An Evaluation of Priority Load Sharing

Shashi K. Purohit, Derek L. Eager, Richard B. Bunt
Department of Computational Science,
University of Saskatchewan,
Saskatoon, Canada S7N OWO

October 10, 1992

Abstract

Dynamic load sharing is an effective mechanism by which a system moves jobs from congested system nodes to lightly loaded nodes. Policies for dynamic load sharing have been applied in distributed systems and large-scale multi-computers, where the general goal is to keep all the processors as busy as possible, since this promises to reduce the average response time of jobs and make optimal use of system resources. Additional performance improvements can result, however, if load sharing policies attempt to keep all the processors not only busy, but busy doing the most productive work.

Priority load sharing refers to this new type of load sharing. The paper presents and analyzes the fundamental issues underlying priority load sharing. The potential of priority load sharing and its dependence on workload characteristics are determined. It is found that load sharing potential can be quite high for certain types of workloads, and that this potential is higher for more broadly focussed workloads. The appropriate level of complexity for priority load sharing policies is studied and it is found that, policies that exchange state information prior to load exchanges can outperform policies that don't exchange state information.

1 Introduction

Load sharing can be defined as a technique by which a system with multiple processing units attempts to improve overall system performance by moving work from congested system processors to lightly loaded processors. The movement of work between the different processors is aimed at utilizing the entire processing power of the system more effectively. The increased use of distributed and shared memory mutiprocessors, and of locally distributed systems, has generated considerable interest in load sharing.

Many different mechanisms for load sharing have been proposed in the literature. Most of this work [2, 3, 4, 5, 6, 9, 10, 12, 17, 18] has considered load sharing in locally distributed systems consisting of a number of hosts (workstations or possibly larger computer systems) interconnected by a communication network and served by a common operating system. Some load sharing research [7, 11, 15] has also concerned large-scale multi-computers consisting of a number of processor-memory pairs and a low latency interconnection network. Policies for load sharing have ranged from static ones that do not react to the instantaneous system state (load on the processors), to adaptive (also termed dynamic) load sharing policies that do react to the instantaneous system state. Load sharing decisions in static policies are based on average system behaviour, making the policies quite simple. Adaptive load sharing policies are more complex because they require a mechanism by which current system state information is collected. Previous load sharing research has shown, however, that adaptive load sharing policies are almost always more effective (yield better system performance) than static policies. In this paper only adaptive load sharing policies are considered.

In distributed systems, typically the units of work to be considered by a load sharing policy are sequential jobs. Load sharing may be accomplished using a *transfer policy*, which determines whether to process a job locally or remotely, and a *placement policy*, which decides where (to which processor) to send a job for processing when a transfer is necessary. In large-scale multi-computers, similar policies may be used, but the units of work are the tasks that result from splitting a single application (perhaps dynamically) into separate threads of control.

This research is concerned with load sharing in a third type of environment quite distinct from those considered before. Parallel processing on a local area network based distributed system, or a small or medium scale shared-memory multiprocessor, is considered. It is assumed that communication between all pairs of processors is equally costly. Of interest is the parallel processing on such a system of a parallel application whose tasks have an associated priority ordering. Each processor executes tasks in its local queue in order of their priority. In such a context, a load sharing mechanism should not just smooth out the processor loads, but should also distribute the workload among the processors in such a way that the profile of tasks taken up for execution by the system follows an order as close as possible to a global priority order. The goals of load sharing are thus two-fold: keep all the processors busy (utilize the processing power of the entire system), and keep the highest priority work in execution.

In such a "prioritized" environment a "load imbalance" can be defined to exist whenever a processor has more than one task that belongs to the top N tasks in priority ranking, where N is the number of processors in the system. A state of perfect balance exists if, given P tasks in the system that are ordered in non-increasing priority order, task i is assigned to processor $i \bmod N$ (a "spiral" arrangement).

Combinatorial search is an example of a domain in which a prioritization of tasks is useful. In this domain, a task represents the work of "expanding" (finding derivatives of) a candidate problem solution. It is possible to associate "priorities" with the various tasks that indicate how important or promising each task is in leading towards the goal or optimal solution. There are at least two advantages of working in global priority order. First, it may lead to faster recognition of a "good" solution to the problem. This solution may or may not be the optimal solution to the problem; however, the speed of reaching an acceptable solution to a problem is sometimes considered more important than the optimality of the solution. Second, it is sometimes possible not to perform some tasks without affecting the quality of the solution at all. In either case, execution is "non-work-conserving" since a prioritized execution may result in less total work being done.

A number of parallel algorithms have been written that exploit the "non-work-conserving" property of search problems and that have shown promising results [1, 8, 13, 16]. Marsland and Popowich [13] proposed and implemented a parallel alpha-beta search algorithm called "principal variation splitting" in which certain tasks in the search are evaluated in preference to others. This prioritization of tasks helped provide good search bounds (alpha and beta cutoffs), thereby eliminating the search of large portions of the search space. Their algorithm was implemented on a locally distributed system in which the processors were interconnected (at least logically) in the form of a tree. Almquist et al. [1] proposed a parallel alpha-beta search algorithm on a shared memory multiprocessor that explicitly employed task priorities to schedule work among the system processors. The algorithm showed significant speedups, and the total number of tasks evaluated during the search was much less than in the case where priorities were not employed. Wah and Ma [16] proposed a parallel branch and bound algorithm in which the most promising tasks are evaluated in preference to other tasks. They employed a specific type of hardware architecture; a uni-directional ring network, to achieve this. Kale and Saletore [8] have also emphasized the need to evaluate tasks in search in priority order.

The success of these parallel algorithms motivates the independent study of priority load sharing. It is expected that the results from such a study will be generally applicable to many problem domains, rather than being constrained to particular algorithms or to combinatorial search problems.

The general goal of this paper is to present and analyze the fundamental issues underlying load sharing in a prioritized task environment. It should be noted that the introduction of priorities requires substantial modifications to existing load sharing concepts. Specifically, the following questions are studied:

- what potential performance improvements does load sharing in a prioritized task environment offer, and how does this potential depend upon workload characteristics ?
- what kinds of policies can be used to perform load sharing in a prioritized environment effectively, and what is their appropriate level of complexity ?

Results obtained from simulations of priority load sharing policies employing synthetic workloads (generated using a model for prioritized workloads) show that for certain types of prioritized workloads substantial performance gains can result from load sharing. For certain other workload types, priority load sharing may hold little or no promise. The notion of "focus", introduced in Section 3 of the paper, is particularly useful for describing and characterizing prioritized workloads. Workloads that are very narrowly focussed, and workloads that are very broadly focussed, hold little promise for priority load sharing. Only workloads between these two extremes hold promise for load sharing. Among these intermediate workloads, a workload that is more broadly focussed has a higher promise for load sharing. The performance of priority load sharing policies that exchange state information prior to exchange of tasks was found to be superior to that of policies that don't exchange state information. Other issues that impact the complexity of priority load sharing policies are identified and raise possibilities for future work in this area.

The rest of the paper is organized as follows. Section 2 discusses some of the distinguishing characteristics of priority load sharing and some of the major issues. Section 3 describes a model for prioritized workloads and Section 4 is devoted to the analysis of priority load sharing. Finally, Section 5 presents the conclusions and directions for future research in this area.

2 Characteristics and Issues

There are two major differences between priority load sharing and conventional load sharing. First, processor queue lengths (or utilizations or other similar load measures) cannot be used in the priority load sharing context since the very notion of load imbalance is different. As described earlier, in priority load sharing a load imbalance exists whenever a processor has more than one task in the top N tasks, as ranked in priority order (where N is the number of processors). The notion of "load" is now much more "relative"— a processor can determine its "load" only through determination of the priorities of its tasks in comparison to those of other processors. A processor that is "heavily loaded" is one with many tasks with relatively high priorities. In contrast, in conventional load sharing the notion of load is much more absolute— an idle processor is definitely underloaded, and a processor with three or four jobs in execution is (in most environments) definitely loaded. Gathering information regarding the priorities of tasks at remote nodes could entail significant overhead costs.

The second major difference from conventional load sharing is that major load imbalances could develop very quickly from states of good load balance, even during a single task processing time, if that task happened to cause many new tasks to be generated, all of higher priority than existing tasks. Thus it appears that continually maintaining good load balance would require exchanges of task priority information after every set of new tasks is generated. In all probability this will be infeasible. Perhaps maintaining a sub-optimal load balance would be feasible; it is unclear, however, what degree of sub-optimality would be acceptable. A sub-optimal load balance is not only sub-optimal, but also has the extra overhead of communicating load sharing information to achieve that load balance.

These considerations suggest that priority load sharing policies might have to be more complex than their non-priority counterparts. Thus the appropriate level of complexity of a priority load sharing policy is an important issue that needs careful consideration.

Perhaps the most fundamental issue in this respect is whether the policy requires the exchange of state information in making load sharing decisions. In the context of priority load sharing, such state information could include the top two task priorities at a processor; if a remote processor has two or more tasks that are higher priority than all the local tasks, a task can be transferred from the remote processor to the local processor.

Policies that do not exchange state information may perform poorly, because they may transfer many tasks unfruitfully, resulting in high overhead. In policies that rely on state information, a processor can acquire this information through use of a central "load sharing server", or by "probing" a subset of the other processors in the system. The replies to the probes contain information on task priorities at the remote nodes, and can be used to make more selective task transfer decisions.

A processor may probe a small subset of the system processors, or it may probe all the processors in the system. Another issue is the frequency with which state information is exchanged between processors. High frequency of exchanges may entail significant overhead costs, while a very low frequency may not provide accurate state information. The different possible choices with respect to these issues can be used to derive different policies for priority load sharing.

3 A Model for Prioritized Workloads

In this section, a parametric model for prioritized workloads is presented. There are two main motivations for such a model. First, a parametric model allows the study of priority load sharing under widely varying workload characteristics, under controlled conditions, something that the use of sample real applications would not allow.

Second, available real applications that have prioritized task structures (chess programs, for example) often have application dependent features (e.g., the details of pruning in alpha-beta search) whose treatment would defocus attention from the purpose of this research—the study of general principles concerning priority load sharing.

In order to be able to draw very general conclusions regarding priority load sharing, only principal characteristics of workloads with prioritized task structures that have major impact on priority load sharing are of interest. The workload model should allow exploration of a full range of different behaviours with respect to these principal characteristics.

The primary workload characteristic that can impact the potential for load sharing is the extent to which it is "focussed" [14]. A workload can be considered highly focussed if at any point of time there are only a few very high priority tasks. In this case load sharing would (at best) attempt to concentrate all the processing power in the high priority region, but since there are only a few high priority tasks at a given time instant, and thus little effective parallelism, the potential benefits of load sharing would be low. On the other hand, when the workload is less focussed, there is more potential parallelism, and thus at least some possibility for parallel computing (and load sharing in particular) to offer benefits. At the other extreme, for a completely unfocussed workload in which all tasks are of roughly equal priority, the potential benefits of priority load sharing would be low.

Several different search workloads (those occurring in combinatorial search) were characterized in [14], the general conclusion being that priority evolution in search algorithms may be depth oriented or breadth oriented to varying degrees, leading to varying degrees of focus. A "pure" depth-first priority evolution is one where, at each task expansion step, a "child" of the last task expanded is chosen next for expansion. In a "pure" breadth-first priority evolution, at each task expansion step, a task that belongs to the same "generation" as the task last expanded is picked next for expansion, until this is no longer possible (all tasks of the current generation are exhausted). Realistic strategies fall between these two extremes. It was also found that all search algorithms use the concept of task priority implicitly or explicitly, where task priority indicates the ordering that the algorithm uses to look at tasks. In general priority is a function of two factors:

1. the algorithm "mechanics", i.e., considerations that are specific to the algorithm in use (such as those relating to "pruning").

2. task promise as determined by an evaluation function.

In this work, task priorities are generated as real numbers, with higher priority corresponding to larger values; note that *relative* priority can only be determined by comparison with other tasks.

3.1 Model Construction

A number of issues need to be considered in constructing the workload model. These include:

1. The model should be able to produce workloads which vary across a full spectrum of focussed and unfocussed behaviours.

2. Priority evolution in the model should be as simple as possible. In particular, the priority of a task should be based only upon the priorities of its ancestors (although, as noted above, *relative* priority will depend on that of all other tasks as well).

3. The model should have as few parameters as possible, without restricting the scope of behaviours.

There are two main parts to such a model: the mechanism for generating the tasks that constitute the workload, and the mechanism by which tasks are assigned priorities. The task generation mechanism that was chosen is quite simple: at each task expansion step at a processor the highest priority task is picked for expansion. Expansion of this task produces a fixed number of child tasks; in the experiments whose results are reported in this paper, this number was chosen to be four.

The priority of a task may be quite strongly correlated with the priority of its parent task, or to the priority of some or all of its other ancestors. For example, in game tree searches each move is part of a certain set of moves or game plan. For simplicity, it was assumed that child priority depends only on the priority of the parent task and not directly on that of other ancestors. The priority of the child task may then be either similar to that of the parent, significantly less than that of the parent, or significantly greater than that of the parent. Based on these ideas, and after a number of revisions, the following model was formulated:

$$priority_{child} = priority_{parent} + r + EC * \delta,$$

where EC is a large constant,
δ is 1 with probability E_p and 0 with probability 1 - E_p,
and r is a random number uniformly distributed in the range -L to U, where L $\geq$ 0 and U $\leq$ 1

The last term in this expression is motivated by the fact that, in the domain of combinatorial search, often most candidate solutions to a problem are similarly mediocre, but occasionally a very promising one will be found.

The parameters L and U, the lower and upper limits of the random number r, and the parameter E_p, influence the nature of the workload that is produced. First, consider the effect of E_p, with L and U held constant. Whenever δ equals 1, there is a large increase in priority, the magnitude of which depends upon the value of EC. If these increases are relatively infrequent then most of the time most task priorities are similarly mediocre, but occasionally a high priority task is present. Priority evolution is thus quite breadth-first, the resulting workload being quite broadly focussed. On the other hand, if δ equals 1 quite frequently, then priority evolution is more depth-first, producing narrowly focussed workloads.

Second, consider the effect of U and L with E_p held constant. If U $\gg$ L then most of the time r will be positive, and the deviation from the parent priority will be positive. Child priorities will thus in general be greater than their corresponding parent priorities, producing workloads that are more depth bound (narrowly focussed). In the reverse situation (U $\ll$ L), child priorities will in general be lower than their corresponding parent priorities, producing workloads that are more breadth bound (broadly focussed). Other ranges of U and L will produce some mix of depth, breadth, and backtracking behaviour.

When U and L are comparable, increasing U (for a fixed L) will cause the depth characteristic to increase, and decreasing it will cause it to decrease. The explanation for this is straightforward: as U increases there will be more positive deviations than negative deviations, and so the workload will become more depth bound; the reverse happens when U is decreased. Similar variations in the workload nature can be realized by varying L, and thus only one of these two parameters need be varied, while the other is kept constant.

In summary, although the model is characterized by the parameters L, U, and E_p, L (or U) can be held constant at appropriate values, while either E_p or U (or L) or both can be varied to produce the required workload types. Experiments with the model [14] revealed that the model can produce a full range of behaviours, from very broad, unfocussed behaviour to very narrowly focussed behaviour.

4 Analysis of Priority Load Sharing

This section addresses the two main questions of interest, namely,

- what potential performance improvements does load sharing offer in a prioritized task environment, and how does this potential depend upon workload characteristics ?
- what kinds of policies can be used to perform load sharing in a prioritized task environment effectively, and what is their appropriate level of complexity ?

A comparison of the performance of "perfect load sharing" with "no load sharing", as the system workload is varied, is used to answer the first question. In perfect load sharing there is no cost associated with exchanging state information or transferring tasks between the system processors, and thus it provides an upper bound to load sharing performance. The second question is addressed by comparing the performance of two policies, one that is based on state information exchange between processors and one that does not exchange state information. Since information exchange is the most fundamental issue concerning the complexity of policies, such a comparison is central to the determination of the appropriate level of complexity of priority load sharing policies.

The rest of this section is organized as follows. Section 4.1 describes the metrics that will be used to evaluate load sharing performance. Section 4.2 considers the promise of priority load sharing and its dependence on workload characteristics. It includes a description of the experiments that were performed, and the conclusions from the experimental study. Section 4.3 is devoted to the complexity issue. The specific policies that are studied and the simulation model used to evaluate their performance are described, followed by descriptions of the experiments performed and their results.

4.1 Metrics for Evaluating Load Sharing Performance

Two measures "progress rate" and "average rank", are used to evaluate load sharing performance, with the former being the principal measure that is employed.

The rank of a processor is defined as the rank of its highest priority task within a listing of all system tasks ordered by task priorities. "Average rank" is defined as the average of the ranks of the system processors, averaged over time. If the average rank is low, the load is well balanced in the sense that the highest priority tasks are spread out in a balanced way among all the processors. On the contrary, if the average rank is high then some processors are working on low priority tasks and the load is poorly balanced. Under perfect load sharing or global scheduling with N processors in the system, the average rank is *(N+1)/2*.

"Progress rate" is defined as the rate of change with respect to time of the maximum priority of the tasks currently in the system. This definition is motivated as follows. The goal of load sharing is to ensure that the highest priority tasks are kept in execution, in preference to lower priority tasks. The success of load sharing in achieving this goal is measured well by average rank. Average rank, however, does not capture the benefit to the application of achieving this goal; i.e., of having as many of its high priority tasks in execution as possible. It may be the case, for example, that in some particular application, progress towards a problem solution is determined only by whether the top two or three of the highest priority tasks can be kept in execution, rather than whether all N can be, where N is the number of processors in the system.

Progress rate attempts to measure directly the rate of progress towards a problem solution, by using the absolute priority values produced by the workload model of Section 3 as indicators of progress. Since these priority values are meaningful only in relative terms, all results are normalized by dividing the progress rate by the progress rate in the uniprocessor case. This yields values that can be interpreted as "speedups" with respect to application progress rate and thus presumably required execution time as well.

4.2 Promise of Load Sharing

A comparison of the performance of perfect load sharing to that of no load sharing reveals the promise of load sharing and its dependence on workload characteristics. Only progress rate is considered in this comparison, as the average rank results for perfect load sharing are known a priori, and those for no load sharing do not yield any useful insights.

Prior to discussing the experiments that were performed, recall from Section 3 that workloads that are somewhat broadly focussed have a higher potential for load sharing than do narrowly focussed workloads. In fact, a completely focussed workload holds no promise for load sharing at all. Task priorities matter in this case, but there aren't enough high priority tasks to load share. In a completely unfocussed workload, priorities are irrelevant, so it does not matter if tasks are load shared or if each processor is allowed to work in isolation. Here again there is no promise for load sharing. Only for the intermediate workloads might load sharing hold some promise, and therefore these are the types of workloads that should be considered in the experiments.

Simulation experiments were performed for perfect load sharing and for no load sharing, in a system with ten processors. The results are accurate to within 5%, at a confidence level of 95%. The parameter E_p was varied over the set of values (0.001, 0.005, 0.01, 0.02, 0.04, 0.08, 0.16, 0.32), while EC and U were held constant at 1000 and -0.008 respectively.

As shown in Figure 1(a), normalized progress rate under load sharing decreases as E_p is increased. An increase in E_p corresponds to an increase in the depth characteristic (focussing characteristic) of the workload; this is seen in Figure 2, which shows the depth percentage, defined as the percentage of task expansions in which a child task of the task last expanded is chosen next for expansion, in these parameter ranges. The promise of load sharing, as

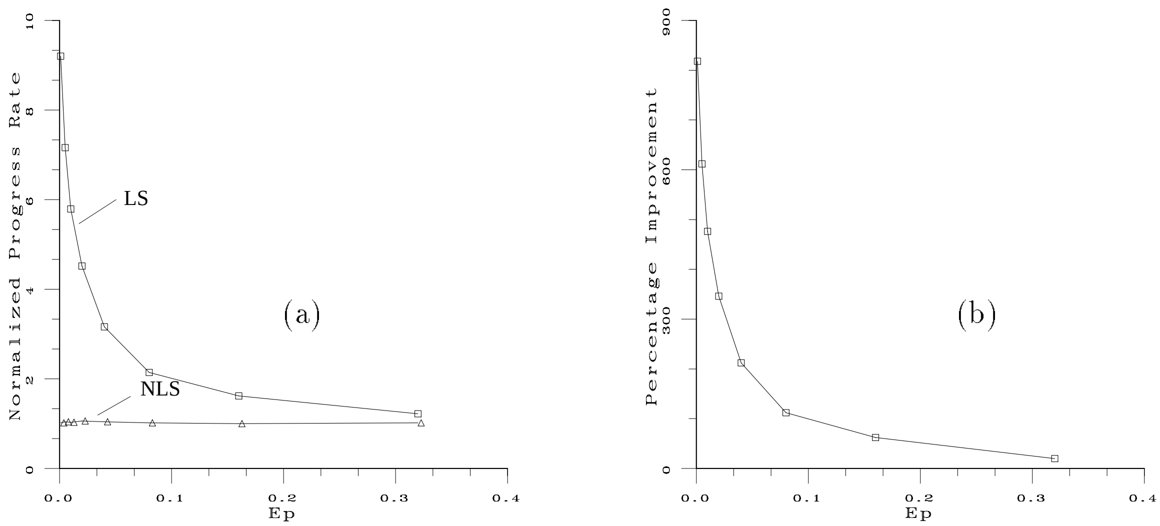


Figure 1: (a) Normalized Progress Rate vs. E_p for Load Sharing (LS) and No Load Sharing (NLS); (b) Percentage Improvement as a Function of E_p.

opposed to no load sharing, can be described in terms of the percentage improvement due to load sharing. Figure 1(b) shows the percentage improvement as a function of E_p. From this Figure it can be seen that load sharing holds a higher promise (a percentage improvement up to 820%) for workloads that are less depth bound, or more broadly focussed. Load sharing has virtually no performance benefits for strongly depth bound workloads.

These observations lead to two conclusions:

- large performance improvements are possible when load sharing is employed in a prioritized task environment.
- the improvement in performance depends upon the nature of the workload— in general, a workload that is more broadly focussed, in comparison to another, has a higher promise for load sharing.

4.3 The Complexity Issue

The performance of a load sharing policy that exchanges state information prior to load exchanges is compared with the performance of a policy that doesn't exchange state information, and with perfect load sharing and no load sharing, to arrive at some basic conclusions regarding the appropriate complexity for load sharing policies. The comparison is done principally for a workload that holds a high promise for load sharing, as identified in the previous section. Results are shown for the following parameter values: L = 0.04, U = .01, E_p = 0.001, and EC = 1000.

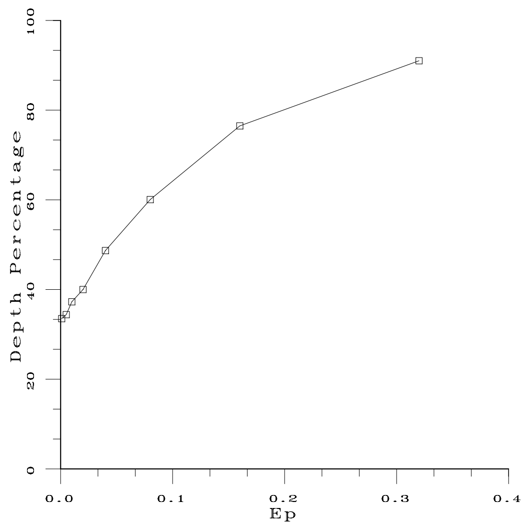


Figure 2: Depth Percentage in the Range of E_p Values Considered.

First the policies considered for implementation and the overhead costs modelled are discussed. Message passing terminology is employed, although this does not rule out the possible use of a shared memory programming model in such a context. This is followed by a discussion of the principal performance comparisons that were made using progress rate and average rank as performance metrics. Finally, the sensitivity of the information exchange based policy to policy parameters is discussed and some conclusions are drawn.

4.3.1 Overhead Costs and Policies

A policy called Parallel Random Distribution (PRD) was chosen to represent policies that don't exchange state information prior to load exchanges. In the PRD policy, during each step each processor expands a task and distributes the child tasks to randomly selected processors. PRD then looks at its own message queue and picks up any tasks that may have arrived from other processors. PRD may achieve a good load distribution, but may suffer from high overheads since it distributes load indiscriminately. This may degrade performance significantly when the task transfer cost is high.

A policy called INFO_EX was chosen to represent policies based on exchange of state information. INFO_EX periodically "probes" a small subset of the system processors, chosen randomly, to obtain state information. Based on this information INFO_EX then makes the necessary load sharing decisions. Settable parameters of the policy are "period" and "probe limit". Period is the frequency with which "probes" are sent out (between probes a processor does local processing only), as measured by the number of task expansion steps between searching for higher priority work through probing. The probe limit is the number of processors that a processor probes whenever it decides to search for higher priority work.

A processor that receives a probe compares the priority in the probe message (which is that of the highest priority task at the sending processor) with the priority of its second highest priority task, to check if it has two or more tasks that are higher priority than those at the sending processor. If the second highest priority is less than the received priority, INFO_EX sends a "probe-ack-reject" (probe acknowledgment rejection message); otherwise it dequeues the relevant task from its local queue and sends a "probe-ack-accept" message, which includes the priority of the second highest priority task. It is necessary to dequeue the task so that it is not considered for expansion locally, and is also not considered for an exchange with some other processor whose probe arrived at the same time. The source processor, the one that had sent out the probes, waits until it has replies from all probed processors. It then sends an "accept-exchange" to that processor that has the highest priority task to share, among those probed, and to the rest it sends a "reject-exchange" message. On receiving an "accept-exchange" message a processor sends the dequeued task to the respective processor. If a "reject-exchange" message is received instead, the processor enqueues the dequeued task.

Two kinds of overhead costs are modelled (recall that the cost of communication is assumed identical between all pairs of processor):

- processor cost of acquiring state information. This overhead is incurred by both the sending and the receiving processor of each protocol message.

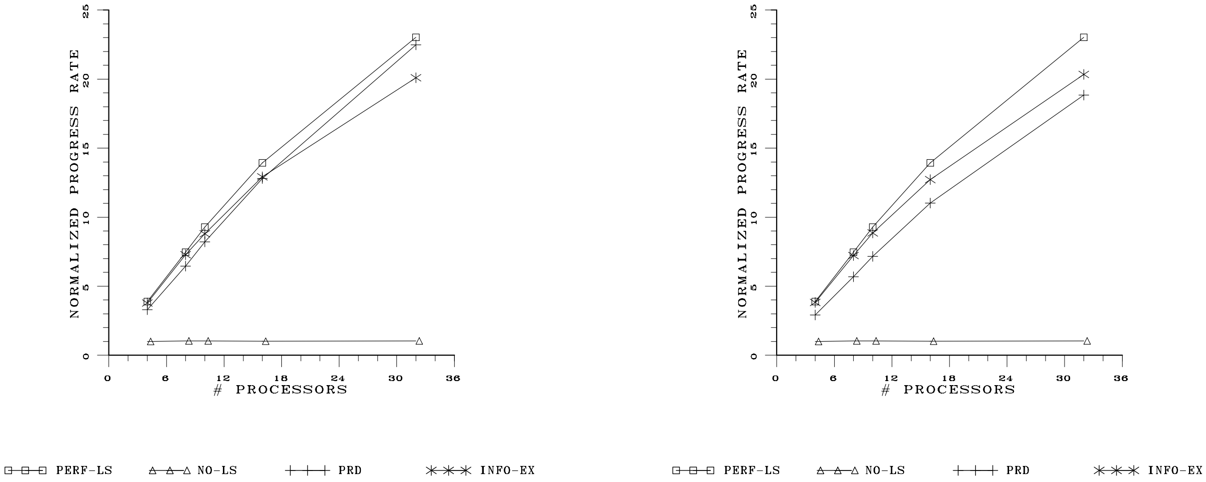
- processor cost of transferring a task from one processor to another processor. This overhead is incurred by both the sending and the receiving processor. This cost is assumed to be significantly higher than the state exchange cost.

4.3.2 Principal Performance Comparisons

Progress rate was measured for the PRD policy, the INFO_EX policy, perfect load sharing (PERF_LS), and no load sharing (NO_LS), for a number of different task transfer and state exchange costs, and number of processors in the simulated system. Four levels (0.04, 0.08, 0.16, 0.32) were chosen for the task transfer cost and three (0.002, 0.008, 0.032) for the state exchange overhead. (These values are in units of the task processing time.) The number of processors was varied over five levels (4, 8, 10, 16, and 32). No other parameters were separately varied.

For the INFO_EX policy, the following settings were used for the policy parameters. Period, the frequency with which a processor sends out probes, was set to 2. This means that after two task expansion steps, a processor sends out probes to attempt to find a processor from which a higher priority task can be obtained. The probe-limit parameter was set at N/2, where N is the number of processors in the system.

The full experiment consisted of a total of 60 (4x3x5) simulation runs using the INFO_EX policy, a total of 20 (4x5) simulation runs for PRD, and 5 simulation runs each for perfect load sharing and no load sharing. As before, results are accurate to within 5%, at a confidence level of 95%. Some of the key results are shown in Figures 3 to 6, which show the variation of



(a) (b)

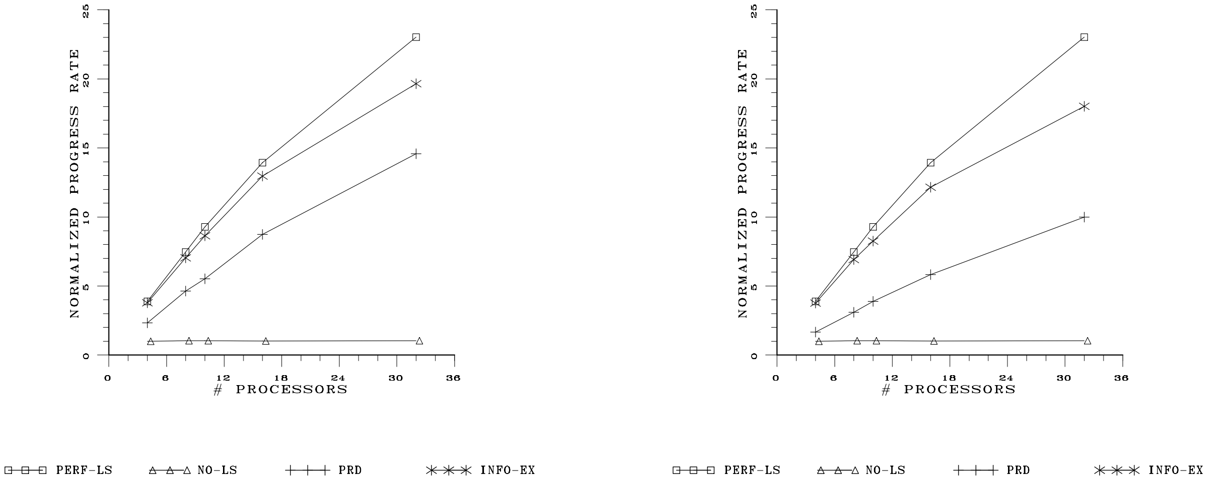
Figure 3: Normalized Progress Rate vs. Number of Processors (task transfer cost = 0.04 in (a) and 0.08 in (b); state exchange cost = 0.002).

progress rate (or speedup) with the number of processors in the system for the four policies under consideration. The curve for perfect load sharing is the upper limit on load sharing performance. The maximum speedup observed with 32 processors for the workload under consideration is roughly 24. With no load sharing, no speedup is obtained, regardless of the number of processors in the system.

The relative performance of PRD and INFO_EX changes as the task transfer cost varies in Figures 3(a), 3(b), 4(a) and 4(b). In Figure 3(a), PRD performs better than INFO_EX, but only for large systems. For small to medium sized systems INFO_EX is the better performer. When the task transfer cost is increased by 100% (see Figure 3(b)), INFO_EX consistently outperforms PRD. For higher task transfer costs (Figures 4(a) and 4(b)) the difference in performance between INFO_EX and PRD increases. INFO_EX is clearly much more robust with respect to task transfer costs.

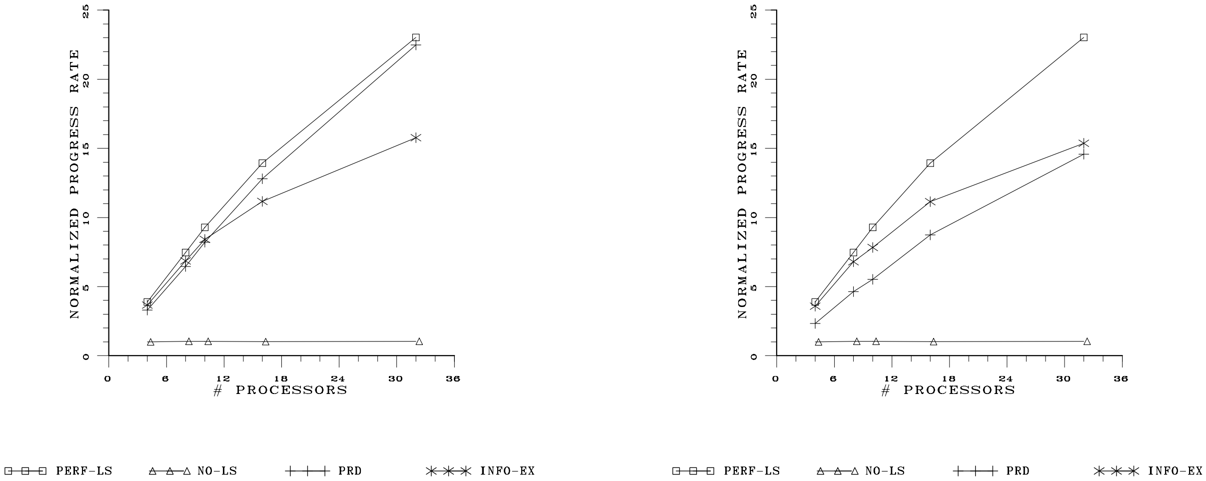
When the state exchange overhead becomes almost comparable to the task transfer cost, as when the state exchange overhead = .032 and the task transfer cost = 0.04 (Figure 5(a)), the performance of INFO_EX degrades significantly. As the task transfer cost is increased, keeping the state exchange overhead fixed at 0.032, once again INFO_EX emerges as the better performer, (Figure 5(b) and 6).

A comparison of the performance of PRD with that of perfect load sharing shows that PRD and PERF_LS perform comparably when the task transfer cost is low (0.04, Figure 3(a)). For higher task transfer costs (.08, .16, .32), however, PRD performs poorly—there is a large difference in performance compared to PERF_LS. INFO_EX does not closely approach the performance of PERF_LS for any combination of overhead costs considered;



(a) (b)

Figure 4: Normalized Progress Rate vs. Number of Processors (task transfer cost = 0.16 in (a) and 0.32 in (b); state exchange cost = 0.002).



(a) (b)

Figure 5: Normalized Progress Rate vs. Number of Processors (task transfer cost = 0.04 in (a) and 0.16 in (b); state exchange cost = 0.032).

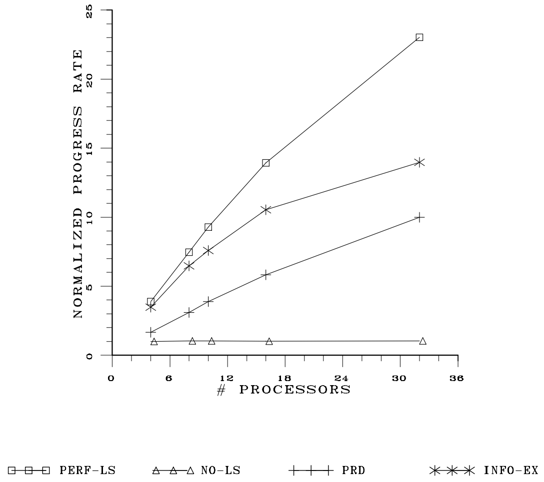


Figure 6: Normalized Progress Rate vs. Number of Processors (task transfer cost = 0.32 and state exchange cost = 0.032).

its performance is slightly inferior to PRD for the case when PRD approaches PERF_LS. INFO_EX is fairly robust, however, as its performance does not degrade excessively as overhead costs are increased.

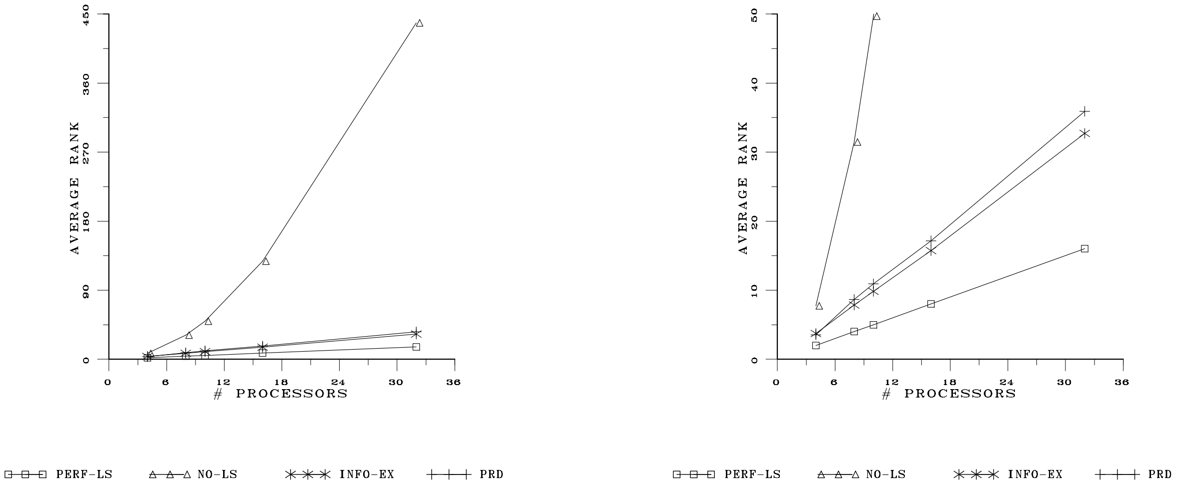
Average rank was measured for PRD, INFO_EX, and no load sharing as the number of processors in the system is varied. The workload used in these experiments was the same as in the progress rate experiments. The number of processors was varied over five levels (4, 8, 10, 16 and 32). Note that the average rank for perfect load sharing is known *a priori*.

The results of these experiments are shown in Figure 7(a) and 7(b). While Figure 7(a) covers the full range of observed values for average rank, Figure 7(b) highlights the curves of prime interest: PRD, INFO_EX and PERF_LS. The main observation is that although both PRD and INFO_EX perform well compared to no load sharing, they still leave a lot of room for improvement. INFO_EX performs better than PRD for all the system sizes examined, although perhaps surprisingly, the difference is not substantial.

4.3.3 Sensitivity of INFO_EX to Policy Parameters

In this section, the sensitivity of the results described above for the INFO_EX policy to its parameters of period (controlling the frequency with which load sharing information is exchanged) and probe limit (the number of processors that are probed prior to a load exchange) is studied. In the experiments in the previous section, period was 2 and probe limit was N/2.

To assess sensitivity to probe limit, an experiment was performed in which normalized



(a) (b)

Figure 7: Average Rank as a Function of Number of Processors (period = 2 and probe limit = N/2, where N = number of processors in the system).

progress rate was measured for different values of the probe limit with a system size of 32 processors. The levels chosen for the probe limit were 3, 6, 12, and 24. Note that the normalized progress rate for a probe limit of 16 was available from the experiments in the previous section. The other variable parameter in the experiment was the state exchange overhead. Task transfer overhead was held constant. Normalized progress rates are shown graphically in Figure 8. Each curve corresponds to a different state exchange overhead.

Consider the curve obtained for a state exchange overhead of 0.002. The curve shows an initial increase in the normalized progress rate as the probe limit is increased, but thereafter it tends to saturate beyond a probe limit of roughly 5. As the probe limit is increased further, there is a very slight drop in the normalized progress rate due to the increase in the overhead cost (state exchange overhead).

The curve labeled 0.008 shows behaviour similar to that mentioned above. However, the curve labeled 0.032 shows a sharper drop in normalized progress rate as the probe limit is increased. This can be explained by the fact that now the overhead of probing is very high. A general conclusion, then, is that when the state exchange overhead is not very high, speedup is quite insensitive to the probe limit beyond a certain "threshold" probe limit (it should not be very low compared to the number of processors in the system). The value used in the experiments in the previous section (N/2) is a very conservative value in the sense that it is higher than necessary, at least for large numbers of processors. In contrast, when the state exchange overhead is high, a smaller probe limit should be used.

Consider now sensitivity to period. An experiment was performed in which normalized progress rate was measured for different values of period, with a system size of 32 processors.

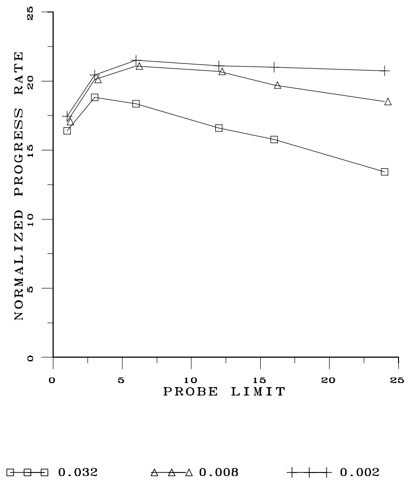


Figure 8: Sensitivity of Normalized Progress Rate to Probe Limit.

The levels that were chosen for period were 1, 4, 8, and 16. Note that the normalized progress rate for a period of 2 is already known from the previous section. The other variable of the experiment was the state exchange overhead. Task transfer overhead was held constant.

The results are shown graphically in Figure 9. The curve corresponding to a state exchange overhead of 0.002 shows a continuous decline in the normalized progress rate as period is increased. This leads to the rather intuitive conclusion that when the overhead of exchanging state information is low, the higher the rate of probes, the better the performance. The curve corresponding to a state exchange overhead of 0.008, however, does not lead to exactly the same conclusion. Normalized progress rate increases when the period is increased from 1 to 2. Thereafter, normalized progress rate decreases continuously. A similar observation can be made from the curve corresponding to a state exchange overhead of 0.032, but now performance with a period as high as 8 is better than the performance with a period of 1.

The general conclusion from these results is that the performance of INFO_EX is quite sensitive to period. The choice of 2 as the value for period in the experiments of Section 4.3 is justifiable since, from the above curves, it can be concluded that this value yields quite good performance on average. A more complex load sharing policy may attempt to choose the period parameter adaptively.

5 Conclusions and Future Research

The study of the fundamental issues underlying priority load sharing provided valuable insights into the nature of prioritized workloads, and the nature of the policies that can be used

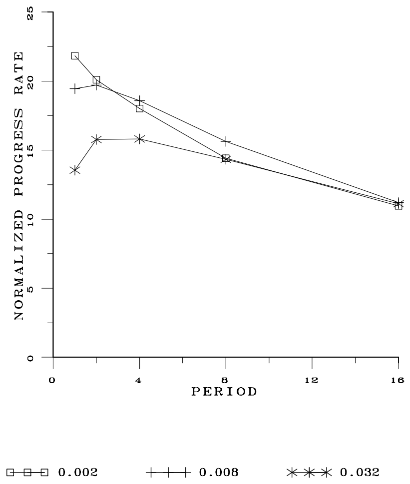


Figure 9: Sensitivity of Normalized Progress Rate to Period.

to perform load sharing effectively in a prioritized task environment. “Focus” was found to be a useful way to characterize prioritized workloads from a load sharing perspective. Policies for load sharing should, in general, collect and employ state information prior to any redistribution of workload. Many aspects of load sharing policies can be chosen independently of application dependent considerations, since the choice of such policies can be based in part on a number of high level issues rather than low level application specific issues. It is generally felt that many applications would benefit from priority load sharing, since they would be quite moderately focussed.

The priority load sharing policy INFO_EX studied in this paper is a server-initiated policy in which processors attempt to find other processors from which they can obtain work. Experiments to determine the variation of load sharing promise with the nature of the workload have shown that load sharing holds higher promise for less focussed workloads. In such a case, most probes in server-initiated policies would be failures (no actual tasks are transferred). With a source-initiated policy, probes to distribute work might be sent out only when a processor detects a sudden, large increase in the priority of its highest priority tasks, in which case wasted probes might be relatively very few. (Of course, it may be difficult to recognize such situations.) Thus source-initiated policies may perform better than server-initiated policies for less focussed workloads. A study of source-initiated policies, with all the other policy considerations being the same, can be a direct extension of this work.

6 Acknowledgments

Funding for this research was provided, in part, by the Natural Sciences and Engineering Research Council of Canada (NSERC), through Research Grants OGP0000264 and OGP0003707 and Strategic Grant STR0045250.

References

[1] Almquist K., McEnzie N. and Sloan K., "Parallel Algorithms for Searching Game Trees," Unpublished Manuscript, Department of Computer Science and Engineering, University of Washington, Seattle, October 1989.

[2] Barak A. and Shiloh A., "A Distributed Load Balancing Policy for a Multicomputer," *Software Practice and Experience,* Vol. 15, No. 9, September 1985, pp. 901-913.

[3] Bryant R. and Finkel R. A., "A Stable Distributed Scheduling Algorithm," *Proceedings 2nd International Conference on Distributed Computing Systems,* May 1981, pp. 314-323.

[4] Casavant T. L. and Kuhl J. G., "Analysis of three Dynamic Distributed Load Balancing Strategies with Varying Global Information Requirements," *Proceedings IEEE 7th International Conference on Distributed Computing Systems,* August 1987, pp. 185-192.

[5] Eager D. L., Lazowska E. D. and Zahorjan J., "A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing," *Performance Evaluation,* Vol. 6, No. 1, April 1986, pp. 53-68.

[6] Eager D. L., Lazowska E. D. and Zahorjan J., "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering,* Vol. SE-12, No. 5, May 1986, pp. 662-675.

[7] Kale L. V., "Comparing the Performance of Two Dynamic Load Distribution Methods," Report No. UIUCDCS-R-87-1387, Dept. of Computer Science, Univ. of Illinois, November 1987.

[8] Kale L. V. and Saletore V., "Parallel State-Space Search for a First Solution," Report No. UIUCDCS-R-89-1549, Dept. of Computer Science, University of Illinois, October 1989.

[9] Krueger P. and Finkel R. A., "An Adaptive Load Balancing Algorithm for a Multicomputer," Tech. Report 539, Dept. of Computer Science, Univ. of Wisconsin Madison, April 1984.

[10] Leland W. E. and Ott T. J., "Load Balancing Heuristics and Process Behavior," *Proceedings ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems,* May 1986, pp. 54-69.

[11] Lin F. C. H. and Keller R. M., "Gradient Model: A Demand-Driven Load Balancing Scheme," *Proceedings IEEE 6th International Conference on Distributed Computing Systems,* May 1986, pp. 329-336.

[12] Livny M. and Melman H., "Load Balancing in Homogeneous Broadcast Distributed Systems," *Proceedings ACM Computer Network Performance Symposium,* April 1982, pp. 47-55.

[13] Marsland T. A. and Popowich F., "Parallel Game-Tree Search," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vol. PAMI-7, No. 4, July 1985, pp. 442-452.

[14] Purohit S., "Load Sharing in a Prioritized Task Environment," M.Sc. Thesis, Department of Computational Science, University of Saskatchewan, Saskatoon, Sask, March 1992. (available as Research Report 92-3.)

[15] Shu W. W. and Kale L. V., "A Dynamic Scheduling-Strategy for the Chare Kernel System," *Proceedings of Supercomputing-89,* November 1989, pp. 389-398.

[16] Wah B. W. and Ma Y. W. E., "MANIP- A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems," *IEEE Transactions on Computers,* Vol. C-33, No. 5, May 1984, pp. 377-390.

[17] Wang Y. and Morris R. J. T., "Load Sharing in Distributed Systems" *IEEE Transactions on Computers,* Vol. C-34, No. 3, March 1985, pp. 204-217.

[18] Zhou S., "A Trace-Driven Simulation Study of Dynamic Load Balancing," *IEEE Transactions. on Software Engineering,* Vol. 14, No. 9, September 1988, pp. 1327-1341.