Proceedings of the 1995 Winter Simulation Conference
ed. C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman

DISplay: A SYSTEM FOR VISUAL-INTERACTION IN DISTRIBUTED SIMULATIONS

Edward Mascarenhas
Vernon Rego

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907, U.S.A.

Janche Sang

Department of Computer and Information Science
Cleveland State University
Cleveland, Ohio 44115, U.S.A.

ABSTRACT

We propose an application-independent Visual Interaction Library (VIL) well suited to distributed simulation. Advantages of this system include ease of use, flexibility, code reuse, and modularity. Our design ideas are manifest in the DISplay system, a graphical user-interaction and display library which binds to any parallel software system. We provide examples of its interactive use in the the dynamic display of results from sequential queueing simulations, and distributed particle-physics simulations. These examples illustrate synchronization of multiple remote display requests and potential for enhanced parallel simulation. Also presented are provisions for customized user-interaction dialogs – for application-related input, and bi-directional interaction – between user and application.

1 INTRODUCTION

Graphical visualization is a useful simulation aid. Several commercial software packages provide run-time animation and graphical views of simulation output. Visualization is undoubtedly useful, and sometimes crucial in difficult modeling problems and performance-related simulation experimentation. Examples of its use arise in decision making, model validation, result-display, model illustration/animation, performance monitoring, simulation education, etc. In this paper we present the DISplay Unix-based system based on X11 and PEX (Gaskins 1992). Display is a Visual Interaction Library (VIL) enabling Visual Interactive Simulation (VIS) in sequential and distributed environments.

For the graphics novice, access to interactive visualization is usually possible only through specific simulation software systems. When available, visualization functionality is often tightly bound to application domains (e.g., queueing, manufacturing systems). As an aid to simulationists who require graphics-interaction functionality without being tied to particular software systems, simulation domains, or steep system-learning curves, we designed and built an easy-to-use and application-independent VIL. Advantages of this approach include *ease of use* – specialized display knowledge is encapsulated in the system so that graphics novices can produce application-specific displays, *flexibility* – the system can be used for a variety of applications, including parallel and distributed simulations, *code reuse* – graphical interaction code need not be reconstructed for different applications, and *modularity* – visualization and interaction portions of the application can be layered on top of the application, facilitating design changes and code modification.

1.1 Visual Interactive Systems

The subject of user-interaction in the simulation context is not new. How can a simulationist interact with an executing simulation and view results in real time ? More importantly, how can this be done in an application-independent manner ? O'Keefe (1987) discusses the basic ideas underlying potentially responsive systems, and proposes a methodology for Visual Interactive Simulation. Rooks (1991) also gives a proposal for Visual Interactive Simulation, outlining general requirements and a potentially unifying framework. Requirements delineated in the Rooks proposal include

Intervention: allowing the initiation of interaction with the simulation model. Modes of interaction are inspection, specification and visualization.

Inspection: allowing access to all simulation data for reading and/or writing.

Specification: allowing specification of model parameters.

Visualization: allowing diverse visualizations of model data, so that model dynamics and relationships of interest can be illustrated.

In early literature on this subject, the term VIS (O'Keefe 1987) was used to include activities which are now generally regarded as components of Visual Interactive Modelling (VIM) (Bell and O'Keefe 1987, Rooks 1991), where a computer model of a given system is created. More recently, researchers have attempted to separate and refine functions associated with VIS and VIM. In the rest of this paper, we focus our attention on VIS, with special emphasis on parallel and distributed simulation.

Early software systems supporting VIS were generally restricted to animations of application components of simulation models. Later, these systems added user-interaction capabilities in various forms. More recent systems supporting VIS include WITNESS (Thompson 1993) — which also allows interactive model building, Cinema (Glavach and Sturrock 1993) — which are used with the SIMAN simulation language, providing real-time or post-processed animation and building of a model, SIMSCRIPT II.5 (Russel 1989) — a language which provides an integrated graphical interface called SIMGRAPHICS, and TESS — a system associated with the SLAM (Pritsker 1986) simulation language.

2 The DISplay ARCHITECTURE

The DISplay software architecture is based on the client-server paradigm. Here, the simulation application, which is created by an analyst or programmer, is treated as a client. Client calls are made by invoking functions resident in the DISplay client library. The server portion of the architecture consists of a **connection server (CS)** to which clients connect, and a **DISplay server (DS)** which handles all graphics and user interaction requests from a distributed application (see Figure 1). Typically, the client connects to the **CS** during application initialization; the **CS** delivers a handle (i.e., socket) to the application for communication with the **DS**. The **DS** is created by the **CS** using a unix *fork* operation, and reads incoming messages from the application, creating the necessary display *tasks* and interaction *dialogs*, as specified by the application. Continuing display and user interaction is based on application specifics and objectives. Messages between the application and the **DS** form a well-defined set of primitives, part of the DISplay protocol. The **DS** connects to an X workstation to execute specified display and user interaction commands. The analyst can interact with an executing application from the workstation where the display is done, thus enabling a requisite interaction capability between analyst and model.

The **CS** performs an important function in the mechanism described above. It must be located on a well known machine and port, so that applications can readily connect to it. The connection is made using a unique `Name`, which identifies the application.

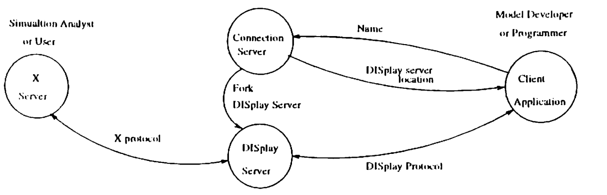


Figure 1: The Client Server Architecture

The **CS** maintains a list containing each active **DS**, along with its associated `Name`. In a multiprocessor application, say using PVM, Conch, or some other system, several processes, possibly residing on distinct (distributed) processors, may request a single, cooperative graphical display and user interaction. In this case, the **CS** delivers the same **DS** handle to each of these processes (from the same application), so that they may all connect to the same **DS**.

2.1 Use of DISplay in Parallel Simulation Environments

There are several approaches to developing parallel discrete event simulations. Of these, the primary approaches are *conservative*, *optimistic*, and *adaptive*. In the *conservative* approach events are processed strictly in order of occurrence, maintaining causality at all times. The approach is prone to deadlock, something that can be avoided via recourse to so-called *null messages* (Chandy and Misra 1979). In the *optimistic* approach, events are processed as they become available; potential causality conflicts are disregarded until a causality error is detected. Upon detection of such an error, invalidated simulation work is undone, the causality error corrected and the simulation re-executed from the point at which the error occurred. This approach requires some form of state-saving, so that simulation re-execution from a given point is possible. The *adaptive* approach is a mixture of the optimistic and the conservative approaches. Fujimoto (1990) provides a survey of these approaches.

Without loss of generality, we assume that the simulation environment is made up of a cluster of workstations which cooperate to execute a given model. Each workstation may host a variable number of processes. During a parallel simulation, distinct processors may be involved in computations associated with different virtual simulation times. Messages sent to the **DS** by different processors arrive at the **DS** with distinct virtual-time stamps. To provide the user with a consistent view of the simulated system, the **DS** buffers, sequences and processes messages only when it is certain that subsequent messages cannot induce causality violations. One way of achieving this is by adopting a conservative parallel simulation protocol in DISplay. Each channel (source destination link between processes, as viewed from the destination end) is associated with a channel time. This time is equal to the time-stamp of the first message queued on the channel buffer, if one exists, or the time-stamp of the last message retrieved from the channel buffer if the channel buffer is empty. At any given time, the first message from the buffer of the channel with the smallest channel time is selected for processing.

If a process has no messages to send, it avoids an indefinite delay of message selection at the DS by forwarding a `Time Update` message, which is basically the equivalent of a null message in the conservative approach. Observe that the **DS** may also be used in an optimistic parallel simulation, in which case messages will not be sent to the **DS** until they are committed and cannot be rolled back. This occurs when the time-stamp of a message is smaller than a parallel simulation's global virtual time. In this case however, `Time Update` messages are not required but synchronization is necessary so that a consistent view of the simulation is provided to the user. An example of the use of interactive graphics in the optimistic simulation system SPEEDES is given by Tung and Steinman (1992). For applications that have no notion of time, it is feasible to send all messages with the same time-stamp of 0.

3 THE DISplay SERVER

The power of the display server (**DS**) lies in its generality – it does not store application-specific state. It is application-independent. The **DS** accepts information from an application for the purpose of display, and it is up to the user to make decisions on types of displays and types of interactions desired with an application. The **DS** presents information to the user in a well-defined manner, accepts input which it forwards to the application – if the application requires this input, and displays results – if the application requests such a display. All interpretations of presentation are left to the user. Functions that the **DS** provides execute independently of the application. For example, clicking on a button to display a graph will not affect the executing application. This function is handled at the **DS** itself.

The DISplay system provides *tasks* and *dialogs* as basic mechanisms for implementing graphical displays and user interactions, respectively. *Tasks* are of two types: `Single message tasks` require a single message to be delivered, and their action is taken based on that message alone. `Multiple message tasks` require a setup phase, where the task is created locally and then executed at the server. The creation

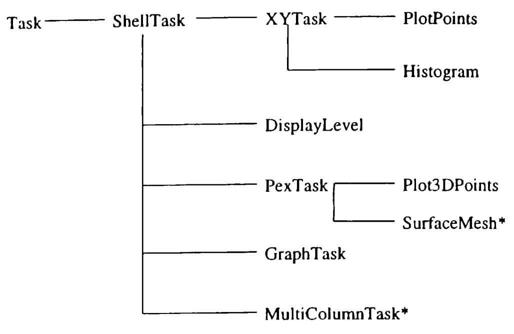


Figure 2: The C++ Task Hierarchy

mechanism returns a task identifier which is used as a handle in all subsequent messages related to the task. After task creation, multiple task-related messages may be sent to the **DS** to invoke the intended function. These messages are interpreted in the context of the task identifier. Tasks are deleted using an `EndTask` message. In a parallel execution environment, it is necessary for several processes to share a single resource at the server. DISplay permits *task* sharing between processes.

User interactions are also handled in a similar manner. Each possible user-model interaction is assigned a unique dialog identifier, with a *form* based dialog created. A dialog setup phase creates the dialog locally and then sets it up at the **DS**. Queries and answers using these dialogs are interpreted based on the dialog identifier and the position of the data within a dialog. It is possible to link dialogs to specific tasks, and to have tasks linked to specific dialogs. For example, the node or arc of a graph task can be associated with dialogs. Also, when a dialog is activated it may simply display an active task like a plot of points.

3.1 DISplay Classes

A central class, called the `controller` class, keeps track of all connected processes and their associated channels. It has a predefined maximum number of channels which are setup as each process connects to the **DS**. Each `channel` class records the socket over which its communication takes place, channel virtual-time, and a list of tasks and dialogs indexed by identifiers. Messages received on the channel are queued in FIFO mode. Other important classes in the system include the `task` and `dialog` classes.

3.1.1 Tasks

DISplay *Tasks* are used for any output requiring multiple messages. *Tasks* form an extensible set of textual, graphical and representational schemes for displaying the dynamics of a simulation and its end results. The C++ class hierarchy shown in Figure 2 describes the arrangement of tasks. The Task class provides control functions for maintaining the status of a task. When new task classes are added, these may utilize the functionality provided by the base task classes, such as the Task class. This facilitates extension of the interface with newer displays.

3.1.2 Dialogs

DISplay *dialogs* are basic mechanisms for user interaction. A *dialog* is defined as a *form* consisting of a set of values. Associated with each value is a set of items which describe that value. In the current implementation, this set of items is limited to a textual description of the item and a specification describing the type of the value. The textual description can also be used as a query to the user to input a value of the required type. Since all messages are logged, it is possible to do a post-run determination of times of interaction between user and application. This may be important in performance-tuning and analysis of simulations. Dialogs are of four types,

User Query Dialog: The request for value information is sent from the **DS** to the application. The application handles the query and returns the required values. Invocation is done by the user, who may fill in some of the values in the dialog. The application may use these in computation, returning computed values.

User Command Dialog: The user sends information to the application asynchronously. The application must be willing to handle these dialog messages through appropriate user written handlers which are registered with the dialog on creation.

Model Query Dialog: This functionality is similar to remote calls. The application pops up the dialog at the **DS** at predefined points in its execution, and waits for the user to reply. The application remains blocked until the user reply is received.

Model Display Dialog: Information is passed to the **DS** at predefined points in the application execution. The **DS** collects the information and displays it only when the user invokes the dialog (by clicking on the associated button). The information is received only when the application chooses to send.

4 VISUAL INTERACTION LIBRARY

Client functions are of two types: functions which send messages to the **DS** and functions which perform local processing prior to sending a message. Functions of the former type begin with an s prefix, while functions of the latter type begin with an sd prefix. At present the VIL contains 34 basic functions, and several convenience functions that invoke basic functions. In Figure 3 is shown an example with typical

```
main(int argc, char **argv)
 {
      char *simname; /* simulation name */
      char *host;    /* server location */
      char *service; /* server port */
      int  sock;     /* handle to the DS */
      int  ret;      /* return code */
      int window=TRUE, log=TRUE;
      int num_of_procs;

      ...;
      /* make the connection */
      sock = sGetServer(simname, host, service);
      if (sock == SDERROR) {
       /* handle the error here */
      }
      /* register process */
      ret = sNewProcessMsg(sock, window,
                           log, num_of_procs);
      /* rest of processing and messages to DS */

      ret = sEndServer(sock); /* close conn. */
 }
```

Figure 3: Example of Connecting to the Display Server

use of functions to connect to the **DS**. In setting up the connection, it is possible to specify whether logging is required, and whether a new main window is required. By specifying the number of processes in the simulation, all processes are guaranteed connection before the **DS** begins to act on messages from the simulation. For more details on the use of the client library functions see the User's Manual (Mascarenhas and Rego 1994).

Important functions for drawing in a window include sColPoint(), sPltPoint(), sPltPoly(), sPltArc(), sPltSurface(), sHistPoint(), and sDisplayLevel() - capable of 2D and 3D drawings. Function sColPoint() colors a point in the window with a specified color. The developer programs with world coordinates and color names. The **DS** converts world coordinates into corresponding window coordinates, and maps colors to pixel values that can be displayed on the specified X workstation. Function sPltPoint() plots a line between two points. The corresponding convenience function for performing 2D plots is sPltPoint2D(). Function sPltPoly() draws a polygon and sPltArc() displays an arc, optionally filling with some color. The function sPltSurface() displays a surface in 3D when provided with an array of points. Function sHistPoint() places a histogram line on the task window, and function sDisplayLevel() displays the specified level of a meter or variable. With the aid of such functions, diverse abstract displays like scatter plots, line plots, pie charts, histograms, x-y-z plots, and level indicators may be shown.

DISplay allows representational displays to be created using networks. Networks can be used to represent displays for a variety of physical systems. In queueing systems graph nodes may represent servers, with arcs representing possible customer routing between servers. In most cases some form of visual representation of the physical object is used for a graph node. The default is to use a rectangular box with a label. It is possible to associate dialogs with nodes and arcs. Clicking on a node or arc with which a dialog is associated will cause the associated dialog to execute at the **DS**. Moreover, the graph can be modified dynamically. Nodes and arcs may be added or deleted. The colors of the nodes and arcs may be changed to signify a change in the value associated with the node or arc.

5 EXAMPLES OF DISplay USE

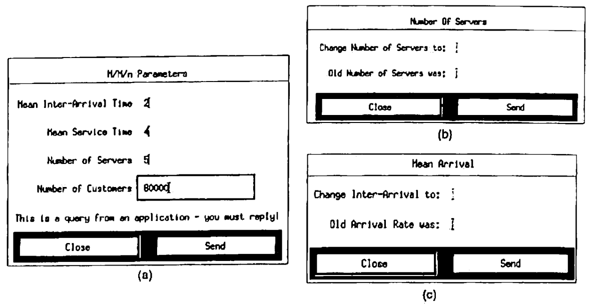
The DISplay system software is useful in a variety of situations including product demonstration, gaming, learning, modeling, performance monitoring, parallel debugging, etc. The use of the **DS** for monitoring of parallel applications in *EcliPSe* is described by Knop el al. (1995). Here we focus on simple examples demonstrating the use of DISplay in general simulations. The first example is that of an `M/M/n` queueing simulation executing sequentially. This example shows how user interaction facilities and basic displays are used. Results are displayed using line plots and histograms. It is instructive to note that the user can dynamically alter simulation variables such as customer arrival rate and/or number of servers, to examine system behavior under such changes.

The second example involves a simple parallel simulation in particle physics, where particles move about randomly within a 2D grid. What is of interest to the analyst is the dynamics of particle movement and final particle positions. We run this model using the *Ariadne* light-weight process library (Mascarenhas and Rego 1995), layered upon the *Conch* distributed computing environment (Topol 1992).

5.1 M/M/n Queueing System Simulation

An M/M/n queue is an n-server queue with Markovian arrivals and services. Customers queue in FIFO mode in a single queue, awaiting service. We fix the mean service time μ and examine the effect of *varying* arrival rate λ and number of servers n on the measured responses of mean queue size, actual queue size and server utilization. The M/M/n application was developed using the process-oriented simulation language CSIM (Schwetman 1992).

In this example application, a user interface was rapidly generated using three dialogs. One was a synchronous, model-prompted dialog, where the application blocks and queries the user to enter input parameters (i.e., λ, μ, n and total number of cus-


M/M/n Parameters
Mean Inter-Arrival Time 2
Mean Service Time 4
Number of Servers 5
Number of Customers 80000
This is a query from an application - you must reply!

```
int sock; /* socket for communication */
int iatm; /* the mean inter-arrival time */

static SdQtnAns dialog_parm[] = {
     {"Mean Inter-Arrival Time", SDFLOAT},
     {"Mean Service Time", SDFLOAT},
     {"Number of Servers", SDINT},
     {"Number of Customers", SDINT},
};   /* parameters of the simulation */

/* dialog identifiers */
static int parm_dialog_id, arr_dialog_id;
int ds_setup() /* demonstrates setup */
{
     ...;
     parm_dialog_id = sdCreateDialogNotask
       ("M/M/n Parameters", SDMODEL_QUERY,
        Number(dialog_parm), dialog_parm, NULL);
     ...;
     sBeginDialog(sock, clock, parm_dialog_id,
        arr_dialog_id, "Change Parameters");
     return(sock);
}

get_parameters(int sock) /* Synch. dialog */
{
     char *reply[4]; /* replies placed here */
     double dbl, timereply;
     int ivl;

     timereply = clock;
     sQueryReply(sock, &timereply,
       parm_dialog_id, reply);    /* query user */
     if ((dbl = atof(reply[0])) > 0.0)
          iatm = dbl; /* Inter-arrival Time */
     if ((dbl = atof(reply[1])) > 0.0)
          svtm = dbl; /* service time */
     ...;
     sStrPrintMsg(sock, clock, "M/M/n parameters");
     sStrPrintMsg(sock, clock, "%f%d%d",svtm,iatm,ns);
}

/* handle mean inter-arrival  */
int handle_arrival_change(char *reply[])
{    /* reply contains user input */
     float new_iatm;
     char **send_reply;

     if (((new_iatm = (float)atof(reply[0])) == 0)
        || (new_iatm == iatm)) return(0);
     wait(event_list_empty);
     send_reply = sdCreateReply(arr_dialog_id);
     sdAddToReply(arr_dialog_id, 0, new_iatm);
     sdAddToReply(arr_dialog_id, 1, iatm);
     iatm = new_iatm;
     sReplyDispMsg(sock, clock, arr_dialog_id,
                   send_reply);/* update screen */
     return(1);
}
```

Figure 4: Dialogs and Code for the M/M/n System

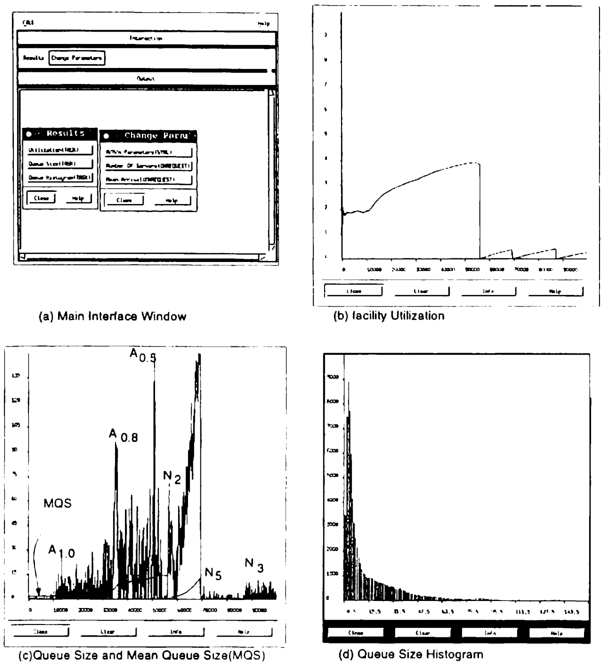


Figure 5: User Interface Window and Results

tomers to be simulated). Once these parameters are initialized, the application continues to execute. The other two dialogs are asynchronous, user-command dialogs. These allow the user to dynamically alter λ and n. New values for these parameters are sent to the application, which echoes back both new and old values to the user interface. Appropriate handlers were provided in the application to act on these parameter changes.

The different dialogs are shown in Figure 4, with the synchronous dialog displayed in Figure 4(a). The code segment shown in Figure 4 shows how this dialog is set up and used. Each entry in the dialog form has a corresponding entry in the `SdQtnAns` structure. The dialog is set up locally using `sdCreateDialog()`, with `SdQtnAns` passed to the latter as a parameter. Dialog initiation at the **DS** is accomplished with `sBeginDialog()`, and invocation via the function `sQueryReply()`.

To display the state of the system three displays are used. The first display depicts the utilization level of the multi-server facility. The second display shows both the mean queue size and the actual queue size on a single graph. The third display shows a queue size histogram, made by updating samples on each customer departure from the facility. Figure 5 contains a user interface window for the queueing application, and also windows which display simulation results dynamically during model execution.

5.2 Particle Dynamics on a 2D Lattice

Particle dynamics studies are of considerable interest in the fields of physics and materials science. Examples where their study is useful to name just a couple, are fluid flows in porous media and electrical conduction. These phenomena are often modeled as *random walks* on disordered clusters. The model described below is in essence the one known as the *ant in the labyrinth* and is due to deGennes (Nakanishi 1993). The *random walker* (ant) moves at random only on certain accessible sites of a lattice, where the *fraction* of accessible sites on the lattice is q, $0 < q < 1$. We simulate the movement of the ant for T time-steps and then compute the mean square displacement of the ant. The goal is to find the relation between this displacement and the values of q and T.

This example shows how the DISplay system can be used with distributed or parallel applications. Here, DISplay was valuable in verifying program correctness through visual examination of program results, and also in monitoring the performance of different algorithms. Initially, a small percentage ($(1-q) = 2\%$) of all grid points are labeled as inaccessible. Particles are then assigned to a maximum of 10% of the remaining lattice, with positions assigned randomly. The application proceeds in a sequence of *time-steps*. All particles are visited in some sequence, and given a chance to move to a randomly chosen neighboring lattice point in a single time-step. For such a move to be successful, two conditions must hold. First, the destination point should be accessible to the particle (i.e., it should not be labeled inaccessible). Second, the point should not already be host to another particle. If either condition is false, the particle remains where it is until it can make another attempt to move in the next phase. Wrap-around is used to handle particles that move across the outer boundary.

As described above, the application is simple to code on a uniprocessor. After creating a grid to represent the lattice and marking sites as inaccessible, a sequence of time-steps ensues. All necessary information is available to the single host processor locally. Porting the application to a multiprocessor or distributed system is not difficult, but requires some care. A good way to parallelize the application is to place portions of the lattice on distinct processors, so that work can be shared. The lattice is divided into slices, with each slice (and thus all particles on the slice) assigned to a distinct processor. Processors work on their slices independently, marking sites as inaccessible and generating particles. Each processor need only be aware of the status of the slice that it has been assigned. Because each processor does not have all the information it needs for independent simulation (since events in a time-step may involve particles at other processors, and particle movement

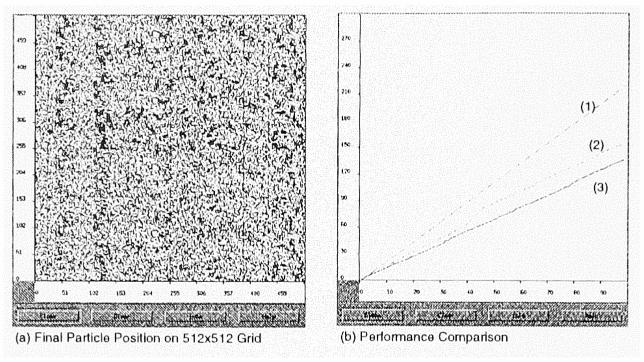

(1)
(2)
(3)
(a) Final Particle Position on 512x512 Grid
(b) Performance Comparison

Figure 6: Particle Simulation Grid and Performance

between slices or across boundaries requires between-processor information), some form of processor synchronization is required. This synchronization must occur after every time-step so that particles that migrate from one processor to another (across slices or across the lattice boundary) do not arrive at a destination processor *late* in simulation time. Note that particles which migrate only to find destination sites already occupied must migrate back to their original positions on source processors. In this application, we test three different synchronization mechanisms, obtaining performance displays for each on a single graph.

5.2.1 Methods of Synchronization

The first two methods of synchronization involve simple *centralized* mechanisms. A master process synchronizes computing done by slave processes. The master process also sends timing information to the **DS** at each time-step. In the first method, each particle (on a slave) sends a message to the master process after it has been considered for movement. Particles that must migrate send a message only after they have arrived at a final destination location. When all particles have been considered, each slave awaits a signal from the master. The master sends each slave a signal only when it has received completion messages from all particles. If the number of slave processes is p and the total number of particles is n, a total of $(n+p)$ synchronization messages are sent at the end of each time-step. For a simulation with T time-steps, the total cost in terms of messages is $M = T(n+p)$.

The second synchronization mechanism is a small variant of the first. Instead of forcing each particle to send a message to the master process after the particle has been considered for movement, a single message is sent by each slave process after it has finished processing all its particles in a time-step. Migrating particles send messages to the master process independently. When all particles have been accounted for, the master sends each slave a message initiating the next time-step. If we denote the number of migrations at time step i by m_i, the total number of synchronization messages required in this scheme is $M = T(2n) + \sum(m_i)$.

The third synchronization mechanism is *decentralized*, based on a conservative parallel simulation protocol (Chandy and Misra 1979). The master process does not play any role in synchronization; instead, time-stamps on migrating particles are used for synchronization. If no particles migrate from a processor, the processor sends null-messages to other processors, thus giving them the necessary time-stamp information. In this application, a process interacts with only two other processors– those assigned to neighboring slices of the lattice. Processors in receipt of migrating particles must send acknowledgements to source processors because the underlying communication software does not guarantee delivery within a fixed time. A source process can continue execution only after such an acknowledgement is received. The total number of synchronization messages required in this case is $M_{max} = T(2n) + \sum(m_i)$. The *max* subscript denotes the fact that an explicit synchronization message may not be sent if the last particle at any time-step in a processor migrates.

The distributed application was implemented in C using the *Ariadne* lightweight process library and the Conch communications library. In this example we use a network of five Sun-4 workstations. One of the workstation was made to host a master process (with id `0`) which controlled the execution of four slave processes on distinct workstations: each workstation hosted a single process. Lattice dimensions were set at 512×512, and a total of $p = 256$ particles were used. Each particle was implemented as a lightweight process in *Ariadne*. When particles migrate from one processor to another, the thread representing the particle migrates. The simulation was run for a total of 100 time-steps, with particle-movement graphed dynamically by DISplay. In Figure 6(a) is shown such a DISplay plot of particles, with the dark streaks representing particle paths. The display was implemented using a `Global Plot Points` task. By declaring the task as global when initiated, all processes are able to share this task at the server. At the end of the simulation the starting and the ending positions of each particle, the root mean square displacement and the number of wrap-arounds performed by each particle is printed in the user interface window.

From our simple message-cost analysis, it would appear that synchronization mechanisms two and three are superior to the first mechanism. The graphical results obtained via DISplay indeed show this to be the case. In Figure 6(b) is shown a performance graph for each of the three synchronization mechanisms. Methods two and three have performed equally well, as predicted by our simple analysis. But since method 3 requires only local synchronization (though with almost as many messages as method 2), it has a slight performance advantage.

6 CONCLUSION

The DISplay system is an application-independent tool for effecting Visual-Interaction in sequential or parallel simulation. It provides the model developer with tools for parametric description of a large set of graphical outputs and capabilities for data visualization, inspection and specification. The basic types of simulation output described here are common to most simulations. This realization motivated us to develop a generalized and extensible display capability. We have used DISplay to visualize parallel simulated annealing, numerical analysis algorithms, particle-physics, and computer network simulations.

ACKNOWLEDGMENTS

This research was supported in part by NATO-CRG900108, ONR-9310233, and ARO-93G0045.

REFERENCES

Bell P. C. and R. M. O'Keefe. 1987. Visual Interactive Simulation – History, Recent Develeopments, and Major Issues. *Simulation*, 49(3):109–116.

Chandy K. and J. Misra. 1979. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Trans. on Softw. Eng.*, SE-5(5):440–452.

Fujimoto R. 1990. Parallel Discrete Event Simulation. *CACM*, 33(10):30–53.

Glavach M. and D. Sturrock. 1993. Introduction to SIMAN/Cinema. In *Proceedings of the 1993 Winter Simulation Conference*, 190–192.

Knop F., et al. 1995. Fail-Safe Concurrent Simulation with EcliPSe: An Introduction. *Simulation Practice & Theory (to appear).*

Mascarenhas E. and V. Rego. 1994. DISplay: User's Manual. Technical Report 94-082, Purdue University, Department of Computer Sciences, 1994.

Mascarenhas E. and V. Rego. 1995. Ariadne: Architecture of a Portable Threads System Supporting Thread Migration. *Software-Practice and Experience (to appear).*

Nakanishi H. 1993. Anomalous Diffusion in Disordered Clusters. In *On Clusters and Clustering, From Atoms to Fractals*, ed. P. J. Reynolds, 373–382. Elsevier Science Publishers B.V.

O'Keefe R. M. 1987. What is Visual Interactive Simualtion? (And is There a Methodology for Doing it Right?). In *Proceedings of the 1987 Winter Simulation Conference*, 461–464.

Pritsker, A. A. B. 1986. *Introduction to Simulation and SLAM II.* 3rd ed. New York: Halstead Press.

Rooks M. 1991. A Unified Framework for Visual Interactive Simulation. In *Proceedings of the 1991 Winter Simulation Conference*, 1146–1154.

Russell E. C. 1989. *Building Simulation Models with SIMSCRIPT II.5.* CACI Products Company.

Schwetman H. 1992. *CSIM Users' Guide.* Microelectronics and Computer Technology Corporation.

Thompson W. 1993. A Tutorial for Modelling with the WITNESS Visual Interactive Simulator. In *Proceedings of the 1993 Winter Simulation Conference*, 228–232.

Topol B. 1992. Conch: Second Generation Heterogeneous Computing. Master's thesis, Department of Mathematics and Computer Science, Emory University.

Tung Y.-W. and J. Steinman. 1992. Interactive Graphics for the Parallel and Distributed Computing Simulation. In *Proceedings of the 1992 Winter Simulation Conference*, 695–700.

AUTHOR BIOGRAPHIES

EDWARD MASCARENHAS is a Ph.D. student in Computer Sciences at Purdue University. He received a Masters degree in Industrial Engineering from NITIE (Bombay, India), and a Masters degree in Computer Sciences from Purdue University (West Lafayette) in 1993. His research interests include parallel computation, distributed simulation, and multithreaded programming environments.

VERNON REGO is a Professor of Computer Sciences at Purdue University. He received his M.Sc.(Hons) in Mathematics from B.I.T.S (Pilani, India), and an M.S. and Ph.D. in Computer Science from Michigan State University (East Lansing) in 1985. He was awarded the 1992 IEEE/Gordon Bell Prize in parallel processing research, and is an Editor of *IEEE Transactions on Computers*. His research interests include parallel simulation, parallel processing and software engineering.

JANCHE SANG is an Assistant Professor in Computer and Information Sciences at Cleveland State University. He received the B.S. degree in Computer Science from National Taiwan University (Taipei, Taiwan) in 1984, the M.S. degree in Computer Science from Michigan State University (East Lansing) in 1987, and the Ph.D. degree in Computer Sciences from Purdue University (West Lafayette) in 1994. His research interests include parallel and distributed computing, computer networks, and software engineering. He was awarded the Maurice Halstead Award for software systems research in 1994.