ハイパフォーマンス
コンピューティング 48－4
（１９９３．８．１９）

Dataparallel-C を用いたベクトル化コード生成とAP1000への実装

小柳 洋一　堀江 健志*
(株)富士通研究所　並列処理研究センター†

Dataparallel-C (以下DPC)は SPMD ベースのデータパラレル処理指向並列言語である．C言語に仮想プロセッサの概念を導入し，陽に並列性を内在したプログラムの記述を可能とする．近年の高性能ベクトル演算プロセッサチップを各ノードに搭載した並列計算機においては，DPCの仮想プロセッサの特徴を利用して，並列化に加えベクトル化によってさらにデータパラレルモデルを高速に実行できる可能性がある．本稿では，DPC コンパイラを並列化とベクトル化の両方を行なえるコンパイラに拡張し，DPCプログラムからベクトル化コードを複雑な依存解析なしに生成できること示す．また，高並列計算機AP1000用コード生成を実際に行ない評価する．

Vector Code Generation in Dataparallel-C on the AP1000

Yoichi Koyanagi, Takeshi Horie
Parallel Computing Research Center, FUJITSU LABORATORIES LTD.

Dataparallel-C (DPC) is a parallel language based on SPMD model. DPC looks like C language and has the notion of virtual processors to express the explicit parallelism. Recently some parallel computer systems have special vector processing chips in each processing element. With this feature of DPC we can generate parallelized and vectorized codes for these systems. In this paper we show the advanced DPC compiler that can generate vector codes not using complex data dependency analysis. We also evaluate the codes for the AP1000 generated by this compiler.

*E-mail:{ykoya,lions}@flab.fujitsu.co.jp
†211 川崎市中原区上小田中1015

1 はじめに

Dataparallel-C[1] (以下DPC)は SPMD ベースのデータパラレル処理指向並列言語である．C言語に仮想プロセッサの概念を導入し，陽に並列性を内在したプログラムの高水準な記述を可能とする．DPCはいくつかの並列計算機上に実装されており，実用的なプログラムが並列計算機の上で簡潔に記述，実行できるようになっている．このDPCの並列性は，現在まではMIMD マシン，共有メモリマシンなどの並列計算機によって引き出され，問題を高速に解く試みがなされてきた．一方，近年高性能なベクトル演算プロセッサチップが開発されるようになり，並列計算機の各ノードに付加されて数値演算を支援するアーキテクチャも現れてきた[3]．このようなハードウエアにおいては，DPCの仮想プロセッサの特徴を利用して，並列化に加えベクトル化によってさらにデータパラレルモデルを高速に実行できる可能性がある．

本稿では，DPC コンパイラを並列化とベクトル化の両方を行なえるコンパイラに拡張し，DPCプログラムからベクトル化コードを複雑な依存解析なしに生成できること，さらに，前述のアーキテクチャに相当する，数値演算アクセラレータオプション(NCA)を付加した高並列計算機AP1000[5]のためのコードを実際に生成して，演算性能向上の予測を行なった結果を示す．DPCのこのような拡張により，

- 並列プログラムを記述する立場から見た場合，DPCによるデータパラレルプログラムはそれが並列処理されるかベクトル処理されるかを規定しないので，プログラムを変えることなく並列化重視，またはベクトル化重視のコンパイルストラテジを選択することができる．
- コンパイラを作る立場から見た場合，DPCはもともと並列性を内在した記述であるので，複雑な依存解析なしに容易にベクトル化コードを生成するコンパイラに拡張可能である．

2 並列言語 Dataparallel-C

2.1 仮想プロセッサ

DPC では，多数の仮想プロセッサを考え，これらに並列に処理すべきデータをマッピングして計算を行なう．domainによる宣言で仮想プロセッサの定義やそのデータ構造を決める．[domain *name*] ブロック内では，各仮想プロセッサの local view でプログラムを記述する．DPC で DAXPY を記述すると，list1 のようになる．この例では，1024個の仮想プロセッサを宣言して，各プロセッサが2個の`double`の変数をもっている．`[domain vector]` ブロックの中で，これら各仮想プロセッサが自身の変数についての計算を行なうように記述する．結果的に，長さ1024のdoubleの配列の各要素でDAXPYが並列に計算されることになる．

[domain *name*] ブロックの外の変数は各仮想プロセッサにglobal な変数となる．分散メモリマシンの場合，各物理プロセッサがコピーをもち，この変数への代入は broadcast の通信ライブラリ呼び出しとなる．他のdomainの変数は，陽に仮想プロセッサのindexを用いたり「隣の」プロセッサを指定するマクロなどによりアクセス可能で，適当な通信ライブラリの呼び出しが生成される．

2.2 通信ライブラリ

DPCは，DPCソースプログラムから通信ライブラリの呼び出しを含むCのソースプログラムにコンパイルする．それを各並列計算機のCコンパイラを用いて，その計算機に合わせて作成された通信ライブラリとリンクし実行モジュールを作成する(図1)．マシンの違いや通信方式の違いはこの通信ライブラリのインターフェース内で吸収される．

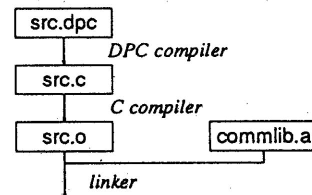

図 1: コンパイル手順

DPCソースのコンパイル時に，実際にプログラムを実行させる時の物理プロセッサ数を指定する．list1 で，物理プロセッサ数を16としてコンパイルすると，1プロセッサあたり64仮想プロセッサを受け持つlist2 が生成される．

3 ベクトル化コード生成

3.1 方針

通常物理プロセッサよりも多く宣言された仮想プロセッサは，list2 のように構造体の配列をループで逐次的に処理するようなコードにコンパイルされ

List 1 daxpy.dpc: DPCによるDAXPY

```
1   #define N 1024
2   domain vector {
3       double x, y;
4   } v[N];
5
6   void DPC_main() {
7       double a;
8       ...
9       [domain vector].{
10          y = a * x + y;
11      }
12  }
```

List 2 daxpy.c: daxpy.dpc からコンパイルされたCソース

```
1   ...
2   struct vector {
3       double x, y;
4   } v[64];
5
6   void DPC_main() {
7       int DPC_vpi;
8       double a;
9       ...
10      for (DPC_vpi = 0; DPC_vpi < 64; DPC_vpi++) {
11          v[DPC_vpi].y = a * v[DPC_vpi].x + v[DPC_vpi].y;
12      }
13  }
```

List 3 neighb.dpc: 通信を含む例

```
1   #define N 1024
2   domain vector {
3       double x, y;
4   } v[N];
5
6   void DPC_main() {
7       double a;
8       ...
9       [domain vector].{
10          x = a * x;
11          y = y + successor()->x;
12      }
13  }
```

List 4 neighb.c: neighb.dpcからコンパイルされたCソース

```
1   ...
2   struct vector {
3       double x, y;
4   } v[64];
5
6   void DPC_main() {
7       int DPC_vpi;
8       double a;
9       ...
10      for (DPC_vpi = 0; DPC_vpi < 64; DPC_vpi++) {
11          v[DPC_vpi].x = a * v[DPC_vpi].x;
12      }
13      ... [ call communication library ] ...
14
15      for (DPC_vpi = 0; DPC_vpi < 64; DPC_vpi++) {
16          v[DPC_vpi].y = v[DPC_vpi].y + DPC_temp_1;
17      }
18  }
```

る．このループのイテレーション間ではデータの依存関係はない．依存関係がある，すなわち仮想プロセッサ間でのデータの参照，更新がある場合には通信ライブラリのコードが挿入され，ループは分割されて生成される．例えばlist3は，successor()->マクロによってひとつ隣の仮想プロセッサの変数の値を参照しているが，これをコンパイルした結果はlist4のようになり通信ライブラリの前後で二つのループが生成される．

このように仮想プロセッサループは常にイテレーション間に依存関係がない．このため，この部分の処理はベクトル処理可能で，対応したベクトルコードを生成することができる．domainブロック内の仮想プロセッサの一つの変数を単純に一つのベクトル変数とみなすことで，非常に明解にベクトル化を行なうことができるのである．変数の依存解析などの複雑な手順をもちこむ必要がないという優位性があるのはこのような理由による．

DPCコンパイラ内部の処理は，DPCソースを構文解析してtreeの内部表現を作り，domainブロック内は通信ライブラリの呼び出しや仮想プロセッサループなどを含むtreeに変換し，最後にこのtreeを再帰的にトラバースしてCのプログラムテキストを書き出す(図2).

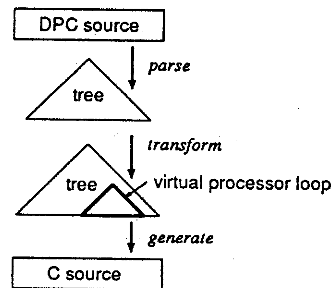

図 2: DPCコンパイラ内部の処理

変換されたtreeにおいて仮想プロセッサループを表現する部分のsub treeを抽出するのは容易であり，ベクトル化コード生成のための処理はこのsub treeがCソースに書き出される部分を拡張するだけで可能である．ベクトル化コード対応の書き出し関数を新たに作成して付加し，仮想プロセッサループのsub treeが現れたらこの関数に渡してやれば良い．以降は，この関数の動作を述べている．

3.2 前処理

現段階では，ひとまとまりのベクトル演算命令のコードを，文単位で生成することとする．仮想プロセッサループ内で，文毎にベクトル化可能かどうかを調べてtree のノードに印をつけておく．ベクトル化不可能な文は，関数呼び出しを含む式，for，ifを除く制御構造，などである．これらは，あとで再び文単位に仮想プロセッサ分のループで囲まれる．

3.3 式

domain内のスカラ変数をベクトル変数とみなしてベクトルプロセッサのレジスタに乗せて式を処理する．参照される変数はベクトルレジスタを割り当ててロードする．domain変数のベクトルレジスタへのロードは，domainが実際には struct の配列になっていることを考慮して，この struct のサイズをストライドとして行なえば良い(図3)．代入

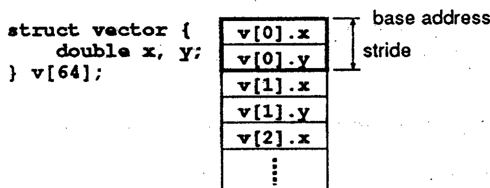

図 3: ベクトル化対象の変数

に対応してベクトルレジスタをメモリにストアする．四則演算はベクトルレジスタ間の演算命令を生成する．list1 のDAXPYは，list5 のようなソースにコンパイルされる(Listで示しているのは疑似コード)．

List 5 daxpy.dpc のベクトル化コード(疑似コード)

```
1    ...
2    struct vector {
3        double x, y;
4    } v[64];
5
6    void DPC_main() {
7        int DPC_vpi;
8        double a;
9        ...
10       LD      64,              VLEN;
11       LD      a,               S0;
12       LDV     (v[0:63].x),     V0;
13       MULSV   V0,              S0,     V1;
14       LDV     (v[0:63].y),     V2;
15       ADDV    V1,              V2,     V3;
16       STV     V3,              (v[0:63].y);
17       ...
18   }
```

3.4 if文

if文は，条件式の真偽をマスクレジスタに設定して各文をマスクつき演算で行なうことで実現する．if文がネストした場合や else文の処理は，マスクレジスタの演算をおこなって適当なマスク値に再設定しなければならない．具体的な方法は後節で述べる．

3.5 for文

for文の繰り返しが各仮想プロセッサで異なる場合，すなわちループ変数が仮想プロセッサで独立の場合にはループの振舞いが独立になってしまうのでベクトル化に向かない．ループ変数の振舞いが各仮想プロセッサで一致している場合に限り，これを検出してループ内の各文をベクトル化の対象とする．振舞いが同じなので，ループの制御構造はスカラプロセッサ側で実現することができる．

4 AP1000への実装

4.1 AP1000とNCAの構成

このように拡張したDPCを用いて，高並列計算機 AP1000 に数値演算アクセラレータオプション(NCA)を付加したアーキテクチャのためのベクトル化コード生成を実現した．NCAは，各セルに1個のベクトル演算プロセッサ μVP[2] と16MBのSRAMを付加するものである．μVPのベクトル長は可変であるが使用できるレジスタの数も変わってしまうので，レジスタ割り当ての簡略化のため利用できる最小のレジスタ数を仮定してコード生成を行なった．諸元を表1に示す．

Peak performance (50 MHz)	206 MFlops (single) 106 MFlops (double)
Internal registers (double)	8 vectors (length 128) 16 scalars 2 masks (length 128)

表 1: μVP諸元

μVPによる演算は図4のような手順となる．AP1000 のセルであるスカラプロセッサ(SPARC IU)が，SRAM に割り付けられている仮想プロセッサの変数やμVPのスカラレジスタを初期化する．その後IUがμVPに起動コマンドを与えると，μVPはSRAMに置かれている自身のための命令コード列を取り込み，ベクトル演算を IU とは独立に実行し始める．SRAMのベクトル処理対象のデータをload，演算，storeし，一連の命令列の実

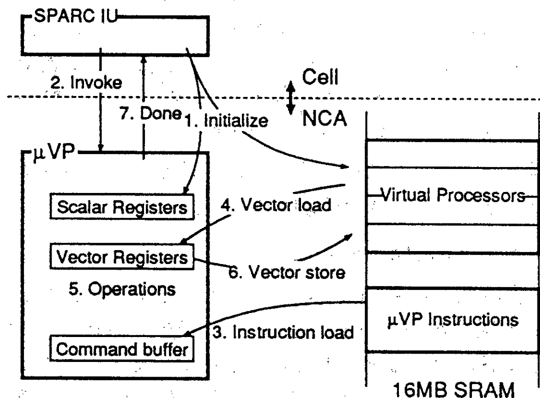

図 4: μVPによる演算の手順

行を終了するとステータスを変化させることでIUに通知する．現段階のDPCのベクトル化においては，μVPの実行中，IU は単にステータスをチェックし終了を待つようにしている．

4.2 コンパイル手順

ベクトル化コードは，DPCが生成するCソースの中に特定のパターンを含んだコメントとして埋め込まれて生成される．このCソースは μVPを起動する手続きを含む，セルのための通常のCプログラムとなっており，同じソースをフィルタプログラムで処理すると μVPのためのアセンブラソースが分離される．μVPアセンブラは命令コードを定数データで表現したSPARC のアセンブラで処理可能な形式にアセンブルするので，これら全てをAP1000のCコンパイラで処理すればセルの実行モジュールが出来上がる(図5).

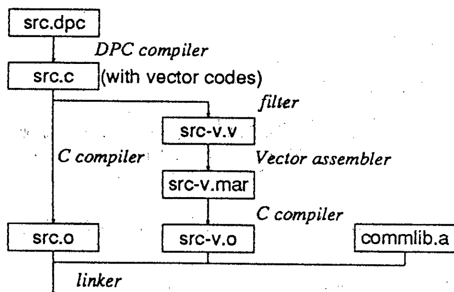

図 5: コンパイル手順

4.3 if文の実装

μVPは表1のとおりベクタ長128ではマスクレジスタの数が2個しかないので，ネストしたif文に対して対処が必要となる．ここでは，マスクデータをメモリにスタック的に退避する方法で実現した．マスクレジスタは，最初はマスクなしに初期化しておき，通常の全てのベクトル演算をマスクつき演算にしておく．if文が現れたら，条件式からマスクデータを生成しマスクレジスタに設定してブロック内の演算に入る(図6). if文がネストした場合，条

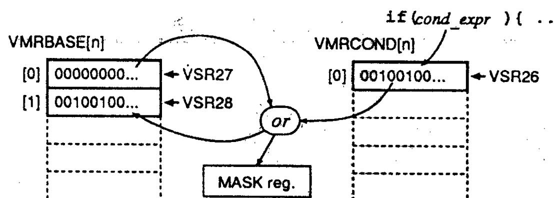

図 6: if の処理

件式とカレントのマスクデータを退避して，新たなマスクデータとの論理和(論理積ではない．1で偽だから)をカレントのマスクとして以降を実行する(図7). else文は，直前の条件式のマスクパターンを

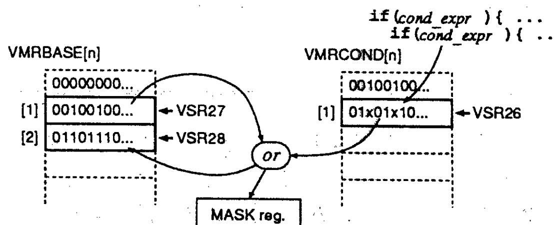

図 7: if のネスト処理

論理反転したものを直前のカレントのマスクデータと論理和をとることで実現する(図8).

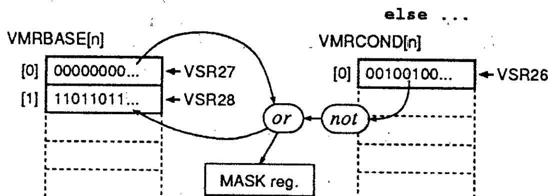

図 8: else の処理

4.4 μVPによるメモリコピー

DPC では，一つの物理プロセッサ内の仮想プロセッサ間のデータ転送のために多くのメモリコピーの手続きを呼び出す．μVPの load/store 命令を用いることでSRAM上のデータのメモリコピーを高速に行なうことができる．

5 性能見積り

5.1 行列乗算のDPCプログラム

このようなベクトル化コード生成によって実際にどのようなコードが生成され，どの程度の演算速度が得られるかを見積もってみる．例として，行列乗算のプログラムをとりあげる(list6)．正方行列とし，行数の仮想プロセッサを定義して行毎に並列に演算することを示している(図9)．$C = A \times B$ で，A，C は行毎，B は列毎に各仮想プロセッサがデータを保持する．list6 の21行目で，B の列データを一斉に隣へ代入してサイクリックに演算を進めていくアルゴリズムとなっている．

List 6 matmul.dpc: DPCによる行列乗算

```
1   #define N 512
2   domain matrix {
3       double      a_col[N];
4       double      b_row[N];
5       double      c_col[N];
6       double      buf[N];
7       double      tmp;
8   } x[N];
9
10  matrix_mult() {
11      int i, j;
12
13      [domain matrix].{
14          buf = b_row;
15          for (i = 0; i < N; i++) {
16              tmp = 0.0;
17              for (j = 0; j < N; j++) {
18                  tmp += a_col[j]*buf[j];
19              }
20              c_col[(i+ID)%N] = tmp;
21              predecessor()->buf = buf;
22          }
23      }
24  }
```

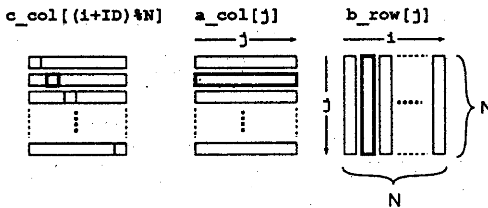

図 9: matmul.dpc のデータ構造

5.2 ベクトル化コードの例

DPCが生成したベクトル化コードをlist7 に示す．行列乗算の計算時間のほとんどを占めるlist6の18行目の積和がベクトル命令になっていることがわかる．list7 の49行目は，プロセッサIDを知るための関数呼び出しを含む式なので，現時点ではベクトル化せずループで処理している[1]．

IU は，ベクトル処理対象のデータのアドレスをスカラレジスタにloadし，VPURUN(*addr*)関数で *addr* からのベクトル命令を起動する．この関数はμVPの実行終了を待って制御が戻る．よって，現在はスカラプロセッサのフェーズとベクトルプロセッサのフェーズが分離している．

5.3 演算時間の見積り

現時点ではAP1000の μVPを使用するソフトウエア環境が整備されておらず，実際にμVPを起動して評価できるまでの実装に至っていない．そこでμVPが占めると思われる演算時間をあらかじめ求めておき，その部分を実際に演算は行なわずに対応した時間だけ消費するスカラプロセッサの空ループ関数に置き換えてAP1000を動作させる方法で，ベクトル化コードの演算性能の見積りを行なった．

512×512 の行列の乗算を2回繰り返した演算について，この方法で見積もったベクトル化コードの演算実行時間(全体)，およびその中での通信を行なっている部分の時間を図10に示す．また，同じプログラムをAP1000 の FPU で計算した場合の演算時間，通信部分の時間も同様に示した．横軸は物理プロセッサ数である．

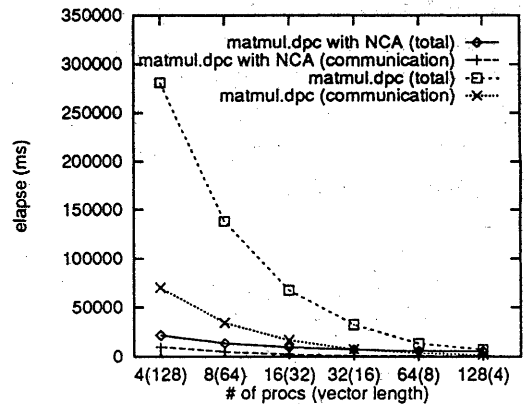

図 10: matmul.dpc の性能見積り (elapse)

問題のサイズが一定なので仮想プロセッサ数は一定である．従って，ベクトル長は物理プロセッサ数が多くなると短くなる．このように，DPCによるコンパイル時に指定する物理プロセッサ数によって，スカラプロセッサによる並列性を重視するか，

[1] プロセッサIDを知ることに関しては関数を使わない方法で実現できる見通しが立っている．

List 7 matmul.dpcから生成されたベクトル化コード

```
1    struct matrix {
2        double a_col[512];
3        double b_row[512];
4        double c_col[512];
5        double buf[512];
6        double tmp;
7    };
8    struct matrix x[64];
9
10   int matrix_mult()
11   {
12      int i;
13      int j;
14
15     /* [domain matrix]. */
16      { ...
17
18      for (; i < 512; )
19        { ...
20
21        for (; j < 512; )
22          {
23          VPUIO(UVSR(30)) = sizeof(struct matrix)/4;        /* stride of 32 bit data member */
24          VPUIO(UVSR(31)) = sizeof(struct matrix)/8;        /* stride of 64 bit data member */
25
26   #if 0  /* --- vectorized expresson (  79) ------------ */
27          x[DPC_vpi].tmp += x[DPC_vpi].a_col[j] * x[DPC_vpi].buf[j];
28   #endif /* ----------------------------------------- */
29
30          DPC_vpi=0;
31          VPUIO(UVLEN) = DPC_num_vp_matrix;        /* vector length (loop limit) */
32   /* vcode:      STARTLABEL(vc01);                */
33          VPUIO(UVSR(4)) = (unsigned int) &(x[DPC_vpi].a_col[j]);
34   /* vcode:      VLD64( VSR(31), VSR(4), VR(0) );                  */
35          VPUIO(UVSR(5)) = (unsigned int) &(x[DPC_vpi].buf[j]);
36   /* vcode:      VLD64( VSR(31), VSR(5), VR(8) );                  */
37   /* vcode:      VMULDM( VR(8), VR(0), VR(8), VMR(0) );            */
38          VPUIO(UVSR(6)) = (unsigned int) &(x[DPC_vpi].tmp);
39   /* vcode:      VLD64( VSR(31), VSR(6), VR(16) );                 */
40   /* vcode:      VADDDM( VR(8), VR(16), VR(8), VMR(0) );           */
41   /* vcode:      VST64( VR(8), VSR(31), VSR(6) );                  */
42   /* vcode:      VSTOP();                */
43          {extern vc01(); VPURUN(vc01);}
44
45          }
46
47          /* Can't use VPU code, hummm...  Y. Koyanagi */
48        for (DPC_vpi = 0; DPC_vpi < DPC_num_vp_matrix; DPC_vpi++) {
49          x[DPC_vpi].c_col[(i + (DPC_get_index_matrix(DPC_nodenum, DPC_vpi) - 0)) % 512]
50            = x[DPC_vpi].tmp;
51        }
52
53          ... [ call communication library ] ...
54        }
55      }
56   }
```

ベクトル性を重視するかを選択できることになる．図10において，ベクトル重視の領域ではベクトルプロセッサの効果がよく現れていて演算性能が大きく改善される．グラフの通信部分の時間の中には，同一物理プロセッサ内の仮想プロセッサ間のメモリコピーの時間も含まれているが，μVPによるメモリコピーでこの部分も大きく高速化されている．

図11はこの結果をMFlops値および1セルあたりのMFlops値で示したものである． ベクトル長が短くなるに従ってセルあたりのMFlops値は低下している．ベクトル長が長い時は，ベクトル化によって10倍以上の演算性能向上が見込めるが，μVPのピーク性能 100 MFlopsと較べるとその利用効率はまだとても低く，ベクトル化コードを生成できない部分をできるだけ減らしていく改善が必要である．

6 最適化・拡張

6.1 strip mining

ベクトル処理のベクタ長は一つの物理プロセッサが担当する仮想プロセッサ数であり，動作させるベクトルプロセッサのベクタ長より長い場合が有り得る．現在はこれについてはまだ対処していないが，仮想プロセッサループ内ではベクタ長が変化しないので，これに対処するための strip mining は生成したベクトルコード全体を繰り返すように実行させることで容易に実現できる．

6.2 VPU命令生成の最適化

ベクトル処理対象の式に対するベクトルコードの生成は，現段階では再帰的にtreeをトラバースして命令列を書き出すだけの単純なものである．よって，ベクトルレジスタの割り付け，冗長なベクトル

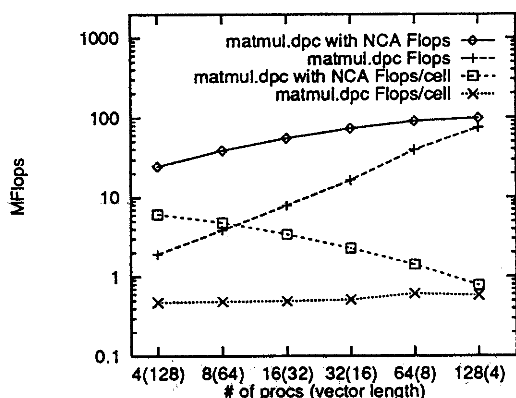

図 11: matmul.dpc の性能見積り (MFlops, MFlops/cell)

ロード命令の削除，命令順序の入れ換え，演算強度の軽減といった，深いデータ依存解析を必要としない覗き穴最適化を施す余地がまだ多く残されており，複雑な式に対して効果が期待できる．

6.3 ベクトルプロセッサとスカラプロセッサのオーバーラップ動作

AP1000 の NCA のようなアーキテクチャの場合，スカラプロセッサとベクトルプロセッサは独立して動作できる．このとき動作がオーバーラップするようなコードを生成することでさらに演算効率を上げることができる．処理しているベクトルデータが次にどの時点で使われるかを判断(または指示)できれば，ベクトルプロセッサの終了を待つポイントを遅延することでスカラプロセッサが同時に別の処理を行なうことができる．

6.4 演算と通信のオーバーラップ動作

さらにDPC一般の拡張として，演算と通信のオーバーラップが考えられる．DPC が生成するコードは演算部分と通信ライブラリの呼び出しとに分離されている．この場合も，通信されるデータがどの時点で使われるかを判断できれば，通信ライブラリを通信開始と終了に分離し終了のポイントを遅延することでオーバーラップが可能である．

6.5 並列化・ベクトル化ストラテジの自動判定

DPCのコンパイル時に物理プロセッサ数を指定することで並列化・ベクトル化の重みを選択できることを述べたが，これをコンパイラ自身が決定できるようにする拡張が考えられる．仮想プロセッサ数，プログラムの挙動，通信の割合，実行マシンの特性などをもとに判定したいが，実現には多くの検討すべき点があると思われる．

7 まとめ

本稿では，仮想プロセッサの概念をもつDPCの性質を利用してベクトル化コードを簡潔に生成することができ，かつ実際のベクトル演算プロセッサを持つ並列計算機で効率的に演算が行なえるコードを生成するコンパイラに拡張できることを示した．関連研究として，CM-5 のための CM-Fortran, C* がベクトル化を含むコード生成を進めている[4]．配列に対する演算を，スカラプロセッサのループでなくベクトル演算プロセッサを起動して処理する点は共通している．CM-Fortran と C* を共通の中間形式に変換し，段階的にベクトル化や最適化を行なっている．

今後は，実際にNCAを動作させて早急に定量的な評価を行なうと共に，前節で述べた拡張をさらに順次検討していく．

謝辞

日頃御指導，御助言いただく，並列処理研究センター石井センター長，白石担当部長，池坂主任研究員，佐藤主任研究員，ならびに並列処理研究センターの同僚諸氏に感謝いたします．

参考文献

[1] Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers.* The MIT Press, 1991.

[2] Hideyuki Iino et al. A 289MFLOPS single-chip supercomputer. In *IEEE 39th International Solid State Cuircuits Conference*, 1992.

[3] John Palmer and Guy L. Steele Jr. Connection machine model CM-5 system overview. In *Frontiers of Massively Parallel Computation*, pp. 474 – 483. Thinking Machines Corporation, IEEE Computer Society and NASA, October 1992.

[4] Gray Sabot. A compiler for a massivelly parallel distributed memory MIMD computer. In *Frontiers of Massively Parallel Computation*, pp. 12 – 20. Thinking Machines Corporation, IEEE Computer Society and NASA, October 1992.

[5] 石畑宏明, 稲野聡, 堀江健志, 清水俊幸, 池坂守夫. 高並列計算機AP1000のアーキテクチャ. 電子情報通信学会論文誌 *D-I*, Vol. J 75-D-I, No. 8, pp. 637–645, 1992.