計算機アーキテクチャ 89−15
（１９９１．７．１９）

細粒度高並列プログラムの実行の視覚化

舘村 純一，　小池 汎平，　田中 英彦
{tatemura,koike,tanaka}@mtl.t.u-tokyo.ac.jp
東京大学 工学部

概要

本論文では、並列論理型言語 Fleng のデバッガ HyperDEBU におけるプログラム実行の視覚化について述べる。Fleng は細粒度で高並列な言語であり、その実行の観察・操作の難しさを解消するため、HyperDEBU は視覚化により実行状況の把握を支援し、バグの絞り込みを効率化する。この視覚化機能はユーザが知識を与えることによって低レベルから高レベルまでの視覚化をサポートし、知識の与え方に応じて効果的なデバッグを可能とする。HyperDEBU は制御の流れとデータの流れそれぞれについて視覚化を行なうが、そこで必要となる、動的に生成されるプロセスやデータを表示する手法についても述べる。

Visualization of Fine-grained Highly Parallel Programs

Jun-ichi Tatemura, Hanpei Koike, Hidehiko Tanaka
{tatemura,koike,tanaka}@mtl.t.u-tokyo.ac.jp
Faculty of Engineering, The University of Tokyo,
Hongo 7-3-1, Bunkyo, Tokyo, 113 JAPAN

Abstract

In this paper, we describe the visualization of parallel logic programs with our debugger HyperDEBU. Fleng is a fine-grained highly parallel language and is one of the family of parallel logic programming languages. To resolve the difficulty to observe and manipulate the status of a program, HyperDEBU helps a user to comprehend execution by using visualization, and to locate bugs efficiently. This debugger gets knowledge from the user in order to support many levels of visualization. HyperDEBU visualizes both control-flow and data-flow . Since goals and data themselves are created dynamically, it uses a technique of dynamic location of display objects.

1 はじめに

プログラムを実行させてみて予期せぬ動作をする場合、それをデバッグするにはまず実行の様子を把握することが必要である。特に高並列プログラムにおいては実行の流れが多数あるので、どこでどのようなことが起きているのかをまず理解することがより重要な課題になる。逐次プログラムの場合は、適度に情報をフィルタリングしながら一つの実行の流れを順に見ていけばデバッグが可能であった。これに対して、細粒度高並列プログラムでは、制御の流れもデータの流れも多数が分散して存在するので、それをそのまますべてトレースするのは困難である。フィルタリングして一部の流れだけをトレースしようとしても、多数の、しかも動的に生成・消滅する流れの中から何をいかにして選択すればよいかは非常に難しい問題である。このことから、逐次プログラムと比べて、細粒度高並列プログラムでは状況を把握することがいっそう困難かつ重要であるといえる。

細粒度で高並列なプログラムのデバッグの問題を解決するためには、まずプログラムの実行の巨視的な状態が見えることが重要であり、実行情報を抽象化したグローバルな視野が必要となる。このためには、デバッガが実行情報をユーザにいかに見せるかといった、プログラム実行の視覚化手法の問題を解決しなければならない。

一般に、プログラムの実行の視覚化としては次のようなものが研究されている。

- ビジュアルデバッガ
- アルゴリズム・アニメーション

ビジュアルデバッガとしては、Prolog のデバッガ PROEDIT2 [4] などが挙げられる。ビジュアルデバッガではプログラミング言語レベルの抽象度の図形が用いられる。

アルゴリズムアニメーションとしては、Balsa [5]、ESP のプログラム可視化システム [6] などがあげられる。アルゴリズムアニメーションは、アルゴリズムの理解、プログラミングの教育、仕様 / 設計の確認などに用いられ、プログラムの仕様を描画用プログラムとして与え対象プログラムにプローブを埋め込んで動作させるなどの方法により、設計・アルゴリズムレベルの抽象度の図形でプログラムの動作を表現する。

これらの手法を細粒度高並列プログラムのデバッグに適用する場合には、以下のような問題点について考慮しなければならない。

- ビジュアルデバッガ (抽象度低レベル)
  - 細粒度 / 高並列なプログラムでは大規模で複雑になり、理解が困難である。
- アルゴリズム・アニメーション (抽象度高レベル)
  - 完全な仕様が必要となり、手間がかかる、与える仕様のバグなどの問題がある。
  - 予期しない動作の視覚化がデバッグでは重要になる。
  - バグを絞り込んでいく過程で、より低レベルなビューも必要となる。

本論文では、マルチウインドウデバッガ HyperDEBU における並列論理型プログラム実行の視覚化機能について述べ、細粒度で高並列なプログラムの実行をどのように視覚化するかを論じる。

2 節では、視覚化の対象とする並列論理型言語 Fleng について、3 節では、そのデバッガ HyperDEBU について概説する。4 節で細粒度高並列プログラムの視覚化の問題をどのように解決するか、その方針と手法について論じ、5 章と 6 章でその具体的な手法について述べる。

2 並列論理型言語 Fleng

Concurrent Prolog や GHC などの Committed-Choice 型言語 (CCL) は論理型プログラミングを並列に実行するためガードの概念を導入して通信・同期が記述できるように制御機能を強化した並列論理型言語である。Fleng [1] もこの CCL の一つであり、他の CCL に較べてその言語仕様が簡潔になっている。Fleng はガードゴールを持たず、ヘッドのみがガードの働きをする。よってヘッドユニフィケーションだけで定義節がコミットされる。

Fleng は、個々のゴールが並列実行される細粒度並列言語である。いくつかのプロセスが静的に存在してデータを介して相互作用をし合うのではなく、ゴールは動的に生成・消滅していく。このために、ユーザがゴールの実行を観察したり操作したりするのは困難であり、解決すべきデバッグ上の問題となっている。

Fleng の実行のモデル化

デバッガはプログラムの実行をモデル化しユーザはそれを通してプログラムを観察・制御するものであると考えられるから、デバッグの問題を解決するには、細粒度並列プログラムの実行を表現するのに適したモデルを導入しなければならない。

従来の並列プログラムのデバッグ手法では、実行情報を表すのに抽象度の低いモデルしか用意されていないことに問題がある。我々は Fleng プログラムのデバッグのために、通信するプロセスとしてその実行をモデル化した [2]。このモデルでは任意の一つのゴール (top goal) と、そのゴールから生成される全てのゴール (subgoals) をまとめて一つのプロセスととらえる。プロセスの動作の様子は、外部からはプロセスの入出力としてとらえられる。このモデルには次のような特徴がある。

- 階層的 = プロセスは幾つかのサブプロセスに分割できる (幾つかのプロセスからより抽象度の高いプロセスが構成される)。
- 多面的 = ゴールの集合 / 計算木 / 入出力因果関係といった複数のビューを持つ。

これを概念図として表したものが図 1 である。

3 マルチウインドウデバッガ HyperDEBU

我々は、Committed-Choice 型言語 Fleng のデバッガとして、多次元的インタフェースを用いたマルチウインドウデバッガ HyperDEBU を開発した [3]。従来のマルチウインドウデバッガの多くは各プロセスに逐次プログラム用のデバッガを割

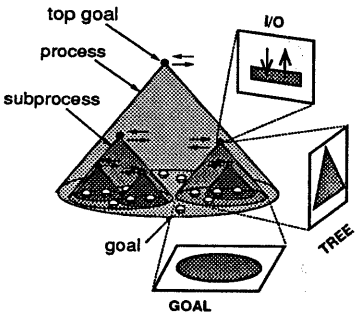

図 1: プロセスの概念図

り当てるものであった。しかし、単にウインドウを割り当ててプロセスごとの情報を分割しただけでは多次元的な情報を取り扱い得ない。このデバッガは、制御 / データの流れが形成する複雑なグラフ構造を観察 / 操作するための多様な視野としてウインドウを提供する。ユーザは、このウインドウ上に表現されたプログラムの実行情報を構成するリンクをたどることによってさらに希望するウインドウを開いてゆくことが可能である。

3.1 HyperDEBU の構成

図 2は HyperDEBU の全体像である。これは以下のウインドウから構成されている。

1. トップレベルウインドウ：プログラムの実行をグローバルに観察操作するウインドウ
2. プロセスウインドウ：任意のプロセスに割り当てることのできるウインドウ
   (a) TREE ウインドウ: プロセス内の計算木 (コントロールフロー) の観察・操作を行なう
   (b) I/O tree ウインドウ: プロセス内の入出力因果関係 (データフロー) の観察・操作を行なう
   (c) GOAL ウインドウ: プロセス内のゴール (スナップショット) の観察・操作を行なう
3. ストラクチャウインドウ: 任意の構造データに割り当てることのできるウインドウ

3.2 HyperDEBU の機能

HyperDEBU では、以下にあげる各機能が協調してユーザのバグ探索を支援する。

1. 多様な視野を用いたバグの絞り込み
   トップレベルウインドウとプロセスウインドウは、グローバルな視野からよりローカルな視野までをユーザに提供する。プロセスウインドウは、トップレベルウインドウに表示されている各プロセスから開くことができ、プロセスの3つのビューに対応したサブウインドウによって多様な側面からプロセスを観察・操作できる。これらのサブウインドウからサブプロセスを別のウインドウとして開くことができ、これによって効果的なバグの絞り込みが行なわれる。
2. プログラム実行の視覚化
   プログラムの実行の視覚化は、トップレベルウインドウのグローバルなビューとして実現されている。視覚化において何に着目して抽象化を行なうかはユーザの主観によるので、 HyperDEBU ではこれをブレークポイントという形でユーザが指定する。
3. ブレークポイント
   HyperDEBU では、デバッグにおける「ブレークポイント」を拡張して考え、デバッガがユーザから実行前に予め与えられた知識ととらえる。デバッガは、この情報をプログラムの実行制御・実行の視覚化・静的デバッグに活用する。
4. プログラム・コードのブラウジング
   ブレークポイントは静的な情報をもとにユーザが実行前に設定しなければならないので、静的情報の把握を支援する機能が要求される。

これらの機能の中で、実行の視覚化は、プログラムの実行状況の把握を支援することで、バグの絞り込みを効率化している。

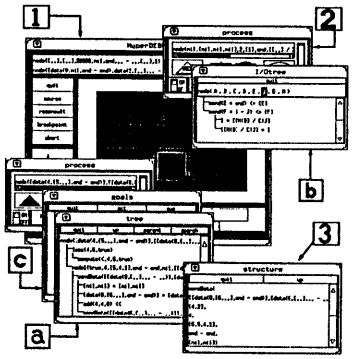

図 2: HyperDEBU の概観

4 HyperDEBU における実行の視覚化

HyperDEBU では、プログラムの実行の視覚化を行なうことにより、実行状況の把握を支援する。本節では、どのような方針のもとに視覚化を行なうかを述べた後、それを実現する手法についてふれ、次節でその詳細について議論する。

4.1 視覚化の基本方針

低レベルな視覚化を行なう従来のビジュアルデバッガや、高レベルな視覚化を行なうアルゴリズムアニメーションに対し、HyperDEBU では「付加的な知識」を与えることによって低レベルから高レベルまでの視覚化を行なうという方針を取る。これは、プログラム全部について完全な知識を与えなくても、

知識を与えない部分は低レベルな視覚化でサポートし、知識の与え方に応じて高レベルなデバッグを可能にしようというものである。

デバッグに利用する場合についていえば、アルゴリズムアニメーションのような高レベルの視覚化をソースプログラムのみから全くの自動で実現するのは困難であると思われる。これは、同じプログラムでもどの部分をどのように見たいかというユーザの意図が状況によって変化するからである。デバッガでは「見たいものを見たいように見せる」ことが重要であるから、ユーザの主観を反映させる必要がある。このためには、ユーザが視覚化のための知識を指定することは不可欠だが、ソースコードの解析結果を適用したり、何らかのテンプレートを用意するなどの支援により、半自動的に視覚化できることが望ましい。

4.2 視覚化手法

プログラムの実行の視覚化は、トップレベルウインドウのグローバルなビューとして実現されている。

視覚化を行なうべきものとしては制御の流れ、およびデータの流れが存在するが、CCL ではそれぞれについて、

- 制御の流れ：ゴールリダクション
- データの流れ：ガードとユニフィケーション

といった対応がとれる。それぞれの履歴は計算木と入出力因果関係という形で TREE ウインドウと I/Otree ウインドウの上に表現されている。しかし、CCL の様な細粒度高並列プログラムでは、個々のゴール・個々のデータを全て視覚化したのでは繁雑になり過ぎる。トップレベルウインドウでは、制御の流れに関しては、ある特定のゴールについてのプロセスのみを表示し、データの流れに関しては、そのプロセス間に存在する特定のデータの流れだけを表示することにする。ここで、何が「特定」かはユーザの主観によるところが大きいので、ユーザが指示を与える必要がある。HyperDEBU では、これをブレークポイントとして指定する機能を備えている。

また、Fleng などの CCL の実行の視覚化において注意すべきことの一つとして、動的にデータ・ゴールが生成されるために静的に配置が決まらないという点がある。このために、動的な視覚化手法が必要となる。

以降では、制御 / データそれぞれについての具体的な視覚化手法について、何がどのように表示されるかを述べる。ただし現在は、HyperDEBU ではデータの流れに関する視覚化は実装途中であるので、ここではその手法と予想される表示を用いて議論を進めている。

5 制御の流れの視覚化

5.1 何が視覚化されるか

トップレベルウインドウでは、特定のゴールにに関するプロセスのみを表示することにより、制御の流れに関するグローバルな視野を提供する。図 3 はトップレベルウインドウの表示である。これはプロセスの概念図 (図 1) を上から見たものとしてとらえることができる。

各プロセスは、ウインドウの中に表示された矩形で表現される。

- 矩形の模様はプロセスの状態を表す。白色・淡灰色・濃灰色はそれぞれ active, suspend, terminated を表す。
- 矩形の入れ子構造はプロセスとサブプロセスの関係を表す。
- マウスカーソルが矩形上に入るとゴール表示部にトップゴールが表示される。
- 矩形をマウスで操作することにより、それぞれプロセスウインドウを開くことができる。

これらの表示は、実行状態を反映して動的に変更され、プロセスの生成や状態変化、トップゴールの引数のデータの変化が把握できる。これにより、プログラムの実行状況を把握することができる。

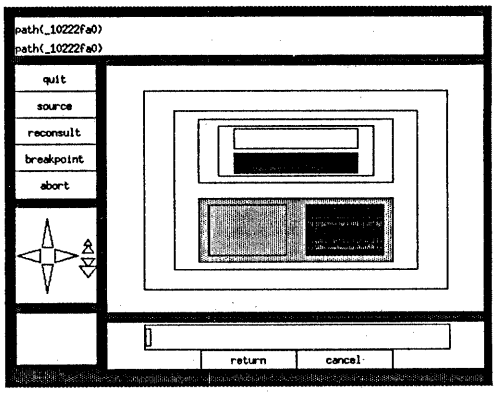

図 3: トップレベルウインドウ

ユーザによる指定方法

プロセスはプログラム実行中に現れるすべてのゴールについて考えられるが、この中でどれを矩形として表示するかはユーザがブレークポイントによって指定する。これによって、特定の制御の流れを大まかに観察することが可能となる。

5.2 どう配置されるか

トップレベルウインドウにおけるプロセス (矩形) の配置は以下のような規則に基づいている。

階層の深さ D ($D = 0, 1, 2, \cdots$) のプロセス P の中に階層の深さ $D+1$ のプロセスが N 個サブプロセスとして存在しているとする。この時、これらのサブプロセスはそれぞれ、P の横を x 分割、縦を y 分割した領域に配置される。ここで (x, y) は、$D = 2k$ ($k = 0, 1, 2, \cdots$) の場合、

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{cases} \begin{pmatrix} m \\ m \end{pmatrix} & \text{if } m^2 \le N < m(m+1) \\ \begin{pmatrix} m+1 \\ m \end{pmatrix} & \text{if } m(m+1) \le N < (m+1)^2 \end{cases}$$

$D = 2k+1$ ($k = 0, 1, 2, \cdots$) の場合、

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{cases} \begin{pmatrix} m \\ m \end{pmatrix} & \text{if } m^2 \leq N < m(m+1) \\ \begin{pmatrix} m \\ m+1 \end{pmatrix} & \text{if } m(m+1) \leq N < (m+1)^2 \end{cases}$$

となる (m は自然数)。

これによって、例えば $D = 2k$ の場合、プロセスの生成につれて矩形は図4のように増加していく。

この手法は、動的にプロセスが生成する場合でも比較的実現しやすく、しかも次のような特徴がある。

- 矩形中の面積ができるだけ有効に活用される。
- 矩形の形が保存されやすい。
  - 一つの矩形の中には互いに合同な矩形が均質に配置される。
  - x と y の差が高々 1 である。
  - x と y の大小が階層によって異なる。
- 矩形が増加した時にもとの矩形同士の位置関係が保存される。

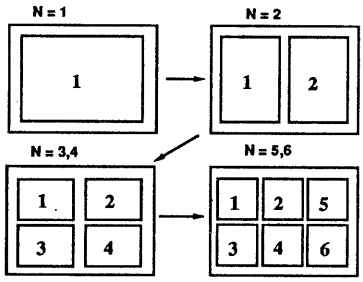

図 4: トップレベルウインドウにおけるプロセスの配置

5.3 具体例

Fleng プログラムの典型的なパターンとして、

- 問題を分割しながら並列に処理を行なう。
- 並行プロセスが特定の数だけ存在して通信をし合いながら実行が進む。

といったものがあげられるが、ここでは、それぞれについてサンプルプログラムを用いて制御の流れに関する視覚化を試み、その手法の有効性を評価する。

最初の例として、クイックソートを視覚化する。これは、ソートすべきデータを、ある値の大小で二分していくことで並列に処理が行なわれるプログラムである。

プロセスを表す矩形の中にはそれぞれ二つの矩形が存在する。これは、サブプロセスを意味しており、問題が分割されて処理が進んでいく様子を表している。分割する向きが階層によって違うので矩形が潰れるのを防いでいる。

次の例として、最短経路問題を解くプログラムを視覚化する。これは、各リンクにコストのついた有効グラフ上のあるノードから他のノードまでの最短経路を求めるプログラムである。ここでとりあげるプログラムは、各ノード毎に一つのゴールを割り当て、それらが通信をし合いながら解を求めていくというものである。

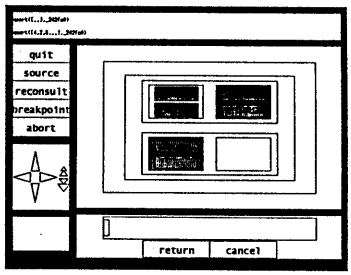

図 5: 制御の流れの視覚化例 1 – クイックソート –

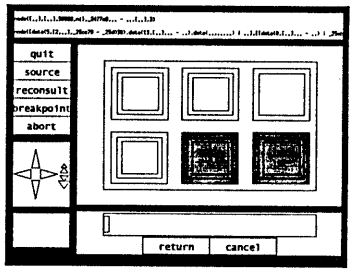

図 6: 制御の流れの視覚化例 2 – 最短経路探索 –

このプログラムの主要部分は各ノードに当たるゴールを生成する部分と、そのゴールが互いに通信をする部分に分けられるが、ここでは各ノードに当たるゴールを視覚化しているので、それらを生成する部分は抽象化されている。このため、最初の階層ではノードの数だけのプロセスが表示され、それぞれが再帰的に実行されていることがわかる。

5.4 問題点と課題

現在のトップレベルウインドウにおけるプログラムの実行の視覚化機能には次のような問題が残されている。

1. 各矩形がどのプロセスを表すのか判別しにくい。
2. 同階層のプロセス間の関係がわからない。
3. 階層が深くなり過ぎる。

問題点 1, 2 に関しては、次節で議論するデータの流れに関する視覚化を適用すればプロセス間の関係がより明確になり、各プロセスの識別にも役立つようになるであろう。また、色や形で区別できるようにブレークポイントに属性を設定できるようにすることが考えられる。

問題点 3 としてあげたように、大規模なプログラムでは、矩形の入れ子構造が深くなることがある。このような場合、表示すべき矩形がある程度小さくなると画面には表示されなくなる。トップレベルウインドウでは画面の一部を拡大する機能を

持つので、細かい部分に注目したい場合はこの機能を用いればよい。しかし、この機能だけでは離れた場所を同時に観察できないという問題が残っている。これに対しては、プロセスウインドウに同様の視覚化機能を持たせることを検討している。これにより、ある程度の深さから先はプロセスウインドウによって視覚化するようにことができる。このほかに、再帰的なゴールに関するプロセスを特別に扱えるようなブレークポイントを設けることも考えられる。

6 データの流れの視覚化

6.1 何が視覚化されるか

データの流れをグローバルに観察するには、トップレベルウインドウで視覚化されたプロセス間に存在するストリーム通信に着目することが考えられる。

ストリーム通信は次のようにして行なわれる。複数のゴールが変数を共有している時、その中のどれか一つのゴールが何らかの構造データをユニフィケーションによって変数に代入する。これがストリーム出力となる。他のゴールはその変数に値が代入されるのをガードのつけられたユニフィケーションを用いて待ち合わせる。変数の値が確定すると、その値を読んで処理を行なう。これがストリーム入力となる。構造データの中には未定義変数が含まれていて、これを介して次の通信が行なわれる。

このようなストリーム通信の様子を表現するために、ストリームが生成される様子、それが分配される様子、データの入出力が行なわれる様子を視覚化する必要がある。

ストリームが生成されるのは、ボディーゴールに新たな共有変数が生まれた時である。図7 はストリームが生成された時に画面に視覚化される図形の様子を表したものである。図中の(a) の定義節 では、p がリダクションされることにより、共有変数Sを持ったゴールq, r が生成される。このSがストリームとして用いられる場合、ゴールを矩形、ストリームを線で表すと、定義節の下にあるような図形が形成される。(a) は最も基本的な例であるが、(b) は3個以上のゴールが一つのストリームを持っている場合、(c) は一つのゴールが複数のストリームを持っていてそれぞれを他と共有している場合である。これらの図形はストリームに生成時に画面上に表れ、ゴールがリダクションされるにつれて形を変えていく。

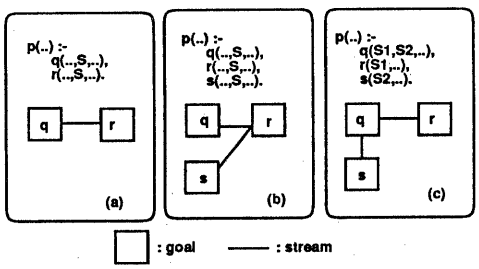

図 7: ストリームの生成

図 8-(a) はゴール p のリダクションにより、ストリームがq, r に分配される様子である。p がブレークポイントにより、プロセスとして視覚化された場合には右側の図のように階層が生じ、q,r はプロセス p を表す矩形の中に生成される。

図 8-(b) は、ゴール p がリダクションされて変数S に構造データ [X|S1] が代入されることによりストリーム出力が行なわれる場合である。未定義変数である S1 は q がストリームとして用いることで通信が継続される。データを円で表すと、画面上では図の左下のように変形される。

図 8-(c) は、ストリーム入力の例である。ゴール p は引数が [X|S] という構造データとして値が確定するのを待ってリダクションされる。これによってストリーム入力が実現される。S は、 q がストリームとして用いることで通信が継続される。この時の図形は図の左下のようになる。

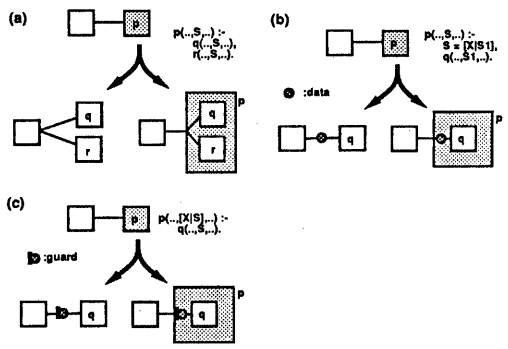

図 8: ストリームの分配・出力・入力

ユーザによる指定方法

このようなデータの流れの視覚化を実現するには、表示すべきストリームをどのように指定するかが問題となる。各定義節のストリーム通信を実現する各部分をブレークポイントとしてユーザがそれぞれ設定するのは繁雑である。これを解消するために、プログラムの静的情報を用いて何らかの支援をする機能が必要であろう。ユーザがどの述語のどの引数をストリームとして着目するかを指定すれば、この情報を用いてプログラムを解析し、各定義節のどの部分を視覚化すればよいかを自動的に決定する機能が考えられる。さらに、この解析の段階で与えられた知識とプログラムとの間に矛盾が見つかれば、静的にバグが検出されることにもなる。

6.2 どう配置されるか

以上に述べた方式で視覚化された図形を表示する際に、重要な問題となってくるのがそれらをどのように配置するかである。これには次のような要件を満たした方式が望ましい。

- 動的な配置に適している
- ストリームの交錯を避ける
- 面積を有効に利用する
- 実現が容易である

本節では、これらの要件を満たすような一方式として、以下にあげる手法を提案する。

プロセス・ゴール・データのグループ化

ここで提案する手法では、前節で述べたコントロールフローを視覚化するプロセスの配置方法を拡張して、プロセス・データ・ゴールのグループ化を行なうことによりデータフローの視覚化を導入する。

互いにストリームでつながれたプロセス、データ、ゴールをひとつのグループとし、このグループを一単位として前節の手法と同様に配置を行なう。グループが生成されるのはストリームが生成された時、または単独のプロセスが生成された時である。グループに属しているプロセス、ゴールおよびデータはそのグループに割り当てられた矩形領域内に配置され、ストリームを表す直線で互いにつながれる。これにより、互いに関係のあるプロセスはまとまって配置され、互いに無関係なプロセス同士のストリームが交錯することはなくなる。

視覚化したいストリームを何も指定しない場合は、一つのプロセスが一つのグループとなるので、制御の流れのみを視覚化した場合と同じ配置になる。これにより、制御の流れの視覚化とデータの流れの視覚化を融合して実現することが可能となる。

グループ内の配置

グループ内のプロセス・ゴール・データは互いにストリームで連結されている。このようなグラフ構造をリンクの交差が少ないように配置するのは、ノード数に関してNP完全な問題といわれている。ここで行なわれる視覚化は動的に行なわれるので、静的に配置する場合と以下の点で異なってくる。

- ノードが一つ増えるたびに最適な配置を求める計算を行なうのは効率上望ましくない。
- 新しいノードが現れる前と後の配置に対応がとりやすく、アニメーションとしてグラフが自然に変化していくことが重要になる。

このため、最適な配置が得られなくても、動的にノードが増えていく様子を効果的に表現することを重視した手法が望まれる。このような手法としては、典型的なパターンを効果的に処理し、それ以外の場合もある程度対処できるものを選択することが重要である。そこで、典型的なプログラムの特性を考え、それに基づいて手法の検討を行なう必要がある。我々の研究室で開発されたアプリケーションプログラムを観察すると、表示すべきグラフは、以下のような特徴を持つと思われる。

- ノードから出るリンクは2～3本程度：ゴールの引数の個数は通常一桁で、ストリーム通信として使っている引数の数はそれよりも少ない。プログラムの大きな流れをつかむために、ユーザが注目したいストリームの数はそのうちの2～3本程度であることが多いと思われる。ただし、ストリームがブロードキャストされる場合はノードから出るリンクの本数が多くなる。本数が多い場合は配置法によらず難解になるであろうから、2～3本の場合を重視して考える。
- 中心となる構造は列か木が多い：プログラムの大域的なデータの流れを表すと、ノードがリンクによって一列に、もしくは木構造をなしてつながってデータフローの中心をなし、それに何本かのリンクが加わっているものが典型的である。差分リストと呼ばれるプログラミング手法がその一例である。
- 新しいノードは近傍のノードとのみ直接つながる：リダクションによってゴールが生成された時に生じるリンクは、親ゴールから受け継いだものか、子供同士にできた新しいリンクである。いきなり遠くのノードと直接つながることはあまり考えられない。

ここで提案する手法は、ノードが増えるたびに矩形を分割していく方法である。グループに割り当てられた矩形領域は、ノードの数だけの矩形領域に分割され、それぞれの中央に各ノードが配置される。あるノードが二つのノードに分割されたときは、そのノードのおかれた矩形領域が生成された時の分割面と垂直にその領域を分割する。各分割時には、各矩形の面積が同じになるように再配置が行なわれる。図9はこのようにしてノードが生成されていく様子を示したものである。楕円で囲まれた部分がリダクションによって変化したノードである。

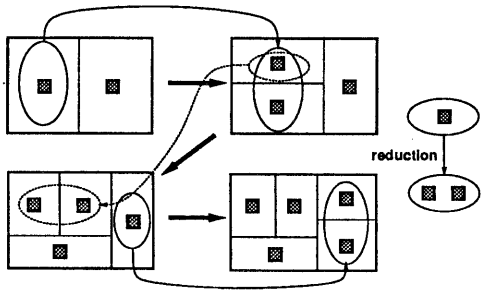

図 9: プロセス・ゴール・データの配置手法1

二つに分割された矩形にそれぞれどちらのノードを配置するかは、そのノードが他のどのノードと連結されているかで決定される。分割した線をy軸と考えると、ノードに直接連結しているノードのx座標の平均値の大きい方が正の側に配置される。図 10 はこの手法を用いた配置の例である。そのうち、ノードaがdとeに分割される時を例にとってみる。y軸は直線acにあたる。dはcに、eはbに直接連結しているのでeの方が右側の領域に配置される。

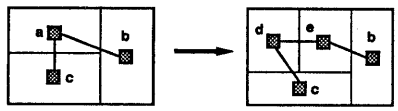

図 10: プロセス・ゴール・データの配置手法2

この手法により、以下のような特徴が得られる。

1. 新しいストリームができる時は、一つのノードから分裂してできたノード間にできるが、分割法によってこれらは隣

合う。

2. 1によって隣合ったノードの一方が分裂して子ノードがそのストリームを受け継ぐ場合も、分割面を垂直にすることで隣合う。
3. それ以前に生成されていたストリームを親から受け継ぐ時には、分割面の選択によってある程度対応できる。
4. 生成されたノード以外の構造は変化せず圧縮されるだけなので、アニメーションに適している。

6.3 制御の流れの視覚化との融合

データの流れを視覚化する際には、データとゴールだけでなく、プロセスを表す矩形も同時に扱える。ブレークポイントの選択により、制御の流れとデータの流れを融合して視覚化することができる。これにより、制御の流れ、データの流れの一方のみを表示するよりも以下の点で効果がある。

- プロセス間の関係がわかりやすくなる：ストリームでつながれたものは同じグループとして近くに配置される。また、ストリームの連結によってプロセスの判別がしやすくなる。
- データフローのグラフの繁雑化を避ける：一番上の層だけデータフローを表示し、プロセスとして抽象化された内部は扱わない。

6.4 例による考察

ここでは、典型的な例題についてどのような表示になるか、その予想図を示し、ここで提案する手法を評価する。

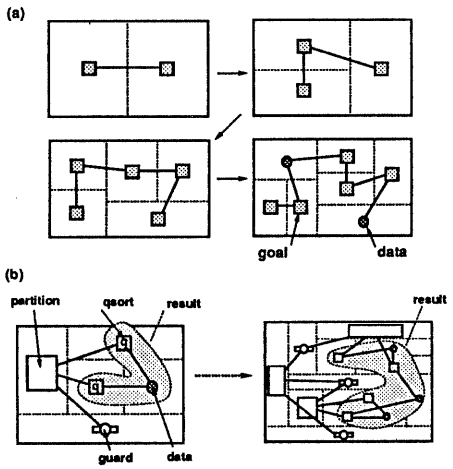

図 11: データの流れの視覚化例

図 11-(a) の例は差分リストを用いて探索問題の解を並列に求めている様子を視覚化したものである。ゴールはストリームによって一列につながれ、探索木にそってサブゴールを生成していく。解を発見したゴールはストリーム (差分リスト) にデータをつなげて終了する。

図 11-(b) はクイックソートの例で、解を収集するためにストリームでゴール qsort が一列につながっているのに加えて、問題を分配する partition からのストリームが視覚化されている。

6.5 問題点と課題

ここで提案したデータの流れに関する視覚化手法の問題点としては、以下のようなものが考えられる。

1. 大規模なプログラムになると、データの大きな流れがいく種類か同時に重なり合う。
2. ストリームにデータが出力されるとノードが単調に増加する。
3. プロセスを表す矩形がつぶれてしまうことが多い。

問題点 1 に関しては、ストリームを表す線を色分けすることで判別しやすくできるであろう。問題点 2 を解決するために、ストリームでつながれたデータを表すノードを縮退させて一つにまとめてしまうことが考えられる。縮退時に履歴を保存するようにしておけば、そのノードからウインドウを開いて、データどのようなゴールによって作られたかを調べることができるであろう。問題点 3 に関しては、データ・ゴール・プロセスについて配置面積に重みづけしてプロセスに割り当てられる矩形領域を大きめにすることにより、ある程度軽減させることはできよう。

7 おわりに

本論文では、 HyperDEBU における細粒度高並列プログラムの視覚化手法について述べた。

HyperDEBU は現在アプリケーション開発に試用することにより評価を行なっているが、データの流れに関する視覚化は現在開発中であるので、その実装を進める予定である。また、この他にパフォーマンスデータの視覚化も検討を進めている。

参考文献

[1] Nilsson, M. and Tanaka, H.: *Fleng Prolog - The Language which turns Supercomputers into Prolog Machines.* In Wada, E.(Ed.): Proc. Japanese Logic Programming Conference. ICOT, Tokyo. June 1986. p209-216.

[2] 舘村, 田中: 並列論理型言語 FLENG のデバッガ, PROCEEDINGS OF THE LOGIC PROGRAMMING CONFERENCE '89, pp133-142 (1989).

[3] 舘村, 小池, 田中 : 並列論理型言語 Fleng のマルチウインドウデバッガ HyperDEBU, JSPP '91 (1991).

[4] 森下真一, 沼尾雅之: PROLOG の視覚的計算モデル BPM とそれに基づくデバッガー PROEDIT2, PROCEEDINGS OF THE LOGIC PROGRAMMING CONFERENCE '86, pp.177-184 (1986).

[5] Brwon,M.H.: Exploring Algorithms Using Balsa-2, IEEE Computer, Vol.21 No.5,pp.14-36 (May 1988).

[6] 市川至, 小野越夫, 毛利友治: プログラム可視化システム, 情報処理学会論文誌, Vol.31 No.12 pp.1801-1811 (1990).