Processing-in-Memory 上の探索木に対するバッチクエリの負荷分散に向けて

奥田 光　鵜川 始陽

Processing-in-Memory(PIM) は，DRAM の各メモリチップに計算機能を加えた PIM ノードで構成されるアーキテクチャである．PIM には，各ノードが自身のメモリにだけ高速にアクセスできるという分散メモリ計算機の側面があるため，通信の削減と負荷分散が重要である．本研究では探索木について負荷分散を行う方法を検討する．探索木を PIM ノードの数よりも多くの部分木に分割し，各 PIM ノードが処理するクエリの量が均等になるように配置する．また，クエリが集中する部分木が変化したときに，部分木を移動することで，クエリの量を均等にする．実在する PIM アーキテクチャ上で部分木を最適に配置した場合，CPU で計算した場合の 2.3 倍のスループットが得られた．部分木を再配置する際の通信速度は 6.4MB/s だった．

1　はじめに

探索木は，階層構造や順序関係のあるデータを表現でき，データベースや機械学習，ファイルシステムなどの様々なアプリケーションに用いられる [14]．近年，扱われるデータ量の増大および CPU の性能の進歩により，探索木を辿る際のメモリアクセスがアプリケーションの重大なボトルネックとなっている [7]．現在広く普及しているノイマン型アーキテクチャで主記憶に木構造を置いた場合，データ量がキャッシュ容量を超えるとキャッシュミスが発生し，主記憶へのアクセスがボトルネックとなる [2] [4]．

このデータ移動のボトルネックを軽減するために，Processing-in-Memory(PIM) アーキテクチャを用いるアプローチが存在する [11]．PIM アーキテクチャは DRAM の各メモリチップに計算機能を加えたアーキテクチャであり，計算をデータの存在するチップで行うことでデータ移動にかかる時間を短縮できる．このメモリチップを以降 **PIM ノード**と呼ぶ．さらに，PIM ノードはメモリチップの数だけ存在するため，PIM アーキテクチャは高い並列性を持つ．

本研究では，PIM アーキテクチャを対象として探索木に対するクエリを実行するアルゴリズムを開発する．PIM アーキテクチャの並列処理性能を利用し高いスループットを得るために，一定の数のクエリをまとめたものを**バッチ**とし，バッチ内のクエリを並列に実行する．

PIM アーキテクチャにおいて探索木を実装する際には，CPU と PIM ノードそれぞれにどのような計算を行わせるかが課題となる．PIM ノードは多数あるが，一つあたりの計算能力とメモリは限られている．一方で，CPU は全ての PIM ノードと繋がっているのに加え，PIM ノード一つと比べて強力な計算能力を持つ．それぞれの特性を生かすためには，探索木をどのように分割し，メモリ上に配置するかが重要となる．

我々は，探索木を部分木に分割し，それぞれを PIM ノードまたは CPU と対応させるアプローチを採用する．木の探索は根から葉に向かって行われるため，部分木に分割することで PIM ノードと CPU との間の通信を減らしながら木を探索することが出来る．

部分木への分割として素直な方法は，キーの範囲による均等分割を行うものである．すなわち，キーの範囲全体を PIM ノードの数と同数に均等分割し，それ

Towards Load Balancing for Batch Queries on Search Trees in Processing-in-Memory.

Hikaru Okuda, Tomoharu Ugawa, 東京大学情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.

ぞれの範囲に対応する部分木に分割する．そして，部分木と PIM ノードを 1 対 1 で対応させることでデータの配置を行う．この方法により，均一なワークロードに対しては CPU を上回る性能が得られる [17]．しかし，ワークロードに偏りがある場合，特定の部分木へクエリが集中すると対応する PIM ノードに負荷が偏り，性能が大きく低下してしまう．

そこで，本研究では探索木を PIM ノードの数よりも多くの部分木に分割し，各部分木へのクエリ数の分布に応じて負荷分散が達成出来るように木を配置する手法を提案する．具体的には，各バッチの処理において，CPU が各部分木へのクエリの数を調べ，それに応じて負荷分散できるよう部分木を移動する．その際，クエリの多い順に部分木を 3 グループに分け，それぞれ以下のように扱う．

1. CPU に配置する．
2. 一つの PIM ノードを占有させる．
3. 一つの PIM ノードに複数配置する．

部分木を移動したのち，各 PIM ノードおよび CPU で実際にクエリの処理を行う．

我々は，B^+ 木 [3] を基にしたアルゴリズムを開発し，実在 PIM アーキテクチャである UPMEM 上に実装した．ただし，部分木の移動は今後の課題である．この実装を用いて，偏りはあるが，偏りが時間的に変化しないワークロードに対して，最適な部分木の配置を求め，その場合のスループットを測定した．また，偏りの変化に対応して木を移動するオーバヘッドを見積った．

2 Processing-in-Memory(PIM) アーキテクチャ

図 1 に PIM アーキテクチャと従来のノイマン型アーキテクチャの構成の違いを示す．PIM アーキテクチャは従来のアーキテクチャが持つ CPU と DRAM に加え，計算機能を持った小さなメモリチップである PIM ノードを多数持つ．データのある場所と計算する場所を近づけることにより，近年深刻化しているデータ移動のボトルネックを軽減することが期待されている [11]．

PIM アーキテクチャの実装の一つとして，UPMEM [13] がある．図 2 に UPMEM の構成を示す．UPMEM は 2560 個の DRAM Processing Unit(DPU) と呼ばれる PIM ノードを持つ．各 DPU は 450MHz で動作するインオーダ 32 ビット RISC コアであり，24 個のハードウェアスレッドによる並列実行が可能である．また，DPU 一つあたり 64MB，システム全体では 160GB の DRAM バンク (MRAM) が DPU からもアクセスできる．DPU と CPU の間のデータの転送は MRAM と CPU のキャッシュとの間で行われる．一方で DPU 間の通信はできないため，CPU を介して行う必要がある．

3 PIM アーキテクチャにおける探索木の実装

3.1 本研究で開発する木構造

本研究では，B^+ 木 [3] を基に，PIM アーキテクチャに向けた高速な木構造の実現を目指す．この木構造はキー k による検索，キー k と値 v のペアの挿入および，キーの範囲 $[k_1, k_2]$ の値を全て得る範囲検索の 3 種類のクエリをサポートする．ただし，範囲検索は本論文では扱わない．一定数のクエリをバッチとしてまとめ，それらを一度に実行するバッチ処理を行う．バッチ処理はレイテンシがあまり重要視されず，高いスループットが求められるアプリケーションに広く用いられている [5]．

3.2 部分木への分割

PIM アーキテクチャに探索木を実装する際，探索木を部分木に分割することでボトルネックとなるデータ移動を減らすことができる．図 3 に従来のアーキテクチャと PIM アーキテクチャにおける探索木の探索の違いを示す．従来のアーキテクチャでは，探索木の全体がキャッシュに収まらない場合，典型的には頻繁にアクセスされる探索木の上部がキャッシュされており，下部は DRAM にのみ存在することになる．例えば，図 3(a) のような探索木の場合，ノード 3，2，1 にアクセスする度に DRAM アクセスが発生し，多くの時間が消費される．一方で PIM アーキテクチャの場合，図 3(b) のようにキャッシュに乗り切らない木の下部を分割して PIM ノードに配置する．これにより，PIM ノードとの通信 1 回とその中での局所的

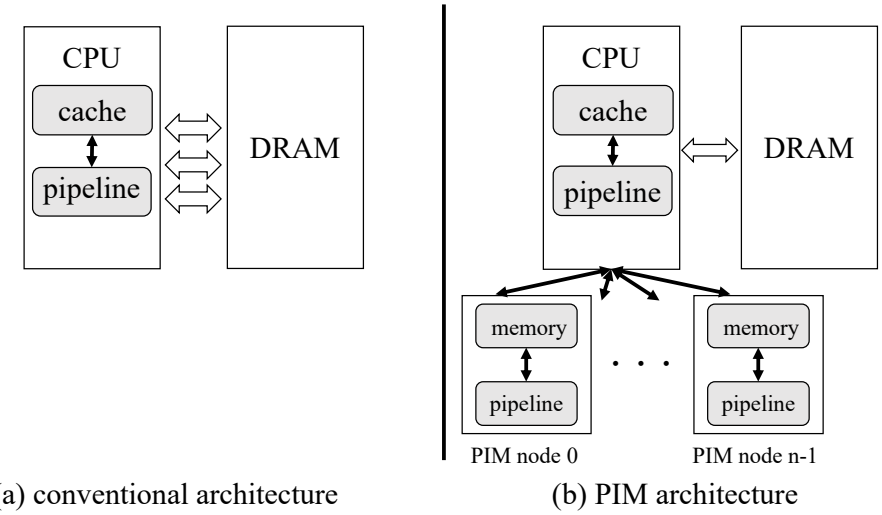

図 1　アーキテクチャの構成 ((a) 従来のアーキテクチャ (b)PIM アーキテクチャ)．矢印はデータの移動を表す．

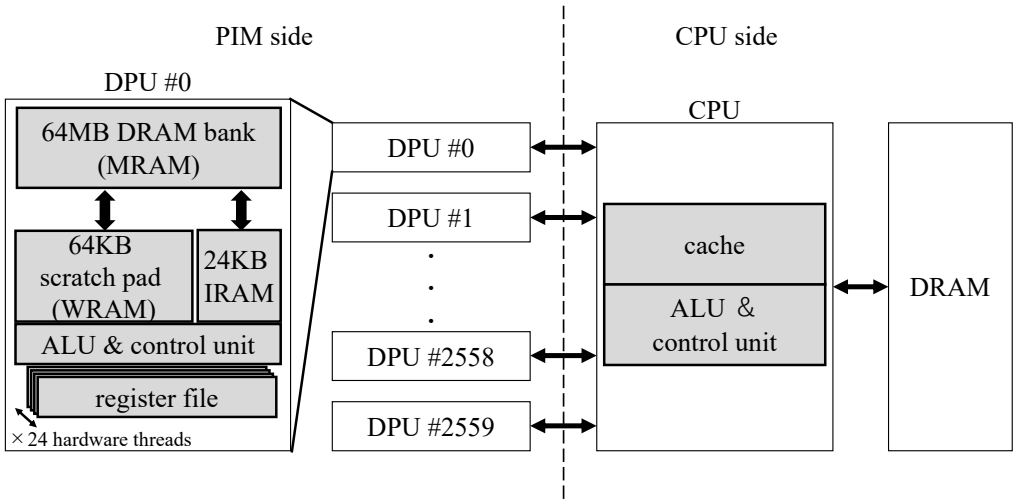

図 2　UPMEM の構成．矢印はデータの移動を表す．

なメモリアクセス 2 回に抑えることが出来る．

さらに，PIM アーキテクチャのもう一つの利点である高い並列性を生かすため，並列アルゴリズムの設計で広く用いられる [1] [12] バッチ並列処理を行う．本研究におけるバッチ並列処理は以下の 3 つの段階で構成される．

1. CPU がバッチを各 PIM ノードが行うべきクエリへと仕分けし，各 PIM ノードにクエリを一斉に送信する．
2. 各 PIM ノードが並列にクエリの実行を行う．また，CPU は自身の担当するクエリを実行する．
3. CPU が PIM ノードから結果を一斉に受け取る．

段階 2 における各 PIM ノードのクエリ数を均等にし，CPU にも適切なクエリを実行させることで，クエリの実行時間を短くすることができる．そして，各 PIM ノードおよび CPU が実行するクエリは段階 1 の時点で各 PIM ノードおよび CPU が持つデータの範囲から決まる．したがって，PIM アーキテクチャ

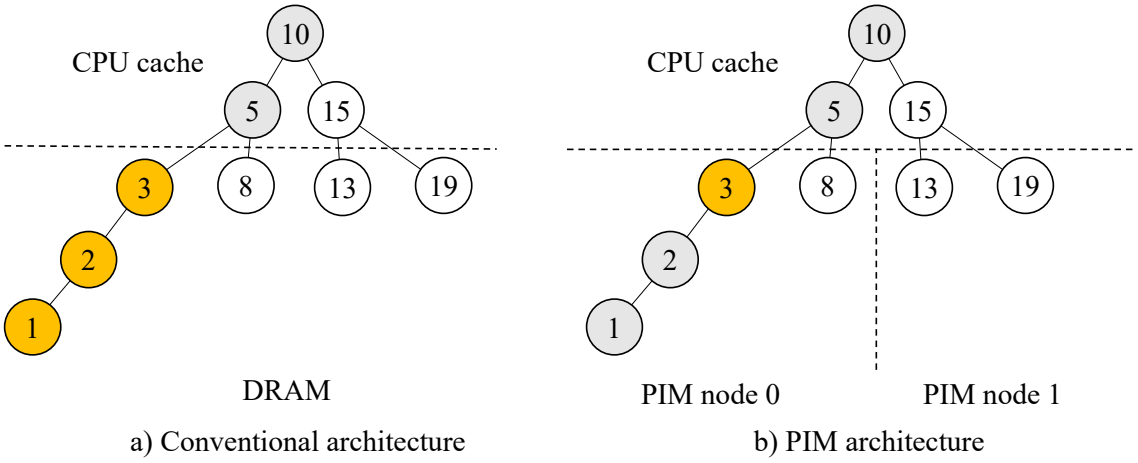

図 3　探索木の探索のアーキテクチャによる違い ((a) 従来のアーキテクチャ (b) PIM アーキテクチャ)．ノード 1 を探索する際，橙色のノードへのアクセスが DRAM アクセスまたは PIM ノードとの通信を生じる．

において負荷分散を達成しより高いスループットを得るためには，クエリの偏りを考慮してどのようにデータを配置するのがが重要となる．

3.3　キーの範囲による木の均等分割

クエリの偏りを考慮しなければ，キーがとりうる範囲を PIM ノードの数で等間隔に分割し，それぞれの範囲に対応する部分木を PIM ノードに配置するのが単純である．この分割と配置の方法を**キーの範囲による均等分割**と呼ぶことにする．図 3 b) はキーの範囲が [0, 20]，PIM ノードが 2 つの場合のキーの範囲による均等分割の例である．木の要素もクエリもキーの範囲全体に一様に分布していれば，この分割方法で負荷分散できる．しかし，実際の応用ではこれらには偏りがあることが多い．

予備実験として，クエリに偏りがある場合に，キーの範囲による均等分割では性能が悪化することを調べた [17]．以降ではその詳細と結果を示す．

3.3.1　分割の方法

UPMEM において以下のように部分木への分割を実装した．キーの範囲全体を (DPU の数)×(各 DPU のソフトウェアスレッド数) 等分する．各ソフトウェアスレッドはそのうち一つのキーの範囲を持った B^+ 木を保持する．各 DPU のソフトウェアスレッド数は 1 から 11 まで変化させる．したがって，システム全体に最大で 2560×11 個の B^+ 木が存在することになる．

3.2 節で述べたバッチ並列処理の流れに従い，CPU はクエリのキーに対応する B^+ 木が存在する DPU にクエリを送るルーターの役割を果たす．本予備実験では，CPU はクエリを各 DPU に送る役割のみを果たし，クエリは実行しない．

図 4 はソフトウェアスレッドが 3 個の例を示している．キー空間全体が 2560×3 等分され，各ソフトウェアスレッドと対応付けられている．

3.3.2　実験評価

32 ビット整数から一様ランダムに選ばれたキーを用いたワークロードおよび，zipf 分布 [16] に従って偏ったキーを用いたワークロードの 2 種類を用いた．操作の割合はいずれも挿入クエリ : 検索クエリ = 1 : 1 とした．36 コア，72 スレッドの CPU においても B^+ 木を実装し，比較を行った．UPMEM の CPU および DPU の性能，比較対象である CPU の性能は 5 章で用いたものと同様である．

表 1 に UPMEM での実験結果を示す．一様なワークロードでは CPU のおよそ 5 倍の性能になることが確認できた．一方で偏ったワークロードにおける性能は CPU の 1/1650 倍となっており，著しく低い．

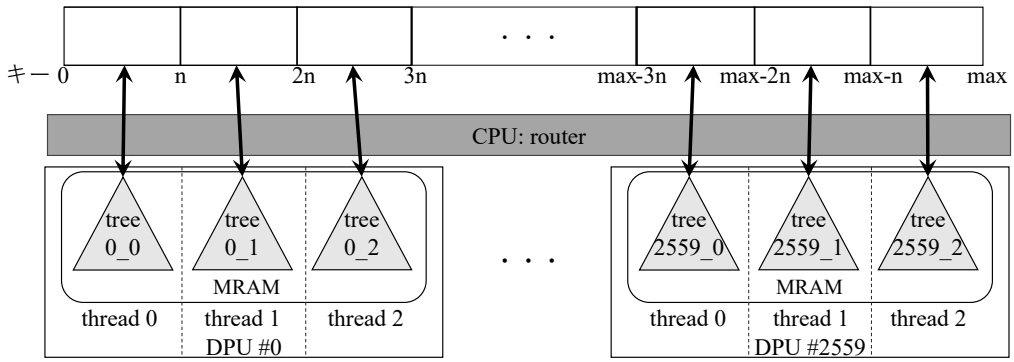

図 4　キーの範囲による木の均等分割．n はキーの最大値を部分木の数で割ったものである．

表 1　キーの範囲による木の均等分割の実験結果 (MOPS/s)．

一様ランダムなワークロード		zipf 分布により偏ったワークロード	
CPU	UPMEM	CPU	UPMEM
62	290	66	0.04

4　負荷分散のための木の分割

4.1　負荷に応じた部分木の配置

提案手法では，以下のようにして負荷に応じた部分木の配置を行う．まず，部分木への分割数を PIM ノードの数よりも大きくする．そして，部分木を PIM ノードに等しく分配するのではなく，負荷の偏りに応じて適切な場所に配置する．具体的には，クエリの集中する部分木は CPU や専用の PIM ノードに配置し．クエリが少ない部分木は一つの PIM ノードに複数配置する．

図 5 に UPMEM における提案手法の概念図を示す．図 5 右上のグラフは各部分木へのクエリの数を模式的に表したものである．グラフ横軸の部分木番号と図 5 の各部分木内の数字が対応している．特にクエリの集中する部分木は CPU で探索を行う (図 5 の部分木 5, 6)．これは，少数の部分木であれば CPU のキャッシュに収まり，また，CPU の方が DPU よりも高速に動作するためである．残りの部分木について，PIM ノード間で分配する．UPMEM の場合，PIM ノードである DPU は複数のハードウェアスレッドを持つため，クエリの集中した部分木はマルチスレッドで並列に探索を行う (図 5 の部分木 0)．一方でクエリが少ない部分木については，一つのスレッドが一つまたは複数の部分木の探索を行う (図 5 の部分木 2, 4, 1, 3, 7, 8)．

4.2　木の移動

クエリの偏りが時間的に変化する場合，クエリの偏りの変化に合わせて部分木を移動させる．現在この機能は検討中であり，未実装である．

CPU は，各部分木の持つキーの範囲を記録しているため，バッチ内において各部分木へのクエリの数を知ることができる．その情報を利用し，バッチの前処理段階において 4.1 節の配置に近づけるよう DPU および CPU の間で部分木を移動させる．この部分木の移動には無視できないオーバーヘッドが生じると予想されるため，5 章にてその影響を見積もる．また，並列に効率良く目的の配置になるよう木を移動させるアルゴリズムは現在検討中である．

5　実験

本章では，以下の二つの疑問に答える実験を行った．

1. クエリの偏りが時間的に変化しない場合，クエリの偏りに対して適切な部分木の配置をとることで，CPU の性能を上回ることが出来るのか．ま

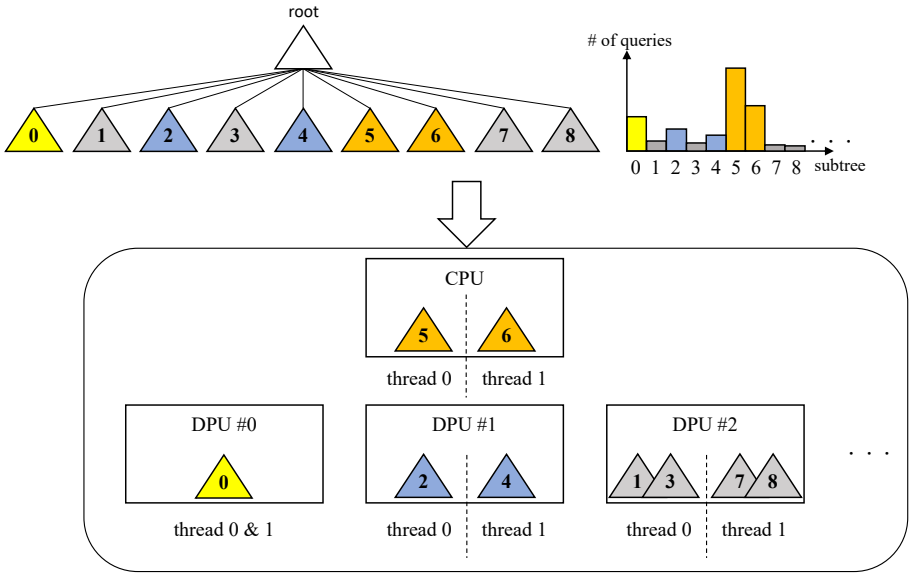

図 5　負荷に応じた部分木の配置．三角形は部分木を表している．グラフは部分木ごとのクエリの数を模式的に表しており，色は負荷の偏りに対応している．

た適切な配置とはどのような配置か．

2. 部分木の移動のオーバーヘッドはどれくらいか．オーバーヘッドを考慮すると，偏りが時間的に変化する場合にどれくらいの部分木の移動が許容されるか．

実験に用いた UPMEM の構成は以下の通りである．

- CPU: Intel Xeon Silver 4215 (2.50GHz)
  - 8 コア，16 スレッド
  - LLC: 11MB
- DRAM: DDR4-2666 256GB
- PIM enabled memory: DDR4-2400 DIMM ×20
  - DPU(450 MHz) ×2560
  - DRAM バンク 64MB ×2560 = 160GB

比較対象の CPU および DRAM は以下の通りである．

- CPU: Intel Xeon Gold 6354 (3.00GHz) ×2
  - 計 36 コア，72 スレッド
  - LLC 78 MiB
- DRAM: DDR4-3200 192GB

5.1　部分木の配置がスループットに及ぼす影響

一つ目の疑問について，負荷の偏りが時間的に変化しないベンチマークを用い，様々な固定の部分木の配置に対してスループットの測定を行った．

5.1.1　実験方法

実装において，DPU をあらかじめ以下の 2 種類に分けた．

- **first-class DPU**: 部分木を一つだけ持ち，複数のスレッドが並列に探索を行う DPU
- **economy-class DPU**: 部分木をスレッド数よりも沢山持ち，1 スレッドが複数の木を探索する DPU

economy-class DPU が持つ部分木の数は 20，ソフトウェアスレッド数は 10 とした．

まず，最もクエリの集中した 16 から 512 個の部分木を CPU に配置した．その次にクエリの集中した 1 から 1536 個の部分木を first-class DPU に，残りの部分木を economy-class DPU に配置した．配置に関するパラメータは CPU に配置する部分木の数，および DPU 全体のうち first-class DPU の占める割合の二つである．このとき部分木の総数は，(CPU における部分木の数)+(first-class DPU の数)+(economy-class DPU の数) × 20 となる．

また，ワークロードに関しては YCSB [6] で発行されるのと同等なものを生成した．具体的には，YCSB

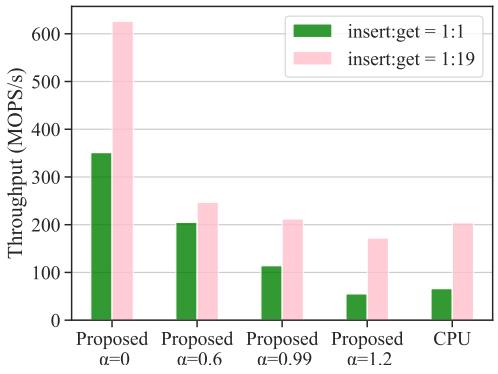

図 6　それぞれのワークロードに対する最大のスループット．

で用いられている zipfian 分布 [16] を使用した．しかし，単純にキーに対して zipfian 分布を適用すると，偏り度合いを表す zipfian 定数 α を大きくしたとき，特定のキーが何度も繰り返し出現する．このとき CPU 側でこのキーに対するクエリを一つにまとめることが簡単に出来てしまう．例えば，zipfian 定数を YCSB のデフォルトである 0.99 に対して 1.2 に設定し，キー空間を 64 ビット整数全体とした場合，最も出現しやすいキーだけで全体の約 18%を占める．したがって後述する関連研究 [9] と同様に，より粗い偏りを得るためキー空間を 2048 個に等分した．そしてクエリごとに zipfian 分布に従って 2048 個の範囲から一つを選び，その範囲内で一様ランダムにキーを選んだ．

ワークロードは検索クエリ，挿入クエリからなり，挿入クエリ : 検索クエリ = 1 : 1 の書き込みメインのワークロード，挿入クエリ : 検索クエリ = 1 : 19 の読み込みメインのワークロードの 2 種類を用いた．キーと値には，64 ビット符号なし整数を用いた．バッチサイズは，100 万クエリとした．

5.1.2　実験結果

図 6 にそれぞれのワークロードに対して得られた最大のスループットを示す．また，CPU で $\alpha = 0$ のワークロードを実行した結果を図 6 の CPU に示す．偏りが少ないワークロードでは 3.3 節の実験結果と同様に高いスループットを維持しつつも，偏ったワークロードに対してもほとんどの場合において CPU を上回る結果が得られた．ただし，前述のとおりかなり偏ったワークロードである zipfian 定数 1.2 の場合のみ，CPU のスループットがわずかに上回った．

図 7，図 8 に部分木の配置とスループットの関係を示す．図 7 が書き込みメインのワークロード，図 8 が読み込みメインのワークロードに対する結果である．3 枚のグラフは左から順に zipfian 定数 0, 0.6, 1.2 に対応している．横軸が部分木のうち CPU に置くものの割合，縦軸が DPU 全体のうち first-class DPU の占める割合であり，それぞれの点に対してスループットがプロットされている．色が濃いほど，スループットが高いことを表している．グラフの原点に近いほど 3.3 節のキーの範囲による均等分割に近づいていく．

ワークロードに偏りがないときは原点に近づくにつれて性能が上がっており，3.3 節の手法が最適であることが確認できる．

一方で偏りがあるワークロードでは，多くの場合 CPU に少数の木を配置し，全体の半分程度の DPU を first-class DPU とすることでスループットが向上することが確認できる．また，読み込みメインのワークロードの方が書き込みメインのものに比べて CPU に配置する木の割合を 2 から 4%程度まで増やしても高いスループットが得られた．これには，読み込みメインのワークロードのほうがキャッシュコヒーレンシを保つための DRAM アクセスが少ないといった原因が考えられる．

5.2　木の移動コスト

次に，木の移動にかかるオーバーヘッドを調べるために，ある DPU から別の DPU へ木を移動するのにかかる時間を測定した．移動した木のサイズを測定した転送時間で割った転送率を図 9 に示す．転送サイズが小さい場合を除き，1 秒あたりに移動できるデータ量はおよそ 6.4MB であることが確認できる．

UPMEM の場合，rank と呼ばれる 64 個の DPU を含む構成単位に対しては同じ MRAM のアドレスならば並列に CPU との通信が可能である．したがって，DPU に部分木の通信のためのバッファを用意しておき，各 DPU の部分木の移動は 1 度まで許容する

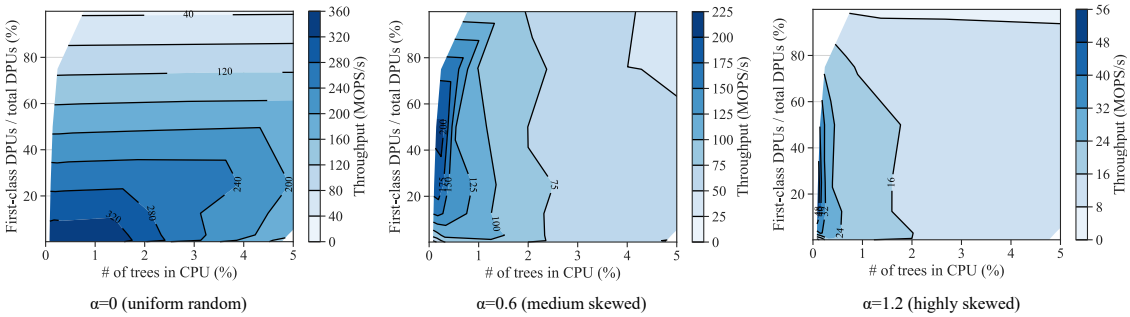

図 7　書き込みメイン (挿入クエリ : 検索クエリ = 1 : 1) のワークロードにおける部分木の配置とスループットの関係．

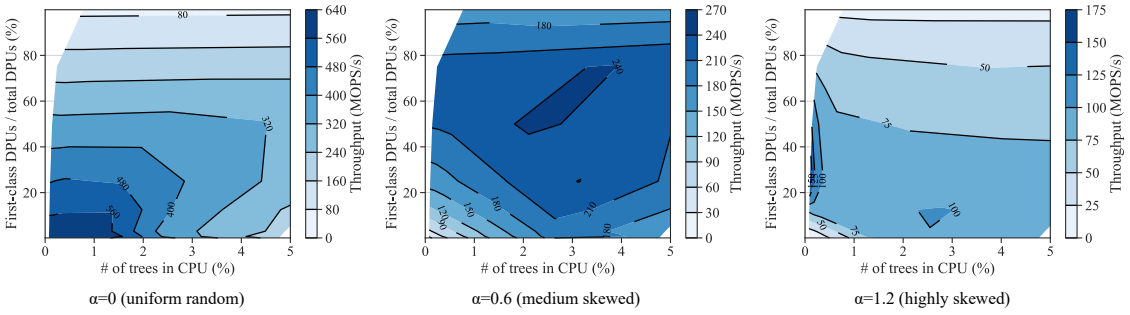

図 8　読み込みメイン (挿入クエリ : 検索クエリ = 1 : 19) のワークロードにおける部分木の配置とスループットの関係．

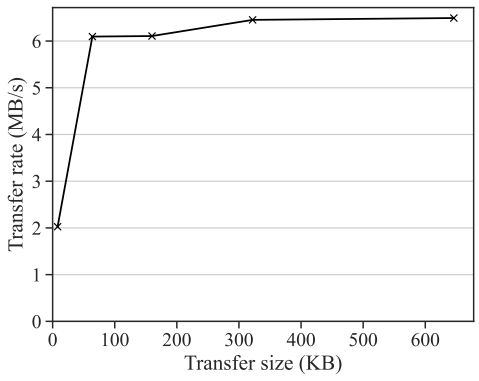

図 9　木のサイズと移動コストの関係．

とすれば，並列通信により 1 秒間に可能な木の移動は最大で $6.4 \times 64 = 409.6$MB ほどまで増えると考えられる．

現在 100 万クエリをまとめて一つのバッチとしているため，例えば 10MOPS/s の性能を求める場合，1 秒間に 10 個のバッチを実行する必要がある．この場合，1 回のバッチで許される木の移動量は最大 $409.6 \div 10 \simeq 41$MB となる．今後は，このような制約の中で，出来るだけ前節で高いスループットが得られた木の配置に近づけるような木の移動法を考案することが課題となる．

6　関連研究

PIM や分散メモリ並列計算機における探索木に関する研究は以前から行われており，それぞれ異なる木の分割法が試されている．Liu ら [10] は，我々の知る限り初めて PIM アーキテクチャに向けて最適化された並列データ構造である PIM-friendly skip list を提案した．しかし，キーの範囲による均等分割が用いられたが分割数が 8 から 16 と少なく，ワークロードの偏りへの対応が課題とされた．

Ziegler ら [15] は，分散メモリ並列計算機における探索木の 3 つの分割法について論じた．3 つの分割

法とは (i) キーのハッシュ値による分割 (ii) 細粒度の分割 (全てのノードをランダムに分散)(iii) 葉ノードは細粒度の分割，内部ノードはキーの範囲による均等分割を行うハイブリッドな分割 である．これらは 8 つのクラスターマシンで構成された分散システムに向けて実装されたが，数千の PIM モジュールを持つ PIM システムではそれぞれ以下の弱点があることが指摘されている [9]．(i) レンジクエリにおいて通信量が膨大である．(ii) ポインタを辿る度に通信が発生する．(iii) 内部ノードの部分で負荷の偏りが発生する．

Kang ら [8] は PIM の計算コストをモデル化し，理論的に負荷分散が保証された PIM-balanced skip list を考案した．PIM-balanced skip list では，根に近い層のノードは全ての PIM ノードに複製される．一方で根から遠い層のノードはランダムに PIM モジュールに分散される．この分割法に加え，キー a とキー b が途中まで共通の探索ルートをとった場合，その間のキー u もその探索ルートを通るという考えを利用し，キー u については木の途中から探索を開始することで，負荷分散が保証されている．しかし，この手法では前処理に時間がかかるとされている [9].

そこで，同じく Kang ら [9] により PIM-balanced skip list を基にした PIM-Tree が考案された．PIM-Tree が PIM-balanced skip list と異なる点は，ノードが存在する PIM ノードで探索するだけでなく，クエリが集中したノードを CPU にコピーし，CPU で探索を行う点である．偏りに応じて動的にノードを移動するという意味では我々も同じ手法を取っているが，分割法が大きく異なる．Kang らは細粒度の分割によって負荷分散を達成するという方針である．一方で我々はより粗粒度の (部分木単位の) 分割によって木を辿る際の通信を出来るだけ減らすという発想から出発し，部分木単位の移動により負荷分散を行う方針をとっている．

7 まとめと今後の課題

本研究では，PIM アーキテクチャにおいて探索木の負荷分散を行う方法を提案した．探索木を PIM ノードの数よりも多くの部分木に分割し，バッチ内の負荷の偏りに応じて部分木の配置を変えることで，クエリの量を均等にする．実在 PIM アーキテクチャ UPMEM 上に B^+ 木を基にした探索木を実装し，時間的に変化しない負荷の偏りを持ったワークロードに対する性能が CPU を上回ることを示した．また，部分木の移動にかかるオーバーヘッドを調べ，移動量の制限を考察した．今後は，部分木の移動を並列に行い，かつ少ない移動でスループットの向上が可能となるような移動法を考案，実装することが課題である．

謝辞 本研究の一部は，JSPS 科研費 JP22H03566 の助成を受けたものです.

参 考 文 献

[1] Acar, U. A., Anderson, D., Blelloch, G. E., and Dhulipala, L.: Parallel Batch-Dynamic Graph Connectivity, *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, New York, NY, USA, Association for Computing Machinery, 2019, pp. 381–392.

[2] Alser, M., Bingöl, Z., Cali, D. S., Kim, J., Ghose, S., Alkan, C., and Mutlu, O.: Accelerating Genome Analysis: A Primer on an Ongoing Journey, *IEEE Micro*, Vol. 40, No. 5(2020), pp. 65–75.

[3] Bayer, R. and McCreight, E.: Organization and maintenance of large ordered indices, *Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control - SIGFIDET '70*, (1970), pp. 107–141.

[4] Boroumand, A., Ghose, S., Akin, B., Narayanaswami, R., Oliveira, G. F., Ma, X., Shiu, E., and Mutlu, O.: Google Neural Network Models for Edge Devices: Analyzing and Mitigating Machine Learning Inference Bottlenecks, *CoRR*, Vol. abs/2109.14320(2021).

[5] Casado, R. and Younas, M.: Emerging trends and technologies in big data processing, *Concurrency and Computation: Practice and Experience*, Vol. 27, No. 8(2015), pp. 2078–2091.

[6] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R.: Benchmarking Cloud Serving Systems with YCSB, *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, New York, NY, USA, Association for Computing Machinery, 2010, pp. 143–154.

[7] Dean, J. and Barroso, L. A.: The Tail at Scale, *Commun. ACM*, Vol. 56, No. 2(2013), pp. 74–80.

[8] Kang, H., Gibbons, P. B., Blelloch, G. E., Dhulipala, L., Gu, Y., and McGuffey, C.: The Processing-in-Memory Model, *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, New York, NY, USA, Association for Computing Machinery, 2021, pp. 295–306.

[9] Kang, H., Zhao, Y., Blelloch, G. E., Dhulipala, L., Gu, Y., McGuffey, C., and Gibbons, P. B.: PIM-

Tree: A Skew-Resistant Index for Processing-in-Memory, *Proc. VLDB Endow.*, Vol. 16, No. 4(2022), pp. 946–958.

[10] Liu, Z., Calciu, I., Herlihy, M., and Mutlu, O.: Concurrent Data Structures for Near-Memory Computing, *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '17, New York, NY, USA, Association for Computing Machinery, 2017, pp. 235–245.

[11] Mutlu, O., Ghose, S., Gómez-Luna, J., and Ausavarungnirun, R.: A Modern Primer on Processing in Memory, *CoRR*, Vol. abs/2012.03112(2020).

[12] Paul, W., Vishkin, U., and Wagener, H.: Parallel dictionaries on 2–3 trees, *Automata, Languages and Programming*, Diaz, J.(ed.), Berlin, Heidelberg, Springer Berlin Heidelberg, 1983, pp. 597–609.

[13] UPMEM: UPMEM, 2023. `https://www.upmem.com/technology/`.

[14] Zhang, H., Chen, G., Ooi, B. C., Tan, K.-L., and Zhang, M.: In-Memory Big Data Management and Processing: A Survey, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 27, No. 7(2015), pp. 1920–1948.

[15] Ziegler, T., Tumkur Vani, S., Binnig, C., Fonseca, R., and Kraska, T.: Designing Distributed Tree-Based Index Structures for Fast RDMA-Capable Networks, *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, New York, NY, USA, Association for Computing Machinery, 2019, pp. 741–758.

[16] Zipf, G. K.: *Human behavior and the principle of least effort: An introduction to human ecology*, Ravenio Books, 2016.

[17] 奥田光, 鵜川始陽: Processing-in-Memory アーキテクチャUPMEM に適した探索木に向けての予備調査, 第 25 回プログラミングおよびプログラミング言語ワークショップ, ポスター発表, March 2023.