2006－EVA－17 (6)
2006／6／20

グリッドコンピューティング環境におけるプロセス連携機能を用いたジョブ管理システム

越本 浩央[1], 義久 智樹[2], 金澤 正憲[2]

[1] (現) クォンツ・リサーチ株式会社, (元) 京都大学大学院情報学研究科
[2] 京都大学学術情報メディアセンター

摘要

グリッドコンピューティングを利用する上で分散プロセスを制御するジョブの記述は避けることが出来ない．しかし，ジョブの記述についてグリッドでは一般的なモデルを提示していないか，あるいは古典的なバッチ形式のものに過ぎない．本研究では圏論の結果を設計・実装に落とすことで，一般化したジョブモデルを提示し，同時に対話性を供えたプロセスの連動機能を実現する．

Job Management System Using Process Syndication Function on the Grid Computing

Hiroo Koshimoto[1], Tomoki Yoshihisa[2], Masanori Kanazawa[2]

[1] (Current) Quants Research Inc. (Former) Graduate School of Informatics, Kyoto University
[2] Academic Center for Computing and Media Studies, Kyoto University

Abstract

On the grid computing, it is not an avoidable essence that we describe job-details including mediation between distributive processes. However, The Grid Technology has no general job-model or only has a classical batch style job-model. In our research, category theory is mainly treated for designing and implementing a job system, and then the general job-model could be presented, that realize interactive processes.

1．背景

分散環境技術，中でもグリッドコンピューティングは，機種特有の性質を越えて機能・資源を共有することを可能にする．この目的のためにグリッド技術では，砂時計モデル[1]に基づく研究開発が行われている．砂時計モデル[1]の役割は，機種や応用ソフトを越えた相互接続性を，中核となるプロトコルの下位・上位に展開するレイヤで実現するものであり，実際に Globus Toolkit[2]を利用したグリッドシステムでは，Globus[2]を中核として下位レイヤにおいて固有のハードウェア資源に合わせた実行環境を，上位レイヤにおいては Globus[2]を利用した応用ソフトを構築して，資源の流動性と応用ソフトの環境独立性を実現している．

グリッド技術を利用する場面で重要になってくるのがジョブである．バッチ形式と対話形式の２通りが想定されるが，いずれにしろ如何なる値をどのように扱うのかという情報を記述しなくてはならない．当然これは環境から独立した情報として記述されるべきである．しかし計算対象となるデータは一般的に固有の環境に記録されたものを想定することが多い．処理の内容であるプログラムについても，並列実行の観点で環境独立性が望まれるが，特定の環境下で複数プロセッサを割当てる指示を記述するのが実情である．この状況は特に理論的な困難によるものでは無いが，運用の現場，特に潤沢な計算資源を持たない環境では，スケーラビリティの点で課題となってくる．そこで，グリッド上のジョブについて環境や応用から独立し，なおかつ効率の点で並列動作が保障されるモデルが期待される．

２．研究の目的

分散環境下のジョブモデルについては既に存在しているが，グリッド環境のように資源性能のバラつきと粒度の差が大きな場合について，特に一般的なモデルは確立されていない．本研究ではこの点について，圏論の枠組みで分散環境下のジョブを分析し，一般的なジョブモデルを検討した．これによりグリッド環境下，特に大小様々な計算資源が混在した環境でジョブを効率的に実行するためのジョブフレームワークを実装することが可能となった．

また応用上は対話的なジョブを実行する場合がある．例えばパラメータを変更しながら結果を人間が判断するようなシミュレーションである．この状況も一般化したモデルの中で議論することが出来る．

平行プロセスの圏論での扱いは Turi&Plotkin[3] による，分配則を用いた並列計算言語に遡ることが出来る．Turi&Plotkin[3] はλ計算で記述した内容を矛盾しないようにπ計算，つまり代入を含めた操作的意味論，へと写像するための自然変換を導入した．またそれより以前，Moggi[4] は同じ道具（モナド）を使って副作用を伴う計算や非決定性などがλ計算として記述出来ることを示し，応用上不可欠である多くの雑多な計算を参照透過な形で実装する手法を示した．近年，Power&Watanabe[5] はモナドとコモナドに関する分配則を一般化したが，本研究はこの分配則を分散プロセスに適用したものである．

３．分配則によるジョブの記述

任意の処理を二つ含むジョブを考える．ここでは処理を式(1)のように書く．

$$\begin{aligned} f &: x \to y \\ g &: y \to z \end{aligned} \qquad (1)$$

それぞれの写像は入力を受け出力を返す UNIX フィルタのようなプロセスと考えれば良い．この２つのプロセスを接続したジョブを考える．つまり f の出力 y を，g の入力 y に繋ぐ場合，共通の実行環境下にあるプロセスであれば $g \cdot f$ とすれば十分だが，実際は別のメモリ空間に存在する y であるため結合が出来ない．

Moggi[4] によれば出力を副作用として考えることで式(2)のような記述を得ることが出来る．

$$\begin{aligned} \hat{f} &: x \to Ty \\ \hat{g} &: Ty \to z \end{aligned} \qquad (2)$$

ただしTyとはyのモナド表現であり，ここでは共有メモリのようなものに相当する．これで2つのプロセスを接続したジョブの記述が可能となる．しかし決定的な問題もある．Tyはプロセス$\hat{f}$と$\check{g}$から共通して参照することが出来る固有の情報でなければならない．ソースコード上で言えば大域変数のようなものである．現実問題として全てのプロセスがこのような前提の上で記述されることは望めない．

次にこの大域変数を局所化する工夫を行う．これにはUustalu&Vene[6]にあるコモナドが利用でき，ジョブは式(3)になる．

$$\begin{aligned} \hat{f} &: x \to Ty \\ \check{g} &: Dy \to z \end{aligned} \qquad (3)$$

ここでDyは入力キューやイベントのようなもので，先行する任意の演算列とは独立して現在の値を取得する仕組みを与える．

ここまでをまとめると，$\hat{f}$が計算結果を出力バッファに書き込み，$\check{g}$が値を入力バッファから読み出す，という流れが確認出来る．ところがこのままでは$\hat{f}$と$\check{g}$の接続が型付λ計算上は成立しない．

分配則は以上の状況設定において，例えば出力バッファを入力バッファに切り替える意味を持った自然変換である．分配則は式(4)に従う．

$$\lambda : DT \to TD \qquad (4)$$

実際このλが抽象化していることは，UNIX シェルがパイプラインに行っている操作などである．あるいはこれまでの議論をネットワーク上に分散したプロセスに置き換えれば，ソケットの send/recv がモナドへの演算，コモナドへの演算に相当し，分配則はルータに相当する．

改めて分配則によるジョブの記述，つまりプロセスの接続を書くと，プロセスは式(5)，ジョブ全体は式(6)で与えられる．

$$\begin{aligned} \hat{f} &: Dx \to Ty \\ \hat{g} &: Dy \to Tz \end{aligned} \qquad (5)$$

$$\begin{aligned} \mu(T\hat{g})\lambda(D\hat{f})\delta \quad &: \quad Dx \\ &\to \quad DxDx \\ &\to \quad DxTy \\ &\to \quad TxDy \\ &\to \quad TxTz \\ &\to \quad Tz \end{aligned} \qquad (6)$$

これで個々の計算は実行環境から独立した記述を得ると同時に，計算の接続規則もまた独立していることが確認出来る．つまり，ジョブを構成するプロセスの依存関係は計算内容に依存しない形で記述可能であることを示している．

4．分散環境下のプロセス連動レイヤ

分配則が計算の接続内容を，個々の計算とは独立して記述出来ることから，この機能を1つのレイヤとして独立させることが可能である．このレイヤを便宜上，プロセス連動レイヤと呼ぶ．プロセス連動レイヤが備えるべき操作的意味は次の通りである．

- 結果を収集する．
  - ➢ $Dx\ Ty$の受信．
- 接続規則を適用する．
  - ➢ 受信したパタンから送信先を決定．
- プロセスに情報を通達する．
  - ➢ $Tx\ Dy$の送信．

具体的な要素技術を比喩に用いれば，ル

ータが果たす役割と考えて構わない．同様の操作的意味論をプロセスのメッセージング上で展開しているに等しい．

(a)プロセス出力の待ち合わせ

ここで２つのプロセスから入力を受けるプロセスが存在したとする．プロセス連動レイヤは構わず受信したメッセージをパタンマッチして送信する．個々のプロセスは Dy 取得のため同期するか，あるいは Dy 依存の計算を遅延するため，メッセージングの段階でこれらの心配をする必要は無い．また個々のプロセスにおいても，実装上ほかのプロセスの出力制御を気にかける必要は無い．正確にはコモナドであるため気にかけることが出来ず，素直に同期処理に入るより他無い．逆にシステム実装の観点では，コモナドが厳密に排他制御を行い待ち合わせを解決しなければいけない．

T=1, Process1→(連動)→Process3(Wait)
T=2, Process2→(連動)→Process3(Run)

(b)対話形式ジョブ

次に対話的なジョブを考える．前述の連動機能においてユーザの数学的構造は加味されていないため，本質的には対話的な状況を想定してはいない．しかしこれは単純な手法で解決可能である．ユーザの情報認知を入力 x，反応を出力 y とするプロセス $f:x\to y$ を考える．GUI 等を介して連動プロトコルを話せば，ユーザというプロセスは連動レイヤの枠内で扱える．ジョブ中の所望の連結部分にこのユーザプロセスを挟むことで，対話性を実現することが可能である．

Process1→ユーザ→Process2→…

(c)リアクティブシステム

更に発展的な課題としてリアクティブシステムを考える．対話的であるだけでなく，全てのコンポーネントが同時的に進行するジョブである．これもまたプロセス側を工夫することで解決される．この場合，計算過程が時間変数を伴う１ステップとなるようにモナド・コモナドを実装する．具体的にはデバッガ相当の機能を利用してプロセスをインタラプトするか，あるいはイベントハンドラでプロセスをラップして，イベントをバッファリングするかである．

Process1(t)→Process1(t+1)→…
↓ ↓
Process2(t)→Process2(t+1)→…
↓ ↓
Process3(t)→Process3(t+1)→…

いずれの発展的状況についても，プロセスをラッピングするモナド・コモナドの実装が重要となる．詳細は次項で述べる．

5．システム実装

プロセス連動レイヤは個々の計算の詳細に立ち入らないが，形式として式(7)を前提としている．

$$\hat{f}:Dx\to Ty \qquad (7)$$

そこでネイティブなプロセスである式(8),

$$f:x\to y \qquad (8)$$

を持ち上げるレイヤが必要となる．本研究では Linux マシン上で動くネイティブプロセスと Web ブラウザ上で動く Java Applet を持ち上げ対象として，それぞれにモナドとコモナドのレイヤを用意した．

(a) モナド

ここで扱うモナドは応用上の観点から特に，Kleisli 構築によって得られるモナドに限定する．モナドは式(9)の３つの演算について閉じていなければならない．

$$\eta : 1 \to T$$
$$\mu : TT \to T \qquad (9)$$
$$j_T : JT \Rightarrow T'J$$

特に３つ目の処理（自然変換）は，任意の処理 $j : C \to C'$ をモナド T に作用するよう持ち上げたものである．この律に従えば任意の計算を記述上で参照透過に保つことが可能となる．なお処理 $j : C \to C'$ の中身は $f : a \to Tb$ で定まる具体的な写像である．

今回はモナドを計算状態のシリアライズとして実装した．分配則ではルーティングルールのマッチングを行うために，具体的な計算結果ではなく計算過程の情報（$Dx\ Ty$）が必要となる．ここで Dx は要求された処理であり，Ty は処理された結果である．連動レイヤは Ty を次の指示へと変換するので継続情報が保持される必要がある．幸いにも継続計算はモナドであるため，分配則に則った計算で利用出来る．

ちなみに連動レイヤを特定のマシンが管理する必要は無い．もし計算プロセス自身が次の配信先を特定出来るなら，自らルーティングを行っても理論的に問題は無い．応用上，トラフィック負荷の低減も期待出来る．

またネイティブプロセスに対するモナドでは，ptrace によってプロセスをフックしてメモリ空間から値を peek している．

(b) コモナド

モナドと同様に Kleisli 構築によるコモナドを扱う．式(10)について閉じていなければならない．

$$\varepsilon : D \to 1$$
$$\delta : D \to DD \qquad (10)$$
$$j_D : D'J \Rightarrow JD$$

ここで任意の処理 $j : C \to C'$ の要素となる写像は $f : Da \to b$ である．

コモナドは計算状態のデシリアライズとして実装した．モナドによって運ばれてくる継続処理は連動レイヤを介してコモナドとなる．ここには処理の記述が施されていなければならない．今回はイベントストリームとして実装した．Uustalu&Vene[6] らの研究でも示されている通り，計算の文脈を保持するストリームはコモナドである．なお実装に際しては大規模なイベントに備えた機構を考えなければならない．残念ながら今回の実装では見送った．モナドと同様にネイティブプロセスには ptrace によって値をメモリ空間に poke している．

(c) 分配則

先ほど述べた通り，厳密に分配則を実現する連動レイヤが必要なわけではない．大規模なデータを特定のリソースに伝達する場合，常に連動レイヤを中継することは明らかに無駄である．幸い分配則は環境から独立して記述されるため，連動がどこで行われようと本質的な差異は存在しない．今回は明示的にバースト転送フラグが付加された処理については，計算プロセスが直接つぎのプロセスと連動する構成になっている．

なおこれらの機能は全て XML-RPC によって実現している．SOAP を利用する方がデ

ータ構造の自由度は高いが，継続渡し形式(Continuation Passing Style)で処理を記述する仕様であるためXML-RPCであっても特に困難は無かった．

6．まとめ

本研究では，分配則によって分散環境下のジョブの振る舞いが記述出来ることを確認し，ジョブの一般モデルを提案した．またこれを効率的に実行する枠組みとして，プロセス連動レイヤとその仕組みを述べた．これにより，並列実行時の待ち合わせ・対話的手続き・リアクティブシステムが実現出来ることを説明した．最後に実装上の工夫と要点を簡潔に紹介したが，効率の点では評価・分析がまだ必要であると考えられる．

なお今回は明記出来なかったが，本システムはヘテロジーニアスな環境下での実行を想定している．その場合，不定期に参加するリソースに合わせて柔軟なタスク割当を行うことが一つの目標である．継続渡しとプロセス連動機能の主旨は本来そこにあると考えている．

[1] I.Foster, C.Kesselman and S.Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations", International J.Supercomputer Applications 15(3), (2001)

[2] I.Foster, "Globus Toolkit Version 4: Software for Service-Oriented Systems", IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779 pp.2-13, (2005)

[3] D.Turi and G.Plotkin, "Towards a Mathematical Operational Semantics", Proc. LICS 97 pp.280-291, (1997)

[4] E.Moggi, "Notions of computation and monads", Information and Computation 93 pp.55-92, (1991)

[5] J.Power and H.Watanabe, "Distributivity for a monad and a comonad", Electronic Notes in Theoretical Computer Science Vol.19, (1999)

[6] T.Uustalu and V.Vene, "Comonadic functional attribute evaluation", TFP2005, (2005)