Java 向けソフトウェア分散共有メモリの実現

早田 恭彦[†1,†3] 中田 秀基[†2]
小川 宏高[†1] 松岡 聡[†1,†4]

近年のハードウェアの急速なコモディティ化とネットワーク技術の発展により，Grid Computing に代表される Wide Area HPC が盛んに研究されている．異なるアーキテクチャの計算機で構成される Grid Computing において，プラットフォームポータビリティとパフォーマンスポータビリティを満たすには Java 言語を利用するのが一つの適当な手法である．しかし，Java をコモディティクラスタ上で動作させるためには，分散共有メモリを実現する必要があるが，既存の研究では JVM への変更が必要であり，ポータビリティが考慮されていない．そこで，本研究では Java のポータビリティを損なわないソフトウェア分散共有メモリのソフトウェアアーキテクチャを考察し，プロトタイプシステム JDSM を作成した．このソフトウェア分散共有メモリを異なる通信インターフェースを用いてコモディティクラスタ上で性能評価を行い，C 言語での実現に比べメモリコンシステンシ処理が大きいことなどが分かった．

Implementation of Software DSM in Java

Yukihiko Sohda,[†1,†3] Hidemoto Nakada,[†2] Hirotaka Ogawa[†1]
and Satoshi Matsuoka [†1,†4]

Rapid commoditization of advanced hardware and progress of networking technology is now making wide area high-performance computing a.k.a. the 'Grid' Computing a reality. Since a Grid will consist of vastly heterogeneous sets of compute nodes, especially commodity clusters, some have articulated the use of Java as a suitable technology to satisfy portability across different machines. Since Java's natural model of parallelism is shared memory multithreading, one will have to support distributed shared memory (DSM) in a portable manner; however, none of the previous work on implementing Java on DSM has been portable solution. Instead, we propose a software architecture whose goal is to achieve portability of DSM implementations across different commodity clustering platforms, and implemented a prototype system JDSM. Benchmark results show that the current implementation on Java incurs increased memory coherency maintenance cost compared to C-based DSMs, thus limiting scalability to some degree, and we are currently working on a solution to alleviate this cost.

1. はじめに

近年のハードウェアの急速なコモディティ化とネットワーク技術の発展により，Grid Computing に代表される Wide Area HPC が盛んに研究されている．異なるアーキテクチャの計算機で構成される Grid Computing では，プログラムがネットワークからダウンロードされ，実行できるという**プラットフォームポータビリティ**を確保する必要がある．これに加えて，異なる環境でも十分な性能を達成できる**パフォーマンスポータビリティ**も確保されなければならない．このような要件には，コード表現レベルでのポータビリティと JIT コンパイラによる最適化を組み合わせた言語処理系が適当であり，現状では Java が最も有力であると考えられている．

しかし，Java の実行系である JVM(Java 仮想マシン) は共有メモリマシンでの動作を前提としているため，PC クラスタをはじめとする分散メモリ型並列計算機上でマルチスレッドで動作させることができない．そこで JVM に対して何らかの形で分散共有メモリシステム (DSM) の機構を提供する必要がある．このような研究[1),2)] はいくつかなされているが，いずれのシステムも JVM への変更を必要としており，プラットフォームポータビリティが考慮されていない．さらに，この JVM への変更により，JIT コンパイラなどの既存の資源を利用できず，十分な性能を達成できていないため，

†1 東京工業大学 情報理工学研究科 数理・計算科学専攻
Tokyo Institute of Technology
†2 電子技術総合研究所
Electrotechnical Laboratory
†3 日本学術振興会特別研究員
JSPS research fellow
†4 科学技術振興事業団
Japan Science and Technology Corporation

ポータブルな高性能計算環境は実現できない．

そこで，本稿ではコモディティクラスタを対象とした，Java 言語へのポータブルかつ PureJava であるソフトウエア分散共有メモリシステムの実現について述べる．このとき，メモリコンシステンシ処理などのアルゴリズムをプラットフォームの特性に応じて柔軟に変更・適用可能にするにはどのようなアーキテクチャを採るべきかまた通信などの各レイヤにおける実装が性能にどのような影響を及ぼすかについて述べる．さらに，この分散共有メモリシステムを 32 台構成の PC クラスタ上で，複数の処理系，Socket・VIA・PM の異なる通信インターフェースを利用して性能評価を行った．この結果，Java 言語で実現されたソフトウェア DSM システムの有効性を確認するとともに，C 言語でのソフトウェア DSM システムに比べ，メモリコンシステンシ処理のオーバーヘッドが大きいこと，これに伴い負荷の不均衡が生じ，実行時間を増大させてしまうことが分かった．

2. JDSM システム

2.1 Java 上でのソフトウェア DSM の設計

Java におけるソフトウェア DSM では，Java の言語仕様の制限[3),4)] などから，従来 C 言語などで行われてきた DSM の実現手法が必ずしも利用できるとは限らない．特にメモリ管理・プログラミングモデルについて，Java 言語の仕様を考慮した設計を行う必要がある．

2.1.1 メモリ管理

DSM でのメモリ管理においては，メモリの管理単位・コンシステンシモデルの選択が重要である．

まず，メモリ管理の単位として Page-Base と Object-Base の選択肢がある．Page-Base で実現する場合，DSM のメモリ管理機構は Java の処理系である JVM 内のメモリへ自由にアクセスできる必要がある．しかし，Java においてはメモリ管理は JVM が行うため，JVM 上で動作する Java プログラムからの操作が不可能である．このため Page-Base を実現する場合，JVM への改変が不可避となる．また，ガーベッジコレクション (GC) によるメモリの移動の危険性も考慮する必要がある．一方，Object-Base で実現する場合，メモリ管理の単位を Java のオブジェクト単位で行う方法が最も自然であると考えられる．Java のオブジェクト単位で管理することによって，Java プログラムからメモリ管理を行うことができ，JVM への改変や GC の影響を考慮する必要がない．また，ポータビリティの点からも JVM への改変の不要な Object-Base が有利である．

次に，メモリコンシステンシモデルとして多くのモデルが考えられるが，現状では Java における DSM の実現に適したモデルが不明であり，一般的には最適なコンシステンシモデルはアプリケーションや実行環境に依存する．そこで，異なるメモリコンシステンシモデルやプロトコル (特に C 言語で比較的有効とされている Lazy Release Consistency モデルや Invalidate, Update プロトコルなど) を容易に実現できるフレームワークに則って実現されることが望ましい．これによって，アプリケーションや実行環境に合わせたメモリコンシステンシプロトコルの利用が容易になる．

2.1.2 プログラミングモデル

対象とするプログラムは Java マルチスレッドプログラムである．このプログラムをクラスタ型並列計算機などの分散メモリ環境上で動作させるには，DSM 機構を利用するプログラムへ変換する必要がある．ここではどのオブジェクトが共有されるか，どのようなアクセスが行われるか，さらにスレッドの生成やスレッド間の関係などを解析し，メモリコンシステンシ処理の追加などの適切な変換を行う必要がある．しかし，一般的なマルチスレッドプログラムに対してこのような解析を行い，実行効率の良い変換を行うのは難しい．また，プログラムに対して直接の変換を行わず，JVM 内の処理で一般のマルチスレッドプログラムをサポートするようなシステム (e.g. cJVM[2)]) では十分な性能を発揮できていない．

そこで我々は，科学技術計算をはじめとする多くの並列プログラムは SPMD に従っており，また，DSM のアルゴリズムも基本的に SPMD で効率よく動作するようになっていることに注目し，対象とするプログラムを SPMD スタイルのマルチスレッドプログラムに限定することにする．これにより，従来の科学技術計算アプリケーションの本システムへの移植性が高まるとともに，高い性能を達成することが期待される．さらに，Java 用の OpenMP(e.g. JOMP[5)]) をクラスタ型並列計算機上にポータブルに実現するためのベースプラットフォームとして用いることが可能である．

また，共有メモリ (オブジェクト) へのアクセスチェックを行う方法として，Master-Proxy モデルを用いる方法とアクセスの前後にアクセスチェックの命令を挿入する方法が考えられる．Master-Proxy モデルの場合，実体オブジェクトとインターフェースが同一な Proxy オブジェクトだけをユーザに提示することで、実体オブジェクトのノード間での移動を隠蔽する．このためアクセスチェックがメモリアクセス内部で行なわれ，全体的なプログラム変換のコードの増加量を減らすことができる．しかし，連続アクセス時のアクセスチェックの除去などの最適化が行えない，Proxy クラスを用意する必要がある，配列オブジェクトのアクセスをメソッド呼び

出しに置き換える必要があるなどの欠点がある．逆に，メモリアクセスの前後にアクセスチェックを挿入する場合，最悪全てのメモリアクセス部分を変換する必要があるが，不要なアクセスの除去などの最適化が行えるなどの利点もある．

さらに，クラスタなどの各ノードでJVMが起動されるため，共有されるオブジェクトが各ノードで同一のものとして見えている必要がある．同じJVM内であればHash値で同一性を確認できるが，異なるJVMの場合はHash値での同一性は保証されない．オブジェクトの同一性を提供する方法は，一般のマルチスレッドプログラムを分散メモリ環境で動作させる場合は困難である．しかし，SPMDスタイルのマルチスレッドプログラムの場合，オブジェクトの生成順序が各ノードで同じであるため，オブジェクトを生成順に登録していくことによって，順番をキーとしてオブジェクトの同一性を容易に保証できる(ただし，条件分岐ブロックなどで生成されたオブジェクトの扱いを考慮する必要がある)．

2.1.3 実装上の選択

さらにこれら以外にも，共有するオブジェクトの選別方法，オブジェクトのコピー(移送)，メソッド呼び出し，配列オブジェクトやオブジェクト型でないPrimitive型の扱いなど実装上の検討すべき点がある．

- 共有するオブジェクトの選別方法としては，プログラム中で生成される全てのオブジェクト，プログラムの初期化プロセスで生成されるオブジェクト(つまり並列実行中に生成されたオブジェクトは共有しない)，プログラム中で必ず共有されるオブジェクトを対象とする方法などが挙げられる．
- ノード間でのオブジェクトのコピー(移送)には，オブジェクトのシリアライズを利用するのが一般的であるが，Serializableでないオブジェクトをコピーしたい場合，自前のシリアライザが必要となる．
- 配列オブジェクトなどのFalse-Sharingを引き起こす可能性の高い巨大なオブジェクトを共有する場合，適切なサイズに分割する，オブジェクト中のフィールドも共有するなどの方法が考えられる．
- Primitive型を共有する場合，wrapperオブジェクトを利用して共有する方法とPrimitive型のみ別の共有手段を用いるなどの方法がある．

2.2 本研究での実現方針

以上の検討を踏まえ，本稿では次に示すデザインポリシーに基づいてソフトウェアDSMシステムを実現する．

メモリ管理単位 Javaのオブジェクト単位とし，メモリコンシステンシモデルは様々なプロトコル(Strict Consistencyや Lazy Release Consistencyなど)を利用可能なフレームワークを提供する．今回はWrite-Invalidateプロトコルを利用したRelease Consistencyモデルを用いる．

対象とするプログラム SPMDスタイルのJavaマルチスレッドプログラムとする．アクセスチェックは，オブジェクトへのread/writeアクセス時に，メモリコンシステンシ処理を行うメソッドを呼び出して行う．共有オブジェクトの登録は，オブジェクト生成時に行う．また，メソッド呼び出しはメソッド呼び出し専用のチェックを用意せず，writeアクセスと同等に扱う．

実装上の選択 共有されるオブジェクトはプログラム中に現れるもののうち，実行プログラムのフィールドで宣言されているオブジェクトとし，計算スレッド内などで生成されるオブジェクトは対象外とする．オブジェクトの移送にはJava付属のシリアライザを利用する．これによりオブジェクトの一意性(`equals`など)も保証できる．ただし，実行時のオーバーヘッドが問題になる可能性もある．配列オブジェクトは適切なサイズに分割して管理可能とする．Primitive型はオブジェクト型でないため対象外とし，扱う場合はユーザがあらかじめwrapして扱うこととする．

2.3 JDSMシステム概要

前節の設計に基づいて本研究で実現するJava上のDSMシステム —JDSMシステムについて述べる☆．JDSMシステムは，JDSMランタイム(`jdsm.dsm`パッケージ)と通信インターフェース(`jdsm.comm`パッケージ)の2つから構成される．通信インターフェースでは，JDSMランタイムで利用する通信APIを定めて複数の異なる通信デバイスの利用を可能としており，その実装も与える．JDSMランタイムでは，通信部分においては通信インターフェース部で定めた通信APIに基づいて記述する．さらに，複数のメモリ管理モデルを利用できるようなAPIを定め，これに基づいてDSM処理部分を記述することによって，異なるメモリ管理モデルを同様に利用可能とする．

2.3.1 通信インターフェース

通信インターフェースの実装としては，可搬性や多く

☆ JDSMシステムで実現されるDSMのセマンティックスは現状のJavaのMemory consistencyの仕様には合わない．様々な理由により現状のJavaのMemory consistencyの仕様は並行処理に適切ではないため，Maryland大学のBill Pughらが中心となって，release consistencyに基づいた新しい仕様[6]が採用に向けて検討されている．

図 1 通信インターフェースの構成

メソッド	動作内容
initialize	通信などの初期化を行う
finalize	通信終了処理を行う
getMyNode	自分のノード番号を得る
getNumNode	プログラムの実行中のノード数を得る
send	メッセージを送信する (送信完了を待つ)
asend	メッセージを送信する (送信完了を待たない)
sendDone	asend 後，送信完了を待つ
recv	メッセージを受信する
probeRecv	メッセージが到着確認

表 1 インターフェース Comm で宣言されているメソッド

の通信インターフェースの利用を考慮し，**図 1** のような構成とした．利用する下位の通信インターフェースとして，Java が提供する Socket や，クラスタ環境で提供されている高速な通信インターフェース (VIA，PM) など多くのものがある．これらをポータブルに扱うため，通信インターフェースを共通の API インターフェースで隠蔽する．異なる通信インターフェースを利用した場合も，その通信インターフェースで API を実現することによって，アプリケーション側の変更を不要とすることができる．ただし，Java の場合はそのオーバーヘッドが問題となる．

通信インターフェース API はパッケージ `jdsm.comm` のインターフェース `Comm` で定義されている．インターフェース `Comm` では基本的な API を定義しており，**表 1** で示すメソッドが宣言されている．下位の通信レイヤはこのインターフェース `Comm` の実装を与えて利用する．

2.3.2 DSM ランタイム

JDSM システムにおける分散共有メモリでは，メモリ管理モデルの実装手段として，抽象クラス `SharedObjectPool` のサブクラス化で実装する方法とする．これによって，環境ごとや対象プログラムごとにメモリ管理モデルを異なるモデルを採用しても，その差異を吸収することができる．この `SharedObjectPool` では**表 2** で示すメソッドが宣言されており，実際のメモリ管理クラスではこれらのメソッドを実装する．なお，通信部分は通信インターフェース部で定めた通信 API を利用した実装になっている．

2.3.3 ユーザプログラム

本システムで対象とするプログラムは，SPMD スタイルのマルチスレッドプログラムである．このプログラムは SPMD 抽象クラス (**図 2**) を実装する必要がある．

メソッド	動作内容
initialize	通信などの初期化を行う
finalize	通信終了処理を行う
setComm	通信インターフェースを設定する
register	共有オブジェクトを登録する
acquire	共有オブジェクトへの排他的アクセス権を得る
release	共有オブジェクトへの排他的アクセス権を解放
update	共有オブジェクトの最新の状態を得る
acquireAsync	非同期 acquire
updateAsync	非同期 update

表 2 SharedObjectPool 抽象クラスで宣言されているメソッド

```
public abstract class SPMD{
  public Comm comm;
  public SharedObjectPool poo;
  public void initialize();
  public void start();
  public void finalize();
}
```

図 2 SPMD 抽象クラス

このSPMD抽象クラスの初期化・終了の `initialize` と `finalize` メソッドは各ノードで 1 回実行される．並列計算の本体となる `start()` メソッドは各ノードに割り当てられたスレッドの数だけ呼び出される．`register` メソッドによる共有オブジェクトの登録は `initialize` メソッド内で行われなければならない．

3. 実　　装

3.1 通信インターフェース

Socket 及び VIA，PM を用いた `Comm` インターフェースの実装について述べる．

3.1.1 SocketComm

`SocketComm` は，Java の `Socket`，`ServerSocket` を用いて実装されている．他のプロセッサ 1 つにつき，1 つの TCP/IP 接続を用いる．プロセッサが N 個である場合には，各プロセッサあたり，$N-1$ 本，全体で $N*(N-1)$ 本の接続が用いられる．

`SocketComm` では複数の接続に対して同時に読み出しを行う必要があるが，Java の Socket には Unix の select システムコールに相当する機能がない．このため各接続に対して一つずつの Thread を割り当て，個々の Thread がそれぞれの接続からの読み出しを行う．

`recv` メソッドは，それぞれの接続に到着したメッセージを読み出すが，この際 busy wait することなしに読み出せることが望ましい．このために，読み出し Thread と `SocketComm` オブジェクトの間に FIFO キューを設けた．読み出し Thread は，読み出したメッセージを FIFO キューに挿入する．`SocketComm` の `recv` メソッ

ドは，このキューからメッセージを取り出す．

キューに対するアクセスは wait/notify を用いて同期を取る．キューにメッセージがなかった場合には，recv した Thread はキューに対して wait する．接続からの読み出し Thread は，メッセージをキューに挿入すると同時に notify を行う．このように，SocketComm は busy wait を全く用いずに実現される．

3.1.2 JVIA 及び VIAComm

VIAComm では通信に Fast Ethernet，Myrinet，Gigabit Ethernet 上など多くのネットワークデバイス上に実装されている VIA(Virtual Interface Architecture)[7] を利用する．この VIA を用いることにより TCP/IP より小さいレイテンシ，高いバンド幅の通信を提供できる．しかし，Java 言語向けの VIA のインターフェースが存在しないため，JNI を経由して VIA へアクセスする JVIA を実現し，その上で Comm インターフェースの実装を行った．

JVIA は，Java から JNI を経由して VIA デバイスへ直接アクセスするのではなく，VIA の通信ライブラリ vipl の各関数を呼び出す．vipl ライブラリでは 7) によって定められた API を提供しているため，ネットワークインターフェースや実装が異なる場合でも，JVIA を変更することなく利用することができる．なお，今回は VIA の実装の 1 つである M-VIA[8] を用いる☆．

VIA の API は全て jvia.JVIA クラスにおいて native method として，VIA のデータ構造などはパッケージ jvia のクラスとして定義する．Java 側から native の VIA API を呼び出すときにデータのマーシャリングを行う必要がある．このマーシャリングのオーバーヘッドを削減するためには，静的 (JIT コンパイラのように準静的な場合もある)，もしくは動的な最適化手法が必要となる．Jaguar[10] では，バイトコード拡張に相当するアノテーションをクラスファイルに埋め込み JIT コンパイル時に最適化を行っている．JVIA ではマーシャリングデータをキャッシュするなどの動的な最適化手法を用いることで，native に近い性能を達成している (**表 4**)．

次に，この JVIA を用いた Comm インターフェースの実装である VIAComm における初期化と送信・受信の処理について述べる．

VIA は Socket と同様に Connection Base であるため，各ノードにおいてノード数分のエンドポイント (VI) を作成し，ノード間の接続を確立する (**図 3**-(1))．またポーリングを実現するために，VI での受信は 1 つの Completion Queue(CQ) に，送信は各 VI ごとの Completion Queue に関連付けられる．

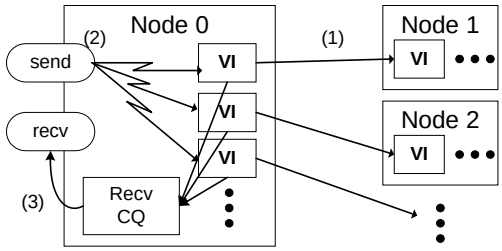


図 3 VIA における通信

データの送信は送り先と接続した VI に対して JVIA.VipPostSend を発行することで行われる (図 3-(2))．送信の完了はその VI の送信用 Completion Queue を監視 (JVIA.VipCQDone) して，完了を確認する．

データの受信は別 thread で行われる．受信スレッドでは，まず各 VI に対して JVIA.VipPostRecv を発行し，受信用 Completion Queue の状態を確認する．Queue が空でない場合，Completion Queue よりデータを取り出し，メッセージキューに追加する．これによって受信したメッセージを VIAComm.recv で取り出すことができる．なお，メッセージキューが空であった場合はブロックする (図 3-(3))．

3.1.3 JPM 及び PMComm

PMComm では PM[11] Version2 を用いる．PM もまた Fast Ethernet，Myrinet，Gigabit Ethernet 上など多くのネットワークデバイス上に実装されている．

PM も Java 言語向けのインターフェースを持たないため，JNI を経由して PM を呼び出す JPM を実装し，この JPM を利用して PMComm を実装する．

JPM では，JNI 呼び出しのオーバーヘッドを削減するため，send/recv という 2 つの native メソッドを用意した．send では pmGetSendBuffer と pmSend を，recv では pmReceive と pmReleaseReceiveBuffer を，それぞれ送受信時に対として呼び出されるメソッドを 1 回の native 呼び出しで行う．

PMComm では，PM はメッセージベースであるため，起動時に接続を確立する必要はないが，最初に各ノードが起動したことをメッセージを送受信することによって確認する．recv では SocketComm と同様に読み出し Thread と PMComm の間に FIFO キューを設け，recv はこのキューからメッセージを取り出す．

3.2 DSM ランタイム

今回の実装では，Write Invalidate プロトコルを用いた Release Consistency を用いて分散共有メモリを実現する．基本的なアルゴリズムは C 言語などでの実現

☆ Berkeley-VIA[9] を用いることも検討したが，実装されていない API があるため今回は利用しない．

```
public abstract class SharedObject{
  Object obj; // 共有されるオブジェクト
  int key; // キー(登録番号)
  int myNode; // ノード番号
  private SharedObjectStatus status;
              // 共有オブジェクトの状態
  void lock(); // 共有オブジェクトのロック
  void unlock();// 共有オブジェクトのアンロック
 ...
}
```

図 4 SharedObject 抽象クラス

と同様である．

各共有オブジェクトは抽象クラス SharedObject(**図 4**) で wrap されて管理される．通常のオブジェクトは抽象クラス SharedObject を継承した SharedObjectOne，配列オブジェクトは抽象クラス SharedObject を継承した SharedObjectForBlock で管理される．以下では，抽象クラス SharedObjectPool に基づいて実装した WriteInvalidate クラスについて，実装したメソッドおよび，それに伴う処理などについて述べる．

initialize メソッド 共有オブジェクトの管理テーブルの初期化，通信インターフェース Comm より実行環境情報 (ノード番号など) の取得，他のノードからのリクエストを処理するためのスレッドを生成などを行う．

register メソッド 分散共有するオブジェクトを引数として，そのオブジェクトを管理テーブルに登録する．管理テーブルに登録するときは，そのオブジェクトを管理用クラスである SharedObject 抽象クラスで wrap して登録する．なお，登録されたオブジェクトの初期の所有者はノード番号 0 番である．

update メソッド 更新すべきオブジェクトを引数として呼び出され，オブジェクトを最新の状態へ更新する．

acquire メソッド ロックしたいオブジェクトを引数として呼び出され，オブジェクトの排他的なアクセス権を得て，そのオブジェクトのローカルコピーを持つノードへ invalidation を発行する．

release メソッド 排他的なアクセスが行われているオブジェクトを引数として呼び出され，排他的なアクセスを解放する．

updateAsync メソッド 非同期の update を提供する．update メソッドでは，要求した共有オブジェクトが得られるのに対して，updateAcync メソッドでは Future オブジェクトが得られる．実際にそのオブジェクトを利用するときに Future オブジェクトに対して touch メソッドを呼び出すことで，目的の共有オブジェクトが得られる．単純な update に比べ，コンシステンシ処理のレイテンシを他の計算で隠蔽することが出来るため，性能向上が期待できる． touch メソッドを呼び出したときにオブジェクトが利用可能な状態になっていない場合はブロックする．

```
public class test extends SPMD{
  Object obj1;
  Object[] obj2;
  Object[] obj3;
  public void initialize(){
    obj1 = new Object();
    obj2 = ... ;
    obj3 = ....;
    pool.register(obj1);    |
    pool.register(obj2);    |  (1)
    pool.register(obj3);    |
  }
  public void start(){
    ...
    pool.update(obj1);        | (2)
    x = obj1;                 |
    Future f;                      |(4)
    f = pool.updateAsync(obj3, 5); |
    pool.acquire(obj1, 10);   |
    obj1[10] = obj1;          | (3)
    pool.release(obj1, 10);   |
    f.touch();                     |(4)
    y = obj3[5];                   |
    ...
  }
  public void finalize(){ ... }
}
```

図 5 サンプルプログラム

acquireAsync メソッド 非同期の acquire を提供する以外， updateAsync と同様である．

finalize メソッド 分散共有されたオブジェクトや管理テーブルなどを破棄する．

他のノードからのリクエスト 他のノードからの以下のリクエストは，別スレッドで処理される．

- オブジェクトコピー要求
- オブジェクトコピー受信
- 所有権要求
- 所有権譲渡
- 所有権移動通知
- Invalidation
- オブジェクトのロック獲得
- オブジェクトのロック解除

3.3 JDSM ユーザプログラム

JDSM システムのユーザープログラムの記述方法をサンプル (**図 5**) を元に説明する．

(1) initialize メソッド内で register メソッドに

よって共有するオブジェクト obj1 obj2 obj3 を登録する．

(2) update メソッドによってアクセスする共有オブジェクト obj1 を更新する．

(3) 共有オブジェクト obj2 への write アクセスは acquire，release メソッドによってロックされる．共有オブジェクト obj2 は配列オブジェクトであるため，ロックする添え字を指定する．

(4) 共有オブジェクト obj3 への updateAsync メソッドでは，まず Future オブジェクトを得る．利用する前に touch メソッドを発行し，更新された共有オブジェクト obj3 を得る．このとき更新処理が終了していない場合はブロックする．

3.3.1 実行の流れ

プログラムは次のような流れで実行される．

(1) 各ノードで JDSM ランタイムが起動される

(2) 各ノード間での接続を確立する

(3) 各ノードで SPMD.initialize を実行

(4) 各ノードで指定されたスレッド数にあわせて SPMD.start を実行

(5) 各ノードで SPMD.finalize を実行

(6) 各ノードで JDSM ランタイムの終了処理を行う

JDSM ランタイムの起動では，Configuration ファイルとノード数，ノード番号及び実行するプログラムとそのオプションを引数として与える．Configuration ファイルには以下のような情報が記述されていてる．

```
# 利用する通信インターフェース
CommClass jdsm.comm.VIAComm
# 通信インターフェースで利用するデバイス名
CommDeice /dev/via_eth1
# ノード情報
CommMapFile node.map
# DSM のコンシステンシモデルの指定
Consistency jdsm.dsm.WriteInvalidate
```

Configuration ファイル jdsm.conf，ノード数 4，ノード番号 0 でプログラム LU を引数 - n1024 で実行する場合，次のように記述する．

```
java jdsm.dsm.SPMDMain -Cjdsm.conf \
  -p4 -m0 LU -n1024
```

4. 性能評価

前章で実装したシステムの有効性を確認するため，Presto PC クラスタ上で異なる Java の実行環境を用いて性能評価実験を行った．実験では，今回実装した各通信インターフェースの基本性能，JDSM ランタイム・バリア同期のオーバーヘッド，アプリケーションとして SPLASH2[12] から LU Kernel を Java へ移植したものを用いたベンチマークを行った．

なお，通信インターフェースとして VIAComm を用いた場合，M-VIA が Reliable な通信を提供できないためアプリケーションにおいて正常な計測が行えなかった．このため，VIA を用いた性能評価は JVIA の通信性能のみである．

4.1 評価環境

評価環境として，松岡研究室の 64 台構成の Presto クラスタ (PentiumII 350MHz，メモリ 256MB，Linux 2.2.14) を用いた．各ノードは 2 系統の 100Mbps の FastEthernet (Intel eepro(Socket で利用)，DEC tulip(VIA で利用)) 及び Myrinet (160MB/s) で接続されている．VIA の実装として M-VIA[8] 1.0 を，PM は RWCP SCore3.0 の PM Version2 を用いた．Java の実行環境として，IBM JDK1.3(IBM1.3)，Sun JDK1.3.0rc1(Sun1.3)，Sun JDK1.2.2 + OpenJIT-1.1.12(Sun1.2.2) を用い，実行時オプションは -Xmx256M である．なお，計測結果は各測定を 5 回行った平均である．これは処理系によってスレッドの切り替えのポリシーが異なるため，特にバリア同期においてスレッド切り替えのコストが大きく，データの分散が大きいためである．

4.2 通信基本性能

4.2.1 Socket

native の Socket と，Java の Socket の 1-way レイテンシとバンド幅を示す (**表 3**)．Java Socket は native の socket に比べ，レイテンシで 1.3 倍の 109.7μsec，バンド幅はほぼ同等の 9.3MB/s であることが分かる．

4.2.2 JVIA

native の VIA(M-VIA) と，それを Java から JNI 経由で呼び出す JVIA の 1-way レイテンシとバンド幅を示す (表 4)．JVIA は VIA のレイテンシの約 1.6 倍の 52μsec，バンド幅は約 9 割強の 10.2MB/s と，大きなオーバーヘッドにはなっていないことが分かる．この差は JVIA による JNI 呼び出し，及びデータのマーシャリングのオーバヘッドと考えられる．

4.2.3 JPM

native の PM と，それを Java から JNI 経由で呼び出す JPM の 1-way レイテンシとバンド幅を示す (**表 5**)．JPM は PM のレイテンシの 1.5 倍の 10.4μsec，バンド幅は 9 割強の 31MB/s と良好な結果が得られている．主なオーバーヘッドは JNI 呼び出し，及び呼び出し時のデータのマーシャリングである．JVIA に比べこのオーバーヘッドが小さいのは，JPM の場合，マーシャリングされるデータが少ないためである．

	Socket		Java Socket	
サイズ (byte)	レイテンシ (usec)	バンド幅 (MB/s)	レイテンシ (usec)	バンド幅 (MB/s)
1	86.68	0.011	109.7	0.009
4	86.81	0.046	109.0	0.036
16	89.00	0.179	111.3	0.143
64	103.7	0.617	125.1	0.511
256	155.4	1.646	177.4	1.442
1024	350.4	2.921	372.9	2.745
4096	671.6	6.098	699.0	5.859
16384	1910.2	8.577	1739.0	9.421
32768	3109.7	10.53	3496.5	9.372

表 3 Socket, Java Socket のレイテンシとバンド幅 (FastEthernet)

	M-VIA		JVIA	
サイズ (byte)	レイテンシ (usec)	バンド幅 (MB/s)	レイテンシ (usec)	バンド幅 (MB/s)
1	31.143	0.032	52.0	0.0192
4	31.124	0.128	52.1	0.0769
16	31.252	0.511	52.5	0.3047
64	38.549	1.660	60.5	1.057
256	72.388	3.536	95.5	2.680
1024	198.91	5.147	226.8	4.520
4096	490.43	8.351	537.5	7.620
16384	1511.1	10.84	1690.1	9.700
32000	2817.6	11.35	3145.5	10.17

表 4 M-VIA, JVIA のレイテンシとバンド幅 (FastEthernet)

現在，PM の性能が 34MB/s 程度で頭打ちになっている．これはノードの PCI の制限により，PCI の DMA-write の性能が 34MB/s 程度であるためである．

4.3 JDSM 基本性能

基本性能として，JDSM ランタイムのオーバーヘッド，バリア同期のオーバヘッドの結果を示す．これらの実験では処理系として IBM JDK1.3 を，通信インターフェースには `SocketComm`，`PMComm` を用いた．

JDSM ランタイムのオーバーヘッドでは，JDSM ランタイムの機構を利用した 2 ノード間の PingPong により，レイテンシとスループットを計測した．比較として，直接 `Comm` インターフェースを利用した結果を示す (**図 6**，**図 7**)．この PingPong では配列オブジェクトを共有し，各ノードで write アクセスを交互に行っている．この処理では `acquire`，`release` 及び invalidation の発行などが行われている．

`Comm` インターフェースを直接利用した場合，`SocketComm` ではサイズ 64KB のとき 11MB/s，`PMComm` では 32MB/s と，十分なスループットを達成できていることが分かる．これに対して JDSM ランタイムを利用した場合，`SocketComm` ではサイズ 128KB のとき 5.4MB/s，`PMComm` では 7.5MB/s と，`Comm` インターフェースを直接利用した時に比べ，4 分の 1 から半分の程度の性能しか出ていないことが分かる．これは `acquire` に伴うロックの獲得やオブジェクト転送に伴うシリアライズのオーバーヘッドが原因であると考えられる．

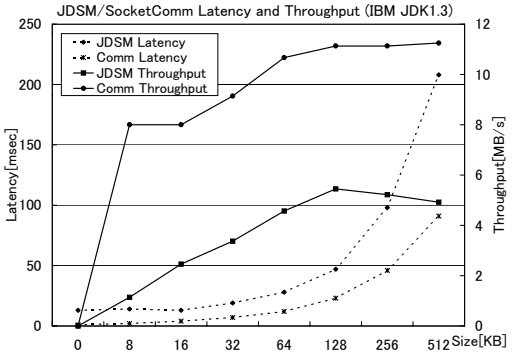


図 6 JDSM ランタイム基本通信性能 (SocketComm)

次に，バリア同期のコストを示す (**図 8**)．`SocketComm` より `PMComm` を利用した方が若干性能がよい事が分かる．しかし，バリア同期では JDSM ランタイムを利用しているため，32 ノードにおいて `SocketComm` で 43msec，`PMComm` で 29msec と，JDSM ランタイムのオーバーヘッドが大きく影響している．このため，`Comm` インターフェースを直接利用したバリア同期を実装中である．

4.4 SPLASH2 LU Kernel

次に Java へ移植した SPLASH2 LU Kernel を用いた性能評価を示す．行列サイズは 1024 とした．

表 6 は JDSM ランタイムを利用しない場合の各処理系での単一プロセッサ上の性能と C 言語での単一プロ

	PM		JPM	
サイズ (byte)	レイテンシ (usec)	バンド幅 (MB/s)	レイテンシ (usec)	バンド幅 (MB/s)
1	6.82	0.147	10.4	0.096
4	6.86	0.586	10.1	0.396
16	7.02	2.266	10.5	1.523
64	9.86	6.478	14.2	4.507
256	18.00	14.25	24.0	10.66
1024	50.30	20.36	63.7	16.07
4096	167.6	24.44	211.0	19.41
16384	481.0	34.06	652.1	25.12
65536	1923.8	34.07	2087.3	31.39

表 5 PM, JPM のレイテンシとバンド幅 (Myrinet)

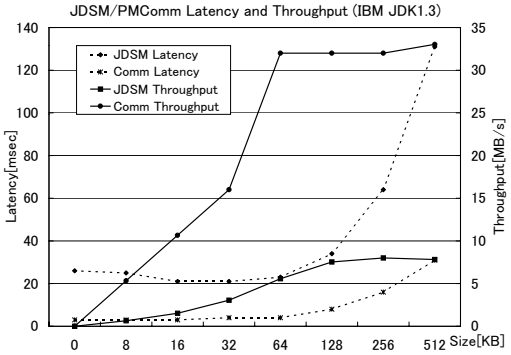


図 7 JDSM ランタイム基本通信性能 (PMComm)

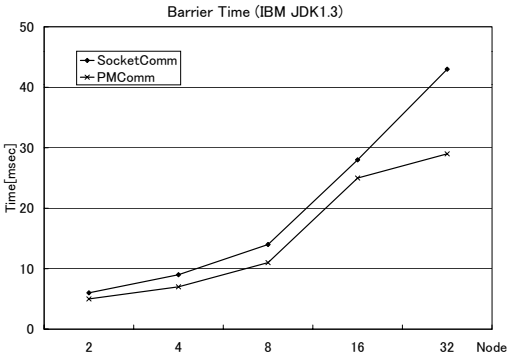


図 8 バリア同期のコスト

	Sun1.3	IBM1.3	Sun1.2.2	C
Time [sec]	16.8	13.4	44.3	7.64

表 6 JDSM ランタイムを利用しない SPLASH2 LU

セッサ上の性能である．**図 9**，**図 10**，**図 11** は通信インターフェースとして `SocketComm` を用いた場合の結果である．

`SocketComm` の結果を見ると，各処理系でばらつきがあるのが分かる．IBM1.3 ではノード数が増えると，実行時間は減少するものの，速度向上は低い．また，Sun1.3 では 8 ノードで 3 倍程度の速度向上を達成している．Sun1.2.2 では 32 ノードでバリア同期の時間が増大し，性能を大きく下げている．しかし，いずれの処理系でも LU Kernel のメインの計算部分である Interior において，8 ノード程度までは比較的速度向上が得られている．16 〜 32 ノードでこの Interior の台数効果が落ちているのは，1 ノードあたりの計算時間が小さくなったため，メモリのコンシステンシ処理のオーバーヘッドが顕著化してきたものと考えられる．

各処理系，特に Sun1.2.2 で 2 ノードのときのバリア同期のコストが大きいことが分かる．これは計算中にスレッドが切り替わらず，acquire や update などに伴う他のノードからの要求処理が後回しにされているためである．このメモリコンシステンシ処理のオーバヘッドにより，負荷の不均衡が生じてしまい，その結果，全体のバリア同期の時間を引き上げているものと考えられる．Sun1.2 で 32 ノードの時のバリア時間の増大の原因は調査中である．

また，C 言語での Software DSM の実装である Shasta[13] では，8 ノードで約 3 倍，16 ノードで約 5 倍の速度向上であることを考えると，改善の余地はあるものの，比較的良好な結果と言える．

最後に，まだ十分チューニングされていないが，通信インターフェースとして `PMComm` を用いた場合の結果を**図 12** に示すが，性能が非常に不安定であるのが分かる．これはメッセージ受信スレッドにおける busy wait が実行時間の半分近くを占めてしまっているためであり，現在改良中である．

5. 関連研究

我々は本研究の先行研究として，C++ 言語を並列分散拡張した言語 MPC++ 上にソフトウェア DSM を実現するとともに，SPMD スタイルでかかれたマルチスレッドプログラムをこの DSM 上で動作するように OpenC++2.5[14] のリフレクション機能を用いてプログラム変換するシステム OMPC++[15] を実現した．OMPC++ システムは JDSM システムと同様に，複数の並列実行環境への適応 (プラットフォームポータビリティ及びパフォーマンスポータビリティ) を考慮して設

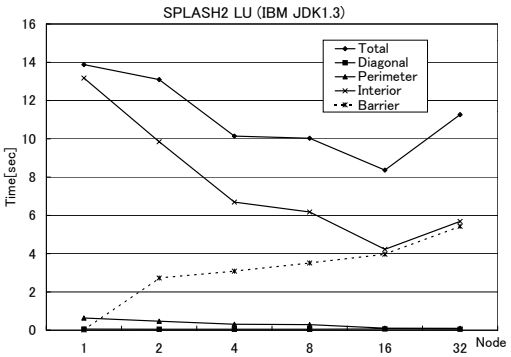


図 9 SPLASH2 LU(IBM1.3) SocketComm

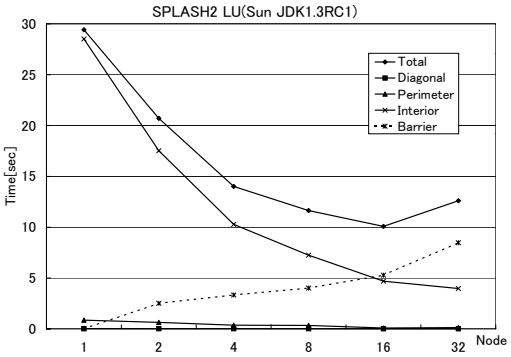


図 10 SPLASH2 LU(Sun1.3) SocketComm

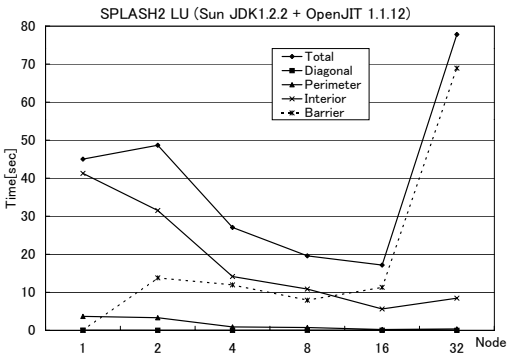


図 11 SPLASH2 LU(Sun1.2.2) SocketComm

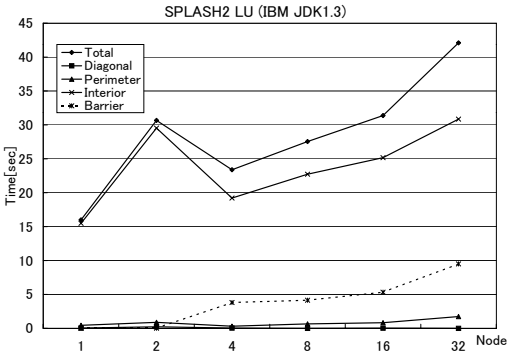


図 12 SPLASH2 LU(IBM1.3) PMComm

計されている．

また，C 言語などを対象としたソフトウェア分散共有メモリシステムは Kai Li[16] 以来，多く研究されてきたが，Java 言語を対象としたシステムは比較的少ない．

Java/DSM[1] Java/DSM は，JVM を改変することによって実現された DSM システムである．Java/DSM の JVM は，C レベルの DSM システムである TreadMarks[17] を内部で利用するように，メモリ管理や GC などが変更されている．DSM の処理は全て TreadMarks で行われるため，Page-Base のシステムである．

cJVM[2),18)] cJVM もまた，JVM を改変することによってクラスタ上に Single System Image を実現する DSM システムである．cJVM での主な JVM の変更は分散ヒープモデル，透過的なメソッドシッピングをサポートしたスレッドモデル，分散クラスローディングなどである．また，cJVM は Master-Proxy モデルを採用した Object-Base の DSM システムである．

JESSICA[19] JESSICA もまた cJVM と同様に JVM を変更することによって Single System Image を提供する．cJVM と異なるのは，Thread の移送やロードバランシングなどを実現していること，また JVM が既存の DSM システム上に実現されていることである．

Hyperion[20] Hyperion は，Java プログラムを C プログラムへ変換して既存の DSM システム・スレッドライブラリを利用する．C プログラムへの変換時に DSM 関連の命令が挿入される．変換された C プログラムで利用される DSM システムは複数のコンシステンシプロトコルを提供しており，また多くの通信インターフェース上で利用可能である．

いずれも JVM への変更が必要であるなど，ポータビリティが大きく損なわれる上，JIT コンパイラなど既存の資源も利用できない．特にポータブルな高性能計算のためには JIT コンパイラの改変を要する JVM の改変は現実的ではない．さらに，並列科学技術計算での性能評価などが行われていないため，実際の性能がどの程度であるかなどが不明である．

6. まとめと今後の課題

本研究では，Grid Computing に向けたプラットフォームポータビリティ及びパフォーマンスポータビリティを実現するため Java 言語を利用することを提案した．その第一段階として，ポータブルかつ PureJava であるソフトウェア分散共有メモリシステムを Java の言

語特性に合わせた設計を行った上で実現した．さらに，このシステムを32台のPCクラスタ上で評価を行い，その有効性を確認した．その結果，Software DSMのランタイムオーバーヘッドが大きいこと，また`Comm`インターフェースを用いた通信は十分な性能を達成できることが分かった．

今後の課題としては，まずはシステムの完成度を上げることである．特にVIAを利用した場合，M-VIAの不安定さが原因で十分な評価が行えなかったため，M-VIAに対して適切な対処を行う必要がある．また，PM版における性能の不安定さを解消し，各処理系での評価を行っていく．

今回，メモリコンシステンシ処理のオーバーヘッドによって，負荷の不均衡が生じてさらなるオーバーヘッドとなるような悪循環が生じた．このため，DSMランタイムにおけるメモリコンシステンシ処理のオーバーヘッドの低減が必要である．また，異なるメモリコンシステンシモデルを採用し，`SharedObjectPool`クラスのサブクラス化で実現できることや性能に与える影響の違いを検証する．また，LU Kernel以外のアプリケーションでも性能評価を行う必要がある．

現在，アプリケーションへの変更は手動で行う必要がある☆．そこでOMPC++で有効であるとされたリフレクションを用いた自動変換を，OpenJIT[22),23)]を利用したJITコンパイル時の変換で実現する予定である．

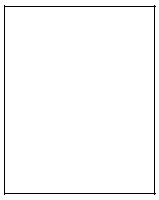
参 考 文 献

1) Yu, W. and Cox, A.: Java/DSM: a Platform for Heterogeneous Computing, *ACM 1997 Workshop on Java for Science and Engineering Computation*, Vol. 43.2, pp. 65–78 (1997).
2) Aridor, Y., Factor, M. and Teperman, A.: cJVM: a Cluster Aware JVM, *Proceedings of International Conference on Parallel Processing '99*, pp. 31–39 (1999).
3) Gosling, J., Joy, B. and Steel, G.: *The Java Language Specification*, The Java Series, Addison-Wesley (1996).
4) Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification*, The Java Series, Addison-Wesley (1996).
5) Kambites, M. and Bull, J.: JOMP – An OpemMP-like Interface for Java, *ACM 2000 Java Grande Conference*, pp. 44–53 (2000).
6) Pugh, B.: Java Memory Model, `http://www.cs.umd.edu/ pugh/java/memoryModel/`.
7) http://www.viarch.org/: Virtual Interface Architecture Specification (1997).
8) National Energy Research Scientific Computing Center: M-VIA, http://www.nersc.gov/ research/FTG/via/ (1999).
9) Buonadonna, P., Geweke, A. and Culler, D.: An Implementation and Analysis of the Virtual Interface Architecture, *Proceedings of SC'98* (1998).
10) Welsh, M. and Culler, D.: Jaguar: Enabling Efficient Communication and I/O from Java, *Concurrency: Practice and Experience, Special Issue on Java for High-Performance Applications* (1999).
11) Tezuka, H., Hori, A., Ishikawa, Y. and Sato, M.: PM: An Operating System Coordinated High Performance Communication Library, *High-Performance Computing and Networking '97* (Sloot, P. and Hertzberger, B.(eds.)), Vol. 1225, Lecture Notes in Computer Science, pp. 708–717 (1997).
12) Woo, S. C., Ohara, M., Torrie, E., Singh, J. P. and Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 24–36 (1995).
13) Scales, D. J. and Gharachorloo, K.: Towards Transparent and Efficient Software Distributed Shared Memory, *Proceedings of the 16th ACM Symposium on Operating System Principles*, Western Research Lab., Digital Equipment Corporation (1997).
14) Chiba, S.: A Metaobject Protocol for C++, *Proceedings of OOPSLA'95*, pp. 285–299 (1995).
15) Sohda, Y., Ogawa, H. and Matsuoka, S.: OMPC++ — A Portable High-Performance Implementation of DSM using OpenC++ Reflection, *Proceedings of Reflection '99, LNCS 1616 Meta-Level Architecture and Reflection*, pp. 215–234 (1999).
16) Li, K. and Hudak, P.: Memory Coherence in Shared Virtual Memory Systems, *ACM Transactions on Computer Systems*, Vol. 7, No. 4, pp. 321–359 (1989).
17) Amza, C., Cox, A., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W. and Zwaenepoel, W.: TreadMarks: Shared Memory Computing on Networks of Warkstations, *IEEE Computer*, Vol. 29, No. 2, pp. 18–28 (1996).
18) Aridor, Y., Factor, M., Teperman, A., Eliam, T. and Schuster, A.: A High Performance Clus-

☆ EPP[21)]を利用した自動変換も実現したが，効率のよい変換結果が得られていない

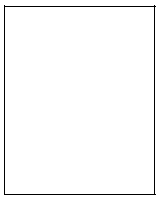
ter JVM Presenting a Pure Single System Image, *Proceedings of ACM 2000 Java Grande Conference*, pp. 168–176 (2000).
19) Matchy, J. M.M., Wang, C., Lau, F. C.M. and Zhiwei, X.: JESSICA: Java-Enabled Single-System-Image Commputing Architecture, *1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '99)* (1999).
20) Antoniu, G., Bouge, L., Hatcher, P., MacBeth, M., McGuigan, K. and Namyst, R.: Implementing Java consistency using a generic, multithreaded DSM runtime system, *Parallel and Distributed Processing. Proc. Intl Workshop on Java for Parallel and Distributed Computing*, LNCS, Vol. 1800 (2000).
21) Ichisugi, Y. and Roudier, Y.: Extensible Java Preprocessor Kit and Tiny Data-Parallel Java, *Proceedings of ISCOPE '97* (1997).
22) Matsuoka, S., Ogawa, H., Shimura, K., Kimura, Y., Hotta, K. and Takagi, H.: OpenJIT —A Reflective Java JIT Compiler, *Proceedings of OOPSLA '98 Workshop on Reflective Programming in C++ and Java*, pp. 16–20 (1998).
23) Ogawa, H., Shimura, K., Matsuoka, S., Maruyama, F., Sohda, Y. and Kimura, Y.: OpenJIT: An Open-Ended, Reflective JIT Compile Framework for Java, *Proceedings of ECOOP '2000 - Object-Oriented Programming*, LNCS, No. 1850, pp. 362–387 (2000).

(平成10年7月14日受付)
(平成10年10月10日採録)



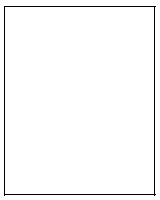
早田　恭彦（学生会員）

昭和49年生．平成10年東京工業大学理学部情報科学科卒業．平成12年同大学大学院情報理工学研究科数理・計算科学専攻修士課程修了．現在，同大学大学院博士課程在学中．日本学術振興会特別研究員．並列・分散システム，オブジェクト指向言語，広域分散システムなどに興味を持つ．ACM会員．



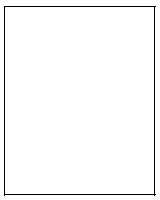
中田　秀基（正会員）

昭和42年生．平成2年東京大学工学部精密機械工学科卒業．平成7年同大学大学院光学系研究科情報工学専攻博士課程修了．博士(工学)．同年電子技術総合研究所研究官．現在同所主任研究官．並列プログラミング言語，オブジェクト指向言語，分散計算システムに関する研究に従事．



小川　宏高（正会員）

昭和46年生．平成6年東京大学工学部計数工学科卒業．平成8年同大学大学院工学系研究科情報工学専攻修了．平成10年同博士課程中退．現在，東京工業大学情報理工学研究科助手．プログラミング言語，言語処理系，オブジェクト指向技術，並列計算機アーキテクチャ，広域分散システムに興味を持つ．ACM会員．



松岡　　聡（正会員）

1963年生．1986年東京大学理学部情報科学科卒，1989年同大学大学院博士課程中退．同大学情報科学科助手，情報工学専攻講師を経て，1996年より東京工業大学情報理工学研究科数理・計算科学専攻助教授．1998年より科学技術振興財団のさきがけ研究員を併任．理学博士．オブジェクト指向言語，並列システム，リフレクティブ言語，制約言語，ユーザ・インターフェースソフトウェアなどの研究に従事．現在進行中のプロジェクトは，世界規模の高性能計算環境を構築するNinfプロジェクト，オブジェクト指向言語の大規模並列クラスタ計算機上の実装，計算環境に適合・最適化を目指すJava言語の開放型Just-In-Timeコンパイラ OpenJIT，各種ユーザインターフェースなど．1996年度情報処理学会論文賞，1999年情報処理学会坂井記念賞受賞．オブジェクト指向の国際学会ISOTAS'96，ECOOP'97，ISCOPE'99のプログラム委員長やOOPSLAなどの数々のプログラム委員・実行委員などを勤める．情報処理学会，ソフトウェア科学会，ACM,IEEE-CS各会員，IEEE Concurrency Magazineの Editor.