JAERI-Data/Code
96-009

疎結合スカラ並列計算機のグローバル総和の高速化

1996年3月

大田敏郎*

日本原子力研究所
Japan Atomic Energy Research Institute

本レポートは，日本原子力研究所が不定期に公刊している研究報告書です。
入手の問合わせは，日本原子力研究所技術情報部情報資料課（〒319-11 茨城県那珂郡東海村）あて，お申し越しください。なお，このほかに財団法人原子力弘済会資料センター（〒319-11 茨城県那珂郡東海村日本原子力研究所内）で複写による実費頒布をおこなっております。

This report is issued irregularly.
Inquiries about availability of the reports should be addressed to Information Division, Department of Technical Information, Japan Atomic Energy Research Institute, Tokai-mura, Naka-gun, Ibaraki-ken 319-11, Japan.

編集兼発行　日本原子力研究所
印　　刷　㈱原子力資料サービス

JAERI－Data/Code 96－009

疎結合スカラ並列計算機のグローバル総和の高速化

日本原子力研究所計算科学技術推進センター
大田　敏郎*

（1996 年 2 月 8 日受理）

Intel Paragon XP/Sにおける global sum 機能に同等で，簡便でより高速なアルゴリズムを適用したサブルーチンを開発した。

その結果，128ノード使用の場合，約10倍のパフォーマンスを得た。結果と共に本アルゴリズムの特徴，実行条件，拡張性などについて報告する。

日本原子力研究所：〒153　東京都目黒区中目黒2-2-54
＊ 業務協力員　㈱ＣＲＣ総合研究所

JAERI－Data/Code 96－009

A Fast Global Sum on the Coarce-grained Scalar Parallel Computer

Toshio OHTA*

Center for Promotion of Computational Science and Engineering
Japan Atomic Energy Research Institute
Nakameguro, Meguro-ku, Tokyo

(Received February 8, 1996)

A new global sum subroutine, which has the same function as the prepared one on the Intel Paragon, is developed. The algorithm is simple and faster.

It makes the performance 10 times faster than the original one in case of 128 nodes. The results will be shown with the characteristics, restrictions and extendability.

Keywords: Paragon, Global Sum, Parallel, Message Passing

* on loan to CRC Research Institute, Inc.

目　　次

Contents

1. はじめに

近年、単一プロセッサの処理速度向上が限界に近づいたことから、複数の CPU（中央演算処理装置）を並列に使用して処理速度の向上を図る並列計算機の普及が広がりつつある。

並列計算機におけるひとつの処理単位をノードとすると、ノード間の接続形態で大きく分けて２つに分類することができる。ひとつはノード間で主記憶を共有する共有メモリ型計算機、もうひとつはノードごとに独立した主記憶を持つ分散メモリ型計算機である。日本原子力研究所が現有する富士通 VPP500、Intel Paragon XP/S はいずれも分散メモリ型の並列計算機である。

分散メモリ型アーキテクチャを採用した並列計算機で、しばしば多用され、ほとんどのシステムで標準に用意されている通信機能の一つに global sum 機能がある。Intel Paragon XP/S（以下、Paragon と呼ぶ）でもシステム標準のメッセージパッシングライブラリである NX ライブラリに gdsum(), gssum(), gisum()（それぞれ倍精度用、単精度用、整数変数用のサブルーチン）が用意され、利用可能である。

しかしながら、NX ライブラリに用意されている global sum は使用ノード数を増やすと、特に総和を求める配列のサイズが大きい場合に著しく性能が低下し、多用するとスケーラビリティの低下を招く。そこで、新規により高速な global sum 機能を持つサブルーチンを作成することにした。また、global sum のような基本的なルーチンについてアルゴリズムを検討するとより広範にその成果を応用することができ、開発することの意義は大きい。

まず、第 2章では、高速版の global sum を作成するにあたってノード間のメッセージ転送命令の基本性能を知る必要があるので各種のメッセージパッシングの基本性能を測定し、得られた結果に従って 2 次元メッシュ形状のネットワークに適したアルゴリズムを考案した。

第 3章および第 4章では、第 2章で述べたアルゴリズムに従ってサブルーチンを作成し、性能の測定、比較を行った。

第 5章では、さらに高速化を行なうにはどのような手法が考えられるかという点についての可能性と、得られた基本的アルゴリズムを他に応用することについて検討した。

第 6章では、得られた結果をまとめた。

なお、具体的なターゲットとしては、科学技術計算で多用される倍精度の配列に対する global sum 命令である gdsum() 互換のインターフェースを持つサブルーチンを作成し、性能評価の対象とした。

2. 基本性能の測定とアルゴリズム

2.1 global sum の簡単な説明

global sum の機能について、簡単に説明すると、Fig.2.1に示したように、各ノードが重複して保有している配列の各要素の値を要素ごとに総和を求め、その結果を全ノードに返す機能である。

global sum では原理上 all-to-all 通信が必ず必要となるので、大規模な配列に対して global sum 命令を実行する際に高いパフォーマンスを得るためには、高速なノード間通信機能と効率のよいアルゴリズムを考えることが必要となる。

また、加算に要する演算時間もベクトル計算機においては総通信時間に比して無視できる範囲であるが、Paragon のようなスカラ計算機では無視できない時間を要する。新規に global sum の機能を作成する場合には以上のようなことに留意して作業を行なう必要がある。

2.2 標準のアルゴリズム

まずシステム標準の gdsum() がどのようなアルゴリズムを用いているかを調べる必要がある。

Intel Paragon で標準として用意されているインタラクティブなパフォーマンス監視ツールである spv を使用し、gdsum() のみを多数回呼び出す簡単なプログラムを実行し、実行状況を観察し、同時に時間計測のシステムコールである dclock() を使用して実行にかかった実時間を測定した。

spv は X-window 上で動作するツールで、Intel Paragon のフロントパネル上に表示される各ノードの使用状況とノード間通信の状況を示す装置を模したグラフィカルなパフォーマンス監視ツールであり、フロントパネルの表示に加えてより詳しい使用状況やネットワークを経由して遠隔地からの使用も可能となっているという特色を備えている。

dclock() はある時刻からの経過時間を倍精度変数に値（単位は秒）を返すシステムコール関数であり、Paragon で実行経過時間を計測する際に最も一般的に使用される。

spv で実行状況を監視した結果、系統だった転送をしているようには見えず、複数通信を特に調停せずに単に一つのノードに各ノードから値を転送し、加算を実行、得られた加算結果を全ノードにブロードキャストしていると判断した。そのため特に総和を求める段階の通信を行う段階でメッセージの衝突あるいは通信バンド幅の飽和などが起こっている可能性が考えられた。

全ての加算処理を１つのノードで行なっていると仮定すると、おおまかに分けて次の２つの手法が考えられる。Fig.2.2, Fig.2.3 に１次元に４ノードが並んだ場合の転送アルゴリズムの説明図を示す。

Fig.2.2の場合、加算処理を行なうノードは受信、加算を繰り返す事になり、実行時間の大半はほとんどのノードが待機状態になるが、加算のためのワーク領域は元の配列と同じ大きさで良い。

Fig.2.3の場合、加算処理を行なうノードの近傍でトラフィックの集中が起こり、メッセージの衝

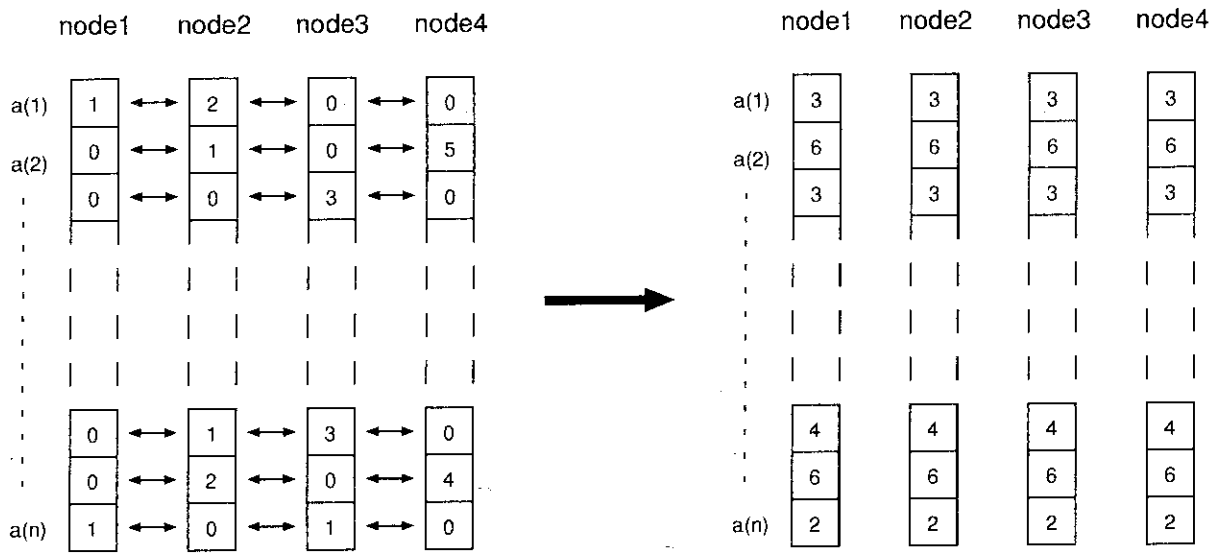

Fig. 2.1 Basic Description of Global Sum

突が起こり、通信バンド幅が飽和する。また、全ての受信が終了した時点で加算を行なうことになるので元の配列のほぼノード数倍分のワーク領域を必要とする。

NX ライブラリの global sum の場合、引き数で自分で陽にもとの配列と同じ大きさのワーク領域を確保する仕様となっており、Fig.2.2に示した方法の転送、加算順序をとっていると判断できる。

なぜこのようなアルゴリズムを採用しているかという理由は幾つか考えられるが、主な理由は汎用性を維持するためと考えられ、実行中のノードの取得、解放への対応、効率の良い転送が行えないノード数への対応、などが考慮されているのではないか。

2.3 基本方針

参考文献 [1] では、Intel Paragon の試験機的な存在であり、同様の２次元メッシュ形状のネットワークトポロジーを採用した Intel Touch Stone Delta で one-to-all, all-to-one , all-to-all の場合についていろいろな通信アルゴリズムの性能を測定し、結果を比較検討している。

その結果によると global sum で発生するような all-to-all の通信には２次元メッシュ状に配置されたノードの各 column あるいは row 内で独立して通信を行う場合にもっともよい結果を示している傾向がある。今回作成した global sum は基本的にはその通信方法で作成した。

単純なアルゴリズムを用いると global sum のみではなく、一般的なアプリケーションの並列化の場合に all-to-all の通信の際、参考にして適用することが可能となる利点も生じる。

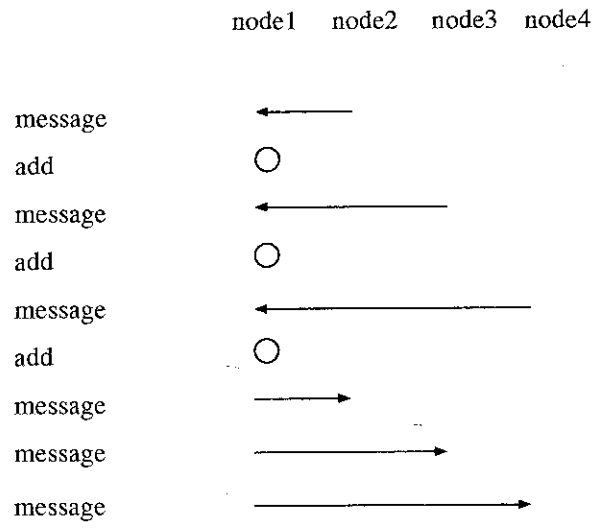

Fig. 2.2 Transfer Algorithm of NX A

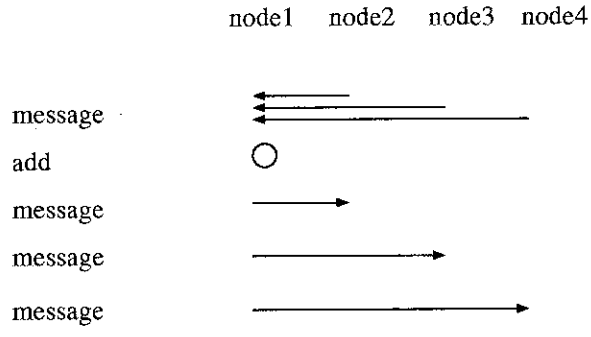

Fig. 2.3 Transfer Algorithm of NX B

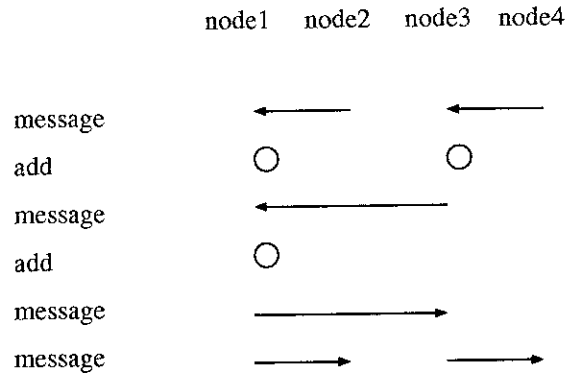

Fig. 2.4 New Transfer Algorithm in 1D

2.4 計算機運用上の条件

現在の日本原子力研究所の Paragon システムの運用法では 256 ノードの Paragon はバッチ専用の運用がなされており、いくつかのパーティションに論理的に分割して運用されている。

その全てのパーティションは (2 のべき乗)x(2 のべき乗) の構成をとっていて、各 column あるいは row 内で通信経路の重なりを生じないように段階的に通信を行う場合において非常に有利である。

この特色を生かすべく、その様な運用方法に特化したアルゴリズムを採用することとした。

2.5 開発方針

単に gdsum() を高速化するのではなく、より汎用性が高い実装方法を取ることにする。つまり、複雑なアルゴリズムを採用するとその手法をエンドユーザが流用するのが困難になりがちだからである。そこで、可能な限りわかりやすくコーディングしやすい手法を取ることとする。

また、計算ノード配置については汎用性を多少犠牲にしても実行状況に制約を加えて高速化を図ることとする。

また、gdsum() のパフォーマンスの悪化の原因が加算を担当するノードへの転送集中であると判断したので転送バンド幅をなるべく生かし、通信経路の重なりを極力起こさないことに留意して開発を行なうこととする。

2.6 アルゴリズム

前節で説明した方針にしたがって Fig.2.4に示されるような配列の転送、加算方法のアルゴリズムを考案した。説明を簡便化するために 1 次元の場合の説明図を Fig.2.4に示す。

図中では例として 4 個のノードを配してあり、順にマスターノードである左端のノードに加算を繰り返しながら転送し、演算の分散を図り、なおかつ通信経路の重なりを避けている。

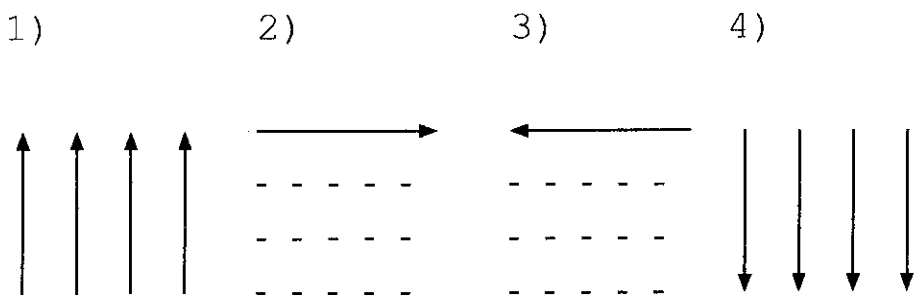

Fig. 2.5 New Transfer Algorithm in 2D

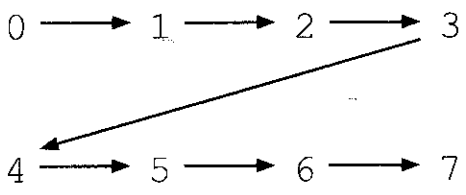

Fig. 2.6 Turn of Nodes

次にこれまでに説明したことを２次元に拡張するためには、まず縦方向に１行目のノードに加算、転送し、次に１行目内だけで左端のマスターノードに加算し、今度は逆順に転送して全てのノードに値をコピーするという手順で実行すればよい。

２次元 4x4, 16 ノード構成の場合の転送方法概念図を Fig.2.5に示す。矢印の方向は転送方向を示し、破線部は転送を行わない行を表している。各列あるいは行内の転送は Fig2.4で説明した方法によっている。

なお、今回は疎結合スカラ並列計算機に適したアルゴリズムについて考察したが、参考までにクロスバー結合ベクトル並列計算機に適したアルゴリズムを付録１に示す。

それぞれのシステムの持つ特色によって同じ機能を実現するのにまったく違ったアプローチが最適なものとなっていることがわかり、興味深い。

2.7 必要な情報

このアルゴリズムを実現するためには全ての転送が１対１で行われるので各ノードが幾何的にどのような配置にあるかを知る必要がある。

まず、矩形に配置されているノード群が何行何列の構成かを知る必要がある。その情報は、NX ライブラリに用意されているシステムコールサブルーチン nx_app_rect() で求めることができる。

次に転送命令を実行する各ノードが、自分は何行目何列めに存在するかを知る必要がある。

各ノードは１行目から行内で順に割り付けられ、その行に全て割り付けられると次の行でまた

順に割り付けられることがわかった。例えば２行４列の８ノードで構成されるパーティションの場合、Fig.2.6に示すような順番で割り付けられる。

以上で前節に述べた効率の良い転送順序を実行する材料は揃ったのでコーディングに移る。なお、これ以降、新規に作成した global sum を便宜上 tgdsum() と呼ぶ。

3. 実装

3.1 開発過程で生じた性能低下

tgdsum() 開発当初は 4 ノードや 8 ノードなどの少ない使用ノード数において gdsum() より遅いという傾向が見られた。原因はノードの配置形状を求めるシステムサブルーチン nx_app_rect() が低速なためである。この命令は global sum 1 回を実行するのに必要な時間に比して非常に長い ms のオーダーの時間を要するため、tgdsum() をコールする度に毎回コールしては致命的な性能低下を招くことになる。

しかしノードの配置は基本的にはプログラム実行中不変なので最初に 1 回だけ実行し、これを各ノードで保持して参照するよう変更して回避することとした。また、nx_app_rect() と同様に必要になる自ノードのノード番号を知るためのシステム関数 mynode() と総使用ノード数を知るためのシステム関数 numnodes() も同様のタイミングでコールして、最終的に必要となる自ノードの存在する行、列についての情報を各ノードで保存するよう変更して、高速化を図った。

3.2 時間計測の際の留意点

一般にプログラム開発の過程で、性能測定を行ないながら作業を進めるにあたって、単一で 1 秒以内程度のごく短い実行単位からより長時間の実行結果を類推する手法はよく用いられるが、並列計算機のプロセス投入の仕組みによっては、不用意にそれを行なうと正しい情報を得られず、チューニングの方向性を誤ってしまうことがあるので注意を要する。

Paragon では投入されたプロセスは各ノードで同期を取らずにスタートする。実行時間を計測したいプログラム単位の先頭で時刻を求めるファンクションコール (Paragon では dclock()) を実行し、最後に再び同様に実行し、その差から実行経過時間を求めるという方法ではノード間で通信が発生する場合にはどこかのノードで本質的には必要の無い通信待ちが生じ、ノードごとに実行経過時間に大幅な差を生じることになる。

これは主としてオブジェクトの主記憶へのロード時間から生じると考えられるが、これを回避してより正確な実行時間計測を行なうには最初に dclock() を実行する直前にグローバル同期命令である gsync() を実行し、各ノードの実行状況を揃えてやるとよい。

ただし、この方法はあくまで小さなプログラム単位の実行時間を正確に測定するために用いるべきであり、実際のアプリケーションプログラムではローディングの時間に比して長時間の実行時間測定で性能評価を行なうべきである。

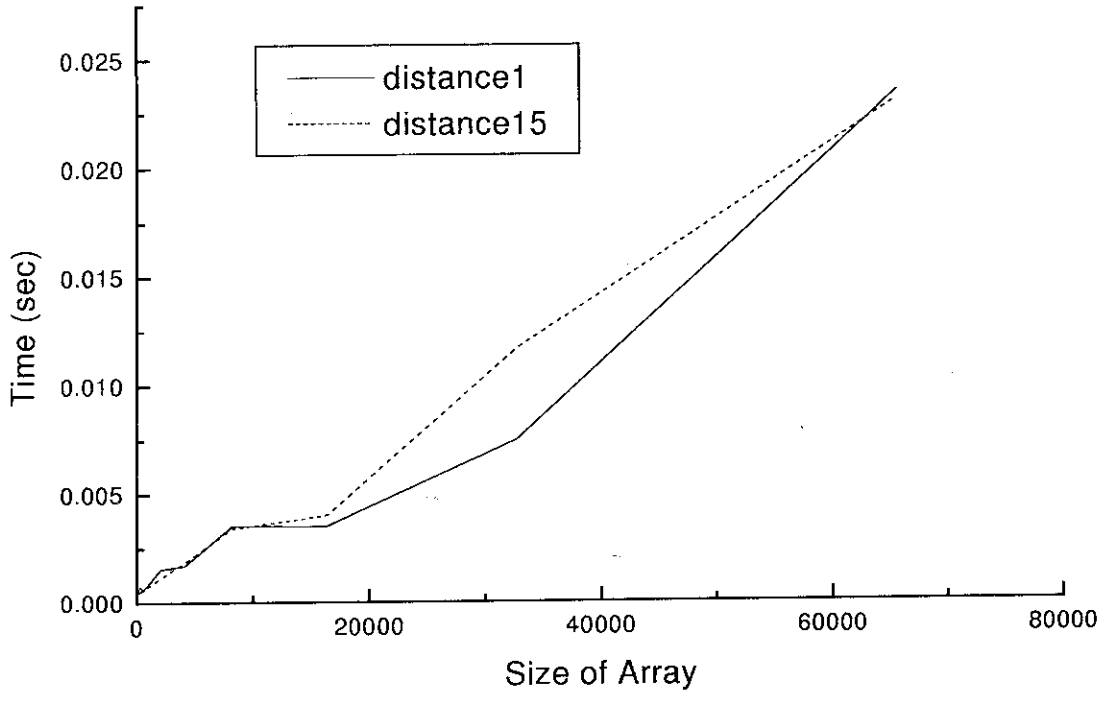

Fig. 3.1 Effects by Distance

3.3 Node 間の距離の影響

ルータでの遅延がどの程度影響が出るかを測定した。隣接したノード間の転送と、15 離れたノード間での転送時間を測定した。結果を Fig.3.1に示す。

結果から、転送にかかる時間はノード間の距離にはあまり関係ないことがわかった。若干線形が乱れているのは他のジョブの影響であると思われる。よってルータでの遅延は考慮する必要はなく、通信経路の重なりを生じさせないことのみ留意すればよい。

3.4 プログラムの説明

プログラムリストを流れに沿って順に以下に説明する。

```
      subroutine tgdsum(tx,n,tt)
c
c     並列計算ライブラリのヘッダファイルのインクルード
      include 'fnx.h'
c
c     配列の宣言
```

```
c     単精度の場合は tx,tt の宣言部を変更
      double precision tx(*)
      integer n
      double precision tt(*)
c
c     ジョブ実行中変化しないパラメータの宣言
      data if1st/1/
      save iamtgd,irowtgd,icoltgd,myrtgd,myctgd
c
c     最初に呼ばれたときだけ nx_app_rect() を実行
      if(if1st.eq.1) then
      iamtgd = mynode()
      istat = nx_app_rect(irowtgd,icoltgd)
c
c     対応できないパーティション構成かどうかをチェック
      if (irowtgd.eq.1.or.icoltgd.eq.1) istat = 1
      if (istat.ne.0) stop 'stop in tgdsum'
c
c     自ノードの存在する行、列を求める
      myrtgd = iamtgd/icoltgd + 1
      myctgd = mod(iamtgd,icoltgd) + 1
c
      if1st = 0
      endif
c
c     倍精度の場合は n*8, 単精度の場合は n*4
      n8 = n*8
c
c     まず各列内で 1 行目のノードに値を集める
      iskip = 1
   10 continue
      iskip2 = iskip*2
      ifsnd = mod(myrtgd-iskip,iskip2)
      ifrcv = mod(myrtgd,iskip2)
c     加算を行わないノードならば配列を送信
      if(ifsnd.eq.1) then
      call csend(2,tx,n8,iamtgd-icoltgd*iskip,0)
c     加算担当ノードならば配列を受信
      elseif(ifrcv.eq.1) then
      call crecv(2,tt,n8)
c     加算実行 low,high の場合は min(),max() に変更
      do ii=1,n
      tx(ii)=tx(ii)+tt(ii)
      enddo
      endif
c
      iskip = iskip*2
c *** repeat 10 until iskip == irowtgd
```

```
      if(iskip.eq.irowtgd) goto 20
      goto 10
   20 continue
c     各列内の転送終了
c
c
c     1行目のノードならば1列目のノードに値を集める
      if (myrtgd.eq.1) then
      iskip = 1
   30 continue
      iskip2 = iskip*2
      ifsnd = mod(myctgd-iskip,iskip2)
      ifrcv = mod(myctgd,iskip2)
      if(ifsnd.eq.1) then
c     加算を行わないノードならば配列を送信
      call csend(4,tx,n8,iamtgd-iskip,0)
c     加算担当ノードならば配列を受信
      elseif(ifrcv.eq.1) then
      call crecv(4,tt,n8)
c     加算実行 low,high の場合は min(),max() に変更
      do ii=1,n
      tx(ii)=tx(ii)+tt(ii)
      enddo
      endif
c
      iskip = iskip*2
c *** repeat 30 until iskip == icoltgd
c     1列目に集まったら終了
      if(iskip.eq.icoltgd) goto 40
      goto 30
   40 continue
c
c     まず1列目内で他のノードに値をコピー
      iskip = icoltgd/2
  110 continue
      iskip2 = iskip*2
      ifsnd = mod(myctgd,iskip2)
      ifrcv = mod(myctgd-iskip,iskip2)
c     対応するノードに配列を送信
      if(ifsnd.eq.1) then
      call csend(6,tx,n8,iamtgd+iskip,0)
c     対応するノードから配列を受信
      elseif(ifrcv.eq.1) then
      call crecv(6,tx,n8)
      endif
      iskip = iskip/2
c *** repeat 110 until iskip == 1
c     1列目内で全てのノードに値がコピーされたら終了
```

```
      if(iskip.eq.0) goto 120
      goto 110
  120 continue
c
      endif
c
c     各列内で他のノードに値をコピー
      iskip = irowtgd/2
  130 continue
      iskip2 = iskip*2
      ifsnd = mod(myrtgd,iskip2)
      ifrcv = mod(myrtgd-iskip,iskip2)
c     対応するノードに配列を送信
      if(ifsnd.eq.1) then
      call csend(8,tx,n8,iamtgd+icoltgd*iskip,0)
      elseif(ifrcv.eq.1) then
c     対応するノードから配列を受信
      call crecv(8,tx,n8)
      endif
c
      iskip = iskip/2
c *** repeat 130 until iskip == 0
      if(iskip.eq.0) goto 140
      goto 130
  140 continue
c     結果のコピー終了
c
c     ここでノード間で同期を取るかは議論の必要あり
c     call gsync()
c ***
      return
      end
```

3.5 使用方法

gdsum() を使って書かれた既存のコードを tgdsum() を使うように書き直すには、新たに tgdsum() のソースリストを加え、

```
CALL GDSUM()
```

となっている行を

```
CALL TGDSUM()
```

と書き換えるだけである。独自のヘッダファイルをインクルードしたり、初期化ルーチンを先頭でコールしたりする必要はない。

gdsum() に対する制限は、ノード の配置が (2のべき乗)x(2のべき乗) になっている必要がある点である。これも、現状では 256 ノードのシステムで実行する限りはどの実行キューにおいてもそうなっているので特別な設定は必要ない。

3.6 性能測定

gdsum(),tgdsum() の両者に対してそれぞれ global sum を実行する倍精度配列のサイズを 1 から 65536 まで段階的に変化させ、実行直前と直後に時間測定システムコールである dclock() をコールした後に差を求め、1 回の実行にかかる実行経過時間を測定した。

なお、改良版は nx_app_rect() のオーバーヘッドを除去するため、計測前に最初に 1 回ダミーでコールしてあるが、これはあくまで 1 回の実行で傾向をとらえるために必要な操作であり、アプリケーションプログラムに tgdsum() を実装する場合には多数回の呼び出しは前提条件となるために問題にならないことを付記しておく。

それを 8 ノード、128 ノードの 2 つの使用ノード数で測定した。結果を次章に示す。

4. 結果

8ノードの場合の結果のグラフをFig.4.1に示す。128ノードの場合の結果のグラフをFig.4.2に示す。縦軸は実行実時間（秒）、横軸は配列のサイズ（倍精度なので8倍するとByte）である。

全体的な傾向として、tgdsum()の方が2倍から10倍程度高速である。

Fig.4.1からわかるように、8ノード で比較すると、gdsum()比で約2倍程度高速になっている。その理由は8ノードの場合はノード数が少ないのでgdsum()の場合に加算を行なうノード付近で生じる加算に伴う通信待ちなどのボトルネックがさほど問題になっていないということが考えられる。

対して、Fig.4.2からわかるように、128ノードで比較した場合、約10倍程度と大幅にtgdsum()の方が高速であり、使用ノード数を増やした場合にgdsum()で深刻なボトルネックが生じていることが推測される。

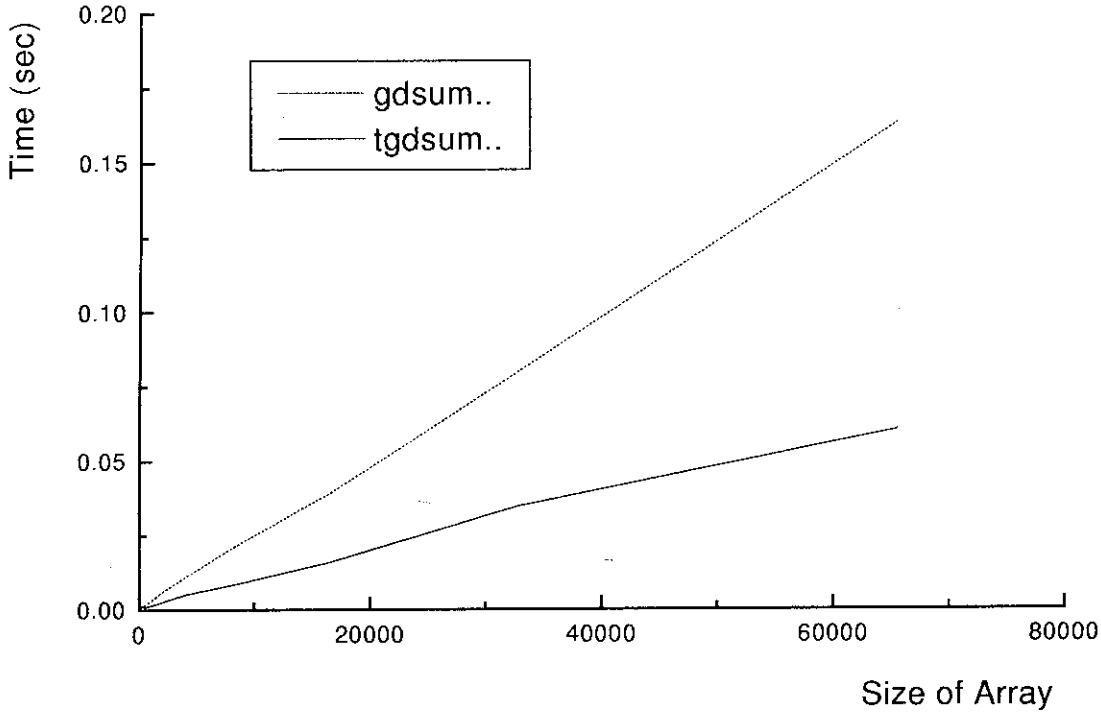

Fig. 4.1 Elapsed Time on 8nodes

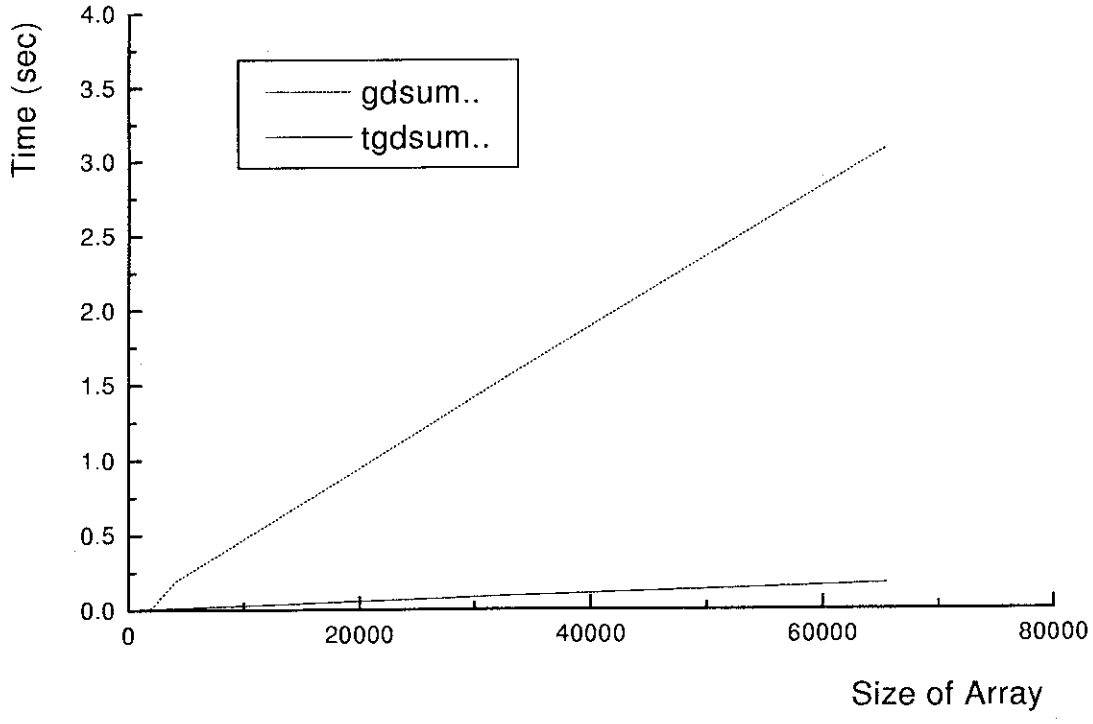

Fig. 4.2 Elapsed Time on 128nodes

5. 考察

今回 tgdsum() を開発するにあたっては、システムの最高性能を引き出すことを狙って開発するという視点ではなく、あくまで汎用的に使用できるようにという見地から、容易に他の状況にも適用できる手法を用いることを優先した。

そのため一部、性能よりわかりやすさを優先したために、アルゴリズム、実際のコード共に随所に高速化のための余地を残している。

ここでは容易に転用できるような手法であることを維持しつつさらに高速化を行なうことができないかという視点で考察することとしたい。

具体的には、転送バンド幅を現状よりさらに有効に使用するためにはどのような工夫が必要かということや、一般的に並列プログラミングにおいて高速化手法として用いられている並列計算時間を通信時間で隠蔽してしまう手法の適用可能性などについて論じることとする。

5.1 相方向転送

現在の tgdsum() では片方向にしか転送していないので、相方向の転送を活用することにより高速化が可能となるか検討してみる必要がある。つまり現時点ではまだまだシステムのカタログ性能を使い切っていないので転送レートは限界のバンド幅に比べて余裕がある。

そこで、現状では全体を転送、加算している配列をいくつかに分割して複数系統で加算、転送を繰り返すように仕様を変更すると、負荷の分散および転送バンド幅の有効活用の可能性が出てくる。

配列を 2 分割して実行する場合について概略を説明すると Fig.5.1のようになる。

このようなアルゴリズムを用いた場合、Fig.5.1中の 1), 4) の段階で相方向転送が利用できて、2), 3) の段階ではそれぞれ独立して加算ができるので演算負荷の分散が図れ、さらなる高速化が図れる。反面、プログラミングは複雑なものになるので容易に転用し難いという短所もある。

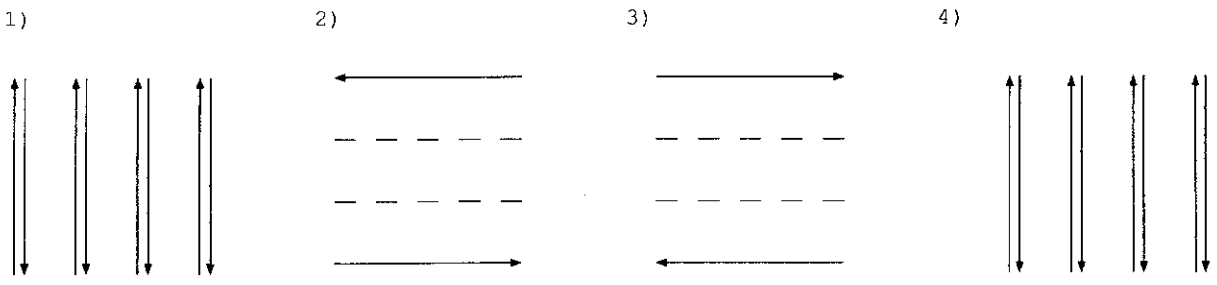

Fig. 5.1 Algorithm of Cross Transfer

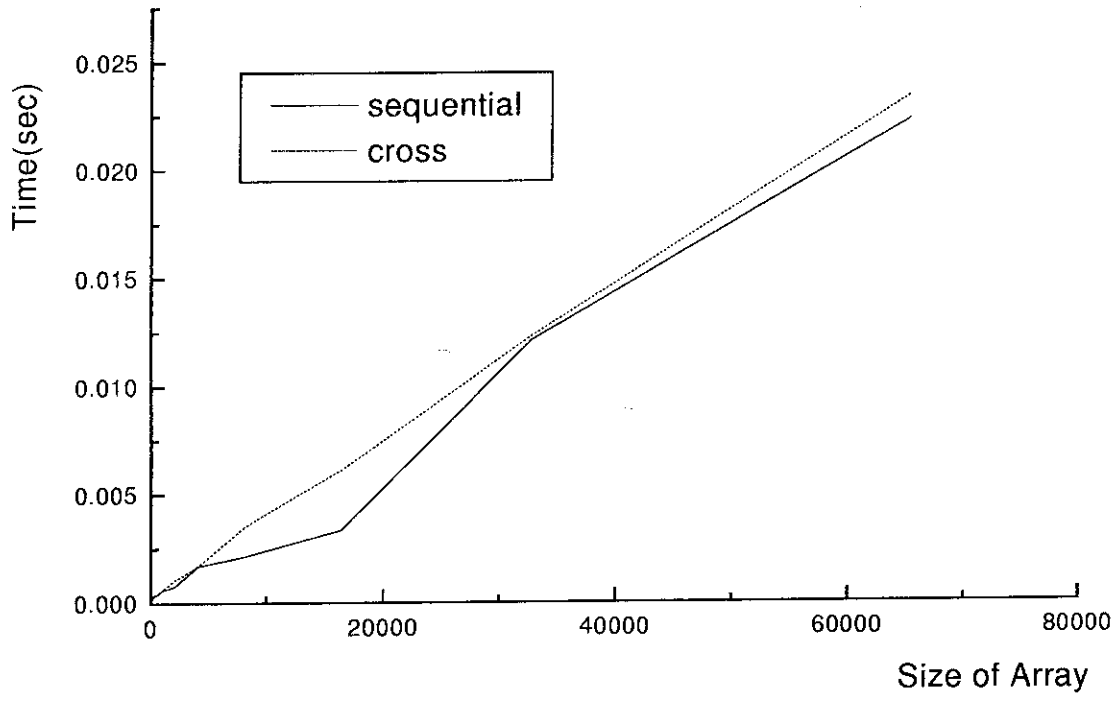

Fig. 5.2 Elapsed Time for Cross Transfer

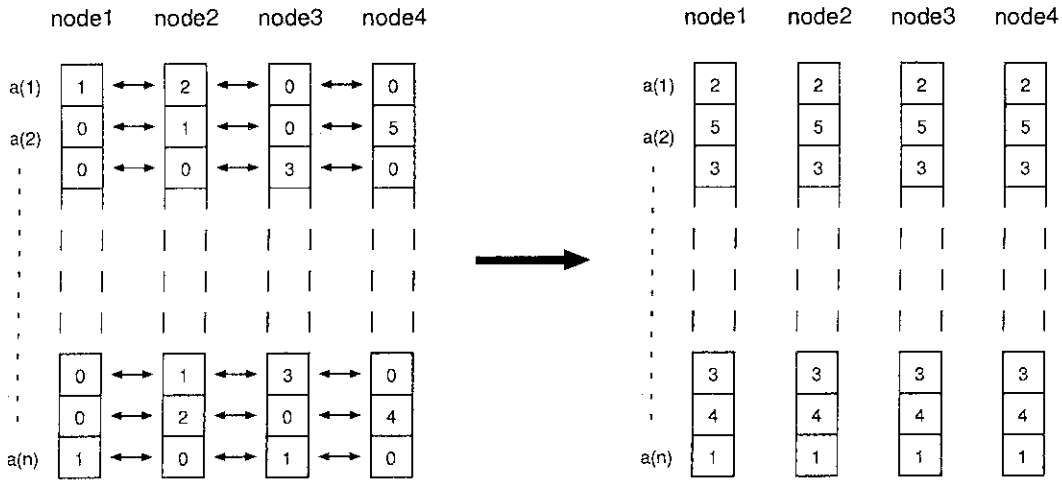

Fig. 5.3 Basic Description of gdhigh()

以上説明したアルゴリズムの適用可能性を探るため、相方向転送の基本性能を測定した。

片方向転送を 2 度繰り返してメッセージを往復させた場合と、相方向転送を実行した場合の実行経過時間を計測、比較した。

Fig.5.2に測定結果のグラフを示す。sequential は片方向、cross は両方向転送を示す。結果をみる限りでは配列サイズ 20000 あたりに相方向転送の方が高速に実行される領域があるが、全体的な傾向はほぼ両者同じ時間を要している。

計算時間の転送による隠蔽などのことを考えると、転送データは小さいほうが有利であり、相方向転送は等の転送量の片方向転送に比して速くはないものの、遅くはなっていないので、工夫次第ではさらなる速度向上が見込める可能性がある。

5.2 他の機能への応用

global sum は配列の要素ごとの合計を全ノードに求める機能であるが、それに似た機能として配列の要素ごとの最大、最小値を求める機能があり、Paragon の NX ライブラリでは倍精度変数の場合は gdhigh(),gdlow() として用意されている。gdhigh() の動作概念図を Fig.5.3に示す。

今回作成した tgdsum() は小変更で gdhigh(),gdlow() 互換のサブルーチンに容易に応用して適用することが可能である。

global sum の場合は、順次配列の同要素同士を加算しながら転送することにより、ひとつのノードに結果の配列を求めることができるが、gdhigh(),ghlow() に関しては、加算を行なう部分を最大値を求める FORTRAN の組込み関数 max(),min() に差し替えることによりほとんど同じコードのままで使用することができる。

また、それぞれの配列定義部を整数変数、あるいは単精度変数に定義しなおすことによって gisum(),gihigh(),gilow(),gssum(),gshigh(),gslow() の高速版を容易に作成することができる。

6. まとめ

今回作成した高速版の global sum である tgdsum() は、標準で用意されている NX ライブラリの gdsum() と比較して、少ない使用ノード数では 2 倍程度、多い使用ノード数では 10 倍程度のパフォーマンス向上を得た。

global sum が実行経過時間の多くを占めるようなアプリケーションプログラムに tgdsum() を適用した場合、大幅なパフォーマンス向上を期待できる。

また、具体的な効率の良い all-to-all 通信アルゴリズムが示されたことにより、その手法を適用し、簡便に効率の良い並列プログラムを開発することができる。

謝　　辞

日本原子力研究所 計算科学技術推進センター 並列処理支援技術開発グループの折居 茂夫氏には多大な助言をいただき、感謝します。

参考文献

[1] Sanne E.Hambrusch, Farooq Hameed, Ashfaq A.Khokhar, Communication operations on coarse-grained mesh architectures, PARALLEL COMPUTING 21 73-751,1995

[2] 川端 英之, 津田 孝夫, 疎結合並列計算機 Paragon の性能評価, ハイパフォーマンスコンピューティング 58-4 19-25,1995

[3] Michael Barnett, David G.Payne, Bobert A. van de Geijn, Jerrell Watts, Broadcasting on Meshes with Worm-Hole Routing, (1993)

[4] Fortran Sytem Calls Reference Manual, Intel Corporation,1995

[5] System Performance Visualization Tool User's Guide, Intel Corporation,1995

6. まとめ

今回作成した高速版の global sum である tgdsum() は、標準で用意されている NX ライブラリの gdsum() と比較して、少ない使用ノード数では 2 倍程度、多い使用ノード数では 10 倍程度のパフォーマンス向上を得た。

global sum が実行経過時間の多くを占めるようなアプリケーションプログラムに tgdsum() を適用した場合、大幅なパフォーマンス向上を期待できる。

また、具体的な効率の良い all-to-all 通信アルゴリズムが示されたことにより、その手法を適用し、簡便に効率の良い並列プログラムを開発することができる。

謝　　辞

日本原子力研究所 計算科学技術推進センター 並列処理支援技術開発グループの折居 茂夫氏には多大な助言をいただき、感謝します。

参考文献

[1] Sanne E.Hambrusch, Farooq Hameed, Ashfaq A.Khokhar, Communication operations on coarse-grained mesh architectures, PARALLEL COMPUTING 21 73-751,1995

[2] 川端 英之, 津田 孝夫, 疎結合並列計算機 Paragon の性能評価, ハイパフォーマンスコンピューティング 58-4 19-25,1995

[3] Michael Barnett, David G.Payne, Bobert A. van de Geijn, Jerrell Watts, Broadcasting on Meshes with Worm-Hole Routing, (1993)

[4] Fortran Sytem Calls Reference Manual, Intel Corporation,1995

[5] System Performance Visualization Tool User's Guide, Intel Corporation,1995

6. まとめ

今回作成した高速版の global sum である tgdsum() は、標準で用意されている NX ライブラリの gdsum() と比較して、少ない使用ノード数では 2 倍程度、多い使用ノード数では 10 倍程度のパフォーマンス向上を得た。

global sum が実行経過時間の多くを占めるようなアプリケーションプログラムに tgdsum() を適用した場合、大幅なパフォーマンス向上を期待できる。

また、具体的な効率の良い all-to-all 通信アルゴリズムが示されたことにより、その手法を適用し、簡便に効率の良い並列プログラムを開発することができる。

謝　　辞

日本原子力研究所 計算科学技術推進センター 並列処理支援技術開発グループの折居 茂夫氏には多大な助言をいただき、感謝します。

参考文献

[1] Sanne E.Hambrusch, Farooq Hameed, Ashfaq A.Khokhar, Communication operations on coarse-grained mesh architectures, PARALLEL COMPUTING 21 73-751,1995

[2] 川端 英之, 津田 孝夫, 疎結合並列計算機 Paragon の性能評価, ハイパフォーマンスコンピューティング 58-4 19-25,1995

[3] Michael Barnett, David G.Payne, Bobert A. van de Geijn, Jerrell Watts, Broadcasting on Meshes with Worm-Hole Routing, (1993)

[4] Fortran Sytem Calls Reference Manual, Intel Corporation,1995

[5] System Performance Visualization Tool User's Guide, Intel Corporation,1995

付録１ クロスバー型ベクトル並列計算機に適したアルゴリズム

Paragon のようなメッシュ型ネットワークトポロジーを持ったマシンと違い、クロスバー型ネットワークを採用したマシンではまた別のアルゴリズムで global sum 機能を高速に実現することができる。

概念的な転送、加算方法を説明すると、クロスバーの場合はメッシュアーキテクチャと違い、１対１通信を同時平行して行なう場合には通信経路の重なりは生じない。ゆえに組み合わせを変えて通信を行なうことにより転送バンド幅を理論値限界まで使い切ることが可能となる。

そこで、まず global sum を行ないたい配列をノードの数で分割して同一要素番号を持つもの同士を各ノードに部分ごとにまとめてまとめて加算を行なうことにより演算速度を上げることができ、ベクトル計算機の場合は転送速度に比して演算時間はほぼ 0 とみなすことができるようになる。この場合、行列の転置を行なうだけなので膨大なワーク領域は必ずしも必要でないこともメリットの一つである。その実行概念図を Fig.1.1に示す。

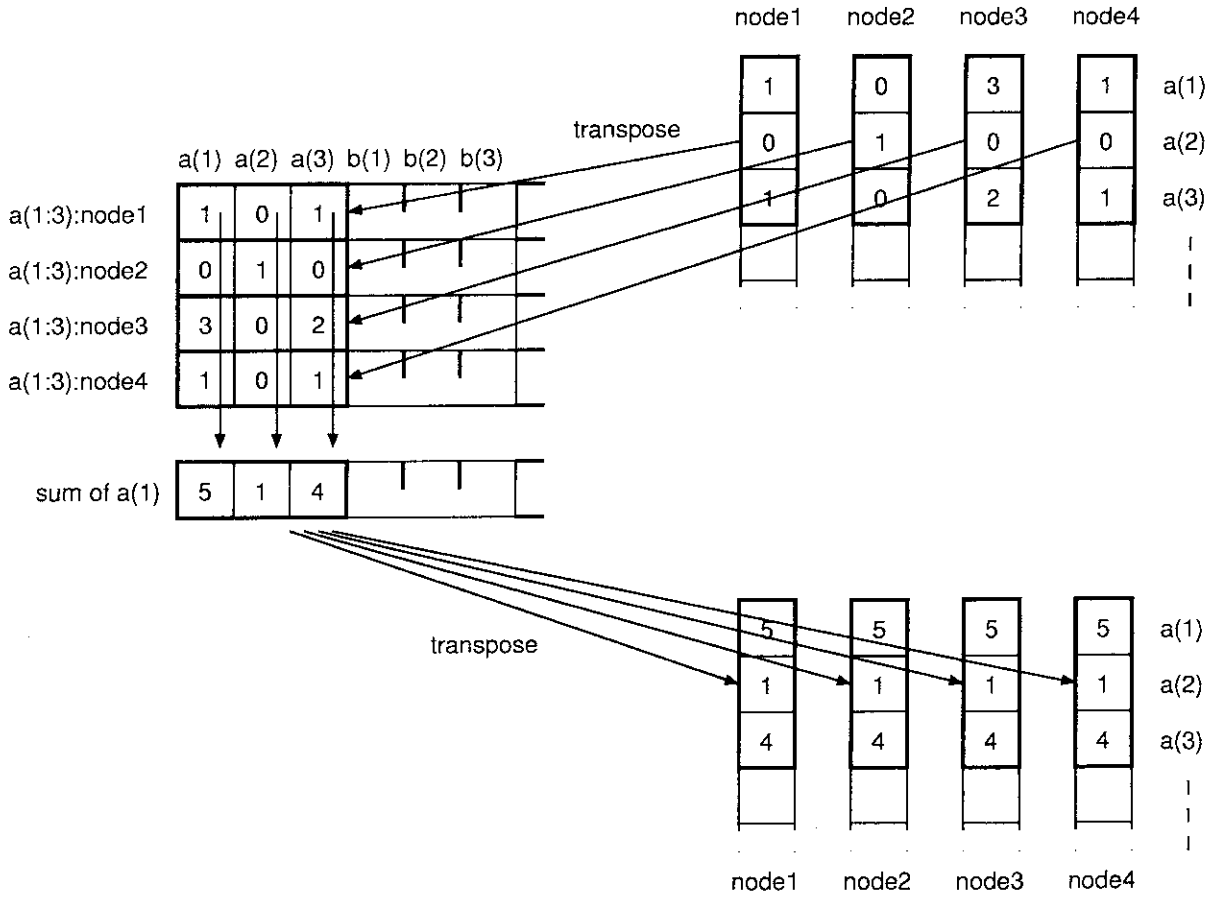

Fig. 1.1 Basic Algorithm on Vector Machine