1. はじめに

現在、商用化され普及しつつあるバス結合型マルチプロセッサは、プロセッサ性能の向上にバスの転送能力が追従できないことから、既にその規模の限界が見え始めている。このため、さらに多数のプロセッサにより大規模な並列処理を行なうために、様々な方法が試みられている。DASH[1], Alewife 等分散共有メモリ型マルチプロセッサに代表される NUMA(Non-Uniform Memory Access model) は、大規模化の有力な候補であるが、分散共有メモリの一貫性を維持するためのハードウェアコストが大きい。

これに対し、複数のプロセッサで共有するメモリを持たず、ローカルメモリを持つプロセッサ同士がメッセージ交換により処理を進める NORA(Non Remote Access Memory model) は、大規模な構成を容易に実現できるが、共有メモリが存在しないため、用途が制限される。ソフトウェアにより仮想的に共有メモリを実現する方法も試みられているが、効率の良い処理は困難である。

一方で、各プロセッサがローカルメモリを持つと共に、スイッチまたはバス等の結合媒体で直接アクセス可能な共有メモリを持つ構成も、大規模マルチプロセッサの有力な候補である。この構成は、NYU Ultracomputer[2] を始めとして古くから検討され、実装が試みられているが、一般に普及するに至っていない。これは、ひとつには、共有メモリとローカルメモリを効率良く運用するための OS が確立していないことに原因があると考えられる。

そこで、本論文では、ローカルメモリを持つ共有メモリ型並列計算機のための OS である LC を提案し、設計・評価を行う。LC の実装は、ローカルメモリを利用することにより高速なプロセス制御を実現する。また、プログラム記述時またはコンパイル時に配置場所を決定できるデータタイプに関しては、それぞれ適切なメモリに割り付けを行ない、配置場所を決定できないデータタイプのために新たにデータが双方のメモリシステムを移動する mobile 型を用意している。

以下、本稿では、第 2節でターゲットマシンについての詳しい検討を行い、第 6節で LC の設計と実装について述べ、第 7節で mobile 型の設計と実装方法について述べる。

2. ターゲットマシン

LC は (図 1) に示すように、ローカルメモリを持ったプロセッサがスイッチ等の結合網により共有メモリに接続された構成を持つマルチプロセッサをターゲットとする。プロセッサから見て、共有メモリとローカルメモリは分離されたアドレス空間上に存在する。他のプロセッサのローカルメモリ上のデータを、共有メモリを介さずにアクセスすることはできない。同期は共有メモリ上の変数によって行なうが、プロセッサ間に割り込みを生じる機構は設けられている。このような構成を持つマルチプロセッサは、結合網にクロスバを用いた C.mmp をはじめとして、MIN(Multistage Interconnection Network) を用いた NYU Ultracomputer、RP3[3]、Butterfly[4] 等多くの実験機、商用機が開発されており、本理工学部でも、特殊な MIN である SSS-MIN を用いた SNAIL[5] が稼働している。

このような構成では、共有メモリとプロセッサ間には複数のパスが存在し、同時にいくつかのメモリモジュールをアクセスすることができる。また、プロセッサと結合網の間にはキャッシュが存在し、共有メモリ上のデータを置くことができる。しかし、この構成では、結合網は単純なバスではないので、スヌープ操作により一貫性を維持することはできず、キャッシングの制御はソフトウェアで行なわれる。

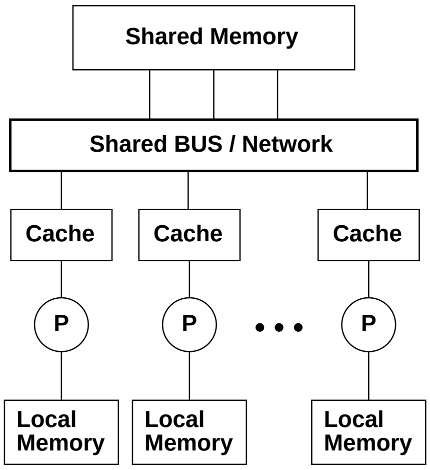

図 1: LC がターゲットとしているマシンの例

このようなシステムは、共有メモリに対する同時アクセスの高速性を生かして大規模な単一ジョブをシステムを占有して実行する使われ方が多い。このため、C.mmp を除くほとんどがバックエンドのマシンとして用いられている。したがって LC もバックエンドでの利用を念頭に置く。バックエンドマシンの OS ではレスポンスタイムが最大の評価要素となり、LC もこの点に関する最適化を行なう。

3. LC の設計方針

現在までに多くの並列オペレーティングシステムが実装されているが、その多くは主記憶共有型、フロントエンド、マルチユーザを対象としている。これらの OS の多くでは、ローカルメモリを効率良く用いることができず、プロセス生成、スレッドの切替えのコストも大きい。一方、ネットワークで接続された分散システムを念頭に置き、ローカルメモリとプロセス間交信に基づく分散 OS も様々な種類が提案されている。しかし、これらの OS は、共有変数を用いて頻繁に交信するアプリケーションを指向していないため、共有メモリを効率良く用いることができない。

そこで、我々は、ローカルメモリと共有メモリを持つ大規模バックエンドマルチプロセッサをターゲットとし、シングルジョブの高速化を目指した軽量の OS である LC を開発することにした。LC の設計方針は以下の通りである。

- ローカルメモリと共有メモリをそれぞれ最大限に利用する。このために、コンパイラまたはユーザの指示によるデータ配置および mobile 型のデータを導入する。
- コンテキストスイッチングを最大限に高速化する。
- 共有メモリ、ローカルメモリの階層化、仮想化を行なう。
- 必要最低限の機能のみを持ち実装、移植が容易な構造を持つ。

4. ジョブ・プロセス・スレッド

まず、OS の処理する単位として、ジョブ、プロセス、スレッドを定義する。

ジョブ：1 つのアプリケーション・プログラムに相当する処理単位。同一ジョブ内ではグローバルメモリを共有。

プロセス：個々のプロセッサに割り当てる処理単位。同一プロセス内では全メモリを共有。

スレッド：実行主体。生成された後は同一のプロセス内で実行を行う。

LC の目的を単一ジョブの高速化とし、当面はマルチジョブを考えないこととする。

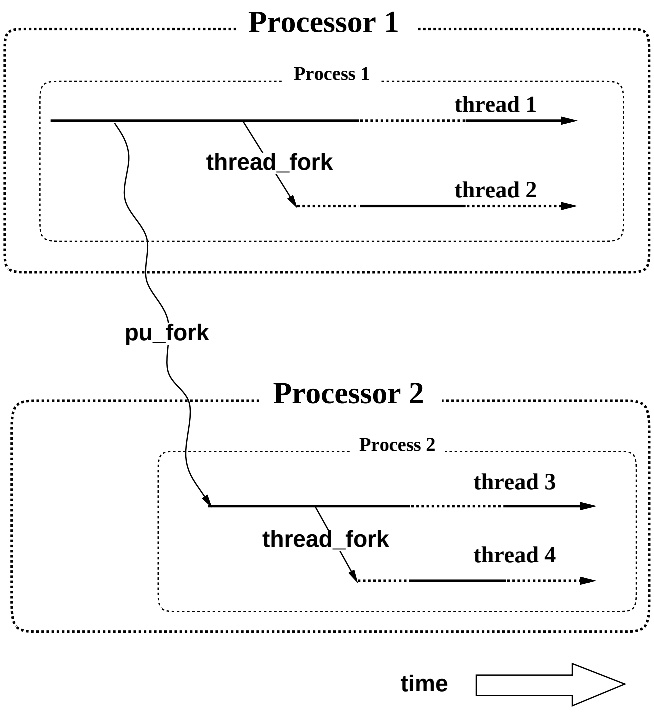

図 2: プロセス、スレッドとプロセッサの関係

プロセスは並列に処理される単位である。プロセッサ数が少ない場合などは 1 つのプロセッサに複数のプロセスを配置する場合もある。しかし、これは 8 プロセッサ用に作成されたプログラムを 8 つ未満のプロセッサしか持たないシステムで動作させる場合などに起きることで、通常 LC では最大プロセッサ数までのプロセスを生成するようにプログラミングすることを前提とする。このように各プロセスは別プロセッサで実行されるため、プロセス間ではローカルなデータは共有されない。

主記憶共有型の OS では、負荷の軽いプロセッサがレディー・キューに入っているプロセスを実行することが多い。そのため、あるプロセスは休止状態に入る前と次の実行では違うプロセッサに割り当てられ処理される可能性がある。しかし、ローカルメモリを持つマシンをターゲットとした場合、このようなプロセス・マイグレーションを行うと、コンテキストスイッチの度にローカルメモリの内容を移動する必要が生じる。これはローカルメモリ・共有メモリ間のデータ移動であるため非常に高価な処理である。さらに、LC では 1 プロセッサにつき 1 プロセスが基本であるため、負荷分散のためにプロセス・マイグレーションは効果がないと考えられる。そのため、LC では、あるプロセスは生成から消滅まで同じプロセッサ上で動作する。

スレッドはプロセスをさらに細かく分けた処理単位である。これはプログラマが処理を記述する上で並行動作として記述する方が良いと思われる時に使用する。同一プロセス上のスレッド (図 2では thread 1 と thread 2 など) は並列に処理されないため、並行動作を行うためにはコンテキストスイッチを必要とする。頻繁なコンテキストスイッチはオーバヘッドをまねくため必要最低限に抑えることが必要となる。そのため、LC ではタイマの割り込みによるタイムシェアリングは行わず、ロック獲得失敗時や入出力待ち、ユーザによる明示的な指示が行われた時にのみコンテキストスイッチを行う。また、ウォッチドッグタイマによるコンテキストスイッチも行う。これはコンテキストスイッチがないために起こるハングアップを防止するためのものである。

5. プログラミング・スタイル

LC では、ローカルメモリ空間、及びグローバルメモリ空間をユーザに解放するので、ユーザは各メモリ空間を意識したアプリケーションに最適なデータ配置を行うことができる。

並列計算機でシミュレーションや数値計算などを行う場合は、1. 変数の初期化などの前処理、2. 実際の計算・処理、3. 結果の出力などの後処理、の手順を踏むことが多い。そのとき、前処理と後処理はあるプロセスのみが行うが、2 番の実際の計算は複数のプロセスで並列に行う。

このスタイルでは、最初唯一のプロセスが前処理を行うが、このプロセスをマスタ・プロセスと呼ぶ。マスタ・プロセスは前処理を行った後、必要な数だけプロセスをフォークし、そのプロセスが実際の計算・処理を行う。その後、結果の集計を行うために同期を行う。同期後はマスタ・プロセス以外のプロセスは終了し、最後にマスタ・プロセスが結果の出力を行う。

LC ではこのスタイルを支援するために、ジョブの処理の開始時はホストマシンと接続されているプロセッサにマスタ・プロセスを作成、実行する。また、プロセスのフォークのためのシステムコールと、同期のための関数群をサポートする。

5.1. ライブラリ化

LC では、OS 本体はシステムコールとして最小限の機能だけを提供し、あとはライブラリとして提供することにする。これは、システムコールにより生ずるオーバヘッド (例えば、ソフトウエア割り込みに伴うレジスタの退避など) を避けるためである。ここで、例えば排他処理に必要なロックにはロック獲得を試み失敗したら、1. 他のスレッドにコンテキストスイッチを起こす、2. もう一度試みるの 2 つの方式が考えられる。システムコールとしてどちらか一方の方式でロック全体が実装されていた場合、ユーザはその方式でしかロックを使用することができない。これに対し、システムコールでは、ロックを試み成否を返す、コンテキストスイッチを起こす、の 2 つをサポートし、ロック自体はライブラリで提供するならば、ユーザはライブラリを変更することで、どちらの方式を用いることが可能で、さらに複雑な方式でも使用することができる。このように、ライブラリとして処理を実装することでシステムの柔軟性を向上させることができる。

6. LC カーネル

本システムのサポートを行うために現在カーネルの実装を行っている。

図 3にジョブ・プロセス・スレッド間のメモリ空間の共有関係を示す。

ジョブは、1 プロセス・1 スレッドで起動され、プロセスの生成により並列実行を行う。また、スレッドは生成されたプロセスに属し他のプロセスへの移動は行わない。

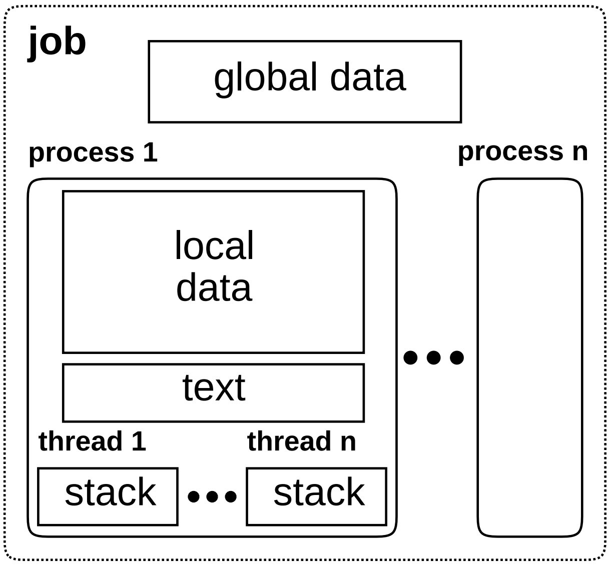

図 3: メモリ空間の共有関係

本実装では、原則的にはプロセスは PU に 1 対 1 で割りつけるものとする。

本カーネルの機能をグローバルメモリ空間とローカルメモリ空間さえ持てば容易に実装可能とするために、出来るかぎり特殊な機能 (例: 指定 PU 郡への PU 間割り込み等) を使用せずに実装を行っている。

6.1.　カーネル機能

ここでは、本カーネルの機能及び、それを実現するシステムコールについて述べる。

本カーネルでは、

- メモリ管理
- ジョブ管理
- プロセス管理
- スレッド管理
- 同期

等の最低限の管理を行う。また、カーネルモードでしか行えない部分を除きライブラリを用いユーザモードで行うこととし、システムコール発効に付随するオーバーヘッドの低減を図る。

6.1.1.　メモリ管理

システムコールを用いずにスレッド等の管理情報等の情報を取得するために、ユーザ空間とカーネル空間を同一空間にマッピングする。したがって、ユーザ空間とカーネル空間はそのアドレスにより分割する。そのため、ユーザ空間とカーネル空間を同一の物理メモリにマッピングするため MMU を使用しメモリ空間の仮想化を行う。

グローバルメモリ空間は全 PU で同一のメモリマップとするために、アドレス変換情報を PU 間で共有する必要がある。

また、物理記憶以上のデータを扱うためメモリのスワップを行う。全ての PU に 2 次記憶がついている事は稀であろうから、ローカルメモリのスワップエリアとしてグローバルメモリを使用し、グローバルメモリのスワップエリアとして 2 次記憶を使用する。

6.1.2.　ジョブ管理

LC はシングルジョブであるため、ジョブの生成・消滅の処理のみを行う。

ジョブの生成は、カーネルによりプログラムをメモリへロードし、register・スタック等を初期化し、ユーザモードに移行することで行う。

また、ジョブは全プロセスの終了もしくは、`exit` システムコールによる明示的な終了により消滅する。

6.1.3.　プロセス管理

本実装では、1 プロセス/1PU であるから生成・消滅のみを扱う。生成は `pu_fork` システムコールにより行う。`pu_fork` は、そのプロセスのトップレベルの関数 `fnc` と `fnc` の引数 `argument` を引数として指定する。`fnc` へはサイズがポインタ型のもの一つのみが引数として受渡し可能である。プロセス生成時には、`pu_fork` システムコールを呼び出した PU のローカルメモリ空間の内容をプロセスの生成先 PU へコピーする。また、生成時点でプロセスの割り当てられていない PU にプロセスを割り当て、全 PU にプロセスが割り当てられていた場合プロセスの生成は失敗する。[1]

また、そのプロセス内の全スレッドの終了をもってプロセスの終了とし、現段階では明示的にプロセスを終了させる関数は用意していない。

6.1.4.　スレッド管理

LC ではスレッドの生成・消滅・停止をサポートする。

生成は `thread_fork` システムコールを用いる。`thread_fork` システムコールの引数は `pu_fork` と同様である。[2]

スレッドはそのトップレベルの関数の終了及び、`thread_exit` システムコールにより消滅する。

`thread_suspend` システムコールにより、スレッドを停止することが可能である。`thread_suspend` は、スレッドの再開要因となる変数のポインタと値を引数とし、スケジューリングの際にその変数が指定した値と一致した場合スレッドは実行を再開する。また、スレッドを指定し即時再開させるようなシステムコールは、そのスレッドが同一 PU 内にない場合、PU 間割り込みを必要とするためサポートしない。

スレッドの切替は、`thread_yield` により明示的に記述された場合に行う。

ウォッチドッグタイマに関しては、全 PU にタイマーがない可能性もあり、この場合 PU 間割り込みによりスレッド切替を行うことは可能であるが、全 PU で同時にスレッドを切替えた場合、スタックの変更によりページフォルトが起こりスワップエリアである、グローバルメモリ空間へのアクセスが集中する可能性がある。これを回避するために、PU 個々の時刻を時刻管理を行える PU で行い、スレッド切替のタイミングをずらさせた場合、PU 間の負荷バランスに問題が生じることが考えられる。

スレッドのスケジューラは、ユーザにより指定可能とし、指定がない場合 Round-Robin となる。

---

[1] `pu_fork` はプロセス生成が成功すると 1、失敗すると 0 を返す

[2] スレッドの管理構造体へのポインタを戻り値とし、生成に失敗した場合 0 を返す

6.2. 並列計算機 ATTEMPT-0 上での実装

LC は、現在バス結合型並列計算機 ATTEMPT-0[6][7] 上でカーネルの実装を行っている。

ATTEMPT-0 は図 4に示すように最大 20 枚の PU ボードを Futurebus[8] を用いて接続することが可能なバス結合型並列計算機である。

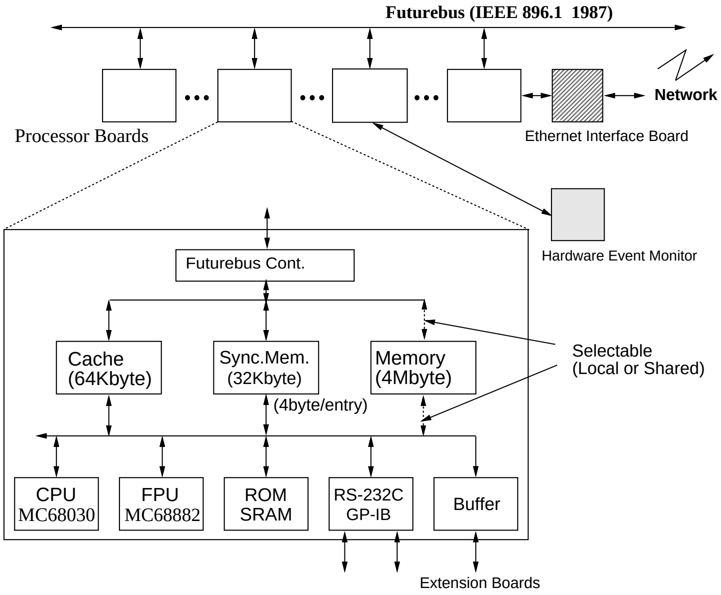

図 4: ATTEMPT-0 の構成

各ボードには MC68030 と 4Mbyte のローカルメモリ、64Kbyte のプライベート・キャッシュが搭載されている。また、シンクロナイザ [7] という特殊なメモリを持つ。PU ボード上の拡張用コネクタにより、Ethernet ボード・SCSI ボードや計測に使用するハードウェア・イベント・モニタを取りつけることができる。アドレス空間のうち、共有メモリとシンクロナイザは全 PU で共有する。

6.3. シンクロナイザ

シンクロナイザは同期領域 (エントリ)X の集合から成り、(ATTEMPT-0 では 8K エントリ) 各 PU(プロセス) はある領域について使用登録を行った場合にのみ、以下の操作が可能となる。

Write(X,data) :
領域 X が lock されていなければ、データが書き込まれる。

Read(X) :
領域 X の値を読み出す。

Fetch&Dec(X) :
領域 X の値を読み出すと共に 1 減算する。これは不可分な操作である。値が 0 になるとそれ以上の減算は行われない。

これらの操作の組合せにより、セマフォ等の基本的な同期操作は容易に実現可能である。また、この機構によりメッセージのマルチキャストが高速に実現される。シンクロナイザでは、Read 操作と 0 に対する Fetch&Dec は共有バスを使用しない。バスを含めた総転送時間を表 1に示す。

表 1: シンクロナイザの動作時間

機能	時間	バス操作の有無
Read(X)	200ns	無し
Write(X,data)	650ns	有り
Fetch&Dec(X)	750ns	有り

また、書き込みを行った場合そのエントリに Write Interrupt Flag がセットされている PU に対し割り込みがかかる。

6.3.1. キャッシュとローカルメモリ

ATTEMPT-0 のキャッシュは、64Kbyte・2 Way セット・アソシァティブのライトスルー・キャッシュで、PU からのアクセスと並列にバスをスヌープすることが可能である。ページサイズは 64bytes とやや大きめに設定されている。

ATTEMPT-0 は各ボードに 128Kbytes の SRAM、4Mbytes の DRAM を持つ。4Mbytes の DRAM はジャンパの切替えによりローカルメモリとしても共有メモリとしても使うことができる。SRAM は割り込みに使用されるオートベクタのテーブルやスタックの領域として使用される。キャッシュアクセスに必要な時間を表 2に示す。

表 2: メモリシステムのアクセス時間

メモリシステム	アクセスの種類	時間 [ns]	転送 byte 数
ローカルメモリ		250	4 bytes
共有メモリ	Read miss	18200	64 bytes
	Read hit	150	4 bytes
	Write	750	4 bytes

6.3.2. カーネルの実装

各カーネル機能の ATTEMPT-0 上での実装に関して述べる。

ジョブ管理 :
`exit` システムコールによるジョブの終了を全 PU に通達するために、シンクロナイザによる PU 割り込みを使用した。

プロセス管理 :
スヌープキャッシュによりトラフィックを減少させられるので、プロセスの生成をグローバルメモリの領域を polling することにより検知させた。

スレッド管理 :
ATTEMPT-0 では、一部 PU にはタイマが存在しないためウォッチドッグタイマはタイマーを持つ PU にタイマーのない PU の時刻管理を行わせシンクロナイザにより割り込みをかけさせることにより実装可能である。しかし、負荷バランスの点で考慮する必要があり、現在まだ実装していない。

メモリ管理 :
MC68030 の MMU を用いてメモリ空間の仮想化を行った。ローカルメモリ空間の仮想化は容易であったが、グローバルメモリに関してはいくつか問題がある

1. ATC(TLB) によるアドレスの誤変換
2. アドレス変換ツリー変更中のアクセス

1. に関しては PU 間割り込みにより各 PU の ATC をフラッシュさせることで解決が可能である。また、2. についてはアドレス変換ツリーの変更時に割り込みをかけ全 PU を停止させることで対応可能である。しかし、現状ではグローバルメモリ空間を直接物理空間にマッピングし仮想化は行っていない。

巨大なデータも扱えるようにローカルメモリ空間及びグローバルメモリ空間のスワップを行う。ローカルメモリのスワップエリアとしてグローバルメモリを使用し、グローバルメモリのスワップエリアとしてはハードディスクを使用する。現在まだグローバルメモリに関してはスワップを行っていない。

同期 :

lock 及び condition 関数 [9] をサポートした

- `lock`: ロックを試み、失敗した場合ブロック
- `unlock`: ロックの解放
- `try_lock`: ロックを試み、失敗した場合 0 を返す
- `cond_wait`: 条件成立までブロック
- `cond_signal`: 条件の成立をその条件で待つ最も古いスレッドに通達
- `cond_broadcast`: 条件の成立をその条件で待つ全てのスレッドに通達

`lock` 失敗時及び `cond_wait` の際には、`thread_suspend` を用いブロックさせ、その他の部分はライブラリによりユーザモードで実行が可能である。

`lock` には、不可分命令を利用して実装する。シンクロナイザを直接使用すれば容易に実装可能だがシンクロナイザには限りがあるため直接使用することができない。そこで、ロックの変数を操作する際にシンクロナイザを用いてその操作自体にロックをかけた。この際、違うアドレスであれば複数の `lock` 動作を同時に行うことが可能であるため一つのシンクロナイザエントリで行うのはロスが大きい。そこで、ロックに用いる変数のアドレスを使用しハッシュ関数により複数のシンクロナイザエントリを用いるよう実装した。

PU 間通信 :

ポインタサイズのデータを指定 PU のキューに入れる (`send`) 及び、そのキューの先頭からデータを取り出す (`receive`) を実装した。

6.4. 評価

カーネルにおける各種時間の測定を行った。`pu_fork` はローカルメモリの使用量によりその処理時間が変化する。本測定ではプログラムサイズ 15.3k bytes のものを用いて測定を行った。この測定にはプロセスの生成される PU でのページインの時間が含まれる。

スレッドの生成及びスレッドの切替の測定の際は page fault が発生しない条件下で行った。

ローカルメモリのページインは新しいページの作成を測定した。

表 3: 処理時間

	処理時間 [μ sec]
プロセス生成 †	161800
生成側 ‡	75400
スレッド生成	340.6
スレッド切替え	44.4
ページイン (ローカルメモリ)	788.5

† `pu_fork` システムコール発行から相手 PU で実行が開始されるまでの時間

‡ `pu_fork` システムコール呼びだし側でローカルメモリの内容をグローバルメモリに転送し次の処理に移るまでの時間

プロセス生成に関しては、まだ最適化が十分なされていないため、このような数値になっている。

6.5. 行列の乗算

行列の乗算を行うプログラムを用いその実行時間の測定を行った。

式:$A\ B = C$

100×100 の行列を用い、行列 A 及び Bをローカルメモリに解である行列 Cをグローバルメモリに配置した。本プログラムは、マスタプロセスから、他の全プロセスを生成し、行単位にスレッドに割り当てプロセス間はスレッド数が等しくなるようにスレッドの割り当てを行った。

図 5に 1PU での実行時間を基準とした台数効果を示す。

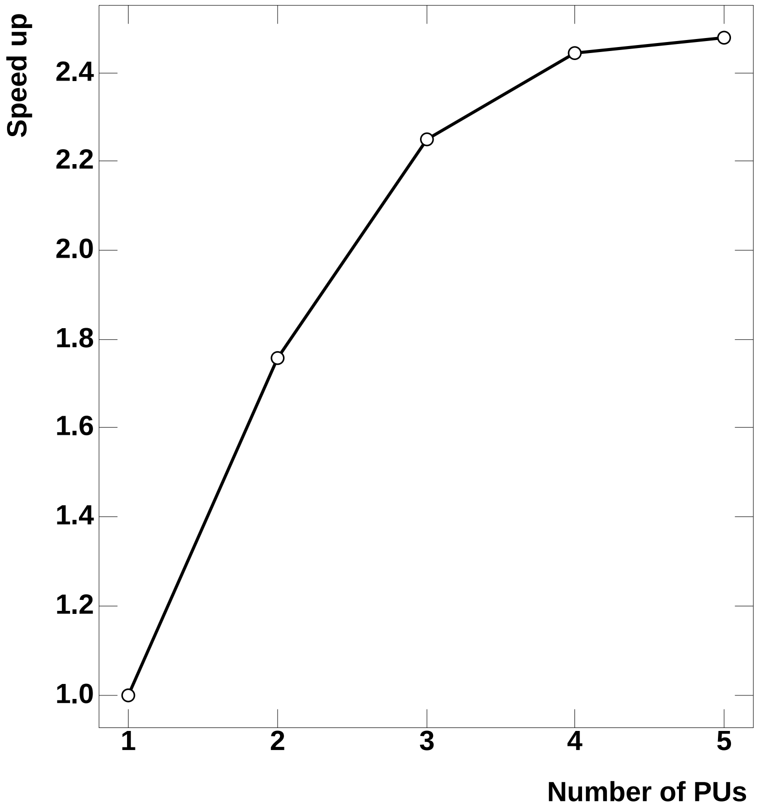

図 5: 行列の乗算の台数効果

図 5から 5PU で 2.48 倍であり飽和が起こっていることが確認された。

これは、プロセス生成のコストが原因ではないかと考えられる。マスタプロセスではプロセス生成が逐次的に行われている。

そのためマスタプロセスではプロセス生成にかかるコストがプロセス数に比例し増加する。各PUでの計算量はほぼ一定であり、従って、実行時間は最後に生成されたプロセスでの計算時間に依存することとなる。実際の実行時間は1PUで25.1秒、5PUで10.1秒である。本プログラムのローカルメモリの使用量は約100k bytesであり、表3にあるように15.3k byteで生成側で75.4ミリ秒、全体で161.8ミリ秒であるからプロセス生成のコストの影響は無視できないオーダーとなる。測定に使用した行列が100×100と小さいものであったので、より大きな行列つまり、個々のプロセスでの計算量が増加すればプロセス生成のコストは無視できるオーダとなるのではないだろうか。

科学技術計算ではデータの初期化後、プロセスを同時に生成することが多いと考えられる。そこで、複数のプロセスを一斉に生成するシステムコールを追加し、生成側のローカルメモリの転送回数を減少させることでオーバーヘッドを削減することが可能である。今後、そのようなシステムコールを追加しその効果を検討する予定である。

7. mobileデータ型

並列プログラミングを記述する場合にはプログラミングモデルとして共有メモリ型とメッセージパッシング型の2つがある。前者は主に分散メモリ型並列計算機に、後者は主記憶共有型並列計算機において用いられる場合が多い。また、前者はプログラムの記述がプログラマにとって容易、後者は書かれたプログラムの局所性が高い、という特徴をそれぞれ持っている。

今回対象としている並列計算機では共有メモリもローカルメモリも備えているので、どちらの記述も可能である。そこで、両者の長所を生かすべく次の方針に基づいてデータのアクセスをするような機構を設ける。

1. プログラムにはなるべくローカルメモリを使う。
2. 共有メモリにアクセスする場合は出来るかぎり特定のプリミティブを使用する。

この方針に基づき、以下のデータ型を用意する。

- ローカルデータ型
- 共有データ型
- mobileデータ型

この3つのデータ形式をプログラマが指定することにより、ローカルメモリの効率的な使用を目指す。

ローカルデータは、そのスレッドに固有のデータであり任意の時点でそのスレッドが更新することができる。また、他のスレッドからそのデータにアクセスすることはできない。このデータは、ローカルメモリに配置される。

共有データは、タスク内の全スレッドからアクセス可能なデータであり、このデータは共有メモリに配置される。

mobileデータは、その配置がローカルメモリと共有メモリ間を動的に移動するデータであり、その管理は特定の関数により行なわれる。また、mobileデータへのアクセスはその関数へのアクセスを介して行なわれる。すなわち、特定の管理関数がそのデータの配置位置を管理し、ユーザプログラムは物理的なデータの位置を知ることなくデータにアクセスすることができる。この機構を用いてローカルメモリを効率良く使用するため、まずデータはローカルメモリに配置する。そして、このmobileデータにアクセスするスレッドが同一プロセス内にあれば、ローカルメモリを使用して(共有メモリを使用することなく)データにアクセスすることが可能である。

ここで、他のプロセッサのプロセスからmobileデータへのアクセスが生じた場合には、管理関数はデータを共有メモリ上に移す。これとは逆に共有メモリ上にあるmobileデータにアクセスするプロセスが1プロセッサ上のみとなった場合、そのデータはローカルメモリに移動する。

この方式は、変数アクセスが関数呼びだしによってなされるので、関数呼び出しによるオーバーヘッドが生じるという欠点がある。しかし、データの管理は完全にソフトウェアで行なわれるので，データサイズはページにとらわれず、任意のデータサイズを扱える。

プログラマはこれらのデータ型を使い分けることによってプログラミングを行なうことになる。

7.1. mobile型の拡張

この節ではmobile型の様々な拡張について述べる。これらの機能はまだ未実装である。

7.1.1. 共有メモリのキャッシング

前節で述べたmobile型ではデータの複製は生じない。すなわち、データは単一のプロセッサ上のローカルメモリか共有メモリに配置され、同時に2箇所に存在することはない。そこで、ここでは複数のコピーが存在するようなmobile型について考える。複数のコピーがローカルメモリに存在すれば、コンシステンシを保つ複雑さは増えるが、共有メモリに存在する場合より高速にアクセスを行なうことができる。

複数のコピーが存在するので、何らかの方法でコンシステンシを保つ必要があるが、それには通常のキャッシュシステムと同様にinvalidate型とupdate型の二つの方法が考えられる。

今回対象としている並列計算機ではローカルメモリを持つが、あるプロセッサから他のプロセッサのローカルメモリに書き込みを行なうことはできない。そこで、複数のデータのコピーを作成するには共有メモリを介して行なうことになる。

7.1.2. 複数のデータ型

並列プログラミングにおいては、スレッド間での値の受渡しなど特定の用途に用いられるデータ構造がある。そのようなデータではアクセスされるプロセスが決まっていたり、書き込みに比べて読み込みがほとんどないなどの特定のアクセスパターンを示す。そこで、このようなデータにmobile型を適用することが可能である。この場合、mobile型はさらに幾つかの細かい型に分けられることになる。

7.2. 他の研究との比較

ソフトウェアで論理的な共有メモリを構成する機構として、Linda[10]が上げられる。Lindaではタプルスペースと呼ばれる論理共有メモリ空間を提供し、既存の言語にこのタプルスペースを付加することによってその言語を並列言語に拡張することが出来る。しかし、このタプルスペースは連想的マッチング処理が必要なため、その処理が重い。これに対し、mobile型ではタプルスペースのような高機能を提供しないため処理が簡単化でき、高速な実行が行なえる。

また、Munin [11]では分散共有メモリ型並列計算機においてアクセスパターンを幾つかの型に分類し、それに適した複数

のキャッシュプロトコルを用意することでデータアクセスの高速化を試みている。この方法はハードウェアで行なうので高速に実行ができるが、それに伴うハードウェア量が巨大になってしまう。これに対し、前述の mobile 型の拡張では同様のことを、既存のハードウェアに手を加えることなく ソフトウェアで行なうことができる。

7.3. mobile データ型の実装

次に mobile データ型の実装について述べる。

mobile データ型では、その物理アドレスの位置が共有メモリとローカルメモリの間で動的に変化する。そこで、物理アドレスを管理するためにテーブルを設け、このテーブルには一つの mobile 型に対応して一つの物理アドレスを格納する (表 4 参照)。

表 4: 変換テーブル

tag	物理アドレス	共有タスク数	アクセス情報

データへのアクセスは 2 段階に分けて行われる。まず第 1 段階では、そのプロセッサで既にその mobile データをアクセスしたかどうかをチェックし、mobile データ型の物理アドレスのテーブル更新等を行なう。そして第 2 段階でテーブル検索によりそのデータの物理アドレスを知り、実際にそのデータへアクセスをする。

以下にその詳細を述べる。

7.3.1. 第 1 段階

まず、第 1 段階ではそのプロセスが以前にその mobile データにアクセスしたかどうかを調べる。もし以前にアクセスされていれば、既に変換テーブルのセット等はされているはずなので、何も処理を行なわずにこの段階を終了する。

これまでにアクセスしていなければ、以下の場合分けにしたがって、変換テーブルを設定する。

- プロセスにローカルな場合
  その mobile データに対応する物理アドレスをローカルメモリに設定する。また、その mobile データの所有者としてプロセスの ID を保存する。
- 複数のプロセスで共有されており、このアクセスにより共有になる場合
  全プロセスで 変換テーブルを更新するために他のプロセスに割り込みをかける。そして割り込み処理ルーチン内でテーブルの更新を行ない、その mobile データの所有者に相当するプロセッサはローカルメモリから共有メモリへデータの転送を行なう。
- 複数のプロセスで共有されており、以前から共有されている場合
  この場合、全プロセスの変換テーブルは既に更新されているので何も処理を行なわない。

7.3.2. 第 2 段階

第 2 段階では、実際のデータへのアクセスを行なう。第 1 段階により mobile データ型の物理アドレスは変換テーブルに格納されている。それを検索することにより物理アドレスを得て、データをのアクセスする。

8. まとめ

共有メモリ型並列計算機用 OS である LC の概要及び実装について述べた。

LC はローカルメモリと物理的な共有メモリの両者を持つ並列計算機をターゲットとし、その各メモリの特性に応じたデータ配置を行うことで性能向上を図ることを目的とする。バックエンドマシンを対象とし、その処理時間を小さくするように設計を行っている。

現在、実装が完了しておらず、そのため各部の検討が十分とはいえない。システムの完成を目指し、実装を完了させ、実アプリケーションによる評価検討及び改良を行う予定である。

参考文献

[1] D.Lenoski, J.Laudon, K.Gharachorloo, W.Weber, A.Gupta, J.Hennessy, M.Horowita and M.S.Lam, “The Stanford Dash Multiprocessor,” Computer vol.24, No.3, pp.63-70, Mar. 1992.

[2] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. Mcauliffe, L. Rudolf, and M. Snir, “The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer,” IEEE Trans. on Comput. vol. c-32, No.2, 1983.

[3] C.F.Pfister, et al., “The IBM Research Parallel Processor Protype(RP3): Introduction and Architecture,” Proc. of Int. Conf. on Parallel Processing, pp. 764–771, 1985.

[4] 金田 悠紀夫,“数値処理マシン,” 情報処理 vol.28, No.1, Jan. 1987.

[5] 笹原 正司, 寺田 純, 大和 純一, 天野 英晴, “SSS 型 MIN に基づくマルチプロセッサ SNAIL の実装と評価”、信学技法 CPSY93-29(SWoPP93), 1993

[6] 鳥居 淳, 天野英晴, “並列計算機テストベッド ATTEMPT の交信機構の評価,” 並列処理シンポジウム JSPP ’91 論文集, pp.205-212, 1991.

[7] H.Amano ,T.Terasawa and T.Kudoh, “Cache with synchronization mechanism,” Proc. of IFIP Congress89, Aug. 1989, pp1001-1006.

[8] “IEEE Standard Backplane Bus Specification for Multi-processorArchitectures: Futurebus,” IEEE , Jun. 1988.

[9] Eric C. Copper and Richard P. Draves, “C Threads”, CMU-CS-88-154, Jun. 1988.

[10] Sudhir Ahuja, Nicholas Carriero, David Gelernter, “Linda and Friends”, Computer, August, 1986

[11] John K.Bennett, John B.Carter, Willy Zwaenepoel, “Adaptive Software Cache Management for Distributed Shared Memory Architectures”, The 17th Annual International Symposium on Computer Architecture, 1990