Task scheduling algorithm for multicore processor system for minimizing recovery time in case of single node fault

Shohei Gotoda, Minoru Ito
Nara Institute of Science and Technology
Nara, Japan
{shohei-g, ito}@is.naist.jp

Naoki Shibata
Shiga University
Shiga, Japan
shibata@biwako.shiga-u.ac.jp

***Abstract*—In this paper, we propose a task scheduling algorithm for a multicore processor system which reduces the recovery time in case of a single fail-stop failure of a multicore processor. Many of the recently developed processors have multiple cores on a single die, so that one failure of a computing node results in failure of many processors. In the case of a failure of a multicore processor, all tasks which have been executed on the failed multicore processor have to be recovered at once. The proposed algorithm is based on an existing checkpointing technique, and we assume that the state is saved when nodes send results to the next node. If a series of computations that depends on former results is executed on a single die, we need to execute all parts of the series of computations again in the case of failure of the processor. The proposed scheduling algorithm tries not to concentrate tasks to processors on a die. We designed our algorithm as a parallel algorithm that achieves $O(n)$ speedup where n is the number of processors. We evaluated our method using simulations and experiments with four PCs. We compared our method with existing scheduling method, and in the simulation, the execution time including recovery time in the case of a node failure is reduced by up to 50% while the overhead in the case of no failure was a few percent in typical scenarios.**

I. Introduction

Recently, cloud computing is growing popular, and various kinds of services are being delivered by cloud computing. In order to make these services more reliable and to improve availability, we need mechanisms to prepare for failure of computing nodes. On the other hand, almost all high-performance processors are now designed as multicore processors. A multicore processor is a semiconductor chip(a die, hereafter) on which multiple processor cores are implemented, with each processor core capable of executing an independent task. In case of failure of a memory or of other devices shared among all processor cores, all processors on a die simultaneously stop execution of their tasks, in which case all of these tasks need to be recovered. Since communication between processor cores on a die has a significantly larger bandwidth and smaller latency than Ethernet or other networking technologies for connecting computers, existing task schedulers for single-core processors tend to load on one of the dies many of the tasks that depend on each other. This makes recovery of tasks more time-consuming.

In this paper, we propose a task scheduling algorithm that makes recovery from a saved state faster on multicore processor systems. The proposed method is based on the scheduling algorithm proposed by Sinnen et al.[1], which takes into account network contention and is capable of generating schedules that are highly reproducible on real computing systems. The proposed algorithm is designed as a parallelized algorithm that can achieve $O(n)$ times of speed-up when n processors are available.

We evaluated our method with simulations and experiments using 4 quad-core PCs, and the execution time including recovery time can be reduced up to 50% in case of failure, while the overhead in case of no failure was a few percent on computation-heavy tasks.

The remainder of this paper is organized as follows. Section II introduces some related works. Section III first explains the key ideas and assumptions on the proposed method, and then presents the proposed methods. Section IV shows the results of the evaluation of execution times in case of failure and no failure. Section V presents our conclusions.

II. Related Works

There are many kinds of task scheduling. In this paper, we assume that task scheduling is assigning a processor to each task, where the dependence of the tasks is represented by a directed acyclic graph(DAG). The problem to find the optimal schedule is NP-hard[4], [9], and there are many heuristic algorithms for the problems[2], [3], [4], [8], [12].

List scheduling is a classical task scheduling method that assigns the processor that can finish each task to the task in order of a given priority of the tasks[13]. The priority can be given by performing topological sorting on the dependence graph of the tasks, for example.

Wolf et al. proposed a task scheduling method in which multiple task graph templates are prepared beforehand, and the assigned processors are determined according to which templates fits the input task[5]. This method is suitable for highly loaded systems, but does not take account of network contention or processor failure.

In many task scheduling methods, there is no consideration of network contention. In these methods, a network is assumed to be ideal communication paths where there is no bandwidth limitation or latency, and sometimes schedules generated by these methods cannot be well reproduced on real computer systems. Sinnen et al. made a network contention model and proposed a scheduling method based on it [1]. In their experiments, improvement of accuracy and efficiency of system utilization are shown by generating schedules while considering network contentions. In their method, failure of computing devices is not considered.

Checkpointing is a method for recovering task execution in case of failure of computing nodes. Gu et al. proposed a method for recovery from failure based on checkpointing in stream processing systems[6], [7]. In this method, a state of a node is saved when execution and transfer of the resulting data of each task node to the subsequent processing node is completed. In case of processor failure, it finds the closest ancestor node which is not affected by the failure and recovers the processing results from the saved state in that node.

Park et al. proposed a scheduling algorithm called team scheduling that assigns a stream program to a multicore architecture[11]. This scheduling improves control over buffer space and achieves lower latency and deadlock-free feedback loops while maintaining similar synchronization overhead to the existing methods.

As far as we surveyed, there is no scheduling method that takes account of multicore processor failure and network contention. In this paper, we formulate the problem of task scheduling for multicore processor systems that reduces recovery time from processor failure taking account of network contention, and propose a heuristic method for the problem.

III. Proposed Method

In this section, we first describe the key idea of our proposal, and define the terms used in this paper. Then, we describe assumptions made in this paper and formulate the problem. After that, we explain the proposed method.

A. Key Idea

A task graph is a DAG that represents the dependence between tasks in a group of tasks. Fig.1(a) shows an example of a task graph consisting of 3 task nodes 1, 2 and 3. It shows that tasks 1 and 2 can be executed in parallel, while task 3 can only be started after outputs from tasks 1 and 2 have been fully delivered to the processor that executes task 3. A processor graph is a graph that represents the topology of a network between processors. Fig.1(b) is an example processor graph showing that 4 processors are connected by 3 network switches. In this paper, we model a dual core processor as a combination of two processors and a switch that connects these processor cores. Thus, this graph can be viewed as a network connecting 2 dual core processors with one network switch. Task scheduling is generating an assignment of a processor graph node to each of the task graph nodes.

Fig.2(a) shows a schedule generated by a method that does not take account of processor failure, from the task graphs shown in Fig.1(a) and Fig.1(b). In this schedule, task nodes 1 and 2 are assigned to processors A and B that are on a same die, and task node 3 is also assigned to processor A, since the communication between processors on a die is significantly faster than that between processors on different dies. The bottom figure in Fig.2(a) shows the schedule in this case, and it represents that task nodes 1 and 2 are executed by processor nodes A and B in parallel. After that, task node 3 is executed on processor node A. As we described in Section II, we can prepare for processor failure by saving the state to the computing node after finishing execution of each task node. However, in systems that utilize multicore processors, one multicore processor failure results in simultaneous multiple processor failure, and this makes the problem more complex. In the example shown in Fig.2, if the die on which processor nodes A and B are implemented fails when task node 3 is being executed, all of the results of execution of task nodes 1, 2 and 3 are lost. In order to recover task nodes 1 to 3 in this case, we need to execute task nodes 1 to 3 again, which more than doubles the execution time, including recovery, that is required in the case of no failure.

Fig.3 shows the schedule generated by the proposed method. If a processor failure occurs in the same timing as described above, we only lose the execution result of task node 3. Since the results by execution of task nodes 1 and 2 are stored in the computing node that houses processor nodes A and B, we can

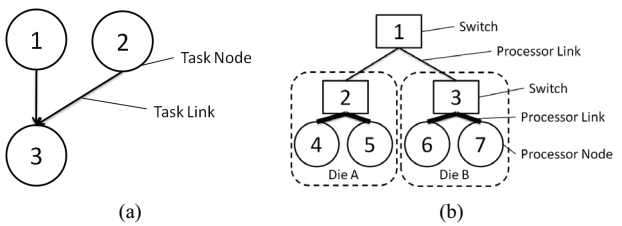


Fig. 1. (a) Example task graph (b) Example processor graph

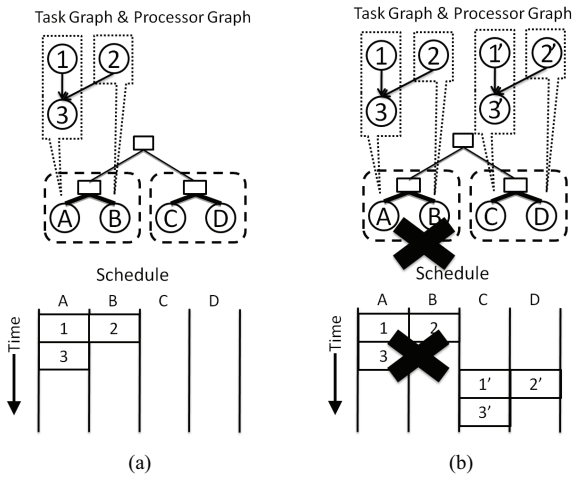


Fig. 2. (a) Schedule by existing method (b) Recovery with existing method

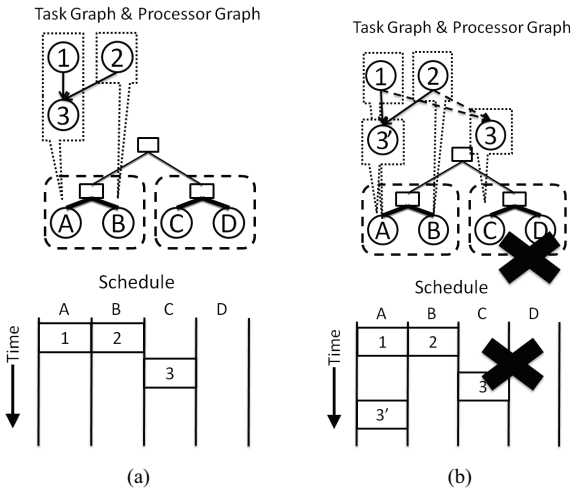


Fig. 3. (a) Schedule by proposed method (b) Recovery with proposed method

use these results to re-execute task node 3 on processor node A, and in this way we can make a quick recovery. On the other hand, if no failure happens, we need extra communication time between two dies. This imposes overhead on the proposed method. We will explain how to minimize this overhead.

B. Definition

Here, we define terms used in this paper. The symbols used are listed in Table I.

Task graph A task graph is a DAG in which each node represents a task to be performed. The computation time to execute task v, which is a node in the graph, is denoted $w(v)$. A directed arc in the graph is called a task link, and a task link from node v_a to v_b indicates that task v_a must be completed before task v_b begins. A task link e also represents communication between two nodes, and if the data transfer is performed using an inter-die communication link, it requires $c(e)$ length of time in order to finish the data transfer. The set of all task nodes and the set of all task links are denoted **V** and **E**, respectively. The task graph is denoted $G = (\mathbf{V}, \mathbf{E}, w, c)$. In this paper, the network model is based on the model proposed by Sinnen et al.[1], and we assume that the following two conditions are satisfied: (i) one processor can execute only one task at a time, (ii) A task node cannot be executed until all execution of parent nodes and all corresponding data transfers are finished. Hereafter, the i-th task node is denoted n_i. The set of all parent nodes of a node n is denoted $pred(n)$.

TABLE I
SYMBOLS USED IN THIS PAPER

Symbol	Meaning
G	Task graph
V	Set of all task nodes
E	Set of all task links
H	Processor graph
P	Set of all processor nodes
R	Set of all processor links
e_{ij}	Task link from i-th task node to j-th task node
$proc(n)$	Processor assigned to task node $n \in V$
$Proc(N)$	Set of processors assigned to task nodes in set $N \subseteq V$
$pred(n_i)$	Set of all parent nodes of n_i

Processor graph A processor graph is a graph that represents the network topology between processors. A node with only one link is called a processor node. A processor node corresponds to one processor core. A node with two or more links is called a switch. A switch is not capable of executing a task but only relays communication. An edge is called a processor link, and it represents a communication link between processors and switches. One multicore processor is represented by multiple processor nodes, a switch and processor links connecting them. The set of all processor nodes and the set of all processor links are denoted **P** and **R**, respectively. The processor graph is denoted $H = (\mathbf{P}, \mathbf{R})$.

C. Assumptions

In this paper, we make the following assumptions.

Each processor node saves the state to the computing node housing that processor node when each task is finished. A state contains the data transferred to the processor nodes that will execute the child nodes of the completed task node. If a processor failure occurs, a saved state which is not affected by the failure is found in the ancestor task nodes. For the lost results, some tasks are executed again using the saved state. We assume that only output data from each task node is saved as a state, which is much smaller than the complete memory image, as in [6]. Thus, saving a state can be performed by just replicating the output data on the main memory, and we assume that saving a state does not consume time or resources.

We assume that only a single fail-stop failure of a multicore processor can occur. When there is a failure of a multicore processor, all processor cores on the die just stop execution simultaneously and never perform a faulty operation. All states saved in the computing node that houses the failed multicore processor are lost in the failure. A failure can be detected by interruption of heartbeat signals. Heartbeat signals are reliably delivered with the highest priority without delay, and a failure is detected within 1 second.

We assume that the failed computing node automatically restart, and after predetermined time (about 1 minute), all of the failed multicore processor comes back online. When a multicore processor is rebooting, it cannot execute a task or perform communication.

When data transfer is performed over network links between two processor nodes, due to bandwidth limitation these network links cannot perform other data transfers. This is called network contention. In this paper, we use the network contention model proposed by Sinnen et al.[1], and we assume the following conditions are satisfied: if data are transferred through a series of processor links, downstream links cannot start data transfer before upstream links; communication inside a same die finishes instantly; all processors on a same die share a network interface that can be used to communicate with devices outside the die; all communication links outside dies have the same bandwidth.

D. Problem Definition

Here, the length of schedule s is denoted $sl(s)$. Suppose that when a schedule s is executed, a fault happens at task node v, and we make the schedule for the remaining tasks by Sinnen's scheduling algorithm. The entire schedule, including the schedule before and after recovery, is denoted $rs(s, v)$.

As inputs to the problem, a task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ and a processor graph $H = (\mathbf{P}, \mathbf{R})$ and which processor belongs to each die are given. Our scheduling problem is to find the schedule s such that:

$$minimize \max_{v \in \mathbf{V}} sl(rs(s, v)).$$

E. Proposed Algorithm

We first explain the overview of the proposed method, and then explain the details using pseudocodes.

The critical path in a task graph is the path from the first task to the last task which has the longest execution time, and this influences greatly the determining of the schedule length. If all the tasks in the critical path are assigned to processors on a die, and a fault happens at the last task in the critical path, the entire execution time of the schedule is doubled, and this is the worst case. We can distribute the tasks in the critical path over multiple dies in order to improve recovery time, but this could increase the overhead in case of no failures, due to extra communication between dies. On the other hand, the last task in the critical path has the largest influence on the recovery time. We therefore distribute the last $nmove$ tasks in the critical path over multiple dies and try to find the best $nmove$. Since the proposed algorithm only tries to distribute the tasks in the critical path, it can quickly generate the schedule.

Algorithm 1 List scheduling

INPUT: Task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ and processor graph $H = (\mathbf{P}, \mathbf{R})$.

1: Sort nodes $n \in V$ into list L, according to priority scheme and precedence constraints.
2: **for** each $n \in L$ do **do**
3: Find processor $p \in \mathbf{P}$ that allows earliest finish time of n.
4: Schedule n on p.
5: **end for**

The classical list scheduling algorithm is shown in Algorithm 1. In the list scheduling, each task is assigned to the processor that allows the earliest finish time of the task, in descending order of *bl*, that is the length of remaining schedule. In the classical list scheduling, network contention is not considered.

The algorithm proposed by Sinnen, et al. is shown in Algorithm 2, which is an enhancement of the list scheduling algorithm by adding network scheduling. In this algorithm, Algorithm 3 is called when findProcessor function is called. Below, we give some explanation for the pseudocode.

Algorithm 2 Scheduling considering network contention

INPUT: Task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ and processor graph $H = (\mathbf{P}, \mathbf{R})$.

1: Sort nodes $n_j \in V$ into list L in descending order of bl, according to precedence constraints.
2: **for** each $n \in L$ do **do**
3: Find processor $p = findProcessor(\mathbf{P}, n_j)$.
4: **for** each $n_i \in pred(n_j)$ in a definite order **do**
5: **if** $proc(n_i) \neq p$ then **then**
6: determine route $R = [L_1, L_2, ..., L_l]$ from $proc(n_i)$ to p.
7: schedule e_{ij} on R.
8: **end if**
9: **end for**
10: schedule n_j on p.
11: **end for**
12: **return** the schedule.

Algorithm 3 Processor selection considering network contention

INPUT: Task node n and set of processors $\mathbf{P}$.

1: **return** $p \in \mathbf{P}$ that allows earliest finish time of n, taking account of network bandwidth usage.

Line 2 to 11: Each task node $n_j \in L$ is assigned a processor in order of the position in L.

Line 3: The processor assigned to n_j is determined by *findProcessor* function. Reserved bandwidth in line 7 is referred here.

Line 4 to 9: Bandwidth of e_{ij} is reserved for the network route between the processor assigned to n_i (which is the parent node of n_j) to the processor assigned to n_j.

The pseudocode for *findProcessor* function in Algorithm 2 is shown in Algorithm 3, and this function corresponds to line 3 in Algorithm 1. Unlike the classical list scheduling, a processor is selected according to both the computation time and the communication time considering bandwidth reserved at line 7 in Algorithm 2.

The proposed algorithm is shown in Algorithm 4. When Algorithm 2 is called at line 3, Algorithm 5 is used as a findProcessor function. Algorithm 5 selects the processor that can finish the given task first, except the processor to which the parent task is assigned. Since the proposed algorithm is based on Algorithm 2, it generates schedules taking account of network contention. Since the loops in Algorithm 4 are independent, these loops can be executed in parallel. We generate the schedule for recovery after a fault using Algorithm 2 and Algorithm 3.

The pseudocode for the proposed method is shown in Algorithm 4. Below, we give some explanation for the pseudocode.

Line 2 to 13: $nmove$ is the number of task nodes moved to another die. For each of all task nodes, we generate a schedule where a failure occurs during execution and a recovery is performed. We find the longest schedule from them, and find

Algorithm 4 Proposed method

INPUT: Task graph $G = (\mathbf{V}, \mathbf{E}, w, c)$ and processor graph $H = (\mathbf{P}, \mathbf{R})$.

1: $F = \emptyset$.
2: **for (in parallel)** $nmove = 1$; $nmove \leq$ number of tasks in critical path **do**
3: $s_1 =$ Schedule generated by Algorithm 2 and Algorithm 5. (refer to the body text)
4: $S = \emptyset$.
5: **for (in parallel)** $failtask = 1$; $failtask \leq$ number of all tasks **do**
6: Find set T of all tasks that will be executed after recovery if task $failtask$ fails.
7: $s_2 =$ schedule generated by Algorithm 2 and Algorithm 3 with T as the given task nodes and processors are available only after finishing current tasks or rebooting.
8: $s =$ concatenated schedule of s_1 and s_2
9: $S = S \bigcup \{s\}$.
10: **end for**
11: $f =$ the longest schedule in S.
12: $F = F \bigcup \{f\}$.
13: **end for**
14: Find the shortest schedule g in F.
15: Let s_3 be the original schedule without failure corresponding to g.
16: **return** s_3.

Algorithm 5 Processor selection for proposed method

INPUT: Task node n and set of processor nodes $\mathbf{P}$. $nmove$ is defined in Algorithm 4.

1: **if** $pred(n) \neq \emptyset$ **then**
2: **if** n is in the first $nmove$ tasks in the critical path **then**
3: **if** there is a processor which is on different die than the $proc(pred(n))$ **then**
4: **return** the processor that can finish the task except processors on the same die as $Proc(pred(n))$, considering network contention.
5: **else**
6: **return** $p \in \mathbf{P}$ that allows earliest finish time of n, considering unused bandwidth between p and $Proc(pred(n))$.
7: **end if**
8: **else**
9: **return** $p \in \mathbf{P}$ that allows earliest finish time of n, considering unused bandwidth between p and $Proc(pred(n))$.
10: **end if**
11: **end if**

$nmove$ that makes the longest schedule shortest.

Line 3: Here, Algorithm 2 is called, and when *findProcessor* is called, Algorithm 5 is executed instead of Algorithm 3. Algorithm 5 selects a processor so that task nodes are not concentrated on a single die. We describe the details later.

Line 5 to 10: For task node $failtask$, a schedule is generated assuming that a failure occurs at $failtask$ and recovery is performed, by concatenating the schedule before failure and the schedule for the recovery.

Line 7: Using Algorithm 2, recovery schedule is generated. Here, Algorithm 3 is executed when $findProcessor$ function is called in Algorithm 2.

Since the iteration from line 2 to line 13 and line 5 to line 10 are independent from each other, these loops can be executed in parallel.

The pseudocode for the processor selection in the proposed method is shown in Algorithm 4. Below, we give some explanation for the pseudocode.

Line 2: Checks if task node n is in the last $nmove$ tasks in the critical path.

Line 3 to 7: Find set of processors $Proc(pred(n))$ that are assigned to the set of task nodes $pred(n)$. $pred(n)$ is the set of parent nodes of n. Then, select the processor that allows earliest finish of n from the processors excluding $Proc(pred(n))$.

Line 4: If there exist dies excluding $Proc(pred(n))$, then select the processor that allows earliest finish of n from them.

Line 6: If there is no die except $Proc(pred(n))$, then select the processor that allows earliest finish of n from all processors.

Line 9: If task node n is not in the last $nmove$ tasks in the critical path, then select the processor that allows earliest finish of n from all processors.

Since the proposed method is based on the algorithm proposed by Sinnen et al, it allows scheduling taking account of network contention.

IV. Evaluation

In order to evaluate the overhead and to seek improvement of recovery time in case of failure, we compared our method with two other methods using a real computing system consisting of 4 PCs. We compared the following three methods.

Contention The scheduling algorithm proposed by Sinnen, et al.[1], that takes account of network contention. After a failure, the task graph that contains the remaining task nodes that need to be executed including the task nodes whose results are lost is generated. Then a schedule for this task graph is generated using Sinnen's method.

Proposed The proposed method. The schedule after failure is generated in the same way as Contention method.

Interleaved This scheduling algorithm tries to spread tasks to all dies as much as possible. This algorithm is also based on Sinnen's algorithm, but it assigns tasks to all dies in turn. The problem of existing methods to deal with a failure is that they tend to assign dependent tasks to one die, and this algorithm is intended to improve this situation. However, this does not take account of the efficiency in case of no failure, where a large overhead is expected. The schedule after failure is generated in the same way as with the Contention method.

We used Robot Control, Sparse Matrix Solver and various random graphs from the Standard Task Graph Set[10], [14] as the input task graph to the three methods.

A. Environment

We used a computing system consisting of 4 PCs connected with Gigabit Ethernet. Each PC has an Intel Core i7 920 (2.67GHz) as the CPU, Intel Gigabit CT Desktop Adaptor EXPI9301CT(PCI Express x1) as the network interface card and 6.0G bytes of memory. We measured execution time with an originally developed program running on Java(TM) SE Runtime Environment (build 1.6.0_21-b07, 64bit) and Windows 7 Enterprise (64bit). We used the standard TCP sockets to transfer data between these PCs.

We used the processor graph that represents the above computing system. We assumed that each PC has 4 processors, and we did not count the hardware threads provided by hyper-threading. We used 450Mbps as the bandwidth of processor links outside the dies, which is the measured value using the above system.

We measured the time for Ubuntu 10.04 operating system running on VMware to reboot after pushing the reset button, and it took 25 seconds on average. Thus, we used 25 seconds as the time required for rebooting after failure. In the experiments, we reproduced the failure by suppressing the corresponding processors to perform computation or communication for 25 seconds, instead of performing a hard reset on the computers. We set the experiments so that each failure would be detected in 1 second.

In order to determine the worst timing of failure, we first made the schedule by Sinnen's method and found the task that takes the largest execution time including recovery when a failure occurs during execution of the task. We made the system fail when this task is being executed.

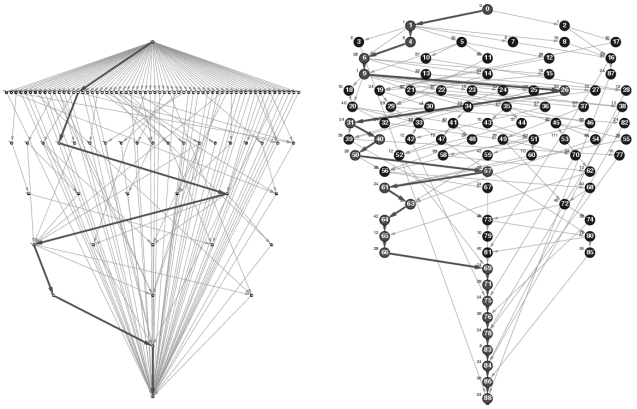
B. Sparse Matrix Solver

Fig.6(a) shows the task graph for the Sparse Matrix Solver that has 98 tasks and 177 links. This graph represents a sparse matrix solver of an electronic circuit simulation generated by the OSCAR FORTRAN compiler. This graph has a relatively high level of parallelism.

Fig.4 and Fig.5 show the results of experiments in case of failure and no failure. The graphs show the measured results on the system with the three methods and the simulated results for the proposed method. We can see that there are some difference between the simulated results and the results on the real system. This is mainly because of the instability of network bandwidth between PCs.

Fig.4 shows the execution time in case of failure. The Proposed method has a shorter execution time than the other two methods regardless of the number of dies (or PCs). In the case of 4 dies, in fact, the Proposed and the Interleaved methods, achieve significantly better execution time than the Contention method.

Fig.5 shows the execution time in case of no failure. The Proposed method has better execution time than the Interleaved method, and the maximum overhead against the Contention method was 12% at maximum.



(a) (b)

Fig. 6. (a) Sparse Matrix Solver (b) Task graph for Robot Control

C. Robot Control

Fig. 6(b) shows the task graph for the Robot Control that has 90 tasks and 135 edges. This graph represents a Newton-Euler dynamic control calculation for the 6-degrees-of-freedom Stanford manipulator, which has a lower level of parallelism compared to the Sparse Matrix Solver. The experimental results for this graph are shown in Fig. 7 and Fig.8. In this experiment, we used only 2 cores in each PC, since the task graph does not have enough size for using all 4 cores in the PCs.

Fig.7 shows the results in case of failure. Proposed method has the best execution time regardless of the number of dies. When the number of dies is 4, in fact, the Proposed method achieves more than 50% of speed-up against the Contention method.

Fig.8 shows the results in case of no failure. The maximum overhead for the Proposed method against the Contention method was 10% at the maximum. The Interleaved method has 30% overhead, which is much larger than the Proposed method. The Proposed method always had smaller overhead than the Interleaved method in this experiment.

The Contention method generates schedules that execute fast when there is no failure. However, many execution results of task nodes are lost when a failure occurs, and that increases the time for recovery. Especially when there are many dies, the Contention method requires larger recovery time.

In case of no failure, the Proposed method has larger overhead than the Contention method. This is due to the extra communication between dies. However, the overhead is much less than that of the Interleaved method, which, as much as possible, spreads tasks over dies.

D. Random Graphs

We used various random graphs with 50 task nodes. The number of links and the level of parallelism differ for each graph. We also varied the CCR of the graphs, where CCR is

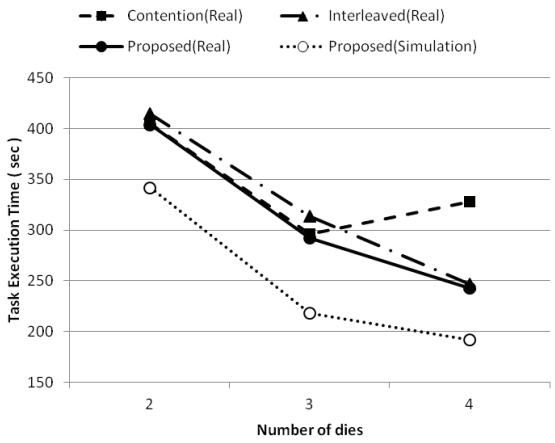


Fig. 4. Sparse matrix solver w/failure

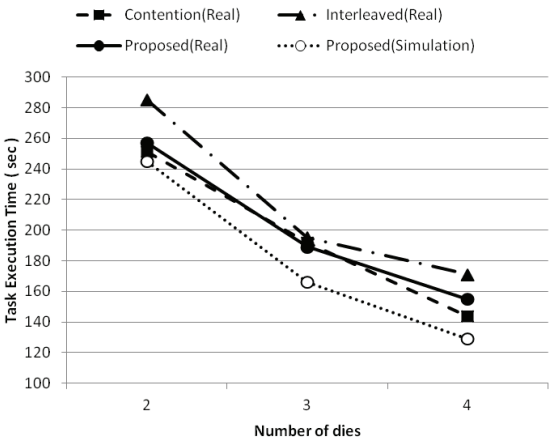


Fig. 5. Sparse matrix solver, no failure

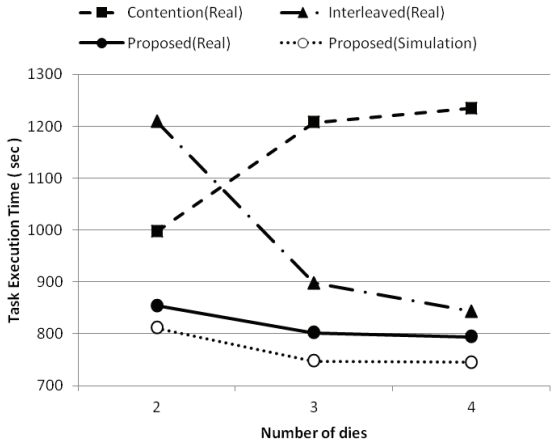


Fig. 7. Robot Control w/failure

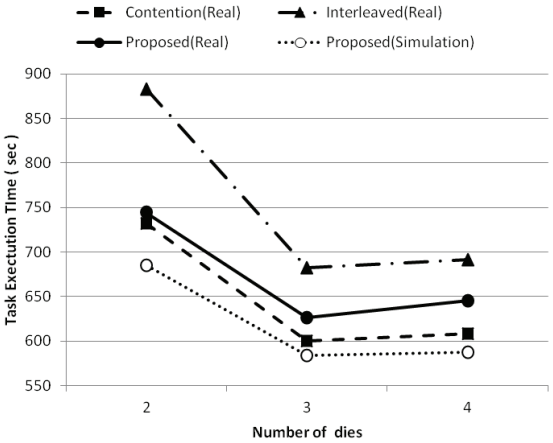


Fig. 8. Robot Control, no failure

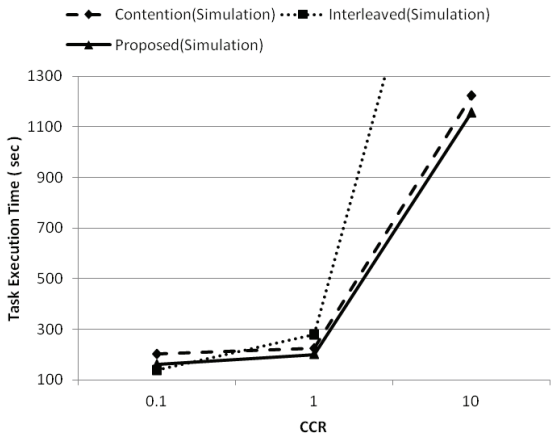


Fig. 9. Random Task Graph w/failure

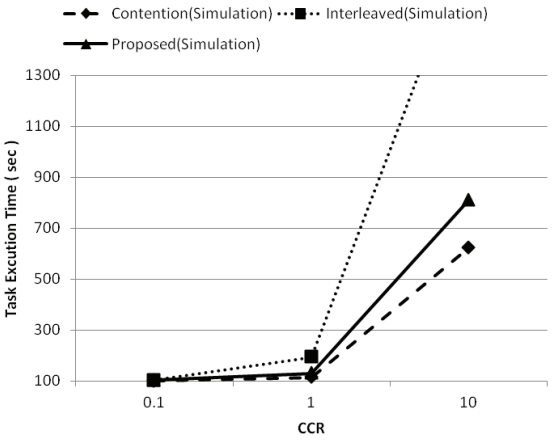


Fig. 10. Random Task Graph, no failure

the total communication time divided by the total computation time, and a large CCR means a communication-heavy task. We measured the execution times when CCR is 0.1, 1 and 10. In this experiment, we fixed the number of dies to 4 and each die has 4 cores.

Fig.9 shows the results in case of failure. The Proposed method has a better execution time than the Contention method regardless of CCR. When CCR is 0.1, the Interleaved method is slightly better than the Proposed method. This is because in computation-heavy tasks, spreading tasks over processor dies is sometimes very effective. However, when CCR is 1 or larger, the Proposed method was the best.

Fig.10 shows the results in case of no failure. This graph shows that as CCR increases, the overhead for the Proposed method and Interleved method increases. This is because the overhead basically comes from the communication time, and more communication is needed in communication-heavy tasks to spread tasks over processor dies. This is especially notable in Interleved method. On the other hand, even when CCR is 10, the overhead for Proposed method was 35%, and it succeeded to improve the result by 6% in case of failure.

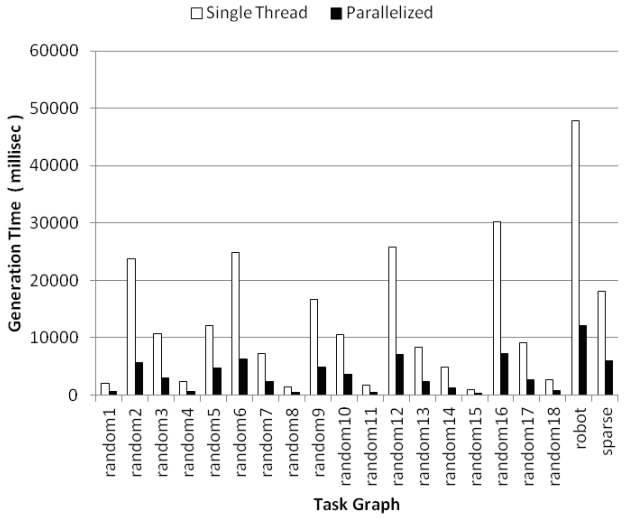


Fig. 11. Schedule generation time

E. Schedule Generation Time

Fig.11 shows the time taken to generate schedules for each task graph when the scheduling algorithm is executed with a single thread and with 8 threads. We used a PC with Intel Core i7 920 processor with hyper threading to measure these results. The proposed algorithm achieved up to 4 times of speed-up.

V. Conclusion

In this paper, we formalized a task scheduling problem for multicore processor systems that reduces recovery time from checkpoints when a single fail-stop failure occurs. We also proposed a scheduling algorithm based on the existing method that takes account of network contention. The basic idea of the proposed scheduling algorithm is to suppress the scheduler in order to assign dependent tasks to processors on a same die. The proposed scheduling algorithm is designed as a parallel algorithm that can achieve $O(n)$ speed-up if n processors are available.

We evaluated our proposed method on a computing system consisting of 4 quad-core PCs connected via Gigabit Ethernet. We confirmed that the proposed method can improve recovery time by up to 50 percent while the overhead in case of no failure was a few percent, in typical scenarios with a 1 or less communication-to-computation ratio.

As future work, we would like to address the case where the processing time of each task node cannot be known beforehand. We are also planning to evaluate our algorithm on larger computing systems.

References

[1] Sinnen, O. and Sousa, L.A. : “Communication Contention in Task Scheduling,” IEEE Trans. on Parallel and Distributed Systems, 16, 6, pp. 503-515, 2005.

[2] Sinnen, O., To, A., and Kaur, M. : “Contention-Aware Scheduling with Task Duplication,” Proc. of JSSPP’2009, pp. 157-168, 2009.

[3] Sinnen, O. Sousa, L.A. Sandnes, F.E. : “Toward a realistic task scheduling model,” IEEE Trans. on Parallel and Distributed Systems, 17, 3, pp. 263-275, 2006.

[4] Amato, N.M. and An, P. : “Task Scheduling and Parallel Mesh-Sweeps in Transport Computations,” Technical Report, TR00-009, Department of Computer Science and Engineering, Texas A&M University, 2000.

[5] Wolf, J., Bansal, N., Hildrum, K., Parekh, S., Rajan, D., Wagle, R., Wu, K-L., and Fleischer, L. : “SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems,” Proc. of the 9th ACM/IFIP/USENIX International Conference on Middleware, pp.306-325, 2008.

[6] Gu, Y., Zhang, Z., Ye, F., Yang, H., Kim, M., Lei, H., and Liu, Z. : “An empirical study of high availability in stream processing systems,” Proc. of the 10th ACM/IFIP/USENIX International Conference on Middleware, pp.23:1-23:9, 2009.

[7] Zhang, Z., Gu, Y., Ye F., Yang H., Kim M., Lei H., and Liu Z. : “A Hybrid Approach to High Availability in Stream Processing Systems,” Proc. of 2010 International Conference on Distributed Computing Systems, pp.138-148, 2010.

[8] Guo, L., Sun, H., and Luo, Z. : “A Data Distribution Aware Task Scheduling Strategy for MapReduce System,” Proc. of the 1st International Conference on Cloud Computing, pp.694-699, 2009.

[9] Tang, X., Li, K., and Padua, D.A. : “Communication contention in APN list scheduling algorithm,” Science in China Series F: Information Sciences, pp.59-69, 2009.

[10] Tobita, T. and Kasahara, H. : “A standard task graph set for fair evaluation of multiprocessor scheduling algorithms,” Journal of Scheduling, 5, 5, pp.379-394, 2002.

[11] Park, J. and Dally, W.J. : “Buffer-space Efficient and Deadlock-free Scheduling of Stream Applications on Multi-core Architectures,” Proc. of the 22nd ACM symposium on Parallelism in algorithms and architectures, pp.1-10, 2010.

[12] Macey, B.S. and Zomaya, A.Y. : “A performance evaluation of CP list scheduling heuristics for communication intensive task graphs,” Proc. of the First Merged Int’l Parallel Processing Symposium and Symposium on Parallel and Distributed Processing 1998, pp.538-541, 1998.

[13] Sinnen, O. : “Task Scheduling for Parallel Systems,” Wiley Series on Parallel and Distributed Computing, 2007.

[14] "Standard Task Graph Set" Home Page http://www.kasahara.elec.waseda.ac.jp/schedule/