システムソフトウェアと
オペレーティング・システム 85－7
（２０００．８．４）

コンパイラによる制御が可能なDSMシステム *Fagus*の実現

横手 聡[†] 齋藤 彰一[††] 上原哲太郎[††] 國枝義敏[††]

[†] 和歌山大学大学院システム工学研究科
[††] 和歌山大学システム工学部

和歌山県和歌山市栄谷 930

{yokote, shoichi, tetsu, kunieda}@lazner.sys.wakayama-u.ac.jp

我々は, ソフトウェア分散共有メモリシステム *Fagus* の実装を行った. *Fagus* は, PC/WS クラスタをターゲットアーキテクチャとする自動並列化コンパイラの実行時環境である. *Fagus* は, コンパイラとプログラマに, cc-COMA(compiler controlled - Cache Only Memory Architecture) 環境を提供する. 一貫性制御は, エントリ一貫性モデルを採用した. ユーザもしくはコンパイラが *Fagus* にキャッシュ制御の指示を与えることで, 効率の良い一貫性制御を実現する. *Fagus* は, 通信には UDP/IP を用い, 100BASE-TX のイーサネット使用時, 約 90Mbps のデータ転送 性能を達成した.

キーワード： 分散並列処理, 分散共有メモリ, ワークステーションクラスタ, 自動並列化コンパイラ

Fagus: A Compiler Controlable DSM system

Satoshi Yokote[†] Shoichi Saito[††] Tetsutaro Uehara[††] Yoshitoshi Kunieda[††]

[†]Graduate school of Systems Engineering, Wakayama University
[††]Faculty of Systems Engineering, Wakayama University

930 Sakaedani, Wakayama-city, Wakayama, 640-8510 Japan

{yokote, shoichi, tetsu, kunieda}@lazner.sys.wakayama-u.ac.jp

We report this paper shows the implementation of our software DSM(Distributed Shared Memory) system named *Fagus*, which is basically designed as a runtime environment system for such automatic parallelizing compilers that are mainly aimed for PC/WS clusters, as their target architecture. *Fagus* provides cc-COMA(compiler controlled Cache Only Memory Architecture) environment not only for programmers but also compilers. As *Fagus* adopts entry consistency model, programmers or compilers can directly give directives on cache controls to *Fagus*, so as to realize efficient consistency control. By using the UDP/IP communicaton of LINUX kernel on a 100BASE-TX ethernet, *Fagus* has achieved about 90Mbps as its data transfer performance.

Keywords: distributed processing, parallel processing, DSM(Distributed Shared Memory)system, automatic parallelizing compiler

1 はじめに

我々は, PC/WS クラスタをターゲットアーキテクチャとする自動並列化コンパイラ実現のため, エントリー一貫性 (EC) モデルに基づくソフトウェア分散共有メモリ (DSM) システム *Fagus* を実現した. *Fagus* は, LINUX 上で動作し, cc-COMA(compiler controlled - Cache Only Memory Architecture) 環境 [1] をユーザに提供する. 通信には UDP/IP を用い, 100BASE-TX のイーサネット使用時, 約 11MB/sec の高いデータ転送性能を達成した.

自動並列化コンパイラ [2] とは, 逐次プログラムを並列プログラムへ自動的に変換するコンパイラである. 並列計算機の可用性を大きく高めるが, 現在ベクトル型計算機や主記憶共有型の計算機において実用化されているに過ぎない. クラスタ構成の並列計算環境では, 計算機間のデータ転送が非常に大きなオーバヘッドとなる. このため, 並列化による速度向上が, 通信による速度低下を常に上回るようデータとタスクを分割し, 各計算機 (以後ノードと呼ぶ) に配置しなければならない. しかし, コンパイル時には, 動的に変化するデータ転送のオーバヘッド予測は困難である. このデータ配置の問題が自動並列化の壁となっている.

データ配置問題の一つの解決策として, 我々は cc-COMA[3] と呼ばれる計算機アーキテクチャを提唱している. COMA 環境では, 全プロセッサがメモリアドレス空間を共有し, 各ノード上のローカルメモリは共有メモリのキャッシュとして振舞う. cc-COMA とは, COMA 環境をソフトウェアでエミュレートし, その制御をコンパイラにゆだねるものである. cc-COMA 環境では, キャッシュの効果により, 自動的にデータが頻繁に参照されるノードに集中するようになる. このため, コンパイラもしくはプログラマは, データの配置を意識することなくコードを生成することができる. 本論文では, cc-COMA モデルコンパイラが生成する並列プログラムの実行時環境として実現したソフトウェア DSM システム *Fagus* の実装について述べる.

以下, 第 2 章で *Fagus* と cc-COMA の関係について述べる. 第 3 章で *Fagus* のシステム構成を述べる. 第 4 章で *Fagus* の評価を行い, cc-COMA モデルコンパイラの作成ため基本的なデータを得る.

2 cc-COMA ランタイムライブラリ

cc-COMA は, 分散主記憶型の並列計算機を対象として COMA をソフトウエアでエミュレートし, キャッシュの制御をコンパイラに委ねるものである. cc-COMA では, COMA の性質から共有メモリは実行時に動的にプロセッサ間を移動し, データ分割が最適であれば移動回数は最小となる. これにより, コンパイル時のデータの配置の問題を解消できる.

cc-COMA では, 各ノード上のローカルメモリはキャッシュであるため, 共有メモリ内容の複製が複数存在する. そのため複製の一貫性制御を行う必要がある. 一貫性制御には, 実行効率を考慮し, 従来の一貫性モデルと比べ, キャッシュ制御の効率が良い EC モデル [4] を用いて一貫性制御を行う. EC モデルでは, 共有メモリをプログラム内で明示的に宣言し, 同期変数と関連づけることが必要である. 共有メモリアクセス開始時に, 当該共有メモリに関連づけられた同期変数を獲得し, アクセス終了時に同期変数を開放することで, メモリオペレーションの正当性を保証する. 関連づけ, 開放, 獲得といったキャッシュ制御のための操作は複雑であるが, これらの操作をコンパイラに委ねることで, プログラムの複雑さが隠蔽される.

Fagus は, 以上で述べたメモリオペレーションの機能を実現するためのランタイムライブラリ *libFagus* をユーザに提供している. 表 1 に *libFagus* のキャッシュ制御インタフェースを示す. 以下, 各インタフェースの概要について述べる. なお, *sync_t* 型は *libFagus* が定義する同期変数の型である.

- sync_t *fagus_create_syncvar(int *id*)

  id で指定した ID を持つ同期変数を作成する. 戻り値は, 成功ならば同期変数へのポインタ, 失敗ならば NULL である.

- void *fagus_create_region(size_t *size*)

  共有メモリの領域を *size* バイト分確保する. 戻り値は, 成功ならば確保した領域のアドレス, 失敗ならば NULL である.

- void *fagus_attach_region
  (sync_t *s, int *id*, void *p, size_t *size*)

  同期変数 s に, 領域 p を *size* バイト分関連づける. 関連づけられた領域は, 共有メモリ ID が *id* の共有メモリとなる. 戻り値は, 成功ならば p, 失敗ならば NULL である.

- int fagus_acquire(sync_t *s, int *mode*)

  同期変数 s を獲得する. *mode* は, RLOCK と WLOCK(3.2 節参照) の 2 種類がある. 共有メモリへのアクセスが読み出しのみの場合は RLOCK, 書き込みを行う可能性がある場合は

表 1　*Fagus* のキャッシュ制御インタフェース

関数名	機能
fagus_create_syncvar	同期変数の作成
fagus_create_region	共有メモリの割り当て
fagus_attach_region	同期変数と共有メモリの関連づけ
fagus_acquire	同期変数の獲得
fagus_release	同期変数の開放

WLOCK を用いる．戻り値は，成功ならば 0，失敗なら-1 である．

- int fagus_release(sync_t *s)

  同期変数 s を開放する．戻り値は，成功ならば 0，失敗ならば-1 である．

3　システム構成

本章では *Fagus* の全体構成を述べる．さらに，通信方式，一貫性制御方式，ノード間同期の方法について述べる．

Fagus システムは，ライブラリとして実現している．ユーザはプログラムに *libFagus* をリンクして使用する．また *Fagus* システムは，POSIX スレッドと UDP/IP を利用し，通信を行っている．現在 *Fagus* は LINUX 上で動作しているが，LINUX そのものに変更を加えていないため，POSIX スレッドと UDP/IP が使用可能な UNIX 系 OS であれば，容易に移植可能である．図 1 に *Fagus* の全体構成を示す．

コンパイラもしくは，プログラマによって生成された並列プログラムは，ユーザによって各ノードへ配置される．実行時には，プロセス内で 1 つないし複数のユーザスレッドが生成され，ユーザスレッド上で並列プログラムの各タスクが実行される．共有メモリ (DSM) は，*Fagus* のメモリ割り当て機能を利用することで，プロセスの仮想メモリ空間上に割り当てられる．ユーザスレッドは，共有メモリアクセス開始時と終了時にそれぞれ *fagus_acquire*() / *fagus_release*() プリミティブを呼び出し，*Fagus* にキャッシュ制御の指示を与える．*Fagus* は，ユーザスレッドの指示に応じて共有メモリのキャッシュ制御を行う．キャッシュ制御により，他のノードとの通信が必要になった場合は，*Fagus* の通信モジュールを通じて他のノードと通信を行う．通信によってキャッシュの転送，排他制御などを実行し，共有メモリを実現している．

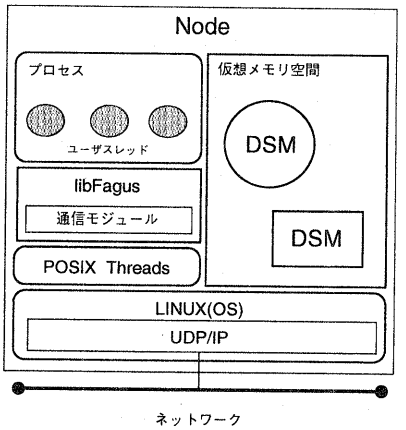

図 1　*Fagus* の全体構成

3.1　通信方式

通信モジュールはスループット性能向上のため，TCP/IP ではなく UDP/IP を用いて実装している．UDP ではメッセージの到達性が保証されないため，メッセージの紛失，順序の相違が発生する．実装ではこれらを考慮し，メッセージ受信側が送信側へ ACK(Acknowledge) を返すことで，メッセージ到達性を保証する方式を採用した．ACK とは，メッセージの到着を，受信側が送信側に通知するための制御メッセージである．送信側は，ACK を受信した時点で，ACK に対応する送信メッセージが確実に配送されたと断定することができる．*Fagus* の通信モジュールは，複数メッセージを 1 回の処理で送信し，それらの ACK を同時に待つことで通信の効率化を行う．

図 2 にメッセージ送信について示す．*Sender Node* はメッセージ送信を行うノード，*Receiver Node* はメッセージを受信するノードである．メッセージ受信処理は，メッセージ受信にともなう ACK 送信のレスポンスを高めるため，専用スレッドで行う．この専用スレッドを *Receiver Thread* と呼ぶ．*Receiver Thread* は，メッセージ受信時に *Sender Node* へ ACK を返す．*Sender Thread* はメッセージ送信を行うスレッドの総称である．メッセージ送信は，ユーザスレッドと一貫性制御を行うスレッドによって行われるため，*Sender Thread* は複数存在する．各スレッドには，それぞれのスレッドの生成時に，*Receiver Thread* との情報交換のためのスレッドテー

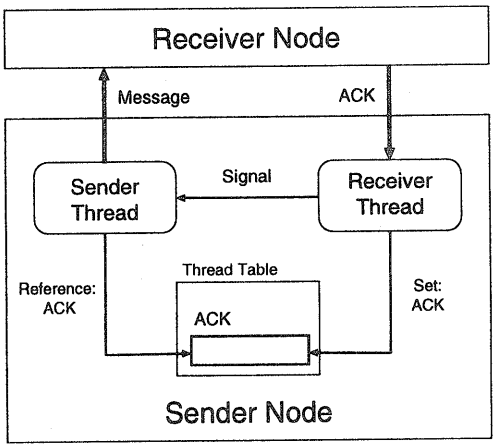

図 2　メッセージ送信

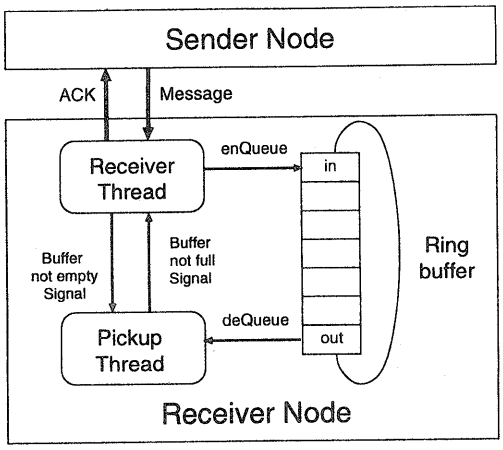

図 3　メッセージ受信

ブル (*Thread Table*) が割り当てられる.

メッセージ送信時, *Sender Thread* は送信するメッセージに関する情報をスレッドテーブルに保存する. sendmsg システムコールを利用して *Receiver Node* へメッセージを送信した後, *Receiver Thread* からのシグナルを待つ (*Fagus* は, 低速な UNIX のシグナル機構を利用せず, Pthreads ライブラリによって提供されるシグナル機構を利用する. 以後, 単にシグナルという場合は, Pthreads のシグナルのことをいう). このシグナルは, すべての送信メッセージに対応する ACK の到着を知らせるものである. *Receiver Thread* は, ACK を受信するとスレッドテーブルに受信 ACK として記録し, 送信メッセージすべての ACK 到着を確認すると *Sender Thread* にシグナルを送る. *Sender Thread* は一定時間内にシグナルを受信しなければ, スレッドテーブル内に記録されている受信 ACK を検査し, 受信していない ACK に対応するメッセージを再送する. 再送は *Receiver Thread* からのシグナルを受信するまで継続され, シグナルを受信した時点でメッセージ送信処理が終了する.

メッセージ受信時の処理を図 3 に示す. 送信ノード (*SenderNode*) から受信ノード (*Receiver Node*) に送信されたメッセージは, 受信ノードの *Receiver Thread* によって受信される. *Receiver Thread* は, メッセージを受信すると送信ノードに対し ACK を返す. 重複して受信したメッセージは *Receiver Thread* によって廃棄され, 有効なメッセージだけが *Ring buffer* に挿入される.

Pickup Thread は, キューからメッセージを取り出し, メッセージに応じた処理を行うスレッドである. *Receiver Thread* は, キューが満杯になると *Pickup Thread* からのシグナルを受信するまでメッセージの受信を中断する. *Pickup Thread* は, メッセージの取り出しによってキューに空きができると, *Receiver Thread* にシグナルを送る. 同様に *Pickup Thread* はキューが空になると, *Receiver Thread* からのシグナルを受信するまで, メッセージ取り出しを中断し, 次のメッセージに備える.

3.2　一貫性制御方式

一般的な DSM システムでは, ページフォルトを基にして一貫性制御を行う. *Fagus* では一貫性モデルに EC モデルを採用しているため, プログラムに指定された同期変数の獲得, 開放時にキャッシュ制御を実行する. このため, キャッシュ制御時に UNIX のシグナル機構が発行する SIGSEGV シグナルのハンドリングが必要なく, オーバヘッドを減少させることが可能である. しかし, 並列プログラム内のすべての共有メモリのアクセス開始時と終了時に, *fagus_acquire* () と *fagus_release* () を実行して, キャッシュ制御を行う必要がある. *fagus_acquire* () には, 書き込み獲得と読み出し獲得の 2 つのモードがある. ユーザは, 共有メモリに書き込む可能性があるときは書き込み獲得を, 読み出しだけの場合は, 読み出し獲得を用いて同期変数を獲得する.

Fagus の各ノードは, それぞれのノードが管理する同期変数の状態によってキャッシュを制御する. キャッシュ制御は, 指定された同期変数に関連づけられたすべての共有メモリに対して行う. また, 同期変数には所有権という概念がある. ある同期変数の所有権は, 同時には 1 台のノードだけが有する. 各

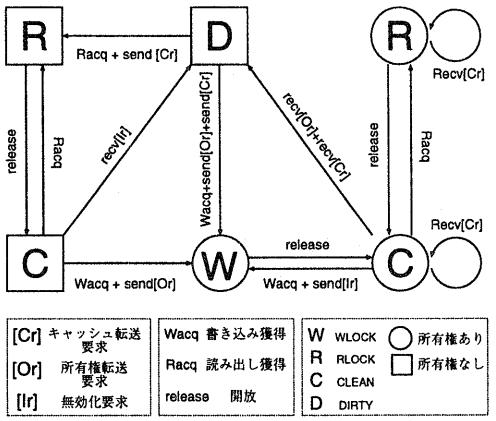

図 4　同期変数の状態遷移

同期変数の所有権を持つノードを, その同期変数の所有者と呼ぶ. 同期変数の所有者のみが, 共有メモリを書き込み獲得することで排他制御を実現する.

各ノードが持つ同期変数には, *WLOCK*, *RLOCK*, *CLEAN*, *DIRTY* の 4 種類の状態がある. 以下に, それぞれのノードの共有メモリのキャッシュの状態について説明する.

- *WLOCK*

  同期変数を書き込み獲得している状態である. 共有メモリへの書き込み, 読み出しアクセスを行うことができる.

- *RLOCK*

  同期変数を読み出し獲得している状態である. 共有メモリへの読み出しアクセスを行うことができる.

- *CLEAN*

  同期変数を獲得していない状態である. 共有メモリのキャッシュは有効で, この状態から獲得を行う場合は, キャッシュの更新は必要ない.

- *DIRTY*

  同期変数を獲得していない状態である. 共有メモリは, 同期変数の所有者によって更新されており, キャッシュが無効になっている. この状態から獲得を行う場合は, キャッシュの更新が必要である.

同期変数の状態遷移を図 4 に示す. 臨界領域外の同期変数の状態は, *CLEAN* または, *DIRTY* である. ただし, 所有者の同期変数の状態は, *DIRTY* にはならない. *CLEAN* の状態から読み出し獲得を行う場合は, 所有者との通信を行うことなく直ちに状態を *RLOCK* にし, 獲得を完了させる. *DIRTY* の状態から読み出し獲得または書き込み獲得を行う場合, 共有メモリがキャッシュされていないので, 所有者にキャッシュの転送を依頼する. さらに, 書き込み獲得を行う場合で, 所有権を持っていなければ所有権の転送も併せて依頼する. どのノードがキャッシュを保持しているかというキャッシュの位置管理は所有者が行い, コピーセットと呼ばれるにキャッシュを保持しているノードを記録する. 所有者は, 他のノードからキャッシュの転送を依頼されたとき, そのノードにキャッシュを転送するとともに, コピーセットに記録する.

書き込み獲得には, その同期変数の所有権が必要である. 書き込み獲得時の処理として, *Fagus* は Write-invalidate 方式でキャッシュの整合性を保つ. 書き込み獲得を行う際に, 所有者はコピーセットを検査し, キャッシュを保持しているノードへキャッシュの無効化要求を送信してから書き込み獲得を完了する. このとき, 開放時にキャッシュ無効化完了のメッセージが到着していなければ, 無効化が完了するまで開放を遅延させる.

書き込み獲得による臨界領域内では, キャッシュの転送要求と所有権の転送要求を受信しても要求の処理を行わない. これらの要求はキューに挿入され, 開放時に処理される. 読み出し獲得による臨界領域内ではキャッシュの無効化要求を受信してもキャッシュを無効化しない. 無効化要求があった場合は, 無効化要求があったことを記録し, 開放時にキャッシュを無効化するとともに, 所有者に無効化完了のメッセージを送信する.

3.3　バリア同期

バリア同期は, ノード間で同期が必要な場合に用いられる. バリアはコーディネータノードと呼ばれるノードによって集中的に処理される. バリア同期を行うノードは, それぞれがプログラム中の同期点に到着すると, コーディネータノードにバリア同期要求を送る. ユーザスレッドが複数存在する場合は, バリア同期を行うすべてのスレッドが同期点に到着したところでバリア同期要求を送信する. コーディネータノードは, 到着したバリア同期要求がバリア同期を行うノード数に達すると, バリア同期完了のメッセージをバリア同期を要求したノードへ送信する. 各ノードは, バリア同期完了のメッセージを受信した時点で, 処理を再開する.

表 2　実験環境

計算機	PC-AT 互換機 (最大 9 台) -CPU Celeron400MHz -主記憶 128MB
OS	Linux (Kernel 2.2.16)
C コンパイラ	gcc 2.95.2
ネットワーク	100BASE-TX イーサネット (スイッチングハブで接続)

4　評価

本章では, データ転送速度の測定と, 行列積演算の実行時間の測定によって *Fagus* の基本的な性能評価について述べる. データ転送速度を求める実験では, 複数台のノードが保持する共有メモリのキャッシュを 1 台のノードで読み出す実験と, 1 台のノードが保持する共有メモリのキャッシュを複数台で読み出す実験を行う. それぞれの実験で読み出しに要した時間を測定し, データ転送速度を求めた. 行列積演算では, 1 台のノードと, 複数台のノードでそれぞれ行列積演算を行い, 演算に要した時間をそれぞれ計測し, 台数効果を求めた. なお, どの実験においても各ノードのユーザスレッド数は 1 である.

本実験は, 最大 9 台の PC を使用し, 1 台に 1 ノードを割り当て実験を行った. 実験環境を表 2 に示す.

4.1　データ転送性能

Fagus の基本的なデータ転送性能の評価を 2 種類の実験を用いて行った. 複数台のノードから 1 台のノードでキャッシュを読み出す実験と, 1 台のノードから複数台のノードでキャッシュを読み出す実験である. それぞれ, 受信ノードは共有メモリがキャッシュされていない状態で *fagus_acquire*() を呼び出し, キャッシュを転送する. 転送時間は, *fagus_acquire*() の呼び出しの直前と, *fagus_acquir*() 完了の直後の時間を測定することで求めた. なお, 個々の共有メモリのサイズは, 64kByte, 256kByte, 1MByte, 4MByte, 16MByte までの 5 種類で実験を行った.

- 実験 1: 複数台のノードから 1 台のノードがキャッシュを読み出す場合

  受信ノード 1 台と, 送信ノード N 台を用意する. 送信ノードは, ノード ID 1 から N まであり, 各ノードは, ノード ID と同じ ID の共有メモリのキャッシュを保持している. 受信ノードは, 各ノードがキャッシュする共有メモリを同時に読み出し, データ転送に要した時間を測定する.

表 3　実験 1: 複数ノードからの読み出し [MB/sec]

ノード数 (N)	1 ノードあたりの転送バイト数				
	64kB	256kB	1MB	4MB	16MB
1	9.53	10.85	11.26	11.37	11.38
2	9.56	10.89	11.27	11.38	11.38
4	9.58	10.89	11.27	11.37	11.38
8	9.61	10.89	11.28	11.38	–

表 4　実験 2: 複数ノードでの読み出し [MB/sec]

ノード数 (N)	1 ノードあたりの転送バイト数				
	64kB	256kB	1MB	4MB	16MB
1	9.55	10.87	11.27	11.38	11.38
2	9.84	10.97	11.29	11.38	11.38
4	9.96	11.01	11.30	11.38	11.39
8	10.01	11.03	11.31	11.38	–

- 実験 2: 1 台のノードから複数台のノードがキャッシュを読み出す場合

  送信ノード 1 台と, 受信ノード N 台を使用する. 受信ノードはノード ID 1 から N まであり, 送信ノードは共有変数 ID 1 から N までの N 個の共有メモリをキャッシュしている. 受信ノードは, それぞれが自身のノード ID と同じ ID の共有メモリを一斉に読み出し, データ転送に要した時間を測定する. なお, 受信を行うノードは, *fagus_acquire*() 呼び出しの前後に, バリアによって同期させ計測時間を同じにする.

実験 1 と実験 2 で測定した転送時間からスループットを求め, それぞれを表 3 と表 4 に示す. 2 つの実験結果から, どの場合においても安定して高いスループット性能が得られたといえる. 本実験に使用したネットワークが 100Mbps であることから, 転送速度の理論値は 12.5MB/s である. 実験 1 と実験 2 の各結果は, ネットワーク性能を十分に使いきっているといる. 実験 1, 2 とも 1 ノードあたりの転送バイト数が少なくなるに従い若干のスループット低下がみられる. これは共有メモリサイズが小さいために転送時間が短くなり, 転送時間全体に占めるオーバヘッドの比率が増したためと考えられる.

```
for( i=0; i<NUM;i++ )
  for( j=0; j<NUM; j++ )
    for( k=0; k<NUM; k++ )
      A[i][j] += B[i][k] * C[k][j];
```

図 5　行列積演算の逐次プログラム

表 5　アクセスパターンの分類

内積型	IJK 型, JIK 型
外積型	KIJ 型, KJI 型
中間積型	IKJ 型, JKI 型

4.2　行列積

複数のノードを用意して並列に行列積演算を行い, 実行時間を計測した. 行列は 2 次元配列で表現し, 配列サイズは 2048 × 2048 × 8(double 型) の 32M バイトである. A, B, C の 3 つの配列を確保し, A=B × C の行列演算を行った. 行列を求める逐次プログラムを図 5 に示す.

本プログラムの 3 重ループは, ループの外側から内側へ向けて制御変数が IJK ととられていることから以後 IJK 型 [5] という. 行列積演算では, 3 重ループの制御変数の順番がそれぞれ入れ替わっても, 行列へのアクセスパターンが変わるだけで, 演算結果は同じとなる. よって, 制御変数の順番の選び方は 6 通りある. 和を作る制御変数 k の位置によって分類すると表 5 のようになる.

実験では, IJK 型 (内積型), KIJ 型 (外積型), IKJ 型 (中間積型) の 3 種類のアクセスパターンについて実行時間を測定した. これは, アクセスパターンによって, CPU のキャッシュの効果が異なり実行時間に大きく作用するからである. さらに, コンパイラによる最適化を行うことで実行時間が大きく異なるため, gcc の-O2 オプションによって最適したプログラムの実行時間も計測した. 逐次プログラムの実行時間と, 並列プログラムの実行時間を計測し, 台数効果を求めた. 行列積を求める並列プログラムを図 6 に示す. なお, cc-COMA モデルの自動並列化コンパイラは開発中のため, コンパイラが生成する並列プログラムが得られない. 図 6 の並列プログラムは, コンパイラが生成すると考えられる疑似コードである.

図 6 の変数 *entrynum* は並列計算を行うノード数, *id* はノード ID, NUM は行列の要素数で 2048 である.

```
fagus_acquire( sc, RLOCK );
for( i=id; i<NUM; i+=entrynum ){
  fagus_acquire( sa[i], WLOCK );
  fagus_acquire( sb[i], RLOCK );

  for( j=0; j<NUM; j++ )
    for( k=0; k<NUM; k++ )
      *(a+(i*NUM)+j)+=
        *(b+(i*NUM)+k) * (*(c+(k*NUM)+j));

  fagus_release( sb[i] );
  fagus_release( sa[i] );
}
fagus_release( sc );
```

図 6　行列積演算の並列プログラム

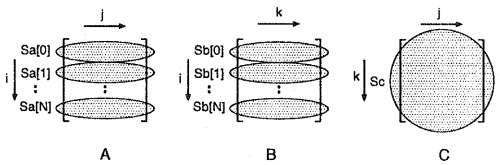

図 7　同期変数と共有メモリの関連づけ

る. ノード ID は各ノードに対し, 0 から *entrynum*-1 まで割り振られている. *a*, *b*, *c* は共有メモリとして確保された二次元配列へのポインタである. *sa*, *sb*, *sc* は, 同期変数である. それぞれ, 共有メモリの *a*, *b*, *c* と関連づけられている. 行列との同期変数の関連づけを図 7 に示す. *sa*, *sb* は同期変数の配列で, それぞれ行列 A, B の 1 行に一つの同期変数を関連づける. 行列 C はすべてのノードが全要素に対してアクセスを行うので, 一つの同期変数 (*sc*) に行列 C 全体を関連づける.

行列積演算に要した時間を図 8 に示す. なお, ノード数が 1 の時間は逐次プログラムの実行時間である. グラフ中の凡例の-*O2* はコンパイラによる最適化を行ったという意味である. 実行時間は, アクセスパターンとコンパイラによる最適化の有無によって大きく変化した. 実行時間は, アクセスパターンが IJK 型で最も長くなり, IKJ 型で最も短くなった. IKJ 型と KIJ 型は, IJK 型と比較して約 2 倍高速であった. 特に KIJ 型は, IJK 型と比較して同期変数の獲得回数が多く, 参照する共有メモリサイズが大き

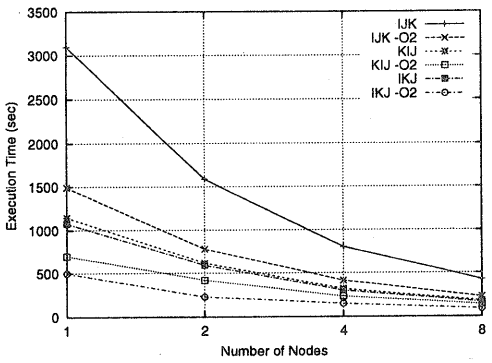

図 8　行列積演算の実行時間

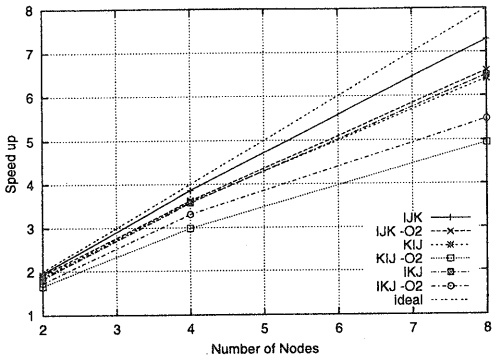

図 9　行列積演算の台数効果

いにもかかわらず, 実行時間ではるかに短い. これは, IKJ 型と KIJ 型の配列アクセスパターンは行方向への連続したメモリアクセスが多いために, メモリ参照の空間的局所性が高まり, CPU のキャッシュが有効に使われたためだと考えられる. また, IKJ 型と KIJ 型では, IKJ 型が高速である. これは, IKJ 型が KIJ 型と比較して同期変数の獲得回数が少なく, 参照する共有メモリサイズが小さかったためと考えられる.

この結果から, 高速な並列プログラムを作成するためには, アクセスするメモリアドレスが連続するようにメモリアクセスを行うことと, 同期変数の獲得回数を少なくすることが有効である.

行列演算の実行時間から求めた台数効果を図 9 に示す. IJK 型 (内積型) が最も台数効果が高く, 実行時間が最短である IKJ 型 (中間積型) は 5.5 倍程度であった. これは, 逐次プログラムでも実行速度が最速であったため, 台数効果が得にくかったことが原因である. 配列サイズがさらに大きくなり, 逐次プログラムではスラッシングが発生するような場合では, 台数効果が高まるのではないかと考えられる.

5　おわりに

分散共有メモリシステム *Fagus* の実装を行った. 本システムで得られるスループットは、*Fagus* のデータ転送速度が安定して高速であることを示した. また、行列積演算実験では, 自動並列化コンパイラ作成に際して高速な並列プログラムの生成には, 当然ながら, アドレスが連続したメモリアクセスと, 同期変数獲得の回数を減少させることが有効であると示された.

今後, *Fagus* のインタフェース部分を残し, 残りの主要部分を LINUX カーネルモジュールとして実装する予定である. 通信と一貫性制御をカーネル内部で行うことで, コンテクストスイッチの発生が大幅に減少し, さらなる性能向上が見込まれる.

参考文献

[1] 城 和貴, 上原 哲太郎, 國枝 義敏 : cc-COMA の概要, 知能情報・高度情報処理シンポジウム報告書, pp. 85-88 (1997).

[2] Yoshitoshi Kunieda, Kazuki Joe, Akira Fukuda, Tetsutaro Uehara, Shoichi Saito, Tetsuya Saito, Mariko Sasakura, Tsuneo Nakanishi : Design and Implementation of an Automatic Parallelizing and Distributing Compiler with Visualization Tools and the Runtime Environment, In Proceedings of the International Conference on PDPTA '2000, pp. 713-719 (2000).

[3] Shoichi Saito, Tetsutaro Uehara, Kazuki Joe and Yoshitoshi Kunieda : cc-COMA: the compiler-controlled COMA as a framework for parallel computing, Innovative Architecture for Future Generation High-Performance Processors and Systems '98, IEEE CS press, pp. 114-119 (1999).

[4] Andrew S.Tanenbaum : 分散オペレーティングシステム, Prentice-Hall (1996).

[5] 津田孝夫 : 数値処理プログラミング, 岩波書店 (1988).