'91 KACC 1991. 10. 22~24

SCHEDULER FOR PARALLEL PROCESSING WITH FINELY GRAINED TASKS

° Takafumi Hosoi†, Hitoshi Kondoh† and Shinji Hara†

†Department of Control Engineering, Tokyo Institute of Technology,
2-12-1 Oh-Okayama, Meguro-ku, Tokyo 152 Japan

ABSTRACT

A method of reducing overhead caused by the processor synchronization process and common memory accesses in finely grained tasks is described. We propose a scheduler which considers the preparation time during searching to minimize the redundant accesses to shared memory. Since the suggested hardware (synchronizer) determines the access order of processors and bus arbitration simultaneously by including the synchronization process into the bus arbitration process, the synchronization time vanishes. Therefore this synchronizer has no overhead caused by the processor synchronization[1].

The proposed scheduler algorithm is processed in parallel. The processes share the upper bound derived by each searching and the lower bound function is built considering the preparation time in order to eliminate as many searches as possible. An application of the proposed method to a multi-DSP system to calculate inverse dynamics for robot arms, showed that the sampling time can be twice shorter than that of the conventional one.

1. INTRODUCTION

Multiprocessor systems have been employed in a wide variety of computer applications not only in the field of information processing but also for the control of robots and in real-time high-speed simulations of dynamic systems, because they can provide excellent response and cost effectiveness. In order to take advantage of parallel processing, it is desirable to attain a minimum execution time with a minimum number of processors. To achieve this, an efficient scheduling algorithm must be developed to allocate a set of tasks to several member processors and to determine the order or sequence of execution of the tasks allocated to each processor. This type of problem, usually referred to as the minimum execution time (schedule length) multiprocessor scheduling problem, has long been studied extensively by a number of pioneering researchers.

This problem, however, has been solved only for the cases with large grained tasks due to the following two reasons: A1) the larger overhead caused by the processor synchronization and the shared memory accesses, when the tasks are finely grained than when the tasks are large ones. and A2) Scheduler algorithms are generally NP complete[2], i.e., it is impossible to construct a pseudopolynomial-time optimization algorithm. These restrictions impose that this problem be usually solved with large grained tasks and a small number of them.

In packing a box with goods we can pack more if the goods are of small size than if they are large ones. Similarly in parallel processing, it is more efficient to handle a job by decomposing it into finely grained tasks than trying to manipulate large ones[3].

In this paper, we propose a scheduler which minimizes the redundant accesses to shared memory with finely grained tasks (a cause of overhead that can be determined before scheduling). Moreover the scheduler itself is implemented using parallel processing techniques. And the processes share the common upper bound and the lower bound function which includes the preparation time for shared memory accesses.

We confirmed the effectiveness of the proposed scheme by applying it to the computation of Newton-Euler equations for dynamic arm control, using a multi DSP system with four TMS320C25 processors. A multi DSP system hardware configuration (synchronizer) has been designed to eliminate the synchronization time (a cause of overhead that cannot be determined before scheduling)[1]. A key factor in the hardware implementation of the control-flow parallelism is high-speed circuitry in order to establish synchronization between multiple processors.

The prototype was implemented using four TMS320C-25's. In this system, the access time of shared memories equals to the access time of local memories.

2. SCHEDULER FOR MINIMUM MEMORY ACCESS

2.1 CONSIDERATION OF PREPARATION COST

Using the system described above, synchronization time vanishes and the access time for shared memories equals to the access time for local memories. In finely grained task scheduling, however, there must be an additional improvement: Another component of inter-processor communication, which is the data transfer time or preparation time, can be also ignored in multi-processor systems with tightly-coupled shared memories, since data can remain on local memory or registers as shown by the following example program section:

```
c:=a-b;   {task i}
d:=b×c;   {task j}
```

1) In cases where data(Var_C) to be referenced in task j and data to be assigned in the preceding task i on the same processor were the same. Looking at the CPU2 instructions, the data load of task j can be omitted if both tasks are assigned to the same processor(Figure 2.1), and 2) Under condition 1) and if the following task j is the only one to refer to data to be assigned in of task i, both the data store of task i and the data load in task j can be omitted(Figure 2.2).

```
                CPU 1                CPU 2
        Load    Acc Var_A
task i  Sub     Acc Var_B
        Store   Acc Var_C
task j                       Load    Acc Var_C
                             Mul     Acc Var_B
                             Store   Acc Var_D
```

Figure 2.1 The data load of task j can be omitted.

```
                Load    Acc Var_A
        task i  Sub     Acc Var_B
        task j  Mul     Acc Var_B
                Store   Acc Var_D
```

Figure 2.2 Both the data store of task i and data load of task j, can be omitted.

where Sub and Mul are commands of subtraction and multiplication, respectively. Acc is an accumulator.

In this multi-processor scheduling, we consider the preparation time between two tasks as a function of their position orders. Since the values of this function can be determinate before scheduling starts, the values are stored in a cost matrix. Hence, the proposed scheduling scheme will reduce the number of shared memory access to improve overhead and to avoid bus bottle-necks.

2.2 PROBLEM WITH COMPUTATION AMOUNT

Scheduler algorithms are generally NP complete as mentioned in Section 1. In addition, it is more difficult to obtain the optimal scheduling if we consider the preparation time, since the calculation time will inevitably increase.

Here, we extend the scheduling in [4] and solve the problem. Let n, m, n_{ready} and m_{av} be the number of all tasks, all processors, ready tasks and available processors, respectively at a certain stage of branching. Then the number of idle tasks n_{idle} to be considered for allocation is given by

$$n_{idle} = m_{av} - 1 \quad \text{for } m_{av} = m$$
$$n_{idle} = m_{av} \quad \text{for } 1 \leq m_{av} < m$$

Hence, in the conventional case, the number of nodes generated from each branching node is given by

$$n_{branch} = {}_{(n_{ready}+n_{idle})}\mathbf{C}_{m_{av}} \tag{1}$$

where $\mathbf{C}$ means the number of combinations(See Figure 2.3).

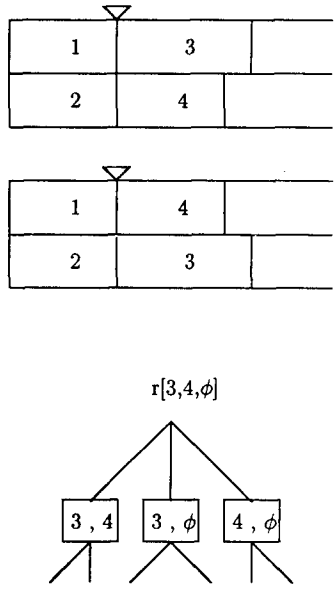


Figure 2.3 The number of nodes generated from each branching node in the conventional case.

In the proposed case, however, since the preparation time is determined by two tasks stretched in a row on the same processor, the number of nodes generated from each branching node becomes

$$n'_{branch} = {}_{(n_{ready}+n_{idle})}\mathbf{P}_{m_{av}} \tag{2}$$

where $\mathbf{P}$ means the number of permutations(See Figure 2.4).

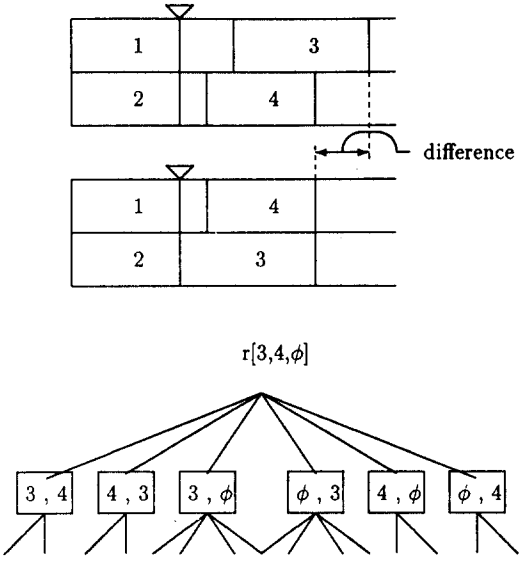


Figure 2.4 The number of nodes generated from each branching node in the proposed case.

Hence, comparing these equations, we have

$$({}_{n_{ready}+n_{idle}}\mathbf{P}_{m_{av}}) / ({}_{n_{ready}+n_{idle}}\mathbf{C}_{m_{av}}) = m_{av}! \qquad (3)$$

This means that complexity of the scheduling algorithm increases seriously by considering the preparation time.

2.3 PARALLEL SCHEDULER AND LOWER BOUND FUNCTION

Here, in order to unbias searching in the search tree, the scheduler itself is processed in parallel by several processors based on heuristic lists containing the order of tasks to be assigned to the processors. These heuristic lists are built according to the following heuristic rules to obtain a better initial solution:

1) Higher precedence to the task with the latest starting time and the smallest floating time.

2) Higher precedence to the task with the latest starting time and the largest total processing time among the following tasks.

3) Apply rule 1) after slight shuffling of the list.

4) Apply rule 2) after slight shuffling of the list.

Each scheduler part is processed independently. Since the branch-and-bound method (B&B) has a property that the frequency of finding a solution decreases as the search progresses, all heuristic lists are shuffled after searching for a specified period of time. In addition, since a B&B can be accelerated by eliminating redundant searching nodes, we use a special scheme to share the upper bound and the lower bound function which estimates the lowest solution from currently active nodes.

The proposed lower bound function is:

$$t_{div}(\pi_a) = \lceil (\sum_{i \in I(\pi_a)} t_i + t_{cost})/m \rceil + t_0 \qquad (4)$$

where,

$$t_{cost} = \max(t_{row}, t_{col}) \qquad (5)$$

$$t_{row} = \sum_{k=1}^{n_{av}-(m-m_{av})} (\min_{i \in N_{av}}(k) \min_{i \in N_{av}, j \in N_{nf}} C_{i,j}) + \sum_{k=1}^{m-m_{av}} \min_{i \in N_{av}}(k) C_{i,n} \qquad (6)$$

$$t_{col} = \sum_{k=1}^{n_{av}-(m-m_{av})} (\min_{j \in N_{nf}}(k) \min_{i \in N_{av}, j \in N_{nf}} C_{i,j}) + \sum_{k=1}^{m-m_{av}} \min_{i \in N_{av}}(k) C_{i,n} \qquad (7)$$

$N_{av}(= I(\pi_a)), n_{av}$: a set of unassigned tasks and its number

N_{nf}, n_{nf}: a set of unfinished tasks and its number

In a set X, let $\min(k)X$ be the k-th smallest element, i.e., $\min(1)X = \min X$. $C_{i,j}$ is the element of a cost matrix which is determined by the former task i(row side) and the latter task j(column side). This lower bound function is defined by the following rules:

If we consider a certain node of the search tree, we can know the smallest number of preparation processes considering the remaining tasks. At first, paying attention to the preparation times which follow the tasks, we get the smallest preparation times from each row of the cost matrix. Secondly, we pick up the preparation times that can certainly occur in increasing order length and the total of these is the smallest preparation time possible. Following we apply the same procedure for the columns excepting the End column. Then, we define t_{cost} as the bigger of t_{row} and t_{col}.

Consider the simple example of Figure 2.5. Task 1 has been already assigned and the number of active processor is one ($m - m_{av} = 1$). The lower bound function is calculated as follows:

1. Pick up the smallest element of each row. In this case, from the cost matrix(Figure 2.6),

   1 2 2 3

2. Since there are four unfinished tasks(task 1,2,3,4) and the number of active processor is one, we get the three smallest numbers among the above in increasing order.

3. Pick up the smallest element from the End column, because the active processor will certainly have an End preparation time. Then t_{row} is the total:

$$t_{row} = 1 + 2 + 2 + 2 = 7$$

4. Similarly the evaluation for column side yields to

$$t_{col} = 1 + 3 + 3 + 2 = 9$$

5. Since the preparation time lower value is the maximum of the two, we obtain $t_{cost} = 9$. Then we calculate the lower bound function considering preparation time.

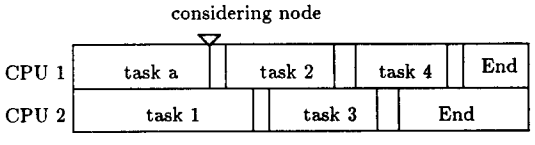


Figure 2.5 The example node.

	task 2	task 3	task 4	End
task 1	3	4	1	4
task 2	—	5	4	2
task 3	3	—	2	5
task 4	6	3	—	3

Figure 2.6 The cost matrix for the example.

Since the scheduler finds the best combination of tasks, it automatically minimizes the redundant access to shared memory.

3. THE SCHEDULER APPLICATION

3.1 RESULT OF SIMULATION

We applied the described scheduler to about a hundred task graphs randomly made. Task graphs to be processed in parallel are artificially generated randomly and divided into several groups to demonstrate the usefulness of parallel processing of finely grained tasks. In this case, we let all preparation times to be 0, since we want to know the effect of scheduling in only finely grained tasks. The scheduled result is shown in Figure 3.1, where we note that the increasing of the number of processors results in a better solution, since the increased number of divisions shortens the critical path. This is the reason why we suggest the scheduling with finely grained tasks in spite of increasing computation amount.

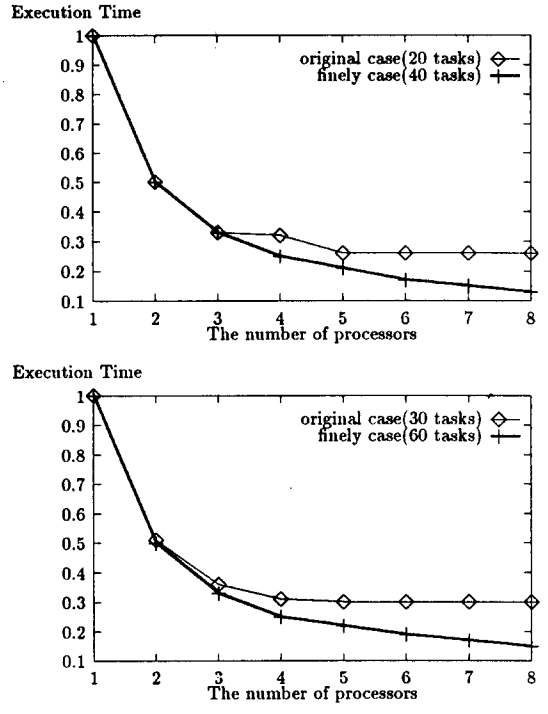
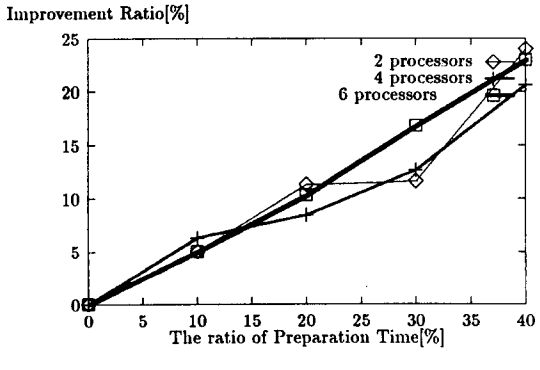


Figure 3.1 The effect of scheduling in finely grained task. (The size of each task in the finely grained case is half of that in the original one.)

The size effect of the preparation time is shown in Figure 3.2. The improvement ratio $IM[\%]$ is defined as

$$IM[\%] = \frac{PN - PC}{PC} \times 100 \qquad (8)$$

where PC is the best solution when considering the preparation time, and PN is the best solution when ignoring it, i.e. letting all the preparation times to be the biggest one that is expected. The ratio of preparation time is the percentage of the sum of all preparation times comparing with the sum of all execution times.



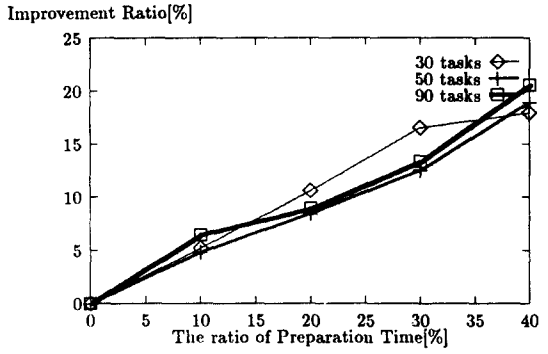


Figure 3.2 The size effect of the preparation time.

We conclude from the above experiment that the improvement ratio is proportional to the preparation time and is not influenced by the number of processors and tasks. Then if the preparation time gets larger, we can not ignore it.

3.2 APPLICATION TO A ROBOT MANIPULATOR

In order to show the effect of finely grained tasks and the consideration of preparation time, we applied the proposed method to the inverse-dynamics calculations (Newton-Euler method) of a three-axes robot arm[5][7] since the control of more and more complex robot systems in real-time high-speed applications is recently being required.

In our experiments, the jobs were divided in two ways: L) large grained tasks (Table 3.1;16 tasks) and F) finely grained tasks (Table 3.2;184 tasks).

Table 3.1 Tasks for large grained scheduling.

task No.	operation	execution time
0	dummy task	0
1	2M	24
2	6M,2A	88
3	2M	24
4	7M,5A	124
5	4M,3A	72
6	8M,7A	152
7	16M,11A	280
8	7M,6A	132
9	3M	36
10	13M,8A	220
11	7M,5A	124
12	21M,16A	380
13	5M,3A	84
14	9M,6A	156
15	dummy task	0

Table 3.2 Tasks for finely grained scheduling.

task No.	operation	execution time
0	dummy task	0
1,..,182	1M or 1A	12 or 8
183	dummy task	0

110M,72A

where A means addition and M multiplication. The scheduling case F) is carried out based on the following three methods;

F1) scheduling considering the preparation time (proposed method).

F2) scheduling ignoring the preparation time and inserting the required preparation time after scheduling.

F3) scheduling including the preparation time and eliminating the redundancy preparation time after scheduling.

The results for our four processors are shown in Table 3.3.

Table 3.3 The Results for four processors.

	scheduling[clock]	execution[clock]
Large grain(16 tasks)	1844	1844
Fine grain(184 tasks)		
F1)	844	844
F2)	476	992
F3)	1040	1004

Although the scheduling result in case F2) (476 clocks) is shorter than the one of case F1) (844), their execution time results (992 and 844 clocks respectively) show that , in fact, method F1) has a better performance. Moreover, the execution result in case F1) is more than twice faster than the one of case L). This means that the scheduling with finely grained tasks is clearly better than with large grained ones, if we can search almost all part of the search tree. In addition, we have changed the number of processors. Figure 3.3 shows the execution time vs. number of processors. From it, we conclude:

C1) When the preparation time is taken into consideration, the execution time in F1) is 9.8-28.1% shorter than that of F2) or F3).

C2) For a small number of processors F1) is far better than F2) and F3).

C3) For cases with more than 4 processors the difference between F1) and F2) and F3) is small but the difference between F1) and L) tends to become significantly larger since L) reaches the critical path with 2 processors.

C4) In general, the finely grained tasks option is better than the large grained one since the later reaches the critical path earlier.

C5) Case F1) is always the better method for any number of processors.

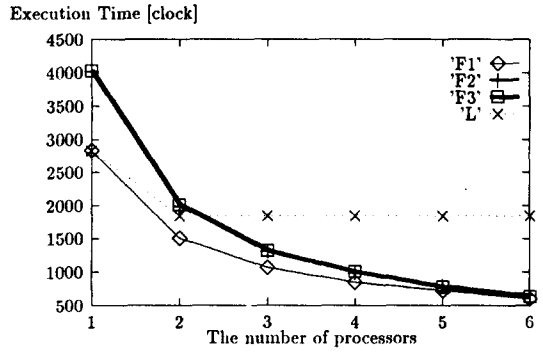


Figure 3.3 The execution time vs. number of processors.

4. CONCLUSION

We proposed a scheduler considering the preparation time and minimizing redundant access to shared memory. The scheduler was implemented to be processed in parallel and shares the common upper bound of each process and the lower bound function considering the preparation time for shared memory access. Applying the proposed system to a multi-DSP configuration to perform inverse-dynamics calculations in a robot arm application showed that the sampling time can be shortened twice as that of the conventional one. Moreover we showed the necessity of considering the preparation time when the scheduling is made with finely grained tasks and confirmed that the proposed method made the scheduling more accurately.

The results also showed that, contrary to the common belief, finely grained tasks applications can not be made to cause communication overhead providing a performance even better than that of large tasks applications.

REFERENCES

[1] H.Kondoh, F.Kobayashi and N.Takehira, "A Bus Arbiter with Inter-Processor Synchronization for Parallel Processing of Operations.", T.IEE Japan, Vol.109-C, pp57-61, No.2, Feb. 1989.

[2] J.K.Lenstra and A.H.G.R.Kan, "Complexity of Scheduling Under Precedence Constrains", Oper. Res., vol.26, pp22-35, Jan. 1978.

[3] B.Kruatrachue and T Lewis,"Grain Size Determination for Parallel Processing", IEEE Software, No.1, pp23-32, Jan. 1988.

[4] H.Kasahara and S.Narita, "A Practical Optimal / Approximate Algorithm for Multi-Processor Schduling Problem", Trans. IEICE, vol.J67-D, No.7, July 1984.

[5] H.Kasahara and S.Narita, "Parallel Processing of Robot-Arm Control Computation on a Multimicroprocessor System", IEEE Journal of Robotics and Automation, vol.RA-1, No.2, June 1985.

[6] R.G.Babb,"Parallel Processing with Large-Grain Data Flow Techniques", Computer, pp.55-61, July 1984.

[7] H.Kondoh, et al., "Parallel Processing of Finely Grained Tasks: Arbitrating Synchronizer and Parallel Scheduler", IECON, 1991 (To be presented)