CHAPTER 2: PREVIOUS WORK

2-1. Introduction

In this chapter, we briefly review the literature representing previous work relevant to our investigation. (We do not attempt to survey the entire literature on distributed and parallel computing; broad, though in some cases dated, surveys are available [Bae73] [AnS83] [Kle85] [Kar87].) We begin in Section 2-2 with the basic question of whether distributed processing is worth investigating. Finding that it is, we turn in Section 2-3 to the question of finding optimal placements. It is possible to compute optimal task or process placements, but only in certain restricted cases. In Section 2-4, we investigate heuristic approaches that might be applied in more general cases. In Section 2-5, we touch upon two other approaches: theoretical behaviour models, and compile-time partitioning and placement.

In Section 2-6 we consider the problem of scheduling a number of parallel and serial jobs on a multiprogrammed multiprocessor. In this situation, the question is how to allocate processors to processes.

The question of balancing computation and communication also arises in the case of highly parallel computations on parallel computers, for example, in solving a partial differential equation representing a fluid dynamics problem on a hypercube architecture. We investigate the literature on this problem in Section 2-7.

For best results, decisions concerning the placement of processes should depend on the behaviour of the processes. While we take an exhaustive approach in this thesis, experimenting with all combinations of process characteristics, in Section 2-8 we review some measurements of the characteristics of real processes in distributed programs.

Finally, in Section 2-9, we summarize our review of previous work.

2-2. Is Distributed Processing Worthwhile?

Kanakia and Tobagi [KaT87] compare distributed and centralized systems dedicated to performing a single set of tasks, where the capacity (processing power) of the processor in the centralized system is equal to the sum of capacities of the processors in the distributed system. When the processing of the centralized system is modelled by a single-stage service queueing model, the average execution time in the centralized system is shorter than that in the distributed system, both for algorithms involving sequences of tasks and for those involving parallel tasks. When the centralized system is modelled as a multi-stage service, where each stage models the service of an individual task, the centralized system is still faster on average than the distributed system for jobs consisting of sequences of tasks. For algorithms involving parallel tasks, the multi-stage model shows that the average execution time in the distributed system is shorter when the coefficient of variation of the service time in the centralized system is greater than one. This is the case, for example, when there are a large number of requests each with a small amount of work.

Kanakia and Tobagi point out that their analysis does not apply to general-purpose computing systems which can run tasks from multiple programs, but only to dedicated systems. Furthermore, they remind us that ‘‘there are other important criteria by which to compare distributed and centralized systems, such as reliability, ease of system design, maintainability, flexibility in adapting to other algorithms, etc.,

but such criteria are not considered'' in their paper.

Grosch's Law, proposed by Herbert Grosch in the 1940s, states that the costs of computer systems increase at a rate proportional to the square root of their computing power. In other words, economies of scale apply to computing systems. Ein-Dor [Ein85], and later Kang, Miller, and Pick [KMP86] re-examined Grosch's Law in the 1980s, and found that computers fall into distinct categories according to cost and power: supercomputers, mainframes, small mainframes, minicomputers, and microcomputers; the slopes of the cost/power curves are different for different categories. It is most effective to use the least powerful category of computer capable of accomplishing the required tasks. Grosch's Law holds within most categories, so it is generally advantageous to use the most powerful computer within a category. So, one powerful microcomputer is preferable to two less powerful microcomputers, but a number of microcomputers are preferable to a minicomputer that has the combined power of the micros. Thus, distributed systems are financially advantageous.

Mendelson [Men87] disputes Ein-Dor's and Kang, Miller, and Pick's results, showing that the categorization they use is a statistical artifact. He claims that, while Grosch's Law had been true in the past, by the early 1980s cost per MIPS was constant, and that there are no longer either economies or diseconomies of scale in the cost of computer systems. With no economic justification for centralized systems, other, non-economic factors determine the choice between centralized and distributed computing. Mendelson suggests that the non-economic factors favour distributed systems, and that the trend toward distributed systems, starting in the late 1970s, is due to the fact that Grosch's Law is no longer true.

King [Kin83] discusses non-economic factors at length, pointing out the importance of considering organizational and political factors, as well as technical and economic ones, in choosing the degree of centralization and decentralization for an organization's computing systems. The options range from extreme centralization to extreme decentralization, and he discusses the organizational and political advantages and disadvantages of various options. He advocates a mixed solution involving somewhat centralized control of computing with somewhat decentralized facilities and functions.

Lantz, Nowicki, and Theimer [LNT85] have evaluated the actual performance of a number of applications on slower local workstations and much faster remote compute servers in the Stanford V System [Che84]. They found that applications can run faster as distributed programs than as purely local programs, despite communication delays. The primary factors affecting performance are, in descending order of importance: the speed of the local workstation, the speed of the remote host, and the high-level communication protocols used.

Thus, there are financial and organizational as well as technical reasons to explore distributed computing.

2-3. Computing Optimal Placements

2-3.1. Optimal Placement of Serially Parallel Programs: Stone and Bokhari

Stone [Sto77] proposes modelling distributed programs with augmented commodity flow graphs, and he shows that well-known, generally efficient algorithms for finding a minimum cutset of the graph find an optimal assignment of the modules to two processors. Stone also describes some incomplete work toward applying this approach to several processors. Unfortunately, Stone's approach applies only to what he calls serially parallel programs, programs in which only one module at a time executes on one processor, while other processors wait or execute modules of other programs.

While the minimum cutset approach may be of very limited usefulness, Stone's discussion includes several very interesting ideas and suggestions concerning module assignment. Stone suggests that working-set-like clusters of communicating modules should be assigned to the same processor to take

advantage of locality. He describes how an assignment algorithm can be used statically at program loading time, or dynamically (either when the program changes phases or working sets, or when the system load changes).

Modules may have very different processing times or costs on different processors due to different hardware or software resources available on the different processors; Stone also considers modules that have ''fixed'' assignments because of resources available on one or a small set of processors. Although heterogeneity may have many dimensions and special cases, Stone suggests dealing with it by modelling the execution time of a module on one processor as a simple multiple of the execution time of the module on a second processor, where the multiple is determined ahead of time over a broad workload.

Stone's technique requires knowing the execution time of each module on every processor, and the number and cost of inter-module communications between every pair of modules. In general, it would be very costly to gather such data. Stone suggests that initial estimates of these quantities can be made by compilers, and the estimates can be used to compute initial assignments. As the program runs, the execution times of modules on whatever processors they run on can be monitored, as can the number and cost of inter-module communications. This real, more accurate data can be used to compute module reassignments dynamically. Monitoring is probably already done for accounting purposes, Stone claims.

Stone also points out that response time may not be the only cost function of interest in determining optimal assignments. Dollar cost may be of more importance than absolute running time. He points out that the costs of module executions and inter-module communications can be given in dollars instead of times, and the same assignment algorithm used; this applies to many other approaches to module assignment as well.

In another paper [Sto78], Stone follows up on the idea of using minimum cutsets to determine optimal module assignments. He proves that, given two processors, with all else held fixed, as the outside load on one processor increases, fewer modules will be assigned to that processor and more to the other, fixed-load, processor. Therefore, Stone suggests, it is possible to precompute the ''critical load factors,'' the threshold at which each module will be reassigned, and so do dynamic, even real-time reassignment using table look-ups. As with his earlier paper, the specific technique suggested may not be very useful, but the general ideas discussed are thought-provoking, particularly the idea of precomputing assignments.

Bokhari [Bok88] has extended Stone's work considerably. He has investigated chain and ring structured computations, where each module i communicates only with modules $i-1$ and $i+1$, in situations in which only contiguous chains of modules may be assigned to processors. He gives a technique for finding an optimal assignment of chains of modules to processors in such a situation, but the complexity of the technique is $O(m^3 n)$ for m modules and n processors. Bokhari also considers situations with a single large host and several much-less-powerful satellites, and develops techniques for finding optimal assignments for several problem structures: (1) chains of parallel, pipelined modules, where each chain runs partly on a satellite and partly on the host; (2) several serial programs operating in parallel; and (3) a single tree-structured program. In all these cases, the first part of each chain, or the most leaf-ward parts of a tree, are executed on the satellites, and the later parts of the chains or the upper parts of the tree are executed on the host. Bokhari's technique for finding assignments involves constructing a doubly-weighted graph, with a sum weight for executing a module on the host and a bottleneck weight for executing the module on a satellite. The larger of the sum and the bottleneck, the SB weight, determines the running time; either the sum of the running times on the host is greater than the running time of any satellite, or one satellite, the bottleneck, is slower than the others and slower than the host. A path through the graph is then found which corresponds to the minimum possible SB weight, and the path defines an assignment of modules to processors. The handling of the problems differs in how the doubly weighted graph is constructed. The time complexities of the algorithms for handling the three types of problems are $O(m^3 n \log m)$, $O(m^4 n)$, and $O(m^2 \log m)$, respectively.

2-3.2. Optimal Placement in Random Programs: Indurkhya et al. and Nicol

Indurkhya, Stone, and Xi-Cheng [ISX86] analyzed optimal partitioning of random programs. The ‘‘tasks’’ of which a program is composed, which we refer to as processes, are represented by random ‘‘running times’’ (CPU demands). Tasks communicate with each other with a certain probability; if they do communicate, the communication times are also random variables. In their model, the total response time of a program in a given configuration is the sum of the running times of the tasks on the busiest (longest-running) processor and the sum of all communication times from processor to processor (communication between tasks sharing a processor costs nothing). This model of total response time assumes that all communication uses a common link such as a shared bus, or a ring that acts like a bus. The model does not take account of synchronization delays; that is, it assumes that tasks are never delayed waiting for messages from other tasks to arrive. We examine synchronization delays in Section 6-7.4, and find that synchronization delays are surprisingly large even for extremely communication-intensive programs. They say that their model also assumes no overlap of computation and communication, that is, a task sending a message is delayed until the message arrives at its destination task; in fact, their model makes the even stronger assumption that there is no overlap between communication between *any* process pair and computation on the busiest processor. We do not believe that this is a good model of most current systems.

To analyze their model, Indurkhya, Stone, and Xi-Cheng must find the expected value of the sum of the runtimes on the busiest processor, which is

$$E\left[\ \mathrm{MAX}\left\{\sum_j r_{1,j},\ \cdots,\sum_j r_{n,j}\right\}\right]$$

where $r_{i,j}$ is the CPU demand (runtime) of the j-th task on the ith processor, and the sums are over the tasks assigned to each processor. To simplify this expression by eliminating the MAX operator, Indurkhya, Stone, and Xi-Cheng introduce the approximation

$$E\left[\ \mathrm{MAX}\left\{\sum_j r_{1,j},\ \cdots,\sum_j r_{n,j}\right\}\right]\ =\ \mathrm{MAX}\left\{E\left[\sum_j r_{1,j}\right],\ \cdots,E\left[\sum_j r_{n,j}\right]\right\}$$

Using this assumption, the authors analyze the optimal task assignment in four cases: (1) when there are two identical processors, (2) when there are P identical processors, (3) when there are two processors with different processing rates, and (4) when there are P processors with different processing rates. For identical processors they find that the optimal assignment is either (a) to spread the tasks out, assigning (approximately) the same number to each processor, or (b) to assign all the tasks to one processor, depending on the ratio of computation to communication. For processors with different processing rates, they find that the optimal assignment is either (a) to spread the tasks out, using as many processors as possible such that the runtime on each processor is as even as possible, or (b) to assign all the tasks to one processor. Again, which assignment is best depends on the ratio of computation to communication, in this case scaled by the ratio of processing rates.

Nicol [Nic89] points out that the approximation of the mean of the maximum used by Indurkhya, Stone, and Xi-Cheng is incorrect, and he re-examines their results concerning identical processors. For two identical processors, Nicol proves the same result by a different method, showing that the total response time of N tasks is minimized by assigning either $N/2$ or N tasks to one of the two processors.

For P processors, Nicol constructs an example showing that Indurkhya, Stone, and Xi-Cheng’s proofs are not correct. Using their analysis, the best assignment of four tasks to three processors is 2:2:0 (two tasks on each of two processors and one processor left idle). Using their approximation, the mean maximum CPU time is the maximum mean CPU time, which in the 2:2:0 assignment is the mean CPU time required by two tasks. A 2:1:1 assignment (two tasks on one processor and one task on each of two other processors) would have the same maximum mean CPU time as 2:2:0 and a larger communication time,

and hence, according to Indurkhya, Stone, and Xi-Cheng, the 2:1:1 assignment would always have a longer total response time. However, the mathematically correct mean maximum CPU time of the 2:1:1 assignment is slightly lower that that of the 2:2:0 assignment, so, if the increase in communication cost is sufficiently small, the 2:1:1 assignment could have a faster response time. Nicol mentions running numerous simulations, in which the assignments suggested by Indurkhya, Stone, and Xi-Cheng's results are always best. He suggests that their results are correct and that a ''better,'' correct proof is needed.

2-4. Heuristic Placement Algorithms

Efe [Efe82] claims that heuristic approaches, which aim only to find a good assignment, rather than the best possible assignment, are ''faster, more extensible, and simpler than optimal solution techniques.'' His proposed approach deals both with outside load on the processors, and with modules which are ''attached'' to one processor or a small set of processors. He proposes a two-phase heuristic approach in which clusters of modules are formed so as to minimize communication, following which modules are moved from cluster to cluster to balance the load on the processors, including the outside load. The ''module clustering algorithm'' (MCA) forms clusters by repeatedly coalescing the two modules which have the largest volume of communication between them. At each step, the clusters are assigned to processors, taking into account any attached modules and attempting to balance the total load on each processor, including the load from the modules and the outside load. After all modules have been coalesced, the cluster assignment which best balances the load is selected. If the loads under the selected assignment are sufficiently well balanced, then the assignment is used. ''Sufficiently well balanced'' is defined in terms of a maximum acceptable deviation in the ratios l_i/l_{avg}, where l_i is the load on processor i and l_{avg} is the average load over all processors.

If the loads are not sufficiently well balanced, then a ''module reassignment algorithm'' (MRA) is invoked. This algorithm involves creating a process graph which represents load and communication. The graph is then adjusted to increase the apparent amount of communication between modules assigned to overloaded processors and modules assigned to underloaded processors (where load includes any outside load). The MCA is run again on the adjusted graph and the balance of the load in the resulting assignment is again evaluated. If the load is sufficiently well balanced the assignment is used, if not, the MRA and the MCA are run again.

Efe claims that the MCA algorithm was tested and that it works ''sufficiently well,'' although there is a ''special case'' in which it terminates unsuccessfully; this ''special case'' is ''likely if the original process graph was tree-like.'' No information is given about the running times of the MCA or the MRA, or about how many iterations are needed to reach an acceptable assignment. These algorithms must be run at program load time since they involve the current load on all processors in the system. The execution time of these algorithms is crucial, not only because their execution time directly increases the program's response time, but also because the system load changes very quickly [Zho87] and, if the computation of the assignment takes long, the load on which it is based may have changed.

Another heuristic algorithm for allocating tasks to processors is proposed by Chu and Lan [ChL87]. The objective function used in most task allocation algorithms is the sum of computation costs and interprocessor communication (IPC) costs. Chu and Lan claim that for real-time systems it is more important to minimize response time; we believe that minimizing user response time is the most appropriate objective for most computing situations. Chu and Lan propose to minimize response time by finding the allocation of tasks to processors which gives the smallest bottleneck processor time: for each assignment they compute the running time of the longest-running processor, the bottleneck processor, then they pick the assignment with the fastest bottleneck time. (Compare with Bokhari's use of bottleneck weights in Section 2-3.1.) Their simulations show that minimum bottleneck assignments also balance the load among processors fairly well.

Chu and Lan also show that precedence relations have a significant effect on the performance of assignments. Job arrivals, and hence module invocations, tend to be bursty. If a long-running module is the last module executed on one processor, before the following module is executed on another processor, then the arrivals at the next processor will be more regular, less bursty, thus reducing queue waiting times at the next processor and improving overall job response time.

They propose a heuristic assignment algorithm which they call P-I-A, which takes into account precedence relationships between modules (P), intermodule communication (I), and the accumulated execution time, or CPU service time, of each module (A), and uses minimum bottleneck as its objective function. Their simulations show that the P-I-A algorithm chooses better assignments than a similar algorithm which does not consider precedence relationships. P-I-A generates assignments which yield response times comparable to those chosen by the ‘‘time-prohibitive’’ exhaustive search method.

Applying the P-I-A algorithm requires a lot of information about module execution times, intermodule communication, and the precedence relationships between modules. Chu and Lan advocate estimating this information through analysis and modelling [CLH84]; however, it might be easier and more effective to derive the required information from data gathered by monitoring example executions. For example, Miller, with others, has discussed the design and implementation of distributed program monitoring tools and the information which can be derived from them, including the derivation of precedence relationships and critical paths [MiY85] [Mil88] [YaM88]; others have also done work in this area (see, for example, [JLS87]).

2-5. Other Approaches: Theoretical Behaviour Models and Compile-Time Placement

2-5.1. Theoretical Behaviour Models

Garg [Gar88] proposes a Synchronous Token-based Communicating State model (STOCS) for the specification and analysis of distributed programs with many identical processes. While we are interested in performance issues for programs of this type, Garg’s model is useful for analyzing basic properties such as the reachability of certain states.

Hwang, Chow, and Anger [HCA88] discuss speedup and communication costs in message-passing distributed systems, including what they call ‘‘hidden overhead’’: ‘‘communication overhead can sometimes be ... hidden by scheduling in such a way that messages are generally sent before the receiver is ready to execute.’’ We have generalized this idea, quantified it, and measured it (see Section 6-7.4). Many researchers use speedup as a measure of success for various algorithms, mechanisms, and techniques. Hwang, Chow, and Anger eschew this approach, preferring to analyze scheduling as one of the basic determinants of speedup. They develop the idea of separate ‘‘efficiency losses’’ due to system overhead (by which they mean communication delay) and due to scheduling (such as scheduling a receiver to run when it is waiting for a message to arrive), and try to decompose the total efficiency loss into these two components.

2-5.2. Compile-Time Placement

We are investigating the problem of scheduling processes to run on processors at execution time, given that the program has already been partitioned into processes. Sarkar [Sar89] has investigated the possibility of doing scheduling at compile time. He rejects run-time scheduling as too time-consuming and presents two compiler-based approaches to partitioning and scheduling. In one, partitioning is done at compile-time and scheduling at run-time, but he does not discuss the run-time scheduling. In the other, both partitioning and scheduling are done at compile-time.

2-6. Scheduling Multiprogrammed Multiprocessors

Our work concentrates on the scheduling and response time of a single distributed program. A related research problem concerns scheduling the execution of a number of jobs, some of which are multi-process programs, on a multiprogrammed multiprocessor or distributed system by allocating processors to jobs.

Blazewicz, Drabowski, and Weglarz [BDW86] develop algorithms for finding optimal schedules, but only for fairly restricted cases. When scheduling is non-preemptive, they consider only tasks requiring one unit of time; non-preemptive scheduling of non-unit-time tasks is NP-complete in the strong sense. When scheduling is preemptive, they are able to handle non-unit-time tasks, but the set of tasks to be scheduled may only include those requiring one processor and those requiring exactly k processors, for some fixed k. While their algorithms for finding optimal schedules are polynomial-time algorithms, the running times are likely to be very long, for example, greater than $O(n^m)$, where n is the number of tasks and m is the number of processors.

Kruskal and Weiss [KrW85] studied whether tasks should be allocated to processors all at once, one by one, or in batches of size K (where $1<K<n/p$, n is the number of tasks, and p is the number of processors), and how many tasks should be allocated to each processor. Their analysis shows that allocating an equal number of tasks (n/p tasks) to each processor all at once has good efficiency. They say that this is true even when taking communication into account, although by communication they mean the cost of processors accessing the task queue. They report simulation results which support their conclusions.

Turek et al. [TWP92] treat the processor allocation problem as a variation of the rectangular bin packing problem, with tasks as rectangles whose width represents the number of processors used and whose height represents the running time. The number of processors available defines the width of the bin, and the objective is to pack the rectangle-tasks into the smallest possible height. The rectangles are malleable in that they can be made wider (using more processors) and shorter (faster running time). Turek et al. investigate the shelf strategy for solving the orthogonal rectangle packing problem, and, by using resource allocation theory and other combinatorial optimization techniques, solve the multiprocessor scheduling problem in the shelf paradigm.

Ghosal et al. [GTS90] simulated three processor allocation policies with a small look-ahead window into the waiting job queue: (1) First Fit (FF), which allocates a job's processor working set (pws) of processors to the first job in the window for which there are enough free processors. This policy is not work conserving, that is, processors may remain unassigned; (2) First Fit plus FIFO (FF+FIFO), which operates like FF but, after allocating a job's pws, it allocates any remaining processors to the first job on the queue; and (3) First Fit plus FIFO with Largest Available (FF+FIFO+LA), which operates like FF+FIFO but allocates the largest number of processors possible to the job if there are no more jobs in the queue. FF+FIFO was found to be better than plain FF because it makes use of all the processors. The LA feature was not advantageous, as allocating all the processors when there are no jobs in the queue did not decrease the execution time of the current job enough to compensate for the additional waiting time that future arrivals face because all the processors are busy.

Sevcik [Sev89] has also simulated processor allocation policies in his investigation of characterizations of parallelism and their use in scheduling. He introduces the parallelism profile and parallelism shape graph, which capture all the information about a program's ability to use multiple processors. He then reports on simulation experiments in which scheduling was done using three rules, based on small and increasing amounts of information: average parallelism only; average parallelism and its variance; and the average, variance, minimum, and maximum parallelism. All three yield identically good performance at low system loads (up to 50% utilization). At higher loads, the rule with the most information yields the best performance. Even so, scheduling using only four parameters performs quite well.

2-7. Highly Parallel Computations

The question of assigning computations to processors in the face of computation, communication, and other conflicting performance considerations also arises in the case of highly parallel numerical computations, such as solving partial differential equations, which are needed in fluid dynamics, structural mechanics, and a number of other applications. Usually, the domain over which the computation is to be run is divided by a grid, and the equation is evaluated at each grid point through a series of iterations which continue until a termination condition is met, usually until the error is sufficiently small. In a parallel computer system, sets of grid points comprising regions of the grid are assigned to each processor. In each iteration, the processor evaluates the equation at each grid point assigned to it, using the past value at each point and at a number of neighbouring points. To perform the next iteration, processors may have to exchange the values of points whose neighbours have been assigned to other processors. Thus, the algorithm used to assign regions to processors determines how well the computational load is balanced among processors and how much interprocessor communication is necessary, and is thus critical to performance.

Many of the methods for solving partial differential equations are adaptive, that is, they use a finer grid in some regions than in others, so that the work required is not uniform over the entire grid. If the full grid is partitioned in a regular manner into regions of the same size and shape, different processors will be assigned different workloads: some processors will be assigned regions divided only by the coarsest grid and containing few grid points, while other processors will be assigned regions divided by finer subgrids and containing many grid points. With different workloads, one of the processors will be the bottleneck, and the others will be underutilized. Berger and Bokhari [BeB87] propose a method of recursive binary partitioning which, at each step, cuts each region into two parts so as to put (as nearly as possible) half of the work of the region in each of the two subregions. The partitioning is repeated recursively until the amount of work in each region is small enough, or until the number of regions is equal to the number of processors available; each region contains the same amount of work as every other region. The regions are all regular rectangles, which makes it easier to figure out in which region neighbouring points lie; on the other hand, the rectangles are not aligned, which makes is difficult to figure out the communication costs involved. Berger and Bokhari analyze the communication costs for mesh, tree, and hypercube communication structures, using two approaches for dealing with the alignment irregularity. In the end, they suggest that it might be better to use a weighted sum of computational cost and communication cost in the partitioning, rather then only the computational cost.

Baden [Bad87] implemented recursive binary partitioning as part of a virtual machine interface intended to insulate programmers from many of the details involved in programming different parallel computer architectures. He found that on the Intel Personal Scientific Computer (iPSC), a hypercube-type multiprocessor with 32 processors, the overhead of the load balancing utilities was less than 10%, including interprocessor communication. He says that ‘‘this is surprisingly low in light of concerns voiced that the recursive bisection strategy would introduce high communication overhead on message-passing architectures.’’ On a Cray X-MP/416, a four-processor shared-memory architecture with vector arithmetic capabilities, the overhead of load balancing was less than 5% [Bad87, p. 6].

Although Hinz [Hin92] discusses the importance of considering communication overhead in partitioning the data space for highly parallel numerical computations, his development, analysis, and evaluation of *N*4 net partitioning ignores communication costs. Later, in comparing *N*4 net partitioning to recursive bisection partitioning, Hinz claims that the *N*4 topology of distorted quadrangles based on a grid allows for a much simpler and more static communication structure than the strips and rectangles produced by recursive partitioning. Thus, he says, the *N*4 topology allows for efficient handling of communication.

Brochard [Bro87] points out the importance of considering ‘‘control costs’’ as well as computation and communication costs. By control costs, he means the time required to create and terminate tasks (forks and joins), and the time required for synchronization and termination detection. Since the former

are done only once for the entire computation and the latter are done in each iteration, he considers only the latter. He evaluates the speedup without any control costs, with linear control costs, and with logarithmic control costs, and finds that the control costs are significant.

Nicol and Willard [NiW88] examine the effects of problem size, problem partitioning, and communication architecture on speedup in the parallel computation of a partial differential equation on an n-by-n grid. They show that for hypercube and grid communication architectures, and for Banyan-type switching network communication architectures, it is best either to use all available processors or just one processor. For bus-type communication architectures, whether synchronous (processor blocked while communicating) or non-synchronous, using all available processors is usually best. For bus architectures it is possible that the optimal number of processors to assign is more than one but less than the number of processors available; this can occur if the ratio of the fixed cost of communication to the per-volume cost of communication is small. In one machine they examined, the Flex/32, the ratio had a value of about 1,000, far from small. Nicol and Willard also analyze the ratio of computation cost to communication cost for the different types of communication architectures.

Nicol and Willard examine the speedups obtained with different types of communication architectures as the problem size increases, given as many processors as desired. For hypercubes and grids, speedup grows linearly in n^2 (for an n-by-n grid); in switching networks, speedup grows proportionally to $n^2/\log(n)$; and, in bus architectures, speedup grows only as $(n^2)^{1/3}$, even if bus access is completely asynchronous. They conclude that communication volume and contention should be avoided as much as possible. The partitioning affects the communication volume, and the communication architecture affects the amount of contention.

2-8. The Characteristics of Processes in Distributed Programs

In their 1986 SIGMETRICS paper, Leland and Ott [LeO86] present their analysis of over 9.5 million UNIX† processes executed at a commercial research laboratory. They found that ‘‘there are three distinct populations of processes: CPU hogs (which make relatively few disk accesses), disk hogs (which use relatively little CPU time), and ‘ordinary’ processes (which use relatively little CPU of I/O, but have no compelling relation between their use of either).’’ A small percentage of the largest processes accounts for a large fraction of all CPU time used. For example, the 98% smallest processes account for only 35% of all CPU time used, while the 0.1% largest processes account for 50% of all CPU used. Thus, the distribution of processes versus service time has an extremely thick tail as CPU service time grows very large. They found that ‘‘it is a serious error to model the CPU requirement of processes by an exponential distribution, or any distribution with an exponential tail.’’ Similar but less regular behaviour was found for the number of disk accesses made by a randomly chosen process. Unfortunately, Leland and Ott’s study does not address the behaviour of the processes of parallel, distributable programs.

Rommel [Rom91] examined nearly 200,000 processes from an academic environment and found that the process service times tended to fall into two categories: short or long, with approximately an order of magnitude difference between them. The service times of short processes were nearly equal, while the long processes appeared to have exponential service times. It does not appear that these processes were from parallel, distributable programs.

Citing Hinz,‡ Rommel says that processes derived from parallel programs usually result in either equal service times or extremely different service times. He gives examples of several types of scientific

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

‡ The cited reference is to a paper by D. Hinz in the IEEE DMCC5 Conference, Charlestown, S.C., April 1990. I have been unable to locate the proceedings of this conference, but a paper by Didier Hinz with exactly the same title published in Acta Informatica [Hin92], described elsewhere in this chapter, does not contain this information.

applications which can be decomposed into parallel programs, explaining which are likely to result in equal and unequal processes.

Ghosal et al. [GTS90] have examined the behaviour of a number of matrix multiplication and sorting programs on a MIMD transputer net consisting of 16 transputers, each with four links to other transputers, and on a SIMD Connection Machine model CM-2 with 16K processors. For the matrix multiplication experiments run on the transputers, they report, among other results, A_{comm} and A_{comp}, the average number of processors that are communicating and computing, respectively. For the six experiments for which A_{comm} and A_{comp} are reported, the ratio A_{comp}/A_{comm} ranges from 0.67 to 36.0, with mean 7.40 and standard deviation 12.83. As the number of processors increases, the computation time goes down and the communication time goes up; the total time is the sum of the two, so where the two curves cross is the minimum total time. For the transputer net, computation time falls very sharply as the number of processors increases, while the communication time rises moderately quickly; in the figures they show, the curves cross at about four to six processors. With small numbers of processors, as in our experiments, the computation time is much larger than the communication time. For the Connection Machine, the cost of communication is much higher, but does not rise much.

2-9. Summary

In this chapter, we have briefly reviewed the literature on process assignment and multiprocessor scheduling. Much of the work surveyed is highly theoretical and solves only very simple or restricted cases. Other work concentrates on short-running modules which are invoked many times, on scheduling several jobs in a multiprogramming environment, or on situations in which the partitioning can be done as part of the scheduling. There is very little work pertaining to moderately small numbers of moderately long-running processes, where the partitioning of work into processes has already been done and process assignment is the only decision to be made, and where minimizing response time is the objective. Accordingly, it is in this area that we have performed our study. The following chapters report on that study.