計算機アーキテクチャ　68－8
(1987・11・12)

階層化モデルに基づく汎用離散系シミュレーション用並列処理アーキテクチャ

AN ARCHITECTURE FOR PARALLEL DISCRETE EVENT SIMULATION

工藤知宏　天野英晴
Tomohiro KUDOH　Hideharu AMANO
慶応義塾大学理工学部電気工学科
Department of Electrical Engineering, Faculty of Science and Technology, Keio University

あらまし　離散事象モデルのシミュレーションを並列処理するための専用システムＥＯＳのアルゴリズム及びアーキテクチャについて述べる。

ＥＯＳでは問題の記述モデルとして待ち行列網をフローグラフを用いて表現する手法を採る。さらに問題の性質を利用してモジュール化したモデルを構築し、時刻管理、事象管理も分散する。

このアルゴリズムでは分散時刻管理において問題となる、事象の処理を行なってよいかどうかの検証をおこなうオーバヘッドを避けるために、履歴を残しつつ処理を進め、後で不当な処理については取り消すという方法を用いている。

またＥＯＳにおける問題の記述についても簡単に触れる。ここでは、モジュール化とオーバヘッドの軽減のためにユーザが積極的に情報を与える方法を用いている。

Abstract: Parallel processing algorithm and architecture to process discrete event simulation are proposed in this report.
We adopt the flow graph model to describe target problems. By dividing the flow graph into several modules, virtual time controls and event controls are distributed. To enhance the performance and to guarantee consistency, an anticipation mechanism with history recording is used. If anticipations are illegal, anticipation processes are canceled using the records.

1．はじめに

計算機に対する需要の中で離散系シミュレーションの占める割合は大きい。現在離散系シミュレーションはGPSS、Simscript、Simulaなどの言語で記述され大型汎用計算機上で解かれるのが一般的であるが、離散系シミュレーションは非常に計算機コストを消費するためこれまで大きな問題を解くことは難しかった。離散系シミュレーションを高速に行なうことができれば、これまで扱えなかったような大きな問題をも扱うことができるようになり、その意義は大きい。

一方離散系シミュレーションは並列性を有する典型的な問題である。その代表的な例としては、論理回路、通信網、工場の生産ライン、交通、計算機等のシステムなどのシミュレーションが挙げられる。これらの系では系の構成要素は並列に動作している。そこでこの問題固有の並列性を利用して並列処理を行なうことにより処理速度を向上させることが考えられる。

これまで、科学技術計算や推論などを対象とした並列処理は盛んに研究され、すでに実用機も作られているが、離散系シミュレーションの並列処理についてはいくつかの基礎的な研究は報告されているものの、論理回路専用のシミュ

レータを除けば実用的なアーキテクチャは提案されていない。これは、主として離散系シミュレーションには、時刻管理等の並列処理に際し問題となるような点があるためである。

1．1．事象駆動と時刻駆動

単一プロセッサ上で離散系シミュレーションを行なうもっとも一般的なモデルは、シミュレーション仮想時刻を持つ時計と生起時刻順に並べられた事象の待ち行列（イベントキュー）があるもので、時計の示す時刻の事象の処理がすべて終了すると待ち行列の先頭にある事象（最も近い時刻に起こる事象）の生起時刻まで時計が進められその時刻に生起する事象の処理が行なわれるというものである。このような手法を事象駆動と呼ぶ。これに対し、シミュレーションの対象となる系が一定周期で同期して変化し同一周期に状態が変化する処理体の数が多いときにはシミュレーション仮想時刻を一定間隔づつ進めその都度全処理体について処理を行なう手法が用いられる。これを時刻駆動（タイムスライス法）と呼ぶ。単一プロセッサ上で処理を行なう場合には、事象駆動によるか時刻駆動によるかは問題の性質により効率のよい方をとるのが一般的である。

複数プロセッサ上で処理を行なう場合には、事象駆動と時刻駆動では取り扱い易さが違ってくる。事象駆動ではイベントキュー及び時計の管理が複雑であるため、効率よく高い並列性を得ることが難しい。これに対し時刻駆動ではイベントキューを持つ必要がなく並列処理は比較的容易であるが、時刻管理の融通性が低くまた、各時刻に状態が変化する処理体の数が少なければ無駄が大きくなり効率が上がらない。即ち単一プロセッサ上でも時刻駆動を用いるような問題に対しては有効であるが、一般の問題には適していないといえる。

ＥＯＳでは事象駆動を用いて並列処理をおこなう。

1．2．時刻管理

事象駆動では特に時刻管理の手法が問題である。時刻管理の手法としては大きく分けて、集中時刻管理と分散時刻管理がある。集中時刻管理は処理系全体で一個の時計のみを持つもので、離散系シミュレーションにおける時刻管理の方法としては最も基本的なものである。この方法は、実現は容易であるが時刻管理が律速となる。時刻管理が一カ所で行なわれる場合、イベントキューを分散すると時刻更新の管理が困難になるので通常イベントキューも一カ所に置く。そうするとプロセッサ数をいくら増やしても時計とイベントキューの管理のオーバヘッドが大きく効率が上がらない。また、イベントキューが一つだと同一時刻のイベントしか並列処理ができず、一般の問題では並列処理をしても効果が小さい。一方、分散時刻管理では、独立な事象は同一時刻のものでなくとも並列に処理することが出来、時刻管理が律速になることもない。しかし、独立な事象であるかどうかを判別することが難しい。

既存の並列離散系シミュレーションのためのアーキテクチャは論理回路等の専用目的のものがほとんどで、集中時刻管理を用いている。これらのシステムはキューを管理する部分と数個の高速な処理装置からなり処理装置の個数はあまり多くない。これは処理装置の数を多くしても集中時刻管理では並列に処理できるのは同一時刻に生起する事象のみであること、時刻管理と各処理装置の処理能力のバランスから、処理装置の数だけ増やしても時刻管理が律速になるという理由からである。

同一時刻に生起する事象以外のものも並列に処理しようとすれば分散時刻管理を用いることになる。分散時刻管理ではＡという事象を処理するときに他の部分ではまだＡよりも前の時刻の事象Ｂを処理していないこともあり得、ＡがＢに影響を受けることがないことが保証されている場合のみそのＡの処理を行なうことができる。この保証のためのアルゴリズムは既に提案され、その正当性が証明されているが［１］高速に処理することは困難である。

ところで離散系シミュレーションの対象となるような問題には多くの場合モジュール性、階

層性が存在する。例えば交通のシミュレーションでは町ごとにモジュールとすることができるしその町も駅、道路といった単位に分割できる。計算機システムのシミュレーションではプロセッサ、入出力装置といった構成要素ごとにモジュール化できるしこれらの構成要素もさらに分割して扱えることが多い。

そこでＥＯＳではこの問題の性質を利用して階層的に時刻を管理する集中時刻管理と分散時刻管理を組み合わせた手法を用いる。

１．３．フローグラフ

シミュレーションの系を示すためにＥＯＳではフローグラフに基づくモデルを用いる。このモデルでは系全体はフローグラフと呼ばれる有向グラフを用いて表される。（図１）このグラフは現実の待ち行列網システムにおける能動的要素である窓口、交換機等をノードにより表現し、客の通り道や通信回線をノード間を結合するリンクとして表現する。また、受動的要素である客や通信の呼はグラフ内を流れるトークンで表現する。トークンにはシミュレーション仮想時刻により時刻印が押される。時間消費はノード内のみで行われる。

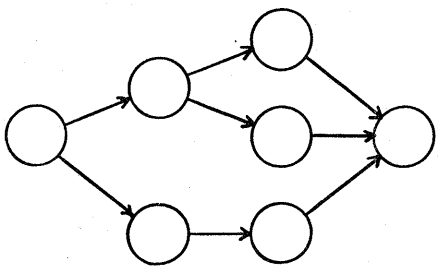

図１　フローグラフ

２．処理モデルとアルゴリズム

ＥＯＳではフローグラフは問題の性質に従いいくつかのモジュールに分割され、さらに各々のモジュール内も分割してモジュールを構成することにより多重に階層化できる。この分割階層化された系はモジュール間の包含関係にしたがって木構造を構成する。この木の根は系全体を表す。木の葉はフローグラフの最小構成要素であるノードを１つ以上含んでいる。（図２）

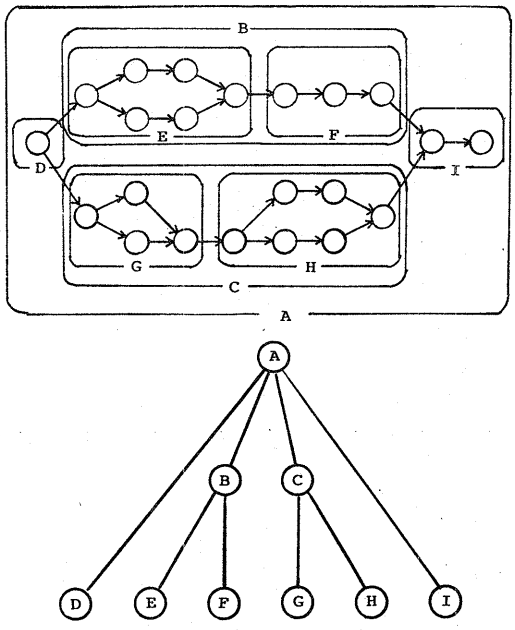

図２　フローグラフの階層、モジュール化

２．１．処理系とアルゴリズムの概要

ＥＯＳの処理系も処理モデルに対応して木構造をとる。ここで各葉を担当する処理プロセッサと各節を担当する同期プロセッサ、根を担当するルートプロセッサを想定する。（図３）何れのプロセッサもシミュレーション仮想時刻を示す時計ＶＴを持つ。ルートプロセッサのＶＴは系全体の基準となり、ＧＶＴと呼ばれる。葉を担当する処理プロセッサは、ＶＴとともにイベントキューＥＱを持っている。詳しいアーキテクチャについては後述する。

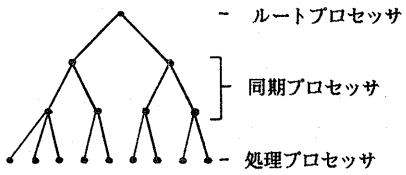

図３　ＥＯＳの処理系

事象の処理は処理プロセッサによって行なわれる。処理プロセッサはまずＧＶＴ時刻の事象を処理する。これは一般の集中時刻管理においても行なわれることで、この処理は安全である。ＧＶＴ時刻の処理をすべて終えると、処理プロ

セッサは、各々のＥＱの内容を各々のＶＴを更新しながら処理する。これは外部から更新後のＶＴ時刻よりも古い時刻印を持ったトークンが到着する可能性があるので、安全ではない。この処理を先取り処理と呼ぶ。より古い時刻印を持ったトークンが到着した場合、先取り処理は非合法であったという。先取り処理時には履歴が残され、処理が非合法であった場合には履歴により非合法な処理以前の状態に戻る。これを回復という。

処理プロセッサ外とのトークンのやりとりは親の同期プロセッサを通して行なわれる。ＧＶＴはすべての処理プロセッサのＶＴの最小値に更新できるので、同期プロセッサは子プロセッサのＶＴの最小値を親のプロセッサに伝える。

ＧＶＴが更新されればそれ以前の履歴は不用となる。そこでＧＶＴが更新されると新しいＧＶＴの値が全プロセッサにブロードキャストされる。

プロセッサ間の情報のやり取りはすべてメッセージにより行われる。トークンもメッセージとして受け渡され、これをトークンメッセージという。

２．２．処理プロセッサのアルゴリズム

処理プロセッサは事象の処理を行う。もし、自身の担当しないノードと接続しているリンクへのトークンの出力がある場合にはトークンメッセージが親の同期プロセッサに送られる。

シミュレーションの開始時にはＶＴはＧＶＴと同じ時刻を示している。処理プロセッサはこの後次のような処理を行う。

（１）まずＶＴ時刻の事象をＥＱから取り出し処理を行う。これは安全な処理である。親からトークンメッセージを受け取ったらＥＱに登録する。

（２）ＶＴ時刻のイベントを全て処理し終えると、ＥＱに事象があれば先取り処理を行う。この際ＶＴを更新するごとに、新しいＶＴの値を親である同期プロセッサに送る。ＥＱが空になればその旨のメッセージを親である同期プロセッサに送る。ここで、

・処理プロセッサは全ての事象の履歴を保存している。

・親のプロセッサからトークンメッセージを受け取り、そのメッセージの時刻印が現在のＶＴよりも小さければ、ＶＴをその時刻印の時刻まで回復し、履歴により自身の担当するノード、トークンの状態をその時刻以前の状態まで回復し、ＶＴを回復する旨のメッセージを親プロセッサに送る。

・親から受け取ったトークンメッセージの時刻印がＶＴよりも大きければＥＱに登録する。

・既に受け取ったトークンメッセージを取り消すというメッセージを親から受け取った場合には、取り消すメッセージがＥＱ内にまだ登録されていればそれを取り消し、既に処理されていれば処理以前の状態まで回復する。

（３）ルートプロセッサから新しいＧＶＴの値を受け取ると、ＧＶＴ時刻以前の履歴を消去する。また、ＶＴがＧＶＴよりも小さければＶＴの値をＧＶＴと等しくする。

（４）ＶＴがＧＶＴと等しければ（１）に、そうでなければ（２）にもどる。

２．３．同期プロセッサのアルゴリズム

同期プロセッサは処理プロセッサ間のトークンメッセージのやり取りと、時刻の管理を行う。具体的には以下の処理を随時行う。

（１）子プロセッサからのトークンメッセージは受け手が自分の子である場合には、その子に送る。そうでなければ、さらに親のプロセッサに送る。この時メッセージの履歴は保存する。

（２）親プロセッサからメッセージを受け取った場合には、宛先のノードを担当する子プロセッサにそのメッセージを送る。

（３）トークンメッセージを送った子プロセッサからＶＴを回復する旨のメッセージを受け取った場合には、履歴を検索し、その子プロセッサが非合法な処理中に出した

トークンメッセージがあれば、宛先のプロセッサにそのトークンメッセージを取り消すという内容を持ったメッセージを伝達する。また、回復した後のＶＴがそのプロセッサ自身のＶＴよりも小さければ、自分自身のＶＴを回復し、その旨を親のプロセッサに伝える。

（４）同期プロセッサは子プロセッサのＶＴの値を常に知っており、ＥＱが空でない全ての子のＶＴの中で最小のものを自身のＶＴとする。全ての子プロセッサのイベントキューが空の場合には、その旨を親プロセッサに知らせる。子プロセッサのＶＴの変化により自身のＶＴが変化した場合には、ＶＴの値を自身の親であるプロセッサに知らせる。

（５）ルートプロセッサがＧＶＴをブロードキャストしたときには、ＧＶＴ以前の履歴を消去し、もしその値がＶＴよりも大きければ、ＶＴをＧＶＴと等しくする。

２．４．ルートプロセッサのアルゴリズム

ルートプロセッサはＧＶＴの管理と子プロセッサ間のトークンメッセージの伝達を行う。

（１）ルートプロセッサは子プロセッサのＶＴの値を常に知っており、全てのＥＱが空でない子プロセッサのＶＴの中で最小のものを自身のＧＶＴとする。全ての子のプロセッサのＥＱが空ならば、そのシミュレーションは終了である。子のプロセッサのＶＴの変化により自身のＧＶＴが変化した場合にはその値を全てのプロセッサにブロードキャストする。

（２）子プロセッサからのトークンメッセージは受け手を担当する子プロセッサに送る。

（３）同期プロセッサと同様にトークンメッセージの履歴を持ち、子プロセッサからＶＴの回復を知らされた場合には、履歴を検索し、子プロセッサが非合法な処理中に出したトークンメッセージがあればそれを取り消すメッセージを宛先の子プロセッサに送る。但し回復したＶＴがＧＶＴよりも小さいことは有り得ない。

３　アーキテクチャ

３．１．設計方針

２章に述べたアルゴリズムは問題によってはかなり大きな並列性を持っている。しかし、ある種の科学技術計算とは異なり、その並列性は平均的には数十から数百のオーダであることが予測される。このため、ＥＯＳのアーキテクチャは以下のような方針で設計される。

（１）比較的少数（数十台）のプロセッサを用いる。

（２）各プロセッサは処理機能毎に専用化し、機能の一部をハードウェア化する。また、プロセッサ内の処理における細かいレベルの並列性もできる限り利用する。

３．２．全体構成

ＥＯＳアーキテクチャの全体構成は図３に従う。ＥＯＳアーキテクチャは多分岐ツリー構造を持つ。ツリーの根をルートプロセッサといい、中間節を同期プロセッサ、葉は処理プロセッサという。原則としてプロセッサ間のメッセージ交換はツリーのリンクによって行われるが、根から全プロセッサに対してのグローバルバスが存在し、これが唯一の大域交信路である。

ツリーの最小システムは２階層であり、同期プロセッサを持たない。プロセッサ数が多い場合は、同期プロセッサを増やすことで、階層のレベルを大きくする事ができる。各プロセッサは２章で既に述べた処理を高速に実行するため以下のような構造を持つ。

（１）処理プロセッサ

処理プロセッサはフローグラフに沿って、シミュレーションを実行するのが主な役目であるが、その全体の処理量のかなりの部分はＥＱ、ＶＴの管理が占める。このことに対応するため、処理プロセッサは２つの独立した処理単位（図４）から構成され、これらの分散処理により高速化を図る。これらは以下の機能を持つ。

・システム管理エンジン

ＥＱ，ＶＴの管理、同期プロセッサに対するメッセージの授受の制御を行う。「回復」の際の履歴の検索等は非常な高速処理が要求されるため、このエンジンはマイクロプログラム制御にによる高速専用処理を行う。

・ノード実行エンジン

ユーザにより記述された実際のプログラムを実行する。コンパイラの作成を容易にするため汎用マイクロプロセッサを用いる。

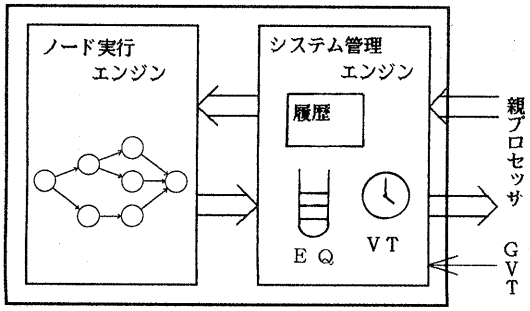

図４　処理プロセッサ

（２）同期プロセッサ

同期プロセッサの構造を図５に示す。同期プロセッサの主な役目はメッセージのルーチング、メッセージの履歴の管理と検索、子プロセッサのＶＴの収集、管理である。このため同期プロセッサはメッセージルータ、メッセージ履歴管理、ＶＴ管理の３つの部分から構成される。これらの操作は専用化され高速性が要求されるため、同期プロセッサは、処理プロセッサのシステム管理エンジン同様マイクロプログラム制御の専用プロセッサである。

子または親プロセッサからメッセージが到着すると、まずメッセージルータによって受け付けられ、シミュレーションのプログラム上で用いられるトークンメッセージか、ＶＴ更新のメッセージかチェックされる。ＶＴ更新のメッセージであった場合、子のＶＴの値は保存され、他の子のＶＴの値と比較されて必要に応じて自分自身のＶＴの値の更新が行われる。そうでなければ、メッセージは宛先にしたがってルーチングされるとともに履歴が保存される。「回復」の際はこの履歴が高速に検索される。

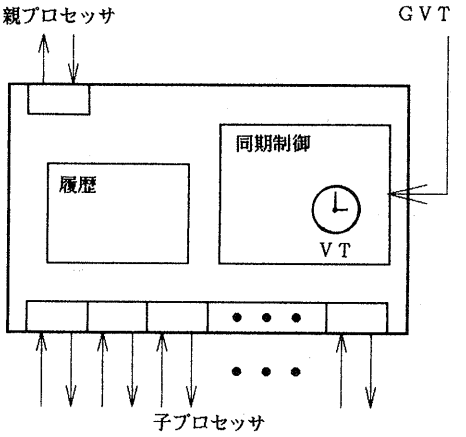

図５　同期プロセッサ

（３）ルートプロセッサ

構造はほぼ同期プロセッサと同様であるが、グローバルバスにＧＶＴをブロードキャストする機能を持つ。

これらのプロセッサの数の設定と、具体的な構造を定めるためにはこのアーキテクチャ自身のシミュレーションを行う必要があり、現在評価検討中である。

４．問題の記述

４．１．従来の言語の問題点

ＥＯＳの時刻管理アルゴリズムはＧＰＳＳを始めとするイベント駆動型の離散系シミュレーション用言語全てに適用することができる。しかし、現在用いられている言語はＥＯＳにおいて問題の記述に用いるには次のような問題点がある。

（１）GPSSはモジュール性、階層性が弱いため、ユーザの書いたプログラムを自動的にモジュール化し、プロセッサにマッピングする必要がある。この際、うまくモジュール化できないと、回復のための作業が頻

繁になり効率が低下する。

（２）Simula, Mayはモジュール性を持つが、言語自体柔軟性が大きいため実装上の問題が多く複数のプロセッサでの高速実行は困難である。またフローモデルとの適合性にも欠けている。

ＥＯＳにおけるシミュレーションは対象とする実世界に既にモジュール性、階層性がある場合に効率が高い。そこで、我々は、やや記述における柔軟性を犠牲にしても実装が容易であり、かつＥＯＳに適した問題に対しては十分な記述力を持つ言語ＨＮＣＣ（Hierarchical Node Oriented Concurrent C）を提案する。プログラマがＨＮＣＣを用いて記述すると、自然にモジュール化、階層化が行われる。このため、このモジュール、階層の単位でプロセッサに対してマッピングを行えば、ＥＯＳにおいて効率よく実行されるプログラムを書くことができる。

４．２．ＨＮＣＣ

ＨＮＣＣは、並行プロセス言語ＮＣＣを階層化し、離散系シミュレーション用の機能と関数を加えたものである。ＮＣＣは静的なノードとネットワークによって系を表現する。ノードはポートを介して他のノードとの交信を行うが、各ノードは自分のポート名のみを知っていればよく、ポート間の結合は独立に記述できる点がこの言語の大きな特徴である。また、交信は１対複数の転送即ちマルチキャストで行われる。完全な同期通信（ランデヴ）はマルチキャストの効率を悪化させるためＮＣＣにおいては受信ポートにサイズ１のバッファを設ける。ＮＣＣの詳細は［２］を参照されたい。ここでは並列計算機のシミュレーションプログラムを例に簡単にＨＮＣＣ特有の機能に関して解説する。

図６にシミュレーションを行う対象の並列計算機システムを示す。このシステムはバスによりプロセッサと共有メモリが結合された非常に簡単な構造を持つ。各プロセッサは共有メモリに対し、PREADの確率で読出し要求,PWRITEの確率で書き込み要求を出し、それ以外の場合は共有メモリをアクセスしない。

このシステムに関してＨＮＣＣで記述したシ

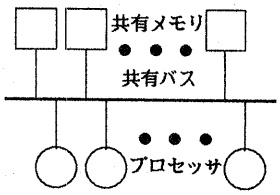

図６　並列計算機システムの例

```
nodeatom PU {
  inport datain; outport request;
  program pu(); }

nodeatom MEM {
  inport access; outport dataout;
  program p_memory(); }

nodeatom BUS {
  inport puin[SIZE],memoryin[MSIZE];
  outport puout[SIZE],memoryout[MSIZE];
  program p_bus(); }

nodeset{
  node{
    PU  pu[i:SIZE](i);
    MEM memory[i:MSIZE](i);
    BUS bus; }
  connect{ /* connect nodeatoms  */
    int i;
    for(i=0;i<SIZE;i++){
        con(pu[i].request, bus.puin[i]);
        con(bus.puout[i], pu[i].datain); }
    for(i=0;i<MSIZE;i++){
        con(memory[i].dataout, bus.memoryin[i]);
        con(bus.memoryout[i], memory[i].access); }
  }
} main();

program{
pu(){
  double op; int m_num;
  while(currenttime()<ENDTIME) {
    op=rand()/MAXNUM;
    if(op<PREAD) {
      m_num=rand()%MSIZE;
      send(request,m_num); send(request,READ);
      receive(datain); }
    else if(op<PREAD+PWRITE){
          m_num=rand()%MSIZE;
          send(request,m_num);
          send(request,WRITE);
          receive(datain); }
         else wait(ACCESS_TIME); }
}

memory(){
        int request;
        while(currenttime()<ENDTIME) {
                receive(access,&request);
                wait(ACCESS_TIME);
                if(request==READ) send(dataout); }
}

bus(){
  int i,m_num,request;
  while(currenttime()<ENDTIME) {
    entry {
      for(i=0;i<SIZE;i++) {
        (puin[i],&m_num):
          receive(puin[i],&request);
          wait(BUSCYCLE);
          send(memoryout[m_num],request);
          if(request==READ) {
            receive(memoryin[m_num]);
            wait(BUSCYCLE); send(puout[i]); }
          else send(puout[i]);
          break; } } }
}
```

図７　ＨＮＣＣによるプログラム例

ミュレーションプログラムを図７に示す。なお、このプログラムはＨＮＣＣの考え方を示すためのものなので、必ずしも正確ではなく、統計情報収集なども含まれていない。

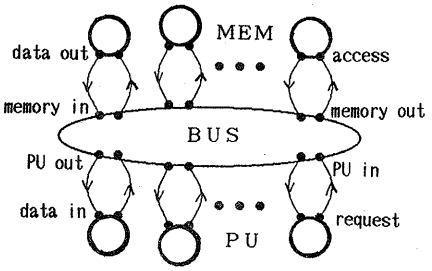

図８　例題のフローグラフ

ＨＮＣＣはＮＣＣ同様、ノードの型定義、ノードの動作定義、ノード間の接続を記述する。この例では３つの基本ノード(nodeatom),PU,MEM,BUSを図８に示すように接続され、全体として複合ノード(nodeset) mainを構成している。離散系シミュレーションに対応するため、次のような関数が利用されている。

・ｗａｉｔ（時間）；

シミュレーション仮想時間を消費する。

・ｃｕｒｒｅｎｔｔｉｍｅ（）；

現在のシミュレーション仮想時刻を返す。

仮想時間の消費はｗａｉｔのみで行われ、そのほかの操作では時間は消費されない。また、先に述べたように受信ポートにはバッファが存在するためメッセージ間の時刻管理は次のように行われる。全てのメッセージには時刻印がおされているものとする。

・ｓｅｎｄ側

相手方の受信ポートにメッセージが存在しなければメッセージをその場で送り、そのまま処理を実行する。

一方、受信ポートにデータが存在する場合は待ち状態になる。ここで、バッファが空いてメッセージが送られた時、もしも受信側の仮想時刻が送信側より進んでいた場合は、その時刻を自分自身の時刻とする。

・ｒｅｃｅｉｖｅ側

既に受信ポートにメッセージが存在する場合もしそのメッセージの時刻が自分自身の時刻より進んでいる場合、その時刻を自分自身の時刻とする。受信ポートにメッセージが存在しない場合は送られてくるまで待って同様の操作を行う。

これらの操作は２章で述べたアルゴリズムで自動的に行われるので、プログラマはこのような管理を気にする必要はない。また、多数のポートで同時にメッセージを待っている場合(BUSにおけるenrey文)、到着するメッセージは仮想時刻順になっていることが保証される。

ＨＮＣＣにおいては複合ノード同士の組合せから更に上位階層の複合ノードを構成することが可能である。一例としては図７のmainに多少手を加え接続する事により、クラスタ構造を持つ並列計算機のシミュレーションを容易に記述することができる。

５．まとめ

ＥＯＳの性能上の問題点としては不当な処理の回復に伴うオーバヘッドが大きくなりすぎる可能性が挙げられる。これに対しては処理プロセッサがトークンメッセージを送らなければならなくなったら先取り処理をやめ、ＧＶＴがＶＴと等しくなるのを待つという方法、ユーザの指定や実行時の統計的に、不当である可能性が高い処理は行わないという方法などが考えられる。

また、履歴が大きくなりすぎる場合には先取り中の一部の時刻の履歴のみを残し、回復するときにはそれらの履歴のある時刻まで戻るようにすることも考えられる。

現在ＥＯＳのアーキテクチャの詳細設計を進めている。各プロセッサの期待できる速度を現在のデバイス技術に照らして求めた後シミュレーションによって効率を確認する予定である。

［参考文献］

［１］吉田隆一、所真理雄：離散系シミュレーションの並列処理、コンピュータソフトウェアＶｏｌ．４、Ｎｏ．１（１９８７）、pp23-33

［２］高橋範朗、朴泰佑、天野英晴、相磯秀夫：並列計算機（ＳＭ）^{2}-IIのための並行プロセス言語ＮＣＣ、３２回情処全大、pp513-514