行列積における並列処理性能の評価

中村孝*、吉田正廣*、山崎裕之*

Performance Evaluation of Matrix Multiplication with Parallelized Programs

by
Takashi Nakamura*, Masahiro Yoshida*, Hiroyuki Yamazaki *

ABSTRACT

We show the performance evaluation of matrix multiplication with parallelized programs.
Matrix multiplication is one of the basic operations in scientific computations.
We parallelized programs by two languages of data parallel and message passing.
We also parallelized by two approaches. One is not to change original programming style and the other is to improve parallel overhead caused by data transfer among processors.

1．はじめに

航技研「数値シミュレータIIシステム」において、基本的で応用上有用なプログラムである行列積を対象に、並列化と性能測定を行った。

数値シミュレータIIシステムの計算エンジンである「数値風洞（NWT）」は今年のはじめに増強され、合計１６６台の要素計算機（PE）となり、理論ピーク性能は２８０GFLOPS、主記憶容量４５GBとなった（図１参照）。また今年度初頭のジョブ処理件数のうち、実に４０％以上のジョブが並列ジョブであった。このことは、数値風洞が航技研CFDにおける有用性を示している。

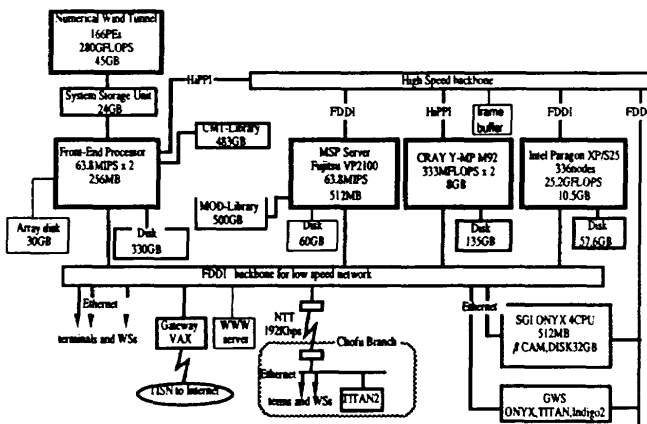

図１．　数値シミュレータIIシステム構成図

行列積プログラムは、プログラム行数が短く演算が少ないので、並列化の理解を得やすくかつ解析が容易である。ここではまずベクトル化の性能、次に種々の並列化を行い性能について検討する。この例ではNWTの１PEのユーザ使用メモリは２００MBとし（ハードウエア上は２５６MB×１６２台、１GB×４台で、ユーザが最大使用可能なメモリは２１０MBおよび９６０MB程度）、行列は正方で、分割は全ての行列で同一分割とした。それはプログラムを見る上で並列化の性能評価の上でも理解を得やすくするためである。

並列化はNWT－FORTRANおよびPARMACSを用いて行った。NWT－FORTRANはデータパラレルアプローチによる並列化で、従来の逐次型処理プログラムに並列化ディレクティブを挿入することにより、配列、DOループの分割等で並列実行を行う。この形式で並列化されたプログラムはそのまま通常のコンパイラで逐次処理が可能である。これに対し、PARMACSはメッセージパッシングアプローチの並列化ライブラリで、PVMやMPIなどと並んで、超並列計算機やワークステーションクラスタなどに広く利用されている。この並列化はPE毎にローカルに処理し、対象全体のどの部分を処理しているかは常にPE番号と共にプログラム中で把握している必要があり、送信／受信／同期ライブラリを呼ぶことによりデータ通信を行い、並列実行を行う。

行列の分割方法は、まず１次元分割で行い、性能を検討しながら種々のチューニングを施した。また、ここでの並列化は、ライブラリ作成などで行われている高度なチューニングまでは行わない。

2．ベクトル性能

並列化を行う前にベクトル長、すなわち行列サイズ（N）による単一PEの性能を求める。行列積を

$$C = A \times B$$

とすると、オリジナルプログラムは図２の様になる。図では現在ベクトル計算機では普通に行われているプログラム形式、すなわちベクトル性能が発揮されるように最内側がベクトル化可能なループのプログラムになっている。DO　１３０のループは演算数が少ないため、DO　１２０のループについて２倍に展開されている（ループアンローリング）。こうすることによりベクトル命令が連続で実行され、高い性能が発揮される。図２において、S２、V２と表示されている。

* 航空宇宙技術研究所数理解析部 National Aerospace Laboratory Computational Sciences Divisjion

This document is provided by JAXA.

```
 isn      include
00001      C MATRIX MULTIPLY
00002      C VECTOR PRODUCT
00003      C
00004            PARAMETER (N=2048)
00005      C
00006            PARAMETER (NI=N,NJ=N,NK=N)
00007            REAL*8 A(NI,NJ),B(NJ,NK),C(NI,NK)
00008            REAL*8 WT1,WT2,WT3,WT4
     .....
00035            CALL GETTOD (WT2)
00036   s        DO 100 K=1,NK
00037   s2       DO 120 J=1,NJ
00038   v2       DO 130 I=1,ni
00039   v2       C(I,K)=A(I,J)*B(J,K)+C(I,K)
00040   v2   130 CONTINUE
00041   s2   120 CONTINUE
00042   s    100 CONTINUE
00043            CALL GETTOD (WT3)
```

図２．　オリジナルプログラム

図３に示すようにＮが２０４８を越えた時点で性能が低下するが、これはＮＷＴの最長ベクトルレジスタ長が倍精度で２０４８語（×８本、６４語×２５６本）であるためで、それを越える場合は、剰余を別処理する必要があり、性能低下が見られる。この例のプログラムの場合はＮが２０４８で最高性能になる。行列積のこのプログラムではベクトル命令はＬ＋Ｍ＋Ａ（Ｌ：ロード、Ｍ：乗算、Ａ：加算）であり、この３つの命令は同時に実行可能であり、ほぼ理論ピーク性能（１．６８ＧＦＬＯＰＳ）となる。

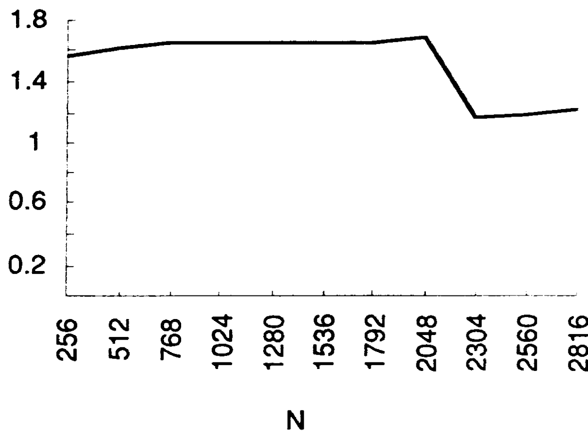

図３．　行列サイズの変化による単体性能

３．並列化

並列化は、行列の分割配置、ＤＯループの分割実行、データ転送および同期に大きく分けられる。

（１）初歩的な並列化

ベクトル性能を維持することを考えて図４に示すように、行列を列方向に等間隔で分割する。またＤＯループはこれにあわせて最外側のＫのループで並列実行する。図５で、第１カラムから！ＸＯＣＬで書き始めている行が並列化の指示行（ディレクティブ）である。また図５で並列化のために挿入した行を網掛で示している。６、１７行は使用するPE台数と型、１２、１６、１９行はA＿G をグローバル宣言し、ローカル配列Aと結合させる。１８行は分割するインデックスの範囲を指定し、２０行は配列Ａ，Ｂ，Ｃのローカル分割配置宣言を示し、３０行で指定された台数の並列実行が開始される。この行が来る前までは１台で実行している。５６行から７３行で囲まれたＤＯループが分割実行される。また、６０行から６４行までがデータ転送を表し、６５行が転送完了待合せを示す。８０行で並列実行が終了し、以降１台のＰＥで実行される。

この方法で並列化された場合、行列Ａは全てのプロセッサから参照されるので、計算実行前にワーク配列に転送しておく。

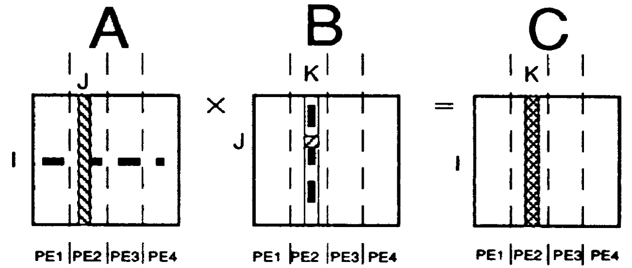

図４．　列方向等分割による並列化

```
 isn      include
00001      C MATRIX MULTIPLY
00002      C VECTOR PRODUCT
00003      C PARALLELIZED ROWWISE
00004      C
00005            parameter (n=2048)
00006            PARAMETER (NPE=2)
00007            PARAMETER (NI=N,NJ=N,NK=N)
00008            REAL*8 A(NI,NJ),B(NJ,NK),C(NI,NK)
00009      C
00010      C PARALLEL
00011      C
00012            REAL*8 A_G(NI,NJ)
00013            REAL*8 WK(NI)
00014            REAL*8 WT0,WT1,WT2,WT3,WT4,WT5,
00015          &        WT6,WT7
00016            EQUIVALENCE (A,A_G)
00017      !XOCL PROCESSOR PEG(NPE)
00018      !XOCL INDEX PARTITION IDXK=(PEG,INDEX=1:NK)
00019      !XOCL GLOBAL A_G
00020      !XOCL LOCAL A(:,/IDXJ),B(:,/IDXK),C(:,/IDXK)
00021      C
     .....
00030      !XOCL PARALLEL REGION
     .....
00055            CALL GETTOD (WT2)
00056      !XOCL SPREAD DO /IDXK
00057   s        DO 100 K=1,NK
00058   s        DO 120 J=1,NJ
00059   s        CALL GETTOD (WT5)
00060      !XOCL SPREAD MOVE
00061            DO 111 I=1,NI
00062              WK(I)=A_G(I,J)
00063        111 CONTINUE
00064      !XOCL END SPREAD (ID)
00065      !XOCL MOVEWAIT (ID)
00066   s        CALL GETTOD (WT6)
00067   m        WT7=WT7+(WT6-WT5)
00068   v        DO 130 I=1,NI
00069   v        C(I,K)=WK(I)*B(J,K)+C(I,K)
00070   v    130 CONTINUE
00071   v    120 CONTINUE
00072   s    100 CONTINUE
00073      !XOCL END SPREAD
00074            CALL GETTOD (WT3)
     .....
00080      !XOCL END PARALLEL
```

図５．　並列化プログラム

This document is provided by JAXA.

図５からわかるように、ＤＯ１２０のループがアンローリングされないためベクトル性能が低下しているのが予測される。また、図６に示すように、全てのＰＥは同時にＰＥ１上のＡの要素を必要とし、アクセスの衝突（競合）が発生している。

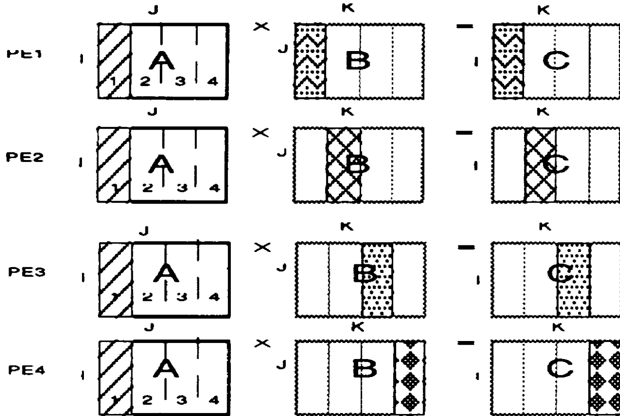

図６．　並列化（同時刻同一プロセッサ参照）

（２）チューニング１、衝突防止

以上の並列化では、同時に多数のプロセッサから同一のプロセッサに割り付けられた行列Ａにアクセスが集中し、衝突が起こる。衝突を防ぐのは図７に示すように各ＰＥの開始する位置をずらすことにより行える。そのプログラムを図８に示す。（図６、７の行列Ａの中の数字は時刻を示す。各プロセッサは同時刻に同一番号の部分を参照している。）

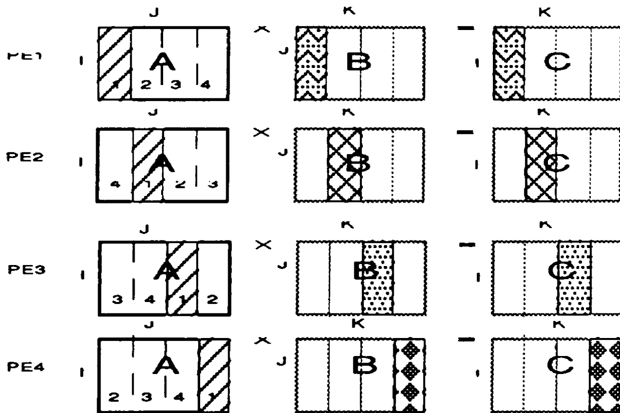

図７．　衝突を防ぐデータ参照

```
  isn        include
00001        C MATRIX MULTIPLY
00002        C VECTOR PRODUCT
00003        C PARALLELIZED ROWWISE
00004        C TUNING 1 FOR MOVE
00005        C  READ PACKET
00006        C
00007              parameter (n=2048)
00008              PARAMETER (NPE=2)
00009              PARAMETER (NI=N,NJ=N,NK=N)
00010              REAL*8 A(NI,NJ),B(NJ,NK),C(NI,NK)
00011        C
00012        C PARALLEL
00013        C
00014              REAL*8 A_G(NI,NJ)
00015              REAL*8 WK(NI)
00016              REAL*8 WT0,WT1,WT2,WT3,WT4,WT5,WT6,WT7
00017              EQUIVALENCE (A,A_G)
00018        !XOCL PROCESSOR PEG(NPE)
00019        !XOCL INDEX PARTITION IDXK=(PEG,INDEX=1:NK)
00020        !XOCL GLOBAL A_G
00021        !XOCL LOCAL A(:,/IDXJ),B(:,/IDXK),C(:,/IDXK)
00022          C
                .....
00067  s           m=idvproc()
00068  s           CALL GETTOD (WT2)
0006         !XOCL SPREAD DO /IDXK
00070  s           DO 100 K=1,NK
00071  m           DO 120 JJj=0,NJ-1
00072  v           jj=jjj+(m-1)*nj/npe
00073  m           j=mod(jj,nj)+1
00074  s           CALL GETTOD (WT5)
00075        !XOCL SPREAD MOVE
00076              DO 111 I=1,NI
00077              WK(I)=A_G(I,J)
00078          111 CONTINUE
00079        !XOCL END SPREAD (ID)
00080        !XOCL MOVEWAIT (ID)
00081  s           CALL GETTOD (WT6)
00082  m           WT7=WT7+(WT6-WT5)
00083  v           DO 130 I=1,NI
00084  v           C(I,K)=WK(I)*B(J,K)+C(I,K)
00085  v       130 CONTINUE
00086  v       120 CONTINUE
00087  s       100 CONTINUE
00088        !XOCL END SPREAD
00089  s           CALL GETTOD (WT3)
```

図８．　並列化プログラム（競合防止）

３）チューニング２、ＷＲＩＴＥパケット

ＮＷＴ－ＦＯＲＴＲＡＮではデータ転送は変数（配列）の代入文で表す（図８、７５行から８０まで）。それをコンパイラが読み取り、転送パケットを生成する。この場合、代入文の右辺または左辺にグローバルとローカル変数がそれぞれ現れるが、グローバルを右辺にローカルを左辺に書くとＲＥＡＤパケットに、ローカルを右辺にグローバルを左辺に書くとＷＲＩＴＥパケットとなる。データは常に右辺から左辺に流れる。ここで、ＲＥＡＤパケットは相手方ＰＥに自ＰＥが必要なデータ番地を教え、転送を依頼する。これに対し、ＷＲＩＴＥパケットは必要としているＰＥにデータを送り込むだけである。ＲＥＡＤパケットの場合は相手の動作に依存するのに対し、ＷＲＩＴＥは相手の受け取り口がふさがっていなければ何時でも送り出せるが、ネットワーク内のデータを押し出すためのダミーパケットの転送時間がオーバヘッドとなる。図９の左にプログラム、右に動作を示す。ｒｅｑｕｅｓｔと書かれているのはＳＰＲＥＡＤ　ＭＯＶＥが実行された時点を示し、斜め点線矢印でパケットの流を示す。ｗａｉｔで転送が完了する。ＲＥＡＤパケットはネットワークを往復するのに対し、ＷＲＩＴＥパケットは一方通行である。

This document is provided by JAXA.

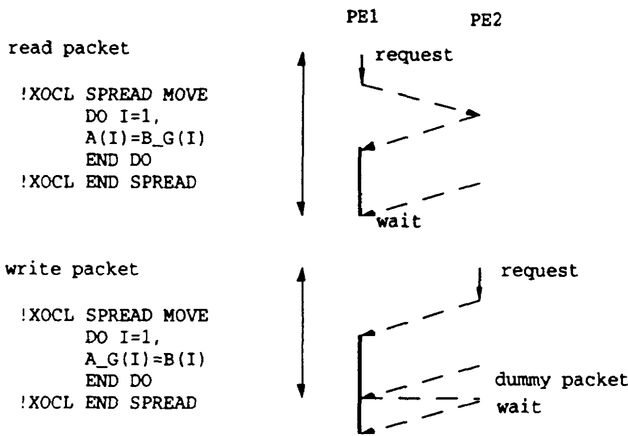

図９．　ＲＥＡＤ／ＷＲＩＴＥパケットの違い

これまでのチューニングにより図１０に示すような並列の効果が実現される。

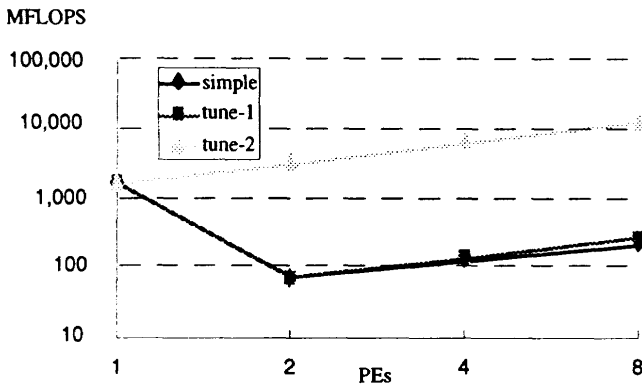

図１０．　並列化による性能変化（N＝２０４８）

４）チューニング３、ブロック転送

転送時間をT_m、パケット作成時間をT_p、転送速度をR_mとすると、データ長Lバイトのパケットを一つ転送するのに要する時間T_mは

$$T_m = T_p + L / R_m$$

で表される。２）では行列の大きさをN、ＰＥ台数をN_{pe}とすると、L＝Nのサイズのパケットで$N/N_{pe} * (N_{pe} - 1)$回の転送を必要とする。また転送するデータ量は$N^2/N_{pe} * (N_{pe} - 1)$である。このチューニングでは衝突は防止されたものの、転送回数（転送パケット数）は１）と同数である。NWTではT_pが５０マイクロ秒程度必要であり、N＝２０４８、N_{pe}＝６４を例にとると、総T_pは約１００ミリ秒であり、データ転送時間は約８０ミリ秒である。そこで、（N，N/N_{pe}）の大きさの作業用配列を用意し、ＰＥ毎に分担して格納している配列Ａをまとめて転送することにより、転送パケット数をN_{pe}－１回（N_{pe}＝６４の場合約３ミリ秒）に減らすことが可能である。このようにブロック化しても全体の転送量は不変である。このプログラムを図１１に示す。

この場合は、プロセッサ台数に応じて総T_p時間は線型に増加するが、データの総転送時間は行列サイズにほぼ依存し、プロセッサ数に対してそれほど増加しない。また演算時間はプロセッサ数に比例して減少する。従って、演算時間がN_{pe}と共に減少するのに対し、転送時間は$N_{pe} * (T_p + L / R_m)$となりN_{pe}と共に増加し、ある点で演算時間より転送時間が多くなり、ＰＥ台数を増やしても実行時間の短縮とはならず、逆に時間が増加する。

```
00015             REAL*8 WK(NI,NJ/Npe)
00080             CALL GETTOD (WT2)
00081       CXOCL SPREAD DO /IDXL
00082       C     DO 121 M=1,NPE
00083             M=IDVPROC()
00084
00085       !xocl barrier
00086
00087             CALL GETTOD(WT8)
00088   m         DO 123 L=1,NPE
00089   m         LL=MOD(L+M-2,NPE)
00090   v         NJMOD=LL*NJ/NPE
00091   v         LM=MOD(NPE-L+M,NPE)+1
00092   v         LMOFF=(LM-1)*NJ/NPE
00093   s         MMOFF=(M-1)*NJ/NPE
00094   s         CALL GETTOD (WT5)
00095       !XOCL SPREAD MOVE
00096             DO 120 J=1,NJ/NPE
00097             DO 111 I=1,NI
00098             WK_G(I,J+LMOFF)=A(I,J+MMOFF)
00099         111 CONTINUE
00100         120 CONTINUE
00101       !XOCL END SPREAD (ID)
00102       !XOCL MOVEWAIT (ID)
00103   s         CALL GETTOD (WT6)
00104   m         WTM(L)=(WT6-WT5)-(WT1-WT0)
00105   v         WT7=WT7+WTM(L)
00106   s         K1=NK/NPE*(M-1)
00107   s         CALL GETTOD(WT9)
00108   s         DO 100 KK=1,NK/NPE
00109             K=KK+K1
00110   s2        DO 122 J=1,NJ/NPE
00111    2        JJ=J+LL*NJ/NPE
00112   v2        DO 130 I=1,NI
00113   v2        C(I,K)=WK(I,J+MMOFF)*B(JJ,K)+C(I,K)
00114   v2    130 CONTINUE
00115   s2    122 CONTINUE
00116   s     100 CONTINUE
00117   v     123 CONTINUE
00118         121 CONTINUE
00119       CXOCL END SPREAD

00120             CALL GETTOD (WT3)
```

図１１．　ブロック転送

図１１で８１行目と８２行目のＳＰＲＥＡＤ　ＤＯ文とＤＯ　１２１の文がコメントアウトされているが、このプログラムでは分割実行をＳＰＲＥＡＤ　ＤＯを用いずに８３行目の関数で自分のＰＥ番号を取得し、これを用いて並列化を行う。こうすることにより、ＰＥ間で精度よく同期を取ることができ、前に述べた転送時の衝突が回避される。

４．　台数効果

チューニング３によるプログラムを用いて１２８台までの台数に対し、各々に行列サイズNを変化させた場合の性能を図１２に示す。図１２には前に述べたように、PE台数を増加させても転送が支配的になると台数増加に反して実行時間が増加し、台数効果が現れない。それに対し、ＰＥ台数と共に問題規模を増大させる場合、スケーラビリティ効果は十分発揮されている。

This document is provided by JAXA.

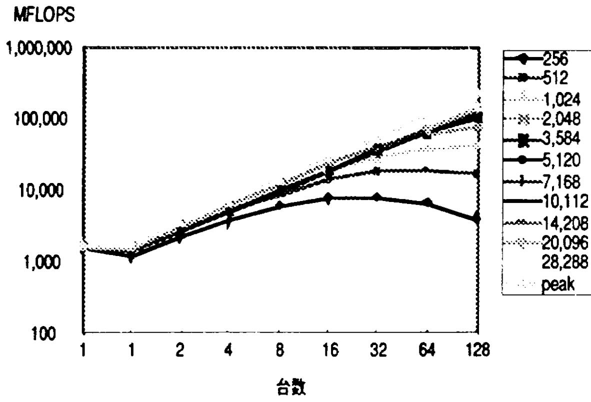

図１２．　台数効果

5．　並列オーバヘッドとさらなるチューニング

並列化の実行時オーバヘッドは大まかに、
・転送時間（転送衝突を含む）
・ロードインバランス（同期時間を含む）
・ＣＰＵ負荷、データ移動（パケット生成など）
・ワーク領域などのメモリ消費
・本質的な逐次処理部分（並列化不可部分）
・並列化によるベクトル性能低下
が考えられる。この問題では、ロードインバランスはなく本質的な逐次処理部もない。ここではさらに転送時間の減少を試みる。転送時間減少には、
・衝突の防止
・高速パケットの使用
・パケットをまとめる
を行ったが、さらに
・非同期転送
・データ転送量の減少（アルゴリズムの変更）
が考えられる。これらのチューニングではワーク領域の増加やプログラムが複雑になるなどのオーバヘッドがある。

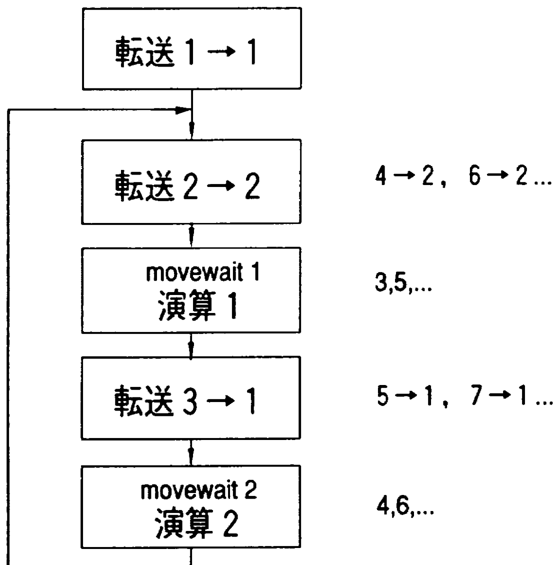

図１３．　非同期転送チャート

１）チューニング４、非同期転送

図７から、明らかなように、あらかじめＡの１の部分を転送し、その終了を待って２の部分を転送しながら１の部分の演算をする。次にこの転送をまって３を転送しながら２の部分を実行する。この場合、１の部分の演算中に２の転送が終了している場合に非同期効果が最大となる。これを繰り返すことにより、転送時間を演算時間の中に隠ぺいすることが可能になり、転送時間を減少させることができる。このフローチャートを図１３に示す。

２）チューニング５、２次元分割

さらに転送データ量を減らすために、２次元分割を考える。図１４の上部に示すように分割することにより、転送データ量そのものおよびパケット数を減少させることが可能である。図１４には１６台のＰＥを使用した場合の１次元分割と２次元分割の違いを示している。１次元分割の場合は行列Ａのみの転送で、データパケット数が$N_{pe}-1$なのに対し、２次元分割の場合はＡおよびＢともに転送する必要があるが、パケットの全体数は$2*(\sqrt{(N_{pe})}-1)$であり、１次元分割に比べて少ない。またデータ量は１次元分割の場合が、$N*N/N_{pe}(N_{pe}-1)$であるのに対し、２次元分割の場合は、$2*N*N/N_{pe}*(\sqrt{N_{pe}}-1)$ですむ。$N=2048$, $N_{pe}=16$の場合、１次元分割では２０４８＊２０４８／１６＊１５、２次元分割では２＊２０４８／４＊２０４８／４＊３である（１５対６）。

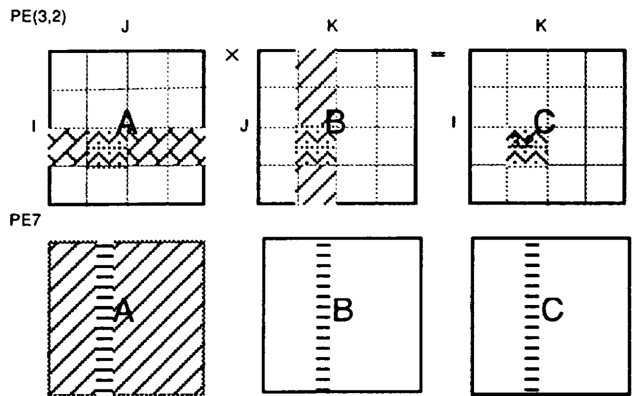

図１４．　２次元分割（１６台の場合）

これらのチューニングにより性能を計測しプロットしたのが図１５である。図１５にはＰＥ台数を６４台とした場合のＲＥＡＤ／ＷＲＩＴＥパケットによる違い、非同期転送、２次元分割の比較を示す。図１６に台数効果を示す。２次元分割はベクトル長が短くなるものの転送が支配的である６４台ではかなり高い性能を示している。

This document is provided by JAXA.

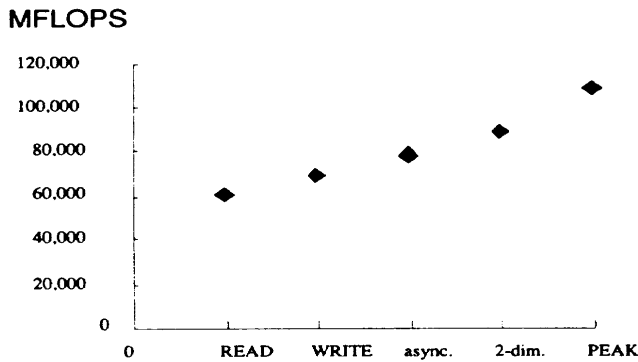

図15. 転送方法による性能（64台の場合）

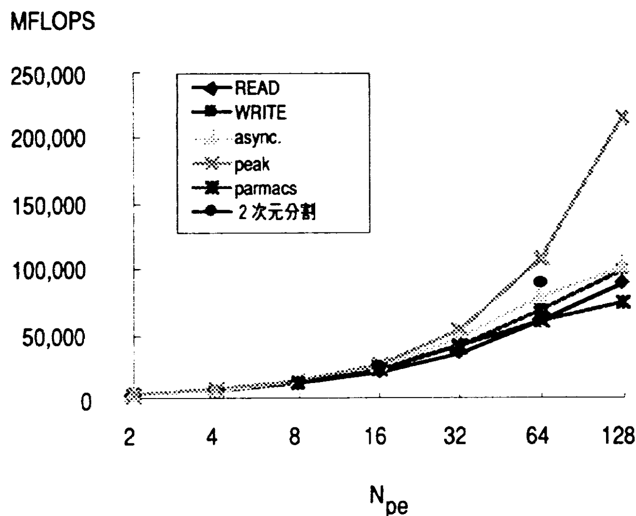

図16. 転送方法と台数効果

6. おわりに

簡単で理解しやすい問題を例に挙げて並列化および性能評価を行った。初歩的な並列化では性能が発揮されない場合、種々のチューニングを考えたが、それほど複雑な並列化を施さなくてもかなりの性能を達成することが示された。まとめると、

1）ベクトル性能をあげる

連続アクセス、ベクトル長の増大、ループアンローリングがされるように書くなどＰＥ本来の性能を引き出す。

2）並列オーバヘッドを考慮する

a）転送時の衝突を防止する

特定のPEへのアクセスをずらす、同期を取って衝突が起こらないようにすることが大事である。

b）高速パケットの使用（可能なかぎりＷＲＩＴＥパケットの使用）

c）転送を少なくする

ブロック化などを行い転送の回数を減らす。ただし、これはメモリオーバヘッドを招くので十分なメモリ量が必要である。

d）転送時間の隠ぺい

非同期転送を行うことにより転送時間そのものを演算時間の中に隠ぺいすることができる。ただし、プログラム複雑化やメモリオーバヘッドを招く。あるいは本質的に非同期転送が行えない場合もある。

e）分割方法を変更する

２次元分割や３次元分割を考慮する。ただし、ベクトル長の減少やプログラムの複雑化を招くことがある。

f）アルゴリズムを変える

アルゴリズムを変えることにより転送を少なくすることが可能である場合があるが、一般的には収束性の悪化を招くこともある。ただし、行列積の場合は収束性は関係がない。

今後は他の有用かつシンプルなアプリケーションによる性能評価、例えばＮＡＳＡ　Ａｍｅｓ研究所のベンチマークテストであるＮＰＢ等の評価、また他の並列化による評価、例えばメッセージパッシングライブラリによる並列化、さらに別の計算機による評価、例えばＰＡＲＡＧＯＮ等を行う予定である。

また、システム全体としての評価、入出力性能、並列実行開始／終了、ＰＥダイナミックアロケーションなどの評価をジョブ単位、システム単位に行うことも予定している。

最後に、並列計算機で性能評価を行う場合には、時間計測ツールや他のパフォーマンス計測ツールをどのように用いれば望むデータが取得できるのか、取得したデータで全てのＰＥの動作を評価可能か否か、などの吟味が必要である。また相対評価する場合は、何を基準とするか、その基準となるものも評価する必要がある。

This document is provided by JAXA.