2004－OS－96　(18)
2004／6／18

細粒度マルチスレッド環境での スケジューリングオーバヘッド低減機構

乃村能成[†]　雨宮聡史[††]　日下部茂[†††]
谷口秀夫[†]　雨宮真人[†††]

細粒度マルチスレッド実行マシンのように多数のスレッドが同時に走行する環境では，スレッドスケジューリングがシステム全体の性能に大きな影響を与える．そこで，ハードウェアでスレッドを制御することで，システム全体の性能を確保する手法が提案されている．しかし，この手法では，個々のスレッドの動作，停止を OS から厳密に制御することはできない．本研究では，上記環境において，OS の負担を最小にしつつ，信頼性が確保できるスレッドスケジューリング機構について提案する．

A Process Scheduling Mechanism Suitable for Fine Grain Multithread Architecture

Yoshinari Nomura,[†] Satoshi Amamiya,[††] Shigeru Kusakabe ,[†††]
Hideo Taniguchi [†††] and Makoto Amamiya[†††]

Microprocessors based on the fine grain multithread architecture are characterized by its thread level parallelism. It essentially generates a large number of threads in a second. Therefore, it is very difficult to schedule all the threads effectively by a conventional software process scheduler. For this reason, these microprocessors have a sort of embedded hardware thread control mechanism. However, if operating system does not pay any cost for the scheduling, system would be uncontrollable. In this paper, we propose a cost-effective process scheduling method suitable for the fine grain multithread architecture.

1. はじめに

近年，CMP や SMT といった，1 つのチップ上で複数の命令流を同時に実行するという考え方が一般に広がり[1)]，実際のプロセッサも登場してきた[2)]．これらの多くは，ソフトウェアからは従来の SMP と同じ構成に見えるようになっている．そのため，SMP 対応 OS をそのまま利用することができる．これらの方式では，OS が従来のプロセス/スレッドモデルをそのまま踏襲している．したがって，プロセッサが多数の実行ユニットを効率的に利用しようとして大量のスレッドを生成したい場合，OS のスケジューリングオーバヘッドが問題になってくる．

† 岡山大学工学部
Faculty of Engineering, Okayama University
†† 九州大学大学院システム情報科学府
Graduate School of Information Science and Electrical Engineering, Kyushu University
††† 九州大学大学院システム情報科学研究院
Graduate School of Information Science and Electrical Engineering, Kyushu University

一方，命令レベルの並列性の抽出は，これに限界があるとしてその追及をやめ，並列処理と親和性の高いデータフローモデルを基盤としてスレッドの並列実行のみを追及するという方向性も示されている[3)4)]．我々が開発している Fuce プロセッサは，この考え方に基づいており，複数のスレッド実行ユニットを搭載した，チップマルチプロセッサの一種である．

Fuce におけるスレッドは継続概念[5)] によるイベント駆動によってのみ制御され，排他的に実行され，中断されることがないという特徴を持っている．つまり Fuce では，1 つのプロセスを複数の細粒度のスレッドに分割し，それぞれを並列に実行する．また，細粒度の走り切りスレッド間における同期によって，命令流が実行される．結果として，Fuce においては，細粒度のスレッドが頻繁に生成消滅を繰り返す．

このようなデータフローモデルに基づいたプロセッサアーキテクチャにおいて，従来の OS が行っているスケジューリング，つまり 1 つ 1 つのスレッドについてプライオリティを意識した厳密なスケジューリングを行うことは，以下の点で難しい．

(1) 一旦走り出したスレッドは，止められない．
スレッドは走り切りで，実行中に中断されることがない．そのため，従来のプリエンプションという考え方に基づいて，走行中のスレッドを停止させて実行権を奪うことができない．

(2) 扱うスレッドの数が多い．
OS が全てのスレッドを対象にしてスケジューリングを行おうとすると，扱うべきスレッドの数が多いため，オーバヘッドが大きくなり，現実的ではない．

OS には，以下のことが要求される．

(1) 多数のスレッドが頻繁に生成消滅を繰り返すので，OS がこれらの個々のスレッド全ての走行状態を管理し，スケジューリングを行うことは，できるだけするべきではない．

(2) スレッドの集合体で形成されるプロセスの実行順序は，OS が管理しなければならない．

(3) OS から見て妥当な応答速度で，プロセスを停止させられなければならない．

(1) への対処として，スレッドの生成消滅については，ユーザレベルで実行可能な命令を用いて，アプリケーション内で自由に生成させて，OS は関与しないという方法が考えられる．しかしながら，この対処により，スレッドが OS の管理しないところで自由に生成されるため，それらによって形成されるプロセスの動作を厳密に制御することは難しい．

そこで，本研究では，Fuce プロセッサが多数の短かい実行時間の細粒度スレッドを頻繁に生成し，それらの継続によって処理を実現していることに着目して，「スレッドの継続を制御する同期カウンタの値を増減することによって，スレッド間の同期を意図的に遅延させ，プロセス全体の動作速度を調整する」という手法について検討している．この手法によって，プリエンプションをサポートしておらず，データドリブンなスレッド実行モデルを採用したプロセッサに対して，OS から見て妥当な応答速度で，プロセスを停止させることができるかについて考察する．また，それによって，従来方式と同様な効果を持つタスクスケジューラの仕組みが与えられるかについて考察する．以下，その詳細と課題について説明する．

2. 細粒度マルチスレッドアーキテクチャ Fuce

本章では，まず Fuce プロセッサのアーキテクチャについて概観する．詳細については，参考文献 3)4) を参照されたい．

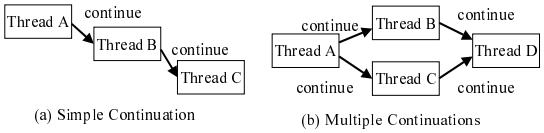

図 1 Concept of continuation.

2.1 Fuce におけるスレッド並列実行モデル

Fuce のスレッド並列実行モデルは，データフロー計算モデルに基づいた継続という概念を核として成り立っている．継続とは，データフロー計算モデルでは 2 つの計算要素間の関係と定義され，Fuce のモデル上では，これはスレッド間の依存関係ということになる．

図 1 に継続の概念を示す．図 1(a) は 3 つのスレッド A,B,C の依存関係を示している．B は A の結果を必要とし，C は B の結果を必要としている．これら 3 つのスレッドを実行するためには，A は計算結果とともに B に対して通知を送り，B は計算結果を C に通知しなければならない．Fuce では，この結果の通知というものを継続と呼び，A は B に継続し，B は C に継続するという．図 1(b) は継続するスレッドが複数の場合を示している．スレッド B と C は，依存関係がないので並列走行が可能であり，スレッド D は B と C から継続されないと走行できない．

あるスレッドに対して継続される元のスレッドの数を fan-in 値，継続する先のスレッドの数を fan-out 値と呼ぶ，スレッド A の fan-out 値は 2 であり，スレッド D の fan-in 値は 2 である．

Fuce のスレッド実行制御は，すべて継続によって決まり，スレッドは継続されるたびに自 fan-in 値をデクリメントして，その値が 0 になったときに，実行可能となり，発火・実行される．そして，そのスレッドは消滅するまでいかなる干渉も受けずに走行することができる．

2.2 Fuce プログラミングモデル

Fuce では，関数インスタンスを中心にしてプログラミングモデルを定義している．一般に関数は，複数のスレッドとしてプログラムされ，その関数の実行環境 (命令列とデータ) として関数インスタンスがある．関数とスレッドの関係を**図 2** に示す． 関数インスタンスの情報は，Activation Control Memory (ACM) に登録される． ACM の構造を**図 3** に示す．ACM は，OS の仮想記憶システムと同様のページ構造となっている． ACM の各ページには，1 つの関数インスタンス情報全てがテーブルに記録されている．記録される情報は，まず関数自体のデータ領域のアドレスであ

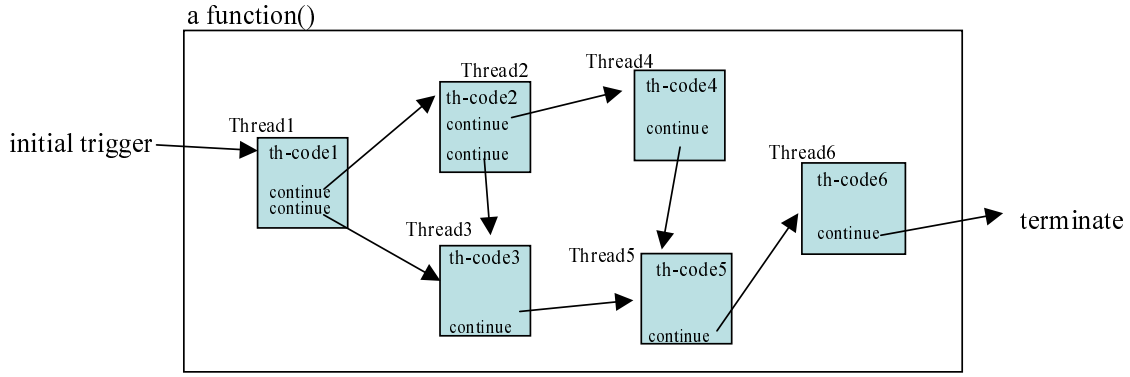

図 2 Function and threads.

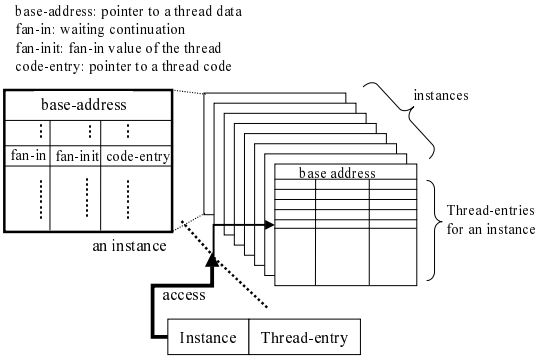

図 3 ACM.

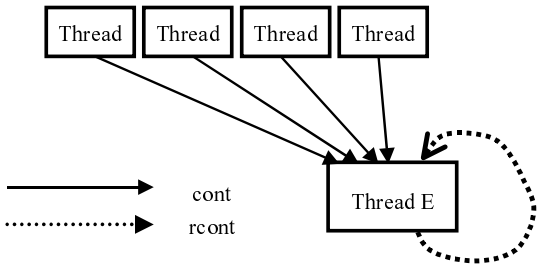

図 4 Mutual exclusion by rcont.

る (図中 base-address)．関数内のスレッド情報は，スレッドの現在の fan-in 値，fan-in 値の初期値 (図中 fan-init) そしてスレッドの先頭アドレス (図中 code-entry) の 3 つの数値がスレッドの数だけ列挙されている．スレッドの ID は，基本的な仮想記憶システムと同様に，ACM のページ番号とページ内オフセットで表す．

このように，Fuce アーキテクチャでは，1 つの関数内に多くのスレッドを生成するプログラミングモデルとなっている．従って，1 つのプログラム中には，多数の同期関係を生成することになる．

Fuce では，継続処理を行う命令に cont と rcont の 2 つがある．以下，それぞれについて説明する．

cont

cont 命令は，引数にスレッド ID を取り，当該スレッドの fan-in 値をデクリメントする．デクリメントされたマイクロスレッドの fan-in 値が 0 になると，対象スレッドは発火し，実行を開始する．通常，cont 命令は fan-in 値が 0 でないスレッドに対して用いられるが，もし fan-in 値が 0 のスレッドに対して発行された場合は，cont 命令がブロックされる．

rcont

rcont は，cont と同じく引数にスレッド ID を取り，当該スレッドの fan-in 値をデクリメントするが，その前に，対象スレッドの fan-in 値を fan-init 値で初期化する．rcont 命令によって，fan-in 値が初期化され，rcont 命令が終了すると，ブロックされている cont 命令は直ちに実行される．このことを利用して，rcont によって排他制御を実現することができる．**図 4** に，rcont による排他制御の様子を示す． Thread E は rcont 命令によって，自分自身に継続しており，それ以外のスレッドは，cont 命令によって Thread E に継続している．Thread E が rcont 命令を実行し終わるまで，他のスレッドの継続は待たされる．Thread E の fan-in 値の初期値が 2 の場合は，他のスレッドの継続は逐次的に実行されるため，Thread E は，同

時に 1 つしか実行されない．

なお，スレッドの実行順序は，同期関係が解決したものから順にハードウェアによってキューに投入され，高速に処理される．

3. 2 レベルスレッドの導入

2 章で示したように，Fuce アーキテクチャにおけるスレッドは，以下のような特徴を持っている．

(1) 短時間に大量の細粒度スレッドが生成消滅を繰り返す．

(2) スレッドの実行順序は，同期関係が解決したものから順にハードウェアによってキューに投入され，高速に処理される．

(2) の特徴により，大量のスレッドが高速に実行可能である．このアーキテクチャ上で動作する OS を考えた場合，スレッドの集合体で形成されるプロセスの実行順序を OS が管理できないということになる．本来 OS の仕事の大きな部分を占めるスケジューリングができないということである．スレッドの生成については，ユーザレベルで実行可能な命令を用いて，アプリケーション内で自由に行うことができる．そこで，スレッドを本来 OS が意識できるスレッドと，Fuce アーキテクチャ上で実現されている細粒度スレッドを区別し，2 レベルのスレッドを定義する．OS 側から管理する最小単位であるスレッドは，Fuce における細粒度スレッドの群であるとする．すなわち，図 3 における ACM テーブルの 1 ページ，つまりプログラミングモデルにおける 1 関数インスタンスに対応する．

すなわち，細粒度スレッドの群がスレッドを形成し，そのスレッドがスケジュールの単位となる．これらのスレッドが複数集まって，プロセスを形成する．**図 5** は，Fuce におけるスレッドと，OS から見たスレッド，プロセスとの関係を表したものである． 図中の microThread が，Fuce チップ上でハードウェアによって実行管理されているスレッド，Thread1 は，OS から管理するスレッドの粒度である．Thread1 内の microThread 群は，Fuce アーキテクチャの ACM 1 ページが管理する内容に対応させる．そうすることで，例えば，プログラムを C 言語で記述したときの関数 1 つ が Thread に対応し，そのインスタンスは ACM テーブルの 1 ページに対応する．OS は ACM のページ単位での実行管理を受け持つことになる．

以降，Fuce 上の細粒度スレッドをマイクロスレッドと呼び，マイクロスレッド群が形成するスレッドグループを OS から見た従来のスレッドと表現し，OS は，これの実行管理を行う．これによって，OS から見た実行管理の粒度が，従来の OS におけるスレッド管理の粒度と同程度にすることができた．以降，これらのスレッドの実行管理手法について議論する．

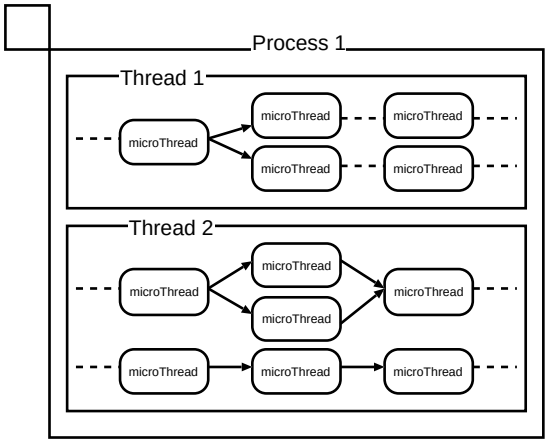

図 5 Micro thread,thread,process.

4. 同期管理表に基づくスレッド実行管理

上記の 2 レベルスレッドモデルの考え方に基づいて，OS から見たスレッドの実行管理をするための方法を考える．

問題にしたいのは，データフローモデルに基づく Fuce において，スレッドは走り切りで，実行中に中断されることがないという点である．つまり，従来のプリエンプションという考え方に基づいて，走行中のスレッドを停止させて実行権を奪うことができない．そこで，本章では，OS から見たスレッドを如何に即座ではないにしろ，うまく停止されられるかどうかについて考察する．

4.1 基 本 機 構

ここでは，OS から見たスレッドを停止させるためには，それに対応する ACM 上の 1 インスタンス，つまり 1 ページ上にエントリされているマイクロスレッドの動作を制御することである．マイクロスレッドは，継続概念によって fan-in 値が 0 になったときに動作する．また，一旦 fan-in 値が 0 になって動作を開始したマイクロスレッドの動作を止めることはできない．したがって，厳密にいうと，ハードウェアスレッド実行キューに入っているマイクロスレッドを即座に停止することはできない．しかしながら，マイクロスレッドは，ごく短かいコードを走り切るという性質を持っており，そのようにコーディングされることが前提とされているため，既に実行を開始した継続動作点で次

のマイクロスレッドが動作しないようにすることならできる．それによって，十分実用的な応答速度で，プロセスの制御ができると見込める．

これらを実現するためには，ある ACM ページ上に存在する全てのマイクロスレッドに対して fan-in 値を意図的に増加させて，同期を意図的に遅らせることである．

例えば，プロセス A, B, C 3 つのプロセスがある．これらの A,B,C のプロセスについて，OS で実行優先度に従って，プロセスの実行速度を調整する．A,B,C のそれぞれが 1 つのスレッドで構成され，それらのスレッドは，それぞれマイクロスレッド A1〜A100, B1〜B100, C1〜C100 から構成されているとする．個々のマイクロスレッドの走行状態は，ハードウェアで管理されているため，OS カーネルからそれぞれを個々に停止，実行することはできない．できたとしても，非常に手間がかかる．2 レベルスレッドの考え方に基づけば，これらは，A, B, C のプロセス中に，1 つづつのスレッドが走行しているように OS からは捉えることができる．ACM に登録されている個々のマイクロスレッドがどのプロセスに属しているかについては，簡易には，マイクロスレッド ID にプロセス ID を含ませたビットパターンを設定することによって可能である．これらのマイクロスレッド A1〜A100 について，fan-in 値を 1 づつ増やすことによって，プロセス A に属する現在走行中のマイクロスレッドが走り切り，継続動作処理を行なっても，継続マイクロスレッドは動作を開始することができず停止する．この原理を利用して，プロセスを走行/停止させることにする．つまり，プロセスに属するマイクロスレッドの走行状態そのものを細かく制御するのではなく，個々のプロセスに属するマイクロスレッドの同期関係に手を加えマイクロスレッドの発火 (実行開始) を制御することで，個々のプロセス走行速度を制御しようとする方式である．これらの実現には，いくつかの方法が考えられる．

(1) コンパイラによって，アプリケーションをコンパイルする際に，全てのマイクロスレッドの fan-in に，OS から継続される特殊な fan-in を埋め込んでおく方法.

(2) OS が定期的に，動的に ACM エントリ上の fan-in 値を増減させる方法.

以降 (1) の方式を静的制御方式 (2) の方式を動的制御方式とよび，それぞれについて以降で説明する．

4.2 静的制御方式

図 6 に，同期の静的追加による制御の様子を示す．

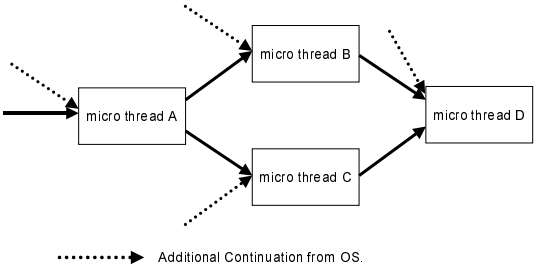

図 6 Thread control by additional continuation.

図中破線で示された継続は，アプリケーションをコンパイルする際に，コンパイラによって挿入された継続である．これらの継続は，OS から定期的に cont 命令が発行されることを意図している．各プロセスの優先順位に基づいて，それに属するマイクロスレッドに発行する cont 命令を増減させることで，プライオリティ制御を行うことができる．この手法は，以下の利点を持つ.

(1) ハードウェアに一切手を加えることなく，コンパイラの変更だけで実現可能である．また，コンパイラが命令列を解析することによって，適切な位置，例えばスレッド継続分岐の手前や，集約の直後などの，効果的に実行を制御可能な点に最低限の継続を挿入することが期待できる.

(2) OS が同期を発行しない限り，各スレッドは実行を開始しないので，OS 主導で，かなり正確なプライオリティ制御を行うことができる.

しかしながら，この手法の欠点は，OS が常に cont 命令を発行せねばならないため，OS による cont 命令発行によって OS のプロセッサ占有率が増加し，システム全体のスループットが低下する可能性がある点である．

4.3 動的制御方式

動的制御方式は，図 3 の ACM エントリ中の fan-in 値を OS によって意図的に 1 づつ増やすことによって実現する方式である．この手法は，以下の利点を持つ.

(1) 静的方式では，OS が cont 命令を発行して，継続動作を繰り返さない限り動作しないのに対して，本方式は，OS が何も関与しない間は，マイクロスレッド間の継続動作によってフルスピードで動作する.

(2) ACM エントリ中の全てのマイクロスレッドの fan-in 値を増加させる必要はない．プロセスの中からある確率でいくつかマイクロスレッドを選択し，それらの fan-in 値を増加させることに

よって，各プロセスのプロセッサ占有率ある程度制御することができる．

しかしながら，本方式では，以下の欠点も同時に存在する．

(1) fan-in 値を強制的に増加させる機械語命令を必要とする．

(2) 排他制御のための rcont を正常に動作させるための保証を行なわなければならない．

(3) ACM エントリ上のどのマイクロスレッドについて fan-in 値を増加させたかを OS は記憶しておかなければならない．

特に (1), (2) を実現するためには，ハードウェアの変更を伴う．以下に，(1), (2) を実現するために追加するべき機械語命令について解説する．

inc-fan-in

inc-fan-init 命令は，引数にマイクロスレッドを表すスレッド ID を取り，当該スレッドの fan-in 値をインクリメントする．

inc-fan-init, dec-fan-init

inc-fan-init/dec-fan-init 命令は，引数にマイクロスレッドを表すスレッド ID を取り，当該スレッドの fan-init 値をインクリメント/デクリメントする．

この命令は，(2) を実現するために必要になる．rcont 命令は，fan-init を利用して排他制御を実現しているため，fan-in 値だけを増加させると，rcont が正常に動作せず，排他制御を実現することができなくなる．したがって，通常は inc-fan-in と inc-fan-init が OS によって連続して実行される．

このように，動的制御方式では，プロセッサの機械語命令を追加する必要があるが，これらの命令は，いずれも Fuce プロセッサ上に既に存在する資源 (カウンタ) の値を増減させるだけの命令であるため，命令追加のための回路規模の増加は非常に小さく，実現は容易である．そのため，この方式は十分現実的で，実現可能なものである．

5. おわりに

本稿では，従来の OS が，逐次実行モデルに基づいて行っているスケジューリングをデータフローモデル上で実現するためのメカニズムについて考察した．提案方式では，全てのスレッドの動作を逐一把握して制御する従来の方式ではなく，ある一定の割合で細粒度スレッドの実行を抑制することによって，全体の実行速度を調整しようとするものである．

現在，本方式に基づいて Fuce プロセッサで動作する OS である CEFOS を開発中である[6]．また，残された課題は以下の通りである．

(1) スレッドの集合体であるプロセスそのもののアカウンティングをどうするのか．プロセスをスケジューリングするためのプライオリティの根拠となる走行履歴をどう取るか．

(2) 各プロセス間の公平性を確保できるか．

今後は，CEFOS 上に本方式を実装し，以上の評価を行いたい．

謝辞 本研究は，文部科学省科学研究費補助金・基盤研究 (A)(2) 「細粒度マルチスレッド処理原理による並列分散処理カーネルウェアの研究」(課題番号:15200002) による．

参考文献

1) Lo, J. L., Eggers, S. J., Emer, J. S., Levy, H. M., Stamm, R. L. and Tullsen, D.M., “Converting Thread-Level Parallelism via Simultaneous Multithreading,” ACM Transaction on Computer Systems, Vol.15, No.3, pp.322–354, 1997.

2) Marr, D.T., Binns, F., Hill, D.L., Hinton, G., Koufaty, D.A., Miller, J.A. and Upton, M. “Hyper-threading technology architecture and microarchitecture: a hyperhtext history,” Intel Technology J. 6,1 2002 (online journal).

3) Amamiya, M., Taniguchi, H. and Matsuzaki, T., “An Architecture of Fusing Communication and Execution for Global Distributed Processing,” Parallell Processing Letters, Vol.11, No.1, pp.7–24, 2001.

4) 雨宮聡史, 松崎隆哲, 雨宮真人, “排他実行マルチスレッド実行モデルに基づくオンチップ・マルチプロセッサの設計,” 情報処理学会研究報告, Vol.2003, No.119, pp. 51–56, 2003.

5) Amamiya, M., “A New Parallel Graph Reduction Model and its Machine Architecture,” Data Flow Computing: Theory and Practice, Ablex Publishing Corporation, pp.445–467, 1991.

6) 日下部茂, 富安洋史, 村上和彰, 谷口秀夫, 雨宮真人, “並列分散オペレーティングシステム CEFOS (Communication-Execution Fusion OS),” 信学技報，Vol.99, No.251, pp.25–32 1999.