

Old Dominion University

ODU Digital Commons

Electrical & Computer Engineering Theses & Dissertations

Electrical & Computer Engineering

Spring 2005

Creating Software [Sic] Environments on an M-Node Beowulf Cluster to Execute Discrete-Event Simulations

Jermaine Fitz-Gerald Headley
Old Dominion University

Follow this and additional works at: https://digitalcommons.odu.edu/ece_etds



Part of the [Computer and Systems Architecture Commons](#), and the [Software Engineering Commons](#)

Recommended Citation

Headley, Jermaine F. "Creating Software [Sic] Environments on an M-Node Beowulf Cluster to Execute Discrete-Event Simulations" (2005). Master of Science (MS), Thesis, Electrical & Computer Engineering, Old Dominion University, DOI: 10.25777/11yf-hq97
https://digitalcommons.odu.edu/ece_etds/362

This Thesis is brought to you for free and open access by the Electrical & Computer Engineering at ODU Digital Commons. It has been accepted for inclusion in Electrical & Computer Engineering Theses & Dissertations by an authorized administrator of ODU Digital Commons. For more information, please contact digitalcommons@odu.edu.

**CREATING SOFTWARE ENVIRONMENTS ON AN M-NODE BEOWULF
CLUSTER TO EXECUTE DISCRETE-EVENT SIMULATIONS**

By

Jermaine Fitz-Gerald Headley

B.S in Computer Engineering May 2003, Old Dominion University

A Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

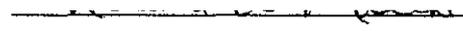
MASTER OF SCIENCE

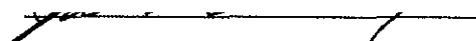
COMPUTER ENGINEERING

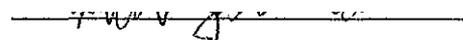
OLD DOMINION UNIVERSITY

May 2005

Approved by:


Roland R. Mielke (Director)


James Leathrum (Member)


Stephen A. Zahorian (Member)

ABSTRACT

CREATING SOFTWARE ENVIRONMENTS ON AN M-NODE BEOWULF CLUSTER TO EXECUTE DISCRETE-EVENT SIMULATION

Jermaine Fitz-Gerald Headley
Old Dominion University, 2005
Director: Dr. Roland R. Mielke

This thesis describes the development of a software tool that facilitates the creation of software environments that make a simulation tool execute k replications of an application program on several nodes of an M -node Beowulf cluster. It is assumed that each cluster-node consists of p processors. The p processors that are contained in the master cluster-node are termed *master processors*, and the p processors that are contained in a slave cluster-node are termed *slave processors*. The slave processors are used to execute the replications, while the master processors are dedicated to schedule the replications and process other housekeeping chores. For each slave cluster-node that is selected, P processors are specified to participate in the execution of the replications, where $1 \leq P \leq p$. The total slave processors selected to execute the replications is N . These slave processors are contained in the set Π with cardinality $|\Pi| = N$. Therefore, the k replications are executed concurrently if $k \leq N$. Otherwise, the k replications are grouped into batches that are executed concurrently as processors become available.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my committee members for their support and contributions throughout the development of this thesis. Without their patience and many hours of guidance on my research, the completion of this thesis would have not been possible. I would like to extend special thanks to my committee director, Dr. Roland R. Mielke, for his contributions. I am indebted to him for the encouragement and support he provided, his many hours of advising, and his untiring efforts spent editing the manuscript.

I would like to acknowledge all other individuals who were supportive of this thesis. It is not possible to list all, but be assured that you have my gratitude for your contributions.

TABLE OF CONTENTS

	Page
LIST OF TABLES	vii
LIST OF FIGURES	viii
NOTATION	x
CHAPTER I	
INTRODUCTION.....	1
1.1 Problem Statement.....	3
1.2 Thesis Contributions.....	4
1.3 Thesis Overview.....	5
CHAPTER II	
BACKGROUND.....	7
2.1 Beowulf Clusters.....	7
2.2 Message Passing Interface (MPI).....	10
2.3 Overview of Xface Tool.....	12
CHAPTER III	
PARAMETRIC JOBS.....	15
3.1 Scheduling Replications.....	15
3.2 Parametric Speedup.....	16
3.3 Job Environments.....	18
3.4 Maintaining the Independence Among Replications.....	22
3.5 Parametric Job Description.....	24

CHAPTER IV

VARIANCE REDUCTION JOBS.....	26
4.1 Variance Reduction Analysis.....	26
4.2 Common Random Numbers.....	30
4.3 Antithetic Variates.....	31
4.4 Synchronizing the Random Numbers.....	31
4.5 VR Job Descriptions.....	33

CHAPTER V

PARTITIONED JOBS.....	35
5.1 The Partitioning Problem.....	36
5.2 Task Precedence Graph Model.....	37
5.3 The Scheduling Problem.....	39
5.4 Scheduling Tasks within a Replication.....	40
5.5 Speedups.....	43
5.6 Partitioned Job Environment.....	50
5.7 Partitioned Job Description	53

CHAPTER VI

IMPLEMENTATION.....	55
6.1 Implementing the Job Environments.....	59
6.2 Process Interaction on Master Processor.....	64
6.3 Process Interaction on Slave.....	66

CHAPTER VII

CASE STUDY.....	69
7.1 Arena Application-specific Environment.....	69
7.2 Parametric Case Study.....	71
Job Description.....	73
Results.....	74

7.3 Variance Reduction under AV Case Study.....	77
Job Description.....	78
Results.....	78
7.4 Partitioned Job Case Study.....	82
Job Description.....	87
Results.....	87
CHAPTER VIII	
CONCLUSION.....	92
8.1 Future Work.....	94
REFERENCES.....	98
APPENDICES	
A. APPLICATION START-UP SCRIPTS.....	102

LIST OF TABLES

Table 5-1. Initial States of Processors.....	45
Table 5-2. Initial States of Tasks.....	46
Table 5-3. States of Processors after First Loop through Algorithm 5-1.....	46
Table 5-4. States of Tasks First Loop through Algorithm 5-1.....	47
Table 5-5. States of Processors Second Loop through Algorithm 5-1.....	47
Table 5-6. States of Tasks after Second Loop through Algorithm 5-1.....	48
Table 5-7. States of Processors after Third Loop through Algorithm 5-1.....	48
Table 5-8. States of Tasks Third Loop through Algorithm 5-1.....	48
Table 5-9. States of Processors Fourth Loop through Algorithm 5-1.....	49
Table 5-10. States of Tasks Fourth Loop through Algorithm 5-1.....	49
Table 5-11. States of Processors after Fifth Loop through Algorithm 5-1.....	49
Table 5-12. States of Tasks after Fifth Loop through Algorithm 5-1.....	49
Table 7-1. Parametric Case Study Job Description.....	73
Table 7-2. Model 02-a Job Description.....	79
Table 7-3. Model 02-b Job Description.....	79
Table 7-4 Partitioned Case Study Job Description.....	88

LIST OF FIGURES

Figure 2-1. Topology of Balrog’s Inter-connect.....	9
Figure 2-2. Remote Connections Among MPI Daemons.....	11
Figure 2-3. MPI Daemons.....	11
Figure 3-1. Master-Slave and Slave-Slave Remote Communications.....	19
Figure 3-2. Virtual Network Topology.....	19
Figure 3-3. Job Environment Created for Windows-based Applications.....	21
Figure 3-4. Job Environment Created for Linux-based Applications.....	21
Figure 3-5. Template of Streams File.....	23
Figure 3-6. Parametric Job Description Forms.....	25
Figure 4-1. M/M/s, s = 1, 2, Models.....	28
Figure 4-2. VR-AV Job Description Form.....	33
Figure 4-3. VR– CRN Job Description Form.....	34
Figure 5-1. DAG Model.....	38
Figure 5-2. Fork and Join Structures	41
Figure 5-3. Sample DAG Model.....	45
Figure 5-4. Ghant Charts Illustrating Optimum Schedule Length.....	50
Figure 5-5. Network of Processors for N = 16 and n = 4.....	52
Figure 5-6. Partitioned Job Description Forms.....	54
Figure 6-1. Hereditary Relationships Among Core Xface Processes on Master.....	57
Figure 6-2. Hereditary Relationships Among Core Xface Processes on Slave.....	57
Figure 6-3. Remote Communication between Master and Slave Processors... ..	61
Figure 6-4. Remote Communication between Master and Leaders.....	63
Figure 6-5. Communication between Leader and Non-leaders in Γ_1	64
Figure 6-6. IPCs on Master Processor.....	65
Figure 6-7. IPCs on Slave Processor in Parametric and VR Jobs Environment.....	67
Figure 6-8. IPCs on Leaders in Partitioned Job Environment.....	67
Figure 6-9. IPCs on Non-leader in Partitioned Job Environment.....	68
Figure 7-1. Arena Application-specific Execution Environment.....	70
Figure 7-2. Arena Model 01.....	72

Figure 7-3. Plots of Theoretical and Actual Execution Times vs. Number of Processors.....	76
Figure 7-4. Plot of Total Overhead vs. Number of Processors.....	76
Figure 7-5. Plot of Speed-up vs. Number of Processors.....	76
Figure 7-6. Arena Model 02.....	77
Figure 7-7. Model 02-a Average Delays.....	80
Figure 7-8. Model 02-b Average Delays.....	81
Figure 7-9. Comparison of Average Delays observed in Model 02 (a) and (b).....	82
Figure 7-10. Model 03: Simulation Model of Production Line.....	83
Figure 7-11. General Structure of Simulation Applications.....	84
Figure 7-12. DAG Representation of Model 03.....	86
Figure 7-13. Mean Times to execute DAG.....	86
Figure 7-14. Execution Times versus Number of Sub-clusters for $m = 1$	90
Figure 7-15. Execution Times versus Number of Sub-clusters for $m = 2$	90
Figure 7-16. Execution Times versus Number of Sub-clusters for $m = 3$	90
Figure 7-17. Comparison of Actual Curves for $m = 1, 2, 3$	91
Figure 7-18. Speedups per Replication versus the Number of Sub-clusters.....	91
Figure 7-19. Comparison of Total Speedups for $m = 1, 2, 3$	91
Figure 8-1. Future Setup on Slave Cluster-node.....	95
Figure 8-2. Future Setup 1 on Slave Cluster-node.....	97
Figure 8-3. Future Setup 2 on Slave Cluster-node.....	97

NOTATION

Symbol	Description
A_j	Inter-arrival time between the $j-1^{\text{st}}$ and j^{th} entities
C_i	The i^{th} alternative configuration of a system
c	Number of alternate configurations of a system
$\text{Cov}\{X_1, X_2\}$	Covariance of the random variables X_1 and X_2
D_j	Delay in queue of the j^{th} entity
$d_i(l)$	Average delay in queue resulting from the i^{th} replication of a model with sample size l
F_A	Probabilistic distribution of inter-arrival times
F_S	Probabilistic distribution of service times
$E\{X\}$	Expected value of random variable X
e_{ij}	Directed-edge connecting tasks T_i and T_j
k	Number of replications
l_i	Sample size of configuration C_i
M	Total number of cluster-nodes in the Beowulf
m	The cardinality of the set Γ , $m = \Gamma $
N	The cardinality of the set Π , $N = \Pi $
n	Number of sub-clusters
P_i	Processor in the set Π
P	Number of processors in a slave cluster-node specified to participate in the execution of the replications
p	Total number of processor in each cluster-node
R	Predefined rule used to place the processors in Π into Γ_i , $1 \leq i \leq n$
r	Number of distinct random number streams defined in the model under execution
S_j	Service time of j^{th} entity
$S_X^2(\mathbf{k})$	Sample variance of X_1, X_2, \dots, X_k

T_i	The i^{th} task in a partitioned program
T_S	Sequential time to execute k replications on a single processor
T_P	Parallel time to execute k replications on N processors
U	Random sequence that drives the k replications
U^i	The i^{th} stream in U
U_k	Random number drawn from the sequence U
$\text{Var}\{X\}$	Variance of random variable X
$W(e_{ij})$	Weight of edge e_{ij}
$W(T_i)$	The node weight or sequential execution time required to execute task T_i
X	Random variable
$\bar{X}(k)$	Sample mean of X_1, X_2, \dots, X_k
Γ	Sub-cluster comprising m slave processor
Π	Set of processors used to execute a job.
μ_X	True expected value of X
ϕ	The total number of tasks in a partitioned program
τ	Total speedup given by the ratio of T_S to T_P
τ_{1D}	Speedup in the first dimension
τ_{2D}	Speedup in the second dimension
$\lfloor a \rfloor$	The floor function invoked on a
$a \bmod b$	The remainder of the division $\frac{a}{b}$
$a \subset b$	a is contained within b
$\{\}$	The empty set

CHAPTER I INTRODUCTION

The implementation of most commercial simulation packages is such that the replications of an application program are executed sequentially on a single processor. Oftentimes in practice, the application program must be replicated many times to improve the precision in the results. If the replication lengths of the simulation are short, taking several milliseconds or even seconds, performing several hundred replications will not take a long time. However, if the replication lengths are long, taking several minutes or even hours, the simulation could take several days to complete. For very long simulations, the total simulation time can be reduced dramatically if the replications are executed concurrently on a parallel platform. Several supercomputing platforms are available, ranging from high-end computing machines that are very sophisticated but expensive to lower-end machines that are not as sophisticated but much cheaper. A Beowulf cluster is an inexpensive low-end supercomputer that provides computing power comparable to that of some sophisticated high-end supercomputer. If each cluster-node consists of p processors, an M -node Beowulf cluster with one master cluster-node and $(M-1)$ slave cluster-nodes can be utilized to execute at most $p(M-1)$ replications concurrently on the slave processors while the master processors are dedicated to schedule the replications. If the number of replications is k and N processors are selected to execute the replications such that $k > N$, the replications are grouped into batches, and the replications within a batch are executed concurrently. The resulting speed-up in the total execution time will be considerable but not N fold since overhead is introduced in each batch execution time.

Several problems are introduced when a simulation application designed to execute sequentially on a single machine is ported to the parallel platform of a Beowulf cluster. First, the operating system on a majority of Beowulf clusters is a version of the Linux kernel because Linux is freely available and cheap to maintain. The problem is that most current commercial simulation packages are designed for Windows platforms. Due to the differences between the two platforms, a middle layer between the Linux operating

system and the simulation application is required. The purpose of the middle layer is to emulate a Windows environment. Both freeware and cheap software tools that emulate Windows environments on top of Linux platforms are available. Therefore, one only needs to search for an efficient Windows emulator that supports the application of choice. However, running Windows-based applications on an emulator introduces a new problem. Certainly, the application will run slower on the emulated Windows platform than it would on a native Windows platform. Therefore, the nature of the problem introduced is to determine if it is worth the efforts to run a Windows-based application on the emulated Windows platform. The solution to this problem depends on whether the speedup achieved to execute the replications concurrently on the slave processors is enough to compensate for the extra time that is required to run the application on the emulated platform.

A second major problem is that the random numbers that drive the simulation replications must be controlled when the simulation is executed on the parallel platform. Simulation packages that are designed to execute sequentially on a single processor automatically control the independence of random numbers between replications by using the last random number U_j generated in the i^{th} replication as the seed in the $i+1^{\text{st}}$ replication. This method of random number control will not work when the simulation is migrated to the parallel platform of the Beowulf. Therefore, a different method that does not create any sequential dependence among the replications is required. Commercial simulation packages use a random sequence U with a very long cycle-length to drive the replications. The long cycle-length guarantees that U will take a long time to recycle during long simulation runs. The sequence U is oftentimes segmented into many smaller fixed-length sub-sequences called streams. For example, in the Arena-Version 5.00.2 [1] simulation package, the cycle length of U is 3.1×10^{57} , and there are 1.8×10^{19} separate streams each of length 1.7×10^{38} ; and each stream is further subdivided into 2.3×10^{15} sub-streams each of length 7.6×10^{22} . The stream U^i is specified by the index i . Segmenting U allows several different streams to be used at the various random points in the simulation model by simply specifying the stream index to use with a particular probability distribution at a given random point. The method of random number control

on the parallel platform that is described in the thesis utilizes the assignment of separate streams to the random points in the simulation model to insure that all k replications can begin concurrently. Furthermore, the streams used in the replications are carefully controlled to prevent stream overlapping. The danger of driving the replications with overlapping streams is that the independence in the results produced by the replications is not guaranteed.

1.1. Problem Statement

This thesis describes the development of a software tool that facilitates the creation of software environments that make a simulation tool execute k replications of an application program on several nodes of an M -node Beowulf cluster. It is assumed that each cluster-node consists of p processors. The p processors that are contained in the master cluster-node are termed *master processors*, and the p processors that are contained in a slave cluster-node are termed *slave processors*. The slave processors are used to execute the replications, while the master processors are dedicated to schedule the replications and process other housekeeping chores. For each slave cluster-node that is selected, P processors are specified to participate in the execution of the replications, where $1 \leq P \leq p$. The total slave processors selected to execute the replications is N . These slave processors are contained in the set Π with cardinality $|\Pi| = N$. Therefore, the k replications are executed concurrently if $k \leq N$. Otherwise, the k replications are grouped into batches that are executed concurrently as processors become available.

One of the software environments that the tool creates support variance reduction under two frequently used variance reduction techniques (VRTs), Antithetic Variates (AV) and Common Random Numbers (CRN). In addition, a software environment is created to execute a single replication on m processors. The application programs that are executed in the latter environment must be partitioned into ϕ inter-dependent tasks, and a task precedence graph model must be used to represent the tasks. The tasks are executed on the m processors according to the dataflow [2] strategy specified by the task graph. The software environments the tool creates are not application-specific. They are robust

enough to support several simulation applications, and both Windows-based and Linux-based applications are supported.

The software tool that is developed is called *XFace*. The XFace tool provides a front-end graphical user interface (GUI) to the user. The GUI allows the user of the tool to easily load and execute k replications of an application program on N slave processors in the Beowulf cluster. The GUI also allows the user to monitor the execution status of the application program. The back-end of the tool creates the software environments, executes the replications, and monitors the processing of the replications on each slave processor.

1.2. Thesis Contributions

There are several job scheduling tools that are designed to schedule batch jobs to the machines in a Beowulf cluster. The common objective of these tools is to reduce the execution time of parallel applications by harnessing the power of several cluster-nodes. The XFace tool falls into this category of applications, but it is unique in the sense that that it seeks to batch replications of simulation applications that were designed to execute sequentially on single-processor machines. The contributions of the tool developed in the thesis are as follows:

- The XFace tool creates software environments that control the random numbers driving the replications so that the independence of the results produced by the replications is guaranteed,
- An environment is created that supports the parametric execution of k replications of a simulation application on N processors,
- An environment is created that supports variance reduction under AV,
- An environment is created that supports variance reduction under CRN.
- An environment is created that supports the execution of each of the k replications on m processors in a dataflow strategy, and the concurrent execution of the k replications on n sub-clusters.

1.3. Thesis Overview

The organizational structure of the remaining portions of the thesis is as follows. Chapter 2 presents the background information. The chapter begins with a brief overview of Beowulf clusters. It then presents an overview of the Message Passing Interface (MPI) that is used to implement remote-process communication (RPC) among processes on remote processors. The chapter concludes with an overview of the XFace tool.

Chapter 3 presents the implementation concepts of parametric jobs. The main scheduling algorithm used to schedule iterations is presented in Section 3.1. The equation used to estimate the parametric speedup in the total execution time when the program is executed in the software environments created by the XFace tool is developed in Section 3.2. The software environments are presented in Section 3.3. These software environments apply to all three job-types. However, the environments do not fully support partitioned jobs. Therefore, they are extended in Section 5.6 to support partitioned jobs. Section 3.4 presents the method proposed to maintain the independence among the replications executed on the parallel platform. Finally, the job description forms used to submit parametric jobs are described in Section 3.5.

Chapter 4 presents the variance reduction jobs. Section 4.1 presents a general overview of the analysis of variance reduction on the outputs of a simulation. Sections 4.2 and 4.3 present the concepts behind CRN and AV, respectively. Detailed analyses of CRN and AV can be found in [3], [4], [5]. Section 4.4 stresses the importance of synchronizing the random numbers used in the simulation when CRN or AV is applied. Finally, the job description forms used to submit variance reduction jobs are described in Section 4.5.

Chapter 5 presents the design concepts of partitioned jobs. The partitioning problem and scheduling problem are briefly addressed in Section 5.1 and Section 5.3, respectively. The task precedence graph model used to represent partitioned programs is described in Section 5.2. The scheduling algorithm used to schedule the tasks in the task graph to the

m processors in a sub-cluster is presented in Section 5.4. The equation used to estimate the speedup in the total execution time of partitioned jobs is presented in Section 5.5. The software environments described in Section 3.3 are extended in Section 5.6 to support partitioned jobs. Finally, the job description forms used to submit partitioned jobs are described in Section 5.7.

Chapter 6 presents the implementation of the XFace tool. The core processes and shell scripts in the XFace implementation are presented in Section 6.1. The implementations of the virtual networks described in Chapters 3 and 5 are then presented in Section 6.2. The interaction among processes on the master processor is described in Section 6.3. Finally, the interactions among processes on each slave processor during the job execution phase are described in Section 6.4.

Chapter 7 presents three case studies, one for each job-type. The Arena application is used in the first two case studies. Section 7.1 presents the job environment that is created specifically to execute Arena programs. The parametric case study is then presented in Section 7.2. This is followed by the presentation of the variance reduction under AV case study in Section 7.3. Finally, the partitioned case study is presented in Section 7.4.

Finally, Chapter 8 presents concluding remarks and identifies future work to enhance the XFace tool.

CHAPTER II BACKGROUND

This chapter presents brief background information on Beowulf clusters and the MPI protocol. The chapter also gives a general overview of the XFace tool.

2.1. Beowulf Clusters

Since the Beowulf [6], [7] project began at NASA in 1994, the main goals of the Beowulf have always been to explore the possibilities of building an efficient High Performance Computing (HPC) system that is low cost and easily upgradeable with minimal efforts. Therefore, the components of the cluster-nodes and network switches in the Beowulf are commercially available off-the-shelf components that are not vendor-specific. The idea that brought the Beowulf into existence is adopted from the Network of Workstations (NOW) project [7], [8], which consists of a network of workstations with each workstation consisting of several high-end, powerful microprocessors. The Beowulf exploits machines with cheaper, less powerful microprocessors that are intended for the PC market. Thus, the result is a system that provides efficient high performance computing at low cost. The high performance to cost ratio of the Beowulf is very appealing to the HPC community. As a result, the Beowulf is replacing several expensive, sophisticated high performance supercomputers in certain application areas. Architecturally, the Beowulf is a distributed memory supercomputer that is used to parallelize the execution of large applications. However, several variations of this architecture exist. In some Beowulf architectures, the cluster-nodes are equipped with at least two processors that provide symmetric multiprocessing (SMP). The processors in an SMP cluster-node are interconnected via shared-memory. Another variation involves cluster-nodes that have one processor dedicated for processing and a second dedicated for communication on the private network connecting the cluster-nodes. However, in the latter variation, applications are oftentimes written that utilize the second processor for processing.

The cluster-nodes in a Beowulf are interconnected via a private interconnect. Several topologies, such as Star, Mesh [9], and Crossbar [10], are used for the

interconnect. One major disadvantage of Beowulfs is that the speeds of the switches in the interconnect are not as fast as the speeds of the processors in the cluster-nodes. Consequently, the bandwidth of the private interconnect results in long communication latencies, rendering the bandwidth the bottleneck in the computing performance of Beowulfs. As switching technology advances, fast network protocols are being developed that exploit the increasing speeds of the interconnect switches to provide high bandwidth on the order of gigabits per second. For example, optical Gigabit Ethernet providing up to 10 gigabit per second bandwidth have been reported [11], [12]. However, as the speeds of the interconnect switches grow, the speeds of the processors in the cluster-nodes grow at even a faster rate. Therefore, the resultant effect is that the processing speed to communication bandwidth ratio continues to grow. Tremendous research initiatives aimed at developing sophisticated high-speed communication protocols that will increase the bandwidth of the interconnect are in progress. For example, the Virtual Interface Architecture (VIA) project [13] is one such initiative. The VIA trades reliability for speed and reduces the kernel-level communication overhead associated with message passing by giving applications direct access to the network cards in the cluster-nodes.

A Beowulf system can be either heterogeneous [14] or homogeneous. In heterogeneous systems, the configuration of the Beowulf is such that multiple operating systems may run on the cluster-nodes, and the cluster-nodes may contain processors with different architectures. On the other hand, in homogenous system, the cluster-nodes are very similar. All cluster-nodes run the same operating system, and the processors all have the same architecture. Homogenous Beowulfs are the more widespread of the two.

The Linux kernel is the dominant operating system that runs on most Beowulfs, mainly due to the fact that the Linux kernel is an open-source kernel that is freely available to the general public. In addition, the Linux kernel fully supports Beowulf systems, and tremendous amounts of freeware are developed for clusters running the Linux kernel.

The architecture of the Beowulf (Balrog) on which the XFace tool was developed and also on which the case studies presented in Chapter 7 were conducted is as follows. Balrog is a 32-node dual processor system and consists of:

- 32 AMD 760MPX motherboards,

- Each cluster-node has dual AMD Athlon MP 2600 processors running at 2.1 GHzertz,
- The master cluster-node has 2.048 GByte DDR PC2100 ECC Registered System Memory, and 60 GByte EIDE Hard Drive at 7200RPM,
- Each slave cluster-node has 1.024 Gbyte DDR PC2100 ECC Registered System Memory, and 20 Gbyte EIDE Hard Drive at 7200RPM,
- HP Procurve 4108GL 36-port 10/100/1000 Gigabit Ethernet Switch for the private interconnect.

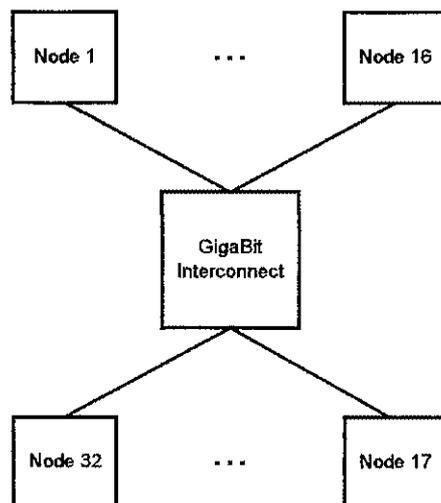


Figure 2-1. Topology of Balrog's Interconnect

The private interconnect in Balrog is based on the Star topology, as illustrated in Figure 2-1. Every cluster-node can setup direct connections with every other cluster-node.

The processes running on remote cluster-nodes use message-passing protocols such as MPI [15], [16] and PVM [17], [18] to communicate. The MPI message-passing library is used in the implementation of the XFace tool.

2.2. Message Passing Interface (MPI)

MPI is an industry standard message-passing protocol that is well supported on HPC systems, especially distributed memory systems. Portability and efficiency are two major features of the MPI message-passing protocol. Thus, MPI can be implemented on a wide range of HPC systems. The portable implementation of MPI is MPICH [19], and it enables MPI programs to be easily ported among different HPC systems. MPI maps MPI processes to the processors in MPI nodes, where the MPI nodes are the cluster-nodes that are utilized to run an MPI program. The same MPI process is executed on each processor in the MPI nodes. However, each MPI process is executed in a unique address space. Therefore, the global variables are stored in different physical locations in memory so that the global variables written by one MPI process are not seen by the other MPI processes. One or more MPI daemons run on each MPI node. The MPI daemons enable communication via message passing among both remote and local MPI processes. Figure 2-2 illustrates the remote connections among MPI daemons on 4 MPI nodes. Every MPI daemon can reach every other MPI daemon. The local MPI processes on an SMP MPI node can share the same MPI daemon, or an MPI daemon can be dedicated to each MPI process. In the case when the local MPI processes share the same MPI daemon, the local MPI processes communicate locally via shared memory, but use message passing to communicate with remote MPI processes. Figure 2-3 (a) illustrates the case when two local MPI processes share the same MPI daemon and use shared memory for local communication. In the case when a MPI daemon is dedicated to each local MPI process, the local MPI processes communicate via message passing, as illustrated in Figure 2-3 (b).

MPI processes are ranked with unique consecutive integers from 0 to N. On Beowulfs running Linux, MPI programs read a file named "machine.LINUX" at initialization to identify the cluster-nodes to be used as MPI nodes. Each cluster-node that will participate in the execution of an MPI program is listed in "machine.LINUX" in the format <cluster_node_alias[:P]>. The first entry specifies the alias of the cluster node. The second entry P is optional. If the cluster-node is an SMP node consisting of p

processors, P specifies the number of processors to use. If P is not specified, MPI uses a default value of one.

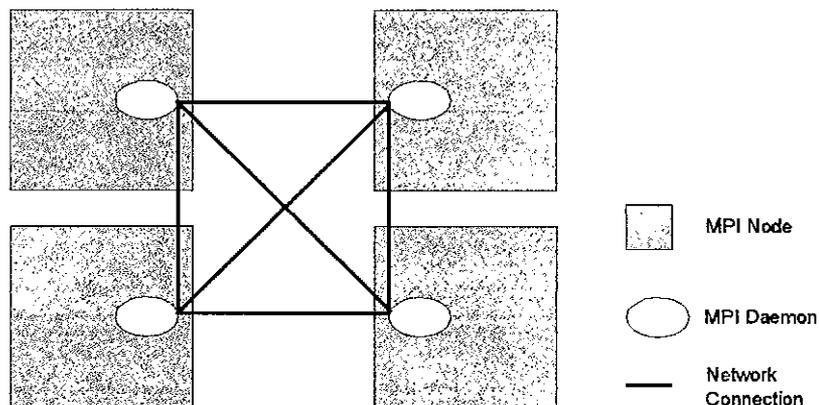


Figure 2-2. Remote Connections Among MPI Daemons

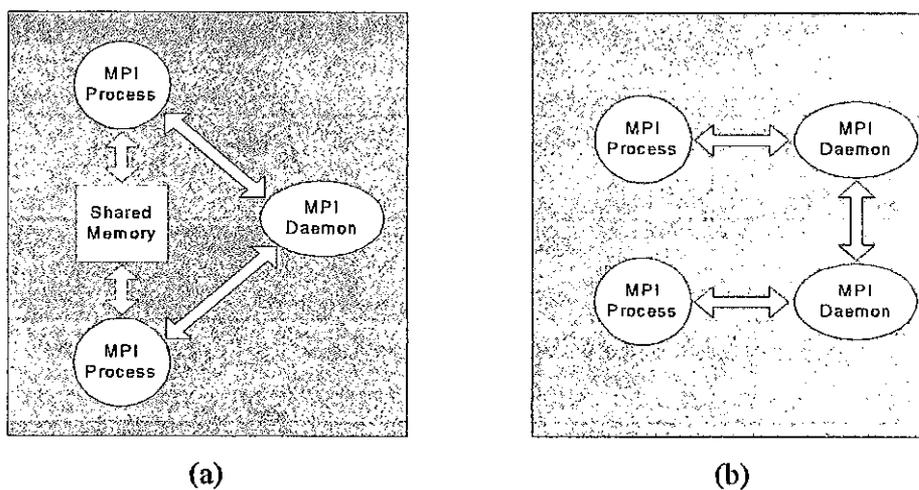


Figure 2-3. (a) Shared MPI Daemon (b) Dedicated MPI Daemons

2.3. Overview of XFace Tool

The XFace tool utilizes the parallel platform of a Beowulf together with the MPI messaging passing protocol to execute k replications of a simulation application concurrently on N processors. Hereafter, the terms “simulation program” and “program” are used interchangeably throughout the remaining portions of the thesis. A program that is replicated k times is naturally a parallel program since the replications are independent and thus can be executed concurrently. Furthermore, the program structure may consist of blocks of codes that may be executed in parallel according to the dataflow strategy that materializes during the execution of each replication of the program. Thus, it may be possible to achieve speed-ups in the overall program execution time in two ways. In the first, the speed-up in the program execution time is realized by concurrently executing the k replications on N processors, where each replication is executed on a single processor. In the second, the speed-up is realized by utilizing the implicit parallelism within the program by assigning m processors to execute each replication. Therefore, if the k replications are executed concurrently on n sub-clusters, such that the i^{th} sub-cluster contains m_i processors, both ways of achieving speedups in the program execution time are realized. For ease of reference, the speedup in the first way is termed *1st-dimension speedup*, and the speedup in the second way is termed *2nd-dimension speedup*.

The XFace tool executes jobs, each of which consists of a set of job files. Each set of job files contains one or more application programs that must be replicated k times, and input files that are read by the application programs. A job is executed in several iterations on the slave processors that are contained in the set Π . The replications are executed during the execution of the iterations. An *iteration* is the sequential execution of x replications, where $1 \leq x \leq k$. Therefore, the execution of each iteration requires resources such as memory, central processing unit time, and input files from which the replicated programs read data. Thus, iterations are executed as resources become available on the slave cluster-nodes.

Each job that is executed by the XFace tool has three phases: the job initiation phase, the job execution phase, and the job completion phase. The job initiation phase begins

after a job description has been submitted, and ends when the execution of the job is initiated. During the job initiation phase, the job environments are setup on the slave processors, and the job files are loaded onto the slave cluster-nodes. Also, to prevent overwriting the results of the replications, directories are created on each slave cluster-node to store the results. These directories are termed the *iteration directories*. The job execution phase follows the job initiation phase. The job execution phase is the time during which the iterations are being executed on the slave processors. It ends when the execution of the last iteration to execute is finished. After the job execution phase, the results produced by the iterations are scattered on the slave nodes. Therefore, the job completion phase is the time during which the results are gathered from the slave cluster-nodes onto the master cluster-node.

The XFace tool groups jobs into three types: parametric jobs, variance reduction jobs, and partitioned jobs. For parametric jobs, the tool creates the necessary software environment on the M-node Beowulf cluster to realize the 1st-dimension speedup. In this environment, the k replications of the program are executed on N processors, and each replication is executed on a single processor. Additionally, one replication, $x = 1$, is executed in each iteration. Hence, the total number of iterations is the same as the number of replications k . The program that is replicated in the iterations is submitted to the tool via a job description. The job description is a text file that specifies the job-type, the platform that the job is to execute on, the number of replications, the number of application programs submitted, pathnames to the job files, and the slave processors to execute the job. The details of the parametric job description are presented in Section 3.5.

The variance reduction (VR) job-type is an extension of the parametric job-type. The VR job-type adds support for the variance reduction techniques (VRTs) AV and CRN. AV and CRN are applied to simulations to reduce the variance of the output produced by the replications. A variance reduction job that employs AV as the VRT of choice is notated VR-AV, and a variance reduction job that employs CRN as the VRT of choice is notated VR-CRN. In VR-CRN jobs, c programs that models alternate configurations of a system are compared, where $c \geq 2$. To differentiate between the results produced by the programs, each program is assigned a distinct configuration

number. Each of the programs is replicated k times such that the corresponding replications of the programs are synchronized. Therefore, for VR-CRN jobs, the total number of iterations that is executed is the product $c \times k$.

In VR-AV jobs, one application program is submitted in the job description. The replications of the program are executed in pairs of two such that alternate replications are compared. One pair of replications is executed per iteration. Therefore, the program is replicated twice, $x = 2$, in each iteration that is executed. As a result, the total number of iterations that is executed in VR-AV jobs is the product $2 \times k$.

Finally, for partitioned jobs, the tool creates a software environment to realize both the 1st-dimension and 2nd-dimension speedups in the execution times. Each program that is executed in this environment is partitioned into ϕ interdependent tasks. The ϕ tasks are indexed arbitrarily with distinct task integers from 0 to $\phi - 1$. The program is replicated once, $x = 1$, in each iteration that is executed on sub-cluster Γ_i containing m_i slave processors. Therefore, the total number of iterations that is executed in partitioned jobs is equal to k .

The XFace tool uses MPI messages to schedule iterations among the slave processors. With the exception of partitioned jobs, the only messages passed during the job execution phase are scheduling information such as the iteration number, the iteration start time, the iteration end time, and the iteration execution time. Therefore, the total execution times of the parametric and VR jobs that the XFace tool executes are not affected greatly by the communication latencies associated with the private interconnect. However, since data must be passed among the m processors that execute each replication of a partitioned job, the communication latencies will greatly affect the total job execution time of partitioned jobs; but, with proper load balancing, the data passed among processors can be minimized.

CHAPTER III PARAMETRIC JOBS

Consider a simulation program that is replicated k times, where the k replications are independent. That is, the i^{th} replication of the program is driven by unique random number streams U^i , $i = 1, 2, \dots, k$. The independence property of the k replications allows us to execute concurrently all k replications of the program provided that at least k processors are assigned to execute the program. The objective of the parametric feature of the XFace tool is to reduce the overall execution time of the program by dynamically scheduling the k replications among the N processors that are selected to execute the program. The replications are scheduled according to the availability of the N processors.

3.1. Scheduling Replications

When the replications are executed on the parallel platform of the Beowulf, overhead is charged to distribute the executions of the k replications over the N processors selected to execute the program. The prime contributors to this overhead are the time to execute the scheduling algorithm used to schedule the iterations, the times to communicate the iteration numbers from the master processor to the slave processors during scheduling, and the overhead to execute the application on the emulated platform. The total overhead injected into the execution time can be reduced by designing a scheduling algorithm that incurs a small overhead with respect to the total sequential execution time of the program, and by carefully choosing a communication medium that allows for efficient, inexpensive communications among the processors. The main scheduling algorithm that is used in the XFace tool is listed as Algorithm 3-1.

Algorithm 3-1. Main Scheduling Algorithm

Begin

1. For each processor P_j , $j = 1, 2, \dots, N$ in II , mark P_j idle. Set $i = 0$.
2. While $i < k$. do

2.1 For each idle processor P_j in Π , do

i. Schedule iteration i to P_j .

ii. Set $i = i + 1$, and mark P_j busy.

End for

2.2 Wait on any busy processor to finish. Mark each finished processor P_j idle.

(Do background work while waiting.)

End while

End

Step 1 in Algorithm 3-1 is the initialization step, which can be done in $O(1)$ time. Therefore, the total time complexity of Algorithm 3-1 with respect to the number of iterations scheduled to the slave processors is $O(k)$.

3.2. Parametric Speedup

Let T_R be the replication time to execute one replication of a program that is replicated k times. Define the sequential execution time T_S of the program as the time to sequentially execute the k replications on a single processor, and the parallel execution time T_P of the same program as the time to execute concurrently the k replications on N processors. Theoretically, T_S and T_P are related to T_R as follows.

$$T_S(T_R, k) = kT_R \quad (3-1)$$

$$T_{P_{ideal}}(T_R, k, N) = \left(\left\lceil \frac{k}{N} \right\rceil + S(k, N) \right) T_R, \quad (3-2)$$

$$\text{where } S(k, N) = \begin{cases} 0, & k \bmod N = 0 \\ 1, & k \bmod N \neq 0 \end{cases}$$

In Equation (3-2), $\lfloor a \rfloor$ denotes the floor of a , and the expression $a \bmod b$ denotes the remainder of the integer division $\frac{a}{b}$. Equations (3-1) and (3-2) are valid only for fixed values of T_R . If T_R varies, the average T_R can be used to give estimates of T_S and $T_{P_{ideal}}$.

The proof of Equation (3-2) is as follows. Let T_R be fixed, and let $(N \text{ divide } k)$ p times leaving remainder q , such that $N, k \geq 1$. The floor function in Equation (3-2) always returns p , and the step function returns zero if $q = 0$, and one otherwise. In proving Equation (3-2), two cases are considered for $1 \leq N \leq k$: (1) when $q = 0$, (2) when $q \neq 0$. If $q = 0$, the replications are batched into p batches with each batch containing N replications. Therefore, the theoretical time to execute the k replications is $p \times T_R = \left\lfloor \frac{k}{N} \right\rfloor T_R$. On the other hand, if $q \neq 0$, the replications are batched into $(p + 1)$ batches. Hence, the theoretical time to execute all k replications is $(p + 1)T_R = \left\lfloor \frac{k}{N} \right\rfloor T_R + T_R$. Finally, if $N > k$, the replications are batched into one batch of size k , and the total execution time is T_R .

Since Equation (3-2) is ideal, the actual parallel execution time with total overhead T_O is given by (3-3), where T_O is the total overhead associated with the time to set up a batch.

$$T_{P_{actual}}(T_R, k, N) = \left(\left\lfloor \frac{k}{N} \right\rfloor + S(k, N) \right) (T_R + T_O) \quad (3-3)$$

Thus, the effect of the scheduling overhead is to increase the program parallel execution time. In general, speedup is defined as the ratio of the program sequential execution time to the program parallel execution time.

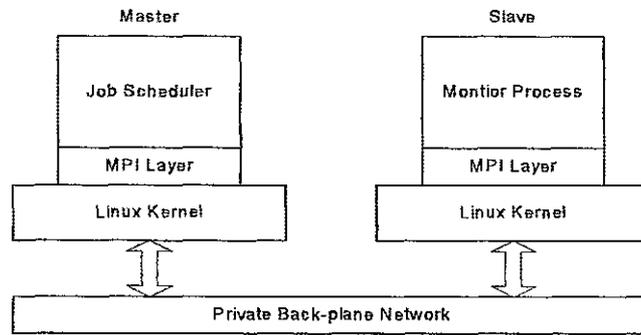
$$\text{speedup} = \tau = \frac{\text{program sequential execution time}}{\text{program parallel execution time}} = \frac{T_S}{T_{P_{actual}}} \quad (3-4)$$

3.3. Job Environments

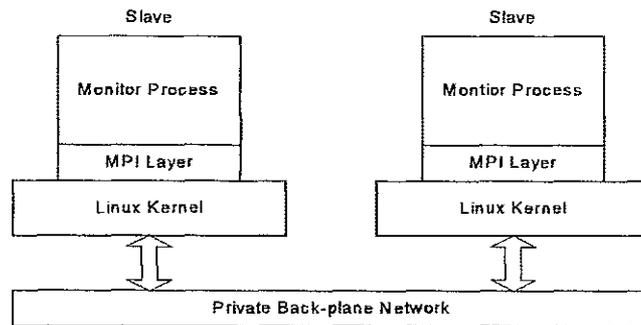
The communication between processes on remote processors in any job environment created by the XFace tool is achieved via MPI library routines. Figure 3-1 (a) portrays the master-slave communication between processes on the master processor and each slave processor, while Figure 3-1 (b) illustrates the slave-slave communication between processes on any two slave processors. The job scheduler and the monitor process are two resources of the XFace tool. The job scheduler runs on the master processor, and employs Algorithm 3-1 to schedule the iterations to the slave processors. One instance of the monitor process runs on each of the N slave processors. The monitor process enables master-slave communication between the master processor and each slave processor, and also enables slave-slave communication between any two slave processors. The job scheduler and the monitor process are described in Section 6.1.

The job scheduler and the monitor process are implemented on top of the MPI library. The MPI layers in Figure 3-1(a) provide the communication channel between the job scheduler and the monitor process. Similarly, the MPI layers in the diagram in Figure 3-1(b) provide the communication channel between two instances of the monitor process. Therefore, the job scheduler and the monitor process communicate by invoking MPI library routines. Likewise, two instances of the monitor process communicate by invoking MPI library routines.

The virtual network topology of the N slave processors and master processor used for scheduling is illustrated in Figure 3-2. The processors submitted to execute the job are labeled “Slave 1” through “Slave N” in the figure. The network links between the master processor and each slave processor in the figure illustrate the master-slave communication links established between the job scheduler and each instance of the monitor process. The job scheduler uses the master-slave communication links to communicate iterations to the slave processors during scheduling. Since the replications of the program under execution are independent, there is no communication between remote processes on any two slave processors when the replications are executed. Hence, the slave-slave communication links are not shown in Figure 3-2.



(a)



(b)

Figure 3-1. (a) Master-slave Communication (b) Slave-slave Communication

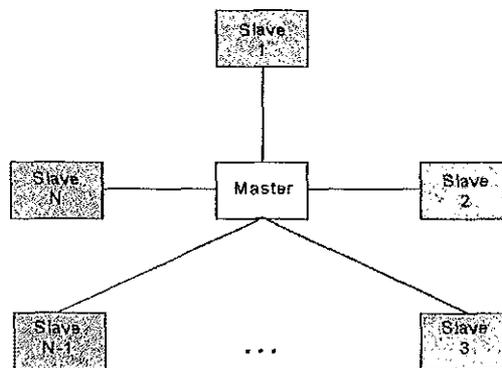


Figure 3-2. Virtual Network Topology

The job environment created on each slave processor for jobs consisting of Windows-based application programs is different from the environment created on each slave processor for jobs consisting of Linux-based application programs. Figure 3-3 depicts the job environment that is created on each slave processor when the application programs submitted in the job description are Windows-based. The interface between the XFace tool and the automation controller in Figure 3-3 is the text file *model.txt*. The automation controller is a program that externally controls the simulation application. The automation controller and the simulation application are both Windows-based applications developed to run on a Windows platform. Therefore, they run on top of the emulated Windows platform created by the Windows emulator. Most Windows-based applications register an object model with the Windows kernel. The application object model creates an interface between the application and the automation program. The automation program controls the application by issuing commands to the application object model.

At the start of each iteration, the monitor process writes the name of the application program and the number of times, x , that the program must be replicated for the current iteration to *model.txt*. The monitor process then starts the *start-win-app* script to initiate the execution of the iteration. The *start-win-app* process in turn starts the automation controller to initiate the execution of the application program. The automation controller starts a new instance of the application if an instance of the application is not already running in the job environment. Once an instance of the application is running, the automation controller opens the application program specified in *model.txt* and replicates the program x times. At the end of the x replications, the automation controller saves changes made to the program, and then closes the program. Finally, the automation program terminates, signaling the end of the iteration. This event triggers the termination of the *start-win-app* script.

Figure 3-4 illustrates the job environment that is created on each slave processor to execute Linux-based application programs. The environment is not applicable for VR jobs. According to the definitions of the parametric and partitioned job-types in Section 2.3, the application program submitted in the job description is replicated once per

iteration. Therefore, only one replication is executed per iteration of every job that is executed in the environment in Figure 3-4. Since the application is native to the Linux kernel, an emulator is not needed. So the monitor process starts the start-lin-app script to execute the replication, which in turn executes the command required to execute the replication.

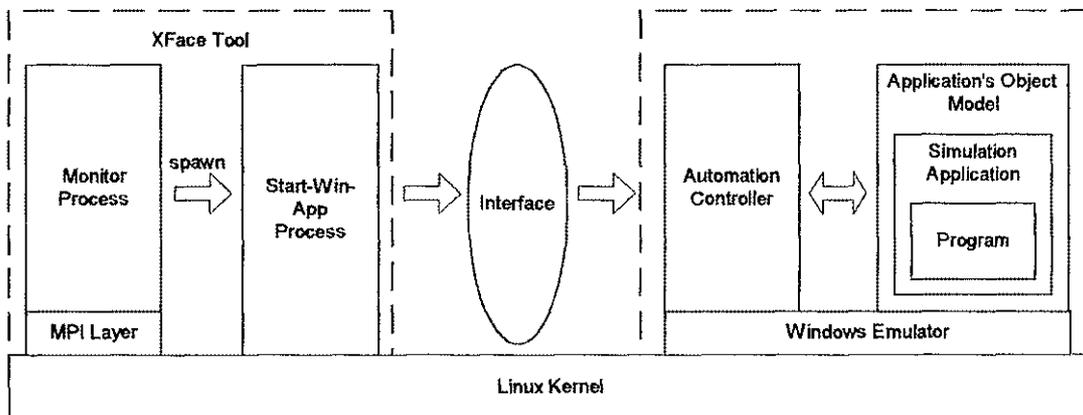


Figure 3-3. Job Environment created for Windows-based Applications.

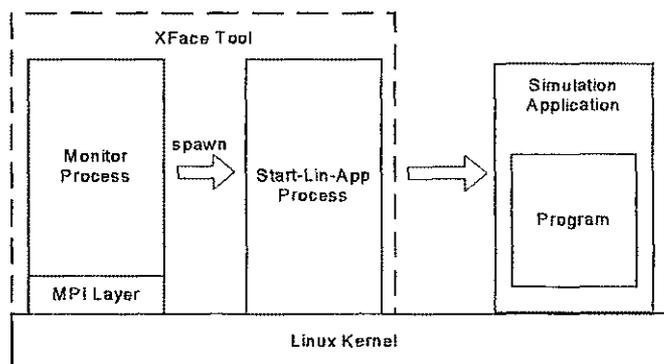


Figure 3-4. Job Environment created for Linux-based Applications.

The job environments in Figures 3-3 and 3-4 are application-specific. The job scheduler does not need any information about the simulation application to schedule the iterations to the slave processors. Also, the monitor process does not depend on the application. The monitor process only requires information about the application program name and the number of times the program is replicated per iteration. This information is made available to the monitor process during the job initialization phase when the job is loaded onto the slave processors. The start-win-app and start-lin-app scripts are also independent of the application. The start-win-app script only needs the application-specific command that is used to initiate the execution of the automation controller. Likewise, the start-lin-app script only needs the command to execute the application program. For Linux-based applications, the command is entered in the job description. However, for Windows-based applications, the user must add the command to execute the automation controller directly to the start-win-app script. This is done only once for each application that the tool must support. The start-win-app script is simple, and support for an application is easily added to the script.

To add support for a Windows-based application, an automation program and an installation of a Windows emulator on the native Linux platform are required. The automation program should provide the controls described earlier. The emulator should emulate the Windows environment that is required to execute both the designated application and the automation program. The command to execute the automation program is added to the start-win-app script. The application start-win-app script is described in Appendix A.

3.4. Maintaining the Independence Among Replications

An application program that is executed in the environment created by the XFace tool must read a text file containing stream indices at the beginning of each simulation replication. The file specifies k sets of stream indices. The first set of indices in the file is used to specify streams in the first replication, the second set of seed values is used to specify the streams in the second replication, and so on. A separator is used to separate the sets, and an end-of-file marker is used to mark the end of the file. A template of the

text file is illustrated in Figure 3-5. The ‘&’ character is used to separate the sets, and the character sequence “EOF” marks the end of the file. The values within a set need not be entered on the same text line in the file.

If the simulation model under execution contains r distinct streams, each of the k sets in the streams file must contain r distinct stream indices. The i^{th} index in the j^{th} set specifies U^i in the j^{th} replication, for $1 \leq i \leq r$ and $1 \leq j \leq k$. The independence among the replications now depends on how the stream indices in each set are chosen. The stream indices in each set should not be generated randomly. This is to guarantee that the sets are disjoint, and no set contains more than one instance of the same index. Instead, the stream indices must be carefully chosen so that the streams do not overlap. For example, if the fixed-length of the streams is L , then the stream U^1 is the sub-sequence in U containing the random numbers U_1 to U_L , U^2 is the sub-sequence containing the random numbers U_L to U_{2L} , and so on. Since U^i and U^{i+1} are adjacent sub-sequences in U , if r random points are defined in the model, and random point r_j requires between L and $2L$ random numbers, then if U^i is assigned to r_j , U^{i+1} should not be assigned to another random point to prevent the overlapping of U^i and U^{i+1} .

```

< 1st set containing r indices >
&
< 2nd set containing r indices >
&
      .
      .
      .
&
< kth set containing r indices >
EOF

```

Figure 3-5. Template of Streams File

During the job initialization phase, the XFace tool parses the streams file in the job description into k smaller files. The parsed files are then distributed with the replications so that the j^{th} set in the unparsed file is distributed with the j^{th} replication. Therefore, at the beginning of j^{th} replication, the program should read the file *stream.txt* and set streams to use in the replication. This requires that the target simulation application should provide support for changing the indices of the streams defined in the model.

3.5. Parametric Job Description

The job description forms that are used to submit parametric jobs are displayed in Figure 3-6. The form used to submit parametric jobs that contain Linux-based applications is displayed in Figure 3-6 (a), and the form used to submit parametric jobs that contain Windows-based applications is displayed in Figure 3-6 (b). The information submitted in a parametric job description is as follows. First, either the Linux platform or emulated Windows platform is selected to run the application. The number of times the application program must be replicated is then entered in the “Total Replications” text box. The pathnames to the program executable file and the streams file containing the stream indices that specify the streams to use in each replication are entered in the “Program Executable” and the “Seeds File” text boxes, respectively. If the program reads input files, the pathnames to the input files are entered in the “Input 1” and “Input 2” text boxes. If the number of input files is greater than two, clicking the “Add More Input Files ...” button displays a dialog box that prompts the user to enter the additional input files. If the application is Linux-based, the command to execute the program is entered in the “Command” text box. Otherwise, the application name is entered in the “Application” text box. Finally, the machines containing the processors to execute the replications are selected from the machines list. If all the required fields in the form are filled, clicking the “OK” button submits the job description. The “Cancel” button cancels the job description process, and the “Help” button provides helpful information about the job description.

Platform Options: Linux Emulated Windows

Number of Replications: Total Replications: 1

Job Files: Program Executable: Input 1: Seeds File: Input 2: Add More Input Files...

Program Execution Commands: Command: View Command Preview

Select the Machines to Execute the Job: n01.vnasc.odu.edu n02.vnasc.odu.edu n03.vnasc.odu.edu n04.vnasc.odu.edu n05.vnasc.odu.edu n06.vnasc.odu.edu n07.vnasc.odu.edu n08.vnasc.odu.edu n09.vnasc.odu.edu n10.vnasc.odu.edu n11.vnasc.odu.edu

OK Cancel Help

(a)

Platform Options: Linux Emulated Windows

Number of Replications: Total Replications: 11

Job Files: Program Executable: Input 1: Seeds File: Input 2: Add More Input Files...

Windows Application Name: Application: View Command Preview

Select the Machines to Execute the Job: n01.vnasc.odu.edu n02.vnasc.odu.edu n03.vnasc.odu.edu n04.vnasc.odu.edu n05.vnasc.odu.edu n06.vnasc.odu.edu n07.vnasc.odu.edu n08.vnasc.odu.edu n09.vnasc.odu.edu n10.vnasc.odu.edu n11.vnasc.odu.edu

OK Cancel Help

(b)

Figure 3-6. Parametric Job Description Forms (a) Windows-based (b) Linux-Based

CHAPTER IV VARIANCE REDUCTION JOBS

In this chapter, variance reduction jobs are described. The chapter presents an overview of the statistical analyses of two VRTs that, when applied to the random outputs of a simulation, reduce the variances in the differences between the statistical measures that are of interest. The analyses presented in the chapter focus on CRN and AV. These VRTs were chosen because of their wide use in practice and also due to their ease of implementation.

The notation used in the chapter is as follows. Random variables are typed in boldface letters. The expected value of the random variable \mathbf{X} is denoted $E\{\mathbf{X}\}$, the variance of \mathbf{X} is denoted $\text{Var}\{\mathbf{X}\}$, and the covariance of the random variables \mathbf{X}_1 and \mathbf{X}_2 is denoted $\text{Cov}\{\mathbf{X}_1, \mathbf{X}_2\}$. Other notation used in this chapter that is not listed here is defined when first used. Detailed definitions of these and other statistical terms used throughout the chapter are presented in [20].

4.1. Variance Reduction Analysis

Suppose we would like to compare two alternative configurations, C_1 and C_2 , of a system. Suppose further that C_1 and C_2 are the simple M/M/1 and M/M/2 models in Figure 4-1; the GI/G/s model is described in [21], [4]. Let the sample size of C_1 be l_1 and the sample size of C_2 be l_2 , $l_1 = l_2 = l$, for replication i , $i = 1, 2, \dots, k$. For a given k and l , henceforth denoted as the sample point (k, l) , the goal is to reduce the variance in the differences between sample values produced by C_1 and C_2 . To simplify the variance reduction analysis, let the expected average delay in queue, $d_i(l)$, resulting from the i^{th} replication of the M/M/s model, $s = 1, 2$, be of interest. To define the quantity $d_i(l)$, suppose the j^{th} entity enters the M/M/s system at time t_j , $j \geq 0$ and experiences service time S_j . Let $A_j = t_j - t_{j-1}$, $j \geq 0$, be the inter-arrival time between the $(j-1)^{\text{st}}$ and j^{th} entities entering the system. The sample value D_j is defined as the delay in queue of the j^{th} entity

during replication i , or equivalently, the time from arrival to the start of service experienced by the j^{th} entity.

$$D_0 = 0, D_j = \text{Max}\{D_{j-1} + S_{j-1}, D_{j-1} + (S_{j-1} - A_j)\}, j \geq 0 \quad (4-1)$$

The quantity $d_i(l)$ is defined as the sample average of the D_j 's in the i^{th} replication.

$$d_i(l) = \frac{\sum_{j=1}^l D_j}{l} \quad (4-2)$$

In a stochastic simulation, the quantities A_j and S_j are random samples drawn from the probabilistic distributions F_A and F_S that specify the inter-arrival and service time distributions, respectively. Therefore, $d_i(l)$ is a random sample of the random variable $\mathbf{d}(l)$.

For the i^{th} replication, denote $X_i = d_i^1(l)$ the expected average delay in queue for C_1 , and $Y_i = d_i^2(l)$ the average delay in queue for C_2 . Consider the random samples $Z_i = X_i \pm Y_i$, for $i = 1, 2, \dots, k$. Assume that the X_i 's, Y_i 's and Z_i 's are statistically independent, and identically distributed (IID). Let μ_X , μ_Y , and μ_Z be the true expected values of the X_i 's, Y_i 's and Z_i 's, respectively. That is, $\mu_X = E\{\mathbf{X}\} = E\{X_1 + X_2 + \dots + X_k\}$, $\mu_Y = E\{\mathbf{Y}\} = E\{Y_1 + Y_2 + \dots + Y_k\}$, and $\mu_Z = E\{\mathbf{Z}\} = E\{Z_1 + Z_2 + \dots + Z_k\}$. Then the variance of \mathbf{Z} is defined in Equation (4-3) [4].

$$\text{Var}\{\mathbf{Z}\} = \text{Var}\{\mathbf{X}\} + \text{Var}\{\mathbf{Y}\} \pm 2\text{Cov}\{\mathbf{X}, \mathbf{Y}\} \quad (4-3)$$

Since Z_i is defined as the sum or difference between the average delays X_i and Y_i in i^{th} replications of C_1 and C_2 , respectively, reduction of the variance between X_i and Y_i

observed in the i^{th} replications of C_1 and C_2 is achieved by reducing the variances in the Z_i 's. Observing Equation (4-3), it is obvious that the variance in the Z_i 's can be reduced by either inducing positive correlation between X_i and Y_i if $Z_i = X_i - Y_i$, or negative correlation between X_i and Y_i if $Z_i = X_i + Y_i$.

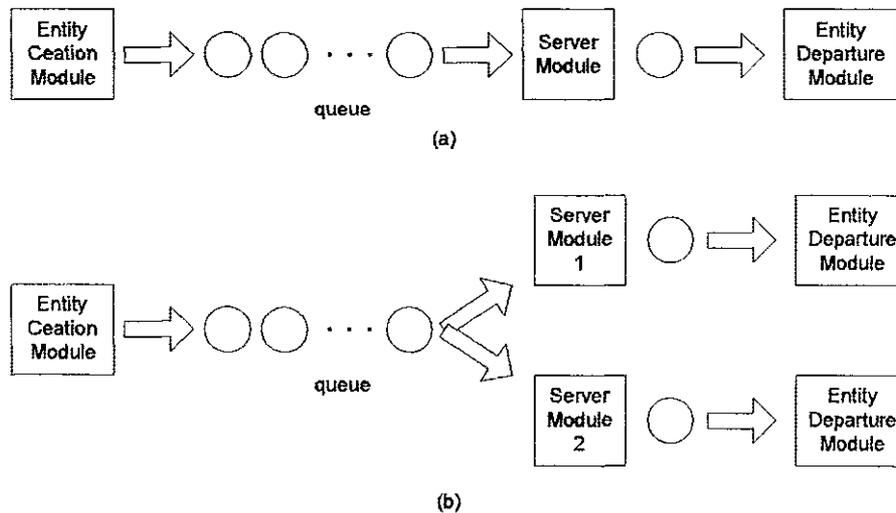


Figure 4-1. (a) M/M/1 Model (b) M/M/2 Model

For a given sample point (k, l) , we would like to define unbiased point estimators of the statistics μ_X , μ_Y , μ_Z , and $\text{Var}\{X\}$, $\text{Var}\{Y\}$, and $\text{Var}\{Z\}$ because oftentimes in practice these statistics are not known. Suppose the X_i 's are generated independently from the same model, that is, the X_i 's over n replications of C_1 are IID. Likewise, if the Y_i 's over n replications of C_2 are also IID, the Z_i 's are also IID. Thus, the sample mean $\bar{X}(k)$ is an unbiased estimator of μ_X and the sample variance $S_X^2(k)$ is an unbiased

estimator of $\text{Var}\{\mathbf{X}\}$, that is, $E\{\bar{\mathbf{X}}(\mathbf{k})\} = \mu_x$, and $E\{\mathbf{S}_x^2(\mathbf{k})\} = \text{Var}\{\mathbf{X}\}$. The sample mean $\bar{\mathbf{X}}(\mathbf{k})$ and the sample variance $\mathbf{S}_x^2(\mathbf{k})$ are defined as follows.

$$\bar{\mathbf{X}}(\mathbf{k}) = \frac{\sum_{i=1}^k X_i}{k} \quad (4-4)$$

$$\mathbf{S}_x^2(\mathbf{k}) = \frac{\sum_{i=1}^k (X_i - \bar{\mathbf{X}}(\mathbf{k}))^2}{k-1} \quad (4-5)$$

Likewise, the sample means $\bar{\mathbf{Y}}(\mathbf{k})$, $\bar{\mathbf{Z}}(\mathbf{k})$, and the sample variances $\mathbf{S}_y^2(\mathbf{k})$, and $\mathbf{S}_z^2(\mathbf{k})$ are unbiased estimators of μ_y , μ_z , and $\text{Var}\{\mathbf{Y}\}$, $\text{Var}\{\mathbf{Z}\}$, respectively, and are defined similarly as $\bar{\mathbf{X}}(\mathbf{k})$ and $\mathbf{S}_x^2(\mathbf{k})$ in Equations (4-4) and (4-5).

Suppose we perform k replications of both C_1 and C_2 such that the i^{th} replications of C_1 and C_2 are synchronized. Furthermore, suppose the pair (X_i, Y_i) resulting from the i^{th} replications of C_1 and C_2 are generated from a common random number sequence, but for different i , $i = 1, 2, \dots, k$, separate common random number sequences are used to generate (X_i, Y_i) . That is, in the i^{th} synchronized replications of C_1 and C_2 , X_i and Y_i of the pair (X_i, Y_i) are correlated, but the X_i 's are IID and the Y_i 's are also IID. Using the results in Equations (4-3) and (4-4), the variance of $\bar{\mathbf{Z}}(\mathbf{k})$ is given in Equation (4-6).

$$\text{Var}[\bar{\mathbf{Z}}(\mathbf{k})] = \frac{\text{Var}\{\mathbf{Z}\}}{k} = \frac{\text{Var}\{\mathbf{X}\} + \text{Var}\{\mathbf{Y}\} \pm 2\text{Cov}\{\mathbf{X}, \mathbf{Y}\}}{k} \quad (4-6)$$

Therefore, the variance of the sample mean $\bar{\mathbf{Z}}(\mathbf{k})$ is inversely proportional to the number of replications. Thus, replicating C_1 and C_2 a large number of times will reduce the variance in $\bar{\mathbf{Z}}(\mathbf{k})$. This is where the parallel platform provided by the Beowulf cluster

proves very handy. However, powerful VRTs such as CRN and AV are oftentimes used in practice to reduce the variance in $\bar{\mathbf{Z}}(\mathbf{k})$. This is due to the fact these VRTs converge the sample mean $\bar{\mathbf{Z}}(\mathbf{k})$ to the true mean $\mu_{\mathbf{Z}}$ at a faster rate than the convergence rate when the models are replicated many times.

4.2. Common Random Numbers

Common Random Numbers is a variance reduction technique that is used to reduce the variance in $\bar{\mathbf{Z}}(\mathbf{k})$ by inducing positive correlation between X_i and Y_i . For the case when $Z_i = X_i - Y_i$, Equation (4-6) reduces to Equation (4-7).

$$\text{Var}\{\bar{\mathbf{Z}}(\mathbf{k})\} = \frac{\text{Var}\{\mathbf{X}\} + \text{Var}\{\mathbf{Y}\} - 2\text{Cov}\{\mathbf{X}, \mathbf{Y}\}}{k} \quad (4-7)$$

Clearly, if in the i^{th} synchronized replications of C_1 and C_2 , X_i and Y_i of the pair (X_i, Y_i) are generated independently, that is, distinct random number sequences are used to drive the simulations of C_1 and C_2 , it follows that X_i and Y_i are uncorrelated and $\text{Cov}\{\mathbf{X}, \mathbf{Y}\} = 0$. Thus, Equation (4-7) reduces as follows.

$$\text{Var}\{\bar{\mathbf{Z}}(\mathbf{k})\} = \frac{\text{Var}\{\mathbf{X}\} + \text{Var}\{\mathbf{Y}\}}{k} \quad (4-8)$$

On the other hand, if in the i^{th} synchronized replications of C_1 and C_2 , X_i and Y_i of the pair (X_i, Y_i) were generated from a common random number sequence, X_i and Y_i are positively correlated and $\text{Cov}\{\mathbf{X}, \mathbf{Y}\} > 0$. Therefore, according to Equations (4-7) and (4-8), the variance of $\bar{\mathbf{Z}}(\mathbf{k})$ is smaller for the case when X_i and Y_i are positively correlated, $\text{Cov}\{\mathbf{X}, \mathbf{Y}\} > 0$, than for the case when X_i and Y_i are independent, $\text{Cov}\{\mathbf{X}, \mathbf{Y}\} = 0$, resulting in the desired variance reduction in $\bar{\mathbf{Z}}(\mathbf{k})$.

4.3. Antithetic Variates

Antithetic Variates (AV) is another VRT that depends on the induction of correlation between the random variables of interest to reduce variance. AV reduces the variance of $Z_i = \frac{X_i^1 + X_i^2}{2}$ by inducing negative correlation between X_i^1 and X_i^2 , where the random outputs X_i^1 and X_i^2 are generated from paired replications of the same model. Therefore, the variance of the sample mean $\bar{Z}(\mathbf{k})$ is given by Equation (4-9).

$$\text{Var}\{\bar{Z}(\mathbf{k})\} = \frac{\text{Var}\{X^1\} + \text{Var}\{X^2\} + 2\text{Cov}\{X^1, X^2\}}{4k} \quad (4-9)$$

Thus, the variance of the sample mean $\bar{Z}(\mathbf{k})$ is smaller when $\text{Cov}\{X^1, X^2\} < 0$ than that for the case when $\text{Cov}\{X^1, X^2\} = 0$. Unlike CRN, antithetic random numbers are used for the same purpose in each paired replications. Thus, for example, if in the i^{th} paired replication of C_1 , the random number U_k is used to generate the k^{th} service time in the first replication of the pair, then $1-U_k$ must be used to generate the k^{th} service time in the second replication. Further information on AV is presented in [3], [4], and [5].

4.4. Synchronizing the Random Numbers

The driving force behind CRN and AV is the proper synchronization of the random numbers that are used during the simulation. Without proper synchronization of the random numbers, neither CRN nor AV would work. This section describes one method of synchronizing the random numbers used in simulations under CRN or AV. The presentation focuses on synchronization techniques applied to simulations under CRN. However, the synchronization techniques that are described in this section are also applicable to simulations under AV.

The idea behind CRN is to induce positive correlation between X_i and Y_i in the i^{th} synchronized replications of C_1 and C_2 by enforcing that the order of random number usage in C_1 and C_2 is the same. That is, in the i^{th} replications of C_1 and C_2 , the same common random number sequence U is used to drive the simulations of C_1 and C_2 . Furthermore, the k^{th} random number U_k drawn from U is used for the same purpose in the i^{th} replications of C_1 and C_2 . Hence, if in the i^{th} replication of C_1 , the random number U_r is used to generate the r^{th} inter-arrival time, then U_r must also be used to generate the r^{th} inter-arrival time in the i^{th} replication of C_2 . Therefore, for CRN to work, it is critical that the random numbers used in the simulation of C_1 and C_2 are not merely the same, but are also properly synchronized.

There are many ways to synchronize the order of random number usage in the simulation of C_1 and C_2 [3], [4]. The method of synchronization of the common random numbers used to drive the simulations of C_1 and C_2 implemented here is as follows. Label each point in C_j , $j = 1, 2$, that generates random variates a random point. Thus, for the M/M/1 queuing model in Figure 3-1 (a), there are two such points, one in the Entity Creation Module where the inter-arrival times are generated and another in the Server Module where the services times are generated. Similarly, for the M/M/2 queuing model in Figure 3-1 (b), there are three random points, one in the Entity Creation Module where the inter-arrival times are generated, and one in each of the two Server Modules where the service times are generated. One way to achieve synchronization of the common random numbers driving the simulations of C_1 and C_2 in each replication is to assign separate random number streams to each random point in C_j , $j = 1, 2$ that are different. However, the assignment must be such that the same stream is assigned to identical random points in C_1 and C_2 . Thus, for C_1 , if we assign stream U^1 to generate inter-arrival times and stream U^2 to generate service times, then we must make the same stream assignments in C_2 , that is, assign U^1 to generate inter-arrival times, and U^2 to generate service times for the two servers. Even though the two server modules in the M/M/2 model constitute two random points, these random points are the same since both random points generate service times for the same purpose in the system being modeled.

4.5. VR Job Descriptions

This section describes the job description forms used to submit VR jobs to the XFace tool. VR jobs are only supported on the emulated Windows environment. The VR-AV job description form is displayed in Figure 4-2. It is very similar to Windows-based parametric job description form in Figure 3-6 (b).

Figure 4-2. VR-AV Job Description Form

The VR-CRN job description form is presented in Figure 4-3. The total number of configurations c , $c \geq 2$, of the system or systems being simulated is entered into the “Total Configuration” text box. Since CRN involves the comparison of at least two alternate models, the minimum value that can be entered in this field is 2. The number of replications of each configuration is entered in the “Total Replication” text box. Each

model will be replicated k times. The application name is entered in the “Application” text box. The slave processors to execute the job are selected from the machines list. Finally, a set of job files is entered for each configuration. When entering the job files, the configuration number is first entered and then the job files are entered in the text fields provided. The “Submit Configuration Files” button is used to submit the job files for a configuration. Each configuration submitted is displayed in the “Configuration Submitted” window. The “View Submitted Files...” button displays the job files submitted for the selected configuration, and the “Delete” button deletes the selected configuration. The “OK”, “Cancel,” and “Help” buttons serve the same purposes as those on the job description forms in Figure 3-6.

The screenshot shows a graphical user interface for submitting job configurations. It includes the following elements:

- Configuration Settings:**
 - Number of Configurations: Total Configurations: 2
 - Number of Replications: Total Replications: 2
 - Windows Application Name: Application: []
- Machine Selection:**
 - Select the Machines to Execute the Job:
 - List of machines: n01.vnasc.edu, n02.vnasc.edu, n03.vnasc.edu, n04.vnasc.edu, n05.vnasc.edu, n06.vnasc.edu, n07.vnasc.edu, n08.vnasc.edu, n09.vnasc.edu, n10.vnasc.edu
- Job File Submission:**
 - Job Files: Job Files for a Specific Configuration
 - Enter Configuration for which Job Files are being submitted: 1
 - Program Executable: []
 - Send File: []
 - Input 1: []
 - Input 2: []
 - Buttons: Add Here Input Files..., Submit Configuration Files
- Submitted Configurations:**
 - Configurations Submitted: []
 - Buttons: View Submitted Files..., Delete
- Navigation:**
 - Buttons: OK, Cancel, Help

Figure 4-3. VR– CRN Job Description Form

CHAPTER V

PARTITIONED JOBS

This chapter presents the concepts behind the partitioned job feature of the XFace tool. The execution of a program that is made up of interdependent units of computation, where the interdependencies among computation units are dictated by the flow of data during program execution, is considered. The partitioned job feature of the XFace tool is used to execute the k replications of a partitioned program on sub-clusters consisting of several processors. Before delving into any further details, a formal definition of the sub-cluster model used to execute each replication of the program is presented. For a given program that is executed using the partitioned job feature of the XFace tool, let the processors that are contained in the set Π be P_h , $h = 1, 2, \dots, N$, and let the N processors be divided into n sets according to a predefined rule R that places each processor P_h into a set Γ_i , $1 \leq i \leq n$. Each Γ_i is considered a sub-cluster with cardinality $|\Gamma_i| = m_i$. Define Γ as the set containing all sub-clusters, so that $|\Gamma| = n$. It is necessary that R satisfies the following condition: For every $\Gamma_i \subset \Gamma$ and $\Gamma_j \subset \Gamma$, $1 \leq i, j \leq n$ and $i \neq j$, $\Gamma_i \cap \Gamma_j = \{\}$, where $\{\}$ denotes the empty set, so that each processor P_h can only be contained in either Γ_i or Γ_j but not both. Therefore, the N slave processors submitted to execute the job are divided into n mutually exclusive sub-clusters according to the placement rule R . The maximum sub-cluster size is given by

$$m_{\max} = \text{MAX} \{ |\Gamma_1|, |\Gamma_2|, \dots, |\Gamma_n| \}. \quad (5-1)$$

Suppose a replication is split into φ computation units. Since the φ computation units are interdependent, even if $m_{\max} > \varphi$, all the computation units cannot execute concurrently for a given replication that is executed on Γ_i consisting of m_{\max} processors. However, the precedence constraints among the φ computation units may be such that sets of independent computation units exist that can be executed concurrently. If this is the case, we can utilize such parallelism if we can partition the program into computation

units and execute each replication of the program on sub-cluster Γ_i consisting of m_i processors.

Unfortunately, executing each replication of the program on sub-cluster Γ_i injects overheads that have the effect of increasing the execution time of each replication. Also, problems that were not present when each replication of the program was executed on a single processor must now be addressed. These problems include the following. A partitioning algorithm is needed to partition the program into interdependent computation units, each of which must be of a certain size or granularity in order to achieve the maximum speedup in the execution time of each replication of the program; a scheduling algorithm is needed to efficiently map the computational units in the partitioned program to the processors in Γ_i ; the executions of the computation units on the processors in Γ_i must be synchronized; the data dependencies that exist among the computational units in the partitioned program must be preserved during the execution of each replication of the program; and data must be forwarded among the processors in Γ_i .

The partitioning and scheduling problems are briefly addressed in Sections 5.1 and 5.3, respectively. The remaining problems are automatically taken care of by representing the partitioned program by the task precedence graph model described in Section 5.2.

5.1. The Partitioning Problem

A program comprises a set of instructions. The instructions perform computations on input data to produce output data. The partitioning of the program is the act of dividing the program instructions into smaller instruction subsets that are executed sequentially on a single processor. The computation performed by an instruction subset is a unit of computation. Each unit of computation is defined as a *task*.

Tremendous research had been done on the partitioning problem; for example, the partitioned problem is addressed in [22], [23]. The partitioning problem is a difficult problem, NP-complete in the strong sense. Several factors must be considered when partitioning a program into tasks, such as but not limited to, the program size, the average task size or granularity of the partitioned program, the number of available processors,

load balancing, and the multiprocessor architecture that will execute the program. The partitioning problem is not the focus of the thesis. Therefore, it is assumed that a parallel algorithm is already designed for the program that will be submitted to the XFace tool, and that the application program is already partitioned into interdependent tasks and is represented by a task precedence graph.

5.2. Task Precedence Graph Model

A program partitioned into interdependent tasks can be modeled by the task precedence graph illustrated in Figure 5-1. The nodes in the graph represent the tasks in the partitioned program. Each directed-edge between any two tasks represents the precedence constraints between the tasks, the flow of data between the tasks, and the communication costs associated with data flowing between the tasks. A task precedence graph is commonly referred as a dynamic acyclic graph (DAG) in the scheduling literature. An in depth discussion of the DAG model is presented in [24].

In general, a DAG $G = (V, E)$ consists of a set of weighted nodes $V = \{i = 0, 1, 2, \dots, n \mid T_i\}$, and a set of weighted directed-edges, $E = \{i, j = 0, 1, 2, \dots, n \mid e_{i,j}\}$, that connect nodes in the graph. Each node $T_i \in V$ is associated with a computational cost $c_i = W(T_i)$, where $W(T_i)$ is the node weight or sequential execution time required to execute task T_i . Each directed-edge $e_{i,j} \in E$ connects the two interdependent tasks T_i and T_j with the edge going from T_i to T_j . The direction of the edge $e_{i,j}$ represents the precedence constraints between T_i and T_j , specifying that the execution of T_i must precede the execution of T_j . The direction of the edge $e_{i,j}$ also indicates that the flow of data is from T_i to T_j during program execution. The weight $W(e_{i,j})$ of $e_{i,j}$ represents the communication cost charged to forward the data produced by the execution of T_i on processor P_r to processor P_h that will execute T_j . In this thesis, for the edge $e_{i,j}$ that connects T_i and T_j , T_i is defined as the *parent* of T_j , and T_j is defined as the *child* of T_i . Furthermore, it is assumed that the edge weights are negligible with respect to the task weights to disregard the communication overhead associated with the edges.

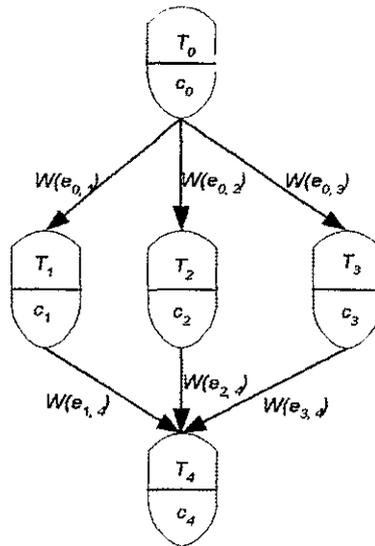


Figure 5-1. DAG Model.

A DAG has entry nodes, exit nodes, and nodes that are neither entry nor exit nodes. An entry node in a DAG is a node that is not dependent on any other node in the DAG for data at the start of program execution, and thus can begin execution if it is scheduled to a processor. Thus, entry nodes in a DAG are the first to execute. An entry node has no parent and k children. For example, for the DAG in Figure 5-1, the node that represents task T_0 is the only entry node in the graph. On the other hand, an exit node has k parents but no children. The precedence constraints that exist among the nodes in the DAG dictate that an exit node cannot begin execution until all its k parents are executed. As a result, exit nodes are the last nodes in the DAG to execute. For the DAG depicted in Figure 5-1, the node that represents task T_4 is the only exit node in the graph. Nodes that are neither entry nor exit nodes have k_1 parents and k_2 children. For the DAG depicted in Figure 4-1, the nodes that represent tasks T_1 , T_2 , and T_3 are neither exit nor entry nodes. Each of these nodes has parent T_0 and child T_4 .

When the program represented by the DAG in Figure 5-1 is executed, task T_0 is executed first. After the execution of T_0 completes, the data produced by T_0 is forwarded

to the processors scheduled to execute tasks T_1 , T_2 , and T_3 . If at least three processors are available, tasks T_1 , T_2 , and T_3 can begin execution simultaneously on separate processors since no data dependency exists among these tasks. Otherwise, if less than three processors are available, tasks T_1 , T_2 , and T_3 cannot all begin execution at once. However, they can begin execution in any order. Finally, task T_4 can begin execution after tasks T_1 , T_2 , and T_3 are executed and all data have been forwarded to the processor that will execute T_4 .

In the examples that are considered in this thesis, DAGs that are of similar structures to that of the DAG in Figure 5-1 are considered. The DAGs have one or more entry nodes that are followed by at least two nodes that can be executed in parallel, and end with one or more exit nodes. It is assumed that the computational costs of the nodes in parallel are much greater than the computational costs of the entry nodes and exit nodes. Also, the assignment of task indices in the DAG is irrelevant, that is, the nodes that represent the tasks can be indexed in any order.

Estimation of the task weights is done prior to the execution of the DAG. Practically, a task weight can be estimated by executing the task with all the required data several times on a dedicated processor. The average of the execution times gives an estimate of the task weight.

5.3. The Scheduling Problem

The scheduling problem is to map the tasks in the DAG to the processors in sub-cluster Γ_i to achieve optimum speedup in one replication of the program represented by the DAG. This is a difficult problem that is also NP-complete. Tremendous research had been done on the scheduling problem. As a result, a plethora of scheduling algorithms exist that schedule the tasks in a DAG to targeted multiprocessors [24]. These scheduling algorithms use a wide variety of heuristics to schedule tasks within a DAG to the targeted multiprocessors. However, the ultimate objective of all these algorithms is to achieve optimum schedule length.

Since the scheduling problem is not the main focus of this thesis, a simple scheduling algorithm is developed to demonstrate the partitioned job feature of the XFace tool. However, it is worthy to note that the algorithm developed and implemented in the tool can be replaced with a more sophisticated scheduling algorithm.

5.4. Scheduling Tasks within a Replication

The scheduling algorithm developed in this section is based on the following assumptions: (1) The scheduling algorithm operates under a non-preemptive scheduling option, that is, once a task begins execution on a processor, the task will execute to completion; (2) the DAG is assumed to have deterministic property, that is, no probabilistic measures are associated with the edges in the DAG. Although it is assumed that the edge weights are negligible with respect to the task weights, Algorithm 5-1 makes use of the edge weights when scheduling. This was done to support future work on the XFace tool. Algorithm 5-1 still works if the weights are assumed to be negligible and equal. The scheduling algorithm utilizes fork and join structures embedded within the DAG during scheduling in an attempt to achieve the optimum¹ speedup in the execution time of one replication of the program. Figure 5-2 (a) and (b) depict fork and join structures, respectively. The terms *head of a fork* and *tail of a join* are used in this section and are defined as follows. The *head of a fork* is the parent task in the fork structure, and the *tail of a join* is the child task in the join structure. For example, T_0 is the head of the fork structure depicted in Figure 5-2 (a), and T_n is the tail of the join structure depicted in Figure 5-2 (b).

1. The current implementation of Algorithm 5-1 in the XFace tool does not yield the optimum schedule length. This is due to the fact that the execution of task T_i on processor P_h may be preempted to forward data to processor P_f on which task T_j will be executed, given that T_j is a child of T_i .

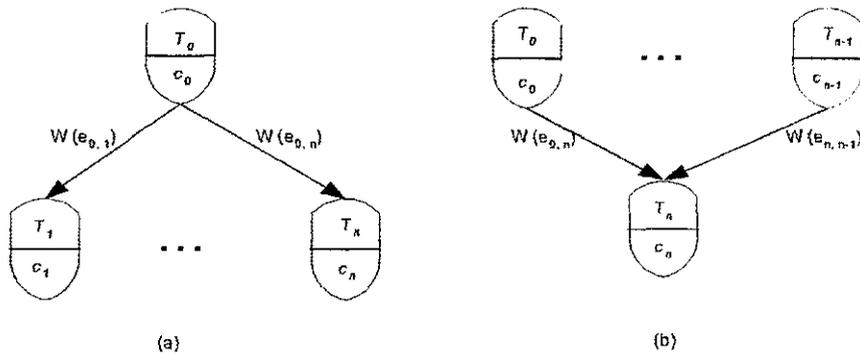


Figure 5-2. (a) Fork Structure (b) Join Structure

In scheduling the DAG in Figure 5-1, the following steps are executed.

- (1) Add tasks that are ready (ready tasks) to execute to the ready list. Ready tasks are those that have no data dependency or a dependency count of zero. The *dependency count* of task T_j is defined as the current number of parent tasks that must produce data that is required for T_j to begin execution.
- (2) For each ready task T_j with a dependency count of zero, map T_j to a processor and add T_j to the ready list.
- (3) Upon the completion of the execution of task T_j , decrement the dependency count of each child of T_j .

During the execution of each replication of the program represented by the DAG model in Figure 5-1, Algorithm 5-1 will be executed several times. It is executed once before any task is executed, and once after each task completes execution. It could be the case that several tasks finish simultaneously. In this case, the scheduling algorithm is executed once for all the finish tasks. Therefore, the input parameters to the algorithm are as listed below.

(1) The DAG that is being executed. A data structure must be defined to store the state of each task. The DAG can be implemented as an array of task structures.

(2) The list of processors that are assigned to execute the DAG. Each processor in the list is either marked “scheduled” or “unscheduled” indicating its availability.

(3) The list of tasks that finish execution prior to the current execution of the scheduling algorithm. The list will be empty the first time the algorithm executes.

(4) The ready list of tasks. All ready tasks will be added to the ready list. The processor assigned to execute a ready task is stored within the task structure.

Algorithm 5-1. Scheduling Tasks within a DAG

Begin

(1) *For each scheduled task T_j that completes execution, decrement the dependency count of each task T_k that is a child of T_j .*

(2) *Add all tasks in the DAG that have a dependency count of zero to the ready list. For each task T_j added to the ready list, eliminate T_j from the scheduling process.*

(3) *For each task T_j added to the ready list:*

i. *If T_j is the head of a fork and not the tail of a join, schedule T_j to the next available processor P_m and assign P_m to the child T_k of T_j such that the edge weight $W(e_{j,k})$ is the heaviest of all edges connecting T_j to one of its children T_k .*

ii. *Else if T_j is the tail of a join and not the head of a fork, schedule T_j to the same processor P_m that its parent T_i was scheduled such that the edge weight $W(e_{i,j})$ is the heaviest of all edges connecting T_j to one of its parents T_i .*

- iii. Else T_j is both the tail of a join and the head of a fork, and processor P_m was assigned to T_j by a parent and processor P_m is available. schedule T_j to the same processor P_m that its parent T_i was scheduled such that the edge weight $W(e_{ij})$ is the heaviest of the edges that connects T_j to one of its parents T_i . Otherwise, if T_j was not assigned a processor, schedule T_j to the next available processor P_n . Assign P_* , $* = m$ or n , to the child T_k of T_j such that the edge weight $W(e_{jk})$ is the heaviest of all edges connecting T_j to one of its children T_k .

End

Algorithm 5-1 groups the executions of communicational intensive tasks on the same processor in an effort to reduce the schedule length.

5.5. Speedups

The ideal parallel execution time for k replications executed on n sub-clusters Γ_i , where m_i processors are contained in Γ_i and every $\Gamma_i \subset \Gamma$ has the same size $m_i = m$, is defined by Equation (5-1). The replication time T_R now depends on the number of processors m that are contained in sub-cluster Γ_i .

$$T_{P_{ideal}}(T_R(m), k, n) = \left(\left\lfloor \frac{k}{n} \right\rfloor + S(k, n) \right) T_R(m) \quad (5-2)$$

$$T_{P_{actual}}(T_R(m), k, n) = \left(\left\lfloor \frac{k}{n} \right\rfloor + S(k, n) \right) (T_R(m) + T_O) \quad (5-3)$$

Speedups in the execution time of the partitioned program can be achieved in two ways. The 1st-dimension speedup results from executing the partitioned program on n sub-clusters. The 2nd-dimension speedup results from executing each replication of the partitioned program on sub-cluster Γ_i containing m processors. The 1st-dimension speedup is denoted τ_{1D} and the 2nd-dimension speedup is denoted τ_{2D} . The 1st-dimension speedup is the ratio of the sequential time T_S to execute the k replications on a single

processor and the parallel time to execute the k replications on n sub-clusters each containing one processor. The 2nd-dimension speedup is the ratio of the parallel time to execute the k replications on n sub-clusters each containing one processor and the parallel time to execute the k replications on n sub-clusters each containing m processors. Therefore, the total speedup is the product of τ_{1D} and τ_{2D} .

$$\tau_{1D}(T_R(l), k, n) = \frac{T_S}{T_{P_{actual}}(T_R(l), k, n)} \quad (5-4)$$

$$\tau_{2D}(T_R(m), k, n) = \frac{T_{P_{actual}}(T_R(l), k, n)}{T_{P_{actual}}(T_R(m), k, n)} \quad (5-5)$$

$$\tau(T_R(m), k, n) = \tau_{1D} \times \tau_{2D} = \frac{T_S}{T_{P_{actual}}(T_R(m), k, n)} \quad (5-6)$$

Ideally, the 1st-dimension speedup τ_{1D} in Equation (5-4) is same as the parametric speedup in Equation (3-4). However, practically, τ_{1D} will be less than the parametric speedup because of the scheduling and communication overheads that are added to the execution time of each replication executed on sub-cluster Γ_i .

Unlike τ_{1D} , τ_{2D} is not easily estimated because τ_{2D} depends on the structure of the DAG, the scheduling algorithm used for scheduling tasks in the DAG, and the number of processors assigned to execute the DAG. However, once these factors are known, an optimum τ_{2D} can be estimated with the aid of Ghant Charts.

The following example demonstrates estimating τ_{2D} , and also demonstrates Algorithm 5-1 at work. For this example, arbitrary task weights are added to the DAG depicted in Figure 5-1. The modified DAG is depicted in Figure 5-3. For scheduling purposes, all edges in the DAG have weight e , where e is negligible with respect to the task weights. The three processors P_1 , P_2 , and P_3 are assigned to execute one replication of the program represented by the DAG in Figure 5-3. Throughout the example, the

possible state of processor P_h , $h = 1, 2, \text{ or } 3$, is either S or NS, where S denotes that processor P_h is scheduled to execute task T_i , and NS denotes that processor P_h is not scheduled to execute any task. The possible execution state of task T_i is E, EP or EC, where E denotes that task T_i is executing, EP denotes that the execution of task T_i is pending, and EC denotes that the execution of task T_i has completed. NA denotes that an item is not applicable. The states of the three processors prior to the first invocation of the scheduling algorithm are displayed in Table 5-1, and the initial states of the tasks in the DAG are displayed in Table 5-2.

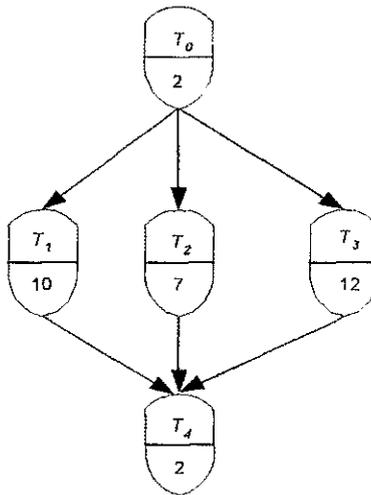


Figure 5-3. Sample DAG Model

Table 5-1. Initial States of Processors

	P_1	P_2	P_3
Processor State	NS	NS	NS

Table 5-2. Initial States of Tasks

	T ₀	T ₁	T ₂	T ₃	T ₄
Dependency Count	0	1	1	1	3
Scheduled Processor	NA	NA	NA	NA	NA
Execution State	EP	EP	EP	EP	EP
Has task been eliminated from Scheduling Process?	No	No	No	No	No

Initially, all processors assigned to execute the DAG are marked unscheduled (NS). The tasks states, displayed in Table 5-2, are explained as follows. The “Dependency Count” row displays the current dependency count of each task. Initially, no processor is scheduled; thus, the “Scheduled Processor” row displays NA (not applicable) for each task, and the execution state of each task is marked as execution pending (EP).

At the beginning of the first loop through Algorithm 5-1, the entry node T₀ has a dependency count of zero. As a result, it is added to the ready list, scheduled to the next available processor P₁, and is eliminated from the scheduling process. Table 5-3 and Table 5-4 display the processors states and tasks states, respectively, after the execution of the first loop of the scheduling algorithm.

Table 5-3. States of Processors after First Loop through Algorithm 5-1

	P ₁	P ₂	P ₃
Processor State	S	NS	NS

Table 5-4. States of Tasks after First Loop through Algorithm 5-1

	T ₀	T ₁	T ₂	T ₃	T ₄
Dependency Count	0	1	1	1	3
Scheduled Processor	P ₀	NA	NA	NA	NA
Execution State	EP	EP	EP	EP	EP
Has task been eliminated from Scheduling Process?	Yes	No	No	No	No

The second loop through Algorithm 5-1 is executed after task T₀ has completed execution. At this point, processor P₁ that was scheduled to execute T₀ is now available. Thus, all three processors are available for task execution. The dependency counts of tasks T₁, T₂, and T₃ are decremented by one. As a result, all three tasks have dependency counts of zero. Thus, each of these tasks is added to the ready list and scheduled to a processor at the end of the loop. The states of the processors and the tasks after the execution of the second loop through Algorithm 5-1 are displayed in Tables 5-5 and 5-6, respectively.

The third loop through the algorithm is executed when the execution of T₂ completes. The dependency count of task T₄ is decremented by one, and now has a value of two. After the execution of the loop, no task is added to the ready list. Tables 5-7 and 5-8 display the states of the processors and tasks after the execution of the third loop through the algorithm.

Table 5-5. States of Processors after Second Loop through Algorithm 5-1

	P ₁	P ₂	P ₃
Processor State	S	S	S

Table 5-6. States of Tasks after Second Loop through Algorithm 5-1

	T ₀	T ₁	T ₂	T ₃	T ₄
Dependency Count	0	0	0	0	3
Scheduled Processor	P ₀	P ₀	P ₁	P ₂	NA
Execution State	EC	EP	EP	EP	EP
Has task been eliminated from Scheduling Process?	Yes	Yes	Yes	Yes	No

Table 5-7. States of Processors after Third Loop through Algorithm 5-1

	P ₁	P ₂	P ₃
Processor State	S	NS	S

Table 5-8. States of Tasks after Third Loop through Algorithm 5-1

	T ₀	T ₁	T ₂	T ₃	T ₄
Dependency Count	0	0	0	0	2
Scheduled Processor	P ₀	P ₀	P ₁	P ₂	NA
Execution State	EC	E	EC	E	EP
Has Task been eliminated from Scheduling Process?	Yes	Yes	Yes	Yes	No

Task T₄ is not added to the ready list until the execution of tasks T₁ and T₃ have completed, which occurs after the fifth loop through the algorithm. The processor states and task states after the fourth and fifth loops through the algorithm are displayed in Tables 5-9 through 5-12.

Table 5-9. States of Processors after Fourth Loop through Algorithm 5-1

	P ₁	P ₂	P ₃
Processor State	NS	NS	S

Table 5-10. States of Tasks after Fourth Loop through Algorithm 5-1

	T ₀	T ₁	T ₂	T ₃	T ₄
Dependency Count	0	1	1	1	1
Scheduled Processor	P ₀	P ₀	P ₁	P ₂	NA
Execution State	EC	EC	EC	E	EP
Has Task been eliminated from Scheduling Process?	Yes	Yes	Yes	Yes	No

Table 5-11. States of Processors after Fifth Loop through Algorithm 5-1

	P ₀	P ₁	P ₂
Processor State	S	NS	NS

Table 5-12. States of Tasks after Fifth Loop through Algorithm 5-1

	T ₀	T ₁	T ₂	T ₃	T ₄
Dependency Count	0	0	0	0	0
Scheduled Processor	P ₀	P ₀	P ₁	P ₂	P ₀
Execution State	EC	EC	EC	EC	EP
Has Task been eliminated from Scheduling Process?	Yes	Yes	Yes	Yes	Yes

The Gantt Chart in Figure 5-4 illustrates the schedule length achieved for the example. A schedule length of 17 time-units is achieved for the execution of the DAG in Figure 5-3. If one replication of the program represented by the DAG were executed on a single processor, the optimum schedule length achievable is 33 time-units. Therefore, using three processors to execute the DAG, a speed-up in the execution time of one replication of the program, τ_{2D} , of 51% is achievable. If instead two processors were used to execute the DAG, τ_{2D} would be less than 51%. Also, if more than three processors were used to execute the DAG, τ_{2D} would not exceed 51%.

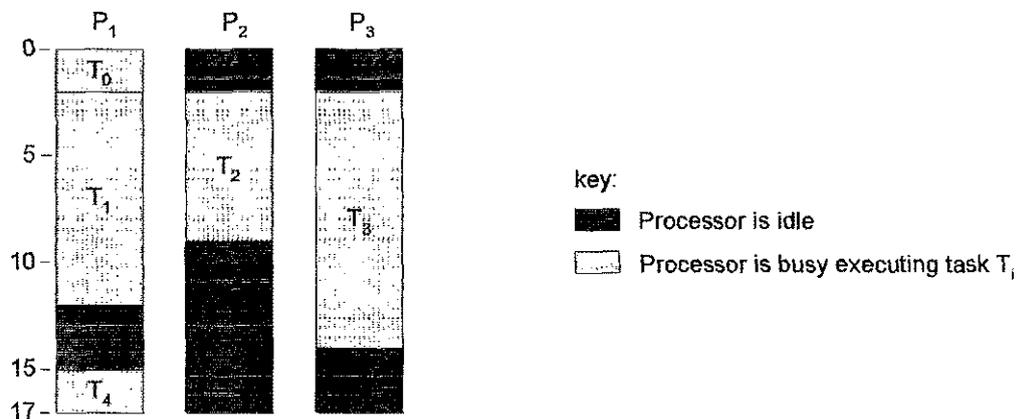


Figure 5-4. Gantt Chart Illustrating Optimum Schedule Length

5.6. Partitioned Job Environment

This section describes the extensions to the execution environments presented in Section 3.3 that are required to support the partitioned job feature of the XFace tool. Figure 5-5 depicts an illustration of the sub-clusters Γ_i , $1 \leq i \leq 4$, in a partitioned-job environment for the case $m_1 = m_2 = m_3 = m_4 = 4$ slave processors are contained in sub-

clusters $\Gamma_1, \Gamma_2, \Gamma_3,$ and $\Gamma_4,$ respectively, and $N = 16$ slave processors are contained in Π . The network illustrates the communication links via MPI library routine calls among processes on remote processors in the Beowulf cluster.

In general, the placement rule R used to place each slave processor $P_j, 1 \leq j \leq N,$ in each sub-cluster $\Gamma_i, 1 \leq i \leq n,$ is as follows. The maximum sub-cluster size m_{\max} allowed is an input parameter that the user enters in the job description. For a given m_{\max} and $N,$ the total number of sub-clusters n contained in Γ is given by the following expressions.

$$n = \frac{N}{m_{\max}}, \quad N \bmod m_{\max} = 0 \quad (5-1a)$$

$$n = \frac{N}{m_{\max}} + 1, \quad N \bmod m_{\max} \neq 0 \quad (5-1b)$$

The placement rule R is such that the first m_{\max} slave processors in Π are placed in $\Gamma_1,$ the second m_{\max} slave processors in Π are placed in $\Gamma_2,$ and so on. If the remainder of the division in (5-1) is zero, $N \bmod m_{\max} = 0,$ each $\Gamma_i \subset \Gamma, 1 \leq i \leq n,$ has the same size, that is, $m_1 = m_2 = \dots = m_n = m_{\max}.$ Otherwise, the remainder of the division in (5-1) is non-zero and each of the first $\Gamma_i \subset \Gamma, 1 \leq i \leq n-1,$ has the same size, that is, $m_1 = m_2 = \dots = m_{n-1} = m_{\max},$ and the last $\Gamma_n \subset \Gamma$ has size $m_n,$ where $1 \leq m_n < m_{\max}.$

The N slave processors in Π are ranked from 1 to $N,$ inclusively. The ranking is done according to the positions of the selected machines in the machine list displayed in the job description form displayed in Figures 5-6. The p processors that are contained in the first selected machine in the list are arbitrarily ranked 1 to $p.$ Similarly, the p processors that are contained in the second selected machine in the list are ranked $(p + 1)$ to $2p,$ and so on. For sub-cluster $\Gamma_i \subset \Gamma, 1 \leq i \leq n,$ the ranks of the slave processors in Γ_i are in the range $(i-1) \times m_{\max} + 1$ to $i \times m_{\max},$ inclusive. Thus, the rank of slave processor $P_j \subset \Gamma_i$ is $j,$ such that $1 \leq i \leq n$ and $(i-1) \times m_{\max} + 1 \leq j \leq i \times m_{\max}.$ The slave

processor $P_j \in \Gamma_i$ with the smallest rank is assigned the leader of sub-cluster Γ_i . For the job environment diagrammed in Figure 5-5, the slave processors with ranks 1, 5, 9, and 13 are the leaders of Γ_1 , Γ_2 , Γ_3 , and Γ_4 , respectively.

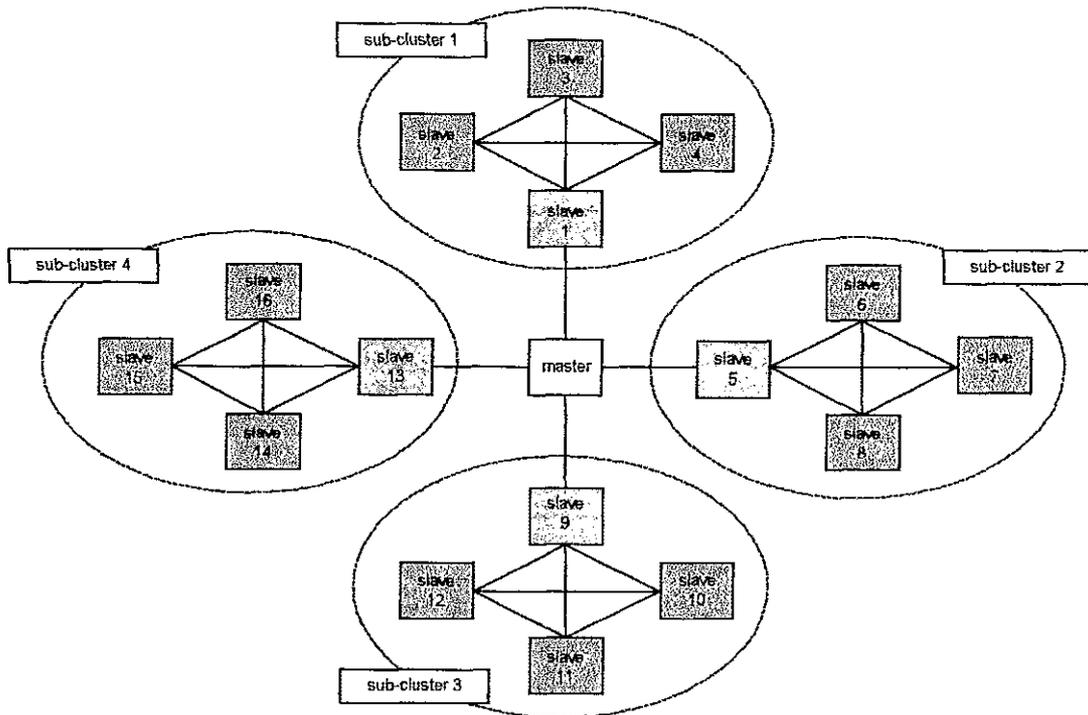


Figure 5-5. Network of Processors for $N = 16$ and $n = 4$

As before, the master processor runs an instance of the job scheduler. However, in this case, the job scheduler schedules the iterations only to the leaders of the sub-clusters. Each leader runs an instance of the task scheduler. For each iteration that is executed on Γ_i , the task scheduler implements Algorithm 5-1 to schedule the processors in Γ_i . All slave processors in Γ_i , including the leader, are scheduled to tasks in the partitioned program.

5.7. Partitioned Job Description

The job description form that is used to submit partitioned jobs is displayed in Figure 5-6. Partitioned jobs with Linux-based application programs are submitted using the version of the form in Figure 5-6 (a), and partitioned jobs with Windows-based application programs are submitted using the version in Figure 5-6 (b). The total number of tasks ϕ , $\phi \geq 2$, in the partitioned program is first entered into the “Total Tasks” text box. The number of replications is then entered in the “Total Replication” text box. If the application is Linux-based, the command to execute the program is entered in the “Command” text box in the form shown in Figure 5-6 (a). Otherwise, the application name is entered in the “Application” text box in the form shown in Figure 5-6 (b). The machines to execute the job are then selected from the machines list. Finally, a set of job files (task files) is entered for each task. For each set of task files, the task index is first entered, and then the task files are entered in the text fields provided. The “Submit Files” button is used to submit the task files. The tasks that are already submitted are displayed in the “Task Files Submitted” window. The “View Submitted Files...” button displays the task files submitted for the selected task, and the “Delete” button deletes the task files submitted for the selected task.

Platform Options:
 Linux Emulated Windows

Number of Tasks: Total Tasks: 2
 Number of Replications: Total Replications: 1

Program Execution Commands:
 Command:

Select the Machines to Execute the Job:
 Max Subcluster Size Allowed: Max Subcluster Size: 2
 Select Machines:
 n01.vwssc.odu.edu
 n02.vwssc.odu.edu
 n03.vwssc.odu.edu
 n04.vwssc.odu.edu
 n05.vwssc.odu.edu
 n06.vwssc.odu.edu
 n07.vwssc.odu.edu

Job Files:
 Enter Task Precedence Graphs:
 Enter Job Files For A Specific Task:
 Enter Task Index for which Job Files are being entered:
 Task Executable:
 Seeds File:

Tasks Files Submitted:

(a)

Platform Options:
 Linux Emulated Windows

Number of Tasks: Total Tasks: 2
 Number of Replications: Total Replications: 1

Windows Application Name:
 Application:

Select the Machines to Execute the Job:
 Max Subcluster Size Allowed: Max Subcluster Size: 2
 Select Machines:
 n01.vwssc.odu.edu
 n02.vwssc.odu.edu
 n03.vwssc.odu.edu
 n04.vwssc.odu.edu
 n05.vwssc.odu.edu
 n06.vwssc.odu.edu
 n07.vwssc.odu.edu

Job Files:
 Enter Task Precedence Graphs:
 Enter Job Files For A Specific Task:
 Enter Task Index for which Job Files are being entered:
 Task Executable:
 Seeds File:

Tasks Files Submitted:

(b)

Figure 5-6. Partitioned Job Description Forms (a) Windows-based (b) Linux-Based

CHAPTER VI IMPLEMENTATION

This chapter presents the implementation of the XFace tool. The top-level layout of the core processes and scripts on the master processor and each slave processor are illustrated in Figures 6-1 and 6-2. The figures diagram the hereditary relationships among the major processes in the XFace tool. The darker shaded blocks represent the fundamental processes implemented in the tool. Blocks that are contained within shaded blocks represent the components of the shaded blocks. The lighter shaded blocks represent some of the major auxiliary shell scripts that complement the XFace implementation. There are several other auxiliary scripts in the tool that are not illustrated in Figures 6-1 and 6-2. Many of these scripts perform very simple housekeeping chores that are too trivial to discuss. Thus, for the conciseness of this chapter, the simple housekeeping scripts are not described.

The XFace Launcher script starts an instance of the XFace application by starting the XFace parent process. The XFace parent process first runs clean-up scripts to initialize the job environment on all processors, and then launches the Xdisplay process. The Xdisplay process displays the job description forms used to submit jobs to the tool. After the submission of a job description, the Xdisplay process runs a set of auxiliary scripts that create the iteration directories, and a second set that loads the job on the slave processors. After the job description is loaded on the slave processors, the Xdisplay process signals the XFace parent process. The reception of this signal triggers the XFace parent process to start the master control process to execute the job. The master control process implements the job scheduler that runs on the master processor, and also the instances of the monitor process that run on the slave processors. The job scheduler schedules the iterations to the instances of the monitor process. The instances of the monitor process initiate and monitor the executions of the iterations. The master control process signals the XFace parent process at the completion of the execution phase. The XFace parent process then runs a set of auxiliary scripts that gather the results from the slave processors to the master processor. The upcoming subsections give detailed

descriptions of the more important processes and scripts illustrated in Figures 6-1 and 6-2.

The XFace parent process is responsible for initializing the job environments, and starting the Xdisplay and master control processes. At start up of the XFace application, the clean-up scripts are run to initialize the job environment on the machines listed in machines.LINUX. During the initialization step, the shared memories and semaphores that failed to de-allocate at the end of the previous job are released. Also, the XFace-related zombie processes, if any, that existed on the machines are killed. The initialization step is very important because every instance of the XFace tool uses the same set of keys to allocate the semaphores and shared memories used for inter-process communications (IPCs). As a result, an instance of the XFace tool will not start if the previous instance was not properly exited and the semaphores and shared memories were not successfully de-allocated. Due to the importance of cleaning up the job environments on the processors involved, the clean up scripts are run both at start up of the XFace application and also at the end of every job that is executed by the tool. A second consequence of using the same set of keys to initialize shared memories and semaphores is that the current implementation of the XFace tool only allows for the execution of one job at a time. It does not support executing multiple jobs concurrently.

The Xdisplay process displays an input GUI window, a monitor GUI window, and the output GUI window at different times. The input GUI window is displayed at start-up of the XFace tool. It is used to select one of the job description forms displayed in Figures 3-6, 4-2, 4-3, and 5-6 that are used to submit jobs to the tool. The monitor GUI window is displayed at the beginning of the job execution phase. It displays the current execution status and statistics of each iteration. It also provides functionalities that allow the user to abort the job execution, and to view the submitted job description at anytime during the job execution phase or job completion phase. Finally, the output GUI window is displayed at the end of the job completion phase. It displays the job run statistics, and provides the user with the functionalities to save the job results and start a new job.

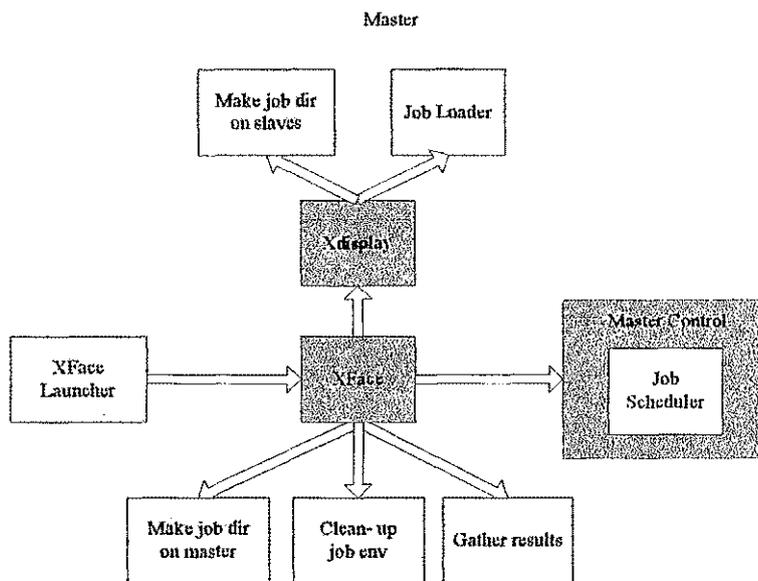


Figure 6-1. Hereditary Relationships Among Core XFace Processes on Master

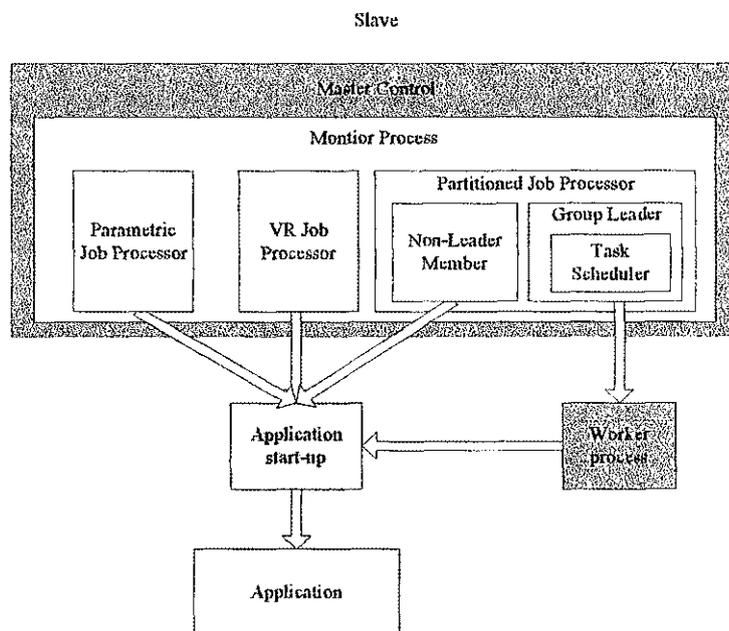


Figure 6-2. Hereditary Relationships Among Core XFace Processes on Slave

The master control process is the backend of the XFace tool that executes the iterations of jobs submitted to the tool. The same instance of the master control process runs on the master processor and also on the slave processors. The part of the master control process that runs on the master processor implements the job scheduler, and the part of the master control that runs on the slave processors implements the monitor process.

The job scheduler is started at the beginning of the job execution phase. During the job execution phase, the job scheduler employs Algorithm 3-1 to schedule the iterations to instances of the monitor process. Each scheduled instance of the monitor process forwards job execution statistics to the job scheduler. The execution statistics include the iteration start time, the iteration end time, and the iteration total execution time. The job scheduler communicates the execution statistics to the Xdisplay process, which in turn uses this information to update the information displayed in the monitor GUI window.

An instance of the monitor process is started on each slave processor $P_j \subset \Pi$ at the beginning of the job execution phase. The instance of the monitor process running on P_j initiates and monitors the execution of the iterations that are scheduled to P_j . The monitor process runs until it is sent the termination signal by the job scheduler. The termination signal can be either of type I, or type II. If a type I termination signal is sent, the monitor process aborts the execution of the current iteration and waits for further instruction from the job scheduler. Further instruction in this case could be either the scheduling of another iteration or the termination signal of type II. If a type II termination signal is sent, the monitor process exits execution. Currently, only the termination signal of type II is implemented. The termination signal of type I is reserved for future expansion of the tool. It will be implemented to abort the execution of a particular iteration from the monitor GUI window.

Depending on the job-type, different instances of the monitor process run on the slave processors. This is illustrated in Figure 6-2. The instance of the monitor process that runs on the slave processors is specific to the job-type. The *parametric job processor* runs on the slave processors if the job-type is parametric; the *VR job processor* runs on the slave processors if the job-type is VR; and the *partitioned job processor* runs on the

slave nodes if the job-type is partitioned. For partitioned jobs, each sub-cluster Γ_i contains one processor that is designated the leader and $(m-1)$ that are designated non-leaders. All the leaders run the same instance of the monitor process. Similarly, the non-leaders run the same instance of the monitor process, which is different from the instance that is run on the leaders.

The task scheduler is a component of the monitor process that runs only on the leaders. For each iteration that is scheduled to the leader P_j in sub-cluster Γ_i , the task scheduler running on P_j employs Algorithm 4-2 to schedule the slave processors in Γ_i to the tasks in the partitioned program. Therefore, the leaders execute the task scheduler as well as the tasks in the partitioned program that are scheduled to the leaders. The execution of the tasks scheduled to each leader is off-loaded to an auxiliary process, the worker process. The worker process that runs on a leader is dedicated to that leader. This is illustrated in Figure 6-2. Thus, during the job execution phase, the task scheduler and the worker process run concurrently on each leader in Π . The task scheduler schedules the tasks in the partitioned program, while the worker process executes the tasks that are scheduled to run on the leader.

The application start-up scripts are used to initiate the application programs that are replicated in the execution of the iterations. There are two sets of application start-up scripts: Windows-based and Linux-based. The Windows-based start-up scripts are used to initiate the execution of Windows-based application programs, while the Linux-based start-up scripts are used to initiate the execution of Linux-based application programs. The application start-up scripts are further described in Appendix B.

6.1. Implementing the Job Environments

This section describes the implementation of the job environments that are created by the XFace tool. The virtual network topologies in Figures 3-2 and 5-5 are easily implemented on top of the MPI library. By default, when the master control process is started on the master processor and slave processors, the processors are placed into one group, notated as the *world-group*. A communication channel is automatically established

between every pair of processors in the world-group. Thus, master-slave communication channels are established between the master processor and each slave processor, and slave-slave communication channels are established between every pair of slave processors. The instance of the master control process that runs on each of the processors in the world-group are ranked with natural numbers in the range 0 to N. The instance of the master control process that runs on the master processor is automatically ranked 0, and the instances that run on the slave processors are ranked from 1 to N. Each processor is tagged with a processor identification number (ID) that is identical to the rank of the instance of the master control process that runs on that processor. Thus, the master processor has processor ID = 0, the slave processor running the instance of the master control process with rank 1 has processor ID = 1, and so on. As a result, when the instances of the monitor process are grouped, the slave processors are automatically grouped in a similar fashion.

For parametric and VR jobs, the slave-slave channels are disregarded since they are not needed. The job scheduler uses the master-slave channels to schedule the iterations to the instances of the monitor process. Figure 6-3 diagrams the remote communications between the job scheduler and the instances of the monitor process running on the slave processors. Figure 6-3 (a) diagrams a general view of the communication channels, while Figure 6-3 (b) diagrams a zoomed view of the communication channel between the job scheduler and an instance of the monitor process. The job scheduler communicates the iteration numbers and the termination signals to the instances of the monitor process. An instance of the monitor process that is scheduled an iteration starts the appropriate application-start-up script to initiate the execution of the iteration, and then sends the iteration start-time to the job scheduler. At the end of the iteration, the instance of the monitor process sends the iteration end time, total execution time, and a request for another iteration to the job scheduler. If all iterations are not scheduled, the job scheduler fulfils the monitor process request by scheduling the next iteration to the slave processor on which the monitor process runs. Otherwise, the job scheduler sends the termination signal of type II. This causes the monitor process to terminate its execution.

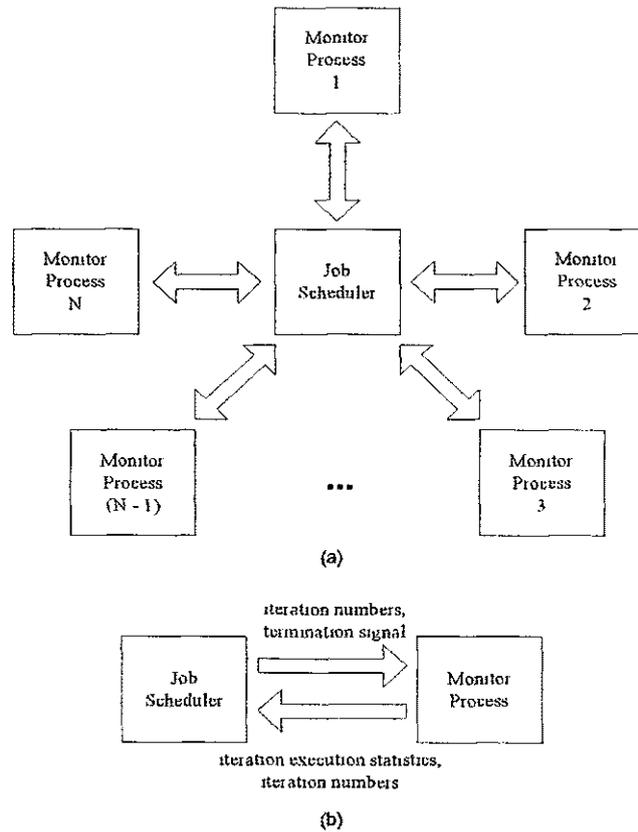


Figure 6-3. (a) Remote Communication between Master and Slave Processors (b) Messages passed between Job Scheduler and Monitor Process

For partitioned jobs, each sub-cluster is implemented as a sub-group of the world-group, which is also easily implemented on top of the MPI library. Sub-groups are created from the world-group, and the sub-groups constitute the sub-clusters, none of which contains the master processor. The sub-clusters are created as follows. Each instance of the monitor process invokes the MPI routine `MPI_Comm_split()` that places the instance of the monitor process instantiating the invocation in a sub-cluster. The `MPI_Comm_split()` routine takes the tuple `<color, key>` as two of its arguments. All instances of the monitor process invoking `MPI_Comm_split()` with the same color value are placed in the same sub-cluster. The key values are used to rank the instances of the

monitor process in the newly formed sub-cluster. The placement rule R described in Section 5.6 is used to generate the number of sub-clusters n that is contained in Γ . Each instance of the monitor process that belongs to sub-cluster Γ_i uses color value = i and key value equal to its rank in the world-group, in the invocation of `MPI_Comm_split()`. Therefore, after the invocations of `MPI_Comm_split()`, the monitor processes, and hence the slave processors, are placed into sub-clusters in accordance with the placement rule R . After the creation of the sub-clusters, each slave processor in Γ_i has two ranks: its rank (world rank) in the world group, and its rank (sub-cluster rank) in sub-cluster Γ_i . The world rank of the slave processor $P_j \subset \Gamma_i$ is a natural number in the range 1 to N , and the sub-cluster rank is a natural number in the range 0 to $(m_i - 1)$. The world rank of P_j corresponds with the rank defined in Section 5.6.

Figure 6-4 illustrates the remote communication channels between the job scheduler and the instances of the monitor process that run on the leaders for the setup in Figure 5-5. The job scheduler schedules the iterations to the instances of the monitor process that run on the leaders. As before, the job scheduler communicates iteration and termination signals to the instances of the monitor process that run on the leaders. The instances of the monitor process that run on the leaders communicate the execution statistics of the iterations and iteration requests to the job scheduler. The execution of each iteration that is scheduled to the leader of sub-cluster Γ_i replicates the partitioned program once on sub-cluster Γ_i . The task scheduler running on the leader schedules the slave processors in Γ_i to the tasks in the partitioned program. The diagram in Figure 6-5 illustrates the remote communications between the task scheduler running on the leader $P_1 \subset \Gamma_1$ and the non-leader slave processors in Γ_1 , where Γ_1 is sub-cluster 1 in the diagram depicted in Figure 6-4. Figure 6-5 also illustrates the IPC between the task scheduler and the worker process. The remote communications between the task scheduler and the instances of the monitor process running on the non-leaders in Γ_1 are achieved via slave-slave communication channels. However, the IPC between the task scheduler and the worker process is achieved via shared memory. For each task that is scheduled to a slave processor in Γ_1 , the task scheduler communicates the task index and

the locations of the data that the task needs to begin execution. In either case, the worker process running on the leader or the instance of the monitor process running on a non-leader communicates task execution statistics and task requests to the task scheduler. When all tasks have been executed, the task scheduler sends the termination signal of type II to all requesting slave processors in Γ_1 .

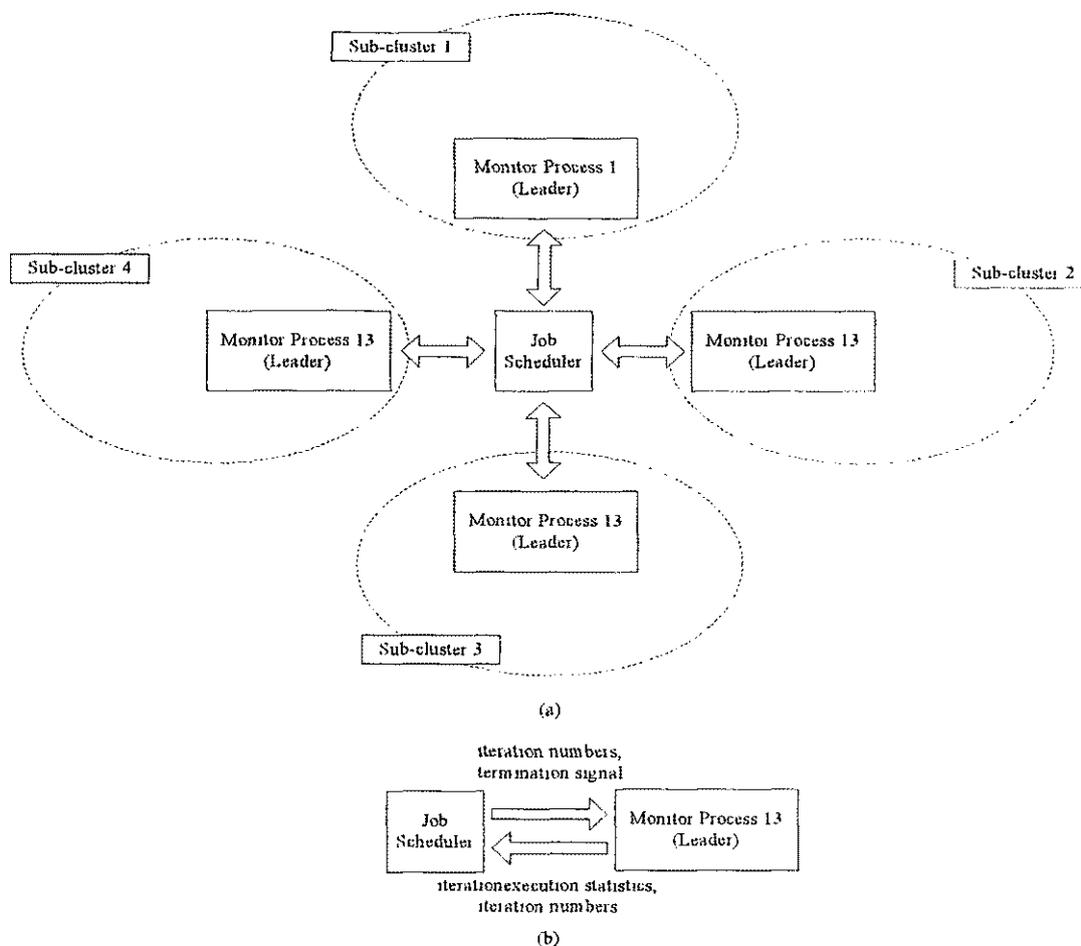


Figure 6-4. (a) Remote Communication between Master and Leaders (b) Messages passed between Job Scheduler and Monitor Process on Leader

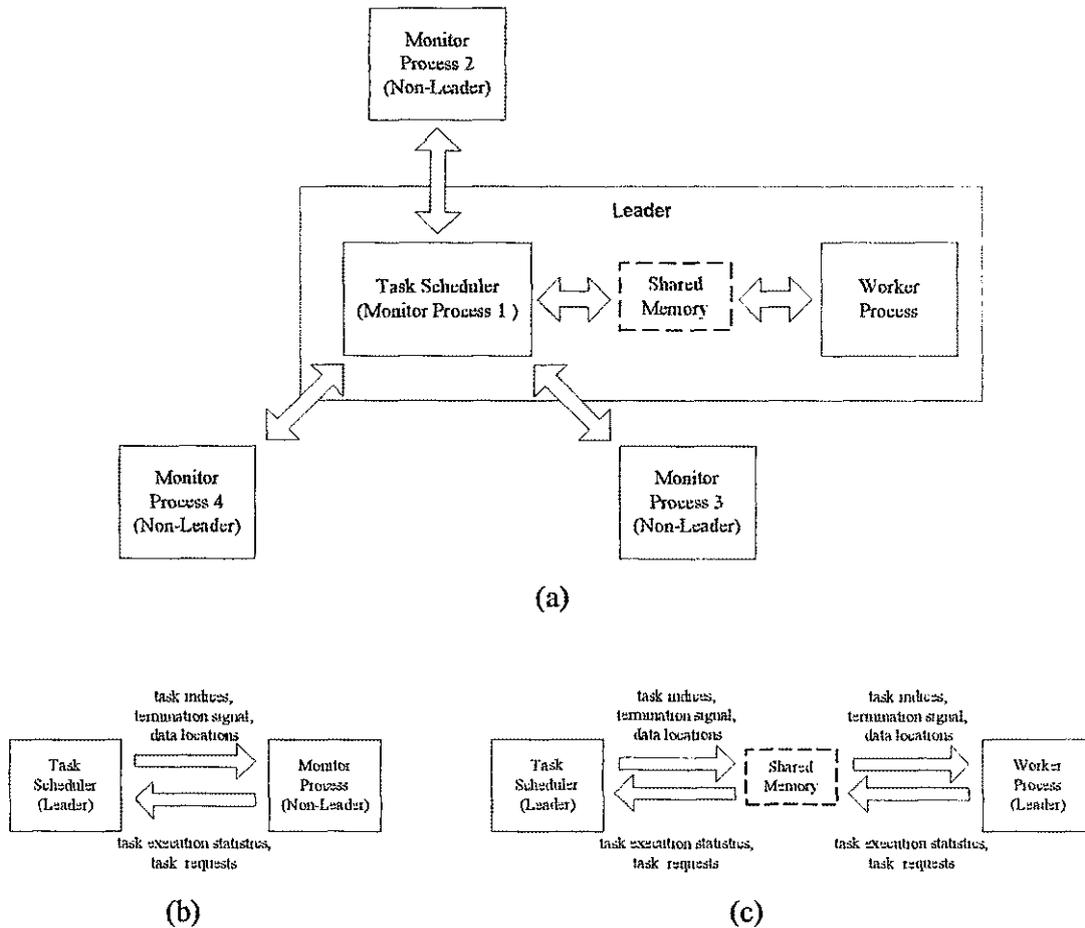


Figure 6-5. (a) Communication between Leader and Non-leaders in Γ_1 (b) Messages passed between Task Scheduler and Monitor Process on Non-Leader (c) Messages passed between Task Scheduler and Worker Process

6.3. Process Interactions on Master Processor

The Linux operating system provides a rich source of inter-process communication mechanisms that allow processes running on the same processor to communicate with each other and also with the system kernel [25], [26]. Processes running on the master processor communicate via text files, shared-memory, and signals.

The diagram in Figure 6-6 depicts the IPCs among the processes on the master processor. The XFace parent process and the Xdisplay process communicates via signals and shared memory. When a job description is submitted, the Xdisplay process writes the number of processors in the set Π to the shared-memory interface between the Xdisplay process and the XFace parent process. The XFace parent process reads this value and uses it to start the master control process. Signals are used to signal asynchronous events among the processes in the figure.

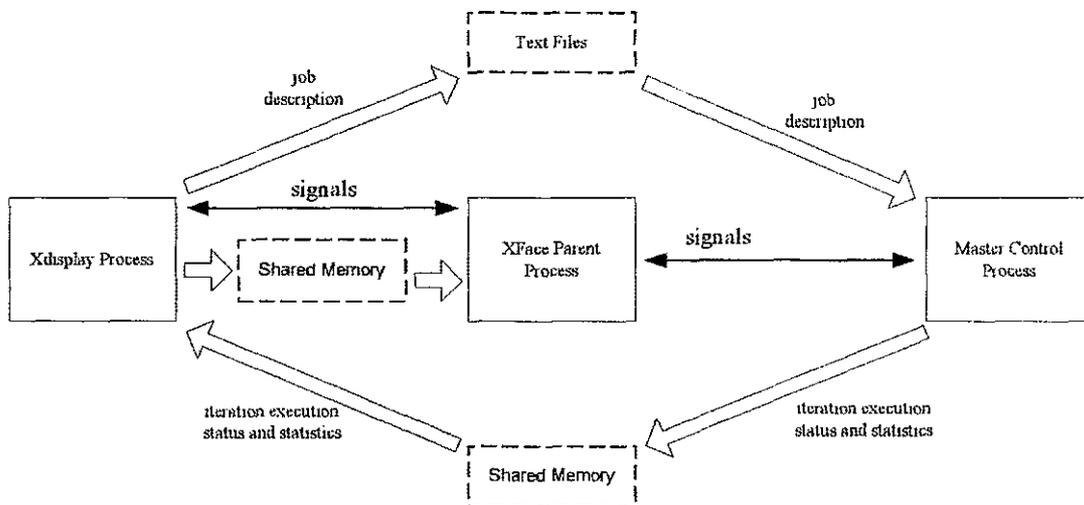


Figure 6-6. IPCs on Master Processor

The Xdisplay process and the master control process communicate via text files and shared-memory. After the submission of a job, the Xdisplay process writes the job description to text files. The alias names of the salve machines are written to the file *machines.LINUX*, and the remaining portion of the job description is written to the file *jobDescription.txt*. These input files are read by the master control process and are used to initiate the job. The contents of *machines.LINUX* are used by the MPI resources to

launch the master control process on the slave processors, and the master control process uses the contents of both files during the job execution phase.

During the job execution phase, the job scheduler writes current execution status and statistics to the shared-memory interface between the master control process and the Xdisplay process. The Xdisplay process reads the contents of the shared-memory and updates the information displayed in the monitor GUI window. Synchronization of all write and read operations performed on the shared-memories is implemented using semaphores [26].

6.4. Process Interactions on Slave

The IPCs among the processes on each slave processor depend on the job type. Figure 6-7 diagrams the IPCs on the slave processor in parametric and VR job environments. The diagram in the figure portrays the communication achieved between processes via command line arguments. For each iteration that is scheduled to a slave processor, the monitor process spawns the application-start-up script to execute the iteration and passes command line arguments. The command line arguments passed are the name of the application-start-up script, the iteration number, the configuration number if the job type is VR-CRN, the job type, the alias name of the machine executing the iteration, the pathname to the application program, and the application name if the platform is Windows or program execution command if the platform is Linux. Therefore, when the application start-up script starts, it has all the necessary information to initiate the execution of the iteration.

The setup is slightly different for partitioned jobs. Figure 6-8 diagrams the IPCs among the task scheduler, the worker process, and the application start-up script. The task scheduler uses the shared-memory interface to communicate task indices and termination signals to the worker process. Likewise, the worker process uses the shared-memory interface to communicate the task execution statistics and task requests to the task scheduler. As usual, signals are used to handle asynchronous events between the task scheduler and the worker process. The worker process writes the remote locations of the data that are needed to start the execution of the task to a shared text file interface. The

worker process then spawns the application start-up script to execute the task, passing the following command line arguments: the name of application start-up script, the iteration number, the task index, the alias name of the slave machine, and the application name if the platform is Windows or the task execution command if the platform is Linux. The start-up script copies the data from the remote locations specified in the text file interface, and then initiates the execution of the task.

The IPCs among the processes on a non-leader processor is illustrated in Figure 6-9. The IPCs in the figure are as described in Figure 6-8 for the IPCs between the worker process and the application start-up script.

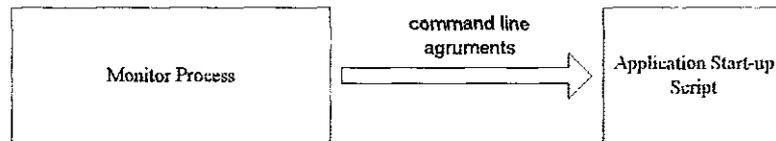


Figure 6-7. IPCs on Slave Processor in Parametric and VR Jobs Environment

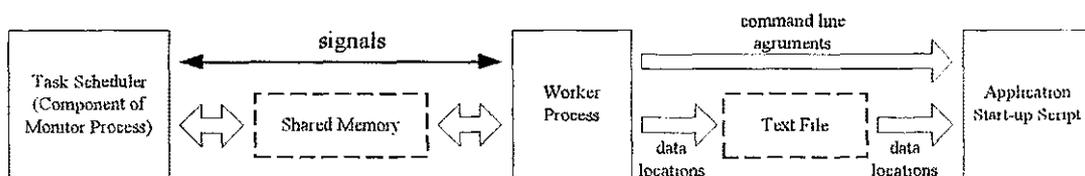


Figure 6-8. IPCs on Leaders in Partitioned Job Environment

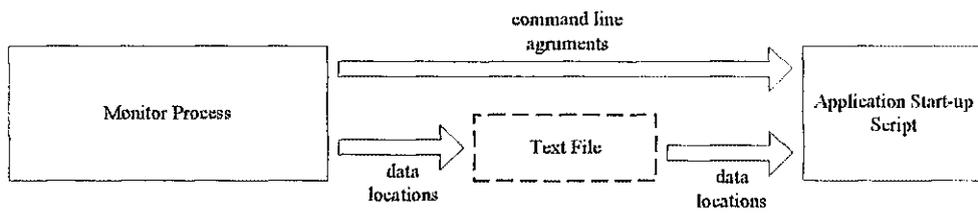


Figure 6-9. IPCs on Non-leader in Partitioned Job Environment

CHAPTER VII CASE STUDIES

This chapter presents three case studies that demonstrate the features of the XFace tool. A case study is presented for each job-type: parametric, VR, and partitioned. The parametric case study is aimed at illustrating the speed-up in the execution time that is achievable for the application program used in the case study. Since the VR and partitioned job-types are extensions of the parametric job-type, the descriptions of the results obtained for the parametric case study are applicable to the results obtained for the VR and partitioned case studies. Therefore, the VR case study is focused on illustrating that the environment the tool creates supports variance reduction under the VRTs presented in Chapter 4. The VR case study employs variance reduction under AV. On the other hand, since the speed-up in the 1st dimension is the same as the speed-up in parametric jobs, the partitioned case study is aimed at demonstrating the speed-up in the execution times in the 2nd dimension.

The target application for the parametric and VR case studies is the student version of the Windows-based application Arena-Version 5.00.2 [1]. At the time of this writing, the current implementation of Arena is designed to run sequentially on single processors. This is also true for other similar commercial discrete event simulation packages. Therefore, due to the unavailability of a simulation application that supports the partitioned feature of the tool, an example program was designed and implemented in the C programming language [27] to demonstrate the third feature of the Xface tool. For brevity in the case studies, the application programs used are made very simple.

7.1. Arena Application-specific Environment

The application-specific environment the XFace tool creates for the Arena application is described in this section. The diagram in Figure 7-1 depicts the Arena-specific execution environment created on each slave processor to execute Arena programs. The environment depicted in the figure is an instance of the environment

diagramed in Figure 3-3. The automation control program that controls the Arena application is implemented in Visual Basic for Applications (VBA) [28]. An implementation of WINE [29] is used to create the Windows environment that is required to run Arena and the VBA automation control program.

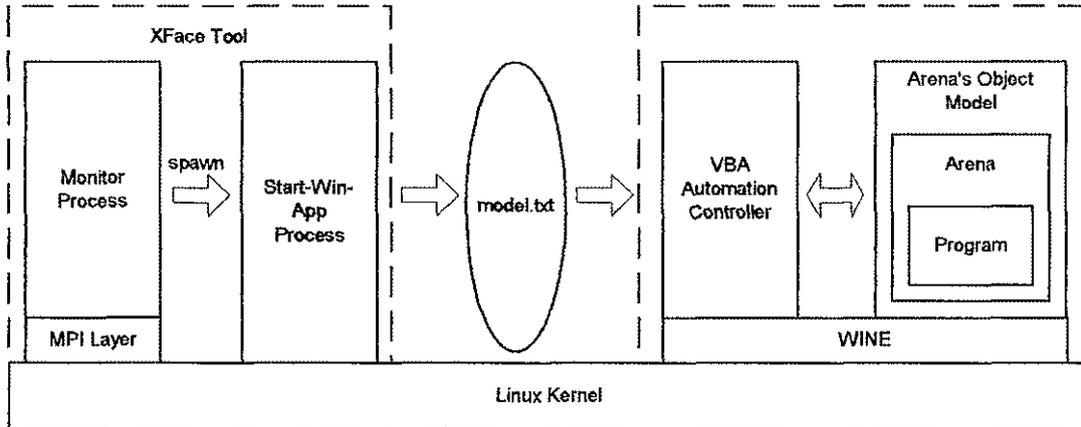


Figure 7-1. Arena Application-specific Execution Environment

The Arena application supports many embedded VBA events that are triggered at various points throughout the simulation run of an Arena program (model) [1]. Each event is handled by a unique service routine that is initiated by Arena when an event is triggered. By default, the event service routines are void of VBA codes. VBA code must be written within each service routine if the routine is to do anything useful when it is invoked. A brief description of the Arena-initiated VBA events that are triggered during the model simulation run are as follows. The event *RunBegin* is triggered prior to Arena checking and initializing the model; the event *RunBeginSimulation* is triggered once after the model is checked and initialized; the event *RunBeginReplication* is triggered at the start of every replication of the simulation; the event *RunEndReplication* is triggered at

the end of every replication; the event *RunEndSimulation* is triggered at the end of the simulation while the generated simulation data are still available; and the event *RunEnd* is triggered at the very end of the simulation run. Detailed descriptions of the Arena-initiated events, the event service routines, and the points in the simulation run cycle where the events are triggered are presented in [1].

The event *RunBegin* allows changes to be made to the structural properties of the modules in the Arena model before the model is checked and initialized. For each Arena model that is executed in the environment diagrammed in Figure 7-1, the *RunBegin* event is used to implement the method proposed in Section 5.4 to maintain the independence among the replications. The service routine that is associated with the *RunBegin* event is *ModelLogic_RunBegin*. The VBA code implemented in *ModelLogic_RunBegin* reads the streams file described in Section 3.4 and sets the random number streams in the model. The VBA code that is implemented in *ModelLogic_RunBegin* for the Arena models that are used in the case studies is listed in Appendix B.

7.2. Parametric Case Study

The Arena model that is used for the parametric case study is illustrated in Figure 7-2. The model is a simple single-server queuing system. The Entity Creation module creates entities arriving in the system, with inter-arrival times selected from an exponential distribution with expected value one minute. The Server module models the server that services the entities. Upon entering the system, if an entity finds the server idle, the entity goes directly into service. Otherwise, the entity waits in a queue until the server becomes available. An entity seizes the server resources at the beginning of service, and releases them when service is complete. The entity service time is modeled as a delay drawn from an exponential distribution with mean five minutes. An entity that has completed service departs the system via the Entity Disposal module.

The random number stream U^1 is used to generate the entity inter-arrival times, and the random number stream U^2 is used to generate the service time delays. The random number streams U^1 and U^2 are defined in the Seeds object. For each stream, the Seeds object defines the seed value that is used to initialize the stream, and the method of

reinitializing the stream between replications. Arena offers four options to reinitialize the streams between replications: No, Yes, AV, and CRN. The “No” option directs Arena not to reinitialize the streams between replications, the “Yes” option directs Arena to reinitialize the streams, and the “AV” and “CRN” options direct Arena to reinitialize the streams with the AV and CRN VRTs built into the Arena application. In the case studies that use Arena programs, the method used to reinitialize the streams between replications is only important for VR-AV jobs in which exactly two replications are executed per iteration. For the VR-AV case study, the method of reinitializing the streams is set to AV. For the other job-types, only one replication is executed per iteration. Therefore, reinitializing the stream between replications is not applicable, so the default option “No” is specified.

Model 01 is setup to advance the simulation clock in real time. The simulation stopping condition is set to stop the simulation after 5 minutes of real time has elapsed since the start of the simulation, regardless of the state of the system. Therefore, the length of each replication is fixed at 5 minutes, or equivalently 300 seconds. Hence, ideally, the sequential time T_S to execute 10 replications of Model 01 is 3000 seconds.

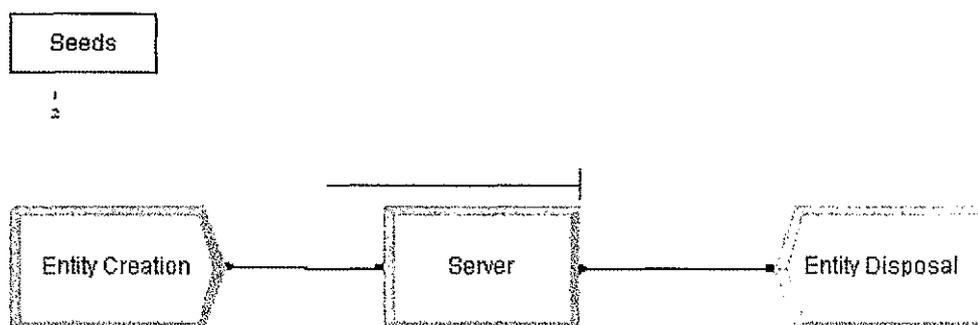


Figure 7-2. Arena Model 01

7.2.1. Job Description

A sample of the job description submitted for the parametric case study is recorded in Table 7-1. The job-type indicates that the parametric job description form is used to submit the job. The job-type directs the XFace tool to set up the parametric environment to execute the job. The next two lines in the job description indicate that 10 replications of the application program will be executed on the Windows platforms created on the slave processors. The next lines specify that the target application is Arena, and the application program that the tool will execute in each job run is Model 01. One slave processor aliased n01.vmasc.odu.edu will execute all 10 replications of Model 01. Finally, the last two lines in the job description specify the pathnames to Model 01 and the streams file that will be used to initialize U^1 and U^2 in each replication.

A total of fifteen job descriptions similar to the one listed in Table 7- 1 were submitted at different times to the XFace tool, with the number of slave processors N varying from 1 to 15.

Table 7-1. Parametric Case Study Job Description

Job Description	
Job type	: PARAMETRIC JOB
Platform to Run Application	: Windows
Number of Replications	: 10
Windows Application	: ARENA
Windows Application Program	: Model 01.doe
Machines Selected to Execute the Job:	
n01.vmasc.odu.edu	
Job Input Files:	
/home/jhead/Examples/param_example/WIN/Model01/Model 01.doe	
/home/jhead/Examples/param_example/WIN/Model01/seeds.txt	

7.2.2. Results

The execution times of the jobs submitted to the XFace tool for the parametric case study are displayed in the plots in Figure 7-3. The theoretical curve is a plot of the theoretical execution times versus the number of processors N . The theoretical execution times were computed for $N = 1, 2, \dots, 15$ and $T_S = 3000$ seconds using Equation (3-2). The actual curve is a plot of the actual execution times resulting from the experiment versus the number of processors N . A comparison of the plots reveals that the resulting job execution times are as expected since the actual curve closely matches the theoretical curve. A major difference between the curves is that each data point in the actual curve is above its counterpart in the theoretical curve. For each N , the difference between the actual execution time and the theoretical execution time is attributed to the overhead T_O that is associated with each batch of replications that is executed on the N processors. This overhead is attributed to the scheduling overhead, the communication overhead, and the overhead to run the Arena application on the Windows emulator.

A plot of the overhead versus N is displayed in Figure 7-4. The overheads are expressed as multiples of the overhead for $N = 10$. The overhead is minimal for $N = 10$, and is approximately 1.70 percent of the sequential time T_S . This is due to the fact that the replications are grouped into one batch; thus, only one batch setup time is experienced. Contrastingly, the overhead is worst for $N = 1$ since the replications are grouped into ten batches and ten batch setup times are experienced.

A plot of the speedup in the total simulation execution times against N is illustrated in Figure 7-5. For each N , the speedup was computed using Equation (3-4) with $T_S(300, 10) = 3000$ seconds and the parallel execution times $T_{P_{\text{actual}}}(300, 10, N)$ taken from the plot in Figure 7-3. As N increases from one to five, the speedup in the execution times increases almost linearly with N , then remains constant as N increases from five to nine, increases for $N = 10$, then finally fluctuates about a constant value for $N > 10$. The shape of the curve in Figure 7-5 is explained as follows. For a given k and N , suppose $p = \left\lfloor \frac{k}{N} \right\rfloor$ and $q = k \bmod N$. The number of batches decreases and the batch size increases as N increases from one to five. Hence, the number of replications that are

executed concurrently on the N processors increases. However, the increase is not linear because for $N = 1, 2,$ and 5 , $q = 0$ and the processors are fully utilized, whereas for $N = 3$ and 4 , q is non-zero and the processors are not fully utilized. For example, for $N = 1$, ten batches each having batch size equal to one are created. Thus, the replications are sequentially executed since each batch size is one. Actually, for $N = 1$, the tool performs worse than sequentially executing the replications on a single processor without the use of the tool because of the total overhead that is associated with the ten batches. As a result, the speedup for $N = 1$ is less than one. However, for $N = 2$, the replications are executed concurrently in five batches having size equal to two. The effect is a reduction in the total execution time of the ten replications by almost a half. Hence, the speedup in the total execution for $N = 2$ almost doubles, achieving a value that is a little less than two. For $N = 3$, the replications are executed concurrently in four batches, with the first three batches having size equal to three and the last batch having size equal to one. During the execution of the batch with size one, two of the three processors submitted to execute the replications are not used. Hence, the speed-up in the total execution time for $N = 3$ is not three-fold. The case for $N = 4$ is similar to the case for $N = 3$. For $N = 5$, the replications are executed concurrently in two batches of 5, thus, the speedup achieved is almost five-fold. Now, for N between 6 and 9, inclusively, the replications are executed concurrently in two batches that are of different sizes. During the execution of the first batch, N replications are executed concurrently. However, only q replications are executed concurrently in the second batch, and $(N - q)$ processors are idle. Therefore, the speed-up in the total execution time remains constant for $N = 5$ to $N = 9$. Finally, for $N = 10$, the replications are executed concurrently in one batch of size 10, which results in the maximum speedup in the total execution time. For $N > 10$, the excess $(N - k)$ processors are idle during the execution of the replications; hence, the speedup remains constant.

In general, the dependence of the speedup in the job total execution time on the number of processors will be similar to that displayed in the curve in Figure 7-5. Furthermore, the application most efficiently utilizes the N processors submitted to execute the job when N divides k evenly.

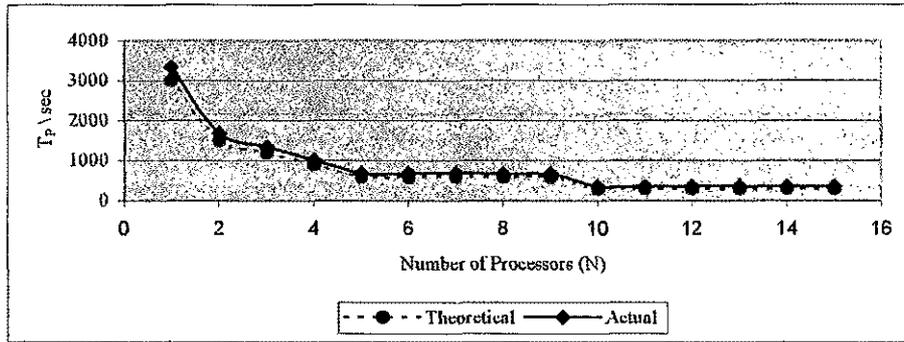


Figure 7-3. Plots of Theoretical and Actual Execution Times vs. Number of Processors

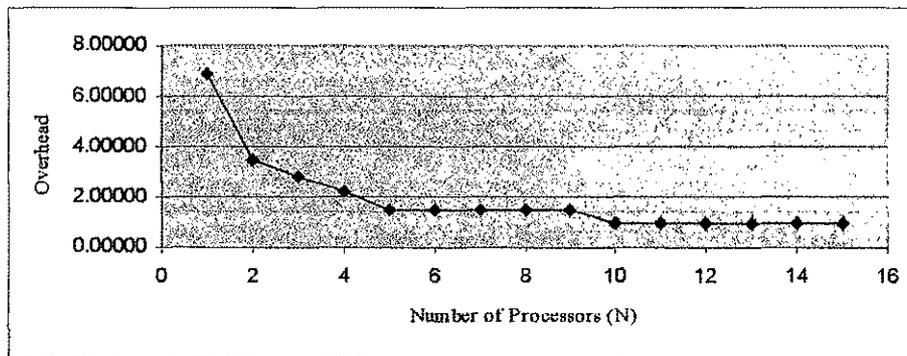


Figure 7-4. Plot of Total Overhead vs. Number of Processors

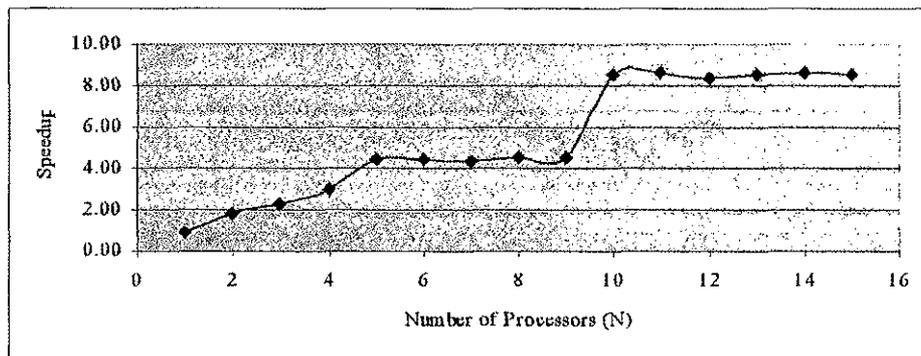


Figure 7-5. Plot of Speed-up vs. Number of Processors

7.3. Variance Reduction under AV Case Study

The model used for the VR-AV case study is shown in Figure 7-6. The model is also a single-server system. The server is modeled with Seize, Delay and Release blocks. The inter-arrival times of the entities entering the system and the service times of the server are generated from exponential distributions with means of two minutes and four minutes, respectively. The maximum capacity of the queue in the system is set at 100 entities. The Count block is used to count the number of entities that have entered service. The simulation is stopped after the one-hundredth entity entering the system begins service. Thus, a total of 100 delays are observed in the queue at the end of each replication. The Reports and ReportLines blocks are used to write the average of the 100 delays experienced in the queue for each replication to the report file “Avg Delays In Queue,” and the finish time for each replication to the report file “Replication Finish Times.”

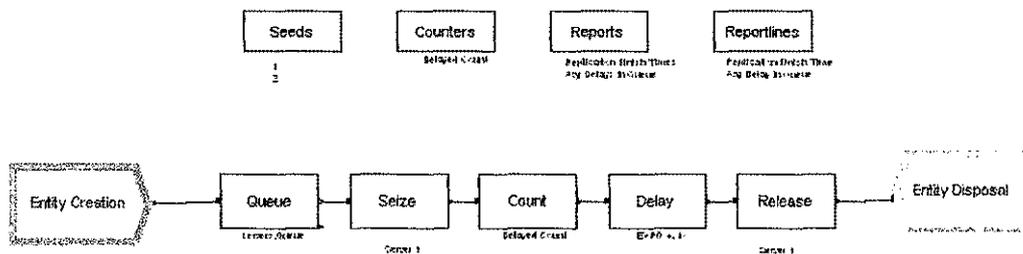


Figure 7-6. Arena Model 02

The two streams U^1 and U^2 that drive the simulation of the model are defined in the Seeds block. During the simulation, U^1 is used to generate the inter-arrival times and U^2 is used to generate the service times. Two sets of experiments are conducted. In the first

experiment, the model is executed using the parametric feature of the XFace tool. This version of the model is Model 02-a. Model 02-a is replicated 100 hundred times. Thus, 100 hundred average delays in queue, X_1, X_2, \dots, X_{100} are observed. In the second experiment, the model is executed using the VR feature of the XFace tool, and the AV VRT described in Section 4.3 is applied to the simulation. This version of the model is Model 02-b. One hundred paired replications of Model 02-b are executed, constituting a total of 200 replications. In the first replication of each pair, the random numbers U_1^1 and U_1^2 are used to generate the inter-arrival and service times, respectively, while $1 - U_1^1$ and $1 - U_1^2$ are used in the second replication. Thus, 100 pairs of average delays in queue, $(X_1^1, X_1^2), (X_2^1, X_2^2), \dots, (X_{100}^1, X_{100}^2)$, are observed, where X_j^1 and X_j^2 result from the first and second replications of each pair, respectively

7.3.1. Job Description

Two job descriptions are submitted to the XFace tool at different times. The job description submitted in the first experiment is listed in Table 7-2. The job is submitted as a parametric job. A total of 30 processors are submitted to execute one hundred replications of Model 02-a. The job description submitted to the tool in the second experiment is listed in Table 7-3. The job is submitted as a VR-AV job, and a total of 30 processors are also submitted to execute 100 replications of Model 02-b.

7.3.2. Results

The 100 average delays in queue obtained from the first experiment are allocated into 10 equally spaced bins. The bin contents are plotted in the histogram in Figure 7-7. Clearly, the 100 average delays have a normal distribution, agreeing with the Central Limit Theorem [20]. Therefore, confidence intervals can be constructed about the sample mean of the 100 average delays. The data summary of the experiment is as follows:

Table 7-2. Model 02-a Job Description

Job Description	
Job type	PARAMETRIC JOB
Platform to Run Application	Windows
Number of Replications	100
Windows Application	ARENA
Windows Application Program	Model 02-a.doe
Machines Selected to Execute the Job	
n01 vmasc.odu.edu	n17 vmasc.odu.edu
n02 vmasc.odu.edu	n18 vmasc.odu.edu
n04 vmasc.odu.edu	n19 vmasc.odu.edu
n05 vmasc.odu.edu	n20 vmasc.odu.edu
n06 vmasc.odu.edu	n21 vmasc.odu.edu
n07 vmasc.odu.edu	n22 vmasc.odu.edu
n08 vmasc.odu.edu	n23 vmasc.odu.edu
n09 vmasc.odu.edu	n24 vmasc.odu.edu
n10 vmasc.odu.edu	n25 vmasc.odu.edu
n11 vmasc.odu.edu	n26 vmasc.odu.edu
n12 vmasc.odu.edu	n27 vmasc.odu.edu
n13 vmasc.odu.edu	n28 vmasc.odu.edu
n14 vmasc.odu.edu	n29 vmasc.odu.edu
n15 vmasc.odu.edu	n30 vmasc.odu.edu
n16 vmasc.odu.edu	n31 vmasc.odu.edu
Job Input Files:	
/home/jhead/Examples/vr_example/AV/Model 02-a/Model 02-a.doe	
/home/jhead/Examples/vr_example/AV/Model 02-a/seeds.txt	

Table 7-3. Model 02-b Job Description

Job Description	
Job type	VR-AV JOB
Platform to Run Application	Windows
Number of Paired Replications	100
Windows Application	ARENA
Windows Application Program	Model 02-b.doe
Machines Selected to Execute the Job	
n01 vmasc.odu.edu	n17 vmasc.odu.edu
n02 vmasc.odu.edu	n18 vmasc.odu.edu
n04 vmasc.odu.edu	n19 vmasc.odu.edu
n05 vmasc.odu.edu	n20 vmasc.odu.edu
n06 vmasc.odu.edu	n21 vmasc.odu.edu
n07 vmasc.odu.edu	n22 vmasc.odu.edu
n08 vmasc.odu.edu	n23 vmasc.odu.edu
n09 vmasc.odu.edu	n24 vmasc.odu.edu
n10 vmasc.odu.edu	n25 vmasc.odu.edu
n11 vmasc.odu.edu	n26 vmasc.odu.edu
n12 vmasc.odu.edu	n27 vmasc.odu.edu
n13 vmasc.odu.edu	n28 vmasc.odu.edu
n14 vmasc.odu.edu	n29 vmasc.odu.edu
n15 vmasc.odu.edu	n30 vmasc.odu.edu
n16 vmasc.odu.edu	n31 vmasc.odu.edu
Job Input Files	
/home/jhead/Examples/vr_example/AV/Model 02-a/Model 02-b.doe	
/home/jhead/Examples/vr_example/AV/Model 02-a/seeds.txt	

Number of Average Delays Experienced in Queue	= 100
Min Average Delays Experienced in Queue	= 1.44694
Max Average Delays Experienced in Queue	= 194.864
Sample Mean	= 102.656
Sample Variance	= 869.342
95% Confidence Interval	= [96.806, 108.506]
Half-width	= 5.84975

Thus, if 100 sets of experiments were conducted, it can be claimed with 95% confidence that the sample mean obtained in one of the experiments chosen at random will be contained within the interval [96.806, 108.506].

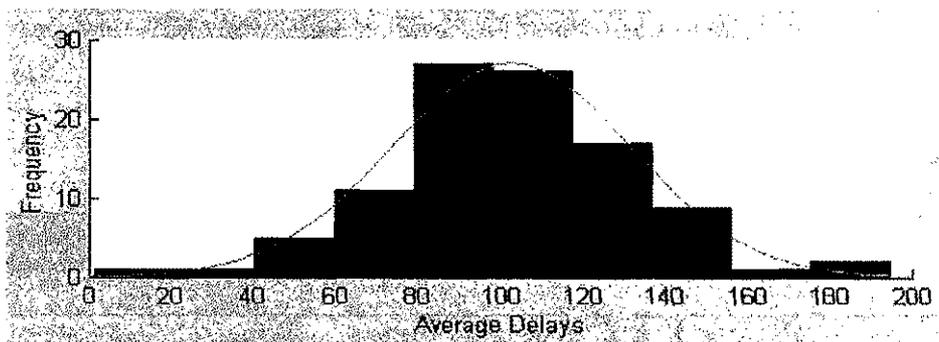


Figure 7-7. Model 02-a Average Delays

The average $X_j = \frac{X_j^1 + X_j^2}{2}$ of the average delays observed in each paired replication in the second experiment is computed. The 100 X_j 's are allocated into 10 equally spaced bins. The bin contents are plotted in the histogram in Figure 7-8. The data summary of X_j 's is as follows:

Number of Average Delays Experienced in Queue	= 200
Number of X_j 's	= 100
Min X_j	= 78.3182
Max X_j	= 124.857
Sample Mean	= 100.425
Sample Variance	= 101.762
95% Confidence Interval	= [98.4239, 102.427]
Half-width	= 2.00141

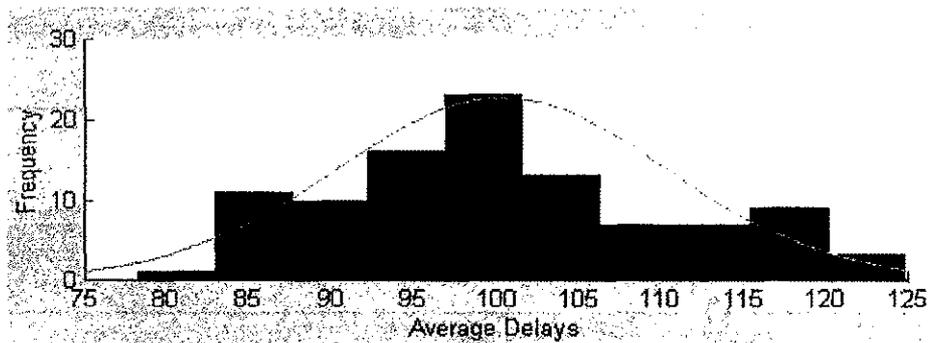


Figure 7-8. Model 02-b Average Delays

The sample variance of the average delays in the second experiment is much smaller than the sample variance of the average delays in the first experiment. The reduction in variance is about 88%. This confirms that applying AV VRT in Model 02-b reduces the variance of the average delay in queue. The reduction in the variance of the sample mean is also reflected in the 95% confidence interval. If one hundred experiments were conducted, it can now be claimed with the same 95% confidence that the sample mean obtained in an experiment chosen at random will be contained within the interval

[98.4239, 102.427]. The fact that the half-width of the 95% confidence interval in the second experiment is less than the half-width of the 95% confidence interval in the first experiment confirms greater precision in the sample mean obtained in the second experiment. That is, the sample mean obtained in the second experiment is closer to the true mean.

To further illustrate the variance reduction achieved under AV, the plots of the one hundred average delays observed in both experiments are displayed in Figure 7-9. The plots reveal that the average delays obtained from Model 02-a are more dispersed about the true mean than the average delays obtained from Model 02-b.

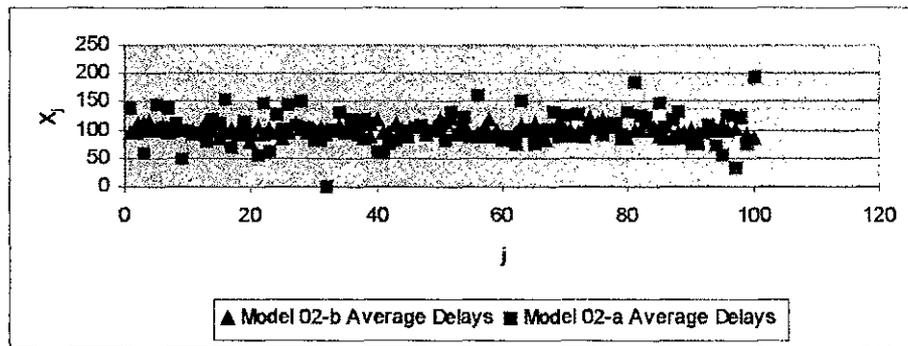


Figure 7-9. Comparison of Average Delays observed in Model 02 (a) and (b)

7.4. A Partitioned Case Study

The purpose of the case study described in this section is to demonstrate the speedups that are achievable by executing one replication of Model 03, shown in Figure 7-10, on sub-cluster Γ_i containing $m = 1, 2,$ and 3 processors. In the case study, little emphasis is placed on the structural properties of Model 03; the model is viewed from a very high level of abstraction. Therefore, each sub-model in Model 03 is viewed as a

black box. Model 03 models a production line that manufactures widgets. The widgets are manufactured from three parts, Part A, Part B, and Part C. The Create Parts sub-model creates the three parts and forwards data describing each part to the appropriate Process Part sub-model. Each Process Part sub-model processes the part and forwards it to the Merge Parts sub-model. The Merge Parts sub-model merges the parts when all three parts are available. The intricate internal workings of the sub-models are irrelevant to the case study. Instead, the partial simulation times of the sub-models are the important aspects in the case study. Therefore, with respect to Model 03 details, it is only assumed that the random numbers in the simulation are controlled using the proposed method in Section 3.4 to guarantee independence in the results.

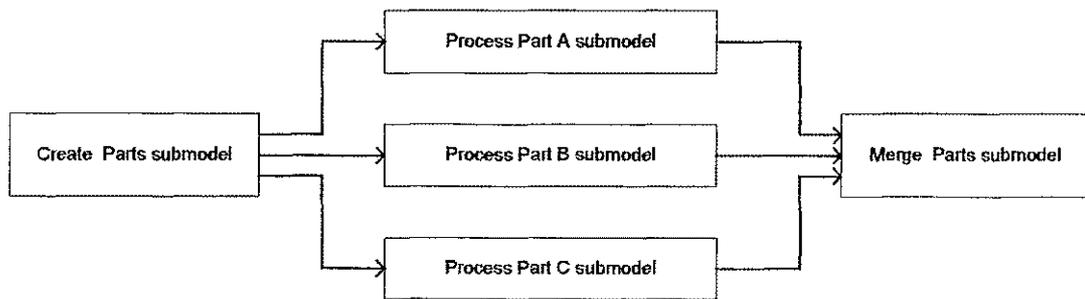


Figure 7-10. Model 03: Simulation Model of Production Line

Suppose Model 03 is setup so that the simulation is advanced in real time. Furthermore, suppose the simulation stopping condition is triggered by an internal event that is not time dependent so that the replication time of the model is random. For simplicity of the case study, assume that the partial times to simulate the sub-models in Model 03 are random, and that these partial times are uniformly distributed as follows. Process Part A partial simulation time is uniform on the interval [3, 5] minutes; Process Part B partial simulation time is uniform on the interval [1, 5] minutes; Process part C

partial simulation time is uniform on the interval [1, 6] minutes; finally, Create Parts and Merge Parts partial simulation times are both uniform on the interval [1, 3] minutes.

This level of abstraction makes sense if the general structure of simulation applications depicted in Figure 7-11 is considered. A typical application such as Arena presents the user with a graphical user-interface that allows the user to describe the model using graphical modules, objects, and blocks. The graphical representation of the model is translated into a high-level simulation language code, such as Siman [30] in the case of Arena, which is then compiled and linked with libraries to generate the model executable code. Therefore, the level of abstraction is justified if it is assumed that the sub-models in Model 03 are translated into high-level codes that can be executed in parallel in a dataflow strategy.

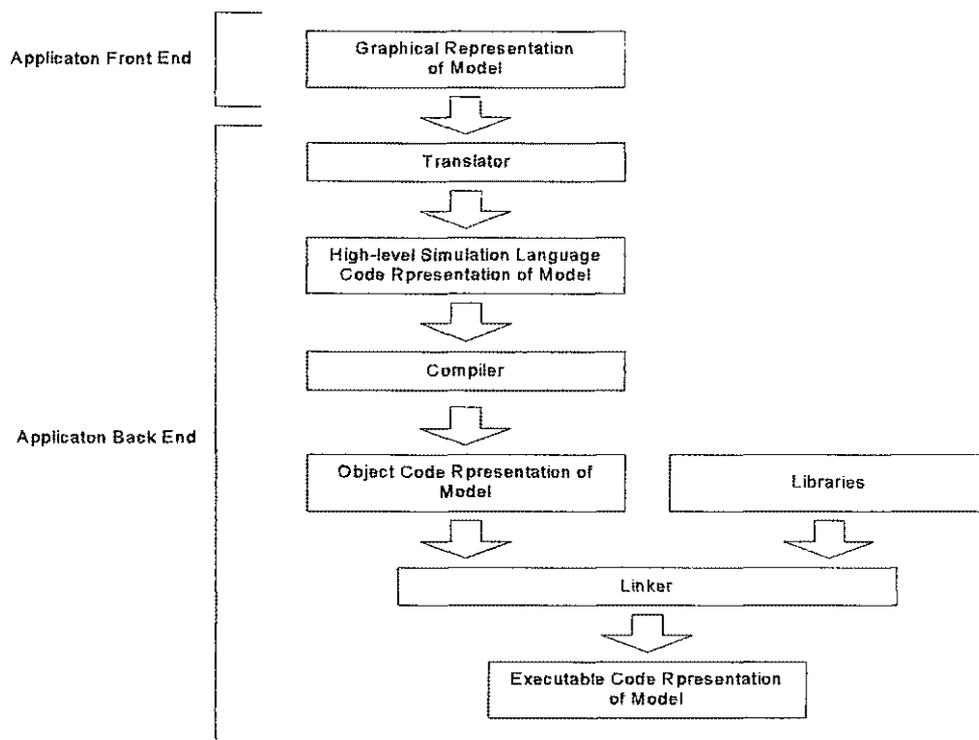


Figure 7-11. General Structure of Simulation Applications

Model 03 was carefully developed so that the computations of the model can be easily partitioned into tasks. The tasks are as follows: The computation of the Create sub-model is considered task T_0 ; the computations of the Process Part A, B, and C sub-models are considered tasks T_1 , T_2 , and T_3 , respectively; finally, the computation of the Merge sub-model is considered task T_4 . As a result, the precedence task graph representation, depicted in Figure 7-12, is very similar to the simulation model. It should be noted that this was done solely to simplify the case study, and that in general the computational model used to represent the computation units of the simulation model will be independent of the structure of the simulation model. To make the case study tractable, the mean computation times of the sub-models are used for the tasks weights.

Each replication of Model 03 is executed on sub-cluster $\Gamma_i \subset \Gamma$ defined in Section 5.1. The mean theoretical times to execute one replication of Model 03 on Γ_i 's containing one processor, two processors, and three processors are 14.5 minutes, 11.5 minutes, and 8 minutes, respectively. These are the times to execute the DAG in Figure 7-12, and are estimated using the Ghant Charts in Figure 7-13.

Three sets of experiment were conducted in which the number of processors, m_i , in Γ_i varied from 1 to 3. In the first experiment, each Γ_i had size $m = 1$, and the number of sub-clusters, n , in Γ varied from 1 to 10. For each value of n , Model 03 was replicated 10 times. The second and third experiments were similar to the first except that each Γ_i had size $m = 2$ in the second experiment, and $m = 3$ in the third experiment. In the experiments, the tasks were implemented in the ANSI-standard C programming language. The C codes that implement the tasks were very simple. Task T_0 and T_4 were both implemented with delays drawn from a uniform distribution on the interval [1, 3] minutes. Similarly, tasks T_1 , T_2 , and T_3 were implemented with delays drawn from uniform distributions on the intervals [3, 5] minutes, [1, 5] minutes, and [1, 6] minutes, respectively. The C implementation of the PMM Linear Congruential Generator (PMMLCG) in [4] was used to generate the random sequence U . The unique random streams U^1 , U^2 , U^3 , U^4 , and U^5 were used to generate the uniform delays in T_0 , T_1 , T_2 , T_3 , and T_5 , respectively.

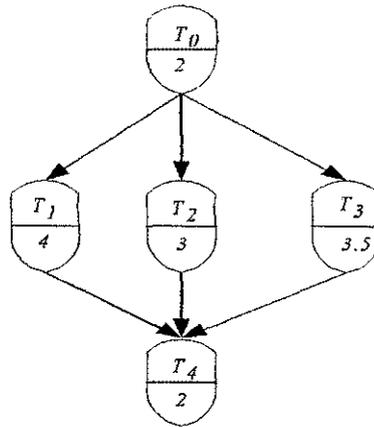


Figure 7-12. DAG Representation of Model 03

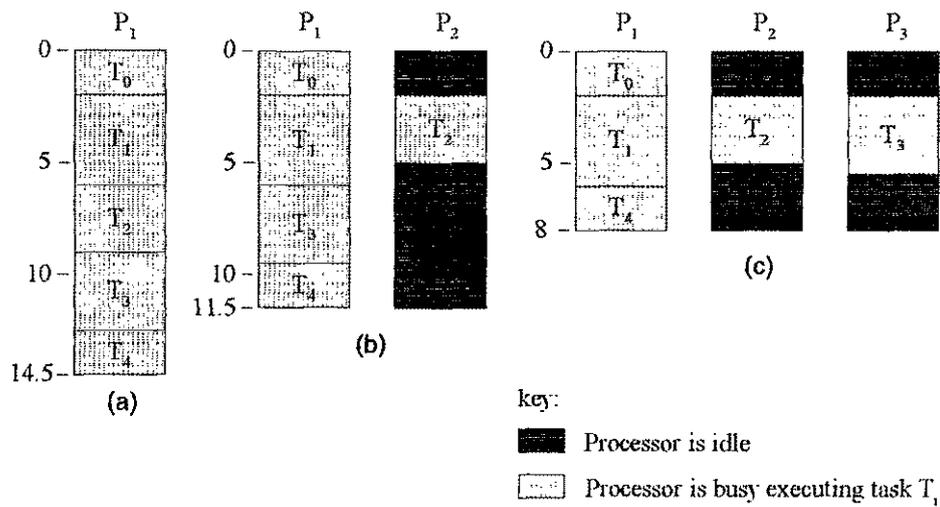


Figure 7-13. Mean Times to execute DAG on (a) One processor, (b) Two processors, (c) Three processors

7.4.1. Job Description

A sample of the job descriptions submitted to the XFace tool in the experiments is listed in Table 7-4. The items in the job descriptions are as follows. The total number of tasks in the partitioned program is five. The partitioned program will be replicated ten times on the native Linux platform running on the processors in Γ_i . The job execution command that executes the tasks executable files is T^* , where $*$ will be replaced with 0, 1, 2, and 4 by the XFace tool. The maximum number of processors that is allowed in Γ_i is two. The processor aliased `n01.vmasc.odu.edu` is the only processor submitted to execute the job. Since only one processor is submitted, only one sub-cluster is created. Finally, a set of job files is submitted for each of the five tasks in the partitioned program. The job files are the task executable, and the seeds file that will be used to initialize the random number streams in each task. The DAG representation of Model 03 is submitted as the text file listed in Figure C-1 in Appendix C.

For the three experiments, a total of 30 job descriptions similar to the one depicted in Table 7-3-1 but with varying number of slave processors N were submitted at different times to the XFace tool. In all three experiments, the number of sub-clusters n varied from 1 to 10. However, the number of processors N varied from 1 to 10 in the first experiment with $m = 1$, 2 to 20 in the second experiment with $m = 2$, and 3 to 30 in the third experiment with $m = 3$.

7.4.2. Results

The execution times obtained from the experiments are plotted against the number of sub-cluster (n), and are displayed in Figures 7-14, 7-15, and 7-16. The curves in each figure are the theoretical mean execution times versus n , and the actual execution times obtained in the experiments versus n . The theoretical mean execution times are computed using Equation (5-1) for $n = 1, 2, \dots, 10$, and the theoretical mean replication times $T_R(m)$ for $m = 1, 2$, and 3. The theoretical mean replication times are $T_R(1) = 14.5$ minutes, $T_R(2) = 11.5$ minutes, and $T_R(3) = 8$ minutes. These times are estimated by the

Ghant Charts in Figure 7-13. The curves in each figure confirm that the actual execution times closely match the theoretical execution times.

Table 7-4. Partitioned Case Study Job Description

Job Description	
Job Type	: PARTITIONED JOB
Number of Tasks	: 5
Platform to Run Application	: Linux
Number of Replications	: 10
Job Execution Command	: T*
Maximum Machines Per Subcluster	: 2
The machines are subclustered as follows:	
Subcluster 1: n01.vmasc.odu.edu	
Task 0 Job Files:	Task 3 Job Files:
/home/jhead/Examples/part_example/job/T0	/home/jhead/Examples/part_example/job/T3
/home/jhead/Examples/part_example/job/seeds0	/home/jhead/Examples/part_example/job/seeds3
Task 1 Job Files:	Task 4 Job Files:
/home/jhead/Examples/part_example/job/T1	/home/jhead/Examples/part_example/job/T4
/home/jhead/Examples/part_example/job/seeds1	/home/jhead/Examples/part_example/job/seeds4
Task 2 Job Files:	
/home/jhead/Examples/part_example/job/T2	
/home/jhead/Examples/part_example/job/seeds2	

For each n such that $k \% n \neq 0$, the number of idle processors increases as m increases from 1 to 3. For example, for $n = 3$, the replications are batched into four batches. However, two of the three sub-clusters are not used during the execution of the last batch. Therefore, during the execution of the last batch, $(2 \times 1) = 2$ processors are idle when $m = 1$, $(2 \times 2) = 4$ processors are idle when $m = 2$, and $(2 \times 3) = 6$ processors are idle when $m = 3$. For each replication, the mean utilization of each processor in a

busy sub-cluster is estimated from the Ghant Charts in Figure 7-13. For $m = 1$, the only processor P_1 is fully utilized. For $m = 2$, the utilizations of P_1 and P_2 are 100% and 26%, respectively. Finally, for $m = 3$, the utilizations of P_1 , P_2 , and P_3 are 100%, 37%, and 32%, respectively.

Figure 7-17 illustrates a comparison of the actual curves for $m = 1, 2$, and 3. For each n , the execution time for $m = 3$ is the smallest and the execution time for $m = 1$ is the largest. These observations reveal that the time to execute one replication of Model 03 decreases as the size of the sub-clusters increases from 1 to 3. For each n , the speedups per replication $\tau_{2D}(k, n, m)$ were computed using Equation (5-3) for $k = 10$ and $m = 2, 3$. The plots of the speedups per replication versus n are exhibited in Figure 7-18. The average speedup per replication for $m = 2$ with respect to $m = 1$ is 1.25, and the average speedup per replication for $m = 3$ with respect to $m = 1$ is 1.68. Therefore, on the average, each replication in the second experiment was executed 1.25 times faster than its counterpart in the first experiment, and each replication in the third experiment was executed 1.68 times faster than its counterpart in the first experiment. These speedups appear small at first glance. However, the total speedup in the second experiment is $1.25 \times \tau_{1D}(k, n)$, and $1.61 \times \tau_{1D}(k, n)$, where $\tau_{1D}(k, n)$ is the speedup resulted from using n sub-clusters to execute the k replications.

The total speedups, $\tau(k, n, m)$, in the execution times were computed using Equation (5-5). Figure 7-19 displays the total speedups in the execution times for the three cases $m = 1, 2, 3$. The general shape of the curve in the figure is the same as the same as the shape of the speedup curve in Figure 7-5. As expected, the speed-up curve for $m = 3$ is shifted further along the positive speedup axis than the speedup curve for $m = 2$, which is in turn shifted further along the positive speedup axis than the speedup curve for $m = 1$. The general shape of the curves results from the speedup in the execution times in the first dimension, $\tau_{1D}(k, n)$, and the displacement in the three curves results from the speedup in the execution times in the second dimension, $\tau_{2D}(k, n, m)$.

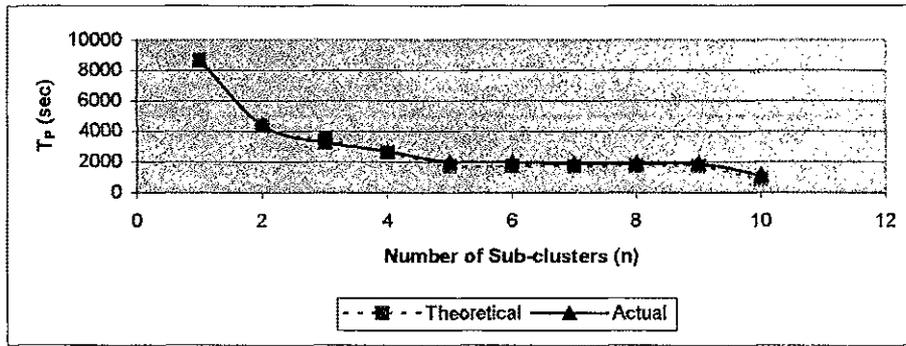


Figure 7-14. Execution Times versus Number of Sub-clusters for $m = 1$

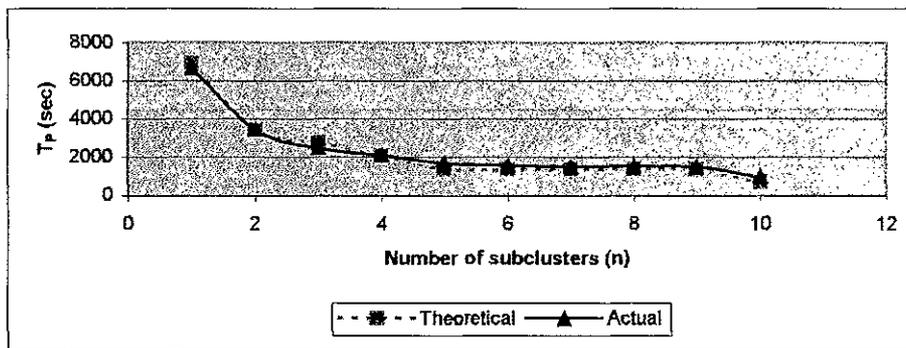


Figure 7-15. Execution Times versus Number of Sub-clusters for $m = 2$

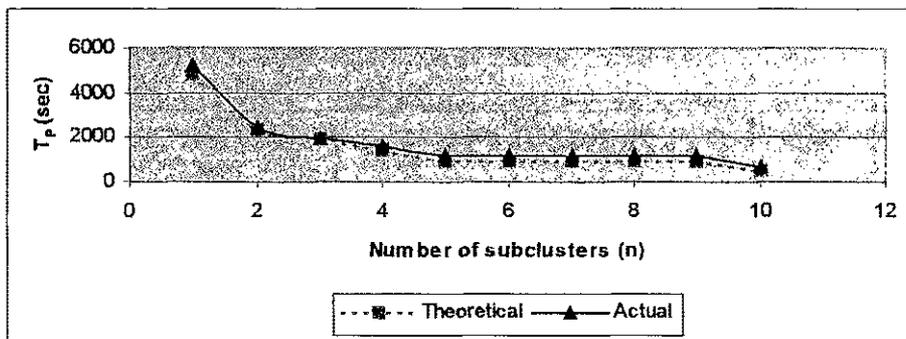


Figure 7-16. Execution Times versus Number of Sub-clusters for $m = 3$

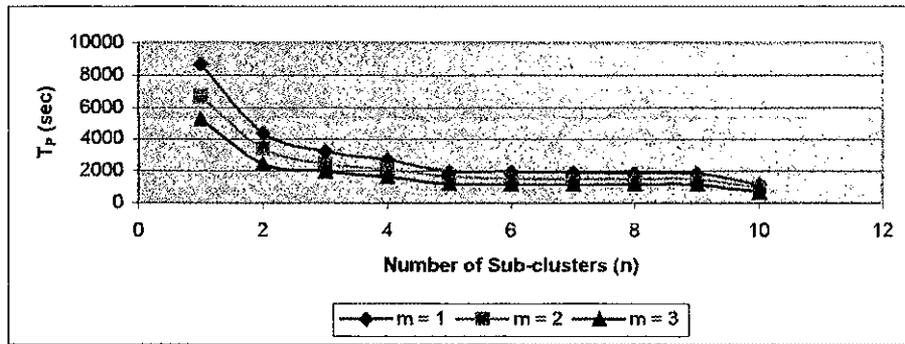


Figure 7-17. Comparison of Actual Curves for $m = 1, 2, 3$

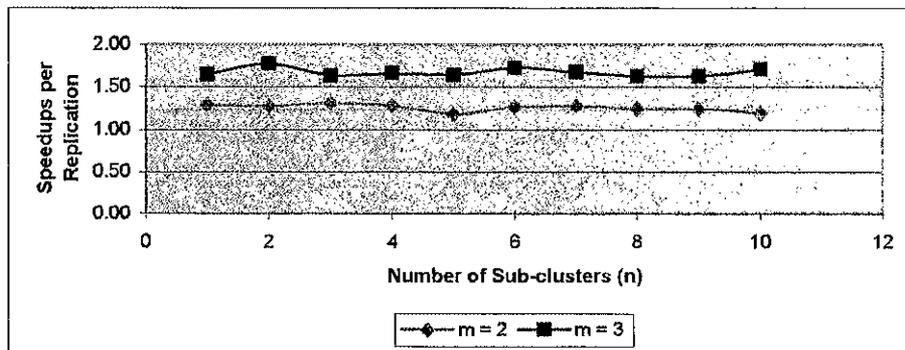


Figure 7-18. Speedups per Replication versus the Number of Sub-clusters

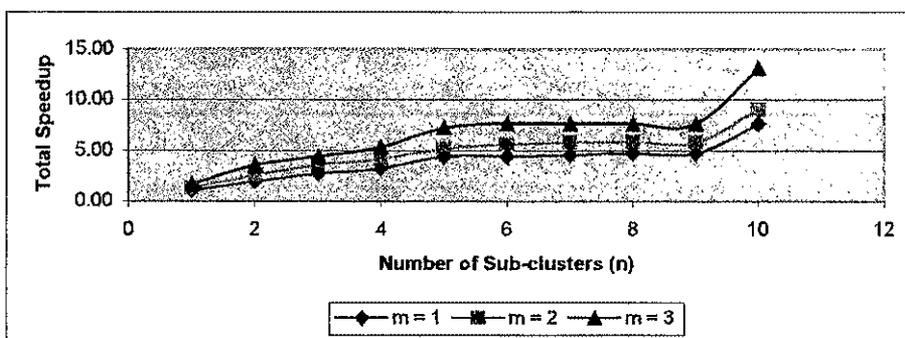


Figure 7-19. Comparison of Total Speedups for $m = 1, 2, 3$

CHAPTER VIII

CONCLUDING REMARKS

The three case studies described in Chapter 7 illustrate the capabilities of the XFace tool. The parametric case study reveals that the use of ten processors to execute ten replications of the sample program resulted in a speedup of over eight times the sequential execution time. The speedup is not 10-fold due to overhead. The dominant contributors of the overhead are the times to startup and execute the application on the emulated platform. In the environment depicted in Figure 3-3, the automation control program must be restarted at the beginning of every iteration that is executed on processor P_j . However, it is not necessary to restart the application program in every iteration. At the beginning of the first iteration that is executed on processor P_j , the automation control must start an instance of the application since one will not exist. However, in the subsequent iterations that are executed on P_j , the automation controller should use the running instance of the application to avoid restarting it at the beginning of every iteration. Additionally, to decrease the overhead associated with executing the automation program, the automation program should be a lightweight process. That is, it should not do more than is required of it. The less it has to do the faster it will run. To illustrate the overhead involved with starting up an application on the emulator, a small experiment was conducted using Arena. In this experiment, a simple automation program was developed. The automation program simply started an instance of Arena on the emulated platform, then immediately closed the Arena instance. Therefore, the execution time of the automation program approximates the time to start an instance of the Arena application on the emulated platform. The execution of the program was timed, and 55 execution times were taken. The average of the execution times is 8.20 seconds. If the execution times of the replications are on the order of a couple seconds, an application startup time of 8.20 seconds will have a severe impact on the replication time. Thus, the speedup achieved, if any, will be very small. This is why it was stated clearly in the opening section of Chapter 1 that the tool works for simulation that takes a long time to complete. The impact of this relatively large startup time on the total execution time of the simulation can be lessened if the application is started once, only in the first iteration

that is executed on processor P_j . If it is assumed that at the beginning of the job execution phase, an iteration is started on all N processors simultaneously, $N \leq k$, then all replications in the first batch will experience the startup time concurrently. As a result, the overall effect on the total execution time is as if the application had started once. Thus, provided that the automation program is a lightweight process, the overhead to run the application on the emulated platform is greatly reduced if the application is not restarted in every iteration.

The case studies reveal that speedups are achievable for all $N > 1$. However, the processors are only fully utilized for some values of N . In particular, for given k and N , the processors are best utilized when $N \leq k$ and N divides k evenly. For these cases, all processors will be utilized during the execution of every batch of replications. However, the utilization will not be 100% in all cases. If the replication lengths are fixed, then all replication should take approximately the same time to complete. Thus, if all replications within a batch are started simultaneously, the processors will be almost 100% utilized. However, the stopping condition of the simulation could be triggered by an event in the simulation that is not time dependent, such as stopping the simulation after 100 customers have completed service. In the latter case, the replication lengths are samples from a random process, because they are mappings of the random input processes driving the simulation. As a result, during the execution of a batch, some replication will take longer than some. Therefore, some processors will be idle while the others are busy completing the longer replications in the batch. To counter this problem, the scheduler schedules a replication when a processor becomes available. With this heuristic, the replications are executed in batches of random sizes if the replication length is random. However, the executions of the batches overlap. Therefore, under this scheduling heuristic, all processors are kept busy as long as there are replications remaining to be executed. Processors are only under utilized when the final batch of replication is executing.

For a given program that is represented by the DAG model, the speedup per replication depends on the structure of the DAG, and the number of processors m used to execute each replication, assuming that the DAG model reflects the most efficient way to

partition the program for the architecture at hand. Therefore, in some programs, the speedup per replication will be significant, while in others it will not be very significant. However, it should be remembered that the total speedup is now the product $\tau_{1D}\tau_{2D}$. Therefore, the significance of τ_{2D} depends on τ_{1D} . If τ_{1D} is large, then even a small τ_{2D} will result in a large total speed-up. For the handcrafted example used in the third case study, τ_{2D} of 1.67 was achieved using three processors to execute each replication, which is not very significant. However, because τ_{1D} was fairly large, the resultant speed up was also huge.

8.1. Future work

This section highlights the future work that could be done on the XFace tool. Currently, support for a Windows-based application is added to the XFace tool by editing the start-win-app script, and entering the execution command that starts the automation program. In the future, support for a Windows-based application could be added from the front-end GUI, thus making the start-win-app script transparent to the user. This could be achieved by adding a backend XFace process, *script editor*, that edits the script, and a dialog box, *Add New Application Support*. The Add New Application Support dialog could be launched from the click of a button or menu item in the tool's front-end GUI. The Add New Application dialog will prompt the user for the information to be entered in the script. Upon closing the dialog, the script editor is started and passed the information entered via shared-file. The script editor edits the script and then exits execution.

The implementation of the task-scheduling algorithm is such that busy processors are interrupted during task execution to forward data to remote processors that are preparing to execute dependent tasks. In the future, a new implementation of the task-scheduling algorithm could be realized. In this implementation, one processor in a slave cluster-node could be dedicated for communicating dependent data and the others dedicated for executing the tasks. The setup on a slave cluster-node is illustrated in Figure 8-1 for the case when each cluster-node contains 2 processors. As described in Section 2.2, the MPI daemons enable communication among remote processes. The MPI

of task T_j execution. This could reduce the execution time of tasks that are executed on each slave processor.

For the case when each cluster-node contains p processors, either of the designs in Figures 8-2 and 8-3 could be implemented on each slave cluster-node. In the diagram depicted in Figure 8-2, tasks executed on the local processors write dependent data to a common shared-memory. All local executing tasks contend for the shared-memory. As result, the task execution times will be increased when the shared memory is highly contended. The diagram depicted in Figure 8-3 solves the shared memory contention problem by allocating dedicated shared memories to the tasks. However, the implementation of this design will require many more shared memory locations than the implementation of that in Figure 8-2. Therefore, the trade-off involved here is that more shared memory locations must be utilized to counter the share memory contention problem. Thus, if memory is available in abundance, the design in Figure 8-3 is chosen over that in Figure 8-2.

The task scheduler should be prevented from scheduling tasks to processors that are dedicated for data forwarding. This could be implemented by excluding all the processors that are dedicated for data forwarding from the set the Π . Consequently, the sub-clusters formed from the processors in Π will not contain processors dedicated for data forwarding.

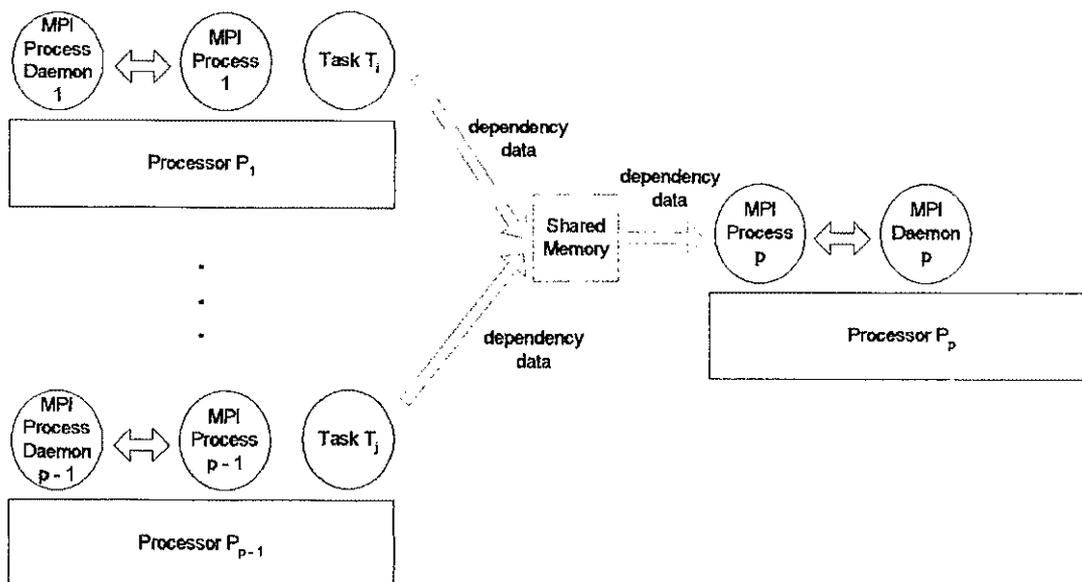


Figure 8-2. Future Setup 1 on Slave Cluster-node

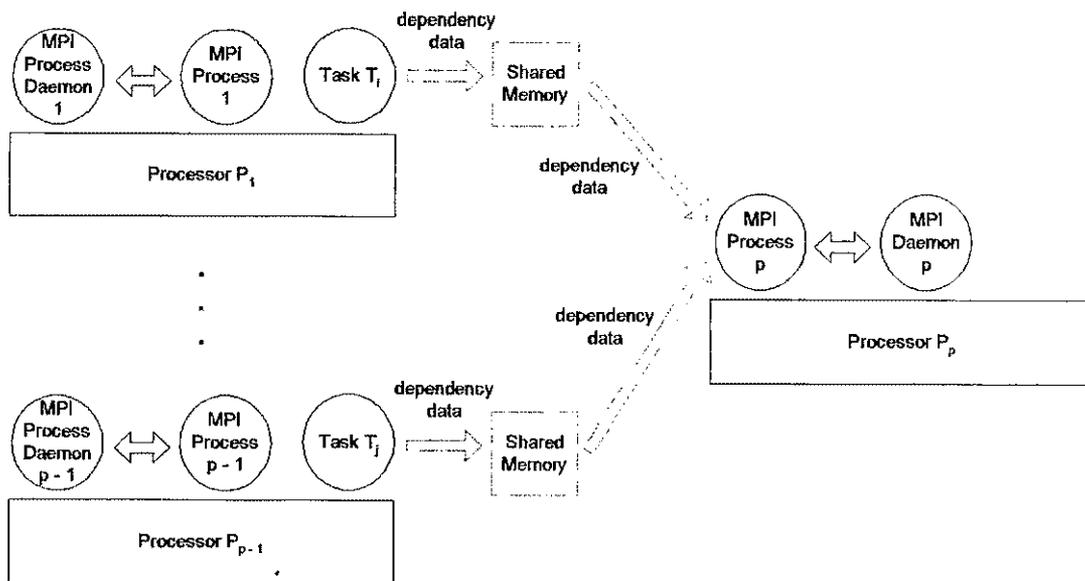


Figure 8-3. Future Setup 2 on Slave Cluster-node

REFERENCE

- [1] Kelton, W.D., Sadowski, R.P., and Sadowski, D.A. "Simulation with Arena," *McGraw-Hill*, 1998.
- [2] Kavi, K.M., and Shirazi, B. "Dataflow architecture: Are dataflow computers commercially viable?," *IEEE Potentials*, October 1992, pp. 27–30.
- [3] Bratley, P., Fox B. L., and Schrage Linus E. "A Guide to Simulation, 2nd ed.," *Springer*, 1987.
- [4] Law, A.M., and Kelton, W.D. "Simulation Modeling And Analysis, 3rd ed.," *McGraw-Hill*, 2000.
- [5] Nelson, B.L. "Robust Comparisons Under Common Random Numbers," *ACM Transactions on Modeling and Computer Simulation*, July 1993, vol. 3, no. 3, pp. 225-243.
- [6] Becker, J.D., Sterling, T., Savarese, D., Dorband E J., Ranawak, A.U., and Packer, V.C. "Beowulf: A Parallel Workstation for Scientific Computation," *Proceedings of International Conference on Parallel Processing*, 1995.
- [7] Ridge, D., Becker, D., Merkey, P., and Sterling, T. "Bewulf: Harnessing the Power of Parallelism in a Pile-of-PCs," *IEEE Proceedings on Aerospace Conference*, 1997, vol.2, pp. 79-91.
- [8] Castagnera, K., Cheng, D., Fatoohi, R., Hook, E., Kramer, B., Manning, C., Musch, J., Niggley, C., Saphir, W., Sheppard, D., Smith, M., Stockdale, I., Welch, S., Williams, R., and Yip, D. "Clustered Workstations and their Potential Role as High Speed Compute Processors," *NAS Computational Services Technical Report RNS-94-003*, NAS Systems Division, NASA Ames Research Center, April 1994.

- [9] Norton, C.D., and Cwik, T.A. "Early experiences with the myricom 2000 switch on an SMP Beowulf-class cluster for unstructured adaptive meshing," *IEEE International Conference on Cluster Computing*, Oct. 2001 Newport Beach, California, U.S.A. Proceedings, pp. 7-14.
- [10] Han G., Klenke, R.H., and Aylor, J.H. "Performance modeling of hierarchical crossbar-based multicomputer systems," *IEEE Transactions on Computers*, Sept 2001, vol. 50, no. 8. pp. 877 –890.
- [11] Woodward, T.K., Lentine, A.L., Fields, J.D., Giaretta, G., and Limacher, R. "First demonstration of native Ethernet optical transport system prototype at 10 Gb/s based on multiplexing of gigabit Ethernet signals," *IEEE Photonics Technology Letters*, Aug 2000, vol. 12, no. 8, pp. 1100 –1102.
- [12] Do-Yeon, K., Sang-Min, L., Chang-Ho, C., Hae-Won J., Yeong-Seon, K. "Trends of 10 gigabit Ethernet switch development in Korea," *IEEE Pacific Rim Conference on Communications, Computers and signal Processing, (PACRIM), 2003*, vol. 02, pp. 1032 – 1035.
- [13] "Virtual Interface Architecture Specification. Version 1.0," Compag, Intel and Microsoft Corporations, Dec 1997, available at <http://www.via.org>.
- [14] Macks, A. "Heterogeny in a Beowulf," *Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications (HPCS'02)*, 2002.
- [15] "Document for Standard Message-Passing Interface," Message Passing Interface Forum, May 28, 1993.

- [16] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. "A High-Performance, Portable Implementation of the MPI Message-Passing Interface Standard," *Parallel Computing*, 1996, vol. 22, no. 6, pp. 789-828.
- [17] Beguelin, A., Dongarra, J., Geist, A., Manchek, R., and Sunderam, V. "Visualization and debugging in a heterogeneous environment," *IEEE Computer*, June 1993, vol. 26, no. 6, pp. 88-95.
- [18] Dongarra, J., Geist, A., Manchek, R., and Sunderam, V. "Integrated PVM framework supports heterogeneous network computing," *Computers in Physics*, April 1993, vol. 7, no. 2, pp. 166-75.
- [19] Gropp, W., and Lusk, E. "A User's guide for MPICH, a portable implementation of MPI," ANL-96/6, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.
- [20] Papoulis, A., and Pillai, S. U. "Probability, Random Variables and Stochastic Processes, 4th ed.," *McGraw-Hill*, 2002.
- [21] Crane, M.A., and Iglehart, D.L. "Simulating Stable Stochastic Systems, I: General Multi-Server Queues," *Journal of the Association for Computing Machinery* Vol. 21, No. 1, Jan 1974, pp. 103 -13.
- [22] Sarkar, V. "Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors," PhD Thesis, Stanford University, April 1987. CSL-TR-87-328.
- [23] Mazumdar, S., Mathew R., and Leathrum, F.L., "A Strategy for Distributing Simulations for Statistical Analysis," *Proceedings of the Summer Computer Simulation Conference (SCSC 2004)*, July 2004, San Jose, CA, USA, pp. 151-156.

- [24] Kwok Y., and Ahmad, I. "Static Scheduling Algorithms for Allocating Directed Task Graphs o Multiprocessors," *ACM Computing Surveys* vol 31, no. 4, Dec 1999.
- [25] Rusling, D. A. "The Linux Kernel Version 0.8-3," available at <http://www.lldp.org/LDP/llk/llk-title.html>.
- [26] Mitchell, M., Oldham, J., and Samuel, A. "Advanced Linux Programming," *New Riders Publishing*, June 2001.
- [27] American National Standards Institute, New York, NY. American National Standard for Information Systems Programming Language-C, ANSI X3.159-1989, 1990.
- [28] Getz, K., and Gilbert, M., "VBA Developer's Handbook," *SYBEX Inc.*, 1997
- [29] "How to use and configure Wine for running Windows applications," Wine User Guide, available at <http://www.winehq.org/>
- [30] Banks, J., Bumette, B., and Rose, R. "Introduction to Siman V and Cinema V," *John Wiley & Sons, Inc.*, 1994

APPENDIX A

APPLICATION START-UP SCRIPTS

```

#!/bin/sh
#####
# Author       : Jermaine Headley
# Project      : Thesis Research
# Implementation Date : April 6, 2004
#
# start-win-app
# This script is used to start the Windows Application specified in $6, and run the Windows Application
# program specified in $5. The script creates the run directory <run*> in the </tmp/XFace> directory,
# where * is the iteration number specified in the command line argument $1. The application program is
# run in the </tmp/XFace/run*> directory so that the application program writes output files in the directory
# </tmp/XFace/run*>.
#
# Command Line arguments:
# $0 = Name of this Script "start-win-app"
# $1 = job run number
# $2 = configuration number
# $3 = the job type
# $4 = name of slave node running this script
# $5 = pathname to the Windows Application Program (model file)
# $6 = Windows Application name in all upper-case
#
#####
PATHNAME="/home/jhead/programs/XFace"
XFACE_DIR="/tmp/XFace"
STREAMSDIR="/tmp/XFace/STREAMS/"
WINE="/opt/cxoffice/bin/wine"
master="balrog.vmasc.odu.edu"

PARAMETRIC=0
PARTITIONED=2
VR_AV=3
VR_CRN=4

# Define Windows Application Name Constants
ARENA="ARENA"

# Command Line Arguments #
node="$4"
model="$5"
win_app="$6"

#
# -----
# Generate the name of the run directory, and the name of the seeds file that is scheduled with the current
# iteration. For AV-CRN jobs, the iteration directory is <"run" + configuration number + iteration
# number>. Similarly, the seeds file is <"seeds + configuration number + iteration number>. For all other
# jobs, the run directory is <"run" + job run number> and the streams file is <"run" + job run number>.
#
# Note: The blocks of codes within this section will not get executed 99% of the time this script is
# invoked because the run directories were already created when the Job Loader script was executed after

```

```
# the job submission. However, in the rear event one of the run directory gets deleted or was not created
# we would like to catch this error because it has the potential of crashing the XFace Application
#
```

```
if test $3 = $VR_CRN
then
JOBDIR="{XFACE_DIR}/JOB/config$2"
RUNDIR="{XFACE_DIR}/run$2$1"

#
# if run directory does not exist create it and copy the seeds file for this run
#
if test ! -d $RUN_DIR
then
echo "creating subdirectory ${RUNDIR}"
mkdir "${RUNDIR}"
cp "$JOBDIR/*" "${RUNDIR}"

seeds_file="{STREAMSDIR}streams${2}${1}"
if test -e $seeds_file
then
cp $seeds_file "$RUNDIR/seeds1.txt"
fi
fi
fi
```

```
if test $3 = $VR_AV
then
JOBDIR="{XFACE_DIR}/JOB"
RUNDIR="{XFACE_DIR}/run$1"

#
# if the run directory does not exist create it, copy the seeds files for this run
# the application program to the run directory
#
if test ! -d $RUN_DIR
then
mkdir $RUNDIR
cp "$JOBDIR/*" $RUNDIR

seeds_file1="{STREAMSDIR}streams$1"
seeds_file2="{STREAMSDIR}streams{'expr $1 + 1'}"

if test -e $seeds_file1
then
cp $seeds_file1 "$RUNDIR/seeds1.txt"
fi

if test -e $seeds_file2
then
cp $seeds_file2 "$RUNDIR/seeds2.txt"
fi
fi
fi
```

```
if test $3 = $PARAMETRIC
```

```

then
JOBDIR="{XFACE_DIR}/JOB"
RUNDIR="{XFACE_DIR}/run$1"

#
# if the run directory does not exist create it, copy the seeds file for this run
# the application program to the run directory
#
if test ! -d $RUN_DIR
then
mkdir $RUNDIR
cp $JOBDIR/* $RUNDIR

seeds_file="{STREAMSDIR}streams$1"
if test -e $seeds_file
then
cp $seeds_file "$RUNDIR/seeds1.txt"
fi
fi
fi
#
-----
#
# An application that displays a graphical user-interface needs a display. Therefore, set the X DISPLAY
# variable so that if the Windows Application requires a display to run one is available. Note that this
# display is only available if the X server is running on $node under the user login.
#
export DISPLAY=$node:0.0
#
-----
#
# Change to the run directory specified by $RUNDIR so that the results of the replications are saved in the
# this directory.
#
cd $RUNDIR

#*****#
#
#          ADD CODE SPECIFIC TO EACH WINDOWS APPLICATION BELOW
#
#*****#

#
#-----
#
#          CODE TO EXECUTE ARENA APPLICATION PROGRAMS
#
# This section of code runs only Arena Version 5.03 application programs.
# The name of the automation controller is a VBA application called "run_arena_model.exe." It is stored
# in the directory <~/XFace/windows>. The run_arena_model.exe program replicates the Arena program
# specified in the file model.txt x times. The Arena program and the number of replications x are specified
# on the first and second lines in model.txt, respectively.
#
# The windows emulator used to run the Arena application and the automation program is a commercial

```

```

# implementation of the "WINE" tool by CodeWeavers. The CodeWeavers Professional 3.0 software
# must be installed at the location </opt/exoffice/bin/wine>.
#
if test $win_app = $ARENA
then

    echo "Running run $1 of the Arena Application Program $model on $node....."

    #
    # The file model.txt is the interface between the XFace tool and the automation controller. It contains the
    # name of the Arena model and the number of replications x that the model must be replicated. Thus, it
    # must be copied from the job directory <$JOBDIR> to the run directory <$RUNDIR> so that the
    # automation controller that automates the Arena application can know the name of the Arena model to
    # open when it starts the Arena application, and the number of times the Arena model must be
    # replicated.
    #
    cp "$JOBDIR/model.txt" model.txt

    #
    # Run x replications of the Arena application program
    #
    ${WINE} ${PATHNAME}/windows/run_arena_model_then_quit.exe

    echo "Run $1 of the Arena Application Program $model on $node completed....."

    #
    # Remove files no longer needed from RUNDIR
    #
    rm -f "${RUNDIR}/model.txt" *.Backup.doe *.opw *.p
fi
#

```

```

exit 0

```

```

#!/bin/sh
#####
# Author      : Jermaine Headley
# Project     : Thesis Research
# Implementation Date : April 6, 2004
#
# start-up-part
# This script is used to start the partitioned application specified in the command line arguments $@.
# The script creates the run directory <run*> in the </tmp/XFace> directory, where * is the iteration
# number specified in the second command line argument. The application is run in the </tmp/XFace/run*>
# directory so that the output files are written to the directory </tmp/XFace/run*>.
#
# Command Line Arguments:
# $0  = Name of this script "start-up-part"
# $1  = job run number
# $2  = task index
# $3  = processor name of the processor on which this script runs

```

```

# $4 - $9 = Command to execute the task
#####
XFACEDIR="/tmp/XFace/"
STREAMSDIR="/tmp/XFace/STREAMS/"
data_files="/tmp/XFace/JOB/data_files"

#
# Parse command line arguments to the array argv
#
i=0
for argument in "$@"
do
  argv[$i]=$argument
  i=`expr $i + 1`
done

argv_len=`expr $i - 1`
job_run="{argv[0]}"
task_index="{argv[1]}"
local_machine="{argv[2]}"
app="{argv[3]}"
unset argv[0]
unset argv[1]
unset argv[2]
unset argv[3]

#
# Create the run directory if it doesn't already exist
#
RUNDIR="{XFACEDIR}run$job_run"
if test ! -d $RUNDIR
then
  echo "creating run directory $RUNDIR....."
  mkdir $RUNDIR

  #
  # Copy the seeds file for the current iteration and the current task
  #
  seeds_file="{STREAMSDIR}streams${job_run}${task_index}"
  cp $seeds_file "$RUNDIR/seeds${task_index}.txt"
fi

#
# Copy the dependency data from the processors on which the parent tasks were run,
# which are specified by the data files names and locations in data_files
#
echo " "
echo "..... start-up-part running on $local_machine....."
echo "Receives task $task_index of job run $job_run....."
OLD_IFS=$IFS
IFS=:
i=1
file=
remote_machine=
for str in `cat $data_files`
do

```

```

if test "$str" != "$local_machine" -a $i = 2
then
  remote_machine="$str"
  echo "copying ${file} from ${remote_machine}:${RUNDIR} to ${local_machine}:${RUNDIR}....."
  rcp "${remote_machine}:${RUNDIR}/${file}" "$RUNDIR"
  i=1
else
  file="$str"
  i=`expr $i + 1`
fi
done
IFS=$OLD_IFS

#
# Execute the tsak if it is a valid executable file
#
if test -x "${app}"
then
  echo "executing the command ${app} ${argv[@]}...."
  cd "${RUNDIR}"
  "${app}" "${argv[@]}"
  exit 0
else
  echo "${app} is not an executable file....."
  exit 1
fi

```

APPENDIX B

AN IMPLEMENTATION OF ARENA MODELLOGIC_RUNBEGIN EVENT HANDLER

```
Option Explicit
Dim SeedsModule As Arena.Module
```

```
#####
'# The function SearchSubModel() recursively searches oSubmodel for a SEEDS element with
'# tag = "seeds". If the SEEDS element is found, the global variable SeedsModule is assigned to the SEEDS
'# element found and the function returns True. Otherwise, the function returns False.
#####
Private Function SearchSubModel(oSubmodel As Arena.submodel) As Boolean
Dim index As Long
Dim s As Arena.submodel

index = 0

'base case - the SEEDS element is in the modules collection of oSubmodel
index = oSubmodel.Model.Modules.Find(smFindTag, "seeds")

If index <> 0 Then
Set SeedsModule = oSubmodel.Model.Modules.Item(index)
SearchSubModel = True
Exit Function
End If

'Recursive case - oSubmodel contains at least one submodel and the SEEDS element
'was not found. Recursively search each submodel in oSubmodel.
For Each s In oSubmodel.Model.Submodels

Exit if SEEDS element was found in s
If SearchSubModel(s) = True Then
SearchSubModel = True
Exit Function
End If
Next

'SEEDS element was not found in oSubmodel so return false
SearchSubModel = False
End Function
```

```
#####
'# The function SetSeedsValues () set the seeds value for each stream defined in the SEEDS element
'# defined in the model. If a SEEDS element, with tag = "seeds", is defined in the model, this function
'# searches for the SEEDS element. If found, it sets the seed values of the streams defined in the SEEDS
'# element with values read from the file "seeds*" stored in the working directory, where * is the replication
'# number. When the model is replicated, the streams are initialized to the new seed values set in the
'# SEEDS element.
'#
'# The entries in the seeds* files are read in as Strings. Reading in the entries as String allocates enough
```

```

'# memory to store very large number, which is very critical for reading in very large values from the
'# seeds* files. The values in the seeds file are separated by commas.
'#
'# Note: For this implementation to be of any use, the streams from which the model gets random numbers
'# must not only be defined in the SEEDS element, but must also be hard coded in the model. Thus, for
'# example, if stream 3 is defined in the SEEDS element, and stream 3 is used to provide random numbers
'# for a uniform distribution on the interval [10 50], stream 3 may be hard coded in an expression
'# as UNIF(10,50,3).
#####
Private Sub SetSeedsValues(currentReplication As Long)
Dim oModules As Arena.Modules
Dim oSubModels As Arena.Submodels
Dim s As Arena.submodel
Dim found As Boolean
Dim index As Long
Dim FileNum As Integer

found = False
index = 0

'First, search the modules collection of ThisDocument.Model for the SEEDS element
Set oModules = ThisDocument.Model.Modules
index = oModules.Find(smFindTag, "seeds")

If index <> 0 Then
Set SeedsModule = oModules.Item(index)
found = True
End If

'If the SEEDS element was not found in the modules collection and ThisDocument.Model
'contains at least one submodel, then each submodel is searched recursively for
'the SEEDS element
If found = False Then
Set oSubModels = ThisDocument.Model.Submodels
If oSubModels.Count <> 0 Then
For Each s In oSubModels
If SearchSubModel(s) Then
found = True
Exit For
End If
Next
End If
End If

'The Search is over. If the SEEDS element was found, the streams that will
'be used for this replication are specified in the file seeds*
If found Then
Dim seedValue As String
Dim operandName, seedsk As String
Dim Char

'Open seeds* file. Exit if the file was not opened successfully, or if the
'file was not found
seedsk = "seeds" & currentReplication & ".txt"

FileNum = FreeFile

```

```

Open seedsk For Input As #FileNum
If FileNum = 0 Then
  Exit Sub
End If

'Read in the seeds values while not EOF. The seeds values are separated
'by a commas '
index = 1
Do Until EOF(FileNum)
  Char = Input(1, #FileNum)

  If Char = "," Then
    operandName = "Seed(" & index & ")"
    SeedsModule.Data(operandName) = seedValue
    index = index + 1
    seedValue = ""
  ElseIf Char <> " " Then
    seedValue = seedValue + Char
  End If
Loop

'If StrComp(seedValue, " ") <> 0 Then
' operandName = "Seed(" & index & ")"
' SeedsModule.Data(operandName) = seedValue
'End If

Close #FileNum
End If

End Sub

#####
'# This function is an implantation of Arena ModelLogic_RunBegin() event handler. It is invoked once
'# before Arena checks the model.
#####
Private Sub ModelLogic_RunBegin()

-----
:
:           ADD YOUR CODE HERE
:
-----

SetSeedsValues (1)

End Sub

```

APPENDIX C

PARTITIONED CASE STUDY TASK GRAPH

```
#####
# Sample Task Graph
#
# The format of the task graph is as follows. Lines that are comments are started with the pound '#'
# character. The first non-comment line must contain the number of task in the graph. The next non-
# comment lines contain information for the task. The format for each task Ti is as follows:
#
#   task_index task_weight number_of_parents number_of_children
#   [<parent, data file> ... <parent, data file>] NULL
#   [<child, edge weight> ... <child, edge weight>] NULL
#
# 1) The first entry specifies the task index of Ti for which the information is being entered. This entry
#    must be an integer.
#
# 2) The second entry specifies the task weight of Ti. This is the computational cost incurred when Ti is
#    executed. It must be a double value.
#
# 3) The next two entries specify the number of tasks that parent Ti and the number of tasks that are
#    children of Ti, respectively. These entries must be integers.
#
# 3) The third set of entries are optional. If task Ti has no parent, the next entries are ignored until the string
#    "NULL" is encountered. Otherwise, n tuples of the format <parent, data file> must be entered and
#    terminated by the string "NULL".
#
#   For each tuple <parent, data file> entered, the child field specifies the index of child Tj and the
#   data file field specifies the file storing the dependency data that Tj produced. Ti needs this data to
#   begin execution.
#
# 4) The fourth set of entries are also optional. If task Ti is childless, the next entries are ignored until the
#    the string "NULL" is encountered. Otherwise, n tuples of the format <child, edge weight> must be
#    entered and terminated by the string "NULL".
#
#   For each tuple <child, edge weight> entered, the child field specifies the index of child Tj
#   and the edge weight specifies the weight of the edge eij connecting Ti and Tj. For the current
#   implementation of the XFace tool, the edge weights are assumed to be negligible with respect to the
#   task weights. However, for scheduling purposes, all edge weights are assumed to be 1. This
#   assumption does not affect the total execution time of the DAG; it is only used to schedule the task in
#   the DAG.
#
# The information for a task need not be entered on the same line. it can be entered on several lines.
# No comments are allowed on lines that contain task information.
#
#####

#-----
# The sample graph consists of 5 tasks
#-----
5

#-----
# Task TO Information
```

```

#
# (1) Task Weight = 2.0
#
# (2) No tasks in the DAG parent T0.
#
# (3) T0 parent three children. The children tasks are T1, T2, and T3.
#
# (4) Since T0 has no parent, it has 0 dependency data file
#
#-----
0 2.0 0 3 NULL <1 1.0> <2 1.0> <3 1.0> NULL

```

```

#-----
# Task T1 Information
#
# (1) Task Weight = 10.0
#
# (2) T1 has one parent. The task that parent T1 is T0.
#
# (3) T1 parent one child. The child task is T4.
#
# (4) Dependency data file is as follows.
#
#     Parent Task  Data File
#           T0      partA.p0
#
#-----
1 10.0 1 1 <0 partA.p0> NULL <4 1.0> NULL

```

```

#-----
# Task T2 Information
#
# (1) Task Weight = 7.0
#
# (2) T2 has one parent. The task that parent T2 is T0.
#
# (3) T2 parent one child. The child task is T4.
#
# (4) Dependency data file is as follows
#
#     Parent Task  Data File
#           T0      partB.p0
#
#-----
2 7.0 1 1 <0 partB.p0> NULL <4 1.0> NULL

```

```

#-----
# Task T3 Information
#
# (1) Task Weight = 12.0
#

```

```

# (2) T3 has one parent. The task that parent T2 is T0.
#
# (3) T3 parent one child. The child task is T4.
#
# (4) Dependency data file is as follows.
#
#     Parent Task  Data File
#         T0       partC.p0
#
-----
3 12.0 1 1 <0 partC.p0> NULL <4 1.0> NULL
#
-----
# Task T4 Information
#
# (1) Task Weight = 2.0
#
# (2) T4 has three parents. The tasks that parent T4 are T1, T2, and T3.
#
# (3) T4 is childless.
#
# (4) Dependency data files are as follows.
#
#     Parent Task  Data File
#         T1       partA.p1
#         T2       partB.p2
#         T3       partC.p3
#
# (5) Since T4 has no children, there are o Parent-Child edge
#
-----
4 2.0 3 0 <1 partA.p1> <2 partB.p2> <3 partC.p3> NULL NULL

```