論文 / 著書情報
Article / Book Information

論題(和文)	メニーコア環境における関係表分割アルゴリズムを用いたハッシュ結合演算の評価
Title(English)	
著者(和文)	中澤 正和, 西 方, 荒堀 喜貴, 横田 治夫
Authors(English)	Masakazu Nakazawa, Fang Xi, Yoshitaka Arahori, Haruo Yokota
出典(和文)	第7回データ工学と情報マネジメントに関するフォーラム論文集, , ,
Citation(English)	, , ,
発行日 / Pub. date	2015, 3

メニーコア環境における関係表分割アルゴリズムを用いたハッシュ結合演算の評価

中澤　正和† 　西　方†† 　荒堀　喜貴†† 　横田　治夫††

† 東京工業大学工学部情報工学科　〒 152–8552 東京都目黒区大岡山 2-12-1
†† 東京工業大学大学院情報理工学研究科計算工学専攻　〒 152–8552 東京都目黒区大岡山 2-12-1
E-mail: †{nakazawa,xifang}@de.cs.titech.ac.jp, ††{arahori,yokota}@cs.titech.ac.jp

あらまし　近年クロック周波数の上昇による CPU の性能向上が限界に達し，処理の高速化のために複数コアを搭載するマルチコア CPU を利用するのが主流となった．ソフトウェア開発もマルチコア CPU を意識したものになり，データベース分野においてもマルチコア CPU を用いたデータベース処理手法が研究されている．しかし大量のコアやハードウェアスレッドを用いた評価はまだ十分に行われていない．そこで本稿ではデータベース処理のうち特に負荷の大きいハッシュ結合演算に焦点をあて，これを 60 コア・240 ハードウェアスレッドを搭載するインテル Xeon Phi コプロセッサ上で実行した際の評価を行う．ハッシュ関数による関係表分割の有無や実行スレッド数等のパラメータの変化が演算性能に与える影響について評価結果を報告する．

キーワード　メニーコア，並列処理，ハッシュ結合

1. はじめに

近年クロック周波数の上昇による CPU の性能向上は限界に達している．電気信号の遅延の問題や，消費電力量と発熱量が増大する問題等があるからである．そこで計算機の処理の高速化を図るために，複数のコアを搭載するマルチコア CPU が使われるようになった．個々のコアのクロック周波数は低いが，処理を複数コアで分散させることで全体として高性能処理を実現できる．

CPU のマルチコア化の影響はソフトウェア開発にも大きく影響を与えている．マルチコア CPU の性能を最大限に引き出すために，並列処理に主眼をおいたプログラミングが行われるようになった．

データベース分野においても，マルチコア CPU を用いたデータベース処理手法が研究されるようになった．また CPU のマルチコア化に加え，メインメモリの大容量化・低価格化が進んだことでデータベース処理をメインメモリ上に収めることが可能となったので，I/O バウンドではなく CPU バウンド・メモリバウンドの側面から研究が行われてきた．

しかし将来さらにコア数の増加が予想される中，大量のコアやハードウェアスレッドを用いた評価はまだ十分に行われていない．そこで本稿ではデータベース処理のうち特に負荷の大きいハッシュ結合演算に焦点をあて，これを 60 コア・240 ハードウェアスレッドを搭載するインテル Xeon Phi コプロセッサ上で実行した際の評価を行う．並列化を意識したハッシュ関数による関係表の分割の有無の影響，実行スレッド数等のパラメータの影響について評価結果を報告する．

2. 関連研究

ハッシュ結合に関する研究は数多く行われてきた．Blanas らの研究ではインテル (Xeon X5650 : 2.67GHz，6 コア，2 コンテキスト/コア) とサン・マイクロシステムズ（現オラクル）(UltraSPARC T2 : 1.2GHz，8 コア，8 コンテキスト/コア) のプラットフォームで複数のアルゴリズムによるハッシュ結合の評価が行われた [1]．この研究で使われたハッシュ結合のアルゴリズムはハッシュ関数による関係表の分割を行わない no-partitioning（ベースライン）と，関係表の分割を行う shared-partitioning，independent-partitioning，radix-partitioning に大きく分けられる．最初に関係表を分割することで各コアで独立に結合を実行できキャッシュ・TLB のヒット率の向上が期待できるが，分割自体のコストやバリア同期のコストの問題が発生して実際には多くの条件下で no-partitioning が最速であるという結果に終わった．

その後 Balkesen らが Blanas らの研究もとに no-partitioning，radix-partitioning の再評価を行い，radix-partitioning の有効性を示している [2]．さらに Balkesen らはハッシュ結合とソートマージ結合の比較も行っている [3]．SIMD 命令の使用等によりソートマージ結合が有利であるといわれる中で，ハッシュベース結合が優位性をもつことを指摘した．

Kaldewey らは GPU を用いたハッシュ結合の研究を行った [4]．GPU 上でのハッシュ結合の問題点としてメモリ容量が数 GB に限られていることや GPU 上のメモリへのデータの転送に時間がかかることを挙げた上で，GPU が CPU サイドのメモリに直接アクセスする UVA(Unified Virtual Addressing) システムを用いて高い演算性能が得られることを示した．

本稿は Blanas らの研究をベースとしている．Blanas らのホームページ [5] で公開されているソースコードを本稿の実験環境で実行できるよう必要な変更を施し，実験を行った．

3. 前提知識

3.1 ハッシュ結合

関係データベースにおいて必要なデータを取得するために複数の関係表を結合する処理が数多く発生する．ハッシュ結合は関係表結合アルゴリズムの 1 つである．2 つの関係表 R・S（|R|≦|S|）と適当なハッシュ関数があるとき，以下の手順で実行する．

（1） ビルド

関係表 R の結合属性をハッシュ関数にかけ，ハッシュ値ごとにタプルをまとめたハッシュ表を作る．

（2） プローブ

関係表 S の結合属性をハッシュ関数にかけ，タプルごとに対応するハッシュ表のスロット (ハッシュバケット) を検索し突き合わせる．

ハッシュ結合を行う前にあらかじめ関係表をパーティションに分割しておくことで，対応するパーティションごとにハッシュ結合を行うことができる．パーティションはビルド・プローブで用いるものとは別のハッシュ関数を関係表の結合属性に適用し，同じハッシュ値のタプルをまとめることで得られる．関係表の分割は関係表のサイズが大きくハッシュ表がメモリに収まらない場合に有効な方法として推奨されている．また関係表を分割することでそれぞれのパーティションのハッシュ結合をコア・スレッドごとに独立に実行できるという点から並列化の対象にもなっている．

関係表を分割する段階をパーティションフェーズ，ハッシュ表を作る段階をビルドフェーズ，タプルとハッシュ表を突き合わせる段階をプローブフェーズと呼ぶ．

3.2 分割アルゴリズム

実験で使用する 4 つのアルゴリズムを説明する．

1 つ目の no-partitioning は関係表の分割を行わず (ベースライン)，他の 3 つは関係表の分割を行う．

関係表はバケット単位でタプルを保持しており，リスト構造をなしているものとする．スレッドへのタプルの分配はバケット単位で行う．

分割数を p，ハッシュバケット数を m，実行スレッド数を n とする．説明図では $2p = m$ の関係があり，1 つのパーティションが 2 つのハッシュバケットに対応している．また簡単のため関係表のバケット数は実行スレッド数と等しく n であるとする．つまり各々のスレッドは関係表のバケットを 1 つのみ処理する．

3.2.1 no-partitioning

図 1 は no-partitioning のアルゴリズムを表している．

no-partitioning は関係表の分割を行わない．ビルド・プローブを全スレッドでハッシュ表を共有しながら行う．ビルドフェーズにおいてタプル書き込みの同時操作を防ぐため，ラッチによって排他制御を行う必要がある．以後 no-part と呼ぶ．

3.2.2 shared-partitioning

図 2 は shared-partitioning のアルゴリズムを表している．

全スレッドで共有しながらハッシュ値に従って p 個のパーティションを作る．パーティションフェーズにおいてタプル書き込みの同時操作を防ぐため，ラッチによって排他制御を行う．ビルド・プローブはスレッドで独立して行う．以後 shared-part と呼ぶ．

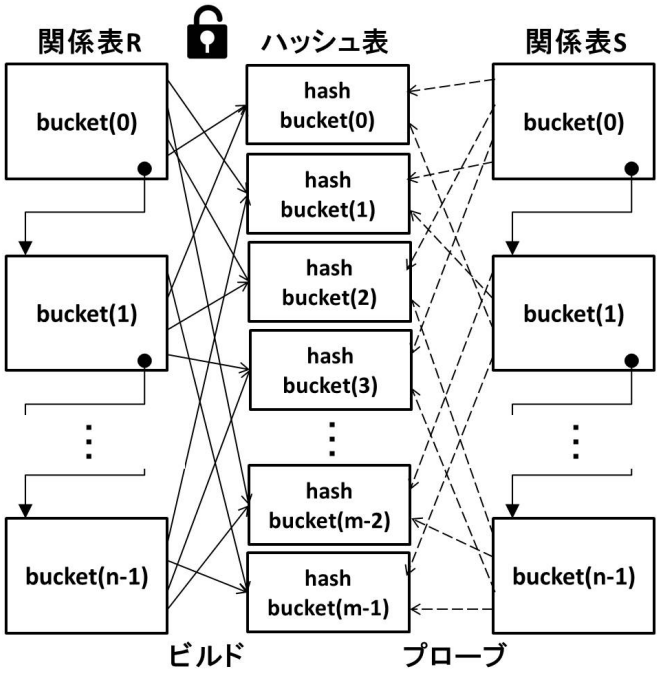

図 1 no-partitioning

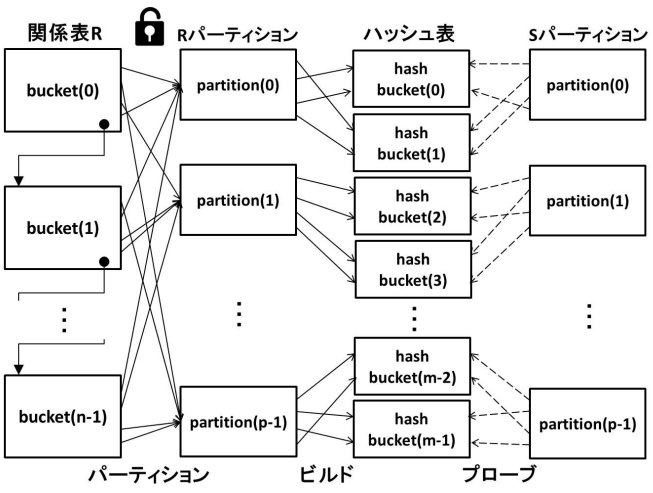

図 2 shared-partitioning

3.2.3 independent-partitioning

図 3 は independent-partitioning のアルゴリズムを表している．

初めに $p*n$ 個のタプル書き込み領域を用意し，各スレッドに p 個ずつ割り当てる．各スレッドがハッシュ値に従って割り当てられた領域にタプルを書き込むことでプライベートなパーティションができる．これらはスレッド間で共有していないので書き込みの排他制御は必要ない．全ての書き込みが終了したら対応するパーティションをリンクで接続し，最終的に p 個のパーティションができる．ビルド・プローブはスレッドで独立して行う．以後 independent-part と呼ぶ．

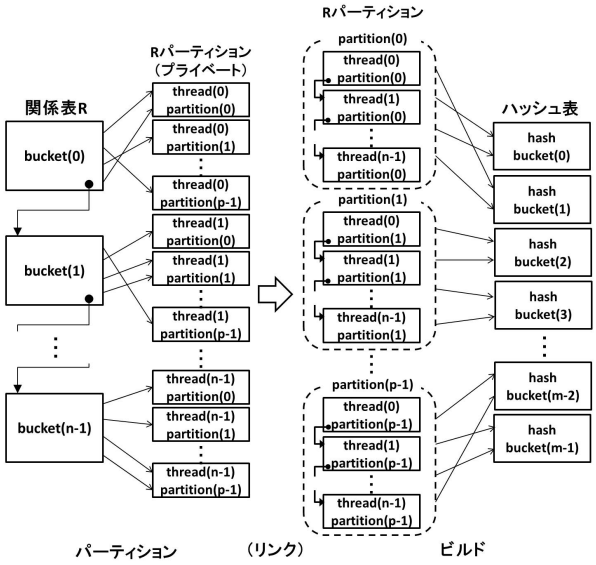

図 3　independent-partitioning

3.2.4　radix-partitioning

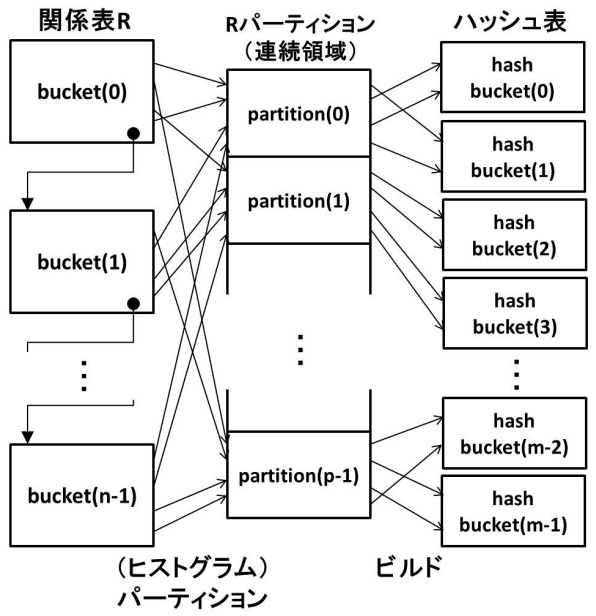

図 4　radix-partitioning

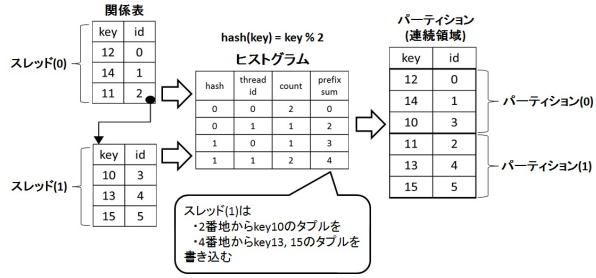

図 5　radix-partitioning の例

図 4 は radix-partitioning のアルゴリズムを表している．関係表の全てのタプルをあらかじめスレッドに分配する．各々のスレッドは割り当てられたタプルを読み込み，結合属性のハッシュ値ごとに数をカウントしておく．次に全スレッドで共有のヒストグラムを作り，連続メモリ領域上に「結合属性のハッシュ値」→「スレッド番号」の優先度で昇順に全タプルを出力した場合，どのスレッドがどのハッシュ値のタプルをどのアドレスから出力するのかを計算する *prefix sum* を行う．その後 *prefix sum* の結果を参照しタプルを書き込む．タプルの出力先が判明しているので排他制御は必要ない．最後にヒストグラムから各パーティションの先頭のアドレスを抜き出し，連続メモリ領域を論理的に分割してパーティションフェーズを終了する．ビルド・プローブはスレッドで独立して行う．以後 radix-part と呼ぶ．

図 5 に radix-part の例を示す．簡単のため分割数を 2，スレッド数を 2 としている．分割に用いるハッシュ関数は入力を 2 で割った余りを返す．

3.3　プログラム中の並列処理

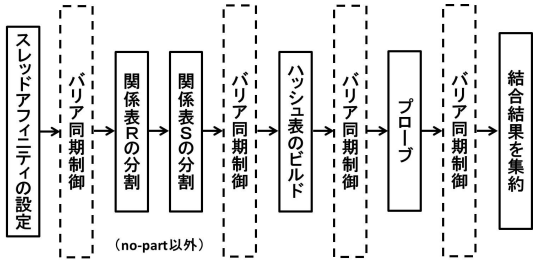

図 6　並列処理の流れ

図 6 は本稿で使用するプログラムの並列処理の流れを表している．ただし本稿ではメモリ容量の制約から結合結果をメモリ上に保存せず，プローブで結合条件が成立するとき 2 つのタプルから新たなタプルを構築する処理のみ行う．

「バリア同期制御」はスレッドの待ち合わせを意味する．全てのスレッドがバリア同期制御前の処理を終了するまで，どのスレッドも次の処理を開始することはできない．

3.4　Xeon Phi

Xeon Phi はインテル社が開発した Intel Many Integrated Core (MIC) アーキテクチャのコプロセッサである．本体をホスト PC の PCI-Express スロットに装着することで動作する．

3.4.1　特　　徴

本稿で用いる Xeon Phi の特徴について以下に列挙する．

- Linux をホスト PC から独立して実行 (カーネル 2.6.38.8)
- 60 コア・240 ハードウェアスレッド (1GHz，4 スレッド/コア，インオーダ実行，2 命令同時実行)
- GDDR5 メモリ (8GB，バンド幅 320GB/s)
- L1 キャッシュ (プライベート，32KB+32KB)
- L2 キャッシュ (プライベート，512KB，一貫性あり)
- ベクトル演算ユニット
- x86 ベースの従来のソースコードを利用可能

3.4.2　スレッドアフィニティ

Xeon Phi 上でプログラムを実行する際，スレッドを特定のコア (ハードウェアスレッド) に限定して実行させることがで

きる．本稿では Linux カーネル 2.6 以降の環境で使用可能な「sched_setaffinity()」API でスレッドアフィニティを設定する．

3.4.3 コアとメモリの接続とアルゴリズムの関係性

Xeon Phi は他の多くの汎用プロセッサとは異なる設計となっているが，本稿に関連して重要であると考えられるコアとメモリの接続について説明する．

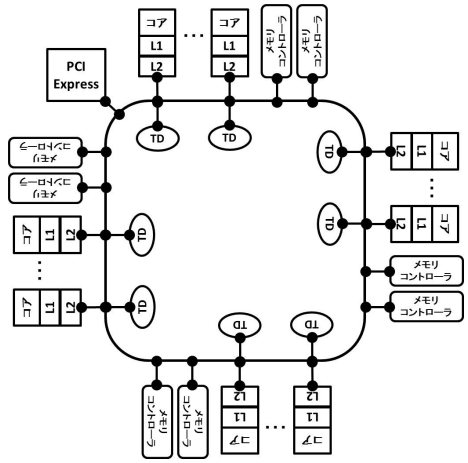

図 7　Xeon Phi のアーキテクチャ簡略図

図 7 は Xeon Phi のアーキテクチャの簡略図である [6]．

Xeon Phi はコア，メモリコントローラ，PCIExpress インターフェースがリングベースの双方向オンダイ相互接続 (ODI) で接続されている．アクセスしたいデータがすでに他のコアの L2 キャッシュ上に存在する場合，ODI 経由でアクセスすることが可能である．また分散タグディレクトリ (TD) によって，全てのコアの L2 キャッシュ上のデータは一貫性を維持するように管理されている．

Xeon Phi にはコア間で共有できる L3 キャッシュが無いが，これは関連研究 [1] とのプラットフォームの違いとして注目すべきことである．共有 L3 キャッシュが無いことによって，全スレッド間でハッシュ表の共有が必要な no-part では ODI 経由で頻繁に L2 キャッシュ間のデータ転送が行われ，性能が低下することが予想される．

またコアからメモリへの物理的距離によってアクセス速度が異なる NUMA(Non-Uniform Memory Access) の問題もある．処理対象のデータがコアから離れるほどアクセス速度は低下するので，分割アルゴリズムを用いてスレッドごとに独立に処理を行っているときスレッドによって処理時間が大きく偏りバリア同期の問題が顕著に表れる可能性がある．

以上で述べた点に注意しながら実験を行った．

4. 実　　験

4.1 データセット

本稿の実験では関連研究 [1] で使用されたスキューの無いデータセットを対象とする．ただしメモリ容量の関係から実際に処理するデータ量は減らしてある．

タプルは結合属性とペイロードの 2 つの属性を持ち，各々 8Byte として合計 16Byte の大きさをもつ．関係表 R の次数は 2^{22}(=4M) でデータ量 64MB，関係表 S の次数は 2^{26}(=64M) でデータ量 1GB とする．関係表 R の結合属性は 1 から 2^{22} までの値をとり，重複はしない．関係表 S の結合属性は 1 から 2^{22} までの値が各々16 ずつ存在するように調整されている．関係表 R のバケットサイズは 64KB，関係表 S のバケットサイズは 1MB である．

表 1 にデータセットに関する情報をまとめる．

表 1　データセット

関係表	次数	データ量	バケットサイズ
R	2^{22}	64MB	64KB
S	2^{26}	1GB	1MB

4.2 4 つのアルゴリズム

4.2.1 目　　的

最初に 3.2 で述べたそれぞれのアルゴリズムについてハッシュ結合にかかるサイクルの比較を行う．no-part 以外は分割数を変化させてその影響も確認する．実行スレッド数は 240 とする．

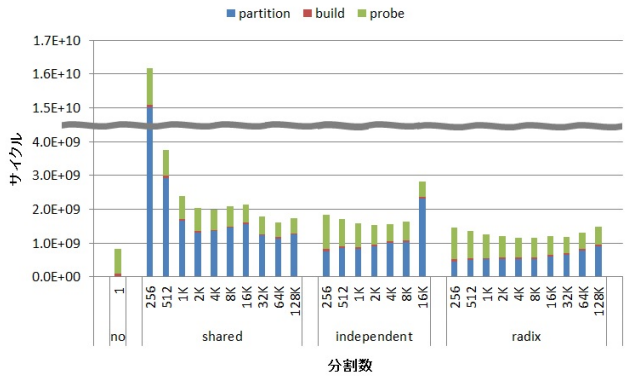

図 8　4 つのアルゴリズムの比較

4.2.2 結　　果

結果は図 8 のようになった．independent-part の分割数 32K 以降はメモリ不足によりプログラムが途中で停止した．

最速は no-part で，他の 3 つのアルゴリズムのどの分割数よりも高速である．

shared-part は分割数が少ないと極めて低速である．これはパーティションフェーズにおいてスレッド数に対してタプル出力先が十分に多くないため，排他制御によりスレッドの実行が制限されてしまうことが原因となっている．

independent-part は分割数が多いと低速になり，メモリ容量の制限からプログラムの実行が止まってしまうこともあった．これはパーティションフェーズで分割数*スレッド数の数だけプライベートパーティションを扱うことでワーキングセットが増大する・メモリ断片化が起きる問題や，パーティションがプライベートパーティションのリスト構造であることにより内部に無駄な空きが生じる問題が原因であると考えられる．

radix-part は no-part に次いで高速であり，分割数に関わらず安定した性能を示している．内訳を見ると分割数が増加する

ことでパーティションの時間が長く，ビルド・プローブの時間が短くなっていることがわかる．またパーティションは全て連続領域に書き込まれているので，メモリ不足やメモリ断片化の問題は起こらない．

関連研究 [1] の結果と比較すると no-part と radix-part の実行サイクルの関係は同じであった．どちらも no-part が最速であり，次いで radix-part が安定的に高速という結果になっている．3.4.3 で Xeon Phi のアーキテクチャとアルゴリズムの関係性について述べたが，この結果からはその影響は見られない．

以後は no-part と radix-part のみに注目して実験を行う．

4.3 スレッドアフィニティの影響

4.3.1 目　　的

3.4.3 に関連してスレッドアフィニティを変更した場合について比較を行う．no-part についてはアルゴリズム的に影響を受ける可能性は低い．一方で radix-part は連続メモリ領域上のパーティションのタプルの順序にスレッド番号が関係するので，異なるアフィニティによって違いが表れると考えた．

Xeon Phi のコア数は 60 でそれぞれに 0 から 59 までの番号が割り当てられている．実際にリング上にどのように配置されているかは不明である．

コア単位のアフィニティの比較を行うため実行スレッド数は 60(1 スレッド/コア) とする．アフィニティは 3 種類行う．1 つ目のアフィニティはコア番号に順番に連続で割り当てる．2 つ目のアフィニティはコア番号に 15 ずつ間隔をあけながら割り当てる．3 つ目のアフィニティはコア番号に 30 ずつ間隔をあけながら割り当てる．具体的なアフィニティを表 2 にまとめる．

表 2　3 種類のアフィニティ

スレッド番号	0	1	2	3	4	5	…	58	59
アフィニティ1（コア番号）	0	1	2	3	4	5	…	58	59
アフィニティ2（コア番号）	0	15	30	45	1	16	…	44	59
アフィニティ3（コア番号）	0	30	1	31	2	32	…	29	59

4.3.2 結　　果

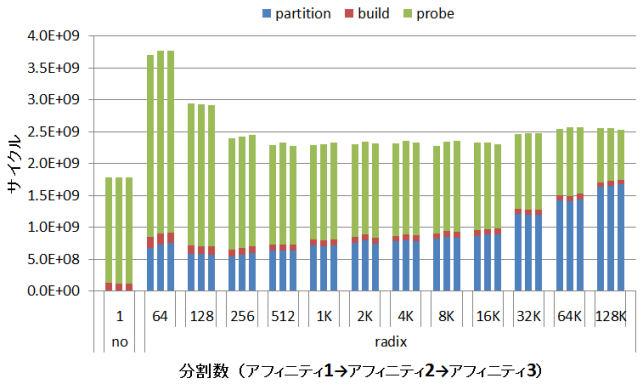

図 9　アフィニティによる比較

結果は図 9 のようになった．それぞれの分割数ごと結果を見ると，アフィニティの違いによる実行速度の差はほとんどない．アフィニティの違いがデータ転送に影響を与え何らかの違いが表れると考えていたが，無視できる程度であるといえるだろう．

以後アフィニティ1 を用いて実験を行う．

4.4 コア数の増加によるスケーリング

4.4.1 目　　的

本実験ではスレッドを実行するコア数を増加させてその影響を確認する．コア数の増加による影響に注目するため 1 スレッド/コアでの実行とする．

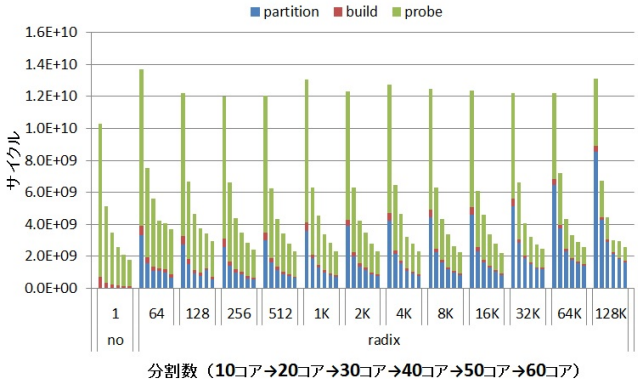

図 10　コア数による比較

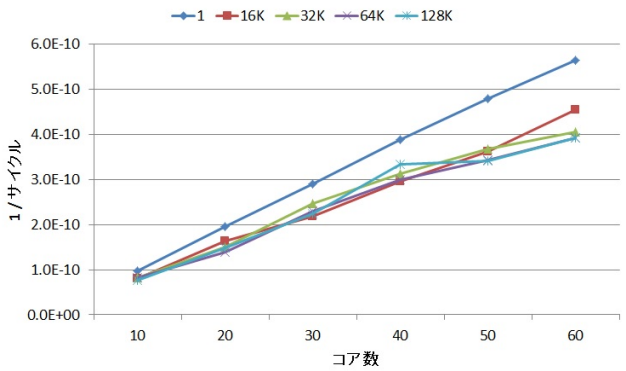

図 11　サイクルの逆数

4.4.2 結　　果

結果は図 10 のようになった．no-part と radix-part のどちらも実行コア数が増加するにつれ高速化していることがわかる．

図 11 は no-part(1) と radix-part の十分大きな分割数 (16K-128K) についてサイクルの逆数をとった結果を示している．この図から実行コア数の増加に対して線型的にスケーリング効果が得られることがわかる．ただし radix-part より no-part の直線の傾きが大きいので，no-part のスケーリングが優れているといえる．

4.5 ハードウェアスレッド数の増加によるスケーリング

4.5.1 目　　的

コア数の増加によるスケーリング効果を確認できたので，続いて SMT(Simultaneous Multithreading) を利用したハードウェアスレッド数の増加によるスケーリングを確認する．Xeon Phi は 1 コアあたり 4 つのハードウェアスレッドを搭載しているので，同時実行スレッドを 1 → 2 → 3 → 4 のように増加させる．つまり全コア数は 60 なので実行スレッド数を 60(1 スレッ

ド/コア) → 120(2 スレッド/コア) → 180(3 スレッド/コア) → 240(4 スレッド/コア) のように増加させてその影響を確認する．

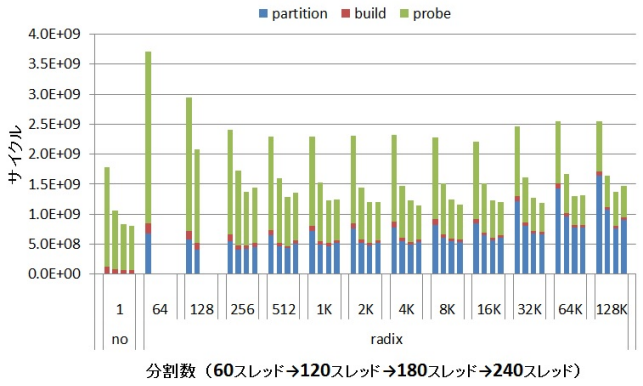

図 12　ハードウェアスレッド数による比較

4.5.2　結　　果

結果は図 12 のようになった．radix-part の分割数 64・128 で分割数が実行スレッド数より少ない項目については，ビルド・プローブフェーズで何も処理を行わないスレッドが発生してしまうため結果を表示しない．

no-part と radix-part のどちらも 60 → 240 で 2 倍程度高速化している．詳細に見ると 60 → 120 で 1.5 倍程度の大幅な高速化，120 → 180 で 1.2 倍程度の高速化，180 → 240 ではほとんど変化が見られず逆に低速化する場合もあるという結果になっている．

この結果からハードウェアスレッド数の増加によるスケーリングにおいては同時実行スレッド数が増えるにつれスケーリング効果は薄くなるといえる．これはインオーダ実行によるデータアクセスレイテンシを隠蔽するという性能向上と，同一コア上の同時実行スレッドがキャッシュ・TLB を共有するという性能低下の 2 つの効果によるものと考えられる．

4.6　スレッド同時実行による性能低下

4.6.1　目　　的

本実験では同一コア上での複数スレッドの同時実行による性能低下をより厳密に検証する．本来は等しい処理性能であるように実行スレッド数を 60 で固定し，同一コア上で実行するスレッド数を 1 スレッド/コア (60 コア) → 2 スレッド/コア (30 コア) → 3 スレッド/コア (20 コア) → 4 スレッド/コア (15 コア）のように変化させてその影響を確認する．

4.6.2　結　　果

結果は図 13 のようになった．no-part と radix-part のどちらも 1 スレッド/コア (60 コア) → 4 スレッド/コア (15 コア) でおよそ 0.85 倍程度低速化している．

これについて radix-part の分割数 32K でキャッシュミス・TLB ミスの計測を行った．計測はパーティション・ビルド・プローブの全てのフェーズを通している．

図 14 は全スレッドの L2 キャッシュミスの合計回数を表している．L2 キャッシュミスは read ミス/write ミスと，他コアの L2 キャッシュからのフィル/メモリからのフィルを組み合わせた合計 4(=2*2) 種類ある．全種類の L2 キャッシュミスの合計

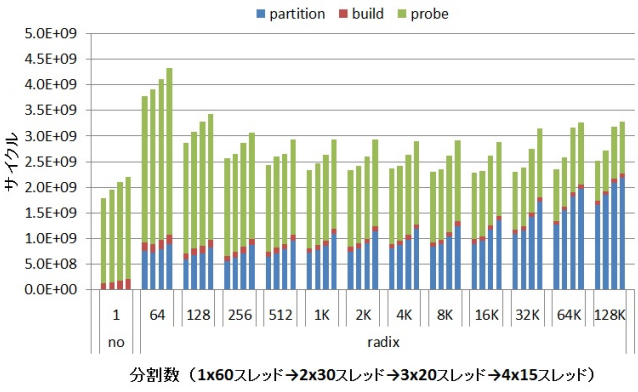

図 13　同時実行スレッド数による比較

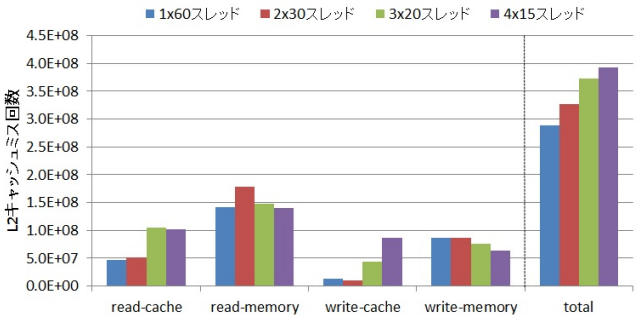

図 14　L2 キャッシュミス

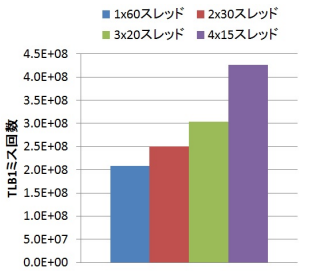

図 15　TLB1 ミス

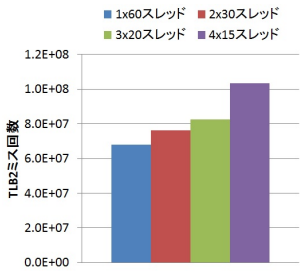

図 16　TLB2 ミス

を右端のグラフで表している．図 14 から同時実行スレッド数を増やすとキャッシュフィルの L2 キャッシュミスは増加し，メモリフィルの L2 キャッシュミスは減少する傾向があることがわかる．また L2 キャッシュミス全体としては増加している．

図 15 は全スレッドの TLB1 ミスの合計回数を，図 16 は全スレッドの TLB2 ミスの合計回数を表している．どちらも同時実行スレッド数を増やすと増加する傾向にある．

このような結果になったのは，同一コアで実行しているスレッドが L2 キャッシュ・TLB を共有していることにより，同時実行スレッド数が増加するとコアあたりのワーキングセットが増加することが原因であると考えられる．

以上より複数スレッドの同時実行は性能低下を引き起こすことが確認できた．

4.7　ハッシュバケットサイズの影響

4.7.1　目　　的

これまでの実験では no-part が radix-part より高速であるという結果が出ている．そこでハッシュ表に関するパラメータに注目してみると，ハッシュバケットに 2 つずつタプルが分配さ

れるような設定になっていたが，キャッシュの局所性を考えるとこれでは関係表分割の効果が十分に活かせられない．

もしハッシュバケットサイズが大きい (ハッシュバケットあたりのタプル数が多い) ならばハッシュバケットのキャッシュへの転送コストが大きくなる．これにより no-part は処理時間が増加すると考えられるが，radix-part はプローブフェーズで同じパーティションのタプルを連続で処理するためハッシュバケットの転送回数が比較的少なく，radix-part が優れた性能を示すと予想できる．

本実験ではハッシュバケットサイズを増加させてその影響を確認する．使用するデータセットは変えないので，結合属性に重複する値を入れるのではなくハッシュ値の衝突を発生させる．実行スレッド数は 60(1 スレッド/コア) とする．

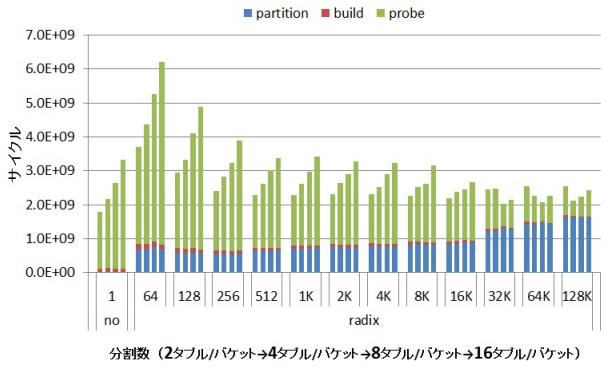

図 17　ハッシュバケットサイズによる比較

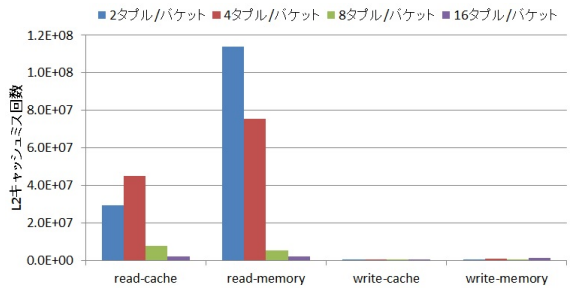

図 18　L2 キャッシュミス

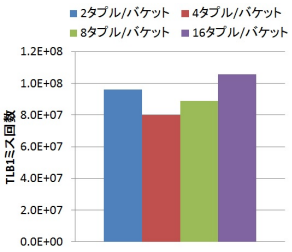

図 19　TLB1 ミス

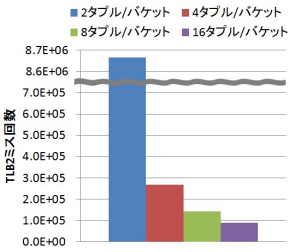

図 20　TLB2 ミス

4.7.2　結　　果

結果は図 17 のようになった．no-part と radix-part の分割数 16K まではハッシュバケットサイズが増加すると低速になることがわかる．これはハッシュバケットの転送コストの増加に加え，プローブフェーズでのタプルの結合属性の比較演算数の増加が影響している．

一方で radix-part の分割数 32K 以降では単調には速度低下していない．これについて，radix-part の分割数 32K でプローブフェーズでのキャッシュミス，TLB ミスの計測を行った．

図 18 は全スレッドの L2 キャッシュミスの合計回数を表している．特に read ミスが減少している様子がわかる．

図 19 は全スレッドの TLB1 ミス，図 20 は TLB2 ミスの合計回数を表している．TLB1 ミスは減少しているとはいえないが，処理性能に大きく影響する TLB2 ミスは減少傾向にある．

以上よりハッシュバケットサイズが大きいならば no-part よりも radix-part のほうが高速であることがわかった．

4.8　関係表のサイズ比の影響

4.8.1　目　　的

これまではサイズ比が 1 対 16 の 2 つの関係表を対象としてきたが，本実験では 2 つの関係表のサイズが等しい場合の影響を確認する．具体的には次数が 2^{25}(=32M) でデータ量 512MB の関係表を 2 つ用意し，これを結合する．結合属性は 1 から 2^{25} までの値をとり，重複はしない．関係表のバケットサイズはどちらも 512KB とする．表 3 に本実験のデータセットに関する情報をまとめる．

実行スレッド数は 4.5 と同じく 60(1 スレッド/コア) → 120(2 スレッド/コア) → 180(3 スレッド/コア) → 240(4 スレッド/コア) のように増加させる．

表 3　データセット

関係表	次数	データ量	バケットサイズ
R	2^{25}	512MB	512KB
S	2^{25}	512MB	512KB

4.8.2　結　　果

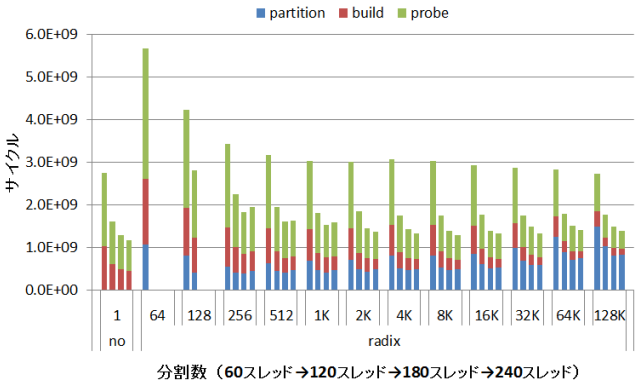

図 21　関係表のサイズが等しい場合の比較

結果は図 21 のようになった．元のデータセットに比べてビルド側の関係表 R の大きさが 512MB/64MB=8 倍となってい

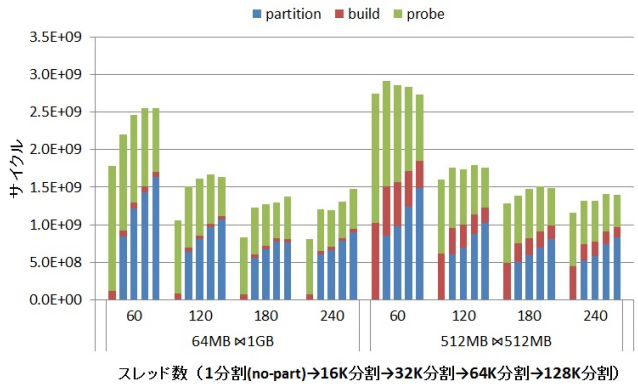

図 22　関係表のサイズによる比較

るので，その分だけビルドに時間がかかっている．

図 22 は図 12 と図 21 の no-part と十分大きな分割数の radix-part について各データセット・スレッド数ごとにまとめたものである．radix-part はビルド側の関係表が大きくてもビルドコストを低く抑えられており no-part との差はかなり小さい．

以上より関係表のサイズが等しい場合では no-part と radix-part は同程度の性能である．

5. おわりに

5.1 本稿のまとめ

本稿では大量のコア・ハードウェアスレッドを搭載する Xeon Phi 上でハッシュ結合演算を実行し，性能評価を行った．ハッシュ関数を用いた関係表分割の有無や実行スレッド数等のパラメータの変化が与える影響について実験を行ったことで以下のことが分かった．

- 4 つのアルゴリズム

no-part：最速．

shared-part：分割数が少ないと競合が発生して低速．

independent-part：分割数が多いとメモリ不足．

radix-part：no-part に次いで高速．分割数に関わらず安定．

- スレッドアフィニティの影響

スレッドアフィニティは性能に影響を与えない．

- コア数の増加によるスケーリング

コア数の増加に対して線型的なスケーリング効果が得られる．

- ハードウェアスレッド数の増加によるスケーリング

インオーダ実行によるデータアクセスレイテンシを隠蔽するという性能向上と，同一コア上の同時実行スレッドがキャッシュ・TLB を共有するという性能低下の 2 つの効果が影響し，次第にスケーリング効果は薄くなる．

- スレッド同時実行による性能低下

全実行スレッド数を固定し，同一コア上の同時実行スレッド数を増加させると低速化．L2 キャッシュミス・TLB ミスの増加を確認．

- ハッシュバケットサイズの影響

ハッシュバケットサイズが十分大きいとき radix-part は no-part より高速である．

- 関係表サイズ比の影響

2 つの関係表サイズが等しいとき no-part と radix-part は同程度の性能である．

以上よりメニーコア環境においてハッシュ結合演算を実行する場合，コア数によるスケーリングが行われればその性能は線型的に向上する．ハードウェアスレッド数の増加によるスケーリングも若干の効果はあるが次第に性能向上率は小さくなる．データセットの結合属性に重複が多く存在する場合などはハッシュバケットサイズが増加するため no-part よりも radix-part を用いるべきである．また 2 つの関係表サイズの差が小さいならば radix-part の使用を検討してよい．

最後に，3.4.3 で述べたような Xeon Phi の特殊なコア・メモリのリング接続の影響が特に見られなかったのは残念であった．L2 キャッシュ間のデータ転送は高速で共有 L3 キャッシュの性能に比べて劣らないものであり，コアからメモリへの物理的距離にかかわらず転送速度は均一な UMA(Uniform Memory Access) システムであると考えて差し支えないかもしれない．

5.2 課　　題

radix-part において分割数を大きくすると分割自体のコストが増加してしまう．現在の分割アルゴリズムにはまだ改善の余地（マルチパス分割の工夫等）があるので検討したい．

本稿ではデータセットにスキューを想定していない．スキューによって no-part では競合の問題が，radix-part ではバリア同期の問題が顕著に表れると考えられるので，実験を行って確かめる必要がある．

本稿の性能評価は Xeon Phi の特徴を十分に反映していないのでプログラムの修正等が必要である．具体的にはソフトウェアプリフェッチ，ヒュージページ，SIMD 命令の積極的な使用が考えられる．

6. 謝　　辞

本研究の一部は、日本学術振興会科学研究費補助金基盤研究（A）(#25240014) の助成により行われた。

文　　献

[1] S. Blanas, Y. Li, and J. M. Patel. “Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs,” in SIGMOD conference, pp. 37-48, 2011.

[2] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. “Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware,” in ICDE conference, pp. 362-373, 2013.

[3] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. “Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited,” in Proceedings of the VLDB Endowment, Vol. 7, No. 1, pp. 85-96, 2013.

[4] D. Kaldewey, G. Lohman, R. Mueller, and P. Volk. “GPU Join Processing Revisited,” in DaMoN workshop, pp. 55-62, 2012.

[5] Spyros Blanas - Home Page. http://web.cse.ohio-state.edu/~sblanas/.（2015/1/13 アクセス）

[6] J. Jeffers, and J. Reinders. “Intel Xeon Phi Coprocessor High Performance Programming,” Elsevier Inc., 2013 (すがわら きよふみ・エクセルソフト株式会社 (訳).「インテル Xeon Phi コプロセッサー ハイパフォーマンス・プログラミング」，株式会社カットシステム，8-9・289-290 頁，2014)