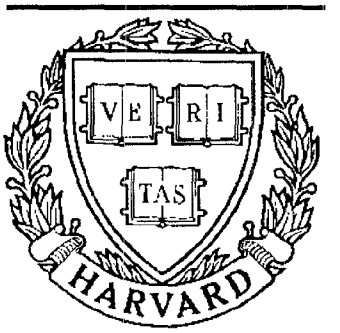
TECHNICAL RESEARCH REPORT

SYSTEMS RESEARCH CENTER



Supported by the National Science Foundation Engineering Research Center Program (NSFD CD 8803012), Industry and the University

Structural Optimization in a Distributed Computing Environment

by B.K. Voon and M.A. Austin

TR 91-104

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
1991		**00-00-1991 to 00-00-1991**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Structural Optimization in a Distributed Computing Environment	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Maryland,Systems Research Center,College Park,MD,20742**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**108**	

Structural Optimization in a Distributed Computing Environment[1]

By B.K. Voon[2], and M.A. Austin, A.M. ASCE[3]

ABSTRACT

This report presents the formulation and testing of a Feasible Sequential Quadratic Programming (FSQP-DIS) optimization algorithm customized to a Distributed Numerical Computing environment (DNC). DNC utilizes networking technology and an ensemble of loosely coupled processors to compute structural analyses concurrently. Each iterate of the FSQP-DIS is partitioned for concurrent computations in the direction calculation, and the steplength calculation. The prototype environment is tested on three applications; a mathematical programming problem, the design of a two-story planar steel frame, and finally, the optimal design of a two-story three-dimensional steel frame.

[1] This research was supported by the National Science Foundation's Initiation Grant NSF BCS 8907722, by the NSF Engineering Research Centers Program: NSFD CDR 8803012, and by the AFSOR University Research Initiative Program under grant AFSOR-90-0105.

[2] Graduate Research Assistant, Department of Civil Engineering and Systems Research Center, University of Maryland, College Park, MD 20742, USA.

[3] Assistant Professor, Department of Civil Engineering and Systems Research Center, University of Maryland, College Park, MD 20742, USA.

TABLE OF CONTENTS

CHAPTER
ONE

Structural Optimization

1.1 Optimization-Based Structural Design

Now that engineering workstations with network connectivity are readily available in the marketplace, opportunities exist for the formulation of new algorithms and software tools that exploit concurrency as a means of increasing computational speed. The strong need for this research dates back to the early 1980's when considerable work was done to better represent real world design problems, and to capitalize on the emergence of engineering workstations. At U.C. Berkeley, for example, Nye et al. [29] proposed an optimization algorithm called the Phase I-II-III Method of Feasible Directions. This algorithm has been successfully applied to a wide variety of engineering problems including the design of chemical polymers [12], integrated circuits [28] and earthquake resistant buildings [5, 4]. For structural engineering problems of a realistic size, however, the quality of user interaction has often been very poor, with time consuming structural analyses - and hence iterations of optimization - severely restricting the size of the problems studied. Together these limitations have not

only limited the appeal of the Berkeley work to others, but also restrained the scope of problems that could be practically investigated.

Researchers at the Systems Research Center, University of Maryland, are attempting to mitigate this problem by focusing their work in two areas. First, a new class of Feasible Sequential Quadratic Programming (FSQP) optimization algorithms has been formulated [41]. These algorithms have superlinear convergence properties, and therefore require fewer iterations to converge than the Phase I-II-III Method of Feasible Directions [41, 40]. Still, this leaves the problem of having to compute the behavior of engineering systems. Our experience indicates that for many optimization problems, more than 90% of the computational effort is dedicated to the calculation of engineering system behavior. Indeed, upwards of 98-99% of the total computational effort is consumed by structural/finite element analyses during the optimal design of earthquake resistant structures [5, 4, 7, 8]. Consequently, any efforts to speed up the engineering analyses will also improve user interaction and reduce design turn-around time.

1.2 Parallel versus Distributed Computing

Two approaches for increasing computational speed are parallel computing and distributed computing. Parallel computing systems consist of several processors that are located within a small distance of each other; their main purpose is joint execution of a computational task. Often, individual processors are designed with this task in mind. Communication among processors is re-

liable and predictable. By contrast, distributed computing systems are loosely coupled. Couloris et al. [15] point out that a number of factors have led to the emergence of loosely coupled systems. They include:

1. A significant reduction in price/performance ratio of VLSI hardware.

2. Highspeed network technologies are now readily available.

3. Interactive service of large centralized computer services is often poor quality with long unpredictable response times. Restricted user interfaces lead to difficulties in customizing hardware and software to a users specific needs.

Not only is it possible for individual processors to be far apart, but the topology of processors in the network may change (addition or subtraction) during the execution of a task. In this respect, communication among processors in distributed computing is much less reliable than in parallel computing [9].

Several issues need to be considered in deciding whether parallel or distributed computing (or combinations thereof) is the best approach to increasing computational speed for the problem being studied. From a financial viewpoint, many engineering companies do not have the resources to buy parallel machines, but can justify the purchase of engineering workstation clusters connected by LAN networking. For example, Pratt and Whitney has recently decided to decentralize its computing resources; instead of emphasizing expansion of their main frame computers, they are moving to purchase 1200 general purpose engineering work stations.

The second issue is "whether or not computational algorithms can be configured to exploit the combined processing power of multiple workstations in a distributed computing environment?" In this study we are interested in increasing computational speed in optimization-based structural design. As already mentioned, the vast majority of computational effort is dedicated to finite element analyses, which are repeated many times. Since our general-purpose structural analysis/finite element packages are implemented on engineering workstations, a good first step is to increase computational speed by identifying locations for potential concurrency in the optimization algorithm, and developing a distributed computing environment for running structural analyses concurrently. This is coarse grained parallelism. It is worth noting that the short history of distributed computing includes applications to number of engineering and numerical analysis problem domains: (a) the asynchronous solution of large sets of numerical equations [9], (b) finite element modeling (and sensitivity analysis) of a large swept wing [14], (c) operations on very large matrices [32], (d) integration of structural dynamics equations [3], and (e) solutions to the traveling salesman problem [26].

Optimization-based structural design on a parallel machine is dismissed at this time because it requires fine grained implementations of both the optimization algorithm, and the finite element method. While considerable work has been done on the formulation of data structures and algorithms for parallel implementations of the finite element method - see, for example, Farhat [17],

Herendeen [20] and Johnsson [22] - this work is still under development.

1.3 Objectives and Scope

The long-term objectives of this research are to formulate algorithms and develop computer software that allows engineers to study problems in the optimal design of large flexible aerospace structures, earthquake resistant structures, and highway bridge structures. Indeed, it is envisioned that when implementations of the finite element method are readily available on massively parallel machines, significant improvements in performance will be possible by writing computational environments that exploit combinations of distributed and massively parallel computing (10^5 or more processors) resources working in tandem [34].

As a starting point, this research has focussed on the formulation and writing of software components to setup, execute, and monitor numerical computations running concurrently on groups of 5 to 10 engineering workstations. This means that increases in speed of less than 10 will be achievable.

The purposes of this report are four-fold. First, the ideas leading to the implementation of the Distributed Numerical Computing Environment are motivated and explained. A new version of the Feasible Sequential Quadratic Programming (FSQP) optimization algorithm that matches the distributed computing architecture is formulated. Numerical experiments are conducted on a small mathematical programming problem, and two structural optimization

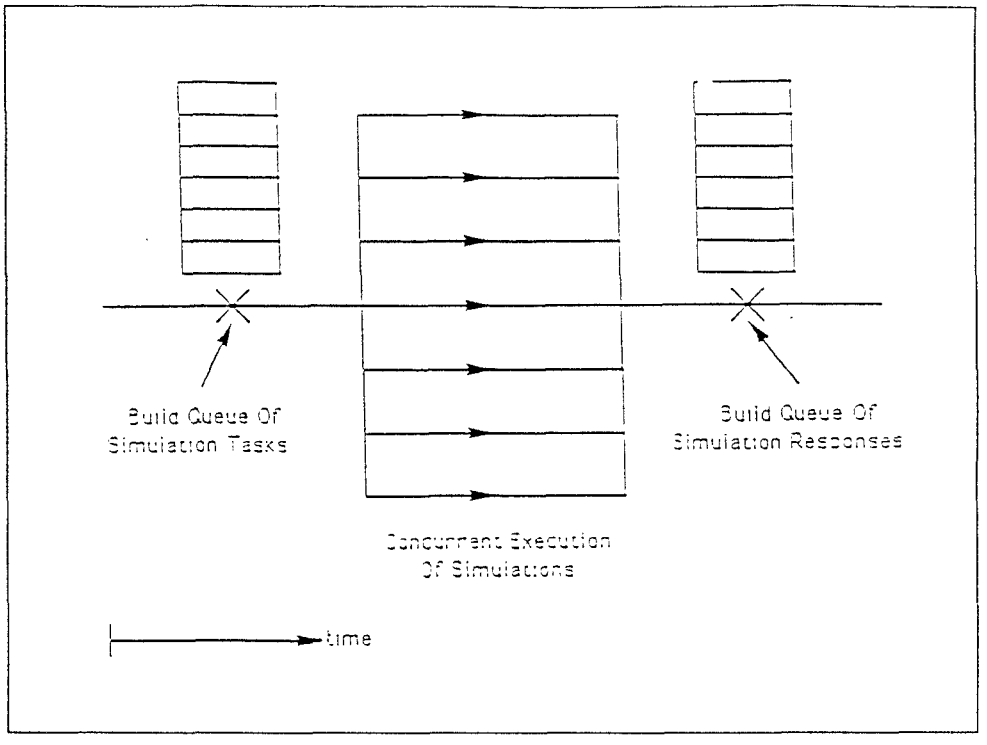


Figure 1.1: Structure of Algorithms

problems. Finally, the performance of the prototype system is critically assessed. Recommendations for future work include ways to improve the performance of structural optimization algorithms within the DNC environment, as well as long-term strategies for obtaining factors of computational speedup in excess of 10.

1.4 Overview of Algorithm Structure

Our research direction stems from the simple observation that optimization and numerical algorithms frequently require many simulations that differ only slightly in their input data. Indeed, Figure 2.1 shows that the same algorithms often contain critical points that cannot be passed until a complete block of

simulations is finished. Since the quantity of input information that distinguishes simulations is minimal, a practical way of increasing computational speed is to setup multiple simulator packages distributed over a workstation network and concurrently compute individual components of the simulation block.

We start each simulation block by building a FIFO (first-in first-out) queue of initial condition data for the simulation components. Packets of simulation requests are then distributed to remote simulators via the network. When a remote simulation finishes, the essential features of the system response/behavior are sent back to the algorithm and temporarily stored in a response queue. The block of simulations is complete when the length of the simulation response queue equals the initial length of the simulation task queue.

1.5 Reading Level

Readers are assumed to be familiar with the C programming language, the UNIX operating system, data structures, and basic computer terminology such as workstation, mouse, window, menu, and keyboard.

CHAPTER
TWO

Distributed Numerical Computing Environment

2.1 Introduction

This chapter describes the prototype implementation of the Distributed Numerical Computing (DNC) environment developed as part of this work. The development goal of DNC is to provide designers with easy-to-implement software tools to setup, execute, and monitor concurrent computations for numerical analysis, optimization, and engineering analysis problems. Concurrency is achieved by distributing tasks over a network of loosely coupled autonomous engineering workstations. In 1990 DNC was used to solve the equations of motion for smooth dynamical systems [6]. This report describes a second application area, the formulation and testing of algorithms for optimization-based structural design.

For historical purposes, we note that the model of loosely coupled processors dates back to 1981 (at least) [25]. Distributed systems are now developed with a very wide range of applications in mind; two implementations that are similar to DNC are SUN's Remote Procedure Call (RPC) [23], and ISIS, a toolkit for

dynamic and fault tolerant distributed computing [10].

The discussion of DNC is divided into ten sections. Sections 2.2 and 2.3 describe the architecture of the DNC environment and background information on InterProcess Communication. The DNC graphical user interface, process manager, and remote engineering simulators are described in Sections 2.4 - 2.6, respectively. Section 2.7 describes how to setting up the socket-based interprocess communication (IPC) in the DNC architecture. Section 2.8 discusses how a message or a piece of data can be sent across the DNC network. The mechanisms DNC employs for synchronizing events are outlined in Section 2.9. Finally, Section 2.10 describes the procedure for setting up the DNC architecture.

2.2 Architecture of DNC Environment

The network topology of the DNC environment is shown in Figure 2.1. Its main components are: (a) A graphical user interface, (b) A process manager, and (c) Several remote engineering simulators. The vehicle for this work is the workstation model consisting of a high resolution bit mapped screen, a multiwindow user interface model with mouse and keyboard input, multitasking, and network connectivity. All components are written in C programming language [24] running under UNIX 4.3BSD [23]. Each component of DNC environment executes on a separate SUN SPARC station.

The C programming language was used for this implementation because of the ease with which complex data structures may be defined and manipulated,

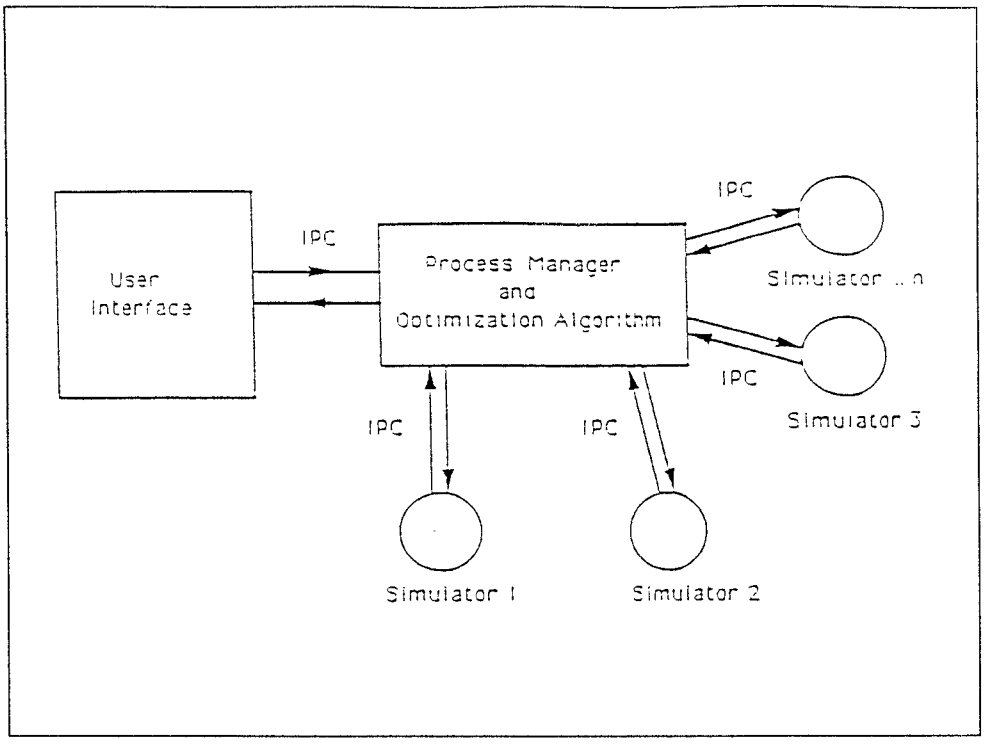


Figure 2.1: Architecture of DNC

and because it allowed for the use of the SunView [38] libraries in the development of graphical user interface.

2.3 Background to Socket-Based Interprocess Communication

At the Systems Research Center (SRC), University of Maryland, the SUN SPARC stations use an operating system called SUNOS. This operating system provides all of the socket-based interprocess communications mechanisms available in versions 4.2BSD and 4.3BSD of the Berkeley UNIX system. Neighboring engineering workstations at SRC are linked by several Local Area Networks (LANs) using coaxial cable for high-speed communication. Transmissions over

communications lines in the LAN are grouped into message packets constructed and transmitted according to precisely defined rules known as a `protocol`. At SRC, socket communication between computers uses either the Internet Transmission Control Protocol (TCP) or the Internet User Datagram Protocol (UDP). The TCP and UDP protocols belong to a family called Internet family (INET).

The basic building block for unrelated process to communicate with each other on different machines, possible running different operating systems is the socket. A socket is a software abstraction for a communication device which create an endpoint for communication; in other words, it is a reference point to which to which message may be sent or received; When a socket is created, a protocol must be specified for the semantics of communication. In this project a `SOCK STREAM` type was used. The stream socket provides sequenced, reliable, two-way connection based byte streams. A stream socket must be in a connected stated before any data may be sent or received on it. This can be done with a `connect()` call. The INET communication protocols used here to implement the `SOCK STREAM` sockets insure that data is not lost or duplicated. For further information, the interested reader is referred to Coulouris [15] and Stevens [36].

2.3.1 IPC Library

DNC uses an InterProcess Commumications library developed by Byrne [13]. The library have facilities to automatically setup client/server models, send and receive data across network sockets, and to close down these systems. For a complete listing of C code in this library, see the appendices of reference [13]. In

```
send_structure(sock, ptr, size)
int sock;
char *ptr;
int size;
{
int sent, acc=0, RETRY=FALSE;
char log_buf[132];
static int count=0, retries=0;

    while (acc < size) {
        if((sent = write(sock, (char *) (ptr+acc), size-acc)) < 0) {
            perror("ipc.c: send_structure()");
            exit(EWRITE);
        } else {
            acc += sent;
            if (acc < size) {
                RETRY = TRUE;
                retries++;
            }
        }
    }
}
```

Table 2.1: IPC Library Function : send_structure()

this section, C code is given only for the two functions for (a) sending packets of data across sockets, and (b) detecting the presence of incoming data on a socket. For example, Table 2.1 shows the script for the function `send_structure()`; the command `send_structure(gui_socket, mp, SIZE);` sends `SIZE` bytes of a data structure pointed to by `mp` along socket `gui_socket`. Similarly, the function call `data_present( gui_socket, long (0))` - shown in Table 2.2 - tests to see if new data has arrived on socket `gui_socket` using a timeout 0 milliseconds.

2.4 User Interface

Figure 2.2 is a screendump of the DNC User Interface developed under the SunView Window systems [38]. The user interface supports a wide variety of

```
data_present(sock, time_out)
int sock;
long time_out;
{
fd_set fds;
struct timeval timeout;
short result;

FD_ZERO(&fds);
FD_SET(sock, &fds);

timeout.tv_sec = time_out;
timeout.tv_usec = 0;

    if ((result = select(FD_SETSIZE, &fds, NOFDS, NOFDS,
                         &timeout)) == ERROR) {
        perror("ipc: select()");
        exit(ESELECT);
    }

    return(FD_ISSET(sock, &fds));
}
```

Table 2.2: IPC Library Function : data_present()

design and analysis activities, as explained in the following subsections:

1. `Simulation Subwindows:` An important purpose of the interface is to report on computational activities at the process manager (see Section 2.6) and remote numerical simulators (see Section 2.5). Subwindows dedicated to this task appear down the right hand side of the interface. Each window contains the machine name, job status, plus a slider showing the percentage of work done in each simulator. Even though all of the simulators are SPARC workstations, the time-to-completion of identical tasks may vary due other background jobs competing for resources.

2. `Terminal Emulation (TTY) Window :` The TTY window displays intermediate and final final results of the remote engineering simulations, and

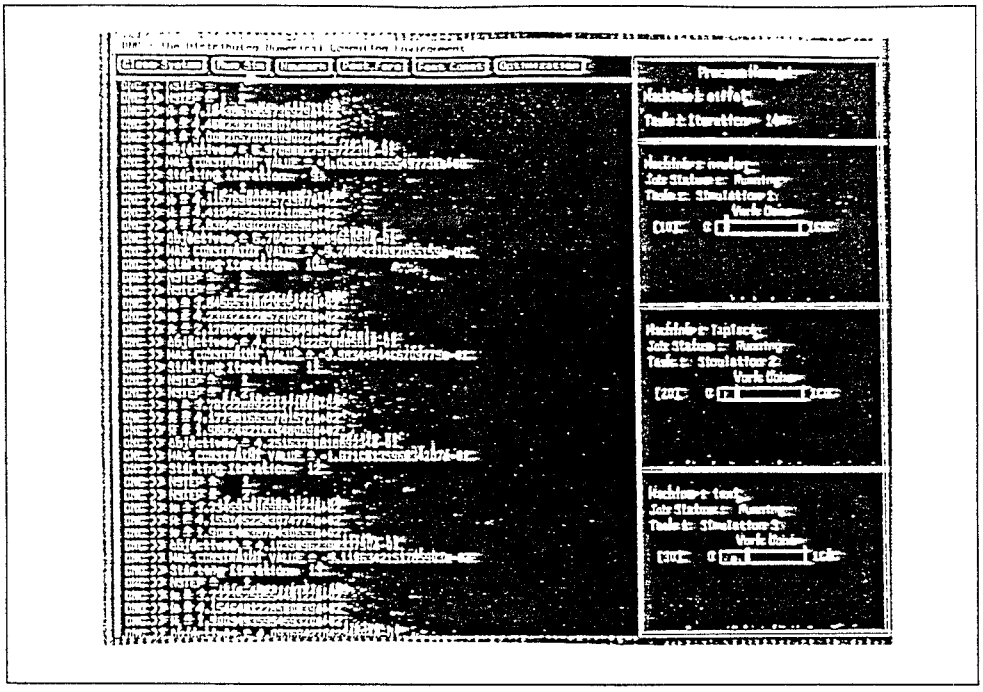


Figure 2.2: DNC User Interface

optimization. In the latter case, for example, this includes the design vector, design objectives, values of design constraints at each iteration. These are done by sending the data or messages via the IPC sockets either from the remote simulators or the process manager. Thus, message sending is an essential activity in the DNC network. Its details will be given in Section 2.8.

3. `Button Command:` Depressing a mouse button triggers a callback function, which in turn sends a message to the process manager telling the manager what job needs to be executed. The DNC User Interface supports mouse button events for: (a) transferring files - simulation datafiles and optimization constraint/objective files - to remote simulators, (b) initiating

the execution of optimization and numerical algorithms, and (c) closing down the DNC environment.

When a post command action includes the mailing of a message to the process manager - possibly requesting a numerical simulation - the user interface should remain unblocked for the processing of further keyboard/mouse events. Unfortunately, standard event-based systems do not behave in this way. A callback function embedded within a standard base window event handler will wait for the arrival of numerical results on an incoming socket before releasing the manager to other tasks. SUN's interposition mechanism [37] overcomes this problem by allowing client programs, such as our user interface, to register event sensitive interposer functions with the window notifier. The subsequent arrival of incoming data on the process manager/user interface socket triggers an interception of window manager control, and a callback to the interposer function. After the interposer function finishes reading the socket, control of events is returned to the base window event handler. Since the interception of window manager control occurs only when data arrives on the socket, users are given the impression that the interface is completely decoupled from the process manager and simulator nodes.

2.5 Remote Simulators

A variety of simulators performing various tasks can be employed in this distributed computing environment. A simulator is a process that read and

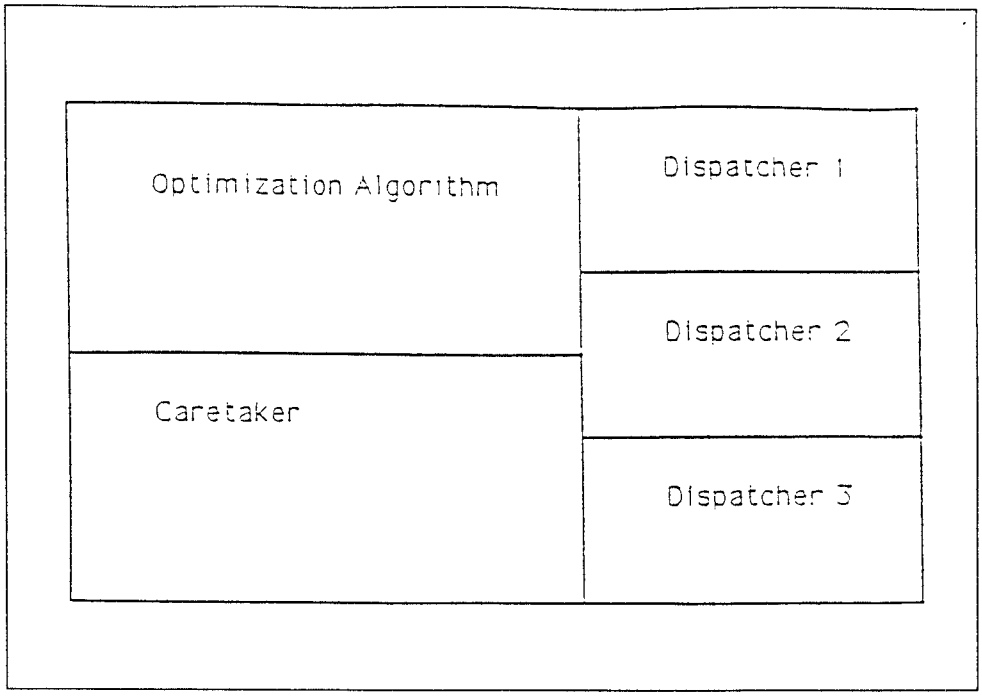


Figure 2.3: Components of Process Manager

analyze a set of input data. It usually checks certain constraint requirements before an output is produced. The remote simulators are full UNIX processes.

2.6 Process Manager

Figure 2.3 shows that the process manager is composed of: (a) Caretaker, (b) Dispatchers, and (c) Numerical and/or optimization algorithms (these are problem dependent). The general purposes of the caretaker and dispatcher threads are to monitor asynchronous events within the DNC environment, schedule concurrent computational activities on the remote engineering simulators, and forward messages to/from the user interface and remote simulators.

2.6.1 Lightweight Processes

When a (parent) process creates another (child) process the parent and child may/may not share some or all of their variables and address space [15]. Processes that share their address space with the parent process, and contain minimal information on the processing state associated with a computation are called `lightweight processes` (lightweight processes are also called `threads`). Multiple threads within a single UNIX task may execute in parallel.

There are several good reasons to implement the DNC process manager as a series of lightweight processes rather than a full UNIX process. They are:

1. Lightweight processes typically operate a single machine, They are efficient because they communicate via shared memory instead of the UNIX filesystem.

2. Implementations of lightweight processes contain the tools to build programming constructs for the synchronization of events within individual threads (the use of monitors allows one lightweight processes temporarily halt the execution of another; see below), to perform I/O, and to respond intelligently to interrupts and runtime exceptions.

3. As pointed out by Martin et al. [27] the use of lightweight processes automatically results in an object-oriented implementation. Associated with each lightweight process object is a well defined set of internal states, operations to change the state, and mechanisms to interact with other

```
main(argc,argv)
int argc;
char *argv[];
{
extern void startup();

    if(argc !=2){
      fprintf(stderr,"usasge: tst #processors\n");
      exit(1);
    }

    THREADgo(atol(argv[1]),2*1024*1024, startup, 0,0,20*1024,2);
}

void
startup()
{

    ..... code deleted ......

    for (ij.i = 0; ij.i<3;ij.i++)
       for (ij.j = 0; ij.j<3;ij.j++)
            THREADcreate(mult, &ij, sizeof(ij), 0, 20*1024,2);

    while(THREADwaitforchild());
}

void
mult()
{
    ..... code deleted ......
}
```

Table 2.3: Simple Example of THREADS Code

objects (see Item 2).

4. Implementations of lightweight process objects are succinct; the dispatcher and caretaker threads described in the following sections are each less than 230 lines of C code !!

While the process manager can be configured to take advantage of items 1 to 4, it is important to remember that processes executed on the remote simulators are asynchronous. The remote simulators do not share memory.

2.6.2 Threads Package from Brown University

Support for the concurrent execution of lightweight processes in the DNC process manager is provided by a Threads package from Brown University [16]. The script of skeleton C code shown in Table 2.3 shows the simplest details of setting up a Threads application. Each thread is written just like a normal C function. However, at run time the Threads package converts each function to a lightweight process.

The entry point for program execution is the `main()` function. The function `Threadgo()` is called to convert the executable program from a full UNIX task to lightweight processes, and to initiate execution of the single thread **`startup`** on machine having `atol(argv[1])` processors. For implementations on the SUN SPARC station, the number of available processors is one. A pool of `2*1024*1024` bytes is allocated to hold the stacks and control blocks for threads. The function `THREADcreate(mult, &ij, sizeof(ij),0,20*1024,2)` creates a new thread of control that executes the thread function `mult` with priority 2, The fourth argument of this function indicates whether or not the parent and child threads should be detached. By setting this argument to zero (i.e. false) the child and parent threads are nondetached. The parent thread executes a call to `THREADwaitforchild()` and will not terminate until the child thread terminates.

```
THREAD_MONITOR prmanager;

void startup()
{
extern void child();

    prmonitor = THREADmonitorinit(0, NULL);

    THREADcreate(child,"prompt1 >> ",0,0,20*1024,2);
    THREADcreate(child,"prompt2 >> ",0,0,20*1024,2);
    THREADcreate(child,"prompt3 >> ",0,0,20*1024,2);
}

void child(prompt)
char *prompt;
{
char buf[80];

    promptandread(prompt,buf,80);
}

void promptandread(prompt,buf,buflen)
char *prompt;
char    *buf;
int   buflen;
{
THREAD_MANAGER_BLOCK manager;

    THREADmonitorentry(prmonitor,&manager);

    ..... code deleted ......

    THREADmonitorexit(prmonitor);
}
```

Table 2.4: Example of THREADS Monitor

2.6.3 Use of Monitors within THREADS package

A monitor is the standard operating systems mechanism for synchronizing events, and providing protection against the incorrect interpretation of shared data due to races in lightweight processes. Once a thread is executing within a monitor, other threads within the same monitor are halted until that monitor is exited.

The script of C code shown in Table 2.4 generates three copies of a thread called `child` and uses a monitor to control the sequencing of prompting (and output) events. First, a call to the function `THREADmonitorinit()` from the parent thread `startup` allocates a monitor and returns a handle for the new monitor of type `THREAD_MONITOR`. Its name is `prmonitor`. In the simplest cases a monitor provides mutually exclusive access to shared data by calling the function `THREADmonitorentry()` to access the data, and `THREADmonitorexit()` to release access control. Once a monitor is entered, other threads can not interrupt the activities inside until the monitor is exited. The variable `manager` stores the address of data structure `THREAD_MANAGER` that is used by the monitor to deal with runtime exceptions.

2.6.4 Optimization/Numerical Algorithm

DNC is setup to solve numerical analysis and optimization problems. The code for these application is written as a C function, but at runtime is converted to a lightweight process. As such, it may interact with the caretaker and

dispatcher threads as described below.

2.6.5 Caretaker

The caretaker thread continuously polls the `gui_socket` socket for the arrival of incoming commands and data from the graphical user interface. Upon request, it creates threads for:

1. Copying and forwarding datafiles to each of the simulators. In the current implementation - described in detail in Chapters 3 and 4 - data files are transferred for the finite element analysis program, and design constraint and design objective routines.

2. Invoking optimization (or numerical analysis threads) as directed, and

3. Closing down the process manager/user interface IPC sockets at the end of optimization process.

2.6.6 Dispatcher Threads

Generally speaking, the dispatcher threads form an intermediate link between the optimization/numerical algorithms, and the remote simulators. One dispatcher thread is created for each remote simulator resource. The specific purposes of each dispatcher are to:

1. Get items from the front of the task queue, and forward them to the remote simulators. Details on building and manipulating the task queue are given in Sections 2.8 and 2.9.

```
typedef enum {
        NOTIFY_TTY              =  1,
        NOTIFY_JOB_STATUS       =  2,
        NOTIFY_TASK             =  3,
        NOTIFY_WORK_DONE        =  4,
        OUTPUT_FILE             =  5,
        QUIT_JOB                =  6,
} MESSAGE_TYPE;

typedef struct message_packet {
        char     source[16];
        MESSAGE_TYPE   type;
        float     work_done;
        char    message[80];
} MESSAGE_PACKET, *MESSAGE_PACKET_PTR;
```

Table 2.5: Data Structure for Message

2. Forward incoming messages from remote simulators onto the graphical user interface.

3. Interact with the optimization (or numerical) algorithm that are waiting for these simulation responses. Incoming simulation responses are stored on a response queue.

2.7 Message Passing Mechanisms and Data Structures

Loosely coupled workstations in the DNC environment communicate via message passing. Message communication occurs when:

1. Remote simulators (or the process manager) want to inform the designer at the user interface on the current stage of activities in DNC.

2. Files need to be transferred across sockets in the DNC network. Notice that we do not assume that file systems are mounted across a LAN.

3. The designer wishes to shut down the DNC system.

Table 2.5 summarizes the data structure that is used to assemble messages. The enumeration type `MESSAGE_TYPE` distinguishes message types. For example, `NOTIFY_TTY` indicates that a message should be displayed on the terminal emulation window of the user interface. When the message type is `NOTIFY_WORK_DONE`, the address of the message is the slider subwindow whose name matches the contents of array `source[16]`. Textual messages are stored in `message[80]`.

2.7.1 Details of Sending and Receiving Messages

The functions `send_structure()` and `data_present()` described in the previous sections form a crucial role in transmitting/receiving messages throughout the DNC network. The features of the DNC message facility are demonstrated by tracking the steps of sending a progress report message from a remote simulator to the user interface.

Table 2.6 is shows code taken from a remote simulator, and demonstrates how a message is assembled and mailed to the user interface. In this particular case, it reports on the percentage of work completed for a particular simulation. When the function is entered, memory is dynamically allocated for the task header and message packet (see Step `[1]`). A message is sent across the simulator-to-process manager socket `sm_socket` in two parts. First, the task header is sent to the process manager indicating that the following block of data will be a message (see Step `[2]`). The contents of the message itself follows; in

```
extern int sm_socket;

{
TASK_PTR          tp;
MESSAGE_PACKET_PTR mp;

  tp = (TASK_PTR)    calloc (1, sizeof(TASK));          /* [1] */
  mp = (MESSAGE_PTR) calloc (1, sizeof(MESSAGE));

  ..... code deleted .......

  tp->datatype  = MESSAGE;                              /* [2] */
  send_structure(sm_socket, tp,   sizeof(TASK));

  mp->type      = NOTIFY_WORK_DONE;                     /* [3] */
  mp->work_done = (float) 100.0*(i/(frame->no_material));
  send_structure(sm_socket, mp, sizeof(MESSAGE_PACKET));

  ..... code deleted .......

}
```

Table 2.6: Building and Sending a Message

this case (see Step [3]) the message type is NOTIFY_WORK_DONE, with the fraction of work completed stored in member mp->work_done.

Table 2.7 summarizes the code needed to detect incoming data on the socket dp->socket. When the function data_present() indicates that new data has arrived, the dispatcher attempts to read and check the successful transmission of two packets of data from dp->socket. The first packet - see Step [5] - is the task header, and indicates the type of packet that follows. Since we are expecting the arrival of a progress report message, the task header will have tp->datatype = MESSAGE. A switch statement separates the different types of message packets. The second message packet is read from the socket dp->socket, and automatically forwarded to the user interface via socket gui_socket; see Step [7].

Table 2.8 shows source code for the function read_input(), which reads incoming data on gui_socket; the functionality is very similar to that of Table

```
extern int gui_socket;

int IPC_Dispatcher(dp)
DISPATCHER_PTR dp;
{

   ..... code deleted .......

   if(data_present(dp->socket, (long) )){                          /* [4] */

   nbytes = read(dp->socket, tp1, sizeof(TASK));                   /* [5] */
   if(nbytes != sizeof(TASK)) {
      printf("error: bytes lost in TASK transfer !!\n");
      exit (1);
   }

   switch(tp1->datatype) {                                         /* [6] */
       case MESSAGE:
            nbytes = read(dp->socket, mp, sizeof(MESSAGE_PACKET));
            if(nbytes != sizeof(MESSAGE_PACKET)) {
               printf("error: bytes lost in MESSAGE transfer !!\n");
               exit (1);
            }
            else  {
               tp1->datatype = MESSAGE;                            /* [7] */
               send_structure(gui_socket, tp1, sizeof(TASK));
               strcpy(mp->source, dp->sim_machine);
               send_structure(gui_socket,  mp, sizeof(MESSAGE_PACKET));
            }
            break;

            ..... code deleted .......

   }
   }
}
```

Table 2.7: Receiving a message at Dispatcher

2.7. The main difference occurs after a message is read. A switch statement with parameter `mp->type` triggers a callback to the function `Handle_Work_Done()`; see Step [8]. The latter function updates the slider in the graphical user interface.

2.8 Building Queues of Tasks and Simulation Responses

As mentioned in Chapter 1, DNC was developed to provide engineers with computational tools to compute engineering simulations concurrently. A typical

```
static Notify_value
read_input()
{
    if(data_present(gui_socket, (long) 0)) {
        switch(tp->datatype) {

            ........ code deleted .......

            case MESSAGE:
                 mp = (MESSAGE_PACKET_PTR)
                       calloc(1, sizeof(MESSAGE_PACKET));
                 nbytes = read(gui_socket, mp, sizeof(MESSAGE_PACKET));

                 if(nbytes != sizeof(MESSAGE_PACKET)) {
                    printf("error: bytes lost in MESSAGE transfer\n");
                    exit (1);
                 }
                 else {
                    switch(mp->type) {                      /* [8] */

                         ........ code deleted .......

                         case NOTIFY_WORK_DONE:
                              Handle_Work_Done(mp);
                              break;
                         default:
                              break;
                    }
                 }
                 break;

        }
    }
}
```

Table 2.8: Receiving a Message at User Interface

DNC architecture contains NO_RESOURCES simulation resources. If a particular component of an algorithm requires N similar simulation tasks that can be computed concurrently, then the first step is to build a queue of simulation tasks. Table 2.9 summarizes the data structure for assembling queues of tasks, and queues of simulation response results. The script:

```
Queue_Task_Init();
for(jh = 1; jh <= NO_RESOURCES; jh++) {

    tp = (TASK_PTR) calloc(1,sizeof(TASK));
    tp->task_no     = (int) jh;
    tp->datatype    = INITIAL;
```

```
typedef enum {
        COMMAND_ONLY      =  1,
        MESSAGE           =  2,
        INITIAL           =  3,
        RESPONSE          =  4,
        DATAFILE          =  5,
        STEPLENGTH        =  6,
        INITIAL_RUN       =  7
} DATA_TYPE;

typedef struct task {
        int              task_no;
        COMMAND_TYPE commandtype;
        DATA_TYPE       datatype;
        union {
              MESSAGE_PACKET_PTR          mp;
              INT_TO_MAN_INITIAL_PTR     imip;
              MAN_TO_SIM_INITIAL_PTR     msip;
              MAN_TO_SIM_RESPONSE_PTR    msrp;
              SIM_TO_MAN_RESPONSE_PTR    smrp;
              MAN_TO_INT_RESPONSE_PTR    mirp;
        } u;
} TASK, *TASK_PTR;

typedef struct tqueue {
        TASK_PTR         tp;
        struct tqueue *next;
} QUEUE_TASK, *QUEUE_TASK_PTR;

typedef struct rqueue {
        SIM_TO_MAN_RESPONSE_PTR  smrp;
        struct rqueue *next;
} QUEUE_RESP, *QUEUE_RESP_PTR;
```

Table 2.9: Data Structure for Queue of Tasks

```
    tp->commandtype = SIMULATION_INIT;

    tp->u.msip = (MAN_TO_SIM_INITIAL_PTR) calloc(1,sizeof(MAN_TO_SIM_INITIAL));
    for(ij = 1; ij <= NDOF; ij++)
        for(ik = 1; ik <= NDOF; ik++) {
            tp->u.msip->mass[ij-1][ik-1]  =  mass[ij-1][ik-1];
            tp->u.msip->stiff[ij-1][ik-1] = stiff[ij-1][ik-1];
        }

    Queue_Task_Add(tp);
}
```

demonstrates how a task queue of length `NO_RESOURCES` could be assembled with initial data for remote simulations. The queue is initialized by calling `Queue_Task_Init()`. Task items are composed of 2 packets of bytes. The first

packet is simply the task queue header. It has `sizeof(TASK)` bytes and is appended to the end of the task queue with the function function `Queue_Task_Add()`. The second packet of information is accessed via the `union` in the `task` data structure; the specific details of the union contents depend on both the location of the queue in the DNC environment, plus details of the application at hand. For example, the name `MESSAGE_PACKET_PTR` points to data structure containing a basic message, as already shown in Table 2.5. In the abovementioned script of code, the acronym `INT_TO_MAN_INITIAL_PTR` is a pointer to the data structure:

```
typedef struct interface_manager {
        int           task_no;
        double       accel[3];
        double    velocity[3];
        double       displ[3];
        double        load[3];
} INT_TO_MAN_INITIAL, *INT_TO_MAN_INITIAL_PTR;
```

which contains information on system response and external loading. It is sent from the graphical user interface to the manager. In Table 2.9, the entities `QUEUE_TASK_PTR` and `QUEUE_RESP_PTR` are pointers to items in the task and response queues, respectively.

2.9 Synchronization of States within Process Manager

In Section 2.6, monitors have been used to provide mutual exclusion to shared data and input-output. Sometimes the need arises to modify data - or execute code - that depends on whether or not certain conditions are true. In such a scenario, a thread would continue execution only it a condition is true. Otherwise, it would suspend itself using the function call `THREADmonitorwait()`.

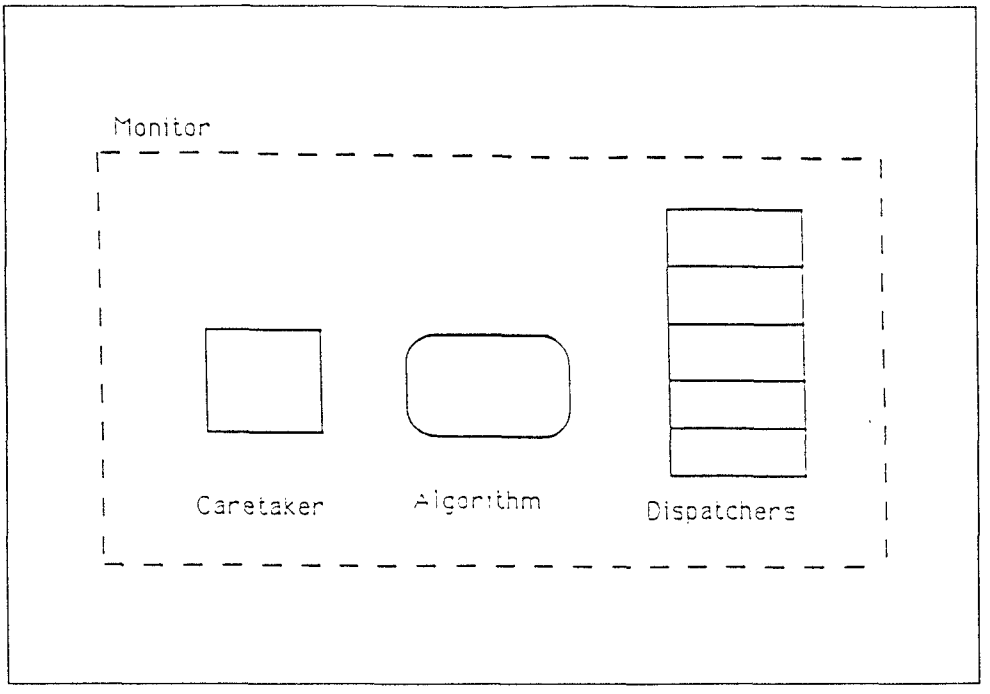


Figure 2.4: Scope of Monitor in Process Manager

After another thread has satisfied the condition, it may wake up the first thread as it exits the monitor by calling `THREADmonitorsignalandexit()`. Finally, a thread may wish to signal another thread before suspending itself. This is accomplished by calling the function `THREADmonitorsignalandwait()`.

2.9.1 Scope of Monitor in Process Manager

The process manager employs a single monitor to synchronize activities inside its thread components. Figure 2.4 shows a schematic of the components, together with a dashed line for the the scope of the monitor; the monitor is known to all of the dispatchers, the caretaker, and the optimization (or numerical analysis) threads.

Recall that the purpose of the monitor is to ensure that only one thread

execute within the monitor at any point in time. The remaining threads `wait` in process queues, and will not execute until certain conditions to become true. DNC uses one queue for the dispatcher threads - called `DISPATCHER_QUEUE` - and a second queue for suspended caretaker and optimization threads. The latter queue is called `MANAGER_QUEUE`. The length of the `DISPATCHER_QUEUE` equals the number of the remote simulators used, and is assembled during the DNC startup procedure. For details, see Section 2.10.

2.9.2 Interplay of Process Manager Threads

The interaction of caretaker, optimization, and dispatcher threads is demonstrated by tracking the state of process activities and queues, and queues of task and simulation response data during the initial stages of assembling and distributing a queue of simulation tasks to remote simulators. Tables 2.10 and 2.11 contain the relevant sections of code in the optimization algorithm and dispatcher threads, respectively.

Let's begin in `fera_algo_proc()`, a thread for building a queue of simulation tasks. When the optimization algorithm enters the monitor, (Step `[1]` in Table 2.10) the monitor queue is empty. Each of the dispatcher threads is suspended at Step [6] of Table 2.11, and waiting on the `DISPATCHER_QUEUE` as shown in the top box of Figure 2.5. A queue containing `n` simulation tasks is built at Step `[2]` of Table 2.10, and stored in the task queue as shown in the lowest box of Figure 2.5. When this is complete, the process manager has process and queue states as shown in Figure 2.5. Notice that both the process

```
int
fera_algo_proc(carep)
MANAGER_PTR carep;
{
THREAD_MANAGER_BLOCK manager;
MESSAGE_PACKET_PTR    mp,mp1;
TASK_PTR tp, tp1;
int i, ntasks, stepno;
SIM_TO_MAN_RESPONSE_PTR smrp;

  THREADmonitorentry(carep->cp->mon, &manager);              /* [1] */

  mp  = (MESSAGE_PACKET_PTR) calloc(1, sizeof(MESSAGE_PACKET));
  tp1 = (TASK_PTR) calloc(1, sizeof(TASK));

  Queue_Task_Init();
  for(i=1; i <= n; i++) {                                    /* [2] */
      tp = (TASK_PTR) calloc(1,sizeof(TASK));

      ..... code deleted .....

      Queue_Task_Add(tp);
  }

  Tasks_Completed = 0;
  ntasks = Queue_Task_GetLength();                           /* [3] */
  while(Tsks_Completed < ntasks)                             /* [4] */
    THREADmonitorsignalandwait(carep->cp->mon,
                    DISPATCHER_QUEUE, MANAGER_QUEUE);

      ..... optimization code deleted .....                 /* [5] */

  THREADmonitorexit(carep->cp->mon);
}
```

Table 2.10: Scheduling Events : Optimization Algorithm Thread

and task queues are ordered, with m and n items, respectively.

Interaction between the optimization and dispatcher queues begins at Step [3] of Table 2.10. Tasks_Completed is a global variable that indicates how many of the simulation tasks have completed; that is, data has been sent to a remote simulator, the simulation computed, and the system response information returned to the appropriate dispatcher. The variable ntasks is the initial length of the task queue. When the set of statements at Step [4] is first approached, the optimization thread signals to the front item on the dispatcher

```
int IPC_Dispatcher(dp)
DISPATCHER_PTR dp;
{
  THREADmonitorentry(dp->cp->mon, &manager);

  ..... code deleted ........

  THREADmonitorwait(dp->cp->mon, &manager);              /* [6] */

  /* Process Events : Signal MANAGER_QUEUE : Wait on DISPATCHER_QUEUE */

  dp->job_status = FINISHED;                             /* [7] */
  while(Continue_Simulation == TRUE) {
      if(dp->job_status == FINISHED) {

         if(Queue_Task_GetLength() > 0) {
            tp = Queue_Task_GetItem();
            send_structure(dp->socket, tp, sizeof(TASK));
            send_structure(dp->socket, tp->u.msip,
                           sizeof(MAN_TO_SIM_INITIAL));  /* [8] */

            ...... code deleted ......

         }
      }
      else {
         if(data_present(dp->socket, (long) 0)) {        /* [9] */

            ...... code deleted ......

         }
      }

      THREADmonitorsignalandwait(dp->cp->mon,
            MANAGER_QUEUE, DISPATCHER_QUEUE);            /* [10] */
  }

  THREADmonitorexit(dp->cp->mon, MANAGER_QUEUE);
}
```

Table 2.11: Scheduling Events : Dispatcher Component

queue - shown by a dashed line - and suspends itself on the MANAGER_QUEUE. This sequence of process activities is shown in Table 2.6.

Activity in the process manager immediately jumps to Step [7] in Table 2.11. The dispatcher thread verifies that items still remain on the task queue - via function Queue_Task_GetLength() - and then grabs the front item in the task queue with the function Queue_Task_GetItem(). Details of the simulation

task are sent to the remote simulator as described in Section 2.7.1. Now the task queue has `n-1` items, and a program state as shown in Figure 2.7. Notice that Step `[9]` is for reading incoming data; it operates as described in Section 2.7.1. At Step `[10]` of the code, the dispatcher thread signals the `MANAGER_QUEUE` and suspends itself at the back of `DISPATCHER_QUEUE`. The state of processes and queues is as shown in Figure 2.8.

The program activity returns to Step `[3]` of Table 2.10 and tests to see if the number of tasks completed equals the initial length the task queue. In this case it isn't, so the optimization thread once again signals to the thread at the front of the dispatcher queue, and places itself on the `MANAGER_QUEUE`. The dispatcher thread `Disp 2` fetches a simulation task from the task queue, sends it to the second remote simulator, signals to `MANAGER_QUEUE`, and suspends itself on the end of the dispatcher queue. The process of control switching between the optimization and dispatcher threads continues until all the simulation tasks have been sent to remote simulators, and the queue of simulation responses equals the initial length of the task queue as shown in event state Figure 2.9. When this condition is eventually satisfied, the program drops to Step `[5]` in Table 2.11.

Notice that even though the simulation tasks are guaranteed to be sent out in order, significant variations in computational speeds of the remote simulators may result in a mixed ordering of quantities in the response queue. This is demonstrated by placing `Resp 2` before `Resp 1` in Table 2.11.

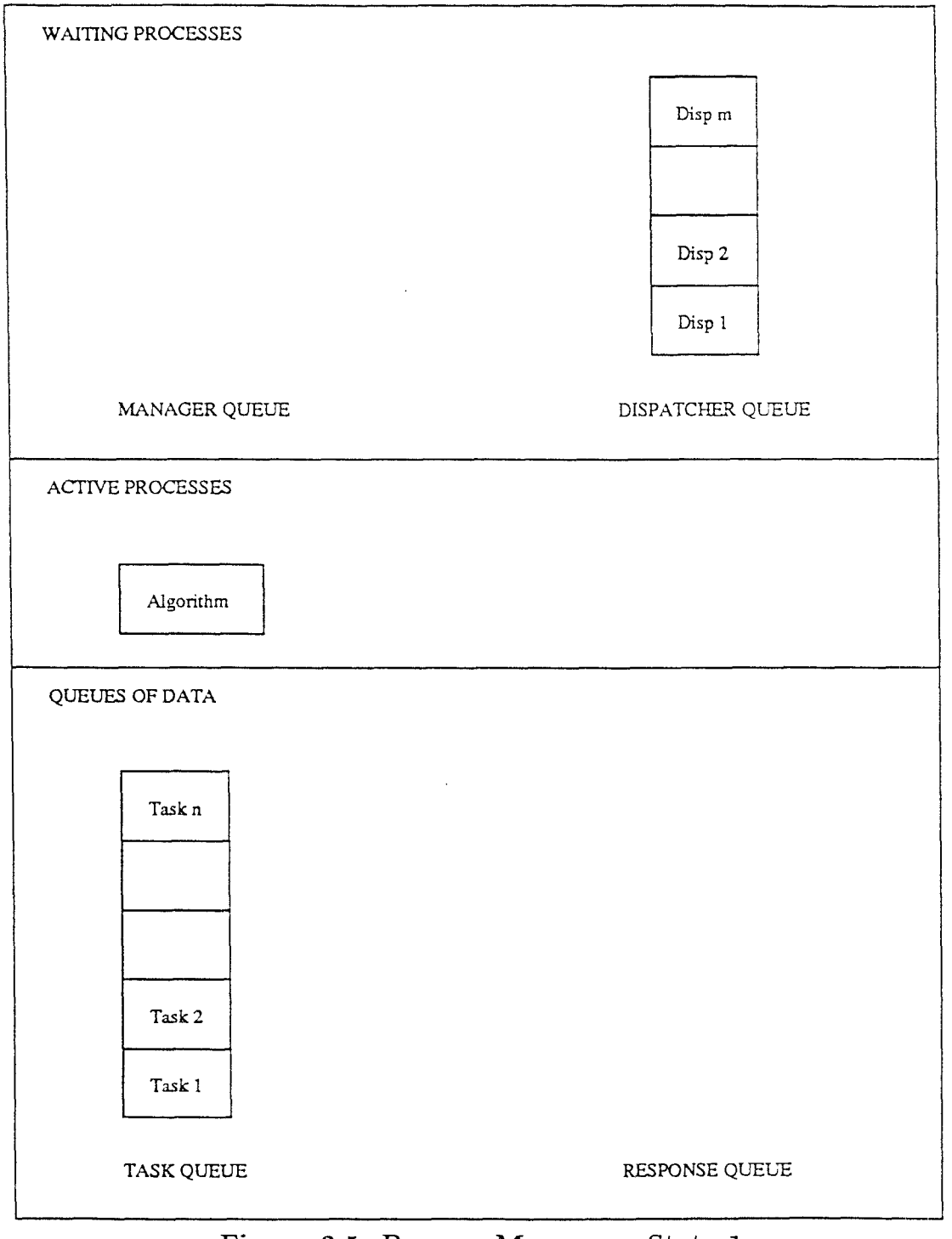


Figure 2.5: Process Manager : State 1

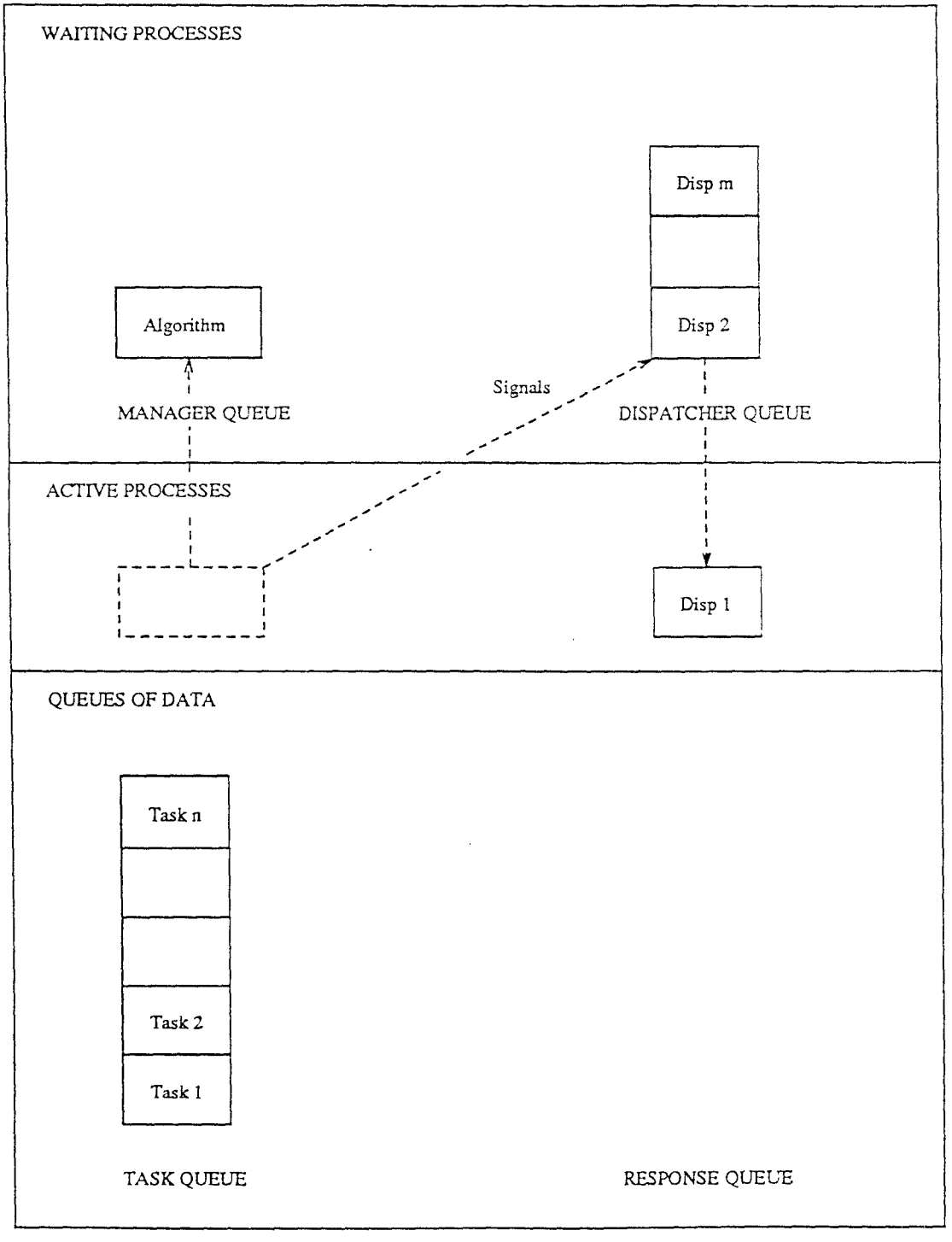


Figure 2.6: Process Manager : State 2

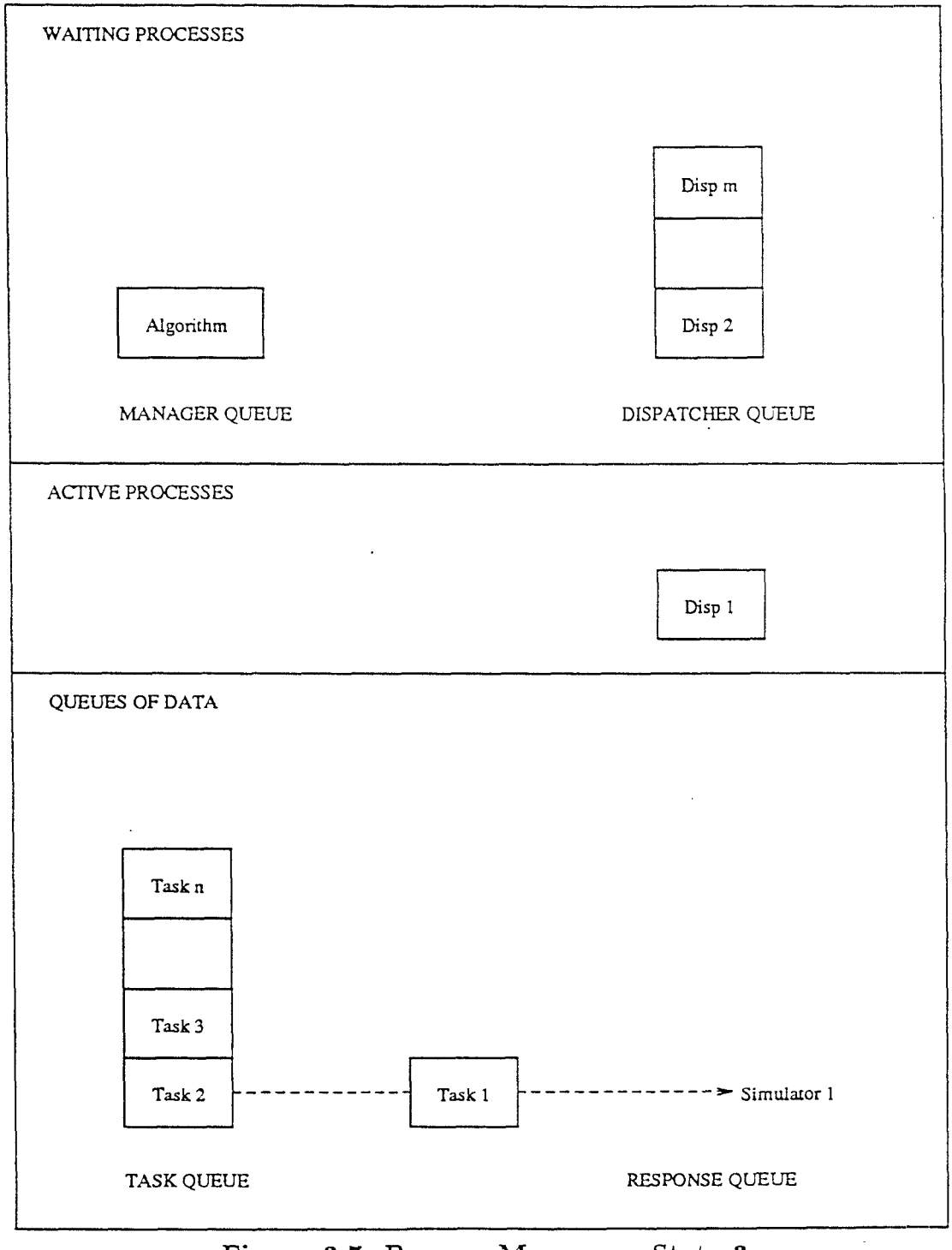


Figure 2.7: Process Manager : State 3

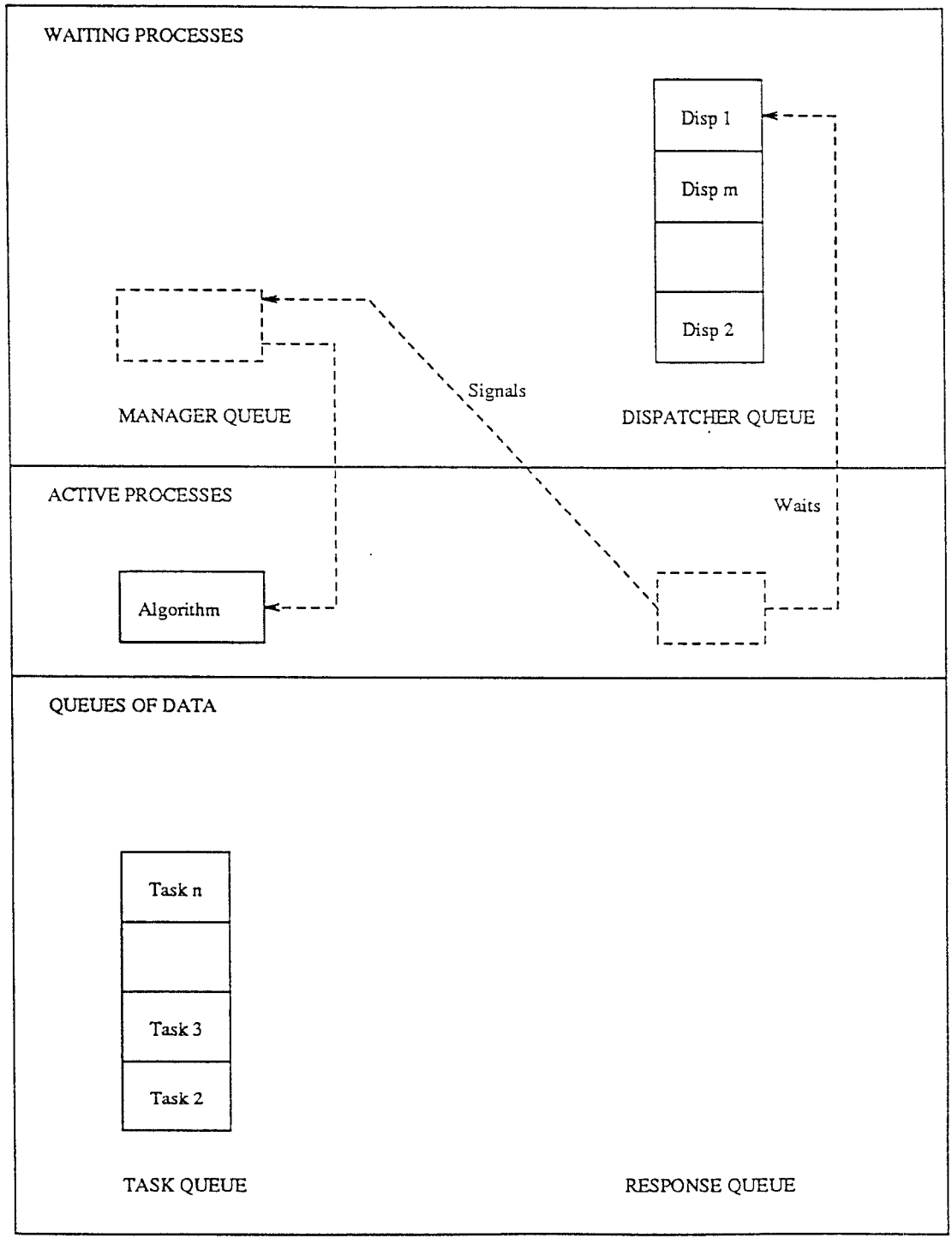


Figure 2.8: Process Manager : State 4

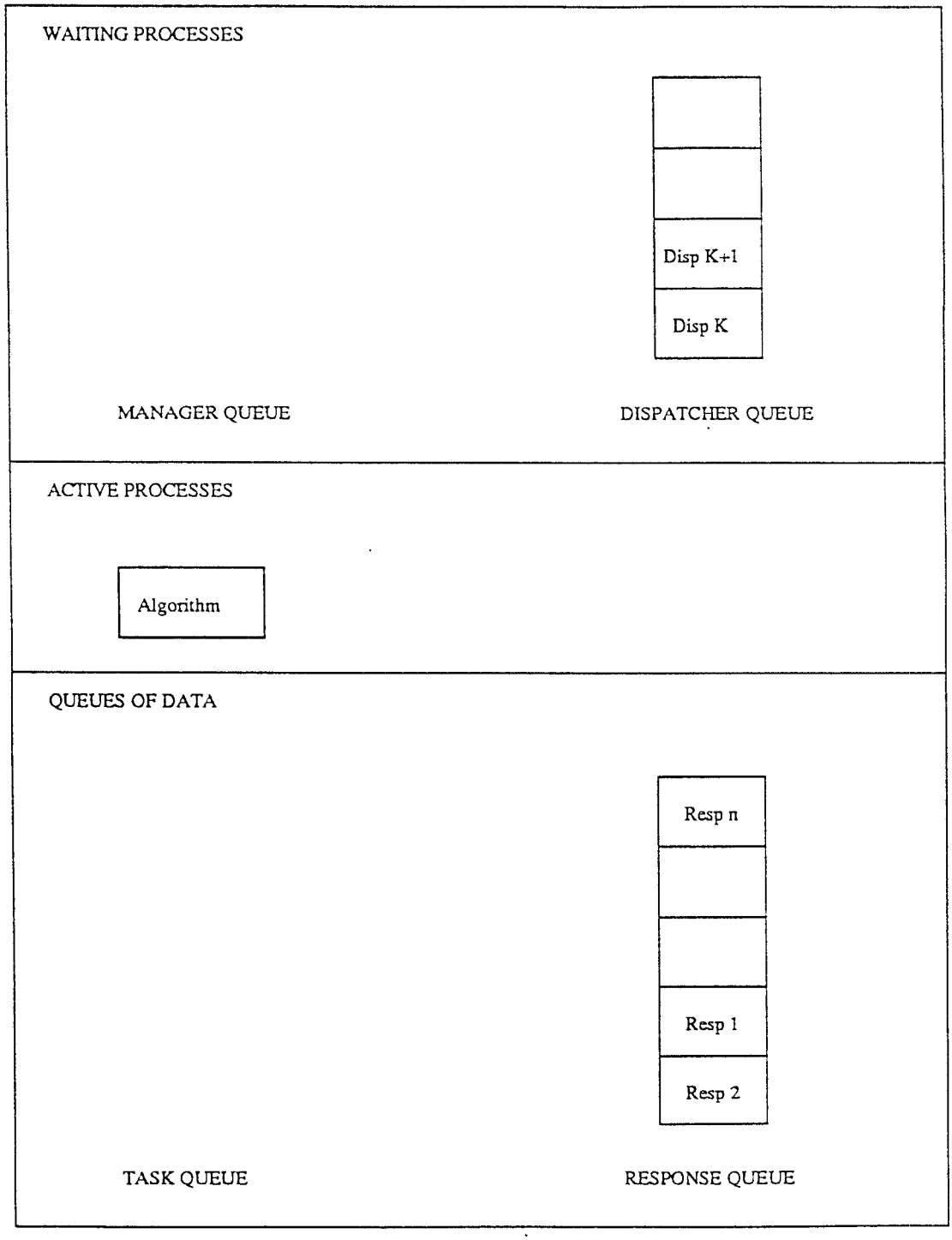


Figure 2.9: Process Manager : State 5

2.10 Setting Up the DNC Architecture

All the machines employed in this research consisted of SUN SPARC workstations. This section describes the step-by-step procedure for setting up the DNC architecture shown in Figure 2.10.

1. `Execution of remote simulators:` One window is created for each of the remote simulators. Then remote login to the simulators and manually execute the command *simulator* on every window of the simulator. Each simulator creates a stream socket (see Section 2.5.1 for the notion of a *socket*), and bounds a name to it. The simulators wait for incoming connection requests from the process manager.

2. `Setup the User Interface - Part 1:` The user interface process is started. First, a socket is created, bound to a name, and put in a listening state for a connection request from the process manager Second, a remote shell command - `rsh` - is made to start the manager on the process manager workstation.

3. `Process Manager:` Execution of the process manager is initiated by an incoming `rsh` command from the user interface machine. As has been mentioned in Section 2.5, the process manager is composed of three type of threads: (a) Caretaker, (b) Dispatchers, and (c) Numerical / Optimization algorithms. The dispatcher and caretaker threads are responsible for making IPC connections to the simulators and user interface, respectively,

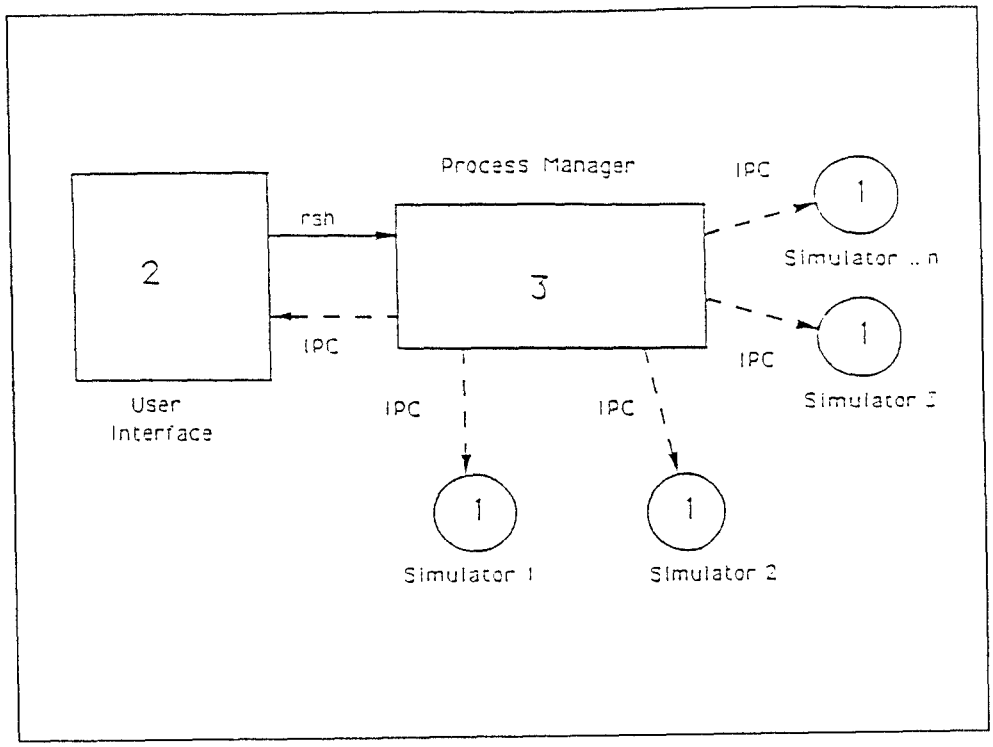


Figure 2.10: Procedure for Assembly IPC sockets in DNC

and as shown in Figure 2.11.

(a) `Startup Thread:` The `startup` thread creates a monitor for the process manager by calling `THREADmonitorinit()`.

(b) `Dispatcher Threads:` One dispatcher thread is created for each remote simulator. Immediately after the dispatcher thread is created, a socket connection is established the the appropriate simulator. The socket name is `ms_socket`. Each dispatcher then suspends itself on the dispatcher queue, and waits to be reactivated by a companion process within the monitor.

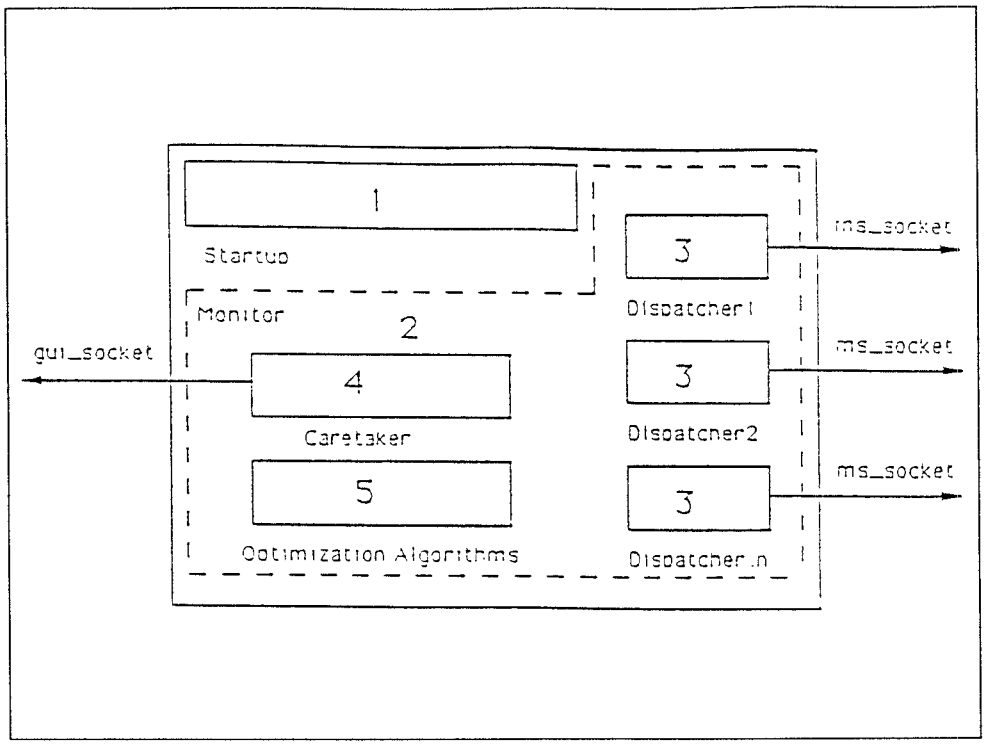


Figure 2.11: Threads for DNC Architecture

(c) `Caretaker Thread:` The caretaker creates a socket called `gui_socket` for connection to the graphical user interface. After a connection has been successfully established - see Step [2] above - then the caretaker continuously polls for incoming data on `gui_socket`.

4. `User Interface - Part 2:` Successful connections cause the Graphical User Interface as shown in Figure 2.2 to appear.

5. `Numerical and Optimization Algorithms:` Algorithms are activated by the caretaker as a thread within the process manager monitor. Details of a Fast Sequential Quadratic Programming (FSQP) optimization algorithm are given in Chapter 3.

CHAPTER

THREE

Distributed Computing Version of FSQP Optimization Algorithms

3.1 Introduction

The main contribution of this research is the formulation of a Feasible Sequential Quadratic Programming (FSQP) optimization algorithm which takes advantage of the DNC architecture described in Chapter 2. The family of FSQP optimization algorithms are based on a Sequential Quadratic Programming (SQP) iteration, modified so as to generate feasible iterates. They are described and analyzed in references [11, 30], and were selected for this study because they are readily available at the Systems Research Center.

The remainder of this chapter is divided into two sections. Section 3.2 gives an introduction to Version 2.0 the sequential FSQP optimization algorithm; please note that large portions of this section have been taken (more or less) verbatim from references [40] and [41]. Section 3.3 explains how the distributed computing version of FSQP has been formulated.

3.2 FSQP Version 2.0 Optimization Algorithms

Version 2.0 of FSQP tackles optimization problems of the form

$$(P) \qquad \min \max_{i \in I^f} \{f_i(x)\} \quad \text{s.t.} \quad x \in X$$

where $I^f = \{1, \cdots, n_f\}$ and X is the set of point $x \in \mathbb{R}^n$ satisfying

$$bl \leq x \leq bu$$

$$g_j(x) \leq 0, \quad j = 1, \cdots, n_i$$

$$g_j(x) \equiv \langle c_{j-n_i}, x \rangle - d_{j-n_i} \leq 0, \quad j = n_i + 1, \cdots, t_i$$

$$\langle a_j, x \rangle = b_j \quad j = 1, \cdots, \ell_e$$

Here the parameters $bl \in \mathbb{R}^n$ and $bu \in \mathbb{R}^n$ are lower and upper box constraints for components in the design vector x. The coefficients $a_j \in \mathbb{R}^n$ and $b_j \in \mathbb{R}$, $j = 1, \cdots, \ell_e$ describe linear equality constraints. The functions $g_j : \mathbb{R}^n \rightarrow \mathbb{R}$, $j = 1, \cdots, n_i$ with $c_j \in \mathbb{R}^n$, $d_j \in \mathbb{R}$, $j = 1, \cdots, t_i - n_i$ represent smooth inequality (possibly nonlinear) design constraints. The merit function for the optimization is the maximum value of multiple design objectives $f_i : \mathbb{R}^n \rightarrow \mathbb{R}$, $i = 1, \cdots, n_f$. If the initial design vector x provided by the user results in an infeasible design, FSQP first generates a feasible point by iterating on the problem of minimizing the maximum of constraints. All subsequent iterates generated by FSQP will be feasible.

The scope of this study is limited to an Armijo line search strategy that gives a monotone decrease in the maximum of the objective functions at each iteration [30]. The monotone line search is composed of two parts; if nonlinear

constraints are present, then the SQP direction is first `tilted` to yield a feasible direction. It is then `bent` to ensure that a step of one is accepted when the design vector is close to a solution. The latter is a requirement for superlinear convergence. This algorithm will be simply called FSQP-AL (for convenience).

3.3 Details of FSQP-AL Algorithm

Given a feasible iterate x, the basic SQP direction d^0 is first computed by solving a standard quadratic program using a positive definite estimate H of the Hessian of the Lagrangian. d^0 is a direction of descent for the objective function; it is almost feasible in the sense that it is at worst tangent to the feasible set if there are nonlinear constraints and it is feasible otherwise. An essentially arbitrary feasible descent direction $d^1 = d^1(x)$ is then computed. Then for a certain scalar $\rho = \rho(x) \in [0,1]$, a feasible descent direction $d = (1-\rho)d^0 + \rho d^1$ is obtained, asymptotically close to d^0. Finally a second order correction $\tilde{d} = \tilde{d}(x, d, H)$ is computed, involving auxiliary function evaluations at $x+d$, and an Armijo type search is performed along the arc $x + td + t^2\tilde{d}$. The purpose of $\tilde{d}$ is to allow a full step of one to be taken close to a solution, thus allowing superlinear convergence to take place. Conditions are given in [30] on $d^1(\cdot)$, $\rho(\cdot)$ and $\tilde{d}(\cdot,\cdot)$ that result in a globally convergent, locally superlinear convergent algorithm.

Nomenclature : For notational convenience let:

$$f'(x,d) = \max_{i \in I^f} \{f_i(x) + \langle \nabla f_i(x), d \rangle\} - f_{I^f}(x)$$

and for any subset $I \subset I^f$ (defined below),

$$\tilde{f}_I'(x+d, x, \tilde{d}) = \max_{i \in I}\{f_i(x+d) + \langle \nabla f_i(x), \tilde{d}\rangle\} - f_I(x+d).$$

Algorithm FSQP-AL.

Parameters. $\eta = 0.1$, $\nu = 0.01$, $\alpha = 0.1$, $\beta = 0.5$, $\kappa = 2.1$, $\tau_1 = \tau_2 = 2.5$.

`Data :` $x_0 \in \mathbb{R}^n$, $\varepsilon > 0$.

`Step 0: Initialization.` Set $k = 0$ and $H_0 =$ the identity matrix. If x_0 is infeasible, substitute a feasible point, obtained as discussed below.

`Step 1: Computation of a search arc.`

i. Compute d_k^0, the solution of the quadratic program $QP(x_k, H_k)$. Compute the Kuhn-Tucker vector

$$\nabla L(x_k, \zeta_k, \xi_k, \lambda_k, \mu_k) = \sum_{j=1}^{n_f} \zeta_{k,j} \nabla f_j(x_k) + \sum_{j=1}^{n} \xi_{k,j} + \sum_{j=1}^{t_i} \lambda_{k,j} \nabla g_j(x_k) + \sum_{j=1}^{\ell_e} \mu_{k,j} a_j.$$

If $\|\nabla L(x_k, \zeta_k, \xi_k, \lambda_k, \mu_k)\| \leq \varepsilon$, stop. If $n_i = 0$ and $n_f = 1$, set $d_k = d_k^0$ and $\tilde{d}_k = 0$ and go to `Step 2`. If $n_i = 0$ and $n_f > 1$, set $d_k = d_k^0$ and go to `Step 1 iv`. Here $\zeta_{k,j}$'s with $\sum_{j=1}^{n_f} \zeta_{k,j} = 1$, $\xi_{k,j}$'s, $\lambda_{k,j}$'s, and $\mu_{k,j}$'s denotes the multipliers, for the various objective functions, simple bounds (only n possible active bounds at each iteration), inequality, bounds (only n possible active bounds at each iteration), inequality, and equality constraints respectively, associated with this quadratic program. The set of active objective functions, for any i such that $\zeta_{k,i} > 0$, by

$$I_k^f(d_k) = \{j \in I^f : |f_j(x_k) - f_i(x_k)| \leq 0.2\|d_k\| \cdot \|\nabla f_j(x_k) - \nabla f_i(x_k)\|\}$$

$$\cup\{j \in I^f : \zeta_{k,j} > 0\}$$

and the set of active constraints by

$$I_k^g(d_k) = \{j \in \{1, \cdots, t_i\} : |g_j(x_k)| \leq 0.2\|d_k\| \cdot \|\nabla g_j(x_k)\|\}$$

$$\cup\{j \in \{1, \cdots, t_i\} : \lambda_{k,j} > 0\}.$$

ii. Compute d_k^1 by solving the strictly convex quadratic program

$$\begin{aligned} \min_{d^1 \in \mathbb{R}^n, \gamma \in \mathbb{R}} \quad & \tfrac{\eta}{2}\langle d_k^0 - d^1, d_k^0 - d^1\rangle + \gamma \\ \text{s.t.} \quad & bl \leq x_k + d^1 \leq bu \\ & f'(x_k, d^1) \leq \gamma \\ & g_j(x_k) + \langle \nabla g_j(x_k), d^1\rangle \leq \gamma, \quad j = 1, \cdots, n_i \\ & \langle c_j, x_k + d^1\rangle \leq d_j, \quad j = 1, \cdots, t_i - n_i \\ & \langle a_j, x_k + d^1\rangle = b_j, \quad j = 1, \cdots, \ell_e \end{aligned}$$

iii. Set $d_k = (1 - \rho_k)d_k^0 + \rho_k d_k^1$ with $\rho_k = \|d_k^0\|^\kappa/(\|d_k^0\|^\kappa + v_k)$, where $v_k = \max(0.5,\ \|d_k^1\|^{\tau_1})$.

iv. Compute $\tilde{d}_k$ by solving the strictly convex quadratic program

$$\begin{aligned}
\min_{\tilde{d}\in\mathbb{R}^n} \quad & \tfrac{1}{2}\langle (d_k+\tilde{d}), H_k(d_k+\tilde{d})\rangle + f'_{I_k^f(d_k)}(x_k, d_k, \tilde{d}) \\
\text{s.t.} \quad & bl \le x_k + d_k + \tilde{d} \le bu \\
& g_j(x_k+d_k) + \langle \nabla g_j(x_k), \tilde{d}\rangle \le -\min(\nu\|d_k\|,\ \|d_k\|^{\tau_2}), \\
& \qquad\qquad j \in I_k^g(d_k^g) \cap \{j : j \le n_i\} \\
& \langle c_{j-n_i}, x_k + d_k + \tilde{d}\rangle \le d_{j-n_i}, \quad j \in I_k^g(d_k) \cap \{j : j > n_i\} \\
& \langle a_j, x_k + d_k + \tilde{d}\rangle = b_j, \quad j = 1, \cdots, \ell_e
\end{aligned}$$

where $f'_{I_k^f(d_k)}(x_k, d_k, \tilde{d}) = f'(x_k, d_k + \tilde{d})$ if $n_f = 1$, and $f'_{I_k^f(d_k)}(x_k, d_k, \tilde{d}) = \tilde{f}'_{I_k^f(d_k)}(x_k + d_k, x_k, \tilde{d})$ if $n_f > 1$. If the quadratic program has no solution or if $\|\tilde{d}_k\| > \|d_k\|$, set $\tilde{d}_k = 0$.

`Step 2 : Line Search :` The line search proceeds as follows. Set $\delta_k = f'(x_k, d_k)$ if $n_i \neq 0$ and $\delta_k = -\langle d_k, H_k d_k\rangle$ otherwise. Compute t_k, the first number t in the sequence $\{1, \beta, \beta^2, \cdots\}$ satisfying

$$f(x_k + td_k + t^2\tilde{d}_k) \le f(x_k) + \alpha t\delta_k$$

$$g_j(x_k + td_k + t^2\tilde{d}_k) \le 0, \quad j = 1, \cdots, n_i$$

$$\langle c_{j-n_i}, x_k + td_k + t^2\tilde{d}_k\rangle \le d_{j-n_i}, \quad \forall j > n_i \ \&\ j \notin I_k^g(d_k).$$

First, the linear constraints that were not used in computing $\tilde{d}_k$ are checked until all of them are satisfied, resulting in a stepsize, say, $\bar{t}_k$. Due to the convexity of linear constraints, these constraints will be satisfied for any $t \le \bar{t}_k$. Then, for $t = \bar{t}_k$, nonlinear constraints are checked first and, for both objectives and constraints, those with nonzero multipliers in the QP yielding d_k^0 are evaluated first. For $t < \bar{t}_k$, the function that caused the previous value of t to be rejected

is checked first; all functions of the same type ("objective" or "constraint") as the latter will then be checked first.

`Step 3. Update Hessian :` Compute a new approximation H_{k+1} to the Hessian of the Lagrangian using the BFGS formula with Powell's modification [31]. In this approach, given x_k, x_{k+1} and H_k, define the variable increment by

$$\zeta_k = x_{k+1} - x_k$$

and the gradient increment of the Lagrange function by

$$\eta_k = \nabla_x L(x_{k+1}, \mu_k, \lambda_k) - \nabla_x L(x_k, \mu_k, \lambda_k). \tag{3.3.1}$$

The new H_{k+1} is then obtained by

$$H_{k+1} = H_k - \frac{H_k \zeta_k (H_k \zeta_k)^T}{\langle \zeta_k, H_k \zeta_k \rangle} + \frac{\eta_k \eta_k^T}{\langle \eta_k, \zeta_k \rangle}. \tag{3.3.2}$$

This formula has the nice feature that if H_k is positive definite (the smallest eigenvalue of H_k is positive) and if $\langle \eta_k, \zeta_k \rangle > 0$, then H_{k+1} remains positive definite.

`Step 4. Update Iterate.`

- Set $x_{k+1} = x_k + t_k d_k + t_k^2 \tilde{d}_k$.
- Increase k by 1.
- Go back to `Step 1`.

The sequence of x_k converges to a stationary point. It has the descent property of being at least a local minimum. For details on the properties of theoretical convergence, see Panier [30].

3.4 Distributed Computing Version of FSQP

Zhou and Tits [41, 40] have implemented Version 2.0 of FSQP-AL as a set of FORTRAN subroutines. The designer provides application dependent subroutines for the design objective and constraint functions. Subroutines to compute the gradients of these functions may also be supplied; otherwise, FSQP-AL estimate gradients via forward finite differences.

The distributed version of FSQP-AL is called FSQP-DIS (for convenience). Before work on FSQP-DIS could proceed, the sequential version of FSQP had to be converted from FORTRAN to C. A C implementation of FSQP-AL was needed so that tools from the Threads Package (for mutual exclusion of shared data and monitors) could be built into FSQP-DIS. With a little help from the `f2c` program at AT&T Bell Laboratories [18], it is amazing that only a few hours was needed to make the FORTRAN to C language conversion, and to verify that both implementations would give identical numerical results on the test problems reported by Zhou [41].

Each iterate of FSQP-DIS is dominated by three computational tasks: (1) computation of the terms $\nabla f_i(x)$ and $\nabla g_i(x)$ in the Jacobian matrix, (2) a single analysis at $(x + d)$ for the computation of $\tilde{d}$ (this is needed to achieve super-linear convergence) and (3) the step length computation. FSQP-DIS employs concurrent computations at Steps 1 and 3 of each iterate.

3.4.1 Step 1 : Computation of Jacobian Matrix

The first major task in each iteration of optimization is computation of terms in the Jacobian matrix

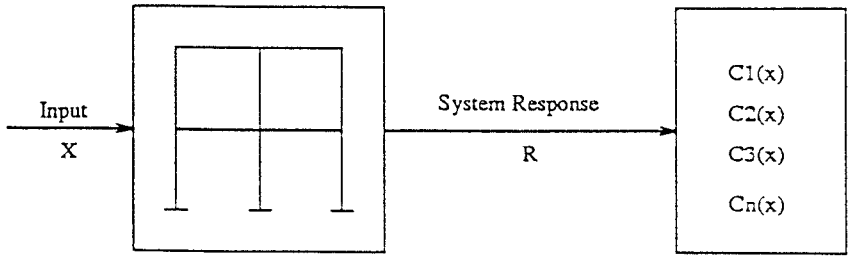
$$[J] = \begin{bmatrix} \frac{\partial D_1}{\partial x_1} & \frac{\partial D_1}{\partial x_2} & \cdots & \frac{\partial D_1}{\partial x_n} \\ \frac{\partial D_2}{\partial x_1} & \frac{\partial D_2}{\partial x_2} & \cdots & \frac{\partial D_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial D_m}{\partial x_1} & \frac{\partial D_m}{\partial x_2} & \cdots & \frac{\partial D_m}{\partial x_n} \end{bmatrix}. \tag{3.4.1}$$

Here $[\partial D_i/\partial x_j]$ is the partial derivative of design performance for the i^{th} specification - constraint $g_i(x)$ or objective $f_i(x)$ - with respect to the j^{th} component of the design vector x. Because analytic expressions for $[\partial D_i/\partial x_j]$ are usually unobtainable, terms in the Jacobian matrix are often approximated by the finite difference scheme

$$\frac{\partial D_i}{\partial x_j} = \left[\frac{D(x+dx_j)-D(x)}{dx_j} \right]. \tag{3.4.2}$$

If the performance specifications are coded as individual functions in fortran subroutines (as described in paragraph 1 of Section 3.4), then there is no computational advantage in computing terms in the Jacobian matrix row-wise versus column-wise. This is true for gradient computations via finite differences and analytical formulae.

Next we note that hundreds (sometimes thousands) of individual specifications are often needed to adequately assess the performance of a large engineering system.



The schematic indicates that specifications $g_i(x)$ and $f_i(x)$ cannot be evaluated until a finite element (or similar) computation is complete; the exception is box constraints $bl \leq x \leq bu$, which can be evaluated at without an engineering analysis. Once the output from these computations is available, it usually contains enough information for all of the constraints and objectives to be evaluated. More important, the computational effort needed to evaluate thousands of specification formulae may be insignificant compared to the computational work required to solve large sets of linear/nonlinear equations for the system response in the first place (e.g. finite element models of structures, SPICE for circuits). This means that the optimization algorithm should be organized to pass through the left hand box of the schematic a minimum number of times. If x contains n components, then a minimum of n simulations must be computed for the gradient approximation via finite differences. Given that the number

of available remote simulators m may be much less that n, by far the simplest implementation is to compute terms in the Jacobian matrix columnwise.

FSQP-DIS computes components of the perturbed design - $D(x + x_j)$ - concurrently, and then estimates entire columns of the Jacobian matrix using the forward finite difference approximation. For this component of the optimization, the result is an algorithm whose computational time is linearly dependent on the number of design parameters, but almost independent of the number of constraints.

Details of Implementation: Figure 3.1 is a script of code that shows the details of building a queue of simulation tasks for the perturbed designs needed for gradients in the Jacobian matrix. The labeled sections are as noted:

1. `Gradient Thread` : A lightweight process (see Section 2.6.1) is dedicated to the column-wise computation of Jacobian matrix components via forward finite differences.

2. `Enter monitor` : This thread enters the process manager monitor, thereby ensuring other threads cannot affect the assembly of the simulation tasks.

3. `Build queue of Tasks` : The function `Queue_Task_Init()` initializes the queue of simulation tasks. Perturbed components of the design vector $x_i + \Delta x_i$ are then stored in a queue of tasks by calling `Queue_Task_Add()`. In addition to $x_i + \Delta x_i$, each task contains the message to run the simulation, i.e., `FSQPD_GENERAL`. The length of the queue equals the number of design parameters.

```
TMP_PTR
nngrfd(carep,tmp,nparam,rteps,udelta,x,cas)          /* [1] */
MANAGER_PTR carep;
TMP_PTR tmp;
int *nparam,cas;
double *rteps,*udelta,*x;
{
THREAD_MANAGER_BLOCK manager;
MESSAGE_PACKET_PTR mp;
double d1,d2,d3,d4,gnew,delta,xj;
int i,j,k,ntasks;
TASK_PTR tp,tp1;

  THREADmonitorentry(carep->cp->mon, &manager);      /* [2] */

  Queue_Task_Init();

  for(i = 1; i <= *nparam; i++) {
      xj = x[i];
      d3 = 1., d4 = fabs(xj);
      d1 = *udelta, d2 = *rteps * max(d3,d4);
      delta = max(d1,d2);
      if(xj < 0.)
         delta = -delta;

      tmp->delta_x[i] = delta;
      x[i] = xj + delta;                             /* [3] */

      tp = (TASK_PTR)calloc(1,sizeof(TASK));
      tp->task_no     = (int) i;
      tp->datatype    = RESPONSE;
      tp->commandtype = FSQPD_GENERAL;
      tp->u.msrp = (MAN_TO_SIM_RESPONSE_PTR)
           calloc(1,sizeof(MAN_TO_SIM_RESPONSE));
      tp->u.msrp->delta = delta;

      for(j = 1; j <= NPARAM; j++)
          tp->u.msrp->x[j] = x[j];

      Queue_Task_Add(tp);
      x[i] = xj;
  }

  Queue_Resp_Init();
  ntasks = Queue_Task_GetLength();
                                                     /* [4] */
  while(Queue_Resp_GetLength() < ntasks)
        THREADmonitorsignalandwait(carep->cp->mon,
                    DISPATCHER_QUEUE, MANAGER_QUEUE);

  return(tmp);

}
```

Figure 3.1: Procedure for Direction Calculation

4. `Send Tasks to Simulators` : Tasks are sent to the simulators, engineering analyses computed, and constraint and objective function evaluated. A summary of constraint and objective function values is returned to the process manager. These quantities and stored temporarily in a response queue, as illustrated in Figure [1.1], and described in Section 2.9.

3.4.2 Step 2 : Correction for Superlinear Convergence

A second order correction $\tilde{d} = \tilde{d}(x, d, H)$ is computed so that convergence close to the optimal solution takes only one step. The calculation of $\tilde{d}$ requires one auxiliary function evaluation at $x+d$, plus the solution of a second quadratic program; details are given in Step 1, part *iv* of Algorithm FSQP-AL.

3.4.3 Step 3 : Steplength Calculation

After the search direction vector has been calculated, trial points for the $[k+1]^{th}$ iteration are selected according to the rule:

$$x_{k+1} = x_k + t_k d_k + t_k^2 \tilde{d}_k \tag{3.4.3}$$

where d_k is the search direction, t_k is the stepsize in the sequence $\{1, \beta, \beta^2, \cdots\}$ and $\beta = 0.5$; for details, see `Step 2` of Section 3.2.1. Instead of selecting single values of t_k in a decreasing sequence, as has been done in FSQP-AL, FSQP-DIS simulates groups of t_k values in parallel over the number of available processors.

Step [1] in the script of code in Figure 3.2 shows that a separate thread is employed for the step length calculation. Steps [2], [3] and [4] cover the gen-

eration of trial design points, and their placement in a simulation task queue. As a naive starting point, the length of the queue is set to NO_RESOURCES, the number of available remote simulators. Step [6] covers the details of this thread interacting with the dispatcher threads, the posting of tasks to the remote simulators, and eventually, the assembly of a simulation response queue that will have a final length = NO_RESOURCES.

Worst case performance occurs when the first trial point (i.e., $t_k = 1$) satisfies the line search requirement. This is because time may be wasted in assembling the complete response queue. Experience indicates, however, that for the first few iterations of optimization, a steplength with $t_k < 1$ this most often needed. If $t_k = 1$ does not satisfy the line search requirement, then objective and constraint values for a smaller values of t_k are immediately available. Time saving is significant because the overhead in building task queues, and sending/receiving data is insignificant compared to the time required to compute additional engineering analyses.

```
TMP_PTR
x_new(carep,nparam,x,steps,di,d,local)                    /* [1] */
MANAGER_PTR carep;
int nparam,local;
double *x, steps, *di, *d;
{
THREAD_MANAGER_BLOCK manager;
MESSAGE_PACKET_PTR mp;
TMP_PTR tmp;
double *xnew,d1,xtemp;
TASK_PTR tp,tp1;
int i,j,k,ntasks;

     THREADmonitorentry(carep->cp->mon, &manager);        /* [2] */

     Queue_Task_Init();
     for(i=1; i<= NO_RESOURCES; i++) {                    /* [3] */
        for(j=1; j<=nparam; j++) {
            if(local == TRUE)                             /* [4] */
               xnew[j] = x[j] + steps*di[j];
            else
               xnew[j] = x[j] + steps*di[j] + d[j]*steps*steps;

            tmp->xnew[i][j] = xnew[j];
        }

        /* Build Queue Of Trial Points */                 /* [5] */

        tp = (TASK_PTR)calloc(1,sizeof(TASK));
        tp->task_no     = (int) i;
        tp->datatype    = STEPLENGTH;
        tp->commandtype = STEPLENGTH_INIT;
        tp->u.msip = (MAN_TO_SIM_INITIAL_PTR)
                     calloc(1,sizeof(MAN_TO_SIM_INITIAL));

        for(k = 1; k <= nparam; k++)
            tp->u.msip->x[k] = xnew[k];

        Queue_Task_Add(tp);
        steps = 0.5*steps;
    }

    /* (d) : Send Trial Points to simulators */

    Tasks_Completed = 0;                                  /* [6] */
    ntasks = Queue_Task_GetLength();
    while(Queue_Resp_GetLength() < ntasks)
          THREADmonitorsignalandwait(carep->cp->mon,
                                     DISPATCHER_QUEUE, MANAGER_QUEUE);

    THREADmonitorexit(carep->cp->mon);

    return(tmp);
}
```

Figure 3.2: Procedure for Steplength Calculation

CHAPTER
FOUR

Numerical Experiments

4.1 Introduction

This chapter discusses the performance of FSQP-DIS (the distributed version of the FSQP optimization algorithm) in the DNC environment. Three application problems are studied. The first is a simple mathematical programming problem having three design variables, two constraints, and one objective function. The second and third problems are to optimize the weight of 2-dimensional and 3-dimensional unsymmetrical steel frame buildings. In each case, experimental results from the distributed implementation are compared to the sequential version of FSQP (FSQP-SEQ).

4.2 UNIX Profiler

All of the components in the DNC environment, including the optimization algorithms, simulators, and graphical user interface, are timed using the UNIX profiler `prof`. This facility produces an execution profile of a program showing the number of times a function is called, the percentage of time spent in

Mathematical Programming Problem	
Initial Guess	$x = [0.1, 0.7, 0.2]$
Final Solution	$x = [0.00000000, 0.0000000, 1.0000000]$
Initial Objective	7.2
Final Objective	1.0
No. of Iterations	3

Table 4.1: Results of Mathematical Programming Problem

executing each function, plus the total cumulative time in running the program.

C programs are setup for profiling by adding the `-p` option to the compilation statement. In addition to keeping extra symbol table information around in the executable program, a recording mechanism leaves a file called `mon.out`, which is interpreted by the program `prof`. Profiling times are reported to an accuracy of $\pm$ 0.01 seconds; see Chapter [8] of reference [23] for details.

4.3 Mathematical Programming Program

This simple problem is borrowed from [41]. It optimizes the function f with three variables x_1, x_2 and x_3

$$f(x) = (x_1 + 3x_2 + x_3)^2 + 4(x_1 - x_2)^2$$

subjected to nonlinear inequality constraints,

$$x_1^3 - 6x_2 - 4x_3 + 3 \leq 0$$

and linear equality constraints,

$$1 - x_1 - x_2 - x_3 = 0$$

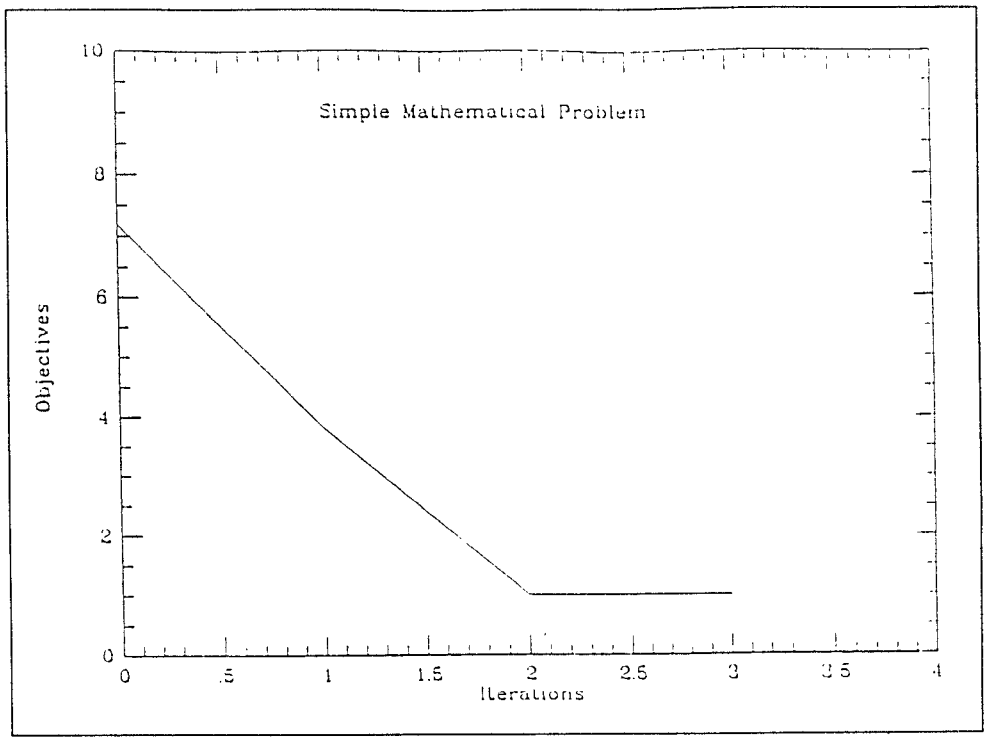


Figure 4.1: Objective Function versus Iteration No

The simple bounds on the variables are

$$0 \leq x_i, \qquad i = 1, \cdots, 3$$

4.3.1 Computer Implementation of Mathematical Problem

The gradients of the objective and constraints functions with respect to design variable components are estimated via forward finite differences. For FSQP-SEQ, the FSQP algorithm, constraints, and objective functions are all located on a single SUN SPARC workstation. In the FSQP-DIS implementation, copies of the objective and constraint functions were positioned at each of three remote SUN SPARC workstations; their names are `Lorentz`, `Galaxy` and `Poisson`.

4.3.2 Results Mathematical Programming Program

The simulation results are summarized in Table 4.1. The values of objective function versus iteration number are plotted in Figure 4.1. The total time required to run FSQP-SEQ and FSQP-DIS are shown in Table 4.2 and Table 4.3 respectively. Figure 4.2 displays the profile data of the simple mathematical problem. The key observations are:

1. For a very simple mathematical problem having just two constraints and one objective, the sequential version runs much faster than the distributed version (in fact twelve times faster). This is because the communication overhead associated in the distributed computing environment, for instance, setting up the DNC architecture, sending data to/from the simulators and so on, is very high compared to required computation of the optimization problem.

2. Figure 4.2 summarizes the profiling output for FSQP-SEQ. The profile indicates that most of the time is dedicated to low level assembly code operations, and input/output. The computational effort to evaluate the constraint and objective functions is negligible.

Timing of Sequential Algorithm	
Machine Type	SUN SPARC
CPU percentage	20
Other users	None
Number of Simulations	35
Simulation Time (Seconds)	0.2

Table 4.2: Timing of Sequential Algorithm

Distributed Optimization					
	GUI	Manager	Sim 1	Sim 2	Sim 3
Machine	Newton	Descartes	Lorentz	Galaxy	Poisson
Type	SPARC	SPARC	SPARC	SPARC	SPARC
CPU (%)	27	25	1	1	1
Other User	None	None	None	None	None
No. Simulations	-	-	21	15	18
Time (Seconds)	5.5	2.5	0.05	0.06	0.06

Table 4.3: Timing of Distributed Algorithm

%time	cumsecs	#call	ms/call	name
21.1	0.04			.rem
10.5	0.06			.div
10.5	0.08			__umac
10.5	0.10			_pow
5.3	0.11			.umul
5.3	0.12			__unpack_double
5.3	0.13			_addcon_
5.3	0.14			_bndalf_
5.3	0.15			_fstat
5.3	0.16			_lpcore_
5.3	0.17			_printf
5.3	0.18			_quotnt_
5.3	0.19			_v2norm_
0.0	0.19	1	0.00	_check
0.0	0.19	23	0.00	_constr
0.0	0.19	2	0.00	_di1
0.0	0.19	3	0.00	_diagnl
0.0	0.19	3	0.00	_dir
0.0	0.19	5	0.00	_dqp
0.0	0.19	1	0.00	_fsqpd
0.0	0.19	1	0.00	_fsqpd1
0.0	0.19	4	0.00	_grcnfd
0.0	0.19	3	0.00	_grobfd
0.0	0.19	2	0.00	_hesian
0.0	0.19	1	0.00	_main
0.0	0.19	5	0.00	_matrcp
0.0	0.19	7	0.00	_matrvc
0.0	0.19	25	0.00	_nullvc
0.0	0.19	12	0.00	_obj
0.0	0.19	4	0.00	_out
0.0	0.19	33	0.00	_qphess
0.0	0.19	20	0.00	_scaprd
0.0	0.19	4	0.00	_slope
0.0	0.19	1	0.00	_small
0.0	0.19	2	0.00	_step
0.0	0.19	61	0.00	_subout

Figure 4.2: Timing Profile for Mathematical Problem

4.4 Finite Element Computer Package

Each of the remote simulators in DNC employs a prototype version the finite element analysis program called FERA [35] to compute structural responses quantities (bending moments, axial forces, nodal displacements, and so on).

FERA [35] is an acronym for Finite Element and Rigidbody Analysis. The program is written exclusively in the C programming language, and as such, makes full use of various data structures - linked list, hashtable and so on - to assist in the management of data. Its library of finite elements includes plane-stress plane strain, two- and three-dimensional frame elements, and DKQ plate element. In addition, it has the ability to model a rigid body connected to flexible elastic elements. Currently, FERA is limited to linear elastic analysis.

For the optimization, two new features were added to FERA. They are: (1) facilities for using the UNIX tool YACC [21] to parse a data file containing parameters for the design constraints and objectives, and (2) procedures for checking the design constraints and objectives.

4.4.1 Data Structures for Design Performance

In Section 2.9 it was pointed out that details of data structures shown in Table 2.9 are application dependent. With this in mind, Figure 4.3 shows the C preprocessor definitions and data structures used to send simulation tasks from the manager to the simulators, and to return constraint values from the simulators to the dispatcher thread. In Table 2.9, `NCONST` and `NOBJ` are the

```
/* ================================================== */
/* Data Structures for Manager/Simulator Message Passing */
/* ================================================== */

#define NCONST              32
#define NPARA                3
#define NOBJ                 1
#define NSIM                 3

/* Manager => Simulator Response Data Structure */

typedef struct ms_result {
        int                 task_no;
        double           x[NPARA+1];
} MAN_TO_SIM_RESPONSE, *MAN_TO_SIM_RESPONSE_PTR;

/* Simulator => Manager Response Data Structure */

typedef struct result {
        int                 task_no;
        double     gradient[NCONST+NOBJ+1];
} SIM_TO_MAN_RESPONSE, *SIM_TO_MAN_RESPONSE_PTR;
```

Figure 4.3: Data Structures for Design Performance Messages

number of design constraints and objectives, and NPARA is the number of design parameters. It is important to note that since these parameters are problem dependent, the process manager must be recompiled before a new problem is executed.

4.4.2 Modeling of 2-D Planar Steel Frame

The two-bay two-story planar frame shown in Figure 4.4 has five column elements and four beam elements. All elements are wide flange structural steel sections with material type ASTM A36, and a minimum yield stress of 36 ksi. The specified dead load is 80 lbs per square foot, and the specified live load is 40 lbs per square foot for the floors. The roof live load is 20 lbs per square foot.

The two-dimensional frame is assumed to be a typical frame in a long struc-

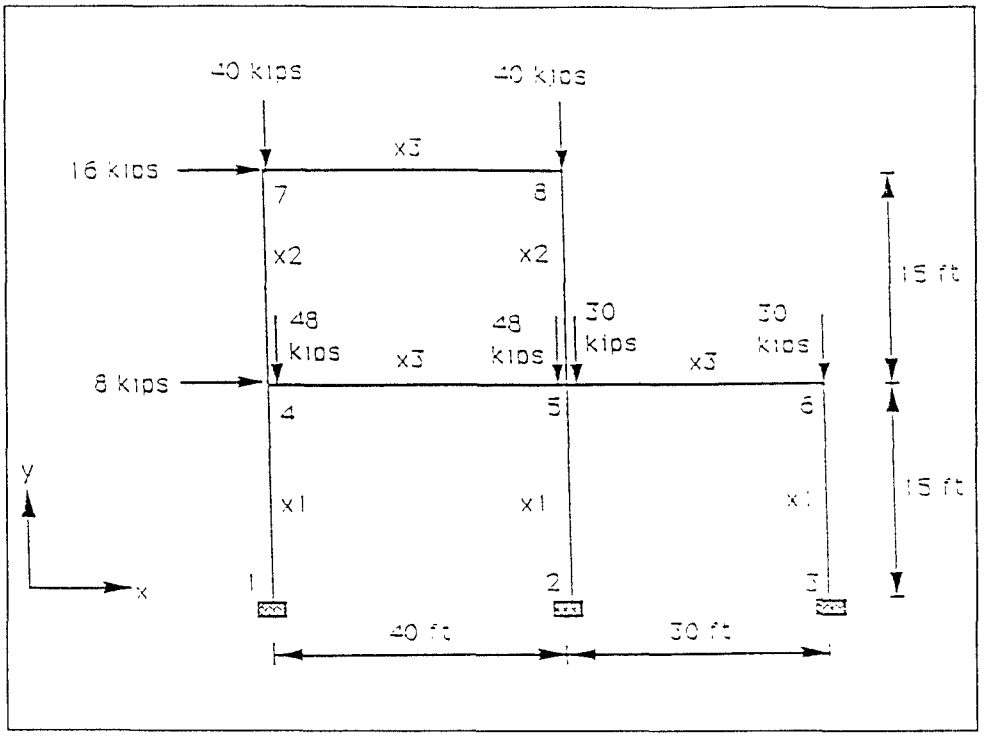


Figure 4.4: 2-D Unsymmetrical Building

ture where frames are spaced at centers 20 feet. Thus, the uniform gravity loads are 0.2000 kips/in for the floor, and 0.1667 kips/in for the roof. In addition, a total moderate seismic lateral loading of 24 kips - this is approximately 10% of the structural weight - was distributed over the height of the structure.

The bases of the three columns are assumed to be fully fixed. Each joint in the steel frame is modeled with two translational and one rotational degree of freedom. Hence, the size of the stiffness matrix is 15 by 15.

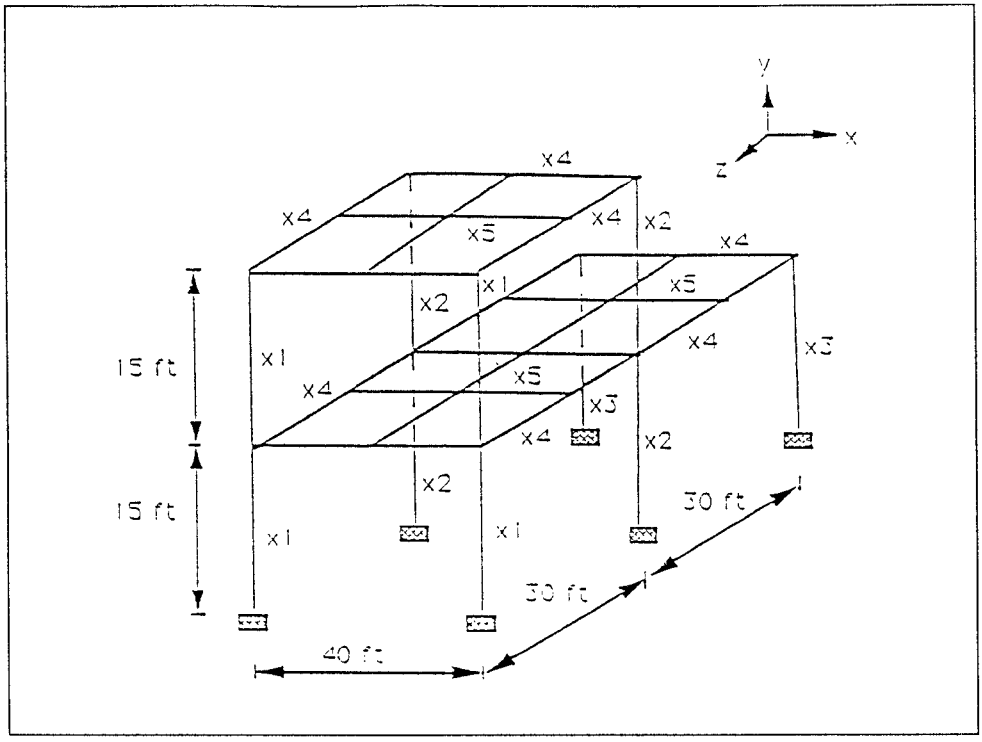


Figure 4.5: 3-D Unsymmetrical Steel Building

4.4.3 Modeling of 3-D Steel Frame Building

Figure 4.5 shows the geometry and element connectivity for the three dimensional steel building frame. It has 10 column elements, 20 beam elements around the exterior of the first and second floors, and 14 beam elements for interior floor support. All elements are assumed to be wide flange structural steel sections with material type `ASTM A36`, and a minimum yield stress of 36 `ksi`.

Figure 4.6 summarizes the external loads applied to the building frame. The floors and roof are assumed to have a nominal dead load of 80 lb per square foot. For the roof, live loads are assumed to be 20 `lbs` per square foot, and on the

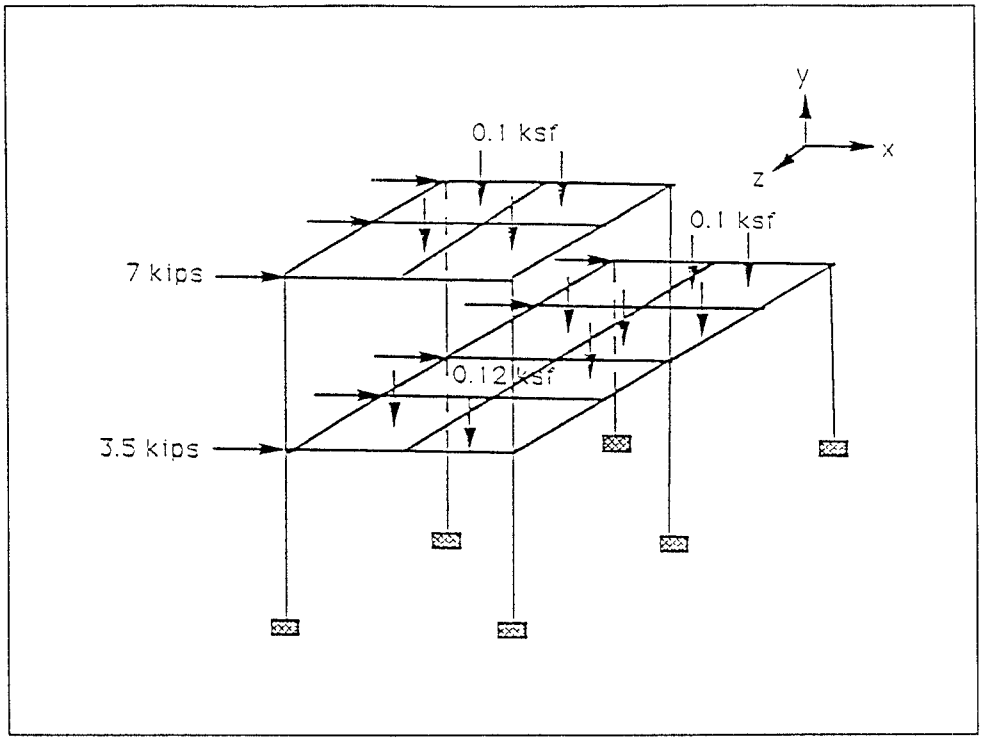


Figure 4.6: 3-D Unsymmetrical Steel Building : Loads

floors, 40 lbs per square foot. The total unfactored weight of the structure is 384 kips. In addition, a total moderate seismic lateral loading of 35 kips - this is approximately 10% of the structural weight - was distributed over the height of the structure.

The six column bases are assumed to be fully fixed. Each joint in the steel frame is modeled with three translational and three rotational degrees of freedom. Hence, the size of the stiffness matrix is 144 by 144.

4.5 Formulation of Optimization Problem

The components of the optimization problem formulation are design parameters, design constraints, and design objectives.

4.6 Design Parameters

For simplicity, the size of each beam and column frame element is represented by a single design parameter, its principal moment of inertia.

4.6.1 Section Relationships

Empirical relationships are used to describe section properties, such as section depth and radius of gyration, as a function of section moment of inertia. The empirical relationships are based on the observation that section depth is approximately proportional to the moment of inertia raised to a rational power. Similarly, the radius of gyration for columns and girders is approximately proportional to the section depth raised to a rational power. Walker [39] used these approximations, and a nonlinear least-squares analysis on wide flange sections to derive the following functional relationships:

For columns with $I \leq 429in^4$

$$D = 1.47I^{0.368}$$

$$R = 0.39D^{1.04}$$

For columns with $I > 429in^4$

$$D = 10.5I^{0.0436}$$

$$R = 0.39D^{1.04}$$

For girders

$$D = 2.66I^{0.287}$$

$$R = 0.52D^{0.92}$$

where `D` = section depth in inches, `I` = moment of inertia in inches^4, and `R` is the radius of gyration in inches. Once the radius of gyration is known, the cross section area is simply given by $A = [I/R^2]$.

4.6.2 Design Parameters for 2D Steel Frame

Figure 4.4 shows that the optimization problem was cast with three design variables. They are:

1. `X1` : Moment of Inertia of the first storey columns.
2. `X2` : Moment of Inertia of the second storey columns.
3. `X3` : Moment of Inertia of the floor beams.

The section moment of inertia for each frame element is constrained to lie in the range of $[125in^4, 5000in^4]$.

4.6.3 Design Parameters for 3D Steel Frame

Figure 4.5 shows that optimization problem for the 3D steel building frame was cast with five design variables. They are:

1. `X1` : Moment of Inertia of the columns at z = 60 ft.
2. `X2` : Moment of Inertia of the middle columns at z = 30 ft.
3. `X3` : Moment of Inertia of the right columns at z = 0 ft.
4. `X4` : Moment of Inertia of exterior floor beams.
5. `X5` : Moment of Inertia of interior floor beams.

The section moment of inertia is assumed to lie in the range of $[125in^4, 5000in^4]$.

4.7 Design Dissatisfaction

The adequacy of a structural design is ascertained by simply comparing: (1) the calculated actions at a designer-prescribed reliability level to (2) the ability of the structure to carry these actions without failure [2, 5]. To facilitate this comparison a single design entity called *designer dissatisfaction* that quantifies the results is defined [29].

$$D(const_i \text{ or } obj_i) = \left[\frac{(\texttt{response} - \texttt{GOOD})}{(\texttt{BAD} - \texttt{GOOD})}\right] \tag{4.7.1}$$

In equation (4.7.1), $D(const_i \text{ or } obj_i)$ is the designer dissatisfaction for the i^{th} design constraint or objective. The parameter `response` is the computed structural response value. The `GOOD` and `BAD` structural response parameters are

given by:

$$\texttt{GOOD} = \texttt{Good\_Value} * \texttt{Constraint\_Value} \tag{4.7.2}$$

$$\texttt{BAD} = \texttt{Bad\_Value} * \texttt{Constraint\_Value} \tag{4.7.3}$$

where `Constraint_Value` is an ideal level of structural response at which failure will occur, and `Good_Value` and `Bad_Value` are dimensionless capacity reduction factors. The `Good_Value` and `Bad_Value` are set by the designer in such a way that `GOOD` corresponds to a dependable level of system performance, and `BAD`, to a structural response level at which undesirable performance is almost assured if exceeded.

Dissatisfaction is not a boolean variable simply describing whether or not a constraint or objective is satisfied, but a function whose value depends on the magnitude of a constraint or objective violation. It is less than or equal to zero for a conservative design, becomes slightly nonzero - i.e., within the interval (0,1) - as the design becomes more economical, and increases above 1 as the design becomes increasingly unconservative.

4.7.1 Definition of Dissatisfaction for FSQP Implementation

Ideally, a maximum dissatisfaction among all of the performance attributes of about 0.5 should be aimed at since this is roughly half way between a design that is too conservative, and one that is believed to be unreliable. Because the FSQP algorithm described in Chapter 3 requires all constraint measures to be

less than zero for a design to remain feasible, equation (4.7.1) was modified to:

$$D(const_i \text{ or } obj_i) = \left[\frac{(\texttt{response} - \texttt{GOOD})}{(\texttt{BAD} - \texttt{GOOD})}\right] - \left[\frac{1}{2}\right] \qquad (4.7.4)$$

Notice that the same effect could be achieved by adjusting the GOOD values, but this would also require a change in its engineering interpretation.

4.7.2 Structural Design Constraints

The performance of two- and three-dimensional frame elements is checked with three constraints. In the spirit of equations (4.7.2) and (4.7.3), element axial forces are required to satisfy the constraint:

$$\text{allowable axial force} < [\texttt{Good\_Value}, \texttt{Bad\_Value}] * \text{ideal axial force.} \qquad (4.7.5)$$

Here the ideal axial force is one of the two possible values. For compressive loading, it is the Euler buckling force with pin-pin end conditions. Otherwise, it is the axial force needed to yield the element in tension.

Bending moments in the beam and column elements are checked at the end points only. Again, the form of the constraint is:

$$\text{allowable bending} < [\texttt{Good\_Value}, \texttt{Bad\_Value}] * \text{ideal bending.} \qquad (4.7.6)$$

The ideal bending moment is that required to cause incipient flexural yielding. Shear forces within the element are not checked. Design constraints based on absolute and relative nodal displacements, such as for story drift and overall frame sway, have not been implemented in this study.

```
group("column") {
       item {
            name "axial_force";
            state      =  ACTIVATED;
            good_value =   0.4;
            bad_value  =   0.5;
       }
       item {
            name "bending_moment";
            state =  ACTIVATED;
            good_value =   0.5;
            bad_value  =   0.6;
       }
}

group("beam") {
       item {
            name "axial_force";
            state =  ACTIVATED;
            good_value =   0.4;
            bad_value  =   0.5;
       }
       item {
            name "bending_moment";
            state =  ACTIVATED;
            good_value =   0.5;
            bad_value  =   0.6;
       }
}

group("objective") {
        item {
             name "volume";
             good_value = 20000;
             bad_value  = 30000;
        }
}
```

Figure 4.7: Design Constraints and Objectives for 2-D Frame

4.7.3 Design Constraints for 2-D Steel Frame

Parameters for the constraints and objectives, as defined in equations (4.7.2-4.7.6), are stored in `beam` and `column` constraint groups as shown in Figure 4.7. A YACC grammar has been written to read and interpret the constraints and objective datafile. Beam and column constraint objects are stored in the FERA symbol table, and retrieved as needed.

The two-dimensional steel frame has 24 design constraints and one design

objective. Axial forces are controlled by setting `Good_Value` and `Bad_Value` to 0.4 and 0.5, respectively. Since most of the columns will be in compression, under combined gravity loads plus moderate lateral loadings, axial forces are assured to be less than 0.45 of the Euler buckling load. The `Good_Value` and `Bad_Value` parameters for flexural bending were set to 0.5 and 0.6, thereby ensuring stresses remain within the working stress range.

4.7.4 Design Constraints for 3-D Steel Building Frame

The design constraints for the three-dimensional frame are the same as for the 2-D steel frame. Since empirical relationships are employed to connect primary and secondary cross section properties of frame members, constraints are not explicitly checked for the interaction of axial forces with biaxial bending. Instead, constraints checking is simplified by first finding the maximum absolute bending moment, and substituting it into equation (4.7.6).

4.8 Design Objective

The design objective of these two problems is to find the minimum volume of frame elements that also satisfies the constraint requirements. The `good_value` and `bad_value` parameters for the 2-D steel frame problem are as shown in Figure 4.7. For the 3-D building design objective, `good_value` = 150000 and `bad_value` = 200000.

4.9 Optimization Results

Recall that in FSQP-DIS, the main FSQP algorithm executes as a lightweight process on the process manager. Copies of the FERA computer package, routines for evaluating constraints and objectives, and YACC [21] code for reading constraint/objective datafiles are located on each remote simulator.

Design vectors are sent from the process manager to the remote simulators. Once the simulation and constraint/objective evaluation is complete, vectors of design dissatisfactions are sent back to the dispatcher thread.

4.9.1 2-D Building

The FSQP-SEQ and FSQP-DIS algorithms both run for 36 iterations, and converge to the results shown in Table 4.4. Figures 4.8, 4.9, 4.10 and 4.11 show objective function value versus iteration no, the maximum constraint value versus iteration no, design variables `X1-X3` versus iteration no, and number of steps needed in line search computation at each iteration. Tables 4.5 and 4.6 summarize the computational work - number of simulations, time of simulation - for the sequential and distributed versions of the optimization algorithm, respectively. Figure 4.12 shows the percentage of time spent in executing each function of the sequential implementation.

2-D Unsymmetrical Building	
Initial Guess	$x = [500.0, 500.0, 500.0]$
Final Solution	$x = [461.1, 231.9, 125.0]$
Initial Objective	1.121458
Final Objective	0.295073
Initial Max. Constraint	-2.133162
Final Max. Constraint	-0.005192
No. of Iterations	36

Table 4.4: 2-D Building : Simulations Results

Timing of 2-D Building : Sequential Optimization	
Machine Type	SUN SPARC
CPU percentage	70
Other User	None
Number of Simulations	188
Simulation Time (Seconds)	7.80

Table 4.5: Timing of 2-D Building : Sequential Optimization

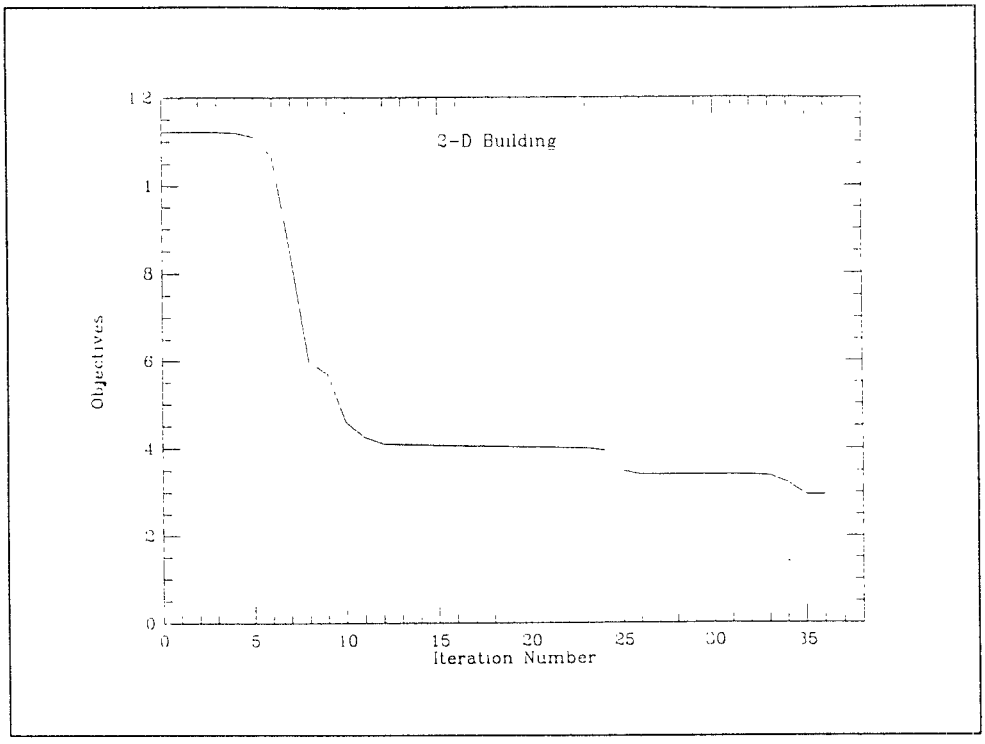


Figure 4.8: 2-D Building : Objective Values

Timing of 2-D Building : Distributed Optimization					
	GUI	Manager	Sim 1	Sim 2	Sim 3
Machine	Banach	Eiffel	Newton	Laplace	Tex1
Type	SPARC	SPARC	SPARC	SPARC	SPARC
CPU (%)	70	31	5	7	6
Other User	None	None	None	None	None
No. Simulations	-	-	73	104	72
Time (Seconds)	15.5	10.8	2.3	4.0	2.8

Table 4.6: Timing of 2-D Building : Distributed Optimization

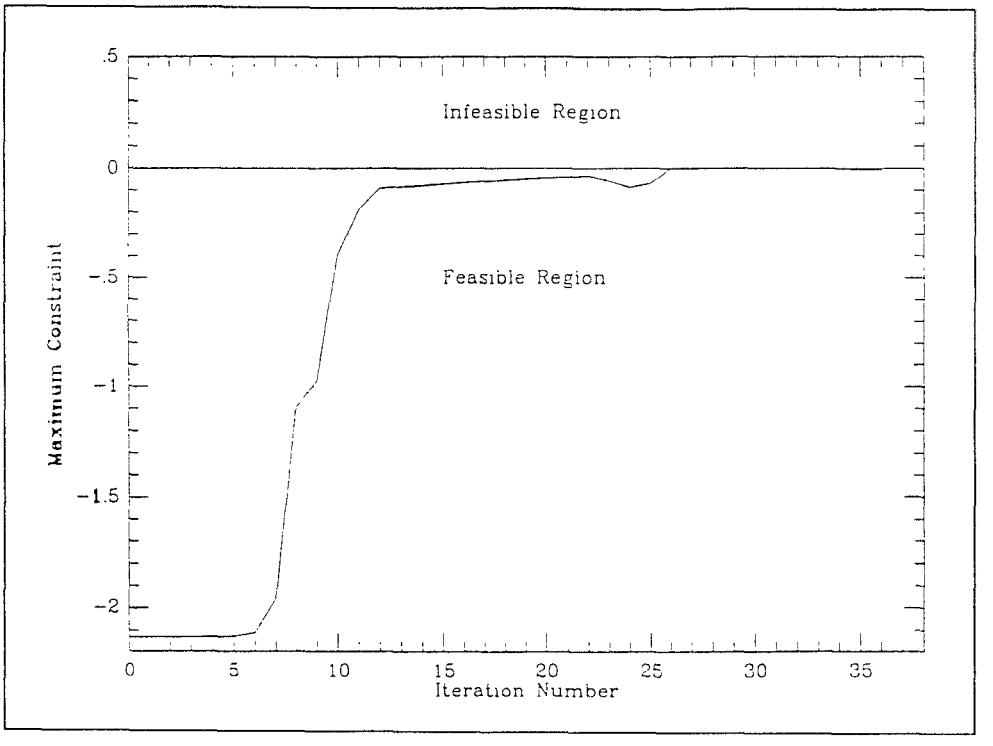


Figure 4.9: 2-D Building : Maximum Constraint Values

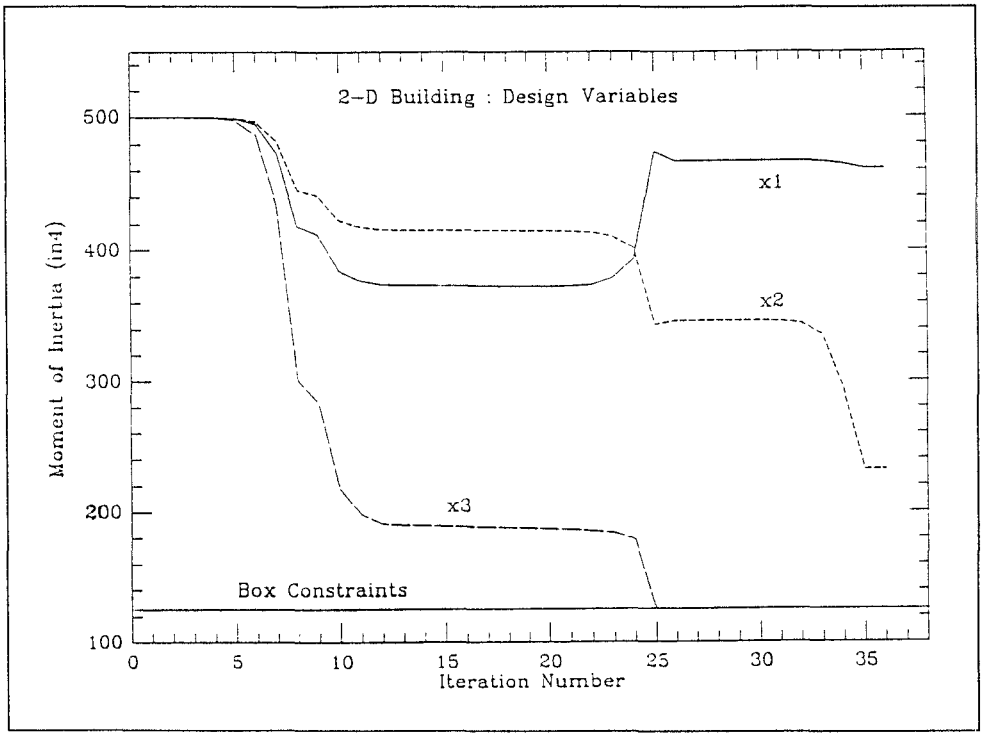


Figure 4.10: 2-D Building : Design Variables

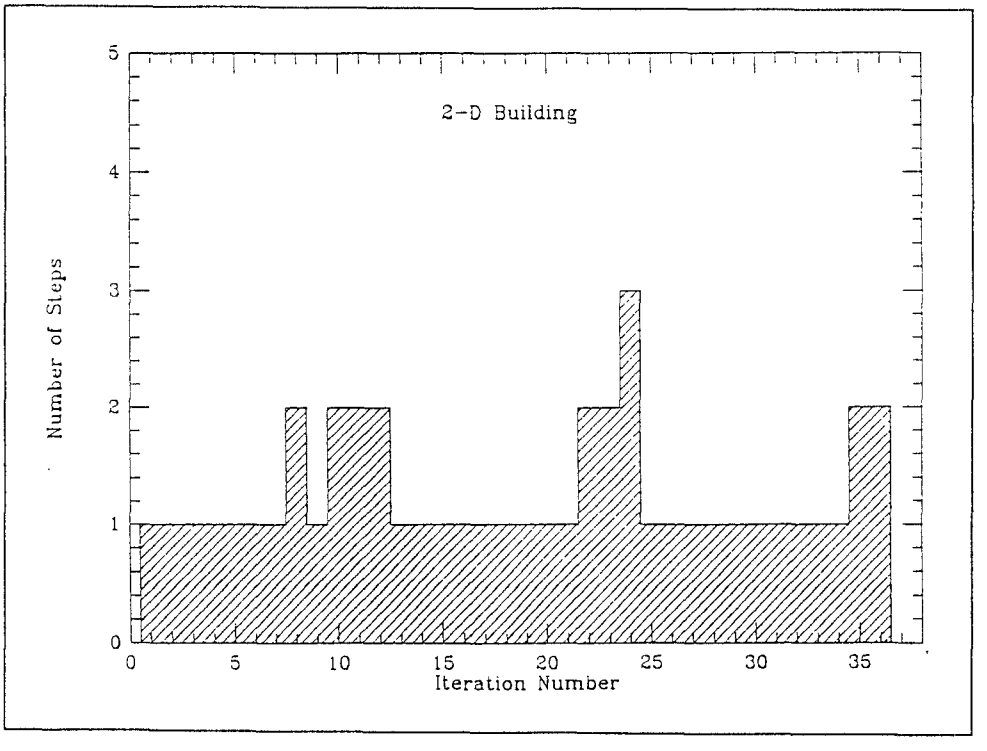


Figure 4.11: 2-D Building : Number of Steps

The following points should be noted on Tables [4.4 - 4.6] and Figures [4.9 - 4.11]:

1. The two-dimension building frame is a very small sized structural problem; approximately 70% of the total computational work is dedicated to finite element simulations. The distributed version executes at almost the same speed as the sequential version of the optimization algorithm (10.8 seconds as compared to 7.8 seconds).

2. The total number of simulations in the sequential version (188) is less than the total number of simulations for the distributed implementation (249). This is because the line search computation of the distributed version is done in groups of three trial designs. Figure 4.11 shows that for most iterations the stepsize, $t_k = 1$ satisfies the line search requirement. In fact, in all but iteration 24, trial steplength simulations for $t_k = 1/2$ (and sometimes $t_k = 1/4$) are discarded once the largest possible stepsize is satisfied. A strategy for mitigating the excess computation will be suggested in Chapter 5.

3. The design variables do not converge at the same rate. In fact, component `X3` converges faster than the others. Only after `X3` converges hits the lower box constraint do significant adjustments to `X1` and `X2` take place.

4. The maximum dissatisfaction among design constraints increases from -2.13 to -0.005 during the course of the optimization. For the starting

```
%time  cumsecs  #call  ms/call  name
 15.8     1.23    188     6.54  _dLU_Decomposition
  4.9     1.61   3008     0.13  _Assign_p_Array
  4.6     1.97                  _pow
  3.8     2.27   4515     0.07  _elmlib
  3.1     2.51    189     1.27  _profile
  3.1     2.75                  _v2norm_

  -------------- output deleted ------------------

  2.8     3.19    188     1.17  _check_dconst
  2.4     3.58   1504     0.13  _rotate
  2.2     3.93   1504     0.11  _Assemble_Stiffness
  2.1     4.09                  _malloc
  1.9     4.39    188     0.80  _dLU_Backsubstitution
  1.9     4.54   4515     0.03  _elmt07
  1.7     4.96    188     0.69  _Fera_Optimization
  1.3     5.42                  _bzero
  1.0     5.76   1504     0.05  _Destin_Array
  0.8     6.17                  _lpgrad_
  0.8     6.23    106     0.57  _matrvc
  0.6     6.43     71     0.70  _dqp
  0.6     6.53    635     0.08  _qphess
  0.5     6.57    188     0.21  _Fsqpd_General
  0.4     6.90     36     0.83  _dir
  0.4     6.96     35     0.86  _hesian
  0.3     7.31     36     0.56  _nngrfd
  0.3     7.33    188     0.11  _pload
  0.3     7.45     35     0.57  _step

  -------------- output deleted ---------------------

  0.1     7.60    188     0.05  _calc_dc_size
  0.1     7.64     35     0.29  _di1
  0.1     7.68      1    10.00  _fsqpd1
  0.1     7.69    840     0.01  _fuscmp
  0.1     7.72    388     0.03  _nullvc
  0.1     7.73                  _qpgrad_
  0.0     7.80    188     0.00  _check_dobj
  0.0     7.80      1     0.00  _fsqpd
  0.0     7.80     70     0.00  _nscaprd
  0.0     7.80    247     0.00  _scaprd
  0.0     7.80     70     0.00  _slope
  0.0     7.80      1     0.00  _small
```

Figure 4.12: 2-D Building : Profile for Sequential Implementation

design, the maximum dissatisfaction corresponds to axial force in the first storey element of the middle column line. The critical dissatisfaction of the optimal design is caused by the ground level bending moment of the right most column.

5. The objective function decreases from 1.12, at the beginning of iteration 1, to 0.30 after 36 iterations.

4.9.2 3-D Building

The 3-D building frame is still a small sized optimization problem. Table 4.7 summarizes the optimization results for the 3-D building frame problem. Both the sequential and distributed implementations run for 63 iterations before the final solution is obtained. Figures 4.13, 4.14, 4.15 and 4.16 show the plots of objective, maximum constraint, design variables and number of steps in line search at every iteration. Table 4.8 shows the total time to run the sequential version of the algorithm. Table 4.9 shows the total time spent in each component of the distributed environment using three simulators.

The distributed version of the optimization algorithm was repeated for three, four, and five remote simulators. Column 2 of Table 4.10 shows the maximum CPU seconds used among the process manager and remote simulators. The speedup in computation is shown in column 3. Speedup is defined as $[T_s/T_d]$, where T_s is the execution time for the sequential implementation, and T_d is the corresponding entry of column 2 for the given number of simulators. An

estimate of the overall efficiency of the system is shown in column 4 of Table 4.10; efficiency is defined as $\left[\frac{T_s}{N \cdot T_d}\right] \cdot 100\%$, where N is the number of remote simulators. Finally, a profile summary of functions executed in the 3-D building optimization is shown in Figure 4.17. The abovementioned tables and figures indicate:

1. The two storey 3-D unsymmetrical building frame is a small sized optimization problem. Figure 4.17 shows that more than 95 % of the computational time is dedicated to the calculation of structural responses/simulations. Less than 5 % of the total computational work is needed for the FSQP optimization algorithm. Notice that this 95% - 5% division in resources for the simulation/optimization is consistent with statements made in Chapter 1 of this report.

2. The maximum value of dissatisfaction for the design constraints increases from -0.21 to -0.000053 as the design is updated from iteration 0 to iteration 63. For both the initial and final designs, the critical dissatisfaction corresponds to bending moments in the interior girders at the first floor level. The objective function decreases from 2.11 initially to 0.89 after 63 iterations

3. The speedup in computations due to the use of multiple simulators has a maximum value of 3.26 when 5 remote simulators are employed. The computational efficiency is approximately 66%.

4. The remote simulations are distributed quite evenly among available sim-

ulators, as indicated in row `No Simulations` of Table 4.9. The total number of simulations for the FSQP-DIS implementation is slightly more than that in the sequential version, 591 versus 494. Again, this occurs for the reasons explained in Section 4.9.1.

5. Figure 4.16 shows that nearly half of the optimization iterates required more than one step in the line search computation; i.e., $t_k < 1$. Some iterations needed more than five trial steps !! Experience indicates that during the latter stages of the optimization, FSQP often needs only one trial steplength. The results of this problem deviate from usual behavior. This observation could be caused by a number of factors. First, the `axial force` design constraint for each beam and column element is not continuously differentiable. Second, the design space could be quite bumpy, possibly containing pockets the algorithm gets cornered in.

3-D Unsymmetrical Building	
Initial Guess	$x = [1100.0, 1100.0, 1100.0, 1100.0, 1100.0]$
Final Solution	$x = [539.3, 634.4, 440.8, 1075.6, 537.6]$
Initial Objective	2.114658
Final Objective	0.888923
Initial Max. Constraint	-0.208572
Final Max. Constraint	-0.000053
No. of Iterations	63

Table 4.7: 3-D Building : Simulations Results

Timing of 3-D Building : Sequential Optimization	
Machine Type	`Sun SPARC`
CPU percentage	98
Other User	None
Number of Simulations	494
Simulation Time (Seconds)	2372

Table 4.8: Timing of 3-D Building : Sequential Optimization

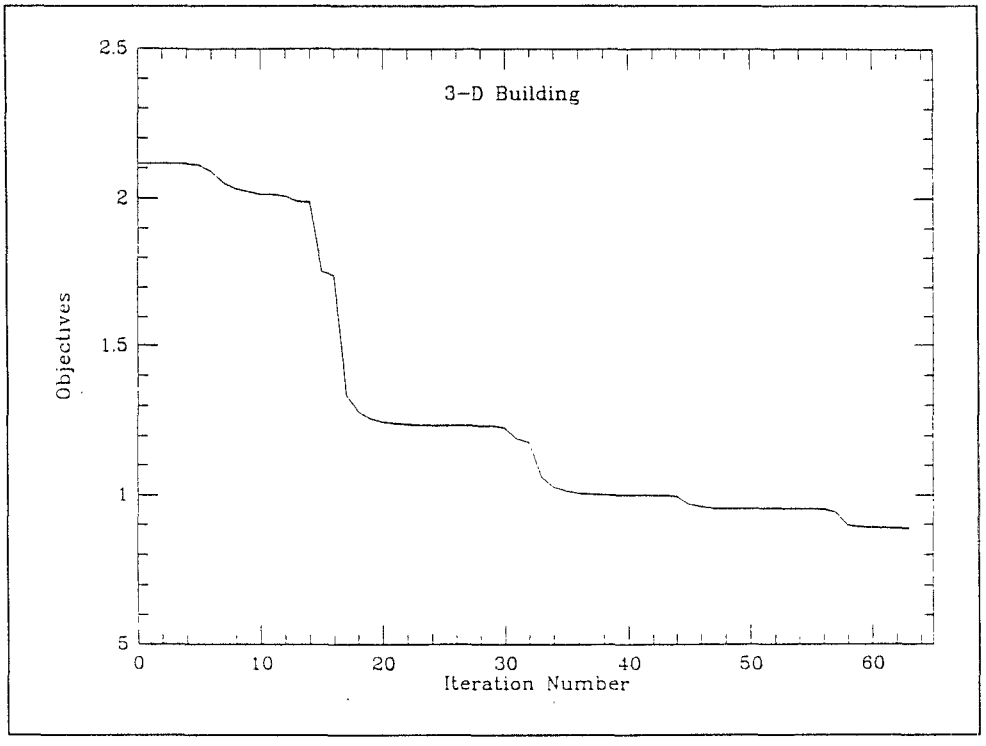


Figure 4.13: 3-D Building : Objective Values

Timing of 3-D Building : Distributed Optimization					
	GUI	Manager	Sim 1	Sim 2	Sim 3
Machine	Newton	Lorentz	Laplace	Poisson	Taylor
Type	SPARC	SPARC	SPARC	SPARC	SPARC
CPU (%)	94	90	87	49	66
Other User	None	None	None	None	None
No. Simulations	-	-	258	136	197
Time (Seconds)	526.3	854.2	1181.5	661.5	901.5

Table 4.9: Timing of 3-D Building : Distributed Optimization

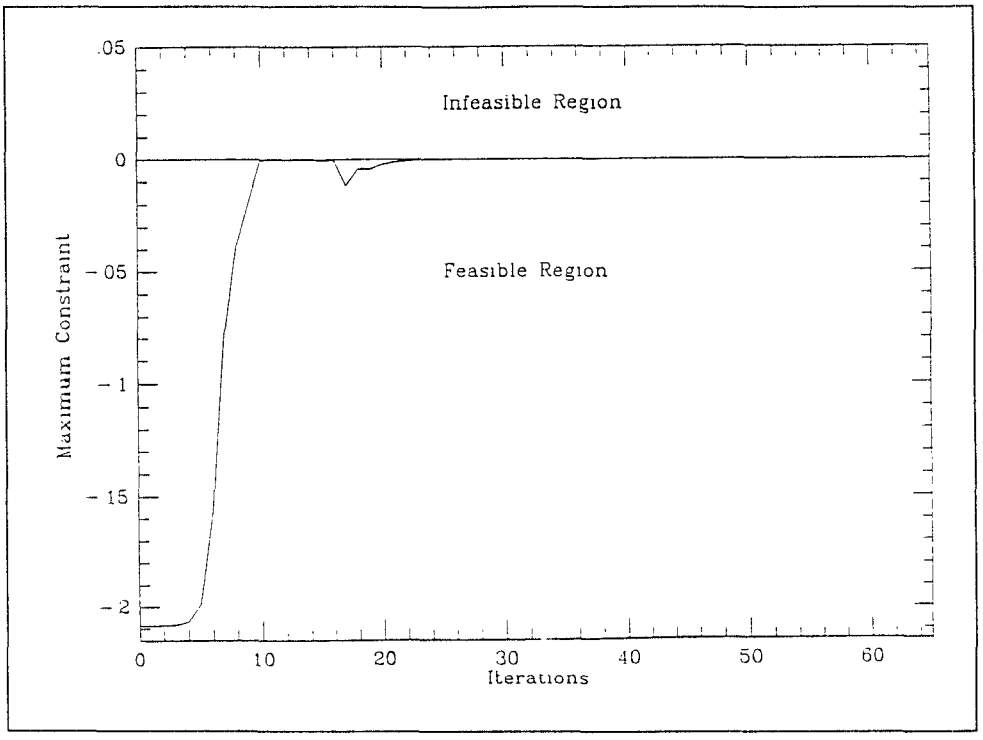


Figure 4.14: 3-D Building : Maximum Constraint Values

3-D Building : Speedup and Efficiency			
No Simulators	Time (seconds)	Speedup	Efficiency (%)
1	2372.0	1.00	100
3	1181.5	2.00	66
4	843.3	2.81	70
5	727.5	3.26	66

Table 4.10: 3-D Building : Speedup and Efficiency

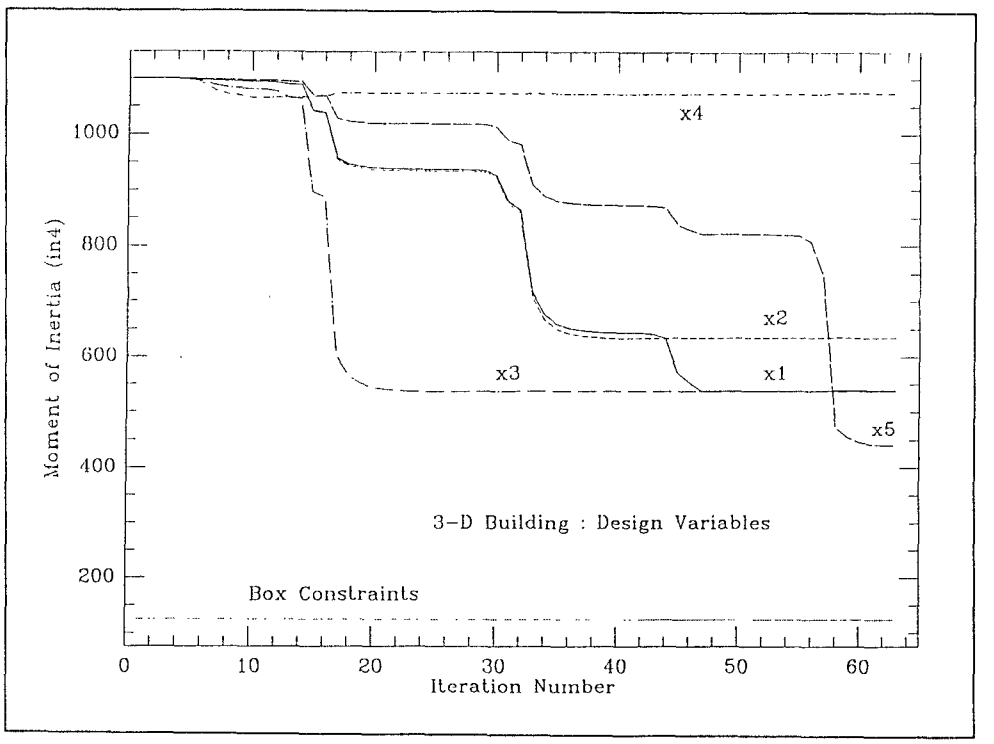


Figure 4.15: 3-D Building : Design Variables

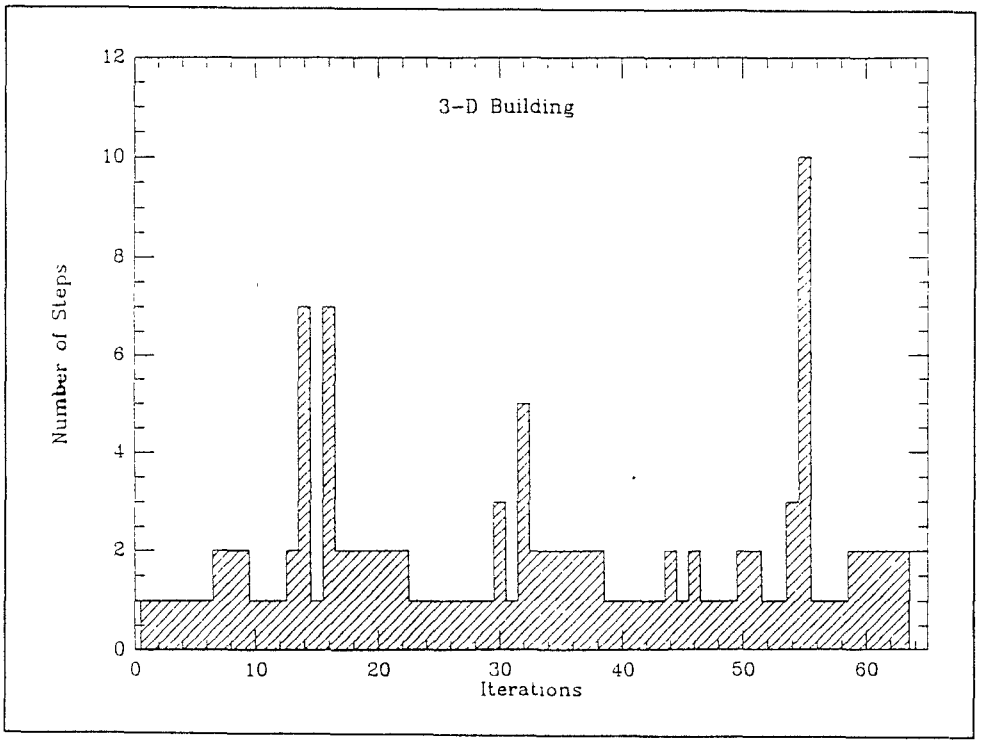
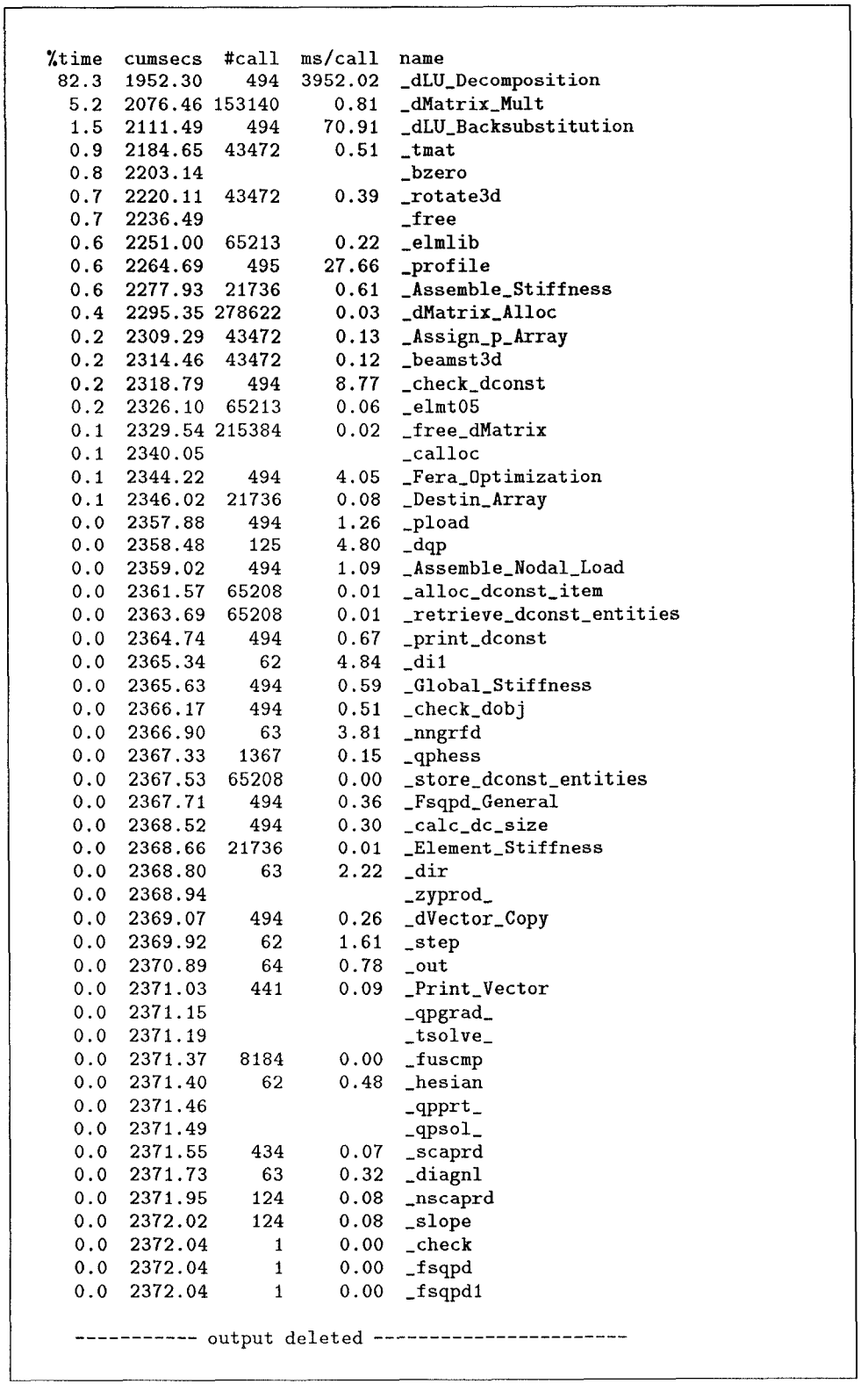


Figure 4.16: 3-D Building : Number of Steps



%time	cumsecs	#call	ms/call	name
82.3	1952.30	494	3952.02	_dLU_Decomposition
5.2	2076.46	153140	0.81	_dMatrix_Mult
1.5	2111.49	494	70.91	_dLU_Backsubstitution
0.9	2184.65	43472	0.51	_tmat
0.8	2203.14			_bzero
0.7	2220.11	43472	0.39	_rotate3d
0.7	2236.49			_free
0.6	2251.00	65213	0.22	_elmlib
0.6	2264.69	495	27.66	_profile
0.6	2277.93	21736	0.61	_Assemble_Stiffness
0.4	2295.35	278622	0.03	_dMatrix_Alloc
0.2	2309.29	43472	0.13	_Assign_p_Array
0.2	2314.46	43472	0.12	_beamst3d
0.2	2318.79	494	8.77	_check_dconst
0.2	2326.10	65213	0.06	_elmt05
0.1	2329.54	215384	0.02	_free_dMatrix
0.1	2340.05			_calloc
0.1	2344.22	494	4.05	_Fera_Optimization
0.1	2346.02	21736	0.08	_Destin_Array
0.0	2357.88	494	1.26	_pload
0.0	2358.48	125	4.80	_dqp
0.0	2359.02	494	1.09	_Assemble_Nodal_Load
0.0	2361.57	65208	0.01	_alloc_dconst_item
0.0	2363.69	65208	0.01	_retrieve_dconst_entities
0.0	2364.74	494	0.67	_print_dconst
0.0	2365.34	62	4.84	_di1
0.0	2365.63	494	0.59	_Global_Stiffness
0.0	2366.17	494	0.51	_check_dobj
0.0	2366.90	63	3.81	_nngrfd
0.0	2367.33	1367	0.15	_qphess
0.0	2367.53	65208	0.00	_store_dconst_entities
0.0	2367.71	494	0.36	_Fsqpd_General
0.0	2368.52	494	0.30	_calc_dc_size
0.0	2368.66	21736	0.01	_Element_Stiffness
0.0	2368.80	63	2.22	_dir
0.0	2368.94			_zyprod_
0.0	2369.07	494	0.26	_dVector_Copy
0.0	2369.92	62	1.61	_step
0.0	2370.89	64	0.78	_out
0.0	2371.03	441	0.09	_Print_Vector
0.0	2371.15			_qpgrad_
0.0	2371.19			_tsolve_
0.0	2371.37	8184	0.00	_fuscmp
0.0	2371.40	62	0.48	_hesian
0.0	2371.46			_qpprt_
0.0	2371.49			_qpsol_
0.0	2371.55	434	0.07	_scaprd
0.0	2371.73	63	0.32	_diagnl
0.0	2371.95	124	0.08	_nscaprd
0.0	2372.02	124	0.08	_slope
0.0	2372.04	1	0.00	_check
0.0	2372.04	1	0.00	_fsqpd
0.0	2372.04	1	0.00	_fsqpd1

---------- output deleted ----------

Figure 4.17: 3-D Building : Profile for Sequential Implementation

CHAPTER
FIVE

Conclusion and Future Work

5.1 Summary and Conclusion

In Section 1.3 it was stated that the purposes of this work were to: (a) describe the implementation of the Distributed Numerical Computing Environment, (b) formulate FSQP-DIS, a distributed computing version of the Feasible Sequential Quadratic algorithm that matches the DNC architecture, and (c) conduct optimization experiments for a small mathematical programming problem, the optimal design of a very small planar steel frame, and the optimization of a small sized three-dimensional steel building.

For the simple mathematical programming problem, the overheads associated with message passing in the DNC architecture are high in comparison to the mathematical computations required. The sequential implementation is faster. For the optimal design of the very small planar steel frame, approximately 70% of the computational work is devoted to finite element analyses. The sequential and distributed implementations have approximately the same speed. More than 95% of the total computational work is dedicated to finite element analy-

ses in the optimal design of the three-dimensional steel building. A maximum speedup of 3.26 was achieved when five simulators were employed for the finite element computations. 3.26 is not close to the theoretical maximum speedup of 5. Indeed, it is our observation that computational speedup is affected by a complex interaction of at least four factors. The factors are: (a) the number of remote simulators, (b) the problem solving strategy within FSQP-DIS, (c) the number of design variables in an optimization problem, and (d) strategies used by DNC to assign tasks to remote simulators.

Factor (b) is related to the performance metric of sequential dependency; that is the fraction of steps in an algorithm that must be sequential because previous results are needed before a particular computation can commence. Amdahl's law [1] states that if f_s and f_p are the fractions of sequential and parallel computation, such that $f_s + f_p = 1$, then as the number of available remote workstations becomes large the maximum speedup that can be obtained is $1/f_s$. However, when configurations of DNC are limited to a moderate number of identical remote workstations (let's say `N`), measured speedup cannot exceed `N` even if $f_s \ll$ `1/N`. Although it is theoretically possible to configure DNC with 20+ workstations, it is unlikely that a user would ever want to manually setup more than 10 remote simulators (see Section 2.10).

Recall that an iterate of FSQP-DIS contains three main tasks: (1) computation of the Jacobian matrix, (2) a single additional analysis for $\tilde{d}$, and (3) the step length computation. When `N` is large enough so that all components of Step

(1) may be executed in time Δt, and Step (3) in time Δt, Steps (1)-(2)-(3) will still take at least $3\Delta t$ no matter how many remote workstations are available. The practical implementation of FSQP-DIS is further complicated by factor (c) interacting with factor (a). If an optimization problem has `M` design variables (let's say 5), then in the current DNC environment there appears to be little computational advantage in having more than `5` identical remote workstations; each workstation is assigned one structural analysis during the computation of partial derivatives in the Jacobian matrix. Moreover, the numerical experiments indicate that only occasionally are more than 3-4 trials needed in the steplength computation, so again, Step (3) of FSQP-DIS would rarely exploit the processing power of 5-10 workstations without wasting analyses. Rather than automatically compute `M` trial step lengths, a better strategy might be to select only 1-2 trial points, and employ the remaining workstations for Step (1) - i.e. gradient computations - in the next iteration. Some ideas on how to proceed with the speculative gradient evaluations are given below.

5.2 Future Work

Future work should focus on reducing the time scale $3\Delta t$. One option is to replace the engineering workstations with very fast parallel computers. This is a long term goal. In the meantime it is recommended that extensions to this work concentrate on adjustments to the FSQP-DIS algorithm - factor (b) - and implementations of smart dispatcher threads - factor (d).

5.2.1 Nonmonotone Search Strategies

A variant of FSQP-AL is the recently developed FSQP-NL algorithm [40]. FSQP-NL uses a nonmonotone line search to force a decrease of the maximum value of the objective functions within either at most four iterations (if there are nonlinear constraints [11]), or at most three iterations otherwise [19]. The nonmonotone line search scheme achieves superlinear convergence with no bending of the search direction, and no function evaluations at auxiliary points. It is recommended that FSQP-AL be replaced by FSQP-NL, thereby reducing the timescale from $3\triangle t$ to $2\triangle t$.

5.2.2 Speculative Gradient Evaluation

This idea arises from the fact that once stepsize $t_k = 1$ satisfies the line search requirements, it is most likely that $t_k = 1$ for the majority of iterations in the optimization process. Instead of wasting simulations on trial points with smaller stepsizes, as has been done in this study, a better strategy might be to start computing gradients needed for the direction calculation in the next iteration even before the previous iteration is complete. This strategy is called speculative gradient evaluation because there is no guarantee the design point for $t_k = 1$ will be acceptable. If $t_k = 1$ fails, then trial points $t_k = 1/2$, $t_k = 1/4$, and so on must be simulated. For a complete discussion, see reference [33].

5.2.3 Smarter Dispatcher Threads

The implementation of dispatcher threads in DNC is naive in the sense that items are fetched from the task queue on a first-in first-out basis. For the structural analyses described in Chapter 4, this strategy has been acceptable because all of the remote simulators were running at approximately the same speed, and each of the simulation tasks required approximately the same computational work. However, in other applications [3, 6] the amount of computational work may vary from task to task. The proposed *smart dispatcher threads* should be able to monitor the computational speed of the remote simulators, and match tasks with the most computational work with the fastest simulators. Some ideas and methods of implementing the assignment are given in Bertsekas and Tsitsiklis [9].

BIBLIOGRAPHY

[1] Amdahl G. M. Validity of the Single-Processor Approach To Achieving Large Scale Computing Capabilities. In *AFIPS Conference Proceedings, Atlantic City, N.J.*, volume 30, April 1967.

[2] Austin M. A. CSTRUCT : An Interactive Computer Environment for the Design and Analysis of Earthquake Resistant Steel Structures. Technical Report Report No UCB/EERC-87-13, Earthquake Engineering Research Center, University of California, Berkeley, September 1987.

[3] Austin M. A. High Order Integration of Smooth Dynamical Systems: Theory and Numerical Experiments. International Journal of Numerical Methods in Engineering; Submitted November, 1991.

[4] Austin M. A., Pister K. S., Mahin A. S. Beyond DELIGHT.STRUCT : Current Research and Software Development. In *Structures Congress 87, ASCE, Orlando*, August 1987.

[5] Austin M. A., Pister K. S., Mahin A. S. Probabilistic Limit States Design of Moment-Resistant Frames under Seismic Loading. *Journal of the Structural Division, ASCE*, August 1987.

[6] Austin M. A., Voon B. K. Development of a Distributed Computing Environment for Optimization-Based CAD of Structures. In *Computer Aided Optimum Design of Structures OPT/91*, June 1991.

[7] Balling R. J., Ciampi V., Pister K. S., Polak E. Optimal Design of Seismic-Resistant Planar Steel Frames. Technical Report Report No EERC 81-20, Earthquake Engineering Research Center, University of California, Berkeley, December 1981.

[8] Balling R. J., Pister K. S., Polak E. DELIGHT.STRUCT: A Computer-Aided Design Environment for Structural Engineering. *Computer Methods in Applied Mechanics and Engineering*, pages 237–251, January 1983.

[9] Bertsekas D. P., Tsitsiklis J. N. *Parallel and Distributed Computation : Numerical Methods.* Prentice-Hall Inc, New York, NY., 1989.

[10] Birman K., Cooper R., Joseph T., Kane K., Schmuck F. The ISIS System Manual : Version 1.2. Technical report, Department of Defense Advanced Research Projects Agency, August 1989.

[11] Bonnans J. F., Panier E., Tits A. Avoiding the Maratos Effect by Means of a Nonmonotone Line Search. II. Inequality Constrained Problems – Feasible Iterates. Technical Report Technical Report SRC-TR-89-42r1, Systems Research Center, University of Maryland, College Park, MD 20742, 1989.

[12] Butala D., Choi K. Y., Fan M. K. H. Multiobjective dynamic optimization of semibatch free radical copolymerization process with interactive cad tools. Technical Report Technical Report TR-87-166, Systems Research Center, University of Maryland, College Park, MD 20742, 1987.

[13] Byrne R. H. Interactive Graphics and Dynamical Simulation in a Distributed Processing Environment. Master's thesis, University Of Maryland, College Park, 1990.

[14] Chang K. H., Santos J. L. T. Distributed Design Sensitivity Computations on a Network of Computers. *Computers and Structures*, 37(3):265–275, 1990.

[15] Coulouris G. F., Dollimore J. *Distributed Systems Concepts and Design.* Addison-Wesley Publishing Company, 1988.

[16] Doeppner T. W. Jr. A Threads Tutorial. Technical Report CS-87-06, Computer Science Technical Report, Brown University, 1987.

[17] Farhat C., Wilson E. L. A New Finite Element Concurrent Computer Architecture. *International Journal for Numerical Methods in Engineering*, 24:1771–1792, September 1987.

[18] Feldman S. I., Gay D. M., Maimone M. W., Schryer N. L. A Fortran to C Converter. Technical Report 149, Computer Science Technical Report, AT & T Bell Laboratories, 1990.

[19] Grippo L., Lampariello F., Lucidi S. A Nonmonotone Line Search Technique for Newton's Method. SIAM J. Numer. Anal., 1986.

[20] Herendeen D. L. Parallel Processing and FEM : Fullfilling the Promise. *Finite Elements in Analysis and Design*, 4:193–202, 1988.

[21] Johnson S. C. YACC-Yet Another Compiler Compiler. Computer Science Technical Report 32, ATT Bell Lab, Murray Hill, N.J., 1975.

[22] Johnsson S. L., Mathur K. K. Data Structures and Algorithms for the Finite Element Method on a Data Parallel Supercomputer. *International Journal for Numerical Methods in Engineering*, 29:881–908, 1990.

[23] Kernighan B. W., Pike R. *The UNIX Programming Environment.* Prentice-Hall Software Series, 1984.

[24] Kernighan B. W., Ritchie R. *The C Programming Language.* Prentice-Hall Software Series, 1978.

[25] Lampson B. W., Paul M., Siegert H. J. *Distributed Systems - Architecture and Implementation.* Springer-Verlag, 1983.

[26] Magee J. N., Cheung S. C. Parallel Algorithm Design for Workstation Clusters. *Software-Practice and Experience*, 21:235–250, March 1991.

[27] Martin B. E., Pedersen C. H., Bedfort-Roberts J. An Object-Based Taxonomy for Distributed Computing Systems. *IEEE Computer*, pages 17–66, August 1991.

[28] Nye W. T., Riley D. C., Sangiovanni-Vincentelli A. L., Tits A. L. DELIGHT.SPLICE: An Optimization-Based System for the Design of Integrated Circuits. *IEEE Trans. CAD Integrated Circuits and Systems, CAD-7*, pages 501–520, 1987.

[29] Nye W. T., Tits A. L. An Application-Oriented, Optimization-Based Methodology for Interactive Design of Engineering Systems. *International Journal of Control*, 43(6):1693–1721.

[30] Panier E. R., Tits A. L. On Combining Feasibility, Descent and Superlinear Convergence in Inequality Constrained Optimization. 1989. to be appear.

[31] Powell M. J. D. A Fast Algorithm for Nonlinearly Constrained Optimization Calculations, 1978. Numerical Analysis, Dundee, 1977, Lecture Notes in Mathematics 630, G.A. Watson, ed., Springer-Verlag, 14 4–157.

[32] Rees S. A., Black J. P. An Experimental Investigation of Distributed Matrix Multiplication Techniques. *Software-Practice and Experience*, pages 1041–1063, October 1991.

[33] Schnabel R. B. Concurrent Function Evaluations In Local And Global Optimization. *Computer Methods In Applied Mechanics And Engineering*, pages 537–552, 1987.

[34] Singhal M. Distributed Computing Systems. *IEEE Computer*, pages 12–15, August 1991.

[35] Sondhi J. S. Development of C-Based Program called FERA - Finite Element and Rigid-body Analysis. Master's thesis, University Of Maryland, College Park, 1991.

[36] Stevens W. R. *Unix Network Programming.* Prentice-Hall Software Series, 1990.

[37] Sun Microsystems, Inc. *Sun Manual, Copyright 1982*, 1988.

[38] Sun Microsystems, Inc. *SunView 1 Programmer's Guide, Copyright 1982*, 1988.

[39] Walker N. D. Authomated Design of Earthquake Resistant Multistory Steel Building Frames. Technical Report Report No. EERC 77-12, Earthquake Engineering Research Center, Univ. of Ca., Berkeley, Ca., May 1977.

[40] Zhou J., Tits A. L. Fast Feasible Direction Methods, With Engineering Applications. In *European Control Conference*, pages 194–199, Grenoble, Paris, July 1991. Hermès, Paris.

[41] Zhou J., Tits A. L. User's Guide for FSQP Version 2.0 - A Fortran Code for Solving Optimization Problems, Possible Minimax, with General Inequality Constraints and Linear Equality Constraints, Generating Feasible Iterates. Electrical Engineering Department and Systems Research Center, University of Maryland, College Park, MD 20742, 1991.