Exploiting Inter Task Dependencies for Dynamic Load Balancing

Wolfgang Becker, Gerlinde Waldmann
University of Stuttgart
Institute for Parallel and Distributed High Performance Systems (IPVR)
Breitwiesenstr. 20-22, 70565 Stuttgart, Germany
email: wbecker@informatik.uni-stuttgart.de, waldmann@informatik.uni-stuttgart.de

Abstract

The major goal of dynamic load balancing is not primarily to equalize the load on the nodes of a parallel computing system, but to optimize the average response time of single requests or the throughput of all applications in the system. Therefore it is often necessary not only to keep all processors busy and all processor ready queue lengths within the same range, but to avoid delays and inefficient computations caused by foreseeable but ignored data flow and precedence constraints between related tasks. We will present concepts for dynamic consideration of inter task dependencies within small groups of tasks and evaluate them observing real applications in a load balancing environment on a network of workstations. The concepts are developed from scheduling of single task graphs towards heterogeneous multi user operation scenarios.

1 Introduction

The availability of parallel processing systems and computer networks offers a multiple of the processing power, which can be obtained even from a fast single processor. However, while applications can fully exploit the capacity of a single processor, it is hard to decompose and distribute applications in a way that they actually run faster on some parallel system. There are problems inherent to the algorithms and hardware, like data dependencies between tasks or communication cost, which limit task grain size and achievable speedup. But there is also the important problem of proper managing parallelism, i.e. how to distribute the tasks and data among the system. Load balancing of an application means to exploit parallel processing power as far as it accelerates the execution, and to assign the tasks of the application to processors in a way that minimizes synchronization and data communication overhead.

Parallel systems should not be reserved by a single application but run several applications concurrently. This enables good usage of the available processing power, provided that the load is evenly distributed among the system. Independent applications do not know about the resource usage of one another. Therefore load balancing in multiuser environments has to assign and distribute the tasks to achieve equalized usage of the system resources.

Load balancing should not just try to get each single task worked off as fast as possible, but maximize the overall application throughput. It is a common, but unrealistic assumption that all tasks are more or less unrelated and the assignment decisions can be performed isolated for each task, just considering the total processor loads. To execute a complex structured group of cooperating tasks as fast as possible, it is sometimes necessary to prefer more critical tasks and neglect less critical ones. This concept of task priorities can be extended to task groups that run concurrently. Further it is often essential to consider the data flow within task groups, because placing a task onto another processor than its predecessors may cause significant data communication overhead.

In this paper we will show, how existing concepts for static scheduling of related tasks can be integrated into dynamic load balancing schemes. The next two sections provide some insight into the problem and the basic mechanisms that can be found in previous research projects. After that, the dynamic load balancing environment, into which the concepts are converted, is introduced and an application to be used for the evaluation of the concepts is presented. The remaining sections develop the integration into dynamic load balancing with different degrees, each followed by performance evaluations.

2 Motivation and basic concepts

Using three scenarios we will show, what the problem of scheduling precedence constraint tasks is about and which potential of performance improvement exists. The examples are kept simple to focus on the key issue. They all use a system of equivalent processors; the goal is to execute a finite set of tasks as fast as possible. Each figure shows the task precedence graph on the left and two schedules (a) and (b) on the right. We do not employ specific scheduling

Proceedings IEEE Third International Symposium on High-Performance Distributed Computing
San Francisco, California, August 1994

strategies but give intuitive good schedules, which do not consider the tasks' dependencies (a) or exploit them (b). The schedules consist of one line per processor; the task executions are drawn on the time axis to the right.

In the first scenario (figure 1) the critical path D-E-F does not start with the largest task. Schedule (b) is faster, because it uses not just the task size as priority for assignment but the whole path length from the task to the end. The same effect would occur, if there were two smaller tasks E1 and E2 instead of E.

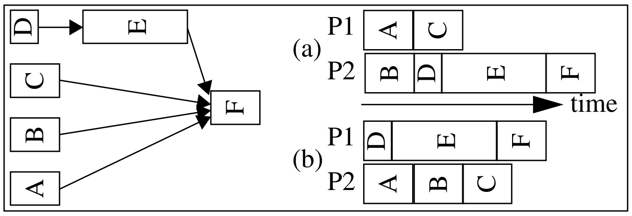

Figure 1: Consideration of the critical path.

The second scenario (figure 2) gives a similar constellation, where the critical part is not only a sequential line of tasks. So the problem arises, how to schedule several critical tasks. Schedule (b) recognizes A and C as most important and hence yields a faster execution.

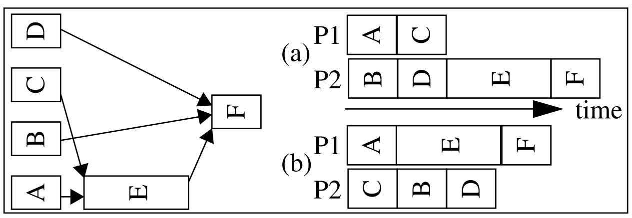

Figure 2: Consideration of multiple critical tasks.

The third example (figure 3) contains no critical path; it has no path longer than other ones at all. But there exists a critical part within the graph that should be executed prioritized, because it requests for more parallel processing power than other. This parallelism is not visible at the leftmost tasks but arises later on. Schedule (b) performs better, because task D is preferred due to its following degree of parallelism.

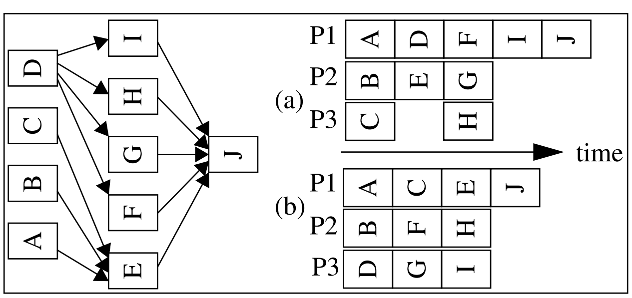

Figure 3: Reservation for future parallelism in the critical tree section.

The examples illustrated the main problems for homogeneous processors only. Consideration of task dependencies is even more important if there are processors of different power. Then higher prioritized tasks should be assigned to faster processors.

After this motivation we will describe the main classical methods for static scheduling of tasks within a group according to precedence constraints.

The first common method is called level scheduling (or 'highest level first'). The task graph is partitioned into levels, where the first level consists of the initial tasks, which have no predecessors. On the next level there are tasks that have only initial tasks as predecessors. This partitioning is continued until all tasks are placed within some level. Figure 4 shows the levelling of the task group used in the second example above. Scheduling works off the tasks level by level. Within one level, the tasks may be prioritized according to their size (heaviest node first strategy).

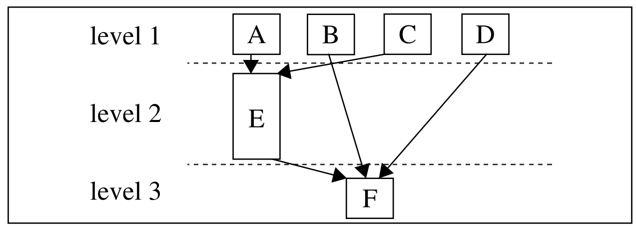

Figure 4: Task graph levelling.

A more sophisticated method, called priority scheduling, gives priorities according to the accumulated execution times along the path following the tasks (exit path length). This is due to the observation that the tasks along the critical path must be executed sequentially and determine the run time of the whole task group. The final tasks, which do not have successors, get priorities according to their size. After that, priorities are calculated backward through the precedence graph. A task's priority is its size plus the highest priority of its successor tasks. Figure 5 shows the priorities assigned to the tasks used in the first example (assuming task sizes of 5, 10 and 20). Priority scheduling assigns all tasks in the order of their priority: the task, which is executable and has the highest priority, is assigned to the best processor, i.e. the processor where it is finished soonest.

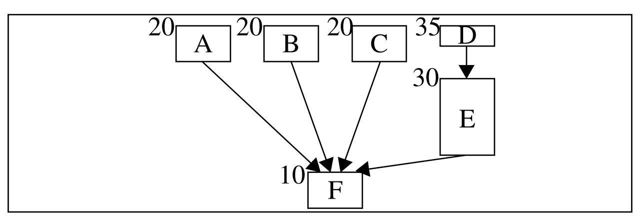

Figure 5: Task priorities according to path length of subsequent tasks.

This exit path length priority calculation can be extended using some weighted path length for prioritizing. The idea is to increment priorities of tasks entailing a high degree of parallelism. To each task's priority (calculated like shown above) the sum of the priorities of all its successor tasks - except the highest one - is added, divided by the number of processors. Note that the proposals in literature divide the sum by the highest priority of its successor tasks, which makes sense only if task sizes are in the magnitude of one. Our formula weights according to the significance of the needed parallelism compared to the available parallelism. Figure 6 shows the priorities for the tasks of the third example with and without weighted path length.

At last we need to mention an apparently restriction for the applicability of the priority schemes developed above. Favoring tasks with higher priorities does mainly help to reduce unnecessary idle times. Idle times can be avoided, if more parallelism can be set free in a low loaded system, so that all processors are kept busy (see the third example above). Further idle times can be avoided, if the system load collapses after some certain time. For example, most static scheduling approaches look at a finite set of tasks; if they are done, no more load will occur. So static scheduling tries to keep all processors employed until the end. But more general situations like synchronization points occur within many applications; they also yield low parallelism and idle processors, if the parallel tasks do not finish simultaneously.

In general, the priorities cannot be fruitfully exploited, if all processors in the system are so heavy loaded that idle times are avoidable, even when load balancing ignores task precedence constraints. In this case, using the priorities may reduce response times for single tasks or task groups; it will not increase overall throughput. So, if idle times do not occur, no dynamic load balancing should happen; if idle times are avoidable by considering isolated tasks only, no inter task dependencies need to be exploited.

3 Previous work

Several proposals for static scheduling and static load balancing can be obtained from literature. For some methods the best and worst case behavior could be verified formally. Most approaches are restricted to specific application types or processor structures and investigate only specific dependency and communication graphs (like in-trees or out-trees, shown in figure 7).

Hu looks at a set of equal sized tasks within an in-tree precedence graph and a system of homogeneous processors [10]. Tasks are given priorities according to their exit path length (see section 2) and the tasks with higher priorities are scheduled preferred (priority scheduling).

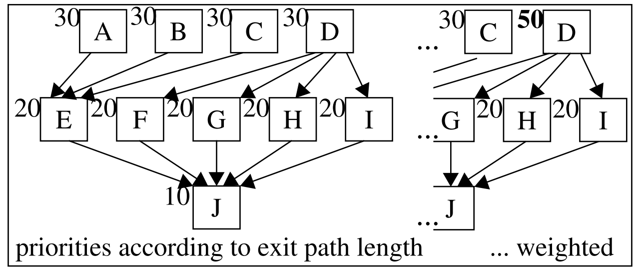

Figure 6: Comparison of normal and weighted priority assignment.

Task scheduling of independent exponential distributed size onto two processors is investigated by [4]. If the precedence constraints are of the form in-tree, then level scheduling is proofed to be optimal. This method converges to the optimum with growing number of tasks even if more than two processors exist, as explained in [14].

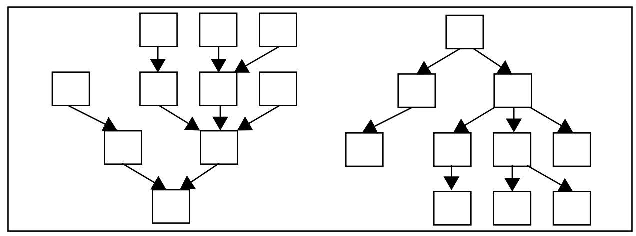

Figure 7: Examples of task graphs structured as an in-tree (left) or an out-tree (right).

Coffman and Graham develop an algorithm to assign in-tree constrained tasks onto two identical processors [6]. It is optimal but suspends and migrates running tasks (preemptive scheduling). This method is extended in [5] for two heterogeneous processors and arbitrary precedence constraint graphs. It is proofed to be still optimal.

The best case behavior of some preemptive and non-preemptive scheduling algorithms is examined in [12].

Pinedo and Weiss [15] investigate methods similar to the level scheduling scheme. The tasks of the in-tree precedence graph are assigned backwards level by level. Tasks sizes are independent exponentially distributed. The strategy is shown to be optimal for non-preemptive as well as for preemptive schemes.

In [13], five heuristic methods are presented within a simulation tool, which assign tasks onto specific processor network topologies (ring, mesh, full connected etc.). Roughly spoken they are simplifications of the priority assignment method for exit path length prioritized tasks.

A Markov-Chain based simulator is developed in [19] to examine response times of different assignment strategies. Task sizes are exponentially distributed. Precedence constraints may be probabilistic, i.e. only one of the successor tasks will be chosen and executed.

Genetic algorithms can also be employed for task scheduling [11].

The following references consider communication relationships between tasks. Intuitively, parallel running communicating tasks should be grouped close together, successor tasks should be assigned near their predecessors, from which they accept data. However, some studies found in literature take the interconnection channels as additional resources, which also have to be reserved and assigned during the scheduling procedure.

Shirazi and Kavi describe an extension of the level scheduling method to take into account communication cost between tasks [17]. Without communication, for the executable task with the highest priority the processor is chosen, which is the first to become idle. Now, to the time, when the processor becomes idle, the communication time needed to get all data from predecessors on other processors is added. Some further enhanced algorithm is given by [18].

A similar method which is fast under the assumption that there are always enough free processors, is presented in [1].

Lewis and El-Rewini [13] (already mentioned above) also extend two methods to minimize delays due to communication with reusing the same processor like the communication partners of the task.

A modified priority scheduling approach is developed in [8] to place communicating tasks to processors that are close together within some certain topology (ring, mesh or hypercube). The communication delays along the exit path are also added to each task's priority.

Winckler gives some strategies to efficiently schedule relevant cases of very small task graphs at run time [21].

4 The dynamic load balancing environment

To evaluate the real applicability of the concepts we integrated the inter task dependency considerations into an environment for dynamic load balancing, called HiCon. This environment is introduced in more detail in [2] and [3]. For brevity we will restrict here to the issues relevant for the task dependency concepts. The overall goal of the HiCon project is the development of advanced, more flexible and adaptive dynamic load balancing for data intensive computations on parallel shared nothing architectures like workstation clusters. Three approaches extent load balancing to manage a wider range of applications and systems than possible nowadays. First, load balancing can be shifted between central and distributed structures. Second, load balancing is able to dynamically adapt its decision parameters to the current application and system state. Finally, load balancing may exploit various resource informations for decision making, like cpu run queue lengths, task queue lengths, location of data partitions or copies and estimations about tasks. In this paper we investigate one further information source to be exploited by dynamic load balancing: the precedence constraints between tasks of small groups.

The HiCon model uses a client - server processing scheme. Applications are functionally decomposed, and a client, which controls the application flow, issues tasks to be worked off by some instance of a server class. Of each server class multiple instances can be distributed among the processors. There is no explicit communication between servers; instead, they cooperate by using global data structures. Data partitions can be accessed in shared or exclusive mode; data and several copies of them may move across the system between servers. Parallelism within an application can be achieved issuing several calls before awaiting results (asynchronous remote procedure calls). In this environment load balancing is expected to assign tasks to servers in a way that maximizes overall throughput. Figure 8 gives a simplified overview of the load balancing architecture. The architecture can be extended to multiple cooperating load balancing components, which manage partitions of the whole parallel system.

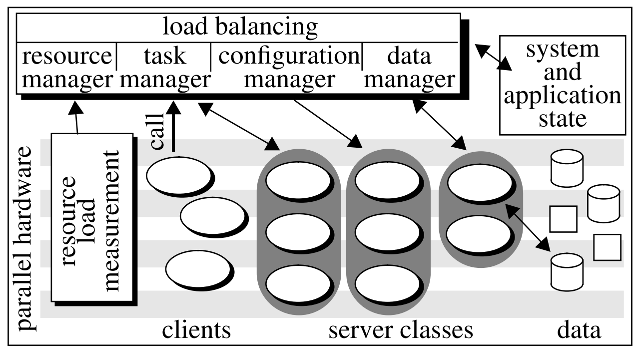

Figure 8: Load balancing architecture.

The task assignment and execution model in HiCon is defined as follows. Tasks can be kept at the load balancing component in central queues per server class, load balancing may assign these tasks at any time in arbitrary order to some server instance. Servers have local task queues which are worked of sequentially in first-in first-out order. Once they are assigned to a server, tasks cannot be migrated any more. On each processor several servers may reside, even multiple servers of the same class. When idle, they do not bind resources, while working they share the processing power by time slicing. If we restrict the configuration to one server per processor, the execution model is similar to the model required by the static scheduling methods.

Load balancing takes periodic measurements of the hardware usage. Otherwise it is passive and possibly reacts on different events like call issues, task finishes, processor load changes or even data movements. It then updates its information tables, task - server ratings and may assign several tasks to certain server instances.

5 A sample application type

For evaluation of the concepts a scenario of parallel database query processing was realized within the load balancing environment [20]. Complex database operations consist of operations like scan, projection, join etc., which partially depend from each other, because they need the intermediate results of other operations. The dependency graph is usually in-tree structured (see section 3). These precedence constraints, together with estimations about the task sizes are available at the start time of the query, because they are generated by query compilers and optimizers. However, static (compile time) scheduling is not applicable, because the sizes of the base relations and in consequence the task sizes change, the system load situation changes and even the location of the data may change. Dynamic query optimizing is necessary, but this is still object of research efforts (for examples see [7], [9]).

The implemented scenario enables the execution of complex structured queries. The available basic operations are projection (election of columns), selection (election of rows which fulfill some predicate), and join (tupel combinations of two relations, which fulfill some matching predicate). The base relations and also the intermediate results can be arbitrarily partitioned horizontal into separate files by clustering key ranges. So the basic operations can by parallelized according to the partitioning of the participating relations. Further there are operations to create complete base relations for start-up.

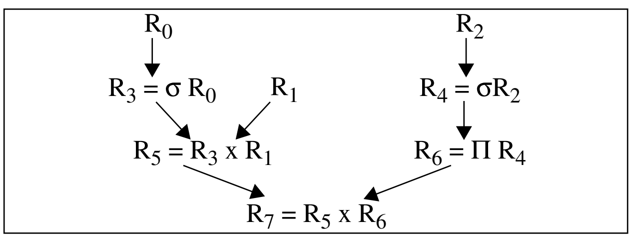

Figure 9: Operator tree of the query example.

The measurements shown below base on executions of the following query (relational algebra notation): $R_7 = ((\sigma_{a1>15000} R_0)\ x_{a0=a0}\ R_1)\ x_{a0=a0}\ (\Pi_{a0}\ (\sigma_{a1>100} R_2))$. Figure 9 gives the sequential operator tree for the query, Figure 10 shows one possible parallel query execution graph.

The tasks are tagged with numbers and the width of a block is proportional to the size of the task. Tasks are in the range of 3 seconds up to 2 minutes. Base relations are partitioned into 14, 21 and 28 partitions respectively and initially contain 1000 tupels per partition. This non-trivial parallelization was chosen for the evaluation, for it proofs that the dependency considerations apply to arbitrary graphs, not only to special forms.

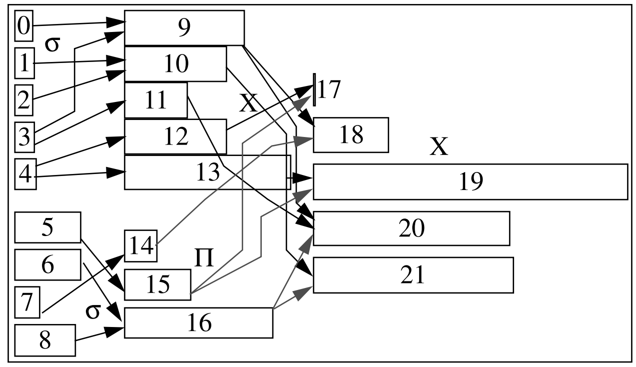

Figure 10: Task dependency graph for the parallel query execution.

The measurements were performed within a network of workstations, a 18 MIPS node for the load balancing component and client, two 34 MIPS nodes and one 16 MIPS node running one server each. The slow processor P3 was used to see, how load balancing takes into account the different processing power. For the multi user concurrency measurements we had to use another slow processor for P2 due to technical reasons.

6 Integrating task dependency consideration into dynamic load balancing

In the first approach we tried to keep close to the traditional scheduling approaches. Announced task groups can be scheduled after the level method or the priority scheduling method. The assignment of the expected tasks to processors is fixed at the announcement time (usually at runtime immediately before the task group is started). However, our experiences showed that it is unrealistic in a real environment to stick to the assignment time points obtained from the schedule. There are non-negligible stochastic deviations in task execution times. So we decided to keep just the reasons for the assignment time points. The client of the application guarantees that actual executable tasks will arrive corresponding to the precedence constraints. With its assignment time points the scheduling yielded *additional precedence constraints*, which should be remembered by the load balancing strategy. This method ensures that the basic results of the scheduling algorithm are exploited, but the success of the planning survives some arbitrary stretching of the participating tasks due to client's miscalculation of task sizes or system performance degradation due to foreign load. For an example of additional precedence constraints look at Figure 1: task B should wait until the finish of more critical task D.

This information (processor assignments and additional precedence constraints for each task of the group) is just stored at the load balancing component after the group announcement. It can be used later for assignment decisions when the single tasks actually arrive, now ready for immediate execution.

For performance evaluation we observed the database retrieval scenario (see previous section) under different load balancing strategies:

- *Round Robin* ignores the scheduling information and yields a simple strict round robin assignment of tasks to servers.
- *Highest Level First / Heavy Node First* uses the information obtained by a level scheduling algorithm (see section 2). We also investigated a modified form that takes into account the communication overhead due to data movements, if a task runs on a different processor than its successor tasks. When choosing the best processor for a task, this communication time is added to the expected execution time.
- *Critical Path* exploits the information from a priority scheduling algorithm with exit path length as task priority (also presented in section 2). The weighted exit path length yields the same scheduling in this case and therefore was not mentioned further.

Figure 11 shows *Gantt* diagrams for the task executions. The gray areas within task execution boxes give the amount of time spent for data communication. While the advantages of the preplanning are obvious, the difference between the scheduling policies are less significant, at least for this example. The actual execution times are partially different from these shown in figure 10, because processor P3 was slower than P1 and P2.

7 Extending task dependency consideration for heterogeneous multiuser operation

The second approach to integrate scheduling techniques for interdependent tasks into dynamic load balancing is more flexible and less sensitive to inaccuracy of task size prediction. It allows to employ all the rating and assignment strategies that were developed for load balancing without consideration of inter task dependencies. Further it is now able to deal with mixed announced and unrelated tasks as well as tasks of different independent task groups. The second approach is defined as follows:

- Clients may announce task groups arbitrarily at run time. Announced tasks will be placed into the central task queue of the class. They are marked as ‘not yet executable’ until the actual call arrives.
- Each task in the central task queue is attributed with a priority. This priority can be the size of the task, or, if the task belongs to an announced group, additional the length of its exit path or just the group’s average task size multiplied by the number of successor tasks. Note that priorities have the unit ‘work request size’ (number of instructions to perform). The idea is that each task’s priority reflects its amount of processing requirements, maybe plus the processing requirements it entails. The scheduling informations like ‘favorite processor’, additional precedence constraints or data flow hints, like used in the approach above, are no longer needed. So

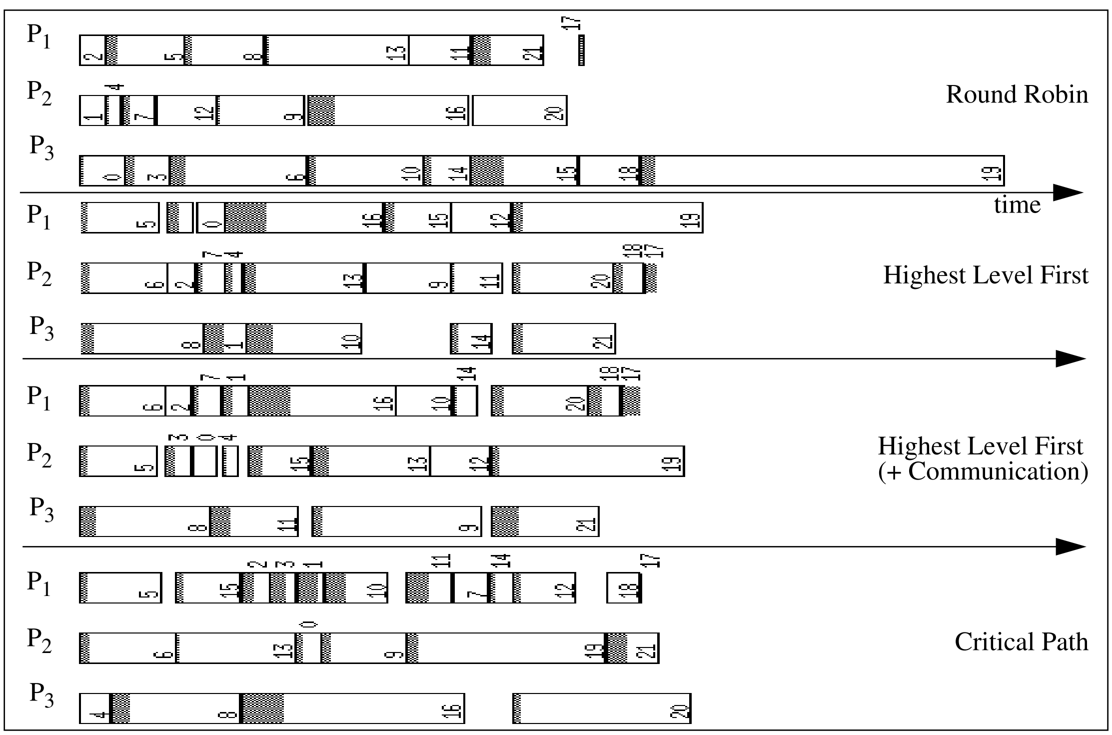

Figure 11: Execution profiles for different task group preplanning methods.

the 'static scheduling' procedure becomes obsolete and collapses into the calculation of each task's priorities.

- The assignment of tasks can be done by arbitrary strategies, maybe with or without consideration of the tasks' priorities. Data affinity can be considered in the very same way like it was done in 'normal' load balancing strategies.

There are two issues, where dynamic load balancing may exploit the precedence constraints: Between several executable tasks the task with the highest priority can be preferred, i.e. it gets the best processor or will be executed earlier than others. Second, executable tasks can be hold back or redirected elsewhere, because an announced, not yet executable task with higher priority should get the processor.

We implemented four ways to calculate the priorities of dependent tasks:

- Predicted task size without look ahead. Each task gets a priority equal to its estimated size.
- The group's average task size is multiplied by the number of successor tasks. This rating is important, if no accurate estimations can be made about the sizes of the individual tasks.
- Priority is set according to the task's exit path length including its own size. This corresponds to the critical path method (section 2).
- Priority is set to the weighted exit path length as described in section 2.

The priorities assigned in this way can be exploited for dynamic load balancing decisions in different fashion. We compared several strategies for task election and server rating. The first step of the load balancing strategy, when it is activated, is to select a task for assignment. There are two implemented strategies for task selection:

- The most simple method is to take the oldest task, which can be executed immediately.
- The second approach always assigns the executable task with the highest priority to the best instance. No reservation for more important tasks that are not yet executable, is made.

When all instances still have enough tasks in their local queues to work continuously, it may be better to delay assignment and hope that some higher prioritized task

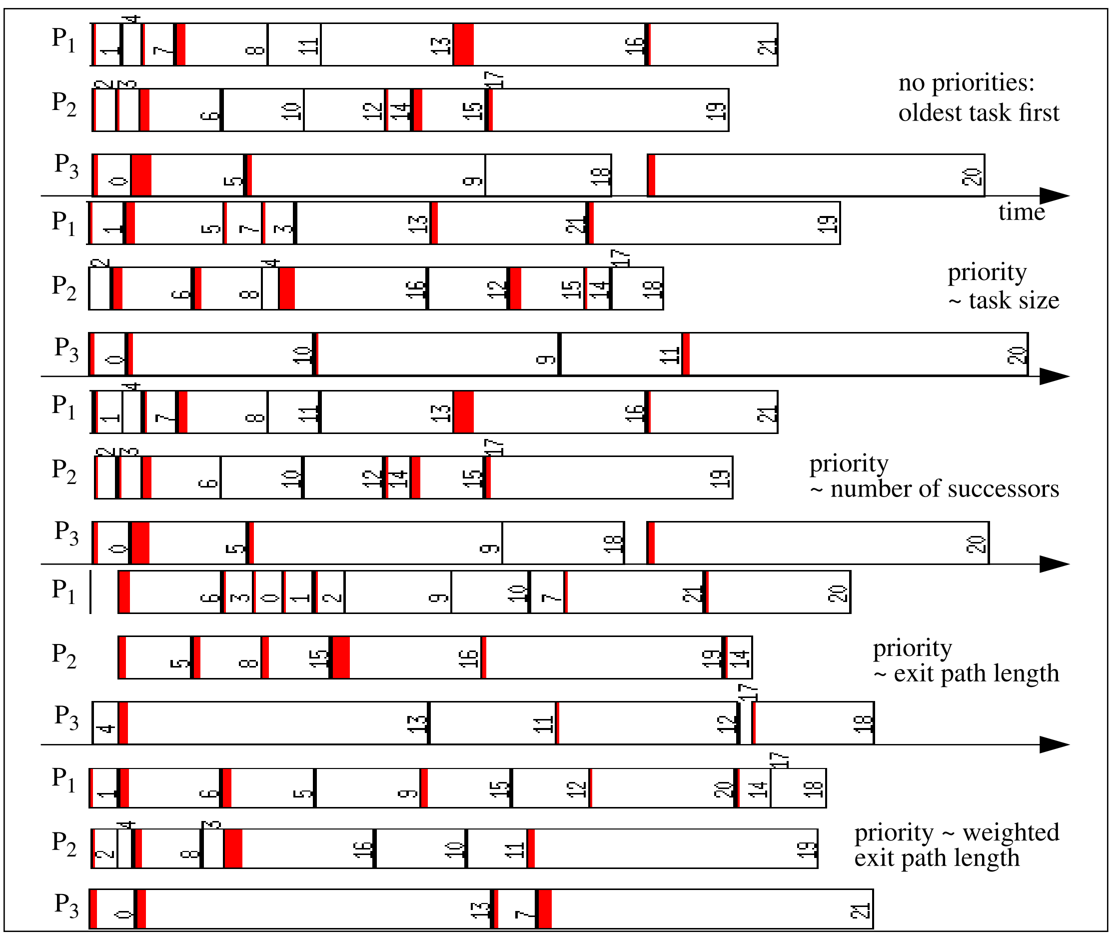

Figure 12: Execution profiles for different task priority approaches.

arrives as executable soon. We did not optimize the strategy in this way, because in situations of heavy, fairly distributed load, load balancing is less important.

At least in multi user operation, but also in other situations, it can happen that low priority tasks starve in the waiting queue, because there are always executable tasks with higher priorities to assign. So it is absolutely necessary to grow each tasks priority as he lies waiting for assignment. Within our approach it is straightforward to extend the original priority to the notion of *dynamic priority*, because priority is measured in the unit of ‘work to be done’ (number of instructions). We define the dynamic priority of an executable task in the central queue as its original priority plus the amount of work that could have been done since it is executable. This amount of missed work is estimated by the task’s waiting time in the queue multiplied by the average power of the processors.

The second step of load balancing action is to select the best suited server for the favored task. Although the load balancing environment allows to consider a variety of factors for this selection, we restricted attention to two strategies:

- *First Free* assigns the task to some idle server. If several servers are free, they are employed in round robin fashion.
- *Data Affinity* hands the task to the idle server that has most of the presumed data local available. If only one server is idle, this strategy acts equivalent to *First Free*. It is possible to force strict assignment, i.e. leave instances unemployed, if they do not have enough of the required data. In this scenario however, both version yielded no significant progress, so we excluded them from evaluation.

The results (figure 12) show that ignoring the precedence constraints or just weighting tasks according to their own size is not sufficient. For the task sizes are predicted comparably accurate and significantly differ in size, the number of successors also fails as base for priorities.

Comparing the results with the response times of the previous section, where assignment was based on scheduling, the more dynamic approach enables the same increase of performance by considering precedence constraints. Finally, we will look at the behavior of the strategies within a multi user operation scenario. For sake of simplicity the

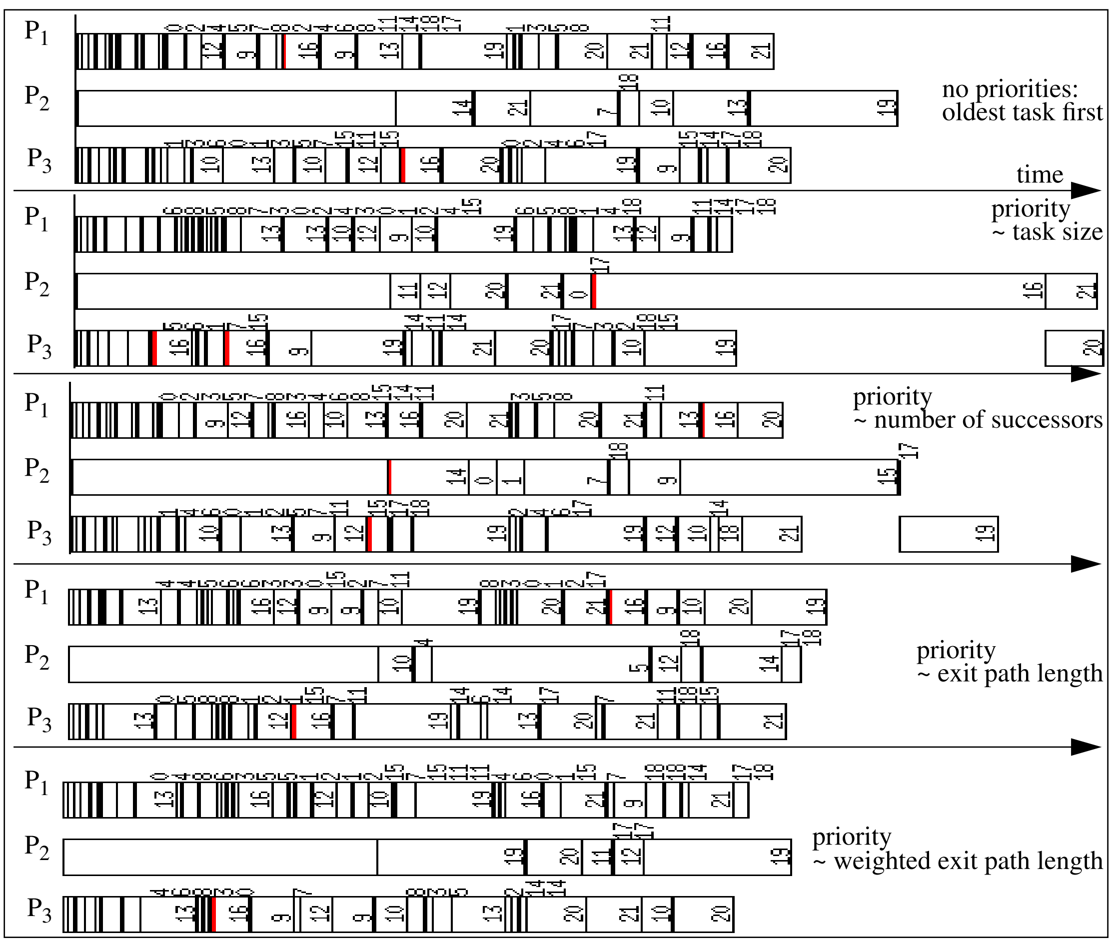

Figure 13: Multi user execution profiles for different task priority approaches.

same parallel query was performed concurrently, each on a disjoint set of relations. Figure 13 gives the execution profiles. The task numbers of the different applications are not discernible in the figure, but in fact the load balancing strategy calculates priorities separated for each task group. The start-up phases that created the base relations, are also displayed as unnamed tasks, because they no longer appear at the beginning altogether.

The diagrams show that in situations of concurrent task group processing the advantages of dependency consideration are even more evident. Again, prioritizing due to task size without look ahead is worse than using no priorities at all. The success of group preplanning in this scenario has the following reason: While within a single application the processing requirements almost appear level by level due to the precedence constraints, so that they implicitly obey a policy similar to highest level first scheduling (see section 2), concurrent applications - if dependencies are ignored - often push tasks into time slots, where other applications would need processing power for really important tasks.

8 Conclusions

This paper developed strategies for dynamic load balancing that exploit estimations about precedence relationships and task sizes within small groups. The estimations may be generated at run-time for a task group announcement and can be exploited by dynamic load balancing later as the executable tasks arrive. The additional information can be used in connection with other current state information. Measurements obtained from a parallel database application within a workstation environment showed the applicability and potential improvements of the suggested concepts.

The concepts are derived from static scheduling, where accurate knowledge of the tasks and their interdependencies is assumed. Dynamic load balancing usually has to deal with less look ahead information and greater deviations from expected behavior. Therefore load balancing decisions should be more conservative and stable. However, when exploiting task precedence constraints, even derivations of 50% in task sizes can decimate the performance gain. In consequence, the main goal of dynamic load balancing remains to get statistical throughput improvements and avoid serious load skew situations.

References

[1] F. Anger, J. Hwang, Y. Chow, *Scheduling with Sufficient Loosely Coupled Processors*, Journal of Parallel and Distributed Computing 9, 1990.

[2] W. Becker, *Globale dynamische Lastbalancierung in datenintensiven Anwendungen*, Fakultätsbericht 1993 /1, University of Stuttgart, Institute of Parallel and Distributed High-Performance Systems, 1993.

[3] W. Becker, *Das HiCon-Modell: Dynamische Lastverteilung für datenintensive Anwendungen in Workstation-Netzen*, Fakultätsbericht 1994 /4, University of Stuttgart, Institute of Parallel and Distributed High-Performance Systems, 1994.

[4] J. Bruno, *On Scheduling Tasks with Exponential Service Times and In-Tree Precedence Constraints*, Acta Informatica 22, 1985.

[5] K. Chandy, P. Reynolds, *Scheduling Partially Ordered Tasks with Probabilistic Execution Times*, Proceedings Operating System Principles, Operating Systems Review, Vol. 9, No. 5, 1975.

[6] E. Coffman, R. Graham, *Optimal Scheduling for Two-Processor Systems*, Acta Informatica 1, 1972.

[7] R. Cole, G. Gräfe, *Dynamic Plan Optimization*, 5th Workshop on Foundations of Models and Languages for Data and Objects, Aigen, Austria, 1993.

[8] H. El-Rewini, T. Lewis, *Scheduling Parallel Program Tasks onto Arbitrary Target Machines*, Journal of Parallel and Distributed Computing 9, 1990.

[9] G. Gräfe, R. Cole, D. Davison, W. McKenna, R. Wolniewicz, *Extensible Query Optimization and Parallel Execution in Volcano*, Workshop Dagstuhl, Germany, 1991.

[10] T. Hu, *Parallel Sequencing and Assembly Line Problems*, Operations Research 9, 1961.

[11] J. Kanet, V. Sridharan, *PROGENITOR: A generic algorithm for production scheduling*, Wirtschaftsinformatik 4, 1991.

[12] S. Lam, R. Sethi, *Analysis of a Level Algorithm for Preemptive Scheduling*, Proceedings Operating Systems Principles, Operating Systems Review, Vol. 9, No. 5, 1975.

[13] T. Lewis, H. El-Rewini, *A Tool for Parallel Program Scheduling*, IEEE Parallel and Distributed Technology, 1993.

[14] C. Papadimitriou, J. Tsitsiklis, *On Stochastic Scheduling with In-Tree Precedence Constraints*, SIAM Journal Computing Vol. 16, No. 1, 1987.

[15] M. Pinedo, G. Weiss, *Scheduling Jobs with Exponentially Distributed Processing Times and Intree Precedence Constraints on two Parallel Machines*, Operations Research 33, 1985.

[16] B. Shirazi, M. Wang, G. Pathak, *Analysis and Evaluation of Heuristic Methods for Static Task Scheduling*, Journal of Parallel and Distributed Computing, Vol. 10, No. 3, 1990.

[17] B. Shirazi, K. Kavi, *Parallelism Management: Synchronisation, Scheduling and Load Balancing*, Tutorial, University of Texas at Arlington, 1992.

[18] G. Sih, E. Lee, *A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures*, IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 2, 1993.

[19] A. Thomasian, P. Bay, *Analytic Queueing Network Models for Parallel Processing of Task Systems*, IEEE Transactions on Computers, Vol. 35, No. 12, 1986.

[20] G. Waldmann, *Dynamische Lastbalancierung unter Ausnutzung von Reihenfolgebeziehungen*, Studienarbeit 1280, University of Stuttgart, Institute of Parallel and Distributed High-Performance Systems, 1993.

[21] A. Winckler, *Context-Sensitive Load Balancing in Distributed Computing Systems*, Proc. ISCA Int. Conf. on Computer Applications in Industry and Engineering, Honolulu, 1993.