Run-time Support for Dynamic Load Balancing and Debugging in Paralex

Özalp Babaoğlu Lorenzo Alvisi Alessandro Amoroso
Renzo Davoli Luigi Alberto Giachini

Technical Report UBLCS-92-3

September 1992

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 via anonymous FTP from the area `ftp.cs.unibo.it:/pub/TR/UBLCS` in compressed PostScript format. Abstracts are available from the same host in the directory `/pub/TR/ABSTRACTS` in plain text format. All local authors can be reached via e-mail at the address *last-name*`@cs.unibo.it`.

UBLCS Technical Report Series

92-1 *Mapping Parallel Computations onto Distributed Systems in Paralex*, by Ö. Babaoğlu, L. Alvisi, A. Amoroso and R. Davoli, January 1992.

92-2 *Parallel Scientific Computing in Distributed Systems: The Paralex Approach*, by L. Alvisi, A. Amoroso, Ö. Babaoğlu, A. Baronio, R. Davoli and L. A. Giachini, February 1992.

92-3 *Run-time Support for Dynamic Load Balancing and Debugging in Paralex*, by Ö. Babaoğlu, L. Alvisi, S. Amoroso, R. Davoli, L. A. Giachini, September 1992.

92-4 *Paralex: An Environment for Parallel Programming in Distributed Systems*, by Ö. Babaoğlu, L. Alvisi, S. Amoroso, R. Davoli, L. A. Giachini, October 1992.

93-1 *Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanism*, by Ö. Babaoğlu and K. Marzullo, January 1993.

93-2 *Understanding Non-Blocking Atomic Commitment*, by Ö. Babaoğlu and S. Toueg, January 1993.

Run-time Support for Dynamic Load Balancing and Debugging in Paralex[1]

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

Technical Report UBLCS-92-3

September 1992

Abstract

Paralex is a programming environment for developing and executing parallel applications in distributed systems. The user is spared complexities of distributed programming including remote execution, data representation, communication, synchronization and fault tolerance as they are handled automatically by the system. Once an application starts execution in a distributed system, it may be interacted with at two levels: by Paralex itself to achieve automatic fault tolerance and dynamic load balancing; or by the user in association with performance monitoring and debugging. In this paper, we describe the set of monitors and control mechanisms that constitute the Paralex run-time system and their use for implementing dynamic load balancing and debugging.

---

1. Partial support for this work was provided by the Commission of European Communities under ESPRIT Programme Basic Research Action Number 3092 (Predictably Dependable Computing Systems), the United States Office of Naval Research under contract N00014-91-J-1219, IBM Corporation 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

Distributed systems comprising powerful workstations and high-speed communication networks represent valuable computational resources. The amount of raw computing power that is present in a typical modern distributed system may be comparable to an expensive, special-purpose super computer. Thus, it is tempting to try to harness the massive parallelism available in these systems for single, compute-intensive applications.

Distributed systems differ from special-purpose parallel computers in that they (i) exhibit total asynchrony with respect to computation and communication, (ii) communicate over relatively low-bandwidth, high-latency networks, (iii) lack architectural and linguistic homogeneity, (iv) exhibit increased probability of communication and processor failures, and (v) fall under multiple administrative domains. As a consequence, developing parallel programs in such systems requires expertise in fault-tolerant distributed computing. 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 clear that effective use of distributed systems for parallel computing requires automated support in handling these complexities.

A number of projects are under way to harness the power of distributed systems and support network-based parallel computing [2,3,8,9,14,12]. The Paralex programming environment is an instance of such a system. Paralex takes graphical descriptions of parallel computations and automatically produces fault-tolerant distributed programs to carry them out. The system is a complete programming environment in that it includes a graphics editor, compiler, executor, run-time support and debugger. The intended application domain is scientific computing since it is best suited to exploit the potential parallelism of distributed systems.

In this paper we concentrate on the run-time system of Paralex in support of dynamic load balancing and debugging. We view both objectives as instances of interacting/controlling a distributed program. In the case of dynamic load balancing, the control is realized by the system based on the global network state, while the debugging interaction is realized by the user based on the state of the computation. A collection of sensors in the form of daemon processes and stubs inserted in the application perform the state monitoring function. The mechanism to exercise control for load balancing relies on passive replication originally introduced for fault tolerance. The control mechanism for debugging, on the other hand, exploits the semantics of the Paralex computational model to implement simple-yet-effective primitives.

The rest of the paper is organized as follows. The next section gives an overview of the Paralex programming paradigm and the environment to support it. In Section 3 we briefly outline the fault tolerance scheme of Paralex as it relates to load balancing. Both the decision procedures and the mechanisms to realize dynamic load balancing are described in Section 4. In Section 5 we discuss the general considerations in designing the Paralex debugger whose details are described in Section 6. Section 7 presents the status of the system and concludes the paper.

2 Overview of Paralex

Paralex is a programming environment for developing parallel applications and executing them on a distributed system, typically a network of workstations. Programs are specified in a graphical notation and Paralex automatically handles distribution, communication, data representation, architectural heterogeneity and fault tolerance. It consists of the following logical components: A graphics editor for program specification, a compiler, an executor, a debugger and a run-time support environment. These components are integrated within a single programming environment that makes extensive use of graphics. Here we give a brief overview of Paralex. Details can be found in [5].

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 the 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 code fragments to indicate the computations. The current prototype limits the structure of the data flow graph to be acyclic.

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 at a node contain values, the computation associated with the node starts execution transforming the input data to output. The computation to be performed by the node must satisfy the "functional" paradigm — mapping input values to output values without any side effects. The actual specification of the computation may be done using whatever appropriate notation is available including conventional sequential programming languages, executable binary code or library functions for the relevant architectures.

Unlike classical data flow, the nodes of a Paralex program carry out significant computations. This so-called *large-grain* data flow model [7] 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.

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. In Paralex, this is accomplished simply by drawing multiple links originating from a node towards the various destinations. To economize on 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.

Note that 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

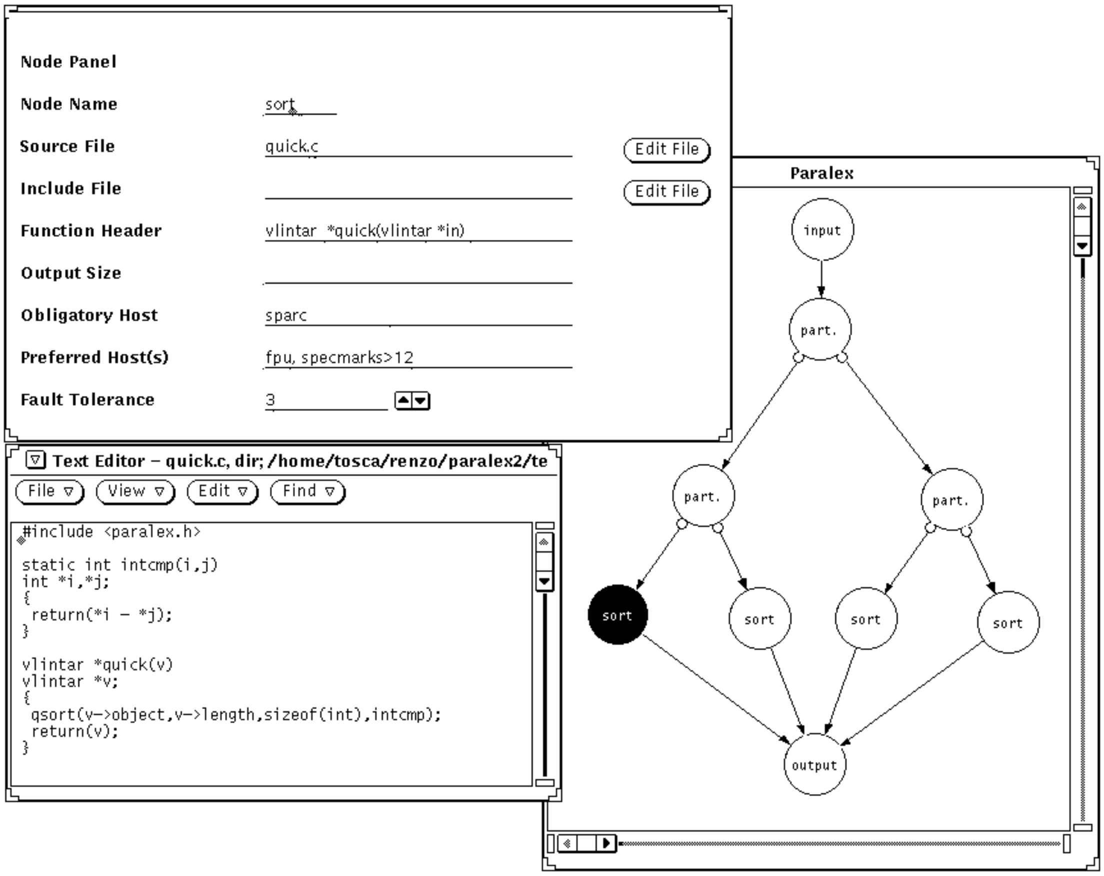

Figure 1. The Paralex Graphics Editor and Node Property Panel.

for distributed parallel programs. We also note that this programming paradigm is in the spirit of that proposed by Steele [18] where a severely-restricted programming paradigm is advocated even for environments that support arbitrary asynchrony.

Paralex programs are composed through an X-windows based graphics editor using nodes, filters and links as building blocks. Computations for the nodes and filters are specified through property panels. Figure 1 illustrates a Paralex program for parallel sorting along with the property panel and code for the highlighted node. Note that the code defining the computation is completely ordinary sequential C and contains nothing having to do with remote communication or fault tolerance. Filters are defined in a completely analogous manner and are visualized as small circles attached to the parent node. Property panels serve to define the name of the node, name of the file containing the code, an abstract interface for the node specified as a function header using ANSI C syntax and the fault tolerance level of the node. The distributed system on which Paralex computations may execute is described through a *site* file consisting of host and their attributes. The node property panel includes fields that may contain queries naming the attributes of the site file so as to constrain the set of hosts where the node may execute.

The program can be compiled once it is fully specified through the data flow graph and the computations to be carried out by the nodes. The first pass of the Paralex compiler is 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. Paralex checks type consistency across node boundaries and automatically generates all of the necessary code to do data encoding/decoding using a common data representation across different architectures. 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.

The compiler must address two aspects of heterogeneity in a distributed system: data representation and instruction sets. Paralex uses the ISIS toolkit [10,11] 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. Heterogeneity with respect to instruction sets is handled by invoking remote compilations on the machines of interest and storing multiple executables for the nodes.

3 Fault Tolerance

As part of 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 invoked, it will be executed on $k+1$ distinct hosts such that the computation will succeed despite up to k failures. 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 the network. This is consistent with the objectives of Paralex — parallel computing in a distributed system — where genuine hardware failures are probably very rare but events such as workstations being turned on and off, rebooted or disconnected from the network are much more likely.

Paralex uses passive replication as the basic fault tolerance technique. Given the application domain (parallel scientific computing) and the hardware platform (networks of workstations), Paralex has to favor efficient use of computational resources over speedy recovery times in its choice of a fault tolerance mechanism. 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 the next section.

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 remaining members are *cohorts* and remain inactive other than receiving broadcasts 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 broadcast from the coordinator at 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 input queues associated with the link. When the coordinator completes computing, it broadcasts the results to the process groups at next level and signals the cohorts (through another intra-group broadcast) 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.

4 Balancing Load Dynamically

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 [4]. 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. The instantaneous measurements are filtered to remove high frequency components and extract long-term trends by exponential smoothing. If two consecutive load measures differ by more than some threshold, the daemon process broadcasts 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 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. 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 evaluates the most recent network load measures to decide if the current coordinator choice is still valid. If not, the controller broadcasts 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 need only inform the head of a chain.

The actual decision to modify a coordinator choice is made by the controller based on a "residual power" metric. For each host that contains a replica of the chain, the controller computes the ratio of the host's raw power (taken as its SPECmark rating) and the actual load of the host plus one. The preferred host for executing the chain is taken as the host with the largest residual power metric. 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 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 2 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

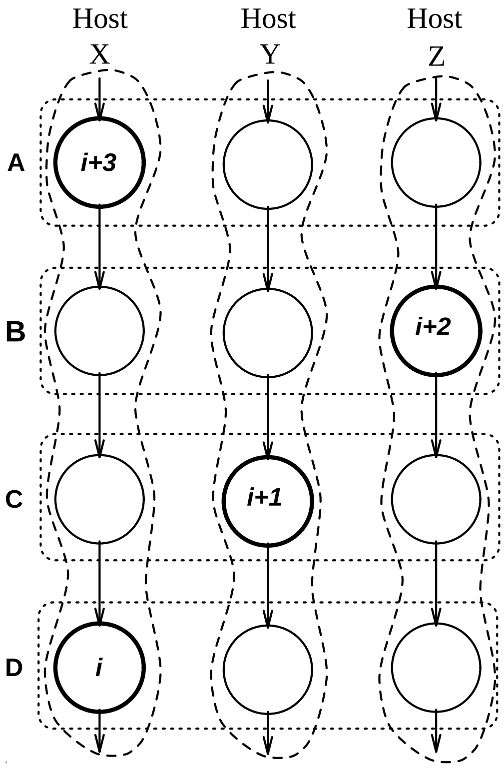

Figure 2. Pipelined Execution of a Replicated Chain.

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.

5 Distributed Debugging

Debugging concurrent programs is in general a difficult problem, where tools developed for sequential debugging often do not give satisfactory results [17][15]. The main reasons for this 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 [13].

When concurrent execution takes place in a distributed system, the debugging task becomes even more difficult. Additional issues that need to be addressed include random communication delays, lack of instantaneous global state information [16], processor failures 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 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.

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 sequential 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 Debugging in Paralex

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 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* 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.

Controlling program execution is accomplished through the following debugger primitives:

Breakpoint Insert A breakpoint can be associated with one or more nodes. When the execution for a particular iteration 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 a "step" is interpreted as the set of stopped nodes. For each step requested, the stopped nodes are allowed to execute but automatic breakpoints are inserted at their immediate successors.

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.

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 a Paralex program is compiled with the debugging option set, each node 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. We refer to these valueless messages sent by the controller as *enabling tokens*. Using this simple mechanism, we are able to exercise complete control over the distributed computation from the debugging host.

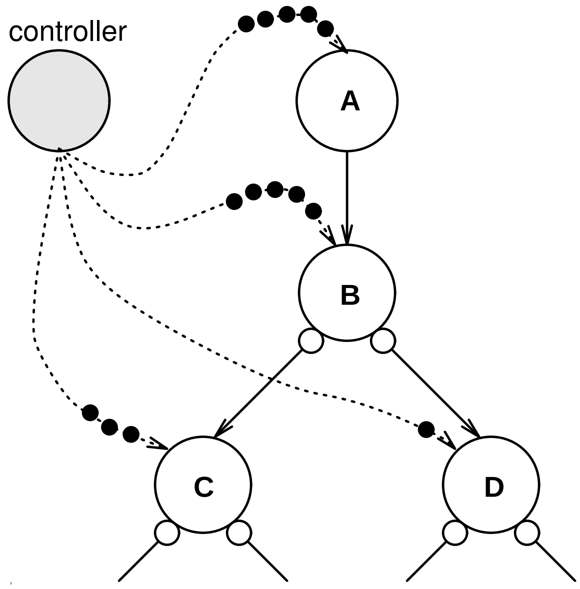

Figure 3. The Original Program Graph and its Modification for Debugging.

Figure 3 illustrates a program graph fragment along with the new links from the controller acting as the fictitious node. The black circles along the links indicate the enabling tokens. The example depicted corresponds to the state where the user has specified the breakpoint $\{(C, 4), (D, 2)\}$ indicating that the computation is to be stopped just before node C starts iteration 4 and node D starts iteration 2. The actual number of tokens sent to nodes A and B is irrelevant as long as it exceeds 3. Nodes that are common descendants of C and D will receive only one token. Otherwise, nodes will receive the same number of tokens as their ancestor in the breakpoint list. It may not be possible to set breakpoints at arbitrarily different iterations due to buffer limitations for links at common ancestors. For example, the situation depicted in Figure 3 would have two data values (corresponding to iterations 2 and 3) being buffered along the link from B to D. The **Step** function is implemented by sending one additional token to each of the stopped nodes.

7 Status and Conclusions

A prototype of Paralex has been developed running on a network of Unix workstations based on M680x0, Sparc, Mips and Vax architectures. The prototype includes the graphics editor, compiler, executor and the associated run-time support. The user interface is based on the Open Look intrinsics. As of this writing, the dynamic load balancing and debugging support have been implemented as described in this paper but are not integrated into the programming environment.

We have used the prototype to program parallel solutions to several important problems drawn from signal processing and thermodynamics application domains. This experience confirms our thesis that significant parallel programs not only can be developed rapidly by non-experts, but can also achieve acceptable performance with respect to speedup [6]. After all, the automatic support Paralex provides for rendering parallel programming painless would be of little value if the resulting program failed to match performances possible by hand coding. We feel that a slight loss in performance due an increased level of abstraction is completely acceptable considering the conceptual economy it provides to the programmer. The situation is not unlike that of sequential programming at a high-level language supported by a compiler as opposed to programming at the machine language level.

The effectiveness of the dynamic load balancing and debugging schemes we have described remain to be verified. Once they are integrated into the rest of the system, we will conduct experiments using the applications already developed. Dynamic load balancing requires careful evaluation since the external load presented to the system must be both repeatable and realistic. Evaluating the debugger is even more difficult since many of the criteria are subjective. We feel, however, that the programming paradigm plus the level of automatic support Paralex is able to provide to applications make debugging simple and effective. The programmer need only concentrate on the logical structure of the application and ignore the difficult parts of programming distributed parallel programs — heterogeneity, distribution and fault tolerance — since they are handled automatically by the system.

Acknowledgements We are grateful to Ken Birman and Keshav Pingali of Cornell University for discussing with us the various aspects of Paralex design and implementation. Alberto Baronio, Marco Grossi, Susanna Lambertini, Manuela Prati, Alessandro Predieri and Nicola Samoggia have contributed to the prototype as members of the Paralex group at Bologna.

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

[5] Ö. Babaoğlu, L. Alvisi, S. Amoroso, R. Davoli and L. A. Giachini. Paralex: An Environment for Parallel Programming in Distributed Systems. In *Proceedings of the 6th ACM International Conference on Supercomputing*, Washington, D.C., July 1992, 178–187.

[6] Ö. Babaoğlu, L. Alvisi, A. Amoroso, A. Baronio, R. Davoli and L. A. Giachini. Parallel Scientific Computing in Distributed Systems: The Paralex Approach. In *Proceedings of Sixth International Symposium on Computer and Information Sciences*, Side, Antalya, Turkey, October 1991, 1093–1103.

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

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

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

[11] 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.

[12] 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. ACM Symposium on Operating Systems Principles*, Litchfield Park, Arizona, December 1989, pp. 147–158.

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

[14] A. S. Grimshaw. An Introduction to Parallel Object-Oriented Programming with Mentat. Technical Report No. TR-91-07, Department of Computer Science, University of Virginia, April 1991.

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

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

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

[18] 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.