情報処理学会論文誌 Vol.55 No.2 575–586 (Feb. 2014)

単一ノード故障時におけるマルチコアプロセッサシステムの回復時間を最小化するタスクスケジューリング

柴田 直樹[1,a)] 後藤田 祥平[1] 伊藤 実[1,b)]

受付日 2013年5月14日, 採録日 2013年10月9日

概要：本論文では，各計算ノードがマルチコアプロセッサであるような並列処理システムにおいて，ネットワークの輻輳を考慮しつつ，マルチコアプロセッサの単一停止故障時の回復時間を最小化するタスクスケジューリングアルゴリズムを提案する．最近開発されたプロセッサのほとんどはマルチコアであり，マルチコアプロセッサが故障した場合は，その上で実行されているタスクをすべてやり直す必要が発生する．ここでは，リカバリのために，各計算ノードで従来手法に基づくチェックポインティングを行うとを仮定する．1 つのノードで互いに依存した計算を長時間行うと，そのプロセッサが故障したときに，最近保存したチェックポイントが失われるため，かなり前のタスクから計算をやり直す必要が生じる．提案手法ではこのようなケースが生じないようなタスクスケジュールを生成する．本手法は並列アルゴリズムとして設計されており，入力サイズが十分大きければ，プロセッサ数が n のときに，$O(n)$ のスケジュール作成時間のスピードアップが達成できる．シミュレーションと実機 4 台を使用した実験により提案手法の評価を行い，故障発生時にタスク処理時間を既存手法より最大で約 30%程度短縮できる一方，故障が発生していないときのオーバヘッドが実験で用いたいくつかの設定において 3%程度に収まることを確認した．

キーワード：タスクスケジュール，マルチコアプロセッサ，単一故障

Task Scheduling Algorithm to Minimize Recovery Time in Case of Single Node Fault in Multicore Processor System

Naoki Shibata[1,a)] Shohei Gotoda[1] Minoru Ito[1,b)]

Received: May 14, 2013, Accepted: October 9, 2013

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 recently developed processors have multiple cores on a single die, and one failure of a computing node causes failure of many processors. In 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 checkpoints are taken when a node sends a result to the next node. If a series of computation that depends on former results is executed on a single multicore processor, we need to execute all part of the series of computation again in case of failure of the processor. The proposed scheduling algorithm tries to avoid generating a schedule that takes long recovery time. We designed our algorithm as a parallel algorithm that achieves $O(n)$ speedup if the input size is sufficiently large where n is the number of processors. We evaluated our method using simulations and experiments with four PCs. By comparing our method with existing scheduling method, we confirmed that the execution time including recovery time in case of a node failure is reduced by 20 to 30% with a few percent of overhead in the execution time in case of no failure.

Keywords: Task scheduling, multicore processor, single fail-stop failure

[1] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology, Ikoma, Nara 630–0192, Japan
[a)] n-sibata@is.naist.jp
[b)] ito@is.naist.jp

1. 序論

近年，クラウドコンピューティングをはじめとする多数の計算機が接続された環境での分散コンピューティングシ

© 2014 Information Processing Society of Japan

ステムが利用されている．クラウドコンピューティングは社会の重要なインフラとなっており，様々なサービスが提供されている．このようなシステムに信頼性を持たせ，可用性を向上させるためには，計算機の故障に備える仕組みが必要となる．このため，故障の発生確率についての調査が行われており [1], [2]，また様々な故障耐性手法が提案されている．一方で，近年高いパフォーマンスを持つプロセッサはいずれもマルチコアプロセッサとして設計されており，これらは 1 つの半導体チップ（ダイ）上に複数のプロセッサコアが実装されている．それぞれのプロセッサで別のタスクを並列して実行することが可能である．計算ノードの電源等が故障すると，ダイ上のすべてのプロセッサが同時に停止し，処理を継続するためにはこれらの上で実行されていた複数のタスクを同時に回復する必要が生じる．既存のシングルコアプロセッサ向けのタスクスケジューリング手法では，通信時間を削減し，パフォーマンスを向上させるために，1 つのダイ上のプロセッサにタスクを集中させてしまう傾向がある．この場合，故障が発生した際，複数のプロセッサ上のタスク処理結果を一度にすべて失ってしまい，タスクの回復にかかる時間が増加する．

本論文では，一時的に動作不能に陥った計算機ノードがリブートによって速やかに回復するケースを想定し，単一のプロセッサが停止故障し，一定時間（1 分程度）経過後，再び利用可能になると仮定する．このような故障を対象とする理由は，データセンタ内で観測されたノード故障の大部分が一時的なものであり，Google データセンタでは，1 つのノードが故障してから復帰するまでの時間が，約 50%の割合で 1 分以下であるからである [2]．本論文では，クラウドコンピューティング環境において，故障が発生した際，高速にリカバリが可能なようにタスクスケジューリングを行う手法を提案する [3]．また，提案手法では，停止故障が発生した場合に，マルチコアプロセッサの 1 つのダイ上にタスクの割当てが偏ることを防ぐため，停止故障が発生した場合に影響が大きくなるタスクを別のダイに割り当てる．また，無故障時の，タスク処理時間に対するオーバヘッドが少ない．提案手法は，Sinnen らの，ネットワークの輻輳を考慮した既存のタスクスケジューリング手法 [4] を拡張したものであり，実際のデバイス上での再現性の高いスケジュールを生成することができる．

故障が発生した場合の対応手法はチェックポインティングを利用する．チェックポインティングはタスクの実行が終了したタイミングで，タスク実行結果のバックアップであるチェックポイントを保存しておく手法で，計算ノードの故障が発生したとき，これらチェックポイントを再帰的にたどり，保存していた結果を再利用し，故障が発生していないノードでのタスク再処理は行わず，故障によって失われたタスクのみ再度スケジューリングを行い全体の処理を継続する．また，本手法は並列アルゴリズムとして設計されており，プロセッサ数が n のときに，$O(n)$ 倍のスケジュール生成の高速化が達成できる．

シミュレーションとクアッドコアの実機 4 台を使用した実験で，20 種類のタスクグラフを用意し，それぞれの場合において故障が発生したとき，故障が発生していないときの両方で既存手法 [4] と提案手法の比較を行い，有効性の評価を行った．提案手法では，故障発生時にタスク処理時間を既存手法より最大で約 30%程度短縮できる一方，故障が発生していないときのオーバヘッドは約 3%程度に収まることを確認した．また，タスクグラフの計算時間と通信時間の割合を変化させたシミュレーションにおいても，つねに提案手法の方が既存手法に比べ，タスク処理時間を短縮できている結果となっている．シングルスレッドでのスケジュール生成時間と並列アルゴリズムによるスケジュール生成時間を比較したところ大幅な時間短縮が達成できていることを確認した．

以降 2 章では，関連研究について述べる．3 章ではまずタスクスケジューリングおよび諸定義について述べ，問題を定式化した後，提案アルゴリズムについて述べる．4 章では，シミュレータと実機を使った実験による提案手法の評価について述べる．5 章でまとめを述べる．

2. 関連研究

タスクスケジューリングは手法により割当て方法が様々であるが，ここでのタスクスケジューリングとは，タスク間の依存関係を非循環有向グラフ（DAG）を用いて表し，プロセッサにタスクを割り当てる手法である．最適スケジュールを見つける問題は NP 困難 [4] であり，この問題に対するヒューリスティックな解法に関する多くの既存研究が存在する．リストスケジューリングは古典的なタスクスケジューリング手法であり，タスクを定められた優先度の順に，最も早く完了できるプロセッサに割り当てる．最も単純なリストスケジューリングでは，タスク間の依存関係グラフに対してトポロジカルソートを行い，この順をタスクの優先度として使用する．

計算機が故障してもタスク処理を継続するための手法としてチェックポインティングがある．Gu らは高可用性システムでのストリームデータ処理環境で，計算機の故障が発生することを想定し，タスクの処理データを回復させる手段としてチェックポインティングに基づく手法を提案している [5]．この手法では，タスクノードの実行が終了し，そのプロセッサに接続されたタスクグラフ上での子ノードを実行するプロセッサにデータの転送が完了した時点でチェックポイントを保存する．プロセッサの故障が発生したとき，故障発生前にタスクグラフ中で実行が終了しているタスクノードが実行されたプロセッサをたどり，故障していないプロセッサのチェックポイントを利用して処理を回復する．Benoit らはタスクスケジューリングでの

ⓒ 2014 Information Processing Society of Japan

故障発生に着目し，故障耐性を持つヒューリスティックな手法を提案している [6]．時間制限のあるアプリケーションでは，要求された時間内に処理結果が得られることを保証するために，レイテンシ制約や故障耐性が重要な概念になる．そのため，レイテンシを最小化すること，すなわち処理時間を最小化するタスクの割当てを達成することを目的とした故障耐性手法が提案されている．Benoit らはタスクの複製を基本とした手法を用いることにより，固定された故障発生数でタスク処理時間の短縮を達成しており，シミュレーションの結果では，比較手法よりオーバヘッドが小さく，タスク処理時間が短縮できていることが示されている．Maehle らは単一故障の発生に対応したタスクスケジューリングを提案している [7]．この手法では故障が発生した場合，チェックポイントを利用して，ロールバックによってスケジューリングを動的に生成する．すべての入力データに対してチェックポイントを取得し，タスクが故障によって失われたとき，自動的に失ったタスクを残りのノードでリスタートするものとなっている．また，グラフィカルなインタフェースが用意されており，並列プログラムの可視化が行え，プログラマによる開発を手助けできるものとなっている．

タスクスケジューリング手法の多くは，ネットワークの輻輳を考慮していない．そのため，ネットワークが遅延のない理想的な通信路として扱われ，タスクの処理に必要な現実の計算時間が正確に反映されていない．Sinnen らはネットワークの輻輳をモデル化し，このモデルに基づいたスケジューリング手法を提案した [4]．文献中の実験では，実環境を用いてタスクスケジューリングを評価しており，ネットワーク状況を考慮することにより，スケジュールの正確性や処理効率が向上することを示している．今日ではデータセンタでの計算ノードの数が劇的に増加しており，ツリー構造に接続された，それぞれの通信路のコストが等しい典型的なネットワークトポロジでは，通信の衝突が頻発しており，通信速度の低下が発生している．Al-Fares らは通信の動的なスケジューリングを行うことでネットワークリソースを効率的に利用できる手法を提案している [8]．ネットワークの流量の大きくなるリンクを検出し，これらのリンクに流れるデータ量がアルゴリズムによって見積もられた後，実際に通信経路を決定している．Mu らは組み込み機器向けのビデオ圧縮やデジタルコミュニケーションのようなアプリケーションで効果を発揮するタスクスケジューリング手法を提案している [9]．この手法ではリストスケジューリングにおける，タスクをプロセッサに割り当てる優先度として，タスクグラフ中の各ノードごとの子ノードと，親ノードの数を同時に考慮する．また，critical child と呼ばれる，先行ノードだけでなく後続ノードも考慮した，タスクを割り当てるプロセッサを決定するためのアルゴリズムを提案しており，この 2 つを組み合わせ，タスク処理時間を短縮している．Wang らは，古典的なリストスケジューリングを拡張して，光グリッドネットワーク環境でネットワークのスケジューリングを考慮したタスクスケジューリング手法を提案している [10]．この手法ではグリッドの使用量をもとにタスク間のデータ転送で利用するパスを決定している．その結果，タスク処理時間の短縮だけでなく，ネットワークリソースの使用量の削減もできていることが示されている．

また，マルチコアプロセッサを考慮したタスクスケジューリング手法も提案されている．Xia らは 3 つのフェーズからなるマルチコアプロセッサを搭載したシステム向けのタスクスケジューリング手法の提案を行っている [11]．各フェーズではそれぞれ，タスク処理を行うスレッドを，入力されたグラフに応じてグループに分割し，次に，グループ内でプロセッサに割り当てるタスクの決定，最後にグループ内でタスクを実行する．このマルチコアプロセッサを考慮した手法により，既存手法よりタスク処理時間の面で性能向上している．Ahmad らはゲーム理論を用いた，マルチコアプロセッサ向けのタスクスケジューリング手法を提案している [12]．この手法ではヘテロジニアス，ホモジニアスの両方のマルチコアプロセッサを搭載したアーキテクチャに対応したスケジューリングが扱われており，スケジューリングの目標はエネルギー消費を最小化することである．すなわちタスク処理時間を最小化することである．この手法ではゲーム理論を採用しており，問題を協力ゲームとして定式化し，効果を得ている．

以上のように，様々な観点からタスクスケジューリングの研究がなされてきている．しかし，ネットワーク上における制約とマルチコアプロセッサの停止故障を同時に考慮したスケジューリング手法は，著者らの知る限りでは存在しない．本論文では，ネットワークの輻輳およびマルチコアプロセッサを主とした計算機環境でマルチコアプロセッサが停止故障することを想定したタスクスケジューリング問題を定式化し，その問題を解くタスクスケジューリングアルゴリズムを提案する．

3. 提案手法

この章では，提案手法のキーアイデアについて述べた後，用語の定義および本論文中での仮定について述べる．その後，問題の形式化を行い，提案手法について説明する．

3.1 基本方針

並列実行可能なひとまとまりのタスク間の依存関係を表現したグラフをタスクグラフと呼ぶ．図 **1** (a) のタスクグラフは，このグラフで表されるひとまとまりのジョブが 1 から 3 の 3 つのタスクノードからなり，タスクノード 1 と 2 は並列に実行することができ，1 と 2 の出力が揃ったときに 3 を開始することができることを表している．また，プ

© 2014 Information Processing Society of Japan

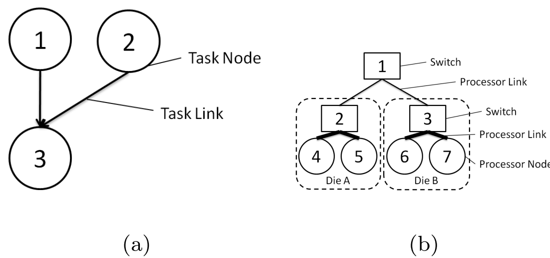


図 1 (a) タスクグラフの例，(b) プロセッサグラフの例
Fig. 1 (a) Example task graph, (b) Example processor graph.

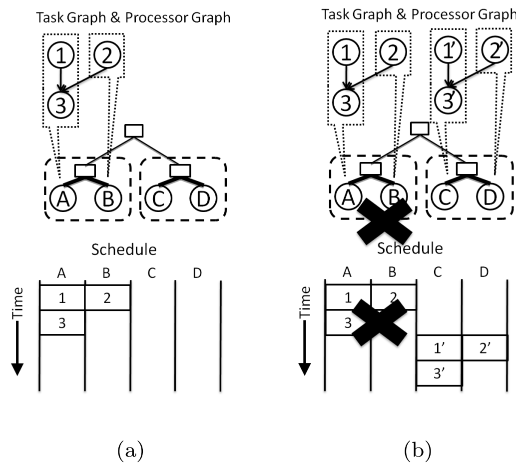


図 2 (a) 既存手法でのスケジュール，(b) 既存手法でのリカバリ
Fig. 2 (a) Schedule by existing method, (b) Recovery with existing method.

ロセッサとネットワークのトポロジを表現したグラフをプロセッサグラフと呼ぶ．図 1 (b) は，プロセッサグラフの例であり，この図は 4 つのプロセッサが，3 つのスイッチにより接続されたグラフを表す．本論文では，デュアルコアプロセッサは，2 つのプロセッサノードと，それらを接続するスイッチとしてモデル化する．したがって，図 1 (b) は 2 つのデュアルコアプロセッサがスイッチ 1 で接続されていることを表すプロセッサグラフである．タスクスケジューリングでは，タスクグラフとプロセッサグラフを入力とし，タスクグラフの各ノードに対するプロセッサグラフのノードの割当てをスケジュールとして出力する．

3.1.1 既存手法での問題

図 2 (a) は，図 1 のタスクグラフおよびプロセッサグラフに対し，故障を想定しない既存手法によってスケジュールを行った結果である．タスクノード 1，2 は，同一ダイ上のプロセッサに割り当てられ，タスクノード 3 を再び同一ダイ上のプロセッサであるプロセッサノード A または B に割り当てることで，ダイ間の通信時間を節約する．図 2 (a) の下側の図がこの場合のスケジュールを表す．2 章で述べたように，各タスクノードの実行終了時に次のタスクノードへの出力を保存しておくことにより，その後の故障に備えることができる．しかし，マルチコアプロセッサを使用したシステムでは，複数のプロセッサノードが同時に故障

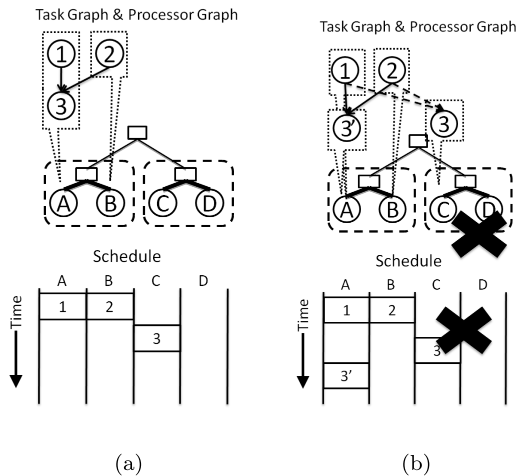


図 3 (a) 提案手法でのスケジュール，(b) 提案手法でのリカバリ
Fig. 3 (a) Schedule by proposed method, (b) Recovery with proposed method.

し，それにともない保存したチェックポイントも消失するため，回復に時間がかかる．図 2 (a) の例において，プロセッサノード A，B が実装されたダイでタスクノード 3 の処理が終了する直前に故障が発生した場合を考えると，プロセッサノード A，B の両方で処理されていたタスクノード 1，2，3 のすべての処理結果を失ってしまう．その後，処理を回復するにはタスクノード 1〜3 をすべて再度実行する必要があるため，最終的にタスクノードすべてが完了するまでに，2 倍以上の時間がかかる．この場合のスケジュールは図 2 (b) のようになる．

3.1.2 提案手法での改善

提案手法によるスケジュールは，**図 3** (a) のようになる．上記と同様のタイミングでタスクノード 3 実行終了直前に故障が発生した場合，失われる実行結果は，既存手法と異なりタスクノード 3 のみである．タスクノード 1，2 の実行結果は，プロセッサノード A と B に保持されているため，その結果を利用してタスクノード 3 の処理をプロセッサノード A で再度実行することにより，回復時間が短縮できる（図 3 (b) 参照）．一方，故障が起きなかった場合について考えると，タスクノード 1 と 2 を実行するプロセッサと 3 を実行するプロセッサが同じダイ上にないため，通信時間が生じる．この通信時間が既存手法と比べた場合のオーバヘッドとなる．提案手法では，このオーバヘッドが小さくなるようなスケジュールを生成する．

3.2 諸定義

本節では，提案手法で必要になる用語について述べる．また，以降で使用する記号を**表 1** にまとめる．

タスクグラフ タスクグラフ G とは並列実行可能なひとまとまりのジョブを表現したものである [14]．タスクグラフは DAG で表すことができ，頂点をタスクノードと呼び，1 つのプロセッサで実行するジョブの一部を表現している．

© 2014 Information Processing Society of Japan

表 1 本論文で扱う記号
Table 1 Symbols used in this paper.

記号	意味
G	タスクグラフ
$\mathbf{V}$	タスクノードの集合
$\mathbf{E}$	タスクリンクの集合
H	プロセッサグラフ
$\mathbf{P}$	プロセッサノードの集合
$\mathbf{R}$	プロセッサリンクの集合
e_{ij}	i 番目のタスクノードから j 番目のタスクノードへのタスクリンク
$w(v)$	タスクノード v の計算時間
$c(e_{ij})$	タスクリンク e_{ij} の通信時間
$proc(n)$	タスクノード n が割り当てられているプロセッサ
$Proc(N)$	タスクノードの集合 N が割り当てられているプロセッサの集合
$pred(n)$	タスクノード n の親タスクノードの集合

タスクノード v の処理に必要な時間を $w(v)$ と表す．辺をタスクリンクと呼び，タスクノード間で必要となる通信を表している（図 1 (a) 参照）．タスクリンク e の通信に必要な時間を $c(e)$ と表す．タスクノードおよびタスクリンクすべての集合をそれぞれ，$\mathbf{V}$，$\mathbf{E}$ とすると，タスクグラフは $G = (\mathbf{V}, \mathbf{E}, w, c)$ と表される．本論文で扱うタスクグラフは Sinnen らのモデル [4] に基づいており，次の 2 つの条件が成り立つとする．(1) 同じプロセッサに割り当てられたタスクは同時実行できず，1 つずつ順番に処理される．(2) タスクノードは，そのすべての親タスクノードの処理が終了し，出力データの転送が終了するまで実行できない．また，あるタスクノード n の親ノードの集合を $pred(n)$ と表す．タスクノード n に対してそのタスクを実行するプロセッサを $proc(n)$ で表し，タスクノードの集合 N に対して，それらを実行するプロセッサの集合を $Proc(N)$ で表す．

プロセッサグラフ プロセッサグラフ H とはプロセッサとネットワークのトポロジを表現している [14]．図 1 (b) にプロセッサグラフの例を表す．リンクを 1 つ持つ頂点をプロセッサノードと呼び，プロセッサの 1 つのコアを表す．リンクを 2 つ以上持つ頂点をスイッチと呼び，これらは計算能力を持たず，データを中継する役割のみを持つ．辺はプロセッサリンクと呼び，プロセッサノードおよびスイッチ間の通信路を表している．1 つのマルチコアプロセッサは，複数のプロセッサノードと 1 つのスイッチ，それらを結ぶプロセッサリンクにより表される．プロセッサノードおよびプロセッサリンクすべての集合をそれぞれ $\mathbf{P}$，$\mathbf{R}$ と表す．プロセッサグラフは $H = (\mathbf{P}, \mathbf{R})$ と表される．

3.3 仮定

本論文では，下記の仮定をおく．

チェックポインティング 各プロセッサノードは，各タスクの終了時に，タスクグラフ上で子ノードにあたるタスクノードに割り当てられたプロセッサに送るデータをチェックポイントとして保存する．プロセッサノードの故障が発生したとき，実行していたタスクのタスクグラフ上での先祖にあたるノードをたどり，故障していないプロセッサノードからバックアップを参照し，子孫にあたるプロセッサノードが再計算していくことによって，故障時に損失した処理結果を回復する．チェックポイントへの書き出しに必要な時間はメインメモリ上での転送のみなので 0 とし，チェックポイントによるリソースの消費は考えないものとする．

故障の発生 故障のモデルとして，マルチコアプロセッサの単一の停止故障を考える．停止故障が発生すると，プロセッサ自体の機能が停止し，誤動作することはない．同一ダイ上のすべてのプロセッサが同時に停止し，それらに割り当てられたタスクはチェックポイントとともにすべて消滅するものとする．プロセッサノードの故障の検出は，ハートビート信号が途切れることにより検出する [13]．ハートビートは，データ転送に優先して配送され，遅延が起こることや，消失することはないとする．ハートビートによる故障の検出は，故障の発生から一定時間（1 秒程度）以内に行われるとする．

停止故障の発生から，一定時間（1 分程度）経過後，故障した計算機は（リブートにより）再び利用可能になるとする．この間，その計算機ではタスクの実行を行うことができないほか，スイッチとその計算機間のネットワークは使用できないものとする．

ネットワークの輻輳 ネットワークの輻輳とはタスクノードが次のタスクノードへのデータを転送する際に，他のタスクノード間の転送によってネットワークが利用されているとき，帯域の制限により同時に転送できないことである．ネットワークの輻輳のモデルとして，Sinnen ら [4] のモデルを仮定する．そのため，以下の 3 つの制約条件を本研究でも適用する．(1) タスクノード間の複数のデータ転送は同時に行えない．(2) 複数のリンクを経由してデータを転送する場合，下流のリンクでのデータ転送の開始時間は上流のデータ転送の開始時間より早くできない．(3) データ転送の上下問わず同一リンク上で複数の転送は同時に行えない．さらに同一ダイ上のプロセッサ間の通信時間は 0，ダイ間のリンクの帯域はすべて一定とする．1 つのダイ上の全プロセッサは，他のダイと通信するためのネットワークインタフェースを共有するものとする．

3.4 問題定義

スケジュール s のスケジュール長を $sl(s)$ と表記する．あるスケジュール s を実行中に，タスクノード v で故障が

© 2014 Information Processing Society of Japan

Algorithm 1 リストスケジューリング

入力：タスクグラフ $G = (\mathbf{V}, \mathbf{E}, w, c)$，プロセッサグラフ $H = (\mathbf{P}, \mathbf{R})$
1: 先行制約と手法ごとの優先度に基づき $\mathbf{V}$ 中のタスクノード n をソートし，リスト L に代入．
2: **for** each $n \in L$ **do**
3: n が最も早く終了できる $\mathbf{P}$ 中のプロセッサ p を見つける．
4: p に n をスケジュール
5: **end for**
6: **return** スケジュール

発生し，故障後の再スケジュールを Sinnen らの手法によって決める．このようにして得られる故障前の処理からリカバリを経てすべてのタスクを実行するまでのスケジュールを $rs(s, v)$ と定義する．本問題の目的関数は以下のようになる．すなわち，目的関数を最小化するような，故障が発生していない場合のスケジュールを問題の出力とする．

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

3.5 提案手法の概要

一般に，タスクグラフにおいて最初のタスクから最後のタスクに至るパスの中で実行時間が最大であるクリティカルパスがスケジュール長を決定する．故障が発生するケースの中で，最悪のケースは，クリティカルパス上のタスクがすべて同じプロセッサに割り当てられ，最後のタスク実行時に故障が発生した場合である．この場合，実行に必要な時間は約 2 倍になる．クリティカルパス上のタスクを複数のダイのプロセッサ上に分散させることで故障時のリカバリ時間を改善することが可能であるが，クリティカルパスの長さは故障が発生していない場合のスケジュール長に強く影響しているため，やみくもにクリティカルパス上のタスクを分散させると，故障が発生していない場合のオーバヘッドが大きくなってしまう．一方，クリティカルパス上のタスクの中で，故障時の影響が最も大きいのは一番下にあるタスクである．そこで，提案手法では，下から順にタスクを異なるダイに移動していくことで故障の影響を小さくすることを試みる．移動する数を一番下の 1 個から，順に増やしてクリティカルパスのタスクすべてまで試し，その中で最も良いスケジュールをアルゴリズムの出力として選択する．

3.6 古典的なリストスケジューリング

古典的なリストスケジューリングのアルゴリズムを Algorithm 1 に示す．リストスケジューリングでは，残りのスケジュール長の大きい順に各タスクノードをプロセッサノードに割り当てる．タスクノードが割り当てられるプロセッサノードは，そのタスクノードを最も早く実行終了できるプロセッサノードである．古典的なリストスケジューリングではネットワークの輻輳を考慮しない．

Algorithm 2 ネットワークの輻輳を考慮したスケジューリング

入力：タスクグラフ $G = (\mathbf{V}, \mathbf{E}, w, c)$，プロセッサグラフ $H = (\mathbf{P}, \mathbf{R})$
1: 先行制約と手法ごとの優先度に基づき $\mathbf{V}$ 中のノード n をソートし，リスト L に代入．
2: **for** each $n_j \in L$ **do**
3: $findProcessor(\mathbf{P}, n_j)$ を通してプロセッサ p を見つける．
4: **for** $pred(n_j)$ 中の n_i **do**
5: **if** $proc(n_i) \neq p$ **then**
6: $proc(n_i)$ から p へのリンク $R = [L_1, L_2, ..., L_l]$ を決定する．
7: R に e_{ij} をスケジュール
8: **end if**
9: **end for**
10: n_j を p にスケジュール
11: **end for**
12: **return** スケジュール

3.7 ネットワークの輻輳の考慮

Sinnen らの手法は，タスクを割り当てるプロセッサの選択が古典的なリストスケジューリングと異なる．Sinnen らの手法を Algorithm 2 で示す．以下では，各行の動作について説明する．

1 行目では，タスクグラフの先行制約等に基づき，タスクノードをソートしリスト L に代入する．

2 行目から 11 行目では，このリスト L に含まれている，各タスクノード n_j を先頭から順に取り出してプロセッサを割り当てるためのループ処理である．この処理をタスクノードごとに繰り返すことにより，ネットワークの利用を考慮したスケジュールが生成される．

3 行目では，タスクノードが割り当てられるべきプロセッサを選択するために findProcessor 関数（後述）を呼び出す．これによってタスクノード n_j に割り当てるプロセッサを決定する．

4 行目から 9 行目では，タスクノード n_j の親タスクノードに該当する n_i に割り当てられたプロセッサから，タスクノード n_j に割り当てるプロセッサ p へのネットワーク帯域の割当てが行われる．これは，該当するプロセッサリンクへのタスクリンクの割当てである．

10 行目では，タスクノード n_j がプロセッサ p に割り当てられることにより，ネットワークの利用を考慮したタスクノード n_j の割当てが完了する．

findProcessor 関数は，Algorithm 1 の 3 行目が書き換わったものである．この関数は，プロセッサノードの集合とプロセッサに割り当てたいタスクノードを引数として，どのプロセッサノードにタスクノードを割り当てると，タスクノードの実行終了が最も早く終了するか探索し，みつけたプロセッサノードを返す関数である．Algorithm 2 では findProcessor 関数として，Algorithm 3 を呼び出す．Algorithm 3 では，古典的なリストスケジューリングで行

© 2014 Information Processing Society of Japan

Algorithm 3 既存スケジューリングでのプロセッサの選択

入力：プロセッサノードの集合 **P** とタスクノード n

1: **return P** 中の各プロセッサ p のネットワークの利用を考慮し，n が最も早く終了できる p.

われるプロセッサの選択とは異なり，Algorithm 2 の 7 行目でスケジュールされたプロセッサリンク上の利用状況が参照され，必要となる通信時間と計算時間の両方を考慮して，その中でタスクノードが最も早く終了できるプロセッサを返す．

Algorithm 3 では，割当てを決めようとしているタスクノード n の親タスクノードの集合 $pred(n)$ で割り当てられているプロセッサと **P** 中の p の間で，利用されるプロセッサリンクにスケジュールされたタスクリンクを参照し，通信に必要な時間を得る．既存スケジューリングでのプロセッサの選択はこの通信時間が含まれ，n が最も早く終了できる p を返すため，ネットワークコンテンションを考慮したプロセッサが選択される．

3.8 故障発生とマルチコアの考慮

提案アルゴリズムを Algorithm 4 に示す．以下では各行の動作について説明する．

2 行目から 13 行目では，$nmove$ はクリティカルパス中のタスクノードを移動させる個数であり，これを 1 からクリティカルパスに含まれるタスクノード数まで順に変化させる．このとき，各タスクの実行中に故障が発生した場合のスケジュールを生成し，それが最長となるスケジュールを得る．$nmove$ を変化させた中で，このスケジュールが最短となるものを求める．それに対応する故障していないときのスケジュールを出力することによって，故障発生時のスケジュール長が最短となるスケジュールが得られる．

3 行目では，Algorithm 2 を呼び出し，得られたスケジュールを s_1 とする．ここでの Algorithm 2 中の findProcessor 関数は，提案手法の一部である Algorithm 5 が呼び出されスケジュールが生成される．Algorithm 5 は 1 つのダイにタスクノードの割当てが偏らないように，割り当てるプロセッサを選択する．詳細については後述する．

5 行目から 10 行目では，各タスクノード $failtask$ に対して，$failtask$ を実行中にプロセッサで故障が発生したとき，故障が発生する前に実行していたスケジュールと故障後に回復のため必要となるスケジュールを連結したものを生成する．

7 行目では，Algorithm 2 によって故障発生後のスケジューリングを行う．ここでの，Algorithm 2 中の findProcessor 関数は既存手法の Algorithm 3 である．

14 行目では，F の中でスケジュール長が最短となるものを選び出すことによって，クリティカルパス上のタスクノードの移動数ごとに生成される，故障発生時にスケジュール長が最長になるスケジュールの中から，スケジュール長が最短のものが得られる．得られたスケジュールの故障していないときのものが，スケジュール s_3 となる．

Algorithm 4 提案手法

入力：タスクグラフ $G = (\mathbf{V}, \mathbf{E}, w, c)$，プロセッサグラフ $H = (\mathbf{P}, \mathbf{R})$

1: $F = \emptyset$
2: **for (in parallel)** $nmove = 1$; $nmove \leq$ クリティカルパスに含まれるタスク数 **do**
3: $s_1 =$ Algorithm 2 (本文参照) によってスケジュールを生成
4: $S = \emptyset$
5: **for (in parallel)** $failtask = 1$; $failtask \leq$ 全タスク数 **do**
6: $failtask$ で故障が発生したとき，回復後実行されるタスク T を求める．
7: $s_2 = T$ に対して Algorithm 2 を呼び出して得られたスケジュール（ただし，入力の各プロセッサは故障発生後の一定時間は利用できず，リブート完了時に再度利用可能になる）
8: $s = s_1$ の故障前の部分と s_2 を結合して得られるスケジュール
9: $S = S \bigcup \{s\}$
10: **end for**
11: $f = S$ 中の最もスケジュール長の長いもの
12: $F = F \bigcup \{f\}$
13: **end for**
14: F 中の最もスケジュール長の短いものを選び，そのスケジュールに対応する，故障が発生していないときのスケジュールを s_3 とする
15: **return** s_3

Algorithm 5 提案手法でのプロセッサの選択

入力：プロセッサノードの集合 **P** とタスクノード n を入力，$nmove$ は Algorithm 4 中の $nmove$

1: **if** $pred(n) \neq \emptyset$ **then**
2: **if** n がクリティカルパスの末尾から $nmove$ 個に含まれる **then**
3: **if** $Proc(pred(n))$ と同一のダイにないプロセッサが存在する **then**
4: **return** $Proc(pred(n))$ と同一のダイにないプロセッサのうちネットワークの利用を考慮し，n が最も早く終了できる p
5: **else**
6: **return** $Proc(pred(n))$ と **P** 中の p のネットワークの利用を考慮し，n が最も早く終了できる p.
7: **end if**
8: **else**
9: **return** $Proc(pred(n))$ と **P** 中の p のネットワークの利用を考慮し，n が最も早く終了できる p.
10: **end if**
11: **end if**

また，2 行目から 13 行目と 5 行目から 10 行目のループ内はそれぞれ独立した処理であるため，並列に実行することが可能である．

提案アルゴリズムでの割り当てるプロセッサを移動させる処理を Algorithm 5 に示す．このアルゴリズムの各行の説明を以下に示す．

© 2014 Information Processing Society of Japan

2 行目では，移動させるタスクノード n がクリティカルパスの末尾から $nmove$ 個に含まれているか確かめる．

3 行目から 7 行目では，n の親タスクノード $pred(n)$ が割り当てられているプロセッサ $Proc(pred(n))$ を求め，得られたプロセッサが含まれるダイ以外のダイにあるプロセッサの中から最も早くタスク n が終了できるプロセッサ p を選択する．

4 行目では，$Proc(pred(n))$ 以外のダイが見つけられる場合に，それらの中から最も早くタスク n が終了できるプロセッサ p を選択する．

6 行目では，$Proc(pred(n))$ 以外のダイが見つけられない場合に，すべてのプロセッサの中から最も早くタスク n が終了できるプロセッサ p を選択する．

9 行目では，タスク n がクリティカルパスの末尾から $nmove$ 個に含まれない場合に，すべてのプロセッサの中から最も早くタスク n が終了できるプロセッサ p を選択する．

提案手法は，Sinnen らの手法に基づき，タスクノードを割り当てるごとに，タスクリンクのスケジュールが行われるため，ネットワークの輻輳を考慮したスケジューリングが可能である．アルゴリズムの 2 行目から 13 行目のループ，およびそこに含まれる 5 行目から 10 行目までのループは，入力となるデータを以前に実行した同ループの実行結果に依存しておらず，したがって並列に実行することが可能である．また，11 行目および 14 行目は，それぞれ 9 行目および 12 行目で最長・最短スケジュールへのポインタを保持しておくことにより，シングルスレッドでも定数時間で実行可能である．したがって，入力サイズが十分大きく，かつループ回数がプロセッサ数に比べて十分大きい場合は，これらのループを複数のプロセッサで並列に実行することにより，プロセッサ数に比例する高速化が可能となる．

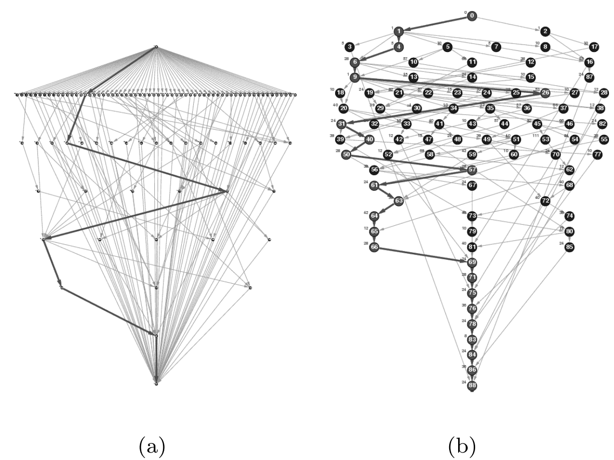
4. 評価

提案手法におけるタスクスケジューリングによってプロセッサの故障発生時のリカバリにかかる時間がどの程度改善されるか，および故障の発生していないときのタスク処理時間のオーバヘッドを，シミュレーションと実機により評価した．

4.1 比較手法

実験において，比較した手法は以下の 3 つである．

Contention ネットワークの輻輳を考慮した Sinnen らの提案したスケジューリングアルゴリズムである [4]．故障が生じた後の振舞いについては，プロセッサの故障発生の影響により，タスク処理結果が失われ，再処理が必要になったタスクノードを抽出し，それらのタスクに対して，再び Sinnen らのアルゴリズムによってスケジューリングを行う．また，故障が発生していないプロセッサ上に割り当てられていたタスクはチェックポインティングにより，タスク処理結果を保持しているため，計算なしに，再利用することが可能である．



(a) (b)

図 4 (a) Sparse Matrix Solver のタスクグラフ，(b) Robot Control のタスクグラフ

Fig. 4 (a) Sparse Matrix Solver, (b) Task graph for Robot Control.

Proposed 提案手法である．提案手法で故障が発生した場合，故障後のスケジューリングは Contention モデルと同じく，故障の影響の受けたタスクに対して Sinnen らのアルゴリズムによってスケジューリングを行う．

Interleaved すべてのプロセッサに順番にタスクノードを割り当て，タスクが 1 つのプロセッサに集中することを防ぐタスクスケジューリング手法．Sinnen らの手法をベースとして利用し，タスクの割当て先プロセッサの決定方法のみ変更を行っている．この手法では，タスク処理が集中することによって故障発生時のタスク処理時間が増加するということに対して，タスク処理を分散させることで，故障発生時の影響を小さくできると想定したものである．しかし，故障が発生しないときは，タスク処理を行うプロセッサが頻繁に変わることになるため，通信時間によってタスク処理時間が増加する．また比較手法で故障が発生した場合の振舞いについては，上記の Contention モデルと同様である．

4.2 実験設定

タスクグラフとして，Standard Task Graph Set [15], [16] の Robot Control と Sparse Matrix Solver，および 18 種類の様々なランダムタスクグラフを用いた（図 4）．

これらの 3 つのタスクスケジューリングアルゴリズムを用いて，ネットワークコンテンションおよび停止故障が発生する状況で，故障が起こらないときのオーバヘッドと，故障発生時におけるリカバリ時間を含んだタスク処理時間について比較を行った．

© 2014 Information Processing Society of Japan

4.3 実験環境

以下に示す実機を用いて評価実験を行った．

計算機 タスクを実行する計算機システムとして，PC 4 台を Gigabit Ethernet により接続したものを使用した．PC の CPU は Intel Core i7 920 (2.67 GHz), Network Interface Card は Intel Gigabit CT Desktop Adaptor EXPI9301CT（PCI Express x1），メモリ 6.0 GB を搭載し，Windows 7（64 bit）上で Java(TM) SE Runtime Environment（build 1.6.0_21-b07, 64 bit）により作成したプログラムを用いた．PC 間のネットワーク接続は TCP ソケットにより行った．

ネットワーク プロセッサグラフとして，上記のシステムに対応したものを用いた．各プロセッサリンクの帯域として，実際に上記のシステムで通信を行って得た 450 Mbps を使用した．

故障の発生 停止故障発生後のリブートに要する時間として，VMware 上の Ubuntu 10.04 でリセットボタンを押してから，OS が再起動するまでの時間を複数回計測し，その平均値である 25 秒を利用した．実験では，故障は擬似的に再現されており，故障の発生したプロセッサでは 25 秒間タスク処理や通信を行わないよう設定し，その後，再びタスク処理が実行可能になり回復のためのスケジュールが開始されるとした．故障の検出時間は 1 秒とした．

故障発生のタイミングとして，それぞれの手法においてスケジュールした場合に，各タスクにおいて故障が発生したケースのうち，リカバリ時間が最長になるものを選んだ．

4.4 性能評価実験

各実験の結果および考察を以下に示す．

実験 1：Sparse Matrix Solver この実験で用いた Sparse Matrix Solver のタスクグラフは，タスク数 98，タスクリンク数 67 であり，The OSCAR FORTRAN compiler で生成される random sparse matrix solver of an electronic circuit simulation を表現したものになっている（図 4 (a)）．**図 5** (c)，(d) はタスクグラフとして Sparse Matrix Solver を与えたときの実験の結果である．これらの図は，シミュレーションによる実行時間および実機による提案および比較手法の実行時間を示している．また，図 5 (a)，(b) はシミュレーションのみで比較した場合の結果になっている．シミュレーションと実機による提案手法の実行時間には若干の差があるが，これは主に実機においてネットワーク帯域が安定しないことが原因である．

図 5 (a) はシミュレーションによる故障発生時の比較である．図より，Proposed は Contention，Interleaved と比べ，ダイ数にかかわらずタスク処理時間が短縮されていることが分かる．また，図 5 (b) は故障が発生していないときのスケジュール長（シミュレーション上のタスク実行時間）を示している．この場合，Proposed は，Contention より通信時間のため約 3%タスク処理時間が増加しているが，Interleaved に比べて大幅に短縮されている．

図 5 (c) は実機実験での故障発生時の実行時間を示している．Proposed は Contention，Interleaved の両方より PC の数（すなわちダイ数）によらずタスク処理時間が短くなっており，特にダイ数が 4 のとき，Proposed と Interleaved は Contention に比べ約 26%の時間短縮を実現している．図 5 (d) は故障が発生していないときであり，Interleaved に比べ，Proposed ではタスク処理時間の増加はいずれの場合でも抑えられており，Contention と比べたオーバヘッドも最大で 12%程度に抑えられている．

Sparse Matrix Solver のタスクグラフは並列度が高く，Interleaved でもプロセッサ数が増加するにともなってタスク処理の並列化が行えるようになり処理時間が減少していく．

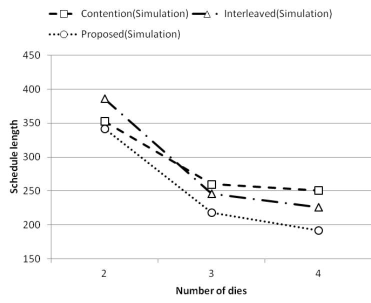
実験 2：Robot Control この実験で用いた Robot Control のタスクグラフは，タスク数 90，タスクリンク数 135 で，Newton-Euler dynamic control calculation for the 6-degrees-of-freedom Stanford manipulator に利用されるものである．並列度は Sparse Matrix Solver に比べ低く，タスクを並列に割り当てにくい傾向にある．シミュレーションによる，比較の結果を図 5 (e)，(f) に，実機における結果を図 5 (g)，(h) に示す．

図 5 (e) はシミュレーションによる故障発生時の比較である．提案手法では，いずれの場合においても Contention，Interleaved よりタスク処理時間が短縮できており，ダイ数 4 の場合に約 30%の高速化が達成されている．既存手法で，ダイの数が増加しているにもかかわらず，タスク処理時間が増加している要因については，タスクグラフに対してダイの数に余裕が生じ，より効率的なタスク割当てを行った結果，故障発生時に同時にタスク処理結果を損失してしまうタスクが増えたためと考えられる．図 5 (f) は故障の発生していないときの結果を示しており，上記で述べたように既存手法ではダイの数が増加することによってタスク処理時間が短縮されていることが分かる．提案手法では，既存手法よりわずかにタスク処理時間が増加しているが，故障発生時の短縮率と比較すると小さいものである．

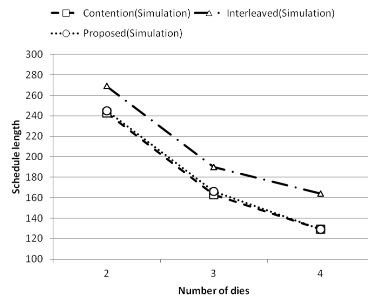
図 5 (g) は実機実験で停止故障発生時の比較である．すべての場合において，提案手法が最も処理時間を短縮できており，特にダイ数が 4 のとき，Contention に比べタスク約 35%の高速化が実現できている．図 5 (c) および (g) において，ダイの数が増加するに従い Contention での実行時間が増加する理由は，故障を想定しない Contention では 1 つのダイ上のプロセッサにタスクを集中させてしまう傾向があり，故障が発生した際タスクの回復に時間がかかるからである．

図 5 (h) は故障が発生していないときの結果で，Contention と比較して，Proposed のオーバヘッドは最大でも約 10%に抑えられている．Interleaved のオーバヘッドは

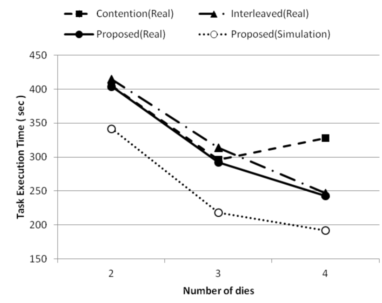
© 2014 Information Processing Society of Japan



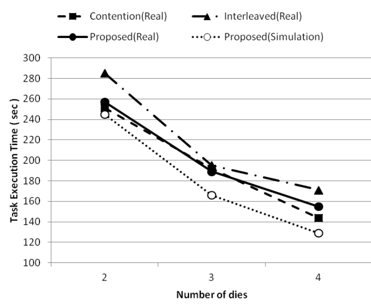
(a) Sparse 故障時 (シミュレーション)



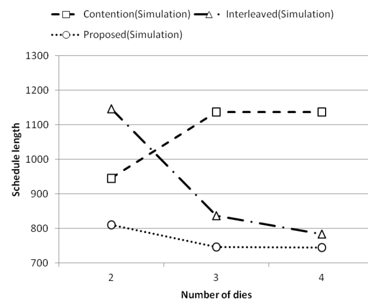
(b) Sparse 無故障時 (シミュレーション)



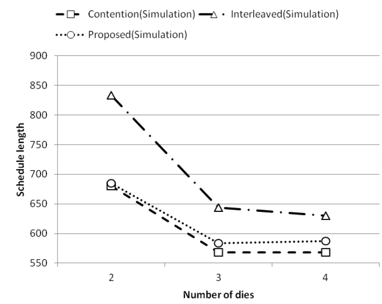
(c) Sparse 故障時



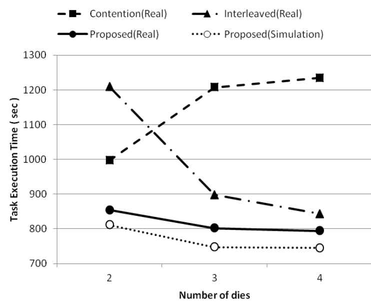
(d) Sparse 無故障時



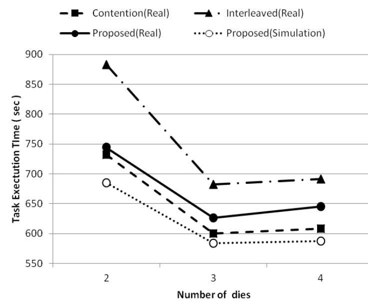
(e) Robot 故障時 (シミュレーション)



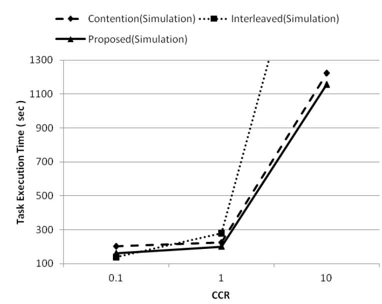
(f) Robot 無故障時 (シミュレーション)



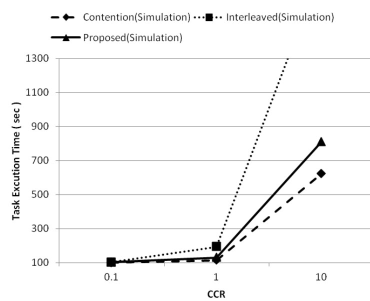
(g) Robot 故障時



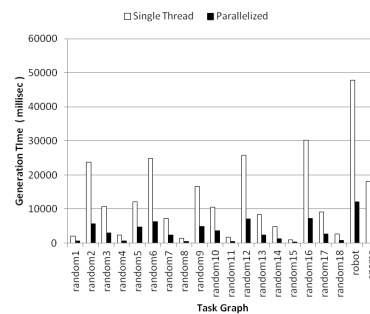
(h) Robot 無故障時



(i) Random 故障時



(j) Random 無故障時



(k) スケジュール生成時間

図 5　評価結果

Fig. 5　Results of evaluation.

© 2014 Information Processing Society of Japan

約 30%となっており，提案手法のオーバヘッドが小さいことが分かる．また，提案手法は比較手法に比べつねに小さなオーバヘッドになっていることが分かる．

Contention では，故障が発生していないときに有利となるスケジュールを行うため，故障発生時に多くのタスクノードの結果を損失してしまい，回復のために再度実行するタスクノードが増加してしまう．そのため，特にプロセッサ数が増加した場合に，Contention ではタスク処理時間が大きく増加している．

故障が発生していないとき，提案手法は既存手法よりタスク処理時間が増加している．これは，データ転送の増加が生じるためである．しかし，無作為にプロセッサ割当てを分散させる比較手法に比べ，提案手法のタスク処理時間の増加は小さいものであり，また，故障発生時のタスク処理時間も比較手法，既存手法の両方と比べても短縮されている．

実験 3：Random Graphs タスク数が 50 のランダムグラフを使用して，評価を行った．これらは，1 つのタスクノードから出力されるタスクリンク数や全体のタスクリンク数が異なり，並列度も様々である．これらのランダムグラフを使用して，平均の処理時間を計測した．また，CCR を 0.1，1，10 に変化させ，タスク処理時間を測定した．CCR はタスクグラフ中の通信量の総和を計算量の総和で割ったもので，通信量と計算量の割合を表す．この実験では，ダイ数を 4 に固定した．

提案手法では，故障発生時（図 5 (i)）にはいずれの場合でも既存手法よりタスク処理時間が短縮される結果となっている．図 5 (j) は故障が発生していないときのタスク処理時間の平均値を示している．CCR の増加にともなって，提案手法のオーバヘッドが増大していることが分かる．これは，通信量が多くなるに従い，プロセッサの割当てを分散させるために，より多くの通信が必要になるためである．CCR が 10 のときにも，オーバヘッドは 35%程度に収まり，また故障発生時には既存手法よりも短い時間でリカバリが完了することが分かる．

実験 4：スケジュール生成時間 図 5 (k) は，各タスクグラフのスケジューリングにかかった時間である．ダイ数 4 でそれぞれ，提案手法のアルゴリズムをマルチスレッドで実行した場合とシングルスレッドで実行した場合を比較している．提案アルゴリズムは，並列化によりいずれのスケジュールの生成においても，大幅にスピードアップがなされており，最大で約 4 倍のスピードアップを達成している．

5. 結論

本論文では，ネットワークの輻輳が発生し，マルチコアプロセッサ搭載の計算機が主となるデータセンタを想定した環境で，プロセッサの停止故障が 1 度発生する場合に，故障発生時のタスク回復処理を含めた処理時間を最小化する問題を定式化し，この問題に対するタスクスケジューリング手法を提案した．

提案手法のタスクスケジューリングの性能を評価するために既存手法とともにシミュレーションと実機実験で比較した．提案手法では故障が発生していないときのわずかなオーバヘッドで，故障発生時のタスク処理時間を短縮できていることが，シミュレーション，実機両方の結果から分かった．また，提案手法ではタスクスケジューリングを行う際，並列処理が可能になっておりスケジュール生成時間の短縮を達成している．

今後の課題として，より大きな計算機システムで評価を行うことや，アルゴリズムの性能向上，CCR の高いタスクグラフでのタスク処理時間のさらなる短縮があげられる．

参考文献

[1] Failure Rates in Google Data Centers - Data Center Knowledge, available from ⟨http://www.datacenterknowledge.com/archives/2008/05/30/failure-rates-in-google-data-centers/⟩.

[2] Ford, D., Labelle, F., Popovici, F., Stokely, M., Truong, V., Barroso, L., Grimes, C. and Quinlan, S.: Availability in Globally Distributed Storage Systems, *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation*, pp.61–74 (2010).

[3] Gotoda, S., Ito, M. and Shibata, N.: Task Scheduling Algorithm for Multicore Processor System for Minimizing Recovery Time in Case of Single Node Fault, *Proc. 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (*CCGrid*), pp.260–267 (2012).

[4] Sinnen, O. and Sousa, L.A.: Communication Contention in TaskScheduling, *IEEE Trans. Parallel and Distributed Systems*, Vol.16, No.6, pp.503–515 (2005).

[5] 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. 10th ACM/IFIP/USENIX International Conference on Middleware*, pp.23:1–23:9 (2009).

[6] Benoit, A., Hakem, M. and Robert, Y.: Fault tolerant scheduling of precedence task graphs on heterogeneous platforms, *IEEE International Symposium on Parallel and Distributed Processing*, pp.1–8 (2008).

[7] Maehle, E. and Markus, F.: Fault-Tolerant Dynamic Task Scheduling Based on Dataflow Graphs, *Proc. IPPS'97, Workshop on FaultTolerant and Distributed Systems*, pp.357–371 (1997).

[8] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N. and Vahdat, A.: Hedera: Dynamic Flow Scheduling for Data Center Networks, *Proc. 7th USENIX Symposium on Networked Systems Design and Implementation* (*NSDI 2010*), pp.281–296 (2010).

[9] Mu, P., Nezan, J.F., Raulet, M. and Cousin, J.G.: A list scheduling heuristic with new node priorities and critical child technique for task scheduling with communication contention, *Workshop on Design and Architectures for Signal and Image Processing* (*DASIP'09*) (2009).

[10] Wang, Y., Jin, Y., Guo, W., Sun, W., Hu, W. and Wu, M.: Joint Scheduling for Optical Grid Applications, *J. Optical Net.*, Vol.6, No.3, pp.304–18 (2007).

[11] Xia, Y., Prasanna, V.K. and Li, J.: Hierarchical Schedul-

© 2014 Information Processing Society of Japan

ing of DAG Structured Computations on Manycore Processors with Dynamic Thread Grouping, *Job Scheduling Strategies for Parallel Processing*, Lecture notes in Computer Science, Vol.6253, pp.154–174 (2010).
[12] Ahmad, I., Ranka, S. and Khan, S.: Using Game Theory for Scheduling Tasks on Multi-core Processors for Simultaneous Optimization of Performance and Energy, *Workshop on NSF Next Generation Software Program in Conjunction with the International Parallel and Distributed Processing Symposium*, pp.1–6 (2008).
[13] 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, *International Conference on Distributed Computing Systems* (*ICDCS 2010*), pp.138–148 (2010).
[14] 吉永一美，小出　洋：ストリーミングデータ処理のためのタスクスケジューリング，情報処理学会研究報告 ハイパフォーマンスコンピューティング，Vol.2006, No.87, pp.139–144 (2006).
[15] Tobita, T. and Kasahara, H.: A standard task graph set for fair evaluation of multi-processor scheduling algorithms, *Journal of Scheduling*, Vol.5, No.5, pp.379–394 (2002).
[16] Standard Task Graph Set, Home Page, available from ⟨http://www.kasahara.elec.waseda.ac.jp/schedule/⟩.

柴田 直樹　（正会員）

1996 年，1998 年，2001 年にそれぞれ大阪大学基礎工学部中退，同大学大学院基礎工学研究科博士前期課程修了，同大学院基礎工学研究科博士後期課程修了．2001 年より奈良先端科学技術大学院大学情報科学研究科助手．2004 年 4 月より滋賀大学経済学部情報管理学科准教授．2012 年 9 月より現在，奈良先端科学技術大学院大学情報科学研究科准教授．分散システム，ITS，並列アルゴリズム等の研究に従事．ACM，IEEE 各会員．

後藤田 祥平

2012 年 3 月奈良先端科学技術大学院大学情報科学研究科博士前期課程修了．現在，ヤフー株式会社に勤務．

伊藤 実　（正会員）

1977 年大阪大学基礎工学部卒業，1979 年同大学大学院基礎工学研究科博士前期課程修了．1979 年より大阪大学基礎工学部助手．1986 年より大阪大学基礎工学部講師．1989 年より大阪大学基礎工学部助教授．1993 年より奈良先端科学技術大学院大学情報科学研究科教授，現在に至る．工学博士．データベース理論，効率的なアルゴリズム開発等の研究に従事．ACM，IEEE，電子情報通信学会各会員．

© 2014 Information Processing Society of Japan