B2-6　　部分木ページングＢ木

小倉匠吾　　三浦孝夫

法政大学工学部 電気工学専攻

概 要

本研究では、B木の仮想化手法を提案する。この目的は、並列処理とデータの偏りの解決にある。B 木ページングを用いた分散化では、B 木を部分木に分け、物理的におのおのを独立に管理する。論理的な B 木と物理的な B 木の対応は、キー値を加工することによる。これにより物理的順序などの物理位置を任意に与えることができる。この結果、部分木単位の移動が可能となりデータの偏りを解消することができる。ここでは、部分木自体を B 木構造を用いて、ページの対応付けに関する性能評価を行い、その有用性を検証する。

1　前書き

本研究では分散 B 木で、どのようにデータを分割し、各計算機にどのように配置するかを論じる。並列環境上では、データを均等に分散することが全体の処理性能の向上につながるため、データの偏りを解決する問題は、広く処理の最適化ととらえることができる。

従来の研究ではデータによって分割する方法として、値域を指定して分割する方法とハッシュ関数を用いて分割する方法が知られている。しかし、値域を指定して分割する方法はデータの分布によって各計算機間でデータ数の偏りが生じる場合がある。ハッシュ関数を用いて分割する方法では、データ数の偏りは少なくなるが、領域指定の問い合わせが出来なくなってしまう。これらの欠点を解決する方法として、B 木のノード単位として分割する方法が考えられている。これはスプリットによって新たなノードが生まれたときに配置する計算機を決定する。決定方法としては、Rondom, Round Robin, Local Balancing[1] などの方法があるが、いずれもデータが削除された場合は保証していない。

また、各計算機に B 木のルートから葉までの部分木を配置する FatBtree[3] という方法も考えられている。これはルートに近いほど多くの計算機がコピーを持ち、葉に近いほどコピーを持たない。データの更新は大抵の場合、葉の近くが書き換えられることから更新処理が少なくなることが期待されるが、ルートの更新が起こった場合には多大な同期のオーバーヘッドが生じてしまう。

本研究では新たな分散 B 木構造として部分木ページング B 木を提案する。2 章では部分木ページング B 木の説明とその処理の流れを論じる。3 章では性能評価を行い、4 章で結びとする。

2　部分木ページングＢ木

部分木ページング B 木では、B 木を部分木単位に分割し並列処理を行う。この方法はデータの重複が起きないのでデータの同期の必要がなく、領域指定の問い合わせ

にも対応している。また、論理的な B 木を物理的な B 木によって仮想化することで、任意に物理位置を変換可能となり、データの偏りの問題に対応している。

任意の物理位置には、データ順序 (辞書式順序) 以外の物理配置も可能である。例えば、”トウキョウ*”の多い部分木と”カナガワ*”の多い部分木を近くに配置したり、”トウキョウ*”の多い部分木と”トクシマ*”の多い部分木を遠くに配置するなどの地理的順序の配置ができる。

2.1 構造

部分木ページング B 木では特定した深さの部分木に分割する。分割した部分木は各々独立して管理する。例えば部分木の深さを 2 とすると図 1 のようになる。

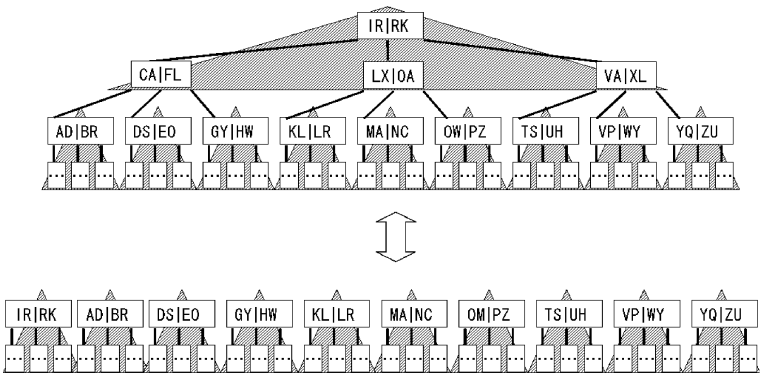

図 1: 部分木分散

部分木ページング B 木は部分木の物理キーと論理キーの対応をとることで、B 木の仮想化をしている。これにより物理位置を変更しても物理キーが変更されるだけで論理的な B 木構造は維持できる。キーの対応には図 2 のようなキー対応表を用意する。ここでは論理キーを部分木のルートのノードの先頭の値とし、物理キーをホスト名とホスト内位置の組合せとする。キー値対応表には Btree を利用し、頻繁に利用するためメモリ内で処理をする。データ数は部分木数であり十分メモリ内に収まることが期待できる。

論理キー	物理キー
AD	(HostA,1)
DS	(HostA,2)
GY	(HostD,2)
IR	(HostC,1)
KL	(HostB,1)
MA	(HostB,2)
OW	(HostA,3)
TS	(HostD,1)
VP	(HostC,2)
YQ	(HostB,3)

図 2: キー対応表

2.2 検索

各部分木の識別には論理キーを用いる。これが深い階層の部分木へのポインタの役割をする。ルートの部分木の論理キーはヘッダー情報として保持することで、ルートから全ての部分木にたどり着くことができる。

例えば図 3 で値”DZ”を検索するには、ルートの論理キー”IR”からキー対応表でルートの物理キーを取得しアクセスする。ルートの部分木で”DZ”を検索し、ノード (CA) の 2 つのポインタのうち 2 つ目のポインタの先に”DZ”があることを判断する。そのポインタとして格納されている”DS”からキー対応表で下層の部分木の物理キーを取得する。その部分木で”DZ”を検索し、値”DZ”を発見する。

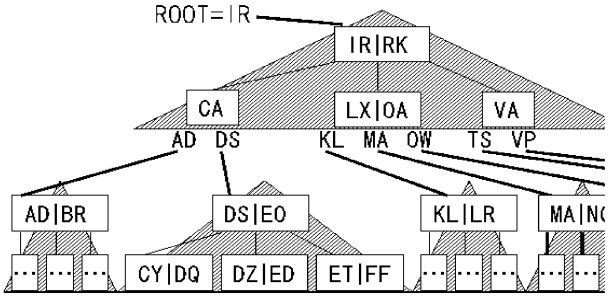

図 3: 部分木間移動

2.3 挿入

図 3 の状態に”EA”を挿入した場合の説明をする。この部分木ページング B 木は位数 1 とする。まず”EA”の挿入場所を検索する。挿入場所は上述の例で検索した”DZ”と隣の”ED”の間である。挿入を行うとこのノードはオーバーフローしてスプリットが起こり、”EA”が部分木のルートに挿入される。ルートでもオーバフローが起こりスプリットが起こる。部分木のルートがスプリットする場合は、部分木がスプリットを起こす。ルートのスプリットで新しく生まれたノード (EO) がルートとなる部分木が生まれる。そして”EA”が上層の部分木に送られる。上層の部分木では”EA”の挿入場所を探し、挿入する。その際に新しい部分木へのポインタ”EO”も挿入される。挿入後の部分ページング B 木の状態を図 4 に示す。

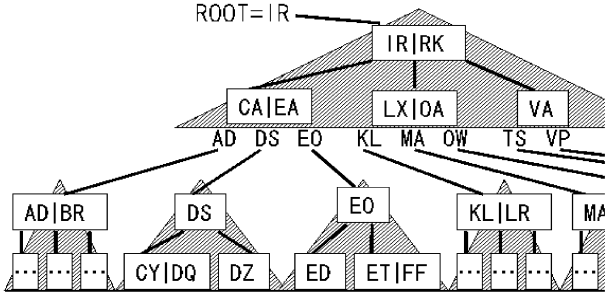

図 4: 部分木スプリット後

2.4 分散処理

並列処理のために、並列環境上の計算機の中の一台をサーバーとて動作させる。サーバーのみがキー対応表を持ち、各ホストに対し処理要求をする。並列環境はPVM(Parallel Vitual Machine)[2] などを利用することで実現できる。基本的には、サーバーがホストに対して部分木の物理キーと命令とデータ (命令,P-key,data) を送信する。それを受信したホストが物理キーの部分木に対して命令を処理し、処理の結果とデータ (結果,data) をサーバーに送信する。そして、サーバーは受信した結果に基づき次の処理を行う。

データの更新により新しく生成される部分木の配置について述べる。スプリットにより新しい部分木が生まれた時には元の部分木と同じに配置する。これは任意に部分木配置を変更できるので、データの転送の必要ない同じホストに配置するのが効率が良いからである。新たにルートの部分木が生まれた時には、サーバーに配置する。この方法は、まずサーバに全てを配置し、そしてあるタイミングで任意のホストに振り分けることになる。その後も任意のタイミングで配置変更をすることが可能である。以下では、挿入と検索の処理を説明する。

検索でのサーバーとホストの処理を図 5 に示す。基本的なサーバーの処理は、検索すべき部分木の論理キーからキー対応表により物理キーを求め、そのホストに対し検索命令を送信することである。各ホストでは検索命令を受け、部分木を検索する。部分木の検索で単純な B 木と異なる点は、部分木内にターゲットの値がなく下層の部分木がある場合は、ターゲットの値が格納されている可能性のある下層の部分木のポインタとして格納されている論理キーを返すことである。全体の処理の流れは、ルートの部分木から検索し、見つからなければ下層の部分木を検索する。これを葉の部分木まで繰り返す。

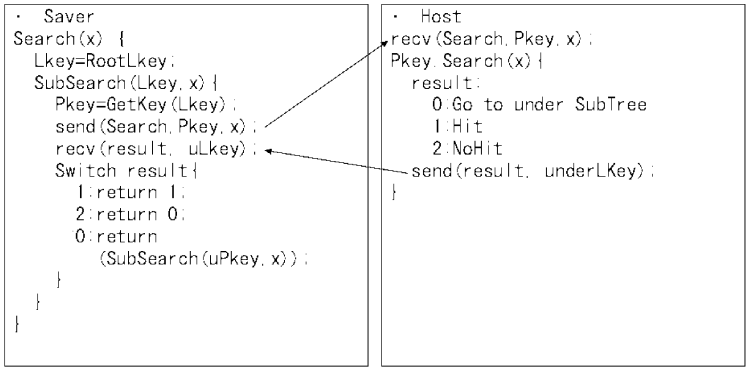

図 5: 検索

挿入でのサーバーとホストの処理を図 6 に示す。基本的なサーバーの処理は、検索と同様の処理で挿入場所を探し、挿入の結果によって行う処理が追加される。挿入によって論理キーが変更した場合 (図では f==1) には、キー対応表の更新や上層の部分木のポインタを書き換える処理をホストに要求する。部分木のスプリットが起きた場合 (図では i==1) には、新しい部分木をキー対応表に登録し、スプリットで上層に送られたデータをその部分木に対し挿入する。これは新しい部分木へのポインタも同時に挿入している。

ホストでの挿入の処理は 2 通りある。通常の処理は検索と同様に挿入場所を探し下層の部分木に進む、しかしスプリットにより下層から送られてきた値に対しては、下層の部分木に進まず挿入を行う。(図では f==1 の場合) 返す結果には、挿入により部分木の論理キーが変更された場合のフラグが含まれる。スプリットが起きた場合には、同ホスト内に新しい部分木を配置し、返す結果に上層に送る値と新しい部分木の

論理キーと物理キーが含まれる。挿入以外の処理として、下層の部分木の論理キーが変更された場合は、その部分木のポインタとして格納されている論理キーを更新する処理をする。

全体の処理の流れは、ルートの部分木から挿入場所を検索し、見つからなければ下層の部分木を検索する。これを葉の部分木まで繰り返す。挿入が終わるとその結果によって通過してきた処理しながらルートまで戻る。ルートの部分木の論理キーが変更した場合 (図では f==1) には、ヘッダー情報として保持しているルートの論理キーを更新する。ルートの部分木がスプリットした場合 (図では i==1) には、それまでのルートの論理キーと送られてきた値と新しい部分木の論理キーで、新しいルートの部分木をつくりサーバーに配置する。

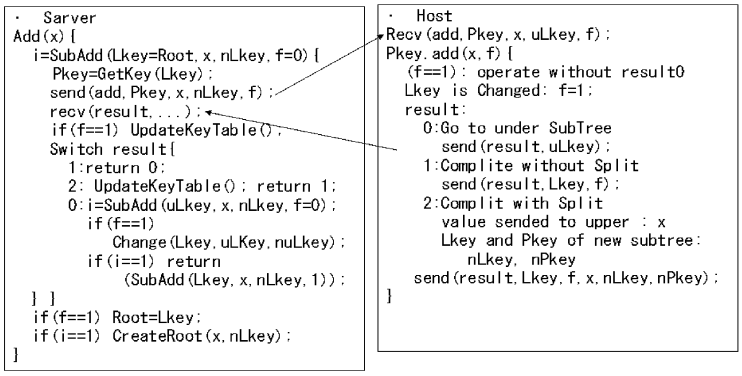

図 6: 挿入

3 実験

3.1 実験方法

部分木ページング B 木を評価するためには、分散による部分木間移動のコスト、仮想化によるページングのコスト、並列処理ための通信や同期のコスト、部分木の配置変更のコストを調べる必要がある。

並列処理と配置変更はその結果として性能が上がることが期待できるが、部分木間移動と仮想化は、それ自体で性能を上げることはなく並列処理や配置変更で性能を上げるための布石となる機能であり、それ自体で性能が上がるわけではない。しかし、並列処理するためには分散する必要があり、配置変更をするためには仮想化が必要である。これは、部分木間移動と仮想化のコストを調べることで、部分木ページング B 木の基本性能を調べることが可能ということである。したがって、本研究では、まず分散と仮想化のコストを調べる。

この章では部分木ページング B 木を並列処理なしで実装し、単一環境の B 木と比較して部分木間移動と仮想化のコストを推測し、その有用性を検証する。

3.2 実験手順

本研究では、OS:FreeBSD4.1,CPU:ペンティアムⅢ (1GHz),IDE Ultra ATA-100 の計算機上で実験する。位数:50,data:4B,pointer:4B の B 木を使用し、部分木ページ

ング B 木の部分木も同条件で深さは 2 までとする。また、キー対応表に用いているメモリ処理の B 木も同条件である。

乱数を用いてダブりのない 1,000,000 個の整数配列を用意する。その整数配列の 0 から n 番目を、初期状態の B 木と部分木ページング B 木にそれぞれ挿入する。次に、n 件の B 木と部分木ページング B 木に対して、整数配列の 0 から n 番目を検索する。挿入した整数を検索するので必ず発見できる検索である。n を 1,000 ～ 1,000,000 まで変化さて、挿入と検索の時間を計測する。

3.3 実験結果

検索と挿入の結果を図 7 と図 8 に示した。それぞれ B 木に対する部分木ページング B 木の処理時間の比率も同時に表記している。グラフ上の 2 本の縦のラインは木の深さの境目を表示している。深さは n=1,000 ～ 7,000 が深さ 2、n=8,000 ～ 400,000 が深さ 3、 n=500,000 ～ 1,000,000 が深さ 4 である。

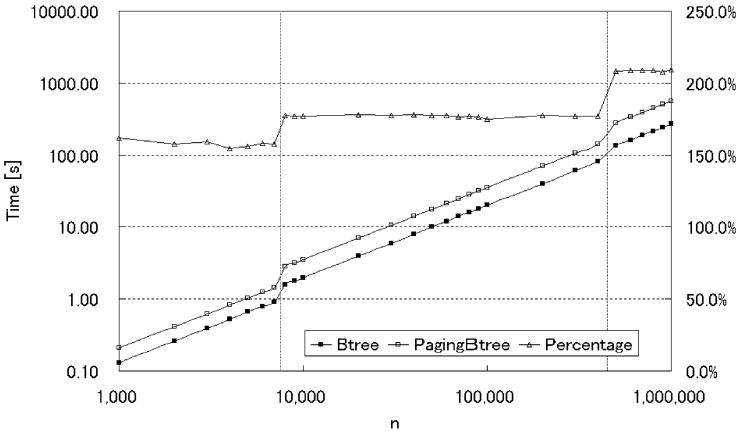

図 7: ランダム全件検索

検索の結果を見ると、比率は深さごとにほぼ一定で、深さが深くなるごとに比率が増えていることから、深さが増えるとコストが増えていることが分かる。深さごとの比率の平均を取ると、深さ 2 : 157.8%, 深さ 3 : 176.9%, 深さ 4 : 208.5%である。

挿入の結果を見ると、比率は深さ 2 の時は一定で、深さ 3 になると増加していき、深さ 4 になると減少している。部分木ページング B 木の部分木が 1 階層の間はコストが一定で、部分木が 2 階層になるとコストが増加していることが分かる。

3.4 考察

この実験結果には Unix のシステムキャッシュが作用していることが予想される。キー対応表と同様に B 木や各部分木もメモリ内で処理されていると思われる。これを仮定して結果を評価する。

B 木の深さはデータとして示した。この時の部分木ページング B 木の各部分木の深さを確認しておきたい。B 木の深さ 2 の時は深さ 2 の部分木が 1 階層で、深さ 3 の時は深さ 1 のルートの部分木と深さ 2 の部分木の 2 階層で、深さ 4 の時は深さ 2 の部分木が 2 階層である。システムキャッシュにより同等の負荷となってしまったキー対応表の負荷を考える。キー対応表のデータ数は部分木の数である。これは 2 階層のルートの部分木とほぼ同等の深さににになることが予想される。B 木の深さ 2 の時のキー対

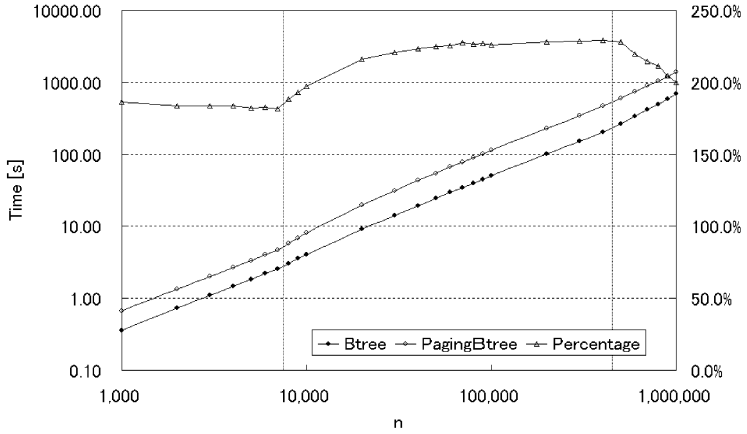

図 8: ランダム挿入

応表はデータ数 1 の深さ 1 である。これらを元に部分木分散 B 木のキー対応表を加えたルートから葉までの距離を考えると、B 木の深さ 2 の時は深さ 1 のキー対応表を 1 度参照しているので深さ 3 で、B 木の深さ 3 の時は深さ 1 のキー対応表を 2 度参照しているので深さ 5 となり、B 木の深さ 4 の時は深さ 2 のキー対応表を 2 度参照しているので深さ 8 となる。

これから、予測される検索の比率は、深さ 2 の時に 150%で、深さ 3 の時は 166%となり、深さ 4 の時には 200%となる。これを実験結果と比べると、各深さとも 5%程の増加となっていることが分かる。これはキー対応表以外の部分木間移動と仮想化のコストということが言える。問題のキー対応表のコストは B 木や部分木がディスク処理なのに対しメモリ処理なので、わずかなコスト増加に留まることが推測される。部分木間移動と仮想化の機能を備えた部分木ページング B 木は B 木に比べて遜色ない検索性能であると言えるであろう。

挿入の場合、部分木間移動以外に部分木の論理キーが変更された場合もキー対応表にアクセスがかかり、木の深さは変化していくので細かな予測を立てるのが難しい。また、部分木のスプリットも起こる。これはディスク処理なので、実際のコストになる。しかし、今回の結果からは、正確な推測は出来ないが、部分木のスプリットの起きない部分木が 1 階層の比率の増加が 80%超なことから、部分木が 2 階層での部分木のスプリットによる比率の増加は 50%を超えることはないことが予測される。

B 木に比べて挿入によるデータの更新の性能は若干劣るものの、並列処理においては、データの重複を持たないためデータの同期のコストが増えることはなく、データの同期が必要な方法に比べて非常に有利であることが期待できる。

4 結び

本稿では仮想化により任意に配置が可能な分散 B 木構造として部分木ページング B 木を実現した。実験によりその有用性を検証し、分散処理などの実験を続ける意義があることが分かった。

今後の課題としては、部分木へのダイレクトアクセス手法, キャッシュの活用, 並列操作などを考えている。

参考文献

[1] B.Seeger and P.Larson "Multi-Disk B-tree" SIGMOD Conference 1991:436-445

[2] PVM (Parallel Virtual Machine) http://www.epm.ornl.gov/pvm

[3] H.Yokota, Y.Kanemasa, and J.Miyazaki, "Fat-Btree: An Update-Conscious Parallel Directory Structure" ICDE 1999.3:pp.448-457