システムソフトウェアと
オペレーティング・システム 67－13
(1994. 12. 9)

分散環境上の並列処理支援システム DYAMONDS の設計

小野 文豊* 福田 晃
{fumito-o,fukuda}@is.aist-nara.ac.jp

奈良先端科学技術大学院大学 情報科学研究科
*住友金属工業（株） システムエンジニアリング事業本部

UNIX 環境下の分散システム上で、ユーザに分散仮想共有メモリのイメージを提供するソフトウェアシステム DYAMONDS について述べる。近い将来の分散環境においては同一エリアのネットワークに単一プロセッサのワークステーションとマルチプロセッサの並列計算機 (またはワークステーション) が混在することが想像されるが、この場合 UNIX ワークステーション間で、ユーザにアドレス空間を共有させる何らかの手段が必要となる。我々は今回上記ハードウェア環境をターゲットとした DYAMONDS において、分散共有メモリ部を作成し、その評価を行った。本稿では、DYAMONDS の設計、実装、および簡単な評価結果について述べる。

F.Ono*, A.Fukuda
{fumito-o,fukuda}@is.aist-nara.ac.jp

Graduate School of Information Science, Nara Institute of Science and Technology
8916-5 Takayama, Ikoma 630-01, JAPAN

*System Engineering Division
Sumitomo Metal Industries, Ltd.
2-16 Kitahama-Higashi, Chuo-ku, Osaka 540, JAPAN

This paper describes the DYAMONDS software system, which provides users distributed virtual shared memory on distributed hardware system. In the near future, distributed environment where each node may be workstation with uni-processor or with multiple ones will appear. The DYAMONDS system is for the hardware environment with UNIX. In this paper, we describe design principles, inplementation, and performance evaluation of DYAMONDS.

1 はじめに

今日のUNIXワークステーションの普及およびネットワーク技術の飛躍的な進歩により、分散環境の構築が非常に一般的なものとなってきた。このまま行けば近い将来には同一エリアのネットワーク上に単一プロセッサのワークステーションとマルチプロセッサの並列計算機(またはワークステーション)が混在するような状況となり、局所的な並列処理を内に含んだより高レベルの分散処理技術が求められることとなろう。

我々はこのような現状をかんがみて、移植性と並列性双方を兼備したユーザレベル・スレッドライブラリPPL[1]を開発した。しかしマルチスレッド処理においては互いにアドレス空間を共有することを前提として高速性を実現しているものの、単一プロセッサのワークステーションの間ではアドレス空間を共有する術を持たない。

本稿では分散環境上のUNIXワークステーションがアドレス空間を共有するための手段である分散仮想共有メモリと、並列計算機上のPPLマルチスレッド・インターフェースの双方をユーザに提供するDYAMONDS (DYnAmic Memory OrgaNizer on the Distributed System) の設計について述べる(図1)。

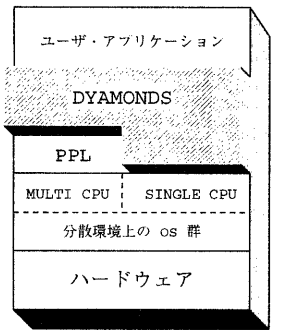


図1: システム・アーキテクチャ

2 分散仮想共有メモリの実現手法

UNIX環境における分散仮想共有メモリの実現手法のうち代表的なものを以下に示す。

メモリマップドファイル UNIXファイルを大域的なアドレス空間の実体として利用する手法。UNIXシステムではmmap()システムコールとして実装されている。一旦ローカルなアドレス空間にマッピングされたファイルはローカルなアドレス指定によりアクセスできるが、

- ファイル更新反映のタイミングがOSに依存している。
- 異なるノード間でデータの一貫性維持が困難である。
- 読み書き操作にディスク装置を介するため無視できない遅延が生じることがある。

等の理由からマルチスレッド・アプリケーションには不向きと考えられる。

外部ページャ Mach[2]マイクロカーネルで採用されている手法。カーネル外部のタスクがページング管理を行うため、ユーザ独自の仮想記憶管理手法を構築する手段が提供されている。

ソフトウェア・キャッシュ 分散共有メモリを各ノードに分割配置し、ネットワークを介したリモート・データへのアクセス軽減のためにキャッシュを導入した手法。UNIXワークステーション上で完全に実装した例は未だまれである([3])。

DYAMONDSにおける分散仮想共有メモリの実現においては、上記の外部ページャの手法を参考とした管理手法のもとで、UNIX環境上で高い移植性を有するソフトウェア・キャッシュを実装している。

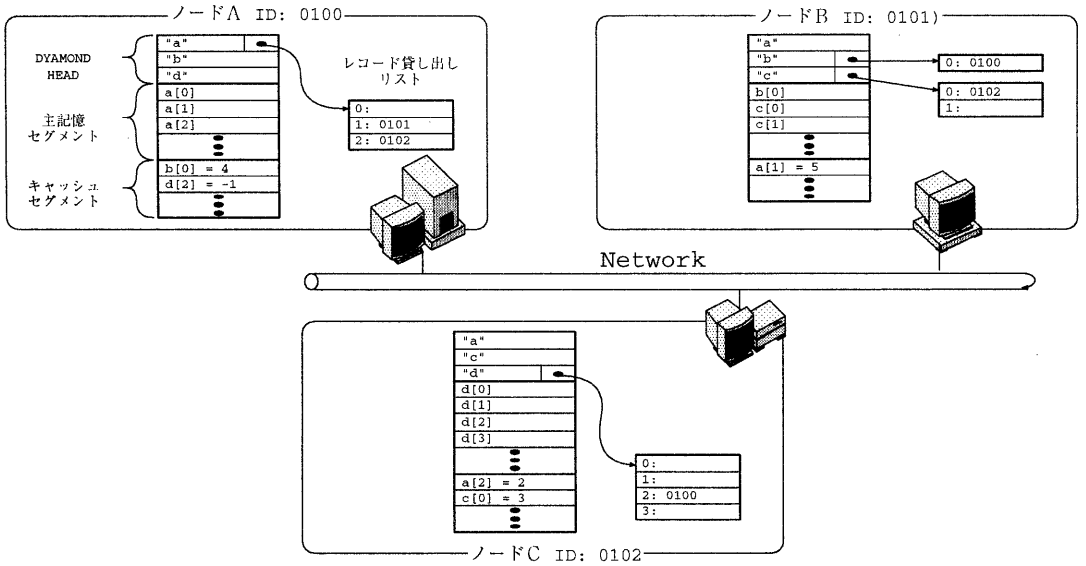


図 2: DYAMONDS の構成

3 DYAMONDS の設計

図 2に構成を示す。各ノードはローカルな物理メモリのノンスワップ空間内に、グローバル・データを格納する主記憶セグメントおよびキャッシュセグメントを有する。各セグメント内に格納されているグローバル・データは DYAMOND HEAD と呼ばれるデータ管理セグメントにより一括管理されている。

主記憶セグメント アプリケーション起動時からグローバル・データの実体を格納するセグメント。グローバル・データは各ノード固有の環境パスの下に存在するインクルード・ファイル "Global.h" においてあらかじめ定義されている。DYAMONDS はこのファイルを C 言語のプリプロセッサおよび固有の構文解析ルーチンに読み込んで未定義データ型の展開処理を行った後、各データ型ごとにグローバルに認識可能な固有の名前と、後述のレコード貸出し管理リストを割り当てる。

キャッシュセグメント 他ノードで名前を割り当てられたグローバル・データをデータの論理的な読み書きの最小単位であるレコード単位で格納するセグメント。キャッシュセグメントはシステム起動時には全くエントリを有しておらず、動的に内容が更新される。

DYAMOND HEAD 主記憶セグメントおよびキャッシュセグメントの内容を、前述の名前をキーとして一括管理するセグメント。キャッシュセグメントと異なり、主記憶セグメントに存在するグローバル・データは一度格納されれば再び追い出されることはなく、したがってこれらのデータの管理責任も終生同じノードが負うこととなる。これらのデータが他ノードのキャッシュセグメントに格納される場合、レコード貸出し管理リストに貸出し先のノードのＩＤが記録される。

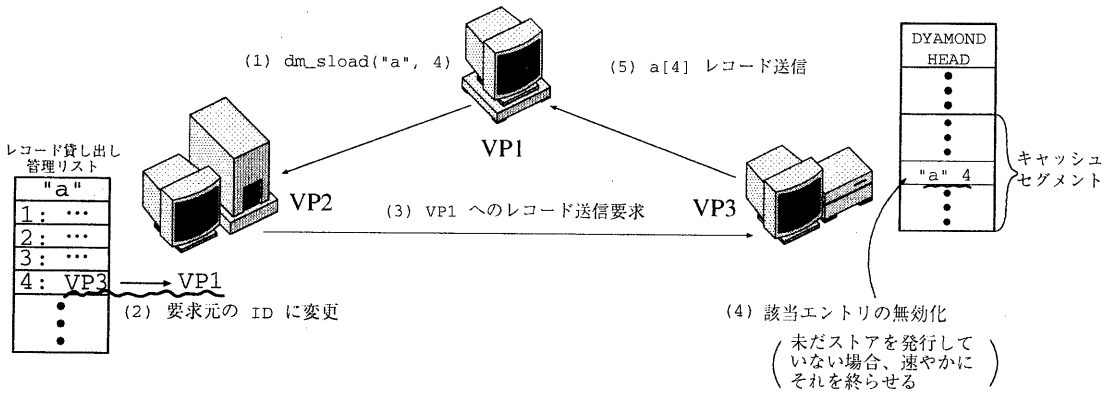


図 3: グローバル・データ読込みの例

4 メモリー貫性制御

DYAMONDS はグローバル・データへのアクセスに関して、以下に挙げる単純なアクセスルーチンの明示的な呼出し以外は一切ユーザーに強要しないという非常に高レベルな機能を有している。

dm_load() グローバル・データの参照。書込み更新を行わないデータの読込みに用いられる。具体的な使用例として、アプリケーション起動時の初期化パラメータの獲得等が挙げられる。最新バージョンの保持は保証されないのでその都度対処が必要となる。

dm_sload() グローバル・データのロック付き読込み。自ノードのキャッシュに最新バージョンが格納されることを保証している。常に次の dm_store() と対で用いなければならない。

dm_store() グローバル・データの書込み／ロック解除。他スレッドからのデータ読出し要求に応じるために自ら保持する最新バージョンを開放する。

前述のように全てのグローバル・データは大域的な名前付けがなされており、上記の呼出しもそれらの名前をキーにして行われる。一貫性の制御はアプリケーション起動時にその名前を定義したノードにおいて行われるので、ロード要求メッセージの局所的な集中現象を避けるためにグローバル・データの静的な分散配置が肝要となる。

図 3の例によりグローバルデータ読込みの手順を示す。

1. ノード VP1 に存在するスレッドが “a” という名前の付いたグローバル・データの (0 から数えて) 4 番目のレコードのロック付き読出し要求を発行する。(要求先のノードが分からない時はマルチキャストメッセージを発行することとなる。)

2. “a” という名前を最初に定義したスレッドはノード VP2 に存在している。このスレッドが名前 ”a” に対応するレコード貸出し管理リストの該当レコード番号の場所に書かれたＩＤを読み取り、今度は要求元ノードのＩＤを格納する。

3. 読み取られたＩＤを持つノードが要求されたレコードの最新バージョンを持っているため、そのノードに要求メッセージを転送する。

4. 転送されたメッセージを受けたノード VP3

は、要求されたレコードのストアを完了させた後、そのレコードのあるキャッシュ・エントリを無効化する。

5. 要求元のノード VP1 に要求されたレコードを送信する。

DYAMONDS におけるメモリ一貫性制御プロトコルにおいては、極力無効化要求メッセージ等によるネットワークのトラフィックの爆発的上昇を軽減するという目的から、新たにレコード貸出し管理という概念を導入することとなった。本プロトコルと Release Consistency[4] に代表される既存のメモリ一貫性アルゴリズムとの整合性の検証が今後の急務である。

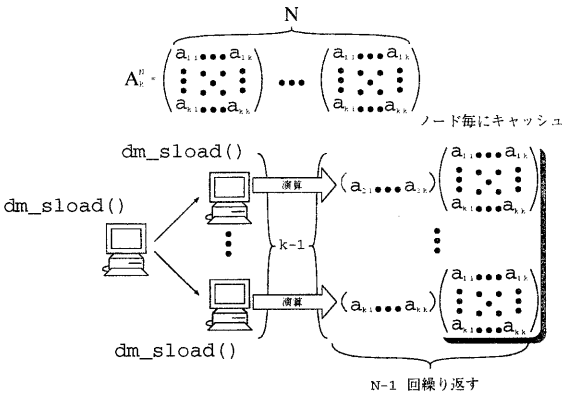


図 4: 行列のベキ乗計算

5 評価と考察

本章では、UNIX 環境において DYAMONDS により実現された分散仮想共有メモリの効果を測定するための評価実験の概要を述べ、実験結果に対する考察を行う。

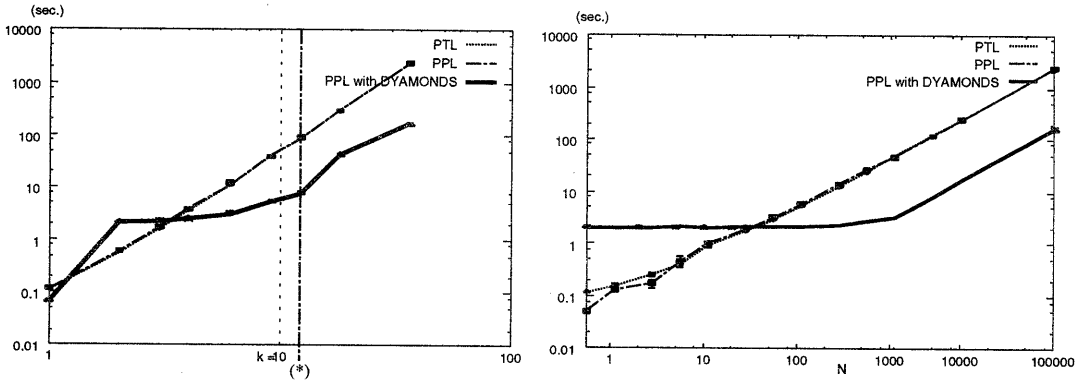
5.1 評価実験

評価実験として、行列のベキ乗計算に要する時間の測定を行った。まず初期設定としてベキ乗すべき行列が各ノードに参照される。次に個々のスレッドが自分が計算を担当する 1 行分のグローバル・データをロック付きで読出す。評価実験アプリケーション起動時にグローバル・データの名前付けを行ったノードに存在するスレッドは、各スレッドの実験終了をやはり対応する行データのロック付き読出しにより待つこととなる (図 4)。

5.2 評価結果の考察

図 5に評価結果を示す。図 5a) は行列の次数 k をパラメータに、図 5b) は乗数 N をパラメータにした時の実行時間である。今回比較の対象として用いたのは、2 種類のユーザレベルスレッドライブラリ PTL[5] および PPL である。図 5において単一プロセッサの 1 台のワークステーション上で PTL および PPL を実行した場合を各々 PTL、PPL として示す。また、行列のベキ乗計算を行列の次数 k と等しい k 台のワークステーションで並列処理した場合 (図 5a) では k 台、図 5b) では 36 台) を PPL with DYAMONDS として示す。この場合、各スレッドはノード数の範囲内で完全に並列な処理が実行され、他ノードに存在するグローバル・データへの位置透過なアクセスが可能となる。(なお OS はいずれも DEC 3000/300 である。)

今回使用したベキ乗の計算は並列計算の途中においてスレッド間通信をほとんど要することがなく、純粋に計算に要する時間のみを測定して並列化およびキャッシュの導入による効果を評価することができる。さらに公平な条件を与えるために PPL/PPL のスケジューラは計算途中におけるコンテキスト・スイッチを避ける FIFO 形式とし、DYAMONDS のサポート付き PPL (PPL with DYAMONDS) においては初期設定に要した時間を加算しないこととした。



a.) k 次行列の 10 万乗の計算時間

b). 36 次行列の N 乗の計算時間

図 5: 行列のベキ乗計算に要する時間

5.2.1 並列化の効果

まず PPL with DYAMONDS におけるスレッドの並列化による効果を評価するため、かけ合わせる回数を 10 万乗に固定し、行列の次数 (k) すなわち並列で処理が実行されるスレッド数を変化させた。図 5a) に示される結果から、現行のプロトタイプにおいてはノード数がグラフ中 (*) に示す数、すなわち 12 台を越えた時点で並列度が頭打ちになっていることが読み取れる。この原因は直観的には稼働ノード数に追い付くだけのバンド許容幅を通信回線が有していないことにより、いわゆるボトルネック現象が生じているためであると推測できる。

5.2.2 キャッシュ導入の効果

次に行列の次数の方を 36 次に固定し、かけ合わせる回数 (N) を変化させた。評価実験アプリケーション起動時に各スレッドが参照した行列データはキャッシュに格納されることにより再利用が可能となるため、同じデータを繰り返し用いることとなる本評価実験等のアプリケーションに対して絶大な効果を発揮する。実際図 5b) より、乗数 N を上げても並列度の効果を相殺することがないということが読み取れる。

今後の課題としては、スレッド間同期の多用によるキャッシュ・ブロックの頻繁なリスト並べ換え操作、キャッシュの許容量をしばしば越える場合のエントリの挿入／削除操作、あるいはそれに伴うガベージ・コレクション等、これらのキャッシュ管理操作そのものによるオーバーヘッドの見積りが第一にあげられる。

6 おわりに

UNIX 環境で分散仮想共有メモリおよびユーザーレベルスレッドを実現する DYAMONDS の概要について述べた。さらに簡単な評価実験アプリケーションを用いて今回作成したプロトタイプについて、並列性とユーザーレベル・キャッシュ導入の効果を見積った。今後の課題をあげるとすればネットワークに関するボトルネックや、キャッシュ管理操作自体のオーバーヘッドの解消等があげられるであろうが、これらの課題は近い将来のハードウェア・アーキテクチャの進歩、高速通信技術技術の確立、あるいは共有データの最適配置

を可能にする言語処理系の開発等の周辺技術に頼る所が大きい。我々はそのような時代に向けて、特定の計算機環境に依存しない高水準なソフトウェア・モデルを構築することを常に念頭において研究を進めて行かなければならない。

謝辞

本稿の執筆に際して貴重な御助言をいただいた奈良先端科学技術大学院大学の福田研究室、荒木研究室の皆様に深く感謝の意を表します。

参考文献

[1] 宮崎, 坂本, 最所, 福田, "異なる仮想プロセッサに適応できるスレッドライブラリ", 情報処理学会システムソフトウェアとオペレーティングシステム研究会, 61-14, Aug. 1993.

[2] M.Accetta, R.Baron, W.Bolosky, D.Golub, R.Rashid, A.Tevanian, and M.Young, "Mach: A New Kernel Foundation for UNIX Development", Proceedings of Summer Usenix, Jul. 1986.

[3] 蔵前, 中條, 前川, "ソフトウェア DSM におけるコヒーレント・キャッシュシステムの実装と評価", 並列処理シンポジウム JSPP'94, 303-310, May. 1994.

[4] K.Gharachorloo,A.Gupta,and J.Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", 17th ISCA, 15-26, 1990",

[5] 安倍, 松浦, 谷口, "BSD UNIX の下でのポータブルマルチスレッドライブラリ PTL の実現", 情報処理学会研究報告, 情報, 1994.