二次元配列上の構成的並列スケルトンの実現

Constructive Parallel Skeletons on Two-Dimensional Arrays

江本 健斗[†]，胡 振江[†‡]，筧 一彦[†]，武市正人[†]
Kento EMOTO, Zhenjiang HU, Kazuhiko KAKEHI and Masato TAKEICHI
[†] 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, University of Tokyo
[‡] 科学技術振興機構 さきがけ研究 21
PRESTO 21, Japan Science and Technology Agency

行列演算のような二次元配列上の演算は，計算科学及びその様々な応用分野において基本的かつ普遍的なものである．PC クラスタのような並列計算機環境が一般的になるにつれ，並列計算機に対する知識をあまり持たない者でさえも二次元配列上の並列演算を簡単にかつ効率的に記述できるような，新しいプログラミングモデルが要求され続けている．一方，並列化可能な操作の枠組を抽象化した並列スケルトンは，有用な並列プログラミングモデルを与えてくれることが知られている．しかしながら，殆どの研究は一次元のデータ（リスト）を扱うスケルトンを対象としており，二次元配列上のスケルトンの研究は少なく，僅かに存在する実装も定式化が不十分である．そこで本研究では，構成的アルゴリズム論に基づく二次元配列上のデータ並列スケルトンを設計し，効率的に実現する手法を探求する．さらに，設計したスケルトンの実装を行い，評価を行う．

1 はじめに

行列演算のような二次元配列上の演算は，計算科学及びその様々な応用分野において基本的かつ普遍的なものである．PC クラスタのような並列計算機環境が一般的になるにつれ，並列計算機に対する知識をあまり持たない者でさえも二次元配列上の並列演算を簡単にかつ効率的に記述できるような，新しいプログラミングモデルが要求され続けている．

並列化可能な操作の枠組を抽象化した並列スケルトンは，有用な並列プログラミングモデルを与えてくれることが知られている [7]．しかしながら，並列スケルトンに関する研究のほとんどは一次元のデータ（リスト）を扱うスケルトンを対象としており，二次元配列上のスケルトンの研究は少ない．また，僅かに存在する実装 [6] も定式化が不十分である．さらに，行列の演算に関しては，行列をブロックに分割するアルゴリズムが行列を列ないし行方向に分割するものより性能がよくなることが示されている [3]．しかし，既存の二次元配列上のスケルトンの研究の殆どは二次元配列のデータ型をリストのリストという型にしており，ブロックに分割するアルゴリズムの記述に不適切であると考えられる．

そこで本研究では，構成的アルゴリズム論に基づく二次元配列上の並列スケルトンを設計し，効率的に実現する手法を探求する．また，データ型として任意箇所で縦横にブロックに分割できる型を用い，ブロックに分割するアルゴリズムを自然に記述できることを期待する．さらに，設計したスケルトンの実装を行い，行列の基本的演算である行列乗算をそのスケルトンを用いて実装し，評価を行う．

2 二次元配列上の計算モデルの定式化

本研究で対象とする二次元配列のデータ型と，それを操作するスケルトンの定義をする．本論文の記法は Haskell[2] の記法に従う．また，特に断らない限り二項演算子は結合的とする．

2.1 データ型

基本となる二次元配列のデータ型 *Array* と，それを各プロセッサに分散したもののデータ型 *DArray* を定義する．

2.1.1 *Array*

三つの構成子 $|\cdot|$ (singlton), $\div$ (above), ϕ (beside) からなるデータ型 *Array* を定義する [1].

$$\begin{aligned}\textbf{data}\ \mathit{Array}\,\alpha = \quad & |\cdot|\ \alpha \\ | \quad & (\mathit{Array}\,\alpha) \div (\mathit{Array}\,\alpha) \\ | \quad & (\mathit{Array}\,\alpha)\ \phi\ (\mathit{Array}\,\alpha)\end{aligned}$$

ここで，$|\cdot|\,a$ は要素が a である 1×1 の *Array* を意味し，$|a|$ と略記する．また，*Array* の要素を取り出

す演算子として，the $|a| = a$ を定義する．$x \oplus\!\!\!\!- y$ は x が y の上に位置することを，$x \phi y$ は x が y の左に位置することを意味する．さらに，$\oplus\!\!\!\!-, \phi$ は結合的な二項演算子であり，次式で定義される abide の関係式を満たす．

$$(x \phi u) \oplus\!\!\!\!- (y \phi v) = (x \oplus\!\!\!\!- y) \phi (u \oplus\!\!\!\!- v)$$

以上の定義により，例えば，2×2 の二次元配列は次のように表現される．

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = (|1| \phi |2|) \oplus\!\!\!\!- (|3| \phi |4|) \\ = (|1| \oplus\!\!\!\!- |3|) \phi (|2| \oplus\!\!\!\!- |4|)$$

2.1.2 *DArray*

二次元配列 *Array* を分配したことを明示するためのデータ型 *DArray* を定義する．

$$\textbf{type} \quad DArray\,\alpha = Array\,(Array\ \alpha)$$

これは *Array* を要素にもつ *Array* であり，一つのプロセッサに一つの要素 $(Array)$ が存在しているとを意味する．次節で定義する通信スケルトンはこの *DArray* を引数にとる．

2.2 二次元配列上のスケルトン

上で定義したデータを操作するスケルトンを定義する．そのために，幾つかの記法を定義しておく．

関数適用は空白を用いて表現し，その引数には括弧を付けない．よって，$f\,x$ は通常の関数の記法では $f(x)$ を表す．全ての関数はカリー化されているとする．すなわち，全ての関数は関数か値を返す一引数関数であるとし，関数適用は左結合的である．よって，$g\,x\,y = (g\,x)\,y$ である．関数適用は二項演算子より結合順序が高く，$f\,a \otimes b = (f\,a) \otimes b$ である．関数合成は $\circ$ で表し，定義より $(f \circ g)\,x = f\,(g\,x)$ である．二項演算子はセクション化により，$a \oplus b = (a\oplus)\,b = (\oplus b)\,a = (\oplus)\,a\,b$ のように関数として用いることができる．任意の二項演算子 $\otimes$ に対し，その引数を逆にした演算子を $\tilde{\otimes}$ と書き，$a\,\tilde{\otimes}\,b = b \otimes a$ とする．組 (x, y) の要素に対して別々に関数 f, g を適用するような関数を $(f \times g)$ と書く．すなわち，$(f \times g)\,(x, y) = (f\,x, g\,y)$ とする．これは，組の要素数が増えても同様である．二つの二項演算子 $\ll$ と $\gg$ を，$a \ll b = a$，$a \gg b = b$ と定義する．

2.2.1 データ並列スケルトン

データ型 *Array* に対して自然に定義されるスケルトン map，red，zipw，scan を定義する．また，それらを用いて幾つかの関数を定義する．

関数 f を *Array* の各要素に対して適用するスケルトン map は次のように定義される．

$$\begin{aligned} \mathsf{map}\,f\,|a| &= |f\,a| \\ \mathsf{map}\,f\,(x \oplus\!\!\!\!- y) &= (\mathsf{map}\,f\,x) \oplus\!\!\!\!- (\mathsf{map}\,f\,y) \\ \mathsf{map}\,f\,(x \phi y) &= (\mathsf{map}\,f\,x) \phi (\mathsf{map}\,f\,y) \end{aligned}$$

二項演算子 $\oplus, \otimes$ が abide であるとき，*Array* を一つの値に縮約するスケルトン red は次のように定義される．

$$\begin{aligned} \mathsf{red}(\oplus, \otimes)\,|a| &= a \\ \mathsf{red}(\oplus, \otimes)\,(x \oplus\!\!\!\!- y) &= (\mathsf{red}(\oplus, \otimes)\,x) \oplus (\mathsf{red}(\oplus, \otimes)\,y) \\ \mathsf{red}(\oplus, \otimes)\,(x \phi y) &= (\mathsf{red}(\oplus, \otimes)\,x) \otimes (\mathsf{red}(\oplus, \otimes)\,y) \end{aligned}$$

二つの同じ形の *Array* の対応する要素同士に関数 f を適用して同じ形の新しい *Array* を作る関数 zipw は次のように定義される．

$$\begin{aligned} \mathsf{zipw}\,f\,|a|\,|b| &= |f\,a\,b| \\ \mathsf{zipw}\,f\,(x \oplus\!\!\!\!- y)\,(u \oplus\!\!\!\!- v) &= (\mathsf{zipw}\,f\,x\,u) \oplus\!\!\!\!- (\mathsf{zipw}\,f\,y\,v) \\ \mathsf{zipw}\,f\,(x \phi y)\,(u \phi v) &= (\mathsf{zipw}\,f\,x\,u) \phi (\mathsf{zipw}\,f\,y\,v) \end{aligned}$$

ただし，上式では x と u のサイズが同じになるように分割されるものとする．zipw の特殊形である対応する要素同士を組にした *Array* を作る関数 zip を次のように定義する．

$$\mathsf{zip}\,(u, v) = \mathsf{zipw}\,(\lambda xy.\,(x, y))\,u\,v$$

引数の *Array* が三つ以上の場合も同様に zip, zipw を定義し，k 個の *Array* を引数にとるものを $\mathsf{zipw}_k, \mathsf{zip}_k$ と書く．また，zip の逆演算を unzip とする．

以上のスケルトンを用い幾つかの関数を定義する．

$$\begin{aligned} id &= \mathsf{red}(\oplus\!\!\!\!-, \phi) \circ \mathsf{map}\,|\cdot| \\ tr &= \mathsf{red}(\phi, \oplus\!\!\!\!-) \circ \mathsf{map}\,|\cdot| \\ rev &= \mathsf{red}(\tilde{\oplus\!\!\!\!-}, \tilde{\phi}) \circ \mathsf{map}\,|\cdot| \\ flatten &= \mathsf{red}(\oplus\!\!\!\!-, \phi) \\ height &= \mathsf{red}(+, \ll) \circ \mathsf{map}\,(\lambda x.\,1) \\ width &= \mathsf{red}(\ll, +) \circ \mathsf{map}\,(\lambda x.\,1) \\ cols &= \mathsf{red}(\mathsf{zipw}(\oplus\!\!\!\!-), \phi) \circ \mathsf{map}\,||\cdot|| \\ rows &= \mathsf{red}(\oplus\!\!\!\!-, \mathsf{zipw}(\phi)) \circ \mathsf{map}\,||\cdot|| \\ \mathsf{red}_c(\oplus) &= \mathsf{map}(\mathsf{red}(\oplus, \ll)) \circ cols \\ \mathsf{red}_r(\otimes) &= \mathsf{map}(\mathsf{red}(\ll, \otimes)) \circ rows \\ \mathsf{map}_c\,f &= \mathsf{red}(\ll, \phi) \circ \mathsf{map}\,f \circ cols \\ \mathsf{map}_r\,f &= \mathsf{red}(\oplus\!\!\!\!-, \ll) \circ \mathsf{map}\,f \circ rows \end{aligned}$$

ここで，$||\cdot||$ は略記であり，$||\cdot|| = |\cdot| \circ |\cdot|$ とする．id は恒等関数，tr は転置関数，rev は上下左右の反転関数，$flatten$ はネストした $Array$ を平坦にする関数，$height, width$ はそれぞれ行数と列数を返す関数であり，$cols, rows$ はそれぞれ列，行を要素とする $Array$ を返す関数である．また，$\mathsf{red}_c, \mathsf{red}_r$ は列ないし行方向のみに縮約する red であり，$\mathsf{map}_c, \mathsf{map}_r$ は列，行を単位として関数を適用する map である．さらに，$\oplus, \otimes$ を abide な演算子としたとき $\mathsf{red}_c, \mathsf{red}_r$ を用いて red が表現できることにも注意する．

$$\begin{aligned}\mathsf{red}(\oplus,\otimes) &= \mathsf{the} \circ \mathsf{red}_c(\oplus) \circ \mathsf{red}_r(\otimes)\\ \mathsf{red}(\oplus,\otimes) &= \mathsf{the} \circ \mathsf{red}_r(\otimes) \circ \mathsf{red}_c(\oplus)\end{aligned} \tag{1}$$

二項演算子 $\oplus, \otimes$ を abide として，red の縮約過程を全て保存するスケルトン scan は次式で定義される．

$$\begin{aligned}&\mathsf{scan}(\oplus,\otimes)\,|a| &&= |a|\\ &\mathsf{scan}(\oplus,\otimes)(x \mathbin{\ominus} y) &&= (\mathsf{scan}(\oplus,\otimes)\,x) \oplus' (\mathsf{scan}(\oplus,\otimes)\,y)\\ &\mathsf{scan}(\oplus,\otimes)(x \mathbin{\oplus} y) &&= (\mathsf{scan}(\oplus,\otimes)\,x) \otimes' (\mathsf{scan}(\oplus,\otimes)\,y)\end{aligned}$$

ただし，$\oplus', \otimes'$ は以下のように定義される．

$$\begin{aligned}&bottom &&= \mathsf{red}(\gg, \oplus) \circ \mathsf{map}\,|\cdot|\\ &last &&= \mathsf{red}(\ominus, \gg) \circ \mathsf{map}\,|\cdot|\\ &sx \oplus' sy &&= sx \mathbin{\ominus} sy'\\ &\quad \textbf{where}\ sy' &&= \mathsf{map}_r\,(\mathsf{zipw}(\oplus)(bottom\,sx))\,sy\\ &sx \otimes' sy &&= sx \mathbin{\oplus} sy'\\ &\quad \textbf{where}\ sy' &&= \mathsf{map}_c\,(\mathsf{zipw}(\otimes)(last\,sx))\,sy\end{aligned}$$

そして，scan もまた red のように列ないし行方向のみの $\mathsf{scan}_\downarrow$, $\mathsf{scan}_\rightarrow$ が定義できる．

$$\begin{aligned}\mathsf{scan}_\downarrow(\oplus) &= \mathsf{scan}(\oplus, \gg)\\ \mathsf{scan}_\rightarrow(\otimes) &= \mathsf{scan}(\gg, \otimes)\end{aligned}$$

また，$\mathsf{scan}_\downarrow$，$\mathsf{scan}_\rightarrow$ で scan を表現することもできる．

$$\begin{aligned}\mathsf{scan}(\oplus,\otimes) &= \mathsf{scan}_\downarrow(\oplus) \circ \mathsf{scan}_\rightarrow(\otimes)\\ \mathsf{scan}(\oplus,\otimes) &= \mathsf{scan}_\rightarrow(\otimes) \circ \mathsf{scan}_\downarrow(\oplus)\end{aligned}$$

最後に，scan を逆向きに実行する scanr と，各行の全ての要素が red_r のその行の要素を持つような $allred_r$ と，同様に列方向の $allred_c$ を定義する．

$$\begin{aligned}\mathsf{scanr}(\oplus,\otimes) &= rev \circ \mathsf{scan}(\tilde{\oplus}, \tilde{\otimes}) \circ rev\\ allred_c(\oplus) &= \mathsf{scanr}(\gg, \ll) \circ \mathsf{scan}(\oplus, \gg)\\ allred_r(\otimes) &= \mathsf{scanr}(\ll, \gg) \circ \mathsf{scan}(\gg, \otimes)\end{aligned}$$

2.2.2 データ通信スケルトン

各プロセッサに $Array$ を分配するスケルトン dist と逆に各プロセッサからデータを集約して $Array$ を作るスケルトン gather，及び，分配したデータの再配置 (巡回) をする $\mathsf{rot}_r, \mathsf{rot}_c$ の定義を行う．

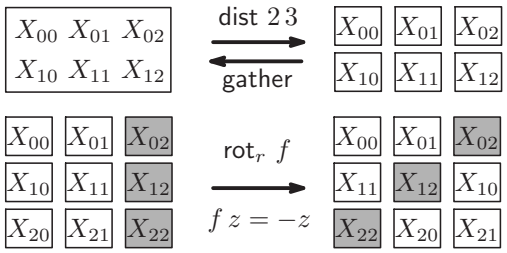

図 1: 通信スケルトンのイメージ

各プロセッサに $Array$ を分配して $DArray$ を作るスケルトン dist は次のように定義される．

$$\begin{aligned}&\mathsf{dist}\,p\,q\,x = (flatten \circ \mathsf{map}(grp_c\,n) \circ grp_r\,m)\,x\\ &\qquad \textbf{where}\ m = \lceil height\,x/p \rceil,\ n = \lceil width\,x/q \rceil\end{aligned}$$

ただし，grp_r は次式で定義し，grp_c も同様とする．

$$\begin{aligned}grp_r\,k\,(x \mathbin{\ominus} y) &= |x| \mathbin{\ominus} (grp_r\,k\,y) \quad \text{if}\ \ height\,x = k\\ grp_r\,k\,x &= |x| \quad \text{if}\ \ height\,x < k\end{aligned}$$

この dist の逆演算となる $DArray$ から $Array$ に戻すスケルトン gather は次のように定義される．

$$\mathsf{gather} = \mathsf{red}(\ominus, \oplus)$$

各行ごとに関数 f で与えられる量だけ要素を巡回させるスケルトン rot_r は次のように定義される．

$$\begin{aligned}&\mathsf{rot}_r\,f = flatten \circ \mathsf{map}\,shift_r \circ addidx_r \circ rows\\ &\quad \textbf{where}\\ &\qquad addidx_r\,u = \mathsf{zip}(\mathsf{map}\,f\,(idx_r\,u), u)\\ &\qquad idx_r = \mathsf{map}(-1) \circ \mathsf{scan}_\downarrow(+) \circ \mathsf{map}(\lambda x.\,1)\end{aligned}$$

ただし，$shift_r$ は，$i > 0$ として次式で定義される．

$$\begin{aligned}shift_r\,(0, x) &= x\\ shift_r\,(i, x \mathbin{\oplus} y) &= y \mathbin{\oplus} x \quad \text{if}\ width\,y = i\\ shift_r\,(-i, x \mathbin{\oplus} y) &= y \mathbin{\oplus} x \quad \text{if}\ width\,x = i\end{aligned}$$

上記のスケルトンのイメージを図 1 に示す．

3 実装

定義したスケルトン $\mathsf{map}, \mathsf{zipw}, \mathsf{red}, \mathsf{scan}$ が，各々並列に実行できることを述べる．そのために，$Array$ 上のスケルトンを式変形し，各プロセッサ上でのローカルな演算とプロセッサ間に跨る演算とに分離する．

まず，map について示す．

$$\begin{aligned}\mathsf{map}\,f &= \mathsf{map}\,f \circ \mathsf{gather} \circ \mathsf{dist}\,p\,q\\ &= \mathsf{map}\,f \circ \mathsf{red}(\ominus, \oplus) \circ \mathsf{dist}\,p\,q\\ &= \mathsf{red}(\ominus, \oplus) \circ \mathsf{map}\,(\mathsf{map}\,f) \circ \mathsf{dist}\,p\,q\\ &= \mathsf{gather} \circ \mathsf{map}\,(\mathsf{map}\,f) \circ \mathsf{dist}\,p\,q\end{aligned}$$

上式により，map f の演算は，map f の引数である $Array$ を $\mathsf{dist}\,p\,q$ で $DArray$ に変換し，各プロセッサ上で独立に map f をローカルの $Array$ に適用し，その結果を gather で集約すればよいことがわかる．これは，zipw についても同様である．

次に，red について示す．

$$\begin{aligned}\mathsf{red}(\oplus,\otimes) &= \mathsf{red}(\oplus,\otimes)\circ\mathsf{gather}\circ\mathsf{dist}\,p\,q\\ &= \mathsf{red}(\oplus,\otimes)\circ\mathsf{red}(\ominus,\oslash)\circ\mathsf{dist}\,p\,q\\ &= \mathsf{red}(\oplus,\otimes)\circ\mathsf{map}\,(\mathsf{red}(\oplus,\otimes))\circ\mathsf{dist}\,p\,q\end{aligned}$$

上式は，各プロセッサ独立にローカルの $Array$ に対して $\mathsf{red}(\oplus,\otimes)$ を実行し，その結果をプロセッサをまたいで $\mathsf{red}(\oplus,\otimes)$ で集約すればよいことを表している．また，式 (1) の性質から，プロセッサをまたぐ red は列方向，行方向の red を順に行えばよく，各方向の red はリスト上の red を複数並列に行えばよい．red の結果は $Array$ ではないため，map とは異なり最後に gather は必要でない．

最後に，scan について示す．

$$\begin{aligned}&\mathsf{scan}(\oplus,\otimes)\\ &= \mathsf{red}(\oplus',\otimes')\circ\mathsf{map}\,|\cdot|\circ\mathsf{gather}\circ\mathsf{dist}\,p\,q\\ &= \mathsf{red}(\oplus',\otimes')\circ\mathsf{map}\,|\cdot|\circ\mathsf{red}(\ominus,\oslash)\circ\mathsf{dist}\,p\,q\\ &= \mathsf{red}(\oplus',\otimes')\circ\mathsf{map}\,(\mathsf{red}(\oplus',\otimes')\circ\mathsf{map}\,|\cdot|)\circ\mathsf{dist}\,p\,q\\ &= \mathsf{red}(\oplus',\otimes')\circ\mathsf{map}\,(\mathsf{scan}(\oplus,\otimes))\circ\mathsf{dist}\,p\,q\\ &= \mathsf{gather}\circ\underline{\mathsf{dist}\,p\,q\circ\mathsf{red}(\oplus',\otimes')}\circ\\ &\qquad\qquad\mathsf{map}\,(\mathsf{scan}(\oplus,\otimes))\circ\mathsf{dist}\,p\,q\end{aligned}$$

ここで，scan の結果は $Array$ となるため，red とは異なり最後に gather を必要とすることに注意する．上式は，各プロセッサ独立にローカルの $Array$ に対して $\mathsf{scan}(\oplus,\otimes)$ を実行し，その結果をプロセッサをまたいで $\mathsf{red}(\oplus',\otimes')$ で集約すればよいことを表している．ただし，red の後にある $\mathsf{dist}\,p\,q$ により，結果はまた $DArray$ となる．下線部のプロセッサをまたぐ $\mathsf{dist}\,p\,q\circ\mathsf{red}(\oplus',\otimes')$ の計算は，今の記述力ではこれ以上簡単に書くことはできないが，red と同様に行方向，列方向に順に計算を行えばよく，各方向の計算はリスト上の scan の計算 [4] と同様にして行うことができる．

4 実験・評価: 行列乗算

定義した $Array$ 上のスケルトンを MPI と C++のテンプレートで実装し，そのスケルトンを用いて行列乗算を二通りに記述し実装した．

一つ目は，Cannon's Algorithm[5] をスケルトンで記述したものである．

$$\begin{aligned}mm_C = {}&\mathsf{gather}\circ(\mathsf{map}\;thd)\circ(iter\,p\,step)\circ\\ &\mathsf{zip}_3\circ arrange\circ distribute\circ init\\ &\textbf{where}\\ &init(A,B) = (A,B,\mathsf{map}(\lambda x.0)\,A)\\ &distribute = (\mathsf{dist}\,p\,p\times\mathsf{dist}\,p\,p\times\mathsf{dist}\,p\,p)\\ &arrange = (\mathsf{rot}_r\;neg\times\mathsf{rot}_c\;neg\times id)\\ &step = \mathsf{zip}_3\circ rearrange\circ\\ &\qquad\mathsf{unzip}_3\circ\mathsf{map}\;lmm\\ &rearrange = \mathsf{rot}_r(\lambda x.1)\times\mathsf{rot}_c(\lambda x.1)\times id\\ &neg\;x = -x\\ &thd\;(x,y,z) = z\end{aligned}$$

ただし，p は自然数であり，lmm は各プロセッサ上にある $Array$ に対して行列の乗算をするプログラム $lmm\,(A,B,C) = (A,B,C+A\times B)$ である．また，$iter$ は反復を表し次のように定義される．

$$\begin{aligned}iter\,k\,f\,x &= x && \text{if}\;\; k=0\\ iter\,k\,f\,x &= iter\,(k-1)\,f\,(f\,x) && \text{if}\;\; k>0\end{aligned}$$

この記述は，データ通信スケルトンによりデータの分配を陽に用いており，複雑な記述となっている．

二つ目は，データ並列スケルトンのみを用いた直感的に理解しやすい簡潔な記述である．

$$\begin{aligned}mm = {}&\mathsf{zipw}_P\;iprod\circ(id\times\mathsf{map}\;tr)\\ &\circ(allrows\times allcols)\\ &\textbf{where}\\ &allrows = allred_r(\oslash)\circ\mathsf{map}\,|\cdot|\\ &allcols = allred_c(\ominus)\circ\mathsf{map}\,|\cdot|\\ &iprod = (\mathsf{red}(\ll,+)\circ)\circ\mathsf{zipw}(\times)\\ &\mathsf{zipw}_P\,(\otimes)\,(x,y) = \mathsf{zipw}\,(\otimes)\,x\,y\end{aligned}$$

上記の二つの記述をスケルトンを用いて実装し，PC クラスタ[1]上で実行し計算時間の測定を行った．ただし，mm の実装は二つあり，データの無駄な複製を抑制する特別な $allred_r$，$allred_c$ を用いた実装を mm' とする．まず，行列のサイズを 200×200 に固定して使用するプロセッサ数を変化させて測定した．測定結果を図 2 に示す．横軸がプロセッサ数，縦軸が計算時間（秒）であり，両軸ともに対数を取っている．いずれも計算時間がプロセッサ数にほぼ反比例しており，台数効果がはっきりと表れている．こ

[1] Pentium 4 Xeon 2.0-GHz dual-processor, メモリ 1GB の PC を Gigabit Ethernet で接続．コンパイラ及び MPI には gcc 2.95，MPICH 1.1.2 を使用． OS は FreeBSD 4.8 を使用．

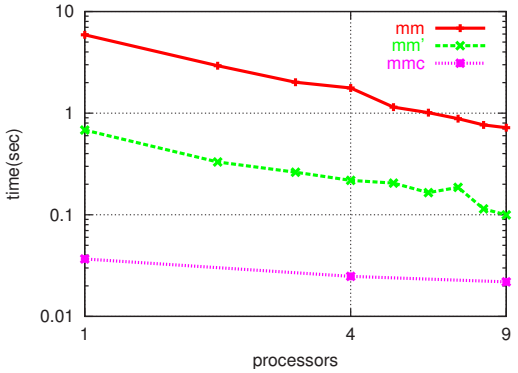

図 2: プロセッサ数による計算時間の変化

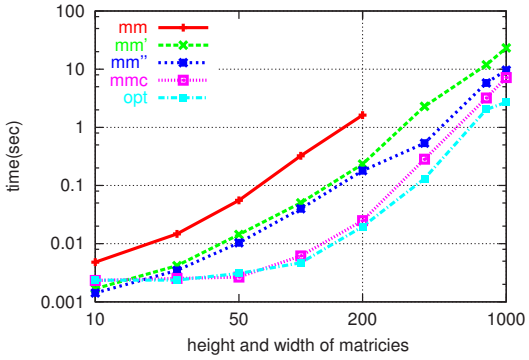

図 3: 問題のサイズによる計算時間の変化

れは，スケルトンによる記述により並列化の効果が得られることを示している．

次に，プロセッサ数を 4 に固定し行列のサイズを変化させて測定を行った．測定結果を図 3 に示す．横軸が行列のサイズ，縦軸が計算時間（秒）であり，両軸ともに対数を取っている．いずれも計算時間が行列のサイズの 3 乗に比例しており，一般的な行列乗算と同じ計算量である．通信スケルトンを用いた mm_C は，キャッシュを考慮して最適化した実装（opt）に対して 2 倍程度の計算時間である．これは，スケルトンを用いたプログラミングが現実的であることを示している．データ並列スケルトンのみの単純な実装 mm は，行列要素の無駄な複製により mm_C の数 10 倍程度時間がかかる．また，メモリの大量消費によりサイズが 400 程度で限界となる．しかし，特別な $allred_r$，$allred_c$ の実装を用いた mm' は，mm と同じ記述であるが mm_C の数倍程度の計算時間である．さらに，$iprod$ において zipw のあとに無駄な中間データを作らない特殊な zipw を用いることで，mm' は mm_C とほぼ同じ計算時間となる（mm''）．これは，効率的に実装できる部分の括り出しや，中間データの削減によりスケルトンで記述されたプログラムが効率化できることを示している．

5 おわりに

二次元配列のデータ型を定義し，その上の並列スケルトンの定義と実装をした．そして，それらを用いて行列の基本演算である行列乗算を記述し，スケルトンによる並列化の効果を確認した．特に，データ並列スケルトンのみの記述では並列計算に対する知識を全く必要とせず，現実的な効率の良い実装を得ることができた点は重要である．

データ並列スケルトンのみの記述は簡潔であり，また，$id = \mathsf{gather} \circ \mathsf{dist}\ p\ q$ である限り二次元配列のプロセッサへの分配の仕方によらず正しく動き，構成的である．しかし，単純な実装では効率が悪いこともある．一方，データ通信スケルトンなども用いた記述は複雑であり，データ分配の仕方も制限され得るが，ブロック分割を利用するアルゴリズムをデータの複製を抑えて記述することができ効率が良くなる．両者の利点である簡潔な記述と効率の両立は一般には難しいが，データ並列スケルトンによる記述はプログラム変換が適用しやすく，変換による効率的なプログラムの導出が考えられる．よって，今後の課題としては，効率化のための変換規則の導出，ブロック分割の再帰型アルゴリズムをデータ並列スケルトンで記述するための変換規則の導出，及びそれらの記述に適したスケルトンの追加が考えられる．

参考文献

[1] R. S. Bird. Lectures on Constructive Functional Programming. Oxford University Programming Research Group Monograph PRG-69, 1988.

[2] R. S. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998.

[3] J. D. Frens and D. S. Wise. QR Factorization with Morton-Ordered Quadtree Matrices for Memory Re-use and Parallelism. In *Proc. 2003 ACM Symp. on Principles and Practice of Parallel Programming*, pages 144–154, 2003.

[4] S. Gorlatch. Systematic Efficient Parallelization of Scan and Other List Homomorphisms. In *EuroPar'96, LNCS 1124*, pages 401–408. Springer-Verlag, Aug. 1996.

[5] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, second edition, 2003.

[6] H. Kuchen. A Skeleton Library. In *EuroPar'02, LNCS 2400*, pages 620–629. Springer-Verlag, Aug. 2002.

[7] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2002.