高度情報化支援ソフトウェアシーズ育成事業

Java による高可搬かつ高性能な分散共有メモリ型並列プログラミング環境 JDSM の開発

論文

株式会社ベストシステムズ

平成 12 年 1 月

目次

表一覧

図一覧

序論 1 ≡

1.1 背景

近年の計算機構成技術の発展によって、ソフトウェアの実行環境はますます多様化している。この計算機構成技術の発展はハードウェアの急速なコモディティ化と対応するものである。特に、並列計算機環境においては、このハードウェアのコモディティ化によって PC クラスタ・ WS クラスタという形態の分散メモリ型並列計算機が多数開発されている。

しかし、このような計算機環境の多様化に肝心のソフトウェアが追従できているとは言いがたい。現在、多様な計算機環境に適合しうる、保守性や可搬性を重視した汎用的なソフトウェアや、さらに、それを実現するためのフレームワークすら十分でないのが現状である。例えばオブジェクト指向技術は確かにソフトウェアの効率の良い構築フレームワークを与えるが、実行環境の変化に適合した最適化を行うには十分ではない。

特に並列計算機環境においては、メモリモデルやプロセッサ構成・ネットワークの構成や形態、さらにプログラミングモデルの違いなど、その構成の多様性から従来の手法での解決が難しい。従来の手法とは、

- HPF に代表される並列プログラミング言語
- SUIF などの自動並列化コンパイラ
- PVM や MPI などのメッセージ通信ライブラリ

のいずれかを用いることで実現されていた。しかし、並列プログラミング言語や自動並列化コンパイラは、言語拡張が必要であったり十分な性能を達成できないなどの問題がある。メッセージ通信ライブラリではプログラムの記述性や保守性が損なわれ、実行環境を強く意識したプログラミングを行わなければ、高い性能を発揮することが難しいなどの問題がある。さらに、処理系自体の可搬で高効率な実装や柔軟な言語拡張が難しいという問題もある。

以上のような観点から、計算機環境、特に並列計算機環境で利用されるソフトウェアにはプラットフォームポータビリティ が強く求められる。具体的には、

- ソフトウェア (システム・処理系) の可搬性
- 実行環境への最適化
- 性能の可搬性

が求められる。

1.2 本研究の目的と提案

前節で述べた並列計算機環境におけるプラットフォームポータビリティを実現する手法を提案し、プラットフォームポータビリティを確保できることを示す。

本研究で提案する手法は、自己反映計算 (リフレクション) を用いてソフトウェアのプラットフォームポータビリティを実現する方法である。自己反映計算は前節で述べたような問題を整合的かつ柔軟に解決する方法であると考える。自己反映計算では通常の計算を表す従来からのベースレベルのコードに加え、自己の計算系を表すメタレベルのコードを記述することができるため、従来例外的かつ固定的に扱わざるを得なかった例外処理や資源管理を整合的に処理できる。特に、Open Compiler と呼ばれる、コンパイラのコンパイル時の挙動を自己反映的に変更する機能を持つシステムでは、これらの機能を容易に実現できることが知られている。Open Compiler ではコンパイラをオブジェクト指向設計に基づきモジュール化してユーザから変更可能にし、更にコンパイル時に静的にメタ計算を行うこと (Compile-time MOP) によって言語の拡張やアプリケーション固有の最適化などを行うことができる。

本研究では、クラスタ型並列計算機上にマルチスレッドプログラムが動作可能な環境を提供する。この理由には、

- ハードウェアのコモディティ化によってコストパフォーマンスの高いクラスタ型並列計算機が急速に普及しており、性能やシステムの拡張性に優れていること
- マルチスレッドプログラミングが一般に並列プログラミングにおいて容易なモデルとされていること

が挙げられる。つまり、クラスタ型並列計算機上にマルチスレッドプログラムが動作可能な環境は並列計算において理想的な環境であると言える。

しかし、クラスタ型並列計算機では、 MPI などのメッセージパッシングライブラリなどを用いた、分散メモリモデルのプログラミングであり、マルチスレッドプログラムは基本的に共有メモリマシンでしか動作しないため、対象となる計算機は共有メモリ計算機である必要がある。

そこで、マルチスレッドプログラムを分散メモリ環境で実現するには、各ノードのメモリを透過的に扱える必要がある。そこで、ポータブルなソフトウェア分散共有メモリを実現し、その上でマルチスレッドプログラムを動作させることが考えられる。しかし、ここでもポータビリティ上の問題が発生する。マルチスレッドプログラムを、ソフトウェア分散共有メモリシステム上で動作させるためには、少なからずもとのマルチスレッドプログラムを修正する必要が出てくるためである。

そこで、本研究では、この直交する 2 つの要素である、クラスタ型並列計算機とマルチスレッドプログラミングを、自己反映計算とソフトウェア分散共有メモリを用いて実現する。つまり、自己反映計算とソフトウェア分散共有メモリを用いて、並列実行環境上でプラットフォームポータビリティを実現する並列プログラミング環境の実現する。

本研究では、 Java 言語に対して実現することとした。 Java 言語を利用する利点としては、

1. 言語仕様において、ソースコードレベルだけでなく、コンパイルされたバイトコードレベルにおいてもポータビリティを提供している
2. C++ 言語と同様にオブジェクト指向言語であるため、プログラムの保守などが容易である
3. 言語仕様においてスレッド・ネットワークのサポートがあるため、ネットワーク環境や並列プログラミングへの親和性が高い
4. ネットワーク経由でプログラムをダウンロードすることが可能
5. C++ 言語に比べ、自己反映計算を適用可能なレベルが多い

が挙げられる。 1 についてみると、自己反映計算によるプログラムの変換は純粋に機能拡張や環境特化にしぼられるため、不要な変換 (スレッドモデルの違いなど) を考慮する必要がない。 4 では、必要に応じてプログラムをダウンロードできるため、環境に特化したプログラムをネットワークからダウンロードしてくる、環境に特化したメタクラスをダウンロードしてくると言ったことが可能となる。さらに、自己反映計算を適用可能なレベルが、ソースコードレベル、バイトコードレベル、プログラムロード時、 JIT コンパイル時など多くの場合が考えられる (表 1.1)。それぞれのレベルにおいて様々な機能拡張や最適化を行うことができる。しかし、既存の環境を変更するような手法、特にランタイム (JVM) の変更を行うような手法は、ポータビリティの点から利用すべきではない。例えば、表 1.1 の実行時に自己反映計算を行うシステムである MetaXa などは JVM への変更が必要となる。 JIT コンパイル時に自己反映計算を行う OpenJIT は、アーキテクチャ依存部分 (マシンコード生成部) が独立しているため、カスタム JIT コンパイラではあるが、可搬性は高く、幅広い処理が可能である。

しかし、 Java 言語を利用した場合、考慮すべき点がある。

1. 実行時性能
2. JVM(Java Virtual Machine) への依存

タイミング	処理系	処理内容	長所	制限
ソースコード	EPP	プリプロセッサ	言語仕様の拡張が用意	ソースコードが必要
コンパイル時	OpenJava	Compile-Time MOP	言語仕様の拡張が用意	ソースコードが必要
バイトコード	BCA	Translater	ソースコードが不要	JVM のクラスファイルロード機構に変更が必要
実行時	Kava, MetaXa	Reflective JVM	動的な拡張が可能	カスタム JVM が必要
JIT コンパイル時	OpenJIT	Compile-Time MOP	ソースコードが不要、ソースレベルからバイナリレベルまでの最適化や機能拡張が動的に可能	カスタム JIT コンパイラが必要

表 1.1: Java における自己反映計算の実現可能なレベル

1 に関しては、必ずしも大きな問題ではない。なぜなら、 JIT コンパイラや GC(Garbage Collection)、 Java 言語向けの最適化手法などが数多く研究され、 C 言語並の性能を達成することが可能である。 2 は、本研究の手法を用いることで解決される。 Java 言語のポータビリティは JVM のポータビリティに依存しており、 JVM の提供する機能に制限されていた。しかし、本研究の提案する手法 — 自己反映計算の適用 — によって JVM に縛られない機能拡張や実行環境特化が行える。

関連研究 2 ≡

本章では、本研究で構築するシステムを構成する要素技術とそれらに対して行われている研究について述べる。

はじめに分散共有メモリ、特にソフトウェアでの実現に重点をおいて説明する。関連研究においては、ソフトウェア分散共有メモリの研究のうち、 Shasta、 TreadMarks、 Java 言語上に実現されたソフトウェア分散共有メモリシステムである cJVM、 Java/DSM について述べる。また、本研究で対象としているクラスタ型並列計算機向けシステムやそれらで利用されている高速な通信ライブラリなどについても述べる。

自己反映計算においては、自己反映計算の理論について述べるとともに、関連研究として C++ 言語、 Java 言語向けに実現しているシステム (OpenC++、 OpenJIT、 EPP など) について述べる。

2.1 分散共有メモリ

分散共有メモリとは、異なるノード上に存在するメモリを透過的に扱う手法である。

他のノードが持つメモリへのアクセスは最終的には I/O などの手続きで解決されるが、その手続きの指定は明示的ではない。つまり、

- プログラム作成時に、プログラム作成者に、注釈 (annotation) を付けさせる。
- コンパイル時に、コンパイラに、静的に検出させる。
- 実行時に、 OS ないしランタイムライブラリに、動的に検出、処理させる。

といった方法でプログラム作成者から見た手続きの指定を隠蔽して、メモリアクセスに類似したインタフェースだけを提供するものである。

2.1.1 アルゴリズム

分散共有メモリの実現で最も重要なのは、分散したメモリの一貫性を保つことである。このメモリの一貫性処理は分散共有メモリ、特にソフトウェア分散共有メモリにおいて、アプリケーションの性能の大幅な低下を招く危険性が高く、効率の良い実装が求められる。効率の良い実装としてポイントとなる点は

- メモリ一貫性処理のオーバーヘッド
- False Sharing

である。メモリ一貫性処理のオーバーヘッドとしては、メモリアクセス (領域や状態) のチェックやメモリの転送に伴う通信オーバーヘッドなどが挙げられる。これを低減するには高速なネットワークを利用する、メモリ管理の粒度を大きくしてメモリアクセスチェックを減らすなどの方法が挙げられる。 False Sharing では、異なるノードが同じメモリ領域を頻繁にアクセスしあうことによって引き起こされる、メモリ一貫性処理のオーバーヘッドがある。 False Sharing を減らすには、メモリ管理の粒度を小さくする、メモリへの複数ノードからの書き込み許すなどの方法がある。

分散共有メモリで利用されるメモリの一貫性を保つモデルとしては以下のようなものが存在する。

Strict Consistency どのような共有変数に対するロードも、一番最近にストアされた値を読み込んでくるようなモデル。つまり、唯一の絶対時間が存在し、システム中の全てのモジュールはその絶対時間の下で動かなければならない。この実装は非常に困難である。

Sequencial Consistency 共有メモリ型並列計算機で、あるプログラムを実行した時、その実行時間が、そのプログラムを実行する全てのプロセッサの命令列をある規則にしたがって逐次に実行した時の実行した時の実行結果と同じであり、個々のプロセッサ上での命令の順番はプログラムに書かれた順番と一致するモデル。

Weak Consistency 全ての同期命令は他の全てのプロセッサに関するロードおよびストアが完了した後に実行され、全てのロードおよびストアは他の全てのプロセッサに関する同期命令が完了した後に実行されるモデル。

Release Consistency 他の全てのプロセッサに関して通常のロードやストアが実行される前に、それ以前の全ての獲得命令が完了している必要があり、他の全てのプロセッサに関して解放命令が実行される前に、それ以前の全てのロードやストア命令が完了していなければならないモデル。

Lazy Release Consistency Release Consistency を緩めたモデル。ある区間ごとに共有メモリへのアクセス情報を保持し、区間を出るときに全ノードで同期をとり、そのメモリ差分 (diff)

をとってメモリの内容を更新する。つまり、ある区間ではメモリの一貫性は崩れるが、その区間を出るときには一貫性が保たれる。

従来、 Release Consistency モデルもしくはそれを改良したモデルが多く用いられてきたが、最近では、性能から Lazy Release Consistency モデルが利用されることが多い。さらに、メモリ情報の保持の方法などからも分類でき、以下のような手法がある。

migration 共有メモリを各ホストに分散させ、なんらかのメモリアクセスがあった時、その領域がローカルになければその領域を持つホストを見つけ出して、その領域をローカルに移動するアルゴリズムである。メモリへのアクセスはそのホスト上で動いているプロセスしかできないので、 Single reader/Single writer と呼ばれる。

このアルゴリズムの利点はプロセスがローカルにある共有メモリへアクセスする時に通信のコストがかからないことである。もう一つの利点は、もし共有メモリブロックのサイズが仮想メモリのページサイズと同じであれば、ホストのオペレーティングシステムの仮想メモリシステムに統合できることである。これは、もし共有メモリがローカルにある場合、それはプログラムの仮想アドレス空間にマッピングすることができ、メモリアクセスが通常の命令で可能になる。ローカルにないメモリへのアクセスがあった場合、ページフォルトが発生するが、このページフォルトをトラップすることにより、そのメモリ領域をリモートから獲得してローカルにマッピングすることができる。

このアルゴリズムの欠点は共有メモリ領域がホスト間を振り回される恐れが大いにありうることである。これはプログラム作成時に共有メモリのブロックサイズを調節することにより、ある程度制御できる。また、ある共有メモリブロックを持つリモートホストを見つけるために、マルチキャストをする必要がある。これはネットワークの負荷を増大させる。しかし、その共有メモリブロックの所有者を常に把握する管理サーバーを設けることにより、負荷を分散させることができる。

Write Invalidate このアルゴリズムは共有メモリの複数のコピーを許すが、その領域へ書き込みがあった場合、他のマシンが持つコピーを無効化しなければならない Multi reader/Single writer タイプである。

あるマシンでローカルにない共有メモリへの read アクセスがあった場合、そのマシンはその共有メモリの所有者へその共有メモリのコピーを要求する。リクエストを受けたマシンではリクエストしたマシンへコピーを送り、その領域のコピーをもつマシンのリストへリクエストしたマシンのエントリーを加える。

あるマシンでローカルにある共有メモリへの write アクセスがあった (その共有メモリの所有者である) 場合、その共有メモリのコピーをもつマシンへその領域の無効を伝える。無効の通知を受けたマシンではその領域を破棄する。そのマシンは再びその領域へのアクセスがない限

り、コピーを持たない。そして、その領域へ書き込むことができる。また、あるマシンでローカルにない共有メモリへの write アクセスがあった (その共有メモリの所有者でない) 場合、その共有メモリの所有者へその領域のコピーと所有権を要求する。要求を受けたマシンはその領域のコピーを返し、所有権を渡す。所有権を獲得したマシンではその共有メモリの無効をその共有メモリのコピーを持つマシンへ伝える。そして、その領域へ書き込むことができる。

Write Update このアルゴリズムは共有メモリへ書き込むまさにその時もその領域のコピーを持つことを許している Multi reader/Multi writer タイプである。

ある共有メモリへの read アクセスがあった場合、それがローカルになければその領域をもつホストにその領域のコピーを要求する。ある共有メモリへの write アクセスがあった場合、シーケンサに書き込むデータを送信する。シーケンサは書き込みのデータを受信すると、そのデータにシーケンス番号を割り当てて、すべてのホストに変更をマルチキャストする。変更を受け取ったホストでは、シーケンス番号順にメモリの内容を更新する。もしこのシーケンス番号に矛盾が生じた場合はシーケンサに再送信が要求される。このためシーケンサでは最近の書き込み要求の情報を保持しておく必要がある。書き込み要求を出したホストはこのマルチキャストを受けると、書き込み要求が受理されたと見なして、ローカルのメモリを更新する。

このアルゴリズムでは、一回の書き込みごとにホストの数＋２回の送信が発生するため、ネットワークの負荷が大きい。また、一度ある共有メモリのコピーを所有したホストはずっとそのコピーを保持しなくてはならない。

実装の容易さ、性能面などから Write Invalidate が用いられることが多いが、最近では、ネットワークの高速化などにより、 Write Update プロトコルやそれを改良した Automatic Update Protocol などが提案されている。

2.1.2 関連研究

ソフトウェア分散共有メモリ

Shasta

Shasta[SGT96] は分散メモリをもつクラスタ型計算機システム上で共有アドレス空間をサポートするシステムである。実装環境には DEC の Alpha ワークステーションが用いられている。

Shasta は他のソフトウェア分散共有メモリシステムに比べて以下のような特徴を持っている。

- 疎粒度で共有データの一貫性を保つことができる
- 一貫した粒度で単一のアプリケーションにおいて異なる共有データの構造を許す
- 共有メモリ向けプログラムをバイナリレベルでプログラム変換を行い、分散共有メモリ環境上で動作可能としている

プログラム変換ではメモリのロードとストアをフックするようにプログラムを書き換えることによって共有アドレス空間を実装している。各共有メモリへのロードとストアでは、そのデータがローカルであるかどうかのチェックを行うコードを挿入し、必要があれば通信を行う。これらのチェックによる実行時のオーバヘッドを減らすために Shasta では多くの工夫を行っている。

Shasta はソフトウェアで実装されているため、異なるタイプのキャッシュコーヒレンスプロトコルのサポートにおいて多くの柔軟性を提供している。 Shasta では relaxed メモリーモデルや複数の通信粒度のサポートなどを含めた多くの最適化を組み合わせたキャッシュコーヒレンスプロトコルを実装している。

Shasta で効率のよい実装の実現のために行われているテクニックは、共有アドレス空間のレイアウト (メモリの管理テーブルと実際のアドレスとの変換をシフト演算のみで行えるようにアドレスの割当を変更)、効果的な実行を行うためのチェックコードのスケジューリング (チェックのバッチ処理化など)、チェックのために値のロードしか行わないシンプルな命令の使用、チェックコードによって起こるキャッシュミスの削減、複数のロードおよびストアのためのチェックの組み合わせなどがある。

しかし、プログラム変換がバイナリレベルであるため、オーバーヘッドの少ない変換を施せる反面、ポータビリティが大幅に損なわれている。特に Shasta では Alpha のメモリチャネルに特化しているため、異なるシステムへの移植は困難である。

Java/DSM

Java/DSM[YC97] は TreadMark[ACD+96] チームによる、異機種分散環境のサポートを目指し、下位の DSM システムとして TreadMarks を用いたカスタム JVM である。

Java/DSM ではすべてのノードに JVM が存在し、すべてのオブジェクトが共有メモリ領域へ割り当てられる。リモートのオブジェクトはローカルオブジェクトと同じように扱うこと (メソッド呼び出しやインスタンスへのアクセスなど) ができる。各ノード間のスレッドの割り当ては自動的に行われ、スレッドの移動などはできない。

ガーベッジコレクションは各ノードで独立して行われるのが望ましいため、各ノードでリモートノードからの参照を管理する必要がある。そこでガーベッジコレクタはリモートノードからのローカルオブジェクトへの参照を保持する export リストとリモートオブジェクトへの参照を保持する import リストの 2 つのリストを管理する。そしてメッセージをリモートノードへ送る前に、 DSM ランタイムがガーベッジコレクタを呼び出し、送られるべきメッセージのチェックをさせる。ガーベッジコレクタはローカルオブジェクトへの有効な参照であるかをチェックして export リストへ追加する。メッセージが受け取るときもまた、 DSM ランタイムがガーベッジコレクタを呼び出す。ガーベッジコレクタでは有効なリモート参照であるかをチェックして import リストへ追加する。Owner では weighted reference counting を用いて、 export リストからリモートからの参照を削除するかどうかを決めている。

TreadMarks はページベースの DSM システムであり、異機種分散環境で必要となるデータの変換

(big endian<->little endian など) は行わないので、 JVM の方で行う必要がある。そのため、データ変換ではそのデータの型情報が必要となる。通常の JVM ではオブジェクトは、実際のデータへの参照である body と、 body へのポインタと型情報を格納した構造体へのポインタの 2 つのフィールドを持つ handle を持っている。これによって各オブジェクトがどのようなデータ型であるかが判明する。 Java/DSM で用いる JVM では効率のために、 body に handle へのバックポインタを持たせている。これによってどんなアドレスからもオブジェクトのサイズや型情報を得ることができ、データ変換が可能となる。

この他の Java/DSM での JVM の変更点はヒープを TreadMarks の共有メモリ割り当てルーチンを用いて割り当てていること、 JVM によってロードされるクラスは共有メモリ領域に置かれることである。

Java/DSM においても、システムのポータビリティは非常に低い。なぜなら、

- 分散共有メモリ機構を提供するために TreadMarks を利用している
- JVM へ (大幅な) 変更を施している

ためである。特に JVM への変更により、システムのポータビリティを低下させるだけではなく、既存の JIT コンパイラなどが利用不可能となるため性能面でも不利になると考えられる。もちろん、カスタム JVM へのカスタム JIT コンパイラを実現することも考えられるが、 Java/DSM の場合、分散環境・下位の分散共有メモリシステムを考慮する必要もあり、 JIT コンパイラの実装は容易ではなく、その移植も膨大な労力が必要となるであろう。

また、 TreadMarks はページベースの分散共有メモリシステムであるため、 Java の Object などのようなページサイズに比べはるかに小さいデータを扱う場合、 False Sharing が再び問題となる可能性もあり、十分な性能が期待できない。

cJVM

cJVM[AFT99] はサーバーアプリケーション向けのクラスタ (分散メモリ環境) 用の JVM で、クラスタ上に Single System Image を実現している。 cJVM の特徴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である。現在、プロトタイプが Myrinet で接続された PC(IBM Intellistation) の WindowsNT 上で動作しており、通信レイヤでは HPVM 上の MPI を用いている。

以下では cJVM の特徴とその実現方法について述べる。

分散ヒープとオブジェクトモデル

分散されたオブジェクトへのアクセスのために master-proxy モデルを用いている。あるオブジェクトが作られたノードにそのオブジェクトの master コピーがあり、それ以外のノードではそのオブジェクトへの proxy が使われる。アプリケーションでこの master と proxy を区別するために、「ある 1 つのメソッドに対して複数の実装を許す」という新たなオブジェクトの

モデルを定めている。そしてこのモデルでは、そのオブジェクトごとに適切な実装のメソッドを切り替えて呼び出すことができる。

cJVM では、あるノードが proxy へアクセスするときの方法は、メソッドシッピングである。proxy はあるメソッドの実行を master copy へリダイレクトする。これはメソッドテーブルを拡張することによって実現されている。 master オブジェクトでは通常のメソッドテーブルを保持している。 proxy オブジェクトのメソッドテーブルは、 master オブジェクトと同じメソッドが用意されているが、それぞれのメソッドは master オブジェクトへリクエストを送り結果を受け取るものである。この proxy オブジェクトでのコードはクラスがロードされたときに生成される。さらに、バイトコードがヒープへアクセスするのでバイトコードも修正する必要がある。バイトコードがアクセスしようとするヒープがローカルにあるかをチェックして、なければリモートアクセスを行う。このリモートアクセスに対処するために各ノードではリクエストを処理するサーバースレッドが動いており、リクエストを受け付けて結果を返す処理を行っている。

スレッドモデル

スケーラビリティの点から、異なるアプリケーションスレッドは異なるノードで実行される必要がある。そこでスレッドの生成方法を変更し、スレッドはロードバランス関数で決められる最適なノードで生成されるようにする (ロードバランス関数は変更可能)。さらにメソッドシッピング (のリクエスト) がきた場合、アプリケーションスレッドの論理的な一意性を保つために、新たに OS レベルのスレッドを生成して実行する。

プログラム解析

cJVM ではクラスのロード時に手続き間のフロー解析を行う。このフロー解析ではインスタンスメソッドのための this 参照の仕様を追跡し、新しく生成されるオブジェクトの型を解析し、メソッドが使うバイトコードを決定する。これによってメソッドのヒープへのアクセス方法と、 proxy の実装方法によって、メソッドをクラスわけできる。また、常にローカルでアクセスされるヒープを決めることができる。さらにメソッドの実装を直接クラスタを利用するように変更することもできる。

分散クラスロード

cJVM では各ノードで、独立して利用するクラスのコードのコピーを持っている。まず、コンスタントプールなどのクラスの内部のデータ構造を正しく構築する直接的な方法として各ノードでコードをロードする。また、アプリケーションからはクラスはオブジェクトとして見えるので、もしクラスが複数のノードで独立してロードされても、 1 つのオブジェクトに見える必要がある。さらに、クラスが初期化されるときアプリケーションやクラスの初期化メソッドが実行されるが、 2 回以上実行されてはならない。そこで、オブジェクトの master-proxy モデ

ルと同じようにクラスにも master コピーを用意する。そしてそのクラスをロードするノードでは、部分的にロードして master ノードへ問い合わせる。

以上のように、 cJVM では Java RMI 的な実現方法をとっており、それを JVM 内部で処理するようにしている。このような方法は Java との親和性が高いが、 master-proxy モデルによるオーバーヘッドが問題となる。しかし、 cJVM システムは科学技術計算や数値計算ではなく、一般的なネットワークにおける共同作業用アプリケーション的なものをターゲットとしているため、必ずしも高速性が求められるわけではないようである。

cJVM においても Java/DSM システムと同じように、 JVM への変更によりポータビリティと性能が損なわれる。しかし、 cJVM では JIT コンパイラを利用できるようインターフェースは定めてあるようである。また、メモリの管理単位が Java の Object 単位であるため、一般に False Sharing は起きにくいが、 Object が巨大になる (フィールドが多いなど) ような場合は考慮されていないため、性能低下を招く危険性がある。なお、 Primitive 型は Object とは異なり、メッセージパッシングでデータのやりとりが行われている。

TreadMarks　TreadMarks[ACD+96] はワークステーションクラスタにおいて並行コンピューティングをサポートするシステムである。 TreadMarks はメモリコンシステンシーモデルに リリースコンシステンシーモデルをさらに緩めた Lazy リリースコンシステンシーモデルを採用している。これにより通信によるオーバーヘッドを削減している。また、複数の書き込みを許すことにより、false sharing をの影響を減らすことによっても通信によるオーバーヘッドを削減している。ただ、通信のオーバーヘッドに比べてメモリ管理のコストは小さいという考えから、メモリ管理における最適化や改良などは行われていない。

クラスタ向けソフトウェアなど

以下では、クラスタ型並列計算機向けの並列実行環境やそれらで利用されている通信インタフェースなどについて述べる。

VIA

VIA(Virtual Interface Architecture)[htt97] とは、 (LAN や WAN よりも密集した) System Area Network (SAN) 向けの通信フレームワークである。

VIA はハードウェア保護のチェックとネットワークへの直接アクセス用の構造を確立するユーザーレベルのネットワークを提供する。従来のネットワークインターフェースでの経験を考慮して、廉価な実装も可能としている。また従来のイーサーネットカードと同様の descriptor processing モデルだけでなく通信ライブラリインタフェースも提供する。

ユーザーレベル通信では、共有通信資源への保護されたアクセスを提供する必要がある。従来、ユーザープロセスは通信をするためにシステムコールを発行する。ネットワークのスタックは、物理

的なネットワークカードを操作するカーネルのデバイスドライバを通して、多くのプロセスからのネットワークへの通信を多重化している。このような方法ではユーザーのアドレス空間を NIC 上へマップしたり、カーネルを経由しないことが容易なため、一人のユーザーへの通信を制限したり、ある通信が他のプロセスの通信を妨げることが可能となってしまう。ユーザーレベルネットワークでは、ユーザープロセスがネットワークのエンドポイントをあらわすユーザーのアドレス空間へアクセスすることを許している。このエンドポイントへ書いたり読み込んだりすることによってメッセージを送信したり受信したりする。

VIA ではエンドポイントはネットワークインタフェースのディスクリプタキューに似ており、virtual interface(VI) と呼ばれる。 OS はユーザープロセスのアドレス空間へ NIC の資源にマップすることで VI の生成と破棄を行う。これによってアプリケーションは他のプロセスの VI へアクセスすることはできなくなる。 VIA は、 point-to-point で connection-oriented であり、 OS が VI の接続や切断に関与したり接続の量や詳細をコントロールすることでネットワークの利用を OS が制御することを提供している。

VI のセットアップ・接続・管理はユーザーレベルのライブラリを通してカーネルドライバと対話的に行われる。カーネルドライバは、ユーザーレベルのプロセスが行わない、 VI の生成接続などを NIC へ通知するための NIC 資源へアクセスする。

各 VI は送信と受信のキューのペアとそのペアへのドアベルを持っている。これらへは直接アクセスできず、ユーザーレベルのライブラリを経由してアクセスされる。データバッファとキューはネットワークカードがアクセスするホストのメモリ領域に割り当てられる。 VIA ではこれらのメモリ領域を registered memory と呼ぶ。アプリケーションによるパケットの送受信は、適切なディスクプリタを生成し、ユーザーレベルライブラリによる NIC の適当なドアベルを鳴らすことによって行われる。このときアドレスマッピングは OS によって行われるため、正確な保護と仮想化が維持される。

このユーザーレベルライブラリ上に MPI などを実装することは可能だが、 VI のバッファにマップできないデータ領域がある場合はデータのコピーが必要となる。

以下に VIA の実装例とその特徴について簡単に述べる。

Berkely VIA[BGC98] ネットワークインタフェースとして Myrinet を用いた VIA。 VI の処理は Myrinet 上で行っている。レイテンシは AM2 より小さいが、バンド幅は AM2 と同等である。

M-VIA NERSC の PC クラスタプロジェクトの一部として開発されている。 GigaEther と 100Base 向けの VIA。 TCP に比べて 1/3 のレイテンシだが、 100Base の場合はバンド幅は TCP と変わらない。 GigaEther の場合は 2 倍ほどのバンド幅である。

JaguarVIA[WC99] Java 向けの高速な VIA インタフェース。 VIA には Berkery VIA を利用している。 JNI ではなく JIT コンパイラを用いて、 VIA のメソッド呼び出しを native への直

接呼び出しとして実現しているため、ラウンドトリップタイム・バンド幅ともに C(native) における実装と同等の性能を達成している。

SCore & PM & MPC++

SCore[HTI+96] は RWCP で開発されている、ユーザーレベルで実現されたクラスタ型並列計算機向け並列実行環境である。 SCore システムは通信ライブラリ PM、並列ジョブスケジューラ・並列実行環境 SCore-D、並列プログラミング言語 MPC++ などから構成され、スケジューリングの機構など並列実行に必要な機能を提供する。複数ユーザーによる共有利用を目的としており、ユーザーの対話性も視野に入れた設計が成されている。

PM[THIS97] [THI96] は当初、 Myrinet を対象として開発された通信ライブラリ・ドライバで、複数のチャネルを持ち、レイテンシが低く、高スループット、可変長メッセージのサポート、メッセージ配送の保証、メッセージ順序の保存、コンテキストスイッチの搭載、マルチキャストのサポート静的ルーティングなどの機能・特徴がある。

また、 PM はマルチユーザープロセスからの利用が可能となっている。このため、 PM ではチャネルを用いて、物理的に一つしかないネットワークをソフトウェア的に多重化している。チャネルは複数個提供されており、プロセスはチャネルを介して通信する。ある送信ノードのチャネルから発信されたメッセージは、受信ノードの同じチャネルに配送される。

現在、 PM は Myrinet 以外に UDP や Giga-Ethernet 上にも実現されている。

MPC++ 2.0 Level 0[Ish96] は C++ のテンプレートと継承の機能を使って高レベルな通信モデルのプログラミングをサポートしている、並列プログラミング言語である。本システムでは MPC++ 2.0 Level 0 の MTTL の機能であるローカル / リモートでのスレッド生成、グローバルポインタ、リダクションなどを用いる。また、リフレクション機能をサポートした Level 1[Ie96] もある。

HPVM & FM

HPVM(High Performance Virtual Machine)[CPL+97] は通信レイヤの FM(Fast Message)[PKC97]、メッセージングの MPI-FM[LC97]、スケジューリングの FM-DCS[SPWC98]、 SHMEM の Put/Get-FM[GC98]、 GlobalArrays-FM などで構成されたクラスタ向けソフトウェア群である。

クラスタの形態には resource stealing なモデルといわゆるクラスタのモデルの 2 つがある。 Condor や Utopia、 LSF のような resource stealing モデルは異機種で、低性能なネットワークで結ばれており、逐次のジョブにおいて high throughput を達成している。いわゆるクラスタは同機種で高速なネットワークで接続されていて高性能を達成している。

HPVM ではこの 2 つを一体化する。

- 高性能な通信に標準的でハイレベルな API を提供
- スケジューリングとリソース管理の一体化

- 異機種環境

また、通信のパフォーマンスは並列計算では重要となるので、HPVM では low-level、high-level の 2 つの通信レイヤから構成されている。low-level なものとして FM(Fast Message)、high-level なものとして MPI、SHMEM、Global Arrays を提供している。

通信 ハイパフォーマンスを提供するハードウェアがあっても、伝統的なソフトウェア・ハードウェアの構造だと、ハードウェアの性能をアプリケーションのレベルまで持ってくるのが困難である。例えば、TCP/IP は 10Mbps のイーサーネット向けに作られたので、100 μ 秒のオーバーヘッドがパケットごとにかかる。しかし、Myrinet だと 2 桁以上速い. そこで HPVM ではハードウェアの性能を生かす、低レベルな通信として Fast Messages(FM) を用意し、さらにその上にハイレベルな API を提供している。

Fast Message FM はもとは Berkeley の Active Messages である。FM1.1 の API には long、short メッセージの送信と、ネットワークからのメッセージの取りだしの 3 つの関数がある。他のメッセージングレイヤとの違いは FM 上に構築されるソフトウェアに対してサービスの保証とメモリ階層の制御が提供されることである。ハイレベルのレイヤでパフォーマンスの低下を招かないためには、低レベルのメッセージングレイヤが次のものを提供する必要がある。

- 通信の信頼性
- メッセージの処理の順序
- 通信スケジューリングの制御

また、FM 上にハイレベルのメッセージングレイヤを実装した場合、次のような問題がある。

- どれだけの通信データが処理されるかではなく、いつ通信データが処理されるかの制御を許可している。
- FM へデータを送る前にヘッダーを付けるためにユーザーのバッファからコピーするコスト。
- FM 内部のバッファから上位のメッセージングレイヤのバッファさらにユーザーのバッファへコピーするコスト。

そこで FM2.0 では data pacing と streamed message を実現した。data pacing によりで処理されるデータの量を制限することができるようになり、streamed message では送信側がメッセージをすべて送り終わる前に受信側でメッセージを処理できるようになりさらにデータのコピーを減らすことができる。また、streamed messages では上位のメッセージングレイヤがメッセージの一部を受け取って、その後のデータを受け取るかどうかを決めることもできる。FM2.0 の性能は FM1.1 とほぼ同等である。

また、FM2.0 を PentiumPro の WindowsNT クラスタへも実装した。PC には write combining をサポートした PCI バスがあるため、従来用いてきた DMA によるオーバーヘッドも避けられる。

Fast Message は GAM インタフェースを用いているが、ハンドルの暗黙実行は行わない。その代わりに、ユーザープロセスは届くメッセージ用のバッファを提供し、デッドロックを回避するためにフローコントロールを行っている。またバルク転送機構なども備えている。

API 次にハイレベルのレイヤでは高機能で使いやすい API を提供する必要がある。しかし、ハイレベルのレイヤではさらにオーバーヘッドが増し、一般的に性能が低下する。FM ではハイレベルのメッセージングレイヤと低レベルのレイヤとのギャップをなくすことにより、高性能な MPI-FM と Global Arrays の 2 つの API を FM 上に実現した。

MPI-FM MPI-FM は MPICH をベースにしている。MPICH ではさらに 2 つのレイヤに分かれており、1 つは ADI という 25 個ほどの関数からなり、もう 1 つは ADI により構築された 125 個の関数である。つまり、FM 上で MPI(MPICH) を動かすためには ADI のみを実装すればよい。MPI-FM は MPP(IBM SP2) に比べて低レイテンシでさらに 2kbyte 以下のメッセージで高いバンド幅を達成している。

Global Arrays Global Arrays は共有アドレス空間の巨大な配列のセクションへアクセス・制御するための Put/Get のセマンティクスを用いたプログラミングモデルである。Global Arrays は SHMEM を用いて実装されているため、SHMEM も移植する必要がある。

AM

Active Message はもともと並列スパコン向けに作られたものだが、多くのネットワークインタフェースに実装された。AM は通信のための一種のアセンブリ言語のつもりであった。内部のキュー構造を見せるのではなく、非常に単純な RPC のようなプログラミングインタフェースを提供していた。小さなメッセージは手続き呼び出しによってレジスタ経由の引数として、大きなメッセージはメモリ間の転送として扱われている。ハンドラは引数としてのメッセージデータがリモートで利用可能になると暗黙のうちに投げられ、リモートのユーザー空間で実行される。ハンドラは割り込みやネットワークのポーリングによっても実行される。

さらに General Active Message(GAM) ではプロセスにユーザーレベルのネットワークを暗黙のうちに関連させ、AM2 以降は仮想ネットワークのエンドポイントを明示的にし、ユーザープロセスが複数のエンドポイントを持つことを可能にし、スレッドによる実行モデルを提供した。AM2 のエンドポイントは単純な送信・受信のキューを持つが、内部のデータ構造は隠されている。

U-NET

U-NET[OW98] はユーザーレベルのデバイスドライバから見えるディスクリプタキューとバッファ構造を仮想化しようとしている。U-NET は VIA にかなり近いが、キュー構造やバッファーエ

リアのためのアドレス解決の方法が異なる。 U-NET は最初の Ethernet や ATM LAN 上の高速なユーザーレベルの通信レイヤである。

2.2 自己反映計算

2.2.1 理論

自己反映計算 (Reflection) とは、計算システムが自分自身の構成や計算過程に関する計算を行うことである。自己反映計算の機能を持つシステムでは、システムが自己の構成・計算過程の表現を参照するだけでなく変更することもできる。そしてそのような表現に対する変更は、自己の構成・計算過程そのものに反映される。

プログラミング言語やオペレーティングシステムといった記述系を自己反映計算の機能を持つシステムとして構成することが、柔軟なソフトウェアの構築に有用であることが認識されているのは、次のような理由からである。

拡張性 言語やオペレーティングシステムをアーキテクチャと実現方式の両方の面でカスタマイズできるようにする。アーキテクチャのカスタマイズにより、アプリケーションの持つ論理構造と言語やオペレーティングシステムの提供する抽象化機構とのギャップを少なくすることができる。一方、既存のデータ表現形式によって対象の持つ論理構造を素直に表現できるにもかかわらず、実行効率が著しく悪くなる場合がある。特定のデータ表現の実現方式を自己反映計算でカスタマイズすることによってこのような効率の低下を避けることができる。

動的適応性 計算資源の最適配置や制御方式を、静的解析によって求めることが困難な問題（分散離散事象シュミレーションにおいて、シュミレート対象の挙動が予測困難な場合など）、もしくはモバイルコンピューティング環境のように構造が動的に変化するような計算環境に対しては、自己の構造や計算過程を適切に操作し、計算環境に動的に適応できるようなソフトウェアの構成方式が求められる。自己反映計算の機構を持った言語やオペレーティングシステムはこのようなソフトウェアを自然な形で記述できる。

以上の 2 点に共通するのは本来目的とする計算のレベル (ベースレベル) と、その計算・記述方式を制御あるいはカスタマイズするレベル (メタレベル)、そしてその間のインタラクションが同一の記述系に基づいていることである。自己反映計算の機構を持った言語やオペレーティングシステムのアーキテクチャが適切に設計されていれば、これらのレベルを独立したモジュールとして分割し、再利用することが可能になる。このようなモジュール化の方式は従来のプログラミング言語の設計やプログラミング方法論ではほとんど意識されることはなかった。しかしこれによって、従来個別の方法で導入されてきた機能（資源管理、例外処理、デバッグ機能、外界とのインターフェースなど）を、言語を提供するアーキテクチャにもとづいた整合的な方式で導入できる。つまり自己反映計算とは新しいモジュール化手法を与えるソフトウェアの構成原理とみなすことができる。

メタオブジェクト・メタクラス

オブジェクト指向言語において、クラス・メッセージ・メソッドなどをオブジェクトとして統一的に実現している言語・計算モデルの諸概念を実現しているオブジェクトをメタオブジェクトという。メタオブジェクトは言語のメタインタプリタの実現である。メタオブジェクトの導入によって、オブジェクト指向言語は手続き的リフレクションを実現することができる。

メタオブジェクトプロトコル

メタオブジェクトのインタフェースの仕様をメタオブジェクトプロトコルという。メタオブジェクトプロトコルでは、メタオブジェクトのクラス定義そのものは与えずに、それらのクラスに関係する総称関数および付随するメソッドの自然言語による外部仕様を与えている。これにより具体的な実装からは独立にメタオブジェクトの仕様や利用方法を与えている。また、メタオブジェクトのカスタマイズは、継承を用いることによって局所的に行うことができる。この結果、カスタマイズしたシステムの安全性や再利用性を確保することができる。メタオブジェクトプロトコル (MOP) には次の 2 タイプある。

ランタイム MOP 初期の手続き的自己反映計算のほとんどが、このタイプのメタオブジェクトプロトコルを実装していた。オーバーヘッドが大きい。例えば、 CLOS のメタオブジェクトプロトコルではメタオブジェクトの実行時のメソッドインヴォケーションによってオブジェクトシステムを実行しているので、実行時のパフォーマンスを挙げることは難しい。

コンパイルタイム MOP コンパイルタイム MOP は言語の振舞の変更などのメタ計算を、ソースコードの変換として記述させ、コンパイル時にそのメタ計算を実行することで実行時のオーバーヘッドを減らすことができる。しかし、ランタイム MOP とは違い、コンパイラが扱うデータを独立して制御することが難しい、 MOP によって変更され異なる方針でコンパイルされたデータを正しく組み合わせるのが難しいなどの問題もある。 [KLRR92]

2.2.2 関連研究

OpenC++

OpenC++[Chi95],[Chi97] は C++ 言語のための Compile-time MOP を提供する。適切なメタクラスを記述し、それを OpenC++ コンパイラによってプログラム変換を含む適切な処理を行うコンパイラを生成する。生成されたコンパイラを用いてベースレベルのプログラムの変換、コンパイルを行う。 Compile-time MOP によりメタ計算の実行時オーバーヘッドがなくすことができる。

OpenC++ のメタオブジェクトプロトコルの基本的なアーキテクチャは、クラスのメタオブジェクト、関数のメタオブジェクトがあり、これらのメタオブジェクトがプログラムの振舞を操作している。 OpenC++ の場合、普通のオブジェクトはランタイムにのみ存在し、メタオブジェクトはコンパイルタイムにのみ存在する。このため、メタオブジェクトがオブジェクトの振舞はプログラムをコ

ンパイルする時に操作するということなる。このとき、メタオブジェクトは適当にプログラムのトップレベルの定義を変換し、必要があれば変換したコードのための実行時の関数や型、データなどを追加する。これによってメタオブジェクトは実行時のスペースとスピードでペナルティーがまったくない。 OpenC++ の処理プロセスは以下のようになっている

- プログラムの解析が終わると、メタオブジェクトプロトコルは２つのメタオブジェクト、クラス Point のためのメタオブジェクトとメンバー関数 MoveTo() のためのメタオブジェクトをつくる。普通、クラスのメタオブジェクトはクラス Class のインスタンスとなる。これにはクラスの定義によって与えられる名前やベースのクラス、メンバーなどの情報が含まれている。関数のメタオブジェクトは普通クラス Function のインスタンスとなる。これにも同じように関数の定義からくる名前やパラメータ、関数本体の解析木の情報が含まれている。これはメンバー関数なので、このメンバー関数を持っているクラスのメタオブジェクトへのポインタも持っている。

- メタオブジェクトの定義のあるもとのバラバラになったコードの代わりに適当なコードをつくるためにメタオブジェクトを呼ぶために、各メタオブジェクトのメンバー関数の CompileSelf() が呼び出される。その後、クラスのメタオブジェクトはそれらのクラスのための普通の C++ 定義を解析木の形で生成する。これは普通、もとの定義だけを生成する。同じように、関数のメタオブジェクトはこれもまた解析木の形で普通の C++ の定義を生成する。これも普通、もとの定義だけを生成する。関数の本体を変換するために、関数のメタオブジェクトは関数の解析木を一つ一つ辿って適当なクラスのメタオブジェクトに各関数本体のコード片のを変換するかどうか照会するサブプロトコルを呼ぶ。関数のメタオブジェクトはクラスのメタオブジェクトのクエリーをつくることによって関数本体をコンパイルするので、プログラムのコンパイルは主に、クラスメタオブジェクトが行う。クラスのメタオブジェクトの通常の動作は一切変換をせずに、与えられた解析木を返すだけである。

- あるメタオブジェクトからつくられた解析木はまとめられて、 C++ のソースファイルに変換される。このソースファイルは C++ コンパイラによってコンパイルされる。メタオブジェクトプロトコルが C++ コンパイラの分離したプリプロセッサとして実装されているので、変換された C++ のソースファイルはテキストファイルなので C++ コンパイラはこのファイルの内容を再び解析しなくてはならない。 OpenC++ ではこのオーバーヘッドを避けるために、C++ コンパイラとメタオブジェクトプロトコルを統合してある。

このように OpenC++ のメタオブジェクトプロトコルは OpenC++ から C++ へのソース to ソースの変換をコントロールするものである。

OpenJIT

OpenJIT[松岡 98][MOS+98] は、自己反映計算 (リフレクション) に基づいた Open Compiler(開放型コンパイラ) 技術をベースとして、アプリケーションや計算環境に特化した言語の機能拡張と最適化が行える JIT コンパイラである。 OpenJIT では、 JIT コンパイラ自身が Java のクラスフレームワークとして記述されており、クラスライブラリの作成者がクラス単位でそのクラスに固有の最適化モジュールを組み込むことを可能としている。これにより、様々な計算環境・プラットフォーム・アプリケーションに対する適合や、複雑な最適化を組み込むことも可能となっている。

OpenJIT の機能は、大きく分けてフロントエンド系 [小川 99][OSM+00] の処理とバックエンド系の処理の 2 つから構成される。フロントエンド系は、 Java のバイトコードを入力とし、高レベルな最適化を施して、バックエンドに渡す内部的中間表現、あるいは再びバイトコードを出力する。バックエンド系は、得られた中間表現あるいはバイトコードに対し、より細かいレベルでの最適化を行い、ネイティブコードを出力する。

以下では、フロントエンド系とバックエンド系のそれぞれの処理について述べる。

フロントエンド OpenJIT の起動は、 JDK の提供する JIT インタフェースを介してメソッド起動ごとに行われる。 OpenJIT が与えられたメソッドに対して起動されると、フロントエンド系は対象となるバイトコード列に対して以下の処理を行う。

まず、ディスコンパイラモジュールがバイトコードから AST を逆変換により得る。この際には、与えられたバイトコード列から、元のソースプログラムから生成されるコントロールグラフの復元を行う。同時に、アノテーション解析モジュールによって、対象となるクラスファイルに何らかのアノテーション情報が付記されていた場合にその情報を得る。この情報は、AST 上の付加情報として用いられる。

次に得られた AST に対し、最適化モジュールによって最適化が施される。最適化は、通常の AST とフローグラフを用いた構築・解析・変換を行い、最適化モジュールのサポートモジュールであるフローグラフ構築モジュール、フローグラフ解析モジュール、プログラム変換モジュールを用いて実現される。フローグラフ構築モジュールは対応するデータ依存グラフ、コントロール依存グラフ等を生成する。フローグラフ解析モジュールはこれらのデータフローグラフに対するデータフロー問題、マージ、浮動点検出等の諸ルーチンをメソッドとして提供する。プログラム変換モジュールは解析情報を元にテンプレートマッチングによって AST 上の変換を行う。

変換後の結果はバックエンドが必要とする中間表現ないしバイトコードに変換されて、出力される。

バックエンド バックエンド系はフロントエンド系によって出力された中間表現ないしバイトコードに対して、低レベルの最適化処理を行い、ネイティブコードを出力する。ネイティブコード変換モジュールはバックエンド系処理全体の抽象フレームワークであり、バックエンド系のモジュールのインタフェースを定義する。このインタフェースに沿って具体的なプロセッサに応

じたクラスでモジュールを記述することにより、様々なプロセッサに対応することが可能となる。

まず、中間コード変換モジュールによって、バイトコードの命令を解析して分類することにより、スタックオペランドを使った中間言語の命令列へと変換を行う。フロントエンド系が中間表現を出力する場合はこの形式で出力するため、この変換処理は省略される。次にRTL 変換モジュールは、得られた命令列に対し、スタックオペランドを使った中間言語から仮想的なレジスタを使った中間言語 (RTL) へ変換する。この際、バイトコードの制御の流れを解析し、命令列を基本ブロックに分割する。また、バイトコードの各命令の実行時のスタックの深さを計算することで、スタックオペランドから無限個数あると仮定した仮想的なレジスタオペランドに変換する。

次に、 Peephole 最適化モジュールによって、 RTL の命令列の中から冗長な命令を取り除く最適化を行い、最適化された RTL が出力される。最後に OpenJIT SPARC プロセッサコード出力モジュールにより、 SPARC プロセッサのネイティブコードが出力される。出力されたネイティブコードは、 JavaVM によって呼び出され実行されるが、その際に OpenJIT ランタイムモジュールを補助的に呼び出す。

EPP

EPP[IR97] は、 Java の柔軟な言語拡張 (文法や意味) を提供するプリプロセッサである。 EPP 自身、 EPP で拡張された Java 言語で記述されている。 EPP における言語拡張では、 EPP plugin を利用する。 plug-in 同士の衝突が起きなければ、複数の plug-in を同時に取り込むこともできる。 EPP プリプロセッサが出力するコードは Java ソースコードであるため、通常の Java コンパイラでコンパイルすることができる。 EPP は汎用手続き型言語的な記述が可能なため、文脈依存、大域脱出など、 BNF 記法では書きにくい ad-hoc な処理が行ないやすい。

EPP の言語拡張機能は、 plugin でもある Symbol と SystemMixin の機構によって実現されている。 Symbol はコロンとそれに続く identifier または文字列リテラルで表される。この Symbol とは C 言語の enum 文で宣言された定数に近いが、あらかじめ有限個の要素を宣言しておく必要がなく、ソースコード中必要に応じて任意の名前 (文字列) を持つ symbol を使うことができる。文字列リテラルにも近いが、文字列とは違い、実行時にはポインタの比較だけで高速に同一性の判定ができ、名前を指定して symbol を動的に生成することもできる。なお、同一の名前を持つ symbol リテラルは、同じ identity を持つインスタンスを参照することが保証される。

SystemMixin は、 複数のクラスの部品となる mixin をいくつかまとめたもので、 collaboration-degin を実現している。 mixin は定義時に特定のスーパークラスを指定しないで定されたクラスである。 mixin は後で他のクラスによて多重継承され、直列化されることよりスーパークラスが決まることを想定して定義される。 EPP は標準の Java 言語パーザを実現する mixin とソースコードで指定された PlugIn を組み合わせて 1 つのプリプロセッサとして動作する。

CLOS

CLOS[KdRB91] は Lisp にオブジェクト指向言語機能を取り入れたものである。この CLOS と、同様に Lisp にオブジェクト指向言語機能を取り入れた Flavors や LOOPS の大きな違いは メタオブジェクトプロトコルを持つことである。 CLOS では CLOS を実現するクラス、メソッドなどの諸要素はメタオブジェクトとして表現されている。そしてこれらのメタオブジェクトは CLOS そのものであり、つまり CLOS のメタインタプリタを構成している。

また、 CLOS のメタオブジェクトは普通のオブジェクトと同じアドレス空間に存在している。よって、各メタレベルで共通のメタオブジェクトはすべて同一のものである。つまり、無限階のリフレクティブタワーの代わりに循環的な構造を持つ。この循環的構造はリフレクティブタワーによるものとほぼ等価な系を構成する。すべてのオブジェクト・メタオブジェクトが同一のアドレス空間に存在していることから、ユーザープログラム内からメタオブジェクトへのアクセスには特別な言語機構は必要なく、通常の総称関数呼び出しを用いることができる。またカスタマイズしたいメタオブジェクトに関するクラス定義も、通常の定義と同様に行うことができる。

CLOS のメタオブジェクトプロトコルではメタオブジェクトのクラス定義は与えられていない。そのかわりに、それらのクラスに関係する総称関数および付随するメソッドの自然言語による外部仕様を与えている。つまり、 CLOS のメタオブジェクトプロトコルでは具体的な実装からは独立にメタオブジェクトの仕様や利用方法を与えている。

OpenJava

OpenJava[TCN97] では JDK の ReflectionAPI を拡張する形で、既存の introspection に加えて intercession を提供する。 introspection は実行時にクラスのメンバなどを調べたり利用する機能でクラスの実装情報を知ることができる。 intercession は実行時にメソッド呼び出しの動作を変更したりクラスに新たなメソッドを追加したりする機能である。この intercession の機能により、言語自体の拡張を備えたライブラリを作成することができる。

しかし、この intercession の実行効率のよい実装は困難である。これを改善するために OpenC++ でも用いられたコンパイルタイム MOP を用いている。 OpenC++ のコンパイルタイム MOP は静的な型に基づいているが、 JDK の ReflectionAPI ではオブジェクトの動的な型に基づいているため、 OpenJava でも動的な型に基づくコンパイルタイム MOP を採用している。これにより単一の MOP が introspection と intercession を提供できるようになり、プログラマは両者の区別を意識せずに自己反映計算を伴うプログラムを書くことができる。

BCA

Binary Component Adaptation(BCA)[KH98a][KH98b] はコンポーネントのバイナリレベルでのオンザフライな適応を提供している。 BCA はソースコードへのアクセスなしで、プログラムのリリース間のコンパチビリティを保証しながら、コンポーネントのロード時かロード中にコンポーネントのバイナリを変更する。つまり adaptation は、対象が以前のリリースと互換性がある限り、コン

ポーネントの新しいバイナリと以前のバイナリの互換性を保証している。この adaptation では、各クラスに対する delta ファイルを用いて実現される。 delta ファイルでは拡張・変換したいクラスに対して用意し、 delta ファイルコンパイラでコンパイルする。そして、クラスのロード時に modifier によって既存のクラスファイルに対して delta ファイルを適用し、クラスファイルの内容を変更する。

JDSM システムの実装 3 ≡

JDSM システムは、提案で述べた手法を Java 言語に適用したシステムである。 JDSM システムは、大きく JDSM ランタイム部と JDSM ランタイム部を利用するように変換を行うプログラム変換部に分けられる。

JDSM ランタイム部は Java 言語レベルの分散共有メモリ機能を提供しており、さらに通信インタフェース部と DSM 部に分けられる。通信インタフェース部では、 DSM 部で利用する通信 API を定めて複数の異なる通信デバイスを利用可能としており、その実装も与えている。 DSM 部では、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 に基づいて記述されている。また、DSM 部では複数のメモリ管理モデルを利用できるような API を定めており、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においても同様に利用することが可能となっている。

プログラム変換部は、 Java 言語で記述された共有メモリ計算機向け並列プログラムを解析し、JDSM ランタイム部を利用することでクラスタ型並列計算機のような分散メモリ環境でも動作するようソースプログラムレベルの変換を行う。この変換には、 Java 言語の Reflective な処理系の 1 つである OpenJIT を利用している。異なる環境でのプログラム変換においては、メモリ管理モデルや通信インタフェースの違いは基本的に JDSM ランタイム部で隠蔽されているので、利用するクラスを変更するだけで済む。

以下では、 JDSM システムにおける通信インタフェース、分散共有メモリ、プログラム変換の実装について述べる。

なお、ソフトウェア設計書における機能仕様に沿った論文の構成をとっていないため、表 3.1 に機能仕様の中機能と各説明の対応を示す。

機能仕様での項目	本論文での項目 () 内は本論文のセクション番号に対応する
JDSM プログラム解析メタクラス部	プログラム解析と変換 (3.3)
分散クラス初期化解析	初期化部 (3.3.1)
read アクセス解析	read アクセス (3.3.1)
write アクセス解析	write アクセス (3.3.1)
JDSM プログラム変換メタクラス部	プログラム解析と変換 (3.3)
分散クラス初期化変換	初期化部 (3.3.2)
read アクセス変換	read アクセス (3.3.2)
write アクセス変換	write アクセス (3.3.2)
write ロック追加変換	write アクセス・メソッド呼び出し (3.3.2)
JDSM ランタイム部	分散共有メモリ (3.2)
JDSM 初期化	initialize、 finalize メソッド (3.2.1)
JDSM 分散オブジェクト処理	register メソッド (3.2.1)
JDSM Read 処理	update メソッド、他のノードからのリクエスト処理 (3.2.1)
JDSM Write 処理	acquire メソッド、 release メソッド、他のノードからのリクエスト処理 (3.2.1)
JDSM 通信インタフェース部	通信インタフェース部 (3.1)
通信インタフェース	通信 API(3.1.1)
汎用通信クラス	JVIA(3.1.2)

表 3.1: ソフトウェア設計書における機能仕様と本論文の項目との対応

図 3.1: 通信インタフェースの構成

3.1 通信インタフェース部

通信インタフェースの実装としてはいくつかの方法が考えられるが、ここでは可搬性やできるだけ多くの通信インタフェースを利用できるように図 3.1 のような構成で実装した。

利用する下位の通信インタフェースとして以下のものが考えられる。

- TCP/IP(Socket)
- VIA(Virtual Interface Architecture)
- PM などのメッセージ通信ライブラリ

しかし、各通信インタフェース毎に異なる実装 (API やデータ構造) を実現すると、各通信インタフェース毎にアプリケーションも変更する必要が出てくるため、システムのポータビリティが損なわれる。

そこで、通信インタフェースを共通の API インタフェースで隠蔽することによって、異なる通信インタフェースを利用した場合もその通信インタフェースで 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。

以下では API と通信インタフェースの実装について述べる。

3.1.1 通信 API

通信インタフェース API はパッケージ jdsm.comm のインタフェース Comm で定義されている。このインタフェースでは以下の基本的な API を定義しており、下位の通信レイヤを用いるときはインタフェース Comm を実装するクラスを定義する必要がある。インタフェース Comm では表 3.2 のようなメソッドが宣言されている。

次に、 VIA を用いたこの Comm インタフェースの実装について述べる。なお、 VIA を用いた実装以外にも、 Socket を用いた Comm インタフェースの実装も行った。

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	メッセージが到着しているかどうか確認する

表 3.2: インタフェース Comm で宣言されているメソッド

3.1.2 JVIA

```
package jdsm.comm;
public class VIAComm implements Comm
```

本研究での Comm の実装では通信インタフェースとして VIA を用いる。 Java 言語向けの VIA のインタフェースが存在しないため、まず Java 向けの VIA の Wrapper である JVIA を実現し、その上で Comm インタフェースの実装を行った。以下では、 JVIA の実装について述べ、次に JVIA を用いた Comm インタフェースの実装について述べる。

JVIA

下位の通信レイヤとして VIA の実装である M-VIA を用いる。 VIA は基本的に C 言語向けのインタフェースしか持っていないため、 Java から VIA を利用するためには JNI(Java Native Interface) を用いる必要がある。そこで、 VIA で定められているデータ構造・ API を実現するパッケージ (JVIA - VIA wrapper for Java) を実装した。

VIA の API は全て jvia.JVIA クラスにおいて native method として定義する。 VIA のデータ構造・タイプはパッケージ jvia のクラスとして定義する (表 3.1.2)。

これらのクラスは C 言語における同名のデータ構造に対応する。 Java にはポインタが存在しないため実際のデータ領域へのポインタは、新たに VIP_BUFFER クラスを定義してそれへの参照としている。また、 Java 側で扱っているデータを native 側へ、逆に native 側から Java 側へマーシャリングする必要がある。

このマーシャリングでは、

- オブジェクトのクラス情報の取得

VIP_ADDRESS_SEGMENT	VIP_CONN_HANDLE	VIP_CONTROL_SEGMENT
VIP_CQ_HANDLE	VIP_DATA_SEGMENT	VIP_DESCRIPTOR
VIP_DESCRIPTOR_SEGMENT	VIP_MEM_ATTRIBUTES	VIP_MEM_HANDLE
VIP_NET_ADDRESS	VIP_NIC_ATTRIBUTES	VIP_NIC_HANDLE
VIP_PROTECTION_TAG	VIP_PVOID	VIP_PVOID64
VIP_SEGMENT	VIP_VI_ATTRIBUTES	VIP_VI_HANDLE

表 3.3: VIA で利用されるデータタイプ・構造

- クラスのフィールド情報の取得
- フィールドデータの取得
- native 側のデータ構造へコピー
- フィールドデータ (= オブジェクト) をさらにマーシャリングする必要がある場合は繰り返す

という操作を行う。

このように Java 側で別途データ領域を確保している場合、 native 側でコピーを行う場合以外ロックをする必要がなく自由に扱えるというメリットがある反面、このマーシャリング・コピーがオーバヘッドとなる可能性がある。将来的にはマーシャリングが原因で性能低下を引き起こしている場合、 Java 側と Native 側のデータを共通化し、アクセスする場合はその領域を pinning することも考えている。

処理の流れ

次に、この JVIA を用いた Comm インタフェースの実装である VIAComm における初期化・終了処理と送信・受信の処理について述べる。

初期化 初期化は VIAComm.initialize メソッドを呼び出すことで行われる。初期化では、ネットワークインタフェースの初期化、ノード情報の取得、他のノードへの接続用エンドポイント VI の領域確保と初期化、通信用バッファ (メモリ) の確保と登録、通信用 Descriptor の登録、 CQ(Completion Queue) の登録、メッセージキューの確保を行う。さらに、他のノードからの接続を処理するためのスレッド (Listen thread) を生成する。他のノードへの接続も行う。全ての接続が完了したら、他のノードとの通信を処理するスレッド (Handler thread) を生成し、他のノードからの受信を処理する。 VI の受信は 1 つの Completion Queue に関連付けられ、送信は各 VI ごとの Completion Queue に関連付けられる。

送信 データの送信は VIAComm.send メソッド、もしくは VIAComm.asend メソッドで行われる。前者は送信が完了するまでブロックする。後者は送信を行ったあと、完了を確認しない。

この場合、送信の完了はVIAComm.sendDoneメソッドを呼び出すことによって確認する。VIAComm.sendメソッドは実質的にVIAComm.asendメソッドとVIAComm.sendDoneメソッドの組合せで実現されている。送信の完了は各VIの送信用Completion Queueを利用する。あるVIに対する送信は、そのVIの送信用Completion Queueを監視して、完了を確認する。

受信 データの受信は全てHandler threadで行われる。Handler threadでは受信用Completion Queueの状態を確認し、Queueが空であれば、yield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。その後、次の受信処理を行うために受信命令を発行しておく。受信したメッセージはVIAComm.recvメソッドで取り出すことができる。このとき、メッセージキューが空であった場合、ブロックする。メッセージを受信したかどうかを確認するためにはVIAComm.probeRecvメソッドを呼び出す。このメソッドはメッセージが空でなければtrueを、空であればfalseを返す。

終了処理 終了処理では、Handler threadの終了、メッセージキューが空になるまでポーリングする。その後、各ノードとの通信を切断し、Descriptor、VI、バッファを解放する。なお、エラーが発生した場合も強制的に終了処理が呼び出される。終了処理中のエラーは無視される。

3.2 分散共有メモリ

JDSMシステムにおける分散共有メモリの実装は、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に基づいて記述されている。また、DSM部では複数のメモリ管理モデルを利用できるようなAPIを定めており、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においても同様に利用することが可能となっている。

JDSMシステムにおいてはメモリ管理モデルの実装手段として、抽象化クラスSharedObjectPoolのサブクラスにおいて実装する方法をとっている。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。そのためにも、抽象化クラスSharedObjectPoolではメモリ管理モデルを実装する上で適切かつ柔軟に対応できるフレームワークを提供している必要がある。そこで、SharedObjectPoolでは表3.4の7つのメソッドが宣言されている。実際のメモリ管理クラスではこの7つのメソッドを実装することになる。

この抽象化クラスSharedObjectPoolを用いることで、多くのメモリ管理モデルが利用できると考えている。しかし、その範疇に収まらないメモリ管理モデルを利用したい場合が存在するかもしれない。その場合は、この抽象化クラスSharedObjectPoolを適時拡張し、後述するプログラム変換をカスタマイズすることでこのような要求を満たすことができるのも、このシステムの特徴でもある。

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	共有オブジェクトに対して、排他的なアクセス権を得る
release	共有オブジェクトに対する排他的なアクセス権を解放する
update	共有オブジェクトの最新の状態を得る

表 3.4: 抽象化クラス `SharedObjectPool` で宣言されているメソッド

今回はメモリ管理モデルとして Write Invalidate モデルを採用したが、そのほかのメモリ管理モデル (Write Update、 Lazy Release Consistency) を利用することも容易である。

処理の流れ

SharedObjectPool では、分散共有メモリのそれぞれの操作は以下のように行われることを想定している。

初期化 初期化は initialize メソッドを呼び出すことによって行われる。この中では共有オブジェクトの管理テーブルの初期化などが行われることを想定している。通信インタフェースは setComm メソッドで設定され、以後システム内ではこの通信インタフェースを利用する。

共有オブジェクト登録 分散共有するオブジェクトは利用する前に登録する必要がある。この登録は register メソッドを用いて行う。引数としては、分散共有するオブジェクトを与える。

read アクセス read アクセスでは、アクセスを行う前にアクセス対象のオブジェクトを引数として update メソッドを呼び出す。これによって最新のオブジェクトの情報を取得してアクセスを行うことができる。

write アクセス write アクセスでは、アクセスを行う前にアクセス対象のオブジェクトを引数として acquire メソッドを呼び出す。これによって最新のオブジェクトの状態を取得し、排他的なアクセスが可能となる (Multiple Writer プロトコルの場合は必ずしも排他的とは限らない)。

また、アクセス終了後 release メソッドを呼び出すことによって排他的なアクセスを解放する。

配列の扱い 1 つの配列オブジェクトを 1 つのオブジェクトとして管理すると効率が悪い場合が多いため、配列オブジェクトを分割して扱えるようにする。そこで update、 release、 acquire などのメソッドにおいて引数として、配列オブジェクト以外に範囲を与えてやることによって、

```
class SharedObject{
    Object obj;
    int key;
    int owner;
    boolean valid;
    boolean locked;
    CopyList copy;
}
```

図 3.2: クラス SharedObject のフィールド

配列を分割して管理できるようにする。これによって配列オブジェクトが原因となる False Sharing を抑えることができる。

メソッド呼び出し 分散共有されているオブジェクトのメソッド呼び出しは、そのメソッドを呼び出したノード上で実行される。つまり、 RMI(Remote Method Invorcation) のような機構はない。メソッドを呼び出すときにも、 acquire メソッドを呼び出して、呼び出されるメソッドのあるオブジェクトの状態を最新の情報にするとともに、排他的なアクセスを行うようにする必要がある。メソッド呼び出し後には release メソッドで排他的なアクセスを解放する必要がある。

3.2.1 Write Invalidate モデル

本システムでは、 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。以下では、 Write Invalidate を用いた分散共有メモリを抽象化クラス 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

initialize メソッドでは、共有オブジェクトの管理テーブルの初期化などを行う。また、通信インタフェース Comm より、実行環境情報 (ノード番号など) を取得する。また、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する。

register メソッド

register メソッドでは、分散共有するオブジェクトを引数としてとり、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 クラス (図 3.2) で wrapping して登録する。この WriteInvalidate クラスの実装では

SPMD スタイルのプログラムを対象としているため、分散共有されるオブジェクトは全てのノードにおいて同じ順序で登録される。この順序をキーとして用いる。キーとしてはオブジェクトのハッシュコードを用いる方法も考えられるが、 Java 言語においては、異なる JVM プロセスにおいてオブジェクトのハッシュキーの同一性を保証していないため、この方法は利用できない。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

update メソッドでは、最新のオブジェクトの状態にしたい、分散共有されているオブジェクトを引数として呼び出される。 update メソッドでは、引数として渡されたオブジェクトを検索して、そのオブジェクトの状態を確認する。万一そのオブジェクトが登録されていない場合は実行時エラーが発生する。オブジェクトの状態が有効であった場合、何も行われずに実行もとへ返る。もし、オブジェクトの状態が無効であった場合は、そのオブジェクトの owner ノードへ最新のオブジェクトの状態を要求する。最新のオブジェクトを得られたら、もとの実行へ戻る。また、他のノードが排他的なアクセスを行っているなどで、オブジェクトがロックされている場合は、そのロックが解除されるまでブロックする。

acquire メソッド

acquire メソッドでは、排他的なアクセスをしたい、分散共有されているオブジェクトを引数として呼び出される。 acquire メソッドでは、オブジェクトの内容が更新されることを前提としているため、 invalidation が発行される。 acquire メソッドでは、引数として渡されたオブジェクトを検索して、そのオブジェクトの状態を確認する。万一そのオブジェクトが登録されていない場合は実行時エラーが発生する。オブジェクトの owner が自分のノードであった場合、コピーを持つノードに対して invalidation を発行し、このオブジェクトをロックして、実行を継続する。オブジェクトの owner が自分のノードでないとき、 owner のノードに対して所有権を要求する。 owner のノードではこのオブジェクトをロックして、所有権が移動することを全てのノードに知らせ、所有権を与え、ロックを解除する。このとき、オブジェクトの状態が無効であった場合は、最新のオブジェクトの状態も得る。そして、このオブジェクトをロックして実行を継続する。

release メソッド

release メソッドでは、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る。 release メソッドでは引数として渡されたオブジェクトを検索して、そのオブジェクトの状態を確認する。万一そのオブジェクトが登録されていない場合は実行時エラーが発生する。オブジェクトがロックされていない場合や owner でない場合も実行時エラーが発生する。 release メソッドでは、そのオブジェクトに対する排他的なロックを解除して実行を継続する。

finalize メソッド

finalize メソッドでは、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処理

他のノードからのリクエストは、 initialize メソッドで生成されたスレッドで行われる。この他のノードからのリクエストには、自分のノードの出したリクエストへの応答なども含まれる。リクエストおよびその処理は以下の通りである。

オブジェクトコピー要求 リモートノードにおいて update、 acquire メソッドが呼び出され、オブジェクトの最新の状態が要求された。要求を受けたノードでは、対象となるオブジェクトの状態をチェックして、そのオブジェクトを送る。

オブジェクトコピー 自分のノードにおいて update、 acquire メソッドが呼び出され、リモートノードからオブジェクトの最新の状態を受け取った。自分のノードの対象となるオブジェクトにその最新の状態を反映させ、オブジェクトの状態を有効にする。 update、 acquire メソッドでは、オブジェクトの状態が有効になるまでブロックしている。

所有権要求 リモートノードにおいて acquire メソッドが呼び出され、その結果、オブジェクトの所有権が要求された。要求を受けたノードでは、自分のノードの対象となるオブジェクトをロックして、コピーを持つノードに invalidation を発行し、コピーノードリストをクリアする。次に、全てのノードにそのオブジェクトにおける所有権が移動したことを通知する。そして、要求したノードへ所有権を譲る。

所有権譲渡 自分のノードにおいて acquire メソッドが呼び出され、リモートノード (オブジェクトの所有者ノード) からそのオブジェクトにおける所有権を受け取った。自分のノードの対象となるオブジェクトの所有者を自分にセットする。 acquire メソッドでは、オブジェクトの所有者ノードが自分のノード番号になるまでブロックしている。

所有権移動 他のノードにおいて acquire メソッドが呼び出され、オブジェクトの所有権が移動した。自分のノードの対象となるオブジェクトをロックし、所有権を持つノードを変更して、ロックを解除する。

Invalidation 他のノードにおいて acquire メソッドが呼び出され、自分がコピーを持つオブジェクトが無効となった。自分のノードの対象となるオブジェクトを破棄して、状態を無効とする。

オブジェクトのロック 他のノードにおける処理によって、自分のノードの対象となるオブジェクトへのロックが要求された。対象となる自分のオブジェクトをロックする。

オブジェクトのロック解除 他のノードにおける処理が終了し、自分のノードの対象となるオブジェクトへのロックが解除された。対象となる自分のオブジェクトをロックを解除する。

```
public class Test{
  AObject[] a;
  int numThreads;
  public Test(int numThreads){
    this.numThreads = numThreads;
    a = new AObject[numThreads];
  }
  public void calc(){
    Thread[] thr = new TestThread[numThreads];
    for(int i = 0 ; i < numThreads ; i++){
      thr[i] = new TestThread(a);
      thr[i].start();
    }
    for(int i = 0 ; i < numThreads ; i++)
      thr[i].join();
  }
  public static void main(String[] args){
    Test test = new Test(Integer.parseInt(args[0]));
    test.calc();
  }
}
```

図 3.3: 変換前 (1)

3.3 プログラム解析と変換

本実装では、 OpenJIT をもちいたバイトコードレベルのプログラム解析・変換を行う。本実装においては、スレッドの移送などは実現されないため、全てのスレッドの利用モデルをサポートするのは困難である。したがって、ある程度対象を制限する必要がある。具体的には SPMD スタイルのプログラムであり、プログラムは初期化 - スレッド生成 - 並列実行 - 終了プロセスのスタイルをとるものとする。

3.3.1 プログラム解析

本実装では、プログラム解析を行いながらプログラム変換を行う必要のある箇所を検出する。検出されるとその場でプログラム変換部が呼び出される。以下では、各解析部分の動作について説明する。

初期化部分

```
import jdsm.dsm.*;
public class Test{
  static SharedObjectPool pool = Main.pool;   //(1)
  static int numNode = Main.comm.getNumNode();//(2)
  static int myNode = Main.comm.getMyNode();  //(3)
  AObject[] a;
  int numThreads;
  public Test(int numThreads){
    this.numThreads = numThreads;
  }
  public void init(){
    a = new AObject[numThreads];
    pool.register(a);                         //(4)
  }
  public void calc(){
    Thread[] thr = new TestThread[numThreads];
    for(int i = 0 ; i < numThreads ; i++){
      if (i != myNode) continue;              //(5)
      thr[i] = new TestThread(myNode, a);
      thr[i].start();
    }
    for(int i = 0 ; i < numThreads ; i++){
      if(i != myNode) continue;
      thr[i].join();                          //(5)
    }
    pool.barrier();                           //(6)
  }
  public static void main(String[] args){ ... }
}
```

図 3.4: 変換後 (1)

対象となるプログラムを走査し、フィールド情報から分散共有すべきオブジェクトおよびそのクラス情報等を取得する。得られた情報は、 read アクセス解析、 write アクセス解析などで分散共有オブジェクトであるかどうかの判定などに利用される。

read アクセス プログラム中の read アクセスについて、分散共有オブジェクトが対象となった read アクセスかどうかを調べ、分散共有オブジェクトであれば、 read アクセスのプログラム変換メソッドを呼び出す。

write アクセス プログラム中の write アクセスについて、分散共有オブジェクトが対象となった write アクセスかどうかを調べ、分散共有オブジェクトであれば、 write アクセスのプログラム変換メソッドを呼び出す。

メソッド呼び出し プログラム中のあるオブジェクトのメソッド呼び出しについて、分散共有オブジェクトが対象となったオブジェクトのメソッド呼び出しかどうかを調べ、分散共有オブジェクトであれば、メソッド呼び出しのプログラム変換メソッドを呼び出す。

3.3.2 プログラム変換

本実装で必ず行われる変換 (初期化部分等) に加え、プログラム解析を行った結果などを利用してプログラム変換を行う。プログラム変換の行われるべき箇所は次のようになっている。

初期化部分

- プログラムは jdsm.dsm.Main クラスより実行される。この jdsm.dsm.Main からプログラムの public static void main(String[] args) を呼び出す
- フィールドに “SharedObjectPool pool”、 “numNode”、 “myNode” を追加する

実行部分

- オブジェクトを new したあとに pool.register(Object) を挿入
- スレッドの生成、実行、 join をフック、自分のノード番号のみを行うようにする
- スレッドの join の後には同期をとる

read アクセス 共有オブジェクトへ read アクセスをしている部分に pool.update(Object) 挿入

write アクセス 共有オブジェクトへ write アクセスをしている前に pool.acquire(Object)、後に pool.re lease(Object) を挿入

メソッド呼び出し 共有オブジェクトのメソッドには write アクセス同様 acquire および release を挿入する

```
public void run(){
  AObject x;
  x = a;                                  //(1)
  b[i] = new BObject();                   //(2)
  a.method1();                            //(3)
}
```

図 3.5: 変換前 (2)

図 3.3 , 図 3.4 は初期化部分などの変換前、変換後のプログラムの様子を示したものである。まず、 (1) においてメモリ管理クラスの宣言を与える。 SharedObjectPool は抽象化クラスであるため、インスタンスとしては実装されたクラスが格納されている。本システムの場合、分散共有メモリの項で挙げた Write-Invalidate プロトコルを用いた WriteInvalidate クラスのインスタンスが格納されている。 (2)、 (3) においては、通信インターフェースから実行ノード数と自分のノード番号を得ている。 (4) では宣言されているフィールドである “a” を共有オブジェクトとしてメモリ管理クラスに登録している。この register メソッドはフィールドのオブジェクトに対して new が行われた直後に挿入される。 (6) ではスレッドの実行、 join を自分のノード番号についてのみ行うような変更が施されている。この変更は現在では、実装の都合上、 OpenJIT による変換ではなく手動で行われている。 (7) ではスレッドの join に代わって、バリア同期をとるように変更されている。

図 3.5 図 3.6 は並列実行部分の変換前、変換後のプログラムの様子を示したものである。 “a”、 “b” は分散共有されているオブジェクトとする。まず、 (1) において、 a は共有オブジェクトであるため、 x への代入は read アクセスとして扱われ、 x への代入の前に update メソッドが呼び出される (Java では Object は参照渡しであるため、代入された x についても a と同様の扱いをする必要が考えられる)。 (2) においては共有されている配列オブジェクト b への代入であるため、 write アクセスとして扱われる。この場合、 b が配列オブジェクトであり、その添字 i への代入であるため、 acquire メソッドは b と添字の番号である i を引数として呼び出され、配列オブジェクト b 全てではなく、 b[i] に対してのみ排他的アクセスが行われる。アクセス後、 release メソッドを呼び出すことによって排他的なアクセスを終了する。 (3) では、共有オブジェクト a に対するメソッド呼び出しである。全てのメソッド呼び出しにおいて、 write アクセスが発生するとは限らないが、 write アクセスであると判断してその前後に acquire、 release メソッドを挿入する。 cJVM のようにメソッドシッピングの機構を提供しているようなシステムの場合は、 Master ノードでメソッドが実行されるため、 write アクセスとみなす必要はなく、前後にチェックメソッドの挿入は不要である。

```
public void run(){
  AObject x;
  a = pool.update(a);                    //(1)
  x = a;
  b[i] = pool.acquire(b, i);             //(2)
  b[i] = new BObject();
  pool.release(b, i);
  a = pool.acquire(a);                   //(3)
  a.method1();
  pool.release(a);
}
```

図 3.6: 変換後 (2)

評価 4 ≡

4.1 評価環境

評価環境として、松岡研究室で開発されている Presto Cluster、 PentiumII 350MHz32 台構成を用いた。各ノードのメモリは 128MB で 2 系統の 100Mbps の FastEthernet で Switch 接続されている。このシステムは、 OS が Linux 2.2.14、 VIA の実装である M-VIA 1.0 を用いている。なお、 M-VIA は 2 系統あるネットワークのうち 1 系統のみで利用可能となっている。コンパイラは gcc 2.95.2 を用い、最適化のオプションは “-O6 - fomit-frame-pointer ” とした。

4.2 性能評価

4.2.1 JVIA 基本性能

VIA を Java から利用する JVIA の基本性能として JVIA のレイテンシとバンド幅を測定した (表 4.1)。これを C 言語での native な VIA と比較すると (表 4.2)、サイズが小さいときにレイテンシが JVIA の方が非常に大きく、サイズが大きくなるにしたがって、差が小さくなっているのが分かる。また、バンド幅でみると、同じようにサイズが小さいときには JVIA の方が小さいが、サイズが大きくなるにしたがって、 native の VIA のバンド幅近く出ているのが分かる。

レイテンシがこのように大きい原因は、Java 側で持っているデータ構造である VIP_DESCRIPTOR の native へのマーシャリングによるものである。この VIP_DESCRIPTOR は内部に VIP_DESCRIPTOR_SEGMENT、さらに VIP_CONTROL_SEGMENT や VIP_ADDRESS_SEGMENT、 VIP_DATA_SEGMENT さらに VIP_PVOID64 など、データ構造が深い。このため、データのマーシャリングに多くの時間が化かってしまっているためである。また、 JVIA の実装では、バッファデータを Java から native のバッファへのコピーが 1 回行われてしまうため、これもオーバーヘッドの要因でもある。

この VIP_DESCRIPTOR のマーシャリングがオーバーヘッドとなっていることを確認するた

サイズ	レイテンシ (usec)	バンド幅 (MB)
0	125.5	0.0
1	124.5	0.0080
4	125.0	0.032
16	131.0	0.122
64	133.0	0.481
256	168.5	1.519
1024	301.0	3.402
4096	625.0	6.554
16384	1825.0	8.97
32000	3310.0	9.66

表 4.1: JVIA のレイテンシとバンド幅

サイズ	レイテンシ (usec)	バンド幅 (MB)
0	30.238	0.000
1	31.279	0.031
4	31.253	0.127
16	31.385	0.509
64	38.636	1.656
256	72.491	3.531
1024	204.680	5.002
4096	490.276	8.354
16384	1526.766	10.731
32000	2835.135	11.286

表 4.2: VIA のレイテンシとバンド幅

サイズ	レイテンシ (usec)	バンド幅 (MB)
0	102.0	0.0
1	106.5	0.0094
4	106.0	0.0377
16	106.5	0.150
64	114.0	0.561
256	155.0	1.651
1024	280.5	3.650
4096	601.0	6.815
16384	1760.5	9.306
32000	3226.0	9.919

表 4.3: JVIA のレイテンシとバンド幅 (一部マーシャリングなし)

プロセッサ数	実行時間 (sec)
1	40.5
2	61.1

表 4.4: SPLASH2 LU Kernel

め、 VIP_DESCRIPTOR のマーシャリングを一部 (25%) 行わないでレイテンシとバンド幅を計測した (表 4.3)。

これから、サイズが小さい場合、レイテンシが大幅に向上していることが分かる。つまり VIP_DESCRIPTOR のマーシャリングを極力行わないことが有効であることが判明したが、VIP_DESCRIPTOR のマーシャリングを行わないと正常な通信が行えないため、通信を行う場合に必要なデータ、更新されるデータのみにしぼってマーシャリングを行う必要がある。しかし、サイズが大きい場合、あまり性能が改善されていない (図 4.1 , 図 4.2)。これは、サイズが大きい場合、通信用のバッファを native から Java 側へのコピーのオーバーヘッドが大きいためと考えられる。そこで、サイズの大小によらず、 Java 側で扱うデータ構造を改良し、データ領域を pinning して、native 側より直接扱うような改良を行うことも必要である。

4.2.2 SPLASH2 LU

次に、 SPLASH2 から LU Kernel を Java 言語へ移植し、その性能を評価した。

LU Kernel を分散共有メモリ機能なし (1 ノード) で動作させた場合、 PC クラスタでは 36.1 秒である。まず、行列サイズ 512 の場合 (図 4.4)、 4 ノード以上では正常な性能が得られなかった。こ

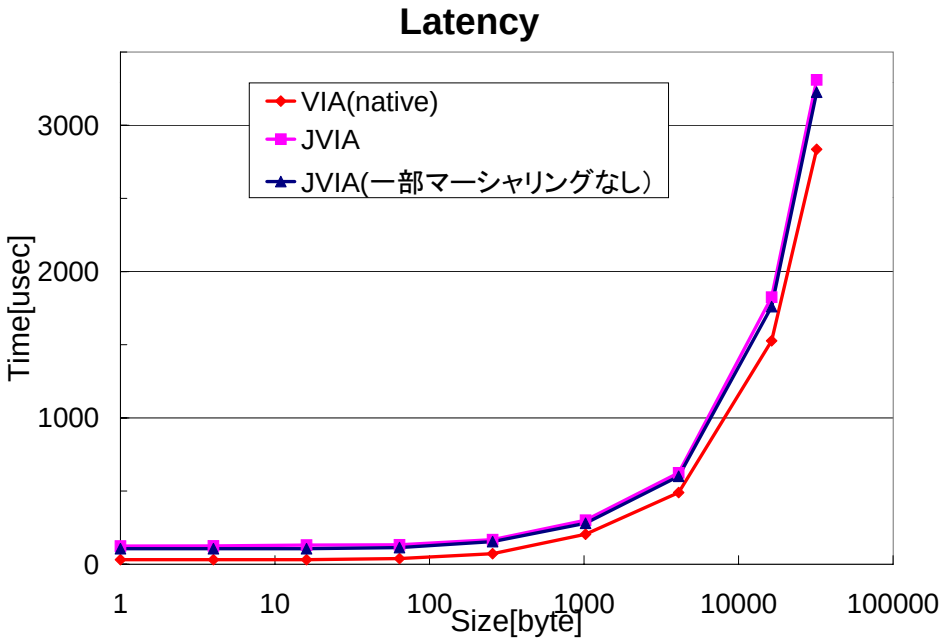

図 4.1: JVIA, VIA のレイテンシ

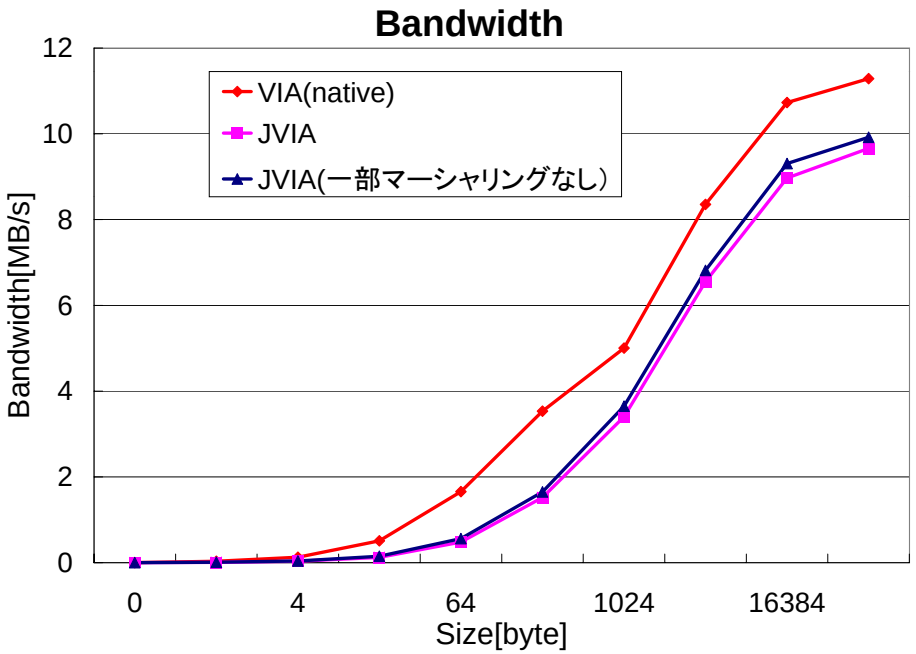

図 4.2: JVIA, VIA のバンド幅

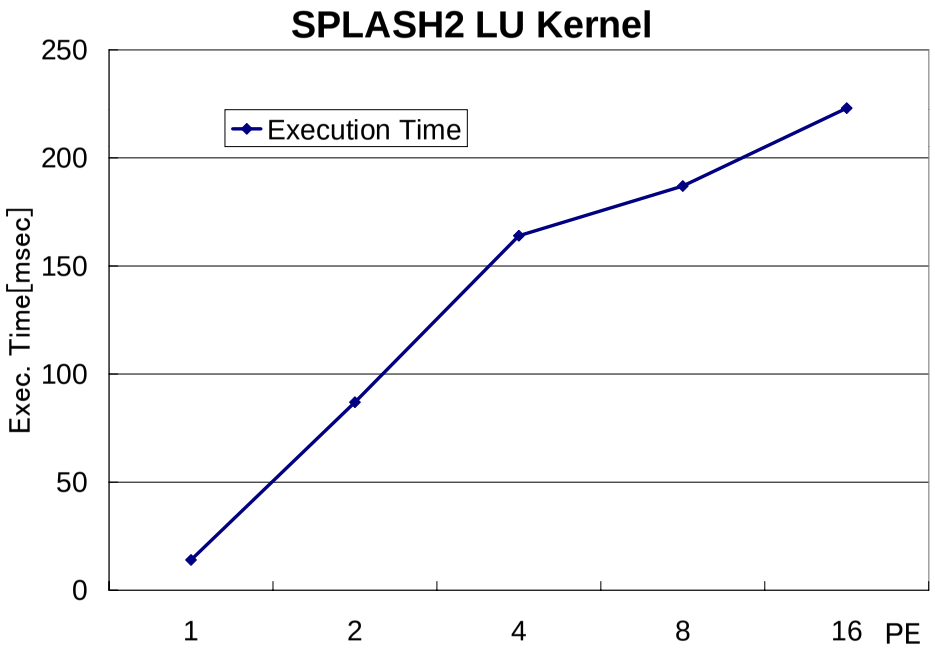

図 4.3: SPLASH2 LU Kernel (Cluster)

の原因としては、 JDSM システムは配列オブジェクトの配列要素ごとの共有機能を提供しているが、これは 1 次元配列に限ったものである。 LU Kernel のように、 2 次元配列を扱う場合、 1 次元目は各要素ごとに共有することが可能であるが、 2 次元目以降は各要素ごとには管理できない。このため、メモリの管理単位が 8byte * 512 = 4KB 以上と管理単位が大きくなってしまったため、False Sharing が多発してしまったと考えられる。これに対しては、管理可能な配列の次元数を増やすことが考えられるが、全ての次元数に対応することは難しい。そこで、現実的に 3 次元程度までは効率よく管理できるようにすべきであることが分かった。また、通信インタフェース JVIA において、データのマーシャリングのオーバーヘッドが指摘されている。これも分散共有メモリ機能の性能低下の要因であるとも考えられる。なお、 2 ノードでは 1 ノードより若干性能が低下しているのが分かる。これは並列化による性能向上が分散共有メモリのオーバーヘッドによって相殺されているためである。

また、行列サイズが小さい場合、並列化の効果が得られないが、参考までに行列サイズを 32 まで小さくした場合の結果を図 4.3 にしめす。グラフから分かるように台数が増えるにしたがって性能が低下しているのが分かる。これは行列サイズが小さいため、並列化の効果よりも、並列化のオーバーヘッドが大きいためである。

結論 5 ≡

5.1 まとめ

本研究では、自己反映計算 (リフレクション) を用いてソフトウェアのプラットフォームポータビリティを実現する手法を提案した。そして、自己反映計算とソフトウェア分散共有メモリを用いて、並列実行環境上でプラットフォームポータビリティを実現する並列プログラミング環境 JDSM を実現した。

JDSM システムでは、プログラム中に明示的に記述する必要のあった並列実行環境向けの機能拡張のコード — ソフトウェア分散共有メモリ機能や分散メモリ環境のサポート — をメタコードとして分離し、 Open Compiler によって機能拡張させることで、プラットフォームの違いをメタコードで吸収させる。これによって、従来の SPMD スタイルで記述されたマルチスレッド並列プログラムを、ソースレベルのプログラム変換を行い、クラスタ型並列計算機上で動作することを可能とさせた。

この JDSM システムでは、自己反映計算を用いてプラットフォームポータビリティを実現できることを完全ではないが，示した。

5.2 今後の課題

今後の課題は、第一に、より完成度の高い JDSM システムの構築である。現状の JDSM は SPLASH2 などのベンチマークを移植して十分な評価を行うほどの十分な安定性を有していない。

M-VIA を用いた通信インタフェースの実装では、 Java 言語における native 呼び出し特有のオーバーヘッドが発生した。これは Java と native 間のオブジェクト・データ構造のマーシャリングのオーバーヘッドのことだが、このオーバーヘッドが大きかったため、十分な性能を発揮することができなかった。

改良方法としては、マーシャリングを極力行わない、もしくは、全く行わないような実装方法を

行うことが考えられる。具体的には、2つの方法が考えられる。1つは、通信インターフェースの実現に必要なデータ領域を全てJava側で確保し、native側では使用時にその領域をpinningする方法である。もう1つは逆にデータ領域を全てnativeで保持し、Java側からの更新はJNIなどを用いて行う方法である。どちらの方法を採るかは、性能や実現の容易さなどから考えていく必要がある。

また、この通信インタフェース部分は、特定の通信レイヤーに依存せずに作られている。しかしながら、本研究ではM-VIAに対する実装のみを行っており、他の多くの通信レイヤーへの適応性が確認されていない。Java/PM[岡崎 99]に代表される、より高速な通信レイヤーの利用や、それらとMVIAによる実装との比較も今後の課題である。

また、メモリ一貫性モデルとしてどのモデルを採用するかについても、柔軟性を持たせた設計となっているが、本研究の実装では一番単純なstrictな一貫性モデルを採用している。

メモリの一貫性のプロトコルには、このほかにWrite Updateプロトコル、Automatic Updateプロトコル、Home-baseなUpdateプロトコルや、コンシステンシモデルとしてはLazy Release Consystencyなどが提案されている。しかし、Java言語においてはソフトウェア分散メモリの有効な実現方法 — コンシステンシプロトコルやモデル — について十分な研究がなされていない。つまり、従来のC言語などで構築されたシステムにおいて有効とされたシステムが、Java言語においても有効であるかが不明であるため、今後、より多くのプロトコルやモデルのソフトウェア分散共有メモリをJava上に実現して、定性的な評価を行う必要がある。また、性能評価の結果から配列をより柔軟に管理できるようなモデルを実現し、False Sharingなどのオーバーヘッドを除去し、大幅な性能向上を図る必要がある。

参考文献

[ACD+96] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Warkstations. *IEEE Computer*, Vol. 29, No. 2, pp. 18–28, Feb. 1996.

[AFT99] Yariv Aridor, Michael Factor, and Avi Teperman. cJVM: a Cluster Aware JVM. In *Proceedings of International Conference on Parallel Processing '99*, pp. 31–39, Jun. 1999.

[BGC98] Philip Buonadonna, Andrew Geweke, and David Culler. An Implementation and Analysis of the Virtual Interface Architecture. In *Proceedings of SC'98*, Nov. 1998.

[Chi95] Shigeru Chiba. A Metaobject Protocol for C++. In *Proceedings of OOPSLA'95*, pp. 285–299, 1995.

[Chi97] Shigeru Chiba. *OpenC++ 2.5 Reference Mannual*, 1997.

[CPL+97] Andrew A. Chien, Scott Pakin, Mario Lauria, Matt Buchanan, Kay Hane, Louis Giannini, and Jane Prusakova. High Performance Virtual Machines (HPVM): Clusters with Supercomputing APIs and Performance. In *Proceedings of Eighth SIAM Conference on Parallel Processing for Scientific Computing (PP97)*, 1997.

[GC98] Louis A. Giannini and Andrew A. Chien. A Software Architecture for Global Address Space Communication on Clusters: Put/Get on Fast Messages. In *Proceedings of the 7th High Performance Distributed Computing (HPDC7) conference*, July 1998.

[HTI+96] Atsushi HORI, Hiroshi TEZUKA, Yutaka ISHIKAWA, Noriyuki SODA, Hiroshi HARADA, Atsushi FURUTA, Tsutomu YAMADA, and Yasuhiro OKA. Parallel programming environment on workstation cluster. In *IPSJ SIG Notes*, 96-OS-73, pp. 121–126. Information Processing Society of Japan, Aug. 1996.

[htt97] http://www.viarch.org/. Virtual Interface Architecture Specification, Dec. 1997.

[Ie96] Yutaka Ishikawa and et.al. Design and Implementation of Metalevel Architecture in C++ – MPC++ Approach –. In *Proceedings of Reflection'96*, Apr. 1996.

[IR97] Yuuji Ichisugi and Yves Roudier. Extensible Java Preprocessor Kit and Tiny Data-Parallel Java. In *Proceedings of ISCOPE '97*, December 1997.

[Ish96] Yutaka Ishikawa. Multiple Threads Template Library -MPC++ Version2.0 Level 0 Document -. TR 96012, RWCP, 1996.

[KdRB91] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[KH98a] Ralph Keller and Urs Holzle. Binary Component Adaptation. In *Proceedings of ECOOP'98*, Jul. 1998.

[KH98b] Ralph Keller and Urs Holzle. Implementing Binary Component Adaptation for Java. Technical Report TRCS98-21, Dept. of Computer Science, University of California, Santa Barbara, Sep. 1998.

[KLRR92] Gregor Kiczales, John Lamping, Luis Rodriguez, and Erik Ruf. An Architecture for an Open Compiler. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*. Xerox PARC, 1992.

[LC97] Lauria and Andrew A. Chien. MPI-FM: High Performance MPI on Workstation Clusters. *Journal of Parallel and Distributed Computing*, Vol. 40, No. 1, pp. 4–18, Jan 1997.

[MOS+98] S. Matsuoka, H. Ogawa, K. Shimura, Y. Kimura, K. Hotta, and H. Takagi. OpenJIT —A Reflective Java JIT Compiler. In *Proceedings of OOPSLA '98 Workshop on Reflective Programming in C++ and Java*, pp. 16–20, Dec. 1998.

[OSM+00] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura. OpenJIT: An Open-Ended, Reflective JIT Compile Framework for Java. To appear in ECOOP'2000, Jul. 2000.

[OW98] D. Oppenheimer and M. Welsh. User Customization of Virtual Network Interfaces with U-Net/SLE. Technical Report CSD-98-995, UC Berkeley, Feb. 1998.

[PKC97] Scott Pakin, Karamcheti, and Andrew A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, Vol. 5, No. 2, pp. 60–73, Apr.-Jun. 1997.

[SGT96] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of ASPLOS VII*, Oct. 1996.

[SPWC98] Patrick G. Sobalvarro, Scott Pakin, William E. Weihl, and Andrew A. Chien. Dynamic Coscheduling on Workstation Clusters. In *Proceedings of the International Parallel Processing Symposium (IPPS '98)*, Mar. 1998.

[TCN97] Michiaki Tatsubori, Shigeru Chiba, and Ikuo Nakata. OpenJava: Yet another reflection support for Java. In *unknown*, 1997.

[THI96] Hiroshi Tezuka, Atsushi Hori, and Yutaka Ishikawa. Design and Implementation of PM: A Communication Libaray for Workstation Cluster. In *Proceedings of JSPP'96*, pp. 41–48. Information Processing Society of Japan, Jun. 1996.

[THIS97] Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa, and Mitsuhisa Sato. PM: An Operating System Coordinated High Performance Communication Library. In Peter Sloot and Bob Hertzberger, editors, *High-Performance Computing and Networking '97*, Vol. 1225, pp. 708–717. Lecture Notes in Computer Science, April 1997.

[WC99] Matt Welsh and David Culler. Jaguar: Enabling Efficient Communication and I/O from Java. In *Submitted for publication*, Aug. 1999.

[YC97] Weimin Yu and Alan Cox. Java/DSM: a Platform for Heterogeneous Computing. In *ACM 1997 Workshop on Java for Science and Engineering Computation*, Vol. 43.2, pp. 65–78, Jun. 1997.

[岡崎 99] 岡崎史裕, 松田元彦, 入口浩一. JavaPM/Myrinet と SORB の性能評価. 情報処理学会ハイパフォーマンスコンピューティング研究会, Dec. 1999.

[小川 99] 小川宏高, 松岡聡, 丸山冬彦, 早田恭彦, 志村浩也. OpenJIT フロントエンドシステムの設計. SWoPP 下関'99, 1999.

[松岡 98] 松岡聡, 小川宏高, 志村浩也, 木村康則, 堀田耕一郎, 高木浩光. OpenJIT— 自己反映的な Java JIT コンパイラ. In *SWOPP'98*, Jul. 1998.