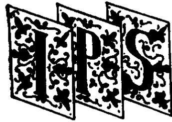
解 説



DASH: スケーラブル共有メモリ型マルチプロセッサ†

漆 原 茂††

はじめに

DASH (Directory Architecture for SHared-memory)[1] は，現在スタンフォード大学コンピュータシステム研究所にて開発されている共有メモリ型マルチプロセッサシステムである．本システムは，数十から数千のプロセッサをサポートするスケーラビリティと，プログラミングモデルを考慮した共有メモリの両者を共に実現している点で注目に値する．本稿ではこのシステムの目的や主な特徴，そして設計や実装とその性能などについて紹介する．

1. DASH プロジェクトの目的とアプローチ

近年のプロセッサ技術の進歩によってマイクロプロセッサとスーパコンピュータの性能のギャップが縮まるにつれ，コストパフォーマンスの点から高性能マイクロプロセッサを用いた並列処理マシンの利点が増してきた．特に，より一層の処理能力を実現するため，価格と性能のバランスを保ちつつプロセッサ数にほぼ比例した性能向上が得られるスケーラブルなマルチプロセッサシステムを構築することが課題となる．スケーラビリティにより，高性能な次世代シングルプロセッサの実現を待つことなく，従来までの小規模なシステム資源をそのまま用いて大規模なシステムを構築することが可能になる．

一方，システム性能のほかに汎用性も大切であり，広範囲のアプリケーションプログラムをサポートするためにシステムがユーザにプログラミングしやすいモデルを提供する必要がある．この点共有メモリ型のマシンには，メッセージパッシング型に比べてプログラマがデータ区分や動的な負荷分散を気にしなくてよいという利点がある．またマルチプログラミング，標準オペレーティングシステム，並列コンパイラなどを支援する上でも共有メモリ型が便利である．

そこで DASH は，スケーラビリティと共有メモリの利点を共に備えたマルチプロセッサシステム構築を目標に置く．従来の共有メモリ型マシンの大きな問題点の一つは，たかだか数十個のプロセッサしか扱えずスケーラビリティを提供できなかったことである．これに対し DASH は分散型ディレクトリ方式によるキャッシュのコヒーレンスプロトコルを提案し，メモリをスケーラブルなネットワーク間の各プロセッシングノードに分散配置することによって，数十から数千のプロセッサを共有メモリモデルでサポートすることを可能にした．またハードウェアによりキャッシュデータの一致性を保つことで，共有メモリへのアクセス遅延を減らしている．

データのブロードキャストを行う従来のスヌーピング方式によるキャッシュは，プロセッサ数が増加するとバスやインタコネクションネットワークが飽和してしまうためスケーラブルなシステムの構築には向いていない．この問題を解決するために提案されたのがディレクトリ方式によるキャッシュである．ディレクトリは各メモリブロックに対し1エントリをもち，メモリブロックのキャッシング状態とデータをもつプロセッサキャッシュへのポインタなどの情報を維持している．メモリブロックへのアクセスにより影響を受けるのはディレクトリエントリに記録されているプロセッサキャッシュだけなので，プロセッサ間の1対1通信を用いたコヒーレンスプロトコルが実装でき，バスやインタコネクションネットワークの飽和を回避できる．またディレクトリ方式はバスやインタコネクションネットワークの実装に依存しないため，さまざまなスケーラブルネットワーク

† DASH: A Scalable Shared-Memory Multiprocessor by Shigeru URUSHIBARA (Computer Systems Laboratory, Stanford University and Oki Electric Industry Co., Ltd.).
†† スタンフォード大学，沖電気工業(株)

を使用することができる．DASH はこのディレクトリ方式を拡張した分散型ディレクトリを実現しクラスタベースのコヒーレンスプロトコルを用いることによって，より一層のスケーラビリティを追及している．

本稿では，まず次章にて DASH の基本アーキテクチャについて説明し，続いて DASH プロトタイプシステムの実装について解説する．さらにソフトウェアによるサポートについて簡単に触れ，プロジェクトの経緯と現状の説明を行い，システムの性能データを紹介する．

2. DASH アーキテクチャ

DASH アーキテクチャの概観を図-1 に示す．クラスタと呼ばれる主な処理ノードが，互いに高速大容量のネットワークで接続されている．主メモリは各クラスタに分散配置され，かつ全てのクラスタからシングルアドレススペースとしてアクセスできるようになっている．1クラスタは，個々にキャッシュをもつ高性能プロセッサ群，メモリ，そしてネットワークインタフェースを含むディレクトリから構成される．ディレクトリは，クラスタ内に配置された主メモリに対応したエントリをもち，各メモリブロックの情報（他クラスタによるキャッシング状態など）を保持する．

メモリ要求により発生するネットワーク上のトラフィックを減少させるため，DASH ではコヒーレントなキャッシュと分散メモリを用いている．すなわち，コヒーレンスプロトコルの許すかぎりデータをキャッシュに保持し，同一のプロセッサによる共有データへのアクセスをフィルタリングする．またメモリを各クラスタに分散させ，局所性のあるデータや共有されないデータへのアクセスをローカルクラスタ内に抑える．さらに，コヒーレンスプロトコルはクラスタ間メッセージ数をできるだけ減らすようにしてある．本章ではキャッシュコヒーレンスプロトコルの概略，コンパイラやプログラマからみたメモリのコンシスタンシモデル，そしてメモリ遅延隠蔽や同期方法のテクニックについて説明する．

2.1 メモリの階層構造とメモリ要求の流れ

DASH のキャッシュは，メモリブロックの所有権に基づく無効化方式によるコヒーレンスプロトコルにより維持されている．各メモリブロックは，uncached/shared/dirty のうちいずれかの状態をとる．uncached はメモリブロックがどのクラスタにもキャッシングされていないことを表す．shared 状態では，一つ以上のクラスタ内のキャッシュにデータが変更されないまま保持されている．dirty は変更されたデータが，あるクラスタ内の一つのキャッシュに唯一保存されている状態である．クラスタ内のディレクトリは，対応する各メモリブロックの状態とそれをキャッシングしているクラスタの情報をもつ．

それぞれのプロセッサからみたメモリシステムは論理的に，プロセッサレベル，ローカルクラスタレベル，ホームクラスタレベル，リモートクラスタレベルの4つの階層構造に分けることができる．プロセッサレベルはプロセッサのキャッシュを指し，このレベルで処理できないメモリ要求はローカルクラスタレベルに送られる．ローカルクラスタレベルは，メモリ要求を行ったプロセッサと同じクラスタ（ローカルクラスタ）内の他プロセッサキャッシュを意味する．もしデータがローカルクラスタ内の他キャッシュに保持されていれば，メモリ要求はクラスタ内だけで処理される．データがローカルクラスタレベルに見つからないとき，要求はホームクラスタレベルに送られる．

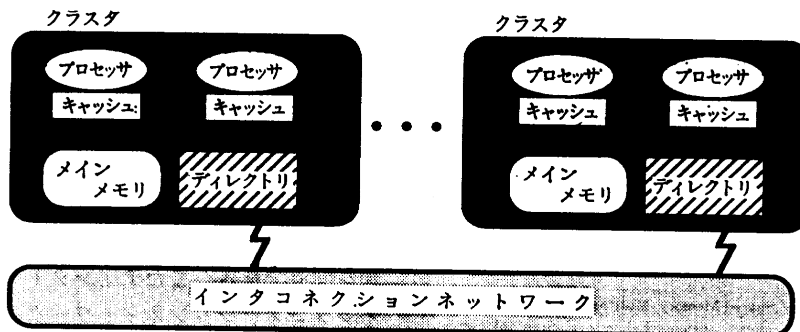


図-1 DASH アーキテクチャ概観

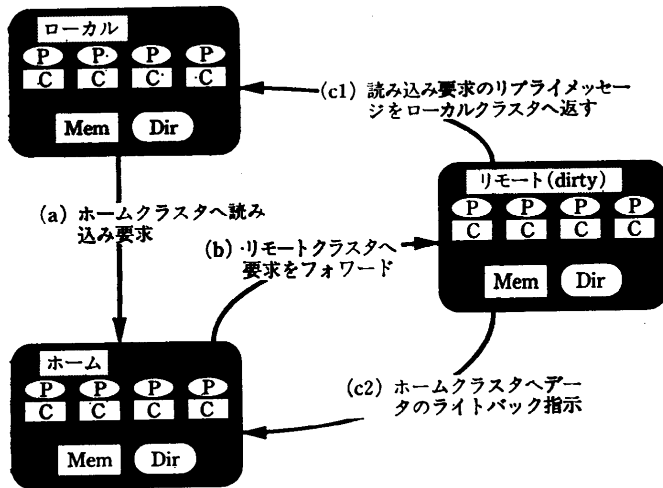


図-2 dirty データへの読み込み要求処理

ホームクラスタとは，要求されたメモリアドレスに対応する主メモリとディレクトリをもつクラスタのことである．メモリアドレスがローカルクラスタ内の主メモリを指すときに限りローカルクラスタがすなわちホームクラスタとなるが，より一般的にはローカルクラスタからホームクラスタへネットワーク経由で要求が送られる．通常の場合，ホームクラスタによってメモリ要求が満たされるが，メモリブロックが dirty であるときまたはメモリ要求が shared 状態のメモリブロックに対する書き込みであるとき，データをキャッシングしているリモートクラスタへホームクラスタからネットワークを経由したアクセスが行われる．この場合ホームクラスタのディレクトリ情報が必要に応じて更新され，リモートクラスタからローカルクラスタに直接データが返される．以下では，具体的に読み込み及び書き込み時のメモリ要求の流れの概略を上記階層構造に従って説明する[2]．

（1） 読み込み要求処理の流れ

データがプロセッサキャッシュ内に存在するとき，単にキャッシュがデータを供給し要求は完了する．反対にデータがプロセッサキャッシュ内に存在しないとき，読み込み要求はローカルクラスタレベルに送られる．データがローカルクラスタ内のどこかのキャッシュに保持されているとき，読み込み要求はローカルクラスタ内だけで処理されクラスタ間の通信は行われない．ローカルクラスタ内に見つからないとき，データアドレスのホームクラスタへ要求が送られる．ホームクラスタはデータアドレスのディレクトリ情報を参照し，データのキャッシング状態を調べる．もしデータが dirty でなければホームクラスタはディレクトリ情報をローカルクラスタとの共有状態に更新し，ローカルクラスタへ読み込みデータを返す．一方データが dirty であるときの処理を**図-2** に示す．ホームクラスタはローカルクラスタからの要求を受けとった後(a)，有効データを保持しているリモートクラスタへ要求をフォワードする(b)．リモートクラスタでは，ローカルクラスタにキャッシュデータのコピーを返すとともに(c1)，ホームクラスタへメモリのライトバック要求を行う (c2)．ライトバック要求を受けたホームクラスタは，主メモリの内容を更新しディレクトリを shared にする．

（2） 書き込み要求処理の流れ

プロセッサからの書き込み要求処理は，キャッシュの所有権の移動が含まれるため読み込み要求とは多少異なる．まずデータがプロセッサキャッシュ内に dirty として存在するとき，書き込み要求は単にキャッシュデータを更新して終了する．それ以外の場合，ローカルクラスタレベルに排他読み込み要求が送られる．すなわちキャッシュの所有権の獲得要求である．ローカルクラスタ内のどこかのキャッシュがすでにキャッシュの所有権をもっているとき，排他読み込み要求はローカルクラスタ内でキャッシュ間データ転送が行われて完了しクラスタ間通信は行われない．反対にローカルクラスタ内のどのキャッシュも所有権をもたないとき，排他読み込み要求がホームクラスタへ送られる．特にメモリが他クラスタ間で共有されているときの処理を**図-3** に示す．ローカルクラスタから排他読み込み要求を受けとった(a)ホームクラスタでは，ディレクトリを調べ状態が uncached または shared の場合はローカルクラスタへ所有権を移動させ (b1)，キャッシュを共有していた総クラスタ数をキャッシュデータとともに返す．shared であった場合はさらにデータを共有している全てのクラスタにホームクラスタからキャッシュ無効化要求が送られる (b2)．この無効化要求を受けとった各クラスタは，キャッ

シュを無効化するとともに完了の確認メッセージをローカルクラスタに送る(c). ローカルクラスタでは，ホームクラスタから返された総共有クラスタ数とこの確認メッセージ数を比べることにより，書き込み要求が全てのクラスタに対し完了したかどうか知ることができる．一方ホームクラスタのディレクトリが dirty であった場合，排他読み込み要求はホームクラスタから有効データを保持しているリモートクラスタへフォワードされる．リモートクラスタは，読み込み要求処理と同様，ローカルクラスタに直接データを返すとともに，ホームクラスタへ所有権移動メッセージを送る．このメッセージを受けとったホームクラスタは，キャッシュ所有権をローカルクラスタへ移動させる．

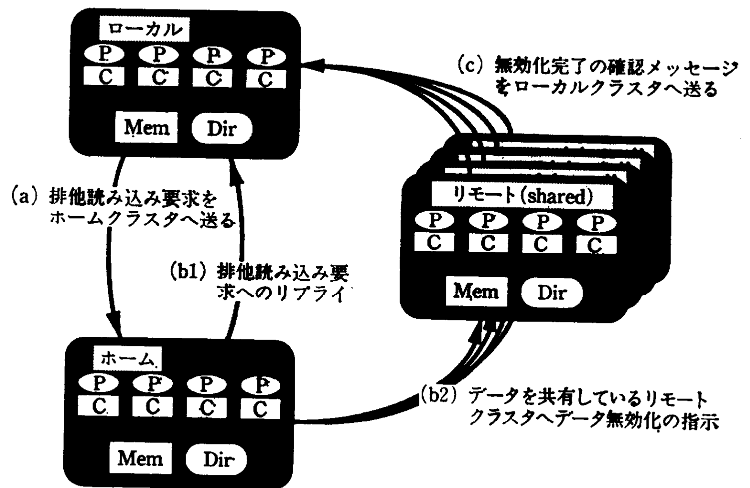


図-3 shared データへの書き込み要求処理

所有権をもたないクラスタからの書き込み要求の完了パターンは，ホームクラスタのディレクトリ状態により三つに分けられる．ホームクラスタのディレクトリ状態が uncached であるとき，ホームクラスタからのリプライをローカルクラスタが受けとった時点で書き込みは完了する．shared であるとき，他クラスタからの全ての無効化完了確認メッセージをローカルクラスタが受けとった時点で完了する．dirty であるとき有効データを保持していたリモートクラスタからのリプライをローカルクラスタが受けとった時点で完了する．いずれの場合も書き込み要求が完了した時点で，システム内に古いデータコピーが存在しないことが保証されている．実際には書き込み要求がキャッシュ所有権を得た時点で，全ての無効化確認メッセージを待たずに次のメモリアクセスを許すことにより，プロセッサ使用率を上げることができる．しかしこれはまた，メモリアクセスをプログラマの予期したのと異なる順序で実行することにつながるため，適度なメモリコンシスタンシモデルをプログラマに対して定義することが重要になる．次節ではこれについて，DASH のアプローチを説明する．

2.2 メモリコンシスタンシモデル

プロセッサとメモリアクセススピードのギャップを埋めるため，しばしばメモリアクセスのキャッシングやパイプラインやバッファリングなどの技術が使われる．反面これらのハードウェアによる最適化により実際のメモリアクセスがプログラマの予期と異なる順序で行われる可能性が発生するため，プログラミングモデルの受ける影響も大きい．そこでプログラマに適度なプログラミングモデルを提供しつつ，どこまでメモリアクセスの順序に自由度をもたせるかが課題となる．従来までに提唱されてきたマルチプロセッサのメモリコンシスタンシモデルのうち最も制限が強いものはシーケンシャルコンシスタンシ[3]である．このモデルは，並列プログラムが単一プロセッサ上である時間間隔をおいて実行されたのと等しくなるようにメモリアクセスを制限する．すなわちこのモデルでは同一プロセッサによるメモリアクセス同士のオーバラップが許されないため，ハードウェアによるパイプラインなどの機能を生かすことができない．そこで他のモデルではメモリアクセス間の順序制限を減らす工夫がなされている．

実際の共有メモリ型の並列プログラムを観察してみると，クリティカルセクションの更新は同期を用いた排他制御（ロック/アンロックなど）により，すでにプログラムの手でコンシスタントに保たれていることが分かる．同期点が正しく指定されているプログラムでは，同期点においてメモリシステムがコンシスタントに保たれていればよ

い．そこで DASH は，同期の解放（アンロックなど）時点においてのみそれ以前のメモリ要求が他のプロセッサに対し完了することを保証するリリース（release）コンシスタンシモデル[4]を採用している．このモデルはクリティカルセクションを抜ける際に更新されたデータがコンシスタントであることを保証するので，ユーザは同期点を用いた適度なプログラミングモデルを使用することができる．またこのモデルによりメモリアクセスのオーバラップが許されるので，プロセッサ使用率を上げることができる．シミュレーションによると，リリースコンシスタンシはシーケンシャルコンシスタンシに比べて 10～40% の性能向上につながることが示されている[5]．

DASH のリリースコンシスタンシの実装はハードウェアにより行われている．プロセッサごとの書き込みバッファの実現により，プロセッサはブロックせずに書き込み要求を実行することができる．書き込みバッファはプロセッサの書き込み要求をキューに蓄え，順番に実行していく．このためプロセッサは，書き込み要求実行中に次の命令を実行できることになる．また，リリースコンシスタンシモデルでは複数の書き込み要求同士の処理もオーバラップさせることができる．これを DASH は，書き込み要求実行時，キャッシュの所有権を取得した時点で他キャッシュの無効化が全て完了するのを待たずに次の書き込み要求の処理を行うことで実装している．

プログラムによっては，二つの異なるメモリアクセス間の順序を保つことが必要になるときがある．そのため，以前のアクセスが完了するまで現在のアクセスを遅らせるなんらかのメカニズムが必要である．そこで DASH では後に続く全ての読み込み及び書き込み要求を遅らせる full fence と，後に続く全ての書き込み要求のみを遅らせる write fence を提供する．fence の実現のためには現在未完了の全てのメモリ要求を記録しておく必要がある．そこでプロセッサごとの未完了メモリアクセス数を示すカウンタをシステム（後述の RAC 内）に用意し，プロセッサがメモリ要求を行うたびにインクリメント，メモリ要求が完了するたびにデクリメントするようにする．full fence は，このカウンタが 0 になるまでプロセッサを待たせることであり，write fence は同様に書き込みバッファを待たせることである．DASH のリリースコンシスタンシは，同期の解放時に暗黙の write fence を行うことで実現される．他の異なるメモリコンシスタンシモデルも，ソフトウェアが明示的に fence を指定することで実現できる．

一方，同一メモリブロックへのアクセスが重なったときの扱いも考慮する必要がある．特に書き込みにより他キャッシュの無効化を待っているメモリブロックに対しメモリアクセス要求が行われたときには注意を要する．これを DASH では，実行中の要求が完了するまで同じメモリブロックに対する新しいアクセス要求をリトライやバッファリングにより遅らせることで解決している．

2.3 同期機構のサポート

DASH はキャッシュコヒーレンスプロトコルを拡張し，以下の二つを同期機構として提供する．

（1） キュー方式によるロック

キャッシュを用いた従来の test-and-test&set 方式によるスピンロックの代わりに，DASH はキュー方式によるロックをサポートする．この方式は，ロックの競合が起きた際，ロックの解放を待つプロセッサをそのプロセッサキャッシュ内でスピンさせ，スピンしているプロセッサのクラスタ番号をロックのディレクトリに保持するものである．ロックが解放されると，ディレクトリのエントリから次にロックを取得するクラスタがランダムに一つ選ばれ，ロックの許可がそのクラスタに送られる．許可を受けとったクラスタでは，ロックの解放を待つどのプロセッサもクラスタ内だけの処理によりロックを取得できる．ロックの解放を一つのクラスタにしか知らせないため，従来の test-and-test&set 方式に比べてロック解放時の不要なトラフィックや遅延が少ない．

（2） fetch-and-operation

この同期方式は，キャッシュを用いないアトミックなインクリメント/デクリメントであり，変更される前の値を返す．実際のオペレーションはメモリ側が直接行うため，遅延やシリアライズされる部分が少ない．この方式はバリアや分散ループなどを実現する際に有用である．

2.4 特殊なメモリオペレーション

DASH のリリースコンシスタンシモデルは書き込み時の遅延の隠蔽に特に有効であるが，

DASH のプロセッサは読み込み時にデータがくるまでブロックしてしまうため，読み込み時のキャッシュミスによる遅延の影響を免れない．この問題に対して，DASH は以下のメカニズムをサポートする．

（1） prefetch

prefetch はソフトウェアによるデータの明示的な先読み要求であり，要求されたメモリブロックのデータがローカルクラスタ内キャッシュに読み込まれる．要求したプロセッサは prefetch の結果を待たずに次の命令を実行することができる．実際の読み込み要求の十分前に先読みを行うことにより，実際の読み込み時にはデータがすでにクラスタ内キャッシュに存在していることが期待できる．prefetch によりキャッシュに読み込まれたデータは他のデータ同様ハードウェアによりコンシスタントに保たれる．たとえば先読みを行ったプロセッサが実際に読み込む前に他プロセッサが書き込みを行った場合，先読みデータはキャッシュプロトコルにより無効化される．このとき，prefetch の効果は減じるがプログラムは正しく動作することが保証されることになる．すなわち prefetch はコンシスタンシモデルに影響を与えないため，コンパイラなどから積極的に使用することができる．DASH は上記 prefetch のほかに，排他 prefetch もサポートする．排他 prefetch ではメモリブロックの先読み時にキャッシュの所有権も獲得する．これは，共有データを読んでから更新するような場合に有用である．prefetch をリリースコンシスタンシモデルと組み合わせることによって 60～130% の性能向上が得られることがシミュレーションで報告されている[6]．

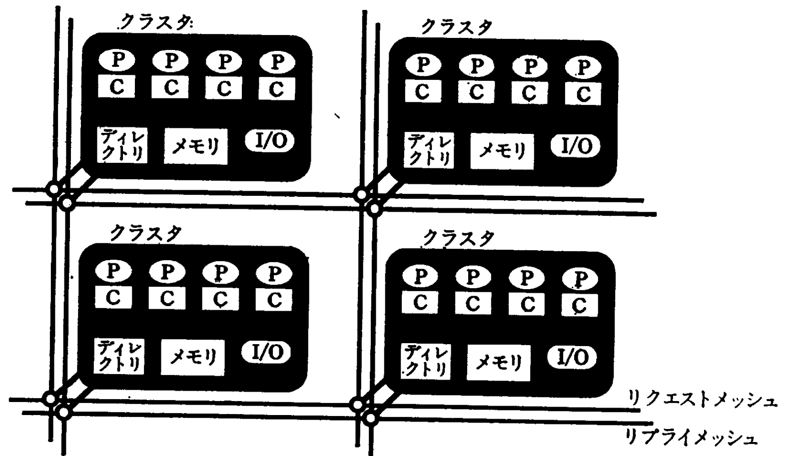


図-4 DASH プロトタイプブロック図（2×2）

（2） update-write と deliver

update-write は，データ書き込み時，他プロセッサキャッシュを無効化するかわりに，ホームクラスタのメモリと存在する全てのキャッシュコピーを更新するものである．このオペレーションはプロデューサのデータを複数のコンシューマが同時に待っているときなどに便利である．deliver はデータのキャッシュコピーを直接複数のクラスタへ送るものである．これによりプロデューサは書き込みによる排他状態のキャッシュデータを明示的にコンシューマのクラスタへ届けることができる．

3. DASH プロトタイプ

前章で述べたアーキテクチャ設計に基づき，スタンフォード大学では DASH のプロトタイプシステムを制作した[2]．本章ではその実装上の各コンポーネントについて解説する．

3.1 クラスタの実装

図-4 に4クラスタ (2×2) モデルの DASH システムのブロックダイアグラムを示す．各クラスタには，シリコングラフィックス社の PowerStation 4D/340* がベースとして用いられている．一つの 4D/340 は4つの 33 MHz プロセッサ (MIPS R 3000/R 3010**) からなる．各プロセッサは，64Kバイトの命令キャッシュと 64K バイトのライトスルーデータキャッシュをもつ．このデータキャッシュ（ファーストレベル）は，リードバッファと4ワードのライトバッファを介して 256K バイトのライトバックキャッシュ（セカンドレベル）に接続されている．ファーストレベル，セカンドレベル共に 16 バイトラインのダイレクトマップキャッシュである．ファーストレベルキャッシュはプロセッサスピード (33 MHz)，セカンドレ

* Power Station and 4 D/340 are trademarks of Silicon Graphics.
** MIPS R 3000 and R 3010 are trademarks of MIPS Computer Company.

ベルキャッシュはバススピード(16.67 MHz) に同期している．

クラスタ内のキャッシュのコヒーレンスはイリノイプロトコルによるスヌープバス方式により保たれている．特にイリノイプロトコルのキャッシュからキャッシュへの転送機能は，リモートメモリのキャッシュミス処理時に有効である．クラスタ内のメモリバスは同期型で，32ビットのアドレスバス，64ビットのデータバスをもつ．最大データ転送能力は毎秒 67 M バイトである．本バスはもともとスプリットトランザクションではなかったので，バスリトライ信号を既存基板につけ加え，バスアービタを変更してリモート要求実行中プロセッサのリトライをマスクするようにした．これによりリモートアクセス完了時にリトライを行うスプリットトランザクションバスを実現している．

シミュレーションによれば数百のプロセッサをディレクトリ方式によって結合することが可能であるが，プロトタイプの実装は最大 16 クラスタ(64 プロセッサ) までに制限されている（これは主に 4D/340 のアドレス空間の制限のためである）．それでもシステム全体で最大 1.6 GIPS, 600 スカラ MFLOPS の性能をもつことになる．

3.2 クラスタ間ネットワーク

DASH のディレクトリコヒーレンスプロトコルはネットワークのトポロジに依存しないが，実際にスケーラブルなネットワークが必要になる．DASH プロトタイプシステムは，異なるクラスタ間の接続にワームホールルーティングを用いたペアのメッシュネットワークを用いている．一つはリクエストメッセージ，もう一つはリプライメッセージに使用され，それぞれ 16 ビット幅をもつ．ネットワークのコントローラには Caltech のメッシュルーティングチップの拡張版が使用されている．ネットワーク内の各ホップの平均遅延は約 50 ns，プロトタイプでは各クラスタへの入出力に毎秒 120 M バイトのバンド幅をもつ．

3.3 ディレクトリの実装

DASH システムのディレクトリは，ディレクトリ方式によるコヒーレンスプロトコルの実装やクラスタ間のネットワーク接続を行うものである．図-5 に示すように DC ボードと RC ボードの二つの基板に分かれる．

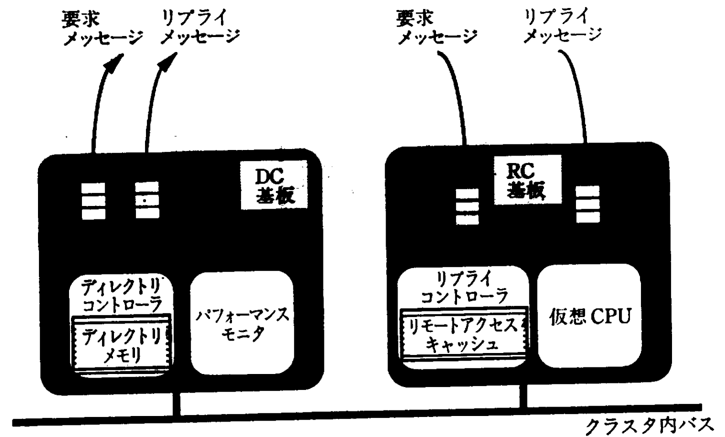


図-5 DASH ディレクトリブロック図

DC ボードは，ディレクトリコントローラ(DC)，パフォーマンスモニタ，そして他クラスタにリクエスト/リプライメッセージを送り出すためのネットワークコントローラからなる．ディレクトリコントローラは，クラスタ内の主メモリに対応したディレクトリ情報を保持するメモリをもっている．各メモリブロックに一つディレクトリエントリが割り当てられ，shared/dirty を示すフラグとクラスタのビットベクタをもつ．このディレクトリメモリは主メモリと同じ DRAM で構成されている．パフォーマンスモニタはクラスタ間及びクラスタ内のさまざまなイベントをトレースするツールである．

RC ボードは，リプライコントローラ (RC)，仮想 CPU (PCPU)，そして他クラスタからのリクエスト/リプライメッセージを受けとるためのネットワークコントローラから構成される．リプライコントローラは，ローカルクラスタ内のプロセッサによる実行中のリモートメモリアクセス情報を保持するとともに，リモートアクセスキャッシュ (RAC) を用いてリモートクラスタからのリプライを受けとりバッファリングする．リモートメモリ要求完了時それまでマスクされていたローカルプロセッサがリトライを行うと，RAC がキャッシュ間転送によりデータを供給する．PCPU は他クラスタからの要求を取り扱う仮想 CPU であり，バッファリングした外部からのメモリ要求を順にクラスタ内バスに実行する．

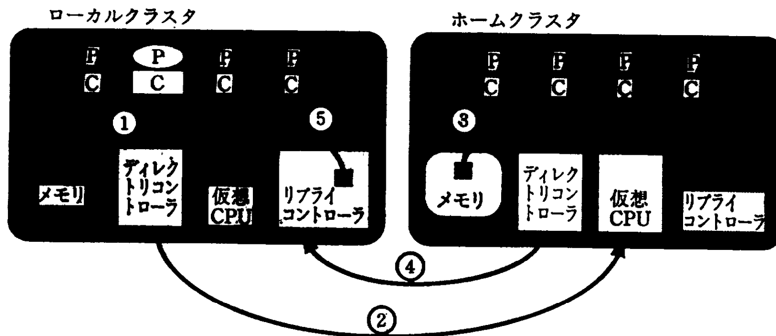


図-6 リモートメモリアクセス例

PCPU の実行によるバスの応答結果はディレクトリコントローラにより他クラスタへ送られる.

3.4 ディレクトリコヒーレンスプロトコル例

ここでは前節までに説明した DASH の各コンポーネント同士が, ディレクトリプロトコルを用いてどう動作するかを, ホームクラスタへのメモリアクセスを例にあげて説明する (**図-6**). プロトコルについては文献7) が詳しい.

簡単のため, アクセスされるメモリブロックはどのプロセッサにもキャッシングされていないものとする. まずプロセッサがローカルクラスタ内バスにリモートクラスタへのアクセス要求を出す(①). DC はリモート要求を判別してホームクラスタの PCPU へネットワーク経由でメモリアクセス要求を送る(②). これを受けとった PCPU は, ホームクラスタバスにデータ要求を出す(③). この結果は, ホームクラスタの DC からローカルクラスタの RC へネットワーク経由で返される(④). 最後に要求元プロセッサがメモリアクセスのリトライを行い, RAC がデータを供給する(⑤).

4. ソフトウェアサポート

DASH のオペレーティングシステムは, 4D/340 上のカーネル (Unix* System V Release 3 を変更したもの) に基づき, クラスタや DASH のハードウェアをサポートするように改良を加えたものである. これにより Unix のマルチユーザ/マルチタスクの環境が DASH システム上で実現されている. 現在, プロセススケジューラやメモリ管理などのアルゴリズムの研究が行われている. このほか, プログラムの並列性を抽出するだけでなくデータの局所性も考慮したコンパイラや, 並列プログラムを記述する言語などの研究も行われている. さらにプログラムの性能評価のためのモニタリングツールの作成なども行われている.

* Unix is a trademark of AT & T.

5. プロジェクトの経緯と現状

前章までに述べた DASH プロトタイプはハードウェア・ソフトウェア共すでに開発中であり, 現在 4 クラスタシステムのデバッグがほぼ完了した段階である. ここでは現在に至るまでの DASH の開発経緯について簡単に触れる.

DASH プロジェクトは, 1988年9月にシステムアーキテクチャの定義を行うことから始まった. 以来 1989 年末までシミュレータの開発やディレクトリプロトコルの定義などが行われ, ゲートレベルの設計が開始されたのは 1990 年に入ってからである. その後基板の設計やレイアウトが順調に進み, 1990 年の秋にプロトタイプシステム用のディレクトリコントローラ基板の初版が製造された. 以後ハードウェアとともにオペレーティングシステムのデバッグも行われ, 1990 年末には 2 クラスタのシステム立ち上げに成功した. そして 1991 年の 4 月には 4 クラスタ計 16 プロセッサの安定したシステムが完成し, ハードウェア・ソフトウェア共十分使用に耐え得るレベルで稼働している. DASH プロジェクトは Anoop Gupta 教授と John Hennessy 教授の指導のもと, 6 人の学生と技術スタッフ 1 人を中心にすすめられている. さらにアプリケーションやシミュレーションなどのため 10 人以上の学生がプロジェクトをサポートしている.

現在, 各種ベンチマークプログラムのプロトタイプシステムへの移植がすすみ, その評価が行われているところである. 次章では, 4 クラスタの

表-1 負荷がないときのメモリ読み込み遅延

メモリレベル	cache 1 st-level	cache 2 nd-level	local memory access	2-cluster access	3-cluster access
読み込み遅延	1 clock (30 ns)	15 clocks (450 ns)	30 clocks (900 ns)	100 clocks (3000 ns)	135 clocks (4000 ns)

プロトタイプシステムの性能について実測データをもとに解説する．

6. DASH の性能

DASH プロトタイプにおける各メモリ階層への読み込み遅延は，システムに負荷がかかってないときおおよそ上のようになる（表-1）．

プロセッサのファーストレベルキャッシュからの読み込みは1クロック，セカンドレベルキャッシュからは 15 クロックかかる．ローカルクラスタへの読み込みミスの 30 クロックに対して，リモートクラスタへのアクセスはその約 3.5 倍かかる[1]．すなわち，リモートメモリへのアクセス遅延はローカルメモリに比べてたいへん大きい．そこで DASH ではプログラムのデータ局所性をうまく抽出し，データアクセスをできるだけキャッシュ内あるいはローカルクラスタ内に抑える工夫が重要になる．

プロトタイプシステム評価は，現実的な並列プログラムを走らせることで行われている．DASH グループが用いているプログラムには，行列演算を行うもの，CAD のルータ，論理シミュレータ，分子シミュレーションなどがある[8]．プログラムのスピードアップは一概には言えないが，十分な局所性をもつプログラムやキャッシュヒット率の良いプログラムでは 16 プロセッサ上でほぼ 10～15 以上のスピードアップが得られることがシミュレーションで報告されている[2]．

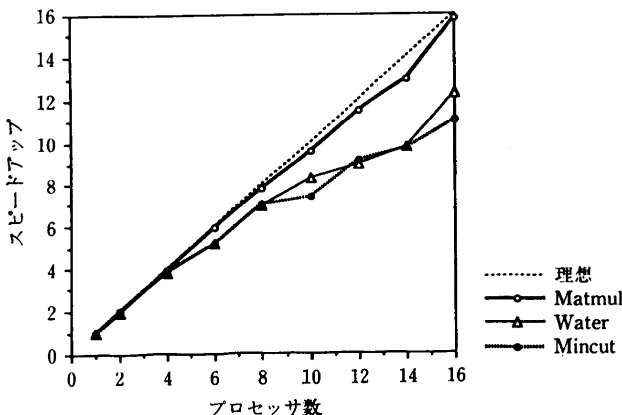


図-7 並列プログラムのスピードアップ

4クラスタ計 16 プロセッサの DASH プロトタイプシステム上で実測したスピードアップを，幾つかのプログラムを例に図-7 に示す．Matmul は行列同士の積を計算するプログラムで，Water は水分子から構成されるシステムのエネルギを求めるものである．Mincut はグラフ問題を解決する．Matmulはキャッシュヒット率が良くほぼ理想に近いスピードアップをし，最大で 114 MFLOPS の性能を引き出している．これはほぼ現 DASH システムの最大性能に近い．Water は十分なデータの局所性をもっておりローカルクラスタ内で処理されるキャッシュミス率がほぼ一定であるため，キャッシュミス時の遅延がクラスタ数増加の影響を受けにくく良いスピードアップをもつ．Mincut の良いスピードアップの原因は高いキャッシュヒット率からくる．いずれも，ディレクトリプロトコルによる共有データのキャッシングの有効性を示すものである．

7. 結　論

スケーラビリティと共有メモリ型モデルの両者の利点を供えたマルチプロセッサシステム DASH について，その設計と実装を述べた．鍵となる幾つかの点は，ディレクトリ方式によるコヒーレンスキャッシュ，スケーラブルネットワーク，分散型メモリとディレクトリなどである．またプロセッサとメモリ遅延のギャップを埋めるため，さまざまな方法が採用されている．特に，メモリのリリースコンシスタンシモデル，データのプリフェッチ，さらにキュー方式による効果的な同期方式などを実装することで，より高い性能を引き出している．現在 DASH プロトタイプシステムは4クラスタ計 16 プロセッサで構成され，安定して稼働している．本年末までには16クラスタ計64プロセッサのシステムが稼働する予定である．本シ

ステムを基盤として，DASH グループでは今後さらに並列処理研究がすすめられていく.

謝辞 本稿執筆にあたり，スタンフォード大学コンピュータシステム研究所の Anoop Gupta 教授や John Hennessy 教授をはじめ，DASH プロジェクトのメンバには大変お世話になった．特に Kourosh Gharachorloo, Daniel Lenoski, James Laudon の各氏に厚く感謝の意を表したい.

参 考 文 献

1) Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.D., Gupta, A. and Hennessy, J.: Overview and Status of the Stanford DASH Multiprocessor. In proceedings of the International Symposium on Shared Memory Multiprocessing, pp. 102-108 (Apr. 1991).

2) Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W. D., Gupta, A., Hennessy, J., Horowitz, M. and Lam, M.: The Stanford DASH Multiprocessor, IEEE Computer, to appear.

3) Lamport, L.: How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs, IEEE Transactions on Computers, C-28 (9), pp. 241-248 (Sep. 1979).

4) Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A. and Hennessy, J.: Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors, In Proceedings of the 17 th Annual International Symposium on Computer Architecture, pp. 15-26 (May 1990).

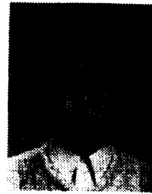
5) Gharachorloo, K., Gupta, A. and Hennessy, J.: Performance Evaluation of Memory Consistency Models for Shared-memory Multiprocessors, In Proceedings of Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (Apr. 1991).

6) Gupta, A., Hennessy, J., Gharachorloo, K., Mowry, T. and Weber, W. D.: Comparative Evaluation of Latency Reducing and Tolerating Techniques. In Procceedings of the International Symposium on Computer Architecture, pp. 254-263 (May 1991).

7) Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A. and Hennessy, J.: The Directory-based Cache Coherence Protocol for the DASH Multiprocessor, In Proceedings of the 17 th Annual International Symposium on Computer Architecture, pp. 148-159 (May 1990).

8) Singh, J.P., Weber, W.D. and Gupta, A.: SPLASH: Stanford Parallel Applications for Shared-memory, Technical Report CSL-TR-91-469, Stanford University (Apr. 1991).

(平成3年7月29日受付)



漆原 茂

1965年生. 1987年東京大学工学部計数工学科卒業. 1987年沖電気工業に入社，現在に至る. 1989～1991年 Stanford 大学 Computer Systems Laboratory 訪問研究員として DASH プロジェクトに参加. 現在沖電気工業コンピュータシステム開発本部に勤務，分散処理システムの研究開発に従事. IEEE 及び ACM 各会員.