計算機アーキテクチャ 101−15
（1993. 8. 20）

キャッシュインジェクションとメモリベース同期機構の高速化

松本 尚　　平木 敬

東京大学 理学部 情報科学科
〒113 東京都文京区本郷 7-3-1

プロセッサやコントローラが他のキャッシュに投機的にデータを注入するキャッシュインジェクションの概念を定義し、その有効性について議論する。メモリ上のデータを伴って他のプロセッサに処理を依頼したい場面において非常に有効な手法である。また、allread 系のプロトコルもキャッシュインジェクション動作を伴うキャッシュプロトコルだと見做せ、データのトラフィック削減効果が得られる。さらに、インジェクション可能なキャッシュを利用して、演算プロセッサとアクセスプロセッサを結合する Decoupled Archicture が構成できる。次に、メモリベース同期機構の利点およびその高速化方針について述べ、現在開発中の超並列計算機 D-machine のアーキテクチャに基づいて、メモリベース同期機構の実装方式について説明する。そして、キャッシュを利用してこの同期機構を高速化する方式について述べる。最後に、キャッシュインジェクションおよびメモリベース同期機構の利用法の典型例を示す。

Cache Injection and High-Performance Memory-Based Synchronization Mechanisms

Takashi Matsumoto　　Kei Hiraki

Department of Information Science, The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan.
e-mail: {tm, hiraki}@is.s.u-tokyo.ac.jp

In this paper, we propose the concept of **Cache Injection**. Cache injection is an action of assigning data into processors' cache by an external element. To define generally, the initiator of data transmission can arbitrarily specify multiple caches as targets of the cache injection. Cache injection technique is useful for implementing various basic mechanisms used in parallel processing systems such as a light message-passing, a latency hiding/reduction by decoupled-architecture approach, an efficient macro-dataflow execution using conventional microprocessors. Then, we describe the merits of **Memory-Based Synchronization** mechanisms and the strategies for their performance improvements. Implementation methods of the proposed mechanisms on the **D-machine** (tentative name) of the Japan University Massively Parallel Processing project are described. The performance of memory-based synchronization mechanisms can be improved by the caching technique with some special treatments, and the methods are presented. Finally, application examples of cache injection and memory-based synchronization are discussed.

1 はじめに

単一プロセッサシステムにおいて、メモリアクセスの空間的局所性および時間的局所性を利用した完全自動高速化を意図して、キャッシュが利用されてきた。このため、コンパイラ等による静的な解析とは無関係に機能する存在であった。しかし、マルチプロセッサシステムにおいては、高性能化のために、静的に解析されたメモリアクセスのタイプによって、キャッシュのプロトコルを動的に切替えることの有効性が唱えられるようになってきた[1, 2]。歴史的にはマルチプロセッサの場合、キャッシュと主記憶間のデータ転送路（共有バスや多段結合網）の転送量を抑えるためにライトバックを基本とした様々なコンシステンシプロトコルが考案された。初期の頃は、プロセッサ間で密に協調動作を行わせるような並列アプリケーションが開発されなかったため、多くのマルチプロセッサでinvaldate系のコンシステンシプロトコルを持つキャッシュが採用された[3, 4, 5]。これらは、書き込みを行う時は該当キャッシュブロックのコピーが一つしか存在しないことを保証して、コンシステンシを維持していた。このため、更新された値を他のプロセッサが利用する場合は、更新を行ったプロセッサのキャッシュから最新の値をメモリ参照時に持ってくる必要があり、レイテンシが大きくなる。この欠点を克服するために、update系のコンシステンシプロトコルがinvalidate系のプロトコルに引き続いて考案された[6, 7]。書き込みを行う際に、該当キャッシュブロックが他のキャッシュに共有されている場合は、コピーを持っているキャッシュの内容をすべて最新の値に更新する方式である。確かに、update方式では、密にプロセッサ間でデータ通信が発生する場合は、そのデータのメモリロケーションが過去に参照された場所であれば、キャッシュ上に最新の値が存在するため、メモリの参照がキャッシュまでで解決し、レイテンシが小さい。しかし、前述したように複数のキャッシュを持ったマルチプロセッサが開発された当初は、並列用オペレーティングシステムの不備（細粒度の並列処理が実行できる環境を提供しない）も手伝って、プロセッサ間で密に協調並列動作を行うアプリケーションプログラムが開発されなかった。このため、キャッシュ間で共有されるべきデータは極めて少なく、invalidate系のプロトコルの方が不必要な共有を早く解消できるので、データのトラフィックも少なく性能も優れていた。近年、並列論理型言語や並列オブジェクト指向型言語といった粒度の小さなプログラムの作成が容易な言語が広がりを見せている。これらの言語によるアプリケーションを効率良く実行するため、また高並列計算機や超並列計算機の並列度を充足する目的のために、より粒度の細かい並列処理が行える、つまりプロセッサ間通信同期が効率良く頻繁に行える、並列アーキテクチャが求められるようになってきている。

プロセッサ間で同時期に共有使用されるデータのメモリ領域に関してはupdate系のプロトコル、ローカルに使用されるメモリ領域に関してはinvalidate系のプロトコルを割り付けるのがデータトラフィックおよびレイテンシの削減上望ましい[1]。メモリ領域がプロセッサ間に跨って存在すべきかどうかの区別、つまりプログラムの変数やデータ領域がプロセッサ間の通信同期に用いられるか、プロセッサローカルな存在であるかの区別は、共有メモリモデルの手続き型言語を使用するプログラマにとっては当然意識すべき重要事項である。また、並列言語や通常言語から自動的に並列実行されるコードが生成される場合には、コンパイラ内で明確にこれらの変数を区別して扱われなければ、並列実行コードの生成はできない。このようにプログラマが明示的に共有メモリアクセスを記述する場合でも、コンパイラが自動的にメモリアクセスコードを生成する場合でも、変数やデータ領域ごとにupdateプロトコルが適しているか、invalidateプロトコルが適しているか決定することは比較的容易である。

この静的なデータのアクセスタイプの情報の利用をさらに押し進めたものとして、allread系とallwrite系のプロトコル（両者を組み合わせたプロトコルも考えられる）が提案された[8, 2]。これらは、協調動作しているプロセッサのグループを考え、読み出し（allread系）または書き込み（allwrite系）時に、キャッシュと主記憶の間でデータ転送が行われる際には、積極的にそのデータをグループ内の他のキャッシュにもマルチキャストして送りつけるプロトコルである。マルチキャストと言ってもデータ転送路が共有バスである場合、他のキャッシュは単にスヌーピングでバスを監視して自分のキャッシュメモリにデータを取り込むだけである。データをマルチキャストで送り付けられたキャッシュは、たとえ当該データのキャッシュエントリが存在しなくても、ライトバック動作を起こさずにデータを受けとれる場合にはキャッシュエントリを確保してデータを格納する。共有バス結合のマルチプロセッサでは同時期に複数のプロセッサで参照すべきデータが一回のバス使用ですべてのキャッシュに格納でき、バストラフィックが削減できる。eagerなデータ割り付けによって、キャッシュのヒット率が高まり、メモリアクセス時のレイテンシが小さくなることも期待できる。このeagerなキャッシュへのデータ割り付けを伴うallread系プロトコルの効果について、DOACROSSループの並列実行に対する性能向上効果が大きいことがシミュレーションによって確認された[9, 10]。また、DOALLループについても性能アップが確認された[11]。

2 キャッシュインジェクション

2.1 キャッシュインジェクションの定義

allread系プロトコルやallwrite系プロトコルで実現されるような外部要因によるキャッシュへのeagerなデータ注入をキャッシュインジェクションと呼ぶことにする。allread/allwrite系のキャッシュプロトコルを使用するInter-Cache Snoop Contorol Mechanism (ICSCM)ではハードウェア量を考慮して、データの受けとり事前に定められたグループ単位の指定がなされることを仮定していた。より一般的に定義するために、キャッシュインジェクションという概念では、データを転送する主体はデータを注入するキャッシュを転送ごとに任意の組み合わせで指定できるものとする。

キャッシュインジェクションと相互に補間する概念としてキャッシュへのプリフェッチがある。キャッシュへのプリフェッチはプロセッサ自らが前持って自分の使用するデータ領域を自分のキャッシュ上にフェッチする操作を行い、データがメモリシステムから返送される間には他の演算を実行し、データを使用する時にはキャッシュとレジスタ間でのデータ転送を済ませてレイテンシを削減する手法である。これに対して、キャッシュインジェクションはプロセッサ(またはその附属キャッシュ)またはDMAコントローラが他のプロセッサに附属したキャッシュにデータを外部から注入する操作を指す。

該当データのエントリがターゲットのキャッシュに存在する場合、ターゲットのキャッシュ側の動作はupdateプロトコルと同じである[1]。しかし、エントリが存在しない場合でも、エントリを新たに確保してデータがキャッシュ内に注入される点がupdateプロトコルと根本的に異なる。また、allreadプロトコルやallwriteプロトコルでは、キャッシュリードミスやライトスルー時のメモリトランザクションを他のプロセッサに附属したキャッシュにも反映させていた。キャッシュインジェクションの概念は、もっと一

[1] 厳密に言えば、差が存在する。書き込みによるupdateトランザクションはキャッシュのブロック（ライン）単位ではなく、ブロック中の書き換えた部分のみ（ワード単位やバイト単位）でもよい。しかし、キャッシュインジェクションは予めエントリがない場合も想定しているので、常にブロック（もしくはその整数倍）単位のトランザクションとなる。

般的に自分のキャッシュの動作と関係なく、データを他のキャッシュに注入する動作そのものを指す。つまり、allread や allwrite はキャッシュインジェクション動作を伴うコンシステンシプロトコルの分類の名称である。

また、プロセス切替やスケジューリングでプロセッサの割当が変わる場面でも、切替え当初は若干の性能低下はある可能性があるが、キャッシュへのデータ注入であるため新たなコンテクストの退避等のオーバヘッドが生じない。また、コンシステンシキャッシュを対象としているので、投機的注入（speculative injection: キャッシュプリフェッチの speculative load に相当）時に同期を取る必要がなく、プログラムの任意時点で投機的注入が可能である。

2.2 インジェクション可能キャッシュの実装方式

高性能なコヒーレントキャッシュを作るためには、アドレスタグ（物理アドレスの上位を格納するタグ）を二重化（プロセッサアクセス側と外部アクセス側の2セットを用意）して、コントロールタグ（キャッシュブロックの状態タグ）をデュアルポート化する必要がある。キャッシュインジェクションにおいては外部からのデータをキャッシュに格納するため、キャッシュのデータ部がシングルポートのメモリで構成される場合、データパスがプロセッサからのアクセスと外部からのアクセスで競合する可能性がある。また、アドレスタグが外部からのトランザクションでも変化するので二重化ではなく、デュアルポート化する必要がある。キャッシュのデータ部に関しては update 系のプロトコルやキャッシュへのプリフェッチをサポートすれば、同様の問題が起こり、外部からのトランザクションを優先するか、デュアルポートメモリを使用する必要がある[2]。つまり、キャッシュのデータ部に関して、キャッシュインジェクションのインプリメントは update 系プロトコルのインプリメント時以上の問題を引き起こすものではない。アドレスタグのデュアルポート化に関しては、元々二重化する必要のある部分であり、ハードウェア量としては大差がない。

キャッシュインジェクション時に dirty なブロック（主記憶への書き戻しが必要なブロック）しか該当セットに存在しない場合は、ハードウェア量の増大とコントロールの複雑化を避けるために、eager な割り付けを諦めるというアプローチは設計上のトレードオフであり、現実的な選択であると考えられる。また、キャッシュインジェクションのターゲットキャッシュの指定は、ハードウェア量を考えると、一個の特定のキャッシュを指定する方式とある特定のグループに属するキャッシュ群を指定する方式の二通り程度に絞ることが同様に現実的なトレードオフだと思われる。

外部からの新しいエントリのインジェクションによるキャッシュリプレースが現在実行中のスレッド（またはプロセス）の動作を大幅に妨げることのないように、キャッシュはダイレクトマップ方式ではなく、複数のセットを持ったセットアソシアティブ方式であることが望ましい。ダイレクトマップのキャッシュを使用して大幅な性能降下の可能性を排除するためには、インジェクションが行われるアドレス領域とそれ以外のアドレス領域がキャッシュ上で重ならないようにアドレス（ページ）管理を行う必要がある。またインジェクションの領域も二分割（またはそれ以上に分割）して、フレームバッファのダブルバッファリングと同じ形式で使用することが望ましい。

[2]ただし、デュアルポート化されていても同一ブロックへタグの書換を伴うようなアクセスが競合した場合は、外部アクセスが優先される。

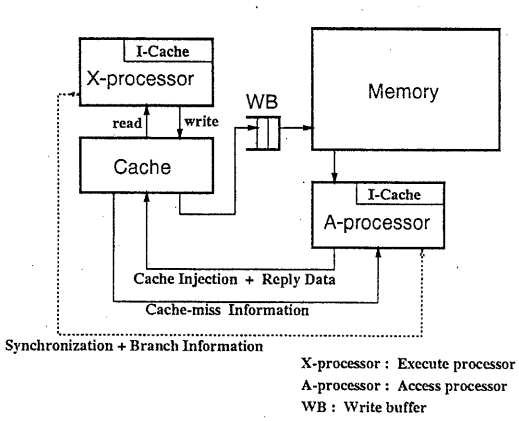

図 1: キャッシュを使った Decoupled Architeture

2.3 軽いプロセッサ間データ通信

ヒープ上にとった領域への計算結果を他のプロセッサに渡して処理を継続させるような場合は、処理アドレスが動的に決まるため、プリフェッチ技術を使ったレイテンシの削減を効果的に行うことができない。キャッシュインジェクションを用いれば、依頼先のプロセッサのキャッシュに計算結果をインジェクションしてから、そのプロセッサに処理を依頼すれば、依頼されたプロセッサはキャッシュミスを起こさずに処理を効率良く実行できる。インジェクションを行う主体側にとっては、インジェクションの動作はメモリへの書き込み動作と同じなので、ライトバッファがあれば、インジェクション動作はすぐに終了する。また、大量のデータをインジェクションする場合は、プロセッサ自体で転送を行わず、DMA を起動すればよい。そして、インジェクションが済むまでは、依頼相手の（インジェクションされる側の）プロセッサは他の処理を実行（または続行）する。

2.4 キャッシュを使った Decoupled Architecture

メモリへのオペランドフェッチのアクセスレイテンシを隠蔽するために、インジェクションが可能なキャッシュを用いて Decoupled Architecture [12] を構成することができる。つまり、演算プロセッサとアクセスプロセッサを本来の Decoupled Architecture のようにハードウェアキューで結合するのではなく、キャッシュを使って結合させる（図1参照）。アクセスプロセッサは演算プロセッサのキャッシュにデータを注入し、演算プロセッサはキャッシュに対してロード命令を実行する[3]。データの受け渡しがキャッシュを介しているため、従来の Decoupled Architecture の重大欠陥であったメモリシステムからのデータ到着順序の遵守の制約がない。また、演算プロセッサとして市販の高性能マイクロプロセッサを改造することなしに使用することができ[4]、コストパフォーマンスの点で非常に有利である。

Decoupled Architecture 以外のメモリアクセスレイテンシの隠蔽法としてキャッシュへのプリフェッチを用いた方法が多く提案されている [13, 14, 15]。しかし、プリフェッチの方式では、高速動作が要求されるプロセッサ単位のキャッシュに対して、lookup-free（または non-blocking）キャッシュであることが要求され、

[3]条件ブランチに関する情報の受け渡しはキューを介して行っても構わない。また、書き込みのアドレス計算は簡単な計算であれば、演算プロセッサ側で行われる。

[4]インジェクション不可能な内蔵キャッシュを持つ市販プロセッサに対しては、外づけの二次キャッシュへのインジェクションを想定している。

高速キャッシュの実装が困難である。これに対してキャッシュインジェクションを利用したDecoupled Architectureでは、演算プロセッサ用キャッシュとしてlookup-freeキャッシュを構成する必要はない。また、キャッシュへのプリフェッチによるレイテンシ隠蔽では、自由度を保つためにプロセッサがプリフェッチ命令を明示的に実行して、外部（メモリシステム）にプリフェッチを指示するタイプのものが多い。このタイプの場合には、プリフェッチ命令の実行によるオーバヘッドが生じる。

アクセスレイテンシ隠蔽方式として、ソフトウェア技法であるloop unrollingやload piplining(またはphase piplining)をプログラムのループ部分に施して、メインメモリからデータをレジスタに直接先行loadする方式も提案されている[16, 17, 18]。確かに、この方式ではキャッシュに対してアクセスプロセッサからのデータ注入と演算プロセッサからのデータアクセスを性能低下を引き起こすことなく両立させる必要がなく、ハードウェアコストが安くなる可能性がある[5]。しかし、オペランドアクセスに時間的な局所性がある場合は、キャッシュを利用した方がデータのトラフィックの削減の観点で圧倒的に有利である。また、時間的局所性がなくても、イテレーションごとに連続した配列要素を使用する場合、キャッシュ方式では転送単位のブロックサイズがレジスタサイズよりも大きいことを利用して、バースト転送によってデータ転送の効率を向上させることができる。キャッシュを使用した場合、ストライドアクセス時に無駄なトラフィックが頻繁に起こるという批判があるが、キャッシュインジェクションによるDecoupled Architectureでアクセスプロセッサがメインメモリ側に存在すれば（図1参照）、アクセスプロセッサの段階でキャッシュのブロック単位にデータをパックし直して転送する手法が可能である。また、サイクリックなイテレーション割り当て等によってデータのfalse sharingが発生する場合であれば、allreadプロトコルによってデータ転送効率を損なわずにプログラム実行が可能である（バースト転送の分だけ逆に効率が向上する）[6]。さらに、メモリ間接参照等の複雑なアクセスパターンの場合はアクセスプロセッサがメモリ参照のための計算を演算プロセッサから独立して行うことで処理効率の向上が可能である。

3 メモリベース同期機構

3.1 メモリシステムへの同期構造の導入

並列フォンノイマン型の計算機であっても、コヒーレントキャッシュを持つメモリシステム有していれば、シングルライタ・マルチリーダの生産者消費者型の同期機構の実現が、full/emptyビットをデータの一部としてメモリロケーションごと（キャッシュ内を含む）に付加することで容易に可能である[8, 2]。同期待ち状態の実現にはプロセッサをサスペンドさせても、ビジーウェイトさせてもよい[7]。いずれにせよ、性能を向上させるためには各キャッシュ上で生産者による書き込みが終了したこと（fullになったこと）が判定できる必要がある。キャッシュコンシステンシが維持されていれば、キャッシュコピーがどのプロセッサに存在しても、各々のキャッシュはキャッシュブロックへの書き込みを検知できるので、正しくfull/emptyの状態を管理できる。このとき、同期情報の通信がデータの通信と不可分に行われ（データ駆動同期が実現される）、データ通信と同期通信の順序関係の制御が必要なく、効率が非常に良い。キャッシュコンシステンシが維持されていれば、この同期機構の動作は保証されるが、キャッシュプロトコルによって性能（データ転送路の使用回数とレイテンシ）は大幅に異なる。細粒度の生産者消費者の同期であれば、生産直後にデータが消費されるため、eagerなキャッシュへのデータの割り付けを伴うプロトコルが性能上有利である。事実、共有バス型マルチプロセッサ上のDOACROSSループについてのシミュレーションではallread系プロトコル使用時の性能が最も優れていた[9, 10]。

3.2 メモリベース同期機構の利点

full/emptyビットによる同期機構のメモリシステムへの導入についてはすでに述べた。この場合、生産者消費者型の同期を効率良くサポートすることができた。しかし、同期機構としてはもっと多くの種類の機構が考えられる。例えば、FIFO機構やバリアや排他制御やセマフォや不可分命令（fetch&add等）が挙げられる。これらの機構が付加ハードウェアとして場合、実プロセッサ単位で従来のシステムには付加されてきた。しかし、マルチジョブやマルチユーザの環境では、プロセッサ単位の付加ハードウェアは非常に使い方が困難で、使用したとしても大きなオーバヘッドを伴う。例えば、プロセッサ単位にFIFOを設けたとして、このFIFOを介して通信を行ったとすると、受け取り側は複数のジョブへのメッセージから自分のジョブへのメッセージを探し出して読み出す必要がある。また、FIFO用のバッファの管理も複数のジョブへのメッセージが混在するため非常に複雑になる。さらに、単一ジョブ内ののマルチスレッド処理であっても、メッセージの対象スレッドを同定するために、かなりのソフトウェアオーバヘッドがかかる。これらの場合、実用上はFIFOバッファの管理サーバー（または管理用スレッド）のようなものを各プロセッサで動かし、各ジョブ（スレッド）ごとのFIFOバッファに転送し直す処理が必要となる。これに対して、メモリベースのFIFO機構を構築した場合、受け取り側がジョブ（スレッド）ごとにアドレスを識別子としたFIFOを設定できるので、送り手側は直接目的のジョブ（スレッド）がFIFOを設定しているアドレスにメッセージを送れば良い。この場合、FIFOバッファの管理サーバー（管理スレッド）は不必要であり、オーバヘッドが大幅に削減できる。

3.3 メモリベース同期機構の高速化

要素プロセッサの数が多くなって、大規模分散共有メモリが構築される場合、同期実行時のレイテンシを隠すためには、同期を同期信号の発信フェーズと受信フェーズに分け、発信フェーズはノンブロッキングの処理であり、受信側で可能なものはついてはバッファリングを行う。そして、受信側でのバッファは受信側のプログラムが実行される実プロセッサに物理的に近い位置に設けられることが望ましい。前記のメモリベースのFIFO機構を例に挙げて説明する。メッセージが物理的に伝送される時間を短縮するのはアーキテクチャ的な工夫では不可能であるため、この時間は他の演算を行ってプロセッサを浪費しないことで、性能を高めるしか方法がない。メッセージの送信側のプロセッサはFIFOにバッファライトを行うだけで、すぐに次の処理を開始することで処理性能を維持する。メモリベースのFIFO機構であれば、自動的に特定のアドレスを識別子とするバッファ上にデータが格納される。受信側のプロセッサはそのアドレスからデータを読み出して使用すれば良い。しかし、この時にバッファがこのプロセッサから物理的に離れたところに形成されていた場合、データの読み出しに遅延が発生してしまう。そこで、このバッファはできるだけ受信側のプロセッサの近くに形成される必要がある。

[5] キャッシュは不要もしくは簡単な構造で済むが、プロセッサ内に必要なレジスタ数は増大する。

[6] 共有バスやリング以外のプロセッサ-メモリ間ネットワークの場合は、allreadプロトコルはデータが返送される経路途中にあるキャッシュのみをallreadの対象とする。

[7] 同期待ちが長時間になることが予め予期される状況ではプロセッサのコンテクストを切替えてもよい

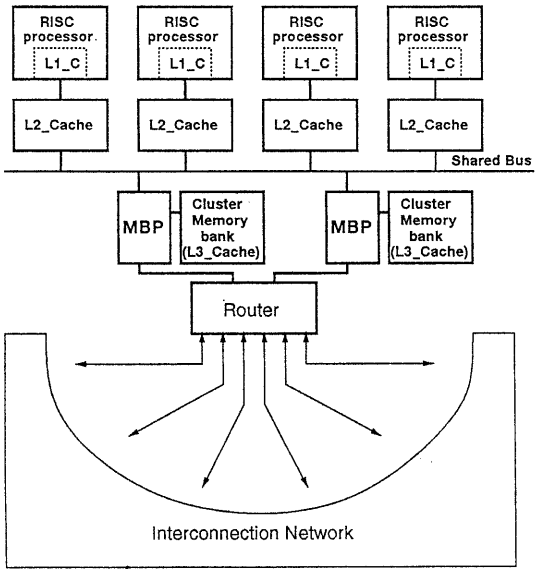

図 2: D-machine のクラスタ構成

3.4 メモリベースの同期機構の実装方式

具体的な議論を行うために、議論の土台となる超並列計算機システムを簡単に説明する。次に、そのシステムで用いられる同期ビット付きのメモリシステムについて説明し、メモリベース同期機構を構成するためのメモリアクセス手段について概説する。

3.4.1 超並列計算機 D-machine (仮称)

以後の議論は現在開発中の超並列計算機 D-machine (仮称) に基づいてなされる。D-machine はクラスタ化された分散共有メモリ型並列計算機であり、Memory-Based Processor (MBP) [19, 20] を使った高性能メモリシステム (Strategic Memory System: SMS) [20, 21] を最大の特徴としている。クラスタ間接続は慶応大学の天野らによる RDT[22] と呼ばれる高性能静的結合網 (ただし、各ノードにおいて接続枝間の hardware-through routing 機能を持つ) によってなされる。図 2に D-machine のクラスタ構成を示す。 図中に示したように、MBP は主記憶 (分散共有メモリ) のメモリバンク (ページ単位でインタリーブ) 毎に配置され、メモリへの要求やメモリ関連の操作を処理する。主な処理内容としては、メモリ管理、メモリ間のデータ転送、メモリ上の不可分操作、コンシステンシ管理、メモリ上の同期操作、スレッドの低レベル管理が挙げられる。MBP は担当するメモリへの要素プロセッサからの要求を自分へのメッセージとして解釈するメッセージベース短命令流処理プロセッサであり、複数の MBP で並列処理が行われるのみならず、単一の MBP の中でもハードウェア資源の許す限り複数のメッセージが同時に処理される (マルチスレデッドアーキテクチャ)。MBP 用のプログラムコードはその MBP の担当メモリバンク内に存在するが、コンシステンシ管理やクラスタ内のプロセッサからのメモリへの単純な load/store といった効率上重要な機能は hardwired logic で実現されている。要素プロセッサからのリモートメモリアクセスを伴う処理要求はクラスタ内の MBP を経由して、ネットワークを介した MBP 間のメッセージの交換で実現される。

3.4.2 D-machine のメモリシステム (SMS)

高機能かつ高性能なメモリベース同期機構 (Memory-Based Synchronization) の実現のために、ワードごとに同期ビットが付加された Reactive Memory (RM) と呼ばれるメモリシステムが使用されている。ネーミングを整理すると、MBP-Core (MBP 内のプログラムによる処理機構) と RM を組み合わせることで、各種のメモリベース同期機構が実現される。また、RM の概念上は、同期ビットの付加されたメインメモリとキャッシュメモリ、そしてそれら操作のために MBP とキャッシュコントローラ内の hardwired logic で実現される機能 (同期ビットのチェックおよび操作機能やプロセッサからのメモリ要求に伴うメモリアクセス機能) と MBP-Core への処理起動機能が含まれる。このように RM と MBP では名称の指し示す範囲にオーバラップがある。さらに、キャッシュプロトコル切替え (ICPCM) と Pseudo-Fullmap 方式 [21] による分散共有メモリ機能と前述のキャッシュインジェクション機能 (CIF: Cache Injection Facility) を MBP と RM に加えたメモリシステム全体を SMS (Starategic Memory System) と呼ぶ。RM がメモリロケーションごとの同期ビットを利用して同期処理を行い、hardwired logic のみでは実現の困難な高機能メモリベース同期機構に対しては MBP-Core のプログラムを起動する。この機能により、FIFO の管理や不可分操作といった高機能な同期処理がメインプロセッサを使用することなしに主記憶側で実現され、メインメモリの操作コストの差によりメインプロセッサで実現するよりも効率が高い。MBP-Core のプログラムは主記憶の担当メモリバンク内に作業領域や FIFO バッファを形成する。

3.4.3 Reactive Memory (RM)

同期ビットが付加されたメモリシステムである Reactive Memory (RM) は I-structure[23] をベースとし、コヒーレントキャッシュと MBP の存在および要素プロセッサがノイマン型であることを考慮に入れた修正と高機能化がなされている。

本節では説明の簡便さのために主記憶にのみに同期ビットが付いているものと仮定する。つまり、同期ビットの操作やチェックを必要とする処理はすべて主記憶上で行われ、同期ビットを利用するデータはキャッシュされないページ属性を持つ。次節において、キャッシュを利用した高性能化について論じる。

同期ビットとして full/empty を表す 1bit と、non-wait / R-wait / W-wait の MBP-Core によるサービス待ち状態を表す 2bit の計 3 ビットがワードごとに付加されている。R-wait は読み出しで MBP-Core のプログラム起動が必要なことを示し、W-wait は書き込みで MBP-Core の起動が必要なことを示す。ただし、効率を上げるためにメモリバンク内で同時に複数のメモリトランザクションを許す場合は、該当ロケーションの処理中で内容がロックされていることを示すビット等も必要になる。full/empty ビット以外の同期ビットは若干の性能低下を厭わなければ、ブロック単位に設けても構わない。

SMS へのアクセスはすべて MBP へのメッセージであるという立場を採用しているため、メモリアクセス時に実行する同期操作の区別は MBP へのメッセージ内のコマンドで区別される[8]。RM (hardwired logic による同期機能実現) レベルでは、高機能同期が直接実行されるわけではない。RM は単に MBP-Core への処理依頼の要・不要および要の場合の起動時期 (アクセス終了前/終了後) をアクセスメッセージのコマンド (書き込み/読み出

[8] MBP へのインタフェースを持った要素プロセッサではこのコマンドをメモリアクセス時にのアドレスとデータの他に出力する。このようなインタフェースを持たないマイクロプロセッサではアドレス出力の上位ビットと MBP 内の要素プロセッサごとのコマンド用レジスタと MBP 内の TLB の属性フィールドを使用して、コマンドを指定する。

しの区別を含む）と操作対象のメモリロケーションの同期ビットから判断する。アクセス終了後の MBP-Core への処理依頼というのは、メインプロセッサによるメモリアクセス自体は MBP-Core の処理終了を待たずに、ノンブロッキングに終了することを意味する。アクセス終了後の処理依頼と処理依頼不要の場合はアクセスメッセージのコマンドと現在の同期ビットの状態から、同期ビットの状態遷移を求めて、同期ビットを更新してプロセッサのアクセスを終了させる[9]。アクセス終了前の処理依頼の場合は、アクセス対象のメモリローケーションの更新も含めてメモリ操作は MBP-Core がすべて行い、MBP-Core がプログラムで明示的にメインプロセッサの処理を終了させる。終了前または終了後にかかわらず、MBP-Core へ処理依頼を行う場合はアクセスメッセージのコマンドと現在の同期ビットの状態から、MBP-Core に対して起動するプログラムが決定され、そのインストラクションポインタおよびその処理の起動を引き起こしたメインプロセッサのアクセス情報（メモリアドレスや書き込みデータや現在の同期ビットの値）が処理起動メッセージとして MBP-Core に送られる。

3.4.4 RM の基本メモリアクセス手段

前記の RM へのアクセス手段として read, write, s-read, s-write, q-read, q-write, t-read, t-write, f-read, p-read, s-enqueue, q-enqueue, b-rreq, b-aprv, b-preq 等が提供される[10]。これらについて簡単に概説する。詳しくは文献 [20] を参照されたい。

同期ビットを無視する読み出し / 書き込みは read/write であり、同期ビットは変化しない。full/empty ビットの状態で生産者 - 消費者の通信と同期を同時に実現するアクセス法が s-read/s-write である。FIFO キュー（Memory-Based FIFO）をブロックごとに構成するアクセス法が q-read/q-write である。

t-read/t-write はユーザがカスタムメードのメモリベースの同期機構を作るためのアクセス手段で、q-read/q-write と似ているが、ブロック単位ではなくワード単位であり、W-wait と R-wait 状態の動作が異なる。R-wait で t-read 状態の場合と W-wait で t-write 状態の場合は、該当ワードもしくはコマンドレジスタの内容をインストラクションポインタとして MBP-Core のユーザプログラムを呼び出す。

f-read は該当ロケーションの full/empty ビットの状態を読み出すアクセス手段で、full 状態で 1 を返し、empty 状態で 0 を返す。p-read は q-read とほぼ同じであるが、該当ロケーションの状態が empty の場合、返り値として 0（ヌルポインタ）を返し、サスペンドや割り込みを起こさない。s-enqueue は該当ロケーションで s-write 待ちのコンテクストを登録するアクセス手段であり、q-enqueue は該当ロケーションで q-write 待ちのコンテクストを登録するアクセス手段であり、共にコンテクストのセーブエリアへのポインタ（論理アドレス）をデータとして MBP-Core に引き渡す。

b-rreq, b-aprv, b-preq は論理的なメモリベースのバリア機構を構成するためのアクセス手段であり、本来プロセッサ間のハードウェアバリアである Elastic Barrier[1, 24] や Fuzzy Barrier[25] を低コストでメモリ上に仮想化して実現するためのものである。厳密には、これらのアクセス手段はすべての状況で MBP-Core のプログラムの起動を要求するので、RM へのアクセス手段ではなく、もっと広い意味の MBP へのメッセージもしくは SMS へのアクセス手段である。

[9]アクセスの対象となっているメモリロケーション以外を操作する必要がある場合は必ず MBP-Core に対する処理依頼が行われる。

[10]他にもキャッシュのプロトコルによるアクセス手段の分類が存在する [21]。

3.5 キャッシュを利用した高速化

前節ではメモリベースの同期機構に対しては、プロセッサのキャッシュを使用しない立場で説明を行った。しかし、例えば Memory-Based FIFO の場合に、受信側のレイテンシを極限まで削減するためには、受信のためのプログラムが実行されているプロセッサのキャッシュに、FIFO バッファの先頭のデータのコピーを作成すればよい。そこで、本節ではキャッシュを利用したメモリベースの同期機構の高性能化法について述べる。

基本的に full/empty ビットのみを付加したメモリシステム [8, 2] の場合と同様に、プロセッサのキャッシュ側にも同期ビットを付加する必要がある。主記憶と同じく full/empty を表す 1bit と、non-wait / R-wait / W-wait を表す 2bit の計 3bit の同期ビットがキャッシュのワードごとに付加され、これらの同期ビットはキャッシュブロックのフェッチの際にデータワードの一部として扱われる。ただし、主記憶が full/empty ビット以外の同期ビットをブロック単位に設ける場合は、キャッシュ側もそれに従う。この拡張を行ってメモリベースの同期機構に関してもキャッシングを行えば、s-read, q-read, p-read, f-read, b-rreq の読み出し系のアクセス手段はキャッシュの段階で値（結果）を得ることができ、レイテンシとデータトラフィック（特に、ビジーウェイト時の p-read, f-read, b-rreq のデータトラフィック）を削減することが出来る。但し、同期ビットの状態によってキャッシュの動作とプロトコルを制御する必要がある。特に、FIFO を構成している場合は、FIFO の先頭のデータのキャッシュコピーがメモリシステム内に一つしか存在しないことを保証する必要がある。また、キャッシュ内にバッファ等を設けるのは、キャッシュの高速実装上非常に困難であるため、バッファ操作やリスト操作を伴うアクセス手段はすべて主記憶の MBP に伝達される必要がある。例えば、p-read でキャッシュにヒットして full かつ R-wait 状態であれば、プロセッサにはキャッシュ内のデータを返送しプロセッサのメモリトランザクションは終了させる。その一方、ワードの状態を empty に変更し、主記憶に当該ワードが一つ読み出されたことを通知する（p-read を伝達する）必要がある。主記憶は R-wait であるので、データリストの先頭のデータをワードにセットする。キャッシュの当該ワードは新しいデータに更新され、それに伴って同期ビットの状態は主記憶の状態と同じになる。

アクセス手段ごとのキャッシュ制御方法を以下に説明する。

s-read キャッシュ管理法は通常の read と同じ。empty 状態の読み出しは遅延され、プロセッサがブロックされるか割り込みが発生する。

s-write キャッシュ管理法は通常の write と同じ。同期ビットの empty 状態から full 状態への変更をキャッシュ内で行う。ライトバックポリシも使用できる。

q-read キャッシュコピーが一つしか存在しないプロトコルを使用する必要がある。つまり、主記憶からキャッシュへのフェッチにおいて、排他的な読み出しを行わなければならない。q-read では empty 状態の読み出しは遅延され、プロセッサがブロックされるか割り込みが発生する。q-read ではデータを更新する必要があるため、キャッシュで full 状態のワードに q-read がヒットした場合、そのデータをプロセッサに返送し、キャッシュ内の該当ワードの同期ビットを empty 状態にする。該当ワードが R-wait 状態であれば、q-read のメッセージを主記憶に返して（キャッシュのライトバック操作の一種と考えられる）、該当ブロックを無効化（または同期ビットを empty&non-wait に）する。つまり、empty かつ R-wait の状態は次のデータを主記憶にリクエストする過渡的な同期ビットの状態である。主記憶側は読み出し待ち

のデータがあった場合、FIFOの先頭にそのデータ（と適正な同期ビット）をセットして、q-readがライトバックされてきたキャッシュにブロックをインジェクションする（キャッシュエントリが無効化されていなければ、コンシステンシ維持動作）。

q-write, s-enqueue, q-enqueue, b-aprv, b-preq これらのアクセスは主記憶の該当アドレスに送り付けられなければならないので、基本的にキャッシュに影響しない。つまり、たとえ該当アドレスをキャッシュしているプロセッサがこれらのアクセスを行ったとしても、キャッシュには影響を与えずに、アクセスメッセージが主記憶に発行される。そして、エントリアドレスのデータ（FIFOの場合は先頭データ）または同期タグが変更された場合のみ、コンシステンシ機構によって、キャッシュの内容が更新（または無効化）される。

t-read, t-write MBP-Coreでユーザプログラムを起動するのが目的なので、基本的にnon-cacheableで扱われる。（q-read, q-writeと同じ扱いも可。）

f-read キャッシュ管理ポリシは通常のreadと同じ。ただし、前述のようにデータを返すのではなく、同期ビットがfull状態であれば1をempty状態であれば0をプロセッサに返答する。

p-read キャッシュ管理ポリシはq-readと同じ。ただし、empty状態の場合、q-readは読み出しを遅延するが、p-readでは0（null）をデータとしてプロセッサに返す。

b-rreq メモリベースのバリアの管理は基本的に主記憶上で行われるが、b-rreqに関してはキャッシュコピーが存在すれば、キャッシュにおいてバリア成立不成立の判断が可能になり、高速化が可能である。キャッシュ管理ポリシはq-readとほぼ同じ。キャッシュで複雑な処理を行うことは、キャッシュの速度低下を意味するので避けるべきである。そこで、バリア構造体内の同期成立カウンタの値が0より大きい場合に、キャッシュのエントリをfull&R-waitにし、0の場合はempty&non-waitにしておく。b-rreqでfull状態であれば、プロセッサに処理を続行させ、empty状態であれば、プログラムの実行をバリア同期待ちにさせる。一方、q-readと同じく、full状態の場合はempty状態に変更して、主記憶に対してb-rreqメッセージを発行する。主記憶側ではこのメッセージ（スレッドID付き）を受けて、該当するスレッドの同期成立カウンタをデクリメントして、まだ0よりも大きければfull状態のデータをキャッシュにインジェクションする。

4 インジェクションとメモリベース同期の利用

本節ではキャッシュインジェクションとメモリベース同期機構の適用例、つまりインジェクションを用いたメモリアクセスレイテンシの隠蔽法やプロセッサ間の通信同期法について、D-machineに基づいて解説する。

4.1 Decoupled Architectureの実現

クラスタ内のメインプロセッサを演算プロセッサと見做し、MBPをそれらに対応するアクセスプロセッサと見做すと、インジェクション機能を持ったL2-cacheを利用したDecoupled Architectureとして動作させることが可能である。そして、MBP-Coreのユーザプログラムを利用して、メインプロセッサに対応したリモートアクセスを含むオペランドフェッチをMBPに行わせ、データをL2-cacheに前もってインジェクションすることでメモリアクセスレイテンシを隠蔽する。また、このアプローチの延長上で、行列計算ではMBPを行列の計算の局所性を高めるために行列要素のメモリ内の配置の並べ替えに使用できる。さらに、連続した配列アクセスのような場合は複数のブロックに跨るデータをバースト転送でMBPからL2-cacheにインジェクション転送すれば、共有バスのバンド幅を増大させることができる。特に、クラスタメモリ（主記憶）にSynchronous DRAMを用いる場合は連続アクセスが極めて高速なので、この方式の価値が高い。

4.2 軽いメッセージ通信への応用

Memory-Based FIFOとfull/emptyビットよる生産者-消費者の同期を組み合わせて軽いメッセージ通信が実現される。メッセージの受信側はクラスタ内（プロセッサに地理的に近いという意味）のメモリ上にFIFOを実現する。メッセージはメッセージ本体とその領域へのポインタからなり、メッセージ本体には必ずfull/emptyが管理された領域を利用し、新しいメッセージを作る場合は受信側のクラスタ内のempty領域をmallocする。送信側はメッセージ本体をmallocした領域に書き込み後（受信側に到着したことを確認する必要はない）、メッセージ本体へのポインタのみを受信側のFIFOに送付（q-write）し、そのポインタを用いて受信側がメッセージを読み出す。データの到着より早く読み出すことはfull/emptyビットで禁止されるので、同期のためのアクセス順序の管理が必要ない。

この状況で、メッセージ本体のデータが主記憶に到着後、MBPを利用してそのメッセージを使用するプロセッサにそのデータをキャッシュインジェクションしておけば、メモリアクセスのレイテンシを低く抑えることができる。

4.3 軽いスレッド切替への応用

軽いコンテクスト切替を可能にするために、次に走るスレッドの選定つまりランキューの管理はMBPがMemory-Based FIFOを使って行う。FIFOの中には新しいコンテクスト（命令アドレスとスタックポインタと退避されたレジスタの内容）へのポインタがキューイングされている。要素プロセッサは各種同期地点でSS-wait（Snoopy Spin wait[26]:実資源の割り当て状況によって同期地点で自らコンテクストを切替えるオプションをもつビジーウェイト方式）を行い、大きな待ちが生じると判断された場合に、現在のコンテクストを退避し、新しいコンテクストを復旧させて、新しい命令アドレスへの分岐を実行する。レジスタの退避復旧がオーバヘッドとなるため、スレッドをスレッドタイプ（例えば同じループのイテレーションは同じスレッドタイプ）ごとに管理して、ランキューもスレッドタイプの数だけ用意する。そして、極力同じスレッドタイプのランキューから次のコンテクストを読み出して、そのスレッドに制御を切替える。このようにして退避復旧するレジスタの数を抑える。

この状況で、次に実行されるコンテクストをMBPがプロセッサへキャッシュインジェクションしておけば、レジスタの復旧のオーバヘッドが小さく抑えられる。

なお、ここに述べられたコンテクスト切替法はオペレーティングシステムの介在なしにユーザモード内で行われる。

4.4 返り値付きメモリベース処理への応用

MBPが主記憶内に複数個駐在していることを前提としているため、test&set, fetch&add, cmp&swap, xchgといったメモリの不可分操作は主記憶上で操作を行うことができる。不可分操作が同一メモリアドレスで競合する場合は、MBPを使って主記憶上でアクセスの順次化が可能になった方が効率が良い。これらの操作は返り値を一つしか持たないが、MBPをユーザに解放したり、もう少し複雑な不可分操作を実装したりすることで、要素プ

ロセッサに対して複数の返り値を要求するMBPの処理も考えられる。この場合、複数の返り値の主記憶内の格納場所へのポインタが全体を代表する返り値として要素プロセッサに返答される。そこで、複数の返り値が格納場所に書き込まれる時点で、キャッシュインジェクションを用いてキャッシュ内にデータを張り付けてしまえば、返り値の読み出し時のオーバヘッドを小さく抑えられる。

4.5 マクロデータフロー実行への応用

本稿で想定しているシステムでは、MBPによって要素プロセッサの実行を細かく制御するというアプローチも可能である。つまり、細粒度で実行形態が静的に予測できないような並列プログラムに対して、動的なマクロデータフロー実行が行える。メモリベース同期機構を用いて、要素プロセッサが実行すべき細粒度のタスクが使用するすべてのデータが揃ったことをMBPが確認後、要素プロセッサで該当タスクを起動させる。この状況において、データが揃った時点で要素プロセッサに対してキャッシュインジェクションを行えば、メモリアクセスのレイテンシが削減できて、効率の良いマクロデータフロー実行が行える。

5 おわりに

プロセッサやコントローラが他のプロセッサのキャッシュにデータを投機的に注入するキャッシュインジェクションの概念を定義し、その有効性と特徴について議論した。メモリ上のデータを伴って他のプロセッサに処理を依頼したい場合には非常に有効であり、軽いメッセージパッシングや手続き呼び出し可能である。また、allread系のプロトコルもキャッシュインジェクション動作を伴うキャッシュプロトコルだと見做せ、allreadプロトコルではデータトラフィックの削減効果が得られる。さらに、インジェクション可能なキャッシュを介して、演算プロセッサとアクセスプロセッサを結合したDecoupled Architectureのレイテンシ隠蔽能力について、他の方式と比較して議論した。ハードウェアコストの若干の問題はあるが、定性的にはかなり優れた方式であることが判った。

また、メモリベース同期機構の利点および高速化方針について述べ、現在開発中の超並列計算機D-machineのアーキテクチャに基づいて、メモリベース同期機構の実装方式について説明した。そして、キャッシュを利用してこの機構を高速化する方式について述べた。

最後に、キャッシュインジェクションおよびメモリベース同期機構の利用法の典型例をD-machineの構成に基づいて示した。

謝辞

本研究の一部は文部省科学研究費（重点領域研究(1)「超並列ハードウェア・アーキテクチャの研究」）による。

参考文献

[1] 松本 尚: 細粒度並列実行支援機構. 計算機アーキテクチャ研究会報告 No.77-12, 情報処理学会, pp.91-98 (July 1989).

[2] T. Matsumoto, et al.: MISC: a Mechanism for Integrated Synchronization and Communication using Snoop Caches. *Proc. of the 1991 Int. Conf. on Parallel Processing*, Vol. 1, pp.161-170 (August 1991).

[3] J.R. Goodman: Using Cache Memory to Reduce Processor-Memory Traffic. *Proc. 10th Int. Symp. on Computer Architecture*, pp.124-131 (June 1983).

[4] R. Katz, et al.: Implementing a Cache Consistency Protocol. *Proc. 12th Int. Symp. on Computer Architecture*, pp.276-283 (June 1985).

[5] M. Papamarcos and J. Patel: A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. *Proc. 11th Int. Symp. on Computer Architecture*, pp.348-354 (June 1984).

[6] E. McCreight: The Dragon Computer System: An Early Overview. Technical Report, Xerox Corp. (September 1984).

[7] C.P. Thacker and L.C. Stewart: Firefly: A Multiprocessor Workstation. *Proc. Second Int. Conf. on Architectural Support for Programming Language and Operating Systems*, pp.164-172 (October 1987).

[8] 松本 尚 他: スヌープキャッシュを用いて通信と同期を統合する機構. 信技報 CPSY 90-42, pp.25-30 (July 1990).

[9] 松本 尚: スヌープキャッシュ制御機構のDOACROSSループへの適用. 並列処理シンポジウムJSPP '92論文集, pp.297-304 (June 1992).

[10] Matsumoto, T. and Hiraki, K.: Dynamic Switching of Coherent Cache Protocols and its Effects on Doacross Loops. *Proc. of the 1993 ACM Int. Conf. on Supercomputing*, pp.328-337 (July 1993).

[11] 松本 尚: スヌープキャッシュ制御機構のDOACROSSループへの適用. 情報処理学会論文誌, Vol.34, No.4, pp.616-627 (April 1993).

[12] J.E. Smith: Decoupled Access/Execute Computer Architectures. *ACM Trans. on Computer Systems*, Vol.2, No.4, pp.289-308 (Nobember 1984).

[13] K. Gharachorloo, et al.: Two Techniques to Enhance the Performance of Memory Consistency Models. *Proc. of the 1991 Int. Conf. on Parallel Processing*, Vol. 1, pp.355-364 (August 1991).

[14] T.F. Chen and J.-L. Baer: Reducing Memory Latency via Non-blocking and Prefetching Caches. *Proc. Fifth Int. Conf. on Architectural Support for Programming Language and Operating Systems*, pp.51-64 (September 1992)

[15] K. Oner and M. Dubois: Effects of Memory Latencies on Non-Blocking Processor/Cache Architectures. *Proc. of the 1993 ACM Int. Conf. on Supercomputing*, pp.338-347 (July 1993).

[16] Intel Corp.: *i860 64Bit Microprocessor Programmer's Reference Manual*. ISBN 1-55523-080-6, Intel Corp. (1989).

[17] 中村 位守 伊藤 中澤: レジスタウィンドウとスーパースカラ方式による擬似ベクトルプロセッサの提案. 並列処理シンポジウム JSPP '92 論文集, pp.367-374 (June 1992).

[18] A. Rogers and K. Li: Software Support for Speculative Loads. *Proc. Fifth Int. Conf. on Architectural Support for Programming Language and Operating Systems*, pp.38-50 (September 1992)

[19] 松本 尚: 局所処理と非局所処理を分離並列処理するアーキテクチャ. 第43回情報処理学会全国大会論文講演集(6), pp.115-116 (October 1991).

[20] 松本 尚, 平木 敬: 超並列計算機上の共有メモリアーキテクチャ. 信技報, CPSY 92-26, pp.47-55 (August 1992).

[21] 松本 尚, 平木 敬: Memory-Based Processorによる分散共有メモリ. 並列処理シンポジウム JSPP '93 論文集, pp.245-252 (May 1993).

[22] 楊 愚魯, 天野 英晴: 超並列向きのプロセッサ結合網の提案. 計算機アーキテクチャ研究会報告 No.96-20, 情報処理学会, (October 1992).

[23] Arvind and R. A. Iannucci: A Critique of Multiprocessing von Neumann Style. *Proc. 10th Int. Symp. on Computer Architecture*, pp.426-436 (June 1983).

[24] 松本 尚: Elastic Barrier: 一般化されたバリア型同期機構. 情報処理学会論文誌 Vol.32 No.7, pp.886-896 (1991).

[25] R.Gupta: The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors. Proc. 3rd Int. Conf. on Architectural Support for Programming Languages and Operating Systems, pp.54-63 (April 1989).

[26] 松本 尚: マルチプロセッサ上の同期機構とプロセッサスケジューリングに関する考察. 計算機アーキテクチャ研究会報告 No.79-1, 情報処理学会, pp.1-8 (November 1989).