"Agency Scheduling" A Model for Dynamic Task Scheduling*

Johann Rost [1], Franz-Josef Markus [2] and Li Yan-Hua [3]

[1] J. Rost GmbH
Glockenhofstr. 24, D-90478 Nürnberg, Germany

[2] Medizinische Universität zu Lübeck
Institut für Technische Informatik
Ratzeburger Allee 160, D-23538 Lübeck, Germany
(email: markus@iti.mu-luebeck.de)

[3] Beijing Research Institute of Telemetry
P. O. Box 9212, 100076 Beijing, P. R. China

Abstract: This paper describes a class of algorithms for scheduling parallel programs represented by macro dataflow graphs (task precedence graphs) onto a multiprocessor system such that the total execution time is minimized. The schedule will be computed dynamically during the runtime of the process system. The model allows to represent centralized and fully distributed algorithms as well as intermediate forms. The algorithms are able to schedule static as well as dynamic dataflow graphs. Knowledge of the execution times of the tasks is not necessary. Some variants of the model have been implemented using a multi-transputer system. Practical experiences are included in the paper.

1 Introduction

MIMD multiprocessor systems are increasingly applied in wide areas of computer science. In order to take advantage of the potential increase of computation power, it is first necessary to split the algorithm into parallel processes. In a second step, these processes are to be assigned to the individual processors of the multiprocessor. In existing applications, a specific assignment for a given multiprocessor and for the current problem is often developed by the software engineer. Even though this solution may be slightly more efficient than automatic scheduling, it still has the disadvantages of high development costs, weak portability and a high degree of fault-proneness.

In recent years several variants of the task scheduling problem received increasing scientific attention. Many authors published especially on the following three problems:

(1) Load Balancing Problem

In a multiuser multitasking environment, the processes are started at any point in time. The tasks are to be assigned to the machines of a multiprocessor such that the load of all processors is balanced. Often deadlines and additional resource requests have to be considered. On the other hand, there is usually no communication structure assumed [LüM93] between the tasks. An overview is given in [CaK88].

* This work is partially supported by Deutsche Forschungsgemeinschaft DFG under contract number Ma 1412/1-2

(2) Task Scheduling with Task Interaction Graph

For each pair of tasks (t1, t2), the amount of communication between t1 and t2 is given. The data transmission can become necessary at any time during the runtime of the process system. Contrary to (1), all tasks are started at the same time. The problem here is to find a mapping of the tasks to the multiprocessor making the best possible use of the interprocessor network. Solutions of the problem are suggested in e.g. [Lee-88, Lo88]. An introductory overview is given in [CHL80].

(3) Task Scheduling with Task Precedence Graph

For some pairs of tasks (t1, t2), the dataflow graph (sometimes called "task precedence graph") defines that t1 must be finished before t2 can start. A schedule must be found which minimizes the execution time of the process system by minimizing both the communication overhead and the idle time of the processor. The problem can be solved either statically (before the process system is started) or dynamically. While the static solutions usually require good estimations for the execution times of the tasks and a fixed (static) structure of the dataflow graph, the dynamic algorithms consume an increased overhead for communication. Static solutions are given, for example, in [KaN84, MaL86]. A dynamic solution which could be called "Team Scheduling" is suggested in [RoM90].

This paper mainly focuses on dynamic solutions of problem (3). Overviews to the subject are given in [Bok87, ReF87, Hwa93].

2 Problem Description

2.1 Machine Configuration Graph

The multiprocessor (here only distributed memory machines are considered) is modelled as a graph with nodes (PMU, processor memory units) and edges (communication links, connections). The processor nodes are labelled with processing power and memory capacity while the edges carry a link capacity. The machine configuration graph as well as the labels are assumed to be given.

2.2 Dataflow Graph

The dataflow graph (Fig. 1) is a bipartite graph consisting of task nodes (circles), data nodes (squares) and edges. The data nodes describe results produced by one task and required by another one. Usually the execution time of the individual tasks is unknown. Sometimes estimations are available. Note that in some kinds of parallel programs the structure of the dataflow graph is not completely known in advance. The degree of branching or the number of loop iterations may be unknown at design time. If more than one dataflow graph can be executed simultaneously, this is called "space sharing". A model ("D^2R, Dynamic Dataflow Representation") which is able to express such features is described in [Ros94].

Fig. 1 shows a dataflow graph which can be used to parallelize some kinds of iterative algorithms for solving partial differential equations ("PDE"). For example, a heatened sheet is divided into two imaginary stripes. This division is made by t0 and will result in data sets d1 and d2. Then an iteration loop (named "loop") is entered. The graphic representation expresses the REPEAT structure of the loop (with at least one iteration). To identify the nodes of different iterations, the loop name is added to the node names within the loop body. During runtime the loop name is substituted by the loop counter value to provide the needed unique identification of the nodes. D^2R provides a similar mechanism when dynamic forks are used which are not applied in the example above. Each of the tasks

t1.\loop and t2.\loop executes one iteration step in one stripe. After completion of the iteration step, new data will be created (dl1.\loop, dl4.\loop). Moreover, the boundaries of the stripes must be exchanged (dl2.\loop, dl3.\loop). Finally the iteration is converging and the data is passed to "tx".

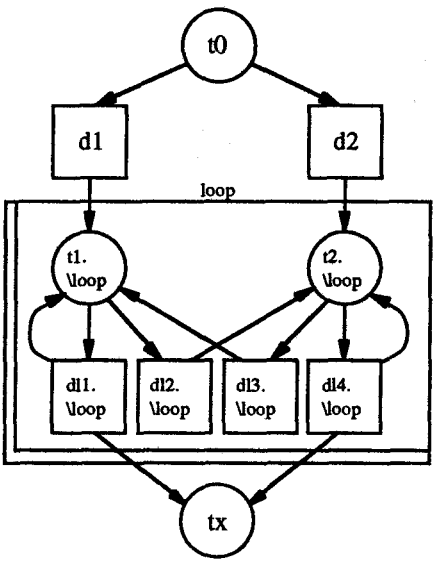

Fig. 1. Dataflow graph

3 Agency Scheduling Model

Up to now only few dynamic algorithms for the task scheduling problem have been suggested. The model suggested permits to easily characterize a wide range of algorithms. According to this model, an algorithm consists of eight rather independent components for which heuristics will be suggested.

3.1 Basic Ideas

There are a number of agencies each of which is responsible for a so-called "employer area". The employer areas constitute a partition of the processors of the multiprocessor, i.e. each processor is assigned to exactly one employer area. In addition, each agency is equipped with an employee area which may either match the employer area or exceed it in size and thus overlap with other employer areas. Jobs generated within the employer area of an agency are reported to the agency, which allocates them within its employee area. In the case of matching areas, the jobs may be allocated via a command protocol. Otherwise a bidding phase is required. If temporarily charged with heavy duty, an agency can delegate jobs to neighbouring areas via a load balancing mechanism. The processor carrying out the function of the agency can be dedicated to this job or can execute user tasks in addition. Moreover, a parameter N is given, defining the number of user processes which are permitted to be executed simultaneously, i.e. multitasking of user processes is allowed. Such agencies are best illustrated by comparing them with human labor offices.

The specification of an algorithm consists of the following eight components:

(1) A set of PMUs (the agencies).

(2) The corresponding employer areas.

(3) The corresponding employee areas.

(4) A boolean variable called "dedicated".

(5) The maximum number of user tasks running simultaneously on one PMU. On each processor, N virtual machines are running, each of which is executing its own protocols for reporting finished tasks and assigning unsolved jobs respectively. The individual user tasks may be running with different priorities.

(6) The protocol for allocating jobs. If each processor is assigned to exactly one agency, an easy command protocol can be applied. Otherwise possible conflicts have to be resolved - for example by using a bidding protocol.

(7) The protocol for reporting finished jobs.

(8) The strategy of balancing the load among the agencies.

Load balancing has been researched intensively. An overview is given in [CaK88].

3.2 "Team Scheduling" A Fully Distributed Variant of the Agency Scheduling Model

In the Team Scheduling algorithm, each processor maintains a *local knowledge* containing the idle and working PMUs, the location of the data sets and the state of the processes. This local knowledge is updated via a *broadcast mechanism* whenever a task is started or finished. After a task t has been finished, the scheduling algorithm will check if t has successors which can be started at that point. In this case, a bidding phase with presumably idle processors is triggered, during the course of which the successors are allocated. For more detail see [RoM90].

Team Scheduling is an extreme example of Agency Scheduling. Each processor is its own (non-dedicated) agency. The employer area is constituted by the processor only. The employee area is established by the entire multiprocessor.

It can be shown [Ros94] that the execution time of the schedule computed by Team Scheduling is at most twice as long as the execution time of the optimum schedule.

$T_{TeamScheduling} \leq T_{opt} * (2\text{-}1/NumberProcessors)$

4 The Components of the Agency Scheduling Model

4.1 Agencies and Areas

An easy way of defining the areas lies in the manual division of the multiprocessor. Using a regular mesh-like connection network, the whole multiprocessor can be divided into X*Y rectangular areas. The agency function will be assigned to a central processor (geometric center of gravity). An example for applying this strategy is given in Fig. 2. In this example, the multiprocessor is divided into four areas of equal size and matching employer and employee areas. The agency of each area is marked by an "A".

In the example of Fig. 3, the employee areas are overlapping. To avoid confusion, the employer areas are not displayed in the graphics - they are identical with the areas of Fig. 2. Note that in this example the employee areas exceed the employer areas in size. While in the

former example an easy command protocol can be used to allocate tasks, in the latter a more complex bidding phase is required which consists of "bid task", "accept task" or "not accept task", and "transfer task". On the other hand, overlapping areas open up a new prospect in load balancing: in this case, it is possible to balance the load without an explicit load balancing function. Instead (implicitly), tasks are assigned to the areas of the neighboring agencies. Applying overlapping areas in heavily loaded systems may cause several (or even many) agencies to send a "bid task" message as soon as an employee gets idle because the idle employee may belong to more than one employee area. To minimize the overhead caused by this message, an additional decision rule can be applied which says that in heavily loaded systems an agency bids a task mainly in its employer area. In other terms, in heavily loaded parts of the systems the overlapping areas are temporarily switched off.

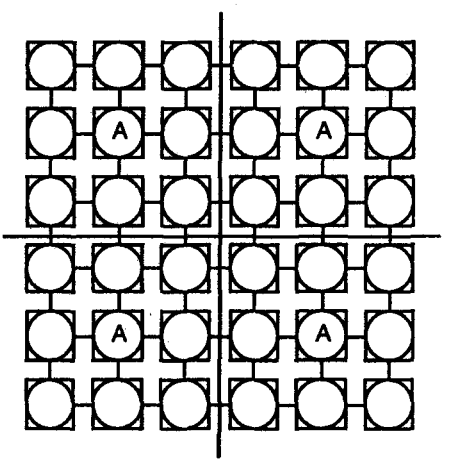

Fig. 2. Disjunct areas

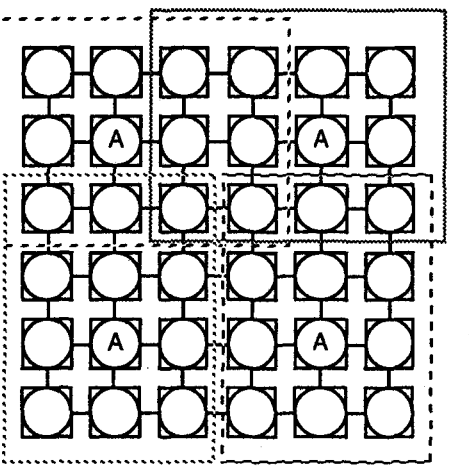

Fig. 3. Overlapping areas

To demonstrate the flexibility of the model by means of extreme examples, a central configuration is shown in Fig. 4. Central scheduling algorithms can be implemented more easily and often cause less dynamic overhead than distributed ones. However, the main disadvantages here are that the central agency constitutes a potential bottleneck ("monitor bottleneck") in larger-scale systems and that it is more difficult to solve the problems of fault tolerance. On the other hand, the extreme example demonstrates the wide scope of the model.

Quite the reverse of the central solution is the fully distributed configuration shown in Fig. 5, which is applied in the "Team Scheduling" algorithm. The drawback of the fully distributed configuration is that "Team Scheduling" needs broadcast messages to be distributed among the agencies (i.e. messages from one agency to all others) to maintain the local knowledge. Since in the discussed configuration all processors perform the agency function, the broadcasts among the agencies are, in fact, broadcasts all over the system, which may cause quite a dynamic overhead in large multiprocessor systems. We have already developed a variant of "Team Scheduling" which needs no global broadcast; it is, however, more difficult to implement this variant. For more details see section 4.3.

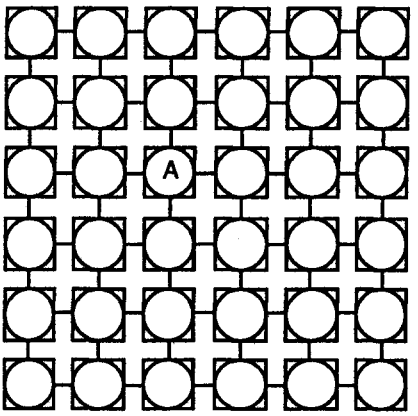

Fig. 4. Central configuration

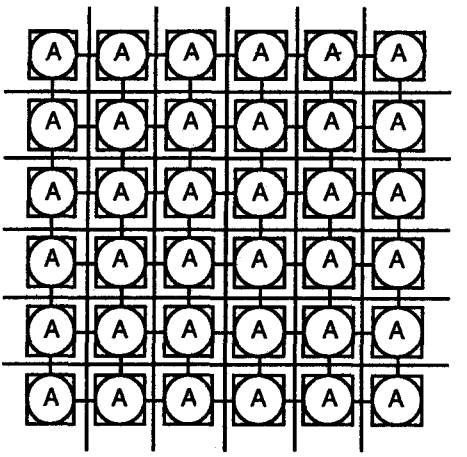

Fig. 5. Fully distributed configuration

4.2 Allocation of tasks

An interesting question arises if a task t depends on the results of more than one task, as for example t1 and t2 (see Fig. 6). In this case there must be an agreement between the agency which has allocated t1 (abbreviated as Agency(t1)) and Agency(t2) as to the question who is responsible for starting t. We tested three rules to achieve this agreement.

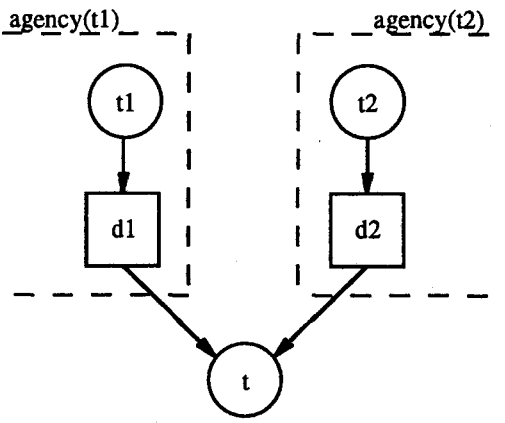

Fig. 6. Task t depending on multiple predecessors

In the following discussion, t_x is an arbitrary successor of t and the value WaitingTime is the time which a broadcast needs to be received all over the system. The function Ord(task) defines the lexicographic order of the name of the tasknode - in the easiest case the node number.

Our first idea was that Agency(t_x) is responsible for task t, if and only if t_x is the predecessor which finished *last*. The advantage is that at this time all required input data is

available and t_x can be started immediately. However, if t_1 and t_2 are finishing at almost the same time this rule can cause a conflict. Both Agency(t_1) and Agency(t_2) feel the other one to be responsible for starting t. To avoid this conflict, rule 1 uses a WaitingTime.

Rule 1:

case 1: if ($\forall$ i ((i $\neq$ x) => t_i is finished))
Agency(t_x) is responsible

case 2: if ($\exists$ i ((i $\neq$ x) $\wedge$ t_i is not yet started))
Agency(t_x) is not responsible

case 3: else:
- Agency(t_x) broadcasts "t_x is finished"
- Agency(t_x) waits for at least two WaitingTime units
- 3.1 Agency(t_x) receives a "finish" broadcast from another Agency(t_i)
The agency with the smaller PMU number is responsible
- 3.2 else
Agency(t_x) is not responsible

Rule 1 has some disadvantages:
- Two broadcasts are required (after start and after finishing of a task).
- The WaitingTime constitutes a source of dynamic overhead.
- Case 2 implies the execution times of a task to be at least WaitingTime units.

Rule 2:

Similar to rule 1 but without broadcast at the start of a task.
=> case 2 cannot arise
disadvantage: WaitingTime occurs almost always.

Rule 3:

case 1: $\forall$ i ((i $\neq$ x) => Ord(t_i) < Ord(t_x))
Agency(t_x) is responsible

case 2: else
Agency(t_x) is not responsible

Rule 3 is based on the idea that Agency(t_x) is responsible, if and only if t_x has the highest lexicographic order among all predecessors of t. The disadvantage of this rule is the more complex implementation. According to the former rules, all input data of t was already computed when an agency became responsible for a task t. This is usually not applicable to rule 3. Therefore, each agency maintains a list of processes for which it is responsible, but which cannot yet be started. After computation of new data sets, this list is searched for tasks which can be started at that point. Despite its complexer implementation we preferred rule 3 because of the dynamic overhead caused by the WaitingTime in rule 1 and 2.

4.3 Maintaining the local knowledge without broadcast

Among other things the local knowledge contains the information which task has been finished on which processor (the so-called "address component"). Parts of this information

are necessary to find out the location of the input data of a task to be started. In the first version of "Team Scheduling", the local knowledge was maintained via a global broadcast mechanism [RoM90]. Since broadcast constitutes a source of dynamic overhead, we have been searching for other solutions.

Note that it is not necessary to inform each agency about every task which is being finished. It is sufficient to send a message to those agencies which are responsible for the predecessors of all successors of the task (perhaps "P"). One difficulty is how to find out the identity of the agencies which are responsible for the predecessors of all successors of P.

Suppose that Agency(P) which has allocated task P wants to send the "finish" message for P to Agency(t) which is responsible for task t. Task t is one of the predecessors of a successor of P. The idea here is to use quite another agency ("A") acting as an intermediary. The trick, however, is that the identity of A = A(P, t) can be computed from the node names of P and t.

One example for function A() is given in the following formula:

(1) Transform the node names of P and t to integer (named "int(P)" and "int(t)"). If the node names are given by ASCII strings, the transformation int() could be the sum of the ASCII values of the individual characters.

(2) Compute A := (int(P) + int(t)) modulo (number of agencies in the system).

If Agency(P) does not know the identity of Agency(t), Agency(P) will act according to the following algorithm:

(1) Agency(P) sends the "finish" message to agency A and requests agency A to transmit the "finish" message to Agency(t).

(2) If agency A knows the identity of Agency(t), it will transmit the message, and the problem is solved; else, A will store the message and steps (3) to (5) are performed.

(3) Later, Agency(t) will be confronted with the similar problem of wanting to tell Agency(P) that t has been finished. If it does not know the identity of Agency(P), it must ask A for help (step (4a)); else, it performs step (4b).

(4a) Due to the message agency A has received in the course of step 1, A knows the identity of Agency(P). So A is able to transmit both of the messages, and the problem is solved.

(4b) Agency(t) sends Agency(P) a message that t has been finished.

(5) Following step (4b), Agency(P) receives the message from Agency(t) *without* the aid of A. Agency(P) sends the message that P has been finished to Agency(t) and tells A that the problem with regard to the identity of Agency(t) is now solved.

5 Test results

5.1 Evaluation of scheduling algorithms

For judging the quality of a schedule, it is necessary to know what to compare the schedule to. The most obvious way, i.e. to compare it with the optimum schedule, fails because the computation of the optimum schedule falls into the class of NP-hard problems. This means that the time needed for computation of the exact solution grows exponentially with the problem size. Therefore, only a few minor problems can be solved to optimum. To cope with this fact, bounds were looked for, the computation of which is easier than that of the exact solution. Two important lower bounds for a given dataflow graph are constituted by

the "critical path" and the "efficiency bound". The critical path is the longest path from an input node to an output node of the dataflow graph (the highest sum of execution times). Whatever scheduling algorithm is employed, the execution time of the process system cannot be shorter than the length of the critical path. The efficiency bound says that the speedup cannot be higher than the number of processors. In other words: efficiency ≤ 1. Both bounds are very important and widely applied (see for example [KaN84]). They deliver a rather good estimation in many cases, but unfortunately not in all.

In addition we applied a third bound expressed in the formula given below. Different to the two bounds described above, which provide a bound for a *given dataflow graph*, the third bound measures only the dynamic overhead of a *given schedule* and cannot judge the overall quality of a schedule. The formula computes the fictitious starting times of the processes if the given schedule could be executed on a multiprocessor system with an infinitely fast communication system and with no computation time required for the decision heuristics. In this way the formula reflects an intuitive idea of the "dynamic overhead".

If (t is an input task) $\wedge$ (t is the first task at the processor PMU(t))
    StartTime(task t) := 0
else StartTime (task t) := $\min_x$ (x fulfils condition (*) as well as condition (**))

$\forall$ p (p is a predecessor of t) =>
    x $\geq$ StartTime(p) + ExecutionTime(p) (*)
AND
$\forall$ p ((PMU(p) = PMU(t)) AND (Order(p) < Order(t))) =>
    x $\geq$ StartTime(p) + ExecutionTime(p) (**)

In this formula according to the given schedule, the expression "PMU(t)" represents the processor which is executing task t and the phrase "Order(p) < Order(t)" means that task p is executed before task t. (*) is a formal description of the critical path rule: a task t cannot be started before all predecessors are finished. (**) states that the user tasks on the same PMU are executed strictly sequentially.

5.2 Measurement of the dynamic overhead

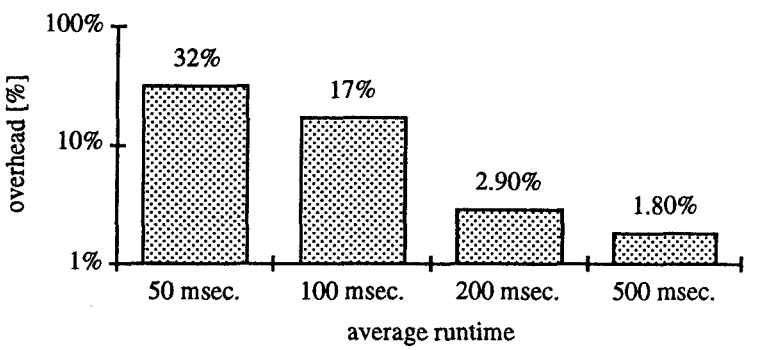

Fig. 7. Variation of execution times

For the measurements we used the multi-transputer system DAMP [BBM91], which is a multi-user system where the users allocate static partitions. More than 2000 measurments

were made using up to 28 nodes of our DAMP system configured as tori. The user-applications were modeled by randomly created artificial dataflow graphs with up to 400 task nodes with varying execution times equally distributed in fixed intervalls. The mean execution times and the observed dynamic overhead are given in Fig. 7. Very short execution times of 50 msec cause an overhead of about 32%. Tasks with medium granularity with execution times of 0.5 sec can be scheduled with an overhead of approx. 1.8%.

Multiple user tasks on one PMU

In a second test, the usefulness of multiple simultaneous user tasks executed on each processor has been analysed. The advantage of such a feature could be denied because, on the one hand, the switching of the multiple simultaneous tasks constitutes a source of dynamic overhead. On the other hand, no advantage is expected here as compared with the alternative of executing the tasks strictly sequentially.

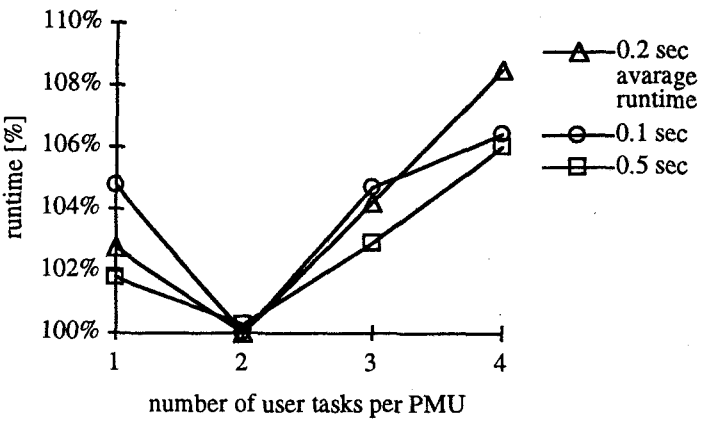

Fig. 8. Variation of the number of user tasks per PMU

In Fig. 8, we measured the execution time for a given example and configuration, and compared it with the best configuration *for this example*. We measured that two user tasks on each processor lead to minimum execution times. Further investigation disclosed the underlying reason as described below.

The allocation of tasks consumes a rather long time (in the test implementation at least some 10 msec) for performing the protocols and transmitting the data. However, most of this time the PMU is waiting idly (sometimes more than approx. 98% to 99%). If the processor has a second task it can use this time in a productive way.

Influence of statistical knowledge on the execution time

In many applications the execution times are not known exactly. One reason might be that the execution times are data-dependent. There arises the question as to the quality of the solutions for static and dynamic task scheduling under such restrictions.

To answer this question, the execution times of the tasks were replaced by estimates (random numbers). The estimates were exponentially distributed. The expectation of the random numbers was identical with the actual value. To these approximately 250 examples,

a static optimum algorithm was applied. The computed results were about 4.6% weaker than the mathematical optimum produced by the same algorithm with exactly known execution times. I.e., a static scheduler cannot produce solutions for the examples given which are better than 104.6% of the minimum schedule length. On the other hand, the dynamic scheduler suggested computed solutions of these problems which were only about 3.4% weaker than the optimum. Due to these experiences we assume that the superiority of static task scheduling is limited to special applications. Note that, different to the actual measurements given in the sections above, these numbers are computed with a simulation tool.

6 Conclusion

Agency Scheduling is a model permitting the representation of a wide variety of dynamic task scheduling algorithms including centralized and fully distributed ones as well as intermediate forms. The model consists of eight almost independent components for which examples are given. An extreme case of the model is the fully distributed "Team Scheduling" algorithm. The model has been implemented and tested using the multi-transputer system DAMP [BBM91]. Measurements with artificial loads have shown that Agency Scheduling is suitable for a wide variety of applications. For medium to large grain dataflow graphs with execution times greater than a few hundred milliseconds the measured overhead is less than 5 %.

As described in [BMM94], we have extended our scheduling system by means for fault tolerance. The basic idea is that the input data is always available on two different PMUs: the PMU which executes the task and that one which keeps the so called checkpoints. After a failure of one processor, the corresponding tasks can be easily restarted by the processors which store the checkpoints. If the checkpoints are replicated again after a failure, a subsequent second failure can be tolerated as well and fail-soft behaviour is achieved. The decision which processor gets which checkpoint under the restriction of limited memory capacity leads to a so called "Bin-Packing-Problem", which in itself is anything but trivial [CGJ87]. In our scheduling system we have tackled this problem by using a simple heuristic. Details of the fault-tolerant algorithm are given in [BMM94].

The measurements presented in this paper are based on automatically created artificial loads to cover a wide variety of load characteristics. Further measurements using real applications are currently under way. For the near future we plan to spend more effort in studying and comparing the performance of our scheduling scheme with others. Furthermore we plan to develop a graphical user environment. It will not only be used to aid the user in developing his application but will also supply help for debugging and performance evaluation. Therefor we want to use the monitoring tool DELTA-T [MaO92] also developed at our institute.

Acknowledgement

The authors wish to thank Prof. Dr. E. Maehle for his continuous support of this work.

References

[BBM91] Andreas Bauch, Reinhold Braam, Erik Maehle: DAMP - A Dynamic Reconfigurable Multiprocessor System With a Distributed Switching Network. In A. Bode (Ed.): Distributed Memory Computing. Lecture Notes in Computer Sciences, Vol. 487, Springer-Verlag 1991, pp. 495-504.

[BMM94] Andreas Bauch, Erik Maehle, Franz-Josef Markus: A Distributed Algorithm for Fault-Tolerant Dynamic Task Scheduling. Proc. 1994 EUROMICRO Workshop on Parallel and Distributed Processing, Malaga, IEEE Computer Society Press 1994, pp. 309 - 316

[Bok87] Shahid H. Bokhari: Assignment Problems in Parallel and Distributed Computing. Kluwer Academic Publishers 1987.

[CaK88] Thomas L. Casavant and Jon G. Kuhl: A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. IEEE Trans. on Software Engineering, Vol. 14, No. 2, February 1988, pp. 141-154.

[CGJ87] E. G. Coffman, Jr. and M. R. Garey, D. S. Johnson: Bin Packing with Divisible Item Sizes. Journal of Complexity 3, 1987, pp. 406-428

[CHL80] Wesley W. Chu, Leslie J. Holloway, Min-Tsung Lan, Kemal Efe: Task Allocation in Distributed Data Processing. IEEE Computer, Nov. 80, pp. 57-69.

[Hwa93] Kai Hwang: Advanced Computer Architecture. Mc Graw Hill, 1993.

[KaN84] Hironori Kasahara, Seinosuke Narita: Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing. IEEE Trans. on Computers, Vol. C-33, No. 11, Nov. 1984, pp. 1023-1029.

[LeA87] Soo-Young Lee, J. K. Aggarwal: A Mapping Strategy for Parallel Processing. IEEE Trans. on Computers, Vol. C-36, No. 4, April 1987, pp. 433-442.

[Lo88] Virginia Mary Lo: Heuristic Algorithms for Task Assignment in Distributed Systems. IEEE Trans. on Computers, Vol. C-37, Nov. 1988, pp. 1384-1397.

[LüL93] Reinhard Lüling, Burkhard Monien: A Dynamic Distributed Load Balancing Algorithm with Provable Good Performance, Proc. of the 5th ACM Symposium on Parallel Algorithms and Architectures (SPAA '93), 1993, pp. 164-173

[MaO92] Erik Maehle, Wolfgang Obelöer: DELTA-T: A User Transparent Software-Monitoring Tool for Multi-Transputer Systems. Proc. EUROMICRO 92, Microprocessing and Microprogramming, 1992, Vol. 32, pp. 245-252

[MaL86] Pauline Markenscoff, Weikuo Liaw: Task Allocation Problems in Distributed Computer Systems. Proc. Conf. on Parallel Processing, Aug. 86, pp. 953-960.

[ReF87] Daniel A. Reed, Richard M. Fujimoto: Multicomputer Networks - Message Based Parallel Processing. The MIT Press, Cambridge MA, 1987.

[RoM90] Johann Rost, Erik Maehle: A Distributed Algorithm for Dynamic Task Scheduling. In H. Burkhart (Ed.): CONPAR 90 - VAPP IV. Proc. Joint Intl. Conf. on Vector and Parallel Processing, Zürich 1990, Lecture Notes in Computer Science, Vol. 457, Springer-Verlag, 1990, pp. 628-639.

[Ros94] Johann Rost: Dynamic Distributed Task Scheduling on Multicomputers based on Dataflow Graphs. Ph.D. Thesis, University Paderborn, Germany, June 1994 (in German).