学位請求論文

並列離散事象シミュレーション環境の研究

1998年12月18日

指導教官　　相田仁 助教授

東京大学 大学院 工学系研究科
電子情報工学専攻 67121

日高宗一郎

目次

表一覧

図一覧

第 1 章

序論

1.1 はじめに

離散事象シミュレーション (Discrete Event Simulation:**DES**) は、計算機、通信、製造業、軍事、医療、環境、経営戦略等の各システムのシミュレーションに用いられている。気象シミュレーションのように流体 (大気) の運動を連続的に追跡するのではなく、例えば通信システムに於いてはパケットの交換機への到着等といった特定の "瞬間" のみに着目して、その出来事 (= 事象) によるシステムの内部状態 (パケットの到着による交換機のメモリ溢れ等) の変遷を追跡するのが DES である。こうした DES も、連続系のシミュレーションと同様に、対象としているシステムの規模が大きくなるに従って膨大な時間を要するため、並列処理による高速化の研究が続けられている。

並列離散事象シミュレーション (Parallel Discrete Event Simulation:**PDES**) の研究の歴史は十数年にもなり、基本的な同期プロトコルおよびそれらに改良を加えたもの、これらの機構を実現するライブラリや言語処理系、逐次シミュレーションプログラムの自動並列化等、DES の並列化のための様々な手法が提案されてきた。並列計算機での並列シミュレーションのための汎用ツールは商用レベルでの実用化の段階にさしかかっている。また、最近では例にもれず、PDES の分野でも近年のネットワーク技術の進歩の影響を受けた動きが出てきている。例えば、WEB ベースシミュレーションは、インターネット上での分散環境でシミュレーションを並列実行する試みである。ATM やギガビット・イーサ等の普及に伴い、ワークステーションクラスタや PC クラスタ等もハイエンドの並列計算機に比肩する程の性能を出し始めている [1]。

一方、並列プログラミングのための環境も序々に整備されており、MPI[2]、PVM[3] 等の通信ライブラリの標準化がなされるに至って可搬性の高いアプリケーションの開発が容易になりつつある。

これらの PDES や汎用並列計算環境の分野での技術の進歩にもかかわらず、DES の効率よい並列化には、まだハードウェアやプロトコル同期機構等の低レベルの知識とプログラミングの習熟が必要なのが現状である。逐次 DES では特に考慮せずに済んだような事柄が並列実行の際には大きな性能低下の要因になったり、プロセッサの待ち時間を減らすための投機的な実行の辻褄を合わせる機構がメモリを多く消費したりするためである。シミュレーションの並列化と同時に、このような不具合を除くための機構もシステムに隠蔽されていることが望ましい。PDES 研究グループと一般の DES を行っている人々との間の認識の隔たりに関する懸念は、今に始まったことではない [37]。PDES は DES を高速化するための唯一の手段ではないが、それでもなお、筆者は他のアプローチ (importance sampling[4] や variance reduction[5] 等) よりも第一原理に忠実にプログラミング出来る点で、最も直感的な方法であると考える。

1.2 本論文の構成

本論文では、第 2章で並列離散事象シミュレーションの概要、シミュレーションの並列化の性能面の検討や実現の枠組、実装の形態、最近の研究動向と PDES の実用化へ向けての課題に対する当該研究分野における取り組みについて述べる。

次に、第 3章以降では筆者の所属研究室で通信システムの設計に用いられていた待ち行列シミュレーションを並列化した経験を基に、今後 PDES の研究に必要な要素についての問題を提起し、以下のような枠組みを提案する。

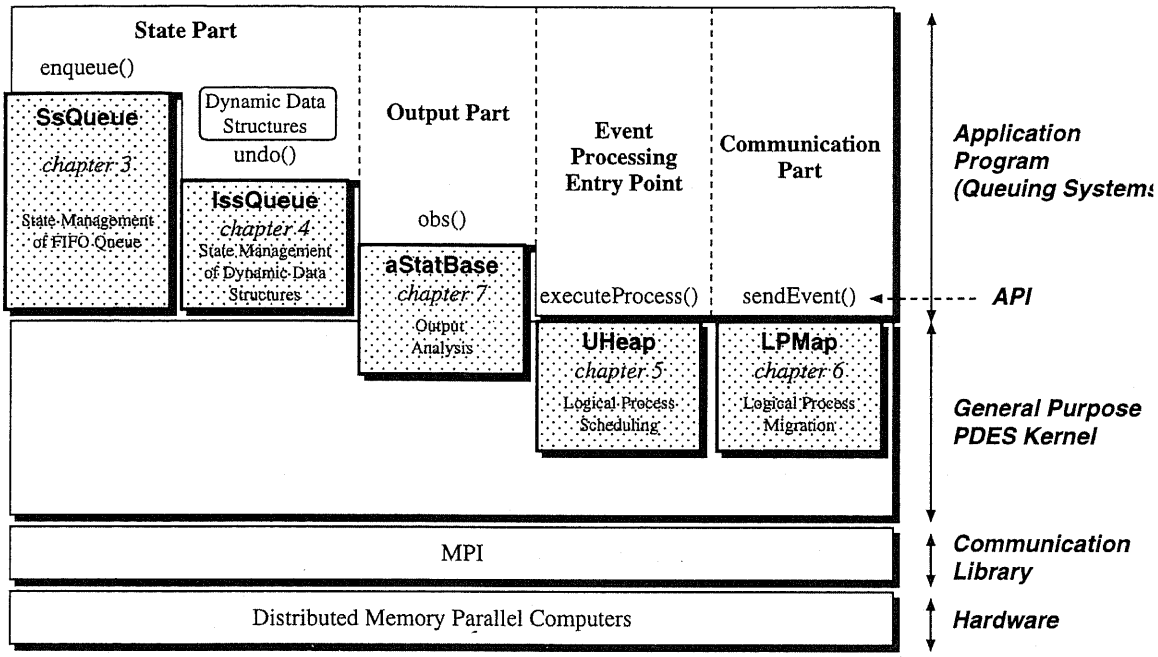

図 1.1: 本研究の位置付け

楽観的プロトコルにおける待ち行列データ構造の状態管理の支援 (第 3章) 待ち行列シミュレーションでは待ち行列データ構造をモデルの内部状態として保持しておく必要があるが、投機的実行のためにロールバックを実現しなければならない楽観的プロトコルの下では、待ち行列データ構造のような動的なデータ構造を容易に状態として実現することが出来ない。

待ち行列シミュレーションで多用される先入れ先出しキュー (FIFO キュー) は一旦挿入された要素間の順序変更は起らないため、その性質を利用した最適化が可能である。

本論文では、FIFO キューをコピーに基づく状態管理を行うタイムワープ汎用 PDES プラットフォームで状態変数として実現する、平易なインタフェースを持つ C++ のクラスライブラリの提案と性能評価を行う。更に多段接続交換網シミュレーションに対する組み込みを行うことにより、実アプリケーションへの適用の容易性と有効性を示す。

楽観的プロトコルにおける動的データ構造のための状態管理の支援 (第 4章) タイムワープ方式のプラットフォームは、同期の管理について、状態回復処理を行うタイミングの決定等の機能を共通して実現している。本論文では更に、前章で提案した枠組みを一般化して、動的データ構造の状態管理を支援する枠組みについて提案する。

サブモデルを効率よく物理プロセッサに集合化する枠組みの支援 (第 5章) PDES において、一般に、並列動作するサブモデルの数と物理プロセッサ数は異なるため、通常単一のプロセッサが複数のサブモデルをシミュレートする。この場合、複数のサブモデル間での実行順序の解決 (スケジューリング) が必要となる。

スケジューリング手法の一つとして、サブモデルが持つ事象リストの先頭要素の時刻印の小さい順に制御を移す最小時刻印 (Least Time Stamp First;LTSF) 法がある。また、保守的同期プロトコルの下では、サブモデル内での送信元毎の外部のサブモデルから到着するイベント間のスケジューリングも必要である。

これ等のスケジューリングのために用いるプライオリティキューの実装方式として、最高優先度の要素の取り出し、要素の挿入が高速に行えるよう様々な方式が提案されている。

上記のスケジューリングの場合、サブモデルの数が変化しない限り要素の数が一定で、各要素が優先度を上下させるという特徴があり、こうした特徴を生かした効率よい実装としてトーナメント形式のデータ構造が提案されている。このデータ構造を用いると、要素の優先度の変更が平均で定数コストで済むが、LTSF スケジューリングのような最高優先度の要素の変更は木構造の根から葉まで全て再評価する必要があるため、常に $O(\log N)$ のコストが必要である。このような問題は、二進木構造を持つヒープを用いて解決することが出来る。

サブモデルのプロセッサへの割り付けを柔軟に行う枠組みの支援 (第 6章) PDES プログラムは外部イベントの交換のため極めて通信集約的であり、サブモデル間通信が全体の性能に与える影響は大きい。サブモデル間の通信パターンと、物理プロセッサ間の結合トポロジとの間で何らかの整合を取ることによって、物理プロセッサ間の結合ネットワークに対する負荷を軽減させ、性能向上を図ることが出来る。PDES は行列計算のような比較的規則性の強い科学技術計算と異なり、静的な通信パターンも実行時の動特性もより複雑であり、最適なサブモデルから物理プロセッサへのマッピングを求めることは非常に困難である。更に、今日のマルチユーザに対応した並列計算機においては、物理プロセッサ網の中の有限集合を実行時に割り付けられることになる。このような場合は、たとえサブモデル間通信パターンが静的であっても最適なマッピングはコンパイル時には決定出来ないため、マッピングの決定は実行時に適応的に行われることが望ましい。

サブモデルの動的マッピングを実現するためには、サブモデルの実行時の物理プロセッサ間の移送機構が必要である。また、移送に関して何らかの基準を設け、プログラムが最適な動作点の近くで安定することを保証することが必要である。

精度判定を含めた出力統計処理の支援 (第 7章) 逐次シミュレーションの分野で通常提供されている、所望の精度が得られた時点で自動的にシミュレーションを停止させる機構 [5] が、PDES の分野ではあまり研究されていない。統計データの収集機構を PVM を用いて実現したものはあるが [6]、精度判定まで任せられる機構はあまり研究されていない。このような統計処理を効率よく実現するには、シミュレーション本体同様負荷をうまく分散させる必要がある。MPI 等の通信ライブラリの普及に伴い、精度判定に必要なリダクション演算も実装しやすくなっていると思われる。

第 2 章

並列離散事象シミュレーション

2.1 離散事象シミュレーション

計算機上で行われるシミュレーションは、物理的な対象の振るまいを計算機の中でモデル化し模擬することであり、モデルの構成概念により、いくつかの分類方法が存在する。流体力学や構造解析等、対象の連続的な変化を扱う**連続系**と論理回路や待ち行列システム等、システムの変化を瞬間瞬間の事象の連なりとして捉える**離散系**に分けるのも、そのひとつである。本論文では後者の並列処理を扱う。一般によりなじみのあると思われる気流シミュレーション等は連続系のシミュレーションに属する。

	離散系	連続系
時間進展	不規則	一定
モデル	状態遷移	微分方程式
解析手法	統計学 / 待ち行列理論	線形代数
シミュレーション エンジン	イベント スケジューラ	行列計算 ライブラリ
高速化手法	importance sampling variance reduction	過剰緩和法 共役勾配法

表 2.1: 離散系シミュレーションと連続系シミュレーション

一方、工学における設計や検証等の様々な場面でも、計算機によるシミュレーションが行われている。試作品を作って実際に動作させて所望の機能を実現しているかを検証するのに比較して、材料等の資源を使わず、比較的速やかに、しかも着目している側面だけに集中して検証を行うことが出来る。このようなシミュレーションに使われている事象駆動の概念について、「街角に設置する公衆電話の数を何台にすれば良いか」という意思決定の例を基に概説する。

図 2.1は、公衆電話に行列が出来る様子と、その状況をモデル化して計算機で実現された仮想的な世界の対応を示す。現実世界での時間に応じて、計算機にも仮想時刻 (**Virtual Time:VT**) が用意される。また、電話を掛ける人が到着する、通話を終えるといった出来事に対応して、計算機の中ではイベントを表現するデータを用意する。イベントには、それが生起した時刻の情報が付随しており、それを**タイムスタンプ**と呼ぶ。イベントはタイムスタンプの順に並べられ、そのリスト (= **イベントリスト**) の先頭から順にイベントを取り出して処理する。イベントの処理はシミュレーション対象の世界の状態に応じて行われる。この例では丁度右側の電話を使っていた人が受話器を於いて立ち去る場面を処理している所である。右側の電話機は一時的に空き (IDLE) になるが、行列に人が待っているため、次の瞬間にすぐ使用中 (BUSY) になり、シミュレーションは進行する。

次の節では、シミュレーションプログラム開発の実際について、最も基本的な機能のみを備えたラ

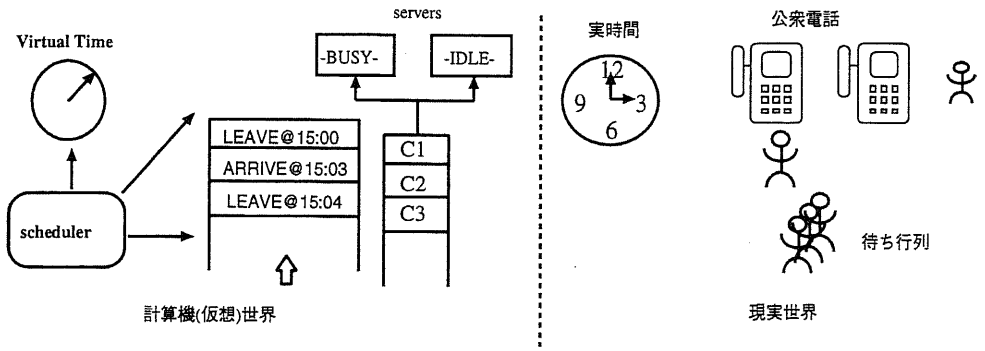

図 2.1: DES の例～公衆電話の設置台数の決定

イブラリである SMPL[5] と、通信システムのシミュレーションに必要なあらゆる構成要素を支援系側が用意する、ユーザインタフェースも洗練されたパッケージである OPNET[7] の両者によるプログラムの開発について述べる。更に次々節ではシミュレーション支援系の基本的な構造について述べる。

2.2 シミュレーションプログラムの開発例

2.2.1 SMPL によるモデルの記述例

SMPL[5] は主に待ち行列システムのシミュレーションを目的としたシミュレーションライブラリである。複数のサーバを持つファシリティと呼ばれるデータ構造が組み込みで用意されており、プログラマはサーバの占有要求、解放に対応するイベントを定義し、対応する処理ルーチンを記述する。サービスを受ける実体はトークンと呼ばれる識別子で管理され、サーバの数より多くのトークンが占有を要求すると、トークンはライブラリの用意する待ち行列に格納され、サーバの解放時に (拒否されて待ち行列に格納された) トークンの要求に対応するイベントが自動的に再スケジュールされることにより、ユーザが待ち行列を明示的に扱わなくても良い仕組みになっている (図 2.2)。

2.2.2 OPNET によるモデルの記述例

OPNET[7] は、状態遷移モデルに基づくネットワークシミュレーションツールであり、ネットワークの階層構造に対応してモデルも階層構造を取る。TCP/IP 等広く知られている通信プロトコルの処理ルーチンは組み込みで用意されており、優れた GUI によりモデルの記述やデバッグを容易にしている。モデルの階層構造はトップレベルのネットワークからサブネット、ノード、プロセスという構成になっている。実際の処理をする部分であるプロセスは、有限状態機械として構成され、ユーザは状態遷移の契機としてのイベントを定義し、遷移の前後の処理を、C に似たプログラミング言語で記述する (図 2.3)。プロセスより上位の階層は GUI により編集することが可能になっている。

2.3 シミュレーションエンジンの構造

前節で取り上げたパッケージは機能の大きさの点で両極端にあると言えるが、一般にシミュレーション支援系は、イベントの生起、スケジュールおよび VT の管理を行うシミュレーションエンジンと呼ばれる機構を備える。スケジュールされたイベントはリスト構造で管理され、イベントには生起時刻

```
1:/* ファシリティの宣言 */
2:f = facility("Telephone",2);
3:while(/* 終了条件 */) {
4:  cause(&event,&token);/* イベントの生起 */
5:  switch(event) {
6:  case 0:
7:      ArriveCustomer(token); /* 到着 */
8:      break;
9:  case 1:
10:     ReleaseTelephone(token);/* 終了 */
11:     break;
12: }
13:}
14:/* 到着処理ルーチン */
15:ArriveCustomer(int token) {
16:  if (request(f,token,1)) {/* サーバの要求 */
17:     return; /* 要求が拒否された場合 */
18:  }
19:  /* 確率分布に基づき保有終了をスケジュール */
20:  schedule(1,expntl(interval));
21:}
```

図 2.2: SMPL によるシミュレーションプログラムの主要部分

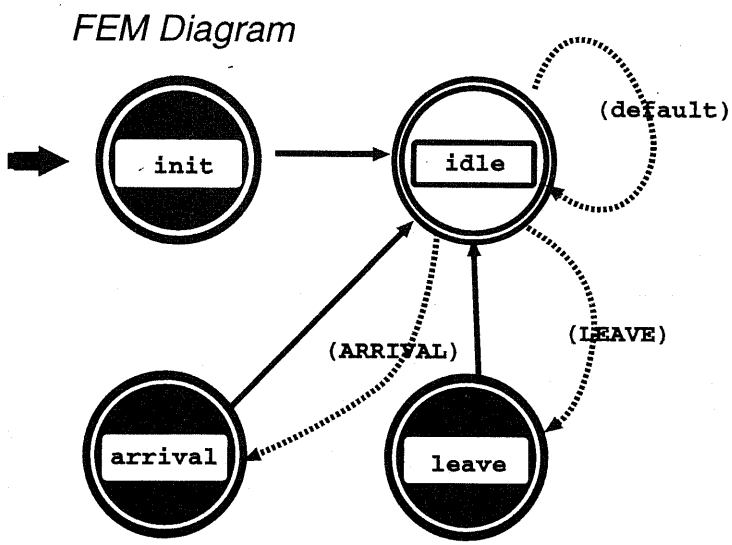

図 2.3: OPNET における状態遷移図によるモデル開発例

(タイムスタンプ) が対応付けられている。生起時刻が最小のイベントを効率良く選択出来るようになっている。次に起こるべきイベントの選択の際に、VT もそれに対応した生起時刻まで進められる。モデルの状態を表すデータ構造はアプリケーションにより異なるが、SMPL の場合はファシリティが、OPNET ではプロセスのプログラムの中で宣言している状態変数が対応している。

2.4 シミュレーションの並列化 [8]

2.1節の例で述べたようなシミュレーションを実行時間の観点から見ると、一台のプロセッサでも十分有意な結果を得ることが出来ると思われる。シミュレーションは実際に実験出来ない (したくない) ことを模擬するため、実際に試作する (実験する) 手間に比べて十分容易である必要がある。ところが近年のシステムの複雑化 / 高度化に伴い、シミュレーションも大規模化して時間がかかるようになってきた。例えば Asynchronous Transfer Mode (ATM) ネットワークにおけるセル損失率の要求は 10^{-9} 程度とされており、シミュレーションに応じて所望の信頼区間を得るためには 10^{11} 回程度のセル到着のシミュレーションが必要となる。

ここで並列処理技術による高速化が考えられるが、対象とするモデルが物理的に複数の領域に分かれる場合は、そのサブモデル毎にシミュレーションエンジンを並列に動作させることにより、全体としてのシミュレーションプログラムの実行速度を向上させることが出来る。

次節以降では、このような並列化の枠組と、そのような並列化をした場合のプロセッサ台数に対する理論的な性能の上限、別の観点からの並列性の抽出、シミュレーションの進め方に関する二つの主なプロトコル、その進行に付随する問題点、並列シミュレーションシステムの支援系の既存の実現形態について述べ、最近の研究動向と PDES の実用化へ向けての課題に対する当該研究分野における取り組みについてまとめる。

2.4.1 論理プロセスによるシミュレーション

シミュレーションの高速化のアプローチとして、モデルに内在する並列性を基に元のシミュレーションタスクを互いに通信し合う**論理プロセス** (Logical Process: 以下 **LP**) に分割して並列実行させることが考えられる。LP はイベントの発生する時空間の部分集合 (region;R) をシミュレートする。通信システムにおける各ルータや端末に相当するものである。図 2.4にそのアーキテクチャを示す。

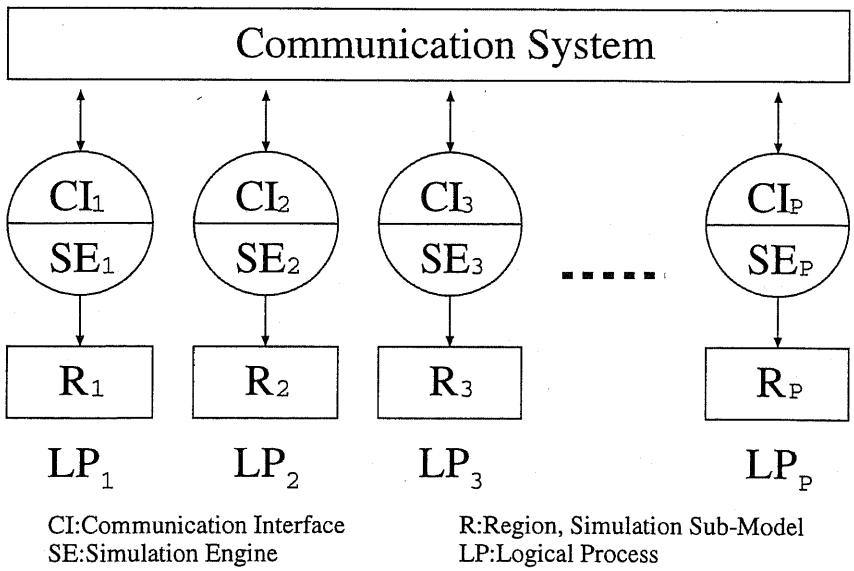

図 2.4: 論理プロセスによるシミュレーション

LP の集合 イベントの生起を同期的または非同期的に並列実行する。

通信システム CS データの交換と各自の実行の同期に用いる。

R,SE,LVT 各 LP はシミュレーションモデルの部分集合 R を割り当てられ、各自のシミュレーションエンジン SE で**局所仮想時間** (Local Virtual Time, 以下 **LVT**) を進めながら事象駆動シミュレーションを行なう。

状態 S 各 LP は元の状態空間の部分集合である局所的状態変数のみにアクセス出来、イベントの生起にしたがって S を書き換えつつシミュレーションを行なう。他の LP 上の S を直接書き換えることは出来ない。

内部イベントと外部イベント 各 LP で処理されるイベントは、**内部イベント**と**外部イベント**に分けられる。内部イベントは内部状態 S のみに因果関係を及ぼし、外部イベントは他の LP の内部状態にも影響の及ぶものである。

通信インタフェース CI 各 LP から遠隔 LP へのイベントの影響の伝搬、遠隔 LP からの各 LP へのイベントの影響の伝搬を適切に行なうために、送信する瞬間の LVT を付加したイベントメッセージを送受、処理する機構を備える。

CI はイベント処理の進め方に於いて**保守的**なものと**楽観的**なものと、大きく 2 種類に分けられる。保守的な CI は LP 間の因果関係の制約が破られないような実行をするのに対し、楽観的 CI では因果関係の矛盾が生じた場合は矛盾のない時点まで遡って再実行する。

また、LVP の進め方について、同期的 LP シミュレーションと非同期的 LP シミュレーションに大別される。同期的 LP シミュレーションはタイムステップシミュレーションとも呼ばれ、LVT をある間隔 Δ で一斉に進めながらシミュレーションを行なう。各 LP が同一の区間をシミュレートしていることが保証されるため、明示的な時刻管理が不要となり、実装が簡単でありかつデッドロックやメッセージトラヒックの輻輳とは無縁であるが、各ステップ毎に LP の負荷のバランスが取れていなければ効率が悪くなる欠点がある。

非同期実行の効果

非同期 LP シミュレーションは、イベントが LP 内部の処理だけで済む場合が多いときに有効である。こうした処理を並列に行なうことにより逐次実行に比べて効率のよい高速化が達成される。ただし実際は他の LP への影響を及ぼすイベントも発生する。[8] では非同期実行の同期的実行に対する性能向上比を以下のように統計的な手法により導いている。各 LP での 1 タイムステップに要する実行時間は負指数分布に従うとする、つまり

$$\forall \mathrm{LP}_i, T_{step,i} \sim \exp(\lambda), \mathrm{E}[T_{step,i}] = \frac{1}{\lambda} \tag{2.1}$$

とすると、同期的シミュレーションの k ステップの実行時間の期待値 $E[T^{synch}]$ は、

$$E[T^{synch}] = k\mathrm{E}[\max_{i=1\ldots P}(T_{step,i})] \tag{2.2}$$

$$= k\frac{1}{\lambda}\sum_{i=1}^{P}\frac{1}{i} \leqq \frac{k}{\lambda}\log(P) \tag{2.3}$$

式 2.3は各タイムステップでの同期の成立にかかる時間の期待値の k 倍であり、恐らく $k\int_0^\infty \left\{1-(1-e^{-\lambda t})^P\right\} dt$ で求められたものと思われる。

一方、LP 間の依存がなくシミュレーションが完全に非同期に実行される場合、実行時間の期待値は、

$$E[T^{async}] = E[\max_{i=1...P}\langle kT_{step,i}\rangle] \geqq \frac{k}{\lambda} \tag{2.4}$$

よって、

$$\lim_{k\to\infty,P\to\infty} \frac{E[T^{sync}]}{E[T^{async}]} \approx \log(P) \tag{2.5}$$

つまり、シミュレーションステップ数とプロセッサ数を大きくしてゆくと、非同期シミュレーションは同期シミュレーションに比べて最大 $\log(P)$ 倍の性能が得られ、逆に同期シミュレーションではプロセッサを P 個用いても速度は $P/\log(P)$ 倍にしかならないことを示している。

ただし、$T_{step,i}$ を有限区間の一定分布と仮定すると両者の比率は定数 2 となることが導かれる。

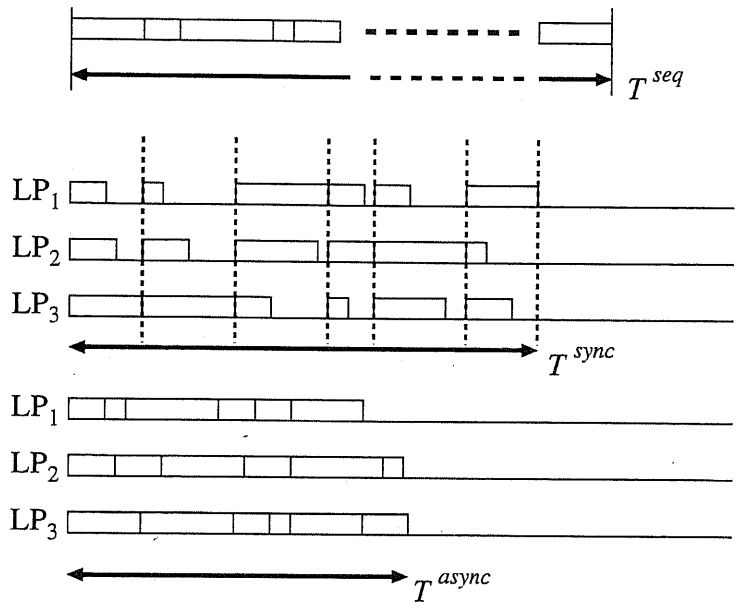

図 2.5: 逐次 / 同期並列 / 非同期並列各シミュレーション方式の実行時間の期待値

時間方向の並列性の抽出

シミュレーションプログラムのサブモデルに基づく並列処理では、サブモデルの数以上の並列性が得られないことから、サブモデル以上の数のプロセッサを生かした高速化が出来ない。そのような限界を打破すべくモデルから空間のみならず時間領域での並列性を抽出する手法が考案されている [9]。この方式は、時間 (VT) 軸 (シミュレーション開始時刻から終了時刻迄) と空間 (モデルの空間的広がり) で張られる 2 次元の領域をプロセッサ数に応じて分割し、各領域でのシミュレーションを並列に行う。各プロセッサは初めに仮定された境界条件を基に担当する領域のシミュレーションを行い、新たに得られた境界条件を隣接するプロセッサに伝達するということを境界条件に変化がなくなるまで繰り返すという点で連続系のシミュレーションにおける緩和法と同じアプローチである。単純な適用では効率が上がらないが、[10] ではイベントの生起パターンに何らかの繰り返しが含まれるような場合の高速化に成功している。

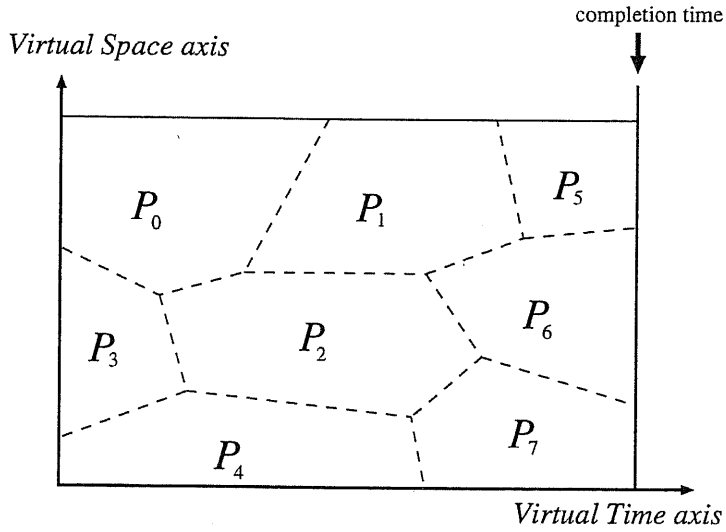

図 2.6: 時間領域の並列性の抽出

2.4.2 シミュレーションプロトコル

イベント駆動シミュレーションに於いては、正しい結果を得るためにはイベントは生起時刻順に処理されなければならない。この制約を満たすためにサブモデルをシミュレートする LP は何らかの同期をとる必要があるが、その手法は大きく二つに大別される。常に生起時刻順に忠実に実行し、現在及び将来受け取る可能性のある外部イベントのタイムスタンプの最小値より先へは決して LVT を進めない保守的なプロトコル [11] と、そのような制約なしに実行を進め、LVT より小さなタイムスタンプの外部イベントの到着等の不具合が生じた時初めてその時刻まで立ち返ってやりなおす楽観的プロトコル [12] である。

保守的プロトコル

保守的プロトコル [11] に基づく LP の構造を図 2.7に示す。外部イベントは送信元 LP 毎の入力バッファ (InputBuffer;**IB**) に振り分けられ (通信システムでのメッセージの FIFO 到着を前提にしている)、各 IB の先頭のイベントのタイムスタンプを表すチャネルクロック (Channel Clock;**CC**) の最小値 (Local Virtual Time Horizon;**LVTH**) までのイベントを処理することにより自動的に同期が取られる。

楽観的プロトコル

楽観的プロトコル [12] に基づく LP の構造を図 2.8に示す。外部イベントはタイムスタンプに拘らず到着順に処理されるため、受信バッファは一つである。LVT より大きなタイムスタンプのイベントが到着する等の因果関係の違反が生じた場合は、原因となった時刻まで状態を戻し (**ロールバック:roll-back**)、誤った処理によって送信した外部イベントを取り消すためのメッセージ (**antimessage**) を送出し、違反の生じる前の状態から再開する。この処理を可能にするため、状態と実行されたイベントの履歴と、送信を取り消す必要のある外部イベントを検出するための送信履歴を保持するバッファが設けられる。一般に履歴を保存する記憶領域は有限であるため、適宜 LP 間で情報をやりとりしてロールバックの可能性のある時刻の最小値 (Global Virtual Time;GVT) を計算してそれ以前の履歴

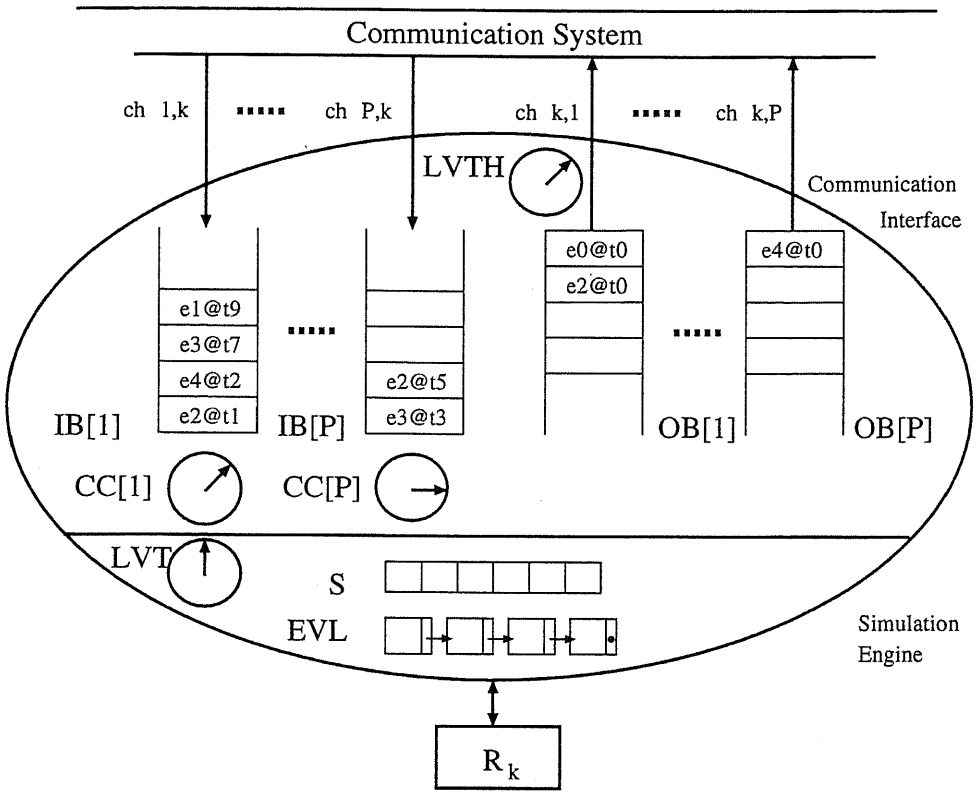

図 2.7: 保守的論理プロセスのアーキテクチャ

を破棄する。

両プロトコルの比較 (表 2.2)

保守的手法と楽観的手法の比較はしばしばなされるが、両者の性能は互いに関係する様々な要因が絡み合うため一概に規定出来ない。実装の方法やプラットフォームにも依存する。保守的手法は、実装も比較的単純であり、データ構造も単純であるが、外部イベントの通信パターンにループが存在するとデッドロックが起こるため、それを回避する機構が必要になり、その機構の性能が全体の性能を大きく左右する。一方楽観的手法は、待ちの状態を作らずイベントを投機的に実行するため潜在的な実行効率は良いが、ロールバックのオーバーヘッドが大きい。履歴の保存にかかるメモリの消費とメモリ管理のオーバーヘッドを抑えるための複雑な実装技術も要求される。

2.4.3 保守的プロトコルと Lookahead

保守的なプロトコルを 2.4.2節で述べた方法のみを用いて実装すると、LP のネットワークにループが存在する場合には、デッドロックが起きる可能性があり、これを回避する付加的な機構が考案されている。例えば送信すべきイベントがなくても LVT の値を情報として送る等である (ヌルメッセージ)。一般に保守的なプロトコルでは次に受け取り得る外部イベントのタイムスタンプの最小値が大きい程、休止せずに連続してイベントの処理が出来るため効率がよい。そこでヌルメッセージを送出する際に、LP の状態と次にスケジュールされているイベントから、次に送出する可能性のある外部イベントのタイムスタンプの最小値を送出する方法がある。現在の LVT とその値の差を **lookahead**

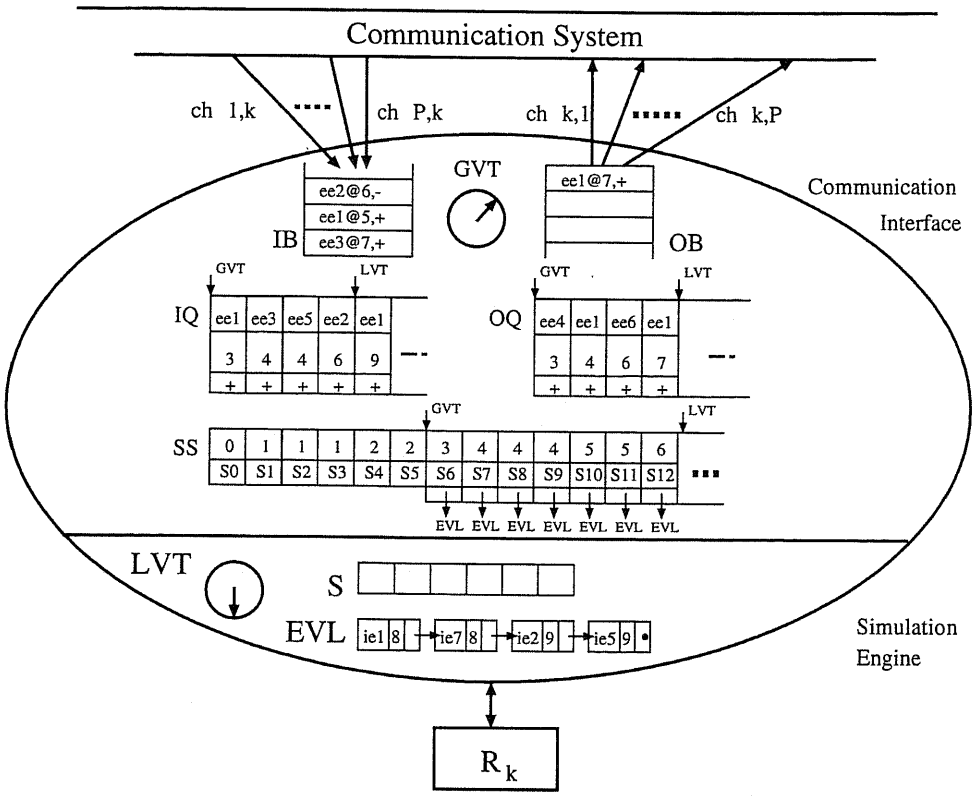

図 2.8: 楽観的論理プロセスのアーキテクチャ

と呼び、この値が大きいほど、同時に、無駄に飛び交うヌルメッセージの数も減らすことが出来る。lookahead は LP 間の伝搬遅延やイベントの最低処理時間等から求める他、シミュレーションエンジン内の乱数列を前もって調べる事により将来の処理時間を予め計算してさらに大きい lookahead を求める研究も行われている [13]。

2.4.4 同時刻イベントの扱い

シミュレーションの実行の過程で、互いに同じタイムスタンプを持つ複数のイベントがスケジュールされる場合がある。このようなイベントの扱いはモデルにより異なるが、毎回同じ結果が得られる (決定論的) 必要のある場合や、その瞬間の状態とイベント同志の優先順位等により決まった処理が必要になる場合は、単にイベントリストの先頭のエントリを選択して実行する方針では不十分で、その時刻にスケジュールされたイベントのうちの一つが実行される前に同時刻のイベントが全て LP 上に存在していなければならない。この条件を満たす為には、保守的なプロトコルのみならず、逐次プロトコルや楽観的プロトコルに対してもある種の lookahead が必要であることが知られている [14]。

2.4.5 プラットフォームの実現形態

モデルの並列実行のために同期プロトコルを直接実装するのは煩雑であるため、同期を司る部分をモデルの記述と分離し、シミュレーションプログラムの開発者がモデルの記述に専念出来るよう、様々なプラットフォームが提案されている。これらは採用している同期方式、ターゲットとして想定されている並列計算機のアーキテクチャ、および実現形態により表 2.3のように分類することが出来る。

方式	保守的 (CMB)	楽観的 (Time Warp)
動作原理	l_{cc}(Local Causality Constraint) 違反は厳密に排除される。*safe*(“良い”) イベントのみ処理する。	l_{cc} 違反を許すかわりに検出されたら回復する。“悪い” イベントも処理する。“良い” イベントはコミットされ、“悪い” イベントは後で必ずキャンセルされる。
同期	プロセッサの閉塞による。プロトコルに起因するデッドロックの回避機構が必須。ヌルメッセージによる回避は通信オーバーヘッドが大きく、デッドロック検出 / 回復プロトコルは集中型のデッドロック管理機構を要する。	ロールバックによる。遠隔抹消 (remote anihilation) によるオーバーヘッドが大きい。ロールバックは終了することが保証されているが、連続すると性能低下とメモリ消費がはげしくなる。
並列性	モデルに内在する並列性は十分に引き出されない。LP 間のイベントの因果関係が疎な場合はプロトコルの保守的な振舞のため却って損失が大きい。	モデルの並列性は十分引き出される。因果関係が密な場合は大抵良い性能が得られる。
Lookahead	これがないと動作不能。性能向上のためにも不可欠。	最適化のために用いられることはあるが本来不要。
負荷の均衡	通信路が均等に割り当てられていれば問題ない。イベントの空間的、時間的な分散の度合は性能に影響しない。	LVT の進度が平均して均等であれば性能を発揮する。イベントの空間的、時間的な分散の度合が大きいと性能の低下を招く。
GVT	暗黙の内に GVT にそって実行されるため GVT を明示的に計算する必要はない。	明示的な GVT 管理が必要 (重い計算)。GVT の集中制御: 通信のボトルネックとなる (ハードウェアサポートがない場合)。分散アルゴリズムは通信のオーバーヘッドが大きく性能は劣るようである。
状態	メモリの使用も保守的になるため任意の大きさの状態空間を仮定するモデルも扱うことが可能。	状態空間と、状態毎のメモリ消費量が小さい程性能は良い。
メモリ	消費量も保守的。	消費が激しい。状態の保存にオーバーヘッド。fossil collection の効率化と GVT の計算の頻繁な、効率良い実行が必要。メモリの枯渇防止に複雑なメモリ管理手法が必要。
メッセージおよび通信	タイムスタンプ順のメッセージの到着とイベント処理。入力チャネルは厳密分離。トポロジは静的。	非タイムスタンプ順到着、タイムスタンプ順イベント処理。単一入力キュー。通信路のトポロジは動的。通信路が FIFO でなくて良いため、より広範のプラットフォームでの実現が可能。
実装	素直な実装が可能。制御が容易でデータ構造も単純。	実装は容易ではなく、デバッグも困難。手の込んだ実行の制御とメモリの配分が不可欠。性能改善のための最適化実装方式がいくつか存在。
性能	主にデッドロック回避機構が決める。イベント当たりの計算と通信のオーバーヘッドは小さい。細粒度のシミュレーションモデルに適している。一般的な性能の議論はできない。	楽観性の制御とメモリ消費の管理が決め手となる。イベント当たりの計算と通信のオーバーヘッドは大きい。粗粒度のシミュレーションモデルに適している。一般的な性能の議論はできない。

表 2.2: プロトコルの比較 [8]

シミュレーションライブラリ

既存の逐次 (C[18, 19, 20], C++[21, 17, 22, 23]) または並列言語で書かれたプログラムから呼び出せるライブラリ関数群として提供されるパッケージとしての形態であり 2.4節で述べた LP の機能を持つ関数をユーザのイベント処理ルーチンから適宜呼び出す要領でプログラミングが行われる。

既存のプログラムを流用した逐次シミュレーションからの移行が比較的容易であり、新たな言語の習得の必要がない反面、システム側から意味上の誤りを検出するのが困難なためデバッグも困難であ

同期方式 / メモリ	楽観		保守	
	言語	ライブラリ	言語	ライブラリ
分散	MAISIE[15]/ Parsec[16]	WARPED[17]	MAISIE/ Parsec	UPS[18] CPSim[19]
共有	—	GTW[20]/GTW++[21] SimKit[22]/WarpKit[23]	—	—

表 2.3: 記憶方式と同期プロトコル、実現形態によるプラットフォームの分類

る。

GTW(Georgia Time Warp)[20] では、共有メモリ上に、空間軸と時間軸の 2 次元のインデックスでアクセス出来る Space-Time Memory を実装し、各 LP の履歴を保存している。他の LP 上の状態履歴に直接アクセス可能であるため、ロールバックの際には分散メモリ上で用いられる antimessage の発行の代りに直接他の LP 上にスケジュールされたイベントを消去する方式 (Direct Cancellation) による効率化が図られている。

CPSim[24, 19] は数少ない商用パッケージの一つである。

シミュレーション言語処理系

シミュレーションライブラリをランタイムライブラリとして用い、専用の言語で記述されたプログラムを、それらのライブラリ関数呼び出しを含むオブジェクトコードに変換する処理系である。新しく言語の習得が必要である難点があるが、シミュレーションライブラリによるプログラムの記述に比べて処理系がプログラムから得られる情報の質が向上し、意味的な誤りの検出や最適化が容易になる。MAISIE[15]/Parsec[16] はコンパイラに与えるオプションにより、生成されるコードの同期方式を切り替えることが出来る。

オペレーティングシステム

TWOS[25] は Time Warp プロトコルによるシミュレーションエンジンの実装であるが、オブジェクト (LP) のスケジューリングやメモリ管理等 OS としての機能を持つ。通常はホストコンピュータの OS と置き換えられることはなく、アプリケーションプログラムとして動作する。モデルはオブジェクト単位に分割され、プライベートな状態とイベントを実行するコードを記述する。オブジェクト間はメッセージを介して明示的に通信を行い、ロールバックや、そのための状態のチェックポインティングはシステムが行う。

以下、楽観プロトコルを用いた分散メモリ向け並列シミュレーションライブラリである WARPED[17] を例に具体的な PDES プログラムの開発について述べる。

2.4.6 WARPED による記述例

WARPED は米国シンシナティ大学で開発されている C++ のクラスライブラリによる TimeWarp シミュレーションライブラリである。通信機構には MPI[2] を利用しているため移植性が高く、実際本学の大型計算機センターに設置されている SR2201[26] でも殆んど手を加えずに動作させることが可能であった。

WARPED のクラスライブラリには、各プロセッサ上で複数の LP がシミュレート出来るように LP

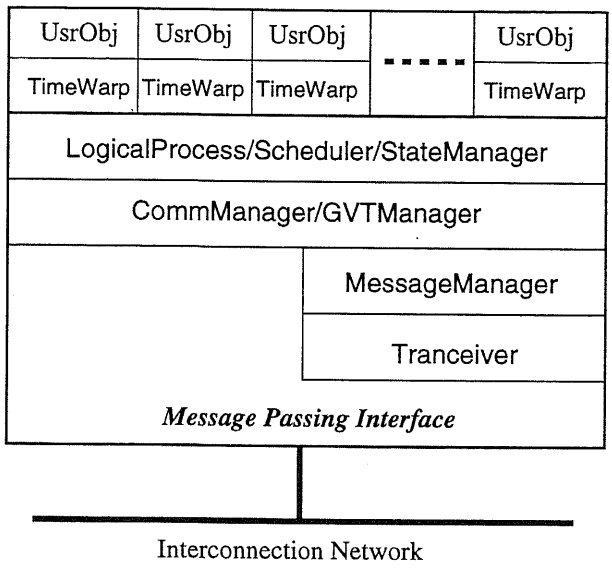

図 2.9: WARPED におけるノードの構成

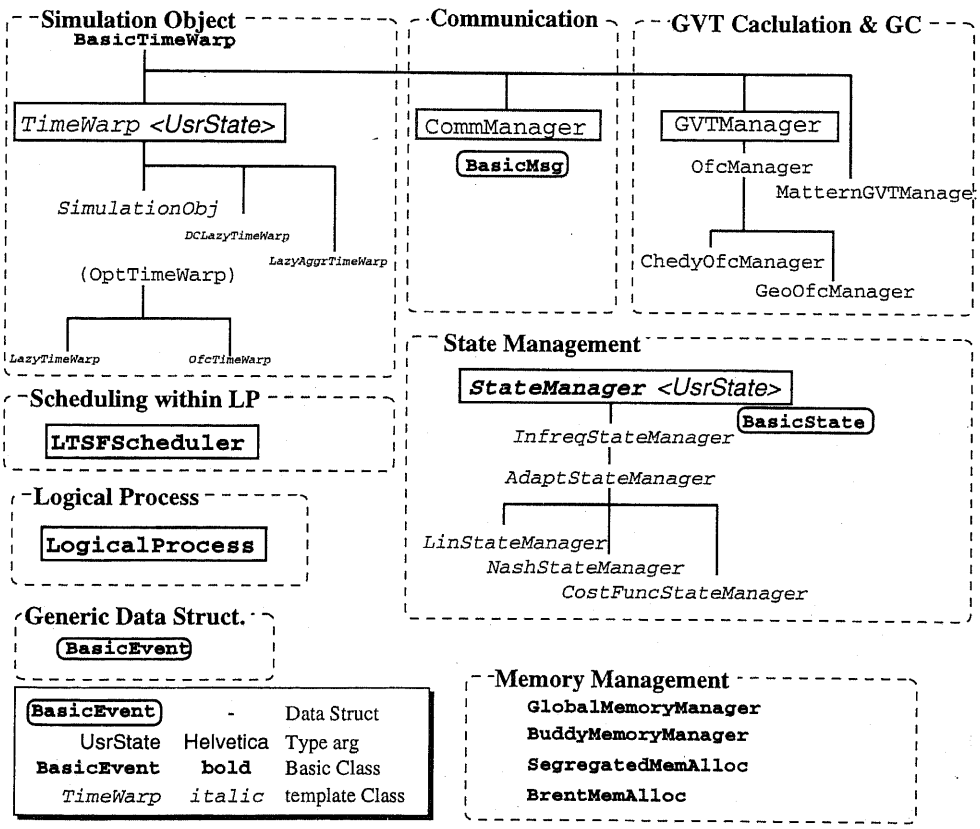

図 2.10: WARPED のクラス階層

を集合化する LogicalProcess クラス[1]、各 LP の実行順を決定する LTSFScheduler クラス、状態履歴の管理を行う StateManager クラス、イベントと MPI メッセージの相互変換と様々なイベント以外の同期メッセージの交換を司る CommManager クラスが含まれる。これらの管理機構はクラスにカプセル化されているため、それぞれ別の管理方針を採用している派生クラスと差し換えることが可能になっている (図 2.10)。

```
 1:#include "SimulationObj.hh"
 2:
 3:// 状態ベクタのデータ構造
 4:class PingState : public BasicState {
 5:public:
 6:  int numSent;  // 送信回数
 7:};
 8:
 9:// simulation object の定義
10:class PingObject :
11:  public SimulationObj<PingState> {
12:  int dest; // 送信相手の ID
13:public:
14:  void executeProcess(); // イベント処理メインルーチン
15:  void finalize();       // 終了処理 (統計情報の収集等)
16:};
17:
18:void PingObject::executeProcess() {
19:  BasicEvent *evt = getEvent(); // 次の event の取得
20:  // 相手へのメッセージの返答
21:  BasicEvent *newEvent = new BasicEvent;
22:  newEvent->dest = dest;
23:  newEvent->recvTime = getLVT() + PROP_DELAY;
24:  sendEvent(newEvent);  // イベント送信
25:  if (id == 0)  cout << "Ping!" << endl;
26:  else cout << "Pong!" << endl;
27:  state.current->numSent++;   // 状態の更新
28:}
```

図 2.11: WARPED 上のシミュレーションプログラムの主要部分

図 2.11に、二つの LP がイベントを投げ合う PingPong シミュレーションを WARPED 上で記述した例を示す。WARPED では、プログラマはシミュレーションオブジェクトの基本クラスである **TimeWarp** クラスを継承して独自の LP を定義し (10 行目～ 16 行目)、カーネル側からイベント処理の必要なタイミングに自動的に呼ばれるメソッド (14 行目の **executeProcess**) の本体を記述する。TimeWarp クラスはユーザの定義する状態ベクトルクラスを引数とするテンプレートクラスとして提供されている[2]。

打ち返した球の数を保存するための状態ベクトルを表現するクラスが 4 行目から 7 行目にかけて定義されている。LP はこの状態ベクトルを用いて定義される (11 行目)。ユーザは状態変数を定義さえすれば、スナップショッティングはライブラリが自動的に行う。

18 行目から 28 行目にかけて、実際のイベント処理の内容が記述されている。**getEvent** メソッドは次に実行すべきイベント構造へのポインタを戻す (19 行目)。23 行目では、打ち返される球の相手への到着時刻を **getLVT** メソッドで得られた現在時刻から伝搬遅延を加味して計算し、24 行目で実

[1] warped では、LogicalProcess という語は各プロセッサ毎に一つづつ配置されて複数のシミュレーションオブジェクトを集合化するものである。ここでは複数のシミュレーションオブジェクトを従来の慣習に従って LP と呼ぶ。

[2] 新しいバージョンの warped[27] では、コンパイルの効率化のために非テンプレートクラスとして提供されているが、機能的に特に大きな差異はないようである。

際にイベントの送信を行っている。27 行目では状態ベクトル中の打ち返した球の数を表す変数を更新している。

2.5 最近の動向

本節では、最近の PDES の研究分野における動向について、同期プロトコルの発展、最近注目されている World Wide Web 上でのシミュレーションについて順に述べる。

2.5.1 同期プロトコルの動向

同期プロトコルは基本的には保守的なものと楽観的なものが存在するが、どちらも全てのアプリケーションに最適である訳ではなく、両者の要素を採り入れた中間的なプロトコルや、両者を動的に切り替える方式等が提案されている。本節では特に、楽観プロトコルに若干の保守的要素を加えて最適化する手法を紹介する。

BTB(Breathing Time Buckets)

楽観的プロトコルとして初期に提案された TimeWarp 方式では、2.4.2節で述べたように LP は局所的なイベント処理のみならず、将来取り消されるかも知れない外部イベントの送出も投機的に行う。しかし送出してしまった外部イベントの取り消しは局所的なロールバックに比べて更にコストが高い。そこで投機的な実行を LP 内に止め、外部イベントは保守的に送信する SPEEDES[28] という楽観的プロトコルが提案されている。

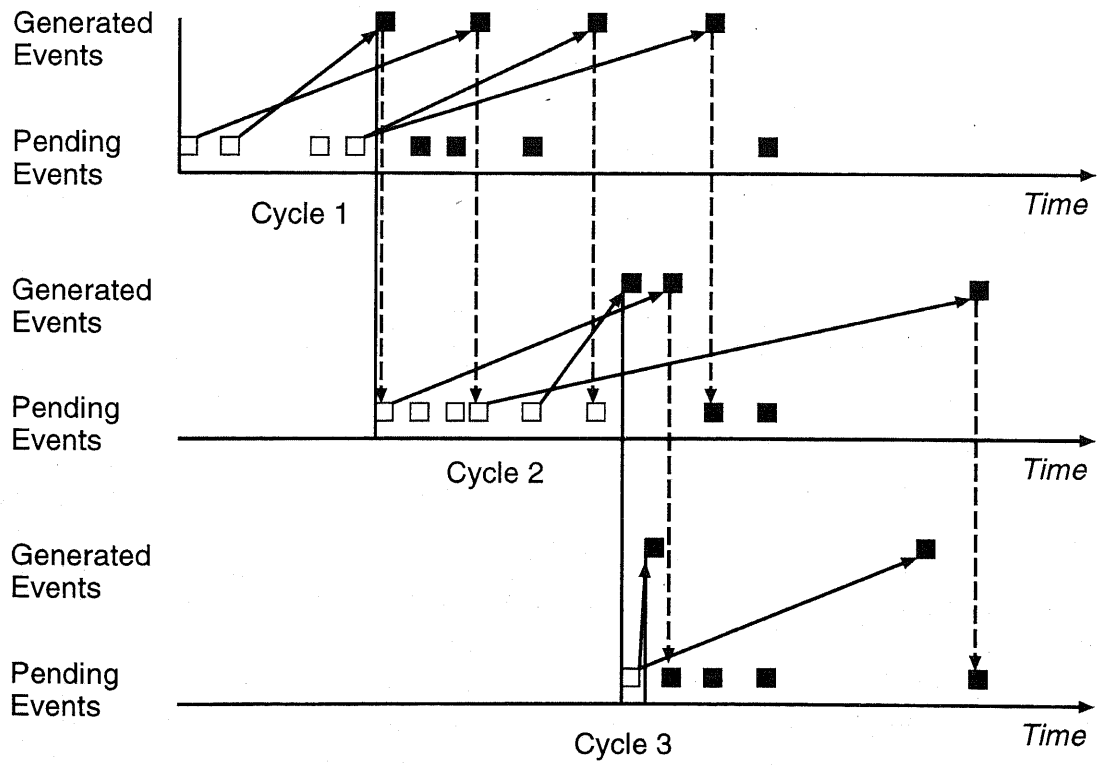

図 2.12: SPEEDES[29] におけるイベントリスト管理

SPEEDES における同期方式について解説する前に、一般にイベントリストを高速に管理する機構としての Event Horizon[29] の概念について図 2.12に示す。従来のイベントリストの管理方式では、新たに生成されたイベントは直ちにイベントリストに挿入されるため、その度毎に時刻順に正しい位置に挿入するコストが必要であるが、本方式ではそのようなオーバーヘッドを削減するために、以下の処理を繰り返す方法を取る。

1. 整順済のイベントリスト EQ の先頭からイベントを取り出して実行する。
2. 発生したイベントを並べ替えずに仮のリスト TQ に追加する。
3. 現在の EQ の最早生起イベントのタイムスタンプ LT を更新する。
4. TQ の最早イベントのタイムスタンプと LT を比較して、LT が小さい場合 (Event Horizon に到達)、6 へ。
5. 1 へ戻る
6. TQ を整順して EQ に挿入し、1 へ戻る

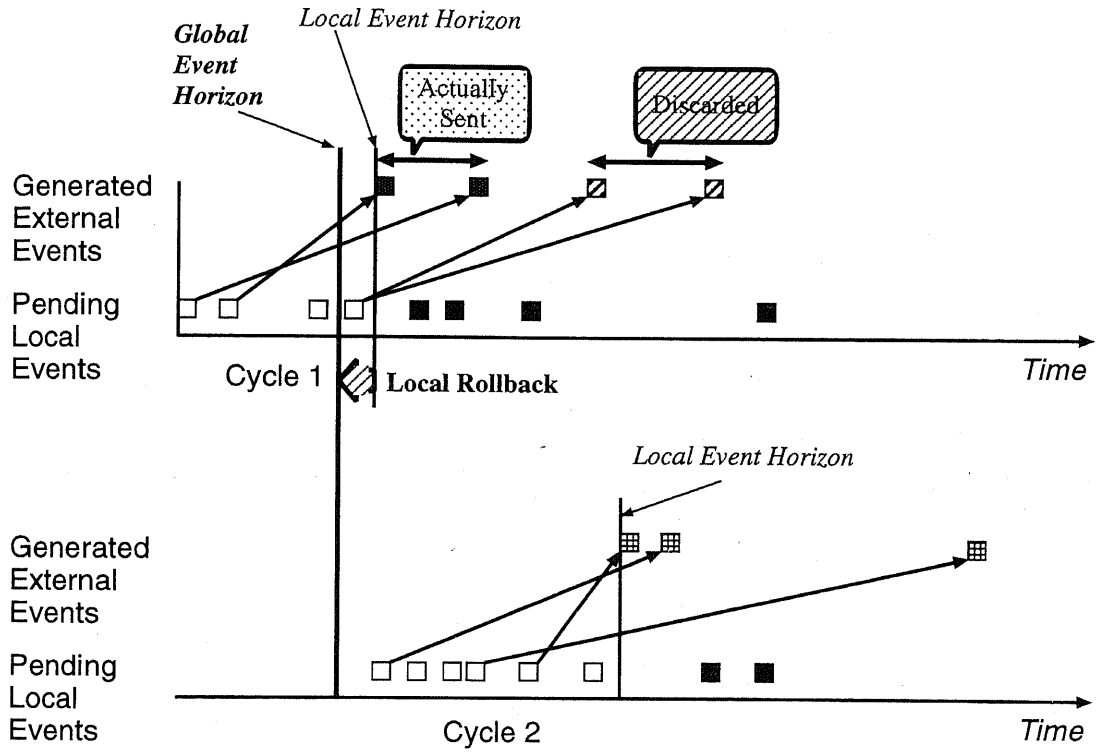

図 2.13: SPEEDES[28] における同期

SPEEDES では、各 LP は投機的に実行を進めるが生成された外部イベントは直ちに送出せずに溜めておき、LP 毎の Event Horizon である Local Event Horizon (LEH) に到達した時点でシミュレーション全体での Event Horizon の計算のためのリダクション演算に加わる。この計算で得られた Global Event Horizon(GEH) が自己の LEH より小さい場合は、ロールバックを行い、GEH より後のイベントの実行により生成された外部イベントは送出せず、それより前のイベントにより生成された外部イベントのみ送出を行う。よって TimeWarp における antimessage の送出がなく、ロールバックが連続的に伝搬するのを防ぐことが出来る。

SPEEDES の保守版も提案されている [30]。この場合はイベントの実行の前に GEH の計算を行い、各 LP がその時刻迄のイベントを非同期に実行することができる。

2.5.2 WEB ベースシミュレーション

World Wide Web は DES の分野でも近年脚光を浴びている。Web ベースシミュレーションに対する明確な定義は未だ与えられていないものの、目的は大別して PDES の枠組みの延長としてのモデルの並列 / 分散実行 [31, 32] と、モデルのリポジトリの分散管理 [33, 34] である。

IDES[31] は、予めシミュレーションを実行可能なサーバプログラムが起動している複数のリモートホスト上に提出して分散実行させる枠組みである。可搬性、可用性に重点を置いて構築されているが、WEB 環境では通常の並列計算機と異なり、各プロセッサの処理能力や通信ネットワークの負荷が不均一であり、さらに時間と共に変動する。このような条件の下で各プロトコルの性能アセスメントを前もって解析的に行った結果 [35]、同期プロトコルに SPEEDES[28] と同様のものを採用している。さらに、WEB 環境で SPEEDES をそのまま用いると GEH の値を得るのに時間がかかり、投機的実行が進み過ぎてロールバックによる計算コストと状態履歴の保存のためのメモリのコストが甚大になるために、先に LEH に到達した LP から順に部分的な GEH の計算に参加し、LEH に到達していない LP もこの部分的 GEH を超えない範囲で投機実行を行うという改良を加えている。

Fishwick は、URL の先頭に `model` と書くことによって WEB 上でモデルを扱うことの可能な新しいリソースロケータの記述方式を提案している [33]。既存のマルチメディアドキュメントと同様に、モデルの検索や分類が出来ることを意図している。

2.6 PDES の問題点に対する取り組み

前節迄に述べたように、十数年来の PDES の研究の結果、同期プロトコルも進化を遂げ、性能向上のための様々な技術や PDES を実現するためのプラットフォームが提案されてきた。にもかかわらず、PDES の主要な国際会議である PADS(International Workshop on Parallel And Distributed Simulation) の基調講演 [36] で、1997 年に主要な研究者のひとりである Nicol が、PDES 技術が DES の領域に実質的に進出出来ていない状況を危惧し、当該分野の生存のための提言を行っている。実はこのような危惧は以前から公に議論されていたことであり [37]、両者を比較することにより最近の数年間で解決された事柄と解決出来なかった問題、近年新たに生じた問題点を探ることが出来る。また、[37] に呼応して PDES の取るべき方向性について議論したものもあり [38]、これらは PDES の研究上の姿勢についての提言と言える。

一方、PDES の内包する技術的な避けがたい問題点とそれ等への対処法に関する議論もされている。

本節では、こうした研究姿勢と技術的側面の各問題点への取り組みについて順に紹介する。

2.6.1 PDES の実用化に向けて

Fujimoto[37, 39] は 1993 年に、「並列ミュレーションのモデルの構築が容易にならない限り、PDES の応用範囲は制限され、ひいては DES そのものの適用範囲が小規模アプリケーションへ制限されることになる。」と警告し、PDES が効果的な手法まで高められるために必要な以下の四つの「銀の弾丸 (Silver Bullets)」を示している。

アプリケーション別ライブラリ DES の応用分野毎に並列ライブラリを構築して、ユーザがプログラムを組まずに済むようにする。通信網や論理回路等に特化したモジュールを提供し、ユーザはチューニングされたこれらのモジュールを複製して相互に結合させ、シミュレーションを実行させることが出来る。

並列シミュレーション用言語 シミュレーションがアーキテクチャに依らない並列シミュレーション言語で記述出来るようになれば、プログラムの可搬性も増し、言語仕様に並列実行を効率化するための情報を与える機構を加えることにより性能が得やすくなる。

共有状態 (ベクトル) のサポート 逐次のシミュレーションに於いては全てのモデルが単一のメモリ上に存在するためメモリ参照だけで全ての状態にアクセスすることが出来るが、PDES ではサブモデル毎に状態が分割され、状態の問い合わせのためには別途外部イベント (query-event) を発行する必要がある。プラットフォーム側で共有状態のサポートが出来ればユーザはこのような手続きから解放される。

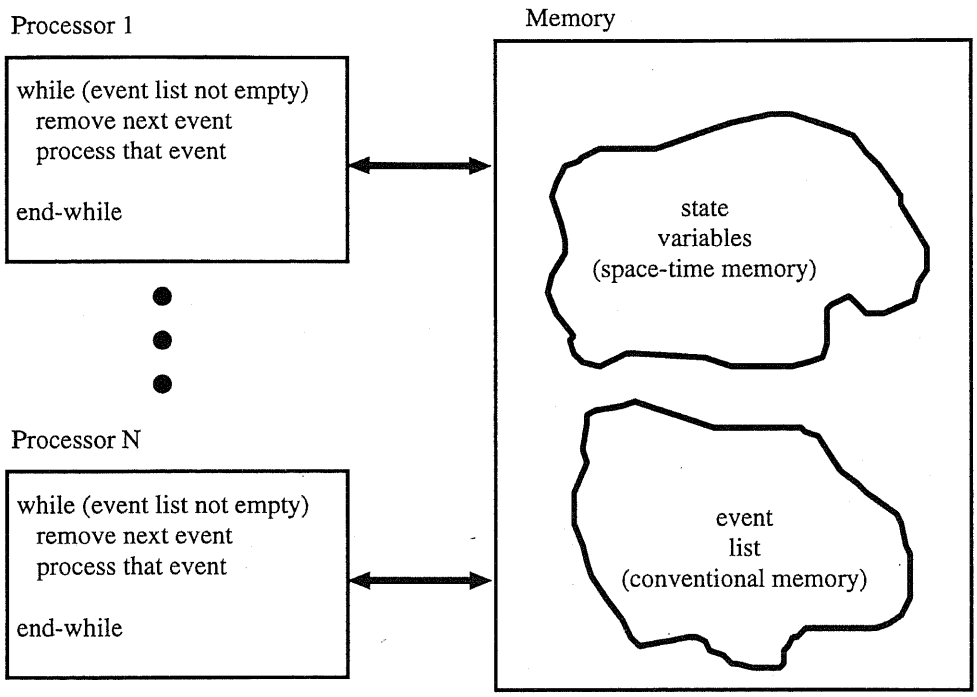

図 2.14: 逐次のイベントリストシミュレータの自動並列化。各プロセッサが同じループを実行する。

自動並列化 共有メモリ型並列計算機の場合には、図 2.14のようにイベントリストと Space-Time メモリを共有メモリ上に配置し、各プロセッサが同一のイベントループを並列に実行するようにすることで容易に実現可能である。イベントリストのボトルネックはデータ構造自体の並列化により回避可能である。但し、乱数発生部分や他の統計データ収集部も並列化されること、アプリケーション自体が並列性を含むことが必要である。

現在も残っている課題

Nicol は 1997 年の国際会議の基調講演で、PDES は依然として実用化には障壁が多いとし、さらにいくつかの提言をしている [36]。

実環境に則したモデルを使うこと PDES の研究で扱われているモデルは必ずしも実際に使われているモデルの規模や環境を正しく反映していない。現実的なアプリケーション環境で評価することが

必要である。実用レベルの規模のモデルを、控え目な並列環境、もしくは分散環境で実行すべきである。

分かりやすい用語を使うこと 状態履歴の保存、ロールバック、GVT、lookahead 等、難解な用語が使われ続けている。同期プロトコルはある程度成熟の域に達していることが、論文投稿件数の割合の減少にも見て取れる。今後はこれらの難解な概念を分かりやすい表現に置き換えていくことが必要である。

ツールを整備すること 並列モデルの構築は依然として容易になっていない。解析、デバッグ、チューニングのための洗練されたツールが待たれる。

Page[38] は、[37] での PDES の DES コミュニティに対するインパクトが小さいという指摘を受けて、PDES の軌道修正を提言している。モデルを LP に対応するサブモデルに分割するという考え方は元来並列実行上の都合から生じたものであるにもかかわらず、モデル化の段階でこのアプローチを採用することによりモデルの正確性が犠牲になり、ひいては違ったモデルを解いて間違った結果を導き兼ねないと警告している。正しい結果を導き出すためには、モデルとプログラムをきちんと分離し、自然なモデル化を行うことが肝要であると指摘している。

Nicol や Page が言及していないことが即解決済みの事柄であるということは出来ないが、共有状態変数のサポート、自動並列化技術については、その技術に適した問題に関してはある程度の成果が得られたのではないだろうか。もちろんそうでない (モデルのレベルでボトルネックを含む、並列性を含まない等) 場合はうまく行ってはいないのではないかと考えられる。

一方、Nicol の「実用レベルの規模のモデルを、控え目な並列環境、もしくは分散環境で実行すべき。」という主張からは、アプリケーション別ライブラリがうまく現実レベルのシミュレーションアプリケーションを受け止めきれていない事情が伺われる。また、「用語が分かりづらい」という指摘は、並列シミュレーション言語をもってしても未だそれらの概念を意識せずに記述出来ないことを示している。

2.6.2 技術的問題

Bagrodia[40] は、性能低下や誤った結果を招く陥穽として、共有変数の使用、ポインタを用いたデータ構造、LP 間結合トポロジにおける遅延の無いサイクル、lookahead 不足、LP 間の結合強度、負荷の不均衡、メッセージトラヒック量、イベントや計算の粒度、チェックポインティングのオーバーヘッドを挙げて対策を述べている。

Nicol[41] は Time Warp プロトコルによる同期機構を採用したシミュレーションプラットフォームに共通して想定される危険性についての警告と、それに対処するための、より安全な楽観的プロトコルを提案している。危険性は投機的実行により LP の状態と受け取る外部イベントの間の整合性が取れない過渡的な状態が生じる可能性にある。

例えば、カウンタをインクリメントする LP と、そのカウンタをリセットする LP を含む並列シミュレーションを考える。前者が後者より早く進みすぎると、カウンタのオーバーフローが生じることがある。この場合、C 言語や C++ 言語等で実装されていると記憶領域アクセス例外等の深刻で回復不能なエラーを招く恐れがある。

このような問題を一般的に避けることは不可能であり、いくぶん保守的な同期戦略を採り入れる以外にない。例えば、全ての送出外部イベントに対して確認応答 (ACK) をもらう迄は新たな外部イベントを送出せず、投機的実行の影響を LP 内にとどめるという戦略を取ることでこうした危険性が回

避できることを示している。このようなより安全なプロトコルを採用することにより性能が犠牲になるが、これは不可避である。

2.7 まとめ

本章では、DES、PDES の概要について述べ、PDES の研究の最近の動向について、同期プロトコルの発展、Internet 環境の整備に触発された Web ベースシミュレーションの動向、シミュレーション言語やライブラリの現状について順に述べ、PDES の技術の普及のために研究者がどのような問題意識を持って研究に臨んできたかを概観した。プロトコルに関する研究は成熟段階だと言われるが、Web ベースシミュレーション等の新たな環境に適応するプロトコルがこれからまだ考案れさ続けると思われる。またプラットフォームも一定の成果は出しているものの、なお記述の煩雑さ等の問題を抱えていると言えよう。

第 3 章

タイムワープ向け FIFO キューのクラスライブラリの実装と評価

LP 間の楽観的同期方式のひとつであるタイムワープは、LP 間の同期を緩め、因果関係の不整合が生じた時点でその回復処理を行うもので、楽観性をうまく制御することによりモデルに内在する並列性を十分に引き出す可能性を持つとされている。

回復処理を実現するためには、LP 毎に内部状態の履歴を保存しておく必要がある。状態管理方式としては、コピーに基づくもの (Copy State Saving;**CSS**) と、チェックポイント間の状態の差分に基づくもの (Incremental State Saving;**ISS**) が提案されている。いずれの方式も多くの汎用 PDES プラットフォームに組み込まれ、自動化されている。

CSS 方式では、ユーザは保存したい状態を状態変数として申告しておくと、プラットフォームは適当なタイミングでそれ等のコピーを保存し、ロールバック時にはロールバック時点の直前の履歴を状態ベクタに書き戻すことにより状態の回復を行う。

ISS 方式では、ユーザは状態の変化を取り消す操作を手続きとして申告する。更に実行時には状態の変更を行う際取り消し操作の識別子を申告する。プラットフォーム側では、ロールバックの際その識別子を基に取り消し操作を逐次的に呼び出し、状態を回復する。

通信、計算機システムで多用される待ち行列シミュレーションでは、状態として待ち行列データ構造 (キュー) を管理する必要がある。キューは一般にはリンクリストを用いた動的なものであり、CSS 方式では直接管理出来ない。一方 ISS 方式を用いると管理が可能になるが、取り消し操作の記述も容易ではなく、記述法もプラットフォームにより異なるのが現状である。また、ロールバック距離に比例した回復コストを要するという問題もある。

待ち行列シミュレーションで特に良く用いられる先入れ先出しキュー (FIFO キュー) は一旦挿入された要素間の順序変更は起らないため、その性質を利用した最適化が可能である。CSS 方式はユーザに対する負担も軽く、FIFO キューが通常の埋め込みデータ構造と同様に扱うことが出来るならば、ISS 方式を用いずに FIFO キューを管理することが可能になる。

本章では、FIFO キューを CSS に基づくプラットフォームで状態変数として実現する、平易なインタフェースを持つ C++ のクラスライブラリの提案と性能評価を行う。更に多段接続交換網シミュレーションに対する組み込みを行うことにより、実アプリケーションへの適用の容易性と有効性を示す。

3.1 はじめに

本章では、タイムワープ方式に基づく並列離散事象シミュレーションにおける待ち行列システムシミュレーションに用いる、高性能で利用が容易なインタフェースを備えた FIFO キューのクラスライ

ブラリの実装と用例、性能評価について述べる。汎用のシミュレーションライブラリや言語処理系はいくつか提案されており、その中でもコピーに基づく状態管理を行うプラットフォームでは、ユーザは状態ベクタを定義するだけで済み、ロールバック等に関知する必要がない利点がある。しかし、状態ベクタは埋め込みデータ構造として定義されなければならないため、FIFO キューのような動的なデータ構造を直接扱うことが出来ない。提案するクラスライブラリはプラットフォームとユーザの双方が埋め込みデータ構造と同様の扱いをすることが出来る。また、上記のデータ構造の先入れ先出しの性質を生かして格納されているデータ全てではなく操作履歴が事実上の最小コストで効率よく保存および回復される。カーネルがシミュレーション時間上の過去の状態履歴を削除する際には、対応するメソッドによるガーベジコレクションが透過的に行われる。

3.2 背景

楽観的プロトコルのひとつであるタイムワープ方式では、シミュレーション時刻に沿ってロールバックするために状態の保存が必要である。しかし、状態保存やロールバックの処理は複雑でアプリケーションプログラマが個別に誤りなしに実装するのは困難である。

このような機構を自動的に行うライブラリや言語はいくつか存在するが、状態の管理方式で分類するとコピーに基づく状態保存 (Copy State Saving; 以下 CSS) を行うもの [17, 42, 22, 15] と、操作の保存に基づくもの (漸進的状態保存 (Incremental State Saving; 以下 ISS)) [43] に大別される。

CSS 方式では、ユーザは状態ベクタを定義するだけで済み、ロールバック等に関知する必要がない利点がある。しかし、状態ベクタは埋め込みデータ構造として定義されなければならないため、FIFO キューのような動的なデータ構造を直接扱うことが出来ない。

本章で提案するクラスライブラリはプラットフォームとユーザの双方が埋め込みデータ構造と同様の扱いをすることが出来る。また、上記のデータ構造の先入れ先出しの性質を生かして格納されているデータ全てではなく操作履歴が事実上の最小コストで効率よく保存および回復される。カーネルが過去の状態履歴を削除する際には、対応するメソッドによるガーベジコレクションが透過的に行われる。

本章では、以下 3.3節で既存の方式およびその問題点、3.4節で提案するクラスライブラリについて、その概要、設計、実装および API(Application Program Interface) を述べる。待ち行列ネットワークシミュレーションへの適用例については 3.5節で述べる。次に 3.6節で埋め込みデータ方式による実装との比較に基づく性能評価を行い、最後に 3.7節でまとめを行う。

3.3 既存の方式およびその問題点

CSS 方式では、ユーザはカーネルに対して保存をして欲しい状態を宣言するだけで済むためユーザの負担は軽い反面、状態ベクタとして固定長のデータ構造 (整数、浮動小数点数等の基本データ構造およびそれら組み合わせ) しか指定することができない。このようなデータ構造は待ち行列システムシミュレーションにおける待ち行列データのような動的なデータ構造の表現には不向きである。

もっとも、埋め込みデータ構造を用いて待ち行列データ構造を表現する方法もいくつか存在する。例えば、待ち行列のコンテナのひとつひとつに LP を割り当てることが考えられる (図 3.1)。このような手法はハードウェアシミュレーションにおける待ち行列データ構造の詳細な振る舞いの解析には有用であると考えられる。しかし、キューを FIFO バッファとしてより抽象的に扱いたい場合には、上記の手法は複雑で非効率的である。さらに、PDES では LP 間で共通の大域的な状態を用いることが出来ないため、次の enqueue および dequeue の対象となる LP はどれかといった情報を伝えるために別途外部イベントメッセージの発行が必要となってしまう。

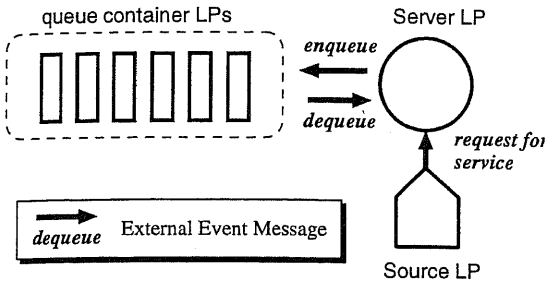

図 3.1: FIFO キューの実装方式 (a) 各コンテナをひとつの LP に対応づける

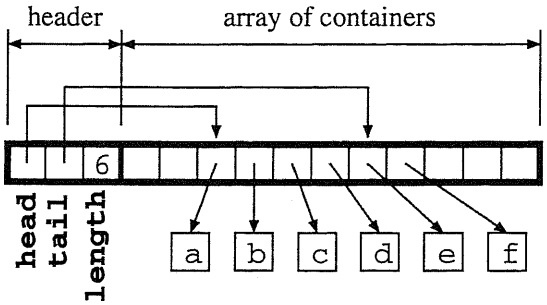

図 3.2: FIFO キューの実装方式 (b) コンテナの配列でリングバッファを構成する

埋め込みデータ構造を用いて状態ベクタの要素としての待ち行列データ構造を実装する別の方法としては、長さが挿入される要素数の最大値に等しいようなコンテナ (ポインタ) の配列を用いることである (図 3.2)。この手法は容易に実装可能であるが、メモリ消費量およびそれに伴う各状態保存のコストが常に配列長に比例してかかってしまう。

さらに、各履歴の間でどの要素が dequeue されたかに関する情報を別途管理しない限り正しくガーベジコレクションを行うことが出来ない。

また、上記の 2 手法はいずれもあらかじめ挿入される要素の最大値を決定しなければならない。このような制約は、ネットワーク設計において所望の損失率の上限を達成するためにはバッファ長を最大どれくらいにすべきかをシミュレーションによって決定したい場合等には煩わしいものとなる。このようなシミュレーションでは、瞬間最大待ち行列長が興味の対象となる場合も有り得る。

このような問題は SPEEDES[43] で採用されているような ISS 機構によって解決することが出来る。ISS を用いると、待ち行列データのような、状態変更の前後に共通部分の多い冗長なデータ構造の状態管理を行う際に、全ての要素自身を保存する必要がなくなる。

SPEEDES でもロールバックが自動的に行われるが、状態キューとユーザの定義した状態ベクタの間でコピーを行う CSS 方式の WARPED[17] 等に対して、状態キューにユーザの定義した取り消し操作を保存しておき、その操作を逆順に適用するという点が異なる。

状態ベクタに対する取り消し可能な操作はすべて ISS によって管理することが出来る。しかし、FIFO キューを ISS で実装する場合、先入れ先出しの性質がうまく生かされず、状態の回復の際には連続する操作が逐一取り消される必要があるため、ロールバック距離に比例したコストがかかってしまう。また、ユーザは取り消し操作の定義を別途カーネルに与えなければならず、その記述方法もカーネルによりまちまちである。

本章で提案する手法では、状態履歴を破棄するコストを除いてロールバック時の状態回復自体のコストは CSS 方式と同様、ロールバック距離に依存しない。カーネルは状態履歴からロールバック後の状態を取り出してユーザにアクセス可能にさせるだけで済むからである。

本クラスライブラリを WARPED 等の CSS に基づくカーネルに組み込むことによって、別の最適化の機会も生かすことができる。カーネルは場合によっては最適化の一環としてイベント実行毎ではなく間欠的に状態保存を行う (Infrequent State Saving, Periodic State saving; 以下 PSS) こともあり得る。本手法は待ち行列への挿入、削除各操作に対してスナップショットを取る必要はなく、複数の操作が単一インスタンスで表現されるため、状態保存の頻度にかかわらず適用可能である。

3.4 提案するクラスライブラリ

3.4.1 クラスライブラリの概要

FIFO キューでは、プライオリティキュー等と異なり一旦挿入した要素間の順序変更は生じないため、差分に基づく状態保存を行うのは容易である。スナップショット間で共通する要素 (直前のスナップショットより前に挿入されて現在のスナップショットまでに削除されていない要素) はポインタを介して共有される。これらの機構は C++ クラスライブラリにカプセル化され、アプリケーションプログラマとシミュレーションカーネルに対しては透過的である。本クラスのインスタンスはそれぞれ構築された時点でのキューのスナップショットを表現するため、以後このクラスライブラリパッケージを *SsQueue* (*S*nap *s*hot *Queue*) と呼ぶ。SsQueue は状態保存と回復に関して、挿入された要素数によらず事実上最小のオーバーヘッド (各インスタンスは 5 個のポインタと 1 個の整数より成る) を達成するだけでなく、C++ をベースとするタイムワープ方式の汎用シミュレーションプラットフォーム間の移植性をも達成することができる。

3.4.2 クラスライブラリの実装

本章でははじめにプログラマの観点での API について、次にシミュレーションカーネル側から見たインタフェースについて、最後に内部表現について述べる。

ユーザ側からの API

ユーザ側の API は以下のように単純である。

- `void enqueue(void *item)`

  item を挿入する。

- `void *dequeue()`

  先頭の要素を削除し、その要素を返す。

- `void *top()`

  先頭の要素を返すが削除はしない。

- `int getLength()`

  現在の要素数を返す。

カーネル側のインタフェース

CSS 方式のシミュレーションカーネルは現在の状態を状態キューにコピーすることにより状態保存を実現している。ロールバックの際には状態キューにおけるロールバック時点の直前の状態が状態変数に書き戻される。SsQueue は代入演算子をオーバーロードすることによりそれぞれの操作を捕捉する。更に、カーネルがガーベジコレクションを行う際には、SsQueue のデストラクタが自身のガーベジコレクションを行う。

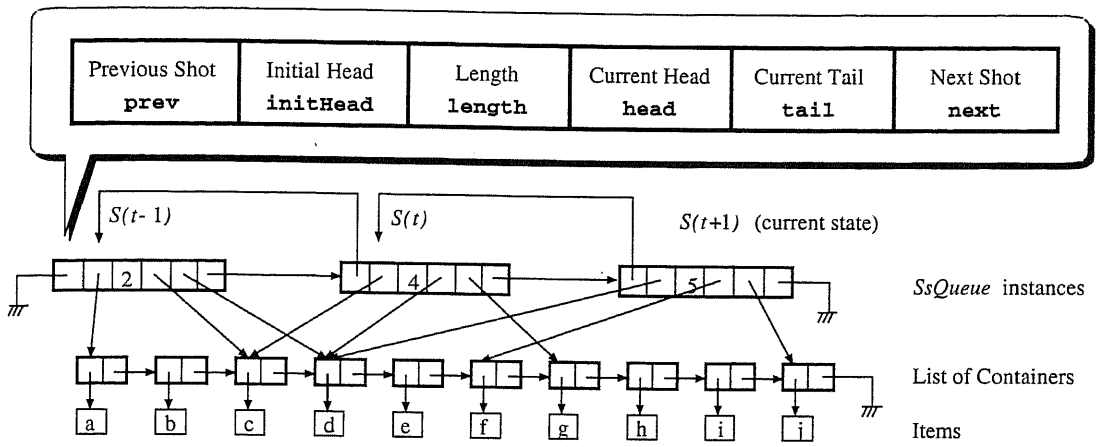

図 3.3: SsQueue インスタンス間の時間的つながりと隣接するスナップショット間の要素の共有

内部表現

SsQueue における隣接するスナップショット間の時間的つながりおよび要素の共有の様子を図 3.3に示す。現在の状態 (図の $S(t+1)$) に対する挿入および削除操作の適用は単純である。要素の挿入時には、対応するコンテナのリスト構造の最後尾に追加され、`tail` ポインタは、そのコンテナを指すように変更される。要素の削除時には、ポインタ `head` が差していたコンテナの保持していた要素がユーザに返され、`head` ポインタは次のコンテナ (図中では'g' を指すコンテナ) へ進められる。削除された要素やそれ等を保持していたコンテナはすぐには削除されない。以下に述べるようにそれらの削除はカーネルが対応するスナップショットを削除するまで延期される。

スナップショット間の時間的つながりはカーネルが状態の保存および回復を行う際に確立される。オーバーロードされた代入演算子がこれらの代入演算を捕捉する。状態の保存の際は、コピー元が現在の状態、コピー先が状態キュー中のスナップショットといった組み合わせで代入演算子が呼ばれる。この場合、SsQueue はこの新しいスナップショットを現在の状態を表現するインスタンスの直前にリンクさせる (つまり、新しいスナップショットの `next` ポインタは現在の状態を表現するインスタンスを指し、現在のインスタンスの `prev` ポインタは新しいスナップショットを指す)。ロールバック時にはロールバック時点の直前のスナップショットを現在の状態変数へ書き戻す操作がカーネルにより行われる。SsQueue は、今度は代入演算子がコピー元を状態キューの中のスナップショットとして呼び出されることによりこの操作を検出する。この時 SsQueue は破棄されたスナップショットと要素を無効化し (これらは実際に削除される前に再利用することも可能である)、回復された状態を新しい現在の状態の直前にリンクさせる。

fossil collection はカーネルがコミットされた状態を削除する際に起動される。図 3.3の $S(t-1)$ を例に取ると、`inithead` が指しているコンテナから `head` の指しているコンテナの直前までが削除されなければならない。図の例では'a' と'b' が削除される。

これらの機構はシミュレーションカーネルおよびユーザの両者にとって透過的であるため、前者に対する変更は不要であり、後者に対しては、ロールバックや状態の回復の管理を強いることはない。各インスタンスは 5 個のポインタ (`prev`, `next`, `inithead`, `head`, `tail`) と 1 個の整数 (`length`) のみ必要とし、状態の保存と回復のコストは充分に小さい。状態の回復の際は、いずれのスナップショットにもそのまま続けて挿入、削除の操作を適用することが出来るため、最小のオーバーヘッドを達成している。

3.5 SsQueue の応用例

本章では大規模待ち行列ネットワークシミュレーションの一例としての Banyan Switch シミュレーションへの適用例について述べる。Banyan Switch は高速広帯域ネットワークの交換機に用いられる多段構成のセルフルーチング交換網である。図 3.4に 3 段構成の Banyan Switch の構成を示す。角の丸い四角形で囲まれた 2×2 の単位スイッチは到着した ATM セルのヘッダを読み取り、出力ポートを選択する。セルが既に使用中の出力ポートを指定している場合はバッファリングされる。この操作が全て並列に動作することにより、8 個の入力ポートから 8 個の出力ポートへの ATM セルのポイント・ツー・ポイントのルーチングを行う 8 × 8 スイッチが実現している。

シミュレーションプログラムは WARPED[17] 上に構築した。WARPED は C++ で記述されたパブリック・ドメインのタイムワープシミュレーションライブラリである。WARPED では、ユーザは LP を表現する基本クラス[1]を継承することにより個々の LP を定義し、そこでカーネルによって呼び出されるイベント処理メソッドを再定義する。LP の基本クラスは状態ベクタクラスを引数とするテンプレートクラスとして提供されている。

Banyan Switch のシミュレーションのためにここでは 4 種類の LP を定義した。入力トラヒックを生成する `CellSrcObj`、2 × 2 ルータをシミュレートする `CellRouterObj`、単位スイッチにおける出力バッファおよびポートを表現する `CellOutBufObj`、そして各出力ポート `CellSinkObj` におけるセルの統計情報を収集する `CellSinkObj` である。図 3.5に `CellOutBufObj` のユーザプログラムに対する SsQueue クラスの組み込みの様子を示す。

2 行目から 6 行目にかけて状態ベクタの宣言が与えられている。SsQueue はその状態ベクタの一部として含まれている (4 行目)。9 行目から 15 行目にかけて、この状態ベクタを用いた LP が定義されている。18 行目から 37 行目にかけて `CellOutBufObj` におけるセル到着の扱いの様子が示されている。出力ポートが使用中の場合は、セルは SsQueue のインスタンスに格納される (24 行目から 33 行目)。

シミュレーションプログラム全体としての性能評価結果として、50,000 μ 秒のシミュレーションにかかった実時間を図 3.6に示す。実験は分散メモリ型並列計算機 SR2201[26] の 1 プロセッサから 32 プロセッサを用いて行った。

比較的良い速度向上が、特に単位スイッチ数が多い場合に得られていることが分かる。

3.6 性能評価

3.6.1 性能解析モデル

状態管理方式の性能解析に関してはいくつか研究がなされている [44]。ここでは [45] で用いられるモデルに基づいて議論する。

文献 [45] では、PSS のオーバーヘッドを

$$O_p = k_r \left\{ [\frac{1+(\chi-1)p}{p\chi}]\delta_s + [\frac{\chi-1}{2}]\delta_e \right\} + \frac{N}{\chi}\delta_s \tag{3.1}$$

とモデル化している。但し、k_r は平均ロールバック数、χ は状態保存間隔,p はロールバック距離が 1 である確率,δ_s は 1 回の状態保存のコスト,δ_e は 1 イベントの平均実行コスト,N は実際に実行されるべきイベント数である。簡単のために、$\chi = 1$ とおくと、通常の CSS の式

$$O_p = k_r\gamma\delta_s + N\delta_s \tag{3.2}$$

[1] warped では、LP という用語はプロセッサ毎に一つ設けられる複数のシミュレーションオブジェクトをとりまとめる主体に対して用いられる。ここでは慣例に従ってこれらの複数のオブジェクトに対して LP という語を用いることにする。

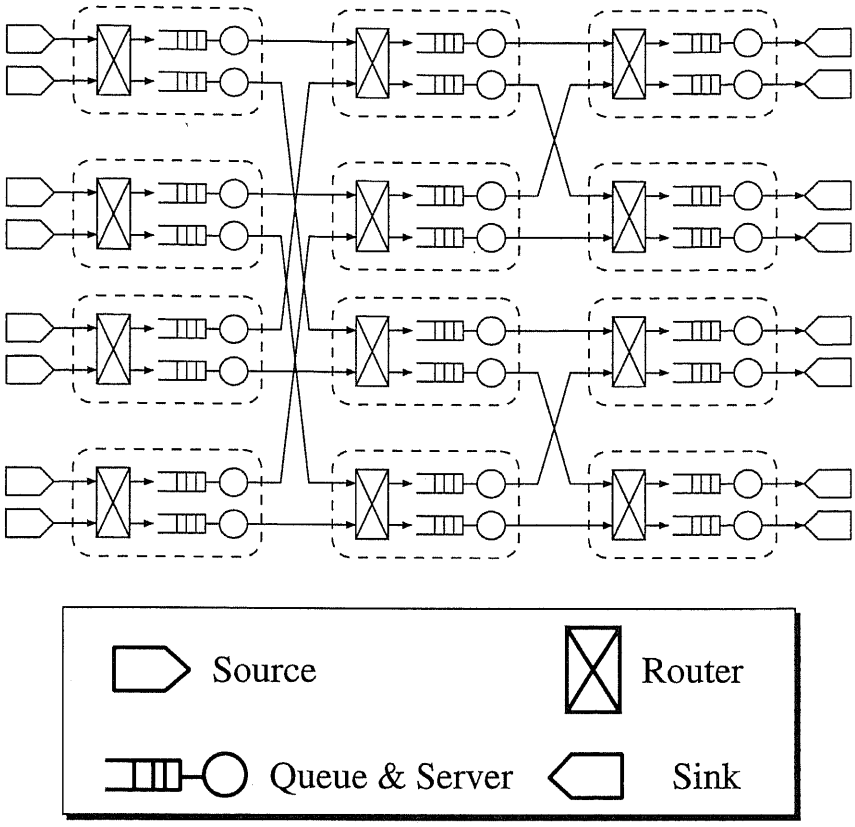

図 3.4: 3 段構成の Banyan Switch に対応する LP の構成

となる。但し $\gamma = 1/p$ は平均ロールバック長である。第一項は誤った処理に関する状態履歴の保存によるオーバーヘッド、第二項は正しい実行に関する状態保存のオーバーヘッドである。

CSS 方式では、状態の破棄に関するコストは無視されているか、δ_s に含まれていると考えられる。

SsQueue による状態の管理コストは、 SsQueue のインスタンスの大きさの埋め込みデータを CSS 方式で管理するコストより小さくはならない。インスタンスコピーそのもののコストは当ライブラリでは 5 個のポインタと 1 個の整数の領域のコピーに必要なコストに等しい。

以下、状態管理の各段階でのコストについて議論する。

状態保存 SsQueue インスタンス同士のリンクのためのポインタの繋ぎかえの操作のみが行われ、そのコストは現在の待ち行列長に依存しない。

ロールバックによる状態回復 カーネル側で書き戻すべき状態を状態バッファから検索する必要があるために、既にカーネル側でロールバック距離に依存した検索のコストがかかっている。本ライブラリ側では、該当するインスタンスを書き戻すだけで良いので保持している要素の数には依らない。

ロールバックによる不要な状態履歴の破棄 カーネル側では、対応するインスタンスの占めている領域の解放を行うため、ロールバック距離に比例したコストを要している。SsQueue 側では、誤って保存されたインスタンスの無効化を行うため、同程度のオーダーのコストがかかるが、カーネルが消費するコストのオーダを超えることはない。

```
 1:// State variables definition
 2:class CellOutBufState : public BasicState {
 3:public:
 4:  SsQueue ssq;  // SsQueue instance
 5:  int isBusy;      // 1 if server is busy
 6:};
 7:
 8:// LP definition
 9:class CellOutBufObj : public SimulationObj
10:< CellOutBufState > {
11:public:
12:  // user-defined event handler
13:  void executeProcess();
14:  ...
15:};
16:
17:// User defined event handler
18:void CellOutBufObj::executeProcess() {
19:  // next event to execute
20:  CellEvent *event = (CellEvent *)getEvent();
21:  switch(event->type) {
22:  case ARRIVECELL: // Arrival of ATM cell
23:     SsQueue *ssq = &(state.current->ssq);
24:     if (!(state.current->isBusy)
25:     && ssq->isEmpty()) {
26:        // cell is forwarded immediately
27:        // without queuing
28:     } else { // buffer this cell
29:        CellInfo *cell = new CellInfo;
30:        *cell = ((ArriveCellEvt *)ace)->cell;
31:        // cell is buffered into SsQueue
32:        ssq->enqueue(cell);
33:     }
34:     break;
35:     ...
36:  }
37:}
```

図 3.5: SsQueue を用いたコーディング例

また、ロールバック時に不要となったコンテナ (誤って enqueue されたものを保持しているリスト構造) の回収は SsQueue ではただちには行われない。後の enqueue 操作で挿入される要素のために再利用される。よって、ロールバック時の状態の無効化にはキュー長に比例したコストを必要としない。

fossil collection dequeue されたデータを実際に消去する操作が行われる。1 ショットの消去あたり、その直前のショット以降に dequeue された要素の数に比例したコストがかかるが、通常の用途ではショット数と操作数は比例関係にあると考えられるため (1 事象処理中に挿入される要素の数は通常たかだか一つ)、その時点で保持している要素数に比例したコストは必要とはならない。

以上の議論から、SsQueue の管理コストも式 3.2と同じ表現になると言える。

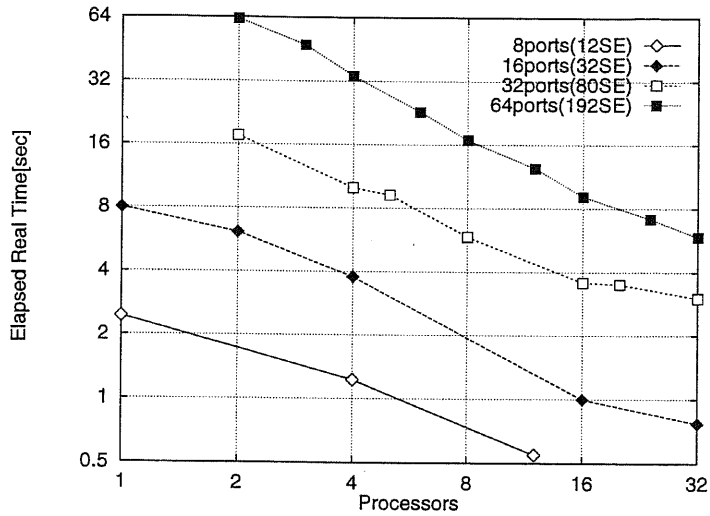

図 3.6: 50,000 μ 秒のシミュレーションにかかった実時間

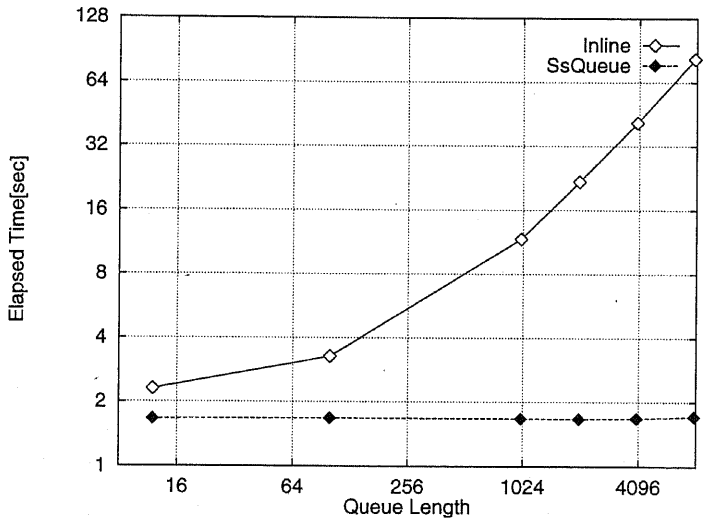

図 3.7: 待ち行列長に関する埋め込みデータ方式との性能比較

3.6.2 評価結果

先にのべた図 3.2の代替方式 (b) である埋め込みデータ構造による実装と比較した。状態管理動作を模擬するプログラムに組み込み、決まった長さのロールバックを繰り返す操作を反復させ、ロールバック長、定常待ち行列長等を変化させて実行時間を測定した。代替方式における代入演算子は、デフォルトのものを用いると全スロットのデフォルトコンストラクタの呼び出しになり効率が悪いので、OS のメモリコピールーチンを直接呼ぶような手続きで上書きした。

図 3.7はロールバックが正常実行の 40% の場合での、要素数を変化させた場合の実行時間である。埋め込みデータ型方式では要素数に応じてリングバッファの長さを変化させた。埋め込みデータ構造に基づく方法ではバッファ長を可変にすることが出来ないため、有限バッファ長のシミュレーションを想定している。

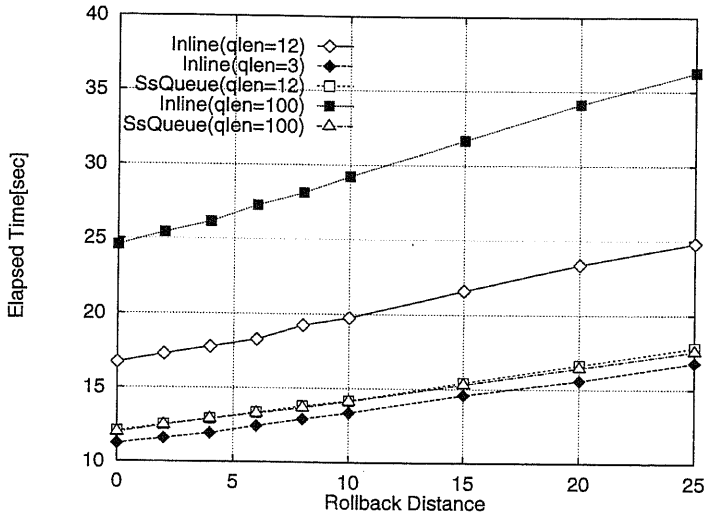

図 3.8: ロールバック長に関する埋め込みデータ方式との性能比較

埋め込みデータ構造による実装では、ロールバック時や fossil collection 時のオーバーヘッドが少ないものの、FIFO キューの最大長に比例したメモリ量を消費するため、要素数が大きくなるにつれてオーバーヘッドが増大していることがわかる。SsQueue のスナップショットの大きさは 4×6 バイトであることを考慮すると、インスタンス分だけの大きさを占める埋め込みデータ型の場合と殆んど遜色がない性能が得られていることが分かる。

図 3.8に、要素数 12、100 の場合においてロールバックの割合を 0% から 50%(ロールバック長 0 ～ 25) に変化させた場合の実行時間を示す。埋め込みデータ方式については要素数 3 の場合の測定結果も併せて示した。y 切片は式 3.2の第二項を示している。傾きが、要素数の十分小さい埋め込みデータ型と同程度であることから、ロールバック時の状態の無効化のコストも十分に小さいことが分かる。要素数が大きい場合も性能は低下せず、ロールバック長が大きくなるに従い埋め込みデータ型に対する優位性が大きくなることに注意されたい。

これ等の測定結果から、SsQueue 固有のロールバックや fossil collection に関するコストは含まれる要素数によらず、メモリコピーのオーバーヘッドに比較して十分に小さいことが実証された。

3.7 まとめ

本節では、タイムワープシミュレーションのための FIFO キューの、高性能で使いやすいクラスライブラリ実装である SsQueue について述べた。本クラスの各インスタンスの占めるメモリ量は充分小さいため、状態保存に必要なコストはキュー内に挿入されている要素数によらず小さく抑えられる。隣接するスナップショット間で要素が適切に共有されているために、ガーベジコレクションのコストも可能な限り最小限に抑えられている。

SsQueue は汎用タイムワープシミュレーションカーネルのベンダとアプリケーションプログラマの仲立ちをすると考えることが出来る。シミュレーションカーネル側は楽観的同期機構をサポートしなければならないが、論理プロセスの状態変数を埋め込みデータとして扱うことが出来る。SsQueue を用いることにより、ロールバックや状態保存を明示的に実装したくないシミュレーションプログラマにとっても、そのようなシミュレーションカーネルを用いて待ち行列データ構造を論理プロセスに組み込むことが出来る。

Banyan Switch シミュレーションへの適用例では、シミュレーションプログラムへの組み込みの容易性と、実アプリケーションへ充分適用可能であることを併せて示すことが出来た。

性能評価の結果、管理コストが含まれる要素数に依存せず十分小さく抑えられること、ロールバックに関するコストが埋め込みデータ型を用いた場合と殆ど変らず小さいことが実証された。

FIFO キュー以外の任意のデータ構造を CSS プラットフォーム上で支援する枠組みについては、次章で述べる。

第 4 章

コピーに基づく状態管理を行うタイムワープ方式プラットフォームにおける漸進的状態保存の支援

本論文では前章で CSS を行うタイムワープシミュレーションプラットフォームで FIFO キューを扱う枠組みである SsQueue について提案を行った。しかし、FIFO キュー以外の動的なデータ構造を扱うことができない。

本章では、上記のクラスライブラリの枠組みから、後で述べる CSS に基づく状態管理の四つの基本的な操作の抽出方法を明確に切出し、別のデータ構造への適用への道すじを示す。前章で提案したクラスライブラリである SsQueue では、queue そのものの構造もカプセル化されていて、単純な操作 (queue/dequeue) しか出来ず、キューの中身に立ち入った操作 (内容の表示等) が出来ない。本提案方式を用いて手続きに基づく ISS を用いて FIFO キューを管理することにより、状態回復のコスト自体は大きくなり効率は落ちるかも知れないが queue の中身に自由にアクセス出来るようになる。

更に、FIFO キュー以外の任意のデータ構造の ISS をサポートを枠組みを提案し、前回の提案をその特殊な場合という立場から改めて FIFO/LIFO キューでの最適化について言及する。

4.1 背景と提案方式の概要

前章までで、CSS/ 間欠的状態保存 (Periodic State Saving;**PSS**) の枠組みの下でも動的なデータ構造である FIFO キューを状態として透過的に扱うことが出来るようになった。しかし、FIFO キュー以外の動的なデータ構造については扱うことが出来ない。

本章では、前章の枠組みを一般化するために、CSS/PSS プラットフォームが行う (1) 状態保存、(2) ロールバック / 状態回復、(3)fossil collection、(4) 誤った処理に関するの履歴の無効化 / 消去 (State Invalidation)、という四つの処理を受動的に検出し、動的なデータ構造のための ISS 方式に対応する処理をプラットフォームから透過的に行うようなクラスライブラリを、ユーザデータ構造とプラットフォームの間に挿入することにより実現した。

具体的には、履歴間の時間的つながりを維持するために C++ の演算子やメソッドの再定義の機構を利用して、代入演算子、デストラクタに対して、それぞれ前回の履歴とのリンク、アンリンクの働きを担当させた。

その結果、所望の機能を持つクラスを、個々のインスタンスを十分小さく抑え (筆者の計算機環境で 12 バイト) て実現出来た。ユーザ側とのインタフェースの切り分けも明確であり、一般的な動的データ構造の ISS に対する設計指針を示すことが出来た。

また、FIFO/LIFO キューのような、任意の時点での構造が、別の時点での構造を含むような特別な性質をもつ線形的なデータ構造の場合はこの枠組みをそのような構造に特化させることによりロールバック時のオーバーヘッド抑制出来ることを示し、ライブラリ側、ユーザ側双方から、通常の固定

長のデータ構造と同一の扱い (代入演算子でスナップショットが取れる等) が出来、容量に制限のない動的データ構造として実現出来ることを示した。

タイムワープの状態管理を意識しないで作成された動的な構造を持つデータ構造と、そのようなデータ構造の扱いを想定していないプラットフォームの間に立って正しく状態管理を行うクラスライブラリを実現したことにより、CSS/PSS 方式のタイムワープシミュレーションプラットフォーム上でシミュレーションを行うユーザ全てが ISS 方式を利用可能になる。本パッケージと動的データ構造、それに対応する DO/UNDO 手続きの定義をセットにして配布することにより、ユーザ、ライブラリ双方から見て通常の静的なデータ構造 (整数や固定長の文字列等) と同様の状態変数としてそれらのデータ構造を扱うことが可能になる。

4.2 動的データ構造と ISS

タイムワープの状態管理の方式は、大別すると CSS/PSS 方式と ISS 方式に分けられる。CSS/PSS 方式は実現も簡単であり、この方式をサポートしたプラットフォームは多い [20, 15, 17]。

しかし前節で述べたように FIFO キューのような動的なデータ構造を CSS では直接扱うことが出来ず、一般には ISS 方式による管理が必要になる。

ISS という呼称は文献により少し違った意味合いで定義されている。

(1) 通常の CSS は状態の変化如何にかかわらずイベントの実行毎に状態のコピーを行うが、それに対して変更された時だけコピーすること (save-if-modified)[20]

(2) 状態ベクタのうち、変更された部分だけをコピーすること (partial-state-saving) [20, 45]

(3) 状態の変更そのものの情報を保存しておき、ロールバック時にその情報に応じて時間順の逆順に取り消し操作を行う (do/undo-state-saving)[28]。

(1) と (3) は、保存に必要なメモリ量が実際に加えられた変更のみに依存するため無駄がないと言える。

(2) は変更された部分だけ保存することにより更にメモリ消費量を抑えることが可能であることを主張するものである。

上記を更に比較すると、コピーに基づく方法である (1)、(2) と、手続きに基づく方法である (3) に大別される。(3) は実現は若干複雑になるが、(1) に対しては変更手続きとしてコピーする手続き、(2) に対しては変更手続きとして変更された部分だけコピーする手続きとして登録することによって本質的に (1) と (2) を肩代わりすることが出来ると言える。逆に (1) と (2) は、固定長のフラットな埋め込み型データ構造を想定しており、複雑な参照関係を含むデータ構造を扱うことが出来ない。本章では、以降 (3) を ISS と呼ぶことにする。

ISS をサポートするプラットフォームもいくつか存在する [28, 46]。これらは、それぞれユーザが状態の回復のための手続きを与えてやらなければならないが、その与え方はシステムによりまちまちである。

楽観的プロトコルプラットフォームにおける状態管理は、CSS 方式と ISS 方式に共通する部分と、方式毎に異る部分に分けることが可能である。前者は時刻同期の機構と連携する部分である。時刻同期部分は、ロールバック時刻に対応する状態の選定、GVT の進捗になり不要になった状態履歴の選定、ロールバックにより無効になった状態履歴の選定を行う。

前者を整理すると、以下の四つの基本的な動作にまとめられる。

1. 状態保存

2. 状態回復

3. ロールバック

4. fossil collection または誤った処理に関する履歴の無効化 / 消去

4.3 本提案方式の位置づけ

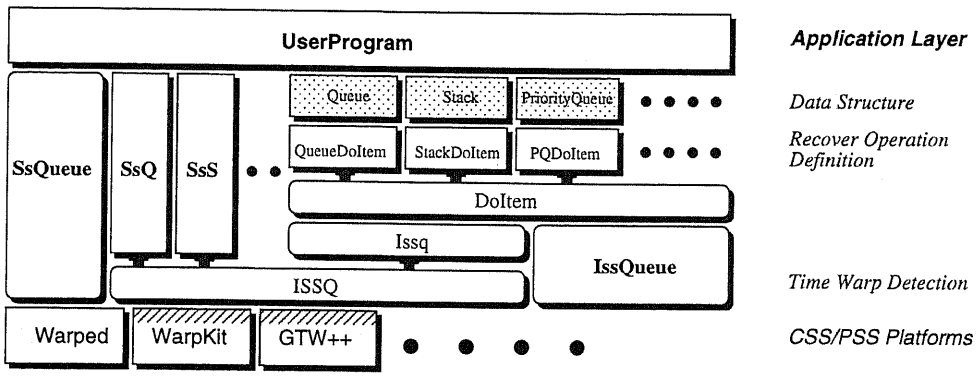

図 4.1: IssQueue のアーキテクチャ

本提案方式は図 4.1のような階層構造を構成する複数のパッケージで実現されている。主なパッケージは CSS カーネルからタイムワープの同期処理を検出する ISSQ、CSS から ISS への変換を行う IssQueue、取り消し手続きの定義の枠組みを与える DoItem である。図中の上向きの矢印はクラスの継承関係を表す。図中の DoItem 階層のある右側半分は手続きに基づく ISS の実現である。通常のデータ構造に対する取り消し手続きの定義を IssQueue クラスに与えることで、CSS カーネル上の ISS を実現している。図中の左半分はロールバックの際に手続きの適用を伴わない ISS の実現である。先に提案した FIFO キューの状態管理方式は、ISSQ と、FIFO キューのための特殊なロールバックや状態回復の実現 (SsQ) を組み合わせたものと言える。FIFO キュー以外にも、LIFO キューのような、任意の時点での構造が、別の時点での構造を含むような特別な性質をもつ線形的なデータ構造の場合は同様に ISSQ と組み合わせることによってロールバック時のオーバーヘッドを抑制出来る。

シミュレーションに必要な仮想時刻 (Virtual Time,Simulation Clock; 以下 VT) の実現はシミュレーションの種類や時間軸上のスケールにより異なる。本来ロールバックは **straggler**(宛先の LVT より小さなタイムスタンプを持つ外部イベント) のシミュレーション時刻へ向って、また fossil collection は GVT の進捗に沿って行われるので、VT の比較処理が必要である。本提案方式では VT に関連する部分はカーネルにより隠蔽されているため状態理部分は VT と独立に動作することが出来る。

4.4 本提案方式の概要

本章では、CSS/PSS のみをサポートするタイムワープ型プラットフォーム上で、ISS をサポートするための枠組みを提案する。本章で実際に実装 / 検証されたライブラリは C++ で記述されているが、今日のタイムワープシミュレーションカーネルの多くは C++([17, 21, 23]) もしくは似たような枠組みのオブジェクト指向的な実現が多いことから妥当なものと考える。また、この枠組みは他のオブジェクト指向的な言語へ容易に適用可能である。

本提案手法を用いると、多くの CSS/PSS シミュレーションライブラリに同一の ISS インタフェースを持たせることが可能となる。

本手法では上記の四つの処理を行うタイミングは全てプラットフォームに委ねることが出来るため、PSS における状態保存間隔の違いや、fossil collection におけるインスタンスの削除の順序の違い (カーネルは折角 CSS で保存した (未コミットの) 履歴の一部を、メモリ不足のために間引くかも知れない。CSS ならば単純に間引いて構わないが、ISS の場合は間引くと状態の回復が不可能になる)、プラットフォームやアプリケーションによる仮想時刻の表現の違い (整数か浮動少数か等) をユーザ側が意識せずに済む。

例えば、WARPED[17] の持っている最適化は多彩である。単なる PSS だけでなく、その間隔をも適応的に調整する方式等である。こうした最適化は、ISS に対しては特にメリットをもたらさないが、CSS で管理してよい状態変数に対する最適化としてはそのまま恩恵を被ることが出来る。

本方式では、カーネルには埋め込みデータの一部に見せることで、ISS と CSS/PSS の混在も可能となる。

4.5 クラスライブラリの実装

4.5.1 CSS/PSS カーネルの定義 / モデル

本提案方式で想定している CSS/PSS プラットフォームは、以下のような形態である

宣言 状態変数を、基本データ型およびクラスの集合としてユーザに定義させる。

状態保存 単一もしくは複数のイベントの実行毎に、上記で定義した状態変数をすべて状態保存用の領域にコピーする。

ロールバック 因果律違反が起きたとき (straggler の到着)、対応する仮想時刻の直前の履歴を状態変数に書き戻す。

Foccil Collection GVT より古い状態履歴は何らかの順序で削除される。

State Invalidation ロールバック時に誤った実行の履歴は削除される。

4.5.2 各四つの操作の検出法と対応して行われる操作

(1) 状態保存,(2) ロールバック / 状態回復 代入演算子を再定義することにより、ライブラリによるインスタンスのコピーを捕捉する。上記の操作は、それぞれ

(1) 状態ベクタ → 状態バッファ

(2) 状態バッファ → 状態ベクタ

という方向の違いがある。カーネルが状態バッファ用に IssQueue クラスのインスタンスを構築した際、未初期化を示すマークをそのインスタンスにつけることにして、未使用のインスタンスと使用

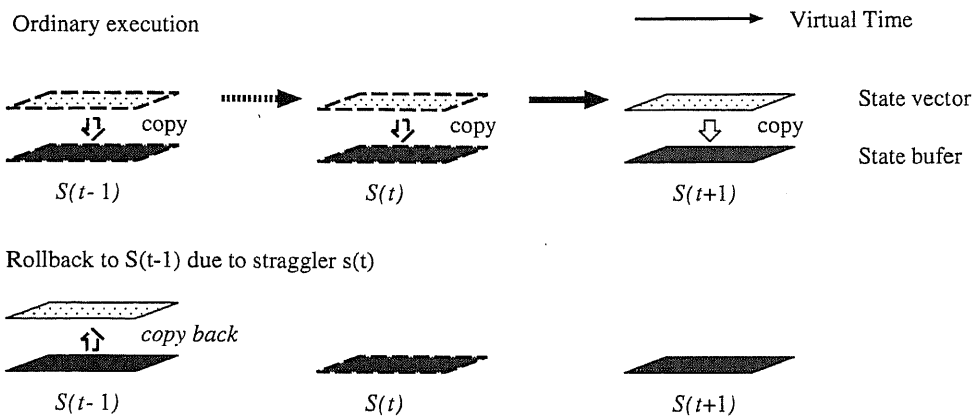

図 4.2: CSS 方式に基づく PDES プラットフォームの状態管理モデル

中のインスタンスを定義することにより、コピーターゲットとなるインスタンスが未使用かどうかで (1) と (2) のケースを区別する。

(1) では、コピー元は状態ベクタでコピー先が状態履歴であるから、内容のコピーを行ったあと、各インスタンス間でカーネルと独立して形成する VT 順のリンクリストにおいて (4.5.3節参照)、コピー先をコピー元と直前に保存されたインスタンスとの間に挿入する。

(2) では、コピー元は状態履歴でコピー先は状態ベクタであるから、状態の回復のケースであり、コピー元はロールバックにより戻った時点での状態を示していると判断する。この場合コピー元より後ろのインスタンスに登録された操作リストを時間と逆順に辿り、undo() メソッドを起動して取り消し操作を行う。同時に取り消し操作で用いたデータ構造は破棄する。また、そのロールバックの操作の間に辿られた履歴インスタンスを無効化しておく。そして内容のコピーを行ったあと、コピー元をコピー先の前に挿入する。

(3) fossil collection カーネルは、メモリの枯渇を防ぐために、適当なタイミングで、シミュレーション全体がロールバックする可能性のある時刻の最大値 (GVT) を求め、それ以前の履歴を破棄する。この場合、カーネルは履歴を構成するインスタンスを消去するために、IssQueue のデストラクタが呼ばれる。デストラクタは自らが一番古い履歴かそうでないかを判断し、一番古い履歴ならば、保持している undo 情報のうち、実際に自分の関与した部分に関する消去を行い、隣のインスタンスとのリンクを切り離す。

(4) 誤った処理に関するの履歴の無効化 / 消去 ロールバックが起った場合、取り消された区間に関する履歴情報は一般に誤りであるため消去されなければならない。不要となった履歴は IssQueue 側で (2) の時に無効の印が付いているため、自分が無効であるフラグが立っている場合 (4) と判断する。

この場合 (3) と同様にデストラクタが呼ばれるが、DoItem インスタンスのメモリ解放は (2) で済んでいるため、ここでは特に何もする必要はない。

4.5.3 一般データ用の IssQueue/DoItem クラス

IssQueue はプラットフォーム側が埋め込みデータとして扱うことが出来るようなクラスであり、プラットフォーム側に状態ベクタの 1 要素として宣言して用いる。個々のインスタンスは、直前の状

態保存から現在の保存の間に行われた操作を DoItem クラスのリストに保持している。ユーザはロールバックしたいデータ構造に関する取り消し可能な操作を DoItem クラスの派生クラスのメソッド undo() として定義し、その操作に必要なデータをその派生クラスに格納する。

プラットフォームの状態バッファは通常時間順に並んだリスト構造となっており、双方向に辿って行くことが可能になっているが、本手法ではプラットフォームに独立した環境を提供するために状態履歴として用いられる IssQueue のインスタンスは互いに独力で時間順の双方向リスト構造を構成する。

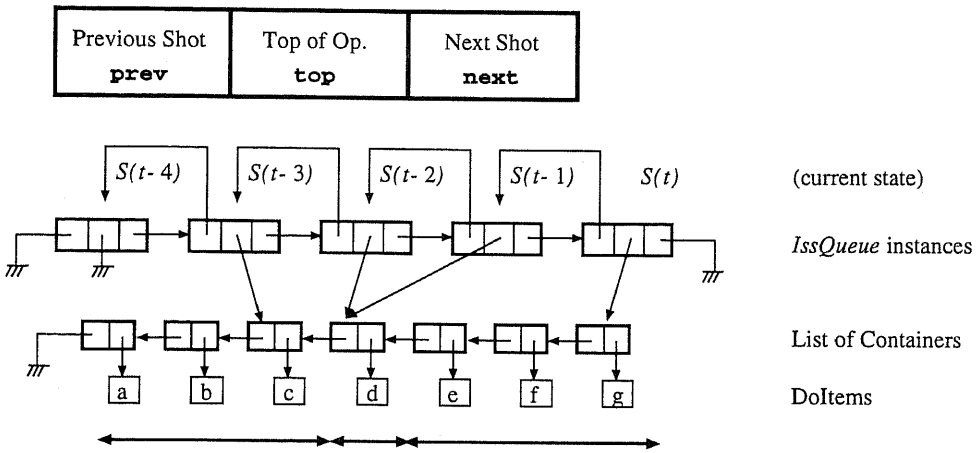

図 4.3: IssQueue の構造

IssQueue および DoItem は、以下のようなデータ構造を持つ。

```
 1:class DoItem {
 2:public:
 3:  int flag; // 記録された操作の方向
 4:  virtual void undo(void)=0; // ユーザ定義の取り消しメソッド
 5:};
 6:
 7:class IssQueue {
 8:public:
 9:
10:  IssQueue& operator=(IssQueue& orig); // 状態保存, 回復操作を捕捉する
11:
12:  ~IssQueue(); // 履歴の破棄 / 無効化を捕捉する
13:
14:  void push(DoItem *elem); // DO/UNDO 情報を追加
15:
16:private:
17:
18:  // DoItem クラスのリストを構成する内部クラス
19:  class ue {
20:  public:
21:     DoItem *elem; // UNDO 情報へのポインタ
22:     class ue *next; // 次のコンテナへのポインタ
23:  };
24:
25:  IssQueue *prev;
```

```
26:  IssQueue *next;
27:  ue *top; // 保存されている DoItem オブジェクトスタック
28:
29:};
```

4.5.4 各層のインタフェース

本提案方式での各層のインタフェースを以下にまとめる。

CSS カーネル側インタフェース 第 3章でのカーネル側インタフェースと同じであり、カーネルの状態用領域確保、解放、状態の保存 / 回復に伴う代入演算に対応して、`new`、`delete`、`operator=` 演算子が呼ばれる。

カーネル WARPED 用には特に変更を加える必要はないが、それ以外に図 4.1に述べている WarpKit[23] や GTW++[21] は、C++ 言語機構による演算を直接行なわず、状態ベクタの先頭アドレスとサイズだけを基に管理を行っているため、カーネルに対応するインタフェースの提供を受けるよう変更を依頼する必要がある[1]。

同期検出インタフェース

- `void saveContent(void ISSQ *orig)`

  `orig` を現在の状態、`this` を状態バッファとして、必要な状態保存を行う。

- `void restoreContent(ISSQ *orig)`

  `orig` をロールバック後の状態、`this` を回復対象の状態ベクタとして、必要な状態回復のためのコピーを行う。

- `void rollback()`

  ロールバック処理を行う。

- `void fossilCollect()`

  無効となった履歴を削除する。

ISS 層

- `void push(DoItem *)`

  状態に加えられた変更の取り消しに必要な情報を格納した DoItem インスタンスを登録する。

4.6 応用例

本提案方式により、CSS/PSS 方式の PDES プラットフォーム上で、可逆な操作が定義出来る任意のデータ構造を状態として扱うことが可能である。

本節では、本提案方式の用例として、IssQueue の一般的な利用手順、およびそれを用いた手続きに基づく ISS によるスタック構造の管理、部分状態保存の実現について述べる。最後に ISSQ 階層を直接用い、スタックのデータ構造における線形的な側面に着目した SsQueue と同様のアプローチの適用例を示す。

[1] これ等のインタフェースは warped では version0.9 以降で明示的に提供されている

4.6.1 IssQueue の利用手順

手順は以下のようにまとめられる

(1) データ構造に対応する IssQueue インスタンスの宣言

(2) データ構造に対する可逆な操作の定義

(3) データを実際に操作する部分での操作の内容の登録

4.6.2 手続きに基づく ISS によるスタック管理の例

図 4.4に、スタック構造の手続きに基づく ISS を WARPED[17] 上で実現した例を示す。

1 行目で IssQueue のデータ構造を定義したヘッダファイルを取り込んでいる。

5 行目から 10 行目迄では、 DoItem を継承して ISS を行いたいデータ構造 (ここではスタック) の取り消し操作のための情報を格納させる。ここでは対象のスタックへのポインタと、 POP されたオブジェクトへのポインタを宣言している。

取り消し操作はそのデータ構造のメンバ関数 undo() で定義される (13 行目～ 21 行目)。変数 flag の値により、対応する操作を行うことが出来るようにする。

ここでは、 PUSH 操作 (flag=0) に対応する反対の操作は POP 操作なので POP 操作を行う動作を記述し、 fossil collection 時に POP された要素も回収されるようにその要素へのポインタも確保する。 POP 操作 (flag=1) に対しては、 PUSH 操作を記述する (2)。

ユーザはデータ構造を非状態変数として用意し (36 行目)、そのデータ構造に対応した IssQueue インスタンスを状態ベクタのひとつとして定義する (27 行目)。

実際の操作 (47 行目) の直後に、対応する DoItem の派生クラスのデータを設定し、状態ベクタ上の対応する IssQueue の push メソッドを用いてそのデータを登録する (52 行目)。

4.6.3 View に基づくスタック管理の例

ここでは、スタック・データ構造の持つ線形的な性質を用いて SsQueue と同様の最適化を施すために ISSQ 階層の上に直接スタック構造の状態管理機構を構築する例を示す。

設計にあたり、以下の点に留意する必要がある。

- snapshot 毎に独立して GC が行えること

  GC は参照されなくなったデータの消去であるので、スタックの場合は POP されたコンテナおよびデータがその対象となる。各インスタンスは前のショット以降に POP されたコンテナを区別出来る必要がある。

- 隣接する snapshot をマージ出来ること

  カーネルの削除順序に依らないで GC を可能にするために、一番古いショット以外のスナップショットが消去された時に、操作履歴を直前のショットとマージ出来る必要がある。

- 個々の shot は保持しているデータサイズに依らず充分小さいこと

  派生クラスで付け加わるメンバはなるべく小さい方がよい。

- 1shot 中に複数の操作履歴を維持出来ること

  カーネルが PSS を行う場合は複数の操作毎に状態保存が行われる可能性があるため、複数の操作に関する情報を保持出来る必要がある。

```
1:#include "IssQueue.hh" // IssQueue ヘッダの取り込み
2:
3:// ISS を行いたいデータ構造 (ここではスタック) の
4:// 取り消し操作のための情報
5:class StackDoItem : public DoItem {
6:   void undo(void); // 取り消しの際に呼ばれるメソッド
7:public:
8:   void *element; // push された要素
9:   queue_t *sp;     // ISS 対象のデータ構造
10:};
11:// ユーザ定義の 取り消しメソッド
12:// DoItem::flag の値により, 取り消し操作と再適用操作を行う
13:void StackDoItem::undo(void) {
14:   if (flag==1) { // push 操作を取り消す
15:      element = q_pop(sp);
16:      flag = 0;
17:   } else {// pop 操作を取り消す
18:      q_push(sp, element);
19:      flag = 1;
20:   }
21:}
22:// ユーザレベルの記述法
23:
24:// 状態ベクタの宣言
25:class UserState {
26:   ···
27:   IssQueue isq1; // 他の inline データ同様に宣言
28:}
29:// ユーザ定義状態ベクタの初期化
30:void UserState::init() {
31:   isq1.init(); // IssQueue オブジェクトの初期化
32:}
33:// シミュレーションオブジェクトの定義
34:class Object : public SimulationObj {
35:   ···
36:   stack_t *stk; // ISS 対象のユーザオブジェクト
37:}
38:void Object::initialize() {
39:   stk = stk_new(); // ISS 対象ユーザオブジェクトの初期化
40:}
41:// 実際の操作 (イベント処理の書き方)
42:void Object::executeProcess() {
43:
44:   /******** スタックへの PUSH 操作 ***********/
45:
46:   char *s;
47:   q_push(sp,(void *)s); // 実際の操作
48:
49:   StackDoItem *sdi = new StackDoItem();
50:   sdi->flag = 1;   // 操作の方向は PUSH
51:   sdi->sp = sp;   // 操作対象となるスタックへのポインタ
52:   state->current.isq1.push(sdi); // 操作の登録
53:
54:   ···
55:}
```

図 4.4: IssQueue を用いたコーディング例

- 状態回復コストは $O(1)$

  CSS におけるロールバック同様、状態の回復コストそのものはロールバック距離に依らない。

以上の特徴を踏まえ、スタックの実装は図 4.5のようになるであろう。

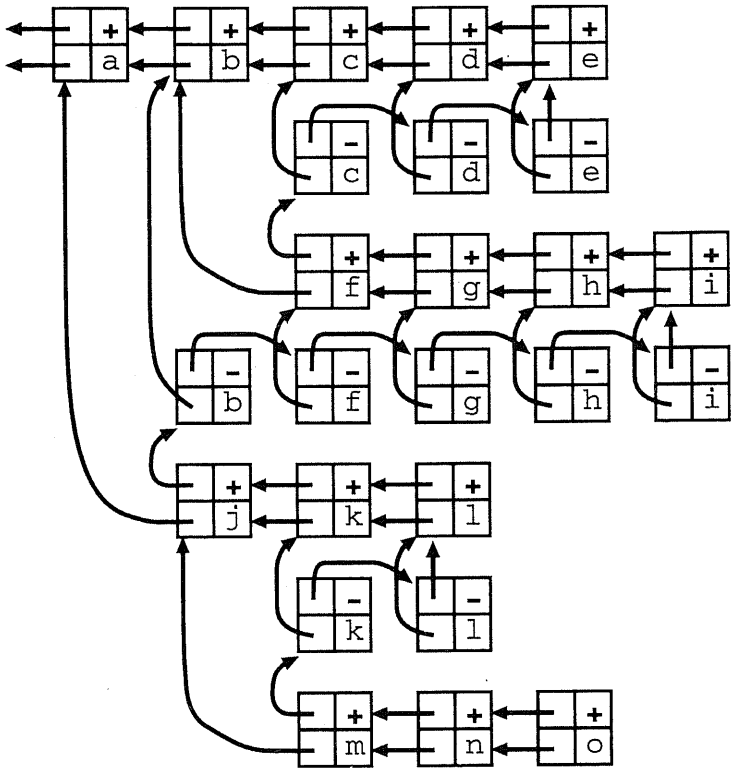

図 4.5: スタック構造の状態管理:View に基づく表現

図の中で、十字のますはスタックに加えられた操作に関する情報を保持するクラスのインスタンスである。左上のポインタは連続して加えられた操作のつながりを示すリンクである。左下のポインタは、加えられた操作が PUSH ならばスタックの底へ向うリンクであり、POP ならば対応する PUSH を表現するインスタンスへのポインタが格納される。右上の符号は PUSH のとき +、POP のとき − である。右下は実際に格納されるデータである。

fossil collection の際にはデータ構造から排出されたデータを実際に消去し、state invalidation の際にはデータ構造へ挿入されたデータを削除する必要があるため、前者の場合は GVT を表現する位置のインスタンスより以前で符号が − であるもののペアを消去し、後者の場合はロールバック地点以降のインスタンスをすべて消去する。

4.6.4 部分状態保存の実現

partial-state-saving を本提案手法を用いて実現する例を図 4.6に示す。

IssQueue クラスを継承した BlkIssQueue クラスのメソッド memcpy_recoverable() による取り消し可能なメモリコピーメソッドを実現している (4 行目)。変数の単純な代入式もこのメソッドで表現出来る。取り消しに必要な情報は上書きされた領域のデータであるので、DoItem クラスの派生クラスである取り消しデータ BlkDoItem クラスでは上書きされた領域の先頭アドレスと、元データを退

```
 1:class BlkIssQueue : public IssQueue {
 2:public:
 3:  // 取り消し可能なメモリコピー
 4:  void memcpy_recoverable(void *dest, void *src , size_t size);
 5:};
 6:
 7:// rollback オブジェクトの定義
 8:class BlkDoItem : public DoItem {
 9:  void undo(void); // 取り消し手続き
10:public:
11:  void *oldStart; // 上書きされる領域のバックアップの先頭アドレス
12:  void *newStart; // 上書きされる領域の先頭アドレス
13:  size_t size; // コピー長
14:};
15:
16:// ユーザ定義の 取り消しメソッド
17:void BlkDoItem::undo(void) {
18:  void *tmp = malloc(size); // swap area
19:  memcpy(tmp,newStart,size); // save the overwritten area
20:  memcpy(newStart,oldStart,size); // restore
21:  memcpy(oldStart,tmp,size); // swap complete
22:  free(tmp);
23:  if (flag == 0) flag = 1; else flag = 0;
24:}
25:
26:// 取り消し可能なメモリコピーの実現
27:void BlkIssQueue::memcpy_recoverable(void *dest, void *src, size_t size) {
28:  BlkDoItem *bdi = new BlkDoItem();
29:  bdi->oldStart=(void *)malloc(size);
30:  memcpy(bdi->oldStart,dest,size); // backup overwritten area
31:
32:  memcpy(dest,src,size); // do perform memcpy
33:  bdi->newStart = dest; // memorize destination
34:  bdi->size = size;
35:  bdi->flag = 1;
36:  push(bdi); // mark alternation of state
37:}
```

図 4.6: IssQueue を用いた部分状態保存の例

避している領域の先頭アドレス、領域の大きさ保持するようにしている (8 ～ 14 行目)。取り消しメソッドの実現では (17 ～ 24 行目)、上書きされるデータと退避されていた元データの交換を行っている。

4.7 性能に関する議論

状態管理方式の性能評価は、平均ロールバック距離、全実行中のロールバック回数、状態の保存あたりのコスト、ISS の場合は加えて一回の操作のコスト等をパラメータにして評価がなされる [44, 45]。CSS 方式のオーバーヘッドは一般にロールバック距離とロールバック回数に比例した誤った実行における状態保存コストと、全イベント数に比例した正常実行時の状態保存コストの和で表現される。我々

は第3章で、SsQueueの管理コストも状態回復時のコストが十分小さくCSS方式と同様のコスト表現になることを示した。

本提案方式の目的は性能を損なわずにCSSカーネル上で一般的なISSを実現することである。CSSカーネルの状態保存毎にISSQの階層でISSQインスタンス分だけのコピーが行われるため、その大きさのデータに関するCSSのコストに加え、IssQueueではDoItemの管理コスト、更にアプリケーションレベルではロールバック長と個々のデータ構造に対する取り消し操作コストの積に比例したコストが必要になると言える。

本提案方式はpartial-state-savingとしてのISSにも用いることが出来るが、その場合の性能評価はCSS方式と比較して変更対象領域の割合が小さい程ISSがCSSより有利になるという評価の仕方がされる。本方式はISSQ分のインスタンスのためのCSS管理コストがかかっているため、もともと小さい領域に関してはカーネルの提供するCSSによる実現が望ましい。

また、本方式はsave-if-modifiedとしても適用出来るが、たとえ状態に変更が加えられない場合でもカーネルのCSS操作によってインスタンス分の領域のコピーが行われるため、同様に単一の変更コストが大きい場合に有利となる。

ISSQを用いて任意のデータ構造のISSを実現する場合、現状では上位層の定義するrollback()等のルーチンは仮想関数として定義されているが、仮想関数テーブルのためにインスタンスの大きさが若干増加し、関数呼び出しのオーバーヘッドも加算される。これを避けるためにテンプレートを用いた静的束縛を行うことも考えられる。

4.8 関連研究

文献[47]では、タイムワープカーネルJade用の状態管理モジュールを外部化する枠組みを提供し、データ構造の種類に応じた最適化をカーネルの外部で行うことを可能にしている。本提案方式は、CSSカーネルが暗黙的に行う同期処理の検出を受動的に行うことに対し、ここで述べられている手法はカーネルが明示的に同期処理手続きを呼び出す点が異なる。CSSカーネルに図4.1のISSQモジュールを組み合わせることにより、等価なインタフェースを提供することが可能である。

4.9 まとめ

本章では、CSS方式に基づくタイムワーププラットフォーム上でISSを実現するための枠組みを提案し、C++パッケージ群による実現と応用例を示した。CSS方式は利用が容易である反面複雑なデータ構造の管理に不向きである。本枠組みを用いることにより、カーネルへ変更を加えることなくISSを透過的に実現することが出来、CSSの利便性を損なわずにISSを実現することが可能になる。

PDES汎用プラットフォームは様々なものが提案され、実現形態もまちまちであるが、個々のシステムは用途向けに多様な最適化が施されている。現在のところいずれかのプラットフォームが凌駕している状況ではないことを鑑みると、提案方式を用いていくつかのCSSプラットフォームに共通のISSインタフェースを与えることによりそれらのプラットフォームの利点を享受しつつ利用の容易性を増大させる一助となると確信する。

第 5 章

論理プロセスのスケジューリング機構

5.1 はじめに

PDES では、一般に、サブモデルの数と並列計算機上のプロセッサの数は異なるため、通常単一のプロセッサが複数のサブモデルをシミュレートする。この場合、複数のサブモデルの間での実行順序の解決が必要となる。

この解決方法の一つとして、サブモデルが持つ事象リストの先頭要素のタイムスタンプの小さい順に制御を移す最小時刻印法 (Least Time Stamp First: 以下 LTSF) が取られる [17]。

また、保守的プロトコルの場合、楽観的プロトコルと異なり因果律を厳密に守るために、別のサブモデルから受け取ったイベントのうちタイムスタンプの最小のものを常に把握しておく必要がある。

これ等のスケジューリングのために、プライオリティキューと呼ばれるデータ構造を用いることが出来る。プライオリティキューは、要素をキーの値の順に取り出すことが出来るデータ構造である。プライオリティキューの実装方式として、最高優先度の要素の取り出し、要素の挿入が高速に行えるよう様々な方式が提案されている [48, 49, 50]。

上記のサブモデル間のスケジューリングや受け取ったイベントの整順にプライオリティキューを用いる場合、サブモデルの数が変化しない限り要素の数が一定で、各要素が優先度を上下させるという特徴があり、こうした特徴を生かした効率よい実装としてトーナメント形式のデータ構造が提案されている [30]。このデータ構造を用いると、要素の優先度の変更が平均で定数コストで済むが、LTSF スケジューリングのような最高優先度の要素の変更は木構造の根から葉まで全て再評価する必要があるため、常に $O(\log N)$ のコストが必要である。このような問題は、二進木構造を持つヒープを用いて解決することが出来る。

本章では、以下、第 5.2節で PDES におけるサブモデルのスケジューリングについて概説し、第 5.3節では、提案するヒープ型と従来のトーナメント型の両データ構造の C++ クラスライブラリによる実装について述べる。第 5.4節で両者の性能比較を行うとともに他の関連研究との比較を行い、5.5節で結論と今後の展望についてまとめる。最後に 5.6節で楽観的プロトコルへの適用について議論する。

5.2 論理プロセスのスケジューリング

図 5.1に、保守的論理プロセスの状態遷移モデルを示す。保守的プロトコルでは入力バッファがひとつでも空であると実行不可能となるため INACTIVE 状態になり、入力バッファに外部イベントが全て到着して ACTIVE になる。ACTIVE な状態において外部イベントを実行していずれかの入力バッファが空になると INACTIVE になる。

PDES において、複数の LP を同一プロセッサ (以下 PE) が担当することは、粒度を調整する上でも、柔軟なプログラミングインタフェースを提供する上でも重要である。一つの PE に配分された複

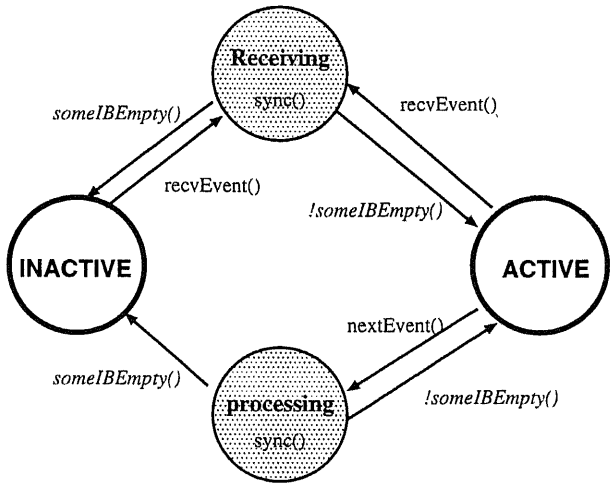

図 5.1: 保守的論理プロセスの状態遷移

数の LP はイベント単位で逐次的に制御を渡される。この制御の渡し方は通常次に実行すべきイベントのタイムスタンプが最も若いものを選択する最小時刻印法 (Least Time Stamp First Scheduling) が採用されている。

各 PE における LP のスケジューリングも、PDES のプロトコルオーバーヘッドの一部になっており、本研究で想定している大規模通信アプリケーションにおいては PE 当たりの LP の数が大きくなると無視し難くなると思われる。文献 [30] では、トーナメント形式のプライオリティキューを用いてスケジューリングの高速化を図っている。しかしこの方式では PE あたりの LP 数を N とすると、最小タイムスタンプを更新した際の書き換え操作が常に $\log N$ 回になる欠点がある。本研究では、二進木データ構造によるヒープに、要素の挿入、削除を伴わずにキー値のみを変更出来るように改良を加えたデータ構造を用いることにより、最小タイムスタンプの変更コストをより小さく抑える手法を提案する。

5.3 プライオリティキューの実現

本節では、LP のスケジューリングに適したデータ構造として、従来の方式であるトーナメント形式のプライオリティキューと、本研究で提案する二進木ヒープを用いたプライオリティキューの両方式の実装について述べる。

5.3.1 トーナメント型プライオリティキュー [30]

トーナメント型のプライオリティキュー [30] を図 5.2に示す。

二進木の葉の部分にデータを格納し、節点には下位の要素のキーのうち優先度の高い方 (図では数値の小さい方) の値を格納する。根の節点に格納されている値がデータ全体の最小値を与える。図中の E は、LP が次に実行可能なイベントを持たず、実行不可能状態 (図 5.1における INACTIVE 状態) にあることを示す最低優先度値を示している。

データのキー値を変更すると、その葉から根に至る経路上の節点の値を再評価するため、データの総数を N とすると最高で $\log N$ 回の処理が必要となる。各段での再評価が必要となる確率が $1/2$ とすると、この再評価回数は N が大きくなると定数 2 に近づく。LP の LTSF スケジューリングに用

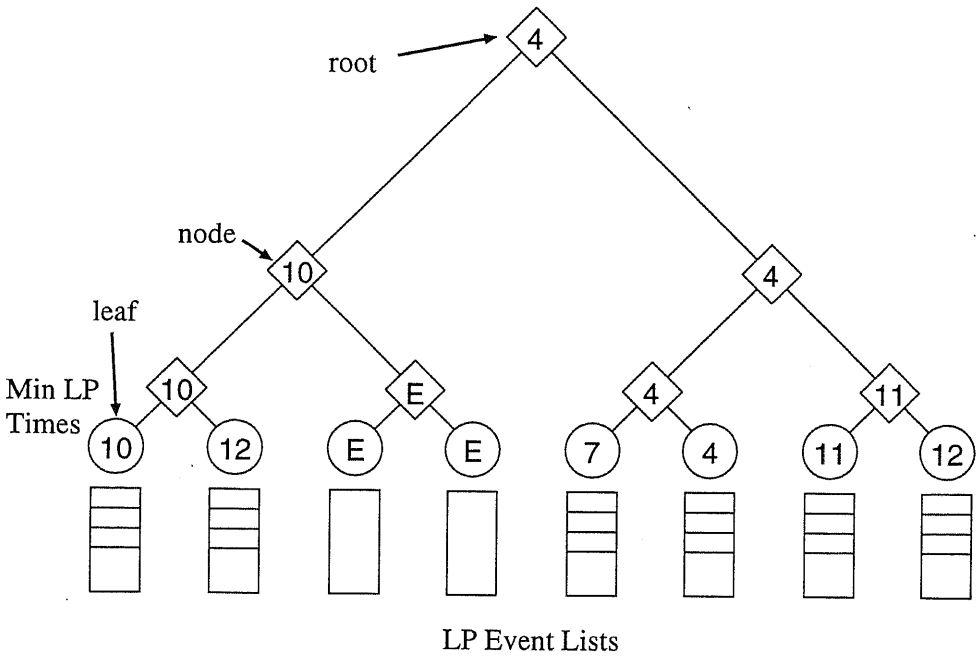

図 5.2: トーナメント型のプライオリティ・キュー [30]

いる場合は、イベントの実行時に書き換えられる値は常に最高優先度の LP のキーである。よって常に $\log N$ 回の再評価が必要になる。

5.3.2 二進木ヒープ型プラオリティキュー

プライオリティキューの各操作を $O(\log N)$ で実現する別のデータ構造として、二進木構造を用いたヒープがある。ヒープは、全節点にデータを保持し、節点とその直下の左右の節点との間に、「親の値は左右の子のいずれよりも小さいか等しい」というヒープ条件を維持するものである [48]。図 5.3に、ヒープにおけるデータの挿入手続きを示す。新しいデータを最下段の右端に置き、ヒープ条件が満たされるまで親との間の置換を繰り返す。根の削除は、最下段の右端の要素で根を置き換え、ヒープ条件が成立するまでその子のうち小さい方との置換を繰り返す。

通常このようなヒープはデータの挿入と (最小要素の) 削除の観点からしか述べられないが、格納されている任意のデータのキー値の更新も同様な手続きで実現出来る。更に、[30] と異なり最小要素のキー値を変更しても、その近傍の子との間の置換で済む場合はより低いコストで実現する可能性がある。

5.3.3 クラスライブラリ・インタフェース

本研究では、LP のスケジューリング以外の一般的な利用も可能にするためと、性能比較上の都合から、トーナメント型とヒープ型の両データ構造を同一のインタフェースを持つ C++ のクラスライブラリとして実現した。

通常の dequeue、enqueue 操作以外に、LTSF スケジューリングや入力バッファの整順のように、一旦削除した後再挿入するのではなく、キューの中に保持したまま優先度のみ変更させるには、変更対象要素の、木の中での位置を追跡出来なければならない。本方式では、要素と所属するノードへの

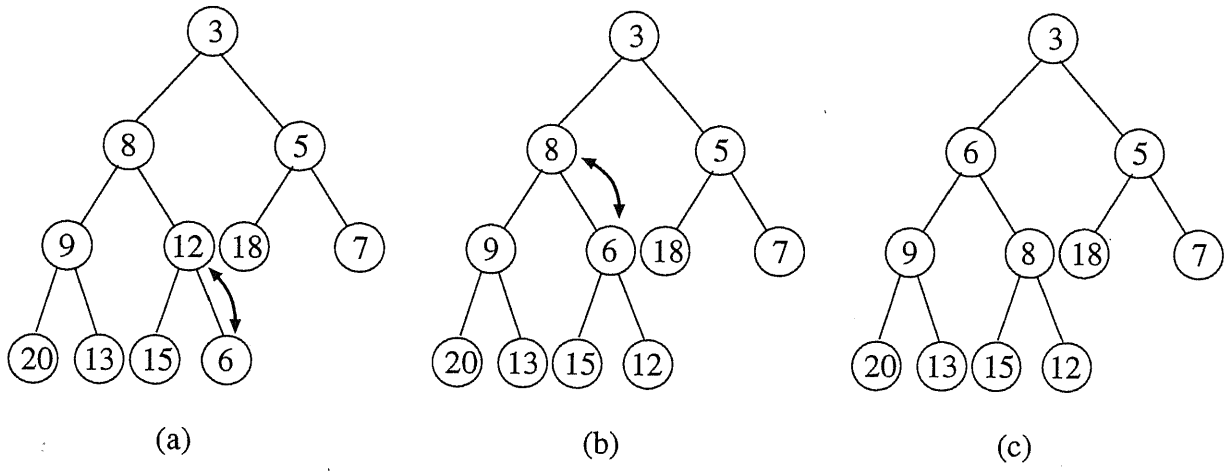

図 5.3: ヒープにおけるデータの挿入

ポインタを組み合わせたデータ構造をプライオリティキューの要素とすることにし、ノードへのポインタを含む基本クラス Item に扱うデータ型を追加した派生クラスを定義することにより実現する。

大小比較の関数は、コンストラクタの引数で、基本要素クラスへのポインタを引数とした関数として与えられる。

更新操作は以下のような呼び出しインタフェースを持つ

```
void sync(Item *)
```

要素のプライオリティの変更後このメソッドを呼ぶことにより、要素はプライオリティキューの中での優先度を反映した位置に移動する。特に、ヒープ型の場合は、

優先度が上がる場合 親との比較を繰り返して入れ替わって行く

優先度が下がる場合 優先度の高い方の下位ノードと比較して入れ替わって行く

という操作が行われる。

トーナメント型のキューの実装では、各ノードには subtree の最高優先度 Item へのポインタと、左右の subtree へのポインタ、親へのポインタが含まれる。

また、両者ともノードの挿入時に挿入場所の特定が $O(1)$ の時間で可能になるように、トーナメント型では最下段の空きノードをスタックに保持し、ヒープ型ではノード同士の横方向のつながりを維持するポインタを用意している。

5.4 性能評価

LP のスケジューリングにおけるプライオリティキューの操作パターンの特徴は以下のようにまとめることが出来る。

1. 最小のキーを持つ LP が (スケジューリングにより) 頻繁にキー値の更新を行う。
2. アクティブな任意の LP がキーの値を変化させる時、その変化は必ず増加である。

3. 要素の数は (LP の移送や、LP の数が実行時に変化するアプリケーションを実装する場合以外は) 変化しない。

4. LP のアクティビティ (実行可能状態か外部イベント待ち状態か) が変化するとき、キー値は最低優先度値から通常値の間で急激に変化する。

トーナメント型のキューとヒープ型キューの全体的な性能比較を表 5.1に示す。

評価項目	ヒープ型	トーナメント型
挿入	平均 定数 最大 $\log N$(minval)	平均 定数 最大 $\log N$(minval) 再構成 N
(先頭要素の)削除	平均 $\log N$ 最大 $\log N$	平均 $\log N$ 最大 $\log N$(minval) 再構成 N
キー変更	平均 定数 最大 $\log N$(min↔max)	平均 定数 最大 $\log N$(minval)
activate	平均 定数 最大 $\log N$(E→min)	平均 定数 最大 $\log N$(E→min)
inactivate	平均 定数 最大 $\log N$(min→E)	平均 定数 最大 $\log N$(min→E)

表 5.1: ヒープ型プライオリティキューとトーナメント型プライオリティキューの性能比較

表中で、E は図 5.2と同様、論理プロセスが実行不可能状態になっている場合の最低優先度を示す特別な値を示している。また、min は最高優先度値、max は最低優先度値を表す。

これらの特徴と表 5.1の結果を照合すると、LTSF スケジューリングに特有の根のデータの更新にはヒープ型が適しているが、最後の特徴により保守プロトコルにおいて LP のアクティビティの変化が頻繁に生じる場合はヒープ型は不利になる場合があることが分かる。

ここでは、特に LTSF スケジューリングに於いて問題となる最優先度要素の更新の性能について、ワークステーション上で様々なイベントスケジューリングパターンを想定してシミュレーションを行った。

シミュレーションではハードウェアに Sun Enterprise 450 (動作周波数 300MHz, 主記憶 1GByte, 外部キャッシュ 1MByte) を用い、最小要素の更新を 100 万回行うのに要した CPU 時間を測定した。ランダムな優先度の下げ幅の生成には以下のよう関数を用いた。x を $0 \leqq x < 1$ の一様乱数として、$f(x)$ が優先度の下げ幅とすると、

$$f(x) = \begin{cases} x & \text{一様分布} \\ -\log x & \text{負指数分布} \\ 1 - \sqrt[40]{x} & \text{Near Future 分布} \\ g(x) & \text{Dense \& Sparse 分布} \end{cases}$$

ただし、

$$g(x) = \begin{cases} 20000 \times \lfloor 200x \rfloor & (x \geqq 0.95) \\ \frac{x}{5} & (x < 0.95) \end{cases}$$

Near Futre 分布は、主なスケジューリングは近い将来に対してであり、遠い将来のスケジュールは稀にしか行われないような場合を想定している。最後の Dense&Sparse 分布は、Near Futre 分布と似ているが、通信シミュレーションに於いて、トラヒックの少ない要素を担当する LP が、それがスケジュールされる時は頻繁にイベントを発生させて active であるが、一旦暇になると (遠い将来のイベントをスケジュール後) 暫く不活性な状態にあるような場合で、かつ再活性化される迄の間隔がある長さの定数倍になっている場合を想定している。

測定結果を図 5.4,5.6,5.5,5.7に示す。

横軸を対数に取っていることから、コストがキュー中の要素数の対数に比例している場合に直線になる。全ての図においてノード数の大きい領域ではそれぞれ直線から逸脱しているが、これはキャッシュミスによる効果であると考えられる。

また、各分布間の縦軸方向のずれは、乱数生成部分のコストの違いであるため、プライオリティキューの性能には関係しないと思われる (Near Future では巾乗計算に時間がかかっている)。

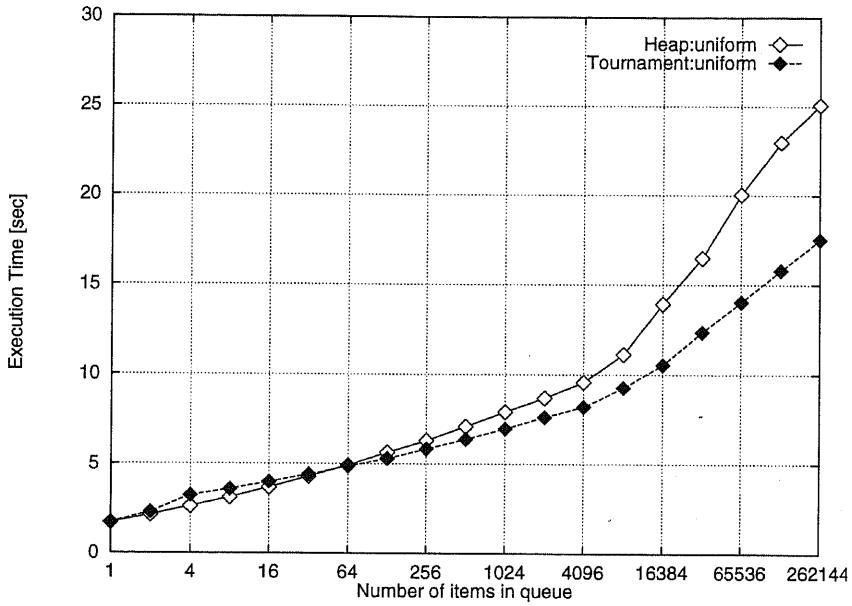

図 5.4: ヒープ型とトーナメント型の最小要素の 100 万回更新の所要時間 (一様分布)

一様分布 (図 5.4)、負指数分布 (図 5.5)、Near Futre 分布 (図 5.6) では、両者ともノード数の対数に比例したコストを要し、ヒープの場合は、下方向へのノードの更新の際に、下位のノード同士の優先度比較が必要な分要素同士の比較回数が多いことを反映して、傾きがトーナメント型より大きくなっているため、ノード数が 64 以下の領域では若干ヒープ型の方が良い性能が得られているものの、要素数が大きくなると性能が逆転してしまうことを示している。

Dense&Sparse 分布では、ヒープ型に於いてはノードの置換が根付近で済んでいることによりノード数の少ない領域では更新がノード数によらない定数コストで実現していることが分かる。

以上の結果から、ヒープ型はノードの入れ替わりの絶対数に関してはトーナメント型の再評価回数より少ない筈であったが、ノードの上から下への移動はトーナメント型の場合に比べて倍の大小比較操作が必要であるため期待通りの性能が得られないことが分かった。

木構造においてはノードの密度は深さに対して指数的に増大するため、大部分は葉に近い部分に分布していることになり、優先度の更新を常に葉から開始するトーナメント型に比較して、ヒープ型で根の方向から更新を行う場合は大部分が更新が葉の部分まで及ぶため $O(\log N)$ のコストがかかって

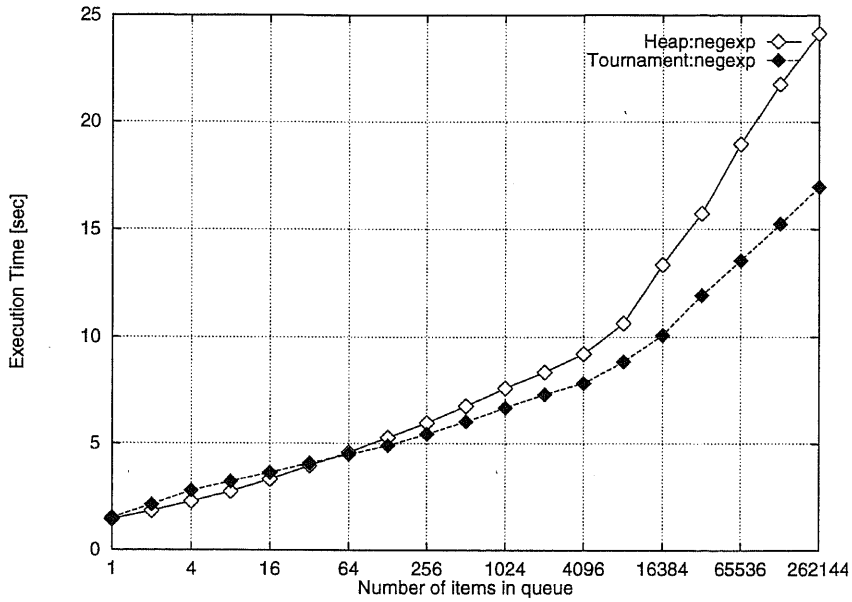

図 5.5: ヒープ型とトーナメント型の最小要素の 100 万回更新の所要時間 (負指数分布)

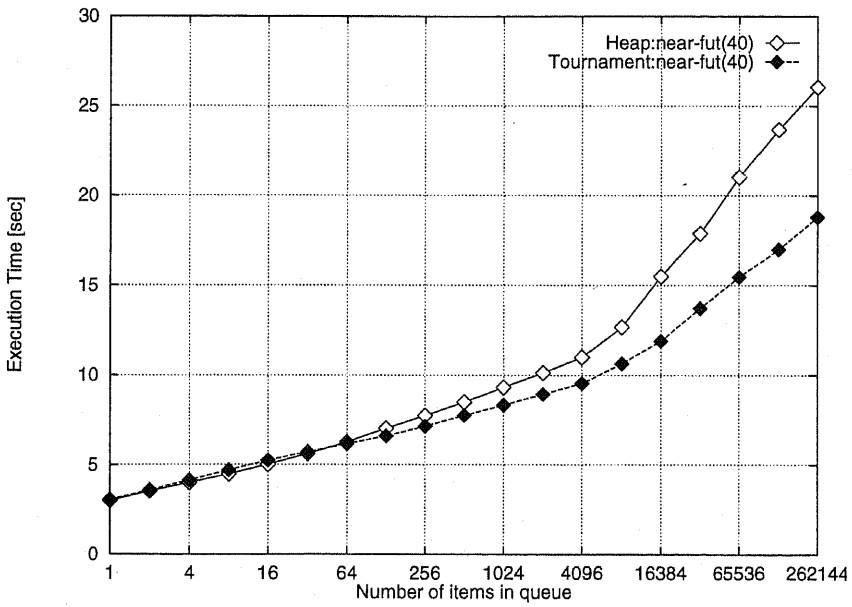

図 5.6: ヒープ型とトーナメント型の最小要素の 100 万回更新の所要時間 (Near Future)

しまう。

但し、メモリ消費量の観点からは、ヒープ型では木構造の全ノードにデータが含まれているので、トーナメント構造に比してノード数は半分になる。今回実装したクラスライブラリでの実際のメモリ消費量は、ノードの構造がトーナメント型より複雑になったため半分にはならず、ノード数が大きい時に約 $2/3$ であった。

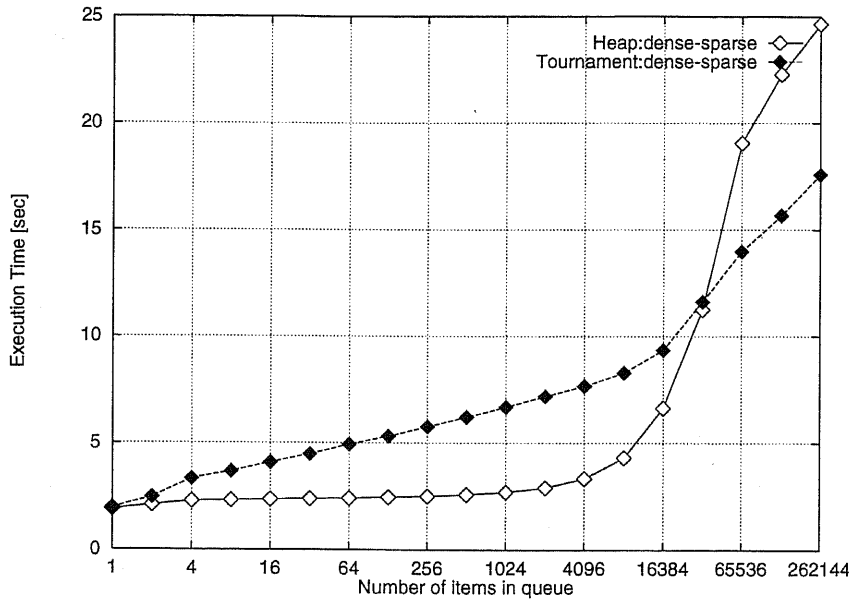

図 5.7: ヒープ型とトーナメント型の最小要素の 100 万更新の所要時間 (Dense & Sparse 分布)

他のデータ構造との比較

プライオリティキューの他の実現形態として、木構造の他にマルチリスト型のプライオリティキューのひとつである Calendar Queue が提案されている [49]。原理的にはハッシュ法が $O(1)$ の操作コストを実現していることと同様で平均 $O(1)$ の操作コストを持つと言われているが、最悪 $O(N)$ のコストが必要となる。

Calendar Queue を改良したものとして Lazy Queue[50] があり、最悪 $O(\log N)$ と言われているが、実際に $O(1)$ の操作コストを得るためには、アプリケーションに応じた最適なパラメータをキューの構築時に与えなければならない点が問題である。

5.5 まとめ

PDES における論理プロセス (LP) のスケジューリング機構を高速に行うためのデータ構造を、従来の二進木ヒープに更新機構を付加した、独立した C++ のクラスライブラリとして実現し、最小時刻印法によるスケジューリングに対応する、最高優先度の要素の優先度を下げるパターンについて、従来提案されていたトーナメント型のプライオリティキューと性能比較を行った。

要素の数が少ない場合、負指数分布や一様分布等通常よく用いられる分布においてトーナメント型より若干良い性能が得られた。

しかし、要素数が増えた場合は従来方式のトーナメント型の方が性能が良いことがわかった。原因は、ヒープ型が本質的に上から下へ (トップダウン) という更新パターンになるため、深さ方向でノード数が多くなっていく木構造では、大部分が下まで更新されてしまうまでノードの入れ替わりが起きてしまうことにあった。また、ヒープでは下向きの更新は優先度の比較数が、更新が常に上向きであるトーナメント型に比べて 2 倍であるため、それも併せて効率が低下してしまった。

ヒープ型について下位のノード同士の優先度比較の結果をキャッシングすることによる最適化も考えられるが、記録した優先度比較結果は周辺でのノード置換が起ると結局無効化されてしまうことがほとんどで、それによる性能改善は見込めないと思われる。

他の性能改善手法としては、[29] や [50] で取られているように、優先度の低い部分の要素の整順を遅らせる一般的な方法があり、論理プロセススケジューリングのパターンを考慮して適用可能か検討する余地が残されている。

5.6 楽観的プロトコルにおける論理プロセスのスケジューリング

前節まででは、保守プロトコルにおける IB 間のスケジューリングと LP 間のスケジューリングに適用可能なヒープデータ構造を提案し、その評価を行った。また、合成負荷に対する単体性能に基づくトーナメント型プライオリティキューとの比較を行った。

本節では楽観的プロトコルのひとつである Time Warp におけるスケジューリングの問題を取り上げ、既存のカーネルに採用されているマルチリスト[1]による実現との比較に関する議論を行う。PE 内での LP 間スケジューリングは LP に対して LVT 順に制御を移す点は保守プロトコルと同様であり、前節で提案したデータ構造は任意の要素の優先順位を更新出来るためそのまま適用可能である。一方、LP 内のイベントスケジューリングはロールバックや再実行を実現するための自明でない工夫が必要となる。本節での議論は主に後者に対するデータ構造と処理の工夫に費やされる。

5.6.1 スケジューリングの観点からの保守プロトコルと楽観的プロトコルの相違

Time Warp は投機的に実行を行うため既に実行したイベント群もロールバックに備えて時間順にアクセス出来るように保存しておく必要がある。

また、外部イベントはタイムスタンプ順に到着しないため、保守プロトコルでは IB の整順が不要であったのに対し、入力キュー (実行済イベントも含めて受信したイベントをタイムスタンプ順に格納しておくキュー; 以下 IQ) での整順のコストが避けられない。

LP 間スケジューリングにおいても各 LP の LVT は単調増加であったが Time Warp の場合はロールバックにより減少する場合がある。

結合トポロジ情報と性能 保守プロトコルでは結合トポロジは所与であったが Time Warp では原則として与えられない。よって保守プロトコルでは IB 別に見た外部イベントの到着順はタイムスタンプ順であると仮定出来たが、Time Warp は送信元の LP が特定出来ないため、通常は送信元を区別しない単一の線形な IQ を構成している。IQ を可変長のヒープで置き換えることも考えられるが、挿入 / 削除処理に未処理イベント数の対数に比例したコストが必要となる。

あるいは、実行時にチャネル生成を動的に行うことにより、IB 毎に IQ を分割して、保守プロトコルの場合と同様に CC のヒープを構成することにより挿入時の探索のコストを抑えることも考えられる。

実現の方向性 WARPED[17] ではスケジューリング機構としてマルチリストが採用されている。マルチリストは、線形リストの構造を取るため、最優先度要素以外の要素の時刻順アクセスが可能なため、ロールバック処理が容易であるという利点がある。ヒープの場合、最優先度要素以外の要素は整順されていないため、LP 内スケジューリングに適用する場合は各 IB 毎にロールバック処理が必要である。一方、ヒープの場合、挿入が要素の数の対数に比例したコストで抑えられるという利点がある。

[1] ここでのマルチリストは前節までのマルチリストと異なり、1 本の線形な要素の列を種類の異なる複数のリンクで結合して別々のリストの集合に見せるものを意味する

5.6.2 スケジューリング性能評価のためのタイムワープのモデルの定義

データ構造の性能は、そのデータ構造へのアクセスパターンを決めるモデルによって決まる。モデルは以下のような複数の要因別に決められる。

通信路に起因する要因

- LP 間の追い越しは起らない

  Time Warp はメッセージの追い越しを許すが、実際には例えば MPI では標準 [2] における 1 対 1 通信の意味の定義により、メッセージは追越禁止となっている。PVM[3] でも同様である (第 5 章: 第 46 頁)。この性質を使うと後述のような外部イベントの受信処理の簡略化が可能となる [51]。この前提は多くの通信ライブラリで妥当なものと考えられる。

プロトコルに起因する要因

- 外部イベントは非タイムスタンプ順で到着し、タイムスタンプ順に処理される

  ただし同一送信元からは

  - 送信元がロールバックしない場合は前項の前提条件によりタイムスタンプ順に到着
  - タイムスタンプが減少する方向のアンチメッセージは一番古いアンチメッセージから最新のアンチメッセージまで順に一度に届く
  - タイムスタンプが減少する方向の正イベントは送信済のメッセージに関するアンチメッセージが全部届いたあと届く (正イベントの系列のタイムスタンプが減少に転ずることはない)

トポロジ情報の扱いに起因する要因

- LP 間結合トポロジ情報を利用するか否か

  通常の Time Warp では LP 毎に単一の IQ を用意し、送信元を区別しないが、送信元の集合が前もって特定出来ていれば、保守プロトコルと同様に送信元毎の入力キューを用意してスケジューリングを行うことが出来る。

  トポロジ情報は実行時に収集するか、ユーザから情報を静的に受け取ることになる。

  送信元を区別する場合、送信元と宛先の組から対応する IB を引くためのマッピングに必要なコストは、総 LP 数の自乗に比例した大きさのテーブルを定数時間で引くか、次数に比例した大きさのテーブルを次数の対数に比例した時間で引くかのトレードオフとなる。但し、外部イベントの送出時に IB の指定をユーザにしてもらうことで、IB 選択のコストは不要になる。

モデルに起因する要因

- イベントの生起パターンもコストに影響する

5.6.3 設計方針

設計の選択肢

保守プロトコルでは LP 内での IB の整順と PE 内の LP 間のスケジューリングの両方にヒープを適用した。LP 間スケジューリングには LP の優先度の上下パターンは保守プロトコルと異なるもの

のヒープの適用は有効であると考えられる。このとき、PE 内 LP 数 N に対して平均して $O(\log N)$ のコストが生じる。

一方、Time Warp では原則として IB は存在しないので単一の線形な IQ が使われる。しかしユーザがトポロジ情報を与えたりすることによって IQ を送信 LP 毎に分け、挿入時のスキャンのコストの抑制を図ることが出来る。

Time Warp では、到着する外部イベントに関してタイムスタンプ順の逆転があり、更にロールバックの必要性から、既に実行したイベントも時間順に高速にアクセスする必要がある。

LP 内スケジューリングに関しては、IQ を IB に分ける場合と分けない場合で 2 通り考えられる。

- 分けない場合 (図 5.8)

  標準的なタイムスタンプ順の線形リストとして IQ をひとつ用意する。IQ の挿入コストはイベント生起パターンにも依るが、格納しているイベントの数に比例して必要となると考えられる。

  イベントの実行による削除のコストは先頭要素の削除なので定数である。

  straggler/ アンチメッセージの到着によるキャンセルイベント / ロールバック時刻の選定と該当要素の削除は該当場所から線形にたどることにより単純に実現可能である。

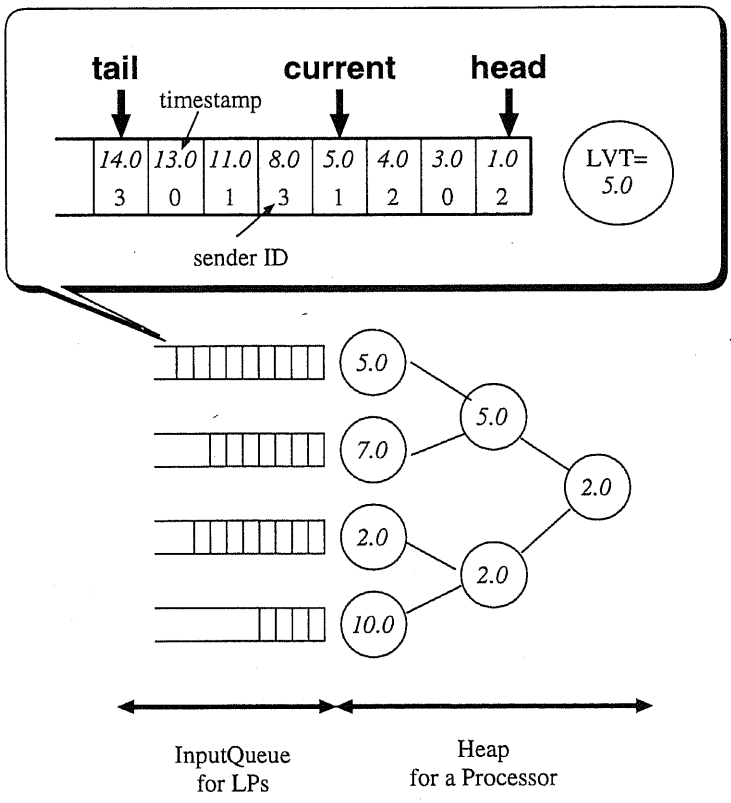

図 5.8: IQ とヒープの組み合わせ

- 分ける場合 (図 5.9)

  IQ を送信元 LP 毎に分割する。

  この場合、既に実行したイベントも引き続き IB に留まらせるか、実行済のイベントは 1 本の線形リストに移すかにより設計方針が異なる。前者の場合はロールバック時に各 IB について探

索処理が必要になるため、ここでは後者を採用する。つまり各 IB には保守プロトコルと同様に未処理のイベントだけが溜っていると仮定出来る。

5.6.2節により正のイベントはタイムスタンプ (降) 順に届き、その逆転が起きるときは必ずアンチメッセージが先に届くため、straggler に関しては空になった IB に関する送信元から到着することになる (投機実行により取り残された送信元)。

更に一般に、到着メッセージのタイムスタンプ (以下 TS_a) と IB の末尾の TS(以下 TS_{tail})、LVT、CC の大小関係により以下のように処理が分かれる。

1. $TS_a \geqq TS_{tail}$
   IB の末尾への追加 (のみ)。
2. $TS_{tail} > TS_a \geqq CC$
   struggler はあり得ない。アンチメッセージの場合 IB 中の TS_a 以降のイベントの全消去。
3. $CC > TS_a \geqq LVT$
   アンチメッセージに関してはり得ない。straggler のみ有り得る。この場合当該 IB は空になっている。よって IB 間ヒープの更新と IB への挿入のみ行う。
4. $LVT > TS_a$
   - straggler のとき
     TS 以降のタイムスタンプを持つ処理済イベントを処理済イベントリストの末尾から順に各 IB へ戻す。当該送信元の分は消去する。処理済イベントリストは整順済なので新たな探索コストは必要ない。ロールバック距離に比例したコストになる。
   - アンチメッセージのとき
     対応する正メッセージ以降の処理済イベントを処理済イベントリストの末尾から順に各 IB へ戻す。当該送信元の分は消去する。コストも straggler の場合と同様である。

ヒープの更新を伴う場合はいずれもオブジェクト数もしくは IB 数 N に対して $O(\log N)$ のコストを要する。

5.6.4 マルチリストとの比較

マルチリストのデータ構造 マルチリストは複数のリストを別の共通のキー値を基に融合させたものと見做すことが出来、図 5.10下のような構造を取る。

図の上半分は CC のヒープと LVT のヒープによる等価な実現である。マルチリストは一般に再帰的に何重にも組むことができる。図ではタイムスタンプ順のリンク、同一 IB のリンク、同一 LP のリンクによる 3 重マルチリストの例である。headObj[*dst*] から上側のリンクをたどることにより宛先 *dst* 宛のイベントリストにアクセス出来る。通常の IQ に相当する。headIB[*dst*][*src*] から下側のリンクをたどることにより LP 番号 *dst* の送信元 *src* に対応する IB の要素に順にアクセス出来る。

マルチリストへの挿入 / 削除操作は多重度数だけのリスト更新操作が必要である。

WARPED では IQ に 2 重マルチリストが使われており、LP 毎のリンクとタイムスタンプ順のリンクの 2 重構造を取り、LP 間スケジューリングの機能を併せ持っている。つまり、タイムスタンプ順のメインリストはスケジューリングに、LP 毎のサブリストは IQ としてロールバック処理等に用いられる。

WARPED ではトポロジ情報を仮定しないが、トポロジ情報を生かして IB に相当するリンクを追加した図のような 3 重マルチリストの構造を採用することも考えられる。

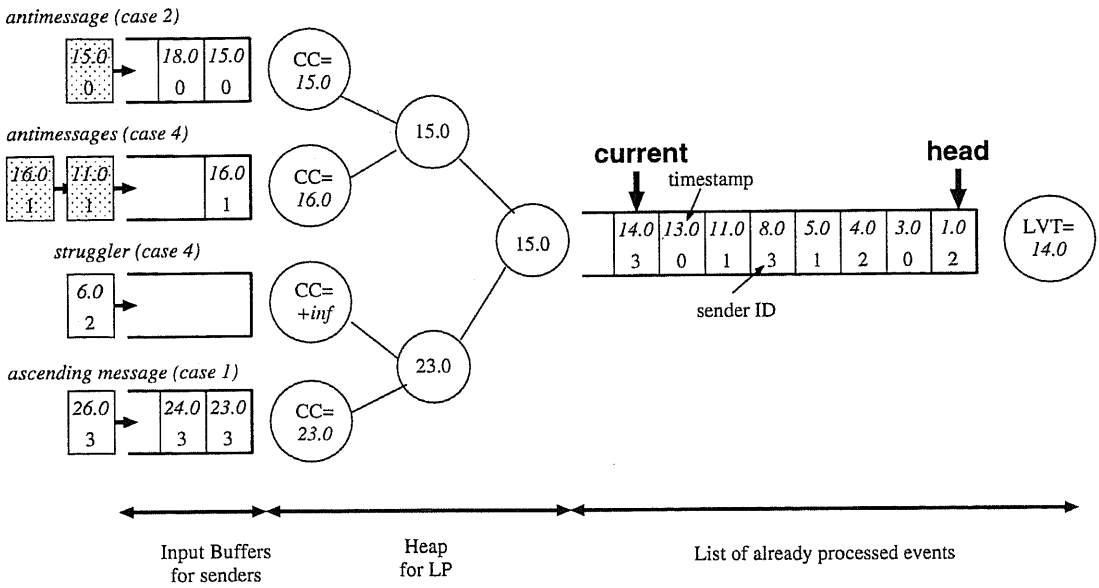

図 5.9: IB 間ヒープと LP 間ヒープの組み合わせ

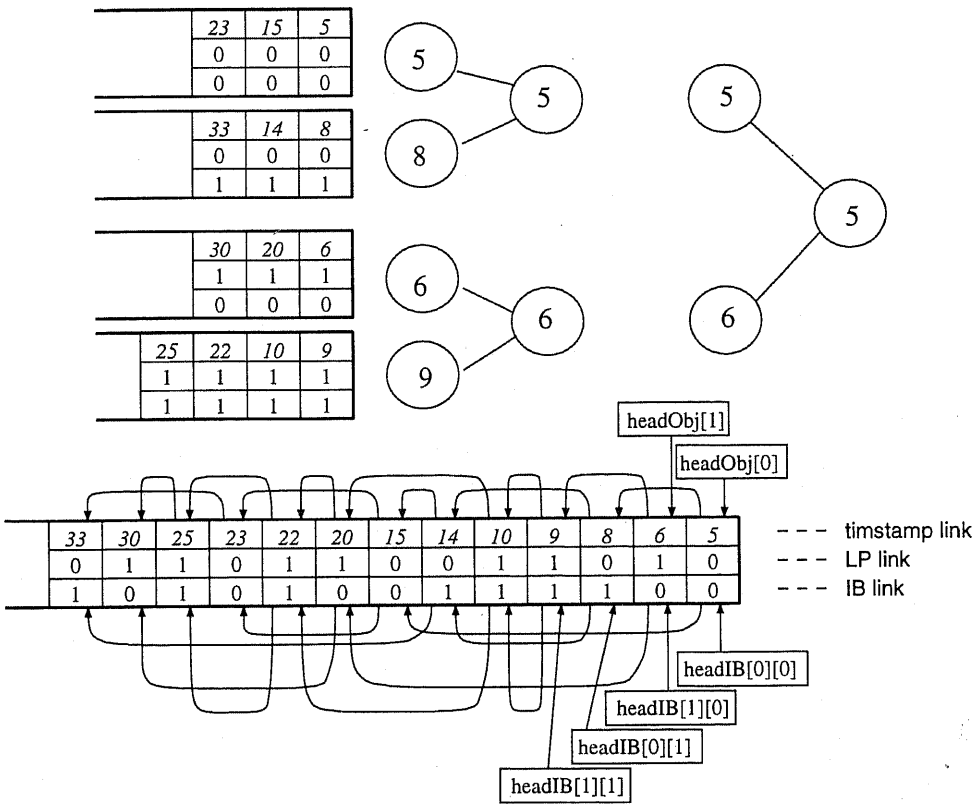

図 5.10: 3 重マルチリスによる Time Warp プロセススケジューリングとヒープによる実現との比較

更に、IB のマルチリストと LP のヒープといった組み合わせを取ることも可能である。ここでは IB に分けない場合と分ける場合に対応して 2 重マルチリストと 3 重マルチリストの場合について議論する。

- 分けない場合

  WARPED の実現と同じであり、メインリストへの挿入に LVT からリスト中の挿入地点までの探索コストがかかり (PE の担当する LP 数にも比例)、サブリストの挿入にも同様のコストがかかる。送信元毎に LVT の進捗が異なる場合は到着するイベントの時刻に偏りが生じ、挿入個所が分散してスラッシングが起る可能性がある。

- 分ける場合

  IB 毎のタイムスタンプ順リスト、LP 毎のタイムスタンプ順リスト、PE 全体のタイムスタンプ順リストの 3 重構成のマルチリストになる。送信元毎の到着順パターンを生かすことが出来る。

5.6.5 まとめ

本節では、前節で提案した、任意の要素の優先度の更新が可能なヒープデータ構造が楽観的プロトコルにも適用可能であることを示し、その実現方式を示した。既存の方式であるマルチリストによる実現は、最優先度の要素の削除が高速である一方挿入処理の負荷が大きくなる可能性がある。ヒープの適用の有効性については様々な外部イベントの到着パターンでの比較により検証する必要がある。

第 6 章

論理プロセス移送による通信の最適化

PDES プログラムは外部イベントの交換を頻繁に行うため極めて通信集約的であり、LP 間通信が全体の性能に与える影響は大きい。更に、LP 間の通信パターンと、物理プロセッサ (Physical Processor;**PP**) 間の結合トポロジとの間で整合が取れていないと、PP 間の結合ネットワークに対する負荷を増大させ、性能低下を招く場合がある。例えば 4 個の PP a,b,c,d が一次元結合されていて、その内側の二つに割り当てられている LP が互いに頻繁に外部イベントを交換している場合、両端の PP に割り当てられている LP 同志の通信は最高性能が出せるだろうか (図 6.1)。

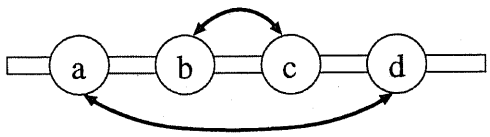

図 6.1: 衝突の起こる通信パターン

無論、今日の並列計算機の相互結合網はこのような一次元結合ばかりではなく、直接結合されている PP の組がなるべく大きくなるようにトーラス、ハイパーキューブ、ハイパークロスバ等が多く採用されている。本論文の主な実験環境である SR2201[52] は 3 次元ハイパークロスバネットワークを採用しており、3D トーラスやハイパーキューブ等に比して高性能である [53]。しかし、PP 数が大きくなると特に、PP 間を直接結合することが困難になる。よって、図 6.1のような衝突は不可避であり、また、目的の PP へ到達するために必要なホップ数も増大することになる。

このように本質的に PP 間通信の衝突は完全にハードウェアで解消することは困難であり、ソフトウェアレベルで PP 間結合網と LP 間の通信パターンとの整合を取る必要が生じる。しかし、PDES は科学技術計算で多用される行列計算等と異なり、通信パターンは多様であり、最適なマッピングを見つけることは困難である。例えば、実際の IP ルータでは、入力されるパケット数のポート間の偏りは時間とともに変化するであろう。

更に、LP 間の結合トポロジにより通信パターンの偏りの情報が得られても、そのトポロジの直感的なイメージと、実際のトポロジとは必ずしも一致しない。例えば、3 段 Banyan-Switch ネットワークは一見平面的な構造を要素として成り立っており図 6.2上のように 3 次元にマップすることによって最適な PP 間通信パターンが得られるように思われるが、実際の結合トポロジは図 6.3のように王冠型をしている。

PP 間結合ネットワークも同様で、ハイパクロスバネットワーク上で 3×4 個の PP 群を XY 平面上に割り当てられた場合の結合トポロジは図 6.4であり、$2 \times 3 \times 2$ 個の PP 群を 3 次元的に割り当てられた場合の結合トポロジは図 6.5のようになっている。ハイパークロスバネットワークでは、同一

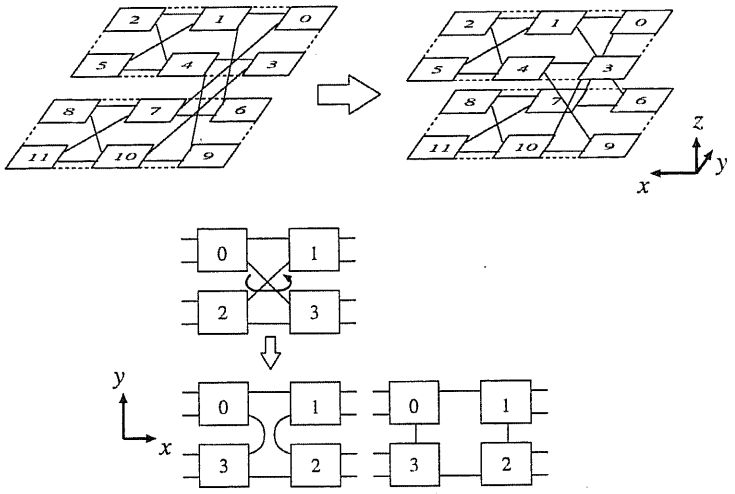

図 6.2: Banyan Switch のマッピング

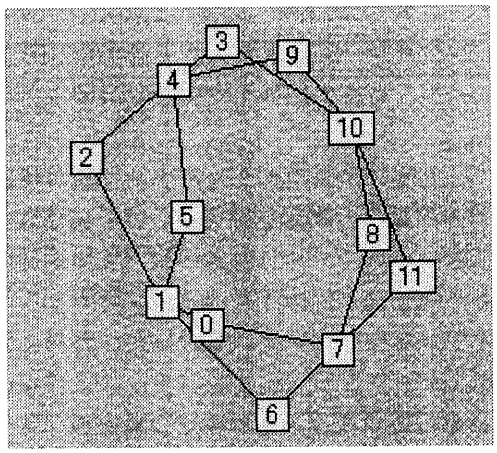

図 6.3: 3 段 Banyan Switch のスイッチ間結合の近接性の表現

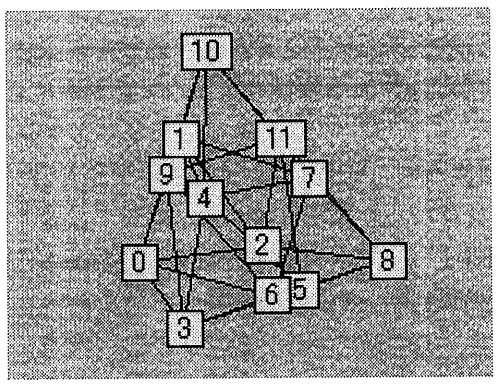

図 6.4: 3×4 のクロスバネットワークのノード間近接性の表現

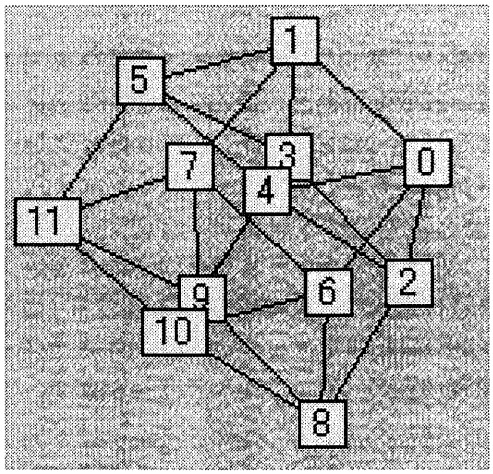

図 6.5: 2 × 3 × 2 のクロスバネットワークのノード間近接性の表現

座標軸上の PP 間はクロスバスイッチで直接結合されているために無理に隣接 PP にマップすることは無意味であることが分かる。

一方、今日の並列計算機では SR2201 のようにマルチユーザ対応のため与えられる PP 集合が実行毎に異なる場合があり、コンパイル時には予測が不可能であり、ユーザが明示的に PP 集合を指定するのも単純ではない。

更に、最適なマッピングの求解にはアーキテクチャの通信性能やアプリケーションレベルでの通信と計算の粒度の関係等複雑な要素が関与し、解析的な求解は困難である。

従って、LP 間トポロジ等の要因で PP 間の通信パターンに強い偏りがある場合はマッピングの決定は実行時に適応的に行われることが望ましい。実行時に LP を何らかの規則に基づき移送し、自らの性能の変化を監視し、現実的な時間の内に性能の最も良いと推定される動作点を探索し、その動作点で実行を続け、二回目以降の実行では、そのようなマッピングから開始して、必要に応じて構成を変えることを通して実行効率を常に最適値に近づけることが可能になると考える。

本章では、このような移送の基準となる PP 間網への負荷をモデル化し、移送アルゴリズムの提案を行う。計算負荷は均等に割り当て済であることを前提とし、移送は LP の交換で行う。LP を実行時に移送可能にすることにより、LP の分割だけしておき全部を単一の PP 上にマップし、それから少しずつ他の PP へ移動させることも可能になり、ユーザがマッピングを考慮せずに済むようにすることも出来る。

SR2201 の採用しているようなハイパクロスバーネットワークでは、軸の乗り換え回数が主なコストとなるが、本枠組みにより 2×2 単位スイッチの 4 段構成の Banyan-Switch 網を単純にマップした場合の乗り換え回数を 1/3 に削減することが出来た [54]。

LP の移送に関する研究は既に行われている [55, 56, 57] が、PP 間網との整合性という文脈での議論や、ユーザレベルオブジェクト移送を可能にする際の問題点への言及は見られない。

本章では以下、LP 間通信と PP 間通信の整合の問題の定式化、PDES での最適化の可能性、移送を可能にするための枠組み、ハイパークロスバ網への適用例、関連研究、今後の課題について順に述べる。

6.1 LP 間通信と PP 間結合網の不整合により生じる問題

一次元的に結合されている PP 間の通信の衝突は図 6.1に示した通りであるが、次数のより高い相互結合網においても、直接結合されていない PP 間の通信同士が途中経路を共有することによる通信リンクの輻輳や、中継ノードの輻輳が起りうる。図 6.2下は、4 × 4 ポートの Banyan Switch を単位スイッチと LP を対応させて PP に一つずつ割り当てた場合の通信パターンの違いを示している。PP 間がハイパクロスバネットワークで結合されている場合でも、単位スイッチ 2,3 を担当する LP の割り付けを交換する前は斜めの結合のための単位スイッチ 0 と 3、1 と 2 の通信にそれぞれ単位スイッチ 1、3 をシミュレートする PP を経由した 2 ホップの通信が行われ、通信の遅延と単位スイッチ 0 と 1、2 と 3 をシミュレートする PP 同士のリンクでの通信の衝突が発生する。単位スイッチ 3 と 2 のマッピングを交換すると、全通信が 1 ホップとなり、経路の衝突も解消させることが出来る。

6.2 PDES における最適化の可能性

前節で述べたような LP の交換による通信の最適化が可能であった背景には、以下のような PDES の特徴を挙げることができる。

処理単位のモデルの定義が明確である 並列動作するサブモデルは PDES では LP という明確な単位に分割されており、特に C++ 等のオブジェクト指向言語による実装ではクラスとしてまとめて扱われることがほとんどである [17, 21, 23]。この場合、移送を行う際に関連するデータ構造をまとめることも比較的容易である。

静的な通信パターンの情報が得やすい 特に保守プロトコルでは LP 間の結合トポロジはシミュレーションカーネルに予め知られている必要があり、その情報から特定の PP 間通信の有無を決定することが可能である。

アプリケーションレベルの通信を捕捉しやすい アプリケーションレベルでの LP 間通信の記述は外部イベントの送出によるものになるため、直接 PP 間通信を起動することはなく、そのカーネル呼び出しを捕捉することにより PP 間通信パターンを実行時に解析することが可能である。

6.3 LP の移送を可能にするための枠組み

本章で提案する LP の移送による通信の最適化を可能にするためには、移送単位となる LP の構造の直列化、移送の前後でシミュレーションをデッドロックさせることなく中断 / 再開させる機構、LP の配分を流動的に出来るようなマッピングの記述が必要である。本節ではこれらについて順に述べ、最後に本論文で実装された WARPED 互換の保守カーネルを用いた事例について述べる。

6.3.1 論理プロセスデータ構造の直列化

LP はイベントリスト、状態リスト、スケジューラ等の動的な構造を多用しているため、移送に際してはそれ等を何らかの順序で連続領域に並べ直す直列化の処理が必要となる。

直列化自体は既に多数議論されており [58]、Java のようなデータにタグ付けが可能なプラットフォームでは言語仕様と密接に関連した形で安全に定義されている [59]。ここではそのような定義のない C++ を用いた実現方式を述べる。

動作原理

C++ のオブジェクトの、クラス単位での直列化を行う。一つのインスタンスを直列化しようとすると、それが参照しているオブジェクトも全て直列化されるように、再帰的な呼び出しを行うような枠組みになっている。

直列化されたデータは動的に確保された連続記憶領域を内部で持つ Packet クラスのインスタンスに格納される。

循環のある構造を正しく維持するために、一度直列化したオブジェクトはハッシュテーブルに登録し、2 度目以降に出会った際は、その直列化を抑止し、同じオブジェクトが重複して直列化されることを防ぐ。

主なインタフェース

直列化 / 復元インタフェース

- `void ser(Packet *pkt, Serializable **old_address, int flag)` — 直列化 / 復元関数 (グローバル)

  flag == 1 で直列化、flag == 0 で復元を行い、復元の場合は *old_address に新しいインスタンスのアドレスを書き込む。

  直列化の場合はポインタ *old_address を手がかりに直列化済かどうか調べ、そうでなければそれをハッシュテーブルに登録し、*old_address の指す領域を pkt へ書き込み、参照メンバの直列化 / 復元補助メソッド ser_aux を呼び出す。

  復元の場合は、ストリーム pkt に格納されているデータを基に循環を含むオブジェクトのデータ構造を復元する。直列化前にインスタンスが格納されていたアドレス *old_address を手がかりにする。ストリーム pkt に含まれているオブジェクトのアドレスは全て直列化時にハッシュテーブルに登録されているため、先ず *old_address をキーにしてそのテーブルを検索して、復元済かどうか調べる。もし未復元なら、復元したインスタンスを格納する領域を新たに確保し、ストリーム pkt から自分のインスタンスの内容をコピーする。そして新しいアドレスと、復元済を示すフラグを先ほど引いたハッシュテーブルのエントリに設定し、復元されたインスタンスの各参照メンバに対して再帰的に復元を行うべく ser_aux メソッドを呼び出す。

- `void Class::ser_aux(Packet *pkt, int flag)` — 参照メンバの直列化 / 復元 (仮想)

  各クラスが責任を持ってメソッドとして定義する。自分の参照する他のオブジェクトの直列化 / 復元関数を再帰的に順次呼び出す。スロットへの参照を引数として再帰的に ser 関数を呼ぶことで実現出来る。インスタンスの領域全体は ser 関数の方で保存 / 回復するが、ser_aux 関数は仮想メソッドとして定義され、最大派生クラスのものしか呼ばれないため、そのクラスで拡張された参照スロットの直列化 / 復元を行う前に、(もしあれば) 基本クラスの ser_aux を呼ぶ必要がある。

- `int Class::getSize(void)` — サイズ問い合わせメソッド (仮想)

  各クラスが責任を持ってメソッドとして定義する。各クラスのインスタンスの大きさを返す。常に`return(sizeof(*this))` で実現出来る。

ストリーム形式

- `void Packet::write(void *source, size_t size)` — 書出し

  source で指される領域から size で示される長さだけのデータをストリーム (Packet クラスのインスタンスが内部に持っている動的に確保された連続領域) にコピーし、currentp ポインタ (次に読み書きが始まる位置) をその大きさだけ進める。書き込み領域が不足すると、その領域を、一定量伸長する。

- `void Packet::read(void *target, size_t size)` — 読み込み

  Packet 内のデータ領域の現在位置から長さ size 分のデータを target で指される位置から書き込み、現在位置ポインタ currentp を size 分だけ進める。

ユーザの記述形式

任意のクラスを直列化可能にするためには、以下のように Serializable クラスから継承するようにし、上記のサイズ問い合わせ関数、直列化 / 復元補助メソッドを定義する。

逐次化 / 復元手続きを同一の手続きで統一していることにより、適用順序が両手続きで一致していることが保証され、それによるストリームデータフォーマットの簡略化が実現している。

```
 1:// UserClass.hh のクラス定義の前に Serializable.hh をインクルード
 2:#include "Serializable.hh"
 3:
 4:// Serializable クラスを継承してクラスを定義
 5:class UserClass : public Serializable {
 6:  int getSize(void); // インスタンスのサイズを返す関数の宣言
 7:  void ser_aux(Packet *pkt, int flag); // 直列化 / 復元補助メソッドの宣言
 8:
 9:  // 以下通常のメンバの宣言
10:
11:};
12:
13:// サイズ問い合わせ関数の実装
14:int UserClass::getSize(void) {
15:  return sizeof(*this);
16:}
17:
18:// 直列化 / 復元補助メソッドの定義
19:void UserClass::ser_aux(Packet *pkt, int flag) {
20:  SuperClass::ser_aux(pkt,flag); // 基本クラス (もしあれば) への適用
21:
22:  // 各参照スロットに対して ser() 関数を適用
23:  ser(pkt,(Serializable **)&slot_1,flag);
24:  ser(pkt,(Serializable **)&slot_2,flag);
25:  ser(pkt,(Serializable **)&slot_3,flag);
26:}
```

図 6.6: C++ オブジェクトの直列化インタフェース

本方式では辿りうるポインタは全て辿って行くため、実際に直列化する場合には、事前に関係のないオブジェクトへのリンクを切り離す必要がある。また、C++ クラスによる実現を前提としている

ため、C++ オブジェクト以外の通常の C 言語のデータ構造の直列化は提案する枠組みでは対処出来ず、個別の直列化、復元の手続きを準備する必要がある。

転送前後でアドレスの意味が違ってくることにも注意が必要である。

6.3.2 シミュレーションの停止 / 再開機構

LP の移送の前後では、LP の PP へのマッピングが変り、外部イベントのルーティングも変るため、その移行を安全に行うことが必要である。外部イベントはアプリケーションレベルでは相手の LP を指定して起動され、具体的な宛先 PP を割り出すのはカーネルの役割である。移送の前後で発行された外部イベントの行き先が正しく変更されるようにする必要がある。

また、LP の移送処理も分散的に行われるためデッドロックを起こさない工夫も必要である。

これ等の処理は必然的にシミュレーションの中断を伴うために、なるべく効率よく遂行されなければならない。

6.3.3 マッピング記述の流動化

マッピングが変化することにより場合によっては PP 内の LP 数も変化する場合がり、特にスケジューリング機構等では、実行時に対象となるレコード数が変化しても良いような設計にする必要がある。よってマッピングの変化に影響を受ける部分を明確に分離することにより LP の配分を流動的に出来るようなマッピングの記述が必要である。

6.3.4 実装した WARPED 互換の保守カーネルによる事例

本節では、本論文で試作した WARPED と同等のインタフェースを持つ、基本的な CMB プロトコル [11] による保守カーネルに対する適用事例について述べる。

カーネルの概要

本カーネルは WARPED 同様、LP に相当する SimulationObj クラスを継承してユーザレベルの LP を定義し、PP 毎にひとつ設ける LogicalProcess クラスに登録し、LogicalProcess::simulate() メソッドによりシミュレーションを開始する。内部では LogicalProcess 毎に LP のスケジューリングを行う SchedRec クラスのインスタンスを保持し、内部では 5章で提案したヒープを持ち、更に LP 毎に IB を表現する IBuf クラスのスケジューリングを行う ObjRec クラスのインスタンスを保持し、IBuf のスケジューリングを行うヒープを持っている。

マッピング情報の表現

LP から PP への割り当てに関する情報は、本保守カーネルも WARPED に準じた形式で保持している。LogicalProcess クラスおよび SchedRec クラスでそれぞれ PP 内 LP 数、PP 内の各 LP と、LogicalProcess 間の相互参照配列、またイベント構造体も送受信 PP 番号を保持するスロットを持っている。

移送のシーケンス

動作状態でのカーネルは、WARPED と同様 LogicalProcess::simulate() 中で外部イベントの受信、スケジューラによる LP の選択、LP 中のスケジューラによる次にイベントを取り出す IB の選択、イベントの実行を繰り返すメインループを実行している。移送の際はこのメインループからの離脱と

復帰を行わなければならない。更に、未送信のメッセージを正しく処理するために移送の前後で一旦通信路をクリアしなければならない。また具体的に何を移送するのかを決める必要がある。本節では上記を踏まえた具体的な移送のシーケンスを示す。簡単のために転送の意思決定自体はひとつの PP(ID=0) による集中制御で行う。

(1) 転送決定 (ID=0)

(2) LogicalProcess::simulate() 中のメインループから脱出 (ID=0)

(3) PP 間通信チャネルのクリア

- PrepareMigrateMsg メッセージを全 PP に放送し、自分は外部イベント受信のループに入り、MigrateAckMsg を他の全 PP から受信するまで待つ (ID=0)
- PrepareMigrateMsg を受信することを契機に、MigrateAckMsg を返信し、外部イベント受信のループに入り、StartMigrateMsg を受信するまで待つ (ID !=0)

(4) MigrageAckMsg が全部受信出来たら、StartMigrateMsg を放送する (ID = 0)

(5) StartMigrateMsg を受け取ったら転送する LP を直列化し、LPMigrateMsg に包装して宛先へ送信し、LP_0 へ MigrationCompletedMsg を送り、自らは受信状態になり、RestartMsg を受信するまで待つ (ID!=0)

(6) MigrationCompletedMsg を全 PP 分受信したら、RestartMsg を放送する (ID=0)

(7) RestartMsg を受信したら、受信した LP を復元して、LogicalProcess::simulate() メインループに復帰する

図 6.7: LP 移送アルゴリズム

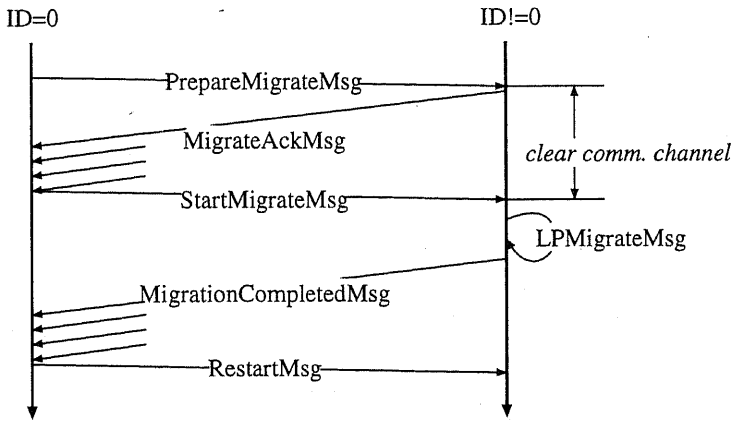

図 6.8: LP 移送のシーケンス

以上の場合は LP の移送の例であったが、PP に集合化された LP 群単位での一括転送の場合は LogicalProcess クラスと LP との対応関係が変らない分実装が容易である反面、カーネルより上位のレベルでの制御が必要である。

6.4 LP 間通信と PP 間結合網の不整合性の表現

6.4.1 基本概念

LP 間通信の性能を低下させる原因はマップされた PP 間の相互結合網上の距離による遅延と、PP 間の経路の競合による遅延に分けることが出来る。前者はマップされた PP 間の距離に比例したコスト、後者は PP 間経路への負荷として捉える。それ等のコストは実際の通信量で重み付けされる。以下では、それ等のコストの表現法について述べる。

6.4.2 コスト関数

PP 間距離 直接結合されている PP 同士は最速で通信出来るが、そうでない PP 通信はホップ数に応じたコストが加算される。トーラス網とハイパクロスバ網では、送受信端の座標のうち異なる組の数だけのホップ数となる。ハイパーキューブ網では送受信ノード番号間のハミング距離となる。

経路の競合 トーラスネットワークでは同一軸上での PP 間通信が複数あるとリンクの衝突が生じ、複数の PP 間通信がその両端の PP で座標を共有していると中継ノードでの衝突が生じる。ハイパクロスバ網では、二つの送受信 PP の組が同一軸上に存在する 3 次元にまたがる通信の場合第 3 の軸上での経路でリンク衝突が生じ (図 6.9)、2 次元平面内での通信でも両端がそれぞれ同一軸上にある場合中継ノード上での衝突が起る。

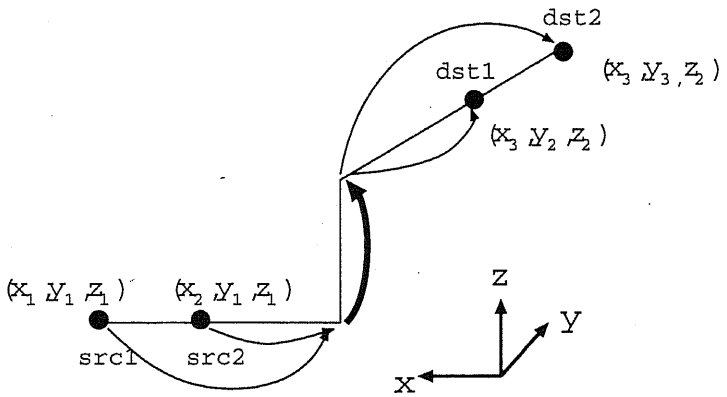

図 6.9: 3 次元クロスバ網におけるリンクの競合

コスト最小化の尺度 上記の競合や距離を、転送された外部イベントメッセージの長さで重み付けをして割り出された距離、リンク負荷、ノード負荷に関するコストの総量を最小化するアプローチと、それぞれの平均からの偏差を最小化するアプローチが考えられる。並列処理の場合最も遅い処理が全体の性能を低下させるため偏差を最小にするアプローチが有効であると考える。

通信集約度の影響 図 6.10の例では、二つの矢印で表現される通信が経路を共有することによりノードとリンクで輻輳を起こしている様子を示しているが、通信が閉塞型で実現されている場合、受信側が受信を完了するまで送信関数は復帰出来ないため次の通信を発行することが出来ず、時間軸上の重なりは存在しないため、右のパターンでの通信の競合も起らないと考えられる。

このことは通信の時間軸上の重なり合いの度合いで表現出来、アプリケーションが通信集約的でない場合は衝突の効果は減少し、距離の効果だけが問題となる。逆にカーネルが message aggregation による最適化を行っていて通信レイヤのメッセージ長が長くなると衝突による影響も大きくなると考えられる。

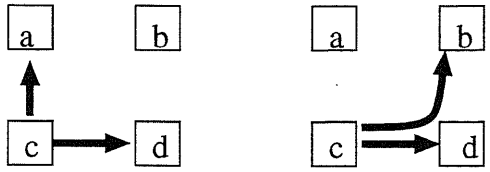

図 6.10: 閉塞型 / 非閉塞型通信の差異による競合の有無

6.5 LP 移送アルゴリズム

本節では、具体的にどの LP をどの PP に移送するかを決定するアルゴリズムについて述べる。

本章では LP の計算負荷の均等化は考慮しないため、計算負荷は既に均等化されていると仮定する。よって LP の移送は計算負荷を壊さないように LP の交換に基づく方法を取る。また、マッピングの種類は全部で (LP 数 +PP 数) の階乗を PP 数の階乗で割った数だけ存在し、全てを考慮するのは非現実的であるが、PP 内 LP の集合を 1 単位として考えて交換を考えると場合の数は PP 数に対して多項式オーダに減少させることが出来る。但し最適解が得られるとは限らなくなる。交換は一組ずつコストの減少に向けて漸進的に行い、コストの減少が見られなくなった時点で終了させる。

アルゴリズム　基本的な枠組みは以下のようである。

1. 各 PP は隣接 PP 同士で、互いに LP を交換するとコストがどの程度変化するか計算する。
2. 全 PP で上記の改善値が最大となっている PP 間で LP を交換する。
3. 上記の操作を改善値がある閾値を下回るまで繰り返し行う。

各 PP でコスト改善値を並列に計算出来るため、改善値の計算量は PP 間のリンク数が小さい場合は低く抑えられるが、交換の候補が限定されるため最適解を得にくくなると考えられる。

そこで、交換の候補の絞り込みの別のアプローチとして、互いに通信する LP 同士を PP 間トポロジ上で近くに配置するために必要な移動についての表現を与え、互いに入れ替わりによってコストを減少させる PP の組で LP を交換することも考えられる。この場合、移動する必然性のない LP は計算から除外することができる。距離が近いほど経路上のノードやリンク数も減少させることが出来るため、通信相手との距離に応じた移動への意思表示をさせることは有益である。

ハイパクロスバ網の場合、LP は通信相手の LP のマップされている PP と同一軸に乗ると距離が小さくなるため、移動先候補は通信相手の軸上に限定される。乗り換え数 0 の通信リンクだけを持つ LP は入れ替え候補から除外される。

6.6 ハイパークロスバネットワークへの適用例

本節では、並列計算機 SR2201[26] のネットワークアーキテクチャへの適用について述べる。

6.6.1 SR2201 の相互結合網

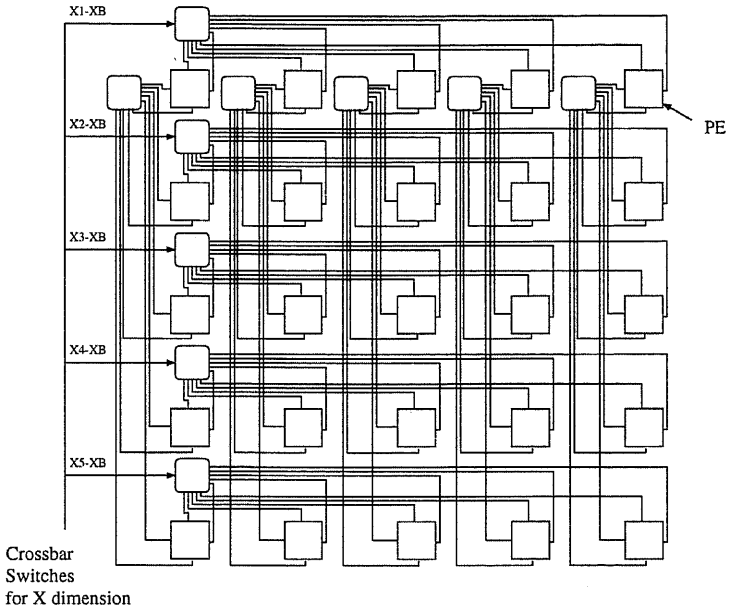

図 6.11: SR2201 のクロスバネットワーク (5×5PE の二次元構成の模式図)

SR2201 の相互結合網は図 6.11のようになっている [52]。プロセッサ間は 3 次元ハイパークロスバー網で接続されている。すなわち、各軸上のプロセッサはクロスバスイッチで直結されており、PP 間メッセージは各軸に沿って X 軸、Y 軸、Z 軸順にルーティングされ高々 3 ホップで完了する。同一軸上の PP 間通信に関しては衝突は起らない。X 軸は 8 プロセッサ、Y 軸は 16 プロセッサ接続されており、Z 軸に関して最大 8 平面まで拡張出来、最大 2048 プロセッサの構成を取ることができる。

図 6.4、6.5で示したように割り当てられた PP 間集合のトポロジの次元が少ない (直線形に近い) ほど次数が大きくなる。距離は軸の乗り換えに起因するもので、経路 / 中継ノードの衝突による輻輳も起りうる。

6.6.2 乗り換えコストと通信衝突の実験結果

軸の乗り換えの有無による通信性能の影響を測定した結果を図 6.12に示す。

図は MPI による閉塞型 1 対 1 往復通信 (MPI_Send と MPI_Recv) を XY 平面上に確保した 2 × 2 個のプロセッサ間で行った場合の実行時間であり、60 万回 4 バイトのデータを交換した。プロセッサは X 座標優先で順に並んでおり、横軸は左上の 0 番の PP と通信を行った PP 番号であり、PP 番号 1 との通信は乗り換えなし、PP 番号 3 との通信は乗り換え有りの場合である。実行時間にばらつきがあるが、乗り換えのある通信の方が遅くなる傾向が確認された。

図 6.13および表 6.1は、それぞれ 2 次元平面上のプロセッサ間通信におけるノード衝突に関する影響を調べるために 2 万の長さの整数配列を 5 万回 1 対 1MPI 通信により単方向に送受信した場合の通信パターンと総実行時間の関係を示す。総実行時間に占める実際の伝送時間を大きく取るためにメッセージのサイズを大きくして測定している。

SR2201 では X 軸優先のルーティングが行われ、パターン 1 とパターン 2 では、3 から 2 への通信が 5 を経由するため輻輳が生じていることが、パターン 3 と 4 ではノード輻輳が生じていないことが

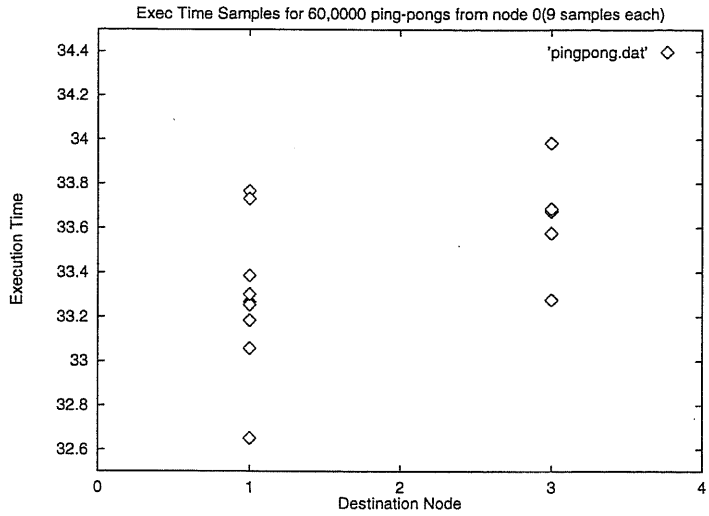

図 6.12: 乗り換えコストに関する実験結果

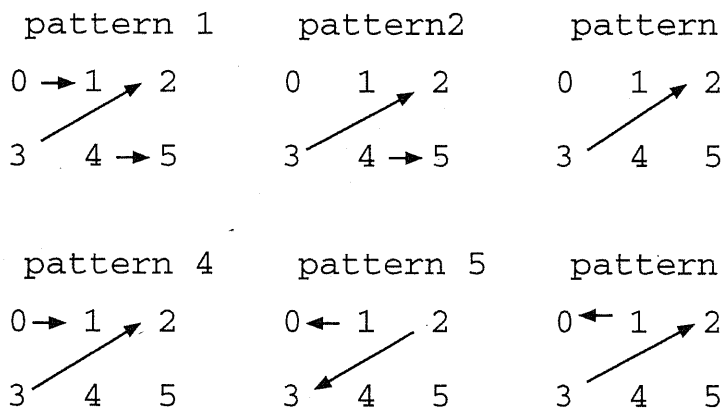

図 6.13: ノード衝突コスト測定に用いた通信パターン

通信パターン No.	総所要時間 [秒]
1	約 30 秒
2	約 30 秒
3	約 24 秒
4	約 24 秒
5	約 30 秒
6	約 24 秒

表 6.1: 通信衝突実験における総所要時間

分かる。また、パターン 5 とパターン 6 はノード 0 での輻輳が生じると思われるが、2 から 3 の通信と重なる通信の方向が逆であるパターン 6 の方は衝突の影響が出ていないことが分かる。

6.6.3 Banyan-Switch ネットワークシミュレーションにおける最適化例

本節では 2 × 2 単位スイッチの 3 段構成による Banyan-Switch 網の単位スイッチ単位を LP に割り当てた場合の最適化例について述べる。

リンク負荷の平均自乗誤差の最小化に基づく方法 PP 数の自乗に比例した全ての LP のペアに関して交換前後でのコストの比較に基づく交換アルゴリズムによるマッピングの遷移を図 6.14に示す。各 PP 間リンクにつき LP 間の通信リンクをもつ場合コスト 1 を加算している。全体のコストは各リンク上のコストの平均との差の自乗の合計である。図はランダムなあるマッピングから開始した場合である。

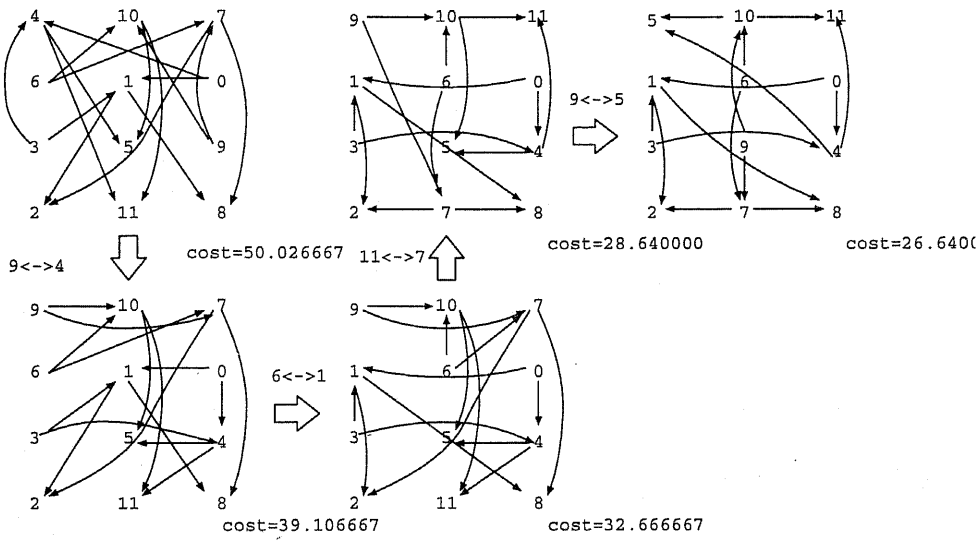

図 6.14: リンク負荷の平均自乗誤差の最小化

ノード間距離と乗り換えコストを考慮した例 コスト関数として、

$$\left(1+\frac{\text{各軸の座標の差}}{10}\right)\times \text{通信量}\times 1.5^{\text{軸乗り換え数}}$$

を選び、入れ替え候補を隣接 PP 上の LP に限定し 6.5節のアルゴリズムを適用した場合のコストの収束について図 6.15、6.16に示す。

この場合も乗り換え回数が小さく抑えられることが分かった。特に 4 段構成の場合は通常描かれる図の通りに 2 次元にマップした場合の初期の乗り換え回数が 32 回に対して交換を繰り返すことにより 8 回に抑えることが出来た。

入れ替え候補を隣接ノードに限定しているため入れ替え候補数はノード数オーダに抑えられるが、収束後のコスト値が初期マップにより異ることが分かる。従って、計算量を増加させずに初期マップに拠らない最適マップを得るには 6.5節の最後で述べたように、トポロジを考慮した自立分散アルゴリズムによる各ノードの自主的な転送先の絞り込みが必要である。

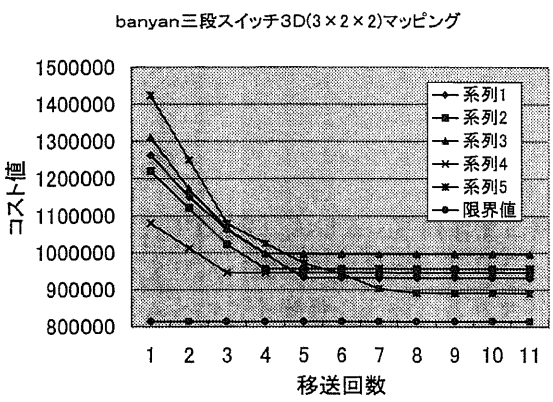

図 6.15: 3 段 Banyan ネットワークのコスト値の遷移

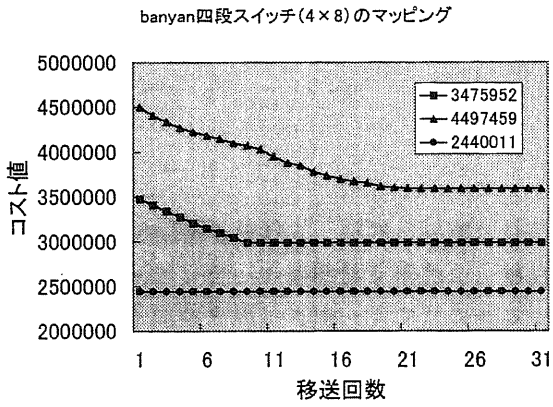

図 6.16: 4 段 Banyan ネットワークのコスト値の遷移

6.7 関連研究

LP の移送自体は国際会議での募集分野のひとつとして挙げられる程、重要性が認識されているテーマである。本章での提案方式は、アプリケーションレベルでの同一 PP に割り当てられる LP の集合の間の通信を変えず、PP 間相互結合網のトポロジとの整合性を取る。

文献 [56] では、LP ではなくプロセスごと移送する方式を採用している。先ず各プロセスのタイムスライスを動的に変化させることによりプロセス間の VT の進行の足並みを揃えようとする。タイムスライスを変更した結果負荷バランスが崩れた場合には、プロセスをアクティブなまま別のプロセッサへ移送するという方針を取る。アプリケーションレベルでのユーザの関与が不要であり簡単であるが、実際に並列計算機上の UNIX プロセスが移送可能である状況は多いとは言えない。

本章の提案方式は OS がプロセス移送のサポートをしている必要はなく、また移送の方針も純粋に通信負荷の観点から決定される点が異なる。

文献 [60] では、実行時データ収集による負荷均等化方式を提案している。現在は 1 回目のパイロット・ランで全 LP ペア間のイベント通信数、全 LP のイベント処理数をファイルに出力し、2 回目以降の実行ではそれらのデータに基づいてマッピングを行う。実行時移送は実現しておらず、PP 間トポロジも考慮されていない。

文献 [55] では、保守プロトコルにおいて IB 長の均等化に基づく LP の移送の枠組みを提案している。負荷の指標は系全体の IB 中の外部イベントとヌルメッセージの合計に、それぞれの処理時間による重み付けを行ったものである。これを均等化するように、負荷の大きいプロセッサ上の負荷の大きい LP を負荷の小さいプロセッサに送る。この際、通信オーバーヘッドを考慮するために、移送対象となる LP は通信相手を最低 r 個持っており、かつそのうち最低一つが移送元とは別のプロセッサにマップされていることを条件にして選択されている。転送のスラッシングを防ぐ為、経験的に得られた *torelance* という指標を決め、その範囲内に負荷バランスが収まっていれば転送をしないという制限を加えている。

負荷バランスの計算そのものを行うプロセッサは別途用いることにより負荷計算そのもののオーバーヘッドを回避している。

文献 [57] では、タイムワープによる論理シミュレータにおける動的負荷均等化について述べている。LP をクラスタ毎にまとめて、クラスタごと移送する方式を採用している。移送の指標として、単位時間あたりのロールバックされなかったイベントメッセージ数を用いて負荷均等化を図っている。

本章の提案方式はそれぞれの関連研究で提案されている方式による負荷均等化の後に適用することにより、アーキテクチャとの整合を取ることが出来るという点で、互いに補完し合うことが出来ると言える。

6.8 まとめ

本章では、PDES における LP 間通信と PP 間結合網の整合性の問題を提起し、移送に必要な枠組みとハイパクロスバ網への適用例を示した。LP 間移送機構はマッピングへの柔軟性を提供し、移送アルゴリズムとコスト関数の分離によりアーキテクチャ毎のコスト関数を用いて同一アプリケーションの他のアーキテクチャへの適用が容易になる。

今後の課題は以下のようである。

他のトポロジのためのコスト関数 トーラス、ハイパーキューブ網に対するコスト関数を同様に定義することが出来る。次数が低いネットワーク程、経路の衝突も多くなり、本アルゴリズムの効果はより大きく出ると考えられる。

移送の枠組みの実装と、移送コストの評価 移送の枠組みと入れ替えアルゴリズムの連携部分が現在未実装である。また、直列化にかかる時間と直列化されたオブジェクトの転送、シミュレーションを一時停止させる時のコストの評価も行う必要がある。

コストモデルの詳細化 今回はハイパークロスバ網について主に乗り換えコストやリンク毎の通信量に重点を置いたコストモデルを用いたが、中継ノードの輻輳の考慮やチャネルの輻輳の考慮、MPI 以外の更に低レベルの通信ライブラリを用いることにより通信集約度が増したときの影響についても評価する必要がある。

第 7 章

統計処理

DES では、対象とする系を設計や評価に適した形で模擬し、効率よく実行するのみならず、実行途中の動作や設計 / 評価に必要な指標 (統計量) を、必要な精度と分析者が処理しやすい形式で出力する機能を備えていなければならない。特に PDES では、LP から個々に出力されるデータの効率よい収集、平均等所望の形式への変換が望まれる。

ところで DES は出力解析の観点からは、ある時点での系の状態を対象とする Terminating Simulation と、系の定常状態を対象とする定常状態シミュレーションに大別することが出来る。前者は終了時刻或はイベントが予め与えられるものであり、多くの PDES プラットフォームも予め終了時刻を与えるものである。一方、後者では系の定常状態での統計量が所望の精度で得られる時刻を予め決定することが出来ず、バッチ平均法等の精度判定機構としばしばそれによる停止処理が必要である。

逐次シミュレーションの分野ではこのような機能は提供されていることがあるが [5]、PDES ではこうした研究は稀であり、そのような機構を備えた汎用 PDES プラットフォームも筆者の知る限り存在しない。

複数の統計データの収集機構を PVM を用いて実現したものはあるが [6]、明示的に PDES の枠組みと結び付けられておらず、精度判定まで任せられる機構はあまり研究されていない。

このような統計処理を効率よく実現するには、シミュレーションモデルの実行と同様に計算と通信をうまく分散させる必要があるだろう。サンプル源となる全ての LP が外部イベントとして統計出力データを 1 個所に集中して送信するといった単純な実装では容易にボトルネックを形成してしまうからである。サンプルデータの収集機構には何らかの木構造を用いた通信の分散が適用可能である。

本章では上記を踏まえ、定常状態シミュレーションの統計出力解析手法のひとつであるバッチ平均法を PDES に組み込む枠組みを提案する。本枠組みは、ひとつのバッチを構成する出力データの収集方法の違い (仮想時刻に基づく手法とサンプル数に基づく手法) による精度の違いに関する検討、プロトコル毎の組み込み手法の相違、実装のレベルの相違 (OS のプロセスレベル、プラットフォームレベル、イベントレベル) に関する検討より成る。

これらの検討を踏まえて、対象とするモデルの性質、実装環境、用いるプロトコル毎に最適な設計のためのガイドラインを示し、必要な統計処理についての、共通部分の一般化、各種統計量と、対応する収集法の一般化、モジュール化を目指す。また、既存の PDES 汎用プラットフォームが統計処理を支援するために必要な改変についての提言を併せて行う。

7.1 シミュレーションと統計処理

定常状態シミュレーションでは、物理系の無作為的 (randomness) な振る舞いを十分長時間模擬することにより、系の興味のある指標に関する統計量を所望の精度で見積もる必要がある。シミュレーションで求められるそのような統計量としては基本的には次に挙げるものがある。

期待値 / 平均 系の何らか指標 Y の定常状態での平均値 $E(Y)$。例えば通信系のシミュレーションでの平均遅延やセル損失率がこれに当たる。

分散 指標 Y についてその変動の程度を見積もりたい場合、分散 $Var(Y) = E(Y^2) - E(Y)^2$ が或は妥当であろう。

確率 例えば設計した計算機システムのシミュレーションに於いて、「ジョブに対するシステム応答時間が 30 秒以内である確率」を知りたい場合。

分位点 / 確率点 例えばまた計算機システムで、「それ以下の時間で 90% のジョブが完了すると言えるような応答時間」を見積もりたい場合。

とりわけ期待値は多くの場合に興味の対象となる基本的な量であり、DES の出力解析の分野においてももっともよく取り上げられている。更に、分散と確率については期待値の推定法に変換できる [61]。そこで本研究でも系の何らかの指標の定常状態での平均値を並列環境で効率よくかつ十分な精度で推定する方法について考察する。

7.2 バッチ平均処理

7.2.1 期待値の推定

定常状態平均 ν はサンプル系列を $\{Y_i\}$ として、

$$\nu = \lim_{i \to \infty} E(Y_i)$$

あるいは現実的には

$$\nu = \lim_{m \to \infty} \sum_{i=1}^{m} Y_i / m$$

と定義される。その推定にはサンプル平均

$$\overline{Y}(n) = \sum_{i=1}^{n} Y_i$$

を点推定として用い、さらに精度を示すために通常 $Var(\overline{Y}(n))$ を見積もってその信頼区間を添える。$\{Y_i\}$ が独立かつ同一の正規分布に従う場合の信頼区間は、

$$\overline{Y}(n) \pm t_{\alpha/2,n-1} S(n)/\sqrt{n} \quad \left(S^2(n) = \frac{\sum^{n} (Y_i - \overline{Y}(n))^2}{n(n-1)} \right)$$

で与えられるが、この式をそのまま DES の出力に用いることはできない。$\{Y_i\}$ が正規分布しないことの他にこの式にそぐわない DES の大きな特徴が 2 つある:

過渡状態 (warm-up) の影響 実行初期の過渡状態は点推定としてのサンプル平均を偏らせてしまう。

出力サンプル系列の自己相関 普通連続するサンプル間には (しばしば正の) 相関がある。(特に待ち行列モデル)

過渡状態の影響を除くためには、実行初期のサンプルを捨てるか或は実行長を十分長くすることが必要である。Warm-up 期間の見積もりもまた簡単でない問題であり、そのためのアルゴリズムも提案されているが [62]、複雑で実装が容易でないなどから一度試験的な実行を行い出力をプロットしてみた上で切捨てるべき期間を決定することも多いようである。

一方、サンプル系列の自己相関の存在は $Var(\overline{Y}(n))$ の見積もりを大きく狂わせる。特に正の相関のある場合は実際よりも小さく見積もることとなり、不当に狭い信頼区間に結果してしまう。従って DES の出力解析に於いてはこの自己相関の影響を除くかその大きさを見積もり、正しく $Var(\overline{Y}(n))$ を推定することが主な課題となる。

7.2.2 バッチ平均法

サンプル系列に自己相関がある場合に平均値を推定する方法の一つにバッチ平均法と呼ばれるものがあり、概念的にも理解しやすく実装が容易で計算量も少ないためしばしば用いられる。そこで本研究でもこのバッチ平均法を並列環境に応用することを目指す。まずここでは、その手法を解説する。

まず、シミュレーションの出力解析では通常次の二つが仮定される。

Covariance stationary: サンプル系列の自己相関 $\rho_{i,i-j}$ がラグ j のみにより一定: $\rho_{i,i-j} = \rho_j$

ϕ-mixing: また、離れたサンプル間の自己相関は小さい: $\sum_j^\infty |\rho_j| < \infty$

そこで、サンプル系列 $\{Y_i\}(i = 1, ..., n)$ を m サンプル毎の $k(= n/m)$ 個の「バッチ」と呼ばれるグループに区切り、そのグループ毎のサンプル平均 (「バッチ平均」):

$$\overline{Y}_j(m) = \sum_{i=m(j-1)+1}^{mj} Y_i \ (j = 1, ..., k)$$

を考えると、「バッチサイズ」 m が十分大きければ上記の仮定からバッチ平均間の自己相関はゼロつまり独立と見なせ、同時に中心極限定理から (同一の) 正規分布に従うと見なせる。そこでこれらのバッチ平均を古典的な信頼区間の式に用いるものである。

7.3 逐次シミュレーションライブラリにおけるバッチ平均処理

逐次 DES ライブラリである SMPL[5] では、単一の出力変数に対するバッチ平均処理による精度判定機構が以下のように自動化されている。

すなわち、シミュレーションの起動前に、信頼区間の信頼係数と信頼区間幅への要求として推定した平均に対する割合 (相対半信頼区間幅)、及び初期の過渡状態として棄却するサンプルの数とバッチサイズを決定しておく。そして、実行時に精度判定用の手続きにサンプルが得られる毎にその値を渡すと、バッチ平均処理アルゴリズムにより所望の精度が得られたと判定した時点でそのことを示す戻り値を返すようになっている。

この様にサンプルの個数を固定せずに、相対半信頼区間幅への要求により実行を終了する用い方を「逐次法」と呼んでいる (並列に対する逐次との違いに注意)。ところが現在我々の知る PDES プラットフォームはあらかじめ実行終了時間なりイベントを決定しておくものである。そこでまずこの様に逐次実行で可能であった自動的な精度判定とそれによる実行の終了を並列環境でも可能にすることを目指す。

(なお、逐次法では得られた信頼区間が真の期待値を含む確率 (被覆確率) は要求する信頼係数よりも小さくなる。特に要求する相対半信頼間幅が大きい場合や信頼区間の計算を始めるまでに収集するバッチ数が小さい場合に顕著である。)

(1) サンプル取得

(2) 累積サンプル数が warmup 数以下ならば未収束と判定して終了

(3) サンプルの総和を更新

(4) サンプル数がバッチサイズの倍数でなければ終了

(5) バッチ平均を計算し、要求する被覆確率を満たす信頼区間が要求する信頼区間幅以内に収まっている場合は所望の精度が得られたと判定して終了

(6) (1) へ戻る

図 7.1: SMPL におけるバッチ平均処理アルゴリズム

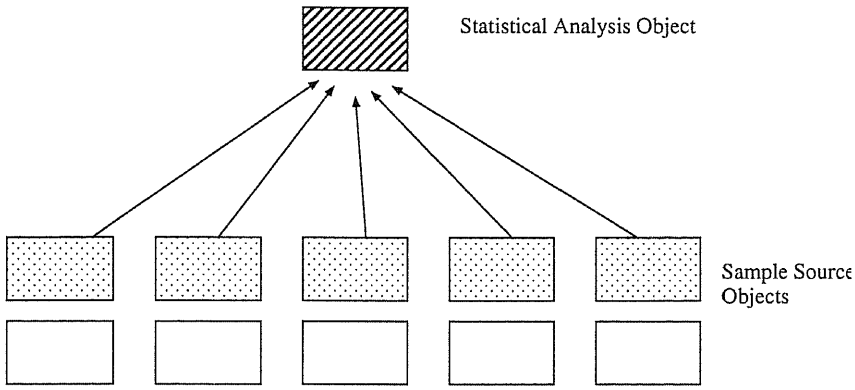

図 7.2: 素朴な統計処理

7.4 素朴な実装から並列環境への適応

7.4.1 素朴な実装

前節迄で概観したバッチ平均処理を、効率等を考慮せず最も素朴なアプローチで実現すると、図 7.1のアルゴリズムのような、サンプルを逐一受け取って精度判定を行うためのオブジェクトをひとつ用意し、サンプルを送出するオブジェクト側では、ひとつのサンプルが得られる度にその値を外部イベントとしてその統計処理オブジェクトへ送出することになろう (図 7.2)。プラットフォームの同期機構により自動的に系列の逐次化が行われるため、逐次実行の場合と同じ系列を得ることが出来る。統計収集オブジェクト側では図 7.1の処理をそのまま実装すればよい。

ところがここで、統計処理オブジェクトの数に対して、サンプルを出力するオブジェクト数が大きい場合は、トラフィックの集中から統計処理オブジェクトの外部イベント受信処理がボトルネックを形成しシステム全体のシミュレーション進捗に支障を来たすことは容易に想像が付く。つまり、統計処理オブジェクトへのイベントの到着間隔が受信処理速度を上回ってしまい、「輻輳」が起こるだろうと考えられる。(この間隔はモデルと特にその並列度により決まる。事象処理が高並列実行されていて出力変数の生成速度が速い場合はボトルネックを形成すると考えられる。)

例えば通信系のシミュレーションではセルやパケットの平均の遅延やスループットを求める必要があるが、その際サンプル源となるノードは複数ありかつその数の多いことがしばしばである。今後シ

ミュレーションの規模はますます増大してゆくと考えると (文献 [63] では、近い将来のネットワークシミュレーションは 1000 万ノード規模になると見込んでいる)、この傾向はますます顕著になるだろう。

7.4.2 素朴な実装での問題の検証

その様子を検証するために以下のような実験を行った。モデルとしては、大規模待ち行列ネットワークシミュレーションの一例としての Banyan Switch を、保守プロトコルによる自作のシミュレーションカーネル上に実装したものを用いた。Banyan Switch は高速広帯域ネットワークの交換機に用いられる多段構成のセルフルーチング交換網である。図 7.3に 3 段構成の Banyan Switch の構成を示す。

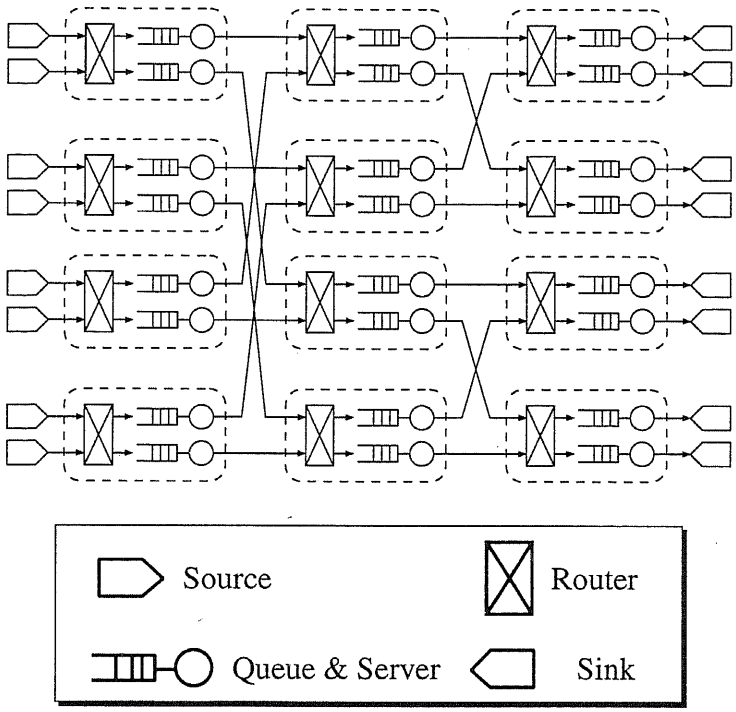

図 7.3: 3 段構成の Banyan Switch に対応する LP の構成

角の丸い四角形で囲まれた 2×2 の単位スイッチは到着した ATM セルのヘッダを読み取り、出力ポートを選択する。セルが既に使用中の出力ポートを指定している場合はバッファリングされる。この操作が全て並列に動作することにより、8 個の入力ポートから 8 個の出力ポートへの ATM セルのポイント・ツー・ポイントのルーチングを行う 8 × 8 スイッチが実現している。

このモデルについて、なんらの統計処理も行わない場合、統計処理オブジェクトを新たなプロセッサ上に用意し各出力ポートでのセル遅延のサンプルを逐一送信する上記「素朴な実装」の場合、更に後で述べるなんらかのデータの集合化が行えたと仮定して 100 個に一回送信を行う場合について、同じ仮想時間の実行に要する実時間を測定した。各出力スイッチを別々のプロセッサに含めるようにマッピングを行い、段数とそれに従ってプロセッサ数を増やすことで全体でのサンプル発生速度を変化させた。なお、実はここで統計処理プロセスはイベントを受信するだけで実際のデータ処理はさせていない。各場合について 15 試行行いその平均値と最初の 4 試行の値を表 7.1に示す。 まず、統計処理プロセスに送信されるデータイベント数が 11,300/ 秒までは、素朴な実装では 1 秒強だけ余計に

表 7.1: 統計処理オブジェクトの付加による実行速度の変化

cells/sec	統計処理プロセスなし					素朴な実装					100 個おきに送信				
6,000	6.5	6.9	7.0	6.4	6.8	7.5	7.7	8.9	8.2	8.3	6.9	6.9	7.0	6.8	6.9
8,000	9.4	10.1	9.9	9.7	10.1	10.7	10.9	11.7	12.2	11.6	10.1	9.6	9.8	10.1	9.9
11,300	15.2	14.4	14.7	13.7	14.3	15.0	14.7	17.1	17.3	15.5	13.9	14.0	13.7	16.6	14.6
14,000	19.5	25.3	22.0	23.3	23.0	*85.7	*238.9	*110.0	*∞	–	23.5	19.9	20.6	*125.0	22.7

* モデルと統計処理プロセスで進捗の同期の崩れたケース。

かかっていることが伺えるが、このモデルでは各出力スイッチでのシミュレーション時間当たりのセル到着数は段数に関わらず不変であるので、その回数のイベント送信に要した時間に相当すると思われる。

そして、統計処理プロセスに送信されるイベント数が 14,000/ 秒では実行に要する時間は急激に増え、しかも不安定になる。また、同時に、出力スイッチを受け持つプロセスと統計処理プロセスでのシミュレーション時間の進みを比較したところ、同期が崩れしばしば実行が滞ることが観察された。いかにも統計処理プロセスに届くイベントの速度がその処理速度を上回り輻輳が起こっている様子が伺える。

以上の結果から、サンプルを統計処理プロセスに逐一送信するという素朴な実装はやはり現実的でないことがわかる。

よって、可能ならばある程度サンプルを何らかの方法で集合化して送信することで統計オブジェクトの受信間隔を広げてやることが望ましい。もしくは統計処理オブジェクトを多段構成にすることも考えられる。100 個おきに送信した場合の結果を見ると、ほぼ期待通り統計処理オブジェクトを用意しない場合と同等の実行速度を示している。但し、11,000/ 秒では一度大きな値を示し素朴な実装での輻輳と同じ現象が見られた。これはモデル側からのイベントフローに波があり、その煽りを受けたかと考えられる。従って、データの集合化の単位はなるべく大きいほうがいいだろうとも考えられる。後の節ではそのための手法について検討する。

7.4.3 同期プロトコルとの兼ね合い

更に同期プロトコルとに関連して考えられる問題として、

保守プロトコル

統計データ用のチャネルのために統計データが発生しないときも大量にヌルメッセージが発行されてしまう。

楽観的プロトコル

内部イベントを発生しないため、それによる投機実行は起らないが、受け取る antimessage のためにロールバックが起る可能性がる。

統計処理オブジェクトでは、受け取った外部イベントが取り消される可能性があるため、中間結果がロールバック可能になるように状態ベクタとして統計データを保持しておく必要がある。

精度判定機構は所望の精度が得られた時点でシミュレーションを自動的に停止させなければならないが、楽観的プロトコルの下では、統計処理オブジェクトは個々のイベントを実行する時点ではそのイベントがキャンセルされるかどうか判断することが出来ないので、シミュレーションの停止処理といった不可逆な処理を行うことが出来ない。

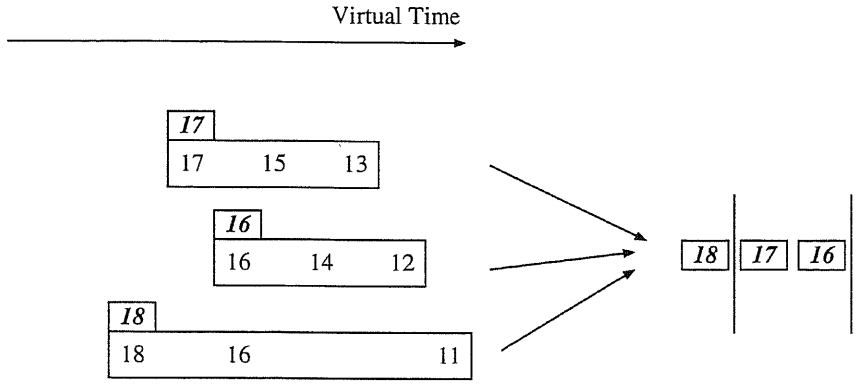

図 7.4: 統計データの集合化

しかし、精度の要求は高めに設定して、終了可能時刻を十分過ぎてから終了させるという方針を取ることは可能である。また、GVT さえ問い合わせることが出来るならば、シミュレーション停止という不可逆な処理を楽観的プロトコル上で発行することが可能になる。また、楽観的プロトコルカーネルには、実際 I/O 処理のような不可逆な処理に関しては実行をコミットされるまで遅延させておく機能を備えている場合がある [17]。

統計収集オブジェクトは、出力変数を受け取る間隔が十分広ければボトルネックを形成しないが、出力変数の集合化を行い、部分和等にまとめて送ると、バッチサイズが予め決められなくなり、バッチサイズの変動とバッチ中のサンプルの時間軸方向の入れ替わりを引きかえに、ボトルネックを緩和することになる。

図 7.4に、統計データの集合化による弊害の起るメカニズムを示す。統計データの集合化は、出力変数の変換と考えることが出来る。例えば「連続する k 個のサンプルの平均値を新しい出力変数とする」ということである。k サンプル間の同一性に関する情報が失われることになる。また、集合化の度合いにより、図のようにバッチに寄与するサンプルが逐次処理と異なる場合が生じる。

ここまでの議論で、いくつかの問題点が明らかになった。つまり、

1. サンプルひとつひとつを外部イベントとして送出すると受信側でボトルネックを生じる恐れがある。

   ボトルネックを解消することは、統計データを木構造ネットワークに沿って転送してゆくことによって可能である。しかし、前節のアプローチではオブジェクトの PP へのマッピングにより PP 間通信パターンが異なってしまうので、マッピング毎に木構造を再構成しなければならない。

2. 楽観的プロトコルを用いる場合は中間結果の状態としての管理や精度判定による終了処理を遅らせる必要がある。

3. ボトルネックを解消するために出力データをまとめて送信することにすると、バッチ処理の境界をまたいでサンプルの時間関係が入れ替わる恐れがある。

7.5 統計処理部の実装レベル

バッチ平均処理を PDES に組み込む場合、ユーザからハードウェアまでの階層構造におけるどの層で実現するかで複数のアプローチが考えられる。またプラットフォームの同期機構によっても実装

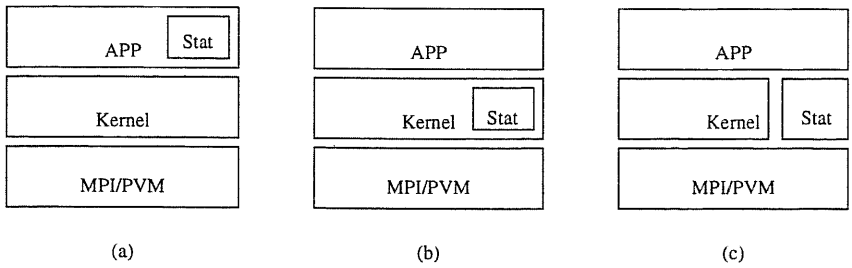

図 7.5: 統計処理モジュールの実装レベル;(a) イベントレベル (b) カーネルレベル (c) プロセスレベル

の事情は異なるであろう。本節では、分散メモリ型並列計算機における PDES へのバッチ平均処理の実装の各戦略について検討する。

7.5.1 イベントレベル実装

カーネルに手を加えず、統計収集のためのオブジェクト間の通信を外部イベントに委ねる方針である (図 7.5(a))。前節に述べた素朴な実装方針で採用した方法である。カーネルに手を加える必要がないため、カーネルの開発元とは独立して実装することが出来る利点がある。また、プロトコル同期機構により外部イベントは厳密に時間順に処理されるため、サンプル処理順の逆転の心配 (前節 3) がない。

7.5.2 カーネルレベル実装

カーネルに統計処理モジュールを組み込む方針である (図 7.5(b))。統計データのリダクション処理に関して、同一 PP にマップされているオブジェクト間に関しては通信なしに行うことが出来、性能上有利になる。また、楽観的プロトコルの場合は GVT の進捗状況を知ることが出来るため、コミットされたデータを区別することが容易である。

オブジェクトの集合化を利用した統計収集の様子を図 7.6に示す。カーネルプロセスは PP に対応しているため、木構造の構成は容易である。統計データはイベントレベルと同様 PP 内のいずれかのオブジェクトの送出する一般の外部イベントに付加 (piggy-back) しても良いし、カーネルメッセージによる通信でも可能である。後者の場合メッセージの同期を明示的に行う必要がある。異なる PP から到着する統計データに関して VT 軸上の時刻順到着は保証されないからである。

楽観的プロトコルに関しては、状態キューはカーネル中には存在しないため、統計データを保持しておくための状態キューを新たに設ける必要が生じる。PP 内のオブジェクト間で LTSF(Least Time Stamp First) スケジューリングが行われていると仮定出来るならば、PP 内でのサンプルの生成順序は時刻順に行われることが保証される。

7.5.3 プロセスレベル実装

図 7.5(c) のように、カーネルプロセスと同じ深さに位置し、カーネルとは独自にユーザプログラムとリンクされるモジュールとしての実現である。カーネルに手を加えなくて済むため、カーネルとは独立した実装が可能である。また、カーネルを介さずに処理が可能である分、処理のオーバーヘッドを抑えることが出来る。

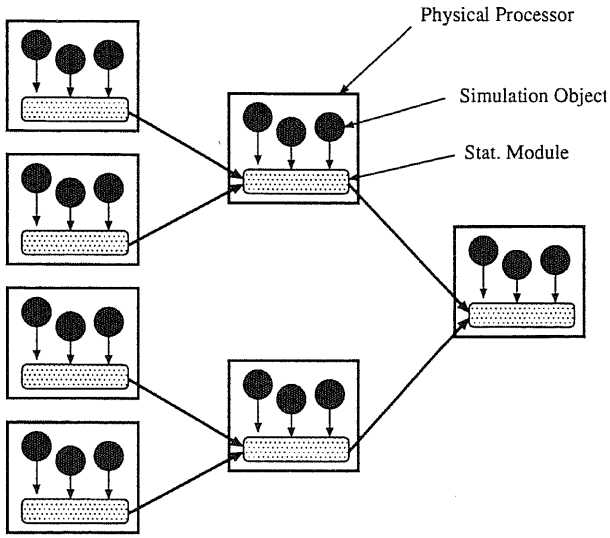

図 7.6: カーネルレベル実装における木構造通信トポロジの構成

統計情報を時々ユーザオブジェクトに問い合わせる手法 (PULL 型)、ユーザオブジェクトから統計処理プロセスに定期的に統計情報を送出する手法 (PUSH 型) が考えられる。いずれもプロトコル処理をバイパスすることにより自らタイムスタンプ順の整順を行う必要が生じる。

7.6 サンプル集合化の手法

本節では、バッチ平均法の並列環境への適用にあたって 7.4節で示唆したサンプルの集合化の手法について検討する。バッチ平均法で必要なのはバッチ毎のサンプル平均の系列である。したがって各オブジェクトは何らかの間隔でサンプルの和と数の情報を送信してやればよいだろう。単純に可能性を挙げれば、オブジェクト毎に個数を割り振りその合計がバッチサイズになるようにする方法と、バッチサイズのサンプルが集められると思われるシミュレーション時間間隔を指定してサンプルを収集する方法が考えられる。そこで以下の 2 節ではこれら 2 つのアプローチについての検討を行う。

なお、 DES での出力解析の手法の評価は、通常、推定された信頼区間の実際の被覆確率によって行う。また前述の様に被覆確率はサンプル系列の自己相関に影響を受ける。従って以下でも基本的に自己相関及び被覆確率の観点から評価する。また、 warm-up 期間やバッチサイズの動的な決定はそれ自体簡単でない問題であるが、ここでは与えられているとする。

なお、 DES で平均を求めようとする量には、

discrete-time mean: 時間軸上で離散的に発生し、サンプル数で平均。 ex. 遅延

$$\widehat{\mu} = \frac{1}{n}\sum_{i}^{n} D_i \quad (D_i \in \mathbf{R})$$

continuous-time mean: 時間軸上で連続的に、時間で平均するもの ex. 平均キュー長)

$$\widehat{\mu} = \frac{1}{T}\int_0^t Q(t)dt$$

の 2 種類がある。これまでは descrete-time mean について議論して来たが、時間で平均する continuous-time mean については、時間間隔でバッチサイズを決めその間の時間積分をバッチ平均法等に用いることができ、逐次実行と同じことが可能である。従って、以下では引続き descrete-time mean について議論する。

7.6.1 オブジェクト毎に個数を割り当てる手法

まずサンプル源となるオブジェクトが k 個あるとし、それらから得られるサンプル系列からサイズ m のバッチを構成するとする。ここでは、i 番目のオブジェクトで $l_i(i = 1, ..., k)$ 個ずつ集めて統計処理プロセスに (和と個数を) 送信し、統計処理プロセスそれらをタイムスタンプ順に並べて m 得られる毎にバッチ平均処理を行うものとする。例えば、逐次実行でトータルの時系列からバッチを構成する場合は $l_i = 1$ である。素朴には $l_i = m$、つまり各バッチをまるまる各サンプル源で構成する場合や、その自然な中間値として均等配分 $l_i = m/k$ がまず思い付く。$l_i = 1$ 以外では、時間軸上で一繋がりのサンプルからバッチを構成することができなくなり、少なくとも逐次実行とは違った系列を処理することになる。(図 7.4)

また、各オブジェクトから得られるサンプル系列に関しては、どのオブジェクトから得られる系列もその分布と速度が等しい場合と、分布や (しばしば同時に) 速度が多少異なる場合が考えられる。通信システムで平均の通信遅延を対称とする場合の例を挙げると、前者の例としては交換機で各出力ポートの人気が等しい場合、後者の例としてはイーサネット等バス型トポロジのもの (ぶら下がる点が端に行くほど程不利になり平均遅延は大きくなる) などが考えられる。

以下ではまず $l_i = m$ 及び $l_i = m/k$ の場合について検討する。前述のように信頼区間の計算の際にはまずは自己相関の影響が大きいため、ここでは、その妥当性をバッチ平均間の自己相関の観点から考察する。

なお、以下の模式図ではサンプル源となるオブジェクト数を 2、バッチサイズを 4 としている。

また、データの例としては、イーサネットのモデル [5] [1]での通信遅延を用いた。このモデルでは各ステーションはキューを持たずに一つの送信要求が完了した後に次の送信要求をスケジュールする作りになっており、遅延は主に複数ステーション間の送信要求の衝突による。従ってサンプル系列の相関については、各ステーション毎の系列では見られず (図 7.7)、ステーション間の同時期のサンプル間に見られる (従ってトータルの系列には相関が現れる。図 7.8) という特徴を持つ。また、ステーションの位置を明確に考慮すればその平均に若干の差を持たせることが出来る。以下の議論では、オブジェクト間の同時期サンプルの相関が問題になるためこのモデルのデータを例にあげることにした。

[$l_i = m$ の場合]

– 各オブジェクトでのサンプル系列の期待値と発生速度が等しい場合

各サンプル源でのサンプル時系列の自己相関が強い場合には有効。けれども、一般に各オブジェクトから同時期に得られるサンプル間にも相関があり、それをそのままバッチ間に持ち込んでしまうため、その相関が強い場合には適当ではないと考えられる。(図 7.11上)

– 各オブジェクトでのサンプル系列の期待値に差がある場合

また、各オブジェクトでのサンプル系列の期待値の間に少しでも差があると、図 7.11下のように得られるバッチ平均の系列にオブジェクト数毎の周期性が現れてしまうため適当でないと考えられる。

[1]文献でのコードはステーションを区別しない作りになっていたため区別するよう変更したものを用いている

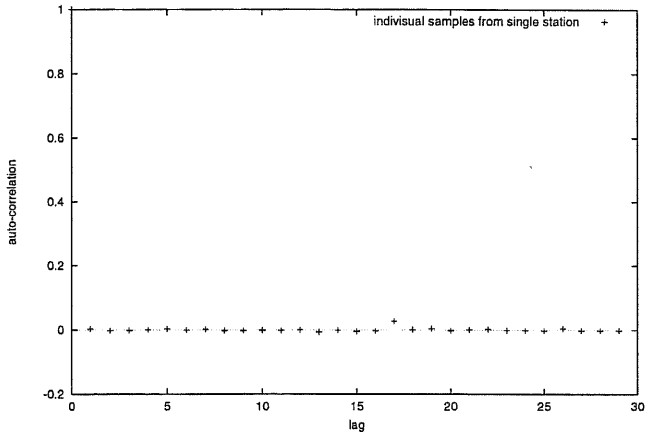

図 7.7: 1 ステーションでの系列の自己相関

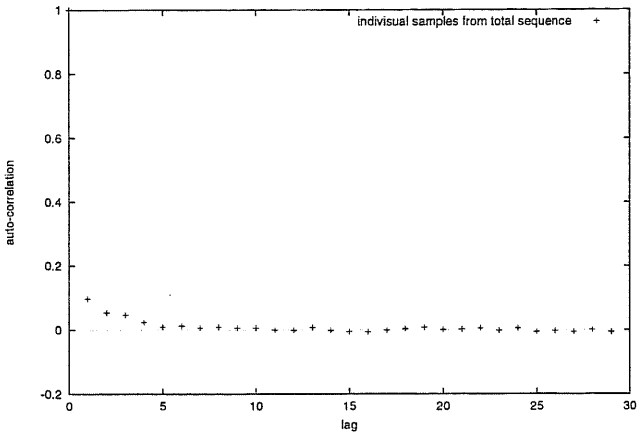

図 7.8: トータルの系列の自己相関

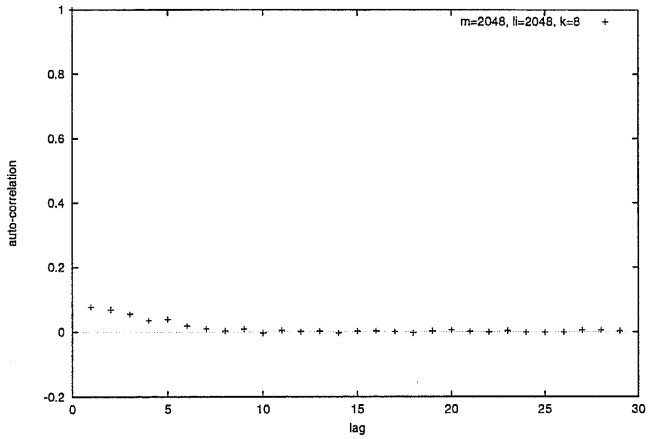

図 7.9: $l_i = m, (k = 8)$ 速度分布均一

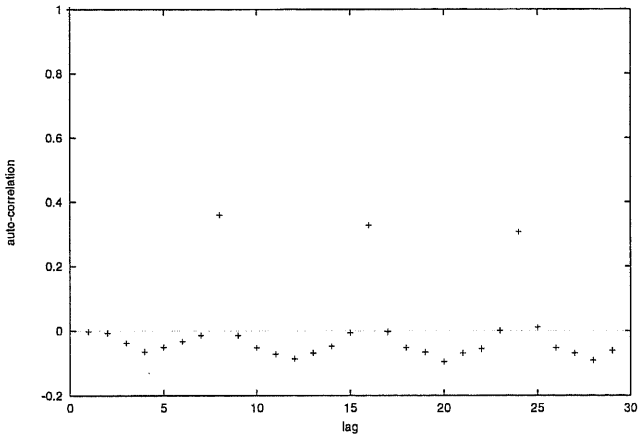

図 7.10: $l_i = m, (k = 8)$ 速度分布不均一

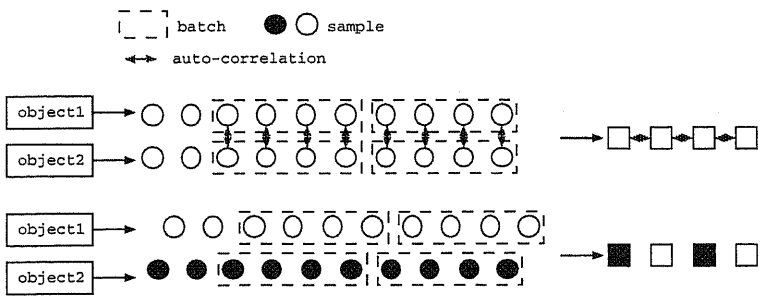

図 7.11: $l_i = m$ での問題の模式図

p

[$l_i = m/k$ の場合]

– 各オブジェクトでのサンプル系列の期待値と発生速度が等しい場合

サンプル発生間隔が厳密に等ければ、トータルでのサンプル系列から得られるバッチの系列と同じ

ものを処理することになる (図 7.13上)。ところが実際はばらつくため、一つのバッチに同じオブジェクトからのブロックが 2 つ入ることがしばしばある。従って、図 7.13に示すように、やはり各オブジェクト間での同時期サンプル間の相関を持ち込んでしまうことが分かった。またこの現象はバッチサイズを大きくしても変わらない。

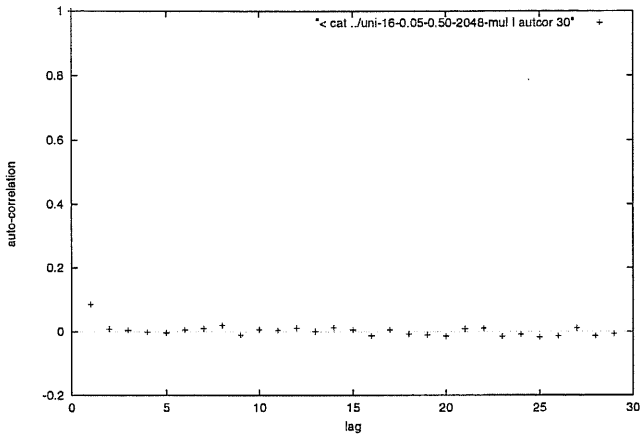

図 7.12: lag 1 に相関が見られる

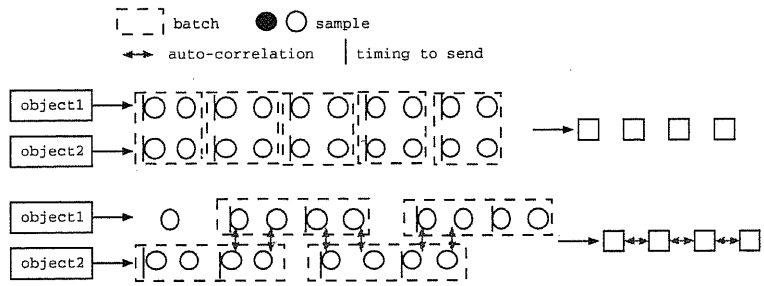

図 7.13: 各オブジェクトにバッチを均等に割り当てた場合の問題その 1

– 各オブジェクトでのサンプル系列の期待値と発生速度に差がある場合

オブジェクト毎にサンプル発生間隔の揺らぎが小さくバッチサイズが大きいと、各オブジェクトで $l_i = m/k$ サンプル集まるペースが安定してしまい、図 7.16上に示すように、やはり周期性が生まれる可能性があると考えられる。さらにその期待値の差が大きくサンプル系列の分散が小さいときにその傾向は強くなる。但し実際には期待値が大きく異なる場合には全体としての平均をとる意味はないだろうから、稀なケースではあるだろう。

この周期性を回避するためには、均等に割り当てるのではなく速度に比例して割り当て、オブジェクト毎に速度の等しい場合に持ち込めば、図 7.16下のようにこの問題を回避できると考えられる。実際イーサネットのシミュレーションで入力負荷および端末間伝搬遅延を操作して無理矢理前述のような条件を生み出したところ、バッチ平均の系列の自己相関に周期性が現れた。(図 7.14) そこで各ステーションでのサンプル発生速度を測定してその比で割り当てたところ、周期的な自己相関は見られなくなった。(図 7.15。但しラグ 1 に残る負の相関はパラメータを操作した事により生まれたもの。)

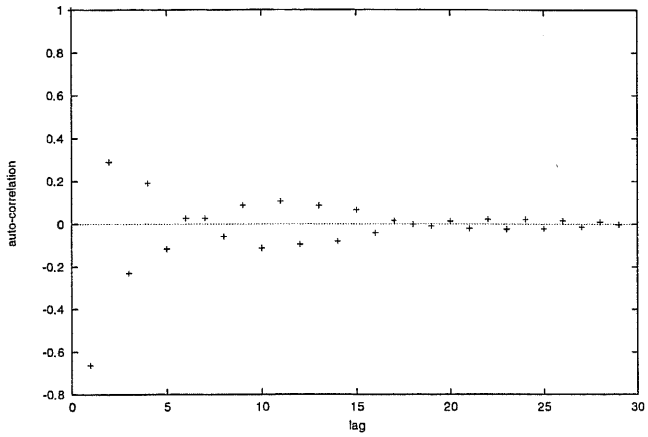

図 7.14: $l_i = m/k$ 速度分布不均一

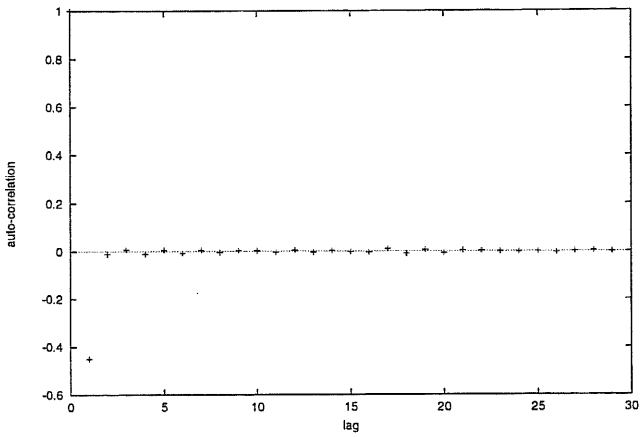

図 7.15: $l_i =$ 速度比で配分 速度分布不均一

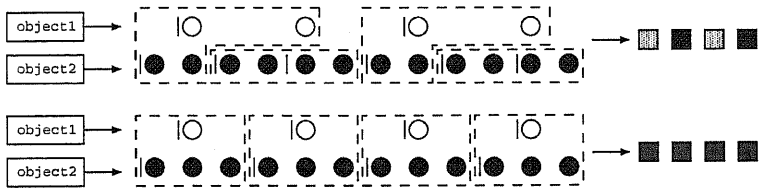

図 7.16: 各オブジェクトにバッチを均等に割り当てた場合の問題その 2

けれども上記の図 7.13下の問題は残る。更に割り振るサイズがオブジェクト毎に違うため、到着順序の変動によって求めるバッチサイズが得られなくなる。

但し、速度比例の個数配分は整数比でしか実現出来ず、更に実際の系列は時刻とともに変動する。従って、統計処理モジュール側で、サンプル源を区別せず、バッチサイズが得られた時点でバッチ平均を計算すると、サンプル間の順序の逆転が避けられず、それによりバッチ平均間でサンプル源間の相関を拾ってしまい、$lag = 1$ の相関が生じてしまう。

また、統計処理モジュール側で、全サンプル源からサンプルが得られた時点でバッチ平均を計算すると、サンプル源間でバッチ平均に寄与する時刻範囲が変動してしまう。

よって、オブジェクト間での相関が強いと予想される場合には個数区切りは注意を要する。また、更には $\sum_i^k l_i > m$ もしくは $\sum_i^k l_i < m$ も考えられるが、$\sum_i^k l_i > m$ では、一つのバッチに寄与するオブジェクトは必ず k より小さくなり、$l_i = m$ と同様の問題を招くと考えられる。従って結局、$\sum_i^k l_i < m$ — 安全のためには、統計処理プロセスの負担にならない程度に出来るだけ細かく分けることが望ましいと言える。けれども、すべての場合によく働くような簡単なポリシーの確立は難しいと思われる。

まとめ

オブジェクト間での相関が強いと予想される場合には個数区切りは若干信頼性が落ちる。そうでない場合、各オブジェクトの条件が同等で各々から得られる系列の分布が等しい場合には、均等配分を用いることができ、条件が同等でなく分布やペースが大きく異なると予想される場合にはペースで配分することが考えられる。すべての場合によく働くような簡単なポリシーの確立は難しいと思われる。

7.6.2 時間毎に収集する手法

今度は、期待するバッチサイズ m が十分大きければ m 個のサンプルを得るために必要なシミュレーション時間間隔は安定するであろうとの期待から、統計処理プロセスがその間隔を実行時に見積もり、各サンプル源にシミュレーション時間間隔を指定してその間のサンプル和を送信してもらってバッチ平均法に応用できないかという発想もあり得よう。そこで以下では、シミュレーション時間間隔 Δt_i(以下「バッチサイズ」に対し「バッチインターバル」と呼ぶ) 中に得られたサンプルから i 番目のバッチ (サイズを m_i とおく) を構成してバッチ平均法に用いることを考察する。

Δt_i の決定に当たっては、単純には実行初期に適当なインターバルを見積もり後は固定 ($\Delta t_i = const.$) してしまうか、過去 j バッチ分のインターバルの平均を用いてなるべく期待する m に近付ける ($\Delta t_i = m_{exp} \sum_{j=1}^{l} \Delta t_{i-j} / \sum_{j=1}^{l} m_i$) ことが考えられる。

但し、後者の場合にはサンプル源となるオブジェクトと統計処理オブジェクト間の通信が終始必要となってしまうことに注意する。保守的プロトコルの場合、インターバル指定のイベントの届くのが遅れるとそれを待つことになる。楽観的プロトコルの場合、データを送信してしまってからインターバル指定の要求が届くことが考えられ、わざわざロールバックを起こす恐れがある。

M/M/1 Queue を用いた実験

DES の出力解析でしばしば引き合いに出される M/M/1 queue ($\rho = 0.8$) を対象として、

- $m_i = m$ (P)
- $\Delta t_i = const.$ (C)
- $\Delta t_i = m\Delta t_{i-1}/m_i$ (A1)
- $\Delta t_i = m \sum_{j=1}^{4} \Delta t_{i-j} / \sum_{j=1}^{4} m_i$ (A4)

として得られたバッチ平均の系列をまず単純にバッチ平均法に用いてみた。得られたバッチサイズ $\{m_i\}$ の変動を表 7.2に、バッチ平均の系列のラグ 1 の自己相関 (A4 のためにラグ 4 も) を表 7.3に、バッチ数固定 (5,30) で 90% の信頼区間の信頼区間を計算した時の点推定と被覆確率を表 7.4に示す。

まず表 7.2より、いずれの場合もバッチサイズ $\{m_i\}$ の変動が正規分布し、その分散が m に反比例して減ることがわかる。また、結局 Δt_i 固定の C の場合がもっとも分散が小さい。表 7.3より、時間間隔によるバッチの指定はバッチ間になんら新たな相関を生み出さない。表 7.4より、表 7.2で見た $\{m_i\}$ の変動の大きさに関わらず、全体的に被覆確率は P の場合に近い値を示しているといえる。

但し、この実験では期待値の点推定としてバッチ平均の平均を用いておりトータルのサンプル系列の平均にはなっていない。図 7.17に得られたバッチサイズとそのときのバッチ平均の平均及び標準偏差の関係を示すが、この関係のために推定した点推定の平均は概して $m_i = m$ の場合より若干小さくなっている (表 7.4)。

そこで次には、ad hoc ではあるが、点推定としてはトータルのサンプル平均を用い、信頼区間幅だけサイズの変動するバッチから見積ることを考え、同じ実検を行った。(それぞれ CN, A1N, A4N とする。) その結果を表 7.5に示す。

また、相対半信頼区間幅が (relative precision) が求める小ささになったら終了する逐次法の場合の 90% 信頼区間の被覆確率を表 7.6に示す。(ここでは relative precision=0.1 とし、信頼区間の計算を始めるバッチ数は、差があればそれを強調だろうと考え最低の 2 とした。)

表 7.2: $\{m_i\}$(バッチサイズ) の分布

	C				A1				A4			
m	16	256	1024	4028	16	256	1024	4028	16	256	1024	4028
mean	16.0	256.0	1024	4028	17.2	257.0	1025	4029	16.2	256.2	1024	4028
std. dev.	3.99	16.0	32.0	63.7	6.67	22.8	45.4	89.8	4.51	17.9	35.8	71.1
skewness	0.25	0.06	0.03	0.01	1.67	0.20	0.10	0.05	0.36	0.09	0.04	0.03
kurtosis	3.08	3.01	3.00	3.00	14.2	3.08	3.02	3.02	3.21	3.01	3.01	3.00

表 7.3: 得られたバッチ平均の系列の自己相関

	lag 1				lag 4	
	P	C	A1	A4	P	A4
16	.74	.72	.72	.72	.24	.22
256	.11	.10	.10	.10	.00	.00
1024	.02	.02	.02	.02	.00	.00
4028	.01	.01	.01	.01	.00	.00

結果は CN,A1N,A4N いずれも P の場合とほぼ等しく、バッチサイズ (インターバル) を大きくするに従い P に近付く。とくに P の場合に十分な被覆確率を示すバッチサイズ (2048 と 4028) ではいずれも P とほとんど同じ値を示す。この結果からインターバルによるバッチの構成は有力視してよかろう。更に、一度決めた後は通信の必要がなく実装も容易な $\Delta t_i = const.$ を用いることが得策と考える。

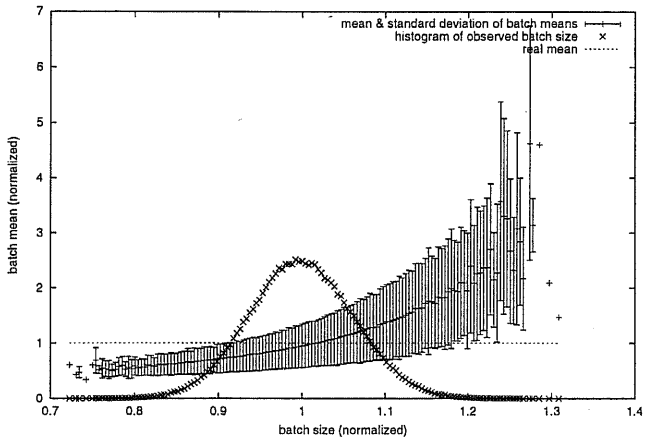

図 7.17: ケース C での m_i のヒストグラム及び得られた m_i 毎のバッチ平均の平均と標準偏差

表 7.4: 単純な場合: 点推定の平均値と、バッチ数 $n = 5, 30$(固定) から計算した 90% 区間の被覆確率 (± その 90% 信頼区間)

	P		C		A1		A4	
	n		n		n		n	
m	5	30	5	30	5	30	5	30
16	1.001 53.5(±.3)	1.001 49.9(±.6)	.949 52.1(±.3)	.949 47.8(±.6)	.950 52.3(±.3)	.950 47.8(±.6)	.949 51.9(±.3)	.949 47.2(±.6)
256	.999 80.7(±.2)	.999 83.5(±.5)	.986 79.8(±.2)	.986 81.4(±.5)	.986 79.9(±.2)	.986 82.1(±.5)	.986 79.9(±.2)	.986 81.6(±.5)
1024	1.000 86.1(±.2)	1.000 87.7(±.5)	.996 85.8(±.2)	.996 87.7(±.4)	.996 85.8(±.2)	.996 87.3(±.5)	.996 85.7(±.2)	.996 87.3(±.5)
4028	1.000 88.8(±.4)	1.000 89.3(±.8)	.999 88.7(±.3)	.999 88.9(±.7)	.999 88.6(±.2)	.999 89.0(±.4)	.999 88.7(±.2)	.999 89.0(±.4)

表 7.5: 点推定を補正: 点推定の平均値と、バッチ数 $n = 5, 30$(固定) から計算した 90% 区間の被覆確率 (± その 90% 信頼区間)

	P		CN		A1N		A4N	
	n		n		n		n	
m	5	30	5	30	5	30	5	30
16	1.001 53.5(±.3)	1.001 49.9(±.6)	.972 53.0(±.3)	.993 49.3(±.6)	.991 53.5(±.3)	1.000 49.9(±.6)	.982 53.0(±.3)	1.000 48.6(±.6)
256	.999 80.7(±.2)	.999 83.5(±.5)	.996 80.1(±.2)	.998 82.2(±.5)	.998 80.4(±.2)	.999 83.1(±.5)	.997 80.4(±.2)	.999 82.7(±.5)
1024	1.000 86.1(±.2)	1.000 87.7(±.5)	.999 86.0(±.2)	1.000 88.0(±.4)	1.000 86.0(±.2)	1.000 87.6(±.5)	1.000 85.9(±.2)	1.000 87.6(±.5)
4028	1.000 88.8(±.4)	1.000 89.3(±.8)	1.000 88.7(±.3)	1.000 89.0(±.6)	1.000 88.6(±.2)	1.000 89.1(±.4)	1.000 88.8(±.2)	1.000 89.3(±.4)

その他のモデルでの実験

実際的な場面で有効かを検証するため、以下のモデルでも実験を行った。但し真の期待値は解析的に得られないため、一度長いシミュレーションを行った結果から得られた値を用いている。

– Banyan-Switch モデル

前述の Banyan-Switch のモデルで同様の実験を行った。ここでも通常のバッチ数固定による手法と時間間隔指定による方法から全く同様の結果が得られた。(表 7.7,7.8)。(なお、通常このシミュレーションはバッチサイズ 10000 で行われる。)

– Ethernet モデル

前述のイーサネットモデルで同様の実験を行った。ここでも同様の結果が得られた。(なお、通常このシミュレーションはバッチサイズ 2000 で行われる。)

以上の結果から、少なくとも待ち行列型のモデルにおいてインターバルでバッチを指定することは、サイズでの指定と同等に扱ってよいだろうと予想する。よって、本研究では並列実行でのバッチ平均法としてはインターバル指定を採択する。

更に、なんらかのバッチサイズに近付けることは本質的でなくまたそのサイズがわかっていること

表 7.6: 逐次法への応用: 90% 半信頼区間幅の平均値、収束までに要したバッチ数。被覆確率 (但し relative precision=0.1, 信頼区間の計算を開始するまでに集めたバッチ数は 2)

m	P	CN	A1N	A4N
16	.097	.096	.097	.096
	186	174	172	174
	46.6(±1.6)	46.6(±1.5)	47.4(±1.5)	45.4(±.2)
256	.093	.093	.093	.093
	52.9	50.1	50.0	50.2
	72.3(±.8)	71.8(±.7)	72.3(±.7)	71.8(±.7)
1024	.090	.090	.090	.090
	15.9	15.7	15.7	15.8
	74.9(±.5)	74.7(±.4)	74.7(±.5)	74.9(±.5)
4028	.082	.083	.083	.082
	6.0	6.0	6.0	6.0
	78.7(±.5)	79.0(±.4)	79.2(±.2)	79.0(±.2)

表 7.7: Banyan-Switch (バッチ数固定): 90% 信頼区間の被覆確率

	P		CN	
	n		n	
m	5	30	5	30
1000	86.6(±.1)	85.9(±.4)	86.6(±.1)	86.1(±.4)
5000	89.3(±.3)	88.9(±.7)	89.2(±.3)	89.9(±.7)
10000	89.5(±.4)	90.0(±1.0)	89.1(±.4)	89.5(±1.0)

表 7.8: Banyan-Switch (逐次法): 相対半信頼区間幅の平均、要したバッチ数、90% 信頼区間の被覆確率 (relative precision=0.1、初期バッチ数 =2)

m	P	CN
1000	.082	0.082
	5.6	5.6
	76.6(±.2)	76.7(±.2)
5000	.068	.068
	3.2	3.2
	85.4(±.3)	85.3(±.3)
10000	.061	.061
	2.8	2.8
	87.2(±.3)	87.7(±.3)

は仮定できない。実行時のバッチサイズの決定については、通常のバッチサイズによるバッチ構成の場合にはアルゴリズムがいくつか提案されている。これは、小さいバッチサイズから始めて多数集めて自己相関を見積り、有意であれば連結してサイズを増やすというものである。そこで、インターバルによる場合にもこれらの手法の応用が必要であろう。

7.7 実装例

現在、以上で述べたアプローチを、 WARPED 上のオブジェクトとして、つまり「イベントレベル」でのライブラリを構築中である。

表 7.9: Ethernet (バッチ数固定): 90% 信頼区間の被覆確率

	P		CN	
	n		n	
m	5	30	5	30
512	86.5(±.2)	88.8(±.4)	86.5(±.2)	88.2(±.5)
1024	88.2(±.3)	89.6(±.6)	88.2(±.3)	89.3(±.6)
2048	89.2(±.4)	90.0(±.9)	89.2(±.4)	89.4(±.9)

表 7.10: Ethernet (逐次法): 半信頼区間幅の平均、要したバッチ数、90% 信頼区間の被覆確率 (relative precision=0.1、初期バッチ数 =2)

m	P	CN
512	.084	.084
	7.5	7.4
	75.2(±.3)	74.8(±.3)
1024	.080	.080
	5.1	5.1
	79.0(±.3)	79.3(±.3)
2048	.074	.074
	3.9	3.8
	82.8(±.3)	83.1(±.4)

データを受け取り処理する統計処理オブジェクト (以下 aStatistitianObj) は一つのクラスとしてパックするとして、サンプル源となるオブジェクトと統計処理オブジェクトの間に入って、個々のサンプルを受け取り自動的に適宜まとめて送信する部分 (aStatLocal) を実装するに当たっては、一つのオブジェクトとして実装して同じプロセッサに載せるかサンプル源オブジェクトに埋め込むクラスとして実装するかの 2 つが考えられよう。

前者の場合はサンプル毎に内部イベントをスケジュールするオーバーヘッドがある。

後者の場合、埋め込まれるクラスは、

- 自動送信のために、コンテナオブジェクトのイベント送信関数にアクセス出来なくてはならない。
- 統計処理オブジェクトからのイベントを他のイベントから分離するため一度それを見せてもらって送り元 LP を確認する必要がある。
- 更に楽観プロトコルの場合は、
  - ロールバックに備えて刻々更新される状態変数のクラスと、変化しないパラメータ及びメソッドを保持する本体のクラスを分離する必要がある。したがってユーザはオブジェクトの状態クラスの StatLocal の状態クラスを含め、かつオブジェクトの状態クラスの初期化メソッド中で StatLocal の状態クラスの初期化関数も呼ぶよう記述しなければならない。更に StatLocal のサンプル収集関数にサンプルを渡す前に毎回状態変数を aStatLocal にセットしてやらなければならない。

統計処理用イベントには

- データメッセージ (StatLocal → StatistitianObj)

- バッチサイズ指定メッセージ (StatistitianObj → StatLocal)
- バッチインターバル指定メッセージ (StatistitianObj → StatLocal)
- 実行終了メッセージ (StatistitianObj → StatLocal)(未実装)

を用意する。

aStatistitianObj は、パラメータとして切捨てるべき warm-up の長さ l と期待するバッチサイズ m と精度、及び個数区切りか時間区切りかを指定する。

7.7.1 多段構成による通信負荷分散

非同期な統計量のリダクションを効率的に行うために前節迄で述べた多段構成を取ることが出来る。上流の複数のサンプル源もしくは中継オブジェクトからのデータと自分のデータを合わせて下流のひとつのオブジェクトへ転送するアプローチである。

- aStatLocal の多段構成
  PP 数を別途必要としないで、PP 数が変っても PP あたり一定の通信量を維持することが出来る。
- aStatistitianObj の多段構成
  統計処理用に別の PP を割り当てて、その PP 間の多段構成を組むものである。ボトルネックを解消するにはサンプル源に比例した数の aStatistitionObj が必要である。他のプロセッサを使うのでモデル側へのフローバランスを崩さない利点がある。

7.8 まとめ

本章では、PDES の枠組みの中で定常状態シミュレーションを行う場合に効率よく出力解析処理を行う枠組みを提案した。

LP から分散して出力される統計量の収集においては、ある程度集合化しない限り PP 間通信のボトルネックの形成が避けられないが、バッチ平均法で精度判定を行う場合、集合化によりバッチ平均間の相関を導入してしまうという問題点を指摘し、出力統計量の分布 / 速度の違いによる集合化の手法のガイドラインを示した。更に、集合化だけでは通信のボトルネックを本質的に除くことが出来ないことが分かり、バッチ処理のための PP 間通信を tree 構造にすることによるボトルネックの解消の方法についての議論を行った。

統計処理を行うためのユーザインタフェースはなるべく逐次に近い形で実現することが望ましいが、プロトタイプの設計により、そのような実現が可能であることを示すことが出来た。

本研究により、汎用 PDES プラットフォームが統計処理のために提供すべき機能が明らかになった。すなわち、ユーザレベルでの実装は効率上無理があり、GVT の問い合わせやシミュレーションの正常終了等の LP の基本的な手続きと同レベルで統計処理インタフェースが提供されることが望ましい。

更に、統計処理データのフローがボトルネックを形成しないような tree の構成を支援する枠組みが必要であることも明らかになった。

第 8 章

結論

本論文では、実世界アプリケーションとしての待ち行列シミュレーションを直接手動で並列化した経験に端を発し、現在の PDES に不足している様々な機能についての提案を行った。

待ち行列シミュレーションでは待ち行列データ構造をモデルの内部状態として保持しておく必要があるが、投機的実行のためにロールバックを実現しなければならない楽観的プロトコルの下では、待ち行列データ構造のような動的なデータ構造を容易に状態として実現することが出来なかった。第 3章では最もプログラミングインタフェースの単純なコピーに基づく状態保存を支援するプラットフォーム上で待ち行列データ構造を状態として透過的に効率良く管理する枠組みを提案し、更に第 4章で一般の動的なデータ構造の管理のための一連のツール群を提案した。

待ち行列システムは交換機内の接点やバッファ等、ネットワークの階層に応じて並列動作する様々な要素が存在し、また様々な規模のネットワークをシステムの要素数より少ない物理プロセッサでシミュレートする必要があった。第 5章では、ヒープを用いて単一物理プロセッサ上で複数のサブモデルを効率よくスケジューリングする手法を提案した。一般的なスケジューリングパターンではあまり大きな性能改善は見られなかったが、楽観的プロトコルにおいて保守プロトコルで与えられていたようなサブモデル間のトポロジに関する情報を生かして最適化を図る可能性を示した。

サブモデル間のトポロジには、スケーラブルな待ち行列システムでは特に規則的な結合関係が存在するが、そのトポロジをうまく物理プロセッサ間結合網のトポロジに整合させるように割り付けプログラムを書くことは困難であった。サブモデル間や、しばしば物理プロセッサ間の結合トポロジは複雑で直感的に得られる印象とは性質を異にするからである。第 6章では、サブモデル間の通信の物理プロセッサ間網に対する負荷をモデル化し、それを最小化するようにサブモデルを実行時に移送しながら漸近的に最適な割り当てに近づけて行く枠組みを提案した。移送に必要なサブモデルのデータ構造の直列化等、移送に必要な具体的な機構について言及した。

待ち行列シミュレーションはシステムが定常状態にあるときの統計的なデータを必要とすることが多く、逐次プラットフォームで提供されている精度判定機構が並列環境で提供されていないことが分かり、第 7章では統計解析処理を性能を損なわずに実現する手法を提案した。並列動作するサブモデル毎に分散して出力されるデータをある程度集合化して流量を削減すると、出力系列間の余分な相関を拾ってしまう問題が明らかになり、それを避けるような集合化の方法を提案し、プロトタイプを実装して実行可能性を示した。

あらゆるシステムの大規模化に伴い同期的、集中制御的な動作が最早不可能に近くなってきた中で、並列離散事象シミュレーションによるモデリングは以前に比べて現在のシステムの姿に適合しやすくなってきているとも言える。こうした大規模システムのシミュレーションの需要はこれからも増大していくと予測され、上記のような支援系の整備を積み重ねて行くことにより、並列離散事象シミュレーションがシミュレーションの高速実行のパラダイムとしてより広く貢献出来ると確信する。

謝辞

本論文をまとめるに当たって、指導教官である相田仁助教授には、テーマの選択から論文の構成法に至るまで細部にわたり懇切丁寧にご指導頂きました。いつも快く相談に乗って下さり、いつでも私の意思を最大限に尊重したアドバイスで導いて下さいました。齊藤忠夫教授には、通信システムの実情を交えながら温かい助言を頂くと共にいつも本研究に感心を寄せて頂き、限りなく元気づけられました。卒論、修論、博論の6年間の齊藤・相田研での研究生活で学ばせて頂くことが出来たことは数知れません。心より感謝致します。

共同研究者である島村 英氏、蔡 晟蔚氏には、それぞれ統計出力解析と論理プロセス移送による通信の最適化についての成果を共有させて頂きました。両氏とのパートナーシップにより、より広範な視点での議論が可能になりました。

研究室配属から実に6年間もの苦楽を共にした同期の森野博章氏には、研究内容はもとより、研究そのものの在り方など、多岐にわたって議論の相手をして頂き、様々なヒントを得ました。同氏から提供されたリング型LANシミュレーションプログラムは筆者の事象駆動の概念への良きイントロダクションとなりました。

青木 輝勝氏には、日頃から研究の相談に乗って頂くと同時に、たゆまぬ激励の言葉を頂きました。本論文で繰り返し登場し、本研究の出発点でもあるBanyan Switch網のモデルは、同氏から提供頂いた逐次版のシミュレーションプログラムに基づいています。

最後に、研究のみならず日頃の大学院での生活においても大変お世話になりました以下の方々に深く感謝致します。

富山 忠広 助手	千葉 新吾 技術官	篠崎 恭子 女史	大江 寿美子 女史
芦川 みさき 女史	竹内 純子 女史	橋本 美保 女史	中井 由美子 女史
オヌル アルティンタシュ氏	大野 隆一氏	小口 正人 氏	川合 史朗氏
玉木 剛 氏	藤田 博文 氏	柳 寅太 氏	金 炳錫 氏
トラナウィカライ トリーデージ 氏		ブンワォラセト ウドムキャット氏	
権 容徳 氏	高橋 由泰 氏		
寺西 祐人 氏	小島 章裕 氏	高橋 淳一 氏	箕浦 真 氏
林 博之氏	大西 広泰 氏	黒沢 崇宏 氏	小早川 知昭 氏
安田 忠彰 氏	山口 健一 氏	神戸 龍信 氏	奈川 英樹 氏
ピサイ セーッタウォン 氏	八木原 英之 氏	吉田 順一 氏	若狭 健 氏
大西 隆之 氏	中井 弘亮 氏	浜村 倫行 氏	村中 孝行 氏
山口 実靖 氏	林 輝煌 氏		
小泉 博一 氏	香山 美奈子 女史	タイ タッチ バオ 氏	タン ティッケ 氏
野上 耕介 氏	フロマン マキシム 氏	星谷 直哉 氏	横山 明久 氏
研究生の皆様	卒論生の皆様	研究員の皆様	私の家族

電気・電子情報・電子工学科教官の皆様、 電気・電子情報・電子工学科事務室の皆様
電気・電子情報・電子工学科秘書室の皆様、電気・電子情報・電子工学科図書室の皆様

1998 年 12 月 18 日

日高 宗一郎

参考文献

[1] 石川 裕. コモディティハードウェアを用いた並列処理技術. 情報処理, Vol. 39, No. 8, pp. 784–791, 8 1998.

[2] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 1995. Ver 1.1 draft.

[3] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[4] J. Keith Townsend, Zsolt Haraszti, James A. Freebersyser, and Michael Devetsikiotis. Simulation of Rare Events in Communications Networks. *IEEE Communications Magazine*, Vol. 36, No. 8, pp. 36–41, August 1998.

[5] Myron H. MacDougall. *Simulating Computer Systems: Techniques and Tools*. MIT Press series in computer systems. MIT Press, Cambridge, Mass, 1987.

[6] Marc Badel, Philippe, Thierry de Pretto, and Guenther Siegel. Stat-Tool: An Extensible and Distributed Object Oriented Statistical Tool for Discrete Event Simulation. In *Proceedings of the 1996 Object Oriented Simulation Conference*, 1996.

[7] MIL 3, Inc. *OPNET Tutorial Manual*, October 1994. Release 2.5.A.

[8] Alois Fercha. Parallel and Distributed Simulation of Discrete Event Systems. In Albert Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*, chapter 35, pp. 1003–1041. McGraw-Hill, 1996.

[9] K. M. Chandy and R. Sherman. Space-Time and Simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pp. 53–57, 1989.

[10] Albert G. Greenberg, Boris D. Lubachevsky, and Isi Mitrani. Superfast Parallel Discrete Event Simulations. *ACM Trans. Model. Comput. Simul*, Vol. 6, No. 2, pp. 107–136, April 1996.

[11] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, Vol. 18, No. 1, pp. 39–65, 1986.

[12] D. A. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, pp. 404–425, 1985.

[13] D. M. Nicol. Parallel Discrete-event Simulation of FCFS Stochastic Queueing Networks. *ACM SIGPLAN Not.*, Vol. 23, No. 9, pp. 124–137, September 1990.

[14] V. Jha and R. Bagrodia. Simultaneous Events and Lookahead in Simulation Protocols. Technical Report CSD 960043, Computer Science Dept, UCLA, Los Angeles, CA, December 1996.

[15] Rajive L. Bagrodia and Wen-Toh Liao. Maisie: A Language for the Design of Efficient Discrete-Event Simulations. *IEEE Transactions on Software Engineering*, Vol. 20, No. 4, pp. 225–238, April 1994.

[16] Rajive Bagrodia, Richard Meyer, Mineo Takai, Yu an Chen, Xiang Zeng, Jay Martin, and Ha Yoon Song. Parsec: A Parallel Simulation Environment for Complex Systems. *IEEE COMPUTER*, Vol. 31, No. 10, pp. 77–85, Oct 1998.

[17] D. E. Martin, T. J. McBrayer, and P. A. Wilsey. WARPED: A Time Warp Simulation Kernel for Analysis and Application Development. In H. El-Rewini and B. D. Shriver, editors, *29th Hawaii International Conference on System Sciences (HICSS-29)*, Vol. 1, pp. 383–386, January 1996.

[18] David M. Nicol and P. Hidelberger. On Extending Parallelism to Serial Simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pp. 60–67, 1995.

[19] Boyan Groselj. CPSim: A Tool for Creating Scalable Discrete-Event Simulations. In *Proceedings of the 1995 Winter Simulation Conference*, pp. 579–583, Arlington, VA, December 1995. BoyanTech, Inc.

[20] Samir Das, Richard Fujimoto, Kiran Panesar, Don Allison, and Maria Hybinette. GTW: A Time Warp System for Shared Memory Multiprocessors. In *Proceedings of the 1994 Winter Simulation Conference*, pp. 1332–1339, 1994.

[21] Kalyan S. Perumalla and Richard M. Fujimoto. GTW++ - An Object-oriented Interface in C++ to the Georgia Tech Time Warp System. Technical Report GIT-CC-96-09, Georgia Institute of Technology, Atlanta, GA, March 1996.

[22] F. Gomes, S. Franks, B. Unger, Z. Xiao, J. Cleary, and A. Covington. SimKit: A High Performance Logical Process Simulation Class Library in C++. In *Proceedings of the 1995 Winter Simulation Conference*, pp. 706–713, December 1995.

[23] *WarpKit Programmers Manual - Draft 1.9.1*, May 1995. http://bungee.cpsc.ucalgary.ca/SIEMENS/WarpKit/WarpKit1.1.tar.Z.

[24] BoyanTech, Inc., McLean, Virginia. *CPSim 1.0s user's guide and reference manual*, 1995. http://www.wdn.com/bti-sim/cpsman.zip.

[25] D. R. Jefferson, et al. The Time Warp Operating System. In *11th Symposium on Operating Systems Principles*, pp. 77–83, November 1987.

[26] Yoshiko Yasuda, et al. Architecture and Performance of the Hitachi SR2201 Massively Parallel Processor System. In *Proceedings of the 11th International Parallel Processing Symposium*, pp. 233–241, April 1997.

[27] Radharamanan Radhakrishnan, Dale E. Martin, Malolan Chetlur, Dhananjai Madhava Rao, and Philip A. Wilsey. An Object-Oriented Time Warp Simulation Kernel. In *Proceedings of the Second International Symposium on Computing in Object-oriented Parallel Environments*, Vol. 1505, pp. 13–23. Springer-Verlag(Lecture Notes in Computer Science), December 1998.

[28] Jeff S. Steinman. SPEEDES: A Multiple Synchronization Environment for Parallel Discrete Event Simulation. *International Journal in Computer Simulation*, Vol. 2, pp. 251–286, 1992.

[29] Jeff S. Steinman. Discrete-Event Simulation and the Event Horizon. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pp. 39–49, 1994.

[30] David M. Nicol, Chris C. Micheal, and Patrick Inouye. Efficient Aggregation of Multiple LP's in Distributed Memory Parallel Simulations. In *Proceedings of the 1989 Winter Simulation Conference*, pp. 680–685, Washington, D.C., December 1989.

[31] David M. Nicol, Ann S. Yoshimura, and Michael M. Johnson. The IDES Framework: A Case Study in Development of a Parallel Discrete Event Simulation System. In *Proceedings of the 1997 Winter Simulation Conference*, pp. 93–99, December 1997.

[32] Alois Ferscha and Michael Richer. Java Based Conservative Distributed Simulation. In *Proceedings of the 1997 Winter Simulation Conference*, pp. 381–388, December 1997.

[33] Paul A. Fishwick. Web-Based Simulation. In *Proceedings of the 1997 Winter Simulation Conference*, pp. 100–102, December 1997.

[34] Robert M. Cubert and Paul A. Fishwick. A Framework for Distributed Object-Oriented Multimodeling and Simulation. In *Proceedings of the 1997 Winter Simulation Conference*, pp. 1315–1322, December 1997.

[35] David M. Nicol, Michael M. Johnson, Ann S. Yoshimura, and Michael E. Goldsby. Performance modeling of the IDES framework. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pp. 38–45, June 1997.

[36] David Nicol. Parallel Discrete Event Simulation: So Who Cares? In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, Lockenhaus, AUSTRIA, June 1997.

[37] Richard M. Fujimoto. Parallel Discrete event simulation: Will the field survive? *ORSA Journal on Computing*, Vol. 5, No. 3, pp. 213–230, 1993.

[38] Ernest H. Page and Richard E. Nance. Parallel Discrete Event Simulation: A Modeling Methodological Perspective. In *Proceedings of the ACM/IEEE/SCS 8th Workshop on Parallel and Distributed Simulation*, pp. 88–93, 1994.

[39] Richard M. Fujimoto. Future Directions in Parallel Simulation Research. *ORSA Journal on Computing*, Vol. 5, No. 3, pp. 245–248, 1993.

[40] Rajive L. Bagrodia. Perils and Pitfalls of Parallel Discrete-Event Simulation. In *Proceedings of the 1996 Winter Simulation Conference*, pp. 136–143, December 1996.

[41] David M. Nicol and Xiaowen Liu. The Dark Side of Risk (What your mother never told you about Time Warp). In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pp. 188–195, June 1997.

[42] University of Cincinnati. *The WARPED Time Warp Simulation Kernel*, July 1996. Ver 0.7.

[43] Jeff S. Steinman. Incremental State Saving in SPEEDES Using C++. In *Proceedings of the 1993 Winter Simulation Conference*, pp. 687–696, 1993.

[44] Josef Fleischmann and Philip A. Wilsey. Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pp. 50–58, 1995.

[45] Avinash C. Palaniswamy and Philip A. Wilsey. An Analytical Comparison of Periodic Checkpointing and Incremental State Saving. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pp. 127–134, 1993.

[46] Edward Mascarenhas, Felipe Knop, Reuben Pasquini, and Vernon Rego. Checkpoint and Recovery Methods in the ParaSol Simulation System. In *Proceedings of the 1997 Winter Simulation Conference*, pp. 452–459, December 1997.

[47] Brian W. Unger, John G. Cleary, Alan Covington, and Darrin West. External State Management System for Optimistic Parallel Simulation. In *Proceedings of the 1993 Winter Simulation Conference*, pp. 750–755, 1993.

[48] Douglas W. Jones. An Empirical Comparison of Priority-Queue and Event-Set Implementations. *Communications of the ACM*, Vol. 29, No. 4, pp. 300–311, 1986.

[49] Randy Brown. Calendar Queues: A Fast $O(1)$ Priority Queue Implementation for the Simulation Event Set Problem. *Communications of the ACM*, Vol. 31, No. 10, pp. 1220–1227, 1988.

[50] Robert Rönngren, Jens Riboe, and Rassul Ayani. Lazy Queue: A new approach to implementation of the pending-event set. *International Journal in Computer Simulation*, Vol. 3, pp. 303–332, 1993.

[51] Yukinori Yamamoto and Kazuo Taki. Parallel Logic Simulator based on Time Warp. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp. 1198–1206. ICOT, ICOT, April 1992.

[52] Yoshiko Yasuda, et al. Deadlock-free Fault-tolerant Routing in the Multi-dimensional Crossbar Network and its Implementation for the Hitachi SR2201. In *Proceedings of the 11th International Parallel Processing Symposium*, pp. 346–352, April 1997.

[53] 保田 淑子, 藤井 啓明, 田中 輝雄, 稲上 泰弘. ハイパクロスバネットワークの通信性能評価. 信学技報 CPSY93-25, August 1993.

[54] 蔡 晟蔚. 並列離散事象シミュレーションにおける論理プロセスの移送による通信の最適化. 東京大学工学部電子工学科 卒業論文 提出予定, February 1999.

[55] Azzedine Boukerche and Sajal K. Das. Dynamic Load Balancing Strategies for Conservative Parallel Simulations. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pp. 20–28, June 1997.

[56] David W. Glazer and Carl Tropper. On Process Migration and Load Balancing in Time Warp. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 3, pp. 318–327, March 1993.

[57] Hervé Avril and Carl Tropper. The Dynamic Load Balancing of Clustered Time Warp for Logic Simulation. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pp. 20–27, May 1996.

[58] Jeff Johnson. Object Serialization, Persistence, and Distribution. http://www.cs.byuh.edu/~johnsonj/research/serialization/papers/report-final.html, 10 1996.

[59] Sun Microsystems,Inc. Object Serialization Stream Protocol. http://java.sun.com/products/jdk/rmi/doc/serial-spec/protocol.doc.html, 1996.

[60] Linda F. Wilson and David M. Nicol. Experiments in Automated Load Balancing. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pp. 4–11, May 1996.

[61] A. M. Law. Statistical Analysis of Simulation Output Data. *Operations Research*, Vol. 31, pp. 983–1029, 1983.

[62] L. W. Schruben. Detecting Initialization Bias in Simulation Output. *Operations Research*, Vol. 30, pp. 569–590, 1982.

[63] Sandeep Bhatt, Richard Fujimoto, Andy Ogielski, and Kalyan Perumalla. Parallel Simulation Techniques for Large-Scale Networks. *IEEE Communications Magazine*, Vol. 36, No. 8, pp. 42–47, August 1998.

[64] 島村 英. 並列離散事象シミュレーションにおける統計解析処理. Master's thesis, 東京大学大学院工学系研究科 情報工学専攻 提出予定, 1999.

発表文献

論文誌

[1] 日高 宗一郎, 青木 輝勝, 相田 仁, 齊藤 忠夫. タイムワープ方式向け FIFO キュークラスライブラリの実装と評価. 電子情報通信学会論文誌, Vol. J81-D-I, . 投稿中.

[2] 日高 宗一郎, 相田 仁, 齊藤 忠夫. コピーに基づく状態管理を行うタイムワープ方式プラットフォームに於ける漸進的状態保存の支援の枠組み. 電子情報通信学会論文誌, Vol. J81-D-I, . 投稿準備中.

国際会議

[3] Soichiro Hidaka, Terumasa Aoki, Hitoshi Aida, and Tadao Saito. A FIFO Queue Class Library as a State Variable of Time Warp Logical Processes. In *Proceedings of the Second International Symposium on Computing in Object-oriented Parallel Environments*, Vol. 1505, pp. 151–158. Springer-Verlag(Lecture Notes in Computer Science), December 1998.

研究会

[4] 日高 宗一郎, 相田 仁, 齊藤 忠夫. 並列離散事象シミュレーションにおけるヒープデータ構造を用いた論理プロセスのスケジューリング. 電子情報通信学会技術研究報告, コンピュータシステム, 第 98 巻, pp. 81–87, August 1998. SWoPP 長岡 '98 CPSY 60 ～ 73.

全国大会

[5] 日高 宗一郎, 相田 仁, 齊藤 忠夫. 待ち行列システムの並列イベント駆動シミュレーションの高速化に関する一考察. 電子情報通信学会総合大会 講演論文集 情報・システム, pp. D–3–7, 東海大学, March 1998. 電子情報通信学会.

[6] 島村 英, 日高 宗一郎, 相田 仁, 齊藤 忠夫. 離散事象シミュレーションの並列分散処理における出力統計処理についての一考察. 第 57 回 全国大会 講演論文集 (第一分冊), p. 21, 名古屋大学工学部, October 1998. 情報処理学会.

[7] 島村 英, 日高 宗一郎, 相田 仁, 齊藤 忠夫. 並列離散事象シミュレーションにおける出力統計処理. 第 58 回 全国大会 講演論文集. 情報処理学会, March 1999.

その他

[8] 日高 宗一郎, 後藤田 洋伸, 橋爪 宏達, 小野 欽司. PDF 電子カタログを用いた回路設計支援システムの研究. 総合大会講演論文集 基礎・境界, p. 109. 電子情報通信学会, March 1997.

[9] トラナウィカライ トリーデージ, 日高 宗一郎, 橋本 昭則, 青木 輝勝, 相田 仁, 齊藤 忠夫. A Study of Ring Reconfiguration in Wireless Token Passing LAN. 総合大会 一般講演, pp. B-5-303. 電子情報通信学会, March 1998.

[10] 齊藤 忠夫, 相田 仁, 青木 輝勝, 日高 宗一郎, トラナウィカライ トリーデージ, 橋本 昭則. 無線LANにおけるシャドウイングを考慮した分散型アクセス制御方式. 電子情報通信学会論文誌, Vol. B-I, . 投稿中.

[11] 齊藤 忠夫, 相田 仁, 青木 輝勝, 日高 宗一郎, ブンウォラセト・ウドムキャット. TST 接続型可変速度 TDM 交換方式. 特許出願中, 1998.

[12] 齊藤 忠夫, 相田 仁, 青木 輝勝, 日高 宗一郎, ブンウォラセト・ウドムキャット. 可変速度トラヒックに適した TDM スイッチアーキテクチャ. 電子情報通信学会論文誌, Vol. B-I, . 投稿準備中.

[13] 首藤 敏, 金 炳錫, 日高 宗一郎, 相田 仁, 齊藤 忠夫. 無線トークンパッシング lan の提案. 総合大会 講演論文集, pp. -, 慶應大学, March 1999. 電子情報通信学会.

[14] 相田 仁, 川合 史朗, 日高 宗一郎. AP-1000 上での NCX コンパイラの実現. 文部省重点領域研究「超並列原理に基づく情報処理基本体系」第 4 回シンポジウム予稿集, pp. 2-112—2-126, March 1994.

[15] 日高 宗一郎, 川合 史朗, 相田 仁, 齊藤 忠夫. 超並列 C 言語 NCX の MIMD 計算機用コンパイラにおける仮想プロセッサ割り当ての手法. 第 49 回 (平成 6 年後期) 全国大会講演論文集 (5), ソフトフェア, 第 49 巻, pp. 75-76, 北海道大学, September 1994. 情報処理学会. 並列化コンパイラセッション 一般講演 4T11.

[16] 川合 史朗, 日高 宗一郎, 相田 仁, 齊藤 忠夫. 超並列 C 言語 NCX の MIMD 計算機用コンパイラの構成. 第 49 回 (平成 6 年後期) 全国大会講演論文集 (5), ソフトフェア, 第 49 巻, pp. 73-74, 北海道大学, September 1994. 情報処理学会. 並列化コンパイラセッション 一般講演 4T10.

[17] 相田 仁, 川合 史朗, 日高 宗一郎, 林 博之. AP-1000 上での NCX コンパイラの実現. 文部省重点領域研究「超並列原理に基づく情報処理基本体系」第 5 回シンポジウム予稿集, pp. 2-2—2-6, October 1994.

[18] 相田 仁, 川合 史朗, 日高 宗一郎, 林 博之. AP-1000 上での NCX コンパイラの実現. 文部省重点領域研究「超並列原理に基づく情報処理基本体系」第 6 回シンポジウム予稿集, pp. 2-2—2-9, March 1995.

[19] 日高 宗一郎, 川合 史朗, 相田 仁, 齊藤 忠夫. 超並列 C 言語 NCX における式の遠隔評価. 第 51 回 (平成 7 年後期) 全国大会講演論文集 (6), ソフトフェア, 並列処理, システム, 第 51 巻, pp. 97-98, 富山大学, September 1995. 情報処理学会. 並列処理: 並列言語処理系と支援系 一般講演 2P-6.

[20] 相田 仁, 日高 宗一郎. AP-1000 上での NCX コンパイラの実現. 文部省重点領域研究「超並列原理に基づく情報処理基本体系」 最終報告書, March 1996.

[21] 日高 宗一郎, 相田 仁, 齊藤 忠夫. 超並列 C 言語 NCX における遠隔評価による通信の最適化. 1996 年 電子情報通信学会 情報・システムソサイエティ大会講演論文集, p. 60, 金沢大学, September 1996. 電子情報通信学会.