

# 自動チューニング化手法

井上裕太

平成30年1月20日

## 目次

<b>1 序論</b>	<b>3</b>
1.1 神経科学においてシミュレーションを行う意義 . . . . .	3
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 . . . . .	3
1.3 先行研究 . . . . .	4
1.3.1 宮本さんの . . . . .	4
1.3.2 片桐先生の . . . . .	4
1.4 研究の目的と手法 . . . . .	4
1.5 本論文の構成 . . . . .	4
<b>2 シミュレーションのモデルと環境</b>	<b>5</b>
2.1 シミュレーションモデル . . . . .	5
2.1.1 Hodgkin-Huxley モデル . . . . .	5
2.1.2 ベンチマークモデル . . . . .	5
2.2 シミュレータ . . . . .	5
2.2.1 全体構成 . . . . .	5
2.2.2 NEURON のコンパイル . . . . .	6
2.2.3 NMODL について . . . . .	8
2.3 シミュレーション環境 . . . . .	8
2.3.1 計算機性能 . . . . .	8
2.3.2 ジョブ実行環境 . . . . .	9
<b>3 最適化の手法</b>	<b>14</b>
3.1 モデルに依存するパラメータ . . . . .	14
3.1.1 SIMD 化 . . . . .	17
3.1.2 配列構造 . . . . .	17
3.2 実行マシンに依存するパラメータ . . . . .	20

3.2.1	スレッド数 . . . . .	20
3.2.2	プロセス数 . . . . .	20
3.2.3	配列サイズの変形 . . . . .	20
3.2.4	配列サイズの変形 . . . . .	20
3.3	コンパイルに関わるパラメータ . . . . .	20
<b>4</b>	<b>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</b>	<b>20</b>
4.1	環境設定スクリプト . . . . .	21
4.2	シミュレータ . . . . .	23
4.2.1	全体構成 . . . . .	23
4.3	トランスパイラ . . . . .	37
4.3.1	nmodl . . . . .	37
4.3.2	実装 . . . . .	38
<b>5</b>	<b>シミュレーション結果</b>	<b>40</b>
<b>6</b>	<b>考察</b>	<b>40</b>
<b>7</b>	<b>結論</b>	<b>40</b>

## 図 目 次

1	NEURON の全体構成 . . . . .	6
2	スーパーコンピュータ京（提供：理化学研究所） . . . . .	9
3	SIMD 化と配列のくくりだし . . . . .	18
4	シミュレータの全体像 . . . . .	24
5	単一ジョブ実行時の挙動 . . . . .	25
6	シミュレータ 構成 . . . . .	26
7	シミュレータ 状態遷移図 . . . . .	28
8	トランスパイラ 構成 . . . . .	39

## 表 目 次

1	京計算ノード構成 . . . . .	8
2	京プロセッサ構成 . . . . .	10
3	クラスタ性能 . . . . .	10
4	京でのジョブ関連コマンド . . . . .	11
5	京でのジョブの状態 . . . . .	12
6	クラスタでのジョブ関連コマンド . . . . .	12

# 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

簡単なモデルからじゃなく実験からとかのデータが本当は必要今ではmodelDBとかでモデルはたくさんあるけどシミュレーションまで持っていくと高速化が必要でもその高速化をいちいちやるのは本当は大変ボトムアップのアプローチを勘弁な方法で助ける手法を開発することでいろいろ楽にする・ボトムアップアプローチ

- ・当研究室ではそうしたモデル構築のために実験などを行い, こうしたデータを元に様々なシミュレーション系を構築してきた

脳機能の理解を目的として、スーパーコンピュータを用いた神経回路のシミュレーションが行われている。また、消費電力やスペックの利用できる時間が限られているといったリソースの問題やまた外部と通信する場合はリアルタイム以上の速度があるといい

- #### ・シングルとマルチノードの違いの説明（）

需要からシミュレーションの高速化・最適化が求められている。

また、現代の計算機にも多様な種類が存在し、それぞれに対する最適化も個別に行われてきた。本研究の目的はそれぞれの細胞モデルのシミュレーションコードを個々のアーキテクチャに合わせて、自動又は半自動的に最適化を行う手法を確立することである。

## 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

- ・神経回路シミュレーションには非常に大きな計算力が必要である一方で、こうした神経回路シミュレーションには非常に大きな計算能力が必要とされてきた。

本研究はスーパーコンピュータ京に関連するポスト京プロジェクトの一環として行われているが、

スーパーコンピュータを用いてもなお計算には多くの時間がかかる。

こうした状態を踏まえ、系の構築だけでなくシミュレーション自体の高速化・最適化が求められている。

- #### ・神経回路シミュレーションの最適化の難しさ

しかし、神経細胞には様々な種類のものが存在するため、個々の神経細胞のイオンチャンネルのモデルを最適化された形で実装するために、これまでそれぞれのモデルに対して多大な努力が行われてきた。・本研究の意義そこで、本研究では個々のイオンチャンネルモデルを自動で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

### 1.3.1 宮本さんの

すごい

### 1.3.2 片桐先生の

すごい

メモリが大事

## 1.4 研究の目的と手法

高速化・最適化への需要への項で述べたように, 本研究は個々の神経細胞のイオンチャネルモデルに対し汎用的な最適化手法を開発することである.

神経回路シミュレーションを行うソフトウェアは多数存在するが, 本研究では先行研究で用いられていた NEURON というソフトウェアを利用する.

NEURON では, 神経細胞のモデルとそれぞれの細胞の関係を微分方程式の形でモデルファイル (MOD ファイル) として記述することができ, nmodl というトランスパイラが MOD ファイルを C ファイルに変換することで実行している.

先行研究では, この生成された C ファイルに着目し手動での最適化を行っていたが, 本研究では C ファイルの生成と実行, 結果の集約を自動で行うことで複数のパラメータを試し, シミュレーションを実行する上で最適なパラメータを選択することを目指す.

## 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などが手動で行った最適化を自動で行うこととする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 モデル

[1]

#### 2.1.2 ベンチマークモデル

### 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 に組み込まれているこ

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によってシミュレーションが実行される。

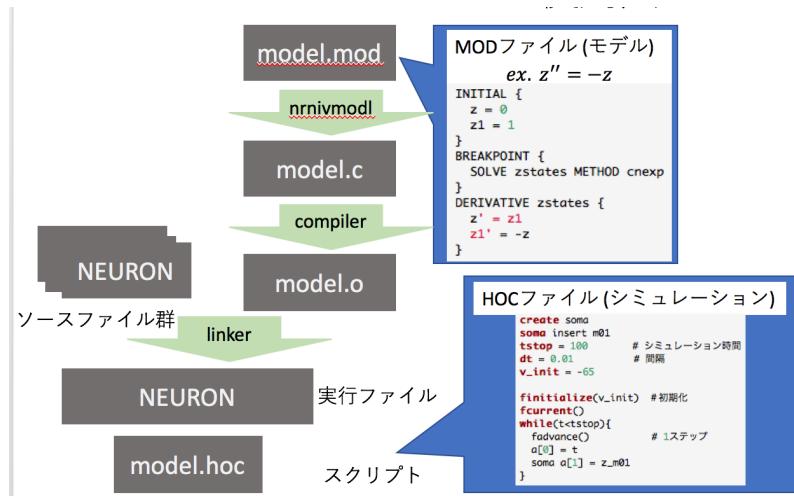


図 1: NEURON の全体構成

宮本らによる先行研究では、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 に変わるトランスペイラ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 1: クラスタでの NEURON のコンパイル

---

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

---

```
$ make  
$ make install
```

---

と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 京でのコンパイル

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 コンパイラー (nmodl) をコンパイルし実行形式を生成
2. MOD コンパイラーが 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 コンパイラー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 2: 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++  
$ 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 3: 京での NEURON 本体のコンパイル

---

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 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
```

---

### 2.2.3 NMODLについて

本研究では MOD ファイルを変換する

## 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2)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 (TODO: 出典).



©RIKEN

図 2: スーパーコンピュータ京 (提供: 理化学研究所)

### 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々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大幅に超えてしまったり,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,大型コンピュータではキューイングシステムを利用してプログラムが実行される.

キューイ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,キューイ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 4: 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"
```

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 ( $2 \times 2$ 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

```

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
16| #----- ステージアウトする先のディレクトリを設定 s -----#
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| #----- のスレッド数を指定 OpenMP -----#
26| export OMP_NUM_THREADS=1

```

表 4: 京でのジョブ関連コマンド

コマンド	説明
pjsub	pjsub 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

```

27 | #---- 利用するの実行形式のパスとオプションを指定する
      NEURON ----#
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=\
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渡すオプション NEURON ----#
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渡すオプション NEURON ----#
51 | MPIEXEC="mpiexec -mca mpi_print_stats 1"
52 | #---- プロファイラを指定(など) gprof ----#
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 5: 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 6: 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 7: 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 8: 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 | INITIAL {
61 |     rates(v)
62 |     m = minf
63 |     h = hinf
64 |     n = ninf
65 |
66 |

```

```

67 ? states
68 DERIVATIVE states {
69   rates(v)
70   m' = (minf - m) / mtau
71   h' = (hinf - h) / htau
72   n' = (ninf - n) / ntau
73 }
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
80     TO 100 WITH 200
81
82 UNITSOFF
83   q10 = 3^((celsius - 6.3) / 10)
84   alpha = .1 * vtrap(-(v + 40), 10)
85   beta = 4 * exp(-(v + 65) / 18)
86   sum = alpha + beta
87   mtau = 1 / (q10 * sum)
88   minf = alpha / sum
89   alpha = .07 * exp(-(v + 65) / 20)
90   beta = 1 / (exp(-(v + 35) / 10) + 1)
91   sum = alpha + beta
92   htau = 1 / (q10 * sum)
93   hinf = alpha / sum
94   alpha = .01 * vtrap(-(v + 55), 10)
95   beta = .125 * exp(-(v + 65) / 80)
96   sum = alpha + beta
97   ntau = 1 / (q10 * sum)
98   ninf = alpha / sum
99 }
100
101 FUNCTION vtrap(x, y) {
102   if (fabs(x / y) < 1e-6) {
103     vtrap = y * (1 - x / y / 2)
104   } else {
105     vtrap = x / (exp(x / y) - 1)
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を必要とするのは 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

・宮本さんの論文を参照また,SIMD 化を行う方法として avx や inline asm などを利用する方法も存在する.

これは現在の課題である汎用性を達成した上で, 汎用性を崩さないよう取り組む内容であるため, 今後の課題としたい.

・変数の配列化によるメモリアクセスの連續化

### 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];
}
```

---

と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 化の観点では配列の各要素がバラバラになってしまい非連続的になるため SIMD 命令を使いにくくなる可能性もある.

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.

( TODO: 事実確認) 演算を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ことと考えられる.

また, 配列をくくり出す際相互に関係のない配列を一つにまとめてキャッシュラインに入りきらなくなるなどの問題が発生する.

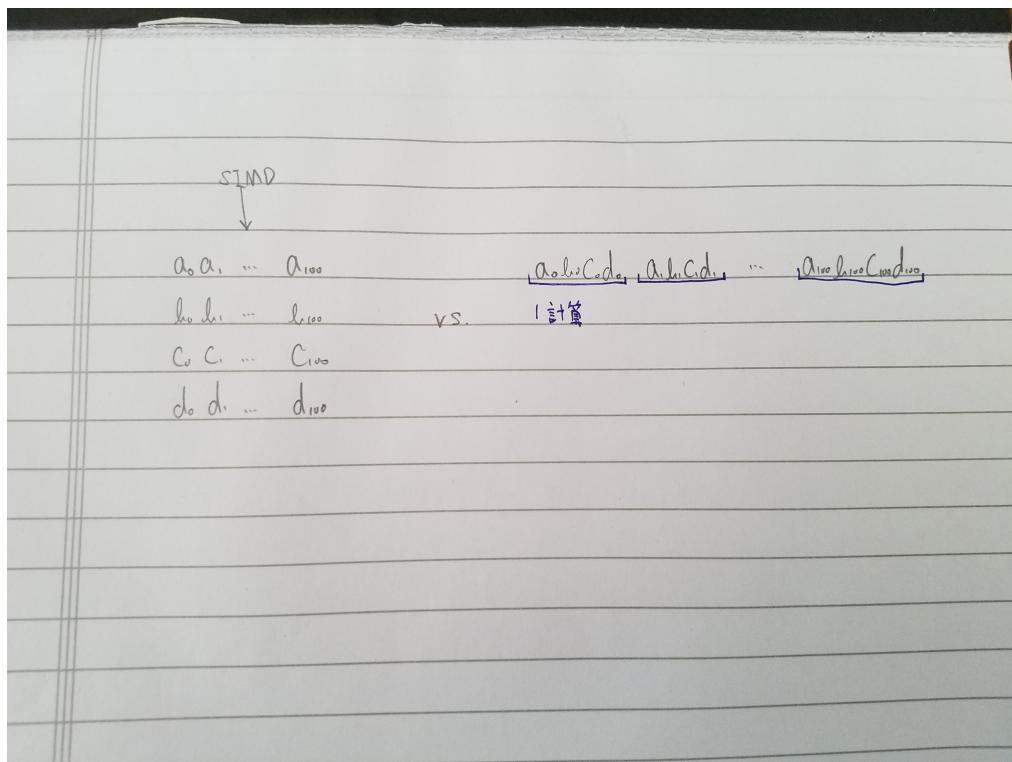


図 3: SIMD 化と配列のくくりだし

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 9: 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 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,
27             array_list.size())
28     } else {
29         code += "static double opt_table{0}[SIZE];\n".format(cnt)
30     }
31     # 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
32     array_cnt = 0
33     for array_name in array_list {
34         if array_list.size() > 1 {
35             code += "#define table_{0}(x) opt_table{1}[(x)][{2}]\n"\
36                 .format(array_name, cnt, array_cnt)
37             array_cnt += 1
38         } else {
39             code += "#define table_{0}(x) opt_table{1}[(x)]\n".format(
40                 array_name, cnt)
41         }
42     }
43     print(code)
44 }
```

### 3.1.2.1 Union-Find 木の利用

MOD ファイル内で

$$a = b * c \quad (1)$$

$$d = e * f \quad (2)$$

$$g = a + h \quad (3)$$

と記述されていたとすると、各ステップの計算において関連しうる変数は (a, b, c, g, h) と (d, e, f) であることがわかる。

そのため、相互に関連しえない変数に関しては互いに影響を及ぼさないため、グループとしてまとめ切り離して考えることができる。

このグループを作成するにあたり、Union-Find 木を用いて MOD ファイルに定義された式の変数を分類した。

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言える。

よって、仮に変数のグループが  $n$  個できた場合でも  $2n$  回の試行を行うことで最適な組み合わせを見つけることができる。（TODO: 論文を引用）・配列の構造変形（時間があれば）なければここを消す

## 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。

一方で、この並列化を行うまでのパラメータは実行するマシンごとに依存するものである。

ここで主に対象としたパラメータは OpenMP のスレッド数と MPI のプロセス数である。

### 3.2.1 スレッド数

OpenMP のスレッドに関するパラメータに関する説明（TODO：わあああ）

### 3.2.2 プロセス数

MPI プロセスに関するパラメータに関する説明（TODO：わあああい）

### 3.2.3 配列サイズの変形

### 3.2.4 配列サイズの変形

## 3.3 コンパイルに関わるパラメータ

TODO: 時間があれば記述する

## 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。

そのため、スーパーコンピュータ京・研究室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.  
上記の要件を満たすため, 以下にあげる 3 種類のプログラムを作成した.  
また, それぞれのプログラムは Python, Shell Script( TODO: reference) を使用して作成している.

## 4.1 環境設定スクリプト

作成したシミュレータ・トランスペイラは Python( TODO: reference) のモジュールとして作成したが, pip( TODO: reference )のようなモジュール管理ツールが存在しない環境(スーパーコンピュータ京)においては, モジュールとして公開するだけでは不十分である.

特にスーパーコンピュータ京では, デフォルトの Python( TODO: reference) のバージョンが 2.6.6, sudo 権限を有しないため外部プログラムのインストールが難しいという環境であったため, Pyenv を利用して汎用的な環境を作成することにした.

以下作成したスクリプトの概要とその用途を示す.

- Makefile

このプロジェクトの Makefile ( TODO: reference) .  
主に利用するのは以下の 3 つのコマンド

---

```
# scripts/setup_env_and_install_libraries.を実行する sh
make install

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

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

---

- scripts/

Makefile で実際に実行されるシェルスクリプト群であり, NEURON 本体のインストールと本研究で用いる Python の環境を整える役割を担っている.

以下に示したように, Pyenv を用いて Python3 ( TODO: reference) のインストールと実行の際に必要となるライブラリをインストールしている.  
(TODO: ライブラリの説明)

Listing 10: setup\_env\_and\_install\_libraries.sh

1	<code>#!/bin/sh</code>
2	<code>setupIfNecessary() {</code>
3	<code># .pyenv が存在しない場合はダウンロード</code>
4	<code># install 時に初回のみ実行される</code>
5	<code>if [ ! -d "~/.pyenv" ]; then</code>
6	<code># github からクローンする</code>
7	<code>git clone https://github.com/pyenv/pyenv.git ~/.pyenv</code>
8	
9	<code># 必要な環境変数の設定</code>
10	<code>echo 'export PYENV_ROOT="\$HOME/.pyenv"' &gt;&gt; ~/.bash_profile</code>

```

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 "$('
13     pyenv init -)"\nfi' >> ~/.bash_profile
14 exec "$SHELL"
15
16 # anaconda3-4.3.0 を Python として利用するための設定
17 pyenv install anaconda3-4.3.0
18 pyenv local anaconda3-4.3.0
19 pyenv rehash
20 fi
21
22 # genie 実行に必要な Python ライブラリのインストール
23 pip install textx
24 pip install pandas
25 pip install jinja2
26 }
26 setupIfNecessary

```

次に示すように、本研究の自動チューニングの対象である NEURON のインストールを行っている。  
NEURON のディレクトリが存在しない場合は、Github からクローンをした後に必要なディレクトリの追加を行う。  
一方で、存在する場合は最新のものへの更新を行っている。

- pull\_required\_projects.sh

Listing 11: pull\_required\_projects.sh

```

1 #!/bin/sh
2 pullIfNotExist() {
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
4     # NEURONneuron_kplusGithub
5     if [ ! -d "neuron_kplus" ]; then
6         git clone git@github.com:hashmup/neuron_k.git neuron_kplus
7         # NEURON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
8         # する -7.2
9         mkdir -p neuron_kplus/nrn-7.2/src/npy24
10        mkdir -p neuron_kplus/nrn-7.2/src/npy25
11        mkdir -p neuron_kplus/nrn-7.2/src/npy26
12        mkdir -p neuron_kplus/nrn-7.2/src/npy27
13    else
14        # がすでに存在している場合は NEURON を最新版に更新する NEURON
15        (cd neuron_kplus &&
16         git checkout . &&
17         git pull origin master)
18    fi
18 }
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以下に示すプログラムを作成した.

- ・MOD ファイルからパラメータとなりうる変数を自動で抽出し, それぞれの関係性を元に配列とその順序の候補を生成する.
- ・ジョブキューのシステムを持っているマシンにおいて, 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.
- ・実行結果を最適化前のデフォルトの結果と比較し, 実行結果に対して影響がないかを確認する.
- ・json 形式で実行するファイル, 各パラメータの範囲 (プロセス数は 1 から 10 など) を指定することができる.

### 4.2.1 全体構成

はじめにシミュレータプログラムを構成する要素について示す.  
( TODO: 章番号 ) にあるアルゴリズムで述べたように, 探索の対象となるパラメータは, モデルに依存するパラメータ, 実行マシンに依存するパラメータそしてコンパイルに関わるパラメータの 3 つに大別される.  
そのうち, モデルとコンパイルに関わるパラメータは実行形式の生成に関与し, 実行マシンに関わるパラメータはジョブスクリプトの生成に関わる.

#### 4.2.1.1 単一ジョブの実行

パラメータのシミュレーションを一度行う際のプログラムの動作を次に示す.

図 ( TODO: 番号 ) にあるようにジョブの実行はジョブの生成, ジョブの実行, ジョブ結果の集約の 3 段階に分かれており, ジョブの生成にかかる時間そしてジョブが実行されるまでのジョブキューでの待機時間がこの一連の動作の実行時間において大きな割合を示す.

ジョブが実行されるまでの待機時間は複数のジョブを実行する場合ではジョブの実行時間と同義になるため, これはシミュレーションの内容に応じて変わるが, ジョブの生成に関してはビルドするプログラムに大きな違いは現れず, 多くの場合ジョブの

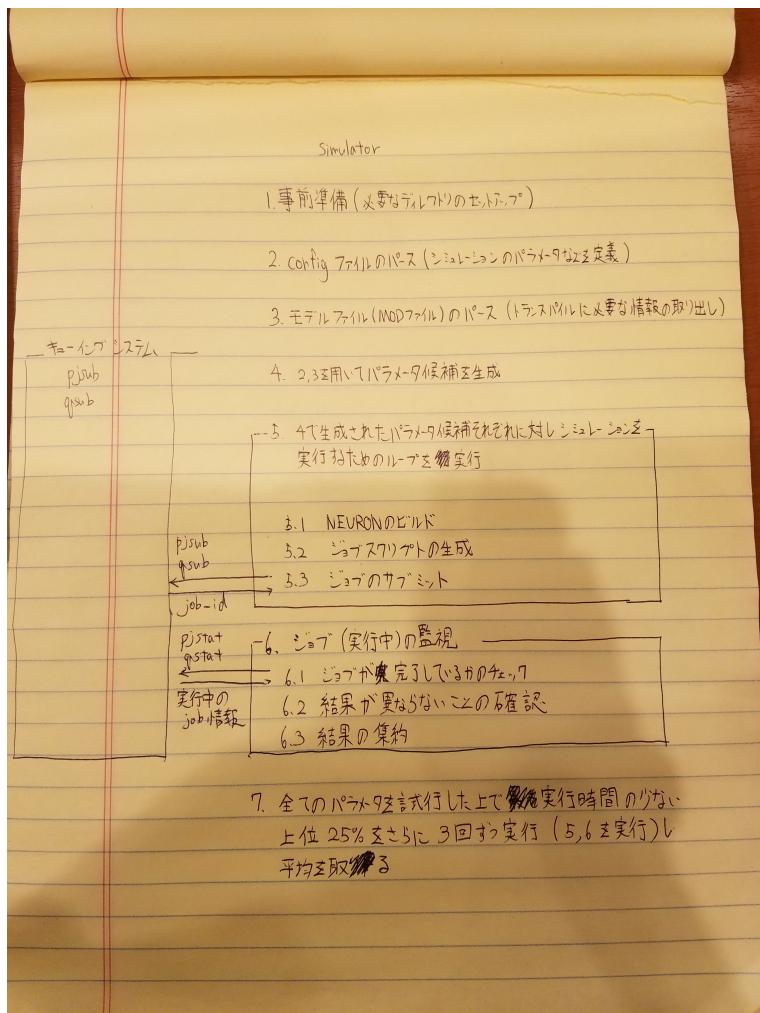


図 4: シミュレータの全体像

生成時間 < ジョブの実行時間という関係が成立する。

また、ジョブの生成部分について詳しく見ると、実行形式とジョブスクリプトそれぞれの生成にかかる時間は表 (TODO: 表を作る) のようになり、スーパーコンピュータ京、研究室クラスタ双方において実行形式の生成にかかる時間がが多いことがわかる。

#### 4.2.1.2 複数ジョブの並列実行

次にシミュレータの詳細について複数のジョブの並列実行を例として示す。

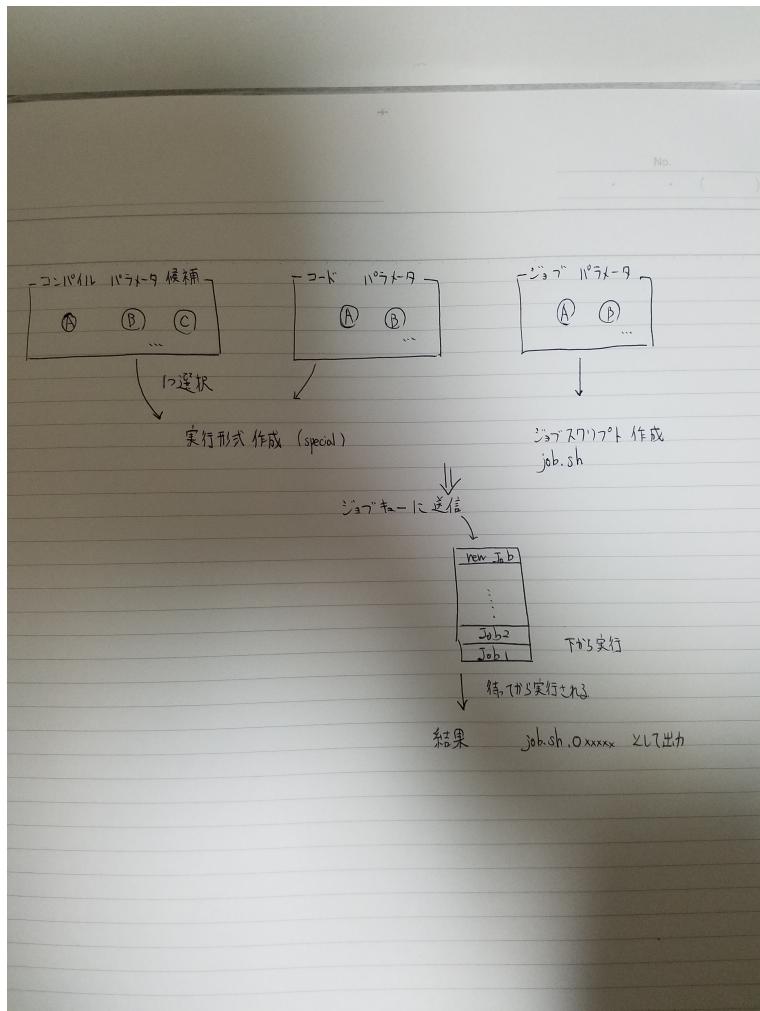


図 5: 単一ジョブ実行時の挙動

単一ジョブの実行例からパラメータを選択するためにコード, コンパイル, ジョブそれぞれの候補から一つを選択するという3重のループを組む際に, 最も内側のループ内でジョブスクリプトの生成を行い, 外側のループで生成した実行形式を使い回すことでシミュレーションをより高速に行うことができる。

また, スーパーコンピュータ京のように非常に多くのノードを持つマシンでない場合, ジョブの実行時間の方が長いため多数のジョブがジョブキューに溜まる状態になる。そのため, 外側のループで実行形式を生成する形を取ることで, ジョブの実行が溜まっているうちに実行形式のビルドを行うことができるようになり, 結果として最初の一回を除き以降のビルドはシミュレーションの実行時間に関与しなくすることができます。

本研究では以上を念頭に置きシミュレータのプログラムを作成した。単一ジョブの実行 (TODO: add ref) で述べたように、ジョブの実行をするにはジョブの生成、ジョブの実行、ジョブ結果の集約が必要となる。その中でジョブの実行は qsub や pbsub といった環境にインストールされているジョブ実行環境を利用するため、シミュレータはジョブの生成と結果の集約の役割を担う。

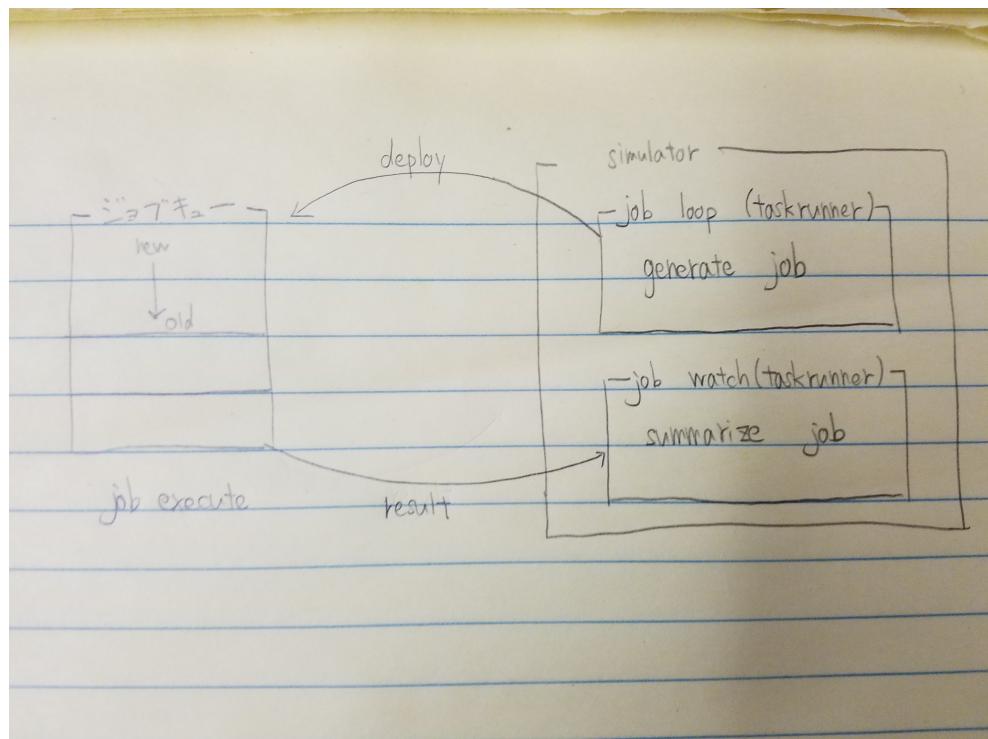


図 6: シミュレータ 構成

そのため、シミュレータは次の疑似コードで示す通りジョブを生成するためのループとメインスレッドから切り離されたスレッドでジョブの実行状況を監視し、ジョブが終了したタイミングで結果の集約を行うメソッドという二つの機能から成り立っている。

Listing 12: シミュレータ疑似コード

```

1 # グローバル変数
2 # ジョブの最大同時実行数を指定
3 MAX_NUM_JOB = 4
4 # 実行中のジョブを保存する配列 ID
5 # ジョブが完了したか否かの判定で利用する
6 running_job = []
7

```

```

8 | main():
9 |     # ジョブの監視を行う関数である watch_job()を別スレッドで立ち上げる
10|     watch_job()
11|
12|     # モデルに関連するパラメータ, コンパイルに関連するパラメータ
13|     # そして実行マシンに関するパラメータについて重のループを組み,  

14|     # 外のループで実行形式を作成し, 最内のループでジョブスクリプトを生成し
15|     # キューイングシステムにサブミットする
16|     while model_param_candidates.has_next()
17|         model_param = model_param_candidates.next()
18|         while compile_param_candidates.has_next()
19|             compile_param = compile_param_candidates.next()
20|             # モデルとコンパイルに関するパラメータを用いて,の実行形式を NEURON
21|             # 作成する
22|             build(model_param, compile_param)
23|
24|             while machine_param_candidates.has_next()
25|                 machine_param = machine_param_candidates.next()
26|                 # 実行マシンに関するパラメータを用いてジョブスクリプトを作成し,,,
27|                 # キューイングシステムにサブミットする
28|                 job_id = run(machine_param)
29|                 # キューイングシステムから返されるジョブを登録する ID
30|                 running_job.add(job_id)
31|                 # 実行マシンに関するパラメータの順番を最初に戻す
32|                 machine_param.reset()
33|                 # コンパイルに関するパラメータの順番を最初に戻す
34|                 compile_param.reset()
35|
36|     run(machine_param):
37|         # ジョブを一度に大量に投げることを避けるため現在実行中のジョブの数が,
38|         # 指定した最大同時実行数以上である時は秒間のスリープを繰り返す 10
39|         while running_job's size >= MAX_NUM_JOB:
40|             sleep 10
41|             # 実行マシンに関するパラメータを用いてジョブスクリプトを作成する
42|             job = make_job(machine_param)
43|             # キューイングシステムにジョブスクリプトをサブミットする
44|             job_id = submit(job)
45|             # キューイングシステムから返されるジョブを返す ID
46|             return job_id
47|
48|     watch_job():
49|         # 初期状態で実行中のジョブがない場合ジョブがサブミットされるまで待つ,
50|         while running_job's size == 0:
51|             sleep 10
52|             # 実行中のジョブそれが完了しているかをチェックし ID完了している場合,
53|             # ジョブの結果を集約し, 実行中のジョブの一覧から取り除く
54|             for job_id in running_job:
55|                 if is_finished(job_id):
56|                     summarize(job_id)
57|                     running_job.remove(job_id)

```

上記の疑似コードを状態遷移図の形で可視化したものが次になる。

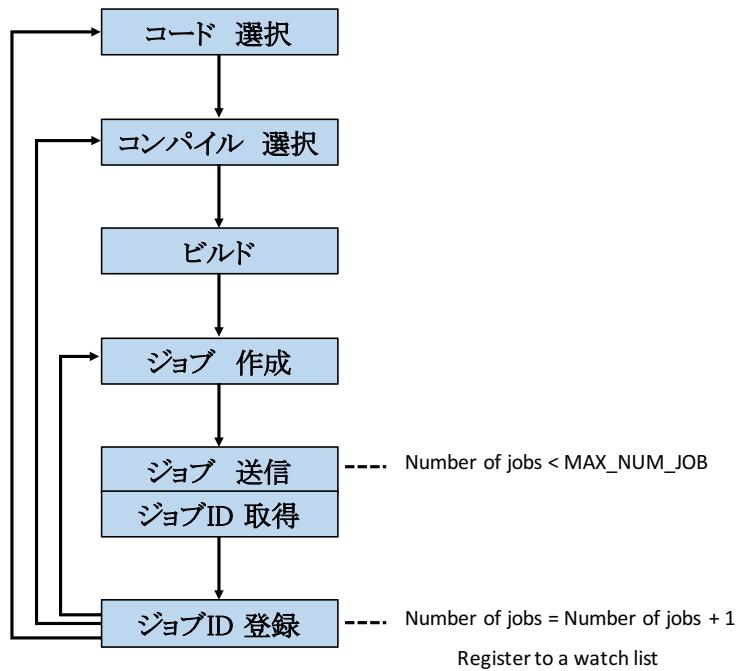


図 7: シミュレータ 状態遷移図

また、環境によってはステージングが存在せず、ジョブキューからジョブが実行される段階になって初めて実行形式とジョブスクリプトにアクセスすることになるが、その場合実行形式はそれぞれのビルドごとに、ジョブスクリプトはすべてのジョブに対して区別する必要がある。

そのため、実行形式に関してはビルドに関連するディレクトリのコピーをシミュレーションを実行する際に作成し、ビルドごとに参照するディレクトリを切り替え、ジョブスクリプトに関してはすべてのジョブに対して一意な ID を降りその ID を元に参照させることにした。

( TODO: add 図)

## ジョブ生成ループ

疑似コード内の 16-34 行目までがジョブ生成のためのループを構成している。

Listing 13: シミュレータ ジョブ生成ループ

```
1 | グローバル変数ジョブの最大同時実行数を指定実行中のジョブを保存する配列ジョ  
2 | ブが完了したか否かの判定で利用するジョブの監視を行う関数であるを別ス  
3 | レッドで立ち上げるモデルに関連するパラメータコンパイルに関連するパラ  
4 | メータそして実行マシンに関連するパラメータについて重のループを組み外  
5 | のループで実行形式を作成し最内のループでジョブスクリプトを生  
6 | 成しキューイングシステムにサブミットする  
7 |  
8 | while model_param_candidates.hasNext()  
9 |   model_param = model_param_candidates.next()  
10 |  while compile_param_candidates.hasNext()  
11 |    compile_param = compile_param_candidates.next()  
12 |    # モデルとコンパイルに関するパラメータを用いて,の実行形式を NEURON  
13 |    # 作成する  
14 |    build(model_param, compile_param)  
15 |  
16 |  while machine_param_candidates.hasNext()  
17 |    machine_param = machine_param_candidates.next()  
18 |    # 実行マシンに関するパラメータを用いてジョブスクリプトを作成し,,  
19 |    # キューイングシステムにサブミットする  
20 |    job_id = run(machine_param)  
21 |    # キューイングシステムから返されるジョブを登録する ID  
22 |    running_job.add(job_id)  
23 |    # 実行マシンに関するパラメータの順番を最初に戻す  
24 |    machine_param.reset()  
25 |    # コンパイルに関するパラメータの順番を最初に戻す  
26 |    compile_param.reset()
```

ループ内部では、プログラムのビルド、ジョブスクリプトの生成、そして生成した実行形式とジョブスクリプトをジョブキューに deploy するという 3 つのことを行っている。

すでに ( TODO: add re) で述べたようにパラメータ候補の選択順を実効形式の生成に関わるもの最先にすることで、非同期的に実行形式の生成とジョブの実行を行うことができる。

## ジョブ実行

Listing 14: シミュレータ ジョブ実行

```
1 | グローバル変数ジョブの最大同時実行数を指定実行中のジョブを保存する配列ジョ  
2 | ブが完了したか否かの判定で利用するジョブの監視を行う関数であるを別ス  
3 | レッドで立ち上げるモデルに関連するパラメータコンパイルに関連するパラ  
4 | メータそして実行マシンに関連するパラメータについて重のループを組み外  
5 | のループで実行形式を作成し最内のループでジョブスクリプトを生成し
```

キューイングシステムにサブミットするモデルとコンパイルに関するパラメータを用いての実行形式を作成する実行マシンに関するパラメータを用いてジョブスクリプトを作成しキューイングシステムにサブミットするキューイングシステムから返されるジョブを登録する実行マシンに関するパラメータの順番を最初に戻すコンパイルに関するパラメータの順番を最初に戻す

```

2 run(machine_param):
3     # ジョブを一度に大量に投げることを避けるため現在実行中のジョブの数が,
4     # 指定した最大同時実行数以上である時は秒間のスリープを繰り返す 10
5     while running_job's size >= MAX_NUM_JOB:
6         sleep 10
7     # 実行マシンに関するパラメータを用いてジョブスクリプトを作成する
8     job = make_job(machine_param)
9     # キューイングシステムにジョブスクリプトをサブミットする
10    job_id = submit(job)
11    # キューイングシステムから返されるジョブを返す ID
12    return job_id

```

スーパーコンピュータ京のような複数のユーザーが用いるシステムにおいて、ジョブを一度に大量に投げるのは好ましくない。

そのため、ジョブをジョブキューに投げる前に事前に設定した最大同時ジョブ実行数と現在の実行中のジョブの数を比較し、最大数と同数なのであれば待機する処理が必要である。

本研究では、グローバル変数として現在実行中のジョブのIDを保持するリストを定義し、そのリストの数と比較することで実現している。

また、後述するジョブ結果の集約においてこのジョブIDを保持するリストは別スレッドから参照されており、リスト内のジョブが完了した段階で mutex によってロックされた上で更新される。

### ジョブ完了判定

次に、ジョブが完了しているかの判定を行う方法について述べる。

ここではスーパーコンピュータ京と研究室クラスタを例にする。

京では pjstat、研究室クラスタでは qstat というコマンドを用いることで現在実行中のジョブを一覧で取得することができる。

この中でジョブの状態(State)を表す S の列に注目すると、まだジョブキューの中で実行を待っている Q、実行中の R、実行完了の C のようにジョブの状態を詳細に知ることができることがわかる。

また、このコマンドの出力結果も同一のフォーマットに従っているため、正規表現を利用することでジョブの状態を取得することができる。

ここではジョブが完了しているか否かの判定を行いたいため、

Listing 15: ジョブ完了判定

```

1 # を用いる場合にを取得するための正規表現 qstatjobID
2 job_cluster_exp = re.compile(
3     "(?P<id>\d+). \w+\s+\w+. \w+\s+\w+\s+\d+: \d+: \d+\s+(?P<state>\w+)\s+
4     +\w+\s+")
5 # を用いる場合にを取得するための正規表現 pjstatjobID

```

```

6   "(?P<id>\d+)\s+\w+. \w+\s+\w+\s+(?P<state>\w+)\s+[\s\w\d
7     \\\[\]\\:\\-]+")
# サブミットされたジョブのをストアしておくディクショナリー ID
8 # サブミットした段階で running_jobs[job_id] = として初期化される 0
9 running_jobs = {}

10
11
12 def is_job_still_running(job_id, environment):
13     # 環境がクラスタの場合
14     if environment == "cluster":
15         # コマンドを実行 qstat
16         res = execute("qstat")
17         # コマンドの返り値を改行で切り分ける qstat
18         job_lines = res.split('\n')
19         for line in job_lines:
20             # を取得する正規表現との各行を比較する jobIDqstat
21             m = job_cluster_exp.match(line)
22             if m is not None:
23                 state = m.group("state")
24                 if job_id == m.group("id"):
25                     # が () であるならば完了している StateCCComplete
26                     if state == "C":
27                         return False
28                     else:
29                         # の値が jobID0すなわち初期値ならば,がの返り値で
30                         # jobIDqstat
31                         # 現れた時にマークする
32                         if running_jobs[job_id] == 0:
33                             running_jobs[job_id] = 1
34                         return True
35             # が以前の実行時に現れ jobIDqstat今回の一回の実行時に一度も現れない場合
36             # qstat
37             # は完了しているとみなせる job
38             if running_jobs[job_id] > 0:
39                 return False
40             else:
41                 return True
42     # 環境が京の場合
43     elif environment == "k":
44         # コマンドを実行 pjstat
45         res = execute("pjstat")
46         # コマンドの返り値を改行で切り分け pjstat
47         job_lines = res.split('\n')
48         for line in job_lines:
49             # を取得する正規表現との各行を比較する jobIDpjstat
50             m = job_k_exp.match(line)
51             if m is not None:
52                 if job_id == m.group("id"):
53                     # では pjstat,がからに切り替わり表示されなくなるまでの
54                     # StateRUNSTO
55                     # 間隔が非常に短いため,での判定は行わずともジョブが完了

```

```

53 |     してから State
54 |     # 結果を集約するまでの時間の差は無視できるほど小さい
55 |     # ここではのマークのみを行う jobID
56 |     if running_jobs[job_id] == 0:
57 |         running_jobs[job_id] = 1
58 |     return True
59 |     if running_jobs[job_id] > 0:
60 |         return False
61 |     else:
62 |         return True

```

とすることでジョブ ID を元にジョブの状態を取得することができる。

#### 4.2.1.3 ジョブ結果の集約

Listing 16: シミュレータ ジョブ結果の集約

```

1 | グローバル変数ジョブの最大同時実行数を指定実行中のジョブを保存する配列ジョ
   | ブが完了したか否かの判定で利用するジョブの監視を行う関数であるを別スレッドで立ち上げるモデルに関連するパラメータコンパイ
   | ルに関連するパラメータそして実行マシンに関連するパラメータについ
   | て重のループを組み外のループで実行形式を作成し最内のループでジョブスクリプトを生成しキューイングシステムにサブミットするモ
   | デルとコンパイルに関するパラメータを用いての実行形式を作成する実行マシンに関するパラメータを用いてジョブスクリプトを作成しキューイングシステムにサブミットするキューイングシス
   | テムから返されるジョブを登録する実行マシンに関するパラメータの順番を最初に戻すコンパイルに関するパラメータの順番を最初に戻すジョブを一度に大量に投げることを避けるため現在実行中のジョブの数が指定した最大同時実行数以上である時は秒間のスリープを繰り返す実行マシンに関わるパラメータを用いてジョブスクリプトを作成するキューイングシステムにジョブスクリプトをサブミットするキューイングシステムから返されるジョブを返す
2 | watch_job():
3 |     # 初期状態で実行中のジョブがない場合ジョブがサブミットされるまで待つ,
4 |     while running_job's size == 0:
5 |         sleep 10
6 |         # 実行中のジョブそれぞれが完了しているかをチェックし ID完了している場合,
7 |         # ジョブの結果を集約し, 実行中のジョブの一覧から取り除く
8 |         for job_id in running_job:
9 |             if is_finished(job_id):
10 |                 summarize(job_id)
11 |                 running_job.remove(job_id)

```

様々なパラメータの組の中から最適な組み合わせを選びたいため、それぞれのジョブの結果とパラメータの組を結びつける必要がある。  
先行研究 ( TODO: ref ) では FLOPS を用いて計算性能を測っていたが、そのためにはそれぞれのモデルでシミュレーションを行う際に浮動小数点演算が何回行われるかを事前または事後的に知る必要がある。  
しかしながら本研究では、事前情報のない新規のモデルに対しても自動チューニングの対象となるため FLOPS は指標として適さない。そのため、本研究では NEURON 内での実行時間をシミュレーション内部で終了時に出力させ、指標として用いること

にした。

Listing 17: ベンチマークで NEURON 内部での実行時間を出力している箇所

```
1 if(pc.id == 0) {
2     printf("[id:0] Modeling Finished.\n")
3 }
4 if(pc.id == 0) {
5     printf("[id:0] Calculation Starts\n")
6 }
7 pc.barrier()
8 modelfin_time = pc.time
9 start_waittime = pc.wait_time
10
11 # 実際の計算
12 pc.psolve(tstop)
13
14 if(pc.id == 0) {
15     printf("[id:0] Calculation Finished !!\n")
16 }
17 pc.barrier()
18 stop_time = pc.time
19 stop_waittime = pc.wait_time()
20 stop_steptime = pc.step_time()
21 stop_sendtime = pc.send_time()
22
23 pc.barrier()
24 if(pc.id ==0){
25     printf("\nRESULT : \n")
26         printf(" * SpikeSendMax=%d\n", pc.spike_statistics() )
27         printf(" * step=%f sec, wait=%f sec, send=%f sec\n", stop_steptime
28             , stop_waittime-start_waittime, stop_sendtime)
29         printf(" * modeling time : %f sec\n", modelfin_time - start_time)
30         printf(" * core time : %f sec\n", stop_time-modelfin_time)
31 }
31 pc.barrier()
```

ここでは 29 行目で出力している core time が実行時間に該当する。

ジョブの実行時間のフォーマットは固定されているため、上記の正規表現を用いて取得することができる。  
Pythonにおいては、

Listing 18: ジョブ実行結果取得

```
1 # シミュレーションにかかった時間を取り出すための正規表現
2 time_exp = re.compile(
3     "\s+*\s+ core time : (?P<decimal>\d+).(?P<float>\d+) sec\s+")
4
5
6 def obtain_time(self, filename):
```

```

7   f = open(filename)
8   lines = f.readlines()
9   f.close()
10  # シミュレーション時間を出力しているのはファイルの中でも行のみだが 1,
11  # 並列で実行している場合表示される順番がランダムになっているため
12  # すべての行を確認する必要がある
13  for line in lines:
14      m = time_exp.match(line)
15      if m:
16          time = int(m.group("decimal")) + \
17              int(m.group("float")) * 10**(-len(m.group("float")))+1
18  return time

```

として取得できる。

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 19: 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,
23           nsendmax, nsend,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 20: 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 21: 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 22: 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     \")"
5 # 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
6 spike_stat_exp = re.compile(
7     "\[(?P<pid>\d+)\] NC = (?P<nc>\d+), SYN = (?P<syn>\d+), \
8     tmp_pre = (?P<tmp_pre>\d+), tmp_post = (?P<tmp_post>\d+)"
9 # 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
10 end_exp = re.compile(
11     "\[(?P<pid>\d+)\] nsendmax=(?P<nsendmax>\d+) nsend=(?P<nsend>\d+
12     \
13     nrecv=(?P<nrecv>\d+) nrecv_useful=(?P<nrecv_useful>\d+)"
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
```

# ハッシュ値を保存するセット

s = set()

# 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,

for filename in files:

f = open(filename)

lines = f.readlines()

f.close()

s.add(self.sort\_and\_hash\_log(lines))

# 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

return len(s) == 1

# の最大値を記憶しておくための変数 processID

maxid = 0

# 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

for line in lines:

m = spike\_stat\_exp.match(line)

if m:

pid = int(m.group("pid"))

spike\_stat[pid] = line

maxid = max(maxid, pid)

continue

m = spike\_exp.match(line)

# 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

if m:

pid = int(m.group("pid"))

idvec = int(m.group("idvec"))

if pid in spike:

spike[pid][idvec] = line

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.

## 4.3 トランスパイラ

先行研究 ( TODO: ref) 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.

### 4.3.2 実装

( 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

- ・nmodl 内でテンプレート化されている共通部分
- ・グローバル変数や関数定義部分
- ・ユーザ一定義関数部分
- ・NEURON 本体と関連する関数部分
- ・ODE (微分方程式) を計算する関数部分
- ・それぞれの関数や変数を NEURON とリンクさせる部分の 6 つに分けることができる.

これらの関数は最後のリンクさせる部分を除き独立性が高いため, 図 8 のように, textX を用いて作成した抽象木から最適化に用いる情報を取り出したのち, それぞれの部分を個別に最適化しベースとなるテンプレートに Jinja2 という Python のテンプレートエンジンライブラリを用いて埋め込む形で実装を行った.

またこの構成にすることで, それぞれのモジュールごとに担当する部位が小さく細かい最適化ができるだけでなくシミュレータ側で optimizer からの情報を元にどの最適化を行うかを指定することができる.

さらに, コードを生成する部分と抽象木を作成する部分そして最適化情報を解析する部分を分離することで, それぞれの独立性が高くなり, 後から最適化の手法を追加することも容易になっている.

#### 4.3.2.1 SIMD 化

SIMD 化を行うに当たって, 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

の部分から取得できることがわかる.  
pynmodl の中では Derivative と名前がつけられているため,

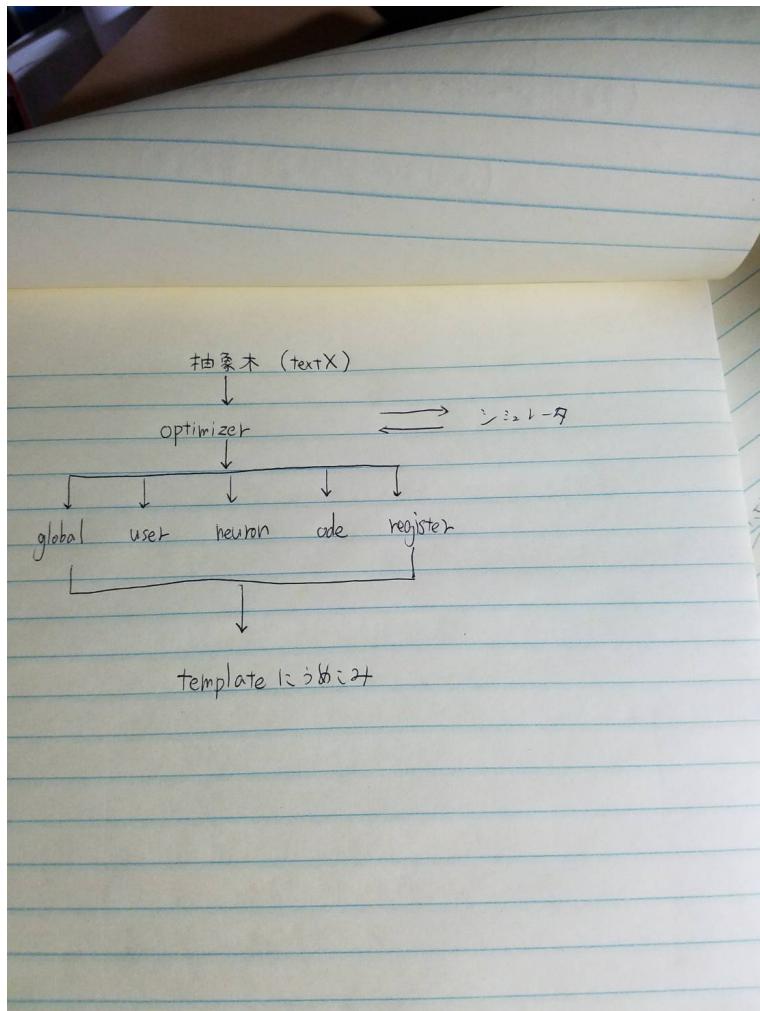


図 8: トランスペイラ 構成

とすることで、一覧を取得することができる。  
 またこうして取得した変数名を実際の C 言語のコードに変換する際に関与するのは定義部分のグローバル変数の定義部分と各関数内での呼び出しであるが、これらに関してはすべてマクロを定義することで配列構造に変わっていたとしても同様のアクセスができるように構成した。

#### 4.3.2.2 配列構造のくくり出し

アルゴリズムの章において、Union-Find 木を用いて MOD ファイル内の式を分類することでくくり出せる変数をグルーピングする手法について述べた。

ここではシミュレータにトランスペイラを組み込んで実行する際の実装について示

す。

## 5 シミュレーション結果

- ・結果を書きます

## 6 考察

- ・考察を書きます

## 7 結論

頑張りました  
Appendix

## 参考文献

- [1] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Bull. Math. Biol.*, Vol. 52, No. 1-2, pp. 25–71, 1990.