並列処理に適した配列範囲記述子 quad の提案と評価

米澤直記[†] 和田耕一[††]

プログラマに対して共有メモリを提供するプログラミング言語を，計算機クラスタなどの分散メモリ環境に実現する手段として，コンパイラによって共有メモリへのアクセス情報を収集して，読み出しアクセス前までに，データを消費するプロセッサのメモリ領域にそのデータを移動させる方法が考えられる．この方法は，データアクセス範囲を適切に表現し，それに基づいて移動すべきデータを特定することで，データ転送を必要最小限に抑えることが可能である．本論文では，コンパイラ内部でアクセス情報を表現するための配列範囲記述子 quad を提案する．quad は，並列プログラムの実行中に発生するアクセス・パターンを簡潔に表現できる特長を持つ．コンパイラは，依存関係にあるデータを適切に移動させるため，まず書き込みアクセスおよび読み出しアクセスを表現する quad 間の積演算により転送すべき配列範囲を求め，対応する通信コードを生成する．本論文では，複数の並列応用プログラムを用いて，実行中にネットワークを介して転送されたデータ転送量，および記述子間の演算コストを基に quad の有効性を評価した．その結果，quad は並列プログラムで出現する典型的なアクセス・パターンを簡潔に表現することができ，従来の配列範囲記述子 BRS1 に比べて記述子間の演算コストを，BRS2 に比べてデータ転送量を大幅に軽減できることを示した．

quad: an Array Section Descriptor for Parallel Computing

Naoki Yonezawa[†] and Koichi Wada[††]

One of the approaches to implement programming languages which provide shared memory for programmers on distributed memory environment is communication code generation, in which a producer sends appropriate data to the consumer. In this approach, the optimal use of network can be achieved by precisely identifying the necessary data to be transferred. This requires a compiler represents the exact range of accessed arrays. In this paper, we proposed a new array section descriptor, called *quad*, which can concisely represent array sections that are accessed in executing parallel programs. To identify data to be transferred, the compiler generates quads that represent written array section and read array section, and then generates codes for intersection operation between those quads. At runtime, according to the results of the intersection operation, the generated communication code sends the data to satisfy the dependency. For evaluation, we executed several parallel application programs using the quad and a conventional array section descriptor, and measured the amount of data transferred and the computational cost of the operations between descriptors. The results showed that the quad represents various access patterns typically observed in parallel programs efficiently, and can be more efficiently calculated when compared against an existing descriptor BRS1, and reduces the amount of data transferred when compared against BRS2.

1. はじめに

High Performance Fortran (HPF)[5]，Split-C[7]，OpenMP[9] といった並列プログラミング言語では，プログラマは，データ授受の手段として共有メモリを想定してプログラミングすることが可能である．これらの言語を計算機クラスタなどの分散メモリ環境で実現するためには，何らかの機構を用いて，他のプロセッサで生産されたデータを，消費プロセッサで正しく読み出せるようにしなければならない．その機構としては，1) ハードウェアまたはソフトウェアで実装された分散共有メモリシステム，2) 依存関係のあるデータを通信するコードを生成するコンパイラがあげられる．1) では，遠隔データをアクセスする際のレイテンシ，およびコヒーレンシ管理にともなうオーバヘッドが性能を抑制する原因となる．また，多くの分散共有メモリシステムでは，コヒーレンシ管理の単位として，ページやキャッシュラインといった固定的な管理

† 神奈川大学理学部情報科学科
Department of Information and Computer Science, Faculty of Science, Kanagawa University
†† 筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻
Department of Computer Science, University of Tsukuba

単位を採用しているが，システムは，ネットワークを介して転送される管理単位に含まれるデータのすべてが利用されるのか，あるいは一部のみが利用されるのかといったことについて関知しないため，必ずしも最適なデータ転送が実現されているとはいえない．一方，2) では，コンパイル時に生成された通信コードにより，依存関係にあるデータだけが生産プロセッサから消費プロセッサに転送されるため，ネットワークに対する負荷が小さくなる．さらに，この転送を，消費プロセッサによる読み出し前までに完了させるようにすれば，プログラムの停止が発生しないという利点が生じる．また，データ転送を計算と重ね合わせることによって，データ転送にかかる時間を隠蔽できる可能性がある．

2) を実現するコンパイラは，以下の機能を持っていなければならないと考えられる：i) 共有データへアクセスしているプログラム中の箇所を特定すること，ii) 読み出し/書き込みアクセスを区別すること，iii) 共有データが配列の場合は，アクセスされる配列範囲を認識すること，である．さらに，iv) 通信すべきデータを特定するため，読み出しアクセスされるデータを表す集合と書き込みアクセスされるデータを表す集合の間で積集合を求める機能が必要である．これまで，このような集合をコンパイラ内部で表現するための記述子として，いくつかの配列範囲記述子（Array Section Descriptor）が提案されてきた[1),2),4),8),10)]．これらの配列範囲記述子では，データの依存関係を調査するため，配列範囲を表現する記述子間での積演算を用意している．また，演算コストや必要メモリ量を小さくするために，様々な改良案が提案されている．しかしながら，従来の配列範囲記述子では，プロセッサに対してブロック・サイクリック割当てされた配列に対するアクセス・パターンを効率良く表現することができない．ブロック・サイクリック割当てとは，データを分割し，プロセッサに割り当てるための方法の 1 つであり，良好な負荷分散を達成するために，並列プログラミングで多用される．実際，上であげた言語にもブロック・サイクリック割当てを指定するための記法が用意されている．これらの記法により，ある配列がブロック・サイクリック割当てされた場合，各プロセッサがアクセスする配列範囲は，一定の長さのアクセス領域と一定の長さの非アクセス領域を交互に含むパターンとなる傾向がある．

本論文では，並列プログラム実行時にアクセスされる配列範囲を効率良く表現することができる配列範囲記述子，quad を提案する．また，配列範囲記述子を利用して並列プログラムを実行した場合のデータ転送量，および記述子間演算コストを基に quad の有効性について論ずる．

以下，本論文は次のように構成される．2 章では quad について説明し，quad 間の演算を定義する．3 章では quad 間演算の実装方法について詳しく述べ，続く 4 章では quad の性能を評価する．5 章では関連研究について述べ，6 章でまとめとする．

2. 配列範囲記述子 quad

本章では，並列処理において使用される配列範囲記述子が満たすべき要件について述べたあと，我々の提案する quad の定義と quad 間の演算について述べる．また，並列プログラムへの quad の適用例を示す．

2.1 並列処理に適した配列範囲記述子

HPF，Split-C，OpenMP といった，共有メモリを提供する並列プログラミング言語を実現するために用いられる配列範囲記述子に対しては，

- 並列プログラムで頻出するアクセス・パターンを簡潔に表現できること，
- データの依存性を解析するための，記述子間での積演算が用意されていること，
- 記述子間演算のオーバヘッドを小さく抑えられるよう，演算コストおよび必要メモリ量が小さいこと，

が求められる．以下，本章では，我々の提案する quad が，どのようにこれらの要件を満たしているのかについて詳しく述べる．

2.2 quad の定義

quad は，4 個の整数 a，b，c，d から構成され，それぞれの整数は，

a: アクセスされる先頭領域の配列オフセット
b: アクセス領域の長さ
c: 非アクセス領域の長さ
d: 繰返し数

を表す．

たとえば，**図 1** (a) に示す OpenMP プログラムの

```
int a[n], c[n];
#pragma omp parallel for schedule(static, bs)
for (i = 0; i < bs * nproc * 5; i++)                /* (a) */
    a[i] = i * i;
#pragma omp parallel for schedule(static, bs)
for (i = 0; i < bs * nproc * 5; i++) {              /* (b) */
    if (i <= 0 || i >= n - 1) continue;
    c[i] = a[i-1] + a[i] + a[i+1];
}
```

図 1 OpenMP プログラムの例
Fig. 1 An example of OpenMP program.

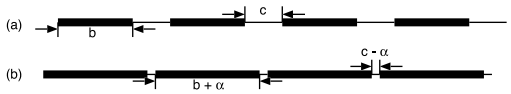

図 2 並列プログラムに典型的に出現するアクセス・パターン
Fig. 2 Typical access patterns observed in parallel programs.

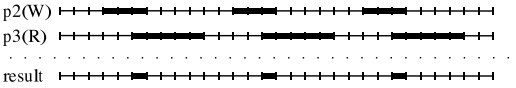

図 3 依存関係のあるデータを求める quad 間の積演算の例
Fig. 3 An example of intersection operation between quads which investigates data dependency.

実行時に，各プロセッサが書き込みアクセスする配列 a の領域は次の quad で表現することができる：
$(pid * bs, bs, bs * (nproc - 1), 5)$
（ここで，pid はプロセッサ ID，$nproc$ はプロセッサ数，bs はブロックサイズである）．

quad 自体の大きさは，quad が表現する配列範囲の大きさに関係なくつねに一定であり，その結果，quad を保持するために必要なメモリ量を小さく抑えることが可能である．また，quad 間の演算を実装する際には，演算コストが繰返し数 d に比例しないように実装することが可能である．実際，3 章で述べる実装方法では，どのようなパラメータの組合せに対しても演算コストがほぼ一定となる．

quad の大きな特長として，並列プログラムで頻出するアクセス・パターンを簡潔に表現することができる点があげられる．共有メモリ型言語では，良好な負荷分散を達成するため，データを分割し，プロセッサへ割り当てるための記法が用意されている．プログラマは，問題の性質に応じて，データの割当て方法として，ブロック割当て，サイクリック割当ておよびブロック・サイクリック割当てを指定することができる．これらのうち，ブロック・サイクリック割当てされたデータに対するアクセス・パターンは，**図 2** (a) に示されるように，一定の長さのアクセス領域と一定の長さの非アクセス領域を交互に含むものとなる．また，分割された空間の境界外の近傍を含めてアクセスする問題に見られるように，割り当てられた領域よりも少し広い領域をアクセスする場合でも，同様のストライドアクセスとなる（図 2 (b)）．従来の配列範囲記述子と異なり，quad では，アクセス領域と非アクセス領域を独立に指定すると同時にそれらの繰返し数を指定することができるため，共有メモリプログラムの実行中に発生する，ストライドを含むアクセス・パターンを簡潔に表現することができる．

ここでは，ブロック・サイクリック割当てのみに焦点を当てたが，ブロック割当てとサイクリック割当てはともにブロック・サイクリック割当ての特別な場合であるため，quad で統一的に扱うことができる．

quad と同様に，限られた個数のパラメータで配列範囲を表現する記述子に，Havlak らの提案する Bounded Regular Section（BRS）がある[4]．BRS では配列範囲を次のように表現する：
(l (下限), u (上限), s (ストライド))

BRS には，アクセス領域の長さが 2 以上の場合ストライドは 1 でなければならず，ストライドが 2 以上の場合アクセス領域の長さは 1 でなければならない，という制約がある．したがって，BRS で表現可能ないかなる配列範囲も quad で表現することができる．実際，BRS(l, u, s) は，quad$(l, 1, s-1, (u-l)/s+1)$ に変換することができる．ただし，s が 1 の場合，$(l, u-l+1, 0, 1)$ とすることもできる．

```
omp_set_num_threads(4);
#pragma omp parallel for
for (i = 0; i <100; i++)
    a[i] = i * i;
#pragma omp parallel for
for (i = 100; i <200; i++)
    a[i] = i * i + 1;
```

図 4 同一配列の排他的領域をアクセスする 2 個のループ
Fig. 4 Two loops that access exclusive parts of the same array.

2.3 quad 間演算

図 1 (b) のプログラムコードでは，各プロセッサは，割り当てられた領域よりもわずかに広い領域を読み出している．この場合，それぞれの読み出し領域の両端の配列要素は，他のプロセッサによって書き込まれるため，読み出しプロセッサに転送される必要がある．このような転送されるべきデータを求めるためには，読み出されるデータおよび書き込まれるデータを表現する quad を用意し，それらの quad の間で積演算を行えばよい．たとえば，プロセッサ 2 からプロセッサ 3 へ転送すべきデータは，$(3*bs-1, 1, bs*nproc-1, 5)$ であり，これは $(2*bs, bs, bs*(nproc-1), 5) \cap (3*bs-1, bs+2, bs*(nproc-1)-2, 5)$ の結果である．**図 3** に，$nproc = 3$，$bs = 3$ のときに，これらの quad で表現される配列範囲を示す．

quad 間の和演算は，いくつかの quad をより少ない数の quad にまとめるために用いられる．**図 4** に示すプログラムでは，最初のループで $(pid*25, 25, 0, 1)$ として，次のループで $(100+pid*25, 25, 0, 1)$ として表現される領域に対してデータが書き込まれている．これ

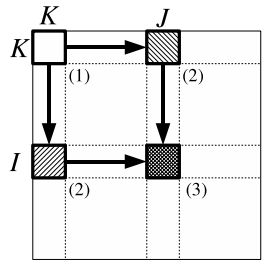

図 5 LU におけるデータの依存関係
Fig. 5 Dataflow in LU.

```
/* factor diagonal block */
if (pid == BlockOwner(B_KK)) {
  lu0(B_KK);
  register_quad(pid, Write, B_KK, P_i);          /* (a) */
}
barrier();
/* divide column k by diagonal block */
for I (K + 1 .. N)
  if (pid == BlockOwner(B_IK)) {
    register_quad(pid, Read, B_KK, P_i+1);       /* (b) */
    fetch(intersection(read_by_me, written_by_others));
                                                 /* ↑ (c) */
    bdiv(B_IK, B_KK);
    register_quad(pid, Write, B_IK, P_i+1);      /* (d) */
  }
```

図 6 LU における quad の適用例
Fig. 6 Using quad in LU.

ら 2 個の quad は，和演算によって $(pid*25, 25, 75, 2)$ とまとめることができる．

quad 間の積演算と和演算の実装方法については，次章で詳しく述べる．

2.4 並列プログラムへの quad の適用例

本節では，SPLASH-2 ベンチマーク[11] の LU を例にとり，quad 間演算によって依存関係にあるデータを求める様子を示す．

LU では行列をブロックに分割し，各プロセッサにブロック・サイクリックで割り当てる．**図 5** に示すように，LU の主ループの K 回目の繰返しでは，

(1) 左上のブロックを担当するプロセッサがブロックに対して部分的な LU 分解を実行する，
(2) このブロックを読み出して，第 K 行，および第 K 列の各ブロックに対して担当プロセッサが値を更新する，
(3) K 行，K 列のブロックを読み出して，残りのブロックに対して担当プロセッサが値を更新する，

という手順で計算する．ブロックを計算をするためには，前の手順における計算結果が必要となるため，手順間にバリア同期が挿入されている．

図 6 に，上記の手順 (1)，および (2) の第 K 列の計算に対応するコードを示す．手順 (1) では，1 個のブロックが担当プロセッサによって書き込まれるため，書き込んだプロセッサ ID（図 6 (a) の pid）と書き込みを表すタグ（図 6 (a) の Write）とアクセスの発生するフェーズ ID（図 6 (a) の P_i）とともに，ブロックの範囲を示す quad（図 6 (a) の B_{KK}）を生成して，メモリ領域に登録する register_quad 関数を呼び出す．書き込みに関するアクセス情報は，他プロセッサに対して同期時に通知される．

手順 (2) では，左上のブロックを読み出し，第 K 列の各ブロックを更新する．そのため，更新を担当するプロセッサは，左上のブロックへの読み出しに関するアクセス情報を生成し，メモリ領域に登録する (図 6 (b))．intersection 関数では，自プロセッサの読み出しに関するアクセス情報と他プロセッサによる書き込みに関するアクセス情報の間で積演算をすることにより，依存関係にあるデータを特定し，fetch 関数によってそのデータを取得する（図 6 (c)）．また，取得したデータを基に計算した結果を B_{IK} に書き込む際に，対応するアクセス情報を登録している（図 6 (d)）☆．

3. quad 間演算の実装

本章で述べる quad 間演算の実装では，オペランドとして 2 個の quad を受け取り，演算結果として quad を返す．また，各オペランド quad のすべてのパラメータは演算までに整数値として値が確定していることが要求される．すなわち，変数や式で表現される配列範囲は受け付けない．したがって，quad 間演算は，プログラムの実行時にのみ行うことができる．ただし，コンパイル時であってもすべてのオペランドが整数として確定している場合は演算可能である．

3.1 実装の基本方針

図 7 の A と B の積は，C である．この結果を過不足なく正確に表現するためには，4 個の quad，すなわち (0, 4, 0, 1)，(7, 3, 0, 1)，(14, 2, 8, 2)，(18, 1, 2, 2) が必要である．このように，quad には，演算を

☆ 4 章で評価するすべてのアプリケーションプログラムでは，読み出されるデータのすべては直前のフェーズで書き込まれるため，図に示したコードにおける積演算によって依存関係にあるデータを正しく求めることができる．並列プログラム実行環境の他の実装方法として，各データをそのアドレスによりホーム領域に割り当てて，読み出しプロセッサが最新のデータをホームから取得するという方法も考えられる．この場合も，quad などの配列範囲記述子を用いてホーム領域の割当て状況と読み出し範囲を表現しておいたうえで，記述子間での積演算により通信すべきデータを特定することができる．この場合，本文中で述べた方法とは異なり，読み出しプロセッサは，書き込みに関するアクセス情報をそのつど受け取る必要はない．

図 7 積演算の真の結果を 1 個の quad で表現できない例
Fig. 7 A case in which the result of intersection operation cannot be represented with a single quad.

施すたびに quad の個数が増えていく可能性がある，という特性がある．そこで，quad 間の演算の実装方針として，以下の 2 種類の方針が考えられる：

- 演算結果が表現するアクセス情報がつねに正確になることを優先し，それを表現するための quad の個数を制限しない．
- 演算結果を表現する quad の個数を 2 個以内に限定し，それらが表現するアクセス情報の正確性を犠牲にする．

我々は，実装が容易であることと必要メモリ量が少ないことから後者の方針を採択した（以下，オペランド quad を $q1$，$q2$，結果 quad を $q3$，$q4$ とする．また，$q1$ の 1 番目のパラメータを $q1.a$ のように書く）．後に，この方針を採用しても，プログラムの実行が非効率にならないことを評価する．演算結果を 2 個の quad で表現することができない場合は，妥当な結果を返すものとする．ここで，「演算結果が妥当である」とは，それらの演算結果を用いても実行結果の正当性に影響を与えないという意味である．このような妥当な結果を，以下では，準最適解と呼ぶ．また，演算結果がアクセス情報を過不足なく表現しているとき，その結果を最適解と呼ぶ．

我々の実装では，演算が最適解を得ることができないときは，以下のような準最適解を返す．

- 和演算は，何もせずにオペランドを準最適解として返す．
- 積演算は，オペランドの長さを求め，短い方を準最適解として返す．ここで，長さは，たとえば $q1$ の場合，$q1.b \times q1.d$ で求められる．ただし，一方の quad が書き込みアクセス情報を表現していることが分かっている場合，その quad を準最適解として返す．もし読み出しアクセス情報を表現する quad を準最適解としたならば，プログラムの記述上は書き込みアクセスの競合がなくても，複数の書き込みプロセッサから同じアドレスのデータが到着する現象，すなわち書き込みに関するレース状態が発生してしまう．本論文では，このような本来生じないはずの書き込みアクセスの競合が，配列範囲記述子間の演算によって引き起こされる現象を，擬似的な書き込みアクセス競合と呼ぶことにする．

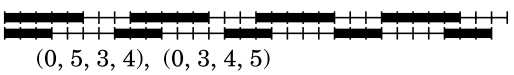

図 8 和演算の結果が準最適解となる場合（7 個の配列要素が両方の結果 quad に含まれる）
Fig. 8 A case in which two resulting quads include the same array elements.

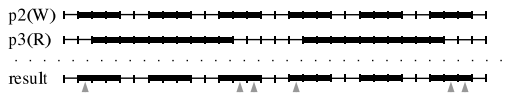

図 9 積演算の結果が準最適解となる場合（6 個の配列要素が冗長に転送される）
Fig. 9 A case in which a resulting quad includes the unnecessary array elements.

和演算における準最適解では，ある配列要素が両方の結果 quad に含まれてしまうことがあるため，このような準最適解を基にメッセージを転送すると，同じ配列要素が複数回転送されてしまう．**図 8** は，準最適解として得られた quad の例とそれらが表現する配列範囲を示している．ここでは 7 個の配列要素が両方の結果 quad に含まれる．一方，積演算における準最適解を基にメッセージを転送すると，読み出し対象とならない配列要素まで転送されてしまう．**図 9** では，積演算の結果が準最適解であったために，冗長に転送される 6 個の配列要素を示している．このように，準最適解は，冗長なメッセージ転送を引き起こすが，読み出し対象の配列要素は正しく転送されるため，プログラムの実行には影響を与えない．

3.2 和演算の実装

和演算では，2 個のオペランド quad のうちどちらかの 4 番目のパラメータ，すなわち繰返し数が 1 の場合，最適解が得られやすい．そのため，この条件を満たす場合を関数 union1 で，そうでない場合を関数 union2 で処理することとした．

3.2.1 関数 union1 の実装

和を求めるためには，オペランド quad 間の位置関係を調べて，どの程度の重なりが生じるか計算する必要がある．なお，以下の説明では，周期が 1 であるほうを $q1$ とする．

$q1$ のアクセス領域の先頭の要素が位置する，$q2$ 上の周期と位相は，それぞれ以下の式で求められる：

周期 $c_h = (q1.a - q2.a)/(q2.b + q2.c)$

位相 $p_h = (q1.a - q2.a) \bmod (q2.b + q2.c)$

また末尾の要素の周期 c_t と位相 p_t も同様に求められる．$q1$ と $q2$ がそれぞれ $(7, 18, 0, 1)$ と $(1, 3, 2, 6)$ であるときの c_h，p_h，c_t，p_t を**図 10** に示す．

関数 union1 のアルゴリズムを**図 11** に示す．まず，

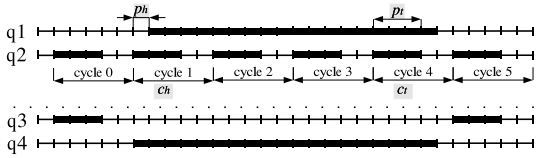

図 10 関数 union1 の例
Fig. 10 An example of union1.

```
union1(q1, q2)
{
    n_li = l_iso(c_h, p_h);
    n_ri = r_iso(c_t, p_t);
    if (n_li == 1 and n_ri == 1) {
        q3 = generate_quad(two_isolated_quads);
        q4 = generate_quad(overlapped_quads);
    }
    else if (n_li > 1 and n_ri == 0
          or n_ri > 1 and n_li == 0) {
        q3 = isolated_quads;
        q4 = generate_quad(overlapped_quads);
    }
    else if (n_li > 1 and n_ri > 1)
        (q3, q4) = (q1, q2);
    else
        q3 = generate_quad(overlapped_quads);
    return (q3, q4);
}
```

図 11 繰返し数が 1 の場合の和演算アルゴリズム
Fig. 11 An algorithm of union operation for the case of $q.d == 1$.

関数 l_iso および関数 r_iso で，$q2$ のアクセス領域のうち，$q1$ との重なりがないものの数を，それぞれ $q1$ の左側と右側で求め，n_{li} と n_{ri} に代入する．もし，n_{li} と n_{ri} がともに 1 の場合（図 10 の例は，この場合に該当する）は，両端の重なりのない領域どうしで $q3$ が生成される．さらに，重なる領域から $q4$ を求めるため，*overlapped_region* を引数とする関数 generate_quad が呼び出される．このとき，関数 generate_quad では，まず，重なる領域の最左の配列要素の添字が p_h と $q2.b$ の大小関係から決定され，その結果 $q4.a$ が求められる．同様に，最右要素も求められる．これら 2 個の要素と $c_t - c_h$ より，$q4.b$ が求められる．ここでは，$q4.d$ は 1 となり，$q4.c$ は意味を持たないため 0 とする．

次に，重なりがない領域が左あるいは右だけに偏っている場合は，$q2$ を適当に短くしたものが $q3$ となり，$q4$ は最初の場合と同様に求められる．両端に重なりのない領域が 2 個以上生じている場合は，最適解が求められない．

最後に，重なりのない領域が生じない場合は，関数 generate_quad によって $q3$ が生成される．

```
union2(q1, q2)
{
    if (c_{2_{h1}} == c_{2_{td}} and p_{2_{td}} < q1.b)
        return q1;
    if ((q1.b + q1.c) mod (q2.b + q2.c))
        return (q1, q2);
    (q3, q4) = generate_quads_similarly_in_union1;
    return (q3, q4);
}
```

図 12 繰返し数が 2 以上の場合の和演算アルゴリズム
Fig. 12 An algorithm of union operation for the case of $q.d > 1$.

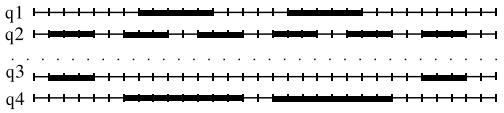

図 13 長い方の周期が短い方の周期の倍数となっている場合
Fig. 13 A case in which a longer cycle is a multiple of a shorter one.

3.2.2 関数 union2 の実装

関数 union2 は，オペランド quad の繰返し数がともに 2 以上である場合を扱う．周期が長い quad を $q1$，短い quad を $q2$ とする（周期が同じ場合，どちらの quad を $q1$ としてもよい）．ここでは，関数 union1 における c_h や p_h と同様のものとして，$q1$ の第 1 アクセス領域の先頭要素および末尾要素の，$q2$ 上の周期と位相 $(c_{1_{h1}}, p_{1_{h1}})$，$(c_{1_{t1}}, p_{1_{t1}})$ を求める．また，第 d アクセス領域についても求める（$(c_{1_{hd}}, p_{1_{hd}})$，$(c_{1_{td}}, p_{1_{td}})$）．さらに，$q2$ についても，同様に $q1$ 上の周期と位相を求める（それぞれ，$(c_{2_{h1}}, p_{2_{h1}})$，$(c_{2_{t1}}, p_{2_{t1}})$，$(c_{2_{hd}}, p_{2_{hd}})$，$(c_{2_{td}}, p_{2_{td}})$）．

関数 union2 のアルゴリズムを**図 12** に示す．まず，$q2$ 全体が，$q1$ のアクセス領域に含まれるという特別の場合について演算する．これは，$q2$ の第 1 アクセス領域の先頭要素の $q1$ における周期 $c_{2_{h1}}$ と，$q2$ の第 d アクセス領域の末尾要素の $q1$ における周期 $c_{2_{td}}$ が等しく，末尾要素の位相が $q1.b$ よりも短い場合である．このとき，関数 union2 は，$q1$ の内容を結果として返す．

次に，長い周期が短い周期の倍数になっているかどうか調べる．倍数になっていない場合は，真の結果が 2 個以内の quad で表現することができないことが多いため，準最適解を返す．

倍数となっている場合のオペランドは，**図 13** のようなオペランドであり，関数 union1 と同様の方法で結果を生成することができる．

3.3 積演算の実装

quad 間の積演算は，和演算と同様の方法で実装す

ることができるため，詳細は省略する．積演算では，n_{li} と n_{ri} の結果が正数の場合，すなわち重なりが生じない領域が存在する場合には，単にその領域を破棄すればよいので，和演算と比較してより簡潔に実装することができる．ただし，長い方の周期が短い周期の倍数となっていても，図 9 のような場合は，結果を 2 個以内の quad では表現することができないため，準最適解とする必要がある．

3.4 多次元配列への拡張

多次元配列へのアクセスを quad で表現するためには，1 次元に 1 個の quad を割り当てればよい．これを多次元 quad と呼ぶ．多次元 quad 間の積演算は，次元ごとに積演算を行い，その結果の集合が，演算結果となる．もしある次元の結果が空となれば，最終的な結果も空とする．一方，多次元 quad 間の和演算の我々の実装では，オペランド quad に制約条件を課している．すなわち，n 次元の quad の和演算のオペランド間で，対応する n 組の quad のうちの $n-1$ 組の quad はまったく同一でなければならない．残りの 1 次元を表現する quad 間で和演算を行い，その結果と $n-1$ 個の quad を合わせて，n 次元 quad 間の和演算の結果とする．もしこの制約条件を満たさない場合は，和演算は何もしない．

3.5 quad 列間の演算

プログラムの実行中のあるフェーズにおいて，あるプロセッサによってアクセスされる配列範囲は，1 個の quad だけでは正確に表現できないことが多い．そこで，配列範囲情報を複数の quad で表現しておき，メモリ領域に蓄積しておくという方法が考えられる．このような蓄積された quad を quad 列と呼び，後に他のプロセッサによるアクセス情報を表現する quad 列との間で積演算を行う対象となる．このような quad 列間の演算は，quad 列を構成する個々の quad 間の演算に還元し，それらの演算結果を集約することによって，実装することができる．

4. 性 能 評 価

quad の有効性を検証するために，3 種類の共有メモリプログラムを実行し，実行時に発生するデータ転送量，および記述子間の演算コストを測定した．

4.1 共有メモリプログラムの実行方式

一般に，共有メモリプログラムは，バリア同期で区切られるいくつかのフェーズから構成される．あるフェーズであるプロセッサが書き込んだデータを，続くフェーズで別のプロセッサが読み出す，というシナリオは共有メモリプログラムの実行時の典型的な振舞いである．本論文の実験では，2.4 節に例示した方法で，メモリアクセスに対応した quad を quad テーブルと呼ばれるメモリ領域に登録する関数をコード中に手動で挿入した．quad テーブルには，quad のほか，配列 ID，アクセスの種類（書き込み/読み出し），プロセッサ ID，フェーズ ID が記載される．

実行時のあるフェーズにおいて転送すべきデータを求めるためには，そのフェーズの読み出しプロセッサのアクセス情報を表現する quad 列と，そのフェーズ以前の他のプロセッサによる書き込みアクセス情報を表現する quad 列を quad テーブルから取り出し，それらの間で積演算を行う．本論文の評価では，この演算結果として生成される quad の個数（生成記述子数）を求め，その quad が表現する配列範囲の大きさ（配列要素の数）をデータ転送量として測定した．なお，積演算の終了後は，結果として生成された quad 数が少なくなるように，quad 列に対して和演算を施している．

この評価では，共有メモリプログラムを単一プロセッサマシン上で擬似並列的に実行している．具体的には，各フェーズをループで囲み，pid を 0 から $nproc-1$ まで変化させている．このような実行方法を採用しても，フェーズごとのアクセス情報を収集することができるため，生成記述子数とデータ転送量を求めることに関しては，実際の分散メモリ環境で実行した場合と等価な結果が得られることが保証される．

比較のために，アクセス情報を BRS で表現した共有メモリプログラムの実行結果も示す．Havlak らは BRS 間の演算を定義している[4] が，彼らの実装方法ではつねに単一の BRS を生成するため，非アクセス領域を挟む領域間の和演算の結果は，実際にアクセスしていない領域もアクセスしたものと表現してしまう．そのため，3 章で述べた，擬似的な書き込みアクセス競合が生じてしまう．これを避けるために，本論文の評価では，BRS の使用法に関して 2 つの方針を採用した：

BRS1： 正確さを重視する方針である．和演算の結果を 1 個の BRS で表現しようとしたときに非アクセス領域を含む場合は，和演算は何もしない．この方針では，BRS の数が大きくなる傾向がある．

BRS2： コストを重視する方針である．読み出しアクセスに関する和演算は，つねに 1 個の BRS を返す．この BRS によって表現される配列範囲は，実際には転送する必要のない要素を含んでいる可能があるが，読み出し側は単にそのような要素を読まずに済ますためにプログラムの実行の正しさ

には影響を与えない．書き込みアクセスに関しては，擬似的な書き込みアクセス競合を避けるためBRS1 と同様に扱う．

4.2 アプリケーション・プログラム

評価に用いたアプリケーション・プログラムは，LU [12]，FFT [12]，N 体問題である．このうち，LU と FFT は，SPLASH-2 ベンチマーク[11] のものを利用している．

SPLASH-2 LU：SPLASH-2 に含まれる 2 種類の LU のうち，本評価では，“contiguous blocks” を利用している．このバージョンでは，演算対象の行列の各要素のアドレスが，部分行列内で，すなわちブロック内で連続となる．ブロックは，ブロック・サイクリックでプロセッサに割り当てられるため，各プロセッサは，ブロック内では連続でアクセスし，ブロック間はアクセスしないというアクセス・パターンを示す．本評価では，行列サイズは $2,048 \times 2,048$，ブロックサイズは 64×64 である．

SPLASH-2 FFT：1 次元ベクタにおいて FFT を行う．その間，行列転置を 2 回実行する．転置操作は，各プロセッサが他プロセッサに割り当てられた要素の一部を読み出し，自プロセッサの割当て領域に書き込むことで達成される．このうち，読み出しアクセスは，一定の長さの非アクセス領域を複数含む領域へのアクセスとなる．本評価では，配列サイズは 2^{12} である．

N 体問題：1 次元空間内での物体の移動をシミュレートする．あるタイムステップにおける物体の位置は，前のステップにおける周辺の物体の集積度合で計算される．すなわち，物体が狭い領域に集まっている場合はより広い領域へ移動する．空間は部分空間に分割され，各プロセッサはそれぞれの部分空間に存在する物体について計算する．物体がプロセッサに平均して均等に割り当てられるように，部分空間はブロック・サイクリックで割り当てられる．本評価では，空間サイズは 4,096，ブロックサイズは 8，物体の数は 256 とした．

4.3 実 行 結 果

表 1，**表 2**，**表 3** は，それぞれのアプリケーション・プログラムを 4 プロセッサと 16 プロセッサで実行した結果を示している．表の上半分が 4 プロセッサの，下半分が 16 プロセッサの結果である．また，各表では，生成記述子数，データ転送量，結果的に消費されなかったデータをメッセージが含む割合，1 フェーズあたりに生成された記述子数を示している．

4.3.1 データ転送量

表には，アプリケーション・プログラムの実行中に発生した，すべての生成記述子数とデータ転送量が示されている．

quad のデータ転送量は，すべてのプログラムにおいて BRS1 と同一となった．BRS1 はデータ転送量に関しては最適であることを考えると，quad は，BRS1 より少ない生成記述子数で最適なデータ転送量を達成することができたといえる．BRS2 では，データ転送量が非常に大きくなっているが，これは，個々の BRS に冗長なデータが含まれてしまっているためである．この点に関して，次項で詳しく述べる．

表 1 LU の実行結果
Table 1 Results of LU.

	4 プロセッサ		
	BRS1	BRS2	quad
生成記述子数	1,054	17,614	184
転送量 (バイト)	34,537,472	577,175,552	34,537,472
不要データ率	1.00	16.71	1.00
平均記述子数	5.73	36.93	1.00
	16 プロセッサ		
	BRS1	BRS2	quad
生成記述子数	3,148	48,868	880
転送量 (バイト)	103,153,664	1,601,306,624	103,153,664
不要データ率	1.00	15.52	1.00
平均記述子数	3.58	10.16	1.00

表 2 FFT の実行結果
Table 2 Results of FFT.

	4 プロセッサ		
	BRS1	BRS2	quad
生成記述子数	336	15	17
転送量 (バイト)	86,016	371,456	86,016
不要データ率	1.00	4.32	1.00
平均記述子数	22.40	1.00	1.13
	16 プロセッサ		
	BRS1	BRS2	quad
生成記述子数	1,860	255	269
転送量 (バイト)	119,040	1,932,224	119,040
不要データ率	1.00	16.23	1.00
平均記述子数	7.29	1.00	1.05

表 3 N 体問題の実行結果
Table 3 Results of N-body.

	4 プロセッサ		
	BRS1	BRS2	quad
生成記述子数	480	2,224	236
転送量 (バイト)	3,840	17,984	3,840
不要データ率	1.00	4.68	1.00
平均記述子数	19.20	61.78	9.44
	16 プロセッサ		
	BRS1	BRS2	quad
生成記述子数	480	10,773	248
転送量 (バイト)	3,840	87,144	3,840
不要データ率	1.00	22.69	1.00
平均記述子数	5.71	14.96	2.95

4.3.2 不要データ率

前述のように，quad と BRS は，限られた個数の整数の組合わせによって配列範囲を表現する．そのため，配列範囲を必ずしも正確に表現することができるわけではない．quad や BRS が表現する配列範囲のうち，転送されたものの使用されることがなかった配列要素の数を表に示す．これは，不要データをいっさい含まない BRS1 を 1 としたときの相対量である．前項で述べたように，quad の結果は BRS1 と同一となった．

BRS2 は FFT において生成記述子数が最小となっているものの，不要データ率は 4.32 および 16.23 と，不要データが多く含まれていることが分かる．一方で，LU と N 体問題においては，BRS2 は quad と比較して 4.68 倍から 95.73 倍の莫大な数の記述子を発生させている．この理由としては，FFT ではすべてのプロセッサが他のすべてのプロセッサと通信する必要があり，転送されるメッセージはつねに必要なデータを含んでいる（不要なデータも多く含んでいる）のに対して，LU と N 体問題では，本来，生産–消費関係にないプロセッサ間でもメッセージ（必要なデータをいっさい含んでいない）が発生してしまうことが考えられる．

4.3.3 同一配列要素の転送回数

3 章で述べたように，quad 間の和演算の結果，ある配列要素が複数の quad に含まれてしまう可能性がある．このような場合，配列要素が複数回送られることになり，ネットワークのバンド幅を浪費する事態を招く．

本評価では，1 組の生産–消費関係において転送された，すべての配列要素の転送回数を調査した．その結果，転送されたすべての配列要素が，3 種類のアプリケーション・プログラムにおいて 1 回のみ転送されたことが判明した．これは，すべての quad 間，および BRS 間の演算で，準最適解が発生しなかったことを意味している．

4.3.4 1 フェーズあたりに生成された記述子数

表に，1 フェーズあたりに生成された平均記述子数を示す．平均生成記述子数が小さいということは，ある生産–消費関係において依存性のあるデータをより少ない記述子で表現する能力，すなわち情報圧縮能力が高いことを意味する．

BRS1 の結果が quad の 1.94 倍から 19.82 倍という結果となったが，これは，BRS1 が記述子数を犠牲にするポリシだからである．FFT において quad は，BRS2 と比較して若干多くの生成記述子数を発生させてしまっている．しかし，BRS2 は多くの不要な配列要素を転送していることを考え合わせると，quad は，少ない記述子数で効率良く配列範囲を表現することができる配列範囲記述子であることが分かる．

表 4 和演算の回数と総サイクル数
Table 4 Total cycles and calls to union operations.

	4 プロセッサ		
LU	BRS1	BRS2	quad
サイクル数 (10^6)	1,524.89	757.47	14.82
呼び出し回数	7,310,344	3,622,269	49,284
平均サイクル数 (10^3)	0.21	0.21	0.30
FFT	BRS1	BRS2	quad
サイクル数 (10^6)	6.33	0.09	0.04
呼び出し回数	31,896	376	10
平均サイクル数 (10^3)	0.20	0.24	3.66
N-body	BRS1	BRS2	quad
サイクル数 (10^6)	75.53	30.17	0.91
呼び出し回数	382,769	152,838	2,044
平均サイクル数 (10^3)	0.20	0.20	0.45
	16 プロセッサ		
LU	BRS1	BRS2	quad
サイクル数 (10^6)	1,722.58	753.17	32.39
呼び出し回数	8,225,171	3,558,076	71,852
平均サイクル数 (10^3)	0.21	0.21	0.45
FFT	BRS1	BRS2	quad
サイクル数 (10^6)	172.76	0.48	0.32
呼び出し回数	877,590	1,888	238
平均サイクル数 (10^3)	0.20	0.26	1.36
N-body	BRS1	BRS2	quad
サイクル数 (10^6)	88.13	34.12	1.02
呼び出し回数	447,830	171,081	2,230
平均サイクル数 (10^3)	0.20	0.20	0.46

4.3.5 積演算の削減効果

表 5 の「呼び出し回数」に，プログラム実行中に行われた積演算の総数を示す．演算対象の 2 本の quad 列あるいは BRS 列の長さをそれぞれ m，n とすると，積演算の回数は $O(mn)$ で増加する．積演算の前には，和演算によって quad 列や BRS 列をなるべく短かくするため，積演算の回数が少ないことは，和演算が広い配列範囲を少ない記述子で表現できることができることを意味する．

BRS2 は quad と比較して，FFT では 0.53 倍ないし 0.65 倍と少なかったものの，LU と N 体問題では 4.3.2 項で述べた理由から，大小関係が逆の結果が得られた．また，BRS1 は quad の 21.42 倍から 254.64 倍とつねに大きくなった．これらのことから，記述子数を犠牲する BRS1 はもとより，非アクセス領域を表現できない BRS2 もアプリケーションによっては積演算回数が多くなるのに対して，quad は記述子数を抑えたまま正確なアクセス領域を表現できることが示された．

4.3.6 演算コスト

最後に，記述子間の和演算と積演算に要したサイ

表 5 積演算の回数と総サイクル数
Table 5 Total cycles and calls to intersection operations.

	4 プロセッサ		
LU	BRS1	BRS2	quad
サイクル数 (10^6)	43.67	5.14	5.18
呼び出し回数	782,397	26,281	7,944
平均サイクル数 (10^3)	0.06	0.20	0.65
FFT	BRS1	BRS2	quad
サイクル数 (10^6)	0.13	0.03	0.06
呼び出し回数	720	15	23
平均サイクル数 (10^3)	0.19	2.33	2.57
N-body	BRS1	BRS2	quad
サイクル数 (10^6)	15.54	0.55	1.06
呼び出し回数	287,232	2,244	1,128
平均サイクル数 (10^3)	0.05	0.25	0.94
	16 プロセッサ		
LU	BRS1	BRS2	quad
サイクル数 (10^6)	62.82	15.09	27.29
呼び出し回数	1,097,682	82,792	51,236
平均サイクル数 (10^3)	0.06	0.18	0.53
FFT	BRS1	BRS2	quad
サイクル数 (10^6)	1.27	0.14	0.39
呼び出し回数	15,300	255	479
平均サイクル数 (10^3)	0.08	0.55	0.82
N-body	BRS1	BRS2	quad
サイクル数 (10^6)	19.51	2.69	2.99
呼び出し回数	359,040	11,220	5,700
平均サイクル数 (10^3)	0.05	0.24	0.52

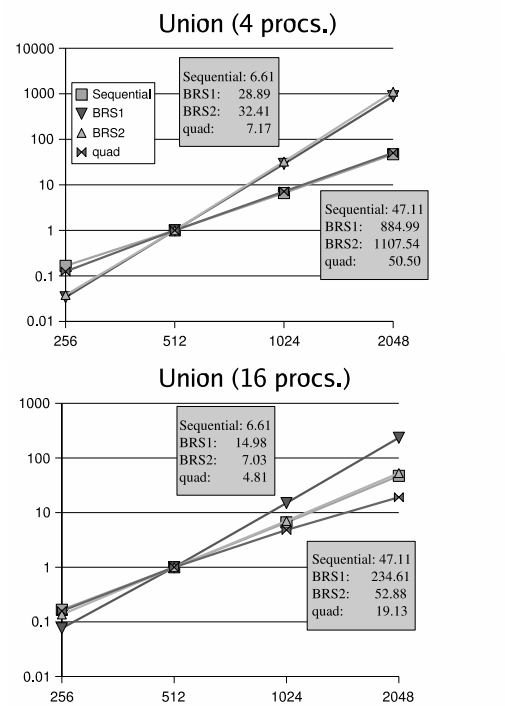

図 14 LU における問題サイズと和演算コストの関係
Fig. 14 The plots of the computational cost of the union operations between descriptors in increasing the size of LU.

クル数について評価する．演算の前後に，Intel 社の Pentium シリーズおよび互換プロセッサに用意されている RDTSC（Read Time Stamp Counter）命令を挿入し，演算に要したサイクル数を計測した．この実験では，アプリケーション・プログラムを-O2 オプションを指定した gcc 3.3.2 でコンパイルし，Linux 2.4.22 が稼働する，AMD AthlonXP 2600+（クロック周波数 1.91 GHz）および 512 M バイトの主メモリを搭載した PC で実行した．

表 4 と表 5 に，それぞれ和演算と積演算に要した総サイクル数，総演算数，1 演算あたりの平均サイクル数を示す．

和演算では，BRS1 と BRS2 の 1 演算あたりの平均サイクル数は，quad よりも少ない結果となった．これは，BRS は quad に比べて実装が簡潔であることが理由として考えられる．しかしながら，BRS1 と BRS2 は quad よりもかなり多くの回数の演算を必要としたため，quad の全体の演算コストすなわち総サイクル数は，すべての場合で BRS よりも少ない結果となった．特に，BRS1 は，最大で quad の 539.33 倍の演算サイクルを要した．

一方，積演算の全体の演算コストは，どのアプリケーションプログラムでも BRS2，quad，BRS1 の順で大きくなった．quad と比較すると，BRS1 は，最大で 14.66 倍の演算サイクルを消費する結果となった．

なお，quad の平均サイクル数が大きいものがいくつかあるが，この現象は，いずれも演算数が少ないときに発生している．この原因としては，命令キャッシュミスが考えられる．

図 14 および **図 15** は，LU において，問題サイズを変化させたときの，各記述子間での，和演算および積演算にかかる時間の変化をそれぞれ示している．また，“Sequential” は，LU を単一プロセッサで実行したときの演算時間，すなわちアプリケーションの総実行時間を表している．各グラフは，両軸ともに対数目盛となっており，問題サイズが 512 のときの結果を 1 としたときの相対値をプロットした．

各グラフともおおむね直線となっており，問題サイズをさらに 2,048 よりも大きくした場合，直線をそのまま延長した値が得られることが予想される．quad の演算コストは，逐次時間の変化に沿って変化する傾向があり，問題サイズが大きくなっても，アプリケーション演算時間に対して，つねに一定の大きさを占めるにすぎないと考えられる．一方，BRS1 の傾きは逐次時間よりもつねに大きいため，問題サイズが大きく

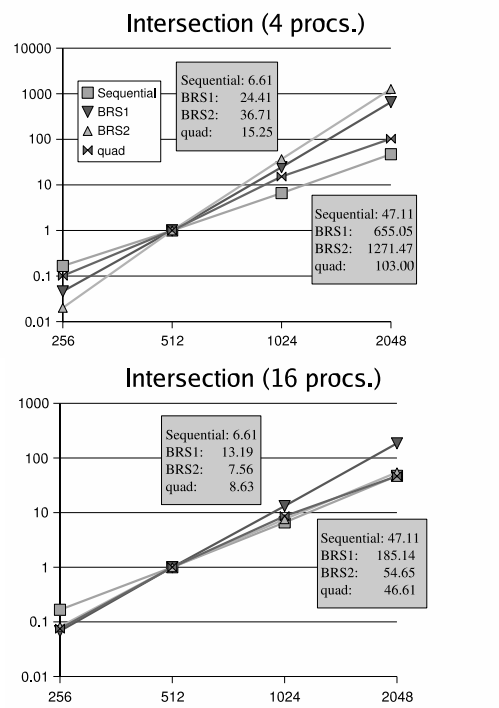

図 15 LU における問題サイズと積演算コストの関係
Fig. 15 The plots of the computational cost of the intersection operations between descriptors in increasing the size of LU.

なったときに，BRS 間演算コストが無視できない大きさになる可能性が高いことが分かる．

5. 関 連 研 究

プログラム中の手続き，あるいはコードセグメントの中でアクセスされる配列の範囲を特定し，表現するための記述子は，主に並列化コンパイラの分野において研究されてきた．並列化コンパイラは，2 つの手続きが排他的な領域をアクセスすると認識すれば，それらの手続きは並列に実行することができると判断する．この際，アクセスする領域を配列範囲記述子で表現する．

これまで，様々な種類の配列範囲記述子が提案されてきたが，これらは 2 つのグループ，すなわち列挙法[2),8)] とサマリ法[1),4),6),10)] に分類することができる．列挙法は，配列要素をすべて列挙する方法であるため，どのようなアクセス・パターンをも表現することが可能であるが，その反面，記述子の大きさが表現しようとする配列要素の数に比例して大きくなってしまうという欠点がある．一方，サマリ法は，表現しようとする配列範囲を要約して表現するため，メモリコスト，記述子間の演算コストをともに抑えることが可能である．我々の quad は，サマリ法に分類される．本論文の評価では，サマリ法を用いても，実際にはアクセス情報の損失は生じず，プログラム実行が非効率になってしまうことはなかった．

サマリ法に分類されるいくつかの記述子[1),10)] は，表現対象の配列範囲を（2 次元）空間に対応させて，その配列範囲を囲む境界直線の方程式の集合として表現する．この方式は，複雑な多角形を表現することができるが，独立した領域を複数含むようなアクセス・パターンを効率良く表現することはできない．一方，我々の提案する quad は，複雑な形の配列範囲を表現することはできないが，本論文で述べたような，並列プログラムにおいて典型的に現れるアクセス・パターンを効率良く表現することができる．

BRS[4)] は，非アクセス領域を表現するためにストライドの概念を導入したが，アクセス領域と非アクセス領域の長さが同時に 2 以上であるようなアクセス・パターンを効率良く表現することはできない．一方，quad はこのようなアクセス・パターンを簡潔に表現することができる．

LMAD（Linear Memory Access Descriptor）[6)] もストライドを導入しているが，ストライドをともなう周期的なアクセス・パターンを表現する場合は，quad の方がより少ないパラメータで表現することができる．

配列範囲記述子を共有メモリプログラムの実行環境に採用した例として，TreadMarks[3)] があげられる．TreadMarks のコンパイラは，BRS を用いてデータ依存性を解析している．TreadMarks はページを共有単位とする分散共有メモリシステムであるため，quad が扱う配列要素という粒度はページを表現するには細かすぎる可能性がある．quad は，配列要素単位で共有データの一貫性を管理するオブジェクトベースの分散共有メモリシステムに適していると考えられる．

6. お わ り に

本論文では，quad という配列範囲記述子を新たに提案し，quad 間演算の実装について述べた．quad は，4 個の整数を用いて，並列プログラムの実行中に出現するアクセス・パターン，すなわち一定の長さのアクセス領域がある間隔をおいて繰り返されるというパターンを効率良く表現することができる．また，quad 間演算のコストおよび quad を保持するためのメモリのコストは，quad が表現するメモリ領域の大きさには依存しない，という特徴がある．

quad を評価するため，3 種類のアプリケーション・プログラムを quad と従来の配列範囲記述子を用いて

実行し，実行中に転送されるべきデータを表現する記述子の個数と，その記述子に基づいて転送されるデータ量を測定した．実験結果から，quad は様々なアクセス・パターンを効率良く表現することができ，従来の配列範囲記述子 BRS2 に比べてネットワークへの負荷を大幅に軽減できることを述べた．また，従来の配列範囲記述子 BRS1 の演算コストは問題サイズの増加にともない無視できない大きさになる可能性があるのに対して，quad 間の演算コストは問題サイズの増加と同調して増加することから，quad は，従来法 BRS1 に対しては演算コストの面で優位であることが示せた．

今後の課題として，次の 2 点があげられる：

- 様々な種類のアプリケーション・プログラムを用いて評価すること
- quad を用いて，分散メモリ環境向けの OpenMP コンパイラを開発すること

我々は，現在，MPI プログラムを生成する OpenMP コンパイラを開発している．コンパイラは，OpenMP プログラム中のメモリアクセスを解析し，必要なデータが適切に読み出しプロセッサへ転送されるように，quad 間演算とメッセージパッシングコードを生成する．生成されたコードは，通常の `mpicc` でコンパイルされ，PC クラスタで実行することができる．

OpenMP コンパイラは，手続き間解析とシンボリック quad の操作を行うように実装する予定である．シンボリック quad とは，それぞれのパラメータに，変数やそれらで構成される式を許す quad である．たとえば，ある手続きがループの本体で呼び出される場合，まず，手続きにおけるアクセス情報を表現するシンボリック quad が生成され，そのシンボリック quad を基に，ループ本体のアクセス情報を表現するシンボリック quad が生成される．

謝辞 本研究の一部は，科学研究費補助金基盤研究 (B)，課題番号 16300012 の助成を受けて実施された．

参 考 文 献

1) Balasundaram, V.: A Mechanism for Keeping Useful Internal Information in Parallel Programming Tools: The Data Access Descriptor, *Journal of Parallel and Distributed Computing*, Vol.9, No.2, pp.154–170 (1990).
2) Burke, M. and Cytron, R.: Interprocedural Dependence Analysis and Parallelization, *SIGPLAN Notices*, Vol.21, No.7, pp.162–175 (1986).
3) Dwarkadas, S., Lu, H., Cox, A.L., Rajamony, R. and Zwaenepoel, W.: Combining Compile-Time and Run-Time Support for Efficient Software Distributed Shared Memory, *Proc. IEEE, Special Issue on Distributed Shared Memory*, Vol.87, No.3, pp.476–486 (1999).
4) Havlak, P. and Kennedy, K.: An implementation of interprocedural bounded regular section analysis, *IEEE Trans. Parallel and Distributed Systems*, Vol.2, No.3, pp.350–360 (1991).
5) High Performance Fortran Forum: High Performance Fortran Language Specification Version 2.0 (1997).
6) Hoeflinger, J.P.: Interprocedural Parallelization Using Memory Classification Analysis, Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois (1998).
7) Krishnamurthy, A., Culler, D.E., Dusseau, A., Goldstein, S.C., Lumetta, S., von Eicken, T. and Yelick, K.: Parallel programming in Split-C, *Proc. 1993 ACM/IEEE conference on Supercomputing*, pp.262–273 (1993).
8) Li, Z. and Yew, P.-C.: Efficient Interprocedural Analysis for Program Parallelization and Restructuring, *Proc. ACM/SIGPLAN PPEALS 1988 Parallel Programming: Experience with Applications, Languages and Systems*, July 1988, SIGPLAN Notices, pp.85–99 (1988).
9) OpenMP Architecture Review Board: OpenMP C and C++ Application Program Interface, version 2.0 edition (2002).
10) Triolet, R., Irigoin, F. and Feautrier, P.: Direct Parallelization of Call Statements, *Proc. ACM SIGPLAN 1986 Symposium on Compiler Construction*, pp.176–185 (1986).
11) Woo, S.C., Ohara, M., Torrie, E., Singh, J.P. and Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proc. 22nd Annual International Symposium on Computer Architecture*, pp.24–37 (1995).
12) Woo, S.C., Singh, J.P. and Hennessy, J.L.: The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors, *ACM SIGPLAN Notices*, Vol.29, No.11, pp.219–229 (1994).

(平成 16 年 1 月 22 日受付)
(平成 17 年 3 月 1 日採録)

米澤　直記（正会員）

昭和 47 年生．平成 11 年筑波大学大学院博士課程工学研究科退学．同年筑波大学電子・情報工学系助手を経て，平成 15 年より神奈川大学理学部情報科学科助手．分散共有メモリシステム，並列プログラミング環境に関する研究に従事．

和田　耕一（正会員）

昭和 31 年生．昭和 59 年神戸大学大学院自然科学研究科博士課程修了．同年神戸大学大学院自然科学研究科助手．昭和 62 年筑波大学電子・情報工学系講師．助教授を経て平成 11 年教授，現在に至る．平成 4〜5 年カナダ，ビクトリア大学客員研究員．並列分散処理とアーキテクチャ，並列シミュレーション，マルチメディア情報処理に関する研究に従事．学術博士．IEEE，ACM 各会員．