電子情報通信学会研究報告 CPSY-95-59, Vol. 95, No. 210, pp. 71–78, 平成 7 年 8 月

分散共有メモリ型並列計算機 KSR 上の細粒度並列処理用実行方式の評価

八杉 昌宏　　瀧 和男

神戸大学工学部情報知能工学科
〒 657 神戸市灘区六甲台町 1-1
Tel: 078-803-1189, Fax: 078-803-1218
E-mail: {yasugi,taki}@seg.kobe-u.ac.jp

あらまし　本研究では、比較的細粒度な並列オブジェクト指向言語 ABCL/ST の、分散共有メモリ型並列計算機 KSR 上への実装を行った。本論文では、その実装を通して、分散共有メモリ型並列計算機上での効率良い細粒度並列処理用実行方式について検討評価したので報告する。特に、プロセッサ間通信においてロックを用いない実行方式を提案し、その評価を行う。本論文では、共有メモリ型計算機上のメッセージパッシングライブラリについて述べるわけではなく、メッセージパッシングとスケジューリングとをセマンティクスとして暗に含むプログラミング言語の実装について述べる。

キーワード　分散共有メモリ, 細粒度並列処理, 並列オブジェクト指向, プロセッサ間通信, ロック

Evaluation of Implementation Schemes for Fine-Grained Concurrent Computing on the Distributed-Shared Memory Computer KSR

Masahiro YASUGI　　Kazuo TAKI

Department of Computer and Systems Engineering
Faculty of Engineering, Kobe University
1-1 Rokkodai Nada Kobe, JAPAN 657
Tel: +81-78-803-1189, Fax: +81-78-803-1218
E-mail: {yasugi,taki}@seg.kobe-u.ac.jp

Abstract　The evaluation of implementation schemes for fine-grained concurrent computing on a distributed-shared memory computer is presented in the contexts of implementation of a concurrent object-oriented programming language ABCL/ST on a distributed-shared memory computer KSR. In particular, we propose and evaluate a lock-less implementation scheme for inter-processor communication. It should be noticed that this paper does not describe a message passing library on a shared-memory system; indeed, this paper shows implementation schemes of a programming language which implicitly includes message passing and scheduling as a part of its semantics.

key words　distributed-shared memory, fine-grained computing, concurrent object-oriented, inter-processor communication, mutex lock.

1　はじめに

分散共有メモリ型並列計算機において、並列オブジェクト指向言語のような細粒度並列処理を必要とする言語を実装するにあたり、その並列実行方式について検討評価したので報告する。

本研究では、比較的細粒度な並列オブジェクト指向言語 ABCL/ST の、分散共有メモリ型並列計算機 KSR 上への実装を行った。本論文では、その実装を通して検討した、共有メモリ型並列計算機上での効率良い細粒度並列処理用実行方式について述べる。特にプロセッサ間通信においてロックを用いない実行方式を提案し、その評価を行う。

本論文では、共有メモリ型計算機上のメッセージパッシングライブラリについて述べるわけではない。むしろ、メッセージパッシングとスケジューリングとを暗にセマンティクスとして含むプログラミング言語の実装について述べる。つまり、ここでは、並列処理の抽象化はライブラリではなく言語によって達成されている。このような言語/コンパイラにより抽象化された計算を実際に行う実体を、本論文ではランタイムアーキテクチャと呼ぶことにする。

2　背景

本節では、並列オブジェクト指向言語 ABCL/ST と、その実装の対象マシンとした分散共有メモリ型並列計算機 KSR についてまとめる。

2.1　並列オブジェクト指向言語 ABCL/ST

並列オブジェクト指向言語 ABCL/ST[6] は、ABCL/1[8] に部分型を含む静的な型付けなどを導入した言語である。

ABCL/1 の計算/プログラミングモデルでは、並列の単位となるのは、独自のメモリと計算能力を持つ多数のオブジェクトであり、これらはメッセージパッシングのみを通して互いに協調しながら並列に計算を行う。また、オブジェクトは動的に生成することができる。

ABCL では、オブジェクトは状態変数とメソッドを持つ。オブジェクトは休眠、待機、活性のいずれかのモードにあり、休眠状態でメソッド実行可能な要求メッセージを受け取ると、活性モードとなってメソッドを実行し、メソッド実行の完了とともに休眠モードに戻る。また、メソッド実行中に別のメッセージが必要な場合は、そのメッセージを受け取って実行を再開するまでの間、待機モードになる。

ABCL/ST は、データ駆動型並列計算機 EM-4[2] を対象マシンとする処理系 ABCL/EM-4[7, 9] のプログラミング言語として設計され、コンパイラ等が実装されてきた [6]。

2.2　分散共有メモリ型並列計算機 KSR1 のアーキテクチャ

KSR シリーズは、物理的に分散配置されたメモリ (キャッシュ) に ALLCACHE エンジンというハードウェア機構を用いて、共有メモリのイメージをハードウェアレベルで提供する ALLCACHE™アーキテクチャを持つ。メインメモリはなくローカルキャッシュのみでマシン全体のメモリが構成されている。

KSR1 のアーキテクチャは、単方向リングの階層的ネットワークになっており、最下層の Ring:0(バンド幅 1GB/s) には最大 32 個の要素プロセッサ (PE) が接続可能で、さらにその上の層の Ring:1 には最大 34 個の Ring:0(=最大 1088PE) が接続可能である。

KSR1 は MIMD 型コンピュータであり、各 PE は、64bit の整数/浮動小数点ユニット (40MIPS, 40MFlops) と、0.5MB のサブキャッシュ、32MB のローカルキャッシュからなる。ローカルキャッシュ間の一貫性管理は、1subpage=128B を単位とするコピー, 無効化などにより行われる。また、今回の実装では利用しなかったが、このサブページ単位でのプリフェッチやポストストア (無効なコピーの更新) が指定できる。

KSR では、OSF/1 ベースの KSR OS が稼働しており、UNIX コンパチのマルチユーザ・マルチタスク環境が利用できる。並列プログラミングには POSIX に基づくスレッドライブラリが利用できる。また本研究では、マンチェスター大学にある 40PE の KSR1 を使用した。

3 実装の基本部分

3.1 C コード生成機能

先に述べたように、ABCL/ST は、元々、データ駆動型並列計算機 EM-4 を対象マシンとしてコンパイラ等が実装されてきた。

EM-4 上のコンパイラで生成されるコードはアセンブリ言語であったが、KSR への移植に際して、コンパイラの生成コードを C 言語とした。以下そこで用いた C コードの生成技法について述べる。(ここでは、しばらくの間、単一プロセッサの場合と考えてもらって良い)。

EM-4 上のコンパイラでは、各オブジェクトのスレッドを高速に切替えるため、EM-4 の持つデータ (パケット) 駆動の機能を利用していた。つまり、パケットキューをスケジューリングキューとして用い、高速なスレッド切替えを実現していた。通常の C 言語には、このようなデータ駆動/スケジューリングの機能はないので、まずスケジューリングの機能の追加した抽象 EM-4 とでもいうべき抽象機械を設計し、これ上へコードを生成するようにした。(このスケジューリングメカニズムは言うまでもまくソフトウェアで実現しなくてはならない)。具体的には、パケットキューは巡回バッファとし、パケットサイズは EM-4 と同様、固定サイズとして実現している。パケットサイズを固定にする利点は、スケジューラが簡単高速になることにある。また、パケットが不連続になったりせず、巡回バッファの終端の判定が簡単になるという利点もある。

C コード生成時の問題として、(a)C 言語ではハードウェアレジスタの詳細が記述できない、(b) グローバルなラベルがないためコンテキストスイッチ/再開の実現が困難である、などがあるが、これらについて、以下のように対処した。

浮動小数点レジスタ、整数レジスタの代わりに、C 言語の浮動小数点型変数、整数型変数を (ABCL/ST コンパイラでレジスタ割り当ての処理を行い、ハードウェアが実際に持つレジスタ数を越えないようにして) 用いた。また、このように浮動小数点型変数 (レジスタ) と整数型変数 (レジスタ) が分離しているレジスタセットに対応できるようにしたことにより、将来、そのようなハードウェアアーキテクチャにコンパイラを移植する際の移植性が高められた。C 言語にグローバルなラベルがない点に関しては、今回の実装では関数名+ラベル番号を用いた。これは ABCL/f[4] の $STACKThreads$[5] で用いられている手法と基本的に同じである。実際、ある関数内のコンテキストスイッチする可能性がある場所にラベルをつけ、それぞれに整数値のラベル番号を割り当てておく。コンテキストスイッチは単にリターンすることで、再開は再び関数を実行し、関数の先頭の switch 文で対応するラベルへの飛び込み文を実行することで行う。

3.2 並列化と並列オブジェクト

KSR1 における実行では、1 プログラムには 1 プロセスを用い、並列化は、使用するプロセッサ数と同数のスレッドをプロセス内部で生成し各プロセッサに割り付けることで行った。つまり、単一アドレス空間内で複数個のスレッドを用いて並列化を行う。

並列実行方式としては、ただ一つのスケジューリングキュー (パケットキュー) を設け、ロックを用いてキューアクセスの排他制御を行う方式も考えられるが、細粒度処理ではアクセス集中によるオーバーヘッドが大きいと考えられるため、採用しなかった。また、このただ一つのキューを用いる方式を単純に並列オブジェクトによる並列処理に適用した場合、複数のスレッドが同時に一つの並列オブジェクトの状態を変えないよう、さらに各オブジェクトにもロックを設けて排他制御を行う必要があるため現実的ではない。そこで、それぞれのプロセッサ (スレッド) に、専用のパケットキューを設けて処理を行わせることにした。

各オブジェクトは、ある決ったプロセッサ (スレッド) に属することにする。こうすることにより、あるオブジェクトは同時に一つ以上のスレッドにより実行されないことを用いて、オブジェクト単位の排他制御はロックを用いずに実現できる。このような方式をとるのは、「オブジェクト内単一スレッド性」という ABCL の計算モデルによるところが大きいが、結果として、スレッドとオブジェクトデータの affinity を高め、オブジェクトデータに関するローカルキャッシュ間のコピーなどを少なくする。

3.3 共有メモリ上のメッセージパッシング

KSR は共有メモリ型計算機であるため、プロセッサ間通信は基本的に共有メモリを通じて行うことになる。この方式でプロセッサ間通信を実装するためには、例えば、共有メモリとしてのパケットキューを用意し、これに/これから、パケットを加える/取り出す必要がある。

別々のプロセッサのためのキューを管理する変数は、false sharing を避けるためには、別々のサブページに配置する必要がある。

パケットを固定サイズとしたことからも分かるように、メッセージ送信はメッセージ本体の転送とパケットのキューへの追加に分けて実現している。メッセージ本体とパケットを分離することでオーバーヘッドもあるが、メッセージがすぐ処理できない時に、メッセージ全体をコピーしなくて済むという利点もある。分散メモリである EM-4 では、メッセージ本体の転送は、遠隔書き込み命令を用いて行っていたが、KSR では、共有メモリを通して暗に行うことができる。

4 Lock-less ランタイムアーキテクチャ

4.1 実装

共有メモリとしてのパケットキューを用意し、これに/これから、パケット加える/取り出すことでメッセージパッシングを実現する際、注意しなければならないのは、あるプロセッサがキューにパケットを加えるとき、他のプロセッサとの競合を防ぐため、ロックをかけるなどの操作が必要になることである。ところが、ロック操作は、特に KSR のようにメモリが物理的には分散している場合には大変コストの高い操作となる。我々はこれに対処するために、ロックを排除したランタイムアーキテクチャ(lock-less runtime architecture) を考案した (図 1)。

図1に示したように、Lock-less ランタイムアーキテクチャでは、プロセッサ数 P に対し、 P^2 個の外部キューを用いてプロセッサ間の競合を避けている。同様のロックを排除した方式は、すでに [3] の論理シミュレーションで用いられていたが、我々の実装では特定のアプリケーション

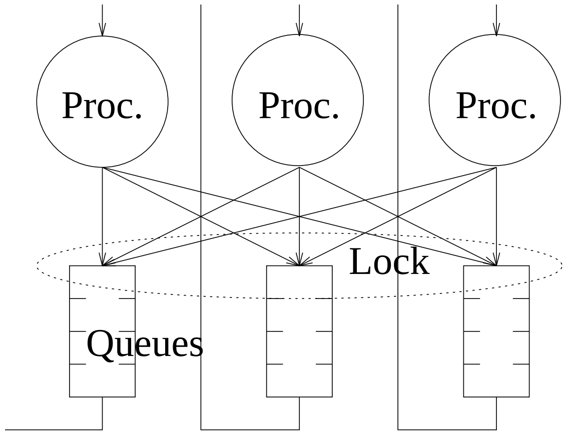

図 1(a) The Typical Implementaion: 各プロセッサはキューにパケットを追加する時にロックを用いる。キューからパケットを取り出す時にはロックは必要ない。

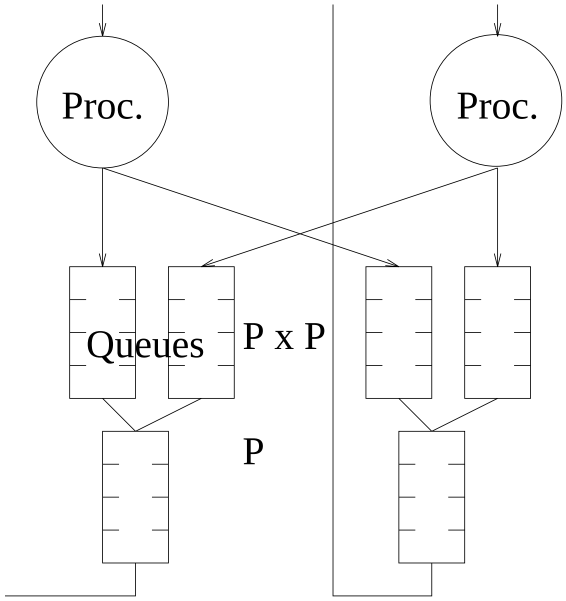

図 1(b) The Lock-less Runtime Architecture: パケットを追加するプロセッサと、パケットを追加される受取側のプロセッサの組について、一つの外部キュー (プロセッサ数を Pとすると全部で P^2個のキュー) を用意する。競合がないため外部キューにパケットを追加する際にロックは必要ない。外部キューのオーバーフローを防ぐため、外部キューのパケットは、受取側のプロセッサにより定期的に大きめのサイズの内部キューに移動させられる。外部キューからパケットを取り出す時や、内部キューのアクセスにもロックは必要ない。

図 1: Runtime Architecture for the KSR

ではなく言語実装のために用いている点、外部キューのオーバーフローを防ぐため内部キューを設けていることなどが異なる。また、実際には図 1(b) と異なり、ローカルな通信の場合 (宛先プロセッサが自分の場合) は直接ロックなしで内部キューにパケットを追加するようにした。Lock を用いる方式 (図 1(a)) についても、以下の節の比較においては、ローカルな通信の場合は図のキューとは別にそれぞれのプロセッサに用意した内部キューに直接ロックなしで追加し、図のキューと内部キューとを切替えながらスケジューリングするものを用いた。

提案する方式では、P^2 個のキューが必要なため、 P がある程度以上大きくなると必要なメモリ量が増えるなどの問題が生ずる。今回はそれほど大きな P は用いてないため、この問題には対処しなかったが、例えば、mP 個 ($m \ll P$) のキューを $\log_m P$ 段接続することで、キューの数を $mP \log_m P$ に減らすことは可能である。

4.2 ベンチマークプログラム

ベンチマークプログラムにはＮ体問題を用いた。

Ｎ体問題とは、Ｎ個の質点が互いに遠隔力を及ぼしながら運動しているニュートン力学系の状態をシミュレートする問題であり、適当な時間間隔で各質点の位置と速度を求める。J. Barnes と P. Hut による木構造を用いたアルゴリズム [1] は、近似により計算量を減らすものでその時間複雑度は $O(N \log N)$ である。ここで近似には、「質点をグループ化しているセルを考え、そのセルサイズと比べて距離が十分離れている質点には、セルに含まれる質点群の重心を用いて力を計算する」というものを用いる。

すでに発表したＮ体問題の並列オブジェクト指向数値計算アルゴリズム [6] は、J. Barnes と P. Hut の逐次アルゴリズムを拡張したもので、その時間複雑度は $O(N)$ 個の並列オブジェクトを用いて $O(\log N)$ となり、非常に高い並列度が得られることが分かっている。また、このアルゴリズムは、セルや質点を並列オブジェクトとして表現し、並列オブジェクト指向計算に特有な、並列オブジェクトの動的 (実行時) 生成や、メッセージ送信のネットワークトポロジーの動的変化を用いるものとなっている。

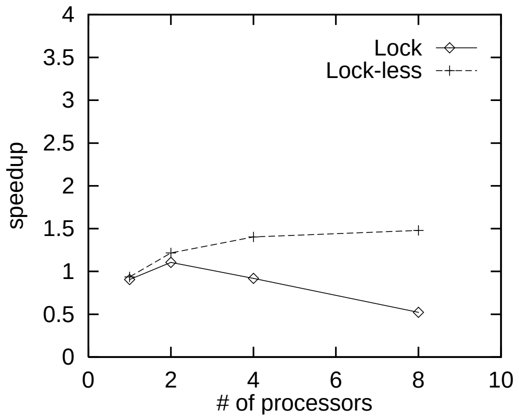

図 2(a) Maximum speedups

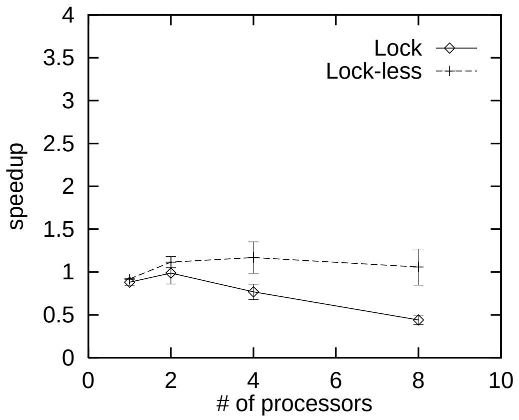

図 2(b) Average speedups and their standard deviations

図 2: Speedups for N-body Problem

今回、ABCL/ST で記述したこのアルゴリズムの三次元版 500 体問題をベンチマークプログラムとして、シミュレーションの 1 ステップ当たりの実行時間を比較に用いた。

4.3 測定結果

三次元版 500 体問題を単一プロセッサ用にコンパイルしたコードを KSR1 の 1 個の PE で実行したときのユーザ時間は、8.93 秒であった。

図 2に、並列実行用コードを 1,2,4,8 個の PE で実行した時の (単一プロセッサ用コードの 1 個の PE での実行に対する) スピードアップを示す。12 回の実行インスタンスについて測定を行い、そのうち最大のスピードアップについてを図 2(a) に、平均と標準偏差についてを図 2(b) に示した (エラーバーの片側幅が標準偏差)。並列実

行時のユーザ時間は、各スレッドのユーザ時間の平均 (総和を PE 数で除算したもの) を用いた。

図 2(a) によると、Lock-less ランタイムアーキテクチャでは、PE 数が 8 のとき最大 1.48 倍のスピードアップが得られたが、ロックを用いる方式では、PE 数が 4 以上で逆にスローダウンが見られた。これは、PE 数が増えた結果、キューのロックをめぐる競合が激しくなるためと考えられる。

一方、図 2(b) によると、Lock-less ランタイムアーキテクチャでも、平均すると 1.05〜1.17 倍程度のわずかなスピードアップしか得られないことが分かる。これについては、7節で議論する。

5 負荷分散指定時の評価

5.1 実装

4節では、並列オブジェクトを単にプロセッサ間にばらまいていたため、負荷は比較的均一でも並列オブジェクト間の通信がほとんどすべてプロセッサ間通信となってしまっていた。データ駆動計算機 EM-4 のように通信ネットワークが強力な場合はこれでもうまくいったが、KSR の場合はこの部分がネックとなりほとんどスピードアップが見られなかった。

ここでは、適切な負荷分散を行った場合について調べる。負荷分散の方法としては、質点の存在する空間 (立方体) を PE 数 (ここでは 2,4,8) に分割し、セルオブジェクト、質点オブジェクトともその中心座標がどの分割された空間に属するかによって割り付ける PE を決定した。近似により遠くのオブジェクトとの相互作用は少ないため、プロセッサ間通信の量を削減することができる。

5.2 測定結果

図 3に、並列実行用コードを 2,4,8 個の PE で実行した時の (単一プロセッサ用コードの 1 個の PE での実行のユーザ時間 8.93 秒に対する) スピードアップを示す。4節と同様、12 回のプログラム実行インスタンスについて、最大値 (図 3(a)), 平均値, 標準偏差 (図 3(b)) を示した。

図 3(a) によると、Lock-less ランタイムアーキテクチャでは、PE 数 4 のとき最大 2.26 倍、PE 数 8 のとき最大 3.47 倍のスピードアップが得られたが、ロックを用いる方式では、PE 数 4 で最大 1.52 倍のスピードアップしか得られず、PE 数 8 のときは最大 1.27 倍と、PE 数 4 の場合よりスローダウンする様子が見られた。

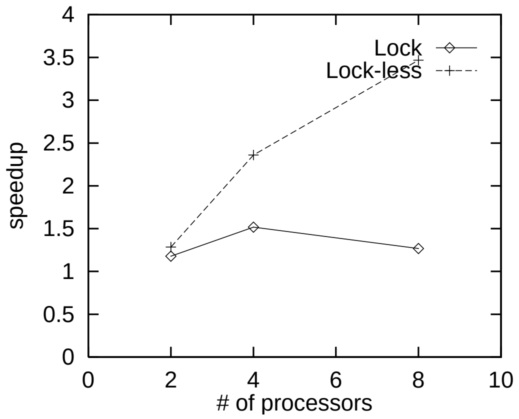

図 3(a) Maximum speedups

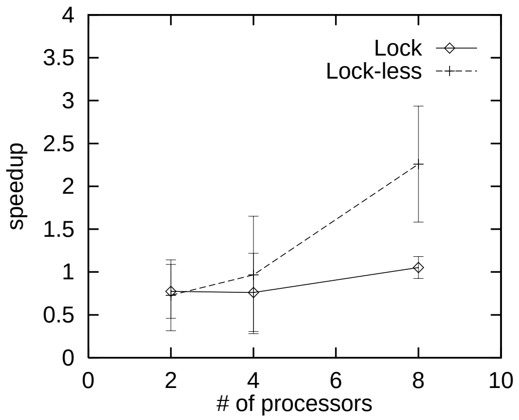

図 3(b) Average speedups and their standard deviations

図 3: Speedups for Load Balanced N-body Problem

6 スタックベース実行

6.1 実装

例えば、[5] に見られるようにタスクをスタックベースで実行すると高い効率が得られることが知られている。[5] では、ヒープをなるべく使わない点に主眼をおいているようにも見受けられるが、スタックベース実行だけでも、(1) データ参照の局所性が高まる、(2) ワーキングセットが小さくなる、の 2 点で、キャッシュや仮想記憶

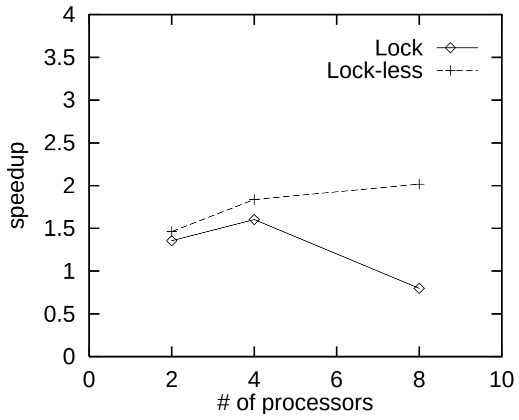

図 4(a) Maximum speedups

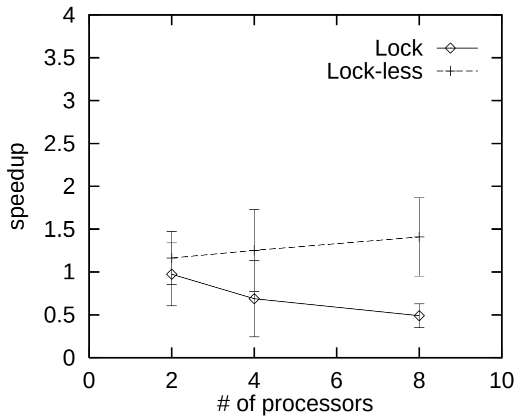

図 4(b) Average speedups and their standard deviations

図 4: Speedups for Load Balanced N-body Problem in a Stack-based Execution

を用いる計算機では実行時間短縮にかなりのインパクトがあると期待できる。

KSR1 への実装では、コンパイラの生成コードを C 言語としているため、ローカルな (同じノード内へ宛てられた) パケットは、パケットキューに追加する代わりに、そのままパケットハンドラを手続き呼び出しすることで、スタックベースの実行を実現した。

6.2 測定結果

三次元版 500 体問題を単一プロセッサ用にコンパイルしたコードを KSR の一つの PE でスタックベース実行したときの経過時間は、約 3.88 秒であった。これは、ここまでのパケットキューのみを用いる場合と比べて、2.30 倍高速である。

図 4に、並列実行用コードを 2,4,8 個の PE で実行した時の (単一プロセッサ用コードの 1 個の PE での実行のユーザ時間 3.88 秒に対する) スピードアップを示す。4節と同様、12 回のプログラム実行インスタンスについて、最大値 (図 4(a)), 平均値, 標準偏差 (図 4(b)) を示した。

図 4(a) によると、Lock-less ランタイムアーキテクチャでは、PE 数 4 のとき最大 1.84 倍、PE 数 8 のとき最大 2.02 倍のスピードアップが得られたが、ロックを用いる方式では、PE 数 4 で最大 1.60 倍のスピードアップしか得られず、PE 数 8 のときは最大 0.79 倍とスローダウンする様子が見られた。

スピードアップそのものは、5節のほうが大きかったが、スタックベース実行のほうが、1PE での処理が速いので、スタックベース実行で 8PE を用いたものが、今回測定したもので最も速かった。

7 議論

4,5,6節の結果から分かるように、いずれの場合も、Lock-less ランタイムアーキテクチャを用いたほうが、良い性能が得られた。その要因としては、ロック獲得の競合、ロックそのものの手間の他に、キューが専用になったことで、同じサブページに異なるプロセッサがパケットを追加するという一種の false sharing が避けられたことも考えられる。

図 2(b), 図 3(b), 図 4(b) の標準偏差のから分かるように、KSR1 における性能評価では、同じ並列コードの異なる実行インスタンスのユーザ時間にはかなりのばらつきがあった。これはマルチタスク・マルチユーザであるため環境の影響を受けやすいためと考えられる。特にネットワークがリング状となっているため、ネットワーク資源を多く消費する他のプロセスの影響を受けやすいと言える。実際、マシンが空いているときにたまたま測定した、PE 数 4 で負荷分散したコードの別の 12 回の実行インスタンスでは、標準偏差 0.28 倍で平均 2.40 倍のスピードアップが得られていた。(先のグラフでは最大 2.26 倍、平均 0.97 倍、標準偏差 0.68 倍であった)。

並列オブジェクト指向計算は、PE 間でデータよりは制御を移動する側面が強く、また、PE 間で移動されるメッセージデータについてはキャッシュしてもあまり意味がないため、共有メモリ

型並列計算機とはあまり相性が良くないと考えられる。しかし、本研究では、プロセッサ間通信の部分をなるべく軽く実装することで、細粒度の並列オブジェクト指向計算の分散共有メモリ型並列計算機による相応のスピードアップが可能なことが分かった。しかしながら、より本質的に共有メモリ型並列計算機との相性を良くするには、並列オブジェクト指向計算にコピー可能オブジェクトなどを導入してキャッシュが意味を持つようにしてやることが望ましいであろう。

8 結論と今後の課題

本研究では、比較的細粒度な並列オブジェクト指向言語 ABCL/ST の分散共有メモリ型並列計算機 KSR 上への実装を行った。本論文では、その実装を通して検討した、共有メモリ型並列計算機上での効率良い細粒度並列処理用実行方式について述べた。特に、プロセッサ間通信においてロックを用いない実行方式を提案し、その評価を行った。

今後の課題としては、KSR1 におけるプリフェッチやポストストアの利用の可能性、バス共有型共有メモリアーキテクチャへの移植などについて研究していきたい。

謝辞

マンチェスター大学コンピュータサイエンス学科の Prof. Ian Watson には同大学 MCC の KSR1 の使用にあたり便宜をはかっていただいた。また同学科の Dr. John Sargeant ら UFO project group には、八杉の同大学滞在中研究面でお世話になった。また、東大の米澤教授には八杉が昨年度御指導いただいた。ここに感謝する。本研究は、八杉が一部日本学術振興会特別研究員制度の助成を受けていた。

参考文献

[1] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[2] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a dataflow single chip processor. In *Proc. of the 16th Annual International Symposium on Computer Architecture*, pp. 46–53, June 1989.

[3] L. Soule' and T. Blank. Parallel Logic Simulation on General Purpose Machines. *Proc.ACM/IEEE Des.Auto.Conf.(DAC)*, pp. 166–171, 1988.

[4] K. Taura, S. Matsuoka, and A. Yonezawa. ABCL/f: A future-based polymorphic typed concurrent object-oriented language – its design and implementation –. In *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, 1994.

[5] K. Taura, S. Matsuoka, and A. Yonezawa. *StackThreads*: An abstract machine for scheduling fine-grain threads on stock CPUs. In *Proc. of Joint Symposium on Parallel Processing (JSPP)*, pp. 25–32, May 1994.

[6] M. Yasugi. A concurrent object-oriented programming language system for highly parallel data-driven computers and its applications. Technical Report 94-7e, Department of Information Science, Faculty of Science, University of Tokyo, Apr. 1994. (Doctoral Thesis, Mar. 1994).

[7] M. Yasugi, S. Matsuoka, and A. Yonezawa. ABCL/onEM-4: A new software/hardware architecture for object-oriented concurrent computing on an extended dataflow supercomputer. In *Proc. of 6th ACM International Conference on Supercomputing, Washington D.C.*, pp. 93–103, July 1992.

[8] A. Yonezawa ed. *ABCL: An Object-Oriented Concurrent System — Theory, Language, Programming, Implementation and Application.* The MIT Press, 1990.

[9] 八杉, 松岡, 米澤. ABCL/onEM-4: データ駆動計算機上の並列オブジェクト指向計算システムの高性能実装. 並列処理シンポジウム (JSPP'92) 論文集, pp. 171–178, June 1992.