博 士 論 文

疎結合並列システムにおける
並列論理プログラム変換による耐故障化に関する研究

指導教官　横田 治夫 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

杉野 栄二

1997 年 1 月 16 日

要 旨

近年、商用の超並列計算機が利用可能になりつつあるが、超並列計算機は高性能を求めて開発されてきたため、これまで信頼性についての研究はあまり行われて来なかった．超並列計算機をターゲットとする並列プログラムは、高性能を求めているため本質的に計算量が巨大であり、長時間かかり得るものである．にもかかわらず、これら並列プログラムは、たとえ要素プロセッサ一つが故障しても実行に失敗する環境にある．

本研究は、一般の商用超並列計算機を想定した耐故障化をめざすものである．耐故障化のための負担をプログラマにかけないために、プログラムの自動変換によって耐故障並列ソフトウェア（FTPS）を生成することを目的とする．得られる FTPS は、プライマリサイト・アプローチに基づいた構成で動作する．各サイトは任意個のプロセッサで構成することができ、サイト内での並列動作を可能にする．耐故障実行させたいユーザプロセスは、複製されてプライマリサイト (PS) とバックアップサイト (BS) で並列動作させられる．PS で動作するユーザプログラムは、非決定的動作をする際にのみ BS に実行ログを送る．BS で動作する複製プログラムは、実行ログをもとに PS と同じ動作をトレースすることで、同じ状態を保ち続ける．外界への出力コミットのために、非決定的実行と出力動作では緩い同期を行う．この方式を形式化したモデルにより、その正当性を示す．なお変換の容易さから、本研究では並列論理型言語で記述される並列プログラムを対象とした．

実装するシステムの基本性能から、本研究で提案する手法で耐故障実行させた場合の実行オーバーヘッドが予測できる． 実行オーバーヘッドは、プログラム中の非決定性とシステムのコミュニケーション性能に依存していることが計算によって示され、ベンチマークプログラムによる実験によって確認される．もしコミュニケーション性能が低いシステムであっても、並列性の高いユーザプログラムであるならば、並列実行させることで耐故障実行によるオーバーヘッドが軽減されることを計算によって示す．

また MTTF は、分割したサイト数の log オーダーで増大することが示される．例えば、MTTF が 1 万時間の単一プロセッサ千台から構成される並列システムは、MTTF が 10 時間となり、これを越えるような仕事の成功は期待できない．この並列システムを非再構成の 1-Resilient システムに組み変えた場合、元の並列システムに比べて性能は 1/2 に落ちることから、同じ仕事に対して 20 時間以上かかり得るが、MTTF は 30 時間になるため十分に成功する可能性がある．このように本システムは、制限される性能の割合よりも高い割合の MTTF の改善を実現し、システム有効度を高めることができる．

目 次

図 目 次

表 目 次

第 1 章

序論

近年、商用の超並列計算機が利用可能になりつつあるが、超並列計算機は高性能を求めて開発されてきたため、これまで信頼性についての研究はあまり行われて来なかった．一方、高信頼性を求めた耐故障計算機も商品化されているが、専用で高価な二重化ハードウェアと専用 OS を備えたものであり、安価で高性能を求めた超並列計算機とは別の発展をしている．並列プログラムが長時間の連続運用を必要とする巨大なものになるにつれて、超並列計算機に対する信頼性の要求も高まりつつある．超並列計算機の耐故障化は、その要素プロセッサの冗長性を利用することで実現するのが安価である． 適用性の高さも考慮するならば、ソフトウェアで行う方法が最適である．ソフトウェアによる耐故障化が実現された後、その機能を効率化する手段として安価な装置を開発することも期待できる．しかし耐故障化を行うことによってプログラムの冗長動作を増やすことから、本来のプログラムの性能を低下させる可能性が大きく、また個々のプログラムをそれぞれ耐故障化する際のプログラマへの負担の大きさが憂慮される．

本研究は、一般の商用超並列計算機を想定した耐故障化の実現を目指すものである．より多くの超並列計算機、並列ソフトウェアに対して適用できることを考え、ソフトウェアによって実現する．またプログラマの負担を軽減させるため、自動変換によって耐故障化されたプログラムを生成する．

耐故障プログラムはプログラム自身の冗長性を増すことから、無故障時の実行効率を低下させる． よって耐故障化が実用的であることを示す上で、この冗長性を解析し実行オーバーヘッドを見積もることが重要になる．さらに実験によって、この見積り値が適当であることも示すことも重要である．

本研究では、故障モデルとして要素プロセッサの停止故障を仮定して並列論理プログラムによる故障回避を行う．ソフトウェアによるハードウェア故障回避を実現するため、扱う故障は単純なプロセッサ停止故障である． ネットワーク故障については扱わないが、プロセッサ停止故障と同等に見える一部の故障はカバーされる．

並列論理型言語は自動変換が容易であり、並列言語として十分な記述力と機能を備えているが、これまで耐故障化に関する研究は行われていない．並列論理プログラムはプロセスモデルとして解釈されることから、本研究の成果はプロセス指向のプログラミングにおいても適用できると思われる．

本研究で提案する手法は非修理系の冗長系を構成し、与えられたシステムを複数サイトに分割する．システムの MTTF は、このサイト数に関する log オーダーで増大することが本文で示される．例えば、MTTF が 1 万時間の単一プロセッサ千台から構成される並列システムを考えると、この並列システムは MTTF が 10 時間となる．このことは、この並列システムを最も有効に使えば、単一プロセッサが 1 万時間に成し遂げる仕事量を 10 時間で得られることを示す．しかし、この並列システムで 10 時間を越えるような仕事の成功は期待できない．実際には並列実行させるためのオーバーヘッドが含まれることや、並列性を十分に引き出すことが難しいことから、処理容量は単一プロセッサのそれよりも減少すると見るべきであろう． よって 1 プロセッサでは 1 万時間のプロセッサ・パワーを必要とするような並列プログラムを故障に会わずに実行することは期待できない．この並列システムを非再構成の 1-Resilient システムに組み変えた場合を考えてみよう． プロセッサ全体を 2 つのサイトグループに分割するために、元の並列システムに比べて性能は 1/2 に落ちることから、上と同じ仕事に対して 20 時間以上かかり得る．ところが、MTTF は 30 時間になるため、十分に成功するだろうことが分かる．このように、本システムは制限された性能の割合よりも高い割合の MTTF を得ることができ、長時間連続運転したいプログラムにとって有効である．

本論文は、以下のように構成される．第 2章で、耐故障ソフトウェアの一般的アプローチと、その実現上の問題点について述べる． これらアプローチを拡張して並列ソフトウェアを耐故障化する概念について第 3章で述べる．第 4章では、本研究で扱う故障モデルと耐故障化の前提について述べ、第 3章の概念を元にして形式化を行い、耐故障実行が保証されることを述べる．第 5章では、実現される耐故障ソフトウェアの信頼性と有効性について論じる．第 6章では、本研究で用いる並列論理型言語 KL1 について概説し、並列論理プ

ログラムがプロセスモデルとして解釈されることを述べる．並列論理プログラムは、その非決定的な動作故に動作の再現が困難である．本章では、この非決定的動作を再現するデバッギング手法についても述べる．このデバッギング手法は、第3章 で示したモデルとも関連性が見られる．第7章では、本研究で提案する耐故障ソフトウェアの構成について述べる．第8章では、アプリケーションプログラムを耐故障ソフトウェアに変換する方法について述べる．第9章では、耐故障化によって予想されるオーバーヘッドについて解析し、見積りを行う．第10章では、実際にプログラムを人工的なベンチマークプログラムと応用プログラムを用意し、これらをnCUBE2上で動作させる．これら実験結果をもとに実行オーバヘッドについて検証する．第11章で今後の課題についてまとめる．

第 2 章

耐故障ソフトウェアの一般的アプローチ

ソフトウェア・フォールトトレランスがソフトウェアの故障すなわちバグをマスクする技術であるのに対して、フォールトトレランス・ソフトウェア（耐故障ソフトウェア）はハードウェアや OS のような計算プラットフォームの故障に耐えるソフトウェアである．従って、ここで扱う故障は、ソフトウェアから検出・回避できるものに限られる．TMR(Triple Moduler Redundancy) システムのように多数決によってビザンチン故障を回避するアプローチもあるが、ここでは特にプロセッサの停止故障に限る．

耐故障ソフトウェアを構成するには、一般に次の３ステップを要する [1]．

1. 耐故障ソフトウェアとして標準的なソフトウェアパラダイムに基づいてアプリケーションプログラムを構成する．

2. 選択したパラダイムを実現する基礎となる機能を抽出する．

3. 各機能を実現するメカニズムをインプリメントする．

本研究では、アプリケーションプログラムを耐故障パラダイムに沿った形で自動生成することが主な目的であるため、各パラダイムの基礎機能の実現については詳説しない．基礎機能については、OS 等のレベルでインプリメントされるものとする．

2.1 耐故障ソフトウェアのパラダイム

耐故障ソフトウェアのパラダイムとして、以下が一般的である．

2.1.1 Replicated state machine （RSM）アプローチ

state machine(SM) は状態変数を持ち、コマンドを受けとることによってそれを更新し、あるいは出力を発する．コマンドの実行は決定的であり、また他のコマンドについてアトミックである．すなわち同時に複数のコマンドを実行することはない．状態変数の変化の列と出力は、入力列によって完全に決定される．

SM は複製されて、分散システムの別マシン上で動作させる．それぞれの複製には同じ入力コマンド列を与え、それぞれ実行させる．

2.1.2 Primary site （PS）アプローチ

RSM アプローチを発展させたものである．RSM アプローチとの相違は、複製された SM のうち一つ（primary site （PS））がコマンドを実行し、他の複製（backup site （BS））は故障に備えて待機する点である．PS は複製 SM の状態を示すチェックポイントを安定記憶に格納しながら動作する．PS の故障時には BS の一つが安定記憶のチェックポイントから状態を回復し、新しい PS として実行を継続する．あるいは、チェックポイントは PS の動作に並行して動的に BS に転送される場合もある．

2.1.3 Object/Action モデル

アプリケーションプログラムは、オブジェクトの集合として構成される．オブジェクトは状態を保持し、状態は安定記憶に置かれる．アクションは、状態を更新しようとするオブジェクトから発せられ、アトミックである． すなわち、状態更新の途中の状態は他のアクションからは見えない．このアトミック・アクションは、データベースの世界ではトランザクションと呼ばれる．

2.2 パラダイムの実現上の問題点

Object/Action モデルは、データベースで用いられるパラダイムであるが、状態が頻繁に更新されるような並列プログラミングには適さず、RSM アプローチと PS アプローチの方が、より一般的なプログラムに適用可能性がある．ここでは、RSM アプローチと PS アプローチについて、実現上の問題点、特に並列プログラムに適用する際の問題点について

考察する．文献 [1] では、PS アプローチ実現上の基礎機能として、メンバシップ とマルチキャストが仮定されているが、これら機能については提供されるものとする．

2.2.1 非決定性

RSM アプローチでは、SM は決定的であることが仮定されている．すなわち、同時に複数コマンドを実行しないことと、入力コマンド列は決められた順序で順に到着することが仮定されている．これは、単一の逐次プログラムに対して、単一の入力ストリームでコマンドが与えられることに相当する．入力ストリームが複数ある場合や、並行・並列プロセスから構成される場合については考慮されていない．

PS アプローチはチェックポインティングをもとにして状態を回復させるため、一貫性のとれたチェックポイントがとれさえすれば非決定的な PS でも実現可能である．

2.2.2 チェックポインティング

PS アプローチでは、故障に備えてチェックポイントを安定記憶に格納しながら実行を続けねばならない．すなわちアプリケーションプログラムは、チェックポイントから状態が回復でき、途中から実行が継続できるように書かれている必要がある．何をチェックポイントとし、チェックポイントからどうやって回復させるかは、プログラマに委ねられる．チェックポインティングライブラリをユーザに提供するアプローチもある [3] が、複雑なプログラム、特に並行・並列プログラミングでは、チェックポインティングの設計は困難でありプログラマへの大きな負担である．また並列動作するプロセス間のコネクションも回復する必要がある．このとき並列プロセスが動的に生成消滅するならば、動的に変化するコネクションを考慮しなければならない．

またマルチコンピュータでのチェックポインティングは、メッセージパッシングを考慮に入れてプロセッサ間でチェックポイントの一貫性を保証することが必要である． さらに超並列マルチコンピュータでのチェックポインティングでは、実現上の問題点として次があげられている．

- 一般に個々のノードが安定記憶へ直接アクセスできない．
- 超並列計算機は時間空間的に高性能を求められるため、使用メモリ量、実行オーバヘッドの低減が強く望まれる．

これに対して、プロセス間のメッセージカウントを導入することで、一般の超並列計算機や分散システムで実現できる低オーバーヘッドな協調チェックポインティングスキームが提案されている [4]．協調チェックポインティングは、すべてのプロセスで協調してチェックポインティングを行うため、故障からのリカバリはそれぞれ独立して行える．これに対して独立チェックポインティングでは、プロセスが個々にチェックポインティングできるかわりに、リカバリ時に協調して一貫性のとれたチェックポイントを選ぶため複雑なリカバリアルゴリズムになる．独立チェックポインティングは、メモリ要求量が大きく、安定記憶へのI/O が頻繁であるため超並列計算機では不利だとされている．

2.2.3 外界とのコミュニケーション

RSM アプローチ、PS アプローチは、クライアント・サーバー型のアプリケーションに適用される．サーバープログラムが複製されて別 PE で動くことで、クライアントからは無故障のように見える．サーバーとして動作する耐故障プログラムと、外界でクライアントとして動作する非耐故障プログラムとのコミュニケーションに関しては、実現上以下を考慮しなければならない．

外界からの入力

RSM アプローチでは、クライアントから複製への送信では、それぞれ同じメッセージを送らねばならない．これは、仮定されたマルチキャスト機能を用いることができる．

PS アプローチでは、BS はチェックポイントから完全に状態を回復するならば、故障がない限りクライアントから BS へ送信する必要はない．PS が故障して BS が PS に切り替わる際には、クライアントから新しい PS へ通信路を張り換える操作が必要になる．

PS アプローチをとった研究例に、富士通研究所の高信頼性 UNIX「風雅」[2] がある．風雅は OS レベルで故障をマスクすることでユーザプログラムの修正、リコンパイルを不要にし、一般的なマルチサーバ UNIX サブシステムを対象とした移植性高いシステムとなっている．風雅では、PS の切替え時の送り先の変化を考慮した通信機構を実現している．切替え時には、故障によって消失した可能性のあるクライアントからの送信を再送することや、再送されたメッセージが冗長メッセージであった場合の処理などが必要になっている．

外界への出力

RSM アプローチでは、故障が起きない限り複製からクライアントへは同じメッセージが送られてくる． 複製が停止故障するとメッセージは、一方からのみ送られる．これらメッセージをクライアント側で調整する機能が必要となる．

PS アプローチでは、待機中の BS からはメッセージが送られて来ない．PS の故障によって BS が新しい PS へ切り替わる際には、新たな PS から重複メッセージが送られないようにする必要がある．このためには、協調チェックポインティングと出力コミットが必要である．

出力コミット

出力コミットは一般に、耐故障実行するプログラムから外界への出力について一貫性を保証するものである．出力コミットができるのは、チェックポインティングがコミットした直後であり、このときプロセスからの出力が行える．これによって、リカバリが起きても出力が重複して送られないことが保証される．出力コミットは、次のチェックポイントまで先送りすることで高速化を図ることができる [4]．しかし外界への出力が頻繁になると、チェックポイントが頻繁に行われるため、そのオーバヘッドがシステム性能を著しく落す．これに対して、出力コミットを効率化する研究も行われている [5] ．文献 [5] では、出力は各プロセスのバッファにセーブされ、実行状態をコミットした後で外界へ送り出す．このコミット操作はユーザプログラムの実行と並行して行われる．

2.2.4 故障検出

ここで扱う故障はプロセッサの停止故障に限っているが、それでも故障検出は一般に困難な課題であり、とくに非同期コミュニケーションを仮定する分散システムでは、遅延と故障による停止を区別することはできない．故障検出は、タイムアウトによる検出がよく用いられる．

2.2.5 システム再構成

RSM アプローチでは、故障が発生したプロセッサはそのまま放棄するだけでよい．

PS アプローチでは、待機している BS の一つを選択して PS として起動させなければならない． このとき、基礎機能としてメンバシップ機能が用いられる．メンバシップ機能により、故障していない BS プロセッサの選択が可能になる．

第 3 章

並列ソフトウェアの耐故障化

並列性を十分に持つ並列ソフトウェアは、中粒度から細粒度のプロセスによって構成されるプログラムと仮定できる． これらプロセスは、互いにメッセージ交換しながら自身の保持する状態値を更新する．プロセスは、決定的なものと非決定的なものが存在する．

Object/Action モデルは、データベースのような長命なデータを状態として、頻繁に書き換えないことを仮定している．このため、一般的な並列ソフトウェア向きではない．RSM アプローチは、決定的な SM しか扱えないという制限がある．PS アプローチの各プロセスは、チェックポインティング情報から再スタートできねばならないが、再スタートしたプロセス間のコネクションをつなぎ直すにはプロセス間接続情報のメンテナンスが必要になる．本研究では SM を非決定的プロセスでも扱えるように拡張し、これを構成要素として PS アプローチに適用する耐故障並列ソフトウェア (FTPS) を提案する．

3.1 非決定的 State Machine (NSM) アプローチ

オリジナルの SM は、入力列だけによって決定的に動作することが仮定されていた．これによって、複製された SM は同じ入力列が保証されるだけで、同じ状態と出力が期待できた．非決定的動作を行なう SM を仮定するならば、入力列が同じであっても複製間で同じ動作は期待できない．複製も同じ動作を行なわせるためには、一方の非決定的動作を再現する必要がある．そこで図 3.1のように、一方の SM の非決定的動作を示す情報（ログ）を もう一方の複製へ伝えることで、複製間で同じ動作を再現する．ここで、これを非決定的 State Machine（Non-deterministic State Machine （NSM））アプローチと呼ぶ．PS ア

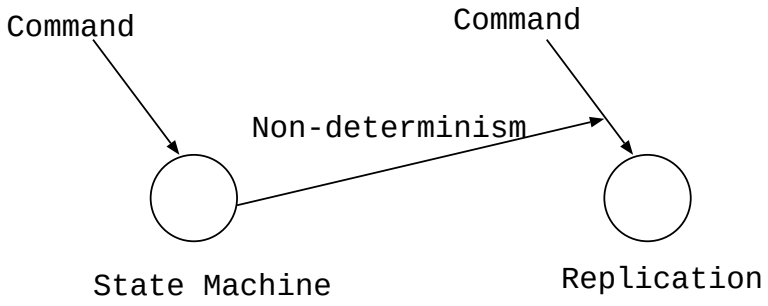

図 3.1: 非決定的 State Machine アプローチ

プローチにならって、ログを生成する方を PS、受けとったログをもとにして動作する方を BS と呼ぶ．また NSM に対して決定的 SM を（Deterministic State Machine）DSM と呼ぶ．PS は一般に NSM であり、これに対応する BS はログも入力とみなした DSM となる．

NSM アプローチは、PS アプローチでチェックポイントを PS から BS へ送ることに似ている．PS アプローチのチェックポイントは、それだけで SM の状態を復元できる情報であったが、NSM のログは実行過程を示すものであり復元情報として利用するには、それまでのすべてのログを必要とする．またチェックポイントのように即座に状態を復元することはできない．しかし、一般のチェックポイントのように、あらかじめアプリケーションで、状態を復元する操作を用意する必要がない．

システムの外界から PS、BS への入力は、それぞれコピーされてマルチキャストされる．PS、BS から外界への出力は、故障が無い限り同じメッセージが送られる．ログの転送について信頼性が保証されるならば、PS の故障時に BS が新たな PS に切り替わっても元の PS の出力と異る出力を送ることはない．またログの転送に信頼性が保証できない場合でも、外界への出力が無いならば重複した実行を行っても外界には影響しない．しかし、ログの転送に信頼性が保証できないシステムで外界への出力がある場合には、出力コミットが必要になる．

3.1.1 出力コミット

一般の並列分散システムでは、ログ転送は非同期通信が用いられる可能性が高く、ログ転送に信頼性が保証されない．このようなシステムに NSM アプローチを適用した場合、外界への出力があるならば、次のようなシナリオで一貫性が保証されない場合がある．

1. PS が非決定的実行をし、その実行ログを BS へ非同期通信で転送する.
2. PS は外界への出力 A を行う.
3. その後で PS が故障するが、上で送ったログが BS へ到着していない.
4. BS は PS の故障を検知して、新たな PS へ切り替わる.
5. BS は PS からの最後のログを受けとっていないので、上の 2へ至る非決定的実行を再現する保証が無い. よって 2とは異る出力 B をするかもしれないし、全く出力しないかもしれない.

このような一貫性の破綻の可能性に対して、次のような出力コミット手続きで対処できる. すなわち PS から外界への出力を、出力コミットされた後とすることで、一貫性が保証される. 出力コミット手続きは、以下のように行われる.

- PS からのログの転送に対して、BS は返信を返す.
- PS は、直前のログに対する BS からの返信を確認したとき、現状態をコミットする.
- PS での非決定的状態遷移と外界への出力は、コミット後にのみ許される.

この出力コミットがこれまで提案されているロールバックリカバリ・システムの出力コミットと異るのは、BS がロールバックしても「同じメッセージが再送されない」ことを保証していたのに対して、BS からも「確実に同じメッセージが送られる」ことを保証することにある.

協調チェックポイントによるロールバックリカバリ・システムでは、出力が頻繁に行われる際にはチェックポイントも頻繁に行われるため、このオーバヘッドがシステム性能を著しく落す. また楽観的メッセージロギングを用いるロールバックリカバリ・システムでは、出力メッセージのバッファリングを行うため、出力が遅延される. 文献 [5] は、分散アプリケーション向けに、オーバヘッドの少ないロールバック・リカバリ方法を提案している. この方法では、協調チェックポインティングと楽観的メッセージロギングを切替える. 出力が頻繁な時に早く出力コミットする必要がなければ、出力をバッファすることで、出力コミットを遅らせ、通常実行時のオーバヘッドを低く抑える. しかし頻繁な出力であって、かつ外界へ早く送らねばならないときには対応できない. 本論文で提案する手法では、

出力のたびにチェックポイントする必要はなく、非決定的実行ログの返信を確認するだけでよい．また、頻繁な出力であっても実行が決定的であるならば、ログの確認は最初に行うだけで済む.

3.2 NSM を用いた PS アプローチによる耐故障並列ソフトウェア

PS アプローチは、各 PE 上に一つのプロセスを置く粗粒度プロセス構成であれば、そのまま並列計算機で実現可能である．しかし PE 上で複数のプロセスが並行動作するような細粒度プロセス構成では、次のような問題点が生ずる.

1. 複数プロセスでサイト（PS、BS）を構成する場合、各プロセスは並行、並列動作するため、それらの実行順序は非決定的となる．並列に動作するプロセスの実行順序を再現することは、一般に困難である.
2. 1PE 上にある複数プロセスでサイトを構成する場合、サイト内プロセスは並列動作できないため並列効果が期待できない.

RSM アプローチの概念は、入力列が同じならば決定的なプロセスは実行順序によらず同じ状態遷移と出力を得ることを示している．NSM アプローチは、ログを追加して与えることで非決定的なプロセスについても同じ状態遷移と出力を保証する．これらを用いることによって、プロセスについて実行順序が変わっても、全体として同じ出力が保証できる.

またサイト内での並列効果を求めるために、複数 PE 上の複数プロセスによってサイトを構成する．すなわち、並列計算機の PE は複数のグループに分割され、一つのグループを PS、残りを BS とする．以後簡単のため、PS、BS 一つづつの構成のシステムについて考察する.

構成 ：NSM を用いた PS システムの構成は次のようになる.

PS の各プロセスは、完全に動作が決定的であれば SM、非決定的動作を含むならば NSM に基づくプロセスとして動作する．NSM プロセスは、その実行ログを対応する BS のプロセスへ転送する．BS の各プロセスは、PS のプロセスそれぞれに対応した SM に基づくプロセスである.

故障検出： PS、BS のそれぞれには故障監視プロセスがあり、故障検出を行う．プロセッサの停止故障を扱うが、プロセッサの停止はその上のプロセスすべてを停止させるため、故障監視プロセスからはプロセスの停止によって故障が検出される．

出力コミット： PS 上の NSM プロセスからは、PS 内の別プロセスも外界とみなされるため、それぞれのプロセスで出力コミットが必要である．

故障回復： 故障監視プロセスによって PS のプロセスの故障が検出されると、BS の全プロセスに割り込みが発生して、以後 PS プロセスへと切替えられる．

3.3 関連する研究

PS アプローチをとった岸本らの風雅 [2] の構成は PE 上の複数プロセスをサイト化したものであり、超並列計算機をターゲットとした研究ではない．また彼らの研究が OS レベルの故障マスクであるのに対して、本研究はプログラミング言語レベルの故障マスクである点も異る．将来本格的な並列 OS を開発するには並列言語を基にするべきであり、プログラミング言語処理系レベルの故障マスクは意義が大きいと考える．

鈴木、片山、Schlichting は、ソフトウェアフォールトトレランスに対するモデルを提案し、並列ソフトウェアによる耐故障を目指している [6]．彼らのアプローチは関数プログラミングをターゲットとしているため、プログラムを関数の集合とみなし、それらを計算木に分解している．このアプローチは、（逐次）論理プログラミングにおいて探索問題を解く場合の、ゴールのリダクション木に良く似ている．本研究ではプロセスモデルで解釈される並列論理型言語を用いるが、プロセスネットワークとみなせる超並列プログラミングは、プロセスどうしが互いにコミュニケーションし、動的にプロセス結合することから、リダクション木を分解するアプローチでは対応し難いと考えられる．KL1 に代表されるこれら並列論理型言語では、一般的に単一代入が基本的特性であり、逐次論理型言語のようなバックトラックも許されていないことが多くロールバックメカニズムの導入も困難である．一般に破壊的なメモリ書き込みは、並列プロセス間の同期を必要とすることから、並列論理型言語のような単一代入の仮定は、超並列プログラミング言語の一つの有力な選択である．また、彼らの研究では安定記憶を前提としているが、超並列システムではその実現は必ずしも容易ではない．本研究では安定記憶の代わりに緩い同期のログを導入することで、実

装を容易にしている．共通点があるとすれば、複数のレプリカが並行して動作し、その一つが結果の出力を返すというところである．

Sharma と Paradhan は、ハードウェア独立でいかなるネットワークにも適用できる協調チェックポインティング戦略の提案をしている [4]．彼らの実装方法では、チェックポイントコーディネーターがプロセスのチェックポイントを管理する構成になっているが、本研究で用いる監視プロセスの構成と類似している． しかし本研究で提案する方法は、対応する 2 プロセス間だけの協調であり独立チェックポイントに近い．マルチコンピュータでのチェックポインティング戦略では、独立チェックポインティングの方がストアが頻繁になる分性能を落すとされるが、本研究のようにディスクへのストアを伴わないならば、プロセッサ間コミュニケーション量が性能低下の要因として大きい．また本研究では、リカバリを行わないためリカバリラインの形成にプロセッサ間コミュニケーションを消費する必要も無い．

Maier は、アトミックアクションに基づいた耐故障性の研究を行っている [3]．彼もまたプログラマが耐故障を実現することの困難さに着目して、ユーザに分かりやすいチェックポインティングライブラリの提供を目指している．

3.4 本研究の特色と位置付け

本章では、決定的な SM を仮定した RSM アプローチを拡張して、非決定的な SM を扱う NSM アプローチを提案した．RSM アプローチから PS アプローチへと発展したように、NSM アプローチから PS アプローチへと発展させた耐故障並列ソフトウェアのアイデアについて概説した．PS アプローチは、風雅の例のようによく用いられるが、PS を複数プロセッサから構成して並列実行可能にした点が本研究の特色である．また、並列論理型言語という純粋に並列言語として設計された言語を対象としたこと、一般的な超並列システムを想定して安定記憶を仮定しないことは、より現実的なアプローチだと思われる．さらに耐故障ソフトウェアの研究として重要なことは、プログラマの負担を軽減することである．特に、並列プログラムの開発には様々な困難を伴うことから、耐故障まで意識した並列プログラミングは過大な負担となる．そこで本研究は、純粋に並列パラダイムに基づいた並列言語に基づくことでプログラム開発効率を向上させ、さらに自動プログラム変換を目指している．

第 4 章

耐故障並列ソフトウェアの正当性

本章では前提となる故障モデルを示し、その下で第3章で提案した並列ソフトウェアの耐故障化アプローチの正当性を示す．そのため、並列プロセス、並列プログラム実行、耐故障実行モデル等を形式化し、故障時の実行が無故障時の実行と無矛盾であることを示す．またプロセス間通信を減らす最適化についても、その正当性を示す．

4.1 故障モデルと耐故障化の前提

本研究ではプロセスの非決定的動作を扱う．複製されたプロセスの非決定的動作は、故障による異常動作と区別できない．このためビザンチン故障や通信脱落故障を判別することが困難なことから、本研究では PE 停止故障（fail silent）に対する故障マスクを考える．ネットワークの切断故障については、リンク切断によって PE が孤立するものと仮定すれば、PE の停止故障として扱える．

本研究では故障マスクが主な目的であり、故障検出ではタイムアウトを用い、故障復旧は前提としない．一般に故障検出は困難な課題であり、特に非同期ネットワークを仮定した大規模な疎結合並列計算機では、故障停止と遅延の区別が有限時間でできないため、遅延も故障とみなして対象 PE を排除し得るものとする．一般の超並列計算機では、停止故障した PE をソフトウェアで復旧するための機能を仮定していないため、故障復旧については考慮しない．

4.2 SM アプローチと NSM アプローチ

以下に SM アプローチ、NSM アプローチに関する定義を与える.

定義 4.1 (選択) 順序づけられた有限集合 $A = \{a_1, \ldots, a_n\}$ から要素 a の**選択**を、

$$choose(A) = a$$

で表す. またこのとき、

$$a = a_i \text{であったならば、} label(a) = i$$

が成り立つとする.

$choose$、$label$ は述語であり、関数でないことに注意すること.

4.2.1 プロセス

定義 4.2 (プロセス) プロセスは、次の集合および関数から定義される.

プロセスの入力の集合 I

プロセスの出力の集合 O

内部状態の集合 S

ヒストリ集合 H

状態遷移関数 $T : I \times S \longrightarrow \{S \times O\}$

$\{S \times O\}$ の要素は有限であり、適当に順序づけらているものとする.

- プロセス状態は、$H \times I \times S \times O$で定義される.
- 状態遷移は一般に非決定的であり、得られた集合の要素から任意に一つ選んで次の内部状態、出力とする.

定義 4.3 (プロセスの実行) プロセス P**の実行**は、プロセス状態 $H \times I \times S \times O$の列であり、

$$P(1, n) = \langle h_1, I_1, S_1, O_1 \rangle, \langle h_2, I_2, S_2, O_2 \rangle, \ldots, \langle h_n, I_n, S_n, O_n \rangle$$

と表す. ただし $O_1 = \phi$

ヒストリは、プロセスの実行について一意である.

定義 4.4 (コミットチョイス実行モデル) すべてのプロセスについて、その状態遷移が必要な入力がすべて確定したときにのみ許されるとき、
コミットチョイス実行モデルに従うと言う.

すなわち次が成り立つとき、プロセス Pの実行がコミットチョイス実行モデルに従う.

$$\forall i(\geq 1)\ I_i\text{の要素がすべて確定した} \overset{\text{if and only if}}{\Longleftrightarrow} choose(T(I_i, S_i)) = \langle S_{i+1}, O_{i+1}\rangle$$

補題 4.1 並列論理型言語に基づくプロセスは、コミットチョイス実行モデルに従う.

証明）並列論理型言語のセマンティクスから自明.

以後、コミットチョイス実行モデルに従うとする.

定義 4.5 (プロセス実行状態の参照) プロセスの実行
$P(i,j) = \langle h_i, I_i, S_i, O_i\rangle, \ldots, \langle h_k, I_k, S_k, O_k\rangle, \langle h_{k+1}, I_{k+1}, S_{k+1}, O_{k+1}\rangle, \ldots, \langle h_j, I_j, S_j, O_j\rangle$ に関して、次のようにヒストリ h_kによって**プロセス状態を参照**する.

$$status_P(h_k) = \langle h_k, I_k, S_k, O_k\rangle$$

定義 4.6 (プロセス実行の決定性) プロセス状態 $\langle h_i, I_i, S_i, O_i\rangle$ に対して $choose(T(I_i, S_i))$ が一意に $\langle S_{i+1}, O_{i+1}\rangle$ を定める、ならば、このときの**実行** $P(i, i+1)$ **は決定的**であるという. さもなければ、**非決定的**であるという.

定義 4.7 (プロセスの決定性) 任意の状態について決定的であるならば、このとき**決定的プロセス**と呼ぶ. そうでなければ、**非決定的プロセス**と呼ぶ.

補題 4.2 $\forall i(1 \leq i)\ |T(I_i, S_i)| = 1 \overset{\text{if}}{\Rightarrow}$ P は決定的プロセスである.

証明）$|T(I_i, S_i)| = 1$ ならば、$choose(T(I_i, S_i))$ は一意にしか選べない.

定義 4.8 (プロセス状態の同値) 二つのプロセス状態は、内部状態と出力が同じであるとき、その時に限り**同値な状態である** ($\cong$) と言う.

$$status_P(h) \cong status_Q(h') \overset{\text{if and only if}}{\Longleftrightarrow} \begin{cases} status_P(h) = \langle h, I, S, O\rangle, \\ status_Q(h') = \langle h', I', S, O\rangle \end{cases}$$

定義 4.9 (プロセスの部分同値) 二つのプロセスが同じ入力に対して同値な状態に遷移するとき、その時に限り**部分同値である**と言う.

$$P(i,i+1) \cong P'(i,i+1) \overset{\text{if and only if}}{\Longleftrightarrow} \left.\begin{array}{l} status_P(h_i) = \langle h_i, I_i, S_i, O_i \rangle \\ status_Q(h_i) = \langle h'_i, I_i, S'_i, O'_i \rangle \end{array}\right\} \overset{\text{if}}{\Rightarrow} status(h_{i+1}) \cong status(h'_{i+1})$$

定義 4.10 ある**区間における二つのプロセスの部分同値**を、次のように定義する.

$$P(i,j) \cong P'(i,j) \quad \overset{\text{if and only if}}{\Longleftrightarrow} \quad \forall t(i \leq t < j)\ P(t,t+1) \cong P'(t,t+1)$$

補題 4.3 プロセスの同値な状態からの決定的な実行は部分同値である.

証明）決定的な実行は遷移は入力に対して一意であるから、

同値な状態 $\langle h, I, S, O \rangle$、$\langle h', I, S, O \rangle$ からの実行は、同値な状態に遷移する.

定義 4.11 (プロセスの同値) プロセスの実行全域において部分同値ならば、単に**同値** ($\equiv$) と言う.

$$P \equiv P' \quad \overset{\text{if and only if}}{\Longleftrightarrow} \quad \forall i,j(i \leq j)\ P(i,j) \cong P'(i,j)$$

定義 4.12 (プロセスの複製) 同値な初期状態が与えられ、同じ状態遷移関数に従うプロセスを互いに**複製**と呼ぶ. すなわち、T_P, T_Qをプロセス P、Q の状態遷移関数とすると、

$$P \text{と} Q \text{は複製} \overset{\text{if and only if}}{\Longleftrightarrow} \left\{\begin{array}{l} status_P(h_1) \cong status_Q(h'_1) \\ T_P = T_Q \end{array}\right.$$

複製どうしは、初期状態が同値で状態遷移関数も同じであるが、状態遷移結果が一意に決まらないので、複製 P、Q に対して同じ入力を与えても、同じ遷移をする保証は無いことに注意すること.

定義 4.13 (プロセスの模倣) プロセス Pとプロセス Q に対して、それぞれの状態遷移において同じ入力に対して同じ要素の選択を行なうとき、Q は Pを**模倣**すると言う.

$$Q \text{は} P \text{を模倣する} \quad \overset{\text{if and only if}}{\Longleftrightarrow} \quad \forall i\ choose(T_P(I_i, S_i)) = choose(T_Q(I_i, S'_i))$$

補題 4.4 複製プロセスの一方が他方を模倣するとき、それらのプロセスは同値になる.

証明）プロセス P、Q が複製で、Q が Pを模倣するときを考える．模倣することから、$\forall i\ choose(T_P(I_i, S_i)) = choose(T_Q(I_i, S'_i))$．一方、複製であることから、$T_P = T_Q$ かつ $status_P(h_1) = status_Q(h'_1)$．よって、$\forall i\ status_P(h_i) \cong status_Q(h'_i)$ より $P \equiv Q$

補題 4.5 (SM アプローチ) 決定的なプロセスの複製は、入力が同じであれば互いに同値になる.

証明）プロセス P、Q が複製で、かつ決定的であるとする.

定義 4.7より、$\forall i\ choose(T_P(I_i, S_i))$、$\forall i'\ choose(T_Q(I'_i, S'_i))$ はそれぞれ一意である．複製であるから $T_P = T_Q$かつ $status_P(h_1) \cong status_Q(h'_1)$ である．よって、帰納法により $\forall i\ choose(T_P(I_i, S_i)) = choose(T_Q(I'_i, S'_i))$ が示される．よって入力が同じ（$I_i = I'_i$）であるなら模倣であり、補題 4.4より、Pと Q は同値になる.

定義 4.14 (トレース遷移関数) ある状態遷移関数Tのプロセスの選択の結果を入力として、同じ選択を行なう**トレース遷移関数**T'を次のように定義する.

$$T'(I, S, label(t)) = \{t\}\ (\text{when } choose(T(I, S)) = t)$$

トレース遷移関数は、$label(t)$ を入力として見れば定義 4.6より決定的なプロセス実行を実現する.

補題 4.6 プロセス Pの遷移関数を T_P、そのトレース遷移関数を T'_Pとする．このとき Pの複製 Q の遷移関数T_Qについて

$$\forall i, \exists h\ choose(T_Q(I'_i, S'_i)) = choose(T'_P(I_i, S_i, h))$$

が成り立ち、同じ入力が与えらるならば、Q は Pを模倣する.

証明）定義 4.14より、

$$\begin{aligned} choose(T_P(I_i, S_i)) &= choose(T'_P(I_i, S_i, label(choose(T_P(I_i, S_i))))) \\ &= choose(T_Q(I'_i, S'_i)) \\ &= choose(T_Q(I_i, S_i)) \quad (I_i = I'_i \text{より}) \end{aligned}$$

定義 4.13より Q は Pを模倣する.

補題 4.7 プロセス Pのトレース遷移関数により遷移関数が定義される複製プロセス Q は、入力が同じとき Pと同値である.

証明） 補題 4.4 補題 4.6より明らか.

定義 4.15 (プライマリプロセスとバックアッププロセス) 次のようなプロセス Pを**プライマリプロセス**、Bを**バックアッププロセス**と呼ぶ.

- Pと Bは複製
- Pの遷移関数を T_P、そのトレース遷移関数を T'_P とするとき、Bの遷移関数が $T_B = T'_P$

補題 4.8 (NSM アプローチ) 入力が同じであれば、バックアッププロセスはプライマリプロセスと同値である.

証明）補題 4.7より明らか.

補題 4.9 プライマリプロセスの実行 $P(i, i+1)$ が決定的であるときに限り、Bの遷移関数を $T' = T$ としても補題 4.8は保存される.

証明） $P(i, i+1)$ が決定的であるので、遷移 $choose(T(I, S))$ は一意に決まる. これを t とすると、$\forall x\ T'(I, S, label(x)) = t = choose(T(I, S))$ であり、$label$は必要ない. よって$T' = T$としても成り立つ.

4.2.2 並列プログラム

定義 4.16 (並列プログラム) **並列プログラム** PPは、次の集合から構成される.

プログラムの入力の集合 I

プロセスの状態の集合 P

プログラムの出力の集合 O

状態遷移関数 $T_{pp} : I \times P \longrightarrow \{P \times O\}$

ヒストリ集合 H

並列プログラムの実行は、$H \times I \times P \times O$ の列

$$PP(1, n) = \langle h_1, I_1, P_1, O_1 \rangle, \langle h_2, I_2, P_2, O_2 \rangle, \ldots, \langle h_n, I_n, P_n, O_n \rangle$$

とし、

$$\text{初期状態 } PP(1,1) = \langle h_1, I_1, P_1, O_1 \rangle \text{ において } |P_1| = 1 \text{ とする.}$$

すなわち、並列プログラムを構成するプロセスは、一つのプロセスを起源として生成されたとする.

すなわち、

$$\forall p \in PP \ \exists q \in PP \text{ s.t. } q\text{が } p \text{ を生成した}$$

定義 4.17 (並列プログラムの決定性) 構成するすべてのプロセスが決定的であるとき、その**並列プログラムは決定的である**と言う.

定義 4.18 (並列プログラムの実行の一貫性) 同じ入力に対して、同じ出力をすることが保証された**並列プログラムの実行は一貫性がある**と言う.

補題 4.10 決定的な並列プログラム実行は、一貫性がある.

証明） 並列プログラムを構成するプロセス数に関する帰納法による.

プロセス一つから構成される場合は自明. 決定的な並列プログラム $PP(|PP| = n)$ が一貫性があるとする. 今 決定的プロセス $q \notin PP$を考えるとき、$PP \cup \{q\}$ は、決定的な並列プログラムである.

$PP \cup \{q\}$ が一貫性をくずす. すなわち入力 Iに対して、Oと $O'(O \neq O')$ を出力したと仮定する. $\exists o \in O, \notin O'$ とするとき、PPまたは p が、o を出力したことになる. さらに PPまたは p への入力が異っていたことになり、入力 Iとは別に、内部で発生した出力αが PPまたは p に与えられたことになる. αの発生も PPまたは p であるから、さらに入力が異ったことになる. このようにたどると、最後には Iの渡し方が異ったことになり矛盾する.

補題 4.11 二つの並列プログラムの間の各プロセスが同値であるとき、並列プログラム実行どうし一貫性がある.

証明） 補題 4.10の証明と同様に、並列プログラムを構成するプロセス数に関する帰納法による.

プロセス一つから構成されるときは自明. 並列プログラム PP、$PP'(|PP| = |PP'| = n)$ において、その各プロセスが同値で、かつ一貫性があるとする.

ここで同値なプロセス $p \equiv p'$ をそれぞれ PP と PP' に加えたとき、一貫性がくずれると仮定し矛盾を導く．一貫性がくずれることから、$PP \cup \{p\}$ と $PP' \cup \{p'\}$ に同じ入力を与えたとき、それぞれ O、$O'(O \neq O')$ を出力する．$\exists o \in O, \notin O$ に対して、PPが出力したとすると、PPへの入力と PP'の入力が異ったことになる．最初の入力は同じであるから、p から PPへの入力と p'から PP'への入力が異ったことになる．p と p'が異る出力をするなら異る入力があったことになる． よって PPから p への入力と PP'から p'への入力が異ったことになる．同様にしてたどると、最初の入力が異ったことになり矛盾する．o を p が出力したとしても同様に矛盾する．

定義 4.19 (プライマリサイト (PS)、バックアップサイト (BS))
プライマリプロセスからなる並列プログラムを**プライマリサイト（PS）**、バックアッププロセスからなる並列プログラムを**バックアップサイト（BS）**と呼ぶ．

定理 4.1 対応する PS と BS は一貫性がある．

証明）補題 4.8と補題 4.11より明らか．

補題 4.12 プライマリプロセスの実行 $P(i, i+1)$ が決定的であるときに限り、Bの遷移関数を $T' = T$ としても定理 4.1は保存される．

証明）補題 4.8 と定理 4.1から明らか．

4.3 耐故障実行モデル

PS と BS として構成される並列プログラムが、無故障時に一貫性が保証されることは定理 4.1に示した．ここでは、上で定義した並列プログラムを拡張して、耐故障実行モデルを定義する．仮定する故障モデルはプロセッサの停止故障 (fail silent) であるため、プロセッサ上のプロセス故障は定義 4.20のように定義される．なお実際のプロセッサ故障では、プロセッサ上のプロセスすべてが故障する．

4.3.1 故障モデルと耐故障

定義 4.20 (プロセスの故障) プロセスの実行において、状態 $status(h_i)$ から先へ遷移しないときを**プロセスの故障**と定義する.

ネットワーク故障の一部も、外界から入力が来なくなることでプロセスの遷移が停止することから、同じ現象を起こし得る.

非同期通信に基づくネットワークにおいては、出力は必ずしも即座にプロセッサの外へは出ない. そこで次を定義する.

定義 4.21 (非同期通信における出力の送出) 出力がプロセスにより行われることを**送出**されると言う.

また次のブーリアン関数を定義する.

$$send_P(x) = \begin{cases} true & P\text{が } x \text{ を送出した} \\ false & P\text{が } x \text{ を送出していない} \end{cases}$$

定義 4.22 (非同期通信における出力の排出) 出力がプロセッサ外へ実際に送られることを**排出**されると言う.

また次のブーリアン関数を定義する.

$$sent_P(x) = \begin{cases} true & P\text{が } x \text{ を排出した} \\ false & P\text{が } x \text{ を排出していない} \end{cases}$$

送出した出力は故障が無い限り、いつか排出される. すなわち次が成り立つ.

$$\forall x \ sent_P(x) \overset{\text{if}}{\Rightarrow} \ send_P(x)$$

プロセスが故障したとき、ある出力が故障により正常に排出されなかったならば、それ以後のすべての出力は排出されない.

また排出された出力は、ネットワークの最大レイテンシ以内に相手先プロセッサに届くものとする.

定義 4.23 (並列プログラムの故障) 並列プログラムを構成する一つ以上のプロセスが故障したとき、その**並列プログラムは故障した**と定義する.

マルチプロセッサ上に分散する並列プログラムにおいて、通信脱落故障のようなネットワーク故障の一部も、定義 4.20の意味のプロセス故障を招く.

定義 4.24 (出力に無矛盾な実行) 故障した並列プログラム実行に対して、一貫性がある故障していない並列プログラム実行が存在するとき、故障した並列プログラムは**出力に無矛盾な実行を継続**すると言う.

定義 4.25 (PS、BS の耐故障性保証) 並列プログラムがPS、BS から構成され、その中のプロセスの故障に対して出力に無矛盾な実行が継続されるとき、**PS、BS が耐故障性がある**と言う.

定義 4.26 (プロセスの耐故障) プライマリプロセスとそれに対するバックアッププロセスは、一方がプロセス故障となっても出力に矛盾なく実行が継続されるとき**プロセスに耐故障性がある**と言う.

定義 4.27 (観測) プロセス Pのプロセス外からの入力σに対する**観測**のブーリアン関数を

$$observe_P(\sigma) \;=\; \begin{cases} true & \text{P が}\sigma\text{を受けとった} \\ fail & \text{P が}\sigma\text{を受けとっていない} \end{cases}$$

と定義する.

補題 4.13 (観測と排出の因果関係) 次の因果関係が成り立つ.

$$observe(\sigma) = true \;(\overset{\text{if}}{\Rightarrow}\; sent(\sigma) = true) \overset{\text{if}}{\Rightarrow}\; send(\sigma) = true$$

証明）定義 4.22より明らか.

4.3.2 故障検出機構

ここでは、プライマリサイト、バックアップサイトの構成の並列プログラム上での故障検出機構をモデル化する．これによって、タイムアウト機構とソフトウェアでプロセッサ停止故障の検出が可能であることを示す.

定義 4.28 (監視プロセス) 各プライマリサイト、バックアップサイト内のプロセッサは、**故障監視プロセス** W_Oを持つ．W_Oは無故障であれば、観測した信号に対して信号を送り返す.

定義 4.29 (サイト監視プロセス) プライマリサイト内の任意の 1 プロセッサとバックアップサイト内の任意の 1 プロセッサは、それぞれ**サイト監視プロセス** W_P、W_B を持つ.

W_P は、プライマリサイト内の全ての W_O に信号を送り、それらから返る信号すべてを観測したならば、W_B へ信号を送る．これを**監視サイクル**と呼ぶ．この W_B への信号は周期的に行われる．W_B も、バックアップサイト内の全ての W_O との信号の交換し、かつ W_P からの信号を観測する．このとき W_B は、タイムアウト検出を行う.

補題 4.14 W_O, W_P, W_B により、プライマリサイト内のプロセッサの停止故障を検出し、適切なバックアップサイトを指定することができる.

証明）プライマリサイト内のプロセッサの停止故障が発生するとき、

- W_P の動作するプロセッサが停止故障していなければ、W_P は次の監視サイクルで、停止故障したプロセッサ上の W_O から返信が来ないことで故障を検出する．このとき、周期的に送る予定の信号を W_B へ出力しない.
- W_P の動作するプロセッサが停止故障した場合には、周期的に送る予定の信号を W_B へ出力できない.

その動作するプロセッサが停止故障していない W_B は、W_P からの周期信号のタイムアウト検出により、プライマリサイトの故障を検出できる．また、このときの W_B は、自サイトの各 W_O と信号交換することで、自サイトの故障を検出できる．従って、自サイトが故障停止していないと判定できる W_B が、適切なバックアップサイトの候補となることができる.

実際には、プロセスの存在しないプロセッサの故障はプログラムの動作に影響しないとして無視してよく、実現上ではプロセスの存在するプロセッサにのみ W_O を置く．また、ここで述べた故障検出機構以外を用いても以後の展開は同じである.

4.3.3 出力コミットと耐故障実行

定義 4.30 (返信) トレース遷移関数において、もとの遷移関数の $label(t_i)$ を観測したならば、これに対して**返信**を送信する．すなわち、

$$send(acknowledge_i) = true \overset{\text{if}}{\Rightarrow} observe(label(t_i)) = true$$

定義 4.31 (コミット遷移関数) プロセス Pの状態遷移関数Tに対して、**コミット遷移関数** T^Cを次のように定義する.

$$T^C(I_i, S_i) = \{t_{i+1}\} \left(\begin{array}{ll} & observe_P(acknowledge_{i-1}) = true, \\ \text{when} & choose(T(I_i, S_i)) = t_{i+1}, \\ & send_P(label(t_{i+1})) = true, \\ & observe_P(acknowledge_i) = true \end{array} \right)$$

ただし、つねに $observe(acknowledge_0) = true$

またこのとき、$label(t_{i+1})$ は、$acknowledge_{i-1}$に対する返信になる.

定義4.31より $\langle h_i, I_i, S_i, O_i \rangle$ から $\langle h_{i+1}, I_{i+1}, S_{i+1}, O_{i+1} \rangle$ へのT^Cによる遷移は $send(label(t_{i+1}))$ によって $label(t_{i+1})$ が送出され、それに対する返信 $acknowledge_i$が返った後となる. 従って出力 O_{i+1}は、その $acknowledge_i$の確認後となる. すなわち、次の補題が成り立つ.

補題 4.15 (出力コミット) コミット遷移関数に従うプロセス Pの状態遷移においては、次が成り立つ.

$$\begin{array}{rl} & send(O_{i+1}) = true \\ & \stackrel{\text{if}}{\Rightarrow} \quad observe(acknowledge_i) = true \\ (& \stackrel{\text{if}}{\Rightarrow} \quad send(label(t_{i+1})) = true \\ & \stackrel{\text{if}}{\Rightarrow} \quad observe(acknowledge_{i-1})) \end{array}$$

すなわち $acknowledge_i$の確認によって、O_{i+1}の出力がコミットされる.

証明） 定義 4.31より明らか.

定義 4.32 (コミットトレース遷移関数) コミット遷移関数 T^Cに従って実行するプロセス Pに対して、プロセス Q 上の**コミットトレース遷移関数** T'^Cを次のように定義する.

$$T'^C(I_i, S_i, label(t_{i+1})) = \{t_{i+1}\} \left(\begin{array}{ll} & observe_Q(label(t_{i+1})) = true, \\ \text{when} & choose(T^C(I_i, S_i)) = t_{i+1}, \\ & send_Q(acknowledge_i) = true \end{array} \right)$$

定義 4.33 (耐故障実行モデル) 耐故障実行モデルは、次で定義される.

プライマリプロセス P : 遷移関数 T^C で実行する.

バックアッププロセス B : 遷移関数 T'^Cで実行する.

監視プロセス W_O, W_P, W_B : 定義 4.28、定義 4.29に従う.

なおヒストリの更新は、遷移が完了したときとする.

補題 4.16 無故障時に全てのプライマリプロセス P、バックアッププロセス Q についてT、T'をそれぞれT^C、T'^Cと置き換えても、定理 4.1 が成り立つ.

証明）故障がなければ、送出した $label$ と $acknowledge$ は排出される．また $observe$ は、無故障時には無条件で成り立つことが保証される．定義 4.31から、つねに $observe(acknowledge_0) = true$ であり、また、$\forall j \geq 1$ について、次が成り立つ.

$$
\begin{aligned}
& observe(acknowledge_j) = true \\
\overset{\text{if}}{\Rightarrow} \quad & observe(label(t_{j+1})) = true \quad \text{(補題 4.13、定義 4.30 より)} \\
\overset{\text{if}}{\Rightarrow} \quad & observe(acknowledge_{j-1})) = true \quad \text{(補題 4.13、補題 4.15 より)}
\end{aligned}
$$

よって、$acknowledge$ と $label$は交互に順番通り観測されることが保証される.
よって T^C、T'^Cの遷移から、列 $t_1, t_2, \ldots, t_n$が 得られ、プライマリプロセス、バックアッププロセスともこの列の遷移となる．各 t_iはTにより生成されたのだから、Tによるプロセスの実行であり．Tによりプライマリプロセスとバックアッププロセスを構成した結果と等しくなる．よって 定理 4.1が成り立つ.

補題 4.17 耐故障実行モデルにおいて、バックアッププロセス Bが状態 $\langle h'_i, I_i, S_i, O_i \rangle$ にあるならば、プライマリプロセス Pは $\langle h_{i-1}, I_{i-1}, S_{i-1}, O_{i-1} \rangle$ または $\langle h_i, I_i, S_i, O_i \rangle$ の状態にある.

証明）バックアッププロセス Bの状態 $\langle h_i, I_i, S_i, O_i \rangle$ から

定義 4.32より $observe_B(label(t_i)) = true$ が保証される.

よって、補題 4.15、補題 4.13より

$$observe_B(label(t_i)) = true \overset{\text{if}}{\Rightarrow} send_P(label(t_i)) = true$$

より、少なくとも Pは $label(t_i)$ の送出後であることが保証される.

これは、$\langle h_{i-1}, I_{i-1}, S_{i-1}, O_{i-1}\rangle$ からの遷移途中を指す.

また定義 4.32から $send_B(acknowledge_{i-1}) = true$

および $send_B(acknowledge_i) = false$ が保証できる.

このことから定義 4.31より、Pは $\langle h_i, \ldots\rangle$ へ遷移することができても、$\langle h_{i+1}, \ldots\rangle$ へは遷移できないことを保証する.

すなわち Pは、$\langle h_{i-1}, \ldots\rangle, \langle h_i, \ldots\rangle$ のどちらかの状態にあることが保証される.

定義 4.34 (プロセスの引き継ぎ) 故障発生後、故障していないバックアッププロセスの T'^C を T^Cへ変更することを**プロセスの引き継ぎ**と呼ぶ.

補題 4.18 耐故障実行モデルにおいて、プライマリプロセス Pの故障が検出された後、バックアッププロセス Bは引き継がれると、プロセスに耐故障性がある.

補題 4.14より、耐故障実行モデルの W_O, W_P, W_Bはプライマリプロセス Pが故障していることを観測でき、適当なバックアッププロセスを指定できる. 指定されたバックアッププロセス Bの現状態が $\langle h'_i, I_i, S_i, O_i\rangle$ ならば、補題 4.17 より Pの状態は、$\langle h_{i-1}, I_{i-1}, S_{i-1}, O_{i-1}\rangle$ か $\langle h_i, I_i, S_i, O_i\rangle$ である.

さらに Bは、$label(t_i)$ に対する返信を返していないことで、Pが $label(t_i)$ を送った直後から、$acknowledge_i$を待つ間のどこかで停止故障したことを保証できる. ここで Pからの O_iの送出は、判定不能であるが、Bは状態 $\langle h'_i, I_i, S_i, O_i\rangle$ と、すでに O_iを送出済みであり、プロセス外から見れば、出力の喪失や重複は見られない. この状態からプロセスの引き継ぎを行っても、出力に矛盾のない実行を継続できる.

定理 4.2 PS と BS の全てのプロセスが、耐故障実行モデルに従い、PS の故障時にプロセスの引き継ぎが行われると、PS と BS に耐故障性がある.

証明）耐故障実行モデルの W_O, W_P, W_B により、PS の故障検出後に適当な BS が指定できる． その BS 中の全バックアッププロセスは、対応するプライマリプロセスの引き継ぎを行うことができる．よって補題 4.18より各プロセスに耐故障性があり、出力に矛盾なく実行を継続できる．定理 4.1より、BS は無故障の並列プログラム PS のある実行と一貫性がある．また BS が故障した場合には、PS の出力と無故障の並列プログラム PS のある実行と一貫性がある． よって PS、BS に耐故障性がある．

4.4 最適化

定義 4.33の耐故障実行モデルの状態遷移では、遷移ごとにラベルと返信の送信が起き、処理効率を低下させる．そこで、以下に最適化方法について述べ、それによって耐故障性が損なわれないことを述べる．

定義 4.35 (ヒストリの色分け) プロセスの実行 $P(i, i+1) = \langle h_i, \ldots \rangle, \langle h_{i+1}, \ldots \rangle$ が非決定的実行のときを $h_i^{n_j}$ と表す．ここで jは、非決定的実行のシーケンス番号である．決定的実行であるとき、ヒストリ h_iを $h_i^{d_j}$ と表す．ここで jは、h_iより前に実行された非決定的実行のシーケンス番号、または 0 である．決定的実行のヒストリの集合を H_D、非決定的実行のヒストリの集合を H_N とすると、

$$h_i = \begin{cases} h_i^{n_j} & h_i \in H_N \\ h_i^{d_j} & \exists\ s(<i)\ h_s = h_s^{n_j}, \forall\ t(s < t \leq i) h_t \in H_D \\ h_i^{d_1} & \forall\ s(<i)\ h_s = h_s^{d_1} \end{cases}$$

定義 4.36 (OPT コミット遷移関数) 状態遷移関数 Tに従うプロセス Pについて、**OPT コミット遷移関数** T^{OC}を次のように定義する．

$T^{OC}(I_i, S_i) = \{t_{i+1}\}$ when

$$
\begin{array}{ll}
(1) & choose(T(I_i, S_i)) = t_{i+1} \qquad \text{for } \langle h_i^d, I_i, S_i, O_i\rangle, O_{i+1} = \phi \\
(2) & \left\{ \begin{array}{l} observe_P(acknowledge_{j-1}) = true, \\ choose(T(I_i, S_i)) = t_{i+1} \end{array} \right. \qquad \text{for } \langle h_i^d, I_i, S_i, O_i\rangle, O_{i+1} \neq \phi \\
(3) & \left\{ \begin{array}{l} observe_P(acknowledge_{j-1}) = true, \\ choose(T(I_i, S_i)) = t_{i+1}, \\ send_P(label(t_{i+1})) = true, \\ observe_P(acknowledge_j) = true \end{array} \right. \qquad \text{for } \langle h_i^{n_j}, I_i, S_i, O_i\rangle
\end{array}
$$

ただし、つねに $observe(acknowledge_0) = true$

定義 4.37 (OPT コミットトレース遷移関数) プロセス Pに対して、コミット遷移関数T^Cに従って実行するプロセス Q 上の **OPT コミットトレース遷移関数** T'^{OC}を次のように定義する.

$$
\begin{array}{l}
T'^{OC}(I_i, S_i, label(t_{i+1})) = \{t_{i+1}\} \text{ when} \\
\quad (1) \quad choose(T(I_i, S_i)) = t_{i+1} \qquad \text{for } \langle h_i^d, I_i, S_i, O_i\rangle \\
\quad (2) \quad \left\{ \begin{array}{l} observe_Q(label(t_{i+1})) = true, \\ choose(T^C(I_i, S_i)) = t_{i+1}, \\ send_Q(acknowledge_j) = true \end{array} \right. \qquad \text{for } \langle h_i^{n_j}, I_i, S_i, O_i\rangle
\end{array}
$$

定義 4.38 (OPT 耐故障実行モデル) **OPT 耐故障実行モデル**は、プライマリプロセスの遷移関数をT^{OC}、バックアッププロセスの遷移関数をT'^{OC}とする以外は、耐故障実行モデルに同じとする.

補題 4.19 プライマリプロセスの実行がすべて非決定的実行であれば、OPT 耐故障実行モデルと耐故障実行モデルは同一である.

証明）定義 4.35により、$\forall i\ h_i^{n_j} = h_i$ であるとき、定義 4.36より $T^{OC} = T^C$、定義 4.37より $T'^{OC} = T'^C$ 、よって定義 4.38より OPT 耐故障実行モデルは耐故障実行モデルと同一である.

補題 4.20 無故障時には、OPT 耐故障実行モデルと耐故障実行モデルは同じ性質を持つ.

証明） 補題 4.16と同様.

補題 4.21 OPT 耐故障実行モデルにおいて、バックアッププロセス Bが状態 $\langle {h'_i}^{n_j}, I_i, S_i, O_i\rangle$ または $\langle {h'_i}^{d_j}, I_i, S_i, O_i\rangle$ にあるならば、プライマリプロセス Pは $\langle {h_s}^{n_j-1}, I_s, S_s, O_s\rangle$ または $\langle {h_t}^{n_j}, I_t, S_t, O_t\rangle$ の状態にある.

証明）$h'_i = h_i^{n_j}$であれば、補題 4.17に同じ．$h'_i = h_i^{d_j}$のとき、定義 4.35より $\exists s\ (s < i)$ s.t. $h'_s = h'^{n_j}_s$ よって、$h_i^{n_j}$について補題 4.17と同様にして示される.

補題 4.22 OPT 耐故障実行モデルにおいて、プライマリプロセス Pの故障が検出された後、バックアッププロセス Bは引き継ぎによって、故障していないプライマリプロセスと同値となる.

証明）補題 4.18と同様に故障が検出される．バックアッププロセス Bの状態 $\langle h'^{x_j}_i, \ldots\rangle, (x = n \text{ or } d)$ に対して、補題 4.21によりプライマリプロセス Pの状態は $\langle h_t^{n_j}, \ldots\rangle$ であることが保証される．$h'^{x_j}_i$ が 非決定的 $(x = n)$ であれば、補題 4.18に同じ．$h'^{x_j}_i$ が 決定的 $(x = d)$ であれば、$P(1, j), B(1, j)$ まで部分同値であり、$P(j + 1, i), B(j + 1, i)$ は 決定的な実行であるから、補題 4.3より部分同値である.

定理 4.3 PS と BS の全てのプロセスが、OPT 耐故障実行モデルに従い、PS の故障時にプロセスの引き継ぎが行われると、PS と BS に耐故障性がある.

定理 4.2および補題 4.22より自明.

4.5 まとめ

本研究で提案する並列プログラムの耐故障化について、形式化しその正当性について示した．耐故障実行モデルは、プライマリプロセスとバックアッププロセスがメッセージ交換によって協調しながら実行を進めることで、出力をコミットする．しかしこのままではメッセージ交換の多さが効率を落すため最適化したモデルを示した． 最適化した耐故障実行モデルは、非決定的実行部分と決定的実行部分を別に扱うことでメッセージ交換を減らす．具体的に対応する言語については後の第 6章で、その言語を前提とした実現方式は第 7章で、プログラムの FTPS への変換方式については第 8章で述べる.

第 5 章

耐故障ソフトウェアの信頼性

本研究で提案する方式について、その信頼性について考察する．

本研究で提案する方式は非修理系の冗長系を構成する．現在の実装方式では、故障したプロセッサを含むサイトは遺棄され、その他の故障していないプロセッサは再利用されない．このようなシステムは、非再構成系と呼ばれ、故障が発生する度に信頼性を低下させる．これに対して、これらプロセッサを再利用することで、信頼性や処理能力を低下させないシステムを再構成系と呼ぶ．ここでは、現在の非再構成系に加え、本研究で提案する手法を再構成系に発展させた場合についても考察する．

なお計算式の導出の詳細については、付録 A に記す．

5.1 評価尺度

耐故障並列システムは、冗長性を与えることで信頼性を増すが、そのために実行プロセッサ数を減少させ処理性能を低下させる．従って、故障に対する単なる信頼性ではなく、処理性能やコストを加味した評価基準も必要である [17]．このような場合一般には、システム有効度とコスト有効度が用いられる [18]．システム有効度は、JIS 信頼性用語で「システムが規定の任務を達成できると期待できる良さの尺度．信頼度、アベイラビリティ、能力などの関数として表す」と定義される．この場合、関数の形としての定義はなく、能力、信頼度、アベイラビリティの積が用いられることが多い．コスト有効度は、システム有効度をライフサイクル全期にわたるコストで割った値であり、投資効果を表す．

ここで、アベイラビリティは修理系において定義されることから、本研究で提案する非

修理系での評価尺度としては適合しない．そこで、MTTF とシステムの処理能力の積をシステム有効度とする．

これは、「平均寿命あたりの処理容量」とみなすことができる．システムの処理能力は、与えられた仕事を並列実行するのに利用できる要素プロセッサ数とする．これは、要素プロセッサの処理能力を基準とした並列計算機の最大処理能力を示す． 本研究で提案するシステムでは、サイトを構成する要素プロセッサ数が、これに相当することになる．従って本研究で提案するシステムは、初期のシステム構成によって処理能力が定まる． コストは、システム全体のプロセッサ数とする．よって次のように定義される．

$$
\begin{aligned}
\text{（最大）システム処理能力} &= \text{与えられた仕事を並列実行するのに利用できる} \\
&\quad \text{最大要素プロセッサ数} \\
\text{コスト} &= \text{システム全体の要素プロセッサ数} \\
\text{最大システム有効度} &= \text{システム処理能力} \times MTTF \\
\text{最大コスト有効度} &= \text{最大システム有効度/コスト}
\end{aligned}
$$

ここで定義するシステム処理能力は、システムが並列プログラムを実行する際の最大能力を示すことから、対象とする並列プログラムに十分な並列性が存在しなければ、実際の有効度は減少する．従って動作させる並列ソフトウェアが実行時に並列プロセッサを占有する率を考慮して、次のように定義される．

$$
\begin{aligned}
\text{利用率 } U &= \text{プログラムが並列実行中に占有するシステムの割合} \\
&\quad (0 \leq U \leq 1) \\
\text{実効システム処理能力} &= \text{システム処理能力} \times U \\
\text{システム有効度} &= \text{実効システム処理能力} \times MTTF \\
\text{コスト有効度} &= \text{システム有効度/コスト}
\end{aligned}
$$

例えば、均質な処理能力の要素プロセッサ n 台で構成される並列計算機では、後で示すように MTTF は単体の要素プロセッサ の 1/n、システム処理能力、コストは n 倍である．よって、最大システム有効度は単体プロセッサと同じで、最大コスト有効度は 1/n となる．このことは、n 台でその MTTF の間にできる総仕事量は、1 台でその MTTF の間にできる総仕事量で限定されることを示す．すなわちコスト有効度を犠牲にして、並列効果によ

る高速な処理を得たものの、MTTF も短くなったため 1 台の最大計算容量としては変わっていない.

5.2 非再構成系としての信頼性

プライマリサイト内の要素プロセッサが故障した場合に、残りの無故障プロセッサを再構成しない場合について考察する.

通常の並列計算機は、構成要素の一つが故障しても実行に失敗することから、直列系のシステムである. 従って n プロセッサから成るシステムの信頼度 $R_{site}(n)$ は、要素プロセッサの信頼度 r_i $(i = 1, \ldots, n)$ に対して一般に

$$R_{site}(n) = \prod_{i=1}^{n} r_i$$

となる.

プライマリサイト/バックアップサイトから成るシステムはバックアップがホットスタンバイのシステムであり、切替え確率を 1 とみなせば並列系のシステムと考えることができる. よって不信頼度が各サイトの不信頼度の積となる. バックアップ k 個の k-Resilient システムでは、各サイトの不信頼度 (F_{site_i} $(i = 0, \ldots, k)$、$site_0$はプライマリサイト) に対して、次のように表される.

$$F_{system}(k) = \prod_{i=0}^{k} F_{site_i}$$

従って信頼度は、次で計算される.

$$R_{system}(k) \quad = \quad 1 - \prod_{i=0}^{k} (1 - R_{site_i})$$

ここで要素プロセッサの故障率を一定とし、信頼度が指数分布に従うとする. すなわち $r_i(t) = e^{-\lambda_i t}$ とすると次を得る.

$$\begin{aligned} R_{site}(n,t) &= e^{-\sum_{i=1}^{n} \lambda_i t} \\ MTTF_{site}(n) &= \int_0^{\infty} R_{site}(t)dt = \frac{1}{\sum_{i=1}^{n} \lambda_i} \end{aligned}$$

各プロセッサの信頼度が均質 ($\forall i,\ r_i(t) = e^{-\lambda t}$) であるとすれば、n プロセッサから成るサイトは次のように 1 プロセッサの 1/n の MTTF になる．

$$\begin{aligned} R_{site}(n,t) &= e^{-n\lambda t} \\ MTTF_{site}(n) &= \frac{1}{n\lambda} \end{aligned}$$

すべてのサイトが均質な信頼度 ($e^{-\lambda t}$) のプロセッサで、同じプロセッサ数 (n) から構成されるとき、システムの信頼度は次のようになる (付録 A.1参照)．

$$\begin{aligned} R_{system}(k,t) &= 1-(1-e^{-n\lambda t})^{k+1} = e^{-n\lambda t}\sum_{i=0}^{k}(1-e^{-n\lambda t})^i \\ MTTF_{system}(k) &= \int_0^\infty R_{system}(k,t)dt = \frac{1}{n\lambda}\sum_{i=0}^{k}\frac{1}{i+1} \end{aligned}$$

従って k-Resilient システムの MTTF は、1 サイトの MTTF の$\sum_{i=0}^{k}\frac{1}{i+1}$倍になる．一般に次の関係があることから、MTTF はサイト数 kに関する log オーダーで増加することが分かる．

$$k > 0,\ \log(k+2) < \sum_{i=0}^{k}\frac{1}{i+1} < 1+\log(k+1)$$

またシステムの処理能力はサイトを構成するプロセッサ数 n であり、サイト数 $k+1$ でコストは $(k+1)n$ となるので次を得る．

$$MTTF_{\text{非再構成}} = \frac{1}{n\lambda}\sum_{i=0}^{k}\frac{1}{i+1} \qquad \text{システム処理能力}_{\text{非再構成}} = Un$$

$$\text{システム有効度}_{\text{非再構成}} = \frac{U}{\lambda}\sum_{i=0}^{k}\frac{1}{i+1} \qquad \text{コスト有効度}_{\text{非再構成}} = \frac{U}{n(k+1)\lambda}\sum_{i=0}^{k}\frac{1}{i+1}$$

サイトを構成するプロセッサ数が n で $k+1$ サイトから成るシステムであるため、システム全体のプロセッサ数は $(k+1)n$ である．よって、この同じシステムを耐故障を考慮しない通常の並列計算機として見ると、$n(k+1)$ 台構成の並列計算機であるから次の値を得る．

$$MTTF_{\text{非耐故障}} = \frac{1}{n(k+1)\lambda} \qquad \text{システム処理能力}_{\text{非耐故障}} = Un(k+1)$$

$$\text{システム有効度}_{\text{非耐故障}} = \frac{U}{\lambda} \qquad \text{コスト有効度}_{\text{非耐故障}} = \frac{U}{n(k+1)\lambda}$$

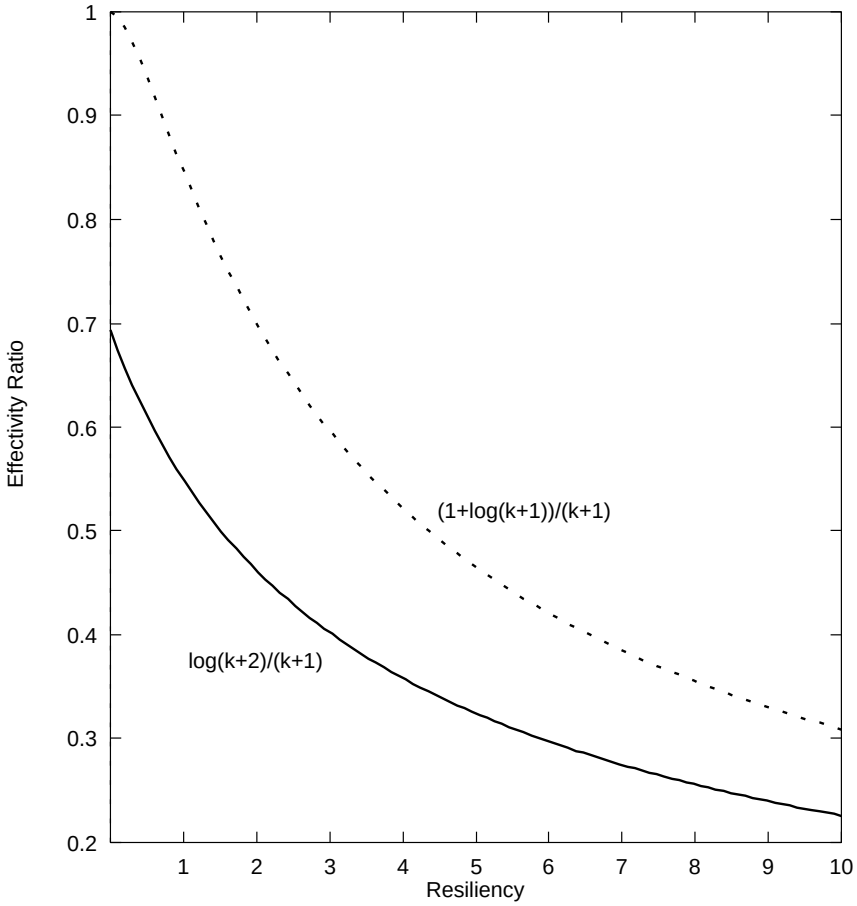

図 5.1: 非再構成系の有効度増加率 対 処理能力減少率

従って耐故障ソフトウェアとすることで、システム有効度、コスト有効度とも $\sum_{i=0}^{k} \frac{1}{i+1}$ 倍となる．ただし、システム処理能力は $\frac{1}{k+1}$倍である．処理能力減少率 $k+1$ に対する有効度増加率 $\sum_{i=0}^{k} \frac{1}{i+1}$は、有効度増加率が $\log(k+2) < \sum_{i=0}^{k} \frac{1}{i+1} < 1+\log(k+1)$ のようにlogで押えられることから、図 5.1のようなグラフの間の値をとる．kに対する有効度の増加に比較して、システム処理能力の低下は急激であるので、並列計算機の処理能力の高さを保持したまま有効度を高くしたいと考えれば、kはあまり大きくできない．

5.3 再構成系としての信頼性

プライマリサイト内の要素プロセッサが故障した場合に、残りの無故障プロセッサを再構成する場合について考察する．再構成の方法には、無故障プロセッサの構成の仕方により次の２つが考えられる．

1. 性能重視型再構成：動作中のプライマリサイト、バックアップサイトに組み入れる．
2. 信頼性重視型再構成：一つのバックアップサイトとして構成し直す．

前者は実装が容易であり実現可能性は大きい．後者は、実行途中のプログラム状態を比較的高速に再現する方法がない限り、実現可能性は低い．ここでは前者の場合についてのみ信頼性について考察する．

5.3.1 性能重視型再構成

ここでは、一回の故障によって一つのプロセッサが利用不能になるとし、瞬時に再構成し実行継続すると仮定する．従ってk-Resilient システム（n プロセッサから成る $k+1$ サイトの構成）は、一回の故障発生によって $(k-1)$-Resilient システム（$n+\frac{n-1}{k}$プロセッサから成る kサイト構成）へとカバレッジ（切替え成功確率）1 で瞬時に切り替わるとする．すなわち、サイトを構成するプロセッサ数 n を kの関数 $N(k)$ とした非再構成系とみなせる．このとき、システムの総プロセッサ数は、$(k+1)N(k)$ で表せる．

$$N(k) = \frac{1}{k+1}\{n(K+1)-K+k\} \quad \left(\begin{array}{ll} N(k) = N(k+1) + \frac{N(k+1)-1}{k+1} & (0 \le k < K) \\ N(K) = n & (\text{初期状態}) \end{array} \right)$$

ここで、Kは初期状態の Resiliency とする．

このシステムは kが故障が一つ発生するたびに減少してゆくため、図 5.2のように k-Resilient システムで正常動作している状態を S_kとしたときのマルコフモデルでモデル化される．このとき吸収状態は S_{-1}であり、システムの障害状態を示す．

信頼度は時刻t で S_iにある確率 $P(S_i,t)$ によって、次のように表される (付録 A.2.1参照)．

$$R_{system}(K,t) = \sum_{i=0}^{K} P(S_i,t)$$

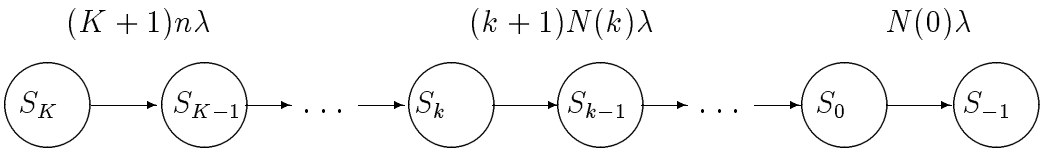

図 5.2: 性能重視型再構成におけるマルコフモデル

$$\begin{cases} P(S_k,t) = \left(\prod_{i=k+1}^{K}\lambda_i\right)\left(\prod_{i=1}^{K-k}(-\frac{1}{i\lambda})\right)\sum_{i=0}^{K-k}(-1)^i{}_{K-k}C_i e^{-\lambda_{K-i}t} & \forall k\ (0\le k<K) \\ P(S_K,t) = e^{-\lambda_K t} \end{cases}$$

よって MTTF は以下のようになる (付録 A.2.2参照).

$$\begin{aligned} MTTF_{system}(K) &= \int_0^\infty R_{system}(K,t)dt \\ &= \frac{1}{\lambda_K} + \sum_{k=0}^{K-1}\left(\left(\prod_{i=k+1}^{K}\lambda_i\right)\frac{1}{(K-k)!}\left(-\frac{1}{\lambda}\right)^{K-k}\sum_{i=0}^{k}(-1)^i{}_{K-k}C_i\frac{1}{\lambda_{K-i}}\right) \end{aligned}$$

この式によって、1-Resiliency、2-Resiliency で始めたシステムについて算出すると以下のようになる (付録 A.2.3参照).

$$\begin{aligned} MTTF_{system}(1) &= \int_0^\infty P(S_1,t)dt + \int_0^\infty P(S_0,t)dt = \frac{1}{2n\lambda} + \frac{1}{(2n-1)\lambda} \\ MTTF_{system}(2) &= \int_0^\infty P(S_2,t)dt + \int_0^\infty P(S_1,t)dt + \int_0^\infty P(S_0,t)dt \\ &= \frac{1}{3n\lambda} + \frac{1}{(3n-1)\lambda} + \frac{1}{(3n-2)\lambda} \end{aligned}$$

ここで注意しなければならないのは、このシステムでのシステム有効度は、システムの処理能力が一定でないために、第 5.1節で定義したものとは異り、次で定義される．すなわち、

$$\begin{aligned} MTTF_{\text{システム}} &= \int_0^\infty \sum_i P(S_i,t)dt \quad \text{であるとき、} \\ \text{システム有効度} &= U\int_0^\infty \sum_i P(S_i,t)N(i)dt \quad \text{と定義される.} \end{aligned}$$

従って、初期状態が 1-Resilietn システムのシステムは、n プロセッサづつ 2 サイトの構成から開始するため、総プロセッサ数が $2n$ であることから次を得る．

$$\begin{aligned}
\text{システム有効度}_{\text{再構成}} &= U\left(\frac{1}{2n\lambda}\times n+\frac{1}{(2n-1)\lambda}\times(2n-1)\right)=U\left(\frac{1}{2\lambda}+\frac{1}{\lambda}\right)=1.5U\frac{1}{\lambda}\\
\text{コスト有効度}_{\text{再構成}} &= 1.5U\frac{1}{\lambda}\times\frac{1}{2n}=1.5U\frac{1}{2n\lambda}
\end{aligned}$$

従ってシステム有効度、コスト有効度とも耐故障しない元のシステムの 1.5 倍となる．以下、簡単のため $U=1$ とする．

初期状態が 2-Resilietn システムのシステムは、n プロセッサづつ 3 サイトの構成から開始するため、総プロセッサ数が $3n$ であり、以下を得る．

$$\begin{aligned}
\text{システム有効度}_{\text{再構成}} &= \frac{1}{3n\lambda}\times n+\frac{1}{(3n-1)\lambda}\times\frac{3n-1}{2}+\frac{1}{(3n-2)\lambda}\times(3n-2)\\
&= \frac{1}{3\lambda}+\frac{1}{2\lambda}+\frac{1}{\lambda}=\frac{11}{6\lambda}=1.83\frac{1}{\lambda}\\
\text{コスト有効度}_{\text{再構成}} &= \frac{11}{6\lambda}\times\frac{1}{3n}=1.83\frac{1}{3n\lambda}
\end{aligned}$$

よって、1.83 倍のシステム有効度が得られる．

以上まとめると表 5.1のようになる．

表 5.1: 耐故障化における有効度の比較

システム (Resiliency)	MTTF	システム有効度	コスト有効度	性能
単体 PE(0)	$1/\lambda$	1	1	1
非耐故障 (0)	$1/(N\lambda)$	1	$1/N$	N
非再構成 (K)	$\frac{K+1}{N\lambda}\sum_{i=0}^{K}\frac{1}{i+1}$	$\sum_{i=0}^{K}\frac{1}{i+1}$	$\frac{1}{N}\sum_{i=0}^{K}\frac{1}{i+1}$	$N/(K+1)$
再構成 (1)	$\frac{1}{N\lambda}+\frac{1}{(N-1)\lambda}$	1.5	$1.5\frac{1}{N}$	$\frac{N}{2}$〜$(N-1)$
再構成 (2)	$\frac{1}{N\lambda}+\frac{1}{(N-1)\lambda}+\frac{1}{(N-2)\lambda}$	1.83	$1.83\frac{1}{N}$	$\frac{N}{3}$〜$(N-2)$

(システムの総プロセッサ数を N、利用率 $U=1$ とする．有効度と性能は単体 PE を基準とした値を示す．)

5.4 信頼性に関するまとめ

本研究で提案する耐故障方式について、その信頼性と有効度について述べた．なんら耐故障を行わない並列システムは、処理性能を単一プロセッサの台数倍にするかわりにMTTFも台数分の1にしてしまう．このため、故障にあわずに実行可能な仕事量は、単一プロセッサと変わらないことを示した．

例えば、MTTFが1万時間の単一プロセッサ千台から構成される並列システムを考えると、この並列システムはMTTFが10時間となる．このことは、この並列システムを最も有効に使えば、単一プロセッサが1万時間に成し遂げる仕事量を10時間で得られることを示す．しかし、この並列システムで10時間を越えるような仕事の成功は期待できない．この並列システムを非再構成の1-Resilientシステムに組み変えるならば、元の並列システムに比べて性能は1/2に落ちるため、仕事が20時間以上かかり得る．ところが、MTTFが30時間になるため十分に成功し得ることが分かる．

本章では、本研究で提案するシステムが、既存並列システムの冗長性を生かしてMTTFをlogオーダーで伸ばすことができることを示した．しかし性能を犠牲にするため、並列システム本来の高速な実行は損なわれる．また、現在の方式では故障したサイトの生き残ったプロセッサを再構成していないため、単純に生存しているサイトに組み入れる再構成方式について有効度を算出してみたが、その有意性は見られなかった．別の新たなサイトを構成する方式については、その実現は困難であることからここでは検討しなかった．

新たなサイトを構成する場合の困難さは、動作途中のプログラム状態の再現にある．現在提案しているシステムでは、バックアップ側も同時進行でプログラムを実行しているため切替えを即時とした． しかし、新サイトを構成して同じプログラム状態を再現するには、最初から実行をやり直すしか今のところ方法がない．従って信頼性解析をする際には、プログラムを最初から再実行する時間も有効性として加味する必要がある．

第 6 章

対象言語とその特徴

本章では、本論文で提案している耐故障並列ソフトウェアを実現する対象として選択した並列論理型言語について、その特徴を述べるとともに、NSM アプローチの一実現例であるデバッギング方式について紹介する．並列論理型言語は、並列言語として十分な記述力と機能を備えている上、自動変換が容易であり、本研究で提案するアプローチを実証する上で適していると判断する．

6.1 並列論理型言語 KL1

本研究で用いる並列論理型言語 KL1[7] は、専用の並列推論マシン向けに新世代コンピュータ技術開発機構（ICOT）で開発された並列言語であり、いくつかの応用分野で利用された実績を持つ．中でも定理証明の分野では未解決問題を解くという成果をあげているが、そのためには 16 台〜64 台の PE を使って、約 10 時間の連続運転を必要としている [8]．その後 ICOT で開発された KL1 コンパイラ KLIC [10, 11] は、C 言語をターゲット言語とし、プロセッサ間通信に PVM をターゲットの一つとして選択できるように設計したことで移植性を高めた．ターゲットを C 言語としたことで、C コンパイラの最適化を利用して高い性能を得ている．最近の超並列計算機のほとんどが C コンパイラと PVM を用意しており、KLIC によってこれら並列計算機や分散環境で KL1 が利用可能になりつつある．

並列論理型言語 KL1 は、FGHC（Flat Guarded Horn Clauses）[9] を元に設計された．プログラムは述語定義の集合から成り、個々の述語はクローズの集合から成る．クローズのシンタクスは次のようになる．

$$\underbrace{H}_{\text{ヘッド}} \quad :- \quad \underbrace{G_1, G_2, \ldots, G_m}_{\text{ガード}} \mid \underbrace{B_1, B_2, \ldots, B_n}_{\text{ボディ}} \quad (m, n \geq 0)$$

ヘッドは、述語ヘッドであり述語名と引数によって識別される．ヘッドとガードは条件部に相当し、ガード部分には単純な組み込み述語のみ許される．ヘッド及びガードにおけるユニフィケーションは、呼出し側変数を変化させない．条件部が満たされると コミット（|）に到達するが、条件部を満たすクローズが複数ある場合には、そのうちのただ一つだけがコミットできる．コミットしたクローズのボディのゴールは、それぞれ並列に動作する．ボディでのユニフィケーションは、呼出し側変数に値を返すことができる．

6.1.1 並列実行

KL1 の各ゴールは論理的に並行に動作できる．実際の並列環境では、明にゴールをプロセッサへ分散させることで並列な実行が可能である．ゴールの別プロセッサへの分散は、プログラム中で`node(N)` プラグマを付加することで指定される．`N` は論理プロセッサ番号であり、プログラムの実行時に動的に与えることもできる．

```
f([N|X]) :- integer(N) | g(N) @ node(N), f(X).
```

6.1.2 優先度

KL1 では次の 2 種類の優先度制御が許される．

1. ゴール間優先度

   プラグマ`lower_priority` を付加することによって、呼出しゴールの優先度よりも低優先度でゴールを動かすことができる．優先度は、同じプロセッサ上でのみ有効である．

   ```
   f([N|X])  :- g(N) @ lower_priority, f(X).
   ```

2. クローズ間優先度

クローズが排他的でない場合、クローズの順序によらずどれがコミットされるかは非決定的である．このとき、クローズの間にalternatively を置くことで、alternatively より上にあるクローズが優先的に試される．

```
f([N|X],Y) :- f1(N,X,Y).      % clause1
alternatively.
f(X,[N|Y]) :- f2(N,X,Y).      % clause2
```

なおalternatively は分散環境では、実装上特別な意味を持つ．

上の例で

```
?- f(X,Y)@node(0), gen(X)@node(1), gen(Y)@node(0).
```

の実行を考える．gen(X) は、PE1 上で値を生成しX を通じてf(X,Y) へ送るのに対して、gen(Y) は、f(X,Y) と同じ PE0 上で値を生成する．f(X,Y) の実行に際して、clause1 は別プロセッサで生成される値によってコミットすることになるため、PE1 から実際の値が送られて来ない限り高優先度であってもコミットし得ない．このような場合に対応するためalternatively は、PE1 へ値の要求メッセージを発送する．この要求によって PE1 からの値転送が促される．

6.1.3 マクロ展開機能

KLIC ではいくつかのマクロ展開機能が提供されており、高い記述力を備えている．

1. 引数対

   論理型言語では、差分リスト等の記述のため、対になった変数を用いることが多い．このようなことから、KLIC ではヘッド、ボディゴールに対して、簡単な記述で変数対を引数に加えることができる．これを引数対と呼ぶ．また、引数対に対する操作述語 (<=等) も各種提供されている．
   例えば、

   ```
   f([N|X])-Y :- Y <= N, g(N)-Y, f(X)-Y.
   ```

   のように記述することで、次のようなクローズが得られる．

```
f([N|X],Y0,Y) :- Y0 = [N|Y1], g(N,Y1,Y2), f(X,Y2,Y).
```

2. if-then

   Prolog と同様に、クローズのボディに (if P then Q) の構造の述語が記述できる. 例えば、

```
f([N|X]) :-
    ( N > 0  -> g(N,X) ;
      N < 0  -> h(N,X) ;
      otherwise ;
      true   -> f(X) ).
```

   のような記述は、マクロ展開によって次のようなクローズになる.

```
f([N|X]) :- f_1(N,X).

f_1(N,X) :- N > 0 | g(N,X).
f_1(N,X) :- N < 0 | h(N,X).
otherwise.
f_1(N,X) :- f(X).
```

   ここで述語名`f_1` は、コンパイラ内で生成される.

3. 関数評価

   算術演算を表す項`Expr` に対して、`~Expr` と記述することによって、これを評価するゴールへ展開する. 例えば、

```
f([N|X]) :- g @ node(~(N+1)), f(X).
```

   は、次のようにマクロ展開される.

```
f([N|X]) :- N1 := N+1, g @ node(N1), f(X).
```

6.2 並列論理プログラムのプロセスモデル解釈

KL1 による並列論理プログラムは、次のようにプロセスモデルとして解釈することができる.

ゴールは、並行もしくは並列に動作するプロセスである．ゴールの引数として保持される定数は、プロセスの保持する状態を示す．変数は、プロセス間の通信ストリームに相当する．

述語定義は、そのインスタンスであるゴールの状態遷移を定義する．クローズは、それぞれＯＲ関係にある状態遷移を示し、遷移条件に応じてそのうち唯一つだけが選択される．ヘッドとガードは遷移条件を示す．ここでのユニフィケーションは、プロセスへの入力に相当する．ボディは、遷移によって起こすアクションを表す．呼出し側変数へのボディユニフィケーションは、プロセスからの出力に相当する．ボディゴールは、遷移時に新たに生成するプロセスを表す．再帰呼出しのゴールは、遷移したプロセスの次の状態を示す．再帰呼出しの存在しないクローズは、状態遷移の終端に相当する．

遷移条件を判定するのに必要な値が、まだ入力されなければプロセスは条件判定を留保する．述語定義のクローズ全体で条件判定が留保されるとき、プロセスは状態遷移しない．この機構によって並列論理プログラムはプロセス間の自然な同期を実現している．

6.3　並列論理プログラムの非決定性

一般的に並列論理型言語は、効率の良い並列実行を可能にするためにプログラムに二つの非決定性を許している．一つはゴールのスケジューリングにおける非決定性であり、もう一つはクローズをコミットする際の非決定性である．これら非決定性は、プロセスの非決定的動作を導く．

1. スケジューリング非決定性

   6.2節の解釈に従えば、ゴールのスケジューリングはプロセスの状態遷移の順序に相当する．すべてのプロセスが同じ状態遷移をするならば、スケジュール順序が変わっても論理的な実行は同じものになり出力も同じ内容が得られる．ただし出力のタイミングは非決定的である．

2. コミット非決定性

   6.1節で述べたように、条件部を満たすクローズが複数存在する際には、どのクローズでもコミットし得る．このとき実際にどのクローズをコミットするかは、非決定的である．クローズのコミット条件は一般に入力ストリームとそこから得た内容によって定義され、非決定的な状態遷移は入力タイミングによって生じる．

プログラムにおけるこの種の非決定性の存在は、以下に示す静的なプログラム形式から判断できる.

6.3.1 決定的な述語定義

遷移条件がクローズ間で排他的であれば、述語定義の状態遷移は決定的である．例えば次の述語定義のように、すべてのクローズが一つの入力ストリーム $Arg1$ に対する条件だけで状態遷移が定義され、クローズ間で条件が排他的であるならば、明らかに決定的である.

$$f(Arg1, Arg2) \ :- \ Arg1 >= 0 \mid g1(Arg2).$$
$$f(Arg1, Arg2) \ :- \ Arg1 < 0 \mid g2(Arg2).$$

一つの入力ストリームは入力列が同じならば到着順序が保証されることから、入力タイミングが変わっても、決定的述語定義は同じ入力列に対して RSM アプローチの条件を満足する．よって、決定的述語定義に基づくプロセスは SM として実現される.

6.3.2 非決定的な述語定義

非決定的な述語定義については、入力列を同じにしても実行を再現することはできない．非決定的述語定義に基づくプロセスは NSM として実現するために、非決定的実行ログが必要である．述語定義が非決定的になるのは、次の二つの原因による.

1. 複数の入力ストリームに対する条件がある.

   例えば次の述語定義に対して $f(X, Y), X = a, Y = b$ を実行した場合、先に入力ストリーム Y に値が到着すれば、第二クローズでコミットする.

   $$f(Arg1, Arg2) \ :- \ Arg1 = a \mid g1(Arg2).$$
   $$f(Arg1, Arg2) \ :- \ Arg2 = b \mid g2(Arg1).$$

   この例では、二つの入力ストリーム全体の入力順序が実行ログとして必要である.

2. 一つの入力ストリームに対する条件が排他的でない.

   例えば次の述語定義に対して $f(X), X = 0$ を実行した場合を考える．実行時の第一クローズの条件判定と第二クローズの条件判定の間に入力ストリーム X に値が到着

したならば、第二クローズでコミットする．

$$f(Arg1, Arg2) \ :- \ \ Arg1 >= 0 \mid g1(Arg2).$$
$$f(Arg1, Arg2) \ :- \ \ Arg1 =< 0 \mid g2(Arg2).$$

この例では、入力ストリームへの到着タイミングが実行ログとして要求される．

上のような非決定的な述語に対しては、状態遷移の推移そのものを実行ログとすることで、入力タイミングが記録できる．この状態遷移のログ量を低くすることが、実現上の問題となる．

6.4 トレース手法 – Instant Replay –

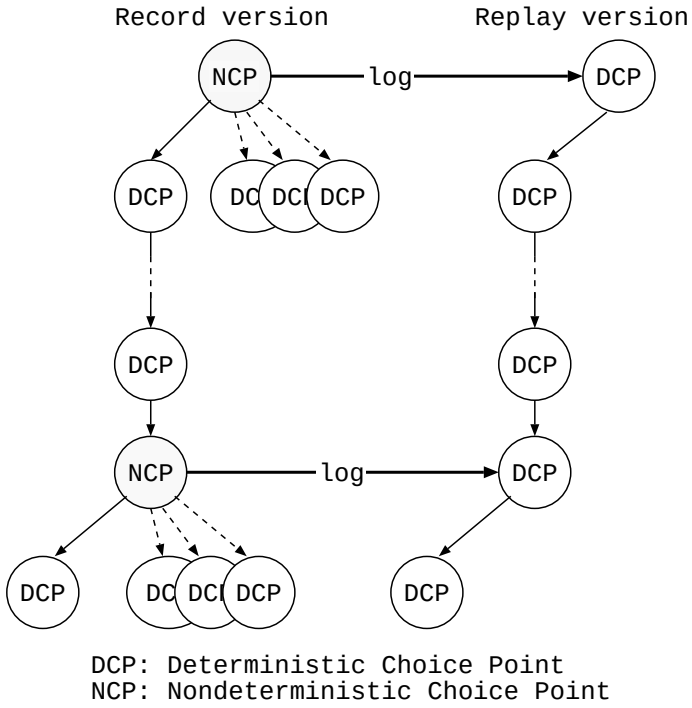

図 6.1: Instant Replay 実行の概念

並列論理型言語の非決定性は、プログラムの再現を必要とするデバッギングにおいても障害となった．しかし 6.3節で述べたように、クローズの非決定的選択が同じであれば、ゴールのスケジュール順序が変わってもプログラムの実行結果は同じになる．そこで非決定的

選択についてのみログをとれば、意味的に等しい再実行が可能になる．このように、並列プログラムのデバッギングスキームとして提案されたInstant Replay [12] を並列論理型言語へ適用したのがShen と Gregory が提案した並列論理型言語のIntant Replay スキームである [13]．Shen らの研究によれば、ほとんどの述語定義は決定的であり、静的解析によって非決定的選択のログ量は大幅に削減できることが示されている．

Instant Replay は、オリジナルの並列論理プログラムに対してプログラム変換によって二つのバージョンのプログラムを生成する．一方がログを生成しながら動作する レコード・バージョン（RecV）であり、もう一方がログを参照しながら動作を再現する リプレイ・バージョン（RepV）である．RepV は、RecV の非決定的選択のログを用いて実行するため決定的なプログラムとなる（図 6.1）．通常はRecV のプログラムを実行し、実行過程で生じるログを確保する．実行過程のログは非決定的実行とそれへ至る分岐のみを含む．もしこの実行でバグを発見したならば、その時に得たログを用いてRepV のプログラムを実行する．これによって、RecV と同じ非決定的実行をたどることができ、同じバグに至る実行が再現される．

このInstant Replay 方式は、第3章で述べたNSM アプローチの実現方式の一つと見ることができる．

6.5 KL1 と耐故障並列ソフトウェア

以上で述べたように、KL1 は高い記述力と機能を備え、高並列なプログラムの実現に適している．Instant Replay 手法に見られるように、オリジナルのプログラムを変換するのも容易に行うことができる．よって本研究で提案する耐故障並列ソフトウェアの実現に、KL1 の利用が適している．しかし、KL1 のような並列論理型言語でなくても、例えばCSP モデルに従うOccam のような言語であっても、本研究で提案するアプローチは適用可能である．

第 7 章

耐故障並列ソフトウェア実現方式

ここでは、前章で紹介した並列論理型言語 KL1 を対象とした場合の耐故障並列ソフトウェアの実現方式について述べる．NSM アプローチの実現方式の一つである Instant Replay のアイデアを参考に、PS アプローチをとることで耐故障並列ソフトウェアを実現する．動的な構成と静的な構成に分けて構成方法の提案を行うとともに、故障検出の実現方法についても述べる．

7.1 耐故障並列ソフトウェアの動的構成

第 3章で述べたように、並列動作する細粒度プロセスに対応するために複数 PE 上の複数プロセスでサイトを構成する（図 7.1）．PE は、$K+1$ 個のグループに分けられ、K-resilient システムを構成する．グループのうち一つが Primary Site Group （PSG）となり、他が Backup Site Group (BSG) となる．耐故障実行をするプロセスは、複製されて PSG、BSG のそれぞれへ送られる．PSG では RecV、BSG では RepV を元にしたプログラムを用いてプロセスを実行する．RecV の生成するログをネットワークを介して RepV に渡すことにより BSG は PSG と同じ実行を再現する．Instant Replay の RecV、RepV とは、第 4章で述べた出力コミットアルゴリズムを含む点異るため、区別するためにそれぞれ FTRecV、FTRepV と呼ぶ．

簡単化のため、以下では 2 個のグループからなる 1-resilient システムについて考える．

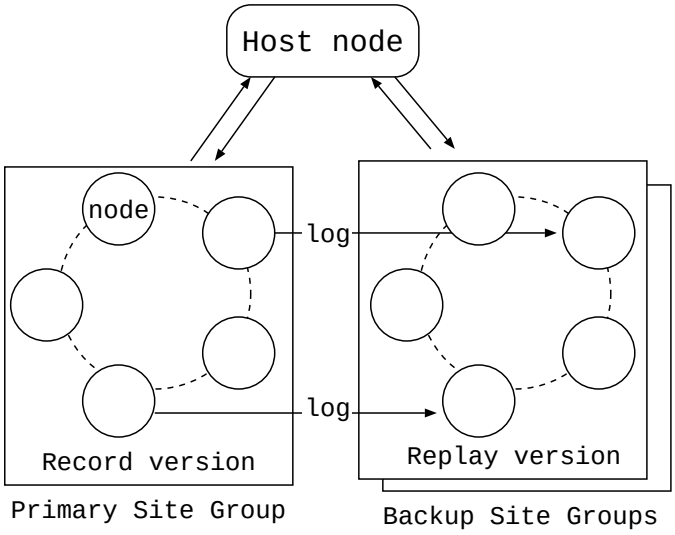

図 7.1: 耐故障並列ソフトウェアの動的構成

7.2 耐故障並列ソフトウェアの静的構成

オリジナルのユーザプログラムは、第 8章に示される変換方式に従って耐故障プログラムに変換される．このプログラムは、ホストノード上で通常実行される部分と各サイトグループ上で耐故障並列実行される部分から構成され、ホストノード上で動作する部分はクライアント、サイトグループ上で動作する部分はサーバとして機能する．この変換されたプログラムと、ライブラリとして用意されたプログラムをリンクして実行形式が得られる．以下では、このライブラリプログラムについて述べる．

ライブラリプログラムは、次から成る．

- 複製プロセスプログラム

  サーバプログラムを複製し、サーバ–クライアント間のコミュニケーションを仲介する．

- 監視プロセスプログラム

  故障検出とサイト内のゴール転送を行う．

7.3 複製プロセスプログラム

耐故障実行をさせるプロセスは、ホスト PE で複製を作る必要がある．このとき問題となるのは、プロセスの持つ変数である．6.2節で述べたように、変数は他のプロセスとの通信ストリームとして使われる．耐故障実行させるプロセスの変数は複製されて、ホスト PE と PSG、BSG とのそれぞれの通信用ストリームになる．並列論理型言語では同一変数への多重書き込みは禁止されるため、同じ変数をそのまま複製として PSG と BSG へ渡すわけにはいかない．そこで、各変数に対してプロセスを生成し、PSG、BSG へ渡した変数と元の変数の間の中継を行う．変数が入出力いずれかに使われるかによって、中継プロセスの性質は異るが、KL1 のプログラミングスタイルから、一つの変数を入出力両方に用いることは無いと仮定してよい．入出力いずれに使われるかを判定するには、静的なモード解析による方法 [15] も考えられるが、ここでは動的な方法をとった．すなわち中継プロセスは、入力または出力のいずれかが送られるのを待って判断する．

風雅では故障時の PS から BS への引き継ぎの際に、ホスト PE と PS、BS との通信において宛先の変化、メッセージの消失、冗長メッセージの発生が起こり得るとして、これに備えた通信機構を開発している [2]．本システムでは、あらかじめすべての BSG への宛先はストリームとしてホスト PE に保持されている．またホストから PS へ送信するメッセージは、すべて BS へも複製して送信する．これにより送信の切替えとメッセージの消失については考慮しない．PSG、BSG から送られるメッセージが同じであることを保証し、先に到着したものを用いることで故障発生ごとの切替えを行なわない．

7.4 監視プロセスプログラム

7.4.1 故障検出

PE の停止故障検出は、グループ内で一定期間ごとにシグナルを送ることで検出する．図7.2のように、グループ内の各 PE には、監視プロセス（Watchdog）を置く．監視プロセスは最高優先度で動作し、他のプロセスの実行によって処理がブロックされない．

PSG の一つの監視プロセスが、一定期間ごとにグループ内にトークンシグナルを送る．これが返ってきたことを確認して、BSG に alive シグナルを送る．返ってこなければ、タイムアウトして PSG 内で故障が発生したものとみなし BSG に fault シグナルを知らせる．

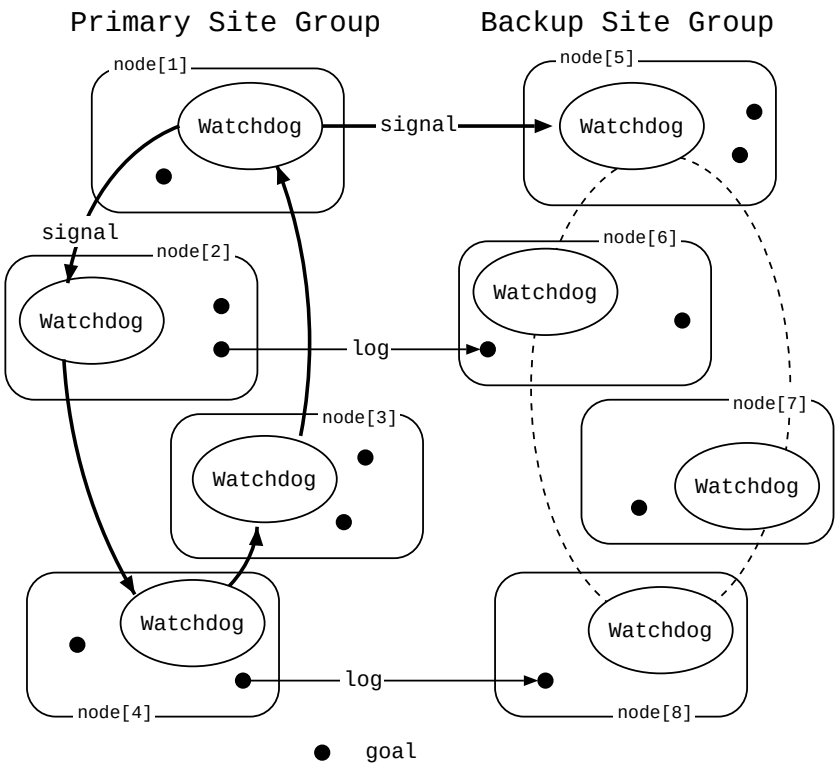

図 7.2: 監視プロセスによる故障検出

またトークンを発生させる監視プロセスの PE が故障した場合には、BSG は alive シグナルが来ないことで故障を検出する.

7.4.2 故障マスク

BSG が故障した場合は、PSG の実行には何も影響しない. PSG が故障したときは、故障検出した BSG の監視プロセスが他の監視プロセスにそれを知らせる. 知らせを受けた各監視プロセスは、自身の管理するプロセスにシグナルを送る. 監視プロセスからシグナルを受けとったプロセスは、FTRepV 実行をし終えた後 FTRecV 実行に移り変わる.

なおプロセスは、BSG 内のメッセージ交換によって自由に生成される. 実行を切替える際に転送中であったメッセージについても、到着後に FTRecV 実行させる.

7.4.3 PE 内ゴール管理と負荷分散

監視プロセスは、PE あたり一つあれば十分である．またプロセスが無い PE に監視プロセスを置いた場合、その PE の故障がサイト故障とみなされてしまう．PE 上で実行されているプロセス数を管理するために、監視プロセスは、PE 内のプロセスの終了判定を行う．また、他 PE から送られてくるプロセス起動メッセージも管理する．PE にまたがって分散している監視プロセスどうしはリング状に結合されており、このリングネットワークを用いて PE 間のプロセス起動メッセージの転送を行う．

監視プロセスの下で実行されるプロセスは、ショートサーキット法を利用して終了判定する．すなわち各プロセスは端点となる 2 変数を持ち、プロセスの終了時にこれら変数をユニファイ（ショート）する．すべてのプロセスの端点がショートすることで、サーキットが閉じプロセス全ての終了が判定できる．プロセスが新たなプロセスを生成する際には、そのプロセスもこのサーキットの一部となる．監視プロセスは自身の管理しているプロセスが全て終了したら、それ自身もショートサーキットと同様にしてリングネットワークから外れて消える．このとき、サイトのグループ管理をしている監視プロセスに消滅したことを知らせる．グループ管理者となっている監視プロセスは、プロセスを転送する際に空き PE があれば、そこに新たな監視プロセスとともにプロセスを転送する．

7.5 実現方式に関する考察

本章では、KL1 を対象にして耐故障並列ソフトウェアの実現方式について述べた．KL1 のゴール転送機能やショートサーキット法を利用することで、容易に実現可能であることを示した。KL1 以外の並列プログラミング言語を用いる場合には、同様の機能を実現する上で別の手段を工夫する必要がある．次の第 8章で述べる変換方式に関しても同様である．

第 8 章

耐故障並列ソフトウェア変換方式

ソフトウェアの耐故障化は、プログラマにとって大きな負担である．また複雑なプログラミングを必要とすることから、バグを含む可能性も高くなる．本研究では、ユーザプログラムを自動変換によって耐故障化することを目標としている． 自動変換に適していることから、本研究では並列論理型言語 KL1 を対象プログラムとして採用した．第 6章で述べたように、Shen らの Instant Replay では並列論理プログラムを自動変換することで、デバッギング可能な並列プログラムを生成している．

以下本章では、KL1 プログラムを耐故障化変換する手続きについて述べる．変換においては、プログラムの非決定的部分の抽出と、実行ログの生成が主要な処理である．最後に、変換によるプログラム増加量についても述べる．

8.1 耐故障ソフトウェアの構造

ユーザプログラムには、

1. 耐故障実行させる述語、
2. その述語内で負荷分散させるゴール

が指定されているものとする．耐故障実行させる述語は、プラグマ `fault_tolerant` が付加される．負荷分散させるゴールは、プラグマ`node` が付加される．負荷分散プラグマは、KL1 オリジナルのプラグマ `node(N)`（`N` は論理ノードアドレス）でも良いが、耐故障実行させる際に実際に転送されるノードは、指定のものとは異る．

変換の概要は、図8.1のようになる．すなわちユーザ指定の述語 method に対して、耐故障ソフトウェアは次のような操作に置き換わる．

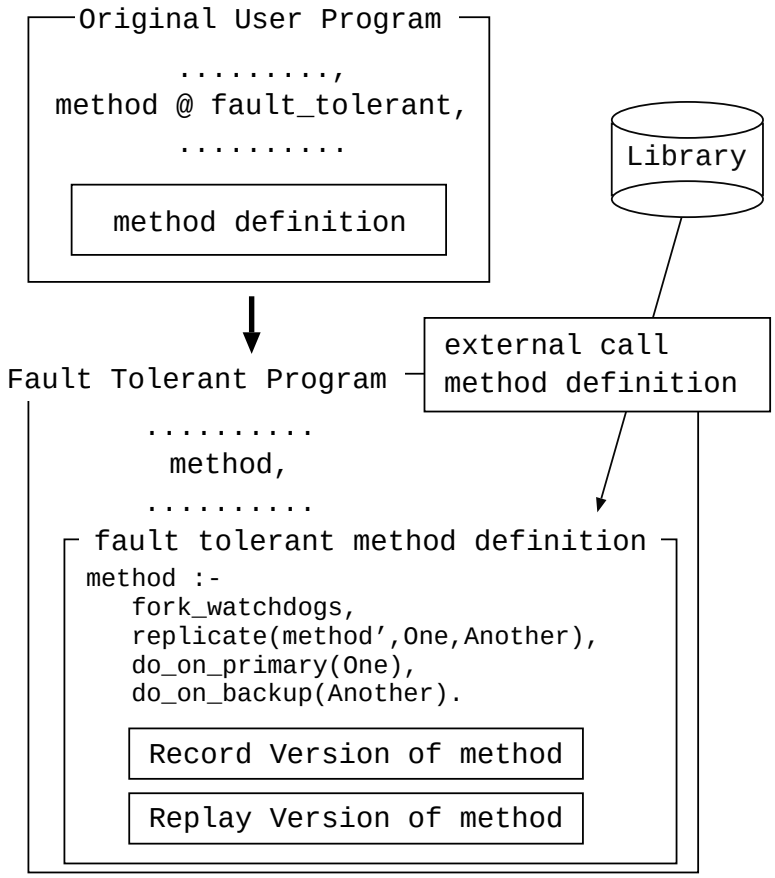

図 8.1: Converting original program

1. `fork_watchdog`: 監視プロセスのネットワークを構成させる．
2. `replicate`: ゴールの複製を作る．
3. `do_on_primary`: Primary Site Group でのゴール実行を指定する．ゴールは、Record Version で実行される．
4. `do_on_backup`: Backup Site Group でのゴール実行を指定する．ゴールは、Replay Version で実行される．

また、ライブラリからユーザプログラムを呼び出す必要から、外部呼出し述語を定義する.

変換後のプログラムは、ライブラリとして用意してある監視プロセス (watchdog)、複製プロセス (replicate) とともにリンクして、実行形式が得られる.

8.1.1 FTRecV/FTRepV プログラムの構造

KL1 プログラムにおけるプロセスの処理は、通例次のような列になる.

1. 入力処理 : 入力値の待ち合わせを含む.
2. 条件分岐 : 成立条件に応じて、いずれかの部分処理に分岐する.
3. 部分処理 : 出力、別プロセスの生成を含む.
4. 繰り返し : 1へ戻る. 並列論理プログラムでは、再帰呼出しに相当する.

   例) 入力[In|In1] を待ち、In の値を条件として、いずれかの部分処理 (クローズ) に分岐する. 各クローズのボディのユニフィケーションは、引数Out への出力となっている.

```
% 非決定的述語
nondet([In|In1],Out) :- In =< 0 |
       Out = [-1|Out1], nondet(In1,Out1).
nondet([In|In1],Out) :- In >= 0 |
       Out = [1|Out1], nondet(In1,Out1).
% 決定的述語
det([In|In1],Out) :- In < 0  |
       Out = [-1|Out1], det(In1,Out1).
det([In|In1],Out) :- In >= 0 |
       Out = [1|Out1], det(In1,Out1).
```

これに対して、FTRecV、FTRepV のプログラムは、次のような構造に変換される.

FTRecV :

1. 入力処理 : 非決定的実行に対する、返信の待ち合わせが追加される.
2. 条件分岐

3. 部分処理 : 条件分岐が非決定的であれば、分岐情報をログとして対応するプロセスへ送る.
4. 繰り返し

   例）ヘッドack(Log) で、返信の待ち合わせを行い、ボディのLog へのユニフィケーションがログの送信にあたる. プロセス内の出力（ユニフィケーション）は、ログに対する返信が返った後でのみ実行される. 出力を含む決定的述語は、最初に先のログに対する返信の待ち合わせをする以外は、オリジナルの述語に同じ.

```
nondet_record([In|In1],Out,ack(Log)) :- In =< 0 |
    Log=c1(Log1),
    (wait(Log1) -> Out = [-1|Out1]),
    nondet_record(In1,Out1,Log1).
nondet_record([In|In1],Out,ack(Log)) :- In >= 0 |
    Log=c2(Log1),
    (wait(Log1) -> Out = [1|Out1]),
    nondet_record(In1,Out1,Log1).

det_record(In,Out,Log) :- wait(Log) | det(In,Out).
```

FTRepV :

1. 割り込み処理 : 監視プロセスからの割り込みがあれば、FTRecV 実行に切り替わる.
2. 入力処理 : 非決定的実行に対する、ログの待ち合わせが追加される.
3. 条件分岐
4. 部分処理 : ログに対する返信を送る.
5. 繰り返し

   例）ログが送られてくる間は、処理本体nondet_replay_1 を呼び出す. そうでなければ割り込みチェックを行い、割り込みがあればFTRecV 実行へ切り替わる. 処理本体では、ガードでログの待ち合わせが含まれる. 決定的述語については、オリジナルと同じである.

```
nondet_replay(In,Out,Log,Control) :-
   (wait(Log) -> nondet_replay_1(In,Out,Log,Control) ;
    alternatively;
    '割り込みチェック'(Control) -> nondet_record(In,Out,NewLog)).

nondet_replay_1([In|In1],Out,Log,Control) :-
   Log = c1(Ack), In =< 0 | Ack = ack(Log1),
      Out = [-1|Out1], nondet_replay(In1,Out1,Log1,Control).
nondet_replay_1([In|In1],Out,Log,Control) :-
   Log = c2(Ack), In >= 0 | Ack = ack(Log1),
      Out = [1|Out1], nondet_replay(In1,Out1,Log1,Control).

det_replay(In,Out) :- det(In,Out).
```

8.2 変換手続き

変換手続きは、次の手順で行われる.

1. マクロ表記を展開する.
2. ホスト PE 上で実行されるプログラムと、耐故障実行されるプログラムとを分離する.

   トップレベル述語（main:main）から、トラバースしてプラグマfault_tolerant の付加されたゴールを探索する.
3. サーバ・プログラムを Instant Replay に基づいて、RecV と RepV のプログラムに変換する.

   この処理は、Instant Replay の変換ソフトウェアを用いる.
4. RecV、RepV のプログラムを耐故障ソフトウェア用の FTRecV、FTRepV に変換する. 各バージョンのプログラムに対して、次のような処理が行われる.

   すべての述語定義に対して、次の処理が行われる.

   (a) 制御用変数の追加：すべての述語定義に対して、終了判定に用いるショートサーキット用の変数が追加される. その他必要に応じて、プロセスへの割り込み、プロセスからの割り出し用のストリーム、終了判定用の変数が追加される.

(b) 負荷分散処理：プロセスの他PE転送処理を、監視プロセスへの割り出しへ変更する．

さらに個々の述語定義については、以下の処理が行われる．

RecV 非決定的述語：非決定的述語については、ログの返信待ちの追加、ログの転送（実際にはInstant Replay処理で追加済み）を追加し、さらに出力処理についてはログの返信待ちを行う．

RecV 決定的述語：決定的述語については、ログの返信待ちが先頭に追加される．

RepV 非決定的述語：述語定義の直前に、割り込み処理用クローズを一つ挿入する．これは監視プロセスから割り込んで、FTRecV実行に切り替えるための処理である．述語定義の本体では、各クローズにログ待ちの処理が追加される．

RepV 決定的述語：

決定的述語については無変換でかまわない．

5. クライアント・プログラムにおけるサーバ呼出し部分を、次のような処理呼出しに変換する．

   (a) 監視プロセスをPSG、BSG上で生成する．

   (b) サーバプロセスを複製する．

   (c) PSG, BSG上に生成された監視プロセスへサーバプロセスを送信する．

   プロセスの複製処理および監視プロセスの生成処理は、耐故障ライブラリとして別に提供される．

6. ホストPE用のクライアント・プログラムと、PSG用、BSG用それぞれの耐故障ソフトウェアを結合する．このとき負荷分散させるゴールに対して、監視プロセスからの呼出し用のクローズを一つ追加する．ユーザプロセスが分散させるゴールは、監視プロセスへ割り出した後このクローズを介して実行される．

8.3 変換例

例として、N-Queen 問題を解くプログラムを以下に示す.

```
% メインモジュール（オリジナル）
:- module main.

main :-  N = 8,
   klicio:klicio([stdout(normal(Out))]),
   util:times(UserTime),   go(N,UserTime,Out).

go(N,UserTime,Out) :-
   count(Result,0,UserTime,Out),
   queen(N,Result) @ lower_priority.

queen(N,Result) :- queen:queen(N,Result) @ fault_tolerant.

count([A|X],K,UT,Out) :- count(X,~(K+1),UT,Out).
count([]   ,K,UT,Out) :-
    util:times(UT1), Out = [putt([~(UT1-UT),K]),nl].
```

メインモジュールでは、util:times(UserTime) で開始時間を計り、
N-Queen プログラムutil:queen:queen(N,Result) をフォールトトレラント実行するように指定している. 得られた結果は、count(Result,0,UserTime) で個数だけ調べ、調べ終ったところで終了時間を測定している. klicio は、入出力ストリームを獲得する述語である. 上の例では標準出力ストリームOut をとり、出力メッセージを送って画面出力させている. ~(K+1) はマクロ表記であり、演算結果と置き換わる.

N-Queen 問題を解くモジュールは、以下のようになる.

```
% N-Queen モジュール（オリジナル）
:- module queen.

queen(4,X) :- queen([1,2,3,4],[],[],X).
......
queen(8,X) :- queen([1,2,3,4,5,6,7,8],[],[],X).

queen([P|U],C, L,I):-           % clause 1
      merge(I1,I2,I),  append(U,C,N),
      c1(P,1,N,L,L,I1) @ node,     % goal 1
      queen(U,[P|C],L,I2).         % goal 2
queen([],[_|_],_,I):- I=[].     % clause 2
queen([],[],   L,I):- I=[L].    % clause 3

c1(T,D,N,[P|R],B,I):- T=\= P+D, T=\=P-D, D1:=D+1|
c1(T,D1,N,R,B,I).
c1(T,D,N,[P|_],B,I):- T=:=P+D | I=[].
c1(T,D,N,[P|_],B,I):- T=:=P-D | I=[].
c1(T,D,N,[],   B,I):- queen(N,[],[T|B],I).

append([A|X],Y,Z):- Z=[A|Z1], append(X,Y,Z1).
append([],   Y,Z):- Z=Y.

merge([X|I1],I2,I) :- I = [X|IT], merge(I1,I2,IT).
merge(I1,[X|I2],I) :- I = [X|IT], merge(I1,I2,IT).
merge([],I2,    I) :- I = I2.
merge(I1,[],    I) :- I = I1.
```

上のプログラムは、リスト位置とその数値で queen を置いたチェス盤面の位置を表す．リストの位置を盤面の行とすれば、初期値として与えたリストが[1,2,3,4,5,6,7,8] のように値が重複しない数値とされているのは、同じ列に queen は 2 つ置けないことを反映する．clause 1 に始まる述語定義では、第三引数にこれまで置いた queen を逆順に置き、この盤面に対して次の行に置けなかった列の位置を第二引数に記録する．第一引数には、次の行での候補の列位置がある．clause 3 は、次の行の候補列位置が無くなり、かつ置けなかった行位置もないなら、これまで置いた queen の盤情報を一つの解として第四引数に返させることを示す．clause 2 は逆に、queen が置ききれず解が見つからなかったことを示す．clause 1 は、これまで queen を置いてきた盤面L に対して、次の行の列P に置いた場

合goal 1 と、次の行についてP 以外の残りの候補で試すゴールgoal 2 の二つの部分問題に分割している． 各部分問題の解は、マージされて一つにされる．clause 1 では、P に置いた場合について、別ノードで実行するように指定している．

このプログラムを耐故障変換すると、以下のようになる．

```
% N-Queen モジュール (FTS)
:- module queen.

queen(N,X) :-
  watchdog:fork(Primary,Backup),
  queen_1({N,X},Primary,Backup) @ lower_priority(10).
```

フォールトトレラント実行の始めに、監視プロセスを各ノードに生成する．監視プロセスは、PSG、BSG それぞれのノードでリングネットワークを構成し、代表する監視プロセスへのストリームがそれぞれ返される．

監視プロセスをノード内で最高優先度で動作させるために、その他のプロセスは低い優先度に指定する．

```
queen_1(Args,Primary,Backup) :-
  copy:replicate(Args,Args1,Args2,ToAbt),
  ToAbt = {Abt1,Abt2}, Log = ack(Log1),
  Primary = {primary,queen,queen,Args1,Log,ToBackup,Abt1},
  Backup  = {backup ,queen,queen,Args2,Log1,ToBackup,Abt2}.
```

オリジナルゴールの引数の複製を作り、PSG と BSG にそれぞれ送る．

次のように、Record Version、Replay Version それぞれの述語定義を展開する．それぞれの述語には、ログ用の変数の他に、監視プロセスとのコミュニケーション用に 2 変数 (Sig,Raise) が、終了判定のショートサーキット用に 2 変数 (引数対SC) が追加される．

ゴールを別ノードへ送る部分は、ゴールを監視用プロセスへ送るユニフィケーションに置き変わる．

また、FTRepV では、監視用プロセスからのコマンドrebirth によって、FTRecV に置き換わるためのクローズが追加される．このクローズは、すでに PSG で生成されたログ情報を消費し終る必要があるので、alternatively を用いて優先度を低くしてある．

```
queen_record(4,X,Log,Sig,Raise)-SC :-
  queen_record([1,2,3,4],[],[],X,Log,Sig,Raise)-SC.
.....
% FT Record Version
queen_record([P|U],C, L,I,ack(Log),Sig,Raise)-SC :-
    Log = c1(L1,L2,L3),
    Raise = {Raise1,Raise2},
    merge_record(I1,I2,I,L1,Sig)-SC,
    append(U,C,N)-SC,
    Raise1 = [goal(primary,queen,c1,{P,1,N,L,L,I1},L2)],
    queen_record(U,[P|C],L,I2,L3,Sig,Raise2)-SC.
queen_record([],[_|_],_,I,ack(Log),Sig,Raise)-SC :-
    Log=c2(Ack), output(Ack,[I|Raise],[[]])-SC.

output(Ack,X,Y)-SC :- wait(Ack) | X = Y.
.....
% FT Replay Version
queen_replay(A,   B,   Log,Sig,Raise)-SC :-
    (wait(Log) -> queen_replay_1(A, B, Log,Sig,Raise)-SC ;
     alternatively;
     Sig = [rebirth|Sig1] -> queen_record(A, B, _,Sig1,Raise)-SC).
queen_replay_1(4,B,Log,Sig,Raise)-SC :-
    queen_replay([1,2,3,4],[],[],B,Log,Sig,Raise)-SC.
.....
queen_replay(A,B,C,D,Log,Sig,Raise)-SC :-
    (wait(Log) -> queen_replay_1(A,B,C,D,Log,Sig,Raise)-SC ;
     alternatively;
     Sig = [rebirth|Sig1] -> queen_record(A,B,C,D,_,Sig1,Raise)-SC).

queen_replay_1([P|U],C, L,I,Log,Sig,Raise)-SC :-
  Log = c1(L1,L2,L3) |
    L1 = ack(Log1), L2 = ack(Log2), L3 = ack(Log3),
    Raise = {Raise1,Raise2},
    merge_replay(I1,I2,I,L1,Sig)-SC,
    append(U,C,N)-SC,
    Raise1 = [goal(backup,queen,c1,{P,1,N,L,L,I1},L2)],
    queen_replay(U,[P|C],L,I2,L3,Sig,Raise2)-SC.
......
```

外部呼出し述語は以下のモジュールで定義される．これらは、監視プロセスが別ノードから投げられたユーザゴールを実行させるのに必要とされる．

```
% 外部呼出しモジュール（FTS）
:- module exgoal.

call_goal(Site,Module,Predicate,Args,Log,GSig,Raise)-SC :-
   call_goal_0(Site,Module,Predicate,Args,Log,
               GSig,Raise)-SC @ lower_priority.

call_goal_0(primary,queen,c1,{A,B,C,D,E,F},Log,GSig,Raise)-SC :-
    queen:c1_record(A,B,C,D,E,F,Log,GSig,Raise)-SC.
call_goal_0(backup, queen,c1,{A,B,C,D,E,F},Log,GSig,Raise)-SC :-
    queen:c1_replay(A,B,C,D,E,F,Log,GSig,Raise)-SC.
.................
```

8.4 変換に関する考察

変換によって、ソースレベルでは 耐故障部分だけで FTRecV、FTRepV と単純にみて 2 倍強になる．オブジェクトコードについてはオリジナル 9 Kbyte に対して、外部呼出し述語を含めて 32 Kbyte で、3.5 倍ほどになる．なお、ライブラリのオブジェクトコードは 59Kbyte 程度である．しかし KLIC 処理系にリンクすると、オリジナルの N-Queen プログラムと変換後の FTS の nCUBE2 向け実行形式のサイズは、それぞれ 1.19Mbyte, 1.26Mbyte であり、約 6%ほどの増加にすぎない．

KL1 プログラムを変換対称に選んだ利点は、ソースレベルでプロセスの非決定的な実行が静的に判別しやすいことと、プロセスの同期機構が言語レベルで保証されていることから、同期タイミングが変わることを気にせず変換できることにある．

他のプログラミング言語であっても、非決定的実行部分の抽出が機械的に可能であり、同期タイミングの変化にも対応できるならば、本研究の成果が適用できるものと思われる．

第 9 章

耐故障並列ソフトウェア実行オーバーヘッド

本章では、耐故障化したプログラムの実行オーバーヘッドについて検討し、無故障時の実行であっても使用に耐える条件を求める．この条件のもとであれば、本研究で提案する方式は十分な効率で動作する．さらに有効に動作する条件を決める因子を特定することで、本方式をより効率化する方法の検討へ発展できる．

9.1 実行オーバーヘッドの分類

実行オーバーヘッドは、プログラムから静的に見積もれるものと、動作させた時に発生し得る動的なものがある．静的なオーバヘッドにより、プログラムに沿って単一プロセッサ上で間断なく動作する際の実行時間を見積もることができる．別プロセスの並行動作、別プロセッサからの非同期なメッセージ送信等の予測困難な動作は動的なオーバヘッドとなって観察するプロセスの実行に影響を与える．動的なオーバヘッドについては見積りは困難であるが、考察することで静的解析からの値と実測値との差を埋める助けとなる．

9.1.1 静的オーバーヘッド

変換後のプログラム構造から、静的にオーバーヘッドを考察する．変換によって以下の処理が、すべてのプログラムに追加される．

1. 監視プロセスのネットワークを構成させる．

2. ゴールの複製を作る．

3. PSG、BSG へゴールを転送する．

監視プロセスを各プロセッサ上で起動させる際には、次のような処理が行われる．

1. PSG、BSG のノードを決定する．

2. ホストノードから PSG、BSG の各プロセッサへゴールを送出する．

3. 送出されたゴールは、各プロセッサ上からホストノードへ返信を送る．

4. 返信を送って来たプロセッサどうしをリングネットワークに接続する．

この際、明示的には各プロセッサとホストノードとの間に、3 回のコミュニケーションが必要である．ゴールの複製については、実際に複製されるデータ量に比例した処理オーバヘッドがある．

第8.1.1節 で述べたように、FTRecV、FTRepV のプログラムでは次の処理が追加される．

FTRecV に追加された処理：

1. 入力処理：直前の非決定的実行に対する、返信の待ち合わせが追加される．
2. 部分処理：条件分岐が非決定的であるならば、分岐情報をログとして送る処理が追加される．

静的には、抽象マシン語レベルで数命令〜10 命令程度での増加が見込まれる．十分大きな述語定義であれば全体として数%〜10%程度の増加である．プロセッサ間メッセージの送受信処理を伴うため、メッセージ処理オーバーヘッドを考慮する必要がある．なお決定的述語については、ほとんど元のプログラムと同じである．

FTRepV に追加された処理：

1. 割り込み処理：監視プロセスからの割り込みをチェックする処理が追加される．
2. 入力処理：非決定的実行に対する、ログの待ち合わせが追加される．
3. 部分処理：ログに対する返信を送る処理が追加される．

入力処理以下については、FTRecV と同様で 10 命令程度にすぎないが、割り込み処理に 1 リダクション追加される．また、入力処理と部分処理についても、プロセッサ間メッセージ処理を考慮する必要がある．

9.1.2 動的オーバーヘッド

耐故障プログラムの実行時には、次のようなオーバーヘッドが考えられる．

1. 監視プロセスに関する処理

   (a) 故障検出

   故障検出のためには、監視プロセス間でのメッセージ交換が必要である．監視プロセスは物理的に別プロセッサ上で動作するため、メッセージ処理を考慮しなければならない．PSG でのメッセージ転送は、PSG のリングネットワークを 1 周する分と、最後に BSG の監視プロセスへの 1 メッセージ転送が含まれる．よってサイトを構成するプロセッサ個数＋１個分のメッセージ処理が必要である．故障検出は、一定サイクルごとにタイマー駆動で発生する．もちろん故障検出サイクルを長くとるほど全体のオーバーヘッドは減少する．

   (b) サイト内ゴール転送

   ゴール転送は監視プロセスを介して行われるため、ユーザのゴールプロセスから監視プロセスへの通知がプロセッサ内で起こり、プロセスチェンジした監視プロセスの 1 リダクションが必要になる．

   (c) 制御ストリームのマージ

   監視プロセスへの制御ストリームは、ノード内で実行するすべてのゴールからマージされる．制御ストリームを使わない場合でも、閉じる処理のために少なくとも 1 メッセージが監視プロセスへ送られる．監視プロセスへ割り出すことのない述語であると静的に解析されれば、このオーバーヘッドは軽減される．

ユーザプログラムとの関係では、通常のゴール転送が、

- 制御ストリームへの制御ストリームへのユニフィケーション
- 制御ストリーム内でのマージ

- 監視プロセスのプロセスチェンジ
- 監視プロセスでのリダクション
- 監視プロセス間のストリームへのユニファイ

  これによって、ゴールはプロセッサ間を一般のデータとして転送される.

2. ホストプロセッサと PSG、BSG 間で動的に通信されるデータの複製処理

   ホストプロセッサ上のクライアントと PSG、BSG 上のサーバプログラムとの通信は、クライアント側からのデータ転送は複製されて PSG、BSG へと送られ、PSG、BSG 側からの転送は先に来たものをクライアントへと渡す. これら処理はホストプロセッサ上の複製プロセスが行う. 互いに転送されるデータ量に比例したリダクションが複製プロセスで必要になる.

3. FTRecV と FTRepV との ログ/返信の転送遅延

   転送遅延はプログラムの性質により大きく異る. ログを転送する必要のある非決定的実行間隔がまばらであれば転送遅延は 0 になる. 逆に頻繁であれば、FTRepV の実行時間＋ネットワーク転送時間かかる. 遅延はログとその返信の転送に関係して発生する. よって遅延時間は非決定的実行の量に影響される.

4. プロセススイッチ

   プロセッサ上の各プロセスは、プロセッサ間メッセージの到着によって動的にプロセススイッチする. また監視プロセスへの割り出しも監視プロセスへのトリガーとなる. KLIC の実行上はサスペンド、リジュームのコストに含まれる.

5. その他ガーベジコレクション等処理系がバックグラウンドで行う処理

   並列論理型言語は動的なメモリ割り付けを許すため、メモリの消費状況によってガーベジコレクションが発生する. その他処理系がバックグラウンドで動的に行う処理については見積り不可能である. 通常では発生頻度が低いと思われるため無視できるが、メモリ消費の激しいプログラムについてはガーベジコレクションの発生回数は大きく実行性能に影響する.

9.2 実行オーバーヘッドの見積り

上の考察を元に耐故障化されたプログラムの実行性能を見積る．まず単一プロセッサで動作させた場合の実行性能を見積もる．実行性能は、元のプログラムとの実行時間比によって示す．次に並列計算機上での実行性能についての見積りを行う．単一プロセッサでの実行とは、PSG、BSG それぞれ 1 プロセッサで構成する場合を指す． 並列実行時には PSG、BSG が複数プロセッサで構成される．

9.2.1 単一プロセッサでの実行見積り

PSG、BSG 各々が 1 プロセッサで構成される際の実行見積りを行う．ここでは監視プロセスについては考慮しない．またホストプロセッサとの通信量は、サーバ側のプログラムだけで静的に見積れないことから、ホストプロセッサとの通信処理については最初に無視した値を見積もる．従って以下の実行見積りは、ホストプロセッサとの通信の少ないものにはそのまま当てはまる．ホストプロセッサ間通信によるオーバヘッドは通信量にのみ影響することから別途に行う．

まず通常のユーザプログラムの実行について見てみると、単一プロセッサ上であるので通信コストは存在しない．よって、実行時間を決定するのはリダクション数とサスペンド数である．

R ：平均リダクション時間

n ：非決定的リダクション実行数

d ：決定的リダクション実行数

S ：サスペンド リジューム時間

s ：サスペンド回数

とすると、通常のプログラム実行時間 TS_{normal} は、総リダクション数 $(n+d)$ とサスペンド数 s に対してかかる処理であり、

$$TS_{normal} = R \times (n+d) + S \times s$$

となる．

耐故障実行においては非決定的実行量が処理を重くする要因となるため、対比のためリダクション実行数を非決定的実行と決定的実行に分けた．ここでは平均リダクション時間 Rを用いるが、決定的リダクション実行時間と非決定的リダクション実行時間の差が大き

く、各々の実行数の差も大きければ無視できなくなる．リダクション時間は、非決定性からのみ決定できないためここでは平均時間とする．

耐故障化したプログラムでは、PSG 側の実行に注目すれば考慮しなければならないのは次である．

- 引数が追加されたため、通常のリダクションコストが元のプログラムよりも若干高くなる．これは非決定的述語において顕著であり、決定的述語については差は小さい．
- ログの送信コストと返信の受信コストが加算される．

  ログは非決定的リダクション実行数分発生する．返信も同じ回数起こる．
- 返信待ち時間がかかる．
- 返信待ちが生じた際のサスペンド回数 s が増加する．

よって、

α : 耐故障プログラムのリダクション時間の増加比率

これは非決定的実行に大きく影響を与えるため、以下では非決定的実行にのみ積算する．しかし決定的実行時間もごくわずかであるが増加しており、より精密に求めるならば別個に加重する必要がある．

s' : FTS のサスペンド回数

M : メッセージ送信（受信）時間

メッセージ処理は非決定的リダクション n 毎に、ログの送信と返信の受信があるので総メッセージ処理時間は、$2nM$ となる．

W : 返信待ち時間

とするとき次を得る．

$$TS_{fts} = R \times (\alpha \times n + d) + S \times s' + 2 \times n \times M + W$$

次に具体的な実行時間を推定してみる．まず非決定的実行は、値の待ち合わせを頻繁に行うことが多いことから、$s = N$ とする．耐故障化プログラムでは、さらにログの返信待ち

も起こることから、2 回づつサスペンド リジュームするとして、$s' = 2 \times N$とする．リダクションコストの増加比率は 1 割増し（$\alpha = 1.1$）程度とし、返信待ち時間無し（$W = 0$）とした時、耐故障化プログラムと元のプログラムの実行時間比を表したものが図 9.1である．

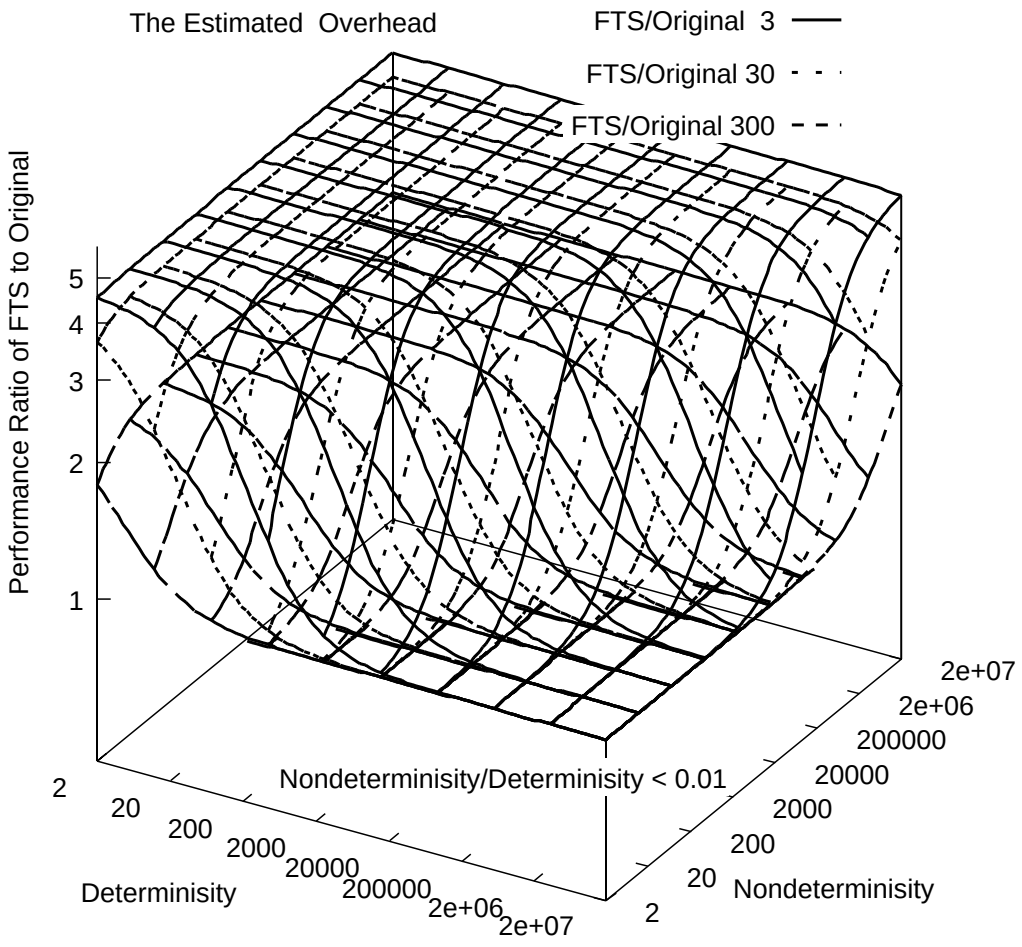

図 9.1: 単一プロセッサでの非決定性/決定性に対する推定オーバーヘッド

図 9.1の曲線は、メッセージ送受信時間を $M = 200$、サスペンド リジューム時間を $S = 150$ とし、リダクション時間は、$R = 3$、$R = 30$、$R = 300$ の場合を示している．リダクション時間は、単純にループするような軽いものから、ユニフィケーションを多数行う上にゴールを多く生成するようなものまで変動し得る．各曲線は、これらゴールの重さの変動に対応したものである．

図 9.1を見ると、各グラフとも非決定性が決定性の 1%以下であれば、元の 2 倍以下で耐故障実行ができると見込める．また、リダクションコスト Rが大きくなるにつれて、非決

定性の比率が高くても耐故障実行のオーバーヘッドは相対的に小さくなることもわかる．

図 9.2は、非決定的実行/決定的実行比率と送受信コスト / リダクションコスト比率に対する性能変化を示したグラフである．送受信コストがリダクションコストの 10%以下であれば、非決定性が決定性の 1000 倍でも耐故障実行は 2 倍までとなる．逆に送受信コストがリダクションコスト 10%以上ならば、非決定性が決定性の 1%以下でなければ十分な性能は得られないことが分かる．

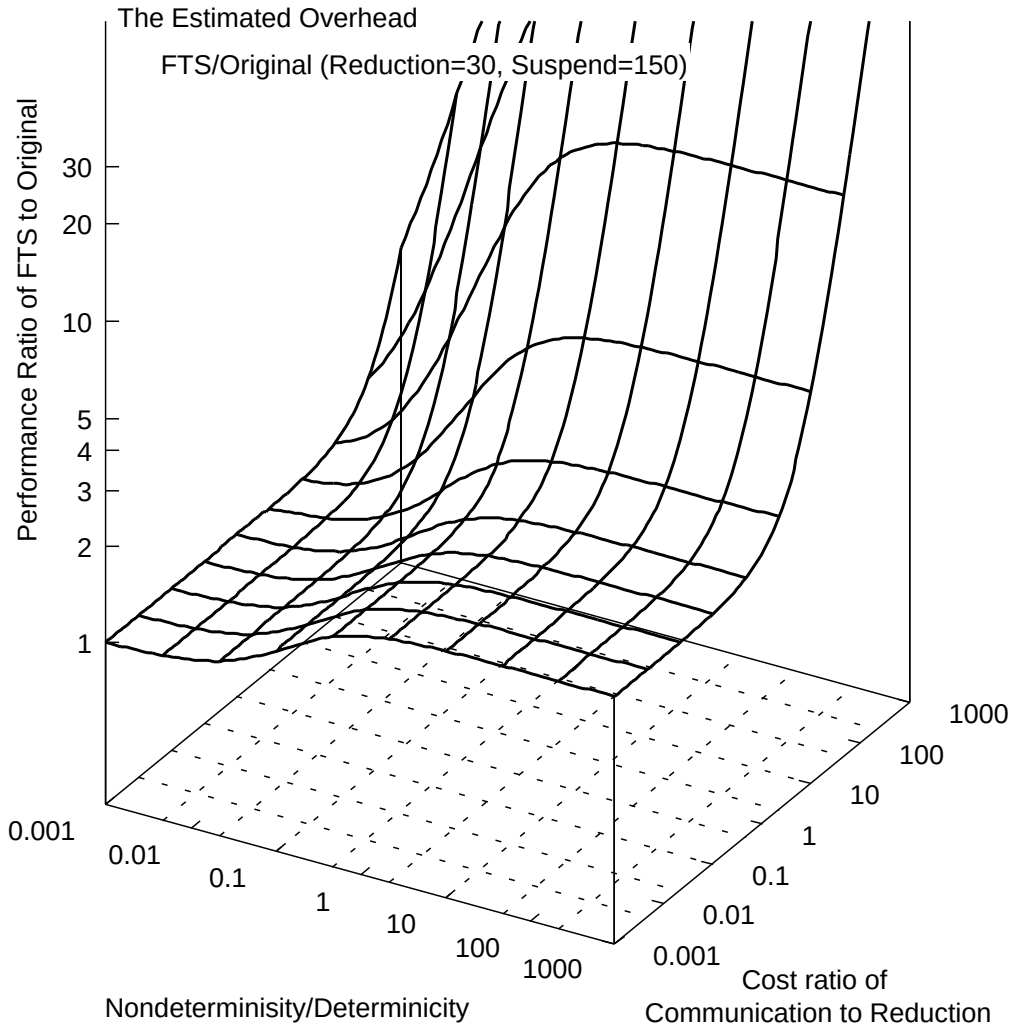

図 9.2: 単一プロセッサでの通信コストを考慮した推定オーバーヘッド

9.2.2 マルチプロセッサでの実行見積り

PSG、BSG が複数プロセッサから構成される場合の耐故障プログラムの実行効率を見積もる．耐故障化する前のプログラムを、同じプロセッサ数からなるマルチプロセッサ上で実行した場合との処理時間の比較によって実行効率を表す.

ここで、

N ：PSG を構成する要素プロセッサ数

O ：単一プロセッサで実行する際の元のプログラムの仕事量

F ：単一プロセッサで実行する際の耐故障プログラムの仕事量

PSG/BSG のコミュニケーションも含む.

とする.

アムダールの法則によれば、Oの並列化可能部分率 p として、N台で並列実行する場合には最大負荷のプロセッサの仕事量が $pO/N+(1-p)O$ になる [20]．しかしこの値にはプロセッサ間の通信は考慮されていない．N分割することによって他のプロセッサ上へ分散されたプロセスとプロセッサ間コミュニケーションが必要な場合、プロセッサ内の単位処理時間を R、プロセッサ間メッセージ処理時間 M、メッセージ数 m とするとき、負荷バランスが均衡したとして並列実行による処理時間は、通信遅延を無視して $(pO/N+(1-p)O)R+mM\ \ (N>1)$ となる．よって並列処理をして意味がある、すなわち効果が得られるためには、

$$\frac{1}{N} \leq \frac{(pO/N+(1-p)O)R+mM}{OR} < 1$$

であることが必要十分である．これから、$0 \leq NmM < (N-1)pOR$ を得る.

$$\left(\begin{array}{c} \frac{1}{N} \leq \frac{p}{N} + 1 - p + \frac{mM}{OR} < 1 \\ -(1-p)(1-\frac{1}{N}) \leq \frac{mM}{OR} < p(1-\frac{1}{N}) \\ 0 \leq \frac{mM}{OR} < \frac{p(N-1)}{N} \end{array} \right)$$

すなわち Nが十分大きいとすれば、$mM/OR < p \leq 1$ より、プロセッサあたりのメッセージ処理時間 mMは、単一プロセッサでの処理時間 OR すなわちマルチプロセッサシステム全体で処理しようとする仕事の単体での処理時間よりも小さくなければならない.

ここで改めて

R : 平均リダクション処理時間

M : 平均メッセージ処理時間

m : Oを並列化することで生じるメッセージ数

とすると、

$$TP_{normal} = \left(\frac{p}{N} + (1-p)\right) OR + mM$$

が並列化した元のプログラムの実行時間である.

元のプログラム自身のコミュニケーション量は耐故障化しても変わらないものとし、また仕事の分散とともに、PSG/BSG のコミュニケーションも分散されるとする．よって、耐故障プログラムの実行時間は、上と同様に

$$TP_{fts} = \left(\frac{p}{N} + (1-p)\right) FR + mM$$

となる．ここで、

α : 耐故障化によって増える仕事量比率（$F = \alpha \times O$ ）

β : 潜在するコミュニケーション量の元のプログラムに対する比率（$m = \beta \times O$）

γ : メッセージ処理とリダクション処理の処理時間比（$M = \gamma \times R$）

とすると、処理時間比は、

$$\frac{TP_{fts}}{TP_{normal}} = \frac{\left(\frac{p}{N} + (1-p)\right) FR + mM}{\left(\frac{p}{N} + (1-p)\right) OR + mM} = \frac{(p + (1-p)N)\alpha + \beta\gamma N}{(p + (1-p)N) + \beta\gamma N}$$

となる.

並列プログラムは本質的に並列性が高いものと考えて、並列化部分率を $p \simeq 1$ として、$\beta \times \gamma$を x 軸にとって、$\alpha = 10$ 固定で N =10〜10000 の場合をプロットしたのが図 9.3である．すなわち、耐故障化することでユーザプログラムは単体で 10 倍の仕事量に増えたとして、10 台から 10000 台規模のプロセッサで耐故障実行させた場合に相当する．なお並列プログラムが意味を持つ条件 $0 \leq NmM < (N-1)pOR$ は、$\frac{N}{N-1}\beta\gamma < p \leq 1$ であるので、$0 \leq \beta\gamma \leq 1$ の範囲だけ意味を持つ．ここで$\beta\gamma(= \frac{mM}{OR})$ は、並列化することで生じる

メッセージの処理時間とプログラムを単体プロセッサで逐次実行した場合の処理時間の比である．

これを見ると、システム規模が大きくなるほど $\beta\gamma$は任意にとっても、性能低下しない．システム規模が 100 台程度であれば $0.1 < \beta\gamma \leq 1$ でなければ 2 倍以上の実行時間となってしまう．従って本方式は大規模並列計算機であるほど有効であり、その性能は、メッセージ量とその処理時間、プロセッサ内の処理量と処理時間の比に依存する．

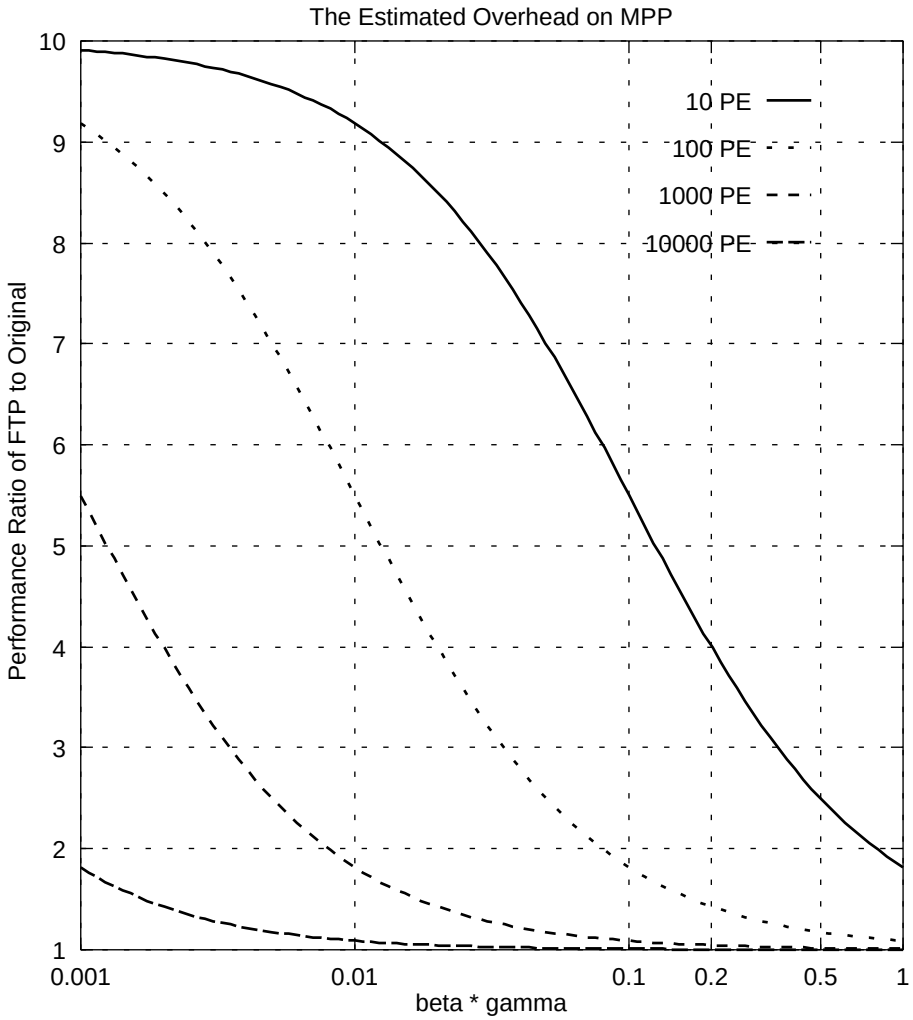

図 9.3: マルチプロセッサでの推定オーバーヘッド $(p = 1)$

図 9.3のグラフは並列化部分率 $p = 1$ として得たものだが、一般には $p < 1$ であり、その値に応じて図 9.4のような曲線を描く．なおこのとき、$0 \leq \beta\gamma < p$ でのみ、並列プログラ

ムとして意味がある．これを見ると、並列度が低い $(p << 1)$ プログラムの場合、$\beta\gamma$も小さくなり、急激に処理時間比が悪化することが分かる．図 $9.3(p = 1)$ の$\beta\gamma = 0.01$ のように 1000PE で 2 倍程度であったはずの処理時間比も、$p = 9$ になると 9 倍程度になる．p が 1 に近いところの変化を見るために、図 9.4を $0.99 \leq p \leq 1, 0.01 \leq \beta\gamma \leq 0.1$ の範囲で、$\beta\gamma$ 軸を対数軸としてグラフを描き直したものが図9.5である．これを見ても並列度がかなり高く $(p \to 1)$ ないと、性能を向上が期待できないことが分かる．メッセージ処理効率が上昇するか、あるいはメッセージ数が減少して処理時間 mMが小さいプログラムの場合には、$\beta\gamma$が減少することから、やはりプログラムの並列性の高さがより支配的になる．これはサイトを構成するプロセッサ数が大きくなるほど顕著である．

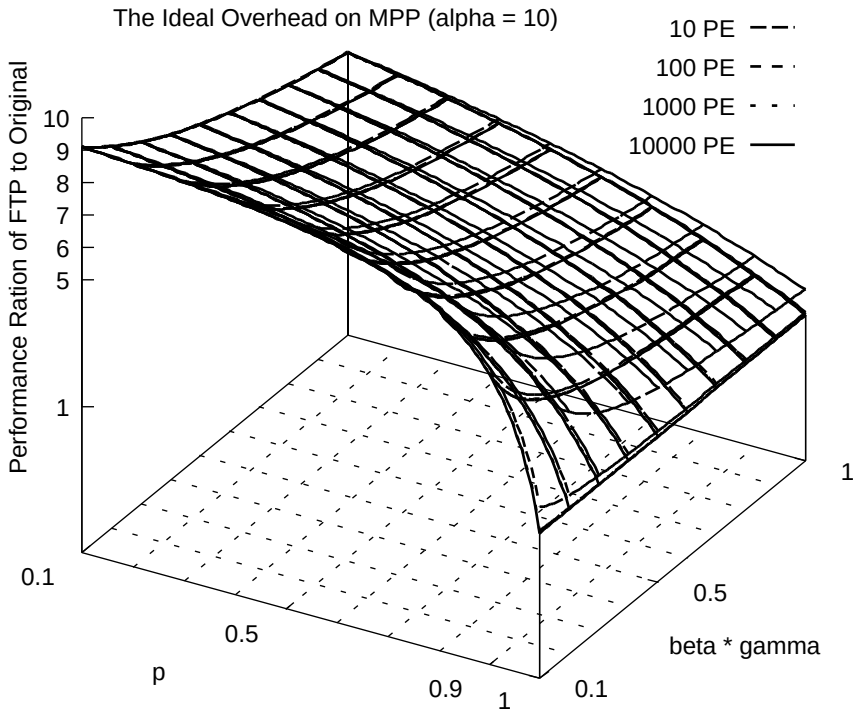

図 9.4: マルチプロセッサでの推定オーバーヘッド $(0.1 < p < 1)$

9.3 実行オーバーヘッド見積りに関する考察

本章では、実装するシステムの基本性能からユーザプログラムを耐故障化した際のオーバーヘッドを見積もった．

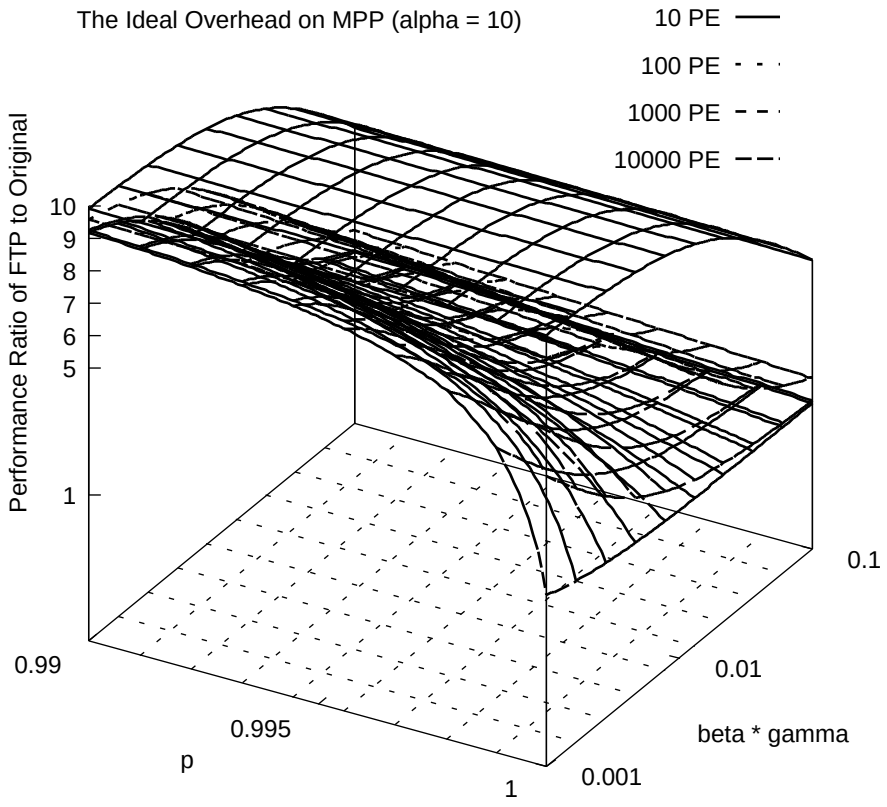

図 9.5: マルチプロセッサでの推定オーバーヘッド $(0.99 < p < 1)$

その結果、耐故障実行が、サイト単一プロセッサ構成で十分な効率で動作するには、メッセージ処理性能がプロセスの単位処理性能の 10%以下であるか、またはユーザプログラムの非決定性が決定性の 1%以下であることが必要であることが分かる．並列システム上での処理時間比を見ると、元のプログラムの並列度が最も大きく影響していることが分かる．プログラムの並列度がかなり高いものでないと、耐故障化処理オーバーヘッドの影響を大きくうける．もちろん並列度が高い場合であっても、負荷分散が十分に行えなければ上の理論値から外れることになる．

なお、この時ユーザプログラムの並列化可能率を $p(0 \leq p \leq 1)$ 、ユーザプログラムの並列実行によって生じるメッセージ処理時間 mM とユーザプログラムの単一プロセッサ上での実行時間 ORの比$\beta\gamma = mM/OR$とするとき、$\beta\gamma < p \leq 1$ の範囲でのみ評価できることを示した．これを満たさないプログラムは、並列実行し遅くなるクラスのプログラムであり、並列計算に向かないからである．よって、並列度が低くなればメッセージ処理時間比

も十分小さくならなければ、並列処理に向かないことが分かる.

従って本研究で提案する方式が、並列効果によって処理オーバーヘッドを低くおさえるためには、元のプログラムにかなり高い並列度があり、かつプロセッサ間コミュニケーション量が多いものでなければならない. 純粋に並列言語として設計された並列論理型言語では、並列性の高いプログラム記述も他の言語記述よりも可能性が高いとはいえ、ユーザの設計するアルゴリズムの並列性に大きく依存することは否定できない. 結局のところ並列効果によるオーバーヘッドの低減の可能性はあるものの、サイト構成するプロセッサが単一プロセッサであるときの処理オーバーヘッドを減少させることが本システムの性能向上につながる.

第 10 章

実験及び評価

nCUBE2 上において、基本機能の性能を測定するプログラム、耐故障性能を計測する人工的なベンチマークプログラム、そして応用プログラムとして 8-Queen 問題を解くプログラムを KLIC を使って動作させた結果について述べる．これら実験結果を第 9章で推定した性能見積りと照合して、予想された性能を引き出せているか分析する．

10.1 実験環境

10.1.1 ハードウェア

nCUBE2 は、MIMD のメッセージパッシング型並列計算機であり、各 PE はハイパーキューブ網によって結合される．各 PE は独自の 64bitCPU を用いており、10MIPS 相当の処理能力を持つ．実験に用いたシステムは 32PE からなり、各 PE が保有するメモリ容量は 32MB の設定である．通信チャネルは、実測で、バンド幅が 2.2MB/s 、通信オーバヘッドは send 140 (μsec)、receive 55 (μsec)である．

10.1.2 処理系

KLIC で変換した C プログラムは、nCUBE 向けの ANSI C 準拠の C コンパイラを用いてコンパイル実行される．KLIC は、PE 間コミュニケーションに対するオプションとしてバッチ転送モードを設けている．通常モードでは要求されるたびに PE 間でデータ構造を転送するため、ネストしたデータ構造は 1 回に 1 レベルずつ転送する．これに対してバッ

チ転送モードでは、ネストしたデータ構造を 1 回にまとめて転送する．nCUBE2 では通信オーバヘッドが大きいため、バッチ転送モードの方が効率が良い場合が多い．しかし通信時のタイミングしだいでは、データ構造が十分に構築されていないため必ずしも通信量が減少しないことがある．以下の実験では特に断らない限りバッチ転送モードを用いる．

10.2 基礎性能測定

10.2.1 リダクションコストの測定

基礎データ測定用プログラムによれば、図 10.1のような単純なループ処理であれば末尾再帰呼出し（TRO）が効くため単一 PE 上で 1.6μ sec/reduction (約 600K reduction/sec) の性能が得られている．append プログラムで Sparc コード約 20 命令であるから、単純なループであれば 10MIPS の nCUBE2 で 2μsec/reduction となり、上の値は妥当である．図 10.2のような 1 演算ごとに値を順次リストで返すプログラムになると、命令数の増加から 3.3μsec/reduction ほどになる．

```
loop(N,X) :- N > 0, N1:=N-1 | loop(N1,X).
otherwise.
loop(_,X) :- X = [].
```

図 10.1: リダクションコスト測定プログラム (単純なループ)

```
loop(N,X) :- N > 0, N1:=N-1 | X = [N|X1], loop(N1,X1).
otherwise.
loop(_,X) :- X = [].
```

図 10.2: リダクションコスト測定プログラム (出力を含む)

図 10.3が、実際の図 10.1と図 10.2の C 命令セットの差分である．この内容は、N（a0）と変数X1（x2）を含むリストセル[N|X1]（x1）を割り付け、X（a1）とユニファイし、X1 を

```
allocp[0] = x2 = makeref(&allocp[0]);
allocp[1] = a0;
x1 = makecons(&allocp[0]);
allocp += 2;
unify_value(a1, x1);
a1 = x2;
```

図 10.3: 差分 C 命令

次の再帰呼出しの引数に設定する操作である．Sparc コードでは、これらの処理は 11 命令になり、1μsec 程度の増加となりほぼ妥当な値である．

10.2.2 サスペンドリジュームコストの測定

次に図 10.4は、ゴールsusp(X) がループ毎に一つサスペンドし次々にリジュームするプログラムである．

```
loop(R,N,X) :- N > 0 | R = 0,     % 前のサイクルの束縛を解消する.
     suspG(R1,X,X1),              % R1 が束縛されるまで待つ.
     N1 := N-1, loop(R1,N1,X1).   % R1 は次のサイクルで束縛される.
otherwise.
loop(R,_,X) :- R = 0, X = [].

suspG(R,X,X1) :- wait(R) | X = X1.  % サスペンドする.
```

```
suspG(R,X,X1) :- X = X1.            % サスペンドしない.
```

図 10.4: サスペンドリジュームコスト測定プログラム

述語loop の 1 回の実行サイクルにおいて、ゴールsuspG がその第一引数の未束縛のためにサスペンドする．次のloop 実行サイクルでsuspG のサスペンド原因が解消され、リジュームする．これに対して、suspG のクローズだけを 図 10.4の下のボックスにあるクローズに置き換えることで、サスペンドしないプログラムが得られる．これら二つのプログラムの

実行時間差から、サスペンドリジュームあたり 165μsec かかることが分かる.

10.2.3 コミュニケーションコストの測定

標準的なプロセス間コミュニケーションは図 10.2のように変数ストリームへコンスセルをユニファイすることで行われる. コミュニケーション時間は、図 10.5のプログラムにより計測を行った.

```
 ?- server(Channel)@ node(1),
    client(N,Channel,End)-Out @ node(0).

server([X|L]) :- X = 0, server(L).
server([]).

count(UT,Ch,End,N,K) :- integer(UT), K < N, K1:=K+1 |
    Ch = [Next|Ch1],
    count(Next,Ch1,End,N,K1).
otherwise.
count(UT,Ch,End,N,K) :- Ch = [], End = 0.

client(N,Ch,End)-O :- Ch=[Start|Ch1],
    client1(Start,N,Ch1,End)-O.
client1(0,N,Ch,End)-O :-  time(UserTime),
    count(UserTime,Ch,E,N,1),
    end(E,UserTime,End)-O.
end(0,UT,R)-O :- time(UT1),
    O <= putt(["(UT1-UT)]), O <= nl, O <= fflush(R).
```

図 10.5: コミュニケーションコスト測定プログラム

このプログラムの１サイクルは、図 10.6のように実行される.

stage 0: PE0 で動作するクライアントプロセスcount は、コンスセル[X1|X2] をX に束縛することでサーバーに値を要求する. しかし変数X のセルがPE0 上にあるためにコンスセルは実際にはまだ送られない. 述語count の次のサイクルcount(X1,X2,..) は、要求した値が来るまでサスペンドする.

stage 1: PE1 で動作するサーバープロセス`server(X)` は、クライアントから空のコンスセルが送られることにより動作する． 変数`X` の実態は PE0 にあるため、PE1 から read（値の要求）メッセージを送ってサスペンドする．

stage 2: read メッセージを受けとった PE0 では、コンスセルは変数`X` にすでに束縛しているので、answer（返信）メッセージにより PE0 に送り返す．

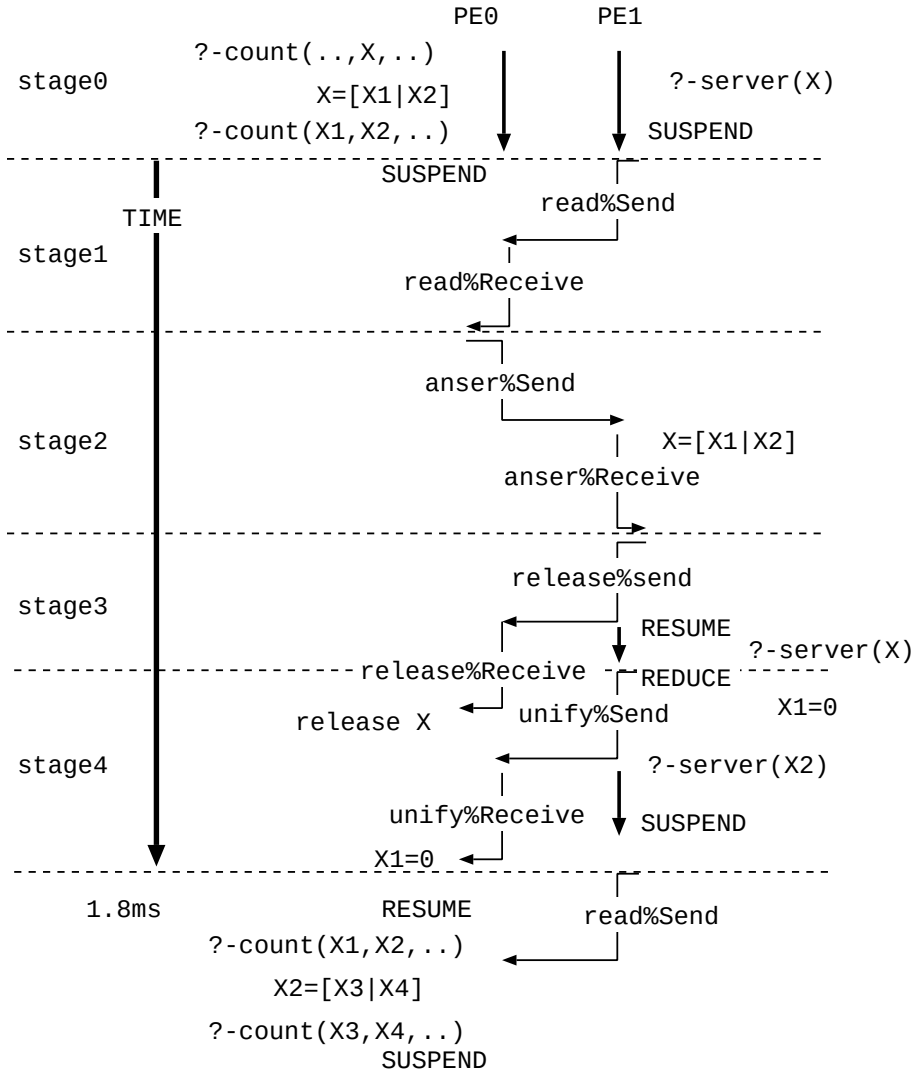

図 10.6: 基礎データの測定（コミュニケーション）

stage 3: コンスセルを受けとった PE1 は、release（分散 GC 用の領域の解放）メッセージを送り返すことで、PE0 上の変数`X` の領域が不要になったことを知らせる．

stage 4: 続いてPE1は、受けとったコンスセルにある変数`X1`に値0を束縛しようとする．`X1`の実態はPE0にあるために unify（値の転送）メッセージが送られる．unify メッセージによってPE0上の`count`の次のステージが開始する．

release メッセージ送出後は連続して実行が可能であるのと、PE内のリダクションはメッセージの送受信に比べればわずかであるので、stage 1〜stage4での1サイクルでは、ほぼ7回の送・受信処理を行っているとみなせる．測定によれば1サイクルあたり1800μsecであることから、受信処理と送信処理が同等であると仮定すれば各々260μsecである．バッチ転送モードでは転送量により多少増加するが、処理性能にとっては処理メッセージ回数が支配的である．

10.2.4 基礎性能測定結果

以上の実験により、表10.1のように見積りに必要な基礎性能パラメタが得られた．この値を用いて、以下に行なう実験の結果から、第9章で求めたオーバーヘッド見積りの正当性を検証する．

表 10.1: nCUBE2 上のシステムにおける基礎性能

		μ sec	
リダクション	R	1.6	$= R_d$ (単純なループ)
		3.3	$= R_n$ (出力を含む)
サスペンド リジューム	S	165	
コミュニケーション	M	260	(送信/受信処理)

10.3 実験方法

本研究で提案する耐故障化プログラムは、プログラム中の非決定性が性能を左右する．普通のユーザプログラムでは、非決定性を変化させることができないため、非決定的実行を制御できる人工的なベンチマークプログラムを用意した．また人工的なベンチマークプログラムだけでは実際のプログラムと大きく異ることも考えられることから、より実際的

な応用プログラムとして 8 Queen 問題の探索プログラムを用意した．8 Queen プログラムは比較的性質も分かっており、ある程度解析もでき利用しやすい．

10.3.1 ベンチマークプログラム

ベンチマークプログラムは、図 10.7のようなプロセスから構成される．

決定的プロセス： 決定的実行を $(d-n)/2$ 回行った後、各実行毎に出力を行う決定的実行を $n/2$ 回行う．全体で決定的実行 $d/2$ 回と出力 $n/2$ を行うプロセスである．実行に含まれる出力以外の処理は、条件判定（整数比較)、演算（整数減算)、再帰呼出しである．

非決定的プロセス: 2 つの生成プロセス各々からのデータを非決定的に受けとり、加算する．従って全体の実行は、n 回である．実行に含まれる処理は、データの受けとり、演算（加算)、再帰呼出しである．

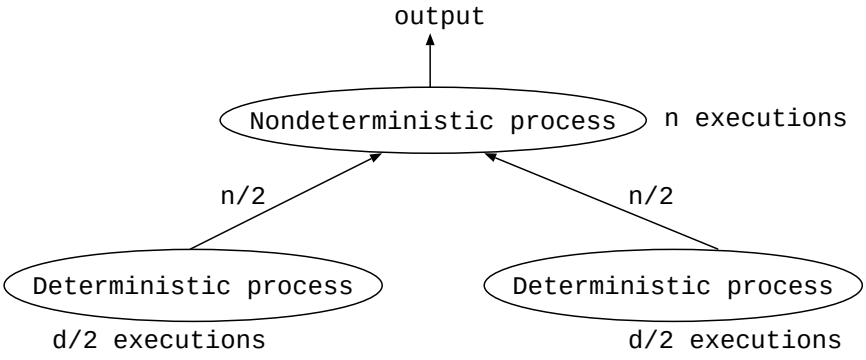

図 10.7: 実験用ベンチマークプログラム

二つの決定的プロセスと一つの非決定的プロセスによって、n 回の非決定的実行、d 回の決定的実行が行われる．この 3 プロセスを 1 ユニットとして、各プロセッサで動作させる．実験は、ノード数の増減に対して負荷一定のものと、ノード数の増加に応じて処理負荷を増やしたものと 2 通り行った．ノード数の増加に応じて処理負荷を増やした実験は、アムダールの法則に従った性能が得られるものと期待される． 一方ノード数によらず負荷一定とした実験は、実行時間を固定にして処理容量を増やすため、より並列処理向きのプログラムによる実験である [20]．

オリジナルプログラムは、ホストプロセッサ（PE0）と複数の処理プロセッサで実行される．処理負荷をノード数に応じて増やしたものは、システムのスケールに応じて処理負荷が増えるタイプのプログラムと見ることができる．負荷一定でノード数を変えて実行するプログラムは、並列効果によって高速に処理を行いたいプログラムに相当する．

10.3.2 応用プログラム

応用プログラムとして 8-Queen 問題の全解探索プログラムを用意した．

Original：8-Queen 問題の全解を探索して、解を非決定的順序で返すプログラムである．プログラムは、部分問題に順次分割して解くスタイルを持つ．

負荷分散させる部分問題を変えたものを 2 種類（type-1、type-2）と分散する部分問題が多いもの（type-3）を用意した．分散先 PE は部分問題のハッシュ値によって決めており、どれも分散のバランスは均衡しやすい．type-1、type-2 は部分問題の再上位だけ分散させるため通信量も少ないが、分散させる部分問題により微妙に挙動が異る．type-3 は、type-2 の部分問題をすべてのレベルについて分散させるため、通信量が多くなる．

耐故障化プログラム（FTP）：NDP で用意した 3 種類（type-1、type-2、type-3）を耐故障化変換して得たプログラムである．

なお FTP では監視プロセスが分散先を勝手に決めるため、NDP と同じ負荷分散にならない．特に、NDP はハッシュ値が自 PE を指せば分散させないが、FTP ではこれも分散対象としている．

type-3 を用意したのは、通信量の多いプログラムに対する特性を見るためである．このプログラムは、通信量は多くなるが負荷単位が小さいためマルチプロセッサ間の負荷を均衡化させやすい．一方、type-1 と type-2 は負荷単位が大きいため並列実行させた際の通信量が小さくなるが、負荷が均衡しないときには性能向上しにくい．ベンチマークプログラムとは異り、実際に分散させる負荷の見積りは難しい．そこで、type-1、type-2 と分散させるプロセスを変えることで、別の負荷バランスの実行を得ることがこのプログラムを用意した目的である．type-1、type-2 は、プログラムの同じクローズに対して、分散させるゴールを変えている．これだけの相違でも以下の実験結果のように、大きく異る挙動のプログラムを得ることができる．

10.3.3 測定項目と測定方法

各プログラムの実行処理時間を測定する．処理時間の測定方法は、ホストプロセッサ(PE0)上で動作する測定プログラムでのタイマー計測による．測定プログラムは、開始時間を獲得したことを確認した後実験プログラムを起動する．実験プログラムが実行結果として返すリストをリスト終端まで読みとった後、終了時間を計る．タイマー呼出し等のため、耐故障化前の元のプログラムの実行でも若干の測定オーバーヘッドがある． 耐故障プログラムでは、監視プロセスと複製プロセスのライブラリプログラムの起動処理も測定時間に含まれる．耐故障プログラムでは、実験プログラムから返すリストは、ホストプロセッサ上で動作する複製プロセスを介して渡されるが、この処理時間も含まれる．

10.4 実験結果

10.4.1 ベンチマークプログラム

実験結果を図 10.8、図 10.9、図 10.10、図 10.11、図 10.12、図 10.13、図 10.14に示す．

図 10.8は、耐故障化する前の元のプログラムに対する耐故障化したプログラムの実行時間比を表す．横軸は非決定的実行数、縦軸は実行時間比である．各曲線は、決定的実行数を 2 から 2000 まで変えた実行を表している．破線は、非決定性/決定性の同比率の点を結んだものである．この実験結果から、非決定性が決定性に対して 0.01%以下であれば、1.3 倍程度で耐故障実行できることがわかる．あるいは十分大きな処理量であれば 0.1%程度であっても 2 倍程度に抑えられている．

この実験結果を第 9章の見積り方法による理想曲線と比較したものが図 10.9である．実測値は非決定性が 200 の結果と 2000 の結果を用いた．

このプログラムでは、全体の実行回数 $(n+d)$ のうち、d 回が単純なリダクション（$R_d = 1.6\mu$sec）、n 回がユニフィケーションを伴うリダクション（$R_n = 3.3\mu$sec）であった．そこでより精密化するためにリダクションコストを

$$R = R_d \times \frac{d}{n+d} + R_n \times \frac{n}{n+d}$$

とした．元のリダクションコストが相対的に小さいため、非決定的実行に対して積算されるリダクションコスト比は$\alpha_n = 1.5$ とした． 精密に求めるために決定的実行に対しても、$\alpha_d = 1.2$ を積算した．サスペンドコスト $S = 165\mu sec$、メッセージ送受信コスト $260\mu sec$

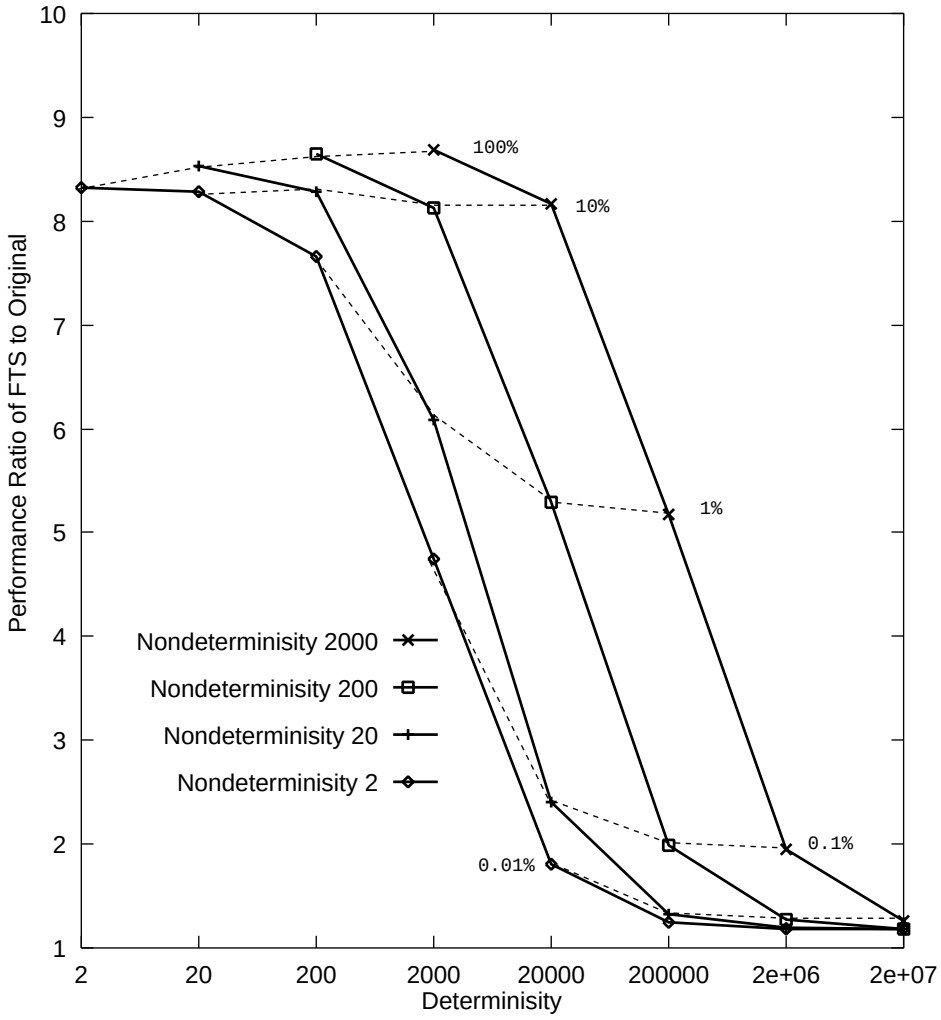

図 10.8: ベンチマークプログラム（非決定性/決定性比による処理性能変化）

とし、サスペンド数は第 9章で述べた $s = N, s' = 2 \times N$ とした．サスペンドと返信待ちが関連したものとして、サスペンド数を適当な係数倍したもの $W = s' \times 250$ とした．また総リダクション数が少ない場合には測定ルーチン他のプログラム起動時の初期実行オーバーヘッドが無視できないことから、実測により得た起動時間（元のプログラムで 2200μsec、耐故障プログラムで 7700μsec）を補正値として加味した．よってそれぞれの初期実行に I、I'かかるとすると、通常のプログラム実行は精密には次の式で表される．

$$\begin{aligned} TS_{normal} &= I + R \times (n + d) + S \times s \\ &= I + R_d \times d + R_n \times n + S \times s \end{aligned}$$

同様に耐故障実行は、次の式がより精密な値となる.

$$\begin{aligned} TS_{fts} &= I' + R \times (\alpha_n \times n + \alpha_d \times d) + S \times s' + 2 \times M \times n + W \\ &= I' + \alpha_n \times R_n \times n + \alpha_d \times R_d \times d + 2 \times M \times n + S \times s' + W \end{aligned}$$

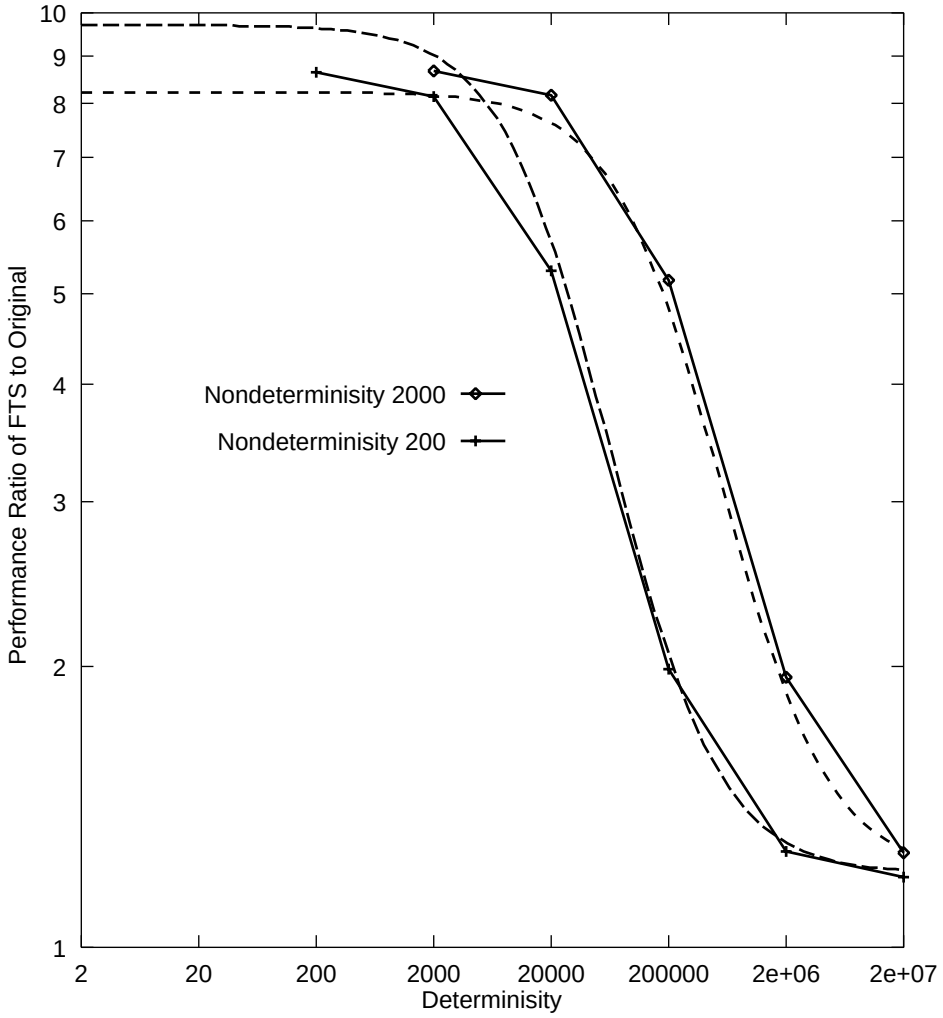

図 10.9: ベンチマークプログラム（見積りオーバーヘッドとの比較）

これらに各値を代入して次の値を得た.

$$\frac{TS_{fts}}{TS_{normal}} = \frac{77000 + (1.2 \times 1.6 \times d + 1.5 \times 3.3 \times n) + 2 \times n \times (260 + 165 + 250)}{2200 + 1.6 \times d + 3.3 \times n + 165 \times n}$$

得られた計算値からのグラフを重ねたところ、ある程度以上総実行数があれば非常によく

一致していることが見てとれる．ただし非決定性 200、決定性 2000 のように総実行数 2200 リダクション程度では、誤差が大きく現れたものと思われる．

図 10.10は、非決定性が 0.01%のプログラムを並列実行させた際の実行結果である．図 10.10の PE 数は処理プロセッサ数を指す．耐故障化プログラムは、ホストプロセッサ（PE0）と PSG、BSG を構成する処理プロセッサによって実行されるため、この PE 数は、PSG を構成するプロセッサ数に相当する．このプログラムは、PE 数に比例して処理量も増加させているため、元のプログラムの実行時間はほぼ一定となっている．また、同じプログラムを単に分散させているため並列化によって増えるコミュニケーションは 0 とみなせる．

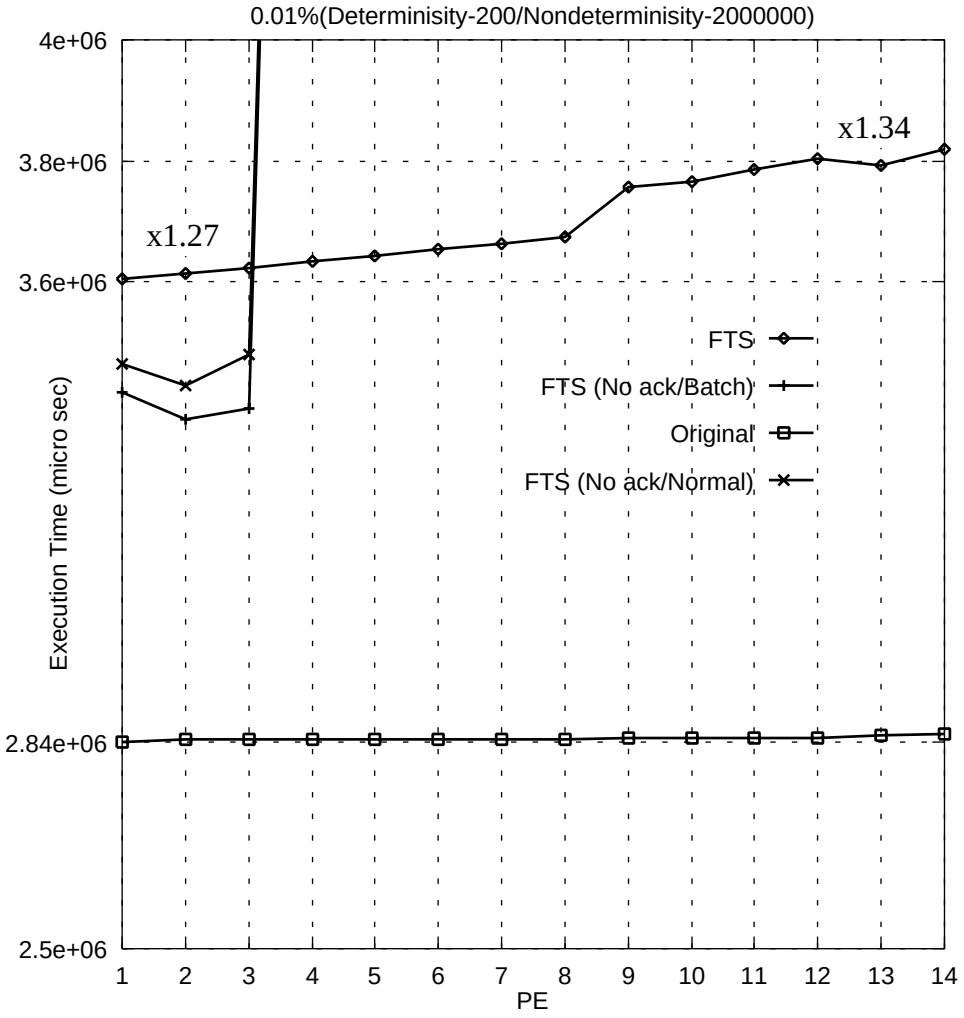

図 10.10: ベンチマークプログラム（並列実行における処理性能変化１）

耐故障化したプログラムは、多少の速度低下があるものの、PE 数の増大に対する速度低下は小さくなっている．この速度低下は、ゴールの転送処理が増大したことと、ゴールの転送にあわせて監視プロセスを起動していることによると思われる．

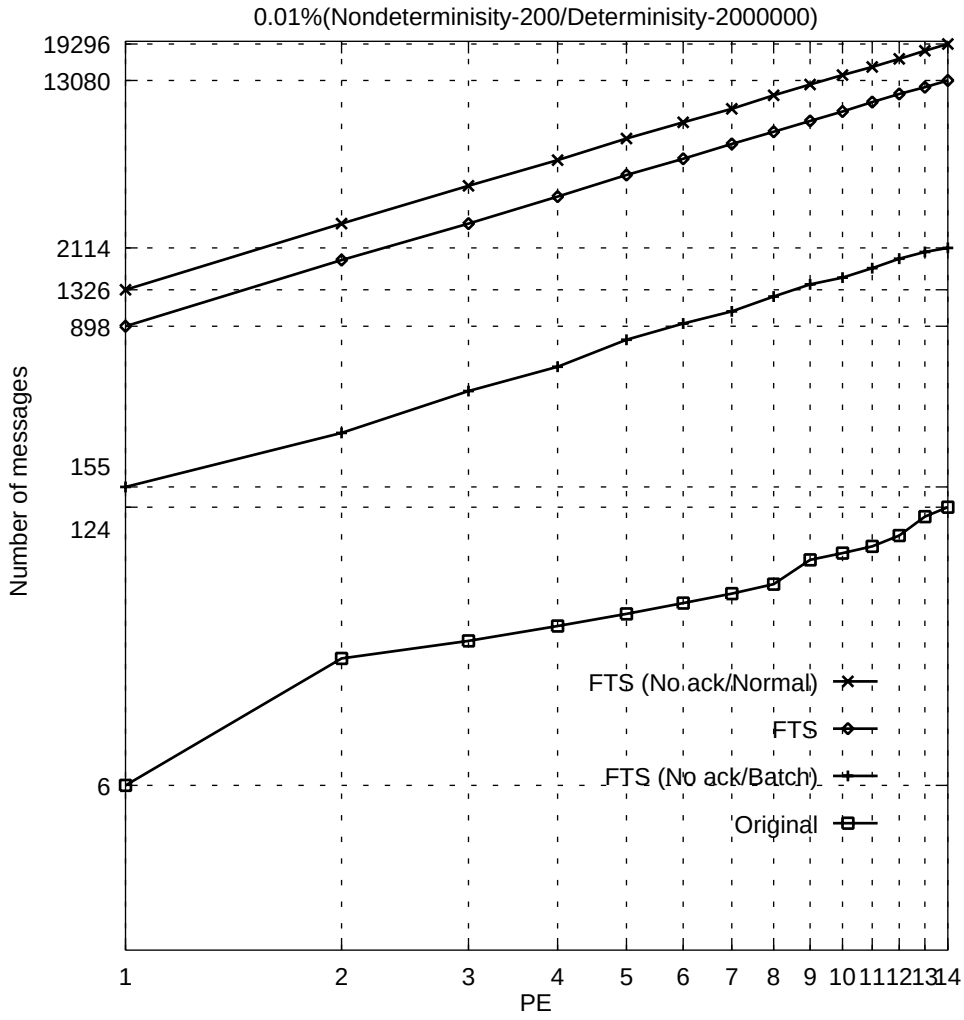

図 10.11: ベンチマークプログラム（並列実行時のシステム総メッセージ数）

測定値の増分から考えて、1 ゴールの転送におよそ 10msec かかっていることになる．ログの返信を返すことによる速度低下を見るために、ログに対して返信を返さないバージョンも動作させてみたが、負荷バランスが崩れてしまい 3PE 以上では性能低下がみられた．これは、PE1 からすべての処理を割り付ける前に、それ以前にゴール割り付けしたプロセッサでの処理が終了し、再割り当てが発生したことによる．比較できるのは PE3 までである

が、バッチモードであっても返信を返すことで増えた処理時間は6%程度と見られる.

図10.11は、このときのシステム全体での総メッセージ送受信処理回数を表す.

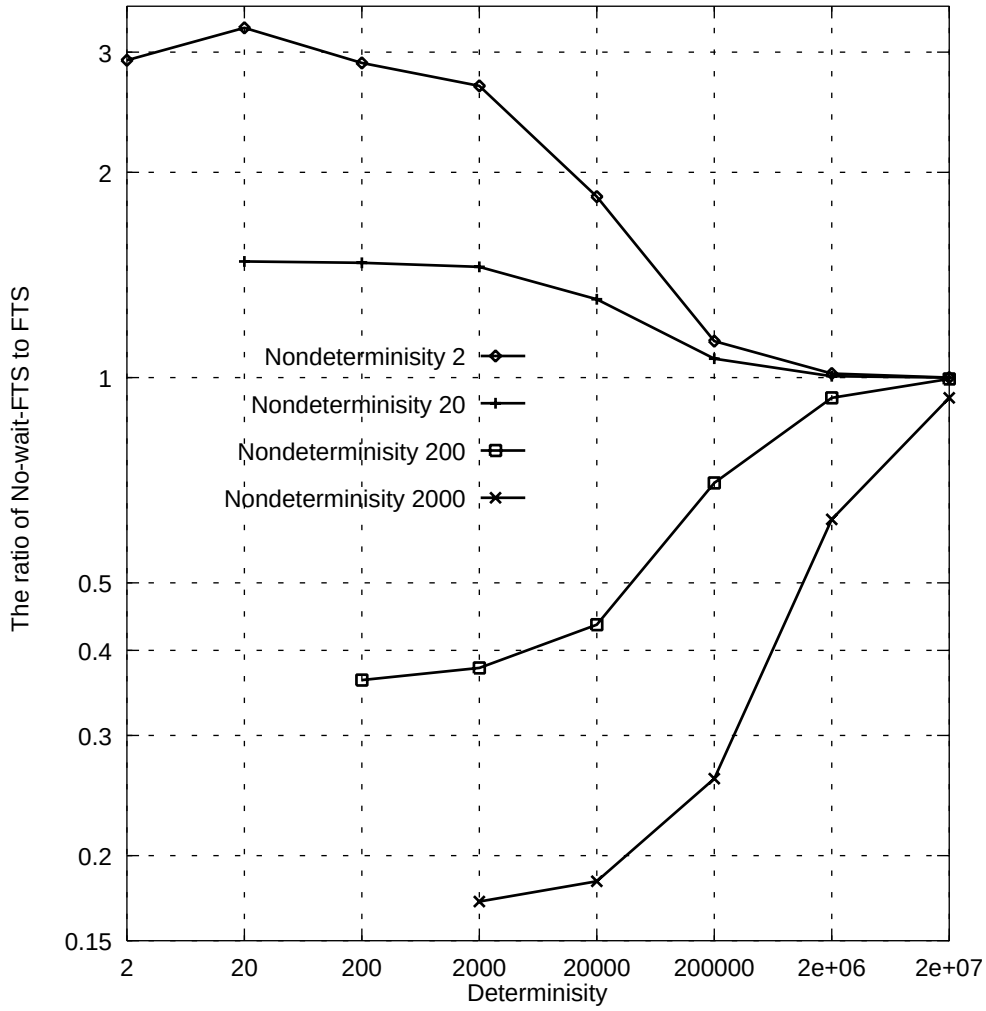

図 10.12: ベンチマークプログラム（ログ返信無し/返信あり処理時間比）

なお耐故障化プログラムでは、PSG、BSG 含めた数であるため、PE あたりでは表記の値の 1/2 である．元のプログラムではメッセージ処理数は、PE あたり高々10 メッセージ以下であるのに対して、耐故障化プログラムは PE あたり 450 メッセージほどになっている．ログと返信が非決定的実行数 200 に対してそれぞれあるため、ほぼこの数値に近い．返信を返さないバージョンの通常モードでは、ログの送受信に read、answer、release のそれぞれ 3 メッセージがあるため、非決定性 200 の 3 倍で 600 メッセージを処理している． これ

に対してバッチ転送モードでは、80 メッセージ程度で済んでいる．

図 10.12はログの返信無しの場合と、返信する場合の耐故障実行の処理時間比を示す． ログ無しの実行を見るとホストノードでのメッセージ送受信処理がわずかに増大しており、全体の実行数が小さい場合には通信処理の占める割合が大きいため性能低下を招いている．

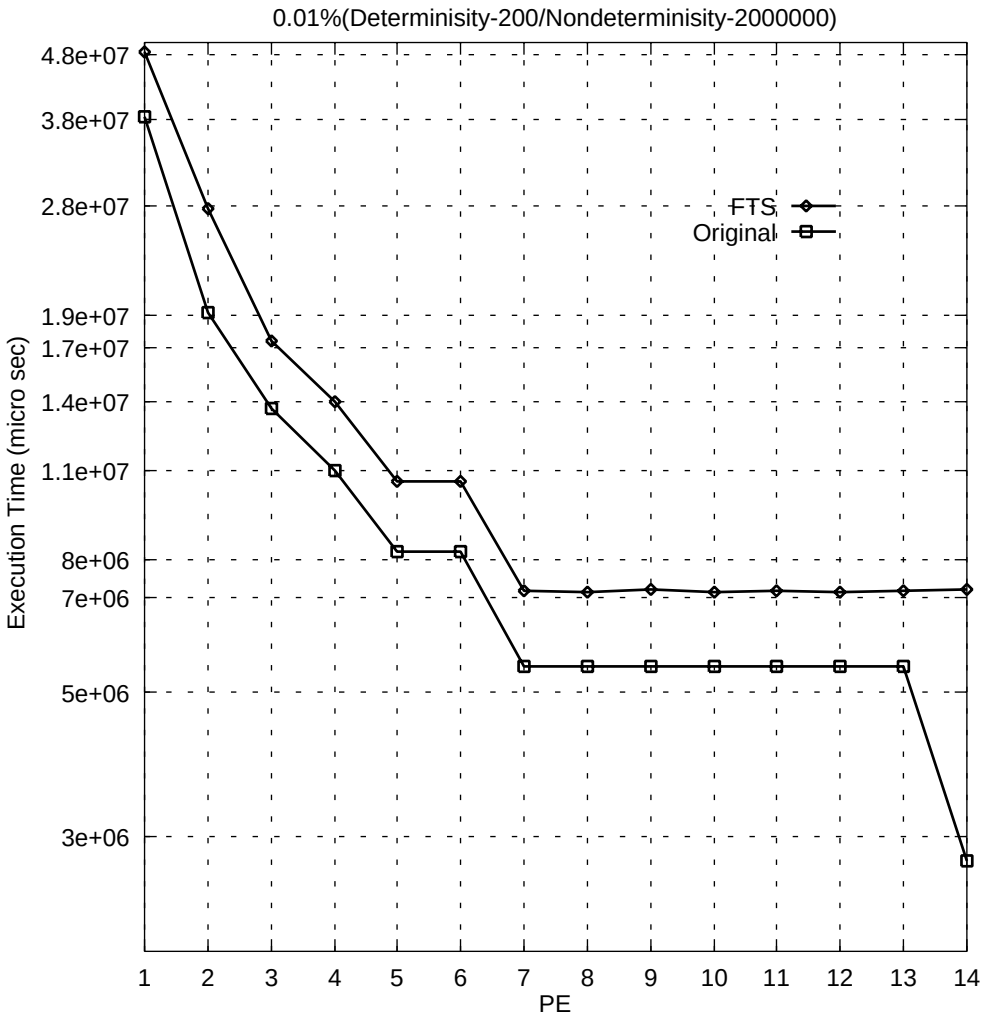

図 10.13: ベンチマークプログラム（並列実行における処理性能変化２ (1)）

図 10.10が、実行プロセッサ数に応じて処理負荷を増やしたのに対して、図 10.13は、プロセッサ数によらず負荷一定で実行した場合の結果である． 負荷は、それぞれ利用可能な処理プロセッサにできるだけ均等に分散するように設定した．図 10.13の PE 数は、図 10.10と同様に処理プロセッサ数を指す．元のプログラムと同じ処理プロセッサ数で比較するこ

とで、耐故障実行のオーバーヘッドを見ることができる．このプログラムでは並列効果のため PE 数に比例して処理時間が短くなっているのが見えるが、負荷の粒度が大きいため分散が均衡しない場合には、思ったほど性能が出ていない．

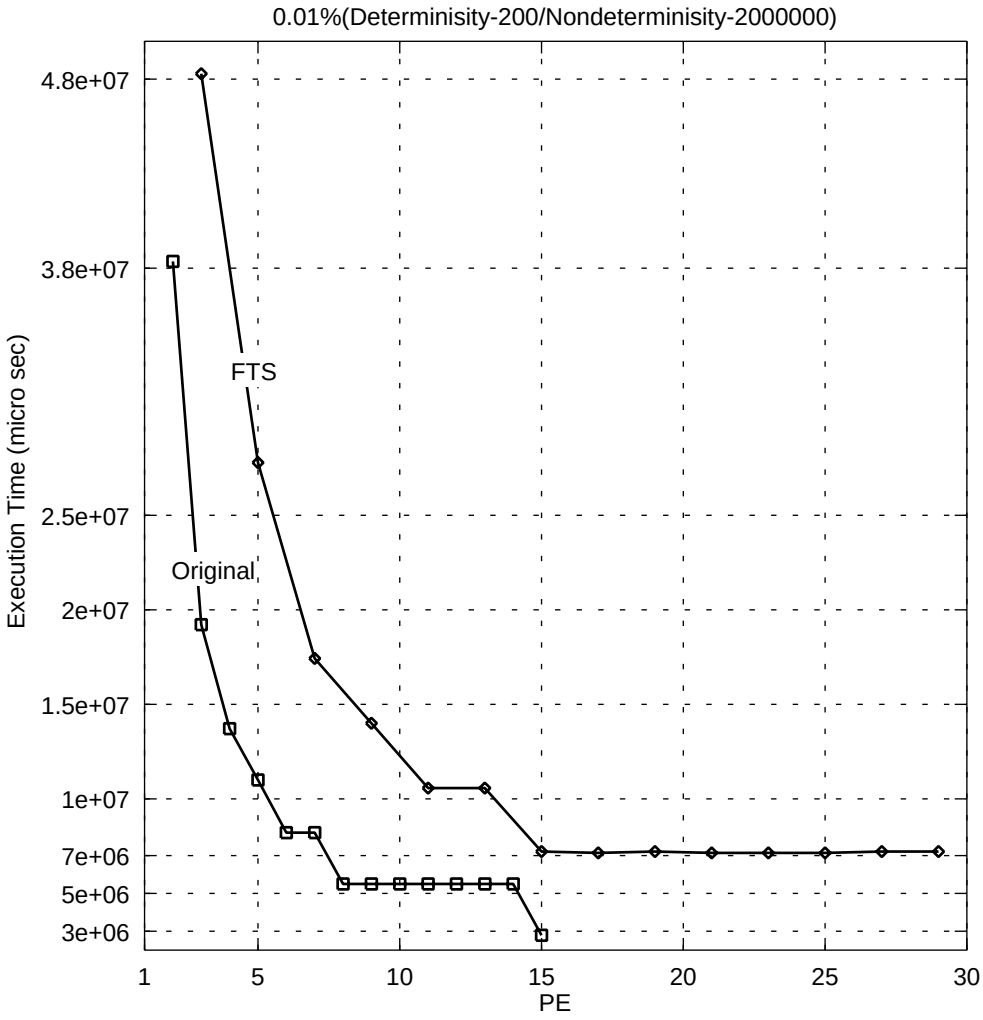

図 10.14: ベンチマークプログラム（並列実行における処理性能変化２ (2)）

同じ実行に対して、システムの総プロセッサ数を PE 数としてプロットしなおしたものが図 10.14である．こちらは、システム全体のプロセッサ数で比較することで、システムとしての実性能比較を見ることができる．このプログラムでの FTS は 1-Resilient システムであるため、実際に問題を解くのに有効な並列プロセッサ数はおよそ 1/2 となっており、実行時間も概ね 2 倍ほどの差となっている．しかし第 5章で述べたように MTTF を 3 倍に引

き延ばし、システム有効度を向上させている.

10.4.2 応用プログラム

8 Queen プログラムの実行結果を図 10.15、図 10.16、図 10.17に示す. PE 数は各サーバに用いた PE 数である. 従って、Original ではクライアントのプロセッサを除く PE 数、FTP ではサイトを構成する PE 数に等しい.

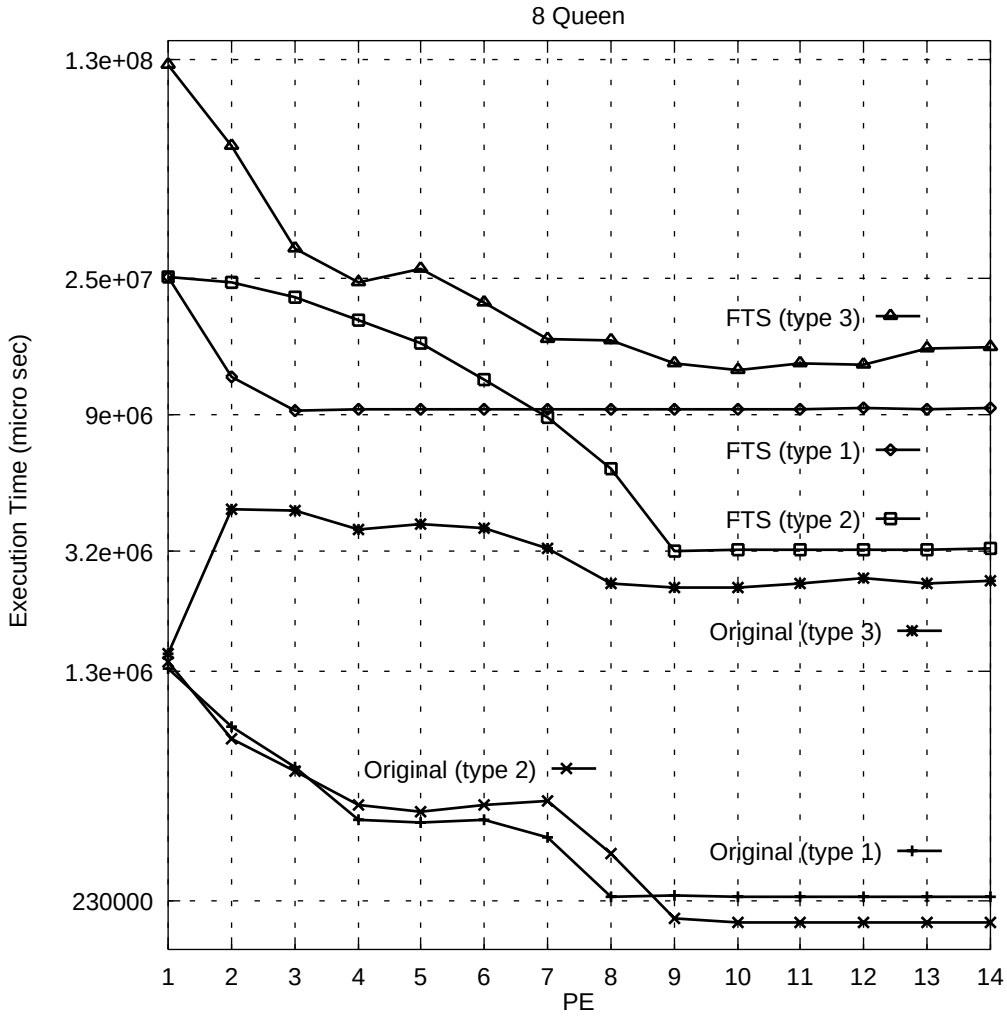

図 10.15: 8 Queen プログラム（並列実行における処理性能変化）

図 10.15は、各プログラムの並列実行結果を示す. Original の type-1 と type-2 は、予想した通り並列効果が現れているが、分散するプロセスが違うことからわずかに性能に差が

現れている．type-3 は通信量が多いことから、やはり並列化による効果が得られなかった．これらに対して FTS は、差があるがどれも並列効果が現れている．Original で近い動作の type-1 と type-2 も、FTS では差がでてきている．

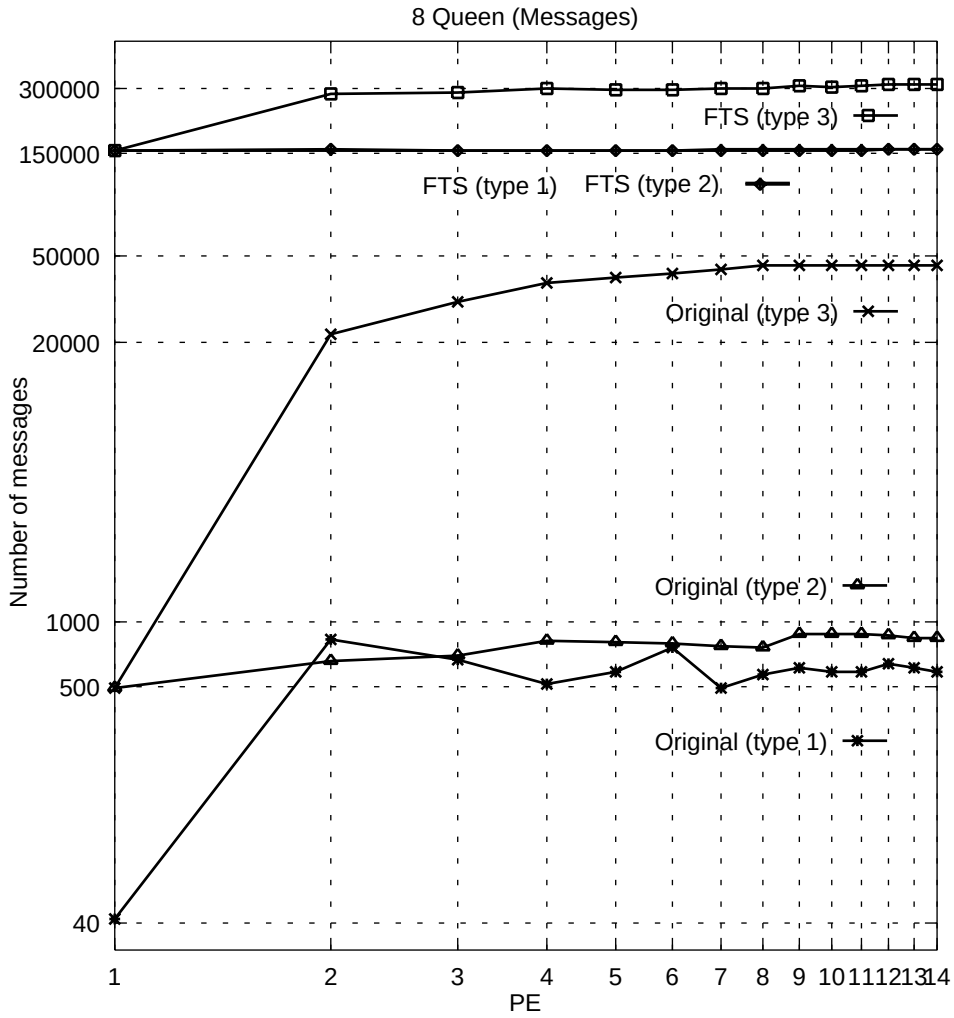

図 10.16: 8 Queen プログラム（並列実行におけるメッセージ数）

図 10.16は、それぞれの実行時のメッセージ数を示す．Original の 1PE の実行で、type-1 から type-2、type-3 のメッセージ数が増えているのは、ホストプロセッサとの通信である．type-1 はバッチ転送モードが効き易いように、解の生成が行われたと思われる．Original の type-1、type-2 とも通信量が少なく負荷バランスしやすいことから、十分に並列効果が現れている．type-3 は、通信量が多く台数効果が得にくくなっている．FTS(type-2) は負

荷分散が均等化し易く、うまく台数効果が得られている．

第 9章での考察から、実験結果を元に各値を算出してみたところ、表 10.2のような値となった．ORと FRは、それぞれ Original プログラムと耐故障プログラムの 1PE での実行時間に相当する．

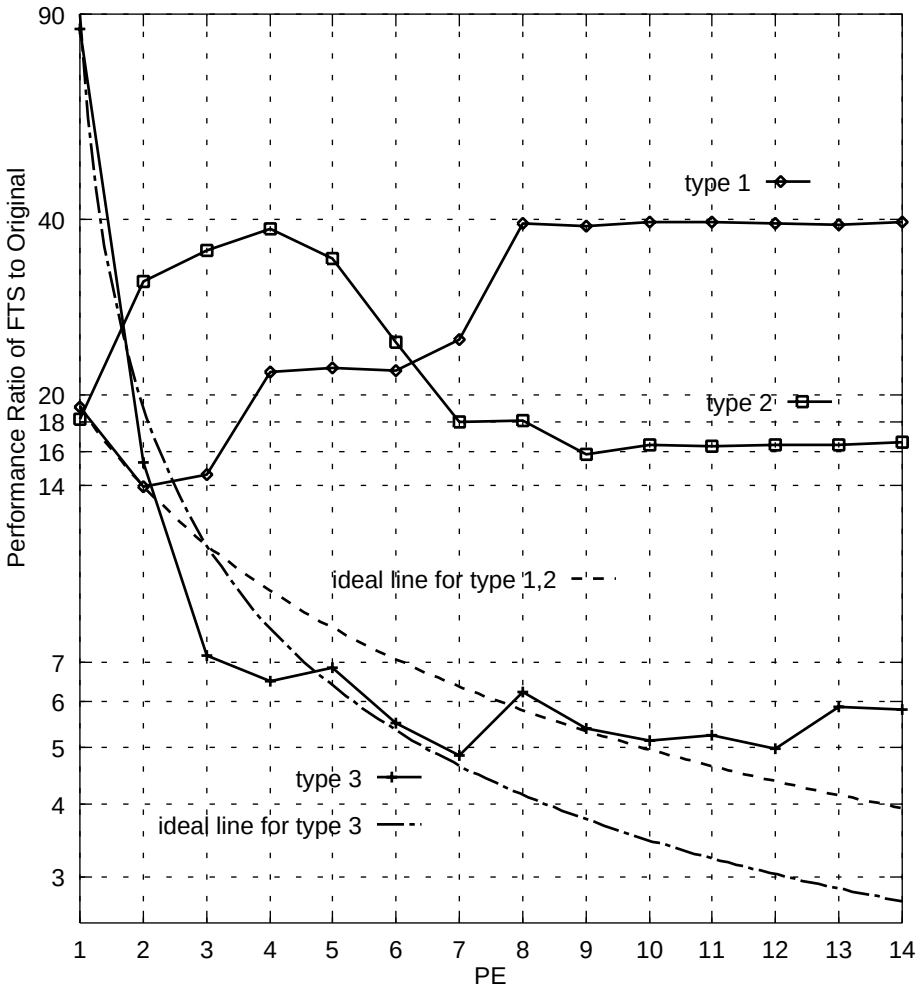

図 10.17: 8 Queen プログラム（並列実行における処理時間比）

メッセージ処理時間は、ベンチマークと同じく $M = 260\mu sec$ とした．m は 2PE でのメッセージ数から 1PE でのメッセージ数を減算したものであり、PE1 でのメッセージ数はバッチ転送が効かない場合をとった．これによると、type-3 は$\beta\gamma > 1$ であり並列実行に適さないことが分かる．type-1,type2 について、第 9章の考察を元にして予測処理時間比を算出す

表 10.2: 応用プログラムの 並列因子

	OR	FR	$\alpha = FR/OR$	m	$\beta\gamma = mM/OR$
type-1,type-2	1.3×10^6	2.5×10^7	19.2	2000	0.4
type-3	同上	1.3×10^8	10^2	19500	3.9

ると、PE 台数 Nに対して

$$\frac{19.2 + 0.4(N-1)}{1 + 0.4(N-1)} \quad (N \leq 1)$$

となる．この予測値および、実測による処理時間比を示したものが図 10.17である．比較的うまく負荷分散した type-2 の 8PE 以下であっても、予測と大きくはずれている．耐故障実行では、プロセッサ間のゴール転送処理が大きくなったため、ゴール転送が多いと予測とずれる可能性があるが、type-2、type-1 ともゴール転送は高々8 回でしかない． 一方、ゴール転送の多い type-3 もプロットしてみると、予測とほぼ一致していることがわかる．よってこの差は、負荷分散方式の相違が大きく影響していると思われる．なお type-3 は、上で示したように並列プログラムとしては不適であり、性能向上が良く見えるのは元のプログラムに並列効果が出ないせいである．

10.5 実験結果に関する考察

ベンチマークプログラムによる実験結果から、単体プロセッサでの耐故障プログラムの実行については、予測性能とかなり近い値を得た．基本性能の測定結果から、メッセージ処理とリダクション処理時間比 $M/R = 80$〜160 であり、非決定性が決定性の 0.01%以下でないと単体での性能は得難いことがわかる．

本研究で提案する手法は、メッセージ処理性能が落ちるほどユーザプログラムに占める非決定性の割合が性能に大きく影響する．Shen らによる Instant Replay の研究では、いくつかのベンチマークプログラムに対して静的解析を行った結果が発表されている [13]．これによれば、100 述語以下の小規模なプログラムではほとんど非決定性を持つ述語はなく、200 述語程度のプログラムで全体の 5〜6%という結果が報告されている．しかし、大規模なプログラムについての調査は行われておらず、また実際に非決定的部分の動作する割合についても分かっていない．

しかしプログラムの並列度が高ければ、耐故障化によって単一 PE サイトで 10 倍になったとしても、うまく負荷分散さえすれば並列実行によって大きく軽減されることも第 9章で示した．これは、元のプログラムが十分に並列性が高く、メッセージ処理がある程度ある場合で、かつ並列効果が得られることを仮定している．しかし負荷一定でプロセッサ数を増やした図 10.13の実験結果を見ても、それほど速度向上していない．これは、ベンチマークプログラムのように、並列性がかなり高いプログラムでもコミュニケーションが少ないためであろうと考えられる．図 10.10のように、プロセッサ数に応じて処理負荷を増やした場合でも、一定の時間がほぼ保存されるだけで比率は縮まらない．

8 Queen プログラムは、純粋な非決定的実行部分は少ないが、決定的実行であっても非決定的実行を含む場合ログを発生させるため、実質的なログ量の全体に占める割合は大きい．実測によると 45121 リダクションのうち、33069 リダクションでログを発生する（73%）．このため単一プロセッサでの耐故障化のオーバーヘッドにより処理時間比で 20 倍近くになっている．理論的には 8PE のサイト構成では、6 倍程度まで落ちるものと期待されるが負荷分散方式の相違からよい比較とはならなかった．

上記ベンチマークプログラムを対象にして、プログラム実行中にプライマリサイトの 1プロセッサを強制的に終了させるプログラムを用意し、実験しによりバックアップが実行を引き継ぐことも確認した．また 8 Queen プログラム測定中に、メモリ不足を起こして処理系で例外を起こした際にも解を得ることができた．

現在の実装では、監視プロセッサのリング結合ネットワークを利用して単純に隣接プロセッサへ負荷を分散させているため、元のプログラムの負荷分散方式のように均等に広がらなかった．ユーザの負荷分散方式を反映できるような実装の改良は、今後の課題である．

第 11 章

今後の課題

実験評価を通じて、以下が今後の課題として残った.

11.1 負荷分散方式について

監視プロセスのリングネットワークによる負荷分散では、中心となる監視プロセス一つが、アイドルプロセッサを集中管理しているため、そこから大きな処理を分散させる場合には均一になりやすいが、その他のプロセッサでゴールを分散させる場合には偏りやすく、思ったように性能が得られない場合が見られた.

もともとのユーザの負荷分散アルゴリズムを反映できるようにするには、各監視プロセスが、サイト内の各プロセッサへのストリームをそれぞれ持ち、物理プロセッサ指定したを、サイト内論理プロセッサアドレスにマッピングして対応するプロセッサへのストリームをつうじてゴールを送らねばならない. このためには次のような問題がある.

- 各プロセッサへのストリームを一括して管理する際、論理プログラミングの枠組では効率を落し易い.

  配列構造などでストリームを管理すると、要素としてのストリーム一つに値を送ると、配列構造全体を複製することになる. KL1 専用マシン PIM で実装されていた KL1 処理系では、マルチバージョンアレイが実装に使われており、配列全体を物理的にコピーせずに論理プログラミングとして整合性を保った利用ができた. KLIC でも同様の機能を用いて実装しないとメモリ消費が激しくなり、GC が頻発することになる.

- 各プロセッサからのストリームからの受信処理を一本にするために、merge 処理が増大する.

  組み込み述語 merge が提供されているが、merge 処理のコストが高いことに変わりはない. 並列言語 Occam でも複数ストリームからの入力処理を行う ALT は非常にコストが高く、並列言語一般に効率的なプログラミングは難しいものと思われる.

KLIC では複数プロセスから一つのプロセスに向かうコミュニケーションについては、組み込み merge を提供することで、ある程度の効率的なサポートを実現している. また逆に一つのプロセスから複数プロセスへのブロードキャストは、同じ共有変数を用いることで実現できる. しかし複数プロセス間のコミュニケーションを行いたいと考えた場合には、プロセス間を直接結合するストリームを張るか、間にルーティングを行うネットワークをプロセスで構成するしかない.

プロセス間に直接つながるストリームを張るには、上のように配列構造のコピーが頻発してしまい非効率である. 従って、プロセスを節点として用いたルーティングネットワークが実現可能性がある. 今回の実装では単純なリングネットワークを構成し、節点プロセスを各物理プロセッサ上に置いた. これは、負荷の分散は論理的な隣接点にのみ送ること想定したためであった. しかし、隣接点以外にも分散させたいと思えば、各要素プロセッサ上にルーティング節点となるプロセスを置くのは、nCUBE2 のよう物理的なネットワークにルーティング機構が備わったシステムでは明らかに非効率である. プロセスを節点として、適当なトポロジーの仮想的なネットワークを各プロセッサ上に構成し、節点プロセスを通じて物理ネットワーク間を分散させることで、効率的なルーティングシステムが構築できるものと思われる.

11.2 プログラム変換について

Instant Replay で提案された方法は、非決定性を含む述語でもログが発生する. 非決定的実行を含む決定的な実行もまた非決定的だとみなされるためである. 実際上はログを送るストリームを分岐させる必要性のためでもある. 従って本方式にとって有利なプログラミングスタイルとしては、非決定的な実行が論理的な構造の上位レベルにあるものと言える. このようなスタイルへの自動変換の可能性の検討も、今後の課題である.

11.3 非決定性の高いプログラムについての効率化

本研究で提案した耐故障プログラムが効率良く動作するためには、単体プロセッサでは、プロセッサ上で行うコミュニケーション処理性能がプロセッサ内の単位処理性能よりも小さい方が有理になる．コミュニケーション処理性能が大きい場合であっても、元のプログラムが並列実行に相応しい性質を備えているならば、並列実行によって理論上は効率化できることを示したが、それでも単体での実行時間のプロセッサ台数分の 1 になるに過ぎない．本実験で用いたシステムは、メッセージ処理にリダクション処理の最大で 80 倍以上かかるものであり、単体プロセッサでの性能はプログラム自身の非決定性の比率に頼るものであった．

非決定性がプロセッサ間のコミュニケーション回数を増やすことが、性能低下の主な原因である．しかし nCUBE2 上の KLIC 処理系では、バッチ転送モードのようにまとめてメッセージを転送することで効率化を図ることができる．従って非決定的実行に関するログ、返信についてパックして転送することができれば、非決定性が高くても、効率化の可能性がある．一つのプロセスについてパックすることは信頼性を損なうことになるので、互いに関係しないプロセスのログをパックすることで、効率化を図れるものと思われる． このようなログのパック転送についても、今後の課題である．

11.4 再構成について

本研究で提案する耐故障実行方式は、現在の実装ではプライマリサイトを構成するプロセッサが 1 台故障しても、サイトを構成する他のプロセッサも放棄したままにしている．これら残りのプロセッサを有効利用するために、再構成の方法についても考察する必要がある．再構成には、次の二つの可能性がある．

1. 動作中のプライマリサイト、バックアップサイトに組み入れる．

2. 一つのバックアップサイトとして構成し直す．

信頼性を向上させるには、2の方が望ましい．実現可能性は 1が優勢であり、現在の実装にも容易に組み込める．しかし、第 5章で述べたように、この再構成手法に有意性はほとんど見られない．2の実現の困難さは、実行途中のプログラム状態を再現することにある．

第 12 章

結論

本研究は、高価な専用の耐故障計算機ではなく、一般の商用並列計算機向けにソフトウェアによる耐故障を提供することを目的としている．耐故障計算機は高価であるだけでなく、ユーザに対して専用のソフトウェアを要求することから、ソフトウェアの開発コストも必要になる．これに対して、特別なハードウェアを期待せずに、ソフトウェア自身でハードウェアや OS の故障に耐える耐故障ソフトウェアの研究が近年行われつつある．耐故障化は、ハードウェアの並列性を利用するが、それぞれのパラダイムが適用対象とするソフトウェアは決定的で逐次のものである．本研究では、非決定的なプロセスに対する耐故障実行パラダイムのアイデアを示し、並列ソフトウェアを並列のまま動作させることを考慮した耐故障手法を提供した．さらに提案したパラダイムに基づいて、動的プロセスから構成される並列プログラミングに対して、耐故障プログラムに変換、実行する手法を提案した．これは複製を多重に動作させるプライマリサイト・アプローチの一種であるが、複数 PE 上で動作する複数プロセスによってサイトを構成し、非同期通信で渡すログをもとにしてバックアップ実行をさせる点が特徴である．ログは、並列論理プログラムにおける非決定的選択の部分でのみ生成される．この方式を形式化しその正当性について示した．

提案した手法で構成した耐故障並列プログラムの信頼性についても考察した．提案した手法は非修理系の冗長系を構成し、サイト数の log オーダーで MTTF を引き延ばすが、サイト分割を多くするほど並列実行に利用できるプロセッサ数を減らし、並列実行能力を落す．超並列計算機は、高性能を求めて開発されているが、MTTF まで考慮すれば処理容量は要素プロセッサ 1 台と変わらない．並列実行させるためのオーバーヘッドが加わることを考えれば、むしろ処理容量は少なくなったと見るべきであろう．言い換えるならば、超

並列計算機はMTTF を圧縮することと引き替えに高性能を得ており、長時間の運用には適していない．これに対して本研究の成果は、多少性能を落しても連続運用時間を引き延ばしたいという選択を可能にする．例えばプログラムの必要とする並列性がプロセッサ台数に比べて平均的に少ないならば、そのハードウェア冗長性を耐故障に振り向けてMTTF を増大させる選択がとれるようにできる．

専用の耐故障計算機は、高価である上に専用のハードウェアやOS を備えたものがほとんどであり、従って専用のソフトウェアを用意しなければならない．また、いくつかの耐故障ソフトウェアのパラダイムも紹介されてきているが、そのパラダイムに沿ったプログラミングはユーザに任されている点で、ユーザにとっては耐故障計算機の状況とあまり変わっていないと思われる．耐故障プログラム開発を容易にする環境についての研究も行われているが、耐故障を意識することなく書かれたプログラムが耐故障実行できるような枠組や、既存プログラムの耐故障化が容易にできることが、今後大いに望まれる．並列論理型言語のプログラミングスタイルと、これを利用した変換は、耐故障並列プログラムを容易に提供できることを示せた．ユーザプログラムの耐故障化変換は、人手に頼った変換ではプログラマへの大きな負担となるばかりでなく、複雑なプログラミングの必要からバグも混入し易い． 本研究の耐故障化変換は自動変換を目標としており、プログラム例をもとにその変換方法を示した．変換後の耐故障プログラムは、ソースレベルでは2 倍以上、オブジェクトコードとしては3.5 倍ほどになるが、実験で用いたプログラムのように小規模なものであれば、実行形式の全体サイズからすればわずかであることから、最終的な実行形式では6%程度の増加にすぎない．

耐故障化したプログラムは、元の処理の他に耐故障のための処理が追加されたため、通常の処理よりも遅くなり得るが、この速度低下の割合を少なくすることが実用性を左右する．プログラムの実行オーバーヘッドを静的に解析したところ、サイトを構成するプロセッサを1 台とした場合、プロセッサ間のコミュニケーション処理オーバーヘッドとプログラム中の非決定的実行部分の占める割合に大きく依存することが分かる．これは実験結果からも裏付けられた．実験に用いたnCUBE2 上の処理系では、プロセッサ間コミュニケーションオーバーヘッドが大きく、プログラム中の非決定的実行部分の占有率がそのまま実行時間に反映した．サイトを構成するプロセッサ数を複数にして並列実行させることで、耐故障化の実行オーバーヘッドを減少させ得るが、このためには元のプログラムの持つ並列性がかなり高く、かつその並列性を十分に活用して負荷分散できることが必要である． この

傾向はプロセッサ間コミュニケーション処理性能が高くなるほど顕著である．

このようにコミュニケーションオーバーヘッドの大きなシステムでは、非決定性の小さいアプリケーションプログラムでなければ実用性は低くなるが、コミュニケーションオーバーヘッドが小さいシステムでの実装であれば、実行オーバーヘッドは少なくて済み、無故障時の使用にも十分実用的である．言い換えるなら、ユーザプログラムに許容される非決定性の割合は、システムのプロセッサ間コミュニケーション処理性能によって変化する．システムのコミュニケーション性能が十分でなければ、耐故障化することで、プログラムの処理オーバーヘッドは増大する．また元のプログラムの並列性が十分あるならば、並列実行によって耐故障化のオーバヘッドを減少させられることも理論的に示した．

さらに実行オーバーヘッドの解析結果から、基礎機能の性能が分かれば、理論的なオーバーヘッド算出式にあてはめて、実際の実行効率と許容される非決定性の割合が算出できることを示した。この算出性能は、プライマリ、バックアップともに 1PE 構成での実際の処理性能と良く適合することも実験を通じて示せた．

決定的なプログラムは、単一プロセッサ環境で直線的に動作させる方が高速な実行が可能であり、並列性は乏しいものと思われる．もともと並列プログラミングにおける非決定性は、プログラムに並列性を許したことから生じている．よって非決定性によって耐故障化のオーバーヘッドがある程度高くなったとしても、並列化によって高速実行できる可能性もある．本研究で提案する耐故障化方式の実行効率は、このように方式自身の実装方法の他にユーザプログラムの性質と、実装するシステムの性質に依存している．

計算機に対するユーザの要求は、高速な実行と大容量の仕事の実行という二面性を常に持つ． 高速性を追求して発展してきた並列計算機であっても、それは変わることがない．並列計算機が利用可能になるにつれて、アプリケーションプログラムは巨大化し、その要求する仕事量も増大してゆくことは間違いない．しかし耐故障性を導入しない限り、MTTF から見た並列システムの仕事量容量は要素プロセッサ 1 台の仕事容量で抑えられる．本研究の成果は、高速、大容量というユーザの要求に対して、ある程度の調整を並列システムに行う自由度を与える．すなわち最大性能を落しても、MTTF を伸ばすことで仕事容量を増大させることをユーザに許す． 実装したシステムは、まだまだ改良の余地があり、性能改善の課題も残されているが、本研究は並列システムの将来の一つの可能性を示せたものと信ずる．

謝辞

本研究を行なうに当たり, 終始御指導を賜わった横田治夫助教授に深謝致します. また、日頃より貴重なご指導、ご示唆を賜わりました日比野 靖教授に最大の尊敬をもって感謝の意を表します. 片山 卓也 教授、堀口 進 教授には、本論文をまとめるにあたりご指導を頂きました. 日頃から有益な御助言をいただきました東京大学工学部電子工学科の近山 隆 教授に深く感謝致します.

Instant Replay 変換を行う Prolog ソースコードを提供していただきましたマンチェスター大学の Kish Shen 博士に感謝致します。

また（財）日本情報処理開発協会 先端情報技術研究所の KLIC タスクグループの皆様には、本研究に関して討論および貴重なご意見を頂きました.

本論文をまとめるに当たって御協力いただいた研究室の諸兄にも厚く御礼申し上げます.

最後に、すでに解散した ICOT（(財) 新世代コンピュータ技術開発機構）と、その第五世代コンピュータ・プロジェクト関係各企業の諸先輩、諸兄へ深く感謝いたします. 現在ここにあり、本研究へ至ったすべての道の始まりはそこにあり、また彼らの存在が公私ともに筆者の励みでした.

参考文献

[1] Schlichting,R.D., "An Introduction to Fault-Tolerant Software", from *Invited Talk of the International Workshop on Fault Tolerance in Information Systems (FIS95)*, 1995.

[2] 岸本 光弘 他、"高信頼 UNIX「風雅」"、第 52 回情報処理学会全国大会論文集、平成 8 年前期、6M1–6（p. 4-61 – 4-72）、1996.

[3] Maier,J.,"Fault-Tolerant Parallel Programming with Atomic Actions", in *Proc. of the International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN)*, pp. 210-219, 1995.

[4] Sharma,D.D. and Pradhan, D.K., "An Efficient Coordinated Checkpointing Scheme for Multicomputers", in *Proc. of ISPAN*, pp. 36-42, 1995.

[5] Johnson,D.B.,' "Efficient Transparent Optimistic Rollback Recovery for Distributed Application Programs", in *Research Report of Carnegie Mellon University (CMU-CS-93-127)* also in *Proc. of the 12th Symposium on Reliable Distributed Systems*, 1993.

[6] Suzuki,M., Katayama,T. and Schlichting,R.D., "Implementing Fault Tolerance with Attribute and Functional Based Model", in *Digest of papers of the Twenty-Fourth Annual International Symposium on Fault-Tolerant Computing (FTCS-24)*, pages 244-253,1994.

[7] Ueda,K. and Chikayama,T.,"Design of the Kernel Language for the Parallel Inference Machine", *The Computer Journal*, 33(6):494-500,1990.

[8] Hasegawa,R. and Fujita,M. "Parallel Theorem Provers and Their Applications", in *Proc. of FGCS'92*, pages 132-154, 1992.

[9] Ueda,K., “Guarded Horn Clauses”, *Logic Programming ’85*, ed. Eiichi Wada, pp. 168-179, Springer-Verlag Lecture Note in Computer Science No.221, 1986

[10] Chikayama,T., Fujise,T., and Sekita,D., “A Portable and Efficient Implementation of KL1”, in *Proc. of PLILP’94*, Springer-Verlag, pages 25–39,1994.

[11] Rokusawa,K., Nakase,A., and Chikayama,T., “Distributed Memory Implementation of KLIC”, *NGC, Vol.14, No.3*, Ohmsha, Ltd. and Springer-Verlag, pages 261–280,1996.

[12] LeBlanc,T.J., Mellor-Crummey,J.M., “Debugging Parallel Programs with Instant Replay”, *IEEE Transactions on Computers C-36*,4, pp. 471-481, 1987.

[13] Shen,K. and Gregory,S., “Instant Replay Debugging of Concurrent Logic Programmings”, New Generation Computing 14, 1. 1995.

[14] Jalote,P., *“Fault Tolerance in Distributed Systems”*, Prentice Hall, 1994.

[15] Ueda,K. et al. “Mode Analysis of GHC Programs”, in *Proc. of the 40th national meeting of the information processing society of Japan* , 1990.

[16] Pradhan, Dhiraj K., “Fault-Tolerant Computer System Design”, Prentice Hall, 1995.

[17] 当麻 喜弘 編著、「フォールトトレラントシステム論」、（社）電子情報通信学会、pp 185-200、1990.

[18] 野中 保雄、島岡 淳、「冗長系 – 理論と実際 –」、日科技連、1990.

[19] Hennessy,J.H., Patterson,D.A.、「コンピュータ・アーキテクチャ」、日経 BP 社、1992.

[20] Kai Hwang, “Advanced Computer Architecture”, McGraw-Hill, 1993.

本研究に関する発表論文

[1] 杉野栄二、横田治夫、「並列論理型言語による疎結合並列計算機向け耐故障化プログラム変換」、情報処理学会論文誌「並列処理」特集号、1997.1 照会後採録.

[2] 杉野栄二、横田治夫、「並列論理型言語におけるフォールトトレランスソフトウェアの構成」, 並列処理シンポジウム JSPP'96 論文集、1996.6、p.211-218.

[3] Eiji Sugino and Haruo Yokota, "An Implementation of Logical Variable Reference in a Concurrent Logic Programming Language by using Channel Variables", in *Proc. of the 6th Transputer/Occam Intn'l Conf.*, IOS Press, 1994.6.

[4] 杉野栄二、古市昌一、稲村雄、瀧和男、「マルチ PSI におけるデバッグ機能・メンテナンス機能」、並列処理シンポジウム JSPP'89 論文集、1989.2、p.275

[5] 杉野栄二、古市昌一、井上久美子、宮崎敏彦、瀧和男、「マルチ PSI における処理系動特性の評価」、情報処理学会第３４回全国大会予稿集、昭和 62 年前期、4P-8 (p.159)

[6] 杉野栄二、古市昌一、石塚裕一、宮崎敏彦、瀧和男、「並列論理型プログラムの特性解析（１）- 動特性解析ツール -」、情報処理学会第３５回全国大会予稿集、昭和 62 年後期、5Q-9

[7] Haruo Yokota and Eiji Sugino, "Fault Tolerance in Parallel Logic Programming Languages", International Workshop on Fault Tolerant Information Systems, Tokyo, 1995.11.

[8] 佐藤 裕幸、近山 隆、杉野 栄二、瀧 和男、「PIMOS の概要 ～並列推論マシン用オペレーティング・システムの構築～」、情報処理学会第３４回全国大会予稿集、昭和 62 年前期、2P-8

[9] 市吉 伸行、六沢 一昭、近山 隆、中島 克人、宮崎 敏彦、杉野 栄二、「並列処理における重みつき参照カウントを用いた実時間 GC」、情報処理学会第３６回全国大会予稿集、昭和 63 年前期、7H-3

[10] Kanae Masuda, Hirokazu Ishizuka, Hiroaki Iwayama, Kazuo Taki and Eiji Sugino, "Preliminary Evaluation of the connection network for the Multi-PSI system", in *Proc. of the 8th European Conference on Artificial Intelligence*, (p.18-23),1988.8.,

[11] 益田 嘉直、石塚 裕一、岩山 洋明、瀧 和男、杉野 栄二、「マルチ PSI における接続ネットワークの試作と評価」、情報処理学会論文誌 第 29 巻第 10 号 (p.995-1003)、昭和 63 年 10 月

付録 A

耐故障ソフトウェアの信頼性 算出式

A.1 非再構成系の MTTF

$$
\begin{aligned}
MTTF_{system}(k) &= \int_0^\infty R_{system}(k,t)dt = \int_0^\infty e^{-n\lambda t}\sum_{i=0}^{k}(1-e^{-n\lambda t})^i dt \\
&= \frac{1}{n\lambda}\sum_{i=0}^{k}\left[\frac{1}{i+1}(1-e^{-n\lambda t})^{i+1}\right]_0^\infty = \frac{1}{n\lambda}\sum_{i=0}^{k}\frac{1}{i+1}
\end{aligned}
$$

上で、以下の結果を利用している.

$$
\begin{aligned}
e^{-n\lambda t}\frac{d}{dt} &= -n\lambda e^{-n\lambda t} \\
(1-e^{-n\lambda t})^i\frac{d}{dt} &= in\lambda e^{-n\lambda t}(1-e^{-n\lambda t})^{i-1}
\end{aligned}
$$

$$
\left(\begin{array}{ll}
1-e^{-n\lambda t}=x \text{ とおくと} & (1-e^{-n\lambda t})\dfrac{d}{dt} = x\dfrac{d}{dx}\dfrac{dx}{dt} \text{ より} \dfrac{dx}{dt} = n\lambda e^{-n\lambda t} \\
& (1-e^{-n\lambda t})^i\frac{d}{dt} = x^i\frac{d}{dx}\frac{dx}{dt} = i(1-e^{-n\lambda t})^{i-1}n\lambda e^{-n\lambda t}
\end{array}\right)
$$

$$
\int e^{-n\lambda t}(1-e^{-n\lambda t})^i dt = \frac{1}{(i+1)n\lambda}(1-e^{-n\lambda t})^{i+1}
$$

A.2 再構成系の信頼度

A.2.1 マルコフモデルにおける状態 S_i にある確率 $P(S_i, k)$

マルコフモデルから、次のように微分方程式が導かれる.

$$\begin{aligned}
P(S_K, t+\Delta t) &= (1-(K+1)n\lambda\Delta t)P(S_K, t) \\
&\cdots \qquad \cdots \\
P(S_{k-1}, t+\Delta t) &= (k+1)N(k)\lambda\Delta t P(S_k, t) + (1-kN(K-1)\lambda\Delta t)P(S_{k-1}, t) \\
&\cdots \qquad \cdots \\
(P(S_{-1}, t+\Delta t) &= N(0)\lambda\Delta t P(S_0, t))
\end{aligned}$$

よって$\lambda_k = (k+1)N(k)\lambda$として、次の微分方程式を得る.

$$\begin{cases}
\dfrac{dP(S_K, t)}{dt} = -\lambda_K P(S_K, t) \\
\dfrac{dP(S_{k-1}, t)}{dt} = \lambda_k P(S_k, t) - \lambda_{k-1} P(S_{k-1}, t) \quad (k = 1, \ldots, K)
\end{cases}$$

$P(S_K, 0) = 1, \forall i < K,\ P(S_i, 0) = 0$ とし、ラプラス変換をとると、

$$\begin{aligned}
t\mathcal{L}\{P(S_K, t)\} - P(S_K, 0) &= -\lambda_K \mathcal{L}\{P(S_K, t)\} \\
t\mathcal{L}\{P(S_{k-1}, t)\} - P(S_{k-1}, 0) &= \lambda_k \mathcal{L}\{P(S_k, t)\} - \lambda_{k-1}\mathcal{L}\{P(S_{k-1}, t)\}
\end{aligned}$$

よって、

$$\begin{aligned}
\mathcal{L}\{P(S_K, t)\} &= \frac{1}{t+\lambda_K} \\
\mathcal{L}\{P(S_{K-1}, t)\} &= \frac{\lambda_K}{t+\lambda_{K-1}}\mathcal{L}\{P(S_K, t)\} = \frac{\lambda_K}{(t+\lambda_K)(t+\lambda_{K-1})} \\
&\cdots \qquad \cdots \\
\mathcal{L}\{P(S_{k-1}, t)\} &= \frac{\lambda_k}{t+\lambda_{k-1}}\mathcal{L}\{P(S_k, t)\} = \frac{\lambda_K \cdots \lambda_k}{(t+\lambda_K)\cdots(t+\lambda_{k-1})}
\end{aligned}$$

ここで、重畳に関して以下が成り立つので、

$$\begin{aligned}
e^{-\lambda_A t} * e^{-\lambda_B t} &= \int_0^t e^{-\lambda_A u} e^{-\lambda_B (t-u)} du = \int_0^t e^{-(\lambda_A - \lambda_B)u - \lambda_B t} du \\
&= \left[-\frac{1}{\lambda_A - \lambda_B} e^{-(\lambda_A - \lambda_B)u - \lambda_B t} \right]_0^t
\end{aligned}$$

$$
\begin{aligned}
&= -\frac{1}{\lambda_A-\lambda_B}\left(e^{-\lambda_A t}-e^{-\lambda_B t}\right)\\
&= -\frac{1}{(A-B)\lambda}\left(e^{-\lambda_A t}-e^{-\lambda_B t}\right) \qquad (\lambda_A-\lambda_B=(A-B)\lambda \text{より})\\
&(-(k+1)\lambda)\sum_{i=0}^{k}(-1)^i{}_kC_i e^{-\lambda_{K-k-1}t} * e^{-\lambda_{K-i}t}\\
&= (-(k+1)\lambda)\sum_{i=0}^{k}(-1)^i{}_kC_i\left(-\frac{1}{(i-k-1)\lambda}\left(e^{-\lambda_{K-k-1}t}-e^{-\lambda_{K-i}t}\right)\right)\\
&= \sum_{i=0}^{k}(-1)^i{}_kC_i\frac{(k+1)\lambda}{(i-k-1)\lambda}\left(e^{-\lambda_{K-k-1}t}-e^{-\lambda_{K-i}t}\right)\\
&= \sum_{i=0}^{k}(-1)^{i+1}\frac{k!}{i!(k-i)!}\frac{k+1}{k+1-i}\left(e^{-\lambda_{K-k-1}t}-e^{-\lambda_{K-i}t}\right)\\
&= \sum_{i=0}^{k}(-1)^{i+1}{}_{k+1}C_i\left(e^{-\lambda_{K-k-1}t}-e^{-\lambda_{K-i}t}\right)\\
&= \left(\sum_{i=0}^{k}(-1)^{i+1}{}_{k+1}C_i\right)e^{-\lambda_{K-k-1}t}+\left(\sum_{i=0}^{k}(-1)^i{}_{k+1}C_i e^{-\lambda_{K-i}t}\right)\\
&= \sum_{i=0}^{k}(-1)^i{}_{k+1}C_i e^{-\lambda_{K-i}t}+(-1)^{k+1}{}_{k+1}C_{k+1}e^{-\lambda_{K-k-1}t}\\
&\quad \text{ここで}\left(\begin{array}{c}\sum_{i=0}^{k}(-1)^{i+1}{}_{k+1}C_i=-({}_{k+1}C_0(-1)^0+\cdots+{}_{k+1}C_k(-1)^k)\\ =-(((-1)+1)^{k+1}-{}_{k+1}C_{k+1}(-1)^{k+1})\end{array}\right)\\
&= \sum_{i=0}^{k+1}(-1)^i{}_{k+1}C_i e^{-\lambda_{K-i}t}
\end{aligned}
$$

逆変換して以下を得る。

$$
\begin{aligned}
P(S_K,t) &= \mathcal{L}^{-1}\left\{\frac{1}{t+\lambda_K}\right\}=e^{-\lambda_K t}\\
P(S_{K-1},t) &= \mathcal{L}^{-1}\left\{\frac{\lambda_K}{t+\lambda_{K-1}}\right\}*P(S_K,t)=\lambda_K e^{-\lambda_{K-1}t}*e^{-\lambda_K t}\\
&= \lambda_K(-\frac{1}{\lambda})(e^{-\lambda_K t}-e^{-\lambda_{K-1}t})\\
P(S_{K-2},t) &= \mathcal{L}^{-1}\left\{\frac{\lambda_{K-1}}{t+\lambda_{K-2}}\right\}*P(S_{K-1},t)=\lambda_{K-1}e^{-\lambda_{K-2}t}*P(S_{K-1},t)\\
&= \lambda_K\lambda_{K-1}(-\frac{1}{\lambda})e^{-\lambda_{K-2}}*(e^{-\lambda_K t}-e^{-\lambda_{K-1}t})\\
&= \lambda_K\lambda_{K-1}(-\frac{1}{\lambda})\left(e^{-\lambda_{K-2}}*e^{-\lambda_K t}-e^{-\lambda_{K-2}}*e^{-\lambda_{K-1}t}\right)\\
&= \lambda_K\lambda_{K-1}(-\frac{1}{\lambda})\left(\frac{1}{2\lambda}(e^{-\lambda_{K-2}}-e^{-\lambda_K t})-\frac{1}{\lambda}(e^{-\lambda_{K-2}}-e^{-\lambda_{K-1}t})\right)
\end{aligned}
$$

$$
\begin{aligned}
&= \lambda_K \lambda_{K-1} (-\frac{1}{\lambda})(-\frac{1}{2\lambda})(e^{-\lambda_K t} - 2e^{-\lambda_{K-1} t} + e^{-\lambda_{K-2}}) \\
P(S_{K-k}, t) &= \left(\prod_{i=K-k+1}^{K} \lambda_i \right) \left(\prod_{i=1}^{k} (-\frac{1}{i\lambda}) \right) \sum_{i=0}^{k} (-1)^i{}_k C_i e^{-\lambda_{K-i} t} \\
P(S_{K-k-1}, t) &= \mathcal{L}^{-1} \left\{ \frac{\lambda_{K-k}}{t + \lambda_{K-k-1}} \right\} * P(S_{K-k}, t) \\
&= \lambda_{K-k} e^{-\lambda_{K-k-1} t} * \left(\prod_{i=K-k+1}^{K} \lambda_i \right) \left(\prod_{i=1}^{k} (-\frac{1}{i\lambda}) \right) \sum_{i=0}^{k} (-1)^i{}_k C_i e^{-\lambda_{K-i} t} \\
&= \left(\prod_{i=K-k}^{K} \lambda_i \right) \left(\prod_{i=1}^{k+1} (-\frac{1}{i\lambda}) \right) (-(k+1)\lambda) \sum_{i=0}^{k} (-1)^i{}_k C_i e^{-\lambda_{K-k-1} t} * e^{-\lambda_{K-i} t} \\
&= \left(\prod_{i=K-k}^{K} \lambda_i \right) \left(\prod_{i=1}^{k+1} (-\frac{1}{i\lambda}) \right) \sum_{i=0}^{k+1} (-1)^i{}_{k+1} C_i e^{-\lambda_{K-i} t}
\end{aligned}
$$

すなわち、次を得る.

$$
\begin{cases}
P(S_k, t) = \left(\prod_{i=k+1}^{K} \lambda_i \right) \left(\prod_{i=1}^{K-k} (-\frac{1}{i\lambda}) \right) \sum_{i=0}^{K-k} (-1)^i{}_{K-k} C_i e^{-\lambda_{K-i} t} & \forall k \ (0 \le k < K) \\
P(S_K, t) = e^{-\lambda_K t}
\end{cases}
$$

A.2.2 信頼度と MTTF

信頼度は時刻 t で S_i にある確率 $P(S_i, t)$ によって、次のように表される.

$$
R_{system}(K, t) = \sum_{i=0}^{K} P(S_i, t)
$$

$$
\begin{aligned}
\int_0^\infty P(S_k, t) dt &= \left(\prod_{i=k+1}^{K} \lambda_i \right) \left(\prod_{i=1}^{K-k} (-\frac{1}{i\lambda}) \right) \sum_{i=0}^{K-k} (-1)^i{}_{K-k} C_i \int_0^\infty e^{-\lambda_{K-i} t} \\
&= \left(\prod_{i=k+1}^{K} \lambda_i \right) \left(\prod_{i=1}^{K-k} (-\frac{1}{i\lambda}) \right) \sum_{i=0}^{K-k} (-1)^i{}_{K-k} C_i \frac{1}{\lambda_{K-i}} \\
&= \left(\prod_{i=k+1}^{K} \lambda_i \right) \frac{1}{(K-k)!} \left(-\frac{1}{\lambda} \right)^{K-k} \left(\frac{1}{\lambda_K} + \cdots + (-1)^i{}_{K-k} C_i \frac{1}{\lambda_{K-i}} + \cdots + (-1)^{K-k} \frac{1}{\lambda_k} \right)
\end{aligned}
$$

であるから、MTTF は以下のよう計算される.

$$
MTTF_{system}(K) = \int_0^\infty R_{system}(K, t) dt = \int_0^\infty \left(P(S_K, t) + \sum_{k=0}^{K-1} P(S_k, t) \right) dt
$$

$$= \frac{1}{\lambda_K} + \sum_{k=0}^{K-1} \left(\left(\prod_{i=k+1}^{K} \lambda_i \right) \frac{1}{(K-k)!} \left(-\frac{1}{\lambda} \right)^{K-k} \left(\frac{1}{\lambda_K} + \cdots + (-1)^i {}_{K-k}C_i \frac{1}{\lambda_{K-i}} + \cdots + (-1)^{K-k} \frac{1}{\lambda_k} \right) \right)$$

A.2.3 MTTF の具体的な計算

$$N(k) = \frac{1}{k+1}\{n(K+1) - K + k\} \quad \left(\begin{array}{ll} N(k) = N(k+1) + \frac{N(k+1)-1}{k+1} & (0 \leq k < K) \\ N(K) = n & (\text{初期状態}) \end{array} \right)$$

および、$\lambda_k = (k+1)N(k)\lambda$であることに注意して、以下のように計算される.

$$MTTF_{system}(1) = \frac{1}{\lambda_K} + \left\{ \lambda_K \frac{1}{K!} \left(-\frac{1}{\lambda} \right)^K \left(\frac{1}{\lambda_K} - \frac{1}{\lambda_{K-1}} \right) \right\}$$

$$= \frac{1}{\lambda_K} + \left\{ \lambda_K \left(-\frac{1}{\lambda} \right)^K \left(\frac{-\lambda}{\lambda_K \lambda_{K-1}} \right) \right\} = \frac{1}{\lambda_K} + \left\{ \frac{1}{\lambda_{K-1}} \right\} = \frac{1}{2n\lambda} + \frac{1}{(2n-1)\lambda}$$

$$MTTF_{system}(2)$$

$$= \frac{1}{\lambda_K} + \left\{ \lambda_K \lambda_{K-1} \frac{1}{K!} \left(-\frac{1}{\lambda} \right)^K \left(\frac{1}{\lambda_K} - 2\frac{1}{\lambda_{K-1}} + \frac{1}{\lambda_{K-2}} \right) \right\} + \left\{ \lambda_K \frac{1}{(K-1)!} \left(-\frac{1}{\lambda} \right)^{K-1} \left(\frac{1}{\lambda_K} - \frac{1}{\lambda_{K-1}} \right) \right\}$$

$$= \frac{1}{\lambda_K} + \left\{ \lambda_K \lambda_{K-1} \frac{1}{K!} \left(-\frac{1}{\lambda} \right)^K \left(\frac{2\lambda^2}{\lambda_K \lambda_{K-1} \lambda_{K-2}} \right) \right\} + \left\{ \lambda_K \frac{1}{(K-1)!} \left(-\frac{1}{\lambda} \right)^{K-1} \left(\frac{-\lambda}{\lambda_K \lambda_{K-1}} \right) \right\}$$

$$\left(\begin{array}{ll} & \lambda_{K-1}\lambda_{K-2} - 2\lambda_K\lambda_{K-2} + \lambda_K\lambda_{K-1} \\ \text{ここで} & = (\lambda_{K-1} - \lambda_K)\lambda_{K-2} - \lambda_K(\lambda_{K-2} - \lambda_{K-1}) \\ & = (-\lambda)\lambda_{K-2} - \lambda_K(-\lambda) = \lambda(\lambda_K - \lambda_{K-2}) = 2\lambda^2 \end{array} \right)$$

$$= \frac{1}{\lambda_K} + \left\{ \frac{1}{K!} \left(-\frac{1}{\lambda} \right)^K \frac{2\lambda^2}{\lambda_{K-2}} \right\} + \left\{ \frac{1}{(K-1)!} \left(-\frac{1}{\lambda} \right)^{K-1} \frac{-\lambda}{\lambda_{K-1}} \right\}$$

$$= \frac{1}{3n\lambda} + \left\{ \frac{2\lambda^2}{2\lambda^2(3n-1)\lambda} \right\} + \left\{ \frac{-\lambda}{(-\lambda)(3n-2)\lambda} \right\}$$

$$= \frac{1}{3n\lambda} + \left\{ \frac{1}{(3n-1)\lambda} \right\} + \left\{ \frac{1}{(3n-2)\lambda} \right\}$$

付録 B

基礎機能測定プログラム

本文掲載以外のプログラムについて添付する.

B.1 コミュニケーションコスト測定プログラム

```
:- module main.
main :-
    unix:argv(Arg),
    (Arg = [X|_] -> ( string_to_integer(X,N,R),
                      go(N,R,End) );
     otherwise;
     true -> (  klicio:klicio([stdout(normal(Out))]),
                Out = [fwrite("N : number of times \n")] )).

string_to_integer(S,N,R) :- string(S,K,8) | s_to_i(0,K,S,0,N,R).

s_to_i(I,K,S,Temp,Result,R) :- I < K |
        string_element(S,I,X),
        (#"0" =< X, X =< #"9" ->
                  ( Temp1 := (Temp * 10) + (X - #"0"),
                    s_to_i(~(I+1),K,S,Temp1,Result,R) ) ;
         otherwise;
         true ->  ( Result = Temp, R = print)).
s_to_i(I,K,S,Temp,Result,R) :- I = K | Result = Temp, R = ordinal.

go(N,Type,End) :-
    klicio:klicio([stdout(normal(Out))]),
    go(N,Type,End)+Out+[].
```

```
go(N,ordinal,End)-Out :-
        server(Channel)@ node(1),
        count(N,Channel,End)-Out.
go(N,print  ,End)-Out :-
        count(N,Channel,End)-Out @ node(1),
        server(Channel).

server([X|L]) :- X = 0, server(L).
server([]).

count(N,Ch,End)-O :-
    Ch = [Start|Ch1],
    count_1(Start,N,Ch1,End)-O.

count_1(0,N,Ch,End)-O :-
    util:times(UserTime),
    count_2(UserTime,N,Ch,E,1),
    end(E,UserTime,End)-O.

count_2(UT,N,Ch,End,K) :- integer(UT), K < N, K1:=K+1 |
        Ch = [Next|Ch1],
        count_2(Next,N,Ch1,End,K1).
otherwise.
count_2(UT,N,Ch,End,K) :- Ch = [], End = 0.

end(0,UT,R)-O :-
    util:times(UT1),
    O <= putt([~(UT1-UT)]), O <= nl, O <= fflush(R).
```

B.1.1 共通主ルーチン

```
:- module main.
main :-
    unix:argv(Arg),
    (Arg = [X] -> ( string_to_integer(X,N,R),
                    go(N,R,End) );
     otherwise;
     true -> (  klicio:klicio([stdout(normal(Out))]),
                Out = [fwrite("N : number of times \n")] )).

string_to_integer(S,N,R) :- string(S,K,8)  |
        s_to_i(0,K,S,0,N,R).

s_to_i(I,K,S,Temp,Result,R) :- I < K |
```

```
        string_element(S,I,X),
        (#"0" =< X, X =< #"9" ->
                  ( Temp1 := (Temp * 10) + (X - #"0"),
                    s_to_i(~(I+1),K,S,Temp1,Result,R) ) ;
         otherwise;
         true ->  ( Result = Temp, R = print)).
s_to_i(I,K,S,Temp,Result,R) :- I = K | Result = Temp, R = ordinal.

go(N,Type,End) :-
    klicio:klicio([stdout(normal(Out))]),
    util:times(UserTime),
    go(N,Type,UserTime,End)+Out+[].

go(N,ordinal,UserTime,End)-Out :- integer(UserTime) |
        count(Result,0,UserTime,End)-Out,
        benchmark:loop(N,Result) .        % 各プログラム呼出し
go(N,print  ,UserTime,End)-Out :- integer(UserTime) |
        benchmark:loop(N,Result),
        count(Result,0,UserTime,End)-Out.

count([A|X],K,UT,R)-Out :- count(X,~(K+1),UT,R)-Out.
otherwise.
count(    _,K,UT,R)-Out :-
    util:times(UT1),
    Out <= putt([~(UT1-UT),K]),
    Out <= nl,
    Out <= fflush(R).
```

B.1.2 nCBUE2 上用タイマープログラム

```
:- inline:
"
#include <signal.h>
#include <unistd.h>
".

:- module util.

times(Time) :-
    inline:"
    {int  start;
     start = micclk();
     %0=makeint(start);
     }":[T-int] | Time = T.
```

付録 C

ベンチマークプログラム

C.1 オリジナルプログラム

```
:-module simulation.
% 非決定的実行 N 回、決定的実行 M 回 を各 PE 上で実行する.

sim(N,M,R) :- M >= N |
    current_node(_,All),
    sim0(N,M,R,1,~(All-1)) @ node(1).

sim0(N,M,R,H,A) :-  H < A, To := H+1 |
R = [R1|R2],
sim1(N,M,R1)@node(To),
sim0(N,M,R2,To,A).
sim0(N,M,R,A,A) :- sim1(N,M,R).

% det/3 は 決定的実行を M/2-N/2 回行った後、
% N/2 回の決定的実行によって値を転送
% する.  二つの det/3 から、N/2 づつ送られた値を元に
% ndet は非決定的実行を N 回行う.
sim1(N,M,R) :- N1 := N/2, M1 := (M-N)/2 |
    ndet(R0,R1,0,R),
    det(M1,N1,R0),det(M1,N1,R1).

ndet([X|A],B,K,C) :- K1 := K+X | ndet(A,B,K1,C).
ndet(A,[X|B],K,C) :- K1 := K+X | ndet(A,B,K1,C).
ndet([],[],K,C)   :- C=K.

det(M,N,R) :- M > 0, M1:=M-1 | det(M1,N,R).
det(M,N,R) :- M =< 0 | det1(N,R).
```

```
det1(N,R) :- N > 0, N1:=N-1 | R=[N|R1], det1(N1,R1).
det1(N,R) :- N =< 0 | R = [].
```

C.2 耐故障プログラム

```
:-module simulation.

sim(N,M,R) :- M >= N |
    watchdog:distribution(Primary,Backup),
    sim_0({N,M,R},Primary,Backup) @ lower_priority(10).

sim_0(Args,[PTop|Primary],[BTop|Backup]) :-
    copy:copy(Args,Args1,Args2,Interrupt),
    Interrupt = {Interrupt1,Interrupt2}, Log = ack(Log1),
    PTop = {primary,simulation,sim,Args1,Log,Signal,Interrupt1},
    BTop = {backup ,simulation,sim,Args2,Log1,Signal,Interrupt2}.

%%%%
outputcommit(Ack,X,Y)-SC :- wait(Ack) | X = Y.

sim_record(N,M,R)+Log+Sig+Raise-SC :-
    current_node(_,All),
    sim0_record(N,M,R,~((All-1)/2))+Log+Sig+Raise-SC.

sim0_record(N,M,R,A)+ack(Log)+Sig+Raise-SC :- A > 1, A1 := A-1 |
Log = c1(Ack1,Ack2),
outputcommit(Ack1,R,[R1|R2])-SC,
Raise = [goal(primary,simulation,sim0,{N,M,R2,A1},Ack1)],
sim1_record(N,M,R1)+Ack2-SC.
sim0_record(N,M,R,1)+ack(Log)+Sig+Raise-SC :-
    Log = c2(Ack), Raise = [],
    sim1_record(N,M,R)+Ack-SC.

sim1_record(N,M,R)+ack(Log)-SC :- N1 := N/2, M1 := (M-N)/2 |
Log = c1(Ack1,Ack2,Ack3),
    ndet_record(R0,R1,0,R)+Ack1-SC,
    det_record(M1,N1,R0)+Ack2-SC,det_record(M1,N1,R1)+Ack3-SC.

ndet_record([X|A],B,K,C)+ack(Log)-SC :- K1 := K+X |
Log = c1(Ack), ndet_record(A,B,K1,C)+Ack-SC.
ndet_record(A,[X|B],K,C)+ack(Log)-SC :- K1 := K+X |
Log = c2(Ack), ndet_record(A,B,K1,C)+Ack-SC.
```

```
ndet_record([],[],K,C)+ack(Log)-SC :-
    Log = c3(Ack), outputcommit(Ack,C,K)-SC.

det_record(M,N,R)+Log-SC :- wait(Log) | det(M,N,R)-SC.

det(M,N,R)-SC :- M > 0, M1:=M-1 | det(M1,N,R)-SC.
det(M,N,R)-SC :- M =< 0 | det1(N,R)-SC.

det1(N,R)-SC :- N > 0, N1:=N-1 | R=[N|R1], det1(N1,R1)-SC.
det1(N,R)-SC :- N =< 0 | R = [].

%%%%%%%
sim_replay(N,M,R)+Log+Sig+Raise-SC :-
    (wait(Log) -> sim_replay_1(N,M,R)+Log+Sig+Raise-SC ;
     alternatively;
     Sig = [rebirth|Sig1] ->      sim(N,M,R)+Raise-SC).
sim_replay_1(N,M,R)+Log+Sig+Raise-SC :-
    current_node(_,All),
    sim0_replay(N,M,R,~((All-1)/2))+Log+Sig+Raise-SC.

sim0_replay(N,M,R,A)+Log+Sig+Raise-SC :-
    (wait(Log) -> sim0_replay_1(N,M,R,A)+Log+Sig+Raise-SC ;
     alternatively;
     Sig = [rebirth|Sig1] ->      sim0(N,M,R,A)+Raise-SC).

sim0_replay_1(N,M,R,A)+Log+Sig+Raise-SC :-
    Log = c1(Ack1,Ack2), A > 1, A1 := A-1 |
Ack1 = ack(Log1),Ack2 = ack(Log2),
R=[R1|R2],
Raise = [goal(backup,simulation,sim0,{N,M,R2,A1},Log1)],
sim1_replay(N,M,R1)+Log2+Sig-SC.
sim0_replay_1(N,M,R,1)+Log+Sig+Raise-SC :- Log = c2(Ack) |
Ack = ack(Log1), Raise = [],
sim1_replay(N,M,R)+Log1+Sig-SC.

sim1_replay(N,M,R)+Log+Sig-SC :-
    (wait(Log) -> sim1_replay_1(N,M,R)+Log+Sig-SC ;
     alternatively;
     Sig = [rebirth|Sig1] ->      sim1(N,M,R)-SC).

sim1_replay_1(N,M,R)+Log+Sig-SC :-
    Log = c1(Ack1,Ack2,Ack3), N1 := N/2, M1 := (M-N)/2 |
Ack1 = ack(Log1), Ack2 = ack, Ack3 = ack,
ndet_replay(R0,R1,0,R)+Log1+Sig-SC,
det(M1,N1,R0)-SC,det(M1,N1,R1)-SC.
```

```
ndet_replay(A,B,C,D)+Log+Sig-SC :-
    (wait(Log) -> ndet_replay_1(A,B,C,D)+Log+Sig-SC ;
     alternatively;
     Sig = [rebirth|Sig1] ->      ndet(A,B,C,D)-SC).

ndet_replay_1([X|A],B,K,C)+Log+Sig-SC :- Log = c1(Ack), K1 := K+X |
Ack=ack(Log1), ndet_replay(A,B,K1,C)+Log1+Sig-SC.
ndet_replay_1(A,[X|B],K,C)+Log+Sig-SC :- Log = c2(Ack), K1 := K+X |
Ack=ack(Log1), ndet_replay(A,B,K1,C)+Log1+Sig-SC.
ndet_replay_1([],[],  K,C)+Log+Sig-SC :- Log = c3(Ack) |
Ack=ack, C=K.

%%%%%% ORIGINAL

sim(N,M,R)+Raise-SC :-
    current_node(_,All),
    sim0(N,M,R,~((All-1)/2))+Raise-SC.

sim0(N,M,R,A)+Raise-SC :- A > 1, A1 := A-1 |
R=[R1|R2],
Raise = [goal(original,simulation,sim0,{N,M,R2,A1},_)],
sim1(N,M,R1)-SC.
sim0(N,M,R,1)+Raise-SC :-
    Raise = [],
    sim1(N,M,R)-SC.

sim1(N,M,R)-SC :- N1 := N/2, M1 := (M-N)/2 |
    ndet(R0,R1,0,R)-SC,
    det(M1,N1,R0)-SC,det(M1,N1,R1)-SC.

ndet([X|A],B,K,C)-SC :- K1 := K+X | ndet(A,B,K1,C)-SC.
ndet(A,[X|B],K,C)-SC :- K1 := K+X | ndet(A,B,K1,C)-SC.
ndet([],[],K,C)-SC   :- C=K.

%%%%%
:- module exgoal.

%%%%%%%%%%%%% External call
call_goal(Site,Module,Predicate,Args,Log,GSig,Raise)-SC :-
    call_goal_0(Site,Module,Predicate,Args,
Log,GSig,Raise)-SC @ lower_priority.

call_goal_0(primary,simulation,sim0,{A,B,C,D},Log,GSig,Raise)-SC :-
    simulation:sim0_record(A,B,C,D)+Log+GSig+Raise-SC.
```

```
call_goal_0(backup, simulation,sim0,{A,B,C,D},Log,GSig,Raise)-SC :-
    simulation:sim0_replay(A,B,C,D)+Log+GSig+Raise-SC.
call_goal_0(original,simulation,sim0,{A,B,C,D},Log,GSig,Raise)-SC :-
    simulation:sim0(A,B,C,D)+Raise-SC.

call_goal_0(primary,simulation,sim,{A,B,C},Log,GSig,Raise)-SC :-
    simulation:sim_record(A,B,C,Log)+GSig+Raise-SC.
call_goal_0(backup ,simulation,sim,{A,B,C},Log,GSig,Raise)-SC :-
    simulation:sim_replay(A,B,C,Log)+GSig+Raise-SC.
otherwise.
call_goal_0(Type,Module,Method,Arguments,Log,GSignal,Raise)-SC :-
    klicio:klicio([stderr(normal(Out))]),
    variable:wrap((Type::Module:Method/Arguments), G),
    Out = [fwrite("Illegal goal invocation : "),
   putwt(G), nl,fflush(_)],
    Raise = [].
```

C.3 主ルーチン

```
:- module main.
main :-
    unix:argv(Arg),
    (Arg = [X,Y] -> ( string_to_integer(X,N,_),
                      string_to_integer(Y,M,R),
                    go(N,M,R,End) );
     otherwise;
     true -> (  klicio:klicio([sterr(normal(Out))]),
                Out = [fwrite("Illegal input !\n")] )).

string_to_integer(S,N,R) :- string(S,K,8)  |
        s_to_i(0,K,S,0,N,R).

s_to_i(I,K,S,Temp,Result,R) :- I < K |
        string_element(S,I,X),
        (#"0" =< X, X =< #"9" ->
                  ( Temp1 := (Temp * 10) + (X - #"0"),
                    s_to_i(~(I+1),K,S,Temp1,Result,R) ) ;
         otherwise;
         true ->  ( Result = Temp, R = print)).
s_to_i(I,K,S,Temp,Result,R) :- I = K | Result = Temp, R = ordinal.

go(N,M,Type,End) :- integer(N), integer(M) |
        klicio:klicio([stdout(normal(Out))]),
```

```
        util:times(UserTime),
        go(N,M,Type,UserTime,End)+Out+[].
otherwise.
go(N,M,Type,End) :-
    klicio:klicio([stderr(normal(Out))]),
    Out = [fwrite("Illegal argument ! "),putt(N),fwrite(" "),putt(M),nl].

go(N,M,ordinal,UserTime,End)-Out :- integer(UserTime) |
        count(Result,0,UserTime,End)-Out,
% TARGET PROGRAM
        simulation:sim(N,M,Result) @ lower_priority(10).
go(N,M,print  ,UserTime,End)-Out :- integer(UserTime) |
        count_print(Result,0,UserTime,End)-Out,
% TARGET PROGRAM
        simulation:sim(N,M,Result) @ lower_priority(10).

% OUTPUT ROUTINE
count([A|X],N,UT,R)-Out :- integer(A) | count(X,~(N+1),UT,R)-Out.
otherwise.
count(    X,N,UT,R)-Out :- end_count(X,N,UT,R)-Out.

end_count(_,N,UT,R)-Out :-
% STOP STOPWATCH
    util:times(UT1),
    Out <= putt([~(UT1-UT),N]),
    Out <= nl,
    Out <= fflush(R).

count_print([A|X],N,UT,R)-Out :- integer(A) |
    Out <= putt(A), Out <= fwrite(" "), Out <= fflush(_),
    count_print(X,~(N+1),UT,R)-Out.
otherwise.
count_print(X,N,UT,R)-Out :- end_count(X,N,UT,R)-Out.
```

付録 D

耐故障プログラムライブラリ

D.1 監視プロセスプログラム

```
/*
#define    WATCHDOG_CYCLE         5
#define    PRIMARY_TIMEOUT       10
#define    BACKUP_TIMEOUT        30
*/

:- inline:
"
#define    WATCHDOG_CYCLE        30
#define    PRIMARY_TIMEOUT       30
#define    BACKUP_TIMEOUT        90
".

:- module watchdog.

distribution(Primary,Backup) :-
    current_node(N,Num),
    (N = 0, Num > 2 ->
     ( util:alarm_timer(AlarmS),
       (AlarmS = normal(Alarm) ->
        (fork_watchdog(1,Num,Primary0,Backup0,Alarm) @ lower_priority,
         connect(Primary0,Primary,"Primary Site Group : ")
                                          @ lower_priority,
         connect(Backup0, Backup ,"Backup Site Group  : ")
                                          @ lower_priority
         ) ;
        otherwise;
```

```
        true -> (message("Alarm Timer not available !\n"),
                 Primary = [], Backup = [])
        )
       );
     otherwise;
     true ->( message("There are not enough nodes to FTS\n"),
              Primary = [],    Backup = [])).

fork_watchdog(N,M,Stream1,Stream2,Alarm) :- N < M,
    inline:"%0=makeint(WATCHDOG_CYCLE);":[WATCHDOG_CYCLE-int] |
        pre_watchdog(S) @ node(N),
        Alarm = [set_alarm(WATCHDOG_CYCLE,TIMEOUT)|Alarm1],
        wait_watchdog_ready(~(N+1),M,Stream1,Stream2,Alarm1,S,TIMEOUT).
fork_watchdog(N,M,Stream1,Stream2,Alarm) :- N >= M |
        Stream1 = [], Stream2 = [], Alarm = [].

wait_watchdog_ready(N,M,Stream1,Stream2,Alarm,S,alarmed) :-
    fork_watchdog(N,M,Stream1,Stream2,Alarm).
wait_watchdog_ready(N,M,Stream1,Stream2,Alarm,normal(S),R) :-
    Stream1 = [S|Stream3],
    fork_watchdog(N,M,Stream2,Stream3,Alarm).

connect(ToWatchdog,Result,Message) :-
    connect_watchdog(ToWatchdog,Result0,A,A,0,N),
    (Result0 = [Top|Result1] ->
       ( Top = [shrink(L)|Top1],
         Result = [Top1|Result1],
         connect_test(L,L1),
         message(Message,L1) )).
connect_test([N|List],L) :- integer(N) |
        L = [N," "|L1],
        connect_test(List,L1).
connect_test([],L) :- L = ["\n"].

connect_watchdog([S0|Streams], Result ,Former,Latter,K,N) :-
    S0 = {Former,S1,Next},
    Result = [S1|Result1],
    connect_watchdog(Streams, Result1, Next, Latter,~(K+1),N).
connect_watchdog([],Result,Former,Latter,K,N) :-
    Result = [],
    Former = Latter,
    K = N.

% ------------------------------------------------------------
%%%%%  Preliminary status
```

```
pre_watchdog(S) :-
    S = normal(S1),
    pre_watchdog0(S1).

pre_watchdog0({Former,S,Latter}) :-
    watchdog_start(S,Former,Latter).

%%%%%  Watchdog Initiation
% (1) loop back test and shrink
watchdog_start(S,[shrink(X)|Former1],Latter) :-

%        message("Shrink [",[N,"]\n"]),

    current_node(N,_),
    X = [N|Y],
    Latter = [shrink(Y)|Former1].
% (2) shrink top
watchdog_start([shrink(X)|S],Former,Latter) :-
    current_node(N,_),

    message("Leader [",[N,"]\n"]),

    X = [N|Y],
    Latter = [shrink(Y)|Latter1],
    (Former = [shrink(P)|Former1] ->
        ( P = [],
          watchdog_leader(S,X,Former1,Latter1)
    )   ).
% (3) GC
watchdog_start(S,[gc|Former1],Latter) :-
    current_node(N,_),
    system_control:gc(BEFORE,AFTER),
    (wait(AFTER) ->
       message("GC on [",[N,"] ",BEFORE," >> ",AFTER,"\n"]),
       Latter = [gc|Latter1],
       watchdog_start(S,Former1,Latter1)).
watchdog_start([worsttime(R)|S],Former,Latter) :-
    current_node(N,_),
    system_control:gc(BEFORE,AFTER),
    (wait(AFTER) ->
       message("GC on [",[N,"] ",BEFORE," >> ",AFTER,"\n"]),
       Latter = [gc|Latter1],
       (Former = [gc|Former1] ->
            R = 0,
            watchdog_start(S,Former1,Latter1)
```

```
    )  ).
%%%%% Watchdog Leader Ready
watchdog_leader(TopGoal, [N|Member], Former,Latter) :-
    TopGoal = {Type,Module,Method,Arguments,Log,PBSignal,Interrupt},
    inline:"%0=makeint(WATCHDOG_CYCLE);
            %1=makeint(BACKUP_TIMEOUT);":
            [WATCHDOG_CYCLE-int,BACKUP_TIMEOUT-int] |
        weight(Member,0,M),
        generic:new(merge,Raise0,Raise),
        exgoal:call_goal(Type,Module,Method,Arguments,Log,
                         GSignal,Raise0,end,SC),
        (Type = primary ->
         ( util:alarm_timer(AlarmS),
           (AlarmS = normal(Alarm0) ->
            (Alarm0 = [set_alarm(WATCHDOG_CYCLE,Cycle)|Alarm],
             watchdog_primary(SC,Cycle,PBSignal,GSignal,Raise,Interrupt,
                      Former,Latter,Alarm,Member,M,M) @ lower_priority
             );
            otherwise;
            true ->
               (message("Alarm Timer not available !\n"),
                Latter = [])
            ) ) ;
         Type = backup  ->
         ( util:alarm_timer(AlarmS),
           (AlarmS = normal(Alarm0) ->
            (Alarm0 = [set_alarm(BACKUP_TIMEOUT,Cycle)|Alarm],
             watchdog_backup( SC,Cycle,PBSignal,GSignal,Raise,Interrupt,
                      Former,Latter,Alarm,Member,M,M) @ lower_priority
             );
            otherwise;
            true ->
            (message("Alarm Timer not available !\n"),
             Latter = [])
            ) ) ).
otherwise.
watchdog_leader(TopGoal,Member,Former,Latter) :-
    Latter = [].

weight([_|Member],K,M) :-weight(Member,~(K+1),M).
weight([],K,M) :- M = K.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  Primary Site Leader
% (1) going in patrol cycle
watchdog_primary(SC,alarmed,PBSignal,GSignal,Raise,Interrupt,
                 Former,Latter,Alarm,Nodes,Weight,Total) :-
```

```
    util:times(BEGIN),
    ( ( integer(BEGIN),
        inline:"%0=makeint(PRIMARY_TIMEOUT);":[PRIMARY_TIMEOUT-int] ) ->
      ( Latter = [ping|Latter1],
        Alarm  = [set_alarm(PRIMARY_TIMEOUT,TIMEOUT)|Alarm1],
        watchdog_primary_patrol(SC,BEGIN,PBSignal,GSignal,Raise,Interrupt,
                        Former,Latter1,Alarm1,Nodes,Weight,Total,TIMEOUT)
    )     ).
alternatively.
% (2) a goal is forwarded
watchdog_primary(SC,C,PBSignal,GSignal,Raise1,Interrupt,
                 [Goal|Former],Latter,Alarm,Nodes,Weight,Total) :-
    watchdog_primary_1(SC,C,PBSignal,GSignal,Raise1,Interrupt,
                       Former,Latter,Alarm,Nodes,Weight,Total,Goal).
% (3) a goal is raised from children
watchdog_primary(SC, C,PBSignal,GSignal,Raise,Interrupt,
                 Former,Latter,Alarm,Nodes,Weight,Total) :-
    Raise = [Goal|Raise1] |
        (%  (5.1) there is no extra nodes
         Nodes = [] ->
         ( Latter = [Goal|Latter1],
           watchdog_primary(SC,C,PBSignal,GSignal,Raise1,Interrupt,
                            Former,Latter1,Alarm,Nodes,Weight,Total) );
         %  (5.2) there are some available nodes
         Nodes = [N|Nodes1] ->
         ( watchdog_primary_expand(SC,C,PBSignal,GSignal,Raise1,Interrupt,
                                   Former,Latter,Alarm,Nodes1,Weight,Total,
                                   Goal,N) )).
% (4) termination
watchdog_primary(end,C,PBSignal,GSignal,[],  Interrupt,
                 Former,Latter,Alarm,Nodes,Total,Total) :-
    Interrupt = [terminate], Latter = [],
    (Former = [] -> Alarm = [], PBSignal = [] ;
     otherwise;
     Former = [Goal|Former1] ->
         ( watchdog_primary_1(end,C,PBSignal,GSignal,[],Interrupt,
                              Former1,Latter,Alarm,Nodes,Total,Total,Goal) )).

%%%%
% (2.1) a goal is forwarded
watchdog_primary_1(SC,C,PBSignal,GSignal,Raise1,Interrupt,
                   Former,Latter,Alarm,Nodes,Weight,Total,Goal) :-
    Goal = goal(Type,Module,Method,Arguments,Log) |
        % (2.1.1) a goal is executed on this node
        (Nodes = [] ->
```

```
        ( on_transition(Type,GSignal),
          generic:new(merge,{Raise1,Raise2},Raise),
          exgoal:call_goal(Type,Module,Method,Arguments,Log,
                           GSignal,Raise2,SC,SC1),
          watchdog_primary(SC1,C,PBSignal,GSignal,Raise,Interrupt,
                           Former,Latter,Alarm,Nodes,Weight,Total) );
        % (2.1.2) a goal is forwarded to new node with a new watchdog
        Nodes = [N|Nodes1] ->
           ( watchdog_primary_expand(SC,C,PBSignal,GSignal,Raise1,Interrupt,
                                     Former,Latter,Alarm,Nodes1,Weight,Total,
                                     Goal,N) )).
% (2.2) neighbor node is shrinked
watchdog_primary_1(SC,C,PBSignal,GSignal,Raise,Interrupt,
                   Former,Latter,Alarm,Nodes,Weight,Total,Goal) :-
    Goal = shrink(N) |
        watchdog_primary(SC,C,PBSignal,GSignal,Raise,Interrupt,
                         Former,Latter,Alarm,[N|Nodes],~(Weight+1),Total).

on_transition(backup, GSignal) :- GSignal = [rebirth].
on_transition(primary,GSignal).
on_transition(original,_).

on_transition(primary,GType,GSignal) :- on_transition(GType,GSignal).
on_transition(backup ,_,_).
on_transition(original,_,_).

watchdog_primary_expand(SC,C,PBSignal,GSignal,Raise1,Interrupt,
                        Former,Latter,Alarm,Nodes1,Weight,Total,Goal,N) :-
%    message("Expand [",[N,"]\n"]),

    generic:new(merge,{Former,End},Former1),
    watchdog(primary,Goal,Next,Latter,N,End) @ node(N),
    watchdog_primary(SC,C,PBSignal,GSignal,Raise1,Interrupt,
                     Former1,Next,Alarm,Nodes1,~(Weight-1),Total).

%%% on Patrol
% (1) fault
watchdog_primary_patrol(SC,BEGIN,PBSignal,GSignal,Raise,Interrupt,
                        Former,Latter,Alarm,Nodes,Weight,Total,alarmed) :-
    message("FAULT was detected on PRIMARY SITE!\n"),
    PBSignal = [fault], GSignal = [fault], Latter = [fault],
    Alarm = [].
alternatively.
% (2) terminate patrol
watchdog_primary_patrol(SC,BEGIN,PBSignal,GSignal,Raise,Interrupt,
```

```
                        Former,Latter,Alarm,Nodes,Weight,Total,TIMEOUT) :-
    Former = [ping|Former1],
    inline:"%0=makeint(WATCHDOG_CYCLE);":[WATCHDOG_CYCLE-int] |
        util:times(END),
        PBSignal = [alive|PBSignal1],
        message("PRIMARY SITE is ALIVE ! : ",[~(END-BEGIN),"\n"]),
        Alarm = [set_alarm(WATCHDOG_CYCLE,Cycle)|Alarm1],
        watchdog_primary(SC,Cycle,PBSignal1,GSignal,Raise,Interrupt,
                         Former1,Latter,Alarm1,Nodes,Weight,Total).
alternatively.
% (3) a goal is forwarded
watchdog_primary_patrol(SC,BEGIN,PBSignal,GSignal,Raise1,Interrupt,
                        [Goal|Former],Latter,Alarm,Nodes,Weight,Total,TO) :-
    watchdog_primary_patrol_1(SC,BEGIN,PBSignal,GSignal,Raise1,Interrupt,
                        Former,Latter,Alarm,Nodes,Weight,Total,TO,Goal).
% (4) a goal is raised
watchdog_primary_patrol(SC,BEGIN,PBSignal,GSignal,Raise,Interrupt,
                        Former,Latter,Alarm,Nodes,Weight,Total,TO) :-
    Raise = [Goal|Raise1] |
        ( % (4.1) there is no extra nodes
         Nodes = [] ->
             (  Latter = [Goal|Latter1],
                watchdog_primary_patrol(SC,BEGIN,PBSignal,GSignal,Raise1,
                                        Interrupt,Former,Latter1,Alarm,Nodes,
                                        Weight,Total,TO) ) ;
          % (4.2) there are some available nodes
          Nodes = [N|Nodes1] ->
             (
              watchdog_primary_patrol_expand(SC,BEGIN,PBSignal,GSignal,Raise1,
                                       Interrupt,Former,Latter,Alarm,
                                       Nodes1,Weight,Total,TO,Goal,N))).

% (3.1) a goal is forwarded
watchdog_primary_patrol_1(SC,BEGIN,PBSignal,GSignal,Raise1,Interrupt,
                          Former,Latter,Alarm,Nodes,Weight,Total,TO,Goal) :-
    Goal = goal(Type,Module,Method,Arguments,Log) |
        on_transition(Type,GSignal),
        generic:new(merge,{Raise1,Raise2},Raise),
        exgoal:call_goal(Type,Module,Method,Arguments,Log,
                         GSignal,Raise2,SC,SC1),
        watchdog_primary_patrol(SC1,BEGIN,PBSignal,GSignal,Raise,Interrupt,
                                Former,Latter,Alarm,Nodes,Weight,Total,TO).
% (3.2) neighbor node is shrinked
watchdog_primary_patrol_1(SC,BEGIN,PBSignal,GSignal,Raise,Interrupt,
                          Former,Latter,Alarm,Nodes,Weight,Total,TO,Goal) :-
```

```
    Goal = shrink(N) |
        watchdog_primary_patrol(SC,BEGIN,PBSignal,GSignal,Raise,Interrupt,
                        Former,Latter,Alarm,[N|Nodes],~(Weight+1),Total,TO).


watchdog_primary_patrol_expand(SC,BEGIN,PBSignal,GSignal,Raise1,Interrupt,
                               Former,Latter,Alarm,Nodes1,Weight,Total,TO,
                               Goal,N) :-
%    message("Expand [",[N,"]\n"]),

    generic:new(merge,{Former,End},Former1),
    watchdog(primary,Goal,Next,Latter,N,End) @ node(N),
    watchdog_primary_patrol(SC,BEGIN,PBSignal,GSignal,Raise1,Interrupt,
                            Former1,Next,Alarm,Nodes1,~(Weight-1),Total,TO).

%%%%% Backup Site Leader
% (1) Signal arrived
watchdog_backup(SC,C,PBSignal,GSignal,Raise,Interrupt,
                Former,Latter,Alarm,Nodes,Weight,Total) :-
    PBSignal = [Sig|PBSignal1], atom(Sig) |
        (% (1.1) regular report arrived from the primary site
         Sig = alive,
         inline:"%0=makeint(BACKUP_TIMEOUT);":[BACKUP_TIMEOUT-int] ->
           ( Alarm = [set_alarm(BACKUP_TIMEOUT,Cycle)|Alarm1],
             watchdog_backup(SC,Cycle,PBSignal1,GSignal,Raise,Interrupt,
                             Former,Latter,Alarm1,Nodes,Weight,Total) ) ;
         % (1.2) fault is informed
         Sig = fault,
         inline:"%0=makeint(WATCHDOG_CYCLE);":[WATCHDOG_CYCLE-int] ->
           ( message("FAULT was informed to BACKUP!\n"),
             message("BACKUP change to PRIMARY !!! "),
             Latter  = [rebirth(R)|Latter1],
             watchdog_backup_rebirth(SC,C,R,GSignal,Raise,Interrupt,
                             Former,Latter1,Alarm,Nodes,Weight,Total) )).
alternatively.
% (2) timeout for regular reports
watchdog_backup(SC,alarmed,PBSignal,GSignal,Raise,Interrupt,
                Former,Latter,Alarm,Nodes,Weight,Total) :-
    message("TIMEOUT on BACKUP!\n"),
    message("BACKUP change to PRIMARY !!! "),
    Latter  = [rebirth(R)|Latter1],
    watchdog_backup_rebirth(SC,C,R,GSignal,Raise,Interrupt,
                            Former,Latter1,Alarm,Nodes,Weight,Total).
alternatively.
% (3) goal is forwarded
```

```
watchdog_backup(SC,C,PBSignal,GSignal,Raise1,Interrupt,
                [Goal|Former],Latter,Alarm,Nodes,Weight,Total) :-
    watchdog_backup_1(SC,C,PBSignal,GSignal,Raise1,Interrupt,
                      Former,Latter,Alarm,Nodes,Weight,Total,Goal).
% (4) a goal is raised
watchdog_backup(SC, C,PBSignal,GSignal,Raise,Interrupt,
                Former,Latter,Alarm,Nodes,Weight,Total) :-
    Raise = [Goal|Raise1] |
        ( %  (4.2) there is no extra nodes
          Nodes = [] ->
             ( Latter = [Goal|Latter1],
               watchdog_backup(SC,C,PBSignal,GSignal,Raise1,Interrupt,
                               Former,Latter1,Alarm,Nodes,Weight,Total) ) ;
          %  (4.1) there are some available nodes
          Nodes = [N|Nodes1] ->
             (
              watchdog_backup_expand(SC, C,PBSignal,GSignal,Raise1,Interrupt,
                                     Former,Latter,Alarm,Nodes1,Weight,Total,
                                     Goal,N)
              )).
% (4) termination
watchdog_backup(end,C,PBSignal,GSignal,[],Interrupt,
                Former,Latter,Alarm,Nodes,Total,Total) :-
    Interrupt = [terminate], Latter = [],
    (Former = [] -> Alarm = [], PBSignal = [] ;
     otherwise;
     Former = [Goal|Former1] ->
         ( watchdog_backup_1(end,C,PBSignal,GSignal,[],Interrupt,
                             Former1,Latter,Alarm,Nodes,Total,Total,Goal) )).

% (3.1) goal is forwarded
watchdog_backup_1(SC,C,PBSignal,GSignal,Raise1,Interrupt,
                  Former,Latter,Alarm,Nodes,Weight,Total,Goal) :-
    Goal = goal(Type,Module,Method,Arguments,Log) |
        (Nodes = [] ->
            ( generic:new(merge,{Raise1,Raise2},Raise),
              exgoal:call_goal(Type,Module,Method,Arguments,Log,
                               GSignal,Raise2,SC,SC1),
              watchdog_backup(SC1,C,PBSignal,GSignal,Raise,Interrupt,
                              Former,Latter,Alarm,Nodes,Weight,Total) ) ;
         Nodes = [N|Nodes1] ->
            (
              watchdog_backup_expand(SC, C,PBSignal,GSignal,Raise1,Interrupt,
                                     Former,Latter,Alarm,Nodes1,Weight,Total,
                                     Goal,N)   )).
```

```
% (3.2) neighbor node is shrinked
watchdog_backup_1(SC,C,PBSignal,GSignal,Raise,Interrupt,
                  Former,Latter,Alarm,Nodes,Weight,Total,Goal) :-
    Goal = shrink(N) |
        watchdog_backup(SC,C,PBSignal,GSignal,Raise,Interrupt,
                        Former,Latter,Alarm,[N|Nodes],~(Weight+1),Total).


watchdog_backup_expand(SC, C,PBSignal,GSignal,Raise1,Interrupt,
                       Former,Latter,Alarm,Nodes1,Weight,Total,Goal,N) :-
%    message("Expand [",[N,"]\n"]),
    generic:new(merge,{Former,End},Former1),
    watchdog(backup,Goal,Next,Latter,N,End) @ node(N),
    watchdog_backup(SC,C,PBSignal,GSignal,Raise1,Interrupt,
                    Former1,Next,Alarm,Nodes1,~(Weight-1),Total).

% =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-  New goal is forwarded with watchdog!
watchdog(Type,Goal,Former,Latter,N,End) :-
    Goal = goal(GType,Module,Method,Arguments,Log) |
        generic:new(merge,Raise0,Raise),
        exgoal:call_goal(GType,Module,Method,Arguments,Log,
                         GSignal,Raise0,end,SC),
        watchdog(SC,GSignal,Raise,Former,Latter,N,Type,End).
otherwise.
watchdog(Type,Goal,Former,Latter,N,End) :-
    illegal_goal(Goal,N),
    Former = Latter,
    End = [shrink(N)].

%%%%%%%% general watchdog (common use for primary/backup)
watchdog(SC,GSignal,Raise,[Goal|Former],Latter,N,Type,End) :-
    watchdog_1(SC,GSignal,Raise,Former,Latter,N,Type,End,Goal).
alternatively.
% (6) a goal is raised
watchdog(SC,GSignal,[Goal|Raise],Former,Latter,N,Type,End) :-
    Latter = [Goal|Latter1],
    watchdog(SC,GSignal,Raise,Former,Latter1,N,Type,End).
% (7) goals terminated under it, then shrink
watchdog(end,GSignal,[],Former,Latter,N,Type,End) :-
    shrink(End,N),
%    GSignal = [],
    Former = Latter.
% (8) termination forced
watchdog(SC,GSignal,[],[],Latter,N,Type,End) :-
%    GSignal = [],
```

```
    Latter = [], shrink(End,N).

% (1) patrol
watchdog_1(SC,GSignal,Raise,Former,Latter,N,Type,End,Goal) :-
    Goal = ping | watchdog(SC,GSignal,Raise,Former,Latter,N,Type,End).
% (2) a goal is forwarded
watchdog_1(SC,GSignal,Raise1,Former,Latter,N,Type,End,Goal) :-
    Goal = goal(GType,Module,Method,Arguments,Log) |
        on_transition(Type,GType,GSignal),
        generic:new(merge,{Raise1,Raise2},Raise),
        exgoal:call_goal(GType,Module,Method,Arguments,Log,
                         GSignal,Raise2,SC,SC1),
        watchdog(SC1,GSignal,Raise,Former,Latter,N,Type,End).
% (3) neighbor shrinked
watchdog_1(SC,GSignal,Raise,Former,Latter,N,Type,End,Goal) :-
    Goal = shrink(K) |
        Latter = [Goal|Latter1],
        watchdog(SC,GSignal,Raise,Former,Latter1,N,Type,End).
% (4) rebirth informed
watchdog_1(SC,GSignal,Raise,Former,Latter,N,Type,End,Goal) :-
    Goal = rebirth(R) |
        R = [{N,Ack}|RR],
        Latter  = [rebirth(RR)|Latter1],

        message("REBIRTH (1) [",[N,"]\n"]),

        watchdog_rebirth(SC,GSignal,Raise,Former,Latter1,N,Type,End,Ack).
% (5) fault informed
watchdog_1(SC,GSignal,Raise,Former,Latter,N,Type,End,Goal) :-
    Goal = fault | shrink(End,N), GSignal = [fault], Latter = [fault].

shrink(End,N) :-
%    message("Shrink [",[N,"]\n"]),
    End = [shrink(N)].

%%%% REBIRTH
% Wait until finishing to sweep out all goals on the fly.
% (1) goal is forwarded
watchdog_backup_rebirth(SC,C,R,GSignal,Raise1,Interrupt,
                        [Goal|Former],Latter,Alarm,Nodes,Weight,Total) :-
    watchdog_backup_rebirth_1(SC,C,R,GSignal,Raise1,Interrupt,
                              Former,Latter,Alarm,Nodes,Weight,Total,Goal).

watchdog_backup_rebirth_1(SC,C,R,GSignal,Raise1,Interrupt,
                          Former,Latter,Alarm,Nodes,Weight,Total,Goal) :-
```

```
    Goal = goal(Type,Module,Method,Arguments,Log) |
        generic:new(merge,{Raise1,Raise2},Raise),
        exgoal:call_goal(Type,Module,Method,Arguments,Log,
                         GSignal,Raise2,SC,SC1),
        watchdog_backup_rebirth(SC1,C,R,GSignal,Raise,Interrupt,
                        Former,Latter,Alarm,Nodes,Weight,Total).
% (2) neighbor node is shrinked
watchdog_backup_rebirth_1(SC,C,R,GSignal,Raise,Interrupt,
                          Former,Latter,Alarm,Nodes,Weight,Total,Goal) :-
    Goal = shrink(N) |
        watchdog_backup_rebirth(SC,C,R,GSignal,Raise,Interrupt,
                        Former,Latter,Alarm,[N|Nodes],~(Weight+1),Total).
watchdog_backup_rebirth_1(SC,C,R,GSignal,Raise,Interrupt,
                          Former,Latter,Alarm,Nodes,Weight,Total,Goal) :-
    Goal = rebirth(RT) |
        RT = [],
        send_ack(R,R1),
        ((wait(R1),
          inline:"%0=makeint(WATCHDOG_CYCLE);":[WATCHDOG_CYCLE-int]) ->

            ( GSignal = [rebirth|GSignal1],
              Interrupt = [fault],
              deadend(DeadEnd),
              Alarm = [set_alarm(WATCHDOG_CYCLE,C1)|Alarm1],
              watchdog_primary(SC,C1,DeadEnd,GSignal1,Raise,_,
                               Former,Latter,Alarm1,Nodes,Weight,Total)
        )   ).

send_ack([{N,Ack}|L],R) :- Ack = 0, send_ack(L,R).
send_ack([],R) :- R = 0.

watchdog_rebirth(SC,GSignal,Raise,Former,Latter,N,Type,End,Ack) :-
    wait(Ack) |

        message("REBIRTH (2) [",[N,"]\n"]),

        GSignal = [rebirth|GSignal1],
        rebirth(Type,NewType),
        watchdog(SC,GSignal1,Raise,Former,Latter,N,NewType,End).

deadend([alive|DeadEnd]) :-
    message("ALIVE\n"),
    deadend(DeadEnd).
deadend([fault|_]) :-
    message("!!! FAULT !!!\n"),
```

```
    unix:exit(-1).
deadend([]).

rebirth(primary,NewType) :- Newtype = backup.
rebirth(backup ,NewType) :- NewType = primary.
rebirth(original,NewType) :- NewType = original.

% -----
illegal_goal(Goal,N) :-
    klicio:klicio([stdout(normal(Out))]),
    variable:wrap(Goal, G),
    join(" is forwarded to node(",[N,")\n"],R),
    Out = [fwrite("Illegal goal "),
           putwt(G),fwrite(R),nl,fflush(_)].

message(S) :-
    klicio:klicio([stdout(normal(Out))]),
    Out = [fwrite(S),fflush(_)].

message(S,X) :-
    klicio:klicio([stdout(normal(Out))]),
    join(S,X,String),
    Out = [fwrite(String),fflush(_)].

join(S,[X|Y],R) :-
    (string(X,_,_) -> X = S1 ;
     integer(X)    -> integer_to_string(X,S1)
     ),
    generic:join(S,S1,S2),
    join(S2,Y,R).
join(S,[],R) :- R = S.

integer_to_string(I,S) :-
    deci_list(I,[],IL,0,N),
    generic:new(string,S,IL,8).

deci_list(X,T,XL,N,M) :- X <  10 |
        XL = [~(X + #"0")|T], N = M.
deci_list(X,T,XL,N,M) :- X >= 10 |
        Mod := (X mod 10)+ #"0",
        X1  := X / 10,
        deci_list(X1,[Mod|T],XL,~(N+1),M).
```

D.2 複製プロセスプログラム

```
:- module copy.

copy(Goal0,Goal1,Goal2)+In :-
    generic:new(merge,In,Out),
    unbound(Goal0,Goal),
    copy_term0(Goal,Goal1,Goal2)+Out.

copy_term0({_,_,Goal},Goal1,Goal2)+T :-  /* unbound */
        arbitrator(Goal,Goal1,Goal2)+T.
copy_term0({Goal}    ,Goal1,Goal2)+T :-
    copy_term(Goal,Goal1,Goal2)+T.

copy_term(Goal,Goal1,Goal2)+T :- atomic(Goal) |
        Goal1 = Goal, Goal2 = Goal.
copy_term([Car|Cdr],List1,List2)+T :-
    copy(Car,Car1,Car2)+T,
    copy(Cdr,Cdr1,Cdr2)+T,
    List1 = [Car1|Cdr1], List2 = [Car2|Cdr2].
copy_term(Goal,Goal1,Goal2)+T :- vector(Goal,L) |
        new_vector(G1,L),
        new_vector(G2,L),
        copy_vargs(Goal,G1,G2,0,L,Goal1,Goal2)+T.
copy_term(Goal,Goal1,Goal2)+T :- string(Goal,_,_) |
        Goal1 = Goal, Goal2 = Goal.
otherwise.
copy_term(Goal,Goal1,Goal2)+T :- functor(Goal,Func,Arity) |
        new_functor(G1,Func,Arity),
        new_functor(G2,Func,Arity),
        copy_args(Goal,G1,G2,1,Arity,Goal1,Goal2)+T.

copy_args(Goal,G1,G2,N,Arity,Goal1,Goal2)+T :- N =< Arity |
        arg(N,Goal,Arg),
        copy(Arg,Arg1,Arg2)+T,
        setarg(N,G1,Arg1,Go1),
        setarg(N,G2,Arg2,Go2),
        copy_args(Goal,Go1,Go2,~(N+1),Arity,Goal1,Goal2)+T.
otherwise.
copy_args(Goal,G1,G2,N,Arity,Goal1,Goal2)+T :-
    G1 = Goal1, G2 = Goal2.

copy_vargs(Goal,G1,G2,N,Size,Goal1,Goal2)+T :- N < Size,
    vector_element(Goal,N,Elem) |
        copy(Elem,Elem1,Elem2)+T,
```

```
        set_vector_element(G1,N,_,Elem1,Go1),
        set_vector_element(G2,N,_,Elem2,Go2),
        copy_vargs(Goal,Go1,Go2,~(N+1),Size,Goal1,Goal2)+T.
otherwise.
copy_vargs(Goal,G1,G2,N,Size,Goal1,Goal2)+T :-
    G1 = Goal1, G2 = Goal2.

arbitrator(X,X1,X2)+T :- wait(X)  | broadcast(X,X1,X2)+T.
arbitrator(X,X1,X2)+T :- wait(X1) | forward(X1,X,X2,1)+T.
arbitrator(X,X1,X2)+T :- wait(X2) | forward(X2,X,X1,2)+T.
alternatively.
arbitrator(X,X1,X2)+T0 :- wait(T0) |
% If the primary fail,
        (T0 = [fault|_]     -> X = X2;
% If the primary normaly terminate,
         T0 = [terminate|T] -> arbitrator(X,X1,X2)+_
         ).

broadcast(X,X1,X2)+T :- atomic(X) | X1 = X, X2 = X.
broadcast([Ax|Ay],B,C)+T :-
    B = [Bx|By], C = [Cx|Cy],
    arbitrator(Ax,Bx,Cx)+T,
    arbitrator(Ay,By,Cy)+T.
broadcast(X,X1,X2)+T :- vector(X,L) |
        vect_to_list(X,0,L,Elems),
        new_vector(V1,L), new_vector(V2,L),
        fillelems(V1,0,L,X1,Var1), fillelems(V2,0,L,X2,Var2),
        arbitrator(Elems,Var1,Var2)+T.
broadcast(X,X1,X2)+T :- string(X,_,_)  | X1 = X, X2 = X.
otherwise.
broadcast(X,X1,X2)+T :- functor(X,Func,Arity) |
        args_list(X,1,Arity)+Args+[],
        new_functor(F1,Func,Arity), new_functor(F2,Func,Arity),
        fillargs(F1,1,Arity,X1,Var1), fillargs(F2,1,Arity,X2,Var2),
        arbitrator(Args,Var1,Var2)+T.

args_list(X,N,Arity)-Args :- N =< Arity |
        arg(N,X,Arg),
        Args <= Arg,
        args_list(X,~(N+1),Arity)-Args.
otherwise.
args_list(X,N,Arity)-Args.

fillargs(F,N,Arity,X,Vars) :- N =< Arity |
        setarg(N,F,V,F1), Vars = [V|Vars1],
```

```
        fillargs(F1,~(N+1),Arity,X,Vars1).
otherwise.
fillargs(F,N,Arity,X,Vars) :- F = X, Vars = [].

vect_to_list(V,N,M,Elems) :- N < M |
        vector_element(V,N,E), Elems = [E|Elems1],
        vect_to_list(V,~(N+1),M,Elems1).
otherwise.
vect_to_list(V,N,M,Elems) :- Elems = [].

fillelems(V,N,M,X,Vars) :- N < M |
        set_vector_element(V,N,Var,V1), Vars = [Var|Vars1],
        fillelems(V1,~(N+1),M,X,Vars1).
otherwise.
fillelems(V,N,M,X,Vars) :- V = X, Vars = [].

forward(X1,X,X2,_)+T :- atomic(X1) | X = X1.
forward([Bx|By],A,C,P)+T :- A = [Ax|Ay], C = [Cx|Cy],
    forward1(Bx,Ax,Cx,P)+T, forward1(By,Ay,Cy,P)+T.
forward(X1,X,X2,P)+T :- vector(X1,L) |
        vect_to_list(X1,0,L,Elem1),
        new_vector(V,L), new_vector(V2,L),
        fillelems(V,0,L,X,Elems), fillelems(V2,0,L,X2,Elem2),
        forward1(Elem1,Elems,Elem2,P)+T.
forward(X1,X,X2,P)+T :- string(X1,_,_) | X = X1.
otherwise.
forward(X1,X,X2,P)+T :- functor(X1,Func,Arity) |
        args_list(X1,1,Arity)+Var1+[],
        new_functor(F,Func,Arity), new_functor(F2,Func,Arity),
        fillargs(F,1,Arity,X,Vars), fillargs(F2,1,Arity,X2,Var2),
        forward1(Var1,Vars,Var2,P)+T.

forward1(B,A,C,1)+T :- arbitrator(A,B,C)+T.
forward1(C,A,B,2)+T :- arbitrator(A,B,C)+T.
```