

LAN 上での Linda タプル管理プロトコルの提案と評価

松田 哲史[†] 武並 佳則[†] 田口 哲也[†]

複数の計算機が LAN でつながれたネットワーク環境上で、Linda で書かれた並列処理プログラムを高速に実行するために、In, Rd (データ読み出し) のレイテンシを削減する手法と、その実現のために必要なタプル管理プロトコルを提案する。本手法では、1つのタプルに対して、(1)そのタプルにアクセスするプロセスの情報を利用する、(2)プロセスがタプルをどのような形態でアクセスするかの情報に基づきタプル管理プロトコルを使い分ける、(3)プリフェッチを行う、の3手段により、レイテンシの削減をはかる。提案するタプル管理プロトコルを用いた場合に、シリアルizabilityを満たすための十分条件を述べる。提案したタプル管理プロトコルの一部を Sun SparcStation 2 上に実現して性能を実測した結果、今回提案した手法がレイテンシ削減に役立つことを確認した。

A Proposal and an Evaluation of Linda Tuple Management Protocol for LAN Environment

TETSUSHI MATSUDA,[†] YOSHINORI TAKENAMI[†] and TETSUYA TAGUCHI[†]

In this paper, we propose methods to reduce the latency of In and Rd operations of a Linda program and tuple management protocols necessary to realize those methods in network environment where multiple computers are connected by LAN. The purpose of reducing the latency is to make the program run faster. In the proposed method, we try to reduce the latency for each tuple by (1) exploiting the information on what processes access the tuple, (2) exploiting the information on how those processes access the tuple and (3) prefetching. A sufficient condition to observe serializability when those tuple management protocols are used is given. We implemented part of the proposed tuple management protocols on Sun SparcStation 2 and confirmed that they can effectively reduce the latency of In and Rd operations.

1. はじめに

今日、複数の計算機が LAN でつながれているネットワーク環境は一般に普及しており、LAN でつながれた計算機ネットワークを1台のマルチプロセッサコンピュータとみなせば、最も普及している並列処理プログラム実行環境と見ることもできる。こうした状況から、ネットワーク環境上で並列処理プログラムを実行する分散並列処理システムが広く研究されている。この目的に用いられる分散並列処理記述言語の1つとして、Yale 大学で研究された Linda¹⁾がある。Linda では、共有データをタプルと呼ばれる単位で、In, Rd (読み出し), Out (書き込み) という3種類の操作でアクセスすることにより、データ共有とプロセス間同期を実現する。Linda を分散並列処理記述言語

として用いる際の課題の1つは、プログラムをいかに高速に実行するかということである。高速化へのアプローチとして、

- In, Rd のレイテンシの削減
- 計算速度と通信速度の比に応じた計算粒度の自動選択
- 動的負荷分散の実行

などが考えられる。本論文では、In, Rd のレイテンシ (In, Rd を実行開始してから、実際にタプルデータを受け取るまでの時間) を減らすための手法と、その実現のために必要なタプル管理プロトコルを提案する。提案する手法は、各タプルに対して、そのタプルを In, Rd, Out するプロセスと用いるべきタプル管理プロトコルを指定し、In, Rd に対してプリフェッチ実行を指定することにより、In, Rd のレイテンシ削減をはかる。タプル管理プロトコルは3種類存在し、複数のプロセスがあるタプルに対して In, Rd, Out を行う形態に基づき、おのおの場合に In, Rd のレイテンシを小さくするように設計されている。

[†] 住友電気工業(株)システムエレクトロニクス研究開発センター

Systems & Electronics R & D Center, Sumitomo Electric Industries, Ltd.

本論文の構成は以下のとおりである。最初に Linda の概要と LAN 上で実装する場合の問題点を説明する。そして、In, Rd のレイテンシを削減する手法として、各タプルに対して、そのタプルを In, Rd, Out するプロセスと用いるべきタプル管理プロトコルを指定し、In, Rd に対してプリフェッチ実行を指定することを提案する。次に、今回開発した3つのタプル管理プロトコルを説明し、これら3つのタプル管理プロトコルを混在させて用いた場合に、In, Rd, Out がシリアライズビリティという性質を満たすための十分条件を述べる。そして、今回提案したタプル管理プロトコルの一部を Ethernet でつながれた Sun SparcStation 2 上に実現して、性能を実測した結果を示す。最後に他の研究との比較を行い、まとめを述べる。

2. Linda の概要と問題点

Linda は、Yale 大学で研究された、プロセス間の同期、通信、プロセス生成を表現する機構である。Linda を逐次処理型の言語 (e.g. C, FORTRAN) と組み合わせることで、逐次処理言語を並列処理言語に拡張することができる。Linda における共有データの単位はタプルと呼ばれ、タプルは $(A_1, \dots, A_n)$ のように項 (A_i) の組として表される。項のデータ型としては、Linda が組み合わせられる相手の言語におけるポインタ型以外のデータが許される。Linda でのプロセス間の同期、通信、プロセス生成はすべてタプルへのアクセスとして表現される。各プロセスは、タプル空間と呼ばれる共有空間へ、タプルの書き込み (Out)、読み出し (Rd)、読み出しかつ消去 (In) という3種類の操作を行うことで通信と同期を実現する。In, Rd で指定するタプルに関しては、項として、通常の変数 (actual と呼ばれる) 以外に、同じ型の任意の変数とマッチングできる formal を用いることが許される。formal は $?x$ (x は組み合わせられる言語の変数) の形で記述される。In, Rd で読み出すタプルの指定はパターンマッチングを用いて行う。つまり、In, Rd は、タプル空間に存在するタプル (すべての項が actual) の中から、対応する actual の項とデータが一致し、対応する formal の項とデータ型が一致するタプルを選んで読み出す。formal の項に関しては、マッチしたタプルの項のデータが変数の値として返される。In, Rd で指定されるパターンにマッチするタプルが Out でタプル空間に書き込まれるまで、In, Rd

を実行したプロセスがブロックされることでプロセス間同期が実現される。プロセスの生成は、Eval でタプルをタプル空間に書き込むことで実現される。

Linda でプロセス間通信を実現する利点は、送信側も受信側も相手のことを全く意識せずに通信を行うことができることである。この抽象化により並列処理プログラムの作成が容易になる。しかし、この利点は、Linda で書かれた並列処理プログラムを実行する場合の実行効率の観点からは欠点となりうる。つまり、In, Rd のレイテンシが大きくなり、プログラムの実行速度が遅くなる可能性がある。以下でも少し詳しく説明する。Linda のタプル空間の実装方式としては、図1に示すように、Linda プログラムの実行に参加するすべての計算機上に、タプル空間内に存在するタプルの管理を行うタプルサーバー (TS) が存在し、これらの TS が Linda プログラムを実行するプロセスや他の TS と要求、応答をやりとりすることで In, Rd, Out を実現するものを採用する。各 TS は受信キューを持っており、TS に対する要求、応答を受信キューに入った順番に受信し、逐次的に処理する。あるプロセスが Out したタプルを他のどのプロセスが In, Rd で読み出すかわからないので、In, Rd を実行するプロセスが存在する計算機Aと、操作の対象となるタプルが存在する計算機Bが異なる場合が少なからずあることとなる。この場合、In, Rd を実行すると、In, Rd 要求が計算機Aから計算機Bに送信され、その要求に対してパターンマッチング操作が計算機Bで行われ、計算機Bから計算機Aへタプルデータが送信されることとなる (図2)。つまり、In, Rd

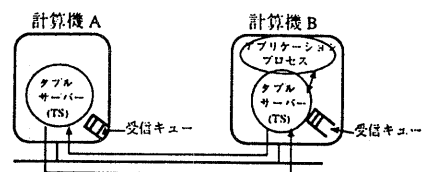


図1 タプル空間の実装方式
Fig. 1 Implementation model of tuple space.

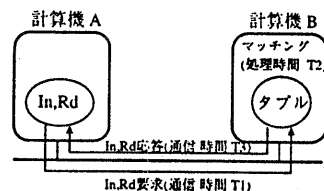


図2 In, Rd のレイテンシ
Fig. 2 Latency of In and Rd operation.

のレイテンシは図2における $T1+T2+T3$ となり、ネットワーク環境では $T1, T3$ が大きくなるので、In, Rd のレイテンシが大きくなる。以下では、通信時間を、送信を開始してから受信を終了するまでの時間（送受信処理のための計算機での処理や LAN のメディア上での伝送時間を含む）と定義し、通信時間 $T1, T3$ を減らす手法について議論する。議論の単純化のために、レイテンシの1つの原因である、1つの TS に要求が集中すること（コンテンション）により受信キューでの待ち時間が長くなることは無視しているが、3章で述べる手法は、タプルを適切に分散することや、タプルの複製を作ることで、コンテンションの問題の緩和にも有効に働く。マッチング処理時間 $T2$ に関しては、ハッシングなどの手法で十分短くできると仮定している。

3. In, Rd のレイテンシ削減手法

本章では、In, Rd のレイテンシを削減するための方法と、そのために必要な情報が何であるかを述べる。現在の実装では、これらの情報はプログラマが一種のアノテーションとして陽に記述する方式をとっている。

3.1 タプルを In, Rd, Out するプロセスの情報の利用

In, Rd のレイテンシについてもう少し詳しく検討する。In, Rd するプロセス、Out するプロセス、タプルを管理する TS の位置関係には、図3に示すように3通りの場合が考えられる。図中の $T1, T2, T3$ は、おのこの要求、応答の通信時間を示す。

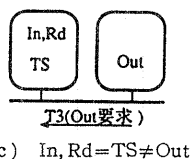
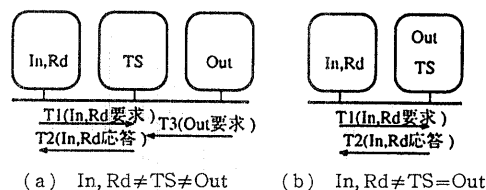


図3 In, Rd するプロセス、Out するプロセス、タプルを管理する TS の位置関係

Fig. 3 Location of a process which In or Rd a tuple, a process which Out it and TS which manages it.

表1 In, Rd のレイテンシ比較表
Table 1 Comparison of In and Rd latency.

図3 (a)	図3 (b)	図3 (c)
$\max(T1, T3+t)+T2$	$\max(T1, t)+T2$	$\max(T3+t, 0)$

データサイズより、 $T2=T3$ と考えて差し支えない。おのこの場合について、In, Rd のレイテンシの値がどうなるかを、表1に示す。表中で t は、In, Rd 実行開始から Out 実行開始までの経過時間を示す（Out が先に実行される場合 $t < 0$ ）。表1より、あるタプルに対して In, Rd, Out を行うプロセスがわかっているれば、In, Rd するプロセスが存在する計算機上の TS でそのタプルを管理するか、Out するプロセスが存在する計算機上の TS でそのタプルを管理することで、In, Rd のレイテンシが削減できることがわかる。

3.2 タプル管理プロトコルに関する情報の利用

次に別の場合を考える。図4(a)に示すように、まだ Out されていない1つのタプルを N 個のプロセスが Rd で読み出す場合に、最後に Rd 応答を受け取るプロセスの Rd レイテンシは $N \times T + (\text{In, Rd 実行開始から Out 実行開始までの経過時間})$ となる。 T は Rd 応答の通信時間である。ここで、図4(b)に示すように LAN 上のブロードキャスト機能を用いることで、ネットワーク上の通信を1回だけ行なって、複数のプロセスからの Rd 要求に対してタプルデータを返すことが可能となる。 T' を Rd 応答をブロードキャストする場合の通信時間とすると、Rd のレイテ

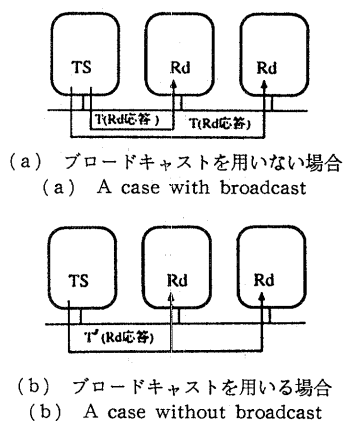


図4 ブロードキャストにより In, Rd のレイテンシが減る例

Fig. 4 Example in which broadcast operation reduces the latency of In and Rd operation.

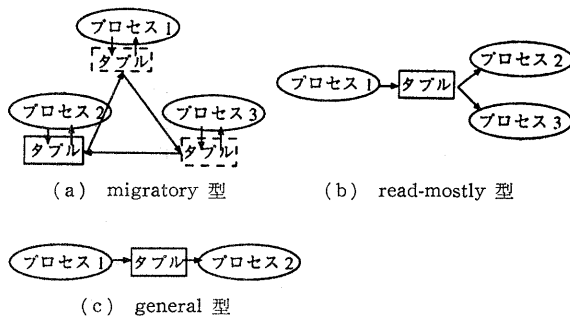


図 5 タプルアクセス形態の分類

Fig. 5 Classification of tuple access behavior.

ンシはどのプロセスから見ても $T' + (\text{In}, \text{Rd}$ 実行開始から Out 実行開始までの経過時間) となる。これにより、 Rd のレイテンシが削減できる。ただし、一般的に同一サイズのデータに対して、ブロードキャストのための通信時間が、1対1の通信のための通信時間よりも長い ($T' > T$) こと、ブロードキャストは本来無関係な計算機に対しても割り込み処理を生じることなどから、すべての場合にブロードキャストを用いるのが良いわけでない。このように、複数のプロセスがあるタプルをどのような形態で In , Rd , Out するかという情報に基づいて、適切なタプル管理プロトコルを用いることで、 In , Rd のレイテンシが削減できる。今回のシステムでは、タプルのアクセス形態を3つに分類し、おのおのの形態に適したタプル管理プロトコルを設計した。アクセス形態の分類は、文献2)に述べられている共有データアクセスタイプの分類を参考にしている。以下に3つのアクセス形態を説明する。

● migratory

1つのタプルが、複数のプロセスに In , Rd , Out されるが、プログラム実行中のある一定期間を区切ってみると1つのプロセスだけがアクセスしている場合に対応するアクセス形態。タプルを用いて分散共有データを実現している場合に有効な可能性がある (図5(a))。

● read-mostly

あるプロセスが Out で書き込んだ1つのタプルを、複数のプロセスが Rd で読み出す場合に対応するアクセス形態 (図5(b))。

● general

1つのタプルを In , Rd するプロセスがタプルに対して一意に決まるアクセス形態。他の2つのアクセス形態に当てはまらない場合のデフォルトアクセス形態でもある (図5(c))。

どのタプルアクセス形態の場合も、タプルを In , Rd , Out するプロセスの情報に基づき、タプルを管理する TS (migratory, read-mostly の場合は複数可) が決められる。

3.3 タプルのプリフェッチ

メモリアクセスレイテンシ削減のための技法であるプリフェッチを、Linda の In , Rd に適用することを考える。あるタプルを In , Rd する場合に、そのタプルを In , Rd する必要があることがわかってから、実際にそのタプルを利用する計算を開始するまでの間に、何か他の十分な量の計算を行うことができる場合、プリフェッチを行ってタプルをローカル計算機上の TS に持ってくることで、 In , Rd のレイテンシを隠せる可能性がある。そうでなくとも、複数のタプルに In , Rd を行う必要がある場合、それらのタプルに対してプリフェッチ要求を同時に発行することで、LAN のメディア上での通信時間と通信処理、タプルマッチング処理をオーバーラップして、 In , Rd のレイテンシが削減できる可能性がある。

4. タプル管理プロトコル³⁾

本章では、3.2 節で述べた3種類のタプルアクセス形態に対応するタプル管理プロトコル (以下プロトコルと略することもある) の概略を説明する。タプル管理プロトコルの指定はタプルの集合に対して行われ、異なるタプル管理プロトコルで管理されるタプルの集合には重なりがないことと、プログラム中の1つの In , Rd , Out 操作によって操作される可能性があるタプルは、すべて同じタプル管理プロトコルにより管理されることを仮定している。そして、3種類のプロトコルを混在させて使用したときにシリアライザビリティを満たすための十分条件を述べる。

4.1 タプル管理プロトコルが使用する通信プロトコル

以下で述べるタプル管理プロトコルが使用するトランスポート層の通信プロトコルを説明する。通信プロトコルはコネクションレスの送達保証付きパケット通信サービスを提供し、1対1, R-multicast, RS-multicast の3種類の通信プロトコルがある。1対1通信プロトコルは、1つのプロセスまたは TS から、1つのプロセスまたは TS へのパケット通信サービスを提供する。R-multicast 通信プロトコルは、1つのプロセスまたは TS から、複数のプロセスまたは TS へ

の同報パケット通信サービスを提供する。RS-multicast 通信プロトコルも、1つのプロセスまたは TS から、複数のプロセスまたは TS への同報パケット通信サービスを提供する。R-multicast との違いは、パケットの配送順序に関する制約条件が厳しいことである。すなわち、異なる送信者 A、B が RS-multicast でパケットを送信したときに、A からのパケットの受信者の集合と、B からのパケットの受信者の集合の両者に含まれている受信者いづれにおいても、A からのパケットと B からのパケットの受信順序が同じであることが保証される⁹⁾。その他のパケット配送順序に関する制約条件としては、同一の送信者が送信したパケットは、同一通信プロトコル内では、送信された順序と同じ順序で受信されることがある。これは、タプル管理プロトコルの実装が正しく実行されるために必要である。これらの通信プロトコルは IP 層の上に実現されており、通信プロトコル処理にかかる時間は、1対1、R-multicast、RS-multicast の順で増加する。RS-multicast 通信プロトコルは、文献10)に述べられている Reliable Broadcast method BB に基づいている。RS-multicast 通信プロトコルで送られるパケットに対応して、ネットワーク上に1つだけ存在するサーバーが、連番を付与するためのパケットを送信するようになっている。受信処理は、その連番の順に従ってパケットを受信者に配送するように制御している。

4.2 migratory プロトコル

プロセスが In、Rd を行うときに、大抵の場合同じ計算機上の TS がマッチするタプルを持っていることを期待している管理プロトコルである。migratory プロトコルは、ある TS に格納されたタプルは十分に長い時間その TS に留まると仮定しているのので、この仮定が成り立たないと実行効率が悪くなる。

migratory プロトコルでは、1つのタプルを管理する可能性がある TS は複数存在するが、ある1時点をとってみるとそのタプルを管理する TS は1つであり、タプルの複製は作られない。よって、1つのタプルが複数の In により読み出されることはない。

プロセスが Out を実行する場合、Out 要求がそのプロセスが存在する計算機上の TS へ1対1通信プロトコルで送信され、タプルはその TS が管理する。Out 要求を受信した TS は、Out 要求に含まれるタプルが、タプル待ちの In、Rd、Prefetch 要求や各種問い合わせとマッチングするか調べ、マッチすればタプルを送信する。

プロセスが In を実行する場合、In 要求を同一計算機上の TS へ1対1通信プロトコルで送信する。その TS にマッチするタプルが存在しなければ、TS は、マッチするタプルを管理する可能性がある TS すべてに対して、In 問い合わせを R-multicast 通信プロトコルで送信してタプルを探す。

プロセスが Rd を実行する場合、Rd 要求を同一計算機上の TS へ1対1通信プロトコルで送信する。その TS にマッチするタプルが存在しなければ、TS は、マッチするタプルを管理する可能性がある TS すべてに対して、Rd 問い合わせを R-multicast 通信プロトコルで送信してタプルを探す。

プロセスがプリフェッチを実行する場合、プリフェッチ要求を同一計算機上の TS へ1対1通信プロトコルで送信する。その TS にマッチするタプルが存在しなければ、TS は、マッチするタプルを管理する可能性がある TS すべてに対して、プリフェッチ問い合わせを R-multicast 通信プロトコルで送信してタプルを探す。プリフェッチされたタプルは、プリフェッチしたプロセスと同一計算機上の TS が管理する。

図6に示す、プロセス P が In を実行し、In にマッチするタプルが TS2、TS4 に存在する場合を例にとって、プロトコルの流れを説明する。() 内の番号は図中の番号に対応する。

(1) プロセス P が In を実行する。In 要求が同じ計算機上の TS1 に1対1通信プロトコルで送信される。

(2) In 要求を受け取った TS1 は、マッチするタプ

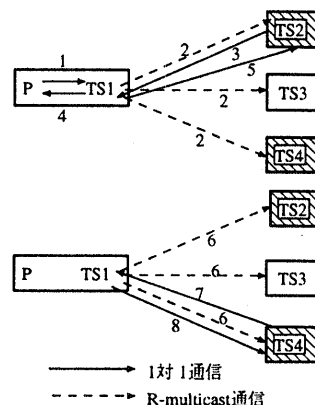


図6 migratory プロトコルの例 (In で TS1 でタプルが見つからない場合)

Fig. 6 Example of migratory protocol (in the case where In does not find a tuple at TS1).

ルがあるかどうかを調べる。マッチするタブルが存在しないので、In 要求をタブル待ち In 要求として記録すると同時に、そのタブルが存在する可能性があるすべての TS (TS2, TS3, TS4) に対して、In 問い合わせを R-multicast 通信プロトコルで送信する。

(3) In 問い合わせを受けた TS2 は、マッチするタブルがあるかどうか調べる。タブルがあるので、タブルを In 問い合わせ応答として要求元の TS1 へ送信して、タブルを Ack 待ちの状態にする。TS3 は、マッチするタブルがないので、In 問い合わせをタブル待ち問い合わせとして記録する。

(4) In 問い合わせ応答を受信した TS1 は、In 問い合わせの元となった In 要求がまだタブル待ちの状態に存在するかどうか調べる。存在するので、プロセス P へタブルを 1 対 1 通信プロトコルで送信して、In 要求を捨てる。

(5) TS1 は、In 問い合わせ応答を送信した TS2 に対して Ack を返す。Ack を受信した TS2 は、Ack 待ちのタブルを捨てる。

(6) TS1 は、In 問い合わせの対象となったすべての TS に対して cancel 要求を R-multicast で送信する。cancel 要求を受け取った TS3 は、対応するタブル待ち In 問い合わせを捨てる。

(7) 2 で In 問い合わせを受信した TS4 は、マッチするタブルがあるかどうか調べる。タブルがあるので、タブルを In 問い合わせ応答として要求元の TS1 へ送信して、タブルを Ack 待ちの状態にする。

(8) In 問い合わせ応答を受信した TS1 は、In 問い合わせの元となった In 要求が TS1 に存在するかどうか調べる。(4) より存在しないので、In 問い合わせ応答を送信した TS4 に対して Nack を返す。Nack を受信した TS4 は、Ack 待ちのタブルを、Out 要求で受け取ったときと同様に処理する。

Rd に対するプロトコルは、タブルを Ack 待ちの状態にしない点と、Ack, Nack を送信しない点を除けば In の場合と同じである。

プリフェッチを実現するプロトコルは、各要求がプリフェッチ用のものとなるだけで、基本的に In のプロトコルと同じである。

4.3 read-mostly プロトコル

1 つのタブルの複数コピーが存在し、それらのコピーが複数の TS に管理される。

プロセス P がタブルを Out する場合、タブルを管理するすべての TS に対して、Out 要求を RS-multi-

cast 通信プロトコルで送信する。Out 要求を受信した TS は、Out 要求に含まれるタブルと、タブル待ちキュー内の In, Rd 要求とがマッチするか調べ、マッチしてかつ In, Rd 要求元と TS が同一計算機上であれば、タブルを In, Rd 要求元のプロセスへ 1 対 1 通信プロトコルで送信する。Out されたタブルが In されなければ、タブルを記憶領域に格納する。

プロセス P がタブルを In する場合、タブルを管理するすべての TS に対して、In 要求を RS-multicast 通信プロトコルで送信する。In 要求を受信した TS は、マッチするタブルが存在するかどうか調べる。存在しない場合、In 要求をタブル待ち In 要求としてタブル待ち In 要求キューに記録する。存在する場合、TS がプロセス P と同じ計算機上に存在するならば、TS はプロセス P へタブルを 1 対 1 通信プロトコルで送信して、タブルを捨てる。In 要求にマッチするタブルが存在し、TS がプロセス P と異なる計算機上に存在するならば、タブルを捨てるだけである。

プロセス P がタブルを Rd する場合、プロセス P と同じ計算機上の TS へ Rd 要求を 1 対 1 通信プロトコルで送信する。Rd 要求を受信した TS は、マッチするタブルが存在するかどうか調べる。存在しない場合、Rd 要求をタブル待ち Rd 要求としてタブル待ち Rd 要求キューに記録する。存在する場合、TS はプロセス P へタブルを 1 対 1 通信プロトコルで送信する。

In, Rd を実行するプロセスと同じ計算機上の TS の記憶領域に、タブルのコピーが格納されるので、プリフェッチに対しては何の処理も行わない。

In, Out 要求は RS-multicast 通信プロトコルで送信されているので、すべての TS は同じ順序で In, Out 要求を受信し、In, Out 要求を受信した順に逐次的に処理する。Out, In 要求の処理結果 (どのタブルがマッチするか、処理後、TS が管理するタブルの中で read-mostly プロトコルで管理されるものの集合を表すキューの内容と、タブル待ちの In 要求のキューの内容) は、要求処理前の、TS が管理するタブルの中で read-mostly プロトコルで管理されるものの集合を表すキューの内容と、タブル待ちの In 要求のキューの内容にのみ依存するように、TS のアルゴリズムが設計されており、プログラム開始時にどちらのキューも空である。よって、仮にある In 要求にマッチしうるタブルがタブル空間内に複数存在したとしても、その In 要求が読み出すタブルはどの TS 上でも同じものとなる。よって、1 つのタブルが複数の In

に読み出されることはない。

4.4 general プロトコル

1つのタプルに対して、そのタプルを管理する TS が1つ決まる。プリフェッチが行われなければ、タプルのコピーは作られない。1つのタプルをプリフェッチできるプロセスは最大1つであり、プリフェッチされたタプルのコピー（高々1つ）を、プリフェッチしたプロセスと同一計算機上の TS が管理する。

プロセスが Out を実行する場合、Out 要求をタプルを管理する TS へ1対1通信プロトコルで送信する。Out 要求を受信した TS は、Out 要求に含まれるタプルが、タプル待ちの In, Rd, Prefetch 要求や各種問い合わせとマッチするか調べ、マッチすればタプルを送信する。

プロセスが In を実行する場合、In 要求をタプルを管理する TS へ1対1通信プロトコルで送信する。In 要求を受信した TS は、In 要求にマッチするタプルが存在するか調べる。タプルが存在する場合、タプルをプロセスへ1対1通信プロトコルで送信して、タプルを捨てる。マッチするタプルが存在しなければ、In 要求をタプル待ち In 要求として記録する。

プロセスが Rd を実行する場合、TS がマッチしたタプルを捨てない点と、プリフェッチが行われている場合の処理を除けば In の場合と同じである。

プロセスがプリフェッチを実行する場合、プリフェッチ要求を同一計算機上の TS へ1対1通信プロトコルで送信する。プリフェッチ要求を受信した TS は、プリフェッチ問い合わせをタプルを管理する TS へ1対1通信プロトコルで送信する。プリフェッチ問い合わせを受信した TS は、プリフェッチ問い合わせにマッチするタプルが存在するか調べ、存在すればタプルのコピーを要求元の TS へ1対1通信プロトコルで送信すると同時に、コピーが要求元の TS に存在することを記録する。マッチするタプルが存在しなければ、プリフェッチ問い合わせをタプル待ちプリフェッチ問い合わせとして記録する。

以下に、3つの場合について general プロトコルの流れを説明する。（ ）内の番号は図中の番号に対応する。

プロセスPが自分でプリフェッチしたタプルを In する場合（図7(a)）

(1) プロセスPはプリフェッチをしているので、同一計算機上の TS1 に In 要求を1対1通信プロトコルで送信。

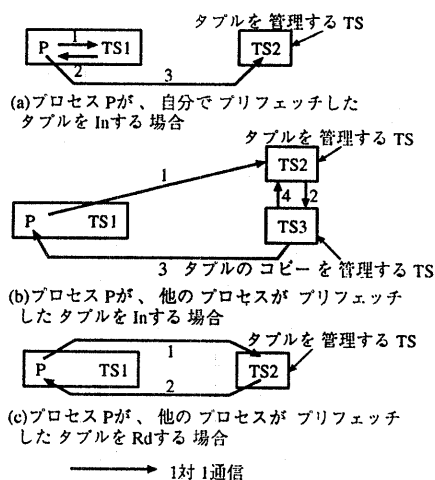


図7 general プロトコルの例
Fig. 7 Example of general protocol.

(2) In 要求を受信した TS1 は、プリフェッチに対するタプルが存在するか調べる。存在するので、タプルをプロセスPへ1対1通信プロトコルで送信する。

(3) タプルを受信したプロセスPは、タプルがプリフェッチによるコピーかどうか調べる。コピーなので、delete 要求をタプルを管理する TS2 へ1対1通信プロトコルで送信する。delete 要求を受信した TS2 は、対応するタプルを捨てる。

プロセスPが、他のプロセスがプリフェッチしたタプルを In する場合（図7(b)）

(1) プロセスPはプリフェッチをしていないので、タプルを管理する TS2 へ In 要求を1対1通信プロトコルで送信。

(2) TS2 は、In 要求に対してマッチするタプルを探す。見つかったタプルがプリフェッチされており、TS3 にコピーが存在するので、TS2 は In 問い合わせを TS3 へ1対1通信プロトコルで送信。タプルは捨てて、In 要求はペンディングにする。

(3) In 問い合わせを受信した TS3 は、タプルのコピーが存在するかどうか調べる。コピーが存在するので、コピーをプロセスPへ1対1通信プロトコルで送信し、コピーを捨てる。

(4) TS3 は、In 問い合わせに対する肯定応答を TS2 へ1対1通信プロトコルで送信。肯定応答を受信した TS2 は、対応するペンディングの In 要求を捨てる。

このように、タプルを管理する TS は、In 要求にマッチしたタプルが他のプロセスにプリフェッチされ

ている場合、そのタプルに対する In 要求を、プリフェッチされたコピーを管理する TS へ転送する。よって、プリフェッチによりタプルのコピーが存在する場合でも、1つのタプルが複数の In に読み出されることはない。

プロセス P が、他のプロセスがプリフェッチしたタプルを Rd する場合 (図 7(c))

(1) プロセス P はプリフェッチをしていないので、タプルを管理する TS 2 へ Rd 要求を 1 対 1 通信プロトコルで送信。

(2) TS 2 は、Rd 要求に対してマッチするタプルを探す。他のプロセスがすでにプリフェッチしているタプルが存在するので、TS 2 はプロセス P へタプルを 1 対 1 通信プロトコルで送信。

4.5 シリアライザビリティを満たすための十分条件

Linda におけるシリアライザビリティを説明するために、Linda プログラムを実行した複数のプロセスが行った、In, Rd, Out の間の因果順序 (causal ordering) $\rightarrow, \Rightarrow$ を以下のように定義する。

- Op 1(=In, Rd, Out) と Op 2(=In, Rd, Out) が、1つのプロセスによって実行され、Op 1 が Op 2 よりも先に実行されたならば Op 1 $\rightarrow$ Op 2。
- Op 2(=In, Rd) が読み出したタプルが、Op 1(=Out) が書き込んだタプルであるならば、Op 1 $\Rightarrow$ Op 2。
- Op 1(=Rd) の読み出したタプルが、Op 2(=In) の読み出したタプルと同じ Out によって書き込まれているならば、Op 1 $\Rightarrow$ Op 2。

このとき、Linda プログラムの実行結果がシリアライザビリティを満たすとは、複数のプロセスが実行した In, Rd, Out の間の因果順序に矛盾しないように、In, Rd, Out をある順序で並べることができることと定義される。例を図 8 に示す。Linda のタプル空間の定義より、Linda プログラムが正しく実行されるための必要十分条件は、この性質が満たされることと、1つのタプルが 2 つ以上の In によって読み出されることがないことの 2 つである。1つのタプルが 2 つ以上の In によって読み出されることがないことは、プロトコルの説明で述べている。よって、シリアライザビリティを満たすことが、Linda プログラムが正しく実行されるための十分条件である。今回提案するタプル管理プロトコル 3 種類を図 1 の実装モデルで実現する場合、次の条件 1 を満たせば、シリアライザビリティ

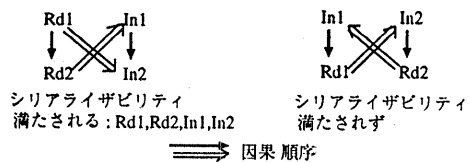


図 8 シリアライザビリティ
Fig. 8 Serializability.

ィが満たされることを証明している。

[条件 1] あるプロセスが Rd, Out 要求を送信する場合、自プロセスが以前送信した In 要求と delete 要求が、すべての送信対象である TS の受信キュー (図 1 参照) に入るまで Rd, Out 要求を送信しない。

5. 評価結果

以上で説明したタプル管理プロトコルを、プリフェッチに関する部分を除いて Sun SparcStation 2 上に実装し、Ethernet を LAN として使用した場合の実測結果を述べる。またプリフェッチにより得られる改善効果を解析する。

5.1 read-mostly プロトコルによる Rd レイテンシ削減効果の評価

1つのプロセスが連続してタプルを Out し、他のプロセスが同時に Rd でそれらのタイプを読み出す場合について、タプルを read-mostly プロトコルで管理した場合と、general プロトコルで Out するプロセスと同じ計算機上の TS で管理する場合とを比較した結果を表 2 に示す。この結果より、読み出すデータのサイズと計算機数が大きくなると、read-mostly プロトコルにより Rd のレイテンシが減らせることがわかる。データサイズ、計算機数が小さい場合に、read-mostly プロトコルを使用した方が Rd のレイテンシが長くなるのは、read-mostly プロトコルが利用する RS-multicast 通信プロトコルの方が、general プロト

表 2 Rd のレイテンシ測定結果
Table 2 Measurement result of latency of Rd operation.

Rd する計算機数 データサイズ	2	3	4
	Rd 1 回あたりの時間 (ms)		
read-mostly 使用			
2 KB	10.06	9.70	9.84
3 KB	10.79	10.61	11.14
5 KB	12.77	12.55	13.12
general 使用			
2 KB	5.41	7.47	9.87
3 KB	7.25	10.02	14.26
5 KB	10.54	15.41	25.24

表 3 read-mostly の Rd にかかる時間の内訳の推測値 (ms)
Table 3 Estimate of the decomposition of Rd latency for read-mostly case (ms).

データサイズ	Rd する計算機数 2			3			4		
	(1)	(2)	待ち時間	(1)	(2)	待ち時間	(1)	(2)	待ち時間
2 KB	3.17	0.61	6.28	3.17	0.61	5.92	3.17	0.61	6.06
3 KB	4.5	0.61	5.68	4.5	0.61	5.5	4.51	0.61	6.02
5 KB	7.17	0.61	4.99	7.17	0.61	4.77	7.15	0.61	5.34

表 4 general の Rd にかかる時間の内訳の推測値 (ms)
Table 4 Estimate of the decomposition of Rd latency for general case (ms).

データサイズ	Rd する計算機数 2			3			4		
	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)
2 KB	1.02	4.08	0.31	1.02	4.08	2.37	1.02	4.08	4.77
3 KB	1.02	5.13	1.1	1.02	5.13	3.87	1.02	5.13	8.11
5 KB	1.02	7.23	2.29	1.02	7.23	7.16	1.02	7.23	16.99

コルが用いる 1 対 1 通信プロトコルよりも通信処理コストが高いからである。以下では、Rd にかかる時間の内訳を、実験から求めた通信処理のための CPU 処理時間の近似式と、簡易テストプログラムで TS 等での処理時間を測定した結果から推測する。read-mostly プロトコルを用いた場合、Rd を行うプロセスが存在する計算機からみての Rd にかかる時間の内訳は、

- (1) Out 要求の受信処理の CPU 処理時間
- (2) 同一計算機上の TS への Rd 要求/応答の送受信時間および TS でのマッチング処理時間

と待ち時間よりなる。これらの推測値を表 3 に示す。表中の番号は上記の番号に対応する。データサイズが大きくなると待ち時間が減少することは、データ転送レートの上と対応しており、RS-multicast 通信プロトコルの ack 待ち時間が減少していると考えられる。

general プロトコルを用いた場合、Rd にかかる時間の内訳は、

- (1) Rd 要求の送信開始から受信終了までの時間
- (2) Rd に対して返されるタブルの送信開始から受信終了までの時間
- (3) TS での処理時間および待ち時間

に分けられる。これらの推測値を表 4 に示す。表中の番号は上記の番号に対応する。表 4 の (3) より、データサイズや Rd する計算機台数が増えるに従って、コンテンツンションによる待ち時間が増大していることがわかる。

5.2 read-mostly プロトコルを用いるアプリケーションでの評価

社内で GaAs デバイスの評価に用いているデバイスシミュレータプログラムを並列化して評価した。デ

バイスシミュレータでは、バンド行列 (5,355×5,355, 帯幅 106×108) の LU 分解が実行時間の 90% 以上を占め、今回は LU 分解の部分をマスター/ワーカモデルで並列化した。各ワーカが担当する仕事はあらかじめ決まっている。マスターはバンド行列のデータを分割して Out し、ワーカの計算結果をすべて In する。ワーカはピボット担当のものとそれ以外のものに分けられ、ピボット担当のワーカは順々に交替してゆく。各ワーカはバンド行列の自分の担当部分の一部を In する。ピボット担当のワーカは、他のワーカが Rd する行消去に必要なデータ (ピボットデータと呼ぶ) を Out して、他のワーカが行消去を行うのと並列に、マスターが In する計算結果を Out する。Out が終了した時点で次のワーカがピボット担当のワーカになり、前ピボット担当ワーカは自分の担当部分の別の一部分を In して、非ピボット担当ワーカとして実行を続けるというサイクルを繰り返す。実際に、

1. read-mostly プロトコルを適用した場合 (read-mostly)
2. general プロトコルを適用し、Out するプロセスと同じ計算機上の TS がタブルを管理する場合 (general)
3. C-Linda の実装である、米国 SCA 社の Network Linda システム (ver. 2.4.6) で実行した場合 (SCA)

このバージョンの SCA Linda は、タブルの複製をサポートしない。general プロトコルでプリフェッチがないようなプロトコルを用いていると想像される。

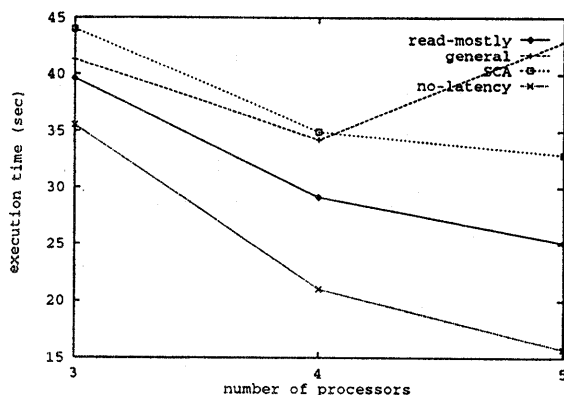


図 9 並列 LU 分解実行時間

Fig. 9 Execution time of parallel LU decomposition.

4. In, Rd のレイテンシが 0 と仮定した場合 (no-latency)

の 4 通りの場合について, LU 分解 1 回の実行時間を図 9 のグラフに示す. no-latency の場合の実行時間は, ワーカー数 n , ワーカーが 1 回に担当する行分のピボットングと前進消去の逐次処理にかかる時間を a , ピボットデータの Out のための通信時間を c としたときに,

$$\text{並列実行時間} = \text{逐次実行時間} \times (3/(2n) + c/b)$$

とモデル化されることより, $c=0$ とおいて求めた. LU 分解実行において, Rd のレイテンシはクリティカルパス上にあるので, Rd のレイテンシが長くなるとプログラムの実行時間が長くなる. グラフより, general プロトコルを用いた場合は, TS における Rd のコンテンションにより Rd レイテンシが増加し, 計算機台数が 4 台から 5 台に増えるとかえって実行時間が長くなっている. read-mostly プロトコルを用いた場合は, general プロトコルを用いた場合よりも実行時間が短く, Rd のレイテンシを相対的に低く抑えられていることがわかる. また, 計算機台数を増やすに従って実行時間が短くなっており, TS における Rd のコンテンションの影響を受けていない. また, いずれの台数でも, read-mostly プロトコルを用いた方が, SCA Linda システムを用いる場合よりも実行時間が短くなっている. 計算機台数 5 台の場合に, Out の通信時間は並列実行時間の計算式より c の値を逆算して, general の場合で 136.9 ms, SCA Linda において 86.5 ms となる. SCA Linda が図 9 の general と同じ方式でタブル管理を行い, 通信プロトコルの実行効率も同じと仮定すると, 通信時間は, 最初にタブルを受け取るワーカーで 73.18 ms, 最後にタブルを受け

取るワーカーで 143 ms と計算される. よって, SCA Linda ではクリティカルパス上で存在するワーカー (次にピボットを担当するワーカー) が一番最初にタブルを受け取っていると推測される.

5.3 プリフェッチの改善効果の解析

general プロトコルを用いたときに, プリフェッチを実行した時点ですでに対象のタブルが Out されておりプリフェッチが成功する場合に, In, Rd のレイテンシをどれだけ減らせるかを解析する. プリフェッチ実行時点から Rd 実行時点までの計算時間を T_c , Rd 要求の送信のための CPU 処理時間を RdS , Rd に対するタブルを受信するための CPU 処理時間を RdR , Rd 要求の送信処理が終了してからタブルの受信が開始するまでの時間を T_w , 同一計算機上の TS への送信または受信にかかる時間を T_1 とする. この場合, プリフェッチを行う場合と行わない場合とで, プリフェッチ開始から Rd 終了までのプログラムの実行時間には図 10 に示すような関係が成り立ち,

プリフェッチなしの場合の実行時間

$$= T_c + RdS + T_w + RdR$$

プリフェッチありの場合の実行時間

$$= \max(T_c + T_1, T_w) + RdS + RdR + 2T_1$$

となる. よって, $T_w > T_c + T_1$ が成り立っているとき,

プリフェッチありの場合の実行時間—プリフェッチなしの場合の実行時間 $= 2T_1 - T_c$

$T_w \leq T_c + T_1$ が成り立っているとき,

プリフェッチありの場合の実行時間—プリフェッ

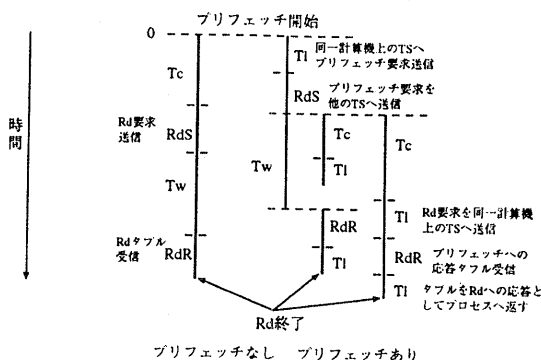


図 10 プリフェッチを実行しない場合と, 実行した場合のプログラム実行時間

Fig. 10 Execution time of a program with and without prefetching.

ちなしの場合の実行時間 $=3T1-Tw$.

今回の SS 2 上での実装では, $T1$ の値は 0.3 ms 程度. 5.1 節で用いたプログラムの場合, コンテンションを考慮しなければ, Tw の値はデータサイズが 1 KB で 2.25 ms, データサイズが 1,220 B 以上の場合 2.83 ms と計算される. これらの値より, 5.1 節のプログラムでデータサイズが 3 KB の Rd について, $Tc < 2.53$ ms ならば $Tc > 2T1 = 0.6$ ms でなければプリフェッチの効果は得られない. $Tc \geq 2.53$ ms の場合, 常に 1.93 ms のレイテンシ短縮効果があり, これが最大の効果となる. 実際には, コンテンションにより Tw の値がこれ以上になることも考えられるので, より大きな効果が得られる可能性もあるが, そのためには Tc が大きくならねばならない.

6. 関連する他の研究との比較

In, Rd のレイテンシ削減に関する研究としては, Linda の発案者である Gelernter らによるもの⁴⁾がある. Linda における In, Rd のレイテンシは, In, Rd 要求とそれに対する応答を送信するための通信時間と, TS におけるタプルマッチング処理に掛かる時間の 2 つに分けられる. 文献 4) では, コンパイル時にマッチングする可能性のあるタプルを検出し, この情報を元に部分計算の手法を用いて In, Rd を実現する実行時ライブラリを書き換えて, タプルマッチングに掛かる時間を減らす手法が述べられている. 通信時間削減の手段としては, 本論文で述べた.

1. 各タプルに対して, そのタプルを In, Rd , Out するプロセスの情報利用
 2. 各タプルに対して, 用いるべきタプル管理プロトコルの情報利用
 3. In, Rd に対して, プリフェッチ利用
- の 3 つの手段のうち, 1 のみが考慮されている. 本研究では, 2, 3 も考慮することにより, In, Rd のレイテンシ削減の可能性を高くしたといえる.

タプル管理プロトコルに関する研究としては, S/net Linda Kernel^{5), 6)}と Xu らによるもの⁷⁾がある.

S/net Linda Kernel は, S/net という, ブロードキャストを通信の基本とするバス結合型のネットワーク上での, Linda タプル空間の実装である. S/net につながるすべての計算機上に TS が存在し, タプル空間中のすべてのタプルの複製を各計算機上の TS が管理する. In, Out 要求は S/net 上でブロードキャストされる. 本論文で説明した read-mostly プロトコ

ルは, 文献 5) に述べられているタプル管理プロトコルと似ているが, In 要求の処理において, 文献 5) のプロトコルではタプル削減の可否を決定する TS はタプルを Out した計算機上の TS と決まっているが, read-mostly の場合は In 要求の受信順序制御により任意の TS が独自に判断するようになっている. このため, 文献 5) のプロトコルは計算機の故障に対してフォールトトレラントではないが, read-mostly プロトコルは使用している RS-multicast 通信プロトコルが計算機の故障に対してフォールトトレラントならば (e.g. ISIS⁸⁾の GBCAST), 計算機の故障に対してフォールトトレラントできる. また, 文献 5), 6) はシリアライゼビリティの問題に言及していない.

Xu らによる研究⁷⁾は, フォールトトレラントなタプル空間実現のためのタプル管理プロトコルに関するものである. Linda プログラムを実行する各計算機上にタプルを管理する TS が存在し, 1 つのタプルの複製が複数の TS によって管理されている. 文献 7) のプロトコルでは, In 要求はすべての TS に対して送信され, 各 TS は自分が管理するタプルの中で In 要求にマッチするものをすべてにロックをかけて, マッチしたタプルすべてを In 要求元へ返す. In 要求元は, すべての TS から応答が返されるのを待ち, 返された応答に含まれるタプルの積集合の中から 1 つ In の対象となるタプルを選び, そのタプルを選んだことをすべての TS に通知する. このプロトコルがシリアライゼビリティを満たすための十分条件が示されている. この記述からわかるように, 文献 7) のプロトコルで送信されるデータ量の方が, 本論文の read-mostly プロトコルで送信されるデータ量よりも多い. また, 文献 7) のプロトコルでは, In 要求元はすべての TS から応答が返るのを待たなければならないが, read-mostly プロトコルでは 1 つの TS (同じ計算機上の TS) からの応答のみを待てばよく, その TS での処理は他の TS との通信を必要としない. よって, read-mostly プロトコルの方が文献 7) のプロトコルよりも実行効率が良いと考えられる.

S/net Linda kernel, Xu らの研究いずれも, プリフェッチをサポートするための機能はタプル管理プロトコルにない.

7. ま と め

ネットワーク環境上で, Linda で書かれた並列処理プログラムを高速実行するための手段として, In, Rd

のレイテンシを削減する手法と、その実現のために必要なタプル管理プロトコルについて述べた。具体的には、各タプルに対して、そのタプルに対して In, Rd, Out を行うプロセス、用いるべきタプル管理プロトコルを指定し、In, Rd に対してプリフェッチ実行を指定することにより、In, Rd のレイテンシ削減をはかる。そして、複数のプロセスがあるタプルに対して In, Rd, Out を行う形態に合ったタプル管理プロトコル3種類を提案し、これら3種類のタプル管理プロトコルを混在させて用いた場合に、シリアル化ビリティを満たすための十分条件を述べた。さらに、今回提案した手法の中の read-mostly プロトコルに関して、Rd のレイテンシが削減されることを実測により示しその有効性を確認し、general プロトコルのプリフェッチの効果を解析した。本研究の意義は、In, Rd のレイテンシを削減するための3つの手法をとりいれ、それを実現するタプル管理プロトコルを開発し、実測と解析によりその有効性を示した点にある。

今後の課題としては、

- In, Rd のレイテンシ削減に必要な情報のコンパイラによる自動抽出方法
特に、共有メモリ型並列計算機上のメモリプリフェッチに関する研究⁹⁾などを参考として、どのような条件の場合にいつプリフェッチを行うかを決定するためのアルゴリズムと、タプル管理プロトコル選択のためのアルゴリズムを検討する必要がある。
 - タプルの位置がプログラム実行時に動的に決まる場合の、In, Rd レイテンシの削減方法検討
 - タプル管理プロトコルのスケラビリティの検討
 - 他の Linda プログラム高速化手法の検討
- が挙げられる。

謝辞 日頃よりご指導頂いております内田所長、川村主研、村瀬主研、および議論に加わって頂いたシステム技術グループ、CAE 技術グループの皆様へ感謝致します。

参 考 文 献

- 1) Carriero, N. and Gelernter, D.: Linda in Context, *CACM*, Vol. 32, No. 4, pp. 444-458 (1989).
- 2) Bennet, J., Carter, J. and Zwaenepoel, W.: Adaptive Software Cache Management for Distributed Shared Memory, *ISCA 1990*, IEEE, pp. 125-134 (1990).
- 3) 松田哲史: Linda タプル管理プロトコル案, 住友電気工業社内ブリーフレポート, 186-693-920024 (1992).
- 4) Carriero, N. and Gelernter, D.: Tuple Analysis and Partial Evaluation Strategies in the Linda Precompiler, *Language and Compilers for Parallel Computing*, Gelernter, D., Nicolau, A. and Padua, D. (eds.), Pitman, pp. 114-125 (1990).
- 5) Carriero, N. and Gelernter, D.: The S/Net's Linda Kernel, *ACM Transaction on Computer Systems*, Vol. 4, No. 2, pp. 110-129 (1986).
- 6) Ahuja, S., Carriero, N., Gelernter, D. and Krishnaswamy, V.: Matching Language and Hardware for Parallel Computation in the Linda Machine, *IEEE Transaction on Computers*, Vol. 37, No. 8, pp. 921-929 (1988).
- 7) Xu, A. and Liskov, B.: A Design for a Fault Tolerant, Distributed Implementation of Linda, *Annual International Symposium Fault Tolerant Computer System*, pp. 199-206 (1989).
- 8) Birman, K. and Joseph, T.: Reliable Communication in the Presence of Failures, *ACM Transaction on Computer Systems*, Vol. 5, No. 1, pp. 47-76 (1987).
- 9) Mowry, T., Lam, M. and Gupta, A.: Design and Evaluation of a Compiler Algorithm for Prefetching, *ASPLOS V*, ACM, Oct. (1992).
- 10) Tanenbaum, A., Kaashoek, M. and Bal, H.: Parallel Programming Using Shared Objects and Broadcasting, *IEEE Computer*, Vol. 25, No. 8, pp. 10-19 (1992).

(平成5年9月24日受付)

(平成6年2月17日採録)



松田 哲史 (正会員)

1962年生。1985年東京大学工学部電気工学科卒業。1987年同工学系研究科電子工学専攻修士課程修了。同年住友電気工業(株)入社。1990年より2年間、米国スタンフォード大学客員研究員。通信ソフトウェア、分散並列処理に関する研究開発に従事。ACM 会員。



武並 佳則 (正会員)

1964年生。1988年早稲田大学理工学部電気工学科卒業。1990年同大学院理工学研究科電気工学専攻修士課程修了。同年住友電気工業(株)入社。並列処理、オブジェクト指向設計、制御系設計に関する研究開発に従事。



田口 哲也 (正会員)

1955 年生. 1978 年大阪大学工学部電子工学科卒業. 1980 年同大学院工学研究科電子工学専攻修士課程修了. 同年住友電気工業(株)入社. 光通信システム, サーボ制御, 画像処

理システム, マルチプロセッサおよび並列処理システムの研究開発に従事.
