システムソフトウェアと
オペレーティング・システム 67－9
(1994. 12. 8)

並列プログラムのための視覚的性能デバッガ

久野 啓，大澤 範高，弓場 敏嗣
電気通信大学 情報システム学研究科
〒182 東京都調布市調布ヶ丘 1-5-1
E-mail: {hisano, osawa, yuba}@yuba.is.uec.ac.jp

あらまし　本研究では、並列プログラムの実行中に出力したログデータをもとに、そのプログラムの実行時の振舞いを理解し、性能デバッグするために、実行状況を視覚的に表示するツールについて考察する。このツールではログデータから性能デバッグに必要な情報を内部モデルに変換する。この内部モデルを用いることにより、表示させるだけの単なる視覚化システムではなく、視覚的に得られた情報をもとに、スレッド単位に視覚的にスケジューリングのシミュレートを行える。また、そのシミュレート結果をソースプログラムに自動的に反映させる。この目的を達成するシステムの設計思想と実装方法について述べる。

和文キーワード パフォーマンスチューニング，視覚化ツール，モデルベースシミュレーション，並列処理環境

A visual performance debugger for parallel programs

Kei HISANO, Noritaka OSAWA, Toshitsugu YUBA
Graduate School of Information Systems
The University of Electro-Communications
Chofugaoka 1-5-1, Chofu, Tokyo 182, Japan
E-mail: {hisano, osawa, yuba}@yuba.is.uec.ac.jp

Abstract　This paper describes a visual performance debugging system for parallel programs, focusing on its design concepts and its implementation. The debugger displays the history of program execution visually. The visual information is useful to understand the behavior of a program and to tune the performance of a program. The system is based on an internal model, which is constructed from log data The internal model enables the system to simulate the behavior of a program without re-execution and to give visual feedback immediately reflecting on the "debugging by demonstration principle".

英文 key words performance tuning, visualization tool, model based simulation, parallel programming environment

1 はじめに

並列プログラムでは、その実行時の動作を把握するのが難しいために、一般に逐次プログラムより開発が困難になる。これは、並列プログラミングにおいて、プログラマは逐次プログラムの作成時に比べて、使用できるPE（要素プロセッサ）の数が多くなり、プログラムの自由度が増えたことによって生じる。プログラマは、数十、数百、あるいはそれ以上のPEの情報を把握しなければならない。さらに各PEへの処理の割り当て、PE間の通信や同期といったことまで考えなければならない。特に、PE間の通信や同期は、これが不適切であると、プログラムが不可解な挙動を起こしたり、デッドロックを起こすといった問題につながる。また、通信オーバーヘッドがかえって処理速度を遅くするといった、いわゆる“パフォーマンスバグ”を生じる。こういった問題が、一般に逐次実行のプログラムに比べて、並列プログラムの実行時の全体の振舞いを理解することを困難にする。この問題を解決するため、複数のPEの状態や通信、同期といった数多くの情報をプログラマが管理する必要がある。それにはプログラムの実行時の振舞いを理解する必要がある。

実行時の状況を把握するためには、プログラム実行中の状態を出力するようにする。なぜなら、スレッドの処理粒度や処理時間は、ソースプログラムを見ただけではわかりにくいからである。この出力は、基本的に数字を主とした文字データである。しかし、人間の視覚系は文字情報より、図式情報のほうが処理を容易に行なえる。このことを考えると並列プログラムの動作を理解するために、視覚化システムを用いるのは適当な方法といえる。そのため、並列プログラミングの環境において、いくつかの視覚化システムが作成されている。しかし、これらの多くは、PE間の通信や同期、実行時の振舞いを表示するだけのものである。

視覚化システムの多くは、実行中の状態をログファイルに出力し、そのデータを表示する。本研究では、このログを工夫し、一歩進んで、視覚的操作をプログラムに反映できるようにする。こうすることで、効率の良い並列プログラムの開発が行なえる。

本性能デバッガを使って並列プログラムを開発する過程を図1に示す。ユーザはライブラリを使ってログデータを出力するソースプログラムを作成し、一度コンパイルし、実行する。性能デバッガは、このログデータから内部モデルを作成し、ユーザは試行錯誤によりチューニングを行なう。ユーザの操作した変更内容は、自動的にソースプログラムに反映され、新しいソースプログラムができる。このサイクルを繰り返すことによって、性能デバッグを行なう。

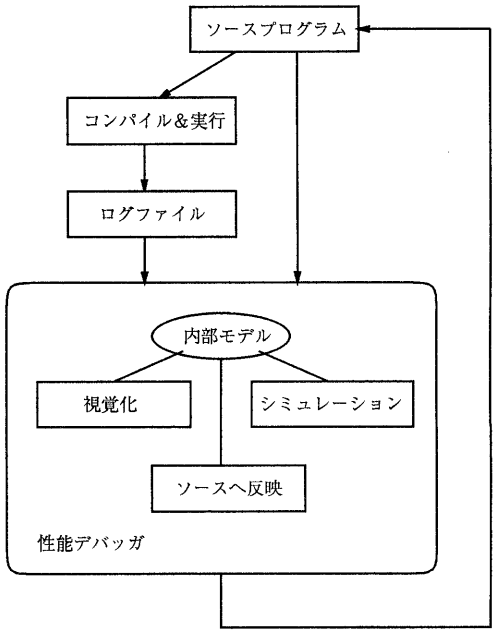

図 1: 開発過程におけるツールの位置付け

2 性能デバッガの設計思想

性能デバッガの概要を図2に示す。本研究では内部モデルベースの性能デバッギングを提案する。内部モデルは、性能デバッグに必要な情報をログデータから作ったものであり、これについては3.1節で述べる。

内部モデルベースの性能デバッギングとは、性能デバッグの操作を全て内部モデルに対して行なうことである。一度ログデータから作られた内部モデルは、このシステムで行なわれる視覚化、シミュレーション、ソースプログラムへの反映の全

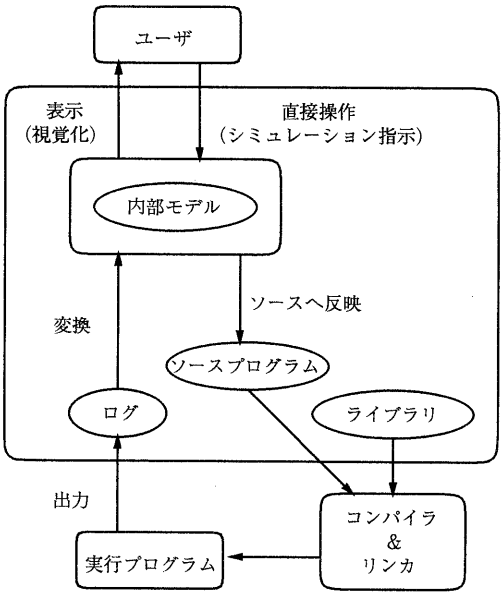

図 2: 内部モデルを用いた性能デバッガ

操作で使われる。図 2に示すように、ユーザの指示は画面を通してその内容が内部モデルに伝わり、内部モデルの変更にともなって表示データが更新される。この繰り返しが、内部モデルベースのシミュレーションにあたる。デバッグ作業の終了時には、内部モデルの内容がソースプログラムに反映される。このように内部モデルを用いることで、視覚化、シミュレーション、ソースプログラムへの反映の全て操作を一括して管理することができる。これらの個々についての詳細は 3節で述べる。

2.1 視覚化

視覚化ツールの機能として、プログラムの動的振舞いを効果的に表示する。並列プログラムの視覚化ツールには、通信相手、PE の稼働率を示したり、ガントチャートを使ったものがある。本デバッガではガントチャートをベースにし、スレッドの生成や、通信の状態も同時に表示できるものにする。視覚化ツールとしてのユーザインタフェースも考慮し、すべてマウス操作で行なえること、場合によってはメニューからでも操作できるようにする。

2.2 内部モデルベースシミュレーション

視覚的に得られる情報をもとに、マウス操作を基本とした変更を行なう。ユーザが行なった変更の結果、プログラム全体がどのように変化するか、内部モデルをベースにシミュレーションを行なう。

並列プログラムの性能向上のために変更する対象となるものは次のようなものが考えられる。

(1) 並列処理粒度

(2) データ割り当て

(3) スケジューリング

(4) 通信

(5) 同期

本研究では、これらの中で、変更が最終的に PE 割り当てに関する問題に帰着する場合を直接操作による変更の対象とする。具体的には、スレッド生成先の変更、並列処理粒度の変更、計算負荷の分散である。他のものはメニューなどにより変更の指示を行なう。

2.3 ソースプログラムへの反映

画面情報を操作した結果をソースプログラムに反映できるようにする。ソースプログラムへの反映はユーザが明示的に指示した場合にのみ実行し、機械的に処理できることなので、システムのほうで自動的に行なう。これは画面で確認しながら、シミュレーションした結果を手動で反映させていたのでは、そこでヒューマンエラーが発生する可能性があるからである。また、試行錯誤で複数の箇所を変更した場合、全ての変更を覚えておくのは困難である。したがって、ユーザの指示により、内部モデルをソースプログラムへ自動的に反映できるようにする。当然のことながら、ソースプログラムに変更が加えられるだけのログデータと、ソースプログラムの記述が必要である。反映手法については 3.5節で述べる。

3 性能デバッガの構成

3.1 内部モデル

本デバッガはログデータから作成した内部モデルによってデータを管理している。この内部モデ

ルは、一つのスレッド、あるいは送信や受信といった処理からなる。これらの処理の一つをオブジェクトとして扱う。オブジェクトには処理オブジェクト、スレッド生成オブジェクト、送信オブジェクト、受信オブジェクトがあり、それぞれのオブジェクトは性能デバッグに必要な情報を持っており、次のような要素を持つ。

- オブジェクトの種類
- 実行したPEの論理番号
- 通信相手あるいはスレッド生成先
- プログラム内変数の値
- 処理時間
- 直前のオブジェクトの開始時刻からの経過時間
- 時系列順に並べた次のオブジェクトへのポインタ

これらの情報はすべてのオブジェクトが持っている。オブジェクトの種類によってはこれ以外の情報も保持している。例えば、処理オブジェクトにおいては、その処理内で起こった送信や、生成したスレッド生成オブジェクトへのポインタを保持し、内部モデルはオブジェクトの関係を表す木を構成している。

すべてのオブジェクトは絶対的な時間は持たずに、処理時間と、直前のオブジェクトとの相対的な位置関係しか保持していない。したがって、あるオブジェクトを別の時刻の場所に移動させたとしても、前後のオブジェクトのポインタを変えるだけで、基本的にオブジェクトの移動は可能である。これにより、内部モデルベースでのシミュレーションを実現する。

あるスレッドを移動させる時、そのスレッドを生成したオブジェクトも、その生成先を変更する必要がある。また、そのスレッドから生成されるすべてのスレッドや通信も発信元が変更されることになる。このように、一つのスレッドが移動することを考えても、実際は非常に多くの変更が必要である。そして、すべての変更は、この内部モデルを操作することで行なう。

3.2 ログ出力用ライブラリ

ログはプログラム内でイベントが発生した時に書き出される。イベントは、スレッド生成、通信、関数（スレッド）などの開始、終了点にで発生する。内部モデルを構成するオブジェクトは、この開始、終了時刻で挟まれた時間幅をデータとして持ち、表示される。

内部モデルを作成するのに必要なログデータは次の項目である。

- イベントの種類
- 実行したPEの論理番号
- イベント発生時刻
- 監視したい変数
- 反映用のソースファイルへのポインタ

デバッガはこれらのデータを内部モデルに変換し、表示、変更に利用する。

性能デバッガを使ってデバッグする時は、ログを出力する必要がある。しかし、デバック終了後には、ログを出力する必要はなくなる。この切替を簡単にし、プログラマの負担を減らすためにも、ライブラリを用意する。ログの出力は、スレッドライブラリと融合した形のライブラリを用意することで、機種依存部を吸収する。

3.3 視覚化

内部モデルの視覚化は各PEを縦軸にとり、水平方向に時間をとるガントチャートをベースにする。一つのオブジェクトは一つの線（四角形）で表し、スレッドの生成や、送信といった処理は、どのPEと通信しているのかがわかるように斜線で示す。

画面の操作はマウスで行ない、クリック、ドラッグといった操作でオブジェクトを移動させる。一つの操作は直ちに内部データに反映される。

その他に視覚化ツールとして、表示の方法に関する工夫を行ない。ユーザに有用な情報も提供する。例えばプログラムの並列度や、PEの利用率といった統計的情報や、通信、スレッド生成のコストのような評価用の情報である。これらの情報について、必要な時に参照できるようにする。

3.4 変更

ユーザが行なう変更指示は内部モデルに反映される。内部モデルデータはそれだけでソースプログラムへの反映が可能なだけの情報を持ってっている。したがって、ユーザは画面で確認したものを次回の実行で得ることを期待できる。変更に関する詳しいメカニズムは3.5節で述べる。

3.5 反映手法

プログラムの変更については、ユーザが任意に書いたプログラムを変更の対象にするのではなく、3.2節で述べたライブラリを利用して、ログを採った部分を変更の対象とする。

変更については、例えば以下に挙げる場合が対象となる。

(1) **リテラルで PE を指定している部分**

ライブラリ呼び出して、スレッドの生成先などを直接指定し、そのリテラル部分を変更の対象とする。

```
ThreadID CreateThread(const PEID id,...);
```

CreateThread がライブラリで、そのスレッドを割り当てる PE を *id*で直接指定する。*id*を変更対象とする。2番目以降の引数はスレッドの名前やそれに必要なパラメータである。

(2) **配列を利用して PE を指定している部分**

ループ内の制御変数をインデックスとして、配列の要素によって、スレッドの生成先を指定している部分を変更対象とする。配列の初期値を変更することで PE の指定を変更する。

(3) **リテラルパラメータを持った関数を利用して PE を指定している部分**

スレッド生成先の PE を生成する関数で、性能向上のためのチューニング用パラメータを持った変数部分を変更対象とする。

3.6 シミュレーション機構

これまでに述べたように、実行時のログデータをもとに、内部モデルが作成され、それをもとに動的振舞いを視覚化している。ユーザはその画面情報を操作し、その結果は内部モデルに反映され、画面表示用のデータも更新される。これは、その時に要した処理時間のままで変更できると仮定して、性能デバッグに特化した、プログラムの実行を内部モデルベースでのシミュレーションすることができることを意味する。一度ログデータを採るだけで、再コンパイルすることなくプログラムの実行のシミュレーションができ、効率良いプログラムへの収束が早まることが期待できる。

処理の因果関係を無視して変更するようなこと、例えば、スレッドを生成する前にそのスレッドの処理を実行するといったことはできないが、制約条件を満たすように変更することで、そのプログラムを疑似的に再実行できる。しかし、あくまでもログデータを採った時と同じ時間で処理できるという仮定のもとで、得られたシミュレーション結果なので、変更したソースプログラムを再コンパイルした後、実行した結果が必ずしもシミュレーション結果と完全に一致するとは限らない。しかし、スレッドの大きさを小さな関数単位とすると、通信やスレッド生成のオーバヘッドは相対的に小さなものとなり、シミュレーション結果に近くなることが期待できる。

3.7 実装環境

ツールはX Window System ベースのアプリケーションにする。ツール自身は並列計算機には依存しない。計算機に依存するところはライブラリで吸収する。

具体的なツールに仕上げるため、今回は並列計算機として、電子技術総合研究所のデータ駆動型並列計算機 EM-4 を用い、同じくプログラミング言語 EM-C を対象としている。

3.2節で示したログデータが作れ、ソースプログラムとの対応さえとれれば、他の計算機および環境に対応できる。

3.8 性能デバッガの概要

前節までに、デバッガの設計について述べてきたが、この節では現在開発中の性能デバッガの概要について述べる。

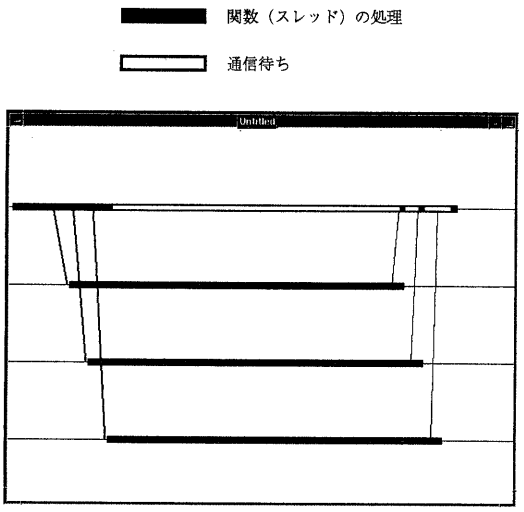

図 3: 性能デバッガの概要（1）

図 3はあるプログラムを 4 つの PE で実行した時のログデータを内部モデルに変換し、画面上に出力したものである。横に各 PE を表す線があり、黒い矩形や斜線がそれぞれオブジェクトと呼んでいる変更対象の単位である。このプログラムでは起動された PE が、他の 3 つの PE に、それぞれスレッドを生成し、その結果をスレッドを生成した PE（PE0 とする）が集めて結果を表示するというものである。この例ではスレッドの生成や、通信に必要な時間より、各 PE での処理の方がはるかに大きく、結果的に PE0 はほとんどの時間を他の PE からの結果待ちの状態で無駄がある。そこで、他の 3 つの PE で行なっている処理を少しずつ（約 20%）PE0 に移すと、図 4のようになる。このような変更の場合、プログラムはあらかじめ、使用する PE の数と各 PE で処理する負荷の大きさが変更できるようになっていなくてはならない。

この例におけるユーザの操作の一例はおよそ次のようになる。

- いずれかの PE の処理オブジェクトの一部

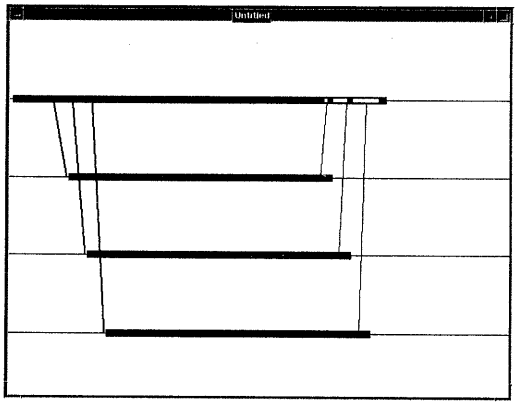

図 4: 性能デバッガの概要（2）

（この例では約 20%）をマウスでドラッグし、選択する。

- 選択した部分を PE0 の待ち状態のところへ移す。
- PE0 に処理オブジェクトが追加される（その分待ち状態が減少する）。
- 残りの PE についても少しずつ PE0 に移動させる。

この一つ一つの操作は画面に表示され、何をしているか確認しながら作業できる。そしてこの変更を施した結果が図 4である。

この場合の処理（内部モデル変更のメカニズム）はおよそ次のようになる。

- PE0 に処理オブジェクトはないので作成される。
- 移された負荷をその PE での処理の割合とする。
- オブジェクトが移動される度に割合を計算する。

これは、スレッドの生成や、通信のコストよりも、スレッドの処理のほうがはるかに大きかったため、このような変更が有効な例でであった。スレッドの生成や、通信のコストの方が、はるかに大きい場合は、逆に、使用する PE の数を減らした方が速くなる場合も考えられる。

4 おわりに

本稿では内部モデルに基づく性能デバッガの設計について述べた。原稿執筆時点で、3.7節で述べた環境において本デバッガを開発中であるが、ログデータを内部モデルに変換し、表示する視覚化ツールの段階であり、シミュレーション、ソースプログラムへの変更結果の反映といった性能デバッガとしての評価をするまでには至っていない。

今後はプロトタイプを完成させ、

- モデルベースシミュレーションの有効性
- ソースプログラムへの自動反映の有効性
- 対象プログラムの規模

といった点からシステムの評価をする。

さらに、内部モデルデータをプログラムで自動的に走査し、パフォーマンスバグを見つけるアシスタント機能や、それを自動的に変更して示す、デモンストレーション機能を実装して性能デバッグシステムとしての完成を目指す。

謝辞

データ駆動型並列計算機 EM-4 の実行環境を御提供頂いた、通産省電子技術総合研究所 計算機方式研究室のデータ駆動型並列計算機研究グループに感謝致します。

本研究の一部は、財団法人 大川情報通信基金の研究助成による。

参考文献

[1] Arndt Bode and Peter Braun, "Monitoring and Visualization in TOPSYS", Performance Measurement and Visualization of Parallel Systems, G. Haring and G. Kotsis (Editors), pp. 97–118, 1993 Elsevier Science Publishers B. V.

[2] Ian Glendinning, Vladimir S. Getov, Stephen A. Hellberg, Roger W. Hockney, David J. Pritchard, "Performance Visualisation in a Portable Parallel Programming Environment", Performance Measurement and Visualization of Parallel Systems, G. Haring and G. Kotsis (Editors), pp. 251–275, 1993 Elsevier Science Publishers B. V.

[3] Ralph Butler and Ewing Lusk, "User's Guide to the p4 Parallel Programming System", ARGONNE NATIONAL LABORATORY,1992.

[4] Virginia Herrarte, Ewing Lusk, "Studying Parallel Program Behavior with Upshot", ARGONNE NATIONAL LABORATORY,1992.