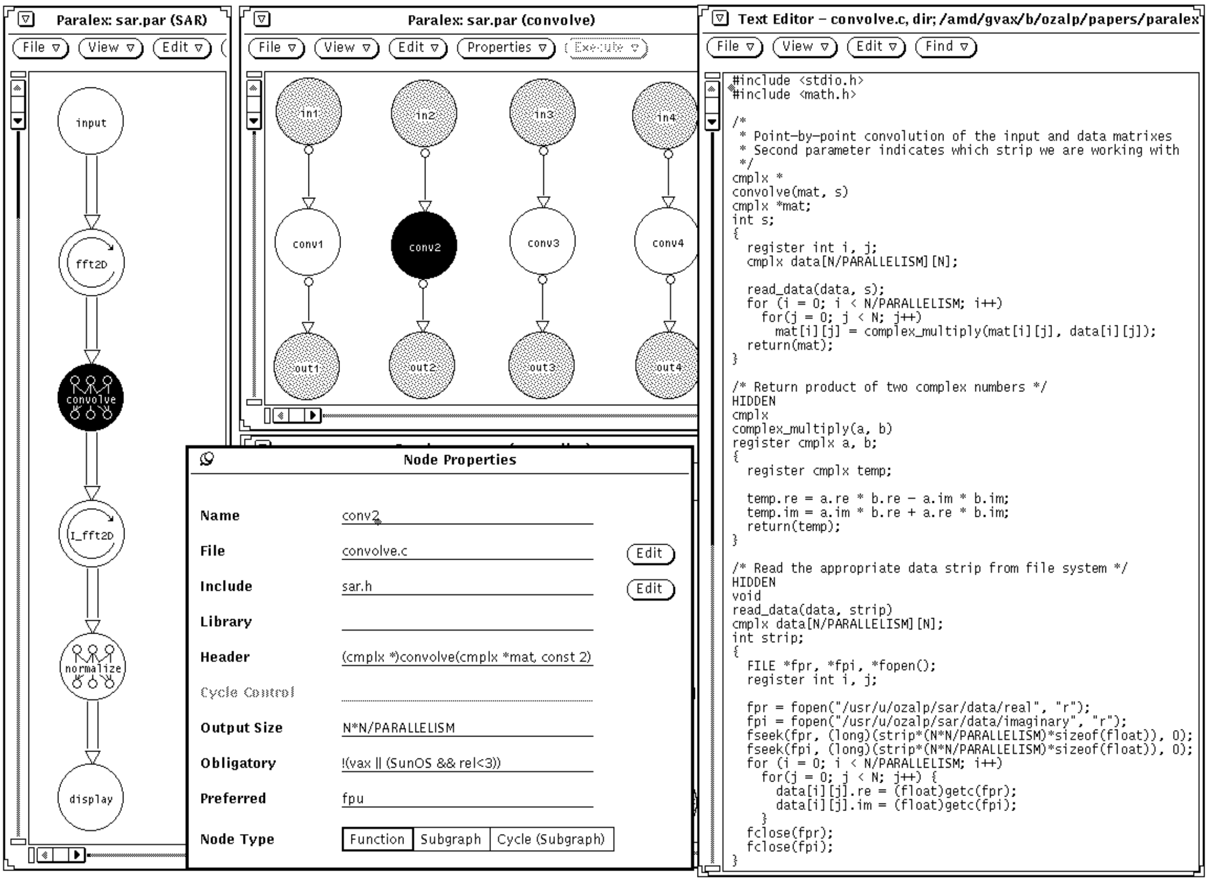

Figure 3. Specifying Node Properties and Computations.

and data types in a Paralex program.

Header The interface specification for the function computed by the node. Must declare all input parameters and results along with their types. The syntax used is ANSI C with an extension to permit multiple return values for functions. Note that ANSI C is used only as an interface specification language and has no implications on the language in which the function is programmed.

Cycle Control Declaration of a Boolean function to be used for controlling cycle termination. The decision to continue the cycle can be a function of the same inputs that the computation is based on.

Output Size In case the dimension of an output structured type is not evident from the header declaration, it must be specified here. For instance, this is the case for C arrays declared as pointers.

Obligatory A Boolean query naming host attributes that must be satisfied in order for the node to execute on a host. Used primarily for including compiled binaries or library functions of a specific architecture as node computations.

Preferred A Boolean query naming host attributes that further constrains the above list of hosts for performance reasons. The Paralex loader uses this information as a hint when making mapping decisions.

Node Type Identifies the node as one of `Function`, `Subgraph` or `Cycle`.

Obligatory and preferred host specifications are done using the site definition file described in Section 3.1. A query is formulated using C Boolean expression syntax. As an example, the obligatory query

```
!(vax || (SunOS && rel < 5))
```

of Figure 3 prevents the node from executing on any host that either has the Vax architecture or is running a release of SunOS earlier than 5.0. Recall that the semantics associated with host attributes in the site definition file are user defined; Paralex uses them at load time in order to decide which hosts each node of the computation can be mapped to. The obligatory host query is used to indicate the weakest requirements for architectural and functionality reasons while the preferred host query typically adds performance-related considerations. In case a particular query produces an empty set of candidate hosts against a particular site definition file, the graphics editor signals a warning. A warning is also generated if the desired fault tolerance for the application cannot be satisfied with the given site definition. We consider these events warnings rather than errors since the binding of computations to hosts is not performed until load time. Given that Paralex programs may be transported and executed on systems other than those where they were developed, the warnings at edit time may be irrelevant at load time.

Clicking on the `Edit` button in the node panel invokes a text editor on the source file or the include file of a node. Figure 3 shows such an editor invoked on the file `convolve.c` containing the source code for the highlighted node "conv2". In this example, the function is programmed in C and makes use of two internal functions to perform the point-by-point convolution between its input and a data matrix read from the file system. Note that the code is ordinary sequential C and contains nothing having to do with remote communication, synchronization or fault tolerance.

The attributes of a filter are specified and edited exactly in the same manner as an ordinary node through a panel associated with the output bubble. The only difference is that most of the fields in the filter panel are inherited from the parent node and are not modifiable. Finally, a link panel is used to name the input parameter of the destination function that is to receive the data value.

3.4 Compiling Paralex Programs

Once the user has fully specified the Paralex program by drawing the data flow graph and supplying the computations to be carried out by the nodes, the program can be compiled. The textual representation of the Paralex program is fed as input to the compiler. Although the compiler may be invoked manually as a command, it is typically invoked from the graphical interface where the program was composed.

The first pass of the Paralex compiler is actually a precompiler to generate all of the necessary stubs to wrap around the node computations to achieve data representation independence, remote communication, replica management and dynamic load balancing. Type checking across links is also performed in this phase. Currently, Paralex generates all of the stub code as ordinary C. As the next step, a collection of standard compilers are invoked: C compiler for the stubs, perhaps others for the node computations, if they are not already compiled. Note that Paralex in no way modifies the source code computations even if they are available. For each node, the binaries for stub and the actual computation are combined only at link time while producing the executable module.

The compiler must also address the two aspects of heterogeneity — data representation and instruction sets. Paralex uses the ISIS toolkit [14, 13] as the infrastructure to realize a universal data representation. All data that is passed from one node to another during the computation are encapsulated as ISIS messages. Paralex automatically generates all necessary code for encoding-decoding basic data types (integer, real, character) and linearizing arrays of these basic types. The user must supply routines to linearize all other data types.

Heterogeneity with respect to instruction sets is handled by invoking remote compilations on the machines of interest and storing multiple executables for the nodes. Versions of the executable code corresponding to the various architectures are stored in subdirectories (named with the architecture class) of the current program. A network file server that is accessible by all of the hosts acts as the repository for the executables.

3.5 Executing Paralex Programs

The Paralex executor consists of a loader, controller and debugger. The debugger is incorporated into the graphical interface and uses the same display as the editor. The loader takes the output of the compiler and the textual representation of the computation graph as input and launches the program execution in the distributed system. As with the compiler, the loader can be invoked either manually as a command or through the graphical interface.

During the loading phase, heuristics are used to solve the *mapping problem* where each node of the Paralex program is associated with a host of the distributed system [5]. The stub code generated by the compiler includes monitoring and control primitives such that the initial mapping of nodes to hosts can be altered during execution in order to achieve dynamic load balancing as explained in Section 5.

4 Fault Tolerance

One of the primary characteristics that distinguishes a distributed system from a special-purpose supercomputer is the possibility of partial failures during computations. As noted earlier, these may be due to real hardware failures or, more probably, as consequences of administrative interventions. To render distributed systems suitable for long-running parallel computations, automatic support for fault tolerance must be provided. The Paralex run-time system contains the primitives necessary to support fault tolerance and dynamic load balancing.

As part of the program definition, Paralex permits the user to specify a fault tolerance level for the computation graph. Paralex will generate all of the necessary code such that when a graph with fault tolerance k is executed, each of its nodes will be replicated at $k + 1$ distinct hosts to guarantee success for the computation despite up to k failures. Failures that are tolerated are of the benign type for processors (i.e., all processes running on the processor simply halt) and communication components (i.e., messages may be lost). There is no attempt to guard against more malicious processor failures nor against failures of non-replicated components such as the network interconnect.

Paralex uses passive replication as the basic fault tolerance technique. Given the application domain (parallel scientific computing) and hardware platform (network of workstations), Paralex favors efficient use of computational resources over fast recovery times in the presence of failures. Passive replication not only satisfies this objective, it provides a uniform mechanism for dynamic load balancing through late binding of computations to hosts as discussed in Section 5.

Paralex uses the ISIS *coordinator-cohort* toolkit to implement passive replication. Each node of the computation that requires fault tolerance is instantiated as a process group consisting of replicas for the node. One of the group members is called the *coordinator* in that it will actively compute. The other members are *cohorts* and remain inactive other than receiving multicasts addressed to the group. When ISIS detects the failure of the coordinator, it automatically promotes one of the cohorts to the role of coordinator.

Data flow from one node of a Paralex program to another results in a multicast from the coordinator of the source group to the destination process group. Only the coordinator of the destination node will compute with the data value while the cohorts simply buffer it in an input queue associated with the link. When the coordinator completes computing, it multicasts the results to the process groups at the next level and signals the cohorts (through another intra-group multicast) so that they can discard the buffered data item corresponding to the input for the current invocation. Given that Paralex nodes implement pure functions and thus have no internal state, recovery from a failure is trivial — the cohort that is nominated the new coordinator simply starts computing with the data at the head of its input queues.

Figure 4 illustrates some of these issues by considering a 3-node computation graph shown at the top as an example. The lower part of the figure shows the process group representation

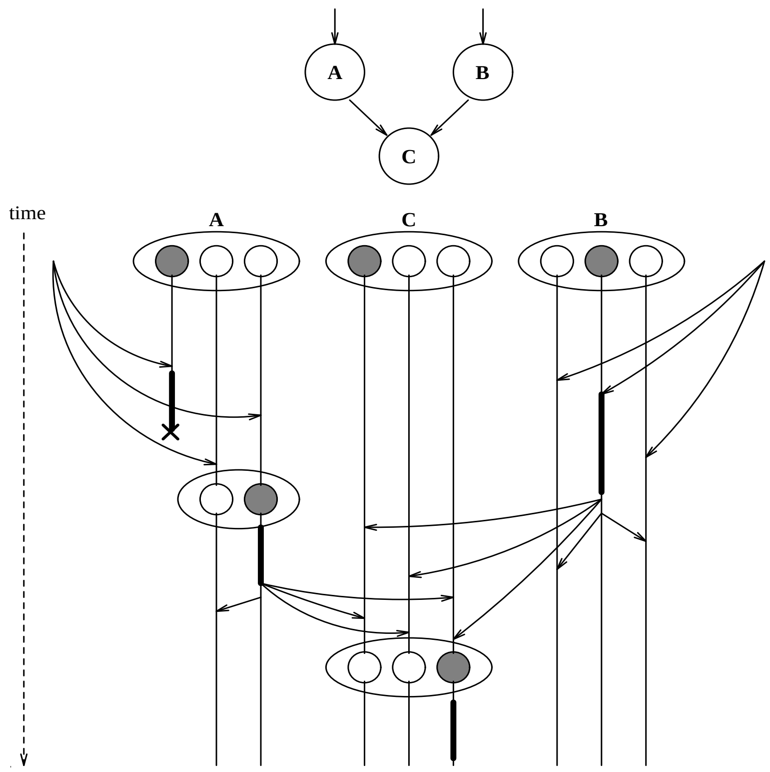

Figure 4. Replication and Group Communication for Fault Tolerance.

of the nodes based on a fault tolerance specification of 2. Arrows indicate message arrivals with time running down vertically. The gray process in each group denotes the current coordinator. Note that in the case of node A, the initial coordinator fails during its computation (indicated by the X). The failure of the coordinator is detected by ISIS and the process group is reformed with the right-most replica taking over as coordinator. At the end of its execution, the coordinator performs two multicasts. The first serves to communicate the results of the computation to the process group implementing node C and the second is an internal group multicast. The cohorts use the message of this internal multicast to conclude that the current buffered input will not be needed since the coordinator successfully computed with it. Note that there is a small chance the coordinator will fail after multicasting the results to the next node but before having informed the cohorts. The result of this scenario would be multiple executions of a node with the same (logical) input. This is easily prevented by tagging each message with an iteration number and ignoring any input messages with duplicate iteration numbers.

The execution depicted in Figure 4 may appear deceptively simple and orderly. In a distributed system, other executions with inopportune node failures, message losses and event orderings may be equally possible. What simplifies the Paralex run-time system immensely is structuring it on top of ISIS that guarantees "virtual synchrony" with respect to message delivery and other asynchronous events such as failures and group membership changes. Paralex cooperates with ISIS toward this goal by using a reliable multicast communication primitive that respects causality [34].

5 Dynamic Load Balancing

Before a Paralex program can be executed, each of the nodes (and their replicas, in case fault tolerance is required) must be associated with a host of the distributed system. Intuitively, the goals of the *mapping problem* are to improve performance by maximizing parallel execution and minimizing remote communication, to distribute the load evenly across the network, and to satisfy the fault tolerance and heterogeneity requirements. Since an optimal solution to this problem is computationally intractable, Paralex bases its mapping decisions on simple heuristics described in [5]. The units of our mapping decision are *chains* defined as sequences of nodes that have to be executed sequentially due to data dependence constraints. The initial mapping decisions, as well as modifications during execution, try to keep all nodes of a chain mapped to the same host. Since, by definition, nodes along a chain have to execute sequentially, this choice minimizes remote communication without sacrificing parallelism.

To achieve failure independence, each member of a process group representing a replicated node must be mapped to a different host of the distributed system. Thus, the computation associated with the node can be carried out by any host where there is a replica. To the extent that nodes are replicated for fault tolerance reasons, this mechanism also allows us to dynamically shift the load imposed by Paralex computations from one host to another.

As part of stub generation, Paralex produces the appropriate ISIS calls so as to establish a coordinator for each process group just before the computation proper commences. The default choice for the coordinator will be as determined by the mapping algorithms at load time. This choice, however, can be modified later on by the Paralex run-time system based on changes observed in the load distribution on the network. By delaying the establishment of a coordinator to just before computation, we effectively achieve dynamic binding of nodes to hosts, to the extent permitted by having replicas around.

The run-time system in support of dynamic load balancing consists of a collection of daemon processes (one per host of the distributed system) and a *controller* process running on the host from which the program was launched. The daemon processes are created as part of system initialization and periodically monitor the local load as measured by the length of the runnable process queue of the local operating system. Note that this measure includes all load present at the host, including those not due to Paralex. Raw load measurements taken over a sliding window are averaged with exponentially decaying weights so as to extract long-term trends. If two consecutive load measures differ by more than some threshold, the daemon process multicasts the new value to an ISIS process group called `Paralex-Monitor`. Each instance of the Paralex controller (corresponding to different Paralex programs being executed on the same distributed system) that wishes to collect dynamic load information joins this process group and listens for load messages. After some time, the controller will have built a table of load values for each host in the system. It is also possible for the controller to obtain a load value by explicitly asking one of the daemons.

In addition to collecting load information, the controller also tracks the state of the Paralex computation. The code wrapped around each node includes ISIS primitives to send the controller an informative message just before it starts computing. The controller uses this information for both dynamic load balancing and debugging as discussed in Section 6.

Coordinator modification decisions for dynamic load balancing purposes are exercised at chain boundaries. When the controller becomes aware of the imminent execution of a node, it examines the program graph to determine if any of the node's successors begins a new chain. For each such node, the controller consults the most recent load levels to decide if the current coordinator choice is still valid. If not, the controller multicasts to the process group representing the first node of a chain the identity of the new coordinator. Since the computation and the load balancing decisions proceed asynchronously, the controller tries to anticipate near future executions by looking one node ahead. If there is a modification of the coordinator for a chain, this information is passed along to all other nodes of the chain by piggy-backing it on the data messages. In this manner, the controller needs to inform only the head of a chain.

The actual decision to modify a coordinator choice is made by the controller based on the *residual power* metric of a host defined as $\mathit{spec}/(\mathit{load}+1)$. The preferred host for executing the chain is taken as the one with the largest residual power among all hosts that contain a replica. Note that the progress of the program is not dependent on the continued functioning of the controller. In case the controller fails, the program proceeds with the default mapping that was

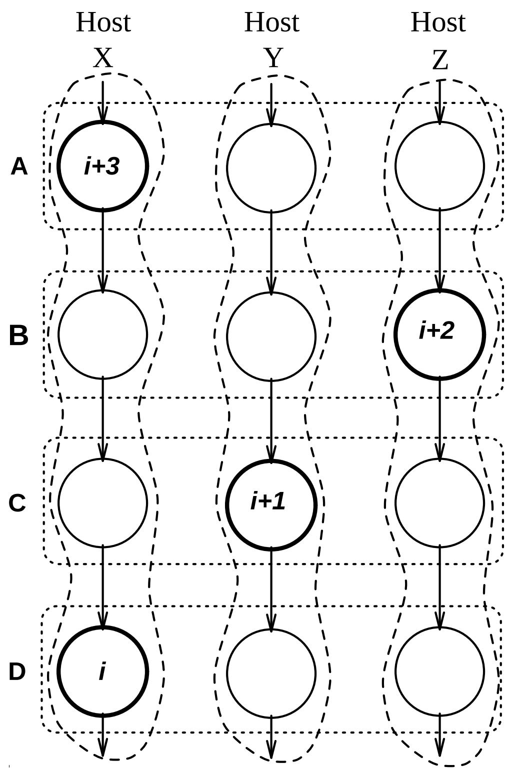

Figure 5. Pipelined Execution of a Replicated Chain.

established at load time, perhaps with degraded performance. Thus, the controller is not a fault tolerance bottleneck and serves only to improve performance by revising the mapping decisions based on dynamic load. Since all communication between the program and the controller is asynchronous, it is also not a performance bottleneck.

Perhaps the most dramatic effects of our dynamic load balancing scheme are observed when the computation graph is executed not just once, but repetitively on different input data. This so-called "pipelined operation" offers further performance gains by overlapping the execution of different iterations. Whereas before, the nodes of a chain executed strictly sequentially, now they may all be active simultaneously working on different instances of the input.

At first sight, it may appear that our mapping strategy of keeping all nodes of a chain on the same host is a poor choice for pipelined execution. Without the dynamic load balancing mechanisms, it will indeed be the case where all nodes of a chain may be active on the same host with no possibility of true overlapped execution. In case of replicated chains, the dynamic load balancing mechanisms outlined above will effectively spread the computational work among various hosts and achieve improved overlap.

The example depicted in Figure 5 consists of a chain with four nodes, A, B, C and D, replicated on three hosts X, Y and Z. For simplicity sake, assume that the hosts are uniform in raw computing power and that there is no external load on the network. When the computation graph is activated with input corresponding to iteration i, assume host X is selected as the coordinator among the three hosts which all have identical residual power metrics. Now, while node A is active computing with input i, the controller will favor hosts Y and Z over X since they have larger residual powers. When node A terminates execution and the graph is invoked again for iteration $i + 1$, the new instance of node A will be started on one of Y or Z as the coordinator. For argument sake, assume host Y is selected. Proceeding in this manner, each new iteration of the graph will result in a new host being activated as long as there are replicas of the chain.

Obviously, eventually hosts will have to be reused as shown in the Figure where host X has both nodes A and D active working on iterations $i+3$ and i, respectively. Note that at any time, only one of the replicas per process group is active. This has to be maintained to guarantee the desired fault tolerance. The net effect of the load balancing mechanism is to spread the computation corresponding to four iterations of the graph over the three hosts that contain replicas. We obtain this without having had to introduce any new mechanisms beyond those already in place for fault tolerance and dynamic load balancing.

6 Debugging

Debugging in Paralex does not involve the usual complexities of distributed debugging. The data flow paradigm with the strict enabling semantics is sufficient to guarantee deterministic executions even in a distributed system. Limiting the interaction of computations to function invocation renders the correctness of Paralex programs immune to communication delays and relative execution speeds. For the same reasons, the presence of probes for monitoring and controlling the program cannot interfere with the execution.

As for the complexities of programming and debugging applications to run in a heterogeneous distributed system despite failures, our approach has been to provide automated support so that the programmer need not confront them manually. The idea is to put the programmer to work on the *application* and not on the support layer. Having abstracted away the problems by automatically generating the code for heterogeneity, distribution and fault tolerance, we have also eliminated the need to debug these functions. Here we make the implicit assumption that Paralex itself and the ISIS support functions it uses are implemented correctly. We feel, however, that this is no different from basing the correctness of a sequential program on the correctness of the compiler used and the run-time support library functions invoked.

The debugging environment is fully integrated with the rest of Paralex — it shares the graphical interface with the development mode and uses the same graphical style of interactions. When a Paralex program is compiled with the debug option set, code generation for fault tolerance support and dynamic load balancing are suppressed. Thus, the execution to be debugged will be non-fault tolerant and not necessarily best performing. As the correctness of the application is independent of these aspects, the debugger does not need to address them.

The debugging features of Paralex consist of two elements: monitoring and controlling. The monitoring element provides visual feedback to the programmer about the state of the execution using the same graphical display from which the program was developed. The states of each node are distinguished as being *waiting for data*, *executing* and *stopped*. When executing, the name of the host on which the node is running is also displayed. It is also possible to trace the flow of data through selected links of the graph. By associating a *trace filter* with a link, only data values of interest can be displayed on the screen or written to a file as the program executes. Trace filters are specified as sequential programs in a manner similar to data filters for nodes. At program termination, Paralex produces a file of elapsed real time statistics for each node of the graph to be used for performance tuning purposes.

Implementation of the Paralex debugger requires no new mechanisms or abstractions. The same controller process used for dynamic load balancing serves also as the debugger run-time support. Recall that each node of the computation sends the controller a message containing the name of the host on which it is about to begin computing. The controller uses this information to update the graphical display with the various states of the nodes.

When compiled with the debugging option, each node of the Paralex program is modified to include a new link from a fictitious node, simulated by the controller. Neither the links nor this node are visible to the user. Having added a new link to each node, however, causes the computation to be delayed until there is an input present on the new link. The user-visible functional interface to the nodes remains unchanged and the input received from the controller is discarded by the stub code at each node. Using this simple mechanism, we are able to exercise complete control over the distributed computation for performing operations that are the analogues of breakpointing and single stepping.

7 Performance Results

Two different scientific applications were programmed with the current Paralex prototype and run on a network of Sun SPARCclassic workstations, each with 24 Megabytes of physical memory running SunOS 4.1.3. The environment used was a typical academic installation consisting of about one hundred hosts on a single standard 10-Megabit Ethernet segment. In all cases, the experiments were conducted with the hosts configured for standard production use with all network services and background system activity enabled. Except for the dynamic load balancing experiments, there were no external logins active during the course of the measurements. The hosts involved in the experiments were not isolated from the rest of the network, which continued to serve the normal internal Department and external Internet traffic. Communication and fault tolerance support were based on version 3.1 of the ISIS toolkit.

7.1 Volume Rendering of Atmospheric Data

For many scientific applications, the visualization of the results may be just as expensive as computing the results themselves. This is particularly true for computations that produce volumetric data [37]. For these applications, extensive computation is required to visualize the internal structure of three-dimensional object rather than simply their surfaces. As such, the visualization process may itself benefit from parallelization.

As an instance of this problem domain, we have used Paralex to program a volume rendering application for visualization of atmospheric model output. The original application, called StormView [3], was built at the University of Tromsø in Norway as part of the StormCast distributed system [31] and has been used by meteorologists to visualize the scalar fields generated by the atmospheric model LAM50S developed by the Norwegian Meteorological Institute. StormView uses a technique called *direct volume rendering* through ray casting whereby information between the iso-surfaces can be visualized without building the intermediate polygon structure.

Input to StormView consist of the *grid object* containing the 3D lattice of scalar values and the *render object* containing a rendering information including the image geometry, the position of the single light source and the perspective projected view point. The ray casting algorithm consists of tracking a ray for each pixel of the final rendered image plane by successively generating intersection paths and accumulating opacity and RGB-intensity along the ray until the grid extent is exhausted or the accumulated opacity exceeds the visibility threshold. The algorithm is easily parallelized by partitioning the image plane among multiple workers such that each can cast rays in parallel into the grid. To achieve complete independence among them, the input data (grid and render objects) need to be available in their entirety to each worker.

We have programmed 4-, 8- and 16-way parallel versions of the StormView application using Paralex. In each case, an input node reads the grid and render objects and broadcasts them to all the worker nodes. In addition to broadcasting the data, the input node statically partitions the image plane into equal-size rectangular regions and sends each worker the coordinates for the region it is responsible. Worker nodes execute identical code to implement the ray casting algorithm as described above. After all of the rays in its region have been traced, a worker sends to an output node the RGB-intensity and opacity values for its pixels. When all pixel values have been received, the output node performs blending of background images and color equalization.

Experiments were performed with the StormView application in order to evaluate the various features of Paralex. The data set used in the tests was the LAM50S atmospheric model relative humidity prognosis over continental Europe for 0:00 January 6, 1993. The model itself was run on a Cray Y-MP at Trondheim, Norway 48 hours earlier. The size of the StormView input data for this data set is a $121 \times 97 \times 18$ grid of scalars (2 bytes each) plus the render object for a total of 429748 bytes. The output image plane is 201×201 pixels amounting to 202005 bytes of color and opacity data. In all cases, the experiments were run for 4 iterations of the input data and the elapsed real time measured.

Parallelism	*Mapping Method*	
	M_1	M_2
Sequential	2023 (1.00)	2023 (1.00)
4-way	534 (3.78)	2120 (0.95)
8-way	269 (7.51)	2165 (0.93)
16-way	138 (14.57)	2170 (0.93)

Table 1. StormView Execution Times (in Seconds) and Speedups with no Replicas.

Parallelism	*Number of Replicas per Node*		
	0	1	2
4-way	534	540	545
8-way	269	281	286
16-way	138	154	165

Table 2. StormView Execution Times (in Seconds) as a Function of the Number of Replicas per Node.

In the first experiment, we compared the performance of the Paralex implementations of StormView to that of a sequential version coded in C and run as a conventional single Unix process. The results are summarized in Table 1 as the mean values of the elapsed times for the 4 iterations along with the resulting speedups (shown inside parentheses) with respect to the sequential version. The Paralex versions were coded for 4-, 8- and 16-way parallelism and run without any replicas (fault tolerance zero) under two different mappings. In mapping M_1, each worker node was run on a separate host while the input and output nodes were run on the same host. Mapping M_2, on the other hand, was obtained by defining a site file with a single host such that all nodes of the program were run on this host. Clearly, under M_2 there can be no real parallelism but all of the Paralex mechanisms for concurrent execution and data communication are exercised and thus it serves to quantify the overhead introduced by Paralex in order to achieve the possibility of parallel execution. From these results, we note that the overhead introduced by Paralex is less than 7% even for the 16-way version with respect to the non-Paralex sequential version. From the mapping M_1 results, we can clearly see that the application benefits from increased levels of parallelism and in all cases the speedup measured is within 90% of the theoretical limit (linear speedup).

The next experiments were designed to measure the cost of replicating nodes so as to facilitate fault tolerance and dynamic load balancing. We ran the 4-, 8- and 16-way parallel versions of the program with 0, 1 and 2 replicas. In all cases, the entire Paralex program was replicated mapped under M_1 such that a node and its replicas (if any) were on the same host. The purpose of this experiment was to quantify the communication (ISIS multicast) overhead in supporting replicas without actually invoking the fault tolerance or load balancing mechanisms. For this reason, it made sense to map each node and it replicas to a single host since doing so limited the number of hosts needed for the experiment to 16. From the results presented in Table 2, we can see that even in the worst case (16-way parallelism, 2 replicas) the performance degradation with respect to the non-replicated case is less than 20%. For the more typical case of a single replica, the degradation is only about 11%.

The final two experiments evaluated the fault tolerance and dynamic load balancing features of Paralex, both of which use the same coordinator-cohort mechanism of ISIS as described in Sections 4 and 5. For the results in Table 3, node failures were simulated by inserting "divide by zero" instructions into the code. Doing so not only triggered all of the failure detection and switch over mechanisms of ISIS, it also triggered the usual abnormal termination actions of Unix processes including production of core files. 4- and 8-way parallel versions of the program were

Parallelism	*Number of Concurrent Failures*		
	0	1	2
4-way	538	542	550
8-way	272	279	297

Table 3. StormView Execution Times (in Seconds) as a Function of Concurrent Failures.

Parallelism	*Concurrent Load*			
	ISIS make		StormView	
	Without	With	Without	With
4-way	1348	740	981	648
8-way	657	372	513	331

Table 4. StormView Execution Times (in Seconds) with and without Dynamic Load Balancing.

run with one replica for each node such that each node and its replica were on different hosts. This limited the experiments to 8-way parallel with single replica since we had a total of 16 uniform hosts available. In all cases, the failures were forced to occur at the beginning of an iteration. In this manner, we are able to isolate the costs inherent to the fault tolerance mechanisms from those due to lost work (which could be as large as 100% if failures occur right at the end). In the case of multiple failures, two different nodes of the program were forced to crash at the same iteration. From the results, we see that coping with two concurrent failures is a less than 9% increase in the execution time. For the case of single failures, the degradation is less than 3%.

To illustrate the effectiveness of the Paralex dynamic load balancing mechanism, we used the same configuration and mapping as the fault tolerance experiment with one replica per node. The two loads considered were the make of the ISIS system and the sequential version of the StormView program itself. The external load was started during the first iteration of the program on one of the hosts running a worker node. The mean time to complete one iteration for the 4- and 8-way parallel versions of the program are shown in Table 4 with dynamic load balancing enabled and disabled. When enabled, the load balancing mechanism effectively migrated the computation away from the loaded host by activating the replica of the node on the unloaded host for successive iterations. Thus, the execution times under dynamic load balancing degraded only for one iteration while those without load balancing were slowed down from beginning to end. From the results we can see that for the "make ISIS" load, dynamic load balancing resulted in about a 43% reduction of execution time while for the "StormView" load, the improvement was about 34%.

7.2 Synthetic Aperture Radar

The second application programmed was the Synthetic Aperture Radar (SAR) problem described in Section 3.2. Recall that SAR is a signal processing problem in which the contours of a terrain are reconstructed from radar phase and amplitude data reflected from a visually-obscured surface. The computational steps required by SAR consist of: two-dimensional discrete Fourier transform, binary convolution, two-dimensional inverse discrete Fourier transform and intensity level normalization for visualization.

Experiments were performed for input matrix dimensions ranging from 64×64 to 1024×1024 in order to compare the performance of the 4- and 8-way parallel implementations of SAR using Paralex with that of a sequential version coded in C. These experiments were run with fault tolerance set to zero, thus disabling replication. The mappings for the parallel executions were as follows: the input (reading the data and filter matrices) and output (display the final contour image) nodes were mapped to one host while the internal nodes were grouped into vertical chains and each mapped to a different host associated with that chain. This resulted in the 4- and the

Parallelism	*Input Dimension* 64×64	128×128	256×256	512×512	1024×1024
Sequential	0.51 (1.00)	2.08 (1.00)	8.90 (1.00)	40.15 (1.00)	179.90 (1.00)
4-way	0.49 (1.04)	1.17 (1.77)	3.88 (2.29)	15.78 (2.54)	69.60 (2.58)
8-way	0.71 (0.72)	1.18 (1.76)	3.10 (2.87)	9.60 (4.18)	41.11 (4.38)

Table 5. SAR Execution Times (in Seconds) and Speedups with no Replicas.

8-way parallel implementations using 5 and 9 hosts, respectively. Each experiment was repeated 10 times and the elapsed times to completion were measured. The results are summarized in Table 5 as the mean of the execution times in seconds. The speedup obtained with respect to the sequential version is enclosed inside parentheses next to each execution time.

For an input data of dimension $n \times n$, the overall complexity of the sequential computation is $O(n^2)$ and is determined by the input, binary convolution, intensity level normalization and output steps, which are linear in the number of data points. The intermediate steps (two-dimensional discrete Fourier transform and its inverse) are performed using $O(nlogn)$ algorithms and thus do not contribute to the overall complexity. The elapsed times for the sequential version indeed increase linearly with a 4.4-fold increase for each doubling of the matrix size. SAR happens to be a problem for which the communication-to-computation ratio is biased towards communication. For example, the 1024×1024 case represents more than 16 Megabytes of input data (radar signal plus the filter, both matrices of complex numbers with 8 bytes per entry) which needs to be communicated at each level of the computation graph. On the SPARCclassic processor, we can see that our sequential implementation requires less than 10 microseconds of computation per byte of input data. This time is comparable to the time it takes to communicate one byte between two nodes through the various software layers, including ISIS, and the network. Thus it is not surprising that the speedup results obtained are rather modest. In particular, for small input sizes (less than 128×128), it hardly pays to parallelize this application. Note that for the StormView application, each byte of input data resulted in about 4.7 milliseconds of computation, which is two orders of magnitude larger than that for SAR.

8 Related Work

Many of the goals of Paralex are shared by other systems that have been built or are currently under development. What is unique about Paralex, however, is the automatic support for fault tolerance and dynamic load balancing it provides and the extent to which the components of the system have been integrated into a single programming environment. For each Paralex design objective, we discuss other projects that share it or that have inspired our particular approach in dealing with it.

8.1 Network of Workstations as Multiprocessors

The idea of viewing a collection of workstations on a local-area network as a parallel multiprocessor has received extensive interest. There are numerous projects that have been experimenting with different abstractions to provide on top of such a system.

Distributed shared memory has emerged as a popular paradigm for programming parallel applications to run on a network of workstations. Systems that have been built based on this model include Amber [22], Munin [20] and Midway [11]. Ideally, the collection of distributed memory management systems collaborate to create the abstraction of a single, coherent shared address space. To make the technique more performant, this ideal strong *sequential consistency* semantics has been progressively weakened at the cost of putting more of the burden on the programmer or the compiler. The virtue of the model is its generality — any problem that can be solved in parallel in a shared memory multiprocessor can also be solved in a network of workstations. The drawback is the increased complexity of the application and/or the system support in order to guarantee correctness despite weak consistency semantics provided by the shared memory abstraction.

The shared memory approach to parallel computing is further explored by the Mentat system [29] in the context of an object-oriented programming language derived from C++. The user encapsulates data and computation as objects and the compiler performs all the necessary data flow analysis to permit parallel invocations whenever possible. The run-time system ensures correct object semantics even though all invocations are performed asynchronously. Yet another object-oriented approach to parallel programming is TOPSYS [10]. The basic programming abstraction in TOPSYS consists of tasks, mailboxes and semaphores realized through library routines. The emphasis of the system is a collection of graphical tools for performance monitoring and debugging.

The idea of broadcast-based parallel programming is explored in the `nigen++` system [2]. Using ISIS as the broadcast mechanism, `nigen++` supports a programming style not unlike the Connection Machine — a single master process distributes work to a collection of slave processes. This paradigm has been argued by Steele to simplify reasoning about asynchronous parallel computations without reducing their flexibility [40].

The system that perhaps comes closest to Paralex in its design goals and implementation is HeNCE [8, 9]. In HeNCE, the graphical representation of a computation captures the precedence relations among the various procedures. Data flow is implicit through syntactic matching of output names to parameter names. HeNCE graphs are dynamic in that subgraphs could be conditionally expanded, repeated or pipelined. Unlike Paralex, HeNCE has no automatic support for fault tolerance or dynamic load balancing.

8.2 Language and Architecture Heterogeneity

Our use of stubs for remote communication and universal data representation as ISIS messages derives its inspiration from Remote Procedure Call (RPC) systems [15]. Our use of these techniques, however, is to permit flow of data across (potentially) heterogeneous architectures rather than flow of control. The semantics of the remote invocation in Paralex might be called "remote function call without return" in the sense that a node supplies (partial) input to another node only upon completion. Results of the remote computation are passed on to other nodes rather than being returned to the invoker. Some notable systems where RPC has been employed to permit mixed-language programming and support for heterogeneous architectures include Mercury [36], MLP [30] and HRPC [12].

8.3 Graphical Programming

Examples of systems that use graphical notations to express parallel computations include Fel [32], Poker [38], CODE [17], Alex [33], LGDF [6, 7] and apE [26]. None of these systems addresses fault tolerance nor provides a programming environment in the sense of Paralex. Of these, perhaps CODE comes closest to Paralex in design goals. In addition to a graphical notation for parallel programs, it supports software reuse through a subsystem called ROPE [18]. The programming paradigm of CODE is based on the model of Browne [16] where programs are represented as generalized dependency graphs with each node having three sets of dependencies: input data, output data and exclusion. CODE does not support cyclic graph structures and the module interfaces are defined through declarative language. The program modules along with the interface definitions are compiled with the TOAD (translator of a declaration) subsystem to produce executables for specific parallel architectures [39]. As such, CODE needs the source versions of the program modules and cannot compose parallel programs out of pre-compiled binaries or libraries. LGDF proposes a graphical notation based on large-grain data flow similar to that of Paralex but lacks a programming environment and run-time support system. apE is an interesting system that shares with Paralex the data flow model of computation but is limited to scientific data visualization applications.

8.4 Architecture Independence

As with any other language proposal, Paralex strives for architecture independence. This is particularly true in the case of parallel programming languages since the competing models for parallel computation are still far too numerous. While the von Neumann architecture serves as the universal target for sequential programming languages, there exists no counterpart for parallel programming languages. There are two ways to address this problem: propose a model of parallel computation and hope that it will be accepted as "the" model, or propose programming languages that can be efficiently mapped to a variety of competing models. Recent proposals by Valiant [41] and Steele [40] fall into the first category. Systems and notations such as Paralex, Par [23], UNITY [21], Linda [28], CODE, P^3L [24], Prelude [42] and Phase Abstractions [35] fall into the second camp. In the case of Paralex, we inherit the properties of the data flow notation and keep further goals for architecture independence rather modest. Within the class of MIMD architectures, we strive for independence from synchrony attributes and communication mechanisms.

9 Discussion

Paralex does not require a parallel or distributed programming language to specify the node computations. The graphical structure superimposed on the sequential computation of the nodes effectively defines a "distributed programming language." In this language, the only form of remote communication is through passage of parameters for function invocation and the only form of synchronization is through function termination. While the resulting language may be rather restrictive, it is consistent with our objectives of being able to program distributed applications using only sequential programming constructs. This in turn facilitates reusability of existing sequential programs as building blocks for distributed parallel programs. We also note that this programming paradigm is in the spirit of that proposed by Steele [40] where a severely-restricted programming paradigm is advocated even for environments that support arbitrary asynchrony.

Paralex computation graphs are static. The only source of dynamism in the computation is the cyclic execution of a subgraph. This same mechanism could also be used to realize conditional subgraph expansion. Other dynamic structures such as variable-degree horizontal expansion or pipelined subgraphs are not possible. While this certainly restricts the expressiveness of Paralex as a language, we feel that it represents a reasonable compromise between simplicity and expressive power. Increasing the dynamism beyond that present would require Paralex to become a full-fledged functional programming language with all of the associated complexities.

The requirement that Paralex functions be pure with no side effects may result in nodes that iterate over large data structures inefficient since the entire structure must be passed from one iteration to the next. Paralex could be easily extended to permit persistent internal state to nodes. The internal group multicast by the coordinator to the cohorts, currently used to signal successful completion, will have to be augmented to communicate the updated values for the persistent variables. Otherwise, the fault tolerance mechanisms remain unchanged. The decision to require pure functions as nodes was made for the sake of simplicity of the current prototype.

10 Status and Conclusions

A prototype of Paralex is running on a network of m680x0, SPARC, MIPS and Vax-architecture Unix workstations. Paralex relies on Unix for supporting ISIS and the graphical user interface is based on X-Windows with the Open Look Intrinsics Toolkit.

The current implementation has a number of known shortcomings. As computations achieve realistic dimensions, executing each Paralex node as a separate Unix process incurs a large amount of system overhead. In the absence of shared libraries, each process must include its own copy of the ISIS services. This, plus the memory required to buffer each input and output of a node contribute to large memory consumption. We are working on restructuring the system to create a single Paralex process at each host and associate separate threads (ISIS tasks) with each node of the computation. In this manner, the library costs can be amortized over all the nodes and buffer memory consumption can be reduced through shared memory. Yet another limitation of the current implementation is its reliance on NFS as the repository for all executables. Obviously, the NFS server becomes a bottleneck not only for fault tolerance, but also for performance since it must supply the initial binary data and then act as the backing store for a large number of hosts. A satisfactory solution to this problem requires basing the storage system on a replicated, fault-tolerant file service.

The current implementation has no universal mechanisms or policies for exercising dynamic administrative control over computational resources. Availability of a host for executing computations on behalf of Paralex applications is all-or-nothing and static — inclusion of the host in the site definition allows arbitrary percentages of it to be used by Paralex, while exclusion of the host makes it unavailable even when it remains idle for extended periods. Ideally, owners of workstations would like to render their machines available to Paralex as long as they can regain their resources when needed. In principle, this can be achieved through the fault tolerance mechanisms of Paralex. A workstation owner could simply kill the Paralex computations active at his machine if there were replicas on other hosts. As far as Paralex is concerned, the failure of a computation due to an intentional kill is indistinguishable from a crash and would trigger the same fault tolerance mechanisms.

Paralex provides evidence that complex parallel applications can be developed and executed on distributed systems without having to program at low levels of abstraction. Many of the complexities of distributed systems such as communication, synchronization, remote execution, heterogeneity and failures can be made transparent automatically. Preliminary results indicate that the performance costs associated with this level of transparency are acceptable. First of all, parallel applications programmed using Paralex achieve satisfactory speedup. For the StormView application, we observed speedups that are within 90% of the theoretical limit. These results also compare very favorably with the those obtained by Asplin through hand coding and extensive tuning [4]. As for automatic support of fault tolerance and load balancing in typical cases, experimental results indicate that replicating each node of a program once and activating the replica switch over mechanisms degrades performance by less than 10%.

Initial results with Paralex as a tool for tapping the enormous parallel computing resource that a network of workstations are very positive. Further experience is necessary to demonstrate its effectiveness as a tool to solve a wider class of practical problems.

Acknowledgements Giuseppe Serazzi and his group at the University of Milan contributed to early discussions on the mapping and dynamic load balancing strategies. Ken Birman and Keshav Pingali of Cornell University were helpful in clarifying many design and implementation issues. Dave Forslund of Los Alamos provided valuable feedback on an early prototype of the system. Jo Asplin of Tromsø kindly gave us his StormView code and helped with the Paralex implementation. Alberto Baronio, Marco Grossi, Susanna Lambertini, Manuela Prati, Alessandro Predieri and Nicola Samoggia of the Paralex group at Bologna contributed to the various phases of the implementation. The presentation of the paper has benefited from suggestions by the anonymous referees. We are grateful to all of them.

References

[1] J.-L. Gaudiot and L. Bic. *Advanced Topics in Dataflow Computing*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[2] R. Anand, D. Lea and D.W. Forslund. Using `nigen++`. Technical Report, School of Computer and Information Science, Syracuse University, January 1991.

[3] J. Asplin. Distributed Parallel Volume Rendering of Atmospheric Model Output. In *Proceedings, Visualisering '93*, pages 66–76. SINTEF Industriell matematikk, Norges Tekniske Hxgskole, University of Linköping, June 1993.

[4] J. Asplin. Distribuert Parallell Volumvisualisering av Skalarfelt fra en Atmosffremodell. M.Sc. Thesis, Department of Computer Science, University of Tromsø, Norway, March 1994.

[5] Ö. Babaoğlu, L. Alvisi, A. Amoroso and R. Davoli. Mapping Parallel Computations onto Distributed Systems in Paralex. In *Proc. IEEE CompEuro '91 Conference*, Bologna, Italy, May 1991, pp. 123–130.

[6] R.G. Babb II. Parallel Processing with Large-Grain Data Flow Techniques. *IEEE Computer,* July 1984, pp. 55–61.

[7] R.G. Babb II. Issues in the Specification and Design of Parallel Programs. In *Proc. 6th International Workshop on Software Specification and Design*, October 1991, pp. 75–82.

[8] A. Beguelin, J.J. Dongarra, G.A. Geist, R. Manchek and V.S. Sunderam. Graphical Development Tools for Network-Based Concurrent Supercomputing. In *Proc. Supercomputing '91*, November 1991, Albuquerque, New Mexico.

[9] A. Beguelin, J.J. Dongarra, G.A. Geist and V.S. Sunderam. Visualization and Debugging in a Heterogeneous Environment. *IEEE Computer*, vol. 26, no. 6, June 1993, pp. 88–95.

[10] T. Bemmerl, A. Bode, *et al.* TOPSYS — Tools for Parallel Systems. SFB-Bericht 342/9/90A, Technische Universität München, Munich, Germany, January 1990.

[11] B.N. Bershad and M.J. Zekauskas. Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, September 1991.

[12] B.N. Bershad, D.T. Ching, E.D. Lazowska, J. Sanislo and M. Schwartz. A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems. *IEEE Trans. on Software Engineering*, vol. SE-13, no. 6, August 1987, pp. 880–894.

[13] K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck and M. Wood. The ISIS System Manual, Version 2.1. Department of Computer Science, Cornell University, Ithaca, New York, September 1990.

[14] K. Birman. The Process Group Approach to Reliable Distributed Computing. *Comm. of the ACM,* vol. 123, December 1993, pp. 36–53.

[15] A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. on Computer Systems*, vol. 2, no. 1, February 1984, pp. 39–59.

[16] J.C. Browne. Formulation and Programming of Parallel Computations: A Unified Approach. In *Proc. Int. Conf. Parallel Processing*, Los Alamitos, California, 1985, pp. 624–631.

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

[18] J.C. Browne, T. Lee and J. Werth. Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment. *IEEE Trans. on Software Engineering*, vol. 16, no. 2, February 1990, pp. 111–120.

[19] N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys,* vol. 21, no. 3, September 1989, pp. 323–358.

[20] J.B. Carter, J.K. Bennett and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. Thirteenth ACM SIGOPS Symposium on Operating Systems Principles*, Pacific Grove, California, October 1991, pp. 152–164.

[21] K.M. Chandy. Programming Parallel Computers. Technical Report Caltech-CS-TR-88-16, Department of Computer Science, California Institute of Technology, August 1988.

[22] J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy and R.J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proc. Twelfth ACM SIGOPS Symposium on Operating Systems Principles*, Litchfield Park, Arizona, December 1989, pp. 147–158.

[23] M.H. Coffin and G.R. Andrews. Towards Architecture-Independent Parallel Programming. Technical Report TR 89-21a, Department of Computer Science, University of Arizona, Tucson, Arizona, December 1989.

[24] M. Danelutto, R. Di Meglio, S. Pelegatti and M. Vanneschi. High Level Language Constructs for Massively Parallel Computing. In *Proc. Sixth International Symposium on Computer and Information Sciences*, Elsevier North-Holland, October 1991.

[25] K. Dixit. SPECulations: Defining the SPEC Benchmark. *SunTech Journal*, vol. 4, no. 1, January 1991, pp. 53–65.

[26] D.S. Dyer. A Dataflow Toolkit for Visualization. *IEEE Computer Graphics and Applications*, July 1990, pp. 60–69.

[27] E. Fairfield. Private communication. Los Alamos National Laboratory, New Mexico.

[28] D. Gelernter, N. Carriero, S. Chandran and S. Chang. Parallel Programming in Linda. In *Proc. Int. Conf. Parallel Processing*, St. Charles, Illinois, August 1985, pp. 255–263.

[29] A.S. Grimshaw. Easy-to-Use Object-Oriented Parallel Processing with Mentat. *IEEE Computer*, vol. 26, no. 5, May 1993, pp. 39–51.

[30] R. Hayes, S.W. Manweiler and R. D. Schlichting. A Simple System for Constructing Distributed, Mixed Language Programs. *Software–Practice and Experience*, vol. 18, no. 7, July 1988, pp. 641–660.

[31] D. Johansen. StormCast: Yet Another Excercise in Distributed Computing. In Frances Brazier and Dag Johansen, editors, *Distributed Open Systems*. IEEE Computer Society Press, New York, 1993.

[32] R.M. Keller and W-C.J. Yen. A Graphical Approach to Software Development Using Function Graphs. In *Proc. Compcon Spring 1981*, CS Press, Los Alamitos, California, pp. 156–161.

[33] D. Kozen, T. Teitelbaum, W. Chen, J. Field, W. Pugh and B. Vander Zanden. ALEX: An Alexical Programming Language. In *Visual Programming Languages,* Ed. Korfnage, Plenum Press.

[34] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM,* vol. 21, no. 7, July 1978, pp. 558–565.

[35] C. Lin and L. Snyder. Portable Parallel Programming: Cross Machine Comparisons for SIMPLE. In *Fifth SIAM Conference on Parallel Processing*, 1991.

[36] B. Liskov, *et al.* Communication in the Mercury System. In *Proc. 21st Annual Hawaii Conference on System Sciences*, January 1988.

[37] V. Ranjan and A. Fournier. Volume Models for Volumetric Data. *IEEE Computer*, vol. 27, no. 7, July 1994, pp. 28–36.

[38] L. Snyder. Parallel Programming and the Poker Programming Environment. *IEEE Computer,* vol. 17, no. 7, July 1984, pp. 27–36.

[39] S.M. Sobek. A Constructive Unifoed Model of Parallel Computation. Ph.D. Dissertation, Department of Computer Sciences, University of Texas at Austin, Austin, Texas, December 1990.

[40] G.L. Steele Jr. Making Asynchronous Parallelism Safe for the World. In. *Proc. 17th Annual ACM Symposium on Principles of Programming Languages*, 1990, pp. 218–231.

[41] L.G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, vol. 33, no. 8, August 1990, pp. 103–111.

[42] W. Weihl, E. Brewer, A. Colbrook, C. Dellarocas, W. Hsieh, A. Joseph, C. Waldspurger and P. Wang. Prelude: A System for Portable Parallel Software. In *Proc. Conference on Parallel Architectures and Languages in Europe (PARLE'92)*, 1992.