Published by the

Institute of Parallel and Distributed High-Performance Systems (IPVR)

Department for Applications of Parallel and Distributed Systems

Faculty for Computer Science

University of Stuttgart

Breitwiesenstr. 20-22, D-70565 Stuttgart, Germany

Efficiency of Server Task Queueing for Dynamic Load Balancing

Wolfgang Becker, Rainer Pollak

Faculty report No. 1994 / 9

CR subject classification C.2.4, C.4, D.4.8

Wolfgang.Becker@informatik.uni-stuttgart.de
Rainer.Pollak@informatik.uni-stuttgart.de

Efficiency of Server Task Queueing for Dynamic Load Balancing

Abstract

In this paper we investigate optimal points of time for task assignment in dynamic load balancing schemes. Normally final assignment of tasks to server queues is made at the latest possible time. The main reason for a late assignment is, that a dynamic load balancer can use most recent information about system and application state for the decision. In general however, assignment can be done at task arrival time, at the moment when a processor or server becomes idle, or when significant load changes in the system occur. We will elaborate preconditions and circumstances for situations, where it is advantageous to assign tasks earlier than necessary, i.e. to queue them at the servers. We verify the results in an experimental load balancing environment.

Contents

1 Introduction

As distributed computer systems become increasingly popular, resource sharing among a number of computers or processors within a MIMD-computer connected by communication networks becomes practicable and desirable. Sharing of computing resources, e.g. processor time, data and hardware devices is usually assisted by load balancing mechanisms. Load balancing is the process of distributing and redistributing the workload submitted to a network of computers to avoid situations where some of the hosts / processors are overloaded while others are underloaded. Work and data are distributed in a way that exploits all system resources and maximizes overall throughput.

Load balancing strategies may be either static or dynamic, While static strategies yield schedules based on averaged system characteristics and profiles of applications running in the system, dynamic approaches also use information about the current system state and application behavior at run time. Dynamic policies may be further subdivided into centralized and distributed structures and, according to the utilized information, into reactive and predictive load balancing strategies. For a detailed classification of load balancing approaches we refer to [4] or [8], where a hierarchical taxonomy is presented. Distributed schemes usually hold tasks only in local server queues and migrate them at arrival time [9], in situations of load imbalance [1], [6], [11] or at task completion time [13]. So our investigations of final assignment time apply especially to centralized, predictive load balancing schemes. Several recent publications are concerned with central-

ized load balancing approaches in similar environments [3], [5], [7], [10], [12], [15], some of them employing local server queues per processor (e.g. [5]). However, their main objective is comparison of different load distribution strategies in terms of considered load factors, not structural aspects like server task queueing and optimal time points for task assignment. Furthermore there is a lack of proposals, which are validated by actual application measurements.

2 General Considerations

For a load balancer there are several time points to assign tasks to processors or servers. Static schemes commit placement at compilation or batch job start-up time, dynamic non-preemptive schemes usually place tasks at arrival time. Preemptive dynamic load balancing methods are additionally able to correct such assignment decisions by migrating tasks, that are currently executing. Although never explicitly investigated, dynamic schemes are principally able to defer assignment of tasks until the best server could be ascertained and / or until one of the suitable servers becomes ready to receive more tasks. One possibility in client server structures is to delay assignment until any or the best suited server announces that it became idle. Another possible way, and this is what we will examine in this paper, is to assign tasks even before they can be executed by the receiving server, for the server may still have some previously received tasks to process.

For such a strategy to be feasible two requirements must be fulfilled. First, there must be a central task queue for each server class. A server class is a group of servers having identical functionality. The central queues are controlled by the load balancing component, which may arbitrarily select tasks from it for assignment. At each server site there has to be a local task queue to contain assigned tasks. Servers process the tasks in their queue in first come first served order or according to their priorities.

The second precondition requires an accurate definition of load balancing events and actions triggered by them. In section 4 we will mention the events and actions, that our experimental environment offers depending on the actually employed load balancing strategy.

- In most publications there exists an event of task arrival, followed by the action of immediate assignment. In decentralized schemes this means pushing the task to some less loaded server.
- In preemptive schemes task migration is triggered also by the event of significant load imbalance. The lightly loaded processor receives a task to the most heavy loaded neighbor. If a central task queue exists, load changes may also trigger task assignment.
- In receiver initiated schemes each time when a server complets its task, an event occurs. This event is followed by receiving a task from an overloaded neighbor. Centralized approaches assign it the eldest or the best fitting task of the central queue to the underloaded processor or server.

Here we briefly look at design alternatives to enable more flexible task assignment time points. One possible extension are time-outs: at task arrival time the load balancer rates the servers and decides whether to assign the task immediately or to establish a time-out. The time-out specifies, when assignment must be committed at the latest, if no other events yield an assignment decision for this task. Nevertheless we will not use time-outs in this paper, because it is rather complex and not completely understood. Instead, we employ a flexible event - action coupling: no matter, which task, processor or server caused the event, the action may assign any set of tasks to arbitrary servers. Furthermore we expand load balancing, that it may keep track of the tasks currently queued at servers, especially the size of the local task queues. This allows to estimate the load at the server and the time until the server turns back to idle state. If load balancing considers data

access affinity (see section 5), it may also estimate which data records will be present at the server when it has worked off its queue.

Once load balancing is capable of assigning tasks more or less at any moment, it is necessary to investigate the advantages and drawbacks expected from early assignment. The first improvement is the saving of server idle time due to load balancing decision and message passing delays. This is especially true in central load balancing schemes and shared nothing computer architectures, which we are concerned with. Each time a server has completed a task execution and has no further tasks queued locally, it must call the load balancing component and wait until a task arrives. Note that employing many servers per processor at a time helps keeping the cpu busy, but often results in synchronization and task switching overhead. Also it complicates load balancing methods significantly. Our considerations in principle apply to both execution models.

The second advantage expected from early assignment is the reduction of load balancing overhead. Load balancing should try to get rid of tasks residing in central queues, if their currently favored placement most probably will not change anymore. The principle to assign tasks as late as possible is based upon the observation, that the load situation may change distinctively. Another placement becomes more lucrative during the time between the task arrived and the time, when the server starts its execution. To exploit this, however, it is required that all task ratings in the central queue are updated each time a relevant load balancing event occurs. In situations of high parallelism, i.e. when a huge set of executable tasks arrive that cannot be served at once, this will certainly cause much overhead. Therefore, load balancing must be able to estimate the possible load change rates in terms of foreign processor load, task completions and data movements. The stability and predictability of the system and application behavior then enables to assign tasks to servers up to a certain degree in advance, i.e. put them into the local task queue.

A third opportunity arising from proper usage of local server task queues is less obvious, but nevertheless important. Most load balancing concepts cannot exploit knowledge about groups of tasks, whether for reasons of simplicity or avoidance of overhead. To detect the most promising degree of parallelism, however, it requires to know in advance, how many tasks, showing a similar profile or data access pattern, will follow. Then load balancing decides whether to set up further servers and whether it pays off to distribute data and copies among the system. The option of earlier task assignment using local queues provides a rather simple way of controlling the degree of parallelism. The algorithm for early assignment without explicit announcement of task groups to load balancing works as follows: the first few tasks will be assigned to the currently available servers, which have appropriate local access to the participant data. After a short time, however, as their local task queues are growing, they become less attractive for further task assignment. Other servers will be employed in spite of start-up overhead, data movement costs or inferior available processing power. Without server queues, load balancing would always decide to keep the next task until the best server becomes free again, because this is obviously faster than giving it to someone else.

The obvious drawback of local task queues is the potential of inappropriate assignment due to rapidly changing system load or data placement. In addition, there is overhead arising from the explicit consideration of the tasks residing in the local server task queues.

In the following section we will introduce a simple analytic model to illustrate the relevant factors and their correlations concerning task assignment time points.

3 The Analytic Model

The underlying system model, as depicted in the left part of Figure 4, fulfils the requirements explained above, providing central and local task queues. We will not use Markov queues [5] to examine the effects of server task queueing but restrict ourselves to straightforward calculations. Both ways cannot adequately reflect the real behavior and interactions, e.g. differences between load balancing methods. So it is necessary to rely on measurements obtained from a real world environment, that confirm the benefits and limitations of our considerations.

Our analysis bases on the following set of input parameters, which, of course, mutually influence each other. They are estimations about system, application and load balancing behavior during some homogeneous phase of the application run. The first set of parameters can be obtained from measurements:

- *p* (processors) states the number of available processors, which equals the number of servers for our considerations.
- *ml* (message latency) gives the average elapsed time for a message passed between the load balancing component and some server including protocol overhead.
- *tt* (task execution time) is the elapsed time for a single task execution averaged over all tasks arrived in this phase and all employed processors, if executed on a central processor without any parallelism.
- *lba* (load balancing time per task assignment) estimates the average time load balancing spent in reacting on events per assignment of a task. This includes server rating for assignment decisions as well as system load state collection. Each time load sharing examines a task in the central queue this value increases accordingly.
- *squ* (server queue usage) is the average server queue size effectively used by a certain load balancing strategy. This is essentially the parameter with which we vary the task assignment time point as discussed above.

The second part of parameters are estimations that cannot be simply extracted by profiling the system:

- *tts* (relative task execution time skew) is a factor, which gives the maximum increase of task execution time due to processor speed and load differences, references to non-local data, etc.
- *lbrs* (load balancing recognized part of skew) is the portion of the system and application inherent skew observed by the strategy. A simple strategy, for example, would equally distribute tasks to processors although this does not yield maximum throughput.
- *if* (information falsification per queued task) describes the load change and data movement rate, i.e. instability and in-predictability of the system state. So we have a factor how much load balancing benefit is lost per task, because the information used at assignment time grew old while the task is waiting in the server queue. The more tasks in the servers local queue exist, the more load balancing benefit will be lost for these tasks.

Upon this abstract characterization of system, load balancing and application behavior we built a set of equations. They yield some overall performance estimation, expressed in the average number of tasks finished per unit of time, assuming a sufficiently loaded system (i.e. there are mostly enough tasks to work on).

The temporary variables have the following meanings:

- *lbov* gives the average overhead for the load balancing component per task from arrival till the assignment decision,
- *qe* is the probability for a server to encounter an empty local queue after having finished a task.
- *qed* is the time a server remains idle until it receives the next task, provided its local queue is empty.
- *etts* estimates the actually observed relative task execution time skew.
- Finally, *ptt* gives the average elapsed time between two task executions on a processor divided by the effective parallelism.

$$lbov = \frac{p \times lba}{squ} \quad \text{(EQ 1)}$$

$$qe = min(\frac{tts \times lbrs + 1}{squ}, 1) \ , \ qed = qe \times (2 \times ml + lbov) \quad \text{(EQ 2)}$$

$$etts = tts \times (1 - lbrs \times (1 - min(if \times (squ - 1), 1))) \quad \text{(EQ 3)}$$

$$ptt = \frac{tts}{p} \times \left(tt + \frac{tt \times etts}{2} + qed \right) \ , \ throughput = \frac{1}{max(ptt, lbov)} \quad \text{(EQ 4)}$$

From these equations we can derive promising application and system parameter ranges in combination with load balancing methods, for which early task assignment in form of server task queueing is profitable. We chose three different levels of load balancing representing stupid, simple and more advanced strategies. All three have the following parameter settings in common:

- *tt* = 25 msec, rather fine grained tasks in a workstation environment. Note, that non-preemptive load balancing should have fine grained tasks, because load skews caused by long running tasks should not happen.
- *tts* = 5, a task placed on some unsuited processor will run five times as long as on average, *p* = 15, a mid range workstation cluster,
- *ml* = 700 µs, software latency in ethernet based tcp communications.

The other input parameters are empirical; they partially depend on the load balancing strategy used. The first strategy, called Round Robin (RR), simply assigns tasks in round robin fashion. There is no chance to detect some execution time skew, *lbrs* = 0, assignment cost are low, *lba* = 10 µs and, for no run time information is used, it cannot grow old, *if* = 0.

The second competitor, called First Free (FF), always assigns tasks to the first server, which has some space left in its local queue; using round robin like above, if more than one is available. We assume theoretical skew avoidance of 15%, i.e. *lbrs* = 0.15, still low assignment cost, *lba* = 50 µs. The only run time information used are the servers' local queue size, which may be viewed as growing old due to task size and processor speed variations, so we estimate *if* = 0.05.

Finally we model a rather advanced strategy, named Data Locality and Processor Speed (DLPS). It takes into consideration several items: the processor speed and current load, the expected task size, access pattern to global data and current data placement; further the time elapsing for execution of the tasks currently residing in the local queues. It uses complex formulas to rate the server applicabilities for each task under the current situation. So it is able to swallow 60% of the execution time skew, *i.e. lbrs* = 0.6, to the debit of assignment cost *lba* = 2 ms. There is a non-negligible factor of information growing old while an assigned task is waiting in the local queue, due to load changes and data movement, so we set *if* = 0.08. Note that all three dynamic strategies are suboptimal and heuristic.

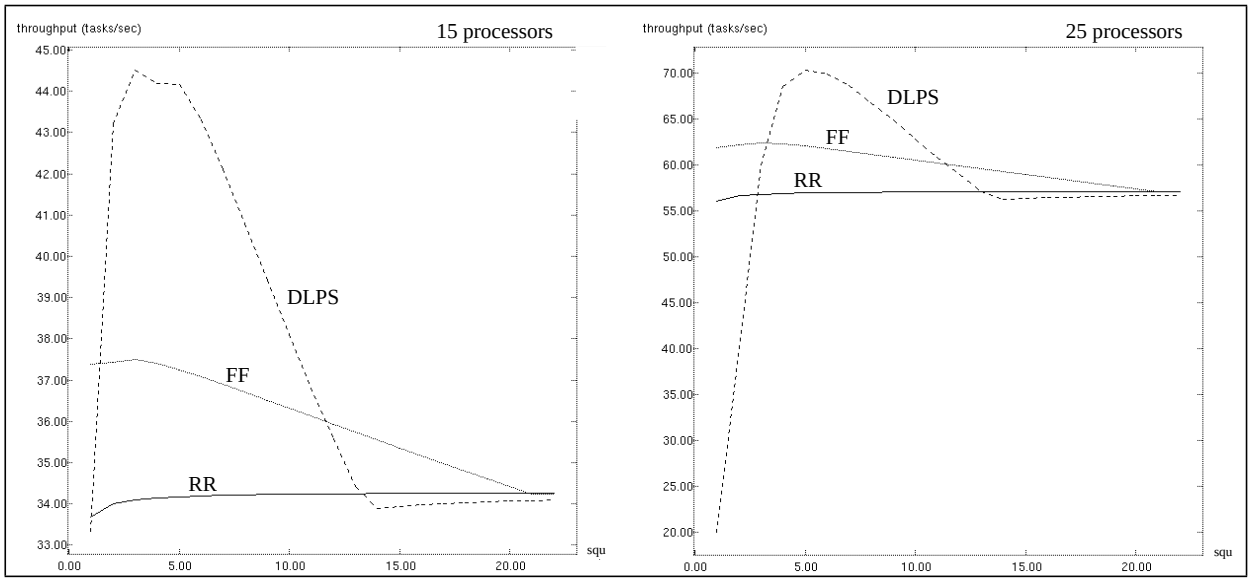

Figure 1: Server queue usage vs. throughput for different load balancing techniques

Figure 4, derived from the equations, compares the strategies at varying degree of server queue usage. A queue size of one task means, that load balancing may assign a task to an server not before it turned back to idle state. While RR throughput slightly increases due to sinking probability for servers to run out of tasks, the same effect at FF is soon predominated by the loss due to early assignment of tasks. Towards large local task queue size all strategies end up in equally distributing tasks regardless of the system behavior. While DLPS suffers from load balancing overhead caused by repeated rating of tasks residing in the central queue and long delays for idle servers at small local queue usage, there exists a range of stable maximum throughput, in which the effects of increasing server usage, shrinking overhead and assignment information getting antiquated, overlap. The right chart gives the performance degradation due to load balancing overhead in larger scaled systems, $p = 25$. Here, load balancing poses a bottleneck if strategies are complex and server queue usage is small. Some other parameter settings show less sensitivity to early assignment, but the results we present below prove, that there is significant influence in practise.

4 The Experimental Environment

The following results base on experiences with a prototype implementation of a dynamic, centralized load balancing environment called HiCon [2]. Applications running under HiCon are essentially client server structured, which has been established as a valid cooperation paradigm for parallel and distributed applications. Servers may operate on global data, which is distributed among the system. Load balancing is realized as a component responsible for server configuration management, task assignment and data movement as indicated in the right part of Figure 4. A central task queue per server class as well as local task queues at the servers enable the required assignment flexibility. Load balancing consists of a set of applicable strategies. Each strategy specifies a couple of actions like updating state information and task ratings, sending tasks from central into local server queues or migrate data partitions between servers. These actions are triggered by events like task arrival, end of task execution, data movement or resource load state change. The interested reader is referred to [2] for a detailed description of the underlying concepts. Instead we will summarize the effects obtained with exploitation of different local queue sizes below.

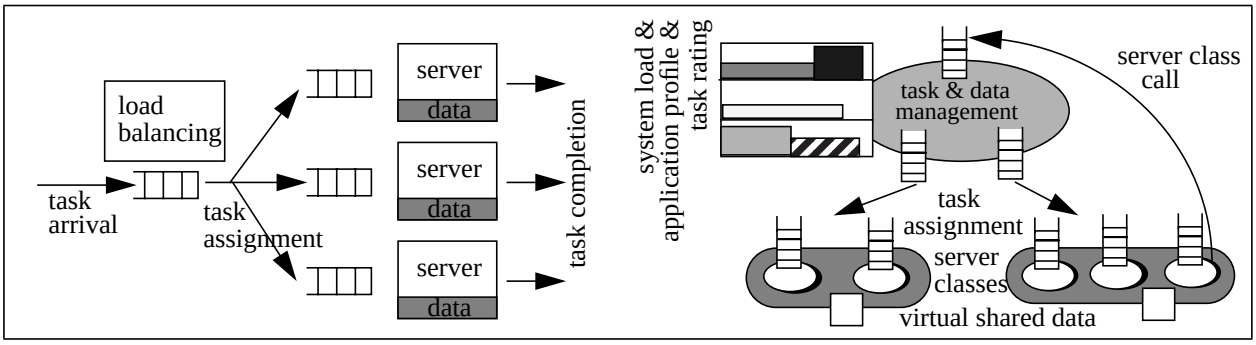

Figure 2: Analytic queueing model and HiCon load balancing structure

We observed three different applications solved in parallel under HiCon load balancing support at varying usage of local server task queues. The server call structures are shown in Figure 4. Two algorithms were implemented for the search of the shortest path between certain nodes within a given graph [2]. It turns out that the three applications above are able to show all effects derived in section 3.

The first one, called Graph, decomposes the search into a client, which just starts the search and combines the results, a server class GFind which collects the immediately reachable nodes from a given set of start nodes and sends the found nodes as call to the third class. This class, GReach, maintains a list of currently reached nodes. It inserts a set of reached nodes into this list and issues several server class calls to GFind, along with a set of nodes, that are worth further investigation. The graph is a set of edges, divided into partitions by start node ranges, each stored in a separate disk file. Similarly the list of reached nodes, a main memory array structure, is partitioned by node ranges, but the partitioning ranges may not match the graph partitioning. The GFind class further maintains a termination counter, a small data partition, which nevertheless tends to be a hot spot due to high update rates. The left part of Figure 4 shows the server call structure.

The second application, Graphx, solves the search problem with a client and one server class only. Here, the client controls the search process explicitly, i.e. sends search tasks, receives their results, divides the results into several new tasks and charges the server class with these new calls. The server class GFindx maintains both the source graph and the list of reached nodes, applies found nodes immediately to the reach list and yields a set of sorted nodes for further investigation.

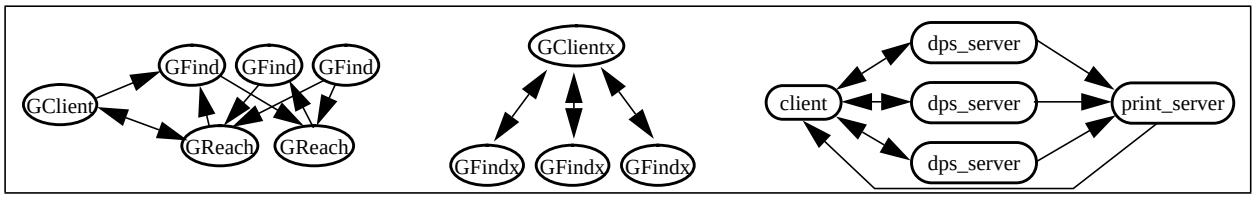

Figure 3: Server cooperation structure

The third application, called distributed picture segmentation (Dps) [14], is the most complex one, for it solves a real world problem in parallel under load balancing support. Segmentation is the process that subdivides an image into its constituent regions or objects. A region is an area in a picture whose points have a common property (discontinuity and similarity of the color values). The algorithm is roughly a split & merge technique; the application is realized by three server classes. The first server class, called client, is responsible for the synchronization of different calls. The second server, named dps_server, carries-out the image segmentation and the third, called print_server, is responsible for the correct saving of the segmentation result. The cooperation structure of the three server classes is shown in the right part of Figure 4. At the beginning

and at the end of the picture segmentation process the application has a high degree of parallelism, whereas in the middle of the processing the degree is low.

5 Measurement Results

The applications were observed under the configuration shown in Figure 4. We used four different load balancing strategies and varied the available server queue size for each application. Figure 4 summarizes the resulting response times. The strategies RR, FF and DLPS(1) have been intro-

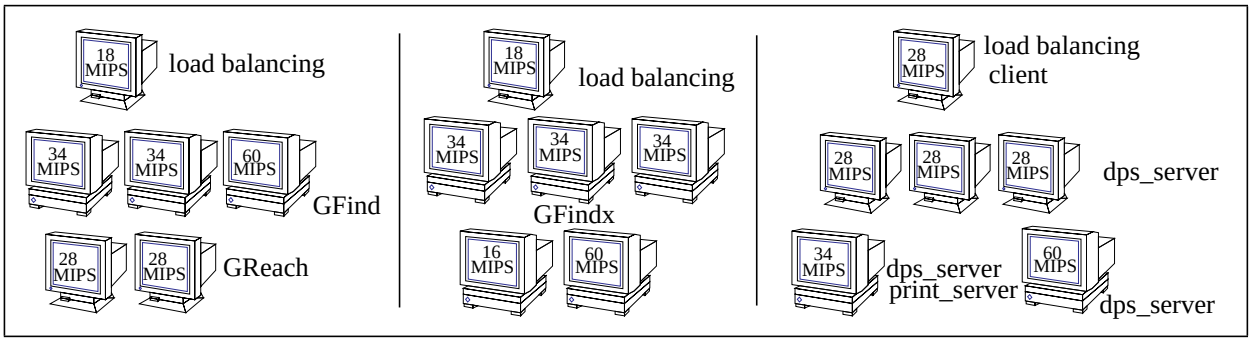

Figure 4: Measurement system configuration.

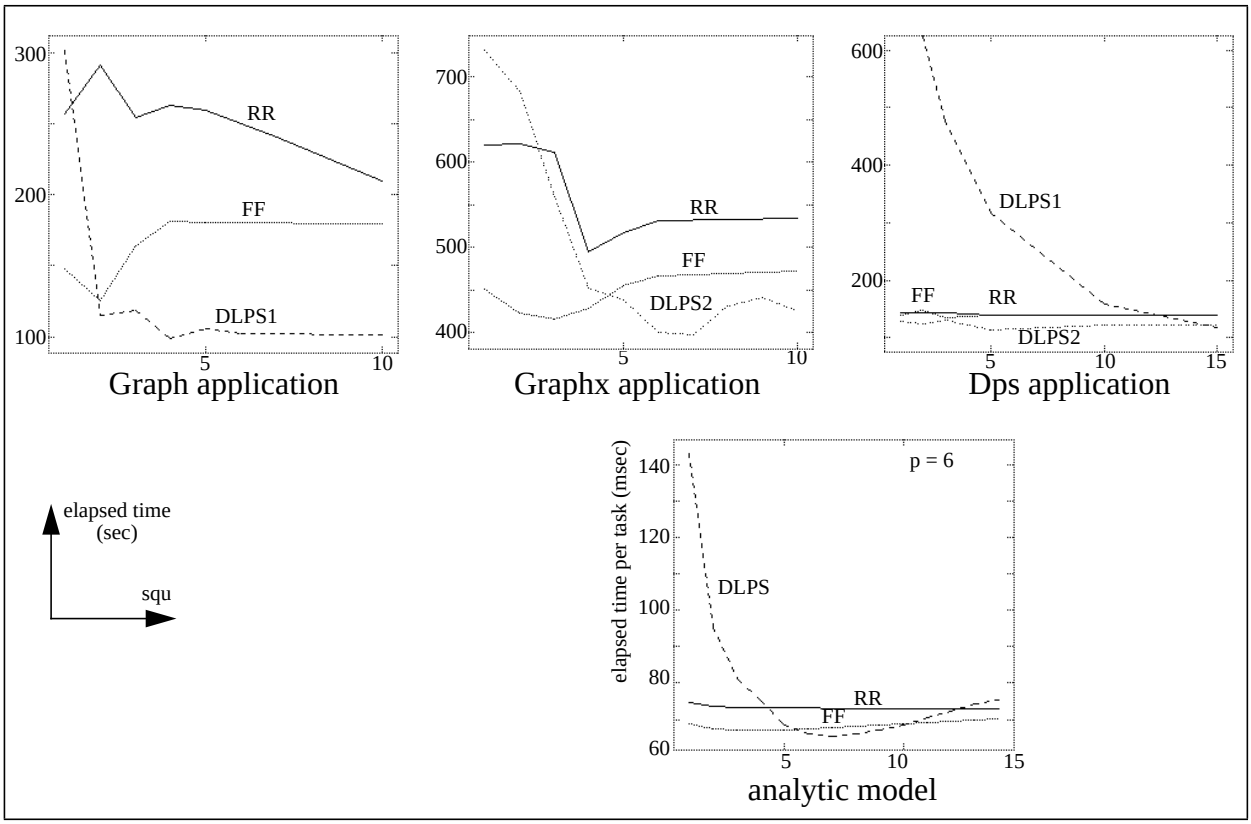

Figure 5:Local queue size vs. elapsed time for different applications and load balancing strategies.

duced in section 2. The strategy DLPS2 is a heuristic modification of DLPS1: it checks at maximum 20 entries in the central task queues per load balancing event. So it works faster but less accurate. The analytic evaluation of section 3 is repeated at the bottom of Figure 4 to show the correlation to the actual measurement results. The y-axis is inverted from throughput into elapsed time. All parameter settings are the same as in section 3, except for the load balancing decision costs, which had to be increased slightly to comply with the measurements.

The first effect we expected was a huge load balancing overhead at complex strategies along with short server queues. In both Graph and Graphx application load balancing never became a bottleneck although placed onto the smallest workstation. The Dps application however, contains a phase in which lots of tasks arrive that cannot be scheduled immediately, each of them announcing access to about a thousand global data elements. So the DLPS1 strategy spent too much time in re-rating all the tasks in the central queue at different events. The right part of Figure 4 tells that

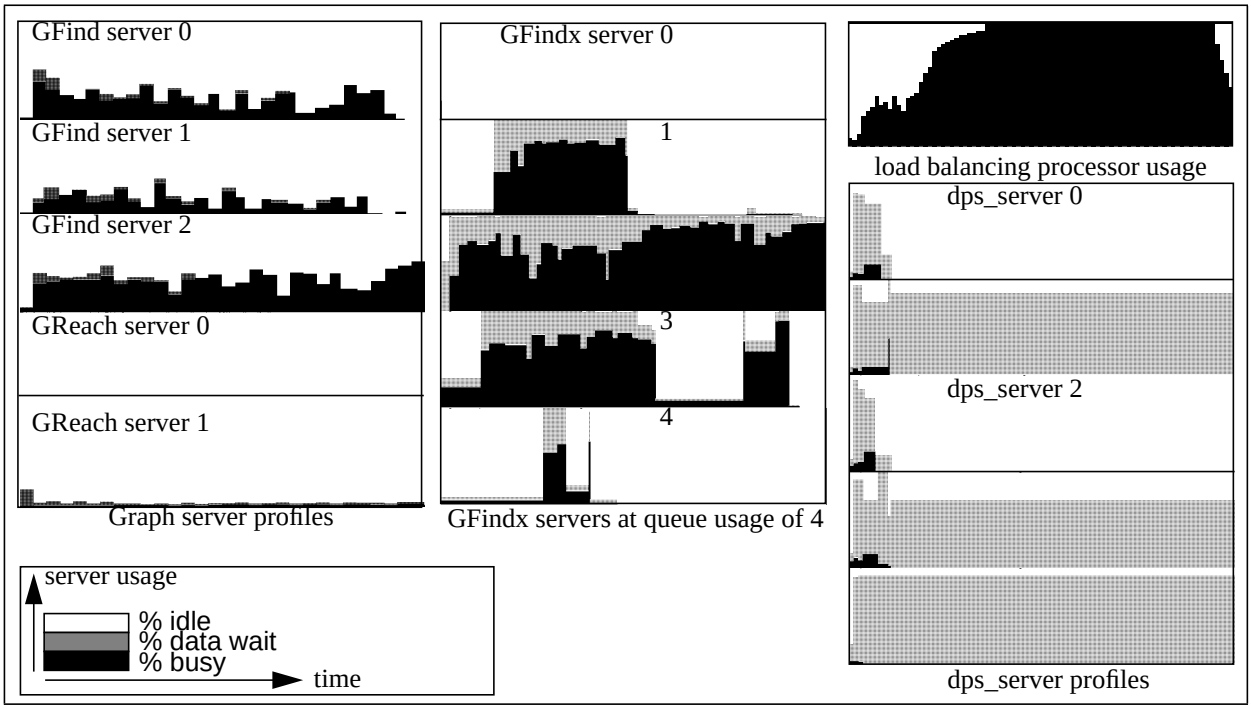

Figure 6: Execution profiles of certain application and load balancing behavior.

servers are mostly waiting for data, because this also involves load balancing actions in this HiCon prototype. At local queue sizes beyond 12 it outperforms the simple strategies and behaves similar to DLPS2, which less accurately updates the information.

The loss of load balancing improvements due to aging information is not as well visible from the measurements as it was in section 2. In spite of server queue usage growing larger, there still remains some difference in throughput between the complex and the simple methods. There are two reasons for it: first, because measurements where not taken during real life multi-user operation, system load and data placement changed comparably slow. Second, the load balancing strategies are implemented in a way they may use the server queues up to the given limit, but we could not force them to really exploit the maximum allowed (if it seemed unreasonable to them).

A better usage of the servers in terms of keeping them busy all the time and exploiting possible parallelism with server queueing could be observed for all three applications. Especially the calls to class GReach within the Graph application yielded very small tasks only. This class becomes the bottleneck under DLPS1 when driven without local queueing. In the case of short running tasks, the delays for getting a next task are comparably expensive. As the left part of Figure 4 shows, the servers were almost unemployed. Class GFind is the bottleneck for all queue sizes beyond 1, which is the main reason for the minimization of the execution time. It should be mentioned, that DLPS1 decided to employ only one server of class GReach because of its high data update rate. Thus it clearly defeats RR for all larger queue sizes, whereas RR more often employs both GReach servers at increasing local queue size.

The Graphx application has even more interesting properties: the maximum parallelism for both strategies is about 14 tasks, so larger local queues could not be exploited. Tasks in Graphx are rather long running (about ten seconds on a 34 MIPS processor), so load balancing was never overstressed. Note that most other researchers look at even larger tasks in the range of minutes or hours, but we are able to further decompose such tasks and thus obtain more load balancing benefit. However, Graphx's major problem without local queueing was that parallelism cannot be exploited, and so the maximum throughput was achieved at a queue usage of five (see middle of Figure 4). This is essentially the third consideration explained in section 2.

Dps under RR and FF marginally benefits from local task queues, because the time to compute the load balancing decision is negligible. So the difference of working with or without a local queue is the elapsed time for a message between the load balancing component and some servers. The more advanced strategies are able to speed up Dps, DLPS1 requiring some local queue size. The strategy DLPS2 is - at least for this application profile - an improvement of DLPS1, for it checks less tasks of the central queues per load balancing event. It approaches the minimum execution time when the local task queues are greater than six.

6 Conclusions

We tried to set up some simple straightforward calculations to estimate the server queueing effects for a wide range of computing systems, applications and dynamic load balancing approaches. The measurement results presented above cannot be viewed as a complete evaluation of early task assignment advantages. But they demonstrate that there are several different dependences on load balancing efficiency.

In principle, allowing large server queues causes no harm as long as the load balancing policies are good enough to decide on their own, which size can be fruitfully exploited. Many heuristics proposed in the literature are still too simple or too restricted in their functionality. Local server queues provide some potential to improve load balancing without adding much decision or information acquisition overhead. Further, adaptive load balancing may adjust the optimum local task queue usage based on the system and application factors we elaborated in this paper.

7 References

[1] K.Baumgartner, B.Wah, *A Global Load Balancing Strategy for a Distributed Computer System*, Workshop on the Future Trends of Distributed Computing Systems in the 1990's, September 1988.

[2] W. Becker, *Globale dynamische Lastbalancierung in datenintensiven Anwendungen*, Fakultätsbericht 1993 /1, Universität Stuttgart, Institut für Parallele und Verteilte Höchstleistungsrechner, 1993.

[3] F. Bonomi, A. Kumar, *Adaptive Optimal Load Balancing in a Heterogeneous Multiserver System with a Central Job Scheduler*, Proceedings Distributed Computing Systems, June 1988.

[4] T. Casavant, J. Kuhl, *A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems*, IEEE Transactions on Software Engineering, Vol. 14, No. 2, February 1988.

[5] Y. Chow, W. Kohler, *Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System*, IEEE Transactions on Computers, Vol. 28, No. 5, May 1979.

[6] D. Eager, E. Lazowska, J. Zahorjan, *A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing*, Performance Evaluation, Vol. 6, March 1986.

[7] K. Efe, B. Groselj, *Minimizing Control Overheads in Adaptive Load Sharing*, Proceedings 9th International Conference on Distributed Computing Systems, June 1989.

[8] D. Ferrari, S. Zhou, *A Load Index for Dynamic Load Balancing*, Proceedings Fall Joint Computer Conference, Dallas, Texas, 1986.

[9] C. Hsu, J. Liu, *Dynamic Load Balancing Algorithms in Homogeneous Distributed Systems*, Proceedings Distributed Computing Systems, 1986.

[10] P. Krueger, M. Livny, *A Comparison of Preemptive and Non-Preemptive Load Distributing*, Proceedings Distributed Computing, June 1988.

[11] F. Lin, R. Keller, *The Gradient Model Load Balancing Method*, IEEE Transactions on Software Engineering, Vol. 13, No. 1, January 1987.

[12] H. Lin, C. Raghavendra, *A Dynamic Load-Balancing Policy With a Central Job Dispatcher (LBC)*, IEEE Transactions on Software Engineering, Vol. 18, No. 2, February 1992.

[13] R. Mirchandaney, D. Towsley, J. Stankovic, *Analysis of the Effects of Delays on Load Sharing*, IEEE Transactions on Computers, Vol. 38, No. 11, November 1989.

[14] R. Pollak, *Lastbalancierte parallele Flächenerkennung*, Diplomarbeit No. 974, University of Stuttgart, March 1993.

[15] P. Yu, A. Leff, Y. Lee, *On Robust Transaction Routing and Load Sharing*, ACM Transactions on Database Systems, Vol. 16, No. 3, 1991.