Mapping Parallel Computations on to Distributed Systems in Paralex

Özalp Babaoğlu **Lorenzo Alvisi** **Alessandro Amoroso**
Renzo Davoli

Technical Report UBLCS-92-1

January 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.

Mapping Parallel Computations on to Distributed Systems in Paralex[1]

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

Technical Report UBLCS-92-1

January 1992

Abstract

Paralex is a programming environment that allows parallel programs to be developed and executed on distributed systems as if the latter were uniform parallel multiprocessor computers. Architectural heterogeneity, remote communication and failures are rendered transparent to the programmer through automatic system support. Parallel programs are specified using a graphical language that is based on coarse-grain dataflow and may include existing sequential code as components. The Paralex loader selects hosts of a distributed system to execute a parallel program so as to satisfy the heterogeneity and fault tolerance requirements while trying to maximize performance. In this paper we address the problems of initial mapping and dynamic alteration of the association between parallel computation components and distributed system hosts. Our results include novel heuristics and mechanisms to resolve these problems despite the complexities introduced by architectural heterogeneity fault tolerance.

1. Appeared as Invited Paper in *Proceedings of IEEE CompEuro '91*, Bologna, Italy, May 1991, pp. 123–130. 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) 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

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. There have been numerous other proposals for networks of workstations as a "poor man's supercomputer" [20,10,8,4,3]. What distinguishes Paralex from these systems is the automatic support for fault tolerance and the level to which the various phases of the program development cycle have been integrated into a uniform environment.

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" [9].

Paralex distributes computations among the hosts of a distributed system for efficient execution. The decision procedures that have to be solved in this context are known as the *mapping problem*. The problem of mapping a Paralex computation structure on to a distributed system prior to execution has aspects in common with *task assignment* [14], *module assignment* [15,17] and the *file placement problems* [12,23]. Once a computation is underway, Paralex permits the initial mapping to be altered based on dynamic load information. This aspect of the mapping problem has strong similarities to adaptive dynamic load balancing, which has been extensively studied in general-purpose distributed computing environments [2,13,18,21,22,24]. In this paper, we describe the heuristics and mechanisms that have been developed to resolve the mapping problem in the presence of fault tolerance and architectural heterogeneity constraints of Paralex computations. Whenever possible, we have opted for efficiency and simplicity over optimality. Novelty of our results include uniform mechanisms for both fault tolerance and dynamic load balancing support.

The rest of the paper is organized as follows. The next section is a brief overview of Paralex including the computational model and the principal components of the user interface. Section 3 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 4 discusses the problem of mapping Paralex computations on to hosts in a distributed system. Both the static and the dynamic aspects of the problem are covered and the algorithms illustrated through an example. Section 5 gives a status report of the current prototype and concludes the paper.

2 Overview of Paralex

Paralex consists of four logical components: A graphics editor for program specification, 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. All of the components consult a site definition file as described in the next section.

2.1 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, specmarks=21
leporello   sparc, fpu, specmarks=13
dongiovanni sparc, fpu, graphics, color, 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
```

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.

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.

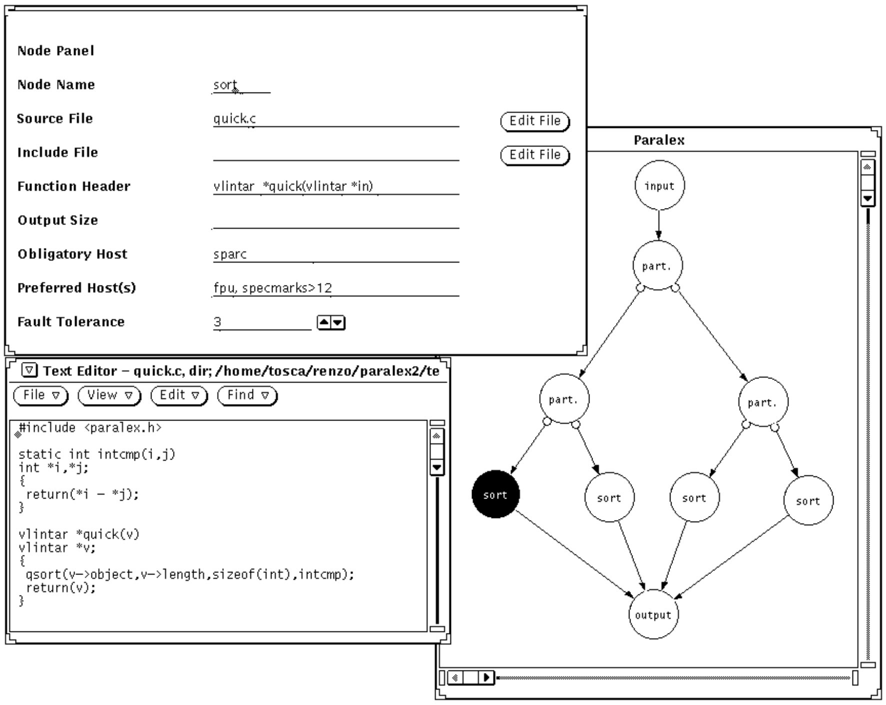


Figure 2. The Program Quick Sort and the Node Panel.

2.2 Specifying Paralex Programs

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 thought of as directed graphs (and indeed are visualized as such on the screen) representing the data flow relations plus a collection of ordinary sequential code fragments to indicate the computations. 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 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.

Unlike classical data flow, the nodes of a Paralex program carry out significant computations. This so-called *coarse-grain* data flow model [5] 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 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.

A graphics editor permits nodes, filters and the data flow relations of a program to be drawn using a point-and-click-style interaction and pop-up menus. Computations for the nodes and filters are specified through property panels. Figure 2 illustrates a Paralex program inspired by quick sort 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 absolutely nothing having to do with remote communication or fault tolerance. Filters are defined in a completely analogous manner.

Of the various fields in a node property panel, those relevant to the mapping problem are the following:

Obligatory Host A set of attributes that must be satisfied by a host to execute the node. Defines a class of hosts that can execute the node using a simple query language. 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 earlier. 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
```

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.

2.3 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 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.

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. 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.

2.4 Executing Paralex Programs

The Paralex executor consists of a loader, controller and debugger. 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 4. 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 4.5.

3 Fault Tolerance

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.

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 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. 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.

4 Mapping Computations to Hosts

We return to the question of loading Paralex programs and examine the algorithms employed to map the computation graph on to 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 and heterogeneity 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 [15]. As such, it is computationally intractable and known to be NP-complete [16]. Our approach in Paralex to this problem is much more pragmatic—we want fast, simple heuristics to map computations on to 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 4.2.

4.1 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.

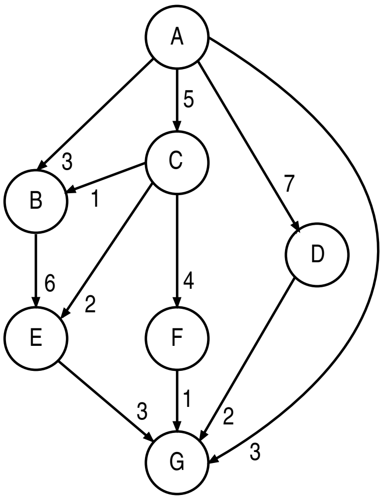
To this end, we annotate the computation graph by including the number of bytes flowing through each link as its weight. This information is extracted from the type definitions associated with the node interfaces. In case of variable-length data types such as arrays, the weight of the link is set to 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 [16], 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

procedure ParallelismMatrix(A, **var** P)
begin
  forall indexes i, j **do**
  begin
    $P[i, j] := 0$
    if $A[i, j] \neq 0$ **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** $\mathcal{C}$)
begin
  $\mathcal{C} = \{\{1\}, \{2\}, \{3\}, \ldots, \{n\}\}$
  while ($P[i, j] = 1$ for some i, j) **do**
  begin
    find $(\mathcal{K}, \mathcal{L})$ such that $A[\mathcal{K}, \mathcal{L}] = max\{A[\mathcal{U}, \mathcal{V}] : \ P[\mathcal{U}, \mathcal{V}] = 1\}$
    $T := \mathcal{K} \cup \mathcal{L}$
    $\mathcal{C} := \mathcal{C} - \{\mathcal{K}\} - \{\mathcal{L}\}$
    forall $s \in \mathcal{C}$ **do**
    begin
      $A[T, s] := A[\mathcal{K}, s] + A[\mathcal{L}, s]$
      $A[s, T] := A[s, \mathcal{K}] + A[s, \mathcal{L}]$
      $P[T, s] := min\{P[\mathcal{K}, s], P[\mathcal{L}, s]\}$
      $P[s, T] := min\{P[s, \mathcal{K}], P[s, \mathcal{L}]\}$
    end
    $A[T, T] := 0$
    $P[T, T] := 0$
    $\mathcal{C} := \mathcal{C} \cup \{T\}$
  end
end

Figure 3. Algorithms to Construct the Parallelism Matrix and Chains.



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

Parallelism 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 4. A Computation Graph and its Adjacency and Parallelism Matrices.

sequence of links that visits all nodes of the subgraph. 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 [17].

The algorithm starts with a matrix representation of the computation graph. Given a computation graph, 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:

$P[i, j] = 1$ 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 3. 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 3 returns in $\mathcal{C}$ 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.

The algorithm is best understood through an example. Figure 4 shows a seven-node computation graph along with its weighted-adjacency matrix representation and its parallelism matrix. The application of the chain construction algorithm of Figure 3 is illustrated in Figure 5.

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

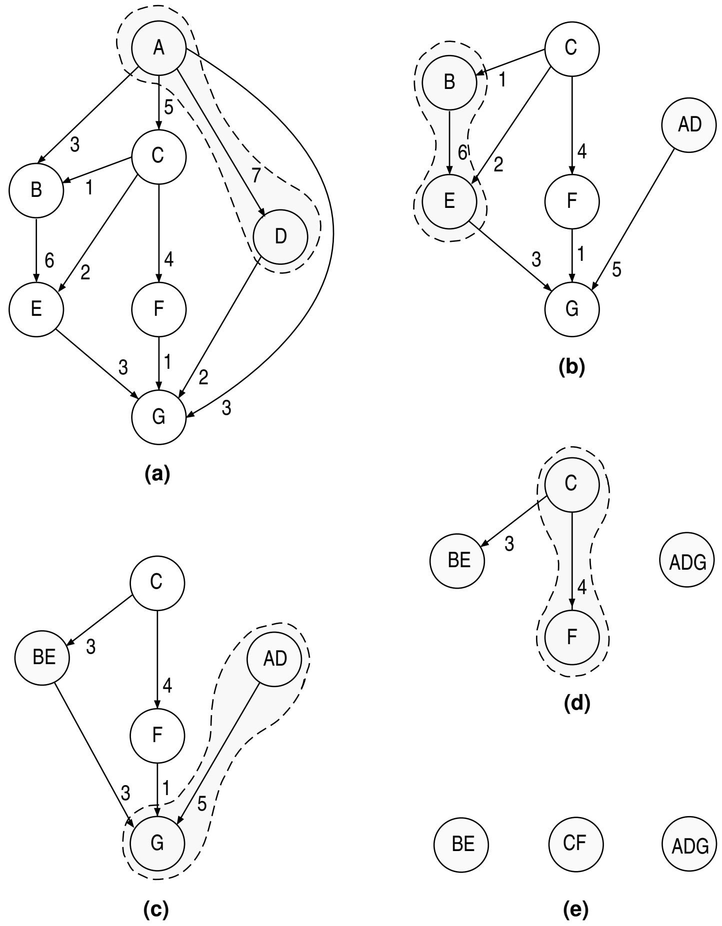


Figure 5. Constructing Chains for the Example Graph.

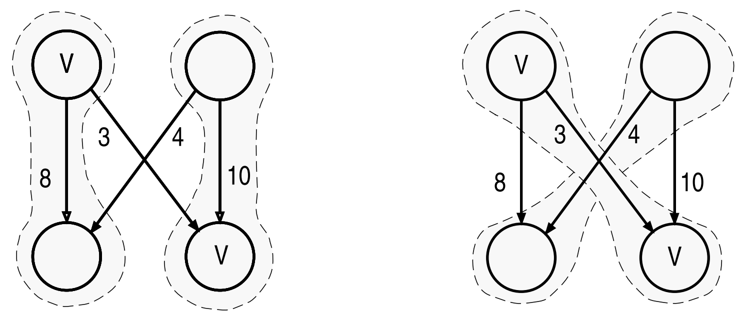


Figure 6. Inopportune Mapping with Architectural Constraints.

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 5 (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 5 (e).

4.2 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 3 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 3 is modified so as to force chains to contain only compatible nodes. We can accomplish this easily by pretending that any two incompatible nodes are in parallel. Encoding architectural constraints in this manner, the chain construction algorithm remains unchanged.

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 6. 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 two chains shown in Figure 6 (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 6 (b) where the two nodes with the constraints 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.

4.3 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 [16]. 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 [11] 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 the host is up and running by "pinging" the Paralex daemon on it.

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.

4.4 Mapping Replicas

Recall that Paralex replicates nodes as ISIS process group members 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 to the same chain are treated as a unit and mapped together.

4.5 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 on 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 of adaptive dynamic load balancing in a general-purpose distributed computing environment has been extensively studied [2,13,18,21,22,24]. 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 Paralex programs, 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 used for fault tolerance—replication of nodes as process group members. 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 2.4, 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 at 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 chain lengths. By including this second term, we avoid instability conditions that can result in so-called "sender-initiated" load balancing schemes [13]. 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 `coord_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 [19].

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 it 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.

4.6 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.

5 Conclusions

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 over complex, expensive optimization computations. We have described algorithms and mechanisms to achieve these goals. The time complexity of our mapping strategy is dominated by the transitive closure algorithm necessary to compute the parallelism matrix. Given that the operation needs to be invoked only once and that its complexity is $O(n^3)$ for a computation graph of n nodes, we consider the cost acceptable. What is particularly important is the efficiency of our dynamic load balancing decision procedures. Since they are linear in time complexity and are executed by the controller asynchronously with respect to the nodes, there is no time penalty implied for the computation.

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 including heat flow computations within a square slab, ray tracing and image construction from surface echo radars signals using a prototype of Paralex. We are interested in evaluating both the expressiveness of Paralex as a language and its performance as a system. In particular, how well our simple choices for the mapping algorithms work in practice will be verified experimentally.

References

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

[2] R. Alonso and L. Cova. Sharing Jobs Among Independently-Owned Processors. In *Proc. IEEE 8th Int. Conf. on Distributed Computing Systems*, 1988, pp. 282–288.

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

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

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

[6] K. Birman, R. Cooper, T. Joseph, K. Kane and F. Schmuck. The ISIS System Manual. Department of Computer Science, Cornell University, Ithaca, New York.

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

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

[9] 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.

[10] 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.

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

[12] L. Dowdy and D. Foster. Comparative Models of the File Assignment Problem. *ACM Computing Surveys*, vol. 14, June 1982, pp. 287–314.

[13] 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.

[14] K. Efe. Heuristic Models of Task Assignment Scheduling in Distributed Systems. *IEEE Computer*, vol. 15, no. 6, June 1982, pp. 50–56.

[15] D. Fernández-Baca. Allocating Modules to Processors in a Distributed System. *IEEE Trans. Software Engineering*, vol. 15, no. 11, November 1989, pp. 1427–1436.

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

[17] 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.

[18] K. Hwang, W. J. Croft, G. H. Goble, B. W. Wah, F. A. Briggs, W. R. Simmons and C. L. Coates. A Unix-Based Local Computer Network with Load Balancing. *IEEE Computer*, vol. 15, no. 4, April 1982, pp. 55–66.

[19] 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.

[20] A. Singh, J. Schaeffer and M. Green. A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstations. *IEEE Trans. on Parallel and Distributed Systems*, vol. 2, no. 1, January 1991, pp. 52–67.

[21] 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.

[22] 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.

[23] B. Wah. File Placement in Distributed Computer Systems. *IEEE Computer*, vol. 17, no. 1, January 1984, pp. 23–33.

[24] S. Zhou. A Trace-Driven Simulation Study of Dynamic Load Balancing. *IEEE Trans. on Software Engineering*, vol. 14, no. 9, September 1988.