Paralex: An Environment for Parallel Programming in Distributed Systems*

Özalp Babaoğlu
Lorenzo Alvisi
Alessandro Amoroso
Renzo Davoli

Department of Mathematics
University of Bologna
Piazza Porta S. Donato, 5
40127 Bologna, Italy

ozalp@dm.unibo.it

Technical Report UB-LCS-91-01

6 February 1991

Abstract

It is our thesis that a distributed system can be programmed, monitored, debugged and generally thought of as if it were a single, uniform multiprocessor machine. The technical issues that need to be addressed in supporting this abstraction include fault tolerance, architectural heterogeneity and high latency of the communication network. The Paralex Project is aimed at testing this thesis for the class of applications that can be described using a data flow-like semantics. The interface presented to the programmer makes extensive use of graphics to define, edit, execute and debug applications. All of the necessary code for distributing the application across a network and replicating it to achieve the desired level of fault tolerance is automatically generated by the system. In this paper we give an overview of Paralex and present the status of a prototype implementation.

*This work was supported in part by the Commission of European Communities under ESPRIT Programme Basic Research Action Number 3092 (Predictably Dependable Computing Systems) and the Italian Ministry of University, Research and Technology.

1 Introduction

There is general agreement that significant future increases in computing power will be possible only through exploiting parallelism. One of the many advantages of distributed systems is their ability to execute several computations on behalf of a single application in parallel, thus improving performance. In fact, at a certain level of abstraction, there is little difference between a distributed system and a single loosely-coupled multiprocessor computer. We cannot, however, treat a distributed system as if it were a uniform multiprocessor parallel machine due to the following characteristics which are typical of distributed systems:

- High latency, low bandwidth communication between processors.
- Presence of multiple processor architectures.
- Failures of both communication links and processors.
- Presence of multiple independent administrative domains.

Thus, if we can address these issues, a distributed computing resource such as a collection of workstations could be viewed and used as if it were a poor man's "super computer." To make a distributed system suitable for long-running parallel computations, support must be provided for fault tolerance. 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 components of the system (workstations) may be under the control of several independent administrative domains (typically a single individual who "owns" the workstation), these events are much more plausible and frequent than real hardware failures.

The proliferation of local-area networks with large numbers of workstations represents significant resources begging to be tapped for parallel computing. It is our belief that reasonable technologies already exist to address the problems related to distribution, communication and fault tolerance of applications in distributed systems. What remains a challenge is the task of *programming* reliable applications that can benefit from the parallelism and fault tolerance that distributed systems have to offer. The Paralex Project has been undertaken to explore the extent to which the programmer can be liberated 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. In fact, the programming paradigm supported by Paralex is best suited for parallel computations that can be viewed as collages of ordinary sequential programs. The interdependencies and data flow relations between computations in a parallel program are expressed in a natural way using a graphical notation. In the limit, interesting new parallel programs can be "programmed" 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" [12].

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 copes

with data representation in a heterogeneous environment and uses the ISIS toolkit to automatically render programs fault tolerant. Section 5 discusses the problem of mapping Paralex computations onto hosts in a distributed system. The Paralex debugger is described in Section 6. Related work is compared and contrasted to Paralex in Section 7. Section 8 gives a status report of the current prototype and concludes the paper.

2 The Paralex Computational Model

The choices made for programming paradigm and notation are fundamental to harness parallelism in a particular application domain [13]. The programming paradigm supported by Paralex is a restricted form of data flow [1]. A Paralex program is composed of *nodes* and *links*. Nodes correspond to computations and the links indicate the flow of (typed) data. Thus, Paralex programs can be though 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. The current prototype limits the structure of the data flow graph to be acyclic. Acyclic graphical notations have also been proposed for structuring module interface import/export relations in the context of real-time operating systems [31].

Unlike classical data flow, the nodes of a Paralex program carry out significant computations. This so-called *coarse-grain* data flow model [4] is a consequence of the underlying distributed system architecture where we seek to keep the communication overhead via a high-latency, low-bandwidth network to reasonable levels. The model is further motivated by our desire to provide automatic support for heterogeneity and fault tolerance and facilitate software reuse. How Paralex addresses each of these issues will be discussed in subsequent sections.

2.1 Computation Nodes

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 links incident at a node contain values, the computation associated with the node starts execution transforming the input data to an output. The computation to be performed by the node must satisfy the "functional" paradigm—multiple inputs, only one output with no side effects. The actual specification of the computation may be done using whatever appropriate notation is available including standard sequential programming languages, parallel programming notations (if the distributed system includes nodes that are themselves multiprocessors), executable binary code or library functions for the relevant architectures. For reasons that will become clear when we discuss fault tolerance, the computation carried out by a node is restricted not to have any permanent internal state nor have any external side effects. In particular, interactions with external components such as files, devices or other computations are not permitted.

Limiting interactions between computations to the passage of data at well-defined boundaries is the foundation of our approach. Not only does it admit simple mechanisms to render the computations fault tolerant, it also allows us to deal with language and architecture heterogeneity. It is also this "black-box" view of computations that permits us to reuse sequential code in constructing parallel programs. Keeping the computation and data flow aspects of a program separate in

Paralex, we are able to incorporate existing code fragments in parallel programs without having to add any new language constructs for data communication. Given the functional nature of the computations, the graphical interconnection representation itself is effectively a data communication "language." Other proposals for basing distributed computations on a data flow-like model with language extensions include the "box-and-string" model of [9] and the CODE system [11].

Note that while nodes can produce only single output values upon termination, there is no reason why the value produced has to be limited to scalars. It is quite possible for a computation to produce composite data types such as arrays, matrices or records as output. When a program is compiled, Paralex will automatically generate all of the necessary code to wrap around a node to achieve data communication. Paralex also checks type consistency across node boundaries and generates the code necessary for data encoding/decoding to achieve a common data representation across different architectures.

2.2 Filter Nodes

There are many situations where the single output value produced by a node needs to be communicated to multiple destinations as input so as to create parallel computation structures that are more than just linear. In Paralex, this is accomplished simply by drawing multiple output links originating from a node towards the various destinations. In computations that contain nodes with large degrees of fan-out where the next stages need only a small subset of the voluminous data produced, this "broadcasting" scheme of disseminating results could be costly in terms of network bandwidth.

Paralex introduces the notion of *filter* nodes that allow data values to be extracted on a per-destination basis before they are transmitted to the next node. 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 computations are executed in the context of the single process that produced the data rather than as separate processes to minimize the system overhead. This logical separation between data extraction/communication and data transformation is again motivated by our desire to permit sequential code to be inserted in computation nodes without having to invent new language constructs.

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 such that the programmer need not switch back and forth between them. It is, however, possible to run the compiler or the executor without the graphical support so that existing Paralex programs can be compiled and executed 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 a crude "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.

```
elettra      sparc, fpu, graphics=2, color, specmarks=21
leporello    sparc, fpu, specmarks=13
dongiovanni  sparc, fpu, graphics, specmarks=10
nabucco      mips, fpu, graphics, specmarks=18
tosca        m68020, fpu, specmarks=9
violetta     m68020, specmarks=5
turandot     prism, fpu, specmarks=12
carmen       vax, graphics, specmarks=6
jago         dongiovanni
otello       dongiovanni
fyodor       dongiovanni
```

Figure 1: Paralex Site Definition File.

The attributes may be binary (e.g., `sparc, fpu`) or numeric (e.g., `graphics, specmarks`). Including the name of a binary attribute for a host signals its presence. Numeric attributes are associated values through assignment and are set to one by default. Including a host name as an attribute permits two hosts to share the same set of attributes. Paralex neither defines nor interprets keywords as attribute names. They are used by the Paralex loader to select 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 should contain its name and the processor architecture family (e.g., `sparc, mips, vax`). In case two hosts share the same processor architecture family but are not binary compatible, additional information (such as the operating system, `SunOS, A/UX`) must be included in the characterization to distinguish them.

The site definitions of this global file can be modified by each user through a similar file in the user's home directory. The local modifications, however, will have effect only if they represent restrictions with respect to the global definitions. For example, a user cannot include a host that had not been foreseen in the global definition or increase the numerical value of an attribute beyond that set globally. This policy prevents users from abusing system resources that are not intended for public use.

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. Future extensions may include explicit network topology information to support parallel computing over non-homogeneous and long-haul communication networks.

3.2 The Graphics Editor

The Paralex graphics editor allows the data flow relations of a program to be specified in a simple and natural manner. The user interactions are based on the classical "point and click" paradigm with pop-up menus selecting operations. Figure 2 illustrates a Paralex window along with the root menu.

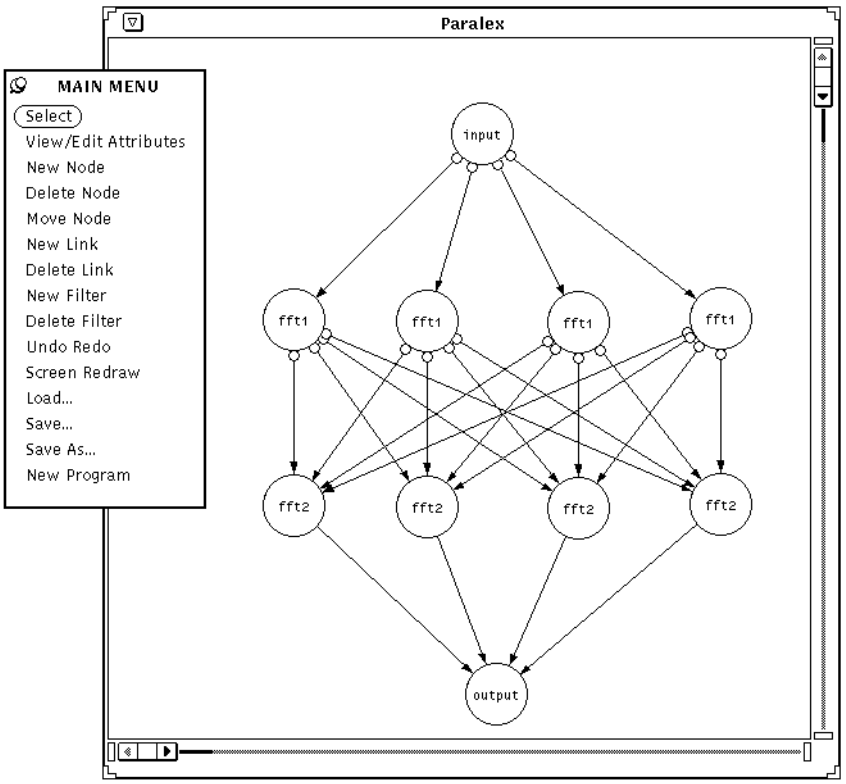


Figure 2: Paralex Graphics Editor with Root Menu.

The program displayed is a parallel Fast Fourier Transform (FFT). It divides the input vector into four segments that are transformed in parallel at two levels with the appropriate data permutation between the two levels. This algorithm is one of several ways of parallelizing the FFT problem.

To create a program such as the one illustrated, the user first selects the *New Node* item from the menu and designs the nodes of the graph by clicking at the appropriate locations on the canvas.

Next, the nodes are interconnected by selecting *New Link* and clicking at the source and the sink of the desired link. Filter nodes can be inserted to existing links by clicking on them after having selected *New Filter*. Filter nodes are visualized as small circles attached to the source node of a link. At any stage of the editing, objects on the screen can be deleted or moved about. Paralex also permits cut-and-paste operations and undoing the last edit function performed. The current screen contents can be saved for future use or an existing program can be loaded for editing. Paralex saves program graphs as 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 support for the graphics editor environment.

3.3 Specifying Nodes

Next stage in program development is to specify the attributes of a node including the computation to be carried out. This is done by selecting *View/Edit Attributes* from the root menu and clicking on the desired node. The result of performing this operation on the highlighted node "sort" is shown in Figure 3.

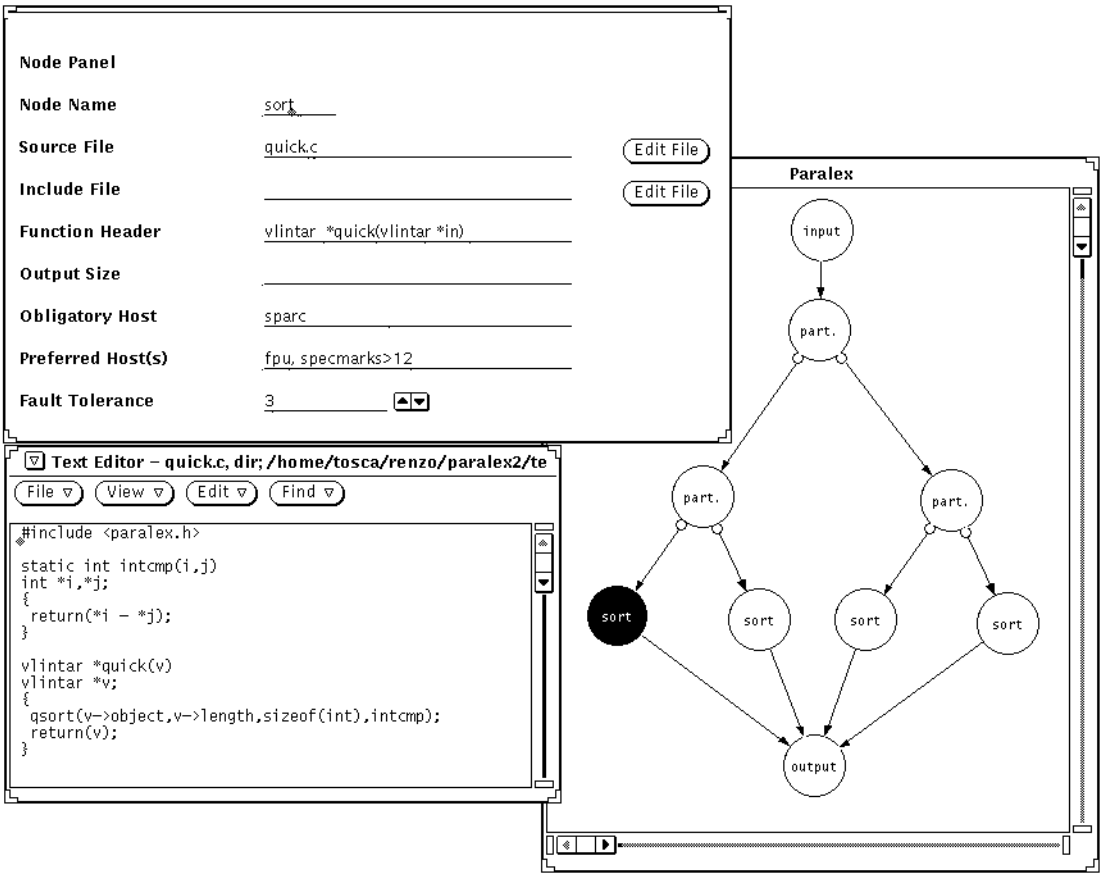


Figure 3: The Program Quick Sort and the Node Panel.

The program illustrated is a parallel sort inspired by the quick sort algorithm. After two levels of partitioning, four sequential sort programs are invoked on each of the resulting four vectors. The output simply concatenates the four segments. The example also illustrates several characteristics of Paralex as a programming language. The computation graph is static and cannot be a function of either the size or the values of the input data. Since loops and recursion are not allowed, recursive structures must be represented by their unfolded equivalents. In our example, the decision to halt the partitioning after two levels is based on granularity considerations—given an input data size, extending the partitioning to levels where each node computes with just a few data items would result in communication costs offsetting all benefits derived from parallel computation.

The node panel of Figure 3 contains the following fields necessary to specify a node:

Node Name Used to identify the node on the screen.

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

Include File Name of the file to be included before compilation. Used to include user-defined data types in a Paralex program.

Function Header The interface specification for the function computed by the node. Includes all input data types and the output data type. The syntax used is ANSI C. Paralex will fill this field automatically if the source file indicated above contains a single function definition. The association between function input parameters and links of the computation graph is accomplished graphically by pointing and clicking.

Output Size In case the output type of a node is an array, this field indicates the number of elements it contains. Requiring the user to specify this number is an artifact of C syntax where arrays can be declared as pointers.

Obligatory Host A set of attributes that must be satisfied by a host to execute the node. The attributes may include host names, an architecture class and hardware configuration. Used primarily for including compiled binaries or library functions of a specific architecture as node computations.

Preferred Host Similar to obligatory host except used as a hint to the Paralex loader for performance reasons rather than as a requirement.

Fault Tolerance The execution of the node will succeed despite this number of host failures. Default is zero (no fault tolerance).

Obligatory and preferred host specifications are done using the site definition file described in the previous section. A query is formulated by simply listing the attributes of the desired host. In case of numeric attributes, the query may contain integer comparison operators and constants. For example, the query

```
sparc, fpu, graphics, specmarks>10
```

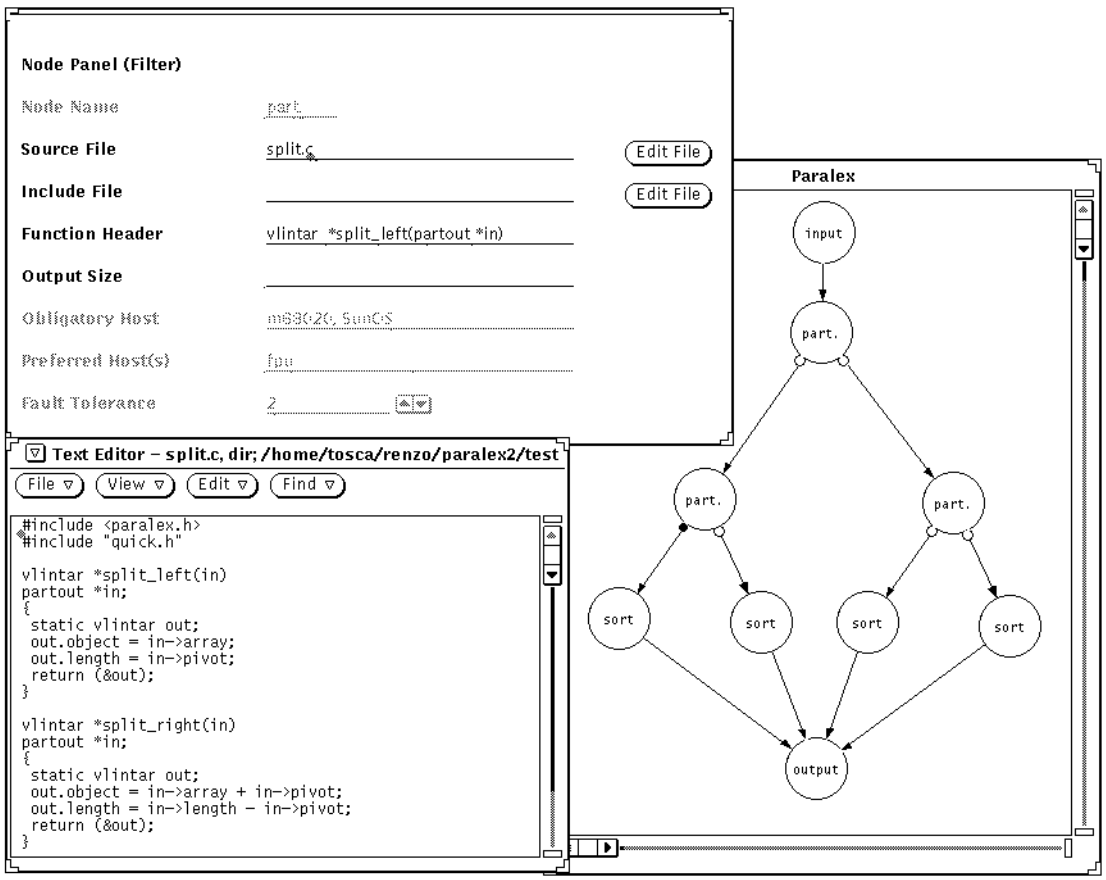


Figure 4: The Program Quick Sort and the Filter Panel.

would match all hosts that contain `sparc`, `fpu` and `graphics` as attributes and have a `specmarks` value of at least 10. Recall that up to this point Paralex does not associate any semantics with these tokens but simply uses them to construct a set of hosts that match the query.

The obligatory host field should be used to indicate the weakest requirements for architectural and functionality reasons while the preferred host field typically adds performance-related queries. In case a particular query produces an empty set of candidate hosts at a particular site, the graphics editor signals a warning. A warning is also generated if the desired replication level 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 File* 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 "quick.c" containing the source code for the node "sort". In this example, the node makes use of the C library `qsort` function to do the actual computation. Note that the code is completely ordinary sequential C and contains absolutely nothing having to do with remote communication or fault tolerance. The data

type `vlintar` is a variable-length integer array defined by Paralex. Paralex also includes definitions for variable-length arrays for all of the other basic data types.

3.4 Specifying Filters

The attributes of a filter node are specified and edited exactly in the same manner as an ordinary node. The only difference is that most of the fields in the node panel are inherited from the parent node that the filter is associated with and are not modifyable (indicated as shaded fields in the panel). Figure 4 illustrates the node panel and source code associated with the left filter of the left second-level partition node.

The output of the partition node is a structure of type `partout` containing a variable length array and the index of the pivot element defining the partitioning. The source file displayed in the editor window contains the code for both filters of the partition node. The left filter of the example (function `split_left`) does nothing more than pass along the left portion of the array up to the pivot element.

3.5 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 graphics editor saves the structural information of a Paralex program as simple text with a simple syntax. This file along with all of the source code for the node computations 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 developed.

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 and replica management for those nodes with fault tolerance needs. 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, the C compiler is invoked to turn each node into an executable module. In future versions, we plan to support multi-language programs written in any combination of C, Fortran, C++ or Lisp.

The compiler must also address the two aspects of heterogeneity: Data representation and instruction sets. Paralex uses the ISIS toolkit [6,7] 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, float, char) 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 in a brute-force manner 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) of the current program.

3.6 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. It is described in Section 6. 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.

Before a Paralex program can be loaded and begin execution, each node of the computation (and its replicas in case fault tolerance is required) must be associated with a particular host of the distributed system. The algorithms to resolve this mapping are embodied in the loader and are described in Section 5. The executable code for each node is copied to the host(s) as determined by these algorithms. Each node is executed as a Unix process that contains both the computation for the node and all of its associated filters.

The controller is part of the Paralex run-time environment and implements dynamic load balancing decisions. It consists of daemon processes running on each of the hosts defined for the site and a controller process that runs on the host from which the program was launched. The daemon processes are used to gather dynamic load information required by the balancing algorithms as described in Section 5.5. The dynamic information collected by the controller is also passed along to the graphical display (if one exists) to provide visual feedback to the programmer.

4 Fault Tolerance

One of the primary characteristics that distinguishes a distributed system from a uniform multiprocessor is the increased possibility of failures during computations. As noted earlier, these may be due to real hardware failures or, more probably, as a consequence 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 library contains the primitives necessary to support fault tolerance and achieve universal data representation.

As part of the program definition, Paralex permits the user to specify a fault tolerance level for each node of the computation graph. Paralex will generate all of the necessary code such that when a node with fault tolerance k is executed, it will be executed on $k + 1$ distinct processors such that the computation will succeed despite up to k failures. Obviously, the number of physical processors available in the distributed system defines an upper limit on the fault tolerance that can be satisfied. The 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 an Ethernet cable.

While it is possible to specify a fault tolerance level for each individual node of a computation, it is typically more appropriate to specify it globally for the entire computation. It may make sense to set different levels of fault tolerance for different nodes if the host reliabilities or the node execution times are highly non-uniform.

Paralex uses ISIS also as the infrastructure to support fault tolerance. For each node that requires fault tolerance, Paralex creates a process group consisting of replicas for the computation. The

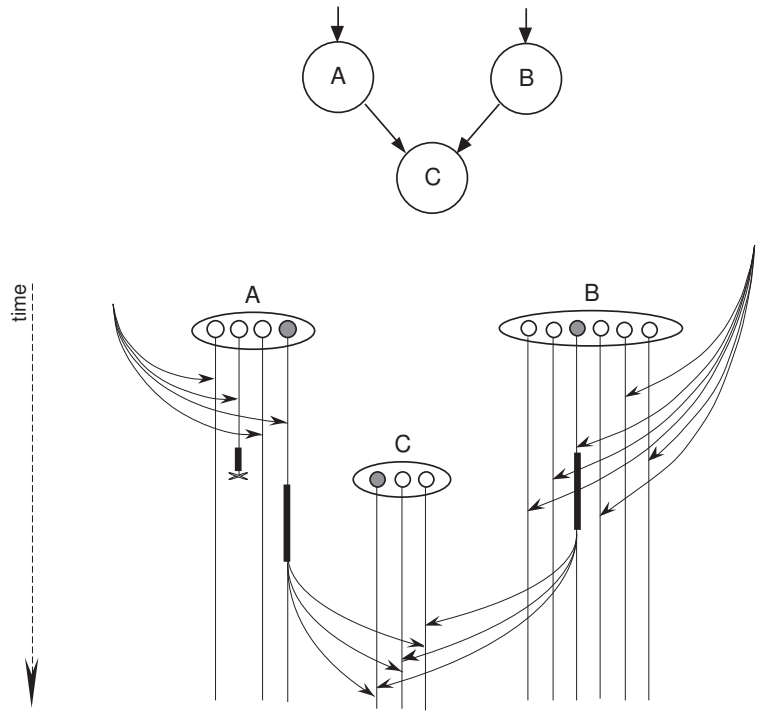


Figure 5: Replication of Nodes and Transformation of Data Communication to Broadcasts.

type of process group formed is the ISIS *coordinator-cohort* where only one of the replicas (the coordinator) carries out the computation and the others (the cohorts) are invoked automatically only in case of failure of the coordinator. As illustrated in Figure 5, simple data transmissions from one node to another turn into ISIS reliable broadcast communication from a coordinator to a process group representing the destination node. The actual flavor of ISIS broadcast primitive we use is CBCAST that guarantees atomicity and delivery order within the process group that is consistent with potential causality [28].

5 Mapping Computations to Hosts

We return to the question of loading Paralex programs and examine the algorithms employed to map the computation graph onto the hosts 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 requirements. Formally, the mapping problem can be viewed as a constrained optimization where the computation graph is embedded in a *system graph*. The nodes of this system graph are the hosts and edges correspond to the physical communication links that exist in the distributed system. Since our primary targets are common distributed systems with broadcast-based communication architectures (such as Ethernet), the mapping and load balancing schemes we consider assume system graphs that are uniform and fully connected.

In its most general form, the mapping problem can be formulated as an integer programming optimization. As such, it is computationally intractable and known to be NP-complete [21]. Our approach in Paralex to this problem is much more pragmatic—we want fast, simple heuristics to map computations onto distributed systems without having to solve complex, expensive optimization problems. Furthermore, the heuristics should be based on information that is available from the computation definition (structure, data types, etc.) and *not* on speculative information such as node execution times. It is our belief that the losses due to the sub-optimality of our solutions will be more than offset by the expedient launching of programs.

For the purposes of mapping Paralex programs to hosts, the unit of concern is a computation node. Since filter nodes are associated with the source computation node and are executed in its context as a single process, they are implicitly mapped. Given the obligatory and preferred host specifications for the nodes of the computation and the site definition file, the Paralex loader constructs two sets containing host names for each node. The set OH_i contains all hosts that can execute node i while the set PH_i is a subset of OH_i containing those hosts that also satisfy the preferred host specification. The mapping algorithm presented in the next section assumes that each node may execute on any host of the system. How we incorporate the restrictions placed by architectural constraints is addressed in Section 5.2.

5.1 Grouping Nodes into Chains

Given the strict enabling semantics of Paralex computations, all nodes that lie along a chain of data flow edges have to be executed sequentially. Thus, it behooves us to map them to a single host keeping all of the data communication along the chain local. Our first goal is to distribute the computation while maximizing parallelism and minimizing non-local data communication.

To this end, we annotate the computation graph by including the number of bytes flowing through each link as its weight. This information is extracted from the type definitions associated with the node interfaces. In case of variable-length data types such as arrays, the weight of the link is set to the maximum of any other fixed-length weight. The strategy then becomes: Identify the maximum number of chains that can be executed in parallel while minimizing the volume of data that flows between chains. While the optimal clustering of nodes with this objective requires solving an NP-complete problem [21], there are efficient heuristics that give acceptable suboptimal solutions.

We define a *chain* as a subgraph of the computation graph such that there is a unique sequence of links that visits all nodes. In other words, all of the nodes of the chain must have sequential precedence constraints. An optimal algorithm would identify a number of chains equal to the width of the graph such that the inter-chain communication is minimized. We adopt a greedy heuristic algorithm that is based on the clustering algorithm of Houstis [25].

The algorithm starts with a matrix representation of the computation graph. Given a computation graph with n nodes, its *weighted adjacency matrix* representation is denoted A and contains data volume between nodes as the weights. From A, we construct the *parallelism matrix*, denoted P to encode the dependency relations between nodes as follows:

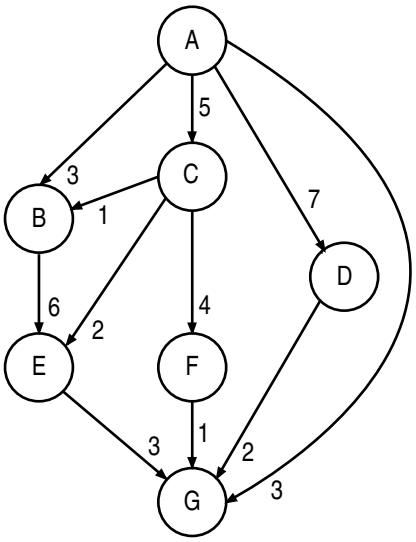
```
procedure ParallelismMatrix(A, var P)
begin
    forall indexes i, j do
    begin
        P[i, j] := 0
        if A[i, j] ≠ 0 then
            P[i, j] := 1
    end
    compute the transitive closure of P in W
    forall indexes i, j do
        if W[i, j] = 1 and P[i, j] = 0 then
            P[i, j] := 2
end

procedure BuildChains(A, P, var C)
begin
    C = {{1}, {2}, {3}, ..., {n}}
    while (P[i, j] = 1 for some i, j) do
    begin
        find (K, L) such that A[K, L] = max{A[U, V] : P[U, V] = 1}
        T := K ∪ L
        C := C − {K} − {L}
        forall s ∈ C do
        begin
            A[T, s] := A[K, s] + A[L, s]
            A[s, T] := A[s, K] + A[s, L]
            P[T, s] := min{P[K, s], P[L, s]}
            P[s, T] := min{P[s, K], P[s, L]}
        end
        A[T, T] := 0
        P[T, T] := 0
        C := C ∪ {T}
    end
end
```

Figure 6: Algorithms to Construct the Parallelism Matrix and Chains.

$P[i,j] = 1$ if i and j are connected directly
$P[i,j] = 2$ if i and j are connected indirectly
$P[i,j] = 0$ if i and j are parallel.

The algorithm to construct matrix P from A is shown as procedure *ParallelismMatrix* in Figure 6. Given the matrices A and P, the chain construction algorithm proceeds by joining pairs of nodes that are connected by the highest weight link and that are not parallel. The step is repeated until no further nodes can be joined. The procedure *BuildChains* of Figure 6 returns in C a set of sets corresponding to the chains of the computation graph. To simplify notation, we assume that the rows and columns of matrices A and P can be addressed using sets.



Weighted-Adjacency matrix

	A	B	C	D	E	F	G
A	0	3	5	7	0	0	3
B	0	0	0	0	6	0	0
C	0	1	0	0	2	4	0
D	0	0	0	0	0	0	2
E	0	0	0	0	0	0	3
F	0	0	0	0	0	0	1
G	0	0	0	0	0	0	0

Connectivity matrix

	A	B	C	D	E	F	G
A	0	1	1	1	2	2	1
B	0	0	0	0	1	0	2
C	0	1	0	0	1	1	2
D	0	0	0	0	0	0	1
E	0	0	0	0	0	0	1
F	0	0	0	0	0	0	1
G	0	0	0	0	0	0	0

Figure 7: A Computation Graph and its Adjacency and Parallelism Matrices.

The algorithm is best understood through an example. Figure 7 shows a seven-node computation graph along with its adjacency matrix representation and its parallelism matrix. The application of the chain construction algorithm of Figure 6 is illustrated in Figure 8.

The algorithm begins by placing each node in a chain by itself. As the first step, the chains containing nodes A and D are unified since they communicate most intensely and are not parallel. Nodes A and D are deleted and replaced by a single node representing the chain $\{A, D\}$. The weights and the parallelism information are updated to reflect the unification resulting in the graph of Figure 8 (b). Note that nodes B and C that used to be sequential with respect to A become parallel with respect to the new chain $\{A, D\}$. The output of the algorithm for this example results in the three chains $\{B, E, G\}$, $\{C, F\}$, $\{A, D\}$ as shown in Figure 8 (e).

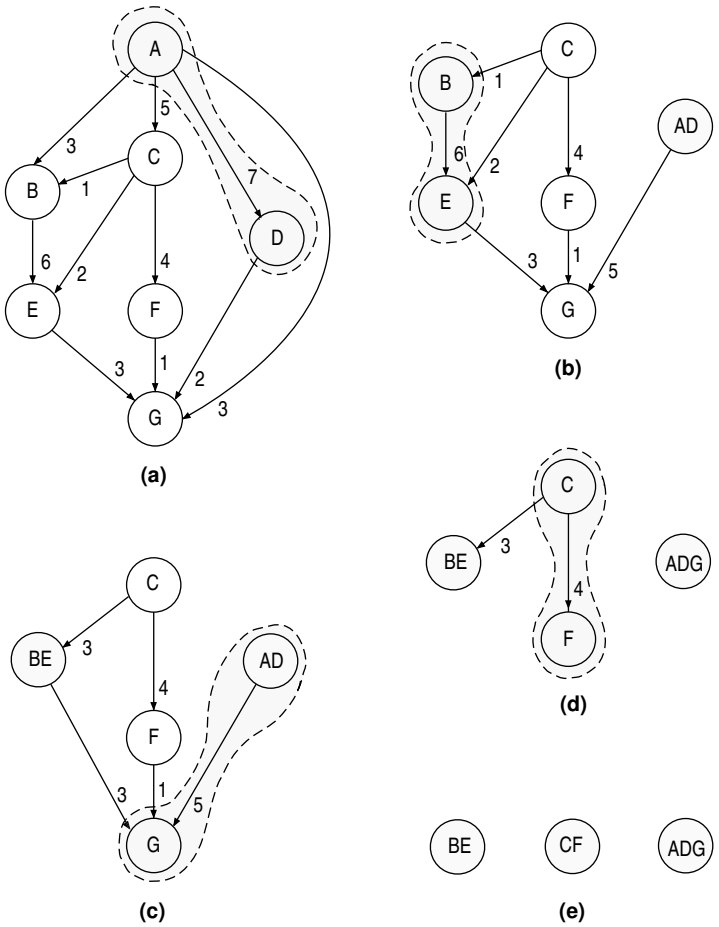


Figure 8: Constructing Chains for the Example Graph.

5.2 Coping with Architectural Constraints

One complication we have ignored so far has to do with the presence of architectural constraints specified as obligatory host queries. The chain construction algorithm of Figure 6 can be easily modified to accommodate architectural constraints.

Two nodes i and j of a computation graph are said to be *compatible* if $OH_i \cap OH_j \neq \emptyset$. In other words, the site contains at least one host that can execute both nodes. The parallelism matrix construction algorithm of Figure 6 is modified so as to force chains to contain only compatible nodes. We can accomplish this easily by pretending that any two nodes that are not compatible to be parallel. Encoding architectural constraints in this manner, the chain construction algorithm remains as before.

While this strategy is effective in building chains of compatible nodes, it can result in inopportune mappings in certain cases. The problem is illustrated in Figure 9. Suppose that the two nodes marked V have architectural constraints such that $OH = V$ for both of them while the remaining two nodes can be executed on any host of the system. Application of our algorithm results in the

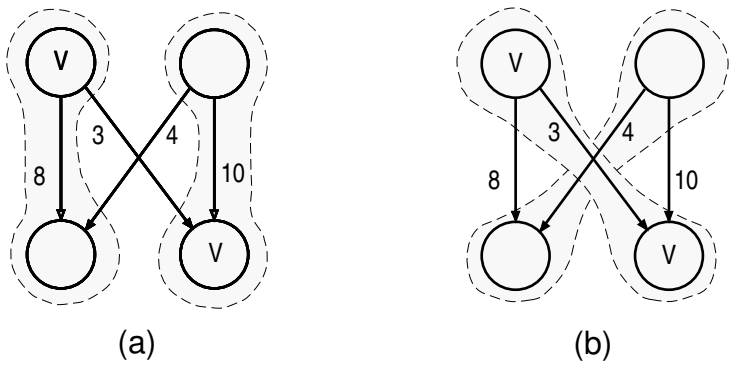


Figure 9: An Inopportune Mapping with Architectural Constraints.

two chains shown in Figure 9 (a). If $|V| \geq 2$, then we can achieve the maximal parallelism possible. If, however, $|V| = 1$, the entire graph is forced to execute on that host. In this case, a much better partitioning of the graph would have been as shown in Figure 9 (b) where the two nodes with the constrains run on the (single) host that can execute them while the remaining two nodes run on some other host. We are currently studying extensions to our algorithm that will incorporate these observations.

5.3 Mapping Chains to Hosts

Once the chains of the computation graph have been identified, we base all further mapping decisions on chains rather than nodes. It is for this reason that we required chains to contain only compatible nodes. We next describe our strategy for mapping these chains to hosts.

For some chain c, the set of hosts that can execute all nodes of the chain is given as

$$OH_c = \bigcap_{i \in c} OH_i$$

Since chains contain only compatible nodes, we know that OH_c is not empty. We associate a length with each chain that is equal to the number of nodes it contains. The optimal mapping of chains to processors requires the solution of yet another NP-complete problem—bin-packing [21]. Our algorithm for packing these chains to processors is again a heuristic: Let c be the longest unassigned chain of the computation. It is mapped to host $h \in OH_c$ such that h has the largest residual power. We define the *residual power* of host h as

$$\frac{Raw\ Power_h}{1 + Load_h}$$

where $Raw\ Power_h$ is the host h's SPECmark rating [30] and $Load_h$ is the sum of the lengths of the chains assigned to h up to now. In case of ties, we choose the host that is preferred by the greatest number of nodes in the chain. In other words, we choose an h such that $|\{i : h \in PH_i\}|$

is maximal. Before a chain is committed to a host, the loader makes sure that it is up and running by "pinging" the Paralex daemon that it is running.

SPECmark is a recently-introduced measure of system performance is derived as the geometric mean of CPU performances for 10 common applications including both integer and floating point computations. The SPECmark rating of a host is obtained from the the site definition file as the value of the attribute `specmarks`. At this point in the mapping, we choose to ignore any dynamic load information that may be available since the information may be stale by the time the nodes become active on the hosts.

5.4 Mapping Replicas

Recall that Paralex replicates nodes as ISIS process groups to render them fault tolerant. The discussion so far has been in the absence of fault tolerance requirements such that no replication is necessary. When some of the nodes of a chain have to replicated to satisfy their fault tolerance specification, the mapping decisions presented so far can be seen as selecting which of the replicas will be nominated the initial coordinator of its respective process group and actually carry out the computation. In this section we address the question of mapping the remaining replicas (cohorts) of the process group.

Clearly, each replica of a node has to be mapped to a separate host to achieve failure independence. The fault tolerance level of any node is thus limited by the number of hosts in its OH set. Even under the simplifying assumption that all of the nodes of a chain have uniform fault tolerance and architectural constraint specifications, the optimal mapping of chain replicas to hosts becomes computationally expensive.

In deciding where to map replicas, the consideration is maximizing load mobility. Informally, load mobility refers to the possibility of dynamically shifting load from one host to another by modifying the coordinator assignment for a processor group. The goal is to assign replicas of chains in such a way that the number of hosts that can be offloaded by reassigning coordinators is maximal. The problem can be formalized as a network flow problem and solved using the min-cut-max-flow theorem. For the time being, we chose to ignore this optimization potential and base out replica mapping on very simple rules.

Replicas of chain c are mapped to hosts in OH_c. In case $|OH_c|$ is greater than the number of replicas, the preferred host information is used to refine the host set. In case the number of replicas exceeds $|OH_c|$, the chain is broken and nodes are mapped individually on compatible hosts. Chains will be broken also if the fault tolerance requirement for their nodes are not uniform. In any case, even subsets of nodes that belong the same chain are treated as a unit and mapped together.

5.5 Shifting Load Dynamically

In selecting the initial host to map a chain, Paralex does not use any dynamic load information. This is based on the observation that the distributed system running Paralex programs is not dedicated to Paralex and there may be other load offered to the system by users. Thus, between the time that the program is loaded and the time that some chain actually starts execution, the load in the system could have changed substantially. Thus, we view the initial mapping of chains to hosts as nothing more than hints that will be re-evaluated during the course of execution.

The problem adaptive dynamic load balancing in a general-purpose distributed computing environment has been extensively studied [35,36,38]. What distinguishes our environment is the mixed nature of the load—that offered by a Paralex program and that offered by the normal users of the system. While we have good information regarding the characteristics of the Paralex, there is typically little information regarding the background load on the system. Thus, we concentrate on decisions to dynamically shift Paralex-generated load and make no attempt to balance the load due to the other uses of the system.

As with the mapping decisions, the unit of load to shift dynamically is a chain rather than a single node. Since we map chain replicas without breaking them in as much as possible, this strategy is feasible. The mechanism we use in Paralex for dynamically shifting load between hosts is the same one is used for fault tolerance—replication of nodes as process groups. By intervening in the ISIS coordinator selection mechanism, we are able to decide which host among those having replicas of a process group will actually perform the computation. The initial mapping at load time establishes a default coordinator for each process group. This mapping remains valid until Paralex modifies it by broadcasting a change to all members of the process group after having observed load changes. This "late binding" of coordinators to hosts achieves dynamic load balancing at little overhead beyond that already incurred for fault tolerance. Clearly, load can be dynamically shifted only between hosts that have replicas of a chain loaded.

The Paralex executor, after having loaded a program and started its execution, remains active on the host from which the program was launched and monitors the system state. We refer to this process as the Paralex *controller*. The monitored information includes both the computation state for debugging purposes and the load state of the distributed system for dynamic load balancing. As described in Section 3.6, each host capable of running Paralex nodes includes a daemon process launched at system initialization. These daemons observe the load on their local systems and periodically broadcast significant changes to an ISIS process group called `Paralex-Monitor`. Each instance of the Paralex controller (corresponding to multiple Paralex programs executing on the same distributed system) that whishes 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. If by the time this information is to be used there are hosts for which the controller has no load data, it can force the appropriate daemons to send the data by invoking a remote service.

The code wrapped around each node by Paralex includes ISIS primitives to send the controller an informative message just before it starts and as soon as it completes execution. By tracking the progress of the program, the Paralex controller uses the dynamic load information to re-assess the mapping decision for a chain just before a node is about to start execution. Note that the progress of the program is not dependent on the continued functioning of the controller. In case of a failure of the node where the controller is executing, the program proceeds with the default mapping that was established at load time, perhaps with degraded performance. Thus, the controller serves only to improve performance by revising the mapping decisions based on dynamic load. In this sense, it is not a fault tolerance bottleneck for the program. Since all communication between the program and the controller is asynchronous, it is also not a performance bottleneck.

The dynamic decision to select a host to execute a chain is the same as that used to initially map the chains—select the host with the largest residual power. The only difference is that the term *Load* in the definition of residual power now includes the actual load of the hosts plus the Paralex

load as measured in chains lengths. By including this second term, we avoid instability conditions that can result in so-called "sender-initiated" load balancing schemes [19]. If the above decision results in a change in the mapping for the chain, the controller performs a broadcast to the process groups corresponding to the remaining nodes of the chain announcing the new host. Members of a process group invoke the ISIS `coordinator-cohort` primitive to affect the decision. Use of causal broadcast primitives of ISIS guarantee consistent views of who the coordinator is among the group members [28].

Our dynamic load balancing strategy continues to treat chains as units. When a node of the chain is decided to be shifted from one host to another, all nodes that follow in the chain are shifted as well. Once a node starts execution at a host, it completes at that node. In other words, Paralex does not attempt to do process migration. The only exception to this is in case of a failure where ISIS starts a cohort to take over the computation that was being performed by the (failed) coordinator. We could realize process migration within the ISIS context but we judge the additional costs of state saving and transfer to be prohibitive with respect to the expected benefits.

5.6 Pipelined Execution

It is common to have the same computation graph operating on a sequence of input data. In contrast to the "single-shot operation," this so-called "pipelined operation" brings about new issues both for the computation semantics and for load balancing.

Since pipelined operation will result in multiple data values to traverse the links of the graph, there must be mechanisms that guarantee consistency of input arriving at the nodes. The computation nodes must not be executed with arguments corresponding to different iterations of the input values. General dataflow systems solve this problem by associating colors with the data tokens and enabling computations to fire only when there are tokens with matching colors. Fortunately, we need not resort to such complex schemes in Paralex even when graphs are used in pipeline mode. The use of causal broadcast for data communication is sufficient to guarantee FIFO delivery of messages along each link. Since our graphs are acyclic, this property alone is sufficient to guarantee the same semantics in both single-shot and pipelined modes. Paralex includes finite buffers at the node inputs to decouple computations between nodes of varying speed. Only when the input buffer is full must a producer node block waiting for the consumer node to free up a slot.

The consequences of pipelined operation for load balancing are more complex. While the single-shot operation requires sequential execution for the nodes of a chain, pipelined operation admits parallelism even within chains for different iterations. Thus, the maximal parallelism of a graph in pipelined mode can be as large as the number of nodes it contains. Our load balancing mechanisms must permit parallelism within a chain to the extent that the chain is replicated.

Our strategy is based on parallelizing chain executions with respect to iterations rather than nodes. In other words, a given iteration of the chain continues to execute sequentially (as it must) on a single host but multiple replicas of the chain may be active on different hosts working on different iterations. Since only one replica of a process group may be actively computing at any given time, we cannot have true parallelism between chain replicas. What is possible, however, is an alternation of the activation of the nodes in the chain between a set of hosts. This has the effect of distributing the load produced by a chain among the set of hosts where it is replicated.

6 The Paralex Debugger

The final component of the Paralex executor is a debugger. It shares the same interface as the graphics editor. It also interacts with the controller to obtain dynamic state information gathered during program execution. In this section we discuss some of the issues in designing a debugger for parallel programs and outline how they are resolved in Paralex.

6.1 A Framework for Debugging

Distributed debugging of parallel programs is in general a difficult problem, where the tools developed for sequential debugging often do not give satisfactory results [29,24]. The main reasons why the traditional sequential approach to debugging is not sufficient in a concurrent environment are the following:

Global Nondeterminism Behavior of a program depends not only on the values of its inputs but also on the interaction with other processes.

Probe Effect The debugger itself might interfere with the program and change its behavior [20].

When the concurrent execution takes place in a distributed system with potential failures, the debugging task becomes yet more difficult. Additional issues that need to be addressed include random communication delays, lack of instantaneous global state information [15], faulty processors and lost messages.

Our strategy in Paralex has been to select the programming abstractions and the automated support mechanisms in such a manner that most of the above concerns become non-issues. 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 the passage of data 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. 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 data representation, 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.

Given this black-box model of computation, we identify two distinct phases in developing a debugged Paralex application:

1. The code corresponding to each node is produced and debugged in isolation, using appropriate programming aids and debugging tools.

2. A Paralex program is written by organizing the nodes in a data-flow graph and the correctness of the global data interactions becomes subject of investigation.

As described in the next section, the role of the Paralex debugger is to support this second phase.

6.2 Debugging in Paralex

The debugging environment is fully integrated within Paralex—it shares the graphical interface with the development mode and uses the same graphical style of interaction. Almost no new skills are required by the programmer to use the debugger. The only evident change is in the content of the pop-up menus provided within the interface.

When a Paralex program is compiled with the debug option set, code generation for fault tolerance support and load balancing optimization are suppressed. Thus, the execution to be debugged will be non-fault tolerant and not necessarily best performing. As we have argued in the previous section, the correctness of the application is independent of these functions, thus the debugger need not 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*, *stopped* and *ended*. 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. Code fragments called *Trace filters* allow selective tracing of data as it flows though the links. They are defined using an interface similar to that of filter nodes.

Controlling the execution is accomplished through the following primitives:

Breakpoint Insert A breakpoint can be associated with one or more nodes. When the flow of execution reaches the set of nodes in the breakpoint list they become *stopped* allowing the user to interactively examine and modify the data appearing as input to the nodes.

Step Simulates a "single-step" execution of the program where the current stopped nodes are allowed to execute but no further.

Restart Restore the initial state and restart the computation from scratch.

Go Allows all of the enabled nodes to execute. Used to start the initial execution or to continue it after a breakpoint.

The implementation of the Paralex debugger requires no new mechanisms or abstractions. A debug process, invisible to the user, is created and executes on the machine with which the graphical interface is associated. Every node of the graph is modified so as to include a new input link from the debug process. Monitoring is accomplished by adding links from the nodes to the debug process and generating code for each node to capture and send the input data.

The introduction of the new links gives the debugger process complete control over the execution without modifying node semantics. Given its knowledge of the global state of the computation, the

debug process does not send enabling messages to those nodes that are in the breakpoint list and their descendents. The **Step** function is implemented by first setting a breakpoint for each of the children of the stopped node, and then enabling its execution.

7 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 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 that is present in other systems, we compare our to them.

7.1 Network of Workstations as Multiprocessors

The idea of viewing a collection of workstations on a network as a parallel multiprocessor appears to be a popular one. There are numerous projects that have been experimenting with different abstractions to provide on top of such a system.

The Amber System [16] developed at the University of Washington creates a shared memory multiprocessor abstraction over a network of Firefly multiprocessor workstations. The system is programmed using the shared objects paradigm with Amber handling all remote invocation and consistency issues.

The idea of broadcast-based parallel programming is explored in the `nigen++` system developed at Syracuse University [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 [34].

The master/slave model forms the basis of the AERO system from the University of California at Berkeley [3]. Unlike `nigen++`, however, the model is completely asynchronous with the master having to explicitly accept results from parallel slave computations.

7.2 Language and Architecture Heterogeneity

Our use of stubs for remote communication and universal data representation as ISIS messages derive their inspiration from Remote Procedure Call (RPC) systems [32,8]. 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 MLP [23] and HRPC [5].

7.3 Graphical Programming

Examples of systems that use graphical notations to express parallel computations include Fel [26], Poker [33], CODE [11], Alex [27] and apE [18]. None of these systems addresses fault tolerance nor provide 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 [12]. The programming paradigm of CODE is based on the model of Browne [10] where programs are represented as generalized dependency graphs with each node having three sets of dependencies: input data, output data and exclusion. apE is an interesting system that shares with Paralex the dataflow model of computation but is limited to scientific data visualization applications.

7.4 Architecture Independence

As with any other language proposal, Paralex strives for architecture independence. This is particular 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 [37] and Steele [34] fall into the first category. Systems and notations such as Paralex, Par [17], UNITY [14], Linda [22] and CODE 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.

8 Status and Conclusions

A prototype of Paralex is being developed on a network of m680x0, SPARC, PRISM, MIPS and Vax-architecture workstations running UNIX. The graphical interface is based on X-Windows using the Open Look Intrinsics Toolkit. As of this writing, the graphics editor and the compiler are functional. The executor exists as a manual command and has not been integrated into the environment. Debugging support and load balancing are currently being worked on.

The Paralex Project provides evidence that a distributed system can be viewed and programmed as if it were a uniform multiprocessor parallel computer. While many of the distributed system characteristics that distinguish it from a uniform multiprocessor can be made transparent automatically, others remain visible to the programmer. In particular, architectural heterogeneity and independent failures can be handled automatically but the low bandwidth of the communication interconnect has to be confronted by the programmer by selecting an appropriate level of granularity in decomposing the application.

In mapping parallel computations to distributed systems, Paralex adopts a very pragmatic and realistic approach—use only information that is structurally available and prefer simple, cheap

heuristics to complex, expensive optimization computations. How well these choices work in practice has to be verified experimentally. We will be using this system to experiment with various aspects of the mapping and load balancing algorithms.

The programming paradigm we have chosen plus the level of automatic support Paralex is able to provide to applications makes debugging simple and effective. The programmer need only concentrate on the logical structure of the application; the difficult parts of programming distributed parallel programs—heterogeneity, distribution and fault tolerance—are handled automatically by the system. Inclusion of the debugger in Paralex was accomplished within the same paradigm as application programs.

While Paralex has demonstrated the feasibility of providing automatic support for parallel programs in a distributed system, the performance benefits of the approach remain to be demonstrated. We are currently engaged in programming a variety of applications in Paralex to evaluate both its expressiveness and its efficiency. The applications include heat flow computations within a square slab, ray tracing and an image processing problem in the context of surface echo radars. Based on these applications, the costs of the various aspects of automatic support will be evaluated and used as the basis for a redesign.

References

[1] W. B. Ackerman. Data Flow Languages. *IEEE Computer,* February 1982, pp. 15–22.

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

[3] D. P. Anderson. The AERO Programmer's Manual. Technical Report, CS Division, EECS Department, University of California, Berkeley, October 1990.

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

[5] 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. D82, August 1987, pp. 880–894.

[6] K. Birman, R. Cooper, T. Joseph, K. Kane and F. Schmuck. The ISIS System Manual.

[7] K. Birman and K. Marzullo. ISIS and the META Project. *Sun Technology,* vol. 2, no. 3 (Summer 1989), pp. 90–104.

[8] 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.

[9] A. Bondavalli and L. Simoncini. Dataflow-Like Model for Robust Computations. *Computer Systems Science and Engineering,* vol. 4, no. 3 (July 1988), pp. 176–184.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States in a Distributed System. *ACM Trans. on Computer Systems*, vol. 3, no. 1, February 1985, pp. 63–75.

[16] 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. University of Washington, Department of Computer Science Technical Report 89-04-01, April 1989.

[17] 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.

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

[19] D. L. Eager, E. D. Lazowska and J. Zahorian. A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing. In *Proc. ACM SIGMETRICS Conf.*, Austin, Texas, August 1985, pp. 1–3.

[20] J. Gait. A Debugger for Concurrent Programs. *Softw. Pract. Exper.* vol. 15, no. 6, 1985, pp. 539–554.

[21] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[22] 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.

[23] R. Hayes, S. W. Manweiler and R. D. Schlichting. A Simple System for Constructing Distributed, Mixed-Language Programs. Technical Report TR 87-1b, Department of Computer Science, University of Arizona, Tucson, Arizona, February 1988.

[24] W. Hong Cheung, J. P. Black and E. Manning. A Framework for Distributed Debugging. *IEEE Software,* January 1990, pp. 106–115.

[25] C. E. Houstis. Module Allocation of Real-Time Applications to Distributed Systems. *IEEE Trans. Software Engineering*, vol. 16, no. 7, July 1990, pp. 699–709.

[26] 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.

[27] 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.

[28] 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.

[29] C. Mcdowell and D.P. Helmbold. Debugging Concurrent Programs. *ACM Computing Surveys,* vol. 21, no. 4, December 1989, pp. 593–623.

[30] R. E. Novak. SPEC, the Industry Standard for System Performance. MIPS Computer Systems technical report, 1990.

[31] A. Shaw and K. Jeffay. Software Engineering of Real-Time Operating Systems. University of Washington, Department of Computer Science Technical Report 88-01-01, January 1988.

[32] S. K. Shrivastava and F. Panzieri. The Design of a Reliable Remote Procedure Call Mechanism. *IEEE Trans. Computers*, vol. C-31, no. 7, pp. 692–697.

[33] L. Snyder. Parallel Programming and the Poker Programming Environment. *IEEE Computer,* July 1984, pp. 27–36.

[34] 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.

[35] M. Stumm. The Design and Implementation of a Decentralized Scheduling Facility for a Workstation Cluster. In *Proc. 2nd IEEE Conf. Computer Workstations*, IEEE, March 1988, pp. 12–22.

[36] M. M. Theimer and K. A. Lantz. Finding Idle Machines in a Workstation-Based Distributed System. *IEEE Trans. Software Engineering*, vol. 15, no. 11, November 1989, pp. 1444–1458.

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

[38] S. Zhou and D. Ferrari. An Experimental Study of Load Balancing Performance. Techincal Report UCB/CSD 87/336, Computer Science Division, University of California, Berkeley, California, January 1987.