数理解析研究所講究録
第1629巻 2009年 225-233

マルチコア・マルチプロセッサ環境向け分枝限定アルゴリズムの研究

筑波大学大学院 システム情報工学研究科
木幡周治 (Shuji Kohata)*, 久野誉人 (Takahito Kuno)
GRADUATE SCHOOL OF SYSTEMS AND INFORMATION ENGINEERING,
UNIVERSITY OF TSUKUBA

概 要

複数のCPUコアを持つ，マルチコア・マルチプロセッサ環境のPCが増加しているが，それに比べその環境を無駄なく活用するための並列アルゴリズムの開発は遅れている．その背景には並列アルゴリズムの理論的難しさや，ハードウェアの機能の複雑化・多様化が挙げられる．本稿ではマルチコア・マルチプロセッサ環境のPCを用いて並列な分枝限定アルゴリズムの実装を行い，必要となる複雑な情報のやりとりをどのように行うかについて研究した．

1 はじめに

今日ではパーソナルコンピュータ (以下PC) の大部分が1台に複数のCPUコアが搭載されたマルチコア・マルチプロセッサ環境となっている．その理由としてはCPUコア単体での性能向上が集積率・発熱・消費電力と多くの問題に突き当たった事が挙げられ，この先もそのようなPCが増加していくものと思われる．

分枝限定法 [2] は最適化の諸問題を解く際によく用いられ，その手法の特性から並列実行に向いているといえる．分枝限定法の並列実行に関しては，大規模計算機での実装実験や，十分なコア数が用意できる前提の理論の中でよく言及されるが，マルチコア環境のような現実的で小規模なものに実装する際にどのくらいの利得があるか，どのような問題が生じるかについてはあまり広く知られていない．

本研究では，基本的な分枝限定法をマルチコア環境上で実装し，分枝限定法の小規模並列化に対しての本質的な課題を探りたい．

第2節ではマルチコア・マルチプロセッサの利用方法について述べる．第3節では分枝限定法の実装について述べる．第4節では計算機実験の内容と結果について述べる．第5節ではまとめと考察を行う．

*kohata@syou.cs.tsukuba.ac.jp

2 マルチコア・マルチプロセッサ環境の利用

マルチコアとマルチプロセッサは1つのPCに複数のCPUコアが搭載される点で共通した技術であるので（図1），特に今回の実験環境であるマルチコアの場合について記述する．マルチプロセッサとはキャッシュメモリ構成等に違いがあり，実装にも差がでる場合がある．

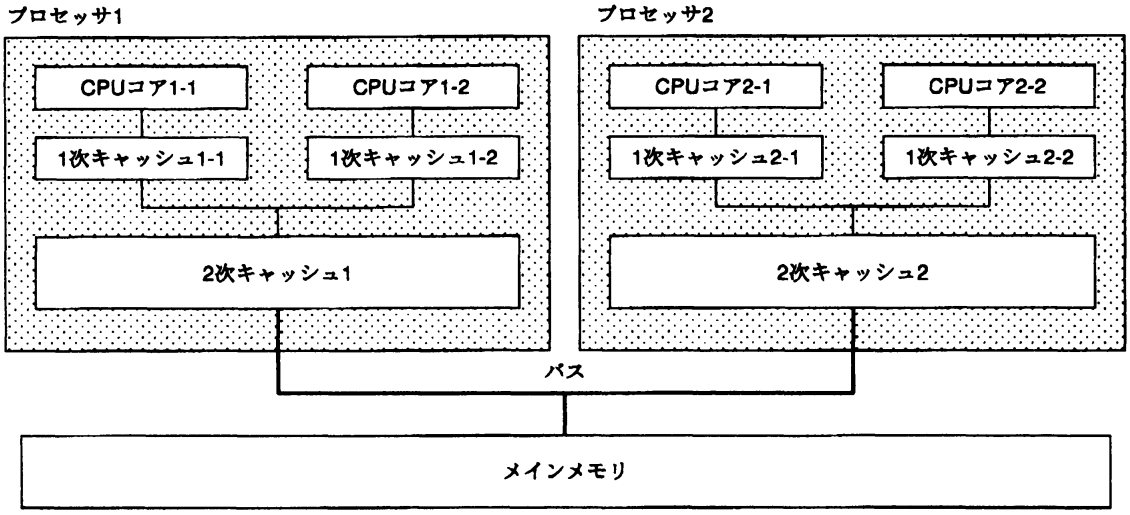

図 1: マルチコア（デュアルコア）でマルチプロセッサ（デュアルプロセッサ）な環境の例

2.1 マルチコア環境上でのプログラミング

マルチコア環境では複数の処置を同時に動かすことが可能なので，1つのプログラム上で効率よく使うためにはプログラムを複数の処理を同時に行うように記述する必要がある．具体的にはプログラムの処理をスレッド呼ばれる単位に分けてそれぞれのCPUコアで別々に実行させる．その際，それぞれのスレッドは実行単位的には別のプログラムであるが，それぞれのCPUコアはメインメモリを共有しているので，データのやりとりは共有メモリ領域にアクセスし，データの読み書きを行う．

2.2 通信のコストと問題点

共有メモリを使った通信にはキャッシュメモリに関したコストと問題点が存在する．

それぞれのCPUコアは普段のメモリ読み書きにおいては共有メモリに直接読み書きせず，CPUコアに近く高速なキャッシュメモリを活用している．よって通信のために共有メモリを使う場合，普段のメモリ読み書きよりもずっと低速に動作することになる．また，ハードウェアの実装によってキャッシュと共有メモリ上の内容の不整合が起こったり，その不整合の可能性を検出して同期処理を行ったりする場合がある．

さらに，複数のCPUコアで同時期に共有メモリの同じ箇所にアクセスするとメモリの読み込みから書き込みまでの間にデータが書き換わってしまう可能性があり (図2)．排他制御による同期

処理が重要となる (図 3).

しかし，そのような同期処理にはかなり大きい時間コストがかかり，排他制御を行わない信頼の薄い通信も状況によっては有用である．[1)]

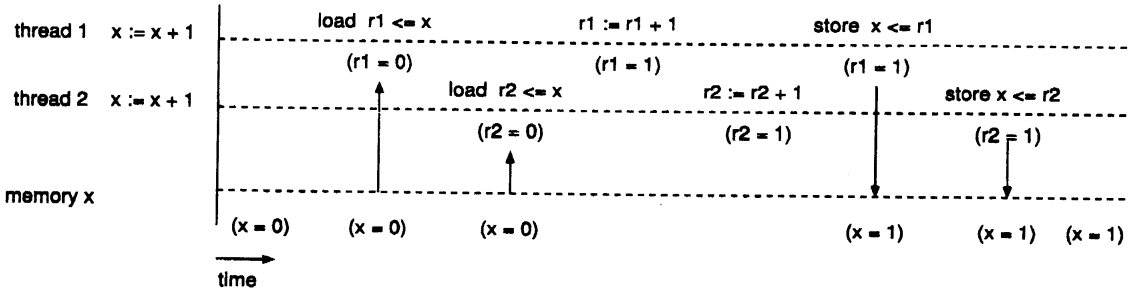

図 2: 同期処理のない共有メモリへのアクセス（x に 1 を 2 回足しても 1 しか増えない例）

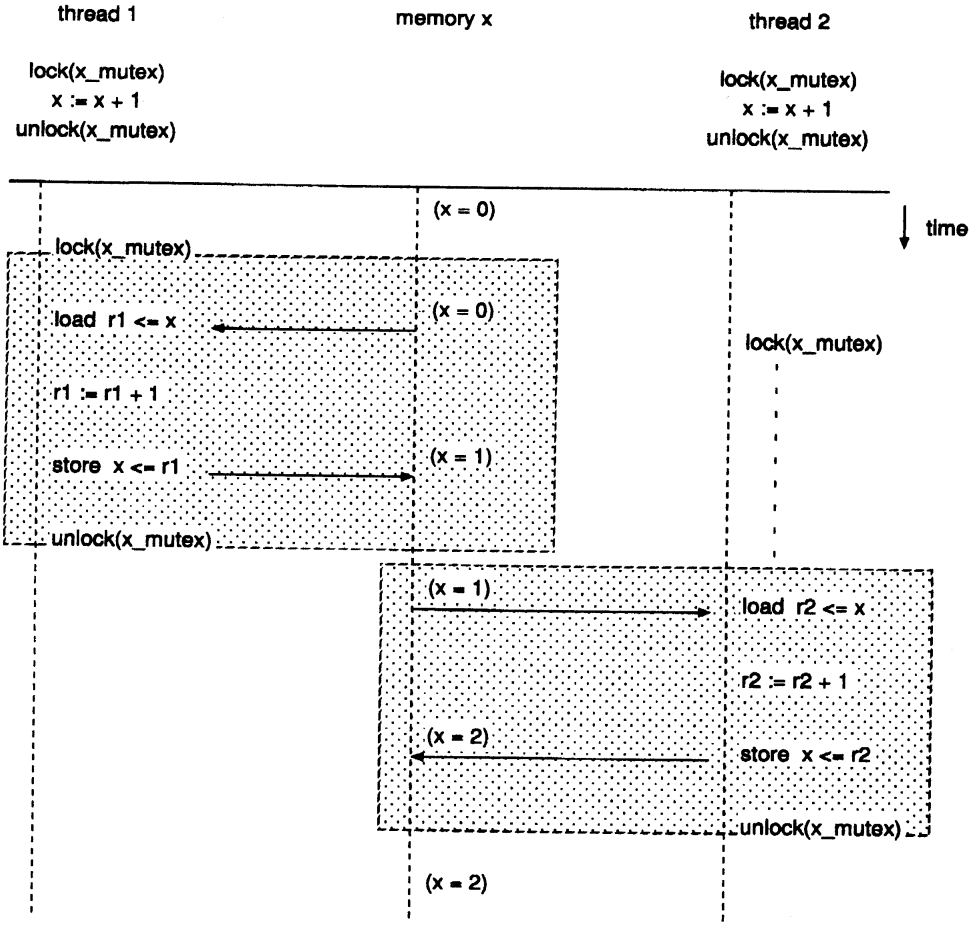

図 3: 図 2 の操作に mutex による排他制御を加えた物

[1)] 現段階ではハードウェアに強く依存してしまうので，工学プログラミングとしては推奨されていない手法ではある．

3 分枝限定法の実装

基本的な分枝限定法として0-1ナップサック問題に対する分枝限定法を実装した．今回はよりシンプルな理解のために発見的な手法による工夫を効率順でのソーティングにしぼった．

3.1 基本的な実装

0-1ナップサック問題は，大きさの決まったナップサックに品物を詰めて，その合計の価値を最大化する問題である．品物がn個あってナップサックの容量がb，j番目の品物の大きさがa_j，価値がc_jであるとき以下のように定式化できる．

$$\begin{aligned} \text{maximize} \quad &: \quad \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad &: \quad \sum_{j=1}^{n} a_j x_j \leq b \\ & \quad x_j \in \{0,1\} \ \ ,j = 1,2,\ldots,n \end{aligned}$$

(通常,a_j, c_j, bはすべて正の数．)

ただし，通常，a_j, c_j, bはすべて正の整数である．また，あらかじめ品物を効率c_j/a_jの大きい順でソーティングしてあるものとする．分枝限定法に用いる基本操作は以下の通りである．

分枝操作

添え字順に品物jを選びそれぞれ$x_j = 0$と$x_j = 1$を固定した2つの部分問題に分割し，$x_j = 1$側から深さ優先で分枝していく．

限定操作

$x_j = 1$に固定された品物の大きさの和が容量bを超えれば終端させる．xの連続緩和により得られる上界値と暫定最適値を比べ，暫定最適値が大きければ終端させる．

3.2 マルチコア上での実装

マルチコア上の分枝限定法の実装で考慮すべき点を以下に列挙する．

問題の割り当て

最初にCPUコア数分の部分問題分割を行い，それぞれのCPUコアに割り当てる．

暫定最適値共有

暫定最適値はどのCPUコアが更新しても，全てのCPUコアにその値が行き渡ることが望ましいので，CPU間の通信を行う必要がある．1つの共有メモリでやり取りする場合，そこに全てのCPUコアのアクセスが集中するため，同期処理によるコストに特に気をつけなければならない．

暫定最適値の共有は確実性が必ずしも必要でないため，信頼の薄い通信の使用も可能である．

部分問題再割り当て

CPUコアに割り当てられた部分問題が終端した場合．そのCPUコアはそれ以降休眠状態となってCPUコア資源に無駄が生じるので，そのような休眠CPUコアに部分問題を割り当て直す仕組みが必要である．

割り当てられた部分問題が終端したCPUコアは探索中の解空間が広い他のCPUコアに対して部分問題の要求を行う．要求を受けたCPUコアは自分に割り当てられた問題を2つに分割し，片方を終端したCPUコアに再び割り当てる．

探索中の解空間が最も広いCPUコアから分割する場合，部分問題割り当て回数は最大でも

$$(\text{CPUコア数}) \times \log(\text{解空間の広さ})$$

で押さえることが出来るため，それ自体は大きなコストにならないが，部分問題の要求があるか否かを常に監視する必要がある．

4 計算機実験

実験のための実装は3章での考察をもとに，補実験を行いつつ性能の良いプログラムを作るつもりで行った．

通信に関しては，暫定最適値の取得や部分問題要求の確認では排他制御を行わず，書き込みが決定した際に排他制御で同期を取るというような手法を取った．

4.1 実験環境

実験環境は以下の通りである．

CPU 型番 intel core2 Quad Q6700

個数 1

コア数 4 (2コア × 2)

動作周波数 2.66GHz (266 ×10)

FBS 周波数 1066MHz

キャッシュラインサイズ 64Byte

1次キャッシュ 16KByte × 4 (1コアに1つ)

2次キャッシュ 4MByte × 2 (2コアに1つ)

メインメモリ 4GB (DDR 1GB × 4)

OS fedora8 x86_64

kernel linux 2.6.23.14

言語 c 言語

コンパイラ gcc 4.1.2

マルチスレッドの実装 マルチスレッドのプログラムを実装するために，2種類の API を利用した．低レベルな関数群を使用した理由は，プログラムの動作構造を理解しやすくするためである．

POSIX Threads(pthread.h) [3] Linux でのスレッドを実装している API である．
スレッドの生成 (create)，同期 (join)，メモリ同期 (mutex) 等の機能が実装されている．

Scheduler(sched.h) [1] Linux でのマルチタスクのスケジューラーを操作するための API である．CPU コアにスレッドをバインドするために利用している．

4.2 テスト問題

分枝限定法の操作の様子を観測したいので，あまり簡単な問題にならないよう注意して問題を作成した．

$$
\begin{aligned}
n &= 10000 \\
b &= 750000 \\
a_j &= 100 + \mathrm{rand}(100) \quad , j = 1, 2, \ldots, n \\
c_j &= 100a_j - \mathrm{rand}(100) \quad , j = 1, 2, \ldots, n
\end{aligned}
$$

(rand(100) は $\{0, 1, 2, \ldots, 99\}$ の乱数)

このような方式で 10 題作った．また，解かせる際の分枝限定操作を簡略化するため最適解は 1 つだけ求まればよいものとした．

4.3 CPU コア数

作成した問題に対して，それぞれ使用する CPU コアの数を 1, 2, 3, 4 と変えて実験した．1 コアについては通信操作をダミーとして残したものと，通信操作をしないもの (n1 core) を別に用意し，時間の比較を明確にした．

4.4 実験結果

表 1 と表 2 に実験結果を示す．また，表の列は CPU コア数を表し，行は，

time 実行時間.

efficiency (通信無し 1 コアでの実行時間×CPU コア数/実行時間)，つまり CPU コアの利用効率.

iteration 分枝限定法の全スレッド合計反復回数.

request 部分問題の再割り当ての全スレッド合計数.

を表す.

4.5 実験結果のまとめ

通信による遅延

1 core と n1 core の場合を比べて分かるように共有メモリによる通信を入れただけで実行速度が 85%程に落ちてしまう．この割合は通信頻度の調整で変化させられるが，あまり頻度を下げると共有がうまくできず複数コアでの結果が改善されない．また，通信に関する補実験の結果，信頼できる通信のみで行った結果では更に実行速度が落ちてしまうことも判明した．

殆ど時間がかからなかった問題に関しては，問題分割やスレッド作成等の時間が大きな割合となったが，時間としては無視できる水準である．

CPU コア数による変化

通信時間を含めなければ，ほぼ CPU コア数分の実行速度の向上が観測された．時間が多くかかった問題では，反復回数の増加も殆ど無視できる程度であった．

5 おわりに

マルチコア環境での並列分枝限定アルゴリズムを実装し，計算機実験を実施して，マルチコア環境が有効に活用できることを示した．

今回の実験のように CPU コア数が少ない場合，いかに通信を避けるかがシングルコアに対する性能を決める条件となることが判明した．つまり，問題の割り当ての時点でどれだけ暫定最適値や部分問題を共有する必要性を減らせるかが重要であり，ヒューリスティクスの利用によりある程度良い暫定最適値を先に与える・値が決定されにくいことが予想される変数を使って部分問題の分割を行う等の工夫が有用である．

通信速度の問題を除けば他の環境と本質的な違いはさほどないため，研究において特にマルチコア・マルチプロセッサ環境にこだわる必要はない．ただし小規模であるがゆえにシングルコアと比較した CPU コア利用率の向上が大きな課題になり，その際ハードウェアに合わせた信頼の薄い通信を組み合わせることで大きく効率が変わる可能性がある．

表 1: 実験結果その 1

問題番号		n1 core	1 core	2 core	3 core	4 core
Q1	time	51m11.417s	60m16.173s	32m28.994s	20m0.013s	15m6.697s
	efficiency	1	0.85	0.79	0.85	0.85
	iteration	295,476,711,749		295,477,159,149	296,211,935,393	298,648,429,919
	request	-		4867	5029	5156
Q2	time	14m59.208s	17m40.377s	9m31.291s	6m8.926s	4m22.846s
	efficiency	1	0.85	0.79	0.81	0.86
	iteration	86,222,355,849		86,223,668,235	90,667,192,498	86,233,149,386
	request	-		4894	5029	5314
Q3	time	8.653s	10.154s	5.499s	3.802s	2.583s
	efficiency	1	0.85	0.79	0.76	0.84
	iteration	830,332,743		830,734,774	931,511,131	838,109,028
	request	-		4990	5016	5262
Q4	time	8.073s	9.372s	5.069s	3.130s	2.430s
	efficiency	1	0.86	0.79	0.86	0.83
	iteration	767,270,171		767,816,088	767,470,550	789,059,817
	request	-		4885	5070	5101
Q5	time	5.611s	6.626s	3.638s	2.432s	1.794s
	efficiency	1	0.85	0.78	0.77	0.79
	iteration	537,950,939		547,505,229	591,599,346	573,072,497
	request	-		5008	4998	5064
Q6	time	0.788s	0.927s	0.525s	0.344s	0.289s
	efficiency	1	0.85	0.75	0.76	0.68
	iteration	74,697,873		75,274,140	76,191,593	77,925,651
	request	-		4836	4939	5012
Q7	time	0.077s	0.093s	0.077s	0.074s	0.083s
	efficiency	1	0.83	0.50	0.35	0.23
	iteration	6,266,469		7,752,099	10,149,264	13,746,672
	request	-		4887	5028	5087
Q8	time	0.044s	0.047s	0.038s	0.048s	0.057s
	efficiency	1	0.94	0.59	0.31	0.20
	iteration	2,924,979		3,013,853	3,005,953	4,026,695
	request	-		5030	4537	4893

※問題番号は多くの時間を必要とした問題の順に並び替えた.

表 2: 実験結果その 2

問題番号		n1 core	1 core	2 core	3 core	4 core
Q9	time	0.036s	0.038s	0.036s	0.046s	0.051s
	efficiency	1	0.95	0.5	0.26	0.18
	iteration	2,383,357		2,674,732	3,214,122	3,072,854
	request	-		4980	4928	5044
Q10	time	0.008s	0.008s	0.008s	0.008s	0.011s
	efficiency	1	1	0.5	0.33	0.18
	iteration	81,765		150,904	220,453	81,770
	request	-		111	83	3

※問題番号は多くの時間を必要とした問題の順に並び替えた.

参 考 文 献

[1] GNU Operating System. CPU Affinity - The GNU C Library. (`http://www.gnu.org/software/libc/manual/html_node/CPU-Affinity.html`)

[2] 茨木俊秀. 組み合わせ最適化 分枝限定法を中心として. 産業図書, 1983.

[3] Lawrence Livermore National Laboratory. POSIX Threads Programming. (`https://computing.llnl.gov/tutorials/pthreads/`)

[4] Loots W. , Smith T.H.C. . A Parallel Algorithm for the 0-1 Knapsack Problem. *International Journal of Parallel Programming*, Vol. 21, No.5, 1992.

[5] Myong K. Yang, Chita R. Das. Evaluation of a Parallel Branch-and-Bound Algorithm on a Class of Multiprocessors. IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 5, NO.1, JANUARY 1994.

[6] 田辺隆人, 望月公晴, 逸見宣博. 並列分枝限定法による混合整数計画問題解法. 2001 年度日本オペレーションズ・リサーチ学会 秋期研究発表会, 2001.