March 3, 1999
RT0310
Multimedia 10 pages

Research Report

Attribute Processor: An Efficient Parallel Processing Method for 3D Graphics Geometry Calculation

K. Kawase, T. Moriyama

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

概要

PHIGS や **OpenGL** などの対話型の三次元グラフィクスインターフェースのジオメトリ演算をマルチプロセッサを用いて処理する場合，描画アトリビュートの処理が高速化のために重要である．本稿では汎用マイクロプロセッサを使用したマルチプロセッサシステムにおいて，描画アトリビュートを効率良く処理する方法としてアトリビュートプロセッサを提案し，実際のアプリケーションの描画命令列のトレースを用いてシミュレーションによる比較を行い，提案する方式の優位性を示した．Efficient handling of drawing attributes is one of the most important design points of parallel rendering of interactive 3D graphics such as **PHIGS** and **OpenGL**. We invented Attribute Processor method for such applications. In this paper we present our method along with several alternatives, and describe the superiority of our method by simulation using real application traces.

1 はじめに

三次元グラフィクスはその計算量の多さから，マルチプロセッサを用いて高速化を図る試みが多くなされてきた．筆者らは対話型のグラフィクスインターフェースである **PHIGS**[3] や **OpenGL**[8] を採用する場合，モデリングデータの構造からくる制約が性能上の問題点となることを指摘してきた（文献 [6]）．

描画プリミティブ（以降単にプリミティブと呼ぶ）の処理は並列化により容易に性能を向上させることができるが，描画アトリビュート（以降単にアトリビュートと呼ぶ）の処理は並列化が困難である．プリミティブだけを並列処理した場合，アトリビュートの処理コストが相対的に増加し，プロセッサの処理能力の半分以上がアトリビュートの処理に費やされる場合も生じる．従って，アトリビュートの処理をいかに高速化するかがシステム全体の性能を上げる鍵である．

本稿では実際の CAD アプリケーションの例としてダッソーシステムズ社の機械設計 CAD CATIA を，そして Standard Performance Evaluation Corporation のグラフィクスベンチマーク OPC Viewperf [9] の中から CAD アプリケーションのデータを用いた CDRS-03 とビジュアリゼーション（可視化）アプリケーションのデータを用いた DX-03 を選び，それらが生成する描画命令列のトレースを採取し，そのトレースデータを用いたシミュレーションを行い，複数のアトリビュート処理方式の実行効率の評価を行った．

なお，本稿では座標変換，照度計算等のジオメトリ演算の処理のみを扱い，ホストシステムからのデータ転送やラスタ処理部は隘路にはならないという仮定を置いている．またラスタ処理部の構成法についても触れない．ジオメトリ演算の詳細については各 API の仕様 [3, 8] を参照されたい．

2 アトリビュート

PHIGS や **OpenGL** におけるアトリビュートに関して簡単に触れておく．表示するための三次元データには，各種のプリミティブ（点，直線，多角形等）の描画を指示するものとアトリビュート（プリミティブの色，光の拡散率，変換行列等）の設定を指示するものがある．

プリミティブの処理中に必要とされるアトリビュートの種類はプリミティブの種類によって異なるが，多くのものは後続するプリミティブ間で共通している場合が多い．たとえば，1 つの物体をいくつかの多角形で表現する場合，変換行列は共通であるし，均一な物質からできていれば色や光の拡散率は変わらない．**PHIGS** や **OpenGL** では，一度設定したアトリビュートの値は同じアトリビュートが再設定されない限り有効である[1]．実行時にはアトリビュート設定指示に従ってある領域にアトリビュートを保存しておき，プリミティブを処理する間にはその領域からアトリビュートを読み出して使用する．このアトリビュートを保存する領域を今後 ASL（Attribute State List）と呼ぶことにする．

[1]アトリビュートは Push/Pop 可能なので Pop 時にもアトリビュートの値は変更され得る．

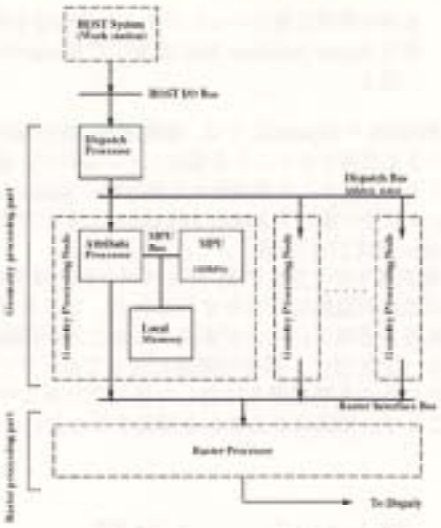

図 1: システムの構成

PHIGS の定義では ASL の大きさは可変長であるが，実装時に大きさを制限することも許されており，通常は 1 K ないし 2 K byte 程度の固定長で実装する．**OpenGL** も同様である．

3 システムの構成と処理の流れ

プロセッサの配置のしかたとしては，直列に並べ 1 つのプリミティブをパイプラインで処理する**パイプライン方式**と，並列に並べ，手の空いたプロセッサにそれぞれ 1 つのプリミティブのジオメトリ計算を順次割り当てる**順次振り分け方式**があるが，後者を用いた方が負荷分散の面での優位であることが示されている（文献 [10, 6, 4, 2]）のでそれを採用することとした．

また，中小規模のマルチプロセッサシステムを構築する際，共有バス型の共有メモリを実装することは汎用性の高いシステムにできる反面，プロセッサ数が増えてくると実装にかかるコストが大きくなる．本稿では実装コストを重視し，共有バス型の共有メモリを持たない方針をとり，1 に示すシステム構成で検討を行った．

演算ノードは Dispatch Bus，Raster Interface Bus および MPU Bus 間のブリッジ機能とアトリビュート処理機能（後述）を持つアトリビュートプロセッサ，ジオメトリ演算を行う汎用マイクロプロセッサ（MPU），そして Local Memory からなる．

大まかな処理の流れは以下の通りである．

1. プリミティブとアトリビュートからなる描画命令列はホストシステムの主記憶内にあり HOST I/O Bus を通して，ジオメトリ処理部のディスパッチプロセッサ（Dispatch Processor）に送られる．
2. ディスパッチプロセッサは Dispatch Bus を通して描画命令列を順次各演算ノード（Geometry Processing Node）に分配する．
3. 演算ノードは配られたプリミティブに対して座標

変換や照度計算といったジオメトリ処理を行い，結果を Raster Interface Bus を通して Raster Processor に送る．

PHIGS や **OpenGL** では，描画命令の発行順序を維持したまま描画を行うことを規定しているので，並列にジオメトリ処理をした描画命令を発行順に Raster Interface Bus 上で並べ直して Raster Processor に渡す方式（Sort-middle 方式 [7]）を使う．

順次振り分け方式では，ディスパッチを行うプロセッサの性能が隘路になりやすいことと，アトリビュートの処理の効率よい扱いが重要であることが指摘されている（文献 [5]）．前者の問題に対しては，ディスパッチのための支援機構を付加した専用プロセッサを用いることで，隘路になることを回避することにした．

4 アトリビュート処理

4.1 アトリビュート処理の問題点

順次振り分け方式で並列処理する場合，演算ノードがプリミティブの処理をしている間，ASL のスナップショットを演算ノードごとに個別に保持しなければならない．何故ならば，全ての演算ノードで共通な ASL を設けたとすると，ある演算ノードが 1 つのプリミティブの描画のために ASL を参照している間は ASL の内容を固定しておく必要があるので，そのプリミティブの描画の指示に続くアトリビュート設定指示を実行することができないからである．

ASL の参照の衝突を避けるために，プリミティブとともに ASL の内容を複製して各演算ノードに転送する方法が考えられるが，一般的にプリミティブ転送のコストに対して ASL の内容の全てを複製・転送するコストは無視できず実用的でない．

4.2 従来の方法

ジオメトリ演算を効率よく並列処理を行うためには，アトリビュートを効率よく演算ノードに割り振ることが重要である．この問題に対して支援機構の導入による解決方法の例をいくつか示す．

4.2.1 Broadcast 方式

すべてのアトリビュートを演算ノードに対して全放送してしまう方法である（文献 [10]）．この方法は全放送の機構と各演算ノードの入力 FIFO（Input FIFO）だけを必要とするので実装は容易である（2）．

しかしこの方法ではアトリビュートの処理に対して並列処理の効果が出ないのでプリミティブあたりのアトリビュートが多いときや，演算ノードの数が増えたときは非常に効率が悪くなる．

各演算ノードでのアトリビュートの処理は以下の手順でおこなう．

1. プロトコルヘッダを読み込み，解析する

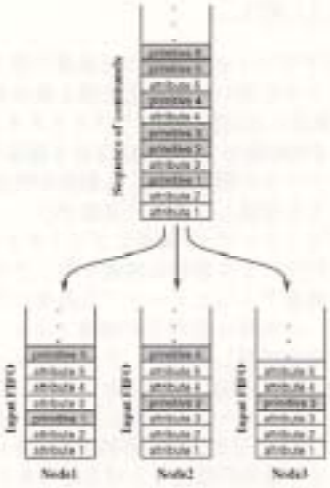

図 2: Broadcast 方式

2. アトリビュートの格納先を計算する
3. アトリビュートを入力 FIFO から読み込む
4. アトリビュートを格納する

これらの処理を汎用プロセッサで行った場合，上記 (1) と (2) の処理はアトリビュートデータのサイズによらず 10 クロック，(3) と (4) の処理にはアトリビュートデータ 1 ワードあたり 2 クロックかかると見積もると，N-way の演算ノードではプロトコルサイズが m のアトリビュートを処理するたびに，全体で $(N-1)\times(10+2(m-1))$ クロックの無駄が生じる．

4.2.2 Broadcast-Unicast 方式

この方法はまずアトリビュートを頻繁に変更される種類とそうでないものの 2 種類に分類し，「頻繁に変更されるアトリビュート」を格納するためのバッファ（3 の ASL sub）を装備する．「頻繁に変更されるアトリビュート」はディスパッチ対象のノードに対して送ると同時に ASL sub にも格納しておく．ディスパッチの対象ノードを切り替える場合には ASL sub の内容をすべて対象のノードに送る．これを catch-up 動作と呼ぶ．「頻繁に変更されないアトリビュート」は Broadcast 方式と同じ方法を用いて各演算ノードで処理をおこなう（文献 [1]）．履歴が問題になるような処理は「頻繁に変更されないアトリビュート」に分類する．

「頻繁に変更されるアトリビュート」のサイズの総和，すなわち ASL sub のサイズが小さいほど catch-up に伴うオーバヘッドが少なくなる．ただし，実際に頻繁に変更されるアトリビュートであるにも関わらず「頻繁に変更されるアトリビュート」に分類されないものがあると，頻繁に全放送が起き効率が低下する．

この方法は **OpenGL** には有効であるが，**PHIGS** では「頻繁に変更されるアトリビュート」の総量が大きいため効率が上がらない．また NAME SET 等のように「頻繁に変更されるアトリビュート」のなかにも履歴が問題になるものがあるので処理が困難である．

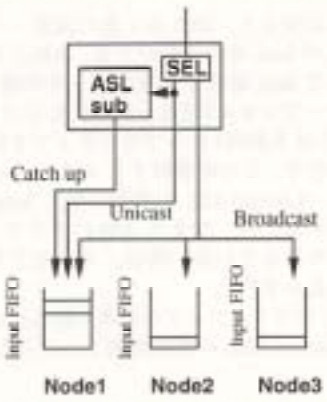

図 3: Broadcast-Unicast 方式

5 提案する解決方法

5.1 アトリビュート転送の最適化

従来の方式の問題を回避するために演算ノードに対するデータ転送量を最小にすることを考える．そのためにディスパッチプロセッサ内部に完全な ASL の複製（4 中の ASL-common）とともに，それぞれの演算ノードに対するアトリビュートの複製（同図 ASL-copy1, ASL-copy2, ASL-copy3）を装備する．ASL-copy の各エントリには ASL-common との差分を表すフラグ（変更フラグ）がついている．

アトリビュートが来ると，ディスパッチプロセッサは ASL-common の値を更新すると同時にディスパッチ対象のノードに送る．ASL-copy の処理はディスパッチ対象かどうかで異なる．ディスパッチ対象のノードに対しては ASL-copy のエントリを更新するとともに変更フラグをクリアする．それ以外のノードに対しては該当する ASL-copy の値と ASL-common の値を比較し，その結果を変更フラグに反映させる．

ディスパッチ対象のノードを切り替える場合は，新しいノードの ASL-copy の変更フラグを調べ，値の変更のあったアトリビュートのみを対象ノードに送出する．それと同時に該当する変更フラグをクリアする．これによって Dispatch Bus の転送量を抑えることができる．

さらに最適化を行う場合には，プリミティブの処理に必要なアトリビュートのセットを予め表としてディスパッチプロセッサ内部に持っておき，以下の 2 つの条件を満たすものだけを送る方法が考えられる．

- これから割り当てようとしているプリミティブの処理に必要なもの．
- 前回の割り当て時から値が変更されたもの．

この処理により catch-up に伴うアトリビュートの転送量を削減できるが以下の問題が残る．

- ディスパッチプロセッサ内部にすべての演算ノードに対応するだけの ASL-copy を持たなければならないので複雑化する．
- 演算ノードを増やすにはディスパッチプロセッサを作り直さなければならない．

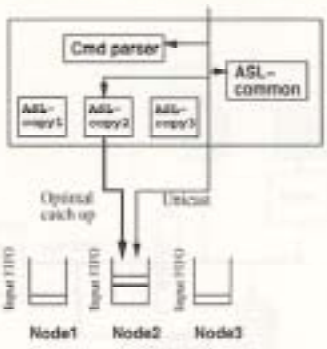

図 4: アトリビュート転送の最適化

- 個々の演算ノードから見れば最適な転送であるが，Dispatch Bus には同じアトリビュートが複数回流れることとなり，演算ノード数の増加とともに Dispatch Bus が飽和する．
- ディスパッチプロセッサ内部での catch-up 処理は並列処理ではないので演算ノードが増加するとその処理が隘路になってくる．

そこで，ディスパッチプロセッサ内部の ASL-copy を各演算ノード内部に移すことによりディスパッチプロセッサの複雑化を抑え，スケーラビリティを確保し，さらに Dispatch Bus のデータ転送量を最小に抑える方法として Broadcast-AP 方式を提案する．

5.2 Broadcast-AP 方式

Broadcast-AP 方式ではアトリビュートは常に全放送され，各演算ノードでアトリビュートの複製を更新する．

そのため，アトリビュート処理用の支援機構であるアトリビュートプロセッサと MPU を組み合わせることを提案する（1 および 5）．MPU には高性能な浮動小数点演算器とスヌープキャッシュを持つものを選択し，アトリビュートプロセッサとの間を共有バスでつなぐ．

アトリビュートプロセッサは MPU とは並列に動作し，アトリビュート設定指示を解釈し処理する．プリミティブディスパッチ時は，前回のディスパッチからのアトリビュートの差分を検出し，変化が起こったアトリビュートが MPU 内部にキャッシュされている場合はスヌープキャッシュの機能を利用してそれを無効化する．従って MPU はディスパッチされたプリミティブの処理に必要でなおかつ前回のディスパッチから変化したアトリビュートのみを共有バスを通して参照すればよいので，アトリビュートプロセッサと MPU の間の通信量は最適化され，ジオメトリ演算に専念できる．

アトリビュート処理機能を内蔵した専用のジオメトリ演算用プロセッサを開発することも考えられる（文献 [6]）が，そのような専用の高性能数値演算用プロセッサを新たに開発することは，高性能な浮動小数点演算機能を持つ汎用 MPU が安価に入手できるようになってきたことから，コストおよび開発期間の面で不利である．

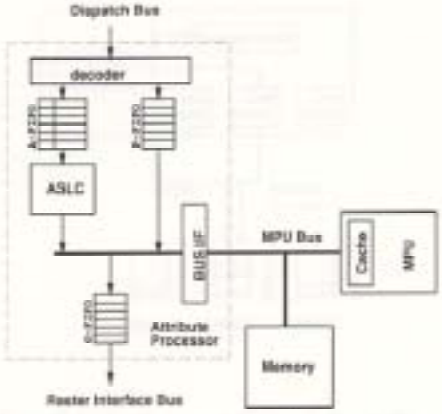

図 5: アトリビュートプロセッサ

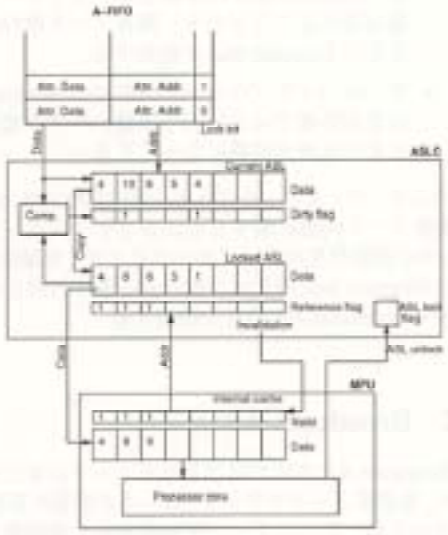

図 6: ASLC の構成

5.3 アトリビュートプロセッサの概要

ディスパッチプロセッサによって発行されたプロトコルは Dispatch Bus を通してアトリビュートプロセッサに渡される．アトリビュートプロセッサは decoder によってプロトコルをアトリビュートとプリミティブに分けて，それぞれアトリビュート FIFO およびプリミティブ FIFO（以降それぞれ A-FIFO，P-FIFO と呼ぶ）に格納する．

A-FIFO に入ったアトリビュートは ASL Controller（以降 ASLC と呼ぶ）によって処理される．6 に ASLC の構成を示す．ASLC には最新の ASL の値（Current ASL）と任意の時点での ASL のスナップショットをとる機能があり，そのスナップショット（Locked ASL）は MPU からはメモリマップド I/O として見える．

基本的な動作は以下の通りである．

1. ディスパッチプロセッサはプリミティブを割り当てるに先立ち，割り当て先の演算ノードに対して ASL の lock 命令を発行する．ASLC は A-FIFO を通して lock 命令を受信するとその時点の ASL のスナップショットを Locked ASL にとる．
2. MPU は P-FIFO からプリミティブを読み込んで処理を行う．この時参照する ASL は Locked ASL である．Locked ASL の参照に際し lock 命令が完了していない時，つまり ASLC がスナップショットをとり終えていない時は，その完了を待ってから読み込みを行う．
3. MPU はプリミティブの処理が終ると ASLC に対して Locked ASL の参照が完了したことを告げる．

5.4 ASLC の詳細

ASCL の機能は以下の通りである．

- Locked ASL の内容と MPU のキャッシュの一貫性を取る．
- A-FIFO からの lock 命令を受け，Current ASL の内容の複製を Locked ASL に取る．
- Locked ASL に有効な複製データがあることを MPU に知らせる．

以下順に説明する．

最近の高性能マイクロプロセッサはキャッシュヒットが最高性能を出すための前提となっているので，なるべく ASL をキャッシュに保持するように工夫する必要がある．Current ASL と Locked ASL のそれぞれに MPU のキャッシュラインサイズ（たとえば 32 byte）ごとに 1 bit のフラグを設ける．Current ASL に付随しているフラグ（Dirty flag）は Current ASL と Locked ASL のそのブロックの内容が一致していない場合 dirty となり，一致している場合は clean となる．Locked ASL に付随しているタグ（Reference flag）は MPU から参照された場合 shared になる．参照される前の状態では private である．Locked ASL が有効な値を保持していることを示すため ASL lock flag というフラグを設ける．ASL lock flag は locked もしくは unlocked の値をとる．

ALS lock flag の初期状態が unlocked であるとして ASLC の動作を説明をする（6 参照）．

A-FIFO の中には ASL 内に格納すべきアトリビュートとそのアトリビュートの ASL 内でのアドレスが入っている．ASLC は A-FIFO の先頭を読み出し，そのアトリビュートを Current ASL の該当するアドレスに格納する．このとき Locked ASL の該当アトリビュートの値と比較し，その結果を Dirty flag に反映させる．ここで注意すべき点としては変更によって 1 度 dirty になったものが，次の変更でまた clean にもどる可能性があるということである．

ASLC への lock 命令は A-FIFO を通して送られる．lock 命令は A-FIFO の中の lock bit で判別される．ASLC は lock 命令を受けると，まず ASL lock flag を見に行く．ASL lock flag は 1 ビットのセマフォであり，すでに locked の時は unlocked の状態になるまで待つ．つぎに Current ASL の Dirty flag を走査して，dirty なブロックをさがす．dirty なブロックを見つけると，そのブロックの内容を Locked ASL に複製して Dirty flag を clean にする．また，複製された先のブロックの Reference

表 1: CATIA のトレースデータの統計

Count	Total words	Ave. words	Element type
5796	11592	2.000	ADD NAMES TO SET
5737	11474	2.000	REMOVE NAMES FROM SET
5683	11366	2.000	SET HIGHLIGHTING INDEX
5672	11344	2.000	SET LINE WIDTH SCALE FACTOR
5672	11344	2.000	SET LINETYPE
5672	11344	2.000	GSE SET FRAME BUFFER PROTECT MASK
5671	285190	50.289	GDP DISJOINT POLYLINE3
143	429	3.000	SET PICK IDENTIFIER

flag が shared 場合は MPU の内蔵キャッシュに古い値がキャッシュされている可能性があるので，MPU Bus に invalidate message を流して，MPU のキャッシュの該当ラインを無効化する．その後 Reference flag を private にする．逆に，Reference flag が private である場合，前回 ASL がロックされてから，そのブロックが参照されなかったことがわかるので，invalidate message を送出する必要はない．Dirty flag が全て clean になり，複製および MPU Bus への invalidate message の送出が完了したら，ASL lock flag を locked にする．

MPU は ASL lock flag をポーリングしており，locked になれば Locked ASL を参照してプリミティブの処理を進める．必要なアトリビュートの参照を終えたら，ASL lock flag を unlocked にして次のプリミティブのディスパッチにそなえる．ディスパッチプロセッサはプリミティブの終了を待たずに次のプリミティブおよび ASL lock 命令を先行して，P-FIFO および A-FIFO につめておくことができる．

6 トレースデータ

実際のアプリケーションがどのようなデータを用いているかを調べるために 3 種のアプリケーションの実行トレースをとった．

まず，三次元機械設計 CAD である CATIA を用い，実行される PHIGS のエレメントタイプのトレースをとった．エレメントタイプごとの出現率順に並べたものの上位を 1 に示す．一番左の列は 1 画面を描画する間の出現回数である．この表よりプリミティブ（GDP DISJOINT POLYLINE3）に対して，6 倍の個数のアトリビュートが発行されていることが分かる．一般的に CAD ではプリミティブ 1 つに対して多くのアトリビュートが付く．

このトレースデータのプリミティブとアトリビュートの並びをもう少し詳しく見てみる．2 はトレースデータの一部分を切りだしたものである．

ここで注目すべき点は，1 つのプリミティブに対して 6 個のアトリビュート設定指示がなされているという点と，同種のアトリビュートについては指定されている属性値が同じであるということである．CAD アプリケーションではモデリングデータの編集を簡単にするために，たとえ同じアトリビュートの値であってもプリミティブの発行時に再度アトリビュートを設定することがある．

表 2: CATIA のトレースデータの一部分

Seq	Element type	Value
+00	ADD NAMES TO SET	00000001
+01	SET HIGHLIGHTING INDEX	00000007
+02	GSE SET FRAME BUFFER PROTECT MASK	FF000000
+03	SET LINE WIDTH SCALE FACTOR	3F800000
+04	SET LINETYPE	00000001
+05	GDP DISJOINT POLYLINE3	.....
+06	REMOVE NAMES FROM SET	00000004
+07	ADD NAMES TO SET	00000001
+08	SET HIGHLIGHTING INDEX	00000007
+09	GSE SET FRAME BUFFER PROTECT MASK	FF000000
+10	SET LINE WIDTH SCALE FACTOR	3F800000
+11	SET LINETYPE	00000001
+12	GDP DISJOINT POLYLINE3	.....
+13	REMOVE NAMES FROM SET	00000004

表 3: CDRS-03 のトレースデータの統計

Count	Total words	Element type
31380	125520	glVertex3fv()
31380	125520	glNormal3fv()
31380	94140	glTexCoord2fv()
71	355	glColor4fv()
71	71	glBegin(GL_TRIANGLE_STRIP)
71	71	glEnd()

次に，**OpenGL** のデータとして OPC Viewperf CDRS-03 と DX-03 の実行トレースを取り，それぞれ出現率順に並べたものの上位を 3 と 4 に示す．CDRS-03 には 7 種のテストが含まれるが，そのうち最もアトリビュートの指定が多い No. 4 のテストを採用した．また，DX-03 は No. 3 のテストを採用した．

CDRS-03 では glBegin(GL_TRIANGLE_STRIP) と glEnd() の間に glTexCoord2fv(), glNormal3fv(), glVertex3fv() の列が最低 4 回から最高 2967 回繰り返されていた．平均繰り返し回数は 442.0 回である．

DX-03 では glBegin(GL_TRIANGLE_STRIP) のと glEnd() の間に glColor4fv(), glNormal3fv(), glVertex3fv() の列が最低 22 個から最高 100 回繰り返されていた．平均繰り返し回数は 95.8 回である．

7 シミュレーション

本稿では 4.2 節で述べた Broadcast 方式および Broadcast-Unicat 方式，そして今回 5.2 節で提案した Broadcast-AP 方法についてシミュレーションによる性能評価を行った．ただし，4.2.2 節で述べた通り Broadcast-Unicast 方式を

表 4: DX-03 のトレースデータの統計

Count	Total words	Element type
93536	374144	glVertex3fv()
93536	374144	glNormal3fv()
93536	467680	glColor4fv()
976	976	glBegin(GL_TRIANGLE_STRIP)
976	976	glEnd()

表 5: 演算ノードの処理コスト

Item	Cost (clock)
アトリビュート処理コスト	$10 + 2(m-1)$ [2]
プリミティブ入力コスト	m
プリミティブ計算コスト	109 (CATIA 付録 A 参照)
	157 (CDRS-03, DX-03 付録 B 参照)
プリミティブ出力コスト	8/頂点

表 6: 頻繁に変更されるアトリビュートの設定

Set	Frequent attribute set
A	glNormal, glTexCoord
B	glNormal, glTexCoord, glColor, glMaterial, glEdgeFlag
C	glNormal

PHIGS に対して適用するのは困難であるため，CDRS-03 と DX-03 のみに適用した．

7.1 シミュレーションの方式

今回のモデルでは共有メモリなどがないため，シミュレーションの効率を考慮してサイクルシミュレータを C++ の Task Library で記述した．

アトリビュートおよびプリミティブの種類ごとにメモリの参照コストや計算コストを表として持ち，シミュレーション時にはそのテーブルの内容とプロトコルの内容を参照しながら計算を行うようにした．

7.2 パラメータの設定

シミュレータの設定可能なパラメータのうち，シミュレーションにおいて仮定した主なパラメータの設定内容を以下に列挙する．

MPU は 100MHz で動作し，単精度浮動小数点の積和を毎クロック実行できるものとする．Dispatch Bus はバス幅 64bit で 50MHz で動作する．

演算ノードでの処理コストを 5 に示す．表中の m はプロトコルのワード数である．アトリビュートの入力は MPU のキャッシュを通して行うものとし，キャッシュの fill-in は演算と並行して実行されるので計算コストには現れないものとした．

また，ディスパッチプロセッサがプリミティブを演算ノードに割り振る際にはラウンドロビン方式を用いるようにした．**OpenGL** では文献 [1] に見られるような手法を用いて glBegin(GL_TRIANGLE_STRIP) を 12 頂点ごとに分割してディスパッチ対象を切り替えるようにした．Broadcast-Unicast 方式においては「頻繁に変更されるアトリビュート」の設定を 6 のように 3 種類とした．catch-up 動作時には各アトリビュートに 1 ワードのヘッダが付加され，さらに 64bit 単位に丸め上げられて転送されるものとした．

[2]算出根拠は 4.2.1 節を参照のこと．また Broadcast-AP 方式の場合はアトリビュートプロセッサ内部で処理されるので見かけ上のコストは 0 である．

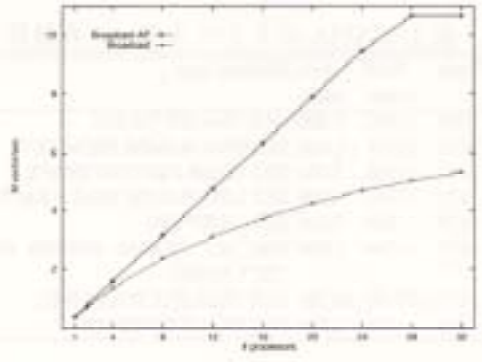

図 7: CATIA のトレースを用いたシミュレーション結果

8 結果

シミュレーションによる実験結果を以下に示す．

8.1 CATIA

CATIA の例では，アトリビュートが多いので 2 方式間で性能に大きな差が出ている (7)．Broadcast 方式ではアトリビュート処理に並列処理の効果が出ないため，演算ノード数を増加した場合の性能向上率が次第に低下している．

一方 Broadcast-AP 方式では演算ノード数に対して線形な性能向上が得られている．演算ノード数が 28 以上で性能が頭打ちになっている領域では Dispatch Bus が飽和していることがシミュレータ上で観測された．

8.2 CDRS-03

次に CDRS-03 の結果であるが Broadcast-AP 方式と Broadcast 方式では CATIA のケースと同様な傾向が見られる (8)．Broadcast-AP 方式および Broadcast-Unicast 方式の設定 A と設定 B では，ある演算ノード数を超えた時点で全く性能向上がみられなくなる．これらは CATIA の例でみられたように Dispatch Bus の飽和が原因である．

Broadcast-Unicast 方式では catch-up の対象とするアトリビュートの設定の仕方で結果が大きく異なる．3 からわかるように，CDRS-03 では glTexCoord と glNormal のみが「頻繁に変更されるアトリビュート」であるので，それらのみを catch-up するようにした設定 A が最も効率が良い．しかしながら，その場合においても catch-up のためにプロセッサの処理を要するため Broadcast-AP 方式に比べて，プロセッサあたりの処理能力の面で劣っている．また，バスの飽和領域においても catch-up のための余分なバス転送が必要なため Broacast-AP 方式に比べて性能が劣っていることがわかる．

Broadcast-Unicast 方式の設定 B では描画中に変更されることのない glMaterial や glEdgeFlag などのアトリ

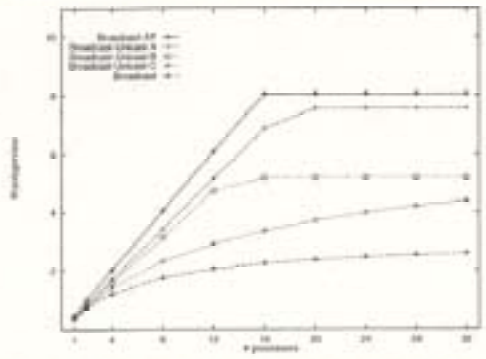

図 8: CDRS-03 のトレースを用いたシミュレーション結果

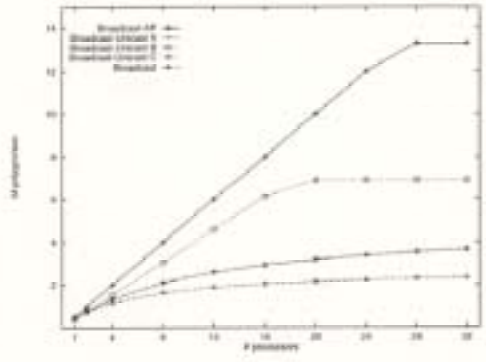

図 9: DX-03 のトレースを用いたシミュレーション結果

ビュートも catch-up の対象としているため演算ノード当りの性能が設定 A に比べさらに低下しており，バスの飽和領域においても性能低下がみられる．

Broadcast-Unicast 方式の設定 C では「頻繁に変更されるアトリビュート」である glTexCoord がブロードキャストされているため，プロセッサの処理能力を浪費しており性能が線形に向上しなくなる．特性としては Broadcast 方式に近い．

8.3 DX-03

DX-03 のシミュレーション結果を 9 に示す．

Broadcast-Unicast 方式の中では設定 B が最も高い性能を出しいる．設定 A は大幅に性能低下し，設定 C と同一の性能にとどまっている．設定 B は無駄な catch-up を行ってはいるが，「頻繁に変更されるアトリビュート」である glNormal と glColor が両者とも catch-up の対象になっており，アトリビュートの全放送が起きていない．それに対し，設定 A では glColor が全放送され，プロセッサの処理能力を大きく浪費していることが性能低下の原因である．

このように Broadcast-Unicast 方式では「頻繁に変更されるアトリビュート」の設定をアプリケーションの特性に合わせて最適に選択しないと性能が低下してしまう．異なる特性のアプリケーションを使用する場合や，1 つのアプリケーションの中で特性が変わるような場合には，常に最適な設定を選択しておくことは難しい．

一方本提案の Broadcast-AP 方式では Dispatch Bus のデータ転送，および演算ノードでのアトリビュート処理が常に最適に行われるので，アプリケーションの特性によらず常に最大性能を引き出す事ができる．

9 おわりに

本稿では汎用プロセッサにアトリビュート処理専用のアトリビュートプロセッサを組み合わせることで，並列三次元グラフィクスシステムのジオメトリ演算部を構築することを提案した．

アトリビュートプロセッサの導入により，ディスパッチバスの転送量と MPU の処理を最適化するとともに，スヌープキャッシュを利用して ASL のデータのうち現在の処理に必要でかつ前回から変化した部分のみを MPU のキャッシュに送り込むことを可能とした．

また，アプリケーションのトレースデータを用いたシミュレーションにより，本方式が従来の方式に比べてアプリケーションの特性によらず優位であることを示した．

最近の動向としては Toolkit を用いて API の呼び出しを減らすことによる最適化の手法も登場してきたが，機械設計 CAD のようなアプリケーションではそれらの手法をすぐに取り入れることは難しいと思われる．今後 **OpenGL** での CAD が出回り出したところで，それらを含めたさらに多くのアプリケーションでの評価を行いたい．

参考文献

[1] Akeley, K.: Reality Engine Graphics, *SIGGRAPH '93 Proceedings*, ACM, Addison-Wesley, pp. 109–116 (1993).

[2] Horning, R. and et al.: System Design for a Low Cost PA-RISC Desktop Workstation, *Proc. of COMPCON91 SPRING*, IEEE, pp. 208–213 (1991).

[3] ISO: *ISO/IEC 9592-1:1989(E), Information processing systems - Computer Graphics - Programmers Hierarchical Interactive Graphics System (PHIGS), Part 1 - functional description* (1989).

[4] Kirk, D. and Voorhies, D.: The Rendering Architecture of the DN10000VS, *ACM Computer Graphics*, Vol. 24, No. 4, pp. 299–308 (1990).

[5] 松本尚, 川瀬桂, 森山孝男: PHIGS の構造体を処理するジオメトリ演算部のマルチプロセッサ上での実行効率評価, 情報処理学会誌, Vol. 34, No. 4, pp. 732–742 (1993).

[6] 松本尚, 川瀬桂, 森山孝男: PHIGS のジオメトリ演算部のための並列処理方式の検討, 情報処理学会誌, Vol. 35, No. 1, pp. 92–101 (1994).

[7] Molnar, S., Cox, M., Ellsworth, D. and Fuchs, H.: A Sorting Classification of Parallel Rendering, *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, pp. 23–32 (1994).

[8] Segal, M. and Akeley, K.: *The OpenGL Graphics System: A Specification (Version 1.1)* (1997).

[9] Standard Performance Evaluation Corporation: Graphics Performance Characterization Group (http://www.specbench.org/gpc/).

[10] Torborg, J. G.: A Parallel Architecture for Graphics Arithmetic Operations, *SIGGRAPH '87 Proceedings*, ACM, Addison-Wesley, pp. 197–204 (1987).

A CATIA の計算時間の内訳

処理内容	時間 (clock)
座標変換 (MC → WC)	20
座標変換 (WC → NPC)	42
クリッピングチェック	12
座標変換 (NPC → DC)	15
後処理	20
合計	109

B CDRS-03, DX-03 の計算時間の内訳

処理内容	時間 (clock)
座標変換 (Model → Eye)	20
照度計算	48
座標変換 (Eye → Clip)	22
クリッピングチェック	12
座標変換 (Clip → Window)	35
後処理	20
合計	157