N90-23006

PARALLEL PROCESSING OF REAL-TIME DYNAMIC SYSTEMS SIMULATION ON OSCAR (Optimally SCheduled Advanced multiprocessoR)

Hironori Kasahara*, Hiroki Honda and Seinosuke Narita
Dept. of Electrical Engineering, Waseda University
3-4-1 Ohkubo Shinjuku-ku, Tokyo, 169, Japan

Abstract

This paper presents parallel processing of real-time dynamic systems simulation on a multiprocessor system named OSCAR. In the simulation of dynamic systems, generally, the same calculation are repeated every time step. However, we cannot apply the Do-all or the Do-across techniques for parallel processing of the simulation since there exist data dependencies from the end of an iteration to the beginning of the next iteration and furthermore data-input and data-output are required every sampling time period. Therefore, parallelism inside the calculation required for a single time step, or a large basic block which consists of arithmetic assignment statements, must be used. In the proposed method, near fine grain tasks, each of which consists of one or more floating point operations, are generated to extract the parallelism from the calculation and assigned to processors by using optimal static scheduling at compile time in order to reduce large run time overhead caused by the use of near fine grain tasks. The practicality of the scheme is demonstrated on OSCAR (Optimally SCheduled Advanced multiprocessoR) which has been developed to extract advantageous features of static scheduling algorithms to the maximum extent.

I. INTRODUCTION

High speed dynamic systems simulation, or solution of ordinary differential equations, has been required to simulate dynamic behaviors of various systems such as airplanes, missiles, nuclear reactors and robots, in real-time. So far, the dynamic systems simulation has generally been performed on traditional analog or hybrid computers or on general-purpose digital computers by using a simulation language like CSMP (Continuous Systems Modeling Program). However, these approaches have several problems, for example, operational accuracy and realization of non-linear functions for the analog computers and high processing cost and real-time input-output for the general purpose main-frame computers.

In an attempt to resolve these problems, the use of parallel processing techniques[13-14] has attracted much attention. In fact, various parallel processing schemes, especially parallel processing using

---

* Prof. Kasahara is also a Visiting Research Scholar of Center for Supercomputing R & D, Univ. of Illinois at Urbana-Champaign, 305 Talbot Laboratory, 104 South Wright Street, Urbana, IL 61801.

Figure 5

OPTIMUM SHAPES

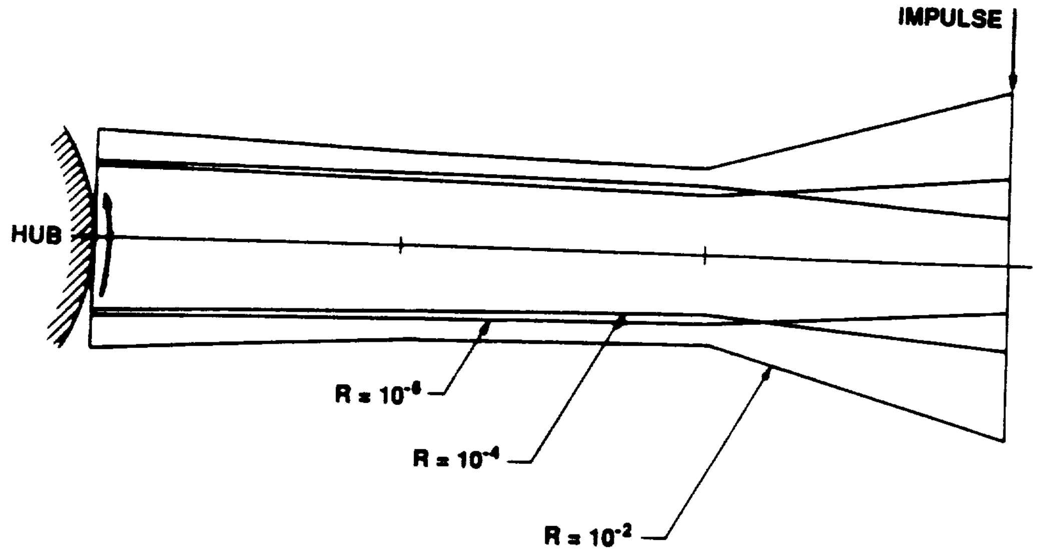
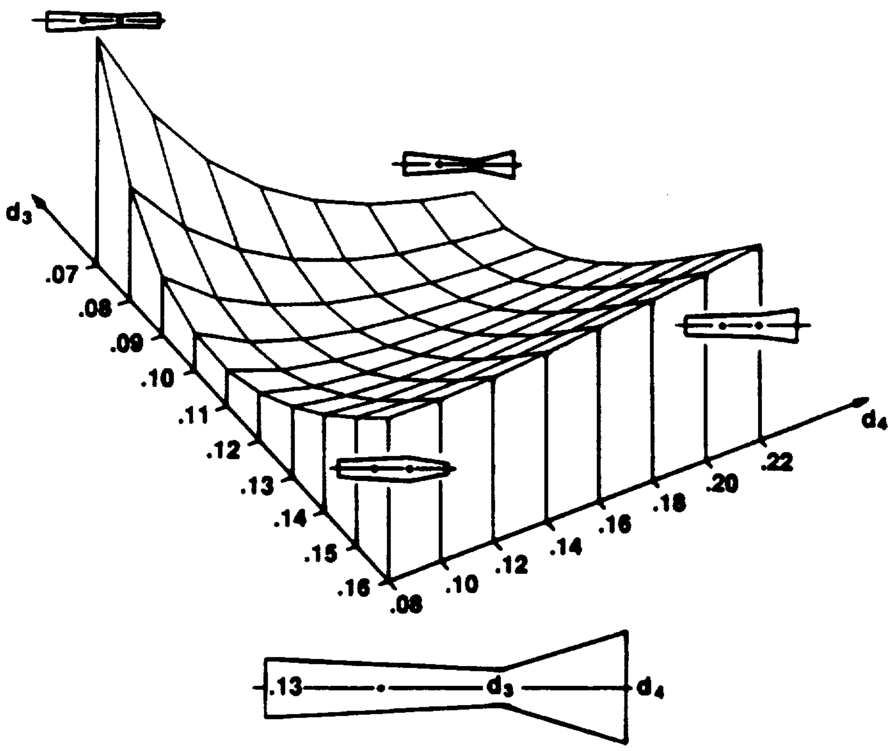


Figure 6

$J_\lambda(d_3,d_4)$ SURFACE NEAR THE MINIMUM, $\lambda = 0.7$



multiprocessor systems[14-15], have been so far proposed[1-4]. The differences among the schemes lie in the choice of task granularity and task scheduling. For example, Korn [1] and Koyama [4] employed a large task size (coarse task-grain) approach where the computation for the numerical integration of each equation in a set of first-order simultaneous differential equations was selected as a task. The generated tasks are assigned properly by the user to a relatively small number of processors. The functional distribution approach by Gilbert, et al[2], dealt with each fundamental operation (four fundamental arithmetic operations, integration and so on) as a task to be assigned to a dedicated hardware operational unit. Yoshikawa, et al[3], also adopted an approach where each fundamental arithmetic operation was assigned to one processor. The common problem left unsolved to these approaches was poor parallel processing efficiency stemming from the lack of efficient methods which allocate the generated tasks onto an arbitrary number of parallel processors in an optimal manner. This paper proposes a parallel processing scheme for the solution of the above-mentioned problem by using static minimum execution time multiprocessor scheduling algorithms[5][10] already developed by the authors for optimum task allocation. The proposed parallelizing compilation scheme consists of the following processes: task generation, optimal task scheduling, and generation of machine codes to be executed on respective processor element.

The effectiveness and practicality of the proposed scheme are demonstrated on OSCAR's processor cluster with sixteen 32-bit RISC-like processor elements which has been designed to extract advantageous features of static scheduling at compile time to the maximum extent.

II. A PARALLEL PROCESSING SCHEME USING STATIC SCHEDULING

Generally, dynamics of most continuous-time systems can be modeled by the following explicit first-order simultaneous ordinary differential equations:

$$dx_i/dt=f_i(t,x_1,x_2,\dots,x_m) \quad (i=1,2,\dots,m)$$

Therefore, the dynamics systems simulation can be regarded as the solution of the ordinary differential equations. Hence, this paper handles parallel solution of the equations using various numerical integration formulae such as Euler, Trapezoidal, 3rd- and 4th-order Adams Bashforth, 4th-order Runge Kutta and 4th-order Adams Moulton (predictor-corrector method) listed in Table 1. In applying these integration formulae, the computation required for each integration step consists of arithmetic assignment statements to evaluate the derivative of each equation and to perform numerical integration. Between consecutive iterations, there exist data dependencies[16-17] from the end of an iteration to the beginning of the next iteration. Furthermore, for real-time simulation, data input and data output are required every iteration or few iterations, namely every sampling period. Therefore, we cannot apply Do-all and Do-across techniques to parallel processing of the dynamic systems simulation which are popular parallel processing schemes for a Do loop on a multiprocessor system[18][19].

Taking into consideration these facts, in order to realize efficient parallel processing of the simulation, we must parallel process a block of arithmetic assignment statements, or a basic block, in each iteration.

However, the parallel processing of the basic block on a multiprocessor system has been thought to be very difficult since data transfer overhead and synchronization overhead are relatively large. The proposed scheme allows us to minimize these overheads and to realize efficient processing by generating optimized machine codes based on the static schedule at compilation time.

A. Task Generation

As mentioned before, in the dynamic systems simulation, we must process each iteration in parallel though we can sometimes unroll a few iterations if data input and output should be made every few iterations. In order to process the iteration in parallel, first of all, we must generate tasks with suitable granularity, which are basic units assigned to processors. As for the task granularity, several levels may be perceived: equation level, operation element level, and intermediate level. In the case of equation level granularity, the computation related to each subscript i for each numerical integration formula listed in TABLE 1 (the computation of a derivative and that of numerical integration corresponding to each formula for each variable X_i) is considered to be a task. When operation element level granularity is adopted, the computation for each derivative or for each numerical integration is subdivided into finer fundamental operation elements such as the four arithmetic operations and trigonometric functions, each of which is taken as a task and allocated to the processors (fine granularity). In the intermediate task granularity, several floating point operations are combined to form a task. For instance, when Van der Pol's equations

$$dx_1/dt = x_2$$
$$dx_2/dt = \varepsilon x_2 - x_1{}^2 * \varepsilon x_2 - x_1$$

is decomposed into fairly small intermediate-level tasks, three multiplication tasks, two subtraction tasks, and two integration tasks (including several floating point operations) are generated. Fig.1 depicts the block diagram representation of the seven tasks, with data dependencies explicitly shown.

There exists no general rule for determining the best task granularity applicable to all kinds of dynamic systems. When parallel processing is performed on a multiprocessor system with little data transfer and synchronization overheads among processor elements, the operation element level granularity is known to be most advantageous to achieve minimum processing time because parallelism can be exploited to the maximum extent. For a large-scale problem (the order of simultaneous equations is very high) or a multiprocessor system with poor data transfer capabilities, however, the operation element level granularity does not always give the best performance. In other words, much attention must be paid to such factors as processor speed, interprocessor data transfer speed, size and parallelism inherent to the problem in hand, and complexities of scheduling mechanisms (both software and hardware) [7]. Namely, we must choose the best granularity for each problem and each multiprocessor system. For this reason, the proposed parallel processing scheme provides with two methods for the input of simulation source programs. The first method employs a simplified simulation language shown in Fig.2, which allows direct input of mathematical equations. The user can specify arbitrary task granularity from the operation element level to the equation level. The second method facilitates the input of block diagram representations such as those employed for analog computer. As shown in Fig.1, each operational element of analog computer (adder, integrator, etc.) can be taken as a task, to realize near fine granularity. Medium granularity can also be dealt with by combining automatically several tasks with near fine granularity. (This process is referred to as task fusion). In what follows, emphasis will be placed on the case of near fine granularity, namely the finest granularity that can be treated by use of the proposed scheme on a multiprocessor system named OSCAR mentioned later.

Next, the proposed parallel processing scheme analyzes precedence relations caused by data dependencies among the generated tasks and represents the task precedence relations by a task graph like Fig. 3 which is a directed acyclic graph (DAG). The precedence constraints represent the restrictions existing among tasks regarding the execution order of tasks. The existence of task i precedent to task j means that the execution of task j cannot be initiated before the completion of task i. The precedence relation can be examined by the data flow analysis among tasks. When the data flow analysis is made, the output variable of each integration task is treated as an initial value. Each node in the task graph stands for a task and an arc between a pair of nodes for the precedence constraint. Nodes 0 and 8 are not actual nodes but dummy nodes introduced for the sake of convenience. They represent the entry node and the exit node, respectively. The figure beside each node represents the estimated processing time of the corresponding task. Since the actual processing time does not usually take on a fixed value but varies with the data to be processed, the average value or the worst-case value is employed as the input[7], which is used in the scheduling algorithms to be described in the subsequent section. When the average value is used for each task, the resultant schedule gives the minimum value of the average processing time of the task set. Similarly, when the worst-case value is used, the worst-case processing time is minimized. However, OSCAR, which is a target machine in this paper, can execute all instructions including a few floating point operations in one clock by employing RISC like processor. Therefore, we don't have the above mentioned problem on OSCAR, a compiler can estimate accurate processing time of each task.

Once a task graph is generated, the minimum possible processing time achieved by parallel processing of the tasks can be estimated as the critical path length t_{cr} of the task graph. In Fig.3, the critical path is shown by double-line segments.

An unique task graph can also be generated by following simple procedures in the case of the block diagram input mode. The task graph shown in Fig.3 represents the computation in one integration step when the tasks are generated in the size of near fine granularity and the numerical integration method employed is Euler, Trapezoidal or 3rd- or 4th-order Adams Bashforth. The integration task involves computation specific to each numerical integration method. When the 4th-order Runge-Kutta method is employed, k_1 through k_4 need to be evaluated, and the computation described by this task graph is repeated four times or the expanded task graph involving the computation repeated four times is processed for each integration step. In the former case, the content of each integration task to be processed differs with the iteration count in order to evaluate k_1 through k_4 and their weighted average. Similarly, when a predictor-corrector method such as the 4th-order Adams-Moulton is used, the task graph is computed twice or the expanded task graph to represent the unrolled computation is processed for each integration step. In the former case, the computation corresponding to the predictor of the integration task is performed first, followed by the computation for the corrector.

As mentioned earlier, the task graph shown in Fig.3 represents the case where the tasks are generated in the size of near fine granularity. When coarse granularity at the equation level is employed for task generation, the portion surrounded by the dashed lines becomes a task. Also, in the case of fine granularity, the portion of each integration task is replaced by a subgraph generated by subdividing it into the operation element level.

It should be mentioned here that parallel processing scheme proposed in this paper is so designed that the tasks generated in either fine or near fine granularity level can be fused automatically without sacrificing much parallelism. As a simple example, when there exist a pair of successor task (son node) with only in-edge and the predecessor task (father node) with only one out-edge, the two tasks are fused into a single task. Even such an easy task fusion technique allows the optimization of resister utilization and avoids unnecessary data transfer for more efficient parallel processing.

B. Scheduling Algorithms

In order to process the set of tasks on a multiprocessor system efficiently, the assignment of tasks onto the parallel processors and the execution order among the tasks assigned to the same processor must be determined optimally. The problem which determines the optimal assignment and execution order can be treated as the traditional multiprocessor scheduling problem of which the objective function is the minimization of the parallel processing time or schedule length [5][8]. To state formally, the scheduling problem is to determine such a nonpreemptive schedule that the execution time or the scheduling length be minimum, given a set of n computational tasks T=(T1, ... ,Tn), precedence constraints among the tasks and n processors with the same processing capability. This problem, however, has been known as a "strong" NP-hard problem [9]. In other words, unless P=NP, it is impossible to construct not only a pseudo-polynomial time optimization algorithm but also a fully polynomial time approximation scheme. With this fact in mind, the authors have successfully constructed a heuristic algorithm named CP/MISF and an efficient practical algorithm called DF/IHS [5]. The former algorithm can provide very precise approximate solutions quite rapidly because of its very low time complexity. The latter algorithm can obtain optimal solutions or approximate solutions with guaranteed accuracies from optimal solutions by combining CP/MISF and depth-first search. In what follows, the two algorithms are explained very briefly. For further details, the reader is referred to the literature [5].

1) CP/MISF(Critical Path/Most Immediate Successors First) Method

This method essentially is a kind of list scheduling algorithms.

step.1 Determine the level l_i for each task. The l_i is the longest path from N_i to the exit node.

step.2 Construct the priority list in the descending order of l_i and the number of immediately successive tasks.

step.3 Execute list scheduling [8] on the basis of the priority list.

Since the list scheduling may be regarded as a method to construct the schedule for the case where a set of tasks are processed in parallel in the data-driven manner considering the priority assigned to each task, it can be easily extended to dynamic scheduling at run time.

Furthermore, the list scheduling can be also modified to eliminate unnecessary data transfer among processors. In the modified algorithm CP/DT/MISF method[10], when the tasks with the same priority are allocated to a processor, a task is allocated to the processor which needs the minimum data transfer to execute the task. This simple modification significantly decreases the data transfer overhead for the multiprocessor system with poor data transfer performance.

Its average performance was evaluated for a total of over nine thousand test cases by comparing the CP/MISF solutions with the lower bound function[11]. Optimal solutions were obtained for 67 percent of the cases tested. Approximate solutions with errors of less than 5 percent were obtained for 87 percent of the cases and those with errors of 10 percent for 98.5 percent of the cases. The worst-case performance of CP/MISF, i.e., the error of the worst-case solution t obtained by CP/MISF from the true optimal solution t_{opt} is given by

$$(t-t_{opt})/t_{opt} \leq 1/m \quad [5].$$

In addition, the time complexity of CP/MISF is $O(n^2+mn)$. For problems with about one thousand tasks, it only takes a few ten seconds on a HITAC M280H system. In summary, CP/MISF is suitable for the solution of very large problems with hundreds or even thousands of tasks.

2) DF/IHS (Depth First/Implicit Heuristic Search) Method
DF/IHS is an optimization/approximation algorithm to determine schedules (solutions) which are always more precise than those by CP/MISF. The method combines CP/MISF and depth-first search in a special manner and reduces markedly space complexity (memory requirements) and average computation (search) time. It is so practical and powerful that optimal schedules for most large-scale problems involving a few hundred tasks for a total of some ten parallel processors can be determined in several seconds to one hundred seconds on an M280H. Optimal solutions could be obtained for 75% of the test problems where the upper limit of search time was set to 180 seconds [5]. The effectiveness of DF/IHS may be recognized by considering the fact that use of dynamic programming could provide optimal solutions for small problems with less than 40 tasks even for two parallel processors. In the case of parallel processing on a limited number of processors, it is known that there exist such task graphs that the minimum processing time cannot be attained by data driven execution or the list scheduling [12]. For these task graphs, use of DF/IHS can determine the optimal schedule that gives rise to the minimum processing time by forcing some processors to be idle for a certain time period. This fact implies the possibility of more efficient parallel processing than data flow machines. In summary DF/IHS is very useful when CP/MISF fails to obtain an accurate solution for problems with several hundred tasks.

C. Machine Code Generation

For the efficient execution on an actual multiprocessor system, the optimal machine codes tailored to the given system must be generated by using the scheduled results. The scheduled results give us the information about tasks to be executed on each processor element, the execution order of tasks on the same processor element, the rough estimates of waiting time of the tasks which wait for the data from other tasks assigned to other processors, the tasks to be synchronized and so on. Therefore, we can generate the machine codes for each processor by putting together the codes for the tasks assigned to the processor and attaching the codes for synchronization and data transfer among processors. The "version number" method is used for the synchronization among tasks. The version number corresponds to the number of times of iterations or integration steps. Each "writer" task updates the version number on the common memory to the number of current integration step for itself after it finishes writing the shared data. And each "reader" task checks the version number if the number is the same as the number of current integration step to the reader task. All processor elements (PE's) have the same version numbers during one integration step and update or increase the number at the end of the integration step. Updating the version number on each PE by respective PE's allows us to eliminate the need to update the version number (or to reset a flag used in test & set or semaphore) attached to each shared data on a common memory when the next integration step is started. Therefore, the version number method can minimize the frequency of access to the common memory for task synchronization in this application.

We can also optimize the codes to minimize various processing overheads by making full use of all information which is obtained as the result of static scheduling. For example, the information about task assignment and execution order allows the optimized use of the registers of the processor when the tasks allocated to the same processor exchange data. The optimal use of registers reduces the processing time markedly. The knowledge about the estimated waiting time helps prevent the degradation of data transfer performance caused by frequent bus access to check the existence of the required data (data level synchronization) by the waiting task. In other words, if it is estimated that the task must wait the data for a long time, the frequency to check a flag on a common memory is reduced. In addition, we can minimize the synchronization overhead by carefully taking into consideration the information about the tasks to be synchronized, the task assignment and the execution order. For example, let tasks A, B and C be allocated to processor 1 and tasks D and E to processors 2 and 3 respectively as shown in Fig.4 and data among the tasks be transferred via a common memory. Then task B does not need to check the flag which shows the completion of task A because both tasks are allocated to the same processor. Task E has no need to check the flag which indicates the completion of task D because the termination of task D has already been confirmed by task C or B.

In the parallel processing scheme, the transfer of output data of integration tasks is not represented on a task graph since data flow analysis is performed on the assumption that output data of the integration tasks has been given as initial values. In actual processing, however, those data must be transferred to several tasks allocated on other PE's between the end of an integration step and the beginning of the next integration step since, during one integration step, all the tasks except the integration tasks use the output data of the integration tasks gênerated in the previous integration step. The data transfer at one time causes bus congestion. In order to prevent the bus congestion, two copies of machine codes for each PE which are assigned different data storages are generated and executed alternatively for every integration step. Generating the two copies of codes allows each integration task in a copy of codes to write or transfer its output data, as soon as it completes execution, onto a data storage assigned for the next integration step or another copy of codes. In other words, it allows distributed bus access and also to eliminate data synchronization to check the completion of the integration tasks because the output data of the integration tasks has already been transferred before the end of each integration step.

The optimal machine codes for each PE generated in the way mentioned above are loaded to the local instruction memory of each processor element and executed asynchronously. The four steps of the proposed parallel processing scheme described in this section can be performed automatically by a special purpose compiler.

III. PERFORMANCE EVALUATION ON OSCAR

This section discusses the performance evaluation of the proposed parallel processing scheme on a prototype multiprocessor supercomputing system named OSCAR being developed by the authors.

In the following, as an example of parallel processing of the practical dynamic systems simulation for evaluating the performance of the proposed scheme on OSCAR, dynamics simulation of a hot strip mill control in a steel making plant is treated. The simulation program can be represented by a block diagram shown in Fig. 5. In this example, near fine task granularity has been chosen in which each integration task consists of several floating point operations and the other tasks consist of only one floating point operation. By the task generation method using near fine granularity, fifty-one tasks involving nine integration tasks were generated. Fig. 6 is a task graph generated from Fig.5 automatically by a special purpose compiler.

OSCAR is a hierarchical multiprocessor system which has a plurality of processor clusters as shown in Fig.7. Its goal is to realize, by the combined use of static scheduling and dynamic scheduling, efficient parallel processing of Fortran programs and a variety of applications including those which have so far been difficult to process efficiently because of a lot of scalar assignments involved.

One processor cluster(PC) hardware has already been completed. On the PC, various parallel processing application will be implemented. The PC involves sixteen processor elements, three common memories, a local control processor and three shared buses. Each PE consists of a 32-bit custom-made RISC-like processor with 64 general purpose registers which executes all instructions including a few floating point operations in one clock (clock:200ns), a 256-KW local data memory, a 2-KW two-port memory to communicate with other PE's, two banks of 128-KW instruction memory and a DMA controller. The DMA controller realizes high-speed transfer of a block of data to the common memories and the two-port memories of other PE's and dynamic loading of a set of instruction codes from the common memories to one of the instruction memory banks during execution. The reduced instruction set and the one-clock-execution of the all instructions make the estimation of task processing time for the scheduling easy and accurate. For interprocessor communication, three types of data transfer modes are provided such as broadcast mode, direct data transfer mode to the two-port memory of another PE or indirect data transfer mode via a common memory. Each mode can be used for both single word data transfer and block data transfer. Each common memory accepts simultaneous accesses from three buses. The data transfer speed of the three buses totals to 60MByte/s.

When we operate one PC of OSCAR, a Unix-based workstation is used as the host computer which generates machine codes for each PE by using static schedule providing the minimal processing time and downloads the codes to each PE. In the generated codes, bus access timing by PE's, data transfer modes and use of 64 registers employed to exchange data among tasks assigned on the same PE are optimized. In addition, redundant task synchronization is also eliminated as mentioned before. An timing chart representing execution of the machine codes is shown Fig.8. This chart can be regarded as a precise simulated result of actual parallel processing on OSCAR. In the figure, for PE3, characters such as "LD30","25R","wait"," PE5" and so on are written. These characters mean to load input data for task 30 from a local data memory to registers, execute task 30 and keep its result in registers, and wait for a while to directly transfer the output data of task 30 to PE5. At that time, PE3 waits for bus access since PE1 is accessing bus for data broadcasting. In OSCAR, another PE cannot access the busses while a PE is broadcasting data. Furthermore "U26 51"," PE2","WAIT","FC44" and "38R" represent to execute task 51 by using output data of task 26 on registers, transfer its output data to PE2, wait for output data of task 44 from PE5, check a flag showing completion of data transfer from task 44, execute task 38 and keep its output data on a register.

Fig.9 shows the measured parallel processing time on OSCAR (solid lines) and simulated parallel processing time (dotted lines and chained lines) of 51 tasks in Fig.6. In this example, 4th-order Adams-Bashforth method was used. The measured processing time on OSCAR of the near fine granularity tasks was reduced from 108.7 us for one PE to 37.2 us (1/2.92) for seven PE's. Next, the task fusion technique which generates a coarser granularity task by combining several tasks in order to reduce data transfer overhead with the minimum loss of parallelism is evaluated. As a simple example, those tasks surrounded by dotted lines in Fig. 6 can be fused and twenty-two medium granularity tasks are generated automatically. Processing time of the medium granularity tasks (after task fusion) decreases from 105.8 us for one PE to 36.8 us (1/3.01) for seven PE's. From the results, it has been confirmed that the determination of the most suitable task granularity is very important and that the automatic task fusion is useful.

The two dotted lines show the simulated processing time. It is clear from the figure that there exists little difference between the measured processing time and the simulated processing time or an execution image of machine codes generated by using static scheduling. In the light of this fact, we can conclude that the generation of the precisely optimized machine code using static scheduling is very useful for OSCAR.

The processing time shown above, however, represents the degraded performance of OSCAR since OSCAR is still in a stage of operation testing. Though OSCAR can normally transfer two words data in 5 clocks, the processing time were measured in a degraded operating condition where two words data transfer takes 9 clocks. Therefore data transfer overhead will be reduced by half in the normal operating condition. The chained lines in Fig.9 show the precisely simulated processing times in the normal condition for the near fine granularity before task fusion and the medium granularity after task fusion. The processing time after task fusion decreases from 104.8 us for one PE to 28.8 us for seven PE's (1/3.64). From the experiment mentioned above, it has been confirmed that OSCAR's architecture, especially one clock execution of all instructions and three types of data transfer modes, allows us to efficiently parallel process the dynamic systems simulation by extracting the advantageous features of static scheduling to the maximum extent.

V. CONCLUSIONS

In this paper, the authors have proposed a parallel processing scheme of the dynamic systems simulation using static optimal multiprocessor scheduling algorithms and shown that the scheme allows us to realize efficient parallel processing on OSCAR which has been designed to extract the advantageous features of static scheduling to the maximum extent. More precisely speaking, the special purpose compiler for OSCAR using the proposed scheme can generate suitable granularity tasks, the minimal execution time schedule and optimized machine codes for each processor in which data transfer and synchronization overheads are minimized and the registers on each processor are used optimally.

Furthermore, it has been confirmed that the architectural support in OSCAR for a parallelizing compiler using static scheduling is very useful. The authors are planning to develop a practical dynamic systems simulator using OSCAR which can simulate dynamics of flying objects like airplanes and missiles, nuclear reactors, robot systems and various industrial plants.

REFERENCES

[1] G. A. Korn, "Back to Parallel Computation:Proposal for a Completely New On-line Simulation System Using Standard Minicomputers for Low-cast Multiprocessing," Simulation , Vol.19, pp. 37-44 (Aug.1972).
[2] E. O. Gilbert, and R. M. Howe, "Design Consideration in a Multiprocessor Computer for Continuous System Simulation," Proc. National Computer Conf., pp. 385-393, AFIP Press, Reston (1978).
[3] R. Yoshikawa, T. Kimura, Y. Nara and H. Aiso, "A Multi-microprocessor Approach to a High-speed and Low-cast Continuous-system Simulation," Proc. National Computer Conf., pp. 931-936, AFIP Press, Reston (1977).
[4] S. Koyama, K. Makino, N. Miki, Y. Iino and Y. Iseki, "On the Parallel Processor Array of Hokkaido University High-speed System Simulator "Hoss"," Proc. 8th IFAC World Cong., pp. 1715-1720, Pergamon Press, Oxford (1981).
[5] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," IEEE Trans. Comput., Vol.c-33, pp. 1023-1029 (Nov.1984).
[6] H. Kasahara and S. Narita, "Parallel Processing of Robot-arm Control Computation on a Multimicroprocessor System," IEEE J. of Robotics and Automation, Vol.RA-1, pp. 104-113 (June 1985).
[7] H. Kasahara and S. Narita, "An Approach to Supercomputing Using Multiprocessor Scheduling Algorithms," Proc. IEEE First International Conf. on Supercomputing Systems, pp. 139-148 (Dec.1985).
[8] E. G. Coffman, "Computer and Job-shop Scheduling Theory," Wiley, New York (1976).
[9] M. R. Garey and D. S. Jonson, "Computers and Intractability : A Guide to the Theory of NP-Completeness," Freeman, San Francisco (1979).
[10] H. Kasahara and S. Narita, "Load Distribution among Real-time Control Computers Connected via Communication Media," Proc. IFAC 9th

Triennial World Congress, pp. 2695-2700 (1984).

[11] E. B. Fernardez and B. Bussel, "Bound on the number of processors and time for multiprocessor optimal schedules," IEEE Trans. comput., Vol.c-22, pp. 745-751, (Aug. 1973).

[12] C. V. Ramamoothy, K. M. Chandy and M. J. Gonzalez Jr., "Optimal scheduling strategies in a multiprocessor system," IEEE Trans. comput., Vol.c-21, pp. 137-146, (Feb.1972).

[13] R.W. Hockney and C.R. Josshope,"Parallel Computers 2: Architecture, Programming and Algorithms," Adam Hilger, 1988.

[14] K.Hwang and F.A.Briggs, "Computer Architecture and Parallel Processing," McGRAW-HILL, 1984.

[15] D.D.Gajski and Peir,"Essential Issues in Multiprocessor Systems," IEEE Computers,Vol.C-18,No.6,pp.9-27,Jun.1985.

[16] U.Banerjee, Dependence Analysis for Supercomputing, Kluwer Academic Publisher,1988

[17] D.A.Padua,D.J.Kuck and D.H.Lawrie,"Highspeed Multiprocessors and Compilation Techniques,"IEEE Trans. Comput. Vol.29,No.9,Sep.,1980.

[18] D.J.Padua, and M.J.Wolfe,"Advanced Compiler Optimizations for Supercomputers," C.ACM, Vol.29,No.12,pp.1184-1201,Dec.1986.

[19] C. Polychronopoulos,"Parallel Programming and Compilers," Kluwer Academic Publishers, 1988.

TABLE I. NUMERICAL INTEGRATION METHODS

Trapezoidal	$X_{i,n+1}=X_{i,n}+h\ (3\dot{X}_{i,n}-\dot{X}_{i,n-1})/2$ Where $X_{i,n}=f_i\ (t_n, X_{1,n}, \cdots, X_{m,n})$
4th_Order Runge Kutta	$X_{i,n+1}=X_{i,n}+(k_{1,i}+2k_{2,i}+2k_{3,i}+k_{4,i})/6$ $k_{1,i}=h f_i\ (t, X_{1,n}, X_{2,n}, \cdots, X_{m,n})$ $k_{2,i}=h f_i\ (t+h/2, X_{1,n}+k_{1,1}/2, X_{2,n}+k_{1,2}/2, \cdots, X_{m,n}+k_{1,m}/2)$ $k_{3,i}=h f_i\ (t+h/2, X_{1,n}+k_{2,1}/2, X_{2,n}+k_{2,2}/2, \cdots, X_{m,n}+k_{2,m}/2)$ $k_{4,i}=h f_i\ (t, X_{1,n}+k_{3,1}, X_{2,n}+k_{3,2}, \cdots, X_{m,n}+k_{3,m}/2)$
4th_Order Adams Moulton	$X^p_{i,n+1}=X^c_{i,n}+h\ (55\dot{X}^c_{i,n}-59\dot{X}^c_{i,n-1}+37\dot{X}^c_{i,n-2}-9\dot{X}^c_{i,n-3})/24$ $X^c_{i,n+1}=X^c_{i,n}+h\ (9\dot{X}^p_{i,n+1}+19\dot{X}^c_{i,n}-5\dot{X}^c_{i,n-1}+\dot{X}^c_{i,n-2})/24$

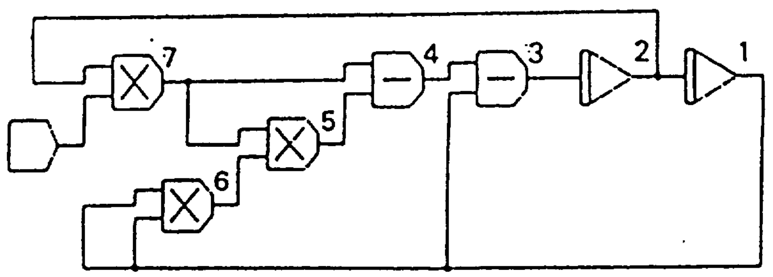


Fig.1 Block diagram for Van der Pol eq..

```
begin
   a=integral(b,0.01);   (1)
   b=integral(c,0.01);   (2)
   c=d-a;                (3)
   d=g-e;                (4)
   e=f*g;                (5)
   f=a*a;                (6)
   g=b*1                 (7)
end.
```

Fig.2 Assignment statements for Van der Pol eq..

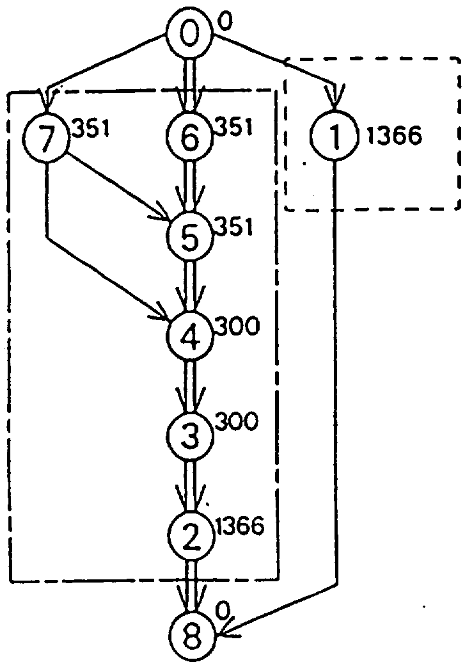


Fig.3 Task graph for Van der Pol eq.

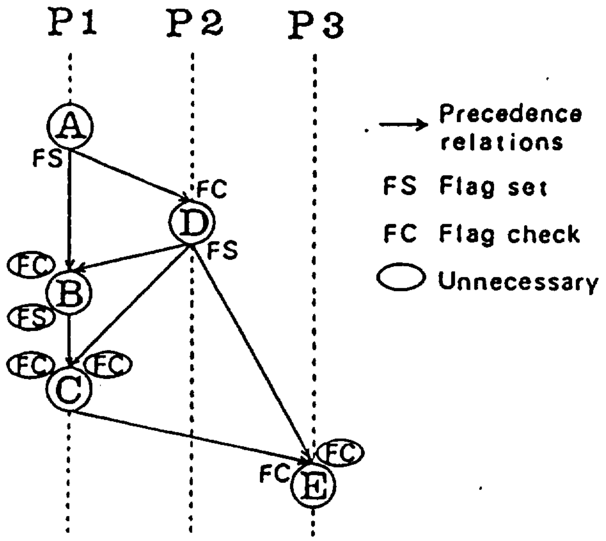


Fig.4 Minimization of synchronization overhead.

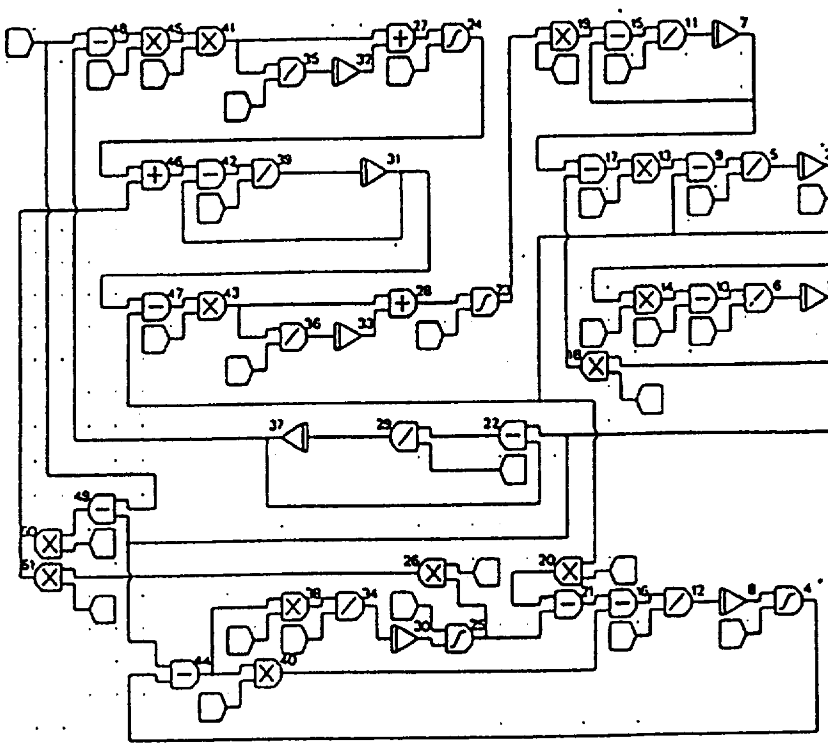


Fig.5 An example of block diagram.

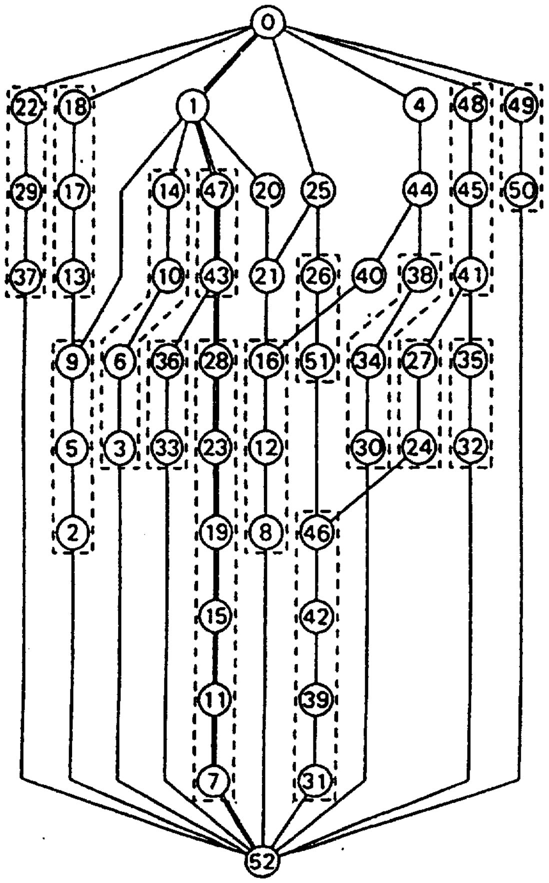


Fig.6 Task graph for Fig.5.

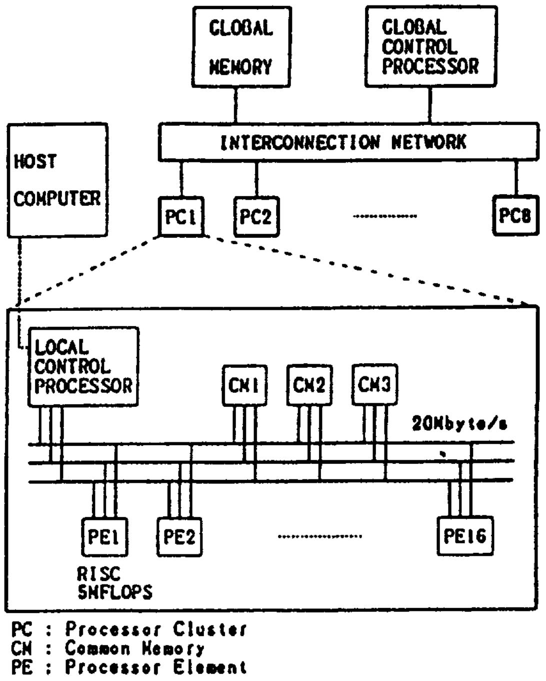


Fig.7 OSCAR (Optimally SCheduled Advanced multiprocessoR)

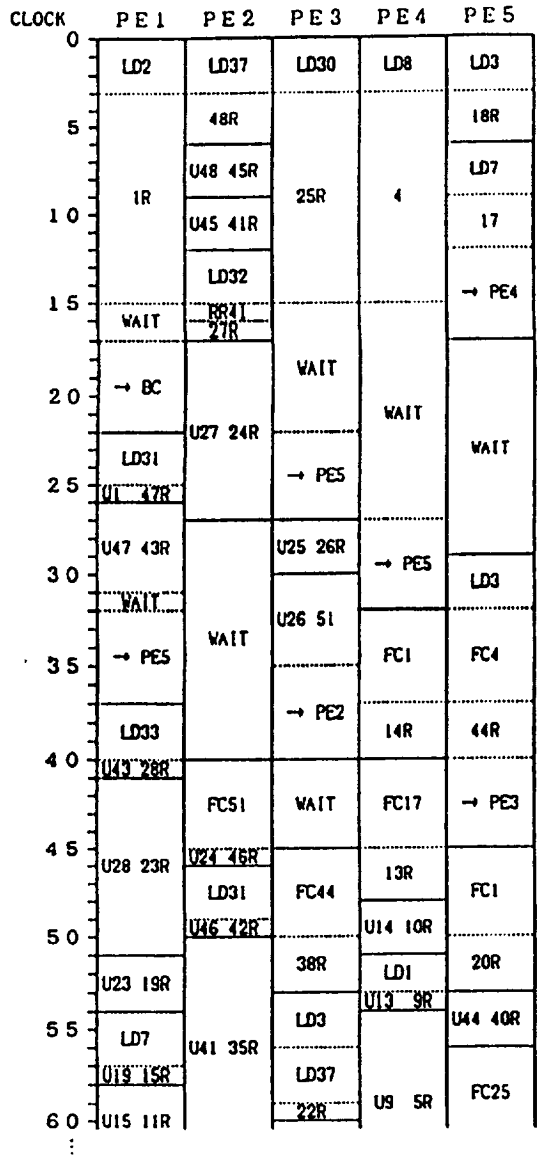


Fig.8 Execution image of machine codes on OSCAR.

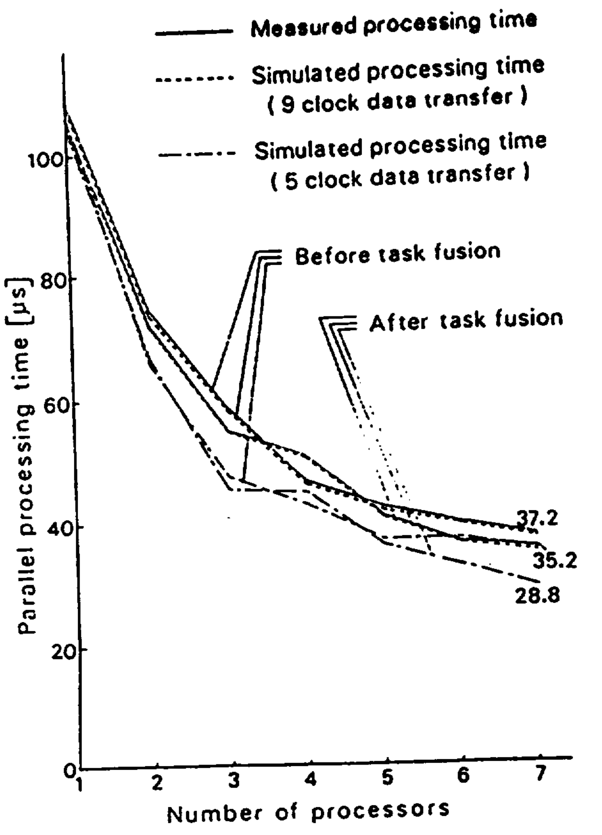


Fig.9 Parallel processing time measured on OSCAR and simulated parallel processing time.