Datarolマシンにおける並列展開戦略

星出，高秀
九州大学システム情報科学研究院知能システム学部門

日下部，茂
九州大学システム情報科学研究院知能システム学部門

谷口，倫一郎
九州大学システム情報科学研究院知能システム学部門

雨宮，真人
九州大学システム情報科学研究院知能システム学部門

出版情報：並列処理シンポジウム，1993年，pp.347-354，1993-05
バージョン：
権利関係：

Datarolマシンにおける並列展開戦略

星出 高秀† 日下部 茂 谷口 倫一郎 雨宮 真人
九州大学大学院総合理工学研究科
† 現在：日本電信電話(株)

概要: 本稿では，Datarolマシンにおける並列展開戦略に関して，スケジューリングと負荷分散の2点から検討する．スケジューリングに関して，プロセス生成木を準深さ優先に展開する並列性制御方式について述べ，問題が持つ並列度に応じた動的な粒度変更が可能な条件付関数活性化方式について述べる．また，負荷分散に関して，プログラムの実行過程を考慮してプロセスディスパッチする動的負荷分散方式を提案する．

Abstract: In this paper, we propose dynamic scheduling scheme and load balancing mechanism on Datarol machine, which is an optimized dataflow machine. In the scheduling scheme, we control function level parallelism according to the availability of hardware resources, and instruction level parallelism in accordance with the load of processor. In the load balancing, a processor dynamically determines whether or not to dispatch processes according to the minimum load information transmitted by network. We evaluate the proposed scheme by software simulation.

1 はじめに

マルチスレッド並列実行を基礎とした並列計算機Datarolマシンは，関数型プログラムの実行時に生じる小粒度の多数のプロセスを並行に実行し，超多重並列処理環境を実現する[Ama88][Ama90]．Datarolプロセッサは，従来のデータ駆動型計算機の問題点を解決する1つの手段として，関数適用時に生成されるインスタンス毎に，レジスタファイルを作業領域として割付け，同一インスタンス内のデータをレジスタファイルで共有することにより，冗長なデータフローを削減する．

Datarolマシンはデータ駆動に基づくマルチスレッド並列実行方式を基礎としているので，プログラムに内在する並列性をすべて自動的に引き出し，計算機資源を使い切ってそれ以上の計算ができなくなってしまう「資源の枯渇問題」を招く可能性がある．

本稿では，Datarolマシンにおいて，資源(レジスタファイルやトークン・キュー，インスタンス名，関数適用を制御するスタックなど)の枯渇を防ぎ，与えられた資源を最大限に利用する並列展開戦略について述べる．プロセスの並列展開は，実行順序を決める時間的配置(スケジューリング)と，プロセッサに割り付ける空間的配置(負荷分散)の2段階で行われる．時間的配置に関して，対象とする問題に依存しない並列性制御方式のためにプロセッサ負荷の見積りについて検討し(2節)，関数内の並列性を抑え，並列処理の粒度を大きくすることで細粒度並列処理のオーバーヘッドを削減する条件付関数活性化(3節)について述べる．また，空間的配置に関して，プログラムの実行過程を考慮してプロセスディスパッチする動的負荷分散方式を提案する(4節)．

Dynamic load balancing mechanism and multi-level dynamic scheduling scheme on Datarol machine
Takahide HOSHIDE, Shigeru KUSAKABE, Rin-ichiro TANIGUCHI, Makoto AMAMIYA
Department of Information Systems Kyushu University

2 並列性制御方式

Datarolプロセッサでは，資源の枯渇問題に対し，インスタンス生成木を準深さ優先に展開し，レジスタファイルをスワップ管理することで対処している[Hoshi91]．2.1で並列性制御方式の詳細化としてプロセッサ負荷値の見積りに関して検討し，2.2でソフトウェアシミュレータによる評価を行う．

2.1 現方式

Datarolプロセッサでの関数レベルの並列性制御を以下の方式で行う．関数適用時に，プロセッサ負荷が設定した値(以下，閾値と呼ぶ)以上になると，関数適用を遅延する．そして，プロセッサ負荷が閾値より小さくなった時点で，遅延していた関数適用を再開する．関数適用の遅延をスタック(以下，コールスタックと呼ぶ)を用いて制御することにより，インスタンス生成木の準深さ優先展開を実現する[Hoshi91]．

プロセッサ負荷値の見積りとして，アクティブなプロセス数を用いる場合とパイプライン中のトークン・キュー(以下，TQと呼ぶ)の長さを用いる場合の2つがある．DFM-II[Take]やSIGMA-1[Shima]におけるプロセッサ負荷値の見積りはアクティブな

プロセス数を用いている．一方，MDFM[Rugg] や EM-4[Koda] においては，TQ の長さをプロセッサ負荷値として用いている．

Datarol マシンにおいて，プロセッサ負荷値をアクティブなインスタンス数 (以下，Na と呼ぶ) とすると，関数内で実行可能な命令数は実行時に定まるので，FU 稼働率を高く維持できない可能性がある．また，Datarol マシンでの TQ の長さによる並列性制御では，実行インスタンス数を陽に制御できないため，実行インスタンスが増加し，スワップ処理がボトルネックとなる可能性がある．したがって，Na と TQ の長さの 2 つでの並列性制御は，実行インスタンス数の増加を防ぎ，なおかつパイプラインを充足させ，プロセッサ稼働率を向上させると思われる．本節では，ソフトウェアシミュレーションによりプロセッサ負荷値の見積りについて検討する．

2.2 シミュレーション評価

プロセッサ負荷値による関数適用を制御するための条件フラグを，*Na*：Na が閾値を越えていれば 1，そうでなければ 0，*TQ*：TQ の長さが閾値を越えていれば 1，そうでないなら 0，と表す．条件フラグとして *TQ* を用いる場合には，スワップ処理がボトルネックとなる可能性があるので，条件フラグとして *TQ* を用いる場合には，条件フラグとして *SQ*(スワップ処理待ちインスタンス数が閾値を越えていれば 1，そうでなければ 0) も合わせて用いる．比較する方式は，遅延条件を *Na*，*TQ+SQ*，*Na+TQ+SQ* とした 3 方式である (以下，それぞれ *N* 方式，*TS* 方式，*NTS* 方式と呼ぶ)．ここで，+は論理 or を表し，再開条件は各遅延条件の否定である．

シミュレーション条件は，プロセッサ台数を 64，1 プロセッサが持つレジスタファイル数を 32，1 レジスタファイル当たりのスワップ処理時間を 50 クロックとした．また，各方式において，*SQ* の閾値は常に 2，*NTS* 方式においては *TQ* の閾値を 0 とした．ベンチマークプログラムは，不均質なインスタンス生成木を生成する N クイーン全解探索問題を用いた†．

図 1 と図 2 にシミュレーション結果を示す．図 1 において，閾値の変化による影響を調べるために，閾値をパラメータ (x 軸) とした各方式の実行時間を示す．図 2 において，問題規模に対する各方式のコールスタックの消費量を示す．

図 1 より，*N* 方式 は閾値が 10 以下の場合において実行時間が長い．これは，Na を低く抑えることにより，プロセッサ内に実行可能な命令を持つインスタンスが少なくなり，FU 稼働率が低下するのが原因である．*TS* 方式と *NTS* 方式は，*TQ* による制御のため，FU の稼働率を高く保つことが可能であるので，閾値に依らず安定した実行時間を示す．閾値を約 10〜16 に設定すると，3 方式とも安定した実行時間を示す．

図 2 より，*TS* 方式は，*N* 方式と *NTS* 方式と比較して，問題規模が大きくなるにつれて資源消費量の増加率が高くなる．これは，*TS* 方式は *Na* を用いていないため，インスタンス生成木の展開が深さ優先に制御されていないことが原因だと考えられる．*N* 方式と *NTS* 方式は，直接 TQ による制御を行う *TS* 方式と同等の TQ の管理能力を持つ．したがって，*N* 方式と *NTS* 方式は，資源消費量の観点から見ると *TS* 方式より有効であると思われる．

以上，ソフトウェアシミュレータによるシミュレーションにより，*N* 方式と *TS* 方式，*NTS* 方式の 3 方式の有効性を検討した．検討の結果，資源消費量の観点から見て，*N* 方式と *NTS* 方式が有効であると思われる．*N* 方式と *NTS* 方式を比べると，監視する条件フラグの数が少ない *N* 方式の方が優れていると思われる．*N* 方式において閾値を約 10〜16 に設定することにより，*NTS* 方式と同等の FU 稼働率を得ることができることを確認した．

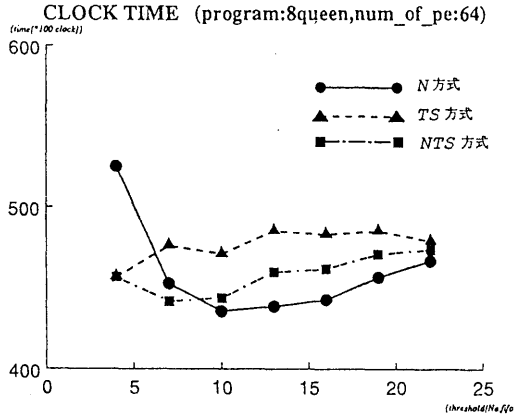


図 1: 閾値の変化に対する実行時間

3 条件付関数活性化方式

Datarol マシンでは，「レジスタファイルにデータを格納することにより直接データ依存関係のない継続命令を指定できる」という特性を利用した柔軟なスケジューリングにより，コンテクスト・スイッチの切替回数を削減し，スワップ処理のオーバーヘッドの削減を行う [Hoshi92]．

以下，3.1 で静的スケジューリング方式について述べ，3.2 で実行時の負荷状況を反映させた，より

†数百〜数千のプロセッサでの大規模な問題実行が望ましいが，WS のパワー不足のため 64 プロセッサでの N クイーンの実行となった

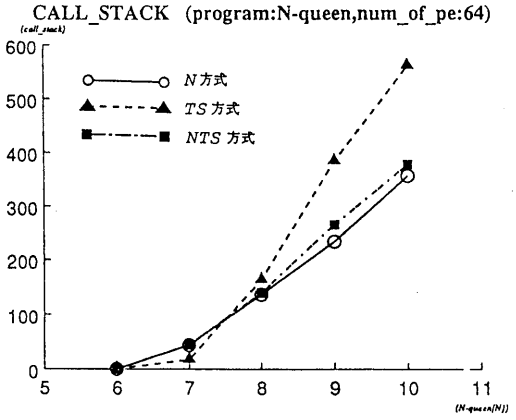


図 2: 問題規模に対するコールスタックの消費量

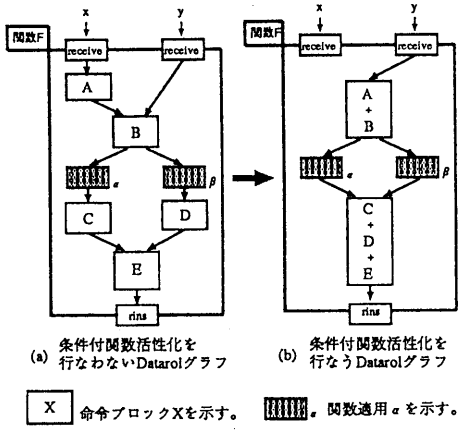


図 3: Datarol グラフの変換

柔軟な制御が可能な動的スケジューリング方式を提案する.

3.1 静的スケジューリング方式

3.1.1 条件付関数活性化

条件付関数活性化は, コンパイル時に明示的な実行の順序付けを行い, 関数内の細粒度並列性を犠牲にして, インスタンスの状態変化 (アクティブ↔サスペンド) 回数を減らすことにより, スワップ処理のオーバーヘッドを削減する. 関数がアクティブになるタイミングは, (1) 引数受取時と (2) 結果値受取時がある. (1) に関しては, 全ての必須引数 (関数の実行に必ず必要となる引数) が到着してから活性化する, (2) に関しては, 他の callee からの結果値が揃ってから活性化する, という条件により活性化を制御する. 以下, 制御を行わない場合を従来方式, 引数受取時の制御を初期活性化制御方式, 結果値受取時の制御を再活性化制御方式と呼ぶ. また, 引数受取時と結果値受取時の両方における制御を条件付活性化制御方式と呼ぶ. 図 3に関数 F の条件付活性化を行う場合と行わない場合の Datarol グラフを示す.

3.1.2 シミュレーション評価

3.1.1で述べた 4 方式を, スワップ処理のオーバーヘッドの影響とプログラムに内在する並列度の影響の 2 点について, ソフトウェアシミュレータによるシミュレーションにより比較評価する.

まず, レジスタファイルのスワップ管理が各方式に与える影響を調べるために, スワップ処理なし (必要なだけレジスタファイルを用意する) とレジスタファイル数が 64,32 の 3 つの場合において各方式を比較する. プロセッサ台数を 32 とし, ベンチマークプログラムは, 図3の変換が可能な 8 クイーン問題の全解探索問題を用いる. 図 4に,「スワップ処理なし」の従来方式の実行時間を 1 としたときの各方式の実行時間を示す. また, 表 1 に, 従来方式に対する各方式のスワップ処理回数削減率を示す.

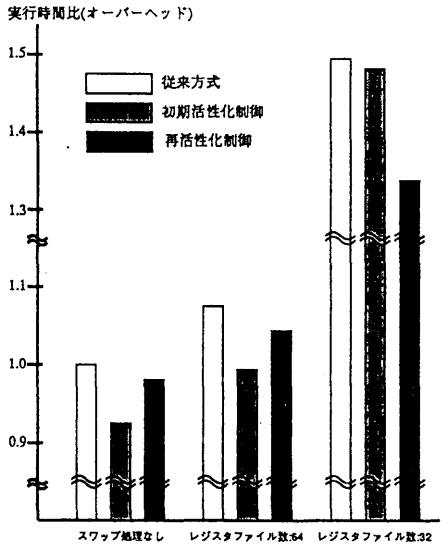
図 4より,「64 レジスタファイル」では, レジスタファイルが十分あり, スワップ処理のオーバーヘッドを隠蔽している. しかし,「32 レジスタファイル」の場合, 実行時に生じるインスタンス数に見合うだけのレジスタファイル数がないため, スワップ処理が実行時間に大きな影響を与えていると思われる. 以下, 図 4と表 1 により初期活性化制御方式と再活性化制御方式について検討する.

図 4より, 初期活性化制御方式は, 従来方式においてスワップ処理のオーバーヘッドが隠蔽できる場合は, 10 パーセント弱の速度向上率を得る. しかし, 従来方式においてスワップ処理のオーバーヘッドが隠蔽出来ない場合は, 速度向上率は小さい. また, 初期活性化制御では, スワップ処理回数の削減はほとんど行われていない (表 1). 以上より, 初期活性化制御による速度向上は, マッチング数の減少が原因だと思われる†. 一方, 再活性化制御方式は, 従来方式でスワップ処理のオーバーヘッドが隠蔽できる場合には速度向上率は小さく, 隠蔽できない場合には速度向上率は大きい (図 4). これは, 再活性化制御方式によりスワップ処理の回数が減少するのが原因だと思われる (表 1). 以上より, 初期活性化制御方式は FU が高稼働を維持できる状況下ではパイプラインの充足率を上げるのに有効で, 再活性化制御方式はスワップ処理のオーバーヘッドの削

†8 クイーンにおいて, 初期活性化制御による静的スケジューリングによりマッチング数の約 47 パーセントが削減される.

表 1: 8 クイーン問題におけるスワップ処理回数削減率 [%]

パターン	初期活性化制御	再活性化制御	条件付活性化制御
64 レジスタファイル	2.2	18.3	9.7
32 レジスタファイル	0.2	20.0	19.2



「スワップ処理なし」の従来方式の実行時間を 1 とした場合の実行時間比を示す.「64 レジスタファイル」と「32 レジスタファイル」における実行時間はスワップ処理によるオーバーヘッドを示す.

図 4: 8 クイーンにおける実行時間比

減に有効である.

次に, プログラムに内在する並列度が各方式に与える影響を調べる. シミュレーション条件は, プロセッサ台数が 4 と 32, レジスタファイル数が 64 とする. ベンチマークとして, N クイーン問題の全解探索問題を用いる. 従来方式に対する速度向上率を表 2 に示す.

表 2 より, 4 プロセッサでの実行時と 32 プロセッサで問題規模が大きい時には, 1 プロセッサが処理するインスタンス数が多いため, 条件付活性化制御により 10〜18 パーセントの速度向上が得られることが分かる. しかし, 1 プロセッサ当たりのインスタンス数が少ない時は, 条件付活性化制御により実行時間が増加している. これは, コード変換によるスケジューリングは静的なため, 実行時の状況が反映されず, 計算機の処理能力内にある並列展開さえも抑制したことが原因である. 以上より, 静的スケジューリングには限界があり, 動的に関数内並列度を制御する動的スケジューリング方式が必要であることが判明した.

3.2 命令ブロックの動的スケジューリング

実行時の状況により関数内並列性を制御するために, プロセッサの稼働状況により命令ブロックの実行遅延を決める動的スケジューリングを提案する. 動的スケジューリングの概念図を図 5 に示す.

3.2.1 抑制アークを用いたスケジューリング

命令ブロックの動的スケジューリングを実現するために, 抑制アークを導入する. 抑制アークを用いた Datarol グラフの概念図を図 6 に示す.

図 6 における関数適用終了時の動作を図 7 に示す. インスタンス状態変化機構 (Instance Activation Controller:IAC) は, 循環パイプライン中のインスタンス制御ユニット内に存在する同期機構で, 同一インスタンス内の複数の callee の同期をとる. IAC 内のフラグは, 他の関数適用が終了しているか否かの情報を示す.

check 命令は, IAC にアクセスする. アクセスした IAC フラグが 0 であれば, 他の関数適用は終了していることを表すので, check 命令は次命令ブロックを叩く. IAC フラグが 1 ならば他の関数適用は終了していないので, check 命令は次命令を叩かない.

表 2: N クイーン問題における速度向上率 [%]

方式	プロセッサ台数	N=5	6	7	8	9
初期活性化制御	4	11.3	15.7	14.5	12.4	12.6
	32	-5.1	-3.5	9.1	8.1	7.5
再活性化制御	4	-7.2	1.9	2.3	2.4	2.9
	32	-5.4	-3.6	0.4	2.6	2.9
条件付活性化制御	4	10.5	13.5	17.6	18.4	18.0
	32	-7.5	-6.2	4.0	9.3	11.0

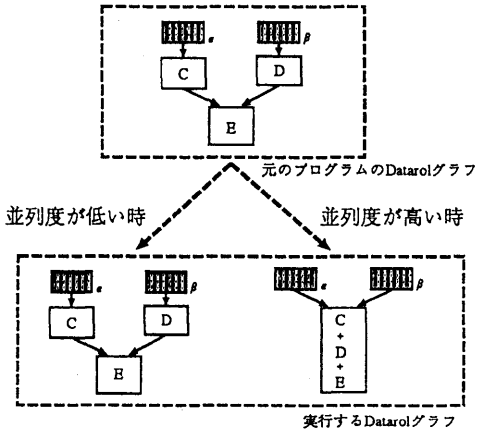


図 5: 動的スケジューリングの概念図

3.2.2 抑制アークの実現方法

図 6 において命令ブロック C の最初の命令 (以下, 命令 C0 と呼ぶ) を叩く命令は関数適用 α とコードブロック D の下の check 命令の 2 つのどちらかである. 命令 C0 の実行のタイミングは, この 2 つの命令のうち最初に命令 C0 を叩いた時である[§]. この動作は, 通常の 2 オペランド命令時の動作の逆である. したがって, 抑制アークの実現は, AC のビットの初期値を 1(通常は 0) にしておき, このビットを用いた疑似的マッチングにより実現可能である. AC ビットの 1 への初期化に関しては, 抑制アークを持つインスタンスがインスタンス名を獲得する (初期活性化) 時に, 該当する命令を疑似的に叩いておくことで解決できる. (しかし, この 1 への初期化によりマッチング失敗によるパイプラインバブルが生じる.)

[§]命令 C0(コードブロック C) の実行可能条件は, 関数適用 α の終了である. check 命令は IAC チェックを行なうので, 関数適用 α が終了していない限りコードブロック C を叩かないことが保証されている.

3.2.3 制御アークの有効性の検討

本動的スケジューリング方式の有効性の検討のため, 動的スケジューリング方式を, スケジューリングを行なわない場合と静的スケジューリングをする場合と比較する.

まず最初に, スケジューリングを行なわない場合と比較する. スケジューリングを行なわない場合には, かなり高い頻度でスワップ処理のオーバーヘッド (数十クロック) が生じる. 一方, 動的スケジューリング方式のオーバーヘッドは非常に小さい (数クロック) と思われる. したがって, スケジューリングを行なわない場合と比べて, 動的スケジューリング方式はプログラムの並列度にかかわらず有効であろう.

次に, 静的スケジューリング方式と比較する. 静的スケジューリング方式では, 実行時のオーバーヘッドがないので, 関数レベルの並列度が高い場合では, 静的スケジューリング方式のほうがスピードアップ率が大きいであろう. これは, 動的スケジューリング方式の check 命令の実行と抑制アークによるマッチング失敗が原因になると思われる. しかし, 関数レベルの並列性が低い場合においては, 動的スケジューリング方式の有効性が発揮されるであろう.

以上より, 本動的スケジューリングは, ハードウェアの変更を全く行なわずに抑制アークを実現できるので, コスト的な観点からも有効であろう.

4 動的負荷分散方式

4.1 現方式と問題点

Datarol マシンでは, 並列性制御方式と両立した動的な負荷分散を行う [Hoshi91]. Datarol マシンでは, リモート・コール命令は無条件にネットワークに出力され, その時点で負荷が一番小さいプロ

セッサに割り付ける戦略を採っていた．しかし，プログラムの実行過程は，以下の 3 つのサイクルからなり，負荷分散の目的は，各サイクルによって異なる．したがって，上記の負荷分散戦略では，ネットワークのトラフィックが増加するという問題と一度プロセッサに割り付けられて遅延されていた関数適用要求のたらい回しが頻発するという問題が生じる．

- **初期サイクル**(初期タスクから数多くのタスクが生成される過程)：全プロセッサにタスクを素早く均一に分散させる．
- **中間サイクル**(生成されるタスクが計算機の処理能力を超えて，遅延される過程)：各プロセッサの稼働率を高く維持する．
- **終盤サイクル**(タスクの生成率が減少し，インスタンス生成木が収束する過程)：各プロセッサの負荷の均等性を保つ．

4.2 動的負荷分散戦略の提案

4.1の問題に対する解決策として，各サイクルの負荷分散の目的を考慮した戦略を採る．Datarol プロセッサは，システム全体で最小負荷であるプロセッサの負荷値をネットワークを通じて知ることができる．この値が大きい時は，リモート・コール命令を自プロセッサで遅延する．そして，ネットワークを通じて得られる負荷 (システム全体での最小負荷) が小さくなると，遅延していた関数適用を，(自プロセッサを含めた) 最小負荷のプロセッサに割り付ける．このような戦略を採ることにより，ネットワークのトラフィックの削減と関数適用要求のたらい回しの削減が可能となる．

この戦略の有効性を検討するためにソフトウェアシミュレータによるシミュレーションを行う．シミュレータには，以下の 3 つのタイプの関数適用命令を用意した．

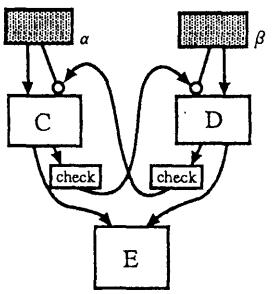
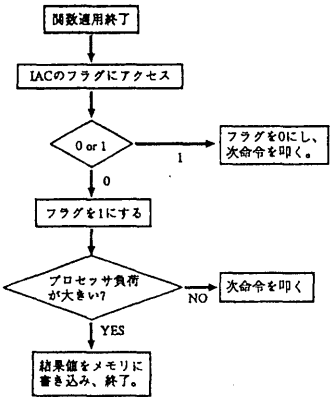


図 6: 抑制アークの導入



IAC のフラグが 0 であれば，もう一方の関数適用は実行中であることを示す．この場合，フラグを反転し，次命令を叩くかどうかを決めるために，プロセッサの負荷を調べる．プロセッサの負荷が大きければ，次の命令ブロックの実行を遅延し，他の関数適用が終了するのを待つ．プロセッサ負荷が小さければ，次命令を叩く．IAC のフラグが 1 であれば，他の関数適用はすでに終了していることを示すので，次命令を叩く．

図 7: 関数適用終了時の動作

- rcall：リモート・コール命令．ネットワークの負荷分散機構を利用し，システムを通じて最小負荷のプロセッサで実行される．
- lcall：ローカル・コール命令．自プロセッサで実行する．
- call：コミュニケーション・ユニットよりシステムにおける最小負荷値を調べ，この値が小さければリモート・コール命令と同じ動作をし，最小負荷が大きければローカル・コール命令と同じ動作を行う．リモート・コールかローカル・コールかの選択は動的に行う．

以下の4つの戦略で負荷分散を行い，Datarolマシンにおける動的負荷分散について検討する．

- 単純戦略1
  インスタンス生成木のすべての関数適用を「rcall命令」で行う．
- 単純戦略2(提案する戦略)
  インスタンス生成木のすべての関数適用を「call命令」で行う．
- 複合戦略1
  インスタンス生成木において，縦型の探索をする関数適用を「lcall命令」で行い，横型の探索をする関数適用を「rcall命令」で行う(コンパイラによる最適化を行う)．
- 複合戦略2(提案する戦略)
  インスタンス生成木において，縦型の探索をする関数適用を「lcall命令」で行い，横型の探索をする関数適用を「call命令」で行う(コンパイラによる最適化を行う)．

4.3 シミュレーション評価

4.2で述べた4つの負荷分散戦略の有効性を検討する．「call命令」のリモート・コールとローカル・コールの選択は，システムにおける最小負荷が7であるときを基準とした¶．ベンチマークプログラムとして，均質なインスタンス生成木をもつフィボナッチ関数と不均質なインスタンス生成木をつくる8クイーンを用いた．

図8と図9に速度向上比を示す．これらの図より，複合戦略2の速度向上比が最も良く，次に複合戦略1の速度向上比が高いことが分かる．複合戦略の速度向上比が良い理由として，コンパイラ(または，プログラマ)による静的負荷分散(関数適用命令の使い分け)を行っていることがあげられる．また，単純戦略1と2を比較すると，格段に単純戦略2の方がよい．これは，プロセッサ間の通信を抑え，自プロセッサのFU稼働率を挙げているためである．図10に，8クイーンを64プロセッサで実行した際の，単純戦略1と2のプロセッサ間通信量を比較する．この図より，単純戦略2は1と比較して，過剰な通信を抑えつつプログラムを実行しているのが分かる．以上より，「call命令」を用いた本動的負荷分散の有効性を確認した．

¶並列性制御の閾値(12)とプロセッサ間通信(プロセッサ台数をNとするとO(logN))を考慮した．

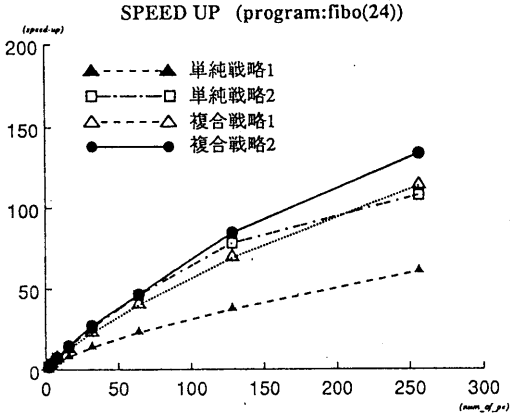


図8: fibo(24)における速度向上比

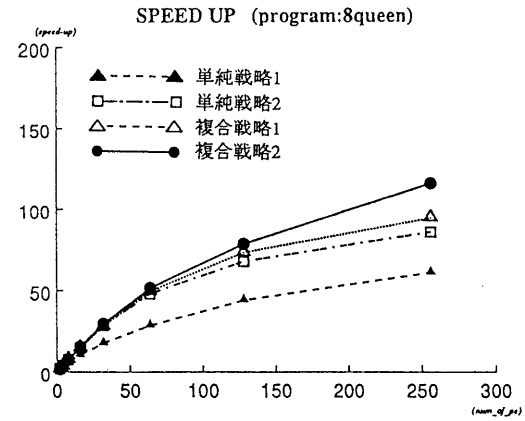


図9: 8クイーンにおける速度向上比

5 まとめ

近年，ノイマン型のパイプラインを持ち込み，スレッドの排他的実行を行うマルチスレッド処

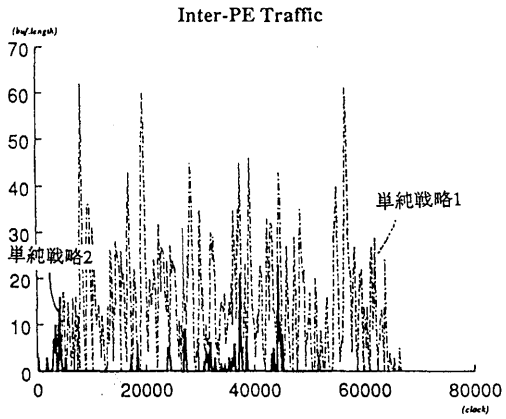


図 10: 単純戦略 1 と 2 の比較 – プロセッサ間通信量 –

理により，従来のデータ駆動型計算機の問題点を解決しようとする研究が盛んに行われている [Datarol-II][EM-4][Epsilon][*T][EM-5]. しかし，これらの計算機おいては，並列性制御に関しての報告が十分になされていない．本稿で提案した並列展開戦略においては，並列性制御によってインスタンス生成木の準深さ優先展開を行い，条件付関数活性化方式によって問題が持つ並列度に応じて動的な粒度変更を行うなど，様々な問題に対して柔軟な対処が可能である．本並列展開戦略とマルチスレッド処理を組み合わせることにより，さらなる実行効率の向上が期待できる．

参考文献

[Ama88] 雨宮真人:"超並列多重処理のためのプロセッサアーキテクチャ", 情報処理学会「コンピュータアーキテクチャ」シンポジウム, pp.99–108, 1988.

[Hoshi91] 星出, 薗田, 谷口, 雨宮: "並列計算機 Datarol マシンにおける資源管理と負荷制御方式", 信学技法, CPSY91-7, pp.25–32, (1991)

[Hoshi92] 星出, 日下部, 谷口, 雨宮: "Datarol マシンの資源管理方式に関する検討 –プロセス状態検出方法と状態変化制御機構–", 情処 45 全大, 2L-9, pp.127–128, (1992)

[Datarol-II] 川野, 星出, 日下部, 谷口, 雨宮: "Datarol アーキテクチャにおけるスレッド実行機構に関する考察", 情処研報, OS92-56, pp.81–88, (1992)

[Shima] 島田, W.Bohm, 平木, 関口: "科学技術計算用データ駆動計算機SIGMA-1 における分散型並列実行制御", 情処 36 全大, 6B-5, pp.109–110, (1988)

[Koda] 児玉, 坂井, 山口: "データ駆動計算機 EM-4 の関数分散方式", 情処研報, ARC85-9, pp.63–70, (1990)

[Ama90] M.Amamiya and R.Taniguchi: "Datarol: A Massively Parallel Architecture for Functional Language", Proc. SPDP, pp.726–735, (1990)

[Kusa] S.Kusakabe, T.Hoshide, R.Taniguchi and M.Amamiya: "Parallelism Control and Storage Management ins Datarol PE", Proc. IFIP world congress, Vol.1, pp.535–541, (1992)

[Take] M.Takesue: "A Unified Resource Management and Execution Control Mechanisum for Data Flow Machines", Proc. 14th ISCA, pp90–97, (1987)

[Rugg] Ruggiero C.A and Sargeant J.: "Control of Parallelism in the Manchester Dataflow Machine", Proc FPCA, (1987)

[EM-4] S.Sakai, Y.Yamaguchi, K.Hiraki, Y.Kodama and T.Yuba: "An Architecture of a Dataflow Single Chip Processor", Proc. 16th ISCA, pp.46–53, (1989)

[EM-5] S.Sakai, Y.Kodama, and Yamaguchi: "Architectural Design of a Parallel Supercomputer EM-5", Proc. JSPP'91, pp.149–156, (1991)

[Epsilon] Grafe V.G. and Hoch J.E: "The Epsilon-2 Miltiprocessor System", Journal of Parallel and Distributed Computing, 10, pp.309–318, (1990)

[*T] Nikhil R.S, Papadopoulos G.M. and Arvind: "*T:A Multithreaded Massively Parallel Architecture", Proc. 19th ISCA, pp.156–167, (1992)