

自動チューニング化手法

井上裕太

平成30年1月17日

目次

1 序論	3
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	3
1.3.1 宮本さんの	3
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
2 シミュレーションのモデルと環境	4
2.1 シミュレーションモデル	4
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	7
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
3 最適化の手法	12
3.1 モデルに依存するパラメータ	13
3.1.1 SIMD 化	15
3.1.2 配列構造	15
3.2 実行マシンに依存するパラメータ	16

3.2.1	スレッド数	16
3.2.2	プロセス数	16
3.2.3	配列サイズの変形	16
3.3	コンパイルに関わるパラメータ	17
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	17
4.1	環境設定スクリプト	17
4.2	シミュレータ	19
4.2.1	全体構成	19
4.3	トランスパイラ	30
4.3.1	nmodl	30
4.3.2	アルゴリズム	31
4.3.3	実装	31
5	シミュレーション結果	31
6	考察	31
7	結論	31

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）	8
3	単一ジョブ実行時の挙動	20
4	シミュレータ構成	22
5	シミュレータ状態遷移図	23

表 目 次

1	京計算ノード構成	8
2	京プロセッサ構成	9
3	クラスタ性能	9
4	京でのジョブ関連コマンド	10
5	京でのジョブの状態	11
6	クラスタでのジョブ関連コマンド	11
7	クラスタでのジョブの状態	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

- ・ボトムアップアプローチ

- ・また, 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, 消費電力やシミュレーションの割り当て時間といったリソースの問題やリアルタイムデータ同化への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っている。
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.

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

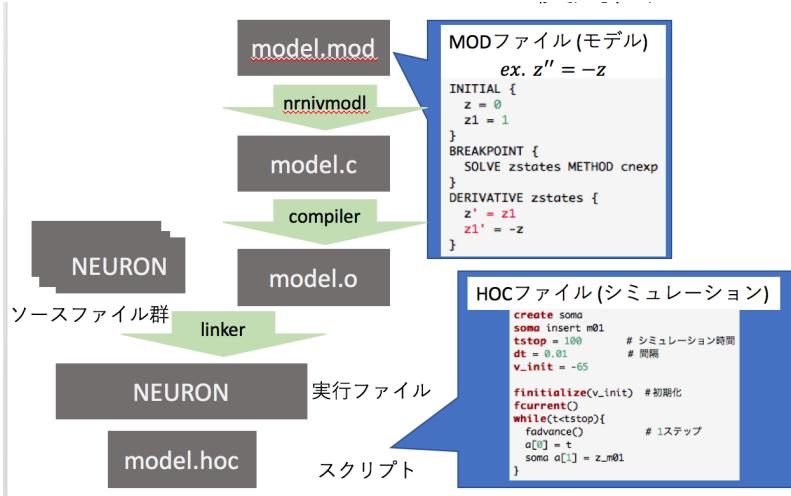


図 1: NEURON の全体構成

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

(TODO: 宮本さんのかから引用) 一般の x86 環境では,

Listing 1: 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

```

とすることで NEURON をコンパイルし, 実行形式を得ることができる。また, NEURON はデフォルトでは GUI 関係のライブラリもリンクするが, クラスタ上では必要な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 京上でのコンパイル

(TODO: 宮本さんのかから引用) 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 については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

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

```
$ 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

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

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）。



図 2: スーパーコンピュータ京（提供: 理化学研究所）

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

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

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

Listing 4: 京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 export OMP_NUM_THREADS=2
6 NRNIV="../specials/x86_64/special_mpi"
7 HOC_NAME="../hoc/bench_main.hoc"
8 NRNOPT=\
9 " -c MODEL=2" \
10 " -c NSTIM_POS=1" \
11 " -c NSTIM_NUM=400" \
12 " -c NCELLS=256" \
13 " -c NSYNAPSE=10" \
14 " -c SYNAPSE_RANGE=1" \
15 " -c NETWORK=1" \
16 " -c STOPTIME=50" \
17 " -c NTHREAD=16" \
18 " -c MULTISPLIT=0" \
19 " -c SPIKE_COMPRESS=0" \
20 " -c CACHE_EFFICIENT=1" \
21 " -c SHOW_SPIKE=1"
22 LPG="lpgparm -t 4MB -s 4MB -d 4MB -h 4MB -p 4MB"
23 MPIEXEC="mpiexec -mca mpi_print_stats 1"
24 PROF=""
25 cd $PBS_O_WORKDIR
26 echo "${PROF} ${MPIEXEC} ${NRNIV} ${NRNOPT} ${HOC_NAME}"
27 time ${PROF} ${MPIEXEC} ${NRNIV} ${NRNOPT} ${HOC_NAME}

```

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 1
s 0 1 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 export OMP_NUM_THREADS=2
6 NRNIV="..../specials/x86_64/special -mpi"
7 HOC_NAME="..../hoc/bench_main.hoc"
8 NRNOPT=\"
9 " -c MODEL=2" \
10 " -c NSTIM_POS=1" \
11 " -c NSTIM_NUM=400" \
12 " -c NCELLS=256" \
13 " -c NSYNAPSE=10" \
14 " -c SYNAPSE_RANGE=1" \

```

```

15 " -c NETWORK=1" \
16 " -c STOPTIME=50" \
17 " -c NTHREAD=16" \
18 " -c MULTISPLIT=0" \
19 " -c SPIKE_COMPRESS=0" \
20 " -c CACHE_EFFICIENT=1" \
21 " -c SHOW_SPIKE=1"
22 LPG="lpgparm -t 4MB -s 4MB -d 4MB -h 4MB -p 4MB"
23 MPIEXEC="mpiexec -mca mpi_print_stats 1"
24 PROF=""
25 cd $PBS_O_WORKDIR
26 echo "${PROF} ${MPIEXEC} ${NRNIV} ${NRNOPT} ${HOC_NAME}"
27 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: aaa

```
1 TITLE hh_k.mod squid sodium, potassium, and leak channels
2
3 COMMENT
4 This is the original Hodgkin–Huxley treatment for the set of sodium,
5 potassium, and leakage channels found in the squid giant axon membrane.
6 ("A quantitative description of membrane current and its
7 application
8 conduction and excitation in nerve" J.Physiol. (Lond.) 117:500–544
9 (1952).)
10 Membrane voltage is in absolute mV and has been reversed in polarity
11 from the original HH convention and shifted to reflect a resting potential
12 of –65 mV.
13 Remember to set celsius=6.3 (or whatever) in your HOC file.
14 See squid.hoc for an example of a simulation using this model.
15 SW Jaslove 6 March, 1992
16 ENDCOMMENT
17
18 UNITS {
19     (mA) = (milliamp)
20     (mV) = (millivolt)
21     (S) = (siemens)
22 }
23 ? interface
24 NEURON {
25     SUFFIX hh_k
26     USEION na READ ena WRITE ina
27     USEION k READ ek WRITE ik
28     NONSPECIFIC_CURRENT il
29     RANGE gnabar, gkbar, gl, el, gna, gk
30     GLOBAL minf, hinf, ninf, mtau, htau, ntau
31     THREADSAFE : assigned GLOBALs will be per thread
32 }
33 PARAMETER {
34     gnabar = .12 (S/cm2) <0,1e9>
35     gkbar = .036 (S/cm2) <0,1e9>
36     gl = .0003 (S/cm2) <0,1e9>
37     el = -54.3 (mV)
38 }
39 STATE {
```

```

41 |     m h n
42 |
43 |
44 | ASSIGNED {
45 |     v (mV)
46 |     celsius (degC)
47 |     ena (mV)
48 |     ek (mV)
49 |
50 |     gna (S/cm2)
51 |     gk (S/cm2)
52 |     ina (mA/cm2)
53 |     ik (mA/cm2)
54 |     il (mA/cm2)
55 |     minf hinf ninf
56 |     mtau (ms) htau (ms) ntau (ms)
57 }
58
59 ? currents
60 BREAKPOINT {
61     SOLVE states METHOD cnexp
62     gna = gnabar*m*m*m*h
63     ina = gna*(v - ena)
64     gk = gkbar*n*n*n*n
65     ik = gk*(v - ek)
66     il = gl*(v - el)
67 }
68
69
70 INITIAL {
71     rates(v)
72     m = minf
73     h = hinf
74     n = ninf
75 }
76
77 ? states
78 DERIVATIVE states {
79     rates(v)
80     m' = (minf-m)/mtau
81     h' = (hinf-h)/htau
82     n' = (ninf-n)/ntau
83 }
84
85 :LOCAL q10
86
87
88 ? rates
89 PROCEDURE rates(v(mV)) { :Computes rate and other constants at
90         current v.
91             :Call once from HOC to initialize inf at resting
92             v.
93             LOCAL alpha, beta, sum, q10
92             TABLE minf, mtau, hinf, htau, ninf, ntau DEPEND celsius FROM
93             -100 TO 100 WITH 200

```

```

94 | UNITSOFF
95 |     q10 = 3^((celsius - 6.3)/10)
96 |         :"m" sodium activation system
97 |     alpha = .1 * vtrap(-(v+40),10)
98 |     beta = 4 * exp(-(v+65)/18)
99 |     sum = alpha + beta
100|     mtau = 1/(q10*sum)
101|     minf = alpha/sum
102|         :"h" sodium inactivation system
103|     alpha = .07 * exp(-(v+65)/20)
104|     beta = 1 / (exp(-(v+35)/10) + 1)
105|     sum = alpha + beta
106|     htau = 1/(q10*sum)
107|     hinf = alpha/sum
108|         :"n" potassium activation system
109|     alpha = .01*vtrap(-(v+55),10)
110|     beta = .125*exp(-(v+65)/80)
111|     sum = alpha + beta
112|     ntau = 1/(q10*sum)
113|     ninf = alpha/sum
114|
115|
116 FUNCTION vtrap(x,y) { :Traps for 0 in denominator of rate eqns.
117     if (fabs(x/y) < 1e-6) {
118         vtrap = y*(1 - x/y/2)
119     }else{
120         vtrap = x/(exp(x/y) - 1)
121     }
122 }
123|
124 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

(TODO: 例) 配列を複数定義し, 一つの計算の中で呼び出す場合, Array of Structure でなく Structure of Array として定義した方が計算が高速化される場合がある.

これは以下に示す図によるものである

(TODO: 図) 一方で, 一つの構造体に多くの変数を定義してしまうとキャッシュラインに入らず, 結果としてキャッシュミスを多発し逆に遅くなるといった問題が生じる.

そこで, MOD ファイル内で定義された計算式の中から変数を抜き出し, それぞれの変数を Union-Find 木 (TODO : reference) を用いてグループにまとめ構造体として利用する変数の候補とし, それぞれの構造体を使うか否かでの高速化を図る.

Union-Find 木を用いているため, キャッシュという観点ではグループ間の関連はないと言えるため, これらのグループは独立にシミュレーションを行うことができる. 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: SOA の論文を引用) ・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)

3.2.3 配列サイズの変形

前節の配列構造において, 複数の配列を一つの構造体としてまとめた際に, メモリサイズが大きすぎる場合キャッシュラインに乗り切らず逆にキャッシュミスが多発するという問題について述べた

そのため, 前節のアルゴリズムによって有効だとされた構造体配列の中でさらにそれぞれの変数を利用するか否か, さらに配列のために確保するメモリサイズを変更することでキャッシュミスが減少するかをためす.

マシンに応じてメモリのサイズが異なるため、キャッシュ利用効率はこの配列のために確保するメモリサイズに大きく依存すると考えられる。

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・トランスパイラは 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

- make install
このコマンドでは以下に示す scripts/setup_env_and_install_libraries.sh を実行する。
- make pull
このコマンドでは以下に示す scripts/pull_required_projects.sh を実行する。
- make setup
上記の二つのコマンドを install,pull の順で実行する。
- 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.

- setup_env_and_install_libraries.sh

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 9: setup_env_and_install_libraries.sh

```
1 #!/bin/sh
2 setupIfNecessary() {
3     # .pyenv が存在しない場合はダウンロード
4     # install 時に初回のみ実行される
5     if [ ! -d "./.pyenv" ]; then
6         # github からクローンする
7         git clone https://github.com/pyenv/pyenv.git ~/.pyenv
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 -)"\nfi' >> ~/.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
```

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 10: pull_required_projects.sh

```
1 #!/bin/sh
2 pullIfNotExist() {
3     if [ ! -d "neuron_kplus" ]; then
4         git clone git@github.com:hashmup/neuron_k.git neuron_kplus
5         mkdir -p neuron_kplus/nrn-7.2/src/npy24
6         mkdir -p neuron_kplus/nrn-7.2/src/npy25
7         mkdir -p neuron_kplus/nrn-7.2/src/npy26
8         mkdir -p neuron_kplus/nrn-7.2/src/npy27
```

```

9 |     else
10|         (cd neuron_kplus &&
11|          git checkout . &&
12|          git pull origin master)
13|     fi
14| }
15| 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 段階に分かれしており、ジョブの生成にかかる時間そしてジョブが実

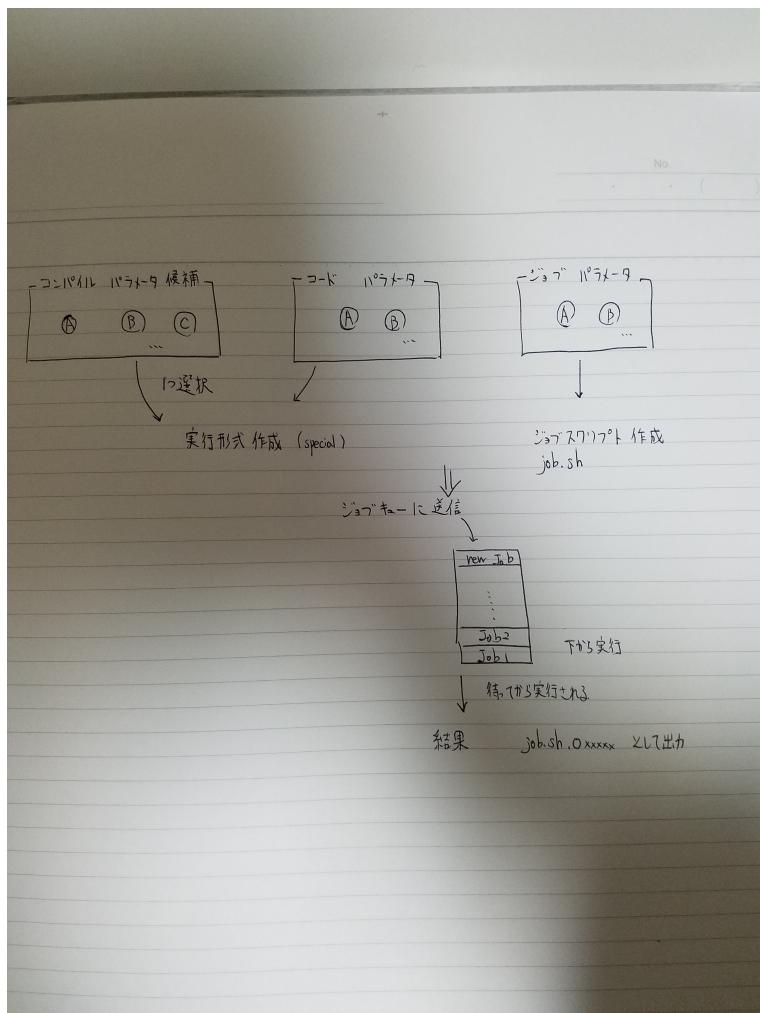


図 3: 単一ジョブ実行時の挙動

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。

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。
 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 11: シミュレータ疑似コード

```

1 # Global variables
2 MAX_NUM_JOB = 4
3 running_job = []
4
5 main():
6     watch_job()
7
8     while model_param_candidates.has_next():
9         model_param = model_param_candidates.next()
10        while compile_param_candidates.has_next():
11            compile_param = compile_param_candidates.next()
12
13            build(model_param, compile_param)
14
15        while machine_param_candidates.has_next():
16            machine_param = machine_param_candidates.next()
17
18            job_id = run(machine_param)
19            running_job.add(job_id)

```

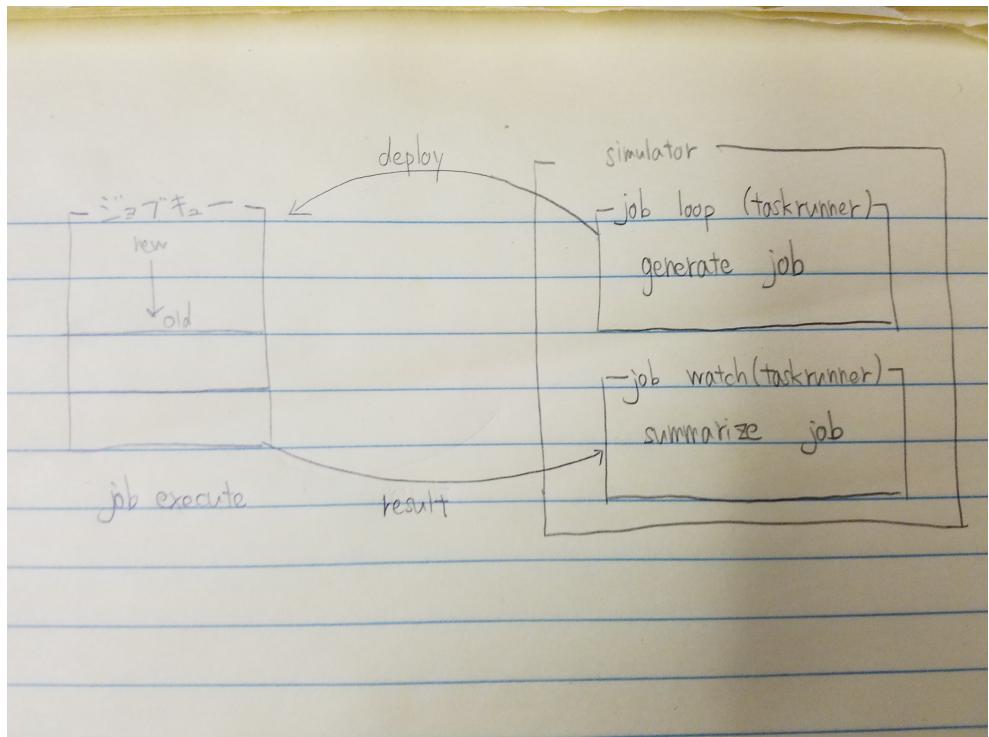


図 4: シミュレータ 構成

```

20
21     machine_param.reset()
22     compile_param.reset()
23
24 run(machine_param):
25     while running_job's size >= MAX_NUM_JOB:
26         sleep 10
27         job = make_job(machine_param)
28         job_id = deploy(job)
29         return job_id
30
31 watch_job():
32     while running_job's size == 0:
33         sleep 10
34         for job_id in running_job:
35             if is_finished(job_id):
36                 summarize(job_id)
37                 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。

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

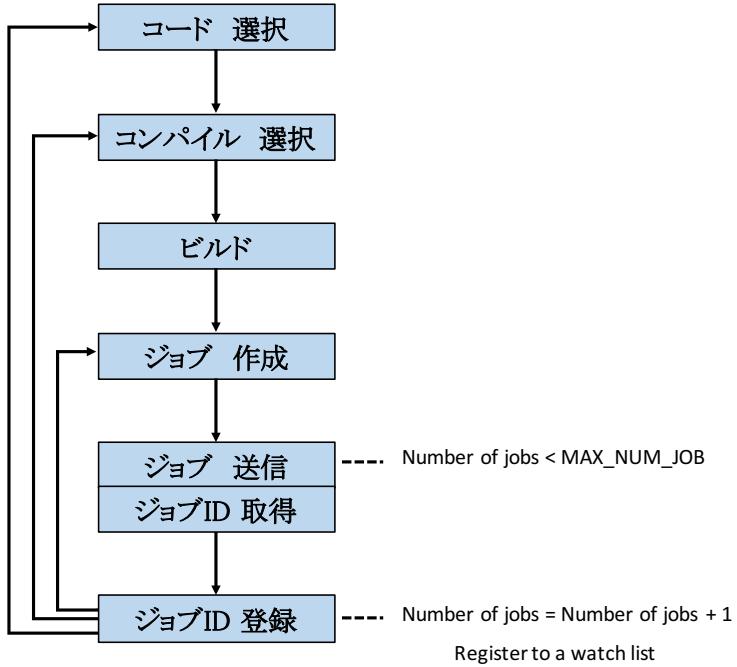


図 5: シミュレータ 状態遷移図

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 8-22 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 12: シミュレータ ジョブ生成ループ

```

1 while model_param_candidates.has_next()
2     model_param = model_param_candidates.next()
3     while compile_param_candidates.has_next()
4         compile_param = compile_param_candidates.next()
5
6         build(model_param, compile_param)
7

```

```

8 |     while machine_param_candidates.has_next()
9 |         machine_param = machine_param_candidates.next()
10| 
11|         job_id = run(machine_param)
12|         running_job.add(job_id)
13| 
14|         machine_param.reset()
15|         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 13: シミュレータ ジョブ実行

```

1 run(machine_param):
2     while running_job's size >= MAX_NUM_JOB:
3         sleep 10
4         job = make_job(machine_param)
5         job_id = deploy(job)
6         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。

4.2.1.3 ジョブ結果の集約

Listing 14: シミュレータ ジョブ結果の集約

```

1 watch_job():
2     while running_job's size == 0:
3         sleep 10
4         for job_id in running_job:
5             if is_finished(job_id):
6                 summarize(job_id)
7                 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 15: 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 16: ジョブ実行結果取得

```

1 time_exp = re.compile(
2     "\s+*\ core time : (?P<decimal>\d+).(?P<float>\d+) sec\s+")
3
4
5 def obtain_time(self, filename):
6     f = open(filename)
7     lines = f.readlines()
8     f.close()
9     for line in lines:
10         m = time_exp.match(line)
11         if m:
12             time = int(m.group("decimal")) + \
13                 int(m.group("float")) * 10**(-len(m.group("float")))+1)
14             return time

```

として取得できる。
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

>> pjstat研究室クラスタ

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

Job ID Name User Time Use S Queue

Job ID	Name	User	Time	Use	S	Queue
13282.cluster	job223.sh	inoue	00:09:29	C		cluster
13283.cluster	job224.sh	inoue	00:04:46	C		cluster
13284.cluster	job225.sh	inoue	00:06:48	C		cluster
13285.cluster	job226.sh	inoue	00:05:38	C		cluster
13286.cluster	job227.sh	inoue	00:05:36	C		cluster
13287.cluster	job228.sh	inoue	00:07:16	C		cluster
13288.cluster	job229.sh	inoue	00:05:38	C		cluster
13289.cluster	job230.sh	inoue	00:07:23	C		cluster
13290.cluster	job231.sh	inoue	00:05:37	C		cluster
13291.cluster	job232.sh	inoue	00:07:15	C		cluster
13292.cluster	job233.sh	inoue	00:05:38	C		cluster
13293.cluster	job234.sh	inoue	00:07:16	C		cluster
13294.cluster	job235.sh	inoue	00:05:37	C		cluster
13295.cluster	job236.sh	inoue	00:07:07	C		cluster
13296.cluster	job237.sh	inoue	00:05:35	C		cluster
13297.cluster	job238.sh	inoue	00:07:07	C		cluster
13298.cluster	job239.sh	inoue	00:05:36	C		cluster
13299.cluster	job240.sh	inoue	00:07:08	C		cluster
13300.cluster	job241.sh	inoue	00:05:38	C		cluster

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 17: ジョブ完了判定

```

1 job_exp = re.compile(
2     "(?P<id>\d+). \w+\s+\w+. \w+\s+\w+\s+\d+: \d+:\d+\s+(?P<state>\w+)\s+
3     +\w+\s+")
4
5 def is_job_still_running(self, job_id):
6     res = self.shell.execute("qstat", [], [], "")[0]
7     if type(res) is bytes:
8         res = res.decode('utf-8')
9     job_lines = res.split('\n')
10    if len(job_lines) > 2:
11        for line in job_lines[2:]:
12            m = job_exp.match(line)
13            if m is not None:
14                state = m.group("state")
15                if job_id == m.group("id") and state == "C":
16                    return False
17    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。

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 18: 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 19: 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 20: 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 21: 実行結果比較コード

```

1 spike_exp = re.compile(
2     "SPIKE : \t (?P<val>\d+.*\d*)\t (?P<idvec>[0-9]+) \[(?P<pid>\d+
3         \")"
4 start_exp = re.compile(
5     "\[(?P<pid>\d+)\] NC = (?P<nc>\d+), SYN = (?P<syn>\d+), \
6         tmp_pre = (?P<tmp_pre>\d+), tmp_post = (?P<tmp_post>\d+)"
7 end_exp = re.compile(
8     "\[(?P<pid>\d+)\] nsendmax=(?P<nsendmax>\d+) nsend=(?P<nsend>\d+
9         \
10        \
11    nrecv=(?P<nrecv>\d+) nrecv_useful=(?P<nrecv_useful>\d+)"
12
13
14
15
16
17
18
19
20
21
22 def verify(self, files):
23     s = set()
24     for filename in files:
25         f = open(filename)
26         lines = f.readlines()
27         f.close()
28         s.add(self.sort_and_hash_log(lines))
29     print(len(s))
30     return len(s) == 1
31
32
33
34
35
36
37
38
39
40
41
def sort_and_hash_log(self, lines):
    _lines = []
    start = {}
    end = {}
    spike = {}
    maxid = 0
    for line in lines:
        m = start_exp.match(line)
        if m:
            pid = int(m.group("pid"))
            start[pid] = line
            maxid = max(maxid, pid)
            continue
        m = spike_exp.match(line)
        if m:
            pid = int(m.group("pid"))
            idvec = int(m.group("idvec"))
            if pid in spike:
                spike[pid][idvec] = line
            else:
```

```

42 |         spike[pid] = {}
43 |         spike[pid][idvec] = line
44 |     continue
45 |     m = end_exp.match(line)
46 |     if m:
47 |         pid = int(m.group("pid"))
48 |         end[pid] = line
49 |     if maxid > 0:
50 |         maxid += 1
51 |     for pid in range(maxid):
52 |         _lines.append(start[pid])
53 |         for line in spike[pid]:
54 |             _lines.append(spike[pid][line])
55 |             _lines.append(end[pid])
56 |     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 をペ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) を利用した。
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

トランスパイラを作成するにあたり参考にした NEURON に付属しているトランスパイラである nmodl について述べる。

4.3.2 アルゴリズム

4.3.3 実装

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.