2005－CG－118　(11)
2005／2／7

# クラスタベース没入型ディスプレイのための透過性を考慮したアプリケーション・プラットフォームの構築

**石田 善彦** * **橋本 直己** * **佐藤 誠** *

**あらまし**　高解像度で広視野の映像を投影する没入型ディスプレイにおいて，描画処理に PC クラスタを利用することが多くなっている．しかし，アプリケーションの並列化には煩雑な作業が必要なため開発コストが増加するという問題があり，また，ネットワーク性能がボトルネックとなり十分な実行性能が得難いことが指摘されている．本研究では，ネットワーク負荷を低く抑えることのできる Master-Slave モデルを用い，API 呼び出しに対して介入処理を行なうことによって既存アプリケーションを透過的にクラスタ環境へ対応させることが可能なソフトウェア環境を提案する．また，実際のクラスタベース没入型ディスプレイにおいて提案システムを構築し，評価実験を行なう．

# A Transparent Application Platform for Cluster-Based Immersive Projection Displays

Yoshihiko Ishida* Naoki Hashimoto* Makoto Sato*

**Abstract**　PC-Clusters tend to be used for immersive projection displays on which high-resolution and wide field of view images are projected. However, development costs increase for parallel programming, and there is a possibility that the program cannot execute at an enough speed since the network bandwidth might become a bottleneck. This paper proposes an application platform based on the Master-Slave Model which only requires low network bandwidth. By intercepting API calls, this application platform allows an existing application to execute transparently in clusters . We implemented the proposed system in an actual cluster-based immersive projection display and conducted evaluation experiments.

## 1.　はじめに

近年, VR 分野では, ユーザの視野を覆う大型のスクリーンを用いる没入型ディスプレイの研究が盛んに進められている．没入型ディスプレイは，ユーザに高い臨場感を提示するために，広視野，高解像度の映像を表示する．インタラクションが伴う仮想空間を提示するためには, リアルタイムに映像を生成する必要があり, 映像の生成には高いグラフィックス処理能力が要求される．従来は, 描画処理に高性能なグラフィックスワークステーションが利用されてきたが，ここ数年で, PC のグラフィックス処理能力の向上に伴い, 複数台の PC から構成される PC クラスタを用いることが多くなっている．PC クラスタを利用する利点としては, グラフィックスワークステーションに比べ, コストパフォーマンスが高く, ディスプレイ規模に応じて, スケーラブルに対応しやすいという点が挙げられる．また, PC クラスタは, 性能の進歩の早いコモディティ製品によって構成されるため, 低コストで最新の技術動向に追随できる点も長所である．

没入型ディスプレイの描画処理システムに関して, PC クラスタという選択肢ができたことで, ハードウェアの面では導入しやすくなってきているが，一方で, ソフトウェアの面では，まだ容易に利用できるとはいえないのが現状である．PC クラスタベースのシステムでは, ノード間の並列・分散処理が不可欠である．それゆえ, 従来とは異なるプログラミングモデルでアプリケーションの開発を行なわなければならず, 並列・分散処理の実装のために開発コストが増加するという問題がある．

また, アプリケーションの種類や実装方法によっては, 並列・分散処理を行なうことでネットワーク帯域がボトルネックとなり, 十分な実行速度を得ることができない場合がある．そのため, PC クラスタを利用する場合, 高性能なネットワーク環境が要求されるとともに, ネットワーク負荷を考慮した開発を行なわなければならず, アプリケーションの開発や利用を困難にしている．

そこで, 我々は, 低いネットワーク負荷で並列処理を実現するとともに，並列化に伴う開発コストが発生しないように，既存のアプリケーションを修正しなくてもクラスタベースの没入型ディスプレイで実行させることができるアプリケーション・プラットフォームを提案する．

## 2.　関連研究

クラスタベースの没入型ディスプレイ環境でアプリケーションを実行するために, これまで, 様々な開発・実行環境が提案されてきた．それらは大きく分けると, Client-Server モデルと Master-Slave モデルの 2 つに分類することができる [1].

Client-Server モデルとは, 1 つのクライアントノード上で, アプリケーションを実行し, 生成される描画情報を複数の描画専用のサーバに配信することによって, 並列的な描画を行なうものである．Client-Server モデルの模式図を図 1 に示す．このモデルを用いた代表的なシステムとして, Chromium [2] が挙げられる．Chromium は, OpenGL ドライバを独自のドライ

*東京工業大学 精密工学研究所
Precision and Intelligence Laboratory, Tokyo Institute of Technology

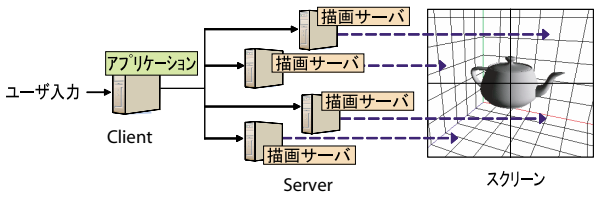


図 1: Client-Server モデル

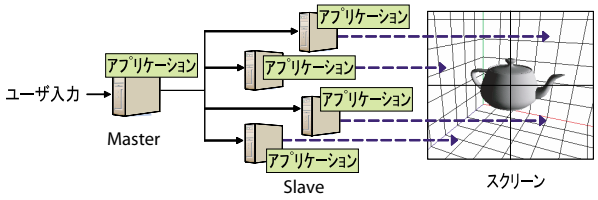


図 2: Master-Slave モデル

バに置き換え, アプリケーションからの描画命令の呼び出しをクラスタの描画サーバに分配することで，分散レンダリングを行なう．この機能は標準の OpenGL のインタフェースと同じインタフェースで提供されるため, OpenGL を利用するアプリケーションは，特にソースコードの修正を行なわなくてもクラスタ環境に対応することができる．このように，アプリケーションが実行環境の違いを意識せずに実行環境が持つ機能にアクセスできることを透過性と呼ぶ．Chromium は高い透過性を持つことが大きな利点であるが，複雑なシーンの描画では配信しなければならない描画命令が増大し, ネットワーク帯域が実行速度のボトルネックになってしまうという問題がある．

また，Client-Server モデルを用いたシステムで，描画命令ではなくシーングラフの形式でデータを配信するものも提案されている [3]. シーングラフを描画サーバ側で保持し，シーングラフの更新分のみをクライアントから配信するだけでよいため，ネットワーク負荷を抑えることができるが，頻繁にオブジェクトの変形やオブジェクト数の増減が発生する場合は, ネットワーク帯域がボトルネックになる可能性がある．

もう一つのモデルである Master-Slave モデルは，同一のアプリケーションをクラスタの各ノードに配置し, ノード間で同期をとりながらそれぞれ異なる領域を描画することで並列に描画処理を進めていくものである．Master-Slave モデルの模式図を図 2 に示す．このモデルを採用するシステムとして，VRJuggler [4] や CAVELib [5] といった開発環境がある．Master-Slave モデルでは, ノード間で描画結果にずれが生じないように，アプリケーションの状態を全てのノードで等しくする必要がある．この処理は, アプリケーションの同期に必要なデータをマスタからスレイブに配信することによって実現される．配信されるデータは, ユーザ入力等のアプリケーションの状態遷移に関わるデータである．同期のためにネットワークを流れるデータ量は少なくて済むため，ネットワーク帯域はボトルネックになりにくいのが利点である．一方で, これらのシステムはライブラリやアプリケーション・フレームワークという形で機能が提供されるため, 既存のアプリケーションに関しては, ソースコードの移植作業が必要となり, これまでのソフトウェア資源を利用しにくいという問題がある．

## 3. 透過性を考慮した Master-Slave モデル実行環境

本節では，我々が提案する，PC クラスタベースの没入型ディスプレイ上で既存アプリケーションを実行可能にするためのアプリケーションプラットフォームについて述べる．これは，高い透過性を持つとともに，低いネットワーク負荷で実行できることを目標としている．

提案システムには，ネットワーク帯域がボトルネックになりにくいよう，通信量を抑えることができる Master-Slave モデルを採用する．つまり，描画を行なう全てのノードでアプリケーションを実行し，ネットワークを介して各ノードの実行にずれが生じないように同期処理を行ないながら描画を進めていく方式を用いる．ここで，同期処理とは，全てのノードにおいてプログラムのある地点で待ち合わせ，プログラムの状態を等しくするために必要なデータをマスターノードからスレイブノードへブロードキャストした後，一斉にプログラムを再開するという処理を指す．

従来の Master-Slave モデルを用いたシステムでは，同期処理を実現するために，アプリケーション開発時にプログラマが特定のコードを組み込む作業を行なう必要がある．しかし，アプリケーション自体に処理を組み込むのではなく，実行環境の側が適応的に同期処理を呼び出すような仕組みを実現することができれば，このようなアプリケーション開発時のオーバーヘッドを避けることができ，また，ソースコードが提供されないアプリケーションに対してもクラスタ環境で実行できるようにすることが可能となる．アプリケーションの側ではなく，実行環境の側で同期処理を行なうことによって PC クラスタに対応する手法は，アプリケーションをソースコードレベルで最適化して対応した場合と比べると性能は劣るものの, 実用上有効な性能で実行できることが報告されている [6].

この仕組みを実現するために，アプリケーションと実行環境とのインタフェース，API(Application Programming Interface) に着目する．アプリケーションは，API を通して，ウィンドウ管理やシステムイベントの取得，入出力処理，描画処理といった実行環境の機能にアクセスする．アプリケーションが API を呼び出すと，処理は，アプリケーション内のルーチンから，API 呼び出しによって実行されるライブラリ内のルーチンへと移る．しかし，API によって呼び出されるライブラリを置き換えたり，アプリケーションが参照するインポートテーブルを変更することによって，API が呼び出されたときに本来の API の処理に代って任意の処理を実行することが可能である．提案システムはこのような介入処理を利用して，クラスタ環境に対応する処理を実行する．アプリケーションの API 呼び出しに対して介入し処理を行なうソフトウェアを，本研究では，API アダプタと呼ぶことにする．アプリケーションは，従来のインタフェースをそのまま利用することができるので，透過的にプラットフォームにアクセスすることが可能である．

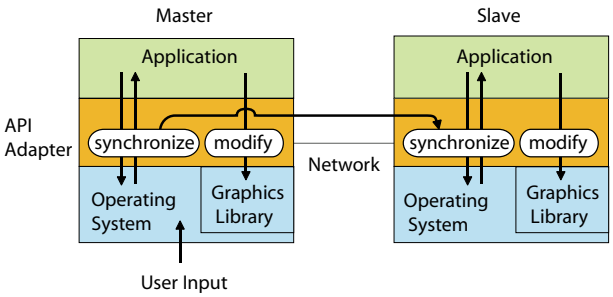


図 3: システム構成

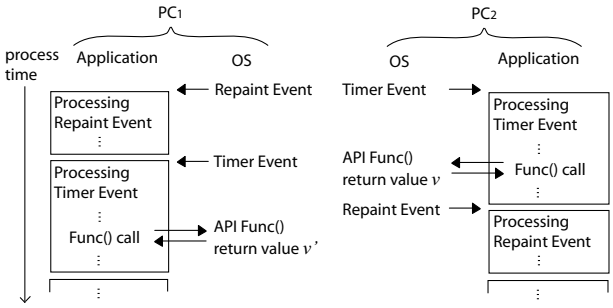


図 4: イベントや API 呼び出しを同期しない場合

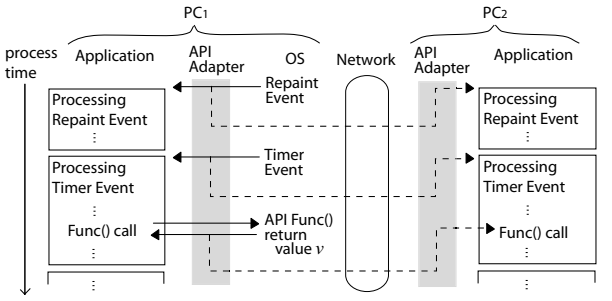


図 5: イベントや API 呼び出しを同期した場合

システムの構成を図 3 に示す．各ノードには同一のアプリケーションと API アダプタを配置する．API アダプタによってマスタノードからスレイブノードへネットワークを介して必要なデータを配信し，同期をとりながらアプリケーションを実行する．また，API アダプタにおいて，グラフィックスライブラリの呼び出しに割り込み，描画領域を変換することで，マルチプロジェクション環境に対応する．これらの API アダプタで行なう処理の詳細について，以下の小節で述べる．

### 3.1 イベントや API の返値の配信による同期処理

描画に不整合が発生しないために，クラスタの各ノードで実行されるアプリケーションの状態は一致している必要がある．一般にアプリケーションの状態は，ユーザ入力やファイル入出力，システムイベントの取得といった実行環境とのやりとりによって一意に決定される．これらの実行環境とのやりとりは，API を通して行なわれるため，API 呼び出しのタイミングと呼び出した結果の作用が等しければ，アプリケーションの状態も同じになると考えることができる．

同期をとらずにクラスタの各ノードでアプリケーションを実行した場合，ノード毎に API を呼び出したときの返値が異なる可能性がある．例えば，ミリ秒精度のシステム時間や高分解能タイマの値を取得する API の返値は，呼び出されるタイミングによって差異が生じる．また，発生したシステムイベントがキューに投入される順序やタイミングは必ずしも一意ではないため，イベントにより駆動されるルーチンの実行結果も一意にならない (図 4)．それゆえ，クラスタの各ノードで同じアプリケーションを同時に実行したとしても，各アプリケーションの状態遷移に差異が生じ，描画内容にずれが発生してしまう．そこで，提案システムでは，API アダプタによって，ノード毎に差異が生じる可能性のある API の返値やシステムイベントをマスタノードからスレイブノードへ配信し同期をとることで，アプリケーションの状態を一致させる (図 5)．

ただし，この手法は，API が呼び出されるタイミングで処理を行なうだけではアプリケーションの状態遷移を一意に保つことができない場合は有効ではない．例えば，複数のスレッドが同じデータ領域に書き込むような実装になっているとき，スレッドの実行タイミングによって実行結果が変化する可能性があるが，ここで用いている手法では，スレッドの実行が切り替わるコンテキストスイッチのタイミングまで制御することはできないため対応できない．しかし，一般的な VR アプリケーションでは，そのような実装になっているとは考えにくいため，この制限は大きな支障にはならないと考える．

### 3.2 没入型ディスプレイのスクリーン構成への対応

没入型ディスプレイのスクリーンへの表示は，通常，複数台のプロジェクタによって投影するマルチプロジェクションによって行なわれる．スクリーンは矩形に分割され，それぞれの領域に対してプロジェクタとそれを駆動する描画ノードが割り当てられる．正しく没入型ディスプレイ上に表示するには，スクリーン構成に合うようにノード毎に描画領域を設定する必要があり，視点や視錐台を適切に変更しなければならない (図 6)．提案システムでは，グラフィックスライブラリの API 呼び出しに介入して各ノードに適切な視点や視錐台の設定を行う．スクリーン構成に応じたパラメータを設定することで，様々なマルチプロジェクションディスプレイに対してスケーラブルに対応することが可能である．

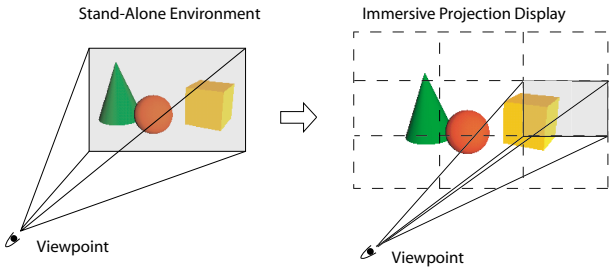


図 6: 視点および視錐台の変換

また，マルチプロジェクション環境において映像更新のタイミングでずれが発生するのを防ぐために，

API アダプタは，スワップロックと呼ばれる機能を提供する．これは，フレームバッファをスワップするタイミングで同期をとる機能である．API アダプタは，フレームバッファをスワップする API が呼び出されたときに，全ノードでバリア同期をとることでこの機能を実現する．

## 4. 評価実験

我々は, 提案システムを実際のクラスタベース没入型ディスプレイ環境である D-vision において構築した．API アダプタは，WindowsOS 上で OpenGL を利用するアプリケーションに対して対応できるよう実装した．提案システムの基本的な性能を知るために，基礎実験として，ポリゴン数を変化させたときの描画性能を Client-Server モデルの Chromium と比較し，また，同期回数が性能に与える影響について考察した．さらに，応用実験として，市販のアプリケーションを D-vision で実行し，提案システムと Chromium の性能の比較を行なった．このときの実行の様子を図 7 に示す．

図 7: 実行の様子

### 4.1 実験環境

まず，今回の実験を行なった環境について説明する．実験に用いた没入型ディスプレイである D-vision は，24 台の PC で構成される PC クラスタによって描画処理を行なう．各ノードの仕様を表 1 に示す．今回の実験では，クラスタのノード間を接続するネットワーク環境として，帯域が 100Mbps である Fast Ethernet を利用した．

D-vision のスクリーンは，$4 \times 4$ の領域に分割されている．中央の縦の $2 \times 4$ の領域に関しては，偏光メガネを利用したステレオ立体視に対応し，1 つの領域に対して 2 重に投影が行なわれる．そのため，合計で 24 の描画領域があり，各領域の描画はクラスタの 24 つのノードにそれぞれ割り当てられている．

表 1: ノードの仕様

| | |
|---|---|
| CPU | Pentium4 2.4GHz |
| Memory | 512MB |
| OS | Windows XP HomeEdition |
| Graphics board | RADEON9700 Pro 128MB |

### 4.2 基礎実験

基礎実験として，描画負荷を変化させたときの性能を計測した．このとき，Client-Server モデルの代表的なシステムであり，提案システムと同様に既存アプリケーションに対して透過的に実行できる Chromium と性能の比較を行なった．また，同期回数による性能の変化について検討するために，性能のモデル化を試みた．実験で用いたアプリケーションは，OpenGL のツールキットである GLUT [7] を用いて作成したコード量 300 行程度の単純なアプリケーションである．このアプリケーションは，シーンの描画を繰り返し実行するだけで，ユーザ操作は特に発生しない．

#### 4.2.1 描画性能の比較

ポリゴン数を変化させて，提案システムと Chromium によってアプリケーションを実行し，性能を計測した．このとき，アプリケーションをイミディエート・モードで実行した場合とディスプレイ・リストを用いて実行した場合のそれぞれについて計測を行なった．また，1 台の PC で実行したスタンドアロン環境時の性能についても計測した．このときの 1 秒間あたりの描画回数，ネットワークの使用率をそれぞれ図 8，図 9 に示す．ここで，ネットワーク使用率とは，ネットワーク帯域に対する入出力平均トラフィック流量の割合である．提案システムとスタンドアロン環境については，イミディエート・モードで実行した場合とディスプレイ・リストを使用した場合で大きな性能差が見られなかったため，イミディエート・モードでの結果のみを示している．

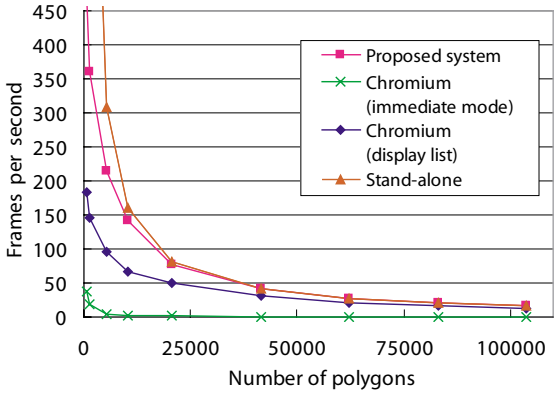


図 8: ポリゴン数とフレームレートの関係

提案システム，Chromium 共に，スタンドアロンで実行した場合と比較すると性能の低下が見られるが，これらのシステムで描画しているのはスタンドアロンよりも広視野の画像であり，また，解像度も 24 台分になっていることに留意しておく必要がある．

提案システムを用いた場合，フレーム毎に同期処理が発生するため，フレームレートが高いときはそれだけ単位時間あたりの同期回数が多くなり，同期処理のオーバーヘッドが大きく発生している．しかし，ポリゴン数が増加し，フレームレートが低くなるにつれてスタンドアロンの性能に近づいている．これは，ポリゴン数の増加により描画負荷が高まることで，同期処理に要する時間よりも描画処理に要する時間の方が支

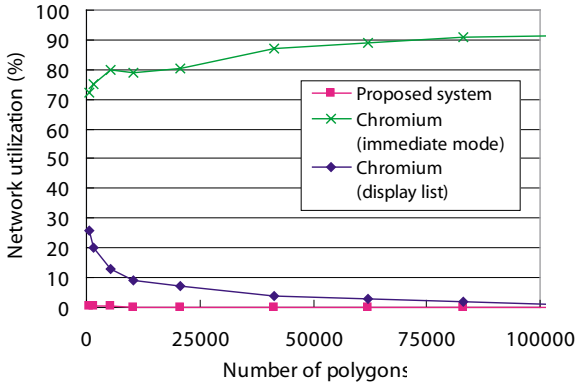


図 9: ポリゴン数とネットワーク使用率の関係

配的になるためである．ポリゴン数が約 2 万を超えると，スタンドアロン環境の性能とほぼ等しくなっている．高品質な映像を表示する VR アプリケーションでは，描画シーンが 2 万ポリゴン以上になることも多く，このような利用環境においては，スタンドアロンと同等の描画速度が得られるといえる．同期処理が必要となる API は，1 フレームあたり約 2 回呼び出されるが，1 回の同期に必要なデータ量は 4 バイトから数十バイトと非常に小さいため，ネットワーク使用率は低く，最大でも 0.65%であった．

Chromium は，イミディエート・モードで実行したときとディスプレイ・リストを使用したときで大きく結果が異なる．イミディエート・モードで実行した場合は，スタンドアロン環境と比べて実行速度の低下が著しく，実用的な性能は発揮できていない．これは，イミディエート・モードでは，描画命令が呼び出される度にサーバに描画命令を送信する必要があるので，ネットワーク負荷が高くなり，帯域がボトルネックになってしまうからである．一方，ディスプレイ・リストを利用する場合は，描画サーバ側に描画命令が保存され，描画時はディスプレイ・リストの識別子を送信するだけでよい．そのため，ネットワーク使用率を低く抑えることができ，提案システムで実行した場合と同様に，ポリゴン数が増加するにつれてスタンドアロン環境の性能に接近する傾向が見られる．

Chromium には，各描画サーバの視錐台に含まれる描画命令のみを配信することで，描画負荷を分散するとともにトラフィック流量を最適化するアルゴリズムが実装されている．しかし，D-vision のスクリーンは，ステレオ立体視のために視錐台が大きく重なる領域があるため，同一の描画命令を複数の描画サーバに送信する必要があり，このアルゴリズムは効果的に作用していない．従って，Chromium は，イミディエート・モードでのステレオ立体視を行なう場合には不向きであるといえる．

### 4.2.2 同期回数による性能の変化

提案システムで実行したときに，スタンドアロンで実行したときと比較してどれだけ性能が低下するかは，描画するポリゴン数よりも，単位時間あたりの同期回数によって影響を受けると考えられる．そこで，提案システムを用いた場合の性能の傾向を考察するために，同期回数をパラメータとした性能のモデル化を行なった．ここでは，クラスタ環境で実行するときに生じるオーバーヘッドがアプリケーションの実行時間に加わっても，アプリケーションの状態遷移に影響を与えず，スタンドアロン実行時と同様に処理が進められると仮定する．今回実験に用いたアプリケーションは，この仮定を満たしている．

API アダプタが API の呼び出しに介入する際のオーバーヘッドは数ナノ秒程度であり，また，API アダプタ内で行なう処理に要する時間は，同期処理のための通信時間を除くと無視できる程度の短い時間である．そのため，提案システムで実行した場合の処理時間は，スタンドアロンで実行した場合と比較して，同期処理に要する時間分だけオーバーヘッドが加わると近似することができる．同期をとりながら実行する場合，最も遅いノードがボトルネックとなることを考慮すると，式 (1) のように実行時間をモデル化できると考えられる．

$$T_{cluster} = \max T_{stand-alone_i} + N_{sync} \times T_{sync} \quad (1)$$

$T_{cluster}$ : クラスタ環境で実行したときの 1 フレームを描画する時間
$T_{stand-alone_i}$ : ノード $i$ において，スタンドアロンで実行したときの 1 フレームを描画する時間
$N_{sync}$ : 1 フレームの間に必要な同期回数
$T_{sync}$ : 1 回の同期に要する時間

実験に用いたアプリケーションに同期を行なう必要がある API 呼び出しを意図的に追加し，1 フレームあたりの同期回数を任意に変化させて性能を計測した．結果を図 10 に示す．このとき，実測値と式 (1) による予測値との相対誤差は，表 2 のようになった．

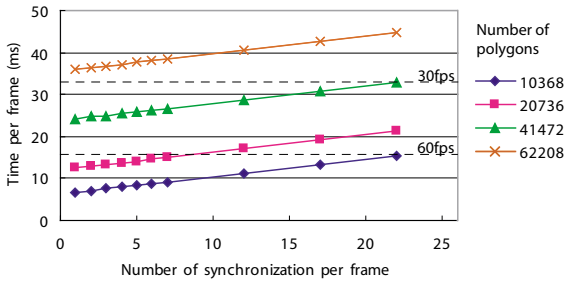


図 10: 同期回数と描画時間の関係

表 2: 実測値と予測値との相対誤差

| ポリゴン数 | 10368 | 20736 | 41472 | 62208 |
|---|---|---|---|---|
| 相対誤差 (%) | 1.159 | 0.525 | 0.338 | 0.288 |

ここで生じている実測値と予測値との誤差は，各ノードの実行タイミングが完全に一致しないために発生する待ち時間に因るものではないかと考えられるが，相対誤差は大きいときでも 1%程度であり，対象としたアプリケーションに関して十分な精度で予測できているといえる．

この結果から，ユーザ入力がフレームレート程度の更新周期で発生する場合は，スタンドアロンで実行したときと比べて性能の低下は小さいといえる．一方，高更新周期の入力デバイスを扱うアプリケーションでは，同期回数が多くなるので，性能が大きく低下すると予測されるが，このようなアプリケーションは事実上少ないため，一般的な用途では問題とならないと思われる．また，同期する必要のあるAPIを，短時間に繰り返し呼び出すアプリケーションについても，同期処理のオーバーヘッドが大きくなる可能性があるが，完全に同期をとらずに，API呼び出しの結果を補間して同期回数を減らすといった方法により，性能を向上できるのではないかと考えられる．

### 4.3 応用実験

応用実験として，市販の1人称視点型シューティングゲームであるQuakeIII [8] を提案システムとChromiumを用いて実行し，描画速度とネットワーク負荷について比較した．QuakeIIIは，基礎実験で実行したアプリケーションとは異なり，ゲーム内の仮想世界を移動したりアイテムを使用するなど，ユーザとのインタラクションが頻繁に発生する．このアプリケーションは市販されているパッケージソフトであるため，ソースコードは提供されていないが，提案システム，Chromium共に，人手によるアプリケーションの修正作業を必要とせずにクラスタベースの没入型ディスプレイに対応できることが確認できた．このときの結果を図11に示す．ここで使用したシーンは，全体で1万ポリゴン程度のものである．

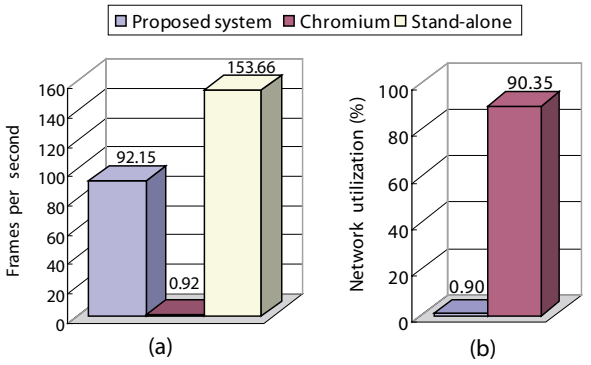


図 11: QuakeIII実行時の性能: (a) フレームレート，(b) ネットワーク使用率

提案システムを用いたとき，システムイベントやシステム時間を取得するAPIなど同期をとらなければならないAPI呼び出しが，ゲームプレイ中に，1フレームあたり約15回程度発生する．1回の同期に要するデータ量は非常に小さくて済むため，ネットワーク帯域はボトルネックにならないものの，繰り返し発生する通信のレイテンシによって，フレームレートはスタンドアロンと比較して40%低下している．しかし，90fps以上のフレームレートで動作しており，十分実用的な速度が確保できているといえる．

QuakeIIIはディスプレイ・リストを使用せず，イミディエート・モードで動作するアプリケーションであるため，Chromiumを用いた場合，フレーム毎に描画命令を送信する必要があり，ネットワーク帯域がボトルネックになっている．その結果，ユーザがインタラクション可能なフレームレートでは，実行できなかった．

## 5. まとめ

本研究では，クラスタベースの没入型ディスプレイ上で既存のアプリケーションを実行することができるアプリケーション・プラットフォームを提案した．このアプリケーション・プラットフォームは，ネットワーク負荷を低く抑えることができるMaster-Slaveモデルで構成するとともに，APIアダプタを導入することによって，透過的にアプリケーションをクラスタ環境に対応させることができる．提案システムを没入型ディスプレイ上で構築し，評価実験を行なった結果，Client-Serverモデルである Chromiumと比較すると，ネットワーク負荷を低く抑えることができ，各ノードが持つグラフィックス性能を十分発揮できることが分かった．また，提案システムでは，同期処理を行なう回数が実行性能に影響を与えるため，同期回数をパラメータとした性能のモデル化を行ない，基礎実験で用いたアプリケーションにおける同期回数と性能の関係について示した．

今後は，1kHzで更新する力覚提示装置を使用するアプリケーション等，同期回数が多いと考えられるアプリケーションに対して，必要な同期回数の計測を行ない，提案システムで対応可能であるか検討していきたい．

## 参考文献


[1] O. G. Staadt, J. Walker, C. Nuber, and B. Hamann: “A Survey and Performance Analysis of Software Platforms for Interactive Cluster-Based Multi-Screen Rendering”, Proceedings of IPT/EGVE 2003, 2003.

[2] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, J. T. Klosowski: “Chromium: A Stream Processing Framework for Interactive Rendering on Clusters”, Proceedings of SIGGRAPH 2002, 2002.

[3] G. Voß, J. Behr, D. Reiners, and M. Roth: “A multi-thread safe foundation for scene graphs and its extension to clusters”, Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization, 2002.

[4] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, C. Cruz-Neira: “VR Juggler: A Virtual Platform for Virtual Reality Application Development”, IEEE VR, 2001.

[5] CAVELib: http://www.vrco.com/

[6] Y. Chen, H. Chen, D. W. Clark, Z. Liu, G. Wallace, K. Li: “Software Envirionments For Cluster-based Display Systems”, IEEE/ACM International Symposium on Cluster Computing and Grid, 2001.

[7] The OpenGL Utility Toolkit: http://www.opengl.org/resources/libraries/

[8] QuakeIII Arena: http://www.idsoftware.com/games/