

卒業論文

神経回路シミュレーションにおける イオンチャネルダイナミクス計算の 最適化コードの自動生成

平成 30 年 1 月 29 日提出

指導教員 神崎亮平 教授

東京大学工学部機械情報工学科

03-150268 井上裕太

目 次

1 序論	6
1.1 神経科学においてシミュレーションを行う意義	6
1.2 神経回路シミュレーションの高速化・最適化への需要	6
1.3 先行研究による知見	6
1.3.1 神経回路シミュレーションの手動での最適化	6
1.3.2 神経回路シミュレーションのベンチマークモデル	7
1.4 研究の目的と手法	9
1.5 本論文の構成	9
2 シミュレーションのモデルと環境	10
2.1 シミュレーションモデル	10
2.1.1 Hodgkin-Huxley マルチコンパートメントモデル	10
2.1.2 Pas モデル	12
2.2 シミュレータ	12
2.2.1 全体構成	13
2.2.2 NEURON のコンパイル	16
2.3 シミュレーション環境	17
2.3.1 計算機性能	17
2.3.2 ジョブ実行環境	17
3 最適化の手法	24
3.1 モデルに依存するパラメータ	24
3.1.1 SIMD 化	26
3.1.2 配列構造	28
3.2 実行マシンに依存するパラメータ	31
3.2.1 ハイブリッド並列化	31
3.2.2 配列サイズの変形	32
3.3 コンパイルに関わるパラメータ	33
4 自動チューニングスクリプトと MOD トランスパイラの構築	33
4.1 環境設定スクリプト	33
4.2 シミュレータ	35
4.2.1 事前準備	37
4.2.2 config ファイルのパース	37
4.2.3 MOD ファイルをパースし抽象木を生成する	39
4.2.4 パラメータ候補群の生成	39
4.2.5 パラメータ候補群に対してシミュレーションを実行	40
4.2.6 ジョブ実行の監視	44
4.2.7 シミュレーションの再実行	50
4.3 トランスパイラ	50
4.3.1 nmodl	51

4.3.2 textX	51
4.3.3 構成	53
5 シミュレーション結果	61
5.1 小規模シミュレーションでのパラメータ比較	61
5.1.1 クラスタ環境	61
5.1.2 京環境	66
5.2 MPI プロセス数	66
5.3 OpenMP スレッド数	66
5.4 SIMD 化	66
5.5 配列のくくり出し	66
5.6 最適化結果の比較	66
6 考察	66
7 結論	67
8 謝辞	69

図 目 次

1	ベンチマークに用いたネットワーク	8
2	Hodgkin-Huxley モデルの細胞膜の等価回路	12
3	マルチコンパートメントモデル (Yusuke Mori. Developing implementations of the estimation of synaptic positions and the communication procedure between simulation and real environment toward whole insect brain simulation. Master's thesis, the university of Tokyo, 2014. より引用)	14
4	NEURON の全体構成	15
5	スーパーコンピュータ京 (提供: 理化学研究所)	18
6	SIMD 命令	27
7	FMA	28
8	配列のくくりだし	30
9	Genie の全体像	36
10	Hodgkin-Huxley の MOD ファイルの抽象木	52
11	変換された C 言語ファイルの構成	53
12	トランスペイラ 構成	55
13	クラスタ 小規模シミュレーション結果	63
14	クラスタ 小規模シミュレーション結果 上位 25%	64
15	クラスタ 小規模シミュレーション結果 パラメータ絞り込み後	65

表 目 次

1	京計算ノード構成	17
2	京プロセッサ構成	19
3	クラスタ性能	19
4	京でのジョブ関連コマンド	19
5	京でのジョブの状態	20
6	クラスタでのジョブ関連コマンド	21
7	クラスタでのジョブの状態	23
8	クラスタでのパラメータ	61
9	クラスタでの絞り込み後のパラメータ	62
10	京でのパラメータ	66

1 序論

1.1 神経科学においてシミュレーションを行う意義

神経科学分野において生物の脳機能を解明することは主たる目的である。それは、人間を含む生物の知能の理解という人間の本源的な要求の発露のみならず、情報処理プログラムのアルゴリズムやロボット、マンマシンインターフェースのような工学応用、アルツハイマー病やパーキンソン病といった疾患の治療や脳の心の健康科学さらにニューロンエンハンスメントによる知能増強まで見通す医学分野の発展にも大きな貢献をすることが期待される。一方で、生物の脳では非常に短い時間で膨大な情報が処理されており例えば人間の脳の情報処理は $10^{21} FLOPS$ だという見積もりがある（計算科学ロードマップ白書 URL は検索して）。これは非常に大きな値にみえるが、コンピュータの能力は年々進歩をつづけており、日本においても京コンピュータにおいて $10^{16} FLOPS$ が 2012 年に達成されて、2020 年に $10^{18} FLOPS$ を目指すポスト京の開発が進んでいることを考えればスーパーコンピュータの利用が前提となるとはいえた生物の脳全体のシミュレーションも神経科学の視野にはいっているといえる。トップダウン的な脳の構築原理が明らかになってない以上、様々な生物実験の結果を微視的なレベルから入れ込んでいく ボトムアップアプローチによってシミュレーションを行い、しかるのちに現実には観察が難しい脳機能の細部までもシミュレーションを介してでも観察できる環境を構築することは大きな意義を持つ。

1.2 神経回路シミュレーションの高速化・最適化への需要

しかしながら生物の脳機能は元来単純な一つモデルで表すことができるものではなく、その機能の解明には多数のモデルを混在させることが必要になるであろうと考えられる。現在 modelDB (TODO: modelDB のレファレンス) をはじめとして、脳機能・イオンチャネルのモデルは多く存在しているが、それらは速度チューニングされていないものであり、スーパーコンピュータのような高価な計算資源でシミュレーションを行おうとする場合、限られた資源を有効に活用するため高速化することが望ましい。一方で、多数のモデルを一つのシミュレーションが含むことを想定すればそれらのモデルすべてに対し手動で高速化を行うには膨大な時間と労力を必要とする。もしくは対象のモデルが一つであってもモデリングする人間が計算チューニングに慣れているとは限らず、そのため多くのコストがかかる可能性がある。それゆえ、実験データから脳機能を再構築するボトムアップアプローチを取るにあたり、モデルを自動的に高速化する手段を開発することは大きな意義を持つ。そこで、本研究では個々のイオンチャネルモデルを自動でシミュレーションを実行する計算機に合わせて最適化するソフトウェアを作成することで、これまで人の手で逐次行われてきた最適化の汎用化を目指す。

1.3 先行研究による知見

1.3.1 神経回路シミュレーションの手動での最適化

神経回路シミュレーションを手動で最適化する取り組みは当研究室でも行われてきた。宮本らの研究 [1] では、神経回路シミュレーションを行うためのソフトウェアである NEURON の上でイオンチャネルモデルの一つである Hodgkin-Huxley 型モデルを対象として最適化を行った。NEURON の詳細については後述 (TODO: NEURON の節にレファレンス) するが、NEURON では MOD ファイルと呼ばれる神経細胞モデルを記述するファイルが C 言語ファイルに変換され実行形式が生成される。しかしながらここで生成される C 言語ファイルには無駄が多いため、生成された

ファイルをより計算機やモデルの特性にあった形に手動で修正することが可能である。
(TODO: 図) 宮本らの研究において対象とした計算機はスーパーコンピュータ京であり, 京上で十分な性能を発揮するため以下にあげるような最適化を行った.

1. 常微分方程式の変数を配列化することによる SIMD 化
2. 配列構造のくくり出し.
3. OpenMP と MPI を用いてのハイブリッド並列化.
4. ノード配置の最適化.

この中で, 本研究では 1-3 を対象として自動最適化を目指す.

1.3.2 神経回路シミュレーションのベンチマークモデル

1.3.1において行った最適化の効果を定性的に評価する必要があった.
そのため宮本らは神経回路シミュレーションのベンチマークモデルを作成し, 次に示す 3 種類のネットワークを構築することで最適化の効果を測った.

1. リングネットワーク
2. ランダムネットワーク
3. Watts and Strogatz ネットワーク

以下に, 各ネットワークの概要と疑似コードを示す. また, 疑似コード内での定数と関数については

1. NCELL : 細胞数
2. NSYNAPSE : 細胞 1 つあたりのシナプス数
3. RND : 1 以上かつ細胞数より少ない整数の乱数
4. makeSynapse(X, Y) : シナプス前末端を細胞 X, シナプス後末端を細胞 Y に作成し接続する

とし, 細胞内のシナプスの位置はランダムとした.

1.3.2.1 リングネットワーク

隣り合った細胞同士でシナプスを作成する.

Listing 1: リングネットワークの作成

```
for (int i=0; i<NCELL; i++) {
    for (int j=0; j<NSYNAPSE; j++) {
        makeSynapse(i, i+1 mod NCELL);
    }
}
```

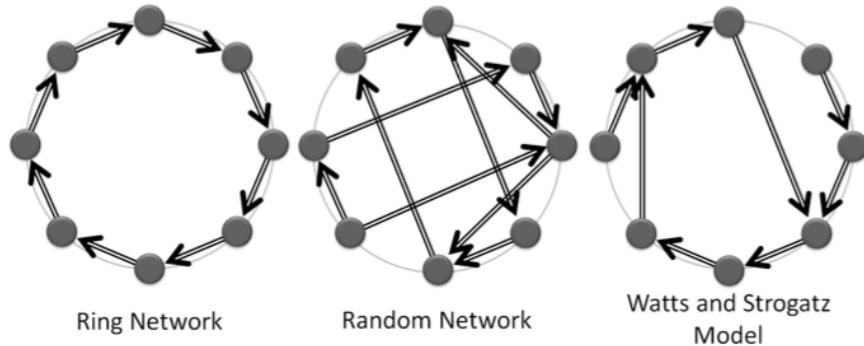


図 1: ベンチマークに用いたネットワーク

1.3.2.2 ランダムワーク

シナプスの前末端はリングネットワークと同様で、シナプスの後末端の位置がランダムなネットワーク。

Listing 2: ランダムネットワークの作成

```

for (int i=0; i<NCELL; i++) {
    for (int j=0; j<NSYNAPSE; j++) {
        makeSynapse(i, (i+RND) mod NCELL);
    }
}

```

1.3.2.3 Watts and Strogatz ワーク

実際の神経回路ネットワークは、リングでも完全なランダムでもないと考えられる。そこで、それら2つのネットワークの中間的なネットワークに位置付けられるWatts and Strogatzネットワークについてもベンチマークの対象としていた。Watts and Strogatzネットワークは、リングネットワークを作成した後、確率 $p \in [0, 1]$ で後末端をランダムに繋ぎかえるものである。

Listing 3: Watts and Strogatz ネットワークの作成

```

P = NSYNAPSE * p;
for (int i=0; i<NCELL; i++) {
    for (int j=0; j<P; j++) {
        makeSynapse(i, (i+RND) mod NCELL);
    }
    for (int j=P; j<NSYNAPSE; j++) {
        makeSynapse(i, i+1 mod NCELL);
    }
}

```

1.4 研究の目的と手法

以上のことと踏まえ、本研究の目的は、

1. NEURON 上でのシミュレーションを自動で最適化するために、最適化に用いることのできるパラメータの候補をパラメータの定義ファイルを作成、または MOD ファイルやマシンを解析することによって生成する。
2. 与えられた最適化のパラメータを用いて、MOD ファイルから最適化された C 言語ファイルを生成するトランスペイラを作成すること。
3. 1 で生成したパラメータ候補に対し、2 を利用して C 言語ファイルを生成し、シミュレーションを実行とシミュレーションの結果を自動で行うシミュレータを作成すること。

の 3 点とする。

1.5 本論文の構成

本論文は全 6 章から構成されている。

本章では本研究の背景と目的を示した。

第 2 章では、本研究が対象とする神経回路シミュレーションの系、そしてシミュレーションを行う環境について述べる。

第 3 章では、本研究で作成したプログラムについての詳細を述べる。

第 4 章では、シミュレーションの結果を示す。

第 5 章では、シミュレーション結果の考察を述べる。

第 6 章では、本研究のまとめ、成果を示した上で将来の課題について述べる。

2 シミュレーションのモデルと環境

2.1 シミュレーションモデル

本研究では、先行研究として触れた宮本らが手動で行った最適化を自動で行うことの目的としているため、最適化の対象となるシミュレーションモデルは同じものを採用した。

2.1.1 Hodgkin-Huxley マルチコンパートメントモデル

2.1.1.1 Hodgkin-Huxley モデル

神経系における情報の伝達は、ニューロンの電気的な活動によって行われる。こうしたニューロンの電気的活動を表現する方法として、1952年にHodgkinとHuxleyによってヤリイカの神経の活動電位の研究を基にした微分方程式のモデルが考案された。

Hodgkin-Huxley モデルでは、各イオンに対するニューロンの細胞膜の等価性を基に、ニューロンの電気的活動を微分方程式を用いて表現している。

他のモデルと比べ計算量が多い一方、実際の生物の神経系の働きに近くたいていのイオンチャネルのモデルを表すことができるという特徴を持っている。

モデルの定式化

ニューロンはイオンを通さない脂質二重膜によって構成され、特定のイオンを選択的に透過させるチャネルと呼ばれるタンパク質が膜上に分布している。この時、図2に示したHodgkin-Huxley モデルの細胞膜の等価回路のように、神経細胞を細胞膜を容量 C_M のキャパシタ、チャネルを透過するイオンの流れを抵抗 R_i （コンダクタンスを用いると $R_i = \frac{1}{g_i}$ と表わせる）、起電力 E_i の電池からなる電気回路と見做すと、膜を流れる全電流 I_M は、

$$I_M = C_M \frac{dV}{dt} + \sum_i I_i \quad (1)$$

$$I_i = g_i(V - E_i) \quad (2)$$

(3)

と膜電位 V に関する常微分方程式の形で表すことができる。尚、式中の i はイオンチャネルの種類を、 E_i はそれぞれのイオンに特有な平衡電位を表している。

また、前述した抵抗の逆数であるコンダクタンス g_i は

$$g_i(V, t) = \bar{g}_i m(V, t)^x h(V, t)^y \quad (4)$$

(5)

と記述される。

\bar{g}_i は g_i の最大値であり、 $m, h \in (0, 1)$ はそれぞれ活性化パラメータ、不活性化パラメータと呼ばれる無次元数である。

また、 x, y はそれぞれ実験データを元に求められる整数である。

活性化パラメータ、不活性化パラメータである m, h はさらにそれぞれ次の微分方程式（式（TODO:

式番号)) で表現される。

$$\frac{m(V,t)}{dt} = \frac{m_\infty(V) - m(V,t)}{\tau_m(V)} \quad (6)$$

$$\frac{h(V,t)}{dt} = \frac{h_\infty(V) - h(V,t)}{\tau_h(V)} \quad (7)$$

$$m_\infty(V) = \frac{1}{1 + \exp(\frac{V - \Theta_m}{k_m})} \quad (8)$$

$$h_\infty(V) = \frac{1}{1 + \exp(\frac{V - \Theta_h}{k_h})} \quad (9)$$

$$\tau_m(V) = \tau_{m_0} + \frac{\tau_{m_1}}{\exp(\frac{V - \Theta_{m_1}}{\sigma_{m_1}}) + \exp(-\frac{V - \Theta_{m_2}}{\sigma_{m_2}})} \quad (10)$$

$$\tau_h(V) = \tau_{h_0} + \frac{\tau_{h_1}}{\exp(\frac{V - \Theta_{h_1}}{\sigma_{h_1}}) + \exp(-\frac{V - \Theta_{h_2}}{\sigma_{h_2}})} \quad (11)$$

(12)

この式中のパラメータはチャネルのキネティクスに関わるパラメータであり、実験データから算出される。

Hodgkin-Huxley モデルを構成する各イオンチャネル

Hodgkin と Huxley が提唱したモデルにおいて、モデルは電位依存性の Na チャネルと K チャネル、常に開いた状態であり静止膜電位を決定するリークチャネルの 3 種類のチャネルから構成されていた。また、各チャネルにおける電流は次の式で表せられる。

$$I_{Na} = \bar{g}_{Na} m^3 h (V - E_{Na}) \quad (13)$$

$$I_K = \bar{g}_K n^4 (V - E_K) \quad (14)$$

$$I_l = g_l (V - E_l) \quad (15)$$

(16)

2.1.1.2 マルチコンパートメントモデル 2.1.1.1 で述べた Hodgkin-Huxley モデルは細胞を一つの回路と見做してモデル化しており、その際に細胞の形態は考慮されていない。しかしながら、実際の細胞は複雑な形態を有しており、細胞全体ではなく細胞の一部のみが発火するような応答を示すことが知られている（TODO add ref）ため、より詳細なシミュレーションを行うためには細胞の形態についてもモデル化する必要がある。

そこで今回マルチコンパートメントモデルという神経細胞の 3 次元的な特性を表現するモデルを導入する。

マルチコンパートメントモデルでは、図 3 のように神経細胞の形態を多数のシリンダーの連なりとして表現する。各シリンダをコンパートメントと呼び、前節で示した Hodgkin-Huxley モデルの計算はコンパートメント単位で行う。すなわち、n 番目のコンパートメントの親となるコンパートメントを n_{parent} とし、子となるコンパートメントをそれぞれ n_{child_i} とするとき、マルチコンパートメント

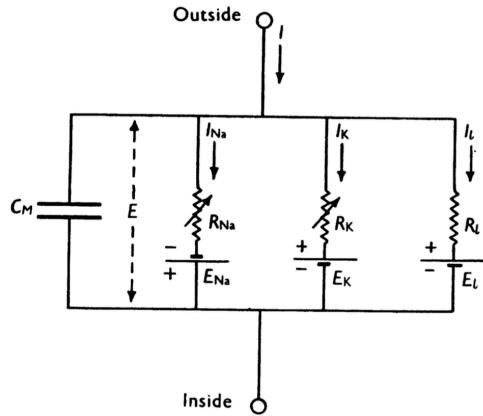


図 2: Hodgkin-Huxley モデルの細胞膜の等価回路

Hodgkin-Huxley モデルにおける式 (1) は,

$$I_m^n = C_m^n \frac{dV}{dt} + \sum_i I_i^n + I^{n_{parent}, n} - \sum_i I^{n, n_{child_i}} \quad (17)$$

(18)

と表される. 尚, $I^{i,j}$ は i 番目のコンパートメントから j 番目のコンパートメントに流れる電流を表す.
(TODO : 図の差し替え)

2.1.2 Pas モデル

(TODO : 時間がなければ消す or 付録としてくっつける)

2.2 シミュレータ

NEURON は, Yale 大学の Hines らによって開発されている神経回路・細胞シミュレーションソフトウェアであり, 神経回路シミュレーションにおいて標準の一つとなっており, 先行研究としてあげた宮本らによる手動での高速化においても対象となったソフトウェアである. そのため, 本研究の目的である神経回路シミュレーションの自動最適化の対象として NEURON を採用した. また, 京や

クラスタといった複数の計算機上で安定して稼働させるために NEURON のバージョンは 7.2 を選択した。

2.2.1 全体構成

NEURON では,MOD ファイルと HOC ファイルと呼ばれる二つのファイルに必要な情報を記述することで神経回路シミュレーションを行っている。

MOD ファイルはその名のとおり神経細胞のモデルを記述するファイルであり,(TODO : 章番号) で示したように神経細胞を数理モデルとして記述する。

一方で HOC ファイルと呼ばれるファイルには MOD ファイルで記述された神経細胞モデル間のつながりや, シミュレーション時間などシミュレーションそのものに関与する設定を記述する。

より具体的には, 図 (TODO: 番号) で示したように, nrnivmodl と呼ばれるトランスペイラによって MOD ファイルは対応する C ファイルに変換される。この C ファイルはさらに GCC や ICC といった C 言語のコンパイラによってオブジェクトファイルになり, ここで生成されたオブジェクトファイルと NEURON 本体がリンクされることによって NEURON の実行形式が作成されることになる。そのため, MOD ファイルとして利用者が作成したモデルは実行時には NEURON に組み込まれることとなる。

最終的にこうして生成された NEURON の実行形式に対してシミュレーションの情報を記述した HOC ファイルを渡すことでシミュレーションが実行される。

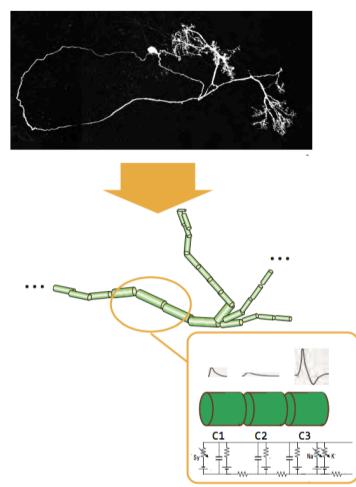


図3: マルチコンパートメントモデル (Yusuke Mori. Developing implementations of the estimation of synaptic positions and the communication procedure between simulation and real environment toward whole insect brain simulation. Master's thesis, the university of Tokyo, 2014. より引用)

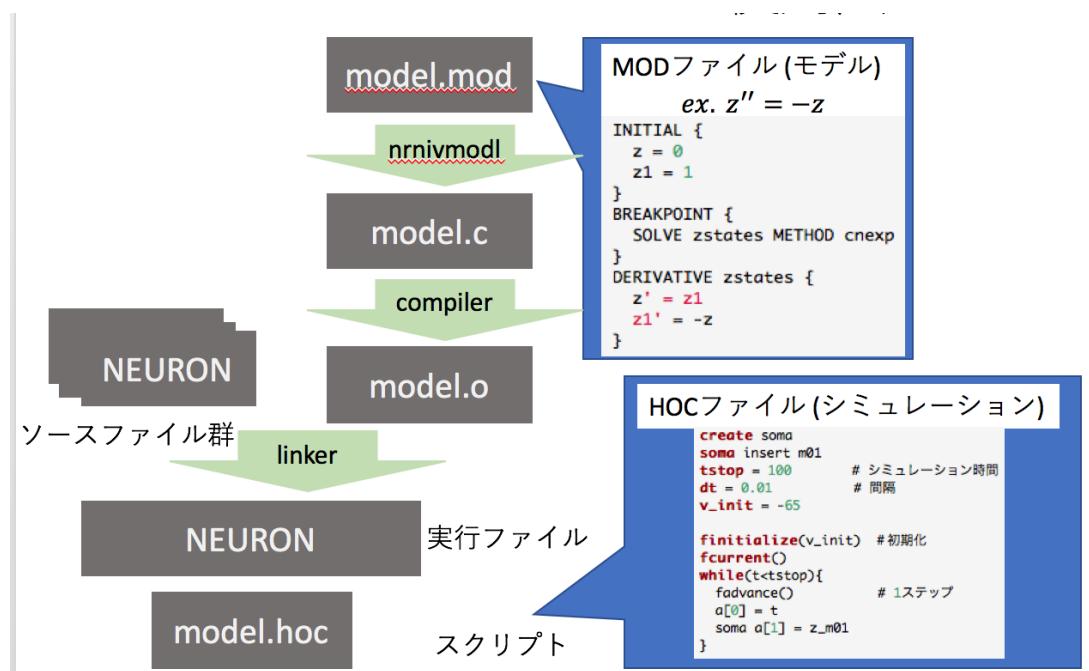


図 4: NEURON の全体構成

宮本らによる先行研究[1]では、主にこのMODファイルからCファイルへの変換に着目し、nrnivmodlによって作成されたCファイルを手動にて最適化することでシミュレーション全体の高速化を達成した。

そのため、本研究においてはnrnivmodlに変わるトランスマッピングを作成し自動での高速化を図る。

2.2.2 NEURON のコンパイル

2.2.2.1 クラスタ上でのコンパイル

クラスタのようなx86の大規模計算機では、

Listing 4: クラスタでのNEURONのコンパイル

```
$ ./configure --prefix='pwd' \
    --without-iv --without-x --without-nrnoc-x11
$ make
$ make install
```

することで計算シミュレータとしてのNEURONをコンパイルし、実行形式を得ることができる。デフォルトのコンパイルオプションではNEURONはGUI関係のライブラリもリンクするが、大規模計算機環境上では必要ないためオプションを渡すことでコンパイル対象から除外している。

2.2.2.2 京上のコンパイル

一方で、京ではログインノードと呼ばれるNEURONのコンパイルを行う環境(x86)とプログラムを実行する環境(sparc64)が異なるため、クロスコンパイルを行う必要がある。

NEURONのコンパイルは内部的には、

1. MODコンパイラ(nmodl)をコンパイルし実行形式を生成
2. MODコンパイラがMODファイルをC言語ファイルに変換した上でコンパイル
3. NEURON本体(nrniv, nrnoc)をコンパイルする
4. 上述の2と3をリンクさせ、実行形式を作成

という手順を踏んでいるが、この中でMODコンパイラ作成についてはログインノード(x86)で実行する必要があるため、1についてネイティブコンパイルで、2,3,4についてはクロスコンパイルで行う必要がある。

またGUI関係のライブラリについてはクラスタ同様京でも必要ないため除外する(宮本修論より引用)

nmodlのコンパイル

Listing 5: 京でのnmodlのコンパイル

```
$ ./configure --prefix='pwd' \
    --without-iv --without-x --without-nrnoc-x11 \
    --with-nmodl-only linux_nrnmech=no \
    CC=gcc CXX=g++
```

ピーク演算性能	10.62PFLOPS
メモリ総容量	1.26PB (ノードあたり 16GB)
計算ノード間ネットワーク	6 次元メッシュ/トーラス (ユーザービューは 3 次元トーラス)
帯域	3 次元の正負各方向にそれぞれ 5GB/s × 2 (双方向)

表 1: 京計算ノード構成

```
$ make
$ make install
```

NEURON のクロスコンパイル

NEURON 本体をクロスコンパイルする前に、上述した nmodl のコンパイルで生成した実行形式を PATH の通っているディレクトリに移動させておく必要がある。nmodl を退避させたのち、下記のコマンドを実行することで NEURON 本体の実行形式が生成される

Listing 6: 京での NEURON 本体のコンパイル

```
$ make clean
$ ./configure --prefix='pwd' \
  --without-x --without-nmodl \
  --host=sparc64-unknown-linux-gnu --build=x86_64-unknown-linux-gnu \
  --without-iv --without-nrnoc-x11 \
  --enable-shared=no --enable-static=yes \
  --with-paranrn --with-mpi --with-multisend \
  linux_nrnmech=no use_pthread=no \
  CC=mpifccpx CXX=mpiFCCpx MPICC=mpifccpx MPICXX=mpiFCCpx
$ make
$ make install
```

2.3 シミュレーション環境

2.3.1 計算機性能

本研究で使用した計算機は、スーパーコンピュータ「京」（以下京、図 5）と研究室クラスタ（以下クラスタ、TODO: 図）である。（TODO: 京についての説明）
京とクラスタの性能諸元と表（TODO: 表番号）に記す [2]。

2.3.2 ジョブ実行環境

京に代表される大型コンピュータの場合、複数の利用者が共同で利用することが基本となる。そのため各個人が各自勝手にプログラムを実行すると、計算が集中することで処理限界を超えてしまったり、逆に全く利用されない時間などが現れてしまい計算資源を有効に活用できない。そのため、



©RIKEN

図 5: スーパーコンピュータ京（提供: 理化学研究所）

大型コンピュータではキューリングシステムを利用してプログラムが実行される。

キューリングシステムにおける一度のプログラム実行の単位はジョブと呼ばれ、プログラムを実行する際に必要なノード数、メモリ、実行するプログラムのパスや前処理といった情報を書き込んだジョブスクリプトを作成し、キューリングシステムにジョブスクリプトをサブミットすることでプログラムが実行される。

2.3.2.1 京でのジョブの実行

Listing 7: 京のジョブスクリプト例

```

1 #!/bin/bash -x
2 ##### ノードの数を指定#####
3 #PJM --rsc-list "node=8"
4 ##### プログラムの実行時間を指定#####
5 #PJM --rsc-list "elapse=00:10:00"
6 ##### 利用するリソースグループを指定#####
7 #PJM --rsc-list "rscgrp=small"
8 ##### ノード内のプロセス数を指定#####
9 #PJM --mpi "proc=64"
10 #PJM -s
11 ##### ステージングの条件を設定#####
12 #PJM --stg-transfiles all
13 #PJM --mpi "use-rankdir"
14 ##### ステージインする際の基本となるディレクトリを指定#####
15 #PJM --stgin-basedir /home/user/neuron_kplus

```

CPU 性能	128GFLOPS (16GFLOPS × 8 コア)
コア数	8 個
浮動小数点演算器構成 (コアあたり)	積和演算器: 4 (2×2 個 SIMD), (逆数近似命令: SIMD 動作) 除算器: 2 個 比較器: 2 個
	浮動小数点レジスタ (64 ビット) : 256 本 グローバルレジスタ (64 ビット) : 188 本
キャッシュ構成	1 次命令キャッシュ: 32KB(2way), 1 次データキャッシュ: 32KB(2-way), 2 次キャッシュ: 6MB(12-way) コア間共有
メモリ帯域	64GB/s (理論ピーク値)
動作周波数	2GHz
ダイサイズ	22.7mm × 22.6mm
トランジスタ数	約 7 億 6000 万個
消費電力	58W (プロセス条件 TYP)

表 2: 京プロセッサ構成

表 3: クラスタ性能

```

16 #-----ステージアウトする先のディレクトリを設定-----
17 #-PJM --stgout "rank=* %r:..//prof/* /data/user/log/"
18 #-----ステージインするファイル群をランクごとに指定-----
19 #PJM --stgin "rank=* ./stgin/* %r:.."
20 #PJM --stgin "rank=* ./specials/sparc64/special %r:.."
21 #PJM --stgin "rank=* ./hoc/* %r:.."
22 #-----環境変数の設定-----
23 ./work/system/Env_base
24
25 #-----OpenMP のスレッド数を指定-----
26 export OMP_NUM_THREADS=1
27 #-----利用するNEURON の実行形式のパスとオプションを指定する-----
28 NRNIV=".//special -mpi"
29
30 #-----シミュレーションファイルを指定する-----
31 HOC_NAME=".//bench_main.hoc"
32 #-----シミュレーションに与えるオプション-----
33 NRNOPT=\
```

表 4: 京でのジョブ関連コマンド

コマンド	説明
pbsub	pbsub サブミットするスクリプトのパスとすることでジョブをキューシステムに登録し, ジョブ ID を出力する。
pjdel	pjdel ジョブ ID とすることで現在実行中または待機中のジョブを停止・削除する。
pjstat	現在実行または待機中のジョブの一覧を表示する

```

34 " -c MODEL=2" \
35 " -c NSTIM_POS=1" \
36 " -c NSTIM_NUM=400" \
37 " -c NCELLS=256" \
38 " -c NSYNAPSE=10" \
39 " -c SYNAPSE_RANGE=1" \
40 " -c NETWORK=1" \
41 " -c STOPTIME=200" \
42 " -c NTHREAD=1" \
43 " -c MULTISPLIT=0" \
44 " -c SPIKE_COMPRESS=0" \
45 " -c CACHE_EFFICIENT=1" \
46 " -c SHOW_SPIKE=1"
47
48 #----- プログラムを実行する際にNEURONに渡すオプション-----#
49 LPG="lpgparm -t 4MB -s 4MB -d 4MB -h 4MB -p 4MB"
50 #----- プログラムを実行する際にNEURONに渡すオプション-----#
51 MPIEXEC="mpiexec -mca mpi_print_stats 1"
52 #----- プロファイラを指定 (gprofなど) -----#
53 PROF=""
54 #----- プログラム実行時のコマンドを出力-----#
55 echo "${PROF} ${MPIEXEC} ${LPG} ${NRNIV} ${NRNOPT} ${HOC_NAME}"
56 #----- プログラムを実行-----#
57 time ${PROF} ${MPIEXEC} ${LPG} ${NRNIV} ${NRNOPT} ${HOC_NAME}
58
59 sync

```

Listing 8: 京でのコマンド実行例

```

$ pjsub job.sh
[INFO] PJM 0000 pjsub Job 7129316 submitted.

$ pjstat
ACCEPT QUEUED STGIN READY RUNNING RUNOUT STGOUT HOLD ERROR TOTAL
 0 1 0 0 0 0 0 0 0 1
s 0 1 0 0 0 0 0 0 0 1

JOB_ID JOB_NAME MD ST USER GROUP START_DATE ELAPSE_TIM NODE_REQUIRE
 RSC_GRP SHORT_RES
7129316 job.sh NM QUE user group [--/-- --:--:--] 0000:00:00 1:- small -

```

表 5: 京でのジョブの状態

ジョブのステータス	説明
QUE	ジョブキューで待機中.
STI	ジョブの実行に必要なファイルをステージインしている.
RUN	ジョブを実行中.
STO	ジョブの実行結果をステージアウトしている.

2.3.2.2 クラスタでのジョブの実行

表 6: クラスタでのジョブ関連コマンド

コマンド	説明
qsub	qsub サブミットするスクリプトのパスとすることでジョブをキューシステムに登録し, ジョブ ID を出力する.
qdel	pjdel ジョブ ID とすることで現在実行中または待機中のジョブを停止・削除する.
qstat	現在実行または待機中のジョブの一覧を表示する

Listing 9: クラスタのジョブスクリプト例

```
1 #!/bin/sh
2 #----- ノードの数、ノード内のプロセス数を指定-----#
3 #PBS -l nodes=1:ppn=4
4 #PBS -q cluster
5 #----- のスレッド数を指定する OpenMP -----#
6 export OMP_NUM_THREADS=2
7 #----- 利用するの実行形式のパスとオプションを指定する NEURON -----#
8 NRNIV="../specials/x86_64/special_mpi"
9 #----- シミュレーションファイルを指定する-----#
10 HOC_NAME="../hoc/bench_main.hoc"
11 #----- シミュレーションに与えるオプション-----#
12 NRNOPT=\"
13 " -c MODEL=2"\ \
14 " -c NSTIM_POS=1"\ \
15 " -c NSTIM_NUM=400"\ \
16 " -c NCELLS=256"\ \
17 " -c NSYNAPSE=10"\ \
18 " -c SYNAPSE_RANGE=1"\ \
19 " -c NETWORK=1"\ \
20 " -c STOPTIME=50"\ \
21 " -c NTHREAD=16"\ \
22 " -c MULTISPLIT=0"\ \
23 " -c SPIKE_COMPRESS=0"\ \
24 " -c CACHE_EFFICIENT=1"\ \
25 " -c SHOW_SPIKE=1"
26 #----- シミュレーションに与えるオプション-----#
27 MPIEXEC="mpiexec -mca mpi_print_stats 1"
28 #----- プロファイラを指定する（など）gprof -----#
29 PROF=""
30 #----- プログラム実行の際のカレントディレクトリ-----#
31 cd $PBS_O_WORKDIR
32 #----- プログラム実行時のコマンドを出力-----#
33 echo "${PROF} ${MPIEXEC} ${NRNIV} ${NRNOPT} ${HOC_NAME}"
34 #----- プログラムを実行-----#
35 time ${PROF} ${MPIEXEC} ${NRNIV} ${NRNOPT} ${HOC_NAME}
```

Listing 10: クラスタでのコマンド実行例

```
$ qsub job.sh
20252.cluster.localdomain

$ qstat
>> qstat
Every 1.0s: qstat Wed Jan 10 01:06:06 2018

Job ID Name User Time Use S Queue
-----
20251.cluster job.sh inoue 00:05:38 C cluster
20252.cluster job.sh inoue 0 R cluster
```

表 7: クラスタでのジョブの状態

ジョブのステータス	説明
Q	ジョブキューで待機中.
R	ジョブを実行している.
C	ジョブが完了した.

3 最適化の手法

本研究では、モデルに依存するパラメータと実行マシンに依存するパラメータ、そしてプログラムのコンパイル時に関わるパラメータ（コンパイルオプション）を調節することでシミュレーション系の最適化を目指した。

以下にそれぞれのパラメータの詳細を示す。

3.1 モデルに依存するパラメータ

以下に Hodgkin-Huxley 方程式のモデルを例としてそれぞれのパラメータを示す。
モデルに依存するパラメータに関しては先行研究 (TODO: add reference) において SIMD 化、配列構造の最適化により計算速度が大きく向上することが示されているため、その二つに加え配列構造の順序を入れ替えることによってキャッシュヒット率の向上に取り組んだ。
Hodgkin-Huxley 方程式は、NEURON 内において MOD 形式で次のように記述されている。

Listing 11: hh.mod

```
1 TITLE hh_k.mod squid sodium, potassium, and leak channels
2
3 UNITS {
4   (mA) = (milliamp)
5   (mV) = (millivolt)
6   (S) = (siemens)
7 }
8
9 ? interface
10 NEURON {
11   SUFFIX hh_k
12   USEION na READ ena WRITE ina
13   USEION k READ ek WRITE ik
14   NONSPECIFIC_CURRENT il
15   RANGE gnabar, gkbar, gl, el, gna, gk
16   GLOBAL minf, hinf, ninf, mtau, htau, ntau
17   THREADSAFE : assigned GLOBALs will be per thread
18 }
19
20 PARAMETER {
21   gnabar = .12 (S/cm2) <0,1e9>
22   gkbar = .036 (S/cm2) <0,1e9>
23   gl = .0003 (S/cm2) <0,1e9>
24   el = -54.3 (mV)
25 }
26
27 STATE {
28   m h n
29 }
30
31 ASSIGNED {
32   v (mV)
33   celsius (degC)
34   ena (mV)
35   ek (mV)
36
37   gna (S/cm2)
38   gk (S/cm2)
39   ina (mA/cm2)
```

```

40 ik (mA/cm2)
41 il (mA/cm2)
42 minf
43 hinf
44 ninf
45 mtau (ms)
46 htau (ms)
47 ntau (ms)
48 }
49
50 ? currents
51 BREAKPOINT {
52   SOLVE states METHOD cnexp
53   gna = gnabar * m * m * m * h
54   ina = gna * (v - ena)
55   gk = gkbar * n * n * n * n
56   ik = gk * (v - ek)
57   il = gl * (v - el)
58 }
59
60
61 INITIAL {
62   rates(v)
63   m = minf
64   h = hinf
65   n = ninf
66 }
67
68 ? states
69 DERIVATIVE states {
70   rates(v)
71   m' = (minf - m) / mtau
72   h' = (hinf - h) / htau
73   n' = (ninf - n) / ntau
74 }
75
76 ? rates
77 PROCEDURE rates(v (mV)) {
78   LOCAL alpha, beta, sum, q10
79   TABLE minf, mtau, hinf, htau, ninf, ntau DEPEND celsius FROM -100 TO 100 WITH 200
80
81 UNITSOFF
82   q10 = 3^((celsius - 6.3) / 10)
83   alpha = .1 * vtrap(-(v + 40), 10)
84   beta = 4 * exp(-(v + 65) / 18)
85   sum = alpha + beta
86   mtau = 1 / (q10 * sum)
87   minf = alpha / sum
88   alpha = .07 * exp(-(v + 65) / 20)
89   beta = 1 / (exp(-(v + 35) / 10) + 1)
90   sum = alpha + beta
91   htau = 1 / (q10 * sum)
92   hinf = alpha / sum
93   alpha = .01 * vtrap(-(v + 55), 10)
94   beta = .125 * exp(-(v + 65) / 80)
95   sum = alpha + beta
96   ntau = 1 / (q10 * sum)
97   ninf = alpha / sum
98 }
99
100 FUNCTION vtrap(x, y) {
101   if (fabs(x / y) < 1e-6) {
102     vtrap = y * (1 - x / y / 2)

```

```

103 } else {
104     vtrap = x / (exp(x / y) - 1)
105 }
106 }
107
108 UNITSON

```

先行研究の中でも示されている通り、この中でプロファイル結果から多くの計算時間が必要とするのは DERIVATIVE (TODO: reference) であり以下のパラメータの多くは DERIVATIVE の計算を行う上でキャッシュヒット率をあげることを目的としている。

3.1.1 SIMD 化

SIMD とは Single Instruction Multiple Data の略のことであり、その名の通り一つの命令を複数のデータに対して同時に適用する命令のことを指す。

C 言語において SIMD 命令は明示的に SIMD の利用を定義する方法 (TODO: add desc) と、コンパイラによる自動的な SIMD 化があるが、本研究では後者のコンパイラによる SIMD 化を促進する手法のみを扱う。

$c = a + b$ という式を SIMD 命令を用いて実行すると図 6 のようになるため、コンパイラが自動的に SIMD 化を行うためには演算対処のデータ構造がベクトル化されていることが必要となる。

本研究で利用する計算機には双方ともこの SIMD 命令を実行できる FMA (Fused Multiply Add) が搭載されており、この FMA では和の計算だけでなく積和演算を行うことができる。

京についての計算性能の理論値は、

- ・1 ノード = 8CPU コア
- ・クロック = 2GHz
- ・SIMD = $2 \times$ FMA / core

となるため、SIMD を使わない場合は

$$\cdot 8 \times 2 \text{ GFLOPS} = 1 \text{ Floating-point Operation} \times 2 \text{ GHz} \times 8 \text{ コア}$$

SIMD を使う場合は

$$\cdot 8 \times 16 \text{ GFLOPS} = 4 \text{ Floating-point Operation} \times 2 \text{ SIMD Unit} \times 2 \text{ GHz} \times 8 \text{ コア}$$

となり計算性能に大きな差が出る。

同様にクラスタにおいての計算性能は、

$$(TODO 値をちゃんとする) \cdot 1 \text{ ノード} = 14 \text{ CPU コア}$$

- ・クロック = 2.4GHz
- ・SIMD = $2 \times$ FMA / core

となるため、SIMD を使わない場合は

$$\cdot 14 \times 2.4 \text{ GFLOPS} = 1 \text{ Floating-point Operation} \times 2.4 \text{ GHz} \times 14 \text{ コア}$$

SIMD を使う場合は

$$\cdot 14 \times 19.2 \text{ GFLOPS} = 4 \text{ Floating-point Operation} \times 2 \text{ SIMD Unit} \times 2.4 \text{ GHz} \times 14 \text{ コア}$$

float

double

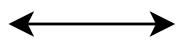
a0	a1	a2	a3
----	----	----	----

*

b0	b1	b2	b3
----	----	----	----

=

c0	c1	c2	c3
----	----	----	----



32bit



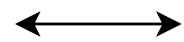
a0	a1
----	----

*

b0	b1
----	----

=

c0	c1
----	----



64bit



図 6: SIMD 命令

となり計算性能に大きな差が出る。

以上から、SIMD 命令を用いることができる環境においては、コンパイラによる SIMD 化を促進することで大きな計算性能の向上が期待できるため最適化の手法としてデータ構造の配列化は大きな意義を持つと考えられる。

float

double

a0	a1	a2	a3
----	----	----	----

*

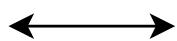
b0	b1	b2	b3
----	----	----	----

+

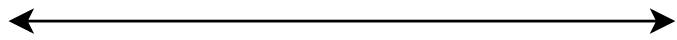
c0	c1	c2	c3
----	----	----	----

=

d0	d1	d2	d3
----	----	----	----



32bit



128bit

a0	a1
----	----

*

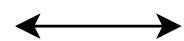
b0	b1
----	----

+

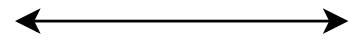
c0	c1
----	----

=

d0	d1
----	----



64bit



128bit

図 7: FMA

3.1.2 配列構造

配列を複数定義し、一つの計算の中で呼び出す場合空間的局所性が低くなり、キャッシュミスを多く生じさせる可能性がある。そのため、同時に利用する配列達を一つの配列としてくくりだすことで

空間的局所性を高くし、高速化を図る手法が考えられる。

```
int a[100], b[100], c[100], d[100];
for i=0 to 100 {
    d[i] = a[i] + b[i] + c[i];
}
```

というプログラムを例とすると、

```
int abcd[100][4];
for i=0 to 100 {
    abcd[i][3] = abcd[i][0] + abcd[i][1] + abcd[i][2];
}
```

することで連続した領域にアクセスさせることができるようにになり、キャッシュ効率の向上が見込まれる。

一方で、SIMD 化の観点では a, b, c, d がそれぞれ独立した形の配列でなくなってしまい、SIMD 命令を使いにくくなる可能性が非常に高い。

以上から空間局所性と SIMD 化という最適化を行う上で大変重要な要素どちらか一方だけを考えるのではなく、適切なハイブリッド構造を用いることを目指した。

演算に利用する変数の数が多い時は SIMD 化は難しい。これは SIMD 演算器のビット数に依存する問題だが倍精度の演算をする場合 1-4 変数（double 型 64bits に対し SIMD 演算器は 64-256bits が一般的）の演算を並列処理できるが、変数の数がより多い場合は同時に実行することが不可能であるからである。

この場合、配列をくくり出すことによって空間局所性を高くする方がより計算処理の高速化を実現できると考えられる。

3.1.2.1 Union-Find 木の利用

配列をくくり出す際、すべての配列をまとめてしまうと SIMD 化できる部分を見逃してしまったり、要素数が多すぎるためキャッシュラインに入りきらなくなるといった問題が発生する。そのためくくり出す配列は計算上関連のあるもの同士にするべきであり、関連のある変数をグループ化するために Union-Find 木を利用した。

MOD ファイルに定義された一つの式の変数を相互に関連する変数であると見なし Union-Find 木に追加していくことで、仮に次のような計算式が MOD ファイルに存在していた場合

$$a = b * c \quad (19)$$

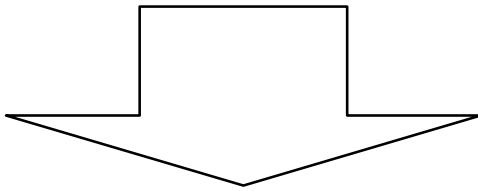
$$d = e * f \quad (20)$$

$$g = a + h \quad (21)$$

(a, b, c, g, h) と (d, e, f) という変数のグループを作成することができる。

3.1.2.2 配列のくくり出し実装

a0	a1	a2	a3	...
b0	b1	b2	b3	...
c0	c1	c2	c3	...
d0	d1	d2	d3	...



a0	b0	c0	d0	a1	b1	c1	...
----	----	----	----	----	----	----	-----

図 8: 配列のくくりだし

以上を踏まえ,SIMD 化と配列のくくりだしのハイブリッドを実現するために以下のアルゴリズムを実装した.

Listing 12: SIMD 化と配列のくくりだしのアルゴリズム 疑似コード

```

1 optimize_array(array_list, statements) {
2     union_find uf
3     related_array_list = []
4     for statement in statements {
5         tokens = parse_statement(statement)
6         for token in tokens {
7             uf.union(tokens[0], token)
8         }
9     }
10    for array_name in array_list {
11        root = uf.find(array_name)

```

```

12     related_array_list[root].append(array_name)
13 }
14 foreach array_list from related_array_list use it or not {
15     print_array_definition(modified_related_array_list)
16 }
17 }
18
19 print_array_definition(related_array_list) {
20     code = ""
21     cnt = 0
22     for array_list in related_array_list {
23         # 配列構造を定義する
24         # 関連する配列の数が一つ以上であればくくり出しを行う
25         if array_list.size() > 1 {
26             code += "static double opt_table{0}[SIZE] [{1}];\n".format(cnt, array_list.size())
27         } else {
28             code += "static double opt_table{0}[SIZE];\n".format(cnt)
29         }
30         # 配列のくくり出しを行っている場合でも画一的にアクセスできるようにマクロを定義する
31         array_cnt = 0
32         for array_name in array_list {
33             if array_list.size() > 1 {
34                 code += "#define table_{0}(x) opt_table{1}[(x)][{2}]\n"\
35                         .format(array_name, cnt, array_cnt)
36                 array_cnt += 1
37             } else {
38                 code += "#define table_{0}(x) opt_table{1}[(x)]\n".format(array_name, cnt)
39             }
40         }
41     }
42     print(code)
43 }

```

こうして Union-Find 木で作成されたグループは、空間局所性または SIMD 化のどちらかがより有効に働くことから、それぞれのグループを配列としてくくり出すか否かを独立に試行することで空間局所性と SIMD 化のハイブリッドを実現することができると考える。

3.2 実行マシンに依存するパラメータ

近年の CPU はシングルコアではなく、マルチコアによって計算を並列化することで全体としての計算能力を向上させている。

この並列化を行う上で、本研究では主に OpenMP と MPI を用いたハイブリッド並列に取り組んだ。またハイブリッド並列を行う上で、OpenMP のスレッド数、MPI のプロセス数そして各スレッドに割り振られるバッファサイズをパラメータとして用いた。

3.2.1 ハイブリッド並列化

3.2.1.1 OpenMP NEURON では、POSIX Thread によるスレッド並列が実装されているが、京上では OpenMP によるスレッド並列化しかサポートされていない。宮本らによる先行研究 [1]において、OpenMP ベースのハイブリッド並列化機構が NEURON に実装されていたため本研究ではこの OpenMP が組み込まれている NEURON を用いてシミュレーションを行った。

上記の実装では OMP_NUM_THREADS という環境変数の値に応じてスレッドの生成数を変えるものであったため、ジョブスクリプトの内部で

Listing 13: OpenMP スレッド数の指定

```
export OMP_NUM_THREADS=16
```

とすることで OpenMP のスレッド数を指定することができる。

3.2.1.2 MPI MPI はほとんどの並列計算機に入っているものであり、NEURON でも特別な変更を加えることなく利用することができる。

京とクラスタで指定する方法は違うものの、双方ともにジョブスクリプトに利用したいプロセス数を記述することで MPI を使用することができる。

クラスタにおいては、

Listing 14: クラスタ MPI プロセス数の指定

```
#PBS -l ppn=8
```

京においては、

Listing 15: 京 MPI プロセス数の指定

```
#PJM --mpi "proc=8"
```

とすることで MPI のプロセス数を指定することができる。

3.2.1.3 ハイブリッド並列化 並列化をする際に OpenMP と MPI と組み合わせることを OpenMP と MPI のハイブリッド並列化という。

OpenMP と MPI はそれぞれ長所と短所を持つが、規模が小さい場合スレッド生成のコストが大きく MPI のみを利用する Flat MPI の性能がハイブリッド化するよりも優れていることが多い。

しかしながらノード数が増えていくに連れて、MPI プロセス間での通信に利用されるネットワーク通信部分がボトルネックとなっていく可能性が高くなっていく。これは単一ノードの持つネットワーク通信に使えるリソースに対し通信対象となる MPI プロセスが増えすぎることが原因となる。

そのため計算規模が大きくなるほど、同じノードの内部では OpenMP を用いて CPU コア間で共有されているメモリを用いて通信を行い、外部ノードとの通信に MPI を使うことでボトルネックとなる通信を分散させることができるハイブリッドの強みを生かすことができる。

(TODO: 図)

また京のようにスレッドバリアと呼ばれるスレッドを高速に生成する機構を持っている計算機も存在するため、OpenMP と MPI を用いる比率を実行マシンに依存するパラメータとして本研究では用い計算機にあった最適化を目指す。

3.2.2 配列サイズの変形

配列サイズすなわちバッファとして利用するメモリの量も実行マシンに依存するパラメータである。

NEURON での計算では、SIMD 化や配列のくくり出しという観点で対象とする変数をベクトル化す

る場合、各々の変数に対して最低でもコンパートメント数 × スレッド数の大きさのバッファが必要となる。

配列をスタック領域に確保しているため、ヒープ領域に確保するように確保するためにシステムコールを呼ぶ必要はない。一方で、計算機上のスタック領域の大きさは限られておりこの領域の大半をバッファとして用いてしまうと実際の計算に支障が出てしまう。

特にスレッド数に比例して必要となるバッファが大きくなるため、ハイブリッド並列を行う際に計算機特性特にメモリに関連した性能との関連が大きくなることが予想される。

以上から適切な配列サイズを対象となるシミュレーションに応じて選択することで最適化のためのパラメータとして利用することとした。

3.3 コンパイルに関するパラメータ

コンパイルオプションについては、本研究で利用した京とクラスタにおいても大きく異なったため本研究においてはモデルと実行マシンに関連するパラメータを用いた最適化に注力し、コンパイルを利用するコンパイラを変えて比較するにとどまった。

デフォルトで利用されている GCC とコンパイル時の最適化に優れている ICC（TODO: icc のレンダレンス）を比較した。

4 自動チューニングスクリプトと MOD トランスパイラの構築

本研究では環境・イオンチャンネルモデルに関わらない自動最適化を目的としているため、京・クラスタ以外のマシンを用いる場合においても環境構築、プログラムの修正・実行にかかるコストは最小限になるべきである。

そのため、自動で神経回路計算の最適化を行うソフトを開発するとともに、環境設定に関する自動で行うスクリプトを作成した。

最適化を行うソフトは図 9 に示したように、シミュレータ部分とトランスパイラ部分に分かれており、シミュレータがトランスパイラとシステム固有のキューディングシステム双方と連携することによって最適化パラメータを探索する構成になっている。

このソフトウェア自体は複雑な処理はしないため、開発のしやすさから Python と Shell Script を用いて作成した。

4.1 環境設定スクリプト

本研究で作成したソフトウェアは、内部で使用している Python のライブラリが Python3 以降を必要としているため、実行する Python のバージョンの制約を受ける。

そのため京に代表される Python3 が存在せず、管理者権限を持つことができない環境においては、管理者権限を必要とせずユーザーの権限の中で Python のプログラムを実行できる環境を構築する必要がある。

この問題を解決するため、Pyenv（TODO: Pyenv の ref）という Python 専用の仮想環境を構築するライブラリを利用し、仮想環境上でソフトウェアを実行することにした。

この方法を用いることで、実行する計算機によってソフトウェアを変更する必要はなくなったが、実行するための環境を構築する手間がかかるため、以下に示す環境構築スクリプトを作成した。

また Makefile を作成し、make コマンドから環境構築のスクリプトの呼び出しやシミュレーションの

実行を行うようにした。

Listing 16: 利用する make コマンド

```
# 環境構築を行うコマンド
# scripts/setup_env_and_install_libraries.sh を実行する
make install

# scripts/pull_required_projects.sh を実行する
make pull

# make install, make の順で実行する pull
make setup

# シミュレーションを行うコマンド
# Makefile と同じディレクトリにあるjob.json を読み込みシミュレーションを実行する
make run

# ファイル名で指定したMOD ファイルをC 言語のファイルに変換する
make compile "ファイル名"
```

Listing 17: 環境構築スクリプトのフォルダ構成

```
Makefile
scripts/
|--- setup_env_and_install_libraries.sh
|--- pull_required_projects.sh
```

Listing 18: 必要なライブラリ

Python

- pandas: シミュレータでパラメータやそれぞれのパラメータに対応する結果を保持するデータ管理ライブラリ(TODO :ちゃんと説明)
- textX: トランスペイラでMOD ファイルから抽象木を生成する際に利用するメタ言語を作成するためのライブラリ
- jinja2: トランスペイラでC 言語のファイルを生成する際に利用するテンプレートライブラリ

NEURON

- neuron_kplus: 宮本らの先行研究によって作成された本体とそのインストールスクリプト等を含んだソース群 NEURON

Listing 19: setup_env_and_install_libraries.sh

```
#!/bin/sh
1 setupIfNecessary() {
2     # .pyenv が存在しない場合はダウンロード
3     # install 時に初回のみ実行される
4     if [ ! -d "~/.pyenv" ]; then
5         # github からクローンする
6         git clone https://github.com/pyenv/pyenv.git ~/.pyenv
7
8
9     # 必要な環境変数の設定
10    echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.bash_profile
11    echo 'export PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.bash_profile
12    echo -e 'if command -v pyenv 1>/dev/null 2>&1; then\n        eval "$(pyenv init -)"\n    fi' >> ~/.bash_profile
13    exec "$SHELL"
14
15    # anaconda3-4.3.0 を Python として利用するための設定
16    pyenv install anaconda3-4.3.0
```

```

17     pyenv local anaconda3-4.3.0
18     pyenv rehash
19 fi
20
21 # genie 実行に必要な Python ライブラリのインストール
22 pip install textx
23 pip install pandas
24 pip install jinja2
25 }
26 setupIfNecessary

```

Listing 20: pull_required_projects.sh

```

1#!/bin/sh
2pullIfNotExist() {
3    # の本体を拡張したが存在しない場合はからクローンする NEURONneuron_kplusGithub
4    if [ ! -d "neuron_kplus" ]; then
5        git clone git@github.com:hashmup/neuron_k.git neuron_kplus
6        # NEURON-7.2 をビルドする際に必要となる空のディレクトリを明示的に追加する
7        mkdir -p neuron_kplus/nrn-7.2/src/npy24
8        mkdir -p neuron_kplus/nrn-7.2/src/npy25
9        mkdir -p neuron_kplus/nrn-7.2/src/npy26
10       mkdir -p neuron_kplus/nrn-7.2/src/npy27
11    else
12        # NEURON がすでに存在している場合は,NEURON を最新版に更新する
13        (cd neuron_kplus &&
14        git checkout . &&
15        git pull origin master)
16    fi
17 }
18 pullIfNotExist

```

4.2 シミュレータ

最適化の方法として、複数のパラメータからモデル、実行環境に即したパラメータを選択するという手法を選択したが、そのためには複数のパラメータでシミュレーションを行いその結果を集約するプログラムが必要となる。

本研究ではこのパラメータ選択を容易かつ高速に行うため、

1. MOD ファイルからパラメータとなりうる情報を自動で抽出し、パラメータの候補を生成する。
2. キューイングシステムに対して、複数のジョブを並行して投げ結果を非同期的に集約できる。
3. 実行結果を最適化前のデフォルトの結果と比較し、最適化を通して実行結果に変化がないかを確認する。
4. json 形式で、実行するファイルや各パラメータの範囲（プロセス数は 1 から 10 など）といったシミュレーションに関する情報を指定することができる。

という機能を持ったシミュレータの開発を行った。

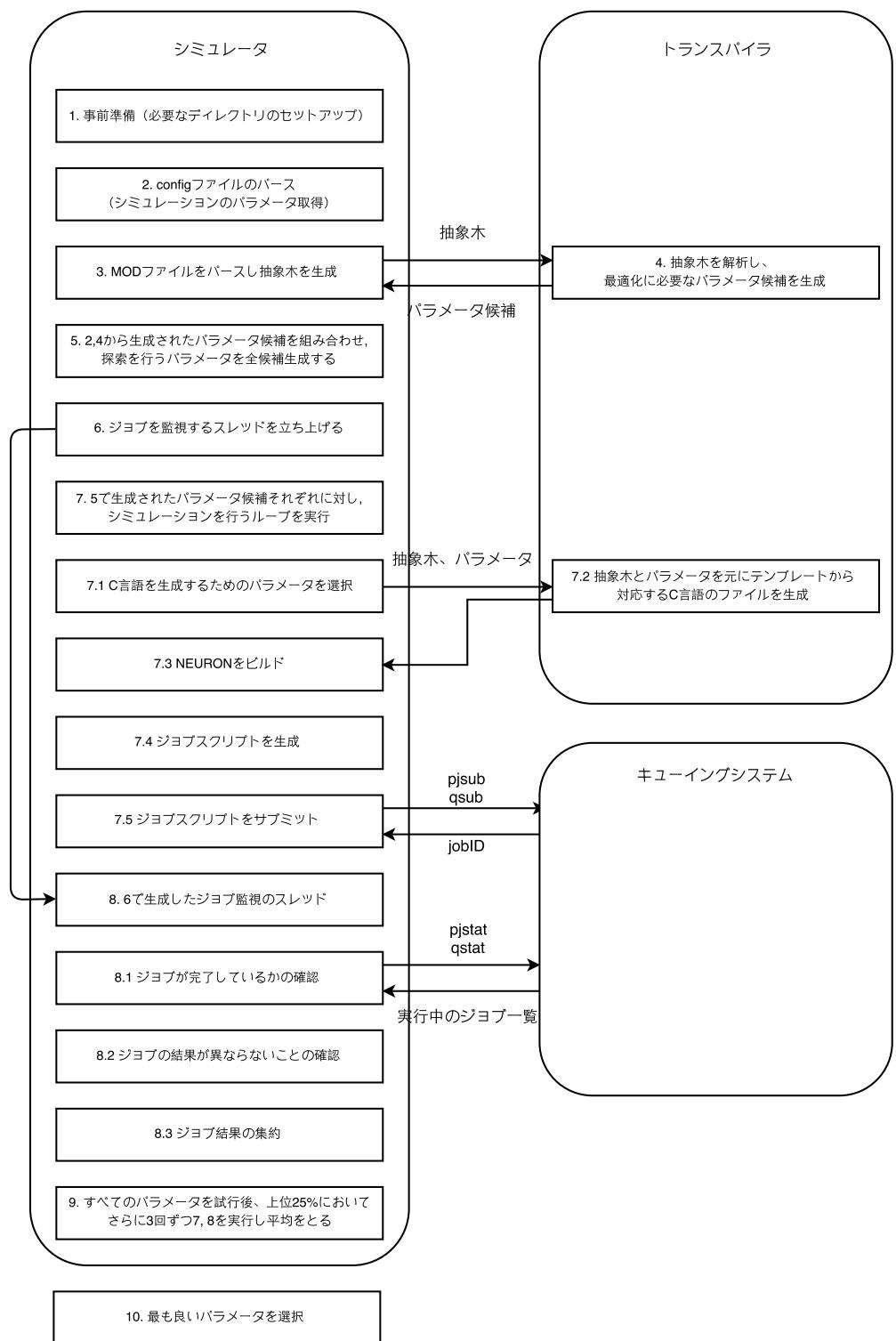


図 9: Genie の全体像

4.2.1 事前準備

シミュレータを起動した際、ジョブスクリプトやビルドスクリプトを置くためのディレクトリの作成や並列で NEURON のビルドを行う場合に実行形式に対するレファレンスが衝突しないようにするため、一時的に必要なディレクトリのコピーを作成するといったシミュレーションを行うために必要な準備をはじめに行う。

具体的には次に示すディレクトリを作成・初期化する。

Listing 21: 作成されるディレクトリ

```
genie/ : ディレクトリ top
|--- tmp/ : 実行結果、ログを保持するためのディレクトリ
|--- neuron_kplus/
    |--- exec/ : 実行形式を保持するためのディレクトリ
    |--- exec.tmp/ : 並行ビルドのためのexec のコピーディレクトリ
    |--- nrn-7.2.tmp/ : 並行ビルドのためのnrn-7.2 のコピーディレクトリ
    |--- specials.tmp/ : 並行ビルドのためのspecials のコピーディレクトリ
```

この中で、末尾に.tmp がつくディレクトリは並行してキューイングシステムにジョブをサブミットする際にレファレンスが衝突しないようにするために必要となる。

NEURON のシミュレーションはキューシステムにサブミットされたのち、順番が来るまでキューで待機状態にある。

そして実行される段階になって初めてジョブスクリプトに指定された NEURON の実行形式が参照される。

ここで実行形式を変更する必要がある際には、そこまでのジョブが完了してから再度ビルドから行うという方法を用いることもできるが、その場合ビルドをする必要がある度にその間ジョブの実行を止めることになり、シミュレーション全体として大きなボトルネックになる。

そこで、ビルドに関わる nrn-7.2, exec, specials それぞれのコピーとなる一時ディレクトリを作成し、ジョブスクリプトの内部から参照する実行形式をビルドが必要な度に切り替えるようシミュレータ内部で設定することでレファレンスの衝突を起こさずにジョブを実行中に次に使う実行形式のビルドを行うことができるようになる。

4.2.2 config ファイルのパース

シミュレーションを行う上で、どのシミュレーションファイルを用いるのか、パラメータとして何を利用するのかといったシミュレーション自体の定義が必要となる。

本研究では JSON 形式で記述された config ファイルをシミュレータに渡すことで定義されたパラメータの範囲で対象となるシミュレーションを最適化するためにパラメータの探索を行う。

次に config ファイルの例とともに、どのようにしてパラメータの範囲を定義するのかを示す。

config ファイルでは、実行形式の生成に関わるパラメータ（コンパイルオプションなど）とジョブスクリプトの生成に関わるパラメータを分けて定義している。

また、それぞれの環境に対して特有のパラメータについては build_config_マシン名や job_マシン名というように、末尾に実行マシンの名前をつけることで区別している。

Listing 22: クラスタに対する config ファイル

```
{  
    # 実行形式の生成に関わるパラメータ  
    "build": {  
        # ビルド対象のパスを定義  
        "build": {
```

```

        "neuron_path": "../nrn-7.2",
        "specials_path": "../specials"
    },
    # クラスタのコンパイルオプションを定義
    "build_config_cluster": {
        "options": [
            "--without-iv",
            "--without-x",
            "--without-nrnoc-x11",
            "--with-paranrn",
            "--with-mpi",
            "--with-multisend",
            "--enable-shared=no",
            "--enable-static=yes"
        ],
        "compile_options": {
            "linux_nrnmech": "no",
            "use_pthread": "no",
            "CFLAGS": "-O3 -fopenmp -DKPLUS -DKPLUS_GATHER_SCATTER -DKPLUS_SPAWN -
                -DCLUSTER_USE_OMP",
            "CXXFLAGS": "-O3 -fopenmp -DKPLUS -DKPLUS_GATHER_SCATTER -DKPLUS_SPAWN
                -DCLUSTER_USE_OMP",
            # 利用するコンパイラを定義する
            "CC": "mpicc, mpiicc"
        }
    }
},
# ジョブスクリプトの生成に関わるパラメータ
"job": {
    # クラスタのジョブスクリプトに関わるパラメータを定義
    "job_cluster": {
        # ノード数
        "nodes": "1",
        # プロセス数 MPI
        "ppn": "2, 28, 2",
        # ロードするモジュールがある場合は定義
        "modules": [
        ],
        # のスレッド数 OpenMP
        "omp_num_threads": "2, 16, 2",
        # 実行するコマンドを定義
        "nrniv": "../specials/x86_64/special -mpi",
        # 実行するシミュレーションを定義
        "hoc_name": "../hoc/bench_main.hoc",
        # シミュレーションのステップ数を定義
        "stop_time": 50,
        # 本体でのループの分割数を定義 NEURON
        "nthread": 16,
        # プロファイラを利用する場合はここで定義
        "prof": ""
    }
}

```

config ファイル内でパラメータの範囲を定義する際、数字の場合はカンマで区切って範囲を定義する。OpenMP のスレッド数を例にすると定義方法は 3 種類あり、それぞれ次に示すように解釈される。

Listing 23: 数値パラメータの定義

```

# のスレッド数 OpenMP
# パラメータ: "start, end"
"omp_num_threads": "2, 8"

```

```

# => [2, 3, 4, 5, 6, 7, 8]

# パラメータ: "start, end, step"
"omp_num_threads": "2, 8, 2"
# => [2, 4, 6, 8]

# パラメータ: "[parameters]"
"omp_num_threads": "[2, 4, 16]"
# => [2, 4, 16]

```

文字列の候補を複数定義する場合は、候補を [] でくくることでその中でカンマで区切られた文字列すべてを候補とすることができます。

Listing 24: 文字列パラメータの定義

```

# 利用するコンパイラを定義する
"CC": "[mpicc, mpiicc]"

```

4.2.3 MOD ファイルをパースし抽象木を生成する

MOD ファイルのパースし抽象木を生成するにあたり、Python のライブラリである textX (TODO: refrence) を利用した。

textX はパーサーとメタモデルを定義することで、独自の Domain Specific Languages を作成することができるという Python のライブラリである。その中でメタモデルについては、NEURON の MOD ファイルから NeuroML という別の神経回路シミュレーションソフトの形式に変換するためのライブラリである pynmodl(TODO: pynmodl のレファレンス)において作成されていたため、pynmodl のメタモデルを利用した。

当初 pynmodl を利用し、NeuroML という XML 形式の情報を利用し実装を行う予定だったが、pynmodl は NeuroML を開発しているチームの中で本論文を執筆時において開発中のライブラリであり MOD ファイルから抽象木を取り出すためのメタモデルのみ実装されている段階であったため、MOD ファイルから textX 形式の抽象木を生成するプログラムとして利用した。

(TODO: 例を加える)

4.2.4 パラメータ候補群の生成

4.2.2 と 4.2.3 で生成したパラメータ群がシミュレータで探索する対象となる。

パラメータは大きく、

1. Config ファイルに定義される NEURON 本体のコンパイルに関わるパラメータ（コンパイルオプション）
2. Config ファイルに定義されるジョブスクリプト生成に関わるパラメータ（MPI プロセス数や OMP スレッド数など）
3. MOD ファイルから抽出された C 言語の生成に関わるパラメータ

に分けられ、本論文執筆時においてはこれらのパラメータのすべての組み合わせを候補群として生成し最適化を図る。

また、パラメータ候補群を生成するループの順番を実行形式に関わるコンパイルオプションと C 言語の生成に関わるパラメータを外側に、ジョブスクリプトの生成に関わるパラメータを内側にすること

で、ジョブの完了を待っている間に必要がある場合は次の実行形式をビルドすることができる。

Listing 25: パラメータ候補の生成 疑似コード

```
# 未実行のシミュレーションのパラメータを保持する配列
pending_sims = []

# それぞれのパラメータに対して重のループを組み 3, 全通りのパラメータ候補群を生成
for compile_param in compile_params {
    for code_optimize_param in code_optimize_params {
        for job_param in job_params {
            pending_sims.append([compile_param, code_optimize_param, job_param])
        }
    }
}
```

4.2.5 パラメータ候補群に対してシミュレーションを実行

シミュレーションの実行は、

1. ジョブ実行スレッドを生成するループ
2. ビルドが必要となるかの判定やパラメータの記録を行うジョブのサブミットの前後処理をする関数。
3. ビルドとジョブのサブミットを行う関数。

の 3 工程で行われる。

4.2.5.1 ジョブ実行スレッド生成ループ

このループでは、未実行のシミュレーションを保持している配列が空になるまでジョブ実行のためのスレッドを逐次生成していく。

その際、連続して多数のジョブを投げることで他のシステム利用者に迷惑をかけることがないよう、事前に設定したジョブの最大同時実行数を超えないようにする必要がある。実装においては、外側のループを未実行のシミュレーションが存在する限り続け、実行中のジョブの数が最大同時実行数を超える場合は一定時間スリープを行うという形にした。

また、ジョブのサブミットを別スレッドで行うため、ジョブの実行順が変わると本来必要のないビルトが必要となる場合があることからスレッドを生成したのちにわずかな時間スリープさせることにした。

Listing 26: ジョブ実行スレッドを生成するループ 疑似コード

```
MAX_NUM_JOBS = 4
run() {
    # キューイングシステムのジョブ実行状況を監視するメソッドを別スレッドで生成
    thread.start(watch_job)
    current_sim_num = 0
    # 未実行のシミュレーションが存在する場合はループを続ける
    while pending_sims.size() != 0 {
        # 現在実行中のジョブの数が最大同時実行数を超えていた場合は待機する
```

```

    if current_sim_num > MAX_NUM_JOBS {
        sleep(15)
    }
    # 実行中のジョブの数が最大同時実行数になるまでジョブをサブミットする
    while True {
        if current_sim_num >= MAX_NUM_JOBS {
            break
        }
        # ジョブのサブミットを別スレッドで行う
        thread.start(deploy_job)
        current_sim_num += 1
        # ジョブのサブミットを別スレッドで行うためスレッドの生成を少しずらす,
        time.sleep(1)
    }
}

```

4.2.5.2 ジョブサブミットの前後処理

ジョブのサブミットは並行して生成されたスレッド上で行われる。そのためその前処理として mutex ロックをかけた上で、今回利用するパラメータの取得（pending_sims から先頭の要素を取り出す）やビルドが必要か否かを判断する際に使う最新のパラメータの更新といった処理が必要となる。後述するジョブのサブミットを行う関数を呼ぶことでジョブ ID を取得することができるが、そのジョブ ID に関連した処理が後処理となる。

一つは、ジョブの監視に利用する実行中のジョブ ID を保持したテーブルに取得したジョブ ID を追加することで、もう一つはジョブ ID と紐付けてパラメータを結果を保存するテーブルに追加することである。

ジョブ ID とパラメータを紐付けることで、そのジョブが完了した際にジョブ ID を通して実行時間とパラメータを関連づけることができるようになる。

Listing 27: ジョブサブミットの前後処理 疑似コード

```

# 実行中のジョブの状態を保持したテーブルでジョブの完了を判断するために利用する
# running_jobs[job_id] が0 の場合は, job_id がpjstat やqstat の出力に一度も現れていない状態
# 1 の場合は, job_id がpjstat やqstat の出力に一度は現れている状態
running_jobs = {}

deploy_job() {
    if pending_sims.size() > 0:
        # 未実行のシミュレーションのパラメータを一つ取り出す
        compile_param, code_optimize_param, job_param = pending_sims.pop(0)

        # 実行形式生成に関わるパラメータが直前に利用したパラメータと異なる場合は
        # 実行形式をビルドし直す必要がある
        shouldBuild = current_compile_param != compile_param or
                      current_code_optimize_param != code_optimize_param

        # サブミットするジョブに用いたパラメータをもっとも直近のものとして更新する
        current_compile_param = compile_param
        current_code_optimize_param = code_optimize_param

        # パラメータを元に実効形式、ジョブスクリプトを生成しジョブをサブミットする
        # 戻り値はとなる jobID
        job_id = deploy(shouldBuild,
                        compile_param,
                        code_optimize_param,

```

```

        job_param)

# job_id を完了判定のためのテーブルに初期状態として記録する
running_jobs[job_id] = 0

# 今回のシミュレーションで利用したパラメータと結果を保持するテーブルに保存するためま
# とめる jobID
merge_params = compile_param +
    code_optimize_param +
    job_param +
    {"job_id": job_id, "time": 0}
# パラメータと実行結果を保存するテーブルにパラメータを追加
# ジョブが完了した段階で元に更新する jobIDtime
result_table.add(merge_params)
}

```

4.2.5.3 ジョブのサブミット

ジョブのサブミットを行う関数では、実行形式の生成とジョブスクリプトの生成そしてキューイングシステムに対してジョブをサブミットするという3つの役割を持つ。

まずははじめに、実行形式の生成を行う必要があるのは実行形式の生成に関するパラメータが現在利用されている実行形式のものと異なる場合であるが、それは compile_param と code_optimize_param を比較してやればよく、その結果が shouldBuild に入っているためこの変数を利用することで判定できる。

実行形式を改めて生成しなおす必要がある場合、すでにサブミットされたジョブスクリプトから参照されている可能性のある実行形式を変更するとシミュレーション結果が異なる可能性が高いのでディレクトリを切り替えることで参照の衝突を防ぐ必要がある。ここでは、use_tmp と言う変数を用いて現在の実行形式は一時ディレクトリに存在するものかオリジナルのディレクトリに存在するものかの判別を行っており、仮に use_tmp の値が真である場合は元のディレクトリ名の末尾に .tmp がついたディレクトリを利用することになる。

また、ジョブスクリプトを生成するにあたり、そのファイル名を各ジョブごとに一意にする必要がある。

これは実行形式の場合と同様にキューイングシステムで順番が回ってきた時にジョブスクリプトが参照されるため、サブミットされた後に順番が回ってくる前にジョブスクリプトが更新されると本来関係のないパラメータを用いてシミュレーションすることになるからである。

本研究ではジョブの数を保持しておき、ジョブスクリプトを job + ”何番目のジョブか” + .sh という形 (job1.sh, job2.sh, ...) で生成することで参照の衝突を防いだ。

実行形式のビルド、ジョブのサブミットに関してはコマンドが決まっているため Shell Script をあらかじめ用意しておき、その Shell Script に引数として一時ディレクトリを使用するか否か、何番目のジョブかといった情報を与えることで行った。

Listing 28: パラメータ候補群に対するシミュレーション3 疑似コード

```

deploy(shouldBuild, compile_param, code_optimize_param, job_param) {
    # 一時ディレクトリを利用するかという情報を退避する
    _use_tmp = use_tmp

```

```

# 実行形式を再度生成する必要がある場合
if shouldBuild:
    # 現在利用しているディレクトリは使えないため、ディレクトリのフラグを切り替える
    use_tmp = not use_tmp

    # 始めの行で退避していた情報を更新する
    _use_tmp = use_tmp

    # 言語のファイルをファイルの情報を元に生成する CMOD
    transpiler.gen(code_optimize_param)

    # コンパイルに関わるパラメータと実行形式を生成するディレクトリの情報を用いて
    # の実行形式を生成する NEURON
    build(compile_param, _use_tmp)
else:
    # ジョブスクリプトへのリファレンスの衝突を防ぐため何番目のジョブかという情報を保持する,
    job_cnt += 1

    # ジョブスクリプトに関わるパラメータとどのディレクトリの実行形式を利用するのかと言う
    # 情報を元に,
    # ジョブスクリプトを "job" + job_cnt + ".sh" と言うフォーマットで生成し
    # ("job1.sh, job2.sh, ..."),
    # キューイングシステムにサブミットする
    job_id = submit(job_params,
                    job_cnt,
                    _use_tmp)
return job_id
}

```

ビルドスクリプトに対して与える引数は次の 3 つになる。

1. 実行マシンに依存する命令セット名（クラスタでは x86_64, 京では sparc64）
2. 最適化した C 言語のファイルを使うか否か（Python から呼び出され、パラメータは boolean なので文字列比較を行っている）
3. 実行形式を生成するパス（一時ディレクトリを用いるかオリジナルのディレクトリを用いるかを決定するため、".tmp" または "" が与えられる）

Listing 29: 実行形式のビルドスクリプト

```

#!/bin/bash -x

ARCH=$1

rm -r ${ARCH}
../exec$2/${ARCH}/bin/nrnivmodl ../mod
if [ $# -eq 1 ]
then
    echo "optimized"
    rm ./${ARCH}/hh_k.c
    cp ~/genie/genie/transpiler/tmp/hh_k.c ${ARCH}/hh_k.c
else
    if [ $2 == 'True' ]
        echo "optimized"
        rm ./${ARCH}/hh_k.c
        cp ~/genie/genie/transpiler/tmp/hh_k.c ${ARCH}/hh_k.c
    then

```

```

        echo "default"
else
fi
fi
./exec$2/${ARCH}/bin/nrnivmodl .. /mod

```

ジョブのサブミットに際しては、ジョブの番号を与えることで一意にジョブスクリプトを指定することができるため、ジョブの番号を引数として与える。

Listing 30: ジョブのサブミットスクリプト

```

#!/bin/bash -x
qsub ../../genie/simulator/tmp/job$1.sh

```

4.2.6 ジョブ実行の監視

ジョブ実行の監視は、

1. ジョブ実行の監視とジョブ実行の後処理（パラメータと実行時間の関連付け）を行う関数
2. 実行中のジョブの完了判定
3. ジョブの結果を集約する関数
4. ジョブの結果が一致していることを確認する関数.

の 4 つから構成される。

4.2.6.1 ジョブ実行の監視と後処理

ジョブの実行スレッドを生成するループの頭で実行中のジョブの監視スレッドは立ち上げられ、以降ジョブの実行とは関係なく定期的に繰り返し実行される。

一度の実行では、キューイングシステムに対して qstat や pjstat を利用して実行中のジョブの情報を取得し、現在実行中のジョブ ID それぞれを比較する。ここでもしジョブが完了している場合は、ジョブ結果の集約を行い、ジョブ ID を元に結果をテーブルに書き込みそして実行中のテーブルからジョブ ID を削除する。

ジョブ完了時にすでに実行中のジョブの数が最大同時実行数に達していた場合は、ここで current_job_num の値が 1 つ減ることで次のループで新たにジョブをサブミットできるようになる。

また、未実行のシミュレーションがなくなり、現在実行中のジョブがすべて完了した段階でパラメータ候補すべてに対してシミュレーションをし終えたといえるためジョブ結果が最適化を通して変わっていないかの確認とパラメータと実行時間を保持したテーブルの CSV 形式での書き出しを行いシミュレータを終了する。

Listing 31: ジョブ実行の監視と後処理 疑似コード

```

watch_job() {
    # 実行中のジョブすべてに対して、ジョブが完了しているかの確認を行う
    for job_id in running_jobs {
        # ジョブが完了している場合、ジョブ結果の集約を行う
        if not is_job_still_running(job_id) {

```

```

# に紐付いたジョブ結果の集約（実行にかかった時間の取得）jobID
time = summary(job_id)

# 結果を保持するテーブルのに紐付いた実行時間を更新する jobID
result_table['job_id'] == job_id['time'] = time

# 現在実行中のジョブからを削除する jobID
running_jobs.remove(job_id)
current_job_num -= 1
}

}

# 未実行のシミュレーションと現在実行中のジョブが一つもない時,
# すべてのシミュレーションが終わったと見なせるので、実行結果が異ならないかの確認を行う
if len(pending_sims) == 0 and len(running_jobs) == 0 {
    # 実効結果が最適化によって異なるかの確認を行う
    if verify() {
        print("All results are the same.")
    } else {
        print("Some results are different.")
    }
    # 実効結果とパラメータを保持したテーブルを形式で保存する CSV
    result_table.to_csv("result.csv")
    return
}
# シミュレーションがすべて終わっていない時はジョブの監視スレッドを再度立ち上げ直す,
sleep(5)
thread.start(watch_job)

```

4.2.6.2 ジョブの完了判定

ジョブの完了判定を行う際には、キューイングシステムの上でのジョブの実行状況を取得する pjstat と qstat の出力結果を利用する。

これらの出力結果が常に一定のフォーマットに従うことは（ TODO: 章番号のレファレンス ）で示したが、一定の出力を持つため正規表現を用いることでジョブ ID とその実行状況を取得可能である。ここで注意することは、ジョブの監視スレッドの実行が一定間隔であり、必ずしも qstat や pjstat 実行時に C や STO といったジョブが完了した状態であるという出力が得られるとは限らないことである。

一方で、ジョブがキューイングシステム内で待機状態や実行状態である時間はジョブの監視スレッドの実行間隔より十分に大きいため、キューイングシステム内に特定のジョブ ID に紐づくジョブが存在しているという記録を残すことは容易である。

以上からジョブの完了の判定は、ジョブが完了状態であるという情報をキューイングシステムから得ることができればその情報を利用し、取得できなかった場合はキューイングシステム上にジョブが存在しなくなった段階でそのジョブは完了しているとみなすことができる。

Listing 32: ジョブ完了判定 疑似コード

```

# の出力に対してジョブとジョブの実行状況を取得するための正規表現 qstatID
job_cluster_exp = re.compile(
    "(?P<id>\d+).\\w+\\s+\\w+.\\w+\\s+\\w+\\s+\\d+:\\d+:\\d+\\s+(?P<state>\\w+)\\s+\\w+\\s+")

# の出力に対してジョブとジョブの実行状況を取得するための正規表現 pjstatID
job_k_exp = re.compile(
    "(?P<id>\\d+)\\s+\\w+.\\w+\\s+\\w+\\s+(?P<state>\\w+)\\s+[\s\\w\\d\\[\\]\\/:\\-]+")

```

```

is_job_still_running(job_id) {
    # 京とクラスタでは用いるコマンドが違うため分岐する必要がある（との違い）qstatpjstat
    if environment == "cluster" {
        # コマンドを実行し実行中のジョブの状態を取得する qstat
        res = execute("qstat")

        # 改行 (\n) で連結された一つの文字列となっているため n, 行ごとに分離する
        job_lines = res.split('\n')

        # コマンドの結果の行に対してそれぞれ正規表現と一致するか確認をする
        # ジョブが存在しない場合は ID, まだキューイングシステムにサブミットされていないか, すでに完了してキューイングシステムからの出力に表示されていないというつのパターンが考えられるため 2, にジョブの状態を保持することで判断する running_jobs
        for line in job_lines {
            m = job_cluster_exp.match(line)

            # 一致する場合, ジョブの実行状況を記録する
            if m is not None {
                state = m.group("state")
                # ジョブがその行と一致し ID, かつジョブの状態が () であるならば CComplete, ジョブは完了したと見なせる
                if job_id == m.group("id") {
                    if state == "C" {
                        return False
                    } else {
                        # ジョブがまだ完了していないため, ジョブに紐付いたジョブがキューイングシステムにサブミットされた状態であると記録する ID
                        if running_jobs[job_id] == 0 {
                            running_jobs[job_id] = 1
                            return True
                        }
                    }
                }
            }
        }

        # ここまでで関数の実行が終わっていないのはの出力にジョブが含まれていないということである qstatID. その上で, 一度でもキューイングシステムの出力に現れているのであれば ジョブは完了しているとみなしそうでないならばまだサブミットされていないとみなす,
        if running_jobs[job_id] > 0 {
            return False
        } else {
            return True
        }
    } else if environment == "k" {
        res = execute("pjstat")
        job_lines = res.split('\n')
        for line in job_lines {
            m = job_k_exp.match(line)
            if m is not None {
                state = m.group("state")
                if job_id == m.group("id") {
                    if running_jobs[job_id] == 0 {
                        running_jobs[job_id] = 1
                        return True
                    }
                }
            }
        }

        if running_jobs[job_id] > 0 {
            return False
        } else {
            return True
        }
    }
}

```

```
    }
```

4.2.6.3 ジョブ結果の集約

ジョブ結果は job + ジョブの順番 + .sh.o + ジョブ ID という形 (job1.sh.o10000, job2.sh.o10001) で出力される。また、結果として出力される内容の中で実行時間を出力している行は複数のプロセスやスレッドで並行的に計算をしているため順番は前後するものの同一のフォーマットに従うため、すべての行の中からこのフォーマットに合致する行を探することで実行時間を取り出すことができる。

Listing 33: ジョブ結果の集約 疑似コード

```
# シミュレーション結果のファイルの中で実行時間を取得するための正規表現
time_exp = re.compile(
    "\s+/* core time : (?P<decimal>\d+).(?P<float>\d+) sec\s+")

summary(job_id, job_cnt) {
    # ジョブとジョブの順番を元に ID, ジョブ結果のファイル名を作り実行時間を取得する
    core_time = obtain_time("job{0}.sh.o{1}".format(job_cnt, job_id))
    return core_time
}

obtain_time(filename) {
    # 与えられたファイル名のファイルに対するアクセスを作る
    f_check = Path("{0}".format(filename))

    # 与えられたファイル名のファイルがまだ存在していない場合待機をする
    while not f_check.exists() {
        time.sleep(5)
    }
    f = open("{0}{1}".format(dir_path, filename))
    lines = f.readlines()
    f.close()

    # ファイルの行全てに対して一行ずつ正規表現と一致する行があるか確認する
    for line in lines {
        m = time_exp.match(line)
        if m {
            # 正規表現で取得した整数部と小数部を実行時間に変換する
            calc_time = int(m.group("decimal")) + \
                        int(m.group("float")) * 10**(-len(m.group("float")))+1
            return calc_time
        }
    }
}
```

4.2.6.4 ジョブ結果の比較

最後に、実際の実行結果が最適化を通して変化していないことの確認も必要である。これは実行結果のファイルを見ることで判断できるが、複数プロセス・スレッドを用いた場合途中の出力結果の順番がランダムになっているという問題があった。ジョブの実行結果は次に示される 3 つの関数で出力される。

Listing 34: ジョブ実行結果出力箇所

```
1 proc print_stat() {
2     for i = 0, cells.count-1 {
3         printf("[proc:%02d] synlist %d\n", pc.id, cells.object(i).synlist.count())
```

```

4 | }
5 |
6 |
7 proc spikeout() {
8   local i, count, rank
9   localobj fobj, tmpmt
10  if(pc.id == 0) {
11    printf("\n\ttime [ms]\t cell_id\n")
12  }
13  pc.barrier()
14  for i=0, tvec.size()-1 {
15    printf("SPIKE : \t %f\t %d\n", tvec.x[i], idvec.x[i])
16  }
17 }
18
19 proc printSpikeStat() {
20   local nsendmax, nsend, nrecv, nrecv_useful
21   nsendmax = pc.spike_statistics(&nsend, &nrecv, &nrecv_useful)
22   printf("[%d] nsendmax=%d nsend=%d nrecv=%d nrecv_useful=%d\n", pc.id, nsendmax, nsend,
23         nrecv, nrecv_useful)
24 }

```

また、その実行結果を一部抜粋した元が次になる。

Listing 35: ジョブ実行結果一部抜粋

```

1 [5] NC = 184, SYN = 94, tmp_pre = 90, tmp_post = 85
2 [7] NC = 198, SYN = 108, tmp_pre = 90, tmp_post = 99
3 [8] NC = 196, SYN = 106, tmp_pre = 90, tmp_post = 97
4 [3] NC = 191, SYN = 101, tmp_pre = 90, tmp_post = 92
5 SPIKE : 5.025000 61
6 SPIKE : 5.025000 54
7 SPIKE : 5.350000 55
8 SPIKE : 5.350000 57
9 SPIKE : 105.350000 62
10 [6] nsendmax=5 nsend=54 nrecv=1512 nrecv_useful=33
11 SPIKE : 5.025000 1
12 SPIKE : 105.025000 234
13 SPIKE : 105.350000 235
14 SPIKE : 105.350000 237
15 SPIKE : 105.350000 239
16 SPIKE : 105.350000 242
17 [26] nsendmax=5 nsend=54 nrecv=1512 nrecv_useful=18

```

仮に同じシミュレーションをシングルプロセス・シングルスレッドで行った場合、上記の結果は

Listing 36: シングルスレッドで実行する場合の実行結果

```

1 [3] NC = 191, SYN = 101, tmp_pre = 90, tmp_post = 92
2 [5] NC = 184, SYN = 94, tmp_pre = 90, tmp_post = 85
3 [7] NC = 198, SYN = 108, tmp_pre = 90, tmp_post = 99
4 [8] NC = 196, SYN = 106, tmp_pre = 90, tmp_post = 97
5 SPIKE : 5.025000 1
6 SPIKE : 5.025000 54
7 SPIKE : 5.350000 55
8 SPIKE : 5.350000 57
9 SPIKE : 5.025000 61
10 SPIKE : 105.350000 62
11 SPIKE : 105.025000 234
12 SPIKE : 105.350000 235
13 SPIKE : 105.350000 237
14 SPIKE : 105.350000 239
15 SPIKE : 105.350000 242

```

```

16 [6] nsendmax=5 nsend=54 nrecv=1512 nrecv_useful=33
17 [26] nsendmax=5 nsend=54 nrecv=1512 nrecv_useful=18

```

のように ID が昇順になる。

そのため、実行結果を比較する際には、3種類の関数 print_stat, spikeout, printSpikeStat からの出力結果をそれぞれソートした上で比較する必要がある。

本研究においては、出力の形式がそれぞれの関数で固定であり、上記の3種類の関数のみがシミュレーションの実行結果と関係しているため、ID を元にソートをかけ、ソート後の出力結果を格納した配列のハッシュ値を比較することで実行結果に変化がないことを確かめた。

Listing 37: 実行結果比較コード

```

1 # シミュレーション結果でそれぞれのスパイクに関わる行の正規表現
2 spike_exp = re.compile(
3     "SPIKE : \t (?P<val>\d+.*\d*)\t (?P<idvec>[0-9]+) \[(?P<pid>\d+)\]")
4 # シミュレーションでスパイク情報を集計した行の正規表現
5 spike_stat_exp = re.compile(
6     "\[(?P<pid>\d+)\] NC = (?P<nc>\d+), SYN = (?P<syn>\d+), \
7     tmp_pre = (?P<tmp_pre>\d+), tmp_post = (?P<tmp_post>\d+)")
8 # シミュレーション結果で各々のプロセスに関わる行の正規表現
9 end_exp = re.compile(
10    "\[(?P<pid>\d+)\] nsendmax=(?P<nsendmax>\d+) nsend=(?P<nsend>\d+) \
11    nrecv=(?P<nrecv>\d+) nrecv_useful=(?P<nrecv_useful>\d+)")
12
13
14 def verify(self, files):
15     # ハッシュ値を保存するセット
16     s = set()
17     # ファイルを一つずつ読み込みソートしたのちハッシュ値をセットに追加する,
18     for filename in files:
19         f = open(filename)
20         lines = f.readlines()
21         f.close()
22         s.add(self.sort_and_hash_log(lines))
23     # すべてのファイルの内容が同じなのであればセットの大きさはである,1
24     return len(s) == 1
25
26
27 def sort_and_hash_log(self, lines):
28     _lines = []
29     spike_stat = {}
30     end = {}
31     spike = {}
32     # の最大値を記憶しておくための変数 processID
33     maxid = 0
34     # ファイルないのすべての行に対して正規表現と合致するものを抜き出す
35     for line in lines:
36         m = spike_stat_exp.match(line)
37         if m:
38             pid = int(m.group("pid"))
39             spike_stat[pid] = line
40             maxid = max(maxid, pid)
41             continue
42         m = spike_exp.match(line)
43         # 各に関してのみ複数行あるため Spike次元で情報をストアする,2
44         if m:
45             pid = int(m.group("pid"))
46             idvec = int(m.group("idvec"))
47             if pid in spike:
48                 spike[pid][idvec] = line
49             else:
50                 spike[pid] = {}

```

```

51     spike[pid][idvec] = line
52     continue
53     m = end_exp.match(line)
54     if m:
55         pid = int(m.group("pid"))
56         end[pid] = line
57     # spike, の統計量 spikeそのプロセスの統計量を,順に並べ替える processID
58     for pid in range(maxid + 1):
59         _lines.append(spike_stat[pid])
60         for line in spike[pid]:
61             _lines.append(spike[pid][line])
62             _lines.append(end[pid])
63     # ソートし終えたファイルの中身のハッシュ値を計算する
64     return hash(tuple(_lines))

```

すべてのジョブ結果のファイルに対し、それぞれの行が各関数に該当する正規表現と一致するか調べ、一致する場合は ID を元にしてならべかえる。
並べ替えが終わった段階で hash 値を計算し、すべてのファイルに対して hash 値が共通のものであるかを確認している。

4.2.7 シミュレーションの再実行

シミュレーションを各パラメータに対して一度だけ実行する場合、並列でジョブを投入しているためメモリ利用状況や他のプロセスの影響も含めて実行時間が状況に応じてある程度変化することが予測される。

そのため、複数回試行した上でその平均実行時間がもっとも短いものを選択するという方法を取り、外部からの影響を減らすことを試みた。

しかしながらパラメータ候補群を全探索する方法で複数回のシミュレーションを行うには非常に時間がかかるため、探索範囲を一度目のシミュレーション結果を元に絞り込むことで探索範囲を狭めることができると考えた。

初回のシミュレーション時に実行時間が短かった上位 25 %を利用するという単純なアルゴリズムではあるが、

1. プロセス数 2-28.
2. スレッド数 2-16.
3. 変数の配列化か否か.
4. 配列のくくり出しをするか否か.

というパラメータ候補群でシミュレーションを行った結果図 (TODO: 図をつける) のようになったため、一定の効果があると言える。

4.3 トランスペイラ

先行研究 [1] では、モデルに依存するパラメータを調節するために、計算モデルが記述された MOD ファイルから nmodl を介して生成された C ファイルを手動で変更を加えることで最適化を図ってい

た。

本研究では、自動チューニングを目的としているため、このプロセスも自動化する必要があり、そのためにこの MOD から C へ変換するトランスパイラを作成した。

MOD をパースするにあたっては Domain-Specific Languages を作成するための Python ライブラリである, textX (TODO: ref) を利用し、また MOD の Context Free Grammar は MOD ファイルから NeuroML を生成するためのプロジェクトである pynmodl (TODO: ref) のプログラムを用いた。

4.3.1 nmodl

NEURON に付属しているトランスパイラである nmodl は、MOD ファイルを lex と yacc (TODO: lex yacc の参照) を用いてパースした情報を C 言語のテンプレートに埋め込むことで対応する C 言語のファイルを出力している。

このテンプレート化された部分の中には NEURON 本体とリンクさせるために必要な情報も多数あるため、本研究で作成するトランスパイラも nmodl の C 言語テンプレートをベースに利用した。
(TODO: もう少し説明をたす)

4.3.2 textX

MOD ファイルから抽象木を構築するために、Domain Specific Language (以下 DSL) 作成用の Python ライブラリである textX[3] を用いた。DSL とは、C 言語等に代表される汎用言語ではなく、微分方程式の記述を行う MOD ファイルのように応用先が特定の領域に限られる言語である。

一般的に言語を作成しようとした時、Lexer, Parser そして Code generation という 3 つの機能を実装する必要がある。しかしながら、言語のメタモデルを作成することで textX は内部的に Lexer と Parser の役割を果たすため、メタモデルを記述したファイル (.tx という拡張子を用いる) と、メタモデルから生成された抽象木からコードを生成するプログラムを実装することで DSL に対するコンパイラを作成することができる。

また、MOD ファイルのメタモデルは前述した pynmodl というプロジェクトで作成されたものを用了いた。

図 10 は Hodgkin-Huxley の MOD ファイルの抽象木を表しており、以下 textX から得られた抽象木から C 言語のプログラムを生成する方法について述べる。

図 10: Hodgkin-Huxley の MOD ファイルの抽象木

4.3.3 構成

(TODO: 章番号) のアルゴリズムで触れた中で, トランスパイラ内で実装を行ったのはモデルに依存するパラメータであり, NEURON 本体で細胞単位での計算の並列化等の設定もできるため, 主に逐次プログラムの最適化を主眼に置いた.

MOD ファイルを C 言語のファイルに変換する際, 変換された C 言語のファイルは,

1. nmodl 内でテンプレート化されている共通部分 (base)
2. グローバル変数や関数定義部分 (definition)
3. それぞれの関数や変数を NEURON とリンクさせる部分 (register)
4. ユーザー定義関数部分 (user)
5. NEURON 本体と関連する関数部分 (neuron)
6. ODE (微分方程式) を計算する関数部分 (ode)

の 6 つに分けることができる.

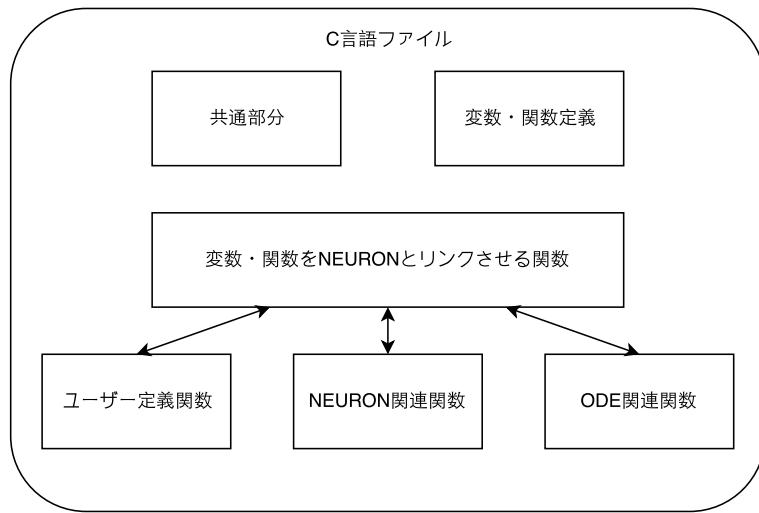


図 11: 変換された C 言語ファイルの構成

この中で計算を行うのは 4, 5, 6 の部分であるが, これらは 2 や 3 の部分を通して疎に繋がってはあるものの, 互いに独立性が非常に高い.

そのため, それぞれに対して個別に最適化を行い最終的に組み合わせて C 言語のファイルを生成する方法を取ることができる.

本研究では図 12 のように, まず textX を用いて MOD ファイルから抽象木を生成し, その抽象木を

解析することで最適化に用いるパラメータの候補を生成したのち, それらのパラメータと抽象木から個々に最適化を行ったものをテンプレートに埋め込むことで対応する C 言語のファイルを生成した.

またこのように C 言語に変換する部分と抽象木自体を解析する部分を完全に分離するだけでなく, それぞれ内部でさらに役割ごとに分割することで, 最適化の方法を追加する際にも変更しやすくなる. 仮にこれらがすべて密に繋がっていた場合, 新しい方法を追加するためには関連する箇所をすべて正確に変更する必要があり, 規模が大きくなるにつれて難しくなっていくが, 細かく担当する範囲を分割することで例えば ODE のみに関連する変更を加えたい場合は, ODE を担当するモジュールのみの変更でよくなる.

さらに, それぞれのモジュールが抽象木の情報とパラメータを直接受け取るため, モジュールごとに細かい最適化を行うことができるとともに, シミュレータからもどの最適化をどの関数に適用するとといった場合分けをしやすくなる.

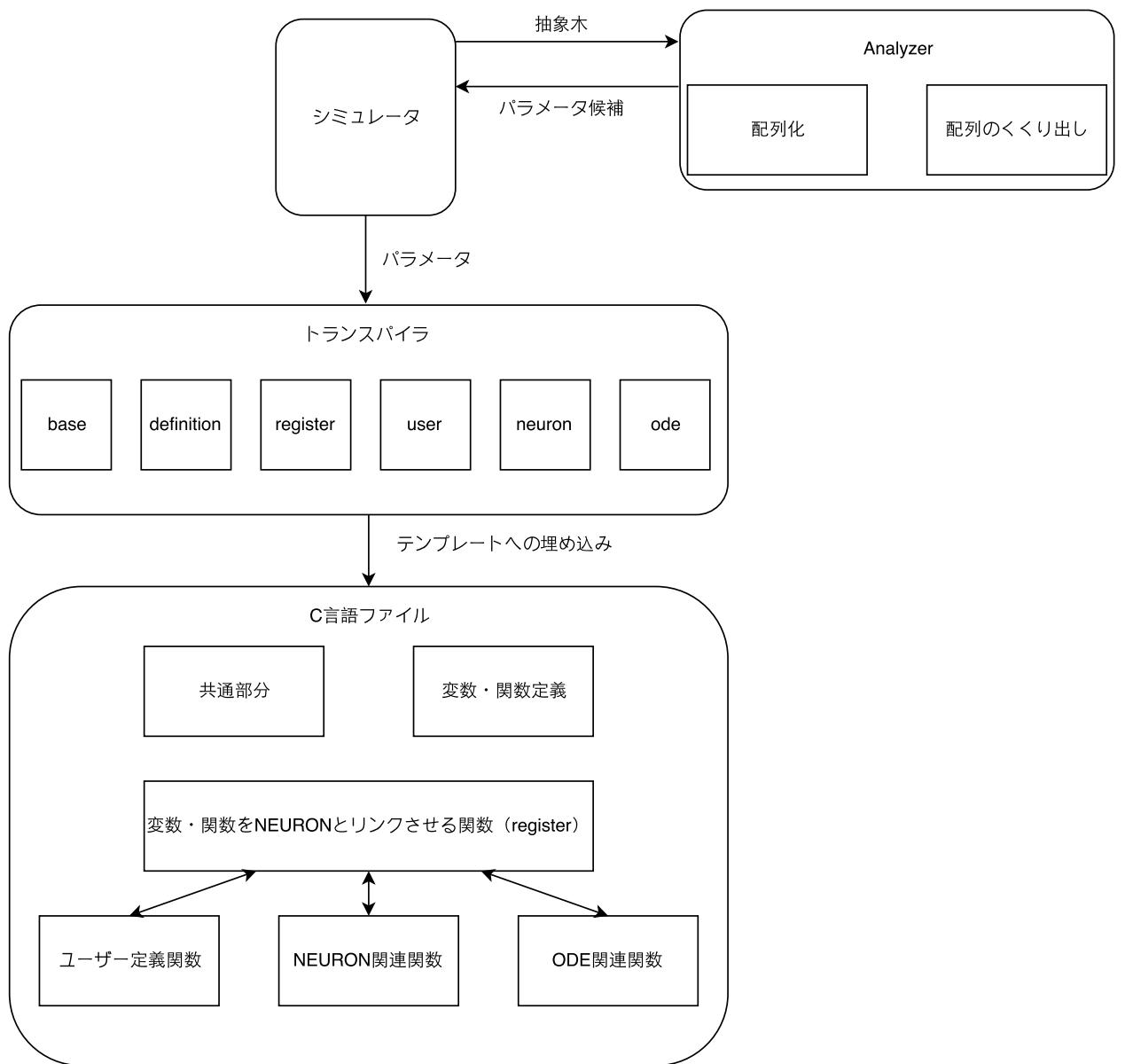


図 12: トランスパイラー 構成

4.3.3.1 変数の取り出し

SIMD 化と配列のくくり出しを行うために、MOD ファイルの中で利用されている変数を取り出す必要がある。

これは前述の hh.mod を例にすると、

Listing 38: Hodgkin-Huxley モデルの計算式

```
? interface
NEURON {
    GLOBAL minf, hinf, ninf, mtau, htau, ntau
}

? currents
BREAKPOINT {
    SOLVE states METHOD cnexp
    gna = gnabar * m * m * m * h
    ina = gna * (v - ena)
    gk = gkbar * n * n * n * n
    ik = gk * (v - ek)
    il = gl * (v - el)
}

? states
DERIVATIVE states {
    rates(v)
    m' = (minf - m) / mtau
    h' = (hinf - h) / htau
    n' = (ninf - n) / ntau
}
```

の部分から取得できることがわかる。

pynmodl でパースした抽象木では、Breakpoint, Derivative, Global という名前がつけられているためそれぞれの式は抽象木の root を root という変数で保持しているとすると、

Listing 39: 計算式の取得

```
derivative_stmts = children_of_type('Derivative', root)[0].b.stmts
breakpoint_stmts = children_of_type('Breakpoint', root)[0].b.stmts
```

として取得することができる。

また、これらの式から変数を取得するには、

Listing 40: 変数の取得

```
# 式を変数に分離する
parse_into_token(exp) {
    # 変数の区切りとなる文字の定義
    term_exp = re.compile("[\\(\\)\\+\\-\\*/\\=\\{\\}\\s]")
    start = 0
    pos = 0
    tokens = []
    # 末尾が変数名で終わる時のために空白を追加する
    exp += " "
    while pos < exp.size() {
        # 区切り文字であるかの確認
        m = term_exp.match(exp[pos])
        if m {
            # 変数名の取得
            token = exp[start:pos]
            # 空文字でなければ変数名のリストに追加
            if token.size() > 0 {
                tokens.append(token)
            }
        }
    }
}
```

```

        # 変数名の開始位置を更新
        start = pos + 1
    }
    pos += 1
}
return list(set(tokens))
}

get_symbols(path, stmts) {
    symbols = []
    for stmt in stmts {
        tokens = []
        # 式の左辺から変数名を取得
        if hasattr(stmt, 'variable') {
            if stmt.variable {
                if hasattr(stmt.variable, 'lems') {
                    exp = stmt.variable.lems
                } else {
                    exp = stmt.variable
                }
                tokens_lhs = parse_into_token(exp)
                if len(tokens_lhs) {
                    tokens.append(tokens_lhs)
                }
            }
        }
        # 式の右辺から変数名を取得
        if hasattr(stmt, 'expression') {
            if stmt.expression.lems {
                tokens_rhs = parse_into_token(stmt.expression.lems)
                if len(tokens_rhs) {
                    tokens.append(tokens_rhs)
                }
            }
        }
        # 変数名の中で重複するものがある場合は重複しているものを削除する
        if len(tokens) {
            symbols.append(list(set(tokens)))
        }
    }
    return symbols
}

```

することで取得でき、Derivative と Breakpoint それに含まれる式を引数とすることで変数名のリストを取得できる。

4.3.3.2 SIMD 化

SIMD 化をするためには、計算式の中で用いられる変数を配列化する必要がある。

ここで必要となるのは、

1. 配列を定義する
2. 計算式の変数名を配列名に変更する

の 2 点である。

これは変数を取得したのちに、SIMD 化を行う変数を選択し式の中で該当する変数を配列化すれば良い。

本研究において配列名は、_変数名_table として定義することとする。

Listing 41: 計算式内の変数の配列化

```
get_simdize_table(root, token_list) {
    code = ""
    # SIMD 化する変数のリストそれぞれに対応する配列を定義する
    for token in token_list {
        code += "static double _{0}_table[BUFFER_SIZE * MAX_NTHREADS];\n" \
            .format(token)
    }
    return code
}

simdize_exp(exp, token_list) {
    # SIMD 化を行った後の式を保持する変数
    simdized_exp = ""
    term_exp = re.compile("[\\(\\)\\+\\-\\/*\\=\\{\\}\\s]")
    start = 0
    pos = 0
    tokens = []
    exp += " "
    while pos < exp.size() {
        m = term_exp.match(exp[pos])
        # 区切り文字であるかの確認
        if m {
            # 変数名の取得
            token = exp[start:pos]
            if token {
                # もし変数名がSIMD 化する対象であるならば配列名に置き換える
                if token in token_list {
                    simdized_exp += "_{0}_table[_iml]" \
                        .format(token)
                } else {
                    simdized_exp += token
                }
            }
            # 区切り文字は元の式を構成するために必要なので変数名の後に追加する
            simdized_exp += exp[pos]
            start = pos + 1
        }
        pos += 1
    }
    # 区切り文字の空白を最後尾に追加しているので、空白を抜かして返す
    return simdized_exp[:simdized_exp.size() - 1]
```

4.3.3.3 配列構造のくくり出し

アルゴリズムの章（TODO：章番号）において、Union-Find 木を用いて MOD ファイル内の式を分類することで、配列としてくくり出せる変数をグループ化する手法について述べた。グループ化された変数をもとに配列をくくり出す際、配列に対するアクセス方法が変わってしまうため対応する箇所をすべて変更しなければならないという問題が生じた。そこで、配列へのアクセスをマクロを通して行うことで同一の方法でくくり出した配列にもアクセスできるようにした。

Listing 42: 計算式内の変数の配列化

```
get_optimize_table(token_list, merge_table_list) {
    """
    " token_list: 配列化する変数名のリスト変数[1, 変数 2,...]
    " merge_table_list: くくり出しを行う変数名のリスト変数[[1, 変数 2, ...], 変数 [3, 変数 4, ...], ...]
    """"
```

```

code = ""
if merge_table_list {
    for i = 0 to merge_table_list.size() {
        # くくり出しを行う変数のグループそれぞれの大きさ分の配列を確保する
        code += "static double opt_table{0}" \
            "[BUFFER_SIZE * MAX_NTHREADS] [{1}];\n" \
            .format(i, merge_table_list[i].size())

        # それぞれの変数に対して一意にアクセスできるようにマクロを定義する
        for j = 0 to merge_table_list[i].size() {
            code += "#define TABLE_{0}(x) " \
                "opt_table{1}[({x})][{2}]\n" \
                .format(merge_table_list[i][j].upper(), i, j)

            # くくり出しを行った配列はすでに定義済みであるため、配列化するリストから除外
            # する
            if merge_table_list[i][j] in token_list {
                token_list.remove(merge_table_list[i][j])
            }
        }
    }
}
# くくり出しが行われなかった変数に対しても同様に配列とその配列にアクセスするためのマクロ
# を定義する
for token in token_list {
    code += "static double _{0}_table[BUFFER_SIZE * MAX_NTHREADS];\n" \
        .format(token)
    code += "#define TABLE_{0}(x) _{1}_table[({x})]\n" \
        .format(token.upper(), token)
}
code += "static double _{0}_table[BUFFER_SIZE * MAX_NTHREADS];\n" \
    .format(param)
return code
}

optimize_exp(exp, token_list, merge_table_list) {
    optimized_exp = ""
    term_exp = re.compile("[(\\()\\)+\\-\\/\\/*\\=\\{\\}\\s]")
    start = 0
    pos = 0
    tokens = []
    exp += " "
    while pos < exp.size() {
        m = term_exp.match(exp[pos])
        # 区切り文字であるかの確認
        if m {
            # 変数名の取得
            token = exp[start:pos]
            if token {
                # 変数が SIMD 化またはくくり出しを行う対象であるならばマクロに置き換える
                if token in merge_table_list or token in token_list {
                    optimized_exp += "TABLE_{0}(_im1)" \
                        .format(token.upper())
                } else {
                    optimized_exp += token
                }
            }
            # 区切り文字は元の式を構成するために必要なので変数名の後に追加する
            optimized_exp += exp[pos]
            start = pos + 1
        }
        pos += 1
    }
}

```

```
# 区切り文字の空白を最後尾に追加しているので、空白を抜かして返す
return optimized_exp[:optimized_exp.size() - 1]
}
```

5 シミュレーション結果

本実験（TODO: ?）では、作成したシミュレータを用いて Hodgkin-Huxley モデルの神経細胞モデル（TODO: ref）からなる（TODO: 章番号）で示したベンチマークネットワークを最適化しシミュレーションを行い、その後シミュレーションの結果を用いて最適化に用いたパラメータを定量的に評価する。

本論文執筆時においてパラメータの探索は全探索を用いているため、パラメータすべての組み合わせを大規模なシミュレーションで行うことは現実的ではない。そのため、（TODO: アルゴリズムの説）で述べたように、実行マシンに関わるパラメータ（プロセス数とスレッド数）は並列実行に関するものであり、モデルに関わるパラメータ（SIMD 化と配列のくくり出し）は逐次実行に関するものであるという事実を利用する。

5.1 節では、シミュレーション時間をスパイクが開始する 100ms に設定したシミュレーションを 3 回行い、その平均をとった実行時間を用いてパラメータの評価を行い、大規模なシミュレーションを行う際に除外できるパラメータ候補の絞り込みを行う。

5.2 節から 5.5 節においては、5.1 節で絞り込んだそれぞれのパラメータに対して規模の変更を通してより詳細なシミュレーションを行う。

5.6 節においては、それまでの結果を利用し NEURON のデフォルトと手動での最適化を図ったシミュレーションとの比較を行う。

また、コンパイラについては京環境で ICC を利用することができなかったため、クラスタ環境上でのみシミュレーションを行い、最適化の指標の一つとするにとどまった。

5.1 小規模シミュレーションでのパラメータ比較

本節では、ベンチマークモデルの中で実際の神経回路ネットワークと最も近いと考えられる Watts and Strogatz ネットワークに対して以下のパラメータを用いてシミュレーションを行った。

5.1.1 クラスタ環境

表 8: クラスタでのパラメータ

パラメータ	値の範囲
ノード数	1
MPI プロセス数	1~28
OpenMP スレッド数	1~16
SIMD 化	行う or 行わない
配列のくくり出し	行う（SIMD 化を行っているならば） or 行わない
シミュレーション時間	100ms
神経細胞数	256

プロセス数についてはクラスタでのコア数の上限まで、スレッド数については NEURON の内部で細胞単位でスレッド並列を行う上限を 16 と設定していたためその 16 を上限として設定した。また、配列のくくり出しに関しては SIMD 化の過程で変数を配列化する必要があるため、SIMD 化をした上で行うか否かという条件とした。

図 13 は、パラメータによる絞り込みを行っていない状態でそれぞれのパラメータに対して実行時間を表示したものである。

MPI process のグラフを例とすると、x 軸はプロセス数ごとに並べた順序（x 軸が 0 である時は、プロセス数 1-28 に対しもっとも実行時間が短いもの）、y 軸は実行時間を示している。

一方で、この図では実行時間が短い部分が一部の非常に遅い実行時間に影響されて潰れてしまっている。そこでパラメータと実行時間の関係を見るために、シミュレータでも利用している実行時間の上位 25% を用いる。その結果を図 14 に示す。

図 14 では、まず MPI プロセス数に関してプロセス数が 14 以下のものとそれよりも大きいもの間で実行時間に大きな乖離があることがわかる。本研究において求めるのは、実行時間がその環境において最も早くなるパラメータ一組であり、ここで求めたいものは 5.2 節以降に詳細にシミュレーションを行う意義のあるパラメータ候補であるため、ためこのように明確に乖離が見られるパラメータは探索対象から除外することができる。

同様にして、他の条件を固定した状態でパラメータの除外を行いパラメータによる有意差が生まれなくなるまで絞り込みを行った結果が図 15 である。

また絞り込みを行ったパラメータは次のようになる。

表 9: クラスタでの絞り込み後のパラメータ

パラメータ	値の範囲
ノード数	1
MPI プロセス数	23~28
OpenMP スレッド数	[1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 13, 14, 16]
SIMD 化	行う
配列のくくり出し	行う or 行わない

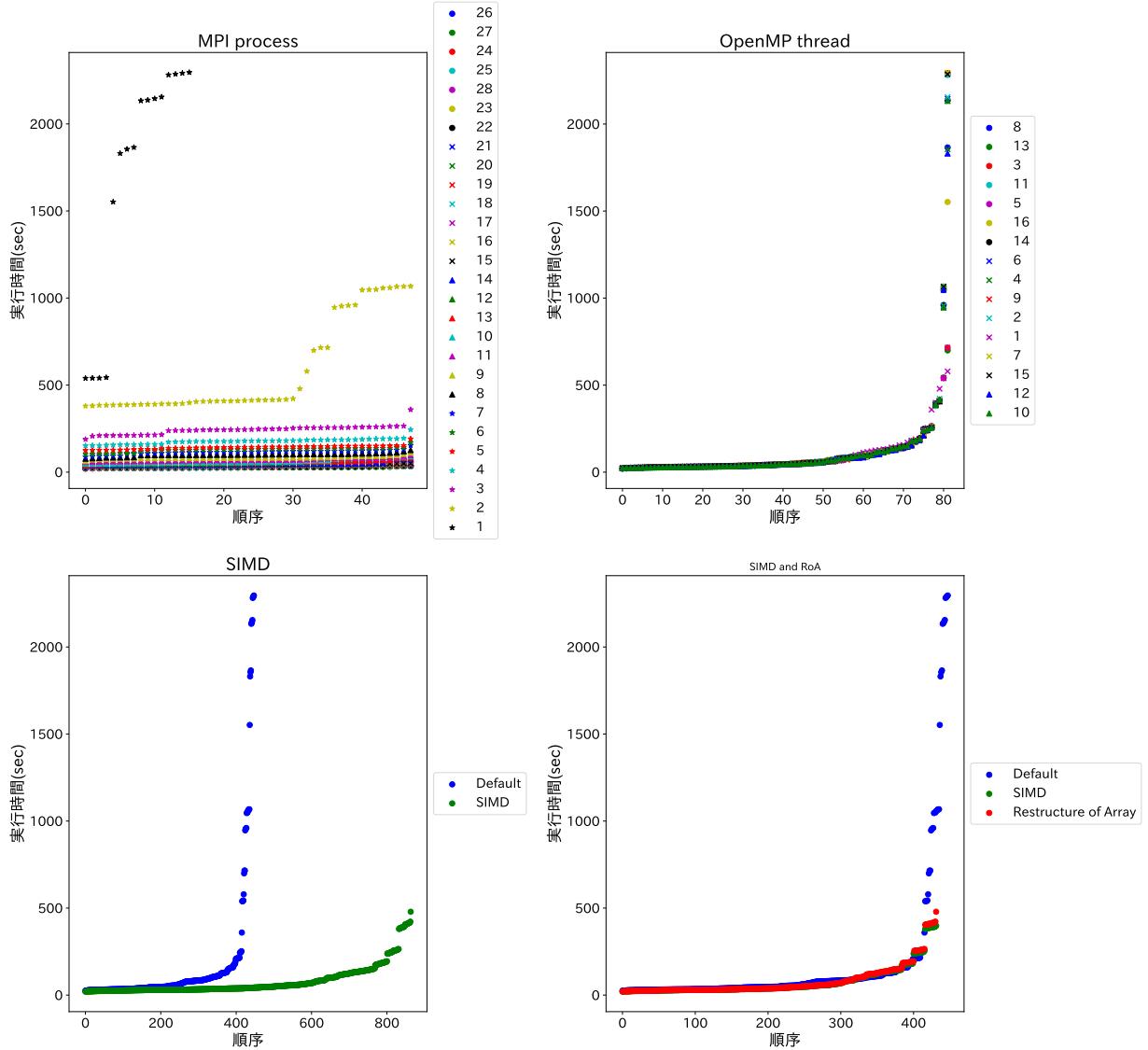


図 13: クラスタ 小規模シミュレーション結果

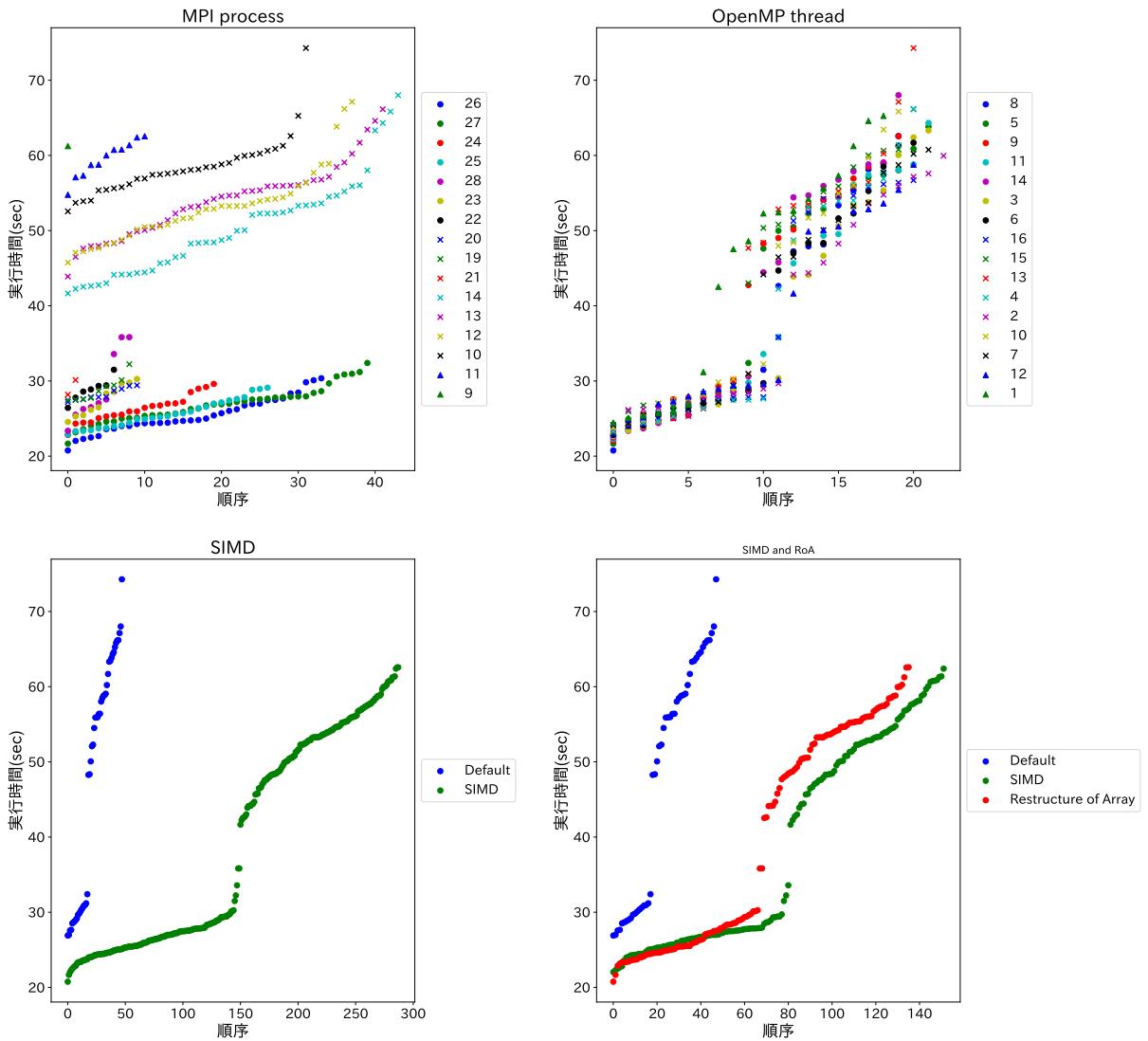


図 14: クラスタ 小規模シミュレーション結果 上位 25%

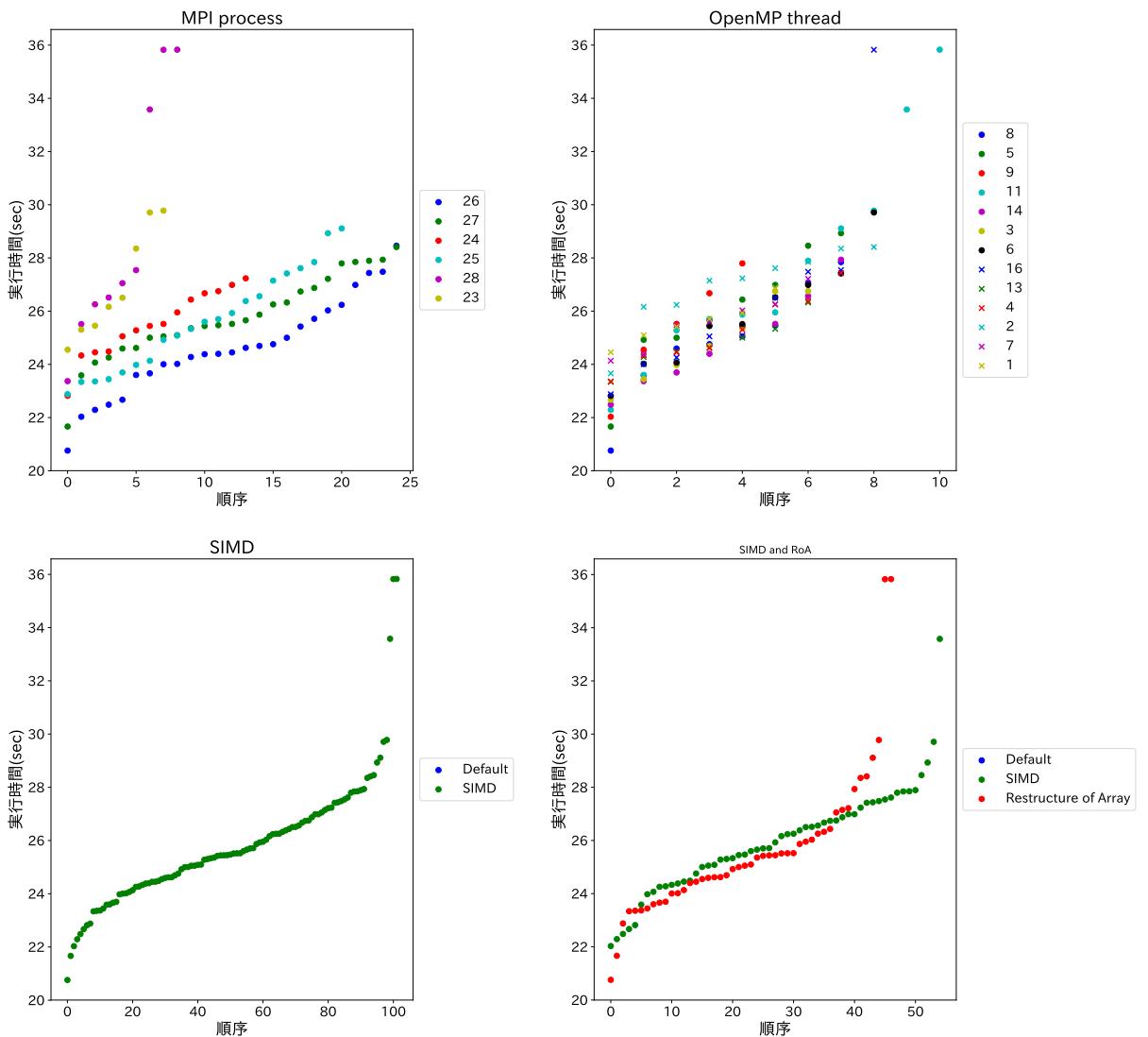


図 15: クラスタ 小規模シミュレーション結果 パラメータ絞り込み後

5.1.2 京環境

表 10: 京でのパラメータ

パラメータ	値の範囲
ノード数	8
MPI プロセス数	1~8
OpenMP スレッド数	1~16
SIMD 化	行う or 行わない
配列のくくり出し	行う (SIMD 化を行っているならば) or 行わない
シミュレーション時間	100ms
神経細胞数	256

京においてもクラスタ同様パラメータの絞り込みを小規模シミュレーションを用いて行った。

5.2 MPI プロセス数

5.3 OpenMP スレッド数

5.4 SIMD 化

5.5 配列のくくり出し

5.6 最適化結果の比較

6 考察

- ・考察を書きます

7 結論

参考文献

- [1] 宮本. No Title. 2014.
- [2] システム紹介 — 理化学研究所 計算科学研究機構 (AICS).
- [3] I. Dejanovic, R. Vaderna, Z. Milosavljevic, and Vukovic. TextX: A Python tool for Domain-Specific Languages implementation. *Knowledge-Based Systems*, Vol. 115, pp. 1–4, 2016.

8 謝辞

本研究は、情報理工学系研究科知能機械情報学専攻の神崎亮平教授のご指導のもと行われました。神崎亮平教授には、研究だけでなく大学院進学や就職といった自分の進路に関して言葉をかけてください、精神的な面で支えていただきました。

微小脳グループのリーダーである加沢知毅氏、宮本大輔さんのお二方には本当にお世話になりました。本当に色々… ハプトさんには、神経回路についての知見をいただいた他、海外の院への進学を考えていた際には快く英語の練習にも付き合っていただきました。

角田さんはアブストを見ていたり、…

最後に色々書きます :p