Vol.2021-HPC-180 No.17
2021/7/21

# 科学技術計算の並列プログラムを生成するための対話的最適化を行うJupyter併用型DSLの設計

西田 秀之[1,a)]　千葉 滋[1]

**概要**：近年，計算機環境を十全に生かすために科学技術計算プログラムの最適化の必要性が高まっている．しかし，複雑な入力データ・計算式を持つケースでは，手動・自動の両最適化にそれぞれ課題が存在する．本研究では，人間とコンパイラの利点を相補的に取り入れるために，Jupyter を用いて対話的に最適化を行うドメイン特化言語（DSL）を検討する．本 DSL では，コンパイラの解析結果を対話的に取得しつつ人間が最適化ポリシーを選択し，最後に負荷分散のなされた並列化プログラムを生成する．

**キーワード**：DSL, コード生成, 科学技術計算

## 1. はじめに

科学技術計算は，科学全般の諸原理を基にして計算機上で物性計算を行う分野である．化学や物理学を好例として，複数の分野において研究・工業の幅広い場面で科学技術計算の取り組みがなされて来た．実務的な側面から，時間のかかる数値解析やシミュレーションなどには高速に実行できるプログラムが欠かせない．しかし，複雑化する計算機環境を十全に活かせる科学技術計算コードの開発は難易度が高い．コード開発者は，興味のある分野と計算機に関する知識の両方を持ってコードの最適化を行う．

ライブラリを代表として，ソフトウェア開発効率を向上させる手段は古くから取り入れられてきた．近年，ドメイン特化言語（Domain Specific Language：DSL）[1] が注目されている．科学技術計算において頻繁に使われて来た Fortran や C ＋＋は汎用言語と呼ばれるものであり，汎用的に使えるように設計された言語である．DSL はそれらとは異なり，特定の領域での仕様に特化している．特化している都合上，言語の表現力には制限がかかるが，特定の目的を果たす時に強力な効果を発揮する．DSL の中でも，汎用言語との併用に優れた実装形式に，内部ドメイン特化言語（Embedded Domain Specific Language : EDSL）が存在する．EDSL は汎用言語のライブラリに近い形で使用することで，汎用言語の機能を利用しつつ独自の記法や機能を実現する．本研究では計算機環境を十全に活かす科学技術計算コードを開発するための EDSL を設計し，コード開発者の一助となることを目指す．

本論文ではまず科学技術計算と既存手法の背景と問題点を説明し (2 節)，次に，Jupyter を用いた対話的最適化手法を取り入れた DSL のアイデアと実例を説明する (3 節)．最後に結論 (4 節) を述べる．

## 2. 科学技術計算とコード最適化支援手法

本節では，研究のアイデアの導入に必要な背景を述べる．科学技術計算の具体例として量子化学計算を取り上げた後，コード最適化のための取り組みを説明する．

### 2.1 量子化学計算

量子化学計算は量子力学の原理を用いて物質の諸性質を計算する化学の一分野である．コンピュータの高性能化に伴って，より複雑な計算の実現や高速度計算が為されてきた．しかし，コンピュータの性能を十全に扱えるアプリケーションの開発には以前として困難が伴う．アプリケーションコードを最適化する他にも，計算理論そのものを計算機環境に合わせて考案する取り組みも行われて来た．高速なコードの実装を困難にする主な要因を以下に挙げる．

(1) 計算式と実際のコードの複雑性から，計算量の把握が難しい．
量子化学計算では，電子という物質の微視的な構成要素を考慮して計算を行う．この計算はある種の N 対計算とも言える上に，電子は存在確率を基に計算するた

[1] 東京大学大学院 情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
[a)] hnishi@csg.ci.i.u-tokyo.ac.jp

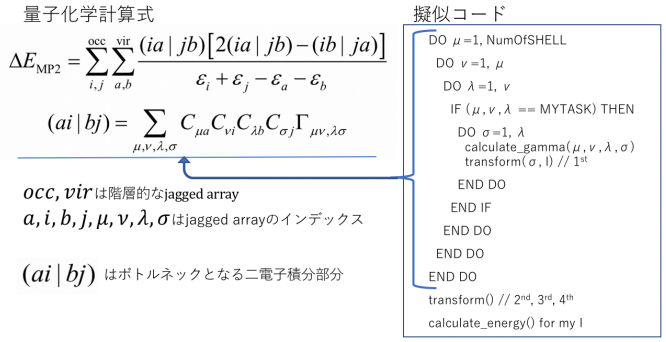


**図 1** 量子化学計算の例とそれを示す擬似コード

め，電子数 n の累乗の計算量が必要と言われている．（なお，厳密には電子軌道を考慮している．）一つの電子を特定するためには，階層的な多数のインデックスを用いなければならない．そのため，N 対計算のループ構造は入れ子が深いものとなる．多重ループ構造においては，各ループの計算量の把握は困難なものとなる．しかし，コードの高速化には，計算量に応じた最適化も求められる．ここで，最も簡単な部類の計算である MP2 法の計算式の一部と OpenMP・MPI のハイブリッド並列化をした際の擬似コードを図 1 に示す．このケースでの計算量の把握の難しさは計算式とコードの乖離および多重ループ構造によってもたらされる．この例では，二電子積分と呼ばれるものが 4 引数関数（Γ）と定数配列（$C$）の総和（Σ）で計算されることが示されている．この分野で頻繁に使われる言語の Fortran や C ＋＋では，ループ文で総和を構成する．どのループがどの総和に対応するかは直感的に分かり辛い上に，一般にループ構造はその自由度の高さにより，ミスを誘発する．このように，実際の計算式と記述されたコードの乖離はコードの保守性・可読性を下げる．

(2) 入力データの人的な把握管理が困難である．
数値計算やシミュレーションを扱うアプリケーションの多くは，入力データと共に使われる．その時の計算量は入力データの内容に応じて増減する．高速なコードの実行には，入力データに応じた最適化が欠かせない．多くの量子化学計算アプリケーションでは，入力データとして計算手法や原子番号・座標などの情報が特定のフォーマットに記載される．しかし，記載されない暗黙的なデータも多数存在する．それらはアプリケーションに埋め込まれた値と入力データを対応させて生成されるケースもある．例えば以下が入力データの例である．

```
$contrl nprint=-5
$basis gbasis=n31 ngauss=6 ndfunc=1
$scf diis=true npunch=0 conv=7 $end
$fmo
 nfrag=10 icharg(1)=
   0,0,0,0,0,0,0,0,0,0
 $end
 $fmoxyz
 H 1.0 -0.559000 -8.242000 -1.948000
 O 8.0 0.302000 -7.920000 -1.673000
 C 6.0 1.298000 -7.649000 -2.677000
 H 1.0 0.927000 -6.886000 -3.362000
```

上部に計算手法などの情報が記述され，下部に用いる原子・分子の座標が記述されている．しかし，ユーザの入力データ取り回しの都合上から必要最低限の情報のみ記述するのが一般的である．実際の計算では，原子情報から求められる電子の配置や関数の情報など原子数の数倍 ～ 数十倍のデータが用いられる．つまり，暗黙的で巨大な入力データが存在するため，人的な計算量の把握が難しい．別ファイルにそれらのデータを記載して読み込ませるケースもあるが，その場合でも巨大なデータを人力で管理することになる．暗黙的な巨大データにより計算量の把握が困難になる例として，分子を部分系に切り分けて並列して計算した後に，集約して一つの分子の物性を求めるケースがある．前述した通り，計算量は電子数の累乗となるが，それが暗黙的であるために計算量の想定が難しい．明示的であっても，巨大な入力データであれば，どの部分系がどれくらいの計算量なのかを見積もるのは人的には困難である．そのため，入力データを考慮したコードの最適化が難しい．

(3) 計算機技術の発展とともに最適化手法の選択肢が多い．
多くの研究により，アプリケーションコードを高速化するための最適化手法が年々開発されてきた．スーパーコンピューティングや GPU の利用なども含めると，非常に広範な研究が日々行われている．量子化学計算においても，それらの技術を取り入れてコードの高速化が図られている．しかし，量子化学の知識と計算機基盤の知識を統合して複雑なコードを書き換えるのは簡単ではない．頻繁に使われる高速化手法として，OpenMP と MPI のハイブリッド並列手法が知られている．ここでは多重ループに対するハイブリッド並列を簡単に示すために，OpenMP と MPI のプラグマを挿入することを考える．分子の物性計算を行うためのハイブリッド並列化した擬似コードを以下に示す．

```
#pragma MPI
for(y=0;y<MAXLEN2;y++){//電子1
  #pragma MP
  for(x=0;x<MAXLEN1;x++){//電子2
    --- inside algorithm ---
  }
}
```

ここに，より速く計算を終わらせるために，分子を部分系に分割してその内部で計算の後，統合する計算手法を取り入れたとする．その擬似コードは以下となる．

```
#pragma MPI
for(z=0;z<MAXLEN3;z++){//部分系
  #pragma MP
  for(y=0;y<MAXLEN2[z];y++){//電子1
    for(x=0;x<MAXLEN1[z];x++){//電子2
      --- inside algorithm ---
    }
  }
}
```

MPI は粗粒度ループに適用される場合が多いため，前述の擬似コードと違う for 文に対してプラグマが挿入されている．しかし，どの位置に MPI を適用するのが最適かを検討するには，入力データを考慮した試行錯誤を要する．例えば，部分系の中でもインデックスが 0 と 1 の計算量が多い場合には，以下のようにループ自体を分割したコードも考えられる．

```
for(z=0;z<2;z++){//部分系
  #pragma MPI
  for(y=0;y<MAXLEN2[z];y++){//電子1
    #pragma MP
    for(x=0;x<MAXLEN1[z];x++){//電子2
      --- inside algorithm ---
    }
  }
}
#pragma MPI
for(z=2;z<MAXLEN3;z++){//部分系
  #pragma MP
  for(y=0;y<MAXLEN2[z];y++){//電子1
    for(x=0;x<MAXLEN1[z];x++){//電子2
      --- inside algorithm ---
    }
  }
}
```

この擬似コードでは，分割されたループによってプラグマの挿入位置が異なっている．ここでは，単純なハイブリッド並列を 2・3 重ループに挿入する場合のみを考えた．しかし，図 1 のように多重ループが大きい場合には最適化可能性が増大するために，上記の例よりも最適化が困難である．次節で紹介する Work Stealing[3] などの動的な手法を取り入れる場合でも，計算手法や入力データに合わせた最適化が必要となる．

### 2.2 既存手法による実装と問題点

#### 2.2.1 汎用言語の延長としての最適化

科学技術計算の高速化のために，一般には Fortran や C++などの速度に優れるプログラミング言語が使用されている．それらの文法に基づいてコードを記述し，高速化のためのライブラリやコンパイラ機能を取り入れて最適化が行なわれてきた．しかし，暗黙的で巨大な入力データが存在する科学技術計算に対して，各入力データに特化した最適化をユーザが静的に書き下ろすのは現実的ではない．

そこで，それらを解決する動的な手法が提案されてきた．例えば，入力データや計算状況に合わせてランタイムに計算手法や計算資源を変更する手法である．これにより，コード開発における負担は最初の実装時のみで完結する．しかし，この動的な手法にも欠点が存在する．その一つが，複雑な入力データには適応できないケースが存在することである．他にも，動的な変更をすること自体に計算がかかり，必ずオーバーヘッドが発生する欠点がある．このため，入力データが事前に分かっている場合には，それに特化したコードを静的に書いておく方が高速であることが多い．例えば，MPI や OpenMP での並列化において使われる動的な戦略として，Work Stealing を挙げる．Work Stealing では，仕事のないプロセスが他のプロセスからタスクを奪い，代わりに計算する手法で，プロセスの待ち時間を減少させるために用いられる．これにより，ユーザはある並列化場所におけるタスクの不均衡は考慮せずに負荷分散を行うことができる．しかし，並列化対象とするタスクよりも細粒度のタスクの不均衡を考慮しなければならない場合には工夫が必要となる．例えば，Lazy Task Creation[4] と呼ばれる細粒度タスクを考慮したスケジューリング戦略では，より複雑なケースでの負荷分散が可能になる．しかし，Work Stealing や Lazy Task Creation では他のプロセッサとの通信においてオーバーヘッドが起こる．事前に入力データの不均衡が分かるのであれば，静的に負荷分散をしておき，他プロセッサとの通信を行わないほうが良い．

#### 2.2.2 既存 DSL の適用における問題点

特定の領域のソフトウェア開発を促進させる手段として，DSL が活用されてきた．DSL は汎用言語 (C 言語や Python など) とは異なり，言語の表現力に制限がかかるが，特定の目的を果たすために強力なサポートを行う．DSL の実装形式の一つに EDSL が存在する．EDSL はアプリケーションの主となる言語の機能を利用しつつ独自の記法や機能を実現する．しかし，量子化学計算分野に特化した EDSL はあまり知られていないため，画像処理用 EDSL として広く知られている Halide[2] を取り上げる．Halide は C++に埋め込まれた DSL で，画像処理の高性能なコードを簡単に記述するためのインターフェイスを持つ．その特徴の一つが，計算そのものの本質を示すアルゴリズムの記述と計算機への最適化を指示するスケジューリングの記述を分離した設計である．この設計の実現には，deep embedding[5] と呼ばれる技術が用いられ，二段階の C++コードのコンパイルを執り行う．ユーザは Halide の記法に則りコードを記述し，通常の C++コードと同様にコンパイル・実行する．Deep embedding により Halide のコードはランタイムに抽象構文木（Abstract Syntax Tree：AST）へと変換される．その後，独自の DSL コンパイラが起動し，スケジューリングの記述に基づいて最適化した C++コードが出力される．例えば，Halide の記述を簡略化した擬似 Halide コードと OpenMP の並列化の導入を考える．

一般的な C++コードと，等価な擬似 Halide コードを以下に示す．

```
// C++コード
#pragma omp parallel
for(int y = 0; y < 100; y++){
  for(int x = 0; x < 100; x++){
    f[x][y] = x + y;
  }
}

// 擬似Halideコード
Halide::Func f;
Halide::Var x, y;
Halide::Buffer<int> output;

f(x, y) = x + y;
f.parallel(y);
output = f.realize(100, 100);
```

Halide では，Func や Var と呼ばれる特別な型を主に用いてコードを記述する．Var 型は通常のループ構造のインデックスに相当し，計算式を表現するために用いられる．まず，Func 型変数の f に x + y の計算式を代入するが，x と y の定義域が未定義であることから分かるように，まだ実際の計算は行われていない．この f に parallel のメソッドを用いて OpenMP 並列化のヒントを与えている．最後に realize メソッドを呼び出し，引数に定義域を導入することで前述した AST が完成し，C++コードへの変換が行われる．出力された C++コードをコンパイル・実行することで，実際の数値を得ることができる．

Halide のような二段階のコンパイルを経て最適化コードを実行する機構をより複雑な科学技術計算向けに設計したい．しかし，Halide の応用には以下のような問題点がある．

(1) 入力データと計算式全体を俯瞰しての人的な最適化が困難である．Halide は画像処理に特化しているため，対象となるデータはシンプルな配列となる．しかし，他の科学技術計算ではより複雑な入力データまで考慮するケースが頻発する．アルゴリズムとスケジューリングが同じファイル・同じタイミングで読み込まれる形式では，計算量などの予測が難しいため，人的な最適化選択が難しい．

(2) より複雑なケースでの自動最適化手法があまり知られていない．Halide では自動でスケジューリングし，最適化されたコードを出力する機能がある [6][7][8][9]．また，類似 DSL においても最適化手法は様々開発されてきた [10][11][12]．しかし，複雑なデータ・複雑な計算式に対しての自動最適化手法はあまり知られておらず，従来法の応用は探索時間が爆発してしまうため，複雑なケースへの適用が困難である．

## 3. Jupyter 併用型 DSL

本研究では，ユーザとコンパイラの知見を相補的に取り込むことでコード最適化を支援する環境構築を目指す．具体的には，最適化のための補助情報を DSL コードのコンパイル時にユーザが取得し，同時に最適化ポリシーを指示することで最適化コードを生成する DSL を提案する．

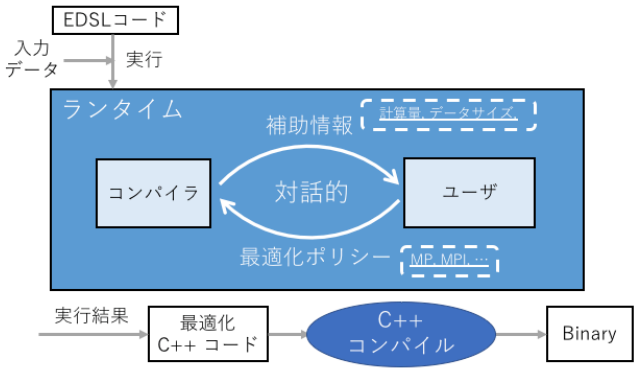


**図 2** システム概念図

### 3.1 対話的コンパイラによるコード出力

システムの概念図を図 2 に示す．本システムでは Halide と同様に EDSL コードは二段階にわけて実行される．一段階目の実行では最適化された C++コードが生成される．二段階目の実行ではこれをコンパイルして実行し，求める計算結果を得る．本システムの特徴は，一段階目の実行時に，コンパイラとユーザが対話的に情報を交換することでコードの最適化を行う点である．コンパイラは事前に入力データを読み込んでいるので，ユーザには予想される計算量やデータサイズなどの補助情報を提供することができる．これは，人的な把握が困難な巨大で複雑なデータの取り扱いをコンパイラがアシストする機構である．ユーザは補助情報と自身の知見を組み合わせて，どのように最適化を行うかを決定する．これにより，コンパイラの自動最適化が困難なケースにも人的なアシストと共に適用が可能となる．

このシステムを Jupyter 上で実現した．Jupyter は広く使われる一種の対話的シェルであり，ノートブック形式のファイルを用いてコードの管理・実行やデータの可視化を簡単に行える開発環境である．Jupyter 上での実行の様子を図 3 に示す．

システムの実行は次の 3 ステップで行う．

(1) EDSL を用いたコードの記述と実行．（セル In[1], セル In[2]）
科学技術計算コード開発者は EDSL をライブラリとして利用し，EDSL の記法でコードの記述を行う．この時，計算機上の最適化手法は考えずに計算の本質を示す記述のみを行う．これにより，コード改修や新規の実装においては先に実行するセルのみを注目すれば良い．このセルの実行時点では，いわゆる deep embedding の手法に基づき，対象となる計算式自体が

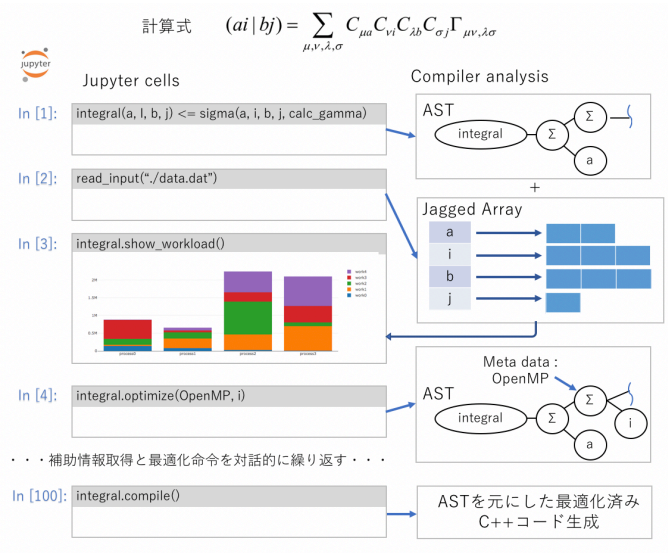


**図 3** Jupyter 上での実行フロー

読み込まれ，AST が生成される（セル In[1]）と共に，入力データの読み込みと暗黙的データの構築が行われる（セル In[2]）．これにより，入力データを分析しながら最適化を行うことができる．

(2) 対話的最適化．
前のステップの実行により，独自に組み込まれたコンパイラにとって AST と入力データが既知の情報となった．そこで，入力データの大きさやユーザが設定した実行環境から補助情報を生成する．一般的なプロファイリングと異なり，予測値を求める．ユーザは別のセルで，本システムへの命令を実行することで，計算式や入力データ情報，各種サポート情報を確認することができる（セル In[3]）．確認した情報を元にユーザは最適化ポリシーを選択する（セル In[4]）．セルの実行により，本システムは AST へのメタデータとして最適化情報を取り込む．このステップを対話的に繰り返すことでユーザが望む最適化の情報を含んだ AST が構築される．

(3) コード出力と計算の実行．
対話的な最適化が終了した後，別のセルで特別な命令を実行することで AST を元に C++の最適化コードが生成される（セル In[100]）．

本研究では，Ruby の EDSL の形で実装を行なった．出力するのは実行速度の観点から C++コードである．

### 3.2 対話的最適化の例

Jupyter を用いた DSL の活用を逐次的に説明する．

本 DSL では，Fortran や C++と同様に扱う変数の明示的な宣言が必要となる．式を定義するためのクラスとインデックスを示すためのクラスのインスタンスを用いて変数宣言を行う．この際，入力データに関する記述も同時に行う．例えば 2.1 節で示した入力データに加えて電子の配置などの暗黙的なデータの生成を行う．本 DSL は Ruby の EDSL であるため，一般的な Ruby の記法を用いてデータ生成を行うことができる．

```
computer = System.instance
computer.set_process_number(4)

input = Func.new("input")
input.set_data(data_file)

nuclear_repulsion = Func.new("nuclear_repulsion")
density = Func.new("density")
kinetic_term = Func.new("kinetic_term")
kinetic_pgto = Func.new("kinetic_pgto")
calc_kinetic_cgto = Func.new("calc_kinetic_cgto")
kinetic_cgto = Func.new("kinetic_cgto")
t = Func.new("t")

nuclear_attraction_CGTO = Func.new("nuclear_attraction_cgto")
v = Func.new("v")
h_core = Func.new("h_core")
g = Func.new("g")
fock = Func.new("fock")

calc_energy_0 = Func.new("calc_energy_0")
energy_0 = Func.new("energy_0")
total_energy = Func.new("total_energy")

subsystem = Rank.new("subsystem")

bfs = Rank.new("cgto")
bfs1 = Rank.new("cgto1")
bfs2 = Rank.new("cgto2")

pgto = Rank.new("pgto")
pgto1 = Rank.new("pgto1")
pgto2 = Rank.new("pgto2")

input.has(subsystem.has(bfs.has(pgto)))
bfs.is(bfs1).is(bfs2)
pgto.is(pgto1).is(pgto2)

input.print_rank()
```

**図 4** DSL での変数宣言

```
nuclear_repulsion[subsystem] <= icdouble(0)
density[bfs1, bfs2] <= icdouble(0)

kinetic_term[pgto1, pgto2] <= icdouble(10.0)
kinetic_pgto[pgto1, pgto2] <= kinetic_term[pgto1, pgto2] * pgto1.attr("norm_factor") * pgto2.attr("norm_factor")

calc_kinetic_cgto[pgto1, pgto2] <= pgto1.attr("coefficient") * pgto2.attr("coefficient") * kinetic_pgto[pgto1, pgto2]
kinetic_cgto[bfs1, bfs2] <= icsigma(pgto.in(bfs1), pgto.in(bfs2), calc_kinetic_cgto)

t[bfs1, bfs2] <= kinetic_cgto[bfs1, bfs2]

nuclear_attraction_CGTO[bfs1, bfs2] <= icdouble(10.0)
v[bfs1, bfs2] <= nuclear_attraction_CGTO[bfs1, bfs2]
h_core[bfs1, bfs2] <= t[bfs1, bfs2]+ v[bfs1, bfs2]
g[bfs1, bfs2] <= icdouble(0.0)
fock[bfs1, bfs2] <= h_core[bfs1, bfs2] + g[bfs1, bfs2]

calc_energy_0[bfs1, bfs2] <= density[bfs1, bfs2] * (h_core[bfs2, bfs1] + fock[bfs2, bfs1])
energy_0[subsystem] <= icsigma(bfs1.in(subsystem), bfs2.in(subsystem), calc_energy_0) * icdouble(0.5)
total_energy[subsystem]<= energy_0[subsystem] + nuclear_repulsion[subsystem]
total_energy.print()
```

**図 5** DSL での計算式記述

次に，図 5 のように計算式の記述を行う．独自にデザインした記法を用いており，階層的な科学技術計算を簡単に実装できるようにしてある．このセルを実行すると次の出力が得られる．

```
total_energy.print()
"input"
" subsystem"
"   cgto : cgto1 : cgto2"
"   pgto : pgto1 : pgto2"
"---- algorithm ----"
"total_energy : "
"  argument rank : subsystem, "
"  function : "
"  binary_: +"
"    lhs : "
"      energy_0 : "
"        argument rank : subsystem, "
"        function : "
"        binary_: *"
"          lhs : "
"            sigma function"
"              loop index : cgto1 in subsystem, cgto2 in subsystem, "
"              function :  "
"                calc_energy_0 : "
```

**図 6** DSL での AST 生成

図 6 のように AST が構築される．最適化機構の実行前には AST の組み換えができるため，複数セルに記述を分散することで，計算式の一部のみを組み替えた実装などが簡単に行える．同時に，暗黙的なデータを含む入力データが読み込まれたため補助情報の生成が行われる．

次に，ユーザがある計算部分に対して MPI のディレクティブを挿入することを想定する．

図 7 のように，ユーザは EDSL が提供するメソッドを利用して MPI のディレクティブを挿入し，次のセルで仕事量を確認する命令を行う．このケースでは棒グラフが表示され，色分けされた四本の棒がそれぞれプロセスに対応す

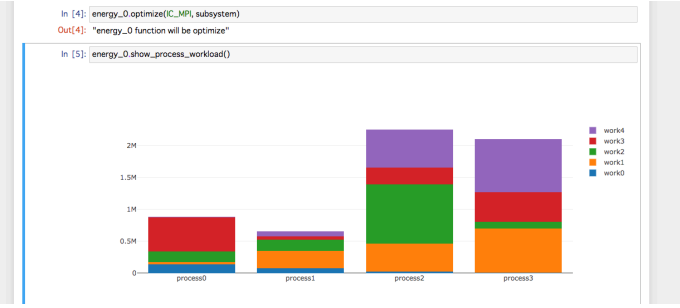


**図 7** DSL での MPI 挿入と仕事量確認

る．色分けされた部分の大きさが各プロセスが持つタスク一つの仕事量に相当する．この補助情報により，プロセス 0 とプロセス 1 のタスクの総仕事量が小さく，プロセス 2 とプロセス 3 のタスクの総仕事量が大きいことが視覚的に理解できる．

ユーザはプロセス 2 と 3 から一部のタスクをプロセス 0 と 1 に移動させる命令を行う．

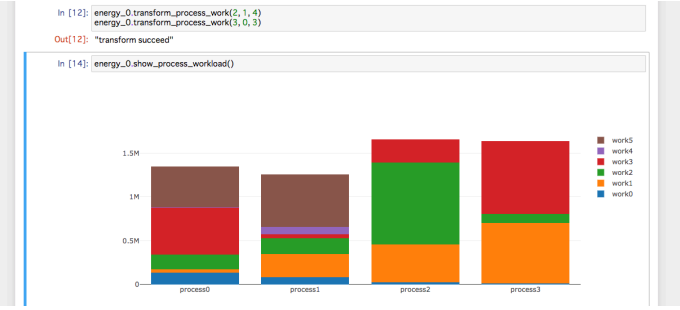


**図 8** DSL での負荷分散

図 8 のように，負荷分散命令後，図 7 と同様にプロセスの仕事量を確認できる．このグラフにより，仕事量の平均化が図られていることが確認できる．

以上が対話的最適化の一連の流れである．仕事量などは数値としても取得できるので，Ruby でスクリプトを記述することで，ユーザは独自のスケジューラを構築することができる．例えば，単純にタスクの大きさを昇順に並べてプロセスに割り振る場合は以下の関数を定義して自動化できる．

```
def scheduler_MPI(func)
  tasks = func.tasks.sortBy(0, UPPER)
  func.parallel(MPI, tasks)
end
```

この関数は式の定義された func を引数に取る．func 定義時の最初の引数を入力データのサイズで昇順に並べ替え，その引数のループに対するインデックス（tasks）を MPI で並列化するものである．より複雑なケースでは，ループを二つに分解しての並列化も行える．

```
def scheduler_complex_parallel(func)
  func1, func2 = func.divide(func.arg[0].
    lengthOf("child_array") > 1000)
  func.sequential(func1, func2)
  inter_func1 = function("inter_calc").at
    ("sigma").at(func1)
  inter_func1.parallel(MPI)
  func2.parallel(MPI)
end
```

この例では粗粒度部分の計算量の大小で二分割し，MPI を適用するループ部分を変更している．まず，func を引数に取り，func 定義時の最初の引数に対応する入力データがもつ，より細粒度の配列の大きさを元に閾値で分割する．sequential メソッドにより，分割された関数は逐次実行される．先に計算される方の関数では，子となる計算の Σ を計算する部分を inter_func1 で定義し，ここに MPI のプラグマを挿入する．後に計算される方の関数では，直接そこに MPI のプラグマを挿入する．これによる負荷分散の流れは図 9 のようになる．

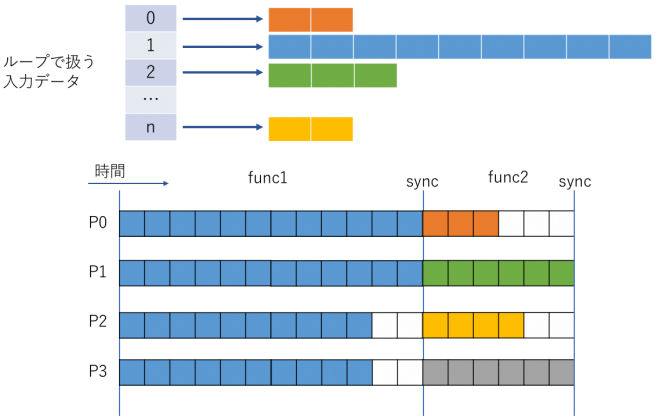


**図 9** 複雑なケースでの負荷分散の流れ

全てのプロセスが，入力データの中でも巨大な配列要素を対象に並列計算した後に，小さい配列要素を割り振って計算を行う．これにより，どれか一つのプロセスのタスクのみが重い状況を回避している．

## 4. まとめと今後の課題

本論文では，科学技術計算の一例として量子化学計算を示し，コードの最適化に関する問題点を挙げた．問題点は主に，入力データの巨大さや暗黙性・計算式の複雑さに由来しており，人的・自動的最適化の障壁となる．本研究では，従来の DSL を参考に科学技術計算に向けて DSL を設計し，Jupyter と併用することで対話的な最適化を行うアイデアを提案した．これにより人間とコンパイラの知識を相補的に活用して最適化を支援できる．人的な管理が困難な入力データや計算式の複雑性にはコンパイラの補助機能により可視化を伴ってアシストし，コードの最適化を行う．これを Jupyter セルで対話的に繰り返すことで，より高速なコードを簡便に開発できる．今後の課題の一つとして，DSL にどこまでの機能を持ち込むかの検討がある．現

在はユーザは Ruby の記法に則ってスケジューラなどを実装するが，DSL へその機能を取り込むことも一つの手である．また，生成される AST へユーザが独自に最適化を組み込むケースも考慮するべきである．本 DSL の開発者が実装した最適化のみを扱える現状よりも，最適化可能性を広げる検討を行いたい．それらの検討を経て，DSL の実装を完成させると共に，性能評価のための実験を行う．実験には，量子化学計算の FMO 法を用いた計算と既存アプリケーションとの比較を考えている．

## 参考文献

[1] Arie Van Deursen, Paul Klint, and Joost Visser.: Domain-specific languages: An annotated bibliography, *Sigplan Notices*, Vol. 35, No. 6(2000), pp.26-36.

[2] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe.: Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines, In *ACM SIGPLAN Notices*, Vol. 48, No. 6(2013), pp. 519-530.

[3] J. Yang and Q. He.: Scheduling Parallel Computations by Work Stealing: A Survey, *Int' l J. Parallel Programming*, pp.1-25, 2017.

[4] E. Mohr, D. A. Kranz and R. H. Halstead.: Lazy task creation: a technique for increasing the granularity of parallel programs, in *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 264-280, July 1991.

[5] Jeremy Gibbons and Nicolas Wu.: Folding domain-specific languages: deep and shallow embeddings (functional Pearl), *SIGPLAN* Not. 49, 9 (September 2014), 339-347.

[6] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand.: Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines, *SIGGRAPH*, 2012.

[7] Jonathan Ragan-Kelley.: Decoupling Algorithms from the Organization of Computation for High Performance Image Processing: The design and implementation of the Halide language and compiler, Ph.D. dissertation, MIT, May 2014.

[8] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian.: Automatically Scheduling Halide Image Processing Pipelines, *SIGGRAPH*, 2016.

[9] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley.: Differentiable Programming for Image Processing and Deep Learning in Halide, *SIGGRAPH*, 2018.

[10] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. PolyMage: Automatic Optimization for Image Processing Pipelines. *ASPLOS*, Mar 2015.

[11] Abhinav Jangda and Uday Bondhugula.: An Effective Fusion and Tile Size Model for Optimizing Image Processing Pipelines, *Proceedings of ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, Vo. sendorf, Austria, Feb 2018 (PPoPP' 18), 15 pages.

[12] Michel Steuwer, Toomas Remmelg, and Christophe Dubach.: Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation, *International Symposium on Code Generation and Optimization, CGO*, 2017.