４．分散共有メモリ（NUMA）

集中共有メモリ方式は、規模がどうしても 8 コア前後で限界に達する。これ以上規模を大きくするためには、共有メモリを分散させるしか手がない。図 1 に示すように、それぞれのノードにメモリを持たせ、全体としてこれを一連のメモリであるかのように見せれば良い。この図で示すノードは単一のプロセッサではなく、集中共有メモリ型のマルチコアのチップから構成される場合が増えてきた。この場合、自分のノードに接続されているメモリの領域は高速でアクセス可能だが、他のノードのメモリはネットワークを介するため、アクセスが遅くなってしまう。すなわち Non-Uniform Access Memory model (NUMA)となる。NUMA の中で、他のノードのメモリ領域をキャッシュ可能で、一貫性をハードウェアで保証してくれるものを CC-NUMA（Cache Coherent NUMA）と呼ぶ。CC-NUMA は、遠隔地のブロックをキャッシュすることで高速にアクセス可能だが、一貫性を保証するハードウェアを必要とするためコストが高くなる。

4.1　ディレクトリ方式

4.1.1 基本的なディレクトリ方式

スヌープキャッシュは、共有バスという、排他的かつ全員がチェックできる資源を使ってキャッシュの一貫性を簡単に管理することができたが、この混雑により規模に制限が加わった。ディレクトリ方式は、それぞれのノードのメモリ（L2/L3 キャッシュ）に管理用のハードウェアを設けると共に、各ブロックに対して共有情報を持たせる。すなわち、アクセスする領域によって、管理を行うメモリが決まることになる。これをホームメモリと呼ぶ。複数のノードがそれぞれ別のホームメモリをアクセスする場合、そのキャッシュ管理のためのアクセスは、それぞれのホームメモリで分散される。

ここでは、ノード 0 のホームメモリを他のノードがアクセスする場合を考える。アクセス対象のメモリブロックにはそのブロックの状態を示すビットと、共有関係を示すビットマップを装備している。ここでは、ブロックの状態は、

U：Un-cached:キャッシュされていない

S：Shared：キャッシュされているが内容は一致している。

D:Dirty：書き込みを行ったキャッシュがあり、ブロックの内容は最新ではない

の三状態を使う。ビットマップはそれぞれノード 0，1，2、4 に対応する 4 ビットを持つ。

各ノードのキャッシュのブロックはそれぞれ次の状態を持つ

I:Invalidate：無効化されている

S:Shared:ホームメモリと同一内容でキャッシュしている

D:Dirty：書き込んでホームメモリと内容が一致しない

以下に例を示しながらブロックをキャッシュする様子を示す。

１）ノード 3 がノード 0 のホームノードを読み出しする場合（図２）

① ノード 3 が読み出したアドレスが、ノード 0 のホームメモリの領域の場合、キャッシュブロックの読み出し要求信号がノード 0 に送る。

② ノード 0 は、要求メッセージを受け取り、ディレクトリを参照することで、要求されたブロックが状態 U であることを知り、以下の操作を行う i)状態を S に変更、ii)要求元のノード 3 に相当するビットマップに 1 をセット、すなわちビットマップは 0001 になる。 iii)ノード 3 に要求されたキャッシュブロックを転送

③ ノード 0 はキャッシュブロックを受け取るとキャッシュ領域に取り込み、対応するディレクトリの状態を S にする。

上記の様子を図 2 に示す。読み出しアクセスならば、どのノードが要求しても同様な処理を行う。図 3 では続けてノード 1 が読み出し要求を出した場合を示す。この場合、ホームメモリの状態は S、ビットマップは 0101 になり、ノード 0, 3 の状態は S となる。以降、ノード 0、ノード 3 は読み出しについてはホームメモリにお伺いを立てずにキャッシュを読むことができる。

では、次にノード 3 が書き込み要求を出す場合はどうなるだろう。

２）ノード 3 が書き込み要求を出す場合（図 4）

① ノード 3 は書き込み要求をホームノードに送り、待ち状態になる。

② ノード 0 は、ディレクトリを検索し、このブロックが S で、ビットマップからノード 1 とノード 3 が同一ブロックを共有していることを知る。

③ ノード 0 は、ビットマップが 1 であるノードで要求元を除いたもの、すなわちノード 1 に対して無効化メッセージを転送する。ビットマップが 1 であるノードが複数あればすべてに無効化メッセージを転送する必要がある。

④ 無効化要求を受け取ったノード 1 は、自分のキャッシュの状態を I として、処理終了の応答（ACK）メッセージをノード 0 に送る。

⑤ ノード 0 は、全てのノードから無効化要求を受け取ったら、i)自分の状態を D とし、ii)ビットマップを、要求元を残してクリアする（0001 となる）。次に iii)書き込み許可を知らせる応答（ACK）メッセージを要求元のノード 3 に送る。

⑥ これを受け取ったノード 3 は、キャッシュに書き込みを行い、自分のキャッシュの状態を D とする。以降、キャッシュの読み、書き共にホームメモリのお伺いを立てずに行うことができる。リプレイスされる場合は、ブロックをホームメモリに書き戻しを行わなければならない。

上記の処理はメッセージのやり取りが煩雑だが、基本的にスヌープキャッシュの処理と同じである。共有バスを持たないため、全ての共有ノードに無効化メッセージを送り、このための応答を集める操作が必要になる点が違うだけである。ここで他のノードが読み出し/書き込み要求を出した場合の処理もスヌープキャッシュと同様の方法で可能である。

３）D 状態のキャッシュに対してノード 2 が読み出し要求を出す場合（図 5）

①ノード 2 はホームメモリに対して読み出し要求を送る。

②ノード 0 はディレクトリを検索し、要求されたブロックが D 状態で、ビットマップより

ノード 3 が最新の状態を持っていることを知る。これに従い、書き戻し要求をノード 3 に送る

④ ノード 3 は、書き戻し要求を受け取り、要求されたキャッシュブロックを書き戻しメッセージに載せてノード 0 に対して転送する。

⑤ ノード 0 は書き戻しメッセージを受け取ると、i)これをホームメモリに書き戻す。ii)書き戻しされたブロックを要求元のノード 2 に転送する。iii)ブロックの状態を S とし、要求元のノード 2 とブロックの転送元のノード 3 のビットマップをセットする(0011)。さらにノード 3 に応答（ACK）メッセージを転送する。

⑥ ノード 2 は、キャッシュブロックを受け取り、自分の状態を S とする。ノード 3 は応答メッセージを受け取り自分の状態を S とする。

以降、ノード 2、ノード 3 は、ホームノードにお伺いを立てずに読み出しを行うことが可能となる。

これも、スヌープキャッシュの書き戻し処理と同じである。ではややしつこいが、今の処理では、ノード 2 は読み出し要求を出したが、これが書き込み要求だったらどうなるだろう。

3）D 状態のキャッシュに対してノード 2 が書き込み要求を出す場合（図 6）

先の操作のうち、⑤で、ノード 2 の要求が書き込みであることが分かると、ii)ブロックの状態を D として、要求元のノード 2 のみのビットマップをセットし（0010)、ノード 3 に応答（ACK）メッセージを転送する。

⑦ ノード 2 は、キャッシュブロックを受け取り自分の状態を D とする。ノード 3 は応用メッセージを受け取り、自分の状態を I にする。

この操作も基本的にスヌープキャッシュと同じであることがわかるだろう。スヌープキャッシュに比べるとディレクトリキャッシュの操作は多数のメッセージを伴い煩雑になる。しかし、複数のノードが同時に違ったホームメモリにアクセスを行う場合、処理が分散されるので、単一の共有バスに頼るスヌープキャッシュに比べてノード数の増加に強い。

では、同じキャッシュブロックに対して複数のノードから同時に要求があった場合はどうなるだろう。複数のノードからの要求は、ホームノードで受け付けた順に上記の処理が行われ、処理の途中で別の要求を受け付けることは行わない。ホームノード以外のノードもホームノードからの応答メッセージが来るまでに、そのキャッシュブロックにアクセスがあった場合は、待たせておくことで、基本的には安全に処理を行うことができる。しかし、メモリアクセスに対する全ての操作を応答メッセージが戻るまで待たせておくと、明らかな無駄が生じる。これについては、アクセスコンシステンシィ問題で扱うことにする。

4.1.2 ディレクトリ方式の性能向上

ディレクトリ方式はスヌープ方式と異なり、Clean Exclusive 状態を導入して無効化信号を省略することはできない。ホームメモリにメッセージを送ることは避けられないためで

ある。しかし、Ownership と同じ考え方で、ホームメモリに書き戻しを行わず、直接キャッシュ間転送を行うことは可能である。図 7 にこの様子を示す。D 状態のホームメモリに対して読み出し要求があった場合、ホームメモリは最新の状態を持っている D(Owned Exclusive)状態のキャッシュに対して、書き戻し要求を出す代わりに、直接要求元のキャッシュにブロックを送るように指示する。この方式により、ホームメモリに逐一書き込んで一致を取ってから転送する必要はなくなり、ブロックを送ったキャッシュは OS（Owned Sharable）状態になり、以降、ホームメモリとの一致の責任を持つ。要求を出したキャッシュは US(Unowned Sharable)になる。OS 状態への書き込みはホームメモリに要求を出して US 状態のキャッシュに対して無効化メッセージを発生してもらわなければならないし、リプレイスされる際はホームメモリに書き戻しを行う。

4.1.3 ディレクトリのコスト削減

ディレクトリ方式は、ホームメモリのブロック毎にディレクトリのエントリを設ける必要があり、その容量はホームメモリの容量に応じて大きくなる。さらに、単純なビットマップ方式は、システム内のノード数に応じて大きくなるため、ホームメモリが大きく、システムサイズも大きい場合にそのコストは膨大なものとなってしまう。
このため、ディレクトリを圧縮して持たせる方法がいくつか提案されている。

(1) リミテッドポインタ方式

ビットマップの代わりにノード番号を入れておく方法は、格納するノード数を限定すればコストを小さくすることができる。これが図 6 に示すリミテッドポインタ方式である。ブロックを共有するノード数は一般的なアプリケーションプログラムでは多くないので、少数のポインタで通常は間に合う。問題は数が足りなくなった場合で、以下の方法が提案されている。i)どれかを選んで無効化してしまう。ii)溢れたらメッセージを全ノードに対してブロードキャストする。両方ともやや乱暴な方法だが、溢れる頻度が少なければ性能低下はさほど大きくない。これに対して溢れた場合はノードに割り込みを掛けてソフトウェアに管理を任せる方法もある。これも性能低下は避けられないが、前者２つよりは小さい。

(2) リンクドリスト方式

ホームメモリからキャッシュに対してポインタを設けて、図 8 に示すようにキャッシュ間でリンクドリストを作ってしまう方式。ポインタはキャッシュディレクトリに状態と共に設ければ良いので、コストは小さくて済む。この方式ではメッセージをたどるのにノード間のメッセージ転送が必要で、時間が掛かるのが欠点である。また、キャッシュからブロックがリプレイスされる場合は、リンクドリストの繋ぎ変えが必要で手数がかかる問題もある。リンクをたどる時間を減らすためにツリー状に構成する方法も提案されている。

(3) ディレクトリキャッシュ方式

ディレクトリ自体をキャッシュしてしまう方式で、この場合、メッセージを制御するコントローラはホームメモリではなくディレクトリキャッシュに設ける。他のノードからのアクセスはディレクトリキャッシュに送られる。ホームメモリに対してディレクトリを設けてしまうと元も子もないので、何らかの形で圧縮して持たせる必要がある。

4.2　メモリコンシステンシィモデル

分散共有メモリ型並列コンピュータは、それぞれのノードのメモリをリモートにアクセスする必要がある。このため、メモリのアクセス自体をきちんとモデル化する必要性が生じた。

(1)シーケンシャルコンシステンシィ

これがメモリコンシステンシィモデルである。メモリコンシステンシィモデルでは、まず正しいアクセス、すなわちシーケンシャルコンシステンシィを定義している。図 9 は文献［］に掲載された有名な例である。プロセッサ 1（P1）は、A を 0 にし、一定の処理の後これを 1 にする。しばらく処理を行った後、B が 0 ならば L1 を実行する。一方 P2 は、B を 0 にし、一定の処理の後これを 1 にする。しばらく処理を行った後、A が 0 ならば L2 を実行する。シーケンシャルコンシステンシィは、L1 と L2 が同時に実行されることがないことを保証する。

このためには、P1,P2 両者が、①（書き込み→読み出し）、（書き込み→書き込み）、（読み出し→書き込み）の順序を変えない。二つの読み出し間の順序は変えても良い。②互いの書き込みが瞬時に共有メモリの反映される、の二つの条件を満足する必要がある。P1、P2 の書き込み、読み出しの順序が変わったり、値の書き込みが変数に反映されるのに少しでも遅延が入ると L1 と L2 を両方が実行してしまう可能性が生じる。もちろん、集中共有メモリでも分散共有メモリでも遅延なしでの書き込みは不可能だが、集中共有メモリではバスの書き込みが順番に行われることでこれを守ることができる。分散共有メモリでも、ホームメモリでアクセスを逐次化して、応答メッセージを使って関連するノードのメモリアクセスを待たせてやれば、シーケンシャルアクセスを守ることが可能である。しかし全てのメモリアクセスに対してこれを実現するのは損失が大きい。そもそも普通のユニプロセッサだってメモリのアクセスは、それが異なった番地に対するものならば必ずしも厳密に守ってはいない。

(2)Total Store Ordering(TSO)と Partial Store Ordering(PSO)

ユニプロセッサは、プログラムの順番を守って実行するインオーダー実行を行う場合であっても、書き込みを行ったデータがメモリに書き込まれるのを待たずに、書き込み書き込データをライトバッファに入れてしまって、次の命令を実行する。これが読み出しで、アクセスする番地が違えば、ライトバッファ中の書き込みを追い越してしまうことは普通にあり得る（番地が同じならばライトバッファから読み出しを行う）。この実装は（書き込み→読み出し）の順序関係を守らない（緩和した）ので、既にシーケンシャルコンシステンシィ

ではなくなってしまう。これを Total Store Ordering(TSO)と呼ぶ。TSO は IBM370 の昔から使われている。では書き込み間の順番は守る必要があるだろうか？これも番地が違えば特に守る必要はないだろう。（書き込み→書き込み）間の順序関係を緩和したモデルで Partial Store Ordering(PSO)と呼ぶ。TSO も PSO も、同期命令（ユニプロセッサの場合はプロセス間の同期）とメモリアクセス命令、同期命令同士の間ではシーケンシャルコンシステンシィが成立しなければならない。すなわち、全てのアクセスは同期命令が実行されるまでに終了していなければならない。しかし、このルールを守れば、それぞれの条件で緩和してもメモリは正しく動作する。

(4) ウィークコンシステンシィ（弱いコンシステンシィ）

アウトオブオーダ実行のユニプロセッサやマルチコアでは、書き込みが読み出しを追い越してしまうことも普通にあり得る。すなわち、全てのメモリアクセスは番地が異なれば任意の順番で実行しても差し支えないのではないか。もちろん、同期命令とメモリアクセス命令、同期命令同士の間ではシーケンシャルコンシステンシィが成立している必要がある。ホームメモリのコントローラで、上記のシーケンシャルコンシステンシィを実現し、共有メモリを用いたデータ転送では必ず同期操作を用いることにすれば、分散共有メモリのアクセス制御はこのウィークコンシステンシィに基づいて行うことができる。すなわち、メモリアクセス間の順序はアドレスが違えば守る必要はない。このことにより、各ノードは、ホームメモリからの応答メッセージを待たずに、次のメモリアクセスに移ることができる。

(5) リリースコンシステンシィ

同期操作は、一般的に、3 章に紹介したように、排他制御を行ってクリティカルセクションに入る部分（Aquire：獲得と呼ぶ）と、クリティカルセクションを解放して、他のプロセッサ（プロセス）にその利用を許す部分（Release：解放と呼ぶ）から成っている。これを分けて管理することにより、複数のクリティカルセクションの並行実行が可能になる。すなわち、新しいメモリアクセスは Aquire が終了してから発行し、Release はその終了を待って行うとする。このような制約を行うことで図 10 に示すように複数の互いに関連のないクリティカルセクションを並列実行することが可能になる。このコンシステンシィモデルは、Release 時までに全てのアクセスを終了させる必要があることからリリースコンシステンシィと呼ばれる。CC-NUMA では同期の実装上、このリリースコンシステンシィに基づいてメモリのアクセスを行っているものが多い。