社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2006－HPC－107（12）
2006／7／31

ハイブリッドタスクスケジューリング手法の提案と評価

井村 芳和
九州工業大学大学院
情報工学研究科

小出 洋
九州工業大学情報工学部
知能情報工学科

概　要

本研究では, 並列プログラムの実行時間短縮を目的として, クリティカルパス (CP) スケジューリング手法とリストスケジューリング手法を組み合わせて使用するハイブリッドタスクスケジューリング手法を提案し, 評価を行う. CP スケジューリング手法はタスクを効率的に計算機に割り当てるが, スケジューリングに時間がかかる. リストスケジューリング手法は計算機に割り当てるタスクを実行可能タスクから無作為に選ぶので純計算の効率は良くないが, スケジューリングに要する時間は少なくて済む. 今回はタスクの数と実行計算機の数の関係により手法を変更することで実行時間の短縮を計った. タスクの数が計算機よりも多ければCP スケジューリング手法を用い, 少なければリストスケジューリング手法を用いる. この手法により, 多くの場合で既存の手法よりも少ない時間で実行終了することができた.

Implementation and Evaluation of a Hybrid Task Scheduling Method

Yoshikazu Imura
Graduate School of Computer
Science and Systems Engineering
Kyushu Institute of Technology

Hiroshi Koide
Faculty of Computer Science of
Systems Engineering
Kyushu Institute of Technology

We propose and evaluate a hybrid task scheduling method in order to reduce elapse time of parallel and distributed programs. Our task scheduling method combines the critical path (CP) scheduling method and the list scheduling method. The critical path scheduling method is efficient to assign tasks to processors, but it needs long time. The list scheduling methods is not efficient, but the scheduling time is very short. We have implemented a hybrid method which changes these scheduling methods by the number of tasks and workers. In the result, when we execute parallel and distributed programs based on 180 task graphs which include 100 tasks, faster than the original task scheduling methods at many cases.

1　はじめに

本研究では, 互いに依存関係のあるタスクから構成されている並列プログラムを実行した際の実行時間の短縮を目指す.

方法としては, クリティカルパス (Critical Path) スケジューリング手法とリストスケジューリング手法を併用したハイブリッドタスクスケジューリング手法を用いる. リストスケジューリングはタスクを割り当てる際に計算がほとんど発生しないが, 効率的かどうかを考慮しない割り当てを行う. CP スケジューリングは, タスクの割り当てに時間がかかるが, 効率的な割り当てを行う. この二つの手法の利点をうまく用いることができれば, 全体の実行時間を減らすことが出来る.

具体的な手法の切り替え方法として, タスクの数とタスクを実行する計算機の数の関係に注目し

た. 以後, このタスクを実行する計算機をワーカと呼ぶ. 今回の実験ではワーカの性能差を考えないため, ワーカの性能は全て同一のものを用いた. 実行するタスクの数がワーカよりも少なければ, タスクの順番を考えても無駄であるため, リストスケジューリング手法を用いる. 実行するタスクの数がワーカよりも多ければ, 優先するタスクを選ばなければならないため, CP スケジューリング手法を用いる. 実行したタスクグラフはランダムに作られた 180 種類のグラフで, タスクの処理は全体の実行が数秒程度になるよう for 文を用いて負荷をかけた. 結果は CP スケジューリング手法を用いた場合は 3.12 秒, リストスケジューリング手法を用いた場合は 3.1 秒, ハイブリッドスケジューリング手法を用いた場合は 2.9 秒と, 従来手法と比較して約 0.2 秒の実行時間短縮となった. この短縮された実行時間はタスクの数が増えるとさらに増えるものと考えられる. ワーカの数が少ない場合はあまり効果は見られなかったが, 全体を平均した実行時間は従来のものよりも良かった.

同一もしくは異種のコンピュータをネットワークで接続した並列分散環境下で大規模プログラムを並列処理することにより, 実行時間の短縮をはかる. 互いに依存関係を持つ複数のタスクから構成される分散プログラムの並列分散処理を行うには, タスクに対しスケジューリングを行い計算機に割り当てていく. このときのスケジューリングでタスクをどう割り当てるかにより全体の実行時間に違いが生じ, 最悪の場合は単一計算機で実行するときよりも実行時間が長くなってしまう可能性もある. タスクの個数とタスクを割り当てる計算機の台数や性能が一定であれば最適なスケジューリングは求められるが, これらが一定でない場合は強 NP 困難な問題として知られており [1], 最適な割り当てを求めることは容易ではなく, 更に全ての並列プログラムとその実行環境は同一ではない. そのため, 現在タスクスケジューリングでは, ヒューリスティックな手法が主に用いられている.

ヒューリスティックな手法である CP スケジューリング手法は, タスクの割り当てにおいては一般的に優秀とされている. しかし, タスクの割り当ての際に発生する計算時間や, タスクの割り当てに用いる CP 長を求めるための計算時間が全体の実行時間に加わる.

2 ハイブリッドタスクスケジューリング手法

並列分散環境における並列処理では分割されたタスク間に依存関係が存在するため, 一般に計算機がタスクの数と同数存在したとしてもタスク全てを割り当てて同時に実行するということはできない. 同一計算機に大量の実行可能なタスクを割り当てることは依存関係があるために実行時間の増大に繋がる. 互いに依存関係のあるタスクの集合を, 計算機にどのように割り当て, どのような順序で実行するかを決定することをタスクスケジューリングと呼ぶ. このタスクスケジューリングの方法により全体の実行時間に違いが出てくるため, タスクスケジューリングには実行時間を最小にするようなタスクの割り当てが求められる. この最適な割り当て方を求める問題を, 実行終了時間最小マルチプロセッサスケジューリング問題という [1].

しかし, 実行終了時間最小マルチプロセッサスケジューリング問題は極めて難しい最適化問題であり, NP 困難である. またタスク間に依存関係を持ち, 各タスクの計算コストが異なり, 実行サーバの数が任意であるという最適化問題は強 NP 困難となる. よって問題の規模が大きくなるにつれて, 最適解を求めるためにかかる時間が指数関数的に大きくなってしまい, 現実的な時間内に最適解を求めることが出来ない [1]. そのため, スケジューリングにはヒューリスティックな手法を用いることが多い.

タスクの割り当てを並列プログラムを実行しながら同時に行うものを, ダイナミックスケジューリングと呼ぶ. 一方, 並列プログラムをコンパイルする際にタスクの割り当てを決定しておくものを, スタティックスケジューリングと呼ぶ [3]. ダイナミックスケジューリングでは, 最適なタスク割り当てを行う際にオーバーヘッドが大きくなってしまうため, 複雑な手法を用いることは難しい. 並列分散環境では, 使用する計算機の状況が変化する. 計算機の台数やどの計算機が利用できるかわからないため, スタティックスケジューリングでは使用することができない. また, 計算機の使用状況も変化するため, コンパイル時に最適であったスケジューリングが実行時には最適でなくなってしまう. よって, 並列分散環境ではダイナミックス

ケジューリングの方が適していると判断し, 本研究ではダイナミックスケジューリングの効率化を目指し, 以下に説明する.

2.1 リストスケジューリング

リストスケジューリングとは, タスクのリストを用いた最も基本的なスケジューリング手法である. 簡単であるため, 多くのスケジューリングアルゴリズムの基礎となっている. リストスケジューリングの手順は以下の通りである.

1. タスクの集合より依存関係が解消されている実行可能タスクを全て見付ける
2. サーバリストを調べ, タスクを割り当てることの出来るタスク実行可能サーバを全て見付ける
3. 1 と 2 で得たリストを, どちらかが無くなるまでタスクをサーバに割り当て, 全てのタスクが終了していなければ 1 へ戻り, タスクが残っていなければ終了する

リストスケジューリングを計算機二台で行った例を図 1 に示す. まず 1, 2, 3 が実行可能となっていて, それを二台の計算機に割り当てる. 図 1 ではタスク番号に従い 1 と 2 を割り当てているが, 違う割り当てを行う場合もある. よって, 最悪な割り当て方を行うこともあれば, 最適な割り当てを行う場合も出てくる.

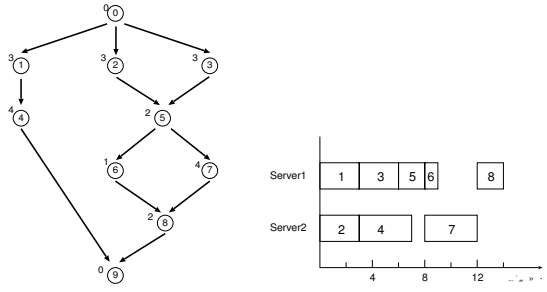

図 1: リストスケジューリング
Fig.1: List Scheduling

2.2 CP/MISF スケジューリング

CP/MISF スケジューリングとは, 基本は CP 長の大きいタスクから優先的に実行サーバに割り当てていく CP スケジューリング手法を基本としている. CP とは, タスクグラフにおける計算コストの総和が最大となる経路のことである. あるタスクから終了タスクまでの CP 上のタスクの計算コストの総和を CP 長という. MISF は Most Immediate Successors First の略で, CP/MISF スケジューリングは CP スケジューリング手法を改良したものである [1]. CP/MISF スケジューリングは, CP 長が同じであった場合は直接依存するタスクが多いものを優先する. 本研究では, CP スケジューリングと呼ぶものは全て CP/MISF スケジューリングに拡張してあるものを指す.

CP スケジューリングは経験に基づくヒューリスティックなスケジューリング手法で, たとえ実行サーバが十分にあってスケジューラが最適な割り当てを行ったとしても, CP 上のタスクを全て実行する時間が最長となる, という考え方によるものである. 経験上良い割り当て方をするということはわかっているが, その割り当て方が最適であるという保証は無い.

CP スケジューリングの手順は以下の通りである.

1. 各タスクから終了タスクが実行され終わるまでの CP 長を求める
2. タスクの集合より, 依存関係が解消されている実行可能タスクを全て見付ける
3. サーバリストを調べ, タスクを割り当てることの出来るタスク実行可能サーバを全て見付ける
4. 2 で得たリストを CP 長の大きい順にソートする. もし CP 長が等しければ依存するタスクが多いものを優先する.
5. 3 と 4 で得たリストのどちらかが無くなるまでタスクを順にサーバに割り当て, 全てのタスクが終了して無ければ 2 に戻り, 全てのタスクを割り当て終えたならば終了

CP スケジューリングを計算機二台で行った例を図 2 に示す. 最初の実行可能タスクは 1, 2, 3 であるが, CP 長の大きい順に割り当てるため優先度は 2, 3, 1 の順になる. 実行サーバは二台なので, 2, 3 が計算機に割り当てられる. 図 1 と比べると, 全体の実行時間の結果は CP スケジューリングのほうが優秀であることがわかる.

CP/MISF スケジューリングは, タスクの割り当てという点だけを見れば, 有効なタスクスケジューリング手法である. しかし, そのタスクの割り当てのために計算を要するという点に問題がある. タスクを割り当てる際に, 実行可能なタスクが 10 個あれば, 10 個の CP 長を用意し比較していくとい

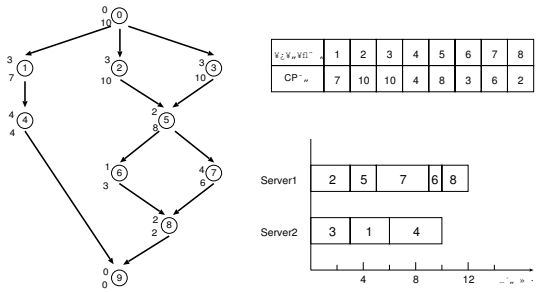

図 2: CP スケジューリング
Fig.2: CP scheduling

う計算を行う. もし同じ CP 長を持つタスクがあれば, 依存関係の数の比較も行う. これは積み重なることにより, 無視できないほどの計算時間を発生させる.

2.3 ハイブリッドタスクスケジューリング

CP/MISF スケジューリングのように, いくつかの手法を同時に用いるスケジューリング手法で有効なものはあるが, 条件により選択手法を変更する手法で有効なものは開発されていない. よって, CP/MISF スケジューリングとリストスケジューリングの二種類のスケジューリングを使い分け, 並列プログラムの速度向上をはかれないかと考え, この二つの手法を組み合わせたハイブリッドタスクスケジューリング手法を提案する.

CP/MISF スケジューリングは 2.2 節で述べた通り, タスク割り当てのために計算時間がかかるが, タスク割り当てにおいては優秀である. リストスケジューリングはタスクの割り当てに時間をかけないが, 不適切な割り当てを行う可能性が高い.

今回はワーカのスケジューリングは考えず, タスクのスケジューリングのみを行う. よって, いくら CP スケジューリングでタスクの優先順位を考えたとしても, 実行可能である全てのタスクをワーカに割り当てることができれば, 優先順位を考えることは無駄であり, リストスケジューリング手法を用いた場合と結果は変わらない. しかしワーカの数が実行可能なタスクよりも少ない場合は, タスクの中で優先順位の高いものを先に実行することは重要であるため, CP スケジューリング手法を用いる.

この方法を用いることにより, 多くのタスクグラフに適応できる汎用性を持つハイブリッドタスクスケジューリングを目指した. また, 多くのタスクグラフに適応できるということは, ある一つのタスクグラフの中での色々な状況に対応した処理もできるということである. そこから, あるタスクグラフでの処理時間の向上も計る.

3 ハイブリッドスケジューリング手法の評価

本章では, 第 2 章で提案したタスクスケジューリング手法を実装し, 並列プログラムを実行して評価を行った.

開発言語には Java を用い, タスクスケジューリングシステムにハイブリッドタスクスケジューリング手法を組み込み, 実験を行った. このタスクスケジューリングシステムは, 異なるホスト上の Java オブジェクト間でメソッドの呼び出しを行う RMI(Remote Method Invocation) を用いて通信を行っている.

評価に用いた並列プログラムは早稲田大学笠原研究室で配布されている, マルチプロセッサでの並列計算におけるタスクスケジューリングの評価用ベンチマークである Standard Task Graph(STG)[2] を実行するタスクグラフとして用意した. 評価に用いた STG は, タスク数が 100 個のランダムに作られた 180 種類のグラフである. タスクの中身の処理は for ループを行い, 回数は全体の計算時間が数秒で終わるように設定した.

3.1 少数計算機を用いた実験

3.1.1 実験環境

同一性能計算機を 5 台用い, 実験を行った. 計算機の性能は表 1 の通りである.

4 台をタスクを割り付けるワーカとして用い, 1 台をスケジューラとして用いた. スケジューラがタスクスケジューリングを行い, ワーカへタスクを割り振っていく. 構成図を図 3 に示す.

また, タスクは for ループ文を 1×10^7 回まわしてタスクのコストを作っている.

表 1: 実験用計算機のスペック
Table.1: Spec. of Machines for the Experimentations

CPU	2.6GHz AMD Opteron152
メモリ	DIMM 4GB
LAN	1Gigabit Ether
OS	Solaris10
その他	Java version 1.5.0_01

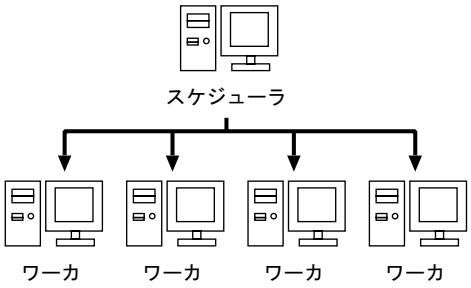

図 3: 評価に用いた環境のネットワーク図
Fig.3: Network Configration for the Experimentations

3.1.2 実験結果

それぞれの STG で 60 回実行し, その平均値の全ての平均を取ると, ハイブリッドスケジューリングは 6.7 秒, CP スケジューリングは 6.7 秒, リストスケジューリングは 6.8 秒となった. 平均値はハイブリッドスケジューリングの結果が CP スケジューリング手法よりも 0.05 秒早かったが, STG 全体の結果を見ると従来手法よりも遅い場合が目立った. 表 2 にそれぞれの手法で最速であった STG の個数を示す.

表 2: それぞれの手法で最速であった STG の数
Table.2: Summary of the Result

scheduling	個数
Hyblid	93
CP	69
List	18

具体的な結果の一部を表 3 に示す.

表 3: 実験結果の具体的な値の一例
Table.3: Examples of The Results of the Experimentations

stg \ sche	hyblid(秒)	CP(秒)	List(秒)
0000	3.6	3.5	3.8
0003	4.6	4.5	4.8
0008	7.3	7.3	7.6
0010	5.4	5.3	5.4
0014	4.4	3.9	4.3
0020	4.5	4.8	4.9
0021	3.8	3.8	3.9
0022	3.6	3.7	3.8
0061	6.7	6.6	6.7
0128	5.1	5.2	5.2
0161	6.9	7.2	7.2

3.2 計算機の数を増やして行った実験

3.1 で行った実験に用いた計算機は大量に用意することができなかったため, 別の計算機を多く用意し, 実験を行った.

3.2.1 実験環境

同一性能の計算機を 12 台用い, 実験を行った. 計算機の性能は表 4 の通りである. 12 台全てをワーカとして用い, 1 台をスケジューラを兼ねて用いた. 構成は 3 のスケジューラがワーカも兼ねたものとなっている.

また, タスクは for ループ文を 1×10^5 回まわしてタスクのコストを作っている.

表 4: 実験用計算機のスペック
Table.4: Spec. of machines for the Experimentations

CPU	PowerPC G4 1.42GHz
メモリ	1GB
LAN	100Mbit Ether
OS	MacOS X Panther
その他	Java version 1.4.2_09

3.2.2 実験結果

それぞれの STG を用いて 90 回実行し, その平均値の全ての平均を取った結果, ハイブリッドスケジューリングは 2.9 秒, CP スケジューリングは 3.1 秒, リストスケジューリングは 3.3 秒となった. 表 5 にそれぞれの手法で最速であった STG の個数を示す.

表 5: それぞれの手法で最速であった STG
Table.5: Summary of the Result

scheduling	個数
Hyblid	178
CP	1
List	1

具体的な値の一部を表 6 に示す.

表 6: 実験結果の具体的な値の一例
Table.6: Example of The Resluts of the Experimentations

stg \ sche	hyblid(秒)	CP(秒)	List(秒)
0110	2.4	2.5	2.6
0111	2.0	2.1	2.2
0112	3.2	3.5	3.5
0113	2.6	3.0	2.8
0114	2.2	2.4	2.3
0115	3.7	4.1	4.1
0116	2.7	3.3	3.3
0117	2.4	2.5	2.7
0118	4.0	4.5	4.4
0119	3.4	3.8	3.7
0031	2.4	2.5	2.3
0161	3.8	3.8	4.0

3.3 評価

mac12 台を用いた環境においては, ほぼ全ての STG での実効速度の向上が見られた. しかし, Solaris 計算機を 5 台用いた環境においては, CP スケジューリング手法のほうが速い状況が多々見られた. この原因としては, ワーカが少ない状況であれば, タスクの数がワーカよりも少ないという状況が少なくなり, 殆ど手法の切り替えが行われないということが考えられる. 手法の切り替えが行われないので, 手法を切り替えるかどうかの判断という処理が加わった分, ハイブリッドでない手法のほうが速い.

4 おわりに

今回の実験により, ハイブリッドタスクスケジューリング手法が有効に働く場合があることがわかった. 今回は, ワーカの性能を同一のもので揃え, 外部からの負荷が少ないものを用いることでワーカのスケジューリングの必要を無くした環境で実験を行った. 実際の並列分散環境ではワーカの性能が全て同一である状況は少なく, また他人と共有するものであるため負荷の変動は激しい. 今後の研究課題として, ワーカのスケジューリングを取り込んだハイブリッドタスクスケジューリング手法の開発を目指し, 実際に利用されているアプリケーションを用いて評価を行っていく予定である.

異機種環境で DAG で表される分散プログラムをスケジューリングするヒューリスティックな手法には DLS や HEFT, LMT といったものがある [4]. また, 既にハイブリッドとして実装されているもので Hyb.BMCT や Hyb.MinMin などといった手法もある [4]. 今後これらの手法と比較する必要がある.

参考文献

[1] 笠原博徳, 並列処理技術, pp. 148-153, コロナ社 (1991).

[2] Standard Task Graph, 早稲田大学笠原研究室 http://www.kasahara.elec.waseda.ac.jp/.

[3] Yu-Kwong Kwok and Ishfaq Ahmad, Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors, ACM Computing Surveys, Vol. 31, No. 4 (1999).

[4] Rizos Sakellariou and Henan Zhao, A Hybrid Heuristic for DAG Scheduling on Heterogeneous Systems, Proc. of 18th International Parallel and Distributed Processing Symposium (IPDPS '04), ipdps, p.111b (2004).