CHAPTER 1: INTRODUCTION

1-1. Introduction and Research Goals

A *distributed computation* is a computation split into several processes. In a distributed system, these processes can all be executed on one processing node, or each can be executed on its own processing node, or the processes can be mapped onto processing nodes in many different ways. The choice of how many processes to execute on each node (the *arrangement* of the processes), and which specific processes to execute on each node (the *configuration* of the processes), will have a significant effect on the overall execution time of the distributed computation. When such a distributed, multi-process program is to be executed in a distributed computer system, a *placement decision* must be made concerning where the program's processes should be executed in order to optimize performance. The goal of this research has been to find an effective and efficient *placement procedure* for making placement decisions.

Unless placement decisions are made randomly, they will have to be based on some characteristics of the distributed program, or of the distributed system, or both. We have studied how these characteristics influence the performance of distributed programs executed in various arrangements and configurations. We have searched for the characteristics which have the strongest effect on the performance, and sought ways of using them in placement procedures. We have also studied how performance changes with different characteristics and over different arrangements and configurations, and developed placement procedures based on the patterns of these changes.

Two characteristics of particular interest are parallel computation and communication. There is a conflict between parallelism and communication for communicating processes executed in a distributed system: if the processes are executed on different processing nodes, there is greater parallelism and larger speedup, but communication takes much longer between processes on different processing nodes than between processes sharing a node. If the processes are clustered together on one processing node, communication is much faster, but parallelism is lost. Thus, there is an engineering trade-off to be made between parallelism and communication, between spreading processes out and grouping them together. This trade-off is explored in the thesis.

If the number of arrangements and configurations were small, it might be possible to examine them all, or to select one on an ad hoc basis. With large numbers of both, however, a more systematic approach to making placement decisions is needed. Thus, computing the number of arrangements and configurations for various numbers of processes and processing nodes was a requirement for our project.

If the difference in performance between configurations is small, there is no need to carefully select a configuration. Thus, we wish to examine the differences in performance between configurations.

In traditional load balancing (or load sharing), it is assumed that there is no communication between processes, so there is no communication penalty to executing processes on separate processing nodes. In this work, the relative amounts of CPU use, I/O activity, and communication activity vary between zero and 100%. Thus, we are exploring the general case, of which traditional load balancing is a special case.

1-2. Plan of the Thesis

In this introductory chapter, we present the problem in greater detail, describe the environment in which the problem occurs, and describe our approach to studying the problem. In Chapter 2 we review some of the related previous work, and in Chapter 3 we analyze the combinatorial explosion that makes the problem hard. In Chapter 4 we discuss our approach to the problem in greater detail, describing our experimental design, and in Chapter 5 we describe our software ''experimental apparatus.'' In Chapters 6, 7, and 8 we describe the experiments run, our analyses, and the results. Finally, in Chapter 9 we summarize the results, discuss some general conclusions, and sketch some possible directions for further research.

1-3. The Environment in which the Problem Occurs

1-3.1. Introduction

There are a great many types of distributed computer systems and multiprocessors, many types of processes, and so on. In this thesis, we explore loosely-coupled, or ''shared-nothing'' distributed systems with heavy-weight processes. In this section, we describe the models of the environment on which we have focused.

1-3.2. Distributed Systems

A *distributed computer system*, or metacomputer, consists of a number of processing nodes connected by some sort of communication network. *Processing nodes* execute processes; they consist of a single CPU, a single I/O facility such as a disk, and a connection to the communication medium. (Since processing nodes are the entities that do the processing, we sometimes refer to them as ''processors.'') Each processing node has a private memory which can be accessed only by the node's CPU; we have not considered shared-memory multiprocessors, nor systems in which processors can access each other's memories. When two or more processes are executed on a processor, the CPU is timeshared between them.

We have considered only homogeneous distributed systems, in which all processing nodes are identical; a very interesting area for future research would be to consider heterogeneous systems including processing nodes of different speeds, different architectures, and even processing nodes which are themselves multiprocessors.

The communication medium we have modelled is a fast local-area network (LAN), such as a fast Ethernet or FDDI ring. We believe that most of our results are also true for metropolitan-area networks (MANs) and even for wide-area networks (WANs), but we have not sought to prove it.

The communication medium is uniform and shared; that is, it takes about the same amount of time for messages to get from any one processing node to any other, and only one message can be in transit through the medium. Processing nodes which have traffic to send compete to use the medium on a first-come-first-served (FCFS) basis. Each node queues its outgoing traffic in a FCFS queue.

In a multiprocessor with shared memory there is little or no tension between computation and communications and hence no engineering trade-off. As in a distributed system, there is a speedup due to parallelism when processes are executed on different processors. Unlike a distributed system, however, in a shared-memory system remote messages may be faster than local messages. To send a message locally, that is, between processes sharing a processor, one process sends the message, that is, it writes it in memory, then a context switch is made, bringing the second process into execution. The second process receives the message, that is, it reads it from memory. For a remote message, that is, between processes on different processors, the sending process sends the message, that is, it writes it into the memory. The receiving process can then immediately receive the message by reading it from memory. The remote

message can be faster by the speed of the context switch (even if we ignore the second context switch back to the local sending process).† So, in a shared-memory system, both computation and communication are made faster by splitting processes up and executing them on different processors.

1-3.3. Processes

A distributable multi-process program involves multiple independent processes. Processes do not share address space or memory; we have not considered threads or lightweight processes which share address spaces.

Processes perform sequences of the following types of *actions:* they request execution on the CPU for a specified service time; they perform I/O actions, which may be reads or writes of specified amounts of data; they send messages of specified length to a specified destination process; they receive messages; and eventually they terminate. Sections 4-5 and 5-2.4.3 further describe our model of process behaviour.

1-3.4. Inter-Process Communication

Processes communicate and synchronize by sending and receiving messages. Communication is asynchronous, meaning that send operations are nonblocking. Arriving messages are appended to an incoming message queue for the destination process; the process is not interrupted or signalled. When a process performs a receive operation, it obtains the first message in its incoming message queue; if its queue is empty, the process is blocked until a message arrives.

1-4. Multi-Process Programs

1-4.1. Our Model of Multi-Process Programs

In our model of multi-process programs, a number of processes are created or started at the same time, or perhaps they had previously existed and are reinitialized or reloaded with code;‡ in any case, they are all started at the same time by a parent process. Thereafter the processes proceed separately; they compute, do I/O, and send and receive messages to and from each other, possibly waiting for messages to arrive in order to receive them. They each terminate at different times because they do different amounts of work and wait different amounts of time for messages to arrive.

A program might go through a number of sequential and parallel phases. In sequential phases, a single process is executing. At the beginning of a parallel phase, the parent process creates or activates a number of children; the children then execute until the last one finishes (the parent process may be suspended or may work as one of its own children). The child processes are then destroyed or deactivated, and the parent process executes another sequential phase. The number of parallel phases and the number of child (worker) processes in each phase could be constant, fixed in the code, or it could be determined dynamically. In such a situation, a new placement decision would be made at the beginning of

† This assumes that the destination process is not interrupted by the arrival of a message. In a loosely-coupled distributed system, the arrival of a message through the network probably interrupts the destination processor, bringing the number of interrupts even higher than we had counted. In a shared-memory system, there is no need for the destination process' processor to be interrupted. The destination process' incoming message queue count is incremented by the sending processor writing into memory, hopefully in some safe way (perhaps protected by synchronization primitives). How does the destination process know that a message has arrived? If the destination process is not doing a receive at the time, it does not need to know. If it is doing a receive at the time, it may be spinning on the zero message count. Only if the destination process is blocked waiting for a message, and the block is not a spin lock, does the destination process need to be interrupted. Even then there are other possibilities: if the destination processor is multiprogrammed and timesliced, the blocked destination process could check the count once per timeslice.

‡ Multiple child processes might be created or spawned only once. Then, when one parallel phase ends, they might be left inactive but still extant. In the next parallel phase, child processes need not be created, but only reinitialized and reactivated (and possibly loaded with new code).

each parallel phase, and processes created or migrated as necessary. It might be possible for the parent process to spawn a placement-decision process to compute the placement decision shortly before the end of the sequential phase, so that the decision would be ready at the beginning of the parallel phase.

If new child processes are created or activated during a parallel phase, after other children have already started, this should be considered a new parallel phase. The placement procedure can be rerun and a new placement decision made based on the new number of child processes.

If the child processes executing during a parallel phase go through a number of phases of significantly different behaviour, a new placement decision can be made before or at the beginning of each phase. At the point of a phase change, all processes can be frozen (or each can be frozen as it hits its phase change point), the results of our study can be applied to make a new placement decision based on the new behaviour which is about to begin, processes can be moved as necessary, and then the processes can be unfrozen and allowed to continue. In this case, the time during which the processes are frozen must be counted as a time-cost in deciding whether it would be worth rearranging or whether it would be better to continue to execute in the old (possibly sub-optimal) configuration. As mentioned, it might be possible to compute the new placement decision in parallel with the executing processes before the phase change; it might even be possible to move the child processes while they are executing.

1-4.2. The Multi-Process Programs We are Modelling

Our chief interest is in large, scientific programs, especially those in regular, production use. We feel that only programs which are large and do a great deal of computation are likely to be made into distributable programs. Transforming a program into a distributable program is neither cheap nor easy, despite improvements in parallelizing compilers.

In order to make the most thorough study possible, and to give our results the broadest applicability, we have defined a space of all possible distributable programs, and explored all parts of that space using a large number of artificially generated programs. This also avoids tying our study to the characteristics of a few, possibly atypical, programs.

1-5. Measure of Performance

The measure of performance we use is the response time of the distributed program, determined by the last-finishing process. Response time is elapsed wall-clock time, measured from the moment all the processes start, all at the same time, to the moment the last-finishing process finishes. There are other measures which might be of interest in certain circumstances, but we feel that overall response time is the measure of greatest practical importance, and the measure of most use in dealing with distributable multi-process programs.

1-6. Research Goals and Non-Goals

This research does not deal with parallel solutions for problems, parallel algorithms, parallel languages (languages for writing parallel programs), or parallelizing compilers. Nor are we investigating operating system support for designing or writing parallel programs. We are considering what should be done at a moment long after all of these things have been done, the moment when the operating system has been instructed to execute some program, and from some repository of information it discovers that this is a distributable multi-process program, one which involves a number of processes (or possibly has several parallel and sequential phases). The operating system then has to make a real-time, execution-time decision: on which processing node in the distributed system should each process be executed.

A related situation, in which we are also very interested, is that of a large program which is ready to be deployed in a production mode, where it is going to be executed a large number of times. In this case,

a manual, ‘‘off-line’’ study may be amply justified. This study could be allowed to use quite a lot of time and resources; since there is no immediate execution-time decision required, there is no urgency to finish it quickly. Such a study could be made thorough enough to identify, not only a good configuration, but the best possible configuration. In fact, the study could identify the best possible configuration for each of a number of situations which could arise at execution time. At execution time, then, the system could look up the best configuration for the then-current situation.

1-7. The Approach

Our general approach is to use simulation. We attempted a number of analytical modelling approaches, which were not fruitful. In each approach, by the time the model was complex enough to be a reasonable representation, it was too complex to readily use and understand. Specifically, we made a number of attempts to use detailed information about timing in each process, for example, exactly when a process sends a message, how long a process computes before sending or receiving a message, how long computation takes, how long a process waits for a message. These attempts were not successful. At that detailed timing level, there is too much information to handle, and the information is very unstable, that is, it changes very easily and one small change in timing for one process causes many changes for other processes.

We also attempted to model the behaviour of processes with a finite state machine. We had hoped that a machine with just a few states could model the important aspects of computation, message sending, and message receiving, but the machine quickly grew to over a dozen states and a large number of transitions. This complex machine was not helpful in understanding the behaviours and interactions of processes, so we abandoned this approach, too.

Other approaches we considered or attempted were Petri net models [Pet77, Pet81, Rei85] and PERT charts [MoP70, Sch88].

At the other extreme, we did not pursue experimental measurement approaches. Building a system which we could then measure would have left us with little flexibility. Gathering information on a number of distributed programs from a number of systems was also fraught with difficulties: the programs and the systems, though different, might have turned out to be quite similar in their essential characteristics; we wanted to explore a wide range of possibilities. Alternatively, the programs and the systems might have been so different as to be incomparable. There was no assurance that we would be able to obtain information on a sufficient number of programs and systems. The programs we investigated might have all been atypical in one way or another. Even at best, the information we gathered would represent only current usage, providing no information on trends and what to expect in the future.

We settled on using a simulation approach. Simulation allows us to bring quite a bit of complexity to our models. At the same time, we were able to make large numbers of simulation runs, covering a wide range of characteristics in a controlled way.

One drawback of simulation, as opposed to measurement, is that the results are only as good as the simulation model. We built our own simulator and supporting software, the Carnival system, which is described in Chapter 5.

For similar reasons, we decided to use artificially generated traces. We wanted to explore all possible combinations of distributed program characteristics. With artificial traces, we were able to generate whatever trace characteristics we wanted, covering the program characteristic space completely. One drawback of this approach is that the traces, and hence the results, are only as good as the trace generator. We have built our own trace generator, GENACTIONS, which is also described in Chapter 5. Another drawback of this approach is that it does not give us any information about the characteristics of real distributed programs. On the other hand, this is also an advantage: our results, based on study of the

complete program characteristic space, should be applicable both now and in the future; results based only on current, real programs could quickly become out-of-date.

Using a methodology based on simulation means that we cannot prove our results, nor make unequivocal statements. We can, however, discover certain results based on our simulation outcomes. These results demonstrate, or support the allegation that the same results which are true in our simulations are also true of real distributed programs in real distributed computing systems, assuming that our simulation model and trace generation model are good.

1-8. Characteristics to Use in Placement Decisions

We would like to use only the most general, most stable information possible in the process of selecting the fastest configuration. The more general the information, the broader the range of situations to which the placement decision applies, and the more stable the information the decision is based on, the less often the decision-making process must be repeated.

If the fastest configuration could be identified based only on information gleaned from the program source code, the placement decision would be very broad and very stable. No matter what the input, the characteristics of the system, the load, and so on would be, the decision would still be correct. A placement decision based on the source code would also be stable; production programs change very seldom, so the source code, and decisions based on the source code, are quite stable. Given the broad applicability and stability of such a decision, the fastest configuration could be precomputed and stored, so that the amount of time needed to make the placement decision at execution time could be reduced to retrieving the precomputed configuration. However, we do not believe that source code contains sufficient information about process behaviour to warrant exploration of source-code-based placement. So, we shall explore placement decisions based on program behaviour.

The program code, combined with typical input data, defines the program's behaviour. Input data includes data values read by the program from interactive terminals and from files, as well as command line arguments, interactive session characteristics or switches (called *environment variables* in UNIX†), system characteristics or switches, and anything else that affects the behaviour of the program. Possibly several input sets may have to be used and somehow combined or averaged to obtain typical behaviour. The traces we use in our experiments are supposed to represent program behaviour on typical input data.

We believe that program inputs are fairly stable, in the aggregate and over the moderately long term. We are considering large, production programs, which have large input data sets. Although the specific input values may change drastically from run to run, at a slightly higher level of abstraction, the characteristics of the input data set which affect performance and placement probably do not change much. In a payroll program, different employees will be on vacation or working overtime from week to week, but the number of employees on vacation or working overtime is probably pretty much the same from week to week in a company with thousands of employees. In an atmospheric dynamics simulation, the air currents being simulated may be very different from run to run, but the number of major weather fronts will probably be similar. The important characteristics of the input data vary over a fairly narrow input domain. To the extent that the important characteristics do change, we believe they drift gradually rather than making sudden large jumps. Thus, determining how fast the values of the key characteristics drift would tell us how stable the input data characteristics are. This information could be used to create a schedule for characterizing the input, and possibly precomputing placement decisions. In many cases, the performance of the distributed program does not change greatly in response to changes in the input data, so placement decisions based on one input data set would be correct for other similar sets. Thus, the apparent or

† UNIX is a registered trademark of UNIX System Laboratories, Inc.

effective stability in the input data characteristics may be even greater than the actual or literal stability, and placements based on the input would be even more general.

The process traces we use are partial descriptions of distributed program behaviour. They contain only the sequence of actions that each process carries out, without any information about process placement or hardware-dependent timing (except for CPU service requests, and even those are normalized times). The traces are actually artificially generated, but they are intended to represent program behaviour in response to typical inputs, independent of hardware, load and other considerations. The *execution behaviour* of the program will depend on the configuration chosen, and on the characteristics of the distributed system used to execute the program.

We would like to be able to select the faster configuration based only on the program behaviour, that is, on the characteristics of the trace, without using any information about the computer or communication systems, or about the system load. If this were possible, we could use the selected configuration no matter what system the program was executed on, and no matter what the load. We would not have to re-evaluate the placement decision based on changes in the system characteristics or changes in the load.

System characteristics are less general and less stable than program and input characteristics. We believe that system reconfigurations and hardware upgrades happen more frequently than changes to production programs. Certainly, when hardware is upgraded, it changes the system characteristics significantly (else why do it?), whereas changes to production programs tend to be small and local, probably having little effect on the performance characteristics, and therefore little effect on placement decisions. When a program is ported from one system to another, the program should not change much, but the system may be completely different, so porting is a change in system characteristics.

The system characteristics, combined with the program and input, determine the *execution behaviour*. Process placement methods based on execution behaviour are implicitly based on the system characteristics. For example, the amount of time processes spend waiting for messages to arrive depends on system characteristics such as the communication hardware and software, as well as on the program and input data. We have used the same system characteristics in all our simulation runs. (The characteristics are described in Section 5-2.3. The effects of changing system characteristics on process placement are discussed briefly in Chapter 9).

The load on various parts of the system, other than that due to the program whose processes are to be placed, also affects the choice of configuration. The system load has a number of components, based on different types of resources in the system, such as CPUs, disks, and so on, and, in a distributed system, multiple instances of resources. A placement decision based on the system load would be neither broad nor stable. It would apply only to the specific system load values it was based on (or a narrow range of similar values), and it would not be stable because the outside load on each component can change rapidly [Zho87]. In our experiments, the distributed program being placed was the only load on the system. (Carnival is capable of simulating a variety of dynamic load situations; see Section 5-2.4.) A brief discussion of the effects of dynamic load on placement can be found in Chapter 9.