社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－ARC－179 (4)
2008／8／5

配列処理向けドメイン特化言語によるマルチコアプログラミング

瀬川淳一† 金井達徳† 城田祐介†

配列処理アルゴリズムには、処理の並列性や参照データの局所性といった最適化ポイントがあり、それらを実装対象のマルチコアアーキテクチャに応じた最適化手法で実装すれば性能を向上させることができる。しかし、最適化を考慮した実装は開発コストが高く、再利用性も低いという問題がある。そこで我々は、特定のマルチコアアーキテクチャに依存しない配列処理向けドメイン特化言語を用いるプログラミングシステムを開発した。本言語は、配列処理向けに定義した演算子と関数適用方式を備える関数型言語であり、配列処理アルゴリズムをループレスで簡潔に表現できるだけでなく、プログラム中で使われている演算子や関数適用方式から、最適化に用いる情報を容易に抽出できるように設計されている。その情報を用いることで、本言語で記述したプログラムはマルチコアアーキテクチャに最適化されたC/C++のソースコードに自動変換できるため、人手による最適化実装が不要となり、開発コストと再利用性の問題を解決することが期待できる。

Multicore Programming with Domain Specific Language for Array Processing

Jun'ichi Segawa †, Tatsunori Kanai † and Yusuke Shirota †

In array processing, parallelism and data access locality are the keys to enhance performance. If they are implemented with optimal programming techniques suitable for each target processor, better performance can be obtained. We have developed a new domain specific language specialized for array processings. This language provides array-oriented operators and various function application methods to write array processing programs concisely. Information for parallel execution and execution scheduling can be extracted from the array-oriented operators and function application methods with simple analysis. By referring this information, complex implementation optimized for multicore processor can be generated automatically. With this language, developers are not only able to write complex algorithm simply, but also free from optimization implementation for target processors.

1. はじめに

静止画、動画像などのマルチメディアデータや物理シミュレーションで用いる各種物理量のデータなど、規模が大きく一様な構造のデータは巨大な配列で管理することが多い。このような巨大な配列に対するアルゴリズムは計算量が多く、素朴に実装してしまうと処理性能を引き出すことができない。しかし、配列処理アルゴリズムには、並列実行可能な処理や参照データの局所性といった最適化ポイントが存在する。そういった最適化ポイントを、処理を実行するマルチコアアーキテクチャに適した並列化や負荷分散、内部メモリの活用といった最適化手法で実装すれば、高速化が可能である。

しかし、アルゴリズムが複雑になると、最適化実装の難易度は高くなる。また、ソースコードの再利用性の面でも、特定のマルチコアアーキテクチャに最適化したソースコードは別のアーキテクチャ上では性能が出ないため、再実装が必要になるという問題がある。

このような最適化実装の難しさや再利用性の低さの要因は、C/C++などの従来の手続き型言語で開発した場合、「アルゴリズム部分の実装」と、アーキテクチャに依存する「最適化部分の実装」が混在するからである。開発者は両方の実装を同時に行わなければならない。もし、両者を分離し、開発者はアルゴリズム部分の実装だけを行い、残る最適化部分の実装を自動化できれば、最適化実装の難しさと再利用性の低さの問題を同時に解決することができる。

そこで我々は、配列処理アルゴリズムの簡潔な記述と、マルチコア向けの最適化情報の自動抽出が可能な関数型言語を開発した[1)~3)]。本言語は、配列処理アルゴリズムのドメインに特化した言語であり、処理系により特定のマルチコアアーキテクチャに応じた最適化を施したC/C++のソースコードに変換して実行する。

本稿では、配列処理言語を用いたマルチコアプログラミングについて述べる。以下、次のような構成となっている。第2章で本言語の特徴を述べ、第3章で本言語による配列処理アルゴリズムの記述方法を説明し、第4章でマルチコアアーキテクチャ上での最適化方式について説明する。そして、第5章で関連研究を紹介し、第6章でまとめとする。

2. 配列処理言語の特徴

本言語は、配列処理の記述のしやすさと、マルチコアアーキテクチャ上での性能を両立させた関数型言語であり、以下の3つの点に着目して言語設計を行った。

配列処理をループレスで記述可能

C/C++などの手続き型言語では配列処理をループを使って記述する。ループの部分は処理性能への影響が大きいため、並列化などの各種最適化手法を適用した複雑な実装が必要になることが多い。本言語では、配列処理向けに定義した演算子と関数適用方式を提供することで、配列処理をループレスで簡潔に記述できるようにする。その上で、処理系によって、性能に大きく影響を与える最適化の実装を行ったループを自動生成する。これにより、開発者はループに関する最適化の実装から解放され、アルゴリズムの実

† (株) 東芝 研究開発センター
Toshiba Corporation, Corporate Research & Development Center

装そのものに専念することができる。

最適化に用いる情報の抽出が容易

従来の手続き型言語の処理系では依存関係の詳細な解析を行って最適化に必要な情報を抽出している。それに対して本言語は、提供する配列処理用演算子のパラメータや関数適用方式の組合せ方から、並列化可能な処理単位や同期ポイント、さらには処理に必要なメモリ量といった、最適化に必要な情報を処理系が簡単に抽出できるように設計している。また、変数への再代入ができない関数型言語にしたことも、依存関係の解析を簡単にしている。

C/C++のソースコードとシームレスな連携が可能

本言語は配列処理向けに特化しているため、本言語には向いていない処理 (配列以外のデータ構造に対する処理や、I/O 処理などプラットフォームに依存する処理など) は、C/C++で実装することを想定している。そのため、本言語から変換されたソースコードは、C/C++による実装 (ホストコード) とシームレスに連携できることが必要である。そのため、本言語のデータ型は C/C++を踏襲した設計にしている。また、本言語で記述されたプログラムはホストコードから簡単に呼び出せるように C/C++の 1 つの関数に変換される。

3. 配列処理言語によるアルゴリズム記述

並列化に必要な最適化情報を取り出しやすいように設計した、配列処理用の演算子と関数適用方式を提供しているのが、本言語の特徴である。ここでは、これらの言語機能を具体的なアルゴリズムの記述例を交えながら説明する。

3.1 データ型

本言語のデータ型は基本型と配列型の二種類に分けられる。基本型には C/C++と同様に整数型と浮動小数点型、および複素数型がある。配列型は多次元の配列を定義可能で、その要素は基本型だけでなく、配列型も指定できる。配列の軸には番号が割り当てられており、各軸のサイズを 1 次元配列 (ベクトル) で表現したものを配列の形と呼ぶ。配列の要素が配列である場合には、そのネストの段数を配列の深さと呼ぶ。配列型のデータは、2 次元の場合は数式における行列と同様に記述し、ベクトルの場合は $\langle 1,2,3\rangle$ のように記述する。

データ型の宣言はホストコードとのデータのやりとりを行うメイン関数の引数および結果に対してのみ行い、右下添え字で記述する。データ型の宣言は、基本型の場合は型の名称を記述し、配列型の場合は、外側の配列から順に形をベクトルで記述し、最後に末端の要素の型を記述する。ただし、配列のサイズを限定しない場合は $\langle *,*\rangle$ のようにサイズを明示しない。

3.2 基本演算子とアグリゲーション演算子

本言語の基本演算子はオペランドに配列を指定することを許すことで、配列の全要素に対する並列処理を記述できるように定義している。すなわち、C/C++の四則演算子や比較演算子などの基本的な演算子および標準的な超越関数は、オペランドが配列型の場合は配列の要素に対して演算を適用する。また、2 項演算子で一方が基本型で他方が配列型の場合は、基本型のデータを配列型のデータと同じ形の配列に拡張してから演算を適用する。基本演算子をこのように定義することで、要素に対する演算が互いに独立であることが保証され、並列化可能な処理として抽出できる。

例えば、2 枚の画像を (配列 $M1, M2$) を合成するアルファブレンド処理は次のように記述できる。

$$\alpha \leftarrow 0.3$$
$$AlphaBlend(M1_{\langle *,*\rangle\#uint8}, M2_{\langle *,*\rangle\#uint8})_{\langle *,*\rangle\#uint8} \leftarrow \alpha \times M1 + (1-\alpha) \times M2$$

C/C++にはない本言語特有の演算子として、配列から要素の総和や最大の値の要素といった集約された値を計算するアグリゲーション演算子を提供している。アグリゲーション演算子には、総和 (Σ)、総積 (Π)、最大値 ($\triangle$)、最小値 (∇)、最大値を持つ要素の座標 ($\vec{\Delta}$)、最小値を持つ要素の座標 ($\vec{\nabla}$) がある。

3.3 切り出し演算子とマップ関数呼び出し

従来の手続き型言語で多重ループで表現していたような複雑な処理の多くは、本言語では、部分配列の切り出し演算子 ($\bigoplus$) と、配列要素への関数の並列適用を行うマップ関数呼び出し ($\forall$) を組み合わせて記述する。例えば画像のフィルタ処理では、対象画素を中心とした周辺画素に対してフィルタ定数を使った畳み込み演算を行って結果の画素を生成する処理を全画素に対して行う。この処理は、各々畳み込み計算が互いに独立で並列処理可能であり、$\bigoplus$ 演算子とマップ関数呼び出しにより簡潔に記述でき、かつ、並列化可能である。

$\bigoplus$ 演算子は、図 1 に示すように、配列から部分配列を切り出し、切り出した配列を要素とする配列を生成する演算子であり、$\bigoplus_{base\#step}^{size\#esize}$ と記述する。4 つのパラメータは切り出しパターンをベクトルで表現したもので、*base* で切り出しの開始座標、*step* で切り出す部分配列間の間隔、*size* で部分配列の個数、*esize* で部分配列の形を、それぞれ指定する。なお、$\bigoplus$ 演算子が複数の部分配列を切り出すのに対し、単一の部分配列を切り出す演算子 $\bigodot_{base}^{size}$ も提供している。$\bigodot$ 演算子では *base* で切り出し開始座標、*size* で切り出す部分配列の形を指定する。

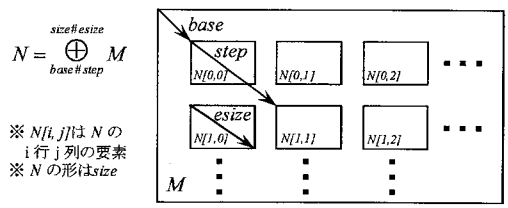

図 1 繰り返し切り出し演算子

マップ関数呼び出しは、次に示すように関数を引数の配列の要素に対して適用し、結果を適用した要素と同じ座標に格納する関数適用形式である。マップ関数呼び出しは、呼び出す関数名に $\forall$ をつけて指定する。

$$\forall f\left(\bigoplus_{base\#step}^{size\#esize} M\right) \equiv \begin{bmatrix} f(N[0,0]) & f(N[0,1]) & \cdots \\ f(N[1,0]) & f(N[1,1]) & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

※ $N[i,j]$ は M から切り出された i,j 要素の部分配列

$$threshold \leftarrow 30; \quad d \leftarrow 82.0; \quad f \leftarrow 32.0$$

$$Lsize \leftarrow \langle 5,5\rangle; \quad Rsize \leftarrow \langle 9,89\rangle; \quad Lbase \leftarrow \langle 0,0\rangle - (Lsize-1)/2; \quad Rbase \leftarrow Lbase - (Rsize - Lsize)/2$$

$$SAD(dst, src) \leftarrow \sum |dst - src|$$

$$Distance(parallax) \leftarrow \frac{d \times f}{\|parallax\|}$$

$$Search(l,r) \leftarrow \begin{cases} \vec{\nabla}\forall SAD\left(\star l, \bigoplus_{\langle 0,0\rangle \# \langle 1,1\rangle}^{\rho(r)-\rho(l)+1\#\rho(l)} r\right) - \dfrac{Rsize - Lsize}{2} & \textbf{when} \quad \triangle l - \nabla l > threshold \\ \langle 0,0\rangle & \textbf{otherwise} \end{cases}$$

$$StereoMatch(L_{\langle *,*\rangle \# uint8}, R_{\langle *,*\rangle \# uint8})_{\langle *,*\rangle \# double} \leftarrow \forall Distance\left(\forall Search\left(\bigoplus_{Lbase\#\langle 1,1\rangle}^{\rho(L)\#Lsize} L, \bigoplus_{Rbase\#\langle 1,1\rangle}^{\rho(R)\#Rsize} R\right)\right)$$

図 2 ステレオマッチングによる距離算出プログラムの記述例

本言語は関数型言語であるため、マップ関数呼び出しの各々の配列要素に対する関数適用間には依存が無いことが保証でき、それぞれの関数適用を並列化可能な処理として扱うことができる。

具体例として、エッジ検出などに使われる 8 近傍の 2 次微分を求めるラプラシアンフィルタの記述例を示す。

$$Convolution(src) \leftarrow \sum \left(src \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \right)$$

$$Laplacian(M_{\langle *,*\rangle \# uint8})_{\langle *,*\rangle \# int} \leftarrow \forall Convolution\left(\bigoplus_{\langle -1,-1\rangle \# \langle 1,1\rangle}^{\rho(M)\#\langle 3,3\rangle} M\right)$$

この記述例では、ラプラシアン用のフィルタ定数を使った畳み込み計算を行う *Convolution*() と、これを使ってラプラシアンを求める *Laplacian*() を定義している。*Laplacian*() では、入力画像 M の各画素を中心とする形が $\langle 3,3\rangle$ の部分配列を切り出し、それに対しマップ関数呼び出しで *Convolution*() を適用する。なお、この記述例に出てくる ρ 演算子は、配列の形を返す演算子である。

マップ関数呼び出しで呼ばれた関数の中から、さらに別の関数をマップ関数呼び出しで呼び出すことで、より複雑な処理を簡潔に記述できる。図 2 は本言語を用いてステレオマッチングを記述した例である。ステレオマッチングとは、左右 2 つのカメラで対象物を撮影し、左右の画像の視差から対象物までの距離を計算するアルゴリズムである。

この記述例では、最初にブロックの大きさやカメラ間の距離といった定数の定義を行い、続いて類似度を差分の絶対値の総和 (Sum of Absolute Differences) で計算する *SAD*()、視差から距離を求める *Distance*()、類似度の高いブロックを探索する *Search*()、そしてメイン関数である *StereoMatch*() を定義している。*StereoMatch*() では、$\bigoplus$ 演算子で左画像 L からサンプルブロック、右画像 R から候補ブロックを切り出す。*Search*() では、候補ブロック r から、サンプルブロック l と同じ大きさのブロックを切り出し、最もサンプルブロック l と類似度の高いブロック、すなわち *SAD*() の値が小さいブロックを $\vec{\nabla}$ で探索し、サンプルブロックとの座標の差を計算する。そして、*Distance*() で座標の差の長さ $\|parallax\|$、カメラ間の距離 d、フォーカス値 f から距離を計算する。

Search() が条件式になっているのは、サンプルブロック内の画像が平坦で正しいマッチングが取れないブロックの計算を省くためである。 なお、*Search*() 内で用いている $\star$ 演算子は、マップ関数呼び出しの引数に対して作用する演算子である。マップ関数呼び出しは、通常は引数の要素に対して関数が適用されるのに対し、$\star$ 演算子が付けられた引数は、引数そのものに関数が適用される。

3.4 連結演算子

$\bigodot$ 演算子や $\bigoplus$ 演算子による部分配列の切り出しを使って、実行すべき処理をより小さな処理に分解して定義してゆくのが本言語の記述スタイルであるが、分解した個々の処理結果を統合することも必要になる。それを行うのが、配列を連結して新しい配列を生成する連結演算子である。本言語では、2 つの配列を連結する連結演算子 ($\boxdot$) と、配列の配列に対して各要素配列を並べて連結した配列を生成する繰り返し連結演算子 ($\boxplus$) を導入する。

$\boxdot$ 演算子は、以下のように二つの配列をパラメータで指定した軸方向に連結した 1 つの配列を返す。

$$\begin{bmatrix} 0 & 1 \end{bmatrix} \boxdot_1 \begin{bmatrix} 2 & 3 \end{bmatrix} \equiv \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \qquad \begin{bmatrix} 0 \\ 1 \end{bmatrix} \boxdot_2 \begin{bmatrix} 2 \\ 3 \end{bmatrix} \equiv \begin{bmatrix} 0 & 2 \\ 1 & 3 \end{bmatrix}$$

一方の $\boxplus$ 演算子は、以下のように深さが 2 段以上の配列に対し、外側から 2 段目の配列の要素を 1 段目の配列の要素に展開することで 1 段浅い配列を作る。

$$\boxplus \begin{bmatrix} [0 & 1] & [2 & 3] \\ [4 & 5] & [6 & 7] \end{bmatrix} \equiv \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix}$$

これを使用した例として、図 3 に高速フーリエ変換 (FFT) の計算の一部であるバタフライ演算の記述例を示す。*Butterfly*() がメイン関数であり、*Iterate*() が、*Cross*() を再帰呼び出しを用いて繰り返し呼び出す。*Cross*() は、たすき掛けを計算する関数であり *Iterate*() に属する **where** 節にて定義している。**where** 節とは、それが属している式からのみ参照可能な関数、変数を定義する構文である。また、*Iterate*() では再帰呼び出しを行っているが、本言語の再帰呼び出しはスタックを消費しないループに変換可能な末尾再帰にのみ対応している。

Iterate() と *Cross*() の処理の流れを示したのが図 4 である。*Cross*() は、引数としてデータ列 *sub* とフーリエ係

$$Iterate(V,W,size,count) \leftarrow \begin{cases} V & \textbf{when} \quad size = 1 \\ Iterate\left(\boxplus\forall Cross\left(\bigoplus_{\langle 0\rangle\#\langle size\rangle}^{\langle count\rangle\#\langle size\rangle} V, \star W\right), W, half, 2\times count\right) & \textbf{otherwise} \end{cases}$$

$$\textbf{where} \quad half \leftarrow size/2$$

$$Cross(sub,W) \leftarrow \left(U+L\right) \underset{1}{\boxdot} \left((U-L)\times Wh\right)$$

$$\textbf{where} \quad U \leftarrow \bigodot_{\langle 0\rangle}^{\langle half\rangle} sub; \quad L \leftarrow \bigodot_{\langle half\rangle}^{\langle half\rangle} sub; \quad Wh \leftarrow \boxplus \bigoplus_{\langle 0\rangle\#\langle count\rangle}^{\langle half\rangle\#\langle 1\rangle} W$$

$$Butterfly(V_{\langle *\rangle\#cdouble})_{\langle *\rangle\#cdouble} \leftarrow Iterate(V,W,size,1)$$

$$\textbf{where} \quad size \leftarrow \rho_1(V); \quad W \leftarrow \cos\left(\frac{2\times\pi\times\iota(size)}{size}\right) - \sin\left(\frac{2\times\pi\times\iota(size)}{size}\right)\times \mathrm{i}$$

図 3 FFT で用いられるバタフライ演算プログラムの記述例

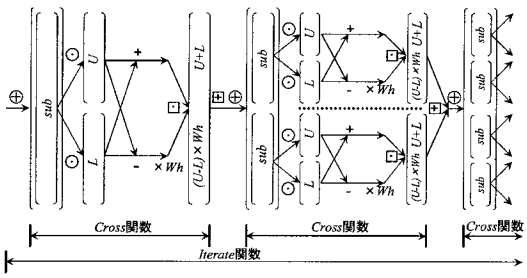

図 4 バタフライ演算の処理の流れ

数 W を受け取る。そして、$\bigodot$ 演算子を使って sub を上半分配列 U と下半分配列 L に分割し、返り値の上半分を $U+L$ で、下半分を $(U-L)\times Wh$ でそれぞれ計算し、$\boxdot$ 演算子で連結する。このときの Wh は、渡されたフーリエ係数 W のうち、$\bigoplus$ 演算子で必要な要素を選択し、$\boxplus$ 演算子で連結したものである。

$Cross()$ は、$Iterate()$ により繰り返し呼び出されるが、繰り返しの回数が進むごとに、たすき掛けの分割数は 2 倍になり、演算対象のデータ列の大きさは半分になる。そのため、$Iterate()$ では、$Cross()$ のマップ関数呼び出しの返り値を $\boxplus$ 演算子でフラットな配列に変換し、$\bigoplus$ で内側の配列の個数が 2 倍、大きさが半分になるように再分割を行う。

なお、メイン関数 $Butterfly()$ の **where** 節でフーリエ係数を求めるのに ι 演算子を用いている。ι 演算子は、0 から順に引数に指定された個数の整数を並べて要素としたベクトルを生成する演算子である。また、ρ 演算子で右下添え字でパラメータを指定しているのは指定した軸の長さを計算するためであり、型宣言の $cdouble$ は複素数型を意味している。

こうして記述したバタフライ演算のプログラムに対し、その計算結果を並べ直す関数 $Sort()$ を定義して組み合わせると、1 次元 FFT のプログラムは次のように記述できる。

$$FFT1D(V_{\langle *\rangle\#cdouble})_{\langle *\rangle\#cdouble} \leftarrow Sort(Butterfly(V))$$

$Sort()$ 関数は、$\bigoplus$ 演算子と $\boxplus$ 演算子を使って、$Butterfly()$ と同様に配列の要素の入れ替えを行うことで実現できる。

3.5 軸演算子

本言語では任意の次元の配列型を定義できるので、配列の軸に対する柔軟な操作ができれば、より多様なアルゴリズムを記述できる。そこで、軸を操作する ◎ 演算子を提供している。◎ 演算子は、軸の追加や削除、および、軸の向きを変更する演算子である。軸に対する操作の指定は ◎ 演算子のパラメータで行う。軸を削除する場合は、削除する軸以外の残す軸の番号を列挙したベクトルを指定する。例えば、形が $\langle 16,1\rangle$ の 2 次元配列を形が $\langle 16\rangle$ のベクトルに変換するには、第 1 軸を残せばよいのでパラメータに $\langle 1\rangle$ と指定する。反対に軸を追加する場合は、追加する軸に 0 を指定する。例えば、形が $\langle 16\rangle$ のベクトルを長さが 1 の第 2 軸を追加して 2 次元配列にする場合は、パラメータに $\langle 1,0\rangle$ と指定する。軸の向きを変えるには、軸の番号を負の数で表現する。例えば、1 次元配列の向きを逆にするには、パラメータに $\langle -1\rangle$ を指定すればよい。また、パラメータにベクトルを列挙すれば、内側の配列に対しても軸操作が可能である。

具体例として ◎ 演算子による 1 次元 FFT のプログラムの 2 次元拡張について考える。2 次元配列に対して FFT を適用するには、2 次元配列から第 1 軸 (列方向) に沿ったベクトルを切り出し、マップ関数呼び出しを使って 1 次元 FFT を適用した結果を $\boxplus$ 演算子で連結する。その結果を、今度は第 2 軸 (行方向) に沿ったベクトルで切り出し同様の処理を行う。このとき、2 次元配列から $\bigoplus$ 演算子で切り出した部分配列は 2 次元配列であり、1 次元 FFT は入力データを 1 次元配列と想定しているため次元の不一致が生じる。そこで、◎ 演算子により $\bigoplus$ 演算子で切り出した部分配列を 2 次元配列から 1 次元配列に変換し次元の不一致を解消する。そして、1 次元 FFT のマップ関数呼び出し結果を $\boxplus$ 演算子で連結するために、外側の 1 次元配列と内側の 1 次元配列をそれぞれ 2 次元に拡張する。こうすることで、2 次元 FFT は以下のように記述できる。

$$FFT2D(M_{\langle *,*\rangle\#cdouble})_{\langle *,*\rangle\#cdouble}$$

$$\leftarrow \boxplus \underset{\langle 1,0\rangle\#\langle 0,1\rangle}{\circledcirc} \forall FFT1D\left(\underset{\langle 1\rangle\#\langle 2\rangle}{\circledcirc} \bigoplus_{\langle 0,0\rangle\#\langle 1,0\rangle}^{v\#h} \boxplus \underset{\langle 0,1\rangle\#\langle 1,0\rangle}{\circledcirc} \forall FFT1D\left(\underset{\langle 2\rangle\#\langle 1\rangle}{\circledcirc} \bigoplus_{\langle 0,0\rangle\#\langle 0,1\rangle}^{h\#v} M\right)\right)$$

$$\textbf{where} \quad v \leftarrow \langle \rho_1(M), 1\rangle; \quad h \leftarrow \langle 1, \rho_2(M)\rangle$$

3.6 リダクション関数呼び出しとスキャン関数呼び出し

マップ関数呼び出しを用いれば配列の要素全体に対する並列処理を記述できるが、記述したいアルゴリズムによっては配列の要素に対する逐次的な処理が必要な場合もある。このような処理を記述するための関数適用形式として、本言語はリダクション関数呼び出し ($\triangleright$) と逆リダクション関数呼び出し ($\triangleleft$)、およびスキャン関数呼び出し ($\trianglerighteq$) と逆スキャン関数呼び出し ($\trianglelefteq$) を備えている。

リダクション関数呼び出しを用いれば、次に示すように、配列を走査しながら要素に対して関数を逐次的に適用していく処理を記述できる。

$$\triangleright f(init, M) \equiv f(\cdots f(f(init, M[0,0]), M[0,1]), \cdots)$$

最初に、初期値 $init$ と第 2 引数の配列 M の 0 行 0 列の要素 $M[0,0]$ に対して関数 f を適用し、続いて、その結果 $f(init, M[0,0])$ と第 2 引数の配列 M の 0 行 1 列の要素 $M[0,1]$ に対して再び関数 f を適用する。このイテレーションを配列 M のすべての要素に対して繰り返し、最後のイテレーションの関数 f の返り値をリダクション関数呼び出しの返り値とする。リダクション関数呼び出しにより、アグリゲーション演算子のように配列から集約された値を求める関数を定義できるようになる。

リダクション関数呼び出しの例として、パターン認識における画像の特徴ベクトルとして使われる 2 値画像の重心を求めるアルゴリズムの記述例を示す。

$$SumCoord(s, p, index) \leftarrow \begin{cases} s + index & \textbf{when} \quad p = 1 \\ s & \textbf{otherwise} \end{cases}$$

$$ImageGravity(M_{\langle *,* \rangle \#uint8})_{\langle 2 \rangle \#int} \leftarrow \triangleright SumCoord\big(\langle 0,0 \rangle, M, \iota(\rho(M))\big) \Big/ \sum M$$

この記述例では、値が 1 の画素の座標の総和を $\triangleright Coord()$ で求め、それを値が 1 の画素の総数で割って重心を求める。$Coord()$ には、座標の総和の初期値 $\langle 0,0 \rangle$ と、2 値画像 M、座標を要素とする配列 $\iota(\rho(M))$ を引数として与える。このときの ι 演算子は、引数がベクトルであるため、次のように形が引数のベクトルで、要素が座標である配列を返す。

$$\iota(\langle 2,3 \rangle) \equiv \begin{bmatrix} \langle 0,0 \rangle & \langle 0,1 \rangle & \langle 0,2 \rangle \\ \langle 1,0 \rangle & \langle 1,1 \rangle & \langle 1,2 \rangle \end{bmatrix}$$

リダクション関数呼び出しが最後の結果のみ返り値として返すのに対し、スキャン関数呼び出しは、中間結果をすべて返す関数適用方式である。スキャン関数呼び出しは、以下に示すように中間結果を引数の配列の要素と同じ座標に格納した 1 つの配列を返り値として返す。

$$\trianglelefteq f(init, V) \equiv \langle f(init, V[0]),\ f(f(init, V[0]), V[1]),\ \ldots \rangle$$

なお、逆リダクション関数呼び出しと逆スキャン関数呼び出しはどちらも引数の配列の要素を逆の順番で取り出して関数適用を行う。

3.7 画像処理以外のアルゴリズムへの適用

本言語は、画像処理を主なアプリケーションとして想定しているが、それ以外の分野にも応用可能である。

例えば、熱伝導方程式の解を求める方法として陽解法という手法がある[5]。陽解法では、時刻 t の熱分布から、時刻 $t + dt$ の熱分布を求める。このとき、ある場所の時刻 $t + dt$ の熱量を求めるときに必要な入力は、その場所の近傍の時刻 t の熱分布である。そのため、画像処理の近傍処理と同様に $\bigoplus$ 演算子で近傍を部分配列で切り出し、その値を参照してから $t + dt$ の熱量を求める。この処理を再帰呼び出しで繰り返せば、次のように記述できる。

$$dt \leftarrow 0.004; \qquad dx \leftarrow 0.1; \qquad r \leftarrow dt/(dx)^2$$

$$Next(sub) \leftarrow \sum (\langle r, 1 - 2 \times r, r \rangle \times sub)$$

$$Iterate(V, t) \leftarrow \begin{cases} V & \textbf{when} \quad t > 1.0 \\ Iterate\left(\forall Next\left(\bigoplus_{\langle -1 \rangle \#\langle 1 \rangle}^{\rho(V)\#\langle 3 \rangle} V\right), t + dt\right) & \textbf{otherwise} \end{cases}$$

$$HeatConduct(V_{\langle * \rangle \#double})_{\langle * \rangle \#double} \leftarrow Iterate(V, 0)$$

また、可視化処理の 1 つであるボリュームレンダリングも本言語で記述できる。ボリュームレンダリングは、3 次元空間上の濃度を表現したボクセルデータを平面上に投影したときの光量を計算することで、空間濃度を平面で表現する技術である[4]。投影する平面の単位面積当たりの光量 Ray_i は平面の垂線と交わる k 番目の単位体積当たりの濃度を α_k、それに対するカラー値を α_k の関数 $C(\alpha_k)$ とすると次の式で計算できる。

$$Ray_i = \sum_{k=0}^{K} \left(C(\alpha_k) \prod_{p=k}^{K} (1 - \alpha_p) \right)$$

これを本言語で記述するときのポイントは、$\sum$ の中に $\prod$ が現われている点である。$\prod$ のインデックス変数 p の初期値が $\sum$ のインデックス変数 k を使って $p = k$ となっているため、k が 1 増える毎に $\prod$ の計算する項の数が 1 ずつ減っていく。本言語では、逆スキャン関数呼び出しを用いて以下のように記述できる。

$$Ray(\alpha) \leftarrow \sum \Big(C(\alpha) \times \big(\trianglelefteq\times(1, 1 - \alpha) \big) \Big)$$

$$VR(M_{\langle *,*,* \rangle \#double})_{\langle *,* \rangle \#double} \leftarrow \forall Ray \left(\bigodot_{\langle 1,2 \rangle \#\langle 3 \rangle}^{\langle \rho_1(M), \rho_2(M), 1 \rangle \#\langle 1,1,\rho_3(M) \rangle} \bigoplus_{\langle 0,0,0 \rangle \#\langle 1,1,0 \rangle} M \right)$$

$Ray()$ では、$\prod$ の計算する項が 1 ずつ減っていくのを、$\trianglelefteq\times$ で総積の中間結果を残すことで表現している。

メイン関数の $VR()$ では、入力した 3 次元ボクセルデータから第 3 軸と同じ向きのベクトルを切り出し、各々のベクトルに対して $Ray()$ を適用している。

4. マルチコアアーキテクチャへの最適化方式

本言語で記述したプログラムは、処理系によりマルチコアアーキテクチャ向けに最適化された C/C++ のソースコードに変換する。このとき、マルチコアアーキテクチャ上で性能を出すには、処理の並列化、コア間の負荷分散、

高速な内部メモリの活用がキーポイントとなる。これを以下のステップで変換を行うことで実現する。なお、最適化方式の詳細については、文献[1),2)]を参照されたい。

4.1 同期ポイントで処理を分割

最初に、本言語で記述されたプログラムを、同期が必要なポイントで分割する。ここでいう同期とは、並列実行の処理の結果を集めるための待ち合わせ処理のことである。同期が必要なポイントは、アグリゲーション演算や、再帰関数呼び出し、リダクション関数呼び出し、スキャン関数呼び出しを行っている箇所であるので、簡単に見つけることができる。アグリゲーション演算は最後の集約した結果を計算するときに同期が必要となる。また、再帰関数呼び出し、リダクション関数呼び出し、スキャン関数呼び出しは、関数を逐次的に繰り返し呼び出すが、このとき呼び出す関数内で並列処理が実行される場合があるため、関数を繰り返し呼び出すたびに同期が必要になる。

4.2 並列化可能な処理単位（パラレルユニット）の抽出

次に、分割した処理から並列実行可能な処理単位（パラレルユニット）を抽出する。パラレルユニットは、配列に対する基本演算子の適用やマップ関数呼び出し、およびアグリゲーション演算のオペランドの評価を行っている箇所を手がかりにして簡単に見つけることができる。基本演算子やマップ関数呼び出しは、要素単位への演算や関数の適用が互いに独立であるため、各々の適用処理をパラレルユニットとして抽出できる。アグリゲーション演算も、オペランドに指定された配列型の式の各要素の演算間、あるいはそれらを複数のグループに分けて部分的に集約する演算間には依存性が無いため、パラレルユニットとして抽出できる。

4.3 抽出したパラレルユニットの分配

抽出したパラレルユニットは各コアに分配して実行する。並列に処理可能なパラレルユニットの数は処理の内容によっては非常に多くなり、パラレルユニットを1つずつコアに分配して並列処理すると粒度が細かすぎてオーバヘッドが大きくなってしまう。たとえば簡単な画像フィルタの場合には画素単位の処理がパラレルユニットになる。そこで、隣接するパラレルユニットをグループ化し、それを分配の単位とする。このグループをレンジと呼ぶ。

レンジのサイズ、すなわち1つのレンジに属するパラレルユニットの数は、パラレルユニット毎のメモリアクセス量と各プロセッサコアの内部メモリ（キャッシュやローカルメモリ）の容量から算出する。パラレルユニット毎のメモリアクセス量は、$\bigoplus$ 演算子のパラメータや、配列の要素のデータ型の大きさから簡単に計算することができる。

こうして求めたレンジのプールを用意し、その中から各コアの負荷状況に応じて順次レンジを分配することで、各コアの負荷を均等化する。このとき、各コアには隣接するレンジを分配するように分配順序を制御し、内部メモリに既に読み込まれているデータを再利用できるようにすることで、内部メモリを効率的に利用できるように処理順序を最適化することができる。

5. 関連研究

本言語と同様に配列データを主な処理対象にしているプログラミング言語にAPL[6)]がある。APLは、基本演算の要素毎の適用や、配列の構造を操作する演算子を提供するなど記述力が高く本言語も参考にしているところが多い。

一方で、C/C++の自動並列化の限界や、並列処理用に最適化されたソースコードのメンテナンス性の低さからCt[7)]やFortress[8)]といったマルチコア向けのプログラミング環境が注目されている。

Ctは並列処理用のC++テンプレートライブラリで、並列処理用のデータ型と専用の演算子を提供している。配列処理言語は、基本的に $\bigoplus$ 演算子を用いて対象配列を分割し、トップダウン的にアルゴリズムを記述していくのに対して、Ctは配列の要素へのアクセスから処理を積み上げるボトムアップ的な記述を行うため、アルゴリズムの詳細化の仕方が異なってくる。

Fortressは、本言語と同様にマルチコア向けのプログラミング言語である。Fortressでは、並列処理用のループ構文を提供することで、並列処理を簡潔に記述することを実現しており、本言語と記述スタイルが異なる。

6. おわりに

本稿では、配列処理に特化したドメイン特化言語を用いたマルチコアプログラミングに関して述べた。本言語を用いれば、最適化を意識することなくプログラミングできるため、より本質的なアルゴリズムの開発に注力することができる。また、画像処理や物理シミュレーション等の数式表現に近い表現でプログラミングできるため、マルチコアプログラミングに対して敷居の高かったユーザ層を取り込むことが期待できる。

今後の課題は、様々なアーキテクチャへの対応と応用領域の拡張である。本言語は、様々な粒度での並列性が抽出しやすいため、マルチコア向けの並列化だけでなく、SIMD化といった細粒度の並列化や、GPGPUにも適用可能と考えている。実際、近傍処理におけるSIMD化に成功しており、性能改善が実現できている[3)]。一方、応用領域の拡張に向けては、線形代数の各種演算子や疎行列への対応を検討しなければならない。

参考文献

1) 金井 他, “組み込みプロセッサのメモリアーキテクチャに依存しない画像処理プログラムの記述と実行方式”, 情報処理学会論文誌コンピューティングシステム, Vol. 48, No. SIG 13(ACS 19), pp. 287–301, Aug. 2007.
2) Jun'ichi Segawa, et al., “The Array Processing Language and the Parallel Execution Method for Multicore Platforms”, The First International Symposium on Information and Computer Elements(ISICE2007), pp. 98–103, Sep. 2007.
3) 城田 他, “配列処理言語におけるSIMD化向けプログラム変換”, SWoPP2008(掲載予定).
4) 鳥脇純一郎, 3次元ディジタル画像処理, 明晃堂, 2002.
5) 伊理 他, 数値計算の常識, 共立出版, 1985
6) 橘川 他, 初めて学ぶAPL, 共立出版, 1994.
7) Anwar Ghuloum, et al., “Future-Proof Data Parallel Algorithms and Software on Intel Multi-Core Architecture”, Intel Technology Journal Vol.11, issue 04, Nov. 2007.
8) Eric Allen, et al., “ProjectFortress: A Multicore Language for Multicore Processors”, Linux Magazine, Sep. 2007.