Improving Performance of Virtual Reality Applications Through Parallel Processing

LEONIDAS DELIGIANNIDIS ldeligia@cs.uga.edu
Department of Computer Science, University of Georgia, Athens, GA 30602, USA

ROBERT J.K. JACOB jacob@cs.tufts.edu
Department of Computer Science, Tufts University, Medford, MA 02155, USA

Abstract. DLoVe (Distributed Links over Variables evaluation) is a new model for specifying and implementing virtual reality and other next-generation or "non-WIMP" user interfaces. Our approach matches the parallel and continuous structure of these interfaces by combining a data-flow or constraint-like component with an event-based component for discrete interactions. Moreover, because the underlying constraint graph naturally lends itself to parallel computation, DLoVe provides for the constraint graph to be partitioned and executed in parallel across several machines, for improved performance. With our system, one can write a program designed for a single machine but can execute it in a distributed environment with minor code modifications. The system also supports mechanics for implementing or transforming single user programs into multi-user programs. We present experiments demonstrating how DLoVe improves performance by dramatically increasing the validity of the rendered frames. We also present performance measures to measure statistical skew in the frames, which we believe is more suitable for interactive systems than traditional measures of parallel systems, such as throughput or frame rate, because they fail to capture the freshness of each rendered frame.

Keywords: Parallel Processing, Distributed Virtual Reality, Performance Evaluation, Constraints, Multi-User Interfaces

1. Introduction

The state of practice in user-interfaces today is the familiar direct manipulation, GUI (Graphical User Interface), or WIMP (Window, Icon, Menu, Pointer) style interface [3]. The next generation's computer interfaces have been called non-WIMP [22] and are characterized by parallel and continuous interactions and some examples include virtual reality and virtual environments [14]. Existing languages and models, event-based software, methods and tools do not satisfy the next generation requirements. Most of today's examples of non-WIMP interfaces have been designed and implemented with event-based models that are more suited to previous interface styles. Because there is no software that can describe continuous, parallel interaction explicitly (which is needed for virtual reality) and the old models (event-driven) fail to capture continuous, parallel interaction explicitly, new interfaces have required considerable low-level programming. While some of these interfaces are very inventive, they have made such systems difficult to develop, reuse, and maintain.

DLoVe (Distributed Links over Variables evaluation) is a new model for next generation dialogues. This model expresses non-WIMP formal structure in the same way that existing technology expresses command-based, textual and event-based dialogues. This model combines a data-flow or constraint-like component for the continuous relationships with an event-based component for discrete interactions. The modules describing the continuous relationships between objects form a constraint graph, which can be evaluated by DLoVe's constraint engine. The constraint engine ensures that all relationships between the related objects are satisfied. Individual continuous relationships can be enabled or disabled on the fly. Programs designed in this framework, not only capture continuous and parallel interaction explicitly, but also allow programs to be executed in parallel when there is a need for faster computation [28].

In DLoVe, one can write programs designed for a single machine but can execute them in a distributed environment with minor code modifications. The majority of research done in parallel systems has concentrated either upon computational models, parallel algorithms, or machine architectures. By contrast, little attention has been given to software development environments or program construction techniques required in order to translate algorithms into operational programs [38].

For implementing this framework, DLoVe assigns roles to different machines. For example, one machine, the Coordinator, is responsible for rendering the graphics on the screen and reading all input devices, and all other machines, the Workers, are responsible for keeping the constraints between Variables up-to-date and serving requests to the Coordinator. The Workers work in parallel on different parts of the constraint to bring the entire constraint graph up-to-date. As a result, the Coordinator has more time to spend refreshing the screen and providing an immersive environment to the user, because the responsibility of keeping all the Variables up-to-date is taken away from it and given to other machines, the Workers, which are dedicated to this task. All machines/processes involved in executing a program in parallel have an identical copy of the constraint-graph so that every machine/process works on the same constraint graph. However, each machine/process is only responsible for evaluating part of the constraint graph.

Often when you have a more expressive language, it costs you something in performance. But in our case, the better language improves performance because of the parallelization techniques utilized.

2. Parallel and distributed systems

The need for more and more computing speed in rendering and simulating Virtual Environments has caused many people to consider use of parallel or distributed computing. In parallel computing, several machines or processors are devoted to solving one problem in the shortest possible time. In distributed computing, system resources from a network of general-purpose computers work on solving many problems at the same time.

Very often applications need more computational power than a sequential computer can provide. One way of overcoming this limitation is to improve the operating speed of processors and other components so that they can offer the power required by computationally intensive applications such as Virtual Environments [29] [30].

A computing cluster is a collection of interconnected computers working together. This cluster functions as a single system from the point of view of users and applications. Such clusters can provide a cost-effective way to gain features and benefits that have historically been found only on more expensive proprietary shared memory systems [29]. Shared memory clusters offer a simple and general programming model, but they suffer from scalability problems.

Parallel and distributed programming involves more challenges than serial programming, such as data and task partitioning, task scheduling, and synchronization [10] [16] [37]. Writing parallel and distributed software may require substantial investment of time and effort [11]. Software performance is greatly affected by bandwidth and message latency, both of which are difficult to predict without direct measurement.

3. DLoVe and other distributed systems

There are three main differences between DLoVe and other parallel systems. While DLoVe's tasks appear externally similar to those in Parallel Virtual Machine (PVM), task allocation is done at compile time, so that there is not appreciable overhead for task management. The purpose of task allocation, in DLoVe, is to allow the Coordinator to always request the same Variables from the same Workers. In other words, the queries the Coordinator sends to the Workers are partitioned, so that the Workers can execute multiple different queries in parallel.

DLoVe's task handling, unlike PVM or Message Passing Interface (MPI), is designed to support multi-user, multi-input application development. Adding a second user to DLoVe's framework, adds a second Coordinator. This means that the Workers now have to serve requests for both Coordinators making each of the Workers work harder, consume more resources, and load the network with more messages.
The third difference concerns performance requirements. DLoVe is designed primarily for Virtual Reality applications and thus requires high frame rate. Distributed applications using DLoVe's framework are characterized by real-time computations and constraints. Thus, not only number of evaluations, but also timing factors need to be taken into account when evaluating DLoVe [15].
Timing constraints in DLoVe arise from interaction requirements between the Coordinator and the user, and between the Coordinator and the Workers. The communication between the Coordinator and the Workers is described by three operations: sampling, processing, and responding. The Coordinator continuously samples data from the input devices. Sampled data is sent to the Workers that process it immediately. Then the Workers send the processed data back to the Coordinator in response to its request. All three operations must be performed within specified times; these are the timing constraints [15] [36]. For example, if the user moves his real hand, and the movement of his virtual hand appears after a couple of seconds on the screen, the user may be confused and disoriented, making the application unusable.

4. Constraints and solving

Our continuous model is similar to a data-flow graph or a set of one-way constraints between actual inputs and outputs and draws on research in constraint systems [31] [33]. The model provides the ability to "re-wire" the graph from within the dialogue. Several researchers are using constraints for 2-D graphical interfaces [4] [5] [26] [27] [32]. Kaleidoscope [2] is a constraint-based language motivated by 2-D WIMP interfaces, and it explicitly supports temporary constraints.
VIVA [35] introduced some level-of-detail time management techniques in a data-driven, real-time constraint application. The CONDOR system uses a constraint or data-flow model to describe interactive 3-D graphics [23]. TBAG also uses constraints effectively for graphics and animation in the interface [7]. Gleicher provides constraints that are turned on and off by events [21]. Other recent work in 3-D interfaces uses a continuous approach [24] or a discrete, but data-driven approach [19].
Software architectures for virtual reality interfaces have been developed by Feiner and colleagues [34] and by Pausch and colleagues [20]. Green and colleagues developed a toolkit for building virtual reality systems [9]. Most of this work has thus far concentrated on the architecture or toolkit level, rather the user interface description language. Lewis, Koved, and Ling, addressed non-WIMP interfaces with one of the first UIMSs for virtual reality using concurrent event-based dialogues [13].

5. The paradigm

DLoVe is designed on a two-component model for describing and programming the fine-grained aspects of non-WIMP interaction (such as virtual environments). It is based on the notion that the essence of a non-WIMP dialogue is a set of continuous relationships, most of which are temporary. This model combines a data-flow or constraint-like component for the continuous relationships with an event-based component for the discrete interactions, which can enable or disable individual continuous relationships.
Other current Graphical User Interfaces (GUIs) or (WIMP) interfaces, also involve parallel, continuous interaction with the user. But, most other user interface description languages and software systems are based on serial, discrete, token-based interaction.
DLoVe is designed to provide a fundamentally continuous, rather than discrete, treatment of naturally continuous phenomena such as time and motion. However, it does treat discrete events as discrete events and it provides a mechanism for communication between the continuous and the discrete sub-system.

5.1. Continuous Subsystem

DLoVe's sub-system consists of object elements that define the relationship between variables. The entire set of these elements connected together form a constraint-like graph. Changes on one end of the graph propagate to the other end. Interaction that is conceptually continuous is encoded directly into these elements, and thus the application does not need to deal with tracking events from conceptually continuous devices. Examples of conceptually continuous interaction include, drinking from a cup in a virtual world, throwing a ball in a virtual park, and driving a car in a virtual city.

Continuous time in DLoVe is handled via Variables and Links, explained below. Based on these Links and Variables a network can be designed to describe the behavior of objects and interaction techniques.

Variables are objects in DLoVe that store values and know which Links need them as inputs and which for output. They are invariant data flow graph elements that serve as both continuous and short-term data repositories. Some Variables are directly connected to input devices, some to outputs and some to application semantics. They are used for communication within the user interface model or they are just used to hold intermediate results of Link calculations.

Links are objects that contain functions and are attached at both ends to Variables. Links get input from Variables and place the result of their calculations into other Variables. The body of a Link specifies how the attached Variables are related. Links can be enabled or disabled in response to user inputs. When a Link is disabled, it is as if this Link were not part of the constraint network anymore. By enabling and disabling Links we can quickly change the constraint network on the fly since only a flag needs to be set or cleared to indicate that a Link is enabled or disabled.

Conditions are also provided to enable and disable groups of Links instead of enabling or disabling Links individually. For example, we can attach five Links to a Condition; every time we want to enable all five Links we can just enable this particular Condition, which then enables all five Links individually.

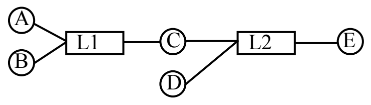

Figure 1. Constraint graph consisting of Links and Variables.

Figure 1 shows the Variables as circles and the Links as rectangles. When a Link is created, it is enabled by default. A DLoVe graph is read from left to right. For example, the Variables 'A' and 'B' are inputs to Link L1, and the Variable 'C' is its output Variable. When L1 is disabled, it is as if this Link was deleted from the network. However, a disabled Link is still part of the data structure and when it becomes enabled again it knows how it is supposed to be attached to its Variables. The relationship between 'A', 'B' and 'C' is terminated temporarily until 'L1' becomes enabled again.

5.2. Discrete Subsystem

There are however, other interactions that are fundamentally discrete (event-based) and for that we use the event-based component of DLoVe. Such examples include button presses, menu choices and gesture recognition verifications. TBAG applications [8] generally deal with such discrete input events by retracting some existing constraints and asserting new ones. Bramble uses a similar mechanism [25].

DLoVe handles the discrete time using Event Handlers, objects that capture tokens and respond to them. Event handlers contain a user specified body that describes the response to tokens. The application sends a token to all event handlers, and only those event handlers that are interested in the token execute their body. The responses might include setting Variables, making custom procedure calls and setting or clearing Conditions on Links. Event handlers recognize states and state transitions, and can provide different services depending on the state they are in. For example, the user might intersect his/her virtual hand with a virtual object. The event handler will receive an INTERSECT token and it will move to its 'intersect' state. In this state if the user presses the left mouse button, the event handler enables a Condition and transitions to the 'dragging' state. As a result the object is now attached to the virtual hand so that wherever the user moves his/her hand the object follows the movement of the virtual hand. When the user

releases the mouse button, the event handler disables the Condition, transitions to the ‘start’ state, and the continuous relationship hand-object is terminated.
A token is a structure similar to a record in Pascal or struct in C. It contains a timestamp, an id, and other optional fields such as position and other user defined variables. For instance, when the user’s virtual hand intersects with a virtual object, the user can press the left mouse button to send a token. This token indicates that the virtual object should be attached to the user’s hand position in space, so that when the user moves his/her hand, the object moves along with his/her hand. Releasing the mouse button sends another token that indicates that this hand–object relation should be terminated. In this case, when the user’s hand intersects with the virtual object and then the user presses the left button; the LEFTDN token is send to all event handlers. The event handler that is responsible for attaching the virtual object to the user’s virtual hand enables the appropriate Link so that a hand-object relation is established.
The application reads all X events and hands them over to a DeviceXWindow DLoVe object. This object then turns X events into tokens and sends them to all event handlers.
Enabling a Link is similar to asserting a constraint, and disabling a Link is similar to retracting a constraint from the constraint graph. Even though enabling and disabling Links is similar to retracting and asserting constraints, enabling and disabling occurs more quickly; Enable only marks a Link as part of the graph and Disable as not. The global structure of the constraint graph does not vary. When a Link becomes enabled, a single flag is cleared to indicate that this particular Link is part of the constraint graph again. There is no need to remember and set all the dependencies, since the object was never deleted from the memory. This makes DLoVe very efficient for modifying the constraint graph on the fly. At creation time the programmer specifies the dependencies of a Link and does not have to remember them ever again when he/she needs to assert/retract a constraint.
Let us say that in a virtual factory there are moving objects on an assembly line. Let us also assume that Variable v2 is time and Variable v1 is the position of a virtual hand. When the object on the assembly line is moving, only Link L2 is enabled. This way, the object’s position depends on time. However, if the user grabs the object with his virtual hand to examine the moving part/object, Link L2 becomes disabled, while Link L1 becomes enabled. Now the user has the ability to lift the object off the assembly line and examine it. The object is no longer on the assembly line, and its position does not depend on time. The constraint graph in the following figure is (a) when the user examines the object, and is (b) when the object is moving on the assembly line.

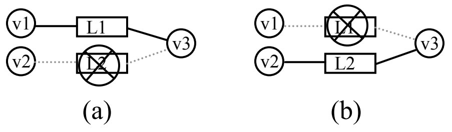

Figure 2. Enabling and disabling Links

The state machine that depicts the virtual factory example is given below. Tokens are fired when there is a specific transition in the state machine which enable or disable Links in the constraint graph.

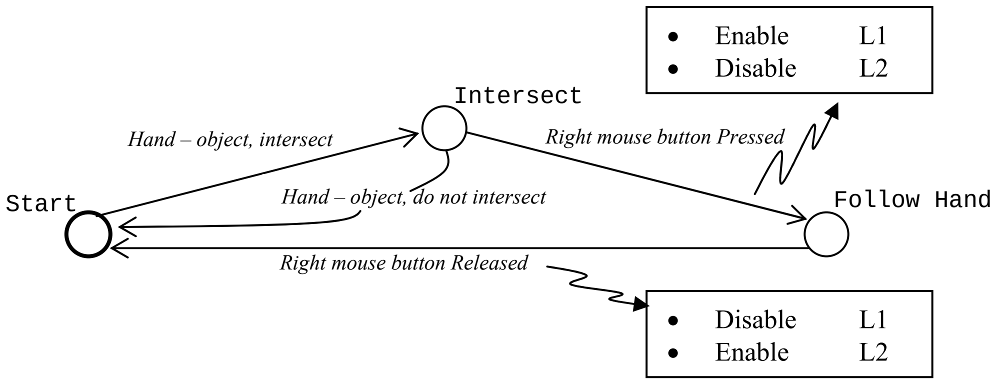

Figure 3. State Diagram of the Event Handler in the factory example

When the user's hand intersects with the moving object, the event handler receives a token (e.g. "ENTER"), the object becomes highlighted, and the event handler transitions to the 'Intersect' state. At this state, if the user presses the right mouse button, the event handler receives another token (e.g. "LEFTDN"), transitions to the "Follow Hand" state, enables Link L1, and disables Link L2. Now a Hand – object relation has been established, and the object follows the movement of the hand. When the mouse button is released, the event handler receives another token (e.g. "LEFTUP"), transitions to the 'start' state, disables Link L1, and enables Link L2. At this point, the hand-object relationship is terminated.

6. Architecture for parallel processing

One of the major difficulties with existing software lies in transforming a program written for a sequential machine into a program that runs on multiple machines on a network in parallel. A program written in DLoVe however, can be initially written to execute on a single sequential machine; with minor programming effort the same program can execute in parallel on multiple workstations. The only difference between the parallel and serial versions is that different libraries are used in linking against this program. When compiling for parallel execution, compilation generates two different executables, the executable for the Coordinator (the machine mainly responsible for the graphics and the input devices), and another executable for the Workers (the machines responsible for doing constraint calculations). The Coordinator is a workstation with a display device and input devices. It is responsible for reading all data from the input devices and generating the graphics. Workers are only responsible for doing calculations based on the Coordinator's requests. Workers are workstations that do not need to have any input or output devices attached to them: "headless workstations".
Every program that utilizes the DLoVe framework must call in the beginning of their main() the Link::InitCommunication() and Link::InitSystem() calls. When a DLoVe program is executed in non-distributed mode, the Link::InitCommunication() and Link::InitSystem() calls do not do anything and simply return. But, when the program executes in distributed mode, these calls partition the constraint graph and set up the connections between the Coordinator and the Workers to enable the application to perform parallel evaluations on the constraint graph.
The only information that needs to be shared among all machines in the distributed environment is the IP address and the port number the Coordinator is listening on. When multiple Coordinators participate, one of them is designated as the "Master" Coordinator and the rest as "Slaves". All Workers and all Slave Coordinators connect to the Master Coordinator that coordinates who connects to whom and when. The difference between the Master and the Slave Coordinators is that the Master Coordinator accepts connections from all Workers and all Slave Coordinators. Additionally, the Master Coordinator instructs the Slave Coordinators to accept connections from the Workers, and it also instructs the Workers to connect to the Slave Coordinators; all information about IP addresses and port numbers is distributed from the Master Coordinator. Figure 4 shows the sequence of message exchanges and connection setup between the Coordinator (referred as Master) and the Workers. Each Worker requests a connection to the Coordinator and identifies itself. Then the Coordinator partitions the constraint graph and assigns tasks to each Worker. Then the application starts executing and the Coordinator sends requests to the Workers that process data in parallel and send the results back to the Coordinator.

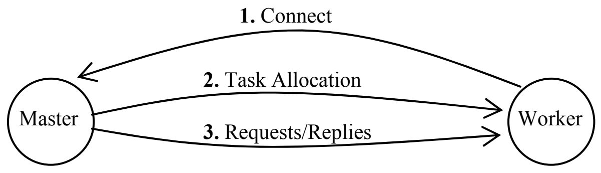

Figure 4 . Master – Workers, connection set up.

When there are multiple Coordinators, figure 5, initially Workers and Slave Coordinators request a connection to the Master Coordinator. After accepting all connections from all the Workers and all Slave Coordinators, the Master Coordinator partitions the constraint graph, assigns tasks to the Workers and instructs the Slave Coordinators to start accepting connections from the Workers. Then, the Master

Coordinator instructs each Worker to connect to each Slave Coordinator. When all Workers connect to all Slave Coordinators, the application starts executing and the Coordinators (Master and Slaves) send requests to Workers that process data in parallel and send the results back to the appropriate Coordinators.

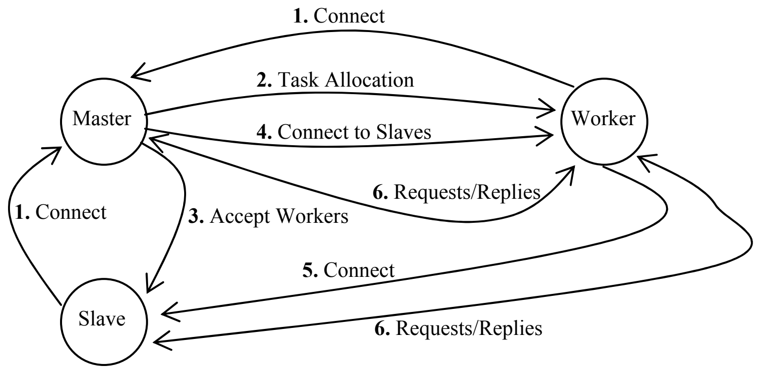

Figure 5. Master – Slaves – Workers, connection set up.

In the distributed mode, several modules (called "managers") cooperate during the connection setup ('Set up' section of figure 6). Runtime managers ('Run Time' section in figure 6) handle communication between Coordinator and Workers when the application is in execution. This communication always consists of requests from the Coordinator(s) that invoke responses from Worker(s). There is no direct communication between Workers other than though the Coordinator(s). The "Connection Manager" is a module that accepts connections from the Workers. On the Workers' side, the Connection manager is a module that initiates connections to Coordinators. During this process, each authenticates itself, exchanging information with the Coordinator and establishes a communication channel. The "Executive Manager" on the Coordinator is only involved when the system executes in Multi-user mode and the Slave Coordinators connect to the Master Coordinator.

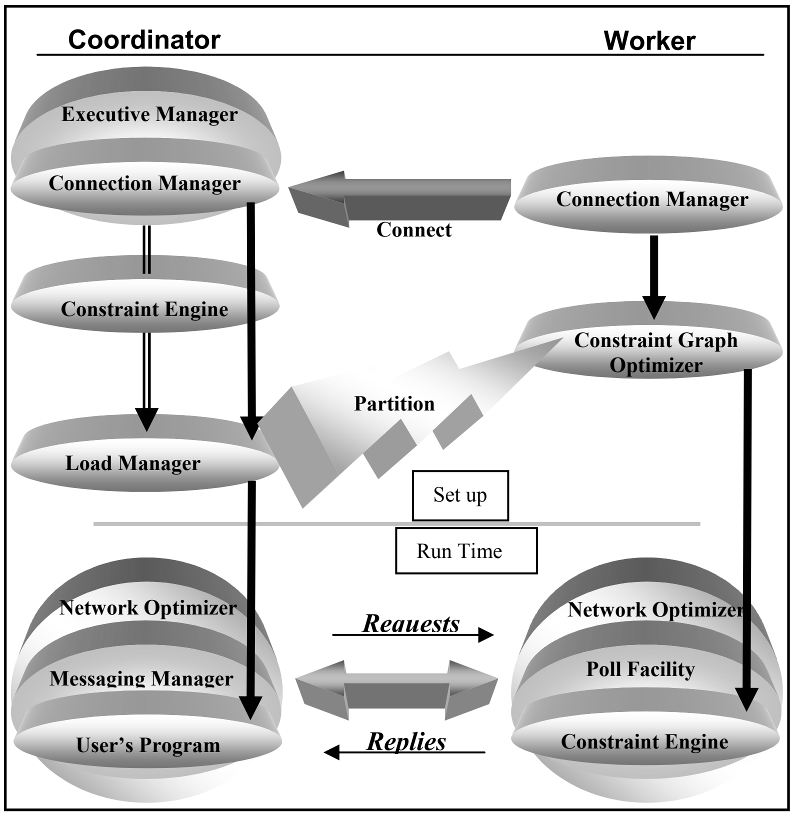

Figure 6. DLoVe's managers

After the Master Coordinator accepts all connections from all machines, partitions the constraint graph and assigns a disjoined subset of Variables to each Worker using its "Load Manager". These are the Variables a Worker will be "responsible for computing/keeping up to date". When each Worker receives its assignment, it executes its own algorithm in the "Constraint Graph Optimizer" manager to determine which sub-graph of the whole graph it will need to evaluate in order to compute its assigned Variables. For example, the following graph will be partitioned into two sub-graphs as shown below:

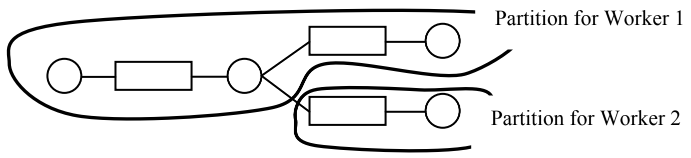

Figure 7. Partition into two sub-graphs

Even though Worker 1 has been assigned 3 Variables, by bringing the rightmost Variable up to date, all the assigned Variables become up to date because of the dependencies in the constraint graph. After partitioning is completed, the Coordinator(s) begins executing the main application and the Workers start listening to requests from the Coordinator(s). The Coordinator reads all its input devices, requests Variables in parallel from the Workers, and renders the screen. The Workers handle all requests from the Coordinator. We should note that the Coordinators own a copy of the constraint engine as well. Variables that describe the field of view of the immersed person are updated locally by the Coordinators; since the calculations involved are computationally inexpensive.
The last set of Managers in the 'Run Time' section handle all request-reply pairs between the Coordinator and Worker(s). The Coordinator executes the user's program. Using the "Message Manager" it incorporates requests into network messages that can be passed to Workers over the network. The "Network Optimizer" in the Coordinator is responsible for building large messages out of smaller ones, so that it can use the network more efficiently. It also makes sure not to overflow network capacity by sending too many messages over the network, similar to a technique used in [39].
Critical messages, such as commands that enable or disable a Link, are always sent. Non-critical messages may be dropped if Workers seem overloaded. Each Worker uses the "Poll Facility" to get requests from the Coordinator and unpacks messages that may contain multiple requests. Then the "Constraint Engines" in the Workers process all the requests; updating the constraint graph and communicate the result back to the Coordinator.

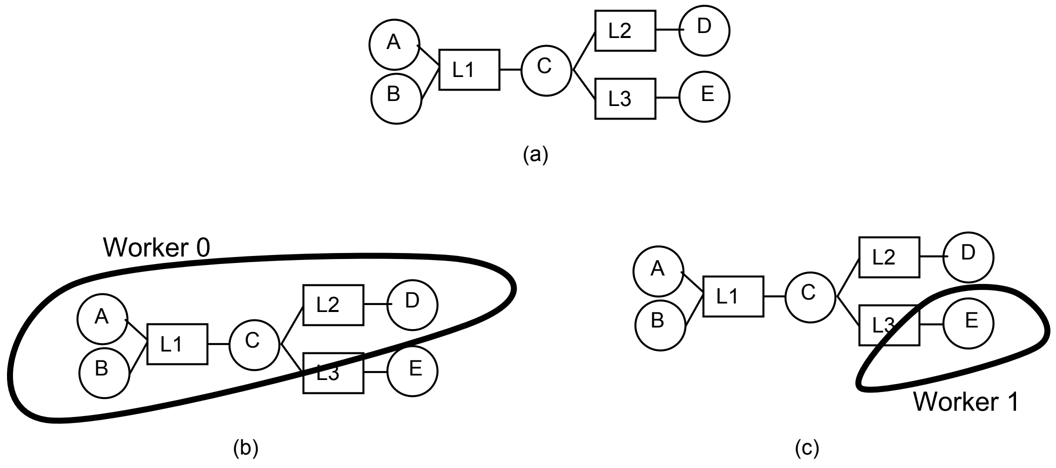

Figure 8. Partitioned graph to two Workers.

Every Worker and the Coordinator owns the same constraint graph representation. However, the Coordinator partitions the graph and assigns Variables to Workers. The Workers pay attention only to Variables that they have been assigned by the Coordinator. The set of Variables assigned to them represents a sub-graph of the entire constraint graph. For example, in the above example (a) where only two Workers are participating, Worker 0 will be assigned the Variables in (b) and Worker 1 will be assigned the Variable in (c). After the Workers run the optimization function, the Worker 0 will pay all its attention to Variable 'D' as shown in (a) below and Worker 1 to Variable 'E' as shown in (b) below. This way the Worker 0, for example, knows that by bringing the Variable 'D' up-to-date, all its Variables assigned by the Coordinator, are also up-to-date.

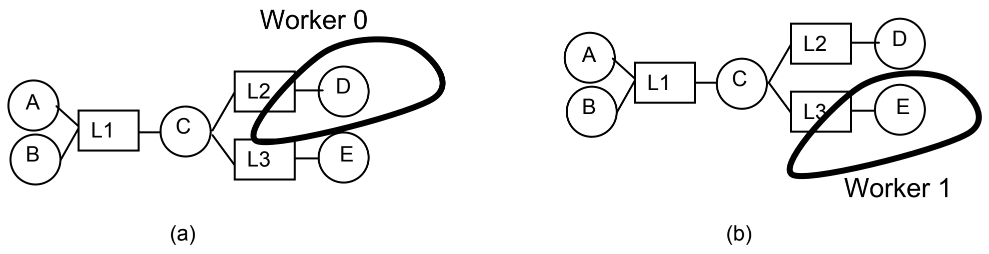

Figure 9. Partitioned graph after the optimization algorithm is run on the Workers.

7. Performance evaluation

Two factors greatly influence the goodness of a graph partition. A given graph should be partitioned into concurrent modules to obtain the shortest possible program execution time. Secondly, one must choose the best size for each concurrent module that will result in fastest execution, while using a minimal number of processors. The grain-packing problem is related to optimal scheduling where an "optimal" scheduler executes in minimum time, well known to be NP complete in general.

DLoVe's partition algorithm may be compared against load balancing and scheduling, but in reality it is someplace in between. Load Balancing, and more precisely Dynamic Load Balancing, tries to keep all processors equally busy, but does not try to reduce overall execution time. Only when the calculations are more expensive than the communication, the applications may run faster [12]. Dynamic Load Balancing handles process migration and reacts to conditions that vary in the network. DLoVe's partition algorithm determines at compile time how to partition the constraint graph. DLoVe does not modify the partition of a given graph dynamically to efficiently implement Dynamic Load Balancing. It tries to partition the graph assuming that all Links are equally computationally expensive.

DLoVe assigns tasks to Workers in a manner similar to a scheduler. A single task in a scheduler corresponds to a single Link where in DLoVe a task corresponds of a set of interconnected Links. DLoVe uses the constraint graph itself to give extra information, making determining task allocation easier. Though a real scheduler allows tasks to have any cost, DLoVe assumes that all Links comprising a task have the same cost. This may cause DLoVe to create unbalanced task schedules but seems to work fine for many constraint graphs used in Virtual Reality.

DLoVe uses a greedy round-robin scheduling algorithm, which is not optimal in general. For example, assume there are 5 tasks with costs 3, 2, 2, 1, 1, and suppose there are two Workers available. Using greedy round-robin assignment, Worker 1 will be assigned tasks with costs 3+2+1=6 while Worker 2 will be assigned tasks with costs 2+1=3. An optimal scheduler might assign tasks of cost 3+1+1=5 to Worker 1 and tasks of cost 2+2=4 to Worker 2, with imbalance of 1 time unit.

7.1. Strategy for analysis

One of the applications we designed to analyze the characteristics of the DLoVe framework was a Virtual Park. In the Virtual Park there were 32 Humanoids that were interacting among themselves, a virtual ball, and the user(s), the fully immersed person(s).

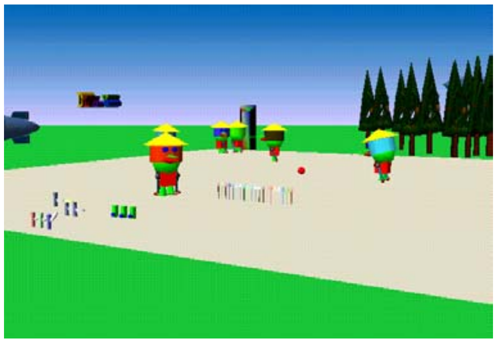

Figure 10. DLoVe Virtual Park

We measured the number of evaluations and the frame rate we were getting out of the system. We had two versions of the program, the first one that run on a single machine, and the second version that run on a distributed environment utilizing three machines. In conventional parallel processing systems the number of evaluations, throughput, is what counts. However, in Virtual Reality latency is also very critical. High throughput that produce frames in front of the fully immersed person that are out-of-date make Virtual Reality unusable. Results of these experiments show that traditional techniques for performance analysis do not accurately describe distributed VR performance.

Initial measurements suggested that conventional measures used to evaluate parallel processing systems are less useful in VR [1] [6] [11].

Table 1 shows that by using more Workers, we increase the throughput of the system but we decrease the frame rate. The throughput increase is due to parallel computations performed by the multiple Workers, where the decrease in frame rate is due to the fact the DLoVe is implemented on top of TCP. The Coordinator needs to send multicasts to all Workers and thus it has to simulate multicasting [17], since it is not supported by TCP. By simulating multicasting, the Coordinator spends more time trying to send the requests to the Workers and less time to render the display. We measured the number of evaluations of one of the Links in the Humanoids which was needed in every frame to figure out how many times a Humanoid was brought up to date.

Table 1. Throughput and frame rate performance.

Number of Workers	Number of Evaluations	Frame Rate
1	1700	20
2	3500	18
3	4500	15

When we use more Workers, we get fewer frames per second because the Coordinator saturates the network faster due to sending the same messages to multiple Workers. The Coordinator sends the same messages to all Workers. For example, when there are three Workers the Coordinator has to send the same message three times, once for each Worker. This adds a lot of overhead to the Coordinator and as a result it spends less time updating the display. This is because the Coordinator is trying to write too many messages on the network, and the network becomes saturated. The Coordinator keeps trying until it successfully writes the messages on the network, losing critical time on updating the display.

The distributed version outperformed the non-distributed version in frame rate, because in the non-distributed version one machine is trying to simulate all 32 Humanoids, leaving it no time to update the display. In fact, the non-distributed version was so slow (2 frames per second) that it would disorient any person using it. The frame rate on the distributed version could be improved if a different technique was used, for network communication, such as multicasting instead of unicasting. If DLoVe outperforms the non-distributed version using TCP/IP for point-to-point communication, it promises even greater performance if it is re-implemented using multicasting.

However, throughput of a Virtual Reality system does not describe how well it performs. A more critical factor in VR is how high a frame rate it can achieve, and how accurately each frame represents the virtual world.

Accuracy of rendition is difficult to measure. The Coordinator sends requests to Workers to have certain Variables updated so it can render its display. The Workers update the requested Variables and the results are sent back to the Coordinator. These replies are not instantaneous, but arrive with some latency that depends on performance upon the network. So, one measure of accuracy is message latency. Another is how up-to-date the frames are when rendered, relative to user input and the real state of the virtual world.
For more information on the configuration of the machines, the network, and the software utilized refer to [17].

7.2. State machine of the run time system

The Coordinator can be thought of as possessing five independent states as shown in Figure 11. The Coordinator builds a message consisting of several requests and sends it out to the Worker(s). Initially, the Coordinator is in a 'start' state. When it sees a need to request Variable's value or to enable or disable a Link, it transitions to the 'build' state within which it builds the message that it may send to the Workers. At the end of the main loop, just before sending the message, it checks its internal counters, which indicate the number of pending requests on the network to each Worker. If a counter is below a chosen threshold, it transitions to 'send' state, sends the message, and returns to the 'start' state. Else, it discards the message and returns to the 'start' state.
While in the 'start' state, it checks to see if any replies came back from the Workers. If replies have arrived, it transitions to the 'get' state and processes all the replies. When all replies are processed, it transitions back to the 'start' state where it starts building messages all over again. The 'start' state is a state indicating idle time and it is used as a starting point in describing the functionality of the Coordinator. The fifth state is the 'render' state in which the Coordinator transitions to render the display. The following figure shows the five states and the transitions between them. It also shows where we placed application hooks to measure the number of messages DLoVe transmitted (# of Sends), number of messages discarded (# of Drops) and finally the number of messages that the Workers sent to the Coordinator (# of Gets).

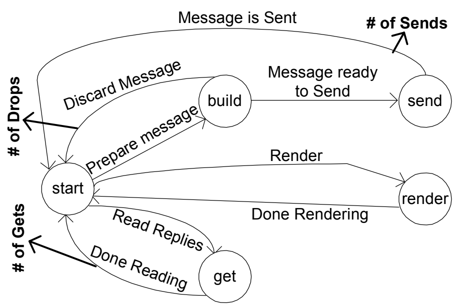

Figure 11. State machine of the run-time system

Performance in time required measuring the elapsed time between pairs of state transitions as shown in the figure 12. t_{build} is the time it takes the Coordinator to build a message before sending it to the Workers or discarding it, which is the difference between t_{build_end} and t_{build_start} , or between t_{build_start} and $t_{build\_discard}$ representing the time at which the build ended and when it started. t_{send} is the time it takes the Coordinator to put the message on the network, which is the difference between t_{send_end} and t_{send_start} indicating the time the Coordinator completed the transmition of the message and the time it started transmitting. t_{get} is the time it takes the Coordinator to process the replies from the Workers, which is the difference between t_{get_start} and t_{get_end} indicating the time it started reading replies and the time it read all replies from all Workers. t_{cycle} indicates the time it takes a request to come back as a reply, which is the difference between t_{build_end} and t_{get_end}. This is the latency of each message and this is what in which we were the most interested. One of the goals we want to achieve is for the Coordinator to receive as many replies as possible from the Workers. However, if the messages are all too old, then user interaction is minimized and this is not what we want since DLoVe is designed for real-time applications such as Virtual Reality.

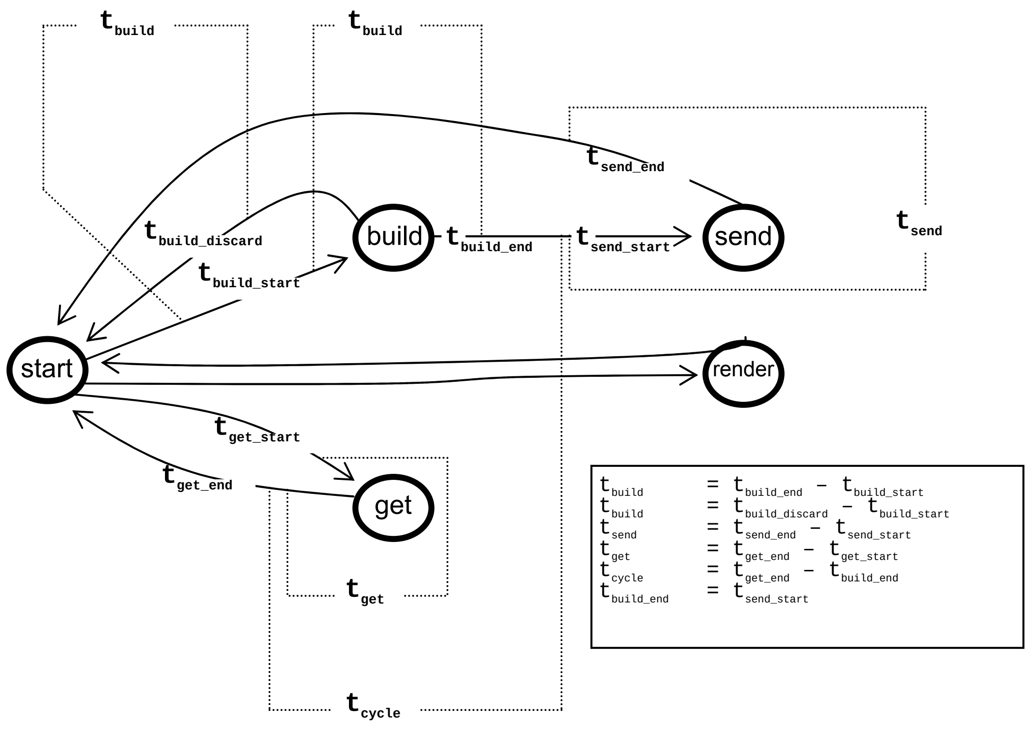
t build
t build
t send_end
t send
t build_discard
build
t build_end
t send_start
send
t build_start
start
render
t get_start
t get_end
get
t get
t cycle
t build = t build_end – t build_start
t build = t build_discard – t build_start
t send = t send_end – t send_start
t get = t get_end – t get_start
t cycle = t get_end – t build_end
t build_end = t send_start

Figure 12. Measuring latency in the internal state machine

7.3. Frame Validity and Statistical Skew

Latency shows how long a message spends on the network to get to the Workers and then come back to the Coordinator - time required by a Worker to evaluate a Link is not counted. However, this does not show how accurate the frame rendering is and it does not capture the interactive performance of such user interfaces. The system might be doing a great deal of processing by generating many frames, but displaying each of them late. It would score well by traditional parallel processing measures, but would seem very sluggish to the user. We therefore used total throughput along with total latency of a displayed video frame to measure performance. That is, when a frame is displayed, what is the oldest piece of data that was used to generate that frame? We believe that frame latency, in conjunction with frame rate, captures the interactive performance of a VR user interface better than total processing throughput. Frame latency shows how valid or how late the information is at the time the Coordinator renders the display.
To perform this analysis, we tracked each message through the network. Message latency shows how long a message spends on the network to get to the Workers and then to come back to the Coordinator (minus the time required by a Worker to bring the requested Variable up-to-date). Message latency is the time this request spent in the network. However, this does not show how accurate a rendered frame is. To visualize how valid the frames are, we used a statistical clock skew that plots the oldest information used to render the display [18]:

skew = (wall clock) *minus* min(time of request of all Variables)

For every frame, the minimum time of request of all output Variables is subtracted from the current time (wall clock). This skew describes the worst difference between what is rendered and what the user is doing. We plotted many graphs that show this skew over time [17]. Every time a request comes back from a Worker it is time-stamped with the current time indicating the time the Variable is lastly updated. When the Coordinator is about to render the display it gets the system's clock and subtracts from it the minimum time of the Variables in the state "render".

7.4. Throughput in DLoVe

To better understand the characteristics of DLoVe, we designed a very simple application where the rendering of the display took zero time. We designed this program to see the how much the network impacts the performance of an application that utilizes DLoVe's framework. The application was simply sending requests to the Workers as fast as possible. There were three Workers performing parallel computation on a constraint graph that is shown below:

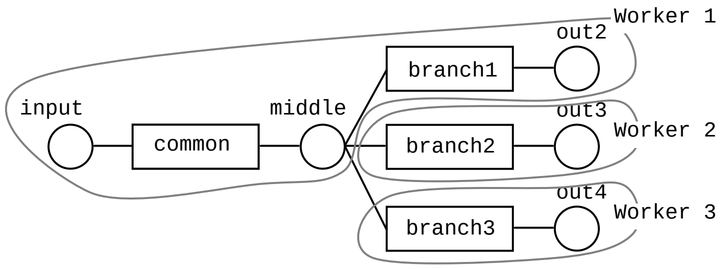

Figure 13. Partitioned constraint graph.

Using the statistical skew described above, we plotted the latency of each frame (see [17] for many more graphs). The graph below shows the latency of each frame that describes the worst difference between what is rendered and what the user is doing.

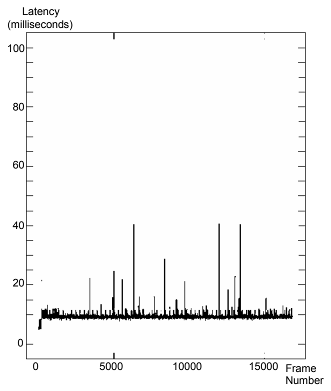

Figure 14. Frame Latency Graph.

The message latency stays consistent through out the execution of the application while keeping the frame rate as high as possible. Finally, observe that these speedups are made possible by a high-level constraint-based UIDL, which placed no assumptions or restrictions on the processing sequence. Our experience with DLoVe demonstrates that introducing a higher-level, declarative user interface description language provided extra degrees of freedom to the underlying implementation. Rather than causing a performance penalty, the constraint-based language made possible the increased performance we obtained through parallel processing.

8. Conclusion

The experiments demonstrated that DLoVe not only describes specification of Virtual Reality programs well, but also improves overall performance of applications designed in its framework by dramatically increasing the validity of the rendered frames. In addition, DLoVe supports mechanics for implementing or transforming single user programs into multi-user programs. DLoVe can be used to implement large-scale

Virtual Reality applications, and when speed is required, DLoVe's framework is able to provide the additional CPU cycles needed by real time applications by utilizing multiple machines.
We showed the need for a different method of measuring performance that describes DLoVe accurately, where traditional methods fail by providing seemingly acceptable performance measures that in actuality reflected poor performance. Throughput is not enough to describe performance, but throughput and frame rate in conjunction with statistical skew, which measures the freshness of each rendered frame, do accurately describe DLoVe's performance.

References

1. Amdahl, G.M. Validity of the single-processor approach to achieving large scale computing capabilities. In AFIPS Conference Proceedings vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483-485.
2. B. Freeman-Benson and A. Borning, "The Design and Implementation of Kaleidoscope'90, a Constraint Imperative Programming Language," Proc. IEEE Computer Society International Conference on Computer Languages, pp. 174-180, April 1992.
3. B. Shneiderman, "Designing the User Interface: Strategies for Effective Human-Computer Interaction", Second Edition, Addison-Wesley, Reading, MA, 1992.
4. B. Vander Zanden, B.A. Myers, D.A. Giuse, and P. Szekely, "Integrating Pointer Variables into One-Way Constraint Models," ACM Transactions on Computer-Human Interaction, vol. 1, no. 2, pp. 161-213, June 1994.
5. B.A. Myers, D.A. Giuse, R.B. Dannenberg, B. Vander Zanden, D.S. Kosbie, E. Pervin, A. Mickish, and P. Marchal, "Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces." IEEE Computer, vol. 23, no. 11, pp. 71-85, November 1990.
6. Berman, Kenneth A. and Paul, Jerome L. "Fundamentals of Sequential and Parallel Algorithms". PWS Publishing Company, 1997.
7. C. Elliot, G. Schechter, R. Yeung, and S. Abi-Ezzi, "TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications," Proc. ACM SIGGRAPH'94 Conference, pp. 421-434, Addison-Wesley/ACM Press, 1994.
8. C. Elliott, G. Schechter, R. Yeung and S. Abi-Ezzi. "TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications", In Proc. ACM SIGGRAPH '94, pages 421-434, August, 1994.
9. C. Shaw, M. Green, J. Liang, and Y. Sun, "Decoupled Simulation in Virtual Reality with the MR Toolkit," ACM Transactions on Information Systems, vol. 11, no. 3, pp. 287-317, 1993.
10. Clark D. D. and D. L. Tennenhouse , "Architectural considerations for a new generation protocols", SIGCOMM 1990, Sep. 1990, Computer Communication Review, 20(4), 200-208
11. Hesham El-Rewini & Ted G. Lewis, "Distributed and Parallel Computing". Manninning Publications Co. 1998.
12. Hesham El-Rewini, Theodore G. Lewis, and Hesham H. Ali. "Task Scheduling in Parallel and Distributed Systems". PTR Prentice Hall, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 1994.
13. J.B. Lewis, L. Koved, and D.T. Ling, "Dialogue Structures for Virtual Worlds," Proc. ACM CHI'91 Human Factors in Computing Systems Conference, pp. 131-136, Addison-Wesley/ACM Press, 1991.
14. J.D. Foley, "Interfaces for Advanced Computing", Scientific American, v257, n4 p127-135, October 1987.
15. Jeffrey J. P. Tsai, Yaodong Bi, Steve J. H. Yang, and Ross A. W. Smith, "Distributed Real-Time Systems", John Wiley & Sons, Inc. 1996.
16. Jon Crowcroft, "Open Distributed Systems", Artech House, Inc., MA 1995.
17. L. Deligiannidis, "DLoVe: A specification paradigm for designing distributed VR applications for single or multiple users", Doctoral dissertation, Tufts University, Feb. 2000.
18. L. Deligiannidis, Alva. Couch, R.J.K. Jacob, "Performance Characterization of Distributed Virtual-Reality Applications: A Case Study", Proc. of the 3rd International Conference on Internet Computing, CSREA Press and IEEE Internet Computing Magazine, June 2002.
19. L.D. Bergman, J.S. Richardson, D.C. Richardson, and F.P. Brooks, "VIEW - An Exploratory Molecular Visualization System with User-Definable Interaction Sequences," Proc. ACM SIGGRAPH'93 Conference, pp. 117-126, Addison-Wesley/ACM Press, 1993.

20. M. Conway, R. Pausch, R. Gossweiler, and T. Burnette, “Alice: A Rapid Prototyping System for Building Virtual Environments,” Adjunct Proceedings of ACM CHI'94 Human Factors in Computing Systems Conference, vol. 2, pp. 295-296, 1994.
21. M. Gleicher, “A Graphics Toolkit Based on Differential Constraints,” Proc. ACM UIST'93 Symposium on User Interface Software and Technology, pp. 109-120, Addison-Wesley/ACM Press, Atlanta, Ga., 1993.
22. M. Green, and R.J.K. Jacob, “Software Architectures and Metaphors for Non-WIMP User Interfaces”, Computer Graphics, v25, n3 p229-235, July 1991.
23. M. Kass, “CONDOR: Constraint-Based Dataflow,” Proc. ACM SIGGRAPH'92 Conference, pp. 321-330, Addison-Wesley/ACM Press, 1992.
24. M.P. Stevens, R.C. Zeleznik, and J.F. Hughes, “An Architecture for an Extensible 3D Interface Toolkit,” Proc. ACM UIST'94 Symposium on User Interface Software and Technology, pp. 59-67, Addison-Wesley/ACM Press, Marina del Rey, Calif., 1994.
25. Michael Gleicher. “A Graphical Toolkit Based on Differential Constraints”. UIST '93 November 1993, pages 109-120.
26. R.D. Hill, “The Rendezvous Constraint Maintenance System,” Proc. ACM UIST'93 Symposium on User Interface Software and Technology, pp. 225-234, Atlanta, Ga., 1993.
27. R.D. Hill, T. Brinck, S.L. Rohall, J.F. Patterson, and W. Wilner, “The Rendezvous Architecture and Language for Constructing Multiuser Applications,” ACM Transactions on Computer-Human Interaction, vol. 1, no. 2, pp. 81-125, June 1994.
28. R.J.K. Jacob, L. Deligiannidis, and S. Morrison, ”A Software Model and Specification Language for Non-WIMP User Interfaces” ACM Transactions on Computer-Human Interaction, Vol. 6(1) pp. 1-46 (March 1999).
29. Rajkuma Buyya, “High Performance Cluster Computing” Vol. 1 (Architectures and Systems), Prentice Hall PTR, New Jersey 1999.
30. Rajkuma Buyya, “High Performance Cluster Computing” Vol. 2 (Programming and Applications), Prentice Hall PTR, New Jersey 1999.
31. S. Hudson and I. Smith, “Practical System for Compiling One-Way Constraint into C++ Objects,” Technical Report, Georgia Tech Graphics, Visualization, and Usability Center, 1994.
32. S.E. Hudson, “Graphical Specification of Flexible User Interface Displays,” Proc. ACM UIST'89 Symposium on User Interface Software and Technology, pp. 105-114, Addison-Wesley/ACM Press, Williamsburg, Va., 1989.
33. S.E. Hudson, “Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update,” ACM Transactions on Programming Languages and Systems, vol. 13, no. 3, pp. 315-341, July 1991.
34. S.K. Feiner and C.M. Beshers, “Worlds within Worlds: Metaphors for Exploring n-Dimensional Virtual Worlds,” Proc. ACM UIST'90 Symposium on User Interface Software and Technology, pp. 76-83, Addison-Wesley/ACM Press, Snowbird, Utah, 1990.
35. S.L. Tanimoto, “VIVA: A Visual Language for Image Processing,” Journal of Visual Languages and Computing, vol. 1, no. 2, pp. 127-139, June 1990.
36. Selic, Bran and Gullekson, Garth and Ward, Paul T. “Real Time Object Oriented Modeling”. John Wiley & Sons, Inc., 1994.
37. Shoch, J. F. and J. A. Hupp, “The Worm programs – early experience with a distributed computation”, Communications of the ACM 25(3) 1982.
38. V. S. Sunderam, PVM: A Framework for Parallel Distributed Computing, Concurrency: Practice and Experience, 2, 4, pp 315--339, December, 1990.
39. V. Tsaoussidis, H. Badr, "TCP-Probing: Towards an Error Control Schema with Energy and Throughput Performance Gains" The 8th IEEE Conference on Network Protocols, ICNP 2000, Osaka, Japan, November 2000.