ハイパフォーマンス
コンピューティング 77－28
（1999. 8. 4）

決定木の並列化とその評価

久保田 和人† 仲瀬 明彦†
酒井 浩† 小柳 滋†

{kazuto, nakase, h-sakai, oyanagi}@isl.rdc.toshiba.co.jp

新情報処理開発機構 並列応用東芝研究室†

概要

数百ギガから数テラバイトクラスのデータに対するデータマイニングを実用時間で行えるシステムの構築を検討している。その知見を得るために、データマイニングの代表的な手法である決定木を並列化して高速化し、百メガバイト程度のベンチマークデータおよび実データを用いて効果を調べた。決定木では、ルートから順にノードが生成されていく。1つのノード内の処理を並列化する手法(ノード内並列)と、複数のノードを並列に処理する手法(ノード間並列)を実装し、C4.5というフリーソフトをSMPマシンをターゲットとして並列化した。プロファイリングで処理のボトルネックを調べ、その部分のスレッドプログラミングを用いて並列化した。ノード内並列は、データの性質によらず8CPUで3倍から6倍程度の高速化が図れた。ノード間並列は、生成される木の偏りに大きく影響を受け、4倍程度高速化されたものから、全く高速化されないものもあった。

Parallelization of Decision Tree Algorithm and its Performance Evaluation

Kazuto Kubota†, Akihiko Nakase†, Hiroshi Sakai†
and Shigeru Oyanagi†

Parallel Application Toshiba Laboratory
Real World Computing Partnership†

Abstract

We are planning to develop a practical data-mining system to the data of form several 100giga byte to tera byte class. In order to obtain the knowledge for the construction of the system, the decision tree which is the typical technique of a data-mining is parallelized and accelerated. It applied to the data of a 100mega byte class, and was evaluated using benchmark data and real data. On the decision tree, nodes are generated from a root node to leaf nodes. The technique (intra-node parallel) of parallelizing processing in one node and the technique (inter-node parallel) of processing two or more nodes in parallel were implemented. A free software called C4.5 was parallelized for SMP machine. The bottleneck of processing was investigated by profiling and it was parallelized using thread programming. The effect of intra-node parallization was not affected by the characteristic of data, but was able to attain improvement in the speed of 3 to about 6 times by 8CPUs. Inter-node parallization received influence in the deviation of the tree generated greatly, and there was from what was accelerated about 4 times to what is not accelerated at all.

1. まえがき

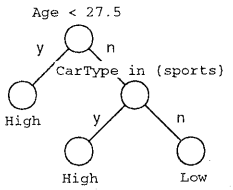
流通業やサービス業を始めとする多くの分野で、データマイニングと呼ばれる、計算機を用いて多量のデータから規則や特徴を抽出する手法が利用されている。例えば、金融機関では取引の減った顧客の分析を行うことで、その原因を探っている。また、デパート等の小売業では、家族構成や過去の買物の履歴をもとに、応答確率の高い顧客を見出し、その条件に合う顧客だけにダイレクトメールを送付することにより、コストの削減を図っている。データマイニングでは、本質的に大規模なデータを扱わなければならなず、ビジネス分野のハイパフォーマンスコンピューティングとして位置付けることができる。

本稿では、データマイニングの重要な手法である決定木について、並列化を試みた結果について報告する。並列化のベースとして用いた決定木生成アルゴリズム(プログラム)は、Quinlanらによって開発されたC4.5[1]と呼ばれるフリーソフトである。これは決定木生成プログラムとしては定評のあるものである。対象としたプラットフォームは、現在サーバー計算機のアーキテクチャの主流であるSymetric Multi Processor(SMP)とした。2種類の並列化手法を試み、4種類のテストデータに対してその効果を調べた。

本報告書は、以下の構成をとる。第2節では、決定木およびC4.5の紹介を行う。第3節では、並列化方針を2つ示

Age	Car Type	Risk
23	family	High
17	sports	High
43	sports	High
68	family	Low
32	truck	Low
20	family	High

(a)



(b)

図1　トレーニングセットと決定木 (文献 1) より)

し、SMP 上での具体的なインプリメント手法について述べる。第 4 節では、テストデータを用いて実験を行った結果について報告する。第 5 節では考察を行い、第 6 節では関連研究について述べる。第 7 節では、まとめと今後の展望について述べる。

2. 決定木生成プログラム C4.5

2.1 決定木とは

決定木は、データマイニングで用いられる classification アルゴリズムの一手法である。他の手法として、ニューラルネットを用いたものや、遺伝的アルゴリズムを用いたものがあるが、決定木は、これらの手法と比べて高速であり、同程度以上の質の解が得られることが知られている [1)2)]。

決定木では、トレーニングセットと呼ばれるデータ集合が与えられる (図 1(a))。1 つのデータ (レコードと呼ぶ) は、複数の属性と 1 つのクラスを持つ。属性がそのレコードを分類するための情報であり、クラスは分類先の情報である。属性は、カテゴリ値と呼ばれる離散値を取る場合 (図 1(a) の Car Type 属性) と連続値をとる場合 (図 1(a) の Age 属性) がある。決定木生成アルゴリズムにより、トレーニングセットから決定木 (図 1(b)) と呼ばれる分類器が生成される。これは、クラスの与えられていないテストセットと呼ばれるデータを、属性の値に応じて適切なクラスに分類するための木である。

木作成の方法は、トレーニングセットの使用方法で 2 種類に分類できる。1 つは、全データを用いて木を作成する方法であり、もう 1 つは、データーの一部を使った木の作成を繰り返す方法である。一般には、全部のデータを使って木を生成する方が良い木が得られ、場合によっては処理時間も短くなることが知られている [1)]。木の作成では、通常、木の大きさに制限を設けずに、トレーニングセットの情報をなるべく詳細に反映する木を作成し、その後で木の枝刈りを行なう。はじめから木の大きさを考慮して小さい木を作成する手法もあるが、通常、前者の方が良い木が得られる [1)]。木の生成に比べ、枝刈りの時間は無視できるので、木の生成の高速化が重要な課題となる。

2.2 C4.5 の木生成アルゴリズム

C4.5 は Quinlan らによって開発されたフリーソフトである。決定木生成プログラムとして定評のあるものである。決定木生成プログラムの他に、ルール生成プログラムが 1 つのパッケージとしてまとめられている。

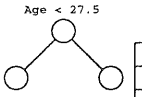
図 2 に C4.5 の木生成アルゴリズムを示す。これは分割統治法に基づくものである。ノード生成 (FormTree) では、トレーニングセットを分割した際の評価値が計算され (EvalAtt)、その中の最大の評価値をとる分割によってトレーニングセットが分割される (DivData)。分割されたデータは引数となり、再帰的に FormTree が呼ばれる。

```
FormTree( data )      /* ノード生成 */
{
    EvalAtt( data );    /* 評価値計算 */
    DivData( data );    /* データ分割 */

    for each sub data i
        FormTree( subdata[i] );
}
```

図 2　C4.5 における木生成アルゴリズムのフロー

Age	Car Type	Risk
23	family	High
17	sports	High
43	sports	High
68	family	Low
32	truck	Low
20	family	High



Age	Car Type	Risk
23	family	High
17	sports	High
20	family	High

Age	Car Type	Risk
43	sports	High
68	family	Low
32	truck	Low

図 3　分割後のトレーニングセット (文献 1) より)

図 1 を用いて具体例を示す。まず、図 1(b) のルートノードにおいて、どのような分割を選択すべきかが決定される。この場合、Car type と Age の 2 つの属性についてテストが行われる。それぞれの属性について分割した場合の評価値が計算される。分割の種類は、カテゴリー値をとる Car type の場合は、すべてのカテゴリー値に分割する場合の 1 通りである。本来 C4.5 は、グループに分割することもできるが、今回この部分は並列化の対象外とした。連続値をとる Age の場合は、2 分割が行われる。連続値の場合、分割の候補のとりうる数は、"異なるデータの数 − 1" 個となる。この例では、全てのレコードで *Age* の値が異なっているので、候補は 5 通りである。これらについて評価値を計算し、その中の最大値を取る分割方法が選択される。

評価値の計算方法としては、利得比基準 (gain ratio) という値が用いられる。これは、ある分割 X において、

$$gain\ ratio(X) = gain(X)/split\ info(X)$$

と定義される。ここで、

$$gain(X) = info(T) - info_X(T)$$

$$info_X(T) = \sum_{i=1}^{n} \frac{|T_i|}{|T|} \times info(T_i)$$

である。ここで T は分割前の集合であり、ある分割によって n 個の部分集合 $T_i (1 < i \leq n)$ 個に分割されるものとする。$info(S)$ はエントロピーと呼ばれる量であり

$$info(S) = -\sum_{j=1}^{k} \frac{freq(C_j, S)}{|S|} \times \log_2 \left(\frac{freq(C_j, S)}{|S|} \right)$$

で定義される。ここで、S は集合を示し、$freq(C_j, S)$ はクラス C_j に属する S 内の要素の数である。k はクラスの数である。また、$split\ info(X)$ は、

$$split\ info(X) = -\sum_{i=1}^{n} \frac{|T_i|}{|T|} \times \log_2\left(\frac{|T_i|}{|T|}\right)$$

と定義される。この他に、評価値の計算方法は、gini index[3]や、カイ自乗検定といった方法[4]が用られる。

分割方法が決定された後、その分割に基づいてデータは分割される。この例の場合、$Age < 27.5$ という分割方法が選択され、図3に示す2つのデータセットが生成される。

3. 並列化方針およびインプリメント

3.1 並列化の原理

決定木アルゴリズムを並列化するためには、ノード内の処理(FormTree)を並列化する方法とノード間にある並列性を活かす方法がある。各ノードで扱うデータ数が大きい場合は、ノード内の処理を並列化することで処理の高速化を図ることができる。ノード間の処理には親子間の依存関係しかないので、木がある程度バランスしていれば、ノード間にある並列性を活かすことで処理の高速化が図れる。ここでは、2つの並列化手法をSMP上にインプリメントする。

3.2 ノード内並列化のインプリメント

3.2.1 並列化方針

ここではデータ並列のアプローチを採ることにする。データ並列性には、レコード方向の並列性と属性方向の並列性がある。ここでは、その両方の並列化を試みる。

3.2.2 レコード並列

並列化方針

逐次版のC4.5の実行プロファイルを取り、その中で処理時間を要している部分の並列化を図ることにする。ここでは、SA、SB、F2、F7の4種類のデータを用いた。SA、SBは、スーパーマケットの売り上げ情報データであり、レコードサイズはそれぞれ、13万および33万程度である。SAは離散値の種類が大規模であり、SBは連続値の評価に処理を要するという特徴がある。F2、F7は人工的に作成されたデータであり、両者とも100万レコードである。9の属性とクラスからなり、全ての属性は連続値である。F2は木が偏って成長し、F7は平均して成長するという特徴がある。

4種類のデータを用いてプロファイリングした結果を表1に示す。使用計算機は、Sun Enterprise 5000である。OSは、Solaris 2.6で、使用したコンパイラは、Sparc Compiler 4.0である。-xpgオプションを用いてプロファイルを取った。各テストデータの全体の処理時間およびデータをディスクから入力してメモリ上にデータ構造を生成する時間を表1に示す。以下では、入力およびデータ構造生成時間を省いた処理時間に焦点を絞る。表2に処理時間の割合を示す。いずれの場合においても、Group、Quicksort、GreatestValueBelow、EvalContinuousAttという関数に処理時間を要していることがわかる。Groupは、特定の条件を満たすデータを抽出する関数であり、Quicksortは、ソート関数である。GreatestValueBelowは、ある条件下で最大値を求める関数であり、EvalContinuousAttは、連続値を持つ属性を分割した際の評価値を求める関数である。他の関数の処理時間の割合は、全体に比べて1%以下であり、

表1 逐次処理の場合の処理時間

	SA	SB	F2	F7
処理時間(秒)	3975.3	1054.1	2486.1	3006.1
入力時間(秒)	504.5	71.3	27.4	27.4

表2 各テストデータの処理時間の内訳

SA

関数名	割合(%)	実行回数
Group	60.13	89158
Quicksort	24.53	28234
GreatestValueBelow	5.54	1311
....	....	....

SB

関数名	割合(%)	実行回数
Quicksort	46.59	34100
GreatestValueBelow	27.74	1362
EvalContinuousAtt	13.41	34100
Swap	2.86	185522121
Worth	1.51	76673411
EvalDiscreteAtt	2.33	11184
Group	1.26	29260
....	....	....

F2

関数名	割合(%)	実行回数
Quicksort	53.16	5454
EvalContinuousAtt	15.92	5454
GreatestValueBelow	13.41	606
Swap	6.82	1127658591
Group	2.24	3030
ComputeGain	1.84	29021813
....	....	....

F7

関数名	割合(%)	実行回数
GreatestValueBelow	42.91	2942
Quicksort	33.81	26478
EvalContinuousAtt	10.50	26478
Swap	4.61	715151110
ComputeGain	1.62	24272743
Group	1.50	14710
....	....	....

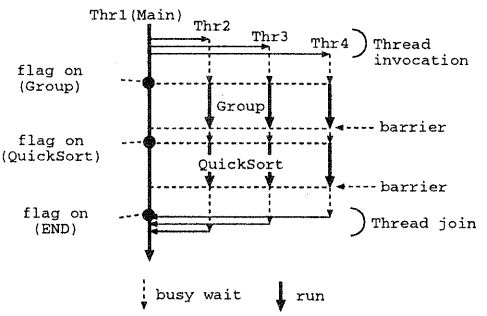


図4 スレッドによるノード内並列化(レコード並列)

関数コール1回あたりの処理時間も数μsec程度なので、ここでは、この4つの関数の並列化を行うことにする。

スレッドライブラリを用いた並列化

個々の関数内のループを並列化する方針をとる。スレッドライブラリには、Solaris Threadを用いた。各関数に処理が入る毎にスレッドを起動していてはオーバーヘッドが大きい。したがって、プロセッサ台数分のスレッドを起こしてビジーウェイトさせておく方法を取る。N台のプロセッサがあった場合、メインスレッド1つと$N-1$のサブスレッドを起動する。基本的にメインスレッドが処理を行い、必要に応じてサブスレッドと協調して処理を行う。処理のイメージは、図4のようになる。メインスレッドは、$N-1$のサブ

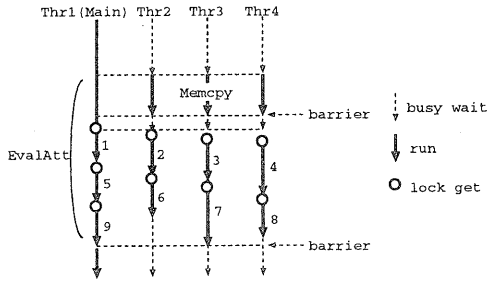


図5　スレッドによるノード内並列化 (属性並列)

スレッドを起動する (Thread invocation)。サブスレッドは、フラグ変数をポーリングしながらビジーウェイトに入る。メインスレッドの処理が進み並列化されている関数に入ると、メインスレッドはフラグ変数に値を書き込むことで、サブスレッドの処理を開始させる。最初は、関数 Group が起動される。起動される関数の種類は、フラグ変数に書き込む値により決定される。各スレッドで処理が終ると同期が取られ、メインスレッドは処理を進め、サブスレッドはビジーウェイトに入る。続いて、関数 QuickSort が起動される。このような処理を繰り返すことで処理が進められ、最終的に親スレッドがプログラムの終了地点に到達すると、フラグ変数に END が書き込まれ、この値を見てサブスレッドは処理を終了し、メインスレッドは、サブスレッドを join して処理が終了する。

並列化されている関数で、扱うデータ数が小さい場合は、逐次版の関数がメインスレッドによって実行される。切り替えのための閾値はユーザが与える。並列化された各関数の処理内では barrier がとられる。各関数の処理時間は 1 回あたり数 msec 程度なので、例えば、mutex lock を用いたような naive な barrier を用いては、オーバーヘッドが大きい。ここでは、高速な barrier として、fetch and add 命令による集中カウンタバリアを用いている [10]。このバリアは、8 プロセッサの場合数 μ ～数 10μsec で済むが、mutex lock を使うと数 msec もの時間を要してしまう。

3.2.3 属性並列

C4.5 のノード内処理における、分割方法決定のための評価値を求める処理は (図 2の EvalAtt)、属性毎に行われている。したがって、この部分の並列化を図る。

図 5に並列化の方法を示す。図は、4 スレッドで 9 属性の処理を行った場合である。図中の白丸は、ロックの確保を示し、数字は処理を行っている属性の番号である。レコード並列の場合と同様に、スレッドをプロセッサ台数だけ起動しておく。Thread 0 がメインの処理を担当し、並列化部分に入ると全スレッドが処理を行うようにする。並列化部分の処理に先立って、Thread 0 の持つデータの分割情報を他のスレッドにコピー (図 5の memcpy の部分) する必要がある。各スレッドは、排他制御を行いながら自分の担当する属性を決定し、該当する属性の評価値を計算する。排他制御には mutex lock を用いているが、ここでは barrier の場合と同様に、fetch and add 命令を用いた高速な mutex lock を用いている。

3.2.4 属性並列、レコード並列ハイブリッド

ノード内処理では、属性間の並列性が出せる処理は限られている。したがって、ノード内処理の第 3 のアプローチとして、属性並列とレコード並列を組み合わせることを考える。これは、属性間の並列性がある部分には属性並列を適用し、それ以外の部分には、レコード並列を適用するというものである。QuickSort, EvalContiniousAtt, EvalDiscreteAtt は属性並列が適用される。一方、Group, GreatestValueBelow にはレコード並列が適用される。スレッドを用いたインプリメントは、図 4と図 5 の方式を組み合わせたものとなる。

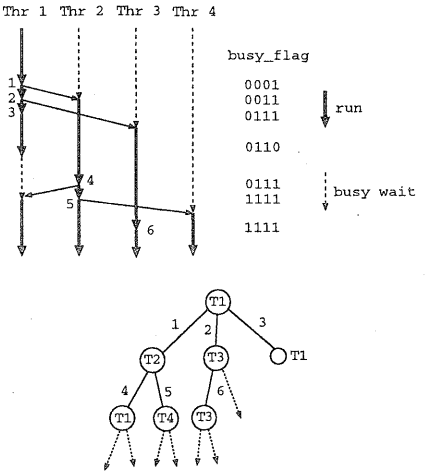


図6　スレッドによるノード間並列化

3.3 ノード間並列化のインプリメント

ここでは、基本的にノード生成処理毎にスレッドを割り付けるという方針をとる。

C4.5 では、FormTree という関数がノードを生成する関数であり、この関数が再帰的に呼ばれることで木が生成される。オリジナルのプログラムでは、各 FormTree 関数の実行結果がルートノード方向に集計されて最終的な木が構成される。並列化のために、呼び出した側の FormTree は、子供の FormTree の終了を待たずに終了できるように、FormTree およびデータ構造を変更する。

FormTree の呼び出し毎にスレッドを割り付けると、多量のスレッドが生成されスケジューリングのオーバヘッドが増大するので、ここでは、ノード内並列の場合と同様に、プロセッサ台数分のスレッドを起動し、フラグ (busy_flag) を用いてスケジューリングを管理する方法をとる。すなわち、無闇にスレッドを起動するのではなく、スレッドが空いている場合のみジョブを割り付け、スレッドが全てビジーの場合は、自分で処理を継続するというスケジューリング方法をとる。また、ジョブの大きさが小さい場合も自分で処理を継続することにする。この閾値はユーザが与える。

図 6に処理の流れを示す。各スレッドの処理フローのところに記された数字は、子供の FormTree の呼び出しを示している。右側の木の枝の番号と対応している。木のノードに示した T1～T4 は、そのノードをどのスレッドが処理するかを示している。実線はスレッドが処理を行っている部分、点線はビジーウェイトしている部分である。1 で Thread 2 に処理が渡され、2 で Thread 3 に処理が渡される。この時、busy_flag の該当ビットがセットされる。子供の FormTree に渡すデータ数が小さい場合は、自分で処理を行う (3 の場合)。Thread 1 は、3 の処理を終了した後に、busy_flag を

クリアしビジーウェイトに入る。この後、4と5で Thread 2が Thread 1と Thread 4を起動することで全ての Thread がビジーとなる。6で Thread 3は、FormTree を起動しようとするが、空きスレッドが無いため自分自身で処理を継続することになる。

4. 性能評価

4.1 実験環境

実験に使用したデータは、第3節で用いた SA,SB,F2,F7 である。それぞれのデータから生成される木の様子を図7に示す。SA は、(a) に示すようにルートノードが多数の葉を持つ木であり、SB は、(b) に示すようにバランスしていない木である。F2 は、(c) に示すように木が一方向に成長する特徴があり、(d) に示すように F7 はある程度分散して成長する特徴がある。

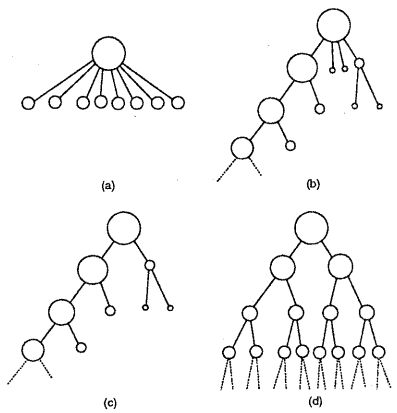


図7 SA, SB, F2, F7から生成される木の特徴。○の大きさは処理するデータの大きさを表す。(a)SA: ルートノードが多数の葉を持つ。(b)SB: 偏っている。(c)F2:偏っている。(b)F7:ある程度バランスしている。

合計4種類のテストデータに対して4種類の並列化手法を試みた。使用計算機は、Sun Enterprise 5000 (8 CPU) である。OS は Solaris 2.6 であり、コンパイラには Sparc Compiler 4.0 を用いた。コンパイルオプションは、`-x04 -xtarge=ultra` である。実行時間は、ファイル入力およびメモリ上での構造生成の後から木の生成が終了するまでを計測した。

4.2 実験結果

ノード内並列3種類 (intra-node parallel(record parallel, attribute parallel, hybrid))、ノード間並列 (inter-node parallel) で並列処理した場合のスレッド数と性能の関係を図8〜11に示す。SA(図8) では、レコード並列およびハイブリッドで8スレッドの時に6倍程度の高速化が図られている。これは、レコード並列による `Group` と `QuickSort` の並列化が、うまく機能したためと考えられる。属性並列で性能が上がらなかったのは、処理の負荷が特定の属性に集中していたためと考えられる。一方、ノード間並列で性能が上がらなかったのは、生成される木がルートノードで莫大なカテゴリ数を持つ属性での分割が選択され、そのまま葉となるものだったからである。ノード間並列では、ルートノードの処理は並列化されないという問題点がある。

SB(図9) の場合は、レコード並列、属性並列、ノード間

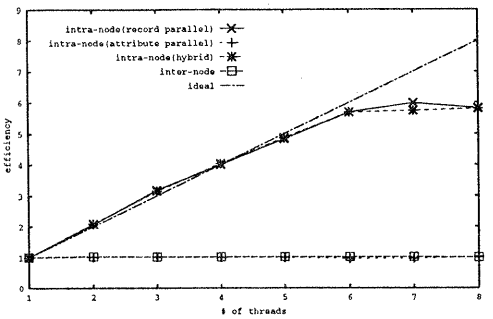


図8 SAにおけるスレッド数と処理効率の関係

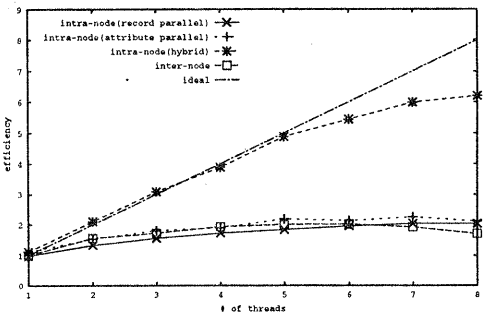


図9 SBにおけるスレッド数と処理効率の関係

並列のいずれの手法を用いても2倍程度の高速化にとどまっている。レコード並列で効率が上がらないのは、`QuickSort` に代表される連続値分割のための評価値の計算処理の並列化がうまくいっていないためである。属性並列で効率が上がらないのは、属性並列が効く部分が処理全体のうちの限られた部分だからである。ハイブリッドで8スレッドの時、6倍程度の高速化が図られているのは、レコード並列と属性並列が、互いに並列化できなかった部分の高速化を補ったためだと考えられる。一方、ノード間並列の性能が限られているのは、生成される木のルートからレベル4までの間で、著しく偏ったデータの分割が行われるためである。

F2(図10) は SB と同様な傾向が見られる。レコード並列に関しては8スレッドで1.7倍程度の高速化にとどまっている。これは SB の場合と同様な理由による。属性並列では8スレッドで2倍程度の高速化となっている。F2は9種類の属性しか持たないため、スレッド数が増えた場合に性能を出すのは難しい。このため SB と違い、ハイブリッド処理でも性能向上は、8スレッドの時3倍程度にとどまっている。ノード間並列で殆ど性能が上がっていないのは、生成される木が SB の場合よりもさらに偏ったものだからである。

F7(図11) では、ノード間並列が8スレッドの時3.7倍程度の高速化を達成している。これは、F7で生成される木の偏りが少なかったためである。レコード並列は8スレッドの時2.8倍、属性並列は1.5倍程度の高速化にとどまっている。レコード並列で性能が上がらないのは SB,F2 の場合と同様に `QuickSort` をはじめとする関数の並列化に限界があるためある。また、属性並列で効果が上がらないのは、並列化できない部分の処理が見えてしまうためである。ハイブリッドを用いると、性能向上は4.5倍程度になる。

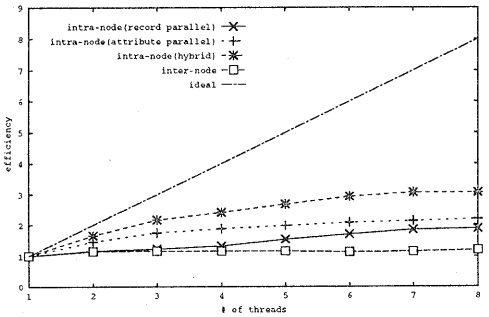


図10　F2におけるスレッド数と処理効率の関係

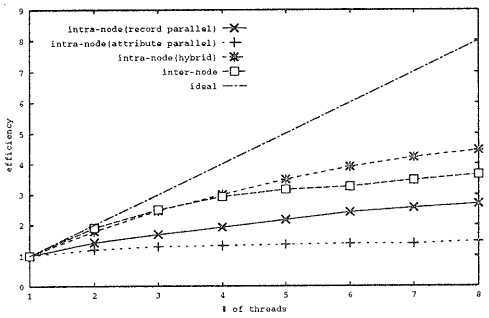


図11　F7におけるスレッド数と処理効率の関係

5. 考　　察

今回の実験結果から、並列化手法とデータの間に相性があることが明らかとなった。ノード間並列が有効なのは、木が分散して生成される場合のみである。この場合でも、ある程度までしか台数効果はあがらなかった。ノード間並列では、先に指摘したように一番粒度の大きいルートノードの処理が並列化できないという問題点がある。ノード内並列では、レコード並列と属性並列を試みたが、片方だけでは一部の問題を除いて、満足する性能をあげることはできなかった。今回の実験では、一部を属性並列、残りをレコード並列とする方法が一番良い性能を示した。しかしながら、ノード内並列だけではノードが細分化されると処理の粒度が小さくなるため、処理の効率化には限界がある。ノード内並列とノード間並列は、直交する並列化手法なので、この組み合わせを試みることは可能である。

6. 関連研究

逐次版の決定木アルゴリズムとしてCart[3]、およびC4.5の前身であるID3[5]がある。CartとC4.5は、それぞれIND-Cart、IND-C4へと改良されている[6]。また、Cartを並列化したものに、Darwinがある。Agrawalらは、前処理で一度だけソートを行いノード生成時にはソートが不要な決定木生成アルゴリズムSLIQ[7]、SPRINT[8]を発表している。SLIQ, SPRINTは、分散メモリ並列計算機をターゲットとして並列化されている。また、SPRINTをSMP向けに並列化した研究も行われている[9]。

7. まとめと今後の展望

本報告では、データマイニングで用いられる決定木アルゴリズムを、SMPをターゲットとして並列化した。C4.5というフリーソフトをベースとして4種類の並列化を行い、複数のテストデータを用いてその効果を定量的に調べた。以下に得られた知見をまとめる。

- 並列化手法とのデータの間に相性がある。
  - 木が偏らずに成長する場合にはノード間並列が有効。
  - 木が偏って成長する場合にはノード内並列が有効。
  - レコード並列と属性並列の組合せが現時点では最も有効。
- ノード内並列、ノード間並列のどちらか一方の並列化だけでは不十分で、両者を融合を検討する必要がある。
- 既存プログラム(アルゴリズム)をベースとした並列化には限界があるので、ノード内処理が軽く、かつスケーラビリティの高いアルゴリズムを考えて行く必要がある。

将来的には、数百ギカ～数テラバイトクラスのデータを実用的な処理時間で扱うことのできるシステムの構築を考えている。今回はプラットフォームとしたSMPを採用したが、メモリやdisk容量、コストパフォーマンスの面からPCクラスタも有望なプラットフォームの候補として考えている。

参考文献

1) J. Ross Quinlan, "C4.5: Programs for Machine Learning," Morgan Kaufman, 1993.
2) Rakech Agrawal, Tomass Imielinski, and Arun Swami, "Database mining: A performance perspective," IEEE Trans. on Knowledge and Data Engineering, 5(6):pp.914–925, Dec. 1993.
3) L.Breiman, J.H.Freidman, R.A.Olshen, and C.J.Stone," Classification and Regression Trees," Wadsworth, Belmont, 1984.
4) 福田 剛志, 森本 康彦, 徳山 豪, "多値属性を用いた最適なデータセグメンテーションを生成するアルゴリズム", 信学技報, DE98-22, pp.83–91, Oct. 1998.
5) J. Ross Quinlan, Induction of decision trees. Machine Learning, 1:pp.81–106, 1986.
6) NASA Ames Research Center, Introduction to IND Version 2.1, GA23-2475-02 edition, 1992.
7) Manish Mehta, Rakesh Agrawal, and Jorma Rissanen, "SLIQ: A fast scalable classifier for data mining," Proc. of the Fifth Int'l Conf. on Extending Database Technology, March, 1996.
8) John Shafer, Rakesh Agrawal, and Manish Mehta, "SPRINT: A Scalable Parallel Classifier for Data Mining," Proc. of the 22nd VLDB Conf. 1996.
9) Mohammed Javeed Zaki, Ching-Tien Ho, Rakesh Agrawal, "Scalable Parallel Classification for Data Mining on Shared-Memory Multiprocessors," IEEE ICDE, pp.198–205, March 1999.
10) http://www.rwcp.or.jp/lab/pdperf/barrier-collection