1　はじめに

近年の並列計算機の発展は目ざましく、小規模、中規模のシステムは既に商用化の時代を迎え、数千から数万プロセッサから成る大規模並列、超並列計算機の研究開発が盛んである。大規模な並列計算機の多くは、単一または複数のプロセッサとメモリをバスで結合してノードを形成し、このノード間を高速の結合網で接続した構成を持つ。

しかし、システムが大規模になるに従い、システム開発時、稼働後のメインテナンスが困難になる。システム開発時には、ハードウェアがまだ安定動作に至っていない時のプログラムのロード、デバッグが困難で、システムソフトウェアの開発に多大な負担がかかる。また、システムが稼働した後の故障診断、回復の必要性も大きくなる。

これらの機能を果たすためには、対象とするシステムと独立した、低コスト、高信頼性のサブシステムが必要になる。本論文ではこのような機能を持つ並列計算機サブシステム UBA(Universal Backup Architecture) を提案し、主にそのデバック機能について述べる。

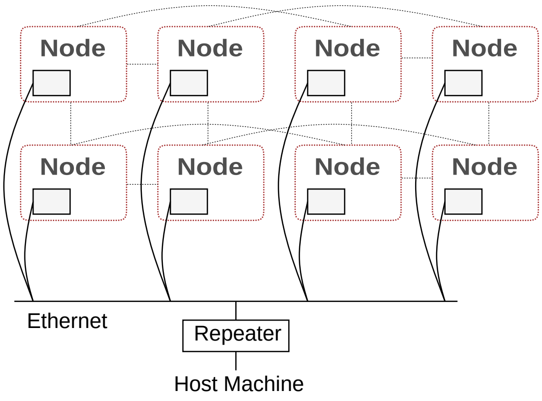

図 1: UBA システムの構成

2　UBA の概念

UBA は、図 1に示すように、対象システムと独立のネットワークにより接続されたプローブユニットから構成される。このネットワークは、現在は簡単な Ethernet を考えており、リピータを通してホスト計算機のネットワークに接続する。UBA の構成要素となるプローブユニットは簡単なマイクロプロセッサシステムであり、対象システムのノードのバスに接続される (図 2)。接続は対象システムの全ノードに対して行なうのが理想だが、プローブユニット自体やサブネットワークのコストが増大するため、システムに応じて数を調整する必要がある。

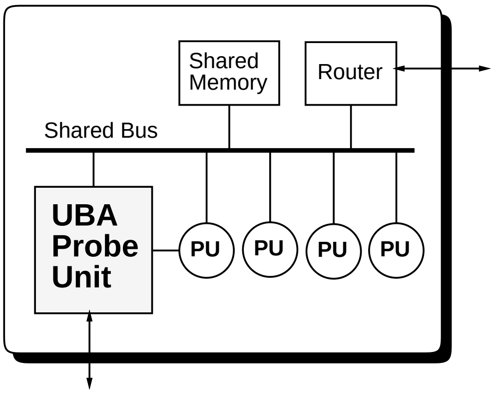

図 2: プローブユニットの接続

UBA は以下の機能を実現する。

- システム開発時
    - テストプログラムロード、簡単なモニタ
    - プログラムデバック支援
- システム稼働時
    - 故障診断、回復指令
    - 性能モニタ
    - OS を介さないプログラムのデバック支援

従来にもメインテナンス用システム [1]、モニタ用システム [2][3] が提案されているが、UBA は対象とするシステムとの独立性を高め、故障診断、回避、モニタだけでなく、プログラムロード機能、デバック支援機能等を持つ総合的なシステムである。

本論文ではデバッグ支援機能について検討し、UBAのプローブユニットに必要な機能を提案する。

今回はUBAのシステムのうちプローブユニット単体で使用できる機能について述べる。またターゲットとなる大規模並列計算機のノードとしてATTEMPT-0[3]を想定する。

3 プログラムデバッグ支援機能

並列計算機は並列度の高い計算を高速に行うために使用される。このような計算を行う場合、OSを介さずにプログラムをじかに記述し高いパフォーマンスを得ようとすることが多い。また、計算機の立ち上げ時にもOSが存在しないことが多いため、プログラムは直接記述される。

以上のようなプログラムではOSの機能を利用してデバッグすることができないためOS上で動作する一般的なデバッガではデバッグできない。UBAではこのように計算機上で直接実行するプログラムのデバッグを支援するために「共有メモリブレークポイント」とその拡張機能である「条件チェック機構」を提供している。ユーザはプローブユニットに対して`%condition`を使用して停止条件を記述するすることで、プログラムが誤動作したときにプロセッサを停止させ、データの検証を行える。

ここではまず、以後の解説のためC言語のプログラム例を図3に示す。このプログラムはキューを使用して複数のプロセッサが協調作業するプログラムの典型的な一例である。プロセッサ間で共有されるデータは、キュー(プログラム中に変数としては現れていない)、終了宣言を示すフラグ(`end_flag[]`)、バリア同期を行う際に使用されるカウンタ(`barrier_count`)である。

このプログラムの主要部分(並列に処理される実質的な部分)は関数`work()`である。この関数では以下の作業を終了条件が満たされるまで繰り返す。

1. キューからデータを取り出す(`get_data()`)
2. もし、キューが空であったら、
    (a) 「自分は終了した」と宣言し、
    (b) 他人も終了したかチェックをする。(`check_end()`)
    (c) もし、全員終了しているならば、ループを抜けプログラム終了。
3. キューからデータが取り出せたら、
    (a) 「自分は作業中である」と宣言し、
    (b) 実際の仕事(`real_work()`)を行う。
    (c) この仕事を行った結果、別のプロセッサのキューにデータを追加する必要があるならば、関数`put_queue()`を使用してキューに追加する。

このプログラムは3つのプロセッサ(プロセッサ番号0~2)で実行されるものとする。

3.1 共有メモリブレークポイント(1)

図3では、バリア同期のための変数`barrier_count`をマスタ(`I_am_master`が真であるプロセッサ)が初期化している。しかし、この初期化が済む前にスレーブの実行が関数`barrier()`へ移ってしまうと、`barrier_count`の値が正しくないため、スレーブはバリア同期を素通りしてしまい、結果としてプログラムが誤動作する可能性がある。

この場合、バリア同期がうまく動作していないことがわかれば、デバッグのための変数`barrier_count`をチェックする必要が生ずる。

このようなケースに対処するためにUBAでは共有メモリに存在しているデータに対するブレークポイント—UBAの持つ共有交信装置用のモニタリングシステムを使用して共有メモリへのアクセスを監視し、指定されたアドレスに対してアクセスが行われた場合に全プロセッサの処理を停止させる機能を持つ。

```
proc()
{
  if( I_am_master )
    initialize();
  barrier();
  work();
}

initialize()  /* 初期化 */
{
  barrier_count = 0;
  /* initialize queues */
  /* initialize end_flags */
}

barrier()  /* バリア同期 */
{
  barrier_count++;
  while( barrier_count < PU_NUM )
    ;
}

work()  /* 処理の中心 */
{
  while( 1 ) {
    if( get_data( &data ) == EMPTY ) {
      end_flag[my_puid] = TRUE;
      if( check_end() == TRUE )
        break;
    }
    else {
      end_flag[my_puid] = FALSE;
      if( real_work( data, &new, &puid )
        == CREATE_NEW_DATA )
        put_queue( puid, new );
    }
  }
}
```

図 3: 誤動作するプログラムの例

この例の場合、ブレークポイントは図 4のように記述する。

`%condition` ブロックは後述する条件チェックを記述するものだが、特に条件を記述しなければ、指定した変数 (この場合は `barrier_count`) をアクセスした場合にプロセッサを停止させる。

```
%condition barrier_count {
    ;
}
```

図 4: ブレークポイント (1)

3.2 共有メモリブレークポイント (2)

ある変数にアクセスした場合に必ず停止させるならば、図 4の記述で良いが、あるプロセッサがアクセスした場合にのみ停止するなら図 5のようにする。

この例ではアクセスしたプロセッサの番号を返す関数 %pu() を使用して条件を記述している。

```
%condition barrier_count {
    %pu(barrier_count) == 1;
}
```

図 5: ブレークポイント (2)

3.3 条件チェック機構 (1)

複数の変数の間である条件が成り立つとするなら、その条件を `%condition` ブロックに記述することでチェックすることができる。

図 3では終了判定を配列 `end_flag[]` によって行っている。しかし、この変数はキューからデータを取り出すという作業をした後で更新されるためタイミングによっては全ての仕事が終了しないうちにプログラムが終了してしまうことがある。図 6はこの条件を示している。PU 0, PU 2 のキュー

にはデータが無く、PU 1 のキューにのみデータがあるとする。PU 1 はキューからデータを取り出し、処理を行う。PU 1 の行なった処理の過程で PU 0 に送るデータが発生したため、PU 1 が PU 0 のキューにデータを入れたとする。その後、PU 0 が `get_data()` を行って `end_flag` を更新するが、その操作より前に、PU 1, PU 2 が終了判断してしまうと、全プロセッサの `end_flag` が `TRUE` なので PU1, PU2 は終了してしまう (図中の網掛け部分)。しかし、PU 0 の処理によっては PU 1, PU 2 に対して新たにデータが生成される可能性があるため、本来この時点で終了すべきではない。

この場合、チェックすべき条件は「全ての `end_flag[]` は `TRUE` であるが、キューにはデータが存在している」というものである。この条件は図 7のように記述できる。ここで、`queue[i]` はプロセッサ番号 i のキューにデータが入っているかどうかを示す変数である。具体的には、キューが FIFO で実現されているなら FIFO に存在しているデータの個数、またリストによって実現されているなら、先頭のデータを指し示すポインタなどに対する条件を記述する。

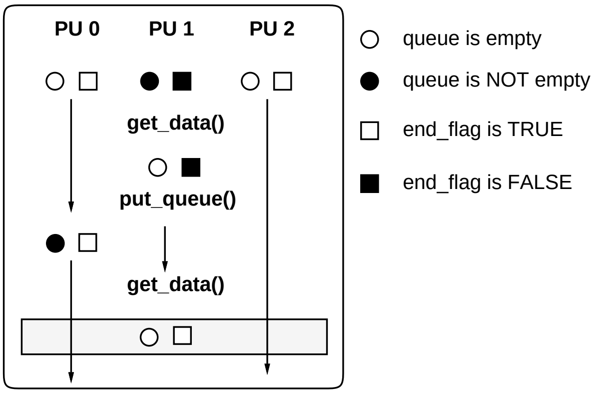

図 6: 終了判断のミスの例

3.4 条件チェック機構 (2)

ある変数は単調増加すると仮定できるなら、その条件を `%condition` ブロックに記述することで条件を満たさないようなアクセスがあった場合にプロセッサを停止させることができる。

図 3においては `barrier_count` を更新する際に排他処理をしていないために 2 つ以上のプロセッサで同時に変数を更新してしまい、結果としてバリア同期から抜けられなくなる可能性がある。この場合、変数 `barrier_count` は「必ず直前の値より大きくなる」という性質を利用し、`%condition` ブロックを図 8のように記述すれば、誤動作を検出することができる。`%condition_write` ブロックにより変数へ書き込みのアクセスが起きた場合にのみ条件判断を行う。

`%history` ブロックでは履歴を取る変数を指定する。この例では変数 `barrier_count` の履歴を過去 1 回分保存することを指定している。履歴は条件判断が行われると一つ古くなっていくものとする。すなわち、現在の値が添え字 0 に格納され、条件判断の直前の値が添え字 -1 に格納される。

```
%condition
  end_flag[0] end_flag[1] end_flag[2] {
     (end_flag[0] && end_flag[1]
      && end_flag[2]) &&
    ! (queue[0] && queue[1] && queue[2])
}
```

図 7: 条件記述例 (1)

```
%history {
   barrier_count#1;
}

%condition_write barrier_count {
   barrier_count#0 <= barrier_count#-1;
}
```

図 8: 条件記述例 (2)

以上の停止条件はコンパイラによってプローブユニットが読み込む形式に変換される。その際、ターゲットプログラムを指定することで変数名からアドレスへの変換も同時に行われるので、条件の記述に変数名を使用することができる。

4 プローブユニットの構成

3章で解説したプローブユニットのデバッグ支援機能を実現するためには、以下のようなプリミティブな機能が必要となる。

1. ノード内の結合網を監視し、アクセスされたアドレスを取得する機能
2. ノード中の全プロセッサを停止させる機能
3. アクセスをしたプロセッサの番号を取得する機能
4. 共有データに対してアクセスを行う機能
5. 指定されたデータの履歴を保存する機能

これらの機能のうち、5についてはソフトウエアで実現できるが、それ以外の機能についてはハードウエアによるサポートが必要となる。

また、ブレークポイントとアクセスされたアドレスの比較はソフトウエアでも可能だが、パフォーマンス向上のためハードウエアによって行われる。これは速度が必要な処理をハードウエアによってサポートすることで、できる限りターゲットマシンに干渉しないようにするためである。

プローブユニットの構成を図9に示す。共有メモリブレークポイントはファイルまたは簡易モニタによってプローブユニット内の「ブレークポイントレジスタファイル」(以下BPRF)に登録される。アクセスされたアドレスがこのBPRFに存在するならばプロセッサは停止させられる。その後、簡易モニタに制御を移し、ユーザからの入力を待つ。

`%condition`ブロックが空ではないブレークポイントはアドレス部と条件部を分けて記憶する。アドレス部は共有メモリブレークポイントと同じようにBPRFに登録する。条件部の判断はBPRFにヒットした場合にのみ行う。判断の結果、偽である場合プロセッサの処理は継続される。

履歴の保存は`%history`ブロックで指定した個数だけ行なわれるので、実行中にメモリが不足する事態は生じない。

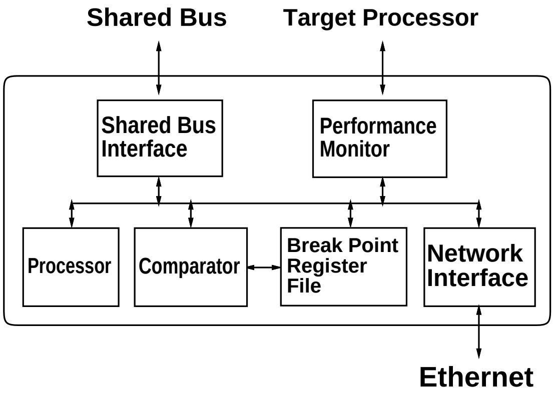

図9: プローブユニット構成図

プローブユニットは二種類のインタフェースにより対象とするノードと接続される。ひとつは共有交信装置用のモニタであり、対象システムの共有バスまたは交信ネットワークに接続される。

もうひとつのインタフェースは、接続されたノードのプロセッサのプライベートな情報をモニタするために設けられ、そのプロセッサのプライベートキャッシュに関する情報、エラーに関する情報、共有資源への要求等をモニタする。このインタフェースは対象とするプロセッサ側から端子を付けてとり出す必要がある。

プローブユニットの対象システムへの接続方法はシステムにより異なる。今回は対象とするシステムがIEEE Futurebus P896.1 (1987)[4]を用いたマルチプロセッサテストベットATTEMPT-0であるため、現状ではFuturebusに標準化しており、それ以外のシステムでは変換ハードウェアが必要である。

通常、このプローブユニットはFuturebus上のアドレス、データ、コマンド、ステータスのモニタにより、統計情報とエラーの検出を行なうが、それ自身Futurebusのモジュールに成る能力を持っており、以下の機能を実現することができる。

- アービトレーションメッセージにより、システムの全プロセッサに対する割り込みをかけ、処理の停止を要求する機能
- バスを介して共有メモリをアクセスする機能

アクセスするプロセッサはアービトレーションの過程で認識できるので、Futurebusでは前記のハードウエアに対する要求を満すことができる。

UBAのプローブユニットは以上のような構成となるが、要求される機能の確認とテストを行うため、現在は並列計算機シミュレータMILL上に実装されている。

5 並列計算機シミュレータ MILL

MILL (Multiprocessor Instruction Level simu-Lator) [5] は並列計算機をプロセッサのインストラクションレベルでシミュレートする、execution driven 型のシミュレータである。現在は、ATTEMPT-0[3]の形態に合わせた構成となっており、ATTEMPT-0用に開発されたアプリケーションが、再コンパイルするだけでMILL上で実行可能である。

MILLでは、プロセッサ部、キャッシュ部、同期機構部、バス及び共有メモリ部、時刻管理部の各部がそれぞれ独立したスレッドとなって、並行に実行される。現在は、Sun Microsystems社のLightweight Processesライブラリを用いてSPARC Station上に実装されている。

5.1 MILLの動作の概要

プロセッサ部はMotorola社68000シリーズのCPUをシミュレートする。これは、ATTEMPT-0がMC68030を使用しているためであり、他のCPUに置き替えることも可能である。

プロセッサ部はコンパイルにより生成されたオブジェクトコードを読み込み、実行を開始する。各プロセッサはコード領域を共有している他は、レジスタなどのCPU資源やスタック、データ領域、ローカルメモリを各々独立に持つ[1]。

共有データへのアクセスはキャッシュメモリを介して行なわれる。キャッシュメモリは、各プロセッサに対応して個別に存在し、担当プロセッサからのアクセスに対してサービスを行なう。キャッシュコヒーレンシプロトコルはBerkeleyプロトコル、Dragonプロトコル等から選択可能で、ブロックサイズや、アソシアティブ数なども設定できる。

同期機構としてはATTEMPT-0の同期機構である、シンクロナイザ[6]を実現している。シンクロナイザは受信者選択型のマルチキャストを行なう少量の同期メモリであり、ATTEMPT-0同様、各プロセッサに分散して実装されている。シンクロナイザへの書き込みは、常にバスを介して放送されるが、読み出しは、各プロセッサが個別に行なうことが出来る。ATTEMPT-0 (及びMILL) 上のアプリケーションプログラムでは、このシンクロナイザを用いて排他制御やバリア同期などを実現している。

キャッシュコヒーレンシ維持のためのバスアクセス (例えば無効化など) や、共有メモリアクセス、シンクロナイザによるデータの放送などを行なう必要の生じた場合は、バスを介して処理を行なう。バスはATTEMPT-0で採用したFuturebusをモデル化しており、アービトレーションも行なう[2]。

以上の各部の制御を行なうのが時刻管理部である。キャッシュリクエスト、バスリクエストなどの

---

[1] 現在のところ、on-chip キャッシュ、ローカルメモリに対するキャッシュなどはサポートされていない。

[2] 現在は、簡素化のためフェアネス機構は省略されている。

タイムスタンプから実行順序を決定し、プロセッサ間の相互作用を実現する。

6 MILL 上への実装

MILL 上に UBA を実装する際には、システム駆動時に使用される故障診断、回復指令および性能モニタについては実装を行わない。これらの機能はハードウエアによって実装されなければ意味を持たないので MILL 上への実装は見送った。

そこで、今回はプログラムのロード、簡単なモニタおよび OS を介さないプログラムのデバッグ支援機能を MILL 上に実装した。

4章で述べた機能について検討すると全て共有バス上に出力されるデータに対してのアクセスが行えれば実現することができる。

そこで、MILL のモジュールのうち共有バスに関係するモジュールである「バス及び共有メモリ部」を変更することで実装を行う。ただし、共有メモリからの読みだしはキャッシュメモリにヒットして共有バスまでアクセスが来ない可能性があるため「キャッシュ部」についても変更をした。

以上、MILL 上に実装を行うことでプローブノードの機能の確認とテストにかかる時間を短縮することができた。

7 おわりに

本論文ではデバッグ支援にターゲットを絞って考察し、MILL 上で実装を行った。今後はこの結果に基づいてプローブユニットハードウエアの試作を行い、ATTEMPT-0 への実装を行う予定である。また、プローブユニット間の結合や、故障診断などの UBA システムの他の部分についても、研究を進めていく予定である。

参考文献

[1] Hiraki, K. Nishida, K. Sekiguchi, S. Shimada, T. : *Maintenance Architecture an its LSI implementation of a dataflow computer with a large number of processors,* Proc. of the 1986 International Conference on Parallel Processing, pp.584-591, Aug. 1986.

[2] 白川、油谷、丹波：並列,分散システム用ハードウェアモニタの構成について, 情報処理学会論文誌 Vol.32 No.8, pp.981-990, Aug. 1991.

[3] 鳥居、竹本、天野、小椋：バス結合型並列計算機の交信用メモリの性能評価, 情報処理学会論文誌 Vol.33 No.3, pp.307-319, Mar. 1992.

[4] IEEE Standard Backplane Bus Specification for Multiprocessor Architectures: Futurebus, IEEE, 1987.

[5] 寺沢、天野 ： バス結合型並列計算機のプロセッサ間交信評価システム, 情報処理学会研究報告 Vol.91 No.64, pp.185-192, Jul. 1991

[6] Amano, H. Terasawa, T. Kudoh, T. : *Cache with Synchronization Mechanism,* Proc. of the IFIP 11th World Computer Congress, pp.1001-1006, Aug. 1989.