データベースシステム 113−25
(1997. 7. 15)

分散共有メモリ計算機における並列ハッシュ結合演算処理方式の設計と実装

今井 洋臣 †‡, 中野 美由紀 †, 喜連川 優 †

† 東京大学生産技術研究所
‡ ソニー アーキテクチャ研究所 ソフトウェアラボラトリー

近年の増大する一方のデータベース処理に対し、、比較的並列プログラムの実装が容易かつ台数拡張性の高い分散共有メモリ並列計算機が今後のデータベースシステムおけるプラットフォームとして着目を浴びている。しかしながら、共有メモリ型計算機における単純な実装方式のままでは、メモリの一貫性を保持するためのコストがかかり、期待した性能が出ないことがある。

本報告では、関係データベースでも負荷の高い結合演算を対象として、分散共有メモリ計算機上での並列データベースアルゴリズムの実装方式について検討する。ハッシュ結合演算処理におけるデータバッファ、ハッシュテーブル領域の獲得およびアクセス方針に関し、アクセス局所性に着目した実装方式を提案し、実際に分散共有計算機 HP-Convex Exemplar 1600 上に実装し、性能評価を行い、我々が提案するバッファ管理方式が有効であることを示す。

Design and Implementation of Parallel Hash Join in Distributed Shared Memory Machines

Hiro-omi Imai†‡, Miyuki Nakano† and Masaru Kitsuregawa†

†Institute of Industrial Science, University of Tokyo
‡ Software Laboratory Architecture Laboratories Sony Corp.

The distributed shared memory(DSM) architecture is considered to be one of the most likely parallel computing environment candidate for the near future because of its ease of system scalability and facilitation for parallel programming. However, a naive program based on shared memory execution on a DSM machine often deteriorates performance. In this paper, we propose four buffer management strategies for parallel hash join processing on the DSM architecture and actually implement them on the HP Exemplar SPP 1600. From results, it is shown that careful buffer management of parallel join processing on DSM can produce considerable performance improvements in comparison with a naive implementation.

1　はじめに

近年，データベースシステムは肥大化するデータへの容易な拡張性，複雑化する問合せへの迅速な応答などの高性能化が求められており，商用並列計算機上に多くの並列データベースシステムが実装されるようになった．しかしながら，並列データベース処理の研究は共有メモリ型または分散メモリ型計算機における並列関係データベースシステムを対象とした研究は多く行われているが，分散共有メモリ型計算機上での並列処理アルゴリズムの研究は今だ少ない．今後増大する一方の並列データベース処理およびユーザアプリケーションの移植性を考慮すると，比較的プログラムの実装が容易で，台数拡張性の高い分散共有メモリ並列計算機上での並列データベース処理アルゴリズムの研究，開発はこれからの重要な課題と言える．

分散共有メモリ並列計算機の主記憶空間へのアクセスはよく知られているように，採用されている分散共有メモリアーキテクチャにもよるが，すでにローカルな空間にマッピングされているページへの参照と他ノード空間にマッピングされているページへの参照，書き込みではアクセス時間に非常に大きな差がある．したがって，広大な主記憶空間上に一様にデータを保持，アクセスするデータベース処理は，単純な実装のままでは，期待した性能が出ない．この影響を軽減する方法として各ノードに分散している物理的なメモリの配置を考慮してメモリアクセスの局所性を高めるようにアプリケーションを実装することが考えられる．

本稿では分散共有メモリマシン上での関係データベース処理実装方式を検討するために，並列ハッシュ結合演算を商用マシン (HP Exemplar SPP-1200) 上に実装し，分散共有メモリマシン上のメモリアクセスの局所性を考慮した実装方式の検討を行う．

2　実験機 Exemplar SPP1200

Exemplar SPP1200 は分散共有メモリ型計算機である．8CPU を密結合してノードを構成し，ノード間をネットワークで疎結合している．ノード内は共有メモリ，ノード間は分散共有メモリである．分散共有はキャッシュコヒーレントで実現され，メモリの一貫性は 64byte のキャッシュライン単位で保たれる．

ノード上のメモリは大きく次の 3 つの領域に分類される．

1. グローバルメモリ
   4Kbyte 単位でノード間にまたがって割り付けられたメモリ領域．どのノードからも参照可能である．プログラマはアドレスがどのノード上に割り付けられたか特定できない．
2. ローカルメモリ
   ノードに明示的に割り付けられたメモリ領域で他ノードからも参照可能である．
3. ネットワークキャッシュ (以下では単にキャッシュ)
   他ノードのデータをキャッシュするのに使われるメモリ領域．プログラマが操作することは出来ない．

本研究室のマシンは 4 ノード構成で，32CPU と 2GB のメモリを持っている (8CPU+512MB/node 図 1).

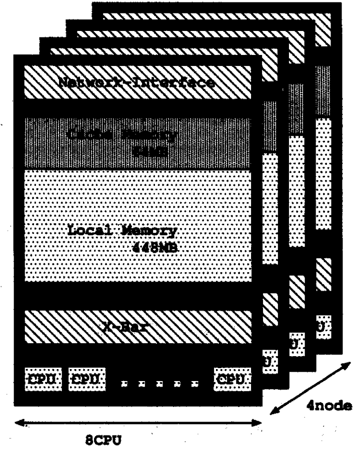

図 1:SPP1200 diagram

3　並列ハッシュ結合演算実装方式

3.1　並列 Grace ハッシュ結合演算処理

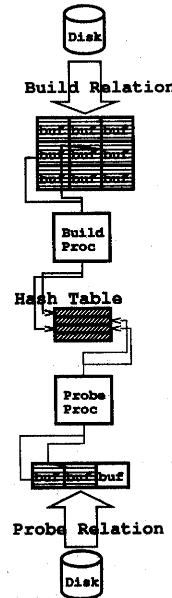

図 2:Hash_Join

並列 Grace ハッシュ結合演算処理はビルドリレーションからハッシュテーブルを生成するビルドフェーズ及びそのハッシュテーブルを走査しプローブリレーションとの結合処理を行うプローブフェーズからなり，各々のフェーズ処理のためには入出力処理を行うディストリビューションプロセス，ビルド処理を行うビルドプロセス，プローブ処理を行うプローブプロセスが構成されている．

ビルドフェーズ

ディストリビューションプロセスがディスクから読みだしたタプルを主記憶上のデータバッファに書き込む．ビルドプロセスがデータバッファからタプルを読み出し，そのポインタをハッシュテーブルに登録する．

プローブフェーズ

ディストリビューションプロセスがディスクからタプルを読みだし，データバッファに書き込む．プローブプロセスがデータバッファからタプルを読みだし，ハッシュテーブルを走査，タプルをつき合わせる．ビルドフェーズと異なり，ハッシュテーブルを走査し結合処理を行った後はプローブリレーションを保持する必要はない．そのため数個のデータバッファしか用意せず，そのバッファを再利用してプロセス間のデータ通信を行う．なおビルドフェーズ プローブフェーズともバッファはリスト形式で管理されている．

3.2 各実装方式

本稿の実装では並列ハッシュ結合演算を対象として単純な共有メモリ型実装方式 (SE1,SE2 方式) とメモリの物理的な分散を意識しアクセスの局所性を考慮した実装方式 (SN1,SN2 方式) を実装し，性能を比較した．表 1 に各実装方式の特徴をまとめ，本節では各実装方式の詳細について述べる．

	SE1	SE2	SN1	SN2
ハッシュテーブル	Global	Global	Local	Local
データボディ	Global	Local	Local	Local
ビルドフェーズ プロセス間 データ転送	inter node	intra node	inter node	inter node
プローブフェーズ プロセス間 データ転送 (通信バッファ使用)	inter node	intra node	inter node	intra node

表 1 各種実装方式

3.2.1 SE1 方式

SE1 ではハッシュテーブル データバッファ共にグローバルメモリ上に確保している (図 3).

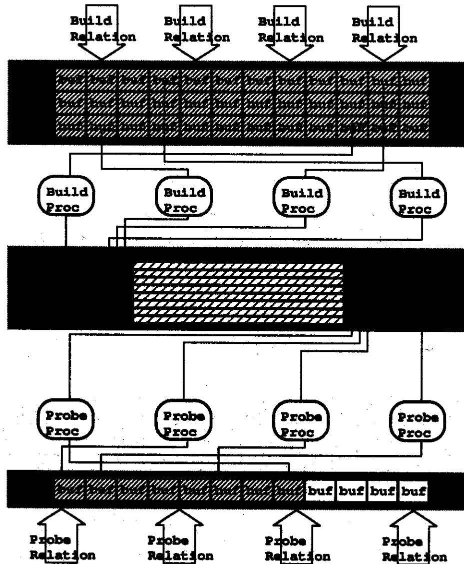

図 3:SE1

ビルドフェーズ ボディを書き込むためのデータバッファがグローバルメモリ上に取られ，単一のフリーリストによって管理される．

ディストリビューションプロセスはフリーリストからグローバルメモリ上のバッファを獲得し，ディスクから読み出したタプル書き込む (図 3 上)．バッファが一杯になるとそのバッファをビルドプロセスのデータリストへリンクする．データリストはグローバルに唯一つあり，各ビルドプロセスに共有されている．

ビルドプロセスはグローバルメモリ上に単一のハッシュテーブルを作成する．ビルドプロセスが前出のデータリストからバッファを取り込み，タプルをキーの値を基にハッシュテーブルへタプルのポインタを登録する．ハッシュテーブルに登録する際はロックを必要とする．

プローブフェーズ 数十個のバッファがグローバルメモリ上に確保し，単一のフリーリストで管理される．

ディストリビューションプロセスはフリーリストからグローバルメモリ上のバッファを獲得し，ディスクから読み出したタプル書き込む (図 3 下)．バッファが一杯になるとそのバッファをプローブプロセスのデータリストへリンクする．データリストはグローバルに唯一つあり，各プローブプロセスに共有されている．

プローブプロセスはデータリストからバッファを取り込み，グローバルなハッシュテーブルを走査し，プローブする．プローブリレーションは保持しておく必要は無い．読み出しを終えたバッファはディストリビューションプロセスのフリーリストへリンクされ再利用される．

3.2.2 SE2 方式

SE2 方式は SE1 方式に若干の改良を加えたもので，ハッシュテーブルはグローバルメモリ上に取られるものの，データバッファはローカルに確保される．

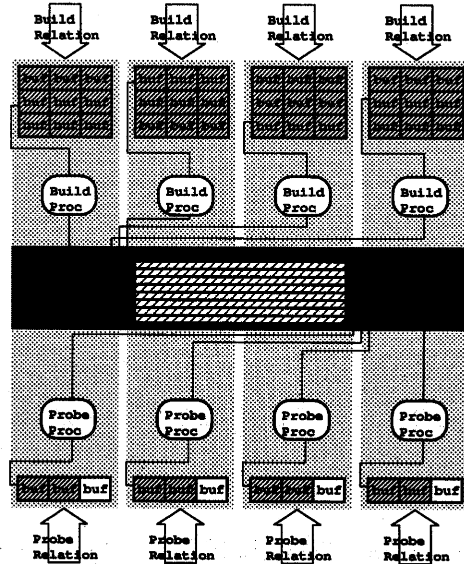

図 4:SE2

SE1 方式の問題点はディストリビューションプロセスの役割が単にディスクからデータを読み出すのみであったにもかかわらず，他ノード上のメモリへ書き込みを行っていた点である．この点を改善するため SE2 方式ではディストリビューションプロセスが読み出したデータはローカルメモリ上のバッファに書き込まれ，書き込まれたバッファは自ノード上のビルドプロセス / プローブプロセスにリンクする．したがって SE2 ではハッシュテーブルデータはグローバルに確保されるものの，データバッファはローカルに確保される．

ビルドフェーズ ボディを書き込むためのデータバッファがローカルメモリ上に取られ，各々ローカルなフリーリストによって管理される．

ディストリビューションプロセスはフリーリストからローカルメモリ上のバッファを獲得し，ディスクから読み出したタプルを書き込む (図 3)．バッファが一杯になるとそのバッファを自ノード上のビルドプロセスのデータリストへリンクする．データリストは各ビルドプロセスが独自に持っている．

ビルドプロセスはグローバルメモリ上に単一のハッシュテーブルを作成する．ビルドプロセスがデータリストからバッファを取り込み，キーの値を基にハッシュテーブルへタプルのポインタを登録する．ビルドプロセスがハッシュテーブルに書き込みを行う際にはロックを要する．

プローブフェーズ 数個のバッファがローカルメモリ上に確保され，ノード独自のフリーリストで管理される．

ディストリビューションプロセスはローカルなフリーリストからローカルメモリ上のデータバッファを獲得し，ディスクから読み出したタプルを書き込む (図 4)．バッファが一杯になるとそのバッファを自ノード上のプローブプロセスのデータリストへリンクする．データリストは各プローブプロセスが独自に持っている．

プローブプロセスはデータリストからバッファを取り込み，グローバルなハッシュテーブルを走査し，プローブする．プローブリレーションは保持しておく必要が無いため，読み出しを終えたバッファはディストリビューションプロセスの自ノード上のディストリビューションプロセスのフリーリストへリンクされ再利用される．

3.2.3 SN1 方式

ハッシュテーブルが分割されてノード毎にローカルなテーブルを持っており，それぞれのテーブルのエントリは重ならない．

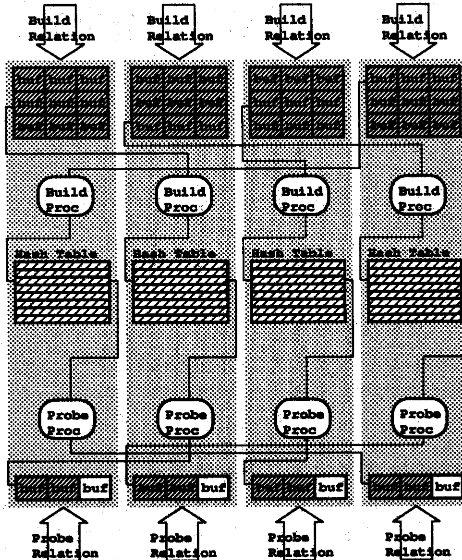

図 5:SN1

ビルドフェーズ ボディを書き込むためのデータバッファが各ローカルメモリ上に取られ，各々ローカルなフリーリストによって管理される．

ディストリビューションプロセスはローカルなフリーリストからローカルメモリ上のデータバッファを獲得する．ディストリビューションプロセスによってディスクから読み出されたタプルはそのキーの値を基にハッシュ関数によってどのテーブルに属するかを決定され，そのテーブルを持っているビルドプロセスに割り当てられたデータバッファに書き込まれる．

バッファが一杯になるとそのバッファを処理先のビルドプロセスのデータリストへリンクする．データリストは各ビルドプロセスが独自に持っている．

ビルドプロセスはローカルメモリ上にローカルなハッシュテーブルを作成する．ビルドプロセスはローカルのデータリストからバッファを取り込み，タプルのキーの値を基に自ノード上のローカルなハッシュテーブルへタプルのポインタを登録する．

プローブフェーズ 数個のバッファがローカルメモリ上に確保され，ノード独自のフリーリストで管理される．

ディストリビューションプロセスはビルドプロセスと同様に動作する．

プローブプロセスはデータリストからバッファを取り込んだ後ローカルなハッシュテーブルを走査し，プローブを行う．

プローブプロセスが読み終えたバッファは再びディストリビューションプロセスのフリーリストへリンクされ再利用される．

3.2.4 SN2 方式

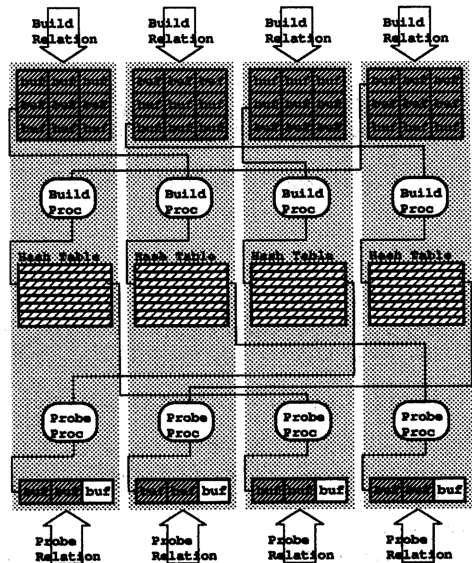

図 6:SN2

SN2 方式は SN1 方式と比較してプローブフェーズのみ異なっている．

SN1 方式ではプローブフェーズにおいてディストリビューションプロセスはディスクから読み込んだタプルをキーの値を基にバッファに振り分け，一杯になったバッファは適切な (自ノード or 他ノード上の) ビルドプロセスのデータリストにリンクしていた．SN2 方式のプローブフェーズではディ

ストリビューションプロセスがバッファを振り分けることはなく，一杯になったバッファは必ず自ノード上のプローブプロセスのデータリストにリンクされる．一方プローブプロセスはタプルのキーの値を基に自ノード上のハッシュテーブルのみでなく他ノード上のハッシュテーブルも参照しながらプローブする (図6下)．そのためハッシュテーブルの参照はグローバルに行われることになる．

4 測定結果と考察

4.1 測定結果

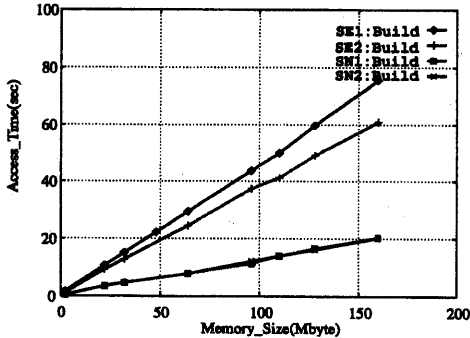

図 7:Build Time

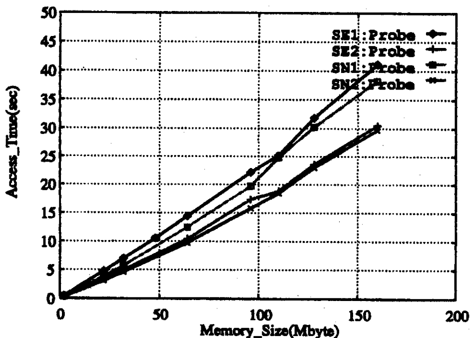

図 8:Probe Time

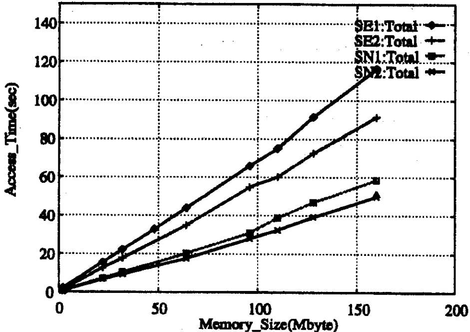

図 9:Total Time

SE，SN 方式におけるデータサイズを変化させた場合のビルドフェーズ，プローブフェーズの合計処理時間の結果を図 7,8,9 に示す．本測定では 4 ノードを用い，データサイズを 8MB ～ 640MB まで変化させた．ここではメモリアクセスストラテジーの差を確認することが目的であるため，ディスクアクセス時間は除いている．またノード間のメモリアクセスにはメモリコヒーレンスを保つためにディレクトリ情報を作成しなければならないが，その影響を軽減するため warm_start で測定を行う．全体の処理時間からもっとも性能の良い SN2 方式は SE1 方式と比較して 58% 以上の性能向上がみられ，メモリの物理的な分散を意識してプログラミングすることは効果があるといえる．以下ではビルド / プローブの各フェーズについて述べる．

4.2 ビルドフェーズ処理時間

ビルドフェーズの実行時間に関しては SN1,SN2 方式が SE1,SE2 方式より大きく性能向上がみられる (図 7).

SE1 方式ではデータバッファがグローバルに確保されているためディストリビューションプロセスがディスクから読み出したタプルを書き込む場合に他ノードメモリ上への書き込みを一定の割合で引き起こし，書き込みコストが高くなる (図 10)． SN1,SN2,SE2 方式はこの問題が無い．

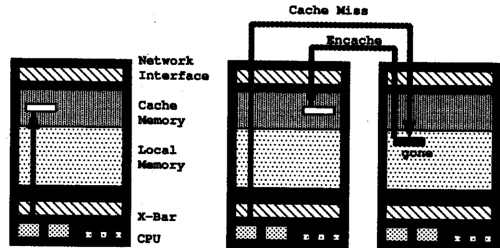

図 10:Local Write & Remote Write

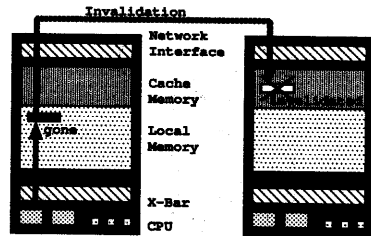

図 11:Local Write with Invalidation

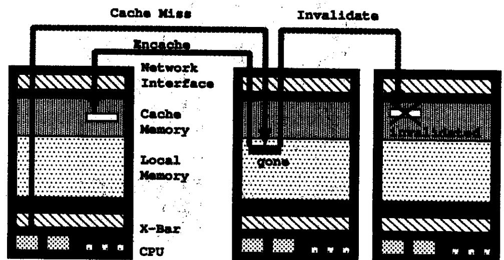

図 12:Remote Write with Invalidation

SE1,SE2 方式ではビルドプロセスはハッシュテーブル作成をグローバルな空間に行わねばならず，この際に他ノー

ド上への書き込みが生じてしまう．これは先程と同じ問題である．さらにキャッシュライン (64byte) にはテーブルエントリが 8 つ格納される大きさなので 1 つのエントリにアクセスした場合に残りの 7 エントリが既に他ノードのキャッシュに取り込まれている確率が高い．このためローカルメモリに割り付けられたテーブルエントリに対するアクセスでも残りのエントリの割り付けられたキャッシュラインに対して invalidation を引き起こし (図 11)，他ノード上に割り付けられたエントリに対するアクセスでも invalidation を伴う (図 12)．

SE1,SE2 方式ではハッシュテーブルは複数のビルドプロセスによって同時に書き込みをされるため，ロックを取りながら書き込みを行わなければならない．
自ノードへの単純な書き込みは約 0.8$\mu sec/access$，他ノードへは約 4$\mu sec/access$ である．しかしロックを伴うと自ノードへの書き込みが約 6.0$\mu sec/access$，他ノードへは約 14.5 $\mu sec/access$ にまで時間がかかる．このためテーブルへの書き込みコストが増大し処理時間が著しく増大する．

4.3 プローブフェーズ処理時間

プローブフェーズの実行時間では SE2,SN2 方式が SE1,SN1 方式と比較して性能が良い．

SE1,SN1 方式ではプローブフェーズにおいてリスト形式で管理された数枚のページを用いてディストリビューションプロセス - プローブプロセス間のデータ転送を行っている．その際ディストリビューションプロセスはローカルメモリ上に確保されたデータバッファに書き込みを行い，プローブプロセスはそのデータバッファをキャッシュに取り込んで参照している (図 13)．

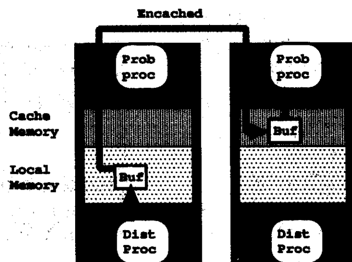

図 13: プロセス間データ転送 1

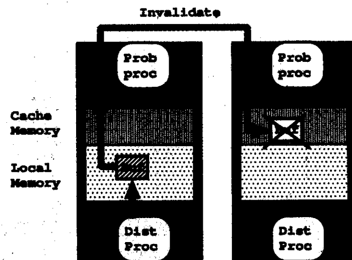

図 14: プロセス間データ転送 2

しかしそれらのページは使い回しされるため，常に他ノード上のキャッシュに取り込まれている状態にある．そのためディストリビューションプロセスは読み出したタプルを自ノード上のデータバッファへ書き込むが，その領域が他ノード上のキャッシュに取り込まれているため invalidatoin を引き起こす (図 14)．ローカルな書き込みは約 0.8$\mu sec/access$ だが，invalidation を伴った場合は約 5.8$\mu sec/access$ となり，書き込みコストが著しく増大し処理時間のネックとなる．

一方 SE2,SN2 方式ではプロセス間のデータ転送をノード内に限定している．ディストリビューションプロセスによって書き込まれたバッファは必ず同じノード上のプローブプロセスによって処理される．これによってバッファに書き込む際に他ノードへ invalidation を発行する必要が無くなった．代わってプローブプロセスはプローブを行うために他ノード上のハッシュテーブルを参照することになった．他ノード上のメモリ参照は約 4.0$\mu sec/access$ であるが，これは invalidation を伴う書き込みコスト約 5.8$\mu sec/access$ よりは軽い．つまり SE2,SN2 方式は invalidation コストを他ノード上のメモリを参照するコストで置換することで処理時間を短縮している．

4.4 キャッシュサイズが warm_start に与える影響

本測定では warm_start で測定を行ったため，メモリの一部はキャッシュに取り込まれている状態にある．そのためローカルメモリに書き込みを行った場合でもキャッシュに取り込まれているコピーに対し invalidation を引き起こす場合がある．ローカルメモリへの書き込みは約 0.8$\mu sec/access$ であるが，invalidation を伴う場合には約 5.8$\mu sec/access$ となり，書き込みコストが増大する (図 11)．また他ノード上のメモリへアクセスした場合でも invalidation を伴うことがあり，単純にアクセスした場合 (約 4$\mu sec/access$) に比較してアクセスコストが増大する (約 5.8$\mu sec/access$ 図 12)．この問題はキャッシュのサイズが大きくなるに従ってより大きな影響を持つ．

5 終わりに

本報告では分散共有メモリ計算機上での並列データベース処理を単純な共有メモリ型方式とメモリの分散を意識しアクセスの局所性を高めた方式とで実装し，アクセスの局所性を持たせることの有効性を示した．

またノード間でデータ転送を行う際に通信バッファを用いると性能が劣化するがこれをリモート参照で置換することで性能が向上することを示した．

参考文献

[1] A.Shardal and J.F.Naughton: *Using Shared Virtual Memory for Parallel Join Processing*, Proc. of SIGMOD '93, pp.119-128, 1993