2015 年度 修士論文

ハイブリッド制約言語 HydLa 処理系の並列化による高速化

提出日 ： 2016 年 1 月 23 日
指導 ： 上田 和紀 教授

早稲田大学大学院 基幹理工学研究科
情報理工・情報通信専攻

学籍番号：5114F012-7
伊藤 剛史

概要

伊藤 剛史

時間の変化に伴う連続的な変化と離散的な変化の両方を含む動的システムをハイブリッド (ダイナミカル) システムと呼ぶ．ハイブリッドシステムは制御工学や生命工学などにおける幅広い分野への応用が可能である．例えば，自動車の衝突被害軽減ブレーキのシステムを考えたとき，前方の障害物との距離は基本的に連続的に変化し，運転手への警告の発生や自動ブレーキの動作などの状態は離散的に変化する．このようなシステムをハイブリッドシステムとしてモデリングすることで，その挙動のシミュレーション・検証が可能となるため，バグやミスを見つける際に有用となる．

HydLa は微分方程式を含む制約によりハイブリッドシステムをモデリングする宣言型言語である．HydLa の処理系である HyLaGI は，数式処理に基づく計算誤差のない高信頼なシミュレーションや，パラメータを含むシステムの記号実行シミュレーションを特徴とする．これは，浮動小数点演算に基づく既存のハイブリッドシステム処理系にはない特徴である．

本研究は，HyLaGI がより複雑な問題に対応できるよう，シミュレーション処理の並列化により高速化することを目的とする．現在，HyLaGI は数式処理バックエンドとして Mathematica を使用しており，Mathematica が提供する API により別プロセスで動作する Mathematica と通信し，数式処理を行っている．そこで，まず HyLaGI のバックエンドを並列化し，複数の Mathematica プロセスと同時に通信し，数式処理を行えるようにした．この際，Mathematica プロセスの立ち上げや通信を並列に行うことで，1 プロセスで動作するときと比較したオーバーヘッドを抑えた．その後，この並列化バックエンドを使用して，多数の条件式の判定が行われる処理や解軌道が分岐した際のシミュレーションをスレッドにより並列に実行できるようにし，シミュレーションの高速化を目指した．

結果として，条件式の判定はスレッド数に応じて高速化したことが確認でき，特に条件式の個数がスレッド当たり 500 個を超える様な問題ではスレッド数倍の性能が得られた．また，解軌道が分岐した際のシミュレーションの並列化によりシミュレーション時間の短縮に成功し，最大で 17 スレッド時に 4.47 倍まで高速化された．

Abstract

Takeshi Ito

A hybrid (dynamical) system is a dynamical system that shows continuous and discrete behavior. Hybrid systems are applicable in a wide range of areas including control engineering and biotechnology. For example, in collision avoidance systems for cars, distances from obstacles are basically continuous values, while conditions of warning and automatic brake change discretely. We can simulate and verify the behaviors of such systems by modelling them as hybrid systems.

HydLa is a constraint-based declarative programming language for hybrid systems. Constraints are equations that define the behavior of the values of variables. HyLaGI is an implementation of HydLa and is characterized by highly dependable simulation based on (i) formula manipulation that is free from errors caused by floating-point arithmetic, and (ii) symbolic execution of systems with parameters. These features are different from the other hybrid system simulators.

The purpose for this research is parallelize simulation of HyLaGI to handle more complex or larger models. HyLaGI manipulates formulas by Mathematica running on a separate process via the API provided by Mathematica. We parallelized the backend of HyLaGI that communicates with Mathematica to manipulate formulas in parallel. We also parallelized start-up and communication task for Mathematica to reduce the overheads of using multiple Mathematica processes. Then, we parallelized the evaluation of many guard constraints and the simulation of branched trajectories by multithreaded backends.

As a result, the evaluation of guard constraints was speeded up with the number of threads. In particular, for a model that produced more than 500 equations per thread, the performance of the process was speeded up scalably with the number of threads. Parallelized simulations of branched trajectories speeded up the entire simulation time up to 4.47 times with 17 threads.

目次

図目次

第 1 章

はじめに

1.1　研究の背景と目的

近年，コンピュータによって制御される自動車や航空機などの研究・製品化が活発に行われている．これらのシステムはセンサの入力からコンピュータによって判断が行われ，アクチュエータなどを動作させるような，実世界のデバイスとソフトウェアとの相互作用によって制御される．このようなシステムの挙動のシミュレーションや検証は今後より重要な課題になると予想される．

時間の変化に伴う連続的な変化と離散的な変化の両方を含む動的システムをハイブリッド (ダイナミカル) システム [1] と呼ぶ．ハイブリッドシステムは制御工学などにおける幅広い分野への応用が可能である．例えば，自動車の衝突被害軽減ブレーキのシステムを考えたとき，前方の障害物との距離は基本的に連続的に変化し，運転手への警告の発生や自動ブレーキの動作などの状態は離散的に変化する．このようなシステムをハイブリッドシステムとしてモデリングすることで，その挙動のシミュレーション・検証が可能となるため，バグやミスを見つける際に有用となる．

HydLa[2] は微分方程式を含む制約によりハイブリッドシステムをモデリングする宣言型言語である．HydLa の処理系である HyLaGI[7] は，数式処理に基づく計算誤差のない高信頼なシミュレーションや，パラメータを含むシステムの記号実行シミュレーションを特徴とする．これは，浮動小数点演算に基づく既存のハイブリッドシステム処理系にはない特徴である．

本研究は，HyLaGI がより複雑な問題に対応できるよう，シミュレーション処理の並列化により高速化することを目的とする．これにより，HyLaG が自動車の衝突被害軽減ブレーキの設計のような，より現実的・実用的な用途で応用されることを目指す．現在，

HyLaGI は数式処理バックエンドとして Mathematica[15] を使用しており，Mathematica が提供する API により別プロセスで動作する Mathematica と通信し，数式処理を行っている．そこで，まず HyLaGI のバックエンドを並列化し，複数の Mathematica プロセスと同時に通信し，数式処理を行えるようにする．その後，この並列化バックエンドを使用して，多数の条件式の判定が行われる処理や解軌道が分岐した際のシミュレーションをスレッドにより並列に実行できるようにし，シミュレーションの高速化を目指す．

1.2　論文構成

まず 2 章では，ハイブリッドシステムモデリング言語である HydLa と，その処理系 HyLaGI について説明を行う．次に 3 章で，HyLaGI のシミュレーション速度に関する課題と，並列化の可能性について述べる．さらに 4 章で，3 章で挙げた並列化の実装方法についての説明を行う．5 章では，4 章で述べた実装によって，どのような改善が見られたか，例題を用いて性能評価を行い考察する．最後に 6 章で本論文をまとめ，今後の課題を述べる．

第 2 章

HydLa および HydLa 処理系 HyLaGI

本章では，ハイブリッドシステムやそのモデリング言語 HydLa，及び，その処理系である HyLaGI について解説する．

2.1　ハイブリッドシステム

時間の変化に伴う連続的な変化と離散的な変化の両方を含む動的システムをハイブリッドシステム [1] またはハイブリッドダイナミカルシステムと呼ぶ．制御工学や生命工学などの幅広い分野における問題をハイブリッドシステムとしてモデリングすることが可能であり，各々のシステムの挙動のシミュレーションや検証に役立つ．

2.1.1　ハイブリッドシステムの例

ハイブリッドシステムとしてモデル化できる例をいくつか取り上げる．

バウンシングボール

単純な物理学のモデルの例として，ある高さから落としたボールが床と衝突し，弾む様子をハイブリッドシステムとしてモデル化することを考える．モデルを単純化するために，ボールは体積を持たない質点とする．そのため，ボールには重力と床に衝突した際の反発力のみが働く．時刻 t におけるボールの高さを $y(t)$，ボールの高さの初期値を y_0 とすると，初期状態は以下のように表せる．

$$y(0) = y_0,\ y'(0) = 0$$

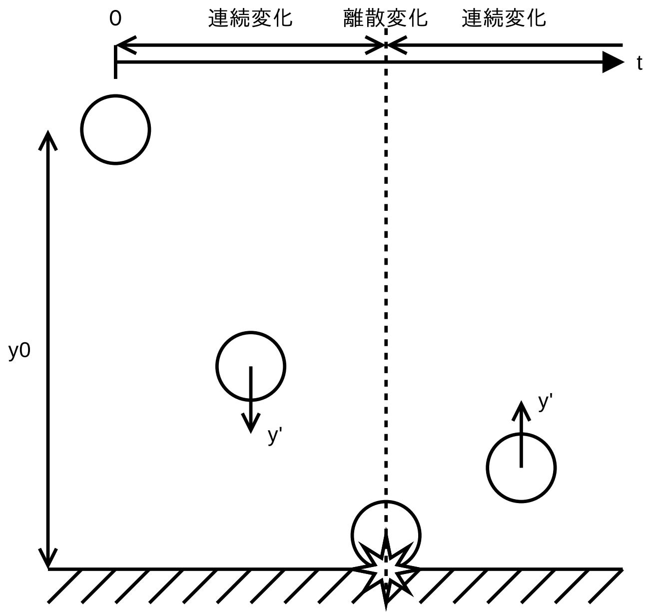

図 2.1　バウンシングボールのモデル

このモデルにおいて，ボールには常に床へ向けた重力加速度が生じる．これは連続的な変化である．重力加速度を g とすると，数式で以下のように記述できる．

$$y''(t) = g$$

ボールが床に衝突した際，ボールと床の反発係数に応じてボールが反発する．これを離散的な変化として捉えると，モデル化が容易になる．ボールと床との衝突の際の反発係数を e，衝突直前の時刻を t_{prev} とすると，衝突直後のボールの速度は漸化式で以下のように記述できる．

$$y'(t) = -ey'(t_{\mathrm{prev}})$$

このモデルの時間の経過に対する変化の様子を図 2.1 に示す．

このように，連続的な変化と離散的な変化の両方を含むハイブリッドシステムとしてモデル化することで，数式で容易に記述することができる．

ビリヤード

次に，ビリヤードのモデルを考える．このモデルでは端に壁のある空間上にボールが並んでおり，1 番目のボールに初速度を与え，他のボールに衝突させる．単純化のためボー

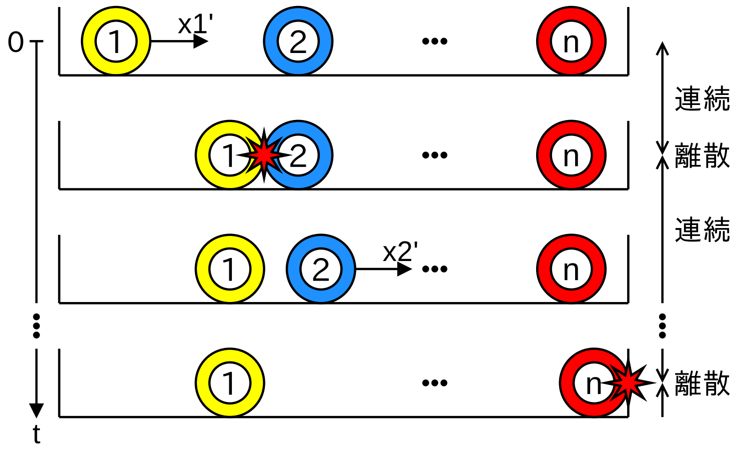

図 2.2　ビリヤードのモデル

ルは大きさを持たず，すべて同じ重さとする．また，発生する衝突はすべて完全弾性衝突であるとする．このモデルを図 2.2 に示す．

時刻 t における n 番目のボールの位置を $x_n(t)$ と表すと，初期状態として各ボールの初期位置 $x_n(0)$ と，1 番目のボールの初速度 $x_1'(0)$ が与えられる．

すべてのボールは慣性により，常に等速運動をする．これは連続的な変化として以下のように表せる．

$$x''(t) = 0$$

ボールが他のボールと衝突した際，完全弾性衝突が発生する．すべてのボールの重さが同じであるという仮定のもとでは，速度の交換が発生する．この衝突を離散的な変化としてモデル化する．衝突の直前の時刻を t_{prev} とすると，n 番目のボールと $n+1$ 番目のボールの衝突は以下のように表せる．

$$x_n'(t) = x_{n+1}'(t_{\mathrm{prev}})$$
$$x_{n+1}'(t) = x_n'(t_{\mathrm{prev}})$$

また，ボールと壁とが衝突した際，ボールの速度が反転する．これも離散的な変化として，以下のように表せる．

$$x'(t) = -x'(t_{\mathrm{prev}})$$

このように，ビリヤードも連続的な変化と離散的な変化の両方を含む，ハイブリッドシステムとしてモデル化することができる．

2.2　HydLa

HydLa[2] は微分方程式を含む制約によりハイブリッドシステムをモデリングする宣言型言語である．HydLa では，一般的に使用する数式とほぼ同様の形式で制約を記述する．また，制約間の優先順位を記述する制約階層 [3] の概念の導入により，ハイブリッドシステムのモデルを簡潔に記述できる [4]．HydLa によって記述されたモデルを HydLa プログラムと呼ぶ．

2.2.1　構文

本論文で扱う HydLa の構文は文献 [5] に示されたものである．これは，文献 [2] に記述されている構文にリスト記法 [6] が導入されたものである．この構文を図 2.3 に示す．

$$
\begin{array}{rlll}
\text{(hydla program)} & H & := & (\ DF\,\texttt{.} \mid DC\,\texttt{.}\ )\texttt{*} \\
\text{(declaration)} & DC & := & M \mid PL \mid PL[E] \mid DC, DC \mid DC \texttt{<<} DC \mid (\ DC\ ) \\
\text{(definition)} & DF & := & MPname(\vec{X})\{MP\} \mid Cname(\vec{X})\ \texttt{<=>}\ C \\
 & & \mid & ELname\ \texttt{:=}\ EL \mid PLname\ \texttt{:=}\ PL \\
\text{(list condition)} & LC & := & MPname\ \text{in}\ PL \mid Variable\ \text{in}\ EL \mid E\ \texttt{=!=}\ E \\
\text{(list size)} & LS & := & \mid (PL \mid EL) \mid \\
\text{(priority list)} & PL & := & \{MP\ (,\ MP)*\} \mid \{MP \mid LC\ (,\ LC)*\} \mid PLname \\
\text{(module priority)} & MP & := & M \mid MPname(\vec{E}) \mid MP,MP \mid MP\ \texttt{<<}\ MP \mid (\ MP\ ) \\
\text{(module)} & M & := & C \\
\text{(constraint)} & C & := & A \mid Cname(\vec{E}) \mid C\ \texttt{/\char`\\}\ C \mid G\ \texttt{=>}\ C \mid \texttt{[]}\ C \mid (\ C\ ) \\
\text{(guard)} & G & := & A \mid G\ \texttt{/\char`\\}\ G \mid G\ \texttt{\char`\\/}\ G \mid (\ G\ ) \\
\text{(atomic constraint)} & A & := & E\ (\ relop\ E\ )\texttt{+} \\
\text{(expression list)} & EL & := & \{E\ (,\ E)*\} \mid \{E \mid LC\ (,\ LC)*\} \mid ELname \\
 & & \mid & \{RE\,\texttt{..}\,RE\} \\
\text{(range expression)} & RE & := & \text{変数を含まない式} \mid \text{名前が数字で終わる変数} \\
\text{(expression)} & E & := & \text{通常の式} \mid E\texttt{'} \mid E\texttt{\^{}}E \mid E\texttt{-} \mid EL[E]
\end{array}
$$

図 2.3　HydLa の構文 [5]

HydLa プログラムは図 2.3 の (hydla program) の構文で記述される．

制約の優先順位は (declaration) で定義される構文で記述される．記号 `<<` は制約の優先関係を示す記号であり，A<<B と記述することで制約 B が制約 A より優先されることを示す．

$CName$ は定義された制約の名前を表す．引数 $\vec{X}$ は束縛変数の列である．

HydLa プログラムにおける変数はすべて時刻 t の関数である．

(priority list) は制約の優先度定義を要素とするリストであり，(expression list) は式を要素とするリストである．また，$PLName$ 及び $ELname$ はそれぞれのリストの名前である．リストは $\{\texttt{INIT}(i) \mid i \text{ in } \{1,2,3,4\}\}$ のような記法で記述され，これは $\{\texttt{INIT}(1), \texttt{INIT}(2), \texttt{INIT}(3), \texttt{INIT}(4)\}$ と同義である．リストの要素へのアクセスは $L[E]$ のような記法を用いる．これは，リスト L の E 番目の要素にアクセスする．リストの要素数は $|L|$ のように記述することで取得できる．

リスト記法の導入によってオブジェクト数 n のモデルを HydLa プログラムで記述した場合の記述量が最大で $O(n^2)$ から $O(1)$ に削減され，多数のオブジェクトを含むモデルなどの記述が容易となった．

2.2.2　HydLa によるモデリング

2.1.1 節で挙げた例を HydLa を用いてモデリングする．

バウンシングボール

HydLa によって記述されたバウンシングボールのモデルを図 2.4 に示す．

この HydLa プログラムでは，INIT，FALL，BOUNCE の 3 つの制約が宣言されている．INIT は初期状態を示す制約で，ボールの高さとして $y = 10$ かつ，初期速度として $y' = 0$ であることを表す．FALL は重力による落下を示す制約で，ボールに常に ([ ]) 重力加速度が生じる ($y'' = -10$) ことを表す．BOUNCE は床との反発を示す制約で，常に ([ ])y の左極限値が 0 となったとき ($y- = 0$)，ボールの速度に $-4/5$ が乗算される ($y' = 4/5 * y'-$) ことを表す．

5 行目ではこれら 3 つの制約の優先関係が宣言されており，BOUNCE が FALL より優先されることを表す．

実際にこのプログラムを実行すると，ボールが空中にある場合は 3 つすべての制約が採用される．また，ボールが床と衝突したとき ($y- = 0$) に FALL と BOUNCE の矛盾が発生するが，5 行目の優先関係に基づき INIT と BOUNCE が極大かつ無矛盾な制約の集

```
INIT <=> y = 10 & y’ = 0.
FALL <=> [](y’’ = -10).
BOUNCE <=> [](y- = 0 => y’ = -4/5 * y’-).

INIT, FALL << BOUNCE.
```

図 2.4　HydLa によるバウンシングボールのモデル

合として採用される．

ビリヤード

HydLa によって記述されたビリヤードのモデルを図 2.5 に示す．

この HydLa プログラムではリスト記法が使用されている．1 行目において `X` が `x0` から `x20` までの変数を含む集合であると定義している．次に 4 から 7 行目にかけて初期条件 `INIT`，ボール同士の衝突を表す制約 `COL`，ボールの等速運動を表す制約 `CONST`，ボールと壁との衝突を表す制約 `WALL` が定義されている．10 から 16 行目にかけては `X` の要素に対して 4 から 7 行目の制約の定義を適用するための制約が定義される．最後に 18 行目で 1 つ目のボール (`X` の 1 つ目の要素) に初速度を与える初期条件が宣言され，19 行目で 10 から 16 行目で定義された制約が宣言される．

実際にこのプログラムを実行すると，ボール同士またはボールと壁の衝突時以外はすべての制約が矛盾せずに採用される．ボールの衝突が発生する際，優先度リスト `COL_PRIORITIES` と `WALL_PRIORITIES` に基づき，各ボールの等速運動を表す制約 `CONST` を除いた制約集合が採用され，各ボールの速度変化が発生する．

このように，リスト記法を利用することで多数のオブジェクトを含むモデルを少ない記述量で表すことができる．

```
X := {x0..x20}.

// constraints
INIT(b,b0,vb0) <=> b=b0 & b'=vb0.
COL(b1,b2) <=> [](b1-=b2- => b1'=b2'- & b2'=b1'-).
CONST(b) <=> [](b''=0).
WALL(b) <=> [](b-=-1 | b-=2*|X| => b'=-b'-).

// program lists
INITS := { INIT(X[i],2*i-2,0) | i in {2..|X|} }.

COL_PRIORITIES :=
 { ( CONST(X[i]), CONST(X[j]) ) << COL(X[i], X[j]) |
                     i in {1..|X|-1}, j in {i+1..|X|} }.
WALL_PRIORITIES :=
 { CONST(X[i]) << WALL(X[i]) | i in {1..|X|}}.

INIT(X[1],0,1),
INITS, COL_PRIORITIES, WALL_PRIORITIES.
```

図 2.5　HydLa によるビリヤードのモデル

2.3　HyLaGI

HyLaGI[7] は HydLa の処理系であり，数式処理に基づく計算誤差のない高信頼なシミュレーションや，パラメータを含むシステムの記号実行シミュレーションを特徴とする．これは，浮動小数点演算に基づく既存のハイブリッドシステム処理系にはない特徴である．

2.3.1　シミュレーションのプロセス

HyLaGI におけるシミュレーションのアルゴリズムについて，文献 [7] から引用して図 2.6 に示す．

HyLaGI では，離散変化を計算する Point Phase(PP) と連続変化を計算する Interval Phase(IP) を繰り返しながらシミュレーションが進行する．図 2.6 においては 6 から 12 行目が PP，13 から 20 行目が IP の計算となる．

本研究が対象とするガード条件判定は $CalculateMCS$ に含まれる処理である．

2.4　関連研究

関連研究として，他のハイブリッドシステムのシミュレータについて説明する．

2.4.1　Acumen

Acumen[8] はサイバーフィジカルシステムのモデルのシミュレーションと可視化を目的とした言語及び処理系である．HyLaGI と同様にハイブリッドシステムのモデリングに対応している．

HyLaGI と異なり数値計算に基づくシミュレーションが行われ，区間演算を用いた精度保証が行われている．また，モデルの記述に手続き型言語を用いる．

2.4.2　Flow∗

Flow∗[9] はハイブリッドオートマトンの検証を目的としたツールである．ハイブリッドシステムなどのモデルをハイブリッドオートマトンで記述し検証できる．

HyLaGI と異なりハイブリッドオートマトンでモデルを記述し，また数値計算に基づく

```
Input: HydLa プログラム HydLaProgram，シミュレーション終了時刻 MaxT
MS := TopologicalSort(SolveCH(HydLaProgram))
M_all := MaxModuleSet(MS)
V := GetVariables(HydLaProgram)
T := 0; S := true; CP := true; E := ∅
while T <_CP MaxT do
  // PP
  S := SubstituteMinTime(S,T)
  (S,CP,E,_,_,_) := CalculateMCS(S,MS,E,CP,T,CheckConsistencyPP)
  if S = false then
    break
  end if
  (S,CP) := AddParameters(S,CP,V)
  // IP
  (S,CP,E,M,A+,A-) := CalculateMCS(S,MS,E,CP,T,CheckConsistencyIP)
  S := SolveDifferentialEquation(S)
  if S = false ∨ ¬ IsUnique(S,V) then
    break
  end if
  (MinT,CP) := GetElement(CompareMinTime(
    {FindMinTime(S ∧ CP ⇒ g) | (g ⇒ c) ∈ A-}
    ∪ {FindMinTime(S ∧ CP ⇒ ¬ g) | (g ⇒ c) ∈ A+}
    ∪ {FindMinTime(S ∧ CP ∧ M-) |
                              M- ∈ (M_all \ M)}
    ∪ {FindMinTime(S ∧ CP ∧ ¬ M+) | M+ ∈ M}
    ∪ {(MaxT-T,true)}))
  T := MinT+T
end while
```

図 2.6　HyLaGI のシミュレーションの非決定実行アルゴリズム [7]

シミュレーションが行われ，区間演算を用いた精度保証が行われている．

2.4.3 Hybrid CC

HybridCC[10] はハイブリッドシステムを記述するための制約に基づくモデリング言語である．HydLa と同様に制約を用いており，ハイブリッドシステムの簡潔な記述を可能にしている．インタープリタの処理系が実装され公開されている．

HyLaGI と異なり数値計算に基づくシミュレーションが行われ，計算結果の精度保証は行われない．

2.4.4 KeYmaera

KeYmaera[11] はハイブリッドシステムのモデルについての検証器である．到達可能性や活性などの性質について検証可能であり，HyLaGI と同様に数式処理による計算によりパラメータを含むモデルの記号実行が可能である．

HyLaGI と異なりシミュレータではなく検証を目的としている．そのため，HyLaG が行うような変数の解軌道の導出は行わず，モデルがある性質を満たすかどうかの検証に特化している．また，モデルの記述に独自の手続き型言語を用いる．

2.4.5 SpaceEx

SpaceEx[12] は連続システムとハイブリッドシステムの検証を目的としたツールである．複数の検証アルゴリズムを実装可能としており，現在 2 つのアルゴリズムが実装されている．

HyLaGI と異なりハイブリッドオートマトンでモデルを記述する．また，精度保証が行われるかどうかは検証アルゴリズム次第であり，現在実装されている 2 つのアルゴリズムの内 1 つは精度保証が行われない．

第 3 章

HyLaGI の並列化

本章では，HyLaGI のシミュレーション速度における課題と，並列化の可能性について説明する．

3.1　数式処理

3.1.1　数式処理シミュレーションの短所と対策

2 章において述べた通り，HyLaGI は数式処理に基づくシミュレーションを特徴とする．これにより，数値計算に基づくシミュレータと比較して，計算誤差がない，記号実行が可能であるなどの特徴を実現している．しかし，数式処理に基づくシミュレーションは速度の面において，数値計算に対して劣る．主な原因として，変数値を表す数式がシミュレーションが進むにつれて長くなることが挙げられる．

例として，2.1.1 節で挙げたバウンシングボールのモデルで説明する．バウンシングボールのモデルでは，ボールが床と衝突するごとにボールの速度を示す変数 y' に床との反発係数 e が乗算され，y を表す式が長くなる．

より複雑な数式による変化を伴うモデルでは，シミュレーションが進むにつれて数式がより複雑化し，バックエンドとの通信量やメモリ使用量，処理時間が増大していく．

この問題への対策として，数式処理と区間演算を組み合わせたシミュレーション手法が考案・実装され，計算精度を保証しながら処理時間を抑えることに成功した [13]．

3.1.2　数式処理バックエンドの並列実行

HyLaGI では，数式処理のためのバックエンドとして Wolfram Research の Mathematica[15] を使用している．Mathematica のエンジンである MathKernel は HyLaGI とは別のプロセスで実行され，Mathematica が提供する外部プログラム向けの API(MathLink) により通信を行う．この通信には，Linux/Unix 環境においては TCP/IP，共有メモリ，パイプがサポートされるが [16]，HyLaGI ではパイプが使用される．

数式処理を並列に行う方法としては，複数の MathKernel プロセスを起動してそれぞれと個別に通信を行う方法と，Mathematica のマルチスレッド機能を使用する方法とが考えられる．前者では現在 HyLaGI が使用している Mathematica の命令がそのまま利用できるのに対し，後者ではスレッドを使用するための命令に変更する必要がある．また，前者では MathKernel プロセスを立ち上げる時間がオーバーヘッドとなるが，HyLaGI ではシミュレーション開始前に立ち上げられた MathKernel プロセスがプログラム終了時まで利用されるため，起動時の立ち上げを並列に実行することで 1 プロセスの場合と比べたオーバーヘッドを抑えることができる．

そのため，本研究では複数の MathKernel プロセスを立ち上げる方法で数式処理バックエンドの並列化を行った．

3.2　ガード条件判定

3.2.1　HyLaGI におけるガード条件判定

HydLa では，特定の条件を満たしたときに有効になる制約をガード条件を使用することで記述できる．HyLaGI は HydLa プログラムで宣言されているガード条件付きの制約について，各フェーズで成立・非成立となる時刻を計算する．これは，ガード条件が増えるとそれに応じて判定が必要となるガード条件が増加することを示す．

例えば，2.1.1 節で挙げたビリヤードの問題では，ボールの数を n 個とすると任意の 2 個のボールの衝突判定が必要となるため，${}_nC_2$ 個のガード条件が必要となる．さらに各ボールに 2 つの壁との衝突判定が存在するため，$2n$ 個のガード条件判定が加わる．よって，ボールの数が増加すると $O(n^2)$ でガード条件の個数と計算量が増加する．これは，道路を走行している複数台の車の運転のシミュレーションなど，多数のオブジェクトが存在する問題においてガード条件判定が大きなボトルネックとなる事を示し，ビリヤードの例

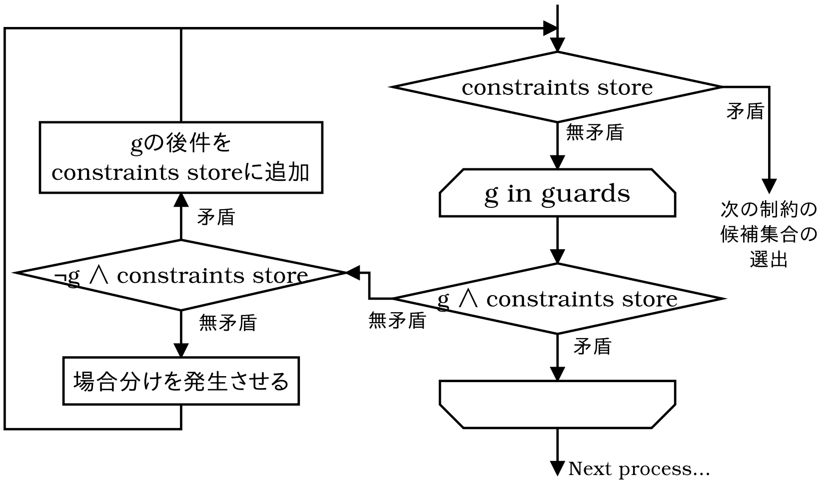

図 3.1　HyLaGI のガード条件判定のフローチャート

ではガード条件判定が全体の実行時間の 9 割を超える時間を占める．

この対策として，HydLa へのリスト記法の導入と同時に変数の連続性を利用した判定処理の省略が行われた [6]．これは，変数の連続性から再計算が不要な制約に関するガード条件判定を省略するものである．また，直前のフェーズにおける情報を再利用し，必要な差分のみを再計算することによる高速化も行われた [14]．これらの手法により，ビリヤードの問題において要するガード条件判定の時間は，最初のフェーズを除いて $O(1)$ となった．ただし，これらの手法を用いても，最初のフェーズの判定は省略することが不可能であり，この判定の高速化が課題である．

3.2.2　ガード条件判定のフローチャート

図 3.1 にガード条件判定のフローチャートを示す．図中の constraints store は制約ストアを示し，ある時点で成立している制約 (前提条件) の集合である．また，gurds は判定の対象となるガード条件の集合である．

まず最初に，constraints store の無矛盾性判定が行われる．この判定により矛盾が発見されると，制約階層に基づき次の制約の候補集合が選出される．例えばビリヤードのモデルにおける，ボールの等速直線運動と衝突の制約の矛盾がここで発見される．

次に，集合 gurds に含まれるすべてのガード条件について，constraints store との無矛盾性判定が行われる．gurds のある要素を g とすると，まず g と constraints store が矛盾するかどうかが判定される．この判定が矛盾する場合，g は constraints store の条件下で一切成立しない事を示す．

```
SWITCH <=> [](x < 10 => s = 1).
BRAKE <=> [](s = 1 => x’’ = -1).
```

図 3.2　ガード条件が連鎖的に成立する例

$g \wedge$ constraints store の判定が無矛盾となった場合は，constraints store の条件下で常に g が成立するのか，あるいは一部だけで成立するのかを判定するために，g の否定と constraints store との無矛盾性判定が行われる．この判定が矛盾した場合，g は constraints store の条件下で常に成立することを示し，g の後件が constraints store に追加される．無矛盾となる場合は constraints store の条件下で g が成立する場合としない場合の場合分けが発生し，解軌道が分岐する．

3.2.3　ガード条件判定の並列化

この判定では，gurds に含まれるすべてのガード条件に対してループ処理で判定が行われるが，ここでガード条件同士に依存関係がなければ，複数のガード条件に対して並列に判定を行なって良いと考えられる．ガード条件同士で依存関係が生じる例としては，あるガード条件が成立しその後件が constraints store に加えられた結果として，他のガード条件が成立する場合が考えられる．

例えば，図 3.2 のようなガード条件付き制約が考えられる．制約 SWITCH のガード条件 $x < 10$ が成立し後件の $s = 1$ が constraints store に加えられると，それにより制約 BRAKE のガード条件も成立する．

しかし，このような依存関係が存在する場合でも，g の後件が constraints store に加えられた後に改めて最初からガード条件判定が行われるため，最終的に成立するべきガード条件はすべて発見される．これは，HydLa が制約が宣言する順序に依らずにモデルを記述できる言語であり，どのような順序で宣言しても正しく実行するためである．

結果的に，この判定は問題なく並列化できることが判明した．

この判定を並列化した場合には，スレッド数に対して十分な判定を行うべきガード条件が存在すると仮定して，スレッド数倍に高速化されることが期待できる．

3.3　ケース分岐

3.3.1　HyLaGI におけるケース分岐

HyLaGI において，記号実行による解軌道の分岐をケース分岐と呼ぶ．ケース分岐は IP と PP の両方で発生する可能性がある．HyLaGI は各フェーズをノードとした木を構築しながらシミュレーションを行い，ケース分岐が検出されると木も枝分かれする．この探索は深さ優先探索で行われる．

ケース分岐が多数発生するモデルの場合，すべてのケースにおいて状態変化が起きなくなるか，指定された深さまで探索されるまでシミュレーションが行われるため，シミュレーション時間が長くなる．

3.3.2　分析対象とするモデル

ケース分岐の挙動の分析のために，本論文では床に穴が開いたバウンシングボールのモデルを使用する．このモデルでは，2 次元空間でのバウンシングボールにおいて，床の一定の区間 $(7 \leq x \leq 10)$ が凹型に窪んでおり，またボールに横方向の初速度をパラメータ付き $(1 \leq x' \leq 5)$ で与える．このモデルではボールの穴への落ち方や壁への衝突の仕方でケース分岐が発生する．このモデルと解軌道の模式図を図 3.3 に，HydLa プログラムを図 3.4 に示す．

図 3.5 に，床に穴が開いたバウンシングボールにおけるケース分岐の様子を示す．この探索木は深さ優先探索により一番左の枝から順に探索され，構築される．このモデルでは，最終的に 33 通りまでケース分岐が発生し，フェーズ数は 135 フェーズとなった．

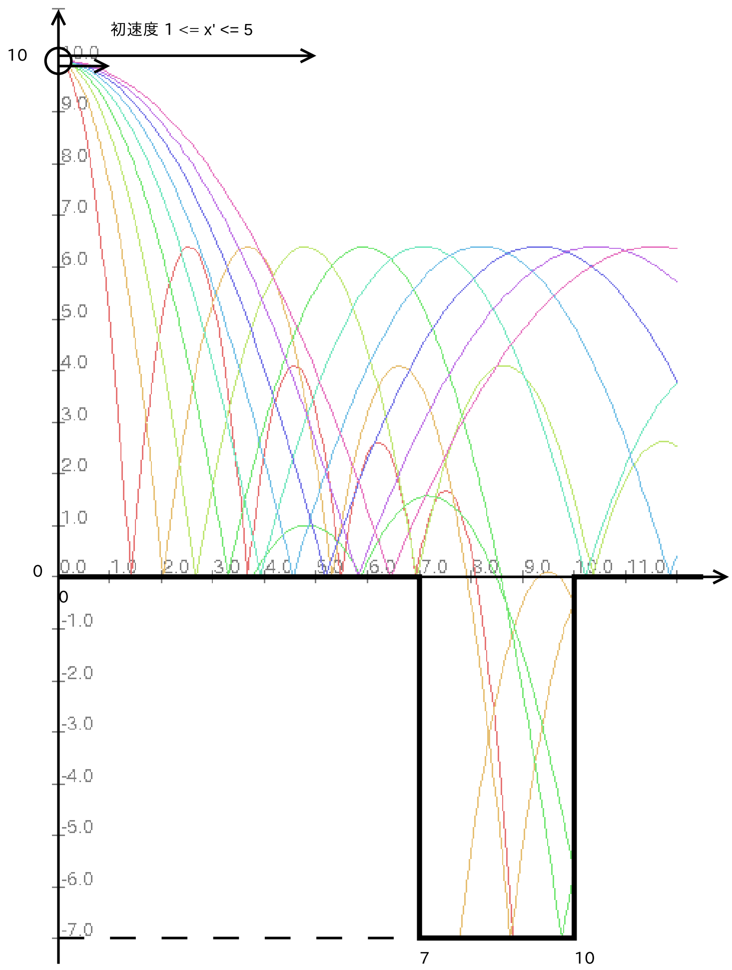

図 3.3　穴あきの床におけるバウンシングボール

```
INIT <=> y = 10 /\ y' = 0 /\ x = 0 /\ 1 <= x' <= 5.
FALL <=> [](y'' = -10).
XCONST <=> [](x'' = 0).
XBOUNCE <=> []((x- = 7 \/ x- = 10) /\ y- < 0 => x' = -x'-).
BOUNCE <=> [](y- = -7 \/ (x- <= 7 \/ x- >= 10) /\ y- = 0 => y'
    = -(4/5) * y'-).

INIT, FALL << BOUNCE, XCONST << XBOUNCE.
```

図 3.4　床に穴が開いたバウンシングボールの HydLa プログラム

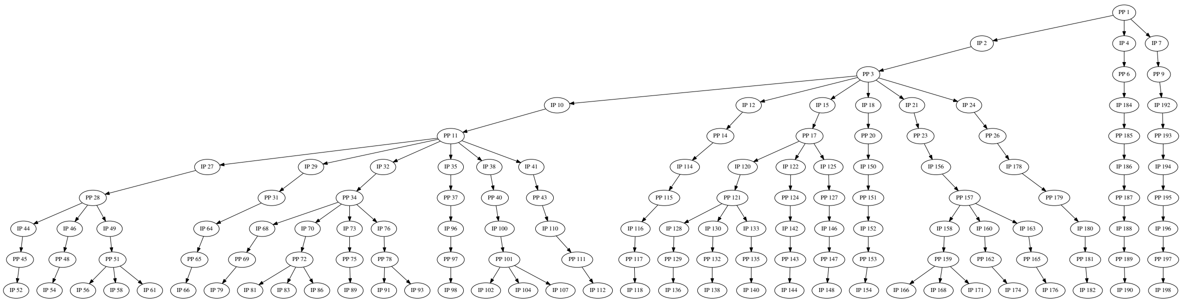

図 3.5　穴あきの床におけるバウンシングボールのケース分岐の様子

3.3.3　ケース分岐が発生した際のシミュレーションの並列化

ケース分岐が発生すると，分岐した解軌道同士に依存関係がないため，それぞれのケースのシミュレーションを並列に行うことが可能であると考えられる．実際には，分岐するフェーズの直後のフェーズ同士には依存関係が存在している．これは，解軌道の式が同様なフェーズの再計算を行わないことで高速化されているためである．

モデルのケース分岐が枝ごとに偏りなく発生し，各ケースのシミュレーションが理想的に並列実行され，CPU の数が分岐するケースの数だけ十分にあるとすれば，1 ケースのシミュレーションにかかる時間と同様の時間ですべてのケースのシミュレーションが実行できると考えられる．

穴あきの床におけるバウンシングボールのモデルでは，図 3.5 よりケース分岐に偏りが有り，シミュレーションが 10 フェーズ分の時間で完了できるとは考えづらい．

第 4 章

並列化の手法・実装方法

この章では，3 章において挙げた並列化の実装方法について説明する．

4.1　並列化の手法

本研究では，並列化プログラミングの方法として，2011 年に ISO によって標準化されたプログラミング言語 C++ の規格である C++11 に追加された，スレッディングライブラリを使用した．標準規格に基づくライブラリを使用することで，様々な OS・コンパイラで動作できる．このライブラリに含まれる機能は，スレッドの起動と終了の待機，ミューテックスなどの排他制御や同期に関する機能といった，スレッドプログラミングを行うための基本的なものであり，スレッドプールのような高度な機能は含まれていない．実質的に POSIX に規定されている pthread のラッパーライブラリであると言える．

4.2　数式処理バックエンドの並列化

並列化の方針として，まず数式処理バックエンドの並列化を行った後，このバックエンドを利用してガード条件判定とケース分岐時のシミュレーションの並列化を行う．

既存の HyLaGI のプログラムの構成図を図 4.1 に示す．HyLaGI において，数式処理バックエンドは Backend クラスによって管理される．このクラスのインスタンスはシミュレーション開始前に MathKernel プロセスと共に生成され，プログラム終了まで MathKernel との通信を管理する．そのため，このインスタンスを複数生成することで複数プロセスの MathKernel を使用することができる．

MathKernel プロセスの立ち上げは概ね 1 秒程度かかる処理であり，多数のプロセスを

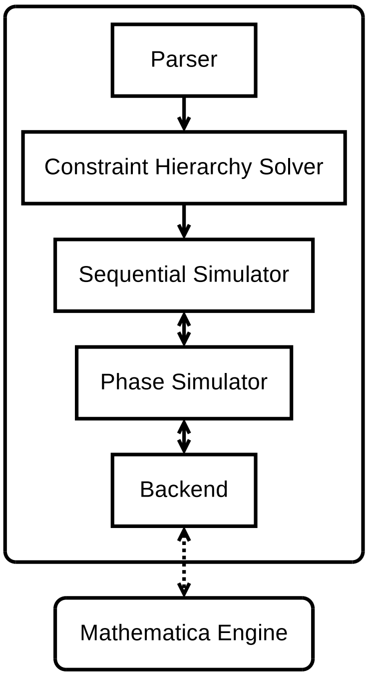

図 4.1　HyLaGI の構成図

立ち上げようとする場合は大きなオーバーヘッドとなる．そのためこの処理を並列に行うよう実装し，使用するバックエンドの数を増やした場合でも立ち上げ時間がオーバーヘッドとなることを防いだ．

実際に複数のバックエンドを使用した並列計算は，各々の計算を行う箇所ごとに実装した．より高度に抽象化された方法として複数のバックエンドと計算タスクを管理するクラスを実装し，計算結果を非同期的に返すスレッドプールの様な仕組みを実装することで，実際の計算を行う箇所で複数のバックエンドの存在を意識せずに並列計算が行えると考えられる．しかし，HyLaGI では各 MathKernel プロセスに事前にフェーズの前提条件などを送信・設定する必要があり，スレッドプールを実装した際にこれらを効率的に行うことは困難であると考えられたため，これらの実装は行わなかった．

4.3　ガード条件判定の並列化

HyLaGI のガード条件判定は図 4.1 における，Phase Simulator 内で行われる．このクラスは，シミュレーションにおける各フェーズでの計算処理を担う．

並列化されたガード条件判定のフローチャートを図 4.2 に示す．これは，既存の HyLaGI のガード条件である図 3.1 での，ガード条件判定を行うループを並列化し，また

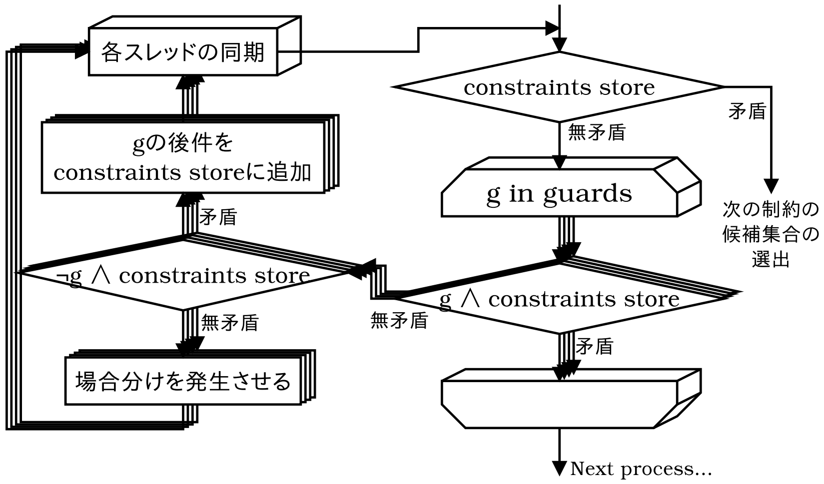

図 4.2　並列化されたガード条件判定のフローチャート

ループ終了時の同期処理を追加したものである．3.2.3 節で示した通り，成立するガード条件の発見順序はシミュレーションに影響しないため，図中の「各スレッドの同期」ブロックで行われる処理は，他のスレッドの途中終了か完全終了の待機かのどちらでもシミュレーションを誤る問題は起きない．

実際の実装では，guards 内に含まれる成立するべきガード条件が早期に発見されることで，constraints store の無矛盾性判定を含む外側のループが回る回数を減らせることを期待して，他のスレッドが完全に終了するまで待機するように実装した．

この並列化を行った HyLaGI の構成図を図 4.3 に示す．

4.4　ケース分岐時のシミュレーションの並列化

ケース分岐に対応するシミュレーションの深さ優先探索は，図 4.1 における Sequential Simulator で行われる．このクラスは，深さ優先探索を行いながらフェーズを辿り，実際にシミュレーションを進める処理を担う．

理想的な並列化の方法としては，Phase Simulator 内でケース分岐が検出された直後に新たな Phase Simulator のインスタンスを生成し，別のケースのシミュレーションを並列に行うことで，無駄な時間なく並列にシミュレーションできると考えられる．しかし，そのためには Phase Simulator に割り当てるバックエンドを動的に増加・減少させることが必要であり，現状の実装で困難であると考えられるため，この方法での実装は見送った．

代わりに Sequential Simulator 内の深さ優先探索を並列化した．これは，ケース分岐が検出され，各フェーズの次のフェーズのリストについて，並列にシミュレーションを行

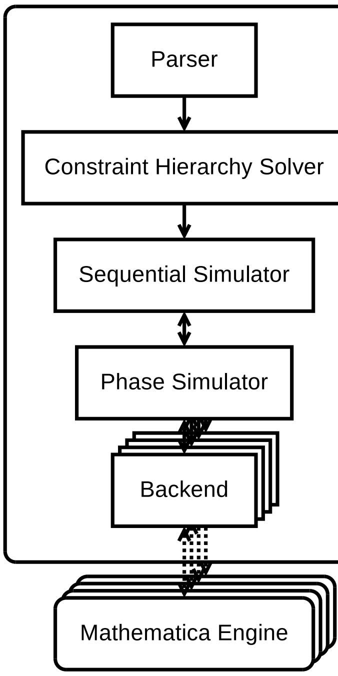

図 4.3　ガード条件判定の並列化を行った HyLaGI の構成図

うものである．ケース分岐はフェーズのシミュレーション中に検出されるため，探索を事前に計画的に進めることが出来ず，この方法では理想的な並列効果は出ないが，各 Phase Simulator に割り当てるバックエンドの管理は容易になる．また，穴あきのバウンシングボールのようにケースが多岐にわたり分岐する問題ではこの方法でも十分に効果が出ると考えられる．

4.3 節におけるガード条件判定の並列化に加えて，この並列化も行った HyLaGI の構成図を図 4.4 に示す．

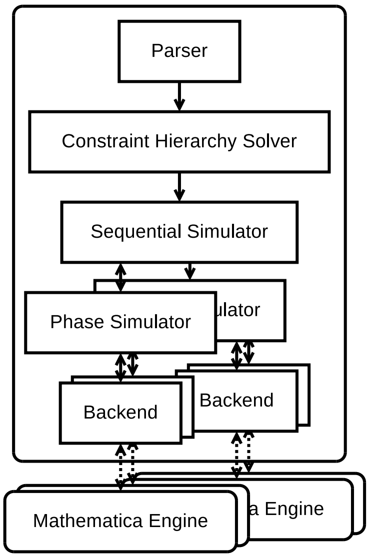

図 4.4　ガード条件判定・ケース分岐の並列化を行った HyLaGI の構成図

第 5 章

性能評価

この章では第 4 章で実装した機能について，例題を用いて性能評価を行い考察する．

5.1　実験環境および計測結果について

本章における実験環境について，表 5.1 に示す．
本章における実験では，時間を計測するために 5 回計測を行った．

表 5.1　実験環境

OS	Gentoo Linux (kernel 4.3.0-gentoo, x86_64)
CPU	Intel®Xeon®Processor E7-4860 (24M Cache, 2.26GHz)
	10 cores (20 threads) × 4 sockets
メモリ容量	252 GiB
コンパイラ	clang version 3.7.0 (tags/RELEASE_370/final)
	最適化オプション: `-O`
数式処理	Mathematica 9.0

5.2　ガード条件判定の並列化

5.2.1　ビリヤードのモデルにおけるガード条件判定の処理時間

ガード条件判定の並列化の性能を評価するにあたって，ベンチマークとして 2.1.1 節におけるビリヤードのモデルを使用した．並列化された処理系の性能評価を行う前に，ガード条件判定の並列化が有意義であることを確認するために予備実験としてビリヤードのボールの数に対するガード条件判定を含めた処理時間の計測を行った．シミュレーションは 20 フェーズまで行った．また，このモデルのシミュレーションがベンチマークとして適していることも示す．

実験結果を表 5.2 に，これをグラフにしたものを図 5.1 に示す．

図 5.1 より，処理時間がボールの数が増えるのに応じて $O(n^2)$ で増加していることが確認できる．また，シミュレーション全体の処理時間に対して 9 割以上の時間をガード条件判定が占めている事が確認できる．

このことから，ガード条件判定の並列化による高速化が有意義であると言える．また，このモデルのシミュレーションはほとんどの時間をガード条件判定に費やしているため，ガード条件判定の高速化に関するベンチマークとして適していると言える．

表 5.2　1 スレッド時のビリヤードにおけるボールの数に対する処理時間 (μsec)

ボール数	シミュレーション全体		ガード条件判定	
	平均値	標準誤差	平均値	標準誤差
100	199462200	594119	161378200	533697
120	318475800	563491	268425400	483876
140	474526400	239545	410825400	227237
160	682913400	5389727	604240800	5015335
180	936403000	924577	841171600	891365
200	1254176000	1405007	1141136000	1345179

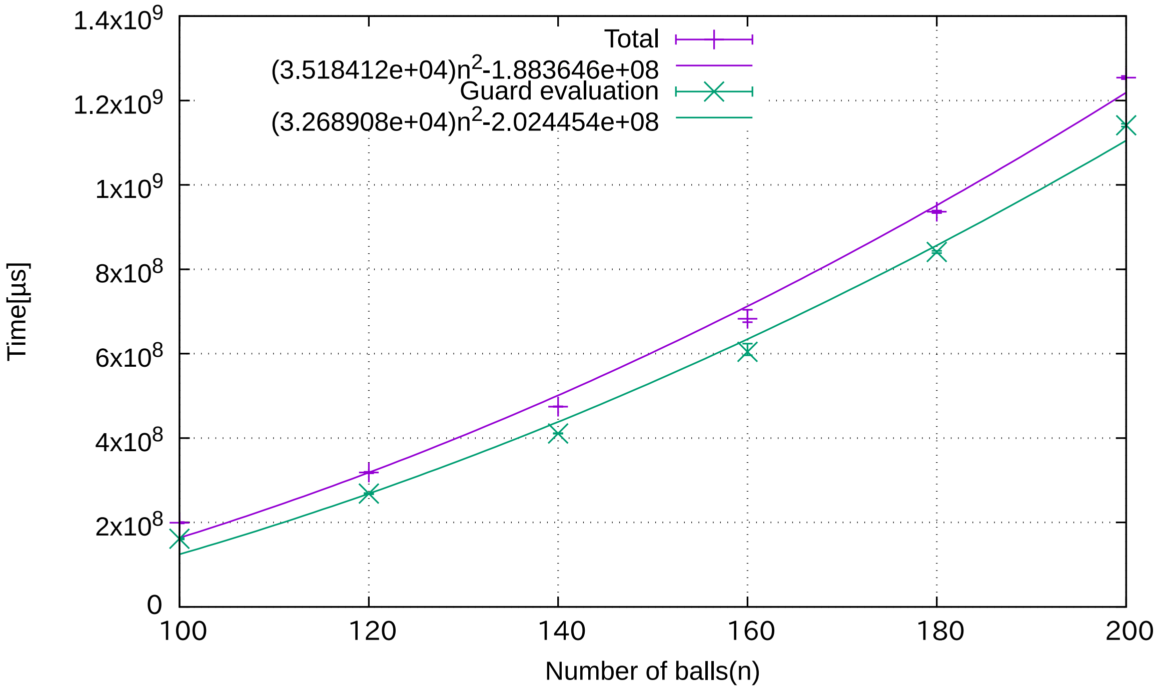

図 5.1　1 スレッド時のビリヤードにおけるスレッド数に対する処理時間

5.2.2　ビリヤードのモデルにおける並列ガード条件判定の効果

次に，ボールの個数が 200 個であるビリヤードのモデルのシミュレーションをベンチマークとして，並列化したガード条件判定の効果について検証を行った．シミュレーションは 20 フェーズまで行った．実験結果を表 5.3 に，これをグラフにしたものを図 5.2 に示す．また，速度向上について示すため，1 スレッド時の時間を各スレッド時の時間で割った数値のグラフを図 5.3 に示す．

図 5.2 より，シミュレーション全体及びガード条件判定の処理時間両方が，スレッド数の増加に応じて減少していることが確認できる．その上で図 5.3 より，ガード所件判定の処理時間は 16 スレッド辺りまでほぼスレッド数倍になっていることが確認できる．32 スレッドの場合に速度向上の度合いが小さくなっているが，これはスレッド数に対してガード条件の個数が十分でないためであると考えられる．

シミュレーション全体の実行時間は，ガード条件判定の時間が減少するほどそれ以外の時間の割合が増加し，速度向上の度合いが鈍くなることが確認できる．しかし，並列化のオーバーヘッドによる極端な処理時間の増大は見られず，計測した範囲ではスレッド数が

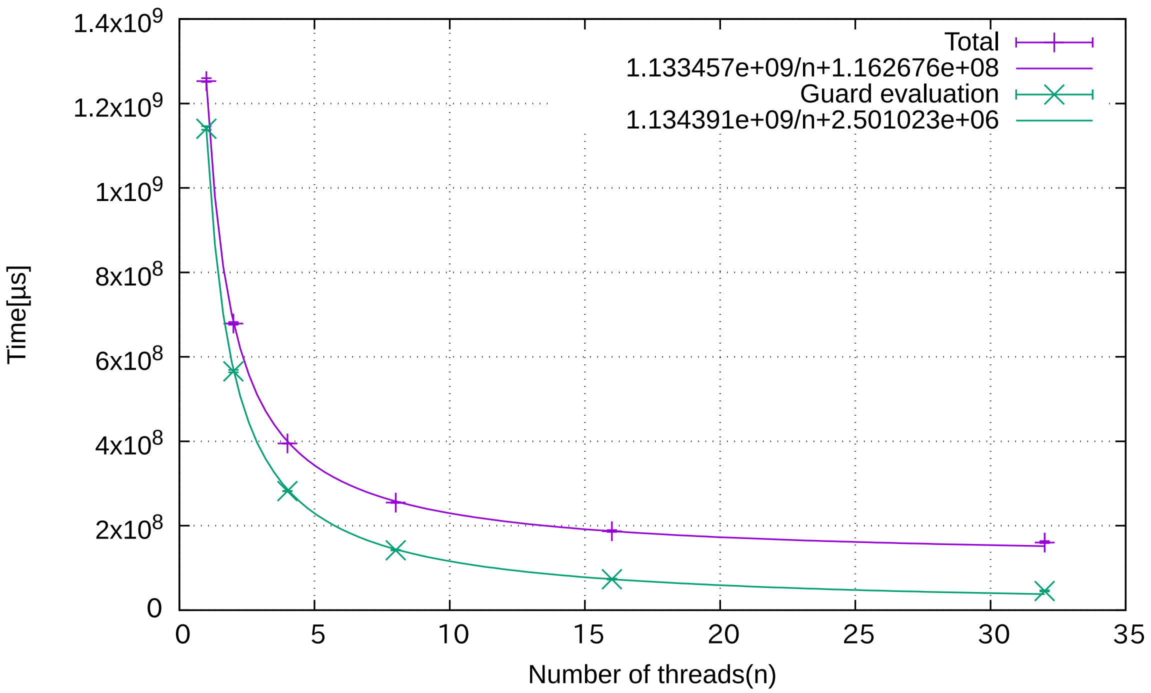

図 5.2　200 ボールのビリヤードにおけるスレッド数に対する処理時間

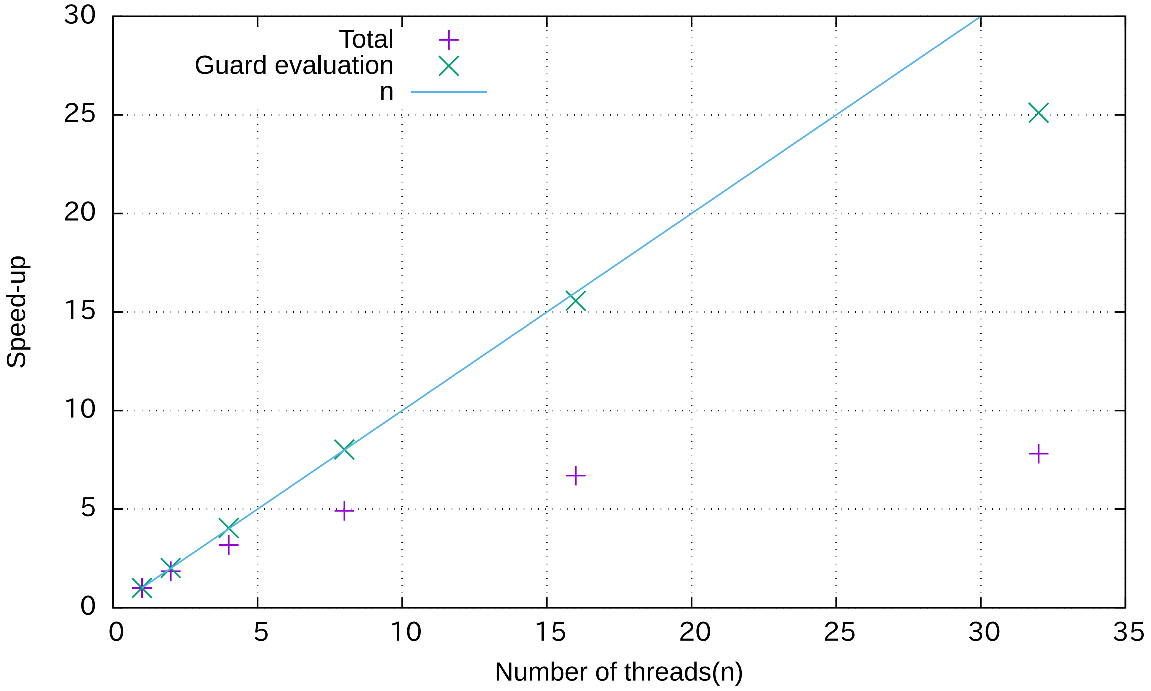

図 5.3　200 ボールのビリヤードにおけるスレッド数に対する速度向上

増えるごとに高速化していることが分かる．

結果として，ガード条件判定については期待通り高速化することに成功した．また，全体の実行時間に関しても高速化しており，満足できる結果が得られた．

5.2.3　ガード条件の個数を減らした場合の並列ガード条件判定の効果

続いて，ボールの個数に対する並列ガード条件判定の効果を確認する．これにより，ガード条件の個数がどの程度以上あれば並列判定の効果が得られるかを評価する．シミュレーションは 20 フェーズまで行った．計測結果を表 5.4 に，各ボールの数に対する速度向上の度合いを表したグラフを図 5.4 から 5.8 に示す．

図 5.4 より，ボールの数が 20 個の場合は理想的なスレッド数倍の速度向上は見られないものの，ガード条件判定部分は高速化していることが確認できる．一方で全体の実行時間に関しては 4 スレッド以上の場合ほぼ速度向上が見られず，遅くなる場合も見られる．

図 5.5 から 5.8 より，ボールの数が 40 個の場合は 2 スレッド程度，60 個の場合は 4 スレッド程度，80 から 100 個の場合は 8 スレッド程度まで理想的な速度向上が見られる．3.2.1 節で述べた通り，ビリヤードの問題においてはボールの個数 n に対して ${}_nC_2 + 2n$ 回のガード条件判定が発生するため，本実験環境においては 1 スレッド辺り概ね 500 個以上のガード条件判定が発生する場合に理想的な速度向上が見られると考えられる．

表 5.3　200 ボールのビリヤードにおけるスレッド数に対する処理時間 (μsec)

スレッド数	シミュレーション全体		ガード条件判定	
	平均値	標準誤差	平均値	標準誤差
1	1253128000	1742309	1139992000	1651548
2	678760000	1040814	565639000	953385
4	394912200	140357	282054000	177611
8	255034800	329659	142062200	231174
16	186935600	691952	73200860	453325
32	160328800	874160	45390860	363574

表 5.4　ビリヤードにおけるボールの個数とスレッド数に対する処理時間 (μsec)

ボール数	スレッド数	シミュレーション全体		ガード条件判定	
		平均値	標準誤差	平均値	標準誤差
20	1	7994346	35285	2889804	15203
	2	6718976	24178	1655332	24134
	4	6224706	23536	905842	1652
	8	6346994	43344	611811	4053
	16	6272884	181736	413901	1971
	32	7030124	178723	336672	4488
40	1	25198440	14851	14145140	9546
	2	18476220	45937	7420006	36812
	4	15160640	41721	4081360	31150
	8	13467500	66242	2411080	81213
	16	13135380	126279	1975640	119809
	32	13794260	139628	1339922	297488
60	1	58963740	117393	40312200	85268
	2	40115060	196738	21418520	151482
	4	29978020	62522	11237220	34832
	8	24859200	64083	5897600	40611
	16	22451300	156948	3266470	22276
	32	21213740	50504	1994414	16777
80	1	115566000	99850	87928880	95840
	2	73628720	372789	45955260	260970
	4	50951560	54374	23322320	36033
	8	39674220	122258	11968200	62324
	16	34517680	153313	6341102	16771
	32	32624180	156866	3751076	41407
100	1	199081600	115882	160971200	100561
	2	120838200	269722	82760740	229615
	4	79754060	88288	41728420	58560
	8	59235000	45230	21252760	23048
	16	49559320	194484	11113740	68430
	32	45760560	634976	6203744	52690

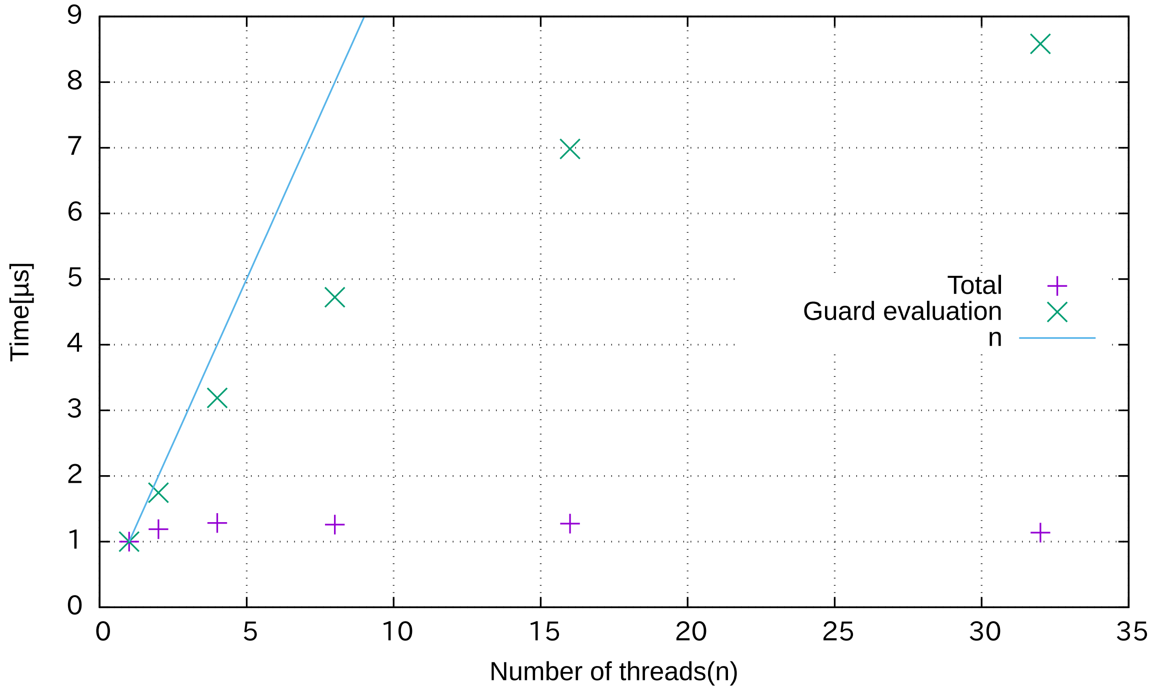

図 5.4　20 ボールのビリヤードにおけるスレッド数に対する速度向上

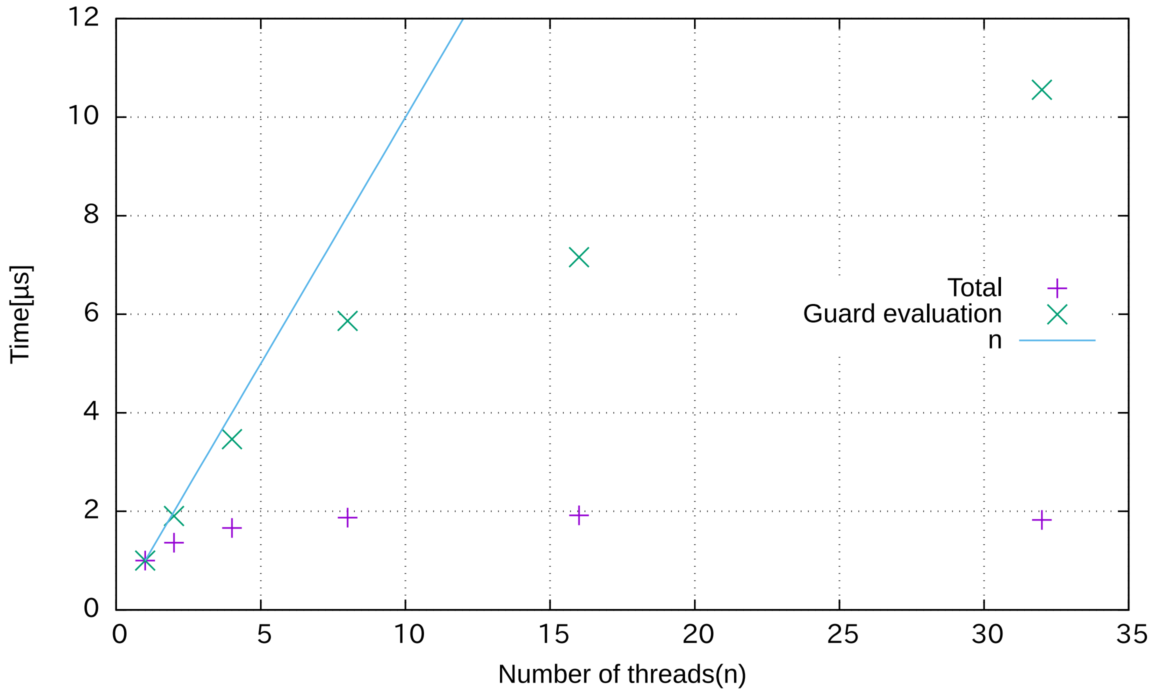

図 5.5　40 ボールのビリヤードにおけるスレッド数に対する速度向上

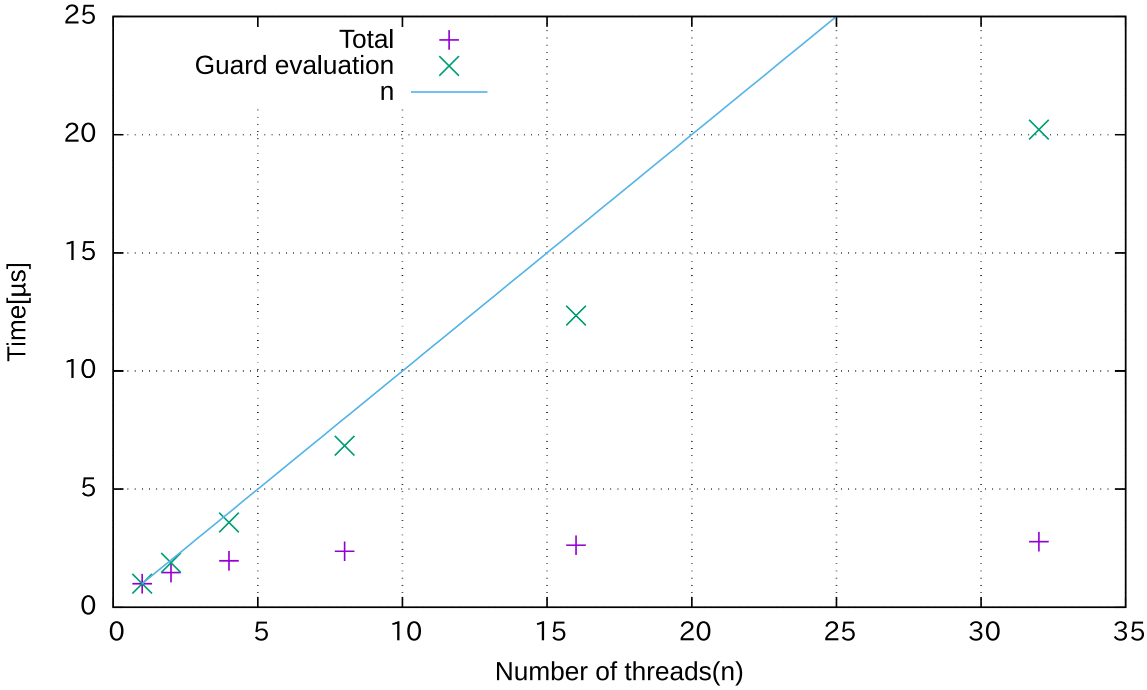

図 5.6　60 ボールのビリヤードにおけるスレッド数に対する速度向上

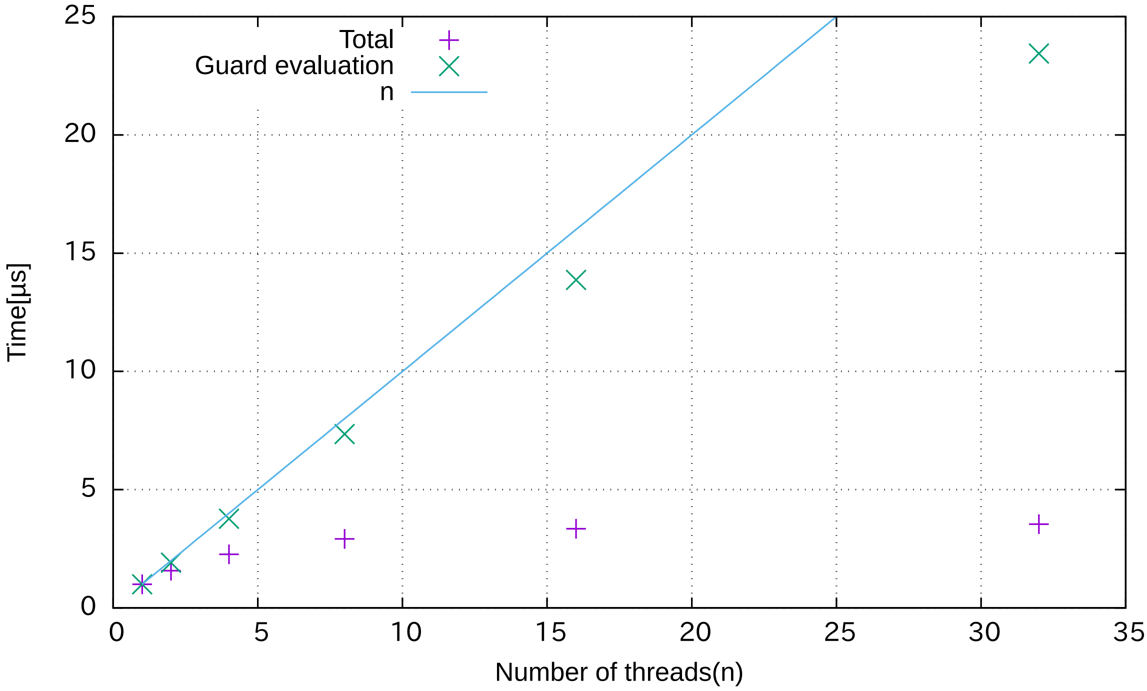

図 5.7　80 ボールのビリヤードにおけるスレッド数に対する速度向上

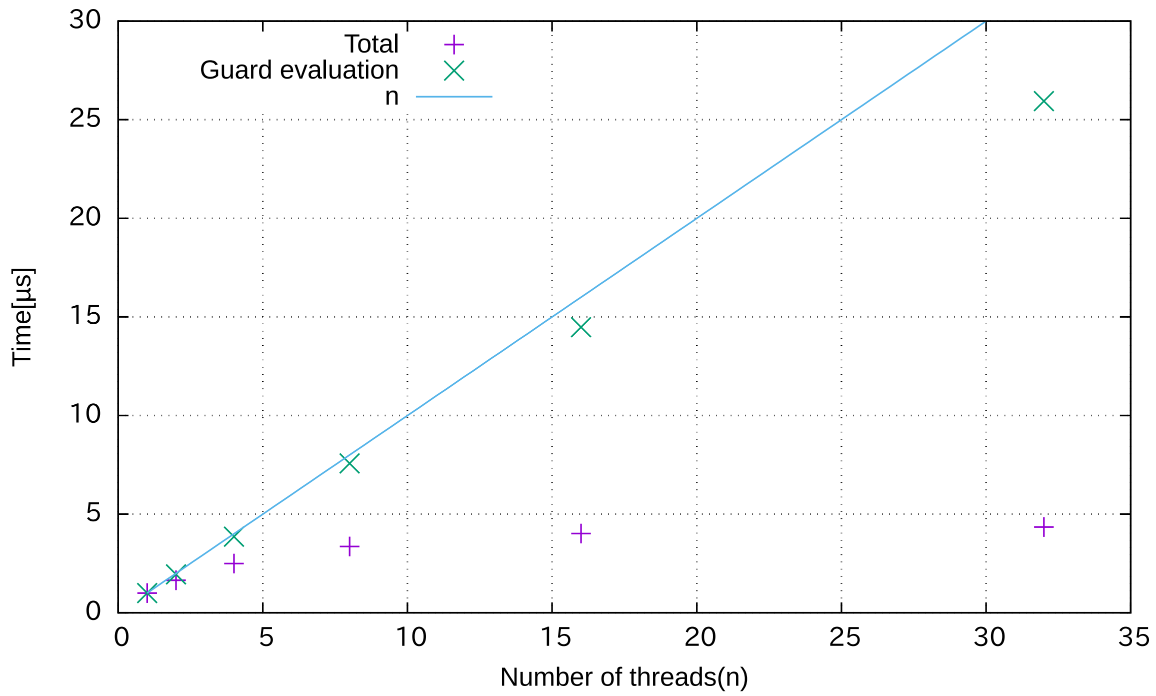

図 5.8　100 ボールのビリヤードにおけるスレッド数に対する速度向上

5.2.4　その他のモデルにおける並列ガード条件判定の効果

ビリヤード以外のモデルにおける，ガード条件判定の並列化の効果を検証する．この並列化が特に効果的であるのは，ガード条件判定が多数回発生するような，多数のオブジェクトをシミュレーションするようなモデルである．

道路を走る車列

このモデルは，道路を走る 30 台の自動車が前の車との距離に応じてブレーキやアクセルを操作する様子を再現したものである．このモデルの HydLa プログラムを図 5.9 に示す．

このモデルを 8 フェーズまでシミュレーションした際の計測結果を表 5.5 に，グラフにしたものを図 5.10 に，1 スレッド時と比較した速度向上を表すグラフを図 5.11 に示す．

図 5.10 より，スレッド数を増やすほどガード条件判定の時間は短くなり，それに応じてシミュレーション全体の実行時間も短くなることが確認できる．また，図 5.11 より，ガード条件判定に関してはスレッド数が増えるごとに速くなっていることが確認できる

```
INIT(x,x0,v0,timer) <=> x=x0 & x'=v0 & timer=0.
SW_ACC(xl,xf,timer) <=> [](xf'- < 30 & xl- - xf- >= 10 => timer
    '=1 & xf'' = xf'').
SW_BRK(xl,xf,timer) <=> [](xf'- > 0 & xl- - xf- <= 3 => timer
    '=-1 & xf'' = xf'').
ACC(xl,xf,timer) <=> [](timer- >= 1 => xf''=3).
BRK(xl,xf,timer) <=> [](timer- <= -1 => xf''=-5).
CONST(x) <=> [](x''=0).
TIMER_OFF(timer) <=>[](timer=0).

X := {x0..x29}.
T := {timer0..timer29}.

{ INIT(X[i],1000-20*i,20+i/|X|,T[i]) | i in {1..|X|} }.
CONST(X[1]), TIMER_OFF(T[1]).
{ ( CONST(X[i]), TIMER_OFF(T[i]) )
  << ( SW_ACC(X[i-1],X[i],T[i]), SW_BRK(X[i-1],X[i],T[i]) )
  << ( ACC(X[i-1],X[i],T[i]), BRK(X[i-1],X[i],T[i]) )
  | i in {2..|X|} }.
```

図 5.9　HydLa による道路を走る車列のモデル

表 5.5　道路を走る車列のモデルにおけるスレッド数に対する処理時間 (μsec)

スレッド数	シミュレーション全体		ガード条件判定	
	平均値	標準誤差	平均値	標準誤差
1	38813400	71847	17290540	42192
2	32391580	158524	9063216	42089
4	28440140	137259	4890974	35787
8	26717740	439119	2800126	55731
16	26316320	515421	1990878	241254
32	27849660	167988	1050362	39391

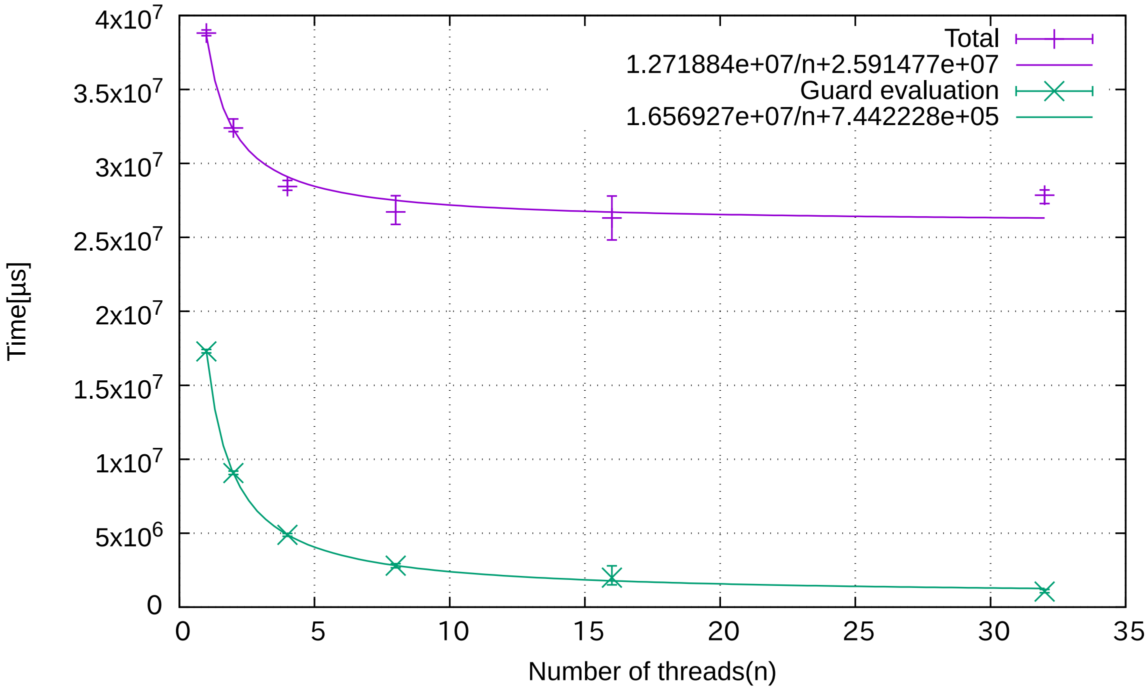

図 5.10　道路を走る車列のモデルにおけるスレッド数に対する処理時間

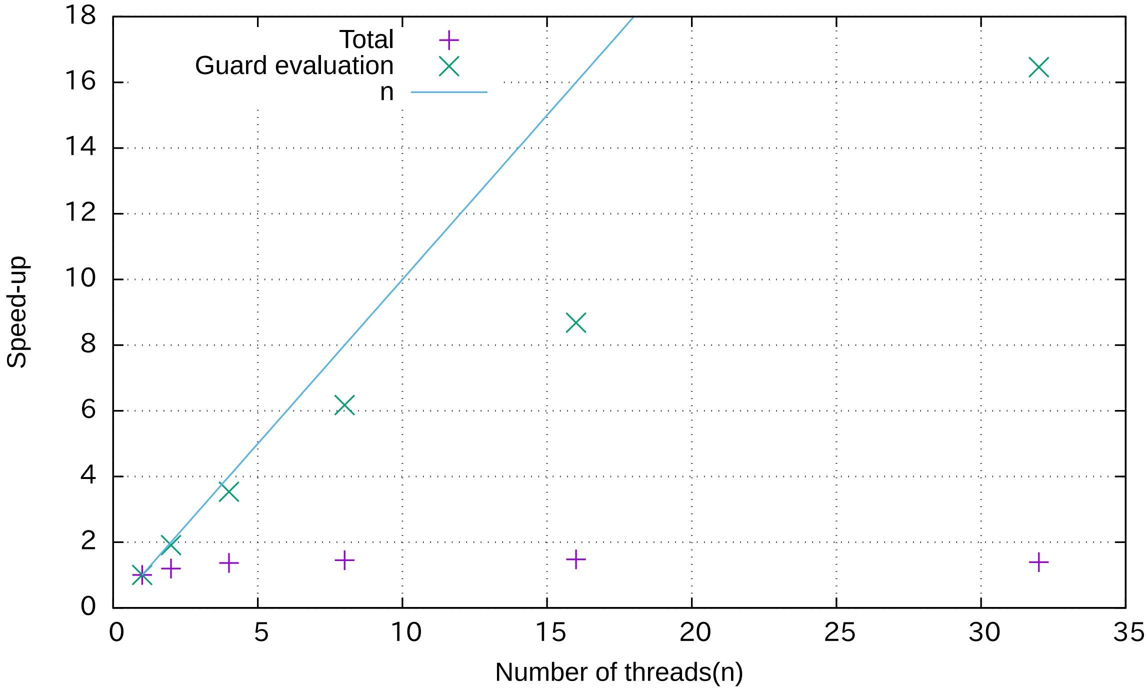

図 5.11　道路を走る車列のモデルにおけるスレッド数に対する速度向上

が，このモデルのシミュレーションにかかる時間に対してガード条件判定の占める割合が小さいため，全体の実行時間への効果は限定的である．よって，このモデルに関してはガード条件判定の並列化による効果は小さいと言える．

すれ違う歩行者

このモデルは，2 次元平面上に一定の速度で前進する歩行者が 20 人存在し，他の歩行者と一定以上近づいた場合に左右に曲がって避けようとする様子をモデル化したものである．このモデルの HydLa プログラムを図 5.12 から 5.14 に示す．

このモデルを 8 フェーズまでシミュレーションした際の計測結果を表 5.6 に，グラフにしたものを図 5.15 に，1 スレッド時と比較した速度向上を表すグラフを図 5.16 に示す．

図 5.15 より，このモデルにおいてはシミュレーション全体に対してガード条件判定の占める時間の割合が多く，ガード条件判定の並列化により大きく高速化できていることが分かる．また，図 5.16 より，ガード条件判定は 16 スレッドまでスレッド数倍高速化されており，全体の実行時間に関しても 5 倍以上高速化されている事が確認できる．よって，このモデルについてはガード条件判定の並列化による効果が大きいと言える．

ウォータータンク

複数個の水が入るタンクがあり，それぞれのタンクが一定の速さで排水される．これらのタンクに注水できるホースが 1 つあり，あるタンクの水位が一定以下となったらそのタンクに注水するという制御を行う．[17] このモデルの HydLa プログラムを図 5.17 に示す．

表 5.6　すれ違う歩行者のモデルにおけるスレッド数に対する処理時間 (μsec)

スレッド数	シミュレーション全体		ガード条件判定	
	平均値	標準誤差	平均値	標準誤差
1	269928800	724516	240595800	642144
2	159172400	306127	129758800	251001
4	91542820	47241	62143460	45278
8	59653920	76604	30336880	51749
16	44913680	43624	15400480	15633
32	38553920	62964	8817294	41405

```
INIT(x, y, v, s, time, x0, y0, v0, s0) <=>
    x = x0 & y = y0 & x’ = v0*cos(s0) & y’ = v0*sin(s0) & s = s0
         & v = v0 & time = 0.

ANGLE <=> [](angle = Pi/6).
HUMAN_RADIUS <=> [](radius = 0.5).

TURN_LEFT(x1, y1, v1, s1, t1, r, x2, y2) <=>
    []((((x1- + r-*cos(s1-)) - x2-)^2 +
       ((y1- + r-*sin(s1-)) - y2-)^2 <= (r-+(2*radius-))^2 &
       (cos(s1-) * (y2- - y1-) - (x2- - x1-) * sin(s1-)) <= 0 &
       t1- <= 0 & v1- > 0
                   => s1 = s1- + angle- & t1 = 3).

TURN_RIGHT(x1, y1, v1, s1, t1, r, x2, y2) <=>
    []((((x1- + r-*cos(s1-)) - x2-)^2 +
       ((y1- + r-*sin(s1-)) - y2-)^2 <= (r-+(2*radius-))^2 &
       (cos(s1-) * (y2- - y1-) - (x2- - x1-) * sin(s1-)) > 0 &
       t1- <= 0 & v1- > 0
                   => s1 = s1- - angle- & t1 = 3).

COLLISION(x1, y1, v1, x2, y2, v2) <=>
    []((x1- - x2-)^2 + (y1- - y2- )^2 = (2*radius-)^2 &
       (v1- > 0 | v2- > 0)
                   => v1 = 0 & v2 = 0 ).

STRAIGHT(t1, s1, s0) <=>
    [](t1- = 2 => s1 = s0-).

CONST(vs) <=> [](vs’=0).
TIMER(time) <=> [](time’ = -1).
```

図 5.12　HydLa によるすれ違う歩行者のモデル (1/3)

```
VELOCITY(x, y, v, s) <=> [](x' = v*cos(s) & y' = v*sin(s)).

X := {x1..x20}.
Y := {y1..y20}.
V := {v1..v20}.
S := {s1..s20}.
T := {t1..t20}.

Xinit := {-52, -52, 74, 14, -18, 48, 60, -24, 92, 80, 80, -6, 4, 52, -28, 88, 98, 92, 58, -74}.
Yinit := {70, -70, -24, 33, -52, 58, 4, 68, -70, -34, 46, -84, 94, -76, -84, -28, -80, -98, -22, 78}.
Vinit := { 1 | i in {1..100} }.
Sinit := {3*Pi/2, Pi/2, 2*Pi/2, 3*Pi/2, 2*Pi/2, 3*Pi/2, 5*Pi/3, 5*Pi/6, 5*Pi/3, 4*Pi/3, 4*Pi/3, 11*Pi/6, 0, 0, 7*Pi/6, 1*Pi/6, 2*Pi/2, 0, 7*Pi/6, 1*Pi/3}.

ANGLE, HUMAN_RADIUS,
{ INIT(X[i], Y[i], V[i], S[i], T[i], Xinit[i], Yinit[i], Vinit[i], Sinit[i])
    | i in {1..|X|} },

{ (TIMER(T[i]),CONST(S[i])) << TURN_LEFT(X[i],Y[i],V[i],S[i],T[i], 1, X[j], Y[j]) << $TRUE
      | i in {1..|X|}, j in {1..|X|}, i =!= j}.

{ (TIMER(T[i]),CONST(S[i])) << TURN_RIGHT(X[i],Y[i],V[i],S[i],T[i], 1, X[j], Y[j]) << $TRUE
      | i in {1..|X|}, j in {1..|X|}, i =!= j },
```

図 5.13　HydLa によるすれ違う歩行者のモデル (2/3)

```
{ (CONST(V[i]),CONST(V[j])) << COLLISION(X[i],Y[i],V[i],X[j],Y[
    j],V[j])
        | i in {1..|X|-1}, j in {i+1..|X|} },

{ VELOCITY(X[i],Y[i],V[i],S[i]) | i in {1..|X|} }.

{ STRAIGHT(T[i], S[i], Sinit[i]) | i in {1..|S|} },

{ TIMER(i) | i in T }.
```

図 5.14　HydLa によるすれ違う歩行者のモデル (3/3)

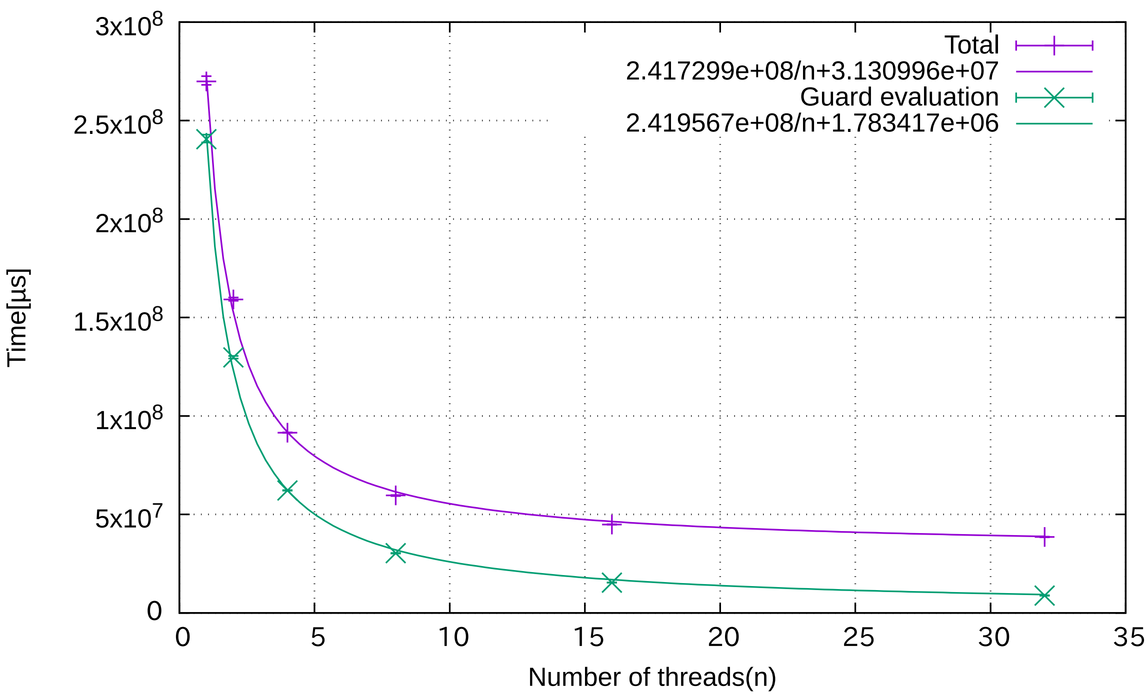

図 5.15　すれ違う歩行者のモデルにおけるスレッド数に対する処理時間

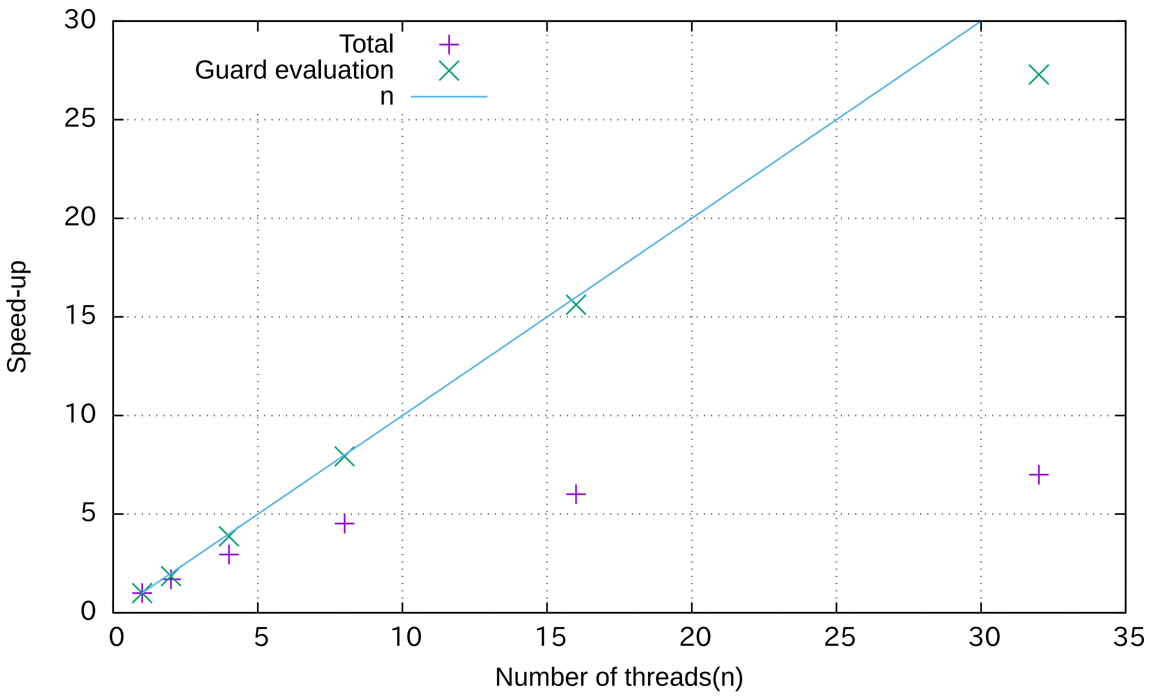

図 5.16　すれ違う歩行者のモデルにおけるスレッド数に対する速度向上

このモデルを 6 フェーズまでシミュレーションした際の計測結果を表 5.7 に，グラフにしたものを図 5.18 に，1 スレッド時と比較した速度向上を表すグラフを図 5.19 に示す．

図 5.18 より，スレッド数を増やすほどガード条件判定の時間は短くなるが，全体の処理時間に関しては 8 スレッド以上では悪化していることが分かる．また，図 5.16 より，ガード条件判定はスレッド数が増えるごとに高速化しているが理想的には伸びない．これ

表 5.7　ウォータータンクにおけるスレッド数に対する処理時間 (μsec)

スレッド数	シミュレーション全体		ガード条件判定	
	平均値	標準誤差	平均値	標準誤差
1	77301740	258361	35153720	187770
2	60820560	295287	18346080	84687
4	58227400	2084782	10325480	40866
8	56648940	1485722	6068292	64767
16	65664400	1559250	4160712	251123
32	69504380	2776303	1873098	19742

```
Xs := {x1..x30}.
Ss := {s1..s30}.

WATER <=> [](w = 50).
OUT(x,v,s) <=> [](s- = 0 => x' = -v*x) /\
               [](s- = 1 => x' = w-v*x).

CONST(s) <=> [](s' = 0).
POUR(x1,r1,s1) <=> [](x1- = r1- => s1 = 1).
NOT_POUR(x1,r1,s2) <=> [](x1- = r1- => s2 = 0).

INIT(x,xinit,s,sinit,vinit) <=> x = xinit /\ s = sinit /\ x' = vinit.

WATER.
INIT(Xs[1],1,Ss[1],1,w-5*Xs[1]).
{ INIT(Xs[i],i^3,Ss[i],0,-5*Xs[i]) | i in {2..|Xs|} }.
{ OUT(Xs[i],5+i,Ss[i]) | i in {1..|Xs|} }.
{ CONST(Ss[i]) << POUR(Xs[i],1/2*i,Ss[i]) | i in {1..|Xs|} }.
{ CONST(Ss[j]) << NOT_POUR(Xs[i],1/2*i,Ss[j]) | i in {1..|Xs|}, j in {1..|Xs|}, i =!= j }.
```

図 5.17　HydLa によるウォータータンクのモデル

はガード条件の個数がスレッド数に対して十分でないためであると考えられる．さらに，全体の処理時間に対してガード条件判定の占める割合が小さいため，全体の実行時間の伸びは小さいものとなっている．

よって，このモデルについてはガード条件判定の並列化による効果は小さいと言える．

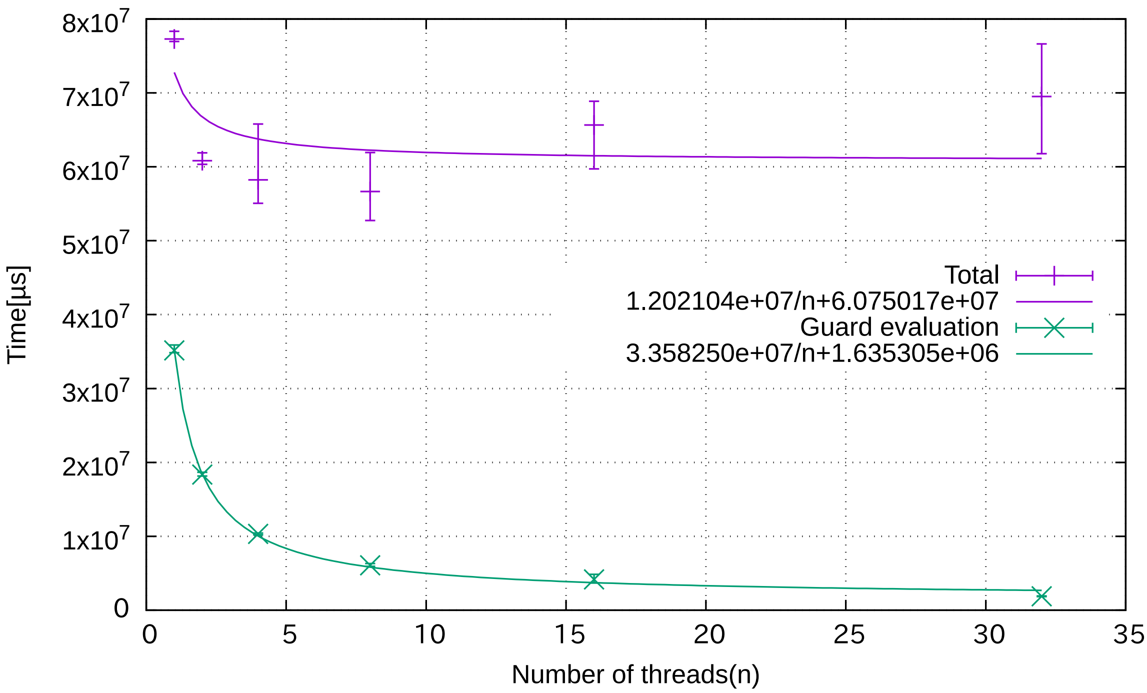

図 5.18　ウォータータンクにおけるスレッド数に対する処理時間

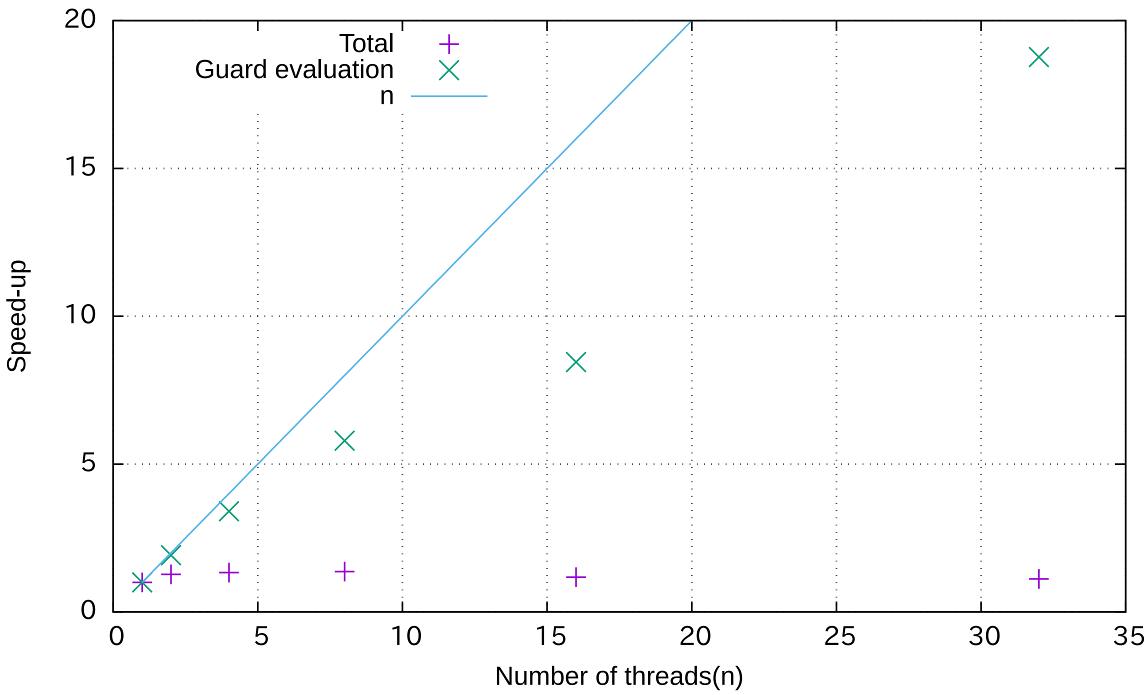

図 5.19　ウォータータンクにおけるスレッド数に対する速度向上

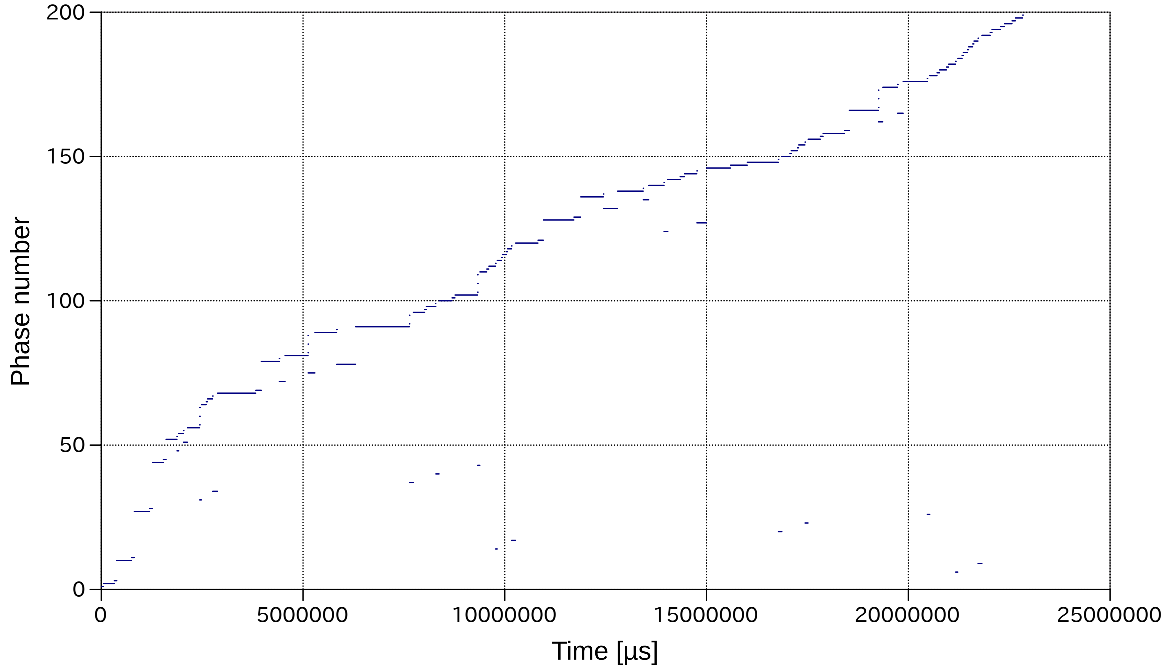

図 5.20　穴あきの床におけるバウンシングボールのタイムライン

5.3　ケース分岐時のシミュレーションの並列化

5.3.1　穴あきの床におけるバウンシングボールの分析

ケース分岐時のシミュレーションの並列化の効果を評価するため，3.3.2 で挙げた穴あきの床におけるバウンシングボールのモデルを用いた．まず，このモデルにおける並列化の効果を予測するため，このモデルを通常通り実行した上で各フェーズの開始時刻・終了時刻を計測し，シミュレーションのタイムラインを作成した．ここでの時刻はプログラム開始時を起点としたものである．計測した結果を表 5.8 に，タイムラインを図 5.20 に示す．

このシミュレーションにおけるフェーズ間の依存関係は図 3.5 に示したものと同様である．フェーズ間の依存関係から，解軌道が分岐した直後から並列にシミュレーションが行われた場合のタイムラインを図 5.21 に示す．

図 5.21 より，最後のフェーズの終了時刻は 22835786μsec から 4003084μsec となり，5.7 倍程度の高速化が予想される．この際，並列動作するスレッド数は最大 11 スレッドである．

5.3.2　穴あきの床におけるバウンシングボール

続いて，並列シミュレーションを実装した処理系によって実際にシミュレーション時間を計測した．10 フェーズまでシミュレーションした際の実験結果を表 5.9 に，これをグラフにしたものを図 5.22 に，1 スレッド時と比較した速度向上を表すグラフを図 5.23 に

表 5.8　穴あきの床におけるバウンシングボールにおける各フェーズの開始・終了時刻 (μsec)

フェーズ	開始時刻	終了時刻	フェーズ	開始時刻	終了時刻	フェーズ	開始時刻	終了時刻
1	3527	54798	99	8290591	8290606	151	17066126	17097571
2	54878	321169	40	8290973	8372412	152	17097631	17250877
3	321236	385462	100	8372471	8691470	153	17250942	17283834
10	385530	751015	101	8691530	8766435	154	17283901	17439391
11	751074	817689	102	8766503	9326120	155	17439451	17439466
27	817755	1198451	103	9326185	9326200	23	17439863	17516510
28	1198513	1267643	106	9326372	9326387	156	17516577	17818600
44	1267707	1537063	109	9326545	9326560	157	17818669	17887938
45	1537128	1605437	43	9326858	9382449	158	17888001	18419873
52	1605513	1875603	110	9382516	9553515	159	18419940	18536872
53	1875679	1875695	111	9553575	9605187	166	18536935	19258753
48	1875981	1921565	112	9605249	9771514	167	19258814	19258830
54	1921632	2035934	113	9771578	9771593	170	19259041	19259056
55	2035999	2036015	14	9772015	9812626	173	19259249	19259264
51	2036251	2134868	114	9812691	9916229	162	19259516	19367812
56	2134933	2438305	115	9916291	9943643	174	19367877	19735070
57	2438372	2438387	116	9943713	10042653	175	19735131	19735146
60	2438605	2438621	117	10042713	10069671	165	19735406	19872205
63	2438806	2438821	118	10069731	10167249	176	19872269	20469226
31	2439142	2482113	119	10167310	10167326	177	20469291	20469307
64	2482176	2600415	17	10167807	10267376	26	20469721	20532415
65	2600474	2632506	120	10267441	10822684	178	20532478	20711268
66	2632571	2760019	121	10822745	10954845	179	20711328	20774458
67	2760081	2760097	128	10954908	11712787	180	20774521	20945730
34	2760496	2881390	129	11712851	11881576	181	20945796	20999700
68	2881454	3829247	136	11881640	12443882	182	20999763	21172500
69	3829323	3964529	137	12443944	12443959	183	21172563	21172579
79	3964593	4410868	132	12444230	12795194	6	21173061	21228332
80	4410929	4410944	138	12795251	13430215	184	21228397	21332700
72	4411230	4554135	139	13430277	13430293	185	21332761	21362171
81	4554201	5124045	135	13430521	13567408	186	21362237	21466060
82	5124103	5124118	140	13567468	13946416	187	21466124	21495793
85	5124314	5124329	141	13946474	13946491	188	21495861	21596886
88	5124499	5124514	124	13946775	14040283	189	21596951	21626597
75	5124736	5295433	142	14040340	14345599	190	21626664	21727587
89	5295496	5833533	143	14345638	14454316	191	21727651	21727666
90	5833592	5833607	144	14454379	14759686	9	21728206	21823839
78	5833838	6304625	145	14759726	14759734	192	21823903	22028495
91	6304685	7635654	127	14759953	14999074	193	22028569	22074149
92	7635718	7635733	146	14999130	15593602	194	22074212	22285501
95	7635950	7635966	147	15593661	16008485	195	22285565	22383821
37	7636244	7732743	148	16008549	16778301	196	22383887	22570115
96	7732805	8015261	149	16778363	16778378	197	22570180	22652408
97	8015325	8054670	20	16778800	16867065	198	22652470	22835706
98	8054735	8290528	150	16867133	17066055	199	22835771	22835786

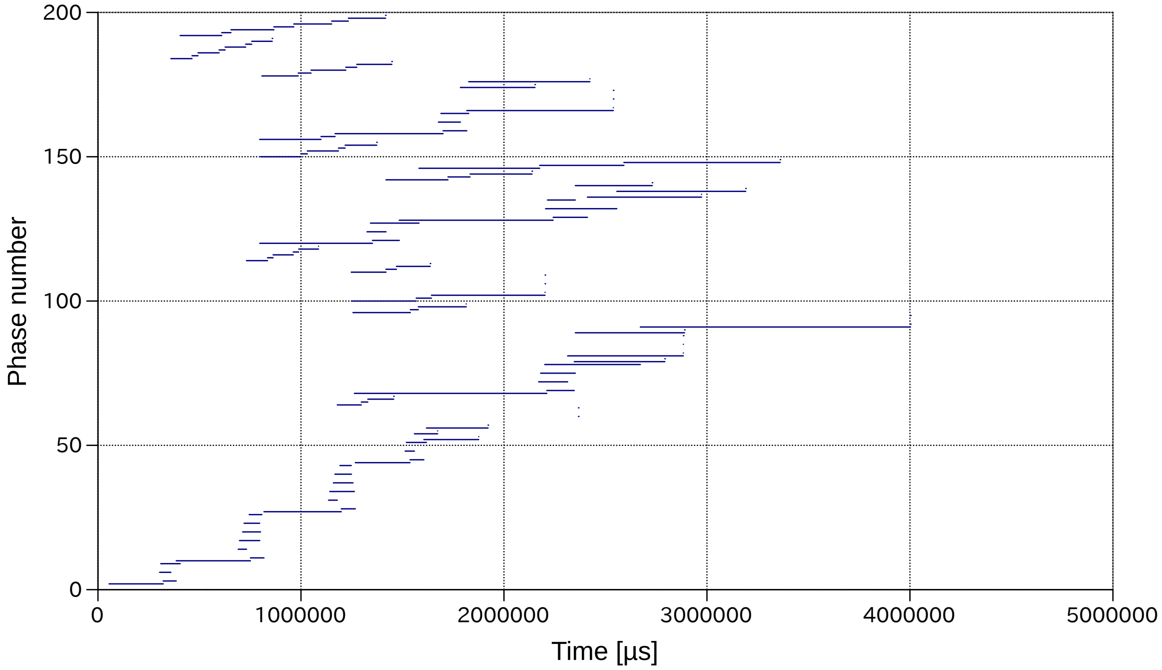

図 5.21　理想的な並列化が行われた穴あきの床におけるバウンシングボールのタイムラインの予測

示す．

図 5.22 より，ケース分岐した際のシミュレーションの並列化により，シミュレーション全体の実行時間が短くなったことが分かる．$5 \leq n \leq 9$ の区間のようにプロットが平坦に並んでいる箇所に関しては，並列化の実装に起因して実際に動作するスレッドが増えないためであると考えられる．また，図 5.23 より，最大で 1.65 倍程度まで高速化できていることが確認できる．図 5.21 での予測ほど高速化できていないのは各ケースのシミュレーションが分岐直後に開始できていないためであり，並列化の実装に起因する．

結果として，ガード条件判定の並列化がシミュレーション時間の短縮に有効であることが確認でき，高速化に成功した．

5.3.3　その他のモデルにおけるケース分岐時の並列シミュレーションの効果

穴あきの床におけるバウンシングボール以外の例題における，ケース分岐時のシミュレーションの並列化の効果を検証する．この並列化が特に効果的であるのは，ケース分岐が多数発生する，パラメータなど非決定的な要素を含むモデルである．

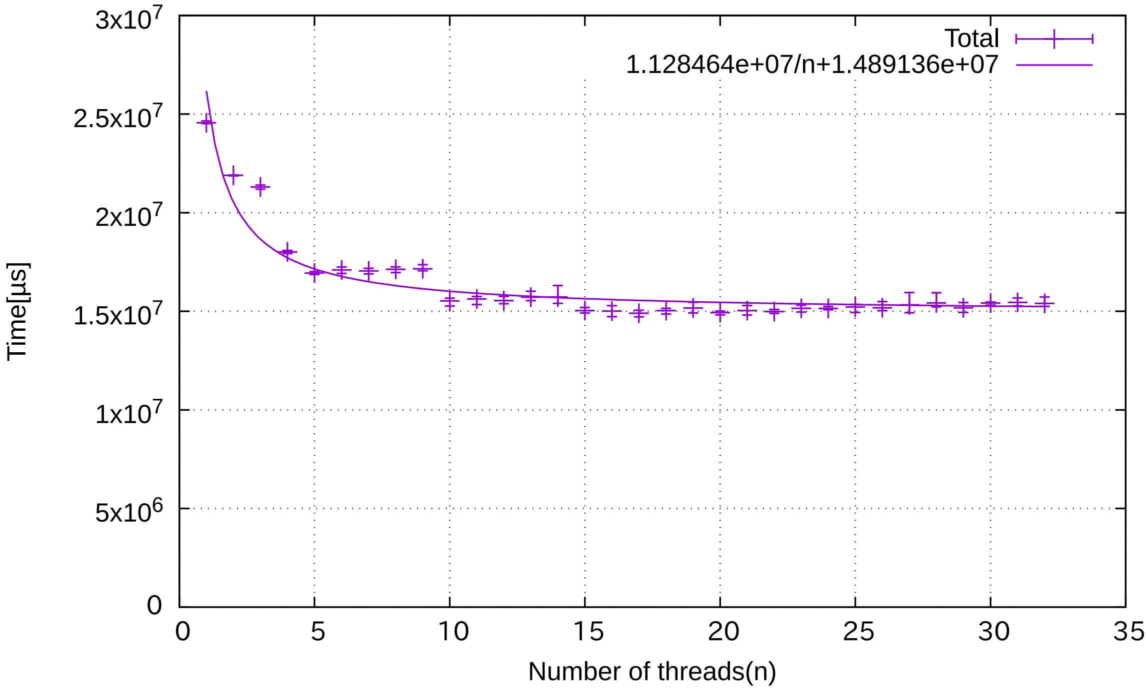

図 5.22　穴あきの床におけるバウンシングボールにおけるスレッド数に対する処理時間

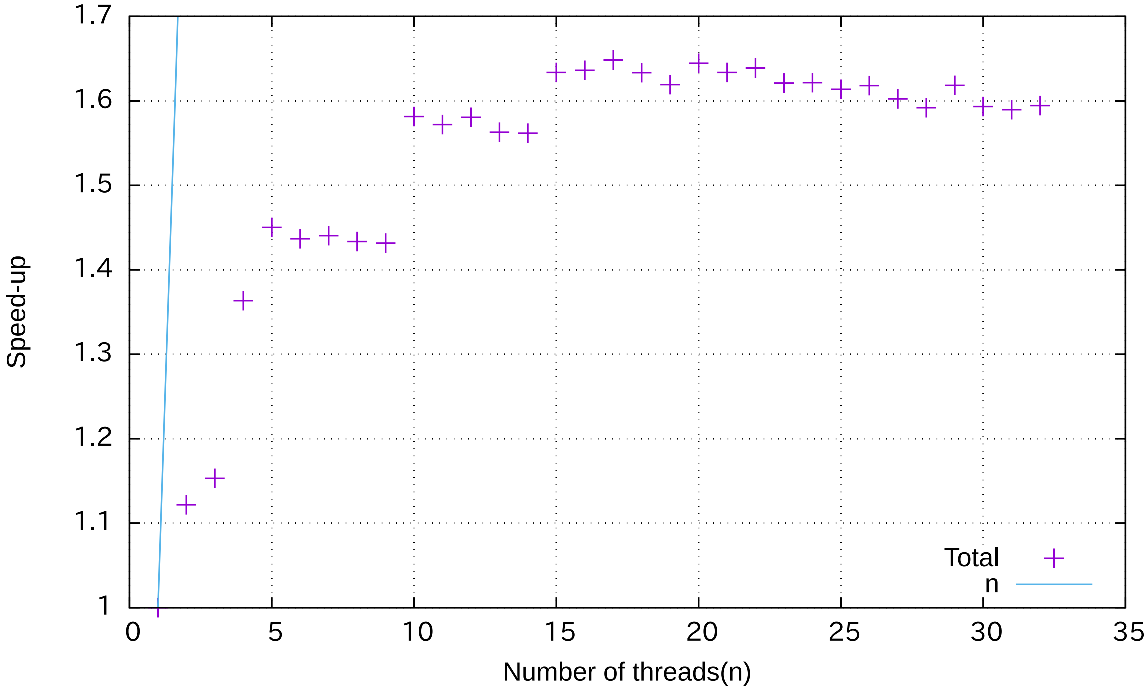

図 5.23　穴あきの床におけるバウンシングボールにおけるスレッド数に対する速度向上

上下動する床におけるバウンシングボール

このモデルでは 2 次元空間上で左右に壁が有り，10 分割された床へ向けて x 軸方向の初速度にパラメータを持たせたボールを落とす．10 分割された床にボールが当たる度に，ボールがあたった部分の両隣の床の高さが上がったり下がったりする．このモデルの HydLa プログラムを図 5.24 から 5.25 に示す．

このモデルを 5 フェーズまでシミュレーションした際の計測結果を表 5.10 に，グラフにしたものを図 5.26 に，1 スレッド時と比較した速度向上を表すグラフを図 5.27 に示す．

図 5.26 と図 5.27 より，並列化によりシミュレーション全体の時間が高速化されたことが分かる．最大で，30 スレッド時に 3.17 倍程度の性能となっているが，16 スレッド以上から性能の向上はほぼ見られない．

表 5.9　穴あきの床におけるバウンシングボールにおけるスレッド数に対する処理時間 (μsec)

スレッド数	シミュレーション全体		スレッド数	シミュレーション全体	
	平均値	標準誤差		平均値	標準誤差
1	24562759	24626	17	14901240	53841
2	21895933	6337	18	15036822	48353
3	21304413	38131	19	15168034	104056
4	18014829	28243	20	14935232	32886
5	16938828	24987	21	15036069	103940
6	17096408	52646	22	14987829	40274
7	17051944	49548	23	15153124	70684
8	17134216	60085	24	15146442	28457
9	17158433	54166	25	15221058	73227
10	15531742	68112	26	15180000	84371
11	15626195	76065	27	15328427	174928
12	15540317	62964	28	15429829	136976
13	15716328	80428	29	15177114	99249
14	15727400	161930	30	15415970	27831
15	15034766	52865	31	15452410	68019
16	15012576	93053	32	15405451	86684

```
INIT <=> x=1/2 & 2<x'<10 & y=10 & y'=0.

WALL0(h) <=> (x-=0 & y-<h => x'=-x'-).
WALL1(h) <=> (x-=1 & y-<h => x'=-x'-).
WALL2(h) <=> (x-=2 & y-<h => x'=-x'-).
WALL3(h) <=> (x-=3 & y-<h => x'=-x'-).
WALL4(h) <=> (x-=4 & y-<h => x'=-x'-).
WALL5(h) <=> (x-=5 & y-<h => x'=-x'-).
WALL6(h) <=> (x-=6 & y-<h => x'=-x'-).
WALL7(h) <=> (x-=7 & y-<h => x'=-x'-).
WALL8(h) <=> (x-=8 & y-<h => x'=-x'-).
WALL9(h) <=> (x-=9 & y-<h => x'=-x'-).
WALL10(h) <=> (x-=10 & y-<h => x'=-x'-).

FLOOR0(h) <=> (y-=h&x->=0&x-<=1 => y'=-y'-&s1=1-s1-).
FLOOR1(h) <=> (y-=h&x->1 &x-<=2 => y'=-y'-&s0=1-s0-&s2=1-s2-).
FLOOR2(h) <=> (y-=h&x->2 &x-<=3 => y'=-y'-&s1=1-s1-&s3=1-s3-).
FLOOR3(h) <=> (y-=h&x->3 &x-<=4 => y'=-y'-&s2=1-s2-&s4=1-s4-).
FLOOR4(h) <=> (y-=h&x->4 &x-<=5 => y'=-y'-&s3=1-s3-&s5=1-s5-).
FLOOR5(h) <=> (y-=h&x->5 &x-<=6 => y'=-y'-&s4=1-s4-&s6=1-s6-).
FLOOR6(h) <=> (y-=h&x->6 &x-<=7 => y'=-y'-&s5=1-s5-&s7=1-s7-).
FLOOR7(h) <=> (y-=h&x->7 &x-<=8 => y'=-y'-&s6=1-s6-&s8=1-s8-).
FLOOR8(h) <=> (y-=h&x->8 &x-<=9 => y'=-y'-&s7=1-s7-&s9=1-s9-).
FLOOR9(h) <=> (y-=h&x->9 &x-<=10=> y'=-y'-&s8=1-s8-).

BOUNCEFLOOR0(h) <=> (WALL0(h)&FLOOR0(h)&WALL1(h)).
BOUNCEFLOOR1(h) <=> (WALL1(h)&FLOOR1(h)&WALL2(h)).
BOUNCEFLOOR2(h) <=> (WALL2(h)&FLOOR2(h)&WALL3(h)).
BOUNCEFLOOR3(h) <=> (WALL3(h)&FLOOR3(h)&WALL4(h)).
BOUNCEFLOOR4(h) <=> (WALL4(h)&FLOOR4(h)&WALL5(h)).
BOUNCEFLOOR5(h) <=> (WALL5(h)&FLOOR5(h)&WALL6(h)).
BOUNCEFLOOR6(h) <=> (WALL6(h)&FLOOR6(h)&WALL7(h)).
```

図 5.24　HydLa による上下動する床におけるバウンシングボールのモデル (1/2)

```
BOUNCEFLOOR7(h) <=> (WALL7(h)&FLOOR7(h)&WALL8(h)).
BOUNCEFLOOR8(h) <=> (WALL8(h)&FLOOR8(h)&WALL9(h)).
BOUNCEFLOOR9(h) <=> (WALL9(h)&FLOOR9(h)&WALL10(h)).

STATE0 <=> s0 = 0. STATE1 <=> s1 = 0. STATE2 <=> s2 = 0.
STATE3 <=> s3 = 0. STATE4 <=> s4 = 0. STATE5 <=> s5 = 0.
STATE6 <=> s6 = 0. STATE7 <=> s7 = 0. STATE8 <=> s8 = 0.
STATE9 <=> s9 = 0.

CONST0 <=> [](s0' = 0). CONST1 <=> [](s1' = 0).
CONST2 <=> [](s2' = 0). CONST3 <=> [](s3' = 0).
CONST4 <=> [](s4' = 0). CONST5 <=> [](s5' = 0).
CONST6 <=> [](s6' = 0). CONST7 <=> [](s7' = 0).
CONST8 <=> [](s8' = 0). CONST9 <=> [](s9' = 0).

SENSOR0<=>[](BOUNCEFLOOR0(s0)). SENSOR1<=>[](BOUNCEFLOOR1(s1)).
SENSOR2<=>[](BOUNCEFLOOR2(s2)). SENSOR3<=>[](BOUNCEFLOOR3(s3)).
SENSOR4<=>[](BOUNCEFLOOR4(s4)). SENSOR5<=>[](BOUNCEFLOOR5(s5)).
SENSOR6<=>[](BOUNCEFLOOR6(s6)). SENSOR7<=>[](BOUNCEFLOOR7(s7)).
SENSOR8<=>[](BOUNCEFLOOR8(s8)). SENSOR9<=>[](BOUNCEFLOOR9(s9)).

LEFT <=> [](x-=0 => x'=-x'-). RIGHT <=> [](x-=10 => x'=-x'-).
FALL <=> [](y'' = -10). XMOVE <=> [](x'' = 0).

INIT, STATE0,STATE1,STATE2,STATE3,STATE4,STATE5,STATE6,STATE7,
    STATE8,STATE9,
    (CONST0,CONST1,CONST2,CONST3,CONST4,CONST5,CONST6,CONST7,
        CONST8,CONST9,FALL,XMOVE)
<< (SENSOR0,SENSOR1,SENSOR2,SENSOR3,SENSOR4,SENSOR5,SENSOR6,
    SENSOR7,SENSOR8,SENSOR9,LEFT,RIGHT).
```

図 5.25　HydLa による上下動する床におけるバウンシングボールのモデル (2/2)

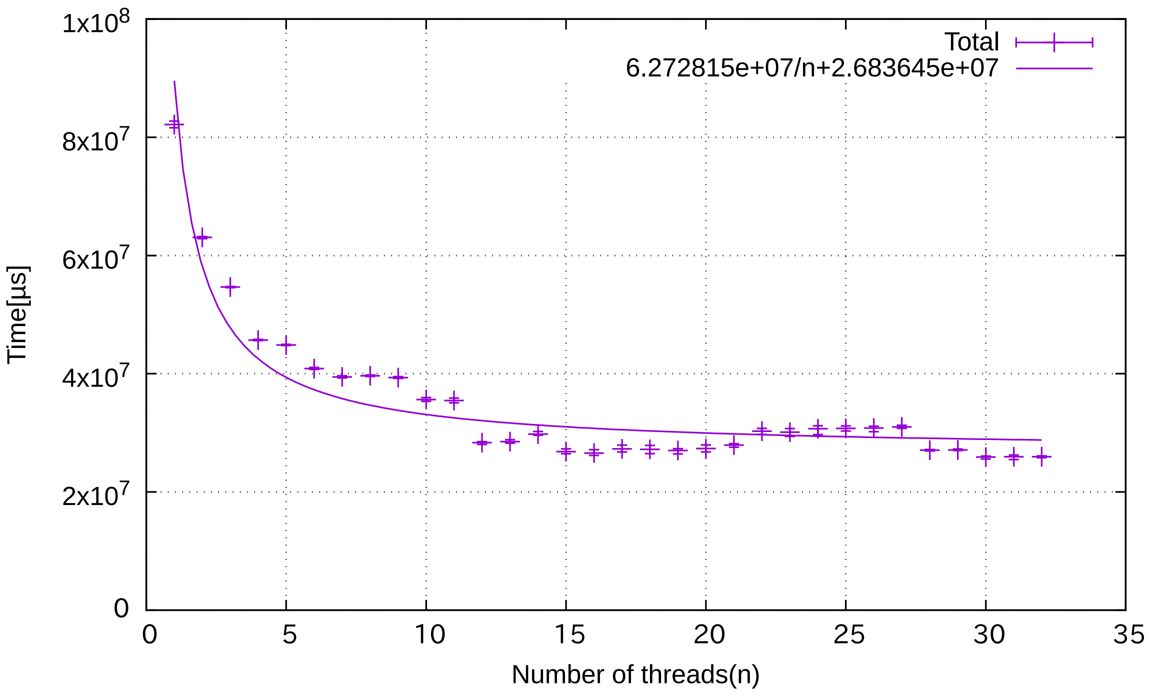

図 5.26　上下動する床におけるバウンシングボールにおけるスレッド数に対する処理時間

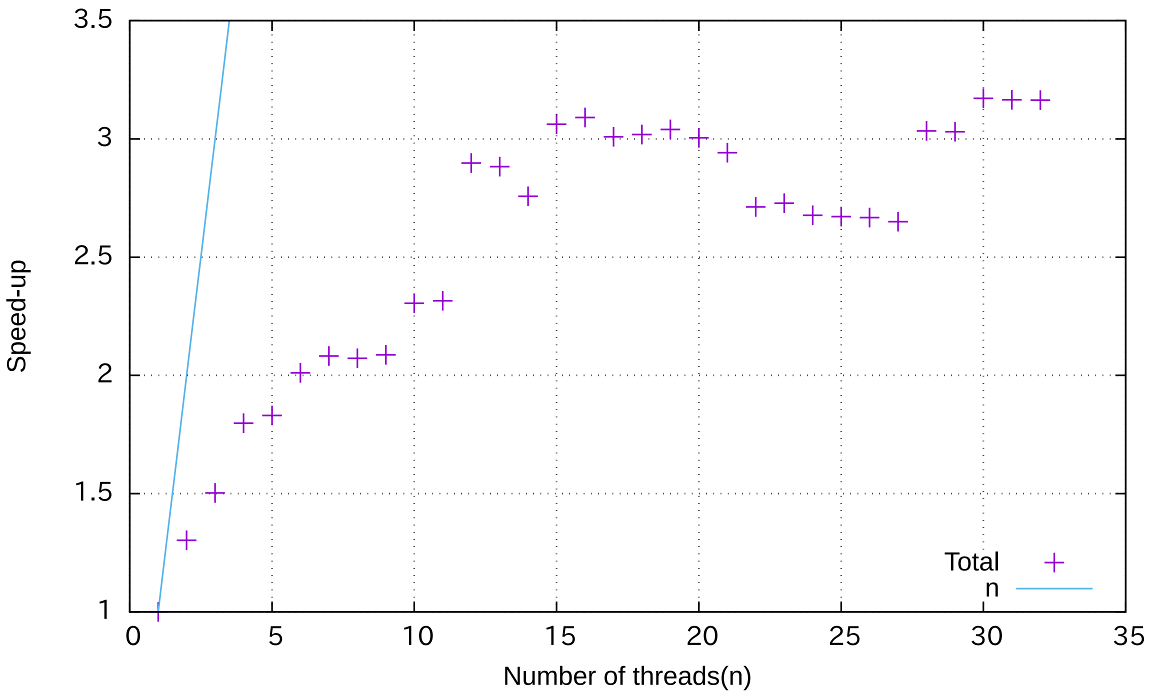

図 5.27　上下動する床におけるバウンシングボールにおけるスレッド数に対する速度向上

マグネットボール

このモデルでは重力と磁力の影響を受けるボールが，1 次元空間において落下する．磁力による力は重力と逆方向に働き，空間に働く磁力は 1 秒毎に 3 種類の値にランダムに切り替わる．そのため 1 秒毎に解軌道は 3 通りに分岐する．このモデルの HydLa プログラムを図 5.28 に示す．

このモデルを 12 フェーズまでシミュレーションした際の計測結果を表 5.11 に，グラフにしたものを図 5.29 に，1 スレッド時と比較した速度向上を表すグラフを図 5.30 に示す．

図 5.29 と図 5.30 より，並列化によりシミュレーション全体の時間が高速化されたことが分かる．最大で，17 スレッド時に 4.47 倍程度の性能となっており，本評価実験で試した例題の中では，ケース分岐時のシミュレーションの並列化により最も性能を発揮し

表 5.10　上下動する床におけるバウンシングボールにおけるスレッド数に対する処理時間 (μsec)

スレッド数	シミュレーション全体		スレッド数	シミュレーション全体	
	平均値	標準誤差		平均値	標準誤差
1	82148081	182623	17	27299210	189457
2	63056523	62525	18	27211259	294301
3	54669840	36731	19	27023474	178464
4	45686095	32271	20	27341741	210871
5	44869109	34262	21	27927552	118971
6	40857736	53907	22	30283516	206082
7	39455936	65605	23	30109671	215053
8	39643894	46649	24	30688411	293300
9	39359358	57105	25	30745993	162834
10	35642723	108191	26	30800045	157566
11	35478257	144028	27	31001499	101033
12	28347797	86826	28	27078253	42997
13	28498868	114859	29	27110158	44604
14	29794148	117549	30	25899365	88039
15	26826771	176844	31	25951230	128189
16	26579092	176934	32	25962594	55496

```
INIT <=> y=10 & y’=0 & mag=0 & timer=0.
FALL <=> [](y’’=-10+mag).
TRUE <=> [](1=1).
TIMER <=> [](mag’=0&timer’=1).
SWITCHON <=> [](timer-=1=>mag=12&timer=0).
SWITCHOFF <=> [](timer-=1=>mag=0&timer=0).
SWITCHW <=> [](timer-=1=>mag=100&timer=0).

INIT,TIMER<<(SWITCHOFF,SWITCHON,SWITCHW)<<TRUE,FALL.
```

図 5.28　HydLa によるマグネットボールのモデル

表 5.11　マグネットボールにおけるスレッド数に対する処理時間 (μsec)

スレッド数	シミュレーション全体 平均値	シミュレーション全体 標準誤差	スレッド数	シミュレーション全体 平均値	シミュレーション全体 標準誤差
1	62458394	83542	17	13981980	151170
2	32364513	23085	18	14194384	117318
3	32379586	51696	19	14272766	77546
4	19055692	45700	20	14254144	241663
5	19161228	65700	21	14556980	170325
6	19336731	75655	22	14381876	144593
7	19357351	84106	23	14545043	124176
8	14708363	44082	24	14382833	233598
9	14785011	76883	25	14157001	156983
10	14899350	87492	26	15086911	185811
11	14741372	34316	27	14452677	120734
12	14910526	113017	28	14863493	246405
13	15009762	47677	29	14921906	152776
14	15098864	110782	30	14656788	180506
15	15277948	223889	31	14833395	262122
16	14141499	87355	32	14517130	195645

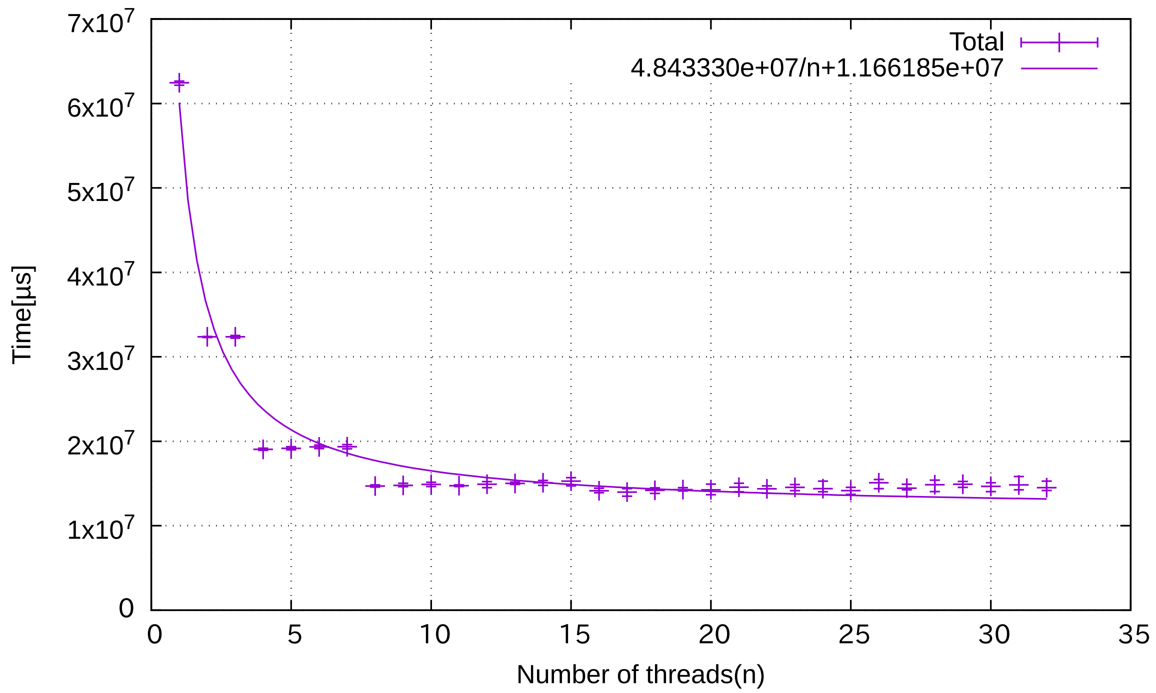

図 5.29　マグネットボールにおけるスレッド数に対する処理時間

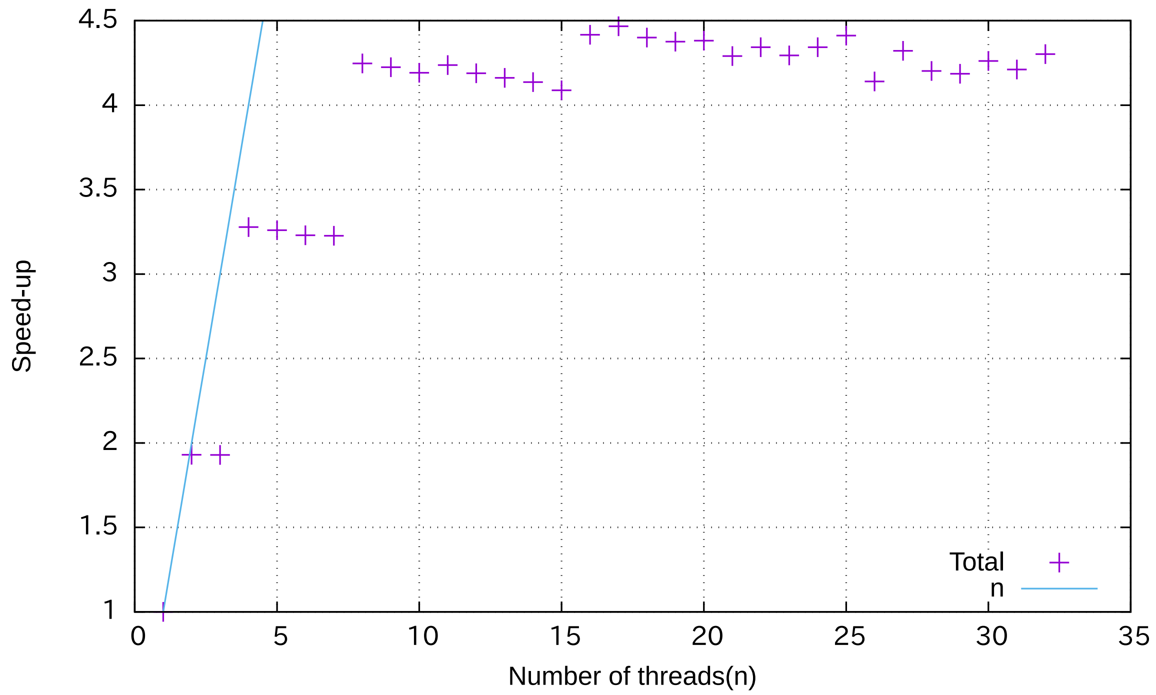

図 5.30　マグネットボールにおけるスレッド数に対する速度向上

た．これは，このモデルは 1 秒ごとに解起動が 3 通りに分岐するが数式は単純なもので各フェーズの計算が軽いため，ケース分岐の並列化の効果が発揮されやすいためだと考えられる．

第 6 章

まとめと今後の課題

本章では本論文のまとめと，今後の課題について説明する．

6.1　まとめ

本論文では，より規模の大きい問題や複雑なモデルに対応するために，HyLaGI を並列化により高速化することを目指した．数式処理を並列実行できるようにバックエンド部を改良し，これを利用してガード条件判定とケース分岐時のシミュレーションの並列化について，可能性と期待できる効果について議論した上で実装した．

並列化を施した処理系について性能評価を行ったところ，ガード条件判定に関しては概ね期待通りの性能を得られ，特にスレッド当たりガード条件の個数が 500 個を超えるような場合はスレッド数倍の速度が得られることが確認できた．また，ケース分岐時のシミュレーションの並列化についても，最大で 4.47 倍程度 (17 スレッド時) までシミュレーション全体の時間を高速化することに成功した．

6.2　今後の課題

今後の課題として，ケース分岐時のシミュレーションの並列化において分岐直後に別の解軌道のシミュレーションを開始できるようにすることで，現状の 1.65 倍から最大で 5.7 倍程度まで性能が向上することが予想される．HyLaGI の現在の実装は 1 スレッドでの逐次実行に最適化されており，計算結果の再利用などが全面で行われているため，無駄な計算を省きつつ並列動作させる実装を行うことは優先度の高い課題である．

さらに発展的な課題として，本研究で行った並列化を基礎とした更なるシミュレーショ

ンの並列化が挙げられる．例えばガード条件判定などの統計的に偏りのある条件分岐について，確率の高い方の分岐先の処理と条件判定を並列に実行する投機的実行を実装することで，使用していない CPU コアを活用しつつさらなる速度向上が期待できる．

また，モデルを空間的・時間的に分割して並列にシミュレーションを行うことも考えられる．モデル内で空間的・時間的に独立して他の部分の影響を全くあるいはほとんど受けない箇所を見つけるか，指定する方法を定義する必要があると考えられる．

謝辞

本研究を進めるにあたり様々な方の指導，助言をいただきました．まず, 3 年間に渡りご指導してくださった上田和紀教授に深く感謝致します．また，実装に伴い数多くの助言・手助けをしていただいた松本翔太さんを始めとした先輩方に感謝致します．さらに，辛い場面でもお互いに支えあった研究室での同輩・後輩に感謝致します．最後に，学費面で支援していただき，学生生活を支えてくださった両親及び祖母に感謝致します．

2016 年 1 月　伊藤 剛史

参考文献

[1] Lunze, J.: Handbook of Hybrid Systems Control: Theory, Tools, Applications, Cambridge University Press, 2009

[2] 上田和紀, 石井大輔, 細部博史: ハイブリッド制約言語 HydLa の宣言的意味論, コンピュータソフトウェア, Vol. 28 No. 1, 2011, pp. 306–311, doi:10.11309/jssst.28.1_306.

[3] Borning, A., Feldman-Benson, B., Wilson, M.: Constraint Hierarchies. Lisp and Symbolic Computation, Vol.5, No.3(1992), Kluwer Academic Publishers, pp. 223–270, doi:10.1007/BF01807506.

[4] 廣瀬賢一, 大谷順司, 石井大輔, 細部博史, 上田和紀: 制約階層によるハイブリッドシステムのモデリング手法, 日本ソフトウェア科学会第 26 回大会, 2D-2, 2009.

[5] 河野文彦: リスト記法の導入と動的な制約集合導出によるハイブリッド制約言語 HydLa とその処理系のスケーラビリティ向上, 早稲田大学大学院基幹理工学研究科, 修士論文, 2014, pp.20–21.

[6] 河野文彦, 小林輝哉, 松本翔太, 上田和紀: 数式処理に基づくハイブリッドシステムシミュレータ Hyrose の大規模モデルシミュレーションに向けた拡張，日本ソフトウェア科学会第 31 回大会, 2014.

[7] 松本翔太, 上田和紀: ハイブリッド制約言語 HydLa の記号実行シミュレータ Hyrose, コンピュータソフトウェア, Vol.30, No.4, 2013, pp.18–35, doi:10.11309/jssst.30.4_18.

[8] Taha, W., Brauner, P., Yingfu Zeng, Cartwright, R., Gaspes, V., Ames, A., Chapoutot, A: A Core Language for Executable Models of Cyber-Physical Systems (Preliminary Report). Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on, 2012, pp.303–308, doi:10.1109/ICDCSW.2012.72.

[9] Chen, X., ÁBrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. Computer Aided Verification. Springer Berlin Heidelberg, 2013, pp.258–263, doi:10.1007/978-3-642-39799-8_18.

[10] Carlson, B., Gupta, V.: Hybrid cc with interval constraints, Hybrid Systems 98: Computation and Control, Springer Verlag, 1998, pp.80–94, doi:10.1007/3-540-64358-3_33.

[11] Platzer, A., Quesel, J: KeYmaera: A Hybrid Theorem Prover for Hybrid Systems (System Description). In IJCAR 2008, LNCS 5195, Springer Berlin Heidelberg, 2008, pp.171–178, doi:10.1007/978-3-540-71070-7_15.

[12] Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O: SpaceEx: Scalable Verification of Hybrid Systems. Proc. 23rd International Conference on Computer Aided Verification, Springer, 2011, pp.379–395, doi:10.1007/978-3-642-22110-1_30.

[13] 和田努, 松本翔太, 上田和紀: ハイブリッド制約言語 HydLa 処理系における数式処理と区間計算を組み合わせたシミュレーション実行, 2015 年度 人工知能学会全国大会（第 29 回）, 1E3-3, 2015.

[14] 小林輝哉, 河野文彦，松本翔太, 上田和紀: フェーズ間の制約差分情報および制約-変数間の依存関係を用いた HydLa 処理系の最適化，人工知能学会第 28 回全国大会, 4C1-2, 2014.

[15] Wolfram Research, Inc.: Mathematica, `https://www.wolfram.com/mathematica/index.html`, 2016

[16] Wolfram Research, Inc.: Wolfram Mathematica Tutorial Collection, Mathlink Development in C, `https://www.wolfram.com/learningcenter/tutorialcollection/MathLinkDevelopmentInC/MathLinkDevelopmentInC.pdf`

[17] Lygeros, J., Tomlin, C., Sastry, S.: Hybrid Systems: Modeling, Analysis and Control, 2008. pp.56–57.

発表文献

[1] 伊藤剛史，松本翔太，上田和紀. ハイブリッド制約処理系 HyLaGI の並列化. 情報処理学会 第 78 回全国大会. 4H-3, 2016, 登壇発表. (予定)

[2] 若槻祐彰，松本翔太，伊藤剛史，和田努，上田和紀. ハイブリッド制約処理系 HyLaGI による微笑誤差を用いたモデル解析. 日本ソフトウェア科学会 第 32 回大会. 2015，登壇発表.（共著）

[3] 伊藤剛史，松本翔太，上田和紀. ハイブリッド制約処理系 Hyrose における無矛盾性判定の並列化による高速化. 2014, ポスター発表.