2022 年度 北見工業大学 工学部 地域未来デザイン工学科
情報デザイン・コミュニケーション工学コース
学士論文

# 数値解析のための新規プログラミング言語における並列計算の実装

数理波動システム研究室
著者　中野太智
指導教員　杉坂純一郎

# 目次

# 論文要旨

Aqualis は数値解析プログラミングのために開発した新規プログラミング言語である。Aqualis は、ユーザーが F#で関数を定義することで命令文を記述し、自動的に Fortran のソースコードを生成する。これにより、コードの可読性を保ちつつ、プログラム処理のパフォーマンスの維持が可能になった。しかし、Aqualis には並列計算を行う手段がなく、計算時間の長さに問題があった。

そこで、本研究では並列プログラミングを可能にするために、OpenMP・OpenACC のディレクティブを挿入するためのメソッドを追加した。これにより、ユーザーはプログラムの並列化が容易になり、計算時間の短縮が可能となった。

# 第 1 章

# 序論

## 1.1 研究背景

電磁界解析は、電波を扱う機器の設計において重要である。電磁界のすべての現象は、マクスウェル方程式で表されるが、容易に解くことはできない。計算機を利用した、様々な手法によって数値電磁界解析が実現されている。その手法の中でも有限差分時間領域 (FDTD) 法、有限要素法、モーメント法は、よく利用されている。また、より効率的な解法や特定の問題に対して高速もしくは高精度で解析可能な、新たな解法の提案も行われている。そのような研究では、ソースコードを記述し、プログラムを自作する必要がある。Aqualis[1] は、この作業を最適化するシステムである。

Aqualis の役割を説明するために、まず数値解析プログラミングの最適化をする上で必要な要素について述べる。ここでは、実行パフォーマンスとコーディングの 2 つの観点から必要な要素について記述する。

実行時において、計算速度が速いこととメモリ消費が少ないことが求められる。FDTD 法や有限要素法を利用した数値電磁界解析プログラムを実行するとき、大量の計算が必要である。そのため、この 2 つは数値解析では重要な要素となる。

コーディング時においては、可読性や保守性が高いことが必要である。簡潔で読みやすく、後の変更や拡張が容易であれば、類似の計算を 1 からプログラミングする必要がなく、他プログラムへの転用も素早く行うことができる。

実行パフォーマンスの観点から見ると、Fortran や C 言語はきわめて高速に動作することから、数値解析プログラミングでも使われている。しかし、コーディング時の観点から見れば、この言語が必ずしも最適ではない。Fortran や C 言語は、高速に動作するが文法がやや複雑であり、可読性・保守性は低い。可読性や保守性を高めるためには、プログラムの全体構造を把握しやすく、ソースコードを簡潔に記述できることが必要である。F#や Scala,Python といった関数型・オブジェクト指向型言語は、その要件を満たしているが、実行パフォーマンスの面で劣る。関数型言語の可読性・保守性については 3.5 節で説明する。これら 2 つの観点から挙げた条件をすべて満たすことは困難である。これを解決するため、Aqualis は可読性は低いがパフォーマンスを優先したコードを自動生成している。

Aqualis とは、可読性・保守性の高い Aqualis のコードから、実行パフォーマンスの高い Fortran のソースコードを自動生成するシステムである。図 1.1 は、Aqualis が担う役割を表した模式図である。ユーザーが記述したコードを Aqualis が読み込み、Fortran もしくは C 言語のソースコードに変換する。Aqualis の実行には、可読性・保守性の高いコーディングが可能である言語として先述した、F#[2] が用いられる。ソースコードを自動生成しプログラミングを行うユーザーを補助するシステムは以前から行われてきた [3][4][5]。他の研究と Aqualis の相違点は、Aqualis 自身がプログラミング言語としてみなすことができる点である。Aqualis は F#のライブラリであるが、F#の関数型プログラミングの特徴を活かし、見かけ上のプログラミング言語として扱えるようになっている。以降の第 3 章でその原理について述べる。

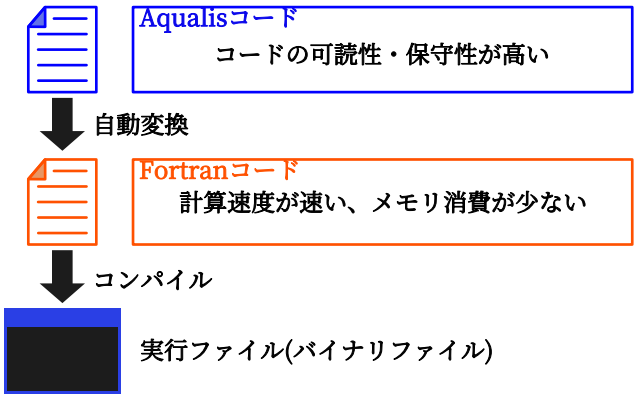


図 1.1　Aqualis の役割。ユーザーが記述した Aqualis コードを Fortran コードに自動変換し、パフォーマンスの優れたプログラムを実行する。

## 1.2　計算量の増大

このような数値解析を行う場合、計算量は膨大になる。例えば、回折現象のシミュレーションを行うとする。観測点 1 点の回折波の計算には、開口面の面積分が必要であるが、開口面を $N$ 点のサンプリング点を取ると、$O(N)$ の演算量が必要となる。また観測面のサンプリング点を同じように $N$ 点取ると、演算量は $O(N^2)$ 必要になる。

同じ観測面上の計算でも異なる観測点であれば、回折波は独立に計算できる。そのため、処理の並列化が可能である。計算時間の短縮には処理の並列化が有効であるが、Aqualis に並列化機能は未実装であった。

## 1.3　目的

本研究の目的は、Aqualis で生成された数値解析プログラムの実行時間の短縮である。数値解析プログラムの例として、今回は回折現象のシミュレーションプログラムを実行する。ここで実行するプログラムは、点光源から、円開口を通した観測面の回折波を計算するものである。このときの計算時間を短縮することで目的が達成されたこととする。

## 1.4　本論文の構成

本論文では、2 章で F#の記法について簡単に述べる。3 章では、Aqualis の基本文法、特徴について述べる。4 章では、Aqualis への並列プログラミングの実装の詳細を述べる。5 章では、並列化した結果、回析現象のシミュレーションプログラムの計算時間を比較し、考察を行う。6 章では、本論文のまとめ、結論を述べる。

# 第 2 章

# F#

この章では、F#の文法について示す。F#は 2005 年にマイクロソフト社が開発した言語である。関数型プログラミングとオブジェクト指向プログラミングを内包したマルチパラダイム言語である。

## 2.1 関数定義

以下のコードは、引数 x に 1 を加えた値を返す関数を定義している。

```
1   let f x = x + 1
```

等号の右側は関数 f に引数 x を適用したときの値を示している。以下のコード

```
1   let f = fun x -> x + 1
```

は、関数 f そのものの定義を記述した例である。前述のコードと同じように、引数 x に 1 を加えた値を返す関数を定義している。等号の右側はラムダ式という。

引数に関数を指定することもできる。以下は、引数に関数を指定し、定義した関数を使用した例である。

```
1   let iter (x:int) (y:int->unit) =
2       for i in 1..x do y i
3
4   iter 4 (fun i -> printfn "%d" i)
5
6   iter 4 <| fun i ->
7       printfn "%d" i
```

”x:int” は引数 x の型が整数型であることを表し、”y:int->unit” は y の型が整数を受け取り何も返さない関数を表す。4 行目と 6・7 行目で定義した関数を使用している。どちらも実行すると変数の値である 1,2,3,4 を出力する。4 行目の”fun i -> printfn "%d" i” は、第 2 引数を表し、整数 x を受け取り、その値を画面出力する関数である。6・7 行目では、逆順パイプライン関数”<|” を用いて、関数を第 2 引数として記述している。

## 2.2 演算子の定義

F#では、新たな演算子の定義や既存の演算子のオーバーロードが可能である。Aqualis で用いられている代入演算子”<==” は、この特徴を利用している。y の値を x に代入するコードを生成する演算子

```
1   let (<==) x y =
```

```
2        //代入するコードを生成する処理
```

を定義すると、

```
1   a <== 1
```

と記述することで、変数 a に 1 を代入するコードを出力することができる。

## 2.3 判別共用体

判別共用体とは、F#で使用されるデータ型の一種である。型の和集合といわれ、定義したパターンのうちいずれかのデータ型を取ることができる。例えば、

```
1   type shape =
2       |rectangle of width : double * height : double
3       |circle of radius : float
```

のように判別共用体 shape を定義することができる。この判別共用体は、長方形の縦と横の長さ (double) もしくは円の半径 (float) のいずれかの情報を持つ。判別共用体は、数値や変数、数式のコードなど様々な型を指定できる。

# 第 3 章

# Aqualis

この章では、Aqualis の基本的な文法、よく使うメソッドを紹介し、Aqualis 上でのコーディング時に行う関数の定義について述べる。

## 3.1 変数宣言

変数の宣言には、ラムダ式を用いて以下のように記述する。

```
1   ch.i <| fun x ->
2       x <== 1
3       print.c x
```

これは、整数 x を宣言したときの例である。”ch.i” とあるが、実数 (倍精度小数点数) を宣言するときは”ch.d”、複素数のときは”ch.z” に変わる。この例では、”x <== 1” で x に 1 を代入し、”print.c x” で変数 x の画面出力をしている。x の使用範囲はインデントで指定できる。上記のコードでは 2 行目から 3 行目が x の使用可能な範囲である。この性質は、プログラムのメモリ消費をおさえることに寄与している。

```
1   ch.i <| fun x ->
2       x <== 1
3       print.c x
4   ch.i <| fun y ->
5       y <== 2
6       print.c x
```

上記は、Aqualis がメモリ消費を自動的に抑えるコードを生成するときの例である。上記のコードでは 2 つの整数を宣言されているが、両者が同時に使用されることはない。上記のコードを読み込み、Aqualis は以下のような 1 つのみの変数宣言を行うコードを生成する。

```
1   integer :: i_cache_001
2   i_cache_001 = 1
3   print *, i_cache_001
4   i_cache_001 = 2
5   print *, i_cache_001
```

”integer :: i_cache_001” は、整数型の変数 i_cache_001 を宣言する命令文である。”print *, …” は変数を画面出力する命令である。変数宣言は 1 行目の部分のみで、変数 i_cache_001 が使い回されている。変数の使用範囲

を明示することで、無駄な変数宣言を防ぎ、メモリ消費を抑えることができる。

Fortran に出力されている整数型の変数名は i_cache_001 だが、もう一つ同じ型の変数を生成すると i_cache_002 となり、末尾の数字が 1 増える。Aqualis は、末尾の数字 001 から 002, 003, …という順でコードの中で今使われていない変数を探索し、ある場合はその変数を使い、ない場合は新たに変数を生成する。これにより、無駄な変数を防いでいる。

## 3.2 四則演算

四則演算は、他のプログラミング言語と同じような記法で書くことができる。

```
1  z <== x + y
2  z <== x - y
3  z <== x * y
4  z <== x / y
```

代入の演算子は”<==” を使用する。

Aqualis では、変数は判別共同体 num0 が使われている。通常の四則演算では num0 型は使用できないが、既存の演算子をオーバーロードして num0 型の演算を可能にしている。x が num0 型、y が int(整数) 型のときのような異なる型同士の演算では、y を num0 型に変換した後、計算し num0 型が出力される。

## 3.3 条件分岐

条件分岐には以下のようなコードを用いる。

```
1  br.if1 (z .= x+y) <| fun () ->
2      print.t "z = x+y"
```

”z .= x+y” は条件式である。これが真となるとき、2 行目の命令を実行する。条件が偽となった時の処理を記述したいときは、以下のように”br.if2” を使用する。

```
1  br.if2 (z .= x+y)
2    <| fun () ->
3      print.t "z eq x+y"
4    <| fun () ->
5      print.t "z neq x+y"
```

これを実行すると、条件”z .= x+y” を満たすとき文字列”z eq x+y” が出力され、満たさないとき”z neq x+y” が出力される。条件式を複数追加するときは以下のように記述する。

```
1  br.branch <| fun b ->
2      b.IF (z .> x+y) <| fun () ->
3          print.t "z > x+y"
4      b.IF (z .= x+y) <| fun () ->
5          print.t "z = x+y"
6      b.EL <| fun () ->
7          print.t "z < x+y"
```

z>(x+y) のとき文字列"`z > x+y`" を出力、z>(x+y) ではなく z=x+y のとき文字列"`z = x+y`" を出力、どちらの条件も満たさないとき文字列"`z < x+y`" を出力する。

## 3.4 ループ

ループを使ったソースコードの例を以下に示す。

```
1  iter.range -5 5 <| fun i ->
2      print.c i
```

この例では、ループ変数 i が引数の-5 から 5 まで、つまり-5,-4,-3,...,5 と変化していく。ループ変数の変数名は、1 行目の"`<| fun i ->`" で指定している。これを実行すると、-5 から 5 までの 11 の整数が出力される。ループ変数が 1 から始まるループの場合、以下のような記述もできる。

```
1  iter.range 10 <| fun i ->
2      print.c i
```

この例では、変数 i は 1,2,3,...,10 と変化する。結果、実行すると 1 から 10 までの 10 の整数が出力されることになる。ループは以下のような記述もできる。

```
1  iter.list [x;y;z] <| fun v ->
2      print.c v
```

"`iter.list`" は、リストを基準にしたループである。この例では、変数 v にリストの要素 x,y,z がそれぞれのループで代入されて、ループが処理される。このコードを実行すると、変数 x,y,z の値が出力される。

## 3.5 関数群の定義

ここまで、Aqualis の基本的な文法について説明してきた。紹介してきたコード例には、ラムダ式を用いて関数を定義し、その関数を引数として使用したものがあった。Aqualis の命令群の多くは、ラムダ式と関数定義によって記述される。これが Aqualis の大きな特徴の一つである。

関数型プログラミングを使うメリットは、可読性が高く、プログラムの改修がし易いという点である。関数型を利用しない場合、各パーツが持つ役割が理解しにくいことがある。以下は、その例を示した Java のコードである。

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public class main {
5      public static void print_list(List<String> str_list) {
6          int i = 0;
7          while (i < str_list.size()) {
8              System.out.println(str_list.get(i));
9              i += 1;
10         }
11     }
12     public static void main(String[] args) {
13         List<String> str_list = new ArrayList<String>();
```

```
14          str_list.add("Hello");
15          str_list.add("World");
16          print_list(str_list);
17      }
18  }
```

これを実行すると、文字列の”Hello” と”World” が出力される。13 行目で文字列型を格納するリスト”str_list” を宣言し、14・15 行目で文字列”Hello”,”World” をリストに追加している。16 行目でリストの要素を出力するメソッド”print_list” を実行、5 行目から 11 行目にそのメソッドの定義が書かれている。”print_list” がどのような役割を持っているのか、この状態では 5 行目から 11 行目の定義内容を確認するまでは理解しにくい。また、”str_list” の要素も複数行にわたって記述されるため、メソッド”print_list” の入力内容も確認しにくい。同じ動作をするプログラムを Aqualis を用いると、以下のように簡潔に記述できる。

```
1  let str_list = ["Hello"; "World"]
2
3  iter.list str_list <| fun s ->
4      print.t s
```

1 行目が文字列型を格納するリスト”str_list” の定義である。この”str_list” をメソッド”iter.list” の第 1 引数として用いて、ループ処理を行っている。この記法では、”str_list” の要素が瞬時に理解でき、コーディング量も短縮できる。”str_list” の要素を変更したいときも瞬時に対応できる。このように、Aqualis で関数定義を用いたプログラミングをすることによって、可読性・保守性を高めている。

# 第 4 章

# 並列計算の実装

Aqualis に並列プログラミングのための機能を実装するため、並列プログラミングのために開発された API である、OpenMP と OpenACC のディレクティブを挿入するメソッドを追加した。

## 4.1 OpenMP

OpenMP とは、Fortran・C/C++ から利用できるマルチコア CPU を利用した並列プログラミングのための API である。図 4.1 は、OpenMP を用いた並列処理を表した模式図である。図 4.1 のように CPU 内のコアもしくはスレッドを利用して同時に複数の計算結果を出力できる。OpenMP は、ディレクティブベースのプログラミング手法を

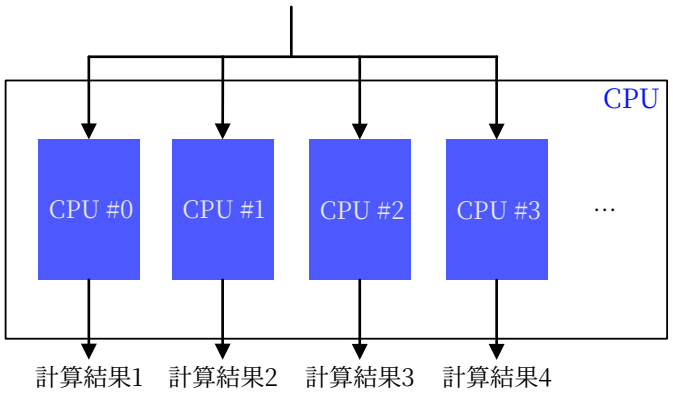


図 4.1　OpenMP での並列処理のイメージ。並列領域にある処理を各スレッドに自動的に割り当てていく。

採用しているため、ソース行に指示文を挿入することで並列化が有効になる。以下のような Fortran のコードがあるとする。

```
1   program omp_serial
2       integer :: i
3       do i = 1, 10
4           print *, i
5       end do
6   end program omp_serial
```

”do … end do” の部分は、ループ処理を行う命令でループ変数 i が 1,2,3,…,8,9,10 と変化し、”print *, i” で変数

i の値を出力させるようになっている。このプログラムのループを並列処理で行うには以下のように、ディレクティブを追加する。

```
1  program omp_parallel
2      integer :: i
3      !$omp prallel do
4      do i = 1, 10
5          print *, i
6      end do
7      !$omp end prallel do
8  end program omp_parallel
```

”!$omp …” と書かれた行が OpenMP の指示文である。このように OpenMP は既存のコードに OpenMP のディレクティブを挿入することで、並列化を行うことができる。

## 4.2　OpenMP の実装

### 4.2.1　ディレクティブの挿入

OpenMP のディレクティブを挿入するメソッドを Aqualis に追加する。”omp.parallelize” を利用することでループの並列化が可能になった。先述の並列化した後のコードを Aqualis で記述すると以下のようになる。

```
1  omp.parallelize <| fun () ->
2      iter.num 10 <| fun i ->
3          print.c i
```

”iter.num 10” は 10 回ループを意味し、先程の”do … end do” のコードを生成する。omp クラスの parallelize メソッドは、OpenMP の指示文である”!$omp …” を挿入する役割を持っている。parallelize メソッドの定義コードを以下に示す。

```
1  static member parallelize code =
2      match p.lang with
3        |F ->
4          let p = p.param
5          p.isOmpUsed <- true
6          p.switch_parmode(true)
7          p.cclose()
8          p.popen()
9          code()
10         p.pclose()
11         p.switch_parmode(false)
12         p.copen()
13         let rec printplist list n counter (s:string) =
14             match (list,counter) with
15               |[],_ ->
16                 if (counter>0) && n=0 then
```

```
17                      p.codewrite("!$omp parallel do private("+s+")"+"\n")
18                  elif (counter>0) then
19                      p.codewrite("!$omp private("+s+")"+"\n")
20                  else
21                      ()
22                |pp::lst,7 ->
23                  if n=0 then
24                      p.codewrite("!$omp parallel do private("+s+") &"+"\n")
25                  else
26                      p.codewrite("!$omp private("+s+") &"+"\n")
27                  printplist lst 1 1 pp
28                |p::lst,_ ->
29                  printplist lst n (counter+1) (s+(if counter=0 then "" else ",")+p)
30            printplist p.pvlist 0 0 ""
31            //一時ファイルの内容を書き込み
32            p.cwrite(p.readpartext())
33            p.codewrite("!$omp end parallel do\n")
34            p.clearpv()
35          |C99 ->
36            let p = p.param
37            p.isOmpUsed <- true
38            p.switch_parmode(true)
39            p.cclose()
40            p.popen()
41            code()
42            p.pclose()
43            p.switch_parmode(false)
44            p.copen()
45            if p.pvlist.Length>0 then
46                let s = String.Join(",", p.pvlist)
47                p.codewrite("#pragma omp parallel for private("+s+")\n")
48            else
49                p.codewrite("#pragma omp parallel for\n")
50            //一時ファイルの内容を書き込み
51            p.cwrite(p.readpartext())
52            p.clearpv()
53          |_ ->
54            ()
```

2 行目の”`match p.lang with …`” は、出力するソースファイルの言語の場合分けを表している。3 行目の”`|F ->`” の後の命令は Fortran のソースファイルを出力するとき、35 行目の”`|C99 ->`” の後の命令は C 言語のソースファイルを出力するときを表している。ここでは、Fortran を出力するときで説明する。parallelisze メソッドには”`code`” という名前の引数があるが、これはメソッドの使用時にラムダ式で定義された関数である。この関数の処理はコー

ドの 9 行目で実行されている。その後の 13 行目から 26 行目で定義され、27 行目で利用されている”printplist”がディレクティブを書き込む処理を制御している関数である。19 行目、24 行目、26 行目、33 行目などで登場する”p.codewrite(…)” は、括弧の中の文字列をソースファイルに書き込む命令文である。ここの括弧の中に”!$omp”で始まる文字列書かれていれば OpenMP のディレクティブをソースファイルに書き込んでいる。”printlist” は、後に説明するプライベート変数を記録したリスト”p.pvlist” を受け取り、ディレクティブの書き込みを行っている。並列化部分の処理を記述する時、ソースファイルに直接記述せずに一度別ファイルに書き込んでから、ディレクティブ書き込み処理を行い、記述する。6 行目と 11 行目の”p.switch_parmode(…)” は並列領域モードの切り替えを行うメソッドである。並列領域モードの時、コードの書き込みは並列処理用の一時ファイルに行われる。一時ファイルに書き込まれたコードは 32 行目で本来のソースファイルに書き込む。この処理を行う理由は 4.2.4 で説明する。

### 4.2.2　プライベート変数

並列プログラミングで注意すべき要素の一つとしてプライベート変数が挙げられる。プライベート変数とは、CPU のスレッドの間でデータの共有を行わない変数のことである。Aqualis で以下のようなコードを記述したとする。

```
1   ch.i1 1000 <| fun a ->
2       ch.i <| fun w ->
3           omp.parallelize <| fun () ->
4               iter.num a.size1 <| fun i ->
5                   w <== i
6                   a[w] <== i
7           ch.i <| fun sum ->
8               sum <== 0
9               iter.num a.size1 <| fun i ->
10                  sum <== a[i]
11              print.c sum
```

3 行目がないとき、つまり逐次処理で行うとき、上のコードは

$$\sum_{i=1}^{1000} i \tag{4.1}$$

の計算を実行するプログラムとなる。しかし、このコードで生成された Fortran のソースコードをコンパイルして実行すると、計算結果である 500500 が出力されず、意図した結果とならない恐れがある。これは並列化されたスレッドの処理で、同じメモリ領域に変数 w のデータアクセスが行われているためである。図 4.2 はこのときのデータアクセスを表した模式図である。同じメモリアドレスに各スレッドがアクセスを行い、変数の値を変更している。このため、スレッドの処理中に他のスレッドが値の変更を行うため、本来とは違う結果が出力されてしまう。

これを解決するためには変数 w をプライベート変数に指定しそれぞれのスレッドが別々のメモリにアクセスする必要がある。図 4.3 はプライベート変数を指定したときのデータアクセスの模式図である。プライベート変数は、スレッド数だけメモリ領域が設けられ、各スレッドが別々のメモリ領域にアクセスが行われている。

プライベート変数を有効にするため、プライベート変数を宣言するメソッドの追加を行った。以下はプライベート変数を宣言したときのソースコードである。

```
1   ch.i1 1000 <| fun a ->
2       ch.private_i <| fun w ->
3           omp.parallelize <| fun () ->
```

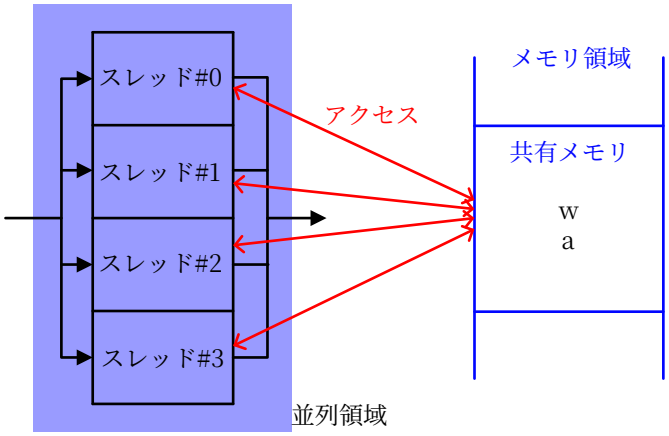


図 4.2　すべての変数が共有変数であるため、異なるスレッドが同じメモリアドレスにアクセスしている。

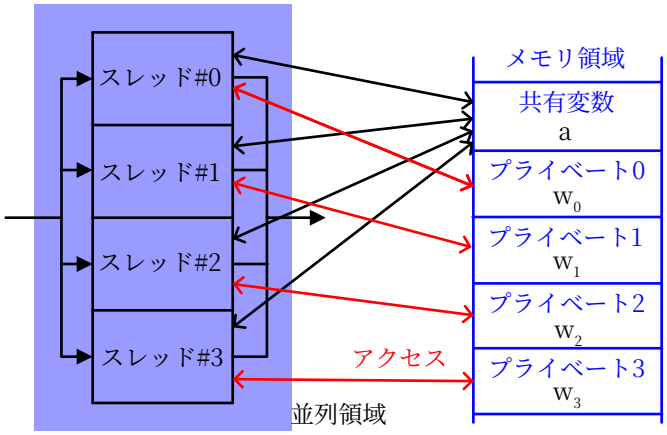


図 4.3　w をプライベート変数に指定したときの模式図。変数 w がスレッドの数だけ生成される。

```
4               iter.num a.size1 <| fun i ->
5                   w <== i
6                   a[w] <== i
7           ch.i <| fun sum ->
8               sum <== 0
9               iter.num a.size1 <| fun i ->
10                  sum <== a[i]
11              print.c sum
```

”ch.private_i” がプライベート変数を宣言するメソッドである。このメソッドを使用することで、前述の問題を回避可能になった。private_i メソッドの定義コードを以下に示す。

```
1   static member private_i code =
2       ch.i <| fun i1 ->
3           p.param.pvreg i1.name
```

```
4           code(i1)
```

2 行目のメソッド”ch.i” で、通常の変数宣言を行い、3 行目の”p.param.pvreg i1.name” でプライベート変数リストに変数名を文字列型で登録している。OpenMP のディレクティブを挿入するメソッドで、この登録された文字列のリストを読み取り、プライベート変数を指定している。

### 4.2.3　スレッド番号の取得

並列処理の実行中、処理を行っている CPU のスレッド番号がわかるようにしている。以下は、並列化領域内でスレッド番号を取得し、画面出力をするソースコード例である。

```
1   omp.parallelize <| fun () ->
2       print.c omp.thread_num
```

”omp.thread_num” がスレッド番号を取得するメソッドである。

### 4.2.4　ディレクティブを挿入するプロセス

ここでは、OpenMP の指示文を挿入する仕組みについて説明する。ここまで OpenMP を利用するためのメソッドの説明をしてきた。先述の通り、Aqualis で OpenMP のプライベート変数を指定できるようになった。しかし、ループ変数をプライベート変数に指定しなければ実行結果に問題が起こることがある。多重ループを実行する際、外側のループを並列化させると、内側のループ変数は共有変数であるため、正しいループが行われない可能性がある。そのため、ループ変数をプライベート変数に指定する必要がある。Aqualis で OpenMP を利用した並列化を行う場合、自動的にループ変数を指定する仕様になっている。

この仕様にするために、Aqualis では少し特殊なプロセスでディレクティブを挿入している。プライベート変数の指定は、並列処理の上の行に記述する。F#コンパイラが omp.parallelize メソッドをコード解析した直後にディレクティブを書き込む処理を行うとき、その時点でループ変数はコード解析の中で登場していない。ループ変数がわからない状態では、OpenMP のディレクティブでプライベート変数を指定できない。そのため単純なプロセスでは、ループ変数をプライベート変数に指定できない。並列領域内のコード解析が終了した時点で、ループ変数を読み込み、ディレクティブの書き込みを行う必要がある。

Aqualis では、ループ変数をプライベート変数に指定するために、並列処理の部分を一時的に別ファイルに書き込んでから、OpenMP のディレクティブを記述している。図 4.4 は、そのときのイメージを表したものである。F#コンパイラが omp.parallelize を読み込んだ時、並列処理を書き込む一時ファイルとプライベート変数リストが生成される。その後、並列領域内のコード解析を実行し、一時ファイルに Fortran もしくは C 言語のソースコードを書き込む。それと同時にプライベート変数リストにループ変数を登録する。並列領域内のコード解析が終了した後、ディレクティブを書き込み、本来書き込むべきソースファイルに一時ファイル内のコードをコピーする。このような処理にすることで、一時ファイルの書き込み処理を行った後で、プライベート変数リストが更新された後のディレクティブを書き込むことができる。

## 4.3　OpenACC

OpenACC とは、アクセラレータデバイスでの並列プログラミングを行うための API である。OpenACC は GPU や DSP などのデバイスで並列化を行うことができるが、今回は GPU での並列化を想定して開発を行った。GPU は CPU と比べコア数が多く同時に大量の処理を実行することに適している。CPU が担う計算を GPU で行うことで、計算時間の短縮を図ることができる。図 4.5 は、OpenACC が処理を CPU から GPU に移す様子を表している。OpenACC では、OpenMP と同様にソースに指示文を追加することで並列処理を実装できる。以下は、OpenACC

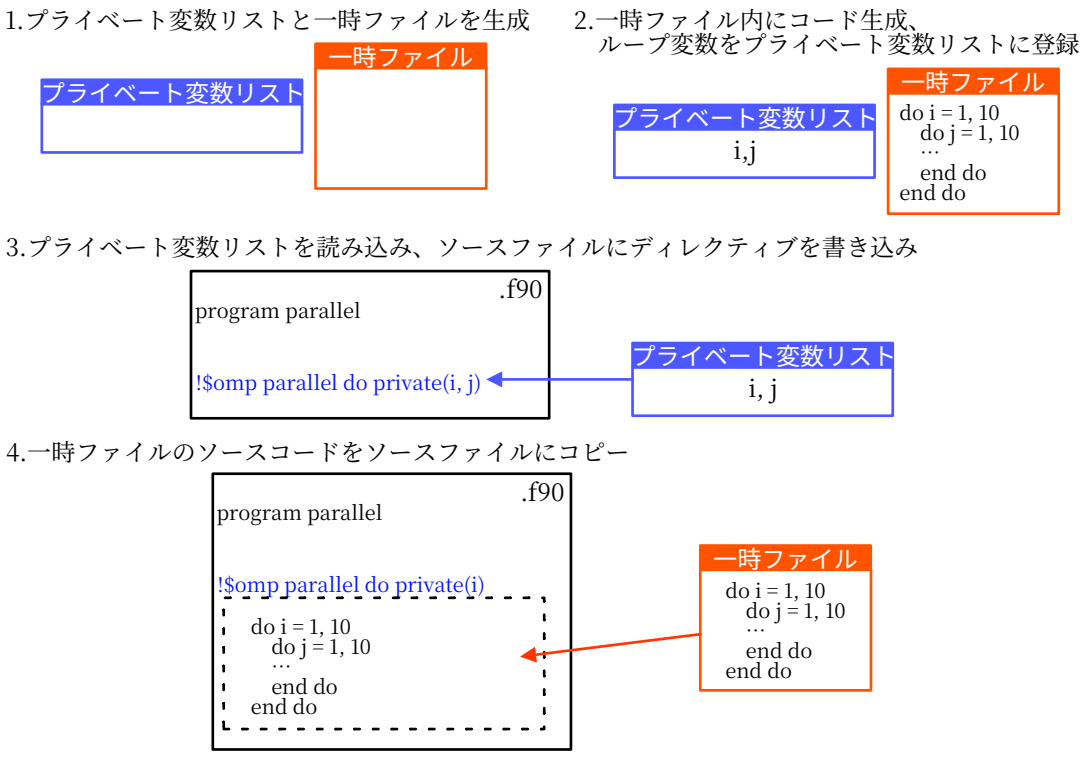


図 4.4 OpenMP のディレクティブを書き込むプロセス

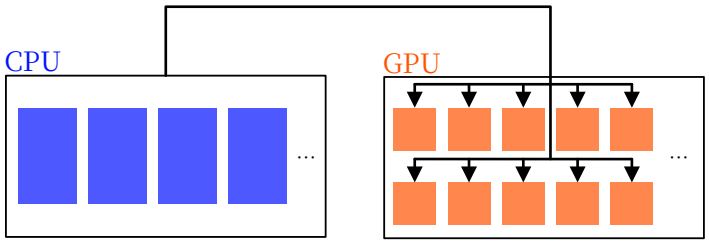


図 4.5 OpenACC では、プログラム処理中で並列領域に到達するとそれまで CPU が行っていた処理を GPU に渡し、GPU の複数の演算コアで処理する

を利用する前の Fortran のソースコードである。

```
1  program oacc_parallel
2      integer :: i
3      real, dimension(1000) :: a
4      do i = 1, size(a)
5          a(i) = i / 1d3
```

```
6       end do
7   end program
```

このコードを OpenACC で並列化すると以下のようになる。

```
1   program oacc_parallel
2       integer :: i
3       real, dimension(1000) :: a
4       !$acc kernels
5       do i = 1, size(a)
6           a(i) = i / 1d3
7       end do
8       !$acc end kernels
9   end program
```

”!$acc kernels” と”!$acc end kernels” と書かれている行がこのソースコードでの OpenACC の指示行である。この 2 つの指示文で囲まれている部分が GPU での処理となることを意味する。このようにディレクティブを挿入することで、GPU で計算処理を行うことができるようになる。

## 4.4　CPU・GPU 間のデータ転送

アクセラレータデバイスで並列プログラミングを行う場合、注意すべきなのはデータの転送である。CPU メモリと GPU メモリは物理的・論理的に分離されている [6]。図 4.6 は、メモリが分離されている状態を表したものである。そのため、必要な変数データを CPU から GPU もしくは、GPU から CPU に転送する必要がある。この転送す

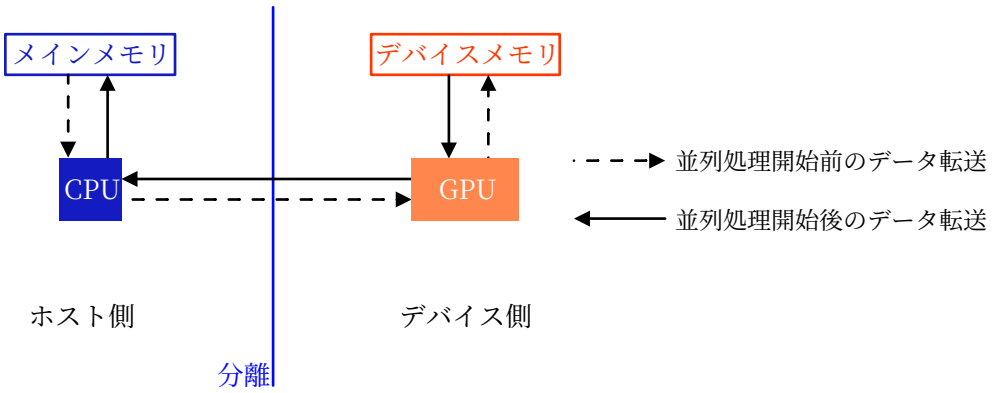


図 4.6　CPU メモリと GPU メモリの接続の状態。並列処理を開始する前と後で必要なデータを転送する必要がある。

るデータが多いほど、プログラムの処理時間が長くなってしまうため、計算時間の短縮を目的とする場合、注意が必要である。

## 4.5 OpenACC の実装

### 4.5.1 OpenACC のディレクティブの挿入

OpenACC のディレクティブを挿入するメソッド oacc.parallelize を追加した。先述した OpenACC の並列化したコードを Aqualis で記述すると以下のようになる。

```
1  ch.d1 1000 <| fun a ->
2      oacc.parallelize <| fun () ->
3          iter.num a.size1 <| fun i ->
4              a[i] <== 1 / 1e3
```

以下に oacc クラスの parallelize メソッドの定義コードを示す。

```
1  static member parallelize code =
2      match p.lang with
3        |F ->
4          let p = p.param
5          p.isOaccUsed <- true
6          p.switch_parmode(true)
7          if p.copy_in_vlist.Length>0 then
8              let instr  = String.Join(",", p.copy_in_vlist)
9              if p.copy_out_vlist.Length>0 then
10                 let outstr = String.Join(",", p.copy_out_vlist)
11                 p.codewrite("!$acc data copyin("+instr+") copyout("+outstr+")\n")
12                 p.codewrite("!$acc kernels"+"\n")
13                 code()
14                 p.codewrite("!$acc end kernels"+"\n")
15                 p.codewrite("!$acc end data"+"\n")
16             else
17                 p.codewrite("!$acc data copyin("+instr+")\n")
18                 p.codewrite("!$acc kernels"+"\n")
19                 code()
20                 p.codewrite("!$acc end kernels"+"\n")
21                 p.codewrite("!$acc end data"+"\n")
22         elif p.copy_out_vlist.Length>0 then
23             let outstr = String.Join(",", p.copy_out_vlist)
24             p.codewrite("!$acc data copyout("+outstr+")\n")
25             p.codewrite("!$acc kernels"+"\n")
26             code()
27             p.codewrite("!$acc end kernels"+"\n")
28             p.codewrite("!$acc end data"+"\n")
29         else
30             p.codewrite("!$acc kernels"+"\n")
```

```
31              code()
32              p.codewrite("!$acc end kernels\n")
33            p.switch_parmode(false)
34          |C99 ->
35            let p = p.param
36            p.isOaccUsed <- true
37            p.switch_parmode(true)
38            if p.copy_in_vlist.Length>0 then
39                let instr  = String.Join(",", p.copy_in_vlist)
40                if p.copy_out_vlist.Length>0 then
41                    let outstr = String.Join(",", p.copy_out_vlist)
42                    p.codewrite("#pragma acc data copyin("+instr+") copyout("+outstr+")\n")
43                    p.codewrite("{"+"\n")
44                    p.codewrite("#pragma acc kernels"+"\n")
45                    p.codewrite("{"+"\n")
46                    code()
47                    p.codewrite("}"+"\n")
48                    p.codewrite("}"+"\n")
49                else
50                    p.codewrite("#pragma acc data copyin("+instr+")\n")
51                    p.codewrite("{"+"\n")
52                    p.codewrite("#pragma acc kernels"+"\n")
53                    p.codewrite("{"+"\n")
54                    code()
55                    p.codewrite("}"+"\n")
56                    p.codewrite("}"+"\n")
57            elif p.copy_out_vlist.Length>0 then
58                let outstr = String.Join(",", p.copy_out_vlist)
59                p.codewrite("#pragma acc data copyout("+outstr+")\n")
60                p.codewrite("{"+"\n")
61                p.codewrite("#pragma acc kernels"+"\n")
62                p.codewrite("{"+"\n")
63                code()
64                p.codewrite("}"+"\n")
65                p.codewrite("}"+"\n")
66            else
67                p.codewrite("#pragma acc kernels"+"\n")
68                p.codewrite("{"+"\n")
69                code()
70                p.codewrite("}"+"\n")
71            p.switch_parmode(false)
72          |_ ->
73            let p = p.param
```

```
74          Console.WriteLine("Error : この言語では並列化を実行できません")
75          p.codewrite("Error : この言語では並列化を実行できません")
```

ここでは、3〜33 行目の Fortran コードを出力するときについて説明する。7 行目から 32 行目までの処理が、ディレクティブを挿入する処理である。”if … else …” や”if … elif …” は、F#の条件分岐の構文であるが、データを転送する変数名を記録したリスト”copy_in_vlist”、”copy_out_vlist” の要素数で条件分岐をしている。要素数が 0 より大きければ、データ転送する変数を指定するディレクティブを書き込み、小さいつまりなければデータ転送する変数を指定するディレクティブを書き込まないように処理する。データ転送する変数を指定するディレクティブの書き込み処理は、11 行目、17 行目、24 行目にある。

### 4.5.2 データ転送にかかわる変数の宣言

CPU メモリと GPU メモリは論理的・物理的に分離されている。そのため、OpenACC ではデータ転送を行う必要がある。

OpenACC では、コンパイラが必要な変数を自動的に選択して転送することがある。しかし、GPU の転送時間がプログラムの処理時間に影響が出ることがある。そのため、計算時間の短縮においてはデータ転送に関わる変数の指定は重要である。そこで、CPU・GPU 間のデータ転送を行う変数を宣言するメソッドを追加した。

```
1  ch.copyin_i1 1024 <| fun a ->
2      ch.copyout_i1 1024 <| fun b ->
3          iter.num a.size1 <| fun i ->
4              a[i] <== i / 1e3
5          oacc.parallelize <| fun () ->
6              iter.num a.size1 <| fun i ->
7                  b[i] <== a[i] * 1e3
```

”ch.copyin i1” が CPU から GPU へデータ転送をする変数のメソッド、”ch.copyout i1” が GPU から CPU へデータ転送をする変数のメソッドである。これにより、CPU・GPU 間のデータ転送にかかわる変数を指定することができるようになった。それぞれのメソッドの定義コードを示す。

```
1  static member copyin_i1 (size1:num0) = fun code ->
2      match p.lang with
3        |F ->
4          ch.i1 size1 <| fun i ->
5              p.param.civreg (i.name+"(1:"+size1.name+")")
6              code i
7              p.param.rmciv (i.name+"(1:"+size1.name+")")
8        |C99 ->
9          ch.i1 size1 <| fun i ->
10             p.param.civreg (i.name+"[0:"+size1.name+"]")
11             code i
12             p.param.rmciv (i.name+"[0:"+size1.name+"]")
13       |_ ->
14         ()
```

```
1  static member copyout_i1 (size1:num0) = fun code ->
```

```
2       match p.lang with
3         |F ->
4           ch.i1 size1 <| fun i ->
5               p.param.covreg (i.name+"(1:"+size1.name+")")
6               code i
7               p.param.rmcov (i.name+"(1:"+size1.name+")")
8         |C99 ->
9           ch.i1 size1 <| fun i ->
10              p.param.covreg (i.name+"[0:"+size1.name+"]")
11              code i
12              p.param.rmcov (i.name+"[0:"+size1.name+"]")
13        |_ ->
14          ()
```

それぞれの 4 行目と 9 行目にあるメソッドは、通常の整数型配列の宣言を表している。それぞれの 5 行目と 10 行目で転送する変数名をリストに追加している。その後、他のコードを記述する処理を行った後、7 行目と 12 行目でその変数をリストから削除している。

それぞれのデータ転送をするタイミングについて図 4.7 に示す。並列領域の最初に CPU から GPU へ、並列領域

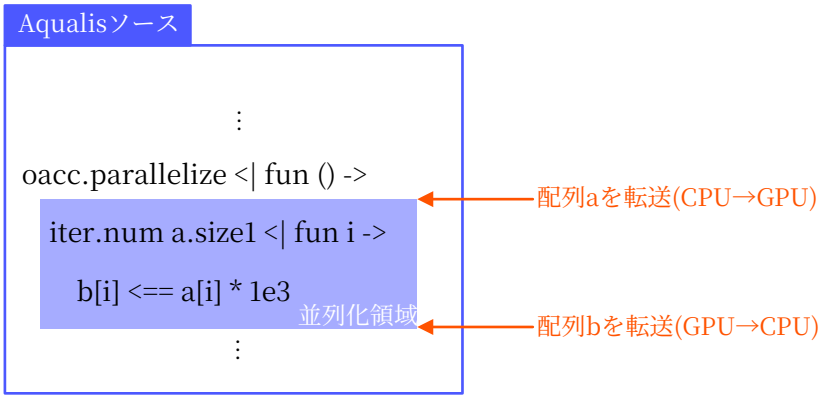


図 4.7　CPU・GPU 間で変数のデータ転送をするタイミング。CPU から GPU へ、並列領域の最後に GPU から CPU へデータ転送を行う。

の最後に GPU から CPU へデータ転送を行っている。

# 第 5 章

# 計算時間の比較

## 5.1　回折現象のシミュレーション

並列計算の導入によって計算時間が短縮されたかを確認するため、図 5.1 のような回折現象のシミュレーションを行った。開口の形は円形とし、観測面上の点を P とおく。開口面と観測面は、光源の波長を $\lambda$ としたとき、一辺が

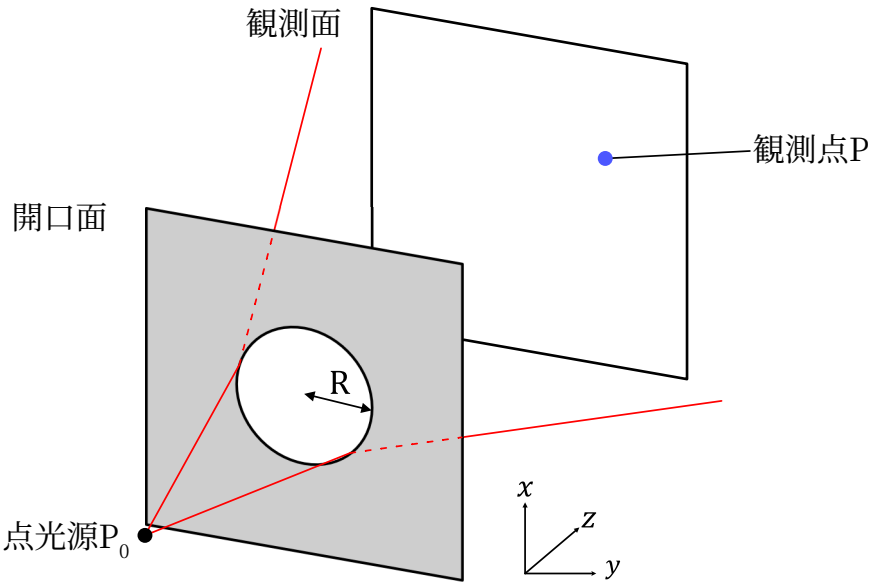


図 5.1　回折現象のシミュレーションの模式図。円形開口を透過した観測面の回折波を計算した。

$10\lambda$ の正方形とし、開口の半径 $R$ は $2\lambda$ とした。開口面と観測点に打つサンプリング点の数は、それぞれ縦に 201、横に 201 の 40401 点とした。

回折の計算には、フレネル・キルヒホッフ回折積分の計算式を利用する。図 5.2 は図 5.1 を $y$ 軸方向から見たときの模式図である。開口面上の任意の点 A を取るとき、$\mathrm{P}_0$ から A に向かう方向の単位ベクトルを $\hat{\boldsymbol{r}}_{\boldsymbol{0}}$、P から A に向かう方向の単位ベクトルを $\hat{\boldsymbol{r}}$ とする。線分 $\mathrm{P}_0\mathrm{A}$ の長さを $r_0$、線分 PA の長さを $r$ とおく。$z$ 軸方向の単位ベクトルを $\hat{\boldsymbol{n}}$ としたとき、$\hat{\boldsymbol{n}}$ と $\hat{\boldsymbol{r}}$ のなす角を $\Theta$、$\hat{\boldsymbol{n}}$ と $\hat{\boldsymbol{r}}_{\boldsymbol{0}}$ のなす角を $\Theta_0$ とする。

開口を閉曲面 S としたとき、観測点 P の回折波 $f(\mathrm{P})$ は以下のように求められる。

$$f(\mathrm{P}) = \frac{jAk}{4\pi} \int\int_S \frac{\exp[-jk(r_0 + r)]}{r_0 r} (\cos\Theta_0 - \cos\Theta) dS \tag{5.1}$$

これがフレネル・キルヒホッフ回折積分の式である。

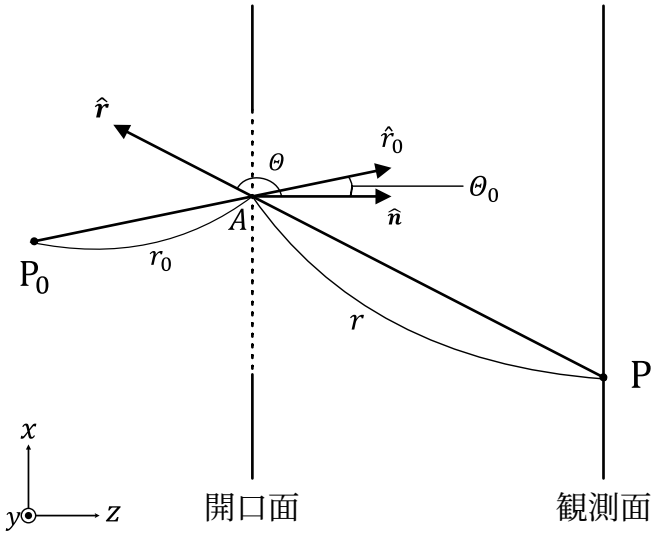


図 5.2　$y$ 軸方向から見たシミュレーションの模式図

## 5.2　実験結果

通常の逐次処理で実行したときの計算時間と OpenMP を利用したときの計算時間、OpenACC を利用したときの計算時間で比較を行った。図は、5 回実行したときの平均の計算時間である。研究室の計算機サーバーの関係上、OpenMP の比較では WindowsPC(CPU:Intel Corei5-11400) で、OpenACC の比較では LinuxPC(CPU:Intel Xeon Bronze 3104, GPU:Quadro P620) でそれぞれ実行した。OpenMP の比較で利用したコンパイラは gcc コンパイラ、OpenACC の比較で利用したコンパイラは PGI コンパイラである。いずれも実行速度をあげるための最適化オプションは指定していない。

表 5.1　逐次処理と OpenMP の計算時間の比較

| 実行方法 | 実行時間 [s] |
|---|---|
| 逐次処理 | 14.256 |
| OpenMP | 2.208 |

表 5.2　逐次処理と OpenACC の計算時間の比較

| 実行方法 | 実行時間 [s] |
|---|---|
| 逐次処理 | 75.59 |
| OpenACC | 29.11 |

## 5.3　考察

回折シミュレーションプログラムの逐次処理と並列処理の実行時間を比較した結果、OpenMP を利用した時、OpenACC を利用した時ともに計算時間が短縮された。このことから、Aqualis のコーディングで数値解析プログラムの計算時間の短縮が可能になったことが分かった。これにより、本研究の目的が達成された。

# 第 6 章

# 結論

本研究では、数値解析プログラミングのための新規プログラミング言語 Aqualis に並列化機能を追加し、計算時間の短縮を図った。

## 6.1 並列化

4 章で述べた通り、Aqualis に OpenMP・OpenACC の指示文を挿入する機能を追加した。これにより、マルチコアを利用した並列プログラミングと GPU デバイスを利用した並列プログラミングが、簡単に行うことができるようになった。

## 6.2 計算時間の短縮

5 章での回折現象のシミュレーション計算の実験では、OpenMP と OpenACC のいずれを利用したときでも計算時間の短縮に成功した。他の数値解析プログラムでも同じ結果が得られることが予想される。ゆえに、本研究の目的が達成された。

# 付録 A

# 追加したメソッド一覧

ここでは、Aqualis に並列計算を実装するために追加したメソッドの一覧を載せる。

## A.1 OpenMP

OpenMP の並列化機能を実装するために追加したメソッドを表 A.1 に示す。

## A.2 OpenACC

OpenACC の並列化機能を実装するために追加したメソッドを表 A.2 に示す。

表 A.1 OpenMP のメソッド一覧

| 名前 | 引数 | 役割 |
|---|---|---|
| omp.parallelize | code:unit->unit | ループを並列化 |
| omp.parallelize_th | th:int, code:unit->unit | ループを指定のスレッド数で並列化 |
| omp.reduction | var:num0, ope:string, code:unit->unit | ループを並列化 |
| omp.reduction_th | th:int, var:num0, ope:string, code:unit->unit | ループを指定のスレッド数で並列化 |
| omp.sections | th:int, code:unit->unit | 指定のスレッド数で並列化 |
| omp.section | code:unit->unit | シングルスレッドで実行 |
| omp.thread_num | - | スレッド番号 (num0) |
| omp.max_threads | - | CPU の最大スレッド数 (num0) |
| ch.private_i | code:num0->unit | プライベート変数の宣言 |
| ch.private_d | code:num0->unit | プライベート変数の宣言 |
| ch.private_z | code:num0->unit | プライベート変数の宣言 |
| ch.private_ii | code:num0*num0->unit | プライベート変数の宣言 |
| ch.private_id | code:num0*num0->unit | プライベート変数の宣言 |
| ch.private_iz | code:num0*num0->unit | プライベート変数の宣言 |
| ch.private_dd | code:num0*num0->unit | プライベート変数の宣言 |
| ch.private_dz | code:num0*num0->unit | プライベート変数の宣言 |
| ch.private_zz | code:num0*num0->unit | プライベート変数の宣言 |
| ch.private_iii | code:num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_iid | code:num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_iiz | code:num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_idd | code:num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_idz | code:num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_izz | code:num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_ddd | code:num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_ddz | code:num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_dzz | code:num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_zzz | code:num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_iiii | code:num0*num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_iiid | code:num0*num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_iiiz | code:num0*num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_iidd | code:num0*num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_iidz | code:num0*num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_iizz | code:num0*num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_iddd | code:num0*num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_iddz | code:num0*num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_idzz | code:num0*num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_izzz | code:num0*num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_dddd | code:num0*num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_dddz | code:num0*num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_ddzz | code:num0*num0*num0*num0->unit | プライベート変数の宣言 |
| ch.private_zzzz | code:num0*num0*num0*num0->unit | プライベート変数の宣言 |

表 A.2 OpenACC のメソッド一覧

| 名前 | 引数 | 役割 |
|---|---|---|
| oacc.parallelize | code:unit->unit | ループを並列化 |
| ch.copyin_i | code:num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_d | code:num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_z | code:num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_ii | code:num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_id | code:num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_iz | code:num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_dd | code:num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_dz | code:num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_zz | code:num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_iii | code:num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_iid | code:num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_iiz | code:num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_idd | code:num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_idz | code:num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_izz | code:num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_ddd | code:num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_ddz | code:num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_dzz | code:num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_zzz | code:num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_iiii | code:num0*num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_iiid | code:num0*num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_iiiz | code:num0*num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_iidd | code:num0*num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_iidz | code:num0*num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_iizz | code:num0*num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_iddd | code:num0*num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_iddz | code:num0*num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_idzz | code:num0*num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_izzz | code:num0*num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_dddd | code:num0*num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_dddz | code:num0*num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_ddzz | code:num0*num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyin_zzzz | code:num0*num0*num0*num0->unit | CPU から GPU に転送する変数の宣言 |
| ch.copyout_i | code:num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_d | code:num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_z | code:num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_ii | code:num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_id | code:num0*num0->unit | GPU から CPU に転送する変数の宣言 |

| 名前 | 引数 | 役割 |
|---|---|---|
| ch.copyout_iz | code:num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_dd | code:num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_dz | code:num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_zz | code:num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_iii | code:num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_iid | code:num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_iiz | code:num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_idd | code:num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_idz | code:num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_izz | code:num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_ddd | code:num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_ddz | code:num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_dzz | code:num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_zzz | code:num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_iiii | code:num0*num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_iiid | code:num0*num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_iiiz | code:num0*num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_iidd | code:num0*num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_iidz | code:num0*num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_iizz | code:num0*num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_iddd | code:num0*num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_iddz | code:num0*num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_idzz | code:num0*num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_izzz | code:num0*num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_dddd | code:num0*num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_dddz | code:num0*num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_ddzz | code:num0*num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |
| ch.copyout_zzzz | code:num0*num0*num0*num0->unit | GPU から CPU に転送する変数の宣言 |

# 謝辞

本研究の遂行にあたり、指導教官の杉坂准教授には大変お世話になりました。杉坂先生が開発なさった Aqualis には、私達、研究室の数理波動システム研究室のメンバーはとてもお世話になりました。私の研究発表の練習にも、夜遅くまでお付き合いいただきました。深く感謝いたします。最後に、数理波動システム研究室の皆様には、本研究の遂行にあたり多大な激励頂きました。ここに感謝の意を表します。

2023 年 2 月　中野太智

# 参考文献

[1] 杉坂純一郎, “Sugisaka/aqualis.” https://github.com/Sugisaka/Aqualis.

[2] F.S. Foundation. https://fsharp.org/.

[3] Y. Zhang and F. Mueller, “Auto-generation and auto-tuning of 3d stencil codes on gpu clusters,” Proceedings of the Tenth International Symposium on Code Generation and Optimization, pp.155–164, 2012.

[4] M. Frigo, “A fast fourier transform compiler,” Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation, pp.169–180, 1999.

[5] E. Lindahl, B. Hess, and D. Van Der Spoel, “Gromacs 3.0: a package for molecular simulation and trajectory analysis,” Molecular modeling annual, vol.7, no.8, pp.306–317, 2001.

[6] 北山洋幸, OpenACC 基本と実践：GPU プログラミングをさらに身近に, カットシステム, 2018.