EC81−15

可変構造多重処理データベースマシンにおけるソートモジュール

SORT MODULE
IN
DYNAMICALLY RECONFIGURABLE AND HIGHLLY PARALLEL DATABASE MACHINE

喜連川 優　M．KITSUREGAWA
鈴木 重信　S．SUZUKI
田中 英彦　H．TANAKA
元岡 達　T．MOTO-OKA
東京大学　工学部
UNIVERSITY OF TOKYO　FACULTY OF ENGINEERING

1. はじめに

近年、データベースの普及に伴い、ソフトウェアサポートによるDBMSの性能の限界が次第に明らかとなり、データベース管理の諸機能をハードウェアでサポートする事を試みたデータベースマシンの研究が各所でなされる様になった。[1] 我々も既にリレーショナルデータベースのサポートを中心にセルラロジックタイプのDBMをより発展させたMIMD型マシン可変構造多重処理データベースマシンの開発を行なってきた。[2,3] 図1,2に示される本マシンのデータ操作部は、多重チャネルリングバスにより多くのプロセッシングモジュール(P)群とメモリモジュール(M)群との間に柔軟な結合関係を生成し、処理負荷に応じた動的プロセッサ割付けを可能としている。又、メモリからプロセッサ群へはデータブロードキャストを行ない、メモリコンフリクトのない高次の並列処理が可能な為、駆動プロセッサ台数にほぼ比例した処理速度の向上が得られる。メモリモジュールには、メイジャマイナ方式の磁気バブルメモリを用い、マークビット管理機構によりデータ実効転送レートの向上を計っている。今回、更に処理の高速化を計る為、ソートモジュールの検討を行なったので報告する。

ソートはデータ処理の基本操作であり、ソートされるとその後の処理は多くの場合容易になる。ここではデータベースマシンへの組込みを意図しているが、ソータ自身としても当然有益と思われる。データベース管理システムの発展が目ざましいが、そこではソートは一層不可欠な機能と考えられる。リレーショナルデータベースでのジョインやプロジェクションはソートされていれば、O(N)で処理可能であるし、又一般データベースでのインデックス生成などでもソートが

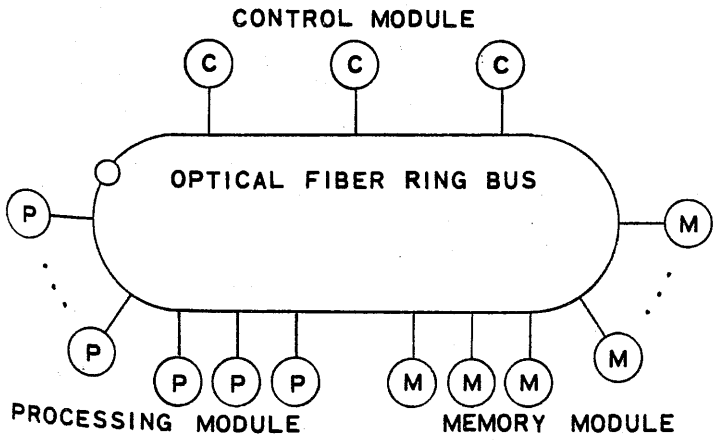

FIG.1 ARCHITECTURE OF DATA MANIPULATION UNIT

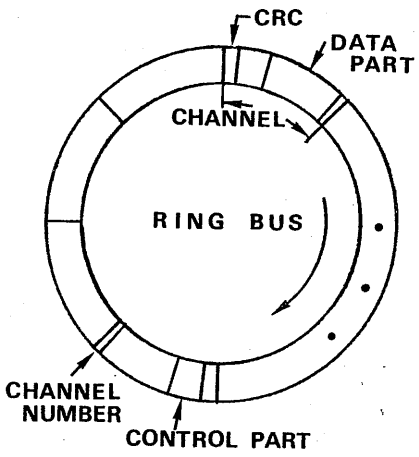

FIG.2 RING BUS DATA FORMAT

必要とされ、高速ソータの需要は大きいと考えられる。更に二次記憶からのデータレートに合ったO(N)ソータは外部記憶に於ける von Neumann Bottle Neck を解消する上でも重要である。

2. ソータ設計指針

ソータ設計に際し、次の様な点に留意した。

1) 入力データストリームに同期したソート

一般にソートはN個のレコードに対しO(N log N)の比較が必要とされ、クイックソート、トーナメントソートなど多くの効率の良いアルゴリズムが存在する。しかし、内部ソートではソートすべきデータが処理開始時点で全てコア内に存在する事を仮定している。一方実際のデータ処理応用分野では多量のデータは二次記憶装置上に存在する場合が多く主記憶にロードする時点で同時にソート出来れば理想的である。ここでは、外部記憶装置からの入力データストリームに遅れることなく同期したソートを行ない処理と入力とをオーバラップさせることによって最後のレコードの入力終了と共にソートされたレコード出力が得られる様な構成を目標とする。(O(N)ソータ)

2) レコード長に対する考慮

キー部のみを扱う場合、或はエンコードされたデータを扱う様な場合には、レコード長は短く、固定的アーキテクチャで問題はない。しかし、一方で一般レコードを扱う場合も考えるべきであり、レコード長の変化に対して柔軟に対処できる構造が好ましい。一般レコードのソートに於いてソート対象をレコード全体ではなくキー部に限定する事によってレコード長に制限を設ける事も可能であるが、パイプラインソーティングに於いてはキー部、非キー部の分離には意味がない場合が多い。即ち、キー部のみソートし非キー部はソート後結合する事も可能であるが、フィールド毎のファイルではなく、レコード単位の従来のファイル構成の場合には二次記憶からのレコード取出しレートによってプロセッサの処理能力が決定され、パイプライン処理では非キー部のデータ移動は処理負荷の増大にはつながらない。従ってレコード全体をソートする方が便利である場合も多く短いレコードのソート又でなく、一般レコードのソートも考える事とする。そしてこの場合には、レコード長の変動が激しいと考えられ、柔軟に対処し効率よくソート出来る様考慮する事とする。但し簡単の為、1ストリーム内ではレコード長一定とする。これは可変構造多重処理データベースマシンでは1リレーション内でタプル長固定としている事による。

3) 半導体RAMチップの利用

LSI化を考える場合、Kung らの Systolic Array[6,7] の如くロジックとメモリセルの一体化が自然であるが、より柔軟な制御構造を入れようとすると1つ1つのセルが複雑なものとなってしまう。セルラロジックではセルの簡素化と構造の柔軟性とは相反する要素と考えられる。ここでは $\log_K N$ 台 (N: ソートすべきレコード数, K: K way sort) のプロセッサと大容量半導体RAMバンクによる構成とした。O(N)ソータの最も簡単な手法はN台のプロセッサを用いる事であるが、ここでは $\log_K N$ 台にし、プロセッサオーバヘッドを少なくした。又半導体RAM技術は一層進歩すると考えられ、RAMチップを利用する事とした。

3. ソートアルゴリズム

3-1 アルゴリズム

ソートのアルゴリズムはマージソートのパイプライン化であり、8),9)によって既に知られている。

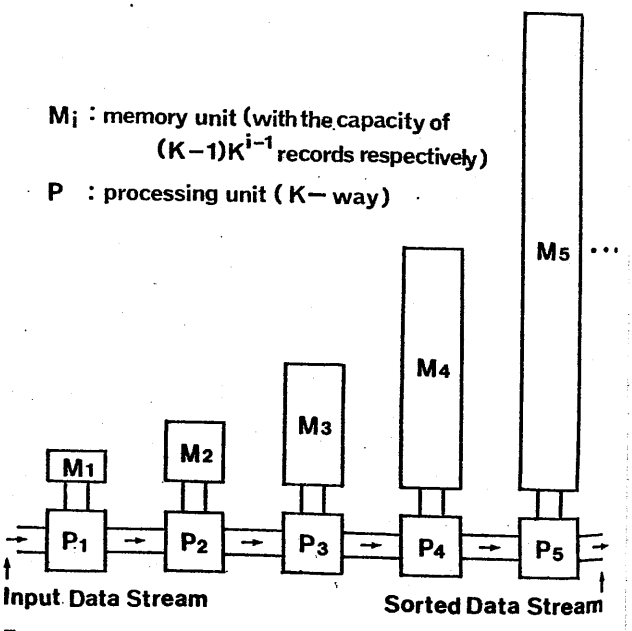

Fig. 3 Global Architecture Of Stream Driven Sorter

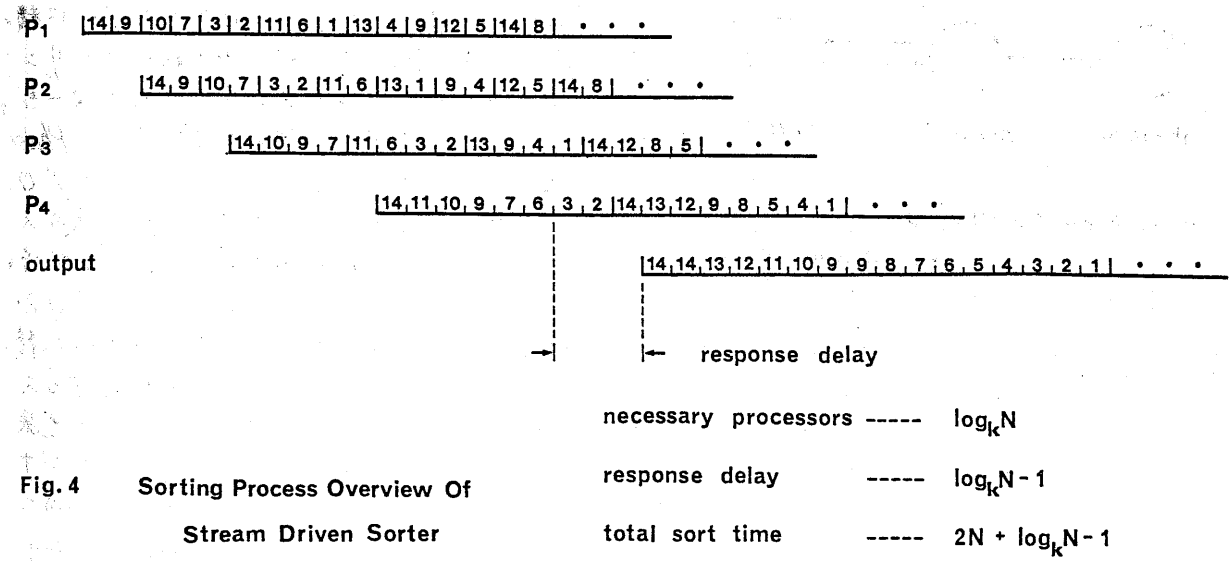

Fig. 4 Sorting Process Overview Of Stream Driven Sorter

今、$N(=K^l)$ヶのレコードのソートを行なうものとすると、K-way mergeを行なうプロセッサをl台用意し一次元状に結合する。この際i番目のプロセッサには$(K-1)K^{i-1}$レコード分のメモリを付加する。全体の構成は図3に示される。Nヶのレコードはシリアルに第一番目のプロセッサに入力され、第i番目のプロセッサは第$i-1$番目のプロセッサから送られてくるK^{i-1}レコードからなるソートされたストリングをK本マージしてK^iレコードからなる1本の長いストリングを生成し、第$i+1$番目のプロセッサへ出力する。図4の如く、各プロセッサはマージすべきK本のストリングに関して、第K番目のストリングの最初のレコードが到着した時点でマージ処理を開始できる。パイプラインのセグメントは簡単の為、上図ではレコードとしているが、これはバイトレベル、ビットレベルにまで落とす事が可能である。(我々はバイトレベルを採用)

3-2 諸特徴量

- 入力レコードストリームが終了してから、ソートされたレコード出力が可能になるまでの遅れ

$$\log_K N - 1$$

これは最終レコードが直列に結合されたプロセッサ間を通り、最終段のプロセッサまで到達するのに必要な時間に対応している。この様に遅れは極めて僅かといえる。

- 入力レコード全体のソートに必要とされる時間

$$2 \cdot N + \log_K N - 1$$

本アルゴリズムでは全レコードをソータに一度入れ、その後とり出す為、最低$2 \cdot N$時間は必要となる。(昇順ソートで、最小レコードが一番最後にやってくるかも知れない為)
時間的オーバヘッドは上で述べた$\log_K N - 1$だけである。

- 所要メモリ容量

$$N$$

各段でプロセッサはそれぞれ、K^iレコードからなるストリングを$K-1$本貯え、最後の1本の到着とともにマージを開始する。それ以後は1レコードの入力とともに1レコード出力する為、各段では$(K-1) \cdot K^i$レコード分のメモリ容量が必要となり、全体としては、$N(=K^l)$即ち、全レコード分のメモリ容量があれば良い事がわかる。

3-3 プロセッサ内処理

プロセッサ内での処理は、基本的には既に述べた如く前段からのK本のストリングをマージして1本にし次段へ出力する事であるが、より細く見ると以下の如く3つのPhaseに分かれる。

- Phase 0 ……最初のストリングロードの時であり、$(K-1) \cdot K^{i-1}$ヶのレコードを当該プロセッサ内メモリに貯えるPhase。次段のプロセッサへはまだデータを出力しない。$K-1$本のストリング入力後Phase 1へ移る。
- Phase 1 …… $(K-1)$本のメモリ内ストリングと入力されてくるK番目のストリングをマージして出力するPhase。当該ストリングの入力が終了するとPhase 2へ移る。
- Phase 2 …… Phase 1で当該ストリングの入

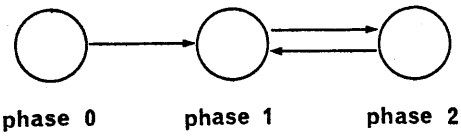

Fig. 5 Phase Transition Of Stream Driven Sorter

力が終わると、次の新しいストリングの入力が始まる。しかし、この段階では前のK本のストリングマージは完結しておらず、K本合わせて(K−1)・K^{i-1}レコードがメモリ内に残っている。Phase 2では、この残っているK本のストリングマージを行ないつつ、次のストリングの入力を行なう。K−1本のストリングを入力するとマージ処理は完了し、Phase 1へ移る。(図5)

4. ソータの実装方式

4−1 メモリ管理

i番目のプロセッサは K−1本のストリングを入力した後、K番目のストリングに対し、1レコード入力しては比較し、1レコード出力する為、常に有効なメモリ容量は一定である。ところが、マージ処理時には、ストリングの先頭を取る順番はランダムであり、当該レコードの出力によって空となった所に順次入力していくのでは、ソートされた順序を維持する事は出来ず何等かのレコード管理機能が必要となる。これに対し、ここでは以下の如き3つの手法を検討した。(以下簡単の為2−way sortを仮定する。)

4−2 Double Memory Method

2−wayの場合、最も簡単な手法は2倍の容量を用意し、マージすべき2つのストリングに対し、各々のエリアを割り当てる事である。(図6)

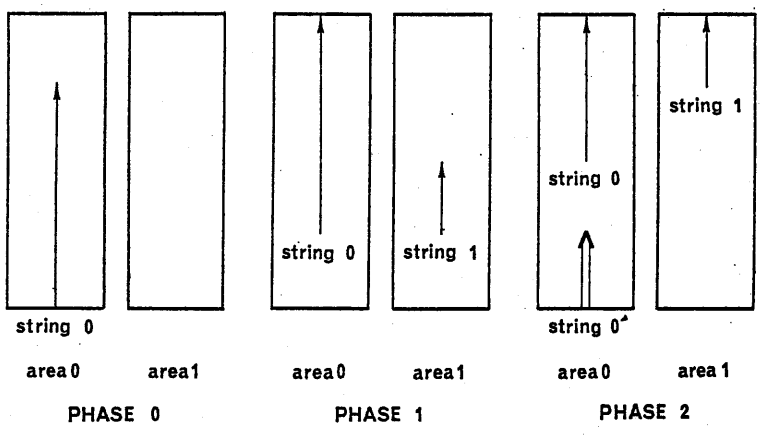

Fig. 6 Sorting Process In Double Memory Method

Phase 0では、ストリング0をエリア0に積み上げてゆき、Phase 1では入力されるストリング1をエリア0に存するストリング0とマージするとともにエリア1に積み上げてゆく。Phase 2ではエリア0,エリア1に残ったストリング0及び、ストリング1をマージすると共に、次のストリング0'をエリア0に積み上げる。いかなる時点に於いても、ストリング0, 1, 0'の合計は2^{i-1}ケのレコードから成り、従ってエリア0に積み上げる事によってストリング0が残存するストリング0に上書きする事はない。即ち、元来、ある時点でプロセッサ内には2つのマージすべきストリング及び、次ストリングの3つのストリングが混在する事になるが、2本分のエリアで足りるのである。この手法は制御が大変簡明ではあるが、i段目では 2^iケの容量を有し、iが大きくなると50%の無駄は無視出来ない。

4−3 Pointer Method

各レコードにポインタ領域を持たせ、ポインタによってレコード間のシーケンスを維持する手法。この手法はレコード部に対してポインタ部のオーバヘッドが無視出来る場合には、制御が容易で有効である。

4−4 Block Division Method

Double Memory Method と Pointer Method の中間策であり、メモリを固定長のブロックに分割する。ブロック内ではデータはソートされており、ブロック間の前後関係を別の構造として持たせるブロック単位のメモリ管理手法である。この手法に関して次の事実が成立する。

「当該プロセッサに必要とされるメモリ領域をlケの等しいブロックに分割すると、l+2ケのブロック分のメモリ領域を用意する事によって、1ブロック分のレコード入力後、必ず1つの空ブロックが生成される。」

この事実を利用し、以下の如く実行する。Phase 0でストリング0をlケのブロックに積み上げた後、Phase 1では、入力されるストリング1をストリング0とマージしながら適当に選んだ空ブロックに積み上げていく。ストリング1を1ブロック入力し終える毎に

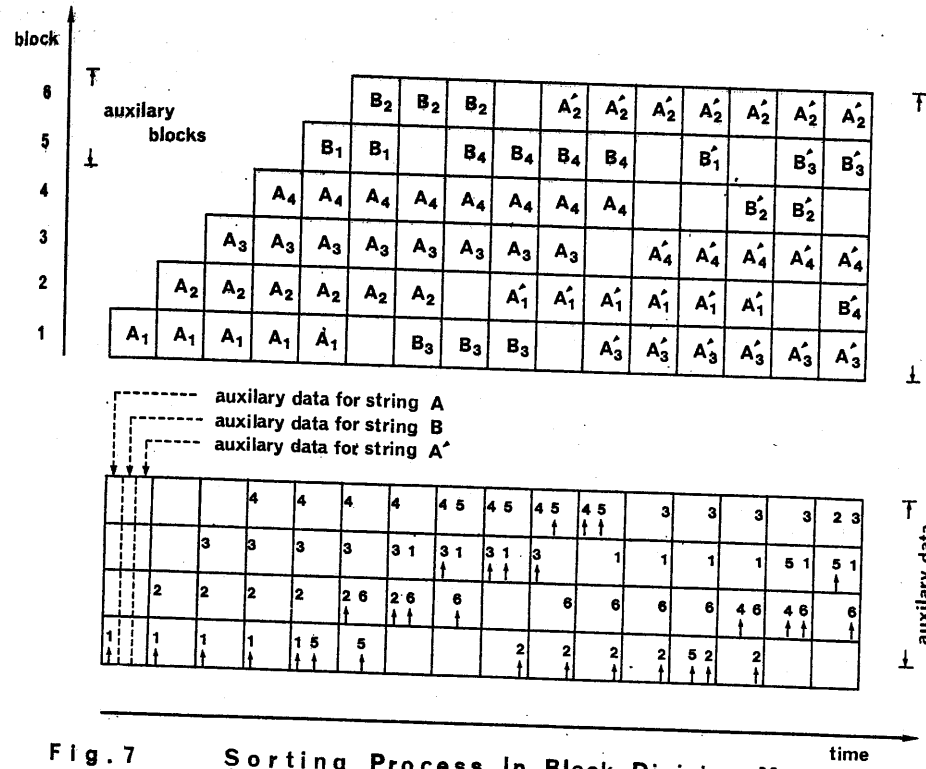

Fig.7 Sorting Process In Block Division Method (Block Fetch Sequence)

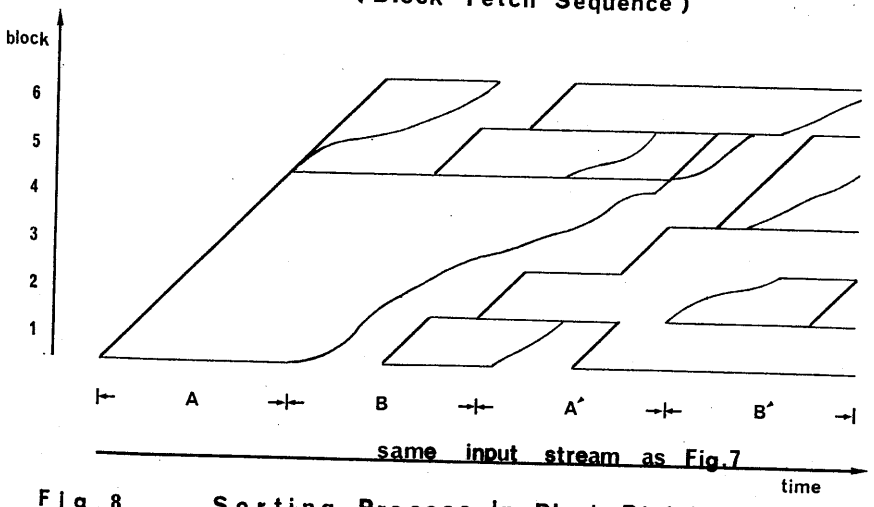

Fig.8 Sorting Process In Block Division Method (String Top Position Trace)

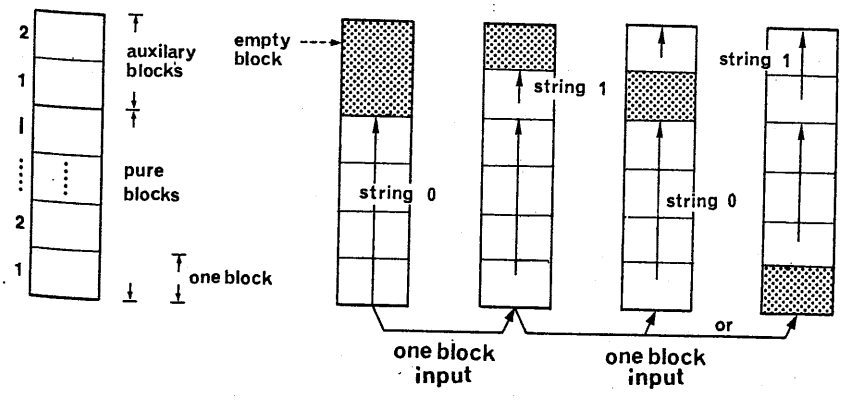

Fig.9 Block Division Method. Empty Block Generation After Each Block Input

先の事実から少なくとも1つ空ブロックの存在する事が保証されており、そこへ入力して行く事が出来る。Phase 2でも同様にストリング'0'を1ブロック入力する毎に、空ブロックが生成され、ストリング0と1のマージと並行してストリング0'の貯蓄がなされる。この際、空ブロックの生成される順番はデータに依存しランダムであるから、ブロック間のシーケンスは補助的データによって維持されているものとする。(図7,8,9)

さて、前記の事実は、以下の点から明らかである。即ち、もし、1つの空ブロックもないとすると、そもそもメモリ内には、完全につまっていないブロックは、高々1つしか許されないから、少なくともlブロック分のレコードと、それに加えてlレコード分必要となってしまう。これは各段での必要とされる一定のメモリ容量と一致しない。

● ブロック管理用補助データの構造

Block Division Method ではブロック間の順序を維持する為の補助的データが必要となる。このブロックレベルでのシーケンス維持に対しても、今まで述べて来たレコードレベルでの手法を再び適用する事が可能である。即ち補助データの補助データを考える事が出来る。しかしこのデータ容量は元来、真のデータに比べると僅かな量であるから、ここでは容量を節約するよりもむしろ簡単な構造を持たせる方が賢明である。即ち、Double Memory か又は All Pointer Method が好ましい。さて All Pointer Method はそもそもレコード長に比べてポインタ長が短い場合に有効な手法と考えられるが今回は、実レコードがブロックNo.でありポインタ長はそれに等しくなる。依って必要なメモリは両者大差なく、ここではむしろ Double Memory Method を採用する事が最も容易と考えられる。

4－5 K－wayへの拡張

これまでは 2－way Sorter の場合について説明して来たが、これは容易に K－way に拡張することが出来る。Kを大きくする事により、1つのプロセッサはより複雑となるが、一定容量を実現する為のプロセッサ数、メモリバンク数は少なくてすみ、又遅延時間をわずかに減少出来る。逆にメモリ容量は 2－way の場合に比べて一台増す毎に大きく変化するが、これに関しては最終段のプロセッサが $2 \leq K' \leq K$ とし、そのパワーを落として調整することによって適切な容量を実現できる。

K－way でのメモリ管理方式は次の様に変更される。

Double Memory Method

K本のストリングをマージする K－way ソータでは、$2\times(K-1)$本のストリング容量のメモリエリアを用いる事によりメモリ管理可能である。

Block Division Method

K－way ソータでは、当該プロセッサの1本のストリングを ℓ ヶの等しいブロックに分割すると、$(K-1)\cdot\ell+K$ ブロック分のメモリ領域を用意する事により、1ブロック分のレコード入力後、必ず1つの空ブロックを生成できる。

4－6 Block Division Method に於ける補助メモリのコスト評価

Block Division Method に於いては真に必要なメモリ容量の他に K ヶのブロック及びブロック間順序を維持する為の補助メモリが必要となる。既に述べた如く、補助メモリとしては、Double Memory Method を利用する事にする。ここでは、各段の補助メモリコストの評価式を求め、最適ブロック分割を行なうとともに、真のデータに対するコスト比を計算する。尚ブロック分割数、way 数 K は簡単の為 2 のべき乗とする。i 番目のプロセッサについて考えると、補助メモリのコストは以下の如く表わされる。

• K－way	$K = 2^a$
• レコード長	M (bit)
• 真に必要なメモリ容量（レコード数）	$(K-1)\cdot K^{i-1}$ $= (2^a-1)2^{a(i-1)}$
• ブロック分割数	$\ell = 2^b$
• 1ブロックの大きさ	$K^{i-1}/\ell = 2^{a(i-1)-b}$
• 補助ブロック数	K
• 補助ブロックの大きさ 以上より	$K^{i-1}/\ell \times K = 2^{ai-b}$
► 全実データブロックの大きさ	$(K-1)\ell + K$ (ブロック) $= (2^a-1)\cdot 2^{a(i-1)} + 2^{ai-b}$ (レコード) $= M\times[(2^a-1)\cdot 2^{a(i-1)}+2^{ai-b}]$ (bit)
• 実データブロック数	$(K-1)\ell + K$ $= 2^{a+b} + 2^a - 2^b$
• 補助データ用ブロックポインタ長	$\lceil \log_2\{(K-1)\cdot\ell+K\}\rceil$ (bit) $\begin{cases} = a+b & (a\leq b) \\ = a+b+1 & (a>b) \end{cases}$
• 補助データ用メモリエリア数 (Double Memory Method)	$2\cdot(K-1)$ $= 2\cdot(2^a-1)$
• 1エリア内レコード数 以上より	$\ell = 2^b$
► 補助データ容量	$\lceil \log_2\{(K-1)\ell+K\}\rceil\cdot 2(K-1)\ell$ $\begin{cases} = (a+b)\cdot 2(2^a-1)\cdot 2^b & (a\leq b) \\ = (a+b+1)\cdot 2(2^a-1)\cdot 2^b & (a>b) \end{cases}$
► 全メモリ容量	$C(M,K,\ell,i)$ (bit) $= C(M,a,b,i)$ $= M\cdot\{(2^a-1)\cdot 2^{a(i-1)}+2^{ai-b}\}$ $\begin{cases} + (a+b)\cdot 2(2^a-1)\cdot 2^b & (a\leq b) \\ + (a+b+1)\cdot 2(2^a-1)\cdot 2^b & (a>b) \end{cases}$

► 補助メモリの全メモリ容量に対するコスト比 $R(M,a,b,i) = C(M,a,b,i)/M(2^a-1)\cdot 2^{a(i-1)} - 1$

今、レコード長M.及びKが与えられると各段で$R(M,a,b,i)$を最小にするブロック分割数ℓを求めることが出来る。図10, 11はZ-wayの場合について各段でブロック容量を変化させたときのコスト比の値と、その際求められた最適ブロック分割数及びブロック容量を示している。ここでレコード長は8 byteとした。キーのエンコードを行なう応用分野では、この程度の長さで充分と考えられるが、Pointer Methodで実現しようとすると、この様に短いレコードの場合には、そのオーバヘッドは無視出来ない。一方Block Division Methodでは図12に示す如く、16段のソータで1～2%程度のオーバヘッドに押さえられる事がわかる。

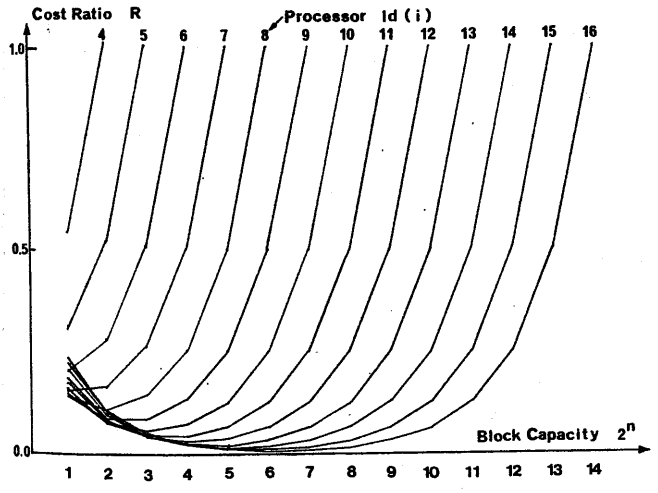

Fig.10 Auxilary Memory Cost Ratio For Each Block Capacity

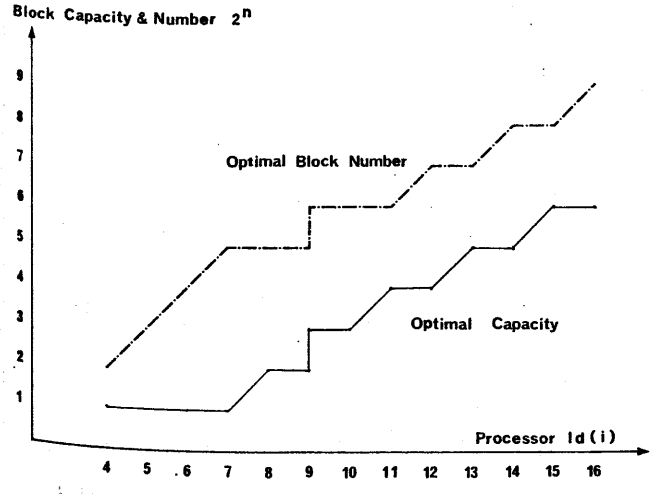

Fig.11 Optimal Block Capacity & Number

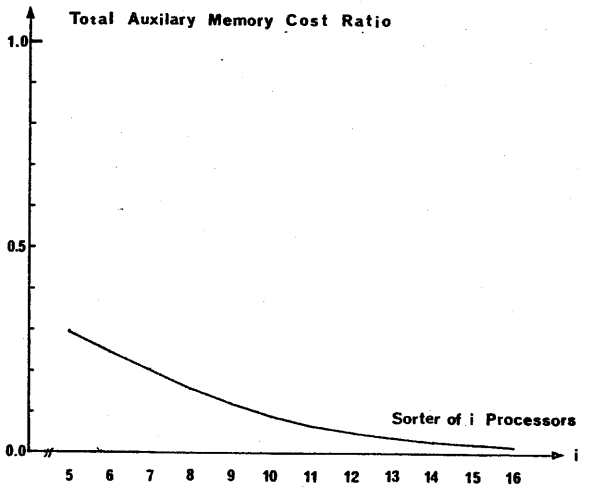

Fig.12 Total Auxilary Memory Cost Ratio For Optimal Stream Driven Sorter of i Processors

5. ソータの柔軟性

本節では、ソータの使用される環境及びその下で有すべき柔軟性について検討する。

5-1 レコード長とメモリ管理方式

a) レコード長が短い場合

キー部のみを高速にソートする場合、又それがエンコードされている様な場合には、1タプルはかなり短いと考えられる。(数バイト) 従って Pointer Method を用いるとポインタ部だけでも2～3バイト必要となり無視出来ない。この場合には Block Division Method を採用する事により効率化が図れる。この方式はブロックを大きくした極限で Double Memory Method に対応し、逆にブロックサイズを小さくした極限で1ブロックが1レコードに対応し、Pointer Method となる。

b) レコード長が長い場合

一般に、データの格納はレコード単位に行なわれ、フィールド分割は行なっていない。この様な場合、既に述べた如くパイプラインソータでは時間的ロスがない為、キー部の分離には意味がなく、フルレコードを直接ソートした方が便利である。レコード長は、かなり長くなると考えられ、ポインタ部の占める割合は僅かである為、制御の容易さから Pointer Method で良い。尚この場合には、レコード長の変化に対して効率よく対処する必要があり、次節以下に述べる。

5-2 レコード長変動に対する効率化 (String Length Tuning)

K-wayソータをレコード長(L)に対して設

計した時、実際のレコード長X(≧L)の大小によらず、高いメモリ使用効率を保ち、出来る限り多くのデータをソートする為には如何なる構成にすればよいかがここでの問題である。第一番目のプロセッサについて高々K·L長のストリングを出力するとすれば、実際のレコード長Xに対し、メモリ効率 η は

i) $X \leq L$ (短い時)

$$\eta_K = \frac{X}{L}$$

ii) $K \cdot L \geq X > L$ (長い時)

$$\eta_K = \frac{\left[\frac{K \cdot L}{X}\right] \cdot X}{K \cdot L}$$

と表わされる。

今、実際のレコード長が設計値よりも長い場合を考えると $y = K \cdot L / X$ とおくことにより $\eta = [y]/y$ と表わされ、最低50%のメモリ使用効率となることがわかる。平均使用効率 $\bar{\eta}_K$ は

$$\bar{\eta}_K = \frac{1}{(K-1)L} \int_L^{K \cdot L} \frac{\left[\frac{K \cdot L}{X}\right] X}{K \cdot L} dX$$

$$= \frac{K}{K-1} \sum_{i=1}^{K-1} \int_i^{i+1} \frac{i}{y^3} dy \qquad y = \frac{K \cdot L}{X}$$

$$= \frac{K}{2(K-1)} \sum_{i=1}^{K-1} \left(\frac{1}{i} - \frac{i}{(i+1)^2} \right)$$

例えば、$\bar{\eta}_2 = 0.75$, $\bar{\eta}_4 = 0.78$, $\bar{\eta}_{16} = 0.81$ を得る。

さて、もレストリングの長さ(1ストリングを構成するレコード数)を変える事が出来れば、原理的には次段のプロセッサは $K[K \cdot L/X]X$ 丈の長さを有するストリングの代わりに $[K^2 L/X] \cdot X$ 丈の長さのストリングを出力出来る筈であり、メモリ効率は

$$\eta_K^1 = \frac{\left[\frac{K^2 \cdot L}{X}\right] \cdot X}{K^2 \cdot L}$$

を得る。

先と同様に 平均効率を求めると

$$\bar{\eta}_K^1 = \frac{1}{(K-1) \cdot L} \int_L^{K \cdot L} \frac{\left[\frac{K^2 \cdot L}{X}\right] X}{K^2 \cdot L} dX$$

$$= \frac{K^2}{2(K-1)} \sum_{i=K}^{K^2-1} \left(\frac{1}{i} - \frac{i}{(i+1)^2} \right)$$

となる。

例えば、$\bar{\eta}_2' = 0.84$, $\bar{\eta}_4' = 0.92$, $\bar{\eta}_{16}' = 0.98$ を得る。実際、

$$K \cdot \left[\frac{K \cdot L}{X}\right] \leq \left[\frac{K^2 L}{X}\right] < K\left\{1 + \left[\frac{K \cdot L}{X}\right]\right\}$$

$$\left[\frac{K^2 L}{X}\right] = \ell_1 \left[\frac{K \cdot L}{X}\right] + \ell_2 \left\{\left[\frac{K \cdot L}{X}\right] + 1\right\}$$

$$K = \ell_1 + \ell_2$$

と表わす事が出来、プロセッサはK本のストリングを次プロセッサに出力するが、この際、ℓ_1 回は $\left[\frac{K \cdot L}{X}\right]$ 丈の長さのストリングを、ℓ_2 回はもう1レコード長くしたストリングを送出すればよい事がわかる。この様にストリング長の増加は ℓ_2 回にわたってなされる為、1回のマージで必要とされる余分の領域は1レコード分だけですむ。即ち、必要とされる容量は「」(t切り上げ)を用いて

$$\max_X \left[\frac{\left[\frac{\left[\frac{K^2 \cdot L}{X}\right]}{K}\right] \cdot X}{L} \right] = \frac{2 \cdot K^2}{K+1} \cdot L$$

と表わされ、従って、実際

$$\frac{K^2}{K+1} L - (K-1)L = \frac{L}{K+1}$$

丈のわずかな増加ですむ事がわかる。

この様に第一番目のプロセッサだけが、ストリング長の調整をする場合でも(1次 Tuning)メモリ効率は可成り改善されたが更に2番目のプロセッサもストリング調整を行なうとすると

$$\left[\frac{K^3 L}{X}\right] = m_1 \left[\frac{K^2 \cdot L}{X}\right] + m_2 \left\{\left[\frac{K^2 L}{X}\right] + 1\right\}$$

$$= m_1 \left\{\ell_1 \left[\frac{K \cdot L}{X}\right] + \ell_2 \left(\left[\frac{K \cdot L}{X}\right] + 1\right)\right\}$$

$$+ m_2 \left\{(\ell_1 - 1)\left[\frac{K \cdot L}{X}\right] + (\ell_2 + 1)\left(\left[\frac{K \cdot L}{X}\right] + 1\right)\right\}$$

と表わされ (ℓ_1, ℓ_2) を m_1 回実行後 $(\ell_1 - 1, \ell_2 + 1)$ を m_2 回実行することで、$\eta = \left[\frac{K^3 L}{X}\right] \cdot X / K^3 L$ を得ることが出来る。2段目以後のプロセッサは、k-way merge を行ない初段のプロセッサがK以下でマージ数を変える事によって全体の調整をすることになる。

以上の方式を一般化する事により更に、高次の調整が可能となる。即ち、

$$\left[\frac{K^i \cdot L}{X}\right] = \ell_1^{i-1} \left[\frac{K^{i-1} \cdot L}{X}\right] + \ell_2^{i-1} \left\{1 + \left[\frac{K^{i-1} \cdot L}{X}\right]\right\}$$

$$\vdots \qquad\qquad \vdots$$

$$\left[\frac{K^3 \cdot L}{X}\right] = \ell_1^2 \left[\frac{K^2 \cdot L}{X}\right] + \ell_2^2 \left\{1 + \left[\frac{K^2 \cdot L}{X}\right]\right\}$$

$$\left[\frac{K^2 \cdot L}{X}\right] = \ell_1' \left[\frac{K \cdot L}{X}\right] + \ell_2' \left\{1 + \left[\frac{K \cdot L}{X}\right]\right\}$$

と表わされ、$i-1$ 番目までのプロセッサが、ストリング長を調整することにより、

$$\eta_K^{i-1} = \frac{\left[\frac{K^i \cdot L}{X}\right] \cdot X}{K^i \cdot L}$$

を得る。

又平均値は

$$\bar{\gamma}_K^{i-1} = \frac{K^i}{2(K-1)} \cdot \sum_{j=K^{i-1}}^{K^i-1} \left(\frac{1}{j} - \frac{j}{(j+1)^2} \right)$$

となる。これを図13に示す。

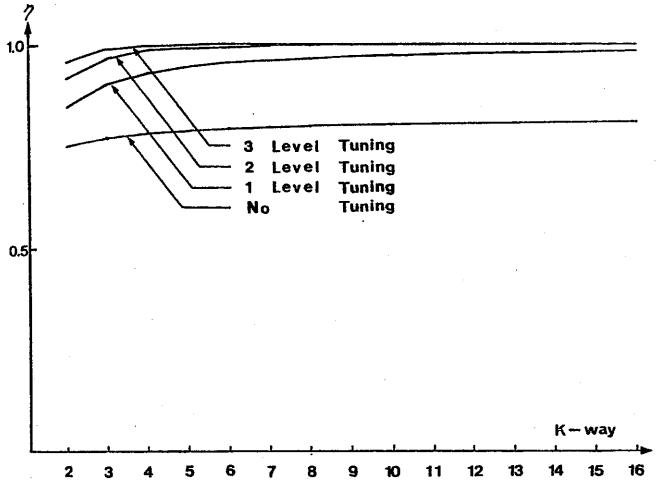

Fig.13 String Length Tuning & Memory Utilization Efficiency

以上の如く、レコード長Lに対して設計したソータにより長さの異なるレコードを取り扱おうとすると、そのままの形ではメモリ使用効率が低くなってしまう。初段での効率が次段へと伝搬してゆき、全体としてのメモリ効率は初段のそれによって決定されるからである。これに対し、ここでは始めの数台のプロセッサに余分のメモリを少々加え、ストリングの長さを調整することにより後段でのメモリ効率を高めた。メモリ容量は段を増すに従って急増し、メモリ効率は最終段でほぼ決定される為、最初のプロセッサのメモリ増量は問題にならない。又、実レコード長の範囲を、$L < X \leq K \cdot L$としたが、$X > K \cdot L$の場合には、2番目のプロセッサが初段として振舞うことにより、同様の制御が可能である。逆に、実レコード長の方が短い場合には元来、$\log_K N$台のプロセッサが必要であるにもかかわらず、現実には$\log_K N$台しかなく、そもそもプロセッシングパワーが足りない為効率が落ちる。従って、短いレコードも効率よく処理する為には、余分のプロセッサを先頭に付加すればよい。この様な方式により、任意のレコード長に対して、そのゆらぎを吸収し効率よくソート出来る事がわかる。

5－3 ストリーム長変動に関して

m台のプロセッサからなるK－way Sorterでは、K^mケのレコードのソートが可能となるが、これより短いストリームの場合、最も簡単な手法は、ダミーレコードをストリームの最後に付加し、K^mケのとしてソートする事である。しかし、これではソータの容量が増大するにつれて、短いストリームに対する出力遅れが甚だしくなる。そこでこれに対しては次の手法を採った。

最初のプロセッサには、最終レコードの先頭にストリングエンドのフラグ(eos)を付加して入力し、以後のプロセッサは、ストリーム内の最終ストリングの先頭レコードにeosを付加して出力する事とする。又、プロセッサはeosの検出により、マージを開始する。この様にするとeosフラグはプロセッサを介する毎にストリーム内での相対位置が前方にずれて来る事になる。そして最後には、eosがストリームの先頭SOS(ストリームスタートフラグ)に一致し、それ以後のプロセッサはメモリを介さずに次段のプロセッサへバイパスさせればよい。こうする事により遅れを最小化できる事がわかる。尚eos, SOSの他に、最終ストリングの最終レコードを識別する為のEORフラグも必要である。これらのフラグはプロセッサ間での通信情報であり、例えば、ポインタ方式によりプロセッサを構成したとすると、レコード毎にポインタを操作する為のオーバヘッドタイムが生ずるが、この時間内に埋め込む事が可能である。

6. ソートモジュールとデータベースマシンに於けるその位置付け

本節ではこれまで述べて来たソータのデータベースマシンの中での位置付けについて述べる。ソータを用いたソートモジュール内での処理及びソートモジュールの駆動環境の2つの立場から以下説明する。特に後者ではHashとソートの利用により効率のよいO(N)処理方式の概容が示される。(図14)

6－1 ソートモジュール内での関係代数処理

a) Projection

重複除去は、処理負荷の重いオペレーションであり、SQLなど陽に指定しないと実行しないシステムも多い。ソータを用いる事によってO(N)で処理出来る事は明らかである。

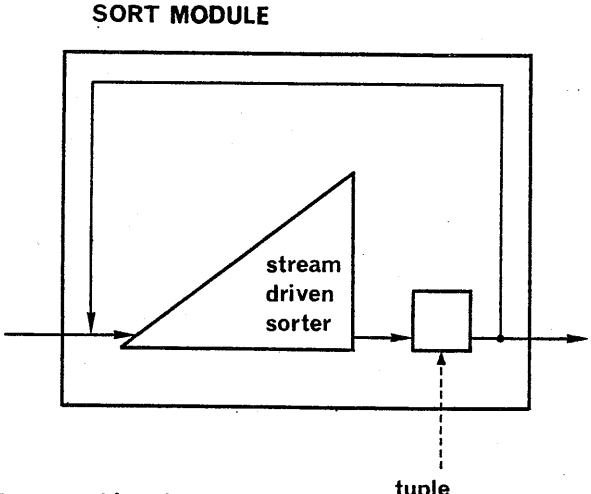

Fig.14 Global Architecture Of Sort Module

b) Join

Joinは直接実行しようとすると両リレーションのカーディナリティの積に比例した処理時間が必要とされるが($O(N_1 \times N_2)$)、両リレーションがソートされていれば、略$O(N_1+N_2)$で実現出来る。Joinは基本的にはソータを2つ用意し、各々に2つのリレーションを割当て、それをマージすればよいが、実際には、ソータを2つ用意する事は、ソータ内で一段プロセッサを増加する事に等しく、一台のソータにより、Joinを実現する事が望ましい。これはレコードのキーと非キー部の間にリレーションid(0or1)を挿入してソートする事によって解決する。又Join処理はマージと変わらないが、直積をとる際バッファオーバフローが生ずる可能性がある。即ち、ソータ出力の2つのリレーションをマージし、直積をとる部位のバッファ内にいずれか一方のリレーションの同一ジョインキーを有するタプル集合が入いり切る場合には問題はないが、入いり切らない場合の方策を考える必要がある。本ソータではソートストリーム出力時にも内部処理が施されており、一端出力したデータは、再び利用する事は出来ない。ここではバッファオーバフローに対し処理対象として残すべきタプルを再びソータに入力し、2回目のマージ時に処理する事とした。尚、処理済のタプルは捨てられる為2回目のスキャンでは不要なタプルはかなり篩い落とされるが、オーバフローし再入力したにもかかわらずJoin出来ないタプルに対しては、リレーションidを反転する事によって捨て去る事が出来る。(図15,16)

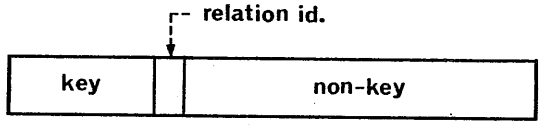

Fig.15 Modified Tuple Format

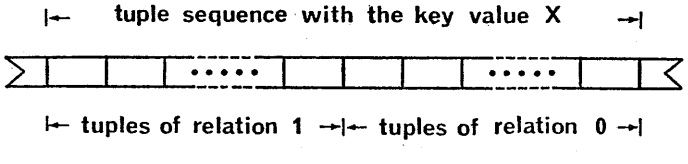

Fig.16 Output Tuple Sequence Of Stream Driven Sorter

以上の如く、Projection, Joinはソータにより、高速処理可能であることがわかる。その他、集合演算操作 Intersection や Difference 更に、Divisionなど、処理負荷の重いオペレーションも同様に1つのソートモジュール内の処理に関しては高速に実行可能である。

6-2 Hashとソートによる関係代数処理

リレーション全体をソータのサイズのブロックに分割し、ソータ内では既に述べた如くO(N)で、ブロック間では$O(n^2)$で処理する事も可能であるが、ここではHashを利用する方式を考案したので、その概略を述べる。Hashの利用は10),11)らによって既に指適されているが、これらはJoinやProjectionの候補を篩い出す前処理として利用するもので、実際の演算自体を高速化するものではない。ここでの方式はHashによるクラスタリング機能を利用するものである。Joinの場合、両リレーションにHashを施し、各々全体を幾つかのバケツに分割すると、結合のとれるタプルは互いに同じ番号を持つバケツのタプル同志に限られ、異なる番号のバケツ同志のマッチングは必要なくなる。この様にリレーションを幾つかの小さいサブリレーションに分ける際、単に分割するだけでは、そのレベルで$O(n^2)$の結合処理が必要となるが、Hashを用いるとそのクラスタリング特性から$O(n)$ですむ。そしてバケツ内でのマッチング処理は本ソータを用いれば、O(N)で処理が可能となり、従って全体として、O(N)のJoinを実現出来る事になる。処理手順は以下の如くなる。リレーションはディスク上に貯えられているものとし、Restriction Filter Processorを通して、必要なタプルのみが磁気バブル

メモリからなるメモリモジュールにステージングされる。この際、メモリモジュールはジョインアトリビュートに関し、Hashを施し、動的にバケツidを生成すると共にマークビットによりこれを管理する。ステージングが終了するとメモリモジュールはバケット毎にタプルを送出する。改良されたM/m方式磁気バブルメモリを用いる事により特定のマークが施されたタプルのみを高速に送出する事が可能となる。[5] この出力をソートモジュールが入力するが、当該バケツの出力が完了した時点で、ソータはソート出力可能となり処理が開始される。一方メモリは次のバケツの出力を始めるが、これは別のソートモジュールに割り付けられる。この様にしてバケツシリアルに処理を進めてゆく事が出来る。バケツの大きさはスイッチングオーバヘッドによって決定され、又大きさのゆらぎに関してはオーバフローの生じない様 Bucket Size Tuning を行ない、プロセッサ使用効率を高めている。

以上、1つのオペレーションとしてO(N)の高速処理が期待出来る事が判るが、更にQuery Tree上オペレータ間でもパイプライン可能である。即ち、当該演算処理中の出力タプルはメモリモジュールに送出される毎にHashingが施され処理フェーズと次オペレーションの準備のHashingフェーズとを重畳する事が出来る。

この様にソータレベルの低次のパイプラインから関係オペレータレベルの高次のパイプラインまで種々のレベルの並列性を効果的に抽出する事により本マシンは高い処理能力を発揮すると考えられる。ここではバケツシリアルな処理方式について述べたが、バケツパラレルな方式[4]及び、パイプラインの擾乱効果、性能評価等、詳細は、別の機会に譲る。

7. おわりに

マージソートのパイプライン化を行なうStream Driven Sorterについて、メモリ管理方式及びその構成法に関する検討を行なった。現在ポインタ方式によるソータのLSI化を考えている。プロセッサ間は8bit巾で結合され、ポインタ部は2バイト長である。レコードの1バイト処理は2回のメモリアクセスにより実現され1M byte/sec程度の入力レートに追従出来ると考えられる。プロセッサ内レジスタ容量に関しては、レコード全体を入れようとすると、レコード長の最大値を制限する事になり、好ましくない為、キー部のみをレジスタ内に置き非キー部はメモリに入れてポインタにより管理する事とした。キー部もメモリに入れ、キー長の制限を取る事も可能であるが、メモリアクセスが増加し処理速度が低下する。非キー部は比較が不要な為余分のメモリアクセスを伴わずメモリに置いても速度が低下する事はない。

又Hashとソートによるデータベースマシンは、Hashによるクラスタリング機能とバケット内ソート処理を結合し、処理負荷の重い関係代数演算子をO(N)で処理出来る事が示された。更に本マシンは種々のレベルの並列性を抽出しており、従来のマシンに比べ高い性能が期待出来る。

参考文献

1) 植村, 前川著 "データベースマシン" 情報処理叢書1 情報処理学会 1980
2) 喜連川他 "可変構造多重処理データベースマシンの構成" 信学技報, EC80-51, 1980
3) 喜連川他 "多重チャネルリングバスに於けるブロードキャスト伝送制御手順" 情報処理学会分散処理研究会5-2, 1980
4) 喜連川他 "可変構造多重処理データベースマシンのシステム構成, ソートモジュール" 情報処理学22回全国大会, 3L-3, 1981
5) 鈴木他 "可変構造多重処理データベースマシンのメモリモジュール" 全上 3L-4
6) H.T. Kung et al "Systolic (VLSI) Arrays for Relational Data Base Operation" Proc ACM SIGMOD 1980
7) H.T. Kung "The Structure of Parallel Algorithms" Advances in Computers Vol 19 Academic Press 1980
8) Shimon Even, "Parallelism in Tape Sorting" CACM vol 17 No 4 April 1974
9) S. Todd, "Algorithm and Hardware for a Merge Sort Using Multiple Processors" IBM J. RES. DEVELOP vol 22 No 5 1978
10) E. Babb, "Implementing a Relational Database by Means of Specialized Hardware" ACM TODS vol 4. No 1 1979
11) D.R. McGregor et al "High Performance Hardware for Database Systems" in Systems for Large Data Bases. North-Holland, 1976