Proposition and Evaluation of Parallelism-Independent Scheduling Algorithms for DAGs of Tasks with Non-Uniform Execution Time

Kirilka Nikolova, Atusi Maeda and Masahiro Sowa
University of Electro-Communications, Graduate School of Information Systems,
Laboratory for Parallel and Distributed Processing
Japan, Tokyo 182-8585, Chofu-shi, Chofugaoka 1-5-1
Tel: (+81) 424-43-5635, Fax: (+81) 424-43-5851
E-mail: {nikol, maeda, sowa}@sowa.is.uec.ac.jp

Abstract: We propose two new algorithms for *parallelism-independent* scheduling. The machine code generated from the compiler using these algorithms in its scheduling phase is *parallelism-independent code*, executable in minimum time regardless of the number of the processors in the parallel computer. Our new algorithms have the following phases: finding the minimum number of processors on which the program can be executed in minimal time, scheduling by an heuristic algorithm for this predefined number of processors, and serialization of the parallel schedule according to the earliest start time of the tasks. At run time tasks are taken from the serialized schedule and assigned to the processor which allows the earliest start time of the task. The order of the tasks decided at compile time is not changed at run time regardless of the number of the available processors which means there is no out-of-order issue and execution. The scheduling is done predominantly at compile time and dynamic scheduling is minimized and diminished to allocation of the tasks to the processors. We evaluate the proposed algorithms by comparing them in terms of schedule length to the CP/MISF algorithm. For performance evaluation we use both randomly generated DAGs (directed acyclic graphs) and DAGs representing real applications. From practical point of view, the algorithms we propose can be successfully used for scheduling programs for in-order superscalar processors and shared memory multiprocessor systems. Superscalar processors *with any number of functional units* can execute the parallelism-independent code *in minimum time* without necessity for dynamic scheduling and out-of-order issue hardware. This means that the use of our algorithms will lead to reducing the complexity of the hardware of the processors and the run-time overhead related to the dynamic scheduling.

Keywords: Parallelism-Independent Scheduling, Static/ Dynamic Scheduling, Multiprocessor Scheduling, Directed Acyclic Graph (DAG), Degree of Parallelism (DOP), Multiple Instruction Issue, In-Order/Out-of-Order Execution.

1. Introduction

In [1] and [2] we proposed a new scheduling method called ***Parallelism-Independent Scheduling Method (PIS),*** enabling the execution of the scheduled program on parallel computers with any degree of parallelism in time comparable to the minimal one. The motivation for the proposition of parallelism-independent algorithms is the following problem of the conventional static scheduling algorithms. They order the instructions/tasks, of the program in such a way, that the program can be executed in minimum time only for one fixed number of processors. As a result, the generated by the compiler machine code is with a fixed degree of parallelism. The program can be executed efficiently only when the degree of parallelism of the machine code matches the degree of parallelism of the parallel computer. If the processor number in the parallel computer is changed, the program has to be rescheduled with another assumption about the degree of the parallelism, and a new machine code has to be assigned for execution, in order to achieve minimal execution time on the new number of processors. However, all the commercial applications are distributed as a machine code and the number of the processors of the executing parallel computer is known only after the distribution of the application. This means that the machine code has to be scheduled after the distribution when the number of the processors of the parallel computer is known. But it is very difficult to schedule the machine code, because some of the data dependencies in it may be missing. Therefore we need a new scheduling method which is applied on the source code and which eliminates the need for generation of new code for each degree of parallelism of the parallel computer the program is run on. We have introduced this kind of method in [1] and [2]. Our method implies fulfilling greater part of the scheduling job at compile time and only allocating the tasks to the processors at dynamic time. We would like to underline the advantage of our algorithm: the order of the tasks determined at compile time is not changed at dynamic time with the change of the number of the processors.

In contrast to the traditional static scheduling methods, the code generated by the compiler using our scheduling method is parallelism-independent code executable on any number of processors in near minimal time. In contrast to the dynamic scheduling method that can be applied for any number of processors, our method does not need out-of-order issue and execution of the instructions. The dynamic scheduling implies additional hardware necessary to look far ahead in the dynamic instruction stream, find independent instructions and schedule them out of order. However the detection and scheduling of the instructions dynamically increases the complexity of the hardware, lengthens the cycle time of the machine and reduces the actual speedup over the scalar processor. The advantage of our algorithms is that there is no need for such complex hardware and dynamic change of the order of the instructions.

In our previous research we have proposed parallelism-independent scheduling algorithms for DAGs of tasks with uniform execution time. In the present new study, we relax the restrictions, imposed on the DAGs in

our past research, and extend our method to make it applicable for DAGs of instructions/ tasks with non-uniform execution times. This makes our method more practical and applicable both for superscalar processors and multiprocessors with shared memory.

In this paper we propose two new algorithms for Parallelism-Independent Scheduling: ***Critical Path Parallelism-Independent Scheduling Algorithm*** (CPPIS) and ***Latest Finish Time Parallelism-Independent Scheduling Algorithm (LFTPIS)***. We compare them between each other and to the CP/MISF (Critical Path Most Immediate Successors First) Algorithm [3], which is applied for each degree of parallelism. CP/MISF has been chosen for the comparison because it is proven that it gives optimal results in 87% of the random cases. The comparison is done in terms of execution time of the scheduled program and the computational time of the scheduling algorithm (the time they spent for scheduling). The performance evaluation is carried out using random DAGs and DAGs representing real applications like the computation of the Fast Fourier Transformation and the solution of a system of linear equations.

The remainder of our paper is organized as follows. In Section 2 we give information about some existing scheduling methods and algorithms, which are directly related to our research. There we discuss the problems they imply and how we suggest to solve these problems. We describe in detail the new algorithms we propose in Section 3. In Section 4 we present the simulations carried out for performance evaluation of the algorithms, and comment the results obtained from these simulations. We draw the conclusions in the last section.

2. Traditional Scheduling vs. Parallelism-Independent Scheduling

The utilization of the big potential of the multiprocessor systems depends considerably on the efficient instruction scheduling and for that reason there are many researches in this area. The parallel program can be represented by a DAG in which the nodes denote the tasks of the program and the edges show the precedence relationships between the tasks. Below we describe briefly the different approaches for solving the scheduling problem the purpose of which is ordering the tasks of the program for achieving *minimal execution time, minimum used resources or efficient load balancing.*

2.1 Static Scheduling

The most common technique for DAG static scheduling is the *list scheduling*: assigning priorities to the nodes of the DAG and allocating the nodes for execution to the available processors in order of their *priorities*. The quality of the algorithms depends on the accuracy with which the priorities of the nodes are defined. Adam, Chandy and Dickson suggest that using the *level of the task* (the longest path from the node to the exit node) as a priority yields the nearest solution to optimal. Many tasks scheduling schemes use the CP/MISF algorithm [3] because the Critical Path (CP – the longest path in the graph) determines the shortest possible execution time of the program. Its steps are defining the level of the tasks, sorting the tasks in decreasing order of their levels and number of their immediate successors and assigning the ready tasks (tasks whose predecessors are already executed) to the idle processors. But all the traditional static algorithms generate code with a fixed degree of parallelism because they use information about the number of the processors when they order the tasks of the program (i.e. there is one task order allowing the minimum execution time of the program on 2 processors and another task order allowing the minimum execution of the program on 3 processors). As a result, the generated code will be executed in minimum time only on a parallel computer with number of processors equal to the degree of parallelism of the generated code.

2.2 Dynamic Scheduling

This approach can be used for scheduling not depending on the degree of parallelism of the parallel computer, however it consumes time and resources which lead to overhead during the program execution. The basic idea of the dynamic scheduling is to perform task ordering and allocation at run-time. For example, many multiple-issue processors employ out-of-order execution hardware in the processor pipeline. Such scheduling hardware can result in good performance without relying on compile time scheduling. However the shortcomings of the out-of-order execution is that there is a necessity for hardware for our-of-order issue and it can lead to the increase of the processor cycle time.

2.3. Parallelism-Independent Scheduling

The use of our parallelism-independent scheduling algorithms makes possible by only one order of the tasks at compile time to achieve good execution times for any number of processors without the need for dynamic scheduling hardware and great time overhead for run-time scheduling.

In [1] we propose 3 Parallelism-Independent Scheduling Algorithms developed under the assumption that the tasks of the parallel program are with equal execution times. The scheduling framework for these algorithms is as follows:

1. Assigning priorities to the nodes of the DAG using some heuristics priority as levels or co-levels of the nodes.
2. Forming of blocks of simultaneously executable tasks taking the data dependencies between the tasks and their priorities.
3. Rearrangement of the tasks within the borders of the block so that they can be executed with different degrees of parallelism
4. Serialization of the schedule by connecting all the blocks
5. Adding of markers for the parallel execution limits.

The non-uniformity of the execution time of the tasks makes the solution of the parallelism-independent problem much more difficult because we need to devise a function for ordering of the tasks different from their levels or co-levels. We describe how we solve this problem in the next section.

3. The Proposed Algorithms

The problem we face is how to order the program's tasks with different execution times so that we can guarantee the execution of the program in this fixed order on any number of processors in time comparable to the minimal one. It is obvious that we cannot simply use the levels of the tasks for parallelism-independent ordering (ordering the tasks by their levels yielding minimum time can be done only for a fixed number of processors). The co-levels of the tasks (the longest path from a node to input node) can be used for such ordering because they denote the earliest possible time (EST) for execution of the tasks. If we order the tasks in increasing order of their EST, then there is greater possibility for a greater sequence of independent tasks in the serialized schedule and efficient execution on any number of processors. However, if we use only EST as a priority, then the execution time of the scheduled program exceeds the results of the CP/MISF algorithm up to 20% proved by experimental results. That is why we try to order the tasks by a combined function, their levels and EST (CPPIS), or their latest finish time and EST (LFTPIS).

The common idea unifying our algorithms is that first a parallel schedule is formed for a fixed number of processors p using the levels or the latest finish time of the tasks as priority. This fixed number of processors p is the minimum number of processors allowing the execution of the program with a time equal to the CP in the DAG. Then this parallel schedule is serialized ordering the tasks according to their EST in the parallel schedule for p processors. In this way we limit the degree of parallelism to p and try to order the tasks so that the program can be executed in time near to the minimal one for number of processors less than p ($2 \leq$ Processor Number $<$ p). The common steps of our algorithms are:

At static-time:

1. Finding the number of processors p
2. Forming a parallel schedule by ordering the tasks by a priority different for the two proposed algorithms (the levels of the tasks or the latest finish time of the tasks) for p processors.
3. Serialization of the schedule according to the tasks earliest start time, as they appear in the parallel schedule for p processors. In this way the tasks are ordered according to two priorities, either their level or LFT and their EST for p processors.

At run-time:

Tasks are taken from the serialized schedule and assigned for execution to the available processor on which they can start execution at the earliest possible time. If a task from that serial order cannot start execution because it is not ready, then it waits until it becomes mature. In the meantime no other task can start execution before it.

3.1 Critical Path Parallelism-Independent Scheduling Algorithm

The steps of CPPIS Algorithm are as follows:

1. Finding p, the minimal number of processors needed for execution of the program with time equal to the length of the critical path CP of the DAG.

Now we will explain how we find this processor number p. At the beginning we estimate P as follows:

$$P = \left(\sum_{i=1..N} T_i\right)/CP \text{ , or}$$

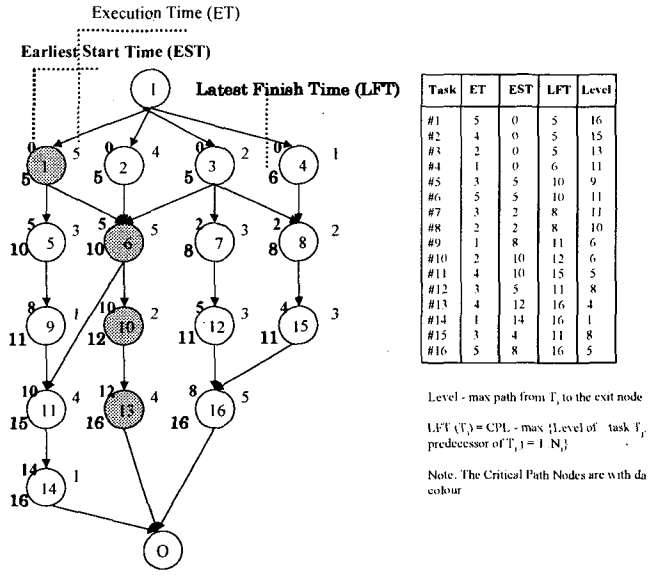
P= Maximal Execution Time of /the DAGCP,

where T_i is the execution time of task i, and the Maximal Execution Time of the DAG is its execution time on 1 processor sequentially. Then we schedule the DAG using the CP/MISF algorithm for number of processors equal to P. If the execution time of the program is greater than P, then we increase the number of the used processors to P+1, until the execution time of the program becomes equal to CP. The procedure is given below:

Repeat
  Schedule for P processors by CP/MISF
  If the Schedule Length > CPLength
    P=P+1
Until Schedule Length = CPLength

2. Scheduling the graph by CP/MISF Algorithm for DOP p
3. Forming a serialized schedule according to the EST of the tasks as they appear in the schedule for DOP p

Steps 2 and 3 of CPPIS Algorithm for the example DAG of Fig. 1 are presented in Fig. 2. The scheduling results for different DOPs are shown in Fig. 3



Task	ET	EST	LFT	Level
#1	5	0	5	16
#2	4	0	5	15
#3	2	0	5	13
#4	1	0	6	11
#5	3	5	10	9
#6	5	5	10	11
#7	3	2	8	11
#8	2	2	8	10
#9	1	8	11	6
#10	2	10	12	6
#11	4	10	15	5
#12	3	5	11	8
#13	4	12	16	4
#14	1	14	16	1
#15	3	4	11	8
#16	5	8	16	5

Fig. 1 An Example DAG with calculated levels, LFT and EST

3.2 Latest Finish Time Parallelism-Independent Scheduling Algorithm

The steps of the LFTPIS algorithm are:

1. Finding p, the minimum degree of parallelism for which the program is executed with time equal to the length of the CP.
2. Scheduling for DOP p, using for priorities the LFT of the tasks. For that purpose the tasks are ordered in increasing order of their LFT and assigned to the p processors in this

order, taking in account the precedence constraints between the tasks. Tasks with smallest LFT have greater priority. The LFT shows at what time a task shall finish execution so that the total execution time of the program does not exceed the CP in the DAG.

3. Serialization of the schedule according to the EST of the tasks in the schedule for DOP p.

Steps 2 and 3 of LFTPIS Algorithm for the example DAG of Fig. 1 are presented in Fig. 4. The scheduling results for different DOPs are shown in Fig. 5. By comparison of the results of the CPPIS and LFTPIS Algorithms for the example graph we use, we can see that for DOP=2 LFTPIS gives better results than CPPIS algorithm.

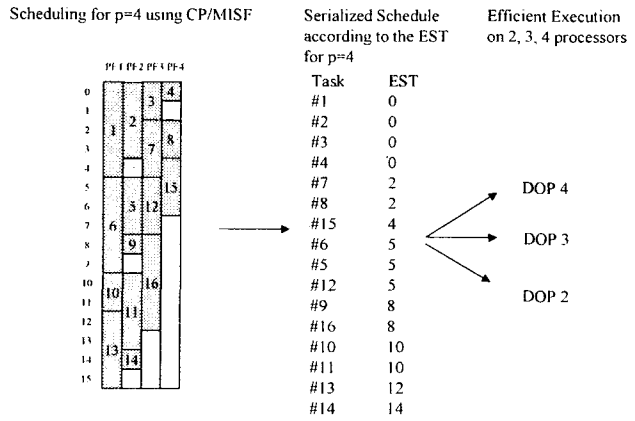


Fig. 2 The steps of CPPIS Algorithm

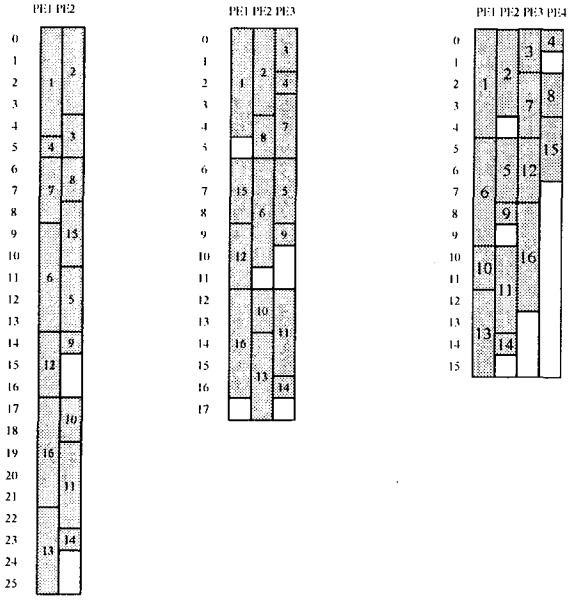


Fig. 3 Scheduling Results for CPPIS on 2, 3 and 4 processors

4. Performance Evaluation

4.1 With Random DAGs

We use random DAGs with different characteristics as number of edges/nodes and different degrees of parallelism in the graphs:

- 20 graphs 250 nodes/1000 edges, p=8
- 20 graphs 500 nodes/2000 edges, p=12
- 20 graphs 1000 nodes/4000 edges, p=16

We have taken the average value for the Execution Time Ratio (ETR):

ETR = Exec. Time of CPPIS (LFTPIS)/Exec. Time of CP/MISF

ETR represents the maximum deviation of the results of CPPIS and LFTPIS Algorithms from those of the CP/MISF Algorithm, when applied separately for each DOP. The results for the maximum average ETR are presented in Table 1. The average ETR for the 20 graphs with 500 nodes/2000 edges and p=12 is presented in Fig. 6. LFTPIS and CPPIS perform very similarly, with slightly better performance of LFTPIS for number of processors less than p and better performance of CPPIS performs for number of processors greater than p.

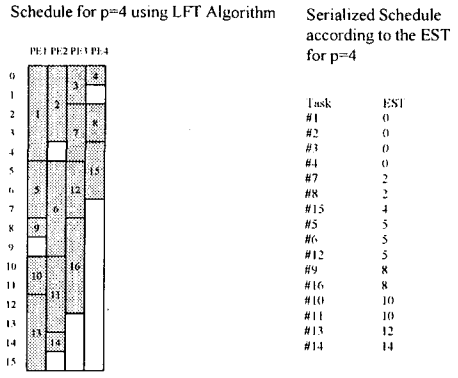


Fig. 4 The steps of LFTPIS Algorithm

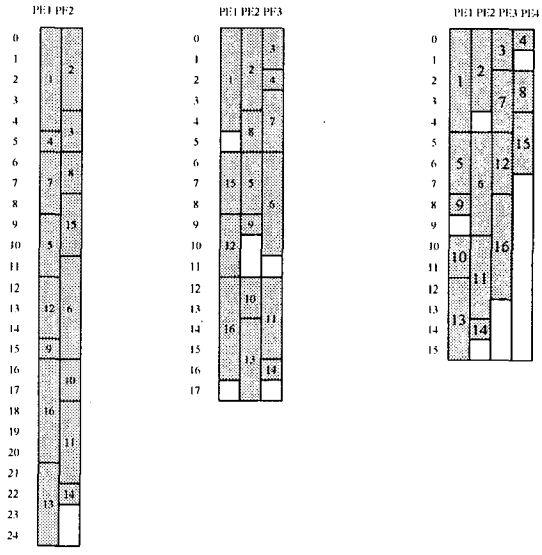


Fig. 5 Scheduling Results for LFTPIS on 2, 3 and 4 processors

If we compare the computation times of CPPIS and LFTPIS Algorithms, we can see that the computational time of LFTPIS Algorithm exceeds that of CPMISF. (Fig. 7). In this figure the time of CP/MISF algorithm is taken for all possible degrees of parallelism, that is the reason it is longer than that of LFTPIS and CPPIS.

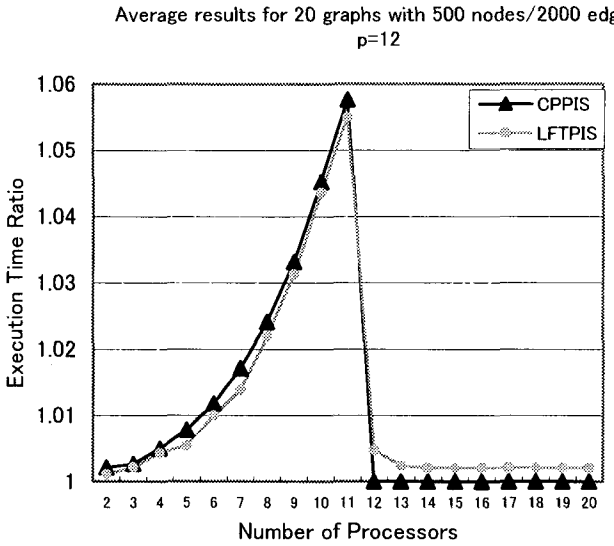


Fig. 6 Average ETR for 20 graphs with 500 nodes/2000 edges, p=12

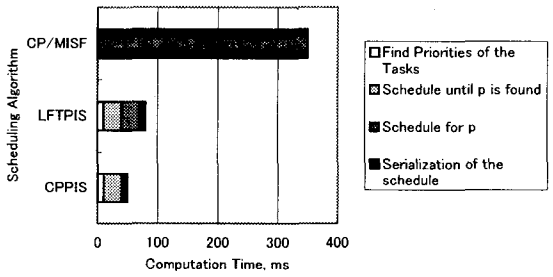


Fig. 7 Comparison of the Execution Time of the Parallelism-Independent Algorithms vs. CP/MISF Algorithm

Table 1

Graph Type	CPPIS Max Average ETR	LFTPIS Max Average ETR
250n./1000 e.	6.2%	6.3%
500n./2000 e.	5.76%	5.49%
1000n./4000 e.	5.15%	4.93%

4.2 With DAGs of Real Applications

We used the DAGs of the programs for computation of the Fast Fourier Transformation (FFT) and linear systems of equations. The results for the FFT are presented in Fig. 8. The best results are obtained using the CPPIS Algorithm. The results of the CPPIS Algorithm do not exceed those of the CP/MISF Algorithm more than 8.5% at DOP=14. As a whole, the deviation from the results of the CP/MISF Algorithm is less than 5% in 85% of the DOPs. The LFTPIS Algorithm gives maximum deviation from the results of the CP/MISF Algorithm also 8.5% for DOP=14, however its results are worse for the other DOPs. It gives results less than 5% exceeding the results of the CP/MISF Algorithm in 70% of the DOPs. We obtain similar results

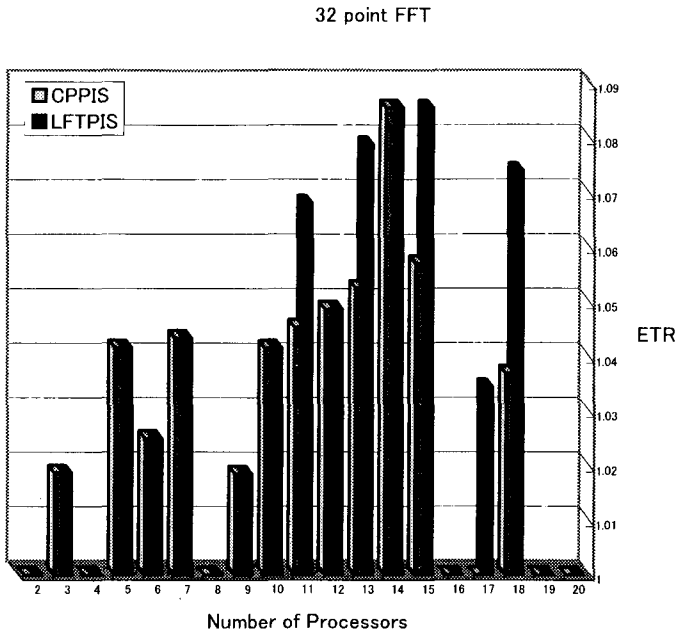


Fig. 8 ETR for the CPPIS and LFTPIS Algorithm

in the case of using the DAG of the program for solution of system linear equations. Then the maximum deviation of LFTPIS and CPPIS from the results of CP/MISF is 8.88%.

5. Conclusions

We propose and evaluate two parallelism-independent algorithms producing code executable in time comparable to the minimal one regardless of the number of the processors in the parallel computer. According to the simulation we have carried out, the maximum deviation of the results of CPPIS and LFTPIS Algorithms from the results of the CP/MISF Algorithm is between 5% and 7% on the average, and not exceeding 10% in the separate random cases we have studied. Based on the results, we can conclude that these algorithms (especially CPPIS) can be used successfully for parallelism-independent scheduling for real applications. We have tested them not only with random graphs but with graphs derivatives of real programs as the computation of the Fast Fourier Transformation and Solution of Systems of Linear Equations and the experiments proved the efficiency of these algorithms.

References

[1] Kirilka Nikolova, Atusi Maeda and Masahiro Sowa, Parallelism Independent Scheduling Method", IEICE Trans. on Fundamentals, June 2000.

[2] Kirilka Nikolova, Atusi Maeda, and Masahiro Sowa, "Parallel-Free Scheduling Method", Proceedings of the International Technical Conference on Circuits and Systems, Computers and Communications, Vol. 2, pp. 1132-1135, IEICE 1999.

[3] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms, for Efficient Parallel Processing", IEEE Transactions on Computers, C-33 (Nov. 1984), pp. 1023-1029.