計算機アーキテクチャ 132－8
システムソフトウェアとオペレーティング・システム 80－8
ハイパフォーマンスコンピューティング 75－8
（1999．3．4）

分散並列処理のためのプラットフォーム *Lemuria* のWindows上での構成と性能評価

近藤 照之† 斎藤 彰一†† 大久保 英嗣†††

†立命館大学大学院理工学研究科
††和歌山大学システム工学部情報通信システム学科
†††立命館大学理工学部情報学科

分散共有メモリは，プロセスの仮想アドレス空間に存在するメモリの一部を，1台の計算機内のプロセスだけでなく，他の計算機のプロセスと共有する機能である．*Lemuria*は，この分散共有メモリ機能を提供する分散並列処理のプラットフォームである．*Lemuria*が他の分散共有メモリと異なっている点は，広域分散環境を考慮して，計算機クラスタを階層化し，それに対応した一貫性制御を行う方式を採用していることである．本論文では，*Lemuria*のWindowsへの移植の手法と性能評価について述べる．

Lemuria: An Implementation and Performance Evaluation of Distributed Shared Memory on Windows

Teruyuki Kondo† Shoichi Saito †† Eiji Okubo†††

†Graduate School of Science and Engineering, Ritsumeikan University
1-1-1 Noji-Higashi, Kusatsu, Shiga 525-8577, Japan

††Department of Computer and Communication Sciences,
Faculty of Systems Engineering, Wakayama University
930 Sakaedani, Wakayama City, 640-8510 Japan

†††Department of Computer Science,
Faculty of Science and Engineering, Ritsumeikan University
1-1-1 Noji-Higashi, Kusatsu, Shiga 525-8577, Japan

A distributed shared memory(DSM) system enables processes on different machines to share a virtual memory. *Lemuria* is a platform which is based on DSM for distributed parallel processing. *Lemuria* is different from conventional DSM and aims at utilizing DSM in wide area distributed environments. For this purpose, *Lemuria* is constructed on layered computer clusters, and adopts a new memory consistency protocol for those clusters. This paper describes an implementation and performance evaluation of *Lemuria* on Windows.

1 はじめに

分散共有メモリは, ノード内のみならずネットワークを介したノード間でもプロセスのメモリを共有をするための機構である. 我々は, 大規模な分散並列処理環境や広域分散環境において, データを計算機間で共有する手段として分散共有メモリが有効であると考えている. 我々が開発を進めている*Lemuria*は, このための分散共有メモリ機能を提供する分散並列処理のプラットフォームである.

*Lemuria*は, パーソナルコンピュータ(PC)やワークステーション(WS)などの一般的な計算機を対象としている. *Lemuria*では, これらの計算機からなる小規模な計算機クラスタをLemuriadと呼ばれるサーバによって管理している. さらに, 複数のLemuriadをReflectorと呼ばれる上位のサーバによって管理することにより階層的な計算機クラスタを構成している. このように, *Lemuria*では, 小規模な計算機クラスタを階層的に結合することにより, 大規模並列処理や広域分散環境において共有メモリを実用的なものとしている.

*Lemuria*が他の分散共有メモリシステムと大きく異なる点は, 階層的な計算機クラスタの上で分散共有メモリを実現している点である. 階層的な構成をとることで, 分散共有メモリ機能を実現するための通信量を抑制することができ, 数百台規模の計算機群においても分散共有メモリを構築することが可能となる. また, *Lemuria*では, 階層的な構造を利用して, 通信のキャッシングや通信の取りまとめを行うことによって, 分散共有メモリアクセスの高速化を図っている[1].

これまで, *Lemuria*は, Unix上で開発を行ってきた. 分散共有メモリはデータ共有の手段であるため, 異機種間でデータを共有することが重要である. そのため, より多様な環境で*Lemuria*を動作させるために, 今回Windows上への移植を行った. 本論文では, *Lemuria*をWindows上へ移植するための手法とその性能評価について述べる.

以下, 本論文では, 2章で*Lemuria*の構成と特徴について述べ, 3章で*Lemuria*の処理方式, 4章で*Lemuria*のWindows上での構成, 5章で性能評価について述べる.

2 *Lemuria*の構成と特徴

*Lemuria*は, PCやWSなどの一般的な計算機によって構成される計算機クラスタにおいて, 分散共有メモリ機能を中心とした, 分散並列処理のための機能をユーザに対して提供する.

2.1 *Lemuria*の構成

*Lemuria*は, 分散共有メモリ機能をユーザに提供するライブラリであるlibLemuriaと, Lemuriad及びReflectorと呼ばれる2種類のサーバから構成されている.

- **libLemuria**
  *Lemuria*の機能を提供するC言語用ライブラリである. ユーザプログラムにリンクして使用する.
- **Lemuriad**
  クラスタごとに配置される*Lemuria*のサーバプロセスである.
- **Reflector**
  複数のLemuriadを管理する上位のサーバプロセスである.

2.2 階層化クラスタ

*Lemuria*の最も大きな特徴は, 階層化されたクラスタ上に, 分散共有メモリを実装している点である. *Lemuria*は, 複数の小規模な計算機クラスタを階層的に結合することによって, 大規模な計算機クラスタを構成している. *Lemuria*におけるクラスタ階層を図1に示す. 図1に示すように, クラスタは以下の3層からなる.

- **クライアントクラスタ**
  複数のクライアントノードとそれを管理する1つのLemuriadから構成される.
- **サーバクラスタ**
  複数のLemuriadとそれらを管理する1つのReflectorで構成される.
- **ルートクラスタ**
  複数のReflectorで構成される.

*Lemuria*では, クライアントノードを数台ごとのクライアントクラスタに分け, このクライアントクラスタごとにLemuriadを配置する. Lemuriadは, 主に, このクライアントクラスタ内の制御と, 自クラスタのクライアントノードと他クラスタとの通信の中継を行う. 上位サーバであるReflectorは, Lemuriad間の通信の管理やReflector間の通信を行う. ルートクラスタの管理は, それぞれのReflectorで分散して行われる.

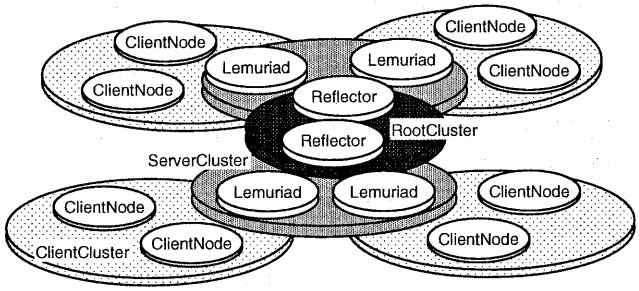

図 1　*Lemuria*の全体構成

2.3　階層化の利点

*Lemuria*が階層化されたクラスタ上に, 分散共有メモリを実装することによる利点を以下に示す.

- 通信量の削減
  Lemuriad と Reflector は, 共有メモリの更新時にコピーの更新の通知を行わず, メモリアクセス要求が発生したときにデータを送信する. これにより, 計算機間の不要な通信が発生しないため, 通信量が削減される.
- クライアント情報の削減
  クライアントノードは, 自クライアントクラスタの情報を保持していればよい. 他クラスタの情報は, Lemuriad や Reflector が提供する. 従って, クライアントの持つべき情報量が削減される.
- 大規模分散処理
  階層的な一貫性制御を行うことにより, 通信量が, ノード数ではなくクラスタ数に比例するため, 大規模分散処理に対応することができる.
- 通信方式が変更可能
  Reflector 間の通信のみを広域分散用のプロトコルに変更することにより, 広域分散環境にも対応するなど, 階層ごとに通信方式を変更することができる.

3　*Lemuria*の処理方式

*Lemuria*で分散共有メモリ機能を実現するための処理方式について述べる (表1 参照).

3.1　*Lemuria*の通信方式

一般的な分散共有メモリでは, 共有メモリを持つ各ノードが直接通信を行う. *Lemuria*では, 階層化を前提としているためクライアントノードは自クラスタのみ直接通信を行い, 異なるクラスタに属するノードへの通信は Lemuriad や Reflector を介して行う. このような通信方式をとることにより, 通信量削減などの階層化の利点を得ることができる.

表 1　*Lemuria*の特徴

実現方式	ソフトウェア
一貫性制御方式	CRC (Write Shared の RC を階層化に対応させたもの)
一貫性制御の単位	ページ
通信プロトコル	UDP
対象システム	階層化による大規模分散システム/広域分散システム

3.2　一貫性制御方式

*Lemuria*における共有メモリの一貫性制御は, Cluster based Release Consistency(以下 CRC と呼ぶ) [2] を Write-Shared 方式によって実現している.

CRC は, Release Consistency(以下 RC と呼ぶ) を階層化された計算機クラスタに適応するように拡張したものである. RC では, 臨界領域を出た後と, バリア同期を行うときに, すべての計算機が一貫性制御の対象となる. CRC では, クライアントクラスタ内に存在するクライアントプロセス同士は RC を用いて一貫性制御を行う. クラスタ間で一貫性制御を行う必要がある場合は, Lemuriad に対しても同様の一貫性制御の依頼を行う. この方式では, クラスタ外のクライアントノードは, データが必要なときに, Lemuriad を介して最新のデータを取得すればよい. Lemuriad 間及び Reflector 間でも以上と同様の一貫性制御を行うことにより, クラスタ全体の一貫性を保つことができる. CRC で

は，さらに，LemuriadとReflector間の更新差分の中継を遅延評価することにより，通信量の削減を図っている．

3.3 排他制御とバリア同期

*Lemuria*における排他制御は同期変数の確保アクセス(acquire access)と解放アクセス(release access)を使用して行われる．同期変数ごとに設定されるアクセス権を各ノードが受け渡すことにより，排他的なアクセスを保証している．

バリア同期はRelflectorにより一括管理される．バリア同期の処理の流れを以下に示す．

(1) クライアントノードは，バリア同期をLemuriadに要求する．

(2) Lemuriadは，クライアントクラスタ内のバリア要求がすべて終了すると，Reflectorにバリア同期の要求を行う．

(3) Reflectorは，すべてのLemuriadからの同期要求を受信した後に当該バリアを管理するReflectorに同期要求を行う．

(4) バリアを管理するReflectorがすべての同期要求を受信した時点でバリア同期が成立する．

3.4 共有メモリアクセスの高速化手法

*Lemuria*では，共有メモリアクセスの高速化のためにLemuriadとReflectorにおいて以下の機能を実装している．

- サーバでのページのキャッシュ
  ページの移動と更新差分の移動を中継する時にページ内容のキャッシュを行う．
- サーバでの通信の取りまとめ
  各種情報の転送を取りまとめることで通信回数と通信量を削減する．
- 遅延評価
  バリア同期の要求や更新差分の中継をすぐに行わず，要求が発生するまで保持して通信の取りまとめを行う．

4 Windows上での構成

*Lemuria*は，これまでUnix上で開発を行ってきた．今回，より多様なプラットフォーム上で*Lemuria*を動作させるために，一般的なPCで動作するOSであるWindows NT 4.0およびWindows 98上への移植を行った(以下Windows NT 4.0およびWindows98をWindowsと呼ぶ)．本章では，*Lemuria*をWindows上に移植する際の問題点と実装方法について述べる．

*Lemuria*は，すでにSolaris2, IRIXなどのUnix上で動作している．しかし，WindowsとはAPIが異なるため，プログラムインタフェースの変更が必要となる．以下にそれらについて述べる(表2参照)．

4.1 ソケット

*Lemuria*では，分散共有メモリを実現するためにUDP上に独自のプロトコルを実装することによって，*Lemuria*内部の通信を行っている．そのため，UnixではBSD socketを用いて通信を行っている．Windows上でも，Winsockによって，BSD socketを利用することができるが，*Lemuria*の性能向上のために重複I/OなどのWinSock2.0固有の機能を利用した．そのため，*Lemuria*をWindows上で動作させるにはWinSock2.0のサポートが必要となる．

4.2 マルチスレッド

*Lemuria*は，マルチスレッドで記述されている．そのため，ユーザプログラムを作成するには，マルチスレッド用のライブラリをリンクする必要がある．Unixでは，POSIX準拠のpthreadを利用していたが，Windowsでは，マルチスレッドに対応しているWin32APIを利用している．pthreadとWin32APIは，同期やスレッド制御などのマルチスレッドに関する基本的な機能は変わらないため，プログラムインタフェースの変更のみを行った．

4.3 メモリアクセスの検出

ページを基本とした分散共有メモリでは，ページの内容の一貫性を維持するために，メモリのアクセス制御機能を利用してページアクセスを検出する．Unix上で，ページアクセスを検出する手順を以下に示す．

(1) アクセス禁止状態のページへアクセスを行うことによって，セグメンテーションフォルトが発生する．

(2) セグメンテーションフォルトのシグナルを取得し，シグナルハンドラによって，アクセス

表 2　*Lemuria*の変更点

比較項目	Unix	Windows
ソケット	BSD socket	WinSock2.0
マルチスレッド	pthread	Win32 マルチスレッド
メモリアクセスの検出	シグナル	構造化例外処理
メモリ保護	mmap() + mprotect()	VirtualAlloc()+VirtualProtect()
サーバプロセス	daemon	サービス

が読み出しか, 書き込みかを判断し, 当該仮想アドレスを取得する.

Windowsでは, シグナルによってセグメンテーションフォルトを検出することができない. しかし, 代わりに構造化例外処理を用いてセグメンテーションフォルトを検出することができる[3]. また, このときアクセスの種類(読み出し/書き込み)と仮想アドレスも取得することができる. よって, 本実装には, 構造化例外処理を使用した.

4.4　メモリアクセス制御

*Lemuria*では, メモリのアクセス制御を利用することにより, 各ノードのプロセスがそのメモリ内容が最新かどうかを判断する. Unixでは, mmap()およびmprotect()システムコールを用いてメモリのアクセス制御を行っている. Windowsでは, VirtualAlloc()とVirtualProtect()を用いて同様のメモリのアクセス制御を行っている.

4.5　サーバプロセス

Unixでは, daemonとしてプロセスを簡単にバックグラウンドで動作させることができる. Windowsでは, 普通のプログラムのままでは, ユーザがログオフした時点でプログラムが停止してしまう. しかし, Windows NT 4.0では, サービスを利用することにより, プロセスをバックグラウンドで動作させることができる. そこで, Windows NT 4.0上では, サービスプロセスとしてLemuriadとReflectorが動作するように実装を行った.

5　性能評価

本章では, *Lemuria*のWindows上とSolaris2上での性能評価について述べる.

表 3　評価環境

比較項目	Unix	Windows
CPU	Intel PentiumII 266MHz	
Memory	64MB	
Network	100Base-TX Ethernet	
Products	富士通製	日立製
OS	Solaris2.5 for x86	Windows NT 4.0
PageSize	8192バイト	

5.1　評価環境

評価に使用した計算機は, Solaris2.5 for x86を搭載した, PentiumIIプロセッサ266MHz, メモリ64MBのPC互換機(富士通製)と, Windows NT 4.0を搭載した, PentiumIIプロセッサ266MHz, メモリ64MBのPC互換機(日立製)を用いた. ネットワークは100Base-TXで接続されている. 評価環境を表3に示す.

5.2　評価アプリケーション

評価アプリケーションには, Splash2[4]より, LU分解(Non contiguous block allocationバージョン)を使用した. 評価に用いた行列の大きさは, 1024×1024である. また行列を分解するブロックの大きさは, 16×16である. 評価は, 最大32ノードでアプリケーションを実行させ, LemuriadおよびReflectorは別のノードで実行した. また, 8ノードと16ノードの実験では, クライアントクラスタの数を変更して評価を行った.

5.3　評価結果

*Lemuria*のWindowsとSolaris2上での評価結果を表4に示す. ただし, ノード数は, Reflector及びLemuriadの台数を除いたクライアントのノード数である. また, Reflector及びLemuriadは, クライアントノードと同じ仕様の計算機で実行している. 表4 におけるFormは, クライアントクラス

表 4　LU分解の実行結果 (単位：秒)

Size	1024×1024							
Nodes	1	2	4	8		16		32
Form	1	2	4	4+4	8	4+4+4+4	8+8	8+8+8+8
Solaris2.5 for x86	357	211	145	97	148	71	80	181
Windows NT 4.0	336	213	148	98	538	98	405	728

タの形態を表している．4+4となっている場合は，Lemuriad 1ノード と クライアント 4ノードで構成されるクライアントクラスタ2つをReflectorで結合している．

LU分解では，Solaris2とWindowsとも 4台までは台数による計算の高速化が認められる．また，1クラスタあたりのノード数が4ノードの場合は，クライアント全体のノード数がWindowsでは 8ノード，Solaris2では16ノードまで台数による高速化が認められる．しかし，Solaris2とWindowsとも16台，4クラスタを境として急激に計算速度が低下した．これは，Lemuriadにおけるメモリ不足により，ページングが急激に増加したことが原因である．また，クライアント数が8と16ノードの場合は，クラスタを分割した場合に高い性能を得た．これは，クラスタを分割した場合の方が，Lemuriadへの通信量が少なく，また使用するメモリの量も少ないため，Lemuriadでの負荷とページングの発生の両方が低く抑えられたためである．

基本的にSolaris2とWindowsでは，同程度に実行速度が推移しているが，Windows の方が少ないノード数で実行速度が低下している．評価結果をまとめると，以下のようになる．

- WindowsとSolaris2とも *Lemuria* を使用することにより並列処理による性能向上が認められる．
- WindowsとSolaris2では，ほぼ同程度の性能を得ることができる．
- より大規模な並列処理を行うためには，サーバでのページングによるボトルネックを回避する必要がある．但し，この場合は，計算機クラスタを分割することにより，サーバの性能低下を抑えることができる．
- Windowsの方が，Solaris2に比べメモリの使用量が多くなりページングが少ない台数で頻繁に発生する．

6　おわりに

本論文では，分散並列処理のためのプラットフォーム *Lemuria* のWindows上での実装とその性能評価について述べた．性能評価により，*Lemuria* がPC上のWindows上でもWSとほぼ同程度の性能を得ることがわかった．またWindowsでは，メモリの使用量が多いため，少ない台数でページングが起こりやすく，これが性能に影響していることがわかった．今後は，以下の点について検討する予定である．

- UnixとWindows間のメモリ共有などの異機種環境での *Lemuria* の評価．
- サーバにおける高負荷時の対策と使用メモリの削減．
- クラスタベースの一貫性制御方式の改良．
- *Lemuria* を用いたプログラム開発の容易性の向上．

参考文献

[1] 斎藤 彰一, 國枝 義敏, 大久保 英嗣: "分散並列処理のためのプラットフォーム *Lemuria* における分散共有メモリの性能評価", 電子情報通信学会論文誌 (採録決定).

[2] 斎藤 彰一, 國枝 義敏, 大久保 英嗣: "広域分散環境における分散共有メモリの実現とその性能評価", 情報処理学会論文誌 (投稿中).

[3] W. Speight and J. Bennett, *"Brazos: A Third Generation DSM System"*, Proc. of the USENIX Windows NT Workshop (1997).

[4] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh and A. Gupta *"The SPLASH-2 Programs: Characterization and Methodological Considerations"*, Proc. of the 22nd International Symposium on Computer Architecture, pp.24-36 (1995).