連載講座: 高生産並列言語を使いこなす (1)

田浦健次朗
東京大学大学院情報理工学系研究科, 情報基盤センター (兼務)

1 はじめに

1.1 背景

現在, 並列処理を行うためのプログラミング方法としては, 1 ノード内の共有メモリを前提とした並列処理に限れば OpenMP [3, 5, 21, 22] が, 複数ノードにまたがる並列処理に対しては MPI [13, 18, 23] が最も広く用いられている. それらの「成功」と関係していると思われる特徴をあげると以下などが挙げられよう.

- 新しいプログラミング言語ではなく, 既存の主流言語 (C 言語や Fortran) の小規模な拡張やライブラリとして提供されている.
- 並列処理の記述方法はユーザフレンドリというよりも, マシンフレンドリ—現在のマシンや OS で提供されている機能に, 素直にマッピングできるよう—に設計されている.
- 元になっている言語—C 言語や Fortran—なども, 同様の性格を持つ言語であるため, 全体としてそれらはユーザから見れば, プログラムの記述は面倒でも性能を理解しやすく, 結果として高い性能が得られることが多い.

一言で言うならばそれらの処理系はプログラマに「現実 (計算機の姿) をありのまま見せる」「できもしない約束をしない」ことを指針に設計されており, その結果, 「約束した通りのこと」を達成している処理系が存在している. そのような処理系が存在しているため, 並列プログラミングの初心者が並列プログラミングについて学べる情報源も, それらに集中している.

一方で, それら現状で主流の並列処理系で書かれた並列プログラムは, 少なくとも逐次プログラムと比べれば記述自体が困難・面倒であり, 可読性や保守性が悪い. 各プログラムがタスクやデータの, 計算機へのマッピングを細かく指定しているため, 複数のプログラムを組み合わせることが困難であり, プログラム部品の再利用性に乏しい. 仮にこれらを「性能のため」と言ってある程度看過するとしても, 同様の理由によりそれらの処理系では計算機の世代が変わる際に大きなプログラムの変更を強いられる. これは性能のために努力を厭わないユーザにとっても, 自分の投資の寿命を短くする由々しき事態である.

1.2 高水準言語の必要性と好機

このような現状が変わって欲しい, または変わって行くべきである— すなわち並列プログラミングが「高水準化」「高生産性化」すべきである— というのは多くの人が自然に描く希望・議題である. 高水準並列言語は非常に古くからある課題で, 長年研究されているが, 現在あるいは近い将来の計算機アーキテクチャの傾向を鑑みるに, そのユーザにとっての必要性および処理系を作る側にとっての好機は, ますます増加していると考えられる. 以下にいくつかの理由を述べる.

- **ノード内 CPU のクロック向上の停止, マルチコア化により, ソフトウェアによる並列化はますます日常的となって行く.**

  そのためエントリコストの低いプログラミング, 並列化のための記述が少ない (記述が楽になるというのみならず, 可読性・保守性を損なわない) 並列化の方法に対する必要性が強まる.

- **ノード内のメモリも非均一化 (NUMA 化)・階層化している.**

  そのためノード内の並列処理であっても, メモリアクセスの局所性を考慮したプログラミングが重要になる. そこで, ノード内の並列処理と複数ノードをまたがった並列処理を統一的な処理系で記述したいという欲求・必要性が増す. 現状しばしば実践されている Flat MPI による記述は, ノード内のコア数が増えるに伴い, メモリ使用量やノード内通信性能の不利益が大きくなるであろう. 理想的には, ノード内・ノード間の並列性を区別しない統一的な記述から, ノード内共有メモリ・ノード間ネットワークへ適切に通信をマッピングできる処理系の必要性が高まる.

- **GPU や, SIMD 命令のワイド化など, 逐次性能よりも高並列コードのスループットを指向したプロセッサが登場している** [11].

  これらのプロセッサや命令は普通に C 言語や Fortran でプログラムを書いていれば利用できるものではなく, 「比較的低水準な言語で書いておけば得てして速い」という単純な図式は崩れていく可能性がある. また, SIMD や GPU はプロセッサの世代が変われば再プログラミングが必須なのが現状で, どんどん複雑になる計算機のために, ほとんどの人がもはや楽しいとも思わない様なプログラムの「書き直し」を強いられる機会が増えて行く. 高水準言語は本来そのような場面を好機として活躍できるはずのものである.

- **多くの実アプリケーションで, プロセッサのピーク性能はどうせ出ない.**

  従ってここでも, C 言語や Fortran 言語など, 比較的低水準な言語で書いておけば速い, という単純な図式ではなくなっている. アーキテクチャの特性を考慮して適切に設計・実装された高級言語, ないし分野特化型言語が, 低水準な言語に性能上も肉薄ないし凌駕できる可能性がある.

実際多くの複雑なアプリケーションでは, 分岐や, 不規則なメモリアクセスによって性能が律速される. 比較的単純な計算カーネルでも, メモリアクセスのバンド幅が律速になる場合が多い. 例えば行列・ベクトル積ではメモリから取り出された行列の要素は, 一度ずつの乗算と加算にしか使われないため, x FLOPS (倍精度) の性能を出すためには, $x/2 \times 8 = 4x$ バイト/秒のメモリバンド幅が必要である. 言い換えれば, バイト/FLOPS 値 $= 4$ が, 行列ベクトル積でプロセッサのピーク性能を出すのに必要なメモリバンド幅である. 現状のスカラープロセッサでは, この値はせいぜい 1.0 以下である.

- **計算を速くすることは第一義的な目的ではない.**

  おそらくこれが最も基本的かつ重要な点である. ユーザが最小化したい「目的関数」は, 結果が出るまでの, プログラム開発や試行錯誤を含めた合計時間であり, プログラムの実行時間ではない. 逐次の計算においてはそのために多くの人が C や Fortran 言語による高速化, Python や Perl シェルスクリプトによるスクリプティング, MATLAB や Excel などの対話的なツールを, 「適材適所」組み合わせて計算を行っている.

並列処理においてもそのような適材適所が行われてしかるべきで, そうなっていないのは, そもそも処理系まで含めて使える選択肢が少ないこと, それらに関する情報源が少ないことなど, 複数の理由が考えられる. しかし最近では, 本連載で紹介する言語の他にも, Java concurrency framework [12] や, fork/join フレームワーク [16] (JDK 7 に導入予定) などのマルチコア環境用の処理系, CUDA [31] や OpenCL などの GPU 用の処理系, Hadoop [15, 29] などのデータ処理用の処理系など, 用途や環境に合わせて多くの処理系が利用可能になってきている. また, 大規模な並列処理用の言語としては米国 DARPA のプログラムで High Productivity Computer Systems (HPCS) プロジェクトが遂行されてきた. 同プロジェクトでは, IBM により X10 [26, 30], Cray により Chapel [4, 8], Sun Microsystems (当時) により Fortress [1, 10] などの, 高生産性を指向した並列言語が提案・実装された. そして HPCS の後継とでも言うべき, Ubiquitous High Performance Computing (UHPC) Program でも, 「今日のシステムよりもプログラミングが簡単である (easier to program than current systems)」ことがプロジェクトの目標の一つに挙げられている.

1.3 本連載について

1.3.1 動機

このような背景を鑑み, 本連載では MPI や OpenMP などの, すでにお馴染みの並列処理の枠組み以外の, 並列処理方法に関する情報を提供し, それらを共通の座標軸に沿って比較することを試みる. もちろんすべての興味深い処理系を網羅することは筆者の能力からして困難であるし, そもそも同列に比較することにさしたる意味がない言語も多い. ここでは以下のような基準により処理系を選択した.

- 特定分野に特化しない, 汎用的な並列プログラミングのための言語ないしライブラリである.
- 高性能計算—いわゆる HPC—を目標・視野にいれて設計されている.
- 今日処理系が実際に入手可能で, 苦労なく構築・設定できる程度に成熟している. 特に, 実際に Linux クラスタ, 特に HA8000 で実行できる.
- 主観を混じえて, 実用性や将来性などが高く, 知っておく教育的・実践的価値が高い.

1.3.2 取り扱う言語・ライブラリ

実際には以下に挙げる 5 つの言語やライブラリを紹介する. なお, 以下で紹介するもののうち TBB だけが新しい並列言語ではなく, C++のライブラリである. しかしいちいち「言語またはライブラリ」のように書くのも煩わしいので, 混乱のおそれがない場合はすべて並列言語という呼び方に統一する.

- **Cilk [6, 14]:** MIT の Leiserson らによる研究プロジェクトの成果で, C 言語に分割統治型の並列性を記述する構文を加え, それを効率よく実装した言語である. 実行プラットフォームとしては, 事実上共有メモリ環境が前提である. その後 Cilk Arts というスタートアップから製品化され, 現在は Intel Cilk Plus [7] という名前で, Intel Parallel Building Blocks [24] の一部としてリリースされている. 本連載では MIT のプロジェクトサイト [6] から配布されている Cilk 5.4.6 を用いている
- **Intel Threading Building Blocks (TBB) [25, 27]:** これも Intel Parallel Building Blocks の一部であるが, 拡張構文ではなく C++のライブラリとして実装されている. ループの並列化や, Cilk 同様の分割統治型の並列性を, メソッド呼出として記述できる. 実行プラットフォームとしては, 事実上共有メモリ環境が前提である.
- **Unified Parallel C (UPC) [9, 28]:** 分散メモリ環境を共有メモリのようにプログラミングできることを指針に設計されている. C 言語の構文を拡張する形で設計されている. ノード間で共有された大域変数や, 複数ノードにまたがる大域的な配列, そのデータ分割の指示などの構文的な拡張と, 遠隔にあるデータを通常のデータを同様の構文でアクセスできる機能を導入している. 並列実行のモデル自体は MPI に近い SPMD 型で, プログラム開始時から一定数のプロセスが実行を開始するというモデルである.
- **Chapel [4, 8]:** Cray が設計・実装を進めている高生産を指向した, 分散メモリ型並列計算機用の言語である. UPC 同様, 遠隔ノードにあるオブジェクトや配列を, ノードの境界を意識することなくアクセスできる. 並列実行のモデル

はMPIやUPCと異なっており, 1つのスレッドが実行を開始し, 実行中に動的なスレッド生成を行うモデルである.

- **X10 [26, 30]:** IBMが設計・実装を進めている高生産を指向した, 分散メモリ型並列計算機用の言語である. 並列実行やデータのモデルはChapelに近い. ただしChapelと異なり, 遠隔のデータを無意識にアクセスすることはできない (コンパイル時もしくは実行時エラー). 任意の文や式の実行場所を移動させる簡便な構文があり, それを用いてデータを参照できる場所へ移動してからデータを参照する, というスタイルでプログラムを書かせる.

また比較のために, 多くの読者に馴染みの深いと思われるMPI, OpenMPや, ユニークな特徴を持つCo-Array Fortran (CAF) [2, 19] についても必要に応じて言及する.

連載を通して, すべての言語に共通の問題を設定し, それを各言語で記述する. それにより,

1. 各言語の記述上のメリット・デメリットを明らかにする,
2. それを通じて, 各並列言語の存在意義, 設計上の決断の意義を明らかにする,
3. 実際に性能を比較することで処理系の「現状」に関して, 客観的な情報を提供する,

ことなどを目指す.

1.3.3 共通の問題

共通の問題として, 計算や通信のみからなる単純なベンチマークの他に以下を設定する.

1. 自明に並列 (Embarrasingly Parallel) な処理
2. 木探索
3. 分子動力学
4. 疎行列ベクトル積

まずはじめに, 最も単純な並列処理の例題を通して, 並列化のための構文, リダクションなどの基本操作の性能, 処理系のスケーラビリティの制約などを探る. 次に取り上げる木探索問題はいわゆるタスク並列の問題で, 主に並列性をどう表現するかという観点から各言語を分析する. 次に, 分子動力学の問題を取り上げる. これは, データ配置とタスク配置をうまく行えば通信・メモリアクセスよりも, 計算負荷中心とし得る問題の例として取り上げる. それらの局所性の最適化が各言語でどのように表現されるかを見る. 最後にメモリバンド幅や通信に律速されやすい疎行列ベクトル積を取り上げる.

表 1: 本連載で扱う問題 (予定)

問題の種類	意図
自明に並列な処理	並列化の表現, 処理系のスケーラビリティ調査
木探索	動的・不規則なタスク (並列再帰呼び出し) の表現・実行
分子動力学	局所性の最適化, 計算集約的な計算
疎行列ベクトル積	局所性の最適化, 通信・メモリ集約的な計算

1.4 本稿の構成

本稿では連載第一回目として, 並列言語の分類を行って, 取り上げる言語の位置づけ・違いを定性的に明らかにしていく. もちろんより詳しい比較は次回以降に行われる.

2 並列言語・ライブラリの分類軸

並列に限らず, あらゆるプログラミング言語やライブラリの設計は, 理想と現実の妥協の産物である. もう少し具体的な言い方をすれば, どこまでを処理系が面倒を見て, どこから先はプログラマにやらせるかの線引きをどう行うかを決断するプロセスと言える. 色々なことを処理系がやってくれるのであればプログラマにしてみれば楽で良いが, それをある程度「うまく」やってくれないのでは意味がない. それどころか, 処理系が下手くそにやった挙句, プログラマが手間を厭わず指示を出したくてもそれを出せない, などということになると, これは最初からすべてをプログラマにやらせる言語よりもダメな言語と言われてしまうことになる.

特に並列処理の場合, データ配置およびタスク配置が常に肝となり, それが性能に与える影響が大きい. 従って, それらを「処理系がやります, お任せください」と大きく出るのか, 「私にはできませんので, プログラマさん, やってください」と控えめに出るのかは大きな決断になる. またその影響もマシンの構成 (特に, ハードウェアによる共有メモリがあるか否か) で大きく異なる. また, 処理系がやることが無事功を奏するかというのも, 実装技術, 対象アプリケーションなどによって異なってくるため, どちらの選択が正しいかということは一般には言えない. そこで色々な言語・ライブラリが各所で異なった決断をしている.

以下ではこれらの重要な設計上の決断を軸にして, 各言語が行った「決断」を明らかにしていく.

2.1 並列性の記述方法による分類

最初の分類軸は, アルゴリズム中に存在する「並列性」を表現するのにどのような記述の手段があり, それがどう実行されるかということである.

2.1.1 タスク並列性

単純な例題としては,

$$f() + g()$$

の $f()$ と $g()$ を並列に評価するのにどのような記述の手段があり, それを処理系がどう実行するかを考える.

プログラマにしてみれば, 「$f()$, $g()$ それぞれを別々のタスクとして実行する (もしくはしてよい)」という構文なり, 注釈なりを加えるだけでそれが達成される, という言語が最も単純であろう.[1] 実際そのような言語は多い.

例えば Cilk では

```
int x = spawn f();
int y = g();
sync;
... x + y ...
```

と書くことでそれが達成される. `spawn f()` は, `f()` を別のタスクとして, 並行に実行することを意味する. Chapel での類似の構文は begin であり, 上記は

```
var x : sync int;
var y : int;
begin x = f();
y = g();
... x + y ...
```

と書ける. begin が, タスクを生成する構文である. Cilk では spawn によって作られたタスクの終了を待つのに sync という文を実行するが, Chapel の例では, 変数 x (sync という型のついた変数) を参照することで, 終了待ちがなされる. 同じことをより簡潔に記述する構文としては, cobegin がある.

```
var x, y: int;
cobegin {
  x = f();
  y = g();
}
... x + y ...
```

また, ループの繰り返しを並行に実行してそれらの終了を待つ, coforall 文も存在する. これらはみなより原始的な, begin によるタスク生成と, sync 変数による終了待ちを組み合わせたものに容易に変換できる.

OpenMP でも, 3.0 以降の仕様で task 構文が導入され, 上記は以下のように書ける.

[1] ここではそもそもそんな注釈すら必要がない—自動並列化・暗黙的並列化—は対象外としている.

```
  int x, y;
#pragma omp task shared(x)
  x = f();
#pragma omp task shared(y)
  y = g();
#pragma omp taskwait
  ... x + y ...
```

X10, TBB も類似の構文を持っている.

これらの記述方法は実に自然で当たり前である. 特に, 既存の逐次プログラムを元に並列化を行う場合, プログラマにかかる負荷は「どこを並列に実行して良いか」を処理系に教えることくらいであり, 表面的な変更点も小さい. ある言語やライブラリがこのような記述法をサポートする場合, それは「タスク並列」をサポートする, と呼ぶのが慣習になっているようである.

2.1.2 タスク並列の実装方式, 特に軽量タスク

タスク並列モデルはわかり易く記述しやすいモデルだが, 処理系の実装は必ずしも単純ではない. ポイントは, タスク並列をサポートすることは, 実行の途中の好きな場所で「並列に実行可能なタスクの生成」を許すことを意味する, という点である. 言い換えれば並列性は動的に生成・消滅し, それに伴い実行可能なタスクの数は動的に変化する. また, 実行中のタスクが同期により一時的に中断することもある. そして, タスクはあくまで実行中に作られるので, 事前にどの文からどのくらいの並列性が生成されるのかを予測することも困難である. 従ってそれらをどう管理してスケジューリングするか, 処理系がそれにどこまでコミットするかで一つの分かれ目が生まれる.

1. **1 タスク =1 OS スレッドとする.** 言い換えればスレッド管理・スケジューリングを OS にまかせる例としては, X10, Chapel の現状の実装がある. 実装は単純になるが, 多数のタスクを生成した場合の CPU・メモリオーバーヘッドは大きい.

2. **1 タスク =1 OS スレッドとするが, その数を制限する.** 例えば作られるタスクの数を一定数以下に制限したり, ネストした並列性を無視するなどして, 大量のタスクの生成を抑止する (**スレッドプール方式;**) 例としては OpenMP の parallel 構文の実装がある. 実装は 1 と同様単純だが, 多数のタスクを生成した場合, いったいどこまでが本当に並列に実行されるのかが理解しづらくなる.

3. **言語処理系が, 軽量なタスク・スレッドの管理とスケジューリングを, 効率的にサポートする.** 詳細は次回以降に述べる. これにより事実上, 無制限のタスク生成が可能になる. 例としては Cilk, TBB, OpenMP の task 構文の実装がある. ただし OpenMP の実装は処理系によって大きく異なる. 少なくとも現状

の GCC 4.3 の実装ではあまり台数効果は期待できないようである．詳細は次号以降で述べる．

タスク並列がサポートされていることの真価は, アルゴリズムに内在する並列性を自然にタスクとして表現するだけで効率よく実行される場合に発揮されるが, それができていると言えるのは 3 の場合だけである．

また, 個々のタスクをどのプロセッサで実行するかも大きな問題となる．Cilk や TBB では, プログラマが個々のタスクの実行プロセッサを指定することなく, 処理系が実行時にあいたプロセッサを見つけて負荷分散を行う．ただしそれらの処理系は, ハードウェア共有メモリを前提としていることに注意されたい．Chapel や X10 では Locale や Place というオブジェクトで, 分散メモリ計算機の 1 ノードを抽象化している．現状の実装では, 一つのノード (Locale や Place) 内では OS のスケジューラによりタスクの負荷分散がなされることになる．タスクをどのノードで実行するかということについては, on や at という指示を用いてプログラマが明示的に指定する．

2.1.3 Single Porgram Multiple Data (SPMD)

このような複雑さ, 実装依存の不透明さを排除するために, 動的なタスク生成という考え方を捨て去り, 実行途中のタスク生成を一切サポートしない並列プログラミングモデルも存在する．そのようなモデルでは, プログラムの開始時点から一定 (起動時に指定した) 数のタスクが生成され, 以降, 実行時にタスクの生成・消滅はできない．当然のことながら, そこでの 1 タスクは OS の 1 プロセスや 1 スレッドに対応しているから, そのようなモデルではわざわざタスクという抽象的な言葉を使わずに, 最初からプロセスもしくはスレッドと呼んだ方が適切である．以降ではプロセスと呼ぶことにするが, 必ずしも OS のプロセスに対応しているとは限らず, スレッドの場合もある．

そのようなモデルでのプログラミングは, 「全プロセスが, 常に大体似た様なところを実行している」というプログラミングスタイルになることが多い (Bulk Synchronous もしくは, Loosely Synchronous な実行)．そして, それらの全プロセスによるバリア同期や集合通信など, 全プロセスが同じ関数を呼び出すことで成立する協調処理 (**集団通信, 集団操作**) をサポートしているのが普通である．

このようなモデルのことを—全く実態を表していない意味不明な名前であるが—Single Program Multiple Data (SPMD) モデルと呼ぶのが一般的な慣習になっている．世の中の並列プログラムで, Mulitple Data (複数のデータ) を対象としてないプログラムは考えられないし, およそすべての並列言語において, プログラムそのものは Single Program (一つのプログラム) として書かれる．だからなぜこれを SPMD というのか, そもそも SPMD—一つのプログラムで複数のデータ—などという, 無意味としか思えない言葉が, 「何」を意味するつもりの言葉として発明されたのか, 全く理解に苦しむのだが, 世の中の慣習に習って本連載でもこのように呼ぶ．

SPMD モデルに基づいた並列言語の代表例は MPI である.[2] MPI では, main 関数が, コマンドラインのオプションで指定された数 (例えば mpirun -np 10 ならば 10 個) だけ実行されるというモデルになっている. プログラム実行中に新しいタスクを生成する自然な方法は, MPI の中には存在しない. UPC や CAF も SPMD モデルである.

なお, OpenMP では, プログラムの main 関数の実行を開始するのはあくまで一つのスレッドで, 複数スレッドでの実行が始まるのは parallel 構文 (`#pragma omp parallel`) に遭遇した地点である. その中で再び parallel 構文に遭遇すれば, 条件によってはさらにスレッドが作られる ([22] 35 ページ). その意味では parallel 構文によってタスク並列モデルをサポートしているとも言えるが, 上述したとおりそれは限定的なものである. つまり, parallel 構文で作られるスレッド数には制限があり, 決して大量のタスクの生成を素直に parallel 構文で表現できるわけではない. 一方, 一回の parallel 構文で作られたスレッド間でのバリア同期など, SPMD モデルでよく使われるプリミティブがサポートされている. これらの理由から OpenMP は SPMD モデルとして使われることが多い. すなわち, プログラムの開始に近い地点でプロセッサ数分のスレッドを parallel 構文で作り, あとは一切 parallel 構文を実行しない, というスタイルである. 同様に X10 においてもバリア同期に相当する構文 (clock オブジェクトと resume/next メソッド; [26] 167 ページ) をサポートしている. つまり, 現状でいかに SPMD モデルが世の中を支配しているか, ということである.

2.1.4 Work sharing 構文

SPMD モデルに基づくプログラミングでは, 立ち上がった各プロセスの動作を記述する. アルゴリズム中の, 並列に実行可能なタスク群に対し, どのプロセスがどのタスクを実行するのかを決め, プログラマは「それらのタスク群を実行するプロセスの動作」を記述することになる. 一方, タスク並列モデルでは, あくまで実行すべき計算「全体」を記述する. プログラマの主な仕事はその計算中のどこが並列に実行可能であるかを, タスクという形で明示することである. 比喩的に言うならば, SPMD は「個々のプロセッサ視点」ないし「断片的な視点 (fragmented view)」でプログラムを記述する. タスク並列モデルでは, 「大域的な視点 (global view)」でプログラムを記述する. 例えば, 以下の for 文:

```
for (i = 0; i < n; i++) {
  f(i);
}
```

の並列実行を大域的な視点で記述したものは,

```
coforall (i in 0..n - 1) {
  f(i);
}
```

[2]繰り返すが, MPI は言語ではなくライブラリだが, 区別せずにすべて言語と呼ぶことにしている.

のようになり (coforall は Chapel の並列 for 文), 断片的な視点で記述したものは,

```
begin_idx = (      my_rank * n) / n_procs;
end_idx   = ((my_rank + 1) * n) / n_procs;
for (i = begin_idx; i < end_idx; i++) {
  f(i);
}
```

のようになる. 後者の記述は表面的に複雑になっているだけでなく, 「プログラム全体として何をしたいのか」がもはや明らかではなくなっている. この違いは再帰呼出のように, 生成されるタスクを容易に分割できない場合により顕著になる. 例えば quicksort に現れる 2 つの再帰呼び出し:

```
quicksort(a, p, q) {
   ...
   quicksort(a, p, r);
   quicksort(a, r, q);
}
```

をタスク並列で並列化したものは Cilk では

```
quicksort(a, p, q) {
   ...
   spawn quicksort(a, p, r);
   quicksort(a, r, q);
   sync;
}
```

でよいが, SPMD でこれを記述するのは相当骨が折れる.

SPMD モデルにおいても, for 文の並列化のような頻出パターンを大域的な視点で記述できる, work sharing 構文をサポートされている場合がある. 例えば UPC では上記を

```
upc_forall (i = 0; i < n; i++; 7*i) {
  f(i);
}
```

と記述できる. 通常の C 言語の for 文よりも区切り (;) が一つ多い. 最後の `7*i` は, affinity 指示と呼ばれ, 繰り返し `i` をプロセス (`7*i` mod P) 上で実行することを指示している (P はプロセス数). 言い換えれば各プロセスは, (`7*i` mod P) が自分のプロセス ID と一致するような `i` だけを選んで上記の for 文を実行する. OpenMP のプラグマ for 構文 (`#pragma omp for`) も似たセマンティクスである. 違いは, それを実行するスレッドはプログラム開始時からではなく, そのを取り囲む parallel 構文で作られたスレッドである, とういことである.

表 2: 並列性の表現という観点から見た並列言語の分類.

	タスク並列	軽量タスク	SPMD	work sharing 構文	タスクマッピング
MPI	N	N	Y	N	すべて手動
CAF	N	N	Y	N	すべて手動
UPC	N	N	Y	Y	`upc_forall` の affinity 句
OpenMP	Y	Y/N[3]	Y	Y	`#pragma omp for` の `schedule` 句
Cilk	Y	Y	N	-	動的負荷分散
TBB	Y	Y	N	-	動的負荷分散
X10	Y	N	Y	-	ノード間: `at` 句, ノード内: OS
Chapel	Y	N	N	-	ノード間: `on` 句, ノード内: OS

これらは構文上は Chapel の coforall と大差なく見えるが, あくまで SPMD モデルに沿った実行を前提としていることに注意されたい. 具体的には, あくまでこれらは, 「すでに作られている」プロセスたちが, この for 文を協力して実行するという意味である. すなわちあくまで「全員参加」を必要とする構文である. Work sharing 構文という名前はこれを反映したものである. 一方タスク並列モデルにおける並列 for 文にはそのような制限は存在しない.

2.1.5 観点の整理

ここまで議論した観点を整理してまとめたものが表 2 である. 以下に各列の説明をする.

タスク並列: タスク並列をサポートしているか? していれば Y.

軽量タスク: している場合, その実装は, 事実上無制限の並列タスクの生成を効率的にサポートしているか (いわゆる軽量タスクをサポートしているか)? していれば Y.

SPMD: SPMD モデルで多用される協調処理 (バリア同期や集団通信) をサポートしているか? していれば Y.

work sharing 構文: タスク並列をサポートしない言語において, for 文を大域的視点で記述できる構文 (work sharing 構文) をサポートしているか? していれば Y.

タスクマッピング: タスクがどこで実行されるかがどう決まるか, それを指示する方法としてどのようなものがあるか.

[3] OpenMP 3.0 以降の task 構文. ただし性能は実装に大きく依存する.

2.2 データのアクセス方法・視点による分類

次に, 並列言語のもう一つの重要な設計上の問題である, データのモデル化に関して分類を試みる. 一言で言うならば, アプリケーションが必要とするデータをどのように配置し, それらへのアクセスをどのように表現するかという問題である. 例えば以下の典型的なステンシル計算:

```
double a[N][N];
  ...
  a[i][j] = 0.25 * (a[i+1][j] + a[i][j+1] + a[i-1][j] + a[i][j-1]);
```

において, 配列 a をどのように宣言し, それぞれの要素へのアクセスをどう記述するか, という問題である.

プログラマの視点からは当然, 逐次プログラミングと同様に, つまり上で書いた通りそのまま書けるのが最も良い. 実際, ハードウェアによる共有メモリがサポートされたプラットフォームでは, そう書かせることに, 処理系としてはなんの工夫もいらない. OpenMP, Cilk, TBB などは全てこのモデルである.

分散メモリ計算機においては a の一部の要素が, 他の計算機上にある場合があり得る. 従ってそれらへのアクセスをどのように書かせるのかで設計上の分かれ目が生ずる.

2.2.1 メッセージパッシング

最も単純なものは, 「処理系は何もしない」というもので, 必要なデータを明示的に送受信するためのプリミティブだけを提供する. そのプリミティブは典型的には send/receive である. これは一般に**メッセージパッシングモデル**と呼ばれる. これに加えて SPMD に基づくモデルでは, ブロードキャスト (1-N 通信), 収集・集約 (N-1 通信), 全対全 (N-N 通信) などの集合通信がサポートされるのが普通である.

メッセージパッシングモデルでは, 一般にプログラミングの労力は大きい. その理由は以下に要約される.

1. データを送信する際にそれをメッセージとして送信できる形 (バイト配列) にシリアライズし, 受信する際にその逆を行うことが必要.

2. データの移動を行うのに, データを供給する (持っている) プロセスと, データを消費する (アクセスする) プロセス両方の関与が必要. 特に, データを供給するプロセスにおいて, 自分が行う計算とは直接関係のない処理を記述することになりがちである.

3. 特に, 自分に対して複数のメッセージが到着し得る状況では, メッセージの受信後の振り分け処理などでさらに面倒なプログラミングが必要になる.

4. そのような場合に, うっかり特定のメッセージだけを受け取るような記述を行うと, デッドロックを起こす可能性もある. 例えば, プロセス A がデータ a, プロセス B がデータ b を保持している状況で,

   - A が B に, 「b を読みたい」というメッセージを送ってその返事を待つ
   - B が A に, 「a を読みたい」というメッセージを送ってその返事を待つ

   というプログラムを書くと直ちにデッドロックする (fetch deadlock). 返事を待っている間も, 相手からのリクエストが来ないかどうかを見張ってその処理をしなくてはならない.

5. 逆に, 非常に大きなデータを一方的に送信するようなコードも, デッドロックする (send deadlock). それは, 多くの場合メッセージ送受信のバッファの大きさに制限があり, ある程度以上のデータがたまったらそれが受信処理によってユーザ指定のバッファにコピーされない限り, これ以上送れない, という状態になるからである.

Fetch deadlock の問題は, もしお互いが「相手が今, 自分のデータを必要としてる」ことをあらかじめ分かっているならば, 「a/b を読みたい」などというメッセージを受け取るまでもなく, いきなり相手にデータを送り付けてしまえば解決する. SPMD スタイル—全プロセスが「大体」同期的に動くスタイル— のプログラムでは, 「誰が今, 自分のデータを必要としてるか」がアルゴリズムの構造上分かっていることが多いため, この解決法が通用する場合も多い. しかし一般に, どのプロセスがどのデータを必要とするかが事前にわからない場合には通用しない.

また, a や b の大きさが巨大で, send deadlock を同時に避けようと思えば, 以下のいずれかが必要になる.

- 送信をノンブロッキング (MPI_Isend など) で行う
- デッドロックしない様に送受信をスケジューリングする. この場合であれば,
  - A は先に a を送信してから b を受信
  - B は先に a を受信してから b を送信

  とする.

メッセージパッシングモデルの代表は言うまでもなく MPI である.

2.2.2 片側通信, 遠隔メモリアクセス

そのような面倒を解消し, 分散メモリ計算機を, より共有メモリ風に見せるための抽象化が提案されている. 面倒の多くの部分はデータの移動に毎回 2 プロセスの協調が必要になること (それに伴うデッドロックの回避) から発生している.

これを解決する最も原始的な解決法が**片側通信 (one-sided communication)** もしくは**遠隔メモリアクセス (Remote Memory Access; RMA)** である. 表面的には put, get などの API で, 他のプロセスのメモリへの書き込み, 読み込みを可能にする. ポイントは, メモリをいじられる側のプロセスが明示的に介在しなくても, 処理が行われるという点である.

MPI 2 は MPI_Get, MPI_Put という API で, RMA をサポートしている. 概念的には, それらのメモリアクセス要求がメッセージとして送信され, それをいつでも処理可能なスレッドがバックグラウンドで動作している, というイメージである. ただし, ハードウェアの NIC が片側通信を直接サポートしている場合もあり, そのような場合, 要求を受け取ったノードの CPU が介在しない RMA が実現されている.

RMA はメッセージパッシングに比べるとプログラマの負荷が低いが, 他のノードの任意のアドレスを最初から自由にアクセスできるわけではなく, 真の共有メモリにある手軽さ・柔軟さはない. MPI 2 では, 全プロセスが MPI_Get, MPI_Put の対象となる領域を事前に登録する必要がある (MPI_Win_create). これは集団操作で, 基本的には全プロセスがこれを呼び出す必要がある. また, メモリアクセスの完了を保証するには別の集団操作 (MPI_Win_fence) が必要である. そして, あるデータがどのノード (プロセス) 内にあるのかを管理するのはあくまでプログラマの責任である.

2.2.3 大域アドレス空間

RMA よりも共有メモリに近い抽象化を提供するのが**大域アドレス空間 (Global Address Space)** である. 共有メモリに近いとは言っても, すべてのメモリが同一のノードにあるわけではなく, ノードが各自のメモリを大域アドレス空間に供出して, 大域アドレス空間を作り上げている. 言い換えれば大域アドレス空間は実際には複数のノードに分割されて保持されている. これを強調して **Partitioned Global Address Space (PGAS)** と呼ばれることが多い.

端的に言えば PGAS とは, 特段の事前準備なく他のノード上の変数や配列要素にアクセスすることを可能にするモデルのことである. PGAS にも大きく 2 種類あり, アクセスしたいデータを指定するのに, 「プロセス p にある変数 x」とか, 「プロセス p にある配列 a の 5 番目の要素」のように, プロセスとともに指定する必要のあるモデルと, 単純に「(大域的な) 配列 a の 23 番目の要素」と指定するだけで良いもの—言い換えればその要素がどこにあるかを処理系が管理してくれるもの— が存在する. 前者の場合, 各プロセスが配列 a を保持しているのであり, a の 5 番目の要素と言ってもプロセス毎に別の要素が存在する.

前者を「局所的な視点 (local view) に基づく PGAS」, 後者を「大域的な視点 (global view) に基づく PGAS」と呼ぶ. 前者の実例が CAF, 後者の実例が UPC, Chapel, X10 である.

なお, 筆者の語感からは, 「大域アドレス空間 (global address space)」というのはまさしく, アドレス「空間」が大域的なことを指すのであって, 「配列 a の 5 番目の

要素」が複数存在している時点でそれを, 大域アドレス空間と呼ぶこと自体が矛盾していると思うのだが, ここでも世の中の慣習には逆らわないことにする.

2.2.4 局所的な視点に基づく PGAS

CAF では, 配列を宣言する際それを co-array であると宣言することができる. 例えば以下の通常の配列の宣言

```
real, dimension(n) :: a
```

を

```
real, dimension(n)[*] :: a
```

と書くとそれは a という co-array の宣言になり, その効果はサイズ n の配列 a を各ノード (CAF 用語では**イメージ**と言うが, 以下でもノードと言うことにする) に作ることである. そして他のノードの要素, 例えばノード番号 3 の 5 番目の要素を参照したければ,

```
a(5)[3]
```

のように書く.

2.2.5 大域的な視点に基づく PGAS

UPC や Chapel では, 大域アドレス空間におかれたデータを, データが実際にどのノードにあるかを意識せずにアクセスできる. 例えば UPC では共有 (shared) と宣言された変数や配列は, すべてのプロセスから同じようにアクセスできる.

```
shared int x;
```

と書かれた大域変数があれば, どのプロセスであってもこの変数 x を, x と書くだけでアクセスできる. 配列を共有であると宣言することもでき, その要素を複数のプロセスで分割することもできる. 例えば,

```
shared int b[100*THREADS];
```

は 100×THREADS (プロセス数) の要素を持つ配列を作り, それを 100 要素ずつプロセスに分割する. このように配列を作った上で, アクセスする際は $0 \leq i < 100 \times$ THREADS の通常の添字を用いてアクセスできる. この際, 実際にどの要素がどのノード上に置かれていようとも, プログラマはそれを知る必要もない. データをアクセスするのに必要なのはあくまで変数や配列の名前や添字 (ないしそれらへのポインタ) だけである.

UPC では shared というラベル (型の一部) を元に, ポインタや配列がローカルな (通常の) ポインタであるか, 大域アドレス空間に対するポインタであるかを, 区別で

きるようになっている．これにより shared でないポインタ参照に対して, オーバーヘッドの少ないコードを生成することを可能にしている．Chapel では配列やオブジェクト (class のインスタンス) は自動的に共有される．比喩的に言えば配列やオブジェクトにはことごとく shared が付けられている, ということである．

X10 のデータモデルはやや特殊である．まずデータの指定方法としては, Chapel や UPC 同様の大域的な視点に基づく大域アドレス空間を提供している．大域配列も提供され, 同じインデクスを持つ要素は世界に一つしかない．配列以外の (1 ノード内に収まる) データについては, 任意のデータを大域参照 (GlobalRef) というデータで包んで, 遠隔からポインタで指すことが可能である．しかし実際にその中身にアクセスするためには, プログラマが明示的にそのノードに実行を移すよう, 明示的に指示を書かなくてはならない．さもなければ実行時エラーになったり, コンパイル時にエラーになったりする．

2.2.6 データのマッピング

大域的な視点に基づく PGAS モデルでは同時に, データが実際にどのノードに配置されるか—データのマッピング—が重要な問題になる．

まず配列に関しては, UPC, Chapel, X10 とも, 多次元の配列を選択した軸に沿ってブロック-サイクリック分割するなどの基本的なパターンを用意している．配列の要素をノードへマッピングするのは概念的には, 配列のインデクスからノードへの関数を指定することになる．Chapel はこの概念に基づいて配列の要素のマッピング方法を, ユーザが任意に定義できる枠組みを用意している．

また, Chapel や X10 では個々のオブジェクトは, デフォルトでは計算が実行されているノード内に生成される．従って, on や at などの計算を移動させるプリミティブを使って, オブジェクトを指定の位置に生成することができる．UPC では共有領域からの動的メモリ割り当て関数が提供されており, それを呼出たノード上にメモリが割り当てられる (upc_alloc)．また, 複数ノードに分散した大域配列を共有領域から割り当てることもできる (upc_global_alloc, upc_all_alloc)．

2.2.7 観点の整理

ここまで議論した観点に従って各言語の特徴を整理したものが表 3 である．

分散メモリ: 分散メモリマシンを対象としているか? それとも, ハードウェア共有メモリのある環境でのみ動くことを, 事実上前提としているか? 分散メモリマシンを対象としていれば Y.

RMA: Y の場合, 遠隔メモリアクセス (RMA) をサポートしているか? していれば Y.

表 3: メモリのモデルという観点から見た並列言語の分類

	分散メモリ	RMA	PGAS	global view	マッピング指示
MPI	Y	Y[4]	N	N	-
CAF	Y	Y	Y	N	-
UPC	Y	Y	Y	Y	block-cyclic
OpenMP	N	-	-	-	-
Cilk	N	-	-	-	-
TBB	N	-	-	-	-
X10	Y	Y[5]	Y	Y	block
Chapel	Y	Y	Y	Y	block-cyclic, ユーザ定義

PGAS: Y の場合, 大域アドレス空間 (PGAS) をサポートしているか? していれば Y.

Global View: Y の場合, データの指定方法は global view に基づくものか, local view に基づくものか. 前者ならば Y.

マッピング指示: Y の場合, オブジェクトや大域配列の, ノードへのマッピング指示

3 まとめと展望

このように言語を分類してみると, 改めて並列言語の設計が, 悪い言い方をすれば「妥協の産物」であるか, もう少し前向きな言い方をすれば, 現実と理想の間で難しい選択をしているか, ということが確認できる.

並列化のための構文や負荷分散について言えば, 任意の場所で並列度を生成できる機能を実装している処理系自体が少ない上, それは共有メモリ環境に限られている. X10 や Chapel が成熟するにはこの部分の実装をすすめる必要がある. また, 自動的な負荷分散は事実上, ハードウェア共有メモリの中以外は, どれもプログラマ指定である.

データのモデル化について言えば, PGAS という抽象化は非常に有用だが, それを自由にやらせると細かい通信が多発し, オーバーヘッドと遅延の塊のようなプログラムが容易に出来てしまう. 処理系による通信の集約 (aggregation) が行われるのが理想であるし, それができなくてもプログラマに「逃げ道」を用意しておく必要がある. UPC では upc_memget, upc_memput などのブロック転送のための API が存在する. しかし連続したアドレスに対するデータにしか通用しないため, 散らかったデータを一回の upc_memget で持ってくることは (たとえそれらのデータが一つの

[4]MPI 2 以降
[5]ただし at 句によって計算を移動させる必要がある

ノード上にあっても) できない. それを直すにはデータの持ち主の方で, それらを連続アドレスに詰め込む必要があるが, それでは MPI と変わらない. また, Chapel や X10 の現状の実装も通信の集約を行っていない.

また, 分散メモリを対象とするどの枠組みも, ノード内・ノード間という 2 レイヤの区別しかおこなっていない. これは現在のマシンの性能特性のそこそこよい抽象化にはなっているが, 今後のアーキテクチャにとっては不足であろう.

- ノード内であっても均一なメモリアクセス性能は期待できない,
- ノード間の通信オーバーヘッドの低いネットワークが普及する,
- 一方で大規模な分散メモリマシンでは同じノード間であっても, ノード間の距離やトポロジーを考慮したデータ配置が必要になる,

など「局所性」の問題をより複雑で重層的なものして行く要素が存在する.

これまでのところ, 高水準言語処理系の未成熟さも手伝って, MPI のように「現実をありのままに見せる」言語だけが成功していると言ってよいだろう. しかし今後, 今述べたようなノード内・外の階層や, ノード内の GPU/CPU のヘテロ構成を, そのままプログラマに露出する言語, GPU が世代を変えるごとに再プログラミングを要求する言語が生き延びるとは考えがたい. あらゆる階層に存在する局所性を「うまく」抽象化して, 局所性の良いプログラムを高水準に記述できる言語 (Cilk などの分割統治に基づく記述はすでにその良い出発点と言える), GPU もしくは今後のスループット指向プロセッサを, 透明に有効活用できる言語処理系 [17, 20] が重要になる. この連載を, 処理系の現状に関して情報提供すると共に, 言語設計上の選択肢の整理・再考する場となれば幸いである.

参考文献

[1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification version 1.0. Technical report, Sun Microsystems, Inc., 2008.

[2] Co-Array Fortran. http://www.co-array.org/.

[3] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.

[4] The Chapel parallel programming language. http://chapel.cray.com/.

[5] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[6] The Cilk project. http://supertech.csail.mit.edu/cilk/.

[7] Intel Cilk Plus. http://software.intel.com/en-us/articles/intel-cilk-plus/.

[8] Cray. Chapel language specification 0.796. Technical report, Cray Inc, 2010. http://chapel.cray.com/spec/spec-0.796.pdf.

[9] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley & Sons Inc., 2005.

[10] Project Fortress community. http://projectfortress.sun.com/Projects/Community/.

[11] Michael Garland and David B. Kirk. Understanding throughput-oriented architectures. *Communications of the ACM*, 53(11):58–66, 2010.

[12] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[13] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press, 1999.

[14] Supercomputing Technologies Group. *Cilk 5.4.6 Reference Manual*. MIT Laboratory for Computer Science.

[15] Hadoop. http://hadoop.apache.org/.

[16] Doug Lea. A Java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, New York, NY, USA, 2000. ACM.

[17] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–110. ACM, 2009.

[18] The message passing interface (MPI) standard. http://www.mcs.anl.gov/research/projects/mpi/.

[19] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17:1–31, 1998.

[20] Satoshi Ohshima, Shoichi Hirasawa, and Hiroki Honda. OMPCUDA : OpenMP execution framework for CUDA based on Omni OpenMP compiler. In *Proceedings of International Workshop on OpenMP*, volume 6132 of *Lecture Notes in Computer Science*, pages 161–173. Springer Berlin / Heidelberg, 2010.

[21] OpenMP. http://openmp.org/wp/.

[22] *OpenMP Application Program Interface Version 3.0*. http://www.openmp.org/mp-documents/spec30.pdf.

[23] Peter Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1996.

[24] Intel Parallel Building Blocks. http://software.intel.com/en-us/articles/intel-parallel-building-blocks/.

[25] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. Oreilly & Associates Inc, 2007.

[26] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. Report on the programming language X10 version 2.1. Technical report, IBM, 2010. http://dist.codehaus.org/x10/documentation/languagespec/x10-latest.pdf.

[27] Intel Threading Building Blocks 3.0 for open source. http://www.threadingbuildingblocks.org/.

[28] Unified Parallel C. http://upc.gwu.edu/.

[29] Tom White. *Hadoop: The Definitive Guide*. Oreilly & Associates Inc, 2009.

[30] X10. http://x10.codehaus.org/.

[31] 青木尊之 and 額田彰. **はじめての** *CUDA* **プログラミング**. I・O ブックス, 2009.