プログラムの並列化

菖蒲迫 健介 (九州大学) [*1]
(2022/12/14)

■ 目的

OpenMP を用いた並列計算の基本的なことをまとめる

■ はじめに

現在 (2022/11/18) の WCSPH 熱対流プログラムは実験段階で適当に付け足していたが，段々と考えがまとまってきたので，高速化したいと思った．今のプログラムのままでは粒子数 100×100 を超えてくると，計算に 1 か月もかかってしまう．これだと研究を全然進められないので，プログラムの見直しを行いたい．具体的には，プログラムの改善と並列化である．並列計算を自分で全て実装するのはかなりヘビーではあるが，惑星スケールの問題を解くからにはどのみち必要になるので，今のうちに勉強することにした．このノートでは並列計算に必要な最低限のことをまとめる．

目次

[*1] 九州大学大学院 理学府地球惑星科学専攻 地球内部ダイナミクス研究室 修士 2 年.
E-mail: shobuzako.kensuke.242@s.kyushu-u.ac.jp

1 基本的なこと

この節では，並列化の準備として計算機の基本的なことをまとめる．

1.1 計算機の特性

■ 計算機の速さ

計算機の速度は「1 秒間の演算処理能力」で測られる．ここで，1 秒間に 1 回の (浮動小数点) 演算能力があることを，1FLOPS(Floating–Point Operations Per Second, フロップス) という[*2]．具体的には，フロップス P を以下で定義する．

$$P = N \times C \times O \tag{1}$$

ここで，N は総演算コア数 (全てのコアの数)，C は CPU のクロック周波数[*3]，O は一つのコアが一回のクロックで同時に実行できる演算 (スレッド数) である．例えば，九州大学のスパコン[*4]ITO(サブシステム A[*5]) の場合

- システムを構成するサーバー数が 2000 台 (総ノード数)
- 一つのサーバーにつき CPU が 2 つ (計算ノード)
- CPU 一つに 18 コア搭載
- $C = 3.0$ GHz (クロック周波数)，$O = 32$ (スレッド数)

であるから

$$\begin{aligned} P &= \underbrace{2000 \times 2 \times 18}_{N} \times \underbrace{3.0 \times 10^9}_{C} \times \underbrace{32}_{O} \\ &= 6912000 \text{ GFLOPS} \\ &= 6.912 \text{ PFLOPS} \end{aligned} \tag{2}$$

となる．ここで，GFLOPS はギガフロップス，PFLOPS はペタフロップスで，後者が近年のスパコンに使われる標準的な単位である．この結果は，九大のスパコンが理論的には 1 秒間に 6912 兆回もの浮動小数点演算を行うことができることを意味する．ちなみに，自分が最近買ったノート PC(LG gram16) は，$N = 12$ (CPU1 つに 12 コア搭載)，$C = 3.4$ GHz，$O = 16$ なので

$$P = 12 \times 3.4 \times 10^9 \times 16 = 652.8 \text{ GFLOPS} \tag{3}$$

iPhone14 に搭載されている A16 Bionic は 6 コア 6 スレッドで，$C = 3.46$ GHz なので

$$P = 6 \times 6 \times 3.46 \times 10^9 = 124.56 \text{ GFLOPS} \tag{4}$$

となる．スパコンが桁違いの演算性能を持つことが分かる．

[*2] ここでの「演算」とは，足し算・引き算・掛け算・割り算のことを指す．
[*3] 一秒間に回路の状態を変える回数，あるいは，命令を実行する回数．
[*4] 明確ない定義はないが，最高レベルの演算性能を持つ計算機のことを指し，そのほとんどは並列計算機である．
[*5] https://www.cc.kyushu-u.ac.jp/scp/system/ITO/01_intro.html

スパコン性能の世界ランキングは毎年 6 月と 11 月に更新されるらしく，Top500 という以下のサイトで発表される．

https://top500.org/lists/top500/

ここでは，Linpack ベンチマーク (連立一次方程式のベンチマーク) が用いられる．先ほどのフロップスはハードウェア性能から算出された値であり，理論的な最大演算速度である．この理論性能値を Rpeak と呼び，実効性能値を Rmax と呼ぶ．具体例を表 1 に書いた．2022 年 11 月の Top500 ランキング 1 位はオークリッジ国立研究所のスパコンで[*6]，2 位は富岳となっていた．100 PFLOPS 以上の実効性能値を誇るスパコンは，今のところ世界で 5 台近くしかない．ただし，スパコン毎に得意な計算があって「ランキングが高いほど計算が速い」と一概には言えない．

表 1: 世界のスパコン性能の一例

	コア数	Rmax (PFLOPS)	Rpeak (PFLOPS)	ランキング
オークリッジ国立研究所の Frontier	8,730,112	1102.00	1685.65	1
理化学研究所の富岳	7,630,848	442.010	537.212	2
JAMSTEC の地球シミュレータ	43,776	9.99	13.45	56
九大の ITO サブシステム A	72,000	4.540	6.912	122

■クロック周波数と CPU について

計算機の性能は日進月歩であるが，クロック周波数は 2003 年以降ほぼ横ばいである (図 1)．

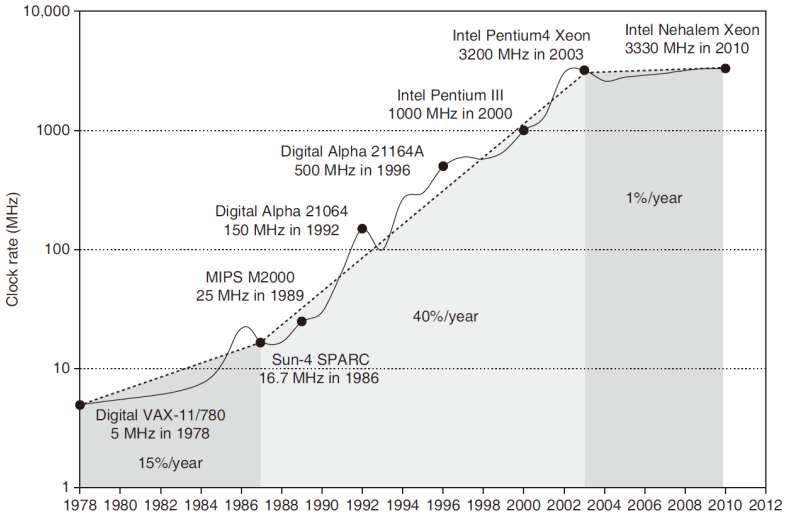

図 1: クロック周波数の変遷 [Hennessy et al., 2012]．

[*6] Wikipedia によると，オークリッジ国立研究所 (Oak Ridge National Laboratory, ORNL) は第二次世界大戦中に建設され，核濃縮が行われていた研究所である．長崎に投下された” ファットマン” に使用されたプルトニウムが生産された場所でもある．

クロック周波数を大きくできれば確かに性能は向上するが，これは物理的な振動に関係しているから，その分発熱量が大きくなる．このせいで冷却も強力でなければならず，それにはかなりの電力が要る．演算速度を向上させる方法として，クロック周波数を上げる他に，コアを増やす，あるいは，演算処理数 (スレッド数) を上げることも考えられる．後者の方法は” 振動” を増やすわけではないから，そこまで電力が要らない．クロック周波数は大きくしない，というのが現代の計算機進化の方針らしい．

そういうことで「CPU やその中のコア数を増やす」というのが基本的なスパコンの方針である．ここで，CPU とは Central Processing Unit の略で，実際に演算を行うパーツである．スパコンに使われる CPU は大きく二種類に分類される．

(1)　マルチコア CPU … クロック周波数を落としたコアを 8 ∼ 32 個並べた CPU
(2)　メニーコア CPU … もっとクロック周波数を落としたコアを 60 個以上並べた CPU[*7]

最近では，GPU(Graphics Processing Unit) の搭載も進んできている．GPU は，計算負荷は小さいが，同時に多くの演算を有する画像・動画処理に適したパーツである．特に，計算科学の観点では機械学習の業界で活躍する．これは単体で使うことはできず CPU と組み合わせて使うため，演算加速器 (アクセラレータ) とも呼ばれる．

■メモリの階層構造

CPU 自体は演算を行う箇所であるから，そのデータのやり取りは他のパーツが受け持つ．これがメモリである．CPU の性能がどれほど高くても，メモリから CPU にデータが届かないと意味がない (データ転送に時間がかかると意味がない) ので，メモリの高速化も重要である．全てのデータを一括して一つのメモリで管理するのは非効率なので，メモリは階層構造を成している (図 2)．CPU から物理的に近い順に，レジスタ，キャッシュメモリ，メインメモリ，ハードディスク (ストレージ) と呼ばれる．

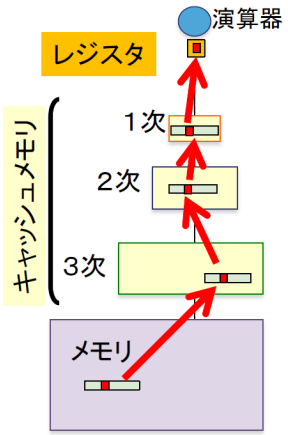

図 2: メモリ階層構造の概念図．九大スパコン超入門講習会の資料より．

[*7] 例えば，Intel Xeon シリーズ．

これは，情報量とアクセス速度によって階層付けられている．

- レジスタ … CPU の内部に設置され，演算器が直接参照できるデータの置き所である．高速なパスで結合されていて，データ転送速度は $\mathcal{O}(1\ \mathrm{ns})$.
- キャッシュメモリ … さらに，レベル 1 からレベル 3 までの階層構造になっている．レベル 1 キャッシュメモリはコア毎に，レベル 2, 3 キャッシュメモリは複数コア毎に組まれる．容量は順に，数 10KB，数 100KB，数 10MB．データ転送速度は $\mathcal{O}(10\ \mathrm{ns})$.
- メインメモリ … CPU 全体に対してデータ管理を行う部分．GB 単位の容量を持つ．データ転送速度は $\mathcal{O}(100\ \mathrm{ns})$.
- ハードディスク … 記憶域とも呼ばれ，世間一般の「容量」を指す．上記 3 つのメモリのデータは電源を切ったら消滅するが，ハードディスク上のデータは保存データとして残る．最近では TB オーダー．データ転送速度は $\mathcal{O}(10\ \mathrm{ms}) = \mathcal{O}(10^7\ \mathrm{ns})$.

レジスタとメインメモリ間の速度と，レジスタ間の速度は 100 倍も異なる．従って，キャッシュやレジスタのデータを上手く再利用するようなプログラミングを組むことが，CPU の性能を活かすために重要となる．

1.2 並列計算の基礎事項

並列計算とは，複数のコアを同時に用いて計算する方法である．概念図を図 3 に示した．

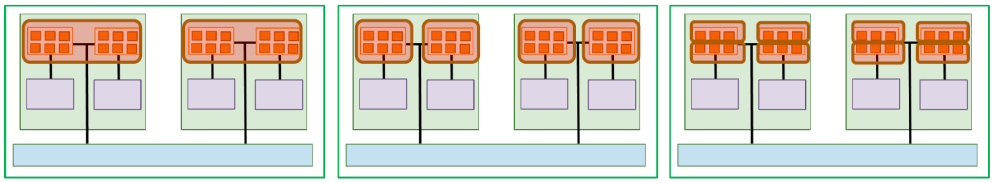

図 3: 並列計算の概念図．九大スパコン超入門講習会の資料より．赤は CPU(6 コア/CPU)，紫はメインメモリ，青がネットワークを示す．今回は 2 つの CPU で 1 計算ノードを構成する．一つのコアで 1 スレッドを割り当てる場合は，(左) 2 プロセス× 12 スレッド．(中) 4 プロセス× 6 スレッド．(右) 8 プロセス× 3 スレッド．

■ 並列計算機のメモリ型の分類

(1) 共有メモリ型 … 一つのメインメモリだけを参照する並列計算の方法
(2) 分散メモリ型 … 複数のメインメモリをネットワーク通信によって参照する並列計算の方法
(3) 分散共有メモリ型 (ハイブリット型) … (1),(2) を複合した方法

共有メモリ型は，図 3 の中図において，6 つのコアで並列化しデータを一つのメインメモリ上で管理する方法である．分散メモリ型は，青のネットワークを使用する並列化である．分散共有メモリ型では，各ノード内の並列化に加えて，複数ノードのネットワーク通信も行うという並列化である．

■ スレッドとプロセス

(1) スレッド … 同じメインメモリを使ってプログラムを実行する流れ
(2) プロセス … 異なるメインメモリを使ってプログラミングを実行する流れ

例えば，図 3 の左図では，一つのメインメモリに対して 12 コアが連結しており，これが二つネットワーク通信によって結合されている．一つのコアで 1 スレッドを割り当てる場合，12 スレッドとなる．よって，2 プロセス× 12 スレッドとなる (分散共有メモリ型)．スレッド/プロセスという概念は，共有メモリ型/分散メモリ型と同じような概念である．

計算ノード内のスレッド分割のみによる並列の流れをマルチスレッドと呼ぶ．他方，計算ノード間の通信を行う流れをマルチプロセスと呼ぶ．二つの違いは，メインメモリを共有するか分散するかである．もちろん，混合したマルチプロセス・マルチスレッドも考えられる．

■ OpneMP と MPI

並列化の実装にはいくつか方法があるが，以下の二つが標準的な枠組みである．

(A) OpenMP (Open Multi–Processing)
マルチスレッド用の並列環境．一つの計算ノード内で完結する並列計算に用いられる．C, C++, fortran コンパイラに対応している．特徴としては，(自動並列化ではなく) ユーザーが並列化指示行を直接記述する点である．ただし，ノード間の並列化はできない[*8]．

(B) MPI (Message Passing Interface)
マルチプロセス用の並列環境．計算ノード間のネットワーク通信を規定するプログラム．郵便物の郵送に似ていて，送り先のアドレスや各データ配列のアドレスなどを記述して通信を行う．一般にかなりややこしい．

MPI による並列化は結構大変そうなので，当面は OpenMP による並列化のみ，つまり，1 計算ノード内で収まる並列化を考える．OpenMP については，第 2 節で詳しく扱う．

1.3 プログラムの最適化

片桐 (2019)『第 1 回 プログラムの高速化の基礎』には，スパコンのことや計算速度向上に関する有用な情報が多数見られた．その内，自分で実装できそうな範囲のことをまとめた．

■ ループ内連続アクセス

配列へのデータアクセスを工夫することで，計算高速化を図ることができる．C 言語と fortran 言語の二次元配列の格納方法を図 4 に示した．これによると，全データを参照する場合，列を固定して行を動かす方が，行を固定して列を動かすよりも効率的ということになる．これを確かめるために，以下のようなテストをしてみた (プログラムは付録を参照)[*9]．

[*8] 16 コア搭載の CPU の場合，16 スレッド以上の並列化はできない (厳密にはスレッド分割は可能だが，平行して演算が進むため意味がない)．最近のスパコンの場合，メニーコアが搭載されていることもあるので，数百のコアを OpenMP のみで使うこともできる．
[*9] 使用プロセッサは，Intel Core(TM) i7–9700(3.00 GHz).

- 正方行列 A, B に対して

$$C = \sum_{i,j}^{n} A_{ij} \times \sum_{k,l}^{n} B_{kl} \tag{5}$$

を定義し，この演算速度をテストする．ここで，A_{ij} の添え字 i は行，j は列を表すとする．

- ループの方向として，次の 4 つが考えられる (予想：(i)<(ii),(iii)<(iv) の順に遅い)．
  - (i) A, B 共に行方向にアクセスする (列を固定して行を動かす)
  - (ii) A は行，B は列方向にアクセスする
  - (iii) A は列，B は行方向にアクセスする
  - (iv) A, B 共に列方向にアクセスする (行を固定して列を動かす)

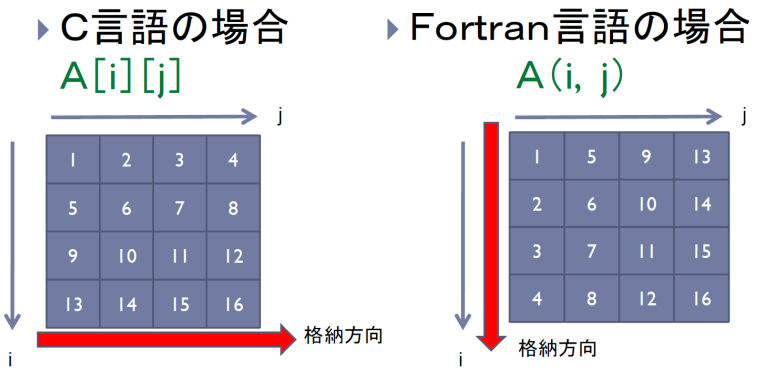

図 4: 配列の格納方式の違い．名古屋大学情報基盤センターより．

結果は図 5 のようになった．この結果は，(i)<(iii)≪(iv)<(ii) の順に時間がかかることを示す．(iii) が速いのは，一番末端のループが連続アクセスされることが結構重要であることを示唆する．配列が小さい範囲においては大きな差にはならないが，500 × 500 程度では約 1.2 倍の速度差になる．二次元以上の配列では行方向に連続アクセスすることが重要であることが分かった．

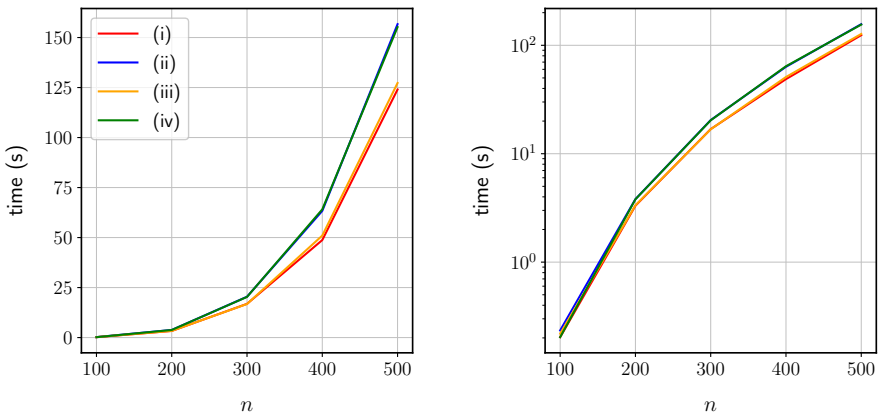

図 5: 配列アクセス順序による計算時間の違い．右図は log プロット．

■ ちょっとした高速化技術

配列アクセス以外の工夫で計算を高速にする技も紹介されていたので，テストしてみた[*10]．

(1) 共通部分の削除

内容：プログラムの冗長な部分を削除する (特に，配列へのアクセスがある場合)

具体例：

(a) ダメな例 [a]

```
do i = 1, n
    d = d + a(i) + b(i) + c
    f = f + d + a(i) + b(i)
end do
```

(b) 良い例

```
do i = 1, n
    tmp = a(i) + b(i)
    d = tmp + c
    f = d + tmp
end do
```

[a] [菖蒲迫メモ] 冗長な計算といっても，単純な計算なら余計な演算を行う (b) の方が遅かった．tmp のような引数の導入は，上の問題のようにループ内に配列が含まれる場合に有効である．恐らく，計算毎にデータの問い合わせが必要になるからだと思われる．

実験：上のプログラムを 100 万回ループさせて，それぞれの cpu time を計測した

結果：(b) の方がかなり速い (図 6)．特に，配列のサイズが大きくなるほど効果があるようだ．

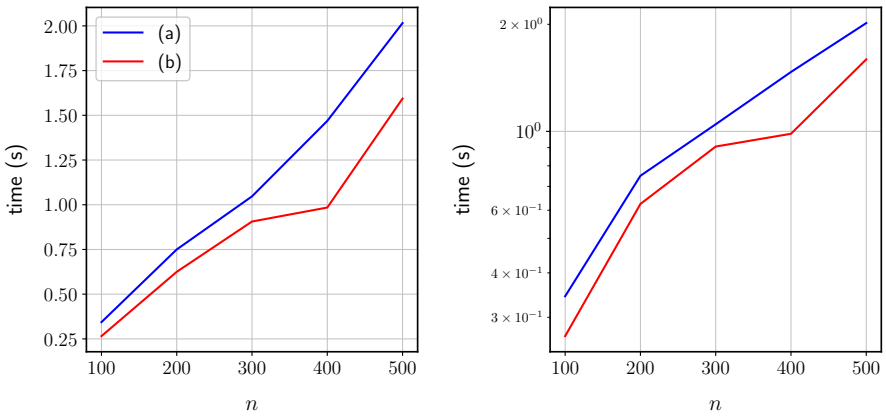

図 6: 共通部分の削除による計算コストの比較．n は配列の大きさ．

[*10] プログラムの詳細は付録を参照．

(2) 割り算をならべく書かない

内容：演算の速さは，足し算 ≒ 引き算 > 掛け算 > 割り算の順番らしいので，ループに入れない

具体例：

(a) ダメな例 [a]

```
do i = 1, n
    a(i) = a(i) / sqrt(3.d0)
end do
```

(b) 良い例

```
tmp = 1.d0 / sqrt(3.d0)
do i = 1, n
    a(i) = a(i) * tmp
end do
```

[a] [菖蒲迫メモ] 割り算が遅いと言っても，単純な数字の割り算は速かった．逆に余計な演算を行う (b) の方が遅い．(a) が遅くなるのは分母に余計な計算がある場合で，例えば上のような `sqrt` による除算が発生する場合である．

実験：上のプログラムを 10 万回ループさせて，それぞれの cpu time を計測した

結果：(b) の方が若干速い (図 7)．とても速くなる訳ではない．

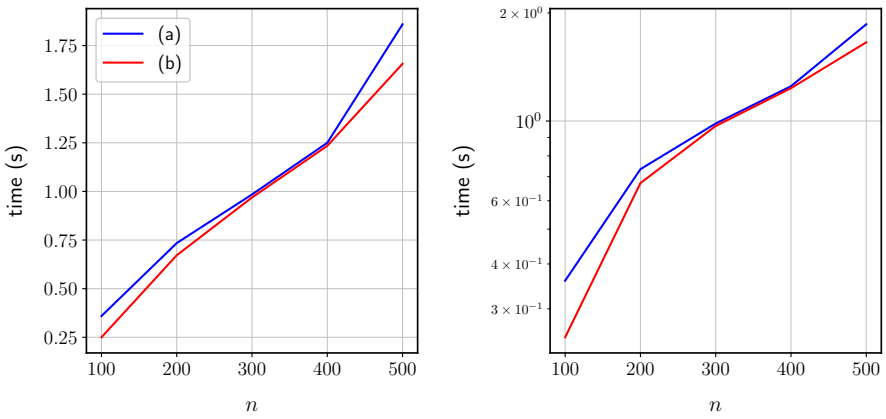

図 7: 割り算を使わないことによる計算コストの比較．n は配列の大きさ．

(3) ループ中に if 分をなるべく書かない

内容：論理演算子は時間がかかるから，ループに入れない

具体例：

(a) ダメな例

```
do j = 1, n
    do i =1, n
       if (i /= j) then
          A(i, j) = B(i, j)
       else
          A(i, j) = 1.d0
       end if
    end do
end do
```

(b) 良い例

```
do j = 1, n
    do i =1, n
          A(i, j) = B(i, j)
    end do
end do
do i = 1, n
      A(i, i) = 1.d0
end do
```

実験：上のプログラムを 10 万回ループさせて，それぞれの cpu time を計測した

結果：(b) の方が速い (図 8).

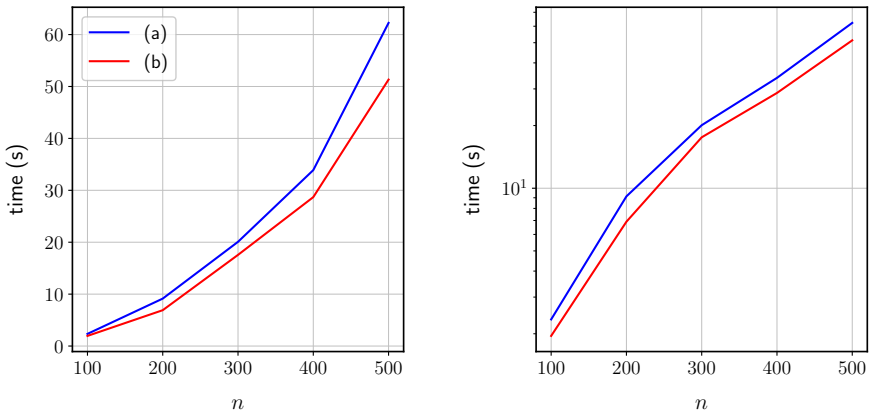

図 8: if 分削除による計算コストの比較．n は配列の大きさ．

(4) サブルーチンの呼び出し回数は最小にする

内容：サブルーチンの呼び出しには時間がかかる

具体例：

(a) ダメな例

```
do j = 1, n
    do i =1, n
       call test_1(ans, arg(i, j))
    end do
 end do
subroutine test_1(ans, arg)
    ans = ans + arg
end subroutine test_1
```

(b) 良い例

```
call test_2(ans, arg(:, :))
subroutine test_2
    do j =1, n
       do i = 1, n
          ans = ans + arg(i, j)
       end do
     end do
end subroutine test_2
```

実験：上のプログラムを 10 万回ループさせて，それぞれの cpu time を計測した

結果：(b) の方が速い (図 9).

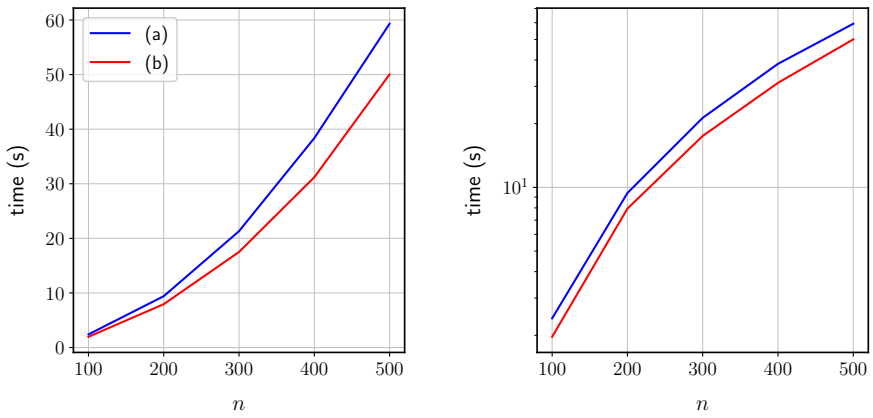

図 9: サブルーチン呼び出しコストの比較．n は配列の大きさ．

2 OpenMP による並列計算

主に，牛島 省 (2006)『OpenMP による並列プログラミングと数値計算法』を参考にした．

2.1 準備

■ 並列計算の例 (リスト 1.2)

```
1 program hello_omp
2     !$ use omp_lib ! ライブラリーの使用宣言
3     !$ call omp_set_num_threads(4) ! 使用するスレッド数を指定
4     !$OMP parallel ! 並列処理の開始
5         write(*,*) 'hello' ! 並列処理の対象
6     !$OMP end parallel ! 並列処理の終了
7 end program hello_omp
```

上のように，OpenMP に関連するコマンドは全て!$で始める．並列処理の対象箇所は

```
!$OMP parallel  …  !$OMP end parallel
```

の中に書く．ターミナルからコンパイルする時は，gfortran の場合

```
gfortran -fopenmp hello_omp.f90
```

のように，オプションを付けて書くことに注意．OpenMP 非対応のコンパイラを用いた場合，コメントアウトと見なされて，並列処理は実行されない[*11]．a.exe で実行し，4 つの hello が出力されれば並列処理が実行できている．

■ 言葉の定義

- マスタースレッド (イニシャルスレッド) … スレッド分割前の大元のスレッド
- フォーク (fork) … 並列演算のための「チーム」を結成すること (!$OMP parallel)
- ジョイン (join) … 「チーム」の解散 (!$OMP end parallel)
- 条件付きコンパイル文 … 並列演算用の呼び出しルーチン (!$ call omp_set_num_threads(4))

■ その他の注意

- 並列演算のコマンドを&で繋ぐ場合，その先頭に!$を必ず書くこと
- 並列演算のコマンドをコメントアウトするには，先頭に!を付け足して，!!$とする

OpenMP による並列化は大きく二つに分かれる．一つは，いくつかの処理を含むプログラム中の比較的大きな範囲を並列化する際に利用され，ここに使われる構文をパラレル構文と呼ぶ．もう一つは，単一のループなど小さな範囲を並列化する際に利用され，ここに使われる構文を結合ワークシェアリング構文と呼ぶ．パラレル構文の方が，フォーク・ジョインやスレッド割り当て等に対するオーバーヘッドが小さく済むので計算上有利となる．

[*11] [菖蒲迫メモ] 自分の場合，fortran をインストールする時に，OpneMP を入れるオプションを選択していなかったので，リスト 1.2 をコンパイルできなかった．そこで，fortran を一旦アンインストールして，OpenMP を含めて再インストールした．

2.2 パラレル構文

■ 基本事項

- フォークによってスレッド分割が起こるが，その後マススレッド演算に入るのは，全スレッド演算が終了してからとなる．つまり，仕事量の偏りがあると非効率となる．
- `parallel` 文は do–end do 間，あるいは，if–end if 文 (ネスト) の間に置いてはいけない．しかし，「節 (clause)」を付けることができる．

■ `parallel` 文に付随する節 (リスト 2.1)

```
1  program chk_clause
2      !$ use omp_lib
3      integer :: flag = 1
4      !$OMP parallel if (flag == 0) num_threads(2) ! 最初のパラレル構文
5          !$ write(*,*) 'p_region_1, id =', omp_get_thread_num()
6      !$OMP end parallel
7
8      !$OMP parallel if (flag == 1) num_threads(2) ! 二番目のパラレル構文
9          !$ write(*,*) 'p_region_2, id =', omp_get_thread_num()
10     !$OMP end parallel
11 end program chk_clause
```

最初のパラレル構文では if 節が偽なので，2 つのスレッドは生成されず，単一のスレッドによる逐次演算が行われる．ここで，実行時ルーチンの `omp_get_thread_num()` はスレッド番号を返す[*12]．そこで，最初のパラレル構文ではマススレッドによるスレッド番号 (0) が一回だけ出力される．二番目のパラレル構文では if 文が真なので，2 つのスレッドが用意され，`omp_get_thread_num()` によって，0 と 1 がターミナルに出力される．

[菖蒲迫メモ] `!$ write` を単に，`write` と書くとマススレッド演算となる

言葉の定義

- 構文 (静的範囲) … 前後の指示文とその間に記述されたプログラム領域
- リージョン (動的範囲) … 構文内に呼び出されるサブルーチンや，構文内の全てのスレッドが実行するプログラム領域
- パラレル構文 … 並列演算の静的範囲
- パラレルリージョン … 並列演算の動的範囲

■ 並列のネスト (リスト 2.3)

パラレルリージョン内部では，スレッドが入れ子状態になる場合も考えられる (図 10)．このように，あるスレッドの下に新しくスレッドを形成することを並列のネスト[*13]という．並列のネストが起こった場合，元のスレッドはマススレッドとなり，スレッド番号が 0 となる．

並列のネストに関するプログラム例を示す (リスト 2.3)．

[*12] スレッド番号とは，パラレル構文内において各スレッドに割り当てられた番号で，0 から始まる．スレッド番号の最大値は，割り当てスレッド数から 1 を引いた数に等しい．逐次演算，もしくは，マススレッドのスレッド番号は 0 である．

[*13] nested parallelism.

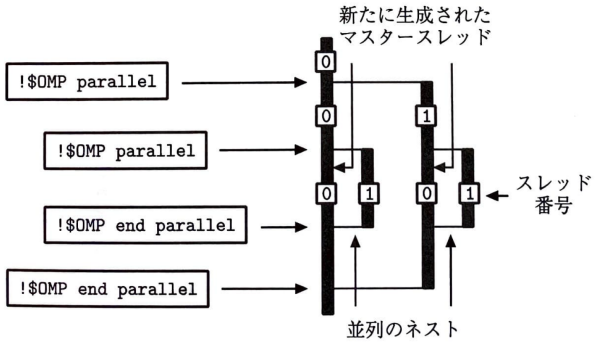

図 10: 並列のネスト．□内の数値はスレッド番号．牛島 (2006) より．

```
1  program chk_nest
2      !$ use omp_lib ! ライブラリーの使用宣言
3      integer main_id
4      !$ write(*,*) 'nest status-1 =', omp_get_nested()
5      !$ call omp_set_nested(.true.) ! 並列のネストの有効化
6      !$ write(*,*) 'nest status-2 =', omp_get_nested()
7      !$OMP parallel num_threads(2) &
8      !$OMP          private(main_id) ! 親のパラレル構文
9          !$ main_id = omp_get_thread_num()
10         !$ call sub_nest(main_id)
11     !$OMP end parallel
12 end program chk_nest
13
14 subroutine sub_nest(main_id)
15     !$ use omp_lib
16     integer main_id
17     !$OMP parallel num_threads(2) ! 子のパラレル構文
18         !$ write(*,*) 'main_id / sub_id=', &
19         !$              main_id, omp_get_thread_num()
20     !$OMP end parallel
21 end subroutine
```

出力結果：

```
 nest status-1    =  F
 nest status-2    =  T
main_id / sub_id  =  0  0
main_id / sub_id  =  0  1
main_id / sub_id  =  1  0
main_id / sub_id  =  1  1
```

プログラム中の実行時ルーチンの説明を以下に書いた．

- omp_get_nested() … 並列のネストが有効か無効かを返す関数
- omp_set_nested() … 並列のネストを有効または無効に設定するサブルーチン
  [注意] OpenMP のデフォルト設定では，並列のネストは無効

プログラム前半で並列のネストが有効化している (F→T)．親のパラレル構文でスレッド数を 2 を指定したので，スレッド番号 0 と 1 の二つのスレッドが起動する．最初の

```
main_id = omp_get_thread_num()
```

で，自分のスレッド番号を変数 main_id に格納する．サブルーチンの呼び出しは各スレッドで行われ，各々 2 つのスレッドにさらに分かれる構造になっている．

出力結果を見ると，親から子へ分かれる時に，確かにマスタースレッドが新しく形成されていることが分かる．ここで，変数 main_id をプライベート変数として定義したから，親のパラレルリージョンでは別々の値として保持され，” 親”(スレッド番号 0) から起こったパラレルリージョンでは，” 別の親”(スレッド番号 1) の main_id の値が参照されない．一方，子のパラレルリージョンでは引数 main_id は共有変数であるから，同じ値が出力される．このように，並列のネストでは「変数がどのパラレルリージョンに対してプライベートになって，共有になるのか」を意識することが大変重要となる．このような変数の性質を，共有属性[*14]という (後述)．

[菖蒲迫メモ]

- omp_set_nested(.true.) としない場合の結果は以下．

```
 nest status-1    =  F
 nest status-2    =  F
main_id / sub_id  =  0  0
main_id / sub_id  =  1  0
```

- 上のリスト 2.3 では条件付きコンパイル文!$ call omp_set_num_threads(2) とせず，節として付随させている．もし，これを書いた場合，全てのパラレルリージョンに対して同じスレッド分割数を定義することになる．

■ その他の注意

パラレルリージョン内のスレッド数に関する注意として，以下がある．

(1)　並列のネストをサポートしていないシステムもある
(2)　スレッド分割数を明記しなければ，自動調整が行われる (動的調整機能)

以上の理由から，以下の並列化では並列のネストを使用せず，スレッド分割は条件付きコンパイル文を用いて明記するというルールを決めておく．また，動的調整機能を明示的に無効化しておく．

```
!$ call omp_set_dynamic(.false.)
```

[*14] sharing attribute.

2.3 共有属性

先述のように，パラレルリージョン内では「演算に用いられる変数が共有変数[*15]であるか，プライベート変数[*16]であるか」を明確に認識しておく必要がある．

- 共有変数 … 全てのスレッドからアクセス可能な変数
- プライベート変数 … 1 つのスレッドだけが使用できる変数

このような変数の性質を共有属性[*17]，あるいは，データスコープ属性[*18]という．

■ 共有属性節の使用例 (リスト 2.6)

```
1  program scope_test
2      !$ use omp_lib ! ライブラリーの使用宣言
3      integer :: ms = 100, mp = 100
4      !$ call omp_set_dynamic(.false.) ! 動的調整機能を無効
5      !$ call omp_set_num_threads(2) ! スレッド数２を選択
6      write(*,*) 'serial␣region,␣mp,␣ms␣=', mp, ms
7      !$OMP parallel default(none) & ! 共有属性節
8      !$OMP private(mp) shared(ms) 共有属性節!
9          !$ mp = omp_get_thread_num()
10         !$ write(*,*) 'id, mp, ms =', mp, ms
11     !$OMP end parallel
12     write(*,*) 'serial␣region,␣mp,␣ms␣=', mp, ms
13 end program scope_test
```

出力結果：

```
serial region, mp, ms   =   100   100
id, mp, ms              =     0   100
id, mp, ms              =     1   100
serial region, mp, ms   =   100   100
```

パラレル指示文にある共有属性節の意味は以下である．

- `default(none)` … 暗黙の変数属性を無効化する
- `private(-)` … プライベート変数
- `shared(-)` … 共有変数

`default` 節がない場合で共有属性が明記されていない場合は，共有変数として扱われる．`none` で無効化した場合は，全ての変数に属性を指定する必要がある．上の例では，プライベート変数 `mp` がスレッド番号を取得し，共有変数 `ms` と一緒に各スレッドで出力させている．最後のマスタースレッドにおける出力で，変数 `mp` の部分が 100 となっている点に注意．すなわち，あるスレッドに属するプライベート変数の値は，パラレルリージョン終了時に消滅し，値が保持されない．また，プライベート変数はパラレルリージョン開始時に別のメモリ上に新しく確保されるため，マスタースレッドの変数とは区別される．パラレルリージョンに入る直前の同一名称の変数値をプライベート変数の初期値として使用したい場合は，`firstprivate` 節を使用する．

[*15] shared variable.
[*16] private variable.
[*17] sharing attribute.
[*18] data scope attribute.

■ 引数の共有属性に関するプログラム例 (リスト 2.9)

パラレルリージョン内のサブルーチン引数の共有属性は，パラレル構文の設定を引き継ぐ．

```
1  program scope_test_2
2      !$ use omp_lib
3      integer :: mp = 0, ms = 100
4      !$ call omp_set_dynamic(.false.)
5      !$ call omp_set_num_threads(3)
6      !$OMP parallel private(mp) shared(ms) ! 共有属性の設定
7          !$ call cal_mp(mp, ms)
8          !$ write(*,*) 'id, mp =', omp_get_thread_num(), mp
9      !$OMP end parallel
10 end program scope_test_2
11
12 subroutine cal_mp(mp, ms) ! サブルーチン
13     !$ use omp_lib
14     integer mp, ms ! 引数は構文内と同じ共有属性を持つ
15     mp = (1 + omp_get_thread_num()) * ms
16 end subroutine cal_mp
```

出力結果：

```
id, mp  =  0  100
id, mp  =  1  200
id, mp  =  2  300
```

■ 大域変数の共有属性に関する誤ったプログラム例 (リスト 2.11)

パラレルリージョン外のサブルーチン引数の共有属性は，パラレル構文外の設定を引き継ぐ．

```
1  module var_mod
2      integer m ! 大域変数を定義
3  end module var_mod
4
5  program wrong_scope
6      use var_mod ! モジュールの使用宣言
7      !$ use omp_lib
8      m = 100 ! 初期値の代入
9      write(*,*) 'm␣before␣m-region␣=', m
10     !$ call omp_set_dynamic(.false.)
11     !$ call omp_set_num_threads(2)
12     !$OMP parallel private(m) ! プライベート変数を定義
13         !$ m = omp_get_thread_num()
14         !$ write(*,*) 'before id, m =', omp_get_thread_num(), m
15         call mplus10 ! サブルーチンの呼び出し
16         !$ write(*,*) 'after id, m =', omp_get_thread_num(), m
17     !$OMP end parallel
18     write(*,*) 'm␣after␣m-region␣=', m
19 end program wrong_scope
20
21 subroutine mplus10
22     use var_mod
23     m = m + 10
24 end subroutine mplus10
```

出力結果：

```
m before m-region   =  100
before id, m        =   0   0
after id, m         =   0   0
before id, m        =   1   1
after id, m         =   1   1
m after m-region    =  120
```

2.4 ワークシェアリング構文

ワークシェアリング構文は，パラレルリージョン内のチームを構成するスレッドに対して 1 つの作業を分割して与えたり，スレッドが行う演算を制御する場合に利用される．ワークシェアリング構文には 4 つの種類がある．

(1) do 指示文 … 巨大なループを分割して，各スレッドで分担
(2) sections 指示文 … 互いに非依存な作業を並列して行う
(3) single 指示文 … 一つのスレッドのみで演算を行う
(4) workshare 指示文 … あるプログラムを 1 回だけ実行するように，複数のユニットに分割

これらの詳細は後から書く．ワークシェアリング構文全般に共通することを以下に書く．

- ワークシェアリング構文のネスト (入れ子構造) は許されない．
- ワークシェアリング構文の最終行 (例えば end do 指示文) では，暗黙の同期が取られる．つまり，全てのスレッドに割り当てられた演算が終了した後に，最終行以降の演算が開始される．この同期を解除するには，nowait 節を用いる．これにより，演算を終了したスレッドは他のスレッドの状況に関係なく，最終行以降の演算を開始する．
- 一方，ワークシェアリング構文の開始行は (例えば do 指示文) では，暗黙の同期が取られない．構文に入る前に同期が必要な場合は，barrier 指示文を使用して明示的に行う．

2.4.1 ループ構文 (do 指示文)

ループ構文は通常パラレルリージョン内部で用いられ，「既に生成されているスレッドのチームに反復演算を分割して割り当て，それらを並列的に処理させる」という働きをする．

・例 1．行列ベクトル積の計算 (リスト 2.13)
二次元正方行列 A とベクトル $\boldsymbol{x}$ のベクトル積 $\boldsymbol{y}$ を考える．

$$y_i = \sum_{j}^{n} A_{ij} x_j \tag{6}$$

これをループ構文で書くと以下のようになる．

```
subroutine mv(y, a, x, n)
    !$ use omp_lib
    integer n, i, j
    real(8) x(n), y(n), a(n,n)
    !$OMP parallel default(none) &
    !$OMP shared(y, a, x, n) &
    !$OMP private(i, j) ! ループの制御変数は省略可能
    !OMP do ! ループ構文開始
    do i = 1, n
        y(i) = 0.0d0
        do j = 1, n
            y(i) = y(i) + a(i, j) * x(j)
        end do
    end do
    !$OMP end do ! ループ構文終了省略可能()
    !$OMP end parallel
end subroutine mv
```

do 指示文は，その直後に記述されたループのみを並列化の対象とする．つまり，上の例では外側の do i のループだけが分割されて各スレッドに割り当てられる．一方，do j のループの方は，スレッド間で分割されることなく，全てのスレッドで 1 から n まで動き，全範囲の反復演算が実行される．

・ 例 2. ベクトル内積の計算 (リスト 2.14)
ベクトルの内積 dp を以下で与える．

$$dp = \sum_{i}^{n} x_i y_i \tag{7}$$

これをループ構文で書くと以下のようになる．

```
subroutine dotp(dp, x, y, n)
    !$ use omp_lib
    integer n, i
    real(8) dp, x(n), y(n)
    dp = 0.0d0 ! 内積の値を保存する変数
    !$OMP parallel default(none) &
    !$OMP shared(dp, x, y, n)
    !$OMP private(i)
    !$OMP do reduction(+ : dp) ! reduction
    do i = 1, n
        dp = dp + x(i) * y(i)
    end do
    !$OMP end do
    !$OMP end parallel
end subroutine
```

(事前に)2 つのスレッドに分かれているとする．この時，2 つのスレッドは異なる i に対して同時に演算を行うから，dp に正しい値が格納されない．このような共有変数に対するアクセスの状態をデータレース[*19]という．今はデータレースを防いで正しく総和を求めたい．そこで，reduction 節が加わっている．使い方は

reduction(オペレーター ： 変数名)

ここで，オペレーターには和，差，積の他に，組み込み関数 max，min や論理演算子を用いることができる．複数の変数名を書く場合は，カンマで区切る．

■ 反復計算におけるスケジューリング

do ループを並列化するときに，演算を各スレッドに割り当てる方法をループのスケジュールという．OpenMP では schedule 節を加えることで，これを制御する．

schedule(type, [, chunk])

ここで，do ループの演算をワークシェアリングする際に，1 つのスレッドに割り当てられる連続した反復演算のまとまりをチャンクという．上の引数 chunk には各スレッドに割り当てる反復回数を指定する．

[*19] data race

type 引数の意味は以下．

(1) static … 反復総回数とスレッド数のみに基づいてチャンクサイズを決める
(2) dynamic … あるチャンクサイズ毎に計算を実行し，終わると次のチャンクサイズ分を要求
(3) guided … dynamic よりも高度な演算の割り当てがなされるが，コスト大
(4) runtime … 環境変数に基づいて実行時のスケジュールが選択される

動的なスケジュールは，反復計算における各ステップ間の演算の計算負荷が均等でない場合に，static よりも適切な方法となる．ただし，動的なスケジュールを行うことによる新たなオーバーヘッドが生じるため，計算コストは増加する．適切なチャンクサイズは，コストやループ演算負荷の偏り等を総合して適切に設定する必要がある．

例えば，行列 a の下三角行列とベクトル x の積を計算する場合は，外側ループの i が大きいほど計算負荷が大きくなる．このため，dynamic を選択すると良い (リスト 2.15)．

```
!$OMP parallel
!$OMP do schedule(dynamic) ! 動的なスケジュールを指定
do i = 1, n
    do j = 1, i ! 外側のループに依存
        z(i) = a(i, j) * x(j)
    end do
end do
!$OMP end do
!$OMP end parallel
```

2.4.2 sections 構文

逐次計算を行うための 2 つのプログラム領域があり，どちらを先に実行しても問題にならない時，これらを 2 つのスレッドに割り当てて並列処理させることが有効である．この場合に，sections 指示文を使用する．例を以下に示す (リスト 2.16)．

```
program list_2_16
    !$ use omp_lib
    integer i
    integer :: nth = 2
    !$ call omp_set_dynamic(.false.)
    !$ call omp_set_num_threads(nth)
    i = 1
    !$OMP parallel firstprivate(i)
    !$OMP sections
        !$OMP section
            i = i + 1
        !$OMP section
            i = i + 1
    !$OMP end sections
    write(*,*) 'i␣=', i
    !$OMP end parallel
end program list_2_16
```

nth = 1 とすると 3 と出力される．これはスレッド数が一つしかないため，両方の演算を一つのスレッドが担当したからである．nth = 2 とすると，3，1 が出力される．つまり，同じスレッドが両方のセクションを実行したことになっている．(sections 構文はややこしい．)

2.4.3 single 構文

パラレルリージョン内部において，あるプログラム領域の演算を一つのスレッドだけに実行させたい場合に用いるのが，single 構文である．例を以下に示す (リスト 2.18).

```
1  program list_2_18
2      !$ use omp_lib
3      integer mp
4      !$ call omp_set_dynamic(.false.)
5      !$ call omp_set_num_threads(2)
6      !$OMP parallel private(mp) ! 並列処理の開始
7          !$OMP single ! シングル構文
8              write(*,*) 'input␣mp:'
9              read(*,*) mp
10             if (mp < 0) stop 'mp␣>␣1'
11         !$OMP end single copyprivate(mp) ! コピーして他のスレッドのプライベート変数として共有
12         !$ write(*,*) 'id, mp =', omp_get_thread_num(), mp ! 各スレッドで出力
13     !$OMP end parallel
14 end program list_2_18
```

出力結果：

```
id, mp  =  0  4
id, mp  =  1  4
```

input は 4 を選んだ．copyprivate(-) 節により，読み込まれた値を他のスレッドのプライベート変数にコピーして用いることができる．なお，この節は single 構文のみとされている．single 構文を付けない場合，スレッド数だけインプットが必要となるので面倒である．

2.4.4 workshare 構文

workshare 指示文と end workshare 構文で囲まれたプログラムの演算は複数のユニットに分割され，それぞれのユニットが 1 回だけ実行されるように，各スレッドに仕事を割り当てる．(使い道は少なそう．)

■ スレッドプライベート指示文 (リスト 2.21)

パラレルリージョン内のプライベート変数は，パラレルリージョンが終了した時点で消滅する．異なるパラレルリージョン毎に値を保持するためには，スレッドプライベート指示文を用いる．

```
1  program chk_thread_private
2      !$ use omp_lib
3      integer myid
4      !$OMP threadprivate(myid) ! スレッドプライベート指示文
5      !$ call omp_set_dynamic(.false.)
6      !$ call omp_set_num_threads(3)
7      !$OMP parallel ! 最初のパラレルリージョン
8          !$ myid = omp_get_thread_num() スレッド番号を取得!
9          !$ write(*,*) 'p-region 1 : myid =', myid
10     !$OMP end parallel
11     !$OMP parallel ! 2番目のパラレルリージョン
12         !$ call print_myid(myid) ! サブルーチンへ
13     !$OMP end parallel
14 end program chk_thread_private
15
16 subroutine print_myid(id)
17     !$ use omp_lib
18     integer id ! 共有属性はスレッドプライベート変数
19     !$ write(*,*) 'p-region 2 : myid =', &
20     !$ omp_get_thread_num(), id
21 end subroutine print_myid
```

通常であれば，これでコンパイルできるはずだが自分は上手くいかなかった．(以降はスレッドプライベート指示文は使わない．)

2.5 スレッドの同期と制御

パラレルリージョン内の演算を複数のスレッドで行う際に，これを明示的に制御しなければ，各スレッド間で進行具合が異なってしまう[*20]．特に，流体計算では各時刻における値を一旦共有してから次のステップに進む必要があるため，スレッド間のデータの同期と制御は重要である．OpneMP では，以下の 6 つの指示文を用いる．

(1) `critical` 指示文 … スレッド内演算を順番に行っていく
(2) `atomic` 指示文 … クリティカル指示文の効率化バージョン (ただし，単一命令のみ)
(3) `barrier` 指示文 … スレッド間の明示的な制御を行う
(4) `ordered` 指示文 … クリティカル指示文に順序を付けたバージョン
(5) `master` 指示文 … マスタースレッドでのみ計算させる
(6) `flush` 指示文 … メインメモリの明示的な同期を行う

2.5.1 critical 指示文

この指示文に囲まれたプログラム領域は全てのスレッドにより実行されるが，「一度に単一のスレッドのみによる処理」が行われる．つまり，あるスレッドが critical 構文内に入ると，その他のスレッドは待機することになる．全てのスレッドが演算を終えた時点で，次の演算に入る．どの順序でスレッドがクリティカルリージョンに入るかはコンパイラの実装に任されている．当然ながら，処理速度は落ちる．

1 から 100 までの和を reduction 節を用いないで計算する例を以下に示す (リスト 2.24)．

```
1  program critical_sum
2      !$ use omp_lib
3      implicit none
4      integer psum, sum, i
5      sum = 0
6      !$ call omp_set_dynamic(.false.)
7      !$ call omp_set_num_threads(3)
8      !$OMP parallel default(none) &
9      !$OMP private(i, psum) shared(sum)
10         !$ psum = 0
11         !$OMP do schedule(static) ! 静的なスケジュールを設定
12         do i = 1, 100 ! 1から100を適当に分担
13             psum = psum + i
14         end do
15         !$ write(*,*) 'id, psum =', omp_get_thread_num(), psum
16         !$OMP critical ! クリティカル構文
17             sum = sum + psum
18         !$OMP end critical
19     !$OMP end parallel
20     write(*,*) 'sum␣=', sum
21 end program critical_sum
```

[*20] ワークシェアリング構文における最終行の「暗黙の同期」は例外．

出力結果：

```
id, psum   =    0     595
id, psum   =    1    1683
id, psum   =    2    2772
sum        =  5050
```

ループ構文で 1 から 100 までの和の計算を各スレッドで分担して計算し，クリティカル構文で各スレッドの psum を順に足し合わせるプログラムである．ここで，マスタースレッド内で全ての演算が行われるから，プライベート変数は破棄されずにクリティカル構文に入ることができる．上の例では，静的なスケジュールを設定しているが，これを動的なスケジュール (dynamic) に変更すると，一つのスレッドのみで演算が終了した．

2.5.1 atomic 指示文

多くの共有メモリシステムには，メモリ上のある最小単位を更新する際に，ある一つのスレッドだけが疎メモリ位置にアクセスすることを保証するハードウェア上の機能が備えられている．この機能を利用すると，クリティカル構文を使うよりも効率的な作業を行うことができる．ここで用いられるのが，atmic 構文である．ただし，atmic 構文で囲んでよいのは単一の命令文だけで，複数行に渡る命令文はクリティカル構文を使用しないといけない．

2.5.1 barrier 指示文

スレッド同期を明示的に行う際に用いる．具体的には，同期したい場所で

```
!$OMP barrier
```

と書く．注意点として，ワークシェアリングを行っている領域に書くことはできない．また，ワークシェアリングの最終行では暗黙の同期がとられるので，barrier 構文を書く必要はない．

2.5.1 ordered 指示文

critical 指示文と同様に，一つ前のスレッド内演算が終わるまで次のスレッドの演算は行われないが，orderd 指示文では，ループで指定された順序通りに逐次化される (リスト 2.28)．

```
1  program list_2_28
2      !$ use omp_lib
3      integer i
4      !$ call omp_set_dynamic(.false.)
5      !$ call omp_set_num_threads(3)
6      !$OMP parallel default(none) &
7      !$OMP private(i)
8      !$OMP do ordered ! オーダー節が必要
9      do i = 1, 100
10         !$OMP ordered
11             write(*,*) omp_get_thread_num(), i
12         !$OMP end ordered
13     enddo
14     !$OMP end do
15     !$OMP end parallel
16 end program list_2_28
```

なお，ループと組み合わせて使う場合は ordered 節が必要となる．

2.5.1 master 指示文

この指示文で囲まれたプログラム領域は，マスタースレッドのみに実行される[*21]．

2.5.1 flush 指示文

全てのスレッドがメインメモリを使えることにに加えて，各スレッドが固有の一時参照メモリ[*22]を持つことが許されている．ここで，一時参照メモリとは，マシンレジスタやキャッシュメモリなどを指す．すなわち，スレッドの一時参照メモリの値とメインメモリの値の不整合を許す使用となっている．このメモリの不整合を防止するために用いられるのが，flush 指示文である．(参考書を読んだがよく分からなかったので割愛．)

2.6 結合ワークシェアリング構文

以上で見てきた並列化の形態は比較的大きなプログラム領域を囲むもので，パラレル構文と呼ばれる．他方，局所的な範囲のみ (例えば，一つのループのみ) を並列化する際に用いられるのが，結合ワークシェアリング構文[*23]である．これを利用する利点はプログラムが簡潔に書ける点である．出来ることは parallel do，parallel sections，parallel workshare の 3 つである．以下にヤコビ法による二次元ラプラス方程式の計算例を書いた (リスト 2.30)．

```
do istep = 1, nstep
    !$OMP parallel do ! 最初のパラレルループ構文
    do j = 2, n-1
        do i = 2, n-1
            g(i, j) = 0.25d0 * (f(i-1, j  ) + f(i+1, j ) &
                              + f(i  , j-1) + f(i  , j+1))
        end do
    end do
    er = 0.0d0
    !$OMP parallel do reduction(+ : er) ! 二番目のパラレル構文
    do j = 2, n-1
        do i = 2, n-1
            er = er + (g(i-1, j) - 4.0d0 * g(i, j) + g(i+1, j) &
                 + g(i, j-1) + g(i, j+1))**2
        end do
    end do
    if (er <= threshold) call finalize_procedure
    !$OMP parallel do ! 三番目のパラレル構文
    do j = 2, n-1
        do i = 2, n-1
            f(i, j) = g(i, j)
        end do
    end do
end do
```

全部で 3 つのパラレルループ構文が書かれている．もちろん，パラレル構文で書くことも可能であり，こちらの方がフォーク・ジョインの数が少ないのでオーバーヘッドは小さく済む．

[*21] スレッド番号が問題にならない場合は，single 構文で記述することもできる．
[*22] temporary view of memory.
[*23] combined parellel work–shareing constructs.

2.7 データ依存性の除去

データ依存性は並列化を行う場合に最も注意しないといけないことである．例えば，次の演算を考える (リスト 2.34)．

```
do i = 2, n
    x(i) = x(i) + x(i-1)
end do
```

スレッド数 2 で計算する場合，スレッド番号 0 または 1 のどちらが先の演算をするかで結果が異なってくる．これはあたかも，スレッド間でデータの「競争」が起こっているような状況なのでデータレースと呼ばれる．この場合，データ依存性を除去する，あるいは，並列化を断念する必要がある．

なお，全てのスレッドが単にデータを参照するだけなら問題は生じない．例えば，以下の場合は並列演算が可能である (リスト 2.35)．

```
do i = 1, n
    z(i) = a * x(i) + y(i)
end do

do i = 1, n
    y(i) = x(i) - x(i-1)
end do

do i = 1, n
    x(i) = a * x(i) + y(i)
end do
```

最初の例は異なる配列への書き出しなので問題ない．二つ目の例も同様である．三つ目の例は制御変数 `i` に応じて各スレッドに演算が割り当てられるので問題ない．

3 熱対流プログラムへの並列化実装

以上を踏まえて，自身のプログラムに以下のような修正および並列化を行った．

- 末端のループが連続的になるように，配列のアクセス順を工夫
- 巨大配列の使用を避け，小さな配列に細分化
- 粒子間相互作用と壁計算における do ループを並列化
- アスキーファイル (テキストファイル) からバイナリーファイルへ

これまでのプログラムとの最も大きな変更は，データレース除去のための「配列の総書き換え」である．これまでは内部粒子と壁粒子 (上・下・右・左・四隅) で分けて配列を用意していた．しかし，この方法は同時処理に適していない．そこで

SP （:，$x, y, u, v, \rho, p, T, \eta, \bar{\rho}, \varpi$）
WALL_b （:，$x, y, u, v, \rho, p, T, \eta, \bar{\rho}, \varpi$）
WALL_t （:，$x, y, u, v, \rho, p, T, \eta, \bar{\rho}, \varpi$）
WALL_r （:，$x, y, u, v, \rho, p, T, \eta, \bar{\rho}, \varpi$）
WALL_l （:，$x, y, u, v, \rho, p, T, \eta, \bar{\rho}, \varpi$）
WALL_br （:，$x, y, u, v, \rho, p, T, \eta, \bar{\rho}, \varpi$）
WALL_bl （:，$x, y, u, v, \rho, p, T, \eta, \bar{\rho}, \varpi$）
WALL_tr （:，$x, y, u, v, \rho, p, T, \eta, \bar{\rho}, \varpi$）
WALL_tl （:，$x, y, u, v, \rho, p, T, \eta, \bar{\rho}, \varpi$）

と書いていたものを

SP_xy （:，x, y）
SP_uv （:，u, v）
SP_rho(:，ρ）
⋮

のように書き換えた．ここで，1 次元目 (:の部分) には粒子数，2 次元目には物理量を格納する[*24]．内部粒子と壁粒子の区別をなくした点と，配列を物理量毎に小分けにした点が新しい．なお，内部粒子と壁粒子を区別するための配列を別途用意している．こうすることで，バックグラウンドセルを用いた並列計算を非常に容易に取り扱うことができるようになった．

加えて，アスキー形式での書き出しは非常に遅いため，これをバイナリー形式での書き出しに変更した．解析する際は Python を使うことになるが，Fortran の unformatted 形式から正しくデータを取り出すためには少々工夫が必要となる[*25]．

[*24] ϖ は particle consistency のことで，1 になることを期待する量である．

$$\varpi \equiv \sum_{j}^{N} \Delta V_j W_{ij}$$

[*25] 特に，バイナリーファイルが append 方式で書かれていると，各時刻のデータに分けるのが (自分にとっては) かなり厄介だった．そこで，書き出しが行われる度に新しいバイナリーファイルを作成するようにしてある．

3.1 プログラムの説明

■ 必要な準備

使用マシンに以下の環境が備わっていることを前提にプログラミングしてある．

- OpenMP が使用可能な gfortran 環境
- Python(numpy, matplotlib 等の基本的なライブラリーも使用)
- Make ファイルの実行環境

■ 実行手順

用いる計算手法は「重力密度分離法を用いた弱圧縮 SPH 法」である[*26]．プログラムを動かすまでの手順は以下である．

プログラムを動かす手順

(1) program_src/input.f90 で計算したい系を選択[a]
(2) ターミナルから program_src まで移動して，`make` コマンドを実行
(3) `[Message] Makefile has been made.` と表示されたら OK
(4) 同じディレクトリーに作成された `zakoken` ファイルを実行
(5) `[Message] Your calculation has been completed !!`と出力されれば，終了

[a] 付属の Excel ファイルで ζ, ξ の値を求めることができる．

■ 配列の説明

基本的に glb_arg.f90 内で配列を定義し，initial_setting.f90 で割り付けを行うことにしている．粒子配列の大きな分類としては

- `SP`：inner 粒子と wall 粒子
- `VM`：vertual marker 粒子
- `MP`：mapping 粒子

例えば，`SP_rho_S(:)` は inner 粒子と wall 粒子の SPH 密度を格納する配列である．二次元目の引数があるものは，x, y 方向の意味である．その他の主な配列の説明は，表 2 の通りである．

■ プログラムの構成について

Fortran で計算して，Pyrhon で解析することを前提にプログラムを組んだ．作成した fortran モジュールは表 3 の通りである．ここで，main.f90 がメインプログラムで，各サブルーチンがここで呼び出されるようになっている．

[*26] 今のところ (2022/12/13)，Gravitational–Density Separated Weakly-compressible SPH と名付けていて，略して GDS–WCSPH 法としている．

表 2: 配列の説明

名前	1 次元目の引数	2 次元目の引数	説明
SP_kind(:)	粒子属性	-	0(inner), 1(bot), 2(top), 3(right), 4(left)
NNPS_arg(:,:)	粒子の ID	所属セル番号	バックグラウンドセル配列
NNPS_9(:,:)	近傍セル番号	セル番号	近傍セル番号の記憶配列
ID_VM_WL(:)	VM の ID	-	各 VM 粒子に対する wall 粒子番号を記憶
NNPS_arg_VM(:,:)	VM の ID	所属セル番号	VM 専用のバックグラウンドセル配列

表 3: モジュールの説明

名前	説明
INPUT	input ファイル
GLB_SETTING	計算で用いるパラメーターの決定
GLB_ARG	割り付け前の配列を定義
cal_EOS	人工的な状態方程式と本当の状態方程式を記載
INITIAL_SETTING	配列の割り付けと初期条件の貼り付け
NNPS	近傍粒子検索に関するサブルーチンを収納
cal_KERNEL	カーネル関数とその微分に関するサブルーチンを収納
cal_WALL	vertual marker 粒子および wall 粒子の計算
cal_main	連続の式/運動方程式/温度の時間発展式/時間積分等を実行
PS_scheme	PS 法を実行
cal_mapping	mapping 粒子の計算
analysis	今のところ V_{rms} を計算
WRITE_OUT	出力 (書き出し) に関するサブルーチンを収納

■ 全体の流れ (アルゴリズム)

プログラム全体のアルゴリズムは以下である (program_src/main.f90 の内容).

(1) 計算システムの確定 (粒子数/カーネル/PS 法/空気対流 or マントル対流など)
(2) inner 粒子，wall 粒子，virtual marker 粒子，mapping 粒子 (解析粒子) の設定
(3) inner 粒子の初期条件を入力
(4) バックグラウンドセルの設定 (近傍 9 つのセル番号を確定)
(5) 近傍粒子探索 (NNPS[*27]) の実行
(6) virtual marker を用いて，初期の壁情報を確定
(7) 初期条件の書き出し
(8) 時間ループの開始

[*27] Nearest Neighbor Particle Searching.

(9) 連続の式，運動方程式，温度の時間発展式の右辺を計算
(10) SPH 密度/速度/温度を更新 (時間積分には 2 次のアダムスバッシュフォース法を使用)
(11) 位置を更新 (もしも，壁を越えていたら完全弾性衝突で跳ね返す)
(12) NNPS の実行 (所属セル番号の更新)
(13) 人工的な状態方程式および本当の状態方程式から，圧力と重力密度を決定
(14)
- PS 法を使う場合 … 壁情報を更新し，PS 法を実行．さらに，NNPS の再実行
- PS 法を使わない場合 … スキップ

(15) 壁情報を更新
(16) 書き出しを行う場合は，ここで実行
(17) 時間ループの終了

■ Python 解析ファイルの使い方

全部で 4 つの Python ファイルを用意してある．

- `plot_analysis.py` … 時系列に沿った図を作成[*28]
- `plot_initial_state.py` … 初期設定確認のための図を作成
- `plot_movie_material.py` … 動画作成のための静止画を作成[*29]
- `delete_file.py` … プログラムダウンロード時の状態にリセットする[*30]

なお，動画作成には別のソフトウェアを使用されたい．`plot_〇〇`ファイルの使い方は以下である．

(1) output_setting ディレクトリ内に出力された `system_check.dat` ファイルの `for Pyrhon plot` 以下の内容をコピー
(2) 各 `plot_〇〇`ファイルの `from system_check.dat` 以下にペースト

■ 出来ること・出来ないこと

基本的には「二次元箱型 (正方形) の弱圧縮 SPH 熱対流計算」専用のプログラムファイルである．そこで，次のような計算はできない．

- 三次元計算
- 自由表面のある問題
- 陰解法による計算
- (今のところ) 熱拡散以外の温度変化項の考慮

基本的なバックグラウンドセルの処理と並列演算のサブルーチンは熱対流以外の問題にも使えるようにプログラミングしてある．そこで，WCSPH を用いた二次元箱型問題であれば，このプログラムを応用して解くことができる．

[*28] デフォルトでは V_{rms}, Nu や速度プロファイル等を描画
[*29] デフォルトでは dpi=300 となっていて，かなり容量を食うので注意．
[*30] mod ファイル/出力ファイル/画像等を一括消去

3.2 並列効率

■ 並列効率の測り方

並列計算の効果を表現する最も一般的な指標は，スピードアップ (speed up) と呼ばれる．

$$S \equiv \frac{T_s(n)}{T_p(n)} \tag{8}$$

ここで，S はスピードアップ，$T_s(n)$ は逐次計算に要する計算時間，$T_p(n)$ は並列計算に要する計算時間である．大きさが n の問題を p 個のプロセッサで解くことを考える．この問題において，並列化が可能な部分にかかる計算時間を $t_p(n)$ とし，並列化が不可能な部分にかかる計算時間を $t_s(n)$ とする．この時，逐次計算を行う場合では

$$T_s(n) = t_s(n) + t_p(n) \tag{9}$$

である．一方，並列計算を行う場合は，理想的には

$$T_p(n) = t_s(n) + \frac{t_p(n)}{p} \tag{10}$$

であるが，実際には通信や並列化のための新たなオーバーヘッド $t_o(n,p)$ が発生するので

$$T_p(n) = t_s(n) + \frac{t_p(n)}{p} + t_o(n,p) \tag{11}$$

と書くべきである．従って，スピードアップ S は

$$S = \frac{t_s(n) + t_p(n)}{t_s(n) + t_p(n)/p + t_o(n,p)} \tag{12}$$

と期待される．n を固定した場合は，p の増加とともにスピードアップが望まれるが，同時に $t_o(n,p)$ の増大するため，実際の並列計算ではある p を超え始めると S が減少する傾向にある．

ここで，並列化効率 ε はスピードアップをスレッド数で割ったものとして定義される．

$$\varepsilon \equiv \frac{S}{p} \leq \frac{t_s(n) + t_p(n)}{pt_s(n) + t_p(n) + pt_o(n,p)} \tag{13}$$

最も理想的な状況とは，プログラム全体が並列化可能かつ $t_o(n,p) = 0$ の場合なので

$$\varepsilon = \frac{S}{p} = \frac{t_p(n)}{t_p(n)/p} \times \frac{1}{p} = 1 \tag{14}$$

となる．これはスレッド数と効率が傾き 1 の直線で表される状況が最大の並列効率であることを示す．しかし，通常は $t_s(n) \neq 0$ で $0 \leq t_o(n,p)$ なので，$0 \leq \varepsilon \leq 1$ となる．(13) 式から明らかであるが，逐次計算領域が大きく ($t_s(n)$ 大)，オーバーヘッドが大きい ($t_o(n,p)$ 大) システムでは効率が出ない．

■ アムダールの法則

全体の計算時間に対する逐次計算部分の割合を $f(n) = t_s(n)/(t_s(n) + t_p(n))$ とおく．この時，(12) 式において $0 \leq t_o(n,p)$ であることを考慮すると

$$S(n,p) \leq \frac{t_s(n) + t_p(n)}{t_s(n) + t_p(n)/p + t_o(n,p)} \leq \frac{t_s(n) + t_p(n)}{t_s(n) + t_p(n)/p} = \frac{1}{f(n) + (1 - f(n))/p} \tag{15}$$

という関係を得る．この式はアムダールの法則 (Amdahl's law) と呼ばれる．この式は，p を増やしてもスピードアップの上限が $1/f(n)$ で制約されることを示しており，並列化に際しては逐次演算を少なくすること ($f(n)$ を小さくすること) が重要であると解釈される．

さて，プログラミングのやり方次第で理想的な条件で見積もられる並列効率よりも高いスピードアップが得られる場合がある．これは各スレッドが扱う問題のサイズ小さくなると，計算機の一時参照メモリ (キャッシュメモリ等) に置かれた変数の使用率が高まることなどが原因と考えられる．この状況は，スレッド数と効率が傾き 1 の直線よりも上側にくるような状況であり，スーパーリニアスピードアップ (superlinear speedup) と呼ばれる．そこで，一般に配列サイズは小さくした方が良いと言われる．

■ 並列効率の確認

せっかくなので，菖蒲迫の弱圧縮 SPH 熱対流プログラムの並列効率を調べてみた．使用マシンは研究室貸し出しのデスクトップパソコンで，8 スレッドで Intel corei7–9700(3.00GHz) を実装している．1000 ステップ (100 ステップ毎に書き出し) 計算した際の結果は図 11 のようになった．スレッド数 2, 4 まではそこそこ効率が良いが，8 までいくとオーバーヘッド気味になるようだ．と言っても，8 スレッドの方がまだ計算が速い．

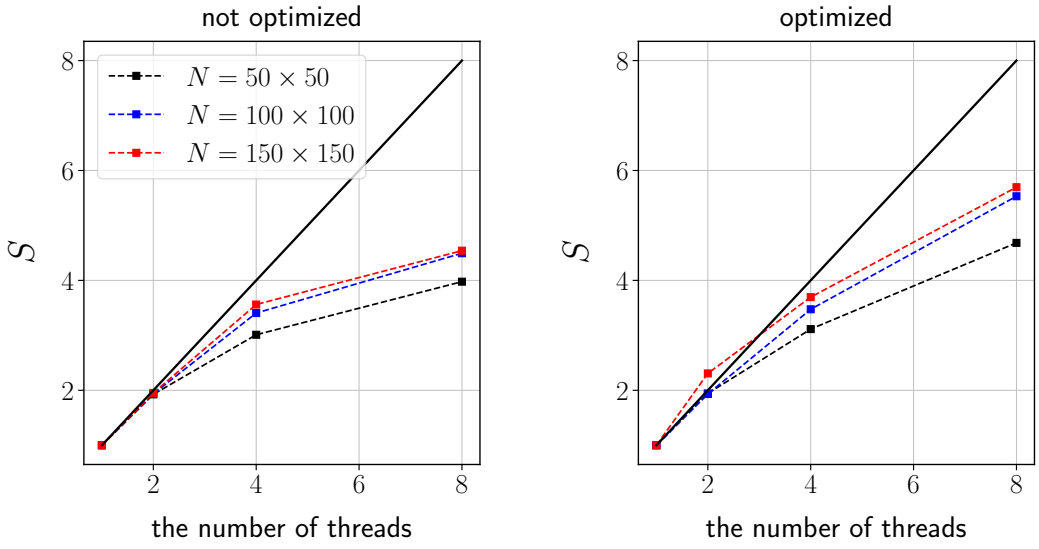

図 11: 並列スピードアップの確認．(左) プログラム見直し前．(右) プログラム見直し後．

謝辞

このノートの作成には，川田 佳史さん (JAMSTEC) の LaTeX テンプレートを使用させて頂きました．この場をお借りして感謝申し上げます．

参考文献

[1] Hennessy, J. L., Patterson, D. A., Asanović, K. (2012) Computer architecture : a quantitative approach, *Morgan Kaufmann Pub.*, 5th

[2] 岩下 武史, 片桐 孝洋, 高橋 大介 (2015) スパコンを知る：その基礎から最新の動向まで, 東京大学出版会

[3] 牛島 省 (2006) OpenMP による並列プログラミングと数値計算法, 第 5 刷, 丸善出版

[4] 牛島 省 (2007) 数値計算のための Fortran90/95 プログラミング入門, 森北出版

ネット上で置かれている資料

[5] 片桐 孝洋 (2019) 第 1 回 プログラムの高速化の基礎, 名古屋大学情報基盤センター
https://www.r-ccs.riken.jp/wp/wp-content/uploads/2020/09/katagiri190411.pdf

[6] 片桐 孝洋 (2019) 第 2 回 並列処理と MPI の基礎, 名古屋大学情報基盤センター
https://www.cc.u-tokyo.ac.jp/events/lectures/X01/shiryou-2.pdf

[7] 片桐 孝洋 (2019) 第 3 回 OpenMP の基礎, 名古屋大学情報基盤センター
https://www.r-ccs.riken.jp/wp/wp-content/uploads/2020/09/katagiri190425.pdf

[8] スーパーコンピュータ超入門講習会, 九州大学情報基盤研究開発センター
https://www.cc.kyushu-u.ac.jp/scp/doc/users/lecture/2019/superintro-2019-1.pdf

付録：計算プログラム

■ ループ内連続アクセス

```
!-------------------------------------------------------------------------------!
! test_arg_access
!-------------------------------------------------------------------------------!

program test_arg_access
    implicit none
    integer :: i, j, k, l, n
    real(8) :: C_1, C_2, C_3, C_4
    real(8) :: time_1, time_2
    real(8) :: delta_time_1, delta_time_2, delta_time_3, delta_time_4
    real(8), allocatable :: A(:,:), B(:,:)

    !-------------------------------------------------------------------------------!
    ! Setting
    !-------------------------------------------------------------------------------!
    write(*,*) 'input␣n␣(size␣of␣argument)'
    read(*,*) n
    if (n < 1) stop '[Error]␣Input␣n>1.'
    allocate(A(n,n), B(n,n))
    call random_number(A(:,:))
    call random_number(B(:,:))
    !-------------------------------------------------------------------------------!
    ! Test (i) --> A: row access, B: row access
    !-------------------------------------------------------------------------------!
    ! zero clear
    C_1 = 0.d0
    call cpu_time(time_1)
    do j = 1, n
        do i = 1, n
            do l = 1, n
                do k = 1, n
                    C_1 = C_1 + A(i,j) * B(k,l)
                end do
            end do
        end do
    end do
    call cpu_time(time_2)
    delta_time_1 = time_2 - time_1
    !-------------------------------------------------------------------------------!
    ! Test (ii) --> A: row access, B: column access
    !-------------------------------------------------------------------------------!
    ! zero clear
    C_2 = 0.d0
    call cpu_time(time_1)
    do j = 1, n
        do i = 1, n
            do k = 1, n
                do l = 1, n
                    C_2 = C_2 + A(i,j) * B(k,l)
                end do
            end do
        end do
    end do
    call cpu_time(time_2)
    delta_time_2 = time_2 - time_1
    !-------------------------------------------------------------------------------!
    ! Test (iii) --> A: column access, B: row access
    !-------------------------------------------------------------------------------!
    ! zero clear
    C_3 = 0.d0
    call cpu_time(time_1)
    do i = 1, n
        do j = 1, n
            do l = 1, n
                do k = 1, n
                    C_3 = C_3 + A(i,j) * B(k,l)
                end do
            end do
        end do
    end do
    call cpu_time(time_2)
```

```
72      delta_time_3 = time_2 - time_1
73      !-------------------------------------------------------------------------!
74      ! Test (iv) --> A: column access, B: column access
75      !-------------------------------------------------------------------------!
76      ! zero clear
77      C_4 = 0.d0
78      call cpu_time(time_1)
79      do i = 1, n
80          do j = 1, n
81              do k = 1, n
82                  do l = 1, n
83                      C_4 = C_4 + A(i,j) * B(k,l)
84                  end do
85              end do
86          end do
87      end do
88      call cpu_time(time_2)
89      delta_time_4 = time_2 - time_1
90
91      ! Result
92      write(*,*) 'Results:'
93      write(*,*) '(i)  :', delta_time_1, 'C=', C_1
94      write(*,*) '(ii) :', delta_time_2, 'C=', C_2
95      write(*,*) '(iii):', delta_time_3, 'C=', C_3
96      write(*,*) '(iv) :', delta_time_4, 'C=', C_4
97
98      ! deallocate
99      deallocate(A, B)
100 end program test_arg_access
```

■ ちょっとした高速化技術

```
1   !-------------------------------------------------------------------------!
2   ! test_speed_up
3   !-------------------------------------------------------------------------!
4
5   program test_speed_up
6       implicit none
7       integer :: i, j, n_1, n_2, t_1, t_2, t_3, loop
8       real(8) :: time_1, time_2
9       real(8) :: delta_time_1, delta_time_2
10      real(8), parameter :: c = 3.d0
11      real(8) :: d, f, tmp
12      real(8), allocatable :: a_arg_1(:), b_arg_1(:), a_arg_2(:,:), b_arg_2(:,:)
13
14      !-------------------------------------------------------------------------!
15      ! Setting
16      !-------------------------------------------------------------------------!
17      write(*,*) 'input n_1 (size of a(i))'
18      read(*,*) n_1
19      write(*,*) 'input n_2 (size of a(i,j))'
20      read(*,*) n_2
21      write(*,*) 'input t_1 (loop conut of test a)'
22      read(*,*) t_1
23      write(*,*) 'input t_2 (loop conut of test b)'
24      read(*,*) t_2
25      write(*,*) 'input t_3 (loop conut of test c)'
26      read(*,*) t_3
27      if (n_1 < 1) stop '[Error] Input n_1>1.'
28      if (n_2 < 1) stop '[Error] Input n_2>1.'
29      if (t_1 < 1) stop '[Error] Input t_1>1.'
30      if (t_2 < 1) stop '[Error] Input t_2>1.'
31      if (t_3 < 1) stop '[Error] Input t_3>1.'
32      allocate(a_arg_1(n_1), b_arg_1(n_1), a_arg_2(n_2,n_2), b_arg_2(n_2,n_2))
33      call random_number(a_arg_1(:))
34      call random_number(a_arg_2(:,:))
35      call random_number(b_arg_2(:,:))
36      !-------------------------------------------------------------------------!
37      ! Test (1) delete common part
38      !-------------------------------------------------------------------------!
39      ! (a) incorrect
40      d = 0.d0
41      f = 0.d0
42      call cpu_time(time_1)
43      do loop = 1, t_1
44          do i = 1, n_1
45              d = d + a_arg_1(i) + b_arg_1(i) + c
```

```
46              f = f + d + a_arg_1(i) + b_arg_1(i)
47          end do
48      end do
49      call cpu_time(time_2)
50      delta_time_1 = time_2 - time_1
51      ! (b) correct
52      d = 0.d0
53      f = 0.d0
54      call cpu_time(time_1)
55      do loop = 1, t_1
56          do i = 1, n_1
57              tmp = a_arg_1(i) + b_arg_1(i)
58              d = d + tmp + c
59              f = f + d + tmp
60          end do
61      end do
62      call cpu_time(time_2)
63      delta_time_2 = time_2 - time_1
64      ! Result
65      write(*,*) 'Result of test (1):'
66      write(*,*) '(a) :', delta_time_1
67      write(*,*) '(b) :', delta_time_2
68      !-------------------------------------------------------------------------------!
69      ! Test (2) don't describe division
70      !-------------------------------------------------------------------------------!
71      ! (a) incorrect
72      call cpu_time(time_1)
73      do loop = 1, t_2
74          do i = 1, n_1
75              a_arg_1(i) = a_arg_1(i) / sqrt(3.d0)
76          end do
77      end do
78      call cpu_time(time_2)
79      delta_time_1 = time_2 - time_1
80      ! (b) correct
81      call cpu_time(time_1)
82      do loop = 1, t_2
83          tmp = 1.d0 / sqrt(3.d0)
84          do i = 1, n_1
85              a_arg_1(i) = a_arg_1(i) * tmp
86          end do
87      end do
88      call cpu_time(time_2)
89      delta_time_2 = time_2 - time_1
90      ! Result
91      write(*,*) 'Result of test (2):'
92      write(*,*) '(a) :', delta_time_1
93      write(*,*) '(b) :', delta_time_2
94      !-------------------------------------------------------------------------------!
95      ! Test (3) don't describe IF in loops
96      !-------------------------------------------------------------------------------!
97      ! (a) incorrect
98      call cpu_time(time_1)
99      do loop = 1, t_3
100         do j = 1, n_2
101             do i = 1, n_2
102                 if (i /= j) then
103                     a_arg_2(i, j) = b_arg_2(i, j)
104                 else
105                     a_arg_2(i, j) = 1.d0
106                 end if
107             end do
108         end do
109     end do
110     call cpu_time(time_2)
111     delta_time_1 = time_2 - time_1
112     ! (b) correct
113     call cpu_time(time_1)
114     do loop = 1, t_3
115         do j = 1, n_2
116             do i = 1, n_2
117                 a_arg_2(i, j) = b_arg_2(i, j)
118             end do
119         end do
120         do i = 1, n_2
121             a_arg_2(i ,i) = 1.d0
122         end do
123     end do
124     call cpu_time(time_2)
```

```
125     delta_time_2 = time_2 - time_1
126     ! Result
127     write(*,*) 'Result␣of␣test␣(3):'
128     write(*,*) '(a)␣:', delta_time_1
129     write(*,*) '(b)␣:', delta_time_2
130
131     ! deallocate
132     deallocate(a_arg_1, b_arg_1, a_arg_2, b_arg_2)
133 end program test_speed_up
```

```
1  ! Test (4)
2
3  program test_subroutine
4      implicit none
5      integer :: i, j, n, t, loop
6      real(8) :: ans_1, ans_2, time_1, time_2
7      real(8) :: time_start, time_end
8      real(8), allocatable :: arg(:,:)
9
10     write(*,*) 'input␣n'
11     read(*,*) n
12     write(*,*) 'input␣loop␣time'
13     read(*,*) loop
14     allocate(arg(n,n))
15
16     call random_number(arg(:,:))
17
18     ! 配列を投げて外で計算される
19     call cpu_time(time_start)
20     do t = 1, loop
21         call test_sub_1(arg(:,:), n, ans_1)
22     enddo
23     call cpu_time(time_end)
24     time_1 = time_end - time_start
25
26     ! 毎回サブルーチンを呼び出す
27     call cpu_time(time_start)
28     do t = 1, loop
29         ans_2 = 0.0d0
30         do j = 1, n
31             do i = 1, n
32                 call test_sub_2(ans_2, arg(i, j))
33             enddo
34         enddo
35     enddo
36     call cpu_time(time_end)
37     time_2 = time_end - time_start
38
39     write(*,*) 'time_1,␣ans_1␣=', time_1, ans_1
40     write(*,*) 'time_2,␣ans_2␣=', time_2, ans_2
41
42 end program test_subroutine
43
44 subroutine test_sub_1(arg, n, ans_1)
45     implicit none
46
47     integer, intent(in) :: n
48     real(8), intent(in) :: arg(n,n)
49     real(8), intent(out) :: ans_1
50     integer :: i, j
51
52     ans_1 = 0.0d0
53
54     do j = 1, n
55         do i = 1, n
56             ans_1 = ans_1 + arg(i, j)
57         enddo
58     enddo
59 end subroutine test_sub_1
60
61 subroutine test_sub_2(ans_2, arg)
62     implicit none
63
64     real(8), intent(inout) :: ans_2
65     real(8), intent(in) :: arg
66
67     ans_2 = ans_2 + arg
68 end subroutine test_sub_2
```