OpenMP 入門 (1)

南里 豪志*　　天野 浩文*

1　はじめに

近年, **共有メモリ型並列計算機**の普及には目覚ましいものがあります．本センターでは，この型の計算機として，平成 12 年 1 月に汎用 UNIX サーバ (FUJITSU GP7000F model 900), 平成 13 年 1 月にスカラー並列サーバ (COMPAQ GS320) を相次いで導入しています．一方, 本センターには，これら以外に，**分散メモリ型並列計算機**でもあるベクトル並列型スーパーコンピュータ FUJITSU VPP5000/64 もあります．規模の違いやベクトルユニットの有無は別として，これらの間には一体どのような違いがあるのでしょうか.

分散メモリ型並列計算機と共有メモリ型並列計算機の構成をおおまかに表したものを図 1 に示します.

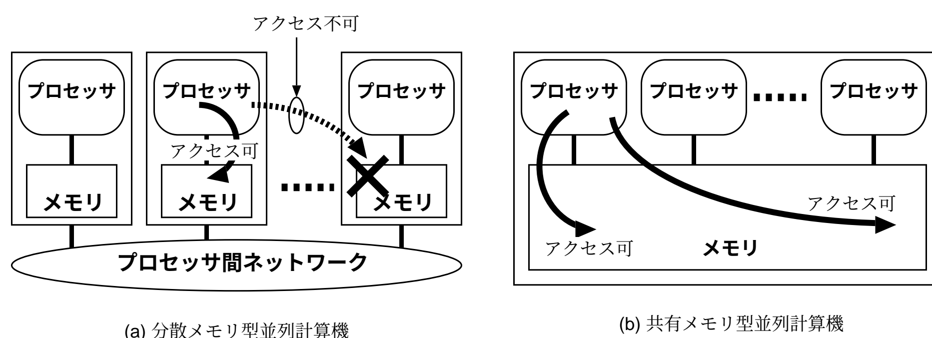

図 1: 分散メモリ型並列計算機と共有メモリ型並列計算機

従来から，共有メモリ型並列計算機は分散メモリ型並列計算機に比べて並列プログラムの作成が容易であるとされてきました．これは，以下のような理由によります.

分散メモリ型並列計算機 (図 1 (a)) においては，メモリは各プロセッサ上に分散して登載されています．それぞれのプロセッサのメモリは他のプロセッサからは直接アクセスできないメモリで，ローカルメモリとも言います．あるプロセッサのローカルメモリに配置されているデータが別のプロセッサで必要となったときには，プロセッサ間のメッセージ通信を用いてデータのやりとりをしなくてはなりません．ところが，現在のコンパイラ技術では，任意の単

* 九州大学情報基盤センター 研究部 スーパーコンピューティング研究部門
E-mail: {nanri,amano}@cc.kyushu-u.ac.jp

一プロセッサ用プログラム (逐次型プログラムとも言います) を，上記のような通信を行う並列プログラムに自動変換することは残念ながらまだ非常に困難です．このため，分散メモリ型並列計算機の利用者は，以下のいずれかの方法をとらざるを得ません．

1. 既存のプログラミング言語を拡張した言語を用いる
   例えば，VPP Fortran[6] のコンパイラで処理できるように，元のプログラムに拡張最適化制御行などを書き加える方法があります．プログラムの書き換えは軽微なもので済むことが多いのですが，HPF[3] などのように，利用できる並列化手法に制限があったり，VPP Fortran などのように特定のメーカの独自言語の場合には他のメーカの計算機で利用できなくなったりするという欠点があります．

2. プロセッサ間通信のための標準的なライブラリで定義されている通信用関数を用いるなどして，元のプログラムを書き換える
   例えば，MPI[5] や PVM[2] のサブルーチンや関数を埋め込んで，プログラムを書き換える方法です．実行文を書き加えたりループを書き換えたりする必要がありますので，「改造」の程度はどうしても多少大きくなります．ただし，国際的な標準規格が制定されているライブラリを用いるのであれば，書き換え後のプログラムは，他のいろいろなメーカの並列計算機でも利用できます．

3. 計算機のメーカが用意した独自の並列プログラミング言語[1] を使用して，新たにプログラムを作り直す
   MPI や PVM が登場する以前の並列計算機ではこれが唯一のやり方でした．プログラムは最初から書き直す必要がありますし，他のメーカの計算機では基本的に利用できないという欠点があります．

一方，共有メモリ型並列計算機 (図 1 (b)) においては，すべてのプロセッサからメモリ上のすべてのデータに同じようにアクセスすることが可能です．これまでのさまざまな研究から，この性質を用いると，単一プロセッサ用に記述されていた逐次型プログラムを自動的に並列化するのが分散メモリ型の場合に比べて非常に容易であることがわかっています．このため，共有メモリ型並列計算機のための自動並列化コンパイル技術がいろいろと開発されてきました．特に，これらの成果を実用化・標準化したものである OpenMP では，**単一プロセッサ用に記述された Fortran プログラムにいくつかの特殊なコメント文を追加する** (C, C++**プログラムの場合には，コンパイラに対する指示文を追加する**) だけで，多くの場合，逐次型プログラムを簡単に並列化することができます．

ここで，MPI を用いる場合との決定的な違いは，**コメント文やコンパイラに対する指示文は実行文ではないので，これを加えただけでは，プログラムの論理的な構造は変化しない**ということです．すなわち，OpenMP を用いたプログラムの書き換えは軽微なもので済むことが多く，サブルーチン呼び出しを書き加えたりループ変数の動きまわる範囲を書き換えてプログラムの論理構造を改造する必要のある MPI の場合に比べ，移行にあたっての最初の段階が非常に楽になります．

---

[1]例えば，Thinking Machines 社の CM-5 という分散メモリ型並列計算機専用に提供されていた並列 C 言語 C* などがあります．以前は本センターでも CM-5 による計算サービスを提供していましたが，すでに運用を終了しています．

これまでは，あまりベクトル化に向かない大規模数値計算プログラムを持っている人や，VPP Fortran による並列化には満足できなかったが MPI による並列化にも抵抗があったというような人もおられたことでしょう．しかし，OpenMP を利用することのできる並列計算機 GP7000F, GS320 が本センターのラインナップに加わったことにより，このような人も，並列処理による実行時間短縮の恩恵を受けられる可能性が高まったことになります．

現在のところ，OpenMP に関する情報源としては，以下のようなものがあります．

- Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D. and McDonald, J.: *"Parallel Programming in OpenMP,"* Morgan Kaufmann Publishers, 2000.
  OpenMP に関するおそらく最初の書籍です．残念ながら，2001 年 9 月現在，OpenMP に関する入門書・解説書はこれくらいしかないようです．日本語による書籍もまだ見当たりません．

- `http://www.openmp.org/`
  OpenMP の規格を管理している一種の業界団体の web サイトです．このサイトでは，OpenMP 仕様書のファイルや，OpenMP に準拠している言語処理系のリスト，FAQ (Frequently Asked Questions, よく聞かれる質問) など，さまざまな情報を入手することができます．また，仕様書の一部については，
  `http://pdplab.trc.rwcp.or.jp/pdperf/Omni/spec.ja/home.html`
  から新情報処理開発機構 (RWCP) および 富士通株式会社による日本語訳を入手することもできます．

九州大学情報基盤センター広報 (全国共同利用版) では，これから 3 回に分けて OpenMP を用いた並列プログラムの作成方法を紹介していく予定です．各記事の内容は以下の通りです．

OpenMP 入門 (1) (本稿)
: OpenMP の概要と簡単なプログラムを紹介し，OpenMP プログラムをコンパイルし実行する方法を解説します．

OpenMP 入門 (2) (次回予定)
: 最も一般的な並列処理の対象であるループ (Fortran における `DO` 文または C(C++) における `for` 文) について，OpenMP を用いた並列化手法を紹介します．

OpenMP 入門 (3) (次々回予定)
: より複雑なプログラムに対する並列化手法を紹介します．また，さらに性能を向上させるための技術や，本センターの計算機での改良前後のプログラムの性能比較を行います．さらに，OpenMP を用いない他の自動並列化コンパイラとの違いについても概説します．

第 1 回である本稿の次節以降の構成は，以下のようになっています．

2 節では，OpenMP が誕生した理由や，OpenMP における並列化の概要，OpenMP の利点を簡単に説明します．3 節では, プログラムを並列化する際の注意点をまとめます．4 節では, いくつかの簡単な例を用いて, OpenMP の基本的な構造と実行イメージを説明します．5 節では, 当センターの計算機上で OpenMP プログラムを翻訳, 実行する方法を紹介します．ま

た, 並列化による速度向上率の計測方法についても説明します. 最後に，OpenMP や並列プログラミングに関する参考文献を，注釈付きでまとめています.

なお, 本記事では主に Fortran プログラムを対象に説明を行ないますが, ほとんどのプログラム例について Fortran だけでなく C 言語によるプログラムも参考として掲載します. ただし, ループの順序や多次元配列の並びが C 言語のプログラムと本文では異なる場合があるので注意して下さい. また, Fortran のプログラム例は自由形式で表記していますが, 固定形式で OpenMP を利用することも可能です.

2　OpenMP の特徴

2.1　OpenMP 誕生の経緯

前節では,「共有メモリ型並列計算機は分散メモリ型並列計算機に比べて並列プログラムの作成が容易である」と述べました. しかし，実は，OpenMP が誕生するまでは，共有メモリ型並列計算機にも問題がありました. それは，共有メモリ型並列計算機のメーカや機種に依存しない共通のプログラミングモデルもツールも無く，それぞれの計算機ごとに個別のプログラムを開発する必要があったことでした. このため，あるメーカの並列計算機のための並列プログラムが用意できても，それは他のメーカの計算機では利用できないことが多かったのです.

そこで，主に計算機メーカに所属する技術者・研究者が共同で開発に乗り出したしたのが, 共有メモリモデルによる並列プログラムの標準規格 OpenMP です. その間に，計算機メーカからも研究機関からも独立した団体である OpenMP Architecture Review Board (ARB) が設立され，OpenMP 規格は，OpenMP ARB によって管理されるようになりました.

その結果，現在では，ほとんどの共有メモリ型並列計算機上で OpenMP に準拠したプログラムを利用できる環境が整いました. また，2001 年 6 月には，さまざまな計算機の性能を比較するための事実上の国際標準である SPEC ベンチマークテストにも OpenMP が追加され, いくつかの共有メモリ型並列計算機の性能測定結果が登録されるようになりました.

今後は，移植性の高い並列プログラムの開発手段として，OpenMP の重要性はさらに高まっていくことでしょう.

2.2　OpenMP の概要

OpenMP は，決して新しいプログラミング言語ではありません. 既存の Fortran や C(C++) の文法の中にある，コメント文や pragma 文を利用して，コンパイラに対し,

「プログラムのここからここまでを並列化しなさい」

と「指示」してやることにしよう，という一種の並列プログラミングモデルまたはスタイルとでも呼ぶべきものです. コメント文や pragma 文のこのような使い方を「知らない」コンパイラは単純にこの文を無視すればよいですし，OpenMP に対応しているコンパイラはこのような文を見てプログラムの並列化を行えばよいことになります.

例として，一次元配列をスキャンして処理する簡単なプログラムを OpenMP を用いて並列化する様子を模式的に書いたものを図 2 に示します．

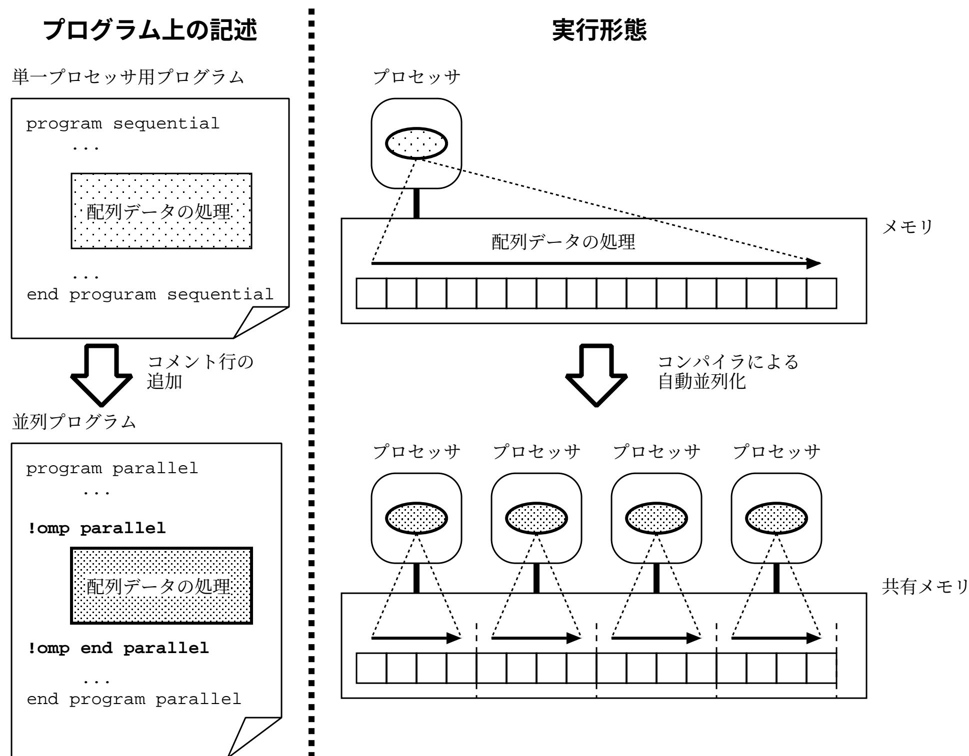

図 2: OpenMP による並列化の様子

図の上側は，一次元配列の要素すべてをループで順に操作する単一プロセッサ用の Fortran プログラムの記述と，その実行形態を模式的に示しています．これを共有メモリ型並列計算機で並列に処理する最も簡単な方法は，この一次元配列をプロセッサの台数と同じ数だけに分割して，個々のプロセッサが自分の持ち分を処理するように変更することです．図の下側は，このプログラムを OpenMP によって並列化する場合の記述と，それをこの方式で並列実行するときの様子を模式的に表しています．

OpenMP では，プログラム内の並列化したい部分 (図中の網掛けの部分) を特殊なコメント文ではさむことによって，どこからどこまでを並列化したいのかコンパイラに指示します．このようにしておくと，一次元配列全体を処理するループを，複数のプロセッサで分担して処理するための実行コードをコンパイラが生成してくれます．

ただし，図中の並列プログラムのコメント文には，具体的にこれが何台のプロセッサで実行されるのかが記述されていません．この数値は，通常，これを実行する計算機のオペレーティングシステムやシェルにおいて，OMP_NUM_THREADS 環境変数に設定されています[2]．図の右下

[2]厳密には，プロセッサの台数を設定するわけではないのですが，これについては，本稿の第 4 節でもう少し詳しく説明することにします．

側は，`OMP_NUM_THREADS` があらかじめ 4 に設定されている状態でプログラムが起動された場合の実行形態を示していることになります．

また，演算や入出力と異なり，コメント文はプログラムの中で実行される構文要素ではありません．したがって，コメント文と環境変数だけでは，動作しているプロセッサのそれぞれに異なる内容の処理を行わせたい，というような場合に，少々不都合を来すこともあります．

そこで，このような機能を補うため，OpenMP では，コメント文と環境変数の他に，いくつかのサブルーチンが容易されています．例えば，`OMP_GET_THREAD_NUM()` 関数を用いると，プロセッサは同時に動いているプロセッサのうちで自分が何番目にあたるかを知ることができ，それに応じて処理の内容を `if` 文で切り替える，といったことができるようになります．

以上のようなことから，おおざっぱに,「OpenMP の機能は以下の 3 つの構成要素によって規定されている」ということができます．

1. コンパイラに並列化の箇所，方法を指示するための，特殊な**コメント文・指示文の文法**
2. コンパイルされたプログラムが実行されるときの環境を決定するための特殊な**環境変数の名前と意味**
3. 並列実行を補助するための特殊な**サブルーチン・関数の名前と機能**

2.3 OpenMP の利点

1 節でも触れましたが，ある計算処理を行うための逐次型プログラムをすでに持っている場合，OpenMP による並列化には，以下のような利点があります．

1. **移行の容易さ**
   最初からプログラムを書き直す必要のある，メーカ独自の並列言語を用いる場合に比べて，移行の負担は格段に軽くなります．また，実行文の追加やループの変形など，かなりのプログラム改造を要する MPI や PVM による並列プログラミングと比べても，移行の最初の段階の手間が極めて少ないという特徴があります．
2. **移植性の高さ**
   メーカ独自の言語と異なり，OpenMP 規格に準拠している並列計算機・コンパイラならば，どこに持って行っても実行が可能です．
3. **単一プロセッサ環境との共存の容易さ**
   OpenMP では，一部のライブラリサブルーチン・関数を除き，実行文の文法には逐次型との違いがありません．したがって，並列化を行ったプログラムでも，多くの場合，単一プロセッサ環境で引き続き利用することができます．
4. **特定のメーカに依存しないこと**
   メーカ独自の並列言語を用いる場合と異なり，OpenMP の規格は，特定のメーカに依存しない国際組織 (OpenMP ARB) によって管理され，そこに参加する多くのメーカが，

この規格の制定・改良に関与しています．また，それぞれのメーカは，自社の計算機に対し，この規格に準拠し高性能かつ信頼性の高いコンパイラを供給しようと努力しています．したがって，たまたま使用している計算機のメーカが万一並列計算機市場から撤退するようなことが起こっても，その計算機と「無理心中」させられる可能性は極めて低いと言えます.

上記の利点のうちのいくつかは MPI や PVM などについてもあてはまることですが，これらすべてを満たしているのは，今のところ OpenMP だけです.

3　プログラムを並列化する前に

もし, 今使っているプログラムを並列化したいと思われているのであれば, おそらく OpenMP は最も容易に試すことの出来る手段の一つです. しかし, それでも並列化するとプログラムのデバッグやチューニングが難しくなります. また, 並列化によって得られる効果が作業に要する時間に見合わない場合も考えられます. 並列化する前に対象となるプログラムについてちょっと調べてみてください.

3.1　並列化前の最適化と動作検証

一般にコンパイラは, プログラムを翻訳するだけでなく, 実行するアーキテクチャに合わせてプログラムを最適化する機能を持っています. この機能を使うと, プログラムによっては並列化しなくても 2〜3 倍程も速度が向上することがあります. コンパイラのオプションを変えるだけで最適化の機能を設定できますので, 並列化する前に一度試してみてください.

また, 一見正常に動作しているように見えるプログラムに実はバグ (間違った部分) が潜んでいることがあります. このバグが, 並列化して実行の順序やデータの配置位置が変わった途端に以上動作を引き起こすようになることは, 十分に考えられます. このようなバグの例としては, 配列の領域外参照, すなわち, 配列の予め宣言した範囲以外の要素を参照する場合が最も一般的です. さらに, 一旦並列化してしまうと同時に流れている複数の処理から誤りを検出するのは非常に困難となります. そこで, 並列化する前にプログラムの動作を検証しておくことをお勧めします.

当センターのスカラー並列サーバ GS320 及び汎用 UNIX サーバ GP7000F 上で利用できる, 最適化や動作検証に関するコンパイラのオプションを, 表 1 に列挙します.

3.2　並列化の効果に関する検討

例えば, 並列化前の処理時間が数分程度のプログラムは, 並列化による効果も高々数分程度です. そのため, 実行する回数が非常に多いプログラムでなければ, 並列化に要する作業量には見合いません. また, 初期値を変えた複数のジョブを同時に実行できるシミュレーションのようなプログラムでは, 並列化するよりもそれぞれのプログラムを数回分同時に進める方が, 作業全体の効率を考えると有効です. ですから, 短いプログラムで並列化の経験を積んだ後は,

表 1: 最適化及び動作検証のためのコンパイラオプション

スカラー並列サーバ GS320 (kyu-ss.cc.kyushu-u.ac.jp)

オプション	意味
`-check bounds`	配列のオーバーフローチェック
`-check format`	フォーマットの不一致をチェック
`-check overflow`	全ての整数演算でオーバーフローチェック
`-check underflow`	アンダーフロー発生時に警告
`-tune ev6 -arch ev6`	GS320 の CPU である EV68 に最適化したモジュールを作成します.
`-unroll` N	ループの展開段数を指示します．あまり大きいものを入れると逆にオーバヘッドが大きくなる可能性があります．(N:ループを展開する段数．通常 6, 8 を入れます。場合によっては 16 でも結構です．)
`-fast`	高速の数値計算ライブラリを使用します．精度が落ちる可能性があります．(倍精度の 15 桁目が異なる程度)
`-O5`	最適化のレベルを上げるもので、効果のある場合は、20〜30 %向上します．これは, 四則計算をコンピュータの速度を出せるように順序を入れ替えます．よって, 数学的には正しいのですが, コンピュータの計算誤差に関して論じると差が生じる場合があります.

汎用 UNIX サーバ GP7000F model 900 (kyu-cc.cc.kyushu-u.ac.jp)

オプション	意味
`-Haesux`	プログラムのデバッグのため，引数の妥当性の検査，添字式・部分列範囲の検査，未定義データの引用の検査などを行ないます．メッセージは翻訳時，実行時に出力されます．実行時間が増大するため，小規模なプログラムに対しデバッグを行ない，デバッグ終了後は必ず，実行可能ファイル，オブジェクトファイルを再作成してください.
`-Kfast`	翻訳しているマシン上で高速に実行させることを指示します.
`-Keval`	演算評価方法の変更の最適化が行われます．実行結果に副作用を生じることがあるため注意が必要です.
`-Kfast_GP=2,prefetch=4 -O4`	メーカーに確認した最大限の最適化オプションです．ただし，実行結果に副作用が生じる可能性があります．また，他の Solaris マシンへの移植もできないため，使用する場合は注意してください. なお, C/C++ の場合は `-Kfast_GP=2,prefetch -Kmfunc` を使ってください.

並列化を行う前に, 対象となるプログラムについて処理時間が十分長いかどうか, 構造的に並列化し易そうかどうか, を検討してみてください.

4　はじめの一歩

4.1　OpenMP プログラムの構造と実行イメージ

前述の通り OpenMP 規格では, 並列化の箇所や方法を指示するコメント行, 並列実行の補助をするサブルーチン・関数, 及び実行時の諸設定を行う環境変数について規定されています. この, OpenMP のコメント行のことを **OpenMP 指示文**, 並列実行の補助をするサブルーチン・関数群をまとめて **実行時ライブラリ関数** と呼びます.

ここでは, OpenMP 指示文と実行時ライブラリ関数を用いた OpenMP のプログラム例を紹介し, その実行イメージを説明します.

4.1.1　OpenMP 指示文

OpenMP の指示文とは, プログラムの並列化方法や並列処理時の設定等を指示する文です. 例えばプログラム中で並列化するループを指示するためにこの指示文を使います. 簡単なプログラムであればこの指示文だけで並列化することができます.

OpenMP 指示文は表 2 のように規定されています. `!$omp` や `#pragma omp` のような OpenMP 指示文であることを示す文字列 (Fortran では接頭詞と呼びます) の後には指示文の種類を示す指示文名を書きます. さらにその後に, 必要に応じて**指示節**と呼ばれる並び項目を追加することができます.

表 2 の Fortran の例では `parallel do` が指示文の名前で, `shared(a)` が指示節となります. 一方 C/C++ の例では `parallel for` が指示文の名前です.

OpenMP に対応したコンパイラはこれらの指示文を解釈し, 適切な並列プログラムに翻訳します. 一方, OpenMP に対応していないコンパイラで翻訳する場合, もしくは OpenMP に対応しているコンパイラで OpenMP の機能を有効にするオプションをつけずに翻訳する場合, これらの指示文は無視されます. すなわち, OpenMP 機能が無効の場合, プログラム中の OpenMP 指示文以外の部分が OpenMP を用いない通常のプログラムとして翻訳されます.

条件コンパイル機能　一つのファイルに並列化前と並列化後のプログラムを共存させることができると, 並列計算機と逐次計算機で同じファイルを使い回すことができて便利です. OpenMP 指示文は, OpenMP 機能が無効である場合は翻訳時に無視されるので, このような目的に適しています. しかし OpenMP が有効である場合と無効である場合で, OpenMP 指示文以外のプログラムの一部を別のコードにしたい場合もあります. そこで OpenMP では, コンパイル時に OpenMP が有効であるか否かによって翻訳する範囲を変更できる条件コンパイル機能が用意されています (表 3). この機能を用いることにより, 並列化前と並列化後のプログラムを同じファイルに共存させることができます.

表 2: OpenMP の指示文

Fortran (自由形式)	`!$omp` で始まる行は OpenMP 指示文として扱われる. (前に空白があっても良い.) 例: `    !$omp parallel do shared(a)`
Fortran (固定形式)	先頭が `!$omp` または `c$omp` または `*$omp` である行は OpenMP 指示文として扱われる. さらに 6 桁目が空白や 0 で無い場合は直前の OpenMP 指示文の継続として扱われる. 例: `c$omp parallel do shared(a)`
C/C++	`#pragma omp` で始まる行は OpenMP 指示文として扱われる. (前に空白があっても良い.) 例: `   #pragma omp parallel for shared(a)`

表 3: OpenMP の条件コンパイル機能

Fortran(自由形式)	先頭が `!$` である行, もしくは空白文字の後の`!$` 以降は OpenMP が有効である場合は翻訳され, 無効である場合は無視される. 例: `  !$   call parallel_init(a)`
Fortran(固定形式)	先頭が `!$` または `c$` または `*$` である行は OpenMP が有効である場合は翻訳され, 無効である場合は無視される. 例: `!$  call parallel_init(a)`

C/C++	OpenMP が有効である場合に限りマクロ変数 `_OPENMP` が定義される.
	例:
	`#ifdef _OPENMP`
	`    parallel_init(a);`
	`#else`
	`    serial_init(a);`
	`#endif`

4.1.2 実行時ライブラリ関数

OpenMP の実行時ライブラリ関数は, 通常の関数やサブルーチンと同様にプログラム中で呼び出すことが出来ます. これらのサブルーチン・関数の例としては, 並列処理時に同時に動作する処理の数を調べる関数や変更する関数等があります. これらの関数を利用すると, プロセッサ数に応じて計算の分担方法を変更したり, 逆にプログラムの処理内容に応じて使用するプロセッサ数を調節したりして, 性能向上を図ることができます. すなわち, プログラムの並列処理効率を向上する場合に実行時ライブラリ関数は重要な役割を果たします. また, 複雑なプログラムを並列化する場合に実行時ライブラリ関数が必要となることもあります.

OpenMP の実行時ライブラリ関数を C 言語もしくは C++言語のプログラムで用いる場合, 以下のようにヘッダファイルを読み込んでください.

```
#include <omp.h>
```

また, Fortran で OpenMP の実行時ライブラリ関数を用いる場合で, 関数からの返り値を利用する場合は, その値の型で関数名を宣言してください. 例えば, 並列実行時に同時に流れる処理の数を返す関数 `omp_get_num_threads()` を利用する場合は以下のように宣言します.

```
integer omp_get_num_threads
```

4.1.3 OpenMP プログラムの実行イメージ

図 3 の OpenMP プログラムを用いて, 並列実行のイメージを説明します. まず Fortran のプログラム例では, 以下の 2 つの OpenMP 指示行が用いられています.

```
!$omp parallel
```

```
!$omp end parallel
```

これらはそれぞれ parallel 指示文, end parallel 指示文と呼ばれます. Fortran の OpenMP プログラムでは, この parallel 指示文と end parallel 指示文で囲まれた範囲が並列に実行されます.

一方, C/C++ のプログラム例には以下の指示文しかありません.

```
#pragma omp parallel
```

これも parallel 指示文です. C 言語, 及び C++言語の OpenMP プログラムでは, parallel 指示文直後の構造ブロックが並列に実行されます.

次に, このプログラムを実行した際の出力例を見てみます.

Fortran

```
program hello
implicit none
integer omp_get_thread_num
print *, "並列世界へようこそ"
!$omp parallel
print *, "並列世界で処理中. 番号 ", omp_get_thread_num()
!$omp end parallel
print *, "さようなら"
end program hello
```

C 言語

```
  #include <stdio.h>
  #include <omp.h>
  main()
  {
    printf("並列世界へようこそ\n");
#pragma omp parallel
    {
      printf("並列世界で処理中. 番号 %d\n", omp_get_thread_num());
    }
    printf("さようなら\n");
  }
```

図 3: parallel 指示文を利用したプログラム例

図 3 のプログラムの実行例:

```
並列世界へようこそ.
並列世界で実行中．番号 1
並列世界で実行中．番号 2
並列世界で実行中．番号 3
並列世界で実行中．番号 0
さようなら.
```

この例では, 並列実行時に同時に進行させる処理数を 4 に設定しています.

実行結果より, parallel 指示文によって並列実行を指示された次の行だけが 4 回出力されていることが分かります.

```
      print *, "並列世界で処理中．番号 ", omp_get_thread_num()
```

このプログラムの実行イメージを図 4 に示します．ここで, プログラムを実行する処理の流れを**スレッド**と呼びます.

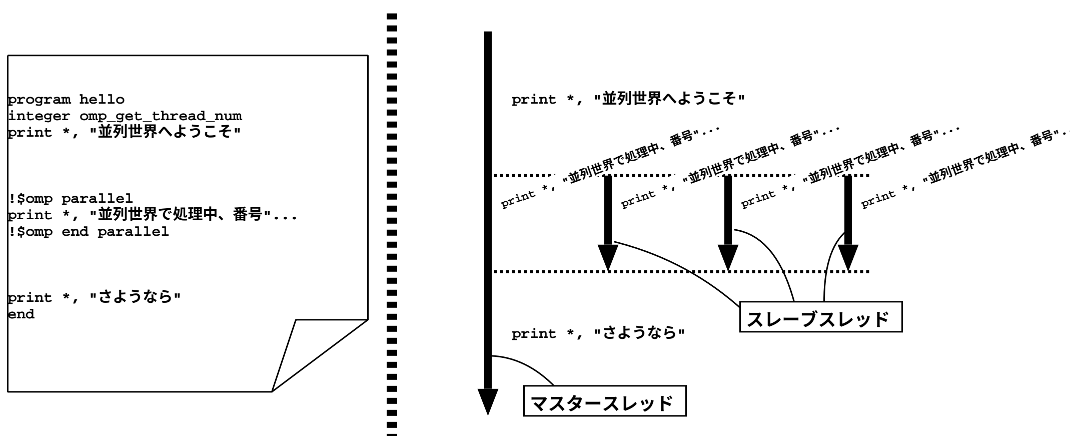

図 4: 図 3 の実行イメージ

プログラム中で, 複数のスレッドが存在してそれぞれ処理を進める範囲は**並列リージョン**と呼ばれます．一方, OpenMP プログラム中の並列リージョン以外の範囲は**逐次リージョン**と呼ばれ, 特定の 1 スレッドのみで処理されます．前述の例では, 「**並列世界で処理中．番号**…」を表示している print 文 (printf 文) だけが並列リージョンで, それ以外の部分が逐次リージョンです．一つの OpenMP プログラムに並列リージョンはそれぞれ何箇所あっても構いませんが, プログラムの冒頭と末尾は逐次リージョンとなります．この逐次リージョンを担当するスレッドは特に**マスタースレッド**と呼ばれます．一方並列リージョンでは, このマスタースレッドから**スレーブスレッド**と呼ばれるスレッドが生成され, 並列処理に参加します.

並列処理に参加するスレッドの数　並列処理に参加するスレーブスレッドの数としては, 実行時ライブラリ関数 `omp_set_num_threads()` で設定した数が最優先されます．プログラム中で

この関数が呼ばれるまでは, 環境変数 OMP_NUM_THREADS で設定した数が用いられます. もしこの環境変数も設定されていない場合, 通常は利用できるプロセッサ数が用いられます.

このプログラムでは, omp_get_thread_num() という OpenMP の実行時ライブラリ関数が用いられていますが, これはスレッド毎につけられたスレッド番号を取得する関数です. 4 個のスレッドがそれぞれ一回ずつこの関数を呼び出し, print 文を実行して各スレッドの番号を表示しています. そのため, 最後の数字を見るとこの表示命令を実行した順番が分かります. OpenMP では各スレッドがそれぞれ独立に実行を進めるので, 表示される数字の順番も毎回変わる可能性があります.

一般に OpenMP プログラムの実行の流れは以下のようになります.

1. マスタースレッドがプログラム冒頭の逐次リージョンを実行.
2. マスタースレッドが並列リージョンに到達するとスレーブスレッドを生成.
3. 各スレッドは OpenMP 指示文の内容に応じて割り当てられた処理を実行.
4. 割り当てられた処理を実行し終えたスレーブスレッドは消滅.
5. 割り当てられた処理を実行し終えたマスタースレッドだけが次の逐次リージョンを実行.
6. プログラム末尾まで 2〜5 を繰り返し.

すなわち, マスタースレッドがプログラムの最初から最後まで存在し続けるのに対しスレーブスレッドは並列リージョン毎に生成され, 並列リージョンが終了すると廃棄されます.

4.2 ループの並列化

簡単なループだと, OpenMP ではループの直前に parallel do 指示文 (C/C++ の場合は parallel for 指示文) を挿入するだけで並列化することができます. ここでは 図 5 のプログラムについて, OpenMP を用いた並列化の方法を説明します.

この例は, ベクトル x に実数 a を掛け, さらに実数 y を足したものをベクトル z に格納しています. このプログラムを 1 台のプロセッサで実行すると, ループのインデックス変数 i が 1 から 100 までプログラム通り 1 つずつ増加し, ループの中身を順に処理していきます.

一方, このプログラムに, 図 6 のように OpenMP 指示文を挿入すると, ループ i の複数回分の繰り返しを同時に処理できます. 例えばこのプログラムの並列実行に参加するスレッド数が 4 である場合, このプログラムは図 7 のように実行されます. すなわちループ i の繰り返しが 4 等分され, i = 1〜25 の計算をスレッド 0 が, i = 26〜50 の計算をスレッド 1 が, i = 51〜75 の計算をスレッド 2 が, i = 76〜100 の計算をスレッド 3 が, それぞれ分担し, 同時に処理を進めます[3]. これにより, 理想的には全体の処理時間を 1/4 に短縮できます.

このように parallel do 指示文 (parallel for 指示文) は, 直後の do ループ (for ループ) を分割し, 並列に実行させることを指示します.

[3] OpenMP ではマスタースレッドが何番のスレッドを担当するかについて規定はされていませんので, 必ず図 7 のようにマスタースレッドがスレッド 0 を担当するとは限りません.

Fortran

```
program ex1
implicit none
integer i
double precision z(100), a, x(100), y

do i = 1, 100
  z(i) = 0.0
  x(i) = 2.0
end do
a = 4.0
y = 1.0

call daxpy(z, a, x, y)

end program ex1

subroutine daxpy(z, a, x, y)
integer i
double precision z(100), a, x(100), y
do i = 1, 100
  z(i) = a * x(i) + y
end do
return
end
```

C 言語

```
#include <stdio.h>
#include <omp.h>
main()
{
  int i;
  double z[100], a, x[100], y;

  for (i = 0; i < 100; i++){
    z[i] = 0.0;
    x[i] = 2.0;
  }
  a = 4.0;
  y = 1.0;
  daxpy(z, a, x, y);
}

void daxpy(z, a, x, y)
double z[], a, x[], y;
{
  int i;
  for (i = 0; i < 100; i++)
    z[i] = a * x[i] + y;
}
```

図 5: 並列化前のプログラム例

Fortran

```
program ex1
implicit none
integer i
double precision z(100), a, x(100), y

do i = 1, 100
  z(i) = 0.0
  x(i) = 2.0
end do
a = 4.0
y = 1.0

call daxpy(z, a, x, y)

end program ex1

subroutine daxpy(z, a, x, y)
integer i
double precision z(100), a, x(100), y
!$omp parallel do
do i = 1, 100
  z(i) = a * x(i) + y
end do
return
end
```

C 言語

```
#include <stdio.h>
#include <omp.h>
main()
{
  int i;
  double z[100], a, x[100], y;

  for (i = 0; i < 100; i++){
    z[i] = 0.0;
    x[i] = 2.0;
  }
  a = 4.0;
  y = 1.0;
  daxpy(z, a, x, y);
}

void daxpy(z, a, x, y)
double z[], a, x[], y;
{
  int i;
#pragma omp parallel for
  for (i = 0; i < 100; i++)
    z[i] = a * x[i] + y;
}
```

図 6: ループを OpenMP で並列化した例

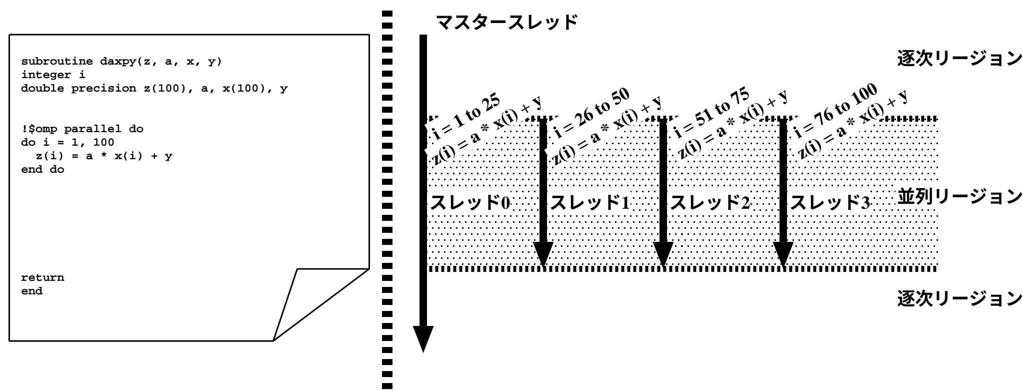

図 7: 図 6 の実行イメージ

並列化が簡単なループと難しいループ　この例のように簡単なループだと, parallel do 指示文 (parallel for 指示文) だけで並列化することができます. しかし, 全てのループで同様の手段が使えるわけではありません. 例えば以下のループを見てみます.

```
do i = 2, 100
   z[i] = z[i] + z[i - 1]
end do
```

このループでは, `z[i]` の計算に `z[i-1]` を用います. この `z[i-1]` は, ループ `i` の一つ前の繰り返しで計算しているので, ループ `i` が順に処理されないと正しい結果が得られません. ところがこのループ `i` を parallel do 指示文 (parallel for 指示文) で並列化すると, 先ほど例のようにループ `i` の繰り返しがスレッドに分割され, 複数回分を同時に処理するので, 処理の順番が変わります. そのため, このような再帰的な計算を行うループをうっかり parallel for 指示文 (parallel do 指示文) だけで並列化してしまうと, 計算結果に誤りが生じてしまいます. このようなループを並列化する技術はいくつか提案されていますがちょっと複雑なので, 次回の「OpenMP 入門 (2)」で説明することにします.

4.3　共有変数とプライベート変数

次に, OpenMP プログラムにおける変数の扱われ方を説明します.

基本的に, OpenMP のプログラムで用いられる変数は全スレッドに共有された領域に配置されます. 例えば図 6 のプログラムでは, 各スレッドが配列 `x`, 及び変数 `a`, `y` を参照し, 計算した結果を配列 `z` に代入しています. ここで, あるスレッドから見た `x[1]` と別のスレッドから見た `x[1]` は全く同じ要素です. そのため, スレッド 0 から参照される `x[1]` の値は, スレッド 1 やスレッド 2 から参照される `x[1]` の値と同じです. すなわち, 配列 `x` は各スレッドから**共有**されています. 変数 `a`, `y` も同じです.

また, 配列 `z` に代入した値は他のスレッドから参照できます. 例えばスレッド 0 が `a*x[1]+y`

を計算した結果が 3.0 だったとして, その値が `z[1]` に代入されると, 他のスレッドから見た `z[1]` の値も 3.0 になります. すなわち, 配列 `z` も各スレッドから共有されています.

一方, ループのインデックス変数 `i` を見てみます. `i` は各スレッドがそれぞれ自分の担当範囲の開始値から終了値まで一つずつ増やし, 対応する配列 `z` の要素の計算を行います. ここで, もし `i` が各スレッドから共有されていると, 同じ `i` に対して各スレッドがそれぞれ自分の担当範囲の値を上書きし, 参照します. 例えばスレッド 0 が `i=1` として処理を始めた後, 次に `i` を参照するまでの間にスレッド 1 が `i` の値を自分の初期値 26 に上書きしたとします. するとスレッド 0 は, `i` の値が自分の担当範囲の終了値 25 を超えているため, 以降の処理を行わず, ループを終了してしまいます. このように, ループのインデックス変数が各スレッドで共有されていると, 正しい計算が行えません.

そこで, 変数 `i` については各スレッドで共有するのではなく, 各スレッドがそれぞれ独自の値を持たせることにします. すなわち, 変数 `i` の値を格納する場所をスレッドの数だけ用意し, それぞれが干渉しないようにします. すると, スレッド 0 の `i` は 1 から 25 まで変化させ, スレッド 1 の `i` は 26 から 50 まで変化させることが出来ます.

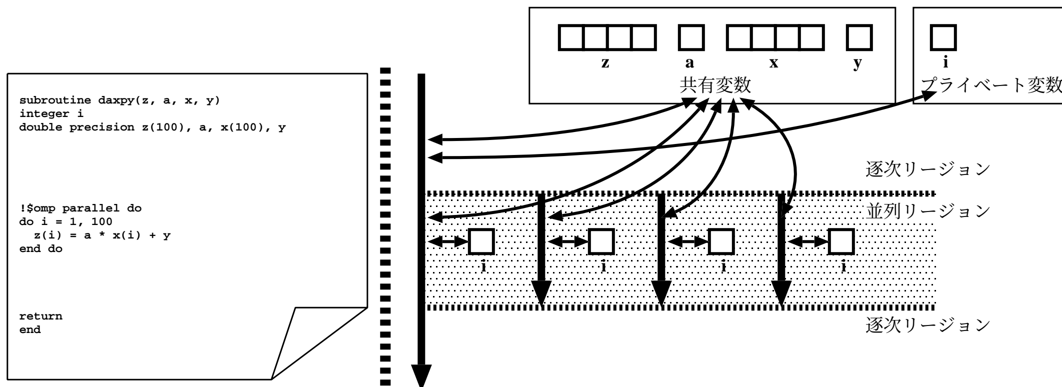

図 8: 共有変数とプライベート変数

この様子を 図 8 に示します. まず逐次リージョンでは, 全ての変数が一つずつ存在し, マスタースレッドからアクセスできます. これが並列リージョンに入ると, 配列 `x`, `z` 及び変数 `a`, `y` はそのまま残り, 全てのスレッドからアクセスされます. 一方, インデックス変数 `i` はスレッド毎に独自の場所が用意されます. 各スレッドは自分の `i` にのみアクセスし, 他のスレッドの `i` にはアクセスできません.

このように OpenMP プログラムでは, 扱うそれぞれの変数について各スレッドから共有させたり, 各スレッドにそれぞれ独自の値を持たせたりすることが出来ます. 各スレッドから共有させる変数を**共有変数**, 各スレッドにそれぞれ独自の値を持たせる変数を**プライベート変数**と呼びます.

ある変数を共有変数とする場合は, 以下のように指示文の後に shared 指示節を追加します.

```
!$omp parallel shared(a)
```

一方, ある変数をプライベート変数とする場合は, 以下のように指示文の後に private 指示節を追加します.

```
!$omp parallel private(a)
```

ある指示文の後に shared 指示節と private 指示節を両方追加しても構いませんが, 同じ指示文の中で, 同じ変数を shared 指示節と private 指示節の両方に入れることはできません. また, 以下のように指示節の中に複数の変数を ,(カンマ) で区切って記述することにより, 複数の変数の扱いを指示することができます.

```
!$omp parallel shared(a, b, c) private(d, e)
```

ところで, 図 6 のプログラムでは parallel do 指示文 (parallel for 指示文) に指示節がありませんでしたが, 共有変数とプライベート変数が正しく区別されていました. これは, OpenMP の暗黙の了解によります. OpenMP プログラムでは各スレッドが同じ記憶空間を共有するので, 何も指示のない変数については基本的に共有変数となります. ただし, 並列化されたループのインデックス変数のような, 明らかにプライベート変数でなければならないものは, 特に指示しなくてもプライベート変数となります. 図 6 のプログラムを, 共有変数とプライベート変数の指示節を省略しないよう書き直すと, 図 9 のようになります.

Fortran

```
subroutine daxpy(z, a, x, y)
integer i
double precision z(100), a, x(100), y
!$omp parallel do shared(z, a, x, y) private(i)
do i = 1, 100
  z(i) = a * x(i) + y
end do
return
end
```

C 言語

```
void daxpy(z, a, x, y)
double z[], a, x[], y;
{
  int i;
#pragma omp parallel for shared(z, a, x, y) private(i)
  for (i = 0; i < 100; i++)
    z[i] = a * x[i] + y;
}
```

図 9: 指示節を省略せずに記述したプログラム

実際は, OpenMP プログラムではほとんどの変数が共有変数となるので, shared 指示節が必要となることはあまりありません. 一方 private 指示節は, 少し複雑なプログラムでしばしば

用いられます. 例えば C/C++ の OpenMP 規格では, 並列化の対象でないループのインデックス変数は共有変数となります[4]. 例えば 2 重ループの外側ループを並列化した場合, 何も指示しなければ内側ループのインデックス変数が共有変数となります. しかし並列実行時に内側ループのインデックス変数が共有変数のままだと, 各スレッドが互いに内側ループのインデックス変数を上書きし合うために, 内側ループを正しく実行することができません. そこで, 内側ループのインデックス変数をプライベート変数とするよう, 指示節を追加する必要があります. 2 重ループを並列化した例として, 行列とベクトルの積を求める OpenMP プログラムを図 10 に示します.

Fortran

```
subroutine matvec(a, x, y)
integer i, j
double precision a(100, 100), x(100), y(100)

!$omp parallel do private(j)
do i = 1, 100
  do j = 1, 100
    y(i) = a(j, i) * x(i)
  end do
end do

return
end
```

C 言語

```
void matvec(a, x, y)
double a[][100], x[], y[];
{
  int i, j;

#pragma omp parallel for private(j)
  for (i = 0; i < 100; i++)
    for (j = 0; j < 100; j++)
      y[i] += a[i][j] * x[i];
}
```

図 10: OpenMP によるベクトルと行列の積

4.4 リダクション変数

次に, 図 11 に示す配列の総和の計算を行うループの並列化を行います.

まず, 図 6 と同様に parallel do 指示文 (parallel for 指示文) をループ i の前に挿入して並列

[4]ちなみに Fortran の OpenMP 規格では, 多重ループの外側ループを並列化した場合, それより内側のループのインデックス変数は何も指示しなければ全てプライベート変数となります. これは, C 言語や C++言語に比べて Fortran の方がプログラムの構造を解析しやすく, 内側ループのインデックス変数を正しく判断することができるためです.

Fortran

```
function total(x)
integer i
double precision t, total, x(100)
t = 0.0
do i = 1, 100
  t = t + x(i)
end do
total = t
end
```

C 言語

```
double total(x)
double x[];
{
  int i;
  double t;
  t = 0.0;
  for (i = 0; i < 100; i++)
    t += x[i];
  return t;
}
```

図 11: ベクトルの総和計算ループ (並列化前)

化した場合を考えます．すると, スレッドの数が 4 の場合は各スレッドが繰り返しを 25 回分ずつ分担し, 並列に計算します．このように並列した場合, 何も指示しなければ, 図 11 の配列 `x` と変数 `t` は共有変数となります．もちろん, ループのインデックス変数である `i` はプライベート変数となります．

ここで, 変数 `t` の値の変化を追ってみましょう．例えば, 配列 `x` の各要素の値が全て 1.0 だったとします．これを並列ではなく逐次で実行した場合, 配列 `x` の各要素が順に足し込まれていき, 最終的な `t` の値は 100.0 となります．

一方, これを並列実行した場合はどうでしょう．一見, 何の問題もなく実行できそうな気がしますが, 実は大きな落とし穴があります．これを説明するために, 各繰り返しの処理を以下のようにさらに細かく分解して見てみます．

操作 1: `t` の値を参照する．

操作 2: `x[i]` の値を参照する．

操作 3: これらの和を計算する．

操作 4: 計算した結果を `t` に格納する．

逐次で実行した場合は, 図 12(a) のようにこれらの各操作が順に行われていきます．一方並列に実行する場合, 図 12(b) のように各スレッドはそれぞれ自分が担当する範囲についてこの

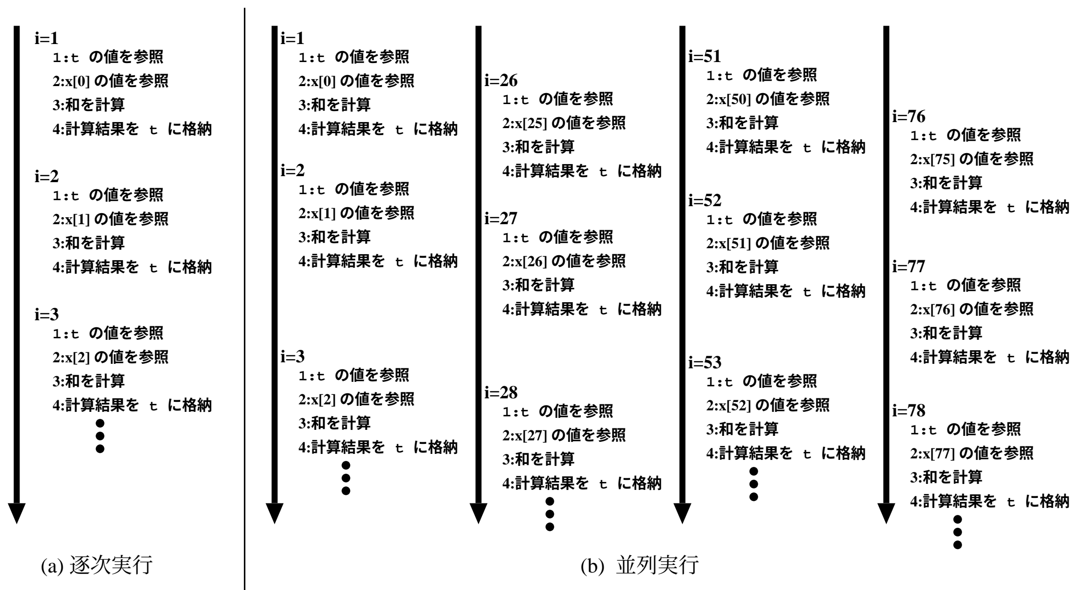

図 12: 共有変数とプライベート変数

一連の処理を並列に実行していきますが, OpenMP では, 何も指定しないと, あるスレッドが一連の処理を終了するまで他の処理が待っている, という保証はありません. そのため, あるスレッドが操作 1 を実行してから操作 4 を実行するまでの間に, 他のスレッドが操作 1 を実行する可能性があります. すると, どちらのスレッドからも変数 `t` の値として同じ値を取得します. 例えば `t` が 12.0 の時にスレッド 0 とスレッド 1 がそれぞれこの値を取得したとします. `x[i]` の値は全て 1.0 なので, それぞれのスレッドで `t` との和を計算した結果は 13.0 になります. 各スレッドはこの結果を `t` に格納しますが, 後から格納するスレッドは先に格納されている値を上書きするだけですので, 結局 `t` の値は 13.0 となります. すなわち, 片方の計算結果のみが反映されて, もう片方の計算結果は廃棄されることとなります. その結果, 最終的な計算結果も 100.0 より少ない値となります.

このような困った事になる理由は, 共有変数に対して各スレッドが同時にアクセスを行うためです. これを回避する手段として OpenMP には, 二つの方法が用意されています. 一つは**排他制御**と呼ばれる方法です. これは, 指定した一連の処理について同時に実行できるスレッドの数を 1 つに制限するものです. 排他制御されている処理をあるスレッドが開始すると, 指定された一連の処理をそのスレッドが終了するまで, 他のスレッドはその処理を開始できません. 図 11 の例だと `t = t + x(i)` の処理を排他制御することにより, あるスレッドが `t` を参照してから計算結果を `t` に格納するまで他のスレッドは `t` を参照できなくなるので, 正しい順序で計算が行えます. しかしこの方法では, ループ `i` の中で同時に 1 つのスレッドしか処理を進めることができないので, せっかく並列化しても処理時間は短縮できません.

この問題を解決するもう一つの方法は, **リダクション変数**を利用するものです. リダクション変数とは, ベクトルの総和や最大値の探索等の “リダクション計算” と呼ばれる計算を並列

に実行するために用意されている, OpenMP における変数の扱われ型です.

リダクション変数は, プライベート変数と共有変数の両方の性質を持っています. まず, 並列ループが開始され, 各スレッドがループを実行している間はプライベート変数として扱われます. そのため, 各スレッドはそれぞれのリダクション変数に対して独自の値を持ちます. その後全スレッドが並列ループの処理を終了すると, 各スレッドのリダクション変数の値は一つにまとめられ, 共有変数のようにマスタースレッドの対応する変数に格納されます. リダクション変数を最後にまとめる際の計算方法には, 和, 積, 最大値等が用意されており, 指示節の中で指示します.

リダクション変数としたい変数がある場合, parallel for 指示文に reduction 指示節を追加します. リダクション変数を用いて図 11 のプログラムを並列化したものが図 13 です. ここで, reduction 指示節 `reduction(+:t)` では, + (和) のリダクション変数として変数 `t` が指示されています. もしリダクション変数が複数ある場合は `reduction(+:t,u,v)` のように ,(カンマ) で区切って列挙します.

このように変数 `t` を "和" のリダクション変数とした場合, 各スレッドは `t` をプライベート変数のように扱い, 計算を進めます. すなわち, `t` には各スレッドの担当する範囲で `x[i]` の要素が足し込まれます. そして全スレッドがループ `i` の担当範囲を実行し終わると, 各スレッドの `t` の値の総和が計算され, マスタースレッドの `t` に格納されます. リダクション変数を用いる方法では, ループ `i` の中身を並列に実行することができるため, 排他制御による方法と違い, 処理時間を短縮することができます.

5　OpenMP プログラムのコンパイルと実行

本節では, 当センターのスカラー並列サーバと汎用 UNIX サーバで OpenMP プログラムを翻訳し, 並列実行するための手順を紹介します. また, 並列化したことによる性能向上の程度を算出する方法についても説明します.

5.1　情報基盤センターの共有メモリ型並列計算機

当センターで OpenMP を利用できる 2 台の共有メモリ型並列計算機について, 特徴を紹介します.

5.1.1　スカラー並列サーバ COMPAQ GS320

スカラー並列サーバ COMPAQ GS320 は, 中〜大規模の計算を行なう汎用の計算機として導入しました. GS320 には Alpha 21264 (731MHz) という高速な CPU が用いられています. この CPU はスカラー演算性能が非常に高く, 昨年米国で行なわれたゲノム解析でも主力として活躍しました. 特にベクトルプロセッサが不得手とするアプリケーションで威力を発揮します. また, 当センターの GS320 には 64GByte の主記憶が搭載されており, 1 つのユーザープログラムで利用できる最大の記憶領域は, 並列, 非並列を問わず 16GByte です.

Fortran

```
function total(x)
integer i
double precision t, total, x(100)
t = 0.0
!$omp parallel do reduction(+:t)
do i = 1, 100
  t = t + x(i)
end do
total = t
return
end
```

C 言語

```
  double total(x)
  double x[];
  {
    int i;
    double t;
    t = 0.0;
#pragma omp parallel for reduction(+:t)
    for (i = 0; i < 100; i++)
      t += x[i];
    return t;
  }
```

図 13: ベクトルの総和計算ループ (並列化後)

スカラー並列サーバ
COMPAQ GS320

汎用 UNIX サーバ
富士通 GP7000F model 900

GS320 には OpenMP に対応した Fortran と C 言語のコンパイラが用意されています. OpenMP に対応した C++ 言語のコンパイラは用意されていません. また, GS320 には非並列の C 言語及び Fortran プログラムを自動的に並列化するコンパイラも提供されます. このコンパイラはプログラムに自動的に OpenMP 指示文を挿入することにより並列化します. 並列化後の OpenMP プログラムを出力することもできますので, OpenMP の勉強等にも利用できます. GS320 で利用できる並列プログラムを記述する他のモデルとしては, 共有メモリモデルではより低レベルなマルチスレッド, 分散メモリモデルでは MPI, PVM, HPF がサポートされています. これらを用いることにより, スーパーコンピュータ VPP5000 や汎用 UNIX サーバ GP7000F model 900 等とソースプログラムを共有することができます.

Compaq Tru64 UNIX もしくは Digital UNIX が動作している Alpha CPU 搭載計算機でコンパイルされた実行形式ファイルのほとんどを GS320 でそのまま実行することができます. そのため, 研究室の Alpha サーバで開発したプログラムを問題規模に応じて GS320 で実行する等, 柔軟な利用が考えられます.

なお, スカラー並列サーバを利用するためには, 事前に本センターの汎用 UNIX サーバ `kyu-cc.cc.kyushu-u.ac.jp` にログインし, 以下のように利用登録のコマンド `touroku` を実行しておく必要があります.

```
kyu-cc% touroku kyu-ss
Password:                    ← kyu-cc のパスワードを入力
   . . .
kyu-cc%
```

その後, 以下のアドレスでスカラー並列サーバに直接ログインすることができます.

`kyu-ss.cc.kyushu-u.ac.jp`

5.1.2 汎用 UNIX サーバ 富士通 GP7000F model 900

汎用 UNIX サーバ 富士通 GP7000F model 900 は, 科学技術計算だけでなくデータ解析や画像処理など様々な分野を対象とした並列計算機です.

GP7000F model 900 は, SPARC64-GP (300MHz) というプロセッサを 64 台搭載しています. このプロセッサはサン・マイクロシステムズ社の SPARC マイクロプロセッサと互換性がある上, 動作する OS(オペレーティングシステム) もサン・マイクロシステムズ社の Solaris 7 ですので, 研究室等に導入されている SPARC を用いたワークステーションと, プログラムを共有することができます. さらに, 当センターの GP7000F model 900 には 64GByte の主記憶が搭載されており, 1 つのユーザープログラムで利用できる最大の記憶領域は, 並列, 非並列を問わず 32GByte です.

GP7000F model 900 にも OpenMP に対応した Fortran, C 言語及び C++言語のコンパイラ, さらに, 非並列の C 言語及び Fortran プログラムを自動的に並列化するコンパイラが提供されます. また, 他に並列プログラムを記述するモデルとしては, 共有メモリ型のマルチスレッド, 及び分散メモリ型の MPI がサポートされており, VPP5000 や GS320 等とソースプログラムを共有することができます.

汎用 UNIX サーバーへは, 以下のアドレスでログインすることができます.

```
kyu-cc.cc.kyushu-u.ac.jp
```

5.2 情報基盤センターでの OpenMP プログラムのコンパイル

現在, ほとんどの共有メモリ型並列計算機上で OpenMP 指示文を処理することの出来るコンパイラが提供されています. OpenMP の規格は特定の機種やコンパイラ等に依存しないので, OpenMP で書かれた並列プログラムは, 基本的にどのコンパイラでも利用することができます. しかしながら, コンパイラの利用法に関しては OpenMP で何も規定されていないので, 機種によって若干の違いがあります.

5.2.1 GS320 でのコンパイル

GS320 では, Fortran 及び C 言語のコンパイラに`-omp` オプションをつけることにより, OpenMP 機能を有効にしてプログラムを翻訳することができます.

なお, GS320 では拡張子が `.f90` のファイルを Fortran の自由形式として扱います. 一方, 拡張子が`.f` もしくは `.for` であるファイルは Fortran の固定形式として扱われます.

以下の例では, ソースプログラムを翻訳・結合した結果生成される実行形式ファイルの名前を `-o` オプションで `example` と指定しています.

```
kyu-ss% f90 -omp example.f90 -o example
```

```
kyu-ss% cc -omp example.c -o example
```

さらに GS320 では, `-check omp_bindings` (C コンパイラでは `-check_omp`) オプションをつけることにより, 特定の OpenMP 指示文を実行時にチェックする機能を有効にすることができます. このオプションが設定されていない場合, OpenMP 指示文に不正な構造があっても発見できません.

5.2.2 GP7000F でのコンパイル

GP7000F では, Fortran, C 言語, 及び C++言語のコンパイラに`-KOMP` オプションをつけることにより, OpenMP 機能を有効にしてプログラムを翻訳することができます.

なお, GP7000F では拡張子が `.f90` もしくは `.f95` のファイルを Fortran の自由形式として扱います. 一方, 拡張子が`.f` もしくは `.for` であるファイルは Fortran の固定形式として扱われます.

以下の例では, ソースプログラムを翻訳・結合した結果生成される実行形式ファイルの名前を `-o` オプションで `example` と指定しています.

Fortran:

```
kyu-cc% frt -KOMP example.f90 -o example
```

C:

```
kyu-cc% fcc -KOMP example.c -o example
```

C++:

```
kyu-cc% FCC -KOMP example.cc -o example
```

さらに Fortran の OpenMP プログラムを翻訳する際, 以下に示すようなオプションも利用できます.

オプション	意味
-Kspinwait	-Kspinwait オプションが指定されると, マスタースレッド以外のスレッドを担当する CPU が, 逐次リージョンでも CPU 時間を消費します. この場合, 並列リージョンに入るとすぐに割り当てられた処理を開始できるので, 経過時間を短縮できます. 本オプションは-KOMP オプションと同時に, 主プログラムの翻訳時に指定する必要があります. -Kspinwait および -Knospinwait オプションの指定を省略した場合は, -Kspinwait となります.
-Knospinwait	-Knospinwait オプションが指定されると, マスタースレッド以外のスレッドは逐次リージョンでは休止状態となり, そのスレッドを担当する CPU は別のプログラムに制御を移します. そのため CPU 時間の消費は少なくなります. しかし並列リージョンに入る際, 各スレッドに再度 CPU を割り当てる時間が必要なので, 経過時間が長くなります. 本オプションは -KOMP オプションと同時に, 主プログラムの翻訳時に指定する必要があります.
-Kthreadstacksize=N	-Kthreadstacksize=N オプションは, スレッド毎のスタック領域の大きさを K バイト単位で指定します ($1 \leqq N \leqq 2147483647$). スタック領域とはサブルーチンや関数の中でだけ定義され, 利用されるローカル変数を格納する領域です. この領域が不足するとプログラムが異常終了するので, 十分な大きさを確保して下さい. 本オプションは -KOMP オプションと同時に, 主プログラムの翻訳時に指定する必要があります. このオプションは, 後述する環境変数 THREAD_STACK_SIZE より優先されます.

5.3 実行

OpenMP プログラムをコンパイルして生成された実行ファイルは OpenMP の有効, 無効に関わらず通常のプログラム同様, コマンドとして実行できます. また, OpenMP が有効であれば, そのプログラムは並列に実行されます.

並列実行に参加するスレッドの数は環境変数 OMP_NUM_THREADS で指定できます. ただしプログラム中で実行時ライブラリ関数 omp_set_num_threads() を用いてスレッド数が変更された場合, その数が優先されます.

また, 汎用 UNIX サーバー GP7000F では, スレッド毎のスタック領域の大きさを指定する環境変数 THREAD_STACK_SIZE が用意されています. 指定できる値は K バイト単位で 1 以上 2147483647 以下となっています.

5.4 並列化による効果の算出方法

プログラムを並列化することによる効果は, 経過時間を計測すると算出することができます. 経過時間とはプログラムの実行を開始してから終了するまでに経過した時間のことであり, UNIX の timex コマンドで計測できます.

例えば以下の例では, 同じプログラムについて OpenMP を無効にして翻訳したもの (ex1-seri) と, OpenMP を有効にして翻訳したもの (ex1-para) をそれぞれ timex コマンドにかけて, 経過時間を計測しています. なお, この例は実行時の環境変数 OMP_NUM_THREADS の値を 2 とし, 汎用 UNIX サーバで実行した際のものです.

```
kyu-cc% timex ./ex1-seri

real       15.93
user       10.94
sys         4.40

kyu-cc% timex ./ex1-para

real       10.54
user       10.99
sys         4.46
```

ここで real の項目に表示されている数字が経過時間 (秒) です. ちなみに user はユーザープログラムそのものを実行するために消費した CPU 時間, sys はプログラムの実行に必要なシステム処理のために消費した CPU 時間です.

OpenMP が無効の場合のプログラムは並列実行されないので 1CPU だけで実行が行われ, 15.93 秒を要しました. 一方 OpenMP が有効の場合, OMP_NUM_THREADS が 2 なので 2CPU で並列実行され, 10.54 秒を要しました. この場合, 処理速度の向上率は 15.93/10.54 = 約 1.51 倍と計算できます. 一般に並列実行による速度向上率は, 並列実行に参加する CPU 台数が上限となります[5]. プログラムのチューニング時の参考にしてください.

ただし, 経過時間は実行する時のシステムの状態に大きく影響されます. システムが混雑している場合は他のプロセスと CPU を取り合うために経過時間が長くなり, 正確な速度向上率が計算できません. そこで当センターでは, 正確な速度向上率を計算するために, 汎用 UNIX サーバ GP7000F (ホスト名: kyu-cc.cc.kyushu-u.ac.jp) に sc8 及び sc32 という名前の専用キューを用意しています. これらのキューは当センターの他のキューと同様, qsub コマンドを用いてジョブを投入することができます. ただし, これらのキューでは同時に一つのジョブしか実行を許さないため, 一旦処理が開始されると, 他のプロセスに邪魔されること無く処理を進めることができます. より正確な速度向上率を計算して, チューニングに役立てたい方は試してみてください.

参考文献

[1] Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D. and McDonald, J.: *“Parallel Programming in OpenMP,”* Morgan Kaufmann Publishers, 2000.
残念ながら，2001 年 9 月現在，OpenMP に関する入門書・解説書はこれくらいしかないようです．日本語による書籍もまだ見当たりません．

[2] Geist, A., Beguelin, A., Dognarra, J., Jiang, W., Manchek, R. and Sunderam, V.: *“PVM: Parallel Virtual Machine—A Users’ Guide and Tutorial for Networked Parallel Computing,”* The MIT Press, 1994.
筆者の知る範囲では，PVM に関する日本語の書籍も見当たらないようです．情報基盤

[5] プログラムのアルゴリズムやデータの大きさによっては, 速度向上率が CPU 台数を超える場合があります.

センターの計算機で PVM を利用する方法は以下の Web ページを参照してください.
http://www.cc.kyushu-u.ac.jp/scp/system/library/MPL/PVM.html

[3] High Performance Fortran Forum, 高度情報科学技術研究機構, 富士通, 日立製作所, 日本電気, 日電, NEC : 「High Performance Fortran2.0 公式マニュアル」, シュプリンガー・フェアラーク東京, 1999 年.
HPF バージョン 2.0 の仕様, 及び拡張仕様の日本語訳です. 情報基盤センターの計算機で HPF を利用する方法は以下の Web ページを参照してください. http://www.cc.kyushu-u.ac.jp/scp/system/library/Fortran/HPF.html

[4] OpenMP Architecture Review Board: *"OpenMP Fortran Application Program Interface,"* http://www.openmp.org/specs/mp-documents/fspec10.pdf, October 1997.
いわゆる Fortran 用 OpenMP バージョン 1.0 の規格書です (解説ではありません).
http://www.openmp.org/specs/ からは，上記以外にも，Fortran 用のバージョン 2.0 や，C(C++) 用の各バージョンの規格書を入手することができます．また，これらの一部については，http://pdplab.trc.rwcp.or.jp/pdperf/Omni/spec.ja/home.html から新情報処理開発機構 (RWCP) および 富士通株式会社による日本語訳を入手することもできます.

[5] Pacheco, P. S.: *"Parallel Programming with MPI,"* Morgan Kaufmann Publishers, 1997.
この日本語訳は，P. パチェコ 著/秋葉 博 訳：「MPI 並列プログラミング」として，培風館より 2001 年 7 月に出版されています．情報基盤センターの計算機で MPI を利用する方法は以下の Web ページを参照してください. http://www.cc.kyushu-u.ac.jp/scp/system/library/MPL/MPI.html

[6] 富士通株式会社：「UXP/V VPP Fortran プログラミングハンドブック V20 用」，富士通株式会社，1999 年.
VPP Fortran の入門書です．本センターの利用者登録が済んでいれば，本センターの web サイトで閲覧することができます．情報基盤センターの計算機で MPI を利用する方法は以下の Web ページを参照してください. http://www.cc.kyushu-u.ac.jp/scp/system/library/Fortran/VPP_Fortran.html