計算機アーキテクチャ 83−15
(1990. 7. 18)

PARALLEL PROCESSING SCHEME OF THE SOLUTION OF STIFF NONLINEAR ORDINARY DIFFERENTIAL ALGEBRAIC EQUATIONS ON OSCAR

Wichian PREMCHAISWADI, Hiroki HONDA, Hironori KASAHARA and Seinosuke NARITA
Dept. of Electrical Engineering, Waseda University
3-4-1 Ohkubo Shinjuku-ku, Tokyo, 169, Japan

ABSTRACT

This paper proposes a parallel processing scheme of variable-step and variable-order implicit integration algorithm, Backward Differentiation Formula (BDF), for solving systems of stiff nonlinear differential-algebraic equations. The BDF method composes of processes, namely, transformation of nonlinear ODE to nonlinear algebraic equations, solution of nonlinear algebraic equations by Newton-Raphson method and computation of the next step size and order. The effectiveness and practicality of the proposed scheme were successfully tested on an actual multiprocessor system OSCAR.

OSCAR上でのスティッフ微分方程式求解の並列処理

Wichian PREMCHAISWADI　本多 弘樹　笠原 博徳　成田 誠之助
早稲田大学理工学部電気工学科

概要

この論文では可変ステップ・可変オーダのインプリシットな積分法であるBDF法を用いた、スティッフな非線形常微分方程式求解の並列処理手法を提案する。このBDF法は非線形微分方程式から非線形方程式への変換、Newton-Raphson法を用いた非線形方程式の求解、次積分ステップ・オーダの決定、という部分からなる。本手法の有効性と実用性はマルチプロセッサシステムOSCAR上で検証される。

1. Introduction

Many dynamic system simulations involve the solution of a set of stiff nonlinear differential-algebraic equations which require very high computing power to achieve the required performance. To obtain high computing power, the use of parallel processing [1-2] has attracted much attention. Various parallel processing schemes for solving a set of differential equations have been proposed [3-8]. These approaches can be divided into two basic categories: namely parallelization of the well-known sequential integration algorithms [3-5] and development of parallel algorithms [6-8]. This paper concerns only the former one because comparing to sequential integration algorithms, most parallel algorithms provide poor stability characteristics [9]. For systems which are moderately stiff this is a concern [4]. Among parallelization of sequential integration algorithms that have been proposed, the differences lie in the choice of a task size. For example, Yura[3] and Franklin[4] used a large task size level or equations segmentation level. Yoshikawa [5] used a small task size level where each fundamental arithmetic operation was assigned to one processor. The common difficulty left unsolved of these approaches was the lack of methods which allocate generated tasks onto an arbitrary number of processors in an optimal manner. The other problem is the use of integration methods which are not suitable for stiff problems because solution of stiff differential equations require implicit methods. In general, implicit techniques result in systems of nonlinear algebraic equations to be solved at each consecutive time level. Between consecutive iterations, there exist data dependencies [10-11] from the end of an iteration to the beginning of the next iteration. These data dependencies are more complicated when varying order and step size integration method is employed. Taking into consideration these facts, parallel processing can be carried out in a block of arithmetic assignment statements or basic block. 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.

This paper proposes the parallel processing scheme of the solution of stiff nonlinear differential equations. The implicit backward differentiation formula (BDF)[12] is used to transform nonlinear ODE to nonlinear algebraic equations. The generated nonlinear algebraic equations is solved by Newton-Raphson method which involve solution of linear equations. The proposed scheme parallelizes full parts of BDF in statement level or relative fine grain task. The scheme consists of the following processes: task partitioning and generation of parallel intermediate codes, block partitioning, data flow analysis and task graph representation, task scheduling and machine code generation.

2. Solution of Differential-Algebraic Equations

This section describes very briefly the use of backward differential formula algorithm for solving a system of differential-algebraic equations, (For further details, the reader is advised to refer to the literature [12]). The scheme uses the backward differential formula for discretization in time domain and the Newton-Raphson method to solve the resulting nonlinear algebraic equations. The system of nonlinear differential equations is usually described by a set of differential-algebraic equations

$$f(x,\dot{x},t) = 0 \qquad (1)$$

where f and x are vectors. Suppose the solution x(t) of Eq.(1) had been found at $t=t_n$, $t=t_{n-1}$,..., and t_{n+1-k}, where the step size $h_j=t_{j+1}-t_j$ need not be uniform. If $x(t_j)$ is denoted by x_j, the solution x_{n+1} of Eq.(1) at $t=t_{n+1}$ must satisfy

$$f(x_{n+1},\dot{x}(t_{n+1}),t_{n+1}) = 0 \qquad (2)$$

If x_{n+1} denotes the approximate value of $\dot{x}(t_{n+1})$, the BDF of order k is given by

$$\dot{x}_{n+1} = -\frac{1}{h}\sum_{i=0}^{k}\alpha_i x_{n+1-i} = g(x_{n+1}) \qquad 1 \le k \le 6 \qquad (3)$$

Substituting Eq.(3) into Eq.(2) obtains

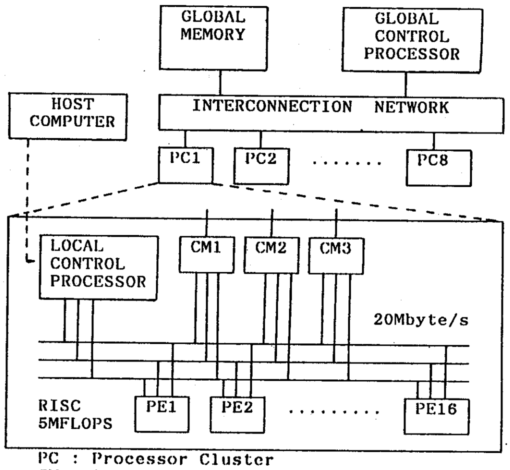
$$f(x_{n+1},g(x_{n+1}),t_{n+1}) = 0 \qquad (4)$$

Eq.(4) is now a system of nonlinear algebraic equations in term of the unknown variable x_{n+1}. Hence, the solution x_{n+1} in Eq.(1) can be found by solving Eq.(4) using the Newton-Raphson algorithm. The solution of nonlinear algebraic equations involve solution of sparse linear equations to be solved at each iteration. In solving sparse linear equations, Ax = b, the symbolic code generation method [13] has been used in LU decomposition and back and forward substitution. The method of generates codes for these calculation beforehand provide explicit data dependencies which can be identified by the use of data flow analysis.

When the Newton-Raphson iteration converges to the solution of current time, local truncation errors are estimated. If the errors are within some tolerance, the next step size and order are computed. In other case, the step size is cut to a smaller value and the analysis is repeated.

3. Architecture of OSCAR

This section describes the architecture of the multiprocessor system OSCAR [14] used in the implementation. OSCAR is a hierarchical multiprocessor system which has a plurality of processor clusters as shown in Fig.1. One processor cluster(PC) involves up to 16 processor elements(PEs), 3 common memories, a local control processor and 3 shared buses. Each PE consists of a 32-bit custom-made RISC-like processor with 64 general purpose registers which executes all instructions (about 90 instructions in one clock (200ns)) including a few floating point operation, as well as a 256-kw local data memory, a 2-kw dual port memory to communicate with other PEs, two banks of 128-kw instruction memory and a direct memory access (DMA) controller. The one-clock execution of all instructions facilitates precise estimation of task processing time needed for task scheduling. For interprocessor communication, three types of data transfer modes. Namely, broadcast mode, direct data transfer mode to a dual port memory of another PE and indirect data transfer mode via a common memory which accepts simultaneous accesses from three buses are provided. The data transfer speed of the three buses totals to 60 MByte/s.



PC : Processor Cluster
CM : Common Memory
PE : Processor Element

Fig.1 Architecture of OSCAR

4. Parallel Processing Scheme

This section describes the process to achieve parallel processing of the procedure in section 2 on a multiprocessor system. The scheme composes of five steps: (1) task partitioning and parallel intermediate code generation (2) block partitioning, (3) data flow analysis and task graph representation, (4) task scheduling, and (5) machine code generation. The details are as follows:

4.1 Task Partitioning and parallel code generation

In order to solve a problem in parallel processing, a solution process has to be decomposed into tasks (the unit assigned to the processor elements) in such a way that parallelism is extracted to the maximum extent and the overhead related with data transfer and synchronization is minimized. There exists no general rule applicable to all applications for the best selection of task granularity. Attention must be given to such factors as the ratio of processor speed to interprocessor data transfer speed, the size and parallelism inherent to the problem in hand, and complexities of scheduling mechanisms [15]. In other words, the task granularity must be chosen prudently with careful consideration given to the multiprocessor system to be used.

In the present application, the statement level granularity or relatively fine granularity has been chosen taking into account of the processing capability and data transfer speed of the multiprocessor system employed.

For specifying an efficient task size and also ease of implementation on OSCAR, a special purpose parallel intermediate language is developed. The language has a format as shown in Fig.2. An operator can have as many operands as needed. Each task is separated by an assignment operator(:=). The control statements, viz.

OPCODE	OPERAND 1	OPERAND 2	....	OPERAND N

Fig.2 Format of a parallel intermediate language

WHILE, REPEAT-UNTIL and IF-THEN-ELSE are implemented.

As an example, an arithmetic equation

$$T = \sum_{i=1}^{3} a_i * b_i \qquad (5)$$

can be written in the parallel intermediate language as follows:

$$\begin{array}{lllllll} +* & a_1 & b_1 & a_2 & b_2 & a_3 & b_3 \\ := & T & t\text{-}1 & & & & \end{array}$$

The above two lines are considered as a task. The meaning of the first line is the same as that of the right hand side of Eq.(5). The second line, t-1 is the result of the calculation of a line before this line. Therefore, the meaning of the second line is the assignment of the value of the result calculated in the line before this line into the variable T. With the intermediate language, it is also easy to convert to a triple form which is used in machine code generation process. For ease of utilization in solving a wide variety of linear and nonlinear differential- algebraic systems, an automatic parallel intermediate code generator has also been developed. After the differential-algebraic equations and its Jacobian matrix are put into the system, parallel intermediate codes are generated automatically. Fig.3 shows an example of generated tasks for the Crout algorithm used for solving a system of linear equations in the Newton-Raphson iteration process. Each element a_{ij} of the matrix is considered as a task. As shown in the figure, a total of seventeen tasks are generated for the Crout algorithm including the triangulation process and the forward and backward substitution processes.

4.2 Block Partitioning

A block is defined as a sequence of consecutive statements in which the flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. Each block is connected by edges that represent the flow of control. The information about each block is represented by a record consisting of a count of the number of triples in the block, followed by a pointer to the leader (first triple) of the block, and by a list of predecessors and successors of the block. In each block, the execution order of tasks depends on the data dependencies between the tasks, not on the order of programming. Using this concept, the tasks in each block can be scheduled and executed in parallel. To ensure that all of the tasks in the same block are executed before the execution of the tasks in a successor block or the repetitive execution of the tasks in the same block, a barrier synchronization code is appended at the end of each block. The information obtained from the determination of blocks and barrier synchronization are utilized to reduce overheads in machine code generation process.

```
   1 2 3 4 5
1  x     x        x
2  x x   F        x
3  x x x F        x
4        x        x    x: Non-zero
5      x F x      x    F: Fill-in


       A          B
```

(a) An example of linear equation

<<LU Decomposition>>
1) $u_{14}=a_{14}/l_{11}$
2) $u_{24}=-l_{21}*u_{14}\ /l_{22}$
3) $u_{34}=-(l_{31}*u_{14}+l_{32}*u_{24})\ /l_{33}$
4) $l_{54}=-l_{53}*u_{34}$
<<Forward Substitution>>
5) $y_1=b_1/l_{11}$
6) $b_2=b_2-l_{21}*y_1$
7) $b_3=b_3-l_{31}*y_1$
8) $y_2=b_2/l_{22}$
9) $b_3=b_3-l_{32}*y_2$
10) $y_3=b_3/l_{33}$
11) $b_5=b_5-l_{53}*y_3$
12) $y_4=b_4/l_{44}$
13) $b_5=b_5-l_{54}*y_4$
14) $y_5=b_5/l_{55})$
<<Backward Substitution>>
15) $y_3=y_3-u_{34}*x_4$
16) $y_2=y_2-u_{24}*x_4$
17) $y_1=y_1-u_{14}*x_4$

(b) Task for Crout algorithm

```
 1) /       u14  l11
    :=      u14  t-1
 2) +*      l21  u14
    +minus  t-1
    /       t-1  l22
    :=      u24  t-1
 3) +*      l31  u14   l32  u24
    +minus  t-1
    /       t-1  l33
    :=      u34  t-1
 4) +*      l53  u34
    +minus  t-1
    :=      l54  t-1
 5) /       b1   l11
    :=      y1   t-1
 6) *       l21  y1
    -       b2   t-1
    :=      b2   t-1
 7) *       l31  y1
    -       b3   t-1
    :=      b3   t-1
 8) /       b2   l22
    :=      y2   t-1
 9) *       l32  y2
    -       b3   t-1
    :=      b3   t-1
10) /       b3   l33
    :=      y3   t-1
11) *       l55  y3
    -       b5   t-1
    :=      b5   t-1
12) /       b4   l44
    :=      y4   t-1
13) *       l54  y4
    -       b5   t-1
    :=      b5   t-1
14) /       b5   l55
    :=      y5   t-1
15) *       u34  x4
    -       y3   t-1
    :=      y3   t-1
16) *       u24  x4
    -       y2   t-1
    :=      y2   t-1
17) *       u14  x4
    -       y1   t-1
    :=      y1   t-1
```

Fig.3 An example of generated tasks

4.3 Data Flow Analysis and Task Graph Representation

This process begins with data flow analysis [10-11] of generated tasks, followed by the description of task precedence relations by the so called task graph which essentially is a directed acyclic graph (DAG). The precedence constraints represent the restrictions existing among tasks regarding the execution order of tasks. The precedence relation can be examined by the data flow analysis among tasks. The example of task graph is shown in Fig.4. Each node in a task graph stands for a task and an arc between a pair of nodes for the precedence constraint. Nodes 0 and 20 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.

Once a task graph is generated, the minimum possible processing time achieved by parallel processing of a set of tasks (the marginal time that cannot be shortened even if as many processors as wanted are employed in parallel) can be estimated as the critical path length t_{cr} of the task graph. In Fig.4, the critical path is shown by double-line segments.

4.4 Task Scheduling

In order to process a 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 have to be determined optimally. The determination of 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 [16][17]. This problem, however, has been known as a "strong" NP-hard problem [15]. With this fact in mind, a practical heuristic algorithm named CP/MISF [15] has been proposed. The algorithm can provide very precise approximate solutions for task allocation and its result has been used for generating machine codes.

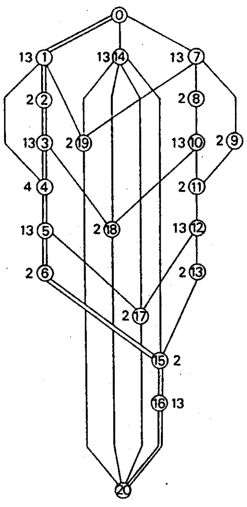


Fig.4 An example of task graph

4.5 Machine Code Generation

For the efficient execution on an actual multiprocessor system, the machine codes are generated using all of the information obtained from the previous processes. The information allows us to minimize synchronization overhead, data transfer overhead and to efficiently use registers to pass the shared data among the tasks assigned to the same processor. For example, the scheduled results give the information about tasks to be executed on each processor element and the execution order of tasks on the same processor element allow efficiently use of the registers for passing data when the tasks allocated to the same processor element. The information from a task graph and a block determination indicate how and when data transfer will occur and where a synchronization code is necessary. It can also give an estimate of waiting time of the task in any processors for the data from other tasks assigned to the other processors. One of the major difficulties common to all multiprocessor structures is the sharing of the required data between processors. To reduce the amount of sharing to a minimum, only the data required by more than one processor is shared.

Although OSCAR has both the local and common memories, the cost to access the data from the common memory is more expensive than that from the local memory. To achieve high speed computation, the machine codes are generated using only local memory and data transfer is performed using dual port memory. Each processor can check the completion of preceding tasks by checking the flags in its dual port memory without having to access to any common bus. This approach helps reduce bus conflict which arise from bus access for checking the completion of preceding tasks. "version number" method and the barrier synchronization method are employed for synchronization among tasks. The version number corresponds to the number of times of iterations or integration steps. Each "writer" task updates the version number to the number of current integration step for itself after it finishes writing shared data. And each "reader" task checks the version number if the number is the same as number of current integration step to the reader task. All processor elements (PE's) have the same version number 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 attached to each shared data when the next integration step is started. Therefore, the version number method can minimize the frequency of access to common buses for task synchronization in this application.

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

5. Performance Evaluation on OSCAR

The performance of the proposed parallel processing scheme was evaluated for four cases of simultaneous differential equations with varying numbers of equations (10, 16, 27 and 52 equations). The complete lists of the equations for case 1 and case 4 are shown in Fig.5.

The tests were run on OSCAR using a UNIX-based workstation for machine code generation. The results of measured parallel processing time plotted against the number of processor elements are shown in Table 1. Fig.6 demonstrates the effectiveness of the present parallel processing scheme for the four cases. In case 4, for example, the processing time with 1 PE (5.74 seconds) is reduced to 2.98 seconds with 3 PEs (reduced to 51.9%) and further to 1.46 seconds with 8 PEs (25.4%). Also notice that the reduction ratio is marked for larger problems.

```
f1 = v1-vs = 0;
f2 = v1-v2-vd = 0;
f3 = v2-vr = 0;
f4 = v2-vc = 0;
f5 = is-id = 0;
f6 = ir+ic-id = 0;
f7 = vs-1 = 0;
f8 = 1.0E-5*(exp(10*vd)-1)-id = 0;
f9 = 10*ir-vr = 0;
f10= 1.0E-04*(dvc/dt)-ic = 0;
```

case 1

```
f1 = is-0.0001 = 0;
f2 = E1-10 = 0;
f3 = E2-10 = 0;
f4 = 3.3E-4*vr1-ir1 = 0;
f5 = 5.0E-5*vr2-ir2 = 0;
f6 = 0.01428*vr3-ir3 = 0;
f7 = 2.0E-4*vr11-ir11 = 0;
f8 = 3.0E-3*vrc1-irc1 = 0;
f9 = 6.6E-5*vr4-ir4 = 0;
f10= 5.0E-4*vr5-ir5 = 0;
f11= 5.0E-5*vr6-ir6 = 0;
f12= 0.01428*vr7-ir7 = 0;
f13= 2.0E-4*vr12-ir12 = 0;
f14= 3.3E-3*vre2-ire2 = 0;
f15= 6.6E-5*vr8-ir8 = 0;
f16= i11-0.98*id1 = 0;
f17= i12-0.98*id2 = 0;
f18= 1.0E-5*(exp(10*vd1)-1)-id1 = 0;
f19= 1.0E-5*(exp(10*vd2)-1)-id2 = 0;
f20= 1.0E-6*(dvc1/dt)-ic1 = 0;
f21= 1.0E-6*(dvc2/dt)-ic2 = 0;
f22= v1-vs = 0;
f23= E1-v2 = 0;
f24= E2-v7 = 0;
f25= vr1-v1 = 0;
f26= vr2-v1+v3 = 0;
f27= vr3-v1+v4 = 0;
f28= vr11-v2+v3 = 0;
f29= vrc1-v5 = 0;
f30= vr4-v3+v6 = 0;
f31= vr5-v6 = 0;
f32= vr6-v6+v8 = 0;
f33= vr7-v6+v9 = 0;
f34= vr12-v7+v8 = 0;
f35= vrc2-v10 = 0;
f36= vr8-v8 = 0;
f37= v11-v3+v4 = 0;
f38= v12-v8+v9 = 0;
f39= vd1-v4+v5 = 0;
f40= vd2-v9+v10 = 0;
f41= vc1-v5 = 0;
f42= vc2-v10 = 0;
f43= ir1+ir2+ir3-is = 0;
f44= iE1+ir11 = 0;
f45= ir4+i11-ir11-ir2 = 0;
f46= id1-i11-ir3 = 0;
f47= irc1-id1+ic = 0;
f48= ir5-ir4+ir6+ir7 = 0;
f49= iE2+ir12 = 0;
f50= i12+ir8-ir12-ir6 = 0;
f51= id2-ir7-i12 = 0;
f52= irc2-id2-ic2 = 0;
```

case 4

Fig.5 Examples of differential-algebraic equations

Table 1. Measured parallel processing time and speed up factor
SP : Speed up factor
Time : Second

No. PE	Case 1		Case 2		Case 3		Case 4	
	Time	SP	Time	SP	Time	SP	Time	SP
1	1.19	1.00	2.49	1.00	3.16	1.00	5.74	1.00
2	0.83	1.43	1.72	1.44	2.25	1.40	3.83	1.49
3	0.70	1.70	1.27	1.96	1.72	1.83	2.98	1.92
4	0.59	2.01	1.04	2.39	1.40	2.25	2.53	2.26
5	0.52	2.28	0.92	2.70	1.20	2.63	2.02	2.84
6	0.47	2.53	0.83	3.00	1.08	2.92	1.81	3.17
7	0.46	2.58	0.77	3.23	0.97	3.25	1.63	3.52
8	0.45	2.64	0.73	3.41	0.90	3.51	1.46	3.93

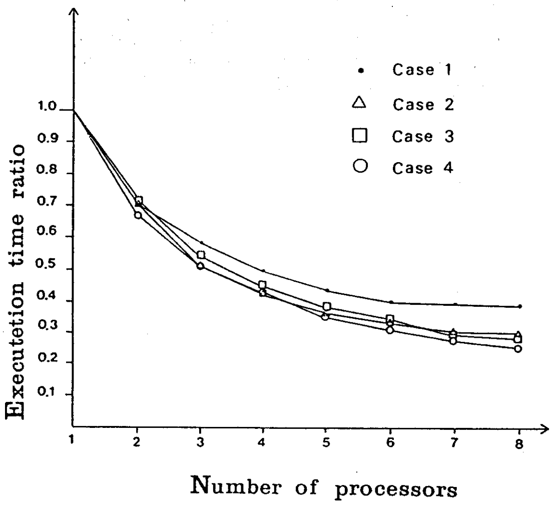


Fig.6 Execution time ratio and number of processors

The method of code generation employed in the proposed scheme is known to be executed much faster than other techniques in sequential computer. So that the speed-up of the scheme as shown in the above result may well be very useful in practical applications.

6. Conclusions

A parallel processing scheme for solving a system of stiff nonlinear differential-algebraic equations on a multiprocessor was proposed. The scheme composes of five steps which are task partitioning and parallel intermediate code generation, block partitioning, data flow analysis and task graph representation, task scheduling, and machine code generation. The proposed scheme was shown to provide efficient parallel processing on an actual multiprocessor system for varying numbers of processor elements. It is expected that the effectiveness and practicality of the proposed scheme will be improved still further by employing faster interprocessor connection media.

References

[1] R.W.Hockney and C.R.Josshope,"Parallel Computer 2: Architecture, Programming and Agorithms", Adam Hilger, (1988).

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

[3] E.Yura, R.Yoshikawa, Y.Nara, T.kimura, H.Aiso, "An approach to parallel processing for continuous dynamic system simulation with microprocessors", Proc. of the 2nd USA-JAPAN Computer Conference, pp. 8/1/1-8/1/6, (1975)

[4] M.A.Franklin, "Parallel Solution of Ordinary Differential Equations.", IEEE.Trans. on Computers., Vol.C-27, No.5, pp.413-420, (May 1978).

[5] R.Yoshikawa, T.Kimura, Y.Nara and H.Aiso, "A Multi-Microprocessor Approach to a High-Speed and Low-Cost Continuous-System Simulation", Proc. National Computer Conf., pp.931-936, AFIP Press, Reston, (1977).

[6] W.L.Miranker and W.M.Liniger, "Parallel Methods for the Numerical Integration of Ordinary Differential Equation", Math. Compt., Vol.21, pp.303-320, (1967).

[7] P.B.Worland, "Parallel Methods for the Numerical Solution of Ordinary Differential Equation", IEEE Trans. on Computers, pp.1045-1048, (October 1976).

[8] S.Horiguchi, Y.Kawasoe, H.Nara, "Parallel algorithms for solving ordinary differential equations and convergence of solutions", Proc. National Conference IECE of Japan, (March 1984).

[9] E.J.H.Kerchoffs, "Multiprocesor algorithms for ODEs", Algorithms and Applications on Vector and Parallel Computers, Elsevier Science Publishers B.V.(North-Holland), (1987).

[10] U.Banerjee, "Dependence Analysis for supercomputing", Kluwer Academic Publisher, (1988).

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

[12] R.K.Brayton, F.G.Gustavson, and G.D.Hachtel, "A New Efficient Algorithm for Solving Differential-Algebraic Systems Using Implicit Backward-Differentiation Formulas", Proc. IEEE, Vol.60, No.1, pp. 98-108, (Jan. 1971).

[13] F.G.Gustavson, W.Liniger, and R.Willoughby, "Symbolic Generation of an Optimal Crout Algorithm for Sparse Systems of Linear Equations", Journal of ACM, Vol.17, No.1, pp.87-109, (January 1970).

[14] H.Kasahara, S.Narita and S.Hashimoto, "Architecture of OSCAR (Optimally Scheduled Advanced Multiprocessor)", Trans. of Japan Inst. Electronics, Infromation and Communication Engineering, Vol.J71D, No.8, (1988).

[15] M.R.Garey and D.S.Johnson, "Computers and Intractability : A Guide to the Theory of NP-Completeness", Freeman, San Francisco, (1979).

[16] E.G.Coffman, "Computer and Job-shop Scheduling Theory", Wiley, New York, (1976).

[17] H.Kasahara and S.Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing", IEEE Trans. Comput., Vol.c-33, pp.1023-1029, (Nov. 1984).